A GENERIC PURPOSE, CROSS-PLATFORM, HIGH EXTENSIBLE VIRTUAL MACHINE

John Vlachoyiannis
darksun4@gmail.com

Panagiotis Kenterlis,
P.Kenterlis@mprolab.teipir.gr

John N.Ellinas
jellin@teipir.gr

Department of Electronic Computer Systems
Technological Education Institute of Piraeus
P. Ralli & Thivon 250, 12244 Egaleo, Greece

ABSTRACT

Along with the emergence and public acceptance of different operating systems and platforms during the past few years, came the need for quick cross-platform development solutions. This need was partially fulfilled by cross-compilers, due to the variety of underlying hardware. Virtual machines (VMs), whose technology was revived by the evolution of hardware and the successful use in the fast evolving Internet (Java) and mobile devices, came to fill the void and became an invaluable tool for developers and users. Emulators/Simulators, graphic engines, operating system VMs, interpreted languages and Turing machines are some of the fields where virtual machines are being used nowadays. The drawbacks of virtual machines are the execution speed (which diminishes as processor speed increases) and the actual development, as the programming of a virtual machine requires time and a more advanced knowledge of programming.

This paper describes a prototype, open source, cross-platform Virtual Machine tool (named Generic purpose Nano Virtual Machine or gNVM), that minimizes the effort of creating a virtual machine from zero code level, by providing a cross-platform, fast, small-sized and highly extensible virtual machine. With gNVM, users can develop virtual machines for 8-bit processors, graphic engines, Universal Turing machines, programming languages, imaginary virtual machines using any kind of instruction set (even in their own national language) or express complicated functions through the simple editing of a human-readable text file, at a nominal performance cost. The implementation in C language ensures portability to other platforms and provides satisfactory execution speed.

KEYWORDS

virtual machine tool, cross-platform, universal Turing machines

1. INTRODUCTION

The classical approach for a VM implementation (more precisely VMs for emulators) is *full virtualization,* which employs a VM which attempts to emulate the full hardware architecture as accurately as possible [2]. The advantage of this method is that software requires no modification to be executed by the emulator (*pure virtualization*) and very accurate responses from the program can be obtained. Conversely, the engineeging cost of building emulation solutions for complicated hardware is enormous. For example, there is a plethora of Gameboy [13] emulators (a very successful handheld video game console by Nintendo) as well as Z80/8085/6502 processor based system emulators for which, even though there is little difference between the function of their Central Processing Unit (CPU), there is constant reinvention of the main parser and execution engine.

In this paper we will present the Generic purpose Nano Virtual Machine (gNVM), a prototype register-based Virtual Machine having significant differences from other purpose-specific VMs such as a cross-platform kernel, dynamic instruction set editable through a human-readable text file, a macro instruction system enabling the construction of complex instructions (*super instructions*) and being able to execute *self modifying code* (SMC). gNVM is a VM Creation Tool that aids in the creation of expandable and fast VMs, operating in many platforms, ranging from Unix to Symbian (mobile), at a minimal engineering cost.

2. OVERVIEW

gNVM strives to aid the fast and easy creation of a register-based VM, highly expandable and cross-platform executable. The main advantage of gNVM is that VM specification (number and name of registers, instruction table), is being read from a text file through initialization (.nvm file) and enables the easy creation of other VMs simply by editing that file. gNVM's capabilities are greatly extended with the presence of the Macro System, where an instruction (called super instruction) can be constructed by a series of smaller instructions, and the Extended VM Application Interface (API), where new instructions can be assigned to user functions.

2.1 GNVM CHARACTERISTICS

gNVM is a Virtual Machine incorporating many pioneering features which are very useful not only to students that like to experiment in assembly or learn about virtual machines, but also to any developer interested in creating a cross platform, powerful VM with ease and more importantly, in the least possible time.

Some of gNVM's features are :

- Easy creation of a new Virtual Machine without prior knowledge or code
- Configuration of the whole VM through comprehensible coding in text files
- Portability (gNVM has been ported even to mobile devices)
- Easy creation of super instructions through gNVM's Macro System
- Abstract number of interrupts, registers and instructions
- Small kernel (less than 40 lines of ANSI-C code)
- Binary translation
- Ability to include comments for every instruction (for educational purposes)
- Easy creation of a new Assembly languages even in native language (Greek)
- A MS-Windows Graphical User Interface (GUI)
- Programming a processor in another processor's assembly language
- Creation of new instructions that call external routines (rendering a 3d scene)
- Ability to emulate self modifying code (SMC)

2.1.1 VM'S INTERNAL STRUCTURE

The most popular virtual machines, like JAVA VM [6] or Microsoft's .NET [7], use a virtual stack implementation rather than the register oriented architectures used in real processors, due to the simplicity of their implementation and ability to produce more compact byte code. Even though, gNVM could be implemented as a stack-based VM, the register model was chosen due to the similarity with the processor's architecture, the low thrashing of stack and the reduction of executed byte code (resulting in faster execution).

Register machines have two major advantages which are being exploited by gNVM. Firstly, register-based VMs can reduce the number of executed instructions (leading to better performance) despite the increase in the number of fetches. Secondly, register-based VMs can effortlessly change the order of VM instructions. Reordering stack machine instructions is very difficult since all instructions use the stack and every instruction depends on the previous one. Register instructions can be reordered (provided there are no data dependences) permitting gNVM's Macro System to create super instructions efficiently and with ease.

2.1.2 CODE DISPATCING

The interpretation of a byte code involves fetching the byte code, performing the function of the instruction and dispatching(fetching, decoding and starting) the next instruction. The most time consuming part of the dispatching circle is decoding of instruction. The simplest and most widely used approach is switch dispatch.

gNVM uses call threading methods for dispatching code which ensures portability (based on ANSI-C) and provides good performance. Call threading is similar to indirect threading, with the overhead of calling a function.

2.1.3 EXPANDABILITY

gNVM instructions are constructed in an object oriented (OO) manner. The creation of an instruction is done simply by editing a .nvm text file following an easily comprehensible format as shown in Table 1.

Name	MOV A, B
Effect	equal_r1_r2
Time of execution	1 ms
Comments	Moves contents of register B to A
gNVM byte-code	14
Real Object code	0x44

Table 1. Instruction Format

Adding new instructions is performed by simply adding an instruction block like the above and defining the properties of the instruction. In addition, Extended VM API allows the user to create new instructions that enhance interaction between VM and other devices or programs (eg., instructing the VM to render a 3d scene). Combining these instructions, using the Macro System, we are are able to create super instructions as easily as creating a simple instruction.

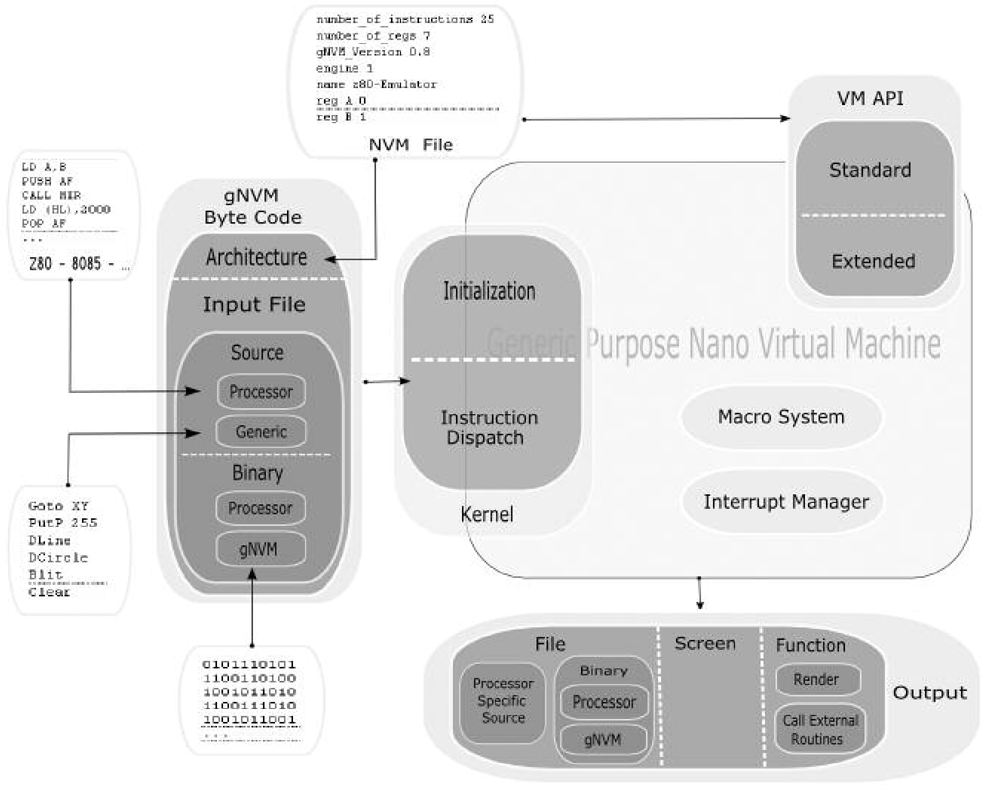
2.1.4 PORTABILITY

Even though many optimizations (such as thread tokening instruction dispatch or parts written in assembly) could provide an increase in performance [3], they could counteractingly obstruct portability to other platforms. A small kernel, along with the selection of the instruction set, ensures portability even in devices with small memory capabilities (like mobile devices) and more importantly, easy debugging.

3. IMPLEMENTATION

In our work we have implemented a virtual machine, according to the principles and architecture described in the prior sections, using function pointers and an instruction dispatch method similar to token threading in ANSI-C language. We have created a register-based VM which performs sufficiently well for our purposes and outperforms stack-based VMs due to the similarity with the processor architecture, the low thrashing of stack and the reduce of executed byte code (resulting in faster execution).

gNVM is a little endian register-based VM (though change to a big endian architecture is possible through the NVM specification file), with abstract number of registers, no garbage collection, ability to handle macros of instructions, multiple interrupts, two memories (one read-only and the other read/write). Additionally it has an easily expandable architecture and may be used for emulating even self modifying code (SMC).



```
number_of_instructions 25
number_of_regs 7
gNVM_Version 0.8
engine 1
name z80-Emulator
reg A 0
reg B 1
```

```
LD A,B
PUSH AF
CALL MIR
LD (HL),2000
POP AF
...
```

```
Goto XY
PutP 255
DLine
DCircle
Blit
Clear
```

```
0101110101
1100110100
1001011010
1100111010
1001011001
...
```

Fig. 1. gNVM's architecture

3.1 INSTRUCTION EMULATION

The instruction table is read from a text file during initialization. The implementation of a simple 8-bit processor (eg., Zilog's z80) is a procedure of describing the instructions of the processor inside the text file as shown in Fig. 2.

```
@LD (BC), A
op_state 136
effect equal_rel_reg_1
time_of_execution 1
comment load to (BC) contents of A
instr_len 1
gNVM_byte 59[1][2][0];
obj 02
```

Fig. 2. An instruction block

The above figure informs VM that "LD (BC),A" instruction has object code "02", executes in "1 ms", has a length of "1 byte" and comment "loads to (BC) contains of A". Changing the name of the instruction to be represented by another human language (for example in Greek), will allow its users to program and debug in their own native language, a feature of great importance when used for educational purposes. Additionally, users may choose to code their programs in their preferred assembly language (e.g, program the 6502 processor using Z80-like assembly instructions). Furthermore, changing an instruction's object code, we are able to convert the resulting binary code to another machine (*binary translation*). For example, a program written for a z80 processor, can be exported to 8085 very easily.

The most efficient method for dispatching the next VM instruction is direct threading [4]. Instruction dispatch consists of fetching the instruction's address and branching it to the routine, as this implements instruction's address.Unfortunately, indirect or direct threading cannot be expressed in C and while switch threading is the fastest and widely used method for emulators, is significantly slower. Using GNU's C compiler (gcc) and the *labels as values* feature (available since gcc 2.0), which makes direct and indirect threading possible, provide a speed increase in instruction emulation, however that would have a detrimental effect on portability as it violates the ANSI-C.

As a result, to maintain high portability and sufficient speed, we use *call threading*, a faster dispatching method than switch. The table of pointers is a table with instruction objects where the execution of each instruction is performed through the simple execution of a function (Object Oriented style). The advantages of using this method are the dynamic binding of instructions and a well formed specification of the instruction, starting from the object code and ending to a description (which can be used for educational purposes on emulators).

The fetching part occurs in virtual memory (fetching gNVM bytecode) but all the reading and writing are performed in real memory (with writing being performed both in real and virtual memory). By having two memories, we can have a compiled version of the program in gNVM byte code for executing and a real memory for writing data and printing the real object code. Any attempt for SMC is detected by a special mark at the beginning of every gNVM byte code. Absence of that mark, means that the data written on non-data memory (execution memory) are byte codes and VM should search the instruction table for the matching gNVM byte code. Failure to do so, doesn’t affect VM operation, however any attempt to execute that instruction, will result in an runtime error.

3.2 VIRTUAL MACHINE API

Each instruction from the instruction table is bound to a function during the initialization. This function can be part of the Standard VM API provided with the current implementation of gNVM, or Extended VM API, where the function is supplied by the user.

3.2.1 STANDARD VM API

Standard API is a set of functions that most processors have, like copying a register to another. The instruction set of Standard VM API is quite large (300 instructions - but it can be dramatically decreased during compilation time for memory or other resources limited devices) and has functions almost identical to an 8-bit processor. The main reason for having such an extensive Standard VM API is that in order to emulate a great range of instructions, we would use the Macro System (adding overthread to the system) or use a similar approach like the .NET Virtual Machine.

The .NET VM provides a very small API and determines the type of argument in real time, resulting in a high performance loss. Our approach is similar to the Parrot VM [8], another very fast register-based VM with many instructions. As previously stated, we can shorten the Standard VM API during compile time, and either avoid the use of same instructions or create them with Macro System, if possible. We can add more functions using the Extented VM API or Macro system by altering the nvm file.

3.2.2 EXTENDED VM API

Extended API is a set of functions described by the user, which effectively allows for user customized functions of any kind. The main difference from Standard VM API is that the gNVM kernel must be recompiled with these functions. By using the Extended VM API , users can create instructions that perform complicated task not directly related to the VM as shown in Fig. 3.

```
@COPY (src_file), (dest_file)
op_state 66
effect ex_copy
time_of_execution 0
comment copies a file
instr_len 5
gNVM_byte 9;
obj 55
```

Fig. 3. Using Extended VM API

The new COPY instruction, calls a function found in extended.h, where it takes the two filenames from the specified memory and makes a copy. It is the same with the "cp source dest" command found in Linux shell. Thus, we can create any kind of command available to gNVM and then use it. Futhermore, Extended VM API, has control over VM properties (like registers and memory), so it is possible to mix functions with VM properties.

Adding functions to the Extended API requires that after the function is coded, then it must be compiled with gNVM's kernel.c and finally added to the extended.nvm file.

3.2.3 MACRO SYSTEM

Macro instructions are instructions formed by grouping other instructions. For example, lets assume you have created instructions COPY (src file,dest file) and DELETE (file). You can create instruction DO ALL that does both of them like in Fig. 4.

@DELETE (filename) op_state 66 effect ex_delete time_of_execution 0 comment deletes a file instr_len 3 gNVM_byte 10; obj 56	@DO_ALL (src_file) op_state 66 effect MACRO time_of_execution 0 comment instruction doing a move instr_len 3 gNVM_byte 0;3;9[X][X];10[3000];0 obj 57

Fig. 4. Using Macro System

By invoking DO ALL, the COPY instruction is executed and then the DELETE instruction with the filename being an argument the must exist in memory address 3000h.

A macro instruction, can contain instructions from both VM APIs and there is no need in recompiling the gNVM kernel. The addition of macro instructions, is carried out by editing of the nvm file.

3.3 INTERRUPTS - IO

An Interrupt manager is being invoked every time after an instruction is dispatched. A table of interrupts (formed by Extended API functions) is being used to look up the occurring interrupt's table entry. Every interrupt object has three members that bind to two functions from Standard or Extended VM API.

• is on
• wait for
• what to do

Initially, the Interrupt manager checks for the "is on" member. If it equals to ’1’, the interrupt is hooked and every VM dispatch cycle the "wait for" function (which is a part of Standard/Extended VM API) is executed. If it returns true (the condition is met, interrupt condition) the "what to do" member/function (Standard/Extended VM API) is being called.

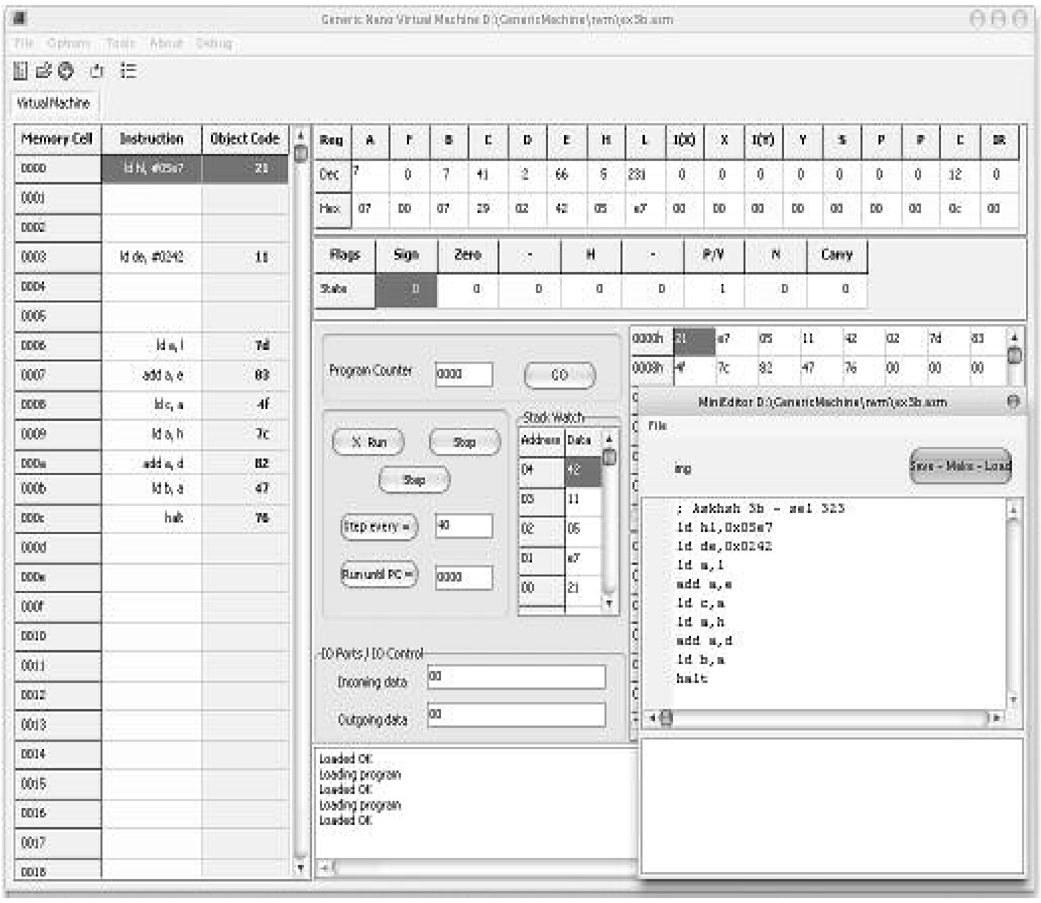
Using this simplistic yet powerful system approach, we can implement interrupts that interact with a variety of sources and devices. For example, we can place a "wait for" hook function to query a POP3 server for email and if a new message arrives then an interrupt will be invoked, calling the "what to do" function (possibly a "fetch email message" function). The Interrupt manager can handle an abstract number of interrupts, however the check for each interrupt, affects VM execution.

4. GNVM IN EDUCATION

One of the most difficult subjects for students entering the field of computer science is the assembly language and how processors work. Usually emulators targeting to that audience are pretty much outdated (many of them run only in MS-DOS), are proprietary (thus students cannot fiddle with their source and find how they work or expand them) and have many problems in their execution to other similar platforms (for example Windows XP). gNVM allows students to experiment on their favorite platform (even on their mobile device) and provides the full source under an open source licence.

By altering the NVM file, students can experiment in writing assembly in their own native language or even create a totally new assembly for their own VM without the need of knowing a programming language. More importantly, they can experiment with interrupts which are mapped to real devices (devices in parallel ports for example) and consult the built-in instruction's commenting system for more information regarding an instruction.

Providing a Windows GUI for writing, assembling and debugging programs, which has a Windows XP look-and-feel, students can experiment more easily with the assembly language. The GUI is a front-end to the gNVM kernel. It consists of a grid for displaying instructions and their object code, a grid for registers and flags (which can be manipulated in real time) and two grids for memory access. If a binary file is loaded, the corresponding ".asm" file (if exists) will be loaded to the a *Mini Editor* where corrections can be made. A stack grid exists for displaying the current stack contents and buttons for controlling program execution (Run, Stop, Step, Step Every X seconds, Run until PC = XXXXh). In addition, text boxes can be configured for interacting with interrupts or IO operations. Also, using the Extended VM API, we can assign interrupts and IO operations to be mapped to/from other real devices, enabling the students to see the results in real (non-emulated) hardware. The appearance of the GUI is illustrated in Photo 1.



Generic Nano Virtual Machine D:\GenericMachine\nvm\ex3b.asm

Memory Cell	Instruction	Object Code
0000	ld hl, #05e7	21
0001		
0002		
0003	ld de, #0242	11
0004		
0005		
0006	ld a, l	7d
0007	add a, e	83
0008	ld c, a	4f
0009	ld a, h	7c
000a	add a, d	82
000b	ld b, a	47
000c	halt	76
000d		
000e		
000f		
0010		
0011		
0012		
0013		
0014		
0015		
0016		
0017		
0018		

Reg	A	F	B	C	D	E	H	L	I(X)	X	I(Y)	Y	S	P	P	C	IR
Dec	7	0	7	41	2	66	5	231	0	0	0	0	0	0	0	12	0
Hex	07	00	07	29	02	42	05	e7	00	00	00	00	00	00	00	0c	00

Flags	Sign	Zero	-	H	-	P/V	N	Carry
State	0	0	0	0	0	1	0	0

0000h	21	e7	05	11	42	02	7d	83
0008h	4f	7c	82	47	76	00	00	00

Program Counter: 0000

Stack Watch

Address	Data
04	42
03	11
02	05
01	e7
00	21

Step every = 40

Run until PC = 0000

IO Ports / IO Control

Incoming data: 00

Outgoing data: 00

```
Loaded OK
Loading program
Loaded OK
Loading program
Loaded OK
```

MiniEditor D:\GenericMachine\nvm\ex3b.asm

```
; Askhsh 3b - sel 323
ld hl,0x05e7
ld de,0x0242
ld a,l
add a,e
ld c,a
ld a,h
add a,d
ld b,a
halt
```

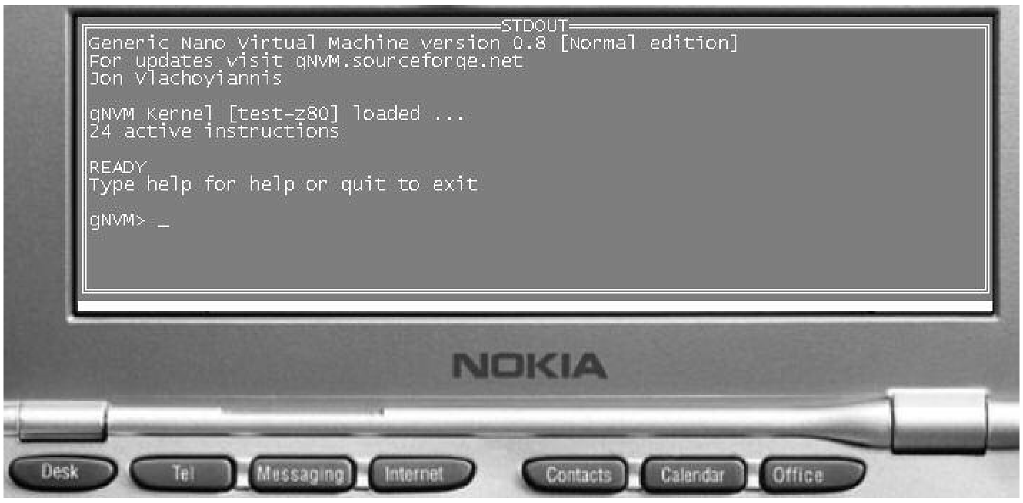
Ph.1 : gNVM emulating a z80 processor in MS-Windows GUI.

Moreover, our work could be adopted for the development of an Operating System (OS). By providing a high-level API through the Extended VM API, students can create OSes without the hassle of dealing with the underlying architecture and focus on their OS development.

5. EVALUATION

In order to evaluate our work, we created a working prototype of a z80 processor, a Windows GUI that can be used for educational purposes and a console-based GUI for Linux and Symbian Operating systems.

To allow for the emulation of other processors, the NVM file needs to be processed. Based on the Z80 nvm file, we created a VM that has a Greek instruction table and accepts interrupts from POP3 servers. The only development tool we used for emulating the processors and changing the instruction table was the emacs text editor [12]. No new API needed to be introduced due to processors' similarity. Furthermore, as a proof-of-concept, we created PUSH instructions using macro instructions and DELETE FILE instructions, showing macro and Extensive API. Demonstrating gNVM's portability, a port to Symbian platform was made and gNVM was executed successfully by a mobile device, as shown in Photo 2.


```
STDOUT
Generic Nano Virtual Machine version 0.8 [Normal edition]
For updates visit gNVM.sourceforge.net
Jon Vlachoyiannis

gNVM Kernel [test-z80] loaded ...
24 active instructions

READY
Type help for help or quit to exit

gNVM> _
```

Ph. 2. gNVM running on a Symbian-based mobile device (Nokia 9210)

Finally, a crude proof-of-concept Java VM was emulated as evidence that even stack-based VMs can be emulated.

6. CONCLUSIONS

gNVM can be used for 8-bit CPU emulation, Universal Turing Machines, Graphic Engine Machines and in general for a great variety of VMs depending on the quality of the extended functions. The open structure of gNVM, provides us with the ability to build scalable systems, easily maintainable and portable, without the requirement of advanced programming skills. As a proof-of-concept, an emulation of a z80 processor is provided along with its graphical user interface for MS-Windows platform. Changing instructions or implementing new ones is achievable by changing a human readable text file. As an example, we can convert the instruction set (all or part of it) to another national language (Greek for example) or to another Assembly language (8085). Moreover, gNVM is already ported to other platforms such as Solaris, Linux, MS-DOS and Symbian where a console interface is provided.

By using gNVM, we can create new emulators and Virtual machines in a short period of time, with many features and (more importantly) without a noticeable performance cost, due to the fact that gNVM is written in ANSI-C and the usage of call threading method for dispatching code. The structure of gNVM, ensures that developers will focus on implementing new ideas and experimenting with new features, instead of writing and debugging code.

7. REFERENCES

[1] R.P. Goldberg. ”Survey of virtual machine research”. IEEE Computer Magazin*e*, 7(6), 1974.

[2] L. Seawright and R. MacKinnon ”VM/370 - a study of multiplicity and usefulness”. IBM Systems Journal, pages 4-17, 1979.

[3] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg and John Waldron "The Case for Virtual Register Machines". Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators, Available at: http://www.acm.com

[4] J. R. Bell. "Threaded code" . Communications of the ACM, 1973.

[5] Xen , The Virtual Machine Monitor , Available at : http://www.cl.cam.ac.uk/Research/SRG/netos/xen/

[6] SUN's JAVA VM, Available at : http://java.sun.com/

[7] .NET Virtual Machine, Available at : http://www.microsoft.com/net/default.mspx

[8] Parrot Register-based VM, Available at : http://www.parrotcode.org/

[9] Quake 3 VM, Available at : http://www.idsoftware.com/business/techdownloads/

[10] Donald Knuth MMIX 2009, Available at : http://www-cs-faculty.stanford.edu/~uno/mmix.html

[11] GNU MDK, Available at : http://www.gnu.org/software/mdk/mdk.html

[12] GNU Emacs, Available at: http://www.gnu.org/software/emacs/

[13] Gameboy, Available at: http://www.nintendo.com/channel/gba