Diplomarbeit

Formal Verification of an IEEE Floating Point Adder

Formale Verifikation eines IEEE-Gleitkomma-Addierers

Christoph Berg
cb@cs.uni-sb.de

Universität des Saarlandes
Fachrichtung 6.2 – Informatik
Lehrstuhl für Rechnerarchitektur und Parallelrechner
Prof. Dr. Wolfgang J. Paul

Mai 2001

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, daß ich die vorliegende Arbeit selbstständig verfaßt und nur die angegebenen Quellen benutzt habe. Ich habe diese Arbeit keinem anderen Prüfungsamt vorgelegt.

Saarbrücken, im Mai 2001

Christoph Berg

Im Gebirge der Wahrheit kletterst du nie umsonst: entweder du kommst schon heute weiter hinauf oder du übst deine Kräfte, um morgen höher steigen zu können.

—

In the mountains of truth you never climb in vain: either you ascend today, or you practise your strengths for climbing higher tomorrow.

Friedrich Nietzsche

Danke

Meinen Dank aussprechen möchte ich all denjenigen, die mich beim Entstehen dieser Diplomarbeit unterstützt haben:

- Christian Jacobi, der vom Betreuer meiner Arbeit zu einem guten Freund wurde, und dessen Unterstützung eine große Hilfe war,
- Prof. Wolfgang Paul, der mir durch seine Vorlesungen und die Vergabe des Themas ermöglicht hat, an einem spannenden Projekt zu arbeiten,
- Jochen Preiß, Daniel Kröning, Sven Beyer, Dirk Leinenbach, Michael Klein und Stefan Kunde für die „soziale Infrastruktur“ am Lehrstuhl,
- meinen Freunden, die mir genügend Ablenkung vom Streß geboten haben, der sich beim Arbeiten immer wieder einstellt,
- und nicht zuletzt meinen Eltern, die mich auf meinem universitären Werdegang immer wieder ermutigt haben.

Abstract

Our group at Saarland University is formally verifying the correctness of a complete microprocessor called VAMP. The PVS theorem prover is used to specify the circuit definitions and to prove their correctness. For the VAMP project, a library of basic circuits was developed. The formal verification of this library is described in the first part of this thesis. Part of the VAMP is an IEEE compliant floating point unit. The second part of this thesis describes the formal verification of the gate level correctness of the VAMP floating point adder.

Zusammenfassung

Die Korrektheit eines vollständigen Mikroprozessors, dem VAMP, wird von unserer Gruppe an der Universität des Saarlandes formal verifiziert. Der Theorembeweiser PVS wird benutzt, um die Schaltkreis-Definitionen zu spezifizieren und deren Korrektheit zu zeigen. Für das VAMP-Projekt wurde eine Bibliothek von Standard-Schaltkreisen entwickelt. Die formale Verifikation dieser Bibliothek ist im ersten Teil dieser Arbeit beschrieben. Teil des VAMP ist eine IEEE-Gleitkomma-Einheit. Die formale Verifikation der Korrektheit des VAMP-Gleitkomma-Addierers auf Gatterebene ist Thema des zweiten Teils dieser Diplomarbeit.

Contents

Chapter 1

Introduction

Floating point hardware consists of complex circuits that tend to have subtle errors. Design flaws are usually eliminated by testing, but it is impossible to test every state the circuit can enter. Even millions of test vectors were unable to find the 1995 Pentium division bug [Pra95].

Proving the correctness using mathematical reasoning overcomes this limitation. But reasoning about single bits in paper-and-pencil proofs is tedious and error prone, so the correctness of the design is not entirely certain.

Formal verification—using theorem proving or other formal techniques—provides a way to rigorously prove the correctness of a design. *Theorem proving* is a general framework for formal reasoning in logic. It allows to prove very complex statements and can—using abstraction or induction—even reason about infinite state spaces. The disadvantage is that a considerable amount of manual work is needed. Finding proofs in a theorem prover is essentially as hard as finding proofs in mathematics.

This thesis covers the formal verification of a library of basic circuits and an IEEE floating point adder using theorem proving.

The VAMP project. Müller and Paul design a complete floating point unit (FPU) on the gate level in their textbook [MP00]. The FPU features an addition/subtraction unit, a multiplication/division unit, a rounder common to the functional units, conversion to/from integer operands, and floating point comparison. Along with the designs come paper-and-pencil proofs for the correctness of the circuits.

The FPU from [MP00] is embedded into the DLX processor. The DLX is a RISC processor based on the MIPS instruction set architecture [HP96]. Features of the DLX implementation in [MP00] are a 5-stage pipeline, precise and nested interrupts, delayed branch, and a cache memory interface. [MP00] also includes paper-and-pencil proofs for the correctness of the DLX integer core.

In the *VAMP* project, our group formally verifies the DLX processor [JK00, BJ01, Krö01, Jac01b, VAM] using the PVS theorem prover [OSR92]. The VAMP—standing for *Verified Architecture Microprocessor*—is an implementation of the paper designs from [MP00] in the PVS language. We verify the correctness of the [MP00] proofs. We successively add new features to the VAMP and formally verify them. The major improvement over the DLX from [MP00] is the implementation of a Tomasulo scheduler. A cache memory interface with TLB is being worked

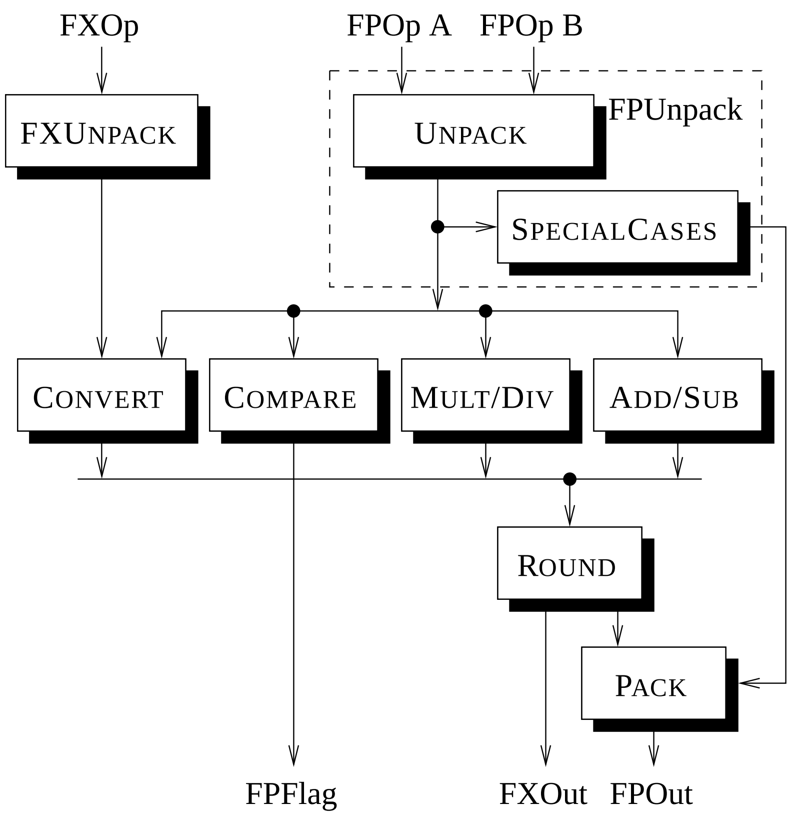

Figure 1.1: The VAMP FPU

on. The topic of this thesis is the formal verification of the VAMP floating point adder correctness.

Our group has implemented a translation tool that converts the hardware specifications from PVS to Verilog HDL [BJKL01]. The Verilog files of the VAMP processor are compiled for a Xilinx FPGA, which is hosted on a PCI board. We are porting the *gcc* compiler and the *GNU C* library to the VAMP architecture to evaluate the verified VAMP processor.

Basic circuits. The correctness of the basic circuits library used in the VAMP processor is proved in the first part of this thesis [BJK01]. Basic circuits are components like adders, multipliers, shifters, and decoders that are used frequently in hardware design. The library has been developed for the VAMP project, but may be used in any hardware verification project, as the circuits are of arbitrary bit width and for general use.

FPU verification. For the verification of floating point hardware, a formalization of the IEEE standard 754 [Ins85] is needed [Jac01b, Jac01a]. Based on this abstraction level, the FPU modules are verified.

The VAMP FPU consists of three major parts (figure 1.1). The unpacker is given the FPU input operands in the IEEE format and converts them to an internal representation; the unpacker also handles integer operands and special cases as operations with NaN or infinite operands. The functional units then compute the operation's result. In the last step, the rounding unit rounds the result and packs it into the IEEE format.

Other than the adder in [MP00], the adder design in this thesis does not handle special cases, as this is done by the unpacker. The adder is a combinatorial circuit. Pipelining circuits by inserting registers and building a functional unit for the Tomasulo scheduler is described in [Jac01b].

Project status. For his PhD thesis, Daniel Kröning has verified the correctness of the VAMP integer core and implemented a verified Tomasulo scheduler [Krö01]. Christian Jacobi has implemented the VAMP FPU functional modules (except for the adder) and the pipeline control for the FPU. The verification of these will be described in his PhD thesis [Jac01b]. The verification of the floating point adder hardware is the topic of this thesis. Sven Beyer has started to work on the verification of the cache memory interface and TLB. Dirk Leinenbach and Sven Beyer have implemented the PVS to Verilog translation tool [BJKL01].

The PVS files—hardware specifications and proofs—and the Verilog sources of the VAMP are publically available at our web site [VAM].

Outline. Chapter 2 is a brief introduction to the PVS theorem prover. A simple example is given to make the reader familiar with PVS proofs. Notations and lemmas used in the following chapters are introduced. Chapter 3 covers the basic circuits library. For each circuit, the circuit implementation is specified and the correctness is proved. Chapter 4 is an overview on the IEEE floating point arithmetic formalization used. The correctness of the floating point adder is proved in chapter 5. A summary and an overview on related work are given in chapter 6.

Chapter 2

The PVS Theorem Prover

PVS (short for *Prototype Verification System*) is a general purpose theorem prover [OSR92]. This chapter gives a brief introduction to the PVS logic, presents a simple example proof to make the reader familiar with PVS, and introduces some of the most important PVS commands. For a further treatment, see the tutorial [COR$^+$95]. At the end of the chapter, we introduce notations, definitions, and lemmas that will be used in the following chapters.

2.1 The PVS Logic

PVS is based on *typed higher order logic* (HOL). Predefined types include $bool$, nat, int, $real$, bit, … From these, sets, functions, tuples, and records are derived. Each PVS file consists of one or more *theories* which in turn contain definitions, lemmas, theorems, and axioms written as HOL *formulas*.

Within proofs, PVS operates on *sequents* of the form $\Gamma \vdash \Delta$, where Γ and Δ are lists of HOL formulas, called *antecedents* and *consequents*, respectively [Gen35]. For the proof to succeed, we have to show that the disjunction of the consequents is a logical consequence of the conjunction of the antecedents:[1]

$$\gamma_{-1}, \ldots, \gamma_{-i} \vdash \delta_1, \ldots, \delta_j \quad \text{is to be read as} \quad \gamma_{-1} \wedge \cdots \wedge \gamma_{-i} \implies \delta_1 \vee \cdots \vee \delta_j\,.$$

A proof of a lemma F starts with the sequent $\vdash F$. Sequents are modified by *proof commands*. Proof commands transform a sequent into several—possibly zero—child sequents. These sequents are subcases of the parent sequent; their conjunction implies the validity of the parent sequent. The proof sequents form a tree. A proof branch is *closed* if the last proof command yielded no children, i.e., if PVS was able to verify the validity of the sequent. A proof is *finished* when there are no open branches left in the proof.

PVS has to verify that all types used in the specification are well-defined and generates so-called *TCCs* (type correctness conditions) if it encounters any non-trivial type usage. For example, in the expression $\sqrt{n}$, n must not be negative. The user then has to prove the TCCs to show that the specification is sound. In most cases, TCCs are simple statements like $\forall\ n \in \mathbb{N}^+ : n - 1 \geq 0$ that can be proved automatically. Sometimes, however, TCCs require a substantial amount of manual work.

[1]Antecedent and consequent formulas are numbered using negative and positive integers, respectively.

```
gauss: Theory
Begin

  Importing bitvectors@sums

  n, i: Var nat

  sum(n): nat = sigma(0, n, Lambda i: i)

  recsum(n): Recursive nat =
    If n = 0 Then 0 Else n + recsum(n-1) EndIf
    Measure n

  gauss(n): real = n * (n + 1) / 2

  sum_is_recsum: Lemma
    sum(n) = recsum(n)

  recsum_is_gauss: Lemma
    recsum(n) = gauss(n)

  sum_is_gauss: Theorem
    sum(n) = gauss(n)

End gauss
```

Figure 2.1: PVS file gauss.pvs

A PVS lemma is called *proved* if its associated proof is finished. It is called *complete* if all lemmas used in the proof and all TCCs generated are proved and complete themselves. Only lemmas that are proved and complete may be considered valid, as unproved lemmas that are used in proofs may be unsound.

2.2 An Example Proof

To give an intuition about how the PVS theorem prover works, we present a rather simple example: the proof of GAUSS's theorem.

Theorem 1 (GAUSS) *For all $n \in \mathbb{N}$:*

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

We define three functions whose equality we want to prove using PVS (figure 2.1):

1. `sum(n): nat = sigma(0, n, Lambda i: i)`

   `sum` uses the `sigma` function. `sigma` sums up a *function* over a finite natural domain. The term i in theorem 1 is an *expression*, not a function. We therefore use the λ term $(\lambda\, i : i)$—

which is the identity function—in the formalization of the Σ operator. In mathematical notation, this is

$$sum(n) := \sum_{0}^{n} (\lambda\, i : i).$$

2. `recsum(n): Recursive nat =`
   `If n = 0 Then 0 Else n + recsum(n-1) EndIf`
   `Measure n`

   Another formalization is given as the recursive function `recsum`. Recursive functions have to be well-founded, i.e., the recursion must terminate eventually; a *measure* has to be supplied whose natural value must decrease with each recursive call. In our case this is trivially true since $n - 1 < n$.

$$recsum(n) := \begin{cases} 0 & \text{if } n = 0 \\ n + recsum(n-1) & \text{else.} \end{cases}$$

3. `gauss(n): real = n * (n + 1) / 2`

   Finally, `gauss` is GAUSS's formula.

We aim to prove GAUSS's theorem, formalized in `sum_is_gauss`. To clarify proof techniques commonly used in PVS, we split this theorem into the two lemmas `sum_is_recsum` and `rec-sum_is_gauss`.

2.2.1 Proof of `sum_is_recsum`

We first show that `sum` and `recsum` are equivalent.

Lemma 1 `sum_is_recsum:`

```
sum(n) = recsum(n)
```

Proof. After starting the PVS prover, we are presented the first sequent.

```
sum_is_recsum :

  |-------
{1}   FORALL (n: nat): sum(n) = recsum(n)

Rule?
```

Note that PVS has augmented the lemma by a $\forall$ quantor binding the free occurrence of n. We aim to prove the lemma by induction on n. We start with the PVS command `(induct "n")`, which yields two subgoals that correspond to induction base and step.

```
Rule? (induct "n")
Inducting on n on formula 1,
this yields  2 subgoals:
sum_is_recsum.1 :

  |-------
{1}   sum(0) = recsum(0)

Rule?
```

Proof branches are named by the prover. Here, the induction base is named `sum_is_recsum.1`. This subgoal is resolved by `(grind)`. This proof command repeatedly expands all definitions and applies various simplification rules. `(grind)` is one of the most powerful proof commands, but in some cases, it will not terminate if recursive definitions are unwinded ad infinitum. Another possibility to prove the induction base was to expand sum, $sigma$, and $recsum$ manually (using `(expand)`), but `(grind)` is more compact.

```
Rule? (grind)
sigma rewrites sigma(0, 0, LAMBDA i: i)
  to 0
sum rewrites sum(0)
  to 0
recsum rewrites recsum(0)
  to 0
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of sum_is_recsum.1.

sum_is_recsum.2 :

  |-------
{1}   FORALL j: sum(j) = recsum(j) IMPLIES
        sum(j + 1) = recsum(j + 1)

Rule?
```

The induction step is named `sum_is_recsum.2`. We replace the quantified j by an arbitrary, but fixed j_1 (a so called *skolem constant*) using `(skosimp*)`. This PVS command also simplifies the sequent: in this case, it splits the implication in formula 1 into an antecedent part -1 and a consequent part 1.

```
Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
sum_is_recsum.2 :

{-1}  sum(j!1) = recsum(j!1)
  |-------
{1}   sum(j!1 + 1) = recsum(j!1 + 1)
```

```
Rule?
```

We have to prove that the equality in formula 1 (the induction claim) is an implication of formula -1 (the induction hypothesis). We want to use properties of the `sigma` operator which is currently hidden in our function `sum`. We therefore expand the definition of `sum`:

```
Rule? (expand "sum")
Expanding the definition of sum,
this simplifies to:
sum_is_recsum.2 :

{-1}  sigma(0, j!1, LAMBDA i: i) = recsum(j!1)
  |-------
{1}   sigma(0, 1 + j!1, LAMBDA i: i) = recsum(1 + j!1)

Rule?
```

Before we can apply the induction hypothesis, we use the lemma SIGMA SPLIT from the PVS bitvectors library. The command `(lemma "sigma_split")` introduces the lemma as a new antecedent formula.

```
Rule? (lemma "sigma_split")
Applying sigma_split
this simplifies to:
sum_is_recsum.2 :

{-1}  FORALL (F: [nat -> nat], high, low, m: nat):
        m >= low AND high > m IMPLIES
         sigma(low, high, F) =
          sigma(low, m, F) + sigma(m + 1, high, F)
[-2]  sigma(0, j!1, LAMBDA i: i) = recsum(j!1)
  |-------
[1]   sigma(0, 1 + j!1, LAMBDA i: i) = recsum(1 + j!1)

Rule?
```

This lemma allows us to split a `sigma` term over F ranging from low to $high$ at any intermediate m. In our case, $F = (\lambda\, i : i)$, $high = j_1 + 1$, $low = 0$, and $m = j_1$. Thus, we instantiate formula -1 accordingly:[2]

```
Rule? (inst -1 "LAMBDA i: i" "j!1+1" "0" "j!1")
Instantiating the top quantifier in -1 with the terms:
 LAMBDA i: i, j!1+1, 0, j!1,
this simplifies to:
```

[2]*Instantiating* a formula means to replace a $\forall$ quantor over a variable in the antecedents (or equivalently, an $\exists$ quantor in the consequents) by a specific value of the proper type for the variable.

```
sum_is_recsum.2 :

{-1}  j!1 >= 0 AND j!1 + 1 > j!1 IMPLIES
       sigma(0, j!1 + 1, LAMBDA i: i) =
        sigma(0, j!1, LAMBDA i: i) +
         sigma(j!1 + 1, j!1 + 1, LAMBDA i: i)
[-2]  sigma(0, j!1, LAMBDA i: i) = recsum(j!1)
  |-------
[1]   sigma(0, 1 + j!1, LAMBDA i: i) = recsum(1 + j!1)

Rule?
```

The lemma's prerequisites $j_1 \geq 0 \wedge j_1 + 1 > j_1$ require j_1 to lie in the range $low \ldots high$, which is true. By `(assert)`, we invoke the PVS decision procedures for linear arithmetic, thereby simplifying the formula and eliminating the redundant prerequisites. We are left with the equation

$$\sum_{0}^{j_1+1}(\lambda\, i : i) = \sum_{0}^{j_1}(\lambda\, i : i) + \sum_{j_1+1}^{j_1+1}(\lambda\, i : i).$$

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sum_is_recsum.2 :

{-1}  sigma(0, 1 + j!1, LAMBDA i: i) =
       sigma(0, j!1, LAMBDA i: i) +
        sigma(1 + j!1, 1 + j!1, LAMBDA i: i)
[-2]  sigma(0, j!1, LAMBDA i: i) = recsum(j!1)
  |-------
[1]   sigma(0, 1 + j!1, LAMBDA i: i) = recsum(1 + j!1)

Rule?
```

The left side of equation -1 (the instantiated lemma SIGMA SPLIT) is equal to the left side of equation 1. To apply the lemma, we *replace* any occurrence of the left side of equation -1 by the right side. Since we do not need the lemma afterwards, we hide the formula.

```
Rule? (replace -1 :hide? t)
Replacing using formula -1,
this simplifies to:
sum_is_recsum.2 :

[-1]  sigma(0, j!1, LAMBDA i: i) = recsum(j!1)
  |-------
{1}   sigma(0, j!1, LAMBDA i: i) +
       sigma(1 + j!1, 1 + j!1, LAMBDA i: i) =
        recsum(1 + j!1)
```

```
Rule?
```

The above four steps—importing a lemma, instantiating it, discharging its prerequisites, and replacing subterms—are the usual way to use lemmas in PVS.

We can now apply the induction hypothesis in formula -1, again by `(replace)`:

```
Rule? (replace -1 :hide? t)
Replacing using formula -1,
this simplifies to:
sum_is_recsum.2 :

  |-------
{1}   recsum(j!1) + sigma(1 + j!1, 1 + j!1, LAMBDA i: i) =
       recsum(1 + j!1)

Rule?
```

The remaining `sigma` term is trivial, `(expand "sigma")` reduces it to $1 + j_1$. Note that PVS tries to generate normal forms of expressions and moves the 1 to the front.

```
Rule? (expand "sigma")
Expanding the definition of sigma,
this simplifies to:
sum_is_recsum.2 :

  |-------
{1}   1 + recsum(j!1) + j!1 = recsum(1 + j!1)

Rule?
```

Next, we unwind one step of the recursion in the second occurrence of `recsum` in formula 1 by `(expand "recsum" 1 2)`. This yields $1 + recsum(j_1) + j_1$ on the right side, which is *syntactically* equivalent to the left side. The PVS prover recognizes this and finishes the proof automatically.

```
Rule? (expand "recsum" 1 2)
Expanding the definition of recsum,
this simplifies to:
sum_is_recsum.2 :

  |-------
{1}   TRUE

which is trivially true.
```

```
This completes the proof of sum_is_recsum.2.

Q.E.D.


Run time  = 0.50 secs.
Real time = 144.64 secs.
```

The proof subgoals generated by the proof commands form a tree representing the proof's structure. PVS visualizes this tree during the proof attempt in a separate window (figure 2.2). □

2.2.2 Proof of `recsum_is_gauss`

To demonstrate a less 'manual' proof, we prove the next lemma using a more powerful proof command.

Lemma 2 `recsum_is_gauss`:

```
recsum(n) = gauss(n)
```

Proof.

```
  |-------
{1}   FORALL (n: nat): recsum(n) = gauss(n)
```

One step, namely `(induct-and-simplify "n")`, suffices to prove this goal. This command starts an induction and repeatedly expands and simplifies expressions. □

2.2.3 Proof of `sum_is_gauss`

Finally, we can proceed to prove our original goal, the theorem `sum_is_gauss`.

Theorem 2 `sum_is_gauss`:

```
sum(n) = gauss(n)
```

Proof.

```
  |-------
{1}   FORALL (n: nat): sum(n) = gauss(n)
```

We first have to get rid of the quantor by using `(skosimp*)`, as in the `sum_is_recsum` proof.

```
  |-------
{1}   sum(n!1) = gauss(n!1)
```

Manually importing a lemma, instantiating it and replacing sub-terms using the equality in the lemma—as in our first proof—is a task that can be automated if PVS is able to guess the right instantiation. We now use the two previously proved lemmas to *rewrite* the formula in our current goal. `(rewrite "sum_is_recsum")` replaces the occurrence of `sum` by the corresponding `recsum` term:

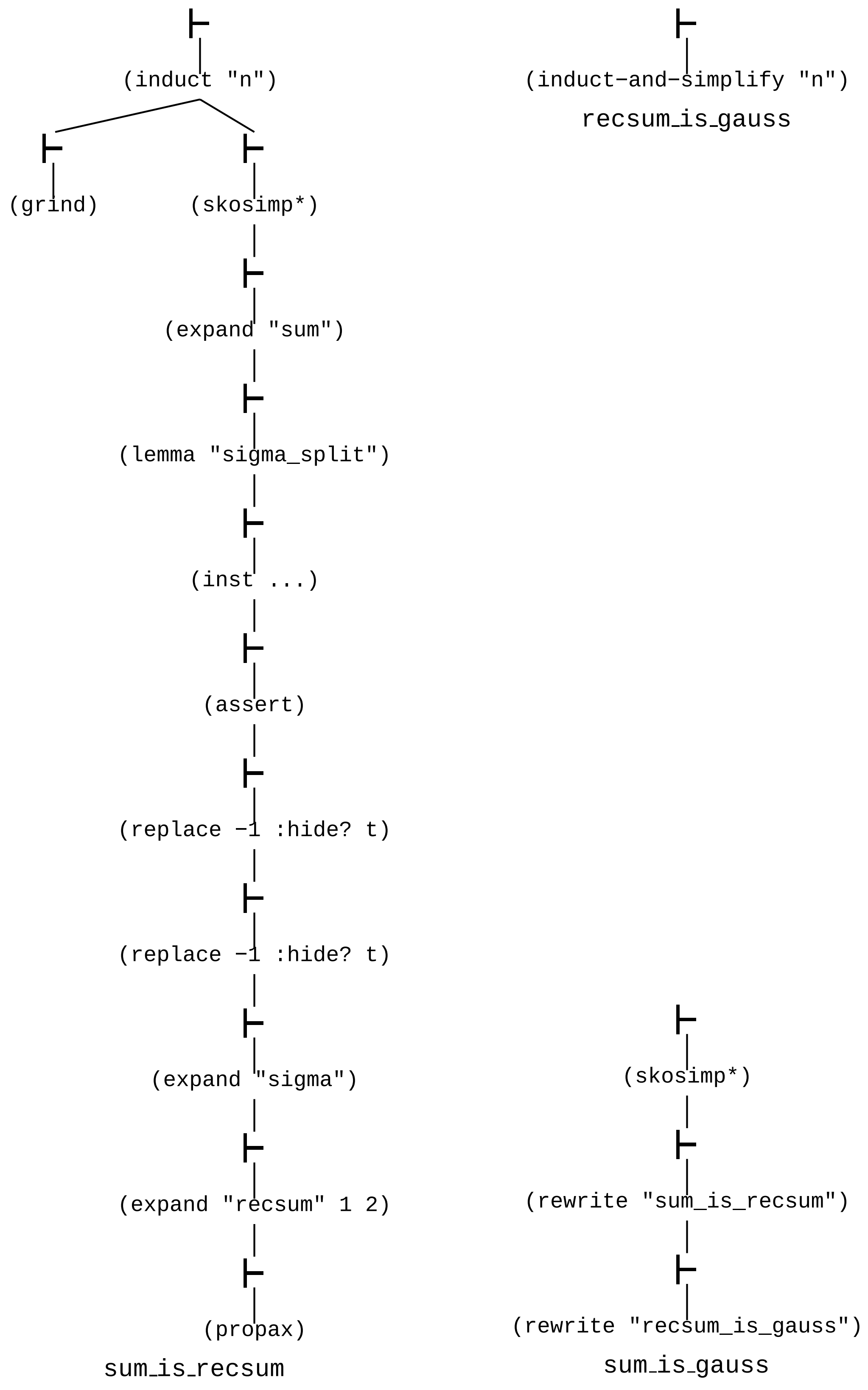

Figure 2.2: The `sum_is_recsum`, `recsum_is_gauss`, and `sum_is_gauss` proof trees

```
  |-------
{1}   recsum(n!1) = gauss(n!1)
```

Finally, `(rewrite "recsum_is_gauss")` replaces `recsum` by `gauss`, leading to the trivial sequent `gauss(n!1) = gauss(n!1)`, which PVS recognizes as true. The theorem is proved; it is complete as well, since all lemmas we used are proved and complete. □

The above proofs demonstrate some of the most important proof techniques used in PVS. We did not try to make the proofs as short as possible, but to give the reader the intuition of how PVS proofs work. In fact, the step `(induct-and-simplify "n")` that was used to prove lemma `recsum_is_gauss` would have resolved the first lemma, `sum_is_recsum`, immediately. Of course, more difficult theorems need many more steps, and our small proofs may be regarded as 'light' proof examples for more complex properties we want to verify.

2.2.4 TCCs

Two TCCs are generated for the gauss theory, both for the recursive call `recsum(n-1)` in the `recsum` function. Since `recsum` is only defined on natural numbers, we have to prove that $n-1$ is non-negative. But `recsum(n-1)` is only called if $n \neq 0$ and hence the TCC

```
recsum_TCC1: OBLIGATION
  FORALL (n): NOT n = 0 IMPLIES n - 1 >= 0
```

is easily proved via `(assert)`. The second TCC assures the termination of recursive `recsum` calls. The measure n supplied in the `recsum` definition must decrease with each recursive call. This TCC is trivially discharged as well.

```
recsum_TCC2: OBLIGATION
  FORALL (n): NOT n = 0 IMPLIES n - 1 < n
```

Although we know that `gauss` returns a natural, we assigned it a type of $real$. Otherwise, we had to prove a third TCC that $\frac{n(n+1)}{2}$ was natural, which we do not want to do here.[3]

2.3 PVS and Mathematics

Syntax. The reader should be convinced by now that the PVS syntax closely resembles mathematical notation, and translating between both is straightforward. For readability, we will use conventional mathematical notation in the remaining part of this thesis.

Proofs. In textbooks, circuits are usually defined by using figures, and correctness proofs argue about these figures. Translating hardware correctness proofs from conventional mathematics to PVS means to formalize the figures in PVS, and then to adapt the proofs using proof techniques exploiting the capabilities of PVS. If the proofs are not entirely formal or use lemmas that are not

[3]A possible approach would be to exploit the fact that `gauss` is equal to `sum` which is known to be natural.

available in PVS, the proofs have to redone from scratch, or several auxillary lemmas have to be proved.

Conventional mathematical proofs use shortcuts where the line of reasoning is obvious—or at least supposed to be obivous. Usually, these ‘proof gaps’ are marked by flowery phrases like *trivial*, *obvious*, *analogous*, and *without loss of generality*. The gaps have to be filled for the formal verification in PVS.

Contrarily, translating PVS proofs back to mathematical notation means to ‘extract’ the essential proof commands from a proof tree, and to provide the necessary intuition on the proof goal. Only about one in every three or four PVS proof commands is worth being mentioned in the mathematical transcript, the others being uninteresting—or in other words trivial—transformations like evaluating expressions (e.g., $1 - 1 = 0$), or using associativity, commutativity, and such. Of course, we will mark these steps using phrases like those mentioned above, but we hope that the reader will be confident—due to the formal verification of the proofs in PVS—that there are no proof gaps left.

2.4 Bitvectors

PVS provides a bitvectors library that provides bits, bitvectors, and a rich collection of lemmas for bitvector transformation and arithmetic [BMS+96].

Notation. PVS defines the type bit, which we will denote by $\mathbb{B}$, to be the type $boolean := \{false, true\}$. For convenience, we will interpret $\mathbb{B}$ as well as the set $\{\mathbf{0}, \mathbf{1}\}$. A bitvector of width n is a function mapping the domain $\{n-1, \dots, 0\}$ to $\mathbb{B}$. We denote the bitvector type by $\mathbb{B}^n$. We will implicitly identify bits $\mathbb{B}$ and bitvectors $\mathbb{B}^1$ of width 1.

A bitvector b of width n is indexed by $b[i]$ with i ranging from $n-1$ to 0. For $h \geq l$, $b[h, l]$ denotes the extracted bit vector consisting of bits h to l of b; $\circ$ is the concatenation operator. For $x \in \mathbb{B}$, x^i denotes the bitvector consisting of bit x repeated i times. The bit operators $\neg$, $\wedge$, and $\vee$ will be applied to bitvectors as well, meaning bitwise application.

The natural number represented by b is denoted by

$$\langle b \rangle := \sum_{i=0}^{n} 2^i \cdot b[i].$$

The two’s complement value of b is

$$[b] := \begin{cases} \langle b \rangle & \text{if } \langle b \rangle < 2^{n-1} \\ \langle b \rangle - 2^n & \text{else.} \end{cases}$$

The range of the n-bit two’s complement numbers is denoted by

$$T_n := \{-2^{n-1}, \dots, 2^{n-1} - 1\}.$$

The proof that T_n is indeed the range of the n-bit two’s complement numbers can be found in the bitvectors library.

Lemmas. In the remaining part of this thesis, we need the following lemmas. Most of these come from the PVS bitvectors library; we do not give the proofs. Unless otherwise noted, let $n \in \mathbb{N}^+$ and $b \in \mathbb{B}^n$.

Lemma 3

$$\langle b \rangle = b[n-1] \cdot 2^{n-1} + \langle b[n-2,0] \rangle \,.$$

Lemma 4 *For all $n, m \in \mathbb{N}^+$, $bn \in \mathbb{B}^n, bm \in \mathbb{B}^m$:*

$$\langle bn \circ bm \rangle = \langle bn \rangle \cdot 2^m + \langle bm \rangle \,.$$

Lemma 5

$$\langle b \rangle \geq 2^{n-1} \iff b[n-1].$$

Lemma 6 *For all $l \in \mathbb{N}_{<n}$:*

$$\langle b \rangle < 2^l \iff b[n-1,l] = \mathbf{0}^{n-l}.$$

Lemma 7 *For all $l \in \mathbb{N}_{<n}$:*

$$\langle b[l,0] \rangle = \langle b \rangle \bmod 2^{l+1}.$$

Lemma 8 *For all $l \in \mathbb{N}_{<n}$:*

$$\langle b[n-1,l] \rangle = \langle b \rangle \operatorname{div} 2^l.$$

Lemma 9

$$\langle \neg\, b \rangle = 2^n - \langle b \rangle - 1.$$

Lemma 10

$$[b] = \langle b \rangle - 2^n \cdot b[n-1].$$

Lemma 11

$$[b] = \langle b[n-2,0] \rangle - 2^{n-1} \cdot b[n-1].$$

Lemma 12

$$[b] < 0 \iff b[n-1].$$

Lemma 13

$$-[b] = [\neg\, b] + 1.$$

Lemma 14

$$\langle b[n-1] \circ \neg\, b[n-1] \circ b[n-2,0] \rangle = 2^{n-1} + \langle b \rangle \,.$$

Chapter 3

Basic Components

Large circuits are built from smaller modules that tend to be used frequently. Therefore, it is profitable to collect these standard modules in a library of basic components. This section describes the library developed as part of this thesis. The library consists of various combinatorial circuits as listed in table 3.1. A summary of this chapter has been published as [BJK01].

Each component is specified as a PVS function. The correctness criterion is formulated as a lemma stating the intended circuit behaviour using a mathematical formula. The correctness criterion is then used as a rewrite rule in proofs of larger circuits that use the component. The circuits are of arbitrary bit width and are designed for general use. Some circuits, however, are limited to bit widths that are powers of 2.

The designs and most proofs were taken from [MP95, KP97, MP00]. As we will only prove the correctness of the circuits, the reader should refer to the cited publications for a further explanation of the circuit functionality. Cost and delay in table 3.1 are asymptotic measures, i.e., n means $O(n)$. We will not give proofs for these here.

The goal of the VAMP project is to obtain a completely verified CPU. The designs are not necessarily optimized for speed. There are faster adder designs than carry chain adders, but as the correctness of the design does not depend on the adder implementation used (as long as the adder is correct), we use the simple, but slow carry chain implementation. Another reason for carry chain adders is the FPGA implementation: due to fast carry lines, the carry chain adder is the most efficient adder on Xilinx FPGAs [Xil00].

Unless noted otherwise, n is a positive natural number in this chapter: $n \in \mathbb{N}^+$.

3.1 Halfadder, Fulladder

Halfadder. The halfadder takes two bits a and c and computes a bitvector of length 2 representing the sum $a + c$ (figure 3.1).

Circuit 1 *Inputs* $a, c \in \mathbb{B}$, *output* $s \in \mathbb{B}^2$.

$$\mathit{halfadder} := (a \wedge c) \circ (a \oplus c).$$

component	symbol	width	cost	delay
halfadder	$halfadder$	1	1	1
fulladder	$fulladder$	1	1	1
carry chain incrementer	$carry\_chain\_inc$	n	n	n
carry chain adder	$carry\_chain$	n	n	n
compound adder	$compound$	n	$n \log n$	$\log n$
generic adder	Add	n	n	n
carry save adder	csa	n	n	1
arithmetic unit	add_sub	n	n	n
subtract unit	sub	n	n	n
absolute value	abs	n	n	n
linear multiplier	$mult\_lin$	n, m	$n \cdot m$	$n + m$
decoder	dec	n	2^n	$\log n$
half decoder	$hdec$	n	2^n	n
encoder	enc	2^n	2^n	n
leading zero counter	lz	2^n	n	$\log n$
cyclic left shifter	cls	2^n	$n \log n$	$\log n$
logic left shifter	lls	n	$n \log n$	$\log n$
logic right shifter	lrs	n	$n \log n$	$\log n$
or tree	$ortree$	n	n	$\log n$

Table 3.1: The components contained in the library

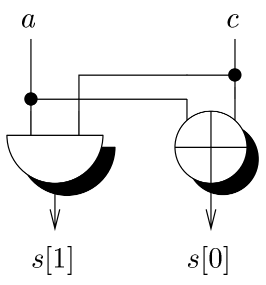

Figure 3.1: Halfadder HA

Lemma 15 (HALFADDER CORRECT) *For all* $a, c \in \mathbb{B}$ *:*

$$\langle halfadder(a, c)\rangle = a + c.$$

Proof. Since the halfadder is a small circuit of fixed size, its correctness is automatically verified by using the PVS command `(grind)`. □

Fulladder. Similar to the halfadder, the fulladder sums up three bits (figure 3.2).

Circuit 2 *Inputs* $a, b, c \in \mathbb{B}$, *output* $s \in \mathbb{B}^2$. *Let* $x := a \oplus b$.

$$fulladder := \big((a \wedge b) \vee (c \wedge x)\big) \circ (x \oplus c).$$

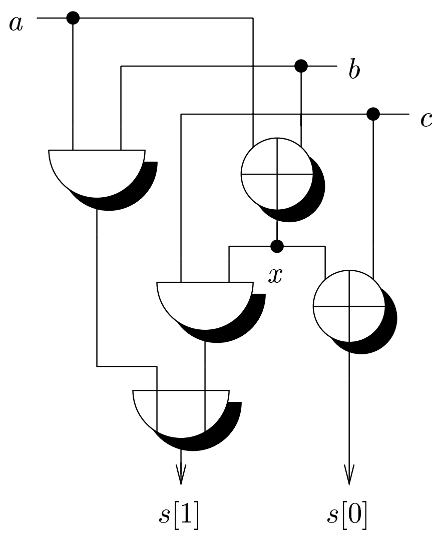

Figure 3.2: Fulladder FA

Lemma 16 (FULLADDER CORRECT) *For all* $a, b, c \in \mathbb{B}$ *:*

$$\langle fulladder(a, b, c)\rangle = a + b + c.$$

Proof. Trivial by `(grind)`. □

3.2 Carry Chain Incrementer

Our first non-trivial circuit is a carry chain incrementer that is built from halfadders (figure 3.3). Depending on a carry-in signal c_{in}, the circuit increments a by one or passes it unmodified. The carry chain incrementer is defined recursively for bitwidths n. We use indices to refer to a specific circuit instance, e.g., $carry\_chain\_inc_{n-1}$ denotes an $n-1$ bit carry chain incrementer. The index is omitted if the circuit width is clear from the context. Later, indices will also be used for circuits that have more than one output (e.g., Add_{ovf}).

Circuit 3 *Input* $a \in \mathbb{B}^n$, $c_{in} \in \mathbb{B}$, *output* $s \in \mathbb{B}^{n+1}$. *For* $n = 1$,

$$carry\_chain\_inc_1 := halfadder(a[0], c_{in}).$$

For $n > 1$, *let* $C := carry\_chain\_inc_{n-1}(a[n-2, 0], c_{in})$,

$$carry\_chain\_inc_n := halfadder(a[n-1], C[n-1]) \circ C[n-2, 0].$$

Lemma 17 (CARRY CHAIN INC CORRECT) *For all* $a \in \mathbb{B}^n$, $c_{in} \in \mathbb{B}$ *:*

$$\langle carry\_chain\_inc_n(a, c_{in})\rangle = \langle a\rangle + c_{in}.$$

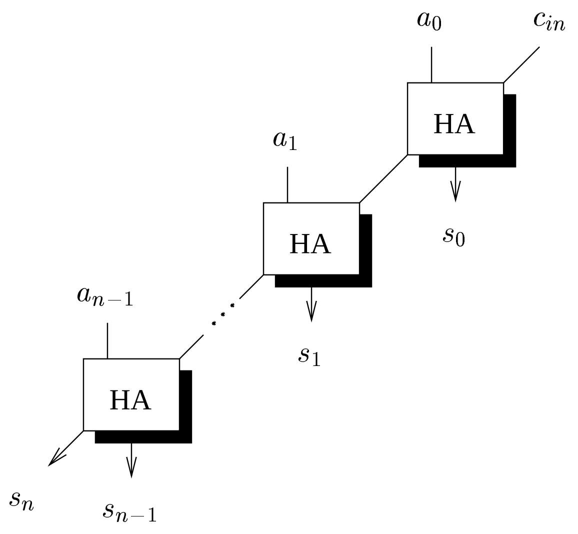

Figure 3.3: Carry chain incrementer

Proof. The proof is by induction on the width n of the input. The induction base $n = 1$ is resolved by `(grind)`.

In the induction step $n + 1$, let $C := \mathit{carry\_chain\_inc}_n(a[n-1, 0], c_{in})$. The claim is

$$\langle \mathit{carry\_chain\_inc}_{n+1}(a, c_{in})\rangle = \langle a\rangle + c_{in}.$$

Expanding the incrementer definition leads to

$$\langle \mathit{halfadder}(a[n], C[n]) \circ C[n-1, 0]\rangle = \langle a\rangle + c_{in},$$

By lemma 4, this is equivalent to

$$\langle \mathit{halfadder}(a[n], C[n])\rangle \cdot 2^n + \langle C[n-1, 0]\rangle = \langle a\rangle + c_{in}.$$

The halfadder is correct by lemma 15:

$$(a[n] + C[n]) \cdot 2^n + \langle C[n-1, 0]\rangle = \langle a\rangle + c_{in}.$$

By lemma 3 and the induction hypothesis, $\langle C\rangle = \langle a[n-1, 0]\rangle + c_{in}$, we have

$$a[n] \cdot 2^n + \langle a[n-1, 0]\rangle + c_{in} = \langle a\rangle + c_{in},$$

which is true by lemma 3. □

Incrementer. When we use an incrementer, we do not care about its actual implementation—as long as the circuit satisfies the above correctness criterion lemma 17. In the following, we will use inc to refer to the carry chain incrementer, but any other correct incrementer implementation could also be used.

$$\mathit{inc} := \mathit{carry\_chain\_inc}.$$

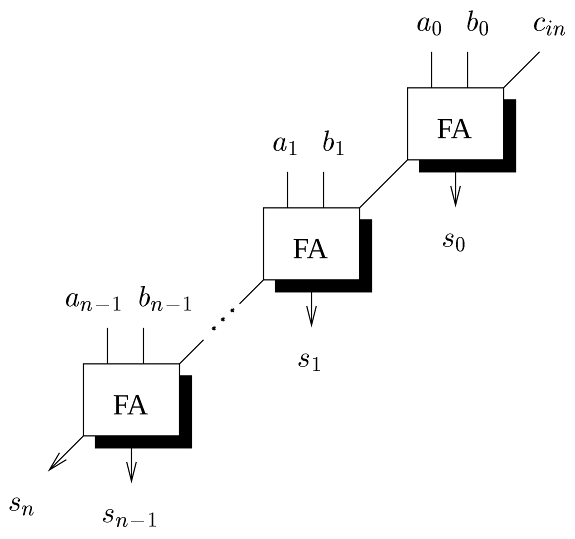

Figure 3.4: Carry chain adder

3.3 Carry Chain Adder

By substituting the halfadders in the carry chain incrementer by fulladders, we obtain a carry chain adder (figure 3.4). It adds two input numbers a and b, and a carry-in c_{in}.

Circuit 4 *Inputs* $a, b \in \mathbb{B}^n$, $c_{in} \in \mathbb{B}$, *output* $s \in \mathbb{B}^{n+1}$. *For* $n = 1$,

$$carry\_chain_1 := fulladder(a[0], b[0], c_{in}).$$

For $n > 1$, *let* $C := carry\_chain_{n-1}(a[n-2, 0], b[n-2, 0], c_{in})$,

$$carry\_chain_n := fulladder(a[n-1], b[n-1], C[n-1]) \circ C[n-2, 0].$$

Lemma 18 (CARRY CHAIN CORRECT) *For all* $a, b \in \mathbb{B}^n$, $c_{in} \in \mathbb{B}$:

$$\langle carry\_chain_n(a, b, c_{in})\rangle = \langle a\rangle + \langle b\rangle + c_{in}.$$

Proof. The proof is the same as the carry chain incrementer proof, with $halfadder$ replaced by $fulladder$. □

Adder. Analogous to inc, we define add to refer to any adder implementation satisfying lemma 18. We use the carry chain adder.

$$add := carry\_chain.$$

3.4 Compound Adder

Another adder implementation is the compound adder (figure 3.5). The compound adder features no carry-in bit, but computes *both* $a + b$ and $a + b + 1$. The compound adder is used in the floating point rounder.

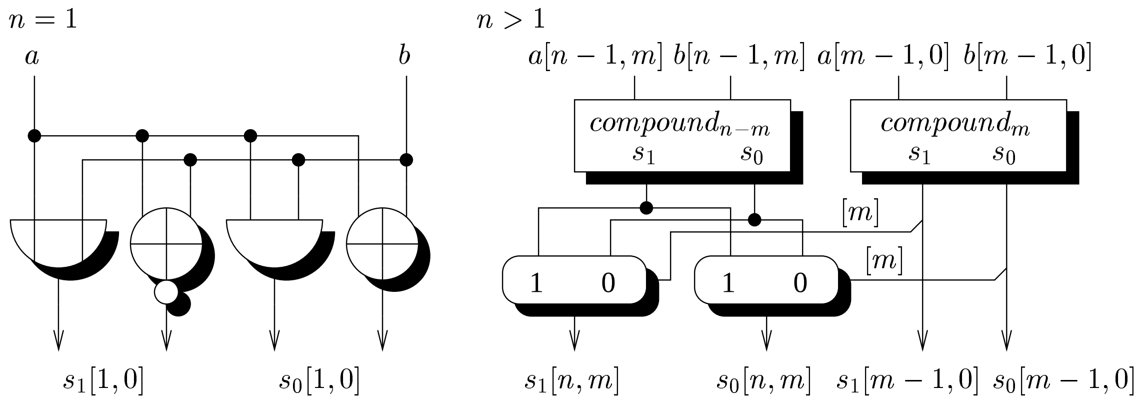

Figure 3.5: Compound adder

Circuit 5 *Inputs $a, b \in \mathbb{B}^n$, outputs $s_0, s_1 \in \mathbb{B}^{n+1}$. For $n = 1$,*

$$compound_{1,s_1} := (a[0] \vee b[0]) \circ \neg(a[0] \oplus b[0]),$$
$$compound_{1,s_0} := (a[0] \wedge b[0]) \circ (a[0] \oplus b[0]).$$

For $n > 1$, let $m := \lfloor \frac{n}{2} \rfloor$, $CH := compound_{n-m}(a[n-1,m], b[n-1,m])$, $CL := compound_m(a[m-1,0], b[m-1,0])$,

$$compound_{n,s_1} := \begin{cases} CH_{s_1} & \textit{if } CL_{s_1}[m] \\ CH_{s_0} & \textit{else} \end{cases} \circ CL_{s_1}[m-1,0],$$
$$compound_{n,s_0} := \begin{cases} CH_{s_1} & \textit{if } CL_{s_0}[m] \\ CH_{s_0} & \textit{else} \end{cases} \circ CL_{s_0}[m-1,0].$$

Lemma 19 (COMPOUND ADDER CORRECT) *For all $a, b \in \mathbb{B}^n$:*

$$\begin{array}{ll} (i) & \langle compound_{s_1}(a,b) \rangle = \langle a \rangle + \langle b \rangle + 1, \\ (ii) & \langle compound_{s_0}(a,b) \rangle = \langle a \rangle + \langle b \rangle . \end{array}$$

Proof. The base $n = 1$ of the induction on n is resolved by case analysis on $a[0]$ and $b[0]$ using `(grind)`.

In the induction step n, let $m := \lfloor \frac{n}{2} \rfloor$, $CH := compound_{n-m}(a[n-1,m], b[n-1,m])$, $CL := compound_m(a[m-1,0], b[m-1,0])$. The four induction hypotheses are

1. $\langle CH_{s_1} \rangle = \langle a[n-1,m] \rangle + \langle b[n-1,m] \rangle + 1$,
2. $\langle CH_{s_0} \rangle = \langle a[n-1,m] \rangle + \langle b[n-1,m] \rangle$,
3. $\langle CL_{s_1} \rangle = \langle a[m-1,0] \rangle + \langle b[m-1,0] \rangle + 1$,
4. $\langle CL_{s_0} \rangle = \langle a[m-1,0] \rangle + \langle b[m-1,0] \rangle$.

(i) In the case $CL_{s_1}[m]$, we show

$$\begin{aligned} \langle CH_{s_1} \circ CL_{s_1}[m-1,0] \rangle = {} & \langle a[n-1,m] \circ a[m-1,0] \rangle \\ & + \langle b[n-1,m] \circ b[m-1,0] \rangle + 1. \end{aligned}$$

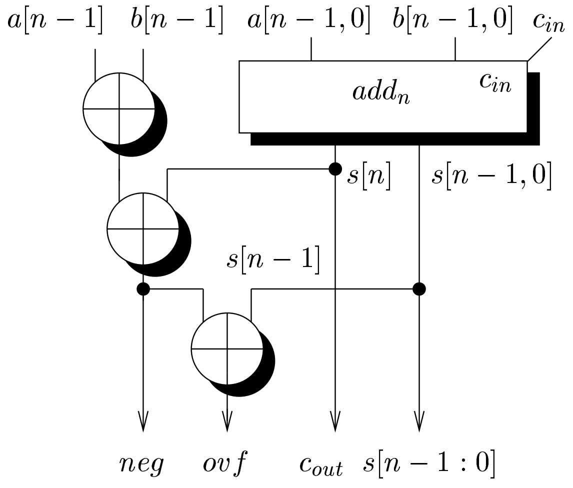

Figure 3.6: Generic adder Add

By lemma 4, this reduces to

$$\begin{aligned}\langle CH_{s_1}\rangle \cdot 2^m + \langle CL_{s_1}[m-1,0]\rangle &= \langle a[n-1,m]\rangle \cdot 2^m + \langle a[m-1,0]\rangle\\ &+ \langle b[n-1,m]\rangle \cdot 2^m + \langle b[m-1,0]\rangle + 1.\end{aligned}$$

By lemma 3, $\langle CL_{s_1}[m-1,0]\rangle = \langle CL_{s_1}\rangle - CL_{s_1}[m] \cdot 2^m$, and $CL_{s_1}[m] = \mathbf{1}$:

$$\begin{aligned}\langle CH_{s_1}\rangle \cdot 2^m + \langle CL_{s_1}\rangle - 2^m &= \langle a[n-1,m]\rangle \cdot 2^m + \langle a[m-1,0]\rangle\\ &+ \langle b[n-1,m]\rangle \cdot 2^m + \langle b[m-1,0]\rangle + 1.\end{aligned}$$

We finish this case by applying the induction hypotheses:

$$\begin{aligned}&\big(\langle a[n-1,m]\rangle + \langle b[n-1,m]\rangle + 1\big) \cdot 2^m\\ +&\big(\langle a[m-1,0]\rangle + \langle b[m-1,0]\rangle + 1\big) - 2^m = \langle a[n-1,m]\rangle \cdot 2^m + \langle a[m-1,0]\rangle\\ &\qquad + \langle b[n-1,m]\rangle \cdot 2^m + \langle b[m-1,0]\rangle + 1.\end{aligned}$$

The proofs for the case $\neg\, CL_{s_1}[m]$ and part (ii) are analogous. □

3.5 Generic Adder

The adders presented so far do not operate on two's complement numbers. In this section, we build an adder Add that provides the signals s, c_{out}, neg, and ovf. The correctness lemmas for binary and two's complement operands are proved. In the construction of Add, we use the adder add from section 3.3. We assume that add is correct, this holds by lemma 18:

$$\langle add(a, b, c_{in})\rangle = \langle a\rangle + \langle b\rangle + c_{in}.$$

In the following, let s denote $add(a, b, c_{in})$.

Circuit 6 *Given a circuit add, inputs $a, b \in \mathbb{B}^n$, $c_{in} \in \mathbb{B}$, outputs $s \in \mathbb{B}^n$, $neg, ovf, c_{out} \in \mathbb{B}$.*

$$\begin{aligned}
Add_s &:= s[n-1,0],\\
Add_{c_{out}} &:= s[n],\\
Add_{neg} &:= s[n] \oplus a[n-1] \oplus b[n-1],\\
Add_{ovf} &:= Add_{neg} \oplus s[n-1].
\end{aligned}$$

We first show that only certain combinations of the bits $s[n]$, $a[n-1]$, and $b[n-1]$ occur:

Lemma 20 *For all $a, b \in \mathbb{B}^n$, $c_{in} \in \mathbb{B}$:*

$$\begin{aligned}
(i) \quad & s[n] \Longrightarrow a[n-1] \vee b[n-1],\\
(ii) \quad & s[n] \Longleftarrow a[n-1] \wedge b[n-1].
\end{aligned}$$

Proof. (i) The most significant bit of s is set, and $\langle s\rangle = \langle a\rangle + \langle b\rangle + c_{in}$. We therefore conclude by lemma 5

$$\langle a\rangle + \langle b\rangle + c_{in} \geq 2^n.$$

If $a[n-1] = b[n-1] = \mathbf{0}$, we have $\langle a\rangle < 2^{n-1}$ and $\langle b\rangle < 2^{n-1}$ by lemma 5. The sum of both is less than $2^n - 1$, which is a contradiction. Hence, we have

$$a[n-1] = \mathbf{1} \vee b[n-1] = \mathbf{1}.$$

(ii) By lemma 5, we have

$$\begin{aligned}
\langle a\rangle &\geq 2^{n-1},\\
\langle b\rangle &\geq 2^{n-1},\\
\langle a\rangle + \langle b\rangle + c_{in} = \langle s\rangle &\geq 2^n.
\end{aligned}$$

Again by lemma 5, we conclude

$$s[n] = \mathbf{1}.$$

$\square$

The sum $[a] + [b] + c_{in}$ cannot always be represented within the n bits of Add_s. The next lemma provides the representation of the sum in $n+1$ bits.

Lemma 21 *For all $a, b \in \mathbb{B}^n$, $c_{in} \in \mathbb{B}$:*

$$\left[Add_{neg} \circ s[n-1,0]\right] = [a] + [b] + c_{in}.$$

Proof. After applying lemma 11 on the left side and lemma 10 on the right, we have to show

$$-Add_{neg} \cdot 2^n + \langle s[n-1,0]\rangle = -a[n-1] \cdot 2^n + \langle a\rangle - b[n-1] \cdot 2^n + \langle b\rangle + c_{in}.$$

By lemma 3, $\langle s[n-1,0]\rangle = \langle s\rangle - 2^n \cdot s[n]$:

$$-Add_{neg} \cdot 2^n + \langle s\rangle - 2^n \cdot s[n] = -a[n-1] \cdot 2^n + \langle a\rangle - b[n-1] \cdot 2^n + \langle b\rangle + c_{in}.$$

$\langle s \rangle = \langle a \rangle + \langle b \rangle + c_{in}$ gives

$$-Add_{neg} \cdot 2^n - 2^n \cdot s[n] = -a[n-1] \cdot 2^n - b[n-1] \cdot 2^n.$$

We expand Add_{neg} and divide by -2^n:

$$(s[n] \oplus a[n-1] \oplus b[n-1]) + s[n] = a[n-1] + b[n-1].$$

By case analysis of all combinations of $s[n]$, $a[n-1]$, and $b[n-1]$ that are permissible by lemma 20, this equation holds. $\square$

A two's complement number represented by a bitvector of width $n+1$ can be represented in n bits if the two topmost bits are the same:

Lemma 22 *For all* $s \in \mathbb{B}^{n+1}$ *:*

$$[s] \in T_n \iff s[n] = s[n-1].$$

Proof. We expand the definition of T_n and apply lemma 3. We show

$$-2^{n-1} \leq (-2^n \cdot s[n] + \langle s[n-1,0] \rangle) \leq 2^{n-1} - 1 \iff s[n] = s[n-1].$$

By lemma 5, this is equivalent to the following, where $S := \langle s[n-1,0] \rangle$:

$$-2^{n-1} \leq \big(-2^n \cdot s[n] + S\big) \leq 2^{n-1} - 1 \iff s[n] = (S \geq 2^{n-1}).$$

This statement is verified by case analysis on $s[n]$ and the truth value of $(S \geq 2^{n-1})$. $\square$

We now use the above lemmas to show the correctness of Add:

Lemma 23 (ADDER CORRECT) *For all* $a, b \in \mathbb{B}^n$, $c_{in} \in \mathbb{B}$, *let* $binsum := \langle a \rangle + \langle b \rangle + c_{in}$, $intsum := [a] + [b] + c_{in}$, *and assume* $\langle add(a, b, c_{in}) \rangle = \langle a \rangle + \langle b \rangle + c_{in}$ *:*

$$\begin{array}{rl} (i) & \langle Add_{c_{out}} \circ Add_s \rangle = binsum, \\ (ii) & Add_{ovf} \iff intsum \notin T_n, \\ (iii) & intsum \in T_n \implies [Add_s] = intsum, \\ (iv) & Add_{neg} \iff intsum < 0. \end{array}$$

where $T_n = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$ *denotes the range of the* n*-bit two's complement numbers.*

Proof. (i) is trivial, since Add_{cout} and Add_s are re-concatenated after having been split in the definition of Add.

(ii) We prove

$$Add_{neg} \oplus s[n-1] \iff [a] + [b] + c_{in} \notin T_n.$$

By lemma 21, we have

$$Add_{neg} \oplus s[n-1] \iff \big[Add_{neg} \circ s[n-1,0]\big] \notin T_n.$$

The proof is finished by lemma 22, which yields

$$Add_{neg} \oplus s[n-1] \iff Add_{neg} \neq s[n-1].$$

(iii) We proceed to prove

$$[s[n-1,0]] = [a] + [b] + c_{in}.$$

We apply lemma 10:

$$\langle s[n-1,0]\rangle - 2^n \cdot s[n-1] = \langle a\rangle - 2^n \cdot a[n-1] + \langle b\rangle - 2^n \cdot b[n-1] + c_{in}.$$

$\langle s[n-1,0]\rangle = \langle s\rangle - 2^n \cdot s[n]$ by lemma 3:

$$\langle s\rangle - 2^n \cdot s[n] - 2^n \cdot s[n-1] = \langle a\rangle - 2^n \cdot a[n-1] + \langle b\rangle - 2^n \cdot b[n-1] + c_{in}.$$

Using $\langle s\rangle = \langle a\rangle + \langle b\rangle + c_{in}$ and simplifying gives

$$s[n] + s[n-1] = a[n-1] + b[n-1].$$

By assumption, we have $[a] + [b] + c_{in} \in T_n$, and hence, by part (ii), $Add_{neg} \oplus s[n-1] = \mathbf{0}$. By the definition of Add_{neg}, we equivalently have $s[n-1] = s[n] \oplus a[n-1] \oplus b[n-1]$. We therefore have to prove

$$s[n] + (s[n] \oplus a[n-1] \oplus b[n-1]) = a[n-1] + b[n-1].$$

This statement has already been proved at the end of the proof of lemma 21.

(iv) We show

$$Add_{neg} \iff [a] + [b] + c_{in} < 0.$$

With lemma 21, we have

$$Add_{neg} \iff \big[Add_{neg} \circ s[n-1,0]\big] < 0.$$

By using lemma 12, this is equal to

$$Add_{neg} \iff \big(Add_{neg} \circ s[n-1,0]\big)[n],$$

which is trivially true. □

3.6 Carry Save Adder

A carry save adder (also called 3/2-adder) sums up three input numbers a, b, c and yields *two* outputs s and t (figure 3.7). The sum of the inputs is equal to the sum of the outputs. This adder is used in wallace tree multipliers and the floating point rounding and multiplication units, and has a constant delay independent of the input width.

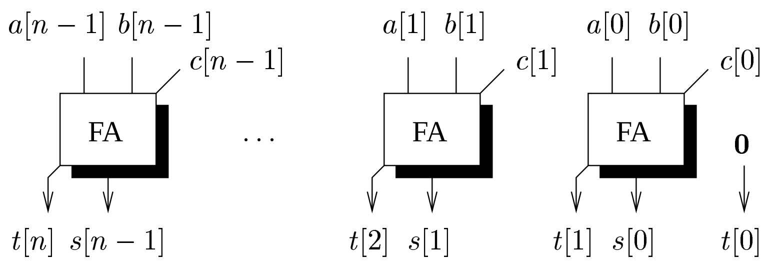

Figure 3.7: Carry save adder (3/2-adder)

Circuit 7 *Inputs* $a, b, c \in \mathbb{B}^n$*, outputs* $s \in \mathbb{B}^n$*,* $t \in \mathbb{B}^{n+1}$*. For* $n = 1$*, let* $F := \mathit{fulladder}(a[0], b[0], c[0])$*,*

$$\mathit{carry\_save}_{1,s} := F[0],$$
$$\mathit{carry\_save}_{1,t} := F[1] \circ \mathbf{0}.$$

For $n > 1$*, let* $(S, T) := \mathit{carry\_save}_{n-1,(s,t)}(a[n-2, 0], b[n-2, 0], c[n-2, 0])$*, and* $F := \mathit{fulladder}(a[n-1], b[n-1], c[n-1])$*,*

$$\mathit{carry\_save}_{n,s} := F[0] \circ S,$$
$$\mathit{carry\_save}_{n,t} := F[1] \circ T.$$

Lemma 24 (CARRY SAVE CORRECT) *For all* $a, b, c \in \mathbb{B}^n$*, let* $C = \mathit{carry\_save}(a, b, c)$ *:*

$$(i) \quad \langle a\rangle + \langle b\rangle + \langle c\rangle = \langle C_s\rangle + \langle C_t\rangle\,,$$
$$(ii) \quad [a] + [b] + [c] = [C_s] + [C_t].$$

Proof. (i) We induct on n. The induction base $n = 1$ is trivial by `(grind)`.

In the induction step for $n+1$, let $(S, T) := \mathit{carry\_save}_{n,(s,t)}(a[n-1, 0], b[n-1, 0], c[n-1, 0])$, and $F = \mathit{fulladder}(a[n], b[n], c[n])$. The induction claim is

$$\langle a\rangle + \langle b\rangle + \langle c\rangle = \langle F[0] \circ S\rangle + \langle F[1] \circ T\rangle\,.$$

By lemma 3, we have

$$\begin{gathered}(a[n] + b[n] + c[n]) \cdot 2^n + \\ \langle a[n-1, 0]\rangle + \langle b[n-1, 0]\rangle + \langle c[n-1, 0]\rangle = fa[0] \cdot 2^n + \langle S\rangle + fa[1] \cdot 2^{n+1} + \langle T\rangle\,.\end{gathered}$$

We can simplify this by the induction hypothesis:

$$(a[n] + b[n] + c[n]) \cdot 2^n = (F[0] + 2 \cdot F[1]) \cdot 2^n$$

This holds because of the correctness of the fulladder (lemma 16).

The proof for (ii) is analogous. □

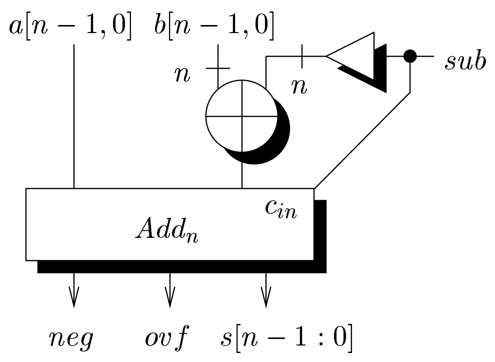

Figure 3.8: Arithmetic Unit add_sub

3.7 Arithmetic Unit, Subtraction

The arithmetic unit add_sub can add and subtract two's complement numbers. It is constructed from Add. The signal sub is used to select addition or subtraction. We define $\pm_x$ as

$$\pm_x := \begin{cases} + & \text{if } x = 0 \\ - & \text{if } x = 1. \end{cases}$$

Circuit 8 *Inputs $a, b \in \mathbb{B}^n$, $sub \in \mathbb{B}$, outputs $s \in \mathbb{B}^n$, $neg, ovf \in \mathbb{B}$.*[1]

$$add\_sub_n := Add_n(a, b \oplus sub^n, sub).$$

Lemma 25 (ADD SUB CORRECT) *For all $a, b \in \mathbb{B}^n$, $sub \in \mathbb{B}$, let $intsum = [a] \pm_{sub} [b]$:*

$$\begin{aligned} (i) \quad & add\_sub_{ovf} \iff (intsum \notin T_n), \\ (ii) \quad & intsum \in T_n \implies [add\_sub_s] = intsum, \\ (iii) \quad & add\_sub_{neg} \iff (intsum < 0). \end{aligned}$$

Proof. Using properties of mod arithmetic and the correctness of Add (lemma 23), it suffices to show

$$\big(\langle b \oplus sub^n \rangle + sub\big) \bmod 2^n = (-1)^{sub} \cdot \langle b \rangle \,.$$

This is trivial by lemmas 7 and 9. □

Subtraction. We define the circuit sub to be a circuit add_sub that always subtracts. Lemma 25 will also be used for the correctness of sub.

$$sub(a, b) := add\_sub(a, b, \mathbf{1}).$$

[1]The output $Add_{c_{out}}$ is not used.

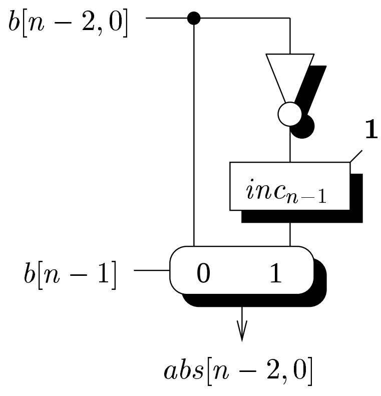

Figure 3.9: Absolute value computation

3.8 Absolute Value

The next circuit is used to compute the absolute value of a two's complement number (figure 3.9). If the input is positive, we pass the lower bits. Otherwise, we compute the two's complement inverse by inverting and incrementing the lower bits.

Circuit 9 *Let $n > 1$. Input $b \in \mathbb{B}^n$, output $abs \in \mathbb{B}^{n-1}$.*

$$abs(b) := \begin{cases} inc\big(\neg\, b[n-2,0], \mathbf{1}\big)[n-2,0] & \textit{if } b[n-1] \\ b[n-2,0] & \textit{else.} \end{cases}$$

Since the most negative two's complement number -2^{n-1} does not have a corresponding positive value that is representable in n bits, we exclude this number in the correctness criterion.

Lemma 26 (ABS CORRECT) *For all $b \in \mathbb{B}^n$, $b \neq \mathbf{1} \circ \mathbf{0}^{n-2}$:*

$$\langle abs(b)\rangle = \big|[b]\big|.$$

Proof. The case $b[n-1] = \mathbf{0}$ is trivial.

In the case $b[n-1] = \mathbf{1}$, $[b] < 0$ by lemma 12, and we have to prove

$$\big\langle inc\big(\neg\, b[n-2,0], 1\big)[n-2,0]\big\rangle = -[b].$$

After applying lemmas 7 and 13, we have

$$\langle inc(\neg\, b[n-2,0], 1)\rangle \bmod 2^{n-1} = [\neg\, b] + 1.$$

inc is correct by lemma 17, and lemma 11 yields:

$$(\langle \neg\, b[n-2,0]\rangle + 1) \bmod 2^{n-1} = \langle \neg\, b[n-2,0]\rangle + 1.$$

This is trivial for $\langle \neg\, b[n-2,0]\rangle + 1 < 2^{n-1}$. The only case where this does not hold is $b[n-2,0] = \mathbf{0}^{n-1}$, which is excluded by the lemma's prerequisites. □

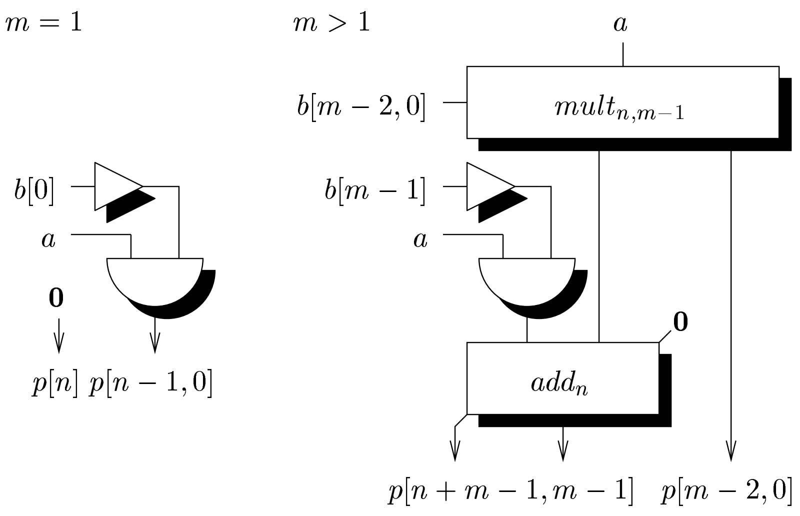

Figure 3.10: Linear multiplier $mult_{n,m}$

3.9 Multiplier

In this section, we construct a linear array multiplier. Multiplication with a single bit is implemented by a bitvector-AND. The single bit products are added up by a linear adder chain (figure 3.10).

Lemma 27 *For all $a \in \mathbb{B}^n$, $b \in \mathbb{B}$:*

$$\langle a \wedge b^n \rangle = \langle a \rangle \cdot b.$$

Proof. Trivial. □

Circuit 10 *Let $n, m \in \mathbb{N}^+$, inputs $a \in \mathbb{B}^n$, $b \in \mathbb{B}^m$, output $p \in \mathbb{B}^{n+m}$. For $m = 1$,*

$$mult\_lin_{n,1} := \mathbf{0} \circ (a \wedge b[0]^n).$$

For $m > 1$, let $M := mult\_lin_{n,m-1}(a, b[m-2,0])$,

$$mult\_lin_{n,m} := add_n\big((a \wedge b[m-1]^n), M[n+m-2, m-1], \mathbf{0}\big) \circ M[m-2,0].$$

Lemma 28 (MULTIPLIER CORRECT) *For all $n, m \in \mathbb{N}^+$, $a \in \mathbb{B}^n$, $b \in \mathbb{B}^m$:*

$$\langle mult\_lin(a,b) \rangle = \langle a \rangle \cdot \langle b \rangle .$$

Proof. We induct on m. The induction base $m = 1$ is trivial with lemma 27.

In the induction step $m+1$, let $M := mult\_lin_{n,m}(a, b[m-1,0])$. We show

$$\big\langle add\big((a \wedge b[m]^n), M[n+m-1, m], \mathbf{0}\big) \circ M[m-1,0]\big\rangle = \langle a \rangle \cdot \langle b \rangle .$$

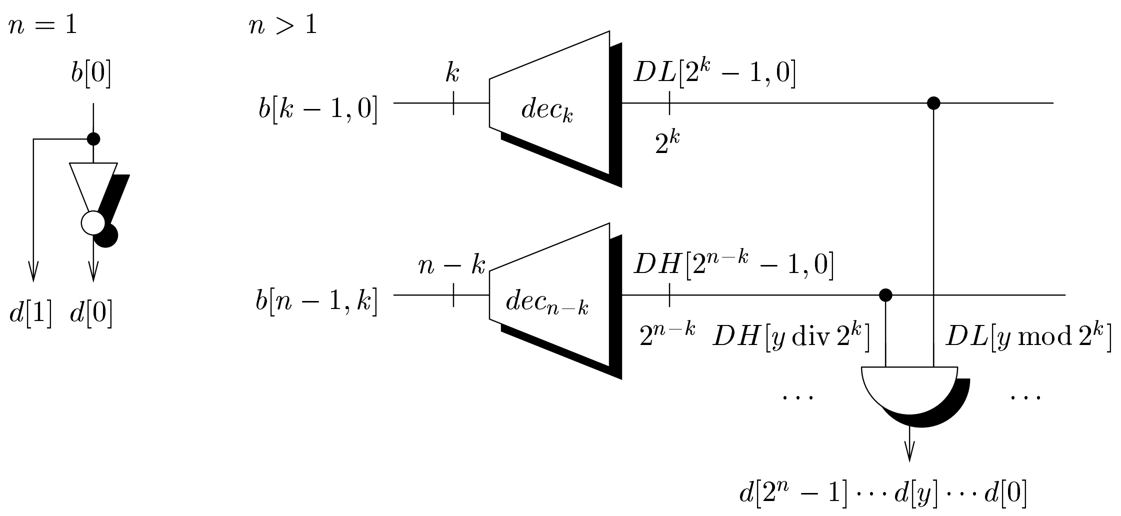

Figure 3.11: Decoder

We apply lemma 4:

$$\left\langle add\big((a \wedge b[m]^n), M[n+m-1,m], \mathbf{0}\big)\right\rangle \cdot 2^m + \langle M[m-1,0]\rangle = \langle a\rangle \cdot \langle b\rangle\,.$$

By lemma 18, add correctly sums up its inputs:

$$\big(\,\langle a \wedge b[m]^n\rangle + \langle M[n+m-1,m]\rangle\,\big) \cdot 2^m + \langle M[m-1,0]\rangle = \langle a\rangle \cdot \langle b\rangle\,.$$

Another application of lemma 4 yields

$$\langle a \wedge b[m]^n\rangle \cdot 2^m + \langle M[n+m-1,0]\rangle = \langle a\rangle \cdot \langle b\rangle\,.$$

By lemma 27 and the induction hypothesis, this is

$$\langle a\rangle \cdot b[m] \cdot 2^m + \langle a\rangle \cdot \langle b[m-1,0]\rangle = \langle a\rangle \cdot \langle b\rangle\,.$$

This is true by lemma 3. □

3.10 Decoder

A decoder (figure 3.11) converts numbers in binary format to unary format, that is, it calculates the function

$$\mathbb{B}^n \to \mathbb{B}^{2^n} : b \mapsto \mathbf{0}^{2^n-\langle b\rangle-1}\mathbf{1}\mathbf{0}^{\langle b\rangle}.$$

Circuit 11 *Input $b \in \mathbb{B}^n$, output $d \in \mathbb{B}^{2^n}$. For $n = 1$,*

$$dec_1 = b[0] \circ \neg\, b[0].$$

For $n > 1$, let $k := \lceil \frac{n}{2} \rceil$, $DH := dec_{n-k}(b[n-1,k])$, $DL := dec_k(b[k-1,0])$, and $y < 2^n$,

$$dec_n[y] := DH[y \operatorname{div} 2^k] \wedge DL[y \bmod 2^k].$$

Lemma 29 (DECODER CORRECT) *For all $b \in \mathbb{B}^n$, $y < 2^n$:*

$$dec(b)[y] \iff \langle b\rangle = y.$$

Proof. We induct on n. The induction base $n = 1$ is solved by `(grind)`.

The induction hypothesis for $n > 1$ is

$$\text{For all } 0 < j < n, b_j \in \mathbb{B}^j, y_j < 2^j : dec_j(b_j)[y_j] \iff \langle b_j \rangle = y_j.$$

Let $k = \lceil \frac{n}{2} \rceil$. We need induction hypotheses for two values of j:

1. $j = k$: This is permissible, since $0 < k = \lceil \frac{n}{2} \rceil < n$. In this case, $b_j = b[k-1, 0]$ and $y_j = y \bmod 2^k$.
$$dec_k(b[k-1,0])[y \bmod 2^k] \iff \langle b[k-1,0] \rangle = y \bmod 2^k.$$
2. $j = n - k$: Here, $n - \lceil \frac{n}{2} \rceil < n$ holds as well. $b_j = b[n-1, k]$ and $y_j = y \operatorname{div} 2^k$.
$$dec_{n-k}(b[n-1,k])[y \operatorname{div} 2^k] \iff \langle b[n-1,k] \rangle = y \operatorname{div} 2^k.$$
For the instantiation, we have to show that $y \operatorname{div} 2^k < 2^{n-k}$, i.e., that we stay within the width of the bitvector. We successively use: div is defined via $\lfloor \cdot \rfloor$, $\forall\ x : \lfloor x \rfloor \leq x$, and $y < 2^n$:
$$y \operatorname{div} 2^k = \left\lfloor \frac{y}{2^k} \right\rfloor \leq \frac{y}{2^k} < \frac{2^n}{2^k} = 2^{n-k}.$$

We start the proof by expanding the dec definition.

$$dec_{n-k}(b[n-1,k])[y \operatorname{div} 2^k] \wedge dec_k(b[k-1,0])[y \bmod 2^k] \iff \langle b \rangle = y.$$

After applying the induction hypotheses, it remains to show

$$\left(\langle b[n-1,k] \rangle = y \operatorname{div} 2^k \right) \wedge \left(\langle b[k-1,0] \rangle = y \bmod 2^k \right) \iff \langle b \rangle = y.$$

By properties of bitvector extraction (lemmas 7 and 8), we have

$$\left(\langle b \rangle \operatorname{div} 2^k = y \operatorname{div} 2^k \right) \wedge \left(\langle b \rangle \bmod 2^k = y \bmod 2^k \right) \iff \langle b \rangle = y.$$

The $\Leftarrow$ part is trivial. The $\Rightarrow$ part holds because the decompositions of y and $\langle b \rangle$ into div and mod components are unambiguous. This is proved by using the PVS mod and div lemmas. □

3.11 Halfdecoder

A halfdecoder (figure 3.12) converts binary numbers to *half unary* notation:

$$\mathbb{B}^n \to \mathbb{B}^{2^n} : b \mapsto \mathbf{0}^{2^n - \langle b \rangle} \mathbf{1}^{\langle b \rangle}.$$

Circuit 12 *Input $b \in \mathbb{B}^n$, output $d \in \mathbb{B}^{2^n}$. For $n = 1$,*

$$hdec_1 := \mathbf{0} \circ b[0].$$

For $n > 1$, let $HD = hdec_{n-1}(b[n-2,0])$,

$$hdec_n := (b[n-1]^n \wedge HD) \circ (b[n-1]^n \vee HD).$$

Lemma 30 (HALFDECODER CORRECT) *For all $b \in \mathbb{B}^n, y < 2^n$:*

$$hdec(b)[y] \iff y < \langle b \rangle.$$

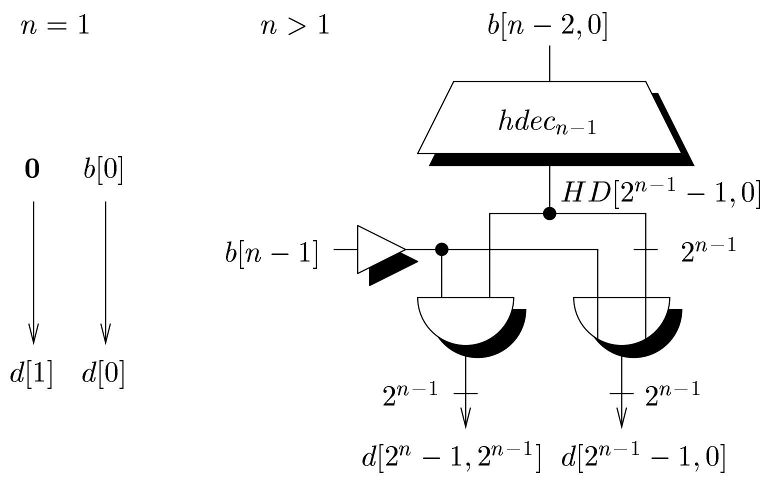

Figure 3.12: Halfdecoder

Proof. Again, the base of the induction on n can be proved by `(grind)`. We prove the statement for $n+1$ with an induction hypothesis for n.

In the case $y < 2^n$, we have to show

$$b[n] \vee hdec_n(b[n-1,0])[y] \iff y < \langle b \rangle \; .$$

By the induction hypothesis, this is

$$b[n] \vee \big(y < \langle b[n-1,0] \rangle\big) \iff y < \langle b \rangle \; .$$

The $\Leftarrow$ part is trivial for $b[n] = \mathbf{1}$.
For $\Rightarrow$ and $b[n] = \mathbf{1}$, we have $2^n \leq \langle b \rangle$ by lemma 5, and therefore, $y < 2^n \leq \langle b \rangle$.
In the case $b[n] = \mathbf{0}$, we have $\langle b[n-1,0] \rangle = \langle b \rangle$.

In the case $y \geq 2^n$, we show

$$b[n] \wedge hdec_n(b[n-1,0])[y-2^n] \iff y < \langle b \rangle \; .$$

We apply the induction hypothesis:

$$b[n] \wedge \big(y - 2^n < \langle b[n-1,0] \rangle\big) \iff y < \langle b \rangle \; .$$

For $b[n] = \mathbf{1}$, we have $\langle b[n-1,0] \rangle = \langle b \rangle - 2^n$ by lemma 3.
In the other case $b[n] = \mathbf{0}$, the $\Rightarrow$ part is trivial by propositional logic reasoning.
For $\Leftarrow$ and $b[n] = \mathbf{0}$, we show that $y < \langle b \rangle$ is false. This holds by lemma 5: $\langle b \rangle < 2^n \leq y$. $\square$

3.12 Encoder

The encoder computes the inverse of the decoder's function: for unary number inputs, it outputs the binary encoding (figure 3.13). The output is not defined for inputs that are not in unary form. The encoder implementation and correctness proof are due to Sven Beyer. We describe the implementation here because it is part of the VAMP basic circuits library. The proof is omitted here.

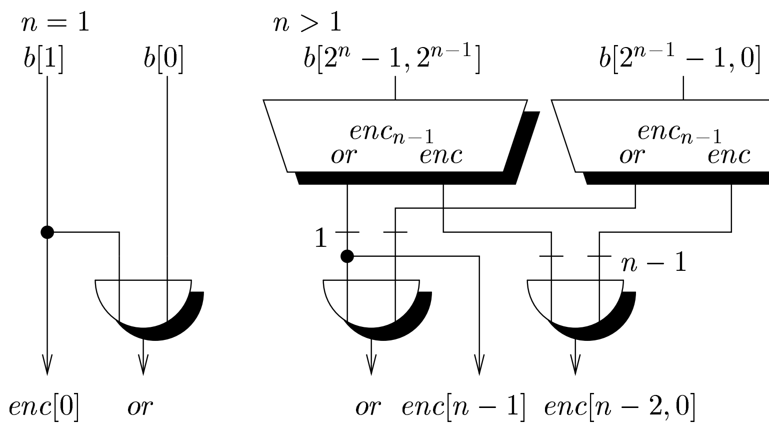

Figure 3.13: Encoder

Circuit 13 *Input $b \in \mathbb{B}^{2^n}$, outputs $or \in \mathbb{B}$, $enc \in \mathbb{B}^n$. For $n = 1$,*

$$\begin{aligned} enc_{1,or} &:= b[1] \vee b[0], \\ enc_{1,enc} &:= b[1]. \end{aligned}$$

For $n > 1$, let $EH := enc_{n-1}(b[2^n-1, 2^{n-1}])$, $EL := enc_{n-1}(b[2^{n-1}-1, 0])$, and $0 \le i \le n-2$,

$$\begin{aligned} enc_{n,or} &:= EH_{or} \vee EL_{or}, \\ enc_{n,enc}[i] &:= EH_{enc}[i] \vee EL_{enc}[i], \\ enc_{n,enc}[n-1] &:= EH_{or}. \end{aligned}$$

Lemma 31 (ENCODER CORRECT) *For all $y < 2^n$:*

$$\left\langle enc_{enc}(\mathbf{0}^{2^n-y-1}\mathbf{1}\mathbf{0}^y)\right\rangle = y.$$

The signal enc_{or} is only used internally and therefore not included in the correctness criterion.

3.13 Leading Zero Counter

The leading zero counter counts the number of zeros at the beginning of a bit string (figure 3.14). This section is a revised version of section 3 from [BJK01].

Before we can prove the leading zero counter implementation correct, we need a formal notion of ‘leading zeros’. We define a function $lzero$ on bitvectors of length n:

$$lzero(b) := \max\left\{i \in \mathbb{N} \mid i = 0 \vee \left(i \le n \wedge b[n-1, n-i] = \mathbf{0}^i\right)\right\}.$$

We start with some lemmas on the $lzero$ function. All these lemmas are fairly obvious, but their proofs are technically complicated in PVS. We omit the proofs.

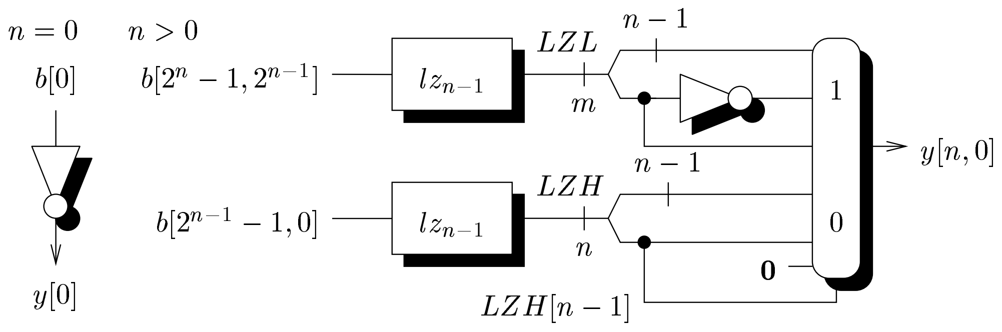

Figure 3.14: Leading zero counter

Lemma 32 *$lzero$ essentially depends on the position of the first $\mathbf{1}$-bit.*[2] *Let $1 \le i \le n - 2$.*

$$\begin{aligned}
&(1) \quad lzero(b) = 0 \iff b = \mathbf{1} \circ b[n-2, 0],\\
&(2) \quad lzero(b) = i \iff b = \mathbf{0}^i \circ \mathbf{1} \circ b[n-i-2, 0],\\
&(3) \quad lzero(b) = n-1 \iff b = \mathbf{0}^{n-1} \circ \mathbf{1},\\
&(4) \quad lzero(b) = n \iff b = \mathbf{0}^n.
\end{aligned}$$

Lemma 33 *$lzero$ is bounded by n :*

$$lzero(b) \le n.$$

Lemma 34 *Leading zero concatenation: For all $l \in \mathbb{N}^+$:*

$$lzero(\mathbf{0}^l \circ b) = l + lzero(b).$$

Circuit 14 *Let $n \in \mathbb{N}$. Input $b \in \mathbb{B}^{2^n}$, output $y \in \mathbb{B}^{n+1}$. For $n = 0$,*

$$lz_0 := \neg\, b[0].$$

For $n > 0$, let $LZH = lz_{n-1}(b[2^n - 1, 2^{n-1}])$ and $LZL = lz_{n-1}(b[2^{n-1} - 1, 0])$,

$$lz_n := \begin{cases} LZL[n-1] \circ \neg\, LZL[n-1] \circ LZL[n-2, 0] & \textit{if } LZH[n-1] \\ \mathbf{0} \circ LZH & \textit{else.} \end{cases}$$

The implementation is correct if it counts the number of leading zeros for all inputs correctly:

Lemma 35 (LZERO CORRECT) *For all $n \in \mathbb{N}_0, b \in \mathbb{B}^{2^n}$:*

$$\langle lz(b) \rangle = lzero(b).$$

[2]The case split is necessary to avoid bitvectors of zero length.

Proof. The proof is by induction on n. The induction base $n = 0$ is easily proved by using lemmas 32.4 and 33.

In the induction step n, let $b_H := b[2^n - 1, 2^{n-1}]$ and $b_L := b[2^{n-1} - 1, 0]$. We first look at the case $LZH[n-1] = \mathbf{1}$. By lemma 5, it follows that

$$2^{n-1} \leq \langle lz_{n-1}(b_H)\rangle \, .$$

The induction hypothesis and lemma 33 yield

$$2^{n-1} \leq lzero(b_H) \leq 2^{n-1},$$

which implies equality. Lemma 32.4 leads to

$$b_H = \mathbf{0}^{2^{n-1}} .$$

By lemma 34, and the induction hypothesis we have

$$\begin{aligned} lzero(b) &= lzero(\mathbf{0}^{2^{n-1}} \circ b_L) \\ &= 2^{n-1} + lzero(b_L) \\ &= 2^{n-1} + \langle lz_{n-1}(b_L)\rangle \, . \end{aligned}$$

With lemma 14, the output y of the multiplexer satisfies

$$lzero(b) = 2^{n-1} + \langle lz_{n-1}(b_L)\rangle = \langle y \rangle \, .$$

When $LZH[n-1] = \mathbf{0}$, by the induction hypothesis, the multiplexer output y satisfies

$$\langle y \rangle = \langle LZH \rangle = lzero(b_L).$$

By lemma 5, we have

$$lzero(b_L) = \langle LZH \rangle < 2^{n-1}.$$

We finish the proof with lemma 32.2, which yields

$$lzero(b_L \circ b_H) = lzero(b).$$

□

3.14 Cyclic Left Shifter

A cyclic left shifter shifts its input to the left. The bits that are shifted out on the left side are filled in on the right side. The circuit is defined for bitvector widths that are powers of two (figure 3.15).

Let $m \in \mathbb{N}^+$, $n := 2^m$ in this section. The formal definition of a cyclic left shift of a bitvector $a \in \mathbb{B}^n$ shifted $i < n$ bits is

$$clshift_n(a, i) := \begin{cases} a[n-i-1, 0] \circ a[n-1, n-i] & \text{if } i > 0 \\ a & \text{else.} \end{cases}$$

Lemma 36 *For all $a \in \mathbb{B}^n$, $b, c \in \mathbb{N}$, $b + c < n$:*

$$clshift_n(a, b+c) = clshift_n\big(clshift_n(a, b), c\big).$$

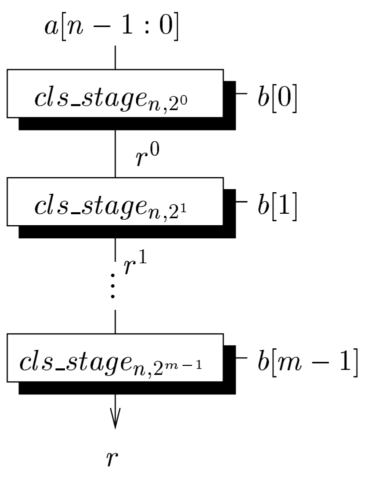

Figure 3.15: Cyclic left shifter cls

Proof. This lemma is resolved by expansion of the bitvector concatenation and extraction definitions, and applying the PVS decision procedures. □

We first define a single shifter stage, consisting of a multiplexer. Depending on a bit b, the stage either passes a unshifted, or shifts a by a fixed amount i to the left.

Circuit 15 *Let* $0 < i < n$. *Inputs* $a \in \mathbb{B}^n$, $b \in \mathbb{B}$, *output* $r \in \mathbb{B}^n$,

$$cls\_stage_{n,i} := \begin{cases} a[n-i-1,0] \circ a[n-1,n-i] & \textit{if } b \\ a & \textit{else.} \end{cases}$$

Lemma 37 *For all* $a \in \mathbb{B}^n$, $0 < i < n$, $b \in \mathbb{B}$ *:*

$$cls\_stage_{n,i}(a,b) = clshift_n(a, i \cdot b).$$

Proof. Trivial, since the $clshift$ and cls_stage definitions only differ marginally. The reader should note the subtle difference between the function $clshift$ that shifts a bitvector by an arbitrary amount, and the cls_stage hardware that shifts by an arbitrary, but *fixed* amount. □

The recursive shifter construction consists of a stack of stages that shift by powers of two (figure 3.15).

Circuit 16 *Let* $s < m$, *inputs* $a \in \mathbb{B}^n$, $b \in \mathbb{B}^m$, *output* $r \in \mathbb{B}^n$.

$$\begin{aligned} cls\_rec_{m,0} &:= cls\_stage_{n,1}(a, b[0]), \\ cls\_rec_{m,s} &:= cls\_stage_{n,2^s}\big(cls\_rec_{m,s-1}(a,b), b[s]\big). \end{aligned}$$

To prove the correctness of the cls_rec circuit, we show that the following invariant holds for each of the m stages of the shifter:

Lemma 38 (CLS REC CORRECT) *For all* $s < m$, $a \in \mathbb{B}^n$, $b \in \mathbb{B}^m$*:*

$$cls\_rec_{m,s}(a,b) = clshift_n(a, \langle b[s,0] \rangle).$$

Proof. We induct on s. The induction base $s = 0$ is a direct consequence of lemma 37.

In the induction step $s + 1$, the induction hypothesis is

$$cls\_rec_{m,s}(a, b) = clshift_n(a, \langle b[s, 0]\rangle).$$

We start by expanding the recursive definition of cls_rec:

$$cls\_stage_{m,2^{s+1}}\big(cls\_rec_{m,s}(a, b), b[s + 1]\big) = clshift_n(a, \langle b[s + 1, 0]\rangle).$$

By using the induction hypothesis, we have

$$cls\_stage_{m,2^{s+1}}\big(clshift_n(a, \langle b[s, 0]\rangle), b[s + 1]\big) = clshift_n(a, \langle b[s + 1, 0]\rangle).$$

By lemma 37 we have

$$clshift_n\big(clshift_n(a, \langle b[s, 0]\rangle), 2^{s+1} \cdot b[s + 1]\big) = clshift_n(a, \langle b[s + 1, 0]\rangle).$$

Applying lemma 36 yields

$$clshift_n(a, \langle b[s, 0]\rangle + 2^{s+1} \cdot b[s + 1]) = clshift_n(a, \langle b[s + 1, 0]\rangle).$$

This is true by lemma 3. □

Finally, we define the cyclic left shifter by an initial call to the recursive definition (figure 3.15):

Circuit 17 *Inputs* $a \in \mathbb{B}^n$, $b \in \mathbb{B}^m$, *output* $r \in \mathbb{B}^n$.

$$cls_m(a, b) := cls\_rec_{m,m-1}(a, b).$$

Lemma 39 (CLS CORRECT) *For all* $m \in \mathbb{N}^+$, $n := 2^m$, $a \in \mathbb{B}^n$, $b \in \mathbb{B}^m$ *:*

$$cls_m = clshift_n(a, \langle b\rangle).$$

Proof. A trivial application of lemma 38, where $s = m - 1$. □

3.15 Logical Left Shifter

In contrast to the cyclic left shifter, the logical left shifter pads the right part of the bitvector with zeros. We proceed in the same way as in the previous section: the shifter is a stack of stages that shift the bitvector by powers of two (analogous to figure 3.15). The difference is in the definition of the stages which do not shift cyclicly. Unlike the cyclic left shifter, the logical shifters are defined for arbitrary bitvector width.

Let $n \in \mathbb{N}^+$ and $logN := \lceil \log n \rceil$ in this section. The definition of the logical right shift of $a \in \mathbb{B}^n$ by i bits is

$$left\_shift_n(a, i) := \begin{cases} a & \text{if } i = 0 \\ a[n - i - 1, 0] \circ \mathbf{0}^i & \text{if } i < n \\ \mathbf{0}^n & \text{else.} \end{cases}$$

Circuit 18 *Let* $0 < i < n$. *Inputs* $a \in \mathbb{B}^n$, $b \in \mathbb{B}$, *output* $r \in \mathbb{B}^n$.

$$lls\_stage_{n,i} := \begin{cases} a[N-i-1,0] \circ \mathbf{0}^i & \textit{if } b \\ a & \textit{else.} \end{cases}$$

Circuit 19 *Let* $s < logN$. *Inputs* $a \in \mathbb{B}^n$, $b \in \mathbb{B}^{logN}$, *output* $r \in \mathbb{B}^n$.

$$lls\_rec_{n,0} := lls\_stage_{n,1}(a, b[0]).$$
$$lls\_rec_{n,s} := lls\_stage_{n,2^s}\big(lls\_rec_{n,s-1}(a,b), b[s]\big).$$

Lemma 40 (LLS REC CORRECT) *For all* $s < logN$, $a \in \mathbb{B}^n$, $b \in \mathbb{B}^{logN}$*:*

$$lls\_rec_{n,s}(a,b) = left\_shift_n(a, \langle b[s,0]\rangle).$$

Proof. The proof is analogous to the proof of lemma 38; we induct on s and expand the recursive definition of lls_rec. The PVS decision procedures then complete the proof. □

Circuit 20 *Inputs* $a \in \mathbb{B}^n$, $b \in \mathbb{B}^{logN}$, *output* $r \in \mathbb{B}^n$.

$$lls(a,b) := lls\_rec_{n,logN-1}(a,b).$$

Lemma 41 (LLS CORRECT) *For all* $a \in \mathbb{B}^n$, $b \in \mathbb{B}^{logN}$ *:*

$$lls(a,b) = left\_shift_n(a, \langle b\rangle).$$

Proof. Trivial application of the previous lemma 40 with $s = logN - 1$. □

3.16 Logical Right Shifter

The logical right shifter is symmetric to the logical left shifter. We therefore only state the specification and the correctness lemma without proof. Again, let $n \in \mathbb{N}^+$ and $logN := \lceil \log n \rceil$.

$$right\_shift_n(a,i) := \begin{cases} a & \text{if } i = 0 \\ \mathbf{0}^i \circ a[n-1,i] & \text{if } i < n \\ \mathbf{0}^n & \text{else.} \end{cases}$$

Circuit 21 *Let* $0 < i < n$. *Inputs* $a \in \mathbb{B}^n$, $b \in \mathbb{B}$, *output* $r \in \mathbb{B}^n$.

$$lrs\_stage_{n,i} := \begin{cases} \mathbf{0}^i \circ a[N-1,i] & \textit{if } b \\ a & \textit{else.} \end{cases}$$

Circuit 22 *Let* $s < logN$. *Inputs* $a \in \mathbb{B}^n$, $b \in \mathbb{B}^{logN}$, *output* $r \in \mathbb{B}^n$.

$$lrs\_rec_{n,0} := lrs\_stage(a, b[0], 1),$$
$$lrs\_rec_{n,s} := lrs\_stage_{n,2^s}\big(lrs\_rec_{n,s-1}(a,b), b[s]\big).$$
$$lrs := lrs\_rec_{n,logN-1}(a,b).$$

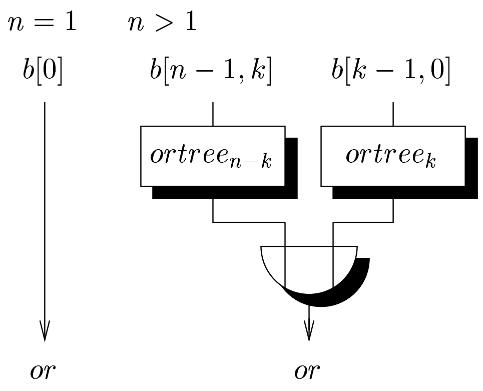

Figure 3.16: Or tree

Lemma 42 (LRS CORRECT) *For all $a \in \mathbb{B}^n$, $b \in \mathbb{B}^{logN}$:*

$$lrs(a,b) = right\_shift_n(a, \langle b \rangle).$$

3.17 Or Tree

An or tree is a tree of OR gates (figure 3.16). It is used for testing whether a bitvector consists only of zeros.

Circuit 23 *Input $b \in \mathbb{B}^n$, output $or \in \mathbb{B}$.*

$$ortree_1 := b[0].$$

Let $k = \lfloor \frac{n}{2} \rfloor$.

$$ortree_n := ortree_{n-k}(b[n-1,k]) \vee ortree_k(b[k-1,0]).$$

Lemma 43 (OR TREE CORRECT) *For all $b \in \mathbb{B}^n$:*

$$ortree(b) \iff b \neq \mathbf{0}^n.$$

Proof. We induct on n. The case $n = 0$ is resolved by `(grind)`.

In the induction step n, we show

$$ortree_{n-k}(b[n-1,k]) \vee ortree_k(b[k-1,0]) \iff b \neq \mathbf{0}^n.$$

The proof is finished by using the induction hypotheses:

$$b[n-1,k] \neq \mathbf{0}^{n-k} \vee b[k-1,0] \neq \mathbf{0}^k \iff b \neq \mathbf{0}^n.$$

□

Chapter 4

IEEE Floating Point Arithmetic

To formally verify the correctness of the VAMP FPU and its components, we need a formal notion of "correctness", i.e., a formalization of the IEEE standard [Ins85] the FPU shall obey. In this chapter, we sketch the formalization of the IEEE standard used in the VAMP verification project. The formalization is primarily based on [Min95, EP97, MP00]. The formalization of the IEEE standard has been implemented and formally verified in PVS by Christian Jacobi [JK00, BJ01, Jac01a, Jac01b]. The IEEE library is described here without proofs. This chapter is a revised version of section 2 from [BJ01].

4.1 Factorings

We abstract IEEE numbers as defined in the standard to *factorings*. A factoring is a triple (s, e, f) with sign bit $s \in \{\mathbf{0}, \mathbf{1}\}$, exponent $e \in \mathbb{Z}$, and significand $f \in \mathbb{R}_{\geq 0}$. Note that exponent range and significand precision are unbounded. The value of a factoring is

$$[\![s, e, f]\!] := (-1)^s \cdot 2^e \cdot f.$$

The standard introduces an exponent width N, from which constants $e_{min} := -2^{N-1} + 2$ and $e_{max} := 2^{N-1} - 1$ are derived. These constants are used to bound the exponent range.

We call a factoring (s, e, f) *normal* if $e \geq e_{min}$ and $1 \leq f < 2$. A factoring is called *denormal* if $e = e_{min}$ and $0 \leq f < 1$. We call a factoring an *IEEE factoring* if it is either normal or denormal.

Lemma 44 *Each $x \in \mathbb{R}_{\neq 0}$, has a unique factoring $(\hat{s}, \hat{e}, \hat{f})$ with $1 \leq \hat{f} < 2$ and $[\![\hat{s}, \hat{e}, \hat{f}]\!] = x$. Each $x \in \mathbb{R}_{\neq 0}$ has a unique IEEE factoring (s, e, f) with $[\![s, e, f]\!] = x$. Zero has two IEEE factorings $(0, e_{min}, 0)$ and $(1, e_{min}, 0)$, called $+0$ and -0, respectively.*

Let $\hat{\eta}$ and η be the functions that map (non-zero) reals x to their corresponding factorings $(\hat{s}, \hat{e}, \hat{f})$ and (s, e, f), respectively. We define $\eta(0) := (0, e_{min}, 0)$.

Lemma 45 *Let $x \in \mathbb{R}$ with $x \neq 0$ in the context of $\hat{\eta}$. It holds:*[1]

$$\hat{\eta}_e(x) = \lfloor \log_2 |x| \rfloor, \qquad \eta_e(x) = \begin{cases} \lfloor \log_2 |x| \rfloor & \textit{if } x \neq 0 \textit{ and } \lfloor \log_2 |x| \rfloor \geq e_{min} \\ e_{min} & \textit{else,} \end{cases}$$

$$\hat{\eta}_f(x) = |x| \cdot 2^{-\hat{\eta}_e(x)}, \qquad \eta_f(x) = |x| \cdot 2^{-\eta_e(x)}.$$

Let P be the significand precision defined in the standard. A significand f is called *representable* if f has at most $P-1$ digits behind the binary point, i.e., if $2^{P-1} \cdot f \in \mathbb{N}$. We call an IEEE factoring (s, e, f) *semi-representable* if f is representable. We call an IEEE factoring *representable* if it is semi-representable and $e \leq e_{max}$ holds. We call a real x (semi-)representable if $\eta(x)$ is (semi-)representable.

Representable numbers exactly correspond to the representable numbers defined in the standard. Common values for (N, P) are $(8, 24)$ and $(11, 53)$, called single and double precision, respectively. The standard defines an encoding of single and double precision IEEE factorings into bit strings of length 32 and 64, respectively. In this chapter, factorings are triples of numbers. In chapter 5, we introduce factorings that are triples of bitvectors.

4.2 Rounding

We proceed with the definition of the rounding function. The IEEE standard defines four rounding modes: round to nearest, up, down, and to zero. We define a function $r_{int}(\,\cdot\,, \mathcal{M})$ for each rounding mode $\mathcal{M} \in \{near, up, down, zero\}$, which rounds reals x to integers [Min95]:

$$\begin{aligned} r_{int}(x, near) &:= \begin{cases} \lfloor x \rfloor & \text{if } x - \lfloor x \rfloor < \lceil x \rceil - x \\ \lceil x \rceil & \text{if } x - \lfloor x \rfloor > \lceil x \rceil - x \\ x & \text{if } \lfloor x \rfloor = \lceil x \rceil \\ 2 \cdot \lfloor \lceil x \rceil / 2 \rfloor & \text{else,} \end{cases} \\ r_{int}(x, up) &:= \lceil x \rceil, \\ r_{int}(x, down) &:= \lfloor x \rfloor, \\ r_{int}(x, zero) &:= sign(x) \cdot \lfloor |x| \rfloor. \end{aligned}$$

By scaling by 2^{P-1}, reals are rounded to rationals with $P-1$ fractional bits:

$$r_{rat}(x, \mathcal{M}) := 2^{-(P-1)} \cdot r_{int}(x \cdot 2^{P-1}, \mathcal{M}).$$

Further scaling with 2^e, $e := \eta_e(x)$, yields the IEEE rounding function:

$$rd(x, \mathcal{M}) := 2^e \cdot r_{rat}(x \cdot 2^{-e}, \mathcal{M}).$$

It is not obvious that this definition conforms with the IEEE standard. The conformance is proved in [Jac01a].

The rounding of reals x can be decomposed into three steps: η-computation, significand rounding, and post-normalization [MP00, Jac01b]. The benefit of this decomposition is that it simplifies the design and verification of the rounder (see theorem 3) [BJ01].

[1] $\eta_e(x)$ denotes the e-component of the factoring $\eta(x) = (s, e, f)$; analogous for other components and $\hat{\eta}$.

First, the η-computation step computes the IEEE factoring $\eta(x)$, where x is the number to be rounded. The significand round then rounds the significand computed in the η-computation to $P-1$ digits behind the binary point. This is formalized in the function $sigrd$:

$$sigrd(X,\mathcal{M}) := \left| r_{rat}\big((-1)^s \cdot f, \mathcal{M}\big)\right|,$$

where $X=(s,e,f)$ is an arbitrary IEEE factoring, and $\mathcal{M} \in \{near, up, down, zero\}$ is a rounding mode.

In case the significand round returns 0 or 2, the factoring has to be post-normalized: if the significand round returns 2, the exponent is incremented, and the significand is forced to 1; if the significand round returns 0, the sign bit is forced to 0 (in order to yield $\eta(0)$). The post-normalization is defined as follows:

$$postnorm(X,\mathcal{M}) = \begin{cases} (s,e,sigrd(X,\mathcal{M})) & \text{if } 0 < sigrd(X,\mathcal{M}) < 2 \\ (s,e+1,1) & \text{if } sigrd(X,\mathcal{M}) = 2 \\ (0,e_{min},0) & \text{if } sigrd(X,\mathcal{M}) = 0. \end{cases}$$

Theorem 3 (DECOMPOSITION THEOREM) *For $x \in \mathbb{R}$ and rounding mode $\mathcal{M} \in \{near, up, down, zero\}$:*

$$postnorm\big(\eta(x),\mathcal{M}\big) = \eta\big(rd(x,\mathcal{M})\big).$$

4.3 α-Equivalence

We now define the concept of α-equivalence and α-representatives [EP97, MP00]. As we will see in theorem 5, this concept is a very concise way to speak about sticky-bit computations.

Let α be an integer. Two reals x and y are said to be *α-equivalent*, if

$$x \equiv_\alpha y \quad :\Longleftrightarrow \quad x=y \ \vee\ \big(\exists\ q\in\mathbb{Z} \colon q\cdot 2^\alpha < x,y < (q+1)\cdot 2^\alpha\big),$$

i.e., if both x and y lie in the same open interval between two consecutive integral multiples of 2^α. Clearly, if such an q exists, it must be

$$q_\alpha(x) := \lfloor x \cdot 2^{-\alpha} \rfloor.$$

The *α-representative* of x is defined as

$$[x]_\alpha := \begin{cases} x & \text{if } x = q_\alpha(x)\cdot 2^\alpha \\ \big(q_\alpha(x)+\frac{1}{2}\big)\cdot 2^\alpha & \text{else,} \end{cases}$$

i.e., if x is an integral multiple of 2^α, the representative of x is x itself, and otherwise the midpoint of the interval between the surrounding multiples of 2^α.

Lemma 46 *Let $x,y\in\mathbb{R}$, and $\alpha,k\in\mathbb{Z}$.*

1. *$\equiv_\alpha$ is an equivalence relation,*
2. *$x \equiv_\alpha [x]_\alpha$,*

3. $x \equiv_\alpha y \iff [x]_\alpha = [y]_\alpha$, *(representative equivalence)*
4. $x \equiv_\alpha y \iff -x \equiv_\alpha -y$, *and* $[-x]_\alpha = -[x]_\alpha$, *(negative value)*
5. $x \equiv_\alpha y \iff 2^k \cdot x \equiv_{\alpha+k} 2^k \cdot y$, *and* $[2^k \cdot x]_{\alpha+k} = 2^k \cdot [x]_\alpha$, *(scaling)*
6. $x \equiv_\alpha y \iff x + k \cdot 2^\alpha \equiv_\alpha y + k \cdot 2^\alpha$, *(translation)*
7. $x \equiv_\alpha y \implies x \equiv_{\alpha+k} y$ *if* $k \geq 0$, *(coarsening)*
8. $x = 0 \iff x \equiv_\alpha 0 \iff [x]_\alpha = 0$, *(zero value)*
9. $0 < x < 2^\alpha \implies [x]_\alpha = 2^{\alpha-1}$. *(small value)*

The following theorem describes the computation of IEEE factorings of representatives:

Theorem 4 *Let $x \in \mathbb{R}$, let $(s, e, f) := \eta(x)$ be the corresponding IEEE factoring, and let $p \geq 0$ be an integer. The IEEE factoring of $[x]_{e-p}$ can be computed by computing the representative $[f]_{-p}$ of f:*

$$\eta([x]_{e-p}) = (s, e, [f]_{-p}).$$

Next, we show that the representative of f can be computed by a *sticky bit computation*. Let $f \geq 0$ be a real in binary format $f_k, \ldots, f_0, \ldots, f_{-l} \in \{0,1\}^{k+l+1}$ such that $\langle f \rangle := \sum_{i=-l}^{k} f_i \cdot 2^i$. Let p be an integer, $k \geq p > -l$. The p-sticky-bit of f is the logical OR of all bits $f_{p-1}, \ldots, f_{-l}$:

$$sticky_p(f) := \bigvee_{i=-l}^{p-1} f_i.$$

Theorem 5 *The representative $[f]_p$ of f can be computed by replacing the less significant bits by the sticky bit:*

$$[f]_p = \langle f[k,p] \rangle + 2^{p-1} \cdot sticky_p(f).$$

Theorems 4 and 5 together allow a very efficient computation of representatives (respectively their IEEE factorings) by or-ing the less significant bits in an or tree, and replacing them by the sticky bit. This technique is well known [Gol96], but introducing the formalism with α-representatives allows for a very concise argumentation about sticky computations.

The valuable property of α-representatives is that rounding x and its representative $[x]_{e-P}$ yields the same result:

Theorem 6 *Let $x \in \mathbb{R}$, $e := \eta_e(x)$, and $\mathcal{M}$ be a rounding mode. It holds*

$$rd(x, \mathcal{M}) = rd([x]_{e-P}, \mathcal{M}).$$

As a consequence, the significand round can be performed on the representative $[f]_{-P}$ of f:

$$sigrd\big((s, e, f), \mathcal{M}\big) = sigrd\big((s, e, [f]_{-P}), \mathcal{M}\big).$$

Theorem 7 *By lemma 46.7, theorem 6 also holds for any $\alpha \leq e - P$:*

$$rd(x, \mathcal{M}) = rd([x]_\alpha, \mathcal{M}).$$

4.4 Exceptions

The IEEE standard defines five exceptions: invalid operation (INV), division by zero (DIVZ), overflow (OVF), underflow (UNF), and inexact result (INX). Our formalization of these exceptions is taken literally from [MP00], as the implementation in the actual hardware is. As a consequence of theorem 6, the exceptions can be detected by considering only the representative of the exact result.

In case of underflow or overflow with the respective trap handler enabled, the standard mandates scaling the result into the representable range, and passing the scaled result to the trap handler. This is called *wrapped exponent*. The scale factor is defined to be 2^A with $A := 3 \cdot 2^{N-2}$.

4.5 Correctness of the FPU

The standard requests that every floating point operation shall return a result obtained as if one first computed the exact result with infinite precision, and then rounded this exact result. We therefore call the FPU correct, if for each operation $\circ \in \{+, -, \times, \div\}$ on all representable numbers x and y, the FPU returns the IEEE bit string encoding of the factoring

$$\eta\big(rd(x \circ y, \mathcal{M})\big).$$

Chapter 5

The Floating Point Adder

In this chapter, we prove the correctness of the floating point adder from the VAMP FPU. A summary of this chapter is part of [BJ01]. The adder is given two factorings a and b and a flag sub. For $sub = \mathbf{0}$, it computes the sum $a + b$, and for $sub = \mathbf{1}$, the difference $a - b$.

5.1 Adder Correctness

Bitvector representation. The VAMP FPU handles single and double precision operands. Since single precision operands are embedded into double precision bitvectors by the unpacker, we only use double precision within the adder. We therefore fix the precision constants $(N, P) = (11, 53)$ in this chapter.

We define the *bitvector factoring type*

$$\mathbb{I}_p := (\mathbb{B}, \mathbb{B}^{11}, \mathbb{B}^p),$$

with the abbreviation $\mathbb{I} := \mathbb{I}_{53}$. In section 4.1, we defined $[\![\cdot]\!]$ on numbers. For $(s, e, f) \in \mathbb{I}_p$, we define

$$[\![s, e, f]\!]_i := [\![s, [e], \langle f\rangle \cdot 2^{i-p}]\!],$$

i.e., e is interpreted as a two's complement number, and f as a binary fraction with i bits in front of the binary point. For convenience, we will omit the index 1:

$$[\![s, e, f]\!] := [\![s, e, f]\!]_1.$$

In section 4.1, we defined (semi-)representable IEEE factorings. We will also apply these notions to bitvector factorings, e.g., $(s, e, f) \in \mathbb{I}$ is called an IEEE factoring if $(s, [e], \langle f\rangle \cdot 2^{i-p})$ is an IEEE factoring (where i is clear from the context).

Correctness criterion. As in chapter 3, we define $\pm_x$ for $x \in \mathbb{B}$ as

$$\pm_x := \begin{cases} + & \text{if } x = \mathbf{0} \\ - & \text{if } x = \mathbf{1}. \end{cases}$$

The VAMP unpacker passes the IEEE factorings a and $b \in \mathbb{I}$ to the adder [BJ01]. Let the exact, infinitely precise result of the operation to be performed be

$$S := [\![a]\!] \pm_{sub} [\![b]\!].$$

The adder computes an approximation factoring s of the exact result S.

The adder passes its output s to the floating point rounder. From s, the rounder computes the rounded result $rd(S, \mathcal{M})$ according to the IEEE standard (cf. section 4.2). To meet the rounder input specifications, we have to prove that the output s of the adder is *close enough* to S. In terms of α-equivalence (see theorem 6), that is

$$S \equiv_{e-P} [\![s]\!],$$

where $e := \eta_e(S)$. Using $\hat{e} := \hat{\eta}_e(S)$ and theorem 7, we will prove the adder correctness criterion

$$S \equiv_{\hat{e}-P} [\![s]\!].$$

By lemma 45, we have $\hat{e} \leq e$, and hence, the prerequisites of theorem 7 are satisfied.

The output s may be an arbitrary factoring, as the VAMP rounder is capable of rounding s correctly even if s is not an IEEE factoring. The rounder specification imposes two additional restrictions: the rounder input exponent must satisfy $e \leq e_{max}$ in case of a denormal input significand, and the value of the input factoring must lie in the range that can be scaled into the range of representable numbers using the wrapped exponent from section 4.4. (See [Jac01b] for details.)

Special cases. For a zero sum S, $\hat{e} = \hat{\eta}_e(S)$ is not defined. We will therefore require

$$S \neq 0$$

as a prerequisite to the adder correctness criterion proved here. In the VAMP FPU, the case $S = 0$ is detected by the unpacker. Other special cases that are handled by the unpacker are operations on NaN and $\pm\infty$. We therefore only treat numeric operands (represented by IEEE factorings). The unpacker implementation is described in detail in [Jac01b].

5.2 Addition Algorithm

We are given the input factorings $a := (s_a, e_a, f_a)$ and $b := (s_b, e_b, f_b)$, and a flag sub. We want to compute $[\![a]\!] \pm_{sub} [\![b]\!]$, represented by $s := (s_s, e_s, f_s)$.

We will implement the floating point addition using the following algorithm:

1. In case of a subtraction, flip the sign bit of b: $s'_b := s_b \oplus sub$,
2. the larger exponent of e_a and e_b is the result's exponent e_s,
3. assume $e_a \geq e_b$, otherwise exchange a and b.
4. align the significand f_b to f_a by shifting it $\delta := |e_a - e_b|$ to the right: $f^*_b := f_b \cdot 2^{-\delta}$,
5. let $sx := s_a \oplus s'_b$,
6. let $sum := f_a \pm_{sx} f^*_b$,
7. the results significand is $f_s := |sum|$,
8. the result's sign is $s_s := s_a \oplus (sum < 0)$.

This addition algorithm for floating point addition is well known [Gol96].

The alignment shift would require shifters of size $\approx 2^{N+1}$, which is impractical. We therefore approximate the shifted significand f_b^* by its $-(P+1)$-representative

$$f_b' := [2^{-\delta} \cdot f_b]_{-(P+1)}.$$

5.3 Correctness of the Addition Algorithm

To justify using the algorithm from section 5.2, we prove the following theorem, which will be used as the core of the adder hardware correctness proof.

Theorem 8 *For all representable IEEE factorings $a := (s_a, e_a, f_a)$ and $b := (s_b, e_b, f_b)$ where $e_a \geq e_b$, let $S := [\![a]\!] + [\![b]\!] \neq 0$, $\hat{e} := \hat{\eta}_e(S)$, $\delta := e_a - e_b$, and $f_b' := [2^{-\delta} \cdot f_b]_{-(P+1)}$:*

$$S \equiv_{\hat{e}-P} 2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f_b'\big).$$

Proof. S can be rewritten as

$$\begin{aligned} S &= [\![s_a, e_a, f_a]\!] + [\![s_b, e_b, f_b]\!] \\ &= (-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_b} \cdot f_b \\ &= 2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b\big). \end{aligned}$$

It remains to prove

$$2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b\big) \equiv_{\hat{e}-P} 2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f_b'\big).$$

For $\delta \leq 2$, we claim

$$f_b' = [2^{-\delta} \cdot f_b]_{-(P+1)} = 2^{-\delta} \cdot f_b.$$

We use the premise that b is a representable IEEE factoring with precision P. It follows that $f_b \cdot 2^{P-1}$ is an integer. Since $\delta \leq 2$, $f_b \cdot 2^{-\delta} \cdot 2^{P+1}$ is also integer. Therefore, $2^{-\delta} \cdot f_b = [2^{-\delta} \cdot f_b]_{-(P+1)}$ by the definition of $[\cdot]_\alpha$, which proves the theorem for $\delta \leq 2$.

Now let $\delta \geq 3$. Starting from

$$[2^{-\delta} \cdot f_b]_{-(P+1)} = f_b',$$

we have by lemma 46.3

$$2^{-\delta} \cdot f_b \equiv_{-(P+1)} f_b'.$$

Applying lemma 46.4 yields

$$(-1)^{s_b} \cdot 2^{-\delta} \cdot f_b \equiv_{-(P+1)} (-1)^{s_b} \cdot f_b'.$$

Lemma 46.6 yields

$$(-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b \equiv_{-(P+1)} (-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f_b'.$$

Lemma 46.5 yields

$$2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b\big) \equiv_{e_a-(P+1)} 2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f'_b\big).$$

Lemma 47 below tells us that $\hat{e} - P \geq e_a - (P+1)$. Therefore, we can coarsen the α-equivalence to $\hat{e} - P$ using lemma 46.7:

$$2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b\big) \equiv_{\hat{e}-P} 2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f'_b\big).$$

□

It remains to prove lemma 47, from which we concluded $\hat{e} - P \geq e_a - (P+1)$ in the above proof.

Lemma 47 *For all representable IEEE factorings $a := (s_a, e_a, f_a)$ and $b := (s_b, e_b, f_b)$, where $\delta := e_a - e_b \geq 2$, let $S := [\![a]\!] + [\![b]\!] \neq 0$, $\hat{e} := \hat{\eta}_e(S)$:*

$$\hat{e} \geq e_a - 1.$$

Proof. By lemma 45 and the definitions of S and $[\![\cdot]\!]$, we have to show

$$\hat{e} = \hat{\eta}(S).e = \lfloor \log_2 |(-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_b} \cdot f_b| \rfloor \geq e_a - 1.$$

Since $e_a - 1$ is integer, this is equivalent to

$$\log_2 |(-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_b} \cdot f_b| \geq e_a - 1.$$

b is an IEEE factoring, i.e.

$$0 \leq f_b < 2,$$

$\delta \geq 2$ implies

$$0 < 2^{-\delta} \leq \frac{1}{4},$$

we conclude

$$2^{-\delta} \cdot f_b < \frac{1}{2}.$$

From the fact that b is an IEEE factoring, we know

$$e_b \geq e_{min},$$

and due to $\delta = e_a - e_b \geq 2$,

$$e_a = e_b + \delta \geq e_{min} + \delta > e_{min}.$$

This means that a is a normal IEEE factoring, hence

$$f_a \geq 1.$$

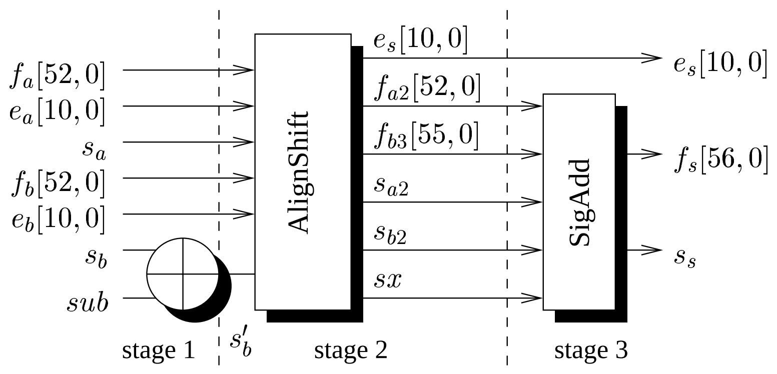

Figure 5.1: Top level schematics for the adder

By checking the four cases for the sign bits s_a and s_b, we conclude

$$|(-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b| \geq \frac{1}{2}.$$

Multiplying by 2^{e_a} yields

$$|(-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_b} \cdot f_b| \geq 2^{e_a-1}.$$

Taking logarithms on both sides finishes the proof. □

5.4 Adder Hardware

We will now show that the VAMP floating point adder (figure 5.1) correctly implements the addition algorithm in section 5.2.

Inputs to the adder are the representable IEEE factorings $a := (s_a, e_a, f_a)$, $b := (s_b, e_b, f_b) \in \mathbb{I}$ and the flag $sub \in \mathbb{B}$. Output is the factoring $s := (s_s, e_s, f_s) \in \mathbb{I}_{57}$.

The adder is divided into three stages. The first stage consists of a single XOR gate to compute s_b' (step 1 of the algorithm). The second stage exchanges a and b if necessary and computes the alignment shift, producing factorings a_2 and b_3 (steps 2 to 5). The third stage then adds/subtracts the significands (steps 6 to 8).

Differences from the adder given in [MP00] are:

- The special cases of NaN and infinite operands are not handled by the adder itself, but the unpacker. A zero result is also a special case. This evades the need for a forth adder stage for the sign computation. The zero tester is moved from the significand add stage to the unpacker [BJ01, Jac01b].

- The formalization does not use binary fractions, but natural numbers. This is due to the fact that the PVS bitvectors library does not support binary fractions. One could formalize binary fractions in PVS, but one would loose the benefits of the bitvector lemmas PVS provides.

5.5 Stage 1: Computing s'_b

Stage 1 implements the subtraction algorithm by inverting the sign bit s_b in case of a subtraction (see figure 5.1).

Circuit 24 *Inputs to stage 1 are the factorings $a := (s_a, e_a, f_a)$, $b := (s_b, e_b, f_b) \in \mathbb{I}$, and the subtract bit $sub \in \mathbb{B}$. Outputs are (s_a, e_a, f_a) and $b' := (s'_b, e_b, f_b) \in \mathbb{I}$. The sign bit of b' is computed as*

$$s'_b := sub \oplus s_b.$$

All other outputs pass stage 1 unmodified.

The following lemma states that the value b' is computed correctly:

Lemma 48 *For all factorings $(s_b, e_b, f_b) \in \mathbb{I}$ and subtract bits sub, the output $(s'_b, e_b, f_b) \in \mathbb{I}$ of stage 1 satisfies*

$$[\![s'_b, e_b, f_b]\!] = [\![s_b, e_b, f_b]\!] \cdot (-1)^{sub}.$$

Proof. The proof is trivial by applying the definitions of s'_b and $[\![\cdot]\!]$. □

Stage 2 requires that its inputs are IEEE factorings, this holds for the output of stage 1 if its inputs are IEEE factorings.

Lemma 49 *For all IEEE factorings (s_a, e_a, f_a), $(s_b, e_b, f_b) \in \mathbb{I}$:*

Output (s'_b, e_b, f_b) is an IEEE factoring.

Output (s_a, e_a, f_a) is an IEEE factoring by prerequisite.

Proof. Trivial by the definition of IEEE factorings. □

The correctness of the stage 1 output is asserted by theorem 9.

Theorem 9 (STAGE 1 CORRECT) *For all factorings (s_a, e_a, f_a) and $(s_b, e_b, f_b) \in \mathbb{I}$, and subtract bits $sub \in \mathbb{B}$:*

$$[\![s_a, e_a, f_a]\!] \pm_{sub} [\![s'_b, e_b, f_b]\!] \;=\; [\![s_a, e_a, f_a]\!] + [\![s'_b, e_b, f_b]\!].$$

Proof. Trivial by lemma 48. □

5.6 Stage 2: Alignment Shift

Stage 2 consists of the alignment shifter that computes the larger of both input exponents (step 2 from the algorithm in section 5.2), swaps a and b if necessary (step 3), and aligns the significands according to the difference of the exponents, thereby computing the representative f_b of the shifted significand (step 4).

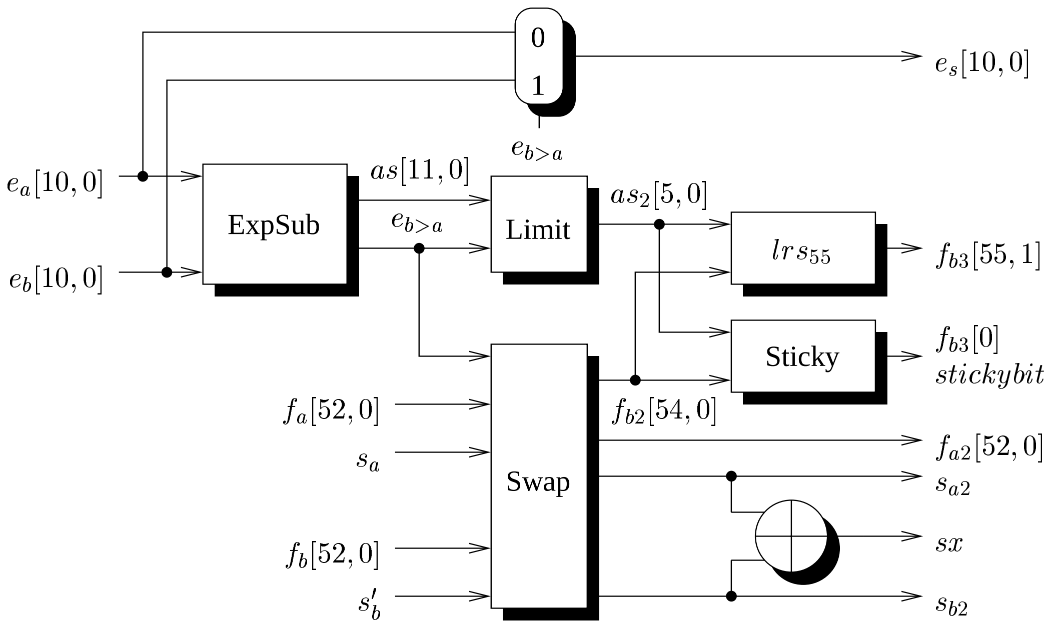

Figure 5.2: Top level alignment shifter schematics

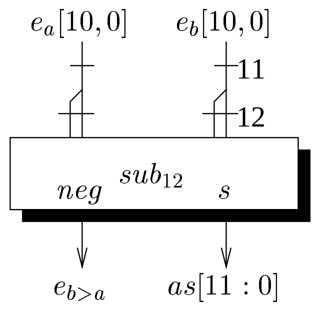

Figure 5.3: Exponent Subtract ExpSub

The alignment shifter itself is divided into several subcircuits (figure 5.2). Overall inputs are $a := (s_a, e_a, f_a)$ and $b' := (s'_b, e_b, f_b)$. Outputs are the (unrounded) result's exponent e_s and the possibly swapped sign bits and significands $(s_{a2}, e_s, f_{a2}) \in \mathbb{I}$ and $(s_{b2}, e_s, f_{b3}) \in \mathbb{I}_{56}$. A bit sx indicates whether we have to add or subtract the significands in the next stage.

5.6.1 Exponent Subtract

Circuit ExpSub computes the alignment shift distance $as := e_a - e_b$, and the flag $e_{b>a}$ indicating that the difference is negative (figure 5.3).

Circuit 25 *Inputs $e_a, e_b \in \mathbb{B}^{11}$, outputs $e_{b>a} \in \mathbb{B}$, $as \in \mathbb{B}^{12}$. Let $sext(b)$ denote the sign extension of b by one bit.*

$$as := sub_s(sext(e_a), sext(e_b)),$$
$$e_{b>a} := sub_{neg}(sext(e_a), sext(e_b)).$$

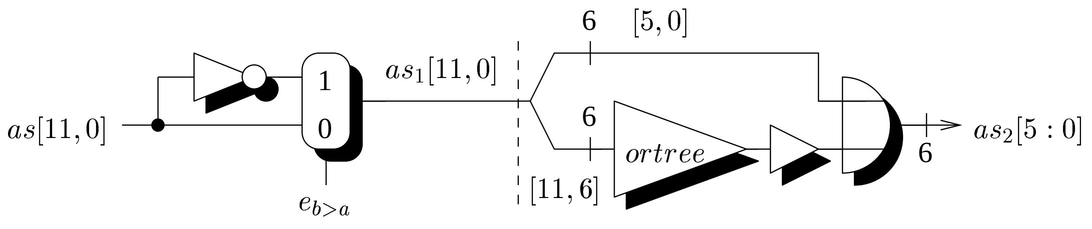

Figure 5.4: Circuit Limit

Lemma 50 (EXPSUB AS CORRECT) *For all* $e_a, e_b \in \mathbb{B}^{11}$*:*

$$as = [e_a] - [e_b].$$

Lemma 51 (EXPSUB EB-EA CORRECT) *For all* $e_a, e_b \in \mathbb{B}^{11}$*:*

$$e_{b>a} \iff [e_b] > [e_a] \iff [as] < 0.$$

Proof. Trivial application of the sub correctness lemma 25. □

5.6.2 Exponent Select

The greater of both exponents is selected by a multiplexer (see figure 5.2).

Circuit 26 *Inputs* $e_a, e_b \in \mathbb{B}^{11}$, $e_{b>a} \in \mathbb{B}$, *output* $e_s \in \mathbb{B}^{11}$.

$$e_s := \begin{cases} e_b & \textit{if } e_{b>a} \\ e_a & \textit{else.} \end{cases}$$

Lemma 52 (EXPONENT ES CORRECT) *For all* e_a, $e_b \in \mathbb{B}^{11}$, $e_{b>a} \in \mathbb{B}$, *let* $e_{b>a} = ([e_b] > [e_a])$*:*

$$e_s = \max(e_a, e_b).$$

Proof. Here, $\max$ is the bitvector two's complement maximum. This lemma assumes the correctness of the $e_{b>a}$ input, which was asserted by lemma 51. The proof is trivial. □

5.6.3 Circuit Limit

The alignment shift distance as is limited to reduce the size of the shifter lrs that aligns the significands (figure 5.4). The first part $limit\_approx$ of the $limit$ circuit computes an approximation as_1 of the absolute value of as. Instead of computing the two's complement, the circuit only negates as, saving an incrementer on the critical path of the floating point adder. The error introduced in the shift distance is compensated by the $swap$ circuit where f_{b2} is shifted by 1 in the erroneous case. The second part $limit\_limit$ limits as_1 to a maximum value of $B = 63$. as_2 is the output of the concatenation of both parts.

Approximating the Shift Distance

Circuit 27 *Inputs* $as \in \mathbb{B}^{12}$*,* $e_{b>a} \in \mathbb{B}$*, output* $as_1 \in \mathbb{B}^{12}$*.*

$$limit\_approx := \begin{cases} \neg\, as & \textit{if } e_{b>a} \\ as & \textit{else.} \end{cases}$$

Lemma 53 (LIMIT APPROX CORRECT) *For all* $as \in \mathbb{B}^{12}, e_{b>a} \in \mathbb{B}$*, let* $e_{b>a} = ([as] < 0)$*:*

$$\langle as_1 \rangle = |[as]| - e_{b>a}.$$

Proof. The assumption on $e_{b>a}$ holds by lemma 51.

In the case $e_{b>a} = \mathbf{1}$, we have to show that

$$\langle \neg\, as \rangle = |[as]| - 1$$

holds. In this case, $[as] < 0$. By lemma 9, this is equivalent to

$$2^{12} - 1 - \langle as \rangle = -[as] - 1.$$

Again because of $[as] < 0$, we conclude that $as[11]$ is set (lemma 12). We finish this case with lemma 10:

$$2^{12} - \langle as \rangle = -(\langle as \rangle - 1 \cdot 2^{12}).$$

In the other case $e_{b>a} = \mathbf{0}$, we have $[as] \geq 0$, and hence, $\neg\, as[11]$ by lemma 12.

$$\langle as \rangle = |[as]| - 0,$$
$$\langle as \rangle = [as].$$

The latter holds by lemma 10. □

Limiting the Shift Distance

Circuit 28 *Input* $as_1 \in \mathbb{B}^{12}$*, output* $as_2 \in \mathbb{B}^6$*.*

$$limit\_limit := as_1[5,0] \vee \big(ortree(as_1[11,6])\big)^6.$$

Lemma 54 (LIMIT LIMIT CORRECT) *Let* $B = 63$*. For all* $as_1 \in \mathbb{B}^{12}$*:*

$$\langle as_2 \rangle = \min(\langle as_1 \rangle, B).$$

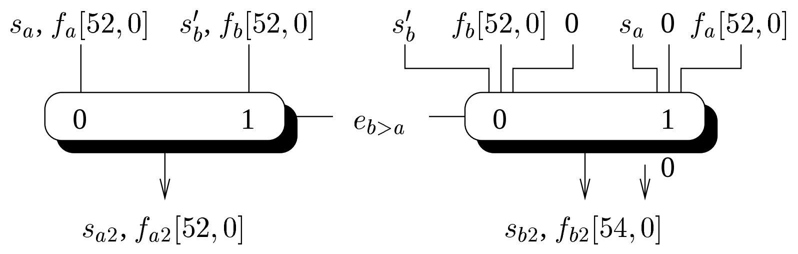

Figure 5.5: Significand and sign bit swapping

Proof. First, we assume $\langle as_1 \rangle > B$. By lemma 6, we have $as_1[11,6] \neq \mathbf{0}^6$, and by lemma 43, $ortree(as_1[11,6]) = \mathbf{1}$. Hence,

$$\langle as_2 \rangle = \langle \mathbf{1}^6 \rangle = B = \min(\langle as_1 \rangle, B).$$

Where $\langle as_1 \rangle \leq B$, $as_1[11,6] = \mathbf{0}^6$, and $ortree(as_1[11,6]) = \mathbf{0}$. We finish with

$$\langle as_2 \rangle = \langle as_1[5,0] \rangle = \langle as_1 \rangle = \min(\langle as_1 \rangle, B).$$

□

Finally, we combine the two circuits and lemmas:

Circuit 29 *Inputs $as \in \mathbb{B}^{12}$, $e_{b>a} \in \mathbb{B}$, output $as_2 \in \mathbb{B}^6$.*

$$limit := limit\_limit \circ limit\_approx.$$

Here, $\circ$ is function concatenation.

Lemma 55 (Limit Correct) *For all $as \in \mathbb{B}^{12}$, $e_{b>a} \in \mathbb{B}$, let $e_{b>a} = ([as] < 0)$:*

$$\langle as_2 \rangle = \min(|[as]| - e_{b>a}, B).$$

Proof. A direct consequence of lemmas 53 and 54. □

5.6.4 Significand Swapping

If exponent e_b is greater than e_a, signal $e_{b>a}$ indicates that the significands f_a and f_b, and the sign bits s_a and s_b have to be swapped (figure 5.5). This is realized by multiplexers. Because of the error in the shift amount introduced by the $limit$ circuit, f_{b2} is shifted one digit to the right here when $e_{b>a}$ is active ($\mathbf{0} \circ f_a$ vs. $f_b \circ \mathbf{0}$ in the f_{b2} definition).

Circuit 30 *Inputs $s_a, s'_b, e_{b>a} \in \mathbb{B}$, $f_a, f_b \in \mathbb{B}^{53}$, outputs $s_{a2}, s_{b2} \in \mathbb{B}$, $f_{a2} \in \mathbb{B}^{53}$, $f_{b2} \in \mathbb{B}^{55}$.*

$$s_{a2} := \begin{cases} s'_b & \textit{if } e_{b>a} \\ sa & \textit{else,} \end{cases} \qquad s_{b2} := \begin{cases} sa & \textit{if } e_{b>a} \\ s'_b & \textit{else,} \end{cases}$$

$$f_{a2} := \begin{cases} fb & \textit{if } e_{b>a} \\ fa & \textit{else,} \end{cases} \qquad f_{b2} := \left\{ \begin{array}{ll} \mathbf{0} \circ f_a & \textit{if } e_{b>a} \\ f_b \circ \mathbf{0} & \textit{else} \end{array} \right\} \circ \mathbf{0}.$$

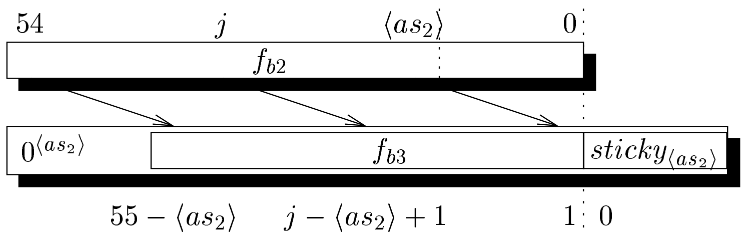
54
j
$\langle as_2 \rangle$
0
f_{b2}
$0^{\langle as_2 \rangle}$
f_{b3}
$sticky_{\langle as_2 \rangle}$
$55 - \langle as_2 \rangle$
$j - \langle as_2 \rangle + 1$
1
0

Figure 5.6: Shifting f_{b2} by $\langle as_2 \rangle$ bits to the right

Lemma 56 (SIGNIFICAND SWAP CORRECT) *For all* $f_a, f_b \in \mathbb{B}^{53}$, $e_{b>a} \in \mathbb{B}$*:*

$$s_{a2} = \begin{cases} s'_b & \textit{if } e_{b>a} \\ sa & \textit{else,} \end{cases} \qquad s_{b2} = \begin{cases} sa & \textit{if } e_{b>a} \\ s'_b & \textit{else,} \end{cases}$$

$$\langle f_{a2} \rangle = \begin{cases} \langle fb \rangle & \textit{if } e_{b>a} \\ \langle fa \rangle & \textit{else,} \end{cases} \qquad \langle f_{b2} \rangle = \begin{cases} 2 \cdot \langle f_a \rangle & \textit{if } e_{b>a} \\ 4 \cdot \langle f_b \rangle & \textit{else.} \end{cases}$$

Proof. The proof is trivial by lemma 4. □

5.6.5 Alignment Shift and Sticky Bit Computation

The alignment shifter shifts the significand f_{b2} to the right, collecting the shifted-out bits in the sticky bit (figure 5.6). We split this task into a logical right shifter lrs and the sticky bit computation $stickybit$ (figure 5.7).

The sticky bit is the logical OR of the lower as_2 bits of f_{b2}. The lower bits are selected by an AND mask driven by a halfdecoder.

Circuit 31 *Inputs* $f_{b2} \in \mathbb{B}^{53}$, $as_2 \in \mathbb{B}^6$, *output* $stickybit \in \mathbb{B}$.

$$stickybit := ortree(hdec(as_2)[54,0] \wedge f_{b2}).$$

This definition conforms with the definition of $sticky$ in section 4.3:

Lemma 57 *For all* $as_2 \in \mathbb{B}^6$, $0 < \langle as_2 \rangle < 55$, $f_{b2} \in \mathbb{B}^{55}$*:*

$$stickybit(as_2, f_{b2}) = sticky_{\langle as_2 \rangle}(f_{b2}).$$

Proof. After expanding $stickybit$ and $sticky$, and applying the correctness lemmas of $ortree$ (lemma 43) and $hdec$ (lemma 30), it remains to show

$$\left((\mathbf{0}^{2^6 - \langle as_2 \rangle} \circ \mathbf{1}^{\langle as_2 \rangle})[54,0] \wedge f_{b2} \right) \neq \mathbf{0}^{55} \iff f_{b2}[\langle as_2 \rangle - 1, 0] \neq \mathbf{0}^{\langle as_2 \rangle}.$$

This is straightforward in PVS by expanding the definitions of $\circ$ and $[\cdot,\cdot]$. □

We now show that the sticky bit is computed correctly for $\langle as_2 \rangle = 0$ and $\langle as_2 \rangle = 55$, and that all $\langle as_2 \rangle \geq 55$ are equivalent to the latter case.

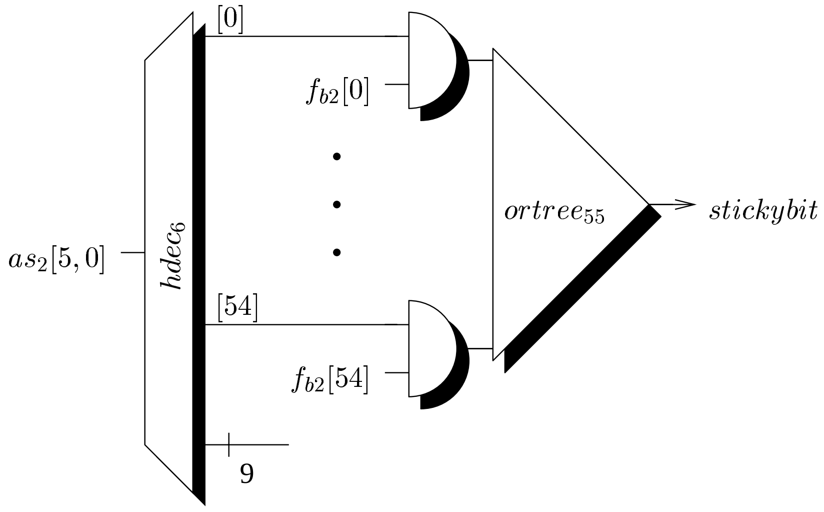

Figure 5.7: Sticky bit computation

Lemma 58 *Let B_{55} be the unique bitvector of width 6 with $\langle B_{55}\rangle = 55$. For $as_2 \in \mathbb{B}^6, f_{b2} \in \mathbb{B}^{55}$:*

1. $\langle as_2\rangle = 0 \implies stickybit(as_2, f_{b2}) = \mathbf{0}$,
2. $stickybit(B_{55}, f_{b2}) \iff f_{b2} \neq \mathbf{0}^{55}$,
3. $\langle as_2\rangle \geq 55 \implies stickybit(as_2, f_{b2}) = stickybit(B_{55}, f_{b2})$.

Proof. The proof is similar to the proof of lemma 57; after applying the correctness lemmas of $ortree$ and $hdec$, PVS is able to prove the statements by expanding $\circ$ and $[\cdot,\cdot]$. □

The representative f_{b3} of f_{b2} is computed by the bitvector concatenation of a logical right shifter and $stickybit$ (see figure 5.2):

Circuit 32 *Inputs $f_{b2} \in \mathbb{B}^{55}$, $as_2 \in \mathbb{B}^6$, output $f_{b3} \in \mathbb{B}^{56}$.*

$$f_{b3} := lrs(f_{b2}, as_2) \circ stickybit(as_2, f_{b2}).$$

Lemma 59 (STICKY SHIFT CORRECT) *For all $as_2 \in \mathbb{B}^6, f_{b2} \in \mathbb{B}^{55}$:*

$$\langle f_{b3}\rangle = [\langle f_{b2}\rangle \cdot 2^{-\langle as_2\rangle - 54}]_{-54} \cdot 2^{55}.$$

Proof. We split the proof in three cases:

1. $\langle as_2\rangle = 0$: In this case, lrs does not modify f_{b2}, and $stickybit = \mathbf{0}$ by lemma 58. We show

$$\langle f_{b2} \circ \mathbf{0}\rangle = [\langle f_{b2}\rangle \cdot 2^{-54}]_{-54} \cdot 2^{55}.$$

Since $\langle f_{b2}\rangle$ is integer, $[\langle f_{b2}\rangle \cdot 2^{-54}]_{-54} = \langle f_{b2}\rangle \cdot 2^{-54}$ by the definition of α-representatives. On the left side oft the equation, we use lemma 4.

$$\langle f_{b2}\rangle \cdot 2 = \langle f_{b2}\rangle \cdot 2^{-54} \cdot 2^{55}.$$

2. $0 < \langle as_2 \rangle < 55$: After applying lemma 57, we have

$$\left\langle lrs(f_{b2}, as_2) \circ sticky_{\langle as_2 \rangle}(f_{b2}) \right\rangle = [\langle f_{b2} \rangle \cdot 2^{-\langle as_2 \rangle - 54}]_{-54} \cdot 2^{55}.$$

   We apply the correctness of lrs (lemma 42):

$$\left\langle \mathbf{0}^{\langle as_2 \rangle} \circ f_{b2}[54, \langle as_2 \rangle] \circ sticky_{\langle as_2 \rangle}(f_{b2}) \right\rangle = [\langle f_{b2} \rangle \cdot 2^{-\langle as_2 \rangle - 54}]_{-54} \cdot 2^{55}.$$

   Lemma 4 yields

$$\left\langle f_{b2}[54, \langle as_2 \rangle] \circ sticky_{\langle as_2 \rangle}(f_{b2}) \right\rangle = [\langle f_{b2} \rangle \cdot 2^{-\langle as_2 \rangle - 54}]_{-54} \cdot 2^{55}.$$

   By lemma 46.5 we can scale the α-representative by $2^{-\langle as_2 \rangle - 54}$, yielding

$$\left\langle f_{b2}[54, \langle as_2 \rangle] \circ sticky_{\langle as_2 \rangle}(f_{b2}) \right\rangle = [\langle f_{b2} \rangle]_{\langle as_2 \rangle} \cdot 2^{1 - \langle as_2 \rangle}.$$

   This is true by theorem 5.

3. $\langle as_2 \rangle \geq 55$: Using lemma 58, we have to show

$$\left\langle lrs(f_{b2}, as_2) \circ (f_{b2} \neq \mathbf{0}^{55}) \right\rangle = [\langle f_{b2} \rangle \cdot 2^{-\langle as_2 \rangle - 54}]_{-54} \cdot 2^{55}.$$

   f_{b2} is shifted out completely by lrs (lemma 42):

$$\left\langle \mathbf{0}^{55} \circ (f_{b2} \neq \mathbf{0}^{55}) \right\rangle = [\langle f_{b2} \rangle \cdot 2^{-\langle as_2 \rangle - 54}]_{-54} \cdot 2^{55}.$$

   Lemma 4 yields

$$\left\langle f_{b2} \neq \mathbf{0}^{55} \right\rangle = [\langle f_{b2} \rangle \cdot 2^{-\langle as_2 \rangle - 54}]_{-54} \cdot 2^{55}.$$

   In case where $f_{b2} = \mathbf{0}^{55}$, this is trivial by lemma 46.8. For $f_{b2} \neq \mathbf{0}^{55}$, we use lemma 46.9,

$$1 = 2^{-55} \cdot 2^{55}.$$

   The application of lemma 46.9 is valid for $\langle f_{b2} \rangle \cdot 2^{-\langle as_2 \rangle - 54} < 2^{-54}$, which is true. □

5.6.6 Alignment Shifter Correctness

The alignment shifter consists of circuits 25 to 32. This is also the definition of $stage\_2$.

Circuit 33 *Inputs* (s_a, e_a, f_a), $(s'_b, e_b, f_b) \in \mathbb{I}$, *signals* $e_{b>a} \in B$, $as \in \mathbb{B}^{12}$, $as_2 \in \mathbb{B}^6$, $f_{b2} \in \mathbb{B}^{55}$, *outputs* $(s_{a2}, e_s, f_{a2}) \in \mathbb{I}$, $(s_{b2}, e_s, f_{b3}) \in \mathbb{I}_{56}$, $sx \in \mathbb{B}$. *Let the wires be connected as defined in circuits 25 to 32.*

We now can prove that the sum of the outputs of the alignment shifter is α-equivalent to the sum of the inputs.

Lemma 60 (ALIGN SHIFT CORRECT) *For all semi-representable IEEE factorings* $a := (s_a, e_a, f_a)$, $b' := (s'_b, e_b, f_b) \in \mathbb{I}$*: Let* $S := [\![a]\!] + [\![b']\!] \neq 0$, $\hat{e} := \hat{\eta}_e(S)$.

$$S \equiv_{\hat{e}-53} 2^{[e_s]} \cdot \left((-1)^{s_{a2}} \cdot \frac{\langle f_{a2} \rangle}{2^{52}} + (-1)^{s_{b2}} \cdot \frac{\langle f_{b3} \rangle}{2^{55}} \right).$$

Proof. We apply lemma 59 for f_{b3}:

$$S \equiv_{\hat{e}-53} 2^{[e_s]} \cdot \left((-1)^{s_{a2}} \cdot \frac{\langle f_{a2} \rangle}{2^{52}} + (-1)^{s_{b2}} \cdot [\langle f_{b2} \rangle \cdot 2^{-\langle as_2 \rangle - 54}]_{-54} \right).$$

We distinguish two cases, depending on the signal $e_{b>a}$:

1. $\neg\, e_{b>a}$: The significand swap is correct (lemma 56 for s_{a2}, f_{a2}, s_{b2}, and f_{b2}):

$$S \equiv_{\hat{e}-53} 2^{[e_s]} \cdot \left((-1)^{s_a} \cdot \frac{\langle f_a \rangle}{2^{52}} + (-1)^{s'_b} \cdot [4 \cdot \langle f_b \rangle \cdot 2^{-\langle as_2 \rangle - 54}]_{-54} \right).$$

Lemmas 52 (e_s) and 55 (as_2) yield

$$S \equiv_{\hat{e}-53} 2^{[e_a]} \cdot \left((-1)^{s_a} \cdot \frac{\langle f_a \rangle}{2^{52}} + (-1)^{s'_b} \cdot \left[4 \cdot \frac{\langle f_b \rangle}{2^{54}} \cdot 2^{-\min([as], B)} \right]_{-54} \right).$$

By lemma 50 (as), we conclude

$$S \equiv_{\hat{e}-53} 2^{[e_a]} \cdot \left((-1)^{s_a} \cdot \frac{\langle f_a \rangle}{2^{52}} + (-1)^{s'_b} \cdot \left[\frac{\langle f_b \rangle}{2^{52}} \cdot 2^{-\min([e_a]-[e_b], B)} \right]_{-54} \right).$$

By lemma 61 below, this is equivalent to

$$S \equiv_{\hat{e}-53} 2^{[e_a]} \cdot \left((-1)^{s_a} \cdot \frac{\langle f_a \rangle}{2^{52}} + (-1)^{s'_b} \cdot \left[\frac{\langle f_b \rangle}{2^{52}} \cdot 2^{-([e_a]-[e_b])} \right]_{-54} \right).$$

This is true by theorem 8.

2. $e_{b>a}$: The significand swap (lemma 56 for s_{a2}, f_{a2}, s_{b2}, and f_{b2}) yields

$$S \equiv_{\hat{e}-53} 2^{[e_s]} \cdot \left((-1)^{s_{b'}} \cdot \frac{\langle f_b \rangle}{2^{52}} + (-1)^{s_a} \cdot [2 \cdot \langle f_a \rangle \cdot 2^{-\langle as_2 \rangle - 54}]_{-54} \right).$$

We use lemmas 52 (e_s) and 55 (as_2):

$$S \equiv_{\hat{e}-53} 2^{[e_b]} \cdot \left((-1)^{s'_b} \cdot \frac{\langle f_b \rangle}{2^{52}} + (-1)^{s_a} \cdot \left[2 \cdot \frac{\langle f_a \rangle}{2^{54}} \cdot 2^{-\min(-[as]-1, B)} \right]_{-54} \right).$$

Lemma 50 (as) yields

$$S \equiv_{\hat{e}-53} 2^{[e_b]} \cdot \left((-1)^{s'_b} \cdot \frac{\langle f_b \rangle}{2^{52}} + (-1)^{s_a} \cdot \left[\frac{\langle f_a \rangle}{2^{53}} \cdot 2^{-\min\left(-([e_a]-[e_b])-1, B\right)} \right]_{-54} \right).$$

By lemma 62 below, we have

$$S \equiv_{\hat{e}-53} 2^{[e_b]} \cdot \left((-1)^{s'_b} \cdot \frac{\langle f_b \rangle}{2^{52}} + (-1)^{s_a} \cdot \left[\frac{\langle f_a \rangle}{2^{52}} \cdot 2^{-([e_b]-[e_a])} \right]_{-54} \right).$$

This is proved by interchanging a and b in theorem 8. □

Lemma 61 *For all semi-representable IEEE factorings $a := (s_a, e_a, f_a)$, $b' := (s'_b, e_b, f_b) \in \mathbb{I}$, where $e_a \geq e_b$, let $\delta := [e_a] - [e_b]$:*

$$\left[\frac{\langle f_b \rangle}{2^{52}} \cdot 2^{-\min(\delta, B)} \right]_{-54} = \left[\frac{\langle f_b \rangle}{2^{52}} \cdot 2^{-\delta} \right]_{-54}.$$

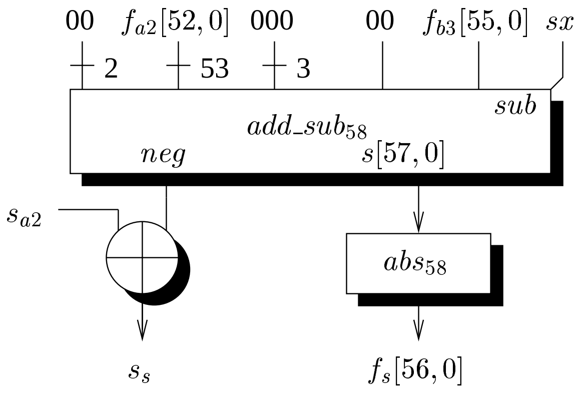

Figure 5.8: Significand addition SigAdd

Proof. By lemma 46.8, the claim is trivial for $f_b = 0$. It is also trivial for $\delta \leq B$, where $\min(\delta, B) = \delta$. For $\delta \geq B$, both sides of the equation are equal to 2^{-55} by lemma 46.9, and the lemma is proved. □

Lemma 62 *For all semi-representable IEEE factorings $a := (s_a, e_a, f_a)$, $b' := (s'_b, e_b, f_b) \in \mathbb{I}$, where $e_a < e_b$, let $\delta := [e_a] - [e_b]$:*

$$\left[\frac{\langle f_a\rangle}{2^{53}} \cdot 2^{-\min(-\delta-1,B)}\right]_{-54} = \left[\frac{\langle f_a\rangle}{2^{52}} \cdot 2^{-\delta}\right]_{-54}.$$

Proof. Analogous to the proof of lemma 61. □

Theorem 10 (STAGE 2 CORRECT) *For all semi-representable IEEE factorings $a := (s_a, e_a, f_a)$, $b' := (s'_b, e_b, f_b) \in \mathbb{I}$: Let $S := [\![a]\!] + [\![b']\!] \neq 0$, $\hat{e} := \hat{\eta}_e(S)$.*

$$S \equiv_{\hat{e}-53} [\![s_{a2}, e_s, f_{a2}]\!] + [\![s_{b2}, e_s, f_{b3}]\!].$$

Proof. Immediate consequence of lemma 60 and the definition of $[\![\cdot]\!]$. □

5.7 Stage 3: Significand Addition

The third stage adds the significands computed by stage 2. Depending on sx, we have to add or subtract the significands.

Circuit 34 *Inputs $(s_{a2}, \cdot, f_{a2}) \in \mathbb{I}$, $(s_{b2}, \cdot, f_{b3}) \in \mathbb{I}_{56}$, $sx \in \mathbb{B}$, output $(s_s, \cdot, f_s) \in \mathbb{I}_{57}$. Let $add\_sub := add\_sub(\mathbf{0}^2 \circ f_{a2} \circ \mathbf{0}^3, \mathbf{0}^2 \circ f_{b3}, sx)$.*

$$s_s := s_{a2} \oplus add\_sub_{neg},$$
$$f_s := abs(add\_sub_s).$$

Lemma 63 (SIG ADD CORRECT) *For all $(s_{a2}, \cdot, f_{a2}) \in \mathbb{I}$, $(s_{b2}, \cdot, f_{b3}) \in \mathbb{I}_{56}$, $sx \in \mathbb{B}$, $sx = s_{a2} \oplus s_{b2}$. Let $sum := [\![s_{a2}, 0, f_{a2}]\!] + [\![s_{b2}, 0, f_{b3}]\!]$.*

$$sum = [\![s_s, 0, f_s]\!]_2.$$

Proof. We show

$$sum = (-1)^{s_{a2} \oplus add\_sub_{neg}} \cdot \frac{\langle abs(add\_sub_s)\rangle}{2^{55}}.$$

Using the correctness of the abs circuit (lemma 26) yields

$$sum = (-1)^{s_{a2} \oplus add\_sub_{neg}} \cdot \frac{|[add\_sub_s]|}{2^{55}}.$$

In the case $s_{a2} = s_{b2} = \mathbf{0}$, we have $sx = \mathbf{0}$ by the assumption on sx. The outputs add_sub_s and add_sub_{neg} are correct by lemma 25. In this case,

$$sum = (-1)^{0 \oplus \left([\mathbf{0}^2 \circ f_{a2} \circ \mathbf{0}^3] + [\mathbf{0}^2 \circ f_{b3}] < 0\right)} \cdot \frac{\left|[\mathbf{0}^2 \circ f_{a2} \circ \mathbf{0}^3] + [\mathbf{0}^2 \circ f_{b3}]\right|}{2^{55}}.$$

We rewrite with lemmas 10 and 4:

$$\frac{\langle f_{a2}\rangle}{2^{52}} + \frac{\langle f_{b3}\rangle}{2^{55}} = (-1)^{(8 \cdot \langle f_{a2}\rangle + \langle f_{b3}\rangle < 0)} \cdot \frac{|8 \cdot \langle f_{a2}\rangle + \langle f_{b3}\rangle|}{2^{55}}.$$

The right side reduces to

$$\frac{\langle f_{a2}\rangle}{2^{52}} + \frac{\langle f_{b3}\rangle}{2^{55}} = \frac{8 \cdot \langle f_{a2}\rangle + \langle f_{b3}\rangle}{2^{55}}.$$

The other cases of s_{a2} and s_{b2} are resolved analogously.

It remains to show that the sum of the inputs is representable in the width of the add_sub output, and that the add_sub output is a valid input for abs. Both statements are proved by using the fact that the range of the inputs is limited by the leading zeros fed into add_sub. □

$Stage\_3$ consists of the significand addition circuit and an additional wire for passing e_s (see figure 5.1).

Theorem 11 (STAGE 3 CORRECT) *For all* $(s_{a2}, e_s, f_{a2}) \in \mathbb{I}$, $(s_{b2}, e_s, f_{b3}) \in \mathbb{I}_{56}$, $sx \in \mathbb{B}$, *where* $sx = s_{a2} \oplus s_{b2}$, *let* $sum := [\![s_{a2}, e_s, f_{a2}]\!] + [\![s_{b2}, e_s, f_{b3}]\!]$ *:*

$$sum = [\![s_s, e_s, f_s]\!]_2.$$

Proof. By multiplication with 2^{e_s} in lemma 63. □

5.8 Putting It All Together

We now combine the above defined stages.

Circuit 35 *Inputs* $(s_a, e_a, f_a), (s_b, e_b, f_b) \in \mathbb{I}$, $sub \in B$, *output* $(s_s, e_s, f_s) \in \mathbb{I}_{57}$.

$$fpadder := stage\_3 \circ stage\_2 \circ stage\_1.$$

Here, $\circ$ *is function concatenation, not bitvector concatenation.*

In section 5.1, we argued that the floating point adder is correct if its output is $(\hat{e}-P)$-equivalent to the correct result, where $P = 53$. This is proved in the next theorem. Since the significand adder output f_s has two bits in front of the binary point, we use $[\![\cdot]\!]_2$.

Theorem 12 (FP ADDER CORRECT) *For all IEEE factorings $a := (s_a, e_a, f_a)$, $b := (s_b, e_b, f_b) \in \mathbb{I}$ and subtract bits $sub \in \mathbb{B}$, let $S = [\![a]\!] \pm_{sub} [\![b]\!]$, and $\hat{e} = \hat{\eta}_e(S)$, where $S \neq 0$. Then the following equation holds for the output $(s_s, e_s, f_s) \in \mathbb{I}_{57}$ of fpadder:*

$$S \equiv_{\hat{e}-P} [\![s_s, e_s, f_s]\!]_2.$$

Proof. We start with

$$S = [\![s_a, e_a, f_a]\!] \pm_{sub} [\![s_b, e_b, f_b]\!].$$

Theorem 9 yields

$$S = [\![s_a, e_a, f_a]\!] + [\![s_b', e_b, f_b]\!].$$

By theorem 10, we have

$$S \equiv_{\hat{e}-P} [\![s_{a2}, e_s, f_{a2}]\!] + [\![s_{b2}, e_s, f_{b3}]\!].$$

By theorem 11, this is

$$S \equiv_{\hat{e}-P} [\![s_s, e_s, f_s]\!]_2.$$

Therefore, the VAMP floating point adder is correct. □

Theorem 12 is an excellent example on how the task of “putting it all together” can be handled in a precise way by application of the specifications of the modules—if the modules have a well defined behaviour.

5.9 Boundary Constraints

It remains to prove the additional rounder input requirements from section 5.1.

$$e_s \leq e_{max}$$

holds trivially, since e_s is chosen from $\{e_a, e_b\}$ and e_{max} is the maximum value in T_N. (We prove this even if the significand is not denormal.)

To show that we stay in the range that is scalable into the range of representable numbers by wrapped exponent (section 4.4), we show

$$2^{e_{min}-A} < |[\![s]\!]| < 2^{e_{max}+A}.$$

Proof. For the lower bound, we first notice that $[\![s]\!] \neq 0$, since $S \neq 0$ by assumption, $S \equiv_\alpha [\![s]\!]$ by theorem 12, and $[\![s]\!] = 0 \iff [\![s]\!] \equiv_\alpha 0$ by lemma 46.8. The smallest positive representable number is $2^{e_{min}-P+1}$. We finish this case by observing $A \gg P$, and hence,

$$2^{e_{min}-A} < 2^{e_{min}-P+1} \leq |[\![s]\!]|.$$

For the upper bound, we start with theorem 12:

$$[\![s]\!] \equiv_{\hat{e}-P} [\![a]\!] \pm_{sub} [\![b]\!].$$

This implies

$$|[\![s]\!]| \equiv_{\hat{e}-P} |[\![a]\!] \pm_{sub} [\![b]\!]|.$$

We conclude

$$|[\![s]\!]| < |[\![a]\!] \pm_{sub} [\![b]\!]| + 2^{\hat{e}-P}.$$

Using the triangle inequality, this is

$$|[\![s]\!]| < |[\![a]\!]| + |[\![b]\!]| + 2^{\hat{e}-P}.$$

a and b are bounded: $[\![a]\!], [\![b]\!] < 2^{e_{max}+1}$, therefore

$$|[\![s]\!]| < 2^{e_{max}+1} + 2^{e_{max}+1} + 2^{\hat{e}-P}.$$

The claim is proved with

$$|[\![s]\!]| < 2^{e_{max}+A}.$$

The VAMP floating point adder meets the VAMP rounder specification. $\square$

Chapter 6

Summary

6.1 The VAMP Project

Our group at Saarland University is verifying the correctness of the VAMP microprocessor. The VAMP is a RISC processor based on the DLX architecture [HP96, MP00]. The VAMP features a five stage pipeline, a Tomasulo scheduler, precise and nested interrupts, delayed branch, and a fully IEEE compliant floating point unit [JK00, Krö01, BJ01, Jac01b]. The correctness of the circuits is proved using the PVS theorem prover [OSR92]. The floating point adder that is formally verified in this thesis is part of the VAMP FPU.

The verified VAMP processor is being implemented on a Xilinx FPGA [BJKL01]. Our group is porting the *gcc* compiler and the *GNU* C library to the VAMP architecture to yield an environment suitable to evaluating the verified hardware.

All PVS sources—specification, implementation, and proof scripts—and the Verilog hardware descriptions are publically available at our web site [VAM].

To the best of our knowledge, this is the first time that a complete floating point unit has been formally verified on the gate level, and all proofs and designs have been published.

6.2 Bugs

‘Bugs’ are either differences between a specification and corresponding implementation, or flaws in the specification itself. There are several ways to find bugs: proof-reading specification and implementation, testing, proofs, or combinations of these. The purpose of formal verification is to find bugs that would otherwise go unnoticed. Formal verification gives—to some extend—the confidence that the implementation adheres to its specification.

Two problems remain, however. On the one hand, the consistency of the specification itself cannot be asserted entirely. The specification may be inconsistent, which will probably be found within the verification process because of inherent errors, or the specification may be consistent, but be something else than what we wanted to formalize. We can only hope that the provably correct theorem ‘the implementation is correct for the specification’ is the same as ‘the implementation is exactly what we intended to build’.

Furthermore, the verification tool—PVS in our case—could be unsound, meaning that it is possible to prove statements that are not true in the mathematical sense. There are some known flaws in PVS, and it is even possible to prove a theorem stating 'false' by exploiting these flaws. We tried not to use any faulty proof commands, although it is possible—though unlikely—that we accidentally got trapped by bugs in PVS that have not yet been discovered.

Bugs found. One major bug in the design of the floating point adder was found in the verification. In [MP00], the sign bit s_s (called s_{s1} in the book) is wrongly computed as

$$s_s := (s'_b \wedge neg) \vee (s_a \wedge \neg (s'_b \wedge neg)).$$

This error results from an erroneous entry in table 8.2 (page 370), where line 7 is marked 'impossible'. The proper entry is 'impossible' in line 8, and '0' in line 7. This change results in the correct equation for the sign bit:

$$s_s := s_a \oplus neg.$$

Other bugs where incorrect bitvector subscripts and exponents in powers of 2. Since these can be considered to be mere typos, we do not list them here.

No serious bugs in the PVS logic were found, i.e., bugs rendering PVS unsound. Several minor flaws in the PVS system were reported to SRI [PVS].

6.3 Related Work

Basic circuits. The verification of a simple adder and an arithmetic logic unit using PVS is reported in [CRSS94]. The PVS bitvectors library [BMS$^+$96] includes a verified carry chain adder. The verification of an adder using various verification systems is described in [SBE88]. In [CB96], Bryant verifies fixed size arithmetic circuits against a mathematical specification.

Given a reference design and assuming its correctness, it is state-of-the-art to automatically verify equivalence with a new design. There are several approaches to this, e.g., boolean equivalence checkers using BDDs or variations [Bry86, BC95, CFZ95]. In [JLMC97], Clarke et al. use function abstraction and BDDs for equivalence checking. In [Sta99], Stanion proves the equivalence of two fixed bit width multipliers.

IEEE standard. Other formalizations of the IEEE standard in theorem proving systems have been given by Miner [Min95] and Harrison [Har99].

Floating point hardware. Aagaard and Seger combine BDD based methods and theorem proving techniques to verify a floating point multiplier [AS95]. Chen and Bryant [CB98] use word-level SMV to verify a floating point adder. Exceptions and denormals are not handled in both verification projects.

Cornea-Hasegan describes the computation of division and square root by Newton-Raphson iteration in the Intel IA-64 architecture [CH98, CHN99]. The verification is done using Mathematica. O'Leary et al. report on the verification of the gate level design of Intel's FPU using a combination of model-checking and theorem proving [OZGS99]. Denormals and exceptions are not covered in the paper.

Moore et al. have verified the AMD K5 division algorithm [MLK98] with the theorem prover ACL2. Russinoff has verified the K5 square root algorithm as well as the Athlon multiplication, division, square root, and addition algorithms [Rus98, Rus99, Rus00].

6.4 Prospect

There are several ways in which to extend the work on the VAMP architecture:

- Modern FPUs support a vast variety of operations, such as square root, trigonometric functions, and compound operations as $\frac{1}{1-x^2}$. One could verify functional units for these and incorporate them into the VAMP FPU.
- Similarly, modern CPUs support more precision modes.
- Efficiency was not a goal of the VAMP project. One could improve both throughput and latency of the verified functional units.
- In an analogous way, the above points apply to the VAMP integer core, pipeline, and control.
- Hardware verification is difficult, and even reusing proofs for similar circuits and modules requires much effort. A better framework for handling (hardware) proofs would be beneficial.
- We have verified the PVS hardware specifications, and translated these to Verilog using a tool that has not been verified. One could formally verify the Verilog specifications.

Bibliography

[AS95] M. D. Aagaard and C.-J. H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *ICCAD*, pages 7–10. IEEE, November 1995.

[BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *32nd ACM/IEEE Design Automation Conference*, Pittsburgh, June 1995. Carnegie Mellon University.

[BJ01] Christoph Berg and Christian Jacobi. Formal verification of the VAMP floating point unit. To appear in *CHARME 2001*, 2001.

[BJK01] Christoph Berg, Christian Jacobi, and Daniel Kröning. Formal verification of a basic circuits library. In *IASTED International Conference on Applied Informatics*. ACTA Press, February 2001.

[BJKL01] Sven Beyer, Christian Jacobi, Daniel Kröning, and Dirk Leinenbach. Correct hardware by synthesis from PVS. Submitted to ICCD 2001, 2001.

[BMS$^+$96] Ricky W. Butler, Paul S. Miner, Mandayam K. Srivas, Dave A. Greve, and Steven P. Miller. A bitvectors library for PVS. Technical Report TM-110274, NASA Langley Research Center, 1996.

[Bry86] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CB96] Y. Chen and R. Bryant. ACV: An arithmetic circuit verifier. In *In Proc. of IEEE ICCD '96*, pages 361–365. IEEE, 1996.

[CB98] Y.-A. Chen and R. E. Bryant. Verification of floating point adders. In *CAV'98*, volume 1427 of *LNCS*, 1998.

[CFZ95] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In *ICCAD*, pages 159–163, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[CH98] Marius Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide, and remainder algorithms. *Intel Technology Journal*, Q2, 1998.

[CHN99] Marius Cornea-Hasegan and Bob Norin. IA-64 floating point operations and the IEEE standard for binary floating-point arithmetic. *Intel Technology Journal*, Q4, 1999.

[COR$^+$95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, 1995.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In *2nd International Conference on Theorem Provers in Circuit Design*, volume 901 of *LNCS*, pages 203–222. Springer, 1994.

[EP97] Guy Even and Wolfgang J. Paul. On the design of IEEE compliant floating point units. In *Proceedings of the 13th Symposium on Computer Arithmetic*. IEEE Computer Society Press, 1997.

[Gen35] G. Gentzen. Untersuchungen über das logische Schließen. In *Mathematische Zeitschrift*, volume 1, pages 176–210, 1935.

[Gol96] David Goldberg. Computer arithmetic, 1996. Appendix A in [HP96].

[Har99] John Harrison. A machine checked theory of floating point arithmetic. In *TPHOL '99*, volume 1690 of *LNCS*. Springer, 1999.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[Ins85] Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[Jac01a] Christian Jacobi. A formally verified theory of IEEE rounding. Unpublished, available at http://www-wjp.cs.uni-sb.de/˜cj/ieee-lib.ps, 2001.

[Jac01b] Christian Jacobi. *Formal Verification of an IEEE Compliant Floation Point Unit*. PhD thesis, Saarland University, Computer Science Department, due in 2001.

[JK00] Christian Jacobi and Daniel Kröning. Proving the correctness of a complete microprocessor. In *Proc. of the 30. Jahrestagung der Gesellschaft für Informatik*. Springer, 2000.

[JLMC97] Somesh Jha, Yuan Lu, Marius Minea, and Edmund M. Clarke. Equivalence checking using abstract BDDs. In *Proc. of IEEE ICCD '98*, pages 332–337. IEEE, 1997.

[KP97] Joerg Keller and Wolfgang J. Paul. *Hardware Design, Formaler Entwurf digitaler Schaltungen*, volume 15 of *Teubner-Texte zur Informatik*. Teubner, 1997.

[Krö01] Daniel Kröning. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.

[Min95] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Report TM-110167, NASA Langley Research Center, 1995.

[MLK98] J Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5K86 floating point division program. *IEEE Transactions on Computers*, 47(9):913–926, 1998.

[MP95] Silvia M. Mueller and Wolfgang J. Paul. *The Complexity of Simple Computer Architectures*. Springer, 1995.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.

[OSR92] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

[OZGS99] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, Q1, 1999.

[Pra95] V. R. Pratt. Anatomy of the pentium bug. In *TAPSOFT'95*, volume 915, pages 97–107, Aarhus, Denmark, 1995. Springer-Verlag.

[PVS] PVS bugs. http://pvs.csl.sri.com/cgi-bin/pvs/pvs-bug-list/, Bugs #474, 529, 538, 551.

[Rus98] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[Rus99] David M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, January 1999.

[Rus00] David M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. In *FMCAD-00*, volume 1954 of *LNCS*. Springer, 2000.

[SBE88] V. Stavridou, H. Barringer, and D.A. Edwards. Formal specification and verification of hardware: A comparative case study. In *Proceedings of the 25th ACM/IEEE conference on Design Automation*, pages 197–204, 1988.

[Sta99] Ted Stanion. Implicit verification of structurally dissimilar arithmetic circuits. In *Proc. of IEEE ICCD '99*, pages 46–50. IEEE, 1999.

[VAM] The VAMP processor homepage. http://www-wjp.cs.uni-sb.de/projects/verification/.

[Xil00] Xilinx, Inc. *Virtex-E 1.8V Field Programmable Gate Arrays, Preliminary Product Specification*, 2000.