TAMPERE UNIVERSITY OF TECHNOLOGY

MIKKO JÄRVELÄ

VECTOR OPERATION SUPPORT FOR TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Examiners: Prof. Jarmo Takala and Pekka Jääskeläinen, Dr. Tech
Examiners and subject approved by the Faculty Council of the Faculty of Computing and Electrical Engineering
October 9th 2013

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Signal Processing and Communications Engineering
Järvelä, Mikko : Vector Operation Support for Transport Triggered Architectures
Master of Science Thesis, 56 pages
June 2014
Major: Embedded Systems
Examiners: Prof. Jarmo Takala and Pekka Jääskeläinen, Dr. Tech
Keywords: compiler, DLP, SIMD, transport triggered architecture

High performance and low power consumption requirements usually restrict the design process of embedded processors. Traditional design solutions do not apply to the requirements today, but instead demands exploiting varying levels of parallelism. In order to reduce design time and effort, a powerful toolset is required to design new parallel processors effectively.

TTA-based Co-design Environment (TCE) is a toolset developed in Tampere University of Technology for designing customized parallel processors. It is based on a modular *Transport Triggered Architecture* (TTA) processor architecture template, which provides easy customization and allows exploiting instruction-level parallelism for high performance execution.

Single Instruction, Multiple Data (SIMD) paradigm provides powerful data-level parallel vector computation for many applications in embedded processing. It is one of the most common ways to exploit parallelism in today's processor designs in order to gain greater execution efficiency and, therefore, to meet the performance requirements.

This work describes how data-level parallel SIMD support is introduced and integrated to the TCE design flow for more diverse parallelism support. The support allows designers to customize and program processors with wide vector operations. The work presents the required modification points along with the new tools that were added to the toolset. Much weight is given for the retargetable compiler, which must be able to adapt to all resources on TTA machines. The added tools were required to provide as much automatic behavior as possible to maintain effective design flow. In addition, the thesis presents how the modifications and new features were verified.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
Järvelä, Mikko : Vector Operation Support for Transport Triggered Architectures
Diplomityö, 56 sivua
Kesäkuu 2014
Pääaine: Sulautetut Järjestelmät
Tarkastajat: prof. Jarmo Takala ja TkT Pekka Jääskeläinen
Avainsanat: kääntäjä, datarinnakkaisuus, SIMD, siirtoliipaisuarkkitehtuuri

Sulautettuja prosessoreita suunniteltaessa niiden tulee usein kyetä korkeaan laskentasuorituskykyyn mahdollisimman pienellä tehonkulutuksella. Nykyajan korkeisiin suorituskykyvaatimuksiin ei voida enää vastata perinteisillä suunnitteluratkaisuilla, vaan prosessorisuunnittelussa vaaditaan eritasoisten rinnakkaislaskentamenetelmien hyödyntämistä. Jotta uusia rinnakkaisprosessoreita voidaan suunnitella tehokkaasti, vaaditaan siihen tehokas kehitysympäristö.

TTA-based Co-design Environment (TCE) on Tampereen Teknillisellä Yliopistolla kehitetty täyden suunnitteluvuon tarjoava kehitysympäristö räätälöitävien rinnakkaisprosessorien suunnitteluun. Se pohjautuu siirtoliipaistuun prosessoriarkkitehtuurimalliin (*Transport Triggered Architecture*, TTA), joka tarjoaa helpon räätälöitävyyden sekä mahdollistaa käskytason rinnakkaisuuden hyödyntämisen korkean suorituskyvyn saavuttamiseksi.

Single Instruction, Multiple Data (SIMD) on datarinnakkaisen laskennan muoto, jolla voidaan tehokkaasti suorittaa rinnakkaista vektorilaskentaa useissa sulautetuissa sovelluksissa. Se on eräs yleisimmistä tavoista hyödyntää rinnakkaisuutta nykypäivän prosessorisuunnittelussa korkeamman laskentatehokkuuden ja sitä myötä suorituskykyvaatimusten saavuttamiseksi.

Tässä työssä kuvataan kuinka datarinnakkaiset laskentaominaisuudet integroitiin TCE:n suunnitteluvuohon laajemman rinnakkaisuustuen tarjoamiseksi. Uudet ominaisuudet mahdollistavat prosessoreiden räätälöinnin ja ohjelmoinnin vektorioperaatioilla. Työssä esitellään suunnitteluvuohon vaaditut muutokset sekä kehitysympäristöön lisätyt uudet työkalut. Painoarvoa annetaan etenkin kohdearkkitehtuureihin mukautuvalle kääntäjälle, jonka tulee säilyttää automaattiset mukautumisominaisuutensa kaikkiin laitteistoresursseihin. Kehitysympäristöön lisättyjen työkalujen vaatimuksena oli lisäksi toimia mahdollisimman automaattisesti tehokkaan suunnitteluvuon ylläpitämiseksi. Lisäksi työssä esitetään, kuinka muutokset ja datarinnakkaisen laskennan toteutuminen on testattu.

PREFACE

The work in this M.Sc. thesis was carried out in the Department of Pervasive Computing at Tampere University of Technology as a part of the Parallel Acceleration Project (ParallaX) project.

I would like to express my gratitude for Professor Jarmo Takala for giving me an opportunity to work on this interesting and challenging project and for his improvement ideas for this thesis. I am also the most grateful to Pekka Jääskeläinen, Dr. Tech, and Heikki Kultala, M.Sc., for their guidance to improve this thesis and advice regarding the work on TCE. I would also like to thank my colleagues in the TCE project and Lasse Lehtonen for creating such a relaxed and interesting atmosphere at work and giving me a helping hand whenever it was needed. Finally, I would like to thank my friends and my family for their support throughout my studies and life.

Tampere, May 15th, 2014

Mikko Järvelä

CONTENTS

LIST OF ABBREVIATIONS

ADF	Architecture Definition File
ASIC	Application-Specific Integrated Circuit
ASP	Application-Specific Processor
DAG	Directed Acyclic Graph
DLP	Data-Level Parallelism
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GPGPU	General-Purpose computation on Graphics Processing Unit
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HDB	Hardware Database
HLL	High-Level Language
IDF	Implementation Definition File
ILP	Instruction-Level Parallelism
IR	Intermediate Representation
LLVM	Low Level Virtual Machine
NVPTX	Nvidia Parallel Thread Execution
OSAL	Operation Set Abstraction Layer
OTA	Operation-Triggered Architecture
RTL	Register-Transfer Level
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SM	Streaming Multiprocessor
SOC	System-On-Chip

SSA	Static Single-Assignment
TCE	TTA-based Co-design Environment
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architecture
VLIW	Very Long Instruction Word
XML	Extensible Markup Language

1. INTRODUCTION

Embedded processors are commonly designed to implement target application functionality under strict design requirements, such as execution time, power consumption and chip area. Especially in applications that require a great deal of intensive computation in short time, performance requirements may be high. Pre-designed "off-the-shelf" *General-Purpose Processors* (GPPs) are usually too slow in terms of speed and consume too much power to be a feasible solution. On the other hand, designing the target application as a fixed function accelerator is time consuming and lacks programmability.

As target applications usually consist of a limited set of computational tasks, customizable embedded processors can be used to provide a feasible solution by tailoring the processor resources optimally for target-specific purposes. These customized *Application-Specific Processors* (ASPs) combine the advantages from both software flexibility and hardware efficiency, as the basic tasks of the application can be executed on the main core, while specialized function units can be used to handle the complex and heavy computation. However, customized processors need to undergo a design process, in which they are tailored with proper resources. This requires time and effort, and as the continuously tightening design requirements have to be also taken into account, processors must be often equipped with parallel computing capabilities.

TTA-based Co-design Environment (TCE) is a toolset developed in Tampere University of Technology for designing and programming processors based on *Transport Triggered Architecture* (TTA) processor template. TTA is a modular and easily customizable processor architecture, which allows exploitation of static *Instruction-Level Parallelism* (ILP), and thus, fits well for designing efficient parallel processors to implement target applications. The toolset aims to provide full and effective design flow for designing TTA processors with comprehensive and partly automatic design tools.

For this thesis, *Single Instruction, Multiple Data* (SIMD) vector operations and high-level vector programming were introduced to the design flow in order to extend TCE to support *Data-Level Parallelism* (DLP) and, therefore, provide more customization options for this degree of parallelism. This thesis describes the TCE design flow and the tools that were modified or created in bringing the SIMD capabilities to the toolset. Most of the emphasis is given for the retargetable compiler. The thesis also presents how the SIMD toolset modifications and new tools were verified.

The structure of this thesis is divided into the following chapters. Chapter 2 describes typical forms of parallelism and presents the basics of customizable processor templates and TTA processors. TCE is also introduced, after which a more detailed overview is done on compilers, as it is one of the key elements in this thesis. Chapter 3 describes the main TCE design flow and introduces the most relevant tools that are influenced by the SIMD extension. The implementation is presented in Chapter 4. Chapter 5 presents the methods that were used to verify the SIMD extension functionality in different parts of the toolset. Conclusions and future work are presented in Chapter 6.

2. CUSTOMIZATION OF PARALLEL PROCESSORS

2.1 Parallelism

Software programs are defined as a sequence of operations that are executed on the underlying hardware. Embedded processors often have strict constraints that the final product must adhere to, such as power consumption or chip area, while possibly high requirements for execution performance must also be met. The traditional way to increase performance by pushing up the clock frequency has lead to issues with power consumption, and has forced processor designers to exploit other means to achieve satisfying execution efficiency. Today, one of the common approaches and a constant research objective is utilizing different levels of parallelism in program execution. In parallel execution, more than one processor hardware resource is utilized to compute calculations concurrently. Instruction-level parallelism and data-level parallelism have been typically used in pursuing higher computing performance [1].

2.1.1 Instruction-Level Parallelism

A common way to utilize processor circuitry better is instruction pipelining, in which instructions are divided into parts and execution into stages. Different parts from different instructions can be executed in the pipeline stages, which allows overlapping of several instructions and having them in execution concurrently. The pipelining allows increasing the clock frequency, as the instruction parts have smaller latency than the whole unified instruction. However, the maximum amount of completed instructions each cycle is one at best, because the pipeline is sequential [2].

Another technique to support ILP is when a processor can explicitly have several instructions completing from execution in a same cycle. This is achieved by extracting independent instructions from the sequential instruction stream and executing them on available parallel resources [1]. Processors that combine pipelining and simultaneous instruction execution are called *pipelined multiple-issue processors*.

Executing a group of parallel instructions requires that no data dependencies exist between the operations. Data dependencies appear when an operation computes a value that is to be read by another operation [3]. An example of parallelizable and non-parallelizable instructions is shown below. Both code excerpts have four different operations, each operation having two registers for input values and an output register for the result value.

```
a)                           b)
  ADD  R1,  R2  -> R3          ADD  R1,  R2  -> R3
  SUB  R4,  R5  -> R6          SUB  R4,  R3  -> R6   # uses result R3
  MUL  R7,  R8  -> R9          MUL  R7,  R6  -> R9   # uses result R6
  SHL  R10, R11 -> R12         SHL  R10, R9  -> R12  # uses result R9
```

The code excerpt on the left is ideal for exploiting ILP. All operations are independent, as none of them use results from the other operations. Since all the operations have no dependencies between each other, they can be parallelized and executed concurrently. In the code excerpt on the right, the code does not parallelize at all, since every operation needs the result value from the previous operation, except for `ADD`. This can be seen as the worst case scenario from the point of exploiting ILP, since the operations need to be executed sequentially from top-down.

Figure 2.1 gives an abstract view of sequential and parallel instruction execution on a processor hardware. Only the main pipeline stages are shown, and the rest (such as write-back) have been left out. The execution on the left shows a fully sequential program execution on a scalar processor, in which only one explicit instruction can be put to the instruction pipeline for execution. The execution on the right presents a *Very Long Instruction Word* (VLIW) multiple-issue processor that is capable of executing four explicit instructions per clock cycle.

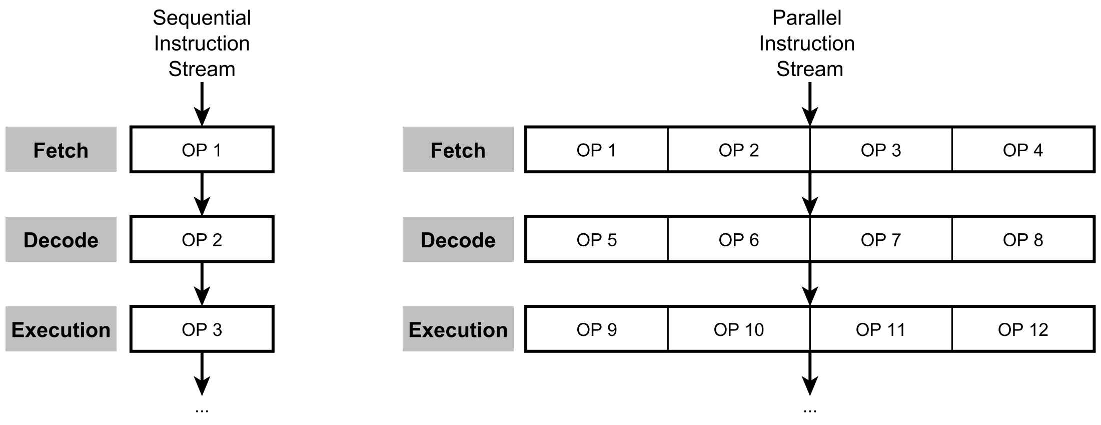


Figure 2.1: Principles of scalar and multiple-issue VLIW execution.

On the VLIW processor, the code excerpt from the above example on the right could not be executed any better than on the scalar processor, but the ideal code exceprt on the left could be parallelized to a single stage in the pipeline. However, instructions in the instruction stream are rarely in an ideal order from the perspective of exploiting ILP. Data independent instructions from the instruction stream have to be rearranged to allow code generation of code that utilizes the machine resources better [4]. Reordering of instructions improves the execution performance [4], as it avoids stalls by exploiting the processor pipelining better. Although the initial execution order of instructions is modified, the original effect of the program must not change. This compiler optimization is called *instruction scheduling*, and is used to improve ILP in program execution by extracting it statically at compile-time or dynamically at run-time. This divides multiple-issue processors to dynamically scheduled superscalar processors and statically scheduled processors.

In dynamically scheduled superscalar architecture, the responsibility of rearranging instructions in the instruction stream is given to the processor hardware. A superscalar processor performs data speculation on a limited-size instruction window to detect ILP at run-time, and issues independent instructions to parallel execution. However, as additional logic is required for the run-time functionality, the hardware complexity is increased and cycle times may become longer when compared to a scalar processor. [5]

The VLIW, which belongs to statically scheduled architectures, relies on the compile-time scheduling. In contrast to the run-time scheduling, the responsibility of exploiting ILP has been moved to the software compiler, which explicitly specifies the instructions that are executed in parallel [5]. Instructions from a sequential instruction stream are scheduled to very long instruction words. The very long instruction words consist of a set of parallel instructions, which utilize processor resources concurrently. The difference between a sequential and a parallel instruction stream is visualized in Figure 2.2.

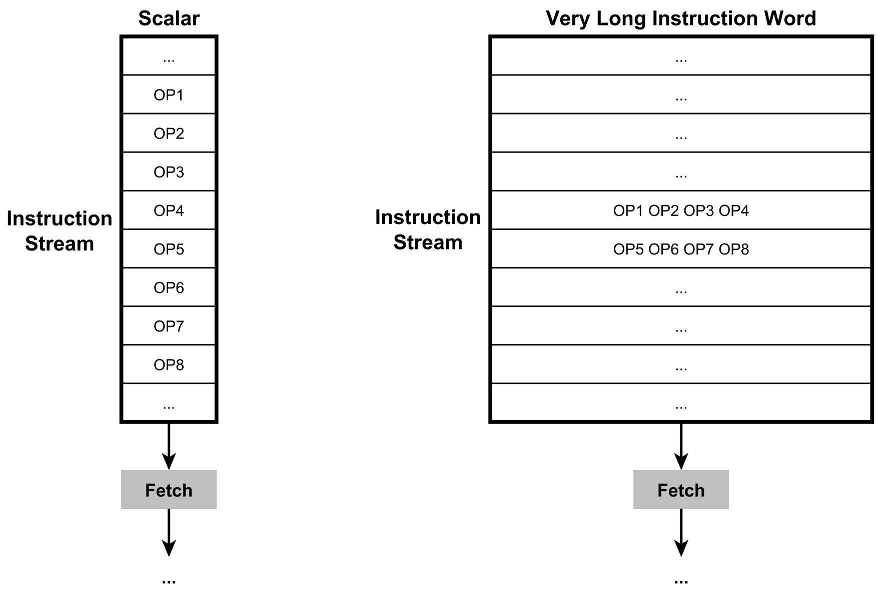


Figure 2.2: Difference between a sequential and a parallel instruction stream.

Instructions are laid out sequentially in the case of a scalar processor. If the processor is a superscalar, an additional hardware unit will be extracting ILP at run-time by fetching multiple instructions from the instruction stream and scheduling them to free parallel execution resources. In the case of the VLIW processor, the instructions have already been scheduled by the compiler to very long instruction words. In the example, the words consist of four parallel instructions, and they can be fetched to the pipeline for parallel execution without any additional logic.

In static scheduling the complexity of the compiler software increases due to the responsibilities in ILP extraction and scheduling logic. On the other hand, the hardware complexity is reduced, since there is no need for an additional hardware unit to do any run-time scheduling. In addition, whereas the dynamic scheduling logic has only a limited knowledge of incoming instructions, in static scheduling the compiler has global visibility over the whole program and its inner dependencies, making the instruction window infinite [5].

2.1.2 Data-Level Parallelism

In contrast to ILP, in which multiple operations are executed concurrently on independent data elements, in DLP the independent data elements are distributed to multiple similar concurrent processing elements. Modern data parallel processing techniques are usually referred to as SIMD processing, in which the same operation is applied to multiple data elements simultaneously. The data elements, or subwords, are single numerical values of a specific type, such as floating point or integer numbers. The elements form wide packed words called *vectors*. SIMD processing with vector operands is widely utilized in today's processors to bring execution efficiency. Historically, processors with vector processing hardware were mainly common in supercomputers. The following will present an example of a well-known and successful supercomputer, and another example is presented of a consumer-based processor.

In 1976, Cray-1 supercomputer was introduced with both scalar and vector processing capabilities. It contained eight 64-element vector registers (64-bit elements) and three vector function units dedicated for vector processing. It was effective especially at computing with short vectors, and implemented a "chaining" technique, in which intermediate result vectors could be used again without circulating them through the memory first. [6]

In 1996, Intel introduced the 64-bit MMX SIMD instruction set extension to its consumer-based X86 processor architecture for performance improvements in multimedia, communications and other numeric-intensive applications [7]. The 64 bits of packed data supported 8x8b, 4x16b, 2x32b and 1x64b data types, and especially the parallel operations on small data elements (8b and 16b) were the fundamental factor behind the performance boost [7]. As of this writing, Intel has scheduled to include the AVX-512 SIMD extension to its future processors, which extends the current SIMD support to 512-bit vectors [8].

SIMD fits best in heavy vector math computing, such as scientific or multimedia, in which great deal of the computation is data parallel. The higher the level of DLP is in the input program, the greater are the advantages from the SIMD-style execution. When compared to doing vector math on a SIMD hardware rather than on a scalar hardware, the execution has several benefits. One of them is the reduced instruction word overhead and increased computation performance. Only one SIMD instruction definition is required to execute N operations, rather than define N separate instructions to do the same amount of calculation on the scalar hardware. ILP offers more flexibility in exploiting parallelism, but suffers from the additional instruction overhead as it defines N separate scalar instructions.

By using specific vectorization methods, the amount of dynamic instructions (executed instructions in a program) have been managed to reduce by over 80 % in multimedia and communications related programs [9]. SIMD execution may also have significant benefits when it comes to energy savings. Hardware SIMD execution units consume more energy than scalar units and increase the average power consumption. However, due to the reductions in execution time and in instructions, such as energy-intensive memory accesses, the total energy consumption may be reduced considerably [10]. As a downside, vectors require wide buses for interconnection and wide registers for temporary storage, which leads to bigger die size and increased chip power consumption.

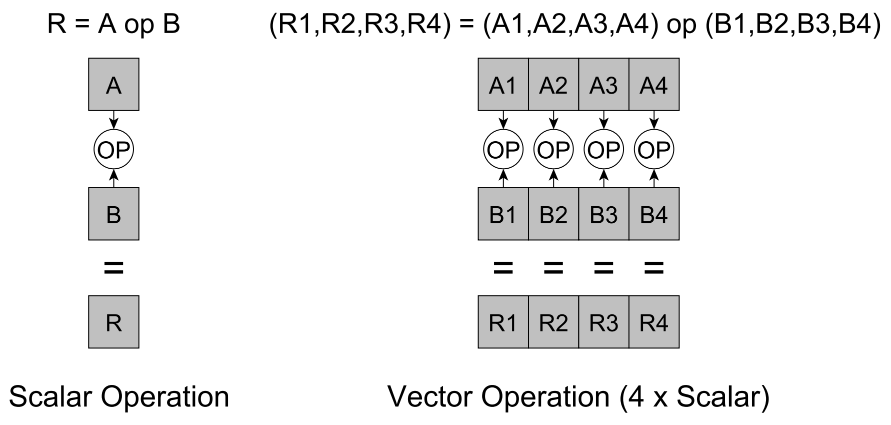


Figure 2.3: Principles of a scalar operation and a vector operation.

Figure 2.3 shows a simple example of a scalar operation and of a corresponding vector operation. The scalar operation on the left is executed by using the two scalar input operands, which results to a scalar output operand. The operands in the vector operation are vectors, which consist of four scalars. The vector operation is performed by executing the scalar operation on each of the Ax and Bx scalar pairs, resulting to a four-sized result vector. In other words, the vector operation performs N parallel operations for the N-sized input vectors.

In addition to embedded parallel processors and general-purpose CPUs, contemporary *Graphics Processing Units* (GPUs) are optimized for SIMD computation [11]. Their high-performance processing units are also increasingly exploited in GPGPU (General-Purpose computing on Graphics Processing Unit) computation. *Single Instruction, Multiple Thread* (SIMT) by Nvidia resembles the SIMD-style execution. The GPUs consist of *Streaming Multiprocessors* (SMs), which consist of a number of scalar processing elements [12]. The scalar processing elements operate on threads. In SIMT execution, a thread is created for each data element, which are then distributed for execution on the scalar processing elements [12].

2.2 Processor Templates

Customized processors are designed by tailoring and specializing the processor architecture resources to fulfill the application requirements. Flexible processor customization is a major factor in pursuit of higher performance, lower power consumption and smaller chip size. Customized processors are used in *System-on-Chip* (SoC) circuits to implement functionality of a target application (or part of it), for which a fixed function accelerator does not provide enough flexibility and a general-purpose CPU is too slow or consumes too much power [13, 14]. They are usually referred to as application-specific processors, or its variations, but this thesis will be using a more general naming convention *customized processors*.

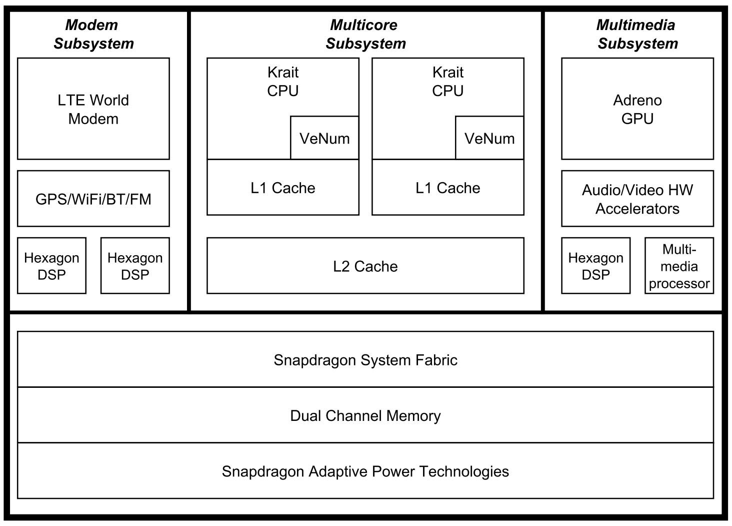


Figure 2.4: Block diagram of MSM8960 chipset. [16]

Heterogeneous multicores utilize multiple specialized cores to divide the workload optimally across the processor, for instance, by allocating graphics processing to a GPU and signal processing to a *Digital Signal Processor* (DSP). In homogeneous multicore processors the cores are replicated on the chip, and the processor consists of identical cores. They are easier to design and to program, as workloads can be executed on any core. However, as the cores are more generic in nature, the maximum number of cores is met earlier than with the heterogeneous architecture.

As heterogeneous architectures allow using specialized cores, they have more efficiency in chip area usage and in adapting to different applications [15]. Some of the cores can also be internally homogeneous, allowing multicore processors to benefit from both achitecture types. An example of a heterogeneous Qualcomm Snapdragon S4 multicore SoC (chipset MSM8960) is presented in Figure 2.4. In order to provide a complete mobile platform, it features several heterogeneous components, such as a multicore CPU, GPU, DSP, modem and multimedia processors, and on-chip memories.

As a drawback compared to a pre-designed "off-the-shelf" processor, customized processors need to undergo a design process. In order to minimize time and design effort to feasible amounts, a co-design toolset is required to design the processor software and hardware in a simultaneous process. The more automated the toolset is, the less time and effort is needed in the design process, which improves time-to-market delivery of the processor.

By using an architecture *template*, the design exploration space can be restricted to the reuse of pre-defined modules, which reduces effort in processor design and modelling [17]. A template defines general attributes of the processor and sets constraints on which ones can be customized, along with the amount of parallelism that can be influenced. Without an architecture template the whole processor would have to be designed from scratch, along with the toolset. The design and implementation of a co-design toolset becomes feasible due to the usage of a template, as it simplifies significantly the creation of some of the design flow tools, such as the compiler [18].

The processor design space exploration often starts with designing an initial architecture. The target software is compiled to the initial design, and then evaluated to find bottlenecks or low resource utilization. The processor is customized by removing unnecessary resources and adding in beneficial ones, after which the compilation and evaluation steps are run again. This sequence is repeated iteratively to enhance the initial processor design until a satisfying one is found. The final, customized parallel processor, is an instance of the template, tailored to meet the requirements of the target application.

2.3 Transport Triggered Architecture

Independent function units are utilized by the very long instruction words in the VLIW architecture. However, VLIW processors have scalability problems that start to occur when the number of function units is increased. Adding new function units grows the interconnection network rapidly, along with the chip area and the power consumption [19]. This has a negative impact on the achievable cycle time and causes severe scalability problems. Another issue is related to the bypass circuitry between function units. The circuitry allows moving a value from a function unit output port directly to a function unit input port without storing the value in a register file first. Delays can be reduced between operations due to this register bypassing, but the circuitry causes another scalability issue by having a complexity of at least $O(n^2)$ [19].

TTA consists of a set of function units (FUs), register files (RFs), buses and sockets [19]. It is a statically scheduled multiple-issue processor architecture template that is able to exploit static ILP by executing multiple operations concurrently. TTA is similar to the VLIW architecture, but can overcome the scalability related issues by requiring less wiring in the interconnection network and less ports in register files [19]. In TTAs, the datapath is visible to the programmer and they differ from traditional "Operation-Triggered Architecture" (OTA) in the way they are programmed [20]. A traditional OTA processor is programmed by specifying the operations that implicitly utilize the transports between FUs and RFs on the datapath. TTAs are programmed by explicitly specifying data transports (*moves*) between FUs and RFs in the interconnection network [20]. Operations are possibly triggered as a side effect of these transports [20].

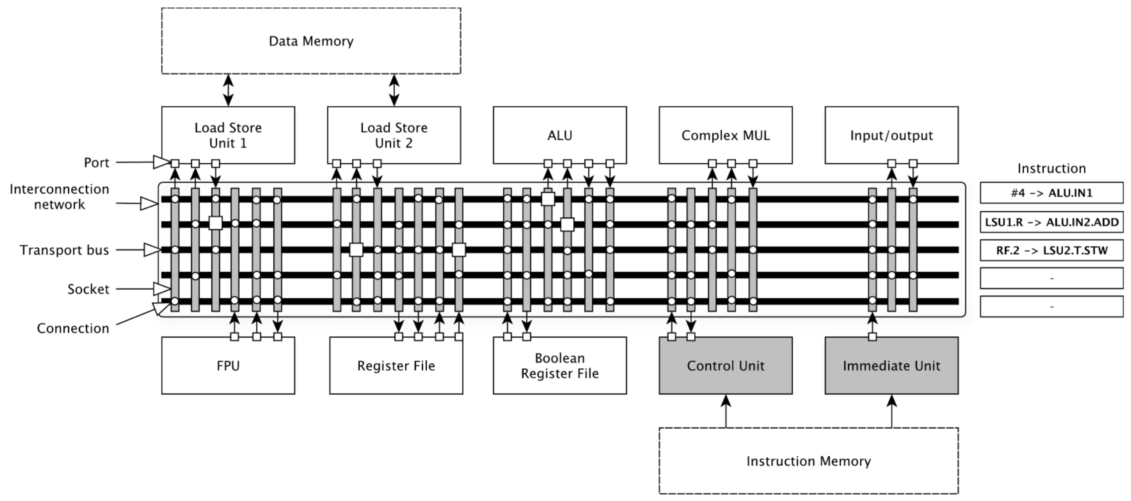


Figure 2.5: Example of a TTA processor.

Figure 2.5 shows an example of a TTA processor with some function units and register files, connected together by the interconnection network. Operands can be transferred from place A to place B by using the buses, while sockets provide connections between the buses and RFs or FUs. The level of ILP is affected by the number of buses: wide instructions can issue a data transfer for each bus. Socket and bus connections in the network dictate the possible data moves that programmers can describe.

Register files provide fast access temporary storage for operands. Function units contain one or several machine operations, which operate on the operands. In order to provide an interface for passing operands to buses, FUs and RFs have input and output ports, which are basically registers to which input and result values are placed [19]. One of the input ports in FUs is a *triggering port*, which is used to indicate that all input operands are ready for execution [19]. After an operand has been written to the triggering port, an operation is executed and the result written to the output port after the operation delay in clock cycles has passed.

2.4 TTA-Based Co-Design Environment

TCE is a toolset for designing and programming customized processors based on the TTA processor template. It is developed in Tampere University of Technology, aiming to provide full retargetable co-design flow from *high-level language* (HLL) programs to *register-transfer level* (RTL) processor implementations and parallel program binaries [21]. TCE includes the necessary tools, such as a processor designer, a compiler, a simulator and a processor implementation generator, to provide a full co-design flow. The customizable architecture components include register files, function units, and the interconnection network.

TCE machines are currently always big-endian, in which the most significant byte of a word is stored to the smallest address and the least significant byte to the largest address. This requires that all memory access operations on TTA machines must handle data in the same way.

In order to minimize manual work and optimize the design process, some of the tools provide fully automatic functionality. In addition, some parts of the toolchain must also be able to retarget to the processor architecture at runtime to make the design flow efficient and feasible. The retargetability requires that the toolchain can describe design flow related information in general data structures that are easily accessible by other tools. The TCE toolset uses TCE-specific languages and formats to model the necessary design flow information, such as processor architecture or implementation details. The toolset and design flow are explained in more detail in Chapter 3.

2.5 Compiler Support for Data-Level Parallelism

Compilers provide a means to convert human readable source codes to programs that can be run on target machines. Compiler frameworks provide complete software modules for designers to build their own compilers, which usually requires some level of modifications to some of the compiler components. This section presents the common structure of compilers, and how they support data-level parallelism.

2.5.1 Compilers

Compilers can be generally broken to three main parts: *frontend*, *middle-end*, and *backend*. The typical structure of a traditional three-phase compiler is presented in Figure 2.6. Each of the top level phases consists of a number of sub-phases, and every sub-phase takes care of a specific task (such as syntactic analysis, optimizing, or code generation) and transforms the source program from one representation to another [22].

By dividing the compilation process to individual phases, compilers can be designed as a group of independent modules. Modular design allows attaching and replacing only the necessary or desired components (such as additional optimization phases) for the compilation sequence, helping the reusability, maintainability and flexibility of the complex compiler software. For instance, to support different programming languages, the frontend can choose between alternative modules, each of which implementing the analysis and code parsing for a specific programming language. Depending on the language of the input program, a suitable frontend module can be selected for the compilation.

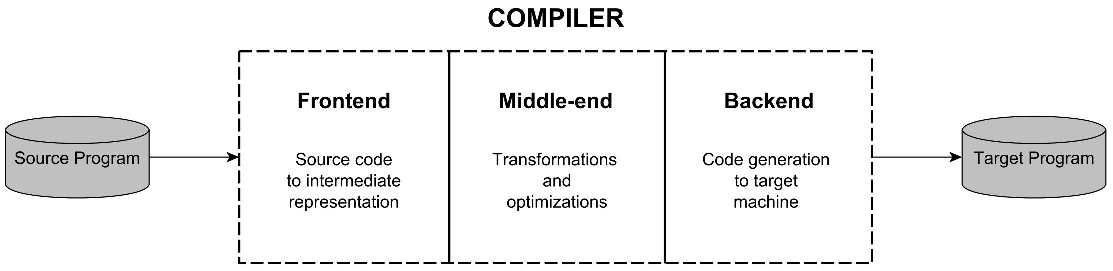

COMPILER
Frontend
Source code to intermediate representation
Middle-end
Transformations and optimizations
Backend
Code generation to target machine
Source Program
Target Program

Figure 2.6: Structure of a three-phase compiler.

Intermediate Representation

Most of the time the source program is in the form of a data structure called *Intermediate Representation* (IR), also called *Intermediate Code*. Intermediate representation is the compiler's general presentation format of the program, which is easier to analyze, optimize, and transform from one form to another. The program is treated as a program for an abstract machine, which consists of an abstract instruction set and an unbounded number of pseudo-registers. Intermediate representation is created from the source code in the compiler frontend, and is usually mostly target and source language independent.

Intermediate representations can be grouped to structural (trees and graphs) and linear (pseudo-code for abstract machine) representations. Types that are combinations of both also exist. During compilation, compilers might construct one or more intermediate representations, possibly having varying forms and abstraction levels. High-level abstract syntax trees may be created for dependence or language related checking, middle-level language-independent form for optimizations, and almost machine-dependent low-level form for optimizations and late compiler phases, such as instruction selection or register allocation. The IR is usually passed as a compact in-memory object between compiler phases for performance reasons. [22]

One common form of the IR is a three-address code, which is not a tree structure, but a sequence of program steps. In three-address code the program is split to *basic blocks*, which are sequences of statements that are always executed sequentially without branching [23]. Another, *Static Single-Assignment* (SSA) form, is a three-address code variation, which differs from it by assigning each variable only once [23]. Both forms are exploited in compiler optimizations, and they may be used in examining different flow analyses of the program.

Frontend

The frontend is the first part of the compiler, which starts the compilation process by performing analysis phase for the source code. Correct syntactic structure (variables, declarations, statements) and static semantic constraints of the source program are checked [24]. The source program is broken into pieces and structured grammatically. If errors are found, the programmer is informed of them [23]. If the analysis phase is passed, the intermediate representation of the source program is generated.

Frontends can support various language extensions, which increase expressiveness of high-level languages. The extended features can allow better maintainability to programs that are aimed to a specific problem domain, while they potentially also allow domain-specific optimizations [25].

High-level vector programming can be enabled by using *vector extensions* in the source language, if the frontend supports the extension. Vector processing is typically expressed in the source code by explicitly defining the desired operations from SIMD instruction sets with built-in intrinsic functions. However, intrinsics are not architecture independent and weaken the portability of the source code. Vector extensions allow high-level vector types and programming in a machine-independent way, which maintains the portability of the source code.

Middle-end

Whereas the frontend is dependent of the source language and the backend is dependent of the target machine, the middle-end is mostly independent of both. During the middle-end (also called optimizer), the program can be thought of as a program for an abstract machine and it is run through a series of transformations and optimization phases. The number of total transform and optimization phases usually depends on the given optimization switch: the larger the switch number, the more phases are run. The optimizations mostly aim at improving the efficiency of the program by reducing execution time, memory consumption, or consumed energy [24].

Backend

The middle-end hands the optimized IR over to the backend, which translates it to target machine specific code. In addition to frontend's and middle-end's target machine independent optimizations, backend usually does final machine-specific optimizations on the IR. The flexibility of the backend is the key element in *retargetability*. Backend usually consists of at least three primary phases: *instruction selection*, *register allocation*, and *instruction scheduling*. Figure 2.7 illustrates the phases in one of the typical execution orders, but it may also vary between compilers.

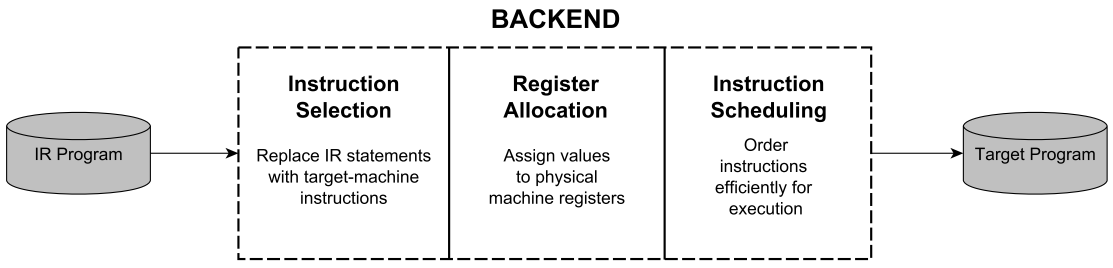

BACKEND
IR Program
Instruction Selection
Replace IR statements with target-machine instructions
Register Allocation
Assign values to physical machine registers
Instruction Scheduling
Order instructions efficiently for execution
Target Program

Figure 2.7: Primary phases in code generation.

The instruction selection phase maps IR statements to target machine assembly instructions [26]. For instance, vector instructions in the IR will be mapped to the machine's SIMD function units, if possible. If not, vector instructions are usually split to smaller vector instructions or ultimately to scalar instructions, and executed on the scalar hardware. The instruction selector has an important role in generating efficient code, as IR operations can be usually executed in various target machine instruction sequences, with significand cost differences [23]. Instruction selection is typically performed for a tree or graph representation, in which parts of the IR code are replaced with target instructions by using pattern-based instruction selection [23].

The register allocation phase is responsible for assigning symbolic variables and intermediate results to physical target machine registers [26]. For instance, vector variables and results are mapped from virtual registers to physical registers that can store temporary vectors of specific sizes. Registers are the fastest way to access operation related data, but usually there are not enough of them to store all operands. Register allocator has to decide which operands should be preserved in registers and which ones need to be stored (*spilled*) to memory for optimal execution [23]. In most environments, a few registers are also reserved for special use, such as stack pointers [23].

The instruction scheduling phase rearranges the selected instructions for optimal machine resource usage. For TTAs, the compiler aims to schedule the instructions into parallel wide instruction words. However, it has to adhere control-dependence, data-dependence, and resource constraints in order to produce code that retains the original meaning of the program [23]. As TTAs are statically scheduled, the scheduler has an important role in producing code that utilizes processor resources as effectively as possible and minimizes stalls in the pipeline. More about TTA instruction scheduling can be found in [27].

2.5.2 Vectorization

Programs usually contain computation on scalars, which could be converted to vector computation. Vector programming in a high-level language has conventionally been slow and prone to errors due to explicit usage of embedded inline assembly or intrinsic functions [28], which require expertise from the programmer. Vector extensions were introduced previously as a solution for machine-independent high-level vector programming. In addition, modern compilers often include a vectorizer as part of optimization modules. Vectorizers aim to transform parallel scalar computation automatically to vector computation.

A vectorizer does analysis for the IR program and replaces scalar computation with vector operations without changing the original semantics of the program [29]. If the underlying hardware supports SIMD execution, the program execution can benefit as described in section 2.1.2. A typical way to extract DLP is by using loop-level vectorization, which vectorizes code by running transformations on loops. Vectorizable loops consist of iterations through array elements from index 0 to N-1, and in every iteration the elements are used by some statements in the loop. If the elements do not have dependencies throughout the iterations and there are no other issues (such as too short iteration count), scalar operations can be merged to vector operations with reduced loop iteration count.

Vectorization can not be always done in compile-time due to potentially overlapping pointer accesses. The problem can be managed by generating both scalar and vectorized version of the same code segment along with additional logic that checks if the pointers overlap in run-time. If the pointers do not overlap, there are no dependencies and execution can be directed to the vectorized code segment.

In addition, although a vectorizer might find several code areas that could be vectorized, it may not vectorize all of them. Vectorizers use cost models to estimate if vectorizing the code is eventually worth it [30]. For instance, if a loop algorithm contains lots of memory accesses that are not in adjacent memory locations (*non-unit stride access*), loading the subwords separately may cause so much overhead that the vectorization would bring very little benefits, or even slows the program execution down compared to the scalar execution. Another diminishing factor is the cost of packing scalars to vectors, and possibly unpacking them back to scalars. The usage of unpacking depends on how the result data is used after computing the result.

3. TTA-BASED CO-DESIGN ENVIRONMENT

3.1 Design Flow

In TCE, processors are designed by using an iterative design space exploration process. It aims to discover the best possible set of processor components and their implementations to execute the target application within the design requirements [31]. The exploration process and the whole TCE design flow consist of several steps, which are visualized in Figure 3.1. The white boxes present the design flow tools, and the light gray containers present the data that is used in different phases in the flow.

The requirements for this thesis project consisted of bringing SIMD capabilities to all essential parts in the TCE design flow. The flow will be explained briefly, after which a more detailed overview is done for tools and data structures that were relevant in this thesis. In addition, fundamentals of the LLVM compiler framework are presented before introducing the TCE compiler. The key areas that required modification for the SIMD support are pointed out as the design flow is presented forward.

The co-design flow starts with having the high-level language source codes of the target application (HLL Program) and the design requirements. An initial processor architecture is customized by using the processor designer tool (*ProDe*). TCE provides an initial architecture with a few components, which can also be used as the starting point for the customization. Operations can be added to the processor function units from operation set libraries.

Once the initial processor architecture is ready, the program is compiled for it by using the TCE compiler (*tcecc*). The compiler takes the architecture and the source program as input and produces a software model of a parallel TTA program that can be run for the processor in TCE simulators [32].

TCE has two simulators: *ttasim*, which provides a command line interface, and *Proxim*, which provides a graphical user interface. The simulators produce simulation statistics, which are utilized to customize the processor again to more enhanced architecture. Implementation model of the processor defines hardware implementations for the architecture components, and can be used to estimate the chip area and power consumption of the architecture. Hardware implementations are stored to TCE-specific hardware database files.

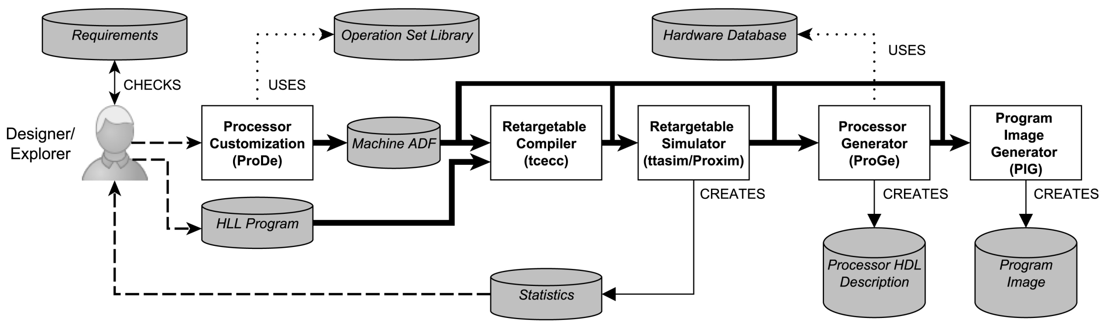

Requirements
CHECKS
Designer/ Explorer
USES
Operation Set Library
Processor Customization (ProDe)
Machine ADF
HLL Program
Retargetable Compiler (tcecc)
Retargetable Simulator (ttasim/Proxim)
CREATES
Statistics
Hardware Database
USES
Processor Generator (ProGe)
CREATES
Processor HDL Description
Program Image Generator (PIG)
CREATES
Program Image

Figure 3.1: TCE design flow.

After the architecture has been customized based on the feedback from the simulation, the program is compiled for the new architecture, simulated and reviewed. This same iteration is repeated until a satisfactory architecture is found within the design requirements. In addition to the processor architecture tailoring, the designer may also be able to make manual adjustments to the source program for more optimal execution on the current architecture. If the designer wants to apply automation to the exploration process, TCE includes an exploration tool called *Explorer* for this purpose.

After an acceptable architecture has been found, the design flow can continue to processor hardware generation. The processor generator (*ProGe*) creates synthesizable RTL implementations for the processor RFs, FUs, and the interconnection network. The implementations are generated by using the architecture implementation model. The compiled program is also converted to actual instruction memory bit image with the program image generator (*PIG*) before uploading it to memory. After the RTL description and the program bit image have been generated, they can be deployed to a *Field-Programmable Gate Array* (FPGA) or used to create an *Application-Specific Integrated Circuit* (ASIC).

3.2 Processor Customization

ProDe is used for designing the architectural models of processors. Processor components that can be customized include register files, function units, buses and sockets. Processor architectures are serialized to XML-based *Architecture Definition File* (ADF) description format, which stores only the information that is needed to generate valid programs for the processor [33]. ADF is the general description format that is used for forwarding processor architecture information to other TCE tools, such as the TCE compiler.

Operation Set Abstraction Layer

Static properties and simulation behavior for operations are defined in an operation set library called *Operation Set Abstraction Layer* (OSAL). Operations defined in OSAL are used in customizing processor functions units, which define the instruction set of the processor. OSAL libraries can be modified and new ones created with operation set editor (*OSEd*) tool.

Static properties of operations are stored into XML-based *.opp* files. An operation has several fields to express its properties, such as name, memory access type, number of input and output operands, and operand attributes. The semantic type of an operand is defined with `type` field. Operand definitions can be currently used to define an operand to be a scalar of some type, which needs to be modified in order to support vector operands for SIMD operations.

OSAL operations require a behavior model definition, which describes the execution semantics for an operation. The behavior model is used for imitating operation's hardware execution with software simulation. It can be defined in two ways: either by writing a C++ behavior description to a *.cc* file, or by describing the behavior as a *directed acyclic graph* (DAG) by exploiting other pre-defined OSAL operations.

The DAG-based behavior descriptions can be expressed with one or multiple `trigger-semantics` field definitions, which is part of operation's static properties. In addition to using the DAG behavior model in simulations, it can also be used for creating instruction selection patterns for the compiler. DAG descriptions make describing the behavior model – especially in the case of SIMD operations – easier for operations that can be built from other operations. It also helps the maintenance of behavior models: if a fix needs to be done to an operation's behavior, which is exploited in a DAG of another operation, the changes are automatically updated to the other behavior models.

`SHL2ADD` operation definition from TCE's *base.opp* file is shown in Figure 3.2. It presents the static properties and the DAG-based behavior description. The properties define the operation name, description, number of input and output operands, operand attributes, and the DAG behavior. Operand attributes state an identifier number and the type of the operand, which is signed integer word in this case. The operation definition can also have other fields that are not shown here, such as `<writes-memory/>` or `<reads-memory/>` to indicate memory accessing.

`SHL2ADD` is meant for calculating an index position for arrays with 32-bit elements. The DAG behavior is expressed by using TCE-specific macros between `<trigger-semantics>` and `</trigger-semantics>` fields. For this operation, it is described as a result of `SHL` and `ADD` operations. `SimValue` is a data structure for representing any types of values in simulations, and is used in TCE as a wrapper to contain the bits of operand values. It can be also used as a temporary value

```
<operation>
  <name>SHL2ADD</name>
  <description>Array indexing for 32-bit data types</description>
  <inputs>2</inputs>
  <outputs>1</outputs>
  <in id="1" type="SIntWord"/>
  <in id="2" type="SIntWord"/>
  <out id="3" type="SIntWord"/>
  <trigger-semantics>
    SimValue shifted;
    EXEC_OPERATION(shl, IO(1), 2, shifted);
    EXEC_OPERATION(add, shifted, IO(2), IO(3));
  </trigger-semantics>
</operation>
```

Figure 3.2: Static properties of the SHL2ADD operation.

storage in DAG descriptions, like the `shifted` variable. `EXEC_OPERATION` defines an operation execution with given input and output operands. `IO(`*id*`)` can be used to pass input and output operands to other operations in the DAG. The first input operand is first shifted two bit positions to the left (32-bit indexing), after which it is added together with the second input operand and the result is written to the output operand.

C++ behavior models are described by using TCE-specific restricted C++ language, which is then compiled to an *.opb* plugin module file [33]. The behavior description for the logical left shift `SHL` operation in *base.cc* file is shown below. The behavior description starts with stating the operation name with `OPERATION(`*opname*`)` field. The *opname* is the same OSAL operation name that is defined in the static properties. The code section between `TRIGGER` and `END_TRIGGER;` is used for describing the actual behavior.

```
OPERATION(SHL)
TRIGGER
    IO(3) = UINT(1) << UINT(2);
END_TRIGGER;
END_OPERATION(SHL)
```

`SHL` operation has two input operands and one output operand defined in its static properties. Input operands can be accessed with `INT`, `UINT`, `FLT`, `DBL` and `HFLT` macros depending on how the bits in the operand should be interpreted. The macros return the operand bits that are contained in a `SimValue` object as the desired type. In `SHL`, the input operands are accessed with the `UINT(`*id*`)` macro, which presents the operands as unsigned integer values. The `IO(`*id*`)` macro must be used when values are assigned to output operands.

As operand attributes and `SimValue` did not support vector operands, only scalar values could be used in the C++ and DAG descriptions. Both were extended to support vector operands and `SimValue` modified to provide ways to access individual vector elements of different size. This allowed creating the DAG behavior models for vector operations by exploiting the scalar counterpart to compute values for independent vector elements. After these changes, SIMD operations could be defined to OSAL libraries and added to processor functions units. The width of register files, ports and buses could be already modified to arbitrary widths, and required no changes.

OSAL Tester

TCE toolset includes a simple program called *testosal* for debugging operation behavior models [33]. It is used by creating input commands, which state the operation name and input operand values. The bits of the input values are stored to `SimValue` objects, which are used in the behavior semantics. Input values can be given in different formats, such as integer or hexadecimal numbers. Output value(s) are calculated by executing the behavior model, after which they are printed. The type of the printed result values can be set with `!output` command. Below is an example of a command line input for scalar `SHL` operation, whose behavior model was introduced previously.

```
>> !output hex
>> SHL 0x12345678 0x10
0x56780000
```

On the first line, *testosal* is set to print result values from operations in hexadecimal format. On the second line, the `SHL` operation is listed with two hexadecimal input values. The latter value (integer number 16) defines the number of bit position shifts to the left for the first value. Two `SimValue` objects are created and initialized with the input values. The objects are passed to the behavior model, which then executes the operation and calculates the output operand value. The value is printed, and as a result, the bits in value `0x56780000` have been shifted 16 bit positions to the left.

OSAL Tester supported expressing and printing scalar values in the command line listing. As the OSAL was extended with vector features and new vector operations were created, their behavior models needed to be tested in *testosal*. Thus, OSAL Tester was modified to initialize and print `SimValue` objects with wide vector values.

3.3 LLVM Compiler Infrastructure

Low Level Virtual Machine (LLVM) is a compiler infrastructure, which consists of modular and reusable compiler and toolchain technologies [34]. TCE uses LLVM as the basis for its *tcecc* compiler, which is responsible for compiling source programs for TTA machines. LLVM is a three-phase compiler and the compilation flow at simplest is the same as presented in Figure 2.6. The compilation flow starts with compiling the source program to *LLVM Intermediate Representation* in the frontend.

3.3.1 Clang Frontend

In order to take full advantage of DLP resources on the architecture, programmers need to be able to explicitly describe vector calculations in the source code. Clang is a C/C++/Objective-C compiler, which aims to be powerful and comprehensive frontend for the LLVM compiler [34]. It supports GCC, OpenCL, AltiVec and NEON vector extensions, which can be exploited to provide high-level language support for vector programming [34].

OpenCL vector types can be created by type defining a scalar type to a vector type with Clang `ext_vector_type` attribute, which sets the desired element count for the vector type. Below is an example of a four-sized integer vector type definition. The vector type is defined by stating the desired type definition symbol (`int4`), which is followed by the `__attribute__` keyword and the vector extension (`ext_vector_type(`*elem. count*`)`) definition inside brackets.

```
typedef int int4 __attribute__((ext_vector_type(4)));
```

The vector type definition can then be used to create vector variables and perform vector computation on them. Below is an example how the vector type is used in a function implementation to perform multiply-accumulate operation on input vectors in C.

```
int4 multiplyAccumulate(int4 a, int4 b, int4 c) {
  int4 result = a * b + c;
  return result;
}
```

The source code does not basically differ from the same function with scalar integer operands. The only difference in the code can be seen in the parameter and return value type declarations, which are of vector type `int4` instead of plain `int`. The vector types provide an easy way to do vector programming on high level.

3.3.2 Intermediate Representation

The LLVM IR is used as the common code representation throughout all compilation phases. The IR aims to be light-weight, low-level while being expressive, typed, and extensible at the same time. It is designed to be used as an in-memory compiler IR, as an on-disk bitcode representation, and as a human readable assembly language representation, allowing the LLVM IR to provide powerful representation for different compiler optimizations and analysis. [34]

The IR code uses three-address representation and has a strong type system for values that can be produced by instructions. In the instruction selection phase, values are assigned to virtual registers. Primitive scalar types are represented as integer types (i1, i8, i16, i32, etc.) or floating point types (half, float, double, etc.). The IR provides DLP support with vector types, which must be a group of elements of the same type. A vector element type can be either an integer or a floating point type, or a pointer to other of the two types [34]. Vector types are defined by using <'element count' x 'element type'> notation in the IR code. For instance, `<4 x i32>` type defines a four-sized vector with 32-bit integer elements and `<16 x float>` a vector with sixteen single-precision floating point elements.

Vector and scalar operations are expressed in the same way in the IR code, only the operands in the operation statement differ. Operands can be expressed as constant or temporary values. The operation definition for the LLVM `add` operation is presented below. The definition starts with the operation name, after which the type (`<ty>`) of the operands must be defined. After the type definition, both input operands are listed and separated by a comma.

```
<result> = add <ty> <op1>, <op2>
```

An example of the addition operation using different vector operands is presented below. The first line defines an addition between two temporary values, the second an addition between a constant vector value and a temporary value, and the third an addition between two constant vectors.

```
<result> = add <4 x i32> %var1, %var2
<result> = add <4 x i32> <10, 20, 30, 40>, %var
<result> = add <4 x i32> <5, 6, 7, 8>, <4 x i32> <-9, 3, -1, 3>
```

In all cases the operation results in a new 128-bit wide vector variable with four sum elements from the executed operation. The compiler could easily optimize the operation on the third line by pre-calculating the result in compile-time.

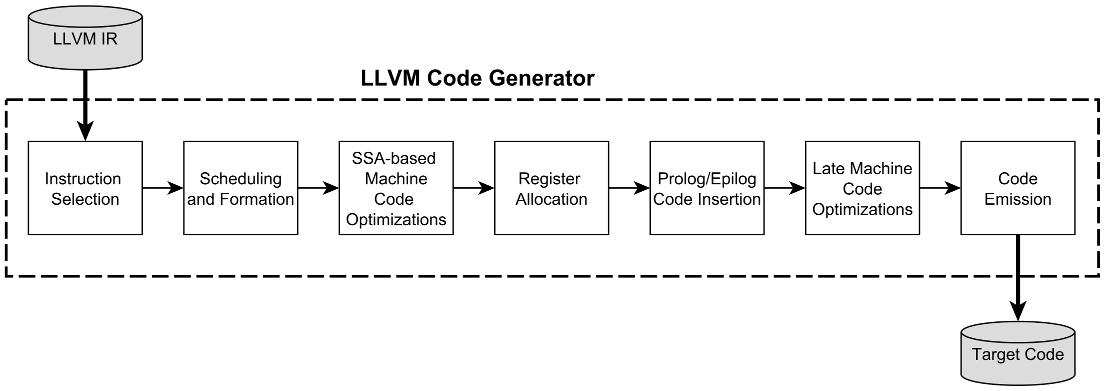


Figure 3.3: Flow of the LLVM code generator.

3.3.3 Backend

New backends can be built by utilizing the LLVM code generator framework, which is provided as part of the LLVM tools. It provides a set of reusable components for converting IR programs to target machine code [34]. Figure 3.3 presents the main phases in the code generator.

The flow starts with instruction selection, which converts the IR into a DAG of target machine instructions. Operation operands are in virtual registers. In the next phase the DAG instructions are assigned to a new order depending on the different constraints of the target machine. For example, the scheduler can order instructions in order to hide instruction latencies. The next phase is optional and performs a series of machine-code optimizations on the SSA-form of the program. The code is then converted from using an infinite virtual register file to using concrete registers. Virtual register references are eliminated and mapped to the target machine's register files. In case the number of physical registers is not enough, spill code is also generated in this phase to move additional register values to memory. In the next phase the prolog and the epilog code is generated for functions, and abstract stack references are resolved to real stack references. Final machine code optimizations, such as spill code scheduling, may be done before the code emission. [34]

When a new backend is created, several abstract interfaces need to be implemented for the code generator. The most important interface is *TargetMachine* base class, which will be derived to a target-specific subclass to provide a backend for the machine. The subclass implements the virtual methods from the base class, which provide methods for accessing target-specific information [34].

Instruction Selector LLVM uses a *SelectionDAG*-based instruction selector, which translates the LLVM IR code to target machine instructions. The SelectionDAG provides an abstract directed acyclic graph representation of the IR code. It represents the code as nodes, which is well-suited for different phases in the code generation, such as instruction selection, scheduling, and very-low-level optimizations. [34]

The instruction selector consists of several phases, of which the most important ones are the legalization of operand types, the legalization of operations, and the actual instruction selection. The legalization phases convert the DAG to only use types and operations that are supported natively by the target machine [34]. For example, unsupported vector operands are split until a supported vector type is found, or all elements are ultimately converted to scalars. After the legalization, IR operations are associated with target machine instructions by using pattern matching.

LLVM provides a skeleton base class of the instruction selector, and some parts of it needs to be implemented separately to make the selector work with the target machine. Some parts of the instruction selector are implemented automatically by the LLVM *TableGen* tool, which generates C++ code implementations from target descriptor files. However, all instructions can not be expressed with the descriptors, and some of the method implementations need to be made manually.

Register Allocator The register allocator converts the code from using virtual register file references to using target machine register files. LLVM provides three register allocators for different allocation purposes, of which the default "greedy" allocator aims to minimize the cost of spill code [34]. Register allocators are machine independent and do not require modifications to work with a target machine.

Target Descriptor Files

LLVM uses *target descriptor* (*.td* suffix) files to express a great deal of domain-specific information of target machines, such as the instruction and register set. Target descriptor files are processed by *TableGen*, which converts the descriptor files to C++ code that is used in code generation in the backend, saving the backend designer from a great deal of repetitive work. In order to reduce the amount of redundant descriptions and make it easier to structure target machine related information, *TableGen* is specifically designed to allow writing flexible domain-specific descriptions. [34]

TableGen files consist of *records*, which express the domain-specific information. Records consist of abstract "classes" and concrete "definitions". Classes are used to build abstractions of the domain information, and *TableGen* syntax allows creating new and more domain-specific classes by inheriting existing classes. Derived classes can reduce the amount of redundant duplication considerably. Concrete definitions that express the domain-specific attributes are created by using the `def` keyword.

Register

Target-specific registers are defined by using `Register` class. The class has the following attributes:

```
  class Register<string n> {
    string Namespace = "";
    string AsmName = n;
    string Name = n;
    int SpillSize = 0;
    int SpillAlignment = 0;
    list<Register> Aliases = [];
    list<Register> SubRegs = [];
    list<int> DwarfNumbers = [];
}
```

The parameter `n` defines the name of the register. Other attributes, such as subregisters, are not specified. More complex classes for registers are usually created by deriving the `Register` class. Below is an example of a class definition for X86 registers, which is used as a generic register class.

```
  class X86Reg<
   string n, bits<16> Enc, list<Register> subregs = []> : Register<n> {
    let Namespace = "X86";
    let HWEncoding = Enc;
    let SubRegs = subregs;
  }
```

The subclass defines two new parameters, encoding bits for the register and an optional subregister list, which is empty by default. Base class attributes can be overridden with the `let` expression in derived classes. New registers can be defined by using the derived class. Below is a simplified example of a register definition from *X86RegisterInfo.td*.

```
  def ZMM0 : X86Reg<"zmm0", 0, [!cast<X86Reg>("YMM0")]>,
    DwarfRegAlias<!cast<X86Reg>("XMM0")>;
```

`ZMM` registers are SIMD registers used by AVX-512 instructions. The register uses the derived `X86Reg` class, defining the register name as “zmm0”, encoding bits to 0, and `YMM0` as a subregister. `cast<>` operator is an assertion check if the given parameter is not an instance of the desired type. `DwarfRegAlias` class declares that a given register uses the same dwarf numbers as the another one, and is useful if two registers should have the same number. Dwarf numbers are used by debugger tools to describe where values may be located during execution.

RegisterClass

`RegisterClass` classes define a group of registers for a set of value types that can use the registers. In addition, the class defines the default allocation order of the registers. The class has four arguments:

1. Name of the namespace,
2. List of LLVM value types the register class uses,
3. Alignment of the registers when stored or loaded to memory, and
4. List of registers that belong to the class. [34]

An example of how the AVX-512 SIMD registers are defined under a `RegisterClass` record is shown below. The record name is `VR512` and the register class is set under “X86” namespace. The value type list includes all vector value types that are 512 bits wide, and the alignment is set to the whole vector width. The 512-bit `ZMMx` registers are listed in ascending order from index 0 to 31 by using TableGen’s `sequence` keyword. Registers can also be listed by writing the individual register names by hand, each name separated by a comma.

```
def VR512 : RegisterClass<"X86", [v16f32, v8f64, v16i32, v8i64],
  512, (sequence "ZMM%u", 0, 31)>;
```

Instruction

`Instruction` class can be used to define instruction records, though it is commonly used to create more complex and target-specific instruction subclasses. *TableGen* uses instruction descriptors to generate implementations to parts of the instruction selector. Part of the class structure is shown below.

```
class Instruction {
  string Namespace = "";
  dag OutOperandList;
  dag InOperandList;
  string AsmString = "";
  list<dag> Pattern;
  ...
}
```

Input and output operands are given as a list in DAG format. Operands are named so that they can be referred to in the instruction assembly string and in the pattern definition. The assembly string attribute is used by the assembly printer if assembly output of the program should be produced. The instruction pattern is used by the LLVM instruction selector to convert the IR code in a DAG representation to a new DAG representation, in which the IR instructions are changed to target instructions. If the instruction pattern matches an instruction that exists in the IR, the instruction record can be used to replace the IR instruction. An example of a derived vector instruction class from the *NVPTXVector.td* (Nvidia Parallel Thread Execution) file is shown below.

```
class NVPTXVecInst<dag outs, dag ins, string asmstr, list<dag> pattern,
  NVPTXInst sInst=NOP> : NVPTXInst<outs, ins, asmstr, pattern> {
  NVPTXInst scalarInst=sInst;
}
```

NVPTXVecInst is an instruction class for vector instructions. It has been derived from the NVPTXInst instruction class, which in turn has been derived from the Instruction base class. The derived class takes the same parameters as the NVPTXInst base class, except for the optional scalar instruction parameter, which is set to no-operation by default if not provided. An example of an instruction record definition using the derived instruction class is shown below.

```
def V4i32Extract : NVPTXVecInst<
  (outs Int32Regs:$dst),
  (ins V4I32Regs:$src, i8imm:$c),
  "mov.u32 \t$dst, $src${c:vecelem};",
  [(set Int32Regs:$dst, (vector_extract
    (v4i32 V4I32Regs:$src), imm:$c))], IMOV32rr>;
```

The record defines an instruction for extracting an i32 element from a v4i32 vector value. Input operands include the v4i32 vector and an immediate integer number, which indicates the element index that should be extracted from the vector. The output operand is the extracted i32 element. An assembly string has also been defined for assembly code generation. The pattern part consists of two pattern fragments. The extract operation for specific operand types is indicated with vector_extract in the inner node. The outer set matches the whole pattern to a result of a vector extract with the given operand types. The scalar correspondent for the vector instruction is defined as the IMOV32rr operation.

More than one instruction record can be created for the same hardware operation. For example, a memory load operation can have a record, in which both the address and data operands are register operands. Another record is also usually defined, in which the address operand is a constant value.

TableGen allows specifying arbitrary selection patterns, which are replaced with the desired instructions. Arbitrary selection patterns can be defined by using the `Pat` class.

```
class Pat<dag pattern, dag result> : Pattern<pattern, [result]>;
```

The first argument is a DAG representation of the pattern that should be replaced, and the second argument is the replacement DAG, which consists of existing instruction records. An example of the `Pat` usage is shown below. The definition is from *X86InstrAVX512.td*.

```
def : Pat<(xor VK16:$src1, (v16i1 immAllOnesV)),
          (KNOTWrr VK16:$src1)>;
```

The pattern on the left presents an "exclusive or" operation between a `v16i1` boolean vector operand (`$src1`) and an immediate vector operand of the same type, in which all bits are ones. An "exclusive or" operation, in which the other operand consists entirely of ones is also a "not" operation, and thus, the result DAG states that this particular "exclusive or" should be replaced with `KNOTWrr`. If the pattern shows up in the IR code, it will replaced with the "not" instruction.

3.4 TCE Compiler

Parallel TTA programs are created by using the automatically retargeting *tcecc* compiler. The main phases of the compiler and code generation are presented in Figure 3.4. The frontend and middle-end consist of the LLVM framework tools, while TCE is responsible for implementing the code generation in the backend. Unlike traditional LLVM backends, TCE backend is implemented as a stand-alone code generator library, which utilizes the LLVM code generation libraries, and thus, is separate from the LLVM compiler tools [32].

Compilation starts by giving the source program to the frontend. The program is run through the basic LLVM tools until the IR format is handed over to the TCE backend from the middle-end as one IR module. In addition to the IR program, the TCE backend also takes the target machine ADF as input to dynamically retarget itself to the architecture. The backend is divided into *static* and *dynamic* parts. The dynamic part is required to achieve the flexible retargetability of the compiler.

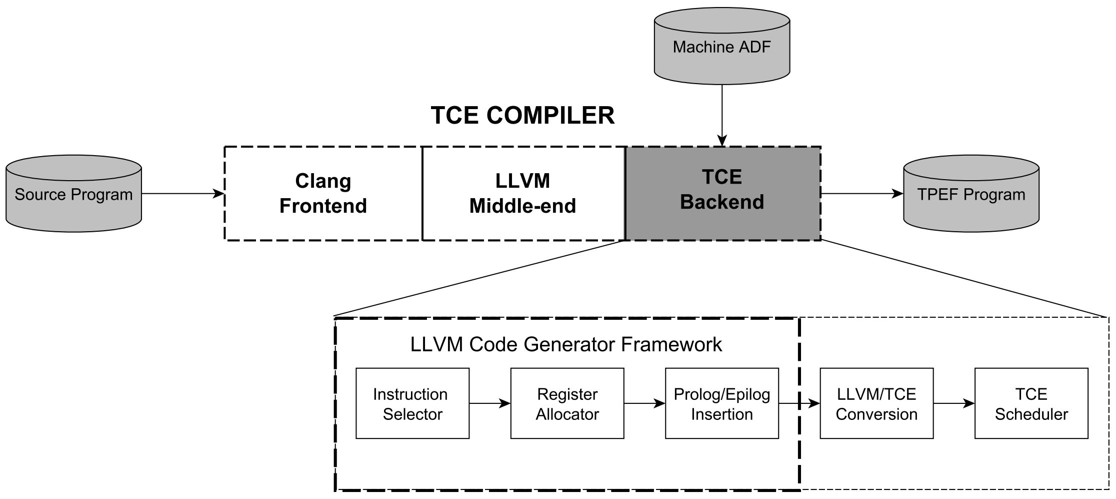


Figure 3.4: Compilation flow of the TCE compiler (adapted from [32]).

The code generation is divided to two phases. A sequential target machine program is first produced by utilizing the LLVM code generator framework (instruction selector, register allocator, prolog/epilog inserter), after which the sequential program is scheduled to a parallel program by using the TCE instruction scheduler. Before passing the sequential program to scheduler, it is converted from the LLVM data structure representation to a TCE program object model by *LLVM-POMBuilder*, which is implemented as a separate LLVM code generator pass. [32]

The compilation produces a *TTA Program Exchange Format* (TPEF) file, which is a binary file format for passing TTA programs in the TCE toolset. TPEF contains the program model in an assembly-level intermediate representation that is used by some TCE tools, such as *ttasim*. TPEF files can be disassembled to a human readable assembly code for a given machine ADF with the TCE Disassembler (*tcedisasm*) tool. The disassembled program shows the program as moves between registers and function units.

As a machine ADF may contain SIMD vector resources, such as vector operations or wide registers for vector operands, the compiler needs to be able to utilize them whenever needed. Vector instructions from the IR code should be mapped to machine vector operations that provide the same functionality on hardware. This required expressing the SIMD resources from TCE-specific ADF and OSAL formats to LLVM-specific target descriptors. The new vector-related target descriptions mainly affect instruction selection and register allocation in the TCE compilation process.

3.4.1 High-Level Language Support

In order to provide easier access to high-level (and target independent) vector programming, programmers using TCE have to be able to use explicit vector variable types. Such capability was brought to C and OpenCL C languages by utilizing the previously mentioned Clang vector extensions. The high-level operations on vector operands will automatically be converted to vector instructions in the IR code. As the SIMD resources are described by using the target descriptors, the instruction selector utilizes the SIMD resources automatically to execute the vector instructions in the IR.

TCE Intrinsics

A hardware operation can be called directly in the program source code by using TCE's operation intrinsics. Intrinsics provide an easy way for calling complex custom operations, that can not be selected via normal instruction selection in the code generation. The semantics for an operation intrinsic call is shown below.

```
_TCE_opname(input operand 1, ..., output operand 1, ...);
```

The operation intrinsics can be used in source codes by including the *tceops.h* header file. This file is automatically generated in every compilation and it lists operation intrinsics for all OSAL operations. The caller is responsible for making sure the input and output operands have the correct type, or incorrect results may occur.

3.4.2 Static and Dynamic Backend Parts

In order to adapt to the dynamic nature of the customizable architecture, the compiler backend is divided to static and dynamic parts. *TargetMachine* interface is implemented by deriving the *TCETargetMachine* subclass from the *LLVMTarget-Machine* base class. It encapsulates all properties and code generation methods of target machines behind one interface, which is illustrated in Figure 3.5. The figure presents an abstracted structure of *TCETargetMachine*, as many of the backend related classes are not shown. For a more detailed overview of the TCE backend, refer to [32].

The static part of the backend contains properties that are common to all TTA processors derived from the architecture template. Source codes of the static part need to be compiled only once, after which a recompilation is required only if modifications are made.

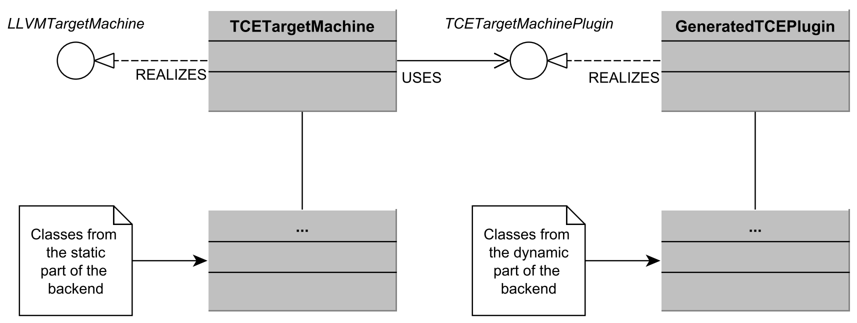


Figure 3.5: Structure of TCE Target Machine [32].

The dynamic part of the backend contains the properties that can be altered in the processor template, such as the instruction set and the register set. *TCETargetMachine* uses the *TCETargetMachinePlugin* interface to access information that is related to the dynamic parts of TTA machines. The dynamic part is compiled in every compilation to a separate plugin, which is then loaded to realize the *TCETargetMachinePlugin* interface. *GeneratedTCEPlugin* is the top level class that realizes the interface and encapsulates all the dynamic properties of TTA machines. The plugin is the main factor in making the backend retargetable to arbitrary architecture configurations.

3.4.3 Dynamic Backend Generation

The main task in implementing the dynamic backend is converting processor architecture information that is modeled with TCE-specific ADF and OSAL description languages to LLVM-specific target descriptors. Dynamic properties of the backend are generated by a class called *TDGen*, which is part of the TCE backend. It is responsible for generating the instruction and register set target descriptor files. Figure 3.6 illustrates the generation of the target machine plugin.

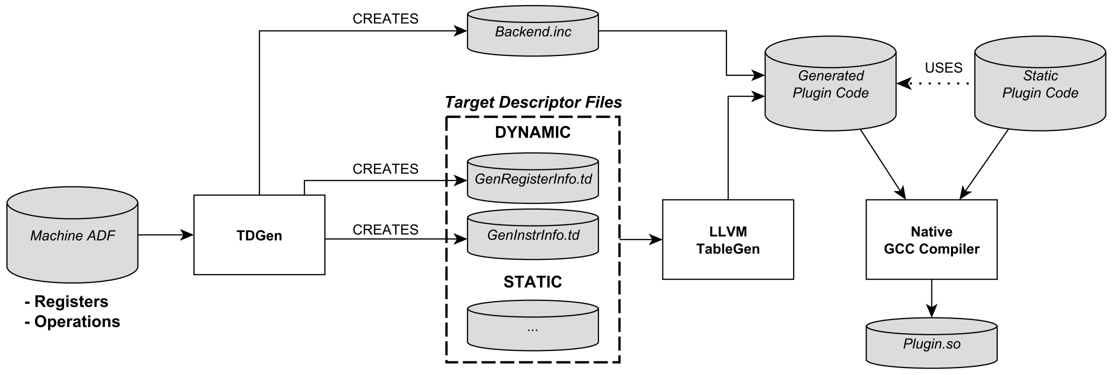


Figure 3.6: Generation of Target Machine Plugin (abstracted from a figure in [32]).

TCE has several static and dynamic target descriptor files included in the compilation process. The static files contain formats and records that are common for all machines. Dynamic files are generated by *TDGen* and used to describe the dynamic properties of target machines. Only the most relevant target descriptor files for this thesis are shown in the figure.

TDGen takes the processor ADF as input and examines register files and function unit operations from the architecture. All register set information is generated to the *GenRegisterInfo.td* descriptor file and instruction set information to the *GenInstrInfo.td* file. *TableGen* processes the target descriptor files and generates C++ implementations for several code generator callback functions and data structures [32]. *TDGen* also generates some C++ implementations for several helper functions that are used in the backend.

Once OSAL was extended to support vector operations and operands, the customized target machine ADF that is passed to *TDGen* may contain wide registers and function units with vector operations. *TDGen* is able to detect scalar features of the architecture and handle the descriptor generation for them. In order to support automatic retargeting to vector resources, modifications were required to the descriptor generation process.

The byte order also requires some attention. Regardless of the memory access operation that is used to move vectors between memory and registers, the order of vector subwords must be preserved. For instance, `v4i8` vector should have a dedicated memory access operation for that specific vector type. However, when the compiler handles `v4i8` vectors, it may optimize value handling by temporarily interpreting the vector as a `i32` value, for which different operations are used for accessing memory. Thus, the new vector operations that access memory must handle the data in the same way with the others to maintain byte and subword order.

3.5 Simulator

This brief overview concentrates on the command line-based *ttasim* simulator. The simulator takes the machine ADF and compiled TPEF as input and uses them to build an *Executable Instruction Memory* structure, which is suitable for simulation [35]. The simulator provides capabilities for debugging, profiling and tracing data for several TCE tools. For example, it is possible to stop the program execution to desired breakpoints, examine values from register files, or print data memory contents. Data memory contents can be printed with the following command.

```
>> x /n 8 /u w sum
```

Character `x` is the command for printing data from the memory. The parameter `/n` is used to indicate how many data words should be printed. Unit size of the data word is indicated with `/u`, which can be 'b', 'h', or 'w' for words of 1, 2, or 4 bytes. The last parameter can be an absolute memory address or a global variable name. The example prints eight 32-bit memory chunks starting from the address of the `sum` global variable.

3.6 Processor Generator

Processor Generator *ProGe* is used to generate synthesizable RTL descriptions of designed TTA processors [36]. *ProGe* uses information from the architecture definition file and from the XML-based *Implementation Definition File* (IDF) to generate the hardware description of the processor.

Whereas the ADF defines the architecture components, the IDF defines which implementations from a *Hardware Database* (HDB) are mapped to those components. Hardware databases contain the HDL definitions and other implementation related data (like cost data) of architecture components. Components can have multiple implementations stored in HDBs. HDB files have *.hdb* suffix, and they can be created and modified with the hardware database editor (*HDB Editor*) tool.

HDB Tester

The hardware database tester (*HDBTester*) is a tool for testing function units and register file implementations from HDBs. In function unit testing, it utilizes the *FUTestbenchGenerator* class, which generates RTL testbench description for the given function units. It uses the operation behavior models to pre-determine expected output values, creates HDL testbench code with the FU under test, simulates it, and compares the values from output ports to the expected values [37].

In order to generate processor RTL for TTA machines with vector function units, the function units need to have implementations. A new tool was added to the TCE toolset that is able to create SIMD function units by exploiting existing scalar function units. The tool automatically creates objects models that define the properties of SIMD function units and stores the FU object models to HDB files. In addition, it generates the RTL implementations of the function units.

By exploiting existing scalar function units the tool was made fully automatic, as its only responsibilities in the VHDL generation are the duplication of scalar FUs and wiring the subwords from vector operands correctly to the scalar FUs. This way most of the vector function units with basic vector operations can be created fast and designers may not have to deal with the RTL code at any point. More complex vector operations may still have to be described manually in RTL.

4. IMPLEMENTATION

4.1 Processor Customization

The first task was to extend the OSAL properties to support vector operands in TCE. In order to support function unit customization with vector operations, the operation set library also needed to be populated with vector operations.

Operation Set Abstraction Layer

Previously, the operands in TCE were assumed to be scalars of some type. In order to express vector operands, two new fields were added to the static properties: `element-width` and `element-count`. The element count defines if the operation is a vector or a scalar. With element count one the operand is a scalar and its total width is just the element width, but for a vector operand its total width is the element width multiplied by the element count.

As the new operand fields were added and vector operands could be described, the TCE operation set library was extended with various vector versions of scalar operations. For example, for every basic float operation (addition, subtraction, etc.) vector versions with different element counts were created. Since this would have required a great amount of manually made and repetitive OSAL descriptions, a Python-based *generate_simd.py* script was added to the TCE source tree to generate the SIMD operations automatically. Figure 4.1 presents how the script extends the operation set library with the SIMD operations.

The script creates vector operations for all basic instruction types, and has been structured so that it is easy to add new vector operations for the automatic generation. The operation set library was extended with vector operations that can process vectors of different sizes up to 1024 bits. This number was picked as an initial target, but the script is generic and can produce vector operations up to any desired width of the power of two.

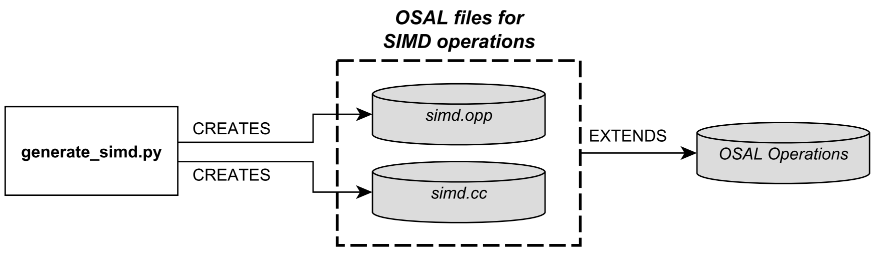


Figure 4.1: The generation script for SIMD operations.

The generated *simd.opp* file contains the static properties for all vector operations. Most of the operations have their behavior model defined there in the DAG format. The rest have their behavior model defined in the *simd.cc* file. `SimValue` was modified to support vector values by defining a wide byte array to its properties. Vector values are assigned to the array bytes, and they can be easily accessed subword-wise by returning a pointer to the array with different pointer types.

A generated SIMD `ADD` operation definition with the element count of four is shown in Figure 4.2. The naming convention is the following: the number before `X` implies the element width, and the number after is the element count. The example operation takes two signed integer vectors of four elements as input, and outputs a vector of the same type. The new `element-width` and `element-count` fields are visible as part of the operand attributes.

Since vector operations can be easily built by using scalar operations, the trigger semantics field can be utilized in most cases. The same simulation behavior pattern applies to most of the vector operations having a correspondent scalar base operation: unpack the N-sized input vector operands `IO(1)` and `IO(2)` to individual scalar operands (a1, a2, a3, a4, b1, b2, b3, b4), execute the base operation on the scalars N times, pack the result scalars (o1, o2, o3, o4) back to a vector operand, and write the vector to the output (`IO(3)`).

```
<operation>
  <name>ADD32X4</name>
  <inputs>2</inputs>
  <outputs>1</outputs>
  <in id="1" type="SIntWord" element-width="32" element-count="4"/>
  <in id="2" type="SIntWord" element-width="32" element-count="4"/>
  <out id="3" type="SIntWord" element-width="32" element-count="4"/>
  <trigger-semantics>
    SimValue a1, a2, a3, a4;
    SimValue b1, b2, b3, b4;
    SimValue o1, o2, o3, o4;
    EXEC_OPERATION(UNPACK32X4, IO(1), a1, a2, a3, a4);
    EXEC_OPERATION(UNPACK32X4, IO(2), b1, b2, b3, b4);
    EXEC_OPERATION(ADD, a1, b1, o1);
    EXEC_OPERATION(ADD, a2, b2, o2);
    EXEC_OPERATION(ADD, a3, b3, o3);
    EXEC_OPERATION(ADD, a4, b4, o4);
    EXEC_OPERATION(PACK32X4, o1, o2, o3, o4, IO(3));
  </trigger-semantics>
</operation>
```

Figure 4.2: Static properties of the ADD32X4 operation.

Some of the vector operations, such as the `PACK` or `UNPACK` operations, can not be expressed by using other OSAL operations. The simulation behavior for the `PACK32X4` operation using a C++ definition is presented below. The packing operation has four scalar inputs, and it outputs a packed vector containing the input scalars.

```
OPERATION(PACK32X4)
TRIGGER
    SUBWORD32P(5)[0] = UINT(1);
    SUBWORD32P(5)[1] = UINT(2);
    SUBWORD32P(5)[2] = UINT(3);
    SUBWORD32P(5)[3] = UINT(4);
END_TRIGGER;
END_OPERATION(PACK32X4)
```

As `SimValue` was modified to support wrapping wide vector values, three new macros were introduced to access individual vector elements. The `SUBWORD8P(`*id*`)`, `SUBWORD16P(`*id*`)` and `SUBWORD32P(`*id*`)` macros return a pointer to the `SimValue` of the desired operand, and can be used to access scalar elements. The macros provide access to subword widths of 8, 16 and 32 bits. The accessed element position is indicated with the square bracket notation.

The `UINT(`*id*`)` macro returns the desired input operand value as a 32-bit unsigned integer scalar. The scalar values are stored to appropriate element positions in the output vector (the fifth operand in the OSAL operand notation) by accessing the elements with `SUBWORD32P`.

When vectors are stored to the memory, the scalar element at index 0 is considered to be the most significant. It is stored to the lowest memory address, whereas the last subword is stored to the last element address position.

OSAL Tester

As vector capabilities have been introduced to TCE, *testosal* users need to be able to define values to vector elements. OSAL Tester was modified so that vector values can be defined with a long hex string. The user needs to pay attention to the bit size of a vector element, and set the hex string accordingly to initialize vector elements with desired values. The example below shows how a four-sized vector with 32-bit elements can be initialized in *testosal*. The `ADD32X4` operation is the same OSAL operation that was introduced previously, and all its operands are vectors of four 32-bit elements.

```
>> !output hex
>> ADD32X4 0x7ffffcc4000002d47fffff2400000094
     0x000002c3000003dc0000008200000147
0x7fffff87000006b07fffffa6000001db
```

The two input vectors are accessed in the DAG behavior description with the `IO(1)` and `IO(2)` macros, as was shown in the XML-based `ADD32X4` operation description above. With `UNPACK32X4` their elements are split to scalar `SimValue` variables. All elements from the vector operands are visualized below.

```
IO(1): 0x7ffffcc4 0x000002d4 0x7fffff24 0x00000094
IO(2): 0x000002c3 0x000003dc 0x00000082 0x00000147
IO(3): 0x7fffff87 0x000006b0 0x7fffffa6 0x000001db
```

Subwords from same element positions are added together with the scalar `ADD` operation, which produces a result scalar. The result scalars are packed back to a four-sized vector by using the `PACK32X4` operation. The result vector is assigned to the output operand `IO(3)`. Vector operations can now be debugged with *testosal*.

4.2 Compiler Support

4.2.1 High-Level Language Support

For DLP purposes, TCE allows using vector data types to provide high-level vector programming. A header file was created to store all the supported vector type definitions, which can be easily included to source codes. An example of different vector types with different subword counts and subword types is presented below.

```
typedef bool  bool16 __attribute__((__ext_vector_type__(16)));
typedef char  char64 __attribute__((__ext_vector_type__(64)));
typedef int   int32  __attribute__((__ext_vector_type__(32)));
typedef float float8 __attribute__((__ext_vector_type__(8)));
typedef half  half4  __attribute__((__ext_vector_type__(4)));
```

Vector data types can be enabled by including the *tce_vector.h* header file. The header file contains all the TCE-supported vector type definitions. It extends the `bool`, `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, and `float` scalar types to various vector types. Subword counts are created in the power of two for each vector type. The subword count starts from count two until the maximum vector width is reached. For example, the vector types for the 32-bit scalar `ADD` are `ADD32X2`, `ADD32X4`, `ADD32X8`, `ADD32X16` and `ADD32X32`.

TCE-supported vector extensions can be used both in C and OpenCL C source languages. In addition, half-precision floating point scalar `half` type can be extended to vector types and utilized in OpenCL C code. Almost all the basic scalar operators apply to the vector types. Operators `!`, `&&` and `||` are not supported, while other basic ones, such as `+`, `-`, `*`, `/`, `&`, `|`, `»`, `==`, or `[]`, are.

4.2.2 Dynamic Backend Generation

The *TDGenSIMD* class examines SIMD resources from TTA machines and generates the target descriptors for them. It utilizes the methods from the *TDGen* base class to handle the scalar resources, and guides handling of the vector resources to its own methods. Figure 4.3 illustrates the same plugin generation as before, except that *TDGenSIMD* is now the dominant generation component.

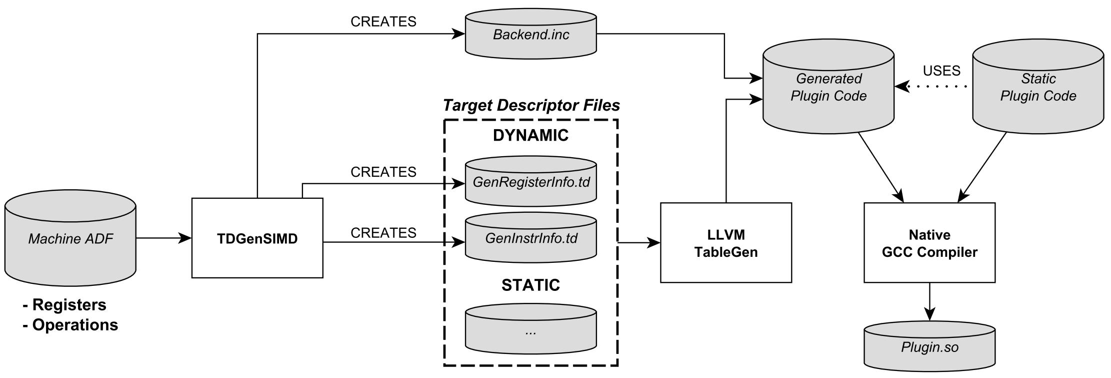


Figure 4.3: Generation of the dynamic Target Machine backend with SIMD capabilities.

In addition, two new classes were added to TCE: *TCEISelLoweringSIMD* and *TCEInstrInfoSIMD*. They are presented in Figure 4.4 with darker colour. Both were derived from existing base classes to separate SIMD and scalar handling. *TCEISel-LoweringSIMD* is responsible for registering the supported vector register classes to the backend. *TCEInstrInfoSIMD* was created to implement *copyPhysReg(...)* method for vectors. The method creates an instruction, which copies a vector value from a register to another.

Register Set Info Generation

All register related information is generated to the *GenRegisterInfo.td* descriptor file. *TDGen* generates register set information for the scalar part of the machine, whereas *TDGenSIMD* is responsible for describing all the vector information. All vector register classes use the *TCEVectorReg* base class, which connects the vector register classes to the *TCE* namespace.

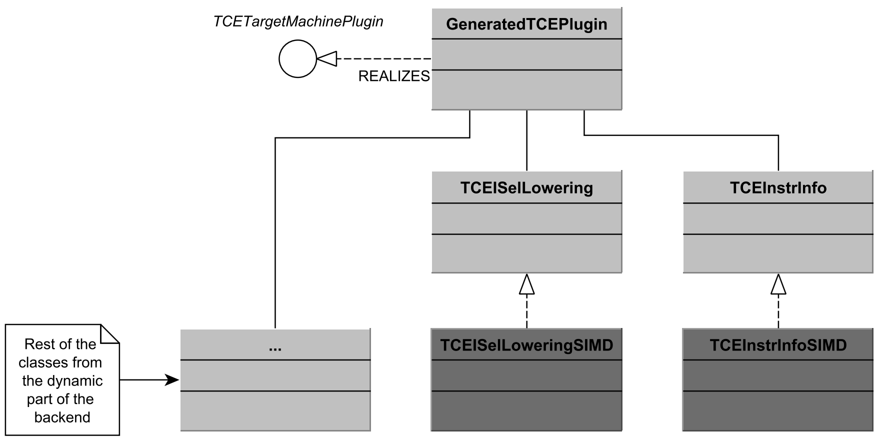


Figure 4.4: Two new classes in the plugin structure.

Table 4.1: Supported vector types for each subword type. Some modifications had to be made to LLVM so that vector widths could be enabled up to 1024 bits for all vector element types. The bolded vector types were extended to LLVM, whereas the rest were originally supported.

i1	i8	i16	i32	f16	f32
v2i1	v2i8	v2i16	v2i32	v2f16	v2f32
v4i1	v4i8	v4i16	v4i32	v4f16	v4f32
v8i1	v8i8	v8i16	v8i32	v8f16	v8f32
v16i1	v16i8	v16i16	v16i32	**v16f16**	v16f32
v32i1	v32i8	v32i16	**v32i32**	**v32f16**	**v32f32**
v64i1	v64i8	**v64i16**		**v64f16**	
v128i1	**v128i8**				

Table 4.1 shows the vector types that are supported in TCE due to this thesis work. These vector types can be arbitrarily in use in the designed TTA machines as vector operands for vector instructions. The boolean vector types (`vXi1`) are required, because vector compare operations produce tightly packed boolean result vectors. For example, the "greater than" comparison for `v4i32` input vectors results to a `v4i1` result vector.

When *TDGenSIMD* examines the target machine, all register files are first iterated through and registers are divided into groups by their widths. For example, the 512-bit register group consists of registers that are gathered from the 512-bit register files. Then, every operation in the target machine is also iterated through, and different vector operand types found from the vector operations are gathered to a vector type list. The vector type list contains all vector types that exist in the target machine, and they need to be supported by vector register classes.

The SIMD part of the register set descriptor file starts by declaring the wide registers that are not declared by *TDGen*. Then, the vector register classes are created for every vector type existing in the vector type list. By exploiting the width information, machine registers from the register groups are associated with those register classes that support a vector type of the same width. Two vector register class definitions are shown below.

```
def V16F32Regs : RegisterClass<"TCE", [v16f32], 512,
                    (add R512_0, R512_1, R1024_0, R1024_1)> ;
def V64I8Regs  : RegisterClass<"TCE", [v64i8], 512,
                    (add R512_0, R512_1, R1024_0, R1024_1)> ;
```

Vector register classes define the supported vector type, alignment, and the set of registers, in which the supported value type can be written to. Although the alignment value is written to be the full width of the vector type, currently all values that go to stack are forced to a 4-byte memory alignment by TCE. Both register classes list four registers that can be used to store vector values. One should note that `R1024_0` and `R1024_1` registers are wider than the supported register class vector type. TCE allows exploiting wider registers to store smaller values, which increases the number of fast-access temporary register locations for operands.

Instruction Set Info Generation

TCE has its own derived instruction format class *InstTCE* that is used to define all instructions. The instruction format class is defined in *TCEInstrFormats.td* and has the following structure:

```
class InstTCE<dag outOps, dag inOps, string asmstr,
              list<dag> pattern> : Instruction {

    let Namespace = "TCE";
    dag InOperandList = inOps;
    dag OutOperandList = outOps;
    let AsmString = asmstr;
    let Pattern = pattern;
}
```

When *TCEInst* is used to create an instruction description, output and input operands are the only required parameters, the other two are optional. The operation pattern is required, if the operation should be utilized automatically by the instruction selector in the instruction selection phase. For most of the OSAL vector operations, a pattern is generated for the instruction record. However, the pattern field can be left empty, if there is no need for an automatic instruction selection, or if the instruction pattern can not be described with LLVM-specific pattern fragments. Instructions that do not have the pattern field can be optionally selected in the *custom instruction selection* phase, in which complex selection algorithms can be defined manually by the backend programmer.

TDGen generates instruction records related to scalar operations, and *TDGen-SIMD* generates records related to vector instructions. The automatically generated target machine instruction info is written to *GenInstrInfo.td*, and the generation of the vector instructions is done in the following steps.

1. All function units of the target machine are iterated through, and for every vector operation an instruction record is written by using the *TCEInst* instruction class. A few notes about writing the instruction records for vector operations:

   - By default, an instruction record with register operands is created for every vector operation. If any of the operands is a scalar, an additional instruction record with immediate scalar operands is created. For instance, the input operand of the `LDW32X4` operation is a scalar integer address. It can be defined to be a register operand and a constant value.
   - If the vector operation is a bitwise operation (`NOT`, `AND`, `IOR` or `XOR`), several instruction records are written for the same operation with different vector operands. Bitwise operations do not contain subword-related data, which means the same operation can be used to implement the operation for all vector types of the same width. For example, for bitwise operation `AND512`, instruction records are created for vector operand types `v64i8`, `v32i16` and `v16i32`. The 32-bit scalar operation is used in executing bitwise operations for boolean vector operands of type `v2i1`, `v4i1`, `v8i1`, `v16i1` and `v32i1`.
   - Pattern generation is not done for `UNPACK`, `VSHUFFLE`, `VBCAST`, `GATHER` and `SCATTER` vector operations. As of this writing, `UNPACK`, `GATHER`, and `SCATTER` do not have explicit LLVM instruction, and thus, cannot have a pattern for the instruction selection. These operations can only be used by utilizing TCE intrinsics. The instruction selection for `VSHUFFLE` and `VBCAST` is done manually in the custom selector, which allows writing hand-made optimizations.
   - There are no load and store operations defined for the `vXi1` boolean vector operand types in OSAL operations. Scalar load and store operations are exploited to implement the memory accessing for these vector types. Boolean `v2i1`, `v4i1` and `v8i1` vector types access the memory by using the byte operations `LDQ` and `STQ`. The `LDH` and `STH` operation are used for `v16i1` vectors, and the `LDW` and `STW` operations for `v32i1` vectors. Any vector load and store instructions of the matching operand widths are exploited for `v64i1` and `v128i1`.

2. A register-to-register move is defined for every vector register class.

3. If the target machine has a vector store and truncate operations for the same vector type, a *truncstore* definition is created for that vector type, which is a combination of the two operations. First, an operand type is truncated

(upper bits cut off) to a smaller type. For example, the `v4i32` type could be truncated to the `v4i16` type. Right after the truncation, the new truncated value is stored to the memory.

4. Scalar-to-vector definitions are written by exploiting the `VBCAST` operation to output an input scalar as a corresponding output vector by copying the input scalar to all output vector elements. These definitions are needed, since the LLVM middle-end might end up using scalar-to-vector conversions in the IR code.

5. Bit conversions are written between every float vector type and the corresponding integer vector type. For every float-integer pair, bit conversion pattern is written to both directions. For example, a bit conversion could be written from the vector type `v16i32` to `v16f32` and the other way around.

An instruction record definition for a vector OSAL operation is presented below. Let us assume the target machine has the `ADD32X16` OSAL vector operation in one of the machine's function units. Let us also assume that the IR code contains the following vector instruction.

```
%res = add <16 x i32> %vecA, %vecB
```

The goal is to get the instruction selector to replace the IR instruction with the target machine-specific `ADD32X16` operation. The OSAL operation has two input and one output vector operands, all having `element-width` of 32, `element-count` of 16, and the signed integer `type`. This means that the input and output vector operands will be using the `V16I32Regs` register class. The pattern for the vector IR instruction can be described in the target descriptor file with the following pattern string:

```
(add V16I32Regs:$op1, V16I32Regs:$op2)
```

The two input operands and their types are described by presenting the register class. The pattern also needs to specify the output operand type, to which the result from the above pattern is set. The whole instruction selection pattern is the following:

```
(set V16I32Regs:$op3, (add V16I32Regs:$op1, V16I32Regs:$op2))
```

The pattern consists now of an inner pattern node, and of the outer pattern node. As all parts for the instruction record are known, the complete form of the record is the following:

```
def ADD32X16uuu : InstTCE<
  (outs V16I32Regs:$op3),
  (ins  V16I32Regs:$op1,V16I32Regs:$op2),
  "",
  [(set V16I32Regs:$op3, (add V16I32Regs:$op1, V16I32Regs:$op2))]>;
```

This record tells the instruction selector to select the `ADD32X16` OSAL operation to execute vector additions with `v16i32` operands. The same instruction record generation is done for all instructions, with the exception that the pattern field may be left empty for some operations. The `uuu` suffix in the record name states, that the operation has two input and one output operands, all of type “u”, which stands for an *integer vector* operand in *TDGenSIMD*. Since this operation doesn’t have any scalar operands, a record version with constant operands is not created. The instruction record name is added to a helper function in the *Backend.inc* file, which maps this instruction record to the target machine’s `ADD32X16` operation name.

Previously mentioned truncstore, scalar-to-vector and bit conversion patterns are defined by exploiting existing instruction records, not by making new ones. Below is an example of one of each patterns.

```
def : Pat<(truncstorev16i16 v16i32:$op2, ADDRrr:$op1),
          (STH16X16rt ADDRrr:$op1, (TRUNCWH16X16tu V16I32Regs:$op2))>;

def : Pat<(v16i32 (scalar_to_vector i32:$in)),
          (VBCAST32X16ur R32IRegs:$in)>;

def : Pat<(v16f32 (bitconvert (v16i32 V16I32Regs:$src))),
          (v16f32 V16F32Regs:$src)>;
```

The truncstore instruction is replaced with a pattern, which first executes a 32-bit subword to 16-bit subword truncation for the vector operand. As the memory address is specified by the scalar address operand, the truncated vector is then normally stored to memory location with the corresponding vector store operation.

The scalar-to-vector pattern is replaced with a simple `VBCAST` instruction, which copies the scalar to all vector elements. Another way to create the conversion could be by building the vector with `PACK` operation.

Bit conversions, or bit casts, change the type of the operand, but do not generate any actual instructions. Bit conversions may occur between types of the same bit size, but the conversion does not change any of the original bits. In the example, `v16i32` vector type is converted to the `v16f32` type by simply writing a pattern which places the source operand to the `V16F32Regs` register class.

Additional Backend Info

TDGen and *TDGenSIMD* generate various helper functions to the *Backend.inc* file. *TDGenSIMD* writes helper functions that are related to SIMD features. The helper functions are generated to *Backend.inc* in compile-time, because the contents of the functions are dynamic and depend on the target machine architecture.

One important helper function is related to the LLVM *TargetLowering* class, in which the registration of all register classes existing in the *GenRegisterInfo.td* file is required. *TDGenSIMD* generates a function implementation, which registers the vector register classes. The code segment below shows partially how the implementation is generated for the `addVectorRegisterClasses()` method, which belongs to the *TCETargetLoweringSIMD* subclass.

```
void TCETargetLoweringSIMD::addVectorRegisterClasses() {
    addRegisterClass(MVT::v128i8, &TCE::V128I8RegsRegClass);
    addRegisterClass(MVT::v16f32, &TCE::V16F32RegsRegClass);
    addRegisterClass(MVT::v16i32, &TCE::V16I32RegsRegClass);
    ...
}
```

A register class is added to the backend by calling the `addRegisterClass` function. An enumeration of the vector register class in the TCE namespace, and the vector type it supports are passed as parameters.

4.3 Hardware Generator

The SIMD function unit generator is an automatic tool for creating vector function unit implementations. A vector function unit is created by exploiting an existing scalar function unit: the scalar FU is duplicated as many times as there are subwords in the vector FU. Port widths are set according to the expected input and output vectors, and wiring to the duplicated scalar FUs is done automatically. Abstract structure of a generated vector function unit is shown in Figure 4.5.

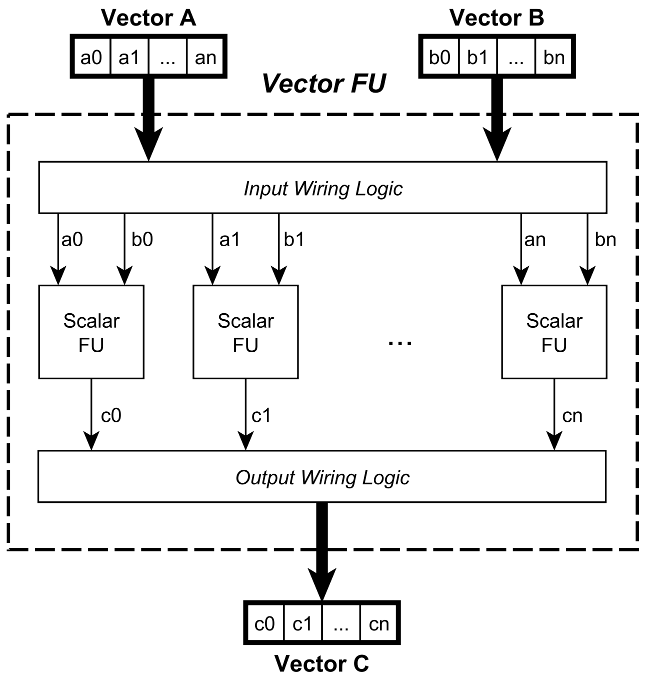


Figure 4.5: Structure of a vector function unit.

In the example, a scalar FU has two inputs and one output. This means the vector FU has two vector inputs and one vector output. Due to the abstraction, some of the inputs have been left out of the image, such as the *clk*, *rst*, *opcode* and input operand *load* signals. Vector elements are directed to individual scalar FUs, along with the *opcode* and *load* signals.

When there is change in the input operands or in the operation code, input vectors are broken to subwords in the input wiring logic and directed to the scalar FUs to calculate the result subwords according to the current operation code. The result subwords are then correctly wired back to a packed result vector. After the result vector has been written to the output, the output port remains in the same state unless there is a new change in the function unit inputs.

The varying result vector types in vector operations cause problems when input subwords are wired to scalar FUs and result subwords are wired back to a packed output vector. For instance, if the vector FU has the operations `ADD32X4` and `EQ32X4`, the result vector types of these two operations are `v4i32` and `v4i1`, respectively. Since the `v4i32` vector type matches the output, result scalars from scalar FUs can be packed normally back together. However, the `v4i1` result vector is a tightly packed boolean vector, and should overwrite only the four lowest bits of the output port. This requires additional logic to select only the lowest bits (boolean values) from every scalar FU result port, and then direct them to the output port.

```
output_control: process (r1data_s)
begin
  case pipelined_opcode_delay2_r is
    when "1" =>  -- EQ32X4
      r1data(0 downto 0)       <= r1data_s(0 downto 0);
      r1data(1 downto 1)       <= r1data_s(32 downto 32);
      r1data(2 downto 2)       <= r1data_s(64 downto 64);
      r1data(3 downto 3)       <= r1data_s(96 downto 96);
    when others =>  -- Direct connections
      r1data   <= r1data_s;
  end case;
end process output_control;
```

Figure 4.6: Output wiring control process.

In order to know how the wiring of the result subwords should be done, the input opcode is pipelined according to the latency of the scalar function unit. When the scalar results are calculated, the correct wiring is decided depending on the pipelined opcode, which tells what was the operation that was put to execution N cycles ago. The same convention is applied to the input wiring control logic, only there the wiring decision is done directly based on the input opcode port without the pipelining.

A VHDL example of the output wiring control is presented in Figure 4.6. The output control process is part of the vector function unit. *r1data_s* is a 128-bit logic vector and contains all four result subwords from the scalar FU outputs. *r1data* is the output port of the vector FU, to which the result vector will be written to. *pipelined_opcode_delay2_r* contains the opcode that was in the opcode input port two clock cycles ago. Opcode “1” is reserved to represent `EQ32X4` operation, and “0” for `ADD32X4`. Operations, whose result subwords do not need special wiring (like `ADD32X4`) are directed to **`when others`** case where the result signal is connected directly with the output port. Operations in need of special wiring have their own **`when`** `OPCODE` case, where the correct wiring is performed. For `EQ32X4`, the boolean result values are picked separately from the result subwords and directed to the output to form a tightly packed boolean result vector. There is also a similar process called `input_control` to do the input wiring, with the exception that the `case` argument is the non-pipelined opcode input port. The opcode is pipelined in a third `opcode_pipeline_control` process, which is sensitive to the *clk* signal.

As input, the generator requires the scalar FU (source HDB and entry ID) that will be converted to a vector FU, the desired subword count, and the target HDB file to which it is registered. When executed, the generator

1. Creates an HDB function unit entry for the new vector FU.
2. Registers the new entry to the target HDB file.
3. Generates a VHDL implementation for the vector FU. The implementation file is placed to “simd” directory, which is created to the same directory where the target HDB file is.
4. Tests correct functionality of the implementation with the HDB tester.

Usage

The generator executable is called *generatesimdfu* and is used as follows:

```
generatesimdfu <options> sourceHdb entryId elementCount targetHdb
```

`sourceHdb` and `targetHdb` are either absolute paths or relative paths to the HDB file. Accepted command line options are:

Short Name	**Long Name**	**Description**
d	`leave-dirty`	Do not delete created testbench files.
s	`simulator`	HDL simulator used to simulate and test the vector function unit. Accepted values are ’ghdl’ and ’modelsim’. Default is ghdl. Simulator executable must be found from PATH.
v	`verbose`	Enable verbose output. Prints information of the FU generation.

Example: create a vector FU of the “add” scalar FU (entry ID 4) from *asic_130nm_1.5V.hdb*. Wanted subword count is 16, and target HDB is *simd.hdb* under TCE’s HDB directory.

```
generatesimdfu /path/to/tce/hdb/asic_130nm_1.5V.hdb 4 16
  /path/to/tce/hdb/simd.hdb
```

This results to a new FU entry in *simd.hdb*, and a VHDL implementation file for the vector FU is created under /path/to/tce/hdb/simd/ directory with file name *fu_add_always_1_v16.vhdl*.

5. VERIFICATION

5.1 Behavior Models of Vector Operations

As presented in Figure 5.1, *generate_simd.py* is used to generate test inputs for each SIMD operation, which are then given as input for *testosal*. The *X_OPNAME.txt* file includes the SIMD operations that are executed in *testosal*. The *X_output.txt* file includes the expected results from the SIMD operations in the same order in which they are listed in *X_OPNAME.txt*. The behavior model tests for SIMD operations are located under *testsuite/systemtest/codesign/osal/SimdOperations/* in TCE environment. In all tests, the `!output hex` command is given in the beginning to ensure all printed result values are in hexadecimal format.

An example of the *18_ADD.txt* contents is shown below. The input file lists all `ADD` vector operations with test input vectors. Vector operands are given as hexadecimal strings, which define the values for vector subwords.

```
!output hex
ADD8X2 0x4454 0x435c
ADD8X4 0x44542414 0x435c0247
ADD8X8 0x44542414205c086c 0x435c02472f72594f
...
```

As the input file is given to *testosal*, it uses the behavior model to calculate result values. The expected result value contents from *18_output.txt* is shown below.

```
0x87b0
0x87b0265b
0x87b0265b4fce61bb
...
```

The result hex string values are compared to expected hex string result values, which *generate_simd.py* has generated before the input files are run in *testosal*. These simple test cases verify that correct bits are produced to result values from every OSAL SIMD operation.

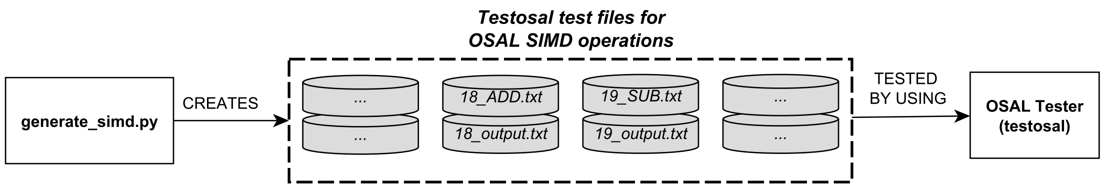


Figure 5.1: Test files used in OSAL Tester tests.

5.2 Compiler Backend Retargeting

This section presents briefly the retargeting tests for the compiler. In addition, the simulator is used to verify that the compiled code behaves correctly in the simulator environment. The compiler retargeting tests verify that *TDGenSIMD* generates correct register classes for vector operands and instruction selection patterns for vector machine operations. The tests are located under *testsuite/systemtest/bintools/Compiler/SimdOperations*, and they are executed in the following steps:

1. Customize a processor architecture with SIMD operations and wide registers.
2. Create a source program that explicitly performs vector computation that matches the SIMD operations in the machine.
3. Compile the source program for the machine without optimizations.
4. Examine the generated machine code in assembly form and check that instruction selector has selected the vector machine operations.
5. Run the target machine program in the TCE simulator and verify that result vector variables have correct values in the simulator memory.

Since *generate_simd.py* has common knowledge of all SIMD operations and vector types in TCE, it is used to generate the steps one and two automatically to minimize manual work. In step one, the script starts with generating the minimal architecture requirements, after which the SIMD resources are added. All SIMD OSAL operations are contained into functions units and, some wide registers are created for the vector operands. Figure 5.2 presents the compiler test files that are generated by *generate_simd.py*.

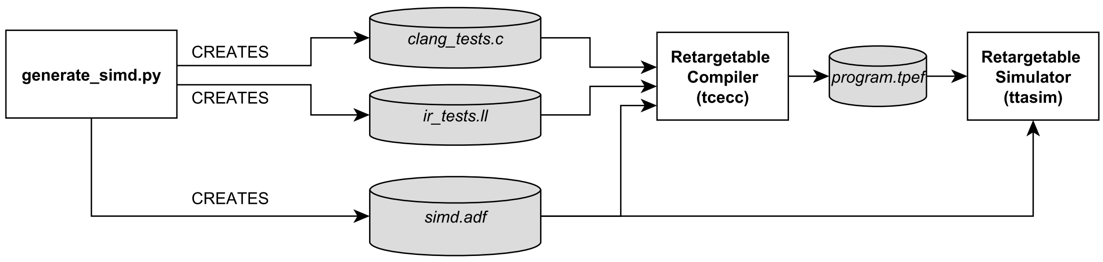
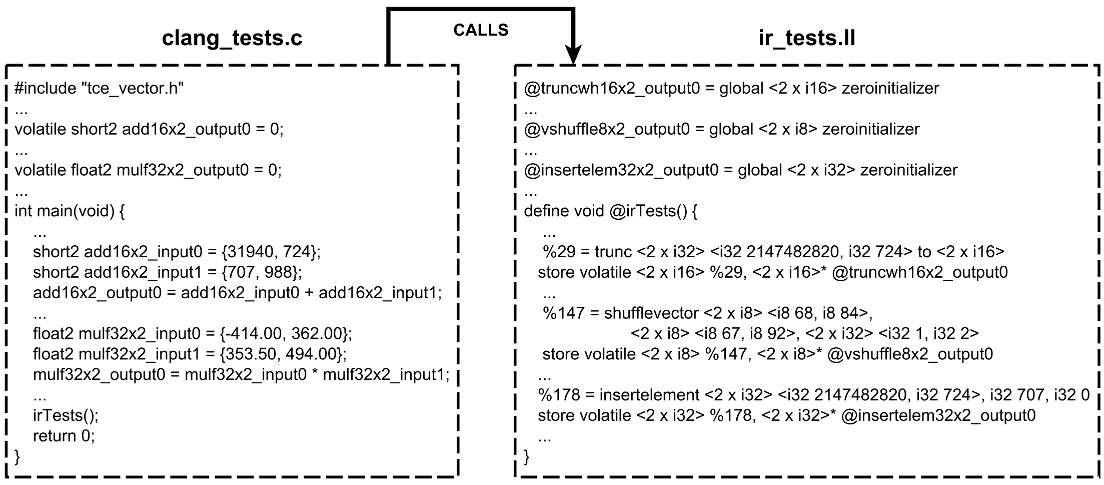


Figure 5.2: Test files used in compiler tests.

Two separate source code files are generated for compiler tests in step two. The *clang_tests.c* file contains most of the integer and single-precision floating point operation tests written in C. It has the `main()` function definition, and thus, acts as the master file. The *ir_tests.ll* file and its contents are written in LLVM IR, and it contains tests for operations with half-precision floating point operands. In addition, it contains tests for the rest of the vector operations that can not be explicitly expressed in C, such as the vector shuffle operation. The reason why the half-precision floating point tests are not in *clang_tests.c* is because as of this writing Clang C frontend has issues with the 16-bit floating point vectors in the IR generation. In *ir_tests.ll* those operations can be described directly as LLVM IR operations, releasing the operations from possible issues with the frontend functionality.

Figure 5.3 presents some of the contents from both test files. On the left, are the vector operation definitions in C language. On the right, are the vector operation definitions directly in LLVM IR. All result vector variables are defined as global variables in both files to get access to their memory locations in the TCE simulator. The memory location access is required so that result values of SIMD operations can be read from the simulator memory and compared to the expected result values.


CALLS
clang_tests.c
ir_tests.ll

```
#include "tce_vector.h"
...
volatile short2 add16x2_output0 = 0;
...
volatile float2 mulf32x2_output0 = 0;
...
int main(void) {
   ...
   short2 add16x2_input0 = {31940, 724};
   short2 add16x2_input1 = {707, 988};
   add16x2_output0 = add16x2_input0 + add16x2_input1;
   ...
   float2 mulf32x2_input0 = {-414.00, 362.00};
   float2 mulf32x2_input1 = {353.50, 494.00};
   mulf32x2_output0 = mulf32x2_input0 * mulf32x2_input1;
   ...
   irTests();
   return 0;
}
```

```
@truncwh16x2_output0 = global <2 x i16> zeroinitializer
...
@vshuffle8x2_output0 = global <2 x i8> zeroinitializer
...
@insertelem32x2_output0 = global <2 x i32> zeroinitializer
...
define void @irTests() {
   ...
   %29 = trunc <2 x i32> <i32 2147482820, i32 724> to <2 x i16>
  store volatile <2 x i16> %29, <2 x i16>* @truncwh16x2_output0
   ...
   %147 = shufflevector <2 x i8> <i8 68, i8 84>,
                  <2 x i8> <i8 67, i8 92>, <2 x i32> <i32 1, i32 2>
   store volatile <2 x i8> %147, <2 x i8>* @vshuffle8x2_output0
  ...
  %178 = insertelement <2 x i32> <i32 2147482820, i32 724>, i32 707, i32 0
  store volatile <2 x i32> %178, <2 x i32>* @insertelem32x2_output0
  ...
}
```

Figure 5.3: Test files used in compiler tests.

The C test file begins with including the *tce_vector.h* file to enable the vector types in the language. The result vector variable names are declared in the global area of the code and the variable values are initialized to zero. In the `int main(void)` function body, input vector operands are created just before the actual vector operation is executed. The operation results are directed to the global result vector variables. The `irTests()` function is called at the end of the file to execute the operations defined in *ir_tests.ll*.

The structure of the *ir_tests.ll* file is quite the same with *clang_tests.c*. The global result variables are declared first, after which the vector operations are defined in the `irTests()` function body. For the sake of an easier example, the input operands are given as constant vectors in the IR code. Some of the (non-visible) input operands are defined as temporary variables, which require loading the value from a memory location. The results are first assigned to temporary result variables, which are then stored to the result variable's memory location.

In order to make sure none of the SIMD operations are optimized out in step three, the compiler is instructed not to optimize the IR code. If an operation in the IR code was optimized out, the instruction selection and, therefore, retargeting for that operation would fail. The compilation process produces a `.tpef` binary file that can be run in the simulator for the processor architecture.

In step four, the binary file is first converted to a human readable assembly format with the *tcedisasm* tool to check that the instruction selector has selected the correct SIMD OSAL operations to the target machine code. The SIMD operations are verified from the assembly code by using the *grep* Unix tool to check that the operations exist in the code. If a SIMD operation can not be found from the file, the instruction pattern generation in *TDGenSIMD* is likely to be incorrect for that specific operation.

In step five, the `.tpef` binary program and the `.adf` architecture description file are used to run the program in the TCE simulator *ttasim*. After the program has been run, the vector operations have been executed and their results have been stored to the result variable locations in the simulator memory. The *generate_simd.py* creates a *ttasim.in* file that contains commands for printing all vector result values from the memory. This way they can be checked against the expected vector values, which are also generated by the script. A few lines from the *ttasim.in* contents are shown below.

```
...
x /n 128 /u b add8x128_output0
x /n 2 /u h add16x2_output0
x /n 4 /u h add16x4_output0
...
```

The bold line shows the result variable name that was used in the C code example in Figure 5.3. The line prints the memory contents starting from the address `add16x2_output0` in chunks of 2 bytes (subword width for `i16` vector element) two times. The printed hex string is `0x7f87 0x06b0`, which is compared to the expected hex string. The same comparison is done for all result variables. The expected hex strings are generated to a single text file under *simd_operations* directory in the same order as *ttasim.in* file makes the simulator to print the vector variables from the memory.

5.3 Vector Function Unit Generator

The vector function unit generator tests are located in *testsuite/systemtest/bintools/SIMDFUGenerator*. The test script *run_simd_fu_generator_test.sh* uses the *generatesimdfu* tool to generate several vector FUs from scalar FUs. Below is a code snippet how the Bash script generates vector ALUs with different element count sizes in a **for** loop. The vector ALUs are created by exploiting a scalar ALU function unit from *asic_130nm_1.5V.hdb*.

```
BINARY=../../../../tce/src/bintools/SIMDFUGenerator/generatesimdfu
SCALAR_HDB_FILE=../../../../tce/hdb/asic_130nm_1.5V.hdb
SIMD_HDB=simd.hdb

for ELEM_COUNT in 2 4 8 16 32
do
  ...
  ENTRY_ID=376 # Minimal requirements ALU with delay 1.
  ./generatesimdfu $SCALAR_HDB_FILE $ENTRY_ID $ELEM_COUNT $SIMD_HDB
  ...
done
```

The scalar ALU contains the minimal operation requirements for any TTA processor. Vector versions are generated with element count of 2, 4, 8, 16 and 32 and stored to a temporary test database file (*simd.hdb*). The *generatesimdfu* tool uses *HDBTester* to create an RTL testbench for all generated SIMD function units and runs it automatically with *ghdl*.

6. CONCLUSIONS

In the master's project described in this thesis, TCE toolset was extended with SIMD capabilities. The implemented SIMD support covers the whole TCE design flow, starting from the high-level vector programming in the source code all the way down to the hardware description generation of vector function units. Most of the work was done on top of existing tools and data structures in the toolset.

This thesis gave an overview of instruction-level and data-level parallelism and customized processor templates. In addition, a short introduction to the TCE toolset was given, followed by a description of compiler concepts. The TCE toolset, the design flow and the most relevant data structures were presented. Parts that required modification for the SIMD extension were pointed out.

Several tools had to be modified to make them recognize and support the SIMD extension. The most important part was to preserve the retargetability of the TCE compiler, as the new SIMD features were introduced to the environment. The compiler backend had to automatically recognize the SIMD resources from customized processor architectures and use them in the program execution. New tools that were created as part of the toolset were made as automatic as possible to minimize manual effort and keep the design flow effective.

The SIMD extension was tested and verified in various points of the design flow. The OSAL tests successfully verified the behavior model for all SIMD OSAL operations. The compiler tests verified the retargetability of the compiler for the new vector operations, and the correct layout of the vectors in the simulator memory. Finally, the automatic generation of SIMD function unit RTL descriptions was verified with the HDB tester.

The goals of this thesis were reached, as processor designers using TCE are now able to take DLP requirements into account in target applications. In case even wider OSAL vector operations should be introduced to TCE in future, the fully automatic SIMD generation script should minimize most of the manual work.

There is still room for improvements and future development. Currently, vectors of any width are forced to a 4-byte address location when they are stored to the memory. The backend should be modified to align vectors to the memory by using the full vector width. For example, a 512-bit vector should be stored to a memory address that is divisible by 64. This would reduce the complexity of the load and store vector operations on hardware.

Vector types can not be currently defined as function parameters or return values. In addition, larger vector machine operations could be exploited to execute smaller vector operations if there are no matching machine operations for them. For example, if the machine supports additions between `v8i32` types, and the IR code contains non-supported `v4i32` vector additions, the wider machine operation could be exploited to execute the smaller operation.

In case the IR code contains multiple sequential “insert element” operations to build a vector, and a correct-sized `PACK` operation exists in the machine, the vector building might be more cost-efficient by exploiting the packing operation. The same scheme applies to multiple sequential “extract element” operations with a matching machine `UNPACK` operation. Truncation for vector elements could also be made by unpacking a vector to independent elements, after which they could be packed back to a vector with a packing operation that inputs independent elements with smaller element width.

REFERENCES

[1] Espasa, R. and Valero, M., “Exploiting Instruction- and Data-Level Parallelism,” *IEEE Micro*, vol. 17, no. 5, pp. 20–27, 1997.

[2] Murakami, K. and Irie, N. and Tomita, S., “SIMP (Single Instruction Stream/-Multiple Instruction Pipelining): A Novel High-speed Single-processor Architecture,” in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA ’89, pp. 78–85, 1989.

[3] González, J. and González, A., “The Potential of Data Value Speculation to Boost ILP,” in *Proceedings of the 12th International Conference on Supercomputing*, ICS ’98, pp. 21–28, 1998.

[4] Bernstein, D. and Rodeh, M., “Global Instruction Scheduling for Superscalar Machines,” *SIGPLAN Not.*, vol. 26, pp. 241–255, May 1991.

[5] Smith, M. and Lam, M. and Horowitz, M., “Boosting Beyond Static Scheduling in a Superscalar Processor,” *SIGARCH Comput. Archit. News*, vol. 18, pp. 344–354, May 1990.

[6] Russell, R., “The CRAY-1 Computer System,” *Commun. ACM*, vol. 21, pp. 63–72, Jan. 1978.

[7] Peleg, A. and Wilkie, S. and Weiser, U., “Intel MMX for Multimedia PCs,” *Commun. ACM*, vol. 40, pp. 24–38, Jan. 1997.

[8] Intel Corporation, *Intel® Architecture Instruction Set Extensions Programming Reference*, December 2013.

[9] Pai, S. and Govindarajan, R. and Thazhuthaveetil, M., “Limits of Data-Level Parallelism,” in *14th Annual IEEE International Conference on High Performance Computing*, 2007.

[10] Lorenz, M. and Marwedel, P. and Dräger, T. and Fettweis, G. and Leupers, R., “Compiler Based Exploration of DSP Energy Savings by SIMD Operations,” in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC ’04, pp. 838–841, 2004.

[11] Langdon, W. and Banzhaf, W., “A SIMD Interpreter for Genetic Programming on GPU Graphics Cards,” in *Proceedings of the 11th European Conference on Genetic Programming*, EuroGP’08, pp. 73–85, 2008.

[12] Balfour, J., “CUDA Threads and Atomics,” 2011. http://mc.stanford.edu/cgi-bin/images/3/34/Darve_cme343_cuda_3.pdf, referenced: 13.05.2014.

[13] Yu, P. and Mitra, T., “Scalable Custom Instructions Identification for Instruction-set Extensible Processors,” in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES ’04, pp. 69–78, 2004.

[14] Bonzini, P. and Harmanci, D. and Pozzi, L., “A Study of Energy Saving in Customizable Processors,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, vol. 4599, pp. 304–312, 2007.

[15] Kumar, R. and Tullsen, D. and Ranganathan, P. and Jouppi, N. and Farkas, K., “Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance,” *SIGARCH Comput. Archit. News*, vol. 32, pp. 64–, Mar. 2004.

[16] Qualcomm, “Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age.” White Paper, 2011.

[17] Nieuwland, A. and Kang, J. and Gangwal, O. and Sethuraman, R. and Busá, N. and Goossens, K. and Peset Llopis, R. and Lippens, P., “C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems,” *Design Automation for Embedded Systems*, vol. 7, no. 3, pp. 233–270, 2002.

[18] Pozzi, L. and Paulin, P.G., “A Future of Customizable Processors: Are We There Yet?,” in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE ’07*, pp. 1–2, April 2007.

[19] J. Hoogerbrugge and H. Corporaal, “Register File Port Requirements of Transport Triggered Architectures,” in *Proceedings of Annual International Symposium on Microarchitecture*, pp. 191–195, November-December 1994.

[20] Hoogerbrugge, J. and Corporaal, H., “Comparing Software Pipelining for an Operation-Triggered and a Transport-Triggered Architecture,” in *In Lecture Notes in Computer Science 641, Compiler Construction*, pp. 219–228, 1992.

[21] “TCE: TTA-Based Codesign Environment.” Web page: http://tce.cs.tut.fi, referenced: 15.04.2014.

[22] Muchnick, S., *Advanced Compiler Design and Implementation.* 1997.

[23] Aho, A. and Lam, M. and Sethi, R. and Ullman, J., *Compilers: Principles, Techniques, and Tools (2Nd Edition).* 2006.

[24] Wilhelm, R. and Seidl, H. and Hack, S., *Compiler Design: Syntactic and Semantic Analysis.* 2013.

[25] Kats, L., “Supporting Language Extension and Separate Compilation by Mixing Java and Bytecode,” Master’s thesis, Utrecht University, The Netherlands, August 2007.

[26] Leupers, R., “Compiler Design Issues for Embedded Processors,” *Design Test of Computers, IEEE*, vol. 19, pp. 51–58, Jul 2002.

[27] A. Metsähalme, “Instruction Scheduler Framework for Transport Triggered Architectures,” Master’s thesis, Tampere University of Technology, Finland, Apr 2008.

[28] Eichenberger, A. and Wu, P. and O’Brien, K., “Vectorization for SIMD Architectures with Alignment Constraints,” in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI ’04, pp. 82–93, 2004.

[29] Sreraman, N. and Govindarajan, R., “A Vectorizing Compiler for Multimedia Extensions,” *Int. J. Parallel Program.*, vol. 28, pp. 363–400, Aug. 2000.

[30] Maleki, S. and Gao, Y. and Garzaran, M.J. and Wong, T. and Padua, D.A., “An Evaluation of Vectorizing Compilers,” in *Parallel Architectures and Compilation Techniques (PACT)*, pp. 372–382, Oct 2011.

[31] J. Mäntyneva, “Automated Design Space Exploration of Transport Triggered Architectures,” Master’s thesis, Tampere University of Technology, Finland, July 2009.

[32] V. Jääskeläinen, “Retargetable Compiler Backend for Transport Triggered Architectures,” Master’s thesis, Tampere University of Technology, Finland, Feb 2010.

[33] Tampere University of Technology, “TTA Codesign Environment v2.0 User Manual,” 2008.

[34] The LLVM Team, “The LLVM Compiler Infrastructure Project.” http://llvm.org, referenced: 15.04.2014.

[35] Jääskeläinen, P., “Instruction Set Simulator for Transport Triggered Architectures,” Master’s thesis, Tampere University of Technology, Finland, Sep 2005.

[36] L. Laasonen, “Program Image and Processor Generator for Transport Triggered Architectures,” Master’s thesis, Tampere Univ. Tech., Finland, 2007.

[37] O. Esko, “ASIP Integration and Verification Flow,” Master’s thesis, Tampere University of Technology, Finland, June 2011.