Embedding Microcode in Sourcecode of the Programming Language O

Thomas Pucklitzsch, Mathias Sporer, Anja Pucklitzsch
Department of Computer Science
Duale Hochschule Sachsen
Glauchau, Germany
e-mail: thomas.pucklitzsch@dhsn.de

***Abstract*—This paper describes capability of the new programming language O, which is still under development, to embed the microcode of the underlying Octopus machine within the O-source code. The challenge is to insert a 36 bit micro-instruction into a 32 bit register and copy it into the microprogram memory to the correct location. This technology enables time critical software components to be executed up to ten times faster compared to the implementation in machine code. The paper illustrates this effect with a simple example and examines the parameters that influence the acceleration.**

Keywords-compiler; microcode; programming language; performance; Octopus

I. Introduction

Computer hardware provides gates for the execution of simple arithmetics and logic functions. The Arithmetic Logic unit (ALU) is able to add two binary numbers or compute a logic operation for every binary digit and store the result into a register. To provide more complex operations, basic hardware operations are connected, resulting in so called microprograms. The microprograms are stored in the microprogram memory and contain a sequence of data words that directly control the hardware. These sequences will be activated by the control unit of the computer. The control unit reads a machine-instruction from the memory and executes the related microprogram. The collection of every available OP-code (operation code) is called "the instruction set" of the computer and this instruction set acts as an interface between programmer and hardware. A change or extension of the microprogram results in a modification of the instruction set of the computer. However, most architectures prevent the modification of the microprogram by the programmer for security reasons by various security technologies. [1] The microcode is located on a very low level of a computer. Programming languages are used on a significant higher level. The programming language O is an unfinished language that can be modified and easily adapted to different instruction set architectures. The motivation of this paper is to present the capability of a dynamic modification of the microcode as part of the programming language O. After presenting related work, the Octopus-machine is introduced. The following sections compare the performance of using user-defined microcode versus machine code. The article concludes with an explaination of how microinstructions are embedded in O-code and how the Ossambler layer and the the Octopus-machine hardware function.

II. Related Work

In [1] a library was introduced that enables the user to change the microcode of a Intel CPU. As described in the paper, working around several security mechanisms is necessary to make the modification possible. The aim of this modification is to increase the performance of time critical operations. The hardware developer tries to prevent changes in the microprogram. To find a way around the security mechanisms and change the microprogram is very hard and a research project of its own. For example, finding the sequence of micro-commands in a x86 architecture is very hard, because micro-commands are not stored in the microprogram memory as a consectutive sequence, but the addresses are mapped with a special permutation algorithm. Without reengineering the algorithm, it is impossible to find the address of the next micro-command. In [2] a framework is introduced, which is able to analyze and patch Intel microcode. For example, an implementation of fast software breakpoints in microcode runs 1000 times faster than the solution in Assembler. This shows the potential of customized microcode.

III. The Octopus-machine

In comparison to Intel CPUs, the Octopus-machine [3] has a very simple computer architecture. It is leaning on a part of the Java virtual machine and implements a stack machine. It was first developed for educational purposes. There are implementations in Python and C, already, but very soon a soft-core for the FPGA Tung Nano 20k will be available, too. The machine was build with the purpose of easy explaining a simple HW architecture that is also able to do some useful things for students. With the Octopus software it is easy to implement your own microcode and generate an assembler for this code that we call Ossembler. Figure 1 shows a screenshot oft the Octopus software. With the help of this Ossembler it is easily possible to write code for the customized Octopus machine, and the Ossembler inserts the correct OP-codes and operates and computes all addresses. In addition to the possibility to create customized microcode, the Octopus-machine supports dynamic import of microcode. Parts of the microprogram memory are writeable through a range of addresses.

IV. Microcode vs. Machine commands

To explain the advantages of inserting customized microcode dynamically, we use a simple example. In this example we compute the Fibonacci numbers. The Fibonacci series

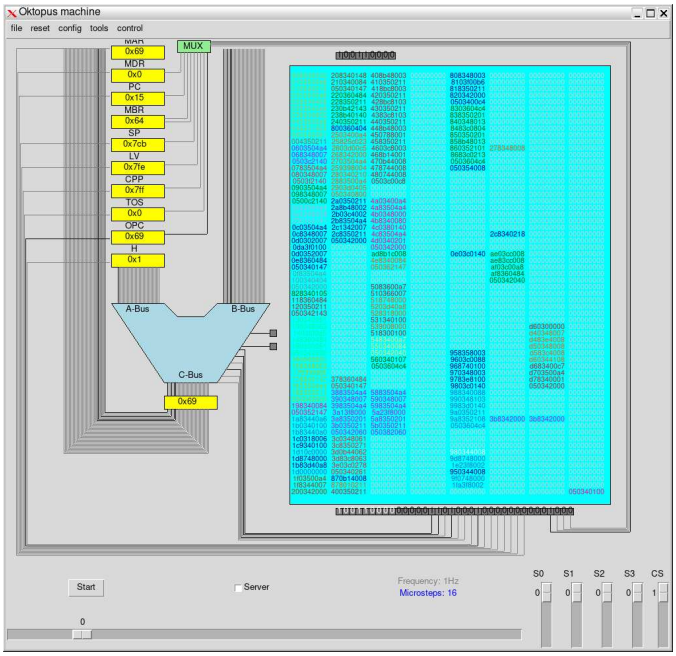

Figure 1. Octopus software

start with two ones and the next number is defined as sum of the two preceding numbers. It is not necessary to print the results because we can see the numbers appearing in the registers and the memory. To compute Fibonacci numbers, we have to add the last two numbers. A simple ossembler code is given that adds two numbers and stores the new Fibonacci number at the address of the first number.

```
oload  var2
odup
odup
:loop
oadd
odup
odup
ostore  var2
oload  var1
oadd
odup
odup
ostore  var1
oload  var2
ogoto  loop
```

Listing 1. compute Fibonacci numbers in ossembly language

Listing 1 starts with the command `oload` that reads a variable and puts it on the stack. The command `odup` doubles the top of the stack, `oadd` adds the two to elements of the stack and `ostore` saves the value on top of the stack in the given variables. The `ogoto` command jumps back to the label `:loop`. Assembly language uses so-called mnemonics insted of the operations codes which are easier for humans to handle than hexadecimal numbers. The ossembler transforms every

TABLE I. NUMBER OF CYCLES PER MICROCOMMAND

command	cycles	occurerence in loop
odup	2	4
oload	6	2
oadd	3	2
ostore	7	2
ogoto	6	1

mnemonic in an operation code that contains the address of a microprogram inside the microprogram memory that is called from the main routine. The Octopus-machine implements a typical complex instruction set architecture. That means every hardware instruction contains a different number of micro-commands. Every microcommand needs a cycle. The measure of computational cost is the number of cycles for one loop iteration. In table 1 you can see the cycles needed for every necessary kind of command and the number of occurrences in the loop cycle. With this information we can compute the number of cycles for one loop iteration. Notice that the main routine needs one cycle to call the next microprogram. With the values from table 1 and the listing we can find out that the algorithm needs 46 cycles for one loop iteration. Now we can compare the coputational costs with a microprogram that is computing the same function. In comparison with the ossembly language, a micro-program for the Octopus-machine looks like this:

```
1. H=MDR=1, wr
2. MAR=MDR=PC=1
3. MDR=MDR+H, wr
4. PC=MAR=PC+1
5. OPC=MDR+H
6. H=MDR
7. MDR=OPC, wr, goto 3
```

Listing 2. compute Fibbonacci numbers with Ossambler

The micro-commands are given in hex-code. Therefore, we show the code in a specific notation to make it better readable. The strings are registers, operations are functions of the Arithemetic Logic Unit and wr is the signal for the memory to write the content of the register MDR to the address given in the register MAR. Because every micro command contains the address of the next micro command, an additional cycle for the loop is not needed. As the Reader can see, the loop needs exactly 5 cycles for one iteration even though the algorithms are more sophisticated than the Ossembler version. As you can see the microprogram version runs nearly ten times faster than the version with machinecode. The main reason for the speedup is that results do not need to be saved to the memory between the operations. The registers hold the results until the next operation. This is the main potential of optimization.

V. MICROPROGRAM MEMORY ORGANIZATION

For inserting the microcode, the address ranges from 192 to 255 and from 448 to 511 are reserved. Figure 2 shows the microprogram memory with the two empty columns for inserting user-defined micro instuctions. The microcode allows

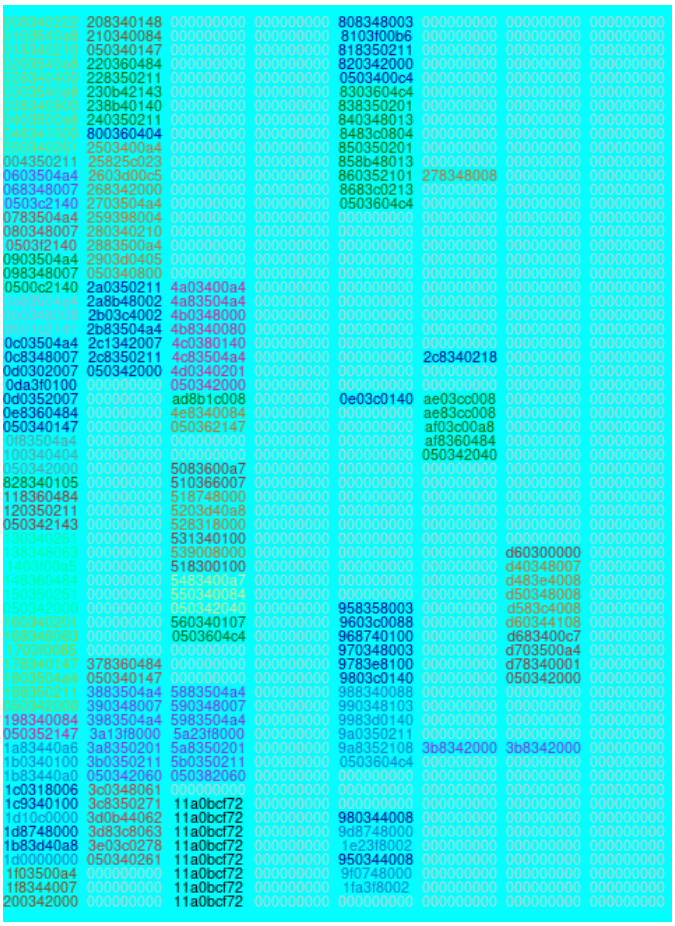

Figure 2. microprogram memory

a conditional branch. To do this, the most significant bit of the address will be inverted. This means the address of the next micro command is nextadress +/- 256. This is the reason for the two address ranges that allow a conditional branch. To write a micro command the addresses can be accessed through 0 to 127. If you write into address 0, the micro command appears in address 192 and if you write in address 64 the micro command appears in address 448. To write into the micro command, memory addresses starting from 2^{15} are being used. The problem is that micro commands for the Octopus machine need a 36 bit bus, but the memory supports 32 bit. 36 bit are needed because every micro command contains the address of the next command. To insert a micro command, two addresses must be provided. First the address where the micro command is located and second the address of the following micro command. Figure 3 shows the structure of the addresses. Because of the address ranges for dynamic micro-commands, only 128 addresses are needed. This is also useful for security purposes. It is not possible to overwrite existing microcode. Only the 128 reserved addresses are accessible. The bus has only 32 bit. The two 7 bit micro addresses are encoded into the memory address. 3 To write a micro command on address 200 in the microprogram memory with the next command on

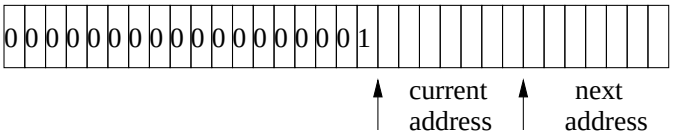

Figure 3. using memory addresses to encode two micro addresses

500, the address is computable $2^{15} + 8 + 116 * 2^7$.

VI. Integration in O and Ossambler

The predefined microcode supports a simple microprogram that takes six bytes and writes a micro command into the microprogram memory. The first four bytes contain the micro command, and the last two contain the actual address of the micro command and the address of the next micro command. The mnemonic of this microprogram is `omic`. To define a mnemonic for the new microprogram a define statement is used in Ossembler. The `define` statement needs three parameters. The mnemonic of the micro program, the address and the number of bytes the microprogram takes from the byte stream. The programming language O is a new language under construction. It is based on Extendable Makeup Language (XML) and allows a recursive structure and self explaining tags. O supports extensions. Depending on the running microcode, new tags that are keywords of O can be defined. The definition of tags can be done in the O language definition. In this definition, every tag is defined and a sequence of Ossambler commands is given. To insert microcode dynamically, a tag called `add_ microcommand` is needed. In this tag every flag of the micro command is defined with its own tag. The Reader can see the definition of a micro command in the following example.

```
<add_microcommand>
<name>new_microcommand</name>
<b-bus>MDR</b-bus>
<c-bus>TOS<c-bus>
<inc>0</inc>
<inva>1</inva>
<enb>1</enb>
<ena>1</ena>
<op>add</op>
<sra1>0</sra1>
<slla8>0</slla8>
<inc>0</inc>
<jmpn>0</jumpz>
<jmpz>1</jumpn>
<jmpc>0</jmpc>
<addr>62</addr>
<naddr></naddr>
<side>right</side>
<nside>left</nside>
</add_microcommand>
```

Listing 3. example O definition

The meaning of the different tags refers to the identical flags of the Octopus machine, which refers to the stack machine described in [4], with exception of the tags <addr>, <naddr>, <side>, <nside> and <name>. These tags provide the address where the micro-command should be stored and the address of the following micro-command. The address can be choosen from the range of 0 to 63. With the tags <side> and <nside> the row of the actual micro-command and the next micro-command can be choosen. The <name> tag defines the name of the micro-command. This name is used to call the microprogram. The actual address of the current micro-command of the example is 126 and the next address is 63. With the knowledge of the architecture of the Octopus machine it is very easy to describe the behavior of the Octopus machine in the language O.

VII. Conclusion

The Octopus machine is a simple stack-machine that allows the user to initialize the machine with an individual microprogram. It is also possible to load additional micro commands at runtime and extend the instruction set of the machine. This is possible by writing a word to the addresses of a special address range in the memory. A predefined microprogram called `omic` realizes the import of dynamic microprograms. The new programming language O is a XML-based language which is extendable. With this language it is easily possible to describe micro commands which are passed to the Ossambler for inserting into a reserved area of the microprogram memory. The possibility to define custom microcode inside the source code of the programming language O is useful for performance improving. The given example shows a speedup of factor 10, but depending on the use case the benefit can be much higher as the software breakpoints in [2] show. This means that the Octopus machine can compute many problems much faster than comparable competitors without such capabilities. Now the Octopus machine is not available as hardware, but a softcore for the use with an FPGA is under construction. The presented method makes it very easy to adapt the hardware to a specific problem in order to achieve better performance without changing the circuit.

References

[1] B. Kollenda et al., “An exploratory analysis of microcode as a building block for system defenses,” 2020.

[2] P. Borrello, C. Easdon, M. Schwarzl, R. Czerny, and M. Schwarz, “Customprocessingunit: Reverse engineering and customization of intel microcode,” 2023.

[3] T. Pucklitzsch and M. Sporer, *O - a new approach for a very simple language for distributed computation*, 2025.

[4] A. S. Tanenbaum and T. Austin, *Computerarchitektur - von der digitalen logik zum parallelrechner*, 2014.