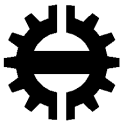

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology

PEKKA JÄÄSKELÄINEN

INSTRUCTION SET SIMULATOR FOR TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Subject approved by Department Council
6th April 2005

Examiners: Prof. Jarmo Takala
Prof. Tommi Mikkonen

PREFACE

The work for this thesis was completed in Institute of Digital and Computer Systems of Tampere University of Technology (TUT) in 2004-2005 for a codesign software implemented as part of the Flexible Design Methods for DSP Systems (FlexDSP) project funded by the National Technology Agency and Nokia.

I would like to thank Professor Jarmo Takala for allowing me to work in this interesting project, for sharing his hardware expertise, and for his improvement ideas for the thesis. Thanks to PhD Andrea Cilio for sharing his expertise in the field of TTAs and for cooperating in the design of the Simulator. I am also grateful to the whole TCE software team for being motivated and for working hard to produce quality software. Finally, I would like to thank my family and my dear Terhi for their support and for being there.

Tampere, August 17, 2005

Pekka Jääskeläinen

TABLE OF CONTENTS

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Institute of Digital and Computer Systems

Jääskeläinen, Pekka Olavi: Instruction Set Simulator for Transport Triggered Architectures

Master of Science Thesis: 59 pages, 2 appendix pages

Examiners: Prof. Jarmo Takala and Prof. Tommi Mikkonen

Funding: The National Technology Agency

September 2005

Keywords: transport triggered architecture, instruction set simulation

Due to specific requirements of some of embedded system applications, general purpose processors are usually not the most optimal ones for the task at hand. Thus, there is a need for application-specific processors, which are tailored for the application and requirements at hand. However, processor design is a demanding task. Therefore, the processor design flow needs to be automated as completely as possible.

TTA Codesign Environment (TCE) is a toolset that provides a semi-automated processor design flow, which includes "design space exploration", which is a process that helps to find an optimal processor architecture for the given application semi-automatically. The processor paradigm utilized in TCE design flow is called transport triggered architecture (TTA). TTA is a relatively simple and highly modularized processor architecture which allows easy customization. One of the leading ideas of TTA is to move complexity from the processor hardware to the compiler. Consequently, the most complicated tool in TCE is the compiler. Instruction set simulation is mainly needed in verifying the compiler output and in design space exploration.

The project completed for this thesis consisted of design, implementation, and verification of an instruction set simulator for TCE. The thesis describes the main requirements and most important software design decisions of the TCE instruction set simulator. In addition, the verification of simulation correctness is described and performance benchmarks are presented. Finally, several improvement ideas and brief plans for implementing them are presented.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

Digitaali- ja tietokonetekniikan laitos

Jääskeläinen, Pekka Olavi: Käskykantasimulaattori TTA-suorittimille

Diplomityö: 59 sivua, 2 liitesivua

Tarkastajat: Prof. Jarmo Takala ja Prof. Tommi Mikkonen

Rahoitus: Teknologian kehittämiskeskus (TEKES)

Syyskuu 2005

Avainsanat: transport triggered architecture, käskykannan simulointi

Yleiskäyttöiset suorittimet soveltuvat huonosti joihinkin sulautettuihin sovelluksiin näiden sovellusten asettamien erityisvaatimusten takia. Sovelluskohtaiset suorittimet räätälöidään kunkin sovelluksen asettamien erityisvaatimusten mukaan. Koska suorittimien suunnittelu on vaativaa ja virhealtista, räätälöinti halutaan tehdä mahdollisimman automatisoidusti, erityisohjelmistoja käyttäen.

TTA Codesign Environment (TCE) on ohjelmistokokonaisuus, joka mahdollistaa puoliautomaattisen suunnitteluavaruuden läpikäynnin. Suunnitteluavaruuden läpikäynnillä tarkoitetaan annettuun sovellukseen parhaiten soveltuvan suoritinmuunnelman etsimistä automatisoidusti. TCE:ssä käytettyä suoritinarkkitehtuuria kutsutaan nimellä "transport triggered architecture"(TTA). TTA on suhteellisen yksinkertainen ja erittäin modulaarinen suoritinarkkitehtuuri, joka mahdollistaa suorittimien räätälöinnin vaivattomasti. Yksi TTA:n perusperiaatteista on siirtää monimutkaisuutta laitteistosta suoritettavaan ohjelmakoodiin. Tästä johtuen monimutkaisin työkalu TCE:ssä on kääntäjä. Käskykantasimulaattoria käytetään varmentamaan kääntäjän ulostulo sekä auttamaan kustannusarvioiden laskemisessa suunnitteluavaruuden läpikäynnissä.

Tätä diplomityötä varten tehty työ koostui TCE:lle tehdyn käskykantasimulaattorin suunnittelusta, toteutuksesta ja testauksesta. Diplomityö kuvaa simulaattorin päävaatimukset ja tärkeimmät ohjelmistotekniset ratkaisut. Lisäksi kuvataan tapa, jolla simulaattorin toimivuus varmennettin, ja esitetään suorituskykytestien mittaustulokset. Lopuksi esitetään muutamia parannusehdotuksia simulaattoriin sekä alustavat suunnitelmat niiden toteuttamiseksi.

LIST OF ABBREVIATIONS

ADF	Architecture Definition File
ALU	Arithmetic-Logic Unit
DSP	Digital Signal Processor or Digital Signal Processing
FU	Function Unit
GCC	GNU Compiler Collection
GCU	Global Control Unit
GNU	Gnu's Not Unix
HDL	Hardware Description Language
IDF	Implementation Definition File
MSM	Machine State Model
OSAL	Operation Set Abstraction Layer
RF	Register File
RISC	Reduced Instruction Set Computer
TCE	TTA-Based Codesign Environment
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architecture
VHDL	Very high speed integrated circuit Hardware Description Language
XML	Extensible Markup Language

1. INTRODUCTION

Processors designed for embedded systems often have stricter requirements than the general purpose-processors used in desktop computers have. Embedded system processors may place stronger limits on such aspects as power consumption, performance, or production price. Furthermore, embedded processors often execute only a limited set of programs.

Unlike general-purpose processors in desktop computers, which run many different programs, application-specific instruction-set processors (ASIP) are codesigned with the type of software they are going to execute in the target product. The ASIP instruction set can be extended with special instructions that could help the application in performing its task. Respectively, instructions having no benefit to the application can be discarded from the instruction set, thus simplifying the processor. For example, during video encoding using certain algorithms, a substantial count of discrete cosine transforms (DCT) are calculated for each encoded video frame. These transforms may take the majority of the processor clock cycles of the encoding task. When the processor and application are codesigned, i.e., are simultaneously "tailored" for the given task, it is possible to design processors which provide special instructions for computing DCT, thus allowing the same functionality be achieved with fewer processor cycles.

However, designing new processors is a demanding task. Especially, the verification of the processor correctness is often time-consuming. Therefore, the task of designing new application-specific processors is usually assisted by a software toolset. One such toolset, called TTA Codesign Environment (TCE), is currently in development at Tampere University of Technology. The processor paradigm used in the toolset is transport triggered architecture (TTA). TTA is a modular and simple processor architecture which allows flexibility through customizable set of processor resources like function units, register files, and transport buses.

TCE provides semi-automated design space exploration, which helps the designer to find the optimal processor architecture for the given application. Major part of the de-

sign space exploration time is spent simulating each evaluated architecture variation. Simulation provides the design space explorer with statistics of each evaluated processor, thus sets the direction to which the design space exploration should proceed and makes it possible to pick the optimal architecture according to the given requirements. The Simulator is also an important tool while verifying and debugging the most complicated tool in the toolset, the instruction scheduler, also known as the compiler backend.

For this thesis, an instruction set simulator, later referred to as Simulator or TCE Simulator, was developed for TCE. This thesis describes the main requirements, design, and verification procedure of the simulator. In addition, as the thesis is the first publication made of TCE, it serves also as a brief introduction to the TCE toolset.

The thesis is divided into following chapters. Chapter 2 introduces the environment for which the Simulator was implemented by describing each tool in the toolset briefly. Additionally, the transport triggered architecture paradigm is introduced, mainly in the programmer's point of view. Chapter 3 describes the main requirements that were placed for the Simulator. Chapter 4 summarizes the high level operational principles of the Simulator, without getting into implementation details. Chapter 5 provides more detailed description of the high-level software design and implementation of the Simulator by describing the responsibilities of the major software modules and the implementation of the main simulation loop in detail. Chapter 6 describes how the Simulator was tested and lists the results of performance benchmarks. Chapter 7 introduces several future improvement ideas for the simulator along with brief implementation plans for them. Chapter 8 concludes the thesis.

2. CODESIGN ENVIRONMENT SUPPORTING PROCESSOR CUSTOMIZATION

In order to develop software targeted to any processor, developers need a development toolchain supporting the target. Such toolchain usually includes at least a high-level language compiler, a simulator, a debugger and a linker.

Toolchains that support customizable processors clearly are more complicated from the ones that are targeted to only a single processor. In retargetable toolchains, each tool, like the compiler and the simulator, needs to be generalized to the level that they can be used seamlessly with any processor architecture variation supported by the system.

Constructing a compiler for customizable processor architectures is particularly demanding task. Since the resources of the target processor to which the code is generated are not set, the resource allocation, optimization, and parallelization algorithms become more complicated.

In order to reduce the problem of supporting customized processor architectures in the toolchain, the processor architecture supported by the system is usually limited by a well-defined processor architecture template. The processor architecture template defines limits for the types of processors supported by the system. The template is often provided for users in form of a processor/machine description language. One such language is LISA [1]. LISA allows describing processors in a language resembling a programming language. The description is used by the retargetable LISA toolchain to adapt to the described architecture.

Processor design space exploration is a process in which an optimal processor architecture is searched in a set of processor architectures. The architecture is varied automatically by a design space exploration algorithm which removes and adds resources to the architecture and evaluates the effects of the modifications by simulating each processor variation. In order to make automatic modifications to the processor architecture straightforward, it is desirable that the chosen processor architecture is flexible in structure.

2.1 Transport Triggered Architectures

Transport triggered architecture (TTA) is an application-specific instruction-set processor (ASIP) architecture template that allows easy customization of processor designs.

This chapter describes TTA mainly in the point of view of a programmer, which is required to understand the operation of the Simulator. The structure of TTA is explained briefly to give definitions for the concepts of TTA without going deeper in details. The architecture is described in more detail in [2] and [3].

Structure

TTA processors are built of independent *function units* and *register files*, which are connected with *transport buses* and *sockets*. Figure 1 represents a simple TTA processor with two function units, one register file, and a *control unit*.

Each function unit implements one or more *operations*, which implement functionality ranging from a simple addition of integers to complex, arbitrary user-defined computation. Operands for operations are transferred through function unit *ports*.

Each function unit may have an independent *pipeline*. In case a function unit is *fully pipelined*, a new operation that takes multiple clock cycles to finish can be started in every clock cycle. On the other hand, the pipeline can be such that it does not accept new operation start requests while an old one is still executing.

Data memory access and communication to outside of the processor is handled by using special function units. Function units that implement memory accessing operations and connect to a memory module are often called load/store units.

Control unit, in case of TTA, can be seen as a special function unit which controls the execution of programs running in the processor. For this, control unit has access to the instruction memory in order to fetch the instructions to be executed. In order to allow the executed programs to transfer the execution (jump) to an arbitrary position in the executed program, control unit provides control flow operations. Control unit usually includes a transport pipeline, which consists of stages for fetching, decoding, and executing program instructions.

Register files contain general-purpose registers, which are used to store variables in programs. Like function units, also register files have input and output ports. The

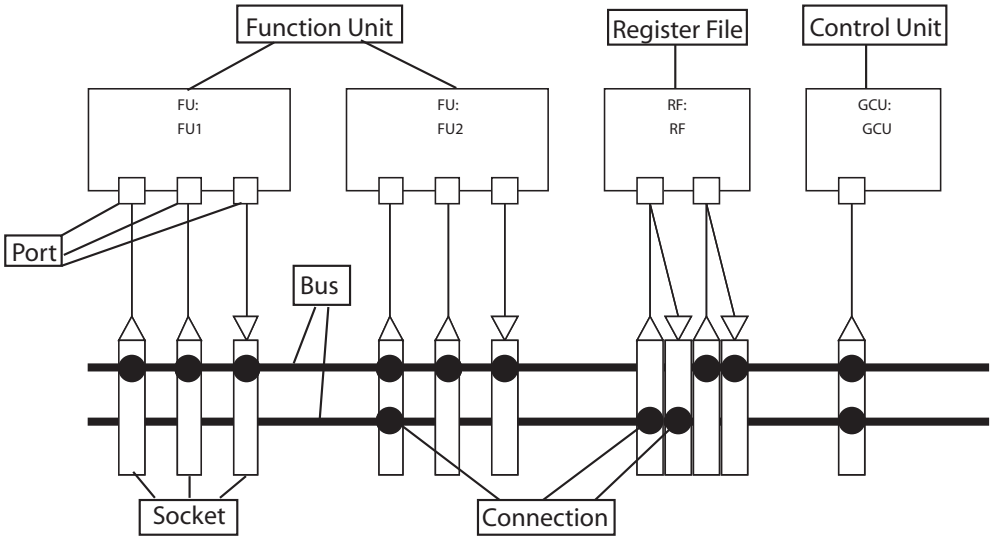

Figure 1. *Simple example TTA processor.*

number of read and write ports, that is, the capability of being able to read and write multiple registers in a same clock cycle, can vary in each register file.

Interconnection network consists of *transport buses* which are connected to function unit ports by means of *sockets*. Due to expense of connectivity, it is usual to reduce the number of connections between units (function units and register files). A TTA is said to be *fully connected* in case there is a path from each unit output port to every unit's input ports.

Sockets provide means for programming TTA processors by allowing to select which bus-to-port connections of the socket are enabled at any time instant. Thus, data transports taking place in a clock cycle can be programmed by defining the source and destination socket/port connection to be enabled for each bus.

Conditional execution is implemented with the aid of *guards*. Each data transport can be conditionalized by a guard, which is connected to a register (usually a 1-bit *boolean register*) and to a bus. In case the value of the guarded register evaluates to false (zero), the data transport programmed for the bus the guard is connected to is *squashed*, that is, not written to its destination. *Uncoditional* data transports are not connected to any guard and are always executed.

It is evident that TTA is suitable for customization as it is possible to define a new TTA processor by simply defining function units, operations implemented in each function unit, register files, count of registers in each register files, count of buses, and connections between units.

Programming

In more traditional processor architectures, a processor is usually programmed by defining the executed operations and their operands. For example, an addition instruction in a RISC architecture could look like the following.

```
add r3, r1, r2
```

This operation adds the values of general-purpose registers r1 and r2 and stores the result in register r3. Coarsely, the execution of the instruction in the processor probably results in translating the instruction to control signals which control the interconnection network connections and function units. The interconnection network is used to transfer the current values of registers r1 and r2 to the function unit that is capable of executing the add operation, often called ALU as in Arithmetic-Logic Unit. Finally, a control signal selects and triggers the addition operation in ALU, of which result is transferred back to the register r3.

TTA programs do not define the operations, but only the data transports needed to write and read the operand values. Operation itself is triggered by writing data to a *triggering operand* of an operation. Thus, an operation is executed as a side effect of the triggering data transport. Therefore, executing an addition operation in TTA requires three data transport definitions, also called *moves*:

```
r1 -> add.1
r2 -> add.2
add.3 -> r3
```

The second move, a write to operand two, triggers the addition operation, which makes result of addition available to be read for the next move.

Sequential TTA programs are generic sequences of general purpose register and operation operand *moves*. The moves of the sequential code are not scheduled to be executed in any target architecture. For this reason, sequential programs are sometimes called *unscheduled programs*.

A simple sequential code incrementing the value of a general purpose register r3, and decrementing the value of r4 until the values become equal, is given in the following example:

```
 1:  1 -> r3
 2:  500 -> r4
 3:  r3 -> add.1
 4:  1 -> add.2
 5:  add.3 -> r3
 6:  r4 -> sub.1
 7:  1 -> sub.2
 8:  sub.3 -> r4
 9:  r3 -> eq.1
10:  r4 -> eq.2
11:  eq.3 -> bool
12:  !bool 3 -> jump.1
```

In lines 1-2, *immediates*, that is, constant numbers, are transferred to general purpose *registers* r3 and r4. Lines 3-4 set the input *operands* for *operation add*. In line 5, the result of the addition is read from the output operand back to register r3. After that, *sub* operation is executed in similar manner. After addition and substraction, operation *eq* is used to compare the values of r3 and r4 for equality. The result of the comparison is transferred to a *boolean register*, which is used in conditional execution in the next line. The last line, control flow operation *jump* is triggered in case the value of the *boolean register* evaluates to false. That is, the program execution is transferred back to line 3 in case the values of the r3 and r4 are not equal. In this example, the operand 2 of *add*, *sub*, and *eq* operations, and operand 1 of the *jump* operation are *triggering*, that is, when the operand is written to, the operation starts computing the results, or in case of the *jump* operation, executes the functionality needed to transfer the program control to the target instruction.

Conditional execution in TTA programs is implemented with *guarded moves*. In the previous example, the move in line 12 is guarded by the negation of the value of register *bool*.

Parallel TTA programs are defined as sequences of TTA instructions. Each TTA instruction defines a set of *moves*. A move defines endpoints for a data transport taking place in a transport bus. For instance, a move can state that a data transport from function unit F, port 1, to register file R, port 2, should take place in bus B1. In case there are multiple buses in the target processor, each bus can be utilized in parallel in the same clock cycle. Thus, it is possible to exploit instruction level parallelism by scheduling several data transports in the same instruction.

Parallel programs are always targeted to some TTA architecture. Consequently, they are also referred to as *scheduled programs*. Parallel programs are final in the sense that

it is possible to generate the program bit image representing the parallel code and run it in a real processor hardware that implements the targeted architecture.

In the next example, the code from the previous example is scheduled to be executed in an example architecture with two buses and two function units. The used example architecture is illustrated in Fig. 1. One of the function units implements the operation *add* and the another implements *sub* and *eq*.

```
1:  1 -> RF.3,                  500 -> RF.4
2:  RF.3 -> FU1.add.1,          RF.4 -> FU2.sub.1
3:  1 -> FU1.add.2,             1 -> FU2.sub.2
4:  FU1.add.3 -> RF.3,          FU2.sub.3 -> RF.4
5:  RF.3 -> FU2.eq.1            RF.4 -> FU2.eq.2
6:  !FU2.eq.3 2 -> GCU.jump.1
```

In this example, both buses of the target machine are almost fully utilized. Each instruction of the scheduled program except the last one performs two parallel moves. In the first line, two registers are initialized in parallel. The registers r1 and r2 of the sequential code are assigned to registers three and four of the register file *RF*. Because the operations *add* and *sub* are in two independent function units, *FU1* and *FU2*, respectively, it is possible to compute the results of the both operations at the same time. In line 5, both the operands of the operation *eq* are transferred in the same clock cycle, as the connections of the target architecture allow it. Finally, the branching move of the last line is guarded directly by the result of the equality operation. Note that, in addition to having guards that are watching a value of a register, it is possible to have guards that watch the value of a function unit output port, as is the case in this example's last instruction. The control unit that implements the control flow operation *jump*, is named *GCU*, as in Global Control Unit.

The assembly notation used in the example refers to function unit ports through operation operands. It could also be possible to refer directly to function unit ports instead, but this version of notation is chosen for clarity. For example, *FU1.add.2* refers to the port of function unit *FU1*, to which the operand two of the operation *add* is bound. The alternative way to refer to the port could be *FU2.P2*, in case the programmer knows that the operand is bound to a port named *P2* in that function unit. The TTA assembly syntax of TCE is fully explained in [4].

Customizable Operation Set

One of the customization points for TTA is the operation set. It is possible for the designer to add a new operation to the target processor which implements arbitrary functionality. This allows, for example, to convert longer chains of operations to a single custom operation execution.

A short example might clarify this idea. Let us assume than an algorithm includes lots of subtractions and additions of same input operands, thus the sequential code would include sequences like this:

```
r1 -> sub.1
r2 -> sub.2
sub.3 -> r3
r1 -> add.1
r2 -> add.2
add.3 -> r4
```

Now, the designer of the TTA system sees that a piece of code including a sequence like this is ranked high in the profiling data, that is, a major part of the execution time is spent running the code. Therefore, he decides to create a new custom operation, *addsub*, which computes both the sum and the difference of the operands it receives and places the difference in the first output operand (operand three) and the sum in the second (operand four). The new custom operation can be used to convert the previous code to the following:

```
r1 -> addsub.1
r2 -> addsub.2
addsub.3 -> r3
addsub.4 -> r4
```

Getting rid of the two moves might not seem much, but it might provide bigger savings in the long run if the sequence is executed in a tight loop with only a few instructions. Furthermore, the same optimization strategy of converting sequences of operations into a single custom operation can be applied to chains of operations of virtually arbitrary length.

Programmer Visible Operation Latency

The leading philosophy of TTAs is to move complexity from hardware to software. Due to this, several additional hazards are introduced to the programmer. One of them is the programmer visible operation latency of the function units. Timing is completely a responsibility of programmer. Programmer has to schedule the instructions such that the result is not read too early or not too late. There is no hardware detection to lock up the processor in case a result is read too early. For example, let us say that the example architecture of Fig. 1 has an operation *add* with latency of 1, and operation *sub* with latency of 3. When triggering the *add* operation, it is possible to read the result in the next instruction (next clock cycle), but in case of *sub*, one has to wait for two instructions before the result can be read. The result is ready for the 3rd instruction after the triggering instruction.

Reading a result too early results in reading the result of a previously triggered operation, or in case no operation was triggered previously, the read value is undefined. On the other hand result must be read early enough to make sure the next operation result does not overwrite the current result in the output port. This is especially a problem in case the function unit is pipelined and can start new operations while old operations are pending. In the following example, FU2 that implements operation *sub* with latency of three is pipelined:

```
1:  1 -> FU2.sub.1,    2 -> FU2.sub.2
2:  1 -> FU2.sub.1,    3 -> FU2.sub.2
3:   [waiting for the result]
4:   [the result of the first triggered operation is ready]
5:  FU2.sub.3 -> RF.1
```

In this case the result of the first subtraction is overwritten with the result of the second subtraction before the instruction at line five is executed. The result of the first substraction is totally ignored, which indicates most probably a program error, as it usually makes no sense to trigger an operation without using the result. Exceptions to this are *operations with state*. In such operations, it might make sense to trigger an operation, just for the sake of its side effects.

A common example of an operation with state is operation *acc*, a simple accumulator operation. When triggered, it adds the given value to its internal register and makes the new value visible in the output operand. An example sequential code which uses an accumulator operation is given in the following example.

```
1:  1 -> acc.1
2:  1 -> acc.1
3:  acc.2 -> r1
```

Accumulator's internal register is incremented twice. Thus the value written to register *r1* in line 3 is two, assuming the initial value of the accumulator's internal register is zero.

Operation latency of zero is not usually supported in case of TTAs, because the output port is usually a register, thus needs a clock edge to update its value. Technically, implementation of zero-latency operations could be possible by using only combinatorial logic [5] in the function unit implementation. In that case, result of an operation can be read only once, in the same clock cycle the operation is triggered:

```
1:  1 -> FU2.sub.1,  2 -> FU2.sub.2  FU2.sub.3 -> RF.1
```

Since the output port *FU2.sub.3* is not a register, its value depends only on operation inputs. When inputs to the operation are changed, the result changes immediately.

Branching

Another user-visible latency in case of TTAs is the latency of control unit's instruction pipeline. TTAs usually have a three-stage instruction pipeline. Programmer sees the result of pipelined instruction execution in form of *delay slots* after branch instructions. Delay slots are due to the fact that at the point the branch instruction reaches the execution stage, new instructions after the branching instruction are already fetched and decoded in the pipeline. It would be a waste of effort to discard these instructions, that is, "flush the pipeline". It is common to have delay slots also in more traditional processor architectures, but the count of programmer visible slots is often limited to one. Principles of traditional processor pipelines are described in detail in [6]. An example illustrating the visibility of instruction pipeline in a TTA program follows.

```
1:  1 -> GCU.jump.1
2:  1 -> RF.3,          500 -> RF.4
3:  RF.3 -> FU1.add.1,  RF.4 -> FU2.sub.1
4:  1 -> FU1.add.2,     1 -> FU2.sub.2
```

In this example, the instruction pipeline produces three delay slots after branches. Due to the delay slots, all the instructions in the example are executed in the given order in a neverending loop.

2.2 TTA-Based Codesign Environment

MOVE framework is the first toolset for codesign of TTA systems [7]. It was originally developed at Delft University of Technology in the Netherlands. Further development and maintenance has taken place at Tampere University of Technology since 2002.

MOVE is a working toolset but its bad software architecture makes it difficult, almost impossible, to extend and use it to experiment new ideas in the field of TTAs. The development of the MOVE framework started at the beginning of 1990's. At that time, C++, the programming language used in implementing the MOVE framework, was still relatively new and the compiler, GCC, used in the project did not yet fully implement even its core features [8]. The result of this can be seen in MOVE code: some of the generic data structures and algorithms are implemented using C macro definitions, which make maintenance of some parts of the code difficult; even worse, the C++ language is practically abused at some parts of the code base. Some parts of the software are written in a way that the resulting code is not guaranteed to work the same way when compiled with different compilers and compiler optimization switches. Finally, MOVE was extended and developed by different researchers to perform the functionality they needed in their research topic without paying enough attention to how those extensions were done.

Due to the previous problems in the original MOVE source code, a project aiming to a complete rewrite of a TTA codesign environment, was started in 2002 at Institute of Digital and Computer Systems of Tampere University of Technology. The main focus on the design of the new framework is on expandability and flexibility, allowing easy experimenting of research ideas on transport triggered architectures. The project name of the new framework is TTA-Based Codesign Environment (TCE). In contrast to MOVE, TCE has been developed in controlled manner. For example, a set of automated system and unit tests are provided to catch regression bugs. In addition, the entire toolset is compiled and tested each night automatically in several different environments with varying operating systems and processor types to ensure portability of the code.

Processor Architecture Template

Processors in TCE are defined by using a file format called Architecture Definition File (ADF) [9]. The file format acts as a template for TTA processors supported by TCE.

It should be noted that ADF is only for defining architectures of the TTA processors. In this case, architecture means the details of processor which are visible for the programmer. Implementation details such as signals not visible to programmers are not part of architecture.

There are some differences in TCE's TTA template compared to the one of MOVE's. Some of these complicate the simulated TTA processor model and affect simulation speed. Most important improvements, which have influence on Simulator's complexity, are the following.

Complex function unit model with independent pipeline models for different operations on a single FU.

In MOVE, each FU may have only one pipeline, even though the FU may implement multiple operations. In TCE, each operation implemented by an FU may use a pipeline model of its own. For example, it is possible to have an addition operation with latency of two and a substraction operation with latency of four, both fully pipelined, in the same FU.

General support for multiple address spaces.

MOVE allows maximum of only two address spaces for data per processor architecture. The properties of the address spaces are fixed. TCE supports fully defining arbitrary number of address spaces for the designed processor, with detailed descriptions for each of the address space. For example, the width of the minimum addressable unit of address space is fully customizable in TCE.

Function unit ports and register file registers can be of any bit width.

MOVE limits bit widths to 1 for boolean registers, 32 for integers, and 32 or 64 for floating point registers. TCE does not have this limitation: integer width can be anything between 1 to 32. The upper limit might be later extended to 64 or 128 bits. The overflow caused by writing wider integers to smaller bit widths has to be simulated by zeroing the extra bits of the integer.

Support for operations with state.

MOVE does not support operations with state. In TCE, it is possible to use state data in custom operations.

Restrictions to Sequential Code

Sequential code in TCE is used mainly as an input from the frontend compiler to the TCE toolset. The sequential code generated by the frontend compiler is independent from the target architecture to make the job of the compiler backend, the instruction scheduler, easier. In order to make sequential code as architecture independent as possible, TCE places following restrictions [4] on the moves of sequential programs.

1. **No operand to operand moves.**

   Moves that transfer the result of an operation directly to an input of an operation are not allowed. For example: *add.3 -> sub.1*. This kind of moves place restrictions to the target architecture: to be able to support a move like this, the target architecture needs to have a connection from the result operand to the input operand.

2. **Only absolute instruction addressing.**

   There are two control flow operations in the base operation set: *call* and *jump*. Both take absolute instruction address as an input. The address operand of a control flow move may be a register or an immediate value (a constant). Relative control flow operations, e.g., jumps to an offset from the current instruction address, are not supported in sequential code.

3. **Only jump operation can be guarded.**

   No other moves than moves to operand of *jump* operation can be guarded. This restriction simplifies the dataflow and control flow analysis of the sequential code while scheduling the program to be executed in the target architecture.

Design Flow

The design flow of TCE is similar to the one of MOVE framework. The design flow can be divided into four phases: *Initialization* phase, which provides input sequential program and initial processor architecture; *Design Space Exploration*, which provides semi-automatic means for finding an optimal processor configuration for the application at hand; *Code Generation and Analysis*, which is either a step in design space exploration or a manual process for scheduling and analyzing the program running in

the target processor; and *Processor and Program Image Generation*, in which the final products of the design flow are generated. The design phases are discussed in the following sections.

Initialization

The first phase of the design flow is illustrated in Fig. 2. The initial sequential code input to TCE is generated by a 3rd party *frontend compiler*. If the program is provided in multiple compilation units, *TPEF linker* can be used for linking them to a single *TTA Program Exchange Format* (TPEF) binary file. The starting point processor architecture for design space exploration, or the final target architecture for scheduling can be defined by using a graphical user interface, called *Processor Designer*.

Frontend compiler is regarded to be a 3rd party application because it is not shipped with the rest of the TCE toolset. Because the supported base operation set is well defined, it should be possible for a 3rd party to port any kind of frontend to produce TCE supported sequential TTA code. Currently, an old GCC compiler [10] version 2.7.0, ported from MOVE framework is used to produce sequential code input to the toolset. At the moment, the frontend has only C language capabilities, which should be enough as it is the most commonly used high-level language for programming embedded systems.

From this phase, the designer of the TTA system usually enters design space exploration phase for semi-automatic optimal processor configuration discovery. Alternatively, designer may run the instruction scheduler and instruction set simulator manually for his initial architecture. This way designer can inspect the statistics produced by the Simulator and modify the architecture to suit better the application's needs. Such procedure is called *manual design space exploration*.

Design Space Exploration

Design space exploration is a process in which several variations of an user-defined starting point architecture are simulated and cost estimated. The goal is to automatically find an optimal architecture for the application at hand. Currently, the algorithm for design space exploration is the same that is used in MOVE framework. It is described in more detail in [11].

The design space exploration process is illustrated in Fig. 3. In a nutshell, *Explorer*

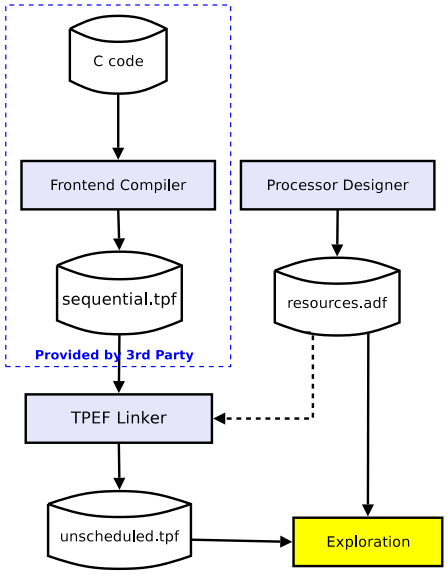

Figure 2. *Initial Inputs for TCE Design Flow.*

removes resources from a given starting point architecture and sends the modified architecture to code generation and analysis phase. From this phase, *Cost Estimator* obtains processor utilization data, which by using a predefined processor cost database estimates costs of executing the given program in the modified target architecture. Cost estimates are calculated for the physical area of the processor, consumed energy, and maximum speed. After exploring hundreds of processor modifications, Explorer finds an optimal processor architecture in the design space for running the given program. Because the starting point architecture is defined manually, this process is said to be a semi-automatic process in contrary to fully automatic one. It should be noted that in this phase the program is simulated with each variation of the target architecture, thus parallel simulation is invoked hundreds of times. Sequential simulation is usually executed only once per design flow exploration process, to provide profiling information for the instruction scheduler.

Explorer creates a database of explored architectures and their total costs. In Fig. 3 this database is referred to as *ExpResDB*, as in exploration result database. In addition, *Cost Estimator* outputs an Implementation Definition File (IDF) for each explored architecture. IDF identifies the implementations, that is, the hardware description language definitions used for calculating the costs of the architecture. ADF and IDF together define a *processor configuration* which is required for generating the final processor description.

After *Explorer* has finished traversing through the design space, a graphical application, *Design Browser*, plots characteristics of all explored processor configurations and

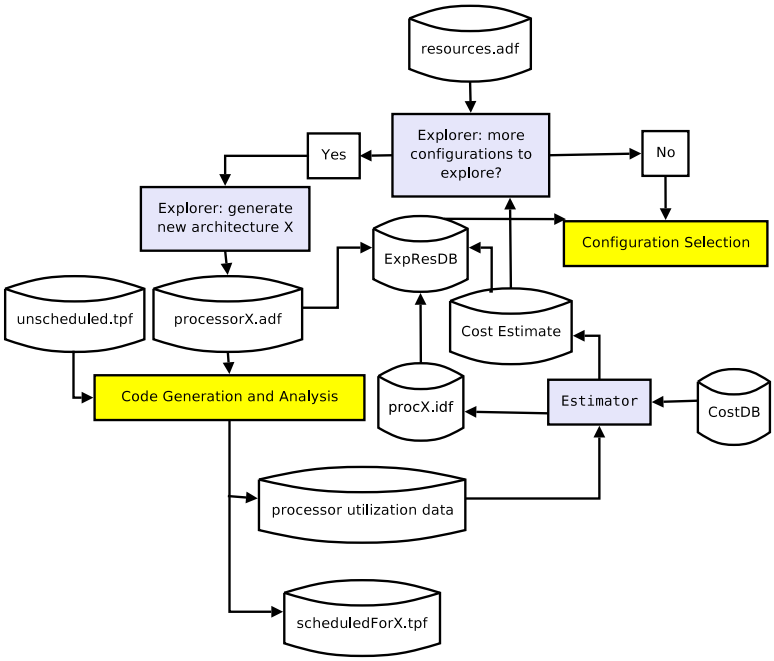

Figure 3. *Exploration.*

allows the user to select individual configurations for closer inspection (Fig. 4). The selected processor configuration is usually taken to *processor and program image generation* phase to generate the final products of the design flow.

Code Generation and Analysis

The code generation and analysis illustrated in Fig. 5 is the most demanding and important part of the TCE design flow. Especially, the *Instruction Scheduler*, also referred

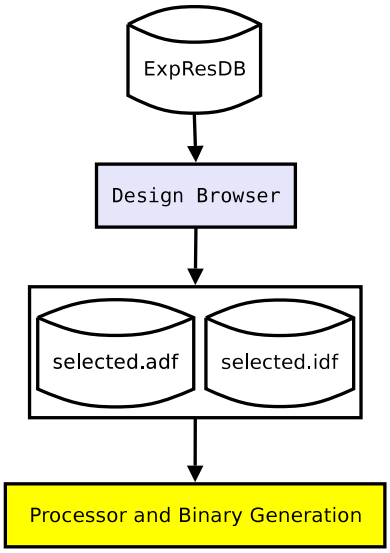

Figure 4. *Processor Configuration Selection.*

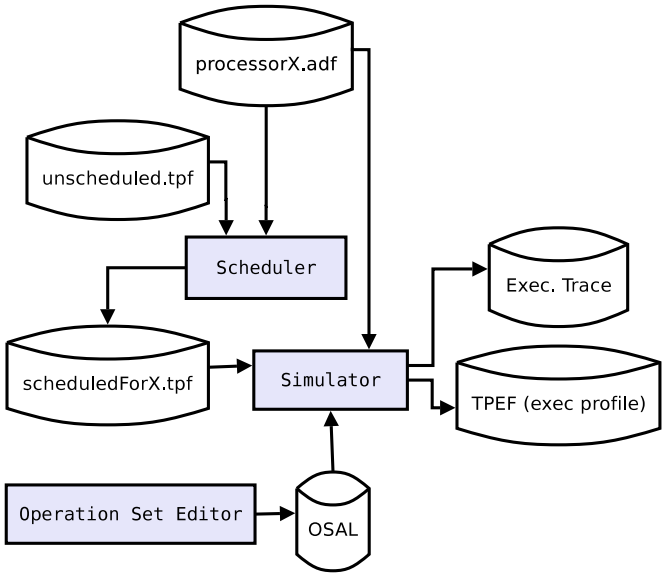

Figure 5. *Code Generation and Analysis.*

to as *scheduler* or *compiler backend*, is a complicated and important application. Its job is to convert sequential programs to parallel programs that utilize the given target architecture as efficiently as possible. Due to manual programming of TTAs being burdensome, and even impossible if semi-automatic design space exporation is used, the quality of the entire toolset is almost directly proportional to the quality of the instruction scheduler.

Given the additional optimization possibilities the TTA provides and the retargetability requirement, it is clear that the algorithms to produce parallelized, optimized parallel code can become complex. Therefore, it was decided that TCE project provides a clean framework for implementing new optimization algorithms for the scheduler. The purpose is to make research and experimentation of advanced algorithms as easy as possible.

Analysis part in this phase includes simulation of the scheduled program to obtain processor utilization data to be used in cost estimation.

Operation definitions of processors designed with TCE are stored in a database called *Operation Set Abstraction Layer* (OSAL) [12]. In addition to static operation data, like the number of input and output operands, OSAL stores simulation behavior of each operation. Behavior definitions are written in C++, and compiled to plugin modules which can be linked dynamically to Simulator in runtime. OSAL operation definitions can be edited and debugged by using a graphical user interface called *Operation Set Editor*.

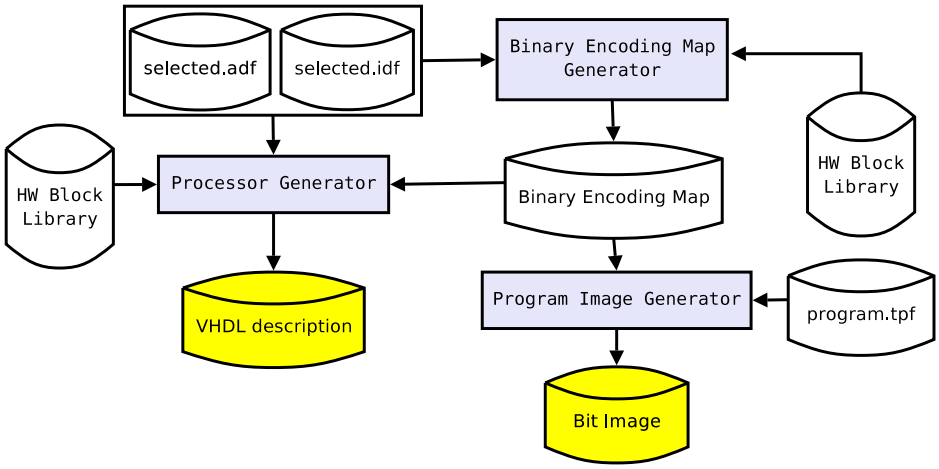

Figure 6. *Processor and Program Image Generation.*

Processor and Program Image Generation

The final phase of TCE design flow shown in Fig. 6 includes generation of hardware description language (HDL) files of designed TTA processors and processor compatible binary images of scheduled programs. Resulting binary image can be uploaded to final processor hardware which is synthesized using the generated HDL files. Processor synthesis is a complicated task out of scope of TCE and, therefore, it is performed with 3rd party applications such as Synopsys tools [13].

Processor Generator takes architecture and implementation definitions of processors as input. Using an user-defined hardware block library, it picks the HDL files that describe the implementation of each machine part of given architecture, generates a "glue" code that connects these blocks, and outputs a synthesizable description of the processor.

Program Image Generator processes a scheduled program in a TPEF file and a *Binary Encoding Map* which tells how each operation code, move source/destination, etc. should be presented in the final binary image. Result is a string of bits representing an executable TTA program. If the binary image is to be compressed, user can provide a compressor plugin that implements the wanted compression algorithm. If a compressor is provided, the only difference to the binary generation process is that the given compression algorithm is applied to the binary image before outputting it.

3. REQUIREMENTS

The difference between an instruction set simulator and processor hardware simulator in the case of TTA is not nearly as big as it is with traditional processor architectures. Since TTA is a simple processor paradigm, the model needed to simulate the instruction set is close to the behavioral model of the entire processor. In fact, it is debatable what is "instruction set" in case of TTA. Since the instructions of TTA programs simply consist of lists of data move descriptions instead of operation codes, the simplest possible TTA instruction set simulator has to actually simulate a significant part of the target processor behavior.

3.1 Retargetability

TCE Simulator must be able to simulate any TTA with any kind of operation set defined by users, without requiring recompilation of the toolset.

Given the simplicity of the architecture, simulating a TTA processor is not a complicated task as such. Complexity is added by the retargetability requirement. Resources of simulated TTA processors are completely defined by users within the limits set by the TTA template of TCE. In addition, the operation set, which in case of TTA means the functionality implemented by function units, is completely customizable by users. Sequential programs, which are not scheduled to any target processor must also be supported by the simulator.

3.2 Accuracy

Accuracy of a processor simulator can be described in two characteristics: the detail level in which the processor functionality is simulated, and the timing accuracy. The simulated processor model in TCE simulator is not a gate level model. The model of the processor is purely architectural: no control signals or any implementation details

are visible. However, the simulation is cycle accurate: the simulation model contains the correct data in each architecturally visible part in all simulated instruction cycles. The accuracy of number of cycles the simulated processor spends in locked state can be improved by defining more accurate simulation models for data memory.

In this thesis, with a TTA instruction cycle is meant the functionality that is performed in a TTA processor during execution of an instruction. Instruction cycle, in case of TTA, is usually of length of the processor's clock cycle. This is due to the fact that data transports of each TTA instruction are usually executed during a single clock cycle.

Functional Accuracy

Simulator does not use the actual binary image that is uploaded to the final processor to simulate the program. Since only the details that are visible to the programmer are needed to be simulated, a higher level abstraction of the simulated program can be used instead. Such simplification can be safely made due to the fact that TTAs commonly use Harvard architecture [6] in which instructions and data are stored in separate memory spaces. Furthermore, the contents of the instruction memory are not usually accessible for the executed program itself. Thus the instruction memory binary image is not visible for programmer, and need not to be included in the instruction set simulation. Finally, since the encoding of the instructions can be fully defined by users, it is possible to produce several different binary representations of the same program. Additionally, the final program bits are often compressed due to the enormous size of TTA instructions, especially in machines with large number of buses [14], which creates additional variation point to the final binary images generated from programs.

Timing Accuracy

Dynamic characteristics, like locked cycles caused by accessing data memories with dynamic latencies or those produced by fetch unit of control unit, are not fully modeled. In case of a lock, processor is almost completely in frozen state, waiting for the lock condition to be resolved. Program execution is not advanced until the condition is resolved. Clock cycles spent in global lock condition are called "stall cycles".

Some support for modeling dynamic data memory latencies is provided in form of letting users to redefine the behavior of the memory model used in simulation. It is

possible to define a memory model that limits the number of simultaneous accesses and emulates dynamic latencies caused by, e.g., caching or dynamic properties of the memory itself. When such a memory model is used, the function unit model produces a global lock condition in case a memory access was not completed in the expected number of clock cycles.

Control unit needs to lock the processor in situations when not enough instruction data can be retrieved from the instruction memory to fill the transport pipeline (Section 2.1). In order to model the stall cycles generated by the fetch unit, it would be necessary to know the exact sizes of each of the instructions and the exact implementation of the parts of the control unit that affect the decision of when to lock the processor. Additionally, the instruction memory, like data memory, might have dynamic properties itself, which would also be needed to be simulated to get the exact stall cycle counts. Such level of details is out of scope of the TCE Simulator, especially since the stall cycles are invisible to the program, and visible only as extra consumed clock cycles. Therefore, it was decided that to get exact stall cycle information of the control unit, it is required to produce a hardware description language (HDL) description of the processor and the real program bits using the tools in the last phase of the TCE design flow. The HDL and program bits can be then simulated in a third party HDL simulator, such as Modelsim [15].

3.3 Simulation Statistics and Traces

Simulator must be able to produce processor *utilization data* to be used in processor cost estimation part of the design space exploration. Specifically, utilization information is used to calculate the energy consumption of a given processor while executing a given program with given inputs. Utilization data consists of number of clock cycles for each processor part in which they were busy. For example, in case of a transport bus, a busy clock cycle means that the bus was written data in that instruction cycle. In case of a function unit, "being busy" means that it was executing at least one operation, in contrast to "being idle", when function unit was passive, producing no new results. Because the estimation algorithm for function unit energy consumption is based on calculating the total energy consumed by operation executions, Estimator needs information of how many times each operation was executed in function units.

Utilization data could be used also in the exploration process directly to decide whether to remove or add resources. It might make sense to duplicate machine parts that are

highly utilized and, on the other hand, get rid of machine parts with utilization percentage close to zero. At present, the exploration algorithm does not use utilization data directly but only through the results of the cost estimator.

Bus trace lists the data contained in each of the bus of the machine in each simulated instruction cycle. Bus trace is useful when verifying the hardware designs of TTA processors. Designs can be verified by comparing a bus trace produced by TCE Simulator to a bus trace produced by a HDL simulator.

Execution time of simulated programs is one of the most interesting statistics from the Simulator. Execution time is the number of clock cycles it takes to execute given program with given inputs. Execution time is also one of the dimensions in the design space exploration.

Some instruction scheduling algorithms profit from program *profiling data*. Profiling data consists of the number of times each instruction is issued in the simulated program. This data can be used to figure out the "hot spots" of the program to be able to direct optimization efforts to the parts of the program that benefit from the optimization the most.

3.4 High Parallel Program Simulation Speed

In design space exploration, hundreds of different processor architecture variations are simulated to figure out how modifications to the starting point architecture affects the end result. Especially, in case the simulated program is long, it might be possible that simulation becomes the bottleneck for the exploration speed. Therefore, one of the main requirements placed to the Simulator is high speed of simulation of parallel code.

Simulation speed of sequential code is not as important, because sequential simulation is usually needed only once per exploration. Sequential code is simulated in exploration to get profiling data for Instruction Scheduler. Since the sequential program, and therefore the profiling data, is same for each explored architecture, it’s enough to simulate the sequential code only once per design space exploration process, and take advantage of the same profiling data in all the different schedules.

3.5 Program Debugging Capabilities

One major use case for Simulator is to be used as a TTA program debugger. Therefore, the Simulator must provide the most important program debugging capabilities like single-stepping the simulated program, stopping simulation at user-defined program addresses (breakpoints), inspecting data memory contents at any point of simulation, and inspecting data in visible machine parts at each simulated instruction cycle. User visible machine parts that can be inspected in the Simulator include registers, function unit ports and buses.

Simulation and debugging capabilities should be possible to control with a Tcl script interpreter [16]. This feature is especially useful when developing automated test benches for new instruction scheduling algorithms.

3.6 Connection to Hardware Simulation

It should be possible to connect TCE Simulator, simulating a TTA processor running a TTA program, to a hardware simulation environment. In such environment, TCE Simulator is treated as a black box without visible implementation details. Only input and output pins of the TTA processor are visible to the rest of the components in hardware simulation.

In this kind of simulation, the simulated hardware is usually a system in which TTA processors are used as "slave processors", accelerating a commonly executed algorithm. Master processors are often general purpose processors which control the slave TTAs. Communication channel between master processor and slave TTA processors can be implemented by means of a shared memory or TTA processors may include special function units that handle the communication. In the former case, the master processor usually stores input data to a known position in the shared data memory, indicates the TTA processor that new data is ready to be processed and waits for the TTA processor to complete processing the data. TTA processor stores the processed output to another location in the shared memory and signals the master processor that the task is done and new results are ready to be read. In latter case, special function units are used to connect the TTA processor to a control or status register shared with the master processor.

4. OPERATIONAL PRINCIPLES

Interpretive and compiled simulation [17] are two common techniques used in instruction set simulator designs. The basic operational principle for an interpretive simulator is to read in the instructions as they appear in the final program image, interpret the bits to figure out which functionality (operations, register transfers, etc.) should be performed, and simulate that functionality by calling a simulation function. Interpretation usually takes a significant part of the simulation time, therefore, interpretive simulators are considered inefficient.

In compiled simulation, the simulated instructions are translated to native instructions of the simulation environment's processor. One implementation possibility of this technique is to generate a high level language program code from the simulated instructions and add the code needed to produce the required simulation traces in it. The resulting source code is compiled with a regular compiler and the produced executable is run as native code in the host environment. This way the interpretation of the target instructions is done offline independently of the simulation instead of doing it during the simulation. The overhead of interpretation is greatly reduced because each instruction is processed only once, contrary to once for each instruction execution as in the simplest implementation of interpretive simulator. Furthermore, because the translated instructions are in a sequence, more optimization possibilities are introduced to the compiler which further speeds up the translated code. Usually compiled simulation is tens, sometimes even hundreds of times faster than interpreted simulation of the same instruction set.

Sometimes instruction set simulators mix ideas from the both techniques, using the compiled simulation idea as an optimization. A popular speedup strategy used in script interpreters, simulators and virtual machines is *just in time compilation* [18]. The idea of it is to translate the simulated instructions to host machine instructions on-demand, at the point they are executed the first time.

TCE Simulator cannot be considered literally to be a traditional interpretive simulator. On the other hand, no translation of instructions to simulator host machine code is

done, thus, it is not a compiled simulator either.

The format of the program simulated by Simulator is not the final binary image that can be uploaded to the instruction memory of the actual processor, but a higher level model which describes the data transports the TTA instructions perform on each clock cycle. The higher abstraction avoids the need to interpret the program bits to figure out the functionality of the simulated instructions.

Preprocessing of the simulated program model can be seen analogous to translation of instructions in the compiled simulation paradigm. The program preprocessor, instead of producing instructions runnable directly in simulator host's processor, produces a new object model which is fast to simulate, leaving as little computation as possible to simulation runtime. The benefit is the same as in compiled simulation: perform as much computation as possible only once for each executed instruction instead of repeating those computations every time an instruction is simulated.

In TCE Simulator, instruction cycle simulation is divided into two parts. First part is the simulation of data transports, that is, the copying of data from source machine parts through the interconnection network to destination machine parts. This part of simulation does not yet simulate the "side effects" of data transports, that is, the execution of triggered operations. Triggered operations and the operation latency are simulated in the second part of the simulation. The second part of simulation models the state of the processor and ensures that results of triggered operations are made visible in function unit result ports in correct instruction cycles.

4.1 Data Transport Simulation

The simulation of transports described by a TTA instruction can be reduced to a loop, which copies data from source processor parts to buses and to destination processor parts as described in each move of the instruction. In order to make transport simulation such simple and efficient, instruction data needs to be preprocessed to a format which contains nothing but references to the variables representing sources, buses and destinations.

The main benefit from preprocessing program data is that as much evaluation as possible is moved offline from simulation runtime. For example, simulating data transports of an instruction which is not preprocessed at all would bring overhead of locating the processor state variables and figuring out the side effects caused by writing to destina-

Algorithm 1 Executing a Processed Instruction

```
1: for all processed_moves M in processed_instruction do
2:     bus_state.Write(source_state.Read())
3:     if not M.IsGuarded() or M.IsGuarded() and guard_reg.Value not 0 then
4:         destination_state.Write(bus_state.Read())
5:     end if
6: end for
```

tion processor state variables. In case instructions were not preprocessed, the simulation of data transports would follow roughly the algorithm presented and described in Appendix A.

On the contrary, data transport simulation with preprocessed instructions is straightforward, since all processing of instructions is done before execution and the side effects of the writes are hidden inside the class that models the destination processor part (see Section 5.6 for details).

Algorithm 1 shows the steps a preprocessed instruction takes to simulate data transports. The algorithm simply traverses through a list of preprocessed moves which describe source, bus, and destination variables of each data transport. For each transport, data is copied from source to bus and from bus to destination. If the move has a guard expression and it evaluates to false, the copy from bus to destination is neglected.

4.2 Processor State Simulation

Processor state is simulated by storing the data of transports in variables representing different parts of processor. For example, each function unit port is a variable and so is each general purpose register. In addition to storing state data, processor state simulation includes simulating the functionality that happens as a side effect of writing to the state variables. Such functionality is mainly realized in function units which execute triggered operations and make their results available in their output ports after the operation latency time has passed.

Programmer Visible State

In order to make the simulation as efficient as possible, only the properties of the processor visible to the programmer are modeled. Such properties are the following:

1. function units,
2. transport buses,
3. registers, and
4. delay slots caused by transport pipeline of control unit.

Function units are most complicated parts that need to be simulated. Programmer visible properties of FUs are: input and output ports, behavior of operations, and latency of operations. The contents of the function unit pipeline registers in each simulated clock cycle is not visible to the programmer, but the latency of operation execution caused by the pipelining is.

Transport buses need to be modeled because of the requirement of being able to produce a bus trace. If there was no such a requirement, simulation of buses could be neglected by simulating the moves to happen directly between move end points, without writing the transported data to buses at all. In addition to the word width, bus also has another property visible to the programmer, the extension mode. When simulating a data transfer from a register or a port to a bus that is wider than the source, the written value needs to be either sign extended or zero extended, depending on the user defined ADF property.

Values in general-purpose registers need to be maintained. On the other hand, the register file ports need not be modeled during simulation. The simulation model does not include ports in the register files at all, but register accesses are modeled to happen directly to the registers.

The global control unit is modeled as a special function unit with a special property of instruction pipeline. The time taken from instruction fetch phase to the execution phase is visible to programmer as delayed jumps. Simulator models delayed jumps by using a simple counter which is initialized to transport stage count at the point a control flow operation is triggered and decremented at each instruction cycle advance until it reaches zero. When zero is reached, the program counter is updated to the new value set by the triggered operation.

Function Unit Model

The core of functionality of TTAs is realized in function units. Each function unit implements one or more operations that performs a function to input data and, in most

cases, produces output data. The behavior of function unit operations is visible to programmers in two ways. Clearly, the function, i.e., the behavior of an operation is the most imporant characteristics of the operation visible to programmer. Additionally, due to the user-visible operation latency of TTA processors, the timing of making triggered operation results visible to function unit output ports is as important as the operation behavior itself.

In TCE Simulator, function unit model is divided to two parts: the function unit model proper and operation behavior model. The function unit model is responsible for starting operations and simulating the user visible latency caused by pipelining. The simulation of the functionality of operation is delegated to a separate operation behavior model. The operation behavior model is responsible for simulating the functionality of "operation ideas". Operation as an idea does not include the operation latency, but only the function of the operation, and, for example, the number of operands and results produced by the operation. This kind of sharing of responsibilities makes it possible to use same operation models with different operation latencies. For example, the addition operation ADD is exactly the same regardless of the visible operation latency of the function unit that implements the operation. 2 + 2 is still 4, even if it takes several instruction cycles to compute the result.

5. IMPLEMENTATION

The software architecture of the simulator is divided into three subsystems: *simulator engine*, *base library*, and *user interfaces*. The subsystems and their main modules are illustrated in Fig. 7. The architecture implements the *model-view-controller* (MVC) user interface paradigm [19], which aims to a strong separation of the user interface code from the model logic code.

Simulator engine is the core of the Simulator. The subsystem is further divided to four main modules: *Machine State Model* (Section 5.1), *Simulation Controller* (Section 5.3), *Debugging System*, and *Frontend*. Debugging System is used to implement the debugging features. Different user interfaces control the simulation through the Frontend, which is a realization of **Façade** design pattern [20]. The main purpose of this class is to hide details of the simulation behind an easy-to-use interface, allowing the internals of the simulator engine to be changed without forcing the user interface code to be changed. Another motivation for the Frontend is to collect common code from different user interfaces to a centralized location. Currently, two user interfaces are implemented: a graphical user interface and a text-based user interface. The text-based user interface is a scriptable interactive console, which is also embedded in the graphical user interface to provide scripting capabilities.

Base library is a collection of modules that represent the major concepts of TTAs. The concepts are represented as easy-to-use object models and reused throughout the entire TCE toolset. *Operation Set Abstraction Layer* (OSAL) provides access to operation data like operand counts and to operation behavior definitions, which are used to simulate operations. *Memory Model* (Section 5.5) consists of a simple interface for defining data memory behavior. *Program Object Model* (POM) is a static representation of TTA programs. It is not used directly in simulation, but a preprocessed program model is built from it when a simulation is initialized. *TPEF* is a file format for storing TTA programs. The software module that handles TPEF files is called *TPEF Handling Module*. In Simulator it is used as a file parser to load the simulated program from an user-defined file. *Machine Object Model* represents the architecture of a TTA pro-

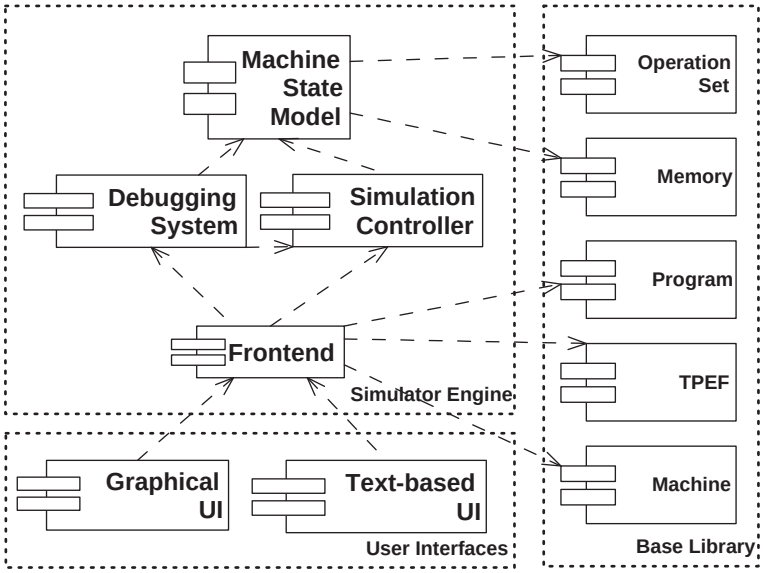

Figure 7. *Overview of the Simulator Architecture.*

cessor (Section 5.1) loaded from an ADF file. This static model is used to build the dynamic simulation model (*Machine State Model*, Section 5.1) for the processor.

5.1 Processor Model

The processor simulation model is called *Machine State Model* (MSM). The main goal while designing MSM was to make it as simple and fast as possible without losing simulation accuracy in the required parts of the processor.

In the point of view of the simulated program, MSM looks like a set of machine parts that can be read from or written to. What happens as a side effect of writing to a port, in case the port is an operation triggering port, is completely hidden inside the model by using inheritance and dynamic binding. This kind of simple model allows to preprocess simulated program instructions to completely resolved instructions which contain nothing by references to the machine state variables the data transports access.

Figure 8 presents a simplified class diagram of the main interfaces and classes of MSM. The elements of the machine state that allow reading and writing implement an interface called *StateData*. The most important classes that implement *StateData*, and thus can act as sources and destinations in data transports are *RegisterState* and *PortState*, which represent general purpose registers and function unit ports, respectively.

The parts of the simulated machine that need to operate when simulated instruction cycle is advanced implement an interface called *ClockedState*. Interface *Clocked-*

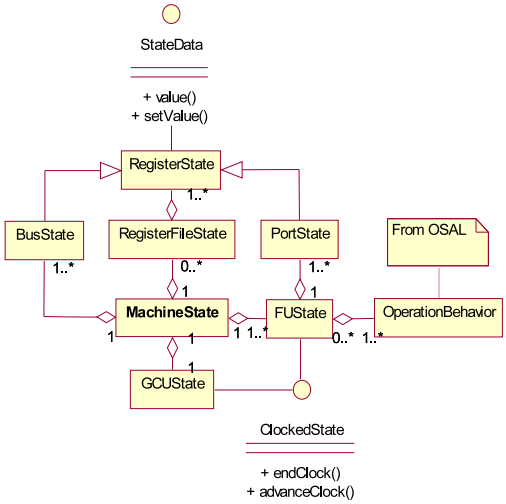

Figure 8. *Machine State Model.*

State provides methods *advanceClock()* and *endClock()*. Method *endClock()* signals the function unit model that all the transports of the instruction are simulated, that is, data of the transports are written to its input ports. Function unit model executes possible triggered operations in this method, and places results to a wait queue that models operation latency.

The simulation of an instruction cycle advance is divided to two methods mainly for interfacing with memory model implementations that place restrictions to concurrent memory accesses. In order to know whether memory accessing operations triggered by an instruction are possible concurrently, memory model needs to know details of all the initiated memory accessing operations initiated by the instruction. After calling *endClock()* of each function unit, which results in initiating possible triggered memory access operations in function units, memory model is asked whether all initiated memory accessing operations are possible concurrently without locking the processor. In case there is no problem, *advanceClock()* is called for each function unit that is waiting for a memory operation result. As a result, the function units that initiated data memory reads receive the requested data from the memory model. In the case the memory model signals that the initiated concurrent accesses are not possible, a global lock is simulated.

MSM is built from *Machine Object Model*, which is an object model for accessing and modifying TTA descriptions stored in ADF files. In case of parallel simulation, a *Machine Object Model* is constructed from a user-defined ADF file. As sequential code is unscheduled, that is, not targeted to any architecture, no input ADF is given by

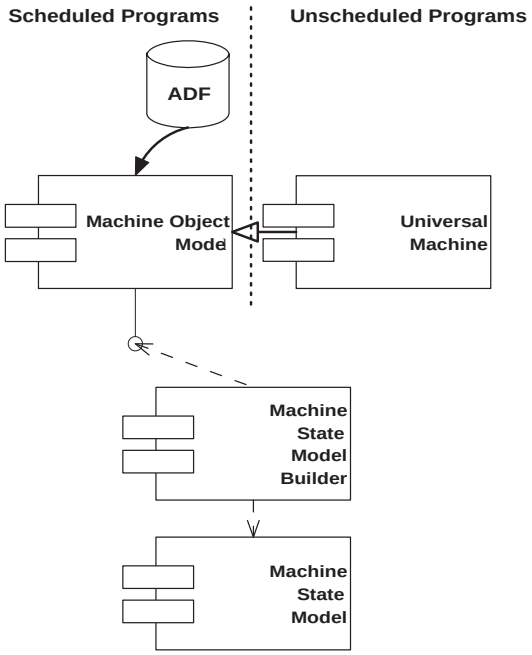

Figure 9. *Building the Machine State Model.*

the user when initializing a sequential simulation. In order to use common simulation code for both types of simulation, a virtual processor model called *Universal Machine* is constructed by the Simulator at the point a sequential program is loaded. Because *Universal Machine* is inherited from the "regular" *Machine Object Model*, thus implements the same interface, the rest of the simulation code can simulate unscheduled code like it was scheduled to an actual TTA processor. Building of the simulation model is illustrated in Fig. 9.

Universal Machine (UM), the virtual TTA processor for simulating sequential TTA code, is built while the sequential program is loaded. After program loading is finished, the *Universal Function Unit* (UFU) of UM contains all the operations the loaded program needs. Latency of all operations in UFU is one, thus result of any operation is always available for the next instruction cycle after the clock cycle in which the operation was triggered. In addition to UFU, UM provides infinite count of integer and floating point registers. It also has a global control unit (GCU) with transport pipeline latency of one, because there are no delay slots in sequential code. Register files and function units are connected with a single bus, thus parallel moves are not possible.

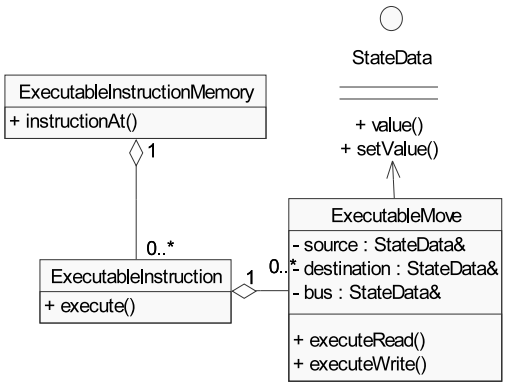

Figure 10. *Preprocessed Simulation Program Model.*

5.2 Program Model

Before starting the actual simulation, the static object model of TTA programs, *Program Object Model* (POM), is converted to a structure that is more suitable for simulation. The resulting model is called *Executable Instruction Memory*.

A class diagram of the model is illustrated in Fig. 10. The frontend class of the model is a simple container class, called *ExecutableInstructionMemory*, which stores the preprocessed instructions. The class representing a preprocessed instruction is called *ExecutableInstruction*. It is a container for *ExecutableMoves* that contain references to state objects of *Machine State Model* that take part in the data transport.

Data transports of an instruction are simulated by calling *execute()* method of the preprocessed *ExecutableInstruction* that represents the instruction. The function calls *executeRead()* and *executeWrite()* for each contained *ExecutableMove*. *ExecutableMove::executeRead()* reads the current data in the source machine part to the bus programmed by the move and *ExecutableMove::executeWrite()* reads the value from the bus and writes it to the destination machine part.

Fig. 11 illustrates the dependencies in the process of building the simulated program model. *POMBuilder* is responsible for building the static object model, *Program Object Model* from given input files that contain the architecture and program descriptions. *Program Object Model* and the processor simulation model, *Machine State Model* are used by *Simulation Program Preprocessor* to build the preprocessed simulation program model.

The process for converting POM *Instructions* to *ExecutableInstructions* is effectively the same as in interpretation part (lines 5-27) of algorithm in Appendix A. The re-

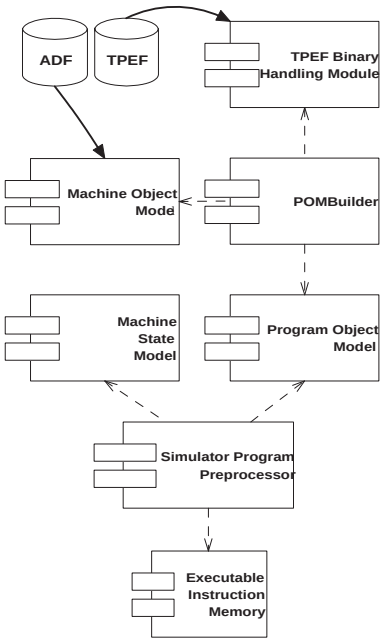

Figure 11. *Building the Executable Instruction Memory.*

solved state references are stored in *ExecutableMove* objects as member variables. The member variables are shown in Fig. 10 with names *source*, *destination*, and *bus*.

5.3 Simulation Controller

The main simulation loop is implemented in a module called *Simulation Controller*. *Simulation Controller* delegates simulation of data transports to preprocessed instructions (*Executable Instruction Memory*) and simulation of processor state to *Machine State Model*.

After initializing the object models used in simulation, *Simulation Controller* is ready to accept simulation commands. There are two main ways to proceed with simulation: single-stepping the simulated program's instructions and running the simulation until instructed to stop.

Method *run()* keeps running the simulation until either the final instruction of the program has been simulated or when another condition for stopping the simulation occurs. Conditions for stopping include user set program breakpoints and explicit simulation interruption through the user interface. In practice, simulation can be interrupted explicitly in the text-based user interface by pressing *ctrl-c* and in the graphical user interface by clicking a *stop* button. Single-stepping with *step()* simply simulates one clock cycle and then returns control back to the user interface.

Simulator implements event-based messaging mechanism to announce simulation events like the end of instruction cycle to interested clients. Event handling is an implementation of the **Observer** design pattern [20].

The main benefit of implementing the event handling mechanism is the ease of adding new simulation event dependent functionality to Simulator without needing to add more overhead to the clock cycle simulation routine. For example, when bus tracing is disabled, no overhead of bus tracing is visible in the routine at all since the simulation loop makes always the same event announcements without knowing how many interested listeners there are for those events.

Clients that are interested in simulation events implement the *Listener* interface and register themselves to the *SimulationEventHandler* which implements the *Informer* interface. When an simulation event is announced, *SimulationEventHandler* informs the observers that are registered to listen to the announced event. Informing the listeners is done by invoking their *handleEvent()* method. For example, when bus tracing is enabled, a *BusTracker* instance which implements the *Listener* interface is registered to listen to clock cycle end events. When an event is announced to the *BusTracker*, its *handleEvent()* fetches the values currently in all the buses of simulated processor and writes them to a log file.

Breakpoints are implemented using the event handling mechanism. *BreakpointManager* of the *Debugging System* module listens to new instruction execution events and stops simulation before an instruction that has a break point set is executed.

Runtime errors of the simulated program, such as a write to a memory location out of bounds of the address space, are also announced as simulation events. The handler of these events informs the user of the reason of the runtime error, thus giving valuable information for tracking bugs in the simulated program.

Finally, the user interfaces of the Simulator may use the event handling mechanism to update their views when there's a change in simulation state, as instructed by the *model-view-controller* architecture [19].

5.4 Operation Set Abstraction Layer

Function units implement one or more operations. Each operation has behavior that needs to be simulated when the operation is triggered.

The module that handles the database of operation definitions is called *Operation Set Abstraction Layer* (OSAL). OSAL provides access to all operation definitions found in the system. Operation data handled by OSAL is divided into static data stored in XML-format files and behavior descriptions that are compiled into dynamic libraries. Usage of dynamic libraries allows loading of behavior simulation functions in Simulator without recompiling the simulator code itself.

One maybe not so apparent detail is that the user-visible operation latency is not a property stored in OSAL descriptions. Latency is defined by the function unit that implements the operation, more specifically, the latency is set by the function unit's pipeline properties. Due to not including the latency property in OSAL, but in ADF function unit properties, it is possible to simulate one operation with different latencies without storing a definition of the operation for each latency.

Static data in OSAL includes vital information of each operation such as the number of operands and results. Additionally, specific properties of each operand of the operation can be given. Such information include details like optionality of operands, whether an operand is for setting a memory address, whether an operand can be swapped with an another operand, and so on. The static properties, expect for the number of operands and the operation name, are mostly used only by the *Instruction Scheduler* to perform different types of program analysis.

An example of a XML-format static data definition of base operation *ADD* is shown in Fig. 12. In the addition operation, the order of input operands does not affect the result, thus the input operands can be swapped, if needed. Such property is marked in OSAL with a *can-swap* definition.

Behavior descriptions are used to simulate the operation. The behavior of an operation is defined as a C++ class of which details are hidden with C preprocessor macros. This set of macros is often called "operation behavior description language", even though it is only camouflage of which purpose is to hide the implementation of the behavior definition plugin interface from TCE users. By hiding details such as the class hierarchy and the factory function that creates an instance of the user defined behavior class, the interface can be later modified by TCE developers without breaking existing operation behavior definitions. Additionally, the macros make behavior descriptions look cleaner to users. The set of macros is defined in file *OSAL.hh*, which is the only TCE header file the operation description files need to directly include.

Code excerpt in Fig. 13 is from *OSAL.hh*. It introduces four important macros, *OPERATION*, *END_OPERATION*, *TRIGGER* and *END_TRIGGER*. The code differs some-

Figure 12. *Example Operation Definition.*

```
<operation>
  <name>ADD</name>
  <inputs>2</inputs>
  <outputs>1</outputs>
  <in id="1">
    <can-swap>
      <in id="2"/>
    </can-swap>
  </in>
  <in id="2">
    <can-swap>
      <in id="1"/>
    </can-swap>
  </in>
  <out id="3"/>
</operation>
```

what from the original, as some irrelevant details are left out. The set of macros might get clearer when it is put next to an example operation description which uses them. Code in Fig. 14 used the macros to describe the behavior of operation *ADD* of the base operation set.

OPERATION macro is expanded to code that defines a new C++ class which implements an interface called *OperationBehavior*. *END_OPERATION*, in addition to ending the class definition, generates a factory function with C linkage (*extern "C"*) to make it possible to instantiate the defined class using the dynamic linking loader which is used to load the behavior definition in the Simulator. In addition to the factory function, a function for destructing the behavior definition class instance is provided. This is to make sure that a correct version of C++ *delete* operator is used to free the created instances [21].

The actual behavior definition code is written between *TRIGGER* and *END_TRIGGER* macros. These macros expand to a *simulateTrigger()* method definition, which defines how the operation behaves when it is triggered. In the example description of *ADD* operation, the first and the second operand are treated as integers, summed, and the result of the summation is written to the third integer operand, which is an output operand. *INT* macro is for casting operands into unsigned integers. There are similar macros

Figure 13. *OSAL.hh: Behavior Definition Macros.*

```
#define OPERATION(OPNAME) \
class OPNAME##_Behavior : public OperationBehavior { \
public: \
        OPNAME##_Behavior(const Operation& parent) : \
                parent_(parent) {}; \
private: \
        const Operation& parent_; \
public:

#define END_OPERATION(OPNAME) \
};\
extern "C" { \
        OperationBehavior* createOpBehavior_##OPNAME(\
                const Operation& parent) {\
                return new OPNAME##_Behavior(parent);\
        }\
        void deleteOpBehavior_##OPNAME(\
                OperationBehavior* target) {\
                delete target;\
        }\
}

#define TRIGGER \
bool simulateTrigger( \
        SimValue** io, \
        OperationContext& context) const {

#define END_TRIGGER }
```

for C floats (*FLT*), and doubles (*DBL*), which are used in floating point operation descriptions. The *RETURN_READY* statement simply signals the Simulator that all the outputs of the operation were computed.

Figure 14. *Operation Behavior Definition of Operation ADD.*

```
#include "OSAL.hh"

OPERATION(ADD)
TRIGGER
        INT(3) = INT(1) + INT(2);
        RETURN_READY;
END_TRIGGER;
END_OPERATION(ADD);
```

Code example in Fig. 15 shows how the description of *ADD* looks like after the macros are expanded.

In addition to the described macros, there are additional macros in OSAL behavior language for defining operations with state, for simulating dynamic latency operations, and for simulating data memory accessing operations, but it serves no purpose to explain them in detail in this thesis. Comments in file *OSAL.hh* and [12] are good sources for thorough explanation on the behavior description language.

5.5 Modeling Data Memory

Data memories accessed by load and store operations are modeled using a simple memory interface. Even though the interface is simple, it gives wide possibilities for modeling different types of memory systems. *Memory Model* interface includes methods for initiating read accesses, querying whether previously requested data is ready to be read, reading data, writing data, and notifying the memory of an instruction cycle advance.

Memory Model implementations might need to model memory systems that limit accesses by some criteria. For example, a memory system may allow maximum of two read accesses to be initiated in the same cycle. In order to allow modeling such limitations, the *Memory Model* interface provides a method for signaling unavailability of the memory to the Simulator. Memory models can implement this method to reflect the properties of the memory they are simulating. For instance, to model a memory system that allows two concurrent read accesses, developer may write an implementation for the method which counts the memory read requests initiated in an instruction cycle and returns false in case the number of requests is more than two.

Figure 15. *Expanded ADD Operation Behavior Definition.*

```
class ADD_Behavior : public OperationBehavior {
public:
        ADD_Behavior(const Operation& parent) :
                parent_(parent) {};
private:
        const Operation& parent_;
public:
        bool simulateTrigger(
                SimValue** io,
                OperationContext& context) const {
                io[3 - 1]->value_.intWord =
                        io[1 - 1]->value_.intWord +
                        io[2 - 1]->value_.intWord;
                return true;
        }
};
extern "C" {
        OperationBehavior* createOpBehavior_ADD(
                const Operation& parent) {
                return new ADD_Behavior(parent);
        }
        void deleteOpBehavior_ADD(
                OperationBehavior* target) {
                delete target;
        }
}
```

The Simulator uses a default *Memory Model* implementation called *IdealSRAM* for simulating data memory access in parallel simulation and an optimized implementation called *SequentialMemory* for sequential simulation. *IdealSRAM* models a memory system that allows an infinite number of concurrent accesses and is always able to serve all the accesses in one clock cycle. Clearly, no practical memory is able to serve unlimited concurrent accesses, thus such a memory model is overly optimistic. It is still safe to use an ideal memory model for simulating the program as the stall cycles caused by conflicting memory accesses are invisible to the simulated program. The drawback of using a less-accurate memory model is that as the stall cycles are not counted in

the total clock cycles spent for running the simulated program, the run time is too optimistic and may lead the *Design Space Explorer* to a wrong direction in the design space. Therefore, if exact results are wanted, writing a more detailed *Memory Model* implementation is strongly encouraged. *SequentialMemory* is optimized with certain special properties of the sequential simulation in mind. For example, all operations in sequential code are simulated with latency of one, thus there can be only one pending request at a time, which avoids the need for request queues.

MemoryContents is a data structure for storing data of the simulated memories. It allows simulating large memories while consuming minimal amounts of simulator host's memory. For example, the width of the address space of the data memory used in sequential simulation is 32 bits, which means four gigabytes of memory space. Naturally, it makes no sense and is often impossible to allocate such an amount of memory for the simulator at once. In order to address this problem, the memory space is divided into chunks at equal distances. Memory for a chunk is allocated only when an address residing in the chunk is written the first time. Reading an address at an unallocated chunk results in returning zero. Because each chunk is the same size, it is possible to find in constant time the chunk and the location in the chunk the requested address refers to.

5.6 Simulation of Instruction Cycle

Previous sections introduced the main modules and their responsibilities in implementing the Simulator functionality. This section aims to describe in detailed manner how those modules interact and work together to actually simulate an instruction cycle.

Thanks to using the preprocessed TTA instruction classes and encapsulation of side effects produced by moves writing to their targets, the top level procedure for simulating a TTA program clock cycle is rather simple. The simulation code resides in method *simulateCycle()* of *Simulation Controller*. The code is shown in Fig. 16 and can be explained line-by-line thanks to its simplicity.

In lines 2 to 4, a preprocessed instruction representing the instruction at current program counter address is fetched from an *ExecutableInstructionMemory* instance. Data transports are simulated by calling *execute()* of the fetched *ExecutableInstruction*.

The side effects of data transports are initiated by specialized function unit port state objects which notify the function unit state object that they were written to, thus an

Figure 16. *Simulation of Instruction Cycle.*

```
try {
        ExecutableInstruction& instruction =
                instructionMemory_->instructionAt(PC);
        instruction.execute();

        machineState_->endClockOfAllFUStates();
        gcu_->endClock();

        memorySystem_->advanceClockOfAllMemories();
        machineState_->advanceClockOfAllFUStates();

        ++gcu_->programCounter();
        gcu_->advanceClock();

        SimulatorToolbox::eventHandler().handleEvent(
                SimulationEventHandler::SE_CYCLE_END);

        if (programEnded()) {
                state_ = STA_FINISHED;
                stopRequested_ = true;
                return;
        }

} catch (const Exception& e) {
        SimulatorToolbox::reportSimulatedProgramError(
             SimulatorToolbox::RES_FATAL, e.errorMessage());
        prepareToStop(SRE_RUNTIME_ERROR);
        return;
}
```

operation should be triggered. The chain of calls in data transport simulation is shown in the sequence diagram of Fig. 17. For each operation implemented by each function unit, there is an instance of a special port state class called *OpcodeSettingVirtualInputPort*. When simulating a data transport with *setValue()* to an object of this class, *setOperation()* of the function unit that owns the port gets called with the operation the virtual input port represents as an argument. Triggering is notified to the function unit

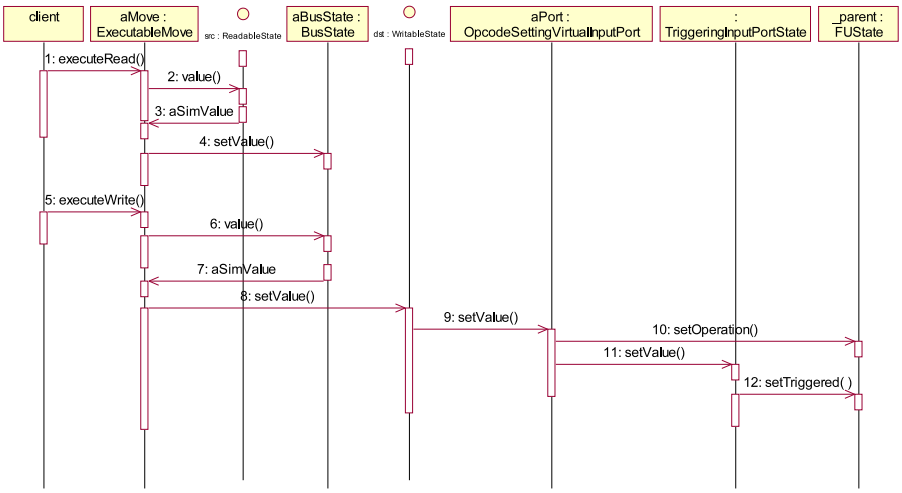

Figure 17. *Data Transport Simulation.*

by delegating the *setValue()* call to an object that represent a real port in the function unit. The port state object, in case the port is supposed to be triggered when written, sets the triggered status to the parent function unit state by calling its *setTriggered()* method. As a result, the function unit has become aware that it should execute the given operation at the point instruction cycle end is announced.

In lines 6 to 13, all function unit state objects, global control unit state object, and all the memory models simulating the data memories, are notified of the advancing of the instruction cycle. Notification of instruction cycle advance to the function units happens in two phases. First, the *endClock()* method is called for each function unit state object. Before calling the *advanceClock()* of the function units, instruction cycle advance is signaled to the memory models. What happens inside *endClock()* and *advanceClock()* and the main reason for the two-phase function unit instruction cycle end signaling is explained in Section 5.1.

At the point a simulation cycle end is announced with *endClock()*, function units that were triggered by data transports simulate the requested operations. The chain of calls for simulating an operation triggering is represented in the sequence diagram of Fig. 18. The simulation of behavior of the triggered operation is delegated to OSAL. This is done by requesting a behavior simulation model for the triggered operation. In sequence diagram the returned model is for ADD, thus the name of the class is *ADD_Behavior*. Operation is executed by calling its *simulateTrigger()* with the operand values currently in function unit state's port objects. *simulateTrigger()* returns results for the simulated operation with the given inputs after which the results are stored in a wait queue. *FUState::endClock()* implements the simulation of opera-

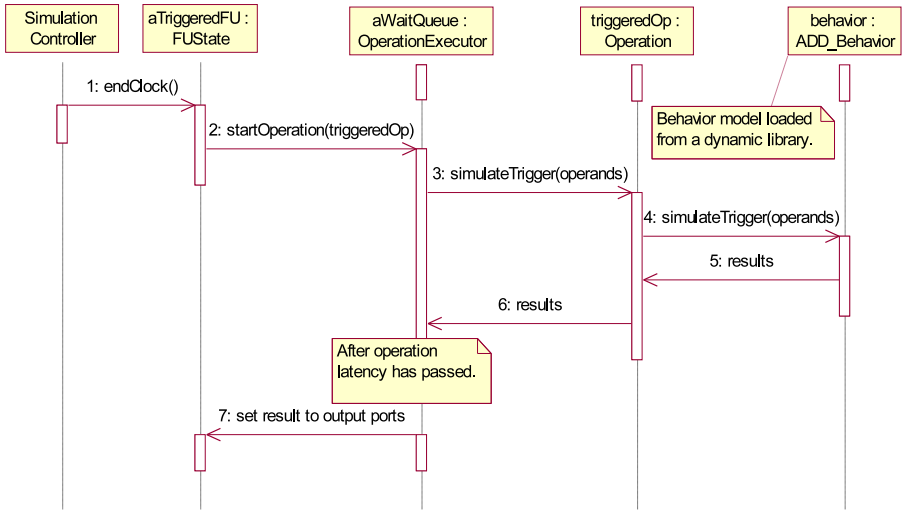

Figure 18. *FUState::endClock(): Simulating Operation Execution.*

tion latency with a helper class *OperationExecutor* which effectively is a wait queue for operation results. Operation results are stored in the queue, and the queue is advanced in each *endClock()* call. At the point a result has stayed long enough, for the time of operation latency, in the queue, result is made visible in the function unit state's output ports.

In order to simulate the delay slots of control flow operations, the effects of triggering a control flow operation such as *jump* are made visible in *advanceClock* of the *GCUState*, called in line 13. The simulation loop increments the program counter value in line 12, which the *GCUState::advanceClock()* overwrites in case the instruction pipeline latency of a previously triggered control flow operation has passed.

The ending of a simulation clock cycle is announced to interested parties in lines 15 to 16 by using the *SimulationEventHandler*.

In lines 18 to 22, the predicate for program end is evaluated by calling *programEnded()*. The function returns true in case the last instruction of the first executed procedure was executed. In such a case, simulation is considered to be executed to the end and simulation is finished succesfully.

Exceptions from simulation loop method calls are caught in lines 24 to 29. Such exceptions are considered run time errors of simulated program, which are announced to interested parties using the simulation event handler. Usually listeners to runtime error events are user interfaces, which print the runtime error messages for the user. Runtime errors always cause the simulation to be aborted.

6. VERIFICATION AND BENCHMARKING

Functional accuracy of simulation was verified by simulating sequential and parallel test programs and comparing output of the programs to known correct output.

Simulation speed was measured by timing the simulation of the test programs. Two figures were calculated from the results: clock cycles simulated in a second, and host clock cycles per simulated cycle, which gives a rough estimate of the number of host clock cycles needed to simulate one target clock cycle. The test programs were also executed with MOVE simulator and results were compared to TCE's results.

All of the benchmarks were executed in a computer equipped with a HyperThreaded(R) Intel(R) Pentium(R) 4 processor running in 2.80 GHz clock frequency and with 1 GB of RAM. Operating system kernel was Linux™ version 2.4.27 with multiprocessor support switched on. Each benchmark was executed ten times in a row. From the results, an average of consumed real time was calculated. Real time was used instead of CPU time because it gives more realistic picture of the speed, as it's the time perceived by users of Simulator. The difference between the measured real time and CPU time was negligible.

MOVE and TCE code base were compiled with GNU GCC compiler version 3.3.5. One of the major drawbacks for MOVE simulation speed is the fact that it's not possible to use aggressive optimization flags when compiling the code of MOVE because of bad coding practices used at some parts of the code base. The best found set of compiler switches using which MOVE could be compiled without producing broken code was *'-O1 -march=pentium4 -finline-limit=5000'*. However, MOVE simulator's code was managed to be compiled separately from the rest of the MOVE code base with more aggressive optimization switches which brought MOVE simulator to roughly the same starting line with the TCE simulator. MOVE simulator's source code and the whole code base of the TCE was compiled with switches *'-O3 -march=pentium4 -finline-limit=5000'*. Switch *'-O3'* turns on the most aggressive optimizations, *'-march=pentium4'* allows Pentium 4 specific optimizations, and *'-finline-limit=5000'* allows large functions inlined to call sites.

6.1 Sequential Simulation

Sequential simulation was tested with an integer-only Ogg Vorbis audio decoder library "Tremor" [22]. The total number of sequential instructions in the program is 98 804. The version of the program used in the test was modified to be suitable for platforms without a 64-bit integer type. It emulates the 64-bit operations Tremor needs by using C functions. According to the CHANGELOG file included in the modified Tremor distribution, the modifications were done by J.A. Bezemer to make Tremor work on MOVE framework. In addition to the modifications by J.A. Bezemer, a version of the dynamic memory allocation function *malloc()*, which does not require operating system support was implemented and used in the test program. Such version was needed because TCE, in contrast to MOVE, does not support emulation of a non-existing operating system.

Verification

The known correct output was generated by compiling the decoder natively in the test environment workstation and by processing an input Ogg Vorbis audio file with the natively compiled version. The used input file was a song of three minutes and nine seconds in length. The resulting output file was verified roughly by listening it using an audio player.

A sequential TTA program was generated from the Tremor source code by using TCE's C frontend. The input file was converted to a C char array which was linked in the test program. Output was produced by using a special operation of which trigger simulation function writes the contents of its only input operand to a file.

The operation behavior definition of the special operation used to output data to a file in simulator host serves as a good example of an operation with state. Its OSAL behavior definition, as illustrated in Fig. 19, consists of a type definition of the state data the operation uses and of an operation behavior definition itself. The state definition is a C++ class definition hidden behind the OSAL macros. After defining the name of the state follows the declaration of the state data, which in this case is the output file stream. Initialization function definition INIT_STATE is used to define the code executed when the state is instantiated. In this case, initialization includes initializing the output file stream to write to a file "ttasim.output". Finally, the destructor which closes the output file stream is defined using FINALIZE_STATE.

Figure 19. *OSAL Behavior Definition of OUTPUT_DATA.*

```
DEFINE_STATE(OUTPUT_STREAM)
        std::ofstream outputFile;
INIT_STATE(OUTPUT_STREAM)
        outputFile.open(
                "ttasim.output",
                std::ios_base::out | std::ios_base::trunc |
                std::ios_base::binary);
END_INIT_STATE;
FINALIZE_STATE(OUTPUT_STREAM)
        outputFile.close();
END_FINALIZE_STATE;
END_DEFINE_STATE;

OPERATION_WITH_STATE(OUTPUT_DATA, OUTPUT_STREAM)
TRIGGER
        STATE.outputFile << static_cast<char>(INT(1));
        RETURN_READY;
END_TRIGGER;
END_OPERATION_WITH_STATE(OUTPUT_DATA);
```

The operation behavior definition is straightforward. The only input is casted to a char and written to the output stream stored in the state instance (referenced to with STATE).

In order to use the defined operation to output the decoded data, a code illustrated in Fig. 20 was inserted in the end of the Tremor's main procedure. The code consists of a simple loop which iterates the array that contains the output data and writes its contents to the output file by using the custom operation. The special operation is accessed with macro WRITETO which generates sequential TTA assembly instructions that write data as input to the operation. The code looks more complicated than it is because the data needs to be swapped before writing it to the output file due to word byte order difference between TCE's TTA and the simulator host processor.

Figure 20. *Code Used to Output Decoded Data in Tremor.*

```
int i = 0;
char first, second;
for (; i < OUTPUT_BUFFER_SIZE; ++i) {
        first = output_data[i];
        ++i;
        second = output_data[i];
        WRITETO(output_data.1, second);
        WRITETO(output_data.1, first);
}
```

Benchmarks

After verifying that the output produced by the simulated TTA program matched the known correct output, the data output code was removed and the test program was benchmarked. Minimal program analysis options were selected for both MOVE and TCE simulator, to achieve the maximum simulation speed. The results are shown in Table 1.

MOVE and TCE have different base operation sets, thus different C frontend compilers, which results in different sequential programs generated by the frontends from the same source code. Additionally, the sequential simulator of MOVE does not use a one-bus "sequential machine" as its internal simulation processor model, but allows some parallelism also with sequential code. Sequential code in MOVE is not simulated as sequences of moves, but as sequences of operation invocations allowing all operand input operands written and results to be read in a single clock cycle. Mainly due to these reasons, the total count of simulated instructions differs dramatically between MOVE and TCE test runs. In order to make comparison between TCE and MOVE sensible, figures are based on executed moves instead of executed instructions.

Table 1. *Benchmark Results: Tremor.*

	moves / second	host cycles / move	total moves
TCE	4,011,700	700	49,321,900,000
MOVE	23,668,000	120	40,905,367,000

The results show that sequential simulation in MOVE is almost six times faster than in TCE. The main reason for this is that TCE uses the same simulation code for parallel simulation by using an artificial Universal Machine that is able to run the sequential

code. Thus, there is no separately optimized simulation code for sequential programs like there is in MOVE. Another major reason is the definition of sequential code in MOVE, which allows move-level parallelism. Because it is possible to perform all operation operand moves in the same instruction, instruction cycle ends need to be simulated less frequently. This might have huge impact to the simulation speed, as the majority of simulation time of TCE Simulator is spent while simulating the clock cycle advances, not while simulating the data transports. Finally, the differences and the added flexibility in TCE's TTA template (see Section 2.2) makes the simulation in TCE less efficient. For example, the simulation of arbitrary width addressable data memory units is completely unnecessary in case of sequential simulation because the used unit widths (minimum addressable unit a 8-bit byte, natural word a 32-bit integer) are the same as in most of the current desktop processors.

As a result of sequential simulation speed being less critical than parallel simulation speed (see Section 3.4), the optimization effort was directed to parallel simulation. In the future, if more speed for sequential simulation is needed, it can be achieved by making more special cases in the simulation code for sequential programs.

6.2 Parallel Simulation

The test program chosen for verification and benchmarking of parallel simulation is an implementation of Viterbi algorithm [23]. The target processor architecture for the program is a fully connected 20-bus TTA with operations distributed in 25 function units. General-purpose registers of the processor are distributed in 7 register files. The algorithm was implemented in C and the resulting sequential TTA code was scheduled and simulated with MOVE. Finally, the output of MOVE instruction scheduler, a parallel TTA assembly file, was converted to TCE's parallel assembler format and simulated in TCE Simulator.

Verification

In this test case, no extra code was inserted to the test program to produce output for verification purposes, but the correctness of the simulation was verified by using TCE Simulator's program debugging capabilities. TCE Simulator implements a feature that allows inspecting simulated data memory and dumping a range of it to the Simulator console. After running the program in TCE Simulator to the end, the memory area

used by the algorithm for storing output data was dumped and compared to the known correct output of the algorithm. The known correct output was provided in the C source code of the algorithm.

Additional verification of parallel simulation was done by simulating different parallel programs and comparing bus traces produced by TCE simulator to bus traces produced by running the program in MOVE simulator and in a VHDL simulator. VHDL implementations of processors used in verification were generated with MOVE processor generator [3] because TCE processor generator was not fully implemented at the time the verification took place.

Benchmarks

The scheduled program contained only 254 instructions, and the total runtime of the algorithm was less than two million cycles. In order to produce longer simulation time to obtain more accurate simulation time measurements, the algorithm was executed thousand times in a row. Results of the benchmarking are displayed in Table 2, which presents the instruction simulation speed, and in Table 3, which presents the move simulation speed.

Table 2. *Benchmark Results: Viterbi, Instruction Execution Speed.*

	instructions / second	host cycles / instruction	total instructions
TCE	668,900	4190	1,799,953,000
MOVE	422,400	6630	1,799,953,000

Table 3. *Benchmark Results: Viterbi, Move Execution Speed.*

	moves / second	host cycles / move	total moves
TCE	2,215,300	1260	5,961,272,000
MOVE	1,399,000	2000	5,961,272,000

It is apparent from the total move and instruction counts that the schedule produced by MOVE's instruction scheduler is not very efficient: even though the processor has 20 buses, only on average of about 3.3 moves were simulated per instruction.

This benchmark shows that TCE is clearly faster than MOVE in parallel simulation. The speedup compared to MOVE is around 58%. The speed difference is most probably due to it being impossible to completely disable runtime statistics computation in MOVE simulator. Utilization counts of processor components are accumulated even

though verbose statistics output is disabled. Additional speedup for TCE is achieved by the preprocessing of instructions and by implementing a mechanism that avoids needless simulation of idle function units.

The effect of specialized code for sequential code in MOVE can be realized by comparing the move execution speed in parallel simulation to move execution speed in sequential simulation represented in Table 1. The parallel simulation speed in this benchmark is almost 17 times slower with MOVE simulator, in contrast to only 1.8 times slower in case of TCE Simulator.

7. FUTURE EXTENSIONS

There are several areas in the Simulator that could be improved to achieve better simulation accuracy and speed. Thanks to the modular structure of Simulator, most improvements should be relatively painless to implement. This chapter introduces several ideas for improvements that could considered in the future. Each improvement idea is accompanied with a short sketch of an implementation plan to serve as a starting point for the possible developers chosen to implement them.

7.1 Parallel Computation

A current trend in computer systems is to try to achieve better performance through parallelism by using multiple processor cores. This is due to the fact that the higher limit for clock frequency achievable with current processor manufacturing technologies is almost reached and duplication of computation resources seems to be a straightforward way to improve the performance of the system.

Benefits from multiprocessor systems are not gained automatically by existing programs. In order to utilize multiple processors, existing programs might be needed to be modified to perform computation in parallel, in multiple threads of execution. All programs are not suitable for parallelization. In case program's execution flow is sequential in a way that all computations depend on the results of the previous computations, it might not be possible to parallelize the program. TCE Simulator is simulating a highly parallel processor architecture, therefore the simulation code is suitable for parallelization. Due to the independent function units of TTA, it is possible to separate the simulation of different function units to multiple threads of execution. Another point for parallelization is the simulation of data transports. Simulation of moves could be distributed evenly among the threads of execution.

In larger scale, design space exploration speed could be improved by distributing computation to a cluster of computers. In such a setting, TCE Simulator would be a network server application serving simulation requests through network. This is imple-

mented easily by writing a new user interface for Simulator. *Network User Interface* would communicate with clients using a network communication protocol. Simulation would be controlled by the interface the same way other Simulator user interfaces do, by using *Frontend* module. The protocol for communicating with the simulation server could include commands for starting a new simulation and receiving simulation results.

7.2 Computing Lock Cycles Generated by Control Unit

Currently, TCE Simulator expects that there are no stalls when retrieving the next executed instruction. This results in the total clock cycle count of simulations being sometimes too optimistic. If the effect of dynamic properties of control unit and instruction memory are wanted to be included in the simulation results, there has to be a a way for users of the Simulator to provide a model for simulating lock conditions caused by the control unit.

One way to implement a mechanism for allowing customized control unit lock condition model is to implement a new class that is used to calculate the lock cycles caused by instruction memory accesses. Users would be allowed to implement this module by using a plugin interface. The interface would consist of a simple function which takes an instruction address as an argument and return the locked processor cycles caused by the request. This kind of mechanism would allow defining models of arbitrary complexity and delivering more exact lock cycle counts caused by the control unit's instruction memory data transfers. For example, an implementation of the model could simulate cache by storing a history of accesses in a data structure similar to the one in the real hardware cache. This data structure would be used to figure out whether requested data would be found in the cache and in case of a cache miss, return a computed count of lock cycles. The default implementation of the interface would always return zero.

7.3 Connection to Hardware Simulation

Some HDL simulators such as "Modelsim" [15] provide an interface for connecting models defined in common programming languages to the simulated environment. For example, the interface of Modelsim called *Foreign Language Interface* [24] imple-

ments a bridge between C language and VHDL, allowing C programs to modify and inspect system's signals which are defined in VHDL.

Communication between the rest of the simulated system and the simulated TTA processor could be implemented by implementing a new memory model for TCE Simulator. The model would not implement the memory storage, but would act as an adapter to the HDL memory block implemented in the hardware model. In case communication through a special operation is wanted, the OSAL operation behavior model that is used to simulate the special operation would include code that uses the C to VHDL interface to communicate with the simulated system.

The only mandatory input to TCE Simulator from the system simulator would be the clock signal. When the TCE Simulator detects that the clock signal is changed, it would simply execute the code that simulates a clock cycle.

7.4 Compiled Simulation

The simulation technique of TCE Simulator resembles the interpretive simulation technique in which each instruction is simulated with function calls that simulate the data transfers and their side effects. The overhead of the function calls itself can be remarkable. Compiled simulation technique avoids the overhead of function calls by translating the simulated program to a program runnable in the host processor.

TCE Simulator could be converted to use compiled simulation technique by replacing the *Simulation Controller* implementation with one that delegates the implementation of the simulation loop to the translated program. It could be possible to implement this by using runtime libraries (plugins) in such way that the TCE code base would not be needed to be recompiled for each simulation. When initializing a compiled simulation, the user-defined program and the processor description would be used to generate a high-level language program, which would be compiled using a regular compiler to a dynamic library. The code from the dynamic library would be used in place of the current simulation loop code.

8. CONCLUSIONS

This thesis described an instruction set simulator for a TTA codesign environment. The main requirements placed for Simulator were the efficiency of parallel TTA program simulation and implementation of program debugging capabilities. The accuracy of Simulator is instruction cycle level and only the architecture of the TTA processor is simulated, that is, Simulator models only the details visible to the programmer. This level of detail allows efficient enough simulation for design space exploration and enables the implementation of program debugging capabilities.

The thesis described the main applications in the codesign environment the Simulator is targeted to. In addition, the TTA concept was described mainly in programmer's point of view, which is enough for understanding the design of the Simulator.

The architecture and design of the Simulator is very modular and follows object oriented design principles. The simulation of data transports (moves) is encapsulated inside a preprocessed instruction model which reduces computation needed during simulation. The state of the processor is maintained in an optimized object model which avoids needless simulation of idle processor parts. The design and implementation of the most important parts of the Simulator were described in more detail, leaving less important details to a separate design document.

Correctness of simulation was verified by simulating sequential and parallel programs and comparing the known correct output of the programs to Simulator's output. The efficiency of the simulation was benchmarked by timing the simulation times of the test programs. The simulation speed was compared to the one of simulator of another codesign environment, the MOVE framework.

Compared to the MOVE simulator, TCE simulator is much slower in sequential simulation, due to the differences in definition of sequential code in MOVE and TCE. In addition, MOVE implements a specially optimized sequential simulation code, which is wanted to be avoided in TCE due to added complexity in providing the debugging features for both types of simulation. Optimization effort was directed to parallel simulation, due to its higher utilization in design space exploration. Contrary to sequential

simulation, parallel simulation is significantly faster in TCE. The parallel benchmark showed speedup of about 60%.

Finally, the thesis introduced several future extensions for improving simulation accuracy and speed. For example, utilization of parallel computation by adding multiple threads of execution or by distributing the simulation to multiple hosts is a self-evident way to improve parallel simulation speed. In addition, applying the compiled simulation technique would probably improve the overall simulation speed tremendously.

9. BIBLIOGRAPHY

[1] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "LISA - machine description language for cycle-accurate models of programmable DSP architectures," in *Proc. ACM/IEEE Conf. Design Automation*. New York, NY, USA: ACM Press, 1999, pp. 933–938.

[2] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. Chichester, UK: John Wiley & Sons, 1997.

[3] J. Sertamo, "Processor Generator for Transport Triggered Architectures," Master's thesis, Tampere Univ. of Tech., Tampere, Finland, Sept. 2003.

[4] A. Cilio, "TCE Architecture Template Programming Interface Functional Requirements," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2004-2005.

[5] M. Ercegovac, T. Lang, and J. H. Moreno, *Introduction to Digital Systems*. New York, US: John Wiley & Sons, 1999.

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. San Francisco, US: Morgan Kaufmann, 1998.

[7] H. Corporaal and M. Arnold, "Using transport triggered architectures for embedded processor design," *Integrated Computer-Aided Eng.*, vol. 5, no. 1, pp. 19–38, 1998.

[8] B. Stroustrup, "A history of C++: 1979-1991," in *Proc. ACM SIGPLAN Conf. History of Programming Languages*. New York, NY, USA: ACM Press, 1993, pp. 271–297.

[9] A. Cilio, H. J. M. Schot, and J. A. A. J. Janssen, "Architecture Definition File: Processor Architecture Definition File Format for a New TTA Design Framework," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2003-2005.

[10] "GCC, the GNU Compiler Collection," Website, 2005, http://gcc.gnu.org.

[11] T. Rantanen, "Cost Estimation for Transport Triggered Architectures," Master's thesis, Tampere Univ. Tech., Tampere, Finland, May 2004.

[12] A. Cilio and P. Jääskeläinen, "Operation Set Abstraction Layer, Functional Specifications," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2003-2005.

[13] *Synopsys Online Documentation*, Synopsys, 2002.

[14] J. Heikkinen, A. Cilio, J. Takala, and H. Corporaal, "Dictionary-based program compression on transport triggered architectures," in *Proc. IEEE Int. Symp. on Circuits and Systems*, Kobe, Japan, May. 23–26 2005, pp. 1122–1125.

[15] *ModelSim SE User's Manual*, Model Technology, 2003.

[16] J. K. Ousterhout, *Tcl and Tk Toolkit*. Addison-Wesley, 1994.

[17] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation," in *Proc. ACM/IEEE Conf. Design automation*. New York, NY, USA: ACM Press, 2003, pp. 758–763.

[18] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, 2003.

[19] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26 – 49, Aug./Sept. 1988.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, USA: Addison-Wesley, 1995.

[21] "C++ Dlopen Mini HOWTO," Website, 2003, http://www.tldp.org/HOWTO/C++-dlopen/.

[22] "Tremor, an integer-only, fully Ogg Vorbis compliant software decoder library," Website, 2005, http://www.xiph.org/ogg/vorbis/.

[23] A. J. Viterbi, "Error bounds for convolutional coding and an asymptotically optimum decoding algorithm," *IEEE Trans. Inform. Theory*, vol. 13, pp. 260–269, Apr. 1967.

[24] *ModelSim Foreign Language Interface Reference*, Model Technology, 2003.

Appendix A

SIMULATION OF UNPROCESSED INSTRUCTION

The algorithm can be divided into following phases:

1. Lines 2-4: Validation of the executed move. Move is valid in case the simulated architecture provides the needed connections from the source to the destination, and no other move is occupying the same resources in the same clock cycle.

2. Lines 5-27: Interpretation. Source, destination, and bus of the move are resolved, that is, corresponding state objects are found from the object model that maintains the processor's simulation state.

3. Lines 28-31: Simulation of the data transfer. Data is first read from the source to bus, from which it is read to the destination.

4. Lines 32-39: Simulation of possible side effects of the data transfer. As a side effect, a data transfer may set the operation to be triggered next in the target function unit and trigger an operation.

Algorithm 2 Simulating an Unprocessed Instruction

```
1:  for all moves M in instruction do
2:      if not Valid(M) then
3:          abort simulation
4:      end if
5:      if M.SourceIsRegister() then
6:          source_state ←FindRegisterState(M.source)
7:      else if M.SourceIsFUPort() then
8:          source_state ←FindPortState(M.source)
9:      else if M.SourceIsImmediate() then
10:         source_state ←CreateImmediateObject(M.sourceImmediateValue)
11:     end if
12:     bus_state = FindBusState(M.bus)
13:     if M.IsGuarded() then
14:         if M.IsPortGuard() then
15:             guard_target ←FindPortState(M.guardedPort)
16:         else if M.IsRegisterGuard() then
17:             guard_target ←FindRegisterState(M.guardedRegister)
18:         end if
19:     end if
20:     if M.DestinationIsRegister() then
21:         destination_state ←FindRegisterState(M.destination)
22:     else if M.DestinationIsFUPort() then
23:         destination_state ←FindPortState(M.destination)
24:     end if
25:     if M.IsOpcodeSetting() then
26:         operation ←FindOperationBehaviorModel(M.opcode)
27:     end if
28:     bus_state.Write(source_state.Read())
29:     if not M.IsGuarded() or M.IsGuarded() and guard_reg.Value not 0 then
30:         destination_state.Write(bus_state.Read())
31:     end if
32:     if M.HasSideEffects() then
33:         function_unit_state ←FindFUState(M.functionUnit)
34:         if M.IsOpcodeSetting() then
35:             function_unit_state.SetNextOperation(operation_behavior)
36:         else if M.IsTriggering() then
37:             function_unit_state.SetTriggered()
38:         end if
39:     end if
40: end for
```