Distributed Simulation and Profiling of Multiprocessor Systems on a Chip

ZDENĚK PŘIKRYL, TOMÁŠ HRUŠKA, KAREL MASAŘÍK
Department of Information Systems
Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 66 Brno
CZECH REPUBLIC
iprikryl@fit.vutbr.cz http://www.fit.vutbr.cz/~iprikryl
hruska@fit.vutbr.cz http://www.fit.vutbr.cz/~hruska
masarik@fit.vutbr.cz http://www.fit.vutbr.cz/~masarik

Abstract: Embedded systems – multiprocessor systems on a chip with application specific instruction-set processors (ASIPs) – become indivisible part of our everyday lives. They are everywhere. Therefore, powerful and flexible way of design and simulation of these systems is needed. The simulators of ASIPs are created using an architecture description language called ISAC. In this paper, the basic concept of simulation, the communication protocol among ASIPs and the three-state synchronization protocol that is used in the distributed multiprocessor simulation are described. Further, the basic concept of profiling and consequential evaluation of profiling information is described. One needs the results from the simulation itself and the results from the profiling to find bugs in the system and to optimize the system as a whole.

Key-Words: Application specific instruction-set processor, multiprocessor system on a chip, simulation, simulator, synchronization protocol, communication protocol, profiling, ISAC.

1 Introduction

Multiprocessor systems on a chip (MPSoC) are very popular these days. In this paper, as a MPSoC we understand a system which can have several processors and basic resources like caches, registers etc. It doesn't contain LCD displays or any other peripheral devices. One can find these kinds of systems in embedded systems or in other more complex systems.

Processors used in these systems are usually application specific instruction-set processors. ASIPs, unlike general purpose processors, have always some special reason to be in system. The reason for this is specific hardware or instruction-set requirements on the processors. Hence they provide a good compromise between high-performance, low-power consumption and flexibility. This means that each processor of such system takes care of something another. For example, if we have a three-processor system in an ogg player then one of them could take of about displaying information to the user, the second processor could be used as the encoder of the ogg format, and it could control a digital-to-analog converter also, and the last one could control these two and accept user action via small keyboard.

As one can expect, designers of such systems want to have a tool, with which they describe a MPSoC and then simulate this system. All this activity we want to do before we synthesize the system and make a silicon layout. These tools usually use some kind of an architecture description language (ADL). ADLs are a special type of languages which allow designers to break away from hardware details. So the designer can focus on the design in a more abstract way. This allows having a system-wide outline of a designed device. It is useful because it decreases the time spent designing the system, and furthermore, it decreases the debugging time of the system. Decreasing necessary debugging time is especially rewarding, because if someone finds a bug in the system, then fixing the bug will be done at the software level. This fix is fast and doesn't bring a need to redesign the silicon layout which is usually very expensive. Thus the system as a whole has a low time-to-market value. Also, designing with those tools is cheaper then designing in the way, in which designers have to write their own simulator from the scratch.

Bugs or the insufficiencies in the design can be found in the results of simulation. More precisely, bugs can be found in the standard simulation results,

which are available at and after every simulation. Insufficiencies, i.e. bottleneck points or bad source code coverage, can be found in the profiling information. If profiling is enabled, additional information is logged during the simulation. Therefore, more statistical information, which is computed from the logged information, is available. As well, the disadvantage of profiling is that it slows simulation down significantly. The results and the profiling information are available for example through the eclipse plugins (figure 11, 12). Nevertheless, we must do profiling, so that we have optimal design of ASIPs on MPSoC and an application.

From another point of a view, the simulation of one processor on another one has many requirements on the resources of a hosted system like the frequency of a hosted processor or available memory. So, if we want to simulate MPSoC on a one-processor system, the requirements of resources will be much greater. The simulation of more complicated MPSoC may be impossible only on a one-processor system.

A one way out of these troubles is distributed simulation. In this case, we divide MPSoC into separate parts and each part we simulate using a different simulator on several hosted systems. It stands to reason that this solution brings some disadvantages too. At first, we have to have some synchronization protocol which controls accessing shared resources. One could say that this is done already by a third party library like for example OpenMPI [6]. But in this case, we have to have support on hosted systems which are unacceptable for us. The reasons why are several, e.g. one has to take care of having the correct libraries on the systems. Furthermore, we have to have a communication protocol. Each simulator has to communicate with each other one by this communication protocol. Also, we have to choose a platform which simulators will use for communication, e.g. TCP/IP.

Despite the mentioned disadvantages, we choose this concept because the concept of distributed simulation has parallel character and it demands fewer requirements on the resources of hosted systems. In the text below a methodology of designing ASIPs on MPSoC with ADL called ISAC is described. Then the concept of simulators follows. In the next part a synchronization algorithm is described, which is used to control accessing shared resources of processors and what methodology is used for distributing simulators to hosted systems. Further, the methodology of profiling ASIPs on MPSoC is described.

2 THE ADL ISAC

The language ISAC is architecture description language which arose out of the language LISA [4]. ISAC was developed at the Faculty of Information Technology at the Brno University of Technology and it is used in the Lissom project. It improves the language LISA, support for hardware description language (HDL) generation is added [3], and furthermore it brings some advantages in the structure of the language itself. It is still under minor development. Architecture of a processor or a multiprocessor system on a chip is described by basic language constructions. These constructions can be divided into two sections.

Firstly, there is a section for describing structural components, so-called resources. As a resource we count, for example memory, caches and so on. Secondly, there is a part for describing the behavior of a processor, its instruction set and timing model. All of these are described by two basic constructions, a construction operation and a construction group.

In next two subsections, we will focus on such constructions, which are important for simulation for the reasons get to be disclosed. Additional information about the language ISAC can be found in [1].

2.1 Resources

As was mentioned, in resources a designer describes the resources of a processor. This section is proposed by the keyword RESOURCES. By resources is meant the basic parts of a processor, like registers (keyword REGISTER), logic circuits, a memory component or a pipeline, etc. Memory components could be ideal memory (keyword MEMORY), real memory or a cache. Some of these resources are mandatory. Every processor must have a program counter, which is a special type of the register (proposed by the keyword PC before the keyword REGISTER), memory of some sort and a memory mapping (keyword MEMORY_MAP). The need to have a program counter is obvious (the processor must know the location of instructions of a program). The program needs to be stored somewhere, so one must have a memory element. And the memory mapping is mandatory because a

processor must know, which address area belongs to particular elements of memory even if the processor has only one element of memory. Each of these structural components has several properties. Among the most important properties is a bit width (keyword bit[<width>]), sizes of the memories (keyword SIZE(<size>)), the number of memory banks or, if the designer wants to design real memory elements, then he must define the latency, etc. All of these constructions could be shared in a MPSoC.

The next example shows the resources part which consists of mandatory resources and four universal registers.

```
RESOURCE {
  // program counter, it has 8-bit with
  PC REGISTER bit[8] pc;
  // universal registers, 8-bit width too
  REGISTER bit[8] ax, bx, cx, dx;
  // special registers for decoding
  REGISTER bit[8] f_data, f_pc;
  // ideal memory, it has 255 8-bit blocks
  // and this memory is executable
  MEMORY bit[8] prog {
    SIZE(255);
    FLAGS(X);
    ...
  };
  // same memory like above, but it is
  // readable and writeable
  MEMORY bit[8] data {
    SIZE(255);
    FLAGS(R, W);
    ...
  };
  // assignment of memory entities to
  // address areas
  MEMORY_MAP defaultmap {
    RANGE(0, 254)->prog [(7..0)]
    RANGE(255, 509}->data [(7..0)]
  };
}
```

Fig. 1: Example of resource sections

2.2 Operations

On the other hand, this section of operations is used for the design of an instruction model and a timing model of ASIPs on MPSoC. The instruction model describes how instructions look like in assembly and machine language. Also this model describes the behavior of instructions. The timing model describes the behavior of a system as a whole. Every operation represents an atomic action in the processor. When the designer wants a more complicated action which consists of more atomic actions, then he could make up a hierarchy. This hierarchy could be created by the basic constructions of the operation section. These constructions are an operation (keyword OPERATION) and a group (keyword GROUP). As well as in resources section, here also are some mandatory operations, like a reset or a master synchronization operation main etc.

The construction group is used when a designer needs to aggregate operations and/or others groups with similar meaning. Also it is used for creating a hierarchy of operations and/or groups. Under the first case we can understand a group called “registers” which aggregates the operations “ax” and “bx”, whereas these two operations could be used for identification resources registers. Under the second case we can understand the creation of a group called “instr_set”. It could also include groups “mem” and “alu” whereas these two groups aggregate related operations within their respective sets.

The construction operation is used for designing atomic entities. It could consist of several sections. In them a designer could describe how the operation looks like in assembly language and in machine language. For this purpose, the assembler (keyword ASSEMBLER) and the coding (keyword CODING) sections are used. Furthermore, when a designer needs to describe the behavior of the operation, then he used the behavior (keyword BEHAVIOR) or the expression (keyword EXPRESSION) sections. A reduced ANSI C language is used in these sections, so the behavior is described by a piece of code.

The most important section in our point of a view is the behavior section. As was mentioned in the text above, this section describes the atomic action of the operation. When a designer designs a multiprocessor system on a chip and the operation of one processor needs to access a resource or resources of another processor, then this access must be in its behavior section. Accessing shared resources of another processor is denoted by the following construction. It consists of the name of a processor which owns the shared resource, a dot and the name of the shared resource. In such case, when the simulation is running at a high level of abstraction, the processor knows where the resource can by found.

The next example shows a small hierarchy of operations and access to the shared resource “bx”. This resource is owned by the processor named “enc”. Note, that the next example show the instruction model.

```
OPERATION ax {
  // description of assembly language
  ASSEMBLER { "ax" };
  // and of machine language
  // 0b denotes binary number
  CODING { 0b00 };
  EXPRESSION { 0; };
}
// operation has similar form like ax
OPERATION bx {
  ...
}
// aggregates similar operation
GROUP register = ax, bx;
OPERATION X {
  ...
  // operation X accesses enc's bx resource
  BEHAVIOR { ax=enc.bx*0.001; };
}
OPERATION Y {
  ...
}
GROUP instr_set = X, Y;
```

Fig. 2: Example of operation sections

When an operation is used for a timing model description, then another two sections are used, a coding root (keyword CODINGROOT) and an activation (keyword ACTIVATION) section. When the operation is used for timing model, we can call it the event. The coding root section denotes when and what will be decoded. It is possible, that in the system, there is more than one instruction decoder present. These instruction decoders can be dependant on what was decoded by previous decoders and they can be delayed. The activation section denotes when and what will be done. In this section each activation can be conditional and delayed. The condition can be formed from constants and/or resource values and/or results from the expression section of an operation, etc.

The next example shows how a timing model might look. We have operation main, which is done every clock cycle. This operation activates other operations etc.

```
// load data from memory, we have to store
// program counter too (it is needed
// to debug the application)
OPERATION fetch {
  BEHAVIOR { f_data=prog[pc]; f_pc=pc;  };
}
OPERATION decode {
  CODINGROOT {
    // we want to decode operation
    // which belongs to instr_set
    instr_set(f_data[f_pc]);
  };
}
OPERATION main {
  // %x means delay of x clock cycle
  ACTIVATION { fetch; %1 decode; };
}
```

Fig. 3: Example of the timing model

3 Simulator

The ways a simulator can simulate ASIPs on MPSoC are divided into two basic approaches; instruction set simulation and cycle-accurate simulation.

The first approach is based on the idea that the basic step of the simulation is the execution of one instruction. In this approach, only the instruction model is simulated. The instruction model is based on coding, behavior and expression sections only. We use some machine code as the simulator's input. After the simulation, we can look at the output of simulation (more information about simulation output is in section 6). As one can expect, there is not much information about the behavior of real ASIPs on MPSoC present, but enough for fast development of an application that will run on them.

The second approach to simulation is based on the clock cycle. In this case, the instruction and timing models are simulated together. The timing model is based on the activation and coding root sections. After the simulation, the designer will have detailed information about the behavior of ASIPs on MPSoC.

The simulation of the instruction model can have two forms; interpreted form and compiled form.

The interpreted form means that every time when an instruction is recognized, its behavior is executed independently. That is why this form is independent of the simulated application.

The compiled form is created in two steps. In the first step, when and what instructions are executed, gets logged. Second, from this log, we create a compiled form of an instruction set simulation. This takes longer to create than the previous form and it is dependant on the simulated application, but the time of the simulation itself is shorter.

The simulation of the timing model can have two forms too; dynamic planning form and static planning form.

The dynamic planning form is based on the idea that there is an event scheduler in the system. The operations, or events if you will, from this scheduler are sequentially executed. Every time that an

operation activates another operation, the activated operation is inserted into the event scheduler.

Unlike of the dynamic planning form, the static planning form is based on the idea, that whole planning is stored in only one state variable [13]. More precisely, the value in this state variable denotes what operations will be executed (in terms of the dynamic planning, what operations are in the event scheduler). This form needs to preprocess the timing model, from which values for the state variable are formed. Again the time of creation is longer, but the simulation itself is shorter.

In the beginning, our simulator was inspired by a simulator that is used in [8] and that is created for ASIP. Later on, it was shown that their approach is insufficient, so we improved it and brought new ideas. The main goal is to have one formal model for the simulation and for the synthesis. One wants to be sure that what he simulates is what he will get in the hardware later. So, our approach is based on formal concepts and from these concepts C code for the simulator or VHDL code for the synthesis is generated. As a basic simulator, the cycle-accurate interpreted static planed simulator was chosen for its performances. There are two formal concepts used. One, called "machine code finite automaton". The other called "event finite automaton".

The machine code finite automaton is a formal model for instruction model simulation, so it accepts machine code as input and eventually it executes some behavior. An example of this automaton is in the figure 4. On the edges are couples. On the first position of the couple, there is a terminal symbol from the input alphabet, i.e. symbols which are accepted and which belong to the machine code language. On the second place, there can be some behavior.

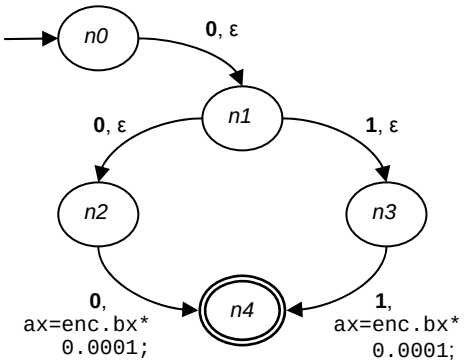

Fig. 4: Example of the machine code finite automaton

This behavior is executed when automaton goes through the edge, i.e. it executes the behavior, expression section.

So, if we go through the automaton edges and if we execute some behavior, it means that we recognized some instruction from the instruction set and we executed its behavior. More information about how the machine code finite automaton is created can be found at [9].

The event finite automaton is formal model for timing model simulation, so it accepts events from the systems and eventually executes its behavior. An example of this automaton is in the figure 5. This time triples are on the edges. On the first position of the triple, there is an event, i.e. operation from activation, coding root section. On the second position, there can be a condition, i.e. if activation of an operation is conditional. On the third position, there can be some behavior. The value of the nodes represents the value that is stored in the state variable of the system.

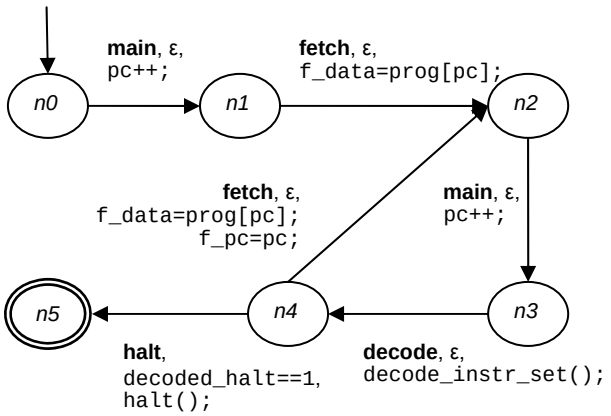

Fig. 5: Example of the event finite automaton

Therefore, if an event occurs in the architecture, then automaton goes through particular edge and changes the value of the state variable. Further, this passage can execute some behavior, i.e. if we have a fetch event, it can load instructions from memory to other resource and can activate decoding events, etc. More information about how the event finite automaton is created can be found at [12].

The described formal models are created for every ASIP that is placed on MPSoC. These models can be connected to each other by the behavior. The instructions can access shared resources; so one has to determine the optimal approach for synchronization of these accesses. This will be described in next section.

At the end of this section, there is a comparison of our solution and solution which is described in [8]. Both solutions are based on the statically planned form simulation, but our solution has several advantages. Firstly, our solution is based on formal models, so the equivalency between the simulators and the final implementation in hardware is guaranteed (VHDL code is generated in similar way as the C code for simulator). Secondly, we do not create compiled form of the simulator, which allows us to change the application that we want to simulate. This is very good in the case when the designer develops and debugs new applications.

If we want to have the fastest simulation, then we can create an extension of the current concept. The compiled form of our simulator will be created. This simulator is as fast as the simulator in [8].

4 Synchronization protocol

A synchronization protocol is used for keeping the local copies of resources in processors in a coherent state. This means that every time a processor wants to obtain the value of some shared resource, the synchronization protocol must guarantee that the processor has a valid value in the local copy of a resource. If the copy is not valid, then the synchronization protocol must get a valid value. Furthermore, if a processor wants to write a new value to a resource, then the synchronization protocol must take care of distribution of this new value into the processors which have invalid values in their local copies of this resource now, or mark the copies as invalid.

There are several ways how one member lets another member know that it needs valid data. It could be signals or by sending messages (in fact, a signal is the simplest form of a message, it carriers only information that something happed, nothing more). Since the second way is more powerful and is more universal, we choose it.

In the subsections below are discussions about the possibilities of synchronization protocols based on messages, their advantages and disadvantages and their usability for our purpose. Also, in the text below by the term processor we mean a simulator of a processor.

4.1 Arbiter vs. broadcast

In general, there are two basic approaches how to communicate among all members, with or without an arbiter. At first we describe communication with an arbiter.

The arbiter is not like any other members in a communication group. Its only job is receiving requests and sending replies. One could say that it works like a proxy. Furthermore for our purposes we would add a cache. This cache would keep information about the states of copies of resources in each processor. Also processors would have their own local copies of shared resources. Thus, if an arbiter knows valid value, then it will send it without bothering a processor which accessed a resource for the last time. When some processors want to write a new value into shared resource, it will send a writing message to an arbiter. The arbiter receives its request and in the cache it is noted that this processors has valid data. But this value is not stored in the cache now. Then it sends to other simulators invalidating messages that their local copy of the resource is invalid. So in cache, it is noted which processor accessed which resource the last. If another processor needs to read a value from shared resource, then it will send a reading message. The arbiter looks into the cache. If it finds, in the cache a valid value, then it sends this valid value to the processor. Also it changes the state of copies of the resource in the other processors according to a protocol. If it does not find a valid value in the cache, then it sends a getting message to the processor which accessed the resource the last. After receiving a reply it rewrites the invalid value in the cache with this value and the same value it sends to the processor which needed this value. And it also changes the state of copies according the protocol. The advantage of this approach is that members do not communicate among each others, so it reduces a number of messages. On the other hand, an arbiter is a weak point because if it breaks then nothing will work.

Another approach is communication by broadcast messages. This means that if some member wants to get something, then it will send a message to all the others. If this message demands a reply, then particular member will send a reply. This second approach eliminates a weak point, an arbiter, but the number of sent messages will be bigger. But after all, this approach has more stability and robustness, so we choose this one and the following two subsections discuses two synchronization protocols based on the broadcast approach.

4.2 Two-state synchronization protocol

The two-state synchronization protocol is the simplest variant of synchronization. A local copy of the resource in a processor could be only in two states. "V" means a copy has a valid value and "I" means that the value in a copy is invalid.

The process of synchronization has the following rules. When a processor needs to write a new value into some resource then it will send a writing message to all the other simulators (it is a broadcast), which use the resource too. After simulators receive this kind of a message, they will change the state of the local copy of the resource to the state "I". On the other hand, when a processor needs to read a value from a resource, it could lead to two cases. Firstly, if the local copy of a value of a resource is in the state "V", then the processor will read this value and continue in computations. Secondly, if local copy is in state "I", then the processor will send a reading message. Then it receives the valid value from certain processor, which has the valid value of the resource. After that it rewrites a local value with the received one, change state of a copy to "V", and continues in its computations.

A disadvantage of this synchronization protocol is, that if a processor writes to resource which it has in exclusive ownership, then it will send writing messages to the other processors again even it is useless. The exclusive ownership means, that the processor wrote a new value to the resource some time ago, and no other processor wanted to write to this resource till now, i.e. all other processors have the state of the copy set to "I".

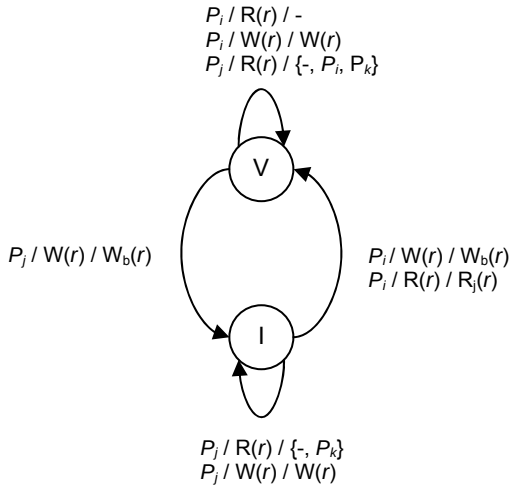

Fig. 6: State diagram for the two-state synchronization protocol

Figure 6 shows a state diagram of a copy of a resource r in a processor Pi. Pi, Pj and Pk are different processors. The below text at transitions have the following meanings: processor / action / broadcast (the index b in subscripts) or message to a particular processor (the other indexes in subscripts).

4.3 Three-state synchronization protocol

In the three-state synchronization protocol we want to get rid of useless broadcast messages when a processor has a resource in an exclusive ownership and it writes to this resource again. The solution is adding a new state which signalizes that a resource is in exclusive ownership.

Now, instead of the two states "V" and "I" we have three states. A state "M", modified, which means that the local copy of a resource is in an exclusive ownership. A state "S", shared, which means that the value of a copy is valid, but a processor is not an exclusive owner. And we have a state "I", invalid, which has same meaning as in the two-state protocol.

Originally this protocol has been designed for a synchronization of the cache in multiprocessor systems with shared memory and the synchronization is also affected by the controller of the bus etc. It is also called the MSI protocol (named after initial letters of names of states). For our purpose we modify the MSI protocol for a cache to the MSI protocol for a resource.

The process of synchronization has the following rules. Writing a new value into some resource it is similar to writing it in the two-state protocol. A processor sends a writing message to all the other processors which use this resource. Then a local copy of the resource in the processor is marked as "M", so the processor is the exclusive owner. The other processors mark their copies as "I", so they will know that they do not have a valid value of the resource. When a processor wants to gets the value of a resource it could lead to three cases. The first two cases are that the processor wants to read the resource and its state is set "M" or "S". It means that the processor is either the last one, which wrote a value to the resource or some another processor has a valid value too. In both cases the processor which reads does not send any messages. The third case is that the value of a local copy of a resource is marked as "I". In that case the processor must send a reading message. A processor which was the last

one which changed the resource answers the valid value to the processor which wants to read the resource. Both of them then set the state of local copies of the resource to "S".

The tree-state protocol removes the disadvantages of the two-state protocol, therefore it is used in the MPSoC simulation in the Lissom project.

Figure 7 shows a state diagram of a copy of the resource r in processor Pi. Pi, Pj and Pk are different processors. The below text at transitions have following meanings: processor / action / broadcast (the index b in subscripts) or message to particular processor (the other indexes in subscripts).

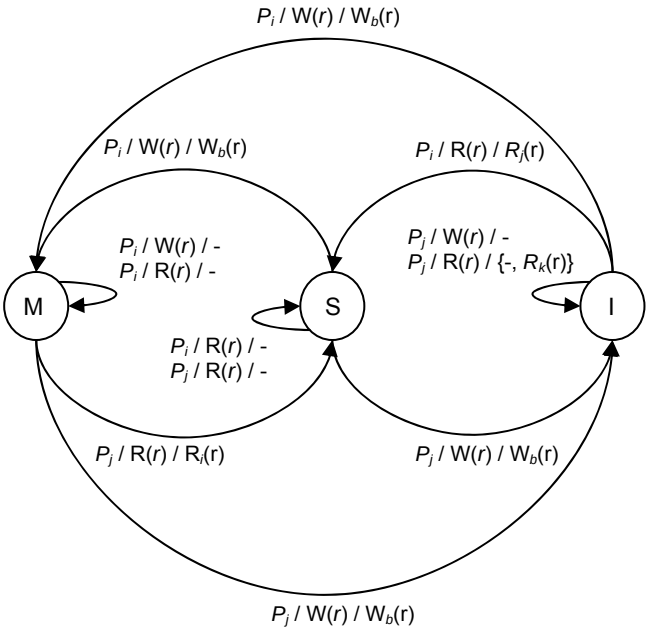

Fig. 7: State diagram for the three-state synchronization protocol

4.4 Experimental results

For testing both synchronization protocols, several applications were created. In this section, we go briefly through two applications and we compare results from the two-state and the three-state synchronization protocol. In both applications, the processors are the same. Only the programs, which run on these processors, are different. In the next two paragraphs, the applications and the results from simulations are described.

The first application is a simple counter from zero to ten. On all processors, the same application is executed. Further, only one register, of one processor, is shared among the processors. The application has the following behavior. In random times, each processor reads the value of the shared register. If the read value is greater then or equal to ten, then the processor will halt. Otherwise, it will increase the read value by one and will write this new value back to the shared register. The whole simulation will end when all processors halt. Note that the mutual exclusion of accesses to the shared resource is done in the processor architecture itself, not in the synchronization protocols. In the figure 8, the average values of sent messages of several simulations are shown. When the two-state synchronization protocol is used, the count of the writing messages is at the maximum, because apart on which processor writes to the shared resource, the broadcast messages are sent every time. On the other hand, when the three-state synchronization protocol is used, the count of writing message can be lesser. It depends on the randomness, i.e. one processor writes new value more times in sequence.

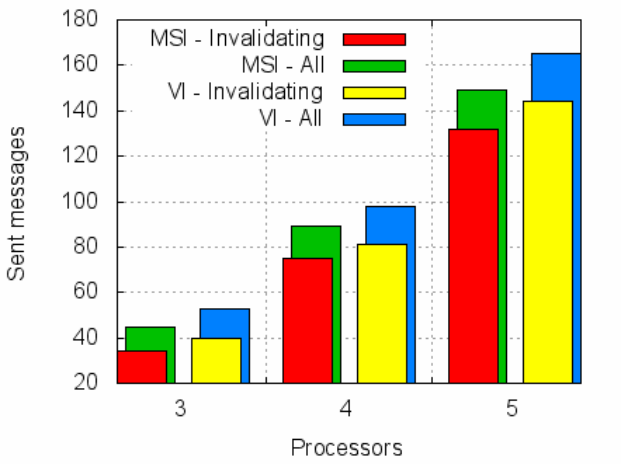

Fig. 8: Graph of the count of messages for counting algorithm

As one can see in this particular application, the advantage of the three-state synchronization protocol is not very significant. As it was mentioned, this is caused by the application character, i.e. every processor wants to read and write to the same shared resource, and by random accesses, i.e. the case when processors alternate in the writing is much more probable than the case when the most of the writing is done by single processor.

The second application is a simple distributed sorting algorithm. On all processors, except one – the master processor, the same application is

executed. Only one memory of master processor is shared among processors. The application has the following behavior. Every processor, except the master processor, sorts the values in a dedicated part of the shared memory. The master processor at the beginning writes random values to its shared memory and at the end, it merges the sorted parts and it saves the sorted values in a different non-shared memory. For the sorting algorithm, a simple bubble sort is used. In the figure 9, values of sent messages of the simulation are shown. When the two-state synchronization protocol is used, the count of the writing messages is again at the maximum. The reason is the same as in the previous application. When the three-state synchronization protocol is used the count of sent messages is notably lesser. This is because every sorting simulator has its own dedicated part of shared memory. That means that when a particular processor is sorting, then it reads and writes to its own part of shared memory and it does not have to send any message.

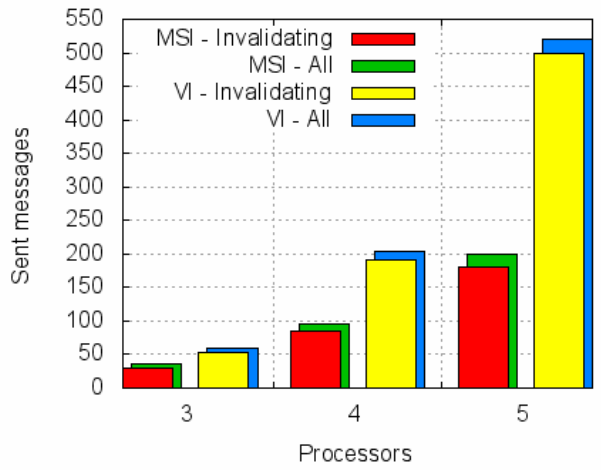

Fig. 9: Graph of the count of messages for sorting algorithm

Now, the advantage of the three-state synchronization protocol is clear. It always depends on the application's character, just how much advantage we will gain, but nevertheless, an advantage is there in 99.9% of cases.

5 Communication protocol

A communication protocol is a collection of rules and everyone who wants to communicate in some communication group must follow the rules of this group. It prescribes a body and a form for messages, which has to be done if a message is lost, which part can communicate with the others etc.

In the Lissom project, a XML fragment is chose as the body and the form of the message. Every message carries a type and it could carry some data. Lissom uses the three-layer architecture, i.e. we have presentation, middle and simulation layers. Each of these layers can run on other hosted systems. In the figure 10 it is shown who with who can communicate. In the bottom is layer of simulators. In this layer simulators can communicate among others simulators (broadcast character) and can communicate with the middle layer. The middle layer takes care of accepting requests from presentation layer and sending back data which will be displayed to user. Also it takes care of controlling and getting data from simulators. Furthermore it takes care of creating simulators and other instruction set tools like assembler or disassembler. When and what is created or run determines the presentation layer, more precisely a user who uses the presentation layer. That means that when the user writes a source code in particular application specific instruction-set, then that user will activate compilation in the presentation layer (by pushing a button or writing a command in command line). The presentation layer accepts this user requirement and creates a message for the middle layer. The message will contain a type, which tells us that we want to compile something, and data: the source code. The middle layer accepts this message and run an assembler and as a parameter gives the source code. After compilation, the middle layer lets us know about success or failure, so it creates a message with particular type and send it back to the presentation layer. This approach, which was described a short while ago, is commonly used for all requirements and replies in the three-layer architecture. The presentation layer has several forms. It could be simple web interface, a command line or it could be plugins to the IDE Eclipse (figure 11, 12). All these user interfaces communicate with the middle layer with the same protocol. This concept has implicit multiuser character. We could have one computer for the middle layer, some computers for simulation, and one or more computers for the designers. On these computers, different kinds of user interfaces can run. Users are identified by theirs names so everyone has their own space to creating projects.

If a user wants to run a simulation, then he has to set some environment variables in the presentation

layer in a user interface. These variables denote which computers are used for simulation, the program which will be simulated etc. Because simulators are created on a computer which is used for the run of the middle layer, it is obvious that on computers the same operation system must be used for simulation. When the simulation is started, the simulators are copied to computers by the middle layer according an environment variable. Coping is done automatically by the middle layer using the protocol for secure copy (the application scp), which was chosen because of multiplatform independence and security features. So, every simulator can run on another computer and this, as was mentioned in the introduction, brings some advantages. After the simulation, the middle layer asks every simulator to send statistic and resource values and this data is interpreted by some kind of user interface in the presentation layer. More information about statistics can be found in the section 6.

So, in one extreme case is that we have three computers for the layers of the communication protocol and n systems for n simulators. The other extreme case is that everything runs on only one computer.

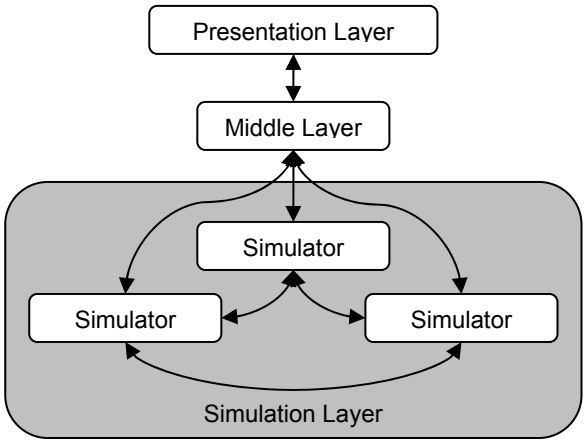

Fig. 10: Three layer model

6 Simulation results and profiling

After the simulation or during the simulation, the designer needs to get values of resources for validation and verification, since he needs to check that the application does the correct computation. In some cases, he also needs profiling information. This kind of information is available only after the simulation and can be basically divided into two types. With the first type, we understand information that is relevant to the processor itself. With the second type, we understand information that is relevant to the simulated application. That means that the designer needs information about the utilization of the instruction set, how often particular instructions access certain resources, and so on. He also needs information about source code coverage, which parts of the source code are bottleneck points, which parts of the source code are the most difficult in terms of time, etc. All this information helps to optimize either the processor or the application which is run on the processor. So, after the profiling, we can find that the utilization of some elements in the design are insufficient and we can remove them from the design without any negative consequence. Or, if we found the bottleneck point in the design, then we can rework this point and remove the insufficiency.

In the Lissom project, every resource has counters for read, write and executable access. Resources, like memory, have counters for every cell. The only exception is cache. The most important information about cache is how many times we hit or miss the value of the resource in the cache. That is why, for cache resources, only hit and miss statistics are logged.

Every time when an operation uses a resource, it is logged. In standard simulation, there is no relation between what operation accesses what resource. That means that only counters are increased and nothing more is logged. Thus the relations are available during profiling only.

If we have a MPSoC, then the simulators can run independently on one or more hosts. They collect theirs own values of counters and resource values during the simulation independently. After simulation, they send their results to the middle layer. This layer puts together all results and resends this collection to the presentation layer. This layer manages to display them in proper way. The same situation occurs if we do the profiling. Every simulator sends profiling information to the middle layer. The middle layer collects them then resends the collection to the presentation layer. In this case, the results of the simulation and the profiling information are displayed independently for every ASIP, i.e. in the IDE Eclipse we would have several tabs.

The example of results of the simulation is shown in the figure 11. One can see a part of a memory called “program” and a few registers and their values. The results from the simulation are

available during the simulation too, mostly used when the designer debugs the application. When an execution is stopped at a breakpoint, the designer can see actual values of resources and their statistics, which is necessary for usable debugging.

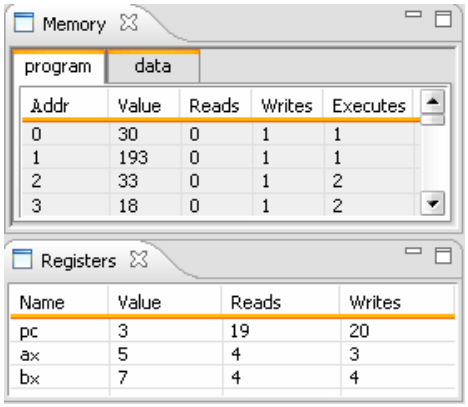

Fig. 11: Screenshot of simulation results

The profiler is based on the upgrade of the behavior on edges in the machine code finite automaton, i.e. a special behavior is inserted after or before basic behavior. The special behavior takes care about a watching system activity and takes care of logging. More information about the profiler can be found in [14].

The profiling information is formed from statistical information, e.g. the instruction set coverage or top five instructions by memory accesses, etc. Furthermore, the mentioned relation between instruction and resource accesses is available. That means that we know how many times each instruction accesses a particular resource.

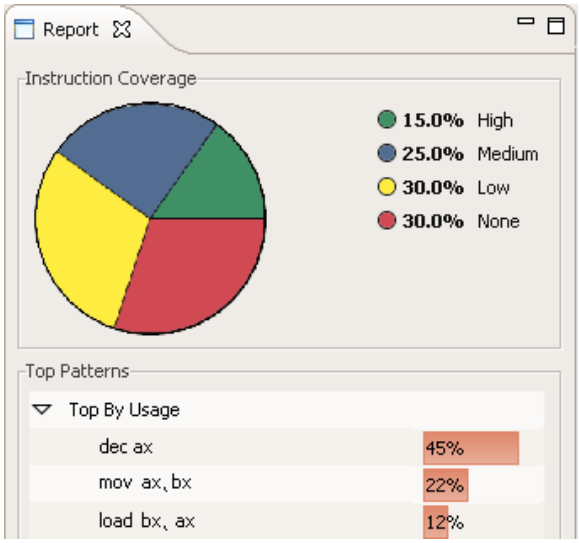

Fig. 12: Screenshot of the profiling

If we have more ASIPs on MPSoC, then the situation is the same like in the simulation without the profiling, i.e. the profiling information is available independently for each ASIP. The profiling information, unlike of results of the simulation, is available on request and after the simulation only. This is needed because retrieving this information during the simulation makes the simulation much slower.

The example of the profiling information is show in the figure 12. As one can see, there is a graph of instruction set coverage and part of the top 5 by the count of the execution statistic.

7 Conclusion

In this paper, a way how to design the new ASIPs which can be used on a multiprocessor system on a chip was proposed. The structure of one simulator, which represents one ASIP on MPSoC, was described. Further a way how to simulate and synchronize a shared resource among ASIPs on this MPSoC was proposed. The architecture description language ISAC is used for description of ASIPs on MPSoC architecture. This language has features that identify and access shared resources. The synchronization protocol is based on modified MSI protocol which is used in multiprocessors systems with shared memory. Because of that, this protocol is not dependant on any third-party libraries which would care of accesses to shared resources. It allows us to have very simple distribution of simulators to hosted systems in a cluster of computers which are used for simulation without many requirements on resources of these systems. The communication protocol is based on three layer architecture which allows us to run these layers on independent computers. Both the synchronization and the communication protocols use sending messages as form of communication act. The body of a message is constructed by the XML language. Further, the profiling and particular statistical information, which can be used for the optimization, were described.

All of these create a powerful, robust and flexible environment for the designer to develop a new application specific instruction-set processors and multiprocessor systems on a chip which uses these processors.

References:

[1] Hruška, T., *Instruction set Architecture*, Internal material, FIT VUT Brno, 2004.

[2] Moskovčák, J., *Design of Communication Protocol for Generic Simulators of Microprocessor*, Master thesis, FIT VUT Brno, 2007.

[3] Novotný, T., *Transformation of the Microprocessor's Description Language to the Hardware Description Language*, Master thesis, FIT VUT Brno, 2007.

[4] Hoffmann, A., Meyr, H., Leupers, R., *Architecture Exploration for Embedded Processors with LISA*, Kluwer Academic Publischers, ISBN-4020-7338-0.X2, 2002.

[5] Dvořák, V., *Architektura a programování paralelních systémů*, VUTIUM Brno, 2004.

[6] Open MPI Team, *Open Source High Performance Computing*, available at www: http://www.open-mpi.org/.

[7] Wieferink, A., Kogel, T., Nohl, A., et al.: *A Generic Tool-Set for SoC Multiprocessor Debugging and Synchronization*, IEEE Int. Conf. On Application-specific Systems, Architectures and Processors (ASAP), The Hague (Netherlands), 2003.

[8] Braun, G., Hoffmann, A., Nohl, A., Meyr, H.: *Using Static Scheduling Techniques for the Retargeting of High Speed, Compiled Simulators for Embedded Processors from an Abstract Machine Description*, Aachen University of Technology, Institute for Integrated Signal Processing Systems, Germany, 2005.

[9] Hruška, T., Kolář, D., Lukáš, R., Zámečníková, E.: *Two-way Coupled Finite Automaton an Its Usage in Translation*, WSEAS Int. Conf. On Circuits, New Aspects of Circuits, Greece, 2008.

[10] Economou, D., Mouratidis, N., Lykakis, G., Tavoularis, A., Kostopoulos, A., Manousaridis, A., Konstantoulakis, G.: *An Innovative SoC Design for Broadband Residential Applications*, WSEAS Transaction on Communication, Greece, 2004.

[11] Reynaga, R., Yupanqui, F.: *Two-dimensional cellular automata of radius one for synchronization task*, WSEAS Transaction on Computers, Greece, 2003.

[12] Masařík, K., Přikryl, Z.,: *Simulace časového modelu mikroprocesoru*, Internal material, FIT VUT Brno, 2008.

[13] Patterson, D., Hennessy, J.: *Computer Organization and Design*, Morgan Kaufmann Publisher, ISBN-978-0-12-370606-5, 2007.

[14] Přikryl, Z.: *Návrh a implementace profileru*, Internal material, 2008.