Automatic Verification of Combinational and Pipelined FFT Circuits

Per Bjesse

Chalmers University of Technology, Sweden
`bjesse@cs.chalmers.se`

Abstract. We describe how three hardware components (two combinational and one pipelined) for computing the Fast Fourier Transform have been proved equivalent using an automatic combination of symbolic simulation, rewriting techniques, induction and theorem proving. We also give some advice on how to verify circuits operating on complex data, and present a general purpose proof strategy for equivalence checking between combinational and pipelined circuits.

1 Introduction

FFT components are a challenge to verify as they compute complex functions involving many arithmetic operations. Bit-level correctness proofs for such circuits are not within the reach of today's technology; an appropriate level of modelling is therefore on the level of individual arithmetic operations on signals carrying numerical data.

In order to make verification techniques industrially interesting, it is generally agreed that a high degree of automation is desirable. Unfortunately classical automatic methods such as propositional logic tautology checking or model checking can not be immediately applied at this level of abstraction. Different extensions of model checking with uninterpreted functions encoded in BDDs have been proposed [VB98]; we instead use theorem proving, but in such a way that no user guidance is needed during the proofs.

As we aim for verification at the arithmetic level, it is imperative to structure the proofs to be as simple as possible; we therefore devise heuristics for the particular class of circuits we verify and apply automatic analyses that aim to reduce the work that has to be done in the theorem prover. For this end we use the Lava hardware development platform that has a powerful language in which we can implement our analyses and write parametrisable scripts that control complex theorem prover interactions [BCSS98].

The work described is an industrial case study with Ericsson Cadlab, Stockholm.

2 The Lava Hardware Development Platform

Lava is a hardware description language and a framework for hardware verification developed at Chalmers and Xilinx [BCSS98]. One of the principal uses of Lava is as a platform for hardware verification experiments.

Lava is embedded in the functional language Haskell; all aspects of the development of hardware from descriptions down to the interfacing to layout tools are expressed in the same language. The use of a polymorphic high level language that supports higher order functions gives very concise hardware descriptions and allows us to devise combinators that capture common design patterns.

The circuit descriptions can be interpreted by symbolic evaluation in a number of different ways; examples of built in standard analyses are circuit simulation, generation of logical formulas in formats suitable for external theorem provers and generation of VHDL. The verification interpretation is parametrised over the proof procedure and allows the passing of optional proof parameters; a user can therefore quickly retarget from one proof procedure to another without losing fine grain control.

3 The Fast Fourier Transforms

The Fast Fourier Transforms (FFTs) are efficient algorithms for computing a length N sequence of complex numbers X given an initial sequence x and a constant W_N defined as $e^{-j2\pi/N}$:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{kn}, \qquad k \in \{0 \ldots N{-}1\}$$

The FFTs exploit symmetries in the *twiddle factors* W_N^k together with restrictions of sequence lengths (for example to powers of two) to reduce the number of necessary computations. Examples of twiddle factor laws that express useful symmetries are

$$\begin{aligned} W_N^0 &= 1 \\ W_N^N &= 1 \\ W_n^k \cdot W_n^m &= W_n^{k+m} \\ W_n^k &= W_{2n}^{2k}, \qquad (n, k \leq N) \end{aligned}$$

The FFT algorithms are often implemented in combinational hardware, and are key building blocks in signal processing applications; the FFTs are rumoured to be the worlds most implemented algorithms in hardware.

The reference FFT is the *decimation in time* Radix-2 algorithm, which operates on input sequences whose length is a power of two [PM92]. If the input length also is a power of four, the *decimation in frequency* Radix-2^2 FFT can be applied [He95]. From a designer's point of view the question is whether the combinational circuits that implement these algorithms are equivalent. As the networks are fundamentally different, verification of equivalence is a non-trivial undertaking.

Combinational implementations are not the only ones possible; pipelined sequential designs can use less circuit area by trading space for time. A pipelined implementation of a size 2^n Radix-2^2 FFT (see figure 1) consists of two simple

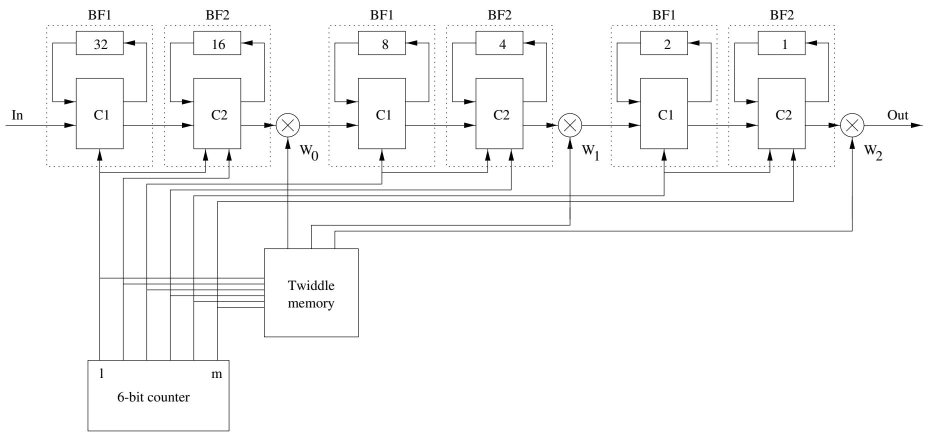

Fig. 1. Structure of pipelined implementation of a size 64 Radix-2^2 FFT

kinds of combinational components ($C1$ and $C2$) that together form a stage; a whole circuit consists of $n/2$ stages. Each primitive block is controlled by synchronisation signals generated by an n-bit counter. This counter also addresses a multi port memory that outputs streams of twiddle factors that are multiplied together with the outputs of each stage.

Figure 2 shows how the pipelined FFT circuit simulates the corresponding combinational circuit over time by reading the inputs in the first sequence of input values $IF(0)$ while spitting out undefined outputs until time lag ($2^n - 1$ for a size 2^n FFT) when the first element of the output sequence $OF(0)$ is generated; the lag time is always constant. At the same time as the outputs are produced, inputs from a new input sequence are read so that the circuit continuously processes data.

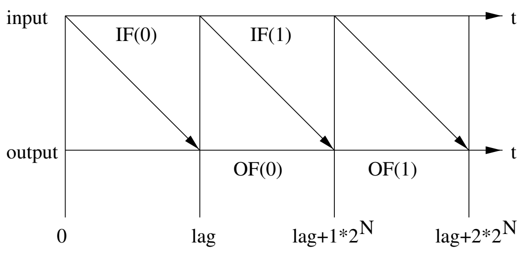

Fig. 2. Operation of the pipelined circuit

4 FFT Low-Level Descriptions

The FFT descriptions are parameterised by the circuit size and are formulated using a number of simple circuits and combinators that are useful for signal processing applications.

A key point is that the regularity of the combinational networks makes the circuits very easy to describe in Lava; the description of the Radix-2 FFT in terms of the signal processing combinators is just 3 lines long (see appendix A).

The Lava circuit descriptions can be used to automatically generate structural VHDL for all parts of the implementations with the exception of the multi port memory component.

5 Verification of Components

As we want automatic proofs, we will only be concerned with equivalence checking for fixed size circuits. We will also exploit designer knowledge and use Lava analyses in order to make the proofs tractable for the external proof procedure. The circuits are modelled on the level of operations on infinite precision complex numbers; this modelling is appropriate as finite representations of complex numbers only can be used for approximate calculation of the FFT. A reasonable notion of implementation equivalence must therefore be defined in terms of infinite precision complex arithmetic.

As a shorthand, we adopt the convention that

$$F(x,y) \equiv F(x(0)..x(i-1), y(0)..y(i-1))$$

if $F \in Form$ (the set of first order logic formulas) and $x, y \in S^i$ where S is any non-empty set.

5.1 Theoretical Basis of the Verifications

Combinational circuits can be viewed as functions f from input to output. Lava's symbolic evaluation can generate formulas δ_f that define the functions we are concerned with in the sense that $T \vdash \delta_f(I,O) \Rightarrow f(I) = O$ if T is a theory containing theorems that are true in a standard interpretation of complex arithmetic.

The formulas that are constructed in the following verifications are expressed in first order logic with equality, and contain variables and two-place function symbols *plus*, *sub*, *tim* and W. The circuit equivalence checking problem is reduced to showing that certain formulas that capture implementation equivalence are members of the theory T which we give axioms for. The axioms are well-known properties of complex arithmetic and some twiddle factor identities. We know that the axioms hold in the interpretation $\mathfrak{I}$ that complies with the following conditions

– The domain is the set of complex numbers

- *plus* designates complex addition
- *sub* designates complex subtraction
- *tim* designates complex multiplication
- W designates the function $f_w(k, N) = e^{-j2\pi k/N}$

All formulas that are derivable from the axioms in a sound proof system are therefore also true in $\mathfrak{I}$.

5.2 Combinational FFT Verification

Are the abstract implementations of the Radix-2 and the Radix-2^2 FFT equivalent for sizes that are an exponent of four?

The fixed size FFT circuits are functions $F_1(I)$ and $F_2(I)$ from complex input sequences to complex output sequences. Lava's symbolic evaluation can generate formulas δ_1 and δ_2 that define these functions. Our criterion for equivalence of the combinational FFT is that

$$\delta_1(I, O_1) \wedge \delta_2(I, O_2) \ \rightarrow \ O_1 = O_2$$

Instead of generating the two defining formulas individually and then combining them together to a resulting formula, we can construct a test bench circuit that directly generates the correctness formula when interpreted symbolically:

```
fftSame n =
  do inp <- newCmplxVector (4^n)
     out1 <- radix2 (2*n) inp
     out2 <- radix22 n inp
     equals (out1,out2)
```

The test bench builds a vector of unrestricted complex variables, which are given to both FFT implementations. The resulting output sequences are then pointwise compared to each other for equality. If the formula describing this system is derivable by the theorem prover using the axioms for the theory T, then it is true in the model $\mathfrak{I}$ and the implementations are equivalent.

Lava's verification interpretation takes a test bench circuit and a proof procedure with some arguments, and automatically generates formulas and runs the proof. The manual step that has to be taken is to choose a prover and possibly give proof options. In this case, we have to choose a first order logic theorem prover, and specify some axioms. These include some simple algebraic laws for the arithmetic operators, such as distributivity of multiplication over addition and that 1 is a unit element for multiplication. The twiddle factor identities from section 3 are also necessary.

Although these axioms with any first order logic prover are in theory sufficient to prove the circuits equivalent, the number of consequences grows very quickly if the rules are applied mindlessly. This combined with the fact that the FFT circuits generate formulas that for larger sizes grow to be megabytes big means that we must give extra proof options in order to make the proofs tractable. Symbolic evaluation of the FFTs for 4 abstract inputs reveals some interesting circuit properties (the input and output vectors are indexed backwards):

```
Lava> symbolic_eval (radix2 2)
[(x3 - W(2, 0) * x1) - W(4, 1) * (x2 - W(2, 0) * x0),
 (W(2, 0) * x1 + x3) - W(4, 0) * (W(2, 0) * x0 + x2),
 W(4, 1) * (x2 - W(2, 0) * x0) + (x3 - W(2, 0) * x1),
 W(4, 0) * (W(2, 0) * x0 + x2) + (W(2, 0) * x1 + x3)
]

Lava> symbolic_eval (radix22 1)
[W(4, 0) * ((x3 - x1) - W(4, 1) * (x2 - x0)),
 W(4, 0) * ((x1 + x3) - (x0 + x2)),
 W(4, 0) * (W(4, 1) * (x2 - x0) + (x3 - x1)),
 W(4, 0) * ((x0 + x2) + (x1 + x3))
]
```

The lack of control logic in the combinational FFT components causes the circuit outputs to be polynomials in the inputs and twiddle factors only. Rewriting of the expressions by simplifying away twiddle factors that are equal to 0 or 1, conversion of the remaining twiddle factors to the form W_N^x and restructuring of arithmetic expressions to sum of products form makes it possible to show the two results equal by syntactic equality alone.

The rewriting has to be done in a particular way for it to be applicable to the larger circuits. If the axioms are given as standard equalities, they can be used in both directions. This is not how the most efficient proof would proceed, as it suffices to use all the rules in one direction only: expand out the polynomials, take away trivial twiddle factors and rewrite the others.

Unidirectional rules are therefore more suitable for our purposes. The theorem prover Otter has efficient such rules that are called demodulators [MW97]; the use of a demodulation rule can be unconditional or restricted by predicates on terms. An important property of these rules is that they are used as often as possible *without accumulating intermediate results*. This reduces the number of consequences and makes normalisation of large expressions tractable.

The demodulation proof rules are specified inside Lava and passed to Otter as two theories. The actual proofs are done by calling the verification interpretation on the test bench and the proof configuration:

```
options = [Prover otter, Theory arithmetic, Theory (twiddle 4)]

Lava> verify options (fftSame 1)
Valid
```

In this way the equivalence of circuits up to size 256 is proven automatically. Statistics for the resulting proofs and some system formula measures such as the number of primitive logical and arithmetic operations are given in table 1. The running times are measured on a 300 MHz Sun Enterprise 450.

Table 1. Statistics for verification of equivalence between combinational FFTs

FFT size	Verification time (s)	Formula size (Bytes)	# of variables	# of formula operations
4	0.09	1179	33	59
16	0.39	10 761	233	433
64	10.31	172 088	1334	2529
256	827	2 886 561	6939	13 313

5.3 Pipelined FFT Verification

We would now like to verify that the sequential pipelined implementation of the Radix-2^2 is equivalent to the combinational circuit. We employ a strategy that is optimised for equivalence checking of combinational and constant delay ("lag") pipelined circuits.

The presentation is divided into two parts: The first part describes the strategy and the second demonstrates how it applies to the particular case of our FFT verification.

A strategy for pipeline equivalence proofs If we observe the pipelined circuit for a single clock period, it is a function from a starting state S and input I to a finishing state S' and a resulting output O.

$$(O, S') = ppl(I, S)$$

We use the term "frame" to refer to a complete in- or output data sequence for the combinational or pipelined circuit. Lava can generate a defining formula $\delta_{ppl}(I, S, O, S')$ for the $ppl(I, S)$ transition function that captures how the circuit behaves over a single clock tick. The objective is to show equivalence between the two implementations for any number of successive frames starting from a (partially) specified initial state, using the following verification strategy which we refer to as $Equiv_\omega$:

1. Generate the defining formula $\delta_{ppl}(I, S, O, S')$ of the pipelined circuit.
2. Define l to be the number of inputs that the pipelined circuit has to consume before it can read the first input of the second frame.
3. Define m as the least number of time steps that the pipelined circuit has to run to allow an observer to deduce that the output from the sequential circuit matches a single frame of output from the combinational implementation.
4. Let $k = max(l, m)$.
5. Let δ^k_{ppl} be the following formula that expresses what behaviour a length k trace of the sequential circuit exhibits

$$\delta_{ppl}(I_0, S_0, O_0, S_1) \wedge \delta_{ppl}(I_1, S_1, O_1, S_2) \wedge \ldots \wedge \delta_{ppl}(I_{k-1}, S_{k-1}, O_{k-1}, S_k)$$

   This is the k-step unrolling of the pipelined transition function.
   We refer to a trace that is a model for δ^k_{ppl} as a T trace, and observe the following:

  - If we define an initialisation state as a state that immediately precedes the processing of a new frame, both S_0 and S_l are initialisation states on all T traces. Furthermore, S_l is the closest initialisation state to S_0.
  - Any infinite trace of the system is made up from infinitely many concatenated T traces; given that $l < k$ successive traces tr_n and tr_{n+1} also overlap with $tr_n(l \ldots k-1) = tr_{n+1}(0 \ldots k-l-1)$.
6. Generate a defining formula for the combinational circuit, $\delta_{cmb}(I, O)$.
7. From δ^k_{ppl} and δ_{cmb}, construct a formula λ that expresses implementation equivalence for a single frame of inputs
8. A proof of λ without any assumptions at all on the initialisation state S_0 implies $\forall S_0 . \lambda$. This corresponds to equivalence for any number of time frames as the circuits will behave in the same way regardless of the initialisation state values before a new frame is processed; a direct proof of λ is hence not realistic. Therefore strengthen the assumptions on S_0 by a formula ϕ that restricts some of the S_0 variables to the initial values given in the pipelined circuit description. If now
$$\phi(S_0) \;\rightarrow\; \lambda$$
is provable, the circuits are equivalent for any number of time frames under the assumption that ϕ is always true in initialisation states. Refer to this assumption as assumption A
9. Try to prove assumption A valid by a proof of
$$\phi(S_0) \wedge \lambda \;\rightarrow\; \phi(S_l)$$
As ϕ holds in the initial state of the circuit, this formula implies A as it asserts that ϕ will hold in the state S_l (that is reached immediately before a new processing cycle is initiated) if ϕ is true in S_0 (that was reached immediately before this frame was processed); A is therefore entailed by induction.
10. If step 8 and step 9 were successful, deduce multi frame equivalence

A valid question is, of course, "Why is it reasonable to assume that a part of the pipelined circuit always is in a state where ϕ holds before a new frame is read?". This is probable as the pipelined circuit is supposed to repeat the frame processing behaviour again; the registers in the control logic should therefore have similar contents in the initialisation states as in the specified initial circuit state.

By having reduced the problem to two simple proofs we have devised a simple strategy for showing pipelined circuits with a fixed lag equivalent to combinational implementations. This strategy is implemented in an automatic Lava proof script that is parameterised over circuit descriptions, frame length, the constant lag and a proof configuration for the frame equivalence proof. This script automatically generates and reduces all formulas as much as possible before calling the theorem prover specified in the proof configuration; the only manual steps are to choose which state variables to restrict and to select a proof procedure. Any prover and extra proof options can be specified in the proof configuration; the pipelined circuit description can also have as many or as few initial values given as desired.

Application to the pipelined Radix-2^2 FFT The script that implements $Equiv_\omega$ proves pipeline equivalence for the FFT circuits with the automatically generated equivalence formula λ defined as

$$\delta^k_{ppl}(I_0..I_{k-1}, S_0..S_k, O_0..O_{k-1}) \wedge \delta_{cmb}(I_0..I_{i-1}, O'_0..O'_{i-1}) \rightarrow O_{lag}..O_{k-1} = O'_0..O'_{i-1}$$

where $lag = 2^N - 1$, $i = 2^N$ and $k = 2^N + lag$.

A sufficient restriction ϕ on the initial state of the pipelined FFT circuit is that the n-bit counter is initialised to 0. The reason why this simple assertion is strong enough to prove the FFT implementations equivalent is that at re-initialisation the rest of the pipeline state is unimportant, new values have to be read for processing anyway. This is likely to hold for most pipelined implementations of combinational circuits.

The initialisation information ϕ is always used by the Lava script to reduce the generated formulas as much as possible while they are produced. This reduction computes the values of logical expressions whenever possible and propagates the resulting new information. As a consequence, the formulas that specify the behaviour of the control logic inside the pipelined FFT are evaluated away and the re-initialisation invariant in step 9 of $Equiv_\omega$ is proved by syntactic equality. The equivalence checking problem for the pipelined FFT is therefore reduced back to a proof of an equivalence formula that turns out to be amenable to normalisation with the theories used for the combinational equivalence checking. The complexity of the resulting proofs are indicated in table 2.

Table 2. Statistics for verification of pipelined equivalence

FFT size	Verification time (s)	Formula size (Bytes)
4	0.05	1227
16	0.61	10 045
64	22.26	162 862
256	1361	2 797 617

5.4 Manual Preparation

Approximately two weeks was spent on studying the FFT implementations, devising signal processing combinators and writing circuit descriptions. The addition of support in Lava's interpretations for complex numbers and the writing of the symbolic simulation interpretation with automatic formula reduction took one week of work each.

Finding the proof procedure was the creative step for the combinational FFT verification. Two other theorem provers, Prover [Stå89] and Gandalf [Tam97], was tried before Otter. Prover lacked crucial arithmetic laws, and Gandalf did not support the unidirectional rules that were needed to make the proofs scale up. A correct set of rewrite rules took some hours work by two users, Koen Claessen and Tanel Tammet, who were unfamiliar with the FFT but knew Otter well. Any other applicable proof procedure would also have needed rewrite rules for the twiddle factors, so we believe that this degree of manual work is unavoidable.

Once the symbolic simulation interpretation with formula reduction was written, a first (more involved) pipeline proof script could be constructed in half an hour. This strategy was successful the first time it was tried; we later simplified the heuristic to the presented form. The only non-reusable steps of the combinational and pipelined verifications were to choose Otter with rewrite rules as the proof procedure and to restrict the synchronisation counter state to the initial state 0.

6 Lessons Learned

The FFT circuits are representatives for a general class of circuits that compute complex functions without using a large amount of boolean control logic. In general, a few guidelines for proofs of circuit equivalence for such circuits can be drawn out of the FFT work:

– For each problem domain, it might be possible to find a small number of generalised proof scripts that can be powerful enough for a particular class of problems to make proofs automatic in most cases. These scripts should be parametrisable by proof options so that they not are too blunt to be reusable.
– As the proofs that have to be done when operations like arithmetic are involved are relatively complex, the prover's job must be simplified as much as possible. The use of automatic partial evaluation and formula reduction can in some cases lessen the need for prover inferences drastically. A tool like Lava that supports analyses like simplification of formulas by propositional reasoning and cone-of-influence analysis can help the designer simplify the problem at hand.
– It is not always necessary to explore the state space of a design. Ordinary induction can sometimes avoid very complex or intractable computations, and make for uncomplicated proofs.
– Normal form rewriting is a powerful technique that can be implemented very efficiently using modern rewrite engines. However, the use of unidirectional rules is crucial to make the strategy applicable to larger circuits.

7 Related Work

The Radix-2 FFT algorithm has previously been verified against the DFT using the ACL2 theorem prover [Gam98]. The level of abstraction in this verification was high and the proof thus required substantial user interaction. In contrast, we have aimed for fully automatic proofs, and verified the hardware FFTs at the netlist level. Our proofs are only for equivalence of fixed size circuits, but are not reliant on circuit regularity.

The pipeline proof principle bears some resemblance to the refinement mapping approach to pipelined microprocessor control verification [BD94,Cyr93]. However, as we are comparing a pipelined circuit against a combinational one, we cannot directly associate a single sequential step with the combinational implementation; we instead correlate whole frames. We also exploit the fact that constant lag pipelined circuits are targeted.

There are alternatives to Otter as a proof procedure: the Stanford validity checker decides quantifier free first order logic with linear arithmetic and uninterpreted functions by boolean case splitting (backtracking), rewrites and congruence closure [BDL96]. SVC has been used extensively in hardware verification, and is used as the decision procedure in the Burch and Dill approach to microprocessor verification [BD94]. Multiway decision graphs are a variation on the ROBDD theme that accommodates abstract data types, uninterpreted function symbols and rewrite rules [ZSC+95]; this data structure has been used to verify non-pipelined microprocessors and an ATM switch [TZS+96]. MDGs give a canonical representation for a fragment of quantifier free first-order formulas and support exploration of abstract state spaces (but do not guarantee convergence of fixpoint computations). As we have demonstrated, it is not always necessary to do such expensive computations; induction and normalising can be both sufficient and efficient.

Both MDGs and SVC need the user to provide rewrite rules or a normaliser for new theories. This means that the manual step of finding a normal form for twiddle factors is also necessary with these proof procedures.

8 Conclusions

This paper has shown how some FFT circuits have been verified from within the hardware development tool Lava after the existing system was extended with complex numbers and a general purpose strategy for equivalence checking of combinational and fixed lag pipelined circuits. The verification has been automatic in the sense that the only manual proof steps has been to select the proof procedure, rewrite rules and the initial state variables to restrict. The proofs are at a relatively low level, which should give a high confidence in the correctness of the modelled circuits; the logical formulas has been generated by symbolic evaluation of the hardware descriptions. No part of the verification has relied on the specific way that the arithmetic operators are implemented, or the representation of complex numbers. However, the proofs are not general in the size of the FFT; different instances have to be proved separately.

We have also presented an induction principle that exploits the problem structure of equivalence checking between a pipelined circuit and a combinational reference circuit, and contributed some suggestions for verification of circuits that contain little control logic but do complicated computations expressed in abstract operations.

9 Future Work

Lava is optimised for developing and verifying hardware. We pay for the strength we gain by limiting the problem domain, however, by presently being unable to reason internally about the proof strategies. Instead we have to go outside the system to a general purpose interactive theorem prover and do high level proofs there. We would like to have Lava integrated with a proof system that would allow us to do this kind of reasoning.

The counter examples that are produced by proof procedures are formatted and passed back to the user by Lava; unfortunately many first order logic theorem provers (including Otter) lack such capabilities. For verification with normal form rewriting to be smooth, it must be easy to find a rewriting theory quickly. It is therefore imperative to have some tool that analyses the output of a failed proof and allows the user to deduce what rules are missing, or gives the user good clues to why the two formulas are not equivalent. This is something that should (and will) be implemented in Lava as a proof analysis.

Acknowledgements

Thanks to Koen Claessen and Tanel Tammet for finding the Otter rewriting theory, and to Mary Sheeran and Byron Cook for careful readings of earlier drafts.

References

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *Proceedings of the third International Conference on Functional Programming.* ACM SIGPLAN, acm press, September 1998.

[BD94] Jerry Burch and David Dill. Automatic Verification of Microprocessor Control. In *Proceedings of the Computer Aided Verification Conference*, July 1994.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer Verlag, November 1996. Palo Alto, California, November 6–8.

[Cyr93] David Cyrluk. Microprocessor verification in PVS. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, December 1993.

[Gam98] Ruben Gamboa. Mechanically verifying the correctness of the Fast Fourier Transform in ACL2. In *Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, 1998.

[He95] Shousheng He. *Concurrent VLSI Architectures for DFT Computing and Algorithms for Multi-output Logic Decomposition.* PhD thesis, Lund Institute of Technology, 1995.

[MW97] William W. McCune and L. Wos. Otter: The CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[PM92] John Proakis and Dimitris Manolakis. *Digital Signal Processing.* Macmillan, 1992.

[Stå89] Gunnar Stålmarck. A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula, 1989. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995).

[Tam97] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[TZS+96] Sofiène Tahar, Zijian Zhou, Xiaoyu Song, Eduard Cerny, and Michel Langevin. Formal verification of an ATM switch fabric using Multiway Decision Graphs. In *IEEE Proceedings of Sixth Great Lakes Symposium on VLSI*, March 1996.

[VB98] Miroslav Velev and Randal Bryant. Bit-Level Abstraction in the Verification of Pipelined Microprocessors by Correspondence Checking. In *Formal Methods in Computer-Aided Design*, volume 1522 of *LNCS*, pages 18–35, Palo Alto, November 1998. Springer Verlag.

[ZSC+95] Zijian Zhou, Xiaoyu Song, Fransisco Corella, Eduard Cerny, and Michel Langevin. Description and Verification of RTL Designs Using Multiway Decision Graphs. In *Proceedings of the Conference on Hardware Description Languages and their applications*, August 1995.

A Appendix

A.1 The Radix-2 FFT Description

Figure 3 shows a size 16 Radix-2 FFT network, where merging arrows indicate addition and constants under a wire indicate multiplication. The Lava description of the size 2^n Radix-2 FFT circuit follows the network structure closely, and is parametrised by n:

```
radix2 n =
  bitRev n >-> compose [ stage i | i <- [1..n] ]
 where
  stage i = raised (n-i) two (twid i >-> bflys (i-1))
  twid  i = one (decmap (2^(i-1)) (wMult (2^i)))
```

The FFT circuit is made up from the sequential composition of an initial bit reversal permutation network (not shown in the picture) and n circuit stages. Stage i is a column of 2^{n-i} components that each contains a twiddle factor multiplication stage sequentially composed with a butterfly network. Given that $x = 2^{i-1}$, a size i multiplication stage performs multiplications with $W_{2^i}^0 ... W_{2^i}^{x-1}$ on the respective wires of one half of a bus, while passing the other half through unchanged.

More information on the signal processing building blocks and the descriptions of the combinational circuits can be found in [BCSS98].

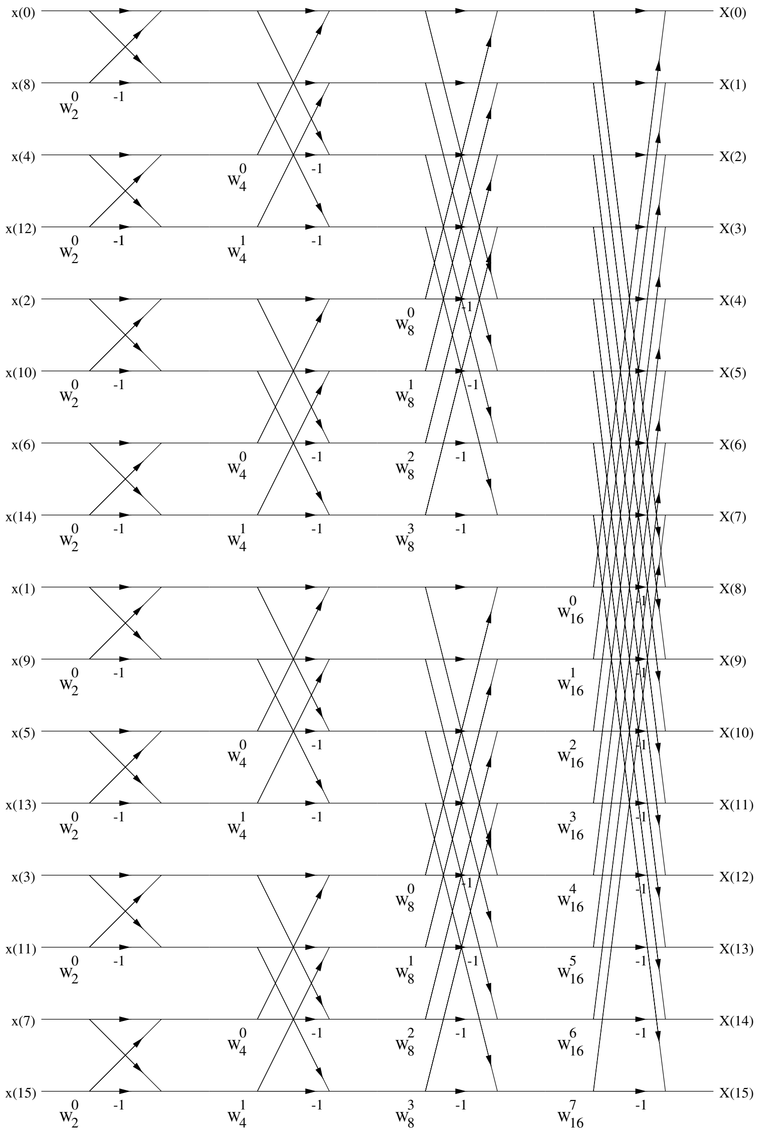

Fig. 3. The structure of a size 16 Radix-2 FFT