TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

HENRY LINJAMÄKI

INSTRUCTION MEMORY HIERARCHY GENERATION FOR CUSTOMIZED PROCESSORS

Master of Science thesis

Examiner: D.Sc. Pekka Jääskeläinen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 14th January 2015

ABSTRACT

HENRY LINJAMÄKI: Instruction Memory Hierarchy Generation
for Customized Processors
Tampere University of Technology
Master of Science thesis, 57 pages, 0 Appendix pages
November 2015
Master's Degree Programme in Electrical Engineering
Major: Embedded systems
Examiner: D.Sc. Pekka Jääskeläinen
Keywords: memory hierarchy, loop buffer, cache, transport triggered architecture, processor generation

Due to the performance gap between processor and memory, instruction memory hierarchy design is a mandatory part of processor designs. Memory hierarchy does not only help on keeping up performance, but it also affects the power consumption of large memories. Designing for low power is important aspect for embedded processors used in mobile devices, and well designed memory hierarchy can reduce power consumption in memories considerably as it can mask high latencies. In this thesis, generation of customized memory hierarchy was implemented and integrated into processor generator of *TTA-based Co-design Environment* (TCE) developed at Tampere University of Technology. The generator reads an description of instruction memory hierarchy and produces a TTA-based processor that includes the specified hierarchy. In addition, cache statistics collection was implemented to support the exploration of suitable memory hierarchies. Implemented features were verified in register transfer level simulation using TCE's processor test benches. Area and power estimations were produced using a synthesis tool, for three low power processor designs targeted to run at least at 1 GHz.

TIIVISTELMÄ

HENRY LINJAMÄKI: Käskymuistihierarkian generointi räätälöidyille suorittimille
Tampereen teknillinen yliopisto
Diplomityö, 57 sivua, 0 liitesivua
Marraskuu 2015
Sähkötekniikan koulutusohjelma
Pääaine: Sulautetut järjestelmät
Tarkastajat: TkT Pekka Jääskeläinen
Avainsanat: muistihierarkia, siirtoliipaisuarkkitehtuuri, silmukkapuskuri, välimuisti, prosessorigenerointi

Prosessoriytimien ja muistien välisten suorituskykyvajeen vuoksi käskymuistihierarkian suunnittelu on erottamaton osa prosessorien suunnittelua. Muistihierarkia ei pelkästään pidä prosessorien suorituskykyä yllä, mutta se voi myös vaikuttaa suurten muistien tehonkulutukseen. Sulautettujen prosessorien suunnittelu vähävirtakulutteisiksi mobiililaitteita varten on myös tärkeää, koska hyvin suunniteltu muistihierarkia voi vähentää tuntuvasti tehonkulutusta eikä vain pelkästään nopeuta muistien käyttöä. Tässä diplomityössä toteutettiin räätälöityjen muistihierarkioiden generointi, joka integroitiin prosessorigeneraattoriin. Tämä generaattori on osa Tampereen teknillisellä yliopistolla kehiteltyä *TTA-based Co-design Environment* (TCE)-kehitysympäristöä. Se lukee syötteenä muistihierarkiakuvauksen, jonka perusteella se luo prosessorin, joka sisältää määritellyn hierarkian. Lisäksi tuotettiin työkalu taltioimaan generoitujen muistihierarkioiden suorituskykytilastoa, jota käytetään sopivan hierarkiakonfiguraation etsimisessä. Toteutetut ominaisuudet verifioitiin rekisterisiirtotason (*register transfer level, RTL*) simulaatiossa käyttäen TCE:n luomia prosessoritestipenkkejä. Pinta-ala- ja tehoarvioita tuotettiin käyttäen synteesityökalua kolmelle vähintään yhden gigahertsin kellontaajuutta käyttävälle matalan tehonkulutuksen prosessorikonfiguraatiolle.

PREFACE

The work in this M.Sc thesis was completed at the Department of Pervasive Computing at Tampere University of Technology during 2014 and 2015.

I would like to thank Pekka Jääskeläinen, D.Sc., for the opportunity to work on this thesis project. The thesis project has been interesting and challenging. Designing and implementing automated tools is something I would like do in the future. I am also grateful for Pekka Jääskeläinen for his guidance in the thesis work.

I would also thank all my coworkers in the Customized Parallel Computing (CPC) group for creating a relaxed and interesting work atmosphere. Especially, I would like to thank Joonas Multanen for his help on the power measurements, and Lasse Lehtonen for his advices.

Finally, I would like to thank my family for their invaluable support on my studies and life.

Tampere, November 23

Henry Linjamäki

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

ADF	Architecture Definition File
AMAT	Average Memory Access Time
ASIP	Application-Specific Instruction Set Processor
CU	Control Unit
DRAM	Dynamic Random Access Memory
FU	Function Unit
HDL	Hardware Description Language
HLL	High Level (programming) Language
IC	Interconnect (Network)
ILP	Instruction-Level Parallelism
IDF	Implementation Definition File
IU	Immediate Unit
LFSR	Linear Feedback Shift Register
LRU	Least Recently Used
MCP	Micro-Controller Processor
MCU	Micro-Controller Unit
NOP	No-operation
PIG	Program Image Generator
ProGe	Processor Generator
RF	Register File (Unit)
RTL	Register Transfer Level
SRAM	Static Random Access Memory
TeGe	(Automated) Test Generator
TCE	TTA-based Co-Design Environment
TCEASM	TCE Assembly Code (File)
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architecture
XML	Extensible Markup Language

1. INTRODUCTION

Energy-efficiency is an important design aspect for today's electronic devices utilizing processors. It affects battery life in mobile devices, heat production and spending on electricity. For example, modern devices, such as smart phones, have increasing demand for more powerful processors and larger main memory for running more advanced applications. Both of the demands increase power consumption.

One solution for preserving or improving energy-efficiency is to design processors that are targeted for an application area. *General purpose processors* (GPP), as they are called, are targeted for executing a wide range of applications. For specific applications, fixed features in GPPs can be excessive and, hence, consume power unnecessarily. Therefore, designing a processor just for a small application area can save energy, as only the needed functionality is included - without performance loss. Energy-efficiency can be further increased by adding right features. By adding suitable accelerators, programs can be executed more efficiently in respect of power consumption.

Power consumption of memories is caused by dynamic activity and leakage. Both of them can be addressed by using memory blocks of low power design. Nevertheless, there is room for improvements. It is well known, that most of the time programs run in loops. This means that same sequences of code retrieved from memory multiple times. Memory hierarchy designs exploit this property by caching data in smaller memories. The smaller the memory is, the faster and less power consuming it is. In high performance computing, caches are mainly used for masking high latencies of memories [1], but in low power applications the power-saving is more pronounced.

There exists several ways to implement memory hierarchies. Designing a memory hierarchy is a trade-off between chip area, performance, cost and power consumption. For finding a suitable hierarchy design, several implementations with different parameters and features may need to be tried out. Resulting candidates need to be tested for correctness. This process can be iterative, laborious and time-consuming.

This thesis project addressed the inconveniences of implementing a memory hierarchy by streamlining some parts of the process. This includes tool-assisted generation of customized instruction memory hierarchy and tools to verify and to collect performance statistics from it.

The implementations of the work are integrated into *TTA-based Co-design Environment* (TCE) [2] where the memory hierarchy generation was requested. TCE, developed at Tampere University of Technology, provides tools for developing *application specific instruction set processors* (ASIP) based on *transport triggered architecture* (TTA). The static nature of TTA suits well for low power applications. TCE covers developing of datapath components aspect, but it lacked customization of instruction memory hierarchies.

This thesis is divided to chapters as follows: Chapter 2 gives background of customized processors the part that is relevant for instruction memory hierarchies. TCE is introduced in more detail. Chapter 3 is an introduction to common instruction memory hierarchy solutions and basic concepts. Chapter 4 introduces processor generation, which is the target for integrating most of the thesis work in TCE. Chapter 5 tells about a loop buffer implementation, its automatic generation in TTA processors and integration into TCE. Chapter 6 explains about how instruction memory hierarchy generation is utilized in TCE, and implemented and integrated in processor generation. This chapter also includes statistics collection from memory hierarchy and its implementation. Chapter 7 tells about new tools for verification. The verification of implementations and its process is explained. Chapter 8 proposes extensions and improvements to the work made in this thesis. Chapter 9 introduces and compares work of others regarding the memory hierarchy customization and loop buffer implementation. Finally, in Chapter 10 the thesis is concluded.

2. CUSTOMIZED PROCESSORS

Customized processors are tailored processor designs targeted at a specific application area. An application area for customized processors is usually narrower than of GPPs, and it can be even be just a couple of selected tasks. Reasons for deciding on a customized processor, instead of a GPP, likely arise from the design constraints. GPPs may have excessive features, inadequate performance, may not meet real-time requirements and can be power-hungry for targeted application domain [3]. With processor customization, considerable improvements over GPPs can be achieved in terms of chip area, energy-efficiency, performance and cost.

There are abundant number of customization points in processor design to choose from: number of computation units and registers, number of cores, processor architecture paradigm, pipeline design, design of instruction set, interconnect network design and others. One an important aspect of processor design is the memory system design. This is due to the large performance gap between the memory and processor cores [4, p. 391].

Customization of processors have been unfeasible due to the enormous design effort for the reasons listed in [1, p. 151]. Fortunately, recent tools are helping the case by assisting in the design of customized processors and automating or streamlining generation and verification of customized processors and, thus, making them more accessible. Such tools are Tensilica Xtensa [5] and TCE [2], for example.

2.1 Transport Triggered Architecture

TTA is a statically scheduled architecture relative to *very long instruction word* (VLIW). Modularity of both the architectures makes them scalable and can be designed for instruction-level parallelism (ILP). Differences between TTA and VLIW are that [6, 3]:

- VLIW processors have complex register file (RF) and bypass network, whereas TTA can avoid it.

- Data paths of a TTA processor are visible and controllable; allowing programmers and compilers to utilize data bypasses and, thus, reducing RF usage.
- Instructions in VLIW specifies operations to be executed. In TTA, instructions specify data transports between FUs and RFs and operations are performed as side effects of the transports.
- As a downside, TTAs usually have wider instruction words than VLIW due to increased control of the interconnect network (IC).

2.1.1 Processor Organization

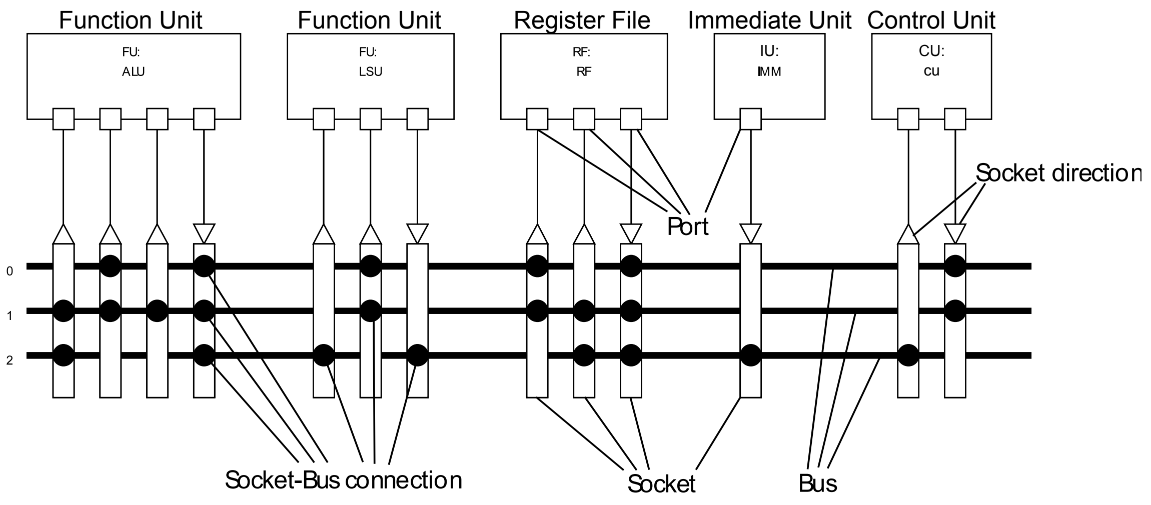


Figure 2.1 *An example diagram of a TTA architecture.*

In the Figure 2.1 is an example TTA processor or "machine". A TTA machine consists of number of TTA units which are connected via IC.

Different types of TTA units are *function unit* (FU), *register file* RF, *control unit* and *Immediate Unit* (IU). *Function units* are units for computations. Each has one or more operations they can execute. For example, in the figure 2.1 arithmetic logic unit (ALU) performs basic arithmetic operations and load store unit (LSU) accesses data memory. *Control unit* (CU) is a special case of FU and controls the flow of programs run in the TTA processor and is responsible for instruction fetching. Example control flow operations are *jump* and *call*. *Register Unit* (RF) is unit for temporary storage. Each RF has a number of register entries of specified width. *Immediate Unit* (IU) holds values of long immediates. IUs are basically read-only RFs whose values are loaded from instructions.

The interconnect of a TTA machine consists of *ports*, *sockets* and *transport buses*. The *ports* are the interfaces of TTA units. Values are transported between them and *sockets*, which connects ports to transport buses, indicated by filled circles in the figure, and arrow denotes the direction of the value transportation. Transport buses move values between sockets in which they are connected. On a bus, a single value can be transported at a time from single input socket to another output socket.

2.1.2 Programming Model

Programs in TTA are made of data transports between ports. In other word, programmers and compilers have control of the IC. This differs from traditional processor architectures called *operation triggered architecture* (OTA), where program specifies operations to be executed as in VLIW. For example, a snippet below adds two values from registers R1 and R2 together and stores its result into register R3:

```
ADD R1, R2, R3
```

TTA instructions consists of *moves* where each one specifies a data transport. An equivalent code of earlier example in two instructions of TTA assembly language is:

```
RF.1     -> ALU.in1t.add, RF.2 -> ALU.in2 ;
ALU.out1 -> RF.3        , ...             ;
```

In the first instruction values are moved from registers to FU *ALU* port *in1t* and *in2*. One of the ports, *in1t*, is a triggering port. the operation *add* is performed when a move is made to this port. The other port, *in2*, is operand port that does not trigger any operation in the *ALU*. In the later instruction result of the operation from ALU's result port is moved to register. The three dots "..." on a slot denotes a *no-operation* (NOP or NOOP), where no transfer is made.

2.2 TTA-based Co-design Environment

TTA-based Co-design Environment (TCE) is a tool set for designing customized TTA processors [2] and developing programs for them. The environment provides tools for design exploration, a retargetable instruction set simulator supporting user defined operations, processor generator, program image generator and retargetable high level language compiler.

TTA Processor Designer *TTA Processor Designer* or ProDe is a tool with graphical user interface for designing TTA processors.

TCECC and TCE assembler TCECC is retargetable compiler for compiling TTA programs from high level languages - such as C/C++. It uses Clang as its front-end [7] and LLVM [8] as its middle-end. Retargetability of tool allows same source code to be compiled to different TTA machines at ease. TCE assembler compiler for handwritten TTA assembly code [9]. Both tools produce output file in *TTA Program Exchange Format* (TPEF).

Processor Generator *Processor Generator* (ProGe) is tool that generates hardware description of TTA processors from architecture and micro-architecture descriptions [10]. This is the tool, where customized instruction memory hierarchy generation is integrated in TCE.

Proram Image Generator *Program Image Generator* (PIG) is tool that takes TPEF programs as input and creates bit-level program images of TPEF for processors generated by ProGe [10].

Retargetable TTA Instruction Set Simulator TTA Instruction Set Simulator (TTAsim) is *TTA instruction set-architecture simulator* for simulating TTA programs in TPEF format in a TTA machine [11]. The simulator is capable to simulate any operation in the machine through *Operation Set Abstraction Layer* (OSAL). OSAL is database for operations and it stores static and dynamic properties of operations. Users can define their own operations, which then can be used in the TTA-simulations.

In this thesis data extracted from TTAsim is used in verification and for automated test case generation.

3. INSTRUCTION MEMORY HIERARCHIES

For embedded processors, both the performance and energy efficiency is a concern. Instructions of programs that processors executes resides in a memory system, that is for example an off-chip dynamic random access memory (DRAM) or an on-chip static random access memory (SRAM). Large memories are inherently slow [1] and consume more power than smaller ones [12]. Therefore, instruction fetching becomes a bottleneck due to slow and energy inefficient memories.

The issues of the large memories can be diminished by exploiting the nature of program execution. Most of the time, instructions of programs are fetched in linear sequence from memory. Programs have sections of instructions, loops and routines, that are repeatedly executed. These properties lead to a behavior that is called the *principle of locality* [1, 4]. It is divided to two forms: a *temporal* and *spatial locality*. *Temporal locality* states that accessed memory location will be likely to be accessed again in the near future. *Spatial locality* states that a subsequent memory access will be nearby the previous one.

With the *principle of locality*, instead of fetching instructions from slow main memory, they are fetched most of the time from smaller specialized memories. These specialized memories, *caches*, are in between the processor core and main memory. They stash data blocks from main memory requested by processor cores for a later reuse. Caches can be divided into two types: *transparent caches* and *software-managed caches* [13]. In the former, processor cores are not aware there is caches in between it and a main memory. However, caches of the latter type are visible to the cores and are controlled by them in some degree.

Processor core, caches and main memory forms a *memory hierarchy* (Figure 3.1). Parts in between the processor core and the main memory of this organization are called *levels*. Levels closer to processors have small caches, which are fast. Moving closer to the main memory, the level increases in capacity at the cost of speed and power consumption. [4]

It is common case that the main memory is shared along multiple processor cores and

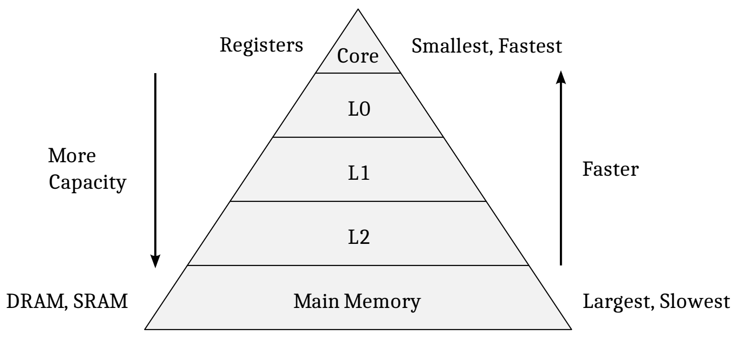


Figure 3.1 *A depiction of memory hierarchy.*

the memory is accessed via single memory bus. Therefore, a scheme for alternation of shared bus access among multiple processor cores is needed - an *arbitrator*. It is a unit that gets access requests to shared resources and assigns turns for the access using some algorithm - an *arbitration scheme*. [14]

The performance of a memory hierarchy depends on properties of its components and characteristics of programs executed in the processor core(s). Two useful metrics to measure the performance of the memory hierarchy and its parts are *miss rate* and *average memory access time* (AMAT). As processor cores request data from the memory, it is handled first in *first level* cache. If the cache has the requested data, it is a *hit*. On the contrary, it is a *miss*, and the cache forwards the request to the next level. The miss rate tells the portion of the misses of all accesses made to a cache [4]. When miss occurs in a cache, the requested data have to be fetched from slower levels. The AMAT tells how long it takes the requested data to arrive to processor cores on average [4].

The rest of this chapter discusses about components of the instruction memory hierarchy, that are implemented and integrated in this thesis. They are instruction cache, loop buffer and instruction bus arbiter.

3.1 Instruction Cache

Caches are transparent memories, which store copy of data fetched from lower cache levels or the main memory for later reuse. In case of instruction caches, the fetched data pieces are instructions of programs.

In the Figure 3.2 a logical structure of a instruction cache is depicted. A *block* is a set of, at least one or more, *sub-blocks* or *words*. It is an unit that is fetched at a time from the lower level. A block is associated with a *tag* and a *valid bit* ('V' in the figure). The tag is part of the highest address bits of memory location

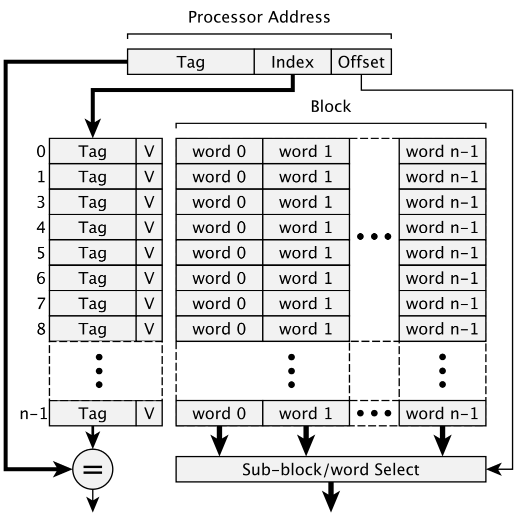


Figure 3.2 *A logical depiction of an instruction cache.*

address, from where a processor core requests an instruction in main memory. The valid bit, a boolean value, tells if a block represents a copy of data from the main memory at some memory location. The tag and valid bit together determines if a requested instruction is present in the cache. If the requested word is present, the cache supplies it to the processor. Since a block may consist of multiple words, the lowest bits of the address are used to select the requested word.

Basic cache data structure is formed from an array of entries of the block, a tag and a valid bit. *index* part of the address is used to point an entry in the cache.

Whole process of retrieving an instruction through the cache is as follows: A processor core fetches an instruction from some memory address. The request is first handled in the cache. Index part is used to select an entry from the cache data array. Next, presence is determined. If the valid bit in the entry indicates false, then it is clear, that the requested word is not in the cache. On the contrary, if the bit states true, the presence deduction is continued by comparing the tags. If the tags do match, the requested word is in the cache, and otherwise it is not. After the presence check, the process continues in one of two following cases. In the first case, if a hit occurred, then the requested word is given to the processor core. In the other case, cache fetches the missing requested word from the lower levels. The fetched block is placed into the block data array at the indexed location, replacing possible other block from a different memory location, and the tag data is updated.

The fetching process described above applies to so called *direct-mapped* caches. The same process for *associative* caches works a bit differently. In Figure 3.3, the types of cache associativity is laid out and their logical structure is depicted. When a block is fetched to a direct-mapped cache, it has only one location in block data array, where it can be placed, and the location is pointed by index part of the address. In direct-mapped caches there is a chance that useful, most frequently referenced block, gets replaced or “evicted”.

In *associative* caches a block can be placed in one of the multiple locations (or “ways”) at the same index depending on the degree of associativity. Such a construct increases the chance to keep the most frequently used blocks in the cache. This is achieved by increasing cache’s associativity at the cost of search complexity [1].

Replacement Policy As cache gets filled, at some point the cache will run out of available slots, where a block can be placed (at index location) freely. Therefore, an algorithm, which decides what block is evicted to make room for another one, is needed. *replacement policy* is an algorithm that selects a block to be replaced in cache set. In this thesis two replacement policies are presented.

Random replacement policy pseudo-randomly selects a block to be evicted. This policy is simple to implement and does not take much area. Disadvantage of the policy is that it may evict useful blocks from the cache, but on average the policy has good performance [1].

In *least recently used* (LRU) replacement policy a block to be evicted from the set is the one that has been referenced the least recently. LRU outperforms random policy in small caches [4]. LRU is usually implemented in small caches since keeping track of the “staleness” of the blocks takes up area [4].

Selection of replacement policy affects to the total size of the cache. The *Random* policy, being simpler to implement, takes less area than the *LRU*. Area requirement with *LRU* policy depends on the degree of associativity.

Loop buffer

Loop buffers or *loop caches* are used as level 0 caches and optimized for program loops [12]. They are useful in programs utilizing tight loops. Reusing instructions from the loop buffer reduces the amount of accesses made to the cache and memory, which may lead to a considerable power saving as programs tend to execute loops most of the time.

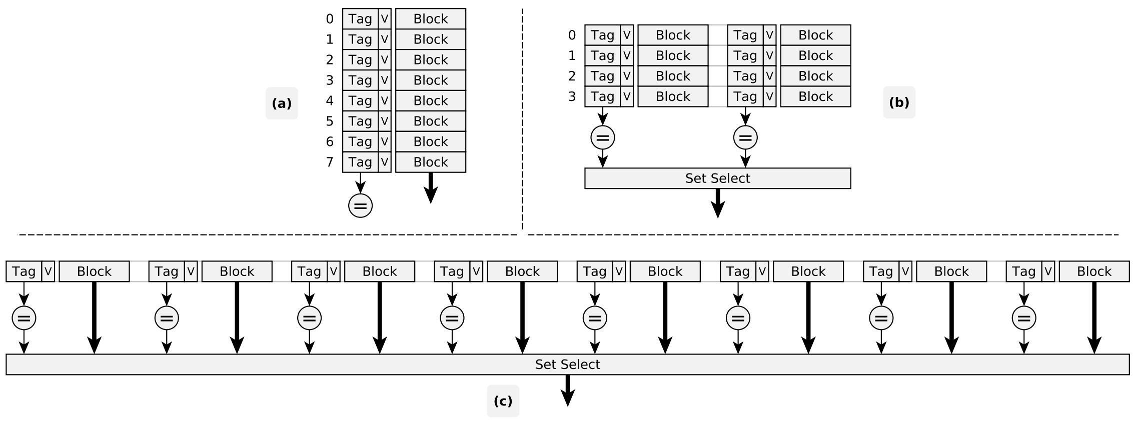


Figure 3.3 *Cache associativity. The figure shows cache configuration each having total of eight blocks in them with (a) direct-mapped, (b) semi-associative (2-way) and (c) full-associative associativity.*

3.2 Bus Arbiter

An arbiter is a hardware unit that prevents simultaneous access to a shared bus and decides, which *master* or *requestor* (a master that is requesting), gets access to it at a time. Masters, who need the access to the shared bus, sends a *request* signal to the arbiter. The arbiter then decides on one of the requestors using so called *arbitration scheme* and signals it a permission by sending a *grant* signal. [14]

Round Robin Arbitration Scheme One of the used arbitration schemes is *round-robin* and it is fair and pre-emptive scheme that guarantees that no requestor will ever starve [14]. That is, every requestor receives access to a shared bus eventually. In the scheme, every requestor is granted a access sequentially for predefined amount of time. When the time runs out or the access is no longer needed, the control of the bus is transferred to a next requestor in the line. While the scheme look for the next requestor, it skips over non-requesting masters.

4. PROCESSOR GENERATION IN TCE

In this thesis, most of the implementations are integrated into the processor generator (ProGe) tool. In this chapter, processor generator is introduced, its generations flow and internal hardware modeling explained briefly to clarify the integration context and, finally, changes made to internals of the ProGe.

4.1 Processor Generator

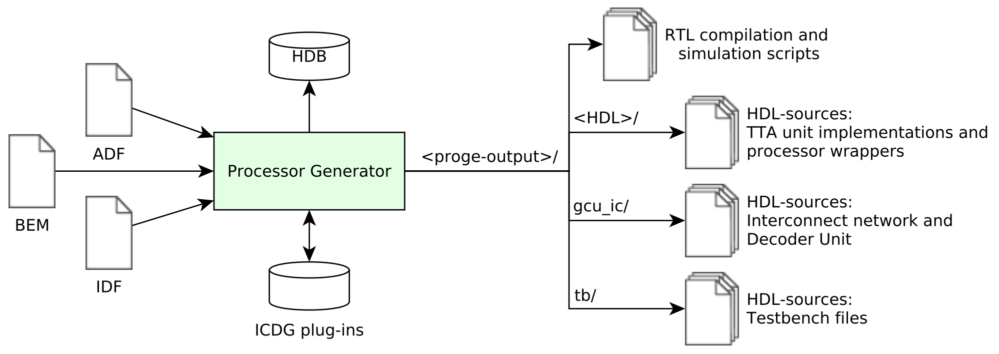

HDB
ADF
BEM
IDF
Processor Generator
ICDG plug-ins
<proge-output>/
RTL compilation and simulation scripts
<HDL>/
HDL-sources: TTA unit implementations and processor wrappers
gcu_ic/
HDL-sources: Interconnect network and Decoder Unit
tb/
HDL-sources: Testbench files

Figure 4.1 *The processor generator tool that takes in description files and produces HDL source files of the processor.*

In the Figure 4.1 is tool, processor generator (ProGe) for creating synhtesizable hardware description of TTA processor. The ProGe takes as input a set of architectural and micro-architectural description files: *architecture definition file* (ADF), *implementation definition file* (IDF) and optionally *binary encoding map* (BEM). All input files are in human-readable *extensible markup language* (XML) format. The processor is constructed using these description files. Resulting output of the ProGe is synthesizable hardware description of the processor in hardware description language (HDL) of choice - for example VHDL or Verilog.

Additionally, the ProGe can create a processor test bench, which allow simulation of program images in the created processor in RTL-simulation. The processor test bench instantiates the generated TTA processor and memory blocks for instruction and data memories. The RTL-simulation of the test bench is controlled by tool specific compilation and simulation scripts, where the former prepares the test bench for the simulations and the latter runs a simulation with program images currently loaded by PIG.

ADF *Architecture definition file* is a file that describes the TTA processor architecture as in Figure 2.1. It only includes semantics about the architecture that programmers and processor implementers need to know for programming valid programs and implementing a processor that can run the programs. What is excluded from the ADF is the actual implementation details of the processor - the micro-architecture.

IDF *Implementation definition file* contains the implementation details of the TTA processor. The IDF refers to static implementations of TTA units, that are stored in *hardware databases* (HDB). Creation of the IC and instruction decoding unit is delegated to the *Interconnect and decoder generator plug-in* (ICDG). IDF point to the plug-in that is used in processor generation.

BEM *Binary encoding map* is a file that defines how the instructions in a TTA machine are encoded. This is optional for ProGe and is implicitly generated from ADF if it is not provided.

4.1.1 Processor Generation Flow

Processor is built in ProGe as follows: The flow starts with reading the input description files, which are deserialized into object model counterparts. Using the model, ProGe then does validation to check if the processor is feasible to be generated.

If the target machine is feasible to implement the ProGe continues to build a processor core template in the HDL-independent structure object model provided by the *Netlist* module. The template is a preliminary model, that includes TTA unit implementations loaded from HDB(s) and instruction pipeline structure. In this phase the implementations of instruction pipeline are not yet selected.

The ProGe calls ICDG-plug-in, which finishes the processor core template by adding dynamically generated IC, and instruction decoder unit and selecting implementation for the instruction fetch unit.

After the ICDG call, the processor core is completed. In case of multi-core TTA processors the ProGe generates a wrapper that instantiates the earlier built TTA core multiple times. In this phase, entities or features shared across the cores are generated too.

In final phase of the flow, the HDL sources of the processor are generated. HDL sources of TTA unit implementations are copied from the HDB(s) and processor structure model is converted to the target HDL using a writer module. Also, processor wide files are written and copied. The files includes packages holding processor wide parameters and some common HDL utilities.

4.1.2 Instruction Pipeline

Relevant to this thesis is instruction pipeline used in TTA processors. Parts of the instruction memory hierarchy are placed within and around of it. In the Figure 4.2 is structure of instruction pipeline that is generated by default in ProGe.

Instruction fetch unit fetches instructions from the memory via the memory interface. There is a couple types fetch units provided by TCE: normal fetch unit that fetches fixed sized instructions and another that fetches variable length instructions [15]. In both types the width of the fetch block is fixed, but in latter fetch unit type, a single fetch block may have zero or more instructions. The fetch unit expects that the instruction memory supplies data from the given address when memory enable signal is active. If the memory can not response (in time) to the request, it can stall the instruction fetching by asserting busy signal. The fetch unit is the control unit in TTA machines and therefore handles control-operations bound to it. If instruction decompression is used in the processor, then a *instruction decompressor* unit implemented by the program image generator. In *instruction decoder* the instruction words are decoded into control signals of data transports.

The process of fetching data from the instruction memory and converting it into control signals takes some cycles. Each unit in the Figure 4.2 takes zero or more cycles to transfer data. In the Figure 4.3 is shown a register transfer diagram of default four cycle delay instruction pipeline that is generated by ProGe. The latency of the pipeline visible in ADF and in TTA program code it appears as delay slots after control-flow operations. For example, if a machine has n delay slots, then a control-flow operation does not take place until n cycles and during those cycles instruction are executed.

The *global lock* and *global lock request* signals are for processor stalling. They are

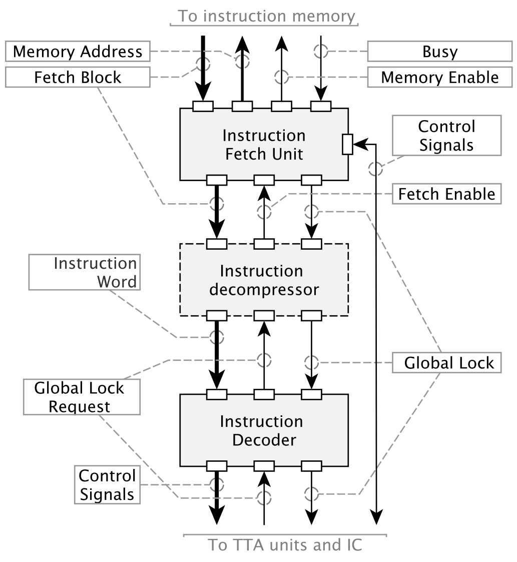


Figure 4.2

used in cases, where a FU can not handle operations in time, or the instruction memory is busy. The active *global lock* signal halts all activity in the processor and it is kept that way until the lock issuer is ready to continue.

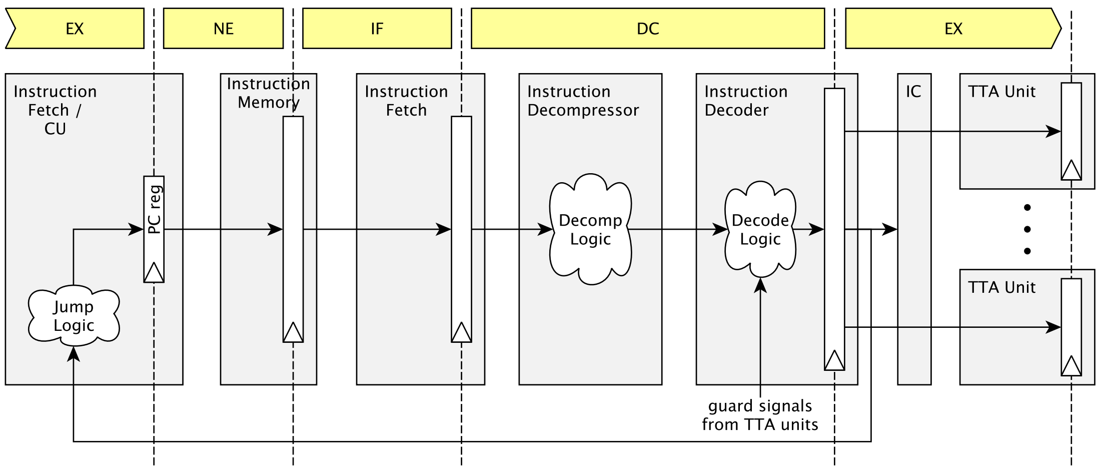


Figure 4.3 *The default four delay instruction pipeline and its stages. the acronyms in the diagram stands as NE for Next, IF for Instruction Fetch, DC for Decode and EX for Execution. Note that the EX stage at right spans to the left side as Instruction fetch unit is part of the CU.*

4.2 Netlist Module

Netlist module provides a way to model hardware blocks HDL independently. The ProGe internally constructs processors by creating building blocks, from the implementation units from HDB and internal library, and connecting them together. The building blocks have enough information so that HDL sources can be written out.

In the Figure 4.4 is presented basic classes that are used to model the hardware structures. A Processor is constructed by using these classes as follows. Netlist blocks are created to model the implementation source files - the ones that are stored in HDB and internal library. The blocks have same interface, formed by netlist ports and parameters, as what they model after. The blocks are empty inside as their realization is provided by the source files.

Next, another block can include the above blocks and define connections between their ports and own like in figure 4.4. For example, a TTA processor is formed like this in the ProGe. The HDL source codes of these combining netlist blocks are created by netlist writers. A netlist writer reads the netlist structure at block level and produces a HDL source file after it. The netlist classes provides enough information, so netlist writers can write valid HDL sources that instantiate successfully other blocks and make connections. For example a netlist port has information of its name, direction and port width. In the table 4.1 shows how classes module maps to language features of Verilog and VHDL.

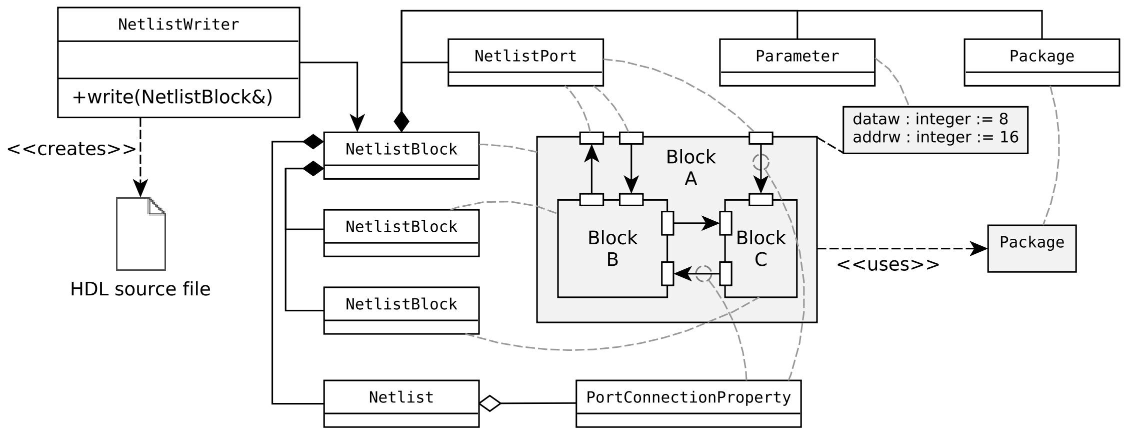


Figure 4.4 *The classes of the Netlist module and their correspondence to logically depicted hardware structure, and a writer class that generates HDL source from netlist block structure.*

More about the Netlist module is explained in master thesis of [10]. However, the module have been extended. The additions and changes are explained in the following:

Isolated netlist blocks Objects of NetlistBlock class are freely modifiable - ports, parameters, sub-blocks and internal connections may be added, changed and re-

Table 4.1 *Netlist classes and their rough correspondence to ones in HDL languages of Verilog and VHDL.*

Netlist module	Verilog	VHDL
NetlistBlock	Module	Entity
NetlistPort	Input/Output	Port
Parameter	Parameter	Generic
Package	Include	Package
PortConnectionProperty	Wire	Signal

moved. This makes the class unsuitable to derive specialized netlist block classes that would need restricted capabilities. Netlist module was changed so all netlist block classes are derived from *BaseNetlistBlock* class that do not public modification methods.

Overriding netlist writer Old way of producing HDL sources was to give a netlist writer a reference to a block for which source file is written. In the new way, the netlist block classes have overridable *write()* function, which allows specialized netlist blocks to write HDL sources by their own.

Hierarchical netlist structure In principle the old Netlist module could model hierarchical block structures - blocks that have sub-blocks that have sub-blocks and so on. However, the bookkeeping of connections between ports in design wide Netlist object was complicated for netlist writers. To support hierarchical block structures in minimal changes, each netlist block has its own Netlist object for bookkeeping its internal connections.

Signal semantics Netlist ports can have a *Signal* object, which gives the signal semantics of the port. The *Signal* object attaches additional information to the port such as its type and active state. This is used, for example, to automatically connect clock and reset ports in netlist blocks.

Port groups A feature for grouping netlist ports were added to compose bus interfaces. The grouping is done with *NetlistPortGroup* class, which has methods for adding ports into the group. Like netlist ports, the netlist port groups can also have semantics through a *SignalGroup* class. This can be used to define rules how two port groups, that are either same or different types, are connected together.

5. CONFIGURABLE LOOP BUFFER

A configurable *loop buffer* is implementation of software-managed, level 0 instruction cache. The loop buffer allows TTA machine to store sequence of instructions in its internal buffer and repeat them as many times as required.

The loop buffer is part of instruction memory hierarchy customization. However, the loop buffer is separated to its own chapter, because it is instantiated differently than the rest.

First, in this chapter, operation principle of loop buffer is explained followed by its utilization in a TTA processor design and program code. Finally, implementation details are discussed.

5.1 Setup Operation and Operation Principle

The implemented loop buffer is software-managed and it does not act by its own. A control mechanism is provided to use the loop buffer. In programs the loop buffer is invoked by a control-flow operation called *lbufs* which stands for **l**oop **buf**fer **s**etup and it is included in OSAL module *base*. The operation takes two input operands: a *loop body size* and *iteration count*. The loop body size tells the size of the instruction block included in looping and the iteration count tells how many times the block is repeatedly executed (Figure 5.1, a). All input values of zero or above are valid.

Execution of the *lbufs* operation starts looping a block of instructions using the loop buffer. The instructions accounted into loop body are instructions next after the operation. However, if a TTA design has delay slots, then the accounted loop body is after them. When execution reaches the defined loop body, the loop buffer starts caching the instructions in the first iteration. After that, the cached instruction are repeated from the buffer until iteration limit is met. While *lbufs* operation is activated no control-flow operations affecting program counter may be executed either in delay slots or loop body. Doing so causes undefined behavior. This restriction lasts until looping is done.

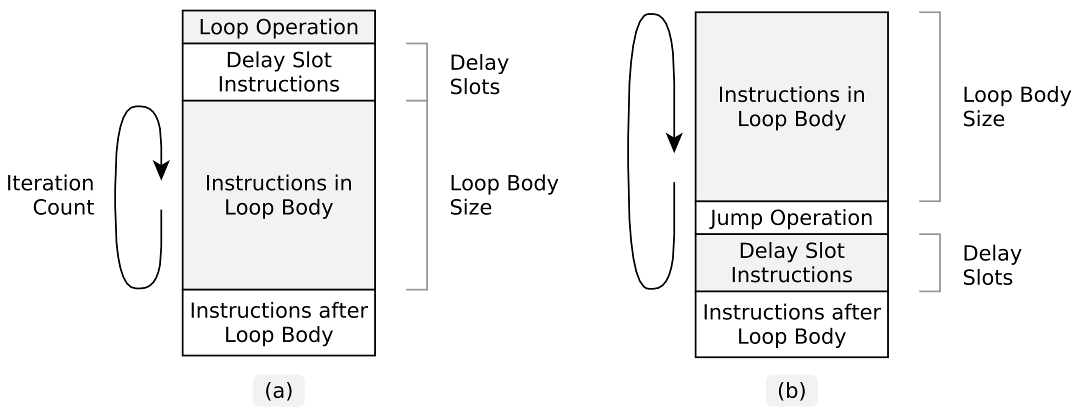


Figure 5.1 *Execution template of lbufs operation (a) and looping using jump operation for comparison (b).*

There exists a special case, where the *lbufs* operates differently. If *iteration count* is zero, then the case is interpreted as no instruction are going to be executed in a loop body. In this case the program execution jumps over the loop body.

Using *lbufs* operation for looping saves from executing delay slots that *jump* operation would have (Figure 5.1, b). A loop using *jump* operation have to execute delay slots for each iteration and this is wasteful for very small loops in size. On the other hand, *lbufs* only needs to execute delay slots once prior entering the loop. Therefore, it can save from lots of NOP instructions in some cases.

5.2 Usage in Program Code

In assembly code the loop buffer is invoked using the *lbufs* operation. In the program listing 5.1 is an example snippet of TCE assembly code utilizing the loop buffer. In the example, the TTA architecture has three delay slots and operands *iteration count* and *loop body size* are supplied via ports *iter* and *pc* of the CU. Port *pc* is the port that triggers the operations in CU.

Program 5.1 *The example invocation of loop buffer operation in TCE assembly code. The loop computes Fibonacci numbers; two numbers per one iteration.*

```
10 -> cu.iter       , 2 -> cu.pc.lbufs          ; # lbufs trigger
...                 , ...                       ; # delay slot 1
0 -> RF.0           , ...                       ; # delay slot 2
1 -> ALU.in2        , ...                       ; # delay slot 3
ALU.out1 -> ALU.in2, RF.0 -> ALU.in1t.add      ; # loop body 1
ALU.out1 -> RF.0    , ALU.out1 -> ALU.in1t.add ; # loop body 2
...                 , ...                       ; # After loop
```

In the code listing on first instruction the machine is instructed to loop two instructions after delay slots for ten times. The *iteration count* (10) is carried in the first move slot and the *loop body size* (2) in the second move slot. The move to *pc* port triggers the *lbufs* operation.

5.3 Using Loop Buffer in Design

The loop buffer is enabled and included into a TTA processor design by adding *lbufs* operation in *CU* and setting appropriate loop buffer bounds. The settable bounds are *loop-buffer-min-instructions* and *loop-buffer-max-instructions*, which are defined in ADF or alternatively in ProDe's CU dialog. The bound *loop-buffer-min-instructions* tells TCE compiler a minimum loop body size it can spawn the *lbufs* operation for. This bound does not affect handcrafted assembly code. The bound *loop-buffer-max-instructions* setting tells the maximum amount of instructions the buffer can hold at once. This directly translates into the depth of the buffer. Setting this to a higher value allows larger loops to be fitted, but in the same time increases power consumption.

5.4 Hardware Implementation

In a TTA processor core, the loop buffer is placed in the instruction pipeline between an *instruction decompression* and an *instruction decoder unit* as in Figure 5.2. The loop buffer receives control signals from *instruction fetch unit*. The whole behavior of loop buffer operation is divided among the fetch unit and loop buffer unit. The fetch unit makes checks if the operand values of *iteration count* and *loop body size* are reasonable to be used in the loop buffer unit. For example, behavior is the same as not invoking the loop buffer at all, if *iteration count* is one or *loop body size* is zero. Also, the fetch unit implements the behavior for the special case: *iteration count* of zero. Rest of the behavior of loop buffer operation is implemented in the loop buffer unit.

The implementation of the loop buffer unit was written in VHDL. The structure of the design is depicted in Figure 5.3. It consists of buffer registers that hold the instructions to be looped, multiplexers to control global lock and global lock request signals and a buffer control that controls the behavior of the loop buffer. Control interface consists of *loop length*, *iterations* and *start*. The start signal initates the loop buffer using the other two as arguments for looping.

The behavior is controlled by *look-ahead* finite state machine (FSM) [16, p. 344]

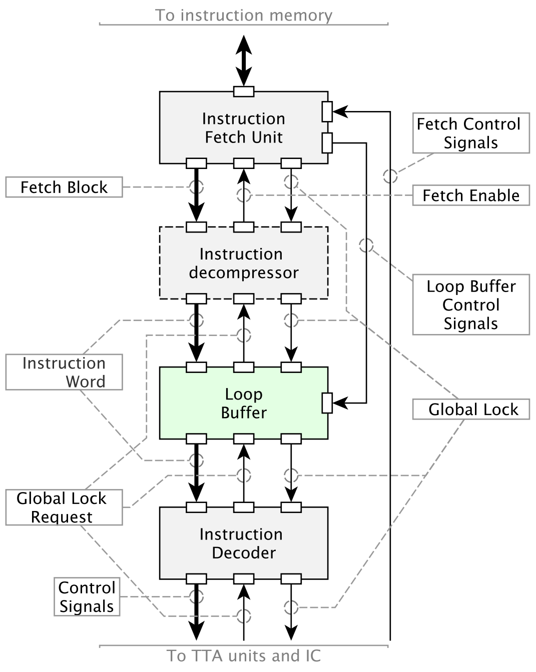


Figure 5.2 *The Loop buffer placement within processor core.*

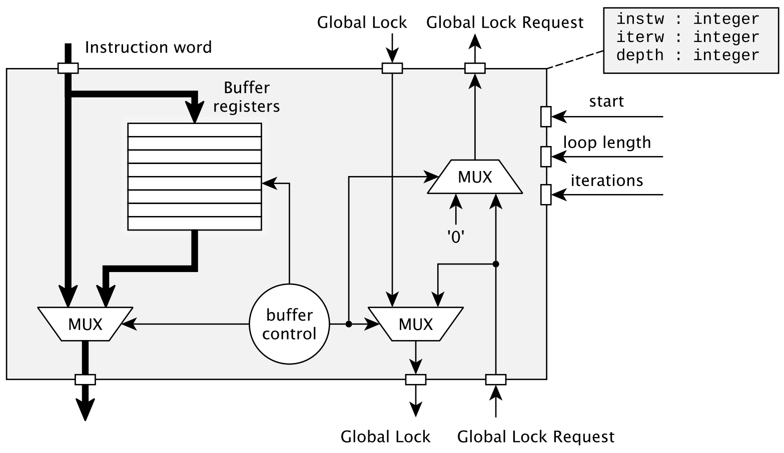


Figure 5.3 *The structure of the loop buffer.*

presented in Figure 5.4 and two counter registers - one for iteration count and another for indexing the buffer. In the FSM there are three distinctive (bolded state labels in the figure) states and other two minor states that are active during stall situations. The event *buffer filled* occurs when amount of stored instructions in the buffer matches *loop length*. The event *buffer filled* occurs after requested block of instructions have been stored and *done looping* occurs after all loop iterations have been executed.

While no looping is requested the loop buffer is inactive (idle state) and instructions

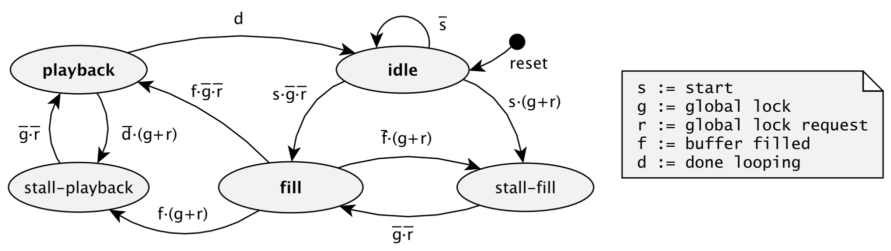


Figure 5.4 *The finite state machine of loop buffer control. The conditions in the transition edges are expressed in boolean algebra.*

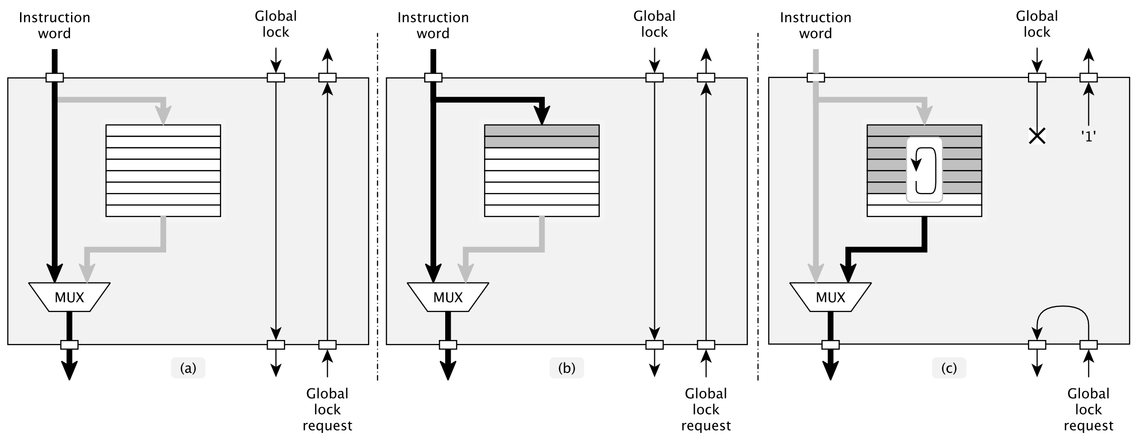


Figure 5.5 *The loop buffer behavior. (a) The loop buffer is in inactive state (idle state). (b) The first iteration of the loop occurs. The instructions are passed through and saved (gray filled boxes) for next iterations (fill state). (c) The saved instructions are repeated from the buffer. The upper part of the pipeline is locked (playback state).*

are passed through it (Figure 5.5: a). Also, *global lock* and *global lock request* signals are passed through and does not affect the loop buffer.

The request to loop block of instructions is initiated by asserting the control signal *start* with two values - *loop length* and *iterations* for duration of a clock cycle. The *loop length* signal controls the range of instructions to be looped starting from the instruction on the cycle, when the *start* was asserted. The *iterations* signal signifies loop count. The loop buffer may also be initiated by supplying either zero as loop length or zero or one as *iterations*, but the buffer will treat them as NOP. After the initiation, the control signals are ignored until the looping process is completed.

After initiation the loop buffer executes first iteration of the loop (Figure 5.5: b). During this the instruction received from the upper stages of the instruction pipeline are both passed through the loop buffer and stored in its internal buffer for subsequent loop iterations. This continues until whole body of a loop has been seen.

After first iteration of the loop, the loop buffer locks up the upper part of the instruction pipeline by asserting *global lock request* signal (Figure 5.5: c). This signal leads to instruction fetch units enable port. The signal causes *instruction fetch* to cease fetching from instruction memory and hence consumes less power. The *global lock request* from the lower part of the pipeline is looped back as *global lock* signal to preserve global lock functionality. On completion, after all iterations have been elapsed, the buffer resumes into its idle state and awaits for new initiation.

As the instruction pipeline is stalled either by a FU or the instruction memory, the loop buffer is needed to be stalled, too. Otherwise, the buffer stores duplicates of the same instructions during stall. The signals *global lock* and *global lock request* affects the loop buffer only when it is activated.

5.5 Integration

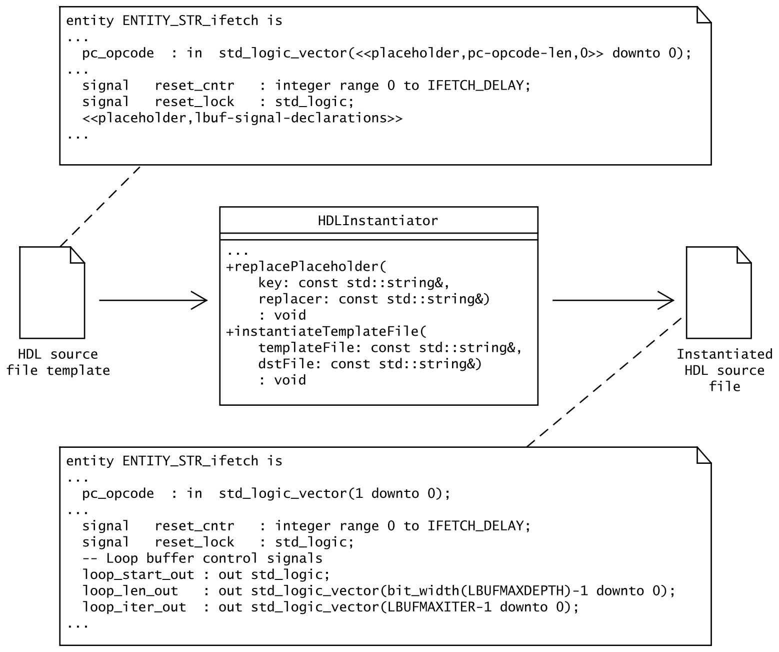
The loop buffer is automatically generated if an ADF describes a loop buffer operation within the *CU*. The process of the inclusion is carried out within default *IC/Decoder generator plug-in*.

The default ICDG plug-in makes *netlist block* model of the loop buffer instatiated within TTA core netlist block and rewires pipeline connections. Actual HDL source file of the loop buffer is copied to the target directory of the processor. Parameters of the loop buffer (buffer depth and iteration port width) are appended to TTA processor globals package.

The loop buffer is controlled by instruction fetch unit which is forefront actor of CU. The unit is also based on template file and by default does not have any control logic for loop buffer. Instead the logic is injected into the file using "placeholders" in template file.

In the Figure 5.6 is the idea of the placeholders. In a template file there is placeholders marked in pattern of <<placeholder, *key*, *default*>> or <<placeholder, *key*>>. A class *HDLInstantiator*'s function *initiateTemplateFile()* reads the file and seeks for the placeholders. For each found placeholder pattern in a file, the instantiator looks up for a entry by *key*. The placeholder pattern is replaced by the entry. If there is no entry for a placeholder, it is replaced with empty string or with a string in an optional field *default*.

Control logic of the loop buffer is injected into the instruction fetch unit like this.



```
entity ENTITY_STR_ifetch is
...
  pc_opcode  : in  std_logic_vector(<<placeholder,pc-opcode-len,0>> downto 0);
...
  signal   reset_cntr   : integer range 0 to IFETCH_DELAY;
  signal   reset_lock   : std_logic;
  <<placeholder,lbuf-signal-declarations>>
...
```

```
HDLInstantiator
...
+replacePlaceholder(
    key: const std::string&,
    replacer: const std::string&)
    : void
+instantiateTemplateFile(
    templateFile: const std::string&,
    dstFile: const std::string&)
    : void
```

```
entity ENTITY_STR_ifetch is
...
  pc_opcode  : in  std_logic_vector(1 downto 0);
...
  signal   reset_cntr   : integer range 0 to IFETCH_DELAY;
  signal   reset_lock   : std_logic;
  -- Loop buffer control signals
  loop_start_out : out std_logic;
  loop_len_out   : out std_logic_vector(bit_width(LBUFMAXDEPTH)-1 downto 0);
  loop_iter_out  : out std_logic_vector(LBUFMAXITER-1 downto 0);
...
```

Figure 5.6 *An example of HDL source template file instantiation with HDLInstatiator object.*

Interface of the unit is altered by adding new ports for the operand port added in the CU and for control signals of the loop buffer. The netlist block representation of the unit is updated according to this.

The *lbufs* operation is defined in OSAL's base module. Operands of *lbufs* operation are mapped as followed: operand 1 is used for *iteration count* and operand 2 is for *loop body size.*

6. INSTRUCTION MEMORY HIERARCHY GENERATION

The loop buffer is automatically generated in TTA designs. However, the customization of instruction memory hierarchy, its structure and its parameters, are defined by the user

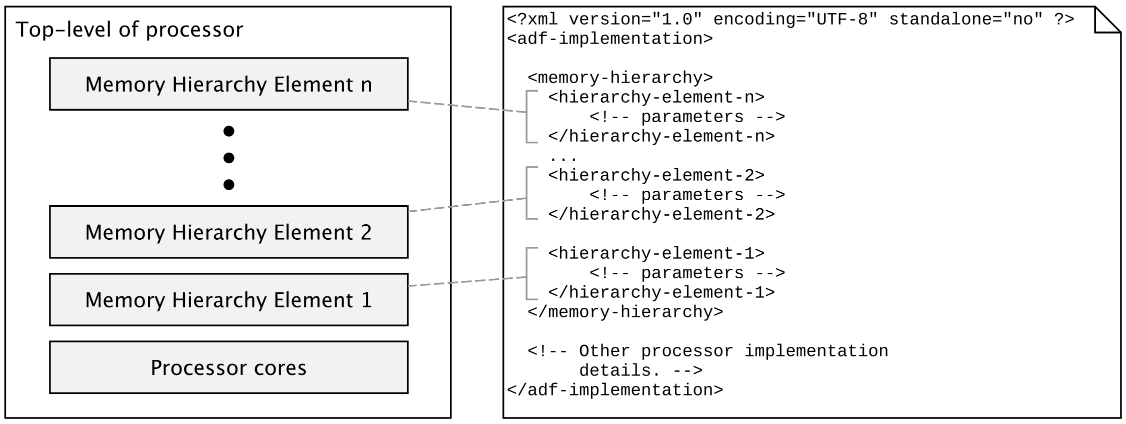
The chapter is divided as follows: First is explained how instruction memory hierarchy is customized for a TTA processor and how processor generator proceeds to create it. Next, available parts of the hierarchy, a *level 1 instruction cache* and an *instruction bus arbiter*, are introduced followed by their implementation and integration details. Finally, a method of cache performance statistics collection is introduced for the generated hierarchies, including the loop buffer.

6.1 Instruction Memory Hierarchy Customization

The instruction memory hierarchy is described in IDF. The idea of hierarchy description is depicted in Figure 6.1. The memory hierarchy structure is defined inside of *memory-hierarchy* element and it consists of *n* hierarchy elements. The bottom-most element connects to processor and to upper element. The top-most element connects to the ports of the processor.

Interfaces between the elements are not defined in the hierarchy elements. Instead, they are implicitly inferred by their type and parameters. For example an arbiter element would combine interfaces from a below element and reveal a single interface to an upper element.

In Figure 6.2 is a concrete example of the memory hierarchy description. In the example, a TTA processor has four cores and in IDF of the processor is memory hierarchy definition consisting of level 1 instruction caches and an arbiter. Since there are multiple cores in the processor, an own level 1 instruction cache is instantiated for each core. If another element, like a cache, has been added on the top of an arbiter, it would be shared, since the arbiter reveals single interface. However,


Top–level of processor
Memory Hierarchy Element n
Memory Hierarchy Element 2
Memory Hierarchy Element 1
Processor cores

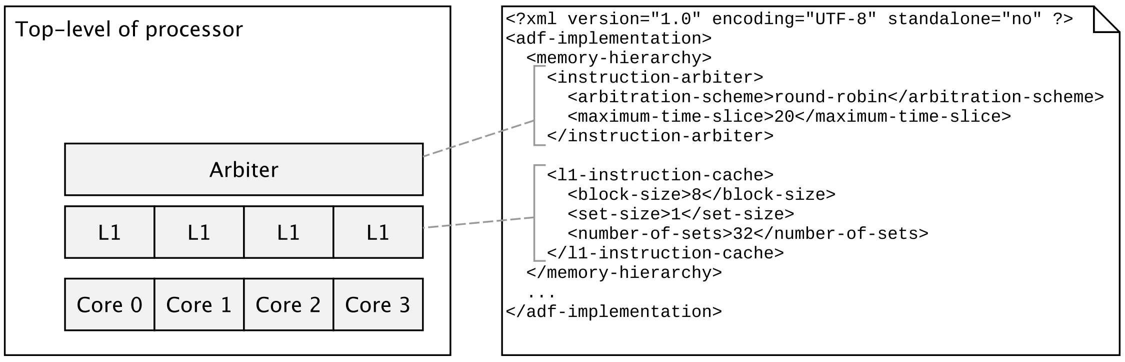
```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<adf-implementation>

  <memory-hierarchy>
    <hierarchy-element-n>
        <!-- parameters -->
    </hierarchy-element-n>
    ...
    <hierarchy-element-2>
        <!-- parameters -->
    </hierarchy-element-2>

    <hierarchy-element-1>
        <!-- parameters -->
    </hierarchy-element-1>
  </memory-hierarchy>

  <!-- Other processor implementation
       details. -->
</adf-implementation>
```

Figure 6.1 *The concept of memory hierarchy definition in IDF. In the right is the IDF having a memory hierarchy description and in the left is depicted logical structure modeled by it.*



```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<adf-implementation>
  <memory-hierarchy>
    <instruction-arbiter>
      <arbitration-scheme>round-robin</arbitration-scheme>
      <maximum-time-slice>20</maximum-time-slice>
    </instruction-arbiter>

    <l1-instruction-cache>
      <block-size>8</block-size>
      <set-size>1</set-size>
      <number-of-sets>32</number-of-sets>
    </l1-instruction-cache>
  </memory-hierarchy>
  ...
</adf-implementation>
```

Figure 6.2 *An example of a memory hierarchy definition.*

the initial implementation of hierarchy generation is limited to three configurations: one depicted in the figure, only L1 caches for each core and arbiter solely.

The two elements that can be included in instruction memory hierarchy are *level 1 instruction cache* and *instruction bus arbiter*. Each one of them are customizable. Parameters for the instruction cache and their use are listed below:

- *block-size.* The value defines how many **instructions** can be fit into a block at minimum.
- *set-size.* This sets associativity of cache.
- *number-of-sets.* This is number of blocks in each set.
- *replacement-policy.* Selects the replacement policy to be used in the cache. Parameter is effective only if *set-size* is more than one. Available policies are

random and *LRU*.

All the parameters are mandatory except the *replacement-policy*. The replacement policy can be omitted for associative caches, in which case the policy is inferred automatically. Values set for *block-size* and *set-size* parameters must be in power-of-two (2^n). There can be additional restrictions due to underlying implementation of a cache. In such case, the ProGe informs, if a parameter setting is not valid or feasible. Total size of the cache in terms of the data storage depends on parameters *block-size*, *set-size*, *number-of-sets* and it is calculated as in equation 6.1.

$$\textit{total-cache-size} = \textit{instruction-size} \cdot \textit{block-size} \cdot \textit{set-size} \cdot \textit{number-of-sets} \tag{6.1}$$

Parameters of the instruction bus arbiter are listed below:

- *arbitration-scheme* defines arbitration scheme.
- *maximum-time-slice* defines how long a requestor can keep control to the bus at maximum. This parameter is effective only for arbitration schemes having time slots.

HDL description of the instruction memory hierarchy is generated by ProGe using the description in IDF. The validity of the hierarchy structure and parameters of its elements are checked. After that, the processor generator proceeds to construct the defined memory hierarchy by forming netlist models of the hierarchy elements, which are placed to appropriate locations in the processor netlist model. For example, level 1 cache is placed into a processor core block and bus arbiter is placed into a block, which wraps all processor cores in a TTA design.

6.2 Instruction Cache

The ProGe instantiate level 1 instruction cache within TTA core based on description in IDF. The cache implementation used in this thesis project is modified Level 1 Instruction Cache made by VLSI Research Group [17]. The cache source files are written in VHDL and they are available at [18]. The instruction cache is widely configurable through its parameters. Configuration parameters in the cache are:

- instruction size in bits,
- block size in number of instructions,
- associativity,
- numbers of sets,
- replacement policy available with pseudo-random and LRU policy,
- width of block data memories and
- data width separately for processor core and instruction memory bus.

Parameter values for setting *block size* and *numbers of sets* may only be in power-of-two. Using any other value may lead to erroneous design that may not cause error in a HDL compiler. However, the ProGe prevents if one attempts to set invalid values for the parameter in IDF.

Parameter *set size* defines the associativity of the cache. Setting this to one makes it direct-mapped and higher than one makes it associative. The *set size* is limited up to four when using LRU policy. However, using *pseudo-random* policy the degree of associativity can be set to higher value.

Blocks in the cache are divided and stored among one or more memory blocks depending on the set block size and width of the memory blocks. In the ProGe, the width is set automatically.

The cache has memory bus interfaces - one towards the processor and another towards the memory. Both the interfaces have parametrizable address and data widths, which can be set separately at each side of the interfaces. This means, that the instruction bus to the memory can be width of the block or can be smaller than the instruction size. However, the widths of data signals for the processor and the instruction memory may not be arbitrary, but instead they must be in power-of-two in relation to each other. Also, the protocol of the processor and memory interfaces are similar to each other meaning, that multiple cache instances could be stacked for making multi-level cache hierarchies. However, the cache model is meant to be used as level 1 cache. In the ProGe, all caches are configured to have same address and data signal widths in both interfaces.

6.2.1 Implementation Details

Top-level structure of instruction cache is depicted in Figure 6.3. Tag and block data are stored and tag matching is done in *cache memory* units. There is as many units as there is sets in the cache. *Cache controller* handles misses, block fetching from memory and invalidation. *Replacement policy* unit, included in associative cache configurations, points a cache memory unit, where a block (re)placement occurs. All configuration parameters of the cache are placed in a package file. ProGe generates this file form IDF and names it as *<processor-entity-name>_l1_parameters_pkg.vhd.*

Instruction memory bus interfaces are at the bottom and top of the module. On the side there are control and status signals. The ports beside the invalidate port are not used in the caches instantiated by the ProGe.

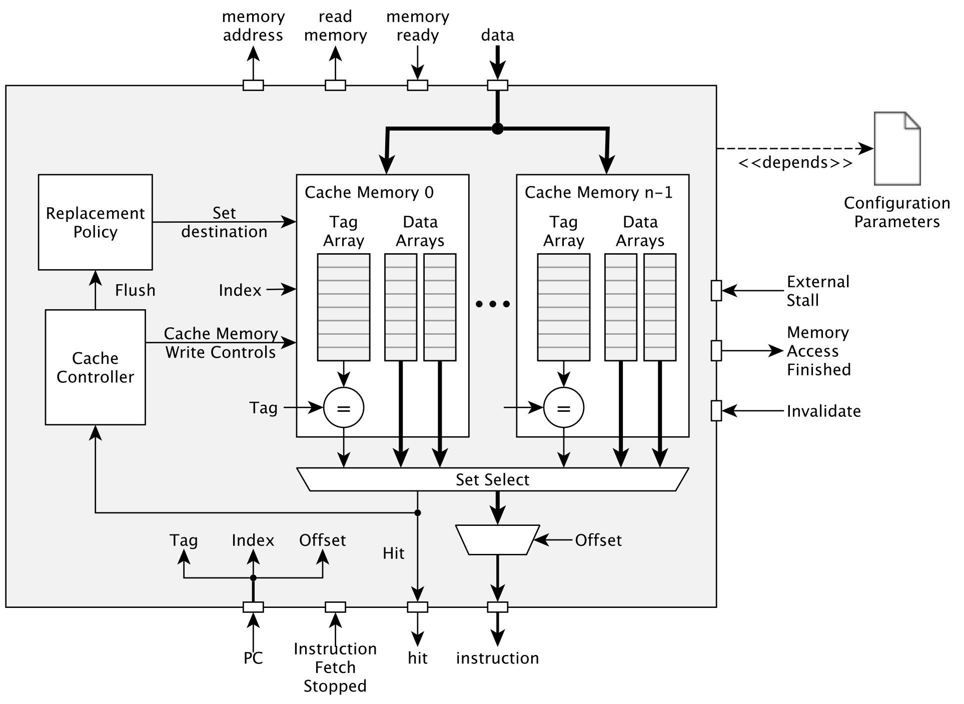


Figure 6.3 *The structure of L1 instruction cache template*

In cache memory module there is separate memory block, which is solely for storing tag data. One entry in it holds a tag part of the address and a valid bit for a block it is associated. The Block data is divided among one or more of memory blocks. All the memory blocks have single port memory interface. Also, data that is written into a memory is visible at output port in the same cycle for reading. All memory blocks in the cache are implemented as register-array by default but they can be replaced with other implementation easily.

The *Cache Controller* manages writing to cache memories and block fetching during cache misses. A FSM of the controller is depicted in Figure 6.4. Initially after reset lift-off the controller initializes the cache memories by deasserting all valid bits (*invalidate* state in 6.4). This process takes as many cycles as there is indexable entries in the cache, so it is O(*number-of-sets*).

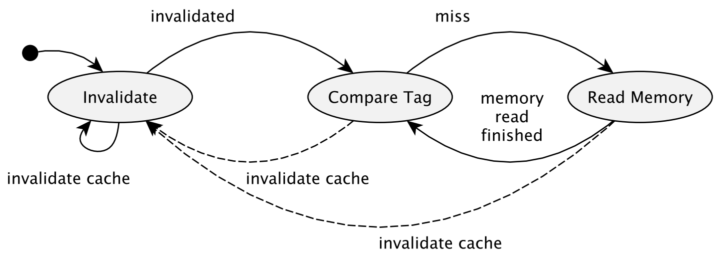


Figure 6.4 *The modified control FSM of level-1 instruction cache template. The dashed arrow line is new addition to the original FSM of the cache [17].*

After initialization the controller stays idle in *compare tag* state until a miss occurs or is commanded to invalidate entries. On miss the controller generates memory accesses to retrieve a block from memory. The memory access addresses are calculated as in equation 6.2.

$$Addr_{imem} = \frac{Addr_{proc}}{\mathit{offset\text{-}bits}} \cdot \left\lfloor \frac{\mathit{block\text{-}size}}{\mathit{imem\text{-}width}} \right\rfloor + i \mid i \in \left[0, \left\lfloor \frac{\mathit{block\text{-}size}}{\mathit{imem\text{-}width}} \right\rfloor\right] \tag{6.2}$$

where $addr_{proc}$ is the address from the processor and $addr_{imem}$ is the memory address for instruction memory. i is counter value in controller for generating enough memory accesses to cover a block.

The controller only generates lower part of the addresses (i in the equation) and they are combined with block start address calculated in the top-level. During a cache miss the last processor address is held until block is retrieved from memory.

An externally controllable cache invalidation feature, which was not in the original cache design, was added. A use case of the feature is in the program loading, where the cache invalidation is needed for discarding blocks that belongs to a old program. The FSM diagram 6.4 shows that cache can be invalidated at any time. The invalidation is triggered by having the invalidation signal (on side in Figure 6.3) asserted for at least one clock cycle. Additionally, the cache can be kept locked and in the invalidated state as long the signal is held asserted.

In associative cache configurations the *replacement policy* unit selects a cache memory unit, in where a retrieved block is placed during miss. Effectively this directs write control signal, from the cache controller, to selected unit. Of the two available replacement policies, *pseudo-random* policy is implemented using *linear-feedback shift register* (LFSR) that changes its state in every clock cycle, and the other *LRU*'s functionality is based on look-up tables. When using LRU replacement policy, the set size parameter is restricted to maximum of four due to predefined tables that only covers set sizes ranged from 1 to 4. Higher set configurations can be supported by adding new tables. Inspecting cache's sources reveals that the order of growth in area for LRU tables is $O(n!)$ where n is *set size*. The *pseudo-random* policy does not have the restriction on the *set size*.

6.3 Instruction Arbiter

In the Figure 6.5 is the top-level structure of the implemented *instruction bus arbiter*. It consists of a single *arbitration scheduler*, *instruction bus guard* for each processor core and a network called as *merge*. The *arbitration scheduler* defines arbitration scheme of the arbiter.

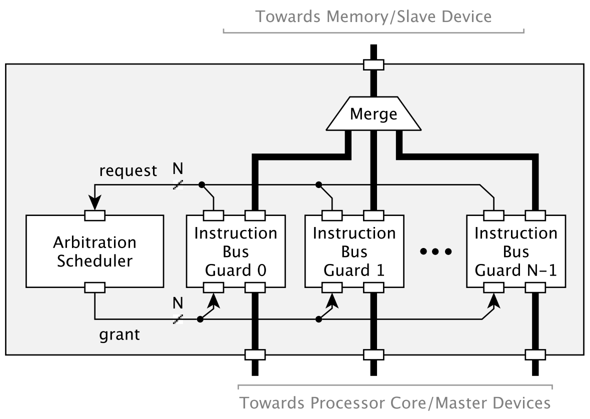

Towards Memory/Slave Device
Merge
request
N
Arbitration Scheduler
Instruction Bus Guard 0
Instruction Bus Guard 1
•••
Instruction Bus Guard N-1
N
grant
Towards Processor Core/Master Devices

Figure 6.5 *The structure of instruction bus arbiter.*

Arbitration Scheme Instantiation

Arbitration scheme is determined by the selected implementation of the *arbitration scheduler*. Currently only one arbitration scheme is available and that is a simple *round-robin*. The *arbitration scheduler*'s responsibility is to assign control to a shared bus among the requesting processor cores.

The scheduler receives access requests indirectly through so called *instruction bus guard* and decides who to give the control. The one given the control is signaled via *grant* signal through the guard. The scheduler assumes that the master currently having the control keeps the request line active as long it needs a shared resource.

The available, implemented *round-robin* scheme is simple by its design. However, some cycles may be wasted while switching the bus control to a next requestor. Switching the control to next requestor is carried by iterating over the requestors in order starting from next after current in control. At every clock cycle, the scheduler checks, if a master is requesting, and then moves on to next one. This is repeated at every clock cycle until a next requestor is met, in which case the control is given to it. In worst case, the number of cycles wasted is $O(n)$ where n is number of requestors.

Instruction Bus Guard

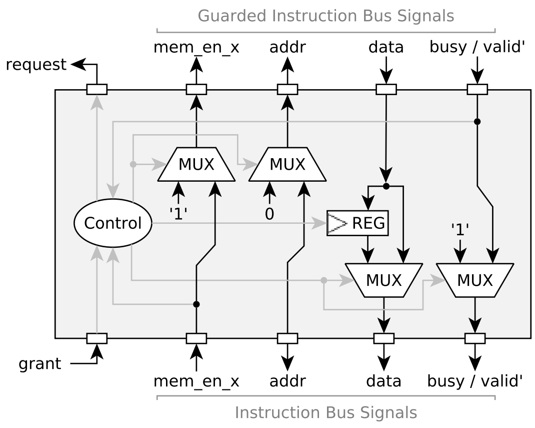


Figure 6.6 *The structure of instruction bus guard block.*

The *instruction bus guard* acts as a gate between the requestor and the shared resource or bus. It lets signals to be passed through only when the requestor is given control, signaled by the arbitration scheduler. In opposite case, where a requestor does not have control, its *guard* signals core telling the bus is in busy state and seals off the signals from the core towards the shared bus. In the sealed state the guard sets signals toward shared bus in inactive or neutral state. That is, in example a active-low signal is set to high and signals of a address is set to all zero (neutral state). This is done due to *merge* network which is explained later.

Actual implementation of the instruction bus guard is presented in Figure 6.6. This implementation is non-restarting: A memory read access made by a requestor is not needed to be repeated later, if the control switch occurs in the middle of an access. For example, assuming a situation, where requester has made a memory read access to a non-busy memory at a cycle, and then the control is switched to another requestor at the next cycle. The memory responds to the read access in a cycle, but since control was switched, the original requestor can not receive it. Instead, the bus guard will temporarily hold the read data for the requestor until the control is granted again.

Merge Network

The *merge* network combines all bus signals from each guard into one bus. The merge network can be thought as a multiplexer, but where all buses are selected, and the actual implementation is simply a mixture of straight wires and or- and and-reduce networks. However, only one bus guard is active at a time and the others do not cause interference because they have their signals set to inactive or neutral state. The shared bus signals coming inward are routed directly to each guard and, thus, each guard can listen activity of the shared bus even when a guard does not have a control on the bus.

6.4 Integration into ProGe

In the Figure 6.7 is a partial diagram of netlist block hierarchy of a TTA processor that is in this case a multi-core design. The Figure shows the placements of instruction memory elements in the netlist model of a TTA processor. The highlighted blocks mark the hierarchy elements and a test bench for multi-core processor which is discussed later in Chapter 7.

As in the Figure 6.7 the loop buffer and level 1 instruction cache is instantiated in TTA core netlist block. The instruction bus arbiter is instantiated within multi-core netlist block.

Integration of Instruction Cache

In the Figure 6.8 is Netlist Block *FlexSoCCache* class presentation of the level 1 instruction cache model introduced earlier in section (6.2). The class is derived from abstract base class called ICacheBlock that is base for all instruction cache implementations.

Derived classes are required to implement functions that provide interfaces of the cache. Functions *coreSideInterface()* and *memorySideInterface()* provide references

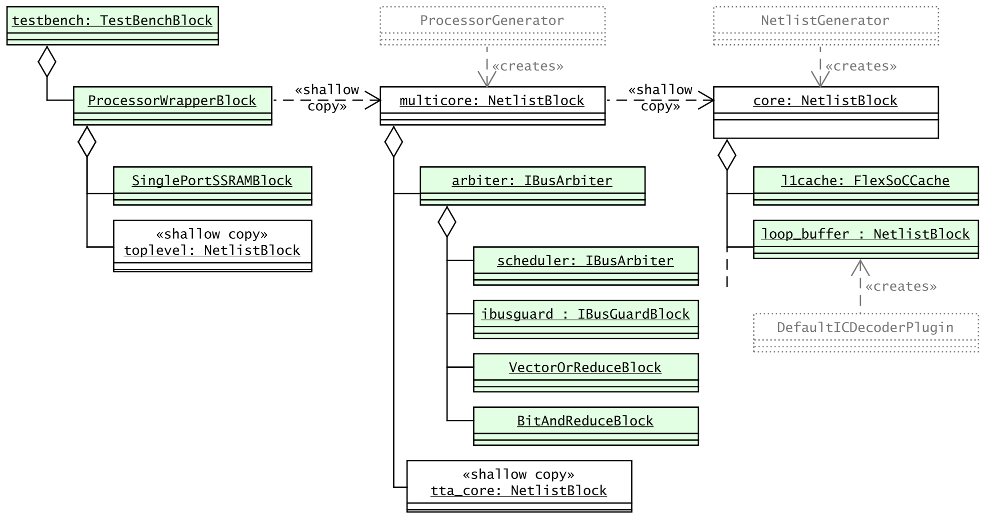


Figure 6.7 *The partially visualized NetlistBlock composition hierarchy of multicore TTA processor with testbench generation. Netlist blocks related to memory hierarchy and test bench are highlighted. Netlist blocks for TTA Units are left out.*

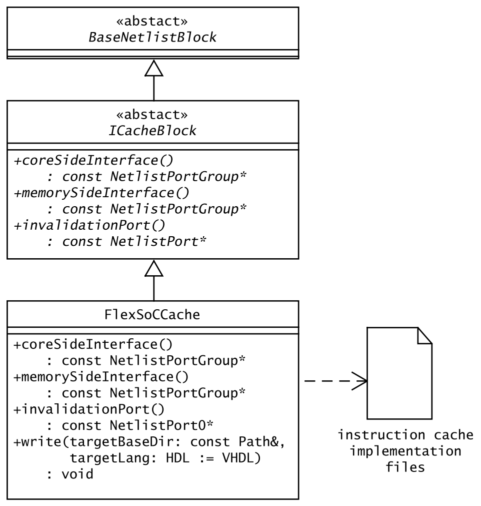


Figure 6.8 *The NetlistBlock class presentation of instruction cache consisting of one abstract class for all instruction caches and one concrete implementation of it*

to port groups for instruction and memory buses. A function *invalidationPort()* returns port for controlling invalidation of cache entries if a cache implementation has one.

The FlexSoCCache instance is built from ProGeContext that holds implementation parameters for the cache. The L1 cache parameters itself are packed in *L1CacheParameters* object that is a presentation of the level 1 instruction cache parameters defined in

IDF.

The FlexSoCCache overrides function *write()* of its base class. The function writes a file called *<processor-entity-name>_l1_parameters_pkg.vhd* holding cache configuration parameters and copies HDL source files of the cache to the output directory of the processor.

Integration of Instruction Bus Arbiter

A netlist model representation of instruction bus arbiter is depicted in Figure 6.9. Class *IBusArbiter* represents top-level netlist block of the arbiter. Its constructor takes interface of a shared resource, a number of devices accessing the shared resource and arbitration scheme as parameters. Based on the interface of the shared resource the *IBusArbiter* instantiates proper sub-instances that can handle the protocol of it. However, currently only one interface is supported.

Classes *ArbiterSchedulerBlock* and *IBusGuardBlock* correspond to *arbiter scheduler* and *instruction bus guard* concepts mentioned in section 6.3. Instances of classes *BitAndReduceBlock* and *VectorOrReduceBlock* are used to construct the merge network. Upon construction of the blocks they have no ports. Ports are dynamically created as needed and connected when functions *hookInput()* and *hookOutput()* are called.

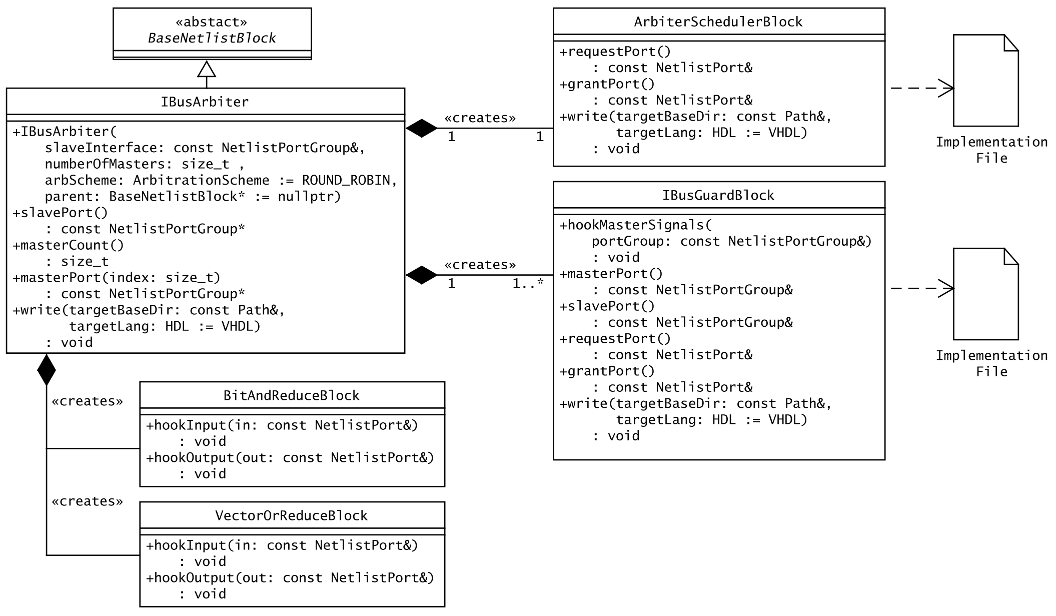

«abstact»
BaseNetlistBlock
IBusArbiter
+IBusArbiter(
slaveInterface: const NetlistPortGroup&,
numberOfMasters: size_t ,
arbScheme: ArbitrationScheme := ROUND_ROBIN,
parent: BaseNetlistBlock* := nullptr)
+slavePort()
: const NetlistPortGroup*
+masterCount()
: size_t
+masterPort(index: size_t)
: const NetlistPortGroup*
+write(targetBaseDir: const Path&,
targetLang: HDL := VHDL)
: void
«creates»
1
1
«creates»
1
1..*
ArbiterSchedulerBlock
+requestPort()
: const NetlistPort&
+grantPort()
: const NetlistPort&
+write(targetBaseDir: const Path&,
targetLang: HDL := VHDL)
: void
Implementation File
IBusGuardBlock
+hookMasterSignals(
portGroup: const NetlistPortGroup&)
: void
+masterPort()
: const NetlistPortGroup&
+slavePort()
: const NetlistPortGroup&
+requestPort()
: const NetlistPort&
+grantPort()
: const NetlistPort&
+write(targetBaseDir: const Path&,
targetLang: HDL := VHDL)
: void
Implementation File
«creates»
BitAndReduceBlock
+hookInput(in: const NetlistPort&)
: void
+hookOutput(out: const NetlistPort&)
: void
«creates»
VectorOrReduceBlock
+hookInput(in: const NetlistPort&)
: void
+hookOutput(out: const NetlistPort&)
: void

Figure 6.9 *The Netlist block class representation of instruction bus arbiter.*

6.5 Implementation of Cache Statistics Collection

It is essential to know how well the generated instruction memory hierarchies performs. Therefore, a method and a tool is provided to collect cache statistics from RTL-simulations.

The tool to report cache statistics along with other information is a terminal application called *rtlstats*. In the listing below is an example report of *rtlstats* tool. It lists all cache units in a processor and, for each of them, reports the total of handled memory accesses, miss-rate and AMAT. The tool also shows the total number of the final instruction memory accesses making out of the processor. To get the report out with *rtlstats*, first, a program is run in RTL-simulation using processor test bench. Then the report is printed on terminal screen by running the *rtlstats* in the directory, where the processor test bench was ran, or specifying the directory with *-x <directory>* option.

```
Run statistics (Core id|Total cycles|Exec cycles|Lock cycles|Lock Rate):
     core 0 |      26713 |      23845 |       2868 |      10.7 % |

Cache statistics:
core 0 level 1: Total accesses: 23848      Miss rate: 1.32 % AMAT: 1.12022

Memory statistics:
        core 0: Total accesses: 2520
```

The other information the *rtlstats* tool reports are total simulated cycles, executed instructions and stall cycles for each processor core.

The reports produced by *rtlstats* are compiled from event data, which are generated during RTL-simulation. The event data are written in so called “dump files”. Some of the dump files, for cache statistics, are *access traces*, which tell about memory access events made in caches and memories in the processor. The access traces are written for each instruction cache and memory unit in the processor for a single simulation run. The access trace files are named as:

- core<id>_l<level>_access_trace.dump for instruction caches
- core<id>_imem_access_trace.dump for instruction memories

The *id* denotes core-id and *level* denotes cache level.

```
           0 |         0000 |               0 |              38 |
          38 |         0000 |               1 |               1 |
          39 |         0001 |               1 |               1 |
          40 |         0002 |               1 |               1 |
          41 |           0003 |              1 |              1 |
          42 |           0004 |              0 |              6 |
          48 |           0005 |              1 |              1 |
          49 |           0006 |              1 |              1 |
          50 |           0007 |              1 |              1 |
          51 |           0008 |              0 |              6 |
```

In the listing above is a snippet of access trace produced from RTL-simulation. From left to right are columns for *cycle number*, *memory address*, *presence* and *access delay*. The *cycle number* tells a cycle when a cache or memory unit accepts to commit a memory access. The *memory address* of the access is displayed in hexadecimal.

The *presence* column tells with a number if a accessed data is present in a device. The number in the column is either zero or one which, respectively, means a hit or a miss in a cache. For non-caching units value is always expected to be one.

The *access delay* tells how many cycles it takes for retrieving an accessed data. The time is measured in cycles where value $n \in \mathbb{N}$ says that accessed data is available after n cycles. A zero delay is also valid and it would mean that data is asynchronously available at the same clock cycle as the memory access is made.

7. VERIFICATION

In this chapter, an automated test program generator for verification purposes is introduced. Then, the improvements made to the TTA processor test bench are discussed. After that, common testing methods and test environment used for verification are explained. Finally, the verification of the loop buffer and the instruction memory hierarchy is carried out and the results are presented.

7.1 Automated Test Generator

Automated test generator (TeGe) is a TCE tool that automatically generates test cases to stress various parts of TTA processors. A case of the TeGe is to generate a set of test cases, which are ran in the RTL-simulation of the processor. Finally, theresults from the simulation are compared to TeGe's verification reference data to see if test cases did pass.

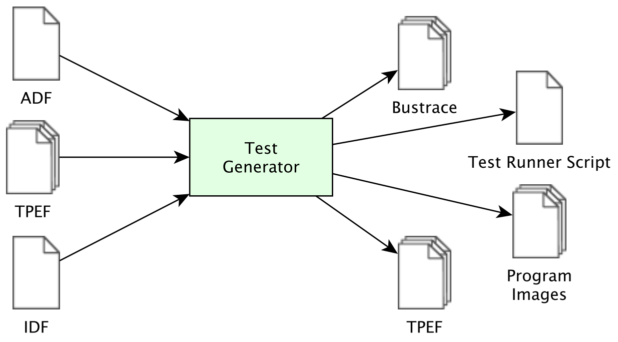


Figure 7.1 *The Test Generator tool.*

In the Figure 7.1, the TeGe tool takes at least an ADF as input and optionally other files such as an IDF and additional user created programs in the TPEF format. Using solely the ADF, the TeGe can generate test cases that only cover the architectural aspects of the processor and, thereby, excludes test cases that could stress micro-architecture features. The test case generation for micro-architectures is enabled by passing the IDF to the TeGe. Optionally, user created test programs can be fed to the TeGe for generating verification reference data automatically on behalf of the user.

With the given input files, TeGe produces test cases, which consist of *test programs* in different formats and *verification reference data.* The test program format includes *TCEASM* (TCE assembly code), *TPEF* and *program images.* The program images are used to run test cases in RTL-simulation. The TCEASM is for inspecting the generated test programs visually. The TPEF is for debugging in the TTA-simulator in case a test case fails in a RTL-simulation. For example, if a test case fails, user can load the failed test case (the file in the TPEF format) into the TTA-simulator and run the test case program until to the point, where the RTL-simulation failed.

Many of the tests, created in the TeGe, are pseudo-random, but the generation is deterministic. That is, giving the same inputs to the TeGe, it should always produce the identical test case set. However, the test cases can be varied by defining a seed value for the pseudo-random generation.

The *test runner* (testrunner.py) script is an utility, which runs all the created test cases (in program image format) in the RTL-simulation and reports whether the test cases do pass or not. The current script is made for the processor test bench generated by the ProGe.

7.1.1 Usage

An sample usage of the TeGe tool is shown below.

```
$ generateprocessor -t -i foo.idf foo.adf -o proge-out
$ generatebits -d -w4 -x proge-out/
$ generatetests -s someseed -a foo.adf -i foo.idf -o tests
$ ./tests/testrunner.py -i tests/ -x proge-out
    OK ALU-operation-tests
    OK icache-test
    OK mul-operation-tests
```

With a ProGe, a TTA processor is generated with processor test bench, which is needed for test runner script. The PIG is ran once to create necessary files for the freshly generated processor. Test cases are created by running test generator by giving ADF and IDF of the processor. The created test cases and test runner script are outputted under *tests/* directory. Executing the test runner script run the test cases in RTL-simulation and their success is reported in terminal screen. The options *-i* and *-x* specifies directory locations of the test cases and the processor respectively. The option *-s* defines the seed value used in pseudo-random generation and it can be numeric value or string. There are more options in TeGe, which can be listed by giving *-h* option.

The TeGe creates test cases for FU operations found in the processor and, by default, it creates tests for the operations by generating random inputs. However, this procedure is not suitable for all operations. For example, a division operation may not have zero value as divisor. Situations like this can be prevented by specifying input validation for operations in OSAL behavior model. In the listing 7.1 is example, where input validation is defined for modulo operation, which takes two operands.

Program 7.1 *A example of input validation definition in OSAL for MOD operation*

```
OPERATION(MOD)
    ...
    INPUT_VALIDATION
        if (UINT(2) != 0) {
            DECLARE_VALID;
        }
    END_INPUT_VALIDATION

END_OPERATION(MOD)
```

In the listing, within an operation behavior definition, is an input validation clause. The TeGe suggests for the operation input values, which are inspected using OSAL's operation input accessor macros (for example the UINT in the listing). For the modulo operation, only the divisor value is needed to be validated and any value for dividend is applicable. If the suggested values are valid, the TeGe is informed by using *DECLARE_VALID* macro. If the macro is not executed, then the TeGe treats the suggested values as invalid and discards them.

TeGe's input generation can be overrode in OSAL by defining custom input values that are suitable for an operation. The beneficial example cases for using this feature are:

- input values that test corner cases of the operation and
- input values that contributes to improve HDL code coverage.

In the listing 7.2 is an example of custom generation of sets of input values (or *test vectors*) for an operation. The test vector, which holds all input values for an operation, is formed by setting the input values using accessor *IO* macros and then accepting them by using *ADD_TESTVECTOR* macro. In the example, *TESTVECTORCOUNT* and *SEED* are optional macros. The former tells the amount of test vectors asked by TeGe. The latter macro provides a seed value for random number generator, so the TeGe can generate the same set of test vectors again between two launches.

Program 7.2 *An example about test vector generation in OSAL for SHR operation.*

```
OPERATION(SHR)
    ...
DEFINE_TESTVECTORS
    RandomNumberGenerator rng(SEED);
    UIntWord bitwidth = MIN(
        static_cast<SIntWord>(BWIDTH(1)),
        static_cast<SIntWord>(OSAL_WORD_WIDTH));
    for (int i = 0; i < TESTVECTORCOUNT; i++) {
        IO(1) = rng();
        IO(2) = rng() % bitwidth;
        ADD_TESTVECTOR;
    }
END_DEFINE_TESTVECTORS
    ...
END_OPERATION(SHR)
```

In the table 7.1 is listed TeGe related macros and their usage and effect. Other OSAL's macros and overall usage of OSAL are explained in TCE-manual [19].

Table 7.1 *OSAL keywords and commands for operand input validation and test vector generation.*

Keyword	Description
INPUT_VALIDATION	Starts input validation clause for a operation.
DECLARE_VALID	Using the keywords accepts the current operand inputs valid. Usable only in input validation clause.
END_INPUT_VALIDATION	Ends input validation clause. If DECLARE_VALID has not met the inputs are treated as invalid.
DEFINE_TESTVECTORS	Starts test vector generation clause for a operation.
SEED	Seed value for random number generator. Usable in test vector generation clause.
TESTVECTORCOUNT	Optional. Tells the amount of test vectors requested by TeGe. However, it does not need to be exact.
ADD_TESTVECTOR	Appends a test vector using current IO input operands in it. Usable only in test vector generation clause.
END_DEFINE_TESTVECTOR	Ends test vector generation clause.

7.1.2 Test Generation Framework

The inner work of test generator is depicted in the Figure 7.2 that is a class diagram of the TeGe's framework. The classes can be roughly divided into three responsibilities: test case generation, test input generation and validation and test case file generation.

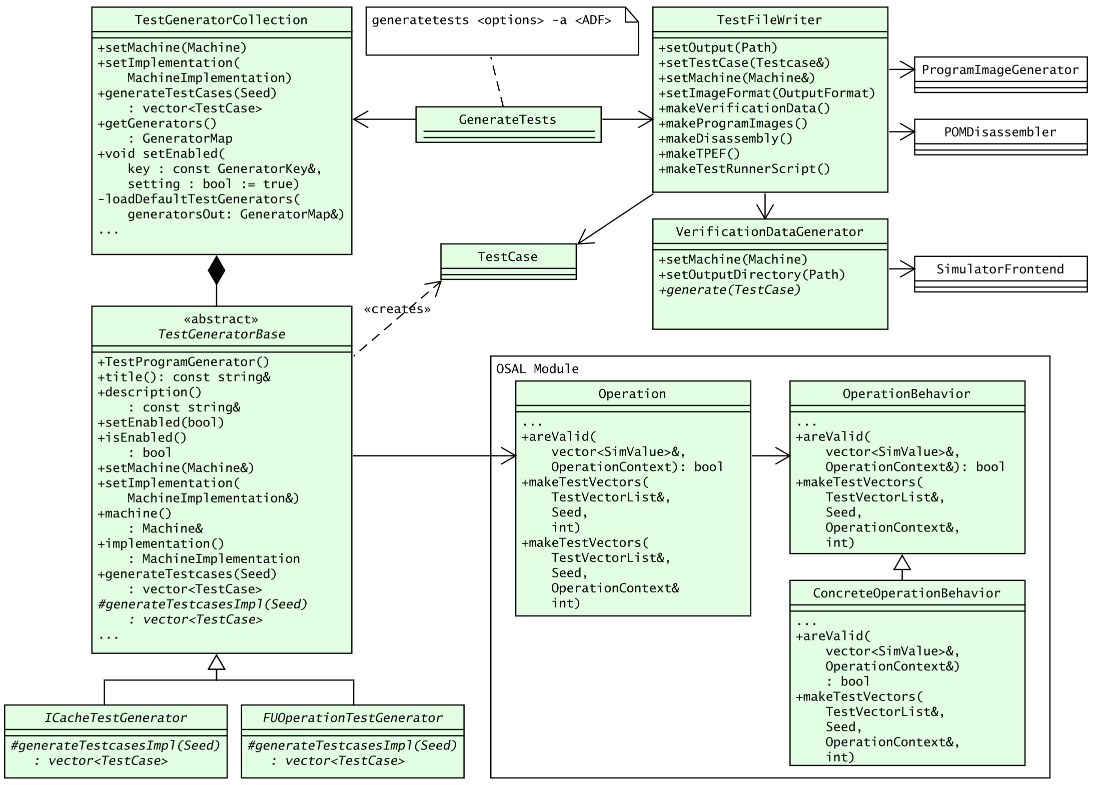

TestGeneratorCollection
+setMachine(Machine)
+setImplementation(MachineImplementation)
+generateTestCases(Seed) : vector<TestCase>
+getGenerators() : GeneratorMap
+void setEnabled(key : const GeneratorKey&, setting : bool := true)
-loadDefaultTestGenerators(generatorsOut: GeneratorMap&)
...
generatetests <options> -a <ADF>
GenerateTests
TestFileWriter
+setOutput(Path)
+setTestCase(Testcase&)
+setMachine(Machine&)
+setImageFormat(OutputFormat)
+makeVerificationData()
+makeProgramImages()
+makeDisassembly()
+makeTPEF()
+makeTestRunnerScript()
ProgramImageGenerator
POMDisassembler
VerificationDataGenerator
+setMachine(Machine)
+setOutputDirectory(Path)
+generate(TestCase)
SimulatorFrontend
TestCase
«creates»
«abstract»
TestGeneratorBase
+TestProgramGenerator()
+title(): const string&
+description() : const string&
+setEnabled(bool)
+isEnabled() : bool
+setMachine(Machine&)
+setImplementation(MachineImplementation&)
+machine() : Machine&
+implementation() : MachineImplementation
+generateTestcases(Seed) : vector<TestCase>
#generateTestcasesImpl(Seed) : vector<TestCase>
...
ICacheTestGenerator
#generateTestcasesImpl(Seed) : vector<TestCase>
FUOperationTestGenerator
#generateTestcasesImpl(Seed) : vector<TestCase>
OSAL Module
Operation
...
+areValid(vector<SimValue>&, OperationContext): bool
+makeTestVectors(TestVectorList&, Seed, int)
+makeTestVectors(TestVectorList&, Seed, OperationContext& int)
OperationBehavior
...
+areValid(vector<SimValue>&, OperationContext&): bool
+makeTestVectors(TestVectorList&, Seed, OperationContext&, int)
ConcreteOperationBehavior
...
+areValid(vector<SimValue>&, OperationContext&) : bool
+makeTestVectors(TestVectorList&, Seed, OperationContext&, int)

Figure 7.2 *The simplified class diagram of the Test Generator Framework. The highlighted classes are new additions or modified in TCE.*

The test generation process goes as followed: The main function (GenerateTests in the diagram) controls, which test generators are used via *TestGeneratorCollection*. Next, also via *TestGeneratorCollection*, the enabled test generators are invoked to create test cases. Each generator creates zero or more emphTestCase object, which includes a test program. The test case objects are passed to *TestFileWriter*, where, finally, the output files produced. The generation of verification reference data is delegated to *VerificationDataGenerator*.

New test generators are implemented by inheriting a base abstract class *TestGeneratorBase* and overriding its only pure virtual function *generateTestcasesImpl()*. The *generateTestcasesImpl()* is function that creates the actual test cases. In the base class, the *generateTestcases()* function does some preliminary checks and then calls *generateTestcasesImpl()* in *template method* design pattern fashion. The new created test generators are added to *TestGeneratorCollection*.

The diagram 7.2 shows two implemented test generators: *FUOperationTestGenerator* and *ICacheTestGenerator*. The former tests operations of each FU in a machine and the latter generates instruction references that touches each block in an instruction cache at least once.

The test generators may need to know details about operations in a machine in order to generate correct test programs. For example *FUOperationTestGenerator* creates test cases for stressing operations in a machine by trying various inputs on the operations. However, the generator does not know what inputs are valid for any operations.

In OSAL module, *Operation* is a class that provides information about properties of operations and their behavior. Two new functions were added: *areValid()* and *makeTestVectors()*. Actual implementations of the functions are in a class *OperationBehavior* and its derived classes. The *OperationBehavior* provides base implementation of the functions and the derived ones overrides these when necessary.

The function *areValid()* tells if input operands for a operation are valid. However, the base implementation of the function in *OperationBehavior* always returns true. The concrete operation behavior objects in OSAL overrides the function, when needed, to provide an actual input validation.

The function *makeTestVectors()* provides a way to generate test inputs for a operation. The function takes two parameters: a seed value for pseudo-random generation and a requested amount of test vectors. The created test vectors, returned by reference, are appended to the given test vector list. Also, some operations may have side effect, thus the argument *OperationContext* is provided in both the functions. The *OperationContext* is class where the state of side effects for a operation are stored for subsequent use of it. A base implementation of the *makeTestVectors()* function in *OperationBehavior* creates set of test vectors pseudo-randomly. All the created test vector are validated with the *areValid()* and invalid vectors are discarded. Classes derived from *OperationBehavior* can override the base function.

7.2 TTA Processor Test Bench Improvements

Most of the testing of memory hierarchy design units were made in RTL-simulations. For verification purposes and statistics collection new additions to the existing processor test bench generation were made:

- A separate test bench generation for multi-core TTA processors,
- Improved RTL-simulation run controls and
- HDL code coverage measurement option.

Multi-core test bench

The preceding test bench generation only accommodated single core processors using static test bench source files. Especially, the test bench generation expects a TTA processor to have specific interface in order to instantiate it successfully. In a multi-core TTA processor there are number of interfaces depending on the number of cores in the processor and the micro-architectural features such as instruction bus arbiter. The interfaces usually are data memory and instruction memory interfaces.

The new test bench, made for the multi-core designs, recognizes the memory interfaces, using the new signal semantics feature of the ports and port groups. The test bench is implemented as netlist block, which dynamically instantiates memory block needed by the processor.

RTL-simulation controls

RTL-simulations are used to extract cache and power usage statistics by running various test programs in the generated processors. For this purpose RTL-simulation time is needed to be limited to cover the duration of the test program. Simulating the program too long or short leads to misleading statistics.

One way to limit simulation time is to use instruction cycle count, which is acquired from TTA-simulation. However, the TTA-simulation does not cover stall cycles that stems from the micro-architecture. Different memory hierarchy settings and the kind of programs run in the simulation causes different stalling behavior. Due to this, the actual simulation time needed to cover just the program length is complex to predict.

A solution was to make the processor test bench to track program execution in the TTA processor and dynamically stop the simulation when the instruction count limit is met.

The simulation time limitation by execution tracking is depicted in Figure 7.3. A TTA core has a port that signals the stall status. Whenever - indicated by the signal - the core is not locked, it is executing instructions. This way the test bench knows how many instruction have been executed at any moment. The executed instruction count is compared to execution limit read from a file. When the counter matches with the limit, the test bench ceases simulation by disabling the clock signal generation. Stopping the clock starves the event based RTL-simulators from events and therefore causes simulation to stop.

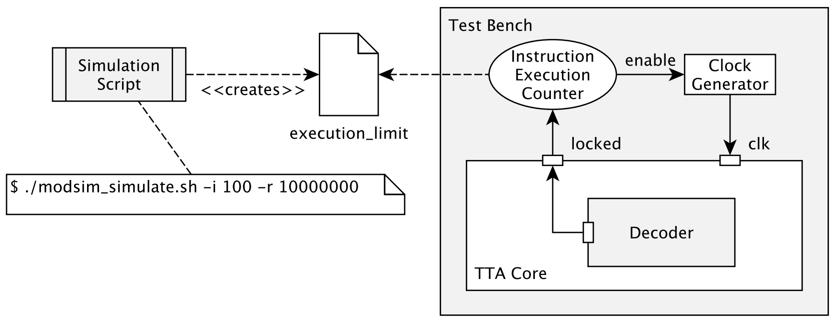


Figure 7.3 *The method of limitation of RTL-simulation time just to cover instruction count.*

The simulation stop by clock disabling is cleaner way than using for example VHDL's assertion, which would show as a simulation error.

The simulation controls are available with processor test benches generated by ProGe and are used through options of the simulation script. The options are *-i <instruction-count>* and *-r <simulation-time>*. The former option sets up the simulation to the given instruction execution count. The latter option overrides script's absolute simulation time, which is by default 52,390 nanoseconds.

HDL Code Coverage

The HDL code coverage is employed to give feedback on the extent of the tests. The feature is enabled by giving *-c* option to both the compilation and simulation scripts, which are generated by ProGe with processor test bench option. This option is only functional with ModelSim compilation and simulation scripts.

When code coverage option is enabled, the RTL-simulations in ModelSim produces two databases (.ucdb), where the code coverage measurements are collected. One of the databases temporarily stores measurements from a single run and another one (accumulated_coverage.ucdb) collects and merges all the coverage measurements from all the previous simulation runs.

The types of code coverage measured with ModelSim [20] are listed below:

- Statement coverage: Measures the executed statements in the code.
- Branch coverage: Measures the choices taken in the branches.
- Focused Expression Coverage (FEC): It is extension to the branch coverage and checks if all inputs combinations contributing to outcome of branch expression are taken.
- Toggle coverage: measures the bit changes in bit-based data types.
- Finite state machine coverage: Measures covered states in FSMs.

The code coverage data is inspected by using either using ModelSim *vcover report* command or alternatively using report script (modsim_report.sh). The latter creates a directory *reports/* where coverage data is outputted in text files. The files consists of a summary file and detailed report file for each design unit in the processor.

7.3 Test Environment

The implementations were tested using a process depicted in the Figure 7.4. Different TTA processor configurations were tested by running various test programs in both the TTA simulator and RTL simulation. Both the simulations produce data that can be compared together: *bus trace* and *output* file. The *bus trace* contains values that are moved in the buses of the IC. *Output* is print out from the test program. If these files from the both simulations matches, the behavior is verified successfully.

The test programs used in verifications were handmade assembly programs, C benchmark programs from CHStone suite [21] and test programs generated by TeGe. Test programs were run in RTL-simulation in two modes: in a normal mode and in *lock generation*. In the former programs are run in a processor without any additional test options. In the latter mode processor is forced to stall heavily. This effectively tests that interlocking in implemented designs do work correctly.

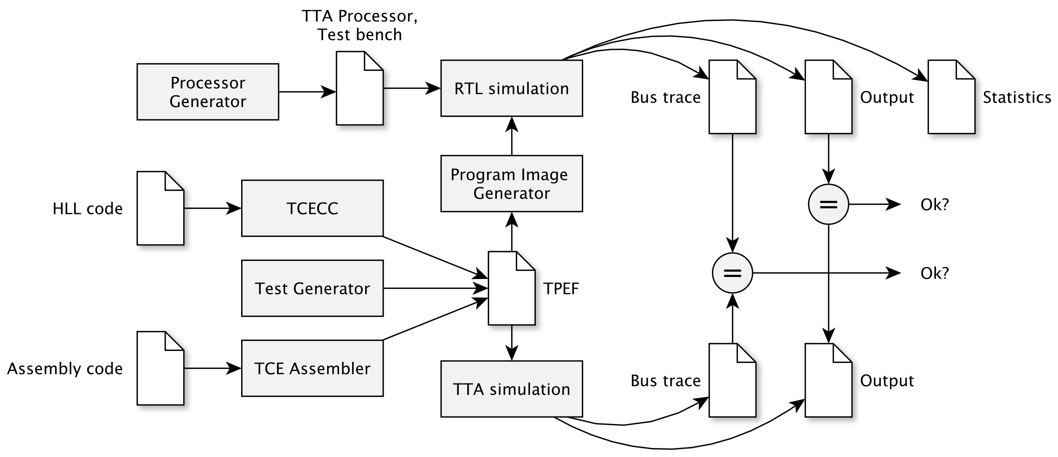


Figure 7.4 *The verification process.*

RTL-simulations were ran and HDL code coverage measured using ModelSim SE 10.1d [20]. Many of the RTL-simulations were ran in parallel with GCU Parallel command line tool [22].

For selected TTA processor configurations, Synopsys' Design Compiler was used to test that implementations of memory hierarchy were synthesizable, and to gather area and power estimations. All the selected processor configurations were synthesized using a 28 nm *fully depleted silicon on insulator* standard cell technology library and power optimizations that includes leakage and dynamic power optimization and clock gating. All synthesized processors were targeted to run at clock frequency of 1 GHz with added margin of 10 %.

One of the TTA architectures used in testing and synthesis was a typical scalar microcontroller processor (MCP) with limited parallelism. The processor has a LSU, a ALU, a separate unit for integer multiplication (MUL), 16x32bit RF, a boolean register and a IU. Interconnect consists of three buses with limited connectivity. Width of both the instruction word and fetch block is 40 bits.

7.4 Loop Buffer Verification

The loop buffer was verified by running handwritten TCE assembly programs in a minimal TTA processor with only a single bus, LSU, ALU and 5x32bit RF. The programs tested basic cases and corner cases that includes executing:

- a loop of non-zero instructions and iterations,
- a loop of zero instructions and zero iterations,

- a loop of more than one instruction and zero iteration,
- a loop of one instructions and more than one iteration,
- back-to-back loops and
- a nested loop, where inner loop uses the loop buffer.

Code coverage for loop buffer were 94.1%, which is satisfactory.

A MCP using a loop buffer of depth of 128 instruction were synthesized successfully. The critical path was formed from ALU's trigger port register to instruction fetch unit's (forefront of CU) program counter register.

Power consumption of the loop buffer design were studied in master thesis of [23]. In the study, power estimations from Synopsys Design Compiler were compared to a small 1-kilobyte SRAM and a level 1 cache model provided by Cacti [24]. The comparison indicated that the improvement in read energy was 5 and 42.9 times in favor of the loop buffer. A case in, where loop buffer reduced cycle count in a program by 20%, reduced total power consumption by 1 : 4.3 times than without the loop buffer.

7.5 Memory Hierarchy Generation

Memory hierarchy generation was tested by generating variety of TTA processor configurations. For cache customization and synthesis testing, the processor configurations comprised of one TTA architecture and three selected memory hierarchy configurations.

7.5.1 Instruction Cache Test Case Generation

TeGe's instruction cache test generator creates a test case where all cache sub-block are accessed at least once and have some of the blocks evicted. The generated test case program consists of pseudo-randomly generated jumps.

7.5.2 Verification

Three different processor configuration featuring *MCP* each having different level 1 instruction cache configuration were verified and synthesized. The cache configurations are:

- direct-mapped, 8 instructions in a block and 32 sets (MCP-L1-8-1-32),
- 4-way, semi-associative with 4 instructions in block and 32 sets. LRU is used as replacement policy and
- direct-mapped, 1 instruction in a block and 128 sets.

Two first cache configurations fit in 256 instructions, which is total of 1280 bytes with MCP's 40 bit wide instructions, and last cache configuration fits in 128 instructions.

In the Table 7.2, 7.3 and Table 7.4, is the test programs used in synthesis testing and their statistics about cycle count and miss-rate from level 1 cache. The programs in the table are from CHStone suite [21].

Table 7.2 *The cycle counts and miss rates for MCP-L1-8-1-16.*

Program	Cycle Count	Miss Rate (%)
adpcm	204,182	11.70
gsm	25,451	1.29
jpeg	13,227,531	0.89
sha	1,021,320	2.31

Table 7.3 *The cycle counts and miss rates for MCP-L1-4-4-16.*

Program	Cycle Count	Miss Rate (%)
adpcm	211,461	22.50
gsm	25,943	2.76
jpeg	13,226,175	1.60
sha	1,038,274	4.56

Table 7.4 *The cycle counts and miss rates for MCP-L1-1-1-128.*

Program	Cycle Count	Miss Rate (%)
adpcm	280,851	90.90
gsm	29,788	15.10
jpeg	15,501,508	13.30
sha	1,151,290	18.10

The area results from synthesis for all the configurations are presented in Table 7.5. From the table, it can be observed that overhead from control logic is small for the caches.

In the tables 7.6, 7.7 and 7.8 are presented average power and total energy consumption estimations of the processor. From the figures, it can be observed that the caches do not generate unreasonable amount of heat. Also, it can be observed

Table 7.5 *The area figures from synthesis.*

	MCP-L1-8-1-32		MCP-L1-4-4-16-LRU		MCP-L1-1-1-128	
	μm^2	%	μm^2	%	μm^2	%
Total	39,037.22	100	41,989.95	100	25,606.62	100
ICache	32,809.73	84.0	35,833.28	85.3	19,298.18	75.4
Storage	32,350.59	82.9	34,571.96	82.3	18,997.13	74.2
Control	459.14	1.1	1,261.32	3.0	301.05	1.2
Core	6,227.49	16.0	6,156.67	14.7	6,308.44	24.6

that power consumption seems to correlate with miss-rates. In caches the power consumption increases as the miss-rate grows, which is probably because of the fetching process. On the contrary, the power consumption is decreased, but this is because of stalling caused by misses in the cache. During the stalls the cores do have only little activity and, hence, consumes less power on average. Especially, with the *adpcm* program, the processor core is stalled half of the time. Observing tables 7.7 and 7.8, can be seen that power and energy consumption is nearly the same, even if the cache storage in the latter is just half of the former in bits. This is most likely due to address logic for the latter, as there is more indexable locations in the cache storage.

Synthesis of both the MCP configurations was targeted at one gigahertz clock frequency successfully. Critical paths were formed in the MUL unit between trigger port and output port registers (MCP–L1-4-4-16-LRU), and in the cache between controller unit's FSM state register and a block data array (MCP-L1-8-1-32).

Table 7.6 *The power figures for MCP-L1-8-1-16.*

	Total		ICache			Core		
Program	*mW*	*nJ*	*mW*	*nJ*	%	*mW*	*nJ*	%
adpcm	7.946	1,460	6.377	1,172	80.3	1.569	288.33	19.7
gsm	7.207	165	3.850	88	53.4	3.357	76.90	46.6
jpeg	6.558	78,072	3.674	43,738	56.0	2.884	34,333.38	44.0
sha	7.605	6,990	4.411	4,055	58.0	3.194	2,935.89	42.0

Table 7.7 *The power figures for MCP-L1-4-4-16.*

	Total		ICache			Core		
Program	*mW*	*nJ*	*mW*	*nJ*	%	*mW*	*nJ*	%
adpcm	10.141	1,930	8.590	1,635	84.7	1.551	295.18	15.3
gsm	8.36	195	5.335	125	63.8	3.025	70.63	36.2
jpeg	7.49	89,158	4.897	58,292	65.4	2.593	30,865.92	34.6
sha	8.882	8,300	5.958	5,567	67.1	2.924	2,732.32	32.9

HDL coverage results for the caches were 76.96% for MCP-L1-8-1-16 and 88.17% for MCP-L1-4-4-16. The results are mediocre and could be better. The most lack

Table 7.8 *The power figures for MCP-L1-1-1-128.*

	Total		ICache			Core		
Program	*mW*	*nJ*	*mW*	*nJ*	%	*mW*	*nJ*	%
adpcm	10.248	1,954	8.674	1,654	84.6	1.574	300.19	15.4
gsm	8.785	204	5.691	132	64.8	3.094	71.87	35.2
jpeg	8.043	95,662	5.414	64,393	67.3	2.629	31,268.70	32.7
sha	9.216	8,562	6.241	5,798	67.7	2.975	27,63.93	32.3

in coverage came from bit toggling and FEC.

The instruction bus arbiter were verified in RTL-simulation on another MCP-like processor. The verification was conducted by changing core count of the processor and forcing processor stalling. Also, the arbiter was tested using along L1 instruction caches and without them.

8. FUTURE WORK

Improved L1 Instruction Cache Integrated level one instruction cache lacks features that could be useful. Especially, cache features such as *early restart* and *critical word first*. The *early restart* feature lets the processor retrieve the requested instruction when it arrives instead of waiting until the whole block is fetched and thus reducing stall cycles. The *critical word first* improves this by making cache to fetch words of a block having the requested instruction first. Other features such as *burst interface* and new *replacement policies* are reasonably easy to implement.

Graphical User Interface Syntax and tag names for defining instruction memory hierarchy in IDF is not easy to remember. Therefore, time is spent for looking up the manual or an example from another IDF. There is already a dialog in ProDe for defining implementations for TTA units, so new TCE users does not need to even look into IDF. This dialog should be extended to cover customization of instruction memory hierarchy too.

9. RELATED WORK

Few patents in US related to loop buffers have been granted [25], [26], [27] and [28]. The first three are still active and the last one is elapsed. Patent in [25] claims a processor, which has a loop buffer and a cache, and a method to determine, from which of the caches instruction is to be fetched, and does not have technical details about a particular loop buffer. The loop buffer invention in patent [26] is software-managed cache, which has capability for nested loops and option to exit the buffer managed loops earlier. The patent of [27] claims processors that use a loop buffer to reduce power consumption. Their description of the loop buffer covers three embodiments: two of them are transparently operating and the third one software-managed. The last patent [28] describes a transparently operating loop buffer, which allows invalidation of instructions stored in the loop storage.

Cadence's Tensilica Xtensa [5] is similar to TCE for designing customized processors. Their processor generator can utilize instruction cache and also a loop buffer. Their loop buffer is configurable up to 256 bytes where as our loop buffer does not have hard limits on this. Design of their loop buffer is not explained publicly. [29]

The implemented automated test generation in the TCE is not an unique feature. For example, both Cadence's Tensilica Xtensa [29] and Synopsys's ASIP Designer [30] already have means for automated test program generation for verifying processor correctness.

10. CONCLUSION

The customization and generation of the instruction memory hierarchies was implemented and integrated into the processor generator of the TCE. Prior this, several changes were made in TCE's processor generator. The generation includes a loop buffer, a level 1 instruction cache and an instruction bus arbiter. The loop buffer is automatically instantiated in a TTA processor design by the inclusion of a loop setup operation. The other parts, transparent to programmers, of the instruction memory hierarchy customization is carried out by explicitly defining the structure. The structure is described in IDF along parameters and feature setting of each parts in it.

Cache statistics collection was implemented. The tool *rtlstats*, made for reporting the statistics from RTL-simulation, shows miss rates and average memory access time for each core in the processor along other statistics such as stall cycles and total simulated cycles.

As side product, a framework for automated test generation or TeGe was implemented along with two test case generators. The TeGe reads ADF and optionally IDF and produces test cases that are run in RTL-simulation. A test runner script takes the test cases, runs them in TTA processor RTL-simulation and reports results.

Several improvements were made to the processor generator's test bench generation. More accurate simulation controls were added to address unpredictable stalling situations. The accurate simulation time limitation is needed for HDL code coverage and power measurements.

Implementations, memory hierarchy generation and its design units, were verified. A processor design with several different instruction memory hierarchy were selected for synthesis area and power measurement.

The accomplishments in this thesis are passable and there is room for improvements. Currently, the memory hierarchy customization options are quite limited and many indented features to be added did not make it to the thesis. Also, the verification of the designs was not too high of quality, which appears on the code coverage.

BIBLIOGRAPHY

[1] Jari Nurmi, editor. *Processor Design: System-On-Chip Computing for ASICs and FPGA*. Springer, 2007.

[2] Tampere University of Technology. *TTA-Based Co-design Environment Home Page*. Available: `http://tce.cs.tut.fi/index.html`, referenced 11/20/2015.

[3] Jari Heikkinen. *Program compression in long instruction word application-specific instruction-set processors*. PhD thesis, Tampere University of Tampere, 2007.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[5] Xtensa customizable processors. `http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable`, referenced 11/18/2015.

[6] Henk Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Ltd., Chichester, England, 1998.

[7] clang: a c language family frontend for llvm. Available: `http://clang.llvm.org/`, referenced: 11/24/2015.

[8] The llvm compiler infrastructure: Llvm overview. Available: `http://llvm.org/`, referenced: 11/18/2015.

[9] Mikael Lepistö. Assembly compiler for parametrizable parallel processor. Master's thesis, Tampere University of Tampere, 2006.

[10] Lasse Laasonen. Program image and processor generator for transport triggered architectures. Master's thesis, Tampere University of Technology, 2007.

[11] Pekka Jääskeläinen. Instruction set simulator for transport triggered architectures. Master's thesis, Tampere University of Technology, 2005.

[12] Luca Benini, Alberto Macii, and Massimo Poncino. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *ACM Trans. Embed. Comput. Syst.*, 2(1):5–32, February 2003.

[13] Bruce Jacob, David T. Wang, and Spencer W. Ng. *Memory Systems*. Morgan Kaufmann, San Francisco, 2008.

[14] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures*. Systems on Silicon. Morgan Kaufmann, Burlington, 2008.

[15] Janne Helkala. Variable length instruction compression on transport triggered architectures. Master's thesis, Tampere University of Tampere, 2015.

[16] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability and Scalability*. Wiley Interscience, 2006.

[17] V. Saljooghi, A. Bardizbanyan, M. Sjalander, and P. Larsson-Edefors. Configurable rtl model for level-1 caches. In *NORCHIP, 2012*, pages 1–4, Nov 2012.

[18] L1 cache vhdl code. Available at: `http://www.flexsoc.org/`, referenced: 11/18/2015.

[19] Customized Parallel Computing group. *TTA Codesign Environment 2.0 (trunk) User Manual*.

[20] Mentor Graphics. *ModelSim SE User's Manual, Software Version 10.1d*, 2012.

[21] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[22] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011. Available: `http://www.gnu.org/s/parallel`, referenced: 11/23/2015.

[23] Joona Multanen. Hardware optimizations for low-power processors. Master's thesis, Tampere University of Tampere, 2015.

[24] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman Jouppi. *CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model*. HP Labs. Available: `http://www.hpl.hp.com/research/cacti/`, referenced: 11/18/2015.

[25] Lawrence A. Booth. Apparatus having a cache and a loop buffer. U.S. Patent 6 757 817 B1, Jun 2004.

[26] Kumar Ganapathy, Ruban Kanapathipillai, and Kenneth Malich. Method and apparatus for loop buffering digital signal processing instructions. U.S. Patent US 6 598 155 B1, Jul 2003.

[27] Matthias Knoth. Processor utilizing a loop buffer to reduce power consumption. U.S. Patent 7 873 820 B2, Jan 2011.

[28] Steven L. George. Instruction fetch look-aside buffer with loop mode control. U.S. Patent 4 626 988, Dec 1986.

[29] Tensilica's (now cadence's) xtensa 10: Process capabilities, application needs drive core evolution trends. *BDTi*, 2013. available: `http://www.bdti.com/InsideDSP/2013/12/11/Cadence`, referenced: 11/18/2015.

[30] Dhanendra Jani and Steve Leibson. How tensilica verifies processor cores. *EE Times*, 2003. available: `http://www.eetimes.com/document.asp?doc_id=1202143`, referenced: 11/18/2015.