ISSN 0361-7688, Programming and Computer Software, 2021, Vol. 47, No. 7, pp. 493–504. © Pleiades Publishing, Ltd., 2021.

Investigation of RISC-V

V. A. Frolov[a,b,*], V. A. Galaktionov[a,], and V. V. Sanzharov[b,***]**

[a] *Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Miusskaya pl. 4, Moscow, 125047 Russia*

[b] *Moscow State University, Moscow, 119991 Russia*

**e-mail: vova@frolov.pp.ru*

***e-mail: vlgal@gin.keldysh.ru*

****e-mail: vadim.sanzharov@graphics.cs.msu.ru*

Received December 14, 2020; revised January 15, 2021; accepted February 8, 2021

Abstract—An instruction set architecture (ISA) is a core around which the rest of a CPU is built. Errors or inflexible solutions once embedded in an instruction set remain with a corresponding generation of processors forever. Hence, one of the key reasons why the growth in the performance of modern CPUs slowed down is that the source code of processors "got corrupted" literally and figuratively: processors become more complex, which makes their further development more difficult. In any case, the development of modern computers (CPUs, GPUs, or specialized systems) is an extremely expensive process, which involves a large number of expensive stages. Therefore, the overall cost of CPU development is a key issue. In this paper, we investigate popular instruction set architectures, as well as make some conclusions about the prospects of RISC-V and other open-source architectures. We try to answer the following questions. Why an instruction set architecture is really important? Why RISC-V is better than the other architectures? Which opportunities does RISC-V open for developers around the world and what competitors does it have?

DOI: 10.1134/S0361768821070045

1. INTRODUCTION

One of our main goals as developers is to write highly efficient applications. The problem is that, nowadays, programming is not an easy process. Typical C++ code uses hardly a tenth of the performance of modern CPUs [1]. This situation is mostly due to the lack of a transparent interface between the programmer and the hardware.

• High-level algorithmic description in C++ (or another language) is translated into assembler code by using various optimization techniques up to automatic vectorization. At this stage, the compiler actively employs machine-dependent optimizations (i.e., it optimizes the code for a particular family of processors).

• In practice, an assembler does not provide any information about the degree to which this code is optimized, even though it is directly translated into a machine code. Here, there is a fundamental problem.

The fact is that, in addition to the architectural level (i.e., the instruction set itself) visible to the compiler and programmer, there is also a microarchitectural level, which is responsible for execution of these instructions on the processor. Thus, the instruction set is a kind of an interface, while a particular generation of processors (e.g., AMD Zen [2] or Intel Cascade Lake) is an implementation of this interface. When an interface becomes outdated (ceases to meet current requirements), the programmer modifies it. However, there are cases where the interface cannot be modified, e.g., if other developers use its old version. For software systems, this is not a big problem because an old interface can usually be implemented through a new one, which makes it possible to move forward while providing backward compatibility. For hardware, however, this is not the case.

Problem 1. Any extra function in an instruction set requires certain resources of the chip (transistors, frequency, heat dissipation, etc.), which increases the complexity of the whole solution. Additional pipeline stages aimed, e.g., at maintaining frequency cause an increase in latency (immediate execution time) of instructions, which can affect the efficiency of the entire system. Since backward compatibility is important for CPUs, with each new generation of a processor that extends its basic interface, the problems grow like a snowball. In this review, we consider the disadvantages of existing instruction set architectures. However, before proceeding to them, there are a few more things we need to discuss.

Problem 2: Lack of cross-platform capability. Presently, high-performance software components (libraries) heavily depend on particular hardware. CPU manufacturers deliberately release open-source soft-

ware stuffed with as many hardware-dependent instructions as possible. For instance, Intel provides an open-source ray tracing library (Embree), as well as many other free libraries, which is fully packed with hardware-dependent *intrinsics*. Needless to say, the cost of the Xeon processors for which this library is optimized is rather high. The cost of porting such a library to a processor with a different instruction set architecture is equivalent to writing its counterpart from scratch. Worst of all, porting the code "head-on" leads to poor performance and, therefore, makes no sense.

Let us consider a simple example. The x64 set has the *shuffle* instruction, which allows the contents of a vector register to be arbitrarily redistributed. Using this instruction, we can implement a cross product of two three-dimensional vectors stored in registers:

```
__m128  shuffle_yzxw(__m128 a_src)
{ return _mm_shuffle_ps(a_src, a_src, _MM_SHUFFLE(3, 0, 2, 1)); }
__m128 cross(const __m128 a, const __m128 b)
 {
   const __m128 a_yzx = shuffle_yzxw(a);
   const __m128 b_yzx = shuffle_yzxw(b);
   const __m128 c =
       _mm_sub_ps(_mm_mul_ps(a, b_yzx), _mm_mul_ps(a_yzx, b));
   return shuffle_yzxw(c);
}
```

However, it turns out that it is impossible to efficiently implement a counterpart of shuffle on ARM with the same order of components (yzxw) as in the general case. Nevertheless, this does not mean that the code using vector products cannot be efficiently implemented on ARM. In this case, computations and data should initially be organized differently (e.g., using full-fledged code vectorization with processing by four elements, which generally works well even on x64). Obviously, an experienced reader might argue that portability problems can be avoided if we confine ourselves only to the constructs of a programming language (e.g., C++ or Ada). However, even if we do not take hardware-dependent libraries into account, some nuances still remain (see below).

Problem 3: Compiler. CPU developers (e.g., AMD) make significant contributions, especially to gcc [2] and Linux kernel [3], because they have a direct interest in it. The compiler and OS are software systems that also depend on hardware. One cannot expect that the same code is equally optimizable for different CPUs, even if they have almost identical functionality (LLVM alleviates this problem, yet only to some extent).

System security (**problem 4**) is a completely separate issue. Here, the situation is very bad because, simply put, no one guarantees it (recall the "spectre" and "meltdown" vulnerabilities). If you want to be protected, then develop your own computer, including the OS, compiler, and drivers, from scratch.

Finally, **efficient inter-thread communication** in a multithreaded program (**problem 5**) has not yet been properly resolved in most instruction set architectures. Presently, this is especially important, as modern processors increase the number of their cores.

In this case, CPU manufacturers pull the blanket on their side, often deliberately adding "convenient" instructions to their processors. Software developers start getting used to this "convenience" without taking into account the difficulties it can cause in the future when porting code to a processor with a different instruction set architecture. We are not saying that "shuffle" instruction, for instance, is a bad idea. We simply say that this problem should not be solved by companies in an individual manner if cross-platform capabilities, backward compatibility, and security are important. There must be a solid standard the developer can rely on to guarantee that his or her program can subsequently be ported to newer, better, faster, more energy efficient, and/or more secure computing systems without significant loss in performance.

Remark. In this regard, we should mention GPUs. Since backward compatibility never was a goal in their case, GPUs have been progressing significantly faster and have managed to achieve impressive results. We do not say that there are no challenges there; however, the problems of software portability for GPUs are solved at the level of graphics and computational APIs, which are much more flexible than an instruction set architecture. Thus far, GPU manufacturers have agreed on an open standard called Vulkan, which is supported by almost all modern desktop and mobile GPUs. Like RISC-V, Vulkan is a well-designed standard. We intend to discuss this topic in one of our future papers. Meanwhile, we can conclude that, as strange as it may sound, software systems and algorithms that extensively use modern GPUs generally have much wider cross-platform capabilities than their CPU-based counterparts.

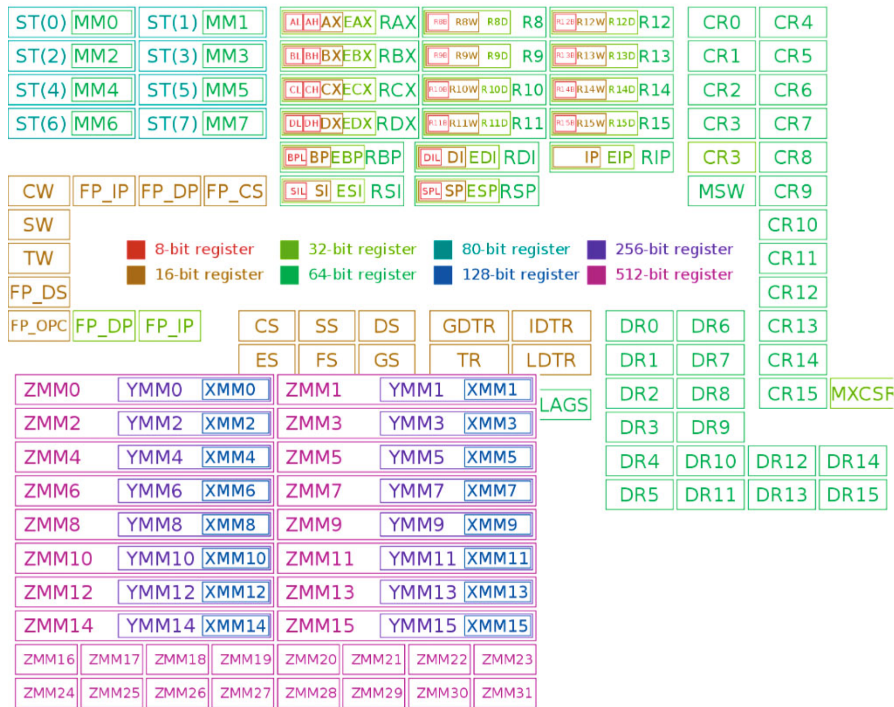

Fig. 1. x64 registers.

2. OVERVIEW OF EXISTING INSTRUCTION SET ARCHITECTURES

First of all, it is necessary to consider the experience in this field accumulated around the world.

2.1. x86/x64

Intel and AMD have achieved significant success in the proliferation of processors of this architecture due to its backward compatibility, aggressive optimizations, and leading manufacturing technologies [4]. However, the design of the instruction set is not their strong suit. This is evidenced at least by the fact that, in their processors, x86/x64 instructions have long been translated to a simpler RISC-like representation. This very much resembles some old program with legacy functionality that is still alive only because everyone is used to it. However, if we put compatibility aside and consider the problem objectively, then x86/x64 “just doesn’t make a lot of sense” ([4, p. 12]). Note that this was said back in 1994 [5]. The main problem with x86/x64 is an extremely bloated (over 2000 items) and poorly structured instruction set. Many of the instructions have not been used or supported for a long time; however, they remain in the instruction set because of backward compatibility. Let us discuss some basic problems.

x86/x64 instruction encoding. In x86/x64, instructions can take different numbers of bytes: from 1 to 15. In this case, with time, shorter operations have become less frequently used. The original idea was to encode the most frequent operations in fewer bytes (recall Huffman codes). Eventually, however, it turned into its opposite. For example, there are a whole lot of six 8-bit instructions for processing numbers in decimal representation; this practice was abandoned in gcc long time ago ([4], p. 12) and it is not implemented in modern CPUs; however, it still valid in the x86/x64 instruction encoding system. The same can be said about the entire x87 coprocessor, which is an atavism in 2020, even though it is used in old 32-bit programs.

Registers. In x86/x64, there are a lot of registers (see Fig. 1). However, at the same time, oddly enough, there are very few of them. The former is due to the backward compatibility between x64 and x86, including segment registers, x87 FPU stack, and other components that have not been in use for a long time. The latter is because there are only 16 useful registers (as compared to just eight in x86). Meanwhile, new registers are permanently implemented on top of the old ones (see Fig. 1) because 32-bit and 64-bit CPUs had to execute existing binary 16-bit and 32-bit codes, respectively.

It may seem that this figure is not so large for modern chips. However, that is not true: first, register memory in modern CPUs can have a large number of ports and, in fact, can be very expensive; second, the register renaming mechanism, which is implemented in all modern CPUs with out-of-order execution of instructions, uses N times more physical registers (where N is generally from 2 to 4) than there are logical

registers in the instruction set. Such a large number of unused registers is almost a crime. However, modern x64 has some other problems.

• Two-address instructions: these instructions always overwrite one of the operands, which hinders standard optimizations and forces the compiler to insert additional move operations. This problem even gave rise to the following meme well-known among programmers: "I like to mov it, mov it."

• Some predicated instructions are designed to improve performance. However, this goal is often not achieved in x64 because their semantics is poorly thought out. For instance, x64 has the conditional load instruction. However, whereas unconditional loading mandatorily requires an exception when referring to an invalid address, it is not specified for conditional loading. Because of this, the compiler quite rarely uses this instruction for optimization when it is necessary to guarantee correct code execution. Incidentally, these instructions are exploited in the well-known "spectre" and "meltdown" vulnerabilities (which, admittedly, affect most of the modern architectures with speculative execution). Upon fixing the critical vulnerabilities in OS drivers, the performance of some applications, on the contrary, dropped by 30%.

• Certain general-purpose registers are not actually general-purpose ones. For instance, the result of the division operation is always in a DX/AX pair, while the result of the shift operation always comes through DX. ESI/EDI also have special semantics. In general, this design pattern leads to inefficient data transfers between registers and the stack [4].

2.2. ARMv7

Currently, x86/x64 have significantly more disadvantages than all other architectures. It is not surprising that x86/x64 lost to ARM in the field of embedded systems, where energy efficiency is important. As compared to x86/x64, ARM certainly seems a better choice. However, the ARM architecture has its own disadvantages.

• Lack of support for 64-bit address space in ARMv7.

• ARMv7 has actually three instruction sets (rather than one): standard mode and two compressed modes (Thumb and Thumb 2). As a result, instruction decoders must understand all three instruction sets, which increases power consumption, latency, and design cost.

• In fact, ARMv7 is not a classic RISC architecture. For instance, its program counter is one of the addressable registers. This means that almost any instruction can alter the control flow (i.e., perform a jump). Worst of all, the last significant bit of the program counter reflects the currently executed instruction set (ARM or Thumb), i.e., even the add instruction can change the instruction set currently executed on the processor!

• Use of condition codes for jumps and predication, in fact, only complicates high-performance implementations with out-of-order execution, even though it makes sense for processors with in-order execution or simple out-of-order execution based on the scoreboard.

• A separate story is complex function call instructions "like, for example, "LDMIAEQ SP, R4-R7, PC," which performs six loads, increments the program counter, and writes seven registers to the stack (including the return address) and, in addition, is a conditional instruction, namely, a conditional jump" [4].

• Finally, ARMv7 is a very large instruction set with over 600 instructions, not including floating point and SIMD.

Of course, the question about the pros and cons of complex instructions, as well as the question about the bloating of the instruction set, is not so simple: is it good or bad to have a large number of instructions? Are complex instructions useful or not? Let us express our opinion by using an analogy. When developing a programming interface, it must reflect—in the simplest and most obvious way—what happens in a system, as well as express the goals pursued by its developer. It should not be too high-level or, conversely, too low-level for its tasks. It should not contain unnecessary functions because this will complicate its subsequent support. A CPU instruction set is a quintessence of a software–hardware interface, and it should contain only functions that are actually implemented at the hardware level. Hence, to answer this question, we should take into account the best practices in this field and make an informed decision in each particular case. If the developer's goals include portability, simplicity, backward compatibility, and security, then it is probably better to avoid complex instructions.

2.3. ARMv8

In 2011, a year after the commencement of the RISC-V project, ARM announced a completely redesigned ARMv8 instruction system with 64-bit addresses and extended set of integer registers. In the new architecture, the ARM engineers removed the ARMv7 functions that made implementation more difficult: the instruction counter is no longer part of the integer register set, there are no more predicated instructions, complex instructions with multiple loads and saves are removed, and instruction encoding is facilitated. However, not all problems were resolved and, in addition, new ones arose.

• Conditional codes are still used in cmov operations, which creates certain difficulties when renaming registers: if a condition is not met, then an instruction must still copy an old value to a new physical register. In fact, this makes conditional move the only instruction that reads three source operands instead of two.

• ARMv8 is not modular. For instance, SIMD instructions that imply the presence of 32 "fat" vector registers are strictly mandatory. This is not suitable for many embedded solutions where low cost and power consumption are important.

In general, ARMv8 is a very large commercial instruction set crammed with various functions: it includes more than 1000 instructions in 53 formats, described on about 6000 documentation pages. However, it still lacks some important things, e.g., an analogue of Thumb (which is important for many embedded systems due to limited memory for code) and fused compare–branch instructions. In addition, the function of placing constants directly in the code of an instruction (the so-called immediate instruction) is limited in both ARM versions: an arbitrary 32-bit constant cannot be loaded directly from program code in one or at least two instructions.

2.4. MIPS

The main problem of MIPS is the insufficient flexibility of its instruction set. Whereas x64 and ARM often contain too-high-level instructions, MIPS instructions go down to the microarchitecture level, assuming strictly defined hardware implementation. For instance, the results of multiplication and division are stored in a special internal 64-bit register. This means that the result is obtained using a separate instruction, even if we need only 32 least significant bits.

```
mult  R2, R3
mfhi  R4
mflo  R5
```

In fact, 32 most significant bits are almost always discarded (what type do you think the result of multiplying two 32-bit numbers in gcc with default settings will have?) and only the least significant part is used. Here, we can see a delay artificially introduced into the instruction set (another example of this delay in MIPS is the branch delay slot, namely, the "nop" instruction, which is inserted immediately after a branch instruction). The problem of different pipeline lengths for different instructions can be resolved by implementing a simple variant of out-of-order execution (scoreboard) while preserving complete determinism and simplicity. On the other hand, the presence of this additional internal state complicates superscalar implementation with out-of-order execution ([4, p. 4]).

There are also problems with floating point in MIPS. It should be noted that, in the MIPS assembler, double-precision instructions use only registers with even numeration. This is because double-precision numbers occupy two adjacent registers, e.g., a pair (f2, f3) when accessing f2.

```
add.s    $f1, $f0, $f1      # single precision add
sub.s    $f0, $f0, $f2      # single precision sub
add.d    $f2, $f4, $f6      # double precision add
sub.d    $f2, $f4, $f6      # double precision sub
```

This solution has certain advantages. Since double precision and single precision are not usually mixed in C++ code, the same set of physical registers can be used more efficiently either for the former or the latter. This contrasts, e.g., with x64 and ARMv8, which use half the 128-bit register for double precision and one quarter of the register for single precision. In addition, conversions from single precision to double precision can be implemented relatively easily (this is also true for x64 and ARMv8). However, there is also a downside here. The use of adjacent pairs of registers complicates superscalar implementation with out-of-order execution when renaming registers: upon renaming, logically adjacent registers cease to be such in the physical register file ([4, p. 6, par. 3]).

2.5. SPARC

A distinctive feature of SPARC is register windows. At first glance, they seem to be a good idea: the actual number of registers is increased using the "base + offset" technique, whereby "R1" actually means "SP+R1." This is convenient: a part of the stack is stored on registers; when calling functions, there is no need to move data anywhere, and the compiler seems to become slightly simpler.

However, register windows are expensive to implement. A large number of registers located "in the stack" are actually not used. Recall that registers are still more expensive than L1 cache due to the need for multi-port memory; for instance, in Elbrus, which is based on SPARC, the register file contains 20 ports, which, in our opinion, is a lot ([6, p. 126]). Moreover, when register memory runs out, an apocalypse in the operating system occurs ([4, p. 6]).

In fact, the register stack could be implemented through a circular buffer, and the unused registers could be swapped in and out from the memory in parallel. However, this requires even more ports in register memory! Something like that is observed in Elbrus ([6, p. 139]); however, the guide [6], in our opinion, does not fully cover this situation. As a result, modern researchers agree on the following: if we have extra transistors, then it is reasonable to just make more registers and let the compiler have fun with inlining functions. This task should not be shifted to hardware.

However, there is another problem. SPARC has a separate set of floating point registers (organized as in MIPS). Communication between integer registers and floating point registers is done only through memory. In addition, there are eight global registers. If a procedure uses floating point arithmetic or modifies global registers, then they must be stored in a stack. Which stack should it be then? We now have two stacks: one in memory for floating point and global registers, and one on registers for integers. If, however, there is only one stack, then we must first save the data into temporary memory cells, only to subsequently put them into our register stack, where they will not be used in any way. As a result, register windows for floating point do not work in SPARC.

2.6. Power

The Power instruction set architecture is a result of the evolution of the instruction sets developed mostly by IBM. This evolution can be represented as follows:

$$\text{POWER} \rightarrow \text{PowerPC} \rightarrow \text{Power ISA v2.x} \rightarrow \text{Power ISA v3.x.}$$

On the one hand, Power is a rather complex instruction set developed by IBM, a company which is not famous for its openness. On the other hand, recently, this architecture was made fully open-source on very attractive terms [7]. Even though Power cannot be called a simple architecture (it includes slightly less than 1000 instructions in 25 formats, including vector instructions), it is still not as complex as ARM. A feature of Power is its modularity: each register belongs to a functional class, and most of the instructions in a class use only registers that belong to this class. Only a small number of instructions pass data between different functional classes. Functional classes include conditional class, fixed-point class, floating-point class, and vector class. Thus, a particular implementation may not support, e.g., floating point operations (like some embedded solutions on the Power architecture). In addition, their support can be disabled relatively easily. Moreover, this differentiation of functionality in the processor provides the compiler with the information about dependencies between registers. However, this approach also has disadvantages: interaction among different functional classes can lead to implementation-dependent delays. Looking ahead, this approach is very similar to RISC-V; perhaps, that is why the RISC-V developers did not mention the Power architecture in [4].

Conditional registers should also be mentioned. Like ARM and SPARC, Power has special registers in which four status bits are set upon executing comparison operations. However, in Power, there are eight copies of them. Each of these copies can be a result of the comparison instruction, and each copy can be a branch source. This redundancy allows instructions to use different conditional states without conflict (when different branches of code have different states for subsequent branches). In addition, it is possible to execute logical operations between these 4-bit registers, which enables one branch to check more complex conditions.

Another feature of Power is two special registers: link and count. The link register is used as a general-purpose register (as in many other architectures) to store the return address when calling a procedure. The count register acts as a counter in loops with a fixed number of iterations. Using this special register, hardware can quickly determine probable branches because the register value is known at the beginning of an execution cycle. In addition, both registers can contain the address of a conditional branch, and there are special instructions to get this address. On the one hand, this solution provides certain speedup when processing branches; on the other hand, it introduces additional complexity in the implementation.

The Power architecture has some other specific instructions: multiple load and store (up to 32 registers), generating a random number in a register, set of 48 instructions to support decimal floating point, etc. For vector extension, 128-bit fixed-point numbers are supported, and there is an extension that adds vector-scalar instructions. Thus, Power has a number of specific features, which can be both an advantage (performance) and disadvantage (complexity). This specificity is reflected in the existing applications and implementations of this instruction set architecture: supercomputers (the first two positions in TOP500 are occupied by Power9-based computers) and microcontrollers (used mainly in the automotive and aircraft industries). Other examples of Power implementations are a component of the Cell Broadband Engine used in Playstation 3 and the Apple Power Mac computers released before 2006.

The large variety of Power applications led to the fact that the ecosystem created by independent developers (rather than the instruction set itself) became one of the main problems of Power. Highly specific applications hinder the formation of a wide developer community capable of producing efficient applications for this architecture and contributing to its support and refinement.

However, in recent years, this situation has been improving. Computers based on Power9 processors [8], which feature open-source firmware of the bootloader and other related components, became available to ordinary users. The instruction set itself became public and open-source FPGA implementations appeared [9]. The Libre-RISCV project aimed at developing an open-source GPU [10], which was initially based on RISC-V, switched to the Power architecture (the reasons, however, were more likely political [11]). A project to create an open-source laptop based on PowerPC is being developed [12]. However, thus far, all in-silicon implementations of the Power

architecture were carried out with the direct participation of IBM. The future will show whether Power, which has become open-source, will be able to catch up with RISC-V or IBM, having once lost its PC market share, will again repeat its mistake. In our opinion, Power is a worthy competitor to RISC-V.

3. INVESTIGATION OF THE RISC-V ARCHITECTURE

The RISC-V instruction set architecture is developed at the University of California, Berkeley. Its basic concept is formulated as follows: RISC-V must become a unified instruction set architecture for computers of all types, from microcontrollers to high-performance systems. In reality, behind this perfectionist agenda, there are a number of practical and very specific goals.

1. **Improving energy efficiency and performance**, lowering the number of transistors, and reducing the cost of developing and supporting processor cores due to a well-designed interface: the developers of processor cores in their basic version are required to implement only a relatively small set of instructions (as compared to other open- and closed-source instruction sets) to obtain a fully operational system. Then, the developers can focus on their specific tasks.

2. **Modularity and extensibility.** Presently, there are many basic instruction sets, and RISC-V is far from being the smallest of them. The main problem with existing instruction set architectures is that a small set of instructions is not universal. It can be focused, e.g., on high performance computing or tasks that require specific hardware functions (multithreading, graphics, cryptography, etc.). RISC-V is designed to solve this problem by implementing a modular structure of its instruction set. PC developers can employ already available extensions (e.g., RVF32 for floating point operations or RVA32 for atomic operations) or add their own solutions: the instruction set is designed to be extensible. Thus, it is assumed that developers of particular systems extend the architecture. This approach fundamentally distinguishes RISC-V from, e.g., ARM: the extension of ARM is not only very problematic, but also the ARM company itself explicitly prohibits its extension in the license agreement.

3. **Reducing the development cost** of various electronic systems by creating an open ecosystem: compilers, operating systems, drivers, and peripherals. In addition, it offers open-access repositories, which already include a lot of high-quality solutions developed by the RISC-V community.

4. **Backward compatibility** of software: programs developed for old CPUs (more precisely, CPUs that support fewer extensions) must run on new ones (CPUs of the same instruction set architecture but with a larger number of supported extensions) without recompilation. In addition, the compatibility of libraries is implemented at the binary level: a static library compiled for one OS and one CPU can be used as it is on another OS and another CPU due to compatibility at the compiler and CPU levels. However, this requires an additional remark. If a program uses some existing extension (e.g., floating point) or, especially, some custom-made extension, then this library may not run as it is on another system. Nevertheless, due to specification of extensions, the situation can be improved! The compiler can recompile a user library while replacing, e.g., floating point instructions or instructions of any other extension with calls of the corresponding functions implemented in software. In the case of commercial instruction sets like x64 or ARM, this is much more difficult to do.

5. **Security.** If vulnerability is found in one RISC-V CPU, then it can easily be replaced with another RISC-V CPU, maybe a less efficient yet more secure one. In this case, all necessary software can be run on the latter with minimal effort and time.

6. **Safety.** Life-critical software (on which people's lives depend) undergoes a special certification procedure and sometimes formal verification, whereby the correctness of a program is proved mathematically. The cost of these procedures can significantly exceed the cost of software development. However, even if the correctness of an operating system kernel is proved, the certification procedure is a separate process and, in that case, it is often impossible to change hardware. That is why the Power architecture is widely employed in the field of civil aviation. The RISC-V standard allows the cost of the certification procedure to be reduced because the instruction set and software largely remain the same. This means that some part of the system is changed in one way or another to meet the requirements of a particular client, and it is important that this part be small to reduce the cost of the certification procedure.

3.1. Core Instruction Set of RISC-V

The RISC-V instruction set has the following distinctive features.

1. **Absence of implicit internal states.** The result of any operation (except for jump instructions) is always placed in a general register. In other words, RISC-V does not have, e.g., state flags, which are used by the cmp instruction in x86. Instead, the result of the comparison instruction is placed in one of the general registers. This approach significantly facilitates superscalar implementation with out-of-order execution; however, as compared to flags or other states, it does not add significant overhead to simple implementations.

2. **Absence of predicated instructions.** More precisely, they are absent in the core set but are included in the vector extension. Predicated instructions are absolutely necessary for VLIW processors (Elbrus) and vector data processing. However, the gain from

Instruction	Format	Meaning
`add rd, rs1, rs2`	R	Add registers
`sub rd, rs1, rs2`	R	Subtract registers
`sll rd, rs1, rs2`	R	Shift left logical by register
`srl rd, rs1, rs2`	R	Shift right logical by register
`sra rd, rs1, rs2`	R	Shift right arithmetic by register
`and rd, rs1, rs2`	R	Bitwise AND with register
`or rd, rs1, rs2`	R	Bitwise OR with register
`xor rd, rs1, rs2`	R	Bitwise XOR with register
`slt rd, rs1, rs2`	R	Set if less than register, 2's complement
`sltu rd, rs1, rs2`	R	Set if less than register, unsigned
`addi rd, rs1, imm[11:0]`	I	Add immediate
`slli rd, rs1, shamt[4:0]`	I	Shift left logical by immediate
`srli rd, rs1, shamt[4:0]`	I	Shift right logical by immediate
`srai rd, rs1, shamt[4:0]`	I	Shift right arithmetic by immediate
`andi rd, rs1, imm[11:0]`	I	Bitwise AND with immediate
`ori rd, rs1, imm[11:0]`	I	Bitwise OR with immediate
`xori rd, rs1, imm[11:0]`	I	Bitwise XOR with immediate
`slti rd, rs1, imm[11:0]`	I	Set if less than immediate, 2's complement
`sltiu rd, rs1, imm[11:0]`	I	Set if less than immediate, unsigned
`lui rd, imm[31:12]`	U	Load upper immediate
`auipc rd, imm[31:12]`	U	Add upper immediate to `pc`

Fig. 2. RISC-V computational instructions. Integer instructions from the RISC-V core set. In fact, there are only 11 different instructions, because most of them are implemented in two formats: R-format (standard) and I-format (whereby the second operand is read directly from the instruction). It should be noted that storing constants in code is a great idea that improves performance. With instructions already being on the chip in one way or another, a constant can be read from instruction memory with zero cost.

using them in scalar code is generally not very large. To improve the efficiency of branching, the RISC-V developers add a simple implementation of a branch predictor (the so-called branch target buffer in the form of a small table of branch addresses), which also allows one to get rid of the branch delay slot. On the other hand, the absence of these instructions facilitates the implementation of superscalar processors with out-of-order execution.

3. **Compact core set of instructions.** The core set (see Fig. 2) includes only 11 basic arithmetic instructions (most of them can occur in two forms, giving a total of 21 instructions), 10 memory access instructions, and 8 branch instructions. Thus, there are 39 instructions in total.

4. **Loading an arbitrary 32-bit constant** from instruction memory in two commands (liu + the next instruction in the I-format). In this case, most of the constants 12 bits or less in length are loaded from program code in one instruction (I-format).

5. **32 general-purpose registers**, which is two times more than in most RISC architectures.

6. **Compressed 16-bit representation** of instructions (RV32C) similar to ARM Thumb for microcontrollers.

7. **Relaxed memory model support** in the core set (see Fig. 3). This model does not deal with actual atomic operations; for them, there is a special RV32A extension. The relaxed memory model is a more explicit (than traditional) expression of data synchronization between threads in a multithreaded program. For this purpose, C++ uses a special part of its standard library (see std::memory_order). The fact is that data communication between threads is actually a very difficult and often expensive operation, taking into account multi-level cache in modern processors. The widely accepted strong memory model (whereby any memory modification made in one thread must be immediately seen in another thread) is one of the main factors that limit performance gain from increase in the number of cores. The relaxed memory model helps the programmer to more clearly express his or her intentions, e.g., by avoiding expensive data synchronization via L2/L3 cache when writing to a memory cell if there is no need in it.

8. **Fused compare–branch instructions**, which reduce the amount of program code.

9. Instructions that **speed up function calls** (jal), virtual function calls, and switch statements (jalr).

Thus, the extremely compact set of less than 40 instructions contains a large number of useful functions.

Instruction	Format	Meaning
`lb  rd, imm[11:0](rs1)`	I	Load byte, signed
`lbu rd, imm[11:0](rs1)`	I	Load byte, unsigned
`lh  rd, imm[11:0](rs1)`	I	Load half-word, signed
`lhu rd, imm[11:0](rs1)`	I	Load half-word, unsigned
`lw  rd, imm[11:0](rs1)`	I	Load word
`sb  rs2, imm[11:0](rs1)`	S	Store byte
`sh  rs2, imm[11:0](rs1)`	S	Store half-word
`sw  rs2, imm[11:0](rs1)`	S	Store word
`fence pred, succ`	I	Memory ordering fence
`fence.i`	I	Instruction memory ordering fence

Fig. 3. RISC-V instructions for working with memory, including special fence instructions for data synchronization in multi-threaded programs.

3.2. Floating Point in RISC-V

The RV32F floating point extension adds 32 new registers. The support of instructions for conversion to integers (including inverse conversion), as well as instructions for immediate data transfer from an integer register to a floating point register, which is a problem for many existing instruction set architectures (where data have to be moved through memory), should be especially noted. In RV32D, there are no move operators for double precision because double-precision registers occupy two times more bits (nevertheless, they are available in RV64D); however, operators for floating-to-integer conversion are implemented in RV32D. The presence of FMA instructions (fused multiply–add/subtract operations), which significantly reduce the amount of code and improve performance, should also be noted.

3.3. RISC-V Vector Extension

The RISC-V architecture has even more useful extensions. Let us consider the RV32V extension for vector operations. It has the following key features.

1. **Vector length is not fixed** in the instruction set, and it is set in a program by using a special instruction. This is a fundamentally different approach, as compared to other architectures, which allows programs compiled for a longer vector to be executed on processors with short vector length, even with lower performance. In modern x64 processors, for instance, this is impossible. If a program is compiled, e.g., with AVX512, then this program can run only on expensive Xeon models and latest 109*0X processors with large vector registers (zmm).

2. **Predicated execution**, i.e., the presence of masks in all vector operations. This approach is equally good for both common CPU programs and massively parallel systems (like OpenCL).

3. **Vector registers** are separated from the other registers, and data transfer between a scalar and a vector is also supported at the hardware level. Hence, in RISC-V, there are no unexpected problems in mixed scalar–vector code as, e.g., in x64 (and especially in x86), where it is often better to explicitly use the vector type __m128, instead of float or int, if we want to be sure that there are no unnecessary data uploads from vector registers to memory.

4. There are some other operators used in vector code, e.g., to load data from memory with a certain step, **gather and scatter**; matrices are also supported.

Overall, we can say that the vector extension is designed taking into account the experience of many other architectures and, like the rest of the instruction set, it is well implemented. In this connection, the enoki library project, which tries to emulate this functionality at the software level on x64 and ARM, should also be noted; moreover, in gcc, variable vector length has been implemented at the compiler level (even though, in gcc, it must be a multiple of four) [13].

4. RISC-V TODAY

Currently, the RISC-V community includes many well-known companies (Fig. 4). Among them, we would like to mention Nvidia, which uses RISC-V and Ada to create self-driving cars, and Alibaba, which introduced an IP block for artificial intelligence in 2019. In addition, RISC-V is already gaining popularity at the level of large government entities: India declared RISC-V a national standard, US DARPA requires RISC-V as a mandatory component for a number of programs (including the entire field of HW security research), Israel Innovation Authority creates a common platform GenPro based on RISC-V, China announced a large grant program to support RISC-V-based solutions (2018), and the European Union discusses a large RISC-V HPC project. In addition, RISC-V forms the basis of training programs in computer science and electronic engineering at many universities. Finally, RISC-V has come to Russia [14].

We are sure that there are more examples that prove the maturity of the standard. What is more important is that, presently, RISC-V already has a firmly established ecosystem:

Fig. 4. Some of the companies that are involved in the development of the RISC-V ecosystem and use this ecosystem in their solutions.

• open-source software: GCC, Linux, BSD, LLVM, QEMU, FreeRTOS, ZephyrOS, LiteOS, and SylixOS;

• commercial software: Lauterbach, Segger, Micrium, ExpressLogic, etc.;

• open-source processor cores: Rocket, BOOM, RI5CY, Ariane, PicoRV32, Piccolo, SCR1 (Syntacore, Russia), Hummingbird, etc.;

• commercial processor cores: Codasip, Cortus, C-Sky, Nuclei, SiFive, Syntacore (Russia), etc.;

• companies using RISC-V internally: Nvidia, Western Digital, Qualcomm, CloudBear (Russia), etc.

4.1. Prospects for Domestic Developers

Here, we would like to mention the most famous domestic processors: Elbrus from MCST, slightly less well-known NMC4 (neuromatrix) from STC “Module,” and “multicore” from Elvis. Both Elbrus and NMC4 are VLIW processors with their own compilers and specific infrastructures. From a technical perspective, in our opinion, these are very good solutions, especially when high performance is required. For instance, the current version of the Elbrus processor holds the absolute record on the number of instructions per cycle (up to 50) and outperforms the latest products from Intel and AMD in many tasks. However, these solutions are expensive and mostly proprietary.

The problem with most of the domestic companies is that their solutions do not sufficiently follow open-source standards, which is why their applicability is limited even in Russia. Considering domestic developments in the field of civil aviation software, we can conclude that the developers are reluctant to work with the closed-source and specific ecosystem of Elbrus, just like with any other non-transparent system. In this connection, it should be noted that, for onboard and many other applications, there are certification procedures and closed-sourced solutions are unacceptable. Moreover, weapons systems exported by Russia are often supplied with foreign electronic components because customers do not trust domestic ones (like any other closed-source systems).

With all its advantages, unfortunately, the VLIW technology (used by Elbrus [6] and NMC4) does not fit in with the RISC-V ideology because of the following fundamental contradiction: whereas the RISC-V approach provides an interface while distinguishing between the architectural and micro-architectural levels, the VLIW approach is exactly the opposite (its instruction set is formed specifically for the microarchitectural level of a particular processor with a given number of blocks of a certain type). For instance, if two different VLIW processors have different numbers of floating point multiplication blocks (and there are no other differences between them), then the compiler generates different programs for them. However, this does not mean that standardization and openness are impossible for VLIW in principle. In this case, OpenCL or Vulkan could be of use: if the execution of a group of threads (e.g., 256 or 512 threads) in the massively parallel model is regarded as a loop, then this loop can obviously be pipelined at the software level. Since, in most cases, OpenCL threads process data

independently, the software mechanism of loop pipelining should provide good results.

4.2. Disadvantages of RISC-V

Despite the obvious advantages of RISC-V over its competitors, there are a number of nuances.

1. Any standardization organization, like the RISC-V consortium (or, e.g., Khronos), has two faces. One of them is openness, compatibility, and ease of entry for small companies. The other one is limited flexibility: only very large sponsors are allowed to add their extensions of an instruction set architecture, and even for them this is a complex process. As mentioned above, the open-source GPU project Libre-RISCV switched to the Power architecture (without changing its name) while referring to the impossibility of obtaining the necessary documentation from the community. However, the cause of the conflict is clear: the developers of Libre-RISCV proposed a vector extension (called Simple-V) that apparently was not of interest to the RISC-V community due to the development of the community's own vector extension.

2. As for high performance, almost all RISC-V materials exhibit a very strong bias towards superscalar cores with out-of-order execution. On the one hand, it is expectable because this way of performance improvement is a natural development of the RISC ideology. On the other hand, it looks like lobbying for a certain technology in order to promote in-house developments.

3. Thus, RISC-V is not a universal solution because, in its current version, it cannot be used as a basis, e.g., for VLIW processors (note that the VLIW approach is not mentioned in RISC-V documentation). Moreover, in the vast majority of cases, the RISC-V community is silent about GPUs, getting away with vague comments in their presentations.

4. As compared, e.g., to Power, the RISC-V ecosystem seems less mature (yet more holistic) because RISC-V still has significantly fewer in-silicon implementations, while Power has long been used in the aircraft industry.

5. CONCLUSIONS

Presently, electronic systems have become so complex that adherence to open standards is now crucial to viability and cost-effectiveness of projects. Even for specific hardware functions, there is no reason not to consider open standards as a basis for development of new technologies. RISC-V is a well-designed standard that allows one to solve the following problems when developing hardware and software systems: compatibility (including backward compatibility in the long term), security, certification, power consumption, efficient multithreading, and cost-effectiveness. This is especially important when designing CPUs and their environment (memory interface, cache, etc.), operating systems, compilers, drivers, and high-performance libraries. When starting a new project in one of these areas, RISC-V is definitely worth considering.

Among instruction set architectures, a worthy competitor to RISC-V is Power (as well as the OpenPOWER project), which in practice proved the possibility of using a unified instruction system for different applications (from embedded systems to supercomputers). It is worth considering if the maturity of the technology is more important then the cost of chip development, or if there is a good ready-made solution available. MIPS has also recently become open-source (due to the progress of RISC-V); however, its extensibility is limited. SPARC is a very outdated solution, which has failed in several applications (first of all, due to register windows). In addition, it has problems with floating-to-integer conversion, which currently makes it one of the worst options.

The other popular instruction set architectures considered in this paper are intellectual property of commercial companies (their licenses only cost from $1 million). ARM, which presently dominates the market, being implemented in a huge number of chips (mainly due to their affordability), will fight to the last against the transition of the global IT community to open-source technologies; however, in our opinion, it is only a matter of time. In addition, with the development of open-source software, the need for x86/x64 support in 2020 has become vanishingly low, and it is likely that personal computers will eventually switch to RISC-V because nowadays x86/x64 "just doesn't make a lot of sense" [5].

REFERENCES

1. Albrecht, T., Pitfalls of object-oriented programming, 2009. http://harmful.cat-v.org/software/OO_programming/_pdf/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf. Accessed April 1, 2020.
2. Venkataramanan, K., [patch][x86_64]: AMD znver2 enablement, 2018. https://gcc.gnu.org/legacy-ml/gcc-patches/2018-10/msg01982.html?print=anzwix. Accessed April 1, 2020.
3. Larabel, M., AMD vs. Intel contributions to the Linux kernel over the past decade, 2020. https://www.phoronix.com/scan.php?page=news_item&px=AMD-Intel-2010s-Kernel-Contrib. Accessed April 1, 2020.
4. Waterman, A.S., Design of the RISC-V instruction set architecture, Technical report no. UCB/EECS-2016-1, University of California at Berkeley, 2016. https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf. Accessed March 31, 2020.
5. Slater, M., AMD's K5 designed to outrun Pentium, Microprocessor report, 1994. http://cgi.di.uoa.gr/~halatsis/Advanced_Comp_Arch/Papers/k5. Accessed March 31, 2020.
6. Kim, A.K., Perekatov, V.I., and Ermakov, S.G., *Mikroprotsessory i vychislitel'nye kompleksy semeistva*

"El'brus" (Microprocessors and Computing Systems of the Elbrus Family), St. Petersburg: Piter, 2013.

7. Blemings, H., Final draft of the Power ISA EULA released, 2020. https://openpowerfoundation.org/final-draft-of-the-power-isa-eula-released. Accessed March 31, 2020.
8. Raptor Computing Systems, Talos II, 2019. https://www.raptorcs.com/TALOSII. Accessed March 31, 2020.
9. Blanchard, A. and Mackerras, P., Microwatt project on GitHub: A tiny Open POWER ISA softcore written in VHDL 2008, 2020. https://github.com/antonblanchard/microwatt. Accessed March 31, 2020.
10. Leighton, L.K.C., Immanuel, Y., Lifshay, J., et al., Libre-RISCV GPU project, 2019. https://libre-riscv.org/3d_gpu. Accessed March 31, 2020.
11. Leighton, L.K.C., [libre-riscv-dev] power pc, 2019. http://lists.libre-riscv.org/pipermail/libre-riscv-dev/2019-October/003035.html. Accessed March 31, 2020.
12. Open Hardware GNU/Linux PowerPC notebooks, 2020. https://www.powerpc-notebook.org/en. Accessed March 31, 2020.
13. GCC manual, 6.49: Using vector instructions through built-in functions, 2019. https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Vector-Extensions.html. Accessed April 1, 2020.
14. Asanovic, K., Redkin, A., et al., *Tekhnicheskii simpozium RISC-V Moscow* (Proc. Tech. Symp. RISC-V Moscow), Moscow, 2019. https://riscv.expert. Accessed April 1, 2020.

Translated by Yu. Kornienko