A Methodology for Testing the Microprocessor Core of a System on Chip with a x86-Compatible Microprocessor II

In memory of prof. Koldaev V. D.

Evgeniy Saksonov
Moscow Technical University
of Communications and Informatics
Moscow, Russia
saksmiem@mail.ru

Mikhail Dyabin
"Kaskad" Ltd.
Moscow, Russia
dyabin@mail.ru

Artyom Reshetnikov
"Accord" Ltd.
Moscow, Russia
a_reshetnikov@hush.com

***Abstract*—This paper continues to discuss the problem of functional testing of embedded microprocessors, started by the authors in one of their previously published articles. Special attention is paid to writing tests for logical and arithmetic instructions. The main principles, adopted by the authors for development of testing software for the system-on-a-chip called "Kaskad-1", are given.**

I. Introduction

While testing an embedded system, the requirements for its testing software depend on the requirements for the embedded system's reliability. In the case when a microprocessor is intended to use as part of a mission-critical system, i.e. such a system that its failure may result in serious consequences, it is necessary to use well elaborated methods for testing, based on deep theoretical justifications. But it is often the case when an embedded system's processor failure does not result in any dangerous implications; for such systems some difficult procedures, like formal verification of their processor's core, are unnecessary.

The methodology for testing microprocessors, presented by the authors in [1], is focused on writing tests quickly, concurrently with the development of the hardware being tested. A separate section of the article was dedicated to possible pitfalls with the proposed methods' implementation. In this paper we review [1] and continue discussion on the problem of functional testing of embedded microprocessors. Nowadays this topic is quite well studied and is actively evolving due to constant increase of microprocessor units' difficulty and constant expansion of use of embedded systems. We would like to mention the article [2], giving a detailed explanation of the theoretical basis for testing electrical circuits, the article [3], where a general methodology of testing microprocessors is presented, and the publication [4] as an example of a paper with a description of formal verification procedures for an exact processor (namely, ARM). Much more papers are actually published on similar themes.

Speaking about confirmation in practice of the methodology suggested in [1] and in this article, we consider the system-on-a-chip called "Kaskad-1", whose testing software was developed by the authors in accordance with the main principles described below. The main purpose of the SoC was *to provide the possibility of data transmission through a high-bandwidth radio channel using a built-in OFDM modem* [5]. This system uses an internally designed domestic x86-compatible microprocessor as a network processor unit, which was constantly upgraded along with strengthening the requirements to the data channel, provided by the system. While the processor's clock rate was being increasing, and while its instruction pipeline was becoming more mature, new versions of the device had to be tested often. Hence, it was natural to improve testing software in order to speed up the overall testing process and simplify the task of error localization.

II. The main principles of testing x86-compatible microprocessor's instruction set

In this section we recall the main ideas, which were discussed in detail in [1], adopted by the authors for development of testing software for the system-on-a-chip called "Kaskad-1". The readers who are familiar with [1] may skip this fragment and continue reading the next section of this article.

A. The general principles of testing built-in systems'-on-a-chip microprocessors

While working on testing the system-on-a-chip called "Kaskad-1", the authors wrote their programs in accordance with the following principle:

Rule 1: *before starting to write any tests, it is necessary to analyze the tested system and to get a sense of which vulnerabilities threaten the system and how the tests, being under development, can protect this system from the detected vulnerabilities.*

Let us give a brief explanation to this rule. Which weaknesses does the tested system have? What kind of errors can theoretically occur in this system? What errors are the

ISSN 2305-7254

most probable? Which errors lead to the most harmful consequences? We think that the questions mentioned above and similar questions must be under consideration before the process of writing tests for the tested system starts. Moreover, we assert *while writing tests, it is necessary to regularly come back to the questions mentioned above from time to time* along with any changes in the tested system are made.

In the other words, we assert tests for a system must be developed alongside with the process of its evolution. First and foremost, the most vulnerable feature of a system must be the feature that is being tested at the current moment.

Let us now look at the SoC "Kaskad-1" (see fig. 1) from this point of view (one of the most modern versions of this system is described in the paper [6]). In the projects, where this system is really necessary and used, the first of all requirements for its microprocessor is to correctly interact with the system's built-in OFDM-modem. To *transfer* data is what any programs, really executed by this microprocessor, do most of the time; while *processing* data is what other processor units do in the final system (not the processor unit which is about to be tested). Data is transmitted through DMA channels between the tested processor and the modem in "Kaskad-1"; at the same time data can be transmitted from many peripheral devices through other DMA channels. The main task of the appropriate software is to coordinate the work of all peripheral devices correctly.

We notice that such programs, which are executed on "Kaskad-1" in practice, usually use a small restricted set of processor instructions; although, these programs may face to different unexpected situations, while interacting with devices, and all of them must be processed by the software in the right way. Our final aim is to make the system run these programs correctly – we should always keep that in mind while writing tests.

Hence, while doing the job of functional testing of the core of the processor of the system "Kaskad-1", special attention should be paid *not* at separate microprocessor instructions, but at processor's work in general under condition of intensive data exchange between different devices, built in the SoC. Tests for the microprocessor "Kaskad-1" must cover as much as possible of different situations, which may arise in real programs, run in practice, which interact with more than one peripheral devices of the system.

It is clear that such tests must complete not a single iteration, but many of them, because the errors of data exchange between devices may occur after a few amount of time, due to very specific conditions of their appearance. And now this brings us to another principle, which may not seem evident to people who are new in writing tests.

We shall notice that after an error in a processor core is detected, the error must be *localized*, which means for the program, which produces an error, it is necessary to simplify its source code in such a way that it will be possible to point at the exact lines of the code where microprocessor instructions evidently produce incorrect result.

Strictly speaking, to localize a microprocessor error means to make its testing program so simple that it would be possible to run it inside a virtual environment, capable of modeling the logic of the circuit containing the tested processor; and after looking at the waveform after modeling it would be possible to see incorrect processor's behaviour clearly, at an exact moment of time on the chart.

A localized testing program must become clear enough for being sure the error definitely occurs in the tested microprocessor, not the error is hardcoded into the program itself. But how to reach such a clarity? This question goes beyond the scope of this paper. We just notice that if the source code of a testing program becomes too complicated, the process of error localization in such a code would be very difficult task.

So, while working with a testing program in order to simplify its code, since it is very easy to put there a new mistake of completely software nature, we offer to take into account the following rule:

Rule 2: *testing programs should be written in such a way that when using them, the process of error localization, in case of any errors become detected, would be not very difficult, if possible, but, at the same time, while doing the job of simplification of source code of these programs, a risk of making new software mistakes would stay low.*

B. Details on testing x86-compatible microprocessor's instruction set

Certainly, while testing a processor core, it is of critical importance to check separate instructions of the microprocessor. Good tests must cover all microprocessor's instructions and all memory address modes by the values of input data sets. For any instruction, in addition to input data from the set of "the most frequent cases in practice", tests must include all theoretically correct "boundary cases", which may be seldom, but possible enough be met in command arguments.

In particular, if an exact command modifies microprocessor's flags, then for each of the flags, affected by the tested command, it is necessary to make sure, that the command really modifies the flag in the correct way. And if a command analyzes data located at the memory space (like, for example, the x86's instruction "Leave", added to the architecture starting from the processors Intel 80186 and Intel 80188), then for the data, analyzed by the command, it is also necessary to cover by tests as much of fundamentally different variants, as it is possible.

The main difficulty, which someone meets while testing instruction set of any microprocessor, is *sometimes it is not clear enough, which input data sets should consider as so different that both of these data sets should be included into the test being under development, and which of input data sets are, vice versa, so similar under consideration that if the tested instruction processes correctly one of these data sets, then most probably the same instruction will process correctly the other one data set, similar to the first one.*

In practice it is not possible to test instructions by brute force, i.e. to cover all of the possible parameter values by tests. But in the most cases it is not necessary to take into account literally all of the possible variants. For example, let we have a task to check an instruction which transfers a value from a microprocessor register into another register; such an action is made by the instruction "Mov" in the architecture "x86". The question is: which input data sets are reasonable for including

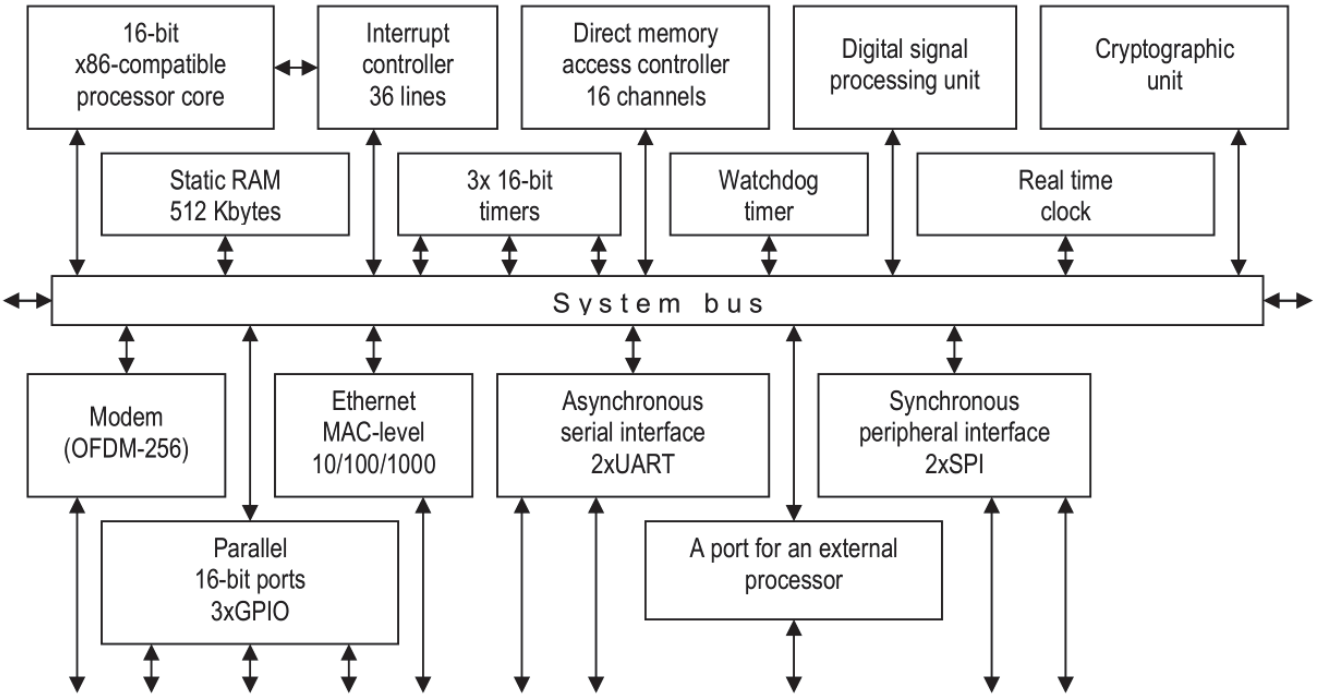

Fig. 1. A block diagram of the SoC "Kaskad-1"

into a test, for, in case the final test runs successfully on a sample of a tested device, someone may think data transferring between registers of this exact sample device works correctly, with high probability?

Remember we are *not* trying to solve the task of formal verification of a processor core. Vice versa, in this example such a program is required to be written, that it would be possible, firstly, to debug microprocessor circuit at the stage of its development by using the program, and secondly, it would be possible to verify samples of the tested microprocessor after its final devices become manufactured, by running the same program.

Which cycles of the testing program should be made more verbose? Which of the cycles can be, vice versa, shortened because of the parameter values, not covered by the final cycle, most probably will not produce any new errors (not being detected by other parameter values, really included into the cycle)? While developing functional tests someone constantly face to similar questions.

In case of testing an instruction set it is necessary to constantly make choice among different parameters, which actually affect the result of a tested instruction: more importance should be given to some of the parameters, by the cost of refusing to check what seems less important; in the other words, some sets of input data have to be sacrificed for some other sets to be included in a test. General recommendations on how it is possible to make such choices were given above, in the *subsection A*. Now it is time to speak in detail about exact principles, adopted by the authors while developing testing software for an exact system-on-a-chip called "Kaskad-1".

Well, say we need to develop a test for the "Mov" instruction of an x86-compatible microprocessor for the case when data is transferred from a register into another register. The task of composing this test can be divided into a bunch of small subtasks.

The first thing we need to check is whether data are transmitted into that register which is selected by the given arguments of the command "Mov". *We may think a fragment of code for solving this problem tests not the instruction "Mov" itself, but the general mechanism of memory addressing*. To check addressing mechanism is a separate task. This task is nowadays quite well studied; a few words on it are given in section IV of the article [1]. The only thing we notice is, depending on different factors (such as the maximum allowed estimated time for testing software to run, the size limits of a testing program, memory limits etc.), either the decision may be taken to test memory addressing using a wide class of microprocessor instructions, which also includes the instruction "Mov" as one of its elements, or it is possible to develop a separate test, focused on memory addressing mechanism, where just a strongly restricted set of microprocessor instructions is used. In the last case someone assumes: if such a test accomplishes its job on a sample of the tested device, most probably the memory addressing mechanism works correctly on the tested sample for any processor instructions, including the instructions which were not used explicitly while running the test.

Next, it is necessary to check whether in case of data transmission from a source register into a destination register, every time when a processor finishes execution of the corresponding instruction, the destination register contains exactly the same

value which was in the source register at the moment when the processor started to transfer data. Common sense dictates that, if a value has been successfully transferred from a source register into a destination register, then any other values will also be transferred correctly from the same source register into the same destination register. Hence, when the value in a destination register is checked after transferring data from a source register, the only important thing to verify is whether the data has been actually transmitted, i.e. it should not happen in the testing program that before processor starts to transmit data a destination register already keeps the same value which is also kept in the source register.

In fact, it is necessary to test *every bit*, whether it is transferred from a register to a register correctly: the reason is while manufacturing samples of a device, the errors may occur which result in incorrect setting of separate bits of some hardware variables used while running particular instructions of a microprocessor. If the task were not to test a hardware implementation of a device, but, for example, its implementation as an emulator, it would be clear in this case that testing all of the bits would become, more probably, unnecessary. Since our task is to test a hardware implementation, a good enough test should cover, by input data sets, all of the bits which may affect the result of the tested instruction's execution.

What is the simplest way to solve the problem mentioned above? The general rule, adopted by the authors while testing the system "Kaskad-1", is:

Rule 3: *whatever instruction is tested, for all of its parameters, regardless of the values of the other parameters of the same instruction, its test (if only it is possible at all) must contain the values 0x00, 0x55, 0xAA and 0xFF – in case of the parameter has the size of 1 byte, or the values 0x0000, 0x5555, 0xAAAA and 0xFFFF – in case of the size of this parameter equals 2 bytes*.

Here, speaking about command parameters, we use the term "parameter" in the most wide sense: we mean not only the arguments passed to the command directly, but also the values of any variables able in principle to affect the result of command's execution: they are processor flags, some implicitly used memory cells, the instruction's offset in the memory space etc.

Of course, while testing an instruction like "Leave", it is not possible to follow the rule, given above, literally. There are some other examples of commands when the number of parameters, capable to change their result, is so huge that, even if we restrict brute force with this rule, we obtain so much amount of calculations, that after running such a test it would be impossible to wait for it to be finished. For these cases the authors offer not to refuse the rule completely, but to take it into account and follow it within reasonable limits: the rule still allows to iterate values for some parameters.

Of course, while testing arithmetic instructions some more values must be included in a test. Here the authors suggest the following:

Rule 4: *regardless of whether an arithmetic instruction's arguments are signed or unsigned, its test must contain either the values 0x00, 0x01, 0x7F, 0x80, 0x81 and 0xFF – in case of the size of the parameter to iterate equals 1 byte, or the values 0x0000, 0x0001, 0x7FFF, 0x8000, 0x8001 and 0xFFFF – in case of this parameter has the size of 2 bytes*.

These values are "bound" for key ranges, that's why it is a good idea to use them for testing instructions as widely as it is possible. But it is important to get in mind some instructions have some more "bound" values, specific to these instructions; all of them also must be included in tests, when possible. Continuing further considerations, we would get into a well-elaborated theory of composing tests, leaving the scope of this paper.

Let us return to the task of developing a test for the instruction "Mov". There is one more block of variables whose values also must be checked before coming to a conclusion that data transferring from register to register works correctly. Here we are speaking about the flags of a microprocessor. The instruction "Mov" must leave all of the processor flags untouched [7, application C]. At least, for all of the general-purpose flags (a better way – for all microprocessor flags) it is necessary to check that if, before running a command of data transferring, a flag was set to 1, after running the command the flag is still set to 1, and if, before running the same command, this flag was set to 0, also: after data transferring is completed, the state of the flag remains unchanged. All of the listed cases must be included in a test for the "Mov" instruction at least one time.

Only when all three kinds of check (addressing, values, flags) are done by a testing program on a sample of the tested device, we may conclude that the data transferring instruction itself is, most probably, implemented without any errors for the case of sending data from a register into a register on the tested sample of the tested device. But such a conclusion does not mean that sending a value from register to register will *always* be completed successfully by the tested instruction: if the tested hardware is not stable, it is quite possible, because of different reasons, for such a transfer to fail in some specific rare situations, which sometimes become difficult to localize. Passing our test will only mean that if transferring a value from a register into another register is done incorrectly, the reason of the error most probably lays not in the tested instruction's implementation, but somewhere inside other units of the processor's core.

Let us now make the final remark:

Rule 5: *while writing tests for an instruction set, it is critically important for the result of any step of the testing program to be completely determined, according to a document, where the tested system is described officially*.

For example, by reading the application C of user's manual [7] of Intel 80C186EC, we notice that the instruction "Test" after being executed affects some microprocessor flags in the following way:

- there are some flags whose values are changed depending on the result of the command;
- some flags leave their state unchanged;
- the flag "AF" becomes not defined after running the instruction "Test".

So, if a program tests the instruction "Test" on a microprocessor, whose architecture is x86-compatible, at the stage of

checking the values of flags the program may either verify processor's flags separately (this way is long, hence unreasonable), or the program may put a definite value into the flag "AF", and when the state of all flags becomes determined, it is possible for the program to verify them all at one time.

III. A FEW REMARKS ON WRITING A BOOTLOADER FOR RUNNING TESTING SOFTWARE

When a tested microprocessor is emulated on an FPGA, it is possible to burn software into ROM every time it is necessary to run it. However, when a system-on-a-chip is manufactured, it may be problematically to burn ROM every time when it is necessary to run a new testing program. To prepare a complicated program, containing a bit test, and to burn it into ROM, is a bad idea since while manufacturing a chip, some errors may occur which does not allow a program to start – this moment is already mentioned in the previous section. A bootloader, burnt into ROM, should be as easy as possible. But what is "easy"? Speaking about this question, let us consider a bootloader used by the authors in the system-on-a-chip "Kaskad-1" and another bootloader of an experimental system, where the previous loader failed to run; we discuss the reasons and show how it was possible to solve the problem.

The bootloader of "Kaskad-1" starts its job with forced minimum hardware initialization. After this, the program checks whether a Flash-memory contains a program available for running. If such a program is found, the bootloader just copies the program into memory and passes the control to the entry point of the program. This way is the main form of using "Kaskad-1" after a chip is manufactured: there is a fixed bootloader, and there is a program in a Flash-memory. In any case, if there is no program in the Flash-memory, the bootloader initializes a serial port, which also is included into the system as a part, and waits for commands which can be sent from, for example, a desktop computer: after connecting a serial cable to both UARTs, obviously.

From this moment, different ways were tried by the authors while testing different emulated circuits. There is the protocol called XMODEM which is well-known and which has some extensions like the protocol YMODEM. The main advantages of XMODEM is its simplicity and abundance of clients for desktop systems, so if you choose XMODEM to be a bootloader of your embedded system, all you need is to implement its embedded part, while a part to run on a desktop is already written, user may choose anyone, maybe this way is the most rapid for getting ready a system. One disadvantage of the protocol must be taken into account: all it can is transfer raw data. If you want to run a program, for example, you have to prepare it in a fixed format, as soon as the bootloader of your system expect data to be sent by XMODEM. You can not change your program's entry point, no way to just write data into memory without jumping into the buffer in attempt to run data as a program. Problems may occur if your system's RAM is not stable. Also, sometimes there is no need to run a program on an embedded system to test some of its part: if its bootloader supports more commands, then just receiving a program and running it, it is possible to test some parts of the system sending commands to bootloader and asking it to send back their result.

Such is the bootloader actually used in the system-on-a-chip "Kaskad-1". This bootloader understands the commands like write a word into a port (the memory space and the input/output port space are separate address spaces in the architecture "x86"), read a byte from port, write array into memory and read array from memory etc. Data is loaded there by using on of such commands. You may load a huge program by sending a sequence of commands to the loader. After the whole data is loaded into memory, the last command is sent to the bootloader – which is, obviously, the command of passing the control – and you program starts working. Since the bootloader initializes a serial port, your program may use it as standard input and output default streams. All commands to bootloader contain two bytes: the first byte is a command prefix, while the second byte is the code of the command to execute. If a command requires parameters, they are given later, according to format of an exact command. One other nice feature of the bootloader is the possibility to calculate a simple checksum of a given region of memory. So, if you need to check whether your program is loaded correctly, before passing the control to the program, you don't need to read back all data.

Another feature of the bootloader of "Kaskad-1" is the possibility to work without any RAM at all. All of the commands (except the command of passing the control: this can be done only through a stack in x86) are executed without saving any data in RAM memory, i.e. only processor registers are used, until a direct command is given to the bootloader where interaction with memory is intended. A traditional way of calling to subroutines is using the command "Call", which saves return address on top of the stack before passing the control at the address pointed by an argument. This method is not used in the bootloader of "Kaskad-1": the register BP is reserved here for saving the current return address. So subroutines are called here without using any stack cells.

An important moment is initialize only the hardware which is really necessary. In case some peripheral devices become damaged in a sample of the device after manufacturing, it will be still possible to use the other parts of the device and to test what has gone wrong. The code of the bootloader also contributes to the idea of getting ready as fast as it is possible – for investigating of possible errors in case something gets wrong with hardware. By the way, from the description of commands given above it is easy to see that writing desktop support for such a bootloader is a quite trivial task. At least, this task stays trivial until the desktop software stays platform dependent. In POSIX operating systems you don't even need to write programs: to write some easy scripts will be enough, since the most difficult part of such a work is to initialize a serial port in the correct way.

Once it was necessary to test an experimental system (not x86-compatible) where a lot of its hardware contain multiple errors. Not only its microprocessor had bugs, but also there were bugs in memory, even its serial port's implementation contained a bug at te beginning of work. When a bootloader was written analogous to the bootloader of "Kaskad-1", nothing could be done with the system, because nothing was working, and there was no idea in which units bugs were localized.

Another bootloader command system was composed with

the purpose to get control over every single byte of transmitted data. It was clear the UART channel was not reliable, so some bytes were lost while sending them. The command system took this into account and offered the possibility to undo every command in case its result, which was sent to UART automatically, was not accepted by a desktop client.

Let us talk about this bootloader in more detail. It has two abstract variables: one of them is called *data register*, another one is an address register. Of course, it was comfortable to use processor registers to hold values of these variables – so their names were born. After running any command of this new bootloader, the least byte of the data register is sent back automatically in order a desktop client can check its value. In case the value received by UART is incorrect, it is possible to send a command to undo, as it was mentioned above – so, two more registers have to be reserved to backup main values. Since it was not clear which processor instructions worked well and which of them did not work, a lot of easy commands were included into the bootloader's command set: different boolean operators, some primitive mathematical operators, shifts. There is a command to send the value of the data register into the address register. Some commands for reading from memory and writing into memory use the contents of the address register as the memory address of the cell to operate on. 16 fast commands shift left the contents of the data register and write hexadecimal digits into its least 4 bits.

We do not give here full lists of the bootloaders' commands because we don't aim to make this paper a manual. Moreover, we describe a general methodology of testing, so a reader can both follow our recommendations or refuse them if they wish. However, after the bootloader, described above, was burnt into ROM of this experimental circuit, by means of a simple GNU/Linux terminal very soon it became clear what exactly did not work in the system. While this system was being evolved, such a difficult bootloader became less and less useful. It was easy to check separate processor instructions from POSIX terminal, but it was not trivial to write a utility which sends data into memory of such a device: there was no command in the bootloader for receiving an array since such a command was dangerous when parts of its arguments could be lost while sending them through UART. Later a special mode of the bootloader was added which enables "dangerous" commands.

In order to check the contents of the data register of this bootloader from desktop, ternary logic is required to consider undefined values of some bits. What is the ternary logic? How do we extend the binary operations to the ternary case?

Every boolean variable can be either True of False. If we know that a boolean variable a is *True*, we write $a = 1$, if we know it's *False*, we write: $a = 0$. If we *don't* know whether the variable a is *True* or it is *False*, we write: $a =?$. Now we think of the variable a" as it can take three different values: 1, 0 and ?". If $a =?$, we say that the variable a is *Undefined*.

Now it is clear what the ternary logic is and why it is useful. Let we have two variables a and b. What is, for example, the result of the conjunction a and b if a is *Undefined*? It depends on b. If $b = 0$, then it is necessary that $(a$ and $b = 0)$. Otherwise, the result is *Undefined*.

Considering the boolean operations in the same way, we extend them to the ternary case and obtain the following tables:

and	0	?	1
0	0	0	0
?	0	?	?
1	0	?	1

or	0	?	1
0	0	?	1
?	?	?	1
1	1	1	1

xor	0	?	1
0	0	?	1
?	?	?	?
1	1	?	0

not	–
0	1
?	?
1	0

Now, the next question: how to implement these tables? A method is to do it directly. But it takes a lot of code and it is not interesting. Another way is to use the Peirce's arrow or the Sheffer stroke. It is easy to check that if we define them as

$$x \text{ nor } y = \text{not } (a \text{ or } b)$$

$$x \text{ nand } y = \text{not } (a \text{ and } b)$$

then the following statements hold true:

$$\text{not } x = x \text{ nor } x = x \text{ nand } x \quad (1)$$

$$x \text{ and } y = (\text{not } x) \text{ nor } (\text{not } y) = \text{not } (x \text{ nand } y) \quad (2)$$

$$x \text{ or } y = \text{not } (x \text{ nor } y) = (\text{not } x) \text{ nand } (\text{not } y) \quad (3)$$

So, we could implement either the Peirce's arrow ("nor") or the Sheffer stroke ("nand") directly and to implement all other operations using the formulas (1) – (3), without hardcoding the tables for them. But what to do with the exclusive disjunction? This is a very important operation because both the addition of numbers and the subtraction use the exclusive disjunction.

If we say that the Peirce's arrow requires a command, then to calculate, for example, the negation, we also need only one command: this command is the Peirce's arrow, its arguments are both the argument of the negation. To calculate the conjunction, we need either 2 commands, if the Sheffer stroke is used, or we need 3 commands in case of the Pierce's arrow. In the first case, the first command is "nand", the second command is "not" (remember "not" needs only one command). In the second case the first two commands are "not", and the third command is "nor". Analogously, the disjunction takes either 2 or 3 commands.

If we use the most evidence formula for the exclusive disjunction:

$$x \text{ xor } y = (x \text{ and not } y) \text{ or } (y \text{ and not } x)$$

then the operation becomes very slow, because we need to calculate two negations (2 commands), 2 conjunctions (4 or 6 extra commands) and the disjunction (3 or 2 extra commands). It is reasonable to implement the exclusive disjunction directly or... to use other operations instead of the Peirce's arrow and the Sheffer stroke.

Really, if we implement the exclusive disjunction directly, then we have two operations directly implemented: the other one is either the Pierce's arrow or the Sheffer stroke, or maybe another one... What if there are two other operations which

make calculations more efficient, being implemented directly? Let us implement, for example, both the Pierce's arrow and the Sheffer stroke. Won't the formula for the exclusive disjunction become much more simple?

A special program (more exactly, a script) was written, looking for the shortest formula. It did not find any formula which would take less than four commands. The shortest formula is the following:

$$x \text{ xor } y = (x \text{ nor } y) \text{ nor not } (x \text{ nand } y)$$

But if we implement the conjunction and the Pierce's arrow directly, we need only 3 commands to calculate the exclusive disjunction:

$$x \text{ xor } y = (x \text{ nor } y) \text{ nor } (x \text{ and } y)$$

For general calculations, it becomes even more efficient than if the exclusive disjunction and the Pierce's arrow be directly implemented. The reason is the integer operation "$x + 1$". The conjunction is used to calculate the carry flag. It means that 3 commands (the conjunction and two Pierce's arrows) are sufficient to calculate both the digit and the carry flag. If we have, for example, the Pierce's arrow and the exclusive disjunction directly implemented, 4 commands are required to calculate both the digit and the carry flag: a command to calculate the exclusive disjunction for the digit and 3 commands to calculate the conjunction for the carry flag.

Let we have the Sheffer stroke and the exclusive disjunction directly implemented. Then we also need only 3 commands to calculate both the digit and the carry flag for the operation "$x + 1$". How to compare this implementation with the case when the Pierce's arrow and the conjunction are implemented directly? The following evaluation can be considered.

Let we have a long sequence of boolean operations: for example, some conjunctions, some disjunctions, some negations. The operations are distributed arbitrary in the sequence, but every two of the operations can be found in the sequence the same number of times. So, for example, we have 10 negations, 10 conjunctions and 10 disjunctions. In the case when the Pierce's arrow and the conjunction are implemented directly, the sequence will take 40 commands. But in the case when the Sheffer stroke and the exclusive disjunction are implemented directly, 60 commands will be required. We see that the first implementation becomes more efficient than the second one. Another example is 10 negations, 10 disjunctions, 10 conjunctions and 10 operations "$x + 1$". Again the first implementation becomes more efficient: 70 commands vs 90 commands.

Though the Sheffer stroke and the Pierce's arrow are both basises for the binary logic functions, the formulas for the exclusive disjunction (hence, for the operations "$x + 1$" and "$x - 1$") are quite long. So, it is reasonable to implement some two operations directly, not only one. What are these two operations? In the case of ternary logic, it is better to take the Peirce's arrow and the conjunction. In the case of binary logic, it is even much better to take the anti-implication and the disjunction.

IV. Conclusion

One of the core questions, which this paper was dedicated to, is which input data sets should be included in tests: consider two different input data sets for testing the same processor instruction; when should they be regarded as so similar as successful completion of a test for one of these data sets will most probably mean the same instruction works correctly with the other input data set? It has been demonstrated in the article, by considering a specific example of the instruction "mov", how it is possible to find answers for such questions.

Although our practical methodology did not require any mathematical methods, we see in the end of paper using of discrete mathematics helps to make computer programs more efficient. By considering the data register and the address register of a bootloader as abstract variables we in fact used math modeling. Nowadays different parts of mathematics appear in programming more and more often. The papers [8] and [9] give some more examples of how discrete mathematics and math modeling can help to solve practical problems.

References

[1] M.I. Dyabin, A.V. Reshetnikov and E.A. Saksonov, "A methodology for testing the microprocessor core of a system on chip with a x86-compatible microprocessor" (in Russian), *Problems of Perspective Micro- and Nanoelectronic Systems Development*, issue 3, 2020, pp. 172-179.

[2] I.A. Chegis and S.V. Yablonsky, "Logical methods of control of work of electrical schemes" (in Russian), *Trudy MIAN SSSR*, vol. 51, 1958, pp. 270-360.

[3] S.G. Bobkov, "A methodology for testing circuits for the "Baget" series of computers" (in Russian), *Programmnyye producty I sistemy*, no. 3, 2007, pp. 2-5.

[4] V.A. Patankar, A. Jain and R.E. Bryant, "Formal verification of an ARM processor", *in Proc. Twelfth International Conference on VLSI Design. Goa, India*, 1999, pp. 282–287.

[5] A.V. Arkhipkin "Data communications equipment for a complex with a light class UAV" (in Russian), *in Trudy II nauchno-prakticheskoy konferentsii "Perspektivy razvitiya I primeneniya kompleksov s bespilotnymi letatelnymi apparatami". Kongress-tsentr parka "Patriot", Moskovskaya oblast*, 14 Apr 2017, pp. 29-34.

[6] V.Ya. Arkhipkin, M.I. Dyabin, V.V. Erokhin and Yu.L. Leokhin, "Designing a high-performance SoC based on a 16-bit processor core", *Problems of Perspective Micro- and Nanoelectronic Systems Development*, issue 4, 2020, pp. 134-139.

[7] Intel Corporation, *80C186EC/80C188EC microprocessor user's manual*, Intel Corp., 1995., 515 p.

[8] E.A. Saksonov, Yu.L. Leokhin and P.V. Panfilov, "Structural Synthesis of the IoT System for the Fog Computing", *2019 24th Conference of Open Innovations Association (FRUCT), Proceedings of the 2019 International Conference on*, (FRUCT) 2019, pp. 381-387. DOI: 10.23919/FRUCT.2019.8711934.

[9] V.N. Azarov, E.A. Saksonov and Yu.L. Leokhin, "Analysis of Information Structure of the Corporate Network of Enterprise", *Quality Management, Transport and Information Security, Information Technologies, 2018 IEEE International Conference on*, (IT&QM&IS) 2018, pp. 9-12. DOI: 10.1109/ITMQIS.2018.8524906.