Instruction Set Emulation for Rapid Prototyping of SoCs*

Jürgen Schnerr[1], Gunter Haug[1] and Wolfgang Rosenstiel[1,2]

[1] FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10–14
76131 Karlsruhe, Germany

[2] Universität Tübingen
Sand 13
72076 Tübingen, Germany

{jschnerr, haug, rosenstiel}@fzi.de

Abstract

In this paper the application of Instruction Set Emulation for rapid prototyping of SoCs will be presented. The emulation works in a way that both the software and the hardware behaviour of the emulated processor core is reproduced cycle accurately. This requires the use of hardware and software components. The hardware component consists of a board containing a VLIW processor and FPGAs. The software component is an instruction set simulator of the core running on the VLIW processor. The FPGAs are used for emulating the SoC bus of this processor core. This way the simulation of an instruction set of a processor core has been extended to a real emulation of this core that can be used for rapid prototyping.

1. Introduction

The increasing complexity of *Systems-on-Chip* (SoC) raises the problem of validation of such complex hardware/software systems. One possibility of validation is prototyping, but often building a prototype using a processor core of the target system is not possible. This can have multiple reasons.

One reason is that only few processors cores are available as bond-out-versions (i.e. chips that are wired ready for connection). Furthermore, regarding a design using a very new processor core, it is possible that only the specification of this processor is available, but the hardware itself is not available yet.

Another reason is that processor cores for SoCs usually use special SoC busses. These busses are different from ordinary processor busses, because they do not have certain restrictions of those. For example, they do not have to deal with a limited number of pins available.

*This work has been partially supported by the BMBF/MEDEA+ project SpeAC A508 under grant number 01M 3049 A.

Therefore simulation and/or emulation have to be used, but up to now significant restrictions exist.

Existing approaches for the validation of complex SoCs suffer either from low accuracy or low performance. Using hardware/software co-simulation a hardware simulator is coupled with an instruction set simulator. Like in conventional hardware simulation the system is validated in a test environment (testbed) [10]. This approach provides a very high accuracy, but for many applications a bad performance. Other disadvantages are that the correctness of the system is only validated against the testbed, and the approach does not deliver a prototype.

The emulation of the processor core using conventional (e.g. FPGA-based) systems [7] provides maximal accuracy but especially for modern processor architectures it is so costly, that it cannot be used in practice. Besides this disadvantage the emulation using FPGAs often suffers from low performance.

The creation of a prototype using another type of processor and recompilation of software normally also leads to a functioning laboratory sample, but the timing of the target system is reproduced very inaccurately. Furthermore, another processor than the target processor uses another communication model, therefore a validation of the connected hardware is only possible in a limited way.

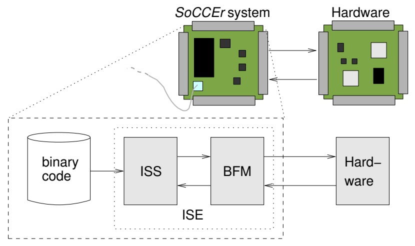

Figure 1. Overview

In the following, an approach to *Instruction Set Emulation* (ISE) that allows a cycle accurate reproduction of the target processor is presented. In this approach an emulation system named ***SoC** processor **C**ore **E**mulato**r*** (*SoCCEr*) will be introduced (shown on the left side of Figure 1).

This system allows a direct connection of further hardware to the emulated processor core in contrast to an *instruction set simulation* (ISS) running on a workstation. The hardware environment used is based on the *RAVE* hardware platform described in [4, 5].

The main part of the hardware of the *SoCCEr* system consists of a VLIW processor and FPGAs. The ISS which executes the instructions of the software of the processor core on the SoC is running on this VLIW-processor whereas the *bus functional model* (BFM) [12] of this processor core was implemented using FPGAs. In the following, this implementation of the BFM will be called *Bus Emulation Module* (BEM).

Using four 96 pin edge-connectors, other hardware like the WEAVER boards [4] can be connected directly to the emulated bus (shown on the right side of Figure 1). This way, a real prototype of the SoC can be built.

In opposite to the approaches mentioned above, this method emulates the target processor fast *and* accurate. Moreover, the possibility to "trade" accuracy for speed and vice versa is given.

The remainder of this paper is structured as follows: Section 2 shows a short overview on the state of the art in instruction set simulation and implementation of a bus functional model. The implementation of the ISS and the BFM in the *SoCCEr* system is shown in Section 3 and Section 4. Finally benchmark results for the ISS are presented in Section 5.

2. State of the art

The approach to *Instruction Set Emulation* presented in this paper consists of two parts, as shown in Figure 1: the ISS and the implementation of the BFM. In usual implementations both parts are implemented in software on a workstation. In contrast to these implementations, FPGAs are used on the *SoCCEr* system for a fast implementation of the BFM. These FPGAs are closely coupled with the VLIW processor running the ISS and allow a direct connection of hardware to the emulated bus. This allows fast bus accesses of the emulated processor to the attached hardware.

2.1. Instruction Set Simulation

The technology of ISS is widely spread in the area of all-purpose computers. It is, amongst other things, used for the migration of software and the evaluation of processor architectures. In this case there are different reasons for the usage of ISS, and the corresponding systems have other properties. However, the basic approach stays the same.

Two different approaches to ISS have emerged: the use of an interpreter and the use of binary translation which means static or dynamic compilation.

An interpreter is a reproduction of the fetch-decode-execute-loop of a processor. The big advantage of an interpreter against the compiled simulation is that in theory the simulation can have any desired accuracy. For example, caches as well as pipelining and branch prediction of the emulated processor can be reproduced.

Static compilation translates the binary code of one processor into binary code of another. An introduction into static compilation and an overview on existing static compilers can be found in [2]. The major advantage of the static compilation technique is a very fast execution of the translated code. An important disadvantage of it is that no calculated branches can be executed.

Fortunately these approaches do not exclude each other. Hybrid forms of them can be used which can combine the advantages of both approaches at least partially. One possibility is to use a so-called fall-back interpreter [1] for instructions that cannot be translated.

As the code of an interpreter or static compiler is highly dependent on the processor it simulates, it has to be rewritten for the implementation of every new processor. To avoid this, the processor can be modelled with an *Architecture Description Language* (ADL). Using a description in an ADL, an interpreter or static compiler can be generated automatically. A good overview on ADLs is shown in [3].

2.2. Bus Functional Model

The BFM of a processor is a cycle-accurate model of the bus transactions of this processor. It cannot execute any instructions; this has to be done by the ISS. A definition of the BFM can be found in [8].

In existing approaches like co-simulation, the BFM is a pure software implementation. It is connected through some kind of IPC (e.g. sockets) to a HDL simulator that simulates the connected hardware. These approaches suffer from low performance and also existing real hardware cannot be simply connected to the emulated processor.

3. Instruction Set Simulation

For the implementation of the ISS a combination of a static compiler with a fall-back interpreter was chosen. This was done because static compilation is the fastest known technique for instruction set simulation. In the case of SoCs, many of the disadvantages of static compilation are of no significance. For example, SoCs usually do not make use of self-modifying code, and no invocation of an application by another application can occur. To overcome the remaining disadvantages, the fall-back interpreter can be used.

At the moment the TriCore and the OpenRISC 1200 processors are emulated by the system. The TriCore [6] processor is microcontroller architecture from Infineon. The OpenRISC 1200 [9] is a freely available RISC processor core.

The TriCore processor was chosen due to a common project with Infineon. Using an evaluation board with this processor made it possible to show the performance and the potential of the emulation by comparing it with an existing system. The OpenRISC 1200 was selected because its specifications and its HDL code are publicly available. Both processors are 32 bit RISC load-store architectures containing additional DSP instructions.

The emulation is running on a TMS320C6201 (C6x) processor from Texas Instruments [13]. This is a VLIW processor with eight functional units meaning that up to eight instructions can be executed in parallel. Due to the parallel execution of multiple instructions a very fast execution of the translated code is possible. Also the parallelisation of the code during compile time results in a highly predictable code execution during run-time.

As the emulated processor is described in an XML file, a later expansion to other processor cores is possible. This file contains, amongst other things, the description of the semantics of each processor instruction.

For the interpreter the semantics of the instruction is described in C code as the interpreter consists of C code that is generated automatically out of the description. Using an assembler description for the interpreter would not give a better performance because of the slow decoding of the instructions.

For the static compiler the semantics of the instructions is modelled in an assembler intermediate representation. This intermediate representation is quite similar to the assembler language of the C6x-processor, but it allows symbolic identifiers, and it is not restricted by the resources of a real processor. In contrast to other existing description languages, the semantics of the instruction is described relatively close to the hardware of the target processor. This leads to faster code than generic solutions which can generate code for different processors.

3.1. Static compilation

Static compilation translates the binary code of one processor into functional equivalent binary code of another. This translation of binary code is described in the following.

The static compiler reads both the object file to be emulated and the processor description. For reading the different object file formats like COFF, it uses appropriate modules. In the same way a module is used for reading the processor description.

After that, the instructions have to be decoded and replaced by their equivalents in assembler intermediate code.

Branches with calculated targets are translated into calls to the fall-back interpreter. This interpreter executes the branch instruction and the following code until the next branch instruction with a non-calculated target occurs. Then it jumps back to the compiled code. Therefore the interpreter needs a table containing the target addresses of all branches with non-calculated targets. This table has to be generated by the static compiler. Furthermore, the static compiler has to generate a table of all I/O accesses it can find.

Now the branch targets and the hardware accesses are known and the program can be divided into basic blocks.

When no hardware access takes place, it is possible to optimise over block boundaries. Possible optimisations like loop-unrolling which takes advantage of the parallelism of the VLIW-processor are known techniques in compiler design.

As up to now the blocks are only available in the assembler intermediate representation, they have to be translated into a valid C6x-assembler program. Therefore it is necessary to perform certain transformation steps. These steps include, amongst others, a scheduling within blocks and the binding of symbolic registers to processor registers.

As this static compiler translates code written for processor cores running on SoCs, it has to deal with some problems which are not occurring in the code translation using conventional static compilers. In the following three sections, these problems will be described and solutions will be presented.

3.1.1. Handling of memory addresses

A system-on-a-chip for instance can contain different kinds of memory. These can be ROM containing the program running on the processor core in addition to RAM and EEPROM for data. The ISS has to provide this memory for the emulated core.

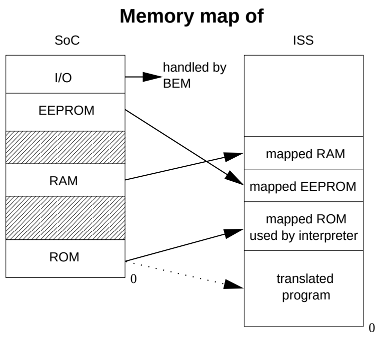

Figure 2. Mapping of memory addresses

This leads to the problem that usually the memory addresses containing data in the SoC are different from memory addresses which are available to the ISS. The solution to this problem is that addresses of load/store instructions to SoC memory have to be mapped to corresponding addresses in the memory of the ISS. Figure 2 shows an example of such a mapping of memory addresses to the emulated system.

Load/store instructions to I/O addresses have to be replaced by accesses to the BEM. Here no address translation takes place. More about theses accesses is explained in Section 4.2.

It is also possible to connect memory to the emulated bus. This can be done in order to validate the read/write accesses of the emulated processor core to the memory. In this case the remapping of the addresses would be done by the address decoding hardware, but for performance reasons usually only I/O accesses are handled by the BEM whereas memory accesses are handled by the ISS.

For a load/store instruction using absolute addressing it is no problem to map the address used by this instruction to the memory map of the ISS and to translate this instruction to instructions making the appropriate I/O accesses respectively.

In the remaining cases where the address cannot be statically determined a dynamic solution is used. The load/store instructions have to be translated to a code sequence which checks the source/target address during runtime. Depending in which range the address resides, an I/O access or a memory access is done. In case of the memory access the address has to be mapped to the corresponding address of the emulation system during runtime.

3.1.2. Synchronisation

For hardware accesses it is important that the code is executed cycle accurately. Cycle accurate execution means that the execution of a translated code piece has to be taken place in exact the same cycle count as the corresponding piece of the original binary code.

Usually there is an external common clock for the processor and the attached hardware, but in this case there is no such common clock. Instead, the emulated processor has to generate the clock cycles for the attached hardware itself. This is done with the help of a hardware device in the BEM. By accessing this device clock pulses can be generated for the attached hardware. The device and how it is accessed is described in Section 4.1.

The most obvious approach to a cycle accurate emulation using this device would be the following: For each instruction it is determined how many clock cycles it uses. Then it is translated to an access to the synchronisation device generating this amount of cycles followed by the translation of the instruction.

Using this approach, a significant potential for optimisation would remain unused. The reason for this is that the attached hardware only gets knowledge about the semantics of the program when there is an I/O access. Until this access the hardware only knows the number of past cycles.

Therefore complete blocks of instructions between I/O accesses along with the appropriate accesses to the synchronisation device can be generated. An example of such a block is shown in Section 4.1.

As the code in these code blocks does not contain hardware accesses, it can be optimised by the static compiler. One resulting possibility for an optimisation is the parallisation of instructions in order to achieve an optimal utilisation of the execution units of the VLIW processor.

4. Bus Functional Model

The BFM is implemented using four FPGAs that connect the C6x processor running the ISS and the external hardware which can be connected directly to the edge-connectors of the FPGA-board.

At the moment an emulation of the WISHBONE [11] bus which is a non-proprietary SoC bus is implemented. Other SoC busses like AMBA or CoreConnect are quite similar. For the implementation of these busses the HDL code of the FPGAs has to be changed. The interface between the BEM and the ISS is realised using registers which can be accessed by the ISS.

The synchronisation between hardware and the emulated software, the handling of bus accesses, and other functions of the BEM will be described in the following sections.

In addition to the described features, the BEM also implements interrupts and configuration ports.

4.1. Hardware-software synchronisation

The hardware connected to the bus has to receive a clock signal. This clock signal has to be the clock signal of the emulated processor. It cannot be the clock signal of the C6x processor. For example, an instruction requiring one clock cycle on the emulated processor may require when it is compiled more than one clock cycle on the C6x processor.

The solution for the problem is that there is no clock that generates the clock for both the attached hardware and the emulated processor. The clock cycles for this hardware are generated by the emulated processor itself.

This is done by accessing the so-called synchronisation device in the BEM using memory mapped I/O. The five least significant bits of the address contain the information about how many clock cycles have to be generated. A store instruction to the address of this range generates the specified amount of clock cycles for the attached hardware.

The following block shows an example. To the left the code for the OpenRISC 1200 is displayed to the right the translation for the C6x.

```
                              STW A1,*B14[S_DEV+3*4]
l.movhi d0, 654               MVKH .S1 654, A1
l.movhi d1, 2322           || MVKH .S2 2322, B2
l.and   d0, d0, d1            AND .L1 A1, B2, A1
                              LDW *B14[S_DEV], B0
```

The code block on the left requires three clock cycles. The translated code block on the right has to require the same amount of clock cycles. This is done by accessing the synchronisation device (`STW` instruction[1]) forcing it to generate three cycles for the hardware. After this access the execution of the translated code sequence is started and will run in parallel with the generation of the clock cycles.

After the C6x has completed the `AND` instruction, the synchronisation device either has completed generating the cycles or is still busy generating the cycles. If it has completed it is no problem to run the next code sequence and generate clock cycles for it. If is still busy the processor has to be stopped until the generation of the clock cycles is finished. This is done by a read access (`LDW` instruction) to the synchronisation device after the execution of the code sequence. The synchronisation device stops the processor by not giving a ready signal to this `LDW` instruction until all cycles are generated.

4.2. Handling of bus accesses by the BEM

Read and write accesses to the BEM are accesses to registers implemented in the FPGAs. Figure 3 shows the structure of this BEM containing all signals and registers mentioned in the next two sections. In this figure the data and the address bus of the C6x processor are labelled as d and a, respectively.

4.2.1. Read access

The following example shows how a read access to a memory address of the hardware is translated. In the following code blocks, again OpenRISC 1200 code is displayed on the left whereas C6x code is displayed to the right.

```
r2 contains address     A1 contains address
l.lwz r1, 0(r2)         STW A1, *B14[ADR_OUT]
                        LDW *B14[DAT_IN], A2
```

The translated read access consists of two C6x instructions. The first one writes the address into the `ADR_OUT` register. No output on the bus happens up to now. The bus cycle is started by the following load instruction. This load instruction to the address of the `DAT_IN` register starts an FSM in the FPGA that handles the bus access and stops the execution of code on the processor during this access.

In the previous section it was shown how the clock cycles for the attached hardware are generated for code not containing I/O accesses. This was done using the synchronisation device. In the case of hardware accesses the clock cycles for the hardware are generated by the FSM. The generated clock signal is called EMUCLK.

If there are no wait cycles, a read access to the hardware consists of two cycles. On the first rising edge of EMUCLK, the signals of the control bus and the address are written

[1]In this and the following examples, the register `B14` is the base register for the hardware access, and the embraced value is the offset which is added to this base address. The symbol “||” before an instruction means that this instruction and the previous one are executed in parallel by the C6x processor.

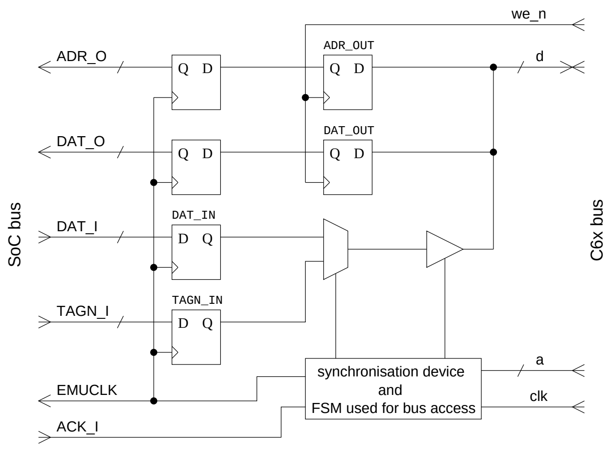

Figure 3. Structure of BEM

to the bus. The address is written to the bus by copying it from register `ADR_OUT` to an address output register that is connected directly to the bus. This prevents that the address is on the bus before the signals on the control bus are set.

On the following trailing edge, the slave that is connected to the bus takes the address and delivers the corresponding data via the bus. After that it sets the acknowledge signal (ACK_I) to show that the bus access was successful. If the slave is not fast enough, it can demand the generation of additional bus cycles by not giving an acknowledge signal.

If an acknowledge was given by the slave, the next rising edge latches the data on the data bus into the register `DAT_IN` of the FPGA, and the C6x processor is signalled that the data is ready. It copies the data from the `DAT_IN` register into a processor register (in this case the `A2` register) and continues with normal execution.

4.2.2. Write access

A write access on the bus is handled basically the same way as the read access described in the previous section. If we take the following example:

```
r2 contains address,    A1 contains address,
r1 contains data        A2 contains data
l.sw 0(r2), r1          STW A1, *B14[ADR_OUT]
                        STW A2, *B14[DAT_OUT]
                        LDW *B14[TAGN_IN], A3
```

Here not only the address but also the data that has to be stored has to be written in a register in the FPGA. After that a read access to the register `TAGN_IN` takes place. The load starts the FSM in the FPGA which stops the processor and makes the write access on the bus in quite a similar way as the read access. When the bus access is finished, the processor loads the tag information of the bus transfer into a register and continues with normal execution.

5. Results

In order to show that the *SoCCEr* system has the potential to emulate systems as fast or nearly as fast as the real system, its performance was compared with that of a TriCore evaluation board.

Two example programs were used to show the performance of the TriCore ISS running on the *SoCCEr* system. Both programs were run in an interpreted and in a compiled version on the C6x processor running at 200 MHz. The results are shown in Figure 4. The performance of the real TriCore processor running at 48 MHz is shown on the right side of the figure. The performance is measured in millions of TriCore instructions per second.

The first of the two examples calculates the greatest common divisor of two numbers. It is control flow dominated. The other example is an elliptical wave filter, a data flow oriented program.

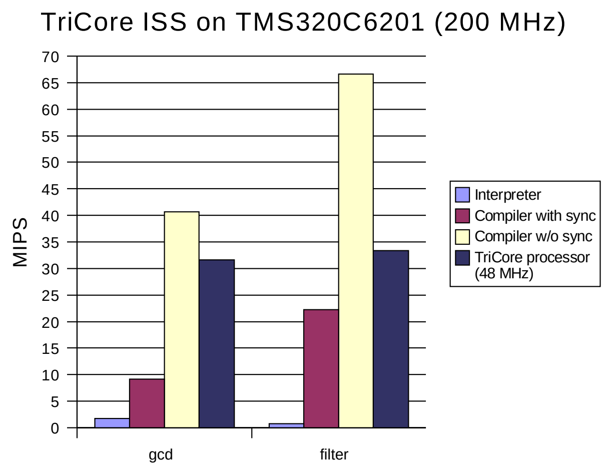

Figure 4. Results running TriCore ISS

The code running in the interpreter uses synchronisation. The compiled code was generated in two versions: one with synchronisation and one without.

As expected before, the interpreted versions have a very low performance. The reason for this is the fetch-decode-execute loop consisting of many branch-instructions. These branches are executed very slowly by the C6x-processor. Another reason for the low performance is that accesses to the synchronisation device cannot be bundled like in the compiled versions.

6. Conclusion

The extension of an existing board using a VLIW processor for the emulation of synthesised RT-level description to a platform for the emulation of processor cores has been shown. In contrast to existing solutions, this approach offers a possibility to emulate the processor within the real environment. This avoids the bottleneck of coupling an ISS running on a workstation using RPC to simulated or emulated hardware in an HDL emulator.

For the implementation of the ISS, the combination of interpretation and static compilation was chosen. The results show that this offers a fast execution of the emulated code without having the disadvantages of a pure static compilation.

Finally, a hardware implementation of a BFM for a SoC bus using FPGAs has been shown.

References

[1] K. Andrews and D. Sand. Migrating a CISC Computer Family onto RISC via Object Code Translation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 213–222, 1992.

[2] C. Cifuentes and V. Malhotra. Binary Translation: Static, Dynamic, Retargetable? In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 340–349, 1996.

[3] A. Halambi, P. Grun, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic Software Toolkit Generation for Embedded Systems-on-Chip. In *Proceedings of the International Conference on VLSI and CAD (ICVC)*, pages 107–116, 1999.

[4] G. Haug, U. Kebschull, and W. Rosenstiel. VLIW Based Emulation of Digital Designs with the RAVE System. In *Proceedings of the International High Level Design Validation and Test Workshop (HLDVT)*, 1999.

[5] G. Haug, U. Kebschull, and W. Rosenstiel. A Hardware Platform for VLIW Based Emulation of Digital Designs. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, page 747, 2000.

[6] Infineon Technologies Corp. *TriCore Architecture v1.3 Manual*, 2000-01 edition, 2001.

[7] J. Gateley and M. Blatt *et al.* UltraSPARC^{TM}-I Emulation. In *Proceedings of the Design Automation Conference (DAC)*, pages 13–18, 1995.

[8] M. Keating and P. Bricaud. *Reuse Methodology Manual For System-on-Chip Designs*. Kluwer Academic Publishers, third edition, 2002.

[9] D. Lampret. *OpenRISC 1200 IP Core Specification*. OpenCores, September 2001.

[10] J. A. Rowson. Hardware/Software Co-Simulation. In *Proceedings of the Design Automation Conference (DAC)*, pages 439–440, 1994.

[11] Silicore Corporation. *Specification for the: WISHBONE System-On-Chip (SoC) Interconnection Architecture for Portable IP Cores*, January 2001.

[12] L. Séméria and A. Ghosh. Methodology for Hardware/Software Co-verification in C/C++. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 405–408, 2000.

[13] Texas Instruments Incorporated. *TMS320C62xx CPU and Instruction Set Reference Guide*, July 1997.