TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology

LASSE LAASONEN

PROGRAM IMAGE AND PROCESSOR GENERATOR FOR TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Examiners: Prof. Tommi Mikkonen and Prof. Jarmo Takala
Examiners and subject approved by Department Council
12th April 2006

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology
Laasonen, Lasse Sampo: Program Image and Processor Generator for Transport Triggered Architectures
Master of Science Thesis: 47 pages
April 2007
Main subject: Software Engineering
Examiners: Prof. Tommi Mikkonen and Prof. Jarmo Takala
Keywords: transport triggered architecture, processor generator, program image generator

There is often need to run applications with high performance and energy-efficiency requirements in embedded systems. The best results can be achieved with a tailor-made processor for the particular purpose. Application specific instruction set processors (ASIP) provide an approach to the problem. To make the design process as cost-effective as possible, it have to be automated at least partially.
TTA-Based Codesign Environment (TCE) is a design environment developed in Tampere University of Technology. It is based on transport triggered processor architecture (TTA) which can be easily tailored. TCE consists of several different applications, by means of which the processor can be designed semi-automatically phase by phase.
In this thesis, two applications were designed and implemented to provide tools that can be used to perform the last phase of designing a processor with TCE. Another of the applications, Program Image Generator (PIG), generates bit-level image of the application to be executed by a TTA-processor. It takes advance of the given encoding map which defines how different instructions should be encoded to bit patterns. In addition, it is given a higher level but compeletely detailed description of the program from which the image is to be generated. The other application is Processor Generator (ProGe) which generates the designed processor in a hardware description language for synthesis. The most important features and the software architecture of the applications are discussed in this thesis. Finally, the verification of the correct functionality of the applications is discussed.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
Laasonen, Lasse Sampo: Ohjelmakuva- ja prosessorigeneraattori siirtopohjaisille arkkitehtuureille
Diplomityö: 47 sivua
Huhtikuu 2007
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Tommi Mikkonen ja prof. Jarmo Takala
Avainsanat: siirtopohjainen prosessoriarkkitehtuuri, prosessorigeneraattori, ohjelmakuvageneraattori

Sulautetuissa järjestelmissä on usein tarve suorittaa sovelluksia, joilla on suuret tehokkuus- sekä virrankulutusvaatimukset. Parhaaseen tulokseen päästään kyseiseen tarkoitukseen räätälöidyllä prosessoriarkkitehtuurilla. Sovelluskohtaiset käskykantaprosessorit (ASIP) tarjoavat erään lähestymistavan tähän ongelmaan. Jotta prosessorin suunnitteluprosessi mahdollisimman kustannustehokas, se täytyy pystyä automatisoimaan ainakin osittain.

TTA-Based Codesign Environment (TCE) on Tampereen teknillisessä yliopistossa kehitetty suunnitteluympäristö, jossa suunnittelun pohjana käytetään helposti räätälöitävää siirtopohjaista prosessoriarkkitehtuuria (Transport Triggered Architecture, TTA). TCE koostuu useista eri sovelluksista, joiden avulla suunnitteluprosessi voidaan hoitaa puoliautomaattisesti vaihe vaiheelta.

Tässä diplomityössä suunniteltiin sekä toteutettiin kaksi sovellusta, joilla voidaan suorittaa prosessorin suunnittelun viimeinen vaihe TCE:ssä. Toinen sovelluksista, ohjelmakuvageneraattori (Program Image Generator, PIG), generoi TTA-prosessorilla suoritettavasta ohjelmasta bittitason kuvauksen. Se käyttää hyväkseen annettua koodauskarttaa, jossa on määritelty, kuinka erilaiset käskyt tulee koodata bittikuvioksi. Lisäksi sille annetaan korkeamman tason, joskin täysin yksityiskohtainen kuvaus generoitavasta ohjelmasta. Toinen sovellus on prosessorigeneraattori (Processor Generator, ProGe), joka generoi TCE:llä suunnitellun prosessorin laitteistonkuvauskielellä syntesointia varten. Diplomityössä on esitelty ohjelmien tärkeimmät ominaisuudet sekä ohjelmistoarkkitehtuuri. Lopuksi on kerrottu, kuinka ohjelmien toiminnallisuus varmennettiin.

PREFACE

This M.Sc thesis was completed in Institute of Digital and Computer Systems of Tampere University of Technology (TUT) in 2005-2006 for codesign software implemented as part of the Flexible Design Methods for DSP Systems (FlexDSP) project funded by the National Technology Agency.

I would like to thank Professor Jarmo Takala for giving me a chance to be part of this interesting project, for sharing his hardware expertise and for his improvement ideas to this thesis. I am also extremely grateful to Teemu Pitkänen and Jari Heikkinen for advise in VHDL and hardware related issues. I would also like to thank all the people in TCE project for making the relaxed atmosphere in which it was nice to work. Thanks to Professor Tommi Mikkonen too for improvement proposals to the thesis.

Tampere, April 1, 2007

Lasse Laasonen

CONTENTS

LIST OF ABBREVIATIONS

ADF	Architecture Definition File
ASIP	Application-Specific Instruction set Processor
CISC	Complex Instruction Set Computer
FFT	Fast Fourier Transform
FU	Function Unit
GCU	Global Control Unit
GPP	General-Purpose Processor
HDL	Hardware Desciption Language
IC	Interconnection Network
IU	Immediate Unit
MAU	Minimum Addressable Unit
MOM	Machine Object Model
PIG	The Program Image Generator
POM	Program Object Model
ProGe	The TTA Processor Generator
RF	Register File
RISC	Reduced Instruction Set Computer
STL	Standard Template Library
TCE	TTA-Based Codesign Environment
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architecture
VHDL	Very high speed integrated circuit Hardware Description Language
XML	Extensible Markup Language

1. INTRODUCTION

The number of solutions of embedded systems is increasing all the time. Processors designed for them usually have stricter requirements than general-purpose processors (GPP) used in desktop computers, for example. Energy consumption of embedded systems is often a critical factor, if the system is powered by a battery. The size of the product must be as small as possible, if it is intended to be fitted into pocket. The production costs of the product are important too, if it is going to be sold in large amount.

To meet the requirements, a custom-designed application specific instruction set processor (ASIP) might be the right solution. Embedded systems usually execute only a limited set of applications, so ASIP is suitable for the purpose. Sometimes, a GPP is used as the main processor of an embedded system and ASIP is a slave processor specialised in a task, such as signal processing, for example. ASIP is codesigned with the type of software it is going to execute in the final product. It can be customised by adding instructions often needed by the sofware to the instruction set. Respectively, instructions that are rarely used, can be removed from the processor to simplify it. It is often the case, that the application executes same set of instructions sequentially several times, for example in a loop. It is possible to create a special instruction that does the same job. By adding this instruction to the ASIP, the number of cycles required to execute the application reduces, thus the performance of the processor gets improved.

Designing ASIPs is not an easy task. Finding the optimal architecture from hundreds of different variations is very time consuming, if not impossible, by hand. Verification of the processor architecture is also very time consuming and prone to errors. There is clearly need for a software toolset which can assist in the designing and verification process. TTA Codesign Environment (TCE) is such a toolset being developed in Tampere University of Technology. It is based on Transport Triggered Architecture (TTA) paradigm. TTA is a processor architecture which can be customised by adding function units (FU), register files (RF) and modifying the interconnection network (IC) which provides data paths between the units. TTA aims to move the complexity from hardware to software. The instructions of TTA are at very low level: They define directly the data moves from unit to another. This results in rather simple decoding logic in the processor hardware, but sets

demanding challenges to the software compiler.

In this thesis, the program image (PIG) and the processor generator (ProGe) were implemented for TCE toolset. Those tools are exploited in the last phase of the design flow followed in TCE. Program image generator generates the bit patterns of the applications compiled for the target architecture and data memory contents. The processor generator is used to generate the structural description of the processor in hardware description language (HDL).

At first, in Chapter 2 the thesis gives an overview of TTA. Chapter 3 concentrates on TCE. It describes the design flow used to design the processor and introduces the main file formats used. The thesis continues by two main chapters: Chapter 4 tells about PIG and Chapter 5 about ProGe. Their software architecture and operational principles come out in those chapters. The verification of both of the applications are discussed in Chapter 6. This thesis is summarised in Chapter 7 by presenting the conclusions of the work done.

2. TRANSPORT TRIGGERED ARCHITECTURE

Transport Triggered Architecture (TTA) is a modular, customizable processor architecture template, which provides an easy way to design optimized application-specific processors. They can be customized by adding required resources to create a processor which is very efficient to execute the program at hand.

Traditionally processors have fixed resources. They are designed to execute all kinds of programs relatively efficiently and therefore are good for multi-purpose uses. The idea of TTA processor is not to compete against these kinds of processors. On the contrary, TTA processors, like other customizable processor architectures, are designed for an application or a small set of applications at hand. By designing processor for some particular applications, it can be better optimised for its purpose than conventional general-purpose processors.

The number of embedded systems is increasing all the time, which probably paves the way for TTA processors too. There is often need for application-specific processors in embedded systems, since energy-efficiency and price-quality ratio are important factors.

This chapter introduces the TTA processor organisation at first and continues by discussing the software characteristics of TTA. Finally, a more detailed view to TTA is taken by explaining the hardware operation principles of TTA. Additional information of TTA concept is described in [1].

2.1 Processor Organisation

TTA processor consists of a global control unit (GCU), interconnection network, function units, register files and immediate units (IU), as depicted in Figure 2.1. Control unit decodes instructions and generates control signals that cause data to move between function units and register files via interconnection network. Function units perform operations to data and register files contain registers for temporary data storage. Immediate units are specialised register files that are written by control unit only. The TTA processor template covered by this thesis is discussed more detailed in 3.2.

The control unit generated by the Processor Generator covered by this thesis is further divided to three sub modules: instruction fetch unit, instruction decompres-

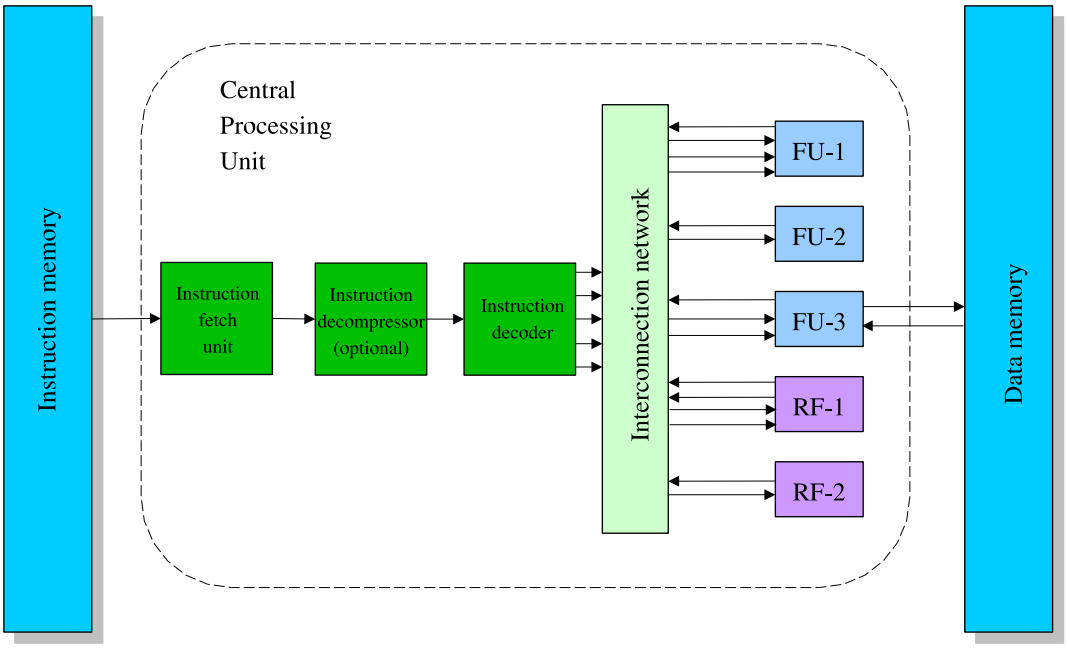

Figure 2.1: TTA Processor Organization.

sor and instruction decoder. The task of instruction fetch unit is to fetch instructions from instruction memory. In the case of compressed instructions, decompressor decompresses them and finally the instruction decoder generates the control signals of the processor to execute the instruction.

Usually processors need some data memory too. This is enabled by including a *load-store* unit in the processor. It acts between the data memory and the data path of the processor. In TTA, the load-store unit is implemented as a normal function unit among others. Load-store unit has operations to load and store data to/from data memory. Since TTA is a customizable architecture, there may be several load-store units and several data memory banks.

2.2 Software Characteristics

TTA has a different approach to programming than conventional processors. Traditionally, in general-purpose processors (GPP), instructions define directly the operations to execute and their operands. The abstraction level is much higher than in TTA instructions. TTA instructions define data moves for the buses in the interconnection network. That is, each instruction defines as many data moves as there are buses in the processor. For each data move, source and destination ports are defined, of course. As a consequence, operations are performed.

To execute an operation, let's say *add*, in TTA, there must be a function unit which supports *add* operation in the processor. *Add* operation has two operands, thus the function unit supporting the operation must have two input ports for the

operands, and an output port for the result. The other input port is so-called *trigger* port. When data is tranported to the *trigger* port, the operation execution is triggered and the result can be read from the output port after the operation latency has expired. That is why the architecture is called Transport Triggered Architecture.

For example, converting the assembly instruction

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

which sums r1 and r2 and puts the result to r3, of conventional processor to TTA instruction results

```
r1 -> fu1.o1, r2 -> fu1.t1.add;
fu1.r1 -> r3;
```

In the example, *fu1* is the function unit that implements the *add* operation. In the first instruction, operands are moved to the input ports of the function unit from registers *r1* and *r2*, and in the second instruction, the result is moved from the function unit to register *r3*. As the example above shows, TTA assembly programmer has to take also the operation latency into account. In this example, the latency of *add* operation is one: The result is ready in the next cycle from triggering the operation.

TTA program code can be *sequential* or *parallel*. In the sequential TTA code, the data moves are defined in a sequential order. They are not mapped to any bus in sequential code. Operation latencies are not taken into account either. When compiling code for TTA, sequential TTA code is generated at first. From the sequential code, the *scheduler* generates the final parallel TTA code which is executed by the processor. The example above shows parallel TTA code. The data moves are mapped to transport buses and all the details of the target architecture, including operation latencies, are taken into account to make sure the code works in hardware.

TTA instructions are at the lower level than RISC and CISC instructions. Therefore writing TTA programs in assembly is harder, but on the other hand, more optimisations can be done in the software. Basically, complexity is moved from the processor hardware to the software. However, it must be noted, that processors using instruction level parallelism are not traditionally meant to be programmed in assembly but a parallelising compiler is exploited.

2.3 Hardware Operation Principles

Hardware of a TTA processor is much simpler than in conventional processor architectures. Because TTA instruction are at a lower level of abstraction, the processor itself does not need to have as complex decoding logic and it does not need to care

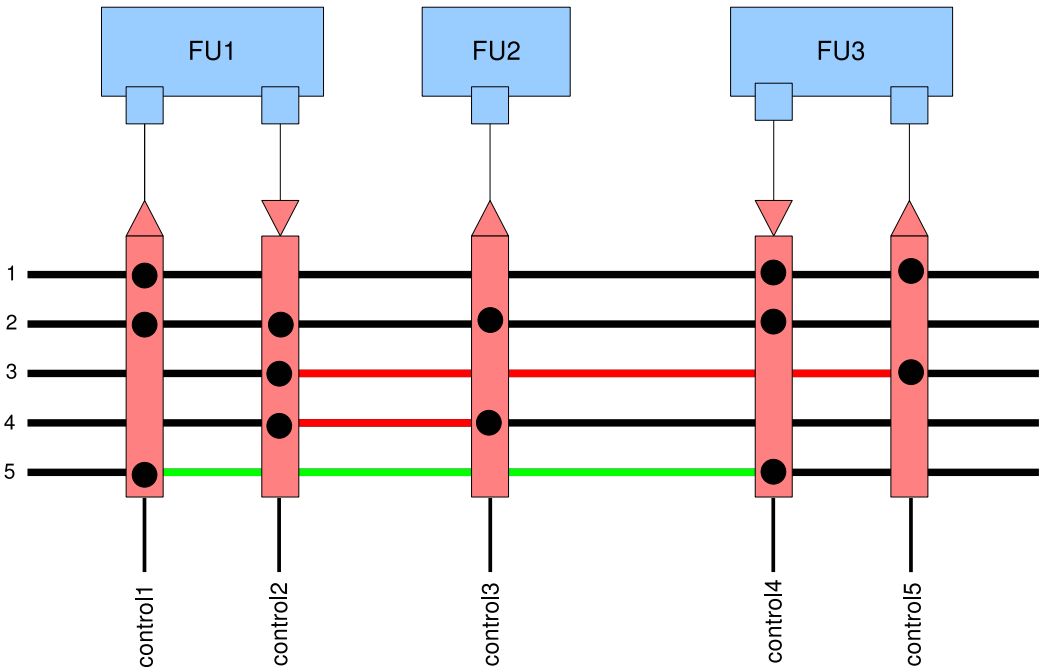

Figure 2.2: Control signals of interconnection network.

about operation latencies, for example. TTA processor simply moves data from place to place as the instruction determines. Of course there are function units, which might be rather sophisticated, but their implementation does not make the decoding logic more complicated at all.

Operation of TTA processor is based on control signals of sockets and units connected to them. Control signals of output sockets determine which buses should be written by the sockets. Respectively, control signals of input sockets determine the bus from which input sockets must read the data. Figure 2.2 depicts a case in which three data move are performed.

There are two kind of control signals for function units and register files: *load signals* and *operation code signals*. Load signals indicate that data should be loaded into a function unit or a register file from an input socket. In addition, reading data from a register file to an output socket requires a load signal to the register file. It is used to control the register file to update the value of the output port.

Operation code signals, in short opcode signals, define the operation to execute by a function unit. All the function units do not need opcode signals, since some of them implement just one operation. Register files also need opcode signals which define the register number from/to which the data is to be loaded.

3. TTA-BASED CODESIGN ENVIRONMENT

The first TTA-based codesign environment was MOVE framework [2]. The environment was originally developed in Delft University in Netherlands. Further development has taken place in Tampere University of Technology in Finland.

Developing the TTA-Based Codesign Environment (TCE) was started in 2002. The main focus of designing TCE was to create a framework that can be used to experiment new algorithms. In many places, where different algorithms could be experimented, TCE uses dynamic modules which can be created later and taken into use without recompiling the whole toolset.

To ensure good, extendable software architecture, TCE has been developed in controlled manner. Each module has been designed carefully before starting to implement them, and automatic unit and system test are ran once a day on several computers and operating systems.

This chapter tells shortly about the design flow when designing a TTA processor and introduces the most important file formats and data structures involved in program image and processor generation.

3.1 Design Flow

Designing a processor architecture from scratch follows a particular design flow having different phases. In some cases, all the phases are not performed or might be done manually. The design flow can be divided into four phases: *sequential code generation*, *design space exploration*, *parallel code generation and analysis* and finally *program image and processor generation*. The phases are briefly described in this section. For more detailed information, see [3].

3.1.1 Sequential Code Generation

Sequential code generation is the first phase of the design flow. It means generating the initial sequential TTA code. The source code can be written in a high-level programming language, such as C. The sequential TTA code is compiled from the source code by a frontend compiler. The compiler converts the high-level program code to sequential assembly code using a well-defined operation set. The target processor architecture is not taken into account in this phase. The sequential code just defines the operations that should be executed and their order.

3.1.2 Design Space Exploration

Design space exploration is phase, in which the optimal processor architecture for the given sequential code is explored. Exploration is performed by reducing resources from the given initial processor architecture in stages. Then the program code is scheduled and simulated for each architecture and processor costs, such as silicon area, price and energy consumption are estimated. Finally, the best processor architecture is found, depending on the given cost restrictions and objectives.

3.1.3 Parallel Code Generation and Analysis

In this phase, sequential TTA code is scheduled for the given processor architecture, and simulated. This phase can also be regarded as part of the design space exploration, since scheduler and simulator are exploited as the best processor architecture is being explored. Scheduling the sequential code for the given target architecture is probably the most challenging task in the design flow. The scheduler tries to schedule the data moves of sequential code to create as efficient parallel code for the given target architecture as possible. The output of the scheduler, parallel TTA assembly code, is simulated by a TTA processor simulator to get the accurate cycle count of execution. The simulator provides also profilation data for the explorer, which helps it to make better choices when exploring different target architectures.

3.1.4 Program Image and Processor Generation

Finally, when the target processor architecture is fixed, a description of the structure of the processor is generated in a hardware description language (HDL). The final bit image of the program to be executed by the processor has to be generated from the parallel TTA assembly code as well. The program image is ready to be loaded to the instruction memory of the target TTA processor. In addition to program image, bit images of the initial data memory contents are generated as well.

This specific phase of the design flow was implemented in this thesis. The Processor Generator (ProGe) does the processor generation and the Program Image Generator (PIG) generates the bit image of the program to be executed by the processor by means of the given encoding rules as well as the data images.

MOVE framework had corresponding tools for program image and processor generation. Those tools set the basis for the tools implemented in this thesis, naturally. However, the TCE versions are more advanced and they are designed from scratch. The processor generator of MOVE framework is covered in [4].

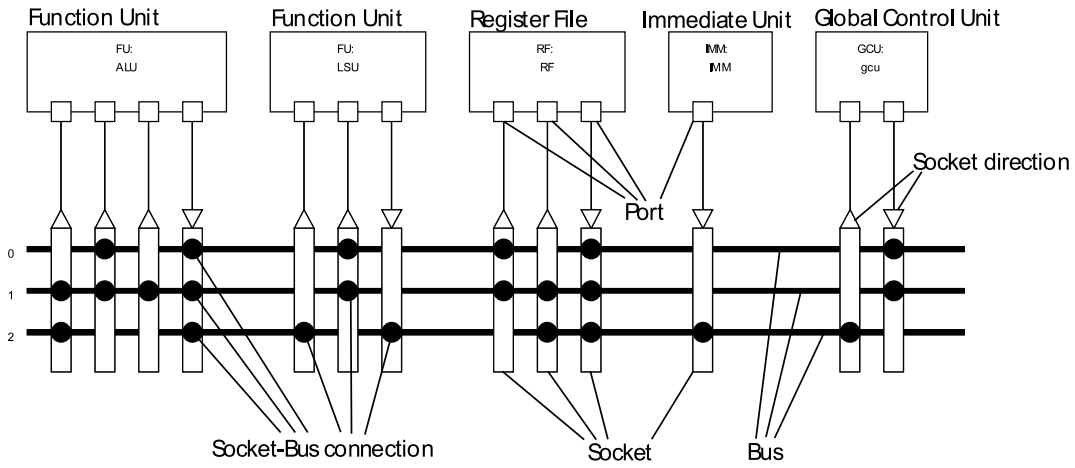

Figure 3.1: Simple TCE TTA processor template.

3.2 Processor Architecture Template

TCE has an architecture template for representing supported TTA processors. Architecture defitions are stored in file called Architecture Definition File (ADF) [5], which is a text file in Extensible Markup Language (XML) format. ADF defines the architectural properties that affect the behaviour of the processor, and thereby the program it can execute. It does not specify the internal implementation of the processor.

ADF is always loaded to Machine Object Model (MOM) before it is used by software modules in TCE. MOM is an object model written in C++, which represents the architecture of the processor and provides interfaces that are rather easy to use by software clients.

Figure 3.1 depicts the TTA processor template supported by TCE. The different processor components are discussed in the following subsections.

3.2.1 Interconnection Network

Interconnection network consists of buses and sockets. Sockets are connected to the buses, which just transport the data. There are two kinds of sockets: input and output sockets. Output sockets put data on the bus and input sockets read the data. In addition to buses, sockets are connected to ports of different units. Output sockets connect to output ports and input sockets to input ports, respectively. These connections form paths from output ports to input ports enabling data transports between them. Of course everything is customisable: the number of buses, width of buses, number of sockets, socket-bus and socket-port connections. Sockets can also be connected to several ports.

3.2.2 Function Unit

Function units provide the data processing capability of the processor. They have at least one input port connected to the interconnection network. Input ports of function units are called operand ports, and output ports result ports. Usually, function units implement basic operations, like add, subtract, multiply, shift etc. but the operations can be however complicated. They get input data from operand ports and the result is written to result port. Some function units do not have result ports at all. Their purpose is to provide interface to some external device, such as a memory block or an LCD display, for example.

3.2.3 Register File

Register files are units that contain registers to store data temporarily. Each register has the same word width, but the width is customisable, or course, as well as the number of registers. The number of input and output, thus read and write ports, can be changed too.

3.2.4 Immediate Unit

Immediate unit is a specialised register file that is written by control unit only. Control unit writes *long immediates* to immediate unit during instruction decoding. From immediate unit, they can be moved to another unit via interconnection network. The data is not transferred to immediate unit via interconnection network but directly from control unit using a dedicated data bus.

3.2.5 Global Control Unit

Global control unit, briefly control unit, is a specialised function unit which usually have at least operations *call* and *jump*. *Call* performs function call and *jump* jumps to different location in the program. GCU controls the processor by generating control signals for units and interconnection network. It fetches instructions and decodes them before generating the control signals. ADF defines the number of transport stages of the pipeline of GCU. This affects the latency of jump and call to take effect, since the number of cycles taken from fetching an instruction to executing it depends on the number of pipeline stages.

3.3 Binary Encoding Map

Binary Encoding Map (BEM) defines the instruction format and binary encodings of instructions of a certain TTA processor architecture. BEM is necessary when generating the program image and when decoding the instruction words. BEM file

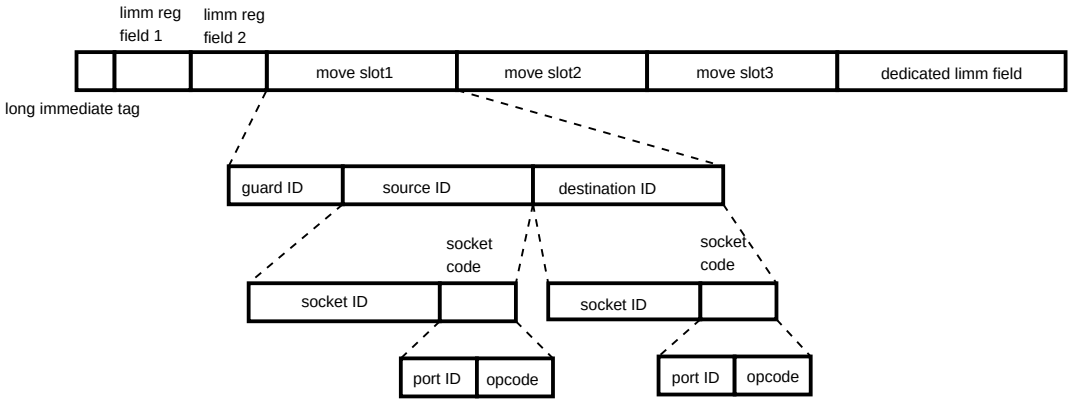

Figure 3.2: TTA instruction format in TCE.

is an XML file, of which format is specified in [6, s.15-21]. Also BEM is loaded to an object model before used by software clients. TCE has an automatic tool which can generate a BEM file for the given architecture by minimising the width of the instruction.

The instruction supported by TCE is depicted in Figure 3.2. The instruction consists of *move slots*, *dedicated long immediate fields*, *immediate control field* and *long immediate destination register fields*. The order of fields is compeletely customisable, Figure 3.2 just depicts one possibility.

3.3.1 Move Slot

TTA instruction has a move slot for each data bus in the processor. Move slot of a bus encodes the data transport on that particular bus. The slot consists of *guard*, *source* and *destination* fields. Source field defines the source of the data transport and destination field the destination, respectively.

Both the source and destination field consists of *socket ID* and optional *socket code*, which is further divided to *port ID* and *opcode*. The *socket ID* field defines the source or destination socket of the move. Naturally, each socket in the interconnection network has a different ID. In addition to socket ID, separate *socket code* is needed if the socket is connected to several ports or if opcode must be transmitted too. *Port ID* field defines the port if there are several ports connected to the socket and *opcode* field gives the opcode. Opcode defines the register number if reading or writing a register file. Opcode is also used to select the operation to trigger in a function unit if there are several possibilities.

The guard field is used if conditional execution is wanted. The guard term, identified by the guard field, may refer to a register of a register file or a result port

of a function unit. If the encoding in guard field refers to a register, the value in the register determines, whether the data move is performed or not. Respectively, the guard term may be taken from a result port value. The guard terms supported are defined in ADF separately for each data bus.

3.3.2 Dedicated Long Immediate Field

Dedicated long immediate field is used to encode a part of *long immediate*. The number of decicated long immediate fields is unlimited. See 3.4 for more information about long and short immediates.

3.3.3 Long Immediate Tag

The long immediate tag defines the instruction template of the instruction. Basically, it tells what move slots and dedicated long immediate fields are used for encoding long immediates in the instruction. That information is required as the instruction is being decoded. Otherwise the instruction decoder would interpret all the move slots as normal data moves.

3.3.4 Long Immediate Destination Register Field

The purpose of long immediate destination register field is to define the register number of immediate unit to which the long immediate is to be written. There are as many long immediate destination register fields as is the maximum number of long immediates per instruction.

3.4 Immediate Support

TCE TTA template supports two kind of immediates in program code: *short immediates* and *long immediates*. They are encoded and handled in the processor in totally different way.

Short immediate is encoded in the source field of move slot. Instruction decoder extracts the immediate value from the instruction word and puts it on the data bus of which move slot it is encoded in. Respectively, the destination field of the move slot defines the destination of the immediate value.

If large immediate values must be supported, the instruction word would become very long if they were encoded in the source field. For this reason, they are better to encode as *long immediates*. They are encoded in several move slots of an instruction. For example, if we have a 36 bit immediate to encode, it can take 12 bits of three move slots. This implies of course that data transports are impossible on the corresponding buses in that instruction.

Instruction decoder extracts the portions of long immediate from the instruction word, combines them, and stores to an immediate unit. The immediate value can be read from the immediate unit as a normal data move later.

3.5 TTA Program Exchange Format

In TCE, sequential and parallel TTA programs are represented in format called TTA Program Exhange Format (TPEF) [7]. TPEF file is in binary format but it is loaded to C++ object model before used by client modules.

3.6 Hardware Database

Hardware Database (HDB) is a relational database which contains information about predefined hardware implementations of function units and register files. HDB contains cost data for estimating processor costs and implementation specific data, such as port names and opcodes, required by the Processor Generator. Complete specification of the database is in [8].

SQLite [9] is used as database management system. Database is totally wrapped within HDBManager class to make it easier to use for clients and to ensure that changing the database management system in the future is possible.

4. PROGRAM IMAGE GENERATOR

Program Image Generator (PIG) is an application in TCE toolset, that generates the complete bit image of a TTA program, as mentioned in 3.1.4. An example to this application was the program image generator of MOVE framework. As usual, the TCE version is a little bit more advanced, however. The MOVE version does support neither code compression 4.3.3 nor instruction relocation 4.3.1.

This chapter introduces what is meant by program image and some concepts relating to memory blocks, at first. It continues by discussing the requirements set to the application. Operational principles of PIG are also introduced, and finally some of the most important implementation details are covered. Figure 4.1 depicts the inputs and outputs of the application.

4.1 Program Image

In this thesis, the concept of program image means bit-level representation of a TTA program. The program image is ready to be executed by the TTA processor hardware it is targeted to. The program image consists of instructions, which might be compressed. Program image is generated using encoding rules defined in BEM. Instruction decoder uses the same encoding rules as it decodes the instructions.

4.2 Memory Concepts

It is important to be familiar with three concepts relating to memories in order to understand the following sections. Those are *memory width*, *memory location* and *minimum addressable unit* (MAU).

- *Memory width* is the width of the word that can be read from the memory at once. In the pictures describing memory contents, the number of bits in each row is the same as the memory width. That is, the memory can be read one row at once.
- *Memory location* is a location in the memory which is referred to by a memory address. The width of a memory location is the minimum addressable unit.
- *Minimum addressable unit* is the number of bits covered by a memory location. Memory width consists of one or more MAUs.

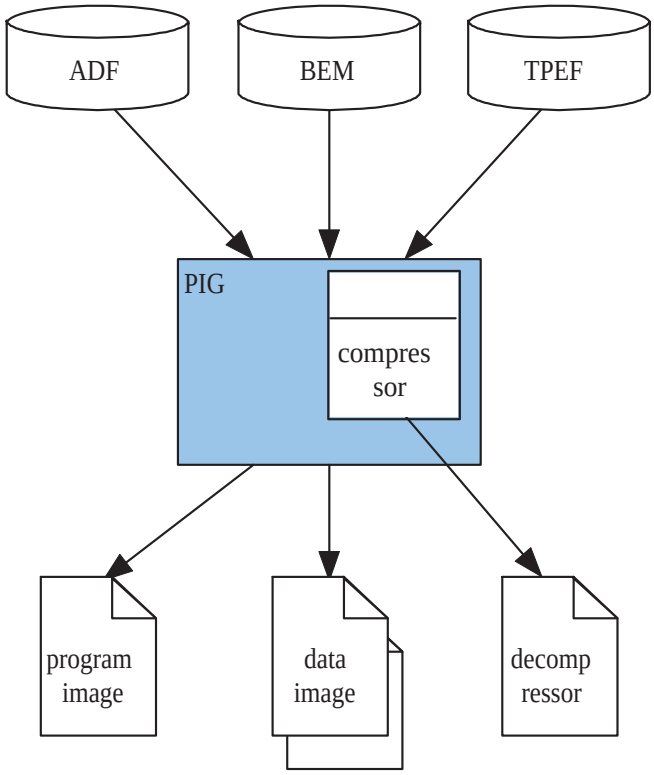

Figure 4.1: Data flow diagram of PIG.

4.3 Requirements

The software development was started by defining requirements for PIG. Some of the requirements came from the experiences gained from the program image generator of MOVE framework. Since TCE is a little bit more sophisticated system, more demands were made on PIG. Some of them were realised when the software was already implemented, but thanks to well designed software architecture, new requirements were quite easy to implement later on.

4.3.1 Instruction Relocation

One of the challenges in generating the program image is instruction relocation. It means that instruction addresses are fixed during image generation since the real addresses depend on target instruction memory layout, instruction widths and instruction positioning method used. These issues are described in subsection 4.4.2.

Jump and *call* addresses must be changed due to instruction relocation. They might be encoded as short or long immediates in the code, or in variables stored in the data memory. PIG would not know which of the immediate or data values should be changed, but the information of values referencing to instruction addresses is given in *reloc sections* of TPEF.

4.3.2 Generation of Data Images

In addition to the program image, PIG must generate also data images. It generates separate image files for each address space defined in the ADF. The contents of data memories are given in TPEF byte by byte, so generating the image is relatively simple. However, due to instruction relocation, the values referencing to instruction addresses must be fixed in data memory too.

4.3.3 User Definable Code Compression Schemes

PIG must support code compression. There are several ways to compress the program code, and new algorithms can be invented, so that TCE users must be allowed to develop code compression algorithms of their own. PIG must also provide a framework, which relieves the development work of code compressor modules.

4.3.4 Multiple Input Programs

This requirement was realised afterwards, when the first version of the application was already implemented. PIG must be able to generate program images of several input programs at once. The feature is useful if the code must be compressed. The compression algorithm gets all the input programs at once, which may result in better compression rate compared to if each program were compressed separately. Another reason is that all the programs must be compressed with same compression data. For example in case of dictionary compression, all the programs must be compressed using the same dictionary. Otherwise they could not be executed in the same processor.

4.3.5 Decompressor Generation

Since PIG can compress the program code, it must generate a decompressor which can decompress it. Basically, this is a requirement for user defined code compressor modules. The decompressor must be generated in HDL, so that it can be taken in use when the processor hardware is generated. The interface of the hardware block is well-defined. See 5.2 for detailed interface definitions.

4.3.6 Different Output Formats

The program and data images are needed in different formats whatever the case may be. PIG must support different output formats which are: binary, ASCII binary and VHDL array form. Binary format is what is loaded to the instruction or data memory of the processor. The ASCII binary format means an ASCII output containing characters '1' and '0'. It is better human readable than binary and can

be used for testing and debugging purposes. The VHDL array form is almost similar to the ASCII binary format. The only difference is that each row is enveloped by '"' marks and each row ends with ','. VHDL arrays can be initialised with this format easily.

4.4 Operational Principles

In a simplified manner, the operation of PIG has four main stages: reading the input files, generating the program image, writing the program image to file and writing the data images. Reading the input files is actually done by other software modules. Generating the program image is the main task, and the most challenging of course. It is generated instruction by instruction, by generating a non-compressed instruction at first and possibly compressing it. The operational principles are discussed in the following subsections.

4.4.1 Input Files

PIG needs ADF, BEM and TPEF as input data. The program is given in TPEF and BEM defines the format of the instructions to generate. ADF is needed too, because BEM and TPEF formats are not interrelated. TPEF defines data moves in higher abstraction level than BEM. For example, TPEF does not tell what sockets must be used to make a data move but BEM defines the instruction format by means of socket IDs. That is, information of the processor architecture is necessary.

4.4.2 Program Image Generation

The program image is generated as a bit vector at first. That is, it is represented as a very long array of bits. The instructions are added one by one, starting from the first instruction of the program and appending new instructions to then end of the bit vector. Instruction bits are generated by user defined code compressor and they are given to PIG framework which adds them to the program bit vector.

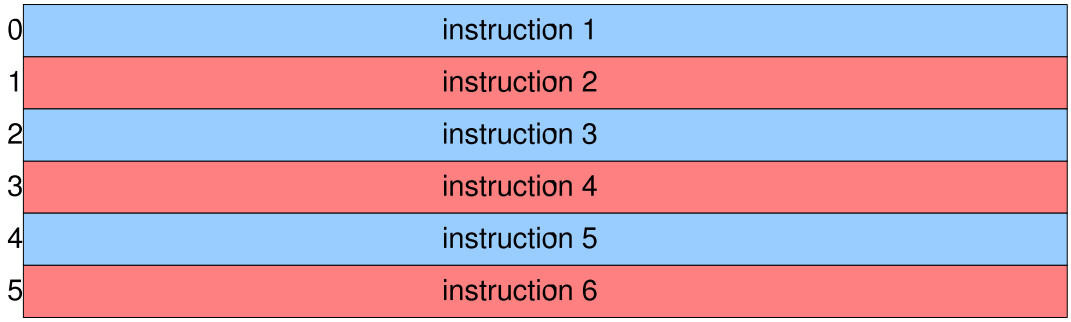

Figure 4.2: Fixed instruction width. One instruction per memory location.

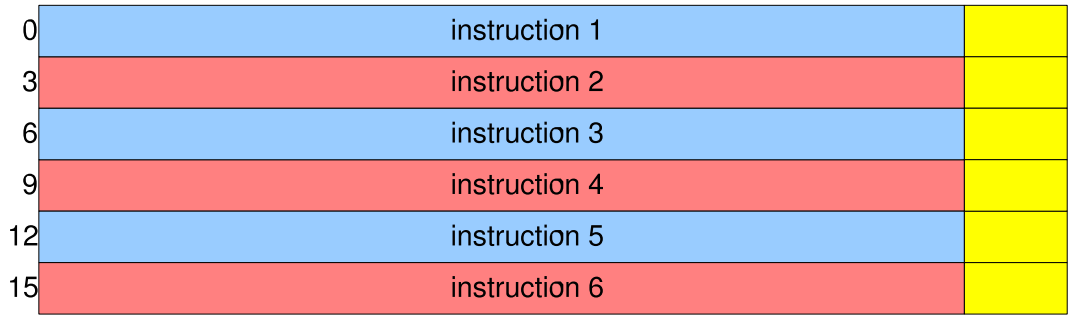

Figure 4.3: Fixed instruction width. One instruction takes 3 memory locations. Each instruction starts at the beginning of memory location.

Depending on the instruction memory layout, the instruction bits can be added to the program bit vector in two ways. In basic case, each instruction starts at the beginning of a memory location, see Figures 4.2 and 4.3. However, if variable width instructions are used, it is inevitable that the width of the instruction is not always a multifold of MAU. In that case, better memory allocation is achieved if the instructions are one after the other without any padding bits between them, see Figure 4.4. This results that all the instructions do not start at the beginning of a memory location, which makes instruction fetching more difficult. In general, at least jump target instructions must start at the beginning of a memory location. The code compressor tells the PIG framework, which instructions must start at the beginning of memory slot, so that PIG can position them correctly.

4.4.3 Bookkeeping of Instruction Addresses

Before starting to generate the program image, PIG inspects the *reloc sections* of TPEF and generates its internal data structures which contain the information of the immediates that refer to some instruction address. For each immediate, there is information which instruction it refers to.

PIG keeps track of final memory addresses of each instructions, so that the immediate and function pointer values in the data memory can be fixed. Each time an instruction is added to the program image, PIG inserts the address of the instruc-

Figure 4.4: Continuous memory fill with variable width instructions. Instruction 6 set to start at the beginning of memory location.

tion added, to its internal map structure, and fixes the referencing immediate values of the instructions added earlier. In addition, if the instruction added contains an immediate that refers to an instruction which is already added, the immediate value is replaced with the correct value.

To make this all possible, the immediates that must be fixed are marked by the code compressor. Since the instructions obtained from code compressor are compressed, their format unknown to PIG. For this reason, code compressor simply marks the bits that represents an immediate. When the instruction bits are given to PIG framework, it checks the marked bits and possibly replaces them.

4.4.4 Data Image Generation

Data images are generated by PIG itself without the aid of code compressors. The program image must be generated before data images, because final instruction addresses must be known to be able to fix function pointer values and other instruction address references in the data memory. Data memory contents are obtained from TPEF directly. PIG just needs to do the fixing of instruction references and generate the bit vector of data memory contents.

4.4.5 Writing the Output

When the bit vectors of instruction and data memories are created, they are finally written to output files of the requested format. Binary format is easy to create, since there is no need to care about line breaks and other styling issues. However, in ASCII formats, the output is broken into lines, which must be of sensible width.

Currently, each instruction is written on different line in ASCII formats. Data memory image is broken into lines by writing the given number of memory locations per line. The user can define how many memory locations are written per line by a command line parameter.

4.5 Implementation

The software architecture of the PIG is divided into three subsystems: *PIG core*, *base library* and *user interface*. The subsystems and their main modules are depicted in Figure 4.5.

PIG core is the main module, in which the program and data image generation occurs. The core provides an interface that can be exploited by different user interfaces. It uses the *base library* which is commonly used in TCE applications. It is not designed just for PIG, but also used elsewhere in the toolset.

Base library is a collection of modules that represent the major consepts of TTA. Base library contains also other modules than the ones used by PIG, which are

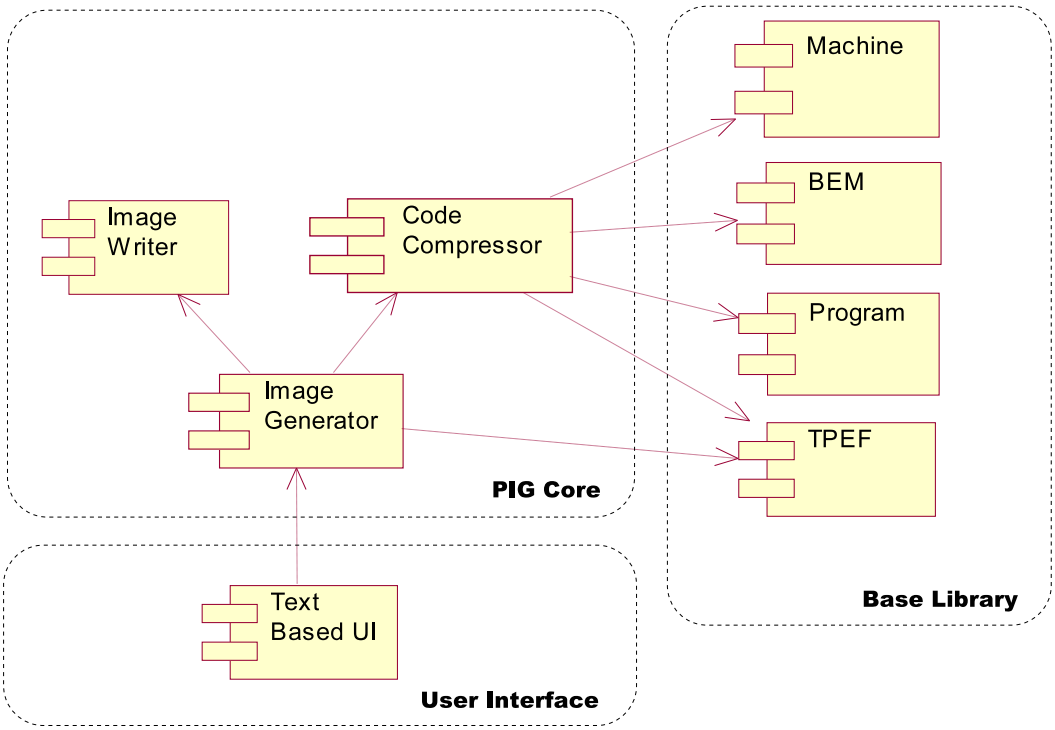

Figure 4.5: Main components of PIG.

TPEF Handling Module, *Program Object Model*, *Binary Encoding Map* and *Machine Object Model*. TPEF handling module is basically an object model representation of TPEF file. The interface it provides is not very user friendly and, therefore, it is converted to *Program Object Model* (POM) before extensive use in PIG. PIG uses the *TPEF Handling Module* as the reader of TPEF files. Also data sections are read from the *TPEF handling module* directly, since they are not included in POM. The *reloc sections* of TPEF are used too to resolve the instruction references. POM is a static higher level representation of TTA programs, in which processor resources are assigned to instruction elements, such as move source and destination terminals. Basically, PIG would not need POM. TPEF contains all the same information but POM is created because it is a lot easier to use. The *Machine Object Model* (MOM) represents the processor architecture read from an ADF and the *Binary Encoding Map* is an object model representation of a BEM file.

4.5.1 InstructionBitVector Class

InstructionBitVector class is used to represent the bit string of instructions. One *InstructionBitVector* instance may contain just one instruction or several ones, that is, complete program image can be represented as an *InstructionBitVector* instance too. Since TTA programs may be quite long, memory usage have to be taken into account when handling them. *InstructionBitVector* class is derived from the *vec-*

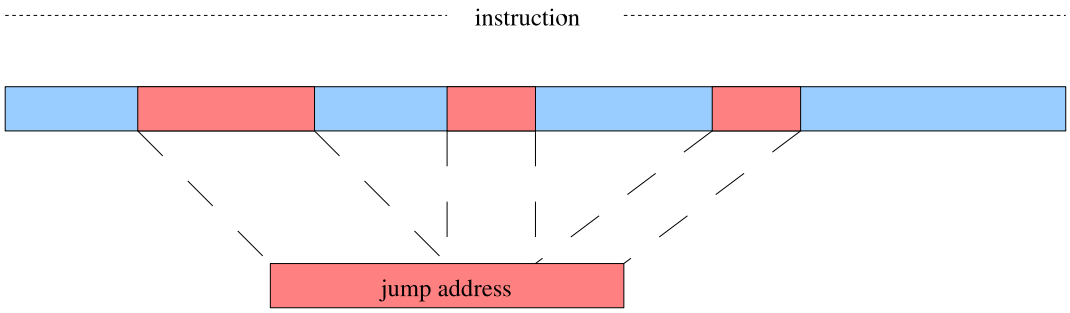

Figure 4.6: Instruction containing an immediate in three parts.

tor<bool> type of C++ standard template library (STL), so it is highly optimised. Each bit takes just one bit of memory.

The class has two special features. It can keep track of instruction boundaries if there are several instructions in the vector. They can be set by *setInstruction-Boundary()* method. In order to make correct line breaks, instruction boundaries must be known when writing the image in ASCII format to a file. The other special feature is that it can store instruction address references. That is, there can be sequences of bits that, either themselves, or combined with other sequences reference to some instruction addresses. Figure 4.6 depicts a case in which there are three bit sequences that represent an instruction address together. An instruction address reference can be created by calling *startSettingInstructionReference()* at first, and then setting the boundaries by *addIndexBoundsForReference()* method. The former method takes the *Instruction* which is referenced, as parameter, and the second takes vector indices for starting and ending point.

The idea of setting the instruction address references is that *InstructionBitVector* can automatically fix the values later, when the real addresses of the instructions referenced are known. When *fixInstructionAddress()* is called, the object automatically modifies the bit sequences referencing to the given instruction. This feature is used by PIG as the instruction relocation is performed.

4.5.2 Code Compressor Framework

The code compressors are implemented as dynamic modules which can be loaded at run time. The framework consists of an abstract base class from which the actual code compressor has to be derived. Figure 4.7 depicts the class diagram of the framework. The base class called *CodeCompressorPlugin* contains three pure virtual methods which must be implemented in each code compressor plugin, and several protected methods that are useful for code compressors.

One of the pure virtual methods is *compress()* which returns the complete compressed program image as a bit vector. The program to be compressed is

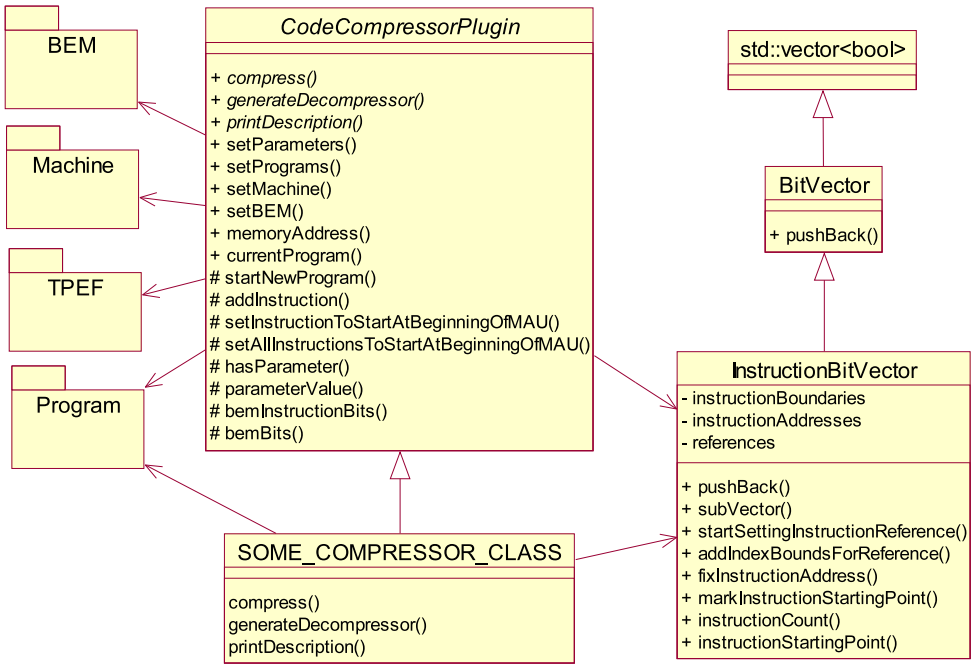

Figure 4.7: Class diagram of the code compressor framework.

given as parameter as *TPEF Handling Module*. Generating the program image is started by calling *CodeCompressorPlugin::startNewProgram()* for which the TPEF module is given. It converts the given TPEF module to POM, which can be obtained by *CodeCompressorPlugin::currentProgram()*. Then, all the instructions that must start at the beginning of memory location (at least jump targets) must be told to the framework by calling *CodeCompressorPlugin::setInstructionToStartAtBeginningOfMAU()* or alternatively, *CodeCompressorPlugin::setAllInstructionsToStartAtBeginningOfMAU()*. The program image is generated by calling *CodeCompressorPlugin::addInstruction()* sequentially for each instruction starting from the first one. The method takes *InstructionBitVector* and *Instruction* objects as parameter. The given *InstructionBitVector* object contains the bits of the compressed instruction and the immediates referencing to instructions addresses must be marked, in order to perform instruction relocation. When the instructions are added by *CodeCompressorPlugin::addInstruction()*, the framework adds the padding bits to the program image automatically, if required. It also keeps track of the memory addresses of the instructions added.

The framework provides also some help to produce the bit vectors of instructions. The *CodeCompressorPlugin* class has *bemInstructionBits()* method which returns uncompressed bits of the given instruction. The uncompressed bits are generated according to the instruction encoding rules defined in BEM.

Another pure virtual method is *generateDecompressor()* which generates the HDL

codes of the decompressor block. Currently, since the processor generator supports only VHDL, the decompressor blocks must be generated in VHDL, in order to be compatible with ProGe. An output stream is simply given to the method, which prints the HDL codes of the decompressor to the stream. The decompressor must have a certain interface which is discussed in Section 5.2.

printDescription() is the third pure virtual method that must be implemented in the code compressor. It prints a description of the plugin to the given stream. One of the features of the *generatebits* application is to show information of the code compressor plugins found. The text generated by this method will be shown.

4.5.3 Dictionary Compressor

For testing the code compressor framework and as an example to future compression experiments, a simple dictionary compressor [10] was implemented. The compressor compresses the instructions at level of the whole instruction, which results in minimal instruction length. Dictionary compression could be also done at the level of move slots, which means each move slot had a dictionary of its own.

4.5.4 Image Writers

The program and data images are written to files by different image writers. They take the bit vector representing the program or data image as an input and print the output to a file.

Different image writers are derived from base class called *BitImageWriter*. The inheritance hierarchy is shown in Figure 4.8. The base class defines a pure virtual method *writeImage()* which should write the image to the given output stream. Currently, there are five different image writers: *RawImageWriter*, *AsciiImageWriter*, *AsciiProgramImageWriter*, *ArrayImageWriter* and *AsciiProgramImageWriter*.

The *RawImageWriter* writes the given bit vector to the given stream in binary format. It is the simplest of the image writers because it does not need to care about line breaks. It just takes 8-bit sequences from the bit vector, converts them to *char* type, and writes the characters to the output stream.

The *AsciiImageWriter* writes the bit vector in ASCII format using '1' and '0' characters. Width of the lines it writes is given in its constructor. Thus, each line has the same width. An example output of *AsciiImageWriter* could be:

```
0110010010010011
1001011001010010
1001001010111000
...
```

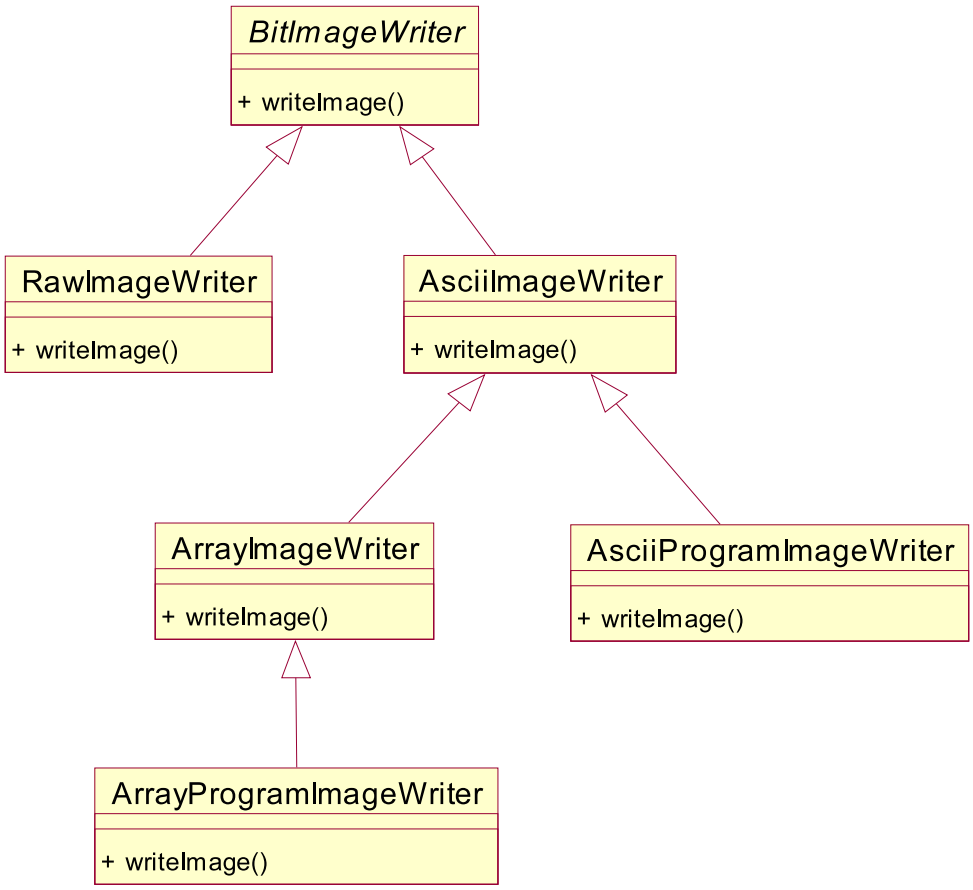

Figure 4.8: Class diagram of image writers.

The *AsciiProgramImageWriter* is a more advanced version of the *AsciiImageWriter* that takes into account the width of the instructions. It breaks the line after each instruction, such that there is one instruction per line, even if the instruction width varies. The *AsciiProgramImageWriter* makes good use of *InstructionBitVector* class, which stores the instruction boundary points.

The *ArrayImageWriter* is derived from *AsciiImageWriter* class which provides also protected helper methods for ASCII writers. The output of *ArrayImageWriter* is similar to *AsciiImageWriter* except that each line is in quotation marks like this:

```
"0110010010010011",
"1001011001010010",
"1001001010111000",
 ...
"0110101010111000"
```

This format is designed for initialising a VHDL array. The output can be copied

and pasted to a VHDL file, in which the array is needs to be initialised. The *ArrayProgramImageWriter* is similar to the *ArrayImageWriter* except that it exploits the instruction boundaries and prints one instruction per line.

4.5.5 User Interface

The user interface is a separate module in PIG design. Currently, PIG can be used only through command line, but in the future, there will probably be a graphical user interface for PIG. The task of command line user interface in PIG is to parse the given command line parameters and create object models of the given ADF, TPEF and BEM files. In TCE, we have a toolkit for parsing command line parameters and the base library contains parsers for given input files, so the user interface itself is very simple. Once the object models are created, the user interface passes them and other parameters to PIG core and it takes responsibility of the rest.

5. PROCESSOR GENERATOR

The other application covered by this thesis, Processor Generator (ProGe), is used to generated synthesisable VHDL definition of the processor designed with TCE tools. This is the last stage in the design flow.

The input files ProGe needs are ADF, BEM, and IDF. In addition, if code compression is used, ProGe needs the decompressor module generated by the compressor plugin of PIG. Figure 5.1 depicts the inputs and outputs of the application.

This chapter concentrates on the ProGe, discussing about the requirements set to the application at first. In Section 5.2 the module division of the processor template is introduced and Section 5.3 covers the most important operational principles of the processor hardware. The software implementation of the application, including class diagrams of the main modules and sequence diagrams of the operation, is described in Section 5.4. Finally, some restrictions of the current implementation are listed in Section 5.5.

5.1 Requirements

The development process of ProGe was started by setting the functional requirements for the application. The HDB was developed in parallel with ProGe, as it was known, what information must be contained in it. The main functional requirements of ProGe are listed here. For thorough requirement document, see [6].

5.1.1 Easy Extendability

At first, it was thought whether ProGe should support other languages in addition to VHDL. We decided it is enough that the initial version supports just VHDL, but is must be designed to allow extending to other languages later. That is, it must be kept totally free of HDL dependencies as long as possible. Just the final stage when writing the output is HDL dependent.

5.1.2 User Definable Implementation of Decoder and IC

A fundamental requirement of ProGe is support of custom, user-defined implementations of interconnection network and instruction decoder. A predefined implementation alternative is provided by TCE.

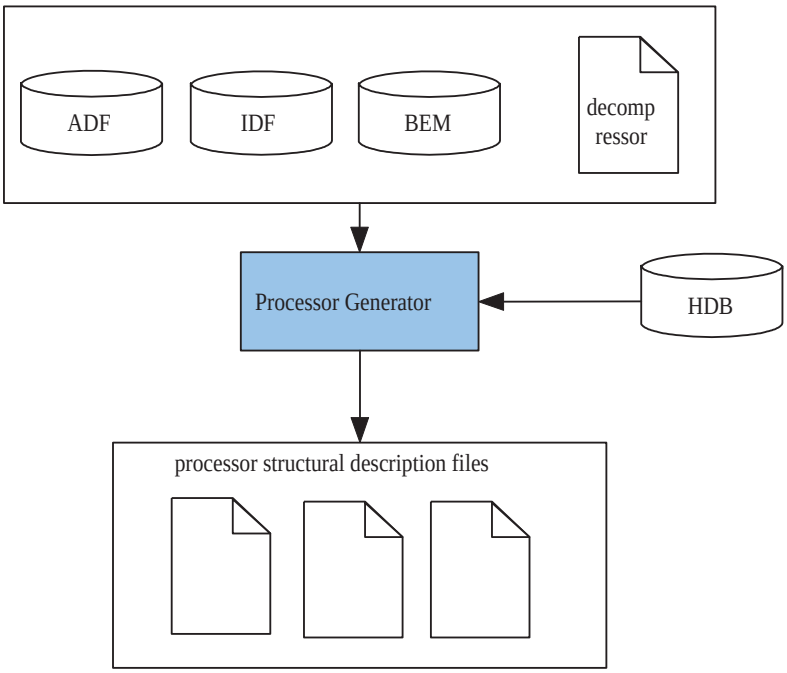

Figure 5.1: Data flow diagram of processor generation.

5.1.3 User Definable RF and FU Implementations

The main building blocks of a target processor, such as function units, register files and immediate units, are too complex to be generated automatically by ProGe. Instead they are generated off-line by the user and stored in HDB. The IDF file given to ProGe tells what blocks should be selected from the HDB.

5.2 Hardware Modules of Processor Template

The processor generated by ProGe has a well-defined module division. Some of the hardware modules are predefined library components described in VHDL, and the others are generated by ProGe. The modules and their interfaces are depicted in Figure 5.2.

The control unit of the processor comprises three modules: *instruction fetch unit*, *instruction decompressor* and *instruction decoder*. Interfaces between them are designed carefully to open the way for different code compression and instruction memory schemes, as shown in Figure 5.2.

Function units, register files and immediate units are library components. They are picked from HDBs, as defined in IDF. Generating them online would be too difficult, if not impossible. They are connected to the decoder for control signals and to interconnection network for data transfer.

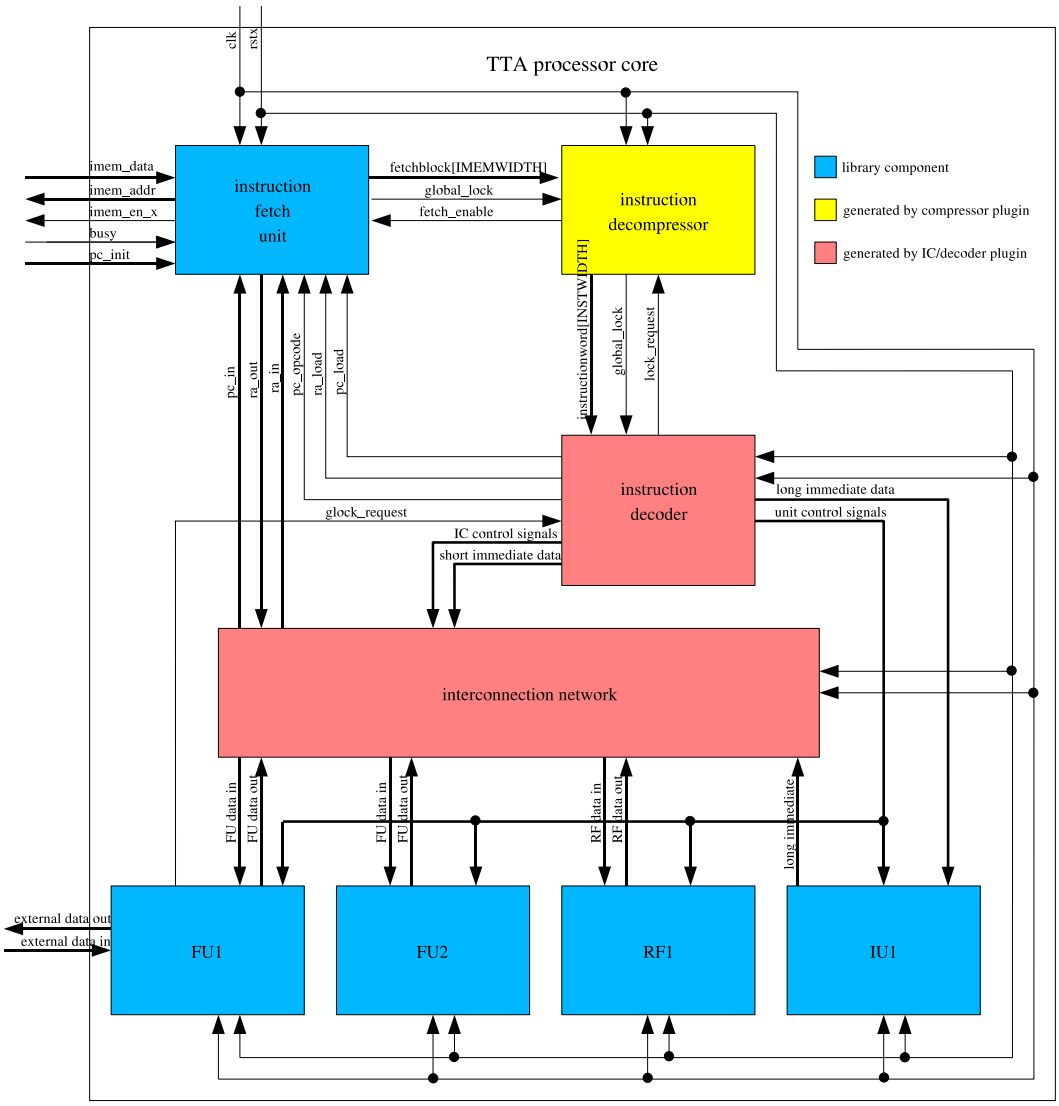

Figure 5.2: Module division of the processor.

5.2.1 Instruction Fetch Unit

The structure of instruction fetch unit is always similar in every processor, so it is a library component. However, it is parameterized by three factors which are width of the address line of instruction memory, MAU of the instruction memory and the width of the instruction memory in MAUs.

5.2.2 Instruction Decompressor

Instruction decompressor obtains fetch blocks from instruction fetch unit, decompresses them and passes uncompressed instructions to instruction decoder. Decompressor is generated by the compressor plugin of PIG. It must communicate with instruction fetch unit and instruction decoder in a particular way. It requests a

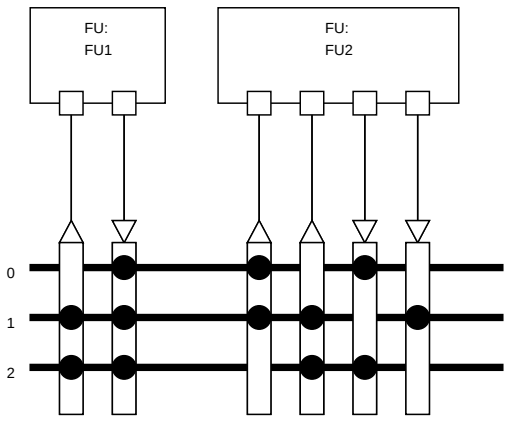

Figure 5.3: An example of interconnection network.

new fetch block by enabling the *fetch_ en* signal to the fetch unit. A fetch block may contain several compressed instructions which implies that instructions are not going to be fetched in each cycle. Another task of decompressor is to pass lock signals. If instruction fetch fails for some reason, instruction fetch unit generates *global lock* signal which must be propagated to instruction decoder by decompressor. Respectively, some function unit may request global lock which is propagated to decompressor by decoder. In this case, decompressor has to propagate it to instruction fetch unit.

5.2.3 Instruction Decoder

Instruction decoder, as well as interconnection network modules are generated by the IC/decoder plugin of ProGe. This part of the processor is totally dependent on the plugin used. ProGe itself does not restrict is anyhow but the decompressor-decoder and fetch unit-decoder interfaces are fixed by the design. Instruction decoder gets an uncompressed instruction from decompressor and generates the processor control signals. Like decompressor, decoder has to propagate lock signals too. Lock request coming from decompressor must be propagated to all the operational units. Also global lock request coming from a function unit must be propagated to other units and decompressor.

5.2.4 Interconnection Network

Interconnection network consists of buses and sockets. They can be implemented in several ways, and the implementation depends on the IC/decoder plugin used. Interconnection network must have the same data paths as defined in the ADF given to ProGe. In addition, there are sockets that are invisible in ADF. Those are used

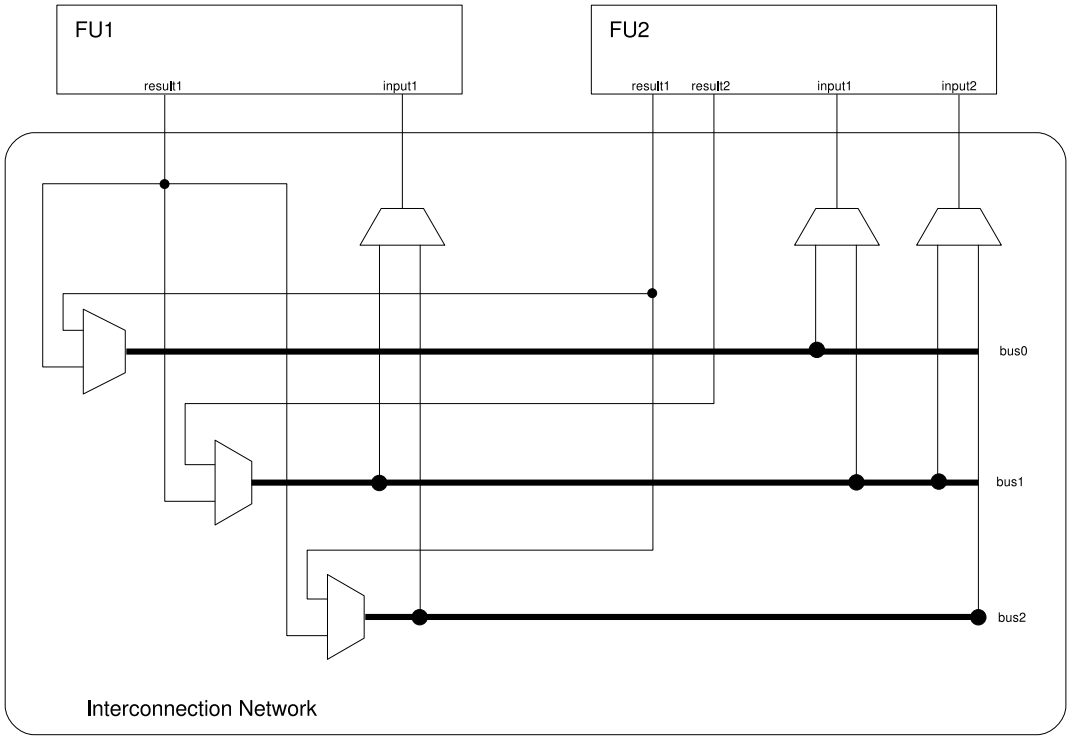

Figure 5.4: Datapath realised with interconnection of multiplexers.

to pass short immediates from instruction decoder. Figures 5.4 and 5.5 illustrate how the interconnection network can be implemented in hardware. The same IC

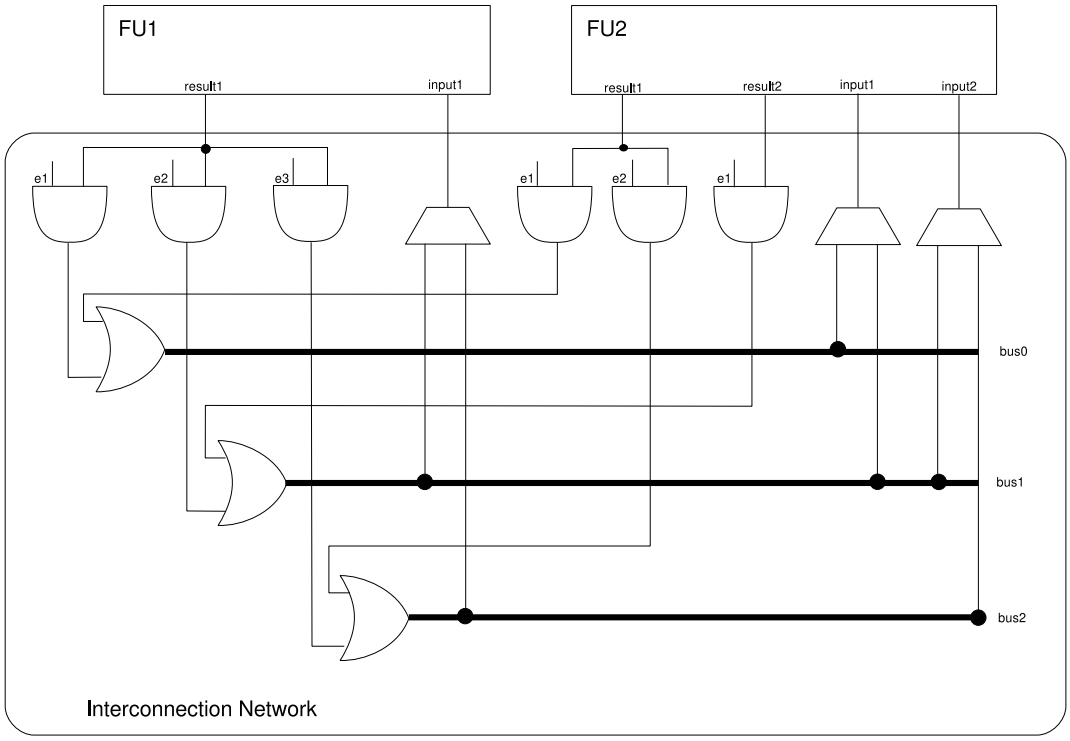

Figure 5.5: Datapath realised with AND-OR network.

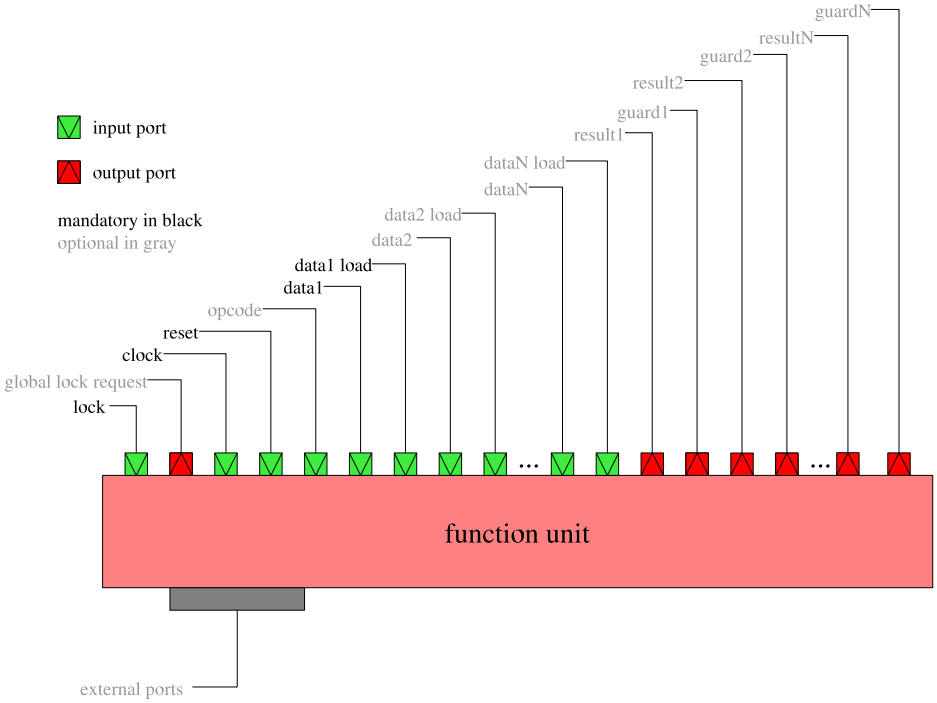

Figure 5.6: The interface of function unit model.

architecture is depicted in Figure 5.3.

5.2.5 Function Unit Model

Function units supported by ProGe must have specific control ports, in addition to data ports. Figure 5.6 depicts the interface of the model. For each input data port, there must be a so called *load port*. The load port is used to load value to the input register of the function unit. When the load is enabled, the value is read from the input socket to the input register. If the function unit supports more than one operation, it must have a port for operation code too. It is used to select the operation to execute. In addition to result ports, there may be a *guard* port for each result. Guard port is a 1-bit port of which value can be used to control conditional execution. Function unit must also have a *lock* port. If the processor has to be locked, each function unit must also be locked by using this port. Some function units may need to request locking of the whole processor. This is called *global lock request*. For example, a *load-store unit* may need to request lock, if it uses a cache memory and a cache miss occurs. Function units may have external ports that are not connected to anywhere in the processor core. In that way, external devices can be connected to the processor. For example, *load-store units* have external ports connected to the data memory. When the processor is generated, the external ports of the function units are added as external to the processor core, as shown in Figure 5.2.

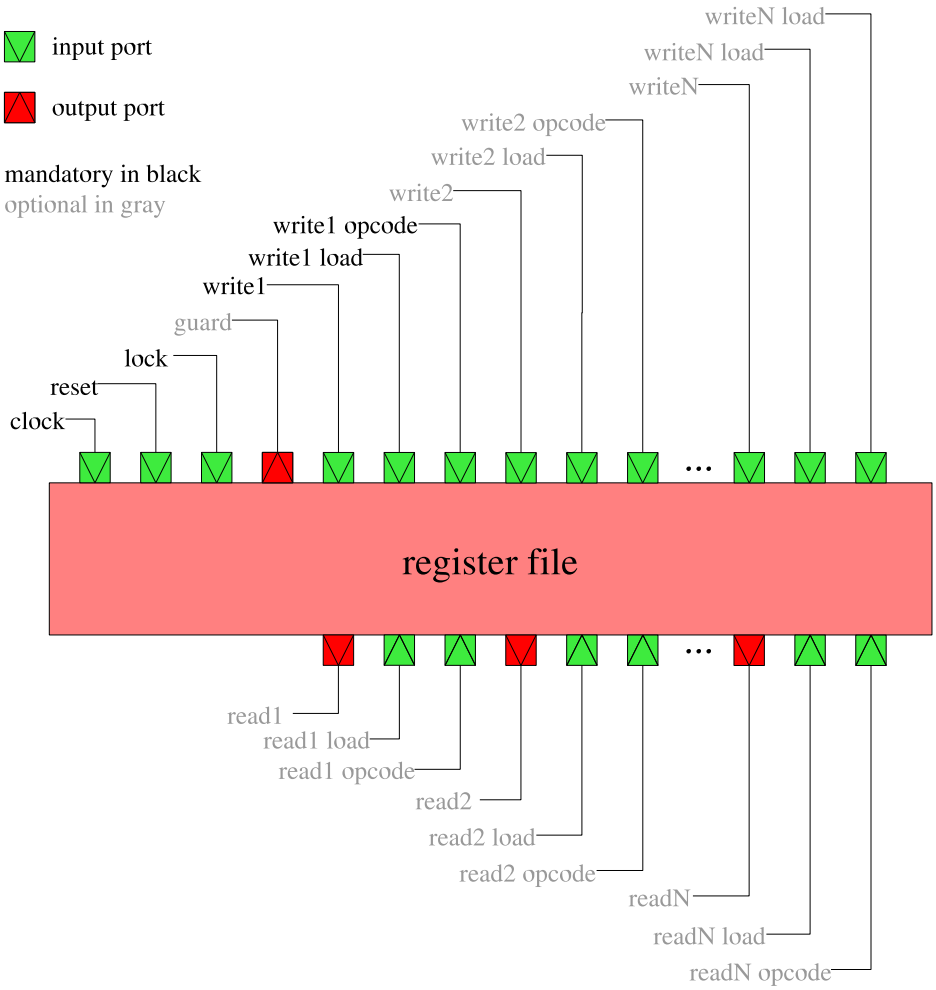

Figure 5.7: The interface of register file model.

5.2.6 Register File Model

Register files must be of a certain model too. The interface of the model is depicted in Figure 5.7. They have a variable number of data read and write ports. For each data port, they have also a load port and an operation code (opcode) port. The opcode port is used to select the register which is read or written. Thus, the width of the opcode port depends on the number of registers in the register file. The load port is needed to load data from write port to a register or to update the value of a read port. Register files may have a *guard* port as well. Its width is the same as the number of registers in the register file. Each bit represents guard value of a register in the register file. The guard value is taken by doing OR-operation for all the bit of the corresponding register. Like FUs, each RF must have a *lock* port, which is used to lock the RF, if the processor has to be locked.

5.3 Operational Principles

ProGe starts the job by generating a netlist of the processor. The netlist contains the processor building blocks and connections between them as shown in Figure 5.2. When the netlist is generated, it is written to a VHDL file by a writer module. It writes the processor core which basically just connects the processor building blocks to each other.

When the netlist is being generated, the function units, as well as the register files and immediate units given in IDF, are added there too. Names of the ports of those blocks must be known. That information is obtained from HDB. IDF tells exactly the FU, RF and IU implementations to select from HDB. In addition to port names, their purposes must be known. That is, data ports of units declared in ADF must be matched up to the ports of the corresponding implementations selected from HDB. For function units, the port matching is done on the grounds of operand bindings. For example, if operand 1 of operation *add* is bound to port *p1* in ADF, and the same operand is bound to port *o1data* in the block selected from HDB, it is obvious that *add* and *o1data* are corresponding ports. It must be noted that if there are several operations in the function unit, the operands of all the operations must be bound to the same ports in the function unit declared in ADF and in the block selected from HDB. Otherwise their architecture is different and they are not compatible.

The implementation of a block selected from HDB may be parameterized by bit width for example. The parameter values must be fixed too when the netlist is being generated. Parameter values are resolved by means of port widths. After port matching, we can see from ADF, what the widths of the ports in the hardware implementation should be. Then, by looking at the width formulas defined for the ports in HDB, the parameter values can be resolved.

Since the implementation of instruction decoder and interconnection network are generated by a plugin module, ProGe cannot generate the whole netlist itself. Instruction decoder and IC modules have different interface depending on the plugin used. Due to that, the netlist is generated in two phases: At first ProGe adds all the blocks except interconnection network to the netlist. Decoder is added too, but not the ports that are connected to the IC. After that, the plugin adds the IC block and required ports to the decoder, and makes the connections from IC to decoder and units.

5.4 Software Implementation

The application is implemented in C++ using object oriented programming. It makes use of machine object model, binary encoding map and implementation de-

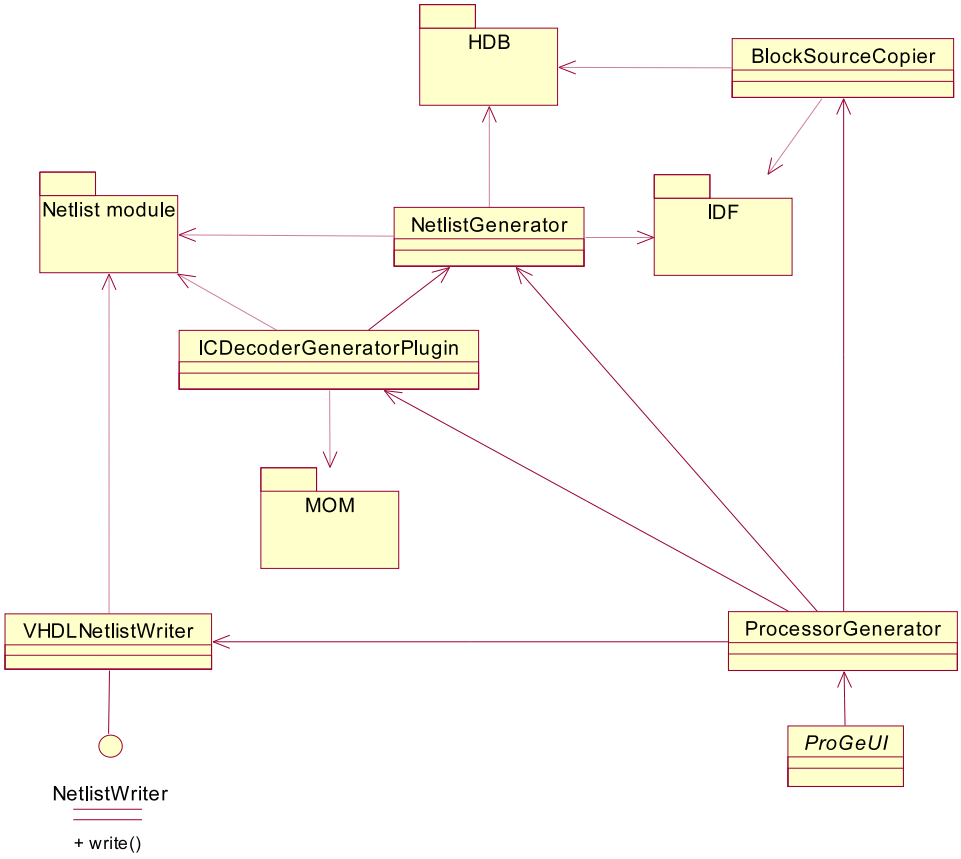

Figure 5.8: Class diagram of the main module of ProGe.

scription module from the base library of TCE. In addition, it accesses HDB through the HDB library of TCE. Figure 5.8 depicts the main class diagram of the application. More detailed class diagrams of the modules are shown in the corresponding subsections.

An overview of the operation of the application can be obtained from the sequence diagram shown in Figure 5.9. *ProcessorGenerator* is a kind of controller class. Generation of the processor is launched when a client (user interface) calls the *ProcessorGenerator::generateProcessor()* method. Before starting to generate the processor, some checks to verify the machine is sensible and generatable are performed. At first the processor is checked for common sanity in *ProcessorGenerator::validateMachine()*. It does some checks to verify the given processor architecture definition is sensible and can be generated in HDL. If this test is passed, the IC/decoder generator plugin makes its own checks to the processor in *ICDecoderGeneratorPlugin::verifyCompatibility()*. It must also be checked, that the register file implementations selected for immediate units, have correct latency. What is the correct latency, depends on the implementation of GCU and IC, thus on the IC/decoder

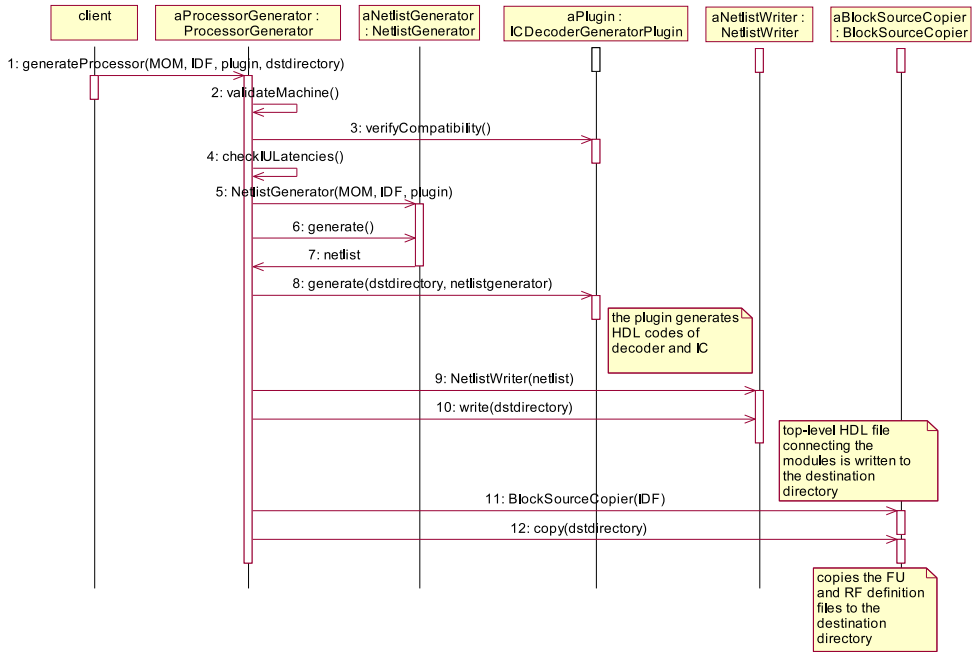

Figure 5.9: High-level sequence diagram of generating the processor.

generator plugin. These are checked in *ProcessorGenerator::checkIULatencies()* method. Now it should be sure the processor can be generated and is a working one. Then the netlist is generated by *NetlistGenerator*, and the HDL codes of IC and decoder by the IC/decoder generator plugin used. Finally, the top-level block that connects the hardware modules to each other according to the netlist, is written in HDL by a *NetlistWriter* instance. To finalize the processor definition, the files implementing the function units and register files must be copied to the output directory by a *BlockSourceCopier* instance.

5.4.1 Netlist

Netlist is represented as an undirected graph in which ports are vertices and connections are edges. The class diagram of the module is depicted in Figure 5.10. The *Netlist* class is derived from *boost::adjacency_list* which comes in the Boost Graph Library [11]. The ports are represented by *NetlistPort* class. They are never alone but always contained by a *NetlistBlock* instance. Respectively, *NetlistBlock* instances never exist alone - they are always contained by a *Netlist* instance.

The blocks in the netlist have a top-down hierarchy in which the blocks may have sub blocks. When the netlist is written in a HDL, for example in VHDL, each block in the netlist is written as an *entity*. Netlist representing a TTA processor has a block representing the processor core as the top-level block. It contains blocks

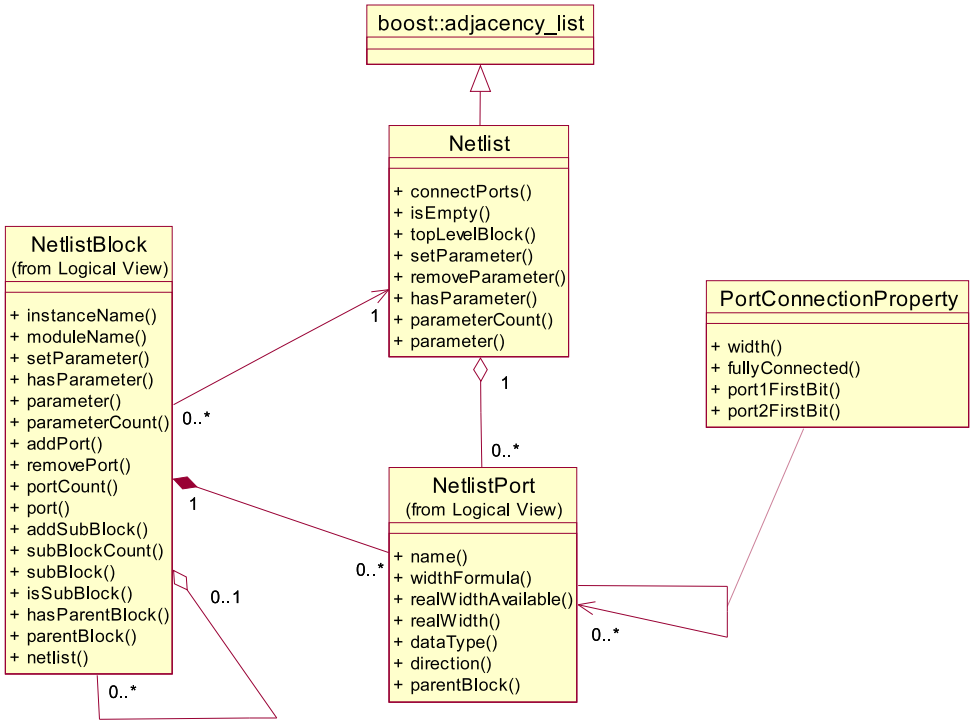

Figure 5.10: Class diagram of the netlist.

representing function units, register files, interconnection network etc. as sub blocks.

Each edge in the netlist contains properties represented by *PortConnectionProperty* class. It tells the width of the connection and the pins that are connected. This makes it possible to connect two ports partially. For example, just the two least significant bits could be connected, even though the ports were wider.

Since netlist is the only input given to HDL writers, it must contain all the necessary data to generate valid HDL code. Each *NetlistBlock* instance has module and instance names defined. The module name is the name of the hardware entity and instance name is the name of the particular instance of the module, naturally. Since the hardware blocks may be parameterised, *NetlistBlock* instances must also have parameters defined with values. *NetlistPort* instances have all the data needed to write the HDL. That is, there are port name, data type, width formula and direction defined.

The netlist is designed to be HDL independent which means the data types of the ports are not real data types as written in HDL, like *std_logic_vector*. Instead there are two possibilities for data type: bit or bit vector. The HDL writer used to write the netlist in some HDL, writes the real types depending on the HDL. For example, bit type is represented as *std_logic* and bit vector as *std_logic_vector* in VHDL.

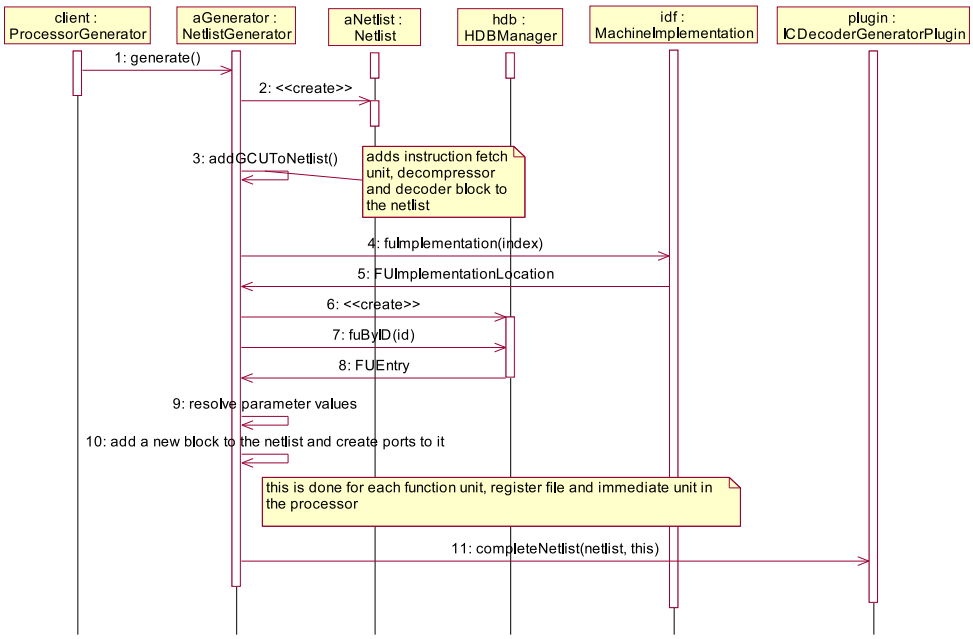

Figure 5.11: Sequence diagram of netlist generation.

5.4.2 Netlist Generator

The netlist is generated by *NetlistGenerator* class. The sequence diagram of the process is depicted in Figure 5.11. *NetlistGenerator* is given the machine object model, machine implementation definition and the IC/decoder plugin to use. It reads the machine implementation definition, requests the block information from *HDBManager* and adds the function units, register files and immediate units as well as instruction fetch unit, decompressor and decoder to the netlist. After that it requests the plugin to add interconnection network and finalises the decoder and connections by calling *ICDecoderPlugin::completeNetlist()* to the plugin.

5.4.3 IC/Decoder Generator

As mentioned before, the IC/decoder generator is a dynamic module to give TCE users the possibility to experiment with different implementation alternatives of instruction decoder and interconnection network. IC/decoder generators must be derived from *ICDecoderGeneratorPlugin* class. They have to implement the four pure virtual methods declared in the base class: *completeNetlist()*, *requiredRFLatency()*, *verifyCompatibility()* and *generate()*.

The *completeNetlist()* method is used to complete the initial netlist generated by *NetlistGenerator*. This method adds IC to the netlist and completes the decoder

by adding the ports required to control the processor. In addition, it creates the required connections to the netlist from IC and decoder.

The *requiredRFLatency()* method returns the required latency of the register file module selected for the given immediate unit. The latency required depends on the latency defined in ADF and the implementation of the processor control. This method is used to verify the selected implementation has an appropriate latency.

The *verifyCompatibility()* method is used to verify the IC/decoder generator plugin is compatible with the machine defined in ADF. It might be possible that the plugin is unable to create control unit or interconnection network as defined in ADF. This method checks the machine architecture definition for compatibility and throws an exception if the plugin is not compatible with the machine.

Finally, the *generate()* method generates the instruction decoder and interconnection network to the given destination directory.

The plugin framework supports also passing of user given parameters. That is, when the processor is being generated, users can define parameters which are given to the IC/decoder generator plugin. For this purpose, there is *setParameter()* method in the interface of *ICDecoderGeneratorPlugin* class. The base class stores the parameters set, and there are *hasParameter()* and *parameterValue()* methods in the protected interface available to the derived classes. They can be used to ask whether the parameter is set, and its value.

It is not forced by the framework, but recommended, to make the IC and decoder generators to different classes. By doing so, it is possible to make new plugins by combining compatible IC and decoder generators. The default IC/decoder generator plugin provided in TCE is made just like that. It consists of four classes: *DefaultICDecoderPlugin*, *DefaultDecoderGenerator*, *CentralizedControlICGenerator* and *DefaultICGenerator*. *DefaultICDecoderPlugin* is the main plugin class derived from *ICDecoderGeneratorPlugin*. *DefaultDecoderGenerator* generates the decoder and *DefaultICGenerator* the interconnection network. It is derived from *CentralizedControlICGenerator* which is a general purpose base class for IC generators of processors with centralised control. The reason for this inheritance hierarchy is to make the *DefaultDecoderGenerator* unaware of *DefaultICGenerator*. Instead, it uses the interface of *CentralizedControlICGenerator* to request the port data required to connect the decoder to the IC block. Because of that, the *DefaultDecoderGenerator* can be used together with any IC generator derived from *CentralizedControlICGenerator*.

5.4.4 Netlist Writer

The task of netlist writers is to write the given netlist in some hardware description language. Currently, ProGe has *VHDLNetlistWriter* as the sole netlist writer

implementation. Netlist writers are derived from abstract base class *NetlistWriter*. They must implement the pure virtual *write()* method which just writes the given netlist to the given destination directory in HDL. The written netlist may be constituted by one or several files. It is up to the netlist writer implementation. The *VHDLNetlistWriter* writes the netlist parameters to a VHDL package to a separate file and the netlist to another one.

5.4.5 User Interface

ProGe has only command line user interface. However, it is taken into account that it might be used from different user interfaces. The user interface is clearly separated from the rest of the application. There is *ProGeUI* class from which different user interfaces should be derived. Basically, the class is a helper for loading data structures, such as machine object model, binary encoding map and machine implementation description from the corresponding input files given. There are several protected methods available to the derived user interface class. The purpose of this design is to make implementing new user interfaces as easy as possible. They just need to read user given data and load the data to the base class. For example, the base class takes care of loading the dynamic module used to generate IC and decoder. Finally, the processor generation is triggered by calling *generateProcessor()* method of the base class.

5.5 Restrictions

Even though ProGe was designed to be rather flexible and to restrict the user defined function units and register files as little as possible, there are some restrictions concerning the supported ADFs and user defined hardware blocks. Some of the restrictions are caused by ProGe itself and the others are IC/decoder generator plugin specific restrictions.

5.5.1 Width Formulas of Function Unit Ports

Since ProGe tries to resolve the parameter values of function units from the width formulas defined for ports and from the port widths defined in ADF, it causes a restriction to width formulas: For each parameter of a block, there must be at least one port of which width formula is directly the name of the parameter without any mathematical operations. For example, if we have a function unit block with parameter *dataw*, there must be a port of which width formula is *dataw*. ProGe does not support parameter resolving from width formulas containing some mathematical operations, such as *dataw*2-1*.

5.5.2 Active High Resets

Reset signals taken by function units and register files must be active low. The hardware database does not contain the information whether they are active low or high, so they are always assumed as active low. It would be quite easy to support also active high resets, but it would require the additional information to HDB.

5.5.3 Bidirectional Ports

Bidirectional ports are not supported by the default IC/decoder generator. In hardware, implementation of a bidirectional port is not cheaper or anyhow better than two separate ports: input and output. That is why it was decided to not support them by the default plugin.

5.5.4 Bridges

The default IC/decoder generator does not support bridges [5]. Even if they are supported by ADF, most of the tools in TCE toolkit does not support them, so it was decided to not support them in the default plugin either.

6. VERIFICATION

To make sure PIG works correctly, it must be verified that the program and data images it generates are correct. The problem here is to get the correct reference outputs. The most sensible way of verificating ProGe is to compile the output files and simulate the behaviour of the processor by a VHDL simulator. To simulate the processor, a program image is needed, which must be generated by PIG. That is, if the result of VHDL simulation with a program image generated by PIG is correct, it is highly probable that both PIG and ProGe works correctly. On the other hand, if the VHDL simulation fails, it is hard to say where the error is. That is why PIG was tested at first to get correct program images for VHDL simulation.

6.1 Verification of the Program Image Generator

For the very first test of PIG, the reference program image was generated by hand. It was a simple program of about 30 instructions. In the second phase, the reference program image was generated by the program image generator of MOVE framework. Luckily, we have compatibility tools to convert files from MOVE format to TCE format and they were of great help in this test. The initial program code was in MOVE assembler format and the processor architecture also in MOVE format. Both of them were converted to TCE format. We do not have converter for binary encoding map, so it had to be converted by hand. Finally, the same program, processor architecture and binary encoding map were in MOVE and TCE formats and the test could be carried out.

6.2 Verification of the Processor Generator

The first tests for ProGe were held by generating an arbitrary processor with it and trying to compile the VHDL codes by a VHDL compiler. When the codes were free of compilation errors, simulating the processor with ModelSim [12] was started. For simulations, a testbench had to be created. The testbench connects the processor to instruction and data memories, generates the clock signal, sets the initial program counter value and other processor control signals to run the processor. The simulation result was verified by means of bus traces. The value on each bus in the processor was printed to a file in each simulation cycle. Similar bus trace obtained from TTASim was used as reference.

6.3 Co-verification

Basically, the co-verification was started when ProGe was able to produce VHDL codes ready for simulation. The input program and data images for simulation were generated by PIG. Four test cases which test a little bit different things in PIG and ProGe were run. The test cases use the same source program but in each case the processor architecture is a little bit different, which causes also changes to the program and data images. Each test case is based on a TTA processor for 1024-point fast fourier transform. The execution of the program takes around 5200 cycles with the processor architecture used. The basics of the architecture and the program are discussed in [13]. In the following subsections, the differences of the cases are covered.

6.3.1 FFT Case With Short Immediates

In this case all the immediates were encoded as short. This case was selected to test whether PIG can generate the program image using short immediates correcly and whether processor generated by ProGe works correctly with them. This is the simplest of the test cases and it was carried out first. In this case, the instruction memory had a MAU of the width of the instruction, that is, each instruction took one memory location, which is the simplest case.

6.3.2 FFT Case With Long Immediates

In this case the short immediates were replaced with long ones. It required a 32-bit immediate unit to the processor architecture. A dedicated long immediate field was added to the instruction format. The use of long immediates was not optimised at all in this case. The long immediates were encoded in the dedicated long immediate field in one part. Due to the dedicated long immediate field, the width of the instructions increased by one bit compared to the short immediate case. This test case tested the ability of PIG to generate program image with long immediates and the ability of the processor to decode the instructions having them. Since the processor handles short and long immediates totally differently, this was an important test case to make sure the processor works with simple long immediates.

6.3.3 FFT Case With Optimised Long Immediates

This is a more sophisticated version of long immediate test. In this case the instruction width was optimised by encoding the long immediates in move slots. This resulted in 137-bit instructions, while the non-optimised version required 150 bits. This case was selected to test how the PIG can dismember the long immediates

to several move slots and how the processor generated by ProGe can combine the portions and store them to an immediate unit.

6.3.4 FFT Case With Compressed Program Image

Dictionary compressor and decompressor were tested in this case. This is similar to short immediate case except that the instructions were compressed by the dictionary compressor resulting in instruction width of 6 bits.

6.3.5 Function Pointer Test

A separate function pointer test was ran too. This test was mainly for PIG, but ProGe was used to verify the result. A simple test program which used function pointers was written in C language. It calls functions which print text to standard output. Because of function pointers, there are instruction addresses encoded as immediates in instructions. PIG has to fix the immediate values, as discussed in 4.3.1. To make the test even more extensive, the MAU of the instruction memory was selected to be 16 while the instruction width was 46 bits. That is, each instruction took three MAUs and the increment in instruction addresses was three.

In this test case the result was not verified by comparing bus traces since the bus traces of TTASim and VHDL simulation differ due to the memory addresses. TTASim operates in higher level: it does not care about the final instruction addresses after program image generation. In this case, each instruction address is three times larger in reality than in TTASim which has an effect to the bus trace, of course. The test case was verified by comparing the output of the program. The program printed some text, so a special function unit which implemented the *stdout* operation was created. In VHDL simulation, the function unit printed the character to the output file, which was then compared to the output of the C program.

7. CONCLUSIONS

In this thesis, the last phase of TTA processor design flow is implemented. Inputs to that phase are processor architecture definition (ADF) and a program or a set of TTA assembly programs scheduled for the particular architecture. The final program and data images to be executed by the processor, as well as structural description of the processor hardware has to be generated in this phase. Two separate applications, Program Image Generator (PIG) and Processor Generator (ProGe) were designed and implemented in order to make the last phase of the design flow as automated as possible.

PIG generates the program and data images from the given input assembly programs and data sections. The program image is generated as a bit-accurate image which is ready to be loaded to the instruction memory of the target TTA processor. Similarly, data images are ready to be loaded to data memories of the target processor. The instructions are encoded by means of the encoding rules defined in BEM, and they can be further compressed to save instruction memory. The output format can be selected from a fixed set. The set is selected to provide necessary formats for VHDL simulation and for real use in TTA processors. Unfortunately, a TCE user might need some different format, which is always the bad side of fixed set. The responsibility of decompressing compressed instruction is left to the code compressor plugin. It must generate the decompressor module which is used in the processor hardware generated by ProGe.

ProGe generates synthesisable processor description according to the given ADF, IDF and BEM files. It supports only VHDL, but it is designed from the premiss that other hardware description languages can be supported in the future. Due to that, HDB must contain only HDL independent data and ProGe generates a HDL independent netlist representation of the processor. The netlist is written by a netlist writer to HDL files. In order to support an additional HDL, a new netlist writer must be implemented. In principle, there should be no problems, but I am a litte bit suspicious of it. A potential source of problems are the data types of ports defined in HDB. There are two data types: *BIT* amd *BIT_VECTOR*. In VHDL, *BIT* is converted to *std_logic* and *BIT_VECTOR* to *std_logic_vector* but I am not sure if the conversions can be done unambiguously in every HDL.

The applications are designed to be as generic as possible. It is achieved by using

plugin modules in critical portions of the code. In PIG, code compression is done by a plugin, which is the only way to enable experimenting new code compression algorithms. In ProGe, the implementation of instruction decoder and interconnection network are generated by a plugin. As designing the plugin interfaces, I noticed you have to be careful with them. Otherwise you will easily end up having the whole application as one big plugin. The challenge in designing plugin frameworks is to try to minimise the development work of new plugins, while trying to keep the restrictions the plugins can do as mimimal as possible. I am not completely satisfied with the result in case of IC/decoder plugins. It requires quite a lot of effort to create a new plugin. For example, the default plugin which was implemented, takes about 3000 lines of raw C++ code. And to generate the same IC/decoder structure in a new HDL, a completely new plugin implementation is required. But, on the other hand, I do not know a better solution.

To measure the success of this thesis, it can be said that the main goal is achieved: Program image and processor generation phase of the TTA processor design flow can be successfully carried out with these applications. The applications are verified with different test cases to be working. They are carefully designed to be expandable and they have a modular design which is easy to maintain. They are not perfect, but good enough to the first version of TCE. They provide a good platform for researchers to experiment different code compression algorithms and IC/decoder implementations.

BIBLIOGRAPHY

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

[2] *The MOVE Framework User's Manual*, Tampere Univ. Tech., 2004.

[3] P. Jääskeläinen, "Instruction Set Simulator For Transport Triggered Architectures," Master's thesis, Tampere Univ. Tech., Tampere, Finland, Sept. 2005.

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

[5] A. Cilio, H. J. M. Schot, and J. A. A. J. Janssen, "Architecture Definition File: Processor Architecture Definition File Format for a New TTA Design Framework," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2003-2006.

[6] L. Laasonen, "TTA Processor Generator: Functional Requirements," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2004-2006.

[7] A. Cilio, "TPEF: TTA Program Exchange Format," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2003-2006.

[8] L. Laasonen, "Hardware Database: Design Notes," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2005-2006.

[9] "SQLite Home Page," Website, http://www.sqlite.org.

[10] J. Heikkinen, A. Cilio, J. Takala, and H. Corporaal, "Dictionary-Based Program Compression on Transport Triggered Architectures," in *IEEE Internation Symposium on Circuits and Systems.*, Kobe, Japan, May 23–26 2005, pp. 1122–1125.

[11] "The Boost Graph Library," Website, 2006, http://www.boost.org/libs/graph/doc/table_of_contents.html.

[12] "ModelSim - a comprehensive simulation and debug environment for complex ASIC and FPGA designs," Website, http://www.model.com.

[13] T. Pitkänen, R. Mäkinen, J. Heikkinen, T. Partanen, and J. Takala, "Low-Power, High-Performance TTA Processor for 1024-Point Fast Fourier Transform." in *Embedded Computer Systems: Architectures, Modeling, and Simulation: Proc. 6th Int. Workshop SAMOS 2006*, ser. Lecture Notes in Computer

Science, S. Vassiliadis, S. Wong, and T. Hämäläinen, Eds., vol. 4017. Springer, 2006, pp. 227–236.