HW/SW Co-Design Toolset for Customization of Exposed Datapath Processors

Pekka Jääskeläinen, Timo Viitanen, Jarmo Takala, Heikki Berg

Abstract Customized processors are an interesting option for implementing software defined radios; they bring benefits of tailored fixed function hardware while adding new advantages such as reduced implementation verification effort and increased post-fabrication flexibility. To reduce the engineering costs and the time-to-market of platforms with new computing devices, the processor customization process should be supported with automated design flows that include tools such as automatically retargeting compilers, instruction-set simulators, and RTL generators. This chapter presents an open source processor co-design toolset that is based on a computation resource oriented design methodology where the primary design choices are the set of resources to include in the processor at hand, instead of focusing on instruction encoding details. The toolset is based on a retargetable high-level language compiler and a scalable exposed datapath template which support different styles of parallelism available in applications. In addition to various published academic processor design examples for SDR algorithms, the tools have been used to design and program processors that have been implemented down to silicon layout level and integrated in commercial grade chips.

1 Introduction

It is nowadays common to integrate multiple different computing devices in a single chip, each device serving a different application domain or accelerating a specific part of an application. While specialized fixed function hardware accelerators have been shown to bring performance and efficiency gains, designing and implementing new designs has remained a high-cost exercise. In addition, programmable cores are

Pekka Jääskeläinen · Timo Viitanen · Jarmo Takala
Tampere University of Technology, Tampere, Finland, e-mail: firstname.lastname@tut.fi

Heikki Berg
Nokia Technologies, Tampere, Finland, e-mail: heikki.berg@nokia.com

often favoured to fixed-function accelerators for their post-manufacture flexibility (e.g., on-the-field bug fixes or algorithm updates) and their ability to reuse compute hardware in case multiple algorithms can be efficiently mapped to the same accelerator.

Customized processors provide a middle ground between fixed function accelerators and generic programmable cores. They bring benefits of hardware tailoring to programmable designs, while adding new advantages such as reduced implementation verification effort. The hardware of customized processor is optimized for executing a predefined set of applications, while allowing the very same design being used to run other, close enough routines by switching the executed software in the instruction memory.

The processor hardware tailoring is dictated by the use case. In case a processor is customized to execute a single application efficiently, but still support software-based functionality updates, terms *Application-Specific Instruction-set Processor (ASIP)* or *Application-Specific Processor (ASP)* can be used to describe the end result. A customized processor can also be optimized for classes of applications such as *Software Defined Radio (SDR)*, in which case the term *Domain-Specific Processor (DSP)* is more suitable.

In any case, the processor customization process is demanding due to the abundance of different parameters in the design space to choose from, and very error-prone, which results in high non-recurring engineering costs. Moreover, as the design process of customized processors is usually iterative in nature, porting the required software program codes to new processor variations needs either manual assembly language program rewrites or updating the compiler so it can generate assembly code for each new processor variant. One approach to simplifying the processor customization process is to compose the processor from a set of component libraries and other verified building blocks, thereby reducing the required verification effort. The software porting problem can be alleviated by automatically adapting software development kits.

In the rest of this chapter, we describe *TTA-Based Co-Design Environment (TCE)*, an open source processor design and programming toolset based on a processor template that efficiently supports *Instruction-Level Parallelism (ILP)*. TCE enables rapid design of high-level language programmable cores ranging from tiny scalar microcontrollers to wide VLIW-style machines with a resource oriented design methodology that emphasizes the reuse of predesigned and preverified components.

2 Exposed Datapath Processor Template

To implement a design and programming toolset for customized processors, the *design space* of the supported customized processor alternatives needs to be limited. This is done by defining a *processor template* that describes the set of customization parameters within which the processors can vary. The parameters drive the retargeting of the software development toolchain, most importantly the compiler and the simulator. Here we have selected exposed datapath as one of the characteristics of the architectural template. The exposed datapath means that some additional processor datapath details are visible to the programmer or compiler such that the programmer can directly control the additional details. Such architectures have been discussed earlier, e.g., MOVE [4], MOVE-Pro [6], FlexCore [13], STA [6], and ELM [5].

The main focus of the TCE toolset is on energy-efficient data-oriented computing scenarios. Therefore, compiler-controlled static structures are favored over hardware-controlled dynamic structures whenever feasible. This can be seen in the choice of the basis for the processor template of TCE which exploits *transport triggering* paradigm in form of *Transport Triggered Architectures (TTA)* [11, 2].

TTA processors have a programmer-exposed *interconnection (IC)* network. A program is defined as a set of *move instructions* between the ports of the datapath components, including function units and register files. The operations are triggered as a side effect of transporting data to designated function unit ports. Fig. 1 presents an example TTA processor with five transport buses, i.e., five data transports can be executed simultaneously. In this processor, each instruction contains five *move slots*, where each slot specifies the data transport carried out in each bus. The figure illustrates execution of an instruction with three parallel moves:

#4 $\rightarrow$ ALU0.i0.ADD;
LSU1.r0 $\rightarrow$ ALU0.i1;
RF0.r1 $\rightarrow$ LSU0.i0.STW

On the first transport bus, an immediate value is moved to input port 0 of the function unit ALU0. The immediate value is actually obtained from the *immediate unit*, which has only one output port. The move carries also information about the operation to be executed; opcode ADD is transported to function unit along with the operand. The second bus transports an operand from load-store unit LSU1 to the input port 1 of the ALU0. The actual load from memory has been specified by one of the previous moves and this move simply transports the result of the memory access. The third bus is used to transport a value from output port 1 in register file RF0 to the input port 0 of the load-store unit LSU0. The third move contains opcode indicating that the transported word is to be stored to memory. The actual store address has been defined by another move to port 1 of the LSU0. The remaining two buses are not used in the example instruction, thus they can be considered executing a NOP.

Figure 1 shows that the instructions control the operation of each transport bus through the instruction unit. The connection to each bus convoys control information, e.g., the source and destination of the transport move, possible opcode for the operation to be executed, etc. The function units are connected to the transport buses with the aid of *sockets*. The interconnection network in our architectural template consists of buses and sockets. The concept of socket is illustrated in Fig. 2; each port of a function unit has a socket, which defines the connections to the buses. When the

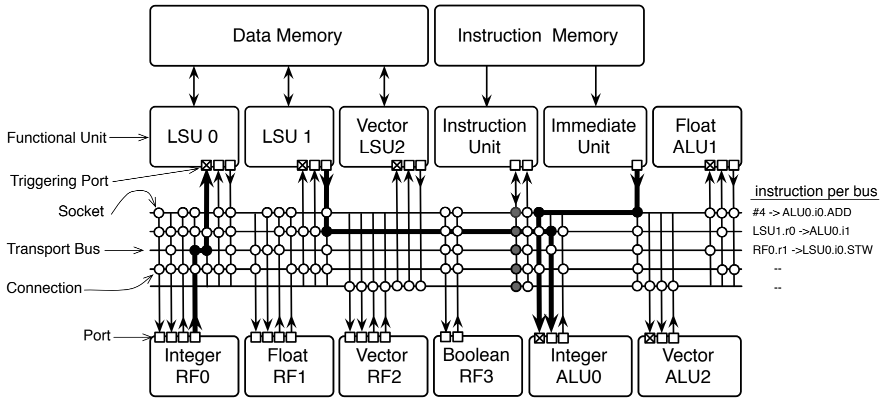

Fig. 1: Example of a TTA processor instantiated from the template. In TTA, data transports between components are explicitly programmed. The example instruction defines *move instructions* for three buses out of five, performing an integer summation of a value loaded from a data memory with a constant while simultaneously storing a previously computed value to memory.

control information in a bus indicates that the port is the destination for the current move instruction, data from the bus is passed to the port. In similar fashion, data from the source port is forwarded to the bus.

In TTAs, operation execution is a side result of operand transport; when operands are available in the function unit, the operation is triggered. In this sense, the execution model reminds static data flow machines. The architecture template used here defines that that one of the input ports is a *trigger port* and a move to this port triggers the operation. Fig. 3 illustrates the concept; the trigger port is indicated by a cross in the input port. A move to this port will latch data from the bus to trigger register and the operation execution starts with operands from trigger port and other operand registers; the function unit in Fig. 3 expects two operands, thus there is one trigger register and one operand register. The operand to the operand register can be moved by an earlier instruction. The operand can also be moved in the same instruction as the trigger port move if there are enough buses available.

Function units can be pipelined independently, and there are several methods to pipeline the function units in TTAs. In TCE, we use semi-virtual time latching [3] where the pipeline is controlled with valid bits depicted in Fig. 3(b). The pipeline starts an operation whenever there is a move to the trigger port, i.e., the *o_load* signal is active. The pipeline operation is controlled by valid bits, which imply that a single pipeline stage is active only once for one trigger move. The result can be read from the result register three instructions after the trigger move. However, the pipeline operates only on the cycles when instructions are issued; if an external or internal event has caused processor to be locked (*glbl_lock* signal is active), the pipeline is inactive. The architectural template requires each operation in function

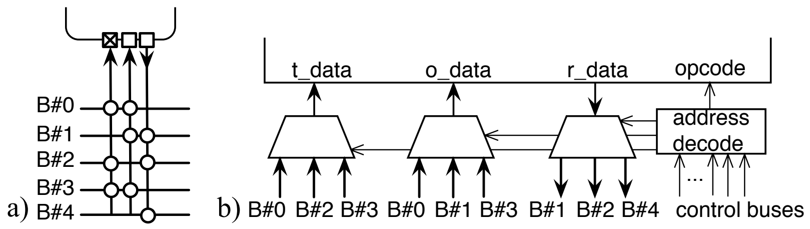

Fig. 2: Principal socket interface for function units: a) high abstraction level representation and b) structure. The result register in the output is optional.

units to have a deterministic latency such that the result read for the operation can be scheduled properly. If the function unit faces an unexpected longer latency operation, e.g., a memory refresh cycle or a function unit has iterative operation of which latency depends on the inputs, the unit can request the processor to be interlocked (by activating the *rqst_lock* signal) until the on-going operation is completed.

SDR applications can often utilize a lot of computational parallelism in its various granularities, but at the same time their usage environment places limits to the power consumption due to thermal design and battery constraints. Therefore, a popular processor architecture choice for SDR applications is a static multi-issue architecture with a simple control hardware. The traditional VLIW architecture fulfills these requirements by providing multiple parallel function units to support ILP while not imposing the hardware complexity from out of order execution support. The VLIW function units in SDR designs can also provide SIMD operations to exploit data level parallelism in the algorithm at hand, which together with ILP can provide very high high operation per watt performance in SDR designs.

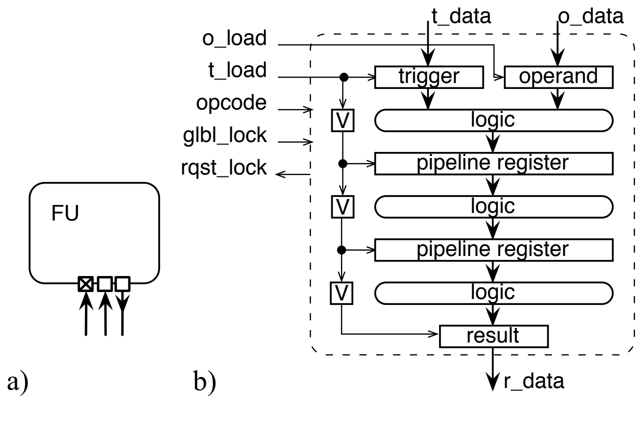

Fig. 3: Function units in transport triggered architecture: a) high-level abstraction representation and b) principal block diagram.

The use of the TTA paradigm to provide a more scalable alternative for traditional VLIW architectures has been studied extensively in [3]. One of the main findings is that TTAs can support more ILP with simpler low-power register files than "traditional" VLIWs because of the capability to route results directly between function unit port registers, without accessing the larger general purpose register files. In addition, the general ability to control the timing of the operand and result data transports alleviates the parallel access requirements of the register files. The simpler register files can remove the register file from the critical path and bring savings in required chip area and power consumption.

The TTA instruction format resembles those of horizontal microcode programmed architectures which are notorious for their poor instruction density. However, our example designs and experiments indicate that the additional instruction bits due to the exposed datapath control are negligible compared to the savings if the workload is data-oriented and the interconnection network is carefully optimized [14, 7].

The exposed datapath template also opens unique non-apparent opportunities. For example, due to the explicit result transfers, the function units are independently executing isolated modular components in the datapath. In the point of view of processor design methodology, the modularity allows point-and-click style tailoring of the datapath resources from existing processor component databases. It also means the function units can have latencies and pipeline lengths from a single cycle to no practical upper bound because the hazard detection hardware does not need to account for the structural hazards resulting from concurrent completion of operation results. Likewise, there is no practical limit to the number of outputs produced by operations. For example, there have been experiments where long latency fixed function accelerators with tens of result words have been integrated to the processor datapath in order to reduce accelerator communication and synchronization costs.

Using the TTA as a template in TCE means that it allows designing completely new TTA processors from the scratch by allowing the user to define the sets of basic TTA components to include. The customizable parameters are summarized in Table 1. An interesting customizable aspect in TTA processors is the interconnection network. As it is visible to the programmer, it enables carefully customizing the connectivity based on the application's need as will be discussed later in this chapter. This can help in achieving implementation goals such as high clock frequency or low power consumpion. Another useful feature is the support for multiple disjoint address spaces: one can add one or more private address spaces for local memories inside a core that can be accessed using address space type qualifier attributes in the input C code.

3 Processor Description Formats

TCE uses several file formats and component libraries for setting the processor template parameters for new designs. The roles of the different files and libraries are designed to enhance processor design and verification effort reuse. These XML-

Table 1: Configuration parameters in TCE's processor template.

Register files ● number of register files For each register file: ● number of registers ● width ● number of read/write ports
Function units ● number of function units For each function unit: ● operation set implemented by FU ● number of input and output ports ● width of the ports ● resource sharing / pipelining ● accessed address space (in case of a load-store unit)
Operation set For each operation: ● number of operands ● number of results ● data type of the results and operands ● operation state data
Instruction encoding ● number of instruction formats For each instruction format: ● immediate (constant) support
Address spaces ● number of address spaces For each address space: ● size ● address range ● the numerical id (referred to from program code)
Top level ● endianness

based file formats are edited using graphical user interfaces without the toolset user seeing the file contents, and if preferred, can be edited also using a text editor.

3.1 Architecture Description

An *Architecture Description File (ADF)* [1] stores the programmer-visible architectural aspects of a designed processor, such as the number and type of register files and function units, operation latencies and resource sharing of the operations inside function units, and the connections between the processor components.

The function unit architectures described in an ADF file can refer to one or more *operation* descriptions. An operation in TCE is a behavioral level abstraction separated from the function unit. Operation descriptions are stored in *Operation Set Abstraction Layer (OSAL)* databases. The granularity of an OSAL operation can range from basic operations to complex multioutput operations with internal state.

Operation descriptions in OSAL contain C++ or DAG-based models to simulate the behavior of the operation in the processor simulator, and static metadata to drive the high-level language compiler, such as the number of operands and results, the semantics of the operations, whether the operation reads or modifies memory, or has other effects to the program state. The automatically retargeting compiler needs these properties together with the architecture data from the ADF.

OSAL and ADF enable processor design space exploration at the architectural level. This allows fast exploration cycles with enough of modeling accuracy to drive the selection of the components.

The separation of the concepts of a function unit (with programmer visible latencies) and the behavioral operation descriptions is a key feature which enables trial-and-error cycles with the operation set design without needing to describe the operation candidates as detailed *Register Transfer Level (RTL)* descriptions.

3.2 Architecture Implementation

ADF and OSAL are not sufficient alone to implement the processor in hardware. For example, each programmer-identical function unit or register file can be implemented in multiple ways, emphasizing different goals such as small area or high clock frequency.

Hardware implementation choices are defined using an *Implementation Definition File (IDF)*. IDFs connect the architecture components described in ADF files to component implementations stored in *Hardware Databases (HDB)*. HDBs contain RTL level VHDL or Verilog implementations of register files and function units, along with their metadata. The component implementations can be easily reused in future architectures, reducing the validation effort of new customized processors.

4 HW/SW Co-Design Flow

Processor customization in TCE is usually conducted as a hardware-software co-design process. The processor design is iterated by varying the processor template parameters defined using the TCE file formats while adapting the software to better exploit the features, such as by calling custom operation intrinsics or enhancing parallelization opportunities from the program side.

The co-design process is supported by a set of tools, which are illustrated in Fig. 4. Initially, the designer has a set of requirements and goals placed to the end result. It is usual to have a real time requirement as the primary requirement, given as a maximum run time for the programs of interest. A secondary goal can be to optimize the processor for low power consumption or minimal chip area. In some cases, there can be a strict area and/or power budget which cannot be exceeded, and the goal is to design a processor that is as fast as possible and still goes below the budget.

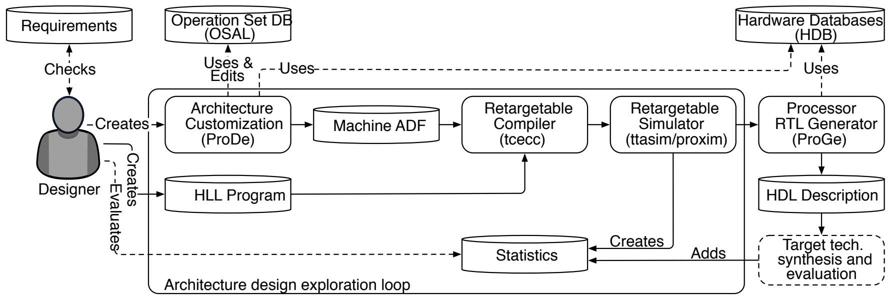
Requirements
Operation Set DB (OSAL)
Hardware Databases (HDB)
Checks
Uses & Edits
Uses
Uses
Designer
Creates
Creates
Evaluates
Architecture Customization (ProDe)
Machine ADF
Retargetable Compiler (tcecc)
Retargetable Simulator (ttasim/proxim)
Processor RTL Generator (ProGe)
HLL Program
HDL Description
Statistics
Creates
Adds
Target tech. synthesis and evaluation
Architecture design exploration loop

Fig. 4: TCE processor customization flow.

The iterative customization process starts with an initial predesigned architecture, which can be a minimal machine shipped with TCE that contains just enough resources for the compiler to provide C programming language support for the design, or a previous architecture that is close to the placed targets, but needs further customization to fully meet them.

4.1 Architecture Design Space Exploration

The designer can add, modify, and remove architecture components using a graphical user interface tool called *Processor Designer (ProDe)*. Each iteration of the processor can be evaluated by compiling the software of interest to the architecture using TCE's retargetable high-level language compiler and simulating the resulting parallel assembly code using an architecture simulator.

The simulator statistics give the runtime of the program and the utilization of the different datapath components, indicating bottlenecks in the design. The processor simulator provides a compiled simulation engine for fast evaluation cycles, and a more accurate interpretive engine for software debugging which supports common software debugging features such as breakpoints.

The accuracy of the TCE processor simulator is at the instruction-set level, but gives enough information to drive the architecture design process due to the static TTA-based processor template. Because the instruction-set controls so fine details of the processor as data transfers, the simulation is closer to a more detailed RTL simulation, especially regarding the metrics of interest.

The architecture exploration cycle enables low effort evaluation of different custom operations. For a completely new processor operation, the designer describes the operation simulation behavior in C/C++ to OSAL, estimates its latency in instruction cycles when implemented in hardware, and adds the operation to one of the function units in the architecture. This way it is possible to see the effects of the custom hardware to the cycle count, before deciding whether to include it in the design or not. As the OSAL behavior description is C/C++, usually the designer can just copy-paste the piece of program code that should be replaced by the custom operation call to get it modeled.

4.2 Processor Hardware Generation

When a design point fulfilling the requirements has been found, or more accurate statistics of a design point is needed, the designer uses a tool called *Processor Generator (ProGe)*, which produces a synthesizable RTL implementation of the processor. Thanks to the modular TTA template, the RTL generation is straightforward and reliable; ProGe collects component implementations from a set of HDBs, produces the interconnection network connecting them, and the generates an instruction decoder that expands the instructions to datapath control signals.

The designer has to implement only the new function units in VHDL or Verilog and add them to an HDB. Usually most of the more generic components in the new processor designs are found in readily in previously accumulated HDBs, and only the special function units that implement application-specific custom operations need to be manually implemented. To assist in the implementation process, a tool is available to automatically validate the function unit implementation against its architecture simulation model.

The generated RTL is fed to a standard third party synthesis and simulation flow. This step generates more detailed statistics of the processor at hand, which can again drive further iterations in the architectural exploration process. The detailed statistics include timing; to verify the clock frequency target is met, chip area; to ensure the design is not too large, and for low power designs, the power consumption when executing the applications of interest.

The detailed implementation level statistics map trivially back to the design actions at the architecture level. For example, the chip area can be reduced by removing architecture components. Similarly, adding more pipeline stages to complex function units, or reducing the connectivity of the interconnection network helps increasing the maximum clock frequency.

5 Automated Architecture Design

We think that the best processor designs should involve some human designer effort instead of attempting to completely automatize the process. After all, when describing a new architecture, what is actually being designed is a programmer-hardware interface, a user interface of a kind. This applies especially to designs that will be implemented with expensive ASIC processes, or are expected to be at least partially assembly-programmed. Such designs are better to be iterated at least partially manually in order to produce designs that are more understandable and logical for programmers. We have found *semiautomated* design space exploration a good compromise in this matter; instead of aiming towards a fully automated processor design process, the designer uses assisting tools for iterating different *aspects* of the design at hand automatically.

One aspect of TTA designs that calls for automated architecture design support is the interconnection network; connectivity between components in larger TTA designs is hard to manage manually due to the huge space of options. Therefore, usually in the earlier phases of TCE design process the connectivity aspect is simply ignored and a fully connected interconnection matrix is used in order to evaluate the compute resource sufficiency. Only after the application of interest is saturated with enough register file and function unit resources, the connectivity is pruned.

A redundant fully connected interconnection network can spoil TTA designs that are usually otherwise very streamlined and energy efficient. In addition to increasing the critical path, worsening the power consumption and expanding the required chip area, the excessive connectivity results in bloated instruction words due to the large number of sources and destinations to encode. Loose instruction words reflect in larger instruction memory requirements and energy waste in instruction fetch and decode.

TCE provides multiple automated *design space exploration tools* for iterating the interconnection network. One of them, called *Bus Merger* [14], starts by creating a connectivity matrix similar to VLIW designs with full FU bypass network connectivity (see Fig. 5). This network is then gradually reduced by merging buses (producing a bus with a union of the connections) that are rarely used at the same time for data transports, on each iteration evaluating the program to the new architecture using the compiler and the simulator. Finally, the original overly complex VLIW-like register files are simplified, thanks to the reduced need for RF ports.

The greedy Bus Merger finds rather good application-specific ICs quickly for VLIW-style designs, but might not be efficient for smaller architectures with a small

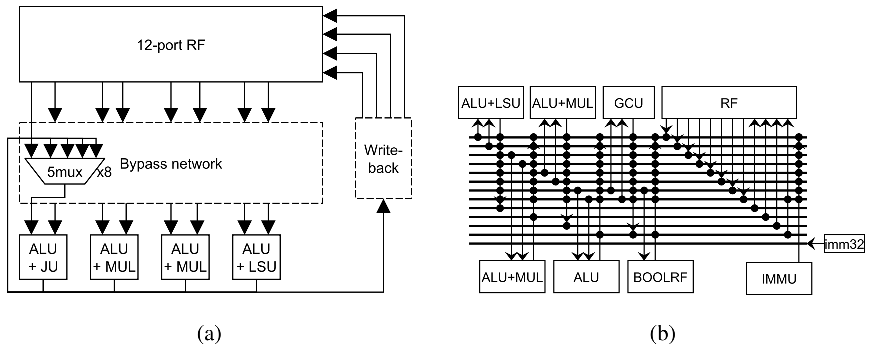

Fig. 5: (a) An example 4-issue VLIW datapath with a fully connected FU bypass network, and (b) the corresponding TTA architecture with equal connectivity.

number of equally highly utilized buses. Another option provided by TCE for automated connectivity optimization is *Connection Sweeper*. As the name suggests, it works in a brute force fashion, sweeping the interconnection network in a trial-and-error fashion, removing connections and testing the effect to the cycle count, stopping when the performance degrades more than a given threshold. It first tries to remove RF connections as they often reside in the critical path.

Other automated design assisting exploration tools in TCE include *Simple IC Optimizer*, a connectivity optimizer that simply removes all unused connections, a useful tool for FPGA softcore designs; *Machine Minimizer*, a brute force tool that tries to remove components until the cycle count increases above a given threshold (*Machine Grower* is its counterpart) and *Cluster Generator*, that helps to produce clustered-VLIW style TTAs. The automated exploration tools can be called from the command line, enabling a design methodology where the machine is partially constructed from a starting point architecture using scripts or *makefiles*.

6 Programming Support

Optimizing the code for the designed processor manually using its assembly language might be a last step taken after the processor design has been frozen and the programmer wants to squeeze off the last loose execution cycles in the application of interest. During the design process, however, the use of assembly is not feasible due to the changing target; whenever the architecture is changed, the affected parts of the assembly code would need to be rewritten. This would make the fast iterative evaluation of different architecture parameters practically impossible, or at least heavily restrict the practical number of evaluated architectures.

Other motivations for using high-level programming languages or intermediate formats as the application description format is to hide the actual instruction-set (which is very low level in TTAs) from the programmer. This avoids constraining new designs with legacy backwards compatibility issues, and the design process is simplified because instruction encoding specifics are pushed to the background.

In our experience, the benefits from using the assembly language in comparison to relying on an HLL compiler often stem from the inability of the compiler to extract adequate parallelism from the program description to utilize all the parallel processor resources, or from the inability to use complex custom operations automatically to accelerate the program. TCE attempts to alleviate this by taking advantage of, in addition to the usual C/C++ languages, the parallel OpenCL standard. The explicit and implicit parallelism of OpenCL C helps in the utilization issue while the automatically generated OpenCL vendor extensions of the TCE compiler can be used to execute custom operations manually from the source code [8].

Due to making the high-level programming of the designed processors a priority, a key tool in TCE is its retargetable software compiler, tcecc. The compiler uses the LLVM [10] project as a backbone and *pocl* [9] to provide OpenCL support. The frontend supports the ISO C99 standard with a few exceptions, most of the C++98 language constructs, and a subset of OpenCL C.

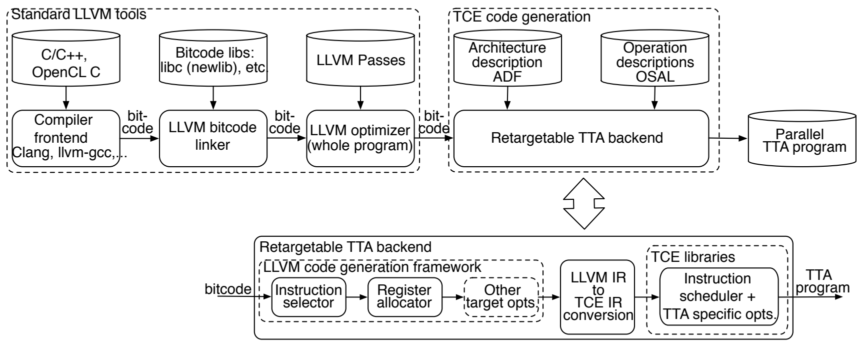

Fig. 6: tcecc compiler internals.

The main compilation phases of tcecc are shown in Fig. 6. Initially, LLVM's Clang frontend converts the source code into the LLVM's internal representation. After the frontend has compiled the source code to LLVM bytecode, the utility software libraries such as libc are linked in, producing a fully linked self contained bytecode program. Then standard LLVM IR optimization passes are applied to the bytecode-level program, and due to it being fully linked, the whole-program optimizations can be applied aggressively. The optimized bytecode is then passed to the TCE retargetable code generation.

To make custom operation evaluation easier, the automatically retargeting backend has the facilities to exploit custom operations automatically. Custom operations can be described in OSAL as data-flow graphs of more primitive operations, which the LLVM instruction selector automatically attempts to detect and replace in the program code.

Complex custom operations, such as those that implement long chains of primitive operations, or those that produce multiple results, often cannot be automatically found from intermediate codes produced from high-level language programs. For this, tcecc produces intrinsics that can be used manually by the programmer from the source code.

For example, this C code snippet calls a custom operation called 'ADDSUB' that computes both the sum and the difference of its operands in parallel and produces them as two separate results:

```
int a, b, sum, diff;
...
_TCE_ADDSUB(a, b, sum, diff);
```

The actual processor described in ADF must then include at least one function unit that supports the operation, one of which is triggered by code generated by the compiler. If the programmer wants to restrict the operation to be triggered in a specific function unit (making the source code more target specific in the process), another intrinsics version can be used:

```
_TCEFU_ADDSUB("ALU1", a, b, sum, diff);
```

This instructs the compiler to generate code to call operation ADDSUB in a function unit referred to as ALU1 in the targeted ADF.

7 Layered Verification

Using a component library based processor design with automated RTL generation is helpful in reducing implementation and verification effort. As the processors can be composed of preverified function unit and register file implementations, the effort required for new designs should reduce gradually as the databases are augmented with new validated components. However, additional verification is still usually preferred to gain trust on the produced processor implementation.

The approach TCE takes to verifying the designs is shown in Fig. 7. It uses a layered top-down approach where each level in the implementation abstraction hierarchy is compared against the previous one. At the first level, the designer, who uses a portable high-level language program as an application description, can implement and verify the software functionality using a desktop environment and the work station's CPU. Printouts to standard output can be used to produce the initial "golden reference" data.

Each layer of implementation abstraction can be compared to the golden reference data by including a standard output instruction in the processor architecture, used to produce the output at the different levels. In RTL simulation, the function unit implementation uses the VHDL or Verilog file printout APIs, and at the FPGA

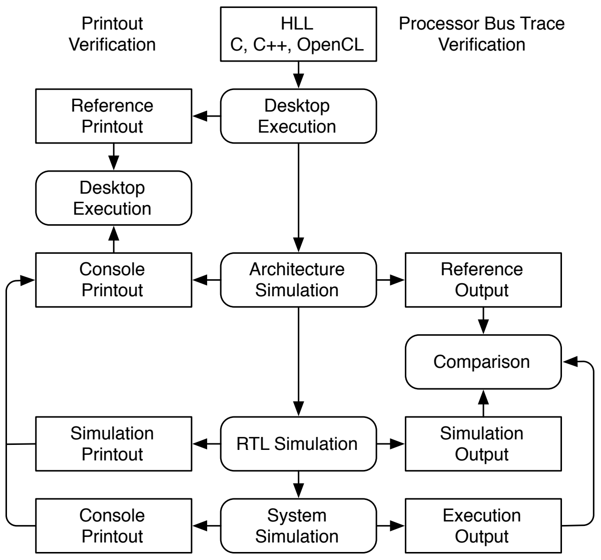

Fig. 7: Verification flow.

prototyping stage it may write to a JTAG-UART console. Further verification can be done by comparing bus traces which contain values in each processor transport bus at each instruction cycle.

Finally, the core is integrated to a system level model. TCE provides facilities to help producing project files and other integration files to different FPGA flows. System level simulation is supported via bindings for SystemC. The bindings allow plugging TTA instruction accurate core simulation models to larger SystemC models to enable cycle accurate performance modeling and functional verification simulations. The following is an example of how to integrate a TTA core simulation model to a system simulation by connecting its clock and global lock signals:

```
...
#include <tce_systemc.hh>
...
int sc_main(int argc, char* argv[]) {
   ...
   TTACore tta("tta_core_name", "processor.adf", "program.tpef");
   tta.clock(clk.signal());
   tta.global_lock(glock);
   ...
}
```

The FPGA flow can produce the same standard output printouts using a development board with a JTAG interface or similar. When using the system level simulation, verification data is produced similarly to the architecture simulator.

Post-fabrication verification is performed using suites of test programs that stress the different aspects of the processor design. At this point, additional verification and debugging can be done by using an automatically generated debug interface which can be used to output verification data and single step the processor.

8 Conclusions

In this chapter, we described TCE, a processor customization toolset based on the exposed datapath transport triggered architecture. TCE provides tool support for iterative processor customization starting from high-level programming languages, producing synthesizable RTL implementations of the processors along with instruction parallel binary code.

TCE is available as a liberally licensed open source project and can be downloaded via the project web page [12]. The toolset is mature and it has been tested with various design cases over the past decades, and is evolving with new developments focusing on the compiler code generation quality and energy saving features, both in the produced hardware and those that can be achieved with the compiler.

References

1. Cilio, A., Schot, H., Janssen, J., Jääskeläinen, P.: Architecture definition file: Processor architecture definition file format for a new tta design framework (2014). URL http://tce.cs.tut.fi/specs/ADF.pdf
2. Corporaal, H.: Transport triggered architectures: Design and evaluation. Ph.D. thesis, TU Delft, Netherlands (1995)
3. Corporaal, H.: Microprocessor Architectures: From VLIW to TTA. John Wiley & Sons, Chichester, UK (1997)
4. Corporaal, H., Mulder, H.: MOVE: A framework for high-performance processor design. In: Proceedings of ACM/IEEE Conference on Supercomputing, pp. 692–701 (1991). DOI http://doi.acm.org/10.1145/125826.126159
5. Dally, W., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R., Parikh, V., Park, J., Sheffield, D.: Efficient embedded computing. Computer **41**, 27–32 (2008). DOI http://doi.ieeecomputersociety.org/10.1109/MC.2008.224
6. He, Y., She, D., Mesman, B., Corporaal, H.: MOVE-Pro: A low power and high code density TTA architecture. In: Proceedings of International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), pp. 294 –301 (2011). DOI 10.1109/SAMOS.2011.6045474
7. Jääskeläinen, P., Kultala, H., Viitanen, T., Takala, J.: Code density and energy efficiency of exposed datapath architectures. Journal of Signal Processing Systems pp. 1–16 (2014). DOI 10.1007/s11265-014-0924-x. URL http://dx.doi.org/10.1007/s11265-014-0924-x
8. Jääskeläinen, P., de La Lama, C., Huerta, P., Takala, J.: OpenCL-based design methodology for application-specific processors. Transactions on HiPEAC **5** (2011). Available online
9. Jääskeläinen, P., de La Lama, C.S., Schnetter, E., Raiskila, K., Takala, J., Berg, H.: pocl: A performance-portable OpenCL implementation. International Journal of Parallel Programming pp. 1–34 (2014). DOI 10.1007/s10766-014-0320-y. URL http://dx.doi.org/10.1007/s10766-014-0320-y

10. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation Optimization, pp. 75–87 (2004)
11. Lipovski, G.: The architecture of a simple, effective control processor. In: 2nd Euromicro Symposium on Microprocessing and Microprogramming, pp. 7–19 (1976)
12. TCE: TTA-based co-design environment (2015). URL http://tce.cs.tut.fi
13. Thuresson, M., Själander, M., Björk, M., Svensson, L., Larsson-Edefors, P., Stenström, P.: FlexCore: Utilizing exposed datapath control for efficient computing. In: Proceedings of International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), pp. 18–25 (2007). DOI 10.1109/ICSAMOS.2007.4285729
14. Viitanen, T., Kultala, H., Jääskeläinen, P., Takala, J.: Heuristics for greedy transport triggered architecture interconnect exploration. In: Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '14, pp. 2:1–2:7. ACM, New York, NY, USA (2014). DOI 10.1145/2656106.2656123. URL http://doi.acm.org/10.1145/2656106.2656123