Formal Semantics of Synchronous Transfer Architecture

Gordon Cichon, Martin Hofmann
Institut für Informatik
Ludwig-Maxilimilians-Universität München
Oettingenstr. 67
81669 München
gordon.cichon@ifi.lmu.de
hofmann@ifi.lmu.de

Abstract: This paper explores the use of formal verification methods for complex and highly parallel state machines. For this purpose, a framework named Synchronous Transfer Architecture (STA) is being used.

STA is a generic framework for digital hardware development that contains VLIW, FPGA, and hardwired ASIC architectures as corner cases. It maintains a strictly deterministic system behavior in order to achieve substantial savings in hardware costs, thus enabling systems with high clock speed, low power consumption and small die area. The high degree of parallelism requires a diligent development methodology to avoid implementation errors. Consequently, formal verification is the methodology of choice for reliable verification.

The contribution of this paper is a formal semantics for the STA hardware architecture framework. This semantics is then used for the formal verification of an optimized parallel implementation of Fast Fourier Transformation (FFT) on STA. This is achieved using a combination of the semantics and symbolic evaluation.

1 Introduction

Synchronous Transfer Architecture [Cic04, CRS$^+$04b] is an architectural framework for the design of special purpose hardware which is used to assist the main processor at demanding computational tasks in small devices such as mobile phones or car electronics, e.g. in advanced driver assistance systems (ADAS). Typical tasks to be offloaded to such specialized hardware are signal processing algorithms such as FFT and filtering, algorithms for error-correcting codes (Reed-Solomon, Viterbi), graphics and image processing, and generic linear algebra (solving equation systems, least mean squares (LMS), singular value decomposition (SVD), Kalman).

Traditionally, such components are implemented either as 1) application specific integrated circuit (ASIC): hardwired circuitry is fast but costly to develop and verify; or as 2) field-programmable gate array (FPGA): reconfigurable logical circuits are still reasonably fast and less expensive to develop than ASIC, but costly to deploy due to high power consumption and chip area; or as 3) digital signal processor (DSP): traditional DSPs do not offer much parallelism, while state-of-the-art microprocessors have a rather high overhead for runtime parallelization of sequential code.

Synchronous transfer architecture (STA) is an architectural framework designed for trading off among the three extremes described above. It allows a fine-grained tradeoff between cost of development and deployment on the one hand, and performance and power consumption on the other. Additionally, and more importantly, STA relies on statically determined parallelism which can considerably save hardware resources, and facilitates simulation and verification.

STA is a collection of DSP components such as arithmetic logic unit (ALU), floating point units, register files and memories, which are dynamically reconfigured. This reconfiguration process can be regarded as a highly parallel assembly program that is read from an instruction memory. All the components of an STA system operate synchronously and in parallel. The assembly language facilitates the dispatch of simultaneous commands to each of these units. Thus, the pipelining policy is exposed at the instruction set architecture. As a result, the highly parallel STA programs may be difficult to understand for a human reviewer. Thus, rigorous verification is essential as in the case of FPGA and ASIC. On the other hand, due to the relatively high abstraction level of assembly language, compared to register transfer language (RTL), rigorous verification is considerably easier than for those.

This paper substantiates the claim that STA facilitates formal verification by providing a formal semantic model of STA and using this model to give a formal functional verification of an industrial-strength implementation of Fast Fourier Transform (FFT).

This paper considers a low-power hardware accelerator with a floating point adder and a floating point multiplier. These two functional units operate in parallel with several integer units (e.g. ALU) that maintain indices and loop counters and with the memories. Thus, it serves as an example about how to deal with a high level of parallelism in such systems.

The FFT implementation considered in this paper completes in 5844 clock cycles. This means near-optimal utilization of the employed floating point processing units. It is the same level of performance that might be expected from a super-scalar microprocessor. However, the STA system does not consume hardware resources for dynamic scheduling, branch prediction, and so on. The STA system is a relatively frugal architecture that consumes about the same area and power as a traditional 32-bit RISC micro-controller, with higher performance. At the same time, the lack of dynamic scheduling makes the architecture strictly deterministic, and thus much more favorable for safety-critical applications.

After describing more details about the STA framework, this paper will present a formal semantic model of STA. This model takes the form of a mathematical function mapping a configuration and its initial memory to its final memory contents. An implementation of this function in a functional programming language (i.e. OCAML) renders it executable. Besides providing a simulator of the STA, this function can be evaluated semantically using symbolic arithmetic expressions, rather than actual values. This allows us to compute the result of the FFT in the form of a vector of symbolic arithmetic expressions.

These expressions can be proven to be indeed equal to the mathematical specification of the FFT by employing automated symbolic algebra.

2 Related work

Related work can be categorized into two different areas: formal equivalence checking of hardware at different levels of abstraction, and formal verification of pipeline implementations.

2.1 Formal Equivalence Checking

Formal equivalence checking is based on hardware models that are represented as finite state machines (FSM). These finite state machines can either be implemented on the abstraction levels of silicon geometry, netlists of register transfer level (RTL). The purpose of formal verification is mainly to prove the equivalence of the different models at various abstraction levels.

Formal equivalence checking is also widespread in the EDA (electronic design automation) community. Almost every EDA vendor offers tools to establish formal equivalence at different abstraction levels [SY, ADK08].

Formal equivalence checking can be performed either by binary decision diagrams (BDDs) [Bry86, BD94] or by Boolean satisfiability (SAT) solvers [BCCZ99]. [BD02] uses integer linear programming (ILP) to verify hardware design. This is an alternative to SAT solvers. The system is described on register transfer level (RTL) as combinational logic that is interpreted as a function that operates on bit vectors.

Bluespec [Arv03, AN08] presents a new hardware description approach based on functional programming. This enables the methodology present in these logic programming languages to be applied to hardware systems. Like in our approach, Bjesse chooses the implementation of an FFT algorithm [Bje99]. However, his target architecture is FPGA, while this paper explores STA.

Furthermore, this paper relies on the assumption that the FFT algorithm itself is functionally correctly specified (as given in [Cap01, Gam02]), and that the numerical stability is provided (as given in [AT04]). These implementation-independent properties of the FFT algorithm have been described in literature previously.

A very common implementation of such FSMs are sequential synchronous circuits (SSC). As it will be explained below, synchronous transfer architectures (STA) are a special case of such SSCs. Consequently, the methodology to ensure correctness of the lower abstraction layers of their implementation can be applied to STAs right away. In fact, an important basis for the verification of STAs is the assumption that their correct implementation is verified using formal equivalence checks. In other words, formal verification of STAs relies on the availability of the methods in this related work to be carried out thoroughly.

As noted, once formal reasoning on FSMs is taking place, it is obvious to also verify certain analytical properties of them. This leads us to the second large area for formal verification: the verification of pipeline processors, as described in the following subsection.

2.2 Pipeline Verification

A large class of system implementations are parallel processors. These are implemented using pipelining and super-scalar scheduling. The conceptual model of these machines is very simple: an ordered sequence of instructions that are supposed to be carried out one after each other. On the other hand, their actual implementation in hardware is a different story.

Intelligent hardware units take a sequential instruction stream, figure out at run-time which parts of it can be carried out in parallel, and carry them out such that this parallelism remains virtually invisible.

This is a huge challenge for hardware implementation. Besides consuming large amounts of resources (die area, electrical power), these systems are very complex and consequently error-prone and hard to verify. Consequently, formal verification has become essential in order to ensure their correctness.

Here are some examples of this approach:

The most recent relevant work has been done by teams at IBM [MBP$^+$04, Cam97], DEC [BBJR97], and Intel [KSKH04]. Industrial strength work in formal verification of microprocessor designs have been performed at Intel [KGN$^+$09], and Centaur [SDSJ11].

Verification of a scalar pipelined RISC processor with the PVS theorem prover is described in [Cyr94]. The processor used is relatively simple as it does not have the sophisticated control of a super-scalar design. Verification of such processors with a focus on the control part and using binary decision diagrams (BDDs) is described in [BD94].

[SJ] describes a framework for verifying a pipelined microprocessor whose implementation contains precise exceptions, external interrupts, and speculative execution using the ACL2 theorem prover. The use of Isabelle by Hewlett-Packard in the design of the HP 9000 line of servers' Runway bus lead to the discovery of a number of bugs uncaught by previous testing and simulation [Cam97].

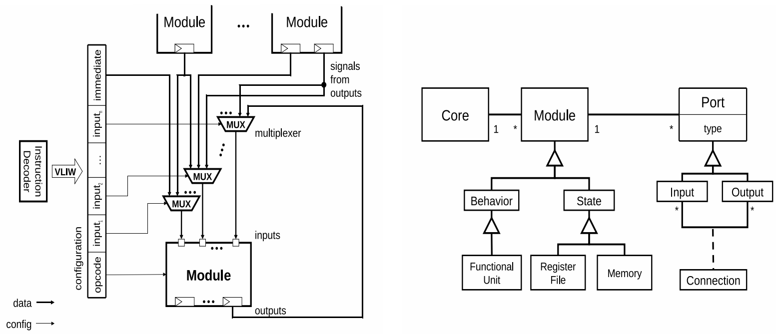
[Bey07] describes formal verification of a cache memory and its integration into an ARM compatible microprocessor called VAMP. It includes an instruction set architecture (ISA) model down to gate-level verification, and the Cambridge ARM model [Fox03] for formalization of this ISA.

[BBM$^+$07] describes full formal verification of the Infineon Tricore processor. It does not only check the correctness of specific properties of the design. It also checks for completeness, i.e. whether all possible input scenarios are covered.

3 Synchronous Transfer Architecture (STA)

The Synchronous Transfer Architecture (STA) [Cic04, CRS$^+$04b] is an architectural framework that enables the design of high-performance, low-power reconfigurable hardware systems. STA aims to shift the effort for the execution of parallel operations from hardware to software.

STA is focused on simplicity and aimed to avoid implementation bottlenecks of super-scalar processors and is thus efficient in hardware. It requires neither local queues for collecting operands, nor a controller that determines when exactly an operation is to be started. In a predictable execution environment, the STA approach triggers the execution of operations explicitly by supplying control signals from its configuration. In contrast to traditional FPGAs, the configuration can change on a per-cycle basis, thus enabling more effective resource sharing.



(a) Modules (b) Machine Description (UML)

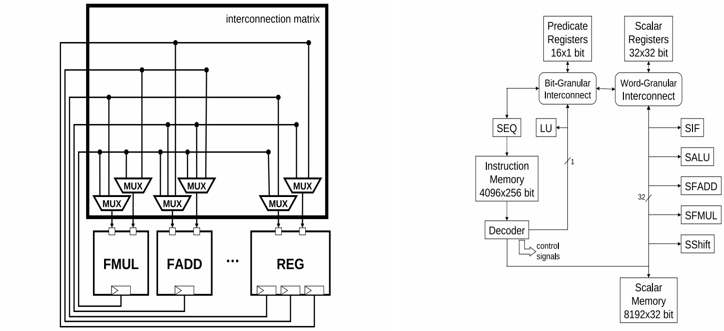
Figure 1: Synchronous Transfer Architecture (STA)

Figure 1(a) shows the architectural framework of STA. The processor is split into an arbitrary number of modules, each with arbitrary input and output ports. To facilitate hardware synthesis and timing analysis, it is required that all output ports be buffered. Each input port is connected to a design-dependent set of output ports, as shown in Figure 2(a). For each computational resource, its STA configuration contains the control signals (opcode) for the functional unit and the multiplexer controls the sources of all input ports and associated immediate fields. (A multiplexer is an electronic device that selects one of several input signals, which one is dependent on a control signal, and forwards it to its output signal.)

Figure 1(b) shows an UML diagram of a STA architecture. A STA core consists of a set of modules. Each module can be either a functional unit performing some computation, or it can be a state module, i.e. a register file or memory. This subdivision enables one to target STA systems with compilers [Cic04, CRS$^+$04a].

In [Cic04], it is demonstrated how arbitrary hardware architectures can be reformulated as STA. This is performed by subdividing the existing hardware modules into their functional and state-specific portion. Figure 2(a) shows all input multiplexers together forming the interconnection matrix between the output and input ports. This system constitutes the synchronous data flow network. The switching matrix may implement arbitrary connections depending on the application, performance, and power-saving requirements.

In the example shown in Figure 2(a), it can also be seen that this interconnection matrix does not need to be fully populated. For example, the input ports of the functional units only have connections to one read port of the register file, not to all three of them.



(a) STA: Interconnection Network (b) Block Diagram

Figure 2: Raccoon

The connection from each output of any functional unit to a write port of a register file is mandatory. While there is a connection from the output of the multiplier to the input of the adder, there is no direct connection from the output of the adder to the input of the multiplier. Operands that need to go this path needs to be routed through the register file. By this reduced inter-connectivity, the complexity of the interconnection network can be reduced from $O\left(n^2\right)$ to a lower complexity class, in case of highly parallel architectures with a large number of functional units.

4 Raccoon Arithmetic Accelerator

The FFT algorithm that is formally verified in this paper is implemented on a specific STA implementation: the **Raccoon** Arithmetic's Accelerator. Figure 2(b) shows a block diagram of the architecture.

It is a small example design, architected to match the die area and power consumption of simple RISC 32-bit embedded micro-controllers, while offering a higher performance.

Raccoon is a simple floating point accelerator with one floating point adder and one floating point multiplier. Around these, there are additional modules that are designed to support the computational resources running at maximum throughput. These resources are: integer arithmetic (ALU, multiplier, barrel shifter, conditional unit), logical unit, register files at word and bit level, data memory, instruction memory.

5 Case Study: Optimized FFT

This section describes the optimized FFT configuration for which functional verification will be provided. It is highly optimized and designed to achieve the best performance and lowest power consumption on the given hardware resources. The hardware resources ("functional units") are a floating point adder and a floating point multiplier. Around these, there are additional supporting hardware resources; in particular, a register file and an integer ALU.

The configuration presented in this paper implements the standard radix-4 FFT as described in [PM96]. In general, Fast Fourier Transform (FFT) is an efficient implementation of the Discrete Fourier Transform (DFT). DFT is a function mapping a vector z of N complex numbers to an equally dimensioned result Z. It is defined by

Definition 1 (DFT) $Z_k = \sum_{n=0}^{N-1} z_n e^{-\frac{2\pi i k n}{N}}$, *where* $0 \leq k \leq N-1$.

FFT is a recursive divide-and-conquer algorithm that evaluates the Z_k in $O(N \log N)$ time as opposed to the $O(N^2)$ gotten from the definition. The subdivision can be performed using various radices, among which radix-4 has the most favorable performance characteristics. In the radix-4 version of FFT, each problem instance of size N is recursively subdivided into four sub-problems of size $N/4$. Figure 3 shows the mathematical reference, in which d-dimensional vectors are represented as (complex-valued) functions from $\{0, \ldots, d-1\}$.

$\texttt{FFT4}(N, n, \vec{z}) =$
/* $N \geq n$, both n, N powers of 4; $\vec{z}$ a complex vector of size n. Returns the DFT of $\vec{z}$. */
$\texttt{if}\ n = 1\ \texttt{then}\ \lambda k.z_0\ \texttt{else}$
$\quad \texttt{for}\ i = 0, 1, 2, 3\ \texttt{let}\ \vec{Z}^{(i)} = \texttt{FFT4}(N, n/4, \lambda j.\vec{z}(4j+i))\ \texttt{in}$
$\quad \lambda k.\texttt{let}\ p = \lfloor \frac{k}{n/4} \rfloor; q = k \bmod n/4\ \texttt{in}$
$\quad \texttt{dragonfly}(N, n, \lambda i.Z^{(i)}(q), qN/n, 2qN/n, 3qN/n)(p)$

Figure 3: Radix-4 FFT, decimation-in-time

The auxiliary function $\texttt{dragonfly}(N, n, \vec{Z}, u, v, w)$ computes the following 4-vector in an optimized fashion.

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & W_u & -0 & 0 \\ 0 & 0 & W_v & 0 \\ 0 & 0 & 0 & W_w \end{pmatrix} \vec{Z}$$

The values $W_0, \ldots, W_N$ are the precomputed twiddle factors, $W_k = e^{-\frac{2\pi i k}{N}}$.

By induction on N one shows easily that the recursive radix-4 algorithm given in Figure 3 is arithmetically equivalent to the definition of the DFT in Definition 1.

The FFT program that is being verified in this paper is an iterative bottom-up version that employs a number of optimizations, such as strength reduction, handling of "twiddle factors”, parallelizing memory access and arithmetic operations. The program overwrites the input values $(z_j)_j$ with the result values $(Z_k)_k$ and operates entirely in-place. However, the value Z_k will be written into the position $\texttt{bitrev}(k)$ where `bitrev` is the permutation of $\{0, \ldots, 256\}$ which in base 4 is given by reading from left to right (“bit reverse”). E.g. $\texttt{bitrev}(17) = \texttt{bitrev}(0101_4) = 1010_4 = 68$ or $\texttt{bitrev}(140) = \texttt{bitrev}(2030_4) = 0302_4 = 50$.

The total numbers of operations is shown in Table 1. The entire program takes 5844 cycles to complete. It can be seen that the execution speed is limited by the floating-point adder (FP add) hardware resource. During the execution of the algorithm, this resource is almost 100% utilized. This means that hardware performance is optimal with respect to the expended resources.

operation	**count**	**utilization**
FP add	5152	88%
FP mul	3733	64%
ALU	2264	39%
MEM load	1273	22%
MEM store	1024	18%

Table 1: Total number of operations

Even though the Raccoon hardware design has only the hardware resources of a scalar RISC processor (i.e. one functional unit of each kind), it achieves a rate of instructions per cycle (IPC) of 2.3. This IPC rate is comparable to that of super-scalar processors [CSS97]. At the same time, Raccoon has a strictly deterministic execution behavior for safety-critical applications and avoids the overhead for dynamic hardware dispatching and multiple functional units. Therefore, Raccoon consumes only a fraction of the hardware resources (silicon area, power consumption) than a super-scalar or VLIW processor. Also, the total latency of the FFT computation with $19.46\mu s$ @300 MHz is favorable. A highly parallel implementation with 17 floating-point units requires $8.5\mu s$ [SCM$^+$05]. GPUs achieve much higher total throughput, but only if they perform a large number of FFTs simultaneously (for latency hiding).

6 Formal semantics

The formal semantics presented in this paper models the dynamic behavior of an STA system as a discrete evolution of **states** each of which maps locations (memory cells, registers, ports) to values. Commands are abstracted from units by allowing them to access arbitrary ports. Pipelines are specified abstractly by providing their reading and (later) writing times for each command; register bypasses are abstracted by treating register writes as instantaneous.

An STA design comprises several components as detailed subsequently; in particular it has sets of locations, values, and commands, as detailed below in Specifications 1, 2, 3 below. These have been called specifications rather than definitions since they specify a format rather than a mathematical object.

Specification 1 (Locations) *The **set of ports** is written as* $\mathtt{port}$*. It comprises output ports of STA units, such as memories, ALUs, floating point units, register files, etc. Ports are volatile in that values written to them are readable only in the same time slot they are written. A special port* $\mathtt{pc}$ *represents the **program counter** and another port* $\mathtt{done}$ *helps detecting program termination. The **set of registers** is denoted by* $\mathtt{reg}$*, while the set of (data) memory addresses is denoted by* $\mathtt{addr}$*. All these sets are assumed to be pairwise disjoint and define the **set of locations** by* $\mathtt{loc} = \mathtt{port} \cup \mathtt{reg} \cup \mathtt{addr}$*.*

Specification 2 (Values) *The set of **values** is written as* $\mathtt{value}$*, comprising bits, integers, **memory addresses*** $(\mathtt{addr})$*, program locations, floating point values, etc.* $\mathtt{value}$ *is lifted and thus contains a special value* $\perp$ *representing undefinedness. For example, all ports and registers contain* $\perp$ *at the beginning of execution.*

The choice of these sets of course depends on the particular STA design to be modeled as do the operations to be defined later on.

Definition 2 (States) *A **state** is a function* $\sigma : \mathtt{loc} \rightarrow \mathtt{value}$ *representing the contents of all locations, i.e., memory cells, registers and ports.*

Specification 3 (Commands) ***command*** *denotes the set of commands which comprise the following four kinds:*

- ***Operations*** *are quintuples written* $\mathtt{oper}(\mathtt{srcs}, \mathtt{dest}, \mathtt{rdts}, \mathtt{wrt}, \mathtt{opn})$*, where* $\mathtt{srcs} \subseteq \mathtt{port}$ *and* $\mathtt{dest} : \mathtt{loc}$ *and* $\mathtt{rdts} : \mathtt{srcs} \rightarrow \mathbb{N}$ *(reading times) and* $\mathtt{wrt} \in \mathbb{N}$ *(writing time) and* $\mathtt{opn} : (\mathtt{srcs} \rightarrow \mathtt{value}) \rightarrow \mathtt{value}$ *(execution function). It is required that* $\mathtt{wrt} > \mathtt{rdts}(p)$ *for all* $p \in \mathtt{srcs}$*. The idea is that if this command is issued at time* t_0 *then each port* $s \in \mathtt{srcs}$ *is read at time* $t_0 + \mathtt{rdts}(s)$ *yielding value* v_s*. Then, at time* $t_0 + \mathtt{wrt}$ *the result* $\mathtt{opn}(\lambda s.v_s)$ *is written into* $\mathtt{dest}$*.*

- ***Register Writes*** *are pairs written* $\mathtt{regwr}(\mathtt{src}, \mathtt{dest})$ *where* $\mathtt{src} \in \mathtt{port}$ *and* $\mathtt{dest} \in \mathtt{reg}$*; when such a command is issued then the value of* $\mathtt{src}$ *is instantly written into the register* $\mathtt{dest}$*. In practice, the value can be written only one step later, but bypasses ensure that the effect is the same.*

- ***Memory loads*** *are written* $\mathtt{load}(\mathtt{src}, \mathtt{dest}, t_1, t_2, t_3)$ *where* $\mathtt{src}, \mathtt{dest} \in \mathtt{port}$ *and* $t_1, t_2 < t_3$*. When the load command is issued at time* t_0*, the following activities take place on the ports: At time* $t_0 + t_1$ *a value* v *is read from the port* $\mathtt{src}$*; at time* $t_0 + t_2$ *a value* v' *is read from memory address* v *and written at time* $t_0 + t_3$ *to port* $\mathtt{dest}$*.*

- ***Memory stores*** *are written* $\texttt{store}(\texttt{src}, \texttt{dest}, t_1, t_2, t_3)$ *where* $\texttt{src}, \texttt{dest} \in \texttt{port}$ *and* $t_1, t_2 < t_3$. *The command is assumed to be issued at time* t_0. *At time* $t_0 + t_1$ *a value* v *is read from the port* `src`*; at time* $t_0 + t_2$ *a value* v' *is read from port* `dest` *and then* v *is written into memory address* v' *at time* $t_0 + t_3$.

In any of these commands attempting to look up an undefined value will result in an undefined overall result. In a particular STA design only a small subset of the possible commands will be available. (This semantics includes all mathematical functions on values. Not all of these are actually realized in a concrete STA design.)

Example 1 (Integer Addition) *For example the integer addition statement*

```
salu.add sreg.r1 decoder.imm
```

that adds the contents of `sreg.r1` *and* `decoder.imm` *and places the result into* `salu.x` *is represented as*

$$\begin{aligned}\texttt{oper}(\ & \{\texttt{sreg.r1}, \texttt{decoder.imm}\}, \texttt{salu.x}, \\ & [\texttt{sreg.r1} \mapsto 0, \texttt{decoder.imm} \mapsto 0], 1, \texttt{op})\end{aligned}$$

where $\texttt{op}(f) = f(\texttt{sreg.r1}) \oplus f(\texttt{decoder.imm})$ *and* $\oplus$ *is 32 bit integer addition. (*f *is a function from srcs (here* `sreg.r1`*,* `decoder.imm`*) to values according to the definition "Commands", which will be given later.) To be precise, this statement is being modeled as* ***several*** *operations; the one just given and the other ones setting appropriate flags. As Chapter 5 of [COR*$^+$*95] explains, this a common way for modeling machine instructions as arbitrary functions.*

Definition 3 (Histories) *A* ***history*** h *is a function from negative integer numbers (-1,-2,-3,...) to states. It represents the previous few states that are relevant for the evaluation of a command. Most states (and in particular all but finitely many) of the states in a history will be everywhere undefined. Attempting to access an undefined value will as usual result in an error. The set of histories is written as* `hist`.

Definition 4 (Updates) *An* ***update*** *is a finite partial function* $\texttt{loc} \to \texttt{value} \cup \texttt{loc}$. *The set of updates is written as* `update`. $u \oplus u'$ *denotes the union of two updates if it is a partial function again; otherwise* $u \oplus u'$ *is undefined. An update* u *with* $\Im(u) \subseteq \texttt{value}$ *is* `normal`.

Lemma 1 *The partial function* $\texttt{resolve} : \texttt{update} \to \texttt{update}$ *normalizes an update by resolving all indirections recursively by:*

$$\texttt{resolve}(u) = \begin{cases} u, \text{ if } u \text{ is normal;} \\ \texttt{resolve}(u') \oplus [l \mapsto \texttt{resolve}(u')(l)], \text{ if } u = [l \mapsto l'] \oplus u' \end{cases}$$

Proof 1 *This function is undefined if any of the lookups* $\texttt{resolve}(u')(l)$ *or if the recursion does not terminate.* `resolve` *can be efficiently implemented by checking the graph spanned by the* $l \mapsto l'$ *mappings for acyclicity.*

Definition 5 (Semantics of commands) *The semantics of a command c is now given as a function $[\![c]\!]$ from histories to updates as follows:*

$$[\![\mathit{oper}(\mathtt{srcs}, \mathtt{dest}, \mathtt{rdts}, \mathtt{wrt}, \mathit{opn})]\!](h) \\ = \{[\mathtt{dest} \mapsto \mathit{opn}(\lambda s.h(\mathtt{rdts}(s) - \mathtt{wrt})(s)]\}$$

Thus, the values of each source $s \in \mathtt{srcs}$ can be found at position $\mathtt{rdts}(s) - \mathtt{wrt}$ in the history.

Example 2 *For example, if s is read at time 5 (after issuing the command) and the destination is written at time 7 (after issuing the command) then at the time the destination is written the value of the source 2 time steps earlier is relevant, hence position -2 in the history. This latency is always fixed and STA cannot handle operations with variable latency.*

The remaining semantic definitions are now self-explanatory. We put

$$[\![\mathtt{regwr}(\mathtt{src}, \mathtt{dest})]\!](h) = [\mathtt{src} \mapsto \mathtt{dest}]$$

and

$$[\![\mathtt{load}(\mathtt{src}, \mathtt{dest}, t_1, t_2, t_3)]\!](h) = [\mathtt{dest} \mapsto v]$$

where $v = h(t_2 - t_3)(a)$ and $a = h(t_1 - t_3)(\mathtt{src})$. Finally,

$$[\![\mathtt{store}(\mathtt{src}, \mathtt{dest}, t_1, t_2, t_3)]\!](h) = [l \mapsto v]$$

where $v = h(t_1 - t_3)(\mathtt{src})$ and $l = h(t_2 - t_3)(\mathtt{dest})$.

Definition 6 (Programs) *A program is a function $P : \{1, \ldots, N\} \to \mathcal{P}(\mathit{command})$ where N is some integer, the length of the program. The idea is that when $\mathtt{pc}$ (program counter) has value n then the commands in $P(n)$ are simultaneously issued Δ_{fetch} time-steps later and—at their writing times they attempt to write into their respective destinations. Δ_{fetch} is a fixed parameter modeling the delay involved in fetching and decoding commands.*

Example 3 *In the Raccoon architecture, there is $\Delta_{fetch} = 2$.*

If another command attempts to write the same location no matter when it was issued then this constitutes a conflict and leads to an error.

Definition 7 *This is being modeled by using **queues** containing pairs (c, i) with c a command and $i \in \mathbb{N}$ modeling the number of time-steps until c writes into its destination ("fires"). The function $\mathtt{adv} : \mathit{queue} \to \mathcal{P}(\mathit{command}) \times \mathit{queue}$ splits off all commands in a queue whose i value is zero and decrements the i-values of the remaining ones.*

A reasonable program will contain at each group of commands one command that alters the program counter (typically by incrementing it). In practical assembly level programs only the non-incrementing $\mathtt{pc}$-operations, e.g. jumps are explicitly written.

Step function. Our aim is to define a function $\texttt{step}$ which takes a program P, a time t, a function $\Sigma : \{0, \ldots, t-1\} \to \texttt{store}$ and a queue q. It returns an updated queue q' and a store σ representing the contents of locations at time t.

Advance We begin by advancing the current queue, thus write $(\texttt{cs}, q_1) = \texttt{adv}(q)$. So $\texttt{cs}$ are the commands that fire now. With $\texttt{cs'} = P(\Sigma(t-1)(\texttt{pc}))$, the updated queue is being formed as $q' = q_1 \cup \{(c, i) \mid c \in \texttt{cs'}, i = \Delta_{fetch} + t_c\}$. Here, t_c is the time when command c fires, e.g., $t_c = 1$ for $\texttt{salu.add}$. The t_c are parameters of the architecture being modeled.

Update Given $\texttt{cs}$ and Σ we can compute the updates that will take place as

$$u = \texttt{resolve}(\bigoplus_{c \in \texttt{cs}} [\![c]\!](\lambda i \lambda l.\Sigma(t+i)(l))$$

Note that there is the possibility of errors due to conflict. Also note that i is a negative number here.

Finally—if no error has occurred so far—the update is being applied to form $\sigma(l) = v$ if $l \mapsto v \in u$. If l is a memory address or a register, $\sigma(l) = \Sigma(t-1)(l)$ retains the previously stored values. Otherwise, $\sigma(l) = \perp$ makes the result undefined.

Summarizing, we have

$$\texttt{step}(P, t, \Sigma, q) = (q', \lambda l. \begin{cases} u(l) & l \in \mathrm{dom}\, u \\ \Sigma(t-1)(l) & t > 0, l \text{ a memory address or register} \\ \perp & \text{else} \end{cases})$$

where q' and u are defined as above.

Complete evaluation. Now, given an initial store σ_0, a sequence of stores is defined by σ_t and queues q_t by $q_0 = \{\}$ and $(\sigma_t, q_t) = \texttt{step}(P, i, \lambda t'.\sigma_{t'}, q_{t-1})$ for $t > 0$. $\sigma = \texttt{eval}(P, \sigma_0)$ designates the complete evaluation up to $\sigma = \sigma_t$ where t is the earliest time when $\sigma_t(\texttt{done}) = \texttt{true}$. If no such t exists or errors have occurred anywhere on the way then $\texttt{eval}(P, \sigma_0)$ is undefined.

This concludes the description of our semantics; it comprises thirteen specifications and definitions. The semantics has been validated by implementing it in OCAML and comparing its outcomes on several example programs with the outputs produced by real STA hardware as well as the outputs produced by an existing System C simulation of STA. The next section gives the announced application of the semantics to the formal verification of the FFT implementation.

7 Functional verification of an FFT implementation

The formal semantics of the Raccoon design has been implemented as a functional program written in OCAML programming language. This program displays a top-level function which from a given instruction memory and initial data memory computes the global state as a function of time.

Since the flow of control in the specific FFT-program does not depend on concrete values of floating point numbers (but only on integer values in loop counters) and because the scheduling of parallelism is completely static due to the STA methodology it is then possible to replace in the functional implementation the actual floating point numbers by symbolic values representing arithmetic expressions. To this end, the following OCAML algebraic data-type

```
type flr = Add of flr * flr | Sub of flr * flr
   | Mul of flr * flr | Lit of string
```

is being used to evaluate the semantics of the STA design for FFT on the initial memory given by $i \mapsto (\mathtt{Lit}\ \mathtt{s_i}, \mathtt{Lit}\ \mathtt{t_i})$ when $i < 4096$ and $i \,\%\, 8 = 0$ and where

$$s_{8k} = \begin{cases} \mathtt{Re(z\_}k\mathtt{)}, \text{if } k < 256 \\ \mathtt{cos(-2*Pi*}k\mathtt{/256)}, \text{if } k \geq 256 \end{cases}$$

$$t_{8k} = \begin{cases} \mathtt{Im(z\_}k\mathtt{)}, \text{if } k < 256 \\ \mathtt{sin(-2*Pi*}k\mathtt{/256)}, \text{if } k \geq 256 \end{cases}$$

Note that the s_i, t_j are **strings** representing arithmetic expressions and not real valued functions or similar.

In this representation, the flexibility of OCAML syntax is useful: `Lit` is a constructor of type string for a data-type representing symbolic values. Thus, any symbolic expression can be represented as a string value, for example `Lit ``Im(z44)''`.

The resulting output then contains arithmetic expressions in the real- and imaginary parts of the 256 input variables and the real- and imaginary parts of the twiddle factors. The symbolic execution takes less than three minutes to complete on a PC (Intel Dual Core 1.6 GHz processor and 2GB RAM).

Our approach then compares these expressions with the recursive reference implementation of the underlying FFT algorithm `FFT4` (see Figure 3). These expressions were checked for symbolic identity, not merely arithmetical equivalence, with the reference. This then implies not only the functional correctness of our STA implementation but also that its behavior on actual floating point numbers including numerical stability is the same as the reference and thus well-understood [Ram70].

Theorem 1 *The result expressions of the symbolic evaluation are identical to the vector of expressions* $\mathit{FFT4}(N, k, \vec{z})$.

Proof 2 *By direct comparison.*

Interestingly, the symbolic evaluation revealed a bug in an earlier version of the STA design for FFT that could not be found by testing alone. In fact the buggy version read an output port one cycle too late. But this did not lead to an observable error since the actual hardware is currently such that result values remain readable at output ports until they are explicitly overwritten.

8 Conclusion and Future Work

This paper presents the first formal semantic model of the STA architectural framework. By applying this framework on a specific architecture, we have performed formal verification of a computationally intensive and highly parallel algorithm, the FFT, using symbolic evaluation. We have also shown that the presented semantic model is suitable as simulator for the architecture; a simulator that is specified in a functional language.

This verification approach is one important contribution to enable shifting effort of scheduling and parallelizing execution for computationally intensive accelerators from run-time into design-time. This shift contributes to better performance, lower power-consumption and better safety of run-time systems. This gain is performed at the expense of higher effort at design-time.

We have chosen a case study with an algorithm that is computationally intensive and does not have a data-dependent control flow. As a next step, we will consider applications with a data-dependent control flow. For example a dot product with variable vector length. This non-trivial control flow will require to reason about a loop invariants and a fix-point in the semantic model.

As the feasibility of our approach has been shown, we plan to apply it on STA systems with an even higher degree of parallelism in the future. This will be systems with a greater number of functional units, like several floating point units of each kind. This will be both independently operating units, like they are used on an FPGA or wide VLIW processor, and uniformously operating units, like a SIMD system.

The semantics defined in this paper has a rather operational flavor; it is supposed to be fairly close to the actual architecture and thus is not further validated here. It would be possible to prove it sound against even more low level semantic models that represent pipelines, wires, the decoding process, etc. This can be achieved using the formal equivalence checking approach that is being discussed in the related-work section.

Having said that, we can use our semantics to rigorously justify more high level semantics that might be more useful for reasoning by invariants: A fix-point semantics will be specified by a continuous operator $[\![P]\!]$ on the domain of functions $\mathbb{N} \to \texttt{store}$. This $[\![P]\!](\Sigma)$ extracts all commands at all times simultaneously and fires them all at once at the right times and locations. In this way, queues are not needed and it should be easier to establish properties of programs with data-dependent control flow using invariants. We plan to justify such fix-point semantics and its application to reasoning.

Proofs about fix-point semantics might be supported by using a computer-aided theorem-prover, like PVS, Coq, Isabelle, and the like. For a specific class of programs, a SMT solver might be the best choice because of its guaranteed determinism.

References

[ADK08] Arvind, Nirav Dave, and Michael Katelman. Getting Formal Verification into Design Flow. In *Proc. FM '08*, pp. 12–32, Springer, 2008.

[AN08] Arvind and Rishiyur S. Nikhil. Hands-on Introduction to Bluespec System Verilog (BSV) (Abstract). In *MEMOCODE*, pp. 205–206. IEEE, 2008.

[Arv03] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification. In *MEMOCODE*, pages 249–. IEEE, 2003.

[AT04] Behzad Akbarpour and Sofiène Tahar. A Methodology for the Formal Verification of FFT Algorithms in HOL. In [HM04], pages 37–51.

[BBJR97] Gabriel P. Bischoff et al. Formal Implementation Verification of the Bus Interface Unit for the Alpha 21264 Microprocessor. In *ICCD*, pages 16–24, 1997.

[BBM+07] Jrg Bormann, Sven Beyer, Adriana Maggiore, Michael Siegel, Sebastian Skalberg, Tim Blackmore, and Fabio Bruno. Complete Formal Verification of TriCore2 and Other Processors. In *Design Verification Conference (DVCon)*, 2007.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of Pipelined Microprocessor Control. In David L. Dill, editor, *CAV*, LNCS 818, pp. 68–80. Springer, 1994.

[BD02] Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *In Proc. VLSI Design Conf.*, pages 741–746, IEEE, 2002.

[Bey07] Sven Beyer. *Putting it all together: formal verification of the VAMP*. PhD thesis, 2007.

[Bje99] Per Bjesse. Automatic Verification of Combinatorial and Pipelined FFT. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 380–393. Springer, 1999.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[Cam97] Albert Camilleri. A hybrid approach to verifying liveness in a symmetric multiprocessor. In Elsa Gunter and Amy Felty, editors, *TPHOLS*, LNCS 1275, pages 49–67. 1997.

[Cap01] Venanzio Capretta. Certifying the Fast Fourier Transform with Coq. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs*, volume 2152 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2001.

[Cic04] Gordon Cichon. *A Novel Compiler-Friendly Micro-Architecture for Rapid Development of High-Performance and Low-Power DSPs*. PhD thesis, Technische Universität Dresden, Germany, 2004.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, , and Mandayam Srivas. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

[CRS+04a] Gordon Cichon et al. Compiler Scheduling for STA-Processors. In *Proc. (PAR-ELEC'04)*, Dresden, Germany, September 2004.

[CRS+04b] Gordon Cichon et al. , Pablo Robelly, Hendrik Seidel, Emil Matúš, Marcus Bronzel, and Gerhard Fettweis. Synchronous Transfer Architecture (STA). In *Proc. (SAMOS'04)*, pages 126–130, Samos, Greece, July 2004.

[CSS97] Yuan C. Chou, Daniel P. Siewiorek, and John Paul Shen. A Realistic Study on Multithreaded Superscalar Processor Design. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Euro-Par*, LNCS 1300, pages 1092–1101. 1997.

[Cyr94] David Cyrluk. Microprocessor Verification in PVS - A Methodology and Simple Example. Technical report, SRI International, 1994.

[Fox03] Anthony C. J. Fox. Formal Specification and Verification of ARM6. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, LNCS 2758, pages 25–40. 2003.

[Gam02] Ruben Gamboa. The Correctness of the Fast Fourier Transform: A Structured Proof in ACL2. *Formal Methods in System Design*, 20(1):91–106, 2002.

[HM04] Alan J. Hu and Andrew K. Martin, editors. *Proc. FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, LNCS 3312. 2004.

[KGN+09] Roope Kaivola, et al. Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, LNCS 5643, pp. 414–429. 2009.

[KSKH04] Zurab Khasidashvili, Marcelo Skaba, Daher Kaiss, and Ziyad Hanna. Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints. In *ICCAD*, pages 58–65. IEEE Computer Society / ACM, 2004.

[MBP+04] Hari Mony, et al. Scalable Automated Verification via Expert-System Guided Transformations. In [HM04], pages 159–173.

[PM96] J.G. Proakis and D.G. Manolakis. *Digital signal processing: principles, algorithms, and applications*. Prentice Hall, 1996.

[Ram70] George Ramos. Roundoff error analysis of the fast Fourier transform. Technical Report STAN-CS-70-146, Stanford University, February 1970.

[SCM+05] Hendrik Seidel, Gordon Cichon, et al. Development and Implementation of a 3.6 GFLOP/s SIMD-DSP using the Synopsys Toolchain. In *Fourteenth Annual Synopsys Users Group Europe*, Munich, Germany, May 2005.

[SDSJ11] Anna Slobodová, Jared Davis, Sol Swords, and Warren A. Hunt Jr. A flexible formal verification framework for industrial scale validation. In Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt, editors, *MEMOCODE*, pages 89–97. IEEE, 2011.

[SJ] Jun Sawada and Warren A. Hunt Jr. Processor Verification with Precise Exceptions and Speculative Execution.

[SY] Erik Seligman and Itai Yarom. Best known methods for using Cadence Conformal LEC at Intel.