Verification of Combinational Logic in Nuprl

David A. Basin*
Peter Del Vecchio**

TR 89-1018
June 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported in part by an IBM Fellowship.

**Supported in part by NSF grant CCR8616552 and ONR grant N00014-88-K-0409.

Verification of Combinational Logic in Nuprl

David A. Basin*
Peter Del Vecchio†

Abstract

We present a case study of hardware specification and verification in the Nuprl Proof Development System. Within Nuprl we have built a specialized environment consisting of tactics, definitions, and theorems for specifying and reasoning about hardware. Such reasoning typically consists of term-rewriting, case-analysis, induction, and arithmetic reasoning. We have built tools that provide high-level assistance for these tasks.

The hardware component that we have proven is the front end of a floating-point adder/subtractor. This component, the MAEC (Mantissa Adjuster and Exponent Calculator), has 5459 transistors and has been proven down to the transistor level. As the circuit has 118 inputs and 107 outputs, verification by traditional methods such as case analysis would have been a practical impossibility.

1 Introduction

Over the last decade there has been a growing interest in applying formal methods to the design of real-world circuits. Some of the best known examples of work in this field are the verification of the VIPER microprocessor by the British Royal Radar Establishment [6], and the verification of the FM8501 by Warren Hunt [10]. In addition, using the results of Barrett [1], INMOS Corporation formally verified the floating-point unit of their IMS T800 microprocessor [11].

The increasing application of formal design methods has been motivated primarily by two factors. First, when specifications are given in a language with well-understood semantics, the behavior and structure of devices can be precisely defined. Second, once the devices have been formally specified, they are subject to analysis by proof in a formal system; one can rigorously demonstrate that their structural descriptions meet their behavioral specifications.

*Dept. of Computer Science, Cornell University, Ithaca NY, 14853. Supported in part by an IBM Fellowship.

†Dept. of Electrical Engineering, Cornell University, Ithaca, NY 14853. Supported in part by NSF grant CCR8616552 and ONR grant N00014-88-K-0409.

We have recently begun a study, at Cornell, of formalizing floating-point hardware. Our goal is to specify and prove properties of general-purpose parameterized components such as shifters, multiplexers, and adders. Our hope is that by building a general collection of proven devices, we can then build circuits such as floating point adders and multipliers simply by gluing together the appropriate specifications; specifics such as exponent and mantissa length are specified by instantiating subcomponent parameters. Moreover, the proofs of correctness for these new devices should follow simply from the proven behavioral specifications of their components.

The Nuprl proof development system [7] provides the formal basis for our research. Nuprl's logic is a higher-order constructive type theory, well-suited for formalizing mathematical arguments, especially those with computational significance. The system itself features: sophisticated editors that support the interactive development of definitions and proofs; a high-level meta-language that facilitates the development of tactics; and powerful decision procedures for arithmetic reasoning.

Within Nuprl we have designed an environment that has been tailored for specifying and reasoning about hardware. The basis of this environment is an encoding of quantified boolean logic, and a set of decision procedures and tactics that aid in reasoning about boolean formulas. To date we have used the environment for specifying and verifying combinational logic. It should serve equally well as a foundation for sequential logic.

We have used our environment to verify a number of hardware components. In this report we show how these components can be combined to specify and verify the MAEC (Mantissa Adjuster and Exponent Calculator). The MAEC is a section of a floating-point adder used in a systolic array FFT processor. This processor is being developed under a contract for NASA and will be used to process images from large ground-based telescopes. The MAEC inputs the mantissas and exponents for two floating-point numbers and adjusts the two mantissas so that bits with equal weight are aligned in the two mantissas. Each of the mantissas has 49 bits. Each exponent has 9 bits. Thus, since the circuit has 118 inputs and 107 outputs, verification by traditional methods such as case analysis would have been a practical impossibility.

Our paper is organized as follows. Section 2 contains an informal account of Nuprl. Section 3 describes our hardware environment. Section 4 provides an overview of the MAEC and the structural descriptions and behavioral specifications of its major components. The final section draws conclusions and suggests future research directions.

2 Nuprl

Nuprl's logic is a descendent of Martin-Löf's constructive type theory [12]. It is an extremely expressive logic; sufficiently rich, for example, to act as a founda-

tion for constructive mathematics. Types are stratified in an unbounded hierarchy of universes. U_1 is the first universe and contains all small types. Small types include data-types common to programmers and mathematicians such as integers, lists, pairing, disjoint union, function space, equality (e.g., $a = b$ in int), and first order propositions. Higher-order logic is defined within the theory via the propositions-as-types correspondence: a proposition is true if and only if the type associated with it is inhabited. Hence, when we speak of “formulas” or “propositions” we mean these defined notions.

The rules of Nuprl deal with *sequents*, objects of the form

$$x_1:H_1, x_2:H_2, \dots, x_n:H_n \gg P. \quad (1)$$

The H_i are referred to as *hypotheses* and the proposition P the *goal* or *conclusion*. “ $\gg$ ” is Nuprl’s equivalent of a turnstile. A sequent is true if the conclusion follows from the hypotheses. Constructively, this means that given members x_i of the types H_i , we can construct a member of the type P . Nuprl contains facilities to extract the computational content of theorems and execute this content in its *evaluator*. This ability to execute the computational content of proofs gives rise to the proofs-as-programs paradigm [7].

Nuprl’s proofs are trees containing sequents and inference rules and are built in a top-down fashion. The root sequent is the goal to be proved. The user applies inference rules which refine the goal into subgoals such that the truth of the goal may be established by the truth of the subgoals. The following, for example, might be a node in a proof tree where $\&$ -introduction occurs.

$$\begin{array}{ll} \gg & P \ \& \ Q \quad \text{By Intro} \\ & \gg P \\ & \gg Q \end{array} \quad (2)$$

Here the goal $P \ \& \ Q$ has been refined to the subgoals P and Q .

Inference rules in Nuprl may either be primitive rules or ML programs called *tactics*. Nuprl tactics are similar to those in LCF [8]: given a sequent as input, they apply primitive inference rules and other tactics to the proof tree. The unproven leaves of the resulting tree become the subgoals resulting from the tactic’s application. Tactics, then, act as derived inference rules; their correctness is justified by the way the type structure of ML is used. This mechanism provides a powerful method of raising the level of inference in Nuprl proofs. Ideally, users supply only the main proof ideas and tactics fill in the details. Moreover, as the tactic calls are themselves part of Nuprl proof trees, the proofs can serve as high-level explanations of formal arguments.

Theorem proving takes place within the context of a Nuprl *library*: an ordered collection of tactics, theorems, and definitions. Objects are created and modified using window-oriented structure editors. Nuprl contains a definition mechanism which is essentially a macro facility. Nuprl’s text editor, together with this definition facility, permits easily readable notations for objects in

Nuprl's type theory. A proof editor is used to construct proofs in a top-down manner, and to view or modify previously constructed proofs. Proofs are retained by the system and can be referred to by either the user or by tactics.

3 The Hardware Environment

A large amount of research at Cornell has been dedicated to designing *environments* in Nuprl [2,3,9]. An environment is a collection of definitions, theorems, and tactics for reasoning about a specific problem domain. Our belief is that it is better to start with a general logic and build specific environments than to work in logics specifically tailored around problem domains or to use proof systems with fixed problem solving strategies. In hardware verification, reasoning typically consists of term-rewriting, case analysis, induction, and arithmetic reasoning. We have built tools that provide high-level assistance for these tasks.

This section contains an overview of our environment. Our emphasis is on device specification and reasoning paradigms. The specifics concerning the MAEC will be covered in the following section. The definitions, theorems, and proof steps that we present appear much as they would on the screen of a Nuprl session.

3.1 Quantified Boolean Logic

Reasoning about combinational logic, whether at the gate or transistor level, requires reasoning about bits and functions that operate on bits. Hence, our environment begins with an encoding of boolean logic. We define the type **bool** to be the two element set containing only the integers zero and one.

$$\mathbf{bool} \equiv \{ i:\mathbf{int} \mid i = 0 \text{ in } \mathbf{int} \vee i = 1 \text{ in } \mathbf{int} \}$$

Truth and falsity (**tt** and **ff**) are defined as one and zero respectively. We say that a boolean formula **p** is true when it satisfies the predicate **tr(p)** defined by

$$\mathbf{tr}(p) \equiv p = \mathbf{tt} \text{ in } \mathbf{bool}.$$

These initial definitions are somewhat arbitrary, as booleans can be defined in terms of any two element set, but they have far reaching consequences. We can now efficiently define other boolean connectives taking advantage of special properties of arithmetic. For **p** and **q** booleans, we define the following operators of type **bool**→**bool**→**bool** (i.e., the operators take two boolean arguments and return a boolean):

$$\begin{aligned} p \wedge_B q &\equiv p * q \\ p \vee_B q &\equiv p + q - (p * q) \\ p \Rightarrow_B q &\equiv (1 - p) + p * q \\ \neg_B p &\equiv (1 - p) \end{aligned}$$

Other binary operators such as $\Leftrightarrow_B$ (boolean iff) and xor (exclusive or) are defined similarly. These operations are only sensible on booleans. However, there is a correspondence between these operators and operators defined on propositions. Specifically, we proved

$$\text{tr}(p \wedge_B q) = \text{tr}(p) \& \text{tr}(q) \text{ in } U_1 \quad (3)$$

$$\text{tr}(p \vee_B q) = \text{tr}(p) \vee \text{tr}(q) \text{ in } U_1 \quad (4)$$

$$\text{tr}(p \Rightarrow_B q) = \text{tr}(p) \Rightarrow \text{tr}(q) \text{ in } U_1 \quad (5)$$

$$\text{tr}(\neg_B p) = \neg \text{tr}(p) \text{ in } U_1 \quad (6)$$

where the connectives $\&$, $\vee$, $\Rightarrow$, and $\neg$ represent the propositional connectives **and**, **or**, **implies**, and **negation**. We shall generally leave off the boolean B subscripts that label the connectives, leaving context to clarify whether the symbols in question are boolean or propositional operators.

Boolean quantifiers are now directly definable in terms of boolean conjunction and disjunction. Specifically, we define the operators $\forall_B$ and $\exists_B$ of type $(\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool}$:

$$\forall_B x. P(x) \equiv P(\text{tt}) \wedge P(\text{ff})$$

$$\exists_B x. P(x) \equiv P(\text{tt}) \vee P(\text{ff})$$

As before, we can establish the following correspondences between these boolean formulas and their associated propositions.

$$\text{tr}(\forall_B x. P(x)) \iff \forall x:\text{bool}. \text{tr}(P(x)) \quad (7)$$

$$\text{tr}(\exists_B x. P(x)) \iff \exists x:\text{bool}. \text{tr}(P(x)) \quad (8)$$

It is worth noting that the underlying constructivity of Nuprl's type theory poses no barriers to reasoning about boolean logic. As integer equality is decidable, it follows that for all booleans p , $p \vee \neg p$ is provable. Moreover as our logic is constructive, the correspondences given in Equations 7 and 8 allow us to extract functions from true AE ($\forall/\exists$) boolean formulas. That is, if we can prove

$$>> \text{tr}(\forall_B x_1, \dots, \forall_B x_n. \exists_B y_1, \dots, \exists_B y_m. p),$$

then we can automatically (via rewriting) prove

$$>> \forall x_1, \dots, \forall x_n. \exists y_1, \dots, \exists y_m. \text{tr}(p).$$

From this second sequent, Nuprl's extractor can produce a function such that given as input $x_1, \dots, x_n$ the extracted function returns values $y_1, \dots, y_m$ such that p is true when the x_i and the y_i are bound to these values.

3.2 Circuit Representation

We represent hardware circuits as relations; the methodology we use is similar to that employed by Mike Gordon's HOL group [5]. A circuit is specified by an n -ary relation $R(a_0, \dots, a_n)$ that is built from logical connectives, constants, simpler relations, and recursion. The a_i represent inputs and outputs and we may think of them as external connections or ports of the circuit. The simpler relations may be basic logic gates (e.g., `nand` or `xor`), transistors, or previously defined relations. If we view each relation as specifying a constraint on the ports a_i , then anding together the various components gives a constraint for the entire circuit. Wires are represented by names; internal wires are hidden by existentially quantifying their names.

Our circuits are specified at the transistor level. We consider two types of transistors, n -type and p -type MOS, which we model as switches.

$$\begin{aligned} \text{ntran}(g,a,b) &\equiv g \Rightarrow (a \Leftrightarrow b) \\ \text{ptran}(g,a,b) &\equiv \neg g \Rightarrow (a \Leftrightarrow b) \end{aligned}$$

Power is a constraint that the value on a line p is always `tt`. Similarly, ground constrains p to `ff`.

$$\begin{aligned} \text{pwr}(p) &\equiv p \Leftrightarrow \text{tt} \\ \text{gnd}(p) &\equiv p \Leftrightarrow \text{ff} \end{aligned}$$

As a simple example, consider the circuit shown in Figure 1. This circuit generates the `carry_out` bit of a one-bit full-adder and is defined as follows:

$$\begin{aligned} \text{Carry_ckt}(a,b,\text{carry_in},\text{carry_out}) &\equiv \\ \exists p0, p1a, p1b, p2, n0, n1a, n1b, n2, \text{carry_outb} : \text{bool}. & \\ \text{Pwr}(p0) \wedge \text{Gnd}(n0) \wedge & \\ \text{ptran}(b,p0,p1a) \wedge \text{ntran}(b,n0,n1a) \wedge & \\ \text{ptran}(a,p0,p1a) \wedge \text{ntran}(a,n0,n1a) \wedge & \\ \text{ptran}(\text{carry_in},p1a,\text{carry_outb}) \wedge \text{ntran}(\text{carry_in},n1a,\text{carry_outb}) \wedge & \\ \text{ptran}(a,p0,p1b) \wedge \text{ntran}(a,n0,n1b) \wedge & \\ \text{ptran}(b,p1b,\text{carry_outb}) \wedge \text{ntran}(b,n1b,\text{carry_out}) \wedge & \\ \text{Pwr}(p2) \wedge \text{ptran}(\text{carry_outb},p2,\text{carry_out}) \wedge & \\ \text{Gnd}(n2) \wedge \text{ntran}(\text{carry_outb},n2,\text{carry_out}) & \end{aligned} \quad (9)$$

If we define a relation `Sum_ckt` for the sum port of the adder, we may then define a one-bit full-adder constructed as follows:

$$\begin{aligned} \text{add_1}(a,b,\text{carry_in},\text{sum},\text{carry_out}) &\equiv \\ \text{Carry_ckt}(a,b,\text{carry_in},\text{carry_out}) \wedge \text{Sum_ckt}(a,b,\text{carry_in},\text{sum}) & \end{aligned} \quad (10)$$

Many of the circuits in the MAEC are defined by primitive recursion. These include an n -bit adder, shifters, multiplexers, and other devices that take bit-vectors for arguments. For example, it is straight-forward to use the relation

Figure 1: Schematic of the Carry_ckt

`add_1` to define an n -bit ripple-carry adder using Nuprl's combinator for primitive recursion on higher types. Thus, we can recursively define an n -bit adder where:

$$\begin{aligned}
 \text{add_n}(0, \text{avec}, \text{bvec}, \text{carry_in}, \text{outvec}, \text{carry_out}) &= \\
 &\text{carry_in} \Leftrightarrow \text{carry_out} \\
 \text{add_n}(n+1, \text{avec}, \text{bvec}, \text{carry_in}, \text{outvec}, \text{carry_out}) &= \\
 \exists \text{cn. add_n}(n, \text{avec}, \text{bvec}, \text{carry_in}, \text{outvec}, \text{cn}) \wedge \\
 &\text{add_1}(\text{a}(n), \text{b}(n), \text{cn}, \text{out}(n), \text{carry_out})
 \end{aligned} \tag{11}$$

In the above circuit, `avec`, `bvec`, and `outvec` are elements of a defined type called **vector**, a vector being a function from integers (the bit position) to booleans (the value of that bit). We often take the view that vectors represent unsigned integers. That is, given an n -bit vector, we define a function `vec_val` where:

$$\begin{aligned}
 \text{vec_val}(0, \text{vec}) &= 0 \\
 \text{vec_val}(n+1, \text{vec}) &= 2^n * \text{Bitval}(\text{vec}(n)) + \text{vec_val}(n, \text{vec})
 \end{aligned} \tag{12}$$

Here `Bitval` is the identity function of type `bool` $\rightarrow$ `int`.

The adder used in the MAEC is somewhat more sophisticated than the adder presented here. The MAEC's adder does have a rippling-carry, but the `Carry_ckt` we used produces an inverted `carry_out`. This saves us from one inversion per full-adder block and thus reduces propagation delay. However, having the `carry_out` inverted means that every other stage in the adder has to use negative logic. The MAEC's adder thus had to be constructed from pairs of full-adder blocks.

3.3 Boolean Reasoning

Our encoding of boolean logic gives us several methods for reasoning about boolean formulas. We can view boolean formulas as arithmetic expressions and prove them by arithmetic evaluation. We can also reason on the propositional level by refining the propositions that the formulas correspond to. Sometimes, though, we reason somewhere in between, refining a sequent until the conclusion follows by evaluating some small part of it. Let us examine these options in more detail.

If p is a closed term, we can invoke Nuprl’s evaluator which will normalize p down to a ground term: either 0 or 1. When p is true it evaluates to 1 and $\gg \text{tr}(p)$ follows immediately. Hence, our encoding permits a reasonably efficient decision procedure for determining the truth of quantified boolean formulas (given the intractable nature of the underlying problem). This decision procedure is sufficient, for example, to prove that any non-inductively specified circuit is equivalent to its arithmetic, gate-level, or transistor-level specification.

When a goal $\text{tr}(p)$ contains variables bound in the hypothesis list $\mathcal{H}$ rather than the term itself, we prove $\mathcal{H} \gg \text{tr}(p)$ by combining case analysis with evaluation. Specifically, we have proved the following lemma which justifies case analysis on propositional valued boolean functions:

$$\forall P:\text{bool} \rightarrow U_1. P(\text{tt}) \vee P(\text{ff}) \implies \forall b:\text{bool}. P(b) \quad (13)$$

Now, given a sequent where b is bound to bool in the hypothesis list and the conclusion is $P(b)$, a (U_1) proposition where b occurs free, we can backchain through Equation 13. This results in the two new goals: $\mathcal{H} \gg P(\text{tt})$ and $\mathcal{H} \gg P(\text{ff})$. When $P(b)$ is $\text{tr}(p)$ we can repeat this case analysis until the subgoals are closed terms which can be proved or disproved by evaluation.

We have constructed a number of tactics for combining case-analysis and evaluation. One of our simpler tactics, **BoolCaseEval**, works as described above; it repeatedly (second-order) matches the conclusion against Equation 13 and completes the proof by evaluation. Others, such as **BoolCaseAnalyze**, are stronger and take added measures such as bringing over to the conclusion any hypotheses upon which the truth of the conclusion might depend and unfolding definitions to expose boolean subterms to match against b in Equation 13. These tactics have worked well in practice.

As in the case of the MAEC, many combinational circuits are too large to prove by evaluation or case-analysis. Also, when proving behavioral specifications for parameterized hardware blocks such as an n -bit adder, case analysis is impossible. In these situations, we reason directly about properties of the propositions using the correspondence theorems given in Equations 3 through 8. We may use these equations directly as rewrite rules to convert boolean formulas to propositions about booleans. However, in practice we have found it convenient to create tactics that use these equations to implement derived introduction and elimination inference rules for these defined operators. For

example, the $\&$ -introduction shown in Equation 2 can be recast as

```
>> tr(P  $\wedge$  Q)   By BoolIntro
>> tr(P)
>> tr(Q)
```

where the tactic **BoolIntro** backchains through Equation 3 and then applies the propositional **Intro** tactic. This approach of rewriting boolean formulas to propositions allows us to reason at the level of predicate calculus and apply Nuprl’s powerful standard tactic collection to the rewritten sequent.

3.4 Tactics

In addition to our tactics that support boolean evaluation, case analysis, and derived boolean inference rules, we have created a number of other special purpose tactics for use in hardware verification. Many of these are built around a general rewrite package. Our rewrite package is similar to Paulson’s [13] which provides the basis for rewriting in LCF and HOL. It provides procedures for creating rewrite functions from theorems (such as Equations 3-8) and provides higher-order operators for sequencing and composing rewrites. We have used it to implement a number of term normalizers and simplifiers that operate on boolean, arithmetic, and propositional expressions. These are described in detail elsewhere [2].

The design and application of Nuprl’s standard tactic collection is one of the subjects of Howe’s thesis [9]. This collection is substantial and includes components that support common forms of inference, arithmetic and other decision procedures, term and sequent manipulation functions, and many other general purpose programs. Many of these tactics find direct use in hardware verification. The most important tactic is **Autotactic**, which is usually applied after refinement steps to prove resultant subgoals. This tactic succeeds in proving subgoals that follow from certain kinds of arithmetic reasoning and has a component which attempts to prove membership goals, i.e., that some term is a member of some type. Such subgoals arise constantly. For example, demonstrating formula well-formedness takes the form of a membership goal. In general, membership is undecidable as it is equivalent to proving that a program meets a specification. However, **Autotactic** usually succeeds in proving well-formedness.

An Example

We conclude this section by returning to the relations **add_1** and **add_n** given in Equations 10 and 11. It is straightforward to prove that these basic low-level relations are equivalent to arithmetic specifications. Once these behavioral specifications are proved, further reasoning about these devices is abstract to their implementation. That is, it no longer matters whether they were implemented on the transistor or gate level, or how they were specified. Instead, their

behavioral specification is used as a rewrite rule that lifts the level of reasoning up to arithmetic.

The behavioral specification for `add_1` is:

```
>>∀a,b,carry_in,sum,carry_out: bool.
  tr(add_1(a,b,carry_in,sum,carry_out)) ⇒
    2 * Bitval(carry_out) + Bitval(sum) =
    Bitval(a) + Bitval(b) + Bitval(carry_in) in bool
```

This and other theorems for non-recursively specified devices are usually proved directly with `BoolCaseEval`; the single refinement step

(IntroAlls THEN BoolCaseEval ...)

serves as the proof. `IntroAlls` does five $\forall$ -introduction steps. `THEN` is a tactical which applies `BoolCaseEval` to the subgoals produced after the $\forall$ -introductions. The “...” indicates that a definition was invoked that runs `Autotactic` on all unproven subgoals. In this proof, `Autotactic`’s only task is to prove that the theorem statement is well-formed.

Similarly, the behavioral specification for `add_n` is:

```
>>∀n: N. ∀avec, bvec, outvec: vector. ∀carry_in, carry_out: bool.
  tr(add_n(n+1,avec,bvec,carry_in,outvec,carry_out)) ⇒
    2n+1 * Bitval(carry_out) + vec_val(n,outvec) =
    vec_val(n,avec) + vec_val(n,bvec) + Bitval(carry_in) in bool
```

Correctness proofs for recursively specified circuits require induction. In this example, induction is performed on n . The base case is proved automatically by `BoolCaseAnalyze`. The inductive case consists of unfolding the definition of the n -bit adder given in Equation 11. The resulting sequent is then reduced to simple arithmetic reasoning by backchaining through the inductive hypothesis, `add_1`’s behavioral specification, and the definition of `vec_val`. The entire proof takes 9 steps and is presented in [2].

4 MAEC — Case Study

4.1 Introduction

The hardware component that we have verified is known to us as the MAEC (Mantissa Adjuster and Exponent Calculator). The MAEC is the front-end to a floating-point adder/subtractor that will be used in a systolic array FFT processor. Briefly, the function performed by the MAEC is:

1. it inputs the exponents and mantissas for two floating-point numbers,
2. it right-shifts one of the two mantissas so that bits with equal weight are aligned in the two mantissas, and

3. it outputs:

- (a) the mantissa that was shifted,
- (b) the mantissa that was not shifted, and
- (c) the larger of the two exponents.

Figure 2: Block Symbol for the MAEC

The inputs to the MAEC are:

$$\begin{aligned}
 \mathbf{MAI} &= \{\text{mai48}, \dots, \text{mai0}\} \\
 \mathbf{MBI} &= \{\text{mbi48}, \dots, \text{mbi0}\} \\
 \mathbf{EA} &= \{\text{ea8}, \dots, \text{ea0}\} \\
 \mathbf{EB} &= \{\text{eb8}, \dots, \text{eb0}\}
 \end{aligned}$$

The vector **EA** is the exponent for mantissa **MAI**. **EB** is the exponent for mantissa **MBI**.

The outputs from the MAEC are:

$$\begin{aligned}
 \mathbf{MAOb} &= \{\text{mao48b}, \dots, \text{mao0b}\} \\
 \mathbf{MBO} &= \{\text{mbo48}, \dots, \text{mbo0}\} \\
 \mathbf{EOb} &= \{\text{eo8b}, \dots, \text{eo0b}\}
 \end{aligned}$$

The vector **MAOb** is a shifted and inverted copy of **MAI**¹. **MBO** is a non-inverted, shifted copy of **MBI**. **EOb** is the inverse of the larger of **EA** and **EB**. Each of the input and output mantissas has 49 bits. The exponents have 9 bits.

4.2 MAEC Schematic

As shown in Figure 3, the MAEC has 5 major subcomponents: one shift computer (**shift_comp**), two shifters (**shift_cell A** and **shift_cell B**), one exponent multiplexer (**exp_mux**), and one shift-by-one block (**shift_one**).

¹We use the convention that a lower-case “b” at the end the name of a boolean or vector indicates that the boolean or vector is inverted. Thus, **MAOb** is an inverted vector.

Figure 3: Top-Level Schematic for the MAEC

The **shift_comp** block inputs the exponents **EA** and **EB**, decides which of the two is larger, and computes the amount that mantissa A (**MAI**) and mantissa B (**MBI**) are to be shifted. The amount that we need to shift mantissa A is output by the **shift_comp** at **MAS**. Similarly, the amount that we need to shift mantissa B is output at **MBS**. The output **max** is high when $\text{vec_val}(9, \text{EB}) > \text{vec_val}(9, \text{EA})$.

The **shift_cells** take two bit-vectors as input: **vector_in** and **shift**. The output, **vector_out**, is a copy of **vector_in**, right-shifted by $\text{vec_val}(6, \text{shift})$ positions.

The exponent multiplexer (**exp_mux**) copies the inverse of either **EA** or **EB** to the output **EOb**. When the input **sel_EB** is high, the inverse of **EB** is copied to **EOb**. When **sel_EB** is low, the inverse of **EA** is copied to **EOb**.

Finally, the **shift_one** block right-shifts its input, **vector_in**, by either one or zero positions, depending on the state of **shift**. When **shift** is low, **vector_outb** is simply the inverse of **vector_in**. When **shift** is high, **vector_in** is right-shifted by one position and then inverted to produce **vector_outb**.

4.3 Behavioral Specification of the MAEC

```

 $\forall$  MAI, EA, MBI, EB, MAOb, MBO, EOb:vector.
maec(MAI, EA, MBI, EB, MAOb, MBO, EOb)  $\Rightarrow$ 
if vec_val(9,EA)  $\geq$  vec_val(9,EB)
then  $\forall$  i:{0..8}. EOb(i) =  $\neg$ (EA(i)) in bool &
 $\forall$  j:{0..48}. MAOb(j) =  $\neg$ (MAI(j)) in bool &
 $\forall$  k:{0..48}.
let index = k+vec_val(9,EA)-vec_val(9,EB) in
MBO(k) = ( (index<49)  $\Rightarrow$  MBI(index) | 0 ) in bool
else  $\forall$  i:{0..8}. EOb(i) =  $\neg$ (EB(i)) in bool &
 $\forall$  j:{0..48}. MBO(j) = MBI(j) in bool &
 $\forall$  k:{0..48}.
let index = k+vec_val(9,EB)-vec_val(9,EA) in
MAOb(k) = ( (index<49)  $\Rightarrow$   $\neg$ (MAI(index)) | 1 ) in bool

```

Figure 4: Behavioral Specification for the MAEC

The behavioral specification the MAEC, as encoded in Nuprl, is shown in Figure 4. This specification tells us that when the value of exponent **EA** is greater than or equal to the value of exponent **EB**, then:

1. the output exponent, **EOb**, is an inverted copy of **EA**,
2. the output A mantissa, **MAOb**, is an inverted copy of the input A mantissa, **MAI**, and
3. the output B mantissa, **MBO**, is a copy of the input B mantissa, **MBI**, right-shifted by $\text{vec_val}(9,EA) - \text{vec_val}(9,EB)$ positions.

When **EB** is greater than **EA**, the situation is analogous.

As can be seen from the description of Figure 4, when right-shifting is done in the MAEC we drop any bits that are right-shifted past bit number zero. Also, non-inverted vectors (e.g., **MBO**) are left-filled with zeros, while inverted vectors (e.g., **MAOb**) are left-filled with ones.

It is interesting to note that the MAEC's behavioral specification is natural and understandable. This results from the flexibility of Nuprl's definition facility and the expressiveness of Nuprl's underlying logic. **If-then-else** constructs are not included in the constructive logic upon which Nuprl is based. Neither are the **let ... in** or **(a $\Rightarrow$ b | c)** forms. However, since Nuprl lets the user custom-define how propositions are displayed, we were able to use very natural constructs to form the goals in our hardware proofs.

4.4 Proof of the MAEC in Nuprl

4.4.1 Introduction

In the following section we will try to give the reader a feel for the capabilities of the Nuprl hardware environment by presenting the behavioral descriptions that were developed and proven for each of the main subcomponents of the MAEC. We will also present sample proof steps which illustrate some of the tactics that have been developed.

The first subcomponent that we will consider is the **shift_cell**. This was proven as a specific case of a general, parameterized linear shifter called the **shifter**.

The next component we'll look at is the **shift_one**. This cell was constructed and proven as a specific case of one of the subcomponents of the **shifter**.

Following the **shift_one** we'll look at the **shift_comp**. This cell was perhaps the most interesting component of the MAEC to prove because it is constructed from blocks that were defined in very different manners (parameterized/non-parameterized and recursively/non-recursively). Also, since this cell performs an arithmetic function, a good deal of arithmetic reasoning was needed in its proof.

Finally, the last cell we'll look at is the **exp_mux**. The **exp_mux** is a simple cell — it contains only two groups of multiplexers. The proof of the **exp_mux**'s behavioral specification was thus concise and straightforward.

4.4.2 The shift_cell

The **shift_cell** is a linear shifter that takes as input one 49-bit vector named **vector_in**, and one 6-bit vector named **shift**. The output, **vector_out**, is a copy of **vector_in**, right-shifted by **vec_val(6,shift)** positions. Like **vector_in**, **vector_out** has 49 bits.

```
>>∀ vector_in, shift, vector_out:vector.  
  shift_cell(vector_in, shift, vector_out) =>  
    ∀ i:{0..48}.  
      let index = i+vec_val(6,shift) in  
        vector_out(i) = ( (index<49) => vector_in(index) | 0 ) in bool
```

Figure 5: Behavioral Specification of the **shift_cell**

The behavioral specification for the **shift_cell** is given in Figure 5. The proof of this specification is interesting for two reasons: first because we have shown that the **shift_cell** used in the layout of the MAEC is logically correct, and second because the **shift_cell** is merely a specific case of a more general hardware block that was proven in Nuprl called the **shifter**.

The **shifter** is a general-purpose vector shifter that is parameterized by:

1. the number of bits in the vector that is to be shifted, and
2. the number of “shift blocks” that it contains.

The “shift blocks” are subcomponents of the **shifter** that, when activated, shift their input by some integral power of 2 positions. A **shifter** composed of n shift blocks will have blocks that shift by 2^0 positions, 2^1 positions, 2^2 positions, and so on up to 2^{n-1} positions. Thus, by turning its subcomponents on and off, a **shifter** with n blocks can implement any shift between 0 and $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ positions. The structure of the **shifter** is defined recursively as follows:

```

shifter(0, num_bits, vector_in, shift, vector_out) =
  shift_block_pair(num_bits, 1, vector_in, shift(1), shift(0), vector_out)
shifter(num_blk_pairs, num_bits, vector_in, shift, vector_out) =
  ∃ mid_vec:vector.
    shift_block_pair(num_bits, -1+2*num_blk_pairs, vector_in,
      shift(-1+2*num_blk_pairs), shift(2*(num_blk_pairs-1)), vector_out) &
    shifter(num_blk_pairs-1, num_bits, mid_vec, shift, vector_out)

```

The cell **shift_block_pair** used in the above definition of the **shifter** contains one pair of shift blocks. The first parameter for this cell (given the value **num_bits** above) specifies the number of bits in the vector that is to be shifted. The second parameter (assigned the value 1 for the base case of the shifter, and $2 \cdot \text{num_blk_pairs}$ for the inductive case) is the $\log_2$ of the number of positions that the upper shift block in the pair shifts its input. The lower block shifts by $2^{2 \cdot \text{num_blk_pairs} - 1}$ positions.

The behavioral specification that was proven for the **shifter** is:

$$\begin{aligned}
 &>> \forall \text{ num_bits}:\{i:\text{Int} \mid 3 \leq i\}. \forall \text{ num_blk_pairs}:\{j:\text{Int} \mid 0 < j \ \& \ 2^{2j-1} < \text{num_bits}\}. \quad (14) \\
 &\quad \forall \text{ vector_in}, \text{ shift}, \text{ vector_out}:\text{vector}. \\
 &\quad \text{shifter}(\text{num_blk_pairs}, \text{num_bits}, \text{vector_in}, \text{shift}, \text{vector_out}) \Rightarrow \\
 &\quad \forall i:\{0..\text{num_bits}-1\}. \\
 &\quad \quad \text{let index} = i + \text{vec_val}(2 \cdot \text{num_blk_pairs}, \text{shift}) \text{ in} \\
 &\quad \quad \text{vector_out}(i) = ((\text{index} < \text{num_bits}) \Rightarrow \text{vector_in}(\text{index}) \mid 0) \text{ in bool}
 \end{aligned}$$

This specification tells us that **vector_out** is a copy of **vector_in**, right-shifted by $\text{vec_val}(2 \cdot \text{num_blk_pairs}, \text{shift})$ positions.

By comparison with Figure 5, we can see that the **shift_cell** is indeed just a specific case of the **shifter**: **shift_cell**(**vector_in**, **shift**, **vector_out**) performs the same function as **shifter**(49, 3, **vector_in**, **shift**, **vector_out**). In fact, in our proof of the MAEC, a **shifter**(49, 3, ...) was used whenever a **shift_cell**(...) was needed.

As our first example of a proof in Nuprl, we will discuss briefly how the **shifter** was proven. Since the structure of the **shifter** is defined recursively, an inductive proof was required, where we induct on the variable **num_blk_pairs**. In the base case we have only one pair of shift blocks, i.e., one **shift_block_pair**. This case was

proven by expanding the definitions of `shifter` and `vec_val`, and then instantiating the following lemma that was proven for the `shift_block_pair` cell:

$$\begin{aligned}
&>> \forall \text{ num_bits}:\mathbb{N}+. \forall \text{ shift_power}:\{j:\text{Int} \mid 0 < j \ \& \ 2^j < \text{num_bits}\}. \\
&\quad \forall \text{ vector_in}:\text{vector}. \forall \text{ shift1}, \text{shift0}:\text{bool}. \forall \text{ vector_out}:\text{vector}. \\
&\quad \text{shifter_block}(\text{num_bits}, \text{shift_power}, \text{vector_in}, \text{shift1}, \text{shift0}, \text{vector_out}) \Rightarrow \\
&\quad \forall i:\{0..\text{num_bits}-1\}. \\
&\quad \quad \text{let index} = i + 2^{n*\text{shift1}} + 2^{(n-1)*\text{shift0}} \\
&\quad \quad \text{vector_out}(i) = ((\text{index} < \text{num_bits}) \Rightarrow \text{vector_in}(\text{index}) \mid 0) \text{ in bool}
\end{aligned} \tag{15}$$

This lemma tells us that the `shift_block_pair` cell will right-shift `vector_in` by $2^n*\text{shift1} + 2^{n-1}*\text{shift0}$ positions to produce `vector_out`.

The inductive case of the proof was done by unrolling the `shifter(num_blk_pairs, ...)` into a `shifter(num_blk_pairs-1, ...)` and a `shift_blk_pair`, instantiating (15) above, and then forward-chaining through the inductive hypothesis. The step in which the unrolling was done is interesting because it is a good example of both rewriting and arithmetic reasoning in Nuprl. The section of the proof relevant to this step is reproduced in Figure 6.

```

1. num_bits:{i:ℕ|i≥3}
2. num_blk_pairs:ℕ
3. 1<num_blk_pairs
⋮ {omitted hypotheses}
10. shifter(num_bits, num_blk_pairs, vector_in, shift, vector_out)
>> let index = i + vec_val(2*num_blk_pairs, vector_in) in
    vector_out(i) = ( index<num_bits => vector_in(index) | 0 )

BY (RewriteHyp ShifterUnroll 10...tmono)

10. ∃ mid_vec:vector.
    shifter_block_pair(num_blk_pairs-1, num_bits, vector_in,
        shift(-1+2*num_blk_pairs), shift(-2+2*num_blk_pairs),
        mid_vec) &
    shifter(num_bits, num_blk_pairs-1, mid_vec, shift, vector_out)
>> let index = i + vec_val(2*num_blk_pairs, vector_in) in
    vector_out(i) = ( index<num_bits => vector_in(index) | 0 )

```

Figure 6: Snapshot of a Rewrite Step in the Proof of the `shifter`

At the top of Figure 6 we have the initial set of hypotheses and the goal to be proven. In the middle we have the tactic that is being executed: (**RewriteHyp ShifterUnroll 10...tmono**). On the bottom we have the hypotheses that were changed and the new goal. (Here only hypothesis number ten changed.)

The rewrite that was used in Figure 6, **ShifterUnroll**, was created from a shifter-unrolling theorem which proves that an unrolling is always valid when

`num_blk_pairs > 1`. The “...tmono” in the refinement rule in Figure 6 indicates that Nuprl’s **Autotactic** was run after the unrolling was finished, and that the tactic **MonoTac** was run on any monotonicity goals that arose. **MonoTac** is a general-purpose tactic that proves most of the monotonicity goals that arise in our hardware proofs. In this case, the following two monotonicity goals arose:

1. `num_blk_pairs:ℕ, 1 < num_blk_pairs >> 2 * num_blk_pairs - 1 ≥ 0`
2. `j:ℕ, 0 < j >> 2 * j - 1 ≥ 0`

In 2. above, `j` is an arbitrary integer. These goals came up because, as we see in (14), we must to prove that `num_blk_pairs` is within the set $\{j:\mathbb{N} \mid 0 < j \ \& \ 2^{(2*j-1)} < \text{num_bits}\}$. Both of these monotonicity goals were proven automatically by **MonoTac**.

4.4.3 The `shift_one`

The `shift_one` cell has two inputs: one vector, 49-bits long named `vector_in`, and one boolean named `shift`. When `shift` is high, `vector_in` is right-shifted by one position and inverted to produce the `shift_one`’s output, `vector_outb`. Like `vector_in`, `vector_outb` has 49 bits. When `shift` is low, `vector_outb` is simply an inverted copy of `vector_in`. The behavior of the `shift_one`, as encoded in Nuprl, is given in Figure 7.

```
>> ∀ vector_in, shift, vector_out:vector.
  shift_one(vector_in, shift, vector_out) =>
    ∀ i:[0..48].
      let index = i+1 in
        vector_out(i) = ( (index<49) => vector_in(index) | 0 ) in bool
```

Figure 7: Behavioral Specification for the `shift_one`

Since the **shifter** that was used to construct the **shift_cell** is composed of blocks which shift their input by 2^n positions, where `n` is any natural number, we were able to implement the `shift_one` cell by using one of these same shift blocks, setting `n` equal to zero. The block that we used is named the **two_to_n_bit_shifter**. The behavioral specification of the **two_to_n_bit_shifter** appears in Figure 8. As can be seen from this specification, the **two_to_n_bit_shifter** is parameterized by the number of bits in the input vector (`num_bits`) and the $\log_2$ of the number of positions that the block will shift (`n`). Thus, the `shift_one` cell is just a **two_to_n_bit_shifter** with `num_bits=49` and `n=0`.

Verifying the `shift_one` was simply a matter of instantiating `num_bits` and `n` in the proven behavioral specification of the **two_to_n_bit_shifter**. The proof of the **two_to_n_bit_shifter** was accomplished by doing a case split on the variable `shift`, substituting `shift` into the boolean logic specifications of the multiplexers that make up the **two_to_n_bit_shifter**, and then using a rewrite tactic called **BoolSimp** to reduce the resulting expressions. An example of **BoolSimp** in action is shown in

```

>> ∀ num_bits:N+. ∀ n:{i:Int|0 ≤ i & 2^i < num_bits}.
  ∀ vector_in:vector. ∀ shift:bool. ∀ vector_outb:vector.
    two_to_n_bit_shifter(num_bits,n,vector_in,shift,vector_outb) =>
      ∀ i:{0..num_bits-1}.
        let index = i+2^n*shift in
          vector_outb(i) = ( (index < num_bits) => ¬(vector_in(index) | 1) ) in bool

```

Figure 8: Behavioral Specification for the `two_to_n_bit_shifter`

Figure 9. In this figure, our intent is to simplify the expressions shown after the “ $\Leftrightarrow$ ” signs shown in hypotheses 7 and 8. **BoolSimp** simplifies these expressions to those in the new hypotheses 7 and 8 (at the bottom of the figure).

```

: {omitted hypotheses}
6. i:{0..num_bits-1}
7. ∀ x.{0..num_bits-(2^n)-1}. vector_outb(x) ⇔ (¬(ff) ∧ ¬(vector_in(x))) ∨ ¬(¬(ff))
8. ∀ y.{num_bits-2^n..num_bits-1}. vector_outb(y) ⇔ (ff ∧ ¬(vector_in(y+2^n))) ∨
  (¬(ff) ∧ ¬(vector_in(y)))
>> vector_outb(i) = ¬(vector_in(i)) in bool

BY (BoolSimpHyp 8...) THEN (BoolSimpHyp 7...)

7. ∀ x.{0..num_bits-(2^n)-1}. vector_outb(x) ⇔ ¬(vector_in(x))
8. ∀ y.{num_bits-2^n..num_bits-1}. vector_outb(y) ⇔ ¬(vector_in(y))
>> vector_outb(i) = ¬(vector_in(i)) in bool

```

Figure 9: Example of the **BoolSimp** Tactic

4.4.4 The `shift_comp`

The `shift_comp` has two responsibilities:

1. it computes the amount to shift the two mantissas which are input to the MAEC and
2. it determines what the MAEC’s output exponent will be.

This block has two inputs — the 9-bit vectors **EA** and **EB** — and has 3 outputs — two 6-bit vectors named **MAS** and **MBS**, and one boolean named **max**. **EA** is the exponent for mantissa A (**MAI**); **EB** is the exponent for mantissa B (**MBI**). The value of **MBS**, i.e., `vec_val(6,MBS)`, is the amount that mantissa B must be shifted. Mantissa A must be shifted by `vec_val(6,MAS)+1` positions. The output **max** (Mantissa A eXtra) is a flag that goes high when the shift for mantissa A is

non-zero. (Equivalently, it goes high when $EA < EB$.) The `vec_val(6,MAS)` part of mantissa A's shift is implemented by `shift_cell A`. The extra "+1" is implemented by the `shift_one` cell.

```

>>∀ EA,EB,MAS,MBS:vector. ∀ max:bool.
  if shift_comp(EA,EB,MAS,MBS,max)
    then if vec_val(9,EA)≥vec_val(9,EB)
      then
        vec_val(6,MAS)=0 in N &
        max=0 in bool &
        if (shift_needed_for_mantissa_B(9,EA,EB) > max_shift_for_n_shift_blocks(6))
          then (vec_val(6,MBS) > shift_needed_to_zero_mantissa(49))
            else (vec_val(6,MBS) = shift_needed_for_mantissa_B(9,EA,EB) in Int)
        else
          vec_val(6,MBS)=0 in N &
          max=1 in bool &
          if (shift_needed_for_mantissa_A(9,EA,EB)-1 > max_shift_for_n_shift_blocks(6))
            then (vec_val(6,MAS) > shift_needed_to_zero_mantissa(49))
              else (vec_val(6,MAS) = shift_needed_for_mantissa_A(9,EA,EB)-1 in Int)

```

Figure 10: Behavioral Specification of the `shift_comp`

The behavioral specification for the `shift_comp` is given in Figure 10. Before we can fully understand this specification, we must first define the following functions which are used therein:

`shift_needed_for_mantissa_A(n, EA, EB),`
`shift_needed_for_mantissa_B(n, EA, EB),`
`max_shift_for_n_shift_blocks(n),` and
`shift_needed_to_zero_mantissa(n).`

The function `shift_needed_for_mantissa_A`, as its name implies, returns the amount of shift needed for mantissa A. By definition, mantissa A must be right-shifted by 0 positions if $EA \geq EB$, and $EB - EA$ positions otherwise. So, by definition:

$$\text{shift_for_mantissa_A}(n, EA, EB) \equiv \begin{cases} 0 & \text{if } \text{vec_val}(n, EA) \geq \text{vec_val}(n, EB) \\ \text{vec_val}(n, EB) - \text{vec_val}(n, EA) & \text{otherwise} \end{cases}$$

`shift_needed_for_mantissa_B` is analogous:

$$\text{shift_for_mantissa_B}(n, EA, EB) \equiv \begin{cases} \text{vec_val}(n, EA) - \text{vec_val}(n, EB) & \text{if } \text{vec_val}(n, EB) < \text{vec_val}(n, EA) \\ 0 & \text{otherwise} \end{cases}$$

The function `max_shift_for_n_shift_blocks(n)` inputs a natural number, n , and returns the maximum amount of shift that n shift blocks can produce. As discussed in the `shift_cell` section of this paper, assuming that the blocks shift by $2^0, 2^1, \dots, 2^{n-1}$ positions respectively, the maximum shift is $2^n - 1$ positions.

Finally, the function `shift_needed_to_zero_mantissa(n)` tells us how many positions we must right-shift a vector of length `n` before all of the bits in the vector become left-filled with zeros. This function is the identity function — if you right-shift an `n`-bit vector by `n` positions while left filling with zeros, the vector will be completely zeroed.

Now that we have defined all of the functions that are used in the specification of the `shift_comp`, we can interpret what the specification is telling us. For the condition that $EA \geq EB$:

1. `MAS`, the vector that tells `shift_cell A` how much to shift mantissa `A`, is zero,
2. `max`, the Mantissa `A` eXtra flag, is zero,
3. if the shift needed for mantissa `B` is greater than the maximum amount that a `shift_cell` can implement, then `MBS` is set to a value that will cause `shift_cell B` to zero mantissa `B`, and
4. if the shift needed for mantissa `B` is less than or equal to the maximum amount that a `shift_cell` can implement, then `MBS` is equal to the amount that we want to shift mantissa `B`.

For the condition that $EA < EB$, the situation is analogous.

The proof of the `shift_comp` is interesting because it combines many different styles of reasoning. The subcomponents of the `shift_comp` — a ripple-carry adder, a pair of multiplexers, and an “overflow” detector — were defined in very different manners and thus required different styles of proof. Also, since this cell performs an arithmetic function, a good deal of arithmetic reasoning was required in its proof.

The adder, for example, was parameterized by the number of bits in the input vectors and was defined recursively. The proof of this block required the use of induction and arithmetic reasoning.

The pair of multiplexers in the `shift_comp` is named the `shift_mux`. This cell is responsible for sending either zero or `shift_needed_for_mantissa_A - 1` into `MAS`, and either either zero or `shift_needed_for_mantissa_B` into `MBS`. This cell is constructed from basic multiplexer cells which are grouped into two separate strings. Each string is constructed by universally quantifying over the bits in the input and output vectors. For example, the multiplexer-string used for the `MAS` vector is represented by

$$\forall x[0..5]. \text{smuxc}(\text{MASlb}(x), \text{A_select}, \text{MAS}(x)).$$

If `A_select` is high, the cell `smuxc` sends the inverse of `MASlb(x)`, an internal signal, to `MAS(x)`. If `A_select` is low, `MAS(x)` is set equal to zero. Due to the non-recursive structure of the `shift_mux`, no unrolling or induction was necessary in its proof — the proof consisted mainly of expanding cell definitions, case splits, substitutions, and boolean simplifications.

The `shift_comp`'s `overflow_detector` is a specific, non-parameterized component designed specifically to deal with the MAEC's 49-bit mantissas and 9-bit exponents. It was not, therefore, defined using recursion or universal quantification, but rather as a flat circuit, with all of its components on one level. Because of its "flatness", this block was verified through rewriting boolean equations and case analysis.

Finally, at the top level of the `shift_comp`'s proof (the point where we glued together the subcomponents), somewhat sophisticated arithmetic reasoning was needed. For example, this cell computes either $EA - EB$ or $EB - EA$, whichever is positive. The `shift_comp` computes only $EA - EB$. This is the subtraction needed when $EA \geq EB$. If it turns out that $EA - EB$ is negative, then $EA < EB$ and we actually wanted to compute $EB - EA$. Rather than now calculating $EB - EA$ directly, the `shift_comp` simply inverts the result of $(EA - EB)$ and uses the property, which we proved, that if $EB > EA$ then $(EB - EA) = \neg(EA - EB) + 1$. Thus, the `shift_comp` calculates either $EA - EB$ when $EA \geq EB$, or $EB - EA - 1$ when $EA < EB$.

Another complication in the `shift_comp`'s proof is that in this cell, as in most hardware subtractors, subtraction is implemented by adding the inverse of the subtrahend to the minuend. Since the functions `shift_needed_for_mantissa_A` and `shift_needed_for_mantissa_B` were defined in terms of subtraction and not inverted addition, we had to prove that one's complement addition is valid.

4.4.5 The `exp_mux`

The `exp_mux` has three inputs: the two 9-bit exponents `EA` and `EB`, and one boolean named `sel_EB`. This cell performs a very simple function — when `sel_EB` is high, the inverse of `EB` is output at `EOB`; when `sel_EB` is low, the inverse of `EA` is copied to `EOB`. The behavioral specification for the `exp_mux` is given in Figure 11. Since the `exp_mux` is such a simple cell, its proof was also very simple. The complete proof consisted only of the following steps (each of these steps corresponds directly to a tactic used in the proof):

1. introducing the universally quantified variables,
2. performing a case split on the boolean variable `shift`,
3. expanding the `exp_mux` down to universally quantified boolean logic equations,
4. substituting the value of `shift` into the logic equations,
5. performing a boolean simplification on the equations,
6. performing an elimination, and
7. rewriting a boolean iff into an equality (i.e., transforming an expression of the form $a \Leftrightarrow b$ into $a = b$ in `bool`).

Total time to define the structure of the `exp_mux` and to create and prove the behavioral specification: approximately 45 minutes.

```

>>  $\forall EA, EB: \text{vector}. \forall easb: \text{bool}. \forall EOb: \text{vector}.
\text{exp\_mux}(EA, EB, easb, EOb) \Rightarrow
\text{if } easb = \text{tt in bool}
\text{ then } \forall i: \{0..8\}. EOb(i) = \neg(EB(i)) \text{ in bool}
\text{ else } \forall j: \{0..8\}. EOb(j) = \neg(EA(j)) \text{ in bool}$ 
```

Figure 11: Behavioral Specification for the `exp_mux`

4.4.6 The MAEC

The behavioral specification of the MAEC was presented in Section 1.4.3. The structural specification is:

```

MAEC (MAI, EA, MBI, EB, MAOb, MBO, EOb) =
 $\forall MAS, MBS: \text{vector}. \forall max: \text{bool}.
\text{shift\_comp}(EA, EB, MAS, MBS, max) \ \& \ \text{exp\_mux}(EA, EB, max, EOb) \ \&
\text{shift\_cell}(MBI, MBS, MBO) \ \&
\exists temp: \text{vector}.
\text{shift\_cell}(MAI, MAS, temp) \ \& \ \text{shift\_one}(temp, max, MAOb)$ 
```

The proof of the MAEC's behavioral specification was quite straightforward. Once the predicate logic specifications of its subcomponents were verified, the proof of the MAEC consisted mainly of instantiating the theorems developed for the subcomponents, performing case splits, introductions, eliminations, arithmetic simplifications, and substitutions.

5 Conclusion

The specification and verification of the MAEC took approximately three man-months. The final library contains approximately 500 objects, 154 of which are theorems, the remainder definitions. Most of the time was spent building preliminaries such as the basic hardware tactic collection, theories for reasoning about bit-vectors, theorems and tactics centered around monotonicity reasoning, and extensions to our rewrite package.

Almost all of the MAEC was built from parameterized subcomponents or components that could easily be generalized. The proof of the MAEC's behavioral specification combined the behavioral specifications of its components in a straight-forward way. This gives us some confidence that we can attain our goal of building a floating-point tool kit where users can build customized floating-point circuits from components that come with proven behavioral specifications. The library we have built is a large step in that direction.

Our effort was the first application of Nuprl to hardware verification. On the whole we are pleased with the results. One of the authors (DelVecchio) had created a behavioral specification for the MAEC in predicate calculus before he knew Nuprl. Due to the expressiveness of Nuprl's logic and its flexible definition facility, the behavioral specification of the MAEC in Nuprl (Figure 4) is essentially identical to his original predicate calculus specification. Another extremely useful feature of Nuprl is its computation system. All terms in Nuprl can be evaluated. In particular closed quantified boolean formulas evaluate to true or false. This property gives us a decision procedure for verifying non-inductively specified devices. If we had to rely on some alternative procedure, such as one based on rewriting, execution would be many orders of magnitude slower. Finally, Nuprl's tree-structured proof editor helps manage the complexity of editing large proofs. Proofs are retained by the system and can be viewed and modified at a later date. Moreover, sections of proofs can be saved and re-executed on different sequents. This ability to mark and copy proofs saved us much effort in reproving theorems after making minor changes to definitions and theorem statements. It also found repeated use as a simple analogy tactic.

There is, of course, room for improvement. On the system side, a major bottleneck is well-formedness reasoning. As previously mentioned Nuprl propositions must be proved well-formed and in general this is recursively undecidable. However, the well-formedness of the propositions that are used to specify device structures and their behaviors is decidable; **Autotactic** will always succeed in constructing the appropriate well-formedness proof. However, this is time consuming; a single rewrite step can generate dozens of well-formedness goals. Several solutions to this problem are under consideration. The simplest is a scheme to cache well-formedness proofs.

Another area for future work is improving our tactic collection. Heuristics to automate induction similar to those used by Boyer and Moore [4] would be tremendously valuable. Arithmetic and equality reasoning is still time consuming and tactics for algorithms such as congruence closure could offer much assistance. Such reasoning is especially tedious when it involves division and remainders. There are plans to build up a library of basic number theory facts and related tactics.

Acknowledgements

The authors wish to thank Professors Miriam Leeser and Geoffrey Brown for discussions and careful proof-reading of this document. Also, DelVecchio gives special thanks to Suzanne Nobrega for her inspiration in this effort.

References

- [1] G. Barrett. Formal methods applied to a floating point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.
- [2] David A. Basin. *Implementing Problem Solving Environments In Constructive Type Theory*. PhD thesis, Cornell University, 1990. To appear.
- [3] David A. Basin. Building theories in Nuprl. In *Logic At Botik, '89*, 1989. To Appear.
- [4] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.
- [5] A. J. Camillieri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borriore, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. North Holland, September 1986.
- [6] A. Cohn. A proof of correctness of the VIPER microprocessor: The first level. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 27–72. Kluwer, 1988.
- [7] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.
- [8] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
- [9] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.
- [10] Warren J. Hunt. The mechanical verification of a microprocessor design. In D. Borriore, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 89–129. Elsevier Science Publishers B. V. (North-Holland), 1987.
- [11] INMOS. T800 architecture. INMOS Technical note 6.
- [12] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
- [13] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.