Hardware Generation for Transport Triggered Architectures

Sebastian Schumb
University of Kaiserslautern, Embedded Systems Group
s_schumb10@cs.uni-kl.de

Contents

1 Introduction **5**

2 Related Work **6**

3 Prerequisites **7**
3.1 Introduction into TTA 7
3.2 The Averest Framework 8
3.3 The F# language 10

4 TTA Implementation **12**
4.1 Buses 15
4.2 Control Unit 15
4.3 Sockets 17
4.4 Function Units 18

5 TTA Generator Tool **25**
5.1 File Formats 25
5.2 Quartz Template Library 28
5.3 Inputfile Processing 31
5.4 Architecture Generation 33
5.5 Quartz Code Generation 34

6 Test Scenarios **38**
6.1 Hello World 39
6.2 Fibonacci 41

7 Summary & Conclusion **42**

8 Future Work **43**
8.1 Architecture Implementation 43
8.2 Generator Tool 44
8.3 Averest 44

9 Bibliography **48**

10 Appendicies **49**
10.1 Test Scenario ADF 49
10.2 Test Scenario Address Space 52
10.3 Test Scenario VHDL Source 53
10.4 Hello World Test Module 56
10.5 Fibonacci Test Module 59

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfs- mittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Kaiserslautern den 31. März 2015

Abstract

Applicationspecific processors play an important role in modern embedded systems, since they can be optimized for a specific task. Transport Triggered Architectures are a well suited for this, as they can be easily customized and allow exploiting parallelism in programs at the instruction level.

This thesis is centred around the development of a transport triggered architecture implemented using the synchronous programming language Quartz, which is part of the Averest framework. The Averest framework can be used to simulate the architecture using its built-in unit testing capabilities. As a further step, it can generate Verilog code for the TTA, which can then be used for hardware synthesis.

Additionally, a TTA generator tool is presented to generate the Quartz source code for a TTA from an architecture description, which allows using the architecture as part of an automated hardware software co-design workflow.

Two test scenarios are introduced to evaluate the developed TTA and the generator tool. Both test scenarios are simulated using the Averest framework and synthesized to hardware, using the Averest framework's Verilog generation tool and a FPGA based test setup.

Zusammenfassung

Applikationsspezifische Prozessoren spielen eine immer wichtigere Rolle in modernen eingebetteten Systemen, da sie einfach auf eine bestimme Aufgabe zugeschnitten werden können. Transport Triggered Architectures sind hierfür gut geeignet, weil man sie sehr gut anpassen kann und weil sie es ermöglichen Parallelität in Programmen auf Interaktionsebene aus zu nutzen.

Diese Arbeit befasst sich mit der Entwicklung einer Transport Triggered Architecture in der synchronen Programmsprache Quartz, die Teil des Averest Frameworks ist. Das Averest Frameworks, mit seiner integrierten Unterstützung für Unit Tests, wird genutzt um die Architektur zu simulieren. Als weiteren Schritt kann es Verilog code für die TTA erzeugen, welcher zu Hardware-Synthese genutzt werden kann.

Zusätzlich wird ein Tool präsentiert, welches den Quartz-Code für eine TTA aus einer Architekturbeschreibung erzeugen kann. Dies erlaubt den Einsatz der Architektur im Rahmen eines automatisieren Hardware-Software-Codesign-Workflows.

Zwei Testszenarien werden vorgestellt im die entwickelte TTA und das Generator-Tool zu evaluieren. Beide Szenarien werden sowohl mit dem Averest Framework simuliert als auch zu Hardware synthetisiert und auf einem FPGA basierten Testsetup getestet.

1 Introduction

As of today, embedded systems have become a very important part of our every day lives. Smart phones and Tablets have rapidly become commodity hardware, replacing traditional computers in many use cases. Additionally, the development of the internet of things and home automation reached a point where equipping our homes with a network of small intelligent sensors and actors becomes affordable.

The requirements for processors used in these devices drastically differ from the requirements for CPUs traditionally used in computer hardware. As the computation speed becomes less important, the energy efficiency and the modularity of a processor architecture become key requirements. Modern processor designs are often optimized for a specific application, by adding or removing hardware peripherals to them. These so called systems on a chip are a logic next step from traditional processor designs, as they allow offloading parts of the computation required by an application, to dedicated hardware units. This makes it possible for the actual processor core to be smaller and simpler while still providing the performance necessary for its application.

Transport triggered architectures (TTA) can be seen as the next step in this development. In contrast to traditional processor architectures, a TTA has a rather minimalistic instruction set. Instead of providing separate instructions for each operation, a TTA processor exposes its internal buses directly to the program, allowing it to directly perform data transports between function units. Computations are executed as a side effect of data transport. A TTA usually has more than one bus, which enables the program to perform multiple bus transactions per cycle. Furthermore, the set of function units in a TTA processor can be heavily customized for a specific application. For example, if an application requires a lot of multiplications, more multiplier function units can be added to architecture enabling it to execute multiple multiplication operations in a single cycle. Since the control and data flow in a TTA is still controlled by a single bus control unit, the parallelism inside a program can exploited, without the additional synchronization overhead of a traditional multi core processor.

The topic of this thesis is the development of a TTA for future research at the Embedded Systems Group of the University of Kaiserslautern. After this introduction and brief summary of related research projects is given. Section 3 will briefly introduce the tools that have been used in the development process. Following that, Section 4 will explain the TTA architecture, which has been designed using the programming language Quartz and the Averest framework [11], also developed by the Embedded Systems Group. This Section will start with a generic overview of the architecture, before discussing the implementation of buses, control unit and the function units in detail. Since creating many TTA processors manually using the Quartz source code is time consuming and error prone, a automated TTA generator tool has been developed. This tool will be the topic of Section 5, starting with a general overview over the tool's architecture. After that, the input file formats for the TTA configurations will be introduced, as well as the Quartz module templates used for code generation. This will be followed by a detailed overview over the different steps performed by the tool to produce the Quartz source code for a TTA. Section 6 will present the test scenarios that have been used during the development phase. This Section will also briefly explain the steps necessary to synthesize a hardware circuit for a FPGA starting from the Quartz code for a TTA processor. Finally, the thesis work is summarized in Section 7 and in concluded in Section 8 by mentioning future work.

2 Related Work

There are two research projects working on the generation of hardware and code for Transport Triggered Architectures.

The more recent one is the *TTA-based Co-design Environment* [13] which is developed by the *Department of Pervasive Computing* at the *Tampere University of Technology*. The goal of the TCE project is to provide a fully integrated development environment for hardware software co-design based on TTAs. It features graphical editors for composing CPUs from a library of function units and for creating the interconnecting structure between these units. A CPU simulator can then be used in combination with a set of analysis tools to identify computational hotspots and performance bottlenecks for a specific piece of machine code running on the generated architecture. If the architecture meets the required performance constraints, it can be converted to VHDL for the actual hardware synthesis. Using the hardware synthesis tools provided by FPGA vendors, the VHDL code can then be translated into a bitstream to implement the design on an FPGA. The code generation for the architecture is handled by a custom plugins for the *LLVM*-backend [10]. Since the LLVM framework is used as base for the compiler it is possible to make use of wide range of existing software for the code generation process. This makes it possible to use the LLVM frontend for C, C++ and OpenCL to generate code for a TTA processor. Furthermore, optimizers which are already part of the LLVM framework can be applied during the code generation. A subset of the XML-format [6] used by the TCE tools is used by the architecture generation tool described later in this document.

The second notable project is the *MOVE* project [7], which was developed by the *Department of Computer Engineering* at the TU Delft. MOVE is the predecessor of TCE. The hardware architectures for the move project were designed from a library of standard logic cells using a Very Large Scale Integration process. Since FPGA were not feasible for the required complexity at that point in time, the architectures had to be implemented as ASICs using CMOS technology. So instead of a higher level hardware description language, the generated architecture was directly rendered as a layout file for fabricating CMOS ASICs. A separate set of optimizer tools allowed to optimized the generated layouts with respect to area, timing or power consumption constraints. The optimizer tools tried to leverage knowledge about the application of a processor, provided by the designer, as well as information from the standard cell library with a linear optimization algorithm. Among the designs which have been actually fabricated during the project is a small TTA processor with three buses, the MOVE32INT processor with four 32 bit wide buses and a set of three TTAs, which has been integrated into a MPG2 video codec chip. Also, the routing of signals between the logic cells could be optimized using third party standard cell routing software to meet the constraints for the fabrication process. The compiler was based on the GNU C-Compiler with a custom backend for generating TTA compatible code. Implementing an architecture as ASIC required a lot of development time, therefore the MOVE toolchain also contained an architecture simulator and debugger, that allowed executing the generated code without the actual hardware. The development of the MOVE project was stopped in favour of its successor TCE.

3 Prerequisites

The following Sections contain an introduction into transport triggered architectures in general, as well as an introduction into the Averest framework and the F# programming language, which have been used to implement to TTA generator tool described later.

3.1 Introduction into TTA

A processor based on a transport triggered architecture usually has a rather small instruction set. In the most extreme case only one instruction for moving data from one bus address to another address is provided. This is because in traditional processor architectures, the computation operations are encoded in the machine instructions. In contrast to that in transport triggered architectures computations are just a side effect of data transport. Moving data to certain bus addresses triggers operations and the results can be retrieved at other bus location once the computation is completed.

The central element in each TTA processor is the bus control unit. It fetches instructions from memory and executes them as bus transactions. Since an architecture can have more the just one bus interconnecting the function units, it is possible to execute multiple bus transactions per cycle in parallel. This is usually implemented by grouping one instruction for each bus into a single instruction word, similar to very large instruction word architectures. In addition to executing bus transactions the control unit also maintains a program counter and is responsible for the control flow of a program. Jumps can either be implemented by exposing the program counter to buses or as a separate instruction. The first possibility reduces the logic required inside the control unit at the cost of increased code complexity for conditional jumps. The second implementation increases the required logic for the control unit, but it reduces the complexity of the programs. Also, conditional jumps can be implemented with less effort if they are implemented directly in the control unit instead of having the program alter the program counter. Therefore, the second approach has been chosen for the TTA presented later in this document.

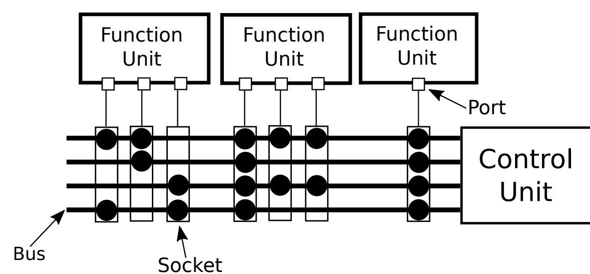

Figure 1: Basic structure of a TTA

The actual computation in a TTA is done by function units. Function units expose ports to the bus interconnect via sockets. Generally there are three types of ports: input ports, trigger input ports and output ports. Input ports are used to input data from the buses into a function unit. The output ports provide data from inside the function units to the buses. If the trigger input port is written, the function unit starts the computation. After finishing, the result is written out to an output port where it can be retrieved via the buses. The complexity of function unit can range from simple adders or multipliers, to more complex operations such as matrix multiplication. Since a function unit can take more than one cycle to complete, units can be designed to use pipelining internally. There is no default set of function units for a TTA, function units can be chosen as needed by the application of the TTA. This makes TTA a good base for applicationspecific processor designs. The use of function units is not only limited to implementing computations. It is also possible to implement the connection to the ram or general purpose input or outputs of a system as a function unit.

Sockets are used connect buses to a function units ports. Ports can be seen as input and output

registers of a function unit. A socket is responsible for decoding bus addresses and reading data from the bus into the port register or the other way around. This design allows the bus connection to be separated from the actual implementation of the functional unit. If a write occurs to the triggering register of the FU, the socket should also trigger the computation inside the FU. It can be advantageous to use different types of sockets in a TTA. For example some ports are only used as output, so the socket connected to those ports does not need the logic to handle writes to the port. It can also be useful to map multiple bus addresses to a single port of the function unit. For example an ALU may have only two input ports for both operands. The socket connected to one of these inputs ports can then map one operand port to multiple bus addresses, one for each function the ALU can perform. If the socket also forwards the written bus address to the ALU, it can be used to determine which operation has to be performed. This eliminates the need for multiple trigger ports and sockets, reducing the overall complexity of the processor for hardware synthesis.

The last component of the TTA is the bus interconnection. The buses of a TTA can be implemented in various different ways, ranging from a simple set of parallel signals for source address, target address and data to a more sophisticated bus system. If each socket is connected to every bus, the system is called fully interconnected. Since fully interconnected system required a lot of hardware resources for address decoding and bus management it is usual to optimize the number of buses a socket is connected to. Partially interconnected systems leverage the fact that a port usually can only be written from a single bus in one cycle therefore it might be sufficient to only connect one bus to the socket.

3.2 The Averest Framework

Averest [11] is a framework for automated verification and hardware software synthesis, developed by the Embedded Systems Group at the TU Kaiserslautern. Its central element is the synchronous programming language Quartz.

Quartz [14] is a strongly typed language with a syntax similar to the C or C++. Unlike other programming languages it is based on well defined formal semantics, making it an ideal language for automated verification. Since Quartz is a synchronous language, programs are executed differently compared to programs in sequential languages. The statements in Quartz program form so called micro steps. All micro steps between two *pause* statements are grouped into a macro step. The micro steps inside a macro step are assumed to be executed all in parallel and in zero time, whereas a macro step can be seen as a logical step forward in time. The Quartz compiler uses a fix point iteration combined with a causality analysis to ensure at compile time that the micro steps inside a macro step are executed in the correct order. This also ensures that each variable is assigned one unique value during a macro step.

```
 1 // Macrotep 1
 2 pause;
 3
 4 // Macrostep 2
 5 b = 42;
 6 c = b * a;
 7 a = 23;
 8 next(a) = a + a;
 9 pause;
10
11 // Macrostep 3
```

Listing 1: Micro and macro steps in Quartz

```
 1 // Macrostep 1
 2 a = 2;
 3 b = 5;
 4 {
 5     c = b * a;
 6     pause;
 7
 8     // Macrostep 2
 9     c = 42;
10 }
11 ||
12 {
13     d = b + a;
14     pause;
15
16     // Macrostep 2
17     d = 23;
18 }
```

Listing 2: Micro and macro steps in Quartz

Listing 1 shows an example of Quartz program with three macro steps. The first macro step of the example contains no statements. In the second macro step, multiple assignments are executed, which are the micro steps of this macro step. Due to the data dependencies of the assignment for *c*, in line 6, on the assignments for *a* and *b*, in line 5 and 7, the micro step containing the assignment for *c* will be scheduled after the other two micro steps. The third macro step is similar to the first one as it also contains no assignments. This example also illustrates another important aspect of the Quartz language. Assignments like $x = x + 1$ are not possible because the micro step would have a data dependency to itself. Therefore, the Quartz compiler would be unable to determine a unique value for x. The same problem occurs for larger dependency cycles as well.

To avoid such cycles Quartz allows delayed assignments using the *next* keyword. The value for delayed assignment is computed using the variable values from the current macro step, but the actual assignment is executed in the next macro step. In the example in Listing 1 *a* is assigned the value 46 in the third macro step, while the assignment statement is located in the second macro step. Since the value of *a* only depends on the value of *a* in the previous macro step, there is no problem for scheduling the micro steps.

The behaviour of variable is similar to the behaviour of electric signals in a logic circuit, as signal in a circuit can not depend its current value. In such cases a register has to be introduced in the circuit, acting similar to the delayed assignment. Additionally, there are two different storage models for variables in quartz. Variables can be either declared as *event* or *memorized*. Memorized variables keep an assigned value over multiple macro steps, until a different value is assigned to them. Event variables on the other hand keep a value only for a single macro step and will be reset to a default value in the next macro step unless there is an assignment in that macro step or a delayed assignment in the current macro step.

Listing 2 is an example for another important aspect of the Quartz language. Quartz offers operators to execute blocks of code or whole modules in parallel. The ||-operator used in this example runs both code blocks parallel, while keeping the macro steps of both blocks synchronized. For this particular operator the execution of the composed blocks ends as soon as all macro steps of the block with the more macro steps have been executed. Other operators allow for different behaviours in case blocks have different number of macro steps. For example it is also possible to finish the execution of the

composition as soon as one of the blocks has executed all its macro steps. This feature allows a very detailed modelling of concurrent systems.

Quartz programs can be structured in modules and packages. Modules are functional blocks that can be instantiated in multiple places in the source code. It is also possible to instantiate a module inside other modules enabling efficient code reuse. Packages can be used to group multiple modules and other packages together hierarchically. Furthermore, the programmer can specify test cases in the form of drivenBy-test for each module in the Quartz source code. A drivenBy-test is written as regular Quartz code, providing all features of the Quartz language. The inputs and outputs of the module under test can accessed as local variables inside the test. This provides a simple way to set inputs of the module and check the output using assertions.

A Quartz program can be compiled to the *Averest Intermediate Format* using the *qrz2aif* utility. AIF uses synchronous guarded actions to represent the control and data flow specified in the Quartz program. A synchronous guarded action is a tuple of a boolean expression, the guard, and an assignment to a variable, the action. If the guard holds, the action of a guarded action is executed. The control flow of a Quartz program can be represented by introducing additional boolean variables as control flow labels and additional guarded actions to update these labels. The *aif2txt* tool can be used to convert the XML data inside AIF files into a human readable list of guarded actions.

The AIF files can be used as input for simulation as well as for code generation to different languages. The *aif2trc* tool simulates the behaviour of the program based on the synchronous guarded actions, by executing the test cases which have been specified in the quartz source code and checking the assertions. A trace of all variable assignments for each macro step is provided as output. Using *aif2trc* enables the programmer to develop systems in Quartz using a test driven design approach. Invoking the *aif2verilog* tool allows generating verilog code from the synchronous guarded actions inside the AIF file. As mentioned later in this document the output of *aif2verilog* is not alway valid synthesizeable Verilog. Therefore, a few corner cases have to avoided when writing Quartz code that is meant for Verilog synthesis. If the result is valid Verilog it is possible to generate a bitstream for FPGA hardware from it using a FPGA vendors synthesis toolchain.

3.3 The F# language

The processor generator tool presented later in this document was developed in *F Sharp* [8]. *F#* or *F Sharp* programming language was initially developed by *Microsoft Research* in 2005. Since 2010 the compiler and standard libraries have been released under the open Apache Licence 2.0. F Sharp is a multi-paradigm language as it allows the programmer to combine imperative, object-oriented and functional code. The language itself has been heavily influenced by functional language like OCaml or Haskell, whereas grouping code to blocks is done using indentation similar to Python. While the language is strongly typed, type annotations can be omitted in many cases.

The compiler makes extensive use of type inference to ensure the type correctness of programs. There are two compilers for F Sharp available. The first one is an addon to Microsoft Visual Studio and is therefore restricted to windows-based platforms. The second compiler is the open source fsharpc by *F# Software Foundation* which is available for most platforms. Both compilers also feature a *F Sharp* interpreter, that can be used to interactively run test code in a command line environment.

The Microsoft .Net frameworks *Common Language Interface*, *CLI* for short, acts as a standard library for *F Sharp* programs. In addition to that *F Sharp* has a built-in functional library, since writing functional code using the object oriented .Net libraries results in unnecessarily complex code. The built in functional types are easily convertible to their .Net equivalents in most cases. *F Sharp* software is compiled to

assemblies for the .Net virtual machine. Under Windows a .Net virtual machine is required to run the assembly. For Linux the open source mono virtual machine [2] can be used.

The processor generator tool was developed using the fsharpc compiler and mono under Linux. It tries to use .Net types where ever possible and only resorts to the built-in types if functional code is used.

4 TTA Implementation

The TTA generator described in this document requires a library of function units, sockets and templates. Templates are Quartz modules containing place holders, which are later replaced, using values or Quartz code blocks, by the generator tool. The format of templates is explained in detail in Section 5.2.

Therefore, the first development step for the generator tool has been to create a TTA implementation in Quartz. The initial TTA implementation has been developed as complete standalone architecture, which can be simulated, tested and synthesized to Verilog code without the generator tool. There have been five primary design goals for the Quartz TTA implementation.

The first one has been code readability. Since the Quartz code is the basis for the templates used by the generator tool, which enables to generate large architectures automatically, the ability to debug these architectures greatly depends on how readable the code of the templates is. Unfortunately the *aif2verilog* utility does not always produce optimal Verilog code. In such cases an improvement in the performance of the Verilog source can be gained, by modifying the Quartz source with respect to the Verilog code generation. This often leads to a Quartz source, being a lot less intuitive and also a lot more complex to maintain, since even small changes can result in a total loss of the performance improvements. Also, considering the fact that these issues should be treated rather as bugs in *aif2verilog*, than bugs in the Quartz source code, the simpler forms of the Quartz source have been preferred, when possible.

The second design goal has been modularity. Modularity is especially important for the implementation of a TTA, since one of its advantages is the ability to add own function units. A modular design also provides to ability to reuse code, reducing the amount of code to be maintained and tested. The *Alu* function unit discussed in detail in the following section is a good example for reusing code. The unit itself only contains code to dispatch operands to internal instances of other function units, such as adders or multipliers. This does not only reduce the amount of Quartz code required, it also simplifies the testing. Assuming the function units used internally have their own set of tests and work correctly, only the operand dispatching inside the ALU unit needs to be tested.

Quartz has the ability to add unit tests to each module in form of *drivenBy*-tests. Therefore, the third design goal has been to provide a full set of tests for each Quartz module. On the one hand, this allows a test driven approach to development and on the other hand, it also enables to check for regressions after refactoring code. Unfortunately writing tests working with variable widths for data words has proven to be difficult and error prone. Hence, the tests all assume a fixed width of 8 bits for data and addresses. The templates used for the generator tool need to work with variable bit widths, requiring to remove of the unit tests from the final templates.

Another important aspect of the TTA implementation has been to minimize the amount of code that later needs to be generated. Moving as much complexity as possible away from the generator tool into the Quartz source has several advantages. First of all, it enables a simpler implementation of the generator tool itself, therefore reducing the effort required to maintain the tool source code. It also simplifies later changes to the tool. The reduction of generated code also makes architectures generated by the tool easier to debug, considering that most of the output is code from the templates, which already has a set of tests.

The last design goal was to ensure that synthesizable Verilog code can be generated for the TTA. Verilog has initially been developed as a simulation language. The capability to do hardware synthesis has been added later by FPGA vendors and is not standardized across different synthesis toolchains. As a result each toolchain only supports its own subset of Verilog for synthesis. These subsets are generally referred to as synthesizable Verilog. While there is no known case where the *aif2verilog* tool generates Verilog that is not synthesizable, it can in some rare cases generate invalid Verilog code. These cases

are discussed in more detail in later sections of this document. Furthermore, the *aif2verilog* shipped with the latest version 2.3.2.0 of Averest fails to produce Verilog for Quartz codes, working fine with version 2.2.4.5. Due to this issue, version 2.2.4.5 was used to develop the TTA implementation. In addition to these limitations imposed by the toolchain used, there are also limitations created by the target hardware. Since FPGA hardware has only a limited number of adders, subtractors or multipliers, the Verilog code must not use too many of these hardware blocks in order to synthesize to a fast hardware implementation. This has been ensured during the development phase by generating the Verilog code for each Quartz module and then checking the required resources, using the Xilinx hardware synthesis toolchain. In some cases the Quartz code had to be reformulated in order to avoid using unnecessary multipliers in hardware.

Furthermore, the final architecture aims to be highly customizable. Consequently, the number of buses and the width of addresses and data for the buses can be configured freely. While in theory arbitrary bit widths are supported, widths should be chosen as multiple of 8 bits for any architectures that will be synthesized to hardware. The reason for this restriction is, that memory is usually aligned to bytes and therefore the width of an instruction which contains one opcode, one source address or immediate operand and one target address should be a multiple of 8 bits. This ensures that the length of complete instruction words, consisting of multiple instructions, is a multiple of 8 bits as well.

Another important feature of the TTA implementation is that there are no function units required by default. The bus control unit itself is able to fetch, decode and execute all possible instructions. A function unit only has to be added to the architecture, if it is actually required to perform computations needed by the program. This allows to always chose the optimal set of function units for each application.

The current architecture design assumes that the program and its data are kept in two different memories. This has two main reasons. First of all, the current design of the bus control unit assumes that the program memory has the same width as one instruction word. While using the same memory width for RAM and program memory is not a problem for simulation, it causes some issues for hardware synthesis. The instruction words are usually at least twice the size of data words, which means only half the storage space for each RAM address can be used by the program. Since the amount of RAM on a FPGA is also limited, it is more efficient to split it into two separated memories: One with a high width for storing the program and one with a lower width to be used as RAM for data. It is also important to keep in mind that the interface to the RAM is just a function unit. Consequently, it is possible to create architectures that do not provide RAM access to the program. In such cases creating RAM on the FPGA can be omitted, while program memory is still required. Secondly, typical block RAM implementations on FPGAs can be accessed at two independent locations at once. These so called dual port memories can be connected to a TTA using two *Ram* function units, allowing the processor to concurrently access the RAM. If code and data share the same memory, this would not be possible, because one port would be required by the control unit to fetch the instructions.

The structure of the Quartz modules used in the implementation was chosen to make it easier to automatically generate different architectures. Figure 2 shows the overall structure. The top level Quartz module is the composed processor module. It contains the control unit module instance along with a composed module instance for each function unit.

```
1  macro nat2BusAddress(number) = nat2bv(number, BusAddressWidth);
2
3  module composedXor(event [BusCount]bv{BusAddressWidth} busSource,
4                     event [BusCount]bv{BusAddressWidth} busTarget,
5                     event [BusCount]bv{BusDataWidth} busData) {
6
```

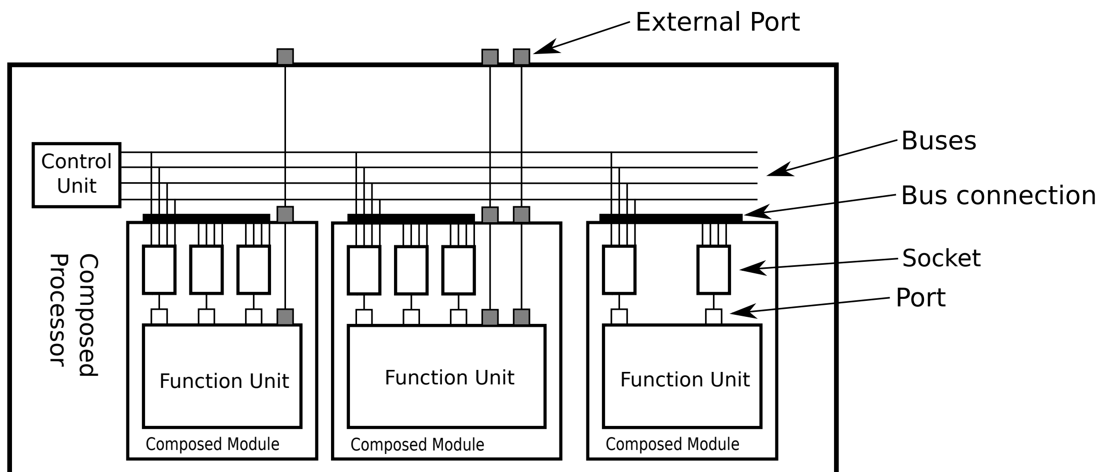

Figure 2: Structure of Quartz modules used in the TTA implementation

```
7      mem bv{BusDataWidth} op1Register;
8      event bool op1Trigger;
9
10     mem bv{BusDataWidth} op2Register;
11     event bool op2Trigger;
12
13     mem bv{BusDataWidth} resultRegister;
14
15     op1Socket : InputSocket(busSource, busTarget, busData,
16                             nat2BusAddress(40), op1Trigger, op1Register);
17     ||
18     op2Socket : InputSocket(busSource, busTarget, busData,
19                             nat2BusAddress(41), op2Trigger, op2Register);
20     ||
21     resultSocket : OuputSocket(busSource, busTarget, busData,
22                             nat2BusAddress(42), resultRegister);
23     ||
24     Xor1 : Xor(op1Register, op2Register, resultRegister, op1Trigger);
25
26   }
```

Listing 3: Composed module for the xor function unit

The composed modules each contain the function unit's module instance and instances of the required socket modules. As shown in Listing 3, sockets are configured and connected to the function unit's module inside the composed module. Composed modules only expose a single bus connection and the external ports of their function unit as an interface. The external ports of a function unit are forwarded through all hierarchy levels to the top level Quartz module. This structure has also advantages for the generator tool, as only the source for the composed modules and the composed processor module have to be generated. The sources for the control unit module, socket modules and the function unit modules only require some configuration macros to be set.

Unfortunately the *aif2verilog* tool generates invalid Verilog source, when an array of bit vectors is used as a top level module's parameter. This is especially the case for the array of instruction bit vectors,

used to pass the instruction word to the control unit. As a workaround, the *VerilogWrapper* module has been designed, which uses a single instruction word bit vector as parameter, splits it up into instructions and assigns them to an array of instruction bit vectors accordingly. Details on this issue can be found in Section 8.3.

4.1 Buses

The buses interconnecting the function units and the control unit are implemented as arrays of bit vectors, where each group of array elements at the same index represents one bus. Two bit vector arrays are used as target and source address for bus transactions. The third bit vector array is used for the actual data being transferred.

A bus transaction is initiated by the bus control unit, which sets the source and target addresses. The socket responsible for the source address sets the data lines accordingly and the socket for the target address transfers the data into a register shared with the function unit. There are no extra signals required to mark the addresses or the data as valid, since the scheduling of microsteps done by the Quartz compiler ensures the correct order of read and write operations during a bus transaction.

Initially an array of three tuples was used as abstraction for the buses, but unfortunately the *aif2verilog* utility is unable to generate Verilog source code for some Quartz constructs involving tuples.

As a convention, the bitvectors used for implementing the buses should always be the first three parameters of any connected Quartz module. Also, they should be named *BusSource*, *BusTarget*, *BusData*. The two addresses should have the data type *event [BusCount]bv*{*BusAddressWidth*} and the data should use *event [BusCount]bv*{*BusDataWidth*}, where *BusCount*, *BusDataWidth* and *BusAddressWidth* are configuration macros, which will be filled in later by the generator tool.

To simplify the design only fully interconnected architectures are supported for now. The main reason for this is that a not fully interconnected design may require a much more complex logic for the bus interconnect. This would not only increase the code complexity, but also increase the amount of combinatorial logic used for in decoding bus addresses, which has a negative influence on the performance of the synthesized hardware. In addition to that, the algorithm in the generator tool, generating the bus address allocation and the sockets instances, would become more complex.

4.2 Control Unit

The bus control unit is the central element of the TTA implementation. It reads the instruction words from memory, executes the bus transactions and updates the program counter.

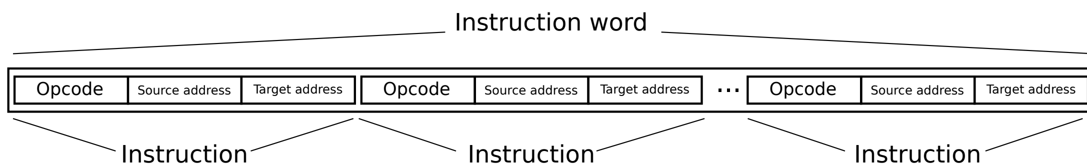

Figure 3: Structure of the instruction word

Figure 3 shows the structure of an instruction word. It consists of one instruction for each bus. The instruction is composed of three bitvectors.

The first bitvector contains the opcode for the instruction. Even tough there are only 4 instructions implemented yet, the opcode already has 8 bits width. The main reason for this is that the widths of RAM

and ROM blocks on FPGAs can only be set in multiples of 8 bits. Therefore, using 8 bits for the opcode and using bus addresses which are multiples of 8 bits ensures that the RAM or ROM for storing the program can be synthesized efficiently. Additionally, using 8 bits provides spaces for future extensions of the instruction set. The second part of the instruction is used as either a immediate operand by the load instruction or as a source bus address. The last part contains either a target bus address or a target location in the program memory for a jump. The program memory is expected to have the same width as the instruction word. This simplifies the control unit, since the program counter can be used directly as a address in the program memory and a single read on the memory yields a complete instruction word.

At the end of a cycle the control unit updates its program counter output and expects the instruction word at this location at its instruction word input at the beginning of the next cycle. In the first cycle after the processor was started a NOP is executed, since the program memory will typically need one cycle to read the instruction word for address zero.

The control unit also reserves two bus addresses : 0 and 1. Address zero is used as source or target to indicate that a bus transaction has no source or no target. This address used whenever the control unit itself is the source or target and should therefore never be handled by a socket. The address one is used to provide read and write access to the program counter from the bus.

Mnemonic	Opcode	Operand 1	Operand 2	Meaning
NOP	0x00	Don't care	Don't care	No operation
MOVE	0x01	Source address	Target address	Set target and source address on the bus
LOAD	0x02	Value	Target address	Set target address and put the value on the bus
JMP	0x03	(Source address)	Target location	(Conditional) Jump to target location

Table 1: Instruction set of the bus control unit

An overview of the currently implemented instructions can be found in Table 1. The NOP instruction ignores both source and target address and does nothing. Its opcode was chosen to be 0x00 and as a result a zero initialized program memory can be treated as a valid program containing only NOPs.

The MOVE instruction is usually the most used instruction in a TTA program. It sets the source and target address of a bus to the instruction's operands. After that, the sockets responsible for these bus addresses set or read the data lines of the bus accordingly.

Loading constants can be achieved using the LOAD instruction. The LOAD instruction writes its first operand to the data lines of the bus, while the second operand is used as target address. The source address is set to zero to indicate that the bus transaction does not have a source socket.

A simple way to interact with the programs control flow is provided by the JMP instruction. It can be used for conditional jumps as well as for unconditional jumps. In both cases the second operand is used as an absolute target address for the jump. For an unconditional jump the second operand should be set to zero, indicating that no bus address should be read. For a conditional jump on the other hand, the second operand of the instruction can be set to any valid bus address. The control unit will then set this address as source address on the bus and the socket handling this address will set the data lines accordingly. If the data written to the bus is zero, the JMP instruction is ignored. Otherwise, the jump is executed and the program execution continues at the target location for the next cycle. Using multiple JMP instructions in the same cycle will result in undefined behaviour. Consequently, this constellation should be avoided by the compiler generating the program.

4.3 Sockets

The concept of sockets is used to separate the logic for bus address decoding and matching from the actual function units. Therefore, it is possible to change to a different bus implementation without any adjustment to the function units. Function units are connected to sockets via shared registers and up to 3 additional signals depending on the type of socket. The term port will be used to refer to this register and the additional signals from here on.

```
1 module Socket(event [BusCount]bv{BusAddressWidth} ?busSource,
2     event [BusCount]bv{BusAddressWidth} ?busTarget,
3     event [BusCount]bv{BusDataWidth} busData,
4     mem bv{BusAddressWidth} ?busAddress,
5     event bool !trigger,
6     mem bv{BusDataWidth} register) {
7
8     ...
9 }
```

Listing 4: Signature of the Socket module

Listing 4 shows the signature of a basic *Socket* quartz module with six parameters. As per the convention introduced in Section 4.1, the first three parameters are the bus source addresses, the bus target addresses and bus data lines. The fourth parameter is the bus address of the socket. This parameter is supposed to be set to a constant value in the source code of the composed module, where the socket is attached to a function unit. The next parameter is the trigger output of the socket. If the socket is connected to the port of a function unit, which triggers the computation in the unit, this signal will be connected to the trigger input of the module. The last parameter is the register shared between the socket and the function unit. A complete example for this can be found in Listing 3.

If one of the source bus addresses matches the bus address parameter, the socket sets the bus data lines to the contents of the shared register. Similarly, if one of the target bus addresses matches, the data on the bus will be written to the register. It is not supported to read data from a socket on one bus, while writing to the same socket at another bus. This leads to undefined behaviour and should be avoided by the compiler, when scheduling the MOVE instructions. Writing to the same socket from two different buses is not supported as well.

Since in most cases sockets are either used only as input or only as output, there are specialized *InputSocket* and *OutputSocket* modules. The *InputSocket* can only be used as target and will write zero to the bus, if invoked as source. Likewise, using an *OutputSocket* as target will have no effect on its shared register. The *OutputSocket* also does not have a trigger output. Using these modules instead of the bidirectional Socket module reduces the amount of required hardware resources.

Other types of functional units may require other kinds of socket behaviour. Therefore, two other types of sockets have been implemented in addition to the basic sockets. The first one is the *MultiaddressInputSocket*. It allows mapping a range of bus addresses to a single socket and adds a parameter containing the offset of the last written address, relative to the base address of the sockets address range, to the function unit port. This kind of socket is, for example, used for the *Alu* function units first operand port. The socket attached to this port maps to a range of 20 addresses. As it also triggers the unit a write to this socket will update the register and the offset parameter of the port and then start the computation inside the *Alu*. The operation to be performed will be selected depending of the offset parameter. Using this design pattern reduces the amount of required sockets for the *Alu* from 20 to one, but introduces a subtractor into the address decoding, which can slow down the overall system. Similar to the basic socket

this socket should only be the target for one bus in each cycle. Writing to it concurrently will result in undefined behaviour. Also since this socket is only intended for input reading from it will always yield zero.

```
module ParallelMultiAddressSocket(
    event [BusCount]bv{BusAddressWidth} ?busSource,
    event [BusCount]bv{BusAddressWidth} ?busTarget,
    event [BusCount]bv{BusDataWidth} busData,
    mem bv{BusAddressWidth} ?RegisterBaseAddress,
    mem nat{MaxAddressOffset + 1} ?RegisterAddressCount,
    event [BusCount]bool !trigger,
    mem [BusCount]bv{BusDataWidth} ?readValue,
    mem [BusCount]nat{MaxAddressOffset} !readOffset,
    mem [BusCount]bv{BusDataWidth} !writeValue,
    mem [BusCount]nat{MaxAddressOffset} !writeOffset) {

    ...

}
```

Listing 5: Signature of the ParalleMultiAddressSocket module

The signature of the last and most complex type of socket, the *ParallelMultiaddressSocket*, is shown in Listing 5. Like the *MultiaddressInputSocket* this socket type handles a range of bus addresses instead of a single address. The port, this socket can be attached to, consists of two shared registers and two offsets for each bus. If the socket is used as a target by a bus, the *writeValue* register and the *writeOffset* parameter pair corresponding to this bus are updated. Also, the trigger flag is set similar to socket types discussed before. If the socket is used as a source instead, it updates the *readOffset* parameter. The function unit attached to it is then responsible for updating the *readValue* register for the same bus to the correct value. This value is then outputted to the bus data lines by the socket. Updates of the *readValue* registers must happen in the same cycle, independent of any computations performed by the unit. The unit should not need to be triggered to perform these updates. Since there is one register and offset pair for each bus, this type of socket is the only one that can be written and read concurrently by multiple buses, as long as no writes conflict with each other. In this case the behaviour of the socket is still undefined. This socket is mainly intended to connect the *RegisterFile* function unit to the buses, where the offsets can be used to identify the register, being read or written. Similar to the MultiaddressInputSocket this socket reduces the amount of required hardware resources while increasing the complexity of the address decoding.

Other types of sockets, like a *ParallelMultiAddressInputSocket*, could be implemented as well, but do not have a use case with the current set function units. Hence, they are not included in the current TTA implementation.

4.4 Function Units

The computations inside transport triggered architecture happen as a side effect of bus transactions inside the function units. In this implementation each function unit is a Quartz module. To be able to automatically generate the composed modules, the parameters for theses Quartz modules follow a naming convention. The first parameters of the module are the function units ports. As mentioned earlier in Section 4.3, a port is a group of parameters used for connecting the function unit to one of its sockets.

The next parameter is the trigger input of the module. Parameters following the trigger input are the external ports of the module. External ports are parameters which are simply forwarded through all levels of the Quartz module hierarchy and are exposed on the top level module. To distinguish external ports from the regular parameters the prefix *ext* is added to their names. These ports can be used by a function unit to interact with hardware outside the processor, like a block RAM on a FPGA, or as general purpose inputs and outputs.

```
1  module Ram(mem bv{BusDataWidth} ?address,
2              mem nat{MaxAddressOffset} ?addressOffset,
3              mem bv{BusDataWidth} value,
4              event bool ?trigger,
5              mem bv{BusDataWidth} !extAddress,
6              mem bv{BusDataWidth} ?extReadValue,
7              event bool !extWriteEnable,
8              mem bv{BusDataWidth} !extWriteValue) {
9          ...
10 }
```

Listing 6: Signature of the Ram function unit

Listing 6 shows the signature of the *Ram* function unit, which is a good example of this convention. The first two parameters *address* and *addressOffset* form a port for a *MultiAddressInputSocket*. The next parameter *value* is a port for a basic *Socket*, followed by the trigger input of the unit. After that, all parameters are external ports used to connect the actual RAM to the unit. Theses parameters are forwarded to the top level module so that they can be connected to the external hardware blocks.

The implemented function units are grouped in three categories: bitwise logic, arithmetic and memory. The bitwise logic package contains the common unary and binary logic operators. This allows direct bitwise operations for bit manipulation as well as boolean logic if the values 1 and 0 are used to represent true and false.

Function Unit	Ports	Operation
And	$op_1, op_2, result$	$result := op_1 \wedge op_2$
Or	$op_1, op_2, result$	$result := op_1 \vee op_2$
Xor	$op_1, op_2, result$	$result := op_1 \oplus op_2$
Not	$op_1, result$	$result := \neg op_1$
ShiftLeft	$op_1, op_2, result$	$result := op_1 \ll op_2$
ShiftRight	$op_1, op_2, result$	$result := op_1 \gg op_2$

Table 2: Function units of the bitwise logic package

Table 2 shows an overview of the function units available in the bitwise logic package. The ports op_1 and op_2 have to be connected to InputSocket, whereas the *result* port has to be connected to a OutputSocket for all theses units. All function units inside these packages require a single cycle to finish their computation.

In addition to the bitwise logic operators, there are also function units for shifting bits left or right. These units are particularity difficult to implement. Quartz does not have operators for it as bit shifts are equivalent to multiplications or divisions by powers of two. The problem with this approach is that the multiplication or division operator will be propagated down to the Verilog level by the Averest tools. Due to the way the Verilog compiler applies constant propagation, it is unable to detect that the multiplication

or division can be implemented by using a shift register. Hence, a hardware multiplier will be used in case of a left shift. In case of a right shift the hardware synthesis will fail, since there are no hardware dividers available on a FPGA. It is possible to work around this problem using a nested combination of the *bv2nat* and *nat2bv* functions.

```
1  event bv{BusDataWidth * 2} tmp;
2
3  if(bv2nat(op2) < BusDataWidth - 1) {
4      if(bv2nat(op2) == 0) {
5          tmp = op1 @ {false :: BusDataWidth};
6      }
7      else {
8          for(i = 1..BusDataWidth - 1) {
9              if(bv2nat(op2) == i) {
10                 tmp = nat2bv(bv2nat(op1) * bv2nat(true @ {false :: i}),
11                                 BusDataWidth * 2);
12             }
13         }
14     }
15 }
16 else {
17     tmp = {false :: BusDataWidth * 2};
18 }
19
20 next(result) = tmp{BusDataWidth - 1 : 0};
```

Listing 7: Synthesizeable left shift

Listing 7 shows the synthesizable version of the shift left operation. The first if statement in line 3 checks whether a shift is needed at all. If the shift amount is larger than the width of the output, the output is simply set to zero. The next if statement in line 4 ensures that an input shifted by 0 is the input itself. This statement can not be moved inside the for loop since the expression $false :: 0$ may result in the syntactically invalid value $0b$ in the Verilog code, which is only the prefix for binary value without any following digits. This is issue discussed in more details later in Section 8.3. Since the for loop in line 8 is unrolled by the compiler, the combination of the loop and the if statement in line 9 can be seen a multiplexer. The multiplexer uses *op2* to select one of its $BusDataWidth - 2$ inputs and its output is connected to the *tmp* bit vector. Line 10 connects each multiplexer input *i* to the expression $op1 \cdot 2^i$. The 2^i values have to be expressed using bit vector concatenation, since this results in constant bit vectors in the Verilog source code. Otherwise, an expression will be generated in their place, resulting in the Verilog compiler not recognizing the code as bit shift. Line 20 finally truncates the *tmp* bit vector to the correct length. The same principle can also be applied for the right shift operation, using division instead of multiplication.

In theory the same result could be achieved, using only bit vector concatenation. The problem with this approach is that it results in the usage of the *BvOfBoolTup* function in the AIF files, which can not be translated into Verilog by the *aif2verilog* tool. This problem is also illustrated with a source code example in Section 8.3.

The *Arithmetic* package contains a set of basic arithmetic operations on integers along with a comparator and the fully featured *Alu* function unit. All function units in this package support unsigned arithmetic as well as signed arithmetic using two complement numbers, expect for the Divider unit, which exists in two separate versions.

Function Unit	Ports	Operation
Adder	$op_1, op_2, result, status$	$result = op_1 + op_2$
Subtractor	$op_1, op_2, result, status$	$result = op_1 - op_2$
Multiplier	$op_1, op_2, result, status$	$result = op_1 * op_2$
UnsignedDivider	$op_1, op_2, result, remainder, status$	$result = op_1/op_2$
		$remainder = op_1 \bmod op_2$
SignedDivider	$op_1, op_2, result, remainder, status$	$result = op_1/op_2$
		$remainder = op_1 \bmod op_2$
Comparator	$op_1, op_2, result$	depends on the offset of op1
Alu	$op_1, op_2, result1, result2, status$	depends on the offset of op2

Table 3: Function units of the arithmetics package

Table 3 provides an overview of all function units available in the *Arithmetic* package. For the simple binary operations the op_1 and op_2 ports have to be connected to *InputSockets*. The *Comparator* and *Alu* units require a *MultiAddressInputSocket* for op_1, as they offer more than just one operation. The *result* and *status* ports can be connected to the buses using an *OutputSocket*.

For convenience reasons *Adder*, *Subtractor* and *Multiplier* units use the same schema for the *status* value. Bit 0 of the status value indicates whether the result of the operation was zero, removing the need for an additional zero check for some programs, for example when implementing loops with a counter the conditional jump can use the status value directly. The bit 1 of the status indicates an over or underflow that occurred during the calculation.

While a multiplier can be easily implemented by applying the multiplication operator to two bit vectors in the Quartz source code, the implementation of a hardware synthesizable divider is more complex. The reason for this is that FPGA hardware usually contains hardware multiplier blocks, being able to perform multiplications of 8 bit or 16 bit numbers using only combinatorial logic. These multiplier blocks can also be chained to allow multiplications of larger numbers. If the Verilog compiler encounters a multiplication operator, multiplier blocks are synthesized accordingly. In contrast to that, dividers can not be implemented efficiently using only combinatorial logic. Therefore, Verilog compilers usually refuse to synthesize code containing a division and the programmer has to manually implement the divider.

```
1  mem bv{BusDataWidth * 2} denominator;
2  mem bv{BusDataWidth * 2} partialRemainder;
3  event bv{BusDataWidth * 2} tmp;
4  mem bv{BusDataWidth} quotient;
5
6  event bv{1} divByZero;
7  event bv{1} zero;
8
9  if(op2 != {0b0 :: BusDataWidth}) {
10     quotient = {0b0 :: BusDataWidth};
11     //denominator = op2 << BusDataWidth
12     denominator = op2 @ {0b0 :: BusDataWidth};
13     partialRemainder = {0b0 :: BusDataWidth} @ op1;
14
15     for(bit = 0..BusDataWidth-1) {
16         //tmp = partialRemainder << 1
17         tmp = nat2bv(bv2nat(partialRemainder) * 2, BusDataWidth * 2);
```

```
18          if(bv2int(tmp) - bv2int(denominator) >= 0) {
19              //next(quotient) = quotient | (1 << (BusDataWidth - bit));
20              next(quotient) = quotient | {0b0 :: bit}
21                                          @ 0b1
22                                          @ {0b0 :: (BusDataWidth - bit) - 1};
23              //next(partialRemainder) = tmp - denominator
24              next(partialRemainder) = nat2bv(bv2nat(tmp)
25                                     - bv2nat(denominator), BusDataWidth * 2);
26          }
27          else {
28              next(partialRemainder) = tmp;
29          }
30          pause;
31      }
32
33      next(quotientResult) = quotient;
34      if(quotient == {0b0 :: BusDataWidth}) {
35          zero = 0b1;
36      }
37      //remainderResult = partialRemainder >> BusDataWidth
38      next(remainderResult) =
39                      partialRemainder{BusDataWidth * 2 - 1 : BusDataWidth};
40  }
41  else {
42      next(quotientResult) = {0b0 :: BusDataWidth};
43      next(remainderResult) = {0b0 :: BusDataWidth};
44      divByZero = 0b1;
45  }
46
47  next(status) = {0b0 :: BusDataWidth - 2} @ divByZero @ zero;
48
49  pause;
```

Listing 8: Synthesizeable divider

Listing 8 shows a simple non-restoring divider for unsigned integers implemented in Quartz. The dividers are the only function units with a latency larger than a single cycle. A division is done in $BusDataWidth + 1$ cycles, making the dividers also the only units with a latency depending on the width of their operands. The *SingedDivider* is implemented by wrapping the *UnsignedDivider* module with additional logic to handle the signs. In contrast to the units introduced previously the dividers have two results ports. The first one contains the result of the division, whereas the second one contains the remainder. This allows performing modulo operations as well.

Since integer division algorithms rely heavily on bit shifting, this implementation has to work around similar issues than the bit shifter function units discussed earlier. Luckily in this case some shift amounts are constant, which allows expressing them as multiplication. An example for this can be found on line 17 of the listing. The bit shift implemented in line 20 to 22 has the form $(1 << x)$, which is simply a way to express 2^x without multiplication. Initially this was implemented using Quartz' built-in *pow* function. Sadly the Verilog source code generated for this version is slightly more complex than the source code generated for the version shown above. The Verilog compiler used several multipliers to synthesize this version, which lead to the current formulation using bit vector concatenation. It is also worth noting that line 20 does not result in incorrect Verilog code, even though the *bit* variable may be 0, creating a bit

vector of zero length, which turned out to be problematic for the shifter function units.

The comparator module can be used for comparing signed and unsigned integers. For convenience the comparator offers the full set of comparisons: less, less or equal, equal, greater or equal and greater.

Address offset	Operation
0	Equal
1	Unsigned less
2	Unsigned less or equal
3	Less (two complement)
4	Less or equal (two complement)
5	Unsigned bigger
6	Unsigned bigger or equal
7	Bigger (two complement)
8	Bigger or equal (two complement)

Table 4: Address offsets for the op1 port of the Comparator function unit

The *op1* port has to be connected to the buses using a *MultiAddressInputSocket*. This socket maps nine addresses to the ports, one for each possible operation. Table 4 shows how the address offset provided by the sockets is mapped to the different operations.

The *Alu* function unit offers the combined functionality of all units in the arithmetic and bitwise logic package. Internally the unit instantiates the other function units and dispatches the inputs and outputs accordingly. Table 5 shows the mapping for the address offset of the *op1* port for the *Alu* FU.

Address offset	Operation
0	Add
1	Subtract
2	Multiply
3	Unsigned divide
4	Signed divide
5	shift left
6	shift right
7	Bitwise not
8	Bitwise and
9	Bitwise or
10	Bitwise xor

Address offset	Operation
11	Equal
12	Unsigned less
13	Unsigned less or equal
14	Less (two complement)
15	Less or equal (two complement)
16	Unsigned bigger
17	Unsigned bigger or equal
18	Bigger (two complement)
19	Bigger or equal (two complement)

Table 5: Address offsets for Alu

The last group of function units is not organized into a separate Quartz package. Instead, the Quartz modules for these units can be found directly inside the *FunctionUnits* package. These modules include a memory interface, a register file and units for general purpose input and output.

Table 6 is an overview of these function units. The *Input* function unit has an external port which is connected through the Quartz module hierarchy to the top level Quartz module by the TTA generator tool. An *InputSocket* is used to connect the mask port of function unit to the buses. If the *mask* port is written, the unit is triggered and a bitwise conjunction of the *mask* and the input value are transferred

Function Unit	Ports	External ports	Operation
Input	$mask, result$	$extInput$	$result = extInput \wedge mask$
Ouput	$value$	$extOuput$	$extOutput = value$
RegisterFile	$value$		
Ram	$address, value$	$extAddress, extReadValue,$ $extWriteEnable, extWriteValue$	Depends on address offset

Table 6: IO and memory interface function units

into the *result* port. The *result* port in term is connected to the buses using a *OutputSocket*. This allows to read bit patterns from external peripherals.

The *Output* module can be used to send bit patterns to external peripherals. It has a single port named *value*, which once written sets the external port *extOutput* to the written data. An *InputSocket* has to be used to connect the *value* port to the bus interconnect.

The *RegisterFile* function unit provides 32 general purpose registers with the same width as the bus data lines. A *ParallelMultiAddressSocket* needs to ne used to connect the *value* port of the unit to the buses. The read and write offsets, supplied by the socket, are used to index the register being accessed. From a programmer's perspective the register file appears as 32 bus addresses, which save a value of the data lines if used as target and restore it if used as source.

The *Ram* function unit provides a basic interface to an external RAM. Its signature can be found in Listing 6. It exposes four external ports at the top level module. Its interface has been chosen to match the interface of block RAM commonly available on FPGA hardware. Internally it is accessible via two ports. The first port should be connected to a *MultiAddressInputSocket*, mapping two addresses to the port. The first address can be used to trigger a read, while the second one can be used to trigger a write. In case of a read the *extAddress* port is set to the value of the *address* port. The data at this RAM address is the expected to be available at the *extReadValue* port in the next cycle, from where it is copied to the *value* port. If a write is triggered the address is also written to the *extAddress* port, the value from the *value* port is copied to the *extWriteValue* and the *extWriteEnable* is set to true. The write to the RAM becomes effective in the next processor cycle. Most FPGA block RAMs support dual port interfaces allowing two concurrent accesses to the same memory. This can be leveraged by adding two *Ram* function units to the architecture and connecting one to each RAM port. Using this scheme, it is possible to perform two concurrent RAM accesses within the same processor cycle.

5 TTA Generator Tool

The following section describes the tool, which was developed to generate transport triggered architectures from architecture descriptions and configuration files. The tool was developed in FSharp using the *fsharpc* compiler and the mono framework under Linux. It uses Quartz code templates and a library of function units derived from the Quartz based TTA discussed in the previous sections. Similar to the TTA implementation, the primary design goals for the generator tool have been extensibility and modularity. Hence, an object oriented design has been chosen for the tool. Functional code is used where it results in shorter code and better readability, for example filtering lists.

The generation is done in three phases, which have been implemented each in separate FSharp modules. In the first phase, the architecture description file (ADF) is checked for format errors and the architecture description is read from it. In the next phase, an architecture is generated from the description, required modules are located, bus addresses are assigned and sanity checks on the architecture description are performed. The last phase, is the code generation. In this phase the Quartz code for the architecture is generated using the library of function units and a small set of Quartz code templates.

The FSharp modules for the first two phases have been designed with as few dependencies on a particular output language as possible. This allows exchanging the code generator module, if code generation in another language becomes necessary.

5.1 File Formats

The TTA generator tool reads data from various input files and writes several output files. Most of them are Quartz source code templates, explained in Section 5.2. The other files are XML files, which will be discussed in this Section.

```
 1  <?xml version="1.0"?>
 2  <adf>
 3    <bus name="B1">
 4      <width>8</width>
 5    </bus>
 6    <bus name="B2">
 7      <width>8</width>
 8    </bus>
 9    <bus name="B3">
10      <width>8</width>
11    </bus>
12    <bus name="B4">
13      <width>8</width>
14    </bus>
15
16    <socket name="RegisterSocket">
17      <connects-to>
18        <bus>B1</bus>
19        <bus>B2</bus>
20        <bus>B3</bus>
21        <bus>B4</bus>
22      </connects-to>
23    </socket>
24
25    <socket name="Adder1Operand1">
26      <connects-to>
27        <bus>B1</bus>
28        <bus>B2</bus>
29        <bus>B3</bus>
30        <bus>B4</bus>
31      </connects-to>
32    </socket>
33
34    <socket name="Adder1Operand2">
35      <connects-to>
36        <bus>B1</bus>
37        <bus>B2</bus>
38        <bus>B3</bus>
39        <bus>B4</bus>
40      </connects-to>
41    </socket>
42
43    <socket name="Adder1Status">
44      <connects-to>
45        <bus>B1</bus>
46        <bus>B2</bus>
47        <bus>B3</bus>
48        <bus>B4</bus>
```

```
49        </connects-to>
50      </socket>
51
52      <socket name="Adder1Result">
53        <connects-to>
54          <bus>B1</bus>
55          <bus>B2</bus>
56          <bus>B3</bus>
57          <bus>B4</bus>
58        </connects-to>
59      </socket>
60
61      <function-unit
62          name="Registers">
63        <module>
64          RegisterFile
65        </module>
66        <port name="value">
67           <connects-to>
68             RegisterSocket
69          </connects-to>
70        </port>
71      </function-unit>
72
73
74
75      <function-unit name="Adder1">
76        <module>
77          Arithmetic.Adder
78        </module>
79        <port name="op1">
80          <connects-to>
81            Adder1Operand1
82          </connects-to>
83        </port>
84        <port name="op2">
85          <connects-to>
86            Adder1Operand2
87          </connects-to>
88        </port>
89        <port name="status">
90          <connects-to>
91            Adder1Status
92          </connects-to>
93        </port>
94        <port name="result">
95          <connects-to>
96            Adder1Result
97          </connects-to>
98        </port>
99      </function-unit>
100
101   </adf>
```

Listing 9: Example for an ADF

The first input file read by the tool, is the architecture description file (ADF). The format for this file is loosely based on the ADF format [6] developed for the TCE project. An example for an ADF can be found in listing 9. The ADF in the listing describes a basic TTA containing only a *RegisterFile* FU and an *Adder* FU. An ADF typically starts by declaring a set of buses, which can be found in the lines 3 to 14 in the example. Each *bus* element has *name* attribute and *width* element. The *name* attribute is an identifier, used to refer to the bus in socket declarations. Since architectures with different bus widths are not supported yet, the *width* value should be the same for all declared buses.

The next step, after the buses have been declared, is to declare the sockets needed for connecting the function units to the buses. In the example listing, a *socket* element can be found from line 52 to 59. Similar to the *bus* elements each *socket* element has a *name* and a *connects-to* element. The *connects-to* element contains a list of *bus* elements, specifying the connection of the socket to the bus. Since currently only fully interconnected architectures are supported, every socket should have connections to all declared buses.

After that, function units can be instantiated. Lines 75 to 99 show a *function unit* element for an *Adder* FU. Again it has a *name* attribute, acting as a label for the FU instance. The first element inside the *function unit* is a *module* element. A *module* element's values is the dotted quartz module path, which locates the FUs Quartz module in the library. The *module* element is followed by a list of *port* elements. The *name* attribute of each port element should match one of the ports of the FU. Each *port* element has *connects-to* element containing the name of the socket the port is connected to.

```
1   <?xml version="1.0"?>
```

```
2   <FunctionUnitInterface>
3       <Ports>
4           <Port trigger="true">op1</Port>
5           <Port>op2</Port>
6           <Port>result</Port>
7           <Port>status</Port>
8       </Ports>
9       <Sockets>
10          <InputSocket port="op1">
11              <AddressOffset>0</AddressOffset>
12          </InputSocket>
13          <InputSocket port="op2">
14              <AddressOffset>1</AddressOffset>
15          </InputSocket>
16          <OutputSocket port="result">
17              <AddressOffset>2</AddressOffset>
18          </OutputSocket>
19          <OutputSocket port="status">
20              <AddressOffset>3</AddressOffset>
21          </OutputSocket>
22      </Sockets>
23  </FunctionUnitInterface>
```

Listing 10: FUDF for the Adder FU

The information in the ADF alone are not sufficient to generate the Quartz source code for the TTA. The socket declarations lack the type of the socket to be used for connecting a port to the buses and an indication, whether a socket can trigger the FU. While this could be included in the ADF as well, doing so would result in a lot of redundant information inside the ADF. Therefore, this additional data is stored in function unit description files (FUDF). Each function unit has its own FUDF, stored along with its Quartz module inside the library directory. Listing 10 shows the FUDF for the *Adder* FU. The first part of the listing from line 3 to line 8 is a list of all ports. These elements must be in the same order than the port parameters of the Quartz module for this FU. The trigger port, the *op1* port in this example, in line 4, is marked by setting its *trigger* attribute to true. The *Ports* element is followed by a *Sockets* element. Inside the *Sockets* element is a *Socket*, *InputSocket*, *OutputSocket*, *MultiAddressInputSocket* or *ParallelMultiAddressSocket* element for each *Port* element. Each socket element is assigned to a *Port* element via its *port* attribute and contains an *AddressOffset* element for each bus address the port will handle. In case of sockets that handle multiple addresses, the *AddressOffset* element can have a *name* attribute, assigning a human readable name to the operation triggered at the offset. Using this information the generator tool is able to generate instances for the function units quartz module including its sockets and for allocating the bus addresses. In addition to the port specifications for the function unit the FUDF also contains a list of dependencies. The reason for this is that some FUs' quartz modules internally use instances of the modules for other FUs. For example the *Alu* FU uses all other arithmetic and logic FUs internally. The generator tool needs a list of the dependencies in order to create the Quartz modules for these internally used FUs. The tool does not have a mechanism for finding transitive dependencies yet. Hence, the dependency list has to list all dependencies of dependency modules as well.

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <BusAddressSpace>
3     <FunctionUnit name="ControlUnit">
4       <Bus name="B1">
```

```
5        <Address port="ImmediateLoadSource">0</Address>
6        <Address port="ProgrammCounter">1</Address>
7      </Bus>
8      ...
9      <Bus name="B4">
10       <Address port="ImmediateLoadSource">0</Address>
11       <Address port="ProgrammCounter">1</Address>
12     </Bus>
13   </FunctionUnit>
14   <FunctionUnit name="Adder1">
15     <Bus name="B1">
16       <Address port="op1">2</Address>
17       <Address port="op2">3</Address>
18       <Address port="result">4</Address>
19       <Address port="status">5</Address>
20     </Bus>
21     ...
22   </FunctionUnit>
23 </BusAddressSpace>
```

Listing 11: Shortened BASF for the example ADF

The bus address allocation has to be saved to a machine readable output file as well, since it will be needed by the compiler to generate machine code for the architecture. Listing 11 is the bus address space file (BASF) for the example ADF from listing 9. It contains a *FunctionUnit* element for each FU in the architecture. The *name* attribute of those elements is taken from the ADF. Inside the *FunctionUnit* elements there is one *Bus* element for each bus the FU is connected to. Again the *name* attributes for those *Bus* elements correspond with the names assigned in the ADF. The *Bus* elements contain *Address* elements for each address, that is allocated for the FU. The *port* attribute of these elements contains the name assigned to the port in the FUDF, which is connected to the bus using a socket at this address. In case of sockets with multiple addresses a dot and the name attribute of the *AddressOffset* element is appended to the port name, for example *address.write*.

5.2 Quartz Template Library

To generate the Quartz source code for a transport triggered architecture the generator tool makes use of library of a Quartz templates. Quartz templates are regular Quartz source code files, containing placeholders, which are replaced by configuration values, module instances or complete code blocks in the code generation phase of the tool.

Templates are stored inside the template directory using the same conventions for paths as Quartz does. For example the Quartz module for the *Adder* FU has the dotted path *FunctionUnits.Arithmetic.Adder*. Its template therefore has to be located in the file *QuartzTemplates/FunctionUnits/Arithmetic/Adder.qrz*.

There are four types of Quartz templates. Most of the templates are Quartz modules implementing function units, sockets and the control unit.

```
1 package FunctionUnits.Arithmetic;
2
3 <%ConfigurationMacros%>
4
5 macro nat2BusData(number) = nat2bv(number, BusDataWidth);
6 macro int2BusData(number) = int2bv(number, BusDataWidth);
```

```
7
8   module Adder(mem bv{BusDataWidth} ?op1,
9                mem bv{BusDataWidth} ?op2,
10               mem bv{BusDataWidth} !result,
11               mem bv{BusDataWidth} !status,
12               event bool ?trigger) {
13      loop {
14          event bv{1} zero;
15          event bv{BusDataWidth + 1} tmp;
16
17          immediate await(trigger);
18          tmp = nat2bv(bv2nat(op1) + bv2nat(op2), BusDataWidth + 1);
19          next(result) = tmp{BusDataWidth-1:0};
20
21          if(tmp{BusDataWidth-1:0} == {false :: BusDataWidth}) {
22              zero = 0b1;
23          }
24
25          next(status) = {false :: BusDataWidth - 2}
26                         @ tmp{BusDataWidth:BusDataWidth} @ zero;
27
28          pause;
29      }
30  }
```

Listing 12: Quartz template for the Adder FU

Listing 12 shows the template for the *Adder* function unit. Templates for function units only contain a single placeholder. Placeholders have the form <*%Identifier%*>. For example, the placeholder in line 3 has the identifier *ConfigurationMacros*. It will be replaced by three macro definitions for *BusAddressWidth*, *BusAddressWidth* and *BusCount*. The same applies to the bus control unit and the sockets as well.

In addition to these modules, there are also modules for simulated program memory, simulated ram and a test module, which can be generated by inserting the *ConfigurationMacros*. The first two modules provide a Quartz implementation for program memory and RAM, while the third can serve as starting point for writing test case for the architecture. The resulting file is a regular quartz source file. These templates are primarily a workaround to the problem that Quartz does not have an own preprocessor yet. It is therefore not possible to place the macro definitions in a central file and include this file where needed. Instead, the macros have to be duplicated for each source file.

As explained earlier in Section 4 the FU quartz modules are connected to their sockets inside composed modules, which are completely generated by the tool. To generate the composed modules, the tool uses a template for the boilerplate code.

```
1   package Composed;
2
3   import Sockets.*;
4
5   import <%BaseModule%>;
6
7   <%ConfigurationMacros%>
8
```

```
 9  macro nat2BusAddress(number) = nat2bv(number, BusAddressWidth);
10
11  module <%ModuleName%>(event [BusCount]bv{BusAddressWidth} busSource,
12                   event [BusCount]bv{BusAddressWidth} busTarget,
13                   event [BusCount]bv{BusDataWidth} busData<%ExternalPorts%>) {
14  <%VariableDeclarations%>
15
16  <%SocketInstances%>
17      ||
18  <%ModuleInstance%>
19  }
```

Listing 13: Quartz template for composed module

As shown in Listing 13 the template for composed modules makes heavy use of placeholders. The first placeholder in line 5 is the to be replaced with the dotted path to Quartz module for the FU, used inside the composed module. For the *Adder* used as example earlier, this placeholder would be replaced with *FunctionUnits.Arithmetic.Adder*. Line 7 contains the *ConfigurationMacros* placeholder, which is replaced by the bus configuration macros, similar to the function unit templates. The *ModuleName* placeholder in line 11 will be changed to the name assigned to the FU in the ADF file. As discussed previously in Section 4.4 FUs might have external ports, which have to be forwarded to the top level module of the processor. This is achieved by replacing the *ExternalPorts* placeholder in 13 with a list of parameter for the external ports. For a *Ram* FU the placeholder would for example be replaced with: `mem bv{BusDataWidth} !extAddress, mem bv{BusDataWidth} ?extReadValue, event bool !extWriteEnable, mem bv{BusDataWidth} !extWriteValue`.
The next placeholder in line 14 is called *VariableDelcarions* and is replaced by a set of declarations for local variables. These local variable are used to connect the FU module instance to the socket instances inside the composed module. The placeholders *SocketInstances* and *ModuleInstances* in line 16 and line 18 are replaced with instance of the sockets required in the composed module and an instance of the Quartz module containing the actual function unit. The socket instances and the function unit instances are executed in parallel by the ||-operator. An example for this can be found in Listing 14, which is a complete composed module for an *Adder* FU, that was generated using the ADF from listing 9.

```
 1  package Composed;
 2  import Sockets.*;
 3  import FunctionUnits.Arithmetic.Adder;
 4
 5  macro BusCount = 4;
 6  macro BusDataWidth = 8;
 7  macro BusAddressWidth = 8;
 8
 9  macro nat2BusAddress(number) = nat2bv(number, BusAddressWidth);
10
11  module Adder1(event [BusCount]bv{BusAddressWidth} busSource,
12                   event [BusCount]bv{BusAddressWidth} busTarget,
13                   event [BusCount]bv{BusDataWidth} busData) {
14
15      mem bv{BusDataWidth} op1Register;
16      event bool Adder1Op1Trigger;
17      mem bv{BusDataWidth} op2Register;
18      event bool Adder1Op2Trigger;
```

```
19 ||     mem bv{BusDataWidth} result1Register;
20 ||     mem bv{BusDataWidth} statusRegister;
21 ||
22 ||     Adder1Op1 : InputSocket(busSource, busTarget, busData,
23 ||                       nat2BusAddress(2), Adder1Op1Trigger, op1Register);
24 ||     ||
25 ||     Adder1Op2 : InputSocket(busSource, busTarget, busData,
26 ||                       nat2BusAddress(3), Adder1Op2Trigger, op2Register);
27 ||     ||
28 ||     Adder1Result1 : OutputSocket(busSource, busTarget, busData,
29 ||                       nat2BusAddress(4), result1Register);
30 ||     ||
31 ||     Adder1Status : OutputSocket(busSource, busTarget, busData,
32 ||                       nat2BusAddress(5), statusRegister);
33 ||     ||
34 ||     Adder1 : Adder1(op1Register, op2Register, result1Register,
35 ||                       statusRegister, Adder1Op1Trigger);
36 || }
```

Listing 14: Generated Quartz source for a composed Adder

The next template needed in the code generation process, is the template for the composed cpu, which connects the composed modules with a control unit and acts as the top level quartz module for the processor. It is similar to the composed module template shown in Listing 13. Again the *ConfigurationMacros* placeholder is used to work around the missing preprocessor. The composed cpu modules, signature also contains an *ExternalPorts* placeholder. In contrast to the composed module this placeholder is replaced with a list containing the external ports of all function units. To avoid name conflicts in this list, the parameters for the external ports are each prefixed with the name assigned to the corresponding FU in the ADF. For example a Ram FU named *RamA* has an external port *extAddress* which will be renamed to *RamAextAddress* in the composed cpu's signature. In addition to that, the template only contains one other placeholder called *ComposedModules*, which is replaced by a list of instances for the composed modules. For the *Adder* example the line creating the instance looks like this: `Adder1 : Adder1(busSource, busTarget, busData);`. The module instances in the list are composed to run in parallel using the ||-operator. Additionally, the composed cpu also contains a control unit instance, which is run in parallel with all composed modules.

The last template required by the generator tool is the template for the *VerilogWrapper* module. As explained in Section 4 the Quartz compiler generates invalid Verilog code, if the top level module of a system contains an array of bit vectors. Therefore, the top level module of the TTA has to be wrapped inside another module, replacing the array of bit vectors with one large bit vector. This is implemented by the *VerilogWrapper* module. It used the same placeholders as the composed cpu module to forward the external ports from the composed cpu to the outside world. There placeholders are needed, since the only module instance inside the *VerilogWrapper* is the composed cpu, which only exposes instructions, a program counter and the external ports as inputs and outputs.

5.3 Inputfile Processing

Configuration files for the generator tool, are supplied in a XML format. One of the advantages of using XML is that libraries for parsing XML based file formats are available for most programming languages. In case of FSharp the *XmlSerializer* class [12] provided by the .Net framework can be used. It allows

serializing and deserializing instances of specially annotated FSharp classes to and from XML files. Internally the introspection capabilities of the .Net framework are used to generate object instances from classes assigned to XML elements.

```
open System.Collections.Generic
open System.Xml.Serialization

type BusElement = class
    [<XmlAttribute("name")>]
    val mutable name : string

    [<XmlElement("width")>]
    val mutable width : int
    new() = {
        name = "UnknownBus"
        width = 0
    }
    override this.ToString() =
        sprintf "BusElement<%s,%d>" this.name this.width
end

[<XmlRoot(ElementName="adf")>]
type ArchitectureElement = class
    [<XmlElement("bus")>]
    val mutable buses : List<BusElement>
    (* ... *)
    new() = {
        buses = new List<BusElement>()
    }
    (*...*)
    member this.getBus(name : string) : BusElement =
        this.getEntry(name,this.buses)

    member this.hasBus(name : string) : bool =
        this.hasEntry(name,this.buses)
end
```

Listing 15: Shortened *ArchitectureDescription* FSharp module

Listing 15 is a shortened version of the *ArchitectureDescription* FSharp module. This module contains the FSharp classes used for deserializing the XML inside the ADF. Each class is annotated to indicate which XML element should be deserialized into an instance of this class. The *XMLRoot* annotation in line 18 specifies the class to use for deserializing the root element of the input XML. Line 20 contains an *XMLElement* annotation making the .Net framework deserialize any *bus* elements, which are children of the root *adf* element, into the *buses* list declared in line 21. The *BusElement* class used for this element has only two members *name* and *width*. Due to the annotations in line 5 and line 8 the value of the *name* attribute for each *bus* element is written to the *name* member of the *BusElement* instance, while the content of the single *with* child element is parsed as an integer and stored in the *width* member. The classes inside the *FunctionUnitDescription* and *BusAddressSpace* modules follow a similar structure. In all three modules the classes do not have any methods, which actively modifying their members or contain any logic, other than accessors to allow faster access to list elements, like the *getBus*

and *hasBus* methods in line 27 and 30. This separation between data structures and the actual algorithms makes it possible to exchange the data formats, without interfering with the actual logic of the generator tool.

The first step of the TTA generation is to read the ADF supplied for the architecture. Afterwards, the data structure obtained by deserializing the XML is passed to the *Architecture* module which generates an output language independent representation of the architecture. This second phase is discussed in detail in the following Section.

5.4 Architecture Generation

The second stage of generating TTA is the architecture generation stage. This phase is performed by the *Architecture* FSharp module. It results in an output language independent representation of the TTA.

The first step of this phase is to extract the bus configuration from the architecture description. A bus configuration consists of three values, which correspond to the three macros used in the Quartz templates for functional units. The *BusCount* value is simply the number of buses found inside the architecture. Since the architecture does not support buses of different widths, the *BusDataWidth* and *BusAddress-Width* are set to the same value as the *width* attribute of the buses in the architecture description. If one of the buses uses a different width, the tool will stop with an error. These three values are stored in the configuration dictionary, using the identifiers of the macros as key. The configuration dictionary is passed on to any further stages of the tool.

As a next step a *FunctionUnit* object is created for each function unit specified for the architecture. The constructor of the *FunctionUnit* class requires three arguments: the name of the FU, a full architecture description data structure and the configuration dictionary. After ensuring that a function unit for the name exists in the architecture description, the constructor tries to load the FUDF for the FU from the template directory. Using the information from the FUDF, it checks whether all the ports specified in the architecture description match a port in the function unit description. It also verifies that the trigger port of the FU is connected to a socket. This is important since not all ports of a FU have to be exposed to buses via a socket. In case the trigger port is not accessible from the buses, the FU can never perform any computation. An architecture description containing such a FU is therefore considered malformed and will result in error if presented to the tool.

Next the constructor creates a socket object for each socket connected to a port of the FU. The different types of sockets are represented each by separate classes derived from the *GenericSocket* class. A socket object has three main purposes. The first one is to supply all list of local variables needed to connect the socket to a function units port. In addition to that, it also needs to provide a list of parameters and a module name to create an instance of the Quartz module for this socket. Afterwards, a socket object will need to generate a list addresses managed by the socket, which will be used to computed a bus addresses to be allocated for the function unit.

As mentioned earlier, a function units port does not necessarily have to be exposed to the buses. Specifying a socket in the ADF can be omitted for any port except the trigger port. Still, the locale variables that would otherwise have been used to connect a port to its socket must be created inside the composed module, since otherwise it would not be possible to create an instance of the Quartz module implementing the FU. To implement this, each socket object has a dummy flag. If the dummy flag of socket object is set, the *Synthesis* module only generates the local variables for the socket, which can be used as parameters to a function unit's Quartz module inside the composed module. The constructor of the *FunctionUnit* class automatically creates dummy sockets for any port specified in the function unit description without a socket connected to it in the architecture description.

The constructors of the *MultiAddressInputSocket* and *ParallelMultiAddressSocket* may also perform an update to the configuration dictionary. In addition to the bus configuration, the dictionary also contains a macro name *MaxOffset*. This macro is set to maximum number of addresses manage by a single socket. Using this macro allows the offset signal of these sockets to have lower width than the bus addresses, which is a basic optimization for saving hardware resources.

The final list of socket objects for the FU is sorted with respect to the order of the ports inside the function unit description. Since this order is the same as the order of the port parameters for the FU's Quartz module, this order simplifies the creation of the parameter list for the instance of this module inside the FUs composed module.

After sorting the list of socket objects for the function unit, the constructor continues extracting the external ports of the unit from the function unit description.

The newly created *FunctionUnit* objects are stored in a list. The final step of the architecture generation is to allocate bus address space for each socket in the architecture. This is done by iterating over the list of all *FunctionUnit* objects and calling the *enumerateAddresses* method of each object. The method takes the smallest unused bus address as a parameter and returns a dictionary. The keys of the dictionary are bus addresses, while the values are human readable names for the addresses, which will be used later in the BASF as explained earlier in Section 5.1. Internally this dictionary is created by invoking the *enumerateAddresses* method of each socket object of the *FunctionUnit* object. Similar to the method on the function unit level the method on the socket level takes the smallest unallocated bus address as its parameter and uses it as a base address for the socket. The returned dictionary is merged into a dictionary of the same format containing all allocated bus addresses. The smallest unallocated address for the next *enumerateAddresses* can be determined easily by incrementing the biggest key of this dictionary.

Finally, the list of *FunctionUnit* objects, the bus address space dictionary and the configuration dictionary are passed to the code generation phase.

5.5 Quartz Code Generation

The final phase of the transport triggered architecture generation is the code generation. This phase is handled primarily by the *Synthesis* FSharp module. The module uses the information collected by the *Architecture* module in the previous phase to generate the Quartz source code for the TTA.

Before the actual Quartz code is produced, the bus address space dictionary is written to two files. The first file is in CSV format. Each line contains a bus address and the name generated for it in the previous phase separated by a tab character. This file is primarily intended as reference for debugging and manually writing machine code for the architecture. The second file is in the BASF XML format explained in Section 5.1. It can be used by the compiler to look up the bus addresses for the ports of any function unit defined in the architecture description. Since the file format already supports other architectures than the simple fully interconnected type, it contains a separate list of bus addresses for each port and bus, making it less convenient to use this file to manually look up addresses.

The first step to create the Quartz source for the processor is to collect a set of all required Quartz modules for the function units. The module names are stored in this set as dotted paths, the same notation as used in the Quartz import statement. This set contains all modules that can be generated from the templates by replacing the *ConfigurationMacros* placeholder with the appropriate set of macro declarations. Each *FunctionUnit* object in the list generated in the previous phase has as *getDependencies* method. It takes the set as a parameter and adds the module implementing the FU, all required socket modules and all the dependencies from the unit's FUDF to it. Independent of the modules added for the function unit, the set will always contain the dotted paths to four special modules. The first one is

the control unit module, which is of course required for any TTA processor regardless of the function units. The other three modules are the simulated program memory, the simulated ram and a basic test case module.

Once the module set is completed a *QuartzModule* object is created for each module. The *QuartzModule* constructor requires only two arguments, the dotted path to the module and the configuration dictionary. Each *QuartzModule* exposes a single public method called *synthesizeModuleSource*. Since Quartz templates are stored using same directory structure than the Quartz modules in the generated TTA, the method can use the dotted path to look up the Quartz template. It will then read the template, generate the configuration macros from the configuration dictionary and replace the *ConfigurationMacros* placeholder with them. The resulting Quartz source code is written to the output folder, which is passed to the method as argument. The directory structure inside the output directory is the same as inside the template directory. If a required directory does not exist, it will be created by the method.

The newly created *QuartzModule* instances are stored inside a module list.

As a next step a *ComposedModule* instance is created for each *FunctionUnit* object passed from the *Architecture* module. The *ComposedModule* class is responsible for creating the composed modules, introduced in Section 4, combining function units and their sockets. Similar to the *QuartzModule* constructor, the *ComposedModule* takes two arguments. The first one is the *FunctionUnit* object, the second one is the configuration dictionary. Since the *ComposedModule* class is derived from the *QuartzModule* class, it also exposes a *synthesizeModuleSource* method with the same signature than the method of the *QuartzModule*. The method internally resorts to the same helper functions, than the methode in its parent class, to generate to configuration macros for the composed module. Unlike method in the *ComposedModule* class, this method does not need to look up a Quartz template using dotted path. Instead, it loads the composed module template shown in listing 13. It then replaces the *ConfigurationMacros* placeholder with the generated macros. Additionally, *BaseModule* is substituted with a dotted path the Quartz module implementing the FU.

Then the method uses the *externalPorts* member of the *FunctionUnit* object to obtain a list of parameters for the external ports of the FU. As explained earlier, the parameters in this list are already sorted to match the signature of the Quartz module implementing the FU. Using this list the method produces a string, containing a parameters list for the external ports, which is appended to the composed module's signature as replacement for the *ExternalPorts* placeholder.

Next the *FunctionUnit* objects *generateVariable* method is used to retrieve a list of local variables needed in order to connect the function unit module to its sockets. A variable declaration in Quartz is produced for each variable in this list. These variable declarations are inserted into the template as a substitute for the *VariableDeclaration* placeholder. The declarations also include any variables produced by the dummy sockets, discussed in the previous section.

With the variable declarations in place, the socket instances for the composed module can be generated. The *FunctionUnit* objects *getSocket* method is used to obtain a list of all sockets objects for the FU. A line of Quartz code, creating an instance of the Quartz module for the particular socket, is produced for each socket. In case of an *Adder* the line for the socket connected to the *op1* port may look similar to this:

```
Adder1Op1 : InputSocket(busSource, busTarget, busData, nat2BusAddress(2),
Adder1Op1Trigger, op1Register);
```

Finally a string is composed containing all produced source code lines, separated by the ||-operator. This results in a set of socket module instances which are executed in parallel. An example for the result can be found in listing 14 from line 22 to line 32. The resulting string is supplied to the Quartz template to substitute the *SocketInstances* placeholder.

Having generated all required local variables and socket instances, only the *ModuleInstance* placeholder remains to be filled in. This placeholder is meant to be substituted by a line of Quartz code, that creates an instance of the module implementing the function unit for the composed module. The parameter list for the instance can be produced by appending the parameters for the FU's external ports, to the list of parameters for the modules regular ports. Combining this parameters list with the name of the Quartz module for the FU yields a source code line similar to those generated for the sockets. Lines 34 and 35 in listing 14 are an example for the result. For a module that has external ports the result may be more complex. Using Ram FU for example requires the following line:

```
RamA : Ram(addressRegister, addressOffset, valueRegister, RamAAddressTrigger,
RamAextAddress, RamAextReadValue, RamAextWriteEnable, RamAextWriteValue);
```

Once the *ModuleInstance* is replaced with resulting line, the template has been fully transformed into a regular Quartz module and can be written to the output directory. Composed modules are stored inside a separate Quartz package named *Composed*. The names, assigned to functional units inside the architecture description, are used as module names for the composed modules. So the module for an *Adder* shown in listing 14 would be saved to *Composed/Adder1.qrz* inside the output directory. Again missing sub directories in the output directory are created automatically during the process.

Since the *ComposedModule* class is derived from the *QuartzModule* class, the newly created *ComposedModule* objects can be added to same module list as the *QuartzModule* objects before.

The last required step in the code generation process is to add a *ComposedCpuModule* object to the module list. The *ComposedCpuModule* is responsible for producing the composed cpu module. Its constructor takes 2 arguments. The first one is the full list of function units collected by the Architecture module described in the previous Section. Similar to the other two classes for generating Quartz modules the second argument is the configuration dictionary.

The *synthesizeModuleSource* method of the *ComposedCpuModule* class works similar to the method of the *ComposedModule* class. It uses a special template for generating the composed cpu module. As all templates, the template for this module has a *ConfigurationMacros* placeholder, which is replaced by the macros generated from the configuration dictionary. It also has an *Eternal Ports* placeholder in its module declaration. This placeholder is replaced by a parameter list containing all external ports of all function units. To avoid name collisions the naming conventions explained in chapter 5.2 have to be applied. Therefore, all parameter names for the external ports are prefixed with the name of their FUs.

In addition to that, it contains a placeholder named *ComposedModules*. To fill in this placeholder a Quartz code line is created for each FU. Each line creates an instance of the composed module for the FU, using the bus signals and the external port parameters of the composed module as its parameters. Similar to the lines for the socket instances inside the composed modules, these lines are joined together using the ||-operator. The resulting block of Quartz code is executed in parallel with the control unit instance. The newly created *ComposedCpuModule* is added to module list as well.

In addition to the *QuartzModule* objects, the *ComposedModule* objects and the *ComposedCpuModule* object a *VerilogWrapperModule* object is required to finish the code generation. The *VerilogWrapperModule* object is constructed similar to *ComposedCpuModule* object. It is used to add the external ports of the composed cpu to the signature of the *VerilogWrapper* module inside its template. It also creates a source code line to create an instance of the composed cpu module inside the wrapper module. Since the *VerilogWrapperModule* class also is derived from the *QuartzModule* class, the *VerilogWrapperModule* object can also be added to module list.

In a final step the *synthesizeModuleSource* on each object inside the module list is called. This creates the necessary directory structure inside the output directory, reads the necessary templates, substitutes their placeholders and writes the Quartz modules to their correct location. As a result the output directory

contains a Quartz package for function units, a package for sockets, a package for composed modules, a test package containing the ram and program memory modules intended for simulation, a composed cpu module and the *VerilogWrapper* module which can be used to generate Verilog code.

6 Test Scenarios

Throughout the development of the Quartz-based TTA implementation and the processor generator tool, two scenarios have been used for testing.

Both scenarios are based on the same architecture. The full ADF for the Architecture used can be found in Section 10.1. It uses 4 buses with 8 bit data and address width. The architecture is fully interconnected, since the generator does not support any other interconnections. The function units for this architecture have been chosen to provide similar computation capabilities to a small 8 bit processor.

During experiments with this architecture a bug in the *aif2verilog* tool has been discovered. If a bus width larger than 8 bits is used, the generation of the Verilog code fails. Simulation using the *aif2trc* utility worked as expected. Consequently, the test scenarios have been restricted to 8 bit wide buses.

The two main function units are a *RegisterFile* unit providing 32 general purpose registers and complete *Alu* unit for performing arithmetic and logic operations. An *Output* function unit has been added to provide a parallel 8 bit wide output port to the processor. In addition to that, two *Ram* function units can be used to access memory. This allows using dual port memory to perform two memory accesses in the same cycle. It also uses separate memories for code and data. This was chosen since as explained before it simplifies running the processor on a FPGA, and it enables to use a dual port memory block for storing data. The address space allocation for this architecture can be found in 10.2.

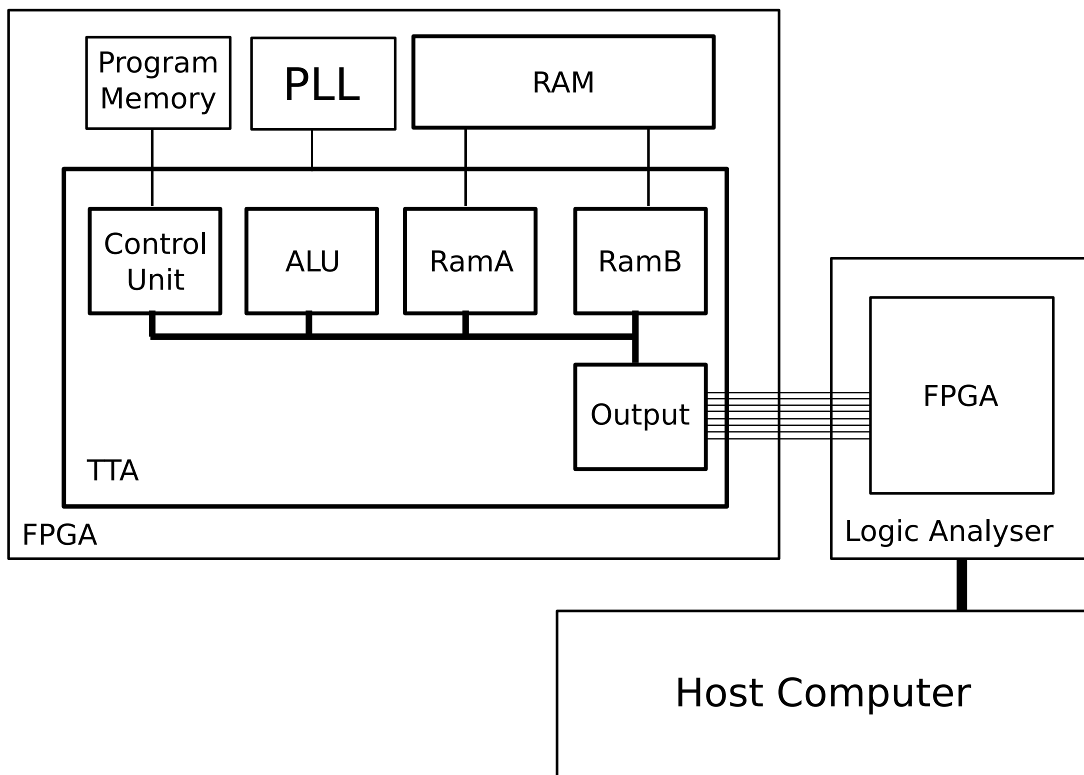

Figure 4: Block diagram of the FPGA setup

The architecture has been tested using a simulation with the *aif2trc* utility as well as on real FPGA hardware. The hardware that has been used in these tests is a *Papilio one 500k* FPGA open hardware development board [4], which uses a *Xilinx Spartan XC3S500* FPGA. It was chosen as a target platform

mainly because of its extensive documentation and because it features an on board *JTAG* programmer, eliminating the need for a separate programmer. The programmer also works under Linux with an open source programming tool for the board. A PLL unit on the FPGA is used to generate a 16 MHz clock for the TTA processor. The machine code is stored inside 96 bits wide and 8 bits deep ROM. A 8 bit wide slice of block RAM with an address width of 8 bits in a dual port configuration is used as memory for data. The RAM is connected to the external ports of the Ram function units of the architecture. Using Xilinx IP blocks for the PLL, RAM and ROM blocks only small amount of VHDL code was needed to connect the processors top level Verilog module to the hardware blocks. The VHDL source for this can be found in Section 10.3. To generate the bit stream from the FPGA, the free version of the *Xilinx ISE Webpack* has been used. Figure 4 shows an overview of this test setup.

The 8 bit parallel output of the Output function unit has been connected to output pins of the FPGA board. A logic analyser can be connected to these pins to capture the programs output while testing. The *Open Bench Logic Sniffer* [3], an open source logic analyser based on a *Xilinx Spartan XC3S250* FPGA has been used for that purpose. The Logic Sniffer is connected to a computer via USB, where the *Logic Sniffer Client* [1] tool can be used to perform captures and analyse the signals. In order to produce the traces shown later in this section, the tool had to be modified to interpret 8 parallel data lines as integer and ASCII data.

For the simulation, the *SimulatedProgramMemory* and the *SimulatedRam* Quartz modules have been used instead of the RAM and ROM blocks of the FPGA. The output of the *aif2trc* tool has been filtered by a set of python scripts in order to create a trace that is easily comparable with the traces generated by the logic analyser software.

The two test scenarios presented in the following chapters have both been run on the hardware test setup and as simulation.

6.1 Hello World

The first test scenario is a TTA version of the canonical hello world program. It uses the first *Ram* function unit to write the string ”Hello World” to the first 10 memory addresses. The other *Ram* FU is used to read out the same addresses and write the result to the 8 bit output.

```
1  .BusCount 4
2  .BusDataWidth 8
3
4  loop:
5  #Write "Hello World" to Ram A via while reading it via B
6  LOAD 0  35  LOAD 72  36  NOP 0 0    NOP 0 0
7  LOAD 1  35  LOAD 101 36  LOAD 0  37  NOP 0 0
8  LOAD 2  35  LOAD 108 36  LOAD 1  37  MOVE 39 40
9  LOAD 3  35  LOAD 108 36  LOAD 2  37  MOVE 39 40
10 LOAD 4  35  LOAD 111 36  LOAD 3  37  MOVE 39 40
11 LOAD 5  35  LOAD 32  36  LOAD 4  37  MOVE 39 40
12 LOAD 6  35  LOAD 87  36  LOAD 5  37  MOVE 39 40
13 LOAD 7  35  LOAD 111 36  LOAD 6  37  MOVE 39 40
14 LOAD 8  35  LOAD 114 36  LOAD 7  37  MOVE 39 40
15 LOAD 9  35  LOAD 108 36  LOAD 8  37  MOVE 39 40
16 LOAD 10 35  LOAD 100 36  LOAD 9  37  MOVE 39 40
17 NOP 0 0     NOP 0 0      LOAD 10 37  MOVE 39 40
18 NOP 0 0     NOP 0 0      NOP 0 0     MOVE 39 40
```

```
19 || JMP 0 loop   NOP 0 0      NOP 0 0      LOAD 0 40
```

Listing 16: Hello world test

Listing 16 shows a simple assembler like representation of the program. Each column of the source code contains the instructions for one bus. A point can be used to set bus configuration values and an identifier followed by colon can be used as label for jump instructions. A simple FSharp program has been used to convert the assembler listing into a binary file containing the machine code as well as in a Quartz source code representation. This way, changes to the test programs could be made without having to adjust the binary for the FPGA version and the Quartz code for the simulated version of the test separately.

The first bus is used to write the addresses 0 to 10 to the port at bus address 35, which is the write address port of the first *Ram* FU. The second bus is used to load the ASCII data into the value port of the same *Ram* FU, in the same cycle, while the third bus writes the same addresses to bus address 37 with a one cycle delay relative to the first bus. Bus address 37 is the read address port of the second *Ram* function unit. The read data can retrieved one cycle later at the bus address 39. This is done using the fourth bus. It moves the data from address 39 to address 40 with one cycle delay relative to the third bus. Address 40 is the value port of the *Output* function unit. Setting the data of the value port becomes effective at the beginning of the next cycle. This demonstrates the ability of the architecture to make efficient use of dual port memory.

NOPs have been added to align the 4 instruction streams correctly. Line 19 contains an unconditional jump back to the beginning of the program as well as a load of zero to the output. Setting the output to zero creates a simple start marker, which can be used to trigger the logic analyser. Also, it prevents the output from being set to 'd' for more than one cycle, resulting in a better looking trace.

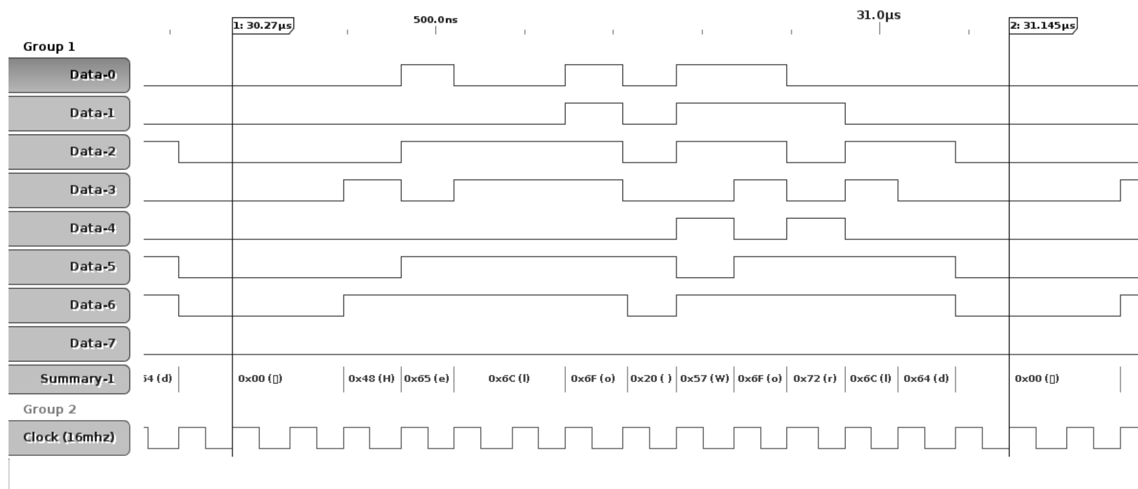

Figure 5: Logic analyser capture for the hello world program

Figure 5 shows the logic analyser capture for this program. The data lines 0 to 7 have been directly connected to the 8 lines of the *Output* function unit. The Summary-1 signal added by the Logic Sniffer Client tool shows the ASCII data being sent out one byte each cycle.

The simulation of the architecture done by the *aif2trc* tool, using a test Quartz module and the RAM and program memory modules, results in exactly the same trace. The Quartz module used for the simulation can be found in Section 10.4.

6.2 Fibonacci

While the first test scenario is intended to test memory interfaces and basic interfaces with hardware blocks on the FPGA, the second test scenario focuses on testing the *Alu*, the register file and the control units jump instruction. A non-recursive program for generating the Fibonacci sequence from 1 to 233 has been chosen for this task.

```
1  .BusCount 4
2  .BusDataWidth 8
3
4  init:
5  #Reg[0] = 1, Reg[1] = 1
6  LOAD 1 2        LOAD 1 3    NOP 0 0     NOP 0 0
7  loop:
8  #Alu.OP1_Add = Reg[0], Alu.OP2 = Reg[1], Output = Reg[1]
9  MOVE 2 41       MOVE 3 61   MOVE 3 40   NOP 0 0
10 #Reg[0] = Reg[1], Reg[1] = Alu.Result
11 MOVE 3 2        MOVE 62 3   NOP 0 0     NOP 0 0
12 # if not overflow goto loop
13 JMPZ 64 loop    NOP 0 0     NOP 0 0     NOP 0 0
14 JMP 0 init      NOP 0 0     NOP 0 0     NOP 0 0
```

Listing 17: Fibonacci test

Listing 17 shows the assembler program used in this test. In line 6 the register 0 and 1 of the register file are initialized to 1. The content of register 1 is copied to the *Output* FU's value port by the third bus in line 9. The first two buses are used to copy the contents of the registers to the *Alu* to compute an addition in the same cycle. Its result can be retrieved at bus address 62 in the next cycle. In line 10, the first bus moves the content of register 1 to register 0. In the third cycle the second bus moves the result of the addition into register 1. There is no data race between these two bus transactions since writes to the register file do not become effective until the next cycle. In line 12, the status port of the *Alu* at bus address 64 is used as condition for a conditional jump back to line 8. If the addition caused an overflow, the overflow bit in the status register would have been set. If this is not the case, the execution continues at line 8. Otherwise, the jump instruction is ignored and the execution continues at line 13. Line 13 is an unconditional jump back to the beginning of the program where the registers have been initialized.

Figure 6 shows the logic analyser capture for this program. The Fibonacci number from 1 to 144 in hexadecimal notation can be found in the Summary-1 signal, which is added by the Logic Sniffer Client as parallel interpretation of the Data-0 to Data-7 signals.

Running the program in a simulated architecture using the *aif2trc* tool results in exactly the same trace for the output data lines. The Quartz module used for the simulation can be found in Section 10.5.

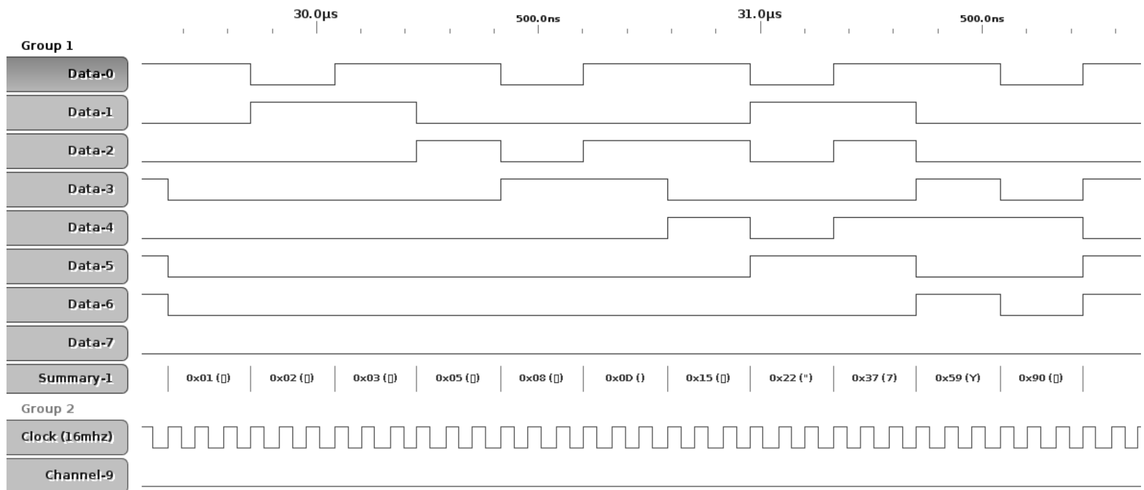

Figure 6: Logic analyser capture for the Fibonacci program

7 Summary & Conclusion

The TTA implementation developed for this thesis has been proven to work well for the small test scenarios presented previously. The test scenarios explained earlier show that architectures simulated using the Averest framework exhibit the same behaviour, as the corresponding implementations on the FPGA test setup. Due to its modular approach that relies heavily on reusing code, the architecture implementation is very compact and can be extended with very little effort, making it an excellent platform for developing custom processors based on transport triggered architectures.

The work done for this thesis also shows that hardware for an architecture can be synthesized easily from its Quartz source codes. Even though a few workarounds were needed, the test on real hardware also worked as expected. The performance of the tested architectures was better than initially expected. Clock speeds of up to 16 Mhz can be achieved without any further optimizations. The overall performance is more than sufficient for the further applications of transport triggered architectures in a research context.

The architecture generator tool provides a simple way to generate a transport triggered architecture from configuration files, enabling to quickly produce and compare different architecture configurations. The tool also already features basic sanity checks to prevent architectures containing errors from being created.

Using Quartz as hardware description language for TTA implementation has greatly simplified the development process, compared to traditional hardware description languages. Due to the simulation and testing capabilities of the Averest framework, it was possible to develop the hardware architecture using a test driven approach. Writing code that was synthesizable to hardware required some workarounds, but only minor changes to the Averest tools should be necessary to resolve these issues. Any bugs or limitations of the framework encountered during the work on this thesis have therefore been documented in the next Section as future work.

8 Future Work

This Section provides a summary of limitations and problems encountered during the work on this thesis, as a starting point for future research. It also contains suggestions for features, which could be added in future versions of the architecture implementation and the generator tool. Furthermore, it documents bugs in the Averest framework, which need to be addressed in order to removed the need for the various workarounds required for the TTA implementation.

8.1 Architecture Implementation

The major limitation of the current implementation is the lack of support for partially interconnected architectures. This restriction is primarily imposed by the current implementation of buses and sockets. Since buses are implemented as three arrays of bit vectors, it is not possible to hand over only a subset of the buses to a socket.

A possible solution to this problem is to implement each bus as a separate set of three parameters to the socket, instead of grouping them into an array. This would not only result into a different parameter list for every socket, it would also require generating the code for sockets, since it would no longer be possible to use a for loop to iterate over buses inside the socket modules.

A more practical approach is to introduce a new parameter to the socket modules. Similar to the constant address parameter for the sockets, it is possible to pass a constant array of booleans to each socket. The array should have one element for each bus in the architecture. An element set to true is used to indicate, that the socket is connected to the corresponding bus and that the socket should handle reads and writes for this bus. Likewise, an element set to false indicates that the socket should ignore transactions on the bus. Since the array can be a constant inside the composed modules and the for loops inside sockets modules are rolled out by the compiler, the additional selection logic can be optimized out compile time. Using this approach, it should therefore be possible to realize other interconnection schemes without a negative impact on the architecture's performance.

A second limitation of the current implementation can be observed, when an architecture is synthesized to a bit stream for FPGAs. The current addressing decoding scheme relies heavily on calculating offsets. A ParallelMultiAddressSocket for example requires two subtractors to calculate the read and write offsets it provides to its function unit. This does not only use a significant amount of dedicated hardware subtractors on the FPGA, it also reduces the overall clock speed of the system, as it increases the length of the critical signal path.

The architecture used for the test scenarios presented in Section 6 has a maximum clock speed of 18 MHz according to the timing analysis tools provided by the ISE Webpack. For comparison same analysis has been applied to the Verilog source code generated only for single function units instead of the complete hardware. The Alu, which is the most complex function unit implemented, has a maximum clock speed of about 64 MHz.

This result shows that the performance could be significantly improved, if the subtractors were removed from the address decoding and therefore from the critical path. An approach to achieve this is to use a different addressing scheme. Instead of a base address, an address prefix can be assigned to each socket handling more than one bus address. For example, a socket handling 4 addresses, which is attached to an 8 bit wide bus, can have a 6 bit wide prefix. If the first 6 bits of the bus address match the sockets prefix, the remaining two bits are used as an offset and forwarded to the function unit. On the one hand, this decoding scheme completely removes the need of subtractors for address decoding, as it allows decoding addresses using only bit vector comparisons. On the other hand, it uses the address

space less efficiently. While this scheme works well for sockets, where the amount of handled addresses is a power of 2, it creates unusable addresses in other cases. Architectures intended for hardware synthesis will usually only be able to use a small portion of the bus address space, since the hardware resources for creating function units will become a limiting factor before the size of the address space. Therefore, it might be beneficial to switch to this addressing scheme.

In addition to these two limitations, the Quartz source code can be significantly simplified as soon as some bugs in the Averest framework have been fixed. As discussed in previous Sections, the TTA implementation contains various workarounds to keep the architecture synthesizable. These workarounds should be replaced with simpler formulations once these issues are resolved.

8.2 Generator Tool

Both improvements suggested in the previous chapter require changes to the TTA generator tool as well.

If the Quartz TTA supports other architectures than the fully interconnected one currently implemented, the generator has to be changed to allow other interconnection schemes as well. This can be achieved by modifying the *FunctionUnit* class in the Architecture module to forward additional information about the buses to the socket objects created by this class. Of course socket classes have to extended as well to process this information and to supply the results to the Synthesis module. In the code generator stage primarily the code generating the composed Quartz modules has to be altered. Depending on the pattern used for implementing the interconnection, it may also be necessary to add logic to generate Quartz source code for the socket modules.

The alternative addressing scheme proposed previously also requires a few changes to the generator tool. To generate alternative address assignment, the methods for allocating the addresses in the socket classes of the *Architecture* module have to modified. Instead of the current linear allocation a more intelligent system has to be implemented to chose the correct prefix length and find free blocks inside the address space.

In addition to these features for supporting architectures of higher complexity, the generator tool could also be improved from a usability point of view. Currently the tool only performs rudimentary sanity checks on its input files and throws rather minimalistic errors, if a problem is detected. Since the probability of introducing errors increases with the complexity of the input files, a better error reporting mechanism may be useful for generating larger architectures. Furthermore, advanced checks could be introduced to warn if a architecture description provided to the tool results in an inefficient use of hardware resources and bus address space.

8.3 Averest

During the work on the TTA implementation several bugs in the Averest framework have been found. This Section aims to document these bugs, so they can be addressed in future version of the framework. The issues explained in this Section have been found in the version 2.2.4.5 of the framework and may have been resolved already in later releases.

The first issue arose at a rather early stage of development, while trying to produce Verilog code for the control unit Quartz module.

```
1
2 module BitVectorArray(event [4]bv{8} bytes) {
3     pause;
```

```
4 }
```

Listing 18: Quartz code using an array as module inout

Listing 18 show a minimal Quartz module used to reproduce the problem. Its input is an array of four 8 bit wide bit vectors, which is similar to the instruction input of the control unit Quartz module. Running *qrz2aif* and *aif2verilog* with this source code results in the Verilog code containing the module declaration presented in listing 19.

```
1 ...
2
3 /* define the module */
4 module BitVectorArray(
5     input  wire  clock,
6     output wire          [7:0] /* btv */ bytes[3 : 0]
7 );
8
9 ...
```

Listing 19: Verilog code for listing 18

Line 6 of listing 19 contains array of four 8 bit wide bit vectors, which is defined as an output of the Verilog module. Unfortunately the Verilog syntax does not allow arrays as module inputs or outputs. Details on this restriction can be found in Section 12.2.3 of the Verilog 2005 standard document [5]. The commonly recommended solution to this issue is to flatten the out the array into a single bit vector. For the TTA implementation, the flattening is done inside the Verilog wrapper module, which has been introduced in Section 4. The *aif2verilog* tool should not silently generate syntactically invalid Verilog for this kind of modules, instead it should either terminate with an appropriate error message, or automatically produce code to flatten out the array so a single bit vector can be used as the module's input or output. With respect to the TTA implementation the latter solution is preferable, as it eliminates the need for a wrapper module.

```
1 module ZeroLengthBitVectorBug(event bv{8} input,
2                                event bv{8} !output) {
3     loop {
4         output = input @ {false :: 0};
5
6         pause;
7     }
8 }
```

Listing 20: Quartz module using a zero length bit vector

The Quartz source code in listing 20 demonstrates a similar bug. Likewise to the issue presented previously, this module results in a syntactically invalid Verilog being generated. This problem was initially found while implementing the bit shift operations used by the arithmetic function units.

```
1 /*
2  *  evaluate immediate actions
3  */
4 assign input__1 = 8'b00000000;
5 assign output__1 = (__init000 ?
6                          {input__1,0'b} :
7                          (__el1000 ?
```

```
8                                   {input__1,0'b} :
9                                   8'b00000000
10                              )
11                          );
```

Listing 21: Excerpt from the Verilog code for listing 20

Listing 21 contains an excerpt from the Verilog code generated for this module. The syntax errors can be found in line 6 and line 8. A zero bit wide constant bit vector results in the generation of *0'b* as value in the Verilog source. As *0'b* is only the prefix of a bit vector provided in binary notation, it has to be followed by sequence of digits. An example for its correct use can be found in line 9 of the listing. This issue should rather be considered a bug of the *qrz2aif* tool instead of *aif2verilog*. As the concatenation of a zero bit wide bit vector can neither change the length nor the value of the bit vector it is concatenated to, it can safely be optimized out, while generating the AIF for the module.

```
1  module BitVectorConcatBug(event bool input1,
2                            event bool input2,
3                            event bv{8} !output) {
4      loop {
5          output = input1 @ input2 @ {false :: 6};
6
7          pause;
8      }
9  }
```

Listing 22: Quartz module concatenating bit vectors and booleans

During the implementation of the arithmetic function units another problem with *aif2verilog* has been encountered. Listing 22 shows a small Quartz module that illustrates the issue. In line 5 the two booleans input1 and input2 are concatenated to a 6 bit wide constant bit vector. A similar construct is required for the status registers used by the arithmetic function units. The AIF file generated for this module uses the *BtvOfBoolTup* function. If the *aif2verilog* utility is invoked for this AIF file the Verilog code generation stops with an error message stating that no Verilog can be generated for *BtvOfBoolTup*. As a workaround, bit vectors containing only a single bit can be used, instead of booleans. Unfortunately such bit vectors can not be assigned directly from boolean expressions. An extra if statement is required for those cases. Again fixing this issue would eliminate the need for various workarounds in the TTA implementation, resulting in a more compact and readable code base.

The test scenarios introduced in Section 6 also revealed a new problem with the *aif2verilog* tool. Architectures generated by the TTA generator tool, which use a higher width than 8 bits for data or bus addresses cause the *aif2verilog* utility to fail. While the simulation of these architectures using *aif2trc* works as expected, while running *aif2verilog* on the same AIF files fails with the error message simply stating : 'Value is too large'. Attempts to further narrow down this issue and to find a workaround have been unsuccessful. Resolving this issue is necessary to be able to generate more complex architectures and to use the full potential of the TTA implementation developed for this thesis.

Finally, as discussed in greater detail in Section 4.4, implementing the function units for shifting bits involved various workarounds to generate Verilog code, which is recognizable as multiplication or division with a constant power of 2, by the Verilog compiler. This was necessary to prevent the hardware synthesis from using hardware multipliers instead of shift registers. To simplify the implementation and to remove the need for these workarounds, a constant propagation feature in the *aif2verilog* tool might be of use. The expressions supplied to the multiplication operator in the Verilog source code, could have

been flattened to single constant value in most cases, enabling the Verilog compiler to recognize it as constant. Ideally this kind of optimization should not be necessary as the Verilog compiler should be able to do it as a part of the compilation process. Unfortunately this failed at least for the code developed for this thesis. Therefore, providing this kind of feature within the Averest tools, could greatly improve the performance of hardware synthesized from Quartz source code.

9 Bibliography

References

[1] *Logic Sniffer Client Homepage*. Available at http://ols.lxtreme.nl/.

[2] *Mono Framework Homepage*. Available at http://www.mono-project.com/.

[3] *Open Bench Logic Sniffer Hardware Specification*. Available at http://dangerousprototypes.com/docs/Open_Bench_Logic_Sniffer.

[4] *Papilio One 500k Hardware Specification*. Available at http://papilio.cc/index.php?n=Papilio.PapilioOne.

[5] (2006): *IEEE Standard for Verilog Hardware Description Language*. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001) , pp. 1–560.

[6] A Cilio, HJM Schot, JAAJ Janssen, P Jääskeläinen & L Laasonen (2003): *Architecture definition file: processor architecture definition file format for a new TTA design framework*. Project Document, Tampere Univ. of Tech., Tampere, Finland 2006.

[7] TU Delft Deparment of Computer Engineering: *MOVE Project Homepage*. Available at https://www.cs.tut.fi/~move/DelftMoveSite/MOVE/.

[8] FSharp Software Foundation: *FSharp Project Hompage*. Available at http://fsharp.org/.

[9] Jan Hoogerbrugge (1996): *Code generation for transport triggered architectures*. TU Delft, Delft University of Technology.

[10] Pekka Jääskeläinen (2010): *TCE project: Co-design of application-specific processors with LLVM-based compilation support*. Available at http://blog.llvm.org/2010/06/tce-project-co-design-of-application.html.

[11] Embedded Systems Group University of Kaiserslautern: Available at http://www.averest.org/.

[12] Microsoft Developer Network: *XmlSerializer Class Reference*. Available at https://msdn.microsoft.com/de-de/library/system.xml.serialization.xmlserializer%28v=vs.110%29.aspx.

[13] Tampere University of Technology Department of Pervasive Computing: *TTA-based Co-design Environment Project Homepage*. Available at http://tce.cs.tut.fi/.

[14] K. Schneider (2009): *The Synchronous Programming Language Quartz*. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany.

[15] Jaakko Sertamo (2003): *Processor Generator for Transport Triggered Architectures* .

10 Appendicies

10.1 Test Scenario ADF

```
1  <?xml version="1.0"?>
2  <adf>
3      <bus name="B1">
4          <width>8</width>
5      </bus>
6      <bus name="B2">
7          <width>8</width>
8      </bus>
9      <bus name="B3">
10         <width>8</width>
11     </bus>
12     <bus name="B4">
13         <width>8</width>
14     </bus>
15
16     <socket name="RegisterFileSocket">
17         <connects-to>
18             <bus>B1</bus>
19             <bus>B2</bus>
20             <bus>B3</bus>
21             <bus>B4</bus>
22         </connects-to>
23     </socket>
24
25     <socket name="RamAAddress">
26         <connects-to>
27             <bus>B1</bus>
28             <bus>B2</bus>
29             <bus>B3</bus>
30             <bus>B4</bus>
31         </connects-to>
32     </socket>
33
34     <socket name="RamAValue">
35         <connects-to>
36             <bus>B1</bus>
37             <bus>B2</bus>
38             <bus>B3</bus>
39             <bus>B4</bus>
40         </connects-to>
41     </socket>
42
43     <socket name="RamBAddress">
44         <connects-to>
45             <bus>B1</bus>
46             <bus>B2</bus>
47             <bus>B3</bus>
```

```
48              <bus>B4</bus>
49          </connects-to>
50      </socket>
51
52      <socket name="RamBValue">
53          <connects-to>
54              <bus>B1</bus>
55              <bus>B2</bus>
56              <bus>B3</bus>
57              <bus>B4</bus>
58          </connects-to>
59      </socket>
60
61      <socket name="OutputValue">
62          <connects-to>
63              <bus>B1</bus>
64              <bus>B2</bus>
65              <bus>B3</bus>
66              <bus>B4</bus>
67          </connects-to>
68      </socket>
69
70      <socket name="AluOp1">
71          <connects-to>
72              <bus>B1</bus>
73              <bus>B2</bus>
74              <bus>B3</bus>
75              <bus>B4</bus>
76          </connects-to>
77      </socket>
78
79      <socket name="AluOp2">
80          <connects-to>
81              <bus>B1</bus>
82              <bus>B2</bus>
83              <bus>B3</bus>
84              <bus>B4</bus>
85          </connects-to>
86      </socket>
87
88      <socket name="AluResult1">
89          <connects-to>
90              <bus>B1</bus>
91              <bus>B2</bus>
92              <bus>B3</bus>
93              <bus>B4</bus>
94          </connects-to>
95      </socket>
96
97      <socket name="AluResult2">
98          <connects-to>
99              <bus>B1</bus>
```

```
100                 <bus>B2</bus>
101                 <bus>B3</bus>
102                 <bus>B4</bus>
103             </connects-to>
104         </socket>
105
106         <socket name="AluStatus">
107             <connects-to>
108                 <bus>B1</bus>
109                 <bus>B2</bus>
110                 <bus>B3</bus>
111                 <bus>B4</bus>
112             </connects-to>
113         </socket>
114
115         <function-unit name="Registers">
116             <module>RegisterFile</module>
117             <port name="value">
118                  <connects-to>RegisterFileSocket</connects-to>
119             </port>
120         </function-unit>
121
122         <function-unit name="RamA">
123             <module>Ram</module>
124             <port name="value">
125                  <connects-to>RamAValue</connects-to>
126             </port>
127             <port name="address">
128                  <connects-to>RamAAddress</connects-to>
129             </port>
130         </function-unit>
131
132         <function-unit name="RamB">
133             <module>Ram</module>
134             <port name="value">
135                  <connects-to>RamBValue</connects-to>
136             </port>
137             <port name="address">
138                  <connects-to>RamBAddress</connects-to>
139             </port>
140         </function-unit>
141
142         <function-unit name="ParalellOutput">
143             <module>Output</module>
144             <port name="value">
145                  <connects-to>OutputValue</connects-to>
146             </port>
147         </function-unit>
148
149         <function-unit name="Alu">
150             <module>Arithmetic.Alu</module>
151             <port name="op1">
```

```
                <connects-to>AluOp1</connects-to>
            </port>
            <port name="op2">
                <connects-to>AluOp2</connects-to>
            </port>
            <port name="result1">
                <connects-to>AluResult1</connects-to>
            </port>
            <port name="result2">
                <connects-to>AluResult2</connects-to>
            </port>
            <port name="status">
                <connects-to>AluStatus</connects-to>
            </port>
    </function-unit>


</adf>
```

Listing 23: ADF for the test scenario architecture

10.2 Test Scenario Address Space

```
Address     Name
=====================================
0           ControlUnit.ImmediateLoadSource
1           ControlUnit.ProgrammCounter
2           Registers.register0
3           Registers.register1
4           Registers.register2
5           Registers.register3
6           Registers.register4
7           Registers.register5
8           Registers.register6
9           Registers.register7
10          Registers.register8
11          Registers.register9
12          Registers.register10
13          Registers.register11
14          Registers.register12
15          Registers.register13
16          Registers.register14
17          Registers.register15
18          Registers.register16
19          Registers.register17
20          Registers.register18
21          Registers.register19
22          Registers.register20
23          Registers.register21
24          Registers.register22
25          Registers.register23
26          Registers.register24
27          Registers.register25
```

```
28            Registers.register26
29            Registers.register27
30            Registers.register28
31            Registers.register29
32            Registers.register30
33            Registers.register31
34            RamA.read
35            RamA.write
36            RamA.value
37            RamB.read
38            RamB.write
39            RamB.value
40            ParalellOutput.value
41            Alu.add
42            Alu.substract
43            Alu.multiply
44            Alu.unsignedDivide
45            Alu.signedDivide
46            Alu.shiftLeft
47            Alu.shiftRight
48            Alu.not
49            Alu.and
50            Alu.or
51            Alu.xor
52            Alu.equal
53            Alu.unsignedLess
54            Alu.unsignedLessEqual
55            Alu.less
56            Alu.lessEqual
57            Alu.unsignedBigger
58            Alu.unsignedBiggerEqual
59            Alu.less
60            Alu.lessBigger
61            Alu.op2
62            Alu.result1
63            Alu.result2
64            Alu.status
```

Listing 24: Bus address space of the test scenario architecture

10.3 Test Scenario VHDL Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Core is
    Port ( --A : out  STD_LOGIC_VECTOR (15 downto 0);
           --B : out  STD_LOGIC_VECTOR (15 downto 0);
           C : out  STD_LOGIC_VECTOR (15 downto 0);
           clk : in STD_LOGIC);
end Core;

architecture Behavioral of Core is
```

```
12      signal instruction: STD_LOGIC_VECTOR (95 downto 0) := (others => '0');
13      signal programCounter: STD_LOGIC_VECTOR (7 downto 0);
14
15      signal clock16: std_logic;
16
17      signal ramAAddress: STD_LOGIC_VECTOR (7 downto 0);
18      signal ramAReadValue: STD_LOGIC_VECTOR (7 downto 0);
19      signal ramAWriteEnable: STD_LOGIC;
20      signal ramAWriteEnable_vec: STD_LOGIC_VECTOR (0 downto 0);
21      signal ramAWriteValue: STD_LOGIC_VECTOR (7 downto 0);
22
23      signal ramBAddress: STD_LOGIC_VECTOR (7 downto 0);
24      signal ramBReadValue: STD_LOGIC_VECTOR (7 downto 0);
25      signal ramBWriteEnable: STD_LOGIC;
26      signal ramBWriteEnable_vec: STD_LOGIC_VECTOR (0 downto 0);
27      signal ramBWriteValue: STD_LOGIC_VECTOR (7 downto 0);
28
29      signal output: STD_LOGIC_VECTOR (7 downto 0);
30
31      component VerilogWrapper port (clock : in STD_LOGIC;
32              longInstruction : in STD_LOGIC_VECTOR (95 downto 0);
33              programCounter : out STD_LOGIC_VECTOR (7 downto 0);
34              RamAextAddress : out STD_LOGIC_VECTOR (7 downto 0);
35              RamAextReadValue : in STD_LOGIC_VECTOR (7 downto 0);
36              RamAextWriteEnable : out STD_LOGIC;
37              RamAextWriteValue : out STD_LOGIC_VECTOR (7 downto 0);
38              RamBextAddress : out STD_LOGIC_VECTOR (7 downto 0);
39              RamBextReadValue : in STD_LOGIC_VECTOR (7 downto 0);
40              RamBextWriteEnable :  out STD_LOGIC;
41              RamBextWriteValue : out STD_LOGIC_VECTOR (7 downto 0);
42              ParalellOutputextOutput : out STD_LOGIC_VECTOR(7 downto 0));
43      end component;
44
45      component ClockDiv
46      port (CLKIN_IN : in STD_LOGIC;
47              CLKDV_OUT : out STD_LOGIC;
48              CLK0_OUT : out STD_LOGIC);
49      end component;
50
51
52      component ProgramMemory port ( clka : in STD_LOGIC;
53                          addra : in STD_LOGIC_VECTOR (7 downto 0);
54                          douta : out STD_LOGIC_VECTOR (95 downto 0));
55      end component;
56
57      component Ram port(
58                      clka : in STD_LOGIC;
59                      wea : STD_LOGIC_VECTOR(0 DOWNTO 0);
60                      addra : in STD_LOGIC_VECTOR(7 DOWNTO 0);
61                      dina : in STD_LOGIC_VECTOR(7 DOWNTO 0);
62                      douta : out STD_LOGIC_VECTOR(7 DOWNTO 0);
63                      clkb : in STD_LOGIC;
```

```
64                      web : in STD_LOGIC_VECTOR(0 DOWNTO 0);
65                      addrb : in STD_LOGIC_VECTOR(7 DOWNTO 0);
66                      dinb : in STD_LOGIC_VECTOR(7 DOWNTO 0);
67                      doutb : out STD_LOGIC_VECTOR(7 DOWNTO 0));
68      end component;
69
70  begin
71
72      clockdiv_inst : ClockDiv port map(
73              CLKIN_IN => clk,
74              CLKDV_OUT => clock16);
75
76      verilog_core : VerilogWrapper port map(
77                              clock => clock16,
78                              longInstruction => instruction,
79                              programCounter => programCounter,
80                              RamAextAddress => ramAAddress,
81                              RamAextReadValue => ramAReadValue,
82                              RamAextWriteEnable => ramAWriteEnable,
83                              RamAextWriteValue => ramAWriteValue,
84                              RamBextAddress => ramBAddress,
85                              RamBextReadValue => ramBReadValue,
86                              RamBextWriteEnable => ramBWriteEnable,
87                              RamBextWriteValue => ramBWriteValue,
88                              ParalellOutputextOutput => output);
89
90      program_mem : ProgramMemory port map (
91                              clka => clock16,
92                              addra => programCounter,
93                              douta => instruction);
94
95      ram_inst : Ram port map (
96                      clka => clock16,
97                      wea => ramAWriteEnable_vec,
98                      addra => ramAAddress,
99                      dina => ramAWriteValue,
100                     douta => ramAReadValue,
101                     clkb => clock16,
102                     web => ramBWriteEnable_vec,
103                     addrb => ramBAddress,
104                     dinb => ramBWriteValue,
105                     doutb => ramBReadValue);
106
107     ramAWriteEnable_vec(0) <= ramAWriteEnable;
108     ramBWriteEnable_vec(0) <= ramBWriteEnable;
109
110     C(7 downto 0) <= output;
111     C(8) <= clock16;
112     C(15 downto 9) <= (others => '0');
113
```

```
114 end Behavioral;
```

Listing 25: Additional VHDL source for the test scenario architecture

10.4 Hello World Test Module

```
1  package Tests;
2
3  import Composed.CpuComposed;
4
5  /*
6   * Bus configuration
7   *
8   * Note: BusAddressWidth and BusDataWidth should not be less then 6.
9   * Otherwise tests might not work as intended.
10  */
11 macro BusCount = 4;
12 macro BusAddressWidth = 8;
13 macro BusDataWidth = 8;
14
15 /*
16  * Instructionset configuration
17  */
18 macro InstructionOpCodeWidth = 8;
19 macro BusInstructionWordWidth
20         = (InstructionOpCodeWidth + 2 * BusAddressWidth);
21 macro InstructionWordWidth
22         = BusCount * BusInstructionWordWidth;
23
24 /*
25  * Opcode macros
26  */
27 macro OpCodeNOP = 0b00000000;
28 macro OpCodeMOVE = 0b00000001;
29 macro OpCodeLOAD = 0b00000010;
30 macro OpCodeJMP = 0b00000011;
31
32 /*
33  * Predefined bus addresses
34  */
35 macro BusAddressDontCare = nat2bv(0, BusAddressWidth);
36 macro BusAddressProgramCounter = nat2bv(1, BusAddressWidth);
37 macro BusAddressAccumulator = nat2bv(2, BusAddressWidth);
38
39 /*
40  * Misc
41  */
42 macro ProgramLength = 256;
43 macro MemoryLenght = 256;
44
45
46 /*
47  * For Tests
```

```
48   */
49  macro nat2BusAddress(number) = nat2bv(number, BusAddressWidth);
50  macro nat2BusData(number) = nat2bv(number, BusDataWidth);
51
52  macro GenerateMOVE(source, target) = OpCodeMOVE
53                                       @ nat2bv(source, BusAddressWidth)
54                                       @ nat2bv(target, BusAddressWidth);
55
56  macro GenerateUnsignedLOAD(data, target) = OpCodeLOAD
57                                           @ nat2bv(data, BusAddressWidth)
58                                           @ nat2bv(target, BusAddressWidth);
59
60  macro GenerateLOAD(data, target) = OpCodeLOAD
61                                    @ int2bv(data, BusAddressWidth)
62                                    @ nat2bv(target, BusAddressWidth);
63
64  macro GenerateJMP(destination) = OpCodeJMP
65                                    @ nat2bv(0, BusAddressWidth)
66                                    @ nat2bv(destination, BusAddressWidth);
67
68  macro GenerateJMPZ(source, destination)
69                                = OpCodeJMP
70                                    @ nat2bv(source, BusAddressWidth)
71                                    @ nat2bv(destination, BusAddressWidth);
72
73  macro GenerateNOP = OpCodeNOP
74                      @  nat2bv(0, BusAddressWidth)
75                      @ nat2bv(0, BusAddressWidth);
76
77
78
79  module HelloWorldTest(mem bv{BusDataWidth} !output) {
80
81      mem [ProgramLength][BusCount]bv{BusInstructionWordWidth} programMemory;
82      event nat{ProgramLength} nextProgramCounter;
83      event [BusCount]bv{BusInstructionWordWidth} instruction;
84
85      mem [MemoryLenght]bv{BusDataWidth} memory;
86
87      mem bv{BusDataWidth} ramAAddress;
88      mem bv{BusDataWidth} ramAReadValue;
89      event bool ramAWriteEnable;
90      mem bv{BusDataWidth} ramAWriteValue;
91
92      mem bv{BusDataWidth} ramBAddress;
93      mem bv{BusDataWidth} ramBReadValue;
94      event bool ramBWriteEnable;
95      mem bv{BusDataWidth} ramBWriteValue;
96
97
98
99      ProgramMemory: SimulatedProgramMemory(nextProgramCounter,
```

```
100                                                 instruction,
101                                                 programMemory);
102     ||
103     SimulatedRamA : SimulatedRam(ramAAddress,
104                     ramAReadValue,
105                     ramAWriteEnable,
106                     ramAWriteValue,
107                     memory);
108     ||
109     SimulatedRamB : SimulatedRam(ramBAddress,
110                     ramBReadValue,
111                     ramBWriteEnable,
112                     ramBWriteValue,
113                     memory);
114
115     ||
116     CpuComposed: CpuComposed(instruction,
117                                 nextProgramCounter,
118                                 ramAAddress,
119                                 ramAReadValue,
120                                 ramAWriteEnable,
121                                 ramAWriteValue,
122                                 ramBAddress,
123                                 ramBReadValue,
124                                 ramBWriteEnable,
125                                 ramBWriteValue,
126                                 output);
127     ||
128     {
129         programMemory[0][0] = GenerateLOAD(0, 35);
130         programMemory[0][1] = GenerateLOAD(72, 36);
131         programMemory[0][2] = GenerateNOP;
132         programMemory[0][3] = GenerateNOP;
133         programMemory[1][0] = GenerateLOAD(1, 35);
134         programMemory[1][1] = GenerateLOAD(101, 36);
135         programMemory[1][2] = GenerateLOAD(0, 37);
136         programMemory[1][3] = GenerateNOP;
137         programMemory[2][0] = GenerateLOAD(2, 35);
138         programMemory[2][1] = GenerateLOAD(108, 36);
139         programMemory[2][2] = GenerateLOAD(1, 37);
140         programMemory[2][3] = GenerateMOVE(39, 40);
141         programMemory[3][0] = GenerateLOAD(3, 35);
142         programMemory[3][1] = GenerateLOAD(108, 36);
143         programMemory[3][2] = GenerateLOAD(2, 37);
144         programMemory[3][3] = GenerateMOVE(39, 40);
145         programMemory[4][0] = GenerateLOAD(4, 35);
146         programMemory[4][1] = GenerateLOAD(111, 36);
147         programMemory[4][2] = GenerateLOAD(3, 37);
148         programMemory[4][3] = GenerateMOVE(39, 40);
149         programMemory[5][0] = GenerateLOAD(5, 35);
150         programMemory[5][1] = GenerateLOAD(32, 36);
151         programMemory[5][2] = GenerateLOAD(4, 37);
```

```
152            programMemory[5][3] = GenerateMOVE(39, 40);
153            programMemory[6][0] = GenerateLOAD(6, 35);
154            programMemory[6][1] = GenerateLOAD(87, 36);
155            programMemory[6][2] = GenerateLOAD(5, 37);
156            programMemory[6][3] = GenerateMOVE(39, 40);
157            programMemory[7][0] = GenerateLOAD(7, 35);
158            programMemory[7][1] = GenerateLOAD(111, 36);
159            programMemory[7][2] = GenerateLOAD(6, 37);
160            programMemory[7][3] = GenerateMOVE(39, 40);
161            programMemory[8][0] = GenerateLOAD(8, 35);
162            programMemory[8][1] = GenerateLOAD(114, 36);
163            programMemory[8][2] = GenerateLOAD(7, 37);
164            programMemory[8][3] = GenerateMOVE(39, 40);
165            programMemory[9][0] = GenerateLOAD(9, 35);
166            programMemory[9][1] = GenerateLOAD(108, 36);
167            programMemory[9][2] = GenerateLOAD(8, 37);
168            programMemory[9][3] = GenerateMOVE(39, 40);
169            programMemory[10][0] = GenerateLOAD(10, 35);
170            programMemory[10][1] = GenerateLOAD(100, 36);
171            programMemory[10][2] = GenerateLOAD(9, 37);
172            programMemory[10][3] = GenerateMOVE(39, 40);
173            programMemory[11][0] = GenerateNOP;
174            programMemory[11][1] = GenerateNOP;
175            programMemory[11][2] = GenerateLOAD(10, 37);
176            programMemory[11][3] = GenerateMOVE(39, 40);
177            programMemory[12][0] = GenerateNOP;
178            programMemory[12][1] = GenerateNOP;
179            programMemory[12][2] = GenerateNOP;
180            programMemory[12][3] = GenerateMOVE(39, 40);
181            programMemory[13][0] = GenerateJMP(0);
182            programMemory[13][1] = GenerateNOP;
183            programMemory[13][2] = GenerateNOP;
184            programMemory[13][3] = GenerateLOAD(0, 40);
185        }
186    }
187
188    drivenby TestCase {
189        for(i = 0 .. 32) {
190            pause;
191        }
192    }
```

Listing 26: Hello World test module

10.5 Fibonacci Test Module

```
1    package Tests;
2
3    import Composed.CpuComposed;
4
5    /*
6     * Bus configuration
7     *
```

```
8    * Note: BusAddressWidth and BusDataWidth should not be less then 6.
9    * Otherwise tests might not work as intended.
10   */
11  macro BusCount = 4;
12  macro BusAddressWidth = 8;
13  macro BusDataWidth = 8;
14
15  /*
16   * Instructionset configuration
17   */
18  macro InstructionOpCodeWidth = 8;
19  macro BusInstructionWordWidth
20              = (InstructionOpCodeWidth + 2 * BusAddressWidth);
21  macro InstructionWordWidth = BusCount * BusInstructionWordWidth;
22
23  /*
24   * Opcode macros
25   */
26  macro OpCodeNOP = 0b00000000;
27  macro OpCodeMOVE = 0b00000001;
28  macro OpCodeLOAD = 0b00000010;
29  macro OpCodeJMP = 0b00000011;
30
31  /*
32   * Predefined bus addresses
33   */
34  macro BusAddressDontCare = nat2bv(0, BusAddressWidth);
35  macro BusAddressProgramCounter = nat2bv(1, BusAddressWidth);
36  macro BusAddressAccumulator = nat2bv(2, BusAddressWidth);
37
38  /*
39   * Misc
40   */
41  macro ProgramLength = 256;
42  macro MemoryLenght = 256;
43
44
45  /*
46   * For Tests
47   */
48  macro nat2BusAddress(number) = nat2bv(number, BusAddressWidth);
49  macro nat2BusData(number) = nat2bv(number, BusDataWidth);
50
51  macro GenerateMOVE(source, target) = OpCodeMOVE
52                                        @ nat2bv(source, BusAddressWidth)
53                                        @ nat2bv(target, BusAddressWidth);
54
55  macro GenerateUnsignedLOAD(data, target) = OpCodeLOAD
56                                              @ nat2bv(data, BusAddressWidth)
57                                              @ nat2bv(target, BusAddressWidth);
58
59  macro GenerateLOAD(data, target) = OpCodeLOAD
```

```
60                                                    @ int2bv(data, BusAddressWidth)
61                                                    @ nat2bv(target, BusAddressWidth);
62
63  macro GenerateJMP(destination) = OpCodeJMP
64                                                    @ nat2bv(0, BusAddressWidth)
65                                                    @ nat2bv(destination, BusAddressWidth);
66
67  macro GenerateJMPZ(source, destination)
68                                              = OpCodeJMP
69                                                    @ nat2bv(source, BusAddressWidth)
70                                                    @ nat2bv(destination, BusAddressWidth);
71
72  macro GenerateNOP = OpCodeNOP
73                        @  nat2bv(0, BusAddressWidth)
74                        @ nat2bv(0, BusAddressWidth);
75
76
77
78  module FibonacciTest(mem bv{BusDataWidth} !output) {
79
80      mem [ProgramLength][BusCount]bv{BusInstructionWordWidth} programMemory;
81      event nat{ProgramLength} nextProgramCounter;
82      event [BusCount]bv{BusInstructionWordWidth} instruction;
83
84      mem [MemoryLenght]bv{BusDataWidth} memory;
85
86      mem bv{BusDataWidth} ramAAddress;
87      mem bv{BusDataWidth} ramAReadValue;
88      event bool ramAWriteEnable;
89      mem bv{BusDataWidth} ramAWriteValue;
90
91      mem bv{BusDataWidth} ramBAddress;
92      mem bv{BusDataWidth} ramBReadValue;
93      event bool ramBWriteEnable;
94      mem bv{BusDataWidth} ramBWriteValue;
95
96
97
98      ProgramMemory: SimulatedProgramMemory(nextProgramCounter,
99                                              instruction,
100                                             programMemory);
101     ||
102     SimulatedRamA : SimulatedRam(ramAAddress,
103                     ramAReadValue,
104                     ramAWriteEnable,
105                     ramAWriteValue,
106                     memory);
107     ||
108     SimulatedRamB : SimulatedRam(ramBAddress,
109                     ramBReadValue,
110                     ramBWriteEnable,
111                     ramBWriteValue,
```

```
112                       memory);
113     ||
114     CpuComposed: CpuComposed(instruction,
115                                 nextProgramCounter,
116                                 ramAAddress,
117                                 ramAReadValue,
118                                 ramAWriteEnable,
119                                 ramAWriteValue,
120                                 ramBAddress,
121                                 ramBReadValue,
122                                 ramBWriteEnable,
123                                 ramBWriteValue,
124                                 output);
125     ||
126     {
127         programMemory[0][0] = GenerateLOAD(1, 2);
128         programMemory[0][1] = GenerateLOAD(1, 3);
129         programMemory[0][2] = GenerateNOP;
130         programMemory[0][3] = GenerateNOP;
131         programMemory[1][0] = GenerateMOVE(2, 41);
132         programMemory[1][1] = GenerateMOVE(3, 61);
133         programMemory[1][2] = GenerateMOVE(3, 40);
134         programMemory[1][3] = GenerateNOP;
135         programMemory[2][0] = GenerateMOVE(3, 2);
136         programMemory[2][1] = GenerateMOVE(62, 3);
137         programMemory[2][2] = GenerateNOP;
138         programMemory[2][3] = GenerateNOP;
139         programMemory[3][0] = GenerateJMPZ(64, 1);
140         programMemory[3][1] = GenerateNOP;
141         programMemory[3][2] = GenerateNOP;
142         programMemory[3][3] = GenerateNOP;
143         programMemory[4][0] = GenerateJMP(0);
144         programMemory[4][1] = GenerateNOP;
145         programMemory[4][2] = GenerateNOP;
146         programMemory[4][3] = GenerateNOP;
147     }
148 }
149
150 drivenby TestCase {
151     for(i = 0 .. 64) {
152         pause;
153     }
154 }
```

Listing 27: Fibonacci test module