Verification of Benchmarks 17 and 22 of the IFIP WG10.5 Benchmark Circuit Suite

S. Hazelhurst and C.-J. H. Seger
Integrated Systems Design Laboratory
Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4

5 October 1995

Abstract

This paper reports on the verification of two of the IFIP WG10.5 benchmarks — the multiplier and systolic matrix multiplier. The circuit implementations are timed, detailed gate-level descriptions, and the specification is given using the temporal logic TL_n, a quaternary-valued temporal logic. A practical, integrated theorem-proving/model checking system based on the compositional theory for TL_n and symbolic trajectory evaluation is used to verify the circuits. A 64-bit version of multiplier circuit (Benchmark 17) containing approximately 28 000 gates takes about 18 minutes of computation time to verify. A 4×4, 32-bit version of the matrix multiplier (Benchmark 22) containing over 110 000 gates take about 170 minutes of computation time to verify. A significant timing error was discovered in this benchmark.
Keywords: symbolic trajectory evaluation, benchmarks, compositional verification, temporal logic, theorem proving.

1 Introduction

The IFIP WG10.5 Benchmark Circuit suite [3] is a suite of circuits designed to allow comparison between different verification sites. Descriptions of the circuits are available [7].

This paper reports on the verification of two circuits of the IFIP WG10.5 suite — one is a multiplier circuit, the other is a systolic matrix multiplier circuit — using a method of verification developed in the ISD Laboratory at the University of British Columbia. As this work is part of a larger project, this is a descriptive report of experiments undertaken. It is not a self-contained report and readers unfamiliar with symbolic trajectory evaluation will need the fuller account of the theory and discussion of the relationship to other work that can be found in work cited later. Much of the work. reported here forms the basis of the first author's PhD research.

Section 2 briefly describes the theoretical basis of the work. Section 3 describes the verification of Benchmark 17 (the multiplier) and Section 4 describes the verification of Benchmark 22 (the

This research was supported by operating grant OGPO 109688 from the Natural Sciences and Engineering Research Council of Canada, a fellowship from the B.C. Advanced Systems Institute, and by equipment grants from Sun Microsystems Canada Ltd. and Digital Equipment of Canada Ltd.

systolic matrix multiplier). In both cases, the circuits are implemented at a detailed gate-level, based on the description given in the suite. The verification includes the verification of timing properties of the circuit. A 64-bit version of multiplier circuit containing approximately 28 000 gates takes about 18 minutes of computation time to verify, while a 4×4, 32-bit version of the matrix multiplier containing over 110 000 gates take about 170 minutes of computation time to verify (in both cases a DEC Alpha 3000 was used for verification).

2 Theoretical background

2.1 Model and logic

Circuits are modelled by the model $(\langle \mathcal{S}, \sqsubseteq \rangle, \mathbf{Y})$, where:

- $\mathcal{S}$, the state space, is a complete lattice under the information ordering $\sqsubseteq$; and
- $\mathbf{Y}$, the next state function, describes the behaviour of the circuit.

For circuits, $\mathcal{S} = \mathcal{C}^n$, where n is the number of components in the circuit, and $\mathcal{C}$ is the lattice shown in Figure 1. L represents a low voltage, H represents a high voltage, X represents an unknown voltage, and Z represents an overconstrained value.

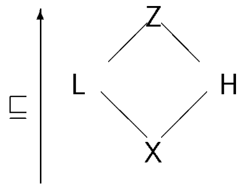

Figure 1: The state space $\mathcal{C}$

The circuits are verified by checking whether assertions of the form

$$\models_{\mathcal{M}} \langle\!| A \Longrightarrow g |\!\rangle$$

hold of the circuit. A and g are formulas of a temporal logic. A, the *antecedent*, describes the input or environment of the circuit, while g, the *consequent*, describes the output of the circuit. Informally $\models_{\mathcal{M}} \langle\!| A \Longrightarrow g |\!\rangle$ says that in all runs of the circuit, if A holds, then so does g.

A full description of the temporal logic can be found in an earlier technical report [1]. The novel feature of this temporal logic is that it is a four-valued logic $\{\bot, \mathbf{f}, \mathbf{t}, \top\}$ — these are *truth values*, not values of the circuit.

For the purpose of this technical report, the logic TL_n can be defined as follows. Let G_n be the smallest set with the following predicates:–

- The constant predicates: $\mathbf{f}, \mathbf{t}, \bot, \top \in G_n$;
- $\forall i \in \{1, \ldots, n\}, [i] \in G$.

Here $[i]$ refers to the i-th component of the state space. A formula g is evaluated with respect to a state by substituting for each $[i]$ which appears in the formula the value of the i-th component of the state. Formally,

- $[i](s) = \begin{cases} \perp & \text{when } s[i] \equiv \mathsf{X} \\ \mathbf{f} & \text{when } s[i] \equiv \mathsf{L} \\ \mathbf{t} & \text{when } s[i] \equiv \mathsf{H} \\ \top & \text{when } s[i] \equiv \mathsf{Z} \end{cases}$
- $\mathbf{f}(s) = \mathbf{f}$;
- $\mathbf{t}(s) = \mathbf{t}$;
- $\perp(s) = \perp$;
- $\top(s) = \top$;

The set temporal logic formulas of the logic TL_n is given by the following abstract syntax:

$$\mathrm{TL}_n ::= \; G_n \mid \mathrm{TL}_n \wedge \mathrm{TL}_n \mid \neg\mathrm{TL}_n \mid \texttt{Next}\,\mathrm{TL}_n \mid \mathrm{TL}_n\,\texttt{Until}\,\mathrm{TL}_n$$

Formulas of the logic are used to specify behaviour of the circuit. The question of verification is to ask whether the circuit behaviour satisfies these given formulas. We ask whether a sequence of states σ satisfies a formula g, written $\mathit{Sat}(\sigma, g)$. This is formally defined. Disjunction and implication can be defined as derived operators. If $\sigma = s_0 s_1 s_2 \ldots$ is a sequence of states, then $\sigma_{\geq i} = s_i s_{i+1} \ldots$.

Definition 2.1 (Semantics of TL_n).

1. If $g \in G_n$ then $\mathit{Sat}(\sigma, g) = g(s_0)$;
2. $\mathit{Sat}(\sigma, g \wedge h) = \mathit{Sat}(\sigma, g) \wedge \mathit{Sat}(\sigma, h)$;
3. $\mathit{Sat}(\sigma, \neg g) = \neg \mathit{Sat}(\sigma, g)$;
4. $\mathit{Sat}(\sigma, \texttt{Next}\, g) = \mathit{Sat}(\sigma_{\geq 1}, g)$;
5. $\mathit{Sat}(\sigma, g\,\texttt{Until}\,h) = \bigvee_{i=0}^{\infty} (\mathit{Sat}(\sigma_{\geq 0}, g) \wedge \ldots \wedge \mathit{Sat}(\sigma_{\geq i-1}, g) \wedge \mathit{Sat}(\sigma_{\geq i}, h))$.

$\square$

Although the formulas of the logic have a restricted syntax, we have shown that all monotonic predicates can be expressed within the logic.

Informally, $\models_{\mathcal{M}} \langle\!\langle\, g \Longrightarrow h \,\rangle\!\rangle$ says that all sequences that satisfy g with truth degree $\mathbf{t}$ also satisfy h with degree $\mathbf{t}$.

This presentation of TL_n has shown the scalar version of TL_n. This definition can be extended to a symbolic version, which by allowing variables to appear in expressions, makes the specifications much more concise.

2.2 Practical issues

To make specifications easier to write and read, some shorthands are introduced. The logic TL_n is defined for the set $\mathcal{C}^n$, which allows us to represent real circuits which have nodes taking boolean values. In many circuits, rather than considering the values of individual components, it is convenient to group a collection of components together, and consider the collection as a group. A common example is a circuit that manipulates numbers — integers or floating-point values. Rather than considering 64 bit-valued nodes, we consider one integer-valued nodes. Thus, we write

$$[A] + [B] = c + d + e$$

to ask whether the sum of the values on nodes A and B is equal to the sum of the integer variables c, d and e. Here A and B are shorthand for 64 bit-valued nodes; c, d and e are shorthand for 64 bit-valued variables; and $=$ and $+$ are shorthand for a the composition of many primitive operations such as conjunction and negation.

We also introduce the shorthand

$$\texttt{Global}\,[(s_0, t_0), \ldots, (s_n, t_n)]\; g = \bigwedge_{j=0}^{n} (\bigwedge_{k=s_j}^{t_j} \texttt{Next}\, k\;\, g),$$

which asks whether g holds between s_j and t_j for $j = 0, \ldots, n$.

2.3 Verification Methodology

The verification methodology proposed is the integrated use of theorem proving and model checking.

The model checking approach is based on the method of symbolic trajectory evaluation (STE) proposed by Seger and Bryant [6]. This work developed STE for a restricted temporal logic, *trajectory formulas* (TF). The Voss system [5] implements STE efficiently.

However, model checking has inherent limitations, and there are many circuits that model checking cannot deal with. Earlier work of ours [2] presented a compositional theory for TF, and showed that it provided a sound theoretical basis for developing a practical, integrated tool combining model checking and theorem-proving. The theorem prover implements a number of inference rules that the verifier can use to prove results. One of these inference rules is symbolic trajectory evaluation. The idea is that the verification problem will be broken down into a number of smaller sub-problems. The individual problems will be verified with STE; these results will then be combined using the other inference rules. Note that the circuit is not partitioned. While it is important to be able to identify some structure in the circuit for the compositional approach to be applicable, it is not necessary to decompose the circuit.

The verifier interacts with the theorem-prover using FL, an ML-like language; together with the use of heuristics, this gives the verifier a flexible and powerful tool for verification.

Recent work of ours [1] has extended STE to TL_n Not only has the theory been extended, but three different techniques for extending the STE-based algorithms were proposed: the direct, testing machine, and mapping methods

We have also extended the compositional theory (unreported so far). The basic compositional theory for TL_n is found in the table below. The side condition $\Delta^{\mathbf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(g_1)$ implies that every sequence that satisfies g_1 also satisfies g; the side condition $\Delta^{\mathbf{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(g_1) \sqcup \Delta^{\mathbf{t}}(h_1)$ implies that every sequences that satisfies both g_1 and h_1 will also satisfy g_2.

Name	Rule	Side condition
Identity	$\dfrac{}{\models_{\mathcal{M}} \langle\!\langle\, g \Longrightarrow g \,\rangle\!\rangle}$	
Time-shift	$\dfrac{\models_{\mathcal{M}} \langle\!\langle\, g \Longrightarrow h \,\rangle\!\rangle}{\models_{\mathcal{M}} \langle\!\langle\, \texttt{Next}\, t\ g \Longrightarrow \texttt{Next}\, t\ h \,\rangle\!\rangle}$	$t > 0$
Conjunction	$\dfrac{\models_{\mathcal{M}} \langle\!\langle\, g_1 \Longrightarrow h_1 \,\rangle\!\rangle \quad \models_{\mathcal{M}} \langle\!\langle\, g_2 \Longrightarrow h_2 \,\rangle\!\rangle}{\models_{\mathcal{M}} \langle\!\langle\, g_1 \wedge g_2 \Longrightarrow h_1 \wedge h_2 \,\rangle\!\rangle}$	
Disjunction	$\dfrac{\models_{\mathcal{M}} \langle\!\langle\, g_1 \Longrightarrow h_1 \,\rangle\!\rangle \quad \models_{\mathcal{M}} \langle\!\langle\, g_2 \Longrightarrow h_2 \,\rangle\!\rangle}{\models_{\mathcal{M}} \langle\!\langle\, g_1 \wedge g_2 \Longrightarrow h_1 \wedge h_2 \,\rangle\!\rangle}$	
Consequence	$\dfrac{\models_{\mathcal{M}} \langle\!\langle\, g \Longrightarrow h \,\rangle\!\rangle}{\models_{\mathcal{M}} \langle\!\langle\, g_1 \Longrightarrow h_1 \,\rangle\!\rangle}$	$\Delta^{\mathbf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(g_1), \Delta^{\mathbf{t}}(h_1) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(h)$
Transitivity	$\dfrac{\models_{\mathcal{M}} \langle\!\langle\, g_1 \Longrightarrow h_1 \,\rangle\!\rangle \quad \models_{\mathcal{M}} \langle\!\langle\, g_2 \Longrightarrow h_2 \,\rangle\!\rangle}{\models_{\mathcal{M}} \langle\!\langle\, g_1 \Longrightarrow h_2 \,\rangle\!\rangle}$	$\Delta^{\mathbf{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(g_1) \sqcup \Delta^{\mathbf{t}}(h_1)$
Specialisation	$\dfrac{\models_{\mathcal{M}} \langle\!\langle\, g \Longrightarrow h \,\rangle\!\rangle}{\models_{\mathcal{M}} \langle\!\langle\, \Xi(g) \Longrightarrow \Xi(h) \,\rangle\!\rangle}$	Ξ a specialisation.
Until	$\dfrac{\models_{\mathcal{M}} \langle\!\langle\, g_1 \Longrightarrow h_1 \,\rangle\!\rangle \quad \models_{\mathcal{M}} \langle\!\langle\, g_2 \Longrightarrow h_2 \,\rangle\!\rangle}{\models_{\mathcal{M}} \langle\!\langle\, g_1\, \texttt{Until}\, g_2 \Longrightarrow h_1\, \texttt{Until}\, h_2 \,\rangle\!\rangle}$	

The example verifications shown in the next two sections show the use of the compositional theory. These examples also can be used to evaluate the three extensions to the STE algorithms.

3 Verifying the multiplier

3.1 Description of circuit

Benchmark 17 of the IFIP WG10.5 Benchmark Suite is a multiplier which takes two n-bit numbers and returns a $2n$ bit number representing their multiplication. This description is heavily dependent on the IFIP documentation.[1]

Let $A = a_{n-1} \ldots a_1 a_0$ and $B = b_{n-1} \ldots b_1 b_0$. Then $A \times B = \sum_{i=0}^{n-1} 2^i (\sum_{j=0}^{n-1} 2^j a_i b_j)$. Implementing this is straightforward: the basic operation is multiplying one bit of A with one bit of B and adding this to the partial sum. The component which accomplishes this basic operation takes four inputs:

- a One bit of the multiplicand,
- b One bit of the multiplier,
- c One bit of the partial sum previously computed,
- CIN A one bit carry from the partial sum previously computed;

and computes computes $a * b + c + CIN$ producing two outputs:

- S One bit partial sum, and
- $COUT$ One bit carry.

The equations for the output are:

$$S = a \wedge b \oplus (c \oplus CIN)$$

$$COUT = a \wedge b \wedge c \vee a \wedge b \wedge CIN \vee c \wedge CIN$$

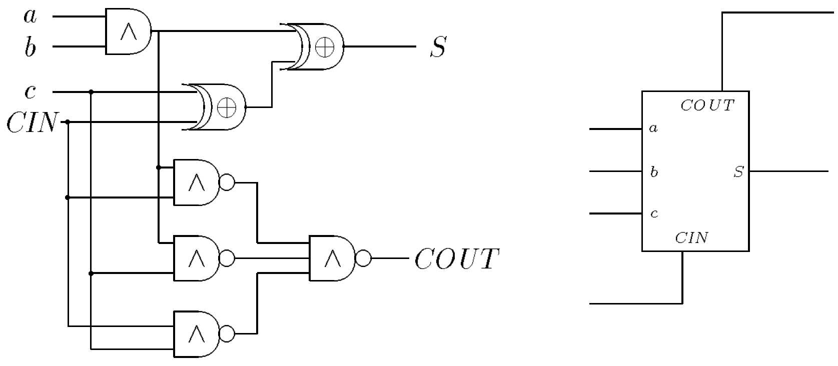

Figure 2: Base module for multiplier

The implementation of the equations (as given in the IFIP documentation) and the graphical symbol used to represent these components is presented in Figure 2.

A vector of these components multiplies one bit of B with the whole of A and adds in any partial answer already computed. It might seem appropriate rather than just having a vector of these components to also have an adder which added in carries from less significant columns to be added in to the results of more significant bits. The problem with doing that is that each stage would be limited by the need for possible carries from the least significant bit to be propagated to the most significant bit, with concomitant increases in the time and number of gates needed.

The approach used in the implementation is to produce two outputs: the first output is the sum of the bit-wise addition of the two inputs, ignoring the carries; and the second output is the carries of the bit-wise addition. Both of these outputs are forwarded to the next stage; here the carries are added in and new carries generated. We can consider the vector of S outputs as one n-bit number and the vector of $COUT$ outputs as another n-bit number. If we consider stage k by itself, if the vector of a inputs is $\tilde{x}$, if the b inputs are all the bit y, and if the vector of c inputs is $\tilde{z}$, then we shall have that $S + 2^{k+1}COUT = \tilde{x}y + z$. (This is something that must be proved in the verification.)

These components are arranged in a grid (Figure 3 shows how a 4 bit multiplier is arranged). The multiplier contains n stages, each of which multiplies one bit of B with A and adds it to the partial result computed so far. After k stages, $n+k$ bits of the partial answer have been computed. The components making up each stage are arranged in columns in the figure. The components making up a row compute one bit of the final answer; carries from less significant bits are added in, and any generated carries are output for the more significant rows to take care of.

In the Figure 3, each of the base components is labelled with indices: $i\colon j$ indicates that the component is the j-th component of the i-th stage.

Having passed through n stages, the full multiplication has been computed. However as the final stage still outputs two numbers, the carries must now all be added in. Therefore the final step in the multiplier is a row of $n-1$ full adders which adds in carries. These full adders are labelled **FA** in Figure 3.

[1] `ftp://goethe.ira.uka.de/pub/benchmarks/Multiplier/`

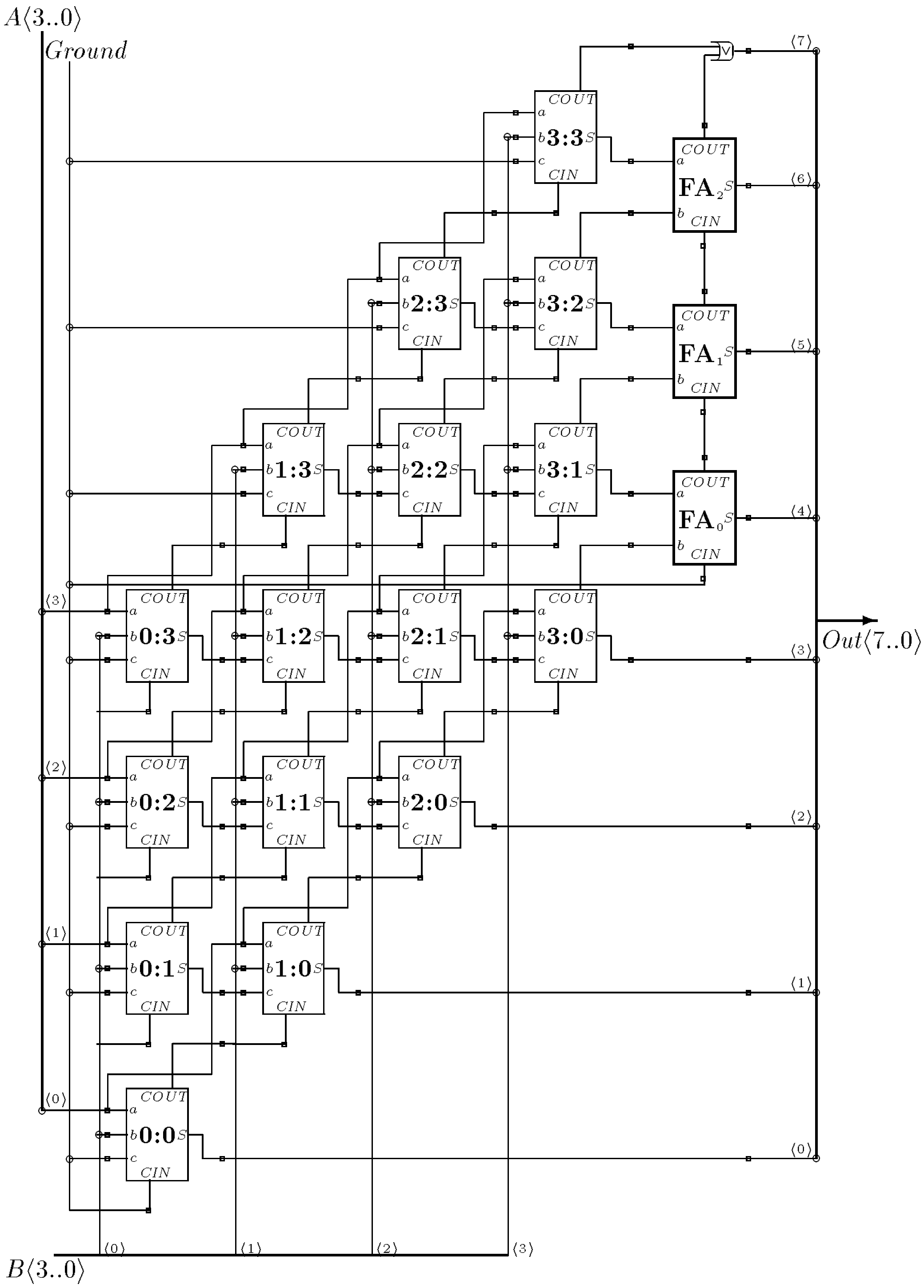

Figure 3: Layout of multiplier

Integer node	Vector of bit nodes
A	The four bit integer input
B	The four bit integer input
O	Output of the or gate
RS_i	S output of stage i for $i = 0, \ldots, 3$
	$\langle BM(i{:}\,3)(S), \ldots, BM(i{:}\,0)(S), BM(i-1:0)(S), \ldots, BM(0:0)(S)\rangle$
RC_i	$COUT$ output of stage i for $i = 0, \ldots, 3$
	$\langle BM(i{:}\,3)(COUT), \ldots, BM(i{:}\,0)(S)$
RS_4	The output Out
	$\langle O, FA_2, \ldots, FA_0, BM(3{:}\,0)(S), \ldots, BM(0{:}\,0)(S)\rangle$

Table 1: Benchmark 17: Correspondence between integer and bit nodes

The implementation of the circuit was done in Voss's EXE format as a detailed gate-level description of the circuit. A unit-delay model was used, although this is essential neither to the implementation nor the verification.

3.2 Verification

This section presents a detailed description of the verification of the four bit multiplier presented in Figure 3. This example is small enough that the complete proof can be described, and this is useful to show how the inference rules are used. However, the example is big enough that there is some tedium involved too; it must be emphasised that in practice the verification is done using FL as the proof script language, which alleviates much of the tedium.

It is also worth mentioning that the verification of a four bit multiplier is well within the capacity of trajectory evaluation to deal with. Although the proof is not independent of data path width since issues of timing are important, it may be useful to do the verification for a small bit width first using trajectory evaluation by itself.

Identifying structure Using the inference rules relies on using the properties of integers to break the limitations of BDDs. Therefore, the first step in the proof is to identify some structure, in particular to identify which collections of nodes should be treated as integers.

Notation: $BM(i{:}\,j)(x)$ refers to node x in the basic module $i{:}\,j$; $FA_i(x)$ refers to node x of the full adder FA_i. For each stage, we consider the collection of a inputs as an integer, the collection of b inputs as an integer, and so on ... Similarly, the collection of S outputs and $COUT$ outputs are both considered as integers. Table 1 presents the correspondences.

The following bit vectors *variables* are used:

a stands for the bit vector $\langle a_3, \ldots, a_0\rangle$;

b stands for the bit vector $\langle b_3, \ldots, b_0\rangle$ (a and b are the inputs to the circuit);

c stands for the bit vector $\langle c_7, \ldots, c_0\rangle$;

d stands for the bit vector $\langle d_2, \ldots, d_0\rangle$.

If N is a bit vector, then $N\langle i\rangle$ refers to the i-th least significant bit (so $N\langle 0\rangle$ is the least significant bit), and $N\langle i..j\rangle$ refers to the (sub)bit vector $\langle N\langle i\rangle, \ldots, N\langle j\rangle\rangle$. We also use the short hand that $RC_i = d$ is short for $RC_i\langle 2..0\rangle = d$ (RC_i is four bits wide, d is three bits wide).

Defining this correspondence has two advantages: the level of abstraction is raised since the verifier can think in terms of integers rather than bit vectors; and the verifier can use properties of integers to prove theorems without having to convert everything into BDDs.

Anomalies in circuit implementation There are a number of aspects of the circuit that can be criticised and improved. The most obvious is that $BM(i\colon 3)(COUT) = 0$ for all i. In turn, this means that one of the inputs to the or gate is always 0, i.e. $RS_4\langle 7\rangle$ depends entirely on $FA_2(COUT)$. The only advantage of this implementation is that it makes the circuit description (slightly) more regular. The cost is the extra circuitry and time required to perform the computation. Furthermore, this implementation makes the proof more complicated. The final step in the proof below will be to show that since $RS_3 + 2^4 RC_3 = ab$ that $RS_4 = ab$; this is only true because the one input to the or gate is zero. Therefore, as the proof is constructed, we shall prove that $BM(i, 3)(COUT) = 0$, complicating the proof slightly. A better implementation would have meant a simpler proof.

The Proof

Stage 0 The first step is to show the first stage performs the correct multiplication/addition.

$$\begin{array}{l}\models_{\mathcal{M}} \text{By STE} \\ \langle\!|\, \texttt{Global}\,[(0,100)]\,([A]=a \wedge [B]\langle 0\rangle = b_0) \\ \quad \Longrightarrow \quad \texttt{Global}\,[(3,100)]\,([RS_0] + 2^1[RC_0] = ab\langle 0\rangle \wedge [RC_0]\langle 3\rangle = 0)\,|\!\rangle\end{array} \tag{1}$$

To make STE as efficient as possible, we use as little information as possible by considering only one bit of b. However, at a later stage we shall want to use all the bits of b, so the next step is to include the rest of b in the result. There are a number ways of doing this. One would be to use the identity rule to show that B has any value imposed on it and then use conjunction with Result 1. However, in this case it is easier to use one of the rules of consequence and strengthen the antecedent.

$$\begin{array}{l}\models_{\mathcal{M}} \text{By rule of consequence from Result 1} \\ \langle\!|\, \texttt{Global}\,[(0,100)]\,([A]=a \wedge [B] = b) \\ \quad \Longrightarrow \quad \texttt{Global}\,[(3,100)]\,([RS_0] + 2^1[RC_0] = ab\langle 0\rangle \wedge [RC_0]\langle 3\rangle = 0)\,|\!\rangle\end{array} \tag{2}$$

This use of the rule of consequence is motivated by the fact that the antecedent of Result 2 uses more information than that of Result 1

Stage 1 The first step is to show Stage 1 performs the correct multiplication/addition. Note, the proof is done for arbitrary input for RS_0 and RC_0 rather than the actual input. This important because STE is used to do the proof; if the actual input (which is function of A and B) were used, in general STE would not be able to cope.

$$
\begin{array}{l}
\models_{\mathcal{M}} \text{ By STE} \\
\langle\!| \, \mathtt{Global}\,[(3,100)] \\
\qquad\qquad [A]=a \, \wedge [B]\langle 1\rangle = b_1 \, \wedge [RS_0]=c\langle 3..0\rangle \, \wedge [RC_0]=d \, \wedge [RC_0]\langle 3\rangle = 0 \\
\quad \Longrightarrow \quad \mathtt{Global}\,[(6,100)] \\
\qquad\qquad\qquad\qquad [RS_1] + 2^2[RC_1] = c\langle 3..0\rangle + 2^1 d + 2^1 ab\langle 1\rangle \, \wedge [RC_1]\langle 3\rangle = 0 \, |\!\rangle
\end{array}
\tag{3}
$$

In proving this result, STE is used; this implies that BDDs are used to represent data as this is necessary for STE. However, once the proof is done, the result is only stored symbolically, and the BDDs used to represent Result 3 are garbage collected.

Having proved this, we now combine Results 2 and 3 using a combination of transitivity and specialisation. This is useful to do since we know something about the values of RS_0 and RC_0; it is feasible to do since the consequent of Result 3 is strictly dependent on the nodes RS_0 and RC_0 — this means that Generalised Transitivity Theorem (GTT) can be used — informally, this says that $c\langle 3..0\rangle + 2^1 d = ab\langle 0\rangle$.

$$
\begin{array}{l}
\models_{\mathcal{M}} \text{ By Generalised Transitivity} \\
\langle\!| \, \mathtt{Global}\,[(0,100)] \; ([A]=a \, \wedge [B]=b) \\
\quad \Longrightarrow \quad \mathtt{Global}\,[(6,100)] \\
\qquad\qquad\qquad\qquad [RS_1] + 2^2[RC_1] = ab\langle 0\rangle + 2^1 ab\langle 1\rangle \, \wedge [RC_1]\langle 3\rangle = 0 \, |\!\rangle
\end{array}
\tag{4}
$$

Now we have the output of stage 1 solely in terms of a and b. This can be rewritten into a more elegant form. The proving system has integer rewriting procedures which automatically rewrites $ab\langle n-1..0\rangle + 2^n ab\langle n\rangle$ as $ab\langle n..0\rangle$. This is done by the theorem prover, and applying the rule of consequence yields the next result:

$$
\begin{array}{l}
\models_{\mathcal{M}} \text{ By rule of consequence} \\
\langle\!| \, \mathtt{Global}\,[(0,100)] \; ([A]=a \, \wedge [B]=b) \\
\quad \Longrightarrow \quad \mathtt{Global}\,[(6,100)] \\
\qquad\qquad\qquad\qquad [RS_1] + 2^2[RC_1] = ab\langle 1..0\rangle \, \wedge [RC_1]\langle 3\rangle = 0 \, |\!\rangle
\end{array}
\tag{5}
$$

Stages 2 and 3 The steps are exactly the same as stage 1.

$$
\begin{array}{l}
\models_{\mathcal{M}} \text{ By STE} \\
\langle\!| \, \mathtt{Global}\,[(6,100)] \\
\qquad\qquad [A]=a \, \wedge [B\langle 2\rangle] = b_2 \, \wedge [RS_1]=c\langle 4..0\rangle \, \wedge [RC_1]=d \, \wedge [RC_1]\langle 3\rangle = 0) \\
\quad \Longrightarrow \quad \mathtt{Global}\,[(9,100)] \\
\qquad\qquad\qquad\qquad [RS_2] + 2^3[RC_2] = c\langle 4..0\rangle + 2^2 d + 2^2 ab\langle 2\rangle \, \wedge [RC_2]\langle 3\rangle = 0 \, |\!\rangle
\end{array}
\tag{6}
$$

$$
\begin{array}{l}
\models_{\mathcal{M}} \text{ By Generalised Transitivity (Results 5 and 6)} \\
\langle\!| \, \mathtt{Global}\,[(0,100)] \; ([A]=a \, \wedge [B]=b) \\
\quad \Longrightarrow \quad \mathtt{Global}\,[(9,100)] \\
\qquad\qquad\qquad\qquad [RS_2] + 2^3[RC_2] = ab\langle 1..0\rangle + 2^3 ab\langle 2\rangle \, \wedge [RC_2]\langle 3\rangle = 0 \, |\!\rangle
\end{array}
\tag{7}
$$

$\models_{\mathcal{M}}$ By rule of consequence from Result 7

$$\langle\!\langle\, \texttt{Global}\,[(0,100)]\,([A]\!=\!a \wedge [B]\!=\!b) \\ \Longrightarrow \quad \texttt{Global}\,[(9,100)] \\ [RS_2] + 2^3[RC_2]\!=\!ab\langle 2..0\rangle \wedge [RC_2]\langle 3\rangle\!=\!0 \,\rangle\!\rangle \tag{8}$$

$\models_{\mathcal{M}}$ By STE

$$\langle\!\langle\, \texttt{Global}\,[(9,100)] \\ [A]\!=\!a \wedge [B]\langle 3\rangle\!=\!b_3 \wedge [RS_2]\!=\!c\langle 5..0\rangle \wedge [RC_2]\!=\!d \wedge [RC_2]\langle 3\rangle\!=\!0 \\ \Longrightarrow \quad \texttt{Global}\,[(12,100)] \\ [RS_3] + 2^4[RC_3]\!=\!c\langle 5..0\rangle + 2^3d + 2^3ab\langle 3\rangle \wedge [RC_3]\langle 3\rangle\!=\!0 \,\rangle\!\rangle \tag{9}$$

$\models_{\mathcal{M}}$ By Generalised Transitivity (Results 8 and 9)

$$\langle\!\langle\, \texttt{Global}\,[(0,100)]\,([A]\!=\!a \wedge [B]\!=\!b) \\ \Longrightarrow \quad \texttt{Global}\,[(12,100)] \\ [RS_3] + 2^4[RC_3]\!=\!ab\langle 2..0\rangle + 2^3ab\langle 3\rangle \wedge [RC_3]\langle 3\rangle\!=\!0 \,\rangle\!\rangle \tag{10}$$

$\models_{\mathcal{M}}$ By By rule of consequence from Result 10

$$\langle\!\langle\, \texttt{Global}\,[(0,100)]\,([A]\!=\!a \wedge [B]\!=\!b) \\ \Longrightarrow \quad \texttt{Global}\,[(12,100)] \\ [RS_3] + 2^4[RC_3]\!=\!ab \wedge [RC_3]\langle 3\rangle\!=\!0 \,\rangle\!\rangle \tag{11}$$

The adder stage The final step in the proof is to ensure that the last, adder stage, adds in the carries correctly. Here possible carries in the least significant bit must be passed to the most significant bit. For large bit widths, this adder stage may take tens or hundreds of nanoseconds, so timing may be important here.

$\models_{\mathcal{M}}$ By STE

$$\langle\!\langle\, \texttt{Global}\,[(12,100)]\,([RS_3]\!=\!c\langle 6..0\rangle \wedge [RC_3]\!=\!d \wedge [RC_3]\langle 3\rangle\!=\!0) \\ \Longrightarrow \quad \texttt{Global}\,[(22,100)]\,([RS_4]\!=\!(c\langle 6..0\rangle + 2^4d)\langle 7..0\rangle) \,\rangle\!\rangle \tag{12}$$

Now, using general transitivity, we have:

$\models_{\mathcal{M}}$ By Generalised Transitivity (Results 11 and 12)

$$\langle\!\langle\, \texttt{Global}\,[(0,100)]\,([A]\!=\!a \wedge [B]\!=\!b) \\ \Longrightarrow \quad \texttt{Global}\,[(22,100)]\,([RS_4]\!=\!ab) \,\rangle\!\rangle \tag{13}$$

Again, the automatic rewrite systems recognises that ab is an eight bit number, and so rewrites $ab\langle 7..0\rangle$ as ab. This concludes the proof.

The appendix has the FL proof script for the multiplier example.

Bit width	Number of gates	D Time (s)	T Time (s)
4	135	3.9	5.4
8	473	9.8	15.0
16	1841	36.0	60.8
32	7265	168.7	371.4
64	28865	1081.9	> 6000

Table 2: Verification times for Benchmark 17 multiplier

Experimental results and comments This IFIP WG10.5 Benchmark 17 multiplier was verified for a number of bit widths (the n bit width case multiplies two n-bit numbers and produces a $2n$ bit number). The time taken to perform the verification on a DEC Alpha 3000 is shown in Table 2: the column labelled 'D Time' shows the time taken using the direct method, and the column labelled 'T Time' shows the time taken using the testing machine approach (all times shown in seconds). These results are useful for evaluating the testing machine approach.

The proof script itself is short (250 lines, about 100 of which are declarations) and straightforward to write, relying only on simple properties of integers. Once structure in the circuit is identified by associating integers with collections of bit valued nodes, the verification no longer has to deal with bits, and at no stage does the verification have to concern itself with how the full adders or the base components are actually implemented.

The reason why STE cannot deal with the verification by itself is not because of the size of the circuit; the problem is that there is no good variable ordering for the multiplication of two bit vectors. However, good variable orderings are definitely possible for verifying the individual components of the multiplier with STE, and good heuristics to find good ordering can easily be automated.

4 Matrix Multiplier

A filter circuit based on a design of Mead and Conway is Benchmark 22 of the IFIP WG10.5 suite [4]. The filter is a matrix multiplication circuit for band matrices. A band matrix of band width w is a matrix in which zeros must be in certain positions (the matrices contain natural numbers), and the maximum number of non-zero items in a row or column is w. This circuit is called 2Syst. Section 4.1 discusses the specification of the circuit; Section 4.2 discusses its implementation; Section 4.3 presents its verification; and Section 4.4 analyses and comments on the verification. Sections 4.1 and 4.2 rely heavily on the benchmark documentation.[2]

4.1 Specification

The suite documentation does not give a general specification of the circuit (it does give a general implementation), but presents the case of $w = 4$. A circuit implemented for a band-width of w can be used to multiply matrices of any size — larger matrices just take longer to multiply; the documentation does not consider the general case, and gives only specification for 4×4 matrices.

[2] `ftp://goethe.ira.uka.de/pub/benchmarks/2Syst/`

Let A and B be the two 4×4 matrices given below:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} \\ 0 & b_{32} & b_{33} & b_{34} \\ 0 & 0 & b_{43} & b_{44} \end{bmatrix},$$

and let $C = A \times B$ be the matrix:

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

The external interface of the 2Syst circuit is shown in Figure 4. The coefficients of matrix A are input on the inputs `a0`, ..., `a3`, the coefficients of B are input on `b0`, ..., `b3`, and the coefficients of C, the result, is output in outputs `c0` to `c6`. (What this picture, taken from the documentation, does not show is that the circuit is clocked and there should be a pin for clock input too.)

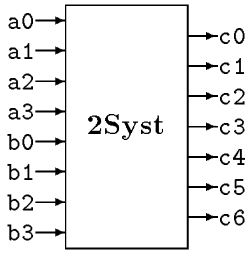

Figure 4: Black box view of 2Syst

Timing

The timing of when and where the inputs must be applied and the outputs become available is critical. The timing for the inputs is presented in Table 3. In clock cycles 0 to 3, all the inputs are initialised by having zero applied to them. Then, for the next ten cycles the matrix coefficients are input to the circuit. For example, in cycle 9, the coefficients a_{23}, a_{42}, b_{32} and b_{42} are input on pins `a0`, `a3`, `b0`, and `b3` respectively, while all other pins have zero applied to them.

Table 4 shows when and where the coefficients of the output matrix can be found. The specification gives some freedom in timing here. It requires that the output be given in clock cycles $t_0, \ldots, t_6$, but does not specify values for the t_j; and, while $t_0 < t_1 \ldots < t_6$, the t_j need not be consecutive clock cycles. This gives some latitude in the implementation of the circuit.

4.2 Implementation

The matrix multiplication $C = A \times B$ can be defined in different ways. Assuming for simplicity that A and B are both $r \times r$ matrices, the usual definition of C is through defining each $c_{ij} = \sum_{k=1}^{r} a_{ik} b_{ki}$.

clock	a0	a1	a2	a3	b0	b1	b2	b3
$0-3$	0	0	0	0	0	0	0	0
4	0	a_{11}	0	0	0	b_{11}	0	0
5	0	0	a_{21}	0	0	0	b_{12}	0
6	a_{12}	0	0	a_{31}	b_{21}	0	0	b_{13}
7	0	a_{22}	0	0	0	b_{22}	0	0
8	0	0	a_{32}	0	0	0	b_{23}	0
9	a_{23}	0	0	a_{42}	b_{32}	0	0	b_{24}
10	0	a_{33}	0	0	0	b_{33}	0	0
11	0	0	a_{43}	0	0	0	b_{34}	0
12	a_{34}	0	0	0	b_{43}	0	0	0
13	0	a_{44}	0	0	0	b_{44}	0	0

Table 3: Inputs for the 2Syst circuit

cycle	c0	c1	c2	c3	c4	c5	c6
t_0				c_{11}			
t_1			c_{12}		c_{21}		
t_2		c_{13}		c_{22}		c_{31}	
t_3	c_{14}		c_{23}		c_{32}		c_{41}
t_4		c_{24}		c_{33}		c_{42}	
t_5			c_{34}		c_{43}		
t_6				c_{44}			

Table 4: Outputs of the 2Syst circuit

An alternative definition is useful in implementing parallel hardware to perform the multiplication: matrix multiplication can also be defined by the recursive equation 14.

$$\begin{aligned} c_{ij}^{(1)} &= 0 \\ c_{ij}^{(k+1)} &= c_{ij}^{(k)} + a_{ik}b_{kj} \\ c_{ij} &= c_{ij}^{(r+1)} \end{aligned} \tag{14}$$

The entries in arrays A and B are n-bit numbers. If the band-width of the matrices is w, the maximum number of non-zero terms in any c_{ij} is w, which means that each entry in c_{ij} is of bit-width $m = 2n + r - 1$.

The basic operation of Equation 14 is performing an addition and a multiplication; this is modelled in the implementation, where the basic cell has an integer multiplier and adder to perform this. The external interface of these cells is shown in Figure 5. The cell has three inputs: `C_In` is an m bit number, containing a partial sum; and `A_In` and `B_In` are n bit data which are either zero or coefficients of the A and B matrices. `A_Out`, `B_Out` are two n-bit output values and `C_Out` is an m-bit output. If in one clock cycle `A_In`, `B_In` and `C_In` have the values a, b and c respectively, then at the start of the next cycle: `A_Out` $= a$, `B_Out` $= b$, `C_Out` $= ab + c$.

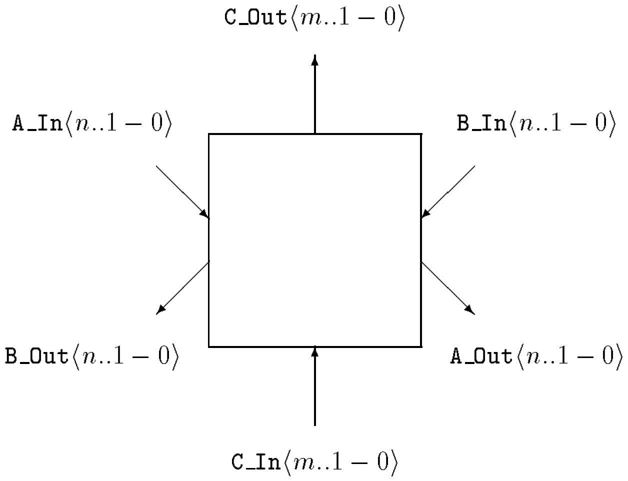

Figure 5: Cell representation

Thus, the cell has two purposes: it acts as a one clock-cycle delay buffer for coefficients of the matrices (which are passed on to neighbouring cells), and performs the basic operation of an addition and multiplication.

Figure 6 shows how the cells are implemented. Each cell contains a multiplier, an adder, and three registers. The multiplier is the one discussed and verified in the previous section, and the adder is a conventional $2n$-bit adder. Each register has an input, an output, and a clock and select pin. By connecting the select and clock pins to the same global clock, the registers become positive-edge triggered: when the clock rises the value at the register's input is latched, output, and maintained until the clock rises again.

These cells are connected in a systolic array: each clock cycle cells performs an addition and multiplication and then passes its results to its neighbours for use in the next cycle. The cells are arranged as presented in Figure 7, and the timings given in Table 3 are designed so that cells get the right inputs at the right time. A simple example will illustrate how this works. To help the description, each cell in the systolic array has been labelled by $i: j$.

The circuit is implemented in Voss's EXE format as a detailed gate level description, using a unit delay model. The implementation is based on the VHDL program given in the benchmark suite documentation.

Example 4.1.
Consider the computation of $c_{21} = a_{21}b_{11} + a_{22}b_{21}$. In the first three clock cycles the circuit is initialised so that at the start of the fourth cycle, all inputs have value zero.

Cycle 4: b_{11} is input on `b1` (input `B_In` of Cell 1:0). (a_{11} is also input in this cycle, but in the example, we only consider values contributing to c_{21}).

Cycle 5: Cell 1:0 will have passed b_{11} to its neighbour, so that b_{11} now becomes an input for Cell 1:1. a_{21} is input on `a2` (the `A_In` input of Cell 0:2).

Cycle 6: Cell 1:1 will have passed b_{11} to the `B_In` input of Cell 1:2, and Cell 0:2 will have passed a_{11} to the `A_In` input of Cell 1:2. At this stage, the `C_In` input of Cell 1:2 has the value 0. Cell

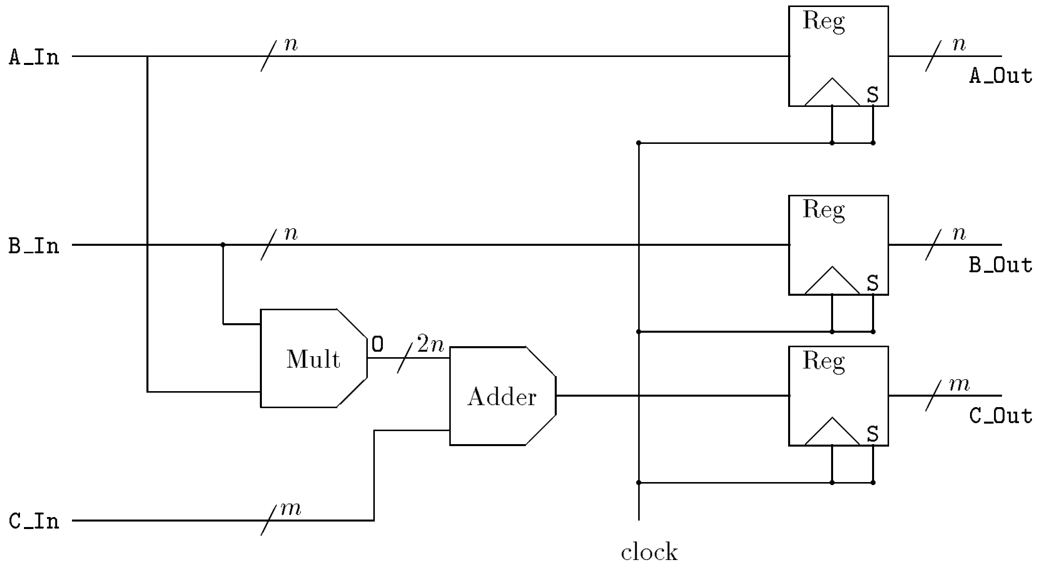

Figure 6: Implementation of cell

1:2 therefore computes $a_{11}b_{11}$. At the same time, b_{21} appears as input on `b0`, which is input `B_In` of Cell 0:0.

Cycle 7: Cell 1:2 will have passed $a_{11}b_{11}$ to Cell 0:1 as its `C_In` input. Cell 0:0 will have passed on b_{21} to Cell 0:1 as its `B_In` input. a_{22} appears on `a1`, which is the `A_In` input of Cell 0:1. Cell 0:1 computes $a_{11}b_{11} + a_{22}b_{21}$.

Cycle 8: Cell 0:1 outputs $a_{11}b_{11} + a_{22}b_{21}$ on its `C_Out` port (which is `c4`).

4.3 Verification

The verification task can be divided into two parts, the verification of the individual components, and using the verification of the components to show that whole array is correct.

Verifying the cells

The verification of a cell must show the multiplier, adder and registers all work correctly. Each cell must be verified individually. This section describes the verification of Cell u:v, and assumes for the sake of this exposition that the clock cycle is 200ns, and the bit-width is 4.

In the discussion below, the $\texttt{A\_In}_{uv}$ and $\texttt{B\_In}_{uv}$ are four-bit nodes, while all variables are 12 bit values. To simplify notation, in all the discussion below, a and b are short hands for $a\langle 3..0\rangle$ and $b\langle 3..0\rangle$ respectively.

It turns out that it useful to divide this proof into three parts:

- Given value a on $\texttt{A\_In}_{uv}$, b on $\texttt{B\_In}_{uv}$, and c on $\texttt{C\_In}_{uv}$, one clock cycle later $ab + c$ appears on $\texttt{C\_Out}_{uv}$;

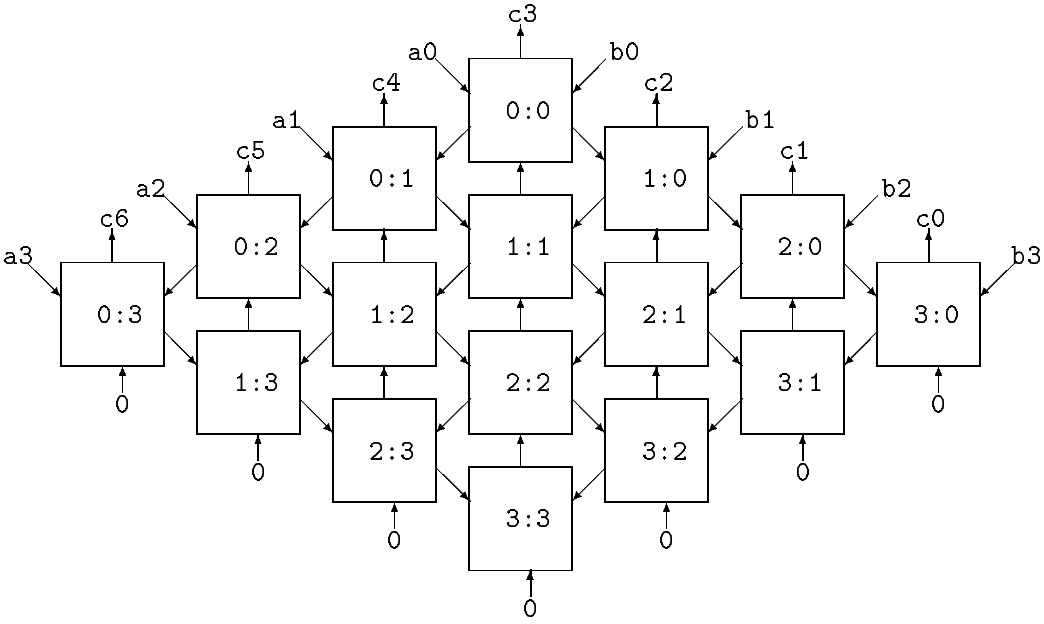

Figure 7: Systolic array

- Given value a on $\texttt{A\_In}_{uv}$, one cycle later a appears on $\texttt{A\_Out}_{uv}$; and
- Given value b on $\texttt{B\_In}_{uv}$, one cycle later b appears on $\texttt{B\_Out}_{uv}$.

When the cells are connected together, port $\texttt{C\_In}_{uv}$ is connected to $\texttt{C\_Out}_{(u+1)(v+1)}$, port $\texttt{A\_Out}_{uv}$ is connected to $\texttt{A\_In}_{u(v+1)}$, and $\texttt{B\_Out}_{uv}$ is connected to $\texttt{B\_In}_{(u+1)v}$. Therefore, the above verification conditions are rewritten as:

- Given value a on $\texttt{A\_In}_{uv}$, b on $\texttt{B\_In}_{uv}$, and c on $\texttt{C\_Out}_{(u+1)(v+1)}$, one clock cycle later $ab+c$ appears on $\texttt{C\_Out}_{uv}$;
- Given value a on $\texttt{A\_In}_{uv}$, one cycle later a appears on $\texttt{A\_In}_{u(v+1)}$; and
- Given value b on $\texttt{B\_In}_{uv}$, one cycle later b appears on $\texttt{B\_In}_{(u+1)v}$.

Of course, it is possible to combine all three into a one, stronger result. However, having three weaker results makes the proof more flexible since at some stages the proof needs only the weaker result, and using a stronger result would clutter things up and be more inefficient.

The costliest part of the proof is to show the multiplier works correctly. As Section 3 showed how the Benchmark 17 multiplier can be verified, for the purpose of this section, Result 15 is assumed (in the actual verification, the multiplier for each cell is reverified).

$$\begin{array}{l} \models_{\mathcal{M}} \text{ By various rules} \\ \langle\!\langle\, \texttt{Global}\,[(0,100)]\ ([\texttt{A\_In}_{uv}]=a\ \wedge\ [\texttt{B\_In}_{uv}]=b) \\ \quad \Longrightarrow\quad \texttt{Global}\,[(22,100)]\ ([\texttt{O}_{uv}]=ab)\,\rangle\!\rangle \end{array} \tag{15}$$

In the cell, the clock has an important effect; to include information of when clocking happens, the rule of consequence is often used to strengthen the antecedent of a result. For convenience, let

$$\begin{aligned} Clock_k \;\; = \;\; & \mathtt{Global}\,[(200k, 200k+99), (200(k+1), 200(k+1)+99)]\;\; ([\mathtt{clock}]\!=\!\mathbf{f}) \;\wedge \\ & \mathtt{Global}\,[(200k+100, 200k+199)]\;\; ([\mathtt{clock}]\!=\!\mathbf{t}) \end{aligned}$$

which is the information about clocking which is needed in the proof of the k-th cycle.

Using this idea, Result 15 is transformed strengthening the antecedent, as well as taking into account the input on `C_In`. Although, this is not useful for its own sake, it is useful in using the essence of Result 15.

$$\begin{aligned} & \models_{\mathcal{M}} \text{ By Rule of consequence} \\ & \langle\!\langle\, \mathtt{Global}\,[(0,100)]\;\; [\mathtt{A\_In}_{uv}]\!=\!a \;\wedge\; [\mathtt{B\_In}_{uv}]\!=\!b \;\wedge\; [\mathtt{C\_Out}_{(u+1)(v+1)}]\!=\!c \;\wedge\;\; Clock_0 \\ & \quad \Longrightarrow \quad \mathtt{Global}\,[(22,100)]\;\; ([\mathtt{O}_{uv}]\!=\!ab)\,\rangle\!\rangle \end{aligned} \tag{16}$$

In the next step we show that the adder works correctly and that the output of the adder is latched for the appropriate time. This can be done with one trajectory evaluation. Note that the time interval in the consequent could be made bigger, but the one given suffices.

$$\begin{aligned} & \models_{\mathcal{M}} \text{ By STE} \\ & \langle\!\langle\, \mathtt{Global}\,[(22,100)]\;\; ([\mathtt{O}_{uv}]\!=\!d\langle 7..0\rangle \;\wedge\; [\mathtt{C\_Out}_{(u+1)(v+1)}]\!=\!c \;\wedge\;\; Clock_0) \\ & \quad \Longrightarrow \quad \mathtt{Global}\,[(200,300)]\;\; ([\mathtt{C\_Out}_{uv}]\!=\!c + d\langle 7..0\rangle)\,\rangle\!\rangle \end{aligned} \tag{17}$$

Results 16 and 17 are now combined by specialising the latter result (substituting ab for d), and using transitivity. Note that this is just a special case of General Transitivity.

$$\begin{aligned} & \models_{\mathcal{M}} \text{ By GTT} \\ & \langle\!\langle\, \mathtt{Global}\,[(0,100)]\;\; ([\mathtt{A\_In}_{uv}]\!=\!a \;\wedge\; [\mathtt{B\_In}_{uv}]\!=\!b \;\wedge\; [\mathtt{C\_Out}_{(u+1)(v+1)}]\!=\!c \;\wedge\;\; Clock_0) \\ & \quad \Longrightarrow \quad \mathtt{Global}\,[(200,300)]\;\; ([\mathtt{C\_Out}_{uv}]\!=\!c + ab)\,\rangle\!\rangle \end{aligned} \tag{18}$$

Result 18 is the core result that has to be proved about the cell. The next two results show that the cell also acts as one cycle delay buffers for values of the A and B matrices. Both of these results can easily be done using STE alone.

$$\begin{aligned} & \models_{\mathcal{M}} \text{ By STE} \\ & \langle\!\langle\, \mathtt{Global}\,[(0,100)]\;\; ([\mathtt{A\_In}_{uv}]\!=\!a \;\wedge\;\; Clock_0) \\ & \quad \Longrightarrow \quad \mathtt{Global}\,[(200,300)]\;\; ([\mathtt{A\_In}_{u(v+1)}]\!=\!a)\,\rangle\!\rangle \end{aligned} \tag{19}$$

$$\begin{aligned} & \models_{\mathcal{M}} \text{ By STE} \\ & \langle\!\langle\, \mathtt{Global}\,[(0,100)]\;\; ([\mathtt{B\_In}_{uv}]\!=\!b \;\wedge\;\; Clock_0) \\ & \quad \Longrightarrow \quad \mathtt{Global}\,[(200,300)]\;\; ([\mathtt{B\_In}_{(u+1)v}]\!=\!b)\,\rangle\!\rangle \end{aligned} \tag{20}$$

Overall verification

Once each of the cells has been individually verified, the proofs about the individual cells must be combined to prove that the systolic array as a whole works correctly.

The proof is modelled on how the systolic array computes its results; in its development the proof traces the behaviour of the circuit as it uses its inputs, computes results, and outputs the answers.

Consider the operation of one cell, Cell $u{:}v$. It has three *input* neighbours from which it gets values (the boundary cells are special cases and easily taken care of):

- Cell $u{:}(v-1)$, its A-left-neighbour from which it gets a value of the A matrix,
- Cell $(u-1){:}v$, its B-right-neighbour from which it gets a value of the B matrix, and
- Cell $(u+1){:}(v+1)$ its C-down-neighbour from which it gets a partial sum;

and three *output* neighbours to which it gives values:

- Cell $u{:}(v+1)$, its A-right-neighbour, to which it gives a value of the A matrix,
- Cell $(u+1){:}v$, its B-left-neighbour, to which it gives a value of the B matrix, and
- Cell $(u-1){:}(v-1)$ its C-up-neighbour, to which it gives a partial sum;

At the beginning of clock cycle k, none, some, or all of the following will be known about Cell $u{:}v$'s input neighbours (recall that a clock cycle is 200 time units long), where the I_j are antecedent TL formulas, and the θ_x are integer expressions:

$$\models_{\mathcal{M}} \langle\!|\, I_1 \Longrightarrow \texttt{Global}\,[(200k, 200k+100)]\ [\texttt{A\_In}_{uv}] = \theta_a \,|\!\rangle \tag{21}$$

$$\models_{\mathcal{M}} \langle\!|\, I_2 \Longrightarrow \texttt{Global}\,[(200k, 200k+100)]\ [\texttt{B\_In}_{uv}] = \theta_b \,|\!\rangle \tag{22}$$

$$\models_{\mathcal{M}} \langle\!|\, I_1 \Longrightarrow \texttt{Global}\,[(200k, 200k+100)]\ [\texttt{C\_Out}_{(u+1)(v+1)}] = \theta_c \,|\!\rangle \tag{23}$$

If all three results are known, then we use conjunction on Results 21–23, and introduce new clocking information. For convenience, let

$$I_4 = I_1 \wedge I_2 \wedge I_3 \wedge \mathit{Clock}_k.$$

This is the conjunction of I_1, I_2 and I_3 and contains necessary clocking information for the k-th cycle. Then we have:

$$\begin{array}{l}\models_{\mathcal{M}} \text{ By Conjunction and Rule of Consequence} \\ \langle\!|\, I_4 \\ \quad \Longrightarrow \quad \texttt{Global}\,[(200k, 200k+100)] \\ \qquad\qquad [\texttt{A\_In}_{uv}] = \theta_a \wedge [\texttt{B\_In}_{uv}] = \theta_b \wedge [\texttt{C\_Out}_{uv}] = \theta_c . \,|\!\rangle \end{array} \tag{24}$$

Then Result 18 is time-shifted forward by k-clock cycles to get:

$$\begin{array}{l}\models_{\mathcal{M}} \text{ By time-shifting} \\ \langle\!|\, \texttt{Global}\,[(200k, 200k+100)] \\ ([\texttt{A\_In}_{uv}] = a \wedge [\texttt{B\_In}_{uv}] = b \wedge [\texttt{C\_Out}_{(u+1)(v+1)}] = c \wedge \mathit{Clock}_k) \\ \quad \Longrightarrow \quad \texttt{Global}\,[(200(k+1), 200(k+1)+100)]\ ([\texttt{C\_Out}_{uv}] = c + ab) \,|\!\rangle \end{array} \tag{25}$$

Using General Transitivity on Results 24 and 25 leads to:

$$\begin{array}{ll} \models_{\mathcal{M}} \text{ By GTT} & \\ \langle\!| \, I_4 & (26) \\ \quad \Longrightarrow \quad \texttt{Global}\,[(200(k+1), 200(k+1)+100)] \;\; ([\texttt{C\_Out}_{uv}] = \theta_c + \theta_a\theta_b) \, |\!\rangle & \end{array}$$

This is a proof of what Cell u:v computes in the k-th cycle. In proving what happens in the $(k+1)$-th cycle, Result 26 is used in the proof of the behaviour of Cell $(u+1)$:$(v+1)$, which is Cell u:v's up-C-neighbour.

Similarly, if Result 21 is known, then precondition strengthening is used to introduce new clocking information to get:

$$\begin{array}{ll} \models_{\mathcal{M}} \text{ By Rule of consequence} & \\ \langle\!| \, I_1 \wedge Clock_k & (27) \\ \quad \Longrightarrow \quad \texttt{Global}\,[(200k, 200k+100)] \;\; [\texttt{A\_In}_{uv}] = \theta_a \, |\!\rangle & \end{array}$$

Then Result 19 is time-shifted by k clock cycles to get:

$$\begin{array}{ll} \models_{\mathcal{M}} \text{ By STE} & \\ \langle\!| \, \texttt{Global}\,[(200k, 200k+100)] \;\; ([\texttt{A\_In}_{uv}] = a) \wedge Clock_k & (28) \\ \quad \Longrightarrow \quad \texttt{Global}\,[(200(k+1), 200(k+1)+100)] \;\; ([\texttt{A\_In}_{u(v+1)}] = a) \, |\!\rangle & \end{array}$$

General Transitivity between Results 27 and 28 then yields:

$$\begin{array}{ll} \models_{\mathcal{M}} \text{ By Rule of consequence} & \\ \langle\!| \, I_1 \wedge Clock_k & (29) \\ \quad \Longrightarrow \quad \texttt{Global}\,[(200(k+1), 200(k+1)+100)] \;\; ([\texttt{A\_In}_{u(v+1)}] = \theta_a) \, |\!\rangle & \end{array}$$

This shows what Cell u:v passes to its A-right neighbour at the end of the k-th cycle, and this result will be used to prove properties of Cell u: $(v+1)$ in the $(k+1)$-th cycle. A similar result shows that in the k-th Cell u:v also passes on the value input on its `B_In` port,

$$\begin{array}{ll} \models_{\mathcal{M}} \text{ By various rules} & \\ \langle\!| \, I_2 \wedge Clock_k & (30) \\ \quad \Longrightarrow \quad \texttt{Global}\,[(200(k+1), 200(k+1)+100] \;\; ([\texttt{B\_In}_{(u+1)v)}] = \theta_b) \, |\!\rangle & \end{array}$$

FL Proof script The FL proof script which performs the proof uses the approach outlined above. First, the behaviour of each cell is individually verified. Then, the proof proceeds by proving properties of the circuit in each clock cycle.

A two dimensional array of proofs is kept: at the start of the k-th cycle, the array's (u, v) entry contains proofs of what the output of Cell u:v input neighbour's are at the end of the $(k-1)$-th cycle. The proof then uses this information to infer as much as possible about the output of Cell u:v at the end of the k-th cycle, and this information is then used to update the array of proofs so that Cell u:v's output neighbours can use this information in the $(k+1)$-th cycle.

Start of cycle	c_0 Cell 3:0	c_1 Cell 2:0	c_2 Cell 1:0	c_3 Cell 0:0	c_4 Cell 0:1	c_5 Cell 0:2	c_4 Cell 0:3
7				c_{11}			
8			c_{12}		c_{21}		
9		c_{13}				c_{31}	
10	c_{14}			c_{22}			c_{41}
11			c_{23}		c_{32}		
12		c_{24}				c_{42}	
13				c_{33}			
14			c_{34}		c_{43}		
15							
16				c_{44}			

Table 5: Benchmark 22: Actual output times

4.4 Analysis and comments

The FL proof script uses STE and the inference rules to prove what the output of the circuit is at different stages – this is summarised in Table 5.

Comparison between Tables 4 and 5 shows that even given the ability for the designer to choose the values of $t_1, \ldots, t_6$, the implementation does not meet the specification.

There are two possibilities. The easier and probably better solution would be to change the specification, in accordance with the results shown in Table 5. However, another solution would be to place one cycle delay buffers on the outputs `c_0`, `c_1`, `c_5` and `c_6`; the amount of extra circuitry is small, would not slow down the circuit, and would lead to a more elegant specification.

The proof script, including the proof of the correctness of all the multipliers and declarations, is approximately 650 lines long. The program itself is straightforward, although the use of a two dimensional array does not show off a functional, interpreted language at its best. The complete verification of a 4×4 systolic array of 32 bit multipliers takes just over 10 hours of CPU time on a DEC Alpha 3000 using the testing machine approach, and just under three hours using the direct method.

This verification uses the testing machine algorithm for STE, showing the weakness of using testing machines. The data structure needed to represent the model of the circuit is approximately 4M in size, making composition of circuit and testing machines difficult. While other implementations of machine composition are possible, the sheer size of the circuits remains an inherent problem. A similar problem can be seen in the verification of the multiplier (Table 2). Since both the size of the circuitry and the number of trajectory evaluations is quadratic in the bit-width, if every time trajectory evaluation must be done, circuit composition must be too, the resulting algorithm will be at least quartic. This explains why the verification of large bit widths becomes so expensive for testing machines.

5 Conclusion

This report has shown that using a quaternary-valued logic TL_n and its compositional theory, a practical integrated theorem-proving/model checking algorithm can be implemented that effectively verifies large circuits. The two benchmark circuits were verified with moderate human intervention and very reasonable computational cost.

References

[1] S. Hazelhurst and C.-J. H. Seger. Model Checking Partially Ordered State Spaces. Technical Report 95-18, Department of Computer Science, University of British Columbia, July 1995. Available by anonymous ftp as ftp://ftp.cs.ubc.ca/pub/local/techreports/1995/TR-95-18.ps.gz.

[2] S. Hazelhurst and C.-J.H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation ad BDD's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, April 1995.

[3] T. Kropf. Benchmark-Circuits for Hardware-Verification. In R. Kumar and T. Kropf, editors, *TPCD'94: Proceedings of the Second International Conference on Theorem Provers in Circuit Design*, Lecture Notes in Computer Science 901, pages 1–12, Berlin, September 1994. Springer-Verlag.

[4] C. Mead and L. Conway. *Introduction to VLSI Design.* Addison-Wesley, Reading, Massachusetts, 1980.

[5] C.-J.H. Seger. Voss — A Formal Hardware Verification System User's Guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, November 1993. Available by anonymous ftp as ftp://ftp.cs.ubc.ca/pub/local/techreports/1993/TR-93-45.ps.gz.

[6] C.-J.H. Seger and R.E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Journal of Formal Methods in Systems Design*, 6:147–189, March 1995.

[7] J. Staunstrup and T. Kropf. IFIP WG10.2 Benchmark Circuits for Hardware Verification. URL: `http://goethe.ira.uka.de/benchmarks/`.

A FL Code for Multiplication Proof

```
// miscellaneous
let high_bit = entry_width - 1; // 0..entry_width-1
let max_time = 800;
let out_time = 3;

//---------------- Node, variable declarations

let     A = Nnode AINP;
let     B = Nnode BINP;
let     RS i = Nnode (R_S i);
let     RC i = Nnode (R_C i)<<(high_bit-1)--0>>;
let     TopBit i = Nnode (R_C i)<<high_bit>>;

let     a   = (Nvar "a")<<(entry_width-1)--0>>;
let     b   = (Nvar "b")<<(entry_width-1)--0>>;
```

```
let     c   = Nvar "c";
let     d   = (Nvar "d")<<(high_bit-1)--0>>;

let     partial {n :: int}  = c  <<(n+high_bit)--0>>;

// BDD variable ordering for each stage of multiplier

let     m_bdd_order {n::int}   =
              n = 0
                => order_int_1 [b, a]
                 |  n=entry_width
                    => order_int_1 [partial n, d]
                       |  order_int_1 [b<<n>>, a, partial n, d];

let     zero_cond i = ((TopBit i)==('0))??;

let     interval  n  =
             n <= entry_width
               => [('(n*out_time), 'max_time)]
                | [('(n*out_time+2*entry_width), 'max_time)];

let     InputAnts = Always (interval 0)
                          (( (A == a) ?? ) and  ( (B == b) ??));
let     OutputCons =
             let lhs = RS entry_width in
             let rhs = (a * b)<<(2*entry_width-1)--0>> in
                   Always (interval (entry_width+1)) ((lhs==rhs)??);

// Antecedent for row n of the multiplier
let     MAnt {n::int}   =
                n = 0
                  =>  Always (interval 0)
                               ( ( (A == a)??) and
                                 ( (B<<n>> == b<<n>>)?? )
                               )
                   |  Always (interval n)
                                (( (A == a)??) and
                                 ( (B<<n>> == b<<n>>)?? ) and
                                 ( (RS (n-1)  == (partial (n-1)))??) and
                                 ( (RC (n-1)  == d)??)   and
                                 ( zero_cond (n-1)) );

// Consequent of row n of the multiplier
let     res_of_row n =
          let power n = Npow ('2) ('n)  in
          let lhs = (RS n) + (power (n+1))*(RC n) in
          let rhs =
            n=0
             => a * b <<0>>
              | ((partial (n-1))+(power n) * d) + (power n)* a * (b <<n>>)  in
                ((lhs == rhs)??);
```

```
let     Con_of_stage n =
          let power n = Npow ('2) ('n)  in
          let lhs = (RS n) + (power (n+1))*(RC n) in
          let rhs =  a * b<<n--0>> in
             Always (interval (n+1))
                 ((lhs == rhs)?? and (zero_cond n));


let     MCon {n::int}  =  Always  (interval (n+1))
                            ((res_of_row n ) and (zero_cond n));

let     Mthm n =
          let bdd_order = (m_bdd_order n) in
          let ant       = MAnt n in
          let con       = MCon n in
            prove_voss bdd_order multiplier ant con;

let     preamble_thm =
         let start = Mthm 0 in
          Precondition InputAnts start;

letrec  do_proof_main_stage  n m previous_step  =
           let curr  = Mthm n in
           let curr' = GenTransThm previous_step curr in
           let current  = Postcondition (Con_of_stage n) curr' in
               n =  m
                  => current
                   | do_proof_main_stage (n+1) m current;

let    main_stage  = do_proof_main_stage 1 high_bit preamble_thm;

let   adder_proof  =
        let post_ant_cond =
               (( (RS high_bit)     == (partial high_bit))??) and
               (( (RC high_bit)     == d)??) and
               (( (TopBit high_bit) == ('0))??)
                 in
       let post_ant = Always (interval entry_width) post_ant_cond
          in
       let power  = Npow ('2) ('entry_width)  in
       let rhs    =  ((partial high_bit) + power * d)<<(bit_width-1)--0>> in
       let post_con_cond = ((RS entry_width) == rhs)?? in
       let post_con      = Always (interval (entry_width+1))
                                   post_con_cond in
         prove_voss (m_bdd_order entry_width) multiplier post_ant post_con;

let    proof = GenTransThm main_stage adder_proof;
```

B FL Code for Matrix Multiplier Proof

```
// miscelleneous

let high_bit = entry_width - 1; // 0..entry_width-1
let max_time = entry_width < 10 => 100 | 10*entry_width;
let clock_time = max_time;   // half a clock cycle
let out_time = 3;
//----------------
let     prove_result = prove_voss_fsm;
let     prove_result_static = prove_voss_static;
//----------------  Node, variable declarations
//----- global
let     Clock    = Bnode CLK;

//----- individual cells
let     A u v = Nnode (AINP u v);      let     B u v = Nnode (BINP u v);
let     IN_C  u v = Nnode (C_Inp u v); let     OUT_C u v = Nnode (C_Out u v);

let     M = make_fsm sys_array;
let     RS u v i = Nnode (R_S u v i);
let     RC u v i = Nnode (R_C u v i)<<(high_bit-1)--0>>;
let     TopBit u v i = Nnode (R_C u v i)<<high_bit>>;


let     a   = (Nvar "a")<<(entry_width-1)--0>>;
let     b   = (Nvar "b")<<(entry_width-1)--0>>;   let     c   = Nvar "c";
let     d   = (Nvar "d")<<(high_bit-1)--0>>;      let     e   = Nvar "e";


let     partial {n :: int}  = e  <<(n+high_bit)--0>>;

// BDD variable ordering for each stage of multiplier
let     m_bdd_order {n::int}   =
              n = 0
                =>  order_int_1 [b, a]
                 |  n=entry_width
                    => order_int_1 [partial n, d]
                       |  order_int_1 [b<<n>>, a, partial n, d];

// timings
let     DuringInterval n f = During (n*out_time, max_time) f;

letrec  ClockAnt n =
         let range = 0 upto (n-1) in
         let false_range = map (\x.(('(2*x*clock_time),
                                     '(2*x*clock_time+clock_time-1))))
                                range in
         let true_range  = map (\x.('(2*x*clock_time+clock_time),
                                     '(2*(x+1)*clock_time-1)))
                                (butlast range) in
```

```
                (Always false_range ((Clock == Bfalse)??)) and
                (Always true_range  ((Clock == Btrue )??));

let     InputAnts u v = DuringInterval 0
                           ((A u v '= a) and  (B u v '= b));

let     zero_cond u v i = TopBit u v i   '= ('0);

// Antecedent for row n of the multiplier
let     MAnt u v {n::int}   =
                n = 0
                  =>    DuringInterval 0
                               ( ( A u v '= a ) and
                                 ( (B u v)<<n>> '= b<<n>>))
                   |    DuringInterval n
                                (( A u v '= a) and
                                 ( (B u v)<<n>>  '= b<<n>>) and
                                 ( RS u v (n-1)  '= (partial (n-1))) and
                                 ( RC u v (n-1)  '= d)      and
                                 ( zero_cond u v (n-1));

// Consequent of row n of the multiplier
let     res_of_row u v n =
          let power n = Npow ('2) ('n)  in
          let lhs = (RS u v n) + (power (n+1))*(RC u v n) in
          let rhs = n=0
             => a * b <<0>>
              | ((partial (n-1))+(power n) * d) + (power n)* a * (b <<n>>)  in
                lhs '= rhs;

let     Con_of_stage u v n =
          let power n = Npow ('2) ('n)  in
          let lhs = (RS u v n) + (power (n+1))*(RC u v n) in
          let rhs =  a * b<<n--0>> in
              DuringInterval (n+1)
                 ((lhs '= rhs) and (zero_cond u v n));

let     MCon u v {n::int}  =  DuringInterval (n+1)
                               ((res_of_row u v n ) and (zero_cond u v n));
let     Mthm u v n =
          let bdd_order = (m_bdd_order n) in
          let ant       = MAnt u v n in
          let con       = MCon u v n in
            prove_result bdd_order M ant con;

let     preamble_thm u v =
         print (nl^"Doing preamble"^nl) seq
         let start = Mthm u v 0 in
         (start catch start) seq
          Precondition (InputAnts u v) start;
```

```
letrec  do_proof_main_stage  u v n m previous_step  =
           let curr  = Mthm u v n in
           let curr' = GenTransThm previous_step curr in
           let current  = Postcondition (Con_of_stage u v n) curr' in
           (print (nl^"   Doing M["^(int2str u)^", "^(int2str v)^
                      "]("^(int2str n)^")"^nl^nl) seq
             (current catch current))
             seq
              ( n =  m
                  => current
                   | do_proof_main_stage u v (n+1) m current);

let    main_stage u v  = do_proof_main_stage u v 1 high_bit (preamble_thm u v);

let   adders_proof u v  =
        let post_ant_cond =
               ( (RS u v high_bit)     '= (partial high_bit)) and
               ( (RC u v high_bit)     '= d) and
               ( (TopBit u v high_bit) '= ('0))
                 in
       let post_ant = DuringInterval entry_width post_ant_cond in
       let power  = Npow ('2) ('entry_width)  in
       let rhs    =  ((partial high_bit) + power * d)<<(bit_width-1)--0>> in
       let post_con_cond = (RS u v entry_width) '= rhs in
       let post_con      =
               During (entry_width*(out_time+2), clock_time)
                          post_con_cond in
       let bdd_order = m_bdd_order entry_width in
       (print "Doing adder" seq (post_con catch post_con)) seq
         prove_result bdd_order M post_ant post_con;

let    cell_out_time = [('(2*clock_time), '(3*clock_time))];
let    register_proof u v =
           let c_ant  = (((RS u v entry_width) '=  (partial entry_width))
                        and ((IN_C u v) '= c)) in
           let c_ant' =
                (ClockAnt 2) and
                (During (entry_width*(out_time+2), clock_time)
                       c_ant) in
           let c_rhs  = (partial entry_width) + c in
           let c_con  = (OUT_C u v) '= c_rhs in
           let c_reg  = prove_result
                          (order_int_1 [c, partial entry_width])
                          M
                          c_ant'
                          (Always cell_out_time c_con)
              in
               ((print "Doing register") seq c_con catch c_con)
              seq
                  c_reg;
```

```
// one_proof u v: proves that the (u,v)-th cell works
// correctly
let    one_proof u v =
        // Prove that multiplier parts work (unclocked)
        let m_stage = main_stage u v in
        (m_stage catch m_stage) seq
        // take into account clocking and the partial sum input
        let new_ants= InputAnts u v  and
                      (ClockAnt 2) and
                      (DuringInterval 0 (IN_C u v  '= c)) in
        let new_thm = Precondition new_ants m_stage in
        // show the adder part of the ceol works
        let a_proof = adders_proof u v in
        (a_proof catch a_proof) seq
        // Add clocking to the adder proof
        let comp_proof =  GenTransThm new_thm a_proof in
        // Show that the registers work
        let r_proof    =  register_proof u v in
          ((r_proof catch r_proof)
            seq
            // stick them all together
        let result = (normaliseCon (GenTransThm comp_proof r_proof)) in
            result);


letrec  make_cell_row_list p_proc u v =
           v=array_depth
               => []
                | let res = p_proc u v in
                  print (snd (time res)) seq
                    (res seq (res:(make_cell_row_list p_proc u (v+1))));

letrec  make_proof_list p_proc u =
         u = array_width
             => []
              | (make_cell_row_list p_proc u 0):(make_proof_list p_proc (u+1));


let    cell_proof_list = make_proof_list one_proof 0;

// Show that the cells also progate their A and B inputs
let    one_proof_propagateA u v =
        let ants = (DuringInterval 0 (A u v '= a)) and (ClockAnt 2) in
        let ab_con = A u (v+1) '= a in
        let ab_reg =
             prove_result (m_bdd_order 0) M ants
                                          (Always cell_out_time ab_con) in
           ab_reg;

let    one_proof_propagateB u v =
```

```
        let ants = (DuringInterval 0 ((B u v) '= b)) and (ClockAnt 2) in
        let ab_con = (B (u+1) v)  '= b in
        let ab_reg =
             prove_result (m_bdd_order 0) M ants
                                      (Always cell_out_time ab_con) in
           ab_reg;

let   Apropagate_proof_list = make_proof_list one_proof_propagateA 0;
let   Bpropagate_proof_list = make_proof_list one_proof_propagateB 0;

let       cell_proof u v = el (v+1) (el (u+1) cell_proof_list);

let       Apropagate_proof u v = el (v+1) (el (u+1) Apropagate_proof_list);
let       Bpropagate_proof u v = el (v+1) (el (u+1) Bpropagate_proof_list);

let     em_thm = ([],[],[]);

//---------------------------------------------------------------
// The *_proof_list contains all the proofs that the individual
// components of the hardware work correctly. The rest of the
// proof shows that when connected together they produce
// the right matrix multiplication result

letrec  InsertActiveTheorem addfn ({u::int},{v::int},{new_thm::theorem}) [] =
                                        [(u, [(v, addfn new_thm em_thm)])]
     /\ InsertActiveTheorem addfn (u,v,new_thm)
                              ((au, alist):brest) =
                letrec PutActiveTheoremIn ({v::int}, {new_thm::theorem}) []
                                          = [(v, addfn new_thm em_thm)]
                    /\ PutActiveTheoremIn (v, new_thm) ((av, avlist):vrest) =
                           v = av
                             =>  (av, addfn new_thm avlist):vrest
                              |  (av, avlist):
                                     (PutActiveTheoremIn (v, new_thm) vrest)
                  in u = au
                          => (au, PutActiveTheoremIn (v, new_thm) alist):brest
                           | (au, alist):
                               (InsertActiveTheorem addfn(u,v,new_thm) brest);

letrec  RetrieveTheorem {u::int} {v::int}    [] =      ([],[],[])
     /\ RetrieveTheorem  u v  ((au, alist):brest) =
                letrec GetActiveTheorem v []  = ([],[],[])
                    /\ GetActiveTheorem v ((av, avlist):vrest) =
                           v = av
                             =>  avlist
                              |  GetActiveTheorem  v vrest  in
                        u = au
                          =>  GetActiveTheorem v alist
                           |  RetrieveTheorem  u v brest;

let    InsertActiveList add_fn thm_list current =
```

```
          itlist (\x.\y.InsertActiveTheorem add_fn x y) thm_list current;

load "iospecs.fl";

let  InputForCells _ _ = [];
let  addfirst  x (a,b,c) = (x:a,b,c);
let  addsecond x (a,b,c) = (a,x:b,c);
let  addthird  x (a,b,c) = (a,b,x:c);

let  InputAtStage n  the_lists =
       val (avals, bvals) = el (n+1) the_inputs in
       let left_list = map (\x.setInput A {x::int} 0 n (el (x+1) avals))
                     (0 upto (array_depth-1))
                   in
       let right_list =
              map (\x.setInput B 0 x n (el (x+1) bvals))
                     (0 upto (array_width-1)) in
       let down_list  =
              (map (\x.setInput IN_C (array_depth-1) {x::int} n ('0))
                  (0 upto (array_width-1)))@
              (map (\x.setInput IN_C x (array_width-1) {n::int} ('0))
                  (0 upto (array_depth-2))) in
         let res1 = InsertActiveList addfirst left_list the_lists in
         let res2 = InsertActiveList addsecond right_list res1 in
            InsertActiveList addthird down_list res2;

let        start_step = InputAtStage 0 [];
let        this_step  = start_step;         let num_step   = 0;

let        PropagateVal addfn row col ok1 {ok2::bool}  res old_list =
             ok1 AND ok2
                  => InsertActiveTheorem addfn (row, col, res) old_list
                   | old_list;

let        PropagateRes row col all res res_l =
                let c_index = "C"^(num2str(array_width-col-1+row)) in
                  all AND (row*col = 0)
                   =>  (c_index, res, (row, col)): res_l
                    |  res_l;

letrec     ProcessStageRow n {row::int}   []  so_far   = so_far
   /\      ProcessStageRow n row ((col, colthms):rest)
                                          (prop_list, res_l) =
             let make_step (a, b, c) =
                let ok  a n  = length a > n in
                let all_thms = (Identity(ClockAnt ((n+1)*2))):(a@b@c) in
                let ab_inps  = (a@b) in
                let all      = ok all_thms 3 in
                let curr_gen = all
                   => Conjunct [cell_proof row col, Apropagate_proof row col,
                                Bpropagate_proof row col] |
```

```
                  length ab_inps = 2
                  => Conjunct
                     [Apropagate_proof row col,Bpropagate_proof row col] |
                  ok a 0
                  => Apropagate_proof row col
                   | Bpropagate_proof row col in
               let curr_thm = Transform (TimeShift (2*n*clock_time))
                                                          curr_gen in
               let inps     = Conjunct all_thms in
               let res      = normaliseCon (GenTransThm inps curr_thm) in
               let new_l    = PropagateVal addfirst
                                 row (col+1) (col<(array_width-1))
                                 (ok a 0) res prop_list in
               let new_r    = PropagateVal addsecond
                                 (row+1) col (row<(array_depth-1))
                                 (ok b 0) res new_l in
               let new_d    = PropagateVal addthird
                                 (row-1) (col-1) ((row*col) > 0)
                                 all res new_r in
               let new_rl   = PropagateRes row col all res res_l
             in
                 empty ab_inps
                  =>    (prop_list, res_l)
                   |    (new_d, new_rl)
            in
                 ProcessStageRow n row rest (make_step  colthms);

letrec     ProcessStageProof n [] so_far   =      so_far
    /\     ProcessStageProof n ((row,rowthms):rest) so_far =
             let current = ProcessStageRow n row rowthms so_far in
               (print ("Doing row "^(int2str row)^nl)) seq
                 (current catch current) seq
                 ProcessStageProof n rest current;

let  do_step n start_step =
       letrec perform m curr_step =
             let current = ProcessStageProof m (InputAtStage m curr_step)
                                              ([], []) in
             (print ("Performing step "^(int2str m)^nl^nl)) seq
             (current catch current) seq
             m = n
               => [snd current]
                | (snd  current):(perform (m+1) (fst current)) in
          perform 0 start_step;

let  output_list  = do_step 15 [];

// present results
let  ShowRes t res_list = el (t+1) res_list;

let  Show t node =
```

```
      let res = ShowRes t output_list in
          find (\(x,y,a,b).(x=node) AND ((a*{b::int}) = 0)) res;

let  OutputOfArray row col =
       let strip (Always r f) = f in
       val (a, th, b, c) =  Show (outputFor row col)
                                 ("C"^(num2str(3+row-col))) in
            strip (con_of th);

letrec PrintRowOutput row col =
        (col = array_width+1)
            =>  nl^nl
             |  ("("^(int2str row)^" ,"^(int2str col)^") :"^
                        (el2str (OutputOfArray row col))^nl)
                 ^(PrintRowOutput row (col+1));

letrec PrintOutput row =
         row = array_depth + 1
            =>  nl
             |  (PrintRowOutput row 1) ^ (PrintOutput (row+1));
```