SOFTWARE BASED CPU EMULATION

Slavomír ŠIMOŇÁK*, Peter JAKUBČO**
*Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, E-mail: Slavomir.Simonak@tuke.sk
**Student of Computer Engineering and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, E-mail: pjakubco@gmail.com

ABSTRACT

Software based emulation of a real hardware is an imitation of its internal design by the software application (model), which behavior is similar to the behavior of the hardware (keeps functionality). Emulators are used with advantage in development (where expensive or old hardware is partially or completely supplied by an emulator), testing and also as a support for machine non-compatible software. Considering a concept of Von-Neumann computer, the central processing unit (CPU) emulation is a core of the computer emulator. Within this paper, theoretical aspects of a CPU emulation are treated and a concrete software solution is also presented.

Keywords: *emulation, CPU, interpretation, static binary translation, dynamic binary translation*

1. INTRODUCTION

It can be supposed that the first emulator was created when an old computer was replaced by a new one, non-compatible with the original. It was necessary to transfer most of programs from previous computer to a new one. It can be done in several ways: **(a)** *re-compiling of source codes on a target machine* - however it's not always simple and possible task (because of hardware differences); **(b)***rewriting programs to a new machine*; **(c)***translation of binary code for old machine into binary code for new machine* including translation of system calls; and **(d)***building an emulator* for the old machine.

Further we use the term "source computer" that stands for an original, old (emulated) computer and a term "target computer" for a real computer we use for emulation.

In general it is a difference between simulation and emulation. A simulator simulates behavior of a system in very accurate way in every aspect. For example a Turing machine can be simulated by another one without any memory, instruction or performance loss. An emulator is less accurate simulator [1], emulator accuracy is discussed in section 6.

Now we sketch a way, in which the simulator (and as a consequence also an emulator) of some machine can be constructed. We will start with well known Turing machines [3–5], and show a connection of Universal Turing machine with Von-Neumann architecture.

2. THE TURING MACHINE

Turing machine (TM) is an abstract computing machine - abstract automaton, as depicted in Fig. 1.

The control unit **M** represents a state mechanism (or a register), which in every time holds a state q. Using actual instruction and an information read by read/write head **h**, **M** can change its state to a new state p.

The tape **t** is boundless (from the left and right), linear sequence of cells, where in every time only a finite number of them is non-empty. Empty symbol in Fig. 1 is marked by symbol B. Every non-empty symbol belongs in a set of an input alphabet Σ of TM.

The head **h** is a mechanism that can "see" content of actual cell (above which is situated) and can perform in one time only one from three admissible actions (in consideration of actual cell):

- read symbol
- rewrite symbol
- move head left, or right

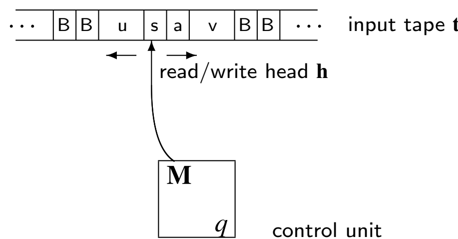

Fig. 1 Turing machine

According to [3]:

Definition 2.1. *In a formal way, TM is ordered 5-tuple* $M = (K, \Sigma, \Gamma, \delta, q_0)$*, where*
- K *is the finite set of states*
- Σ *is the finite set of symbols of input tape, so-called input alphabet*
- Γ *is the finite set of all symbols of TM's tape, i.e.* $\Sigma \subseteq \Gamma$
- $\delta : K \times (\Sigma \cup \Gamma) \rightarrow 2^{K \times (\Gamma \cup D)}$ *is TM's transition function, and* $D \in \{R, L, N\}$
- $q_0 \in K$ *is the initial state of TM M.*

There exists both non-deterministic and deterministic Turing machines, we will further consider only deterministic ones.

An instruction of deterministic TM is represented by a value of its transition function: $\delta(q_i, s_j) = (q_k, s_l \cup D)$, where $\{q_i, q_k\} \in K$; $\{s_j, s_l\} \in \Gamma$; and $D \in \{R, L, N\}$. It is possible to express the instruction in following form:

$$p_n : \; q_i \, s_j \, s_l \, D \, q_k \tag{1}$$

A set of all TM's instructions is called a program of TM.

Situation in TM is clearly characterized by a configuration of TM. According to [4] is a configuration of TM

ordered 3-tuple $c = (q, x, i)$, where q represents actual state of control unit **M**, x represents content of input tape **t** and i is sequential number of tape's actual cell position counted from the first non-empty symbol of the tape **t**, from left to right (started from 1).

Let's suppose, that we have two configurations $c_1 = (q_1, x, i)$ and $c_2 = (q_2, y, j)$, $\{q_1, q_2\} \in K$, $\{x, y\} \in (\Sigma \cup \Gamma)^+$. We say, that TM M has a step of computation from configuration c_1 to c_2 applying the instruction p_n (marking: $c_1 \vdash^{p_n} c_2$), if the following condition is true:

1. $p_n : \exists (q_2, s_2\ R) \in \delta(q_1, s_1) \Rightarrow x = us_1v,\ |u| = i-1,\ y = us_2v,\ j = i+1,\ \{s_1, s_2\} \in \Gamma$
2. or $p_n : \exists (q_2, s_2\ L) \in \delta(q_1, s_1) \Rightarrow x = us_1v,\ |u| = i-1,\ y = us_2v,\ j = i-1,\ \{s_1, s_2\} \in \Gamma$
3. or $p_n : \exists (q_2, s_2\ N) \in \delta(q_1, s_1) \Rightarrow x = us_1v,\ |u| = i-1,\ y = us_2v,\ j = i,\ \{s_1, s_2\} \in \Gamma$

More about Turing machines in general can be found in [5].

2.1. Universal Turing Machine

In work [5] by Alan M. Turing is shown, that it is possible to invent a single machine that can be used for simulation of another computing machine. Turing marked this machine as U and it's called Universal Turing machine (Fig. 2).

Machine U can compute any computable sequence. This sequence is represented by input tape t_U. Let's have another computing machine M. If input tape t_U is composed of encoded machine M with its input and output tape, machine U will compute the same sequence as machine M, i.e. machine M will be simulated on the machine U.

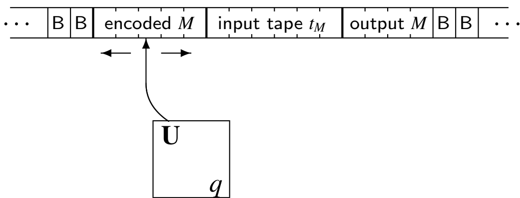

Fig. 2 Universal Turing machine

Outline construction of the U machine
Turing constructed his U machine as an example. However, his way of U's construction is not the only one. Some others tried to minimize the machine (C. Shannon, M. Minsky, S. Wolfram, ...). Here TM U will be described according to [5].

Let's have TM $M = (K, \Sigma, \Gamma, \delta, q_0)$, where $K \in \{q_0, q_1, ..., q_n\}$, $\Sigma \in \{0, 1, 2, \ldots\}$ is linearly ordered, $\Gamma \in (\{B\} \cup \Sigma)$, δ is a transition function and q_0 is initial state of machine M in the sense of definition 2.1.

If we want to construct the machine $U = (K_U, \Sigma_U, \Gamma_U, \delta_U, q_{0U})$, we have to specify the input for the machine (i.e. encoded machine M) and all the rest of U's sets - states (with initial state), alphabets (input/output) and finally a transition function.

First we'll encode the M machine. For encoding purposes Turing used only a form (1) to express the instruction. Every instruction and configuration of machine M can be encoded using only seven symbols $\Sigma_U = \{A, C, D, R, L, N, ;\}$. Next:

1. $\forall q_i \in K$ will be encoded by string $DA^i \in \Sigma_U^+$, where $A^i = \underbrace{AA \cdots A}_{i}$
2. $\forall s_j \in \Sigma$ will be encoded by string $DC^{j+1} \in \Sigma_U^+$, where $C^{j+1} = \underbrace{CC \cdots C}_{j+1}$
3. Empty symbol $B \in \Gamma$ will be encoded by symbol $D \in \Sigma_U$
4. Symbols $\{R, L, N\} \in \Sigma_U$ represents movement of a head: to the right (R), to the left (L) and without movement (N). So they have the same meaning like in machine M.

Input tape $\mathbf{t_U}$ contains two successive strings: **(a)** *a string describing a table of M's instructions*; **(b)** *a string describing input tape t_M*.

Work of U is based (besides its instructions) on its input tape $\mathbf{t_U}$ just like work of M is based on its input tape $\mathbf{t_M}$.

Encoded machine M is assembled into input tape $\mathbf{t_U}$ in following format: it begins with two symbols e one after other, then follows the code started with symbol **;**, and instructions are separated by **;**. Code is also separated out on single empty symbol B. The string of encoded code is then ended by double-colon symbol ::.

Suppose machine M contains two instructions: $q_1S_0S_1Rq_2$; $q_2S_0S_0Rq_3$. These will be encoded as follows:

```
ee;BDBABDBDBCBRBDBABAB;BDBABABDBDBRBDBABABAB::
```

The transition function of machine U is responsible for decoding of symbols on input tape. Places like actual state, instruction, input and action have to be marked. For this purpose machine U uses advisory symbols **u**, **v**, **x**, **y**, and **z**. In stages of U's work are these symbols stored on empty spaces (marked with B) of its tape $\mathbf{t_U}$ on the right side from "marked" place.

For example actual instruction is marked by symbol **z** on the right side of ;, **x** marks actual U's state:

```
ee;zDBAxDBDBCBRBDBABAB;BDBABABDBDBRBDBABABAB::
```

Detailed description of this machine is presented in mentioned work [5].

Thereby is shown that it is possible to simulate one TM by another (Machine U is also TM and has the same facilities). Now it will be enough to show that all real computers represent some subset of Turing machines and therefore also real computers can be emulated (but not always simulated) by another real computer.

3. COMPARISON WITH REAL MACHINES

Turing machines represents a foretoken of computer era that we know today. Conception of single memory (for in-

structions and data) and isolated control unit for executing instructions was an inspiration for well-known mathematician John Von-Neumann, who's computer architecture is used till present time.

Turing machines are the most powerful computers indeed and they're able to cope with any algorithmically computable problem. However a real construction of the machine isn't possible. The reason is its infinite tape.

Nevertheless almost all present computers are turing-complete and therefore everything what can real computer compute, also TM can compute [6].

Every real machine represents in fact a deterministic finite automaton, because it can have only finite number of configurations, exactly as TM.

The difference is only in TM's ability to manipulate with unbounded amount of data, even though TM in final time (as well as real machines) can manipulate only with finite amount of data.

Like TM, even real machines can have such big storage space as they need e.g. by adding more disks or other devices (of course not to infinity). But the fact is that for lot of useful computations neither TM, nor real machines don't need extremely huge storage space.

4. VON NEUMANN ARCHITECTURE

Von-Neumann architecture is a model of computer design, which was first used in EDVAC computer. This model uses single CPU (Central Processing Unit), single separated memory for storing both instructions and data, and input/output devices. Its slightly modified version is shown in Fig 3.

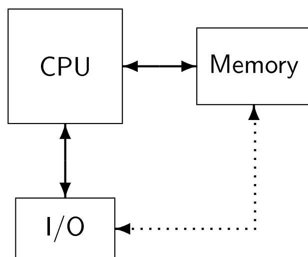

Fig. 3 Von-Neumann architecture

According to [2], Von-Neumann was inspired by the Universal Turing machine and his work was based on Turing's work. The Neumann's conception is also known as *Stored-Program-Computer*. Every Von-Neumann's computer implements Universal TM and in fact correspond to Flynn's architecture **SISD**[1] [7] (Fig. 4).

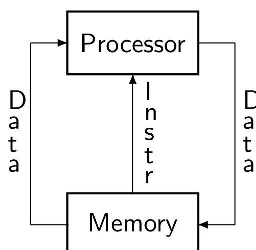

Fig. 4 Flynn's SISD architecture

[1] **S**ingle **I**nstruction **S**ingle **D**ata stream

An emulator has to reflect architecture of emulated computer. Further we will use the Von-Neumann architecture. The core of the architecture is mentioned CPU - Central Processing Unit. This becomes the core of the emulator, too.

5. THE BASIC STRUCTURE OF AN EMULATOR

We can look onto a work of real computer as on some kind of repeated activity. The CPU execute instructions in infinite loop [7], therefore main part of emulator's work is a cycle (loop).

In real computers the rest of hardware communicate with CPU, and work of the hardware is often parallel with CPU's work. In most cases an emulator (with emulated CPU and hardware) is executed as one single process (or thread) that is running on single computer and therefore true parallel work of hardware with CPU is not possible, however their common work is necessary. Parallel work of the hardware can be partially solved using multithreading (and synchronization) or distributive approach.

Parallel work of CPU and hardware in linear (non-parallel) environment is solved using some kind of planner or scheduler. The emulator assigns a time in slices to all devices and also to CPU sequentially. A device with assigned time slice is working, while others aren't. Basic algorithm using simple scheduler is shown in Fig. 5.

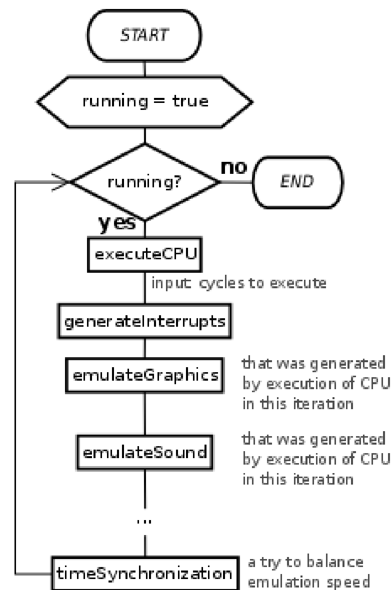

Fig. 5 Basic algorithm of emulator's work

A type of the scheduler in algorithm shown is similar to preemptive round-robin. Every device is (in always the same sequence) assigned a fixed-time slice (or slot), in which the device work. If the slice is elapsed then a new time slice is assigned to the next device.

A CPU can support processing of interrupts (internal, external). If it does so, and are satisfied certain conditions, it is necessary to generate them.

Time synchronization perform a speed balance of running emulation in a way that the speed should be as close as possible to the speed of source machine.

6. ACCURACY

If we are using the term ”emulator”, it has an effect to speak about accuracy, because an emulator skips some specific details of emulated hardware. A skip/replacement of the details of the hardware by some abstraction often leads to increase of emulator's performance.

Following this knowledge the emulator's accuracy is a range in what is the behavior of the emulator similar to the behavior of the real hardware. There exists several levels of accuracy, the most important are: **(a)***accuracy of internal representation of data and states* and **(b)***time accuracy of instruction cycle (timing)*.

An effort of achieving an accuracy in some level can imply a decrease of accuracy in other level. For that reason the designer of an emulator should know what level of accuracy requires the nature of the emulator.

6.1. Internal representation

Internal representation of data, data paths, states and internal communication is not visible from the outside. Therefore an effort of achieving an accuracy in this level is not frequent.

It can be required by the hardware designers that are testing new architectures, or can be used in hardware emulation with atypical need of synchronization.

6.2. Instruction cycle timing

Timing of instruction cycle is an ability of the emulator to perform some instruction (or some number of instructions) in relative same time as if the instruction would be performed on a source machine.

In fact this is the most emphasized level of accuracy, because software running in emulator is often synchronized with the speed of source computer (real-time applications, games).

7. EMULATION TECHNIQUES

Now we know, that executing a code on emulator or on a real machine gives the same results. But techniques used by emulator for code execution can vary.

There exists two main emulation techniques: **(a)***code interpretation* and **(b)***binary translation*. They can be combined together in order to achieve the most powerful emulation.

7.1. Code interpretation

It is the simplest technique, which ”clean form” is in favor used for emulation of older[2] 8 and 16 bit computers (not very complex and fast).

[2]fourth generation, years 1975 till 1990

[3]e.g. processor Zilog Z80, Intel 8080 and others

Processor executes program's instructions in a sequence. This cause a change of CPU's internal states and interaction with processor's environment (interrupts, communication with peripheral devices, etc.)

Instructions are sequentially executed in instruction cycles. Instruction cycle is a sequence of elementary steps, during which the processor executes an operation defined by the instruction. This cycle is divided into two phases [7]: **(a)***phase of instruction fetch* and **(b)***phase of instruction execution*. These phases can overlap[3], as it is shown in Fig. 6.

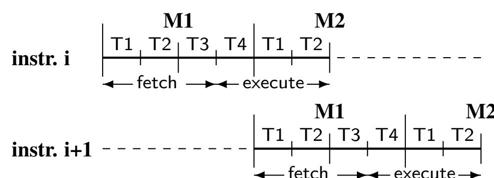

Fig. 6 Instruction overlapping in Zilog Z80 processor

Code interpretation technique allows the emulator to imitate the CPU. Phase overlapping isn't usually implemented, with exception for flow processing of instructions (or VLIW architectures [7]), and can be done only by use of interpretation technique. In this case, phases of instruction cycle are separated (let's suppose an existence of phases **F**etch, **D**ecode, **E**xecute and **S**tore). These are processed in correct sequence for instruction and in every time only one phase is processed.

If phases are overlapped, the easiest way of implementation is sequential (or linear) execution of all ”parallel” phases (i.e. that should be executed in time t_i), and in time t_{i+1} continue executing next sequence of ”parallel” phases (Fig. 7).

According to Fig. 7 an emulator will execute all phases in order: F1-D1-F2-E1-D2-F3-S1-E2-D3-F4-S2-E3-D4-S3-E4-S4.

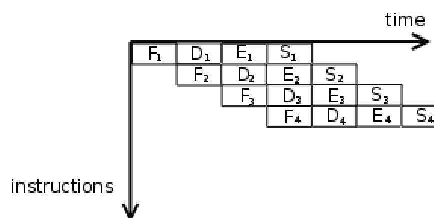

Fig. 7 Example of scalar execution of four instructions

The main algorithm of CPU's work (block `executeCPU` from the algorithm in Fig. 5) is a loop, where some number of instructions is interpreted (decoded and executed). Input for the loop is number of machine cycles, which should be executed (`cte`) and output is number of machine cycles, which were truly executed (usually few machine cycles more as was needed) (`ec`).

```
ec = 0;
while (ec < cte) {
```

```
    opcode = fetch_from_memory(PC);
    PC = PC + 1;
    tmp = decode_and_execute(opcode);
    ec = ec + tmp;
}
```

Fig. 8 CPU's interpretation algorithm

For sake of simplicity we can suppose that all instructions are only one byte long and therefore program counter (PC) increment is correct.

Advantages of the technique are: simple debugging of an emulator, portability, easy instruction cycle timing, the ability of fit the implementation of instructions realization for target CPU.

The main disadvantage is low performance of the emulation (for emulation of mentioned older computers isn't so noticeable, but it will be for emulation of faster and more complex computers), because besides from pure execution time also interpretation management has to be taken into account.

The speed of interpretation can be increased using some similarities between source and target CPU (e.g. flags generation, or some instructions - for example it is possible to translate all instructions of processors i8080, i8085 and Z80 into instructions of x86 processors), or using special technique called *threaded code*. In the technique a program is decoded (in static or dynamic way) resulting into so-called decode table. Its i-th item represents an absolute address i in memory and item value is an address of function (implemented in emulator) used for instruction execution on address i. Then in sequence for next PC (first for initial value of PC) are called functions from the decode table. More about the technique you can find in [1].

7.2. Binary translation

This technique isn't very similar to the work of a real CPU. Its aim is to increase emulator's performance, not by creating "virtual CPU" which interprets a code, but this code (in binary form) is translated statically (once before first run) or dynamically (in run-time) into code for target machine. Resulting code is then executed on a target machine. As it was mentioned, according to the time when the code is translated, we distinguish **(a)***static* and **(b)***dynamic* binary translation. Even if the term "binary translator" can be understood as some kind of compiler, it is a common term for application that performs both translation and execution of the code.

Work of the binary translator can be divided into two phases: **(a)***translation*, wherein code for source CPU is translated into code for target CPU **(b)***execution*, wherein CPU executes translated code. Both phases are very different and should be studied independently.

Translation phase is much about similar to a compiler. Input to a binary translator is however binary (machine) code, therefore it is possible to omit some phases of compilation (lexical, syntactic and semantic analysis) and replace them by just one decode phase.

7.21. Static binary translation

This is not a real technique for CPU emulation. Static binary translator is rather some interface or set of tools, by which is a code for one architecture translated into a code for another architecture. Result is usually an executable file, which can be run on target machine without any other special tools.

However sometimes it is not possible to do this kind of translation. An example is self-modifying code, where it is hard to say, how this code will look in run-time.

This technique is used rarely (as an opposite to dynamic binary translation). More about static binary translation you can read in [11].

7.22. Dynamic binary translation

It is some alternative to static binary translation and its use can be seen mainly in JIT (Just In Time) technology [12], which is the core of .NET technologies, or most of JVM (Java Virtual Machine) implementations.

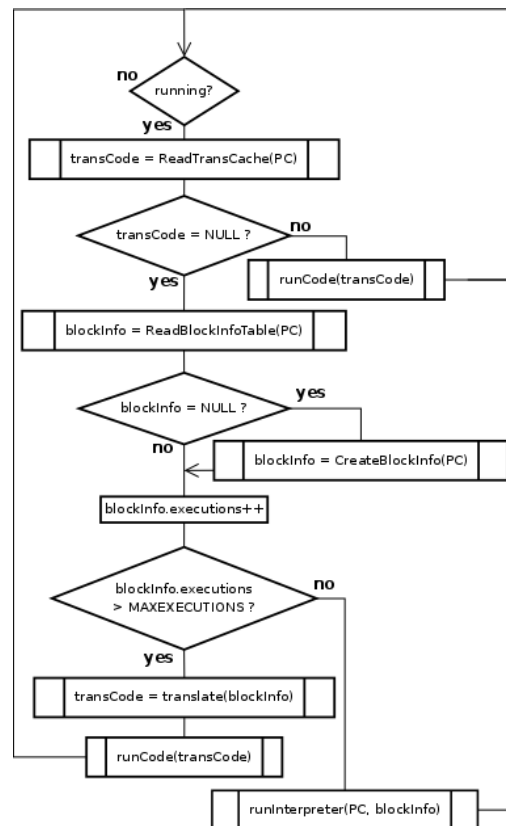

Fig. 9 Dynamic binary translation with interpreter

Basic algorithm of work of dynamic translator is similar to interpretation using threaded code technique (section 7.1). The algorithm consists of two phases, the first performs translation of code part, second executes the translated code. It can work in two ways: either using interpretation, too, or not. Sample algorithm which uses interpretation is shown in Fig. 9.

Given algorithm is using additional cache memory (`translatedCache`), where all the translated blocks of

code are stored. Mostly it is implemented as a hash table, because it needs very fast access.

Table `blockInfoTable` identifies particular (not translated) blocks of code, which besides from an address, code size and profiler informations contains also an information about number of the block's interpreted executions. If the number exceeds maximal value (`MAXEXECUTIONS`), the block is then translated into cache table `translatedCache`. That's how an emulation is speeded-up, because if a code is executed rarely, then isn't worthwhile to do expensive translation (value `MAXEXECUTIONS` is determined in empiric way and therefore represents an input parameter for this algorithm).

More about dynamic binary translation can be found in [1, 13].

8. HIGH LEVEL EMULATION

This technique is used for a performance increase of hardware emulation. First time it was used in emulator Nintendo64 UltraHLE [8].

In run-time the emulator is searching for emulation code of source hardware. If it succeeds, the code is converted into code for target hardware and this code is then used. UltraHLE works in a way that before execution of the emulation it searches entry points of functions (in emulated operating memory) that access to graphic and/or sound card. On found places the emulator puts a "marker".

Then in run-time if marked functions are going to be called, the emulator takes their arguments and use them for implementation of native functions and objects of target hardware, which will have the same or similar behavior[4]. This results to a performance increase of the emulation, because the hardware emulation is provided directly on a target hardware.

More about HLE can be found in [8, 9].

9. EMU STUDIO — MICROCOMPUTER EMULATION PLATFORM

Now we present one concrete solution, which is developed by the second author of this paper as Semestral project since 2006. Project is being solved till today, as his diploma work. Main motivation in development was/is to present an architecture of older (8-bit) computers to students in more practical way.

Initially it was simple emulator (with powerful assembler compiler) of 8-bit Intel 8080 processor, now it became emulation platform for microcomputers, not necessarily 8-bit, where user using advisory addons (plugins) can create arbitrary configuration of any microcomputer (and eventually even some abstract machine). In the present time there exist besides of Intel 8080 also Zilog Z80 processor, which is its 8-bit successor, and several devices are implemented: terminal (display and keyboard), floppy disk drive, serial card.

Seeing that purpose of the project didn't change (it is aimed to students and laics), it was made by keeping very high transparency, configurability and simplicity. User-student trying to learn something about given platform, has to have briefing about important actions in run-time emulation, therefore an interaction precede speed and performance of the emulator[5].

In order to student wouldn't be limited by a target platform, we decided to implement the emulator in Java language (what, from performance point of view, is not optimal decision). As we determined later, present performance in CPU emulation is sufficient. Emulator was tested on several target machines. We was emulating Intel 8080 CPU (which had original frequency 2 MHz). A target machine with Intel Core2 Duo processor, frequency of both cores 1,66 GHz and 1GB RAM can run in range of 50-80 MHz. Using another target machine with Intel Pentium 4 processor, 2.4 GHz, with 248 MB RAM by using the same CPU and program can run in range 30-40 MHz. So performance depends mainly on target processor speed (and/or number of cores) and target RAM size.

9.1. Plugins

It is an advantage if an emulator can adapt itself for software requirements on hardware. This can be done by changing configuration of emulated computer. Older computers (e.g. Altair8800) didn't have so much peripherals, so there couldn't arise any conflict by using all of them.

In that we didn't want to remain with emulation of only one computer what implies a need to support of configurability of an architecture. Thus we splitted the emulator into main module and plugins, which can be dynamically altered. That's a way in which different configurations can be created.

The emulator supports four categories of plugins: **(a)**_compilers_, **(b)**_CPUs, and their disasemblers_, **(c)**_operating memories_ and **(d)**_peripheral devices_.

Every category has assigned its own directory. Accordly an emulator can distinguish between plugin types.

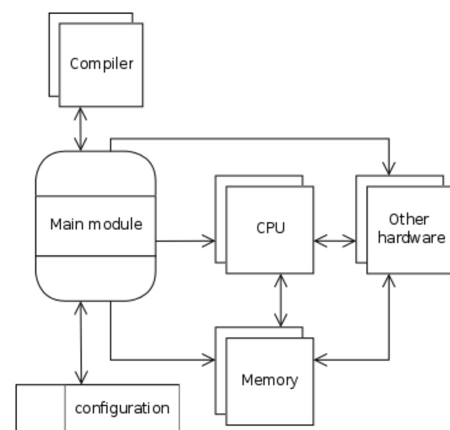

Fig. 10 Cooperation of main module with plugins

Cooperation of plugins with main module is shown in Fig. 10. As can be seen, the schema is similar to Von-Neumann's architecture (Fig. 3).

[4] In UltraHLE case is 3D API of N64 console translated into 3DFX Glide API (subset of OpenGL 3D API)

[5] For this reason is CPU emulated using interpretation, which allows to examine every executed instruction in detail

ISSN 1335-8243 © 2008 FEI TUKE

The configuration of an architecture is therefore made up of plugins selection. Except the peripheral devices (whose number can be arbitrary - according to the CPU support), the configuration has to consist of exactly one plugin from every category. Formally a configuration is ordered 4-tuple $C = (CPU, Compiler, Memory, DeviceSet)$, where $DeviceSet \subseteq \{Device_1, Device_2, ..., Device_n\}$ and n is the number of all devices.

9.2. Main module

This module is a main element of an emulator. It can be executed independently. It manages and selects a configuration of source machine's architecture (immediately after start of the application) and then loads all needed plugins (in the sense of selected configuration). These plugins are initialized with default values (storing/retrieving a configuration of the plugins isn't implemented yet). Main task of the module is to integrate all elements of the system in a way that they would behave like a whole. It also performs basic interaction with the user.

Main window (Fig. 11) is divided into two switchable panels:

Panel of source code gives to user all resources for program creation phase - text editor and interaction with compiler. Text editor supports syntax highlighting with line numbering, what helps to find potential bugs in code. Messages from compiler are written in bottom part of the window. Text editor support functions like *undo*, *redo* and of course it can work with *clipboard*.

Panel of emulator is divided into three parts - debugger (or debugging window), status window and the window of peripheral devices. Debugging window shows a list of instructions located in operating memory with address close to program counter (`PC`) value. A row with actual instruction (to which `PC` points to) is highlighted. In the window the user can control (or manage) a progress of emulation with several functions - *reset*, *run*, *step*, *stop*, *break*, and also the ability of explicit setting of `PC` value (do a "jump"). CPU can support breakpoints (present implementation of mentioned Intel 8080 does). A breakpoint is used to mark some arbitrary memory address.

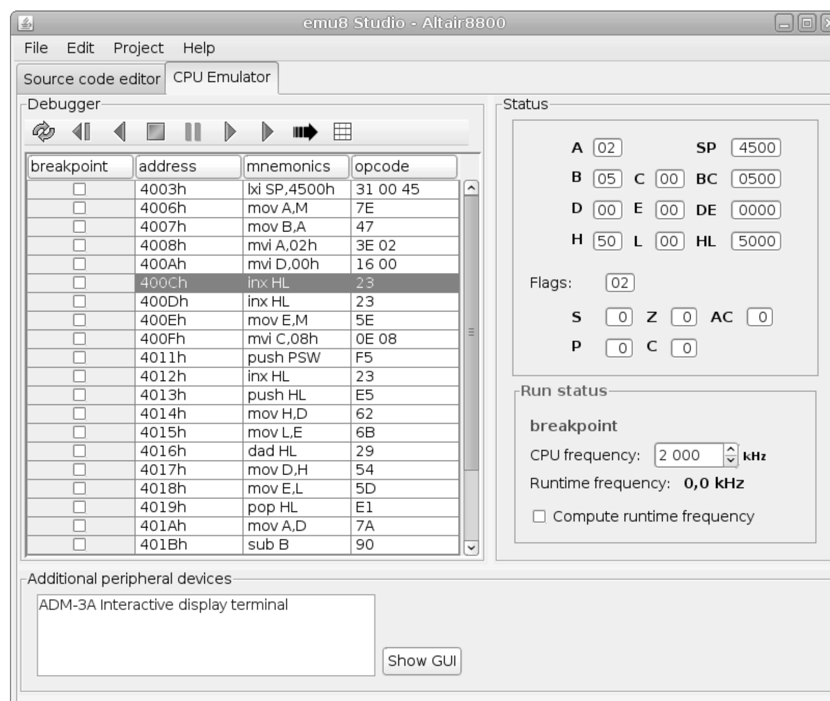

Fig. 11 Main window

If (in run-time) the `PC` will be equal to address which is marked as breakpoint, the emulation is paused. The user can then decide what to do - to continue, step, or stop the emulation.

Status window shows actual state of CPU used and its content is all implemented in the CPU plugin and therefore it is different for every kind of CPU. Usually it displays CPU registers' values, flags and other useful information.

Window of peripheral devices contains a list of all loaded (and usable) devices. By selection of a device it is possible to run its graphical interface (if the device has some).

10. PLUGIN DESCRIPTION

Now we describe plugins for currently implemented architecture of computer MITS Altair 8800 [14], which used Intel 8080 processor, and maximal size of its operating memory was 64kB. It could have these peripheral devices: serial and parallel interface, magnetic tape reader, floppy

disk and harddisk. Interaction with user was through a terminal connected to serial interface.

10.1. The compiler plugin

In present time only assembler compiler is implemented. Its specification is similar to original specification [15]. However it contains some extensions, e.g. unlimited size of labels, unlimited deepness of nested macros, etc.

The output of the compiler is a file in Intel HEX [16] format, which was both in past and in present often used and many (older or specific) programs are compiled into this format. The aim was to preserve "bidirectional compatibility" - a program created in emulator can be run on real source machine and vice-versa program created in real source machine can be run in the emulator.

10.2. The CPU plugin

The CPU emulation uses interpretation technique (described in section 7.1). The communication with main module and other plugins is performed via interfaces.

The CPU gives to the user a possibility to set the clock speed of emulated processor in range from 100 to 99999kHz. Of course these are logical values and in real conditions we didn't reached the top (system has to be a little overdesigned). Implementation of ensuring the frequency on a target machine with different frequency is very interesting (synchronization of frequencies is ensured only in emulator's run-time, not in stepping).

```
running = true;
while (running) {
    startTime = now_time();
    ec = 0;
    while (ec < cte) {
        opcode = fetch_from_memory(PC);
        PC = PC + 1;
        tmp = decode_and_execute(opcode);
        ec = ec + tmp;
    }
    generateInterrupts();
    emulateGraphics();
    emulateSound();
    ...
    // time synchronization
    endTime = now_time();
    timeLength = endTime - startTime;
    if (timeLength < timeSlice) {
        // time correction =
        // emulation is too fast
        wait(timeSlice - timeLength);
    }
}
```

Fig. 12 Algorithm of the emulation with time synchronization

The main algorithm of emulator's work (Fig. 5) is preserved. Let's combine the algorithm with CPU emulation algorithm (Fig. 8) into one single algorithm, where is implemented also the principle of time synchronization.

In the algorithm shown in Fig. 12, after every instruction fetch `PC` is incremented. This is not correct if instructions are more than 1 byte long (suppose N bytes). For that reason let's suppose that `PC` is incremented $(N-1)$ times in function `fetch_from_memory()`.

The algorithm uses new methods: `now_time()` - determines actual time (e.g. in *ms*) and `wait()` - waits certain time interval given as a parameter (e.g. in *ms*).

First some fixed time interval (`timeSlice`) is chosen, in which should be performed one single iteration of an emulation loop. This value is mostly empiric-defined.

Before every run of CPU emulation (second loop) and after of every device emulation is determined actual time. Now a list of conditions follows, which define how the emulation is synchronized.

$$endTime - startTime = timeSlice, \quad (2a)$$

$$endTime - startTime > timeSlice, \quad (2b)$$

$$endTime - startTime < timeSlice, \quad (2c)$$

Emulation is perfect-synchronized, if condition 2a is true. This state is however ideal, not real.

Emulation is slow (run-time frequency of emulation lower than it should be), if condition 2b is true. Algorithm in Fig. 12 doesn't implement speed up of the emulation, which is almost impossible to reach (can be partially reached by skip of showing some frames in video/display emulation, or by decrease sound quality, etc.).

Finally the emulation is fast (run-time frequency is above the value set), if condition 2c is true. Algorithm in Fig. 12 solves this problem by waiting some time (method `wait()`).

Final part is the calculation of the `cte`[6] parameter. This parameter depends on: **(a)***set frequency f*, **(b)***emulation time of other devices t_h*. Next, one machine cycle is performed in one period of CPU frequency and for every instruction we know number of its machine cycles in which the instruction is performed (slices T_i in Fig. 6).

Formula for `cte` calculation without consideration of emulation time of other devices can be derived as follows:

$$T = \frac{1}{f} \quad (3a)$$

$$cte = \frac{timeSlice}{T} \quad (3b)$$

$$cte = timeSlice * f, \quad (3c)$$

If we know the fixed emulation time of other devices t_h, then in the formula 3c value *timeSlice* is replaced by the value of $timeSlice - t_h$.

10.3. The memory plugin

Operating memory is cooperating with CPU in very close manner. According to Von-Neumann's architecture, there are stored both program and data. In the principle, operating memory represents linear ordered sequence of memory cells (mostly implemented as an array).

[6] cycles to execute in one time slice

There exists many types of operating memories. From programming/implementation point of view, it is worth to speak about two types only. Probably the most used type in the present is **RAM** (Random Access Memory) type, which represents a kind of memory, where it is possible to read or write data to any cell. Next type of memory is **ROM** (Read Only Memory) type, which represents a kind of memory, where it is possible to read data from any cell only.

Present implementation of the plugin can create one continuous sequence of cells (array), whose any part can be marked as **RAM** or **ROM** (it is possible to change it dynamically). The user can interact with the plugin by its graphic interface.

Some older architectures (and probably all present) supported direct device access to the memory (DMA). The plugin supports this option, too. In a case that a device wants to use DMA, it has to register itself by certain method of the plugin interface with a parameter representing a range of addresses that device wants to use. Then, if some change happens in the range, the plugin will send a signal to the device, so the device doesn't have to care about periodic checking the memory. However the device programmer should be careful of using address overlapping.

10.4. The terminal device plugin

Currently two peripheral devices are implemented - a terminal and a floppy disk. We will describe the terminal only. It is an emulation of the ADM-3A [17] terminal, that was used by the Altair computers.

It is a pure text-mode display with a keyboard. This way users can enter input data into a computer and also they can see results of the computation.

The device was plugged into serial card MITS SIO through a standard RS-232 interface. Up to now the emulator doesn't support the ability of hierarchical device connections, and for that reason the serial card is implemented into the terminal directly. A work of the terminal however is similar to the real device.

11. CONCLUSIONS AND FUTURE WORK

The main asset of the emuStudio is its structure. The main module is intended to be an underlayer for various plugins that present an emulated architecture. By adequate selection of plugins user can create configurations of emulated hardware that exist in real world. This feature poses the user into position of an architecture designer, and authors didn't seen any available computer emulator software having this ability.

Using this advantage, user can emulate also other machines than PC computers, as embeded devices, or abstract machines.

Thanks to plugin implementation, the emulator is going to be perspective software, which exceeds boundaries of its purpose - emulation of 8-bit processors only. The license of the emulator gives programmers right for rewritting/modifying source code and there also exist developer manuals describing how to create plugins for the platform. Then it is only up to programmer himself, how accurate and interactive plugins will be.

A fact that the whole platform is implemented in Java language, makes the emulator very portable. It can run on every system that supports Java Runtime Environment (JRE), Standard edition.

Devices (such as harddisk, floppy drive, printer, display or keyboard) can be designed in a way that they will be in connection with real hardware and therefore also results will be real.

In the nearest future we want to implement reading/storing configurations for devices, hierarchical device connections and some new devices, such as tape drive. Till now we are able to run several operating systems (all from the RAM/ROM images): CP/M v2.2 (by Digital Research, year 1979), Altair DOS v1.0 (MITS, inc., year 1977) and Altair Basic v4.1 (MITS, inc., year 1977) image.

As the next step we plan to extend the emulator with another computer architectures, and also with abstract machines. The final result should be the emulator platform involving as large set of a teaching material as it is possible.

ACKNOWLEDGEMENT

This work was supported by VEGA Grant No. 1/4073/07 Aspect-oriented Evolution of Complex Software Systems.

REFERENCES

[1] BARRIO, V.M.: Study of the techniques for emulation programming, 2001 `http://personals.ac.upc.edu/vmoya/docs/emuprog.pdf`

[2] DAVIS, M.: The Universal Computer: The Road from Leibniz to Turing. *W. W. Norton and Company*, 2000, 257 pages, ISBN 0-393-04785-7

[3] HUDÁK, Š.: Theoretic Informatics, 2002 (in Slovak) `http://hornad.fei.tuke.sk/predmety/ti/ti.ps`

[4] HUDÁK, Š.: Machine oriented languages. *FEI of Koice*, 2003, 218 pages, ISBN 80-969071-3-1 (in Slovak)

[5] TURING, A. M.: On computable numbers, with an application to the Entscheidungsproblem `http://web.comlab.ox.ac.uk/oucl/research/areas/ieg/e-library/sources/tp2-ie.pdf`

[6] HOPCROFT J., ULLMAN J.: Introduction to Automata Theory, Languages and Computation. *Addison-Wesley*, 1979, 1st edition, ISBN 0-201-02988-X

[7] JELŠINA, M.: Architectures of computer systems. *ELFA, Koice*, 2000, ISBN 80-88964-41-5 (in Slovak)

[8] Ultra HLE `http://en.wikipedia.org/wiki/UltraHLE`

[9] High Level Emulation `http://en.wikipedia.org/wiki/High-level_emulation`

[10] SCHERRER, T.: Home of the Z80 CPU

http://www.geocities.com/SiliconValley/Peaks/3938/z80_home.htm

[11] ANGELONE, M.: Approaches for Universal Static Binary Translation, 2006
http://www.cs.drexel.edu/static/reports/DU-CS-06-02.pdf

[12] Just-in-time compilation
http://en.wikipedia.org/wiki/Just-in-time_compilation

[13] TRÖGER, J.: Specification-Driven Dynamic Binary Translation, 2004
http://savage.light-speed.de/pdf/phd2004.pdf

[14] Altair 8800
http://en.wikipedia.org/wiki/Altair_8800

[15] 8080 Assembly language programming manual, Intel corp., 1975
http://www.tech-systems-labs.com/booksdata/8080-asbly-pro.pdf

[16] Intel Hexadecimal Object File Format Specification, Revision A, Intel corp., 1988
http://pages.interlog.com/~speff/usefulinfo/Hexfrmt.pdf

[17] Lear Siegler, Inc. (LSI) Terminal ADM3A
http://www.tentacle.franken.de/adm3a/

Received March 22, 2008, accepted November 28, 2008

BIOGRAPHIES

Slavomír Šimoňák was born on September 23, 1974. He graduated from the Technical University of Košice, Faculty of Electrical Engineering and Informatics in 1998. He obtained PhD degree in the field of computer devices and systems in 2004. Now he is an assistant professor at Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice. His scientific interests are oriented towards formal methods for design and analysis of discrete systems and their integration, problems related to theory of programming and machine-oriented languages.

Peter Jakubčo was born on 2.6.1985. Currently he is a student at Department of Computers and Informatics on Faculty of Electrical Engineering and Informatics, Technical university in Košice. He is interested in computer emulation, operating systems and programming languages.