Design and Simulation of High Performance Parallel Architectures Using the ISAC Language

Zdeněk Přikryl, Jakub Křoustek, Tomáš Hruška, Dušan Kolář, Karel Masařík, Adam Husár
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
{iprikryl, ikroustek, hruska, kolar, masarik, ihusar}@fit.vutbr.cz

Abstract — **Most of modern embedded systems for multimedia and network applications are based on parallel data stream processing. The data processing can be done using very long instruction word processors (VLIW), or using more than one high performance application-specific instruction set processor (ASIPs), or even by their combination on single chip.**

Design and testing of these complex systems is time-consuming and iterative process. Architecture description languages (ADLs) are one of the most effective solutions for single processor design. However, support for description of parallel architectures and multi-processor systems is very low or completely missing in nowadays ADLs. This article presents utilization of new extensions for existing architecture description language ISAC. These extensions are used for easy and fast prototyping and testing of parallel based systems and processors.

Keywords — architecture description language; ISAC; ASIP; VLIW; multiprocessor system on a chip; simulation; debugging

I. Introduction

Embedded systems have become inseparable parts of our everyday life. A core of such system consists of one or more application-specific instruction set processors. These processors are highly optimized for a given task, such as multimedia processing or network applications. Data processed by these applications can be divided into several streams which can be processed at the same time. Therefore, the most efficient way how to improve the performance is parallelization.

Parallelization can be done either on a software level (e.g. threads) or a hardware level (e.g. more computational units). However, the software level still needs a hardware support (e.g. Intel's Hyper-Threading [16]).

The designer of parallel architectures should have powerful tools which help him or her in a design space exploration. Processor can be described using an architecture description language (ADL) or hardware description language (HDL) [19], [2]. In our point of view, ADL is better, since it hides hardware details. Therefore, it allows easy and fast prototyping of new processors.

In this article, we focus on the description of hardware level parallelism. More specifically, we will focus on the description of basic parallel architectures. The first ones are very long instruction word processors (VLIW). The second ones are multi-processor systems on a chip (MPSoC) which use more than one processor. Based on used processors, the MPSoC can be homogenous or heterogeneous. The homogenous MPSoC is formed from the same type of processors and it is often called multi-core processors. On the other hand, a heterogeneous MPSoC uses a general purpose processor as a control processor and some DSP processor(s) for an audio/video processing. In the following text, the term multi-core processor denotes homogenous MPSoC and the term MPSoC denotes heterogeneous MPSoC. Each of these architectures is discussed in the following sections.

DOI: 10.5176_2010-2283_1.2.46

In complex systems, such as multi-core processors or MPSoCs, interconnections among cores or processors, as well as connection to shared memories, etc., have to be described. The interconnection should be described by using the ADL model as well. The description is used for the simulation and for the hardware description generation.

However, the support for the description of parallel architectures is very weak or completely missing in the nowadays ADLs (e.g. nML [5], LISA [10], EXPRESSION [8], etc.). Therefore, we define new constructions for existing ADL, which allow description of such architectures. The extended ADL is the ISAC language. It was developed within the LISSOM project [15].

Specific ISAC language constructions, that allow the description mentioned above are described in this article. The basic principles of the simulation platform together with the debugging features are also described. Experimental results can be found at the end of the article. The results prove that our solution has great design possibilities, and also, the speed of different types of simulators is very good.

II. State of the Art

A. VLIW Architecture Overview

The history of the very long instruction word (VLIW) processor architecture dates back to 1983 [6]. In the last decades, the VLIW architecture has been very popular, mainly in the embedded systems domain. This popularity has been gained by its high performance, and high instruction level parallelism. The explicit instruction-level parallelism and scheduling of a program execution at compilation time are the main features of this architecture. Each The VLIW instruction specifies a set of operations that are issued in parallel. An instruction contains multiple operation slots. Each of these slots specifies one of the operations that will be issued simultaneously. The VLIW operations are minimal units of the execution and are similar to the RISC instructions [25]. Instructions are encoded and stored in processor memory as *bundles*. Roughly speaking, an encoded operation is called *syllable*. Therefore, the bundle contains several syllables.

From the microarchitectural point of view, the VLIW processors consist of clusters with register files and functional units [7]. The functional units are usually specialized. It means that every functional unit has its own task (adder, multiplier, a unit for memory access, etc.), which is managed by operations. Therefore, this architecture contains several different decoders, while it usually contains only one fetch unit for fetching the whole long instruction words. The clusters can be interconnected, so data needed for a functional unit in one cluster can be transported from another cluster. This is done by special operations.

B. *VLIW Instruction Encoding*

As it has been told previously, instructions are built up from operation slots, typically from four or more. The issued operations are executed simultaneously. The scheduling of these operations is done statically at compilation time. However, when the compiler is unable to plan a useful operation (e.g. functional units must wait for a result of another unit), the NOP (no operation) instruction is issued. The NOP tells the decoder to do nothing. These useless operations are encoded in the same way as the other operations, in the original encoding strategy [6]. This horizontal nature of the instruction set leads to the code size bloat and instruction cache wasting. Therefore, there is an effort to remove useless operations. We talk about the *instruction compression*. The instruction compression makes decoding harder, hence complexity of the fetch and decoding units increase (e.g. the fetch unit has to figure out correct operation dispatching). There are four basic types of the VLIW instruction encoding [7] (see Fig. 1 for the illustration):

Simple encoding encodes every operation, including NOPs, as it is. Therefore, the bundle has the same structure as the original instruction. No operation compression is done, hence the encoding and decoding is trivial, but the code size is large.

Fixed-overhead encoding uses a bit field header that describes the structure of each slot. Therefore, the size of the header equals to the number of slots. The encoding of operations excluding NOPs follows the header. Thus, the bundle size is variable.

Distributed encoding is similar to the fixed-overhead encoding with few differences. The header is distributed directly into syllables as one or two bits (e.g. *parallel bit* or *start/stop bits*). These bits specify if operations run in parallel or not. Information related to NOPs is not stored in the bundle. Every operation is encoded into a syllable, except NOP operations which are not encoded. The bundle size is variable again. This encoding is used in the majority of today's VLIW processors, for example TI C6x [30] or HP/STMicroelectronics Lx ST2xx family [29].

Template-based encoding uses the header for the bundle description, similar to the fixed-overhead encoding. The header contains a template defining operations composition. Main difference between this one and the other encoding types is in the bundle structure. The bundle is of the fixed length, the NOPs are not encoded and the bundle can be created using more than one long instruction. Moreover, one instruction can be encoded into several bundles. Therefore, the fetching and decoding of such bundles is even more difficult because these units have to continuously reconstruct instructions for next cycles and store the rest of bundles. This encoding is used especially in the Intel Itanium processor as *EPIC* (*Explicitly Parallel Instruction Computing*) [26].

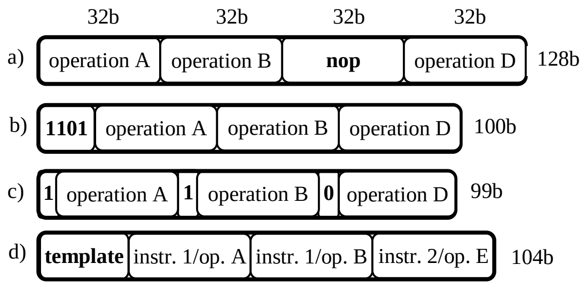


Figure 1. Typical instruction encodings used in the VLIW architecture: a) Simple encoding, b) Fixed-overhead, c) Distributed, d) Template-based

The NOP compression is not the only reason for the instruction encoding. Some architectures use an instruction encoding phase for transformations that are unable to describe in an operation coding description. For example, the HP/ST Lx ST231 transforms operations with long immediate values, which cannot fit in fixed length operations, to two following syllables. The first syllable contains the opcode, the register operands and the first part of immediate value, while the next syllable holds the rest of the immediate value. Another example is the CHILI VLIW processor that uses the *interleaving operation encoding*. In the first syllable, the opcodes of operations are stored; in the next syllables the operands are stored.

The current architecture description languages (ADLs) do not support the VLIW instruction encoding at all. Most of the current ADLs (e.g. nML, LISA, EXPRESSION, etc.) support VLIW processors, but only the simple uncompressed encoding is used. In such situation, a user can define the VLIW processor, but is unable to describe features, such as the NOP compression, the interleaved operation encoding, etc.

C. *Multi-core Architecture Overview*

In the middle of this decade, one of the major performance criteria was the processor frequency. Therefore, every vendor tried to develop a processor with the highest frequency. Unfortunately, this approach has hit a dead end. The main reason for this situation is that cooling of such processors has become an insolvable problem because of very high heat output. An example is the Intel's Pentium 4 [13] which was manufactured by 90nm technology. It never reaches frequency higher than 5 GHz since there is no viable way to cool it down.

Therefore, the designers have come with other way to increase performance. Instead of increasing the frequency, they duplicate a simpler processor with lower frequency. In the multi-core terminology, one core equals to the single simple processor. In contrast to the VLIW processors, each core of the multi-core processor has its own fetch unit and one core may be a simple RISC processor. In order to utilize all cores, an application has to be written in a specific way. The multi-core processor itself cannot identify tasks which can run in parallel. These tasks are identified by the application developer (e.g. a thread creation) or by a compiler. The tasks running on particular cores can communicate with the other tasks via a shared memory. For the performance reasons, the cores usually share caches. Since several cores access the same cache, each access must be performed at defined time which is the same for all cores. Even in the case when each core runs at different frequency (for a performance or power-consumption reasons). Therefore, the accesses have to be synchronized. For example, the synchronization element can be a system bus or a memory controller. This is very important from the simulation point of view (it is described in the section V).

An example of such architecture is the Intel Core 2 Quad [12] which uses three layers of caches. Each core has its own L1 cache, then there are two shared L2 caches (two cores share one L2 cache), and finally there is shared L3 cache and it is shared among all four cores. The previous architecture is shown in Fig. 2.

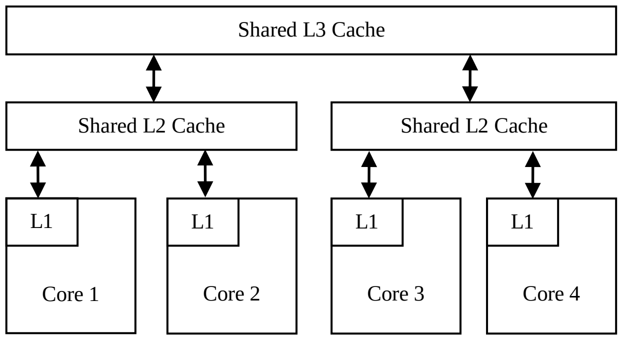


Figure 2. Example of a multi-core architecture

D. *MPSoC Architecture Overview*

Since the manufacturing technology reached 22 nm, more things can be placed on the chip. Hence, the trend of nowadays embedded systems is to place more than one processor on the same piece of a chip [14]. Each of the processors is highly optimized for a given task. For example, a video player can contain one control RISC processor, one digital signal processor (DSP) for audio processing and one or more VLIW processors for video and image processing.

The processors can communicate with others via a shared memory (the same system as in multi-core processors) or by sending/receiving packets (the *network-on-chip architecture* (NoC) [9]) or by interrupts (an interrupt wakes the processor up, it does its job and falls asleep again) or via other interconnection systems (e.g. a *cross-bar*). In the case of shared memory, the situation is quite similar to the multi-core processors (the access to a shared resource is allowed at a defined time which is the same for all processors, which shares the memory). In the case of packet communication, each node in the network has its own memory. When the node needs communicate with other node, it simply sends a packet with all necessary data over the network. Routers, which are also placed on the chip, deliver packet to the receiving node. In the case of interrupts, only one processor usually accesses the memory at a particular time (although it is shared by more processors). For example, it fills a part of memory with data and then another processor can process them. The accesses of two processors are done in independent non-overlapping times. The mentioned different types of communications are also very important from the simulation point of view (it is described in the section V). It should be noted there can be several main memory elements on the same chip, where each one them is used by a subset of processors. Also other components, such as LCD controllers or IO devices, are placed on the same chip. They are usually controlled by one of the processors. An example of MPSoC is shown in Fig. 3.

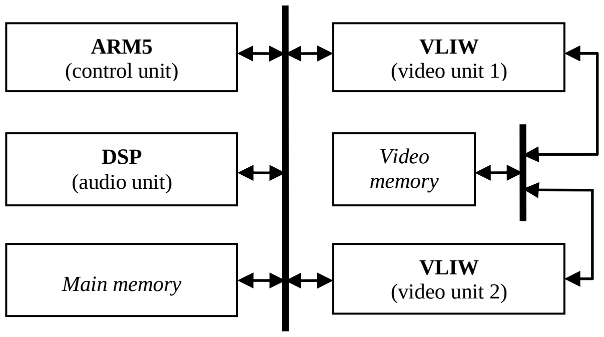


Figure 3. Example of a MPSoC architecture

Nowadays, the MPSoC are designed in a way which uses co-simulation techniques. It means that individual ASIPs are designed using some ADL. The communication among them is simulated using specific modules. These modules can represent routers in the case of network on chip (NoC) architecture, etc. They are often written in the C language or VHDL/Verilog. The simulation of an ASIP together with modules is called *co-simulation*.

III. The ISAC Language

The ISAC (Instruction Set Language C) language was developed in the frame of the Lissom project [15] at Brno University of Technology. It is inspired by the LISA language [6] and it extends the LISA language with new constructions which allow, among other things, to simple model architectures mentioned in section II. The ISAC language [17] belongs into so-called mixed architecture description languages. It means that a processor model consists of several parts. In the *resource* part, processor resources, such as registers or a memory hierarchy, are declared. In the *operation* part, processor's instruction set with behavior of instructions and processor's microarchitecture is described.

The basic constructions of the operation part are the *operation* construction and the *group* construction. An operation can have several sections. The sections are used for four basic model parts of processors. Each model describes an ASIP from a different point of view. There are: the *instruction-set* model, the model of *instruction decoders hierarchy*, the *timing* model and the *behavioral* model. There are usually many operation constructions in an ASIP description.

These operations can be directly connected to each other or grouped according to a functional similarity using the *group* construction. In fact, the group construction creates variants of operations. By using the group and operation constructions, the four mentioned models are built up. The *assembler*, *coding*, *bundle*, *debundle* and *codingroot* sections are used for the instruction-set model. These sections capture format of instructions in assembly and machine language, so they define the instruction in textual and binary forms.

For the behavioral model the *behavior* and *expression* sections are used. In these sections, a subset of ANSI C language can be used. Note that most of the constructs from ANSI C language can be transformed to a hardware description language (e.g. *VHDL*). The behavior section defines the semantics of each operation; so for example, a simple instruction with behavior is described using the assembler, coding and behavior sections, see Fig. 4. The hierarchy of instruction decoders is captured in the *structure* section. In this section, various configurations of instruction decoders can be defined. A processor can use either several instruction analyzers or conditionally used instruction analyzers (the pre-decode and decode phase).

The timing model is described in the *activation* sections. This section is essential for the cycle-accurate simulator and the hardware description generation. It contains links to other operations that are activated for execution either in the same clock cycle or in the future. An activation of an operation can be conditioned. *Events* are operations containing the activation or structure section and operations that are used within these sections. Note that every event can be assigned to a particular pipeline stage, which forms implicit ASIP timing. Timing can be also formed explicitly by an additional ISAC construction. There has to be the special event *main* in the each processor description. It denotes a new clock cycle (i.e. it is the main synchronization event).

It should be noted that the description of instruction set is strictly separated from the description of microarchitecture (the codingroot and structure sections). This approach brings powerful modeling possibilities because there can be more decoders which decode an instruction in a particular order (e.g. Intel's 8051 [28]), while the instruction is encoded in a different order. An example of operations and groups can be found in [21], [17] and in the following sections.

```
RESOURCES {                     // HW resources
  PC REGISTER bit[32] pc;       // program counter
  REGISTER bit[32] regs[16]; // register file
  RAM bit[32] memory {
    SIZE (0x10000); FLAGS (R, W, X);
  };
}

// instruction set description
OPERATION opc_add
  {ASSEMBLER {"ADD"};CODING{0b0};EXPRESSION{0x0;};}
OPERATION opc_sub
  {ASSEMBLER {"SUB"};CODING{0b1};EXPRESSION{0x1;};}
GROUP opc = opc_add, opc_sub;
OPERATION reg REPRESENTS regs
  {// textual and binary description of registers}
OPERATION instruction_set {
  INSTANCE reg ALIAS {rd, rs, rt};
  INSTANCE opc;
  ASSEMBLER { opc rd "," rs "," rt };
  CODING { 0b00 rs rt rd opc };
  BEHAVIOR { // instruction's behavior description
    switch (opc) {
      case 0x0: regs[rd] =
        regs[rs] + regs[rt]; break;
      case 0x1: regs[rd] =
        regs[rs] - regs[rt]; break;
    }
  };
}
```

Figure 4. Example of a simple processor description in the ISAC language.

IV. Modeling of Parallel Systems in the ISAC Language

The ISAC language extensions for parallel architectures design are described in the deep detail in [23]. We will briefly discuss the most important features.

A. *Description of VLIW Instruction Encoding*

In general, the instruction encoding is not a bijective function because the process of the encoding can differ from decoding. In the instruction encoding phase, we lose information about the operation dispatching. This information must be included in the decoding description. Therefore, it is necessary to describe both encoding (used by the assembler) and decoding (used by the disassembler, JIT compiled simulator and VHDL generator).

Both the instruction encoding and decoding is described in the specialized top-level sections (*bundle* and *debundle*) with the modified and very restricted subset of the C language. Furthermore, the templates for generic encoding types are provided. With this approach we are able to describe every encoding type.

In the case of a VLIW architecture which has several slots and the slots have the same instructions working on the different clusters, additional ISAC construction *entity* can be used. This construction describes resources within particular cluster and also the connection to other cluster or clusters (e.g.

one cluster can use registers from the other cluster and vice versa), while the other clusters are simply replicated. The main advantage of this approach is massive reduction of model code redundancy (i.e. model size).

B. Multi-core Processors and Multi-processor System on a Chip

A multi-core processor and can be described using the ISAC language as well. Each processor core is described in a separate model. The processor cores can communicate with other processor cores via shared resources (a resource is marked as a *shared*), such as shared caches. To be more precisely, the resource is marked as shared only in the one model. In other models, the resource has to be marked as *extern* (analogically to the C language modifier, there is no variable modifier called *shared* in the C language because every variable can be shared by default). In addition, it is possible to specify the type of a shared cache and used policy by modifiers *MSI* and *MESI* (based on MSI/MESI protocol). It should be noted that if a cache uses other protocol, the designer is able to use his or her own cache. The bus can be also shared among the processors.

In the case of multi-processor system on a chip, there are usually shared busses or other components. The bus can be described using the ISAC constructions. The situation is similar as in the shared cache description. Sometimes the MPSoC contains special shared functional units, such as routers (used in the case of NoC), which cannot be described via the ISAC constructions. The router can be modeled using the C language (or other language). It creates independent module (also known as *plugin*). The plugin functionality can be used directly in the processors models. The plugin is integrated in the simulator during the simulation Nevertheless, the synchronization or mutual exclusions among the processor accesses have to be solved by the developer within the plugin.

V. Simulating and Debugging of Multi-core Processors and MPSoC

The concept of the simulator generation uses new formal models which were developed within the Lissom project. Namely, it is *two-way coupled finite automaton* (see [11]) and *event automata* (see [24]). The formal models ensure good equivalency between the simulator and hardware representation of the processor. Therefore, no additional huge hardware verification is necessary. In the following subsections the concept of the multi-core processor and MPSoC simulation is described. It should be noted that some of the principles that are described further, such as copying of simulator to network host, are also used in the single processor simulation.

A. Different types of Simulators

Different types of simulators can be used during the MPSoC or multi-core processor simulation. Each simulator type has its advantages and disadvantages. We provide three basic types of simulators.

The first type is *interpreted simulator* [24]. The concept of this simulator is based on a constant fetching, decoding and execution of instructions from the memory. Therefore, the simulation itself is relatively slow. For example, instructions within a loop are fetched and decoded several times in the simulated application, although they were not changed. On the other hand, simulator itself is not dependent on the simulated application, and furthermore, the self-modifying code is supported out of the box. The interpreted simulator creation is also relatively fast.

If the developer wants to increase the speed of a simulation, he or she can use the second type of simulator, the *compiled simulator* [21]. It is created in two steps. In the first step the simulated application is analyzed. The C code simulating the application is emitted based on the analysis. In the second step, the emitted C code is compiled together with the static parts of simulator, such as processor resources etc. It is clear that this version of compiled simulator (also known as *static* compiled simulator) is dependent on the simulated application and the self-modifying code is not supported. Nevertheless, the speed of simulator can be several times faster than the interpreted simulator. The second version of the compiled simulator is the *just-in-time* compiled simulator. It supports the self-modifying code and it is not dependent on the simulated application. It is created in only one step and works in the following way. At the beginning of simulation, the simulator works as the interpreted simulator. The main task of this phase is to find so-called hot-spots (i.e. parts of the simulated application in which the most of simulation time is spent). Then, these parts are compiled, so the subsequent simulation of these parts will be quite faster. Thanks to the first part (hot-spots location) the speed of just-in-time compiled simulator is slower than the speed of static compiled simulator. Still, it can be several times faster than the speed of interpreted simulator. The compiled simulator creation can take more time than the creation of an interpreted simulator (especially just-in-time compiler simulator).

The last type of simulator is the *translated simulator* [22]. It improves the compiled simulator, so it has also two versions (static and just-in-time). The translated simulator is the fastest type of simulator, but it needs additional information about simulated application. It needs starting and ending addresses on all basic block in the simulated application. Thanks to this information, the simulator can be highly optimized. The addresses are stored usually as debug info in the simulated application. These addresses cannot be obtained via the static analysis of the simulated application, because of the indirect jump instructions. Such an instruction uses a value of a register or memory as a destination address of the jump. Therefore, the analysis does not know where the instruction will jump. The only way how to reliably obtain the addresses is to use a high-level language compiler. If it is used for application creation, then it knows exactly where the basic-blocks start and end, and it can simply store the addresses in the application as the debug info. It should be noted that we also provide the C compiler generation. The C compiler is created from the same processor model as the simulator.

The interpreted and compiled simulator can have two levels of accuracy. It can be either the instruction-accurate one or the cycle-accurate one. During the instruction-accurate simulation the basic step of the simulation is a single instruction (one instruction is executed at the time). This simulation type has very good performance, but is quite far from the hardware representation, since the whole microarchitecture is not simulated at all. On the other hand, during the cycle-accurate simulation, the basic step is the single clock cycle. This level is very close to the hardware, because the microarchitecture is simulated now (e.g. pipeline). But the simulation itself is slower. The translated simulation can be done only at the instruction accurate level because of the performance reasons.

Each core of the multi-core processor or each processor in the MPSoC can be simulated using different type of simulator. For example, in the typical MPSoC designs, the already debugged processors are simulated using the compiled or translated simulator. And the processors which are currently debugged are simulated using interpreted simulator.

B. *Concept of Multi-core Processor and MPSoC simulation*

The MPSoC simulation platform in the Lissom project is based on so-called *three-layer* architecture. There are the *presentation*, *middle*, and *simulation* layers (see Fig. 5). The presentation layer accepts commands from the developer, such as the start of a simulation and it displays important information, such as results from a simulation. The presentation layer can have several forms. There is a graphic user interface (GUI) in a form of a plugin for the *Eclipse* platform [4]. Advanced users can use *command line interface* (CLI) allowing the scripting and other advanced techniques, such as the automatic testing, etc. The presentation layer communicates with the middle layer. The middle layer accepts commands and processes them. For example, it accepts a command that the developer wants to create a simulator from a processor description, so the middle layer creates the simulator and sends message to the presentation layer about any possible errors that may occur. The middle layer also takes care of the installation of the simulator into the simulator layer. The simulators can be installed into any suitable host in a network. Note that each of these layers can run on a different host in a network.

As it was already mentioned, for each processor description, an independent simulator is created. This simulator is created at the host where the middle layer is running. Therefore, the simulator can run only on a host which has the same environment (i.e. which has the same operation system as the host with middle layer has).

Then, according to the user configuration, the simulator is transported to the particular host. The configuration file contains all needed information, such as a destination host and port. For copying, the *SCP* application [27] is used since it is multi-platform and it has security features. If the host is *localhost*, then the simulator is not copied anywhere. After the simulator is transferred to the destination host, it is executed and waits until it receives a message which starts the simulation. This message is sent by the middle-layer based on user's command (i.e. user action from GUI or command entered in the command line).

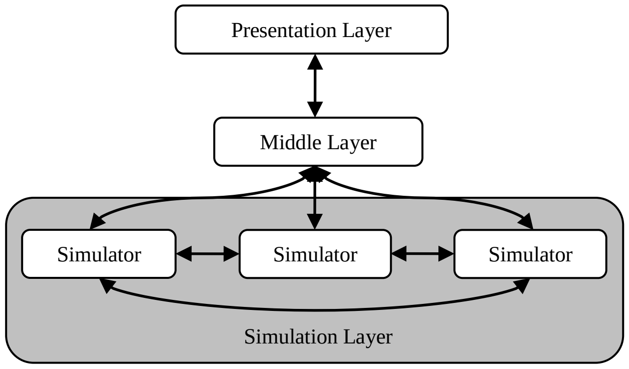


Figure 5. Three-layer model.

The first simulator in the configuration file is so-called *boss-simulator*. We provide two kind of simulation. There are *synchronous* and *asynchronous* multi-core processor or MPSoC simulations. In the asynchronous simulations, the boss-simulator is just an ordinary simulator without any specific tasks. The simulators are synchronized in an application layer. As it was mentioned, this kind of simulation is suitable for example for NoC architecture simulation. In the case of synchronous simulations, the boss-simulator is used for clock cycle generation. The clock cycle generation algorithm is inspired by Bresenham's line algorithm [3], because the simulators are allowed to not have exact divisible frequencies (the configuration file contains information about simulators frequencies). This kind of the simulation is suitable for multi-core processors with a shared memory. According to the clock cycle generation algorithm, the boss-simulator sends the starting messages to particular simulators and waits till it receives ending messages. The boss-simulator has to wait until it receives ending messages from all the simulators (the simulation of one clock cycle can take different amount of time in the different simulator), which received starting message. Only then it knows that all simulators, which had to simulate one clock cycle, finished. Then a new clock cycle begins. Note that the boss-simulator also simulates some processor, so it also receives starting message and sends ending messages.

If the simulator accesses a shared resource, which is not owned by the processor, the similar communication act as in the clock cycle generation is used. It means that the simulator which wants to access the resource sends a message to the simulator which owns the resource. Then the simulator which owns the resource either changes the resource value (write message) or returns its value (read message). The message is sent even if the simulators run on the same host (i.e. accesses are not done via shared memory among the simulators). Access is protected by a variable so concurrent accesses are mutually excluded.

C. *Debugging of Multi-core Processors and MPSoC*

The developer can set breakpoints on any source code line in any application. Note that several types of breakpoints are supported (e.g. conditional breakpoints, counters breakpoints, etc.). If the simulator hits a breakpoint, it stops and sends a stopping message to all other simulators. Then the developer can obtain/set a value of/to any resource of any processor. Also, he or she can control the application execution flow in the step mode. In the case of the synchronous simulation, the step is performed by all simulators (stepping messages are sent to all simulators). In the case of the asynchronous simulation, the step can be performed either by all simulators or by a particular simulator only (the stepping message is sent to the particular simulator only). The simulator can also be resumed from stepping mode (it receives the resuming message). Resuming works in the similar way as the step mode, so either all simulators are resumed or, in the case of asynchronous simulation, only selected simulators can be resumed.

D. *Simulation results*

During the simulation the designer can see values of resources of the particular processor as well as the shared resources. This information can be used for an application and processor model debugging. If the designer needs to optimize the design, he or she needs so-called *profiling information*. This information is collected by the *profiler*. The profiler tracks all important activities in the processor. In general, we distinguish between architecture independent and architecture dependent profilers. The architecture independent profilers work in a way that they inject a new code into an application. This new code keeps eye on important things. The disadvantage of it is that the new code can misrepresent some of statistics, such as a utilization of a particular resource. Therefore, it is mainly used for an application optimization. On the other hand, this kind of profiler is quite fast and can be used among several processor architectures. The architecture dependant profilers have deep knowledge of architecture and therefore, they can get more detailed information, such as cache miss/hit ratio or pipeline utilization. Hence, it can be used for processor architecture optimization. Because they track more things within the architecture, they are slower than the first type of profiler.

The profiler can work on the assembly language level *(low-level profiler*) or on the level of high programmable language (*high-level profiler*), such as the C language. From the low-level profiler point of view, an instruction is important. All statistics are gathered with regards to it. In other words, each executed instruction has information about clock cycles needed for execution or it has information about resources which were used during the instruction execution. From the high-level profiler point of view, a function is usually important (or other high level construction). The function collects other information than the instruction. The function has information about the cache hit/miss ratio and also about clock cycles needed for its execution. The *call-graph* is also reconstructed after the simulation. It captures how the functions were called (i.e. connection among them).

In the Lissom project the architecture dependant profiler is generated. We support low-level and high-level profilers. Both of them are also based on formal models (see [20]). The low-level profiler tracks the accesses to local and shared resources; it can log the executed instructions and computes several statistics, such as the instruction-set coverage or even the program coverage. It also creates a graph for every shared resource. This graph shows the shared resource usage by individual processors. The designer can easily discover bottle-necks in the ASIP design, such as overloaded functional unit, or in the running program, as well as bottle-neck points in the communication, such as an overloaded busses, etc. The high-level profiler can be used especially for the application optimization. It digestedly shows the function statistics. We also provide an interactive visualization of the call-graph (i.e. the developer can simply hide or show a part or parts of call-graph in the different level of details)

VI. Experimental Results

In this section, we briefly provide results of our solution using the ISAC language. For single processor testing we chose two architectures. The first one is *MIPS* architecture (the created model has the simple microarchitecture; the translated simulator is based on the instruction-accurate model). The *MIPS* was developed by MIPS Computers Systems. The instruction-set of *MIPS* is in the version MIPS32 Release 1. The second one is the *VEX* architecture. The *VEX* is a four-slot VLIW processor designed by HP [7]. Each slot processes different types of instructions. The speed comparison of the different simulator types is showed in Fig. 6.

Several programs from *MiBench* [18] test suite (e.g. crc32, sha, dijkstra, etc.) were chosen as the testing algorithms. The results shown in the graphs are the average values from several runs (the maximum and minimum values differ from the average values in tenths of a percent). All simulators were compiled with the *gcc* (v4.3.3) compiler with optimizations –*O3* enabled. The tests were performed on the *Intel Core* 2 *Quad* with 2.8 GHz, 1333 MHz FSB and 4GB RAM running 64-bit *Linux* based operating system.

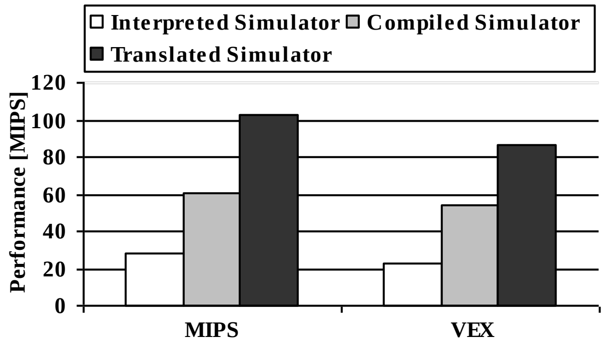


Figure 6. Simulator speeds of single processors.

For the multi-core testing purposes we chose again *MIPS*. We created a system with two cores. Each core has its own L1 cache and both cores share one L2 cache connected to the main memory. Each core solves the same algorithms with different data. For the MPSoC testing, we chose an additional architecture. It is *ARM* [1] architecture (again, created model have simple microarchitecture). The *ARM* model describes the ARMv5 architecture. All the processors used in the MPSoC testing have L1 cache and there is also a main memory shared by all processors. Each processor solves a different task and uses event based notification about finishing its job, so other processor can process next data. This is the asynchronous simulation, so each simulator can run at its maximum speed on a different node in the network.

Results of the multi-core and MPSoC simulations can be seen in Fig. 7. Both simulators ran on the same host. The speed of the multi-core simulation was 2.75 million instructions per second (MIPS). The speed is the same for the all types of simulators. The most of the execution time is spent in synchronization functions, therefore, the speeds are the same (time which is spent in the clock cycle simulation is much smaller than time which is spent in the synchronization functions). The profilers are low-level (i.e. on the assembly language level). Unfortunately, it is quite hard to find other projects which provide similar results, so we cannot directly compare our results with others.

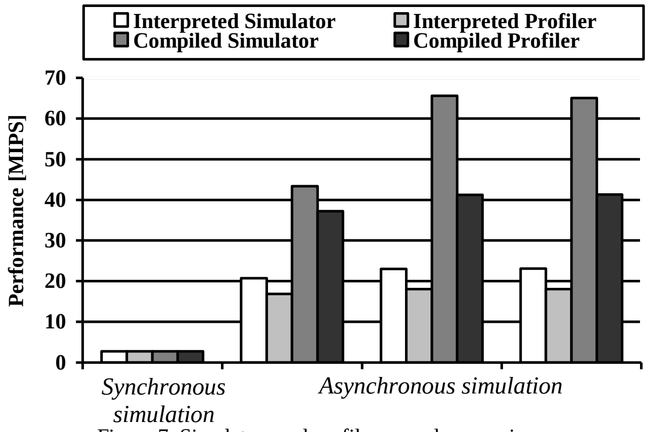


Figure 7. Simulators and profilers speed comparison.

The simulation speed comparison of the ISAC bundle feature can be seen in Fig. 8. The interpreted simulation for three versions of instruction accurate model of VEX processor model has been used: the uncompressed encoding model (Simple encoding); model with Distributed encoding automatically generated from user description in bundle section; and model with optimized build-in Distributed encoding template.

All benchmark algorithms have been compiled with *VEX C Compiler* (v3.42) [7] with no optimizations enabled *–O0*, in order to maximize differences between compressed and uncompressed instruction encoding.

The penalty for the usage of instruction encoding is approximately 3%, mainly because of more difficult decoding phase. The compression rate of the programs can be also seen in Fig. 8. This aspect will positively affect cache hit ratio (note that this speedup can be recognized only in cycle accurate model).

The following table shows line reduction when the *entity* construction is used. As one can see, the reduction is enormous. Therefore, the model is more maintainable.

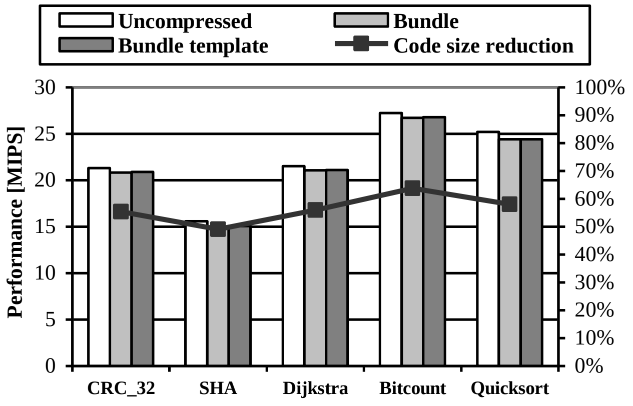
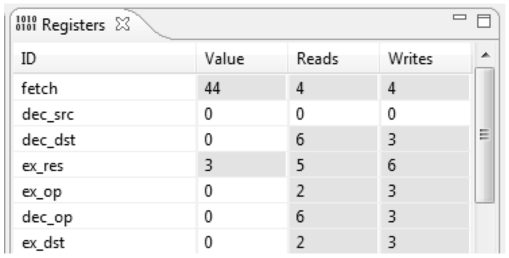


Figure 8. Simulator speeds and code compression ratio based on usage of bundle feature in VEX VLIW processor.

TABLE I. ISAC LINE REDUCTION USING ENTITY STATEMENT.

Processor	ISAC lines	ISAC lines using ENTITY	Line count reduction
VEX	2525	1475	42 %
Chili 2	5630	2650	53 %

An example of the profiler output we can see in Fig. 9. It is a screenshot which was taken after the simulation. It shows some of the processor registers together with the access information. The values show how many times was particular resource read or written. Note that memory elements have additional statistics about execution. Highlighted backgrounds mark changes from the beginning of profiling.



ID	Value	Reads	Writes
fetch	44	4	4
dec_src	0	0	0
dec_dst	0	6	3
ex_res	3	5	6
ex_op	0	2	3
dec_op	0	6	3
ex_dst	0	2	3

Figure 9. Example of profiling information.

VII. CONCLUSION

The support of parallel systems description is very limited (or completely missing) in most of the modern architecture description languages. In this paper, we provide a solution to this problem. We define extensions for an existing ADL (the ISAC language), which allow the modeling of VLIW processors, multi-core processors, and MPSoCs. The constructions for VLIW processors allow describing clusters in a way which does not force the developer to copy and paste the same functionality for different clusters. Therefore, a model of the four-way VLIW processor with two clusters is almost two times shorter in terms of the lines of description code.

Furthermore, constructions for a formal description of the long instruction word encoding and compression have been added. The user can utilize the predefined encoding type or define an entirely new one. Such a feature was completely missing in the contemporary mixed architecture description languages. Usage of the instruction compression leads to smaller applications (in terms of program size). These applications have very good cache hit/miss ratio, which positively affects performance. Exact results depend on the encoding type and amount of program ILP.

The constructions for multi-core processors and MPSoCs allow the description of shared resources and access to them. All of the mentioned constructions allow easy and fast processor and embedded system prototyping.

There are several types of simulators for single processor simulation. Each of them can be used in the different stage of multi-core processor or MPSoC design. There are also two types of single processor profilers, which can be used for processor or application optimization.

We provide two ways of the simulating and debugging multi-core processors and MPSoC. The synchronous simulation can be used with multi-core processors where the processors access the same shared memory. The asynchronous simulation can be used with MPSoC, where the communication is done via sending/receiving packets, interrupts, etc. We provide several types of breakpoints and also several ways of the controlling an application execution flow.

Furthermore, the simulators are based on the formal models allowing better equivalency between the simulators and hardware representations of processors. The co-simulation is also supported.

ACKNOWLEDGMENT

This work was supported by the research funding MPO ČR, No. FR-TI1/038 - System for Programming and Realization of Embedded Systems, BUT FIT grant FIT-S-10-2, doctoral grant GA ČR 102/09/H042, by the Research Plan No. MSM, 0021630528 – Security-Oriented Research in Information Technology, and by the SMECY European project.

REFERENCES

[1] ARM Architecture and Documentation: www.arm.com.

[2] B. Bailey, et al.: “ESL Design and Verification: A Prescription for Electronic System Level Methodology,” Morgan Kauffman Publishers, 2007.

[3] J. E. Bresenham: “Algorithm for Computer Control of a Digital Plotter,” IBM Systems Journal, 4(1): pp. 25-30, January 1965.

[4] Eclipse Platform: www.eclipse.com.

[5] A. Fauth, J. Van Praet, M. Freericks: “Describing instruction set processors using nML”, In: European conference on Design and Test, IEEE Computer Society Washington, 1995.

[6] J. A. Fisher: ”Very Long Instruction Word Architectures and the ELI-512,” Proceedings of the 10th Annual International Symposium on Computer Architecture, pp. 140–150, June 1983.

[7] J. A. Fisher, P. Faraboschi, C. Young: ”Embedded Computing – A VLIW Approach to Architecture, Compilers, and Tools,” Morgan-Kaufmann Elsevier Publishers, ISBN 1-55860-766-8, 2005.

[8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau: “EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability”, In: Design, Automation, and Test in Europe, Springer Netherlands, 2008.

[9] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, D. Lindqvist: “Network on a chip: An architecture for billion transistor era,” In: IEEE NorChip, 2000.

[10] A. Hoffmann, H. Meyr, R. Leupers: ”Architecture Exploration for Embedded Processors with LISA,” Kluwer Academic Publischers, ISBN-4020-7338-0, 2002.

[11] T. Hruška, D. Kolář, R. Lukáš, E. Zámečníková: “Two-Way Coupled Finite Automaton and Its Usage in Translators,” In: New Aspects of Circuits, Heraklion, GR, WSEAS, 2008, pp. 445-449, ISBN 978-960-6766-82-4, ISSN 1790-5117

[12] Intel Core2 Quad-Core Dcumentation. http://www.intel.com/design/core2quad/.

[13] Intel Pentium 4 Processor Documentation. http://www.intel.com/support/processors/pentium4/.

[14] A. Jerraya, W. Wolf: ”Multi-processor Systems-on-chips,” Morgan Kauffman Publishers, 2005.

[15] Lissom Project. http://www.fit.vutbr.cz/research/groups/lissom/.

[16] D. Marr, et al.: “Hyper-Threading Technology Architecture and Microarchitecture,” Intel Technology Journal, 2002.

[17] K. Masařík: “System for Hardware-Software Co-Design,” FIT BUT, ISBN 978-80-214-3863-7, Brno, CZ, 2008.

[18] MiBench Version 1.0. http://www.eecs.umich.edu/mibench/.

[19] P. Mishra, N. Dutt: “Processor Description Languages,” Morgan Kauffman Publishers, ISBN-978-0-12-372487-2, 2008.

[20] Z. Přikryl, T. Hruška: “Cycle Accurate Profiler for ASIPs,” In: 5th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, pp. 168-175, ISBN 978-80-87342-04-6 CZ, Brno, CZ, 2009.

[21] Z. Přikryl, T. Hruška, K. Masařík, A. Husár: “Fast Cycle-Accurate Compiled Simulation,” In: 10th IFAC Workshop on Programmable Devices and Embedded Systems, PDeS 2010, Pszczyna, PL, IFAC, 2010, pp. 97-102, ISSN 1474-6670.

[22] Z. Přikryl, J. Křoustek, T. Hruška, D. Kolář: “Fast Translated Simulation of ASIPs,” In: 6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, Brno, CZ, MUNI, 2010, pp. 135-142, ISBN 978-80-87342-10-7.

[23] Z. Přikryl, J. Křoustek, T. Hruška, D. Kolář, K. Masařík, A. Husár: “Design and Debugging of Parallel Architectures Using the ISAC Language,” In: Proceedings ot the Annual International Conference on Advanced Distributed and Parallel Computing and Real-Time and Embedded Systems, Singapore, SG, GSTF, 2010, pp. 213-221, ISBN 978-981-08-7656-2

[24] Z. Přikryl, K. Masařík, T. Hruška, A. Husár: “Fast Cycle-Accurate Interpreted Simulation,” In Tenth International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions, pp. 9-14, ISBN 978-0-7695-4000-9, Austin, US, ICSP, 2009.

[25] B. R. Rau: “Cydra 5 Directed Dataflow Architecture,” In COMPCON ’88, pp. 106–113, San Francisco, 1988.

[26] M. Schlansker, B. R. Rau: “EPIC: An Architecture for Instruction-Level Parallel Processors,” HP Labs Tech. Rept. HPL-1999-111, February 2000.

[27] SCP homepage: http://www.openssh.com/.

[28] C. Steiner: “The 8051/8052 Microcontroller: Architecture, Assembly Language, and Hardware Interfacing,” Universal Publishers, ISBN: 978-1581124590, 2005.

[29] STMicroelectronics: “ST200 VLIW Series – ST240 SIMD Instruction Set Architecture,” 2006.

[30] Texas Instruments Incorporated: “TMS320C64x/C64x+ DSP – CPU and Instruction Set Reference Guide,” October 2008.

Zdeněk Přikryl

He is a Ph.D. student at the Faculty of Information Technology, Brno University of Technology, Czech Republic. He received his MSc. degree at the same university in 2007. His main research interests are the desing, simulation and hardware realization of embedded systems with one or more application-specific instruction set processors. Nowadays, he is the leader of the simulation team and the hardware realization team in the Lissom project.

Jakub Křoustek

He is a Ph.D. student at the Faculty of Information Technology, Brno University of Technology, Czech Republic. He received his MSc. degree from the same university in 2009. He is currently working on the Lissom research project as the leader of the generic decompiler and debugger development team. His current research interests include the reverse engineering, malware detection and compiler design, with special focus on the code analysis and reverse translation.

Tomáš Hruška

He graduated at the Brno University of Technology, Czech Republic. Since 1978, he’s been working at the Department of Computer Science, Brno University of Technology. He founded the Faculty of Information Technology (FIT) in 2002 and served there as the dean till 2008. Prof. Hruska is currently the vice-dean of FIT. In 1978-1983, he dealt with research in the area of compiler implementation for microprocessor behavior simulation languages. In 1983-1989, he concentrated on design and implementation of both general-purpose and problem-oriented languages. Since 1987 he participated on the project of C language compiler. Since 2006 he’s been working on the design and implementation of the Lissom and Codasip® projects. Prof. Hruska received his CSc. (Ph.D.) in Computer Science and Engineering from the Brno University of Technology, Czech Republic.

Dušan Kolář

He went to Brno University of Technology, Czech Republic, where he studied computer science and cybernetics and obtained his degrees in 1994 and 1998. Since then, he has been working at the university, presently at the Faculty of Information Technology. His main research interests are formal languages and automata and formal models with focus on their exploitation in compilers and formal models transformation.

Karel Masařík

He is a graduate from the University of Technology. He gained his MSc. degree in 2004 and finished his Ph.D. studies at the same university in 2008. He is interested in the design of embedded systems with application-specific instruction set processors using high level description languages. He works currently as a professor assistant at the Brno University of Technology. He is also the CTO of the Lissom project.

Adam Husár

He gained his MSc. degree at the Faculty of Information Technology, Brno University of Technology. Now he does his Ph.D. studies at the same institute. He specializes on embedded systems design with focus on compilers and electronic design automation tools and he has experience with application-specific processor design.