Tampere University of Technology
Department of Computer Systems

Project: TTA Codesign Environment

TTA Codesign Environment v1.1
User Manual

Version: 14
Created: 25.10.2006
Last Modified: 28.01.2010
Document: P-1000
State: complete

Contents

Chapter 1

INTRODUCTION

1.1 Document Overview

This is the user manual for TTA Codesign Environment (TCE). The document describes the usage of all the tools in the toolset, and the most common design flows in the form of tutorials.

Chapter 2 provides an overview to the TCE processor design flow, and Chapter 3 contains tutorials for several common TCE use cases. These should be sufficient for starting to use the TCE toolset. The rest of the chapters describe the use of each tool separately, and they can be referred to for information on more advanced usage the tools.

1.2 Acronyms, Abbreviations and Definitions

ADF	(Processor/Machine) Architecture Definition File.
BEM	Binary Encoding Map. Describes the encoding of instructions.
CLI	Command Line Interface
ExpResDB	Exploration Result Database.
GUI	Graphical User Interface
GPR	General Purpose Register
HDB	Hardware Database
HDL	Hardware Description Language.
HLL	High Level (Programming) Language.
IDF	(Machine/Processor) Implementation Definition File.
ILP	Instruction Level Parallelism.
LLVM	Low Level Virtual Machine
MAU	Minimum Addressable Unit
PIG	Program Image Generator
SQL	Structured Query Language.
TCE	TTA Codesign Environment.
TPEF	TTA Program Exchange Format
TraceDB	Execution Trace Database.
TTA	Transport Triggered Architecture.
VHDL	VHSIC Hardware Description Language.
XML	Extensible Markup Language.

1.3 Typographic Conventions Used in the Document

Style	Purpose
italic	parameter names in running text
[brackets]	bibliographic references
‘single quoted’	keywords or literal strings in running text
’file name’	file and directory names in running text
bold	Shorthand name of a TCE application.

Chapter 2

TCE DESIGN FLOW

2.1 Overview

The main goal for the TTA Codesign Environment (TCE) is to provide a reliable and effective toolset for designing programmable application specific processors, and generate machine code for them from applications written in high-level languages.

In addition, TCE provides an extensible research platform for experimenting with new ideas for Transport Triggered Architectures (TTAs), retargetable ILP code generation, and application specific processor design methodology, among others.

The TCE design flow starts from an application described in a high level language (currently the C language). The LLVM compiler framework [llv08] is used to compile the application to 'bitcode', the intermediate representation of LLVM. The resulting bitcode is then compiled and scheduled to a particular TTA processor by TCE. Traditional compiler optimizations are done in LLVM before bitcode generation, so it is possible to utilize the same bitcode file when exploring different TTA processors for running an application.

The initial software development phase is intended to be separate from the actual codesign flow of TCE. That is, the program is expected to be implemented and tested natively (on a workstation PC) before "porting" it to the TTA/TCE platform. The porting includes ensuring that TTA/TCE runs the program correctly, and optimizing the hardware together with the software by modifying the resources and architecture of the processor to fit the application at hand – a process called hardware/software codesign.

The main phases in the design flow of TCE are illustrated in the following figures. Figure 2.1 depicts the initial inputs to TCE, Figure 2.2 the design space exploration phase, Figure 2.3 the processor configuration selection phase, Figure 2.4 the code generation and analysis phase, and Figure 2.5 the generation of the final outputs: the processor description and the program bit image.

2.2 Main File Formats

This chapter gives an overview of files and databases manipulated by TCE applications and accessible to users of the toolset.

2.2.1 Architecture Definition File (ADF)

Filename extension: .adf

Machine Architecture Definition File (ADF) is a file format for defining target processor architectures. ADF is a minimal specification of the target processor architecture, meaning that only the information needed to generate valid programs for the processor is stored, nothing else.

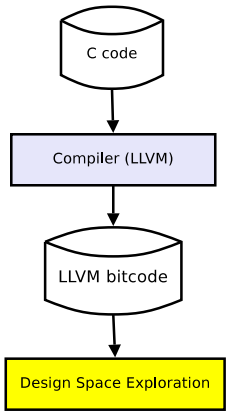

Figure 2.1: The Initial Inputs for TCE Design Flow.

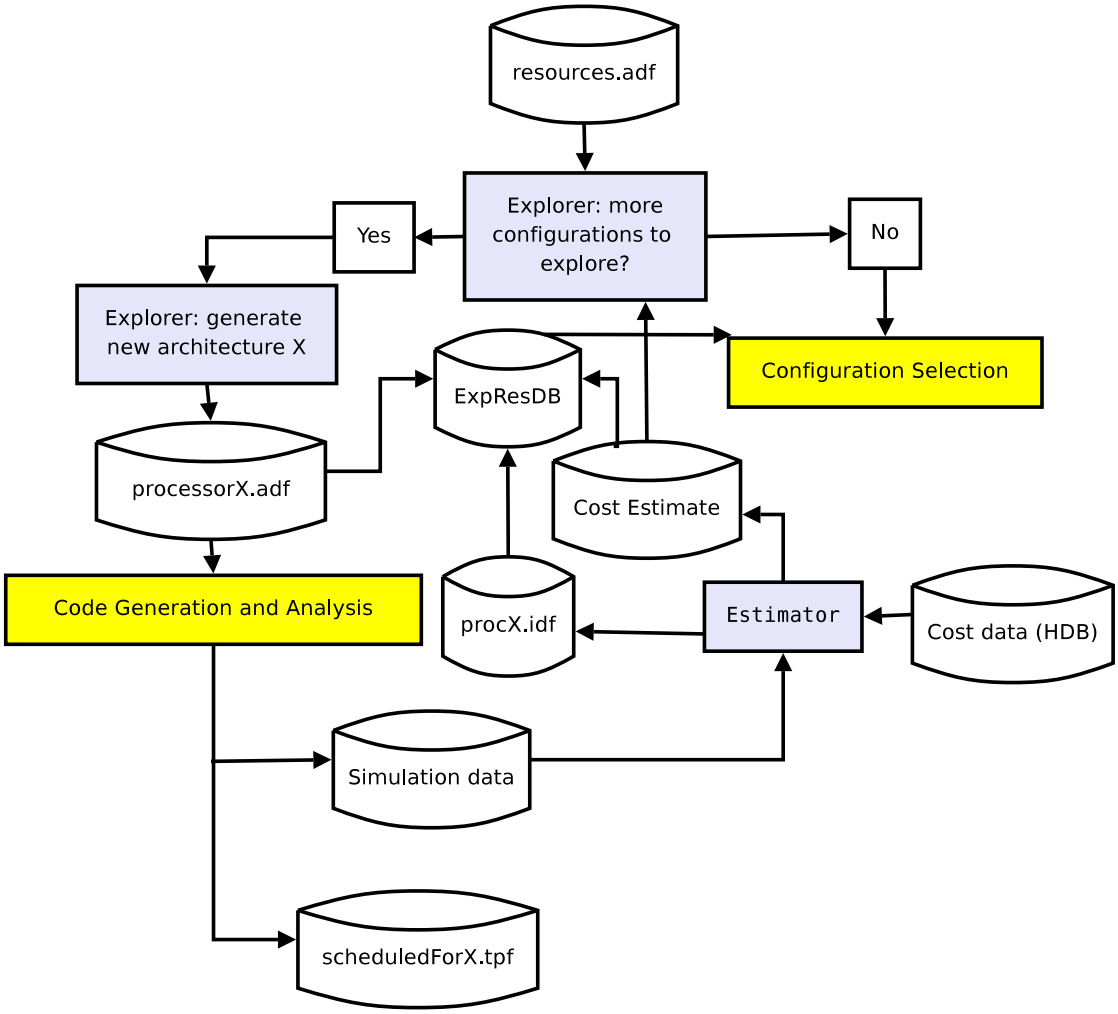

Figure 2.2: Design Space Exploration.

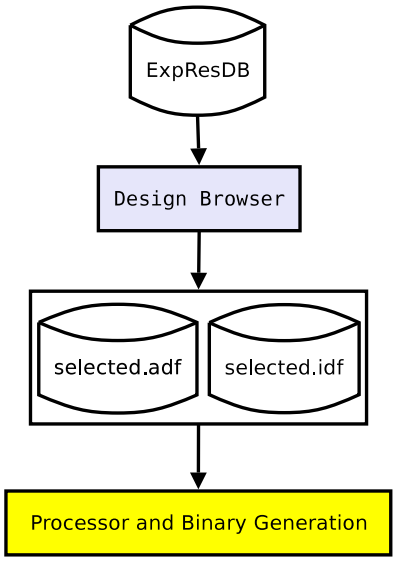

Figure 2.3: Processor Configuration Selection.

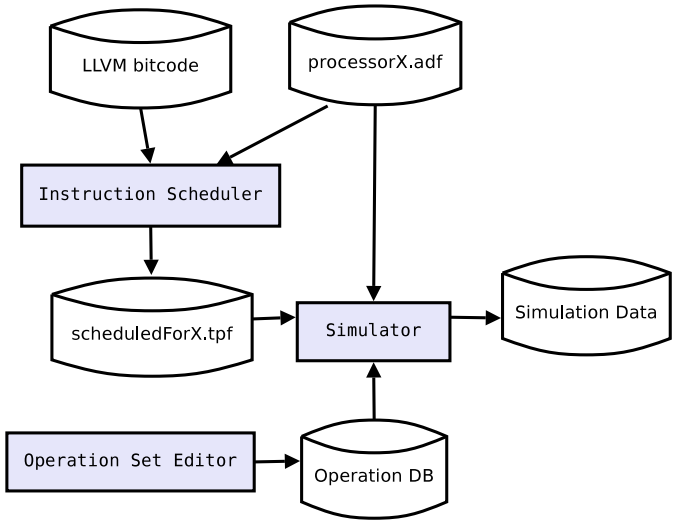

Figure 2.4: Code Generation and Analysis.

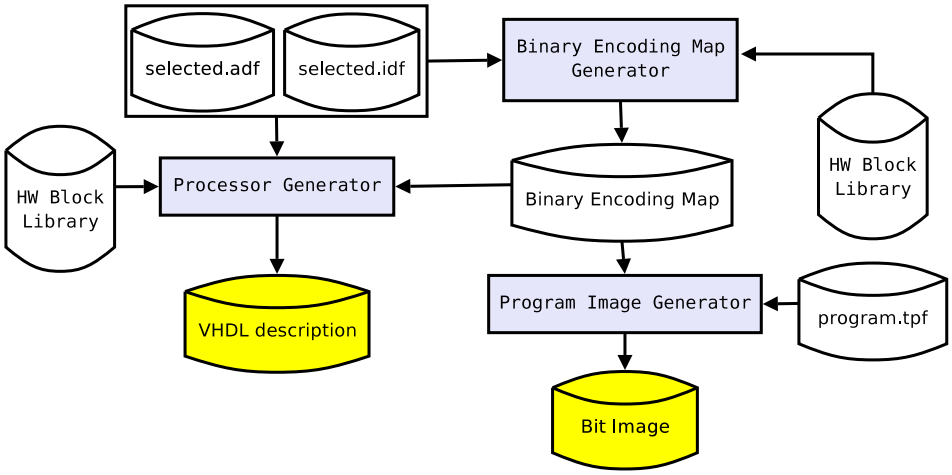

Figure 2.5: Processor and Program Image Generation.

2.2.2 Hardware Database (HDB)

Filename extension: .hdb

Hardware Database (HDB) is the main database used by the Processor Generator and the Cost Estimator. The data stored in HDB consist of hardware description language definitions (HDL) of TTA components (function units, register files, buses and sockets) and metadata that describe certain parameters used in implementations. In addition, HDB may include data of each implementation needed by the cost estimation algorithms.

TCE ships with an example HDB that includes implementations for several function units and register files, and cost data for the default interpolating cost estimation plugin.

2.2.3 Implementation Definition File (IDF)

Filename extension: .idf

Describes which implementations to use for each component in the architecture (defined in an ADF file). Using this information it is possible to fetch correct hardware description language (HDL) files from the hardware block library for cost estimation and processor generation.

2.2.4 Binary Encoding Map

Filename extension: .bem

Provides enough information to produce an executable uncompressed bit image (Section **??**) from TPEF program data.

2.2.5 TTA Program Exchange Format (TPEF)

Filename extension: .tpef

TTA Program Exchange Format (TPEF) is a file format for storing unscheduled, partially scheduled, and scheduled TTA programs. TPEF supports auxiliary sections for storing additional information related to the program, such as execution profiles, machine resource data, and target address space definitions.

2.2.6 Operation Set Abstraction Layer (OSAL) Files

Filename extension: .opp, .cc, .opb

OSAL stores the simulation behavior and static properties of operations in function units.

Simulation behavior of function unit operations is described by implementing simulation functions which can be plugged in to the simulator run time.

The .opp file is an XML file for defining the static properties of operations (for example, how many inputs and outputs an operation has). The .cc is the C++ source file that defines the behavior model for a set of operations. The .opb is the plugin module compiled from the .cc.

Operations are divided in "operation modules". For example, 'base' module, included in the TCE distribution, contains all the operations available to the front end compiler's code generation.

An operation in OSAL is defined by its properties and its behavior. The properties defined by OSAL do not restrict in any way the hardware implementation. For example, latency, bit width or the fact that reading the result of an operation can lock the processor are properties of the implementation, and are not defined in OSAL module.

2.2.6.1 Operation Properties

The following properties define a TTA operation in TCE:

- name (unique string)

- description (string)
- number of inputs (integer)
- number of outputs (integer)
- accesses memory (yes/no)
- has side effects (yes/no)
- clocked (yes/no)
- affected-by (set of operations)
- affects (set of operations)

operation name The operation name is a string of characters starting with a character in set [A-Z_] and followed by one or more character in set [0-9A-Z_] (i.e. lowercase letters are not allowed). All names of operations of the database must be unique. Different data bases can contain operations with equal names.

operation description Optional description of the operation.

inputs Number of inputs of the operation. The number of inputs is a nonnegative integer. It must be positive if the number of outputs is zero.

outputs Number of outputs of the operation. The number of outputs is a nonnegative integer. It must be positive if the number of inputs is zero.

reads/writes-memory Indicates that this operation can access memory. Normally, memory access is also implied from the properties 'mem-address' and 'mem-data' of operation inputs (or the 'mem-data' property of operation outputs). However, it is possible to define operations that perform *invisible* accesses to memory, whereby no input or output is related to the memory access itself. That is, neither the address nor the data moved into or out of memory is explicitly specified by the operation. In these operations, memory accesses occur as a side effect, and none of the inputs or outputs have memory-related properties. To avoid potential errors, the memory property must be specified explicitly even when it is implied by some of the inputs or outputs of the operation. See sections on input and output declarations, below.

clocked Clocked attribute indicates that the operation can change its state synchronously with clock signal and independently from its input.

side-effect Indicates that two subsequent executions of this operation with the same input values may generate different output values. An operation marked with "side-effect" is an operation which writes some state data, affecting further executions of the same operation and other operations sharing that same state data. The other operations that read or write the same state data written by an operation marked with "side-effect" must be marked "affected-by" that operation (see later for an example).

Note: only operations that write the state should marked to have "side-effects". Operations that only read the state data do not have side effects and may be reordered more freely by the compiler.

affected-by In case an operation reads or writes state data written by another operation sharing the same state data (that is marked with the "side-effect" property), the operation should be marked "affected-by" that operation.

This property restricts the compiler's reordering optimizations from moving operations that read or write state data above an operation that also writes the same data, which would result in potentially producing wrong results from the execution.

affects This is an optional convenience property that allows defining the state data dependency the other way around. If an operation is listed in the "affects" list it means that the operation is writing state data that is read or written by the affected operation.

Note: it is not necessary that, if operation A 'affects' operation B, then B must contain A in its 'affected-by' list. Vice versa, if A is 'affected-by' B, it is not needed that B must contain A in its 'affects' list. This allows, for example, a user to add new operations that share state with the base operations shipped with TCE, without needing to modify the base operations.

An example of defining multiple operations that share the same state. A common use case for multiple operations that share state data is a case where one or more operations initialize an internal register file in an FU and one or more operations use the data in the register file to compute their results.

For example, INIT_RF could initialize the internal register file with the given number. This operation should be marked "side-effects". Let's say that another two operations COMPUTE_X and COMPUTE_Y only read the internal RF data, thus they can be freely reordered by the computer with each other in the program code in case there are no other dependencies. As they only read the state data, they don't have and visible side effects, thus they should not be marked with the "side-effects" property. However, as they read the data written by the INIT_RF operation, both operations should be marked to be "affected-by" the INIT_RF.

2.2.6.2 Operation Input Properties

Each input of an operation requires an independent declaration of its properties. An operation input is completely defined by the following properties:

- identification number (integer)
- memory address (yes/no)
- memory data (yes/no)
- can be swapped (set of integers)

identification number Integer number in the range [1,N] where N is the number of inputs as defined in section 2.2.6.1 of operation declaration. If N is zero, then no input declarations can be specified. TCE does not currently allow operations with zero inputs to be defined.

can-swap A list of identification numbers. All the inputs listed can be swapped with this input. The identification number of this input definition is not allowed in the list. The can-swap property is commutative, thus any of the listed inputs is implicitly 'can-swap' with this input and all the other inputs listed. The can-swap declaration need not be symmetrical, but it is not an error if the implied declaration is also specified.

mem-address Optional. Indicates that the input is used to compute (affects the value of) the memory address accessed by this operation.

mem-data Optional. Indicates that the input is used to compute (affects the value of) the data word written to memory by this operation. This property implies that the operation writes to memory.

2.2.6.3 Operation Output Properties

Note: it is not an error if a program, before instruction scheduling, contains an operation where one of the output moves is missing. If all output moves of an operation are missing, then the only useful work that can be performed by the operation is state change.

mem-data Optional. Indicates that the output contains a value that depends on a data word read from memory by this operation. This property implies that the operation reads data from memory.

2.2.6.4 Operation Behavior

To be complete, the model of an operation needs to describe the behavior of the operation. The behavior is specified in a restricted form of C++, the source language of the TCE toolset, augmented with macro definitions. This definition is used for simulating the operation in the instruction set simulator of TCE.

Definition of Operation Behavior Operation behavior simulation functions are entered inside an operation behavior definition block. There are two kinds of such blocks: one for operations with no state, and one for operations with state.

OPERATION(*operationName*) Starts an operation behavior definition block for an operation with name *operationName*. Operations defined with this statement do not contain state. Operation names must be written in upper case letters!

END_OPERATION(*operationName*) End an operation behavior definition block for an operation with no state. *operationName* has to be exactly the same as it was entered in the block start statement `OPERATION()` .

OPERATION_WITH_STATE(*operationName*, *stateName*) Starts an operation behavior definition block for an operation with state. *operationName* contains the name of the operation, *stateName* name of the state. DEFINE_STATE() definition for the *stateName* must occur before this statement in the definition file. Operation and state names must be written in upper case letters!

END_OPERATION_WITH_STATE(*operationName*) Ends an operation behavior definition block for an operation with state. *operationName* has to be exactly the same as it was entered in the block start statement `OPERATION_WITH_STATE()` .

The main emulation function definition block is given as:

TRIGGER ... END_TRIGGER; Main emulation function.

The bodies of the function definitions are written in the operation behavior language, described in Section 2.2.6.5.

Operations with state. To define the behavior of an operation with state it is necessary to declare the state object of the operation. An operation state declaration is introduced by the special statement `DEFINE_STATE()` . See Section 2.2.6.5 for a description of this and related statements. State must be declared before it is used in operation behavior definition.

A target processor may contain several *implementations* of the same operation. These implementations are called Hardware Operations and are described in [CSJ04]. Each Hardware Operation instance belongs to a different function unit and is independent from other instances. When an operation has state, each of its Hardware Operations uses a different, independent instance of the state class (one for each function unit that implements that operation).

An operation state object is unambiguously associated with an operation (or a group of operations, in case the state is shared among several) by means of its name, which should be unique across all the operation definitions.

see issue **??**

Operation state can be accessed in the code that implements the behavior of the operation by means of a `STATE` expression. The fields of the state object are accessed with the dot operator, as in C++. See Section 2.2.6.5 for a complete description of this statement.

Main emulation function. The behavior model of an operation must include a function that, given a set of input operand values and, optionally, an operation state instance, produces one or more output values that the operation would produce.

The definition of an emulation function is introduced by the statement `TRIGGER` and is terminated by the statement `END_TRIGGER;` .

An emulation function is expected to read all the inputs of its operation and to update the operation outputs with any new result value that can be computed before returning.

2.2.6.5 Behavior Description language

The behavior of operations and the information contents of operation state objects are defined by means of the behavior description language.

The emulation functions that model operation behavior are written in C++ with some restrictions. The OSAL behavior definition language augments the C++ language with a number of statements. For example, control may exit the definition body at any moment by using a special statement, a set of statements is provided to refer to operation inputs and outputs, and a statement is provided to access the memory model.

Base data types. The behavior description language defines a number of base data types. These types should be used to implement the operation behavior instead of the C base data types, because they guarantee the bit width and the format.

IntWord Unsigned integer 32-bit word.

FloatWord Single-precision (32-bit) floating-point word in IEEE-754 format.

DoubleWord Double-precision (64-bit) floating-point word in IEEE-754 format.

Access to operation inputs and outputs. Inputs and outputs of operations (henceforth referred to as *terminals*, when a distinction is not needed) are referred to by a unique number. The inputs are assigned a number starting from 1 for the first input. The first output is assigned the number $n+1$, where n is the number of inputs of the operations, the second $n+2$, and so on.

Two sets of expressions are used when accessing terminals. The value of an input terminal can be read as an unsigned integer, a signed integer, a single precision floating point number, or a double precision floating point number using the following expressions:

UINT(*number*) Treats the input terminal denoted by *number* as a number of type IntWord, which is an unsigned integer of 32 bits maximum length.

INT(*number*) Treats the input terminal denoted by *number* as a number of type SIntWord, which is a signed integer of 32 bits maximum length.

FLT(*number*) Treats the input terminal denoted by *number* as a number of type *FloatWord*.

DBL(*number*) Treats the input terminal denoted by *number* as a number of type *DoubleWord*.

Output terminals can be written using the following expression:

IO(*number*) Treats the terminal denoted by *number* as an output terminal. The actual bit pattern (signed, unsigned or floating point) written to the output terminal is determined by the right hand expression assigned to the IO() expression.

Since the behavior of certain operations may depend in non-trivial ways on the bit width of the terminals of a given implementation, it is sometimes necessary to know the bit width of every terminal. The expression

BWIDTH(*number*)

returns the bit width of the terminal denoted by *number* in the implementation of the calling client.

Bit width of the operands can be extended using two different expressions.

SIGN_EXTEND(*integer*, *sourceWidth*) Sign extends the given integer from *sourceWidth* to 32 bits.

Sign extension means that the sign bit of the source word is duplicated to the extra bits provided by the wider target destination word.

For example a sign extension from 1001b (4 bits) to 8 bits provides the result 1111 1001b.

ZERO_EXTEND(*integer*, *sourceWidth*) Zero extends the given integer from *sourceWidth* to 32 bits.

Zero extension means that the extra bits of the wider target destination word are set to zero.

For example a zero extension from 1001b (4 bits) to 8 bits provides the result 0000 1001b.

Example. The following code implements the behavior of an accumulate operation with one input and one output, where the result value is saturated to the "all 1's" bit pattern if it exceeds the range that can be expressed by the output:

```
STATE.accumulator += INT(1);
IntWord maxVal = (1 << BWIDTH(2)) - 1;
IO(2) = (STATE.accumulator <= maxVal ? STATE.accumulator : maxVal);
```

Definition of operation state. Operation state consists of a data structure. Its value is shared by one or more operations, and it is introduced by the statement

DEFINE_STATE(*name*)

where *name* is a string that identifies this type of operation state. This statement is followed by a list of data type fields. The state name string must be generated with the following regular expression:

[A-Z][0-9A-Z_]*

Note that only upper case letters are allowed.

A state definition block is terminated by the statement

END_DEFINE_STATE

Example. The following declaration defines an operation state class identified by the name string "BLISS", consisting of one integer word, one floating-point word and a flag:

```
DEFINE_STATE(BLISS)
  IntWord data1;
  FloatWord floatData;
  bool errorOccurred;
END_DEFINE_STATE;
```

Some operation state definitions may require that the data is initialized to a predefined state, or even that dynamic data structures are allocated when the operation state object is created. In these cases, the user is required to provide an initialization definition inside the state definition block.

INIT_STATE(*name*) Introduces the code that initializes the operation state.

END_INIT_STATE Terminates the block that contains the initialization code.

Some state definitions may contain resources that need to be released when the state model is destroyed. For example, state may contain dynamically allocated data or files that need to be closed. In these cases, the user must define a function that is called when the state is deallocated. This function is defined by a finalization definition block, which must be defined inside the state definition block.

FINALIZE_STATE(*name*) Introduces the code that finalizes the operation state, that is, deallocates the dynamic data contained in an operation state object.

END_FINALIZE_STATE Terminates the block that contains finalisation code.

The state model provides two special definition blocks to support emulation of operation behaviour.

ADVANCE_CLOCK ... END_ADVANCE_CLOCK In case the model of operations state is synchronous, this definition can be used to specify activity that occurs "in the raising edge of the clock signal", that is, at the end of a simulation cycle. The C 'return' statement can used to return from this function.

Access to operation state. Operation state is denoted by a unique name string and is accessed by means of the statement

STATE

Typically, an operation state data structure consists of several fields, which are accessed using the dot operator of C++. For example, the expression `STATE.floatData` in a simulation function refers to the field `floatData` of the state object assigned to the operation being defined. The precise format of an operation state structure is defined by means of the `DEFINE_STATE()` statement, and it is specific for an operation. State must be defined before it can be used in an operation definition.

Access to control registers. Operations that can modify the program control flow can access the program counter register and the return address of the target processor by means of the following expressions:

PROGRAM_COUNTER

RETURN_ADDRESS

Not all operations can access the control registers. Only operations implemented on a Global Control Unit (see [CSJ04]) provide the necessary data. It is an error to use `PROGRAM_COUNTER` or `RETURN_ADDRESS` in operations that are not implemented on a Global Control Unit.

Context Identification Each operation context can be identified with a single integer which can be accessed with `CONTEXT_ID` .

Returning from operation behavior emulation functions. Normally, an emulation function returns control to the caller when control flows out of the definition body. To return immediately, from an arbitrary point in the definition body, the following normal C++ return statement can be used. The return value is boolean indicating the success of the operation execution. In pratice, 'true' is always returned: `return true;` .

Memory Interface. The memory interface of OSAL is very simplified allowing easy modeling of data memory accessing operations. The following keywords are used to define the memory access behavior:

MEMORY.read(address, count, target) Reads *count* units of data from the *address* to the variable *target*.

MEMORY.write(address, count, data) Writes *count* units of data in variable *data* to the *address*.

2.2.7 Simulation Trace Database

Filename extension: .tracedb

Stores data collected from simulations and used by instruction scheduler (profiling data) and cost estimator (utilization statistics, etc.).

2.2.8 Exploration Result Database

Exploration Result Database (ExpResDB) contains the configurations that have been evaluated during exploration (manual or automatic) and a summary of their characteristics. Files that define each tested configuration (ADF and IDF) are stored in the database as well.

2.3 Notes About the Processor Template of TCE

The processor template from which the application specific processors designed with TCE are defined from is called Transport Triggered Architecture (TTA). For a detailed description behind the TTA philosophy, refer to [Cor97]. This section describes certain aspects of the TTA template used in the TCE toolset that might not be very clear.

2.3.1 Immediates/Constants

The TTA template supports two ways of transporting program constants in instructions. *Short immediates* are encoded in the move slot's source field, and thus consume a part of a single move slot. The constants transported in the source field are usually relatively small in size. Wider constants can be transported by means of so called *long immediates*. Long immediates can be defined using a parameter called *instruction template*. The idea is that each TTA instruction is connected to a single instruction template which defines the move slots that contain pieces of a long immediate, if any. The slots cannot be used for regular data transports when they are used for transporting pieces of a long immediate. An instruction containing a long immediate also provides a target to which the long immediate must be transported. The target is so called *immediate unit* which is written directly from the control unit, not through the transport buses. The immediate unit is like a register file expect that it contains only read ports and is written only by the instruction decoder in the control unit when it detects an instruction with a long immediate.

2.3.2 Operations, Function Units, and Operand Bindings

Due to the way TCE abstracts operations and function units, an additional concept of *operand binding* is needed to connect the two in processor designs.

Operations in TCE are defined in a separate database (OSAL, Section 2.2.6) in order to allow defining a reusable database of "operation semantics". The operations are used in processor designs by adding *function units* (FU) that implement the wanted operations. Operands of the operations can be mapped to different ports of the implementing FU, which affects programming of the processor. Mapping of operation operands to the FU ports must be therefore described by the processor designer explicitly.

Example. Designer adds an FU called 'ALU' which implements operations 'ADD', 'SUB', and 'NOT'. ALU has two input ports called 'in1' and 'in2t' (triggering), and an output port called 'out'. A logical binding of the 'ADD' and 'SUB' operands to ALU ports is the following:

```
ADD.1 (the first input operand) bound to ALU.in1
ADD.2 (the second input operand) bound to ALU.in2t
ADD.3 (the output operand) bound to ALU.out

SUB.1 (the first input operand) bound to ALU.in1
SUB.2 (the second input operand) bound to ALU.in2t
SUB.3 (the output operand) bound to ALU.out
```

However, operation 'NOT', that is, the bitwise negation has only one input thus it must be bound to port 'FU.in2t' so it can be triggered:

```
NOT.1 bound to ALU.in2t
NOT.2 (the output operand) bound to ALU.out
```

Because we have a choice in how we bind the 'ADD' and 'SUB' input operands, the binding has to be explicit in the architecture definition. The operand binding described above defines architecturally different TTA function unit from the following:

```
SUB.2 bound to ALU.in1
SUB.1 bound to ALU.in2t
SUB.3 bound to ALU.out
```

With the rest of the operands bound similarly as in the first example.

Due to the differing 'SUB' input bindings one cannot run code scheduled for the previous processor on a machine with an ALU with the latter operand bindings. This small detail is important to understand when designing more complex FUs, with multiple operations with different number of operands of varying size, but is usually transparent to the basic user of TCE.

Reasons for wanting to fine tune the operand bindings might include using input ports of a smaller width for some operation operands. For example, the width of the address operands in memory accessing operations of a load store unit is often smaller than the data width. Similarly, the second operand of a shift operation that defines the number of bits to shift requires less bits than the shifted data operand.

Chapter 3

TUTORIALS

3.1 TCE Tour

This tutorial goes through most of the tools in TCE using a fairly simple example application. It starts from C code and ends up with VHDL of the processor and a bit image of the parallel program. This tutorial will also explain how to accelerate an algorithm by customizing the instruction set, i.e. by using custom operations. The total time to go through the tutorial is about 2 to 3 hours.

The tutorial file package is available at:

`http://tce.cs.tut.fi/tutorial_files/tce_tutorials.tar.gz`

Unpack it to a working directory and cd to tce_tutorials/tce_tour.

3.1.1 The Sample Application

The test application counts a 32-bit CRC (Cyclic Redundant Check). The C code implementation is written by Michael Barr and it is published under Public Domain. The implementation consists of two different version of crc, but we will be using the fast version only.

The program consists of two separate files: 'main.c' contains the simple main function and 'crc.c' contains the actual implementation. Open 'crc.c' in your preferred editor and take a look at the code. The main difference between the crcSlow and crcFast implementations is that crcFast exploits precalculated table values. This is a quite usual method of algorithm optimization.

3.1.2 Generating LLVM Bitcode

We first generate LLVM bitcode from the C code. The bitcode will be later compiled to an architecture dependent TTA program. Compile the C code with the 'tcecc' compiler:

```
tcecc -O3 -o crc.bc -k main,result -D_DEBUG crc.c main.c
```

This produces an architecture independent bitcode program 'crc.bc' of the C code. The switch -k was used to tell the compiler to keep the *main* and *result* symbols in the generated program in order to access them by name in the simulator.

3.1.3 Starting Point Processor Architecture

We use the *minimal architecture* as the starting point. File 'minimal_with_stdout.adf' describes a minimalistic architecture containing just enough resources that the TCE compiler can still compile programs for it with the exeption that it also has a function unit for implementing STDOUT in simulation. Function units in 'minimal.adf' are selected from the hardware database (HDB, Section 2.2.2) so we are able to

generate a VHDL implementation of the processor automatically later in the tutorial. With the exception that the STDOUT function unit is not implemented as it's for debugging.

Copy the 'minimal_with_stdout.adf' included in TCE distribution to a new ADF file which is your starting point architecture:

```
cp $(tce-config --prefix)/share/tce/data/mach/minimal_with_stdout.adf start.adf
```

Take a look at the starting point architecture using the graphical Processor Designer (ProDe, Section 4.1) tool. The Start ProDe with:

```
prode start.adf &
```

If you have GHDL installed in your system and you want to simulate the generated processor I suggest you to decrease the amount of memory in the processor. Otherwise the GHDL generated testbench might consume tremendous amount of memory on your computer. To do this select Edit -> Address Spaces in ProDe. Then edit the bit widths of data and instruction address spaces and set them to 15 bits which should be plenty for our case.

3.1.4 Evaluating the Starting Point Architecture

Now we want to know how well the starting point architecture executes our program. First we need to map ("schedule") the generic bitcode program to the parallel architecture we examined in the previous section. This can be done by compiling the bitcode program against the processor architecture with the command:

```
tcecc -O3 -a start.adf -o crc.tpef -k main,result -D_DEBUG crc.bc
```

This will produce a parallel program called 'crc.tpef' that can be executed with the processor design 'start.adf'. The parallel program is now tied to a specified architecture, so it is no longer architecture independent like the bitcode program.

After successfully compiling the program we can now simulate it. Lets use the command line based simulator called ttasim. Start the simulator with:

```
ttasim -a start.adf -p crc.tpef
```

The simulator will load the architecture definition and the program and execute it. After this you should see ttasim prompt: *(ttasim)*. The simulator can show various pieces of information about the execution of the program and utilization of processor resources. As we gave the DEBUG define to the compiler program execution will also print the correct result and produced result of crc.

To see the cycle count use the following command in ttasim:

```
info proc cycles
```

Lets take a look at the processor utilization. Enter:

```
info proc stats
```

This will output a lot of information about the execution, like utilization of transport busses, register files and function units.

You can also check the result straight from processor memory with command:

```
x /u w _result
```

The checksum should be **0x62488e82** like in the debug printing. Next take a look at the graphical user interface of the simulator called Proxim (Section 6.1.6). You can start it with:

```
proxim start.adf crc.tpef &
```

Run the simulation and check the final cycle count it took to execute the program. The cycle count is written in the bottom bar of the simulator.

Proxim can also show various pieces of information about a program's execution and resource utilization. To check the utilization of the resources of our architecture, select *View>Machine Window* from the top menu. Then enable the utilization visualization: *Right click the machine window > Display utilizations*. The parts of the processor that are utilized the most are visualized with darker red color.

Next, check the profiling data to see which instructions were executed the most. Select: *Source>Profile data>Highlight Top Execution Counts*. Now the disassembly window has the most executed instructions highlighted. You should be able to spot the main loop of the program easily.

3.1.5 Initial optimization

As you noticed the cycle count is tremendous. There are two clear reasons for this. The first one is the use of printing functions. They are quite heavy and they are used for debugging and verification. But as we saw earlier the produced can be checked without printing so for benchmarking we can turn the printouts off. This is done by recompiling the program without -D_DEBUG option. The other reason is that the crcTable array is evaluated at runtime in crcInit function. In an actual system the evaluation would only be done once at startup, so the impact on cycle count would be miniscule in the long run. Nevertheless, we will remove this overhead and precalculate the values so we can consentrate on the optimization of the main CRC algorithm instead of the initialization routines.

A simple method to precalculate the array values is to modify the crcInit function so that it prints the array after it has been calculated. The values obtained are included in the tutorial as the file 'crcTable.dat'. Open 'crc.c' and find the line:

```
crc crcTable[256];
```

To load the precalculated values, modify the line to read:

```
crc crcTable[256] = {
#include "crcTable.dat"
};
```

Next open 'main.c' and remove or comment out the crcInit() function call because we now use the precalculated crcTable. Then recompile the code (this time compile it straight from C-code):

```
tcecc -O3 -a start.adf -o crc.tpef -k main,result crc.c main.c
```

And then simulate it again. Verify the result from memory and check cycle count as done earlier. This time you should see a huge drop in the cycle count. Write down the cycle count as a baseline for future comparison.

3.1.6 Modifying the architecture

As the architecture has only one transport bus the compiler can't exploit much concurrency. So let's start architecture customization by adding another transport bus. First copy the starting point architecture:

```
cp start.adf modified.adf
```

and open the new architecture in ProDe:

```
prode modified.adf &
```

A new transport bus can be added simply by selecting the current bus and pressing "ctrl+c" to copy the bus and then pressing "ctrl+v" to paste it. Then double click the new bus and change its name to "B2". Another way to add a bus is to select "Edit->Add..->Transport Bus...". After you have added the bus in either way you have to connect it to the sockets. Easiest way to do this is to select "Tools->Fully connect IC". Save the architecture, recompile the source code for the new architecture and simulate.

```
tcecc -O3 -a modified.adf -o crc.tpef -k main,result crc.c main.c
ttasim -a modified.adf -p crc.tpef
```

Now when you check the cycle count from the simulator:

```
info proc cycles
```

you should see an significant drop in cycles. Try adding a 3rd bus to the architecture and see how it affects the performance. Again a drop in cycle count is expected. This time also check the processor utilization statistics from the simulator with command:

```
info proc stats
```

You can see from the "operations" table that there are a lot of load and store operations being executed. As the architecture has only 5 general purpose registers this tells us that there are a lot of register spilling. Let's try how the amount of registers affect the cycle count. In ProDe double click the RF-component

and double the amount of registers by changing the size to 10. Then save the architecture, recompile and simulate. Check the cycle count and utilization statistics and you should see that indeed the amount of load and store operations decreased.

Because the register file has only one read port and one write port only one register can be read and written on one clock cycle. Let's add another register file to increase the amount of register accesses per cycle. This can be done by selecting the RF and using copy and paste. Then connect it to the IC. Simulation statistics should indicate performance increase.

Next subject for the bottleneck is the ALU as now all the non-memory operations are performed in a single function unit. From the simulator statistics you can see that logical operations and equal comparison are heavily utilized. Instead of dublicating the ALU let's add more specific FUs from the Hardware Database. Select "Edit->Add From HDB->Function Unit...". Select a FU which has operations eq(1), gt(1) and click "Add". Then select FU with operations and(1), ior(1), xor(1) and click "Add". Close the dialog and save the architecture. Recompile and simulate to see the effect on cycle count.

The architecture could be still modified even further to drop the cycle count. But let's settle for this now and move on to another approach to lower the cycle count.

3.1.7 Accelerating the Algorithm

Custom operations implement application specific functionality in TTA processors. In this part of the tutorial we accelerate the CRC computation by adding a custom operation to the processor design.

3.1.7.1 Evaluating Custom Operation Candidates

First of all, it is quite simple and efficient to implement CRC calculation entirely on hardware. Naturally, using the whole CRC function as a custom operation would be quite pointless and the benefits of using a processor based implementation would get smaller. Instead, we will consentrate on trying to accelerate smaller parts of the algorithm, picking a custom operation that is potentially useful also for other algorithms than CRC.

3.1.7.2 Finding the Bottlenecks

The first thing to do when trying to optimize code is to profile the execution. In Proxim you can trace the most executed instructions (see section 6.1.6.2 for more information). Another way to produce program profile data is to enable "setting profile_data_saving 1" in ttasim before initializing the simulation. This produces a crc.tpef.trace file from which one can dump the function profile using a command "dump_function_profile crc.tpef.trace". A call profile helps locating the functions that are executed the most. Refer to section 6.1.4 for further information.

In this case finding the operation to be optimized is quite obvious if you look at function crcFast(). It consists of a for-loop in which the function reflect() is called through the macro REFLECT_DATA. If you look at the actual function you can see that it is quite simple to implement on hardware, but requires many instructions if done with basic operations in software. The function "reflects" the bitpattern around the middle point like a mirror. For example, the bit pattern **0101 0100** would look like this after reflection: **0010 1010**. The main complexity of the function is that the bit pattern width is not fixed. Fortunately, the width cannot be arbitrary. If you examine the crcFast()-function and the reflect macros you can spot that function reflect() is only called with 8 and 32 bit widths (unsigned char and 'crc' which is an unsigned long).

3.1.8 Analyzing the Custom Operation

A great advantage of TCE is that the operation semantics, processor architecture and implementation are separate abstractions. How this affects designing custom operations is that you can simulate your design by simply defining the simulation behaviour of the operation and setting the latency of the operation to the processor architecture definition. This is nice as you do not need an actual hardware implementation of the operation at this point of the design, but can evaluate different custom operation possibilities at the

architectural level. However, this brings up an awkward question: how to determine the latency of the operation? Unrealistic or too pessimistic latency estimates can give inaccurate results or bias the analysis.

One approach to the problem is to take an educated guess and simulate some test cases with different custom operation latencies. This way you can determine a latency range in which the custom operation would accelerate your application to the satisfactory level. After this you can scetch how the operation could be implemented in hardware, or consult someone knowledgeable in hardware design to figure out whether the custom operation is implementable within the latency constraint.

Another approach is to try and determine the latency by examining the operation itself and considering how it could be implemented. This approach requires some insight in digital design.

Besides latency you should also consider the size of the custom function unit. It will consume extra die area, but the size limit is always case-specific. For accurate size estimation you need to have the actual implementation and synthesis.

Let us consider the reflect function. If we had fixed width we could implement the reflect by hard wiring (and registering the output) because the operation only moves bits about. This could be done easily in one clock cycle. But we need two different bit widths so things would be a bit more complicated. We could design the hardware in such way that it has two operations: one for 8-bit data and another for 32-bit data. On hardware one way to implement this is to have 32-bit wide crosswiring and register the output. In this case the 8-bit value would be reflected to the 8 MSB bits of the 32-bit wiring. Then we need to move the 8 MSB bits to the LSB end and zero the rest. This moving can be implemented using multiplexers. So concerning the latency this can all be done easily within one clock cycle as there is not much logic needed.

3.1.9 Using the Custom Operation

Now we have decided the operation to be accelerated and its latency. Next we will create a function unit implementing the operation and add it to our processor design. First, a description of the semantics of the new operation must be added at least to Operation Set Abstraction Layer (Section 2.2.6). OSAL stores the semantic properties of the operation, which includes the simulation behavior, operand count etc., but not the latency. OSAL definitions can be added by using the OSAL GUI, *OSEd* (Section 4.2.1).

If processors that use the custom operation are to be synthesized or simulated at the VHDL level, at least one function unit implementing the operation should be added to the Hardware Database (Section 2.2.2). Cost data of the function unit needs to be added to the cost database if cost estimates of a processor containing the custom function unit are wanted. In this tutorial we add the FU implementation for our custom operation so the processor implementation can be generated, but omit the cost data required for the cost estimation.

Using Operation Set Editor (OSEd) to add the operation data. OSEd is started with the command 'osed'.

Create a new operation module, which is a container for a set of operations. You can add a new module in any of the predefined search paths, provided that you have sufficient file system access on the chosen directory.

For example, choose directory '/home/*user*/.tce/opset/custom', where *user* is the name of the user account being used for the tutorial. This directory is intended for the custom operations defined by the current user, and should always have sufficient access rights.

1. Right-click on a path name in the left area of the main window. A drop-down menu appears below the mouse pointer.

2. Select **Add module** menu item.

3. Type in the name of the module (for example, 'tutorial') and press *OK*. The module is now added under the selected path.

Adding the new operations. We will now add the operation definitions to the newly created operation module.

1. Select the module that you just added by right-clicking on its name, displayed in the left area of the main window. A drop down menu appears.
2. Select **Add operation** menu item.
3. Type 'REFLECT8' as the name of the operation.
4. Add one input by pressing the *Add* button under the operation input list repeatedly. Select *UIntWord* as type.
5. Add one output by pressing the *Add* button under the operation output list. Select *UIntWord* as type.
6. After the inputs and the output of the operation have been added, close the dialog by pressing the *OK* button. A confirmation dialog will pop up. Press *Yes* to confirm the action. The operation definition is now added to the module.
7. Then repeat the steps for operation 'REFLECT32'

Adding simulation behaviour to the operations The new operations REFLECT8 and REFLECT32 does not yet have simulation behavior models, so we cannot simulate programs that use these operations with the TCE processor simulator. Open again the operation property dialog by right-clicking REFLECT8, then choosing *Modify properties*. Now press the *Open* button to open an empty behavior source file for the module. Type in the following code in the editor window:

```
#include "OSAL.hh"
OPERATION(REFLECT8)
 TRIGGER

 unsigned long data = UINT(1);
 unsigned char nBits = 8;

 unsigned long  reflection = 0x00000000;
 unsigned char  bit;

 /*
  * Reflect the data about the center bit.
  */
 for (bit = 0; bit < nBits; ++bit)
 {
     /*
      * If the LSB bit is set, set the reflection of it.
      */
     if (data & 0x01)
     {
         reflection |= (1 << ((nBits - 1) - bit));
     }

     data = (data >> 1);
 }

 IO(2) = static_cast<unsigned> (reflection);

 return true;
 END_TRIGGER;
```

```
 END_OPERATION(REFLECT8)

OPERATION(REFLECT32)
 TRIGGER

 unsigned long data = UINT(1);
 unsigned char nBits = 32;

 unsigned long  reflection = 0x00000000;
 unsigned char  bit;

 /*
  * Reflect the data about the center bit.
  */
 for (bit = 0; bit < nBits; ++bit)
 {
     /*
      * If the LSB bit is set, set the reflection of it.
      */
     if (data & 0x01)
     {
         reflection |= (1 << ((nBits - 1) - bit));
     }

     data = (data >> 1);
 }

 IO(2) = static_cast<unsigned> (reflection);

 return true;
 END_TRIGGER;
 END_OPERATION(REFLECT32)
```

This code has the behaviours for the both operations. These behavior definitions reflect the input operand integer (with id 1) and writes the result to the ”output operand“ (with id 2) which is the first output and signals the simulator that all results are computed successfully.

Open file ’crc.c’ in your preferred editor. Compare the behaviour definition of reflect operations and the original reflect-function. The function is mostly similar except for parameter passing. On hardware the function unit’s data is read from input ports and written to output ports and the nBits-value is hardcoded into the operations. So in the source code, we write the input data to FU port 1 and read the output from port 2.

Save the code and close the editor. REFLECT8 and REFLECT32 operations now have TCE simulator behaviour models.

Compiling operation behavior. REFLECT-operations have been added to the test module. Before we can simulate the behavior of our operation, the module must be compiled.

1. Right-click on the module name displayed in the left area to bring up the drop down menu.
2. Select **Build** menu item.
3. Hopefully, no errors were found during the compilation! Otherwise, re-open the behaviour source file and try to locate the errors with the help of the diagnostic information displayed in the build dialog.

Simulating the operation behavior. Now the operation simulation behavior definition can be tested.

1. Right-click on the operation name displayed in the left area of the main window, under the operation module that contains it.

2. Select **Simulate** menu item from the drop-down menu.

3. Edit the input values of REFLECT8 (or REFLECT32) by selecting the input operand from the list and typing the values in the text input field and pressing the *Update* button next to the field. In this case it is handy to choose the format to be binary.

4. Press the *Trigger* button. The value of the output should be updated.

5. The format of the displayed input and output values can be modified by selecting different formats from the choice list next to the label 'Format:'. It might be more illustrating if you change input and output format to binary. This way we can enter binary patterns, for example '11001100' (as binary) for the first operand. The trigger should produce the reflected value '00110011' for operation REFLECT8

6. The state of operation can be reset by pressing the *Reset* button.

7. Repeat the test with different input values. After you are convinced the behavior definition works, close the dialog by clicking on the *OK* button.

Now the operation definitions of the custom operations have been added to the Operation Set Abstraction Layer (OSAL) database. Next, we need to add at least one functional unit (FU) which implements these operations so that they can be used in the processor design. Note the separation between "operation" and an "function unit" that implements the operation(s) which allows using the same OSAL operation definitions in multiple FUs with different latencies.

First, add the architecture of the FU that implements the custom operationa to the processor design we defined in the previous step. Let us first backup our starting point processor design to another file so we can later more easily compare the architecture with and without the custom operation support:

```
cp start.adf custom.adf
```

Open the copy in ProDe:

```
prode custom.adf &
```

Then:

1. Add a new function unit to the design, right click the canvas and select: *Add>Function Unit*. Name the FU "reflecter". Add one input port (named astrigger) and an output port (output1) to the FU in the Function unit dialog. Set the input port (trigger) triggering (*Click the port named trigger->Edit->Check dialog "triggers"*). This port starts the execution of the operation when it is written to.

2. Add the operation "REFLECT8" we defined to the FU: *Add from opset>REFLECT8>OK* and set the latency to 1. Click on the REFLECT8 operation and ensure that the operation input is bound to the input ports and the output is bound to the output port. Check that the operand usage is in such a way that input is read at cycle 0 and the result is written at the end of the cycle (can be read from the FU on the next cycle). Thus, the latency of the operation is 1 clock cycles.

3. Repeat the previous step for operation "REFLECT32"

4. Now an FU that supports the custom operation has been added to the architecture. Finally, fully connect the machine to connect the FU to the rest of the architecture. This can be done by selecting *Tools->Fully Connect IC*. Finally save the architecture.

3.1.10 Use the custom operation in C code.

To get some benefits from the added custom hardware, we must use it from the C code. This is done by replacing a C statement with a custom operation invocation.

Let us first backup the original C code.

```
cp crc.c crc_with_custom_op.c
```

Then open 'crc_with_custom_op.c' in your preferred text editor.

1. Add #include "tceops.h" to the top of the file. This includes automatically generated macros which allow us to use specific operations from C code without getting our hands dirty with inline assembly.

   Usage of these macros is as follows:

   ```
   _TCE_<name>(input1, ... , inputN, output1, ... , outputN);
   ```

   where <name> is the name of the operation in OSAL. Number of input and output operands depends on the operation. Input operands are given first and they are followed by output operands if any.

   In our case we need to write operands into the reflecter and read the result from it. We named the operations "REFLECT8" and so "REFLECT32" the macros we are going to use are these:

   ```
   _TCE_REFLECT8(input1, output);
   _TCE_REFLECT32(input1, output);
   ```

   Now we will modify the crcFast function to use the custom op. First declare 2 new variables at the beginning of the function:

   ```
   crc input;
   crc output;
   ```

   These will help in using the reflect FU macro.

   Take a look at the REFLECT_DATA and REFLECT_REMAINDER macros. The first one has got a magic number 8 and "variable" X is the data. This is used in the for-loop.

   In the for-loop the input data of reflect function is read from message[]. Let us modify this so that at the beginning of the loop the input data is read to the input variable. Then we will use the _TCE_REFLECT8 macro to run the custom operations, and finally replace the REFLECT_DATA macro with the output variable. After these modifications the body of the for-loop should look like this:

   ```
   input = message[byte];
   _TCE_REFLECT8(input, output);
   data = (unsigned char) output ^ (remainder >> (WIDTH - 8));
   remainder = crcTable[data] ^ (remainder << 8);
   ```

   Next we will modify the return statement. Originally it uses REFLECT_REMAINDER macro where nBits is defined as WIDTH and data is remainder. Simply use _TCE_REFLECT32 macro before return statement and replace the original macro with the variable output:

   ```
   _TCE_REFLECT32(remainder, output);
   return (output ^ FINAL_XOR_VALUE);
   ```

   And now we are ready. Remember to save the file.

2. This time we skip the bitcode phase and compile directly from C to a parallel TTA program using the new architecture which includes a FU with the custom operation:

```
tcecc -O3 -a custom.adf -o crc_with_custom_op.tpef -k main,result \
  crc_with_custom_op.c main.c
```

3. Simulate the parallel program. You can do it quickly with the command line simulator:

```
ttasim -a custom.adf -p crc_with_custom_op.tpef
```

   Verify that the result is the same as before (*x /u w _result*). Check the cycle count *info proc cycles* and compare it to the cycle count of the version which does not use a custom operation. You should see a significant drop compared to the starting point architecture.

   In addition to custom operations, other ways to improve the cycle count further is to add more resources such as registers and transport buses. You can play around with these parameters easily with ProDe. After modifying the architecture, recompile and simulate the code to see the effect to the cycle count.

3.1.11 Adding an implementation of the FU to the hardware database (HDB).

Now we have seen that the custom operation accelerates our application. Next we'll add a VHDL implementation of the custom FU to Hardware Database (hdb). This way we will be able to generate a VHDL implementation of our processor.

Start HDBEditor (see Section 4.5):

```
hdbeditor &
```

TCE needs some data of the FU implementation in order to be able to automatically generate processors that include the FU.

1. Create a new hdb and name it tour.hdb. Add the "reflecter" function unit from 'custom.adf' file (*edit->add->FU architecture from ADF*). You can leave the "parametrized width" and "guard support" unchecked. Then define implementation for the added function unit entry *right click reflect -> Add implementation....*

2. Open file 'tour_vhdl/reflect.vhdl' that was provided in the tutorial package with the editor you prefer, and take a look. This is an example implementation of a TTA function unit performing the custom 'reflect8' and 'reflect32' operations.

3. The HDB implementation dialog needs the following information from the VHDL:

   1. Name of the entity and the naming of the FU interface ports.

   Name the implemention after the top level entity: "fu_reflect".

   By examining the VHDL code you can easily spot the clock port (clk), reset (rstx) and global lock port (glock). Write these into the appropriate text boxes. You do not have to fill the Opcode port or Global lock req. port fields because the function unit has only one opcode and does not need to cause a global lock to the processor during its execution.

   2. Parameters.

   Parameters can be found from the VHDL file. On top of the file there is one parameter: busw. It tells the width of transport bus and thus the maximum width of input and output operands.

   Thus, add a 32-bit integer parameter named busw to the Parameter dialog.

   3. Architecture ports.

   This settings defines the connection between the ports in the architectural description of the FU and the VHDL implementation. Each input data port in the FU is accompanied with a load port that controls the updating of the FU input port registers.

Choose a port in the Architecture ports dialog and click edit. Name of the architecture port p1 is t1data and load port is t1load. Width formula is the parameter busw.

Name the output port (p2) to r1data and the width formula is now busw because the output port writes to the bus. The output port does not have a load port.

4. Add VHDL source file.

Add the VHDL source file into the Source code dialog. Notice that the HDL files must be added in the compilation order (see section 4.5). But now we have only one source file so we can simply add it without considering the compilation order (Add -> Browse -> tour_vhdl/reflect.vhdl).

Now you are done with adding the FU implementation. Click OK.

3.1.12 Generating the Final Products

In this step we generate the VHDL implementation of the processor, and the bit image of the parallel program.

Preparations for verification Before continuing any further remove the IO-function unit from the architecture as it is not implemented on hardware. Use Prode to delete the FU:

```
prode custom.adf &
```

Save the architecture and compile the program without -D_DEBUG option.

```
tcecc -O3 -a custom.adf -o crc_with_custom_op.tpef -k main,result \
  crc_with_custom_op.c main.c
```

Next we need to generate bus trace from the instruction set simulator so we can verify the RTL simulation. Launch ttasim:

```
ttasim
```

Set bus trace option on:

```
setting bus_trace 1
```

and then load architecture and program:

```
mach custom.adf
prog crc_with_custom_op.tpef
run
```

and after simulation check the cycle count and write it down. You can now quit the simulator. The simulator execution created file 'crc_with_custom_op.tpef.bustrace' which contains the bus trace.

Creating binary encoding and selection implementations Then create an encoding for the instructions in the architecture:

```
createbem custom.adf
```

This should generate 'custom.bem' which is a description on how instructions should be encoded in the architecture.

You can print the encoding info in a more human readable format with:

```
viewbem custom.bem
```

The program should print details on how each type of source/destination, etc. is encoded. You do not have to care about this, it is just for curiosity. Maybe the only important part in the printout is the length of the instruction word. In this case, even though the processor has only one move slot, the instruction word width is more than 40 bits. The size can be reduced with instruction compression and smaller immediate width, which is skipped in the tutorial.

Next, we must select implementations for all components in the architecture. Each architecture component can be implemented in multiple ways, so we must choose one implementation for each component to be able to generate the implementation for the processor.

This can be done in the ProDe tool:

```
prode custom.adf
```

Then we'll select implementations for the FUs which can be done in *Tools>Processor Implementation...*. Note that the selection window is not currently very informative about the different implementations, so a safe bet is to select an implementation with parametrizable width/size.

1. Select implementation for RF: Click the RF name, 'Select RF implementation', find the TCE's default HDB file from your tce installation path (PREFIX/share/tce/hdb/asic_130nm_1.5V.hdb) and select an implementation for the RF from there.

2. Next select implementation for the boolean RF like above. But this time select an implementation which is guarded i.e. select an implementation which has word "guarded_0" in its name.

3. Similarly, select implementations for the function units from TCE's default HDB. Then select implementation for the custom_adder but this time you have to use the 'tour.hdb' created earlier to find the FU we added that supports the REFLECT custom operations.

4. Next select the IC/Decoder generator plugin used to generate the decoder in the control unit and interconnection network: *Browse... (installation_path)/share/tce/icdecoder_plugins/base/ DefaultICDecoderPlugin.so>OK*. This should be selected by default.

5. Enable bus tracing from the Implementation-dialog's IC / Decoder Plugin tab. Set the bustrace plugin parameter to "yes" and the bustracestartingcycle to "5". The IC will now have a component which writes the bus value from every cycle to a text file. Notice that this option cannot be used if the processor is synthesized.

   You do not have to care about the HDB file text box because we are not going to use cost estimation data.

Generate the VHDL for the processor using Processor Generator (ProGe). You can start processor generation from the implementation selection dialog: Click "Generate Processor". For Binary Encoding Map: Select the previously generated .bem (Load from file...). Create and set target directory 'proge-output'. Or alternatively, save the .idf file and execute ProGe from command line:

```
generateprocessor -b custom.bem -i custom.idf -o proge-output custom.adf
```

Now directory 'proge-output' includes the VHDL implementation of the designed processor except for the instruction memory width package which will be created by Program Image Generator. You can take a look what the directory includes, how the RF and FU implementations are collected up under 'vhdl' subdir and the interconnection network has been generated to connect the units (the 'gcu_ic' subdir). The 'tb' subdir contains testbench files for the processor core.

Generate instruction memory bit image using Program Image Generator. Finally, to get our shiny new processor some bits to chew on, we use generatebits to create instruction memory and data memory images:

```
generatebits -b custom.bem -d -w 4 -p crc_with_custom_op.tpef -x proge-output custom.adf
```

Now the file 'crc_with_custom_op.img' includes the instruction memory image in "ascii 0/1" format . Each line in that file represents a single instruction. Thus, you can get the count of instructions by counting the lines in that file:

```
wc -l crc_with_custom_op.img
```

By multiplying the count with the instruction width we found earlier you can get the required size of instruction memory for the program.

Accoringly the file 'crc_with_custom_op_data.img' contains the data memory image of the processor. Program Image Generator also created file 'proge-output/gcu_ic/imem_mau_width.vhdl' which contains the correct MAU width for the instruction memory.

Simulation and verification If you have GHDL installed you can now simulate the processor VHDL. First copy the memory images to the testbench directory:

```
cd proge_output
cp ../crc_with_custom_op_data.img tb/dmem_init.img
cp ../crc_with_custom_op.img tb/imem_init.img
```

Then compile and simulate the testbench:

```
./ghdl_compile.sh
./ghdl_simulate.sh
```

This will take some time as the bus trace writing is enabled. The simulation produces file “bus.dump”. As the testbench is ran for constant amount of cycles we need to get the relevant part out of the bus dump for verification. This can be done with command:

```
head -n <number of cycles> bus.dump > sim.dump
```

where the <number of cycles> is the number of cycles in the previous ttasim execution. Then compare the trace dumps:

```
diff -u sim.dump ../crc_with_custom_op.tpef.bustrace
```

If the command does not print anything the dumps were equal.

3.1.13 Final Words

This tutorial is now finished. Now you should know how to make and use your own custom operations and generate the processor implementation along with its instruction memory bit image.

In this tutorial we used a “minimalistic” processor architecture as our starting point. The machine had only one transport bus and 5 registers so it could not fully exploit the parallel capabilities of TTA. As mentioned earlier, if you are interested you can open the custom.adf in ProDe and add more transport buses, register files and function units and see how they affect the performance.

3.2 From C to VHDL as Quickly as Possible

This tutorial introduces a fast way to generate a processor VHDL model from C source code using the Design Space Explorer.

If you haven’t already downloaded the tutorial file package, you can get it from:
`http://tce.cs.tut.fi/tutorial_files/tce_tutorials.tar.gz`
Unpack it to a working directory and then cd to tce_tutorials/c2vhdl/

A script named c2vhdl should generate all the needed files from the C source code. The script is a bash script using various TCE command line tools, and can be inspected for detailed instructions to generate the VHDL and other files manually.

```
c2vhdl application1/complex_multiply.c
```

Now directory called “proge-output” should include the VHDL implementation of the processor. The current working directory should also contain the instruction memory bit image file and the instruction encoding (bem) file along with the processor architecture definition and implementation definition files.

Passing "-e" as the first parameter to c2vhdl script tells it to print estimation data, like area and energy requirements for the processor it generates.

Tutorial is now finished and you can simulate the generated VHDL implementation of the processor with a VHDL simulator and synthesize it.

3.3 Hello TTA World!

What would a tutorial be without the traditional “Hello World!” example? Interestingly enough, printing out “Hello World” in a standalone (operating system free) platform like the TTA of TCE is not totally straightforward. That is the reason this tutorial is not the first one in this tutorial chapter.

The first question is: where should I print the string? Naturally it is easy to answer that question while working with the simulator: to the simulator console, of course. However, when implementing the final hardware of the TTA, the output is platform dependent. It can be a debugging interface, serial port output, etc.

In order to make printing data easier from TTAs designed with TCE, we have included an operation called STDOUT in the "base operation set" that is installed with the TCE. The simulation behavior of the operation reads its input, expects it to be a 8bit char and writes the char verbatim to the simulator host's standard output. Of course, in case the designer wants to use the operation in his final system he must provide the platform specific implementation for the function unit (FU) implementing the operation, or just remove the FU after the design is fully debugged and verified.

The default implementation of *printf()* in TCE uses the STDOUT operation to print out data. Therefore, implementing a "Hello World" application with TCE is as simple as adding an FU that implements the STDOUT to the processor design and then calling *printf()* in the simulated program. The simulator should print out the greeting as expected.

Here is a simple C code (hello.c) that prints the magic string:

```
#include <stdio.h>

int main() {
    printf("Hello TTA World!");
    return 0;
}
```

Next, compile it to an architecture that implements the STDOUT. In this case we add an FU with STDOUT (see the previous tutorial for instructions on adding FUs to your designs) to the minimal.adf, after copying it to minimalWithStdout.adf:

```
cp $(tce-config --prefix)/share/tce/data/mach/minimal.adf minimalWithStdout.adf
prode minimalWithStdout.adf &
... add the IO function unit with the STDOUT operation to the ADF ...
tcecc -O0 hello.c -a minimalWithStdout.adf -o hello.tpef
```

It should compile without errors. Beware: the compilation can take a couple of minutes on a slower machine! This is because *printf()* is actually quite a large function and the compiler is not yet optimized for speed.

Finally, simulate the program to get the greeting:

```
ttasim -a minimalWithStdout.adf -p hello.tpef --no-debugmode
Hello TTA World!
```

That's it. Happy debugging!

3.4 Streaming I/O

Because TTA/TCE is an environment without operating system, there is also no file system available for implementing file-based I/O. Therefore, one popular way to get input and output to/from the TTA is using shared memory for communicating the data. For stream processing type of applications, one can also use an I/O function unit that implements the required operations for streaming.

TCE ships with example operations for implementing stream type input/output. These operations can be used to read and write samples from streams in the designed TTA processor. The basic interface of the operations allows reading and writing samples from the streams and querying the status of an input or output stream (buffer full/empty, etc.). The status operations are provided to allow the software running in the TTA to do something useful while the buffer is empty or full, for example switch to another thread.

Otherwise, in case one tries to read/write a sample from/to a stream whose buffer is empty/full, the TTA is locked and the cycles until the situation resolves are wasted.

The example streaming operations in the base operation set are called STREAM_IN, STREAM_OUT, STREAM_IN_STATUS, and STREAM_OUT_STATUS. These operations have a simulation behavior definition which simulates the stream I/O by reading/writing from/to files stored in the file system of the simulator host. The files are called *ttasim_stream.in* for the input stream and *ttasim_stream.out* for the output stream and should reside in the directory where the simulator is started. The file names can be customized using environment variables TTASIM_STREAM_OUT_FILE and TTASIM_STREAM_IN_FILE, respectively.

Here is an example C code that implements streaming I/O with the operations:

```
#include "tceops.h"

int main()
{
    char byte;
    int status;

    while (1)
    {
        _TCE_STREAM_IN_STATUS(0, status);

        if (status == 0)
            break;

        _TCE_STREAM_IN(0, byte);
        _TCE_STREAM_OUT(byte);
    }

    return 0;
}
```

This code uses the TCE operation invocation macros from *tceops.h* to read bytes from the input stream and write the same bytes to the output stream until there is no more data left. This situation is indicated with the status code 0 queried with the STREAM_IN_STATUS operation. The value means the stream buffer is empty, which means the file simulating the input buffer has reached the end of file.

You can test the code by creating a file *ttasim_stream.in* with some test data. The code should create a copy of that file to the stream output file *ttasim_stream.out*.

3.5 Implementing Programs in Parallel Assembly Code

This tutorial will introduce you to TTA assembly programming. It is recommended that you go through this tutorial because it will certainly familiarize you with TTA architecture and how TTA works.

3.5.1 Preparations

For the tutorial you need to download file package from `http://tce.cs.tut.fi/tutorial_files/tce_tutorials.tar.gz` and unpack it to a working directory. Then cd to parallel_assembly-directory.

The first thing to do is to compile the custom operation set called cos16 shipped within the parallel_assembly-directory. The easiest way to do this is:

```
buildopset cos16
```

This should create a file named 'cos16.opb' in the directory.

3.5.2 Introduction to DCT

Now you will be introduced to TCE assembler language and assembler usage. Your task is to write TCE assembly code for 2-Dimensional 8 times 8 point Discrete Cosine Transform (DCT_8x8). First take a look at the C code of DCT_8x8 'dct_8x8_16_bit_with_sfus.c'. The code is written to support fixed point datatype with sign plus 15 fragment bits, which means coverage from -1 *to* $1-2^{15}$. The fixed point multiplier, function *mul_16_fix*, and fixed point adder, function *add_16_fix*, used in the code scale inputs automatically to prevent overflow. Function *cos16* takes $x(2i+1)$ as input and returns the corresponding cosine value $\frac{cos(x(2i+1)\pi)}{16}$. The code calculates following equations:

$$F(x) = \frac{C(x)}{2}\sum_{i=0}^{7}\left[f(i)cos\left(\frac{x(2i+1)\pi}{16}\right)\right] \tag{3.1}$$

$$F(y) = \frac{C(y)}{2}\sum_{i=0}^{7}\left[f(i)cos\left(\frac{y(2i+1)\pi}{16}\right)\right] \tag{3.2}$$

$$C(i) = \begin{cases} \frac{2}{\sqrt{2}} & ,i=0 \\ 1 & ,else \end{cases} . \tag{3.3}$$

$$F(x,y) = F(x)F(y) \tag{3.4}$$

3.5.3 Introduction to TCE assembly

First take a look at assembly example in file 'example.tceasm' to get familiar with syntax. More help can be found from section 5.3

Compilation of the example code is done by command:

```
tceasm -o example.tpef dct_8x8_16_bit_with_sfus.adf example.tceasm
```

The assembler will give some warnings saying that "Source is wider than destination." but these can be ignored.

The compiled tceasm code can be simulated with TCE simulator, ttasim or proxim(GUI).

```
ttasim -a dct_8x8_16_bit_with_sfus.adf -p example.tpef
```
, or

```
proxim dct_8x8_16_bit_with_sfus.adf example.tpef
```

It is recommended to use proxim because it is more illustrating to track the execution with it. Especially if you open the Machine Window (View -> Machine Window) and step through the program.

Check the result of example code with command (you can also write this in proxim's command line at the bottom of the main window):

```
x /a IODATA /n 1 /u b 2 .
```

the output of x should be 0x40.

3.5.4 Implementing DCT on TCE assembly

Next try to write assembly code which does the same functionality as the C code. The assembly code must be functional with the given machine 'dct_8x8_16_bit_with_sfus.adf'. Take a look at the processor by using prode:

```
prode dct_8x8_16_bit_with_sfus.adf &
```

The processor's specifications are the following:

Supported operations
Operations supported by the machine are: *mul*, *mul_16_fix*, *add*, *add_16_fix*, *ldq*, *ldw*, *stq*, *stw*, *shr*, *shl*, *eq*, *gt*, *gtu*, *jump* and *immediate transport*.

When you program using TTA assembly you need to take into account operation latencies. The *jump* latency is four clock cycles and load latencies (*ldq* and *ldw*) are three cycles. Latency for multiplications (*mul* and *mul_16_fix*) are two clock cycles.

Address spaces
The machine has two separate address spaces, one for data and another for instructions. The data memory is 16-bit containing 128 memory slots and the MAU of data memory is 16-bits. The intruction memory has 1024 memory slots which means that the maximum number of instructions of 1024.

Register files
The machine contains 4 register files, each of which have 4 16-bit registers, leading to total of 16 16-bit registers. The first register file has 2 read ports.

Transport buses
The machine has 3 16-bit buses, which means maximum of 3 concurrent transports. Each bus can contain a 8-bit short immediate.

Immediates
Because the transport buses can only contain 8-bit short immediates you must use the immediate unit if you want to use longer immediates. The immediate unit can hold a 16-bit immediate. There is an example of immediate unit usage in file 'immediate_example.tceasm'. Basically you need to transfer the value to the immediate register. The value of immediate register can be read on the next cycle.

The initial input data is written to memory locations 0-63 in the file 'assembler_tutorial.tceasm'. Write your assembly code in that file.

3.5.4.1 Verifying the assembly program

The reference output is given in 'reference_output'. You need to compare your assembly program's simulation result to the reference output. Comparision can be done by first dumping the memory contents in the TCE simulator with following command:

```
x /a IODATA /n 64 /u b 0
```

The command assumes that output data is stored to memory locations 0-63.

The easiest way to dump the memory into a text file is to execute ttasim with the following command:

```
ttasim -a dct_8x8_16_bit_with_sfus.adf -p assembler_tutorial.tpef < input_command.txt
> dump.txt
```

After this you should use sed to divide the memory dump into separete lines to help comparison between your output and the reference output. Use the following command to do this (there is an empty space between the first two slashes of the sed expression):

```
cat dump.txt | sed 's/ /   \n/g' > output.txt
```

And then compare the result with reference:

```
diff -u output.txt reference_output
```

When the TCE simulator memory dump is the same as the reference output your assembly code works and you have completed this tutorial. Of cource you might wish to improve your assembly code to minimize cycle count or/and instruction count.

If it is too hard to visualize the whole program in parallel assembly you can start by writing sequential code and then write it to parallel assembly.

You should also compile the C program and run it because it gives more detailed information which can be used as reference data if you need to debug your assembly code.

To compile the C code, enter:

```
gcc -o c_version dct_8x8_16_bit_with_sfus.c
```

If you want the program to print its output to a text file, you can use the following command:

```
./c-version > output.txt
```

To get some idea of the performance possibilities of the machine, one assembly code has 52 instructions and it runs the DCT8x8 in 3298 cycles.

3.6 Running TTA on FPGA

3.6.1 Introduction

This tutorial illustrates how you can run your TTA designs on a FPGA board. Tutorial consists of two simple example sections and a more general case description section.

Download the tutorial file package from:

`http://tce.cs.tut.fi/tutorial_files/tce_tutorials.tar.gz`

Unpack it to a working directory and cd to tce_tutorials/fpga_tutorial

3.6.2 Simplest example: No data memory

3.6.2.1 Introduction

This is the most FPGA board independent TTA tutorial one can make. The application is a simple led blinker which has been implemented using register optimized handwritten TTA assembly. In other words the application doesn't need a load store unit so there is no need to provide data memory. In addition the instruction memory will be implemented as a logic array.

3.6.2.2 Application

The application performs a 8 leds wide sweep in an endless loop. Sweep illuminates one led at a time and starts again from first led after reaching the last led. There is also a delay between the iterations so that the sweep can be seen with human eye.

As stated in the introduction the application is coded in assembly. If you went through the assembly tutorial the code is probably easy to understand. The code is in file 'blink.tceasm'. The same application is also written in C code in file 'blink.c'.

3.6.2.3 Create TTA processor core and instruction image

The architecture we're using for this tutorial is 'tutorial1.adf'. Open it in ProDe to take a look at it:

```
prode tutorial1.adf
```

As you can see it is a simple one bus architecture without a LSU. There are also 2 "new" function units: rtimer and leds. Rtimer is a simple tick counter which provides real time clock or countdown timer operations. Leds is function unit that can write '0' or '1' to FPGA output port. If those ports are connected to leds the FU can control them.

Leds FU requires a new operation definition and the operation is defined in 'led.opp' and 'led.cc'. You need to build this operation defintion:

```
buildopset led
```

Now you can compile the assembly code:

```
tceasm -o asm.tpef tutorial1.adf blink.tceasm
```

If you wish you can simulate the program with proxim and see how it works but the program runs in endless loop and most of the time it stays in the "sleep" loop.

Now you need to select implementations for the function units. This can be done in ProDe. See TCE tour section 3.1.12 for more information. Implementations for leds and rtimer are found from the fpga.hdb shipped with the tutorial files. Notice that there are 2 implementations for the rtimer. ID 3 is for 50 MHz clock frequency and ID 4 for 100 MHz. All other FUs are found from the default hdb.

After you have generated 'tutorial1.idf' generate the binary encoding for the arhitecture:

```
createbem tutorial1.adf
```

Next step is to generate vhdl of the processor:

```
generateprocessor -b tutorial1.bem -i tutorial1.idf -o asm_vhdl/proge-output tutorial1.adf
```

And after that create proram image:

```
generatebits -b tutorial1.bem -f vhdl -p asm.tpef -x asm_vhdl/proge-output tutorial1.adf
```

Notice that the instruction image format is "vhdl" and we request generatebits to create data image at all. Now move the generated 'asm_imem_pkg.vhdl' to the asm_vhdl directory and cd there.

```
mv asm_imem_pkg.vhdl asm_vhdl/
cd asm_vhdl
```

3.6.2.4 Final steps to FPGA

We have successfully created the processor core and instruction memory image. Now we need an instruction memory component that can use the generated image. Luckily you don't have to create it as it is shipped with the tutorial files. The component is in file 'inst_mem_logic.vhd' in asm_vhdl directory and it can use the generated 'asm_imem_pkg.vhdl' without any modifications.

Next step is to connect TTA toplevel core to the memory component and connect the global signals out from that component. This has also been done for you in file 'tutorial_processor1.vhdl'. If you are curious how this is done open the file with your preferred text editor. All the signals coming out of this component are later connected to FPGA pins.

Now you need to open your FPGA tool vendor's FPGA design/synthesis program and create a new project for your target FPGA. Add the three files in asm_vhdl-directory (toplevel file 'tutorial_processor1.vhdl', 'inst_mem_logic.vhd' and 'asm_imem_pkg.vhdl') and all the files in proge-output/gcu_ic/ and proge-output/vhdl directories to the project.

Then connect the toplevel signals to appropriate FPGA pins. The pins are most probably described in the FPGA board's user manual. Signal 'clk' is obviously connected to the pin that provides clock signal. Signal 'rstx' is the reset signal of the system and it is active low. Connect it to a switch or pushbutton that provides '1' when not pressed. Signal bus 'leds' is 8 bits wide and every bit of the bus should be connected to an individual led. Don't worry if your board doesn't have 8 user controllable leds, you can leave some of them unconnected. In that case all of the leds are off some of the time.

Compile and synthesize your design with the FPGA tools, program your FPGA and behold the light show!

3.6.3 Second example: Adding data memory

3.6.3.1 Introduction

In this tutorial we will implement the same kind of system as above but this time we include data memory and use C coded application. Application has the same functionality but algorithm is bit different. This time we read the led pattern from a look up table and to also test store operation the pattern is stored back to the look up table. Take a look at file 'blink_mem.c' to see how the timer and led operations are used in C code.

3.6.3.2 Create TTA processor core and binary images

The architecture for this tutorial is 'tutorial2.adf'. This architecture is the same as 'tutorial1.adf' with the exception that now it has a load store unit to interface it with data memory.

You need to compile the operation behaviour for the led function unit if you already haven't done it:

```
buildopset led
```

Then compile the program:

```
tcecc -O3 -a tutorial2.adf -o blink.tpef blink_mem.c
```

Before you can generate processor vhdl you must select implementations for the function units. Open the architecture in ProDe and select Tools->Processor Implementation...

```
prode tutorial2.adf
```

It is important that you choose the implementation for LSU from the fpga.hdb shipped with the tutorial files. This implementation has more FPGA friendly byte enable definition. Also the implementations for leds and timer FUs are found from fpga.hdb. As mentioned in the previous tutorial, timer implementation ID 3 is meant for 50 MHz clock frequency and ID 4 for 100 MHz clock. Other FUs are found from the default hdb.

Now create binary enconding for the architecture and generate the processor:

```
createbem tutorial2.adf
generateprocessor -b tutorial2.bem -i tutorial2.idf -o c_vhdl/proge-output tutorial2.adf
```

Next step is to generate binary images of the program. Instruction image will be generated again as a vhdl array package. But the data memory image needs some consideration. If you're using Altera FPGA board the Program Image Generator can output Altera's Memory Initialization Format (mif) by default. Otherwise you need to consult the FPGA vendor's documentation to see what kind of format is used for memory instantiation. Then select the PIG output format that you can convert to the needed format with the least amount of work. Of course you can also implement a new image writer class to PIG. Patches are welcome.

Image generation command is basically the following:

```
generatebits -b tutorial2.bem -f vhdl -d -w 4 -o mif -p blink.tpef -x c_vhdl/proge-output
tutorial2.adf
```

Switch '-d' tells PIG to generate data image. Switch '-o' defines the data image output format. Change it suite your needs if necessary. Switch '-w' defines the width of data memory in MAUs. By default MAU is assumed to be 8 bits and the default LSU implementations are made for memories with 32-bit data width. Thus the width of data memory is 4 MAUs.

Move the created images to the vhdl directory:

```
mv blink_imem_pkg.vhdl c_vhdl/
mv blink_data.mif c_vhdl/
```

3.6.3.3 Towards FPGA

Go to the vhdl directory:

```
cd c_vhdl
```

TTA vhdl codes are in the proge-output directory. Like in the previous tutorial file 'inst_mem_logic.vhd' holds the instruction memory component which uses the created 'blink_imem_pkg.vhdl'. File 'tutorial_processor2.vhdl' is the toplevel design file and again the TTA core toplevel is connected to the instruction memory component and global signals are connected out from this design file.

Creating data memory component
Virtually all FPGA chips have some amount of internal memory which can be used in your own designs. FPGA design tools usually provide some method to easily create memory controllers for those internal memory blocks. For example Altera's Quartus II design toolset has a MegaWizard Plug-In Manager utility which can be used to create RAM memory which utilizes FPGA's internal resources.

There are few points to consider when creating a data memory controller:

1. **Latency.** Latency of the memory should be one clock cycle. When LSU asserts a read command the result should be readable after one clock cycle. This means that the memory controller shouldn't register the memory output because the registering is done in LSU. Adding an output register would increase read latency and the default LSU wouldn't work properly.

2. **Address width.** As stated before the minimal addressable unit from the TTA programmer's point of view is 8 bits by default. However the width of data memory bus is 32 bits wide in the default implementations. This also means that the address bus to data memory is 2 bits smaller because it only needs to address 32-bit units. To convert 8-bit MAU addresses to 32-bit MAU addresses one needs to leave the 2 bits out from LSB side.

   How this all shows in TCE is that data memory address width defined in ADF is 2 bits wider than the actual address bus coming out of LSU. When you are creating the memory component you

should consider this.

3. **Byte enable.** In case you were already wondering how can you address 8-bit or 16-bit wide areas from a 32-bit addressable memory the answer is byte enable (or byte mask) signals. These signals can be used to enable individual bytes from 32-bit words which are read from or written to the memory. And those two leftover bits from the memory address are used, together with the memory operation code, to determine the correct byte enable signal combination.

   When you are creating the memory controller you should add support for byte enable signals.

4. **Initialization.** Usually the internal memory of FPGA can be automatically initialized during FPGA configuration. You should find an option to initialize the memory with a specific initialization file.

Connecting the data memory component
Next step is to interface the newly generated data memory component to TTA core. LSU interface is the following:

```
fu_lsu_data_in      : in  std_logic_vector(fu_lsu_dataw-1 downto 0);
fu_lsu_data_out     : out std_logic_vector(fu_lsu_dataw-1 downto 0);
fu_lsu_addr         : out std_logic_vector(fu_lsu_addrw-2-1 downto 0);
fu_lsu_mem_en_x     : out std_logic_vector(0 downto 0);
fu_lsu_wr_en_x      : out std_logic_vector(0 downto 0);
fu_lsu_bytemask     : out std_logic_vector(fu_lsu_dataw/8-1 downto 0);
```

Meanings of these signals are:

Signal name	Description
fu_lsu_data_in	Data from the memory to LSU
fu_lsu_data_out	Data from LSU to memory
fu_lsu_addr	Address to memory
fu_lsu_mem_en_x	Memory enable signal which is active low. LSU asserts this signal to '0' when memory operations are performed. Otherwise it is '1'. Connect this to memory enable or clock enable signal of the memory controller.
fu_lsu_wr_en_x	Write enable signal which is active low. During write operation this signal is '0'. Read operation is performed when this signal '1'. Depending on the memory controller you might need to invert this signal.
fu_lsu_bytemask	Byte mask / byte enable signal. In this case the signal width is 4 bits and each bit represents a single byte. When the enable bit is '1' the corresponding byte is enabled and value '0' means that the byte is ignored.

Open file 'tutorial_processor2.vhdl' with your preferred text editor. From the comments you can see where you should add the memory component declaration and component instantiation. Notice that those LSU signals are connected to wires (signals with appendix '_w' in the name). Use these wires to connect the memory component.

Final steps
After you have successfully created the data memory component and connected it you should add the rest of the design VHDL files to the design project. All of the files in proge-output/gcu_ic/ and proge-output/vhdl/ directories need to be added.

Next phase is to connect toplevel signals to FPGA pins. Look at the final section of the previous tutorial for more verbose instructions how to perform pin mapping.

Final step is to synthesize the design and configure the FPGA board. Then sit back and enjoy the light show.

3.6.3.4 More to test

If you simulate the program you will notice that the program uses only STW and LDW operations. Reason for this can be easily seen from the source code. Open 'blink_mem.c' and you will notice that the look up table 'patterns' is defined as 'volatile unsigned int'. If you change this to 'volatile unsigned char' or

'volatile unsigned short int' you can test STQ and LDQU or STH and LDHU operations. Using these operations also means that the LSU uses byte enable signals.

Whenever you change the source code you need to recompile your program and generate the binary images again. And move the images to right folder if it's necessary.

In addition you can compile the code without optimizations. This way the compiler leaves function calls in place and uses stack. The compilation command is then:

```
tcecc -O0 -a tutorial2.adf -o blink.tpef blink_mem.c
```

3.6.4 FPGA process in general

Yet to be written.

Chapter 4

PROCESSOR DESIGN TOOLS

4.1 TTA Processor Designer (ProDe)

Processor Designer (**ProDe**) is a graphical application mainly for viewing, editing and printing processor architecture definition files. It also allows selecting implementation for each component of the processor, and generating the HDL implementation of the processor. The application is very easy to use and intuitive, thus this section provides help only for the most common problematic situations encountered while using the toolset.

Input: ADF

Output: ADF, VHDL

The main difficulty in using the tool is to understand what is being designed, that is, the limitations placed by the processor template. Details of the processor template are described in [CSJ04].

4.1.0.1 Usage

Processor Desinger can simply be executed from command line with:

```
prode
```

4.2 Operation Set Abstraction Layer (OSAL) Tools

Input: OSAL definitions

Output: OSAL definitions

4.2.1 Operation Set Editor (OSEd)

Operation Set Editor (**OSEd**) is a graphical application for managing the OSAL (Section 2.2.6) operation database. OSEd makes it possible to add, simulate, edit and delete operation definitions.

4.2.1.1 Capabilities of the OSEd

OSEd is capable of the following operations:

1. All operations found in pre-defined search paths (see Section 4.3) are organised in a tree-like structure which can be browsed.
2. Operation properties can be examined and edited.
3. Operations with a valid behavior model can be tested (simulated).

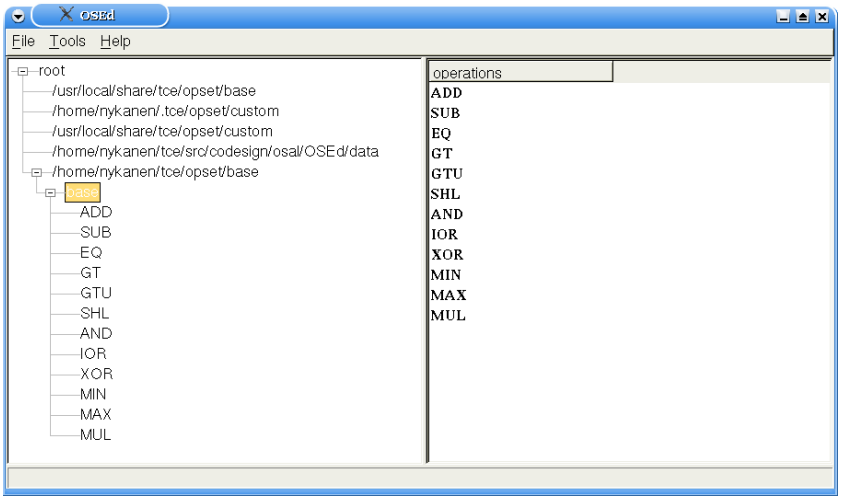

Figure 4.1: OSEd Main window.

4. New operation modules can be added to search paths.

5. Operation definitions can be added to a module.

6. Modules containing operation behaviors can be compiled, either all at once, or separately.

7. Modules can be removed.

8. Contents of the memory can be viewed and edited.

4.2.1.2 Usage

This chapter introduces the reader to the usage of OSEd. Instructions to accomplish the common tasks are given in detail.

Operation Set Editor can simply be executed from command line with:

```
osed
```

The main window is split in two areas. The left area always displays a tree-like structure consisting of search paths for operation definition modules, operation modules, and operations. The right area view depends on the type of the item that is currently selected in the left area. Three cases are possible.

1. If the selected item is a search path, the right area shows all operation modules in that path.

2. If the item is a module, the right area shows all the operations defined in the module.

3. If the item is an operation, the right area displays all the properties of the operation.

Figure 4.1 shows an example of the second situation, in which the item currently selected is a module. The right area of the window shows all the operations in that module. If an operation name is shown in bold text, it means that the operation definition is “effective”, that is, it will actually be used if clients of OSAL request an operation with that name. An operation with a given name is effective when it is the first operation with that name to be found in the search paths. Other operations with the same name may be found in paths with lower search priority. Those operations are not effective.

Figure 4.2 shows an example of an operation property view, that is shown in the right side when an operation is selected on the left side.

property	value	operand value
name	mul	
inputs	2	
outputs	1	
reads memory	no	
writes memory	no	
can trap	no	
has side effects	no	
affected by	none	
affects	none	
input operands	id: 1	
	optional	no
	memory address	no
	memory data	no
	can swap	id: 2
	id: 2	
	optional	no
	memory address	no
	memory data	no
	can swap	id: 1
output operands	id: 3	
	optional	no
	memory data	no
has behavior	yes	

Figure 4.2: Operation property view.

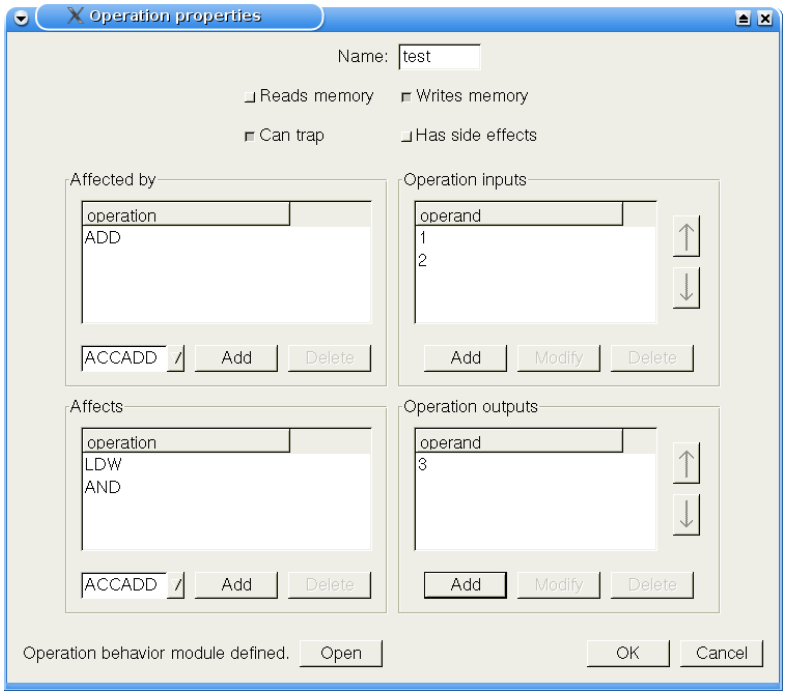

Figure 4.3: Operation property window

Editing Static Operation Properties Figure 4.3 shows the dialog for editing the static properties of an operation.

Operation inputs and outputs (henceforth, “terminal” is used to denote both) can be deleted by selecting an item from the list and clicking the *Delete* button. New terminals can be added by clicking the *Add* button, and can be modified by clicking the *Modify* button. The order of the terminals can be changed by selecting a terminal and pushing on of the arrow buttons. By pushing the downward arrow button, the terminal is moved one step downwards in the list; by pushing the upward arrow button, it is moved one

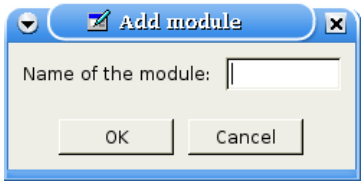

Figure 4.4: Dialog for adding new operation module.

step up on the list.

Operand properties can be modified by clicking on the check boxes. The set of operands that can be swapped with the operand being edited are shown as a list of references to operands (input indentification numbers). A reference to an operand can be removed by selecting the corresponding item in the 'can swap' list and clicking the *Delete* button. A new reference to another operand can be added by selecting an item from the choice list and clicking the *Add* button.

Operation Behaviour Model Behaviour models of operations are stored in separate source files. If the operation definition includes a behaviour model, the behaviour source file can be opened in a text editor of choice by clicking on the *Open* button. If the operation does not have behavior source file, clicking *Open* will open an empty file in an editor. The text editor to use can be defined in the options dialog. All changes to operation properties are committed by clicking the *OK* button and canceled by clicking the *Cancel* button.

Operation Directed Acyclic Graph By treating each operation as a node and each input-output pair as an directed arc, it is possible to construct operation's Directed Acyclic Graph (DAG) presentation. For primitive operations which do not call any other operations, this graph is trivial; one node (operation itself) with input arcs from root nodes and output arcs to leafs. With OSAL DAG language, it is possible to define operation behavior model by composing it from multiple operations' respective models.

Operation's OSAL DAG code sections can be edited by pressing the *Open DAG* button, which opens the DAG editor window. Code section shows the currently selected DAG code from the list box below. A new DAG section can be created either by selecting *New DAG* list box item or pressing the *New* button. By pressing the *Undo* button, it is possible to revert changes to current code from the last saved position. DAG can be saved to operation's definition file by pressing the *Save* button. Unneccessary DAG sections can be deleted by pressing the *Delete* button.

If code section contains valid OSAL DAG code, then the editor window shows a DAG presentation of that code. In order to view the graph, a program called 'dot' must be installed. This is included in Graphviz graph visualization software package and can be obtained from www.graphviz.org.

Operation Modules Figure 4.4 shows the dialog for adding a new operation module to a search path. The name of the module can be entered into the text input field.

Operation modules may consist also of a behaviour source file. Before operation behaviour modules can be simulated, it is necessary to compile the source file. Figure 4.5 shows a result dialog of module compilation.

Data Memory Simulation Model The contents of the data memory simulation model used to simulate memory accessing operations can be viewed and edited. Figure 4.6 shows the memory window. Memory can be viewed as 1, 2, 4, or 8 MAUs. The format of the data is either in binary, hexadecimal, signed integer, unsigned integer, float, or double format. The contents of the memory can be changed by double clicking a memory cell.

Simulating Operation Behavior The behavior of operation can be simulated using the dialog in Figure 4.7. Input values can be edited by selecting a input from the input list and typing the new value in a

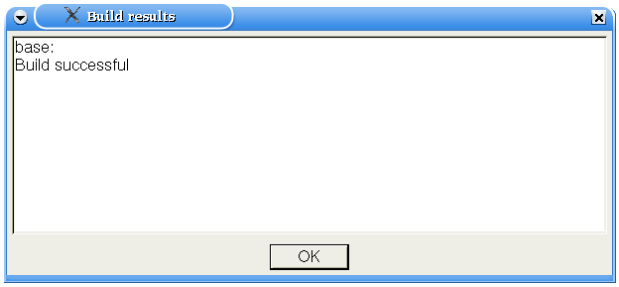

Figure 4.5: Result of module compilation

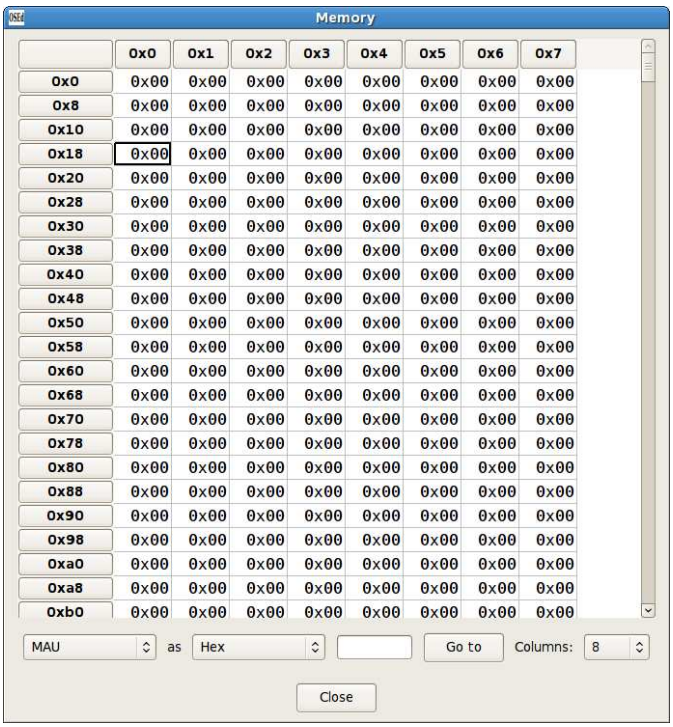

Figure 4.6: Memory window

text field below the input list. Change can be committed by pushing *Update* button. Trigger command and advance clock command are executed by pushing *Trigger* and *Advance clock* buttons. The format of the inputs and outputs can be modified by selecting a new format from the choice list above the *Trigger* button.

4.2.2 Operation Behavior Module Builder (buildopset)

The OSAL Builder is an external application that simplifies the process of compiling and installing new (user-defined) operations into the OSAL system.

The OSAL Builder is invoked with the following command line:

```
buildopset <options> operation_module
```

where *operation_module* is the name of the operation module. Operation module is the base name of a definition file, e.g., the module name of *base.opp* is 'base'. The *operation_module* can also be a full path, e.g., '/home/jack/.tce/opset/custom/mpeg'.

The behavior definition source file is searched in the directory of the *operation_module*. The directory of the *operation_module* is by default the current working directory. User may also enter the directory of the source file explicitly with switch '-s'. The suffix of the behavior definition source file is '.cc'. If no options

Figure 4.7: Operation Simulation

are given, the output file is a dynamic module and is stored in the directory of the *operation_module*.

The OSAL Builder accepts the following command line options:

Short Name	Long Name	Description	
k	`install`	*keyword*	Installs the data file and the built dynamic module into one of the allowed paths. The paths are identified by the following keywords: *base*, *custom*, *user*.
b	`ignore`	*boolean*	Ignores the case whereby the source file containing operation behavior model code are not found. By default, the OSAL Builder aborts if it cannot build the dynamic module. This option may be used in combination with *install* option to install XML data files before operation behavior definitions are available.
s	`source-dir`	*directory*	Enter explicit directory where the behavior definition source file to be used is found.

4.2.3 OSAL Tester (testosal)

The OSAL Tester is a small external application meant for debugging operation behavior models. The Tester lets user to specify lists of operations and constant input values and the outputs the corresponding sequence of result values.

The OSAL Tester can be run in interactive mode or in batch mode (like a script interpreter). The batch mode is especially useful to create input data sets of regression tests.

For all operations having state, a single operation state instance is constructed the first time the operation is simulated. This state instance is then used through the testing session by all operations that share the same operation state.

When started, the OSAL Tester enters interactive mode and prompts the user for operations. Any nonempty line entered by the user is interpreted as one operation, unless the character '!' is given at the beginning of the string. This character (which is not allowed in operation names) introduces *meta-commands* that are recognized by the interpreter and are not treated as potential operations in the architecture repertoire. For example, the OSAL Tester can be quit with the meta-command *!quit*.

Single operands of operations may only be constants and are separated by blanks.

For any line entered by the user, the Tester responds with a text line containing the results computed by the operation.

4.3 OSAL search paths

Default paths, where OSAL operations are seached, are the following:
(in descending search order)

1. *$PWD/data/*
   where $PWD is your current working directory
2. *TCE_SRC_ROOT/opset/base/*

4. Users local custom operations:
   $HOME/.tce/opset/custom/
   where $HOME is users home directory.

5. System-wide shared custom operations: *TCE_INSTALLATION_DIR/opset/base/*
   where TCE_INSTALLATION_DIR is the path where TCE accessories is installed (for example /usr/local/share/tce).

6. Operations in current working directory:
   $PWD/

NOTE! Search paths 1 and 2 are not used in Distributed versions!

Search paths are in descending search order meaning that operations are first searched from first defined paths. If an operation exists in multiple search paths, the last found version is used.

4.4 Processor Generator (ProGe)

Processor Generator (**ProGe**) produces a synthesizable hardware description of a TTA target processor specified by an architecture definition file (Section 2.2.1) and implementation definition file (Section 2.2.3).

Input: HDB, ADF, IDF

Output: VHDL implementation of the processor

There is a command line client for executing this functionality, but the functionality can also be used in the Processor Designer (Section 4.1). This section is a manual for the command line client.

Processor generation can be customized with plugin modules. The customizable parts are the generation of control unit and the interconnection network.

The CLI-based version of the processor generator is invoked by means of a command line with the following syntax:

generateprocessor <options> target

The sole, mandatory argument *target* gives the name of the file that contains the input data necessary to generate the target processor. The file can be either a processor configuration file or an architecture definition file. If the file specified is a PCF, then the names of ADF, BEM and IDF are defined in it.

The given PCF may be incomplete; its only mandatory entry is the reference to the ADF that defines the architecture of the target processor. If the file specified is an ADF, or if PCF does not contain BEM and IDF references, then a BEM is generated automatically and the default implementation of every building block that has multiple implementations is used.

The processor architecture must be synthesizable, otherwise an error message is given.

Short Name	Long Name	Description
b	`bem`	BEM file or the processor. If not given, ProGe will generate it.
l	`hdl`	Specifies the HDL of the top-level file which wires together the blocks taken from HDB with IC and GCU.[1]
i	`idf`	IDF file.
o	`output`	Name of the output directory. If not given, an output directory called 'proge-output' is created inside the current working directory.
u	`plugin-parameter`	Shows the parameters accepted by an IC/Decoder generator plug-in and a short description of the plug-in. When this options are given, any other type of option is ignored and ProGe returns immediately without generating any output file or directory. Even though other options are ignored they must be valid.

[1] In the initial version, the only keyword accepted is 'vhdl'.

4.4.0.1 IC/Decoder Generators

IC/Decoder generators are implemented as external plug-in modules. This enables users to provide customizable instruction decoder and IC implementations without recompiling or modifying the existing code base. One can easily add different plug-ins to experiment with different implementation alternatives, and as easily switch from one to another plug-in. To create a new plug-in, a software module that implements a certain interface must be created, compiled to a shared object file and copied to appropriate directory. Then it can be given as command line parameter to the generator.

4.5 Hardware Database Editor (HDB Editor)

HDB Editor (hdbeditor) is a graphical frontend for creating and modifying Hardware Databases i.e. HDB files (see Section 2.2.2 for details). By default, all the example HDB files are stored in the directory hdb/ of the TCE installation directory.

4.5.1 Usage

This section is intended to familiarize the reader to basic usage of the HDB Editor.

HDB editor can be launched from command line by entering:

```
hdbeditor
```

You can also give a .hdb-file as parameter for the hdbeditor:

```
hdbeditor customHardware.hdb
```

4.5.1.1 Creating a new HDB file

Choose "File" | "Create HDB...". From there, type a name for your .hdb file and save it in the default HDB path (tce/hdb).

After that, you can start adding new TTA components such as function units, register files, buses and sockets from "Edit" | "Add".

4.5.1.2 Adding new components

A new function unit's architecture can only be added through an existing ADF file unlike register files, which can only be added by hand. The ADF files can be done in the ProDe tool. After adding a new architecture, one can add an implementation for it by right-clicking on it and choosing "Add implementation"

The architecture implementation can be given either by hand or by a VHDL file.

After setting up the architecture, one can add new entries (function units, register files, buses, sockets) for the architectures.

4.5.1.3 Adding FU/RF HDL source files

HDL files of Function Unit and Register File implementations must be added in right compilation order i.e. the source file which needs to be compiled first is first in the list and so on.

4.6 Function Unit Interface

Function unit interfaces follow a certain de facto standard in TCE. Here is an example of a such interface:

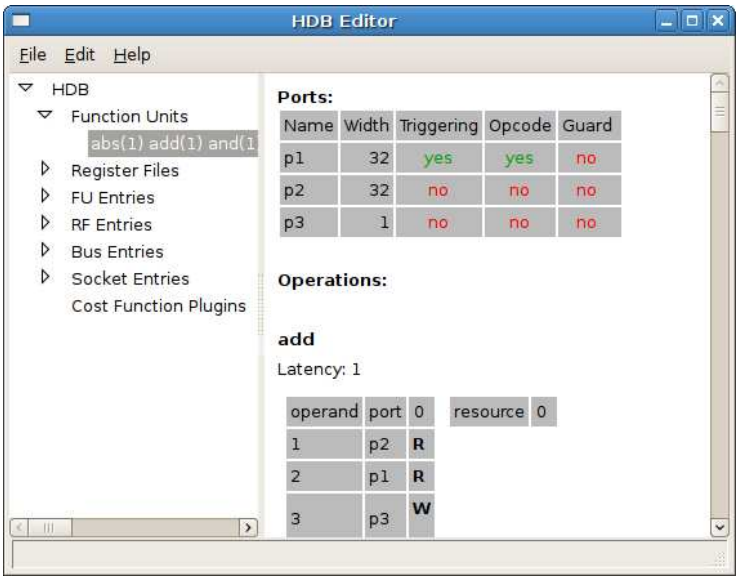

Figure 4.8: HDB Editor Main window.

```
entity fu_add_sub_eq_gt_always_1 is
  generic (
    dataw : integer := 32;
    busw  : integer := 32);
  port (
    -- trigger port / operand2 input port
    t1data   : in std_logic_vector (dataw-1 downto 0);
    t1opcode : in std_logic_vector (1 downto 0);
    t1load   : in std_logic;
    -- operand1 input port
    o1data : in std_logic_vector (dataw-1 downto 0);
    o1load : in std_logic;
    -- result output port
    r1data : out std_logic_vector (busw-1 downto 0);
    -- control signals
    glock  : in  std_logic;
    rstx   : in  std_logic;
    clk    : in  std_logic);
end fu_add_sub_eq_gt_always_1;
```

As you can see the actual implementation interface has more inputs/outputs than the architecture model of a FU. Architecture input ports are called o1data and t1data and the output port is called r1data. In addition there are load ports, operation code port and some control signals.

It is good practice to use (integer) generics for the port widths in the data I/O ports. This makes the managing of Hardware Database (HDB) easier because you can use the name of the generic to define port width formulas instead of using magic numbers. Whether the value of a generic is constant or parametrizable depends on how you define it in the HDB. In the example entity declaration we have used two different generics, data width (dataw) and bus width (busw). The input data ports depend on the dataw and the output port is defined using busw. You can also use only one generic but with 2 generics you can define different widths for input and output data if needed.

The names you set for the ports in VHDL code is up to you. When you define an FU implementation in HDB you'll have to do port mapping where you map the VHDL implementation names to the HDB counterparts. I strongly recommend to use a consistent naming convention to avoid confusion. HDBEditor prefills the common control signal dialog fields with the defacto standard names. If you use your own

naming style for the ports I still recommend to use the standard names for control signals which are used in the example above (glock = Global Lock, rstx = reset (x stands for active low), clk = clock).

The input ports have also load ports. The load signal is active high when the input value of an input port is valid and should be read (most probably to a register inside the FU). If the input port is also a trigger port the operation should be initiated.

If the FU has multiple operations there is also an operation code port in the interface. Opcode port is usually bound to the trigger port because operations have at least one input. And if you wish to initiate an operation with one input operand, the operand has to be written to the trigger port. Operation codes also have to be declared in HDB.

FUs can also have ports connecting to the external of the TTA processor core. They are declared like any other ports in the entity declaration but these ports must be defined as external in HDB.

See the TCE Tour-tutorial (section 3.1) for an example how to add an FU implementation to hdb.

4.6.1 Operation code order

If there are multiple operations in a Function Unit the operation codes should be numbered according to their alphabetical order. E.g. if there is an ALU with following operations: 'Add', 'Sub', 'And', 'Xor' and 'Not' the operation codes should be ordered as following:

Operation name	Operation code
Add	0
And	1
Not	2
Sub	3
Xor	4

HDBEditor and Processor Generator will issue a warning message if this guideline is not used in operation code numbering. Currently other numbering conventions can still be used but the support might be dropped in future versions of TCE. In order to ensure compatibility it is recommended that you use this new convention.

4.6.2 Summary of interface ports

4.6.2.1 Input/Output operand ports

`VHDL-type: std_logic_vector`
TCE naming convention is to use `t1data` for the triggering input port and `o1data`, `o2data` ... etc. for the rest of input operands. Output port is usually called `r1data` .

Use generics when you define widths for these ports.

4.6.2.2 Input load ports

`VHDL-type: std_logic`
The load ports are named after the input operand ports. For example load port of t1data is `t1load` and o1data is `o1load` .

4.6.2.3 Operation code port

`VHDL-type: std_logic_vector`
Operation code port is usually called `t1opcode` and it is bound to the trigger port.

4.6.2.4 Control signals

`VHDL-type: std_logic`
There are four control signals available in FUs:

`clk` is the most obvious, it is the clock signal

`rstx` is active low reset (x stands for active low)

`glock` is a Global Lock signal. For example if the global control unit (GCU) issues global lock the ongoing operation should freeze and wait for the global lock signal to become inactive before resuming execution.

`glock_r` is global lock request. Rarely used in normal function units. It can be used to request global lock state, for example, in case the result does not arrive in the time set by the static latency of the operation.

A common example using this signal is a load-store unit of which memory latency is sometimes larger than the static latency informed to the compiler due to dynamic behavior of the memory system. In that case, a global lock is requested after the static number of cycles has passed and the data has not arrived to the load-store-unit from the memory system.

4.6.3 Reserved keywords in generics

Currently there is only one reserved keyword in generic definitions and it is `addrw` . This keyword is used in load-store units to define trigger port's width according to the data address space width. Processor Generator identifies this keyword and defines the port width from the data address space assigned to the LSU in the ADF.

Chapter 5

CODE GENERATION TOOLS

5.1 TCE Compiler

TCE compiler compiles high level language (such as C/C++) source files provided by the toolset user and produces a bitcode program or a parallel TTA program. Bitcode program is a non-architecture specific sequential program. Parallel TTA program, however, is a architecture specific program that is optimized for the target architecture.

The main idea between these two program formats is that you can compile your source code into bitcode and then compile the bitcode into architecture dependent parallel code. This way you do not have to compile the source code again every time you make changes in the architecture. Instead you only need to compile the bitcode into parallel code.

The frontend compiler uses the LLVM C compiler which again is built on GCC version 4.

Input: program source file(s) in high-level language

Output: a fully linked TTA program

5.1.1 Usage of TCE compiler

The usage of the *tcecc* application is as follows:

```
tcecc <options> source-or-bc-file1 source-or-bc-file2 ...
```

The possible options of the application are the following:

Short Name	Long Name	Description
a	adf-file	Architectures for which the program is scheduled after the compilation. This switch can be used once for each target architecture. Note: there must be 'schedule' installed.
s	scheduler-config	Configure file for scheduling command.
O	optimization-level	Optimization level. 0=no optimizations, 1=preserve program API, 2=do not respect original API, 3 = same that 2
k	keep-symbols	List of symbols whose optimization away is prevented. If you are using this, remember to define at least the 'main' symbol.
o	output-name	File name of the output binary.
d	leave-dirty	Does not delete files from each compilation phase.
c	compile-only	Compiles only. Does not link or optimize.
v	verbose	Prints out commands and outputs for each phase.
h	help	Prints out help info about program usage and parameters.
D	preprocessor-define	Preprocessor definition to be passed to gcc.

I	include-directory	Include directory to be passed to gcc.
L	library-directory	Passed to gcc.
l	library-link	Passed to gcc.
W	warning	Ignored.
-	scheduler-binary	Scheduler binary to use instead of 'schedule' in path.
-	extra-llc-flags	Options passed to llc.
-	plugin-cache-dir	Directory for cached llvm target plugins.
-	no-plugin-cache	Do not cache generated llvm target plugins.
-	rebuild-plugin	Rebuild plugin in the cache
-	clear-plugin-cache	Clear plugin cache completely.

5.1.1.1 Examples of usage

Usage of tcecc quite alike to gcc, excluding that warning options are ignored.

If you wish to compile your source code into optimized bitcode the usage is:

```
tcecc -O2 -o myProg myProg.c
```

On the other hand if you already have an architecture definition file of the target processor you can compile the source code directly to parallel program:

```
tcecc -O2 -a myProcessor.adf -o myProg.tpef myProg.c
```

To compile the bitcode program into parallel program use:

```
tcecc -a myProcessor.adf -o myProg.tpef myProg.bc
```

Or if you want to a different scheduling configuration than the default:

```
tcecc -s /path/to/mySchedulerConfiguration.conf -a myProcessor.adf -o myProg.tpef
myProg.bc
```

Tcecc also has a "leave dirty" flag -d which preserves the intermediate files created by the compiler. After compilation is complete tcecc will tell you where to find these files (usually it is /tmp/tcecc-xxxxxx/). For example if you try to compile your C-code straight into a scheduled program and something goes wrong in scheduling you can find the bitcode program from the temp directory.

```
tcecc -d -O2 -a myProcessor.adf -o myProg.tpef myProg.c
```

After compilation you should see this kind of message:

Intermediate files left in build dir /tmp/tcecc-xxxxxx

where *xxxxxx* is a random pattern of characters.

If you only want to compile the source code without linking (and optimization) use -c flag. Output file is named after the source file with .o appendix if you do not define an output name with -o.

```
tcecc -c myProg.c
tcecc -c -o  /another/path/myProg.o myProg.c
```

With tcecc you can explicitly define symbols you wish to preserve in the binary. This can be useful in debugging and profiling if the compiler removes needed function labels. Symbols are given in a comma seperated list.

```
tcecc -O2 -a myMach.adf -k main,foo,bar -o myProg.tpef myProg.c
```

Plugins

5.1.2 Custom operations

Tcecc compiler automatically defines macros for operations found from operation definition files in OSAL search paths (see section 4.3 for more details). You can use these macros in C code by defining:

```
#include "tceops.h"
```

Macros use the following format:

```
_TCE_<name>(input1, ... , inputN, output1, ... , outputN);
```

where <name> is the operation name defined in OSAL. Number of input and output operands depends on the operation.

5.1.3 Known issues

1. Currently it is not possible to simulate a bitcode format program. But the advantage of bitcode simulation is quite non-existent because the bitcode does not even contain the final basic blocks that the architecture dependent program has.

5.2 Binary Encoding Map Generator (BEMGenerator)

Binary Encoding Map Generator (**BEMGenerator**) creates a file that describes how to encode TTA instructions for a given target processor into bit patterns that make up the executable bit image of the program (before compression, if used).

Input: ADF

Output: BEM

5.2.1 Usage

The usage of BEMgenerator is the following:

```
createbem -o outName.bem myProcessor.adf
```

5.3 Parallel Assembler and Disassembler

TCE Assembler compiles parallel TTA assembly programs to a TPEF binary. The **Disassembler** provides a textual disassembly of parallel TTA programs. Both tools can be executed only from the command line.

Assembler **Input**: program source file in the TTA parallel assembler language and an ADF

Output: parallel TPEF

Disassembler **Input**: parallel TPEF

Output: textual disassembly of the program

Rest of this section describes the textual appearance of TTA programs, that is, how a TTA program should be disassembled. The same textual format is accepted and assembled into a TTA program.

5.3.1 Usage of Disassembler

The usage of the *tcedisasm* application is as follows:

```
tcedisasm <options> adffile tpeffile
```

The *adffile* is the ADF file.
The *tpeffile* is the parallel TPEF file.

The possible options of the application are as follows:

Short Name	Long Name	Description
o	outputfile	The name of the output file.
h	help	Prints out help info about program usage and parameters.

The application disassembles given parallel TPEF file according to given ADF file. Program output is directed to standard output stream if specific output file is not specified. The output is TTA parallel assembler language.

The program output can then be used as an input for the assembler program tceasm.

The options can be given either using the short name or long name. If short name is used, a hyphen (-) prefix must be used. For example -o followed by the name of the output file. If the long name is used, a double hyphen (- -) prefix must be used, respectively.

5.3.1.1 An example of the usage

The following example generates a disassemble of a parallel TPEF in the file *add4_schedule.tpef* and writes the output to a file named *output_dis.asm*.

```
tcedisasm -o output_dis.asm add4_supported.adf add4_schedule.tpef
```

5.3.2 Usage of Assembler

The usage of the *tceasm* application is as follows:

```
tceasm <options> adffile assemblerfile
```

The *adffile* is the ADF file.
The *assemblerfile* is the program source file in TTA parallel assembler language.

The possible options of the application are as follows:

Short Name	Long Name	Description
o	outputfile	The name of the output file.
q	quiet	Do Not print warnings.
h	help	Help info about program usage and parameters.

The application creates a TPEF binary file from given assembler file. Program output is written to a file specified by outputfile parameter. If parameter is not given, the name of the output file will be the base name of the given assembler file concatenated with *.tpef*.

The options can be given either using the short name or long name. If short name is used, a hyphen (-) prefix must be used. For example -o followed by the name of the ouput file. If the long name is used, a double hyphen (- -) prefix must be used, respectively.

5.3.2.1 An example of the usage

The following example generates a TPEF binary file named *program.tpef*.

```
tceasm add4_schedule.adf program.asm
```

5.3.3 Memory Areas

A TTA assembly file consists of several memory areas. Each area specifies the contents (instructions or data) of part of an independently addressed memory (or address space). There are two kinds of memory areas: *code* areas and *data* areas. Code areas begin a section of the file that defines TTA instructions. Data areas begin a section that define groups of memory locations (each group is collectively termed "memory chunk" in this context) and reserve them to variables. By declaring data labels (see Section 5.3.7), variables can be referred to using a name instead of their address.

Memory areas are introduced by a header, which defines the type of area and its properties. The header is followed by several logical lines (described in Section 5.3.4), each declaring a TTA instruction or a memory chunk. The end of an area in the assembly file is not marked. Simply, a memory area terminates when the header of another memory area or the end of the assembly file is encountered.

The memory area header has one of the following formats:

```
\tt
  CODE [\parm{start}] ;\\
  DATA \parm{name} [\parm{start}] ;
```

A code area begins the declaration of TTA instructions that occupy a segment of the instruction memory. A data area begins the declaration of memory chunks reserved to data structures and variables.

A TTA program can work with several independently addressed data memories. The locations of different memories belong to different address spaces. The *name* parameter defines the address space a memory area belongs to. The code area declaration does not have a name parameter, because TTA programs support only one address space for instruction memory, so its name is redundant.

The *start* parameter defines the starting address of the area being declared within its address space. The start address can and usually is omitted. When omitted, the assembler will compute the start address and arrange different memory area declarations that refer to the same address space. The way the start address is computed is left to the assembler, which must follow only two rules:

1. If a memory area declaration for a given address space appears before another declaration for the same address space, it is assigned a lower address.
2. The start address of a memory area declaration is *at least* equal to the size of the previous area declared in the same address space plus its start address.

The second rule guarantees that the assembler reserves enough memory for an area to contain all the (chunk or instruction) declarations in it.

5.3.4 General Line Format

The body of memory areas consists of *logical lines*. Each line can span one or more physical lines of the text. Conversely, multiple logical lines can appear in a single physical lines. All logical lines are terminated by a semicolon ';'.

The format of logical lines is free. Any number of whitespace characters (tabs, blanks and newlines) can appear between any two tokens in a line. Whitespace is ignored and is only useful to improve readability. See Section 5.3.14 for suggestions about formatting style and use of whitespaces.

Comments start with a hash character ('#') and end at the end of the physical line. Comments are ignored by the syntax. A line that contains only a comment (and possibly whitespaces before the hash character) is completely removed before interpreting the program.

5.3.5 Allowed characters

Names (labels, procedures, function units etc.) used in assembly code must obey the following format:

```
[a-zA-Z_][a-zA-z0-9_]*
```

Basically this means is that a name must begin with a letter from range a-z or A-Z or with an underscore. After the first character numbers can also be used.

Upper case and lower case letters are treated as different characters. For example labels **main:** and **Main:** are both unique.

5.3.6 Literals

Literals are expressions that represent constant values. There are two classes of literals: numeric literals and strings.

Numeric literals. A numeric literal is a numeral in a positional system. The base of the system (or radix) can be decimal, hexadecimal or binary. Hexadecimal numbers are prefixed with '0x', binary numbers are prefixed with '0b'. Numbers in base 10 do not have a prefix. Floating-point numbers can only have decimal base.

Example: Numeric literals.

```
0x56F05A
7116083
0b11011001001010100110011
17.759
308e+55
```

The first three literals are interpreted as integer numbers expressed in base, respectively, 16, 10 and 2. An all-digit literal string starting with '0' digit is interpreted as a decimal number, not as an octal number, as is customary in many high level languages.[1] The last two literals are interpreted as floating point numbers. Unlike integer literals, floating-point literals can appear only in initialisation sequences of data declarations (see Section 5.3.8 for details).

EXTENSION: charset directive

String literals. A string literal consists of a string of characters. The the numeric values stored in the memory chunk initialised by a string literal depend on the character encoding of the host machine. The use of string literals makes the assembly program less portable.

Literals are defined as sequences of characters enclosed in double (") or single (') quotes. A literal can be split into multiple quoted strings of characters. All strings are concatenated to form a single sequence of characters.

Double quotes can be used to escape single quotes and vice versa. To escape a string that contains both, the declaration must be split into multiple strings.

Example: String literals. The following literals all declare the same string `Can't open file "%1"` .

```
"Can't open file" '"%1"'
'Can' "'" 't open file "%1"'
"Can't open" ' file "%1"'
```

String literals can appear only in initialisation sequences of data declarations (see Section 5.3.8 for details).

[1] This notation for octal literals has been deprecated.

Size, encoding and layout of string literals. By default, the size (number of MAU's) of the value defined by a string literal is equal to the number of characters. If one MAU is wider than the character encoding, then the value stored in the MAU is padded with zeroes. The position of the padding bits depends on the byte-order of the target architecture: most significant if "big endian", least significant if "little endian".

If one character does not fit in a singe MAU, then each character is encoded in $\lceil m/n \rceil$ MAU's, where n is the MAU's bit width and m is the number of bits taken by a character.

When necessary (for example, to avoid wasting bits), it is possible to specify how many characters are packed in one MAU or, vice versa, how many MAU's are taken to encode one character. The size specifier for characters is prefixed to a quoted string and consists of a number followed by a semicolon.

If $n > m$, the prefixed number specifies the number of characters packed in a single MAU. For example, if one MAU is 32 bits long and a character takes 8 bits, then the size specifier in

```
4:"My string"
```

means: pack 4 characters in one MAU. The size specifier cannot be greater than $\lceil n/m \rceil$. The size '1' is equivalent to the default.

If $m > n$, the prefixed number specifies the number of adjacent MAU's used to encode one character. For example, if MAU's are 8-bit long and one character takes 16 bits, then the same size specifier means: reserve 4 MAU's to encode a single character. In this case, a 16-bit character is encoded in 32 bits, and padding occurs as described above. The size of the specifier in this case cannot be smaller than $\lceil n/m \rceil$, which is the default value when the size is not specified explicitly.

5.3.7 Labels

A label is a name that can be used in lieu of a memory address. Labels "decorate" data or instruction addresses and can be used to refer to, respectively, the address of a data structure or an instruction. The address space of a label does not need to be specified explicitly, because it is implied by the memory area declaration block the label belongs to.

A label declaration consists of a name string followed by a colon:

```
\tt
  \parm{label-name}:
```

Only a restricted set of characters can appear in label names. See Section 5.3.5 for details.

A label must always appear at the beginning of a logical line and must be followed by a normal line declaration (see Sections 5.3.8, 5.3.9 for details). Only whitespace or another label can precede a label. Label declarations always refer to the address of the following memory location, which is the start location of the element (data chunk or a TTA instruction) specified by the line.

Labels can be used instead of the address literal they represent in data definitions and instruction definitions. They are referred to simply by their name (without the colon), as in the following examples:

```
# label reference inside a code area (as immediate)
aLabel -> r5 ;

# label reference inside a data area (as initialisation value)
DA 4 aLabel ;
```

5.3.8 Data Line

A data line consists of a directive that reserves a chunk of memory (expressed as an integer number of minimum addressable units) for a data structure used by the TTA program:

```
\tt
  DA \parm{size} [\parm{init-chunk-1} \parm{init-chunk-2} \ldots] ;
```

The keyword 'DA' (Data Area) introduces the declaration of a memory chunk. The parameter *size* gives the size of the memory chunk in MAU's of the address space of the memory area.

Memory chunks, by default, are initialised with zeroes. The memory chunk can also be initialised explicitly. In this case, *size* is followed by a number of literals (described in Section 5.3.6) or labels (Section 5.3.7) that represent initialisation values. An initialisation value represents a constant integer number and takes always an integer number of MAU's.

Size of the initialisation values. The size of an initialisation value can be given by prepending the size (in MAU's) followed by a semicolon to the initialisation value. If not defined explicitly, the size of the initialisation values is computed by means of a number of rules. If the declaration contains only one initialisation value, then the numeric value is extended to *size*, otherwise, the rules are more complex and depend on the type of initialisation value.

1. If the initialisation value is a numeric literal expressed in base 10, then it is extended to *size* MAU's.
2. If the initialisation value is a numeric literal expressed in base 2 or 16, then its size is extended to the minimum number of MAU's necessary to represents all its digits, even if the most significant digits are zeroes.
3. If the initialisation value is a label, then it is extended to *size* MAU's.

Extension sign. Decimal literals are sign-extended. Binary, hexadecimal and string literal values are zero-extended. Also the initialisation values represented by labels are always zero-extended.

Partial Initialisation. If the combined size of the initialisation values (computed or specified explicitly, it does not matter) is smaller than the size declared by the 'DA' directive, then the remaining MAU's are initialised with zeroes.

Example: Padding of single initialisation elements. Given an 8-bit MAU, the following declarations:

```
DA 2 0xBB ; # equivalent to 2:0xBB
DA 2 0b110001 ; # 0x31 (padded with 2 zero bits)
DA 2 -13 ;
```

define 2-MAU initialisation values: 0x00BB, 0x0031, and 0xFFF3, respectively.

Example: Padding of of multi-element initialisation lists. The following declarations:

```
DA 4 0x00A8 0x11;
DA 4 0b0000000010100100 0x11 ;
```

are equivalent and force the size of the first initialisation value in each list to 16 bits (2 MAU's) even if the integer expressed by the declarations take less bits. The 4-MAU memory chunk is initialised, in both declarations, with the number 0x00A81100. Another way to force the number of MAU's taken by each initialisation value is to specify it explicitly. The following declarations are equivalent to the declarations above:

```
DA 4 2:0xA8  0x11;
DA 4 2:0b10100100  0x11;
```

Finally, the following declarations:

```
DA 2 1:0xA8  0x11;
DA 2 1:0b10100100  0x11;
```

define a memory chunk initialised with 0xA8110000. The initialisation value (in case of the binary literal, after padding to MAU bit width) defines only the first MAU.

When labels appear in initialisation sequences consisting of multiple elements, the size of the label address stored must be specified explicitly.

Example. Initialisation with Labels. The following declaration initialises a 6-MAU data structure where the first 2 MAU's contain characters 'A' and 'C', respectively, and the following 4 MAU's contain two addresses. The addresses, in this target architecture, take 2 MAU's.

```
DA 6 0x41 0x43 2:nextPointer 2:prevPointer ;
```

5.3.9 Code Line

A code line defines a TTA instruction and consists of a comma-separated, fixed sequence of bus slots. A bus slot in any given cycle can either program a data transport or encode part of a long immediate and program the action of writing it to a destination (immediate) register for later use.[2]

A special case of code line that defines an empty TTA instruction. This line contains only three dots separated by one or more white spaces:

```
. . . ; # completely empty TTA instruction
```

A special case of move slot is the empty move slot. An empty move slot does not program any data transport nor encodes bits of a long immediate. A special token, consisting of three dots represents an empty move slot. Thus, for a three-bus TTA processor, the following code line represents an empty instruction:

```
... , ... , ... ; # completely empty TTA instruction
```

5.3.10 Long Immediate Chunk

When a move slot encodes part of a long immediate, its declaration is surrounded by square brackets and has the following format:

```
\tt
  \parm{destination}=\parm{value}
```

where *destination* is a valid register specifier and *value* is a literal or a label that gives the value of the immediate. The only valid register specifiers are those that represent a register that belongs to an immediate unit. See section 5.3.12 for details on register specifiers.

When the bits of a long immediate occupy more than one move slot, the format of the immediate declaration is slightly more complex. In this case, the value of the immediate (whether literal or label) is declared in one and only one of the slots (it does not matter which one). The other slots contain only the destination register specifier.

5.3.11 Data Transport

A data transport consists of one optional part (a guard expression) and two mandatory parts (a source and a destination). All three can contain an port or register specifier, described in Section 5.3.12.

The guard expression consists of a single-character that represents the invert flag followed by a source register specifier. The invert flag is expressed as follows:

1. Single-character token '!': the result of the guard expression evaluates to zero if the source value is nonzero, and evaluates to one if the source value is equal to zero.

[2]The action of actually writing the long immediate to a destination register is encoded in a dedicated instruction field, and is not repeated in each move slot that encodes part of the long immediate. This detail is irrelevant from the point of view of program specification. Although the syntax is slightly redundant, because it repeats the destination register in every slot that encodes a piece of a long immediate, it is chosen because it is simple and avoids any chance of ambiguity.

2. Single-character token '?': the result of the guard expression evaluates to zero if the source value is zero, and evaluates to one if the source value is not zero.

The move source specifier can be either a register and port specifier or an in-line immediate. Register and port specifiers can be GPR's, FU output ports, long immediate registers, bridge registers. The format of all these is specified in Section 5.3.12. The in-line immediate represents an integer constant and can be defined as a literal or as a label. In the latter case, the in-line immediate can be followed by an equal sign and a literal corresponding to the value of the label. The value of the labels is more likely to be shown as a result of disassembling an existing program than in user input code, since users can demand data allocation and address resolution to the assembler.

Example: Label with value. The following move copies the label 'LAB', which represents the address 0x051F0, to a GPR:

```
LAB=0x051F0 -> r.4
```

The move destination consists of a register and port specifier of two types: either GPR's or FU input ports.

5.3.12 Register Port Specifier

Any register or port of a TTA processor that can appear as a move or guard source, or as a move destination is identified and referred to by means of a string. There are different types of register port specifiers:

1. General-purpose register.
2. Function unit port.
3. Immediate register.
4. Bridge register.

GPR's are specified with a string of the following format:

```
\tt
  \parm{reg-file}[.\parm{port}].\parm{index}
```

DISCUSS: pending **??**

where *reg-file* is the name of the register file, *port*, which can be omitted, is the name of the port through which the register is accessed, and *index* is the address of the register within its register file.

Function unit input and output ports are specified with a string of the following format:

```
\tt
  \parm{function-unit}.\parm{port}.[\parm{operation}]
```

where *function-unit* is the name of the function unit, *port* is the name of the port through which the register is accessed, and *operation*, which is required only for opcode-setting ports, identifies the operation performed as a side effect of the transport. It is not an error to specify *operation* also for ports that do not set the opcode. Although it does not represent any real information encoded in the TTA program, this could improve the readability of the program.

Immediate registers are specified with a string if the following format:

```
\tt
  \parm{imm-unit}[.\parm{port}].\parm{index}
```

DISCUSS: pending **??**

where *imm-unit* is the name of the immediate unit, *port*, which can be omitted, is the name of the port through which the register is accessed, and *index* is the address of the register within its unit.

Since any bus can be connected to at most two busses through bridges, it is not necessary to specify bridge registers explicitly. Instead, the string that identifies a bridge register can only take one of two values:

'{prev}' or '{next}'. These strings identify the bus whose value in previous cycle is stored in the register itself. A bus is identified by '{prev}' if it is programmed by a bus slot that precedes the bus slot that reads the bridge register. Conversely, if the bus is identified by '{next}', then it is programmed by a bus slot that follows the bus slots that reads the bridge register. In either case, the source bus slot must be adjacent to the bus slot that contains the moves that reads the bridge register.

Example: possible register and port specifiers.

```
\begin{tabular}{lp{0.75\textwidth}}
\texttt{IA.0}     & immediate unit 'IA', register with index 0\\
\texttt{RFA.5}    & register file 'RFA', register with index 5\\
\texttt{U.s.add} & port 's' of function unit 'U', opcode for operation
'add'\\
\verb|{|\texttt{prev}\verb|}|  & bridge register that contains the value on
the
                     bus programmed by the previous bus slot in previous
cycle\\
\end{tabular}
```

Alternative syntax of function unit sources and destinations. Most clients, especially user interfaces, may find direct references to function unit ports inconvenient. For this reason, an alternative syntax is supported for input and output ports of function units:

```
\tt
  \parm{function-unit}.\parm{operation}.\parm{index}
```

where *function-unit* is the name of the function unit, *operation* identifies the operation performed as a side effect of the transport and *index* is a number in the range [1,*n*], where *n* is the total number of inputs and outputs of the operation. The operation input and output, indirectly, identifies also the FU input or output and the port accessed. Contrary to the base syntax, which is requires the operation name only for opcode-setting ports, this alternative syntax makes the operation name not optional. The main advantage of this syntax is that is makes the code easier to read, because it removes the need to know what is the operation input or output bound to a port, because. The main drawback is an amount of (harmless) "fuzziness" and inconsistency, because it forces the user to define an operation for ports that do not set the opcode, even in cases where the operand is shared between two different operations. For example, suppose that the operand '1' of operations 'add' and 'mul' is bound to a port that does not set the opcode and its value is shared between an 'add' and a 'mul':

```
r1 -> U1.add.1, r2 -> U1.add.2;
U1.add.3 -> r3, r4 -> U1.mul.2;
U1.mul.3 -> r5
```

it looks as if the shared move belonged only to 'add'. One could have also written, correctly but less clearly:

```
r1 -> U1.mul.1, r2 -> U1.add.2;
# same code follows
```

or even, assuming that operation 'sub' is also supported by the same unit and its operand '1' is bound to the same port:

```
r1 -> U1.sub.1, r2 -> U1.add.2;
# same code follows
```

This alternative syntax is the only one permitted for TTA moves where operations are not assigned to a function unit of the target machine.

When operations are not assigned to a function unit of the target machine, they are formally assigned to the Universal Function Unit of the Universal Machine. See Section **??** for details on the Universal Machine and other conventions that apply to unscheduled TTA code. The name of the unit, in this case, may be omitted from the string.

how to distinguish a RF name from an operation name then?

5.3.13 Assembler Command Directives

Command directives do not specify any code or data, but change the way the assembler treats (part of) the code or data declared in the assembly program. A command directive is introduced by a colon followed by the name string that identifies it, and must appear at the beginning of a new logical line (possibly with whitespace before).

The assembler recognises the following directives.

procedure The ':procedure ' directive defines the starting point of a new procedure. This directive is followed by one mandatory parameter: the name of the procedure. Procedure directives should appear only in code areas. The procedure directive defines also, implicitly, the end of procedure declared by the previous ':procedure' directive. If the first code section of the assembly program contains any code before a procedure directive, the code is assumed to be part of a nameless procedure. Code in following code areas that precede any procedure directive is considered part of the last procedure declared in one of the previous code areas.

Example: declaration of a procedure.

```
CODE ;
:procedure Foo ;
Foo:
    r5 -> r6 , ... ;
    . . . ;
    ... , r7 -> add.1 ;
```

In this example, a procedure called 'Foo' is declared and a code label with the same name is declared at the procedure start point. The code label could be given any name, or could be placed elsewhere in the same procedure. In this case, the code label 'Foo' marks the first instruction of procedure 'Foo'.

global The ':global directive declares that a given label is globally visible, that is, it could be linked and resolved with external code. This directive is followed by one mandatory parameter: the name of the label. The label must be defined in the same assembly file. The label may belong to the data or the code section, indifferently.

extern The ':extern directive declares that a given label is globally visible and must be resolved an external definition. This directive is followed by one mandatory parameter: the name of the label. The label must not be defined in the assembly file.

There can be only one label with any given name that is declared global or external.

Example: declaration of undefined and defined global labels.

```
DATA dmem  0x540;
aVar:
    DA 4 ;
:global aVar ;
:extern budVar ;
```

In this example, 'aVar' is declared to have global linkage scope (that is, it may be used to resolve references from other object files, once the assembly is assembled). Also 'budVar' is declared to have global linkage, but in this case the program does not define a data or code label with that name anywhere, and the symbol must be resolved with a definition in an external file.

5.3.14 Assembly Format Style

This section describes a number of nonbinding guidelines that add to the assembly syntax specification and are meant to improve programs' readability.

Whitespaces. Although the format of the assembly is completely free-form, tabs, whitespaces and new lines can be used to improve the assembly layout and the readability. The following rules are suggested:

1. Separate the following tokens with at least one whitespace character:
   (a) Label declaration '*name:*' and first move or 'DA' directive.
   (b) Moves of an instruction and commas.
   (c) Move source and destination and the '->' or '<-' token.
2. Do not separate the following tokens with whitespaces:
   (a) Long immediate chunk declaration and the surrounding brackets.
   (b) Label and, literal and the '=' token in between.
   (c) Any part of a register specifier (unit, port, operation, index) and the '.' separator token.
   (d) Register specifier, label or literal and the '=' in between.
   (e) Invert flag mark ('!' or '?') and the register specifier of a guard expression.
   (f) Initialisation chunk, the number of MAU's it takes and the ':' token in between.
   (g) Colon ':' and the nearby label or directive name.

End of Line. The length of physical lines accepted is only limited by client implementation. Lines up to 1024 characters must be supported by any implementation that complies with these specifications. However, it is a good rule, to improve readability, that physical line should not exceed the usual line length of 80 or 120 characters. If a TTA instruction or a data declaration does not fit in the standard length, the logical line should be split into multiple physical lines. The physical lines following the first piece of a logical line can be indented at the same column or more to the right. In case of data declarations, the line is split between two literals that form an initialisation data sequence. In case of TTA instructions, logical lines should never be split after a token of type 'X' if it is recommended that no whitespace should follow 'X' tokens. To improve readability, TTA instructions should be split only past the comma that separates two move slots:

```
# good line breaking
r2 -> U.sub.1 , [i1=var] , r3 -> U.sub.2 , i1 -> L.ld.1 , [i1] ,
0 -> U.eq.2 ;

# bad line breaking
r2 -> U.sub.1 , [i1=var] , r3 -> U.sub.2 , i1 -> L.ld.1 , [i1] , 0 ->
U.eq.2 ;

# really bad line breaking
r2 -> U.sub.1 , [i1=var] , r3 -> U.sub.2 , i1 -> L.ld.1 , [i1] , 0 -> U.
eq.2 ;
```

Tabulation. The following rules can be taken as starting point for a rather orderly layout of the assembly text, which resembles the layout of traditional assembly languages:

1. The first n characters of the assembly lines are reserved to labels. Instruction or data declarations are always indented by n characters.
2. Labels appear in the same physical line of the instruction or data declaration they refer to. Labels are no more than $n-2$ characters long.

This layout is particularly clean when the TTA instructions contain few bus slots and when multiple labels for the same data chunk or instruction do not occur.

Example: Assembly layout style best suited target architectures with few busses.

```
DATA DMEM
var:       DA 4;

CODE
lab_A:     spr -> U.add.1 , 55 -> U.add.2 , spr -> r12 ;
           [i0=0x7F] , U.add.3 -> spr , i0 -> r2 ;
loop_1:    r2 -> U.sub.1 , r3 -> U.sub.2 , var -> L.ld.1 ;
           r2 -> U.eq.1 , U.sub.3 -> r2 , 0 -> U.eq.2 ;
           ?U.eq.3 loop_1 -> C.jump.1 , L.ld.2 -> U.and.2 ;
           0x1F -> U.and.1 , ... , ... ;
           ... , U.and.3 -> r8 , ... ;
```

An alternative layout of the assembly text is the following:

1. Instruction and data declarations are always indented by *n* characters.
2. Each label declaration appears in a separate physical line of the instruction or data declaration they refer to, and starts from column 0.

This layout could be preferable when the TTA instructions contain so many bus slots that the logical line is usually split into multiple physical lines, because it separates more clearly the code before and after a label (which usually marks also a basic block entry point). In addition, this layout looks better when an instruction or data declaration has multiple labels and when the label name is long.

Example: Assembly layout style best suited targets with many busses.

```
DATA DMEM
var:
    DA 4;

CODE
a_long_label_name:
    spr -> U.add.1 , 55 -> U.add.2 , spr -> r12 , [i0=0x7F], i0 -> r2,
    ... ;
    ... ,  U.add.3 -> spr , ... , ... , ... , ... ;
loop_1:
    r2 -> U.sub.1 , [i1=var] , r3 -> U.sub.2 , i1 -> L.ld.1 , [i1] ,
    0 -> U.eq.2 ;
    r2 -> U.eq.1 , U.sub.3 -> r2 , ... , ?U.eq.3 loop_1 -> C.jump.1 ,
    0x1F -> U.and.1 , ... ;
    L.ld.2 -> U.and.2 , ... , ... , ... , ... , ... ;
    ... , ... , ... , U.and.3 -> r8 , ... , ... ;
```

This example of assembly code is exactly equivalent to the code of previous example, except that the address of ‘var’ data chunk (a 4-MAU word) is encoded in a long immediate and takes 2 move slots.

Layout of Memory Area Declarations. It is preferable to subdivide the contents of memories into several memory area declarations and to group near each other area declarations of different address spaces that are related to each other. This underlines the relation between data and code. The alternative, a single area for each address space, mixes together all data and all procedures of a program.

Mixing Alternative Syntaxes. It is preferable to not mix alternative styles or even syntaxes, although any client that works with the assembly language is expected to deal with syntax variants.

5.3.15 Error Conditions

This section describes all the possible logical errors that can occur while assembling a TTA program.

Address Width Overflow in Initialisation. A label is used as initialisation value of a data declaration, and the label address computed by the assembler exceeds the number of MAU's in the data declaration that must be initialised.

Address Width Overflow in Long Immediate. A label is used as initialisation value of a long immediate declaration, and the label address computed by the assembler exceeds total width of the move slots that encode the immediate, when concatenated together.

Address Width Overflow in In-line Immediate. A label is used as initialisation value of an in-line immediate, and the label address computed by the assembler exceeds width of source field that encodes the immediate.

Unspecified Long Immediate Value. A long immediate is defined, but none of the move slots that encode its bits defines the immediate value.

Multiply Defined Long Immediate Value. More than one of the move slots that contain the bits of a long immediate defines the immediate value.[3]

Overlapping Memory Areas. The start address specified in the header of two memory area declarations is such that, once computed the sizes of each memory area, there is an overlapping.

Multiple Global Symbols with Same Name. A ':global' directive declares a symbol with given name as globally visible, but multiple labels with given name are declared in the program.

Unknown Command Directive. A command directive has been found that is not one of the directives supported by the assembler.

Misplaced Procedure Directive. A ':procedure' directive appears inside a data area declaration block.

Procedure Directive Past End of Code. A ':procedure' directive appears after the last code line in the program.

Label Past End of Area. A label has been declared immediately before an area header or at the end of the assembly program. Labels must be followed by a code line or a data line.

Character Size Specifier too Big. A size specified for the characters of a string literal is greater than the maximum number of characters that can fit in one MAU.

Character Size Specifier too Small. A size specified for the characters of a string literal is smaller than the minimum number of MAU's necessary to encode one character.

Illegal Characters in Quoted String. A quoted string cannot contain non-printable characters (that is, characters that cannot be printed in the host encoding) and end-of-line characters.

5.3.16 Warning Conditions

This section describes all conditions of target architecture or assembly syntax for which the client should issue an optional warning to prepare users for potential errors or problematic conditions.

[3] If the values are identical in every move slot, then the client could issue a warning rather than a critical error.

Equally Named Register File and Function Unit. A register file and a function unit of the target architecture have the same name. This is one of the conditions for the activation of the disambiguation rule.

Port with the Name of an Operation. A register file or a function unit port are identified by a string name that is also a name of a valid operation supported by the target architecture. The first condition (port of register file) is more serious, because it may require triggering a disambiguation rule. The second condition (FU port) is not ambiguous, but is confusing and ugly. The second condition may be more or less severe depending, respectively, whether the operation with the same name is supported by the same FU or by another FU.

Code without Procedure. The first code area of the program begins with code lines before the first ':procedure' directive. A nameless procedure with local visibility is automatically created by the assembler.

Procedure Spanning Multiple Code Areas. A code area contains code line before the first ':procedure' directive, but it is not the first code area declared in the code. The code at the beginning of the area is attached to the procedure declared by the last ':procedure' directive.

Empty Quoted String. Empty quoted strings are ignored.

5.3.17 Disambiguation Rules

Certain syntactic structures may be assigned different and (in principle) equally valid interpretations. In these cases, a disambiguation rule assigns priority to one of the two interpretation. Grammatically ambiguous assembly code should be avoided. Clients that operate on TTA assembly syntax should issue a warning whenever a disambiguation rule is employed.

Disambiguation of GPR and FU terms. When a GPR term includes also the RF port specifier, it can be interpreted also as a function unit input or output.

Normally, the names of units, ports and operation rule out one of the two interpretations. Ambiguity can only occur only if:

1. The target architecture contains a RF and a FU with identical name.
2. One of the RF ports has a name identical to one of the operations supported by the FU that has the same name of the RF.

PENDING: disambiguation of unscheduled TTA code **??**

Ambiguity is resolved in favour of the GPR interpretation. No condition applies to the indices (register index or operation input or output index). The first interpretation is chosen even when it results in a semantic error (an index out of range) whereas the other interpretation would be valid.

Example. Disambiguation rule. The following move is interpreted as a move that writes the constant 55 to the register register with index 2 of register file 'xx' through port 'yy'. If there exists an FU called 'xx' that supports an operation 'yy' which has an input with index 2, this interpretation of the move is never possible.

```
55 -> xx.yy.2
```

Even if the disambiguation rule is not triggered, clients should warn when the target architecture satisfies one of the conditions above (or a similar condition). See Section 5.3.16 for a description of this and other conditions for which a warning should be issued.

Disambiguation of variables and operation terms. In unscheduled code, operation terms cannot be confused with variables. The special RF names 'r', 'f' and 'b' are reserved, respectively, to integer, floating-point and Boolean register files of the universal machine. The assembler does not allow any operation to have one of these names.

PENDING: disambiguation of mixed TTA code **??**

Example. Unambiguous move term accessing a variable. The following move is interpreted as "copy constant 55 to variable with index 2 of variable pool 'r"'. There cannot exist an operation 'r', so the interpretation of the move destination as operation term is impossible.

```
55 -> r.2
```

Disambiguation of Register File and Immediate Unit names Assembler syntax does not differentiate unit names of immediate units from unit names of register files. The same register specifier of a move source

```
x.2 -> alu.add.1
```

can represents a GPR or an immediate register depending on whether 'x' is an RF or a IU.

In this case the GPR interpretation is always preferred over the IU interpretation. However using the same naming for IUs and GPRs restricts severely the programmability of target machine and is not encouraged.

5.4 Program Image Generator (PIG)

Program Image Generator (**PIG**) generates the bit image which can be uploaded to the target machine's memory for execution. Compression can be applied to the instruction memory image by means of instruction compression algorithm plugins.

Input: TPEF, BEM, ADF

Output: program bit image in alternative formats

5.4.1 Usage

The usage of the *generatebits* application is as follows:

```
generatebits <options> ADF
```

The possible options of the application are as follows:

Short Name	Long Name	Description
b	bem	The binary encoding map.
c	compressor	Name of the code compressor plugin file.
u	compressorparam	Parameter to the code compressor in form 'name=value'.
d	dataimages	Creates data images.
g	decompressor	Generates a decompressor block.
o	diformat	The output format of data image(s) ('ascii', 'array' or 'binary'). Default is 'ascii'.
w	dmemwidthinmaus	Width of data memory in MAUs. Default is 1.
x	hdl-dir	Directory root where are the ProGe generated HDL files generated by ProGe. If given, PIG will write imem_mau_pkg and compressor in that directory. Otherwise they are written to cwd.
f	piformat	Determines the output format of the program image and data images. Value may be 'ascii' or 'binary'.
s	showcompressors	Shows the compressor plugin descriptions.
p	program	The TPEF program file(s).

The application prints the program image to the standard output stream. It can be easily forwarded to a file, if wanted. The data images are generated to separate files, one for each address space. Names of the files are determined by the name of the address space and the suffix is *.img*. The files are generated to the directory in which the application is executed.

The binary encoding map input parameter may be omitted. If so, the BEM to be used is generated automatically. The BEM is used in the intermediate format of the program image, before it is compressed by the code compressor plugin. However, if no code compression is applied, the program image output matches the format defined in the BEM.

The options can be given either using the short name or long name. If short name is used, a hyphen (-) prefix must be used. For example -a followed by the name of the ADF file. If the long name is used, a double hyphen (- -) prefix must be used, respectively.

5.4.1.1 An example of the usage

The following example generates a program image and data images of the address spaces in ASCII format without code compression.

```
generatebits -b encodings.bem -t program.tpef -f ascii -d machine.adf
```

5.4.2 Dictionary Compressor

5.4.2.1 Defining New Code Compressors

By default, PIG does not apply any code compression to the program image. However, user can create a dynamic code compressor module which is loaded and used by PIG. To define a code compressor, a C++ class derived from *CodeCompressorPlugin* class must be created and compiled to a shared object file. The class is exported as a plugin using a macro defined in *CodeCompressor.hh* file. The *buildcompressor* script can be used to compile the plugin module.

5.4.2.2 Creating the Code Compressor Module

As mentioned, the code compressor class must be derived from *CodeCompressorPlugin* base class. The code compressor class must implement the virtual compress() method of the *CodeCompressorPlugin* class. An example of a simple dictionary compressor is defined in *compressors/simple_dictionary.cc* file in the source code distribution.

Compress Method The compress method is the heart of the compressor. The compressor method returns the complete program image as a bit vector. The task of the method is to call the *addInstruction* method of the base class sequentially to add the instructions to the program image. The base class does the rest of the job. You just have to provide the bits of each instruction in the parameter of addInstruction calls. Finally, when all the instructions are added, the program image can be returned by calling the *programBits* method of the base class.

The instruction bits are provided as *InstructionBitVector* instances. That class provides capability to mark which bits of the instruction refer to address of another instruction and thus are likely to be changed when the real address of the referred instruction is known. It is the responsibility of code compressor to mark that kind of bits to the instruction bit vectors given in *addInstruction* calls. The base class will change the bits when the referred instruction is added (when its address is known).

The code compressor should tell the base class which instructions must be placed in the beginning of a MAU block. For example, the jump targets should be in the beginning of MAU. Otherwise instruction fetching will get difficult. This can be done by calling the *setInstructionToStartAtBeginningOfMAU* method of the base class. If all the instructions are meant to start at the beginning of MAU, *setAllInstructionsToStartAtBeginningOfMAU* method is handy. By default, all the instructions are concatenated without any pad bits in between them, whatever the MAU of the instruction memory is. Note that *setInstructionToStartAtBeginningOfMAU* / *setAllInstructionsToStartAtBeginningOfMAU* method(s) must be called before starting to add instructions with *addInstruction* method.

Helper Methods Provided by the Base Class The *CodeCompressorPlugin* base class provides some handy methods that might be useful for the code compressor implementation. The following lists the most important ones:

- *program()*: Returns the POM of the program.
- *tpefProgram()*: Returns the TPEF of the program.
- *binaryEncoding()*: Returns the BEM used to encode the instructions.
- *machine()*: Returns the machine.
- *bemBits()*: Returns the program image encoded with the rules of BEM.
- *bemInstructionBits()*: Returns the bits of the given instruction encoded with the rules of BEM.

5.4.2.3 Building the Shared Object

When the code compressor module is coded, it must be compiled to a shared object. It can be done with the *buildcompressor* script. The script takes the name of the object file to be created and the source file(s) as command line parameters. The output of the script, assuming that everything goes fine, is the dynamic module that can be given to the *generatebits* application.

5.5 TPEF Dumper (dumptpef)

TPEF Dumper is a program used for displaying information from the given TPEF.

Input: TPEF

Output: dumped data from TPEF (printed to the standard output)

5.5.1 Usage

The usage of the *dumptpef* application is as follows:

```
dumptpef <options>
```

The possible options of the application are as follows:

Short Name	Long Name	Description
f	file-headers	Prints the file headers.
l	logical	Prints only logical information. Can be used for checking if two files contain the same program and data and connections even if it is in different order.
m	mem	Print information about memory usage of reserved sections.
r	reloc	Prints the relocation tables.
j	section	Prints the elements of section by section index.
s	section-headers	Prints the section headers.
t	syms	Prints the symbol tables.

Chapter 6

CO-DESIGN TOOLS

6.1 Architecture Simulation and Debugging

TTA Processor **Simulator** simulates the process of running a TTA program on its target TTA processor. Provides profiling, utilization, and tracing data for Explorer, Estimator and Compiler Backend. Additionally, it offers debugging capabilities.

Input: TPEF, [ADF]

Output: TraceDB

There are two user interfaces to the simulating and debugging functionalities. One for the command line more suitable for scripting, and another with more user-friendly graphical interface more suitable for program debugging sessions. Both interfaces provide a console which supports the Tcl scripting language.

6.1.1 Processor Simulator CLI (ttasim)

The command line user interface of TTA Simulator is called 'ttasim'. The command line user interface is also visible in the graphical user interface in form of a console window. This manual covers the simulator control language used to control the command line simulator and gives examples of its usage.

6.1.1.1 Usage

The usage of the command line user interface of the simulator is as follows:

```
ttasim <options>
```

In case of a parallel simulation, a machine description file can be given before giving the simulated program file. Neither machine file or the program file are mandatory; they can also be given by means of the simulator control language.

The possible options for the application are as follows:

Short Name	Long Name	Description
a	adf	Sets the architecture definition file (ADF).
d	debugmode	Start simulator in interactive "debugging mode". This is enabled by default. Use –no-debugmode to disable.
e	execute-script	Executes the given string as a simulator control language script. For an examples of usage, see later in this section.
p	program	Sets the program to be simulated. Program must be given as a TTA program exchange format file (.TPEF)
q	quick	Simulates the program as fast as possible using the compiled simulation engine.

Example: Simulating a Parallel Program Without Entering Interactive Mode The following command simulates a parallel program until the program ends, without entering the debugging mode after simulation.

```
ttasim --no-debugmode -a machine.adf -p program.tpef
```

Example: Simulating a Program Until Main Function Without Entering Interactive Mode The following command simulates a sequential program until its main function and prints consumed clock cycles so far. This is achieved by utilizing the simulator control language and the '-e' option, which allows entering scripts from the command line.

```
ttasim --no-debugmode -e "until main; puts [info proc cycles];" -a machine.adf -p program.tpef
```

Using the Interactive Debugging Mode Simulator is started in debugging mode by default. In interactive mode, simulator prints a prompt "(ttasim)" and waits for simulator control language commands. This example uses simulator control language to load a machine and a program, run the simulation, print the consumed clock cycles, and quit simulation.

```
ttasim
(ttasim) mach machine.adf
(ttasim) prog program.tpf
(ttasim) run
(ttasim) info proc cycles
54454
(ttasim) quit
```

6.1.2 Fast Compiled Simulation Engine

The command line version of the Simulator, 'ttasim', supports two different simulation engines. The default simulation engine interprets each instruction and then simulates the processor behavior accordingly. While this is good for many cases, it can be relatively slow when compared to the computer it is being simulated on. Therefore, the Simulator also has a highly optimized mode that uses compiled simulation techniques for achieving faster simulation execution. In this simulation, the TTA program and machine are compiled into a single binary plug-in file which contains functions for simulating basic blocks directly in native machine code, allowing as fast execution as possible.

6.1.2.1 Usage

Example: Simulating a Parallel Program Using The Compiled Simulation Engine The following command simulates a parallel program using the compiled simulation engine. ("-q")

```
ttasim -a machine.adf -p program.tpef -q
```

Currently, the behaviour of the compiled simulation can only be controlled with a limited set of Simulator commands (such as 'stepi', 'run', 'until', 'kill'). Also, the simulation runs only at an accuracy of basic blocks so there is no way to investigate processor components between single cycles.

The following environment variables can be used to control the compiled simulation behavior:

Environment variable	Description	Default value
TTASIM_COMPILER	Specifies the used compiler.	"gcc"
TTASIM_COMPILER_FLAGS	Compile flags given to the compiler.	"-O0"
TTASIM_COMPILER_THREADS	Number of threads used to compile.	"3"

6.1.2.2 ccache

http://ccache.samba.org/

The compiled simulator can benefit quite a bit from different third party software. The first one we describe here is a compiler cache software called ccache. Ccache works by saving compiled binary files into a cache. When ccache notices that a file about to be compiled is the same as file found in the cache, it simply reloads file from the cache, thus eliminating recompilation of unmodified files and saving time. This can be very useful when running the same simulation program again, due to drastically reduced compilation times.

6.1.2.3 distcc

http://distcc.samba.org/

Another useful tool to use together with the compiled simulator is a distributed compiler called distcc. Distcc works by distributing the compilation of simulation engine to multiple computers and compiling the generated source files in parallel.

After installing distcc, you can set ttasim to use the distcc compiler using the following environment variable:

```
export TTASIM_COMPILER="distcc"
```

or if ccache is also installed, use:

```
export TTASIM_COMPILER="ccache distcc"
```

Also, remember to set the amount of used threads high enough. A good number of threads to use would be approximately the amount of CPU cores available. For example, setting 6 compiler threads can be done like following:

```
export TTASIM_COMPILER_THREADS=6
```

6.1.3 Simulator Control Language

This section describes all the Simulator commands that can be entered when the Simulator runs in debug mode. The Simulator displays a new line with the prompt string only when it is ready to accept new commands (the simulation is not running). The running simulation can be interrupted at any time by the key combination CTRL-c. The simulator stops simulation and prompts the user for new commands as if it had been stopped by a breakpoint.

The Simulator control language is based on the Toolset Control Language. It extends the predefined set of Tcl commands with a set of commands that allow to perform the functions listed above. In addition to predefined commands, all basic properties of Tcl (expression evaluation, parameter substitution rules, operators, loop constructs, functions, and so on) are supported.

6.1.3.1 Initialization

When the Simulator is run in debug mode, it automatically reads and executes the initialization command file '.ttasim-init' if found in the user home directory. The '.ttasim-init' file allows user to define specific simulator settings (described in section 6.1.3.2) which are enabled everytime ttasim is executed.

After the initialization command sequence is completed, the Simulator processes the command line options, and then reads the initialization command file with the same name in current working directory.

After it has processed the initialization files and the command line options, the Simulator is ready to accept new commands, and prompts the user for input. The prompt line contains the string '(ttasim)'.

6.1.3.2 Simulation Settings

Simulation settings are inspected and modified with the following commands.

setting *variable value* Sets a new value of environment variable *variable*.

setting *variable* Prints the current value contained by environment variable *variable*.

setting Prints all settings and their current values.

Currently, the following settings are supported.

bus_trace *boolean* Enables writing of the bus trace. Bus trace stores values written to each bus in each simulated clock cycle.

execution_trace *boolean* Enables writing of the basic execution trace. Basic execution trace stores the address of the executed instruction in each simulated clock cycle.

history_filename *string* The name of the file to store the command history, if command history saving is enabled.

history_save *boolean* Enables saving command history to a file.

history_size *integer* Maximum count of last commands stored in memory. This does not affect writing of the command history log, all commands are written to the log if logging is enabled.

next_instruction_printing *boolean* Print the next executed instruction when simulation stops, for example, after single-stepping or at a breakpoint.

procedure_transfer_tracking *boolean* Enables procedure transfer tracking. This trace can be used to easily observe which procedures were called and in which order. The trace is saved in 'procedure_transfer' table of Trace DB. This information could be derived from 'execution_trace', but simulation is very slow when it is enabled, this type of tracking should be faster.

profile_data_saving *boolean* Save program profile data to trace database after simulation.

rf_tracking *boolean* Enables concurrent register file access tracking. This type of tracking makes the simulation speed much worse, so it is not enabled by default. The produced statistics can be browsed after simulation by using the command 'info proc stats'.

simulation_time_statistics *boolean* Prints time statistics for the last command ran (run, until, nexti, stepi).

simulation_timeout *integer* Stops the simulation after specified timeout. Value of zero means no timeout.

static_compilation *boolean* Switch between static and dynamic compilation when running compiled simulation.

utilization_data_saving *boolean* Save processor utilization data to trace database after simulation.

6.1.3.3 Control of How the Simulation Runs

The commands described in this section allow to control the simulation process.

Before simulation can start, a program must be loaded into the Simulator. If no program is loaded, the command *run* causes the following message:

```
Simulation not initialized.
```

run Starts simulation of the program currently loaded into the Simulator. The program can be loaded by *prog* command (see Section 6.1.3.6) or may be given directly as argument, on the command line. Simulation runs until either a breakpoint is encountered or the program terminates.

resume [*count*] Resume simulation of the program until the simulation is finished or a breakpoint is reached. The *count* argument gives the number of times the *continue* command is repeated, that is, the number of times breakpoints should be ignored.

stepi [*count*] Advances simulation to the next machine instructions, stepping into the first instruction a new procedure if a function call is simulated. The *count* argument gives the number of machine instruction to simulate.

nexti [*count*] Advances simulation to the next machine instructions in current procedure. If the instruction contains a function call, simulation proceeds until control returns from it, to the instruction past the function call. The *count* argument gives the number of machine instruction to simulate.

until [*arg*] Continue running until the program location specified by *arg* is reached. Any valid argument that applies to command *break* (see Section 6.1.3.5) is also a valid argument for *until*. If the argument is omitted, the implied program location is the next instruction. In practice, this command is useful when simulation control is inside a loop and the given location is outside it: simulation will continue for as many iterations as required in order to exit the loop (and reach the designated program location).

kill Terminate the simulation. The program being simulated remains loaded and the simulation can be restarted from the beginning by means of command *run*. The Simulator will prompt the user for confirmation before terminating the simulation.

quit This command is used to terminate simulation and exit the Simulator.

6.1.3.4 Examining Program Code and Data

The Simulator allows to examine the program being simulated and the data it uses

x [*/nfu*][*addr*] This low-level command prints the data in memory starting at specified addresses *addr*. The optional parameters *n* and *u* specify how much memory to display and how to format it.

- ***n*** Repeat count: how many data words (counting by units *u*) to display. If omitted, it defaults to 1.
- ***f*** Target filename. Setting this causes the memory contents to be printed as binary data to the given file.
- ***u*** Unit size: 'b' (MAU, a byte in byte-addressed memories), 'h' (double MAU), 'w' (quadruple word, a 'word' in byte-addressed 32-bit architectures), 'g' (giant words, 8 MAU's). The unit size is ignored for formats 's' and 'i'.

If *addr* is omitted, then the first address past the last address displayed by the previous *x* command is implied. If the value of *n* or *u* is not specified, the value given in the most recent *x* command is maintained.

The values printed by command *x* are not entered in the value history (see Section 6.1.3.9).

symbol_address *datasym* Returns the address of the given data symbol (usually a global variable).

disassemble [*addr1* [*addr2*]] Prints a range of memory addresses as machine instructions. When two arguments *addr1*, *addr2* are given, *addr1* specifies the first address of the range to display, and *addr2* specifies the last address (not displayed). If only one argument, *addr1*, is given, then the function that contains *addr1* is disassembled. If no argument is given, the default memory range is the function surrounding the program counter of the selected frame.

6.1.3.5 Control Where and When to Stop Program Simulation

A breakpoint stops the simulation whenever the Simulator reaches a certain point in the program. It is possible to add a condition to a breakpoint, to control when the Simulator must stop with increased precision. There are two kinds of breakpoints: *breakpoints* (proper) and *watchpoints*. A watchpoint is a special breakpoint that stops simulation as soon as the value of an expression changes.

where *num* is a unique number that identifies the breakpoint or watchpoint and *description* describes the properties of the breakpoint. The properties include: whether the breakpoint must be deleted or disabled after it is reached; whether the breakpoint is currently disabled; the program address of the breakpoint, in case of a program breakpoint; the expression that, when modified by the program, causes the Simulator to stop, in case of a watchpoint.

bp *address* Sets a breakpoint at address *address*. Argument can also be a code label such as global procedure name (e.g. 'main').

bp *args* if Sets a conditional breakpoint. The arguments *args* are the same as for unconditional breakpoints. After entering this command, Simulator prompts for the condition expression. Condition is evaluated each time the breakpoint is reached, and the simulation only when the condition evaluates as true.

tbp *args* Sets a temporary breakpoint, which is automatically deleted after the first time it stops the simulation. The arguments *args* are the same as for the *bp* command. Conditional temporary breakpoints are also possible (see command *condition* below).

watch Sets a watchpoint for the expression *expr*. The Simulator will stop when the value of given expression is modified by the program. Conditional watchpoints are also possible (see command *condition* below).

condition [*num*] [*expr*] Specifies a condition under which breakpoint *num* stops simulation. The Simulator evaluates the expression *expr* whenever the breakpoint is reached, and stops simulation only if the expression evaluates as true (nonzero). The Simulator checks *expr* for syntactic correctness as the expression is entered.

When *condition* is given without expression argument, it removes any condition attached to the breakpoint, which becomes an ordinary unconditional breakpoint.

ignore [*num*] [*count*] Sets the number of times the breakpoint *num* must be ignored when reached. A *count* value zero means that the breakpoint will stop simulation next time it is reached.

enablebp [delete|once] [*num ...*] Enables the breakpoint specified by *num*. If *once* flag is specified, the breakpoint will be automatically disabled after it is reached once. If *delete* flag is specified, the breakpoint will be automatically deleted after it is reached once.

disablebp [*num ...*] Disables the breakpoint specified by *num*. A disabled breakpoint has no effect, but all its options (ignore-counts, conditions and commands) are remembered in case the breakpoint is enabled again.

deletebp [*num ...*] Deletes the breakpoint specified by *num*. If no arguments are given, deletes all breakpoints currently set, asking first for confirmation.

info breakpoints [*num*] Prints a table of all breakpoints and watchpoints. Each breakpoint is printed in a separate line. The two commands are synonymous.

6.1.3.6 Specifying Files and Directories

The Simulator needs to know the file name of the program to simulate/debug and, usually, the Architecture Definition File (ADF) that describes the architecture of the target processor on which the program is going to run.

prog [*filename*] Load the program to be simulated from file *filename*. If no directory is specified with *set directory*, the Simulator will search in the current directory.

If no argument is specified, the Simulator discards any information it has on the program.

mach [*filename*] Load the machine to be simulated from file *filename*. If no directory is specified with *set directory*, the Simulator will search in the current directory.

In case a parallel program is tried to be simulated without machine, an error message is printed and simulation is terminated immediately. In some cases the machine file can be stored in the TPEF file.

conf [*filename*] Load the processor configuration to be simulated from file *filename*. If no directory is specified with *set directory*, the Simulator will search in the current directory.

Simulator expects to find the simulated machine from the processor configuration file. Other settings are ignored. This can be used as replacement for the *mach* command.

6.1.3.7 Examining State of Target Processor and Simulation

The current contents of any programmer visible state, which includes any programmable register, bus, or the last data word read from or written to a port, can be displayed. The value is displayed in base 10 to allow using it easily in Tcl expressions or conditions. This makes it possible, for example, to set a conditional breakpoint which stops simulation only if the value of some register is greater than some constant.

info proc cycles Displays the total execution cycle count and the total stall cycles count.

info proc mapping Displays the address spaces and the address ranges occupied by the program: address space, start and end address occupied, size.

info proc stats In case of parallel simulation, displays current processor utilization statistics. In case 'rf_tracking' setting is enabled and running parallel simulation, also lists the detailed register file access information.

info regfiles Prints the name of all the register files of the target processor.

info registers *regfile* [*regname*] Prints the value of register *regname* in register file *regfile*, where *regfile* is the name of a register file of the target processor, and *regname* is the name of a register that belongs to the specified register file.

If *regname* is omitted, the value of all registers of the specified register file is displayed.

info funits Prints the name of all function units of the target processor.

info iunits Prints the name of all immediate units of the target processor.

info immediates *iunit* [*regname*] Prints the value of immediate register *regname* in immediate unit *iunit*, where *iunit* is the name of an immediate unit of the target processor, and *regname* is the name of a register that belongs to the specified unit.

If *regname* is omitted, the value of all registers of the specified immediate unit is displayed.

info ports *unit* [*portname*] Prints the last data word read from or written to port *portname* of unit *unit*, where *unit* may be any function unit, register file or immediate unit of the target processor. The value of the data word is relative to the selected stack frame.

If *portname* is omitted, the last value on every port of the specified unit is displayed.

info busses [*busname*] Displays the name of all bus segments of transport bus *busname*. If the argument is omitted, displays the name of the segments of all busses of the target processor.

info segments *bus* [*segmentname*]] Prints the value currently transported by bus segment *segmentname* of the transport bus *busname*.

If no segment name is given, the Simulator displays the contents of all segments of transport bus *bus*.

info program Displays information about the status of the program: whether it is loaded or running, why it stopped.

info program is_instruction_reference *ins_addr move_index* Returns 1 if the source of the given move refers to an instruction address, 0 otherwise.

info stats executed_operations Prints the total count of executed operations.

info stats register_reads Prints the total count of register reads.

info stats register_writes Prints the total count of register writes.

6.1.3.8 Miscellaneous Support Commands and Features

The following commands are facilities for finer control on the behaviour of the simulation control language.

help [*command*] Prints a help message briefly describing command *command*. If no argument is given, prints a general help message and a listing of supported commands.

6.1.3.9 Command and Value History Logs

All commands given during a simulation/debugging session are saved in a *command history log*. This forms a complete log of the session, and can be stored or reloaded at any moment. By loading and running a complete session log, it is possible to resume the same state in which the session was saved.

It is possible to run a sequence of commands stored in a command file at any time during simulation in debug mode using the *source* command. The lines in a command file are executed sequentially and are not printed as they are executed. An error in any command terminates execution of the command file.

commands [*num*]Displays the last *num* commands in the command history log. If the argument is omitted, the *num* value defaults to 10.

source *filename* Executes the command file *filename*.

6.1.4 Traces

All simulation traces are stored in a SQLite 3 binary file. The file is named after the program file by appending '.trace' to its end. This file can be browsed with the sqlite client. The sqlite client allows browsing contents of the tables using standard SQL queries.

6.1.5 Example Queries

This section provides several useful example SQL queries to retrieve data from the trace database. To query data in a generated trace database file, one should use the 'sqlite3' client to open the database after which any SQL query can be entered. The most useful queries are provides as scripts of which names start with 'dump_*'.

Instruction Execution Statistics Following query returns the instruction execution counts starting from the most frequently executed instruction. In addition, the procedure name to which each instruction belongs is printed.

```
SELECT instruction_execution_count.address AS address,
        instruction_execution_count.count AS count,
        procedure_address_range.procedure_name AS procedure
FROM instruction_execution_count, procedure_address_range
WHERE address >= procedure_address_range.first_address AND
       address <= procedure_address_range.last_address
ORDER BY count DESC;
```

Procedure Execution Profile Following query returns the count of clock cycles spent in each procedure of the program. Listing is ordered such that the procedure in which most time was spent is printed first.

```
SELECT procedure_address_range.procedure_name AS procedure,
       SUM(instruction_execution_count.count) AS cycles
FROM instruction_execution_count, procedure_address_range
WHERE address >= procedure_address_range.first_address AND
       address <= procedure_address_range.last_address
GROUP BY procedure
ORDER BY cycles DESC;
```

This query is provided in script *dump_function_profile*.

Procedure Transfer Trace This query lists the order in which each procedure was executed in the program.

```
SELECT procedure_transfer.cycle AS cycle,
        procedure_transfer.address AS address,
        procedure_transfer.type AS is_exit,
        procedure_address_range.procedure_name AS procedure
FROM procedure_transfer,procedure_address_range
WHERE address >= procedure_address_range.first_address AND
       address <= procedure_address_range.last_address
ORDER BY cycle;
```

This query is provided in script *dump_call_trace*.

6.1.6 Processor Simulator GUI (Proxim)

Processor Simulator GUI (Proxim) is a graphical frontend for the TTA Processor Simulator.

6.1.6.1 Usage

This section is intended to familiarize the reader to basic usage of Proxim. This chapter includes instructions to accomplish only the most common tasks to get the user started in using the Simulator GUI.

The following windows are available:

- *Machine State window* Displays the state of the simulated processor.
- *Disassembly window* for displaying machine level source code of the simulated application.
- *Simulator console* for controlling the simulator using the simulator control language.
- *Simulation Control Window*: Floating tool window with shortcut buttons for items in the *Program* menu.

Console Window Textual output from the simulator and all commands sent to the simulator engine are displayed in the *Simulator Console* window, as well as the input and output from the simulated program. Using the window, the simulator can be controlled directly with Simulator Control Language accepted also by the command line interface of the simulator. For list of available commands, enter *'help'* in the console.

Most of the commands can be executed using graphical dialogs and menus, but the console allows faster access to simulator functionality for users familiar with the Simulator Control Language. Additionally, all commands performed using the GUI are echoed to the console, and appended to the console command history.

The console keeps track of performed commands in command history. Commands in the command history can be previewed and reused either by selecting the command using up and down arrow keys in the console window, or by selecting the command from the **Command History**.

The *Command* menu in the main window menubar contains all GUI functionality related to the console window.

Simulation Control Window Running simulation can be controlled using the **Simulation Control** window.

Consequences of the window buttons are as follows:

- **Run/Stop:** If simulation is not running, the button is labeled 'Run', and it starts simulation of the program loaded in the simulator. If simulation is running, the button is labled 'Stop', and it will stop the simulation.
- **Stepi:** Advances simulation to the next machine instructions.
- **Nexti:** Advances simulation to the next machine instructions in current procedure.
- **Continue:** Resumes simulation of the program until the simulation is finished or a breakpoint is reached.
- **Kill:** Terminates the simulation. The program being simulated remains loaded and the simulation can be restarted from the begining.

Disassembly Window The disassembly window displays the machine code of the simulated program. The machine code is displayed one instruction per line. Instruction address and instruction moves are displayed for each line. Clicking right mouse button on an instruction displays a context menu with the following items:

- **Toggle breakpoint:** Sets a breakpoint or deletes existing breakpoint at the selected instruction.
- **Edit breakpoint...:** Opens selected breakpoint in **Breakpoint Properties** dialog.

Machine State Window The *Machine State Window* displays the state of the processor running the simulated program. The window is split horizontally to two subwindows. The window on the left is called *Status Window*, and it displays general information about the state of the processor, simulation and the selected processor block. The subwindow on the right, called *Machine Window*, displays the machine running the simulation.

The blocks used by the current instruction are drawn in red color. The block utilization is updated every time the simulation stops.

Blocks can be selected by clicking them with LMB. When a block is selected, the bottom of the *status window* will show the status of the selected block.

6.1.6.2 Profiling with Proxim

Proxim offers simple methods for profiling your program. After you have executed your program you can select "Source" -> "Profile data" -> "Highlight top execution count" from the top menu. This opens a dialog which shows execution counts of various instruction address ranges. The list is arranged in descending order by the execution count.

If you click a line on the list the disassembly window will focus on the address range specified on that line. You can trace in which function the specific address range belongs to by scrolling the disassembly window up until you find a label which identifies the function. You must understand at least a little about assembly coding to find the actual spot in C code that produces the assembly code.

6.2 Processor Cost/Performance Estimator (estimate)

Processor Cost/Performance **Estimator** provides estimates of energy consumption, die area, and maximum clock rate of TTA designs (program and processor combination).

Input: ADF, IDF, [TPEF and TraceDB]

Output: Estimate (printed to the standard output)

6.2.1 Command Line Options

The usage of the *estimate* application is as follows:

```
estimate {-p [TPEF] -t [TraceDB]} ADF IDF
```

The possible options of the application are as follows:

Short Name	Long Name	Description
p	program	Sets the TTA program exchange format file (TPEF) from which to load the estimated program (required for energy estimation only).
t	trace	sets the simulation trace database (TraceDB) from which to load the simulation data of the estimated program (required for energy estimation only).
a	total-area	Runs total area estimation.
l	longest-path	Runs longest path estimation.
e	total-energy	Runs total energy consumption estimation.

If `tpef` and `tracedb` are not given, the energy estimation will NOT be performed since the Estimator requires utilization information about the resources. However, the area and timing estimation will be done. If only one of `tracedb` and `tpef` is given, it is ignored.

6.3 Automatic Design Space Explorer (explore)

Automatic Design Space **Explorer** automates the process of searching for target processor configurations with favourable cost/performance characteristics for a given set of applications by evaluating hundreds of processor configurations.

Input: ADF (a starting point architecture), TPEF, HDB

Output: ExpResDB (Section 2.2.8)

6.3.1 Explorer Application format

Applications are given as directories that contain the application specific files to the Explorer. Below is a description of all the possible files inside the application directory.

file name	Description
program.bc	The application byte code.
description.txt	The application description.
simulate.ttasim	TTASIM simulation script piped to the TTASIM to produce ttasim.out file. If no such file is given the simulation is started with "until 0" command.
correct_simulation_output	Correct output of the simulation used in verifying.
max_runtime	The applications maximum runtime.
setup.sh	Simulation setup script, if something needs to be done before simulation.
verify.sh	Simulation verify script for additional verifying of the simulation, returns 0 if OK. If missing only correct_simulation_output is used in verifying.

Below is an example of the file structure of *HelloWorld* application. As The maximum runtime file is missing application is expected not to have a maximum runtime requirement.

```
HelloWorld/program.bc
HelloWorld/correct_simulation_output
HelloWorld/description.txt
```

6.3.2 Command Line Options

The exploration result database <*output_dsdb*> is required always. The database can be queried applications can be added into and removed from the database and the explored configurations in the database can be written as files for further examination.

The possible options of the application are as follows:

Short Name	Long Name	Description
d	add_app_dir	Path(s) of the test application(s) to be added into the DSDB.
a	adf	ADF to add into the DSDB.
n	conf_count	Print the number of machine configurations in the DSDB.
c	conf_summary	Print the summary of machine configurations in the DSDB ordered by: Ordering may be one of the following: *I* ordering by configuration Id, *P* ordering by application path, *C* ordering by cycle count, *E* ordering by energy estimate.
e	explorer_plugin	Design Space Explorer plugin to be used.
b	hdb	HDB to use with exploration.
i	idf	IDF to add into the DSDB, needs also ADF.
l	list_apps	List the applications in the DSDB.
g	list_plugins	List loadable explorer plugins.
p	plugin_info	Get information about a plugin.
u	plugin_param	Parameter to the explorer plugin in form 'name=value'. If parameter value is boolean, use 'true', 'false', 1 or 0.
r	rm_app	ID(s) of the test program path(s) to be removed from the DSDB.
s	start	Starting point configuration ID in the DSDB.
w	write_conf	export the ADF and IDF files from the DSDB with given configuration id. Does not remove the configuration from the DSDB.

Depending on the exploration plugin, the exploring results machine configurations in to the exploration result database dsdb. The best results from the previous exploration run are given at the end of the exploration:

```
explore -e RemoveUnconnectedComponents -a data/FFTTest -hdb=data/initial.hdb data/test.dsdb
```

```
Best result configurations:
 1
```

Exploration plugins may also estimate the costs of configurations with the available applications. If there are estimation results for the configuratios those can be queried with option **–conf_summary** by giving the ordering of the results.

The Explorer plugins explained in chapters below can be listed with a command:

```
explore -g
```

And their parameters with a command:

```
explore -p <plugin name>
```

These commands can help if, for some reason, this documentation is not up-to-date.

6.3.3 Explorer Plugin: SimpleICOptimizer

SimpleICOptimizer is an explorer plugin that optimizes the interconnection network of the given configuration by removing the connections that are not used in the parallel program.

This is so useful functionality especially when generating ASIPs for FPGAs that there's a shortcut script for invoking the plugin.

Usage:

```
minimize-ic unoptimized.adf program.tpef target-ic-optimized.adf
```

However, if you want more customized execution, you should read on.

Parameters that can be passed to the SimpleICOptimizer are:

Param Name	Default Value	Description
tpef	no default value	name of the scheduled program file
add_only	false	Boolean value. If set true the connections of the given configuration won't be emptied, only new ones may be added
evaluate	true	Boolean value. True evaluates the result config.

If you pass a scheduled tpef to the plugin, it tries to optimize the configuration for running the given program. If multiple tpefs are given, the first one will be used and others discarded. Plugin tries to schedule sequential program(s) from the application path(s) defined in the dsdb and use them in optimization if tpef is not given.

Using the plugin requires user to define the configuration he wishes optimize. This is done by giving *-s <configuration_ID>* option to the explorer.

Let there be 2 configurations in database.dsdb and application directory path app/. You can optimize the first configuration with:

```
explore -e SimpleICOptimizer -s 1 database.dsdb
```

If the optimization was succesfull, explorer should output:

```
Best result configuration:
 3
```

Add_only option can be used for example if you have an application which isn't included in application paths defined in database.dsdb but you still want to run it with the same processor configuration. First export the optimized configuration (which is id 3 in this case):

```
explore -w 3 database.dsdb
```

Next schedule the program:

```
schedule -t 3.adf -o app_dir2/app2.scheduled.tpef app_dir2/app2.seq
```

And then run explorer:

```
explore -e SimpleICOptimizer -s 3 -u add_only=true -u tpef=app_dir2/app2.scheduled.tpef
database.dsdb
```

The plugin now uses the optimized configuration created earlier and adds connections needed to run the other program. If the plugin finds a new configuration it will be added to the database, otherwise the existing configuration was already optimal. Because the plugin won't remove existing connections the new machine configuration is able to run both programs.

6.3.4 Explorer Plugin: RemoveUnconnectedComponents

Explorer plugin that removes unconnected ports from units or creates connections to these ports if they are FUs, but removes FUs that have no connections. Also removes unconnected buses. If all ports from a unit are removed, also the unit is removed.

You can pass a parameter to the plugin:

Param Name	Default Value	Description
allow_remove	false	Allows the removal of unconnected ports and FUs

When using the plugin you must define the configuration you wish the plugin to remove unconnected components. This is done by passing *-s <configuration_ID>* to explorer.

If you do not allow removal the plugin will connect unconnected ports to some sockets. It can be done with:

```
explore -e RemoveUnconnectedComponents -s 3 database.dsdb
```

or

```
explore -e RemoveUnconnectedComponents -s 3 -u allow_remove=false database.dsdb
```

if you wish to emphasise you do not want to remove components. This will reconnect the unconnected ports from the configuration 3 in database.dsdb.

And if you want to remove the unconnected components:

```
explore -e RemoveUnconnectedComponents -s 3 -u allow_remove=true database.dsdb
```

6.3.5 Explorer Plugin: GrowMachine

GrowMachine is an Explorer plugin that adds resources to the machine until cycle count doesn't go down anymore.

Parameters that can be passed to the GrowMachine are:

Param Name	Default Value	Description
superiority	2	Percentage value of how much faster schedules are wanted until cycle count optimization is stopped

Using the plugin requires user to define the configuration he wishes optimize. This is done by giving *-s <configuration_ID>* option to the explorer.

Example of usage:

```
explore -e GrowMachine -s 1 database.dsdb
```

6.3.6 Explorer Plugin: ImmediateGenerator

ImmediateGenerator is an Explorer plugin that creates or modifies machine instruction templates. Typical usage is to split an instruction template slot among buses.

Parameters that can be passed to the ImmediateGenerator are:

Param Name	Default Value	Description
print	false	Print information about machines instruction templates.
remove_it_name	no default value	Remove instruction template with a given name
add_it_name	no default value	Add empty instruction template with a given name.
modify_it_name	no default value	Modify instruction template with a given name.
width	32	Instruction template supported width.
width_part	8	Minimum size of width per slot.
split	false	Split immediate among slots.
dst_imm_unit	no default value	Destination immediate unit.

Example of adding a new 32 width immediate template named newTemplate that is splitted among busses:

```
explore -e ImmediateGenerator -s 1 -u add_it_name="newTemplate" -u width=32 -u split=true
database.dsdb
```

6.3.7 Explorer Plugin: ImplementationSelector

ImplementationSelector is an Explorer plugin that selects implementations for units in a given configuration ADF. It creates a new configuration with a IDF.

Parameters that can be passed to the ImmediateGenerator are:

Param Name	Default Value	Description
ic_dec	DefaultICDecoder	Name of the ic decoder plugin.
ic_hdb	asic_130nm_1.5V.hdb	name of the HDB where the implementations are selected.
adf	no default value	An ADF for the implementations are selected if no database is used.

Example of creating implementation for configuration ID 1, in the database:

```
explore -e ImplementationSelector -s 1 database.dsdb
```

6.3.8 Explorer Plugin: MinimizeMachine

MinimizeMachine is an Explorer plugin that removes resources from a machine until the real time requirements of the applications are not reached anymore.

Parameters that can be passed to the ImmediateGenerator are:

Param Name	Default Value	Description
min_bus	true	Minimize buses.
min_fu	true	Minimize function units.
min_rf	true	Minimize register files.
frequency	no default value	Running frequency for the applications.

Example of minimizing configuration ID 1, in the database, with a frequency of 50 MHz:

```
explore -e MinimizeMachine -s 1 -u frequency=50 database.dsdb
```

Chapter 7

FREQUENTLY ASKED QUESTIONS

7.1 Memory Related

Questions related to memory accessing.

7.1.1 What is the endianness of the TTA processors designed with TCE?

Big endian. At the moment, endianness cannot be customized.

7.1.2 What is the alignment of words when reading/writing memory?

The memory acessing operations in the base operation set (ldq, ldh, ldw, ldd, stq, sth, stw, and ldd) are aligned with their size. Operations stq/ldq are for accessing single minimum addressable units (MAU, usually bytes), thus their alignment is 1 MAU, for sth/ldh it is 2 MAUs, and for stw/ldw it is 4 MAUs. Thus, one cannot access, for example, a 4 MAU word at address 3.

Double precision floating point word operations std/ldd which access 64-bit words are aligned at 8-byte addresses. Thus, if your memory is addressed in 16-bit units, double words can be stored at addresses divisible by 4, if memory is byte-addressed, then addresses must be divisible by 8, and so on.

7.1.3 Load Store Unit

In the LSUs shipped with TCE the two LSB bits are used by the LSU to control a so called write mask that handles writing of bytes. This means that the memory address outside the processor is 2 bits narrower than inside the processor. When you set the data address space width in ProDe, the width is the address width inside the processor.

7.1.4 Instruction Memory

The default GCU assumes that instruction memory is intruction addressable. In other words the instruction word must fit in the MAU of the instruction memory. This way the next instruction can be referenced by incrementing the current memory address by one.

How to interface the instruction memory with an actual memory chip is out of scope in TCE because there are too many different platforms and chips and possibilities. But as an advantage this gives the user free hands to implement almost any kind of memory hierarchy the user wants. Most probably you must implement a memory adapter to bind the memory chip and the TTA processor interfaces together.

7.1.5 Stack and Heap

The heap (allocated with 'malloc' in C or 'new' in C++) and stack grow towards each other. Stack growes from the end of the data address space towards the beginning of the address space, while heap grows towards the end of the address space starting from the end of the global data area.

Thus, the correct way to increase the space for heap/stack is to increase the size of your data memory address space in the ADF.

7.2 Processor Generator

7.2.0.1 Warning: Processor Generator failed to generate a test bench

The automatic testbench generator currently does not support any special function units that connect signals out from toplevel.vhdl. This warning can be ignored if you are not planning to use the automatically generated test bench.

7.2.0.2 Warning: Opcode defined in HDB for operation ...

Processor Generator gives a warning message if the operation codes in FU are not numbered according to the alphabetical order of the operations. The VHDL implementation and the HDB entry of the FU should be fixed to use this kind of opcode numbering. See section 4.6.1 for more details.

Chapter 8

TROUBLESHOOTING

This chapter gives solutions to common problems encountered while using TCE.

8.1 Simulation

Problems with simulation, both with command line (ttasim) and graphical user interfaces (proxim) are listed here.

8.1.1 Failing to Load Operation Behavior Definitions

It might be possible that you have defined some custom operations to `~/.tce/opset/custom` which conflict with your new definitions, or the simulation behaviors are compiled with an older compiler and not compatible with your new compiler. Workaround for this is to either delete the old definitions or rebuild them.

8.2 Limitations of the Current Toolset Version

This section lists the most user-visible limitations placed by the current toolset version.

8.2.1 Integer Width

The simulator supports only integer computations with maximum word width of 32 bits.

8.2.2 Instruction Addressing During Simulation

The details of encoding and compression of the instruction memory are not taken into account before the actual generation of the bit image of the instruction memory. This decision was taken to allow simplification in the other parts of the toolset, and to allow easy "exploration" with different encodings and compression algorithms in the bit generation phase.

This implies that every time you see an instruction address in architectural simulation, you are actually seeing an instruction index. That is, instruction addressing (one instruction per instruction memory address) is assumed.

We might change this in the future toolset versions to allow seeing exact instruction memory addresses during simulation, if that is seen as a necessity. Currently it does not seem to be a very important feature.

8.2.3 Data Memory Addressing

There is no tool to map data memory accesses in the source code to the actual target's memories. Therefore, you need to have a data memory which provides byte-addressing with 32-bit words. The data will be accessed using operations LDQ, LDH, LDW, STQ, STH, STW, which access the memory in 1 (q), 2 (h), and 4 (w) byte chunks. This should not be a problem, as it is rather easy to implement byte-addressing in case the actual memory is of width of 2's exponent multiple of the byte. The parallel assembler allows any kind of minimum addressable units (MAU) in the load/store units. In that case, LDQ/STQ just access a single MAU, etc. One should keep in mind the 32-bit integer word limitation of simulation. Thus, if the MAU is 32-bits, one cannot use LDH or LDW because they would require 64 and 128 bits, respectively.

8.2.4 Ideal Memory Model in Simulation

The simulator assumes ideal memory model which generates no stalls and returns data for the next instruction. This so called 'Ideal SRAM' model allows modeling all types of memories in the point of view of the programmer. It is up to the load/store unit implementation to generate the lock signals in case the memory model does not match the ideal model.

There are hooks for adding more memory models which generate lock signals in the simulation, but for the v1.0 the simulator does not provide other memory models, and thus does not support lock cycle simulation.

8.2.5 Guards

The guard support as specified in the ADF specification [CSJ04] is only partially supported in TCE. 'Operators other than logical negation are not supported. That is, supported guards always "watch" a single register (FU output or a GPR). In addition, the shipped default scheduling algorithm in compiler backend requires a register guard. Thus, if more exotic guarded execution is required, one has to write the programs in parallel assembly (Section 5.3).

8.2.6 Operation Pipeline Description Limitations

Even though supported by the ADF and ProDe, writing of operands after triggering an operation is not supported neither by the compiler nor the simulator. However, setting different latencies for outputs of multi-result operations is supported. For example, using this feature one can have an iterative operation pipeline which computes several results which are ready after the count of stages in an iteration.

8.2.7 Encoding of XML Files

TCE uses XML to store data of the architectures and implementation locations (see Section 2.2.1 and Section 2.2.3). The encoding of the XML files must be in 7-bit ascii. If other encodings are used, the result is undefined.

8.2.8 Floating Point Support

The simulator supports both the single (32 bits) and double (64 bits) precision floating point types. However, at this time only single precision float operations (ADDF, MULF, etc.) are selected automatically when starting from C/C++ code. Currently, the compiler converts doubles to floats to allow compiling and running code with doubles with reduced precision.

Chapter 9

Copyright notices

Here are the copyright notices of the libraries used in the software.

9.1 Xerces

Xerces-C++ is released under the terms of the Apache License, version 2.0. The complete text is presented here.

Apache License Version 2.0, January 2004 http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including

but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS

FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Do Not include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

9.2 wxWidgets

Copyright (c) 1998 Julian Smart, Robert Roebling [, ...]

Everyone is permitted to copy and distribute verbatim copies of this licence document, but changing it is not allowed.

WXWINDOWS LIBRARY LICENCE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public Licence as published by the Free Software Foundation; either version 2 of the Licence, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public Licence for more details.

You should have received a copy of the GNU Library General Public Licence along with this software, usually in a file named COPYING.LIB. If not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

EXCEPTION NOTICE

1. As a special exception, the copyright holders of this library give permission for additional uses of the text contained in this release of the library as licenced under the wxWindows Library Licence, applying either version 3 of the Licence, or (at your option) any later version of the Licence as published by the copyright holders of version 3 of the Licence document.

2. The exception is that you may use, copy, link, modify and distribute under the user's own terms, binary object code versions of works based on the Library.

3. If you copy code from files distributed under the terms of the GNU General Public Licence or the GNU Library General Public Licence into a copy of this library, as this licence permits, the exception does not apply to the code that you add in this way. To avoid misleading anyone as to the status of such modified files, you must delete this exception notice from such code and/or adjust the licensing conditions notice accordingly.

4. If you write modifications of your own for this library, it is your choice whether to permit this exception to apply to your modifications. If you do not wish that, you must delete the exception notice from such code and/or adjust the licensing conditions notice accordingly.

9.3 L-GPL

GNU LIBRARY GENERAL PUBLIC LICENSE

==========

Version 2, June 1991

Copyright (C) 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and do not assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

GNU LIBRARY GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as

such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED

OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this library; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990 Ty Coon, President of Vice

That's all there is to it!

9.4 TCL

Tcl/Tk License Terms

This software is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., Scriptics Corporation, and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee

is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only "Restricted Rights" in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as "Commercial Computer Software" and the Government shall have only "Restricted Rights" as defined in Clause 252.227-7013 (c) (1) of DFARs. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the software in accordance with the terms specified in this license.

9.5 SQLite

SQLite is public domain. For more information, see http://www.sqlite.org/copyright.html

9.6 Editline

-

Copyright (c) 1997 The NetBSD Foundation, Inc.

All rights reserved.

This code is derived from software contributed to The NetBSD Foundation by Jaromir Dolecek.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the NetBSD Foundation, Inc. and its contributors.

4. Neither the name of The NetBSD Foundation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY

WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Bibliography

[Cor97] Henk Corporaal. *Microprocessor Architectures; from VLIW to TTA*. John Wiley, 1997.

[CSJ04] Andrea Cilio, Henjo Schot, and Johan Janssen. Processor Architecture Definition File Format for a New TTA Design Framework. S-003, 2004.

[llv08] The LLVM project home page. Website, 2008. `http://www.llvm.org` .