Issues of compatibility of processor command architectures

Issues of compatibility of processor command architectures

T R Zmyzgova, A V Solovyev, A G Rabushko, A A Medvedev and Yu V Adamenko

Kurgan State University, 62 Proletarskaya street, Kurgan, 640002, Russia

E-mail: tr.zmyzgova@gmail.com

Abstract. Modern computers and computing devices are based on the principle of open architecture, according to which the computer consists of several sufficiently independent devices that perform a certain function. These devices must meet certain standards of interaction with each other. Existing standards relate to both the technical characteristics of the devices and the content of the signals exchanged between them. The article considers the issue of creating a universal architecture of processor commands. The brief analysis of the components of devices and interrelations between them (different hardware features of processors, architecture of memory models and registers of peripheral devices, mechanisms of operand processing, the number of registers and processed data types, interruptions, exceptions, etc.) is carried out. The problem of architecture standardization, which generalizes the capabilities of the most common architectures and is suitable for high-performance emulation on most computer architectures, is put.

1. The problem of compatibility of processor command architectures

A processor command architecture is an abstract computer model that describes what operations the processor can perform directly, what data, how and in what amount it can process. The command architecture is also an interface between hardware and software in a multi-level computer architecture [1].

By the incompatibility of command architectures, we understand the fundamental "nontransferability" without loss of the meaning of the code from one architecture to another because of the absence of the required instructions in the architecture of the destination code or the features present in the source code architecture. For the correct functioning of the translated code you need functional equivalence of the program on both architectures [2]. The structure of the architecture of the processor commands is strongly influenced by the physical structure of the processor as well as other important components such as data buses and commands, RAM, etc. As generations of processors supporting a certain command architecture develop, the underlying hardware architecture may be significantly modified and improved to speed up performance, new commands may be added. At the same time, the equipment should preserve (and/or emulate) the peculiarities of the command architecture, even those that can cause additional complexity of the processor circuit (e.g., real mode and undocumented, but used by some software "unrealistic" mode x86). Thus, one and the same command architecture can have a lot of implementations that differ in execution speed, cost and amount of energy used [3].

Most command architectures can be divided into CISC (Complex Instruction Set Computer), RISC (Reduced Instruction Set Computer) and VLIW (Very Long Instruction Word). The difference

between CISC- and RISC-architectures is that RISC-commands usually have a fixed digit encoding, special commands are used for memory access (architecture "load-store"), instructions usually perform fairly simple operations, due to which they are performed faster than very complex CISC instructions. Examples of command architectures of this type are MIPS, ARM. In turn, CISC-architectures often have variable length encoding commands, and arithmetic processing commands can use operands from memory, which reduces the number of necessary instructions for this type of operation, as well as architectures of this type have a large number of commands that support many addressing modes. A typical representative of this type of command architecture is x86.

VLIW architectures are a manifestation of the idea of explicit parallelism at the instruction level. The coding of such instructions allows you to specify the instructions to be executed in parallel. In such architectures there is no need for complex equipment intended for speculative execution and changing the order of instructions. But the compiler is responsible for correct parallelization. An example is IA-64 (Itanium), the architecture of the Russian Elbrus 2000 processor commands [4].

The architecture of processor commands is influenced by hardware features of processors, for example, digit capacity of processed numbers, digit capacity of bytes and machine words, sizes of processed data and registers, byte order, number of general purpose registers, orthogonality of command architecture, means for processing floating point numbers and SIMD instructions, mechanisms of virtual memory organization, input-output architecture, mechanisms of processor interruption, processor flags [5].

It should be emphasized that there are a number of other features that are specific to some architectures and may have no analogues in others. All this can greatly complicate the binary translation of the code of one architecture into the code for another, making the architectures incompatible with each other at the semantic level.

2. Universal team architecture

2.1. General provisions

To solve the problem of incompatibility of team architectures, it is proposed to develop and standardize the team architecture, which would summarize the capabilities of the most common architectures and would be suitable for high-performance emulation on most computer architectures. In addition, the development of this universal architecture in the form of a compatibility layer working over the actual architecture of the processor commands and hiding all the unrecoverable features of the underlying architecture, such as the design of the virtual memory page table. It is also possible to develop specialized processors capable of directly executing the code of this architecture or having a special command architecture, in which it is enough to simply translate the instructions of the universal architecture.

Introduction of the universal architecture can contribute to the development of an operating system cross-platform at the machine code level. Besides, it can contribute to the development, development and "painless" implementation of new command systems without losing compatibility with old software, including the operating system.

Considering the computer as a device consisting of several levels of abstractions [2], it is proposed to introduce an additional level at which the interface of this universal command architecture will be located - the level of intermediate architecture. This level can be conveniently placed above the firmware level because it usually provides hardware services, and it is desirable to hide these details in order to unify hardware management (figure 1). In this case, the universal command architecture will be implemented primarily in a software-based way.

If you use the codesigned virtual machine approach [6], then the real machine architecture can be specifically designed to facilitate the emulation of the universal architecture, or the universal architecture can be implemented in hardware. In the latter case, the level of the intermediate architecture will fully coincide with the level of the command architecture.

It is proposed to make the universal command architecture of CISC architecture, because it will allow you to use a very wide range of commands and commands of variable length, which can have a positive impact on the volume of the code. Complex instructions when translating to code for RISC architecture should be relatively easy to break down into a sequence of simpler instructions. Machine code for such an architecture should be localized and built into modules to facilitate code discovery. It is also necessary to prohibit switching to code addresses between instructions and arbitrary addresses in general, as this complicates the translation of such code. It is suggested to use special instructions to implement the self-modifying code. These restrictions should greatly simplify AOT translation of machine code [7]. Besides, you may implement some ideas of VLIW in the universal architecture - for example, use a special instruction specifying how many instructions following it can be executed in parallel.

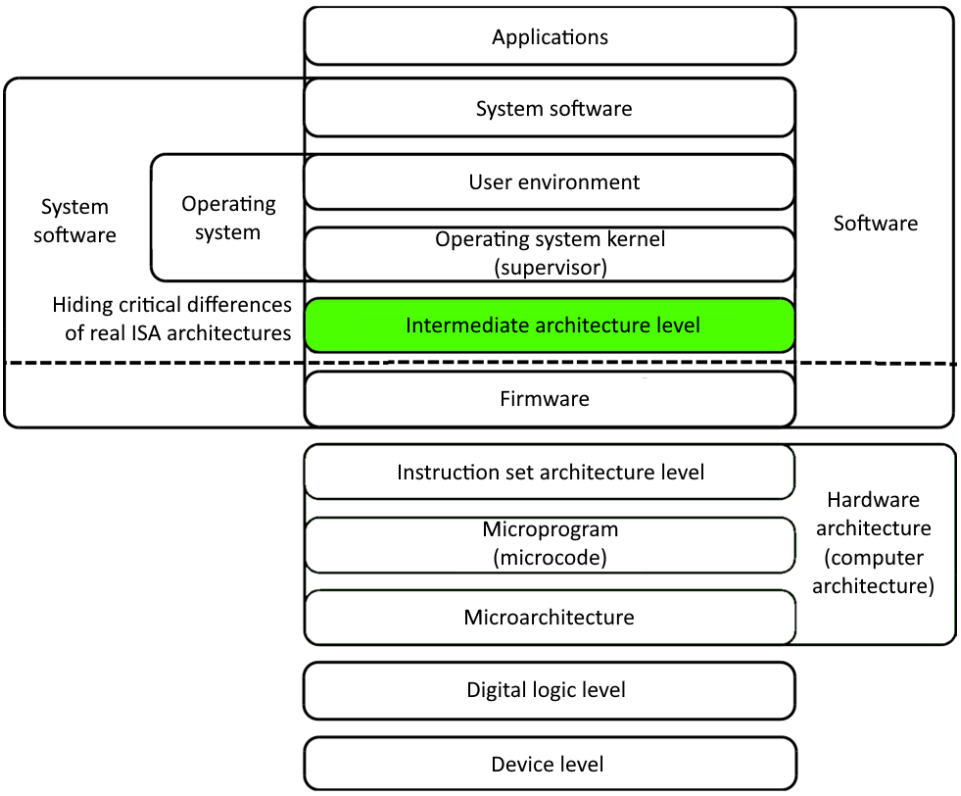

Figure 1. Place universal architecture in a multilayer hierarchical computer level system.

Different features of architectures such as the actual type of processor architecture, actual bit capacity, supported types of virtual memory, memory protection, actual number of registers, processor cache size, memory size, status of memory regions (regular, reserved, region of display of I/O registers, etc.), I/O type, firmware type, available operating modes, etc. can be detected by viewing special system information registers only for reading or using specialized ones CPUID.

To ensure maximum performance, it is possible to provide a mechanism of execution of "native" code for the underlying architecture of the processor, which will reduce the cost of binary translation and may be very appropriate in applications that require significant performance.

2.2. Memory model

The maximum amount of addressed RAM is determined by the size of the address (pointer). You should develop several memory models supported by the universal architecture:

1) Continuous memory model (linear address space) - RAM is a continuous set of binary eight-bit bytes, forming a single address space. It can be used for single-tasking operating systems or system utilities that operate without operating system support. To select a memory cell, its number must be specified in the linear address space.

2) Virtual memory. Each virtual address is projected to a physical address. It can be available in private architecture implementations on an optional basis. Virtual memory support is necessary for modern multitasking OS. For the full use of the virtual memory mechanism it is necessary to cooperate with the OS and implement a universal architecture.

Page organization of memory works transparently for the program code - at a fixed moment of time, part of the pages of the virtual address space is displayed in real physical RAM, and part - is absent in RAM and is in long-term memory, for example, on a hard disk. On the other hand, memory segmentation allows programs to use several independent address spaces called segments. The program code must explicitly specify the segment and the offset in it. Display of virtual memory in physical memory is performed with the help of special tables used by the processor to search for correspondence of the physical memory address to the virtual one. Sometimes tables may have a hierarchical organization to save the space they occupy in RAM.

The architecture should support both segmentation and page organization of virtual memory, at least on an optional basis, as well as a continuous memory model (this is basically possible even in real mode x86). The specific format of display tables should not be regulated. Changing a page table should be done using special instructions, not as a side effect of changing the contents of the memory where the display table is located, because such a change in the display table is difficult to track before the program is launched by simply viewing the machine code without taking into account the context. System information registers or CPUID instructions can be used to determine the number of levels, alignment requirements, dimensions and other parameters of the display table.

It is supposed to use a hybrid model of memory sharing, combining the advantages of Harvard architecture and von Neumann architecture. The processor can also support NX-bit (no execute bit) to prevent code from being executed from the data memory area. This requires an analogue of a page memory organization to mark memory areas with this bit.

Memory and peripheral device registers can be used:

- to be in a separate address space. In this case, they will not overlap with the RAM addresses. Implementing this approach in a universal architecture may be the best solution;
- to be projected to some RAM addresses. In this case a part of RAM addresses may be inaccessible for applied use (this problem is practically leveled out in modern machines where memory sizes are rather large). This variant is more difficult to implement on architectures that do not support this feature. The way of projection can be controlled by the user or can be set in advance by the peculiarities of the real processor architecture.

A channel I/O approach (used in IBM machines) should also be available. In this case, it is also necessary to unify the language of channel programs and broadcast them in the necessary form at runtime. The specific available mode can be viewed in the system information registers.

2.3. Byte order and processor registers

For maximum versatility, operands with any byte order must be processed. Two approaches are possible: the processor has two different modes of operand processing (forward, reverse order of bytes), there is a possibility to switch between these modes, or to develop a command system so that it would be possible to use different commands for different order of bytes (for example, to encode the order of bytes in a certain way in the operation code and instruction mnemonic).

When designing the architecture there is a question - how many registers the universal processor should have?

- Large number of registers (as in RISC architectures). This will simplify the distribution of registers in the program and allow, if necessary, to reserve some registers for service needs. In case of shortage of registers in the real central processing unit, it is possible to project some registers of the universal architecture into the main memory, but it will entail certain performance costs, because it is usually assumed that the work with registers is much faster than working with RAM. This effect can be softened by using the processor cache [2, 4]. In any case, a part of the physical processor's registers may need to be reserved for using the program compatibility layer. figure 2 shows that some of the virtual registers are projected into the main memory.
- Small number of registers (as in x86). This will simplify the emulation of the universal architecture, allowing you to select some of them for the needs of the translator. However, it will increase the so-called "register pressure", which can be significantly weakened by the abundance of addressing modes and other features of the architecture [2].
- Small number of registers (as in x86). This will simplify the emulation of the universal architecture, allowing you to select some of them for the needs of the translator. However, it will increase the so-called "register pressure", which can be significantly weakened by the abundance of addressing modes and other features of the architecture [2].
- You can hide the details of the implementation of local variable storage (which can be stored in registers and/or memory) by referring to them using processor commands like in JVM or CLI. Then you can choose the most efficient variant for implementation on each processor. So, you can completely get rid of registers, and for intermediate operands you can use, for example, the operand stack
- Another variant used in LLVM is the infinite number of registers and the use of the SSA (Static Single Assignment) program representation form. This will allow you to abstract from the specific number of processor registers. At the same time, assigning a variable to each register is possible only once [4]. When launching the program in this form it is necessary to perform the distribution of registers used in the program between a limited number of physical registers of the processor during the code translation, which may cause additional performance costs.

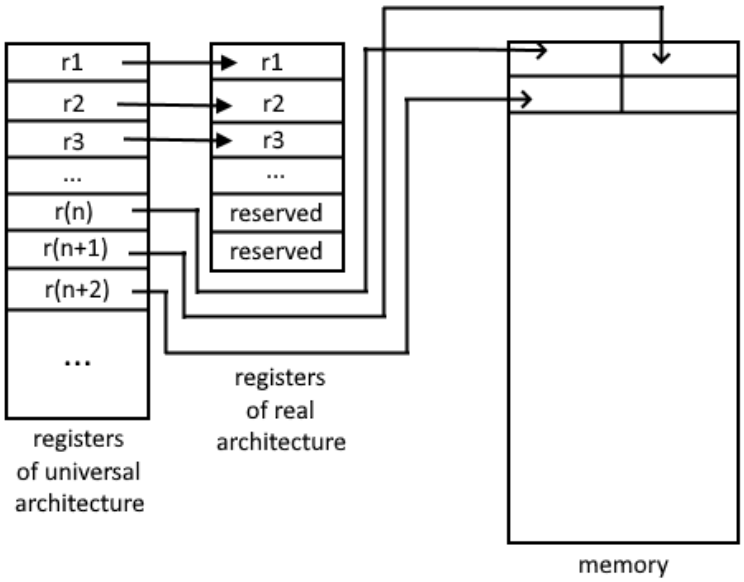

Figure 2. Distribution of registers.

You can also include in the list of registers special volumetric registers, the size of which depends on the implementation and, optionally, change their size. You can save operands of variable length in them. This approach allows you to effectively implement some operations on machines (with a large number of registers or) with large registers.

Note that in this architecture it is undesirable to have a register of flags which should be replaced with special commands for working with boolean variables or to choose the approach of explicit setting of the required flags by instructions, which will greatly simplify the emulation [2].

2.4. Data types directly processed by the processor

The processor shall be capable of handling integers of different digits as well as floating point numbers of different formats. There are several variants of floating point computation formats:

- Force the universal architecture processor to process floating point numbers only in a certain format (e.g., IEEE 754). Advantage of this model is high accuracy and predictability of results of calculations. Disadvantage - low efficiency of implementation on processors with support of other model of calculations with a floating point as it is necessary to emulate calculations in the program way.
- Get rid of using a particular calculation model by the processor. Then the compatibility of the current implementation with the IEEE 754 standard can be checked using the CPUID instruction. The advantage is undoubtedly high efficiency. The disadvantage is the unpredictability of the results, because then the processor does not have to comply with the standard
- The combination of the first two options. Areas of code requiring IEEE 754 arithmetic should be marked in a special way. Otherwise, it is necessary to specify in the specification that the accuracy of calculations is not defined.

Store floating point values in two formats - IEEE 754 and native format (in some cases they will coincide). It is also possible to add hardware support (processor instructions) for arithmetic with variable length operands (long arithmetic). In most cases it can be implemented by the compatibility layer programmatically.

3. Conclusion

When considering the issues of semantic compatibility of different team architectures, the article proposed a solution to the development and standardization of universal team architecture. To solve this problem various hardware features of processors influencing the command architecture, universal architecture of memory models and peripheral device registers, mechanisms of operands processing with any byte order, number of registers and data types processed by the universal processor were stated.

In addition, it should be noted that it is necessary to develop ways of processing internal and external interruptions [8]. You can regulate a wide range of types of interruptions and exceptions of the universal architecture processor, but you need to carry out additional research to find out how effectively the types of interruptions not supported directly by different real processor architectures can be emulated.

Besides, it is necessary to note the organization of compatibility layers which can be assigned, for example, the obligations of implementation of some low-level input-output operations by analogy with BIOS or UEFI. Of course, the compatibility layer should provide a special service of translation of the virtual processor code into the code of the real processor on which the program is executed. In this case, it is necessary to cooperate with the implementation of the universal architecture and operating system for translation, execution of the universal machine code of the architecture and caching of machine code, including in long-term memory.

The process of creating a universal architecture of processor commands is ambiguous and requires a lot of joint research and practical efforts. This article is an attempt to study this issue at least in the first approximation. In any case, the architectural solution should be designed in such a way that the appearance of new requirements and new functionality did not entail modification of the developed logic and was implemented primarily through its extension.

References

[1] Patterson D A and Hennessy J L 2011 A *Quantitative Approach* Computer Architecture (Morgan Kaufmann)
[2] Tanenbaum E and Ostin T 2013 *Computer architecture* (SPb.:Piter)
[3] Joseph D and Dumas II 2005 *Computer Architecture: Fundamentals and Principles of Computer Design* (CRC Press)
[4] Harris D and Harris S 2013 *Digital Design and Computer Architecture* (Morgan Kaufman)
[5] Kim A K, Perekatov V I and Ermakov S G 2013 *Microprocessors and computer systems of the Elbrus family* (SPb.:Piter)
[6] Kostrov B V and Ruchkin B N 2013 *Architecture of microprocessor systems* (Dialog-MIFI)
[7] Braude E 2004 software Development Technology (SPb.:Piter)
[8] Zmyzgova T R and Butenko A Yu 2016 Generation of pseudorandom numbers with entropy source *Informatization and communication* **2** 44-6