TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Formal derivations of binary arithmetic

Mathijssen, A.H.J.

Award date:
2003

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathmatics and Computing Science

MASTER'S THESIS

Formal derivations of binary arithmetic

by

A.H.J. Mathijssen

Supervisor: dr. ir. R.R. Hoogerwoord

Eindhoven, June 2003

Contents

1 Introduction **1**
- 1.1 Subject . . . 1
- 1.2 Method . . . 1
- 1.3 Structure . . . 2

2 Number representations **3**
- 2.1 Natural numbers . . . 3
- 2.2 Integers . . . 4
- 2.3 Alternative integer representations . . . 7

3 Simple arithmetic operations **11**
- 3.1 Complement . . . 11
- 3.2 Increment . . . 12
- 3.3 Decrement . . . 13
- 3.4 Negation . . . 14
- 3.5 Doubling and halving . . . 15

4 Addition and subtraction **17**
- 4.1 Integer addition . . . 17
- 4.2 Binary addition . . . 20
- 4.3 Carry-save addition . . . 22
- 4.4 Subtraction . . . 25

5 Multiplication **27**
- 5.1 Integer multiplication . . . 27
- 5.2 Binary multiplication . . . 29
- 5.3 Multiplication with carry-save addition . . . 31
- 5.4 Booth multiplication . . . 33

6 Division **37**
- 6.1 Integer division . . . 37
- 6.2 Binary division . . . 39

7 Hardware implementations **45**
- 7.1 A formalism . . . 45
- 7.2 Arithmetic operations . . . 51

8 Conclusions and recommendations **63**

Chapter 1

Introduction

1.1 Subject

Over the past decades, increasingly more calculations are performed by programs, in both software and hardware. This is especially the case for the basic arithmetic operations, i.e. addition, subtraction, multiplication and division. Although much can be found in the literature concerning implementations of these operations, there are few efforts to give correctness proofs.

In this thesis we construct implementations for the integer variants of the basic arithmetic operations, provided with correctness proofs. Our main aim is the construction of combinatorial circuits. The method and the results can be used for other purposes, however, such as sequential circuits or software programs.

1.2 Method

In the construction of the implementations we distinguish three different levels of reasoning. The top level is the arithmetic level in which we investigate the structure of the basic arithmetic operations. The results from this level are then transformed to representations in the representation level. The results from this level are finally used in the implementation level to obtain combinatorial circuits for the operations. We will only use the arithmetic level if this simplifies the representation level.

We use a *functional* programming language and a *calculational style* of programming, i.e. programs are derived from their specifications by means of formula manipulation. In this way the programs are correct by construction. This method is developed by Hoogerwoord in [Hoo 1] and [Hoo 2].

We now briefly discuss some concepts used in [Hoo 2]. A *declaration* is a definition (that is admissible) in the functional language. In particular, such definitions may be recursive. For the derivation of declarations from their specifications, we will often use the following strategy. Suppose we need to solve an equation of the shape $x : f{\cdot}x = E$. We can try to rewrite expression E into an equivalent expression of the shape $f{\cdot}F$. This transforms the equation into the equivalent equation $x : f{\cdot}x = f{\cdot}F$, of which $x = F$ is a solution, because of Leibniz' rule.

For any type B, the datatype $\mathcal{L}_*(B)$ is the type of all finite lists with elements of type B. For list s, $\#s$ denotes the length of s and $s{\cdot}i$, with $0 \leq i < \#s$, denotes element i of s. The empty list is $[\,]$, pronounced as "empty". The binary operators $\triangleright$

and $\triangleleft$ are pronounced as "cons" and "snoc", respectively. For $b \in B$ and $s \in \mathcal{L}_*(B)$, the list $b \triangleright s$ has b as its head and s as its tail. $\triangleleft$ is the complementary operation of $\triangleright$. Hence, we have $(b \triangleright s)\cdot 0 = b$ and $(s \triangleleft b)\cdot(\#s) = b$. For $c \in B$, all lists s of length 2 have the shape $b \triangleright c \triangleright [\,]$, $b \triangleright [\,] \triangleleft c$ and $[\,] \triangleleft b \triangleleft c$, which may be abbreviated to $[\,b, c\,]$. In current implementations of functional programming languages, $b \triangleright s$ is evaluated in $\mathcal{O}(1)$ steps, but usually $s \triangleleft b$ is evaluated in $\mathcal{O}(\#s)$ steps. In this thesis, we assume that both expressions are evaluated in $\mathcal{O}(1)$ steps. For our purpose of implementations as combinatorial circuits this doesn't matter.

1.3 Structure

This thesis consists of three parts. The first part, which is chapter 2, deals with the representation of natural numbers and integers.

In the second part, which consists of chapters 3 through 6, we derive declarations for the basic arithmetic operations at the arithmetic and the representation level. In chapter 3 we derive declarations for simple arithmetic functions that we will need in the rest of the second part. In chapters 4 through 6, we derive declarations for addition and subtraction, multiplication and division, respectively.

In the third part, which is chapter 7, we provide for a formalism that gives hardware implementations for a subset of the functional programming language. After that we give hardware implementations for the functions derived in the second part.

Chapter 2

Number representations

In this chapter we define how numbers are represented in the binary number system. We do this with finite lists of bits. These are lists of type $\mathcal{L}_*(\{0,1\})$, or $\mathcal{L}2$ for short. In the first two sections we define how we represent natural numbers and integers by these lists. In the last section we define a number of representations of integers by lists that are not of type $\mathcal{L}2$.

2.1 Natural numbers

We introduce abstraction function *v2*, of type $\mathcal{L}2 \to \mathbb{N}$, which interprets a finite list of bits as a natural number. Function *v2* has the following definition, for list s of length n:

$$v2 \cdot s = (\textstyle\sum i : 0 \leq i < n : s \cdot i * 2^{n-1-i}) \tag{1}$$

This is the usual definition for representing natural numbers in the binary number system. The first bit in the list is the most significant bit and the last bit is the least significant bit. Note that the representation of a number is not unique, e.g. the number 3 can be represented by the lists $[1,1]$ and $[0,1,1]$, because $v2 \cdot [1,1]$ and $v2 \cdot [0,1,1]$ are both equal to 3. However, we have the following *non-redundancy* property for function *v2*, which we give without proof:

$$(\forall t \in \mathcal{L}2 : s \neq t \ \wedge \ \#s = \#t : v2 \cdot s \neq v2 \cdot t) \tag{2}$$

If s contains only 0's, we have $v2 \cdot s = 0$. If s contains only 1's, we have $v2 \cdot s = 2^n - 1$. These values are the minimum and maximum values of $v2 \cdot s$, respectively. Hence $v2 \cdot s$ is in the range $[0, 2^n - 1]$.

Definition (1) is not practical for derivations of binary arithmetic functions. As we will see, many of these functions have recursive definitions. Therefore we would rather have a recursive definition for *v2* as well. We give such a definition without its derivation.

Definition 2.1 For $b \in \{0,1\}$ and $s \in \mathcal{L}2$:

$$\begin{aligned} v2 \cdot [\,] &= 0 \\ v2 \cdot (s \triangleleft b) &= 2 * v2 \cdot s + b \end{aligned}$$

□

In the inductive case, we have used the parameter pattern $s \triangleleft b$. For the parameter pattern $b \triangleright s$, we have:

$$v2 \cdot (b \triangleright s) = b * 2^n + v2 \cdot s \tag{3}$$

This property lends itself less to formula manipulation of arithmetic operations than the inductive case of definition 2.1, because of the exponent n.

From property (3), we have $v2 \cdot (0 \triangleright s) = v2 \cdot s$. With this we can *reduce* or *expand* the representation of a number, e.g. the number 3 can be represented by $[0,1,1]$, but also by $[1,1]$ and $[0,0,1,1]$. Note that we can not always reduce a representation. In this case, we have a *minimal* representation, which is always equal to $[\,]$ or of the form $1 \triangleright s$. The above method is the only way to generate different representations for the same number. This can be easily proved with the aid of non-redundancy property (2).

2.2 Integers

There are various ways to represent integers by finite lists of bits. The three standard representations are sign-and-magnitude, one's complement and two's complement. Of these three, two's complement is most used in practice, because of the ease with which the basic arithmetic operations can be implemented [Omo]. Therefore we use the two's complement representation.

We introduce abstraction function *vn2*, of type $\mathcal{L}2 \rightarrow \mathbb{Z}$, which interprets a finite list of bits as an integer. Function *vn2* has the following definition, for non-empty list s of length n:

$$vn2 \cdot s = -s \cdot 0 * 2^{n-1} + (\textstyle\sum i : 1 \leq i < n : s \cdot i * 2^{n-1-i}) \tag{4}$$

We call integers represented this way binary integers. As in definition (1) of *v2*, the first bit in the list is the most significant bit and the last bit is the least significant bit. Also, the first bit is called the sign bit, because it determines the sign of the value. This follows from the properties $vn2 \cdot s < 0 \equiv s \cdot 0 = 1$ and $0 \leq vn2 \cdot s \equiv s \cdot 0 = 0$. The representation of an integer is not unique, but, analogous to property (2) of function *v2*, we have the following non-redundancy property for function *vn2*:

$$(\forall t \in \mathcal{L}2 : s \neq t \ \wedge \ \#s = \#t : vn2 \cdot s \neq vn2 \cdot t) \tag{5}$$

If $s \cdot 0 = 1$ and $s \cdot i = 0$, for $1 \leq i < n$, we have $vn2 \cdot s = -2^{n-1}$. If $s \cdot 0 = 0$ and $s \cdot i = 1$, for $1 \leq i < n$, we have $vn2 \cdot s = 2^{n-1} - 1$. These values are the minimum and maximum values of $vn2 \cdot s$, respectively. We now have the following corollary.

Corollary 2.2 For $s \in \mathcal{L}2$, with $s \neq [\,]$ and $n = \#s$, we have that $vn2 \cdot s$ is in the range $[-2^{n-1}, 2^{n-1} - 1]$. $\Box$

From definition (1) of *v2* and definition (4) of *vn2*, it can be seen that functions *v2* and *vn2* are strongly related. This is expressed by the following property:

$$vn2 \cdot s = -s \cdot 0 * 2^n + v2 \cdot s \tag{6}$$

We prove this property by the following derivation:

$$
\begin{array}{ll}
& vn2 \cdot s \\
= & \quad \{ \text{ definition (4) of } vn2 \ \} \\
& -s \cdot 0 * 2^{n-1} + (\sum i : 1 \leq i < n : s \cdot i * 2^{n-1-i}) \\
= & \quad \{ \text{ algebra } \} \\
& -s \cdot 0 * 2^{n} + s \cdot 0 * 2^{n-1} + (\sum i : 1 \leq i < n : s \cdot i * 2^{n-1-i}) \\
= & \quad \{ \text{ join } i = 0 \ \} \\
& -s \cdot 0 * 2^{n} + (\sum i : 0 \leq i < n : s \cdot i * 2^{n-1-i}) \\
= & \quad \{ \text{ definition (1) of } v2 \ \} \\
& -s \cdot 0 * 2^{n} + v2 \cdot s
\end{array}
$$

We use property (6) to derive a recursive definition for $vn2$. We do this by case distinction on n. If $n = 1$, then there exists a $b \in \{ 0, 1 \}$, such that $s = [b]$. Then we derive:

$$
\begin{array}{ll}
& vn2 \cdot [b] \\
= & \quad \{ \text{ property (6) } \} \\
& -[b] \cdot 0 * 2^{1} + v2 \cdot [b] \\
= & \quad \{ [b] \cdot 0 = b, \text{ algebra } \} \\
& -2 * b + v2 \cdot [b] \\
= & \quad \{ \text{ definition (1) of } v2 \ \} \\
& -2 * b + b \\
= & \quad \{ \text{ algebra } \} \\
& -b
\end{array}
$$

For the list $s \lhd b$, with $b \in \{ 0, 1 \}$, we have $\#(s \lhd b) = n + 1$. We now derive as follows:

$$
\begin{array}{ll}
& vn2 \cdot (s \lhd b) \\
= & \quad \{ \text{ property (6) } \} \\
& -(s \lhd b) \cdot 0 * 2^{n+1} + v2 \cdot (s \lhd b) \\
= & \quad \{ \text{ definition (1) of } v2 \ \} \\
& -(s \lhd b) \cdot 0 * 2^{n+1} + 2 * v2 \cdot s + b \\
= & \quad \{ \text{ property of } \lhd, \text{ from } 0 < \#s, \text{ algebra } \} \\
& 2 * (-s \cdot 0 * 2^{n} + v2 \cdot s) + b \\
= & \quad \{ \text{ property (6) } \} \\
& 2 * vn2 \cdot s + b
\end{array}
$$

Thus, we have obtained the following recursive definition for $vn2$.

Definition 2.3 For $b \in \{ 0, 1 \}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$
\begin{array}{lcl}
vn2 \cdot [b] & = & -b \\
vn2 \cdot (s \lhd b) & = & 2 * vn2 \cdot s + b
\end{array}
$$

□

In the above derivation, we have used the parameter pattern $s \lhd b$. For the parameter pattern $b \rhd s$ we derive as follows:

$$
\begin{array}{ll}
 & vn2\cdot(b \triangleright s) \\
= & \quad \{ \text{ property (6) } \} \\
 & -(b \triangleright s)\cdot 0 * 2^{n+1} + v2\cdot(b \triangleright s) \\
= & \quad \{ \text{ definition of } \triangleright\text{, property (3) of } v2 \ \} \\
 & -b * 2^{n+1} + b * 2^{n} + v2\cdot s \\
= & \quad \{ \text{ algebra } \} \\
 & -b * 2^{n} + v2\cdot s
\end{array}
$$

Hence, we have obtained the following property:

$$vn2\cdot(b \triangleright s) = -b * 2^{n} + v2\cdot s \qquad (7)$$

2.2.1 List reduction and expansion

Inspired by properties (3) and (7), we have the following useful property:

$$vn2\cdot(b \triangleright b \triangleright s) = vn2\cdot(b \triangleright s) \qquad (8)$$

We prove this property as follows, for s of length n:

$$
\begin{array}{ll}
 & vn2\cdot(b \triangleright b \triangleright s) \\
= & \quad \{ \text{ property (7) of } vn2 \ \} \\
 & -b * 2^{n+1} + v2\cdot(b \triangleright s) \\
= & \quad \{ \text{ property (3) of } v2 \ \} \\
 & -b * 2^{n+1} + b * 2^{n} + v2\cdot s \\
= & \quad \{ \text{ algebra } \} \\
 & -b * 2^{n} + v2\cdot s \\
= & \quad \{ \text{ property (7) of } vn2 \ \} \\
 & vn2\cdot(b \triangleright s)
\end{array}
$$

With property (8) the representation of an integer can be reduced or expanded. Also this is the only way to generate different representations for the same number. This can be easily proved with the aid of property (5). Note that if a representation cannot be reduced, it is minimal. We can also use property (8) to reduce or expand a representation until it has a desired length, if possible. We call this *normalization*. For this purpose we introduce function *norm*, of type $\mathbb{N} \rightarrow \mathcal{L}2 \rightarrow \mathcal{L}2$, which performs normalization. Function *norm* has the following specification.

Specification 2.4 For $m \in \mathbb{N}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$
\begin{array}{ll}
vn2\cdot(norm\cdot m\cdot s) & = vn2\cdot s \ \wedge \\
\#(norm\cdot m\cdot s) & = (\mathbf{min}\ t \in \mathcal{L}2 : vn2\cdot t = vn2\cdot s \ \wedge \ \#t \geq m : \#t)
\end{array}
$$

□

Because $\#t \geq 0$ for every list t, $norm\cdot 0$ minimizes arbitrary representations. We choose the following declaration for function *norm*.

Declaration 2.5 For $m \in \mathbb{N}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$ and $n = \#s$:

$$\begin{array}{lll} norm \cdot m \cdot s = & \textbf{if} & n \leq m \rightarrow expand \cdot (m-n) \cdot s \\ & [] & n \geq m \rightarrow reduce \cdot (n-m) \cdot s \\ & \textbf{fi} & \end{array}$$

□

In this declaration, functions *expand* and *reduce*, both of type $\mathbb{N} \rightarrow \mathcal{L}2 \rightarrow \mathcal{L}2$, expand and reduce a representation with a specified number of elements, if possible, respectively. Functions *expand* and *reduce* have the following specifications.

Specification 2.6 For $m \in \mathbb{N}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$ and $n = \#s$:

$$\begin{array}{lll} expand \cdot m \cdot s & = & norm \cdot (n+m) \cdot s \\ reduce \cdot m \cdot s & = & norm \cdot (n-m) \cdot s \ \wedge \ m \leq n \end{array}$$

□

We give the following declarations for functions *expand* and *reduce* without derivation.

Declaration 2.7 For $m \in \mathbb{N}$, $b, c \in \{0, 1\}$ and $s \in \mathcal{L}2$:

$$\begin{array}{ll} expand \cdot 0 \cdot (b \triangleright s) & = b \triangleright s \\ expand \cdot (m+1) \cdot (b \triangleright s) & = expand \cdot m \cdot (b \triangleright b \triangleright s) \\ reduce \cdot m \cdot [b] & = [b] \\ reduce \cdot 0 \cdot (b \triangleright c \triangleright s) & = b \triangleright c \triangleright s \\ reduce \cdot (m+1) \cdot (b \triangleright c \triangleright s) & = \textbf{if } b = c \rightarrow reduce \cdot m \cdot (c \triangleright s) \\ & \quad [] \ b \neq c \rightarrow b \triangleright c \triangleright s \\ & \quad \textbf{fi} \end{array}$$

□

Because a representation cannot always be reduced, the length of $norm \cdot m \cdot s$ may be greater than m. In this case, we say normalization causes *overflow*. More precisely, $norm \cdot m \cdot s$ causes overflow if and only if $vn2 \cdot s$ is not in the interval $[-2^{m-1}, 2^{m-1} - 1]$.

In most hardware implementations of integer arithmetic operations, all binary integer representations occurring in the operators have the same length, say n. Also it is required that all binary integer representations occurring in the result of the operation have length n as well. We can satisfy this requirement by applying $norm \cdot n$ to these representations and compare the results to n. If one of the results is larger than n, then overflow has occurred.

2.3 Alternative integer representations

In this section we introduce a number of representations of integers by lists of another type than $\mathcal{L}2$. These representations will be used in several implementations of the basic arithmetic functions. We deal with lists of trits, binary pairs, signed bits and signed 1's. We will treat these list according to the binary representations introduced in the previous sections.

2.3.1 Trits

Lists of trits are of type $\mathcal{L}_*(\{0..2\})$, or $\mathcal{L}3$ for short. We introduce function *vn23*, of type $\mathcal{L}3 \to \mathbb{Z}$, which interprets a finite list of trits as an integer. Function *vn23* has the following definition, for non-empty list s of length n:

$$vn23 \cdot s = -s \cdot 0 * 2^{n-1} + (\sum i : 1 \leq i < n : s \cdot i * 2^{n-1-i}) \tag{9}$$

We call integers represented this way ternary integers. Apart from the type of the parameter, this definition is the same as definition (4) of function *vn2*. Unlike function *vn2*, the representation of an integer is redundant, e.g. the lists $[1,0]$ and $[0,2]$ have the same length and $vn23 \cdot [1,0]$ and $vn23 \cdot [0,2]$ are both equal to 2. If $s \cdot 0 = 2$ and $s \cdot i = 0$, for $1 \leq i < n$, we have $vn23 \cdot s = -2 * 2^{n-1}$ and this is -2^n. If $s \cdot 0 = 0$ and $s \cdot i = 2$, for $1 \leq i < n$, we have $vn23 \cdot s = (\sum i : 1 \leq i < n : 2 * 2^{n-1-i})$ and this is $2^n - 2$, by distributivity of $*$ over $+$. These values are the minimum and maximum values of $vn23 \cdot s$, respectively. We now have the following corollary.

Corollary 2.8 For $s \in \mathcal{L}3$, with $s \neq [\,]$ and $n = \#s$, we have that $vn23 \cdot s$ is in the range $[-2^n, 2^n - 2]$. □

Analogous to the derivation of the recursive definition of function *vn2*, we can derive the following recursive definition for function *vn23*.

Definition 2.9 For $b \in \{0..2\}$ and $s \in \mathcal{L}3$, with $s \neq [\,]$:

$$\begin{aligned} vn23 \cdot [b] &= -b \\ vn23 \cdot (s \triangleleft b) &= 2 * vn23 \cdot s + b \end{aligned}$$

□

2.3.2 Binary pairs

From corollaries 2.2 and 2.8 it can be seen that integers, that are represented by finite lists of trits of length n, can not always be represented by finite lists of bits of length n. However, these integers can be represented by finite lists of pairs of bits of length n, because every trit can be represented by a pair of bits.

Lists of pairs of bits are of type $\mathcal{L}_*(\langle \{0,1\}, \{0,1\} \rangle)$, or $\mathcal{L}2p$ for short. We introduce abstraction function *c2pto3*, of type $\mathcal{L}2p \to \mathcal{L}3$, which interprets a list of binary pairs as a list of trits. Function *c2pto3* has the following definition.

Definition 2.10 For $b, c \in \{0,1\}$ and $s \in \mathcal{L}2p$, with $s \neq [\,]$:

$$c2pto3 \cdot s = h {\bullet} s \ \textbf{whr} \ h \cdot \langle b, c \rangle = b + c \ \textbf{end}$$

□

Each trit is now uniquely represented by a pair of bits. Note that the trit 1 can be represented by the two pairs $\langle 0,1 \rangle$ and $\langle 1,0 \rangle$.

We introduce function *vn2p*, of type $\mathcal{L}2p \to \mathbb{Z}$, which interprets an integer from a finite lists of binary pairs. Function *vn2p* has the following specification:

$$vn2p = vn23 \circ c2pto3 \tag{10}$$

We call integers represented this way binary paired integers. From the definition of $\circ$, we have $vn2p \cdot s = vn23 \cdot (h {\bullet} s)$, with $h \cdot \langle b, c \rangle = b + c$. By induction on the length of s, we obtain the following recursive declaration for function *vn2p*, using the definition of $\bullet$ and definition 2.9 of *vn23*.

Declaration 2.11 For $b, c \in \{0, 1\}$ and $s \in \mathcal{L}2p$, with $s \neq []$:

$$\begin{aligned} vn2p \cdot [\langle b, c \rangle] &= -(b + c) \\ vn2p \cdot (s \triangleleft \langle b, c \rangle) &= 2 * vn2p \cdot s + b + c \end{aligned}$$

□

2.3.3 Signed bits

Lists of signed bits are of type $\mathcal{L}_*(\{-1, 0, 1\})$, or $\mathcal{L}s2$ for short. Function $v2s2$, of type $\mathcal{L}s2 \to \mathbb{Z}$, interprets a finite list of signed bits as an integer according to the following definition, for list s of length n:

$$v2s2 \cdot s = (\sum i : 0 \leq i < n : s \cdot i * 2^{n-1-i}) \tag{11}$$

We call integers represented this way signed binary integers. Apart from the type of the parameter, this definition is the same as definition (1) of function $v2$. Unlike $v2$, the representation of an integer is redundant. If s contains only -1's, we have $v2s2 \cdot s = (\sum i : 0 \leq i < n : -2^{n-1-i})$ and this is $1 - 2^n$. If s contains only 1's, we have $v2s2 \cdot s = (\sum i : 0 \leq i < n : 2^{n-1-i})$ and this is $2^n - 1$. These values are the minimum and maximum values of $v2s2 \cdot s$, respectively. Hence $v2s2 \cdot s$ is in the range $[1 - 2^n, 2^n - 1]$. Analogous to the derivation of the recursive definition of function $v2$, we can derive the following recursive definition of $v2s2$.

Definition 2.12 For $b \in \{-1, 0, 1\}$ and $s \in \mathcal{L}s2$:

$$\begin{aligned} v2s2 \cdot [] &= 0 \\ v2s2 \cdot (s \triangleleft b) &= 2 * v2s2 \cdot s + b \end{aligned}$$

□

2.3.4 Signed 1's

The signed 1's representation is a restricted form of the signed bits representation. The restriction is that we only allow lists of type $\mathcal{L}_*(\{-1, 1\})$, or $\mathcal{L}s1$ for short. Integers represented this way can only be 0 or odd.

Because a signed 1 can only have 2 different values, we can represent a signed 1 by a bit. We choose to represent -1 by 0 and 1 by 1. We introduce abstraction function $c2tos1$, of type $\mathcal{L}2 \to \mathcal{L}s1$, which interprets a list of bits as a list of signed 1's. Function $c2tos1$ has the following definition, for $b \in \{0, 1\}$ and $s \in \mathcal{L}2$:

$$c2tos1 \cdot s = h \bullet s \ \textbf{whr}\ h \cdot b = 2 * b - 1 \ \textbf{end}$$

We introduce function $v2s1$, of type $\mathcal{L}2 \to \mathbb{Z}$, with the following specification:

$$v2s1 = vn2s2 \circ c2tos1 \tag{12}$$

We call integers represented this way signed unary integers. From the definition of $\circ$, we have $v2s1 \cdot s = vn2s2 \cdot (h \bullet s)$, with $h \cdot b = 2 * b - 1$. By induction on the length of s, we obtain the following recursive declaration for function $v2s1$, using the definition of $\bullet$ and definition 11 of $v2s2$.

Definition 2.13 For $b \in \{0, 1\}$ and $s \in \mathcal{L}2$:

$$\begin{aligned} v2s1 \cdot [] &= 0 \\ v2s1 \cdot (s \triangleleft b) &= 2 * (v2s1 \cdot s + b) - 1 \end{aligned}$$

□

Chapter 3

Simple arithmetic operations

This chapter demonstrates the technique we use to derive declarations of arithmetic functions. The simple arithmetic functions we introduce are specified in terms binary integers. These functions will also be used in subsequent chapters, when we derive declarations for the basic arithmetic operations.

3.1 Complement

Before we introduce functions that perform simple arithmetic operations, we introduce function *cmpl*, of type $\mathcal{L}2 \rightarrow \mathcal{L}2$, which inverts every element of a list. This function has the following recursive definition.

Definition 3.1 For $b \in \{0,1\}$ and $s \in \mathcal{L}2$:

$$\begin{array}{lcl} cmpl\cdot[\,] & = & [\,] \\ cmpl\cdot(s \triangleleft b) & = & cmpl\cdot s \triangleleft (1-b) \end{array}$$

□

Evaluation of this definition takes $\mathcal{O}(\#s)$ steps. In chapter 7 we show how we can implement this definition in hardware with $\mathcal{O}(1)$ *propagation delay.*

Function *cmpl* has the following useful property.

Property 3.2 For $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$vn2\cdot(cmpl\cdot s) + 1 = -vn2\cdot s$$

□

We prove this property by induction on the length of s. For the base case $s = [\,b\,]$, with $b \in \{0,1\}$, we calculate:

$$\begin{array}{ll} & vn2\cdot(cmpl\cdot[\,b\,]) + 1 \\ = & \quad \{ \text{ definition 3.1 of } cmpl \text{ (twice) } \} \\ & vn2\cdot[\,1-b\,] + 1 \\ = & \quad \{ \text{ definition 2.3 of } vn2 \text{ } \} \end{array}$$

$$
\begin{aligned}
& -(1-b)+1 \\
= & \quad \{ \text{ algebra } \} \\
& -(-b) \\
= & \quad \{ \text{ definition 2.3 of } vn2 \} \\
& -vn2\cdot[\,b\,]
\end{aligned}
$$

For the list $s \triangleleft b$, we derive:

$$
\begin{aligned}
& vn2\cdot(cmpl\cdot(s \triangleleft b)) + 1 \\
= & \quad \{ \text{ definition 3.1 of } cmpl\text{, definition 2.3 of } vn2 \} \\
& 2 * vn2\cdot(cmpl\cdot s) + (1-b) + 1 \\
= & \quad \{ \text{ algebra } \} \\
& 2 * (vn2\cdot(cmpl\cdot s) + 1) - b \\
= & \quad \{ \text{ property 3.2, from induction hypothesis } \} \\
& 2 * (-vn2\cdot s) - b \\
= & \quad \{ \text{ algebra } \} \\
& -(2 * vn2\cdot s + b) \\
= & \quad \{ \text{ definition 2.3 of } vn2 \} \\
& -vn2\cdot(s \triangleleft b)
\end{aligned}
$$

Hence, we have proved the property.

3.2 Increment

Function *inc*, of type $\{0,1\} \to \mathcal{L}2 \to \mathcal{L}2$, which increments a binary integer by a bit, has the following specification.

Specification 3.3 For $c \in \{0,1\}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$vn2\cdot(inc\cdot c\cdot s) = vn2\cdot s + c$$

□

We construct a declaration for this function by induction on the length of s. For the base case $s = [\,b\,]$, with $b \in \{0,1\}$, we derive as follows:

$$
\begin{aligned}
& vn2\cdot(inc\cdot c\cdot[\,b\,]) \\
= & \quad \{ \text{ specification 3.3 of } inc\text{, definition 2.3 of } vn2 \} \\
& -b + c \\
= & \quad \{ \text{ algebra, with } h = b + c \} \\
& h - 2 * c \\
= & \quad \{ \text{ definition of } \mathbf{div} \text{ and } \mathbf{mod}\text{, algebra } \} \\
& 2 * -(c - h \;\mathbf{div}\; 2) + h \;\mathbf{mod}\; 2 \\
= & \quad \{ \text{ definition 2.3 of } vn2 \text{ (twice), from } c - h \;\mathbf{div}\; 2, h \;\mathbf{mod}\; 2 \in \{0,1\} \} \\
& vn2\cdot[\,c - h \;\mathbf{div}\; 2, h \;\mathbf{mod}\; 2\,]
\end{aligned}
$$

Then we may choose the following declaration for the base case of function *inc*:

$$inc \cdot c \cdot [\,b\,] = [\,c - h \text{ **div** } 2, h \text{ **mod** } 2\,] \ \textbf{whr}\ h = b + c\ \textbf{end}$$

We have used the expressions h **div** 2 and h **mod** 2, with $h = b + c$, in the above derivation, because they can be easily implemented in hardware and because they pop up in the next derivation.

For the list $s \lhd b$, we derive as follows:

$$\begin{array}{ll}
 & vn2 \cdot (inc \cdot c \cdot (s \lhd b)) \\
= & \quad \{ \text{ specification 3.3 of } inc\text{, definition 2.3 of } vn2 \ \} \\
 & 2 * vn2 \cdot s + b + c \\
= & \quad \{ \ h = b + c\text{, definition of } \textbf{div} \text{ and } \textbf{mod} \ \} \\
 & 2 * (vn2 \cdot s + h \ \textbf{div}\ 2) + h \ \textbf{mod}\ 2 \\
= & \quad \{ \text{ specification 3.3 of } inc\text{, from induction hypothesis and } h \ \textbf{div}\ 2 \in \{\,0,1\,\} \ \} \\
 & 2 * vn2 \cdot (inc \cdot (h \ \textbf{div}\ 2) \cdot s) + h \ \textbf{mod}\ 2 \\
= & \quad \{ \text{ definition 2.3 of } vn2\text{, from } h \ \textbf{mod}\ 2 \in \{\,0,1\,\} \ \} \\
 & vn2 \cdot (inc \cdot (h \ \textbf{div}\ 2) \cdot s \lhd h \ \textbf{mod}\ 2)
\end{array}$$

Hence, we may choose the following declaration for function *inc*.

Declaration 3.4 For $b, c \in \{\,0,1\,\}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$\begin{array}{lll}
inc \cdot c \cdot [\,b\,] & = [\,c - h \ \textbf{div}\ 2, h \ \textbf{mod}\ 2\,] & \textbf{whr}\ h = b + c\ \textbf{end} \\
inc \cdot c \cdot (s \lhd b) & = inc \cdot (h \ \textbf{div}\ 2) \cdot s \lhd h \ \textbf{mod}\ 2 & \textbf{whr}\ h = b + c\ \textbf{end}
\end{array}$$

□

We could have derived a more efficient declaration for certain cases of s. Instead of declaration 3.4, hardware implementations of this more efficient declaration usually have different execution times for different s of a specific length.

3.3 Decrement

The counterpart of function *inc* is function *dec*, of type $\{\,0,1\,\} \to \mathcal{L}2 \to \mathcal{L}2$, which decrements a binary integer by a bit. Function *dec* has the following specification.

Specification 3.5 For $c \in \{\,0,1\,\}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$vn2 \cdot (dec \cdot c \cdot s) = vn2 \cdot s - c$$

□

The construction of a declaration for this function is analogous to the construction of a declaration for function *inc*. Therefore we give the declaration right away.

Declaration 3.6 For $b, c \in \{\,0,1\,\}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$\begin{array}{lll}
dec \cdot c \cdot [\,b\,] & = [\,h \ \textbf{div}\ 2 + h \ \textbf{mod}\ 2, h \ \textbf{mod}\ 2\,] & \textbf{whr}\ h = b + c\ \textbf{end} \\
dec \cdot c \cdot (s \lhd b) & = dec \cdot (c - h \ \textbf{div}\ 2) \cdot s \lhd h \ \textbf{mod}\ 2 & \textbf{whr}\ h = b + c\ \textbf{end}
\end{array}$$

□

We can also use property 3.2 to derive an alternative declaration for function *dec*:

$$\begin{array}{ll}
 & vn2\cdot(dec\cdot c\cdot s) \\
= & \quad \{ \text{ specification 3.5 of } dec \ \} \\
 & vn2\cdot s - c \\
= & \quad \{ \text{ algebra } \} \\
 & -(-vn2\cdot s + c) \\
= & \quad \{ \text{ property 3.2 of } cmpl, \text{ algebra } \} \\
 & -(vn2\cdot(cmpl\cdot s) + c) - 1 \\
= & \quad \{ \text{ specification 3.3 of } inc \ \} \\
 & -vn2\cdot(inc\cdot c\cdot(cmpl\cdot s)) - 1 \\
= & \quad \{ \text{ property 3.2 of } cmpl \ \} \\
 & vn2\cdot(cmpl\cdot(inc\cdot c\cdot(cmpl\cdot s)))
\end{array}$$

Then we may choose the following alternative declaration for function *dec*.

Declaration 3.7 For $c \in \{0,1\}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$dec\cdot c\cdot s = cmpl\cdot(inc\cdot c\cdot(cmpl\cdot s))$$

□

3.4 Negation

Function *neg*, of type $\mathcal{L}2 \to \mathcal{L}2$, which negates a binary integer, has the following specification.

Specification 3.8 For $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$vn2\cdot(neg\cdot s) = -vn2\cdot s$$

□

To construct a declaration for this function, we derive as follows:

$$\begin{array}{ll}
 & vn2\cdot(neg\cdot s) \\
= & \quad \{ \text{ specification 3.8 of } neg \ \} \\
 & -vn2\cdot s \\
= & \quad \{ \text{ property 3.2 of } cmpl \ \} \\
 & vn2\cdot(cmpl\cdot s) + 1 \\
= & \quad \{ \text{ specification 3.3 of } inc \ \} \\
 & vn2\cdot(inc\cdot 1\cdot(cmpl\cdot s))
\end{array}$$

Hence, we may choose the following declaration for function *neg*.

Declaration 3.9 For $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$neg\cdot s = inc\cdot 1\cdot(cmpl\cdot s)$$

□

We can make this declaration more efficient when we derive by induction on the length of s, using declaration 3.3 of *inc* and definition 3.1 of *cmpl*. We do not show this.

3.5 Doubling and halving

For non-empty binary list s, we have $2 * vn2 \cdot s = vn2 \cdot (s \triangleleft 0)$, from definition 2.3 of $vn2$. Hence we can double a binary integer by adding a 0 at the end of its representation.

Halving of a binary integer is somewhat more complex. We introduce functions *hlvq* and *hlvr* that return the quotient and the remainder after halving, respectively.

3.5.1 Quotient

Function *hlvq*, of type $\mathcal{L}2 \rightarrow \mathcal{L}2$, has the following specification.

Specification 3.10 For $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$vn2 \cdot (hlvq \cdot s) = vn2 \cdot s \ \mathbf{div}\ 2$$

□

To construct a declaration for this function, we derive by case distinction on s. For the case $s = [\,b\,]$, with $b \in \{\,0,1\,\}$, we derive:

$$\begin{array}{ll} & vn2 \cdot (hlvq \cdot [\,b\,]) \\ = & \quad \{\text{ specification 3.10 of } hlvq\text{, definition 2.3 of } vn2 \ \} \\ & -b \ \mathbf{div}\ 2 \\ = & \quad \{\text{ definition of } \mathbf{div}\text{, using } b \in \{\,0,1\,\} \ \} \\ & -b \\ = & \quad \{\text{ definition 2.3 of } vn2 \ \} \\ & vn2 \cdot [\,b\,] \end{array}$$

For the list $s \triangleleft b$, we derive:

$$\begin{array}{ll} & vn2 \cdot (hlvq \cdot (s \triangleleft b)) \\ = & \quad \{\text{ specification 3.10 of } hlvq\text{, definition 2.3 of } vn2 \ \} \\ & (2 * vn2 \cdot s + b) \ \mathbf{div}\ 2 \\ = & \quad \{\text{ property of } \mathbf{div} \ \} \\ & vn2 \cdot s + b \ \mathbf{div}\ 2 \\ = & \quad \{\text{ definition of } \mathbf{div}\text{, using } b \in \{\,0,1\,\} \ \} \\ & vn2 \cdot s \end{array}$$

Then we may choose the following declaration for function *hlvq*.

Declaration 3.11 For $b \in \{\,0,1\,\}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$\begin{array}{ll} hlvq \cdot [\,b\,] & = [\,b\,] \\ hlvq \cdot (s \triangleleft b) & = s \end{array}$$

□

3.5.2 Remainder

Function *hlvr*, of type $\mathcal{L}2 \rightarrow \{0,1\}$, has the following specification.

Specification 3.12 For $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$hlvr \cdot s = vn2 \cdot s \ \mathbf{mod}\ 2$$

□

To construct a declaration for this function, we derive by case distinction on s. For the base case $s = [\,b\,]$, with $b \in \{0,1\}$, we derive:

$$\begin{array}{ll} & hlvr \cdot [\,b\,] \\ = & \quad \{\text{ specification 3.12 of } hlvr\text{, definition 2.3 of } vn2 \ \} \\ & (-b)\ \mathbf{mod}\ 2 \\ = & \quad \{\text{ property of } \mathbf{mod}\text{, using } b \in \{0,1\} \ \} \\ & b\ \mathbf{mod}\ 2 \\ = & \quad \{\text{ definition of } \mathbf{mod}\text{, using } b \in \{0,1\} \ \} \\ & b \end{array}$$

For the list $s \triangleleft b$, we derive:

$$\begin{array}{ll} & hlvr \cdot (s \triangleleft b) \\ = & \quad \{\text{ specification 3.12 of } hlvr\text{, definition 2.3 of } vn2 \ \} \\ & (2 * vn2 \cdot s + b)\ \mathbf{mod}\ 2 \\ = & \quad \{\text{ property of } \mathbf{mod} \ \} \\ & b\ \mathbf{mod}\ 2 \\ = & \quad \{\text{ definition of } \mathbf{mod}\text{, using } b \in \{0,1\} \ \} \\ & b \end{array}$$

Combining the results of the above derivations, we obtain the following declaration for function *hlvr*.

Declaration 3.13 For $b \in \{0,1\}$ and $s \in \mathcal{L}2$:

$$hlvr \cdot (s \triangleleft b) = b$$

□

Chapter 4

Addition and subtraction

In this chapter we introduce the two basic arithmetic operations addition and subtraction. First, we investigate the recursive structure of the addition operation. Then we derive declarations for functions that add a binary integer to a binary integer, a ternary integer and a binary paired integer, respectively. Finally, we use the declarations derived in this chapter to implement subtraction.

4.1 Integer addition

Function *add*, of type $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, adds two integers, with the following specification.

Specification 4.1 For $x, y \in \mathbb{Z}$:

$$add \cdot x \cdot y = x + y$$

□

We derive a declaration for this function. Inspired by corollary 2.2 concerning the range of integers that can be represented by lists of a certain length, we assume $x, y \in \mathbb{Z}$ and $n \in \mathbb{N}$, with $-2^n \leq x, y < 2^n$. For all x and y, such an n exists. We derive by induction on n. For the base case $n = 0$, there exist $b, c \in \{0, 1\}$, such that $x = -b$ and $y = -c$. Then we derive as follows:

$$\begin{array}{ll} & add \cdot (-b) \cdot (-c) \\ = & \quad \{ \text{ specification 4.1 of } add\text{, algebra } \} \\ & -(b + c) \\ = & \quad \{ \ h = b + c, \text{ definition of } \mathbf{div} \text{ and } \mathbf{mod} \ \} \\ & -(2 * h \ \mathbf{div} \ 2 + h \ \mathbf{mod} \ 2) \\ = & \quad \{ \text{ algebra, using } h \ \mathbf{mod} \ 2 = 2 * (h \ \mathbf{mod} \ 2) - h \ \mathbf{mod} \ 2 \ \} \\ & 2 * -(h \ \mathbf{div} \ 2 + h \ \mathbf{mod} \ 2) + h \ \mathbf{mod} \ 2 \end{array}$$

For integers $2 * x + b$ and $2 * y + c$, we have $-2^{n+1} \leq 2 * x + b, 2 * y + c < 2^{n+1}$. Then we may assume $add \cdot x \cdot y = x + y$ as an induction hypothesis. We derive as follows:

$$\begin{array}{ll} & add \cdot (2 * x + b) \cdot (2 * y + c) \\ = & \quad \{ \text{ specification 4.1 of add, algebra } \} \end{array}$$

$$
\begin{array}{ll}
 & 2*(x+y)+b+c \\
= & \quad \{ \text{ specification 4.1 of add, from induction hypothesis } \} \\
 & 2*\mathit{add}\cdot x\cdot y+b+c \\
= & \quad \{ \ h=b+c, \text{ definition of } \mathbf{div} \text{ and } \mathbf{mod}, \text{ algebra } \} \\
 & 2*(\mathit{add}\cdot x\cdot y+h\ \mathbf{div}\ 2)+h\ \mathbf{mod}\ 2
\end{array}
$$

Hence, we have obtained the following declaration for function *add*.

Declaration 4.2 For $b,c \in \{0,1\}$ and $x,y \in \mathbb{Z}$:

$$
\begin{array}{ll}
\mathit{add}\cdot(-b)\cdot(-c) & = 2*-(h\ \mathbf{div}\ 2+h\ \mathbf{mod}\ 2)+h\ \mathbf{mod}\ 2 \\
 & \quad \mathbf{whr}\ h=b+c\ \mathbf{end} \\
\mathit{add}\cdot(2*x+b)\cdot(2*y+c) & = 2*(\mathit{add}\cdot x\cdot y+h\ \mathbf{div}\ 2)+h\ \mathbf{mod}\ 2 \\
 & \quad \mathbf{whr}\ h=b+c\ \mathbf{end}
\end{array}
$$

□

This declaration can be used for a function that adds two binary integers, because of the structure of the recursion and the range of the terms.

We can get rid of the expression h **div** 2 in the recursive case of this declaration by generalizing function *add* to function *adc*, of type $\{0,1\} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, that has the following specification.

Specification 4.3 For $d \in \{0,1\}$ and $x,y \in \mathbb{Z}$:

$$\mathit{adc}\cdot d\cdot x\cdot y = x+y+d$$

□

Then we may choose the following alternative declaration for function *add*.

Declaration 4.4 For $x,y \in \mathbb{Z}$:

$$\mathit{add}\cdot x\cdot y = \mathit{adc}\cdot 0\cdot x\cdot y$$

□

We derive a declaration for function *adc* by induction on n. For the base case, we derive:

$$
\begin{array}{ll}
 & \mathit{adc}\cdot d\cdot(-b)\cdot(-c) \\
= & \quad \{ \text{ specification 4.3 of } \mathit{adc} \ \} \\
 & -b-c+d \\
= & \quad \{ \text{ algebra, with } h=b+c+d \ \} \\
 & h-2*(b+c) \\
= & \quad \{ \text{ definition of } \mathbf{div} \text{ and } \mathbf{mod}, \text{ algebra } \} \\
 & 2*-(b+c-h\ \mathbf{div}\ 2)+h\ \mathbf{mod}\ 2
\end{array}
$$

Analogous to the base case of declaration 3.4 of *inc*, we have used the expressions h **div** 2 and h **mod** 2, with $h=b+c+d$, in the above derivation, because they can be easily implemented in hardware and because they pop up in the next derivation.

For the inductive case, we derive as follows:

$$
\begin{array}{ll}
 & adc \cdot d \cdot (2 * x + b) \cdot (2 * y + c) \\
= & \quad \{ \text{ specification 4.3 of } adc\text{, algebra } \} \\
 & 2 * (x + y) + b + c + d \\
= & \quad \{ \; h = b + c + d\text{, definition of } \mathbf{div} \text{ and } \mathbf{mod} \; \} \\
 & 2 * (x + y + h \; \mathbf{div} \; 2) + h \; \mathbf{mod} \; 2 \\
= & \quad \{ \text{ specification 4.3 of } adc\text{, from induction hypothesis and } h \; \mathbf{div} \; 2 \in \{ 0, 1 \} \; \} \\
 & 2 * adc \cdot (h \; \mathbf{div} \; 2) \cdot x \cdot y + h \; \mathbf{mod} \; 2
\end{array}
$$

Hence, we have obtained the following declaration for function *adc*.

Declaration 4.5 For $b, c, d \in \{ 0, 1 \}$ and $x, y \in \mathbb{Z}$:

$$
\begin{array}{lll}
adc \cdot d \cdot (-b) \cdot (-c) & = & 2 * -(b + c - h \; \mathbf{div} \; 2) + h \; \mathbf{mod} \; 2 \\
 & & \mathbf{whr} \; h = b + c + d \; \mathbf{end} \\
adc \cdot d \cdot (2 * x + b) \cdot (2 * y + c) & = & 2 * adc \cdot (h \; \mathbf{div} \; 2) \cdot x \cdot y + h \; \mathbf{mod} \; 2 \\
 & & \mathbf{whr} \; h = b + c + d \; \mathbf{end}
\end{array}
$$

□

The recursive case of this declaration has one inconvenient property, namely that of *carry propagation*. The parameter d and the expression h **div** 2 are often called the incoming and outgoing carry, respectively. Now the incoming carry is needed for the calculation of the outgoing carry, i.e. the incoming carry is propagated. In hardware implementations, this is usually the bottle-neck for the performance.

To get rid of this carry propagation, we can remove the value of d from h and add it to h **mod** 2. We then obtain as an alternative declaration for the inductive case of *adc*:

$$
\begin{array}{lll}
adc \cdot d \cdot (2 * x + b) \cdot (2 * y + c) & = & 2 * adc \cdot (h \; \mathbf{div} \; 2) \cdot x \cdot y + h \; \mathbf{mod} \; 2 + d \\
 & & \mathbf{whr} \; h = b + c \; \mathbf{end}
\end{array}
$$

The value of h **mod** $2 + d$ can not be represented by a single bit anymore, but it can be represented by a trit. To be able to use the result of function *adc* as a parameter of the same function, we can represent x by a finite list of trits. Inspired by corollary 2.8 concerning the range of integers that can be represented by finite lists of trits of a certain length, we assume $-2^{n+1} \leq x < 2^{n+1} - 1$. For all x such an n exists. We now construct a more general declaration for function *adc* by induction on n. Note that we still assume $-2^n \leq y < 2^n$. For the base case $n = 0$, there exist $b \in \{ 0..2 \}$ and $c \in \{ 0, 1 \}$, such that $x = -b$ and $y = -c$. Then we obtain $adc \cdot d \cdot (-b) \cdot (-c) = 2 * -(h \; \mathbf{div} \; 2 + h \; \mathbf{mod} \; 2) + h \; \mathbf{mod} \; 2 + d$, with $h = b + c$. For the inductive case, we investigate integers $2 * x + b$ and $2 * y + c$, with $b \in \{ 0..2 \}$ and $c \in \{ 0, 1 \}$, for which we have $-2^{n+2} \leq 2 * x + b < 2^{n+2} - 1$ and $-2^{n+1} \leq 2 * y + c < 2^{n+1}$. Then we may assume $adc \cdot d \cdot x \cdot y = x + y + d$ as an induction hypothesis. We now obtain $adc \cdot d \cdot (2 * x + b) \cdot (2 * y + c) = 2 * adc \cdot (h \; \mathbf{div} \; 2) \cdot x \cdot y + h \; \mathbf{mod} \; 2 + d$, with $h = b + c$, because $h \; \mathbf{div} \; 2 \in \{ 0, 1 \}$. Then we have the following alternative declaration for function *adc*.

Declaration 4.6 For $b \in \{ 0..2 \}$, $c, d \in \{ 0, 1 \}$ and $x, y \in \mathbb{Z}$:

$$
\begin{array}{lll}
adc \cdot d \cdot (-b) \cdot (-c) & = & 2 * -(h \; \mathbf{div} \; 2 + h \; \mathbf{mod} \; 2) + h \; \mathbf{mod} \; 2 + d \\
 & & \mathbf{whr} \; h = b + c \; \mathbf{end} \\
adc \cdot d \cdot (2 * x + b) \cdot (2 * y + c) & = & 2 * adc \cdot (h \; \mathbf{div} \; 2) \cdot x \cdot y + h \; \mathbf{mod} \; 2 + d \\
 & & \mathbf{whr} \; h = b + c \; \mathbf{end}
\end{array}
$$

□

To distinguish this declaration from declaration 4.5 of *adc*, which suffers from carry-propagation, this declaration is often called *carry-save* addition.

4.2 Binary addition

Functions *add2*, of type $\mathcal{L}2 \to \mathcal{L}2 \to \mathcal{L}2$, and *adc2*, of type $\{0,1\} \to \mathcal{L}2 \to \mathcal{L}2 \to \mathcal{L}2$, have the following specification.

Specification 4.7 For $d \in \{0,1\}$ and $s,t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$\begin{aligned} vn2\cdot(add2\cdot s\cdot t) &= add\cdot(vn2\cdot s)\cdot(vn2\cdot t) \\ vn2\cdot(adc2\cdot d\cdot s\cdot t) &= adc\cdot d\cdot(vn2\cdot s)\cdot(vn2\cdot t) \end{aligned}$$

□

We construct declarations for these functions using declarations 4.2 and 4.4 of *add* and declaration 4.5 of *adc*. For this purpose, we assume that s and t have equal length.

4.2.1 Declarations for *add2*

We derive a declaration for function *add2* by induction on the length of s. For the base case $s = [\,b\,]$ and $t = [\,c\,]$, with $b, c \in \{0,1\}$, we derive:

$$\begin{aligned} & vn2\cdot(add2\cdot[\,b\,]\cdot[\,c\,]) \\ = \quad & \{ \text{ specification 4.7 of } add2\text{, definition 2.3 of } vn2 \text{ (twice) } \} \\ & add\cdot(-b)\cdot(-c) \\ = \quad & \{ \text{ declaration 4.2 of } add\text{, with } h = b + c \} \\ & 2 * -(h \textbf{ div } 2 + h \textbf{ mod } 2) + h \textbf{ mod } 2 \\ = \quad & \{ \text{ definition 2.3 of } vn2 \text{ (twice), from } h \textbf{ div } 2 + h \textbf{ mod } 2, h \textbf{ mod } 2 \in \{0,1\} \} \\ & vn2\cdot[\,h \textbf{ div } 2 + h \textbf{ mod } 2, h \textbf{ mod } 2\,] \end{aligned}$$

For lists $s \triangleleft b$ and $t \triangleleft c$, we derive as follows:

$$\begin{aligned} & vn2\cdot(add2\cdot(s \triangleleft b)\cdot(t \triangleleft c)) \\ = \quad & \{ \text{ specification 4.7 of } add2\text{, definition 2.3 of } vn2 \text{ (twice) } \} \\ & add\cdot(2 * vn2\cdot s + b)\cdot(2 * vn2\cdot t + c) \\ = \quad & \{ \text{ declaration 4.2 of } add\text{, with } h = b + c \} \\ & 2 * (add\cdot(vn2\cdot s)\cdot(vn2\cdot t) + h \textbf{ div } 2) + h \textbf{ mod } 2 \\ = \quad & \{ \text{ specification 4.7 of } add2\text{, from induction hypothesis } \} \\ & 2 * (vn2\cdot(add2\cdot s\cdot t) + h \textbf{ div } 2) + h \textbf{ mod } 2 \\ = \quad & \{ \text{ specification 3.3 of } inc\text{, from } h \textbf{ div } 2 \in \{0,1\} \} \\ & 2 * vn2\cdot(inc\cdot(h \textbf{ div } 2)\cdot(add2\cdot s\cdot t)) + h \textbf{ mod } 2 \\ = \quad & \{ \text{ definition 2.3 of } vn2\text{, from } h \textbf{ mod } 2 \in \{0,1\} \} \\ & vn2\cdot(inc\cdot(h \textbf{ div } 2)\cdot(add2\cdot s\cdot t) \triangleleft h \textbf{ mod } 2) \end{aligned}$$

Hence, we may choose the following partial declaration for function *add2*.

Declaration 4.8 For $b, c \in \{0,1\}$ and $s, t \in \mathcal{L}2$, with $s \neq [\,]$, $t \neq [\,]$ and $\#s = \#t$:

$$\begin{array}{lcl} add2\cdot[b]\cdot[c] & = & [h \textbf{ div } 2 + h \textbf{ mod } 2, h \textbf{ mod } 2] \\ & & \textbf{whr } h = b + c \textbf{ end} \\ add2\cdot(s \triangleleft b)\cdot(t \triangleleft c) & = & inc\cdot(h \textbf{ div } 2)\cdot(add2\cdot s\cdot t) \triangleleft h \textbf{ mod } 2 \\ & & \textbf{whr } h = b + c \textbf{ end} \end{array}$$

□

Using declaration 3.4 of *inc*, evaluation of this declaration takes $\mathcal{O}((\#s)^2)$ steps. However, using a more efficient declaration of *inc*, the declaration of *add2* can be implemented such that its evaluation only takes $\mathcal{O}(\#s)$ steps[1].

From declaration 4.4 of *add*, we may also choose the following alternative declaration for function *add2*.

Declaration 4.9 For $s, t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$add2\cdot s\cdot t = adc2\cdot 0\cdot s\cdot t$$

□

4.2.2 Declarations for *adc2*

Analogous to the construction of declaration 4.8 of *add2*, we can construct the following partial declaration for function *adc2*, using declaration 4.5 of *adc*.

Declaration 4.10 For $b, c, d \in \{0,1\}$ and $s, t \in \mathcal{L}2$, with $s \neq [\,]$, $t \neq [\,]$ and $\#s = \#t$:

$$\begin{array}{lcl} adc2\cdot d\cdot[b]\cdot[c] & = & [b + c - h \textbf{ div } 2, h \textbf{ mod } 2] \\ & & \textbf{whr } h = b + c + d \textbf{ end} \\ adc2\cdot d\cdot(s \triangleleft b)\cdot(t \triangleleft c) & = & adc2\cdot(h \textbf{ div } 2)\cdot s\cdot t \triangleleft h \textbf{ mod } 2 \\ & & \textbf{whr } h = b + c + d \textbf{ end} \end{array}$$

□

Evaluation of this declaration takes $\mathcal{O}(\#s)$ steps.

The above declaration is partial, because of the restriction to $\#s = \#t$. In hardware implementations this restriction is usually not a problem. Nevertheless we make the above declaration complete. For this purpose we drop the above restriction and consider the two cases $s = [b]$ and $t = [c]$. For the first case, we derive as follows:

$$\begin{array}{ll} & vn2\cdot(adc2\cdot d\cdot[b]\cdot t) \\ = & \quad \{ \text{ specification 4.7 of } adc2\text{, definition 2.3 of } vn2 \} \\ & adc\cdot d\cdot(-b)\cdot(vn2\cdot t) \\ = & \quad \{ \text{ specification 4.5 of } adc \} \\ & vn2\cdot t + d - b \\ = & \quad \{ \bullet \text{ specification 4.11 of } incdec \} \\ & vn2\cdot(incdec\cdot d\cdot b\cdot t) \end{array}$$

Function *incdec*, of type $\{0,1\} \to \{0,1\} \to \mathcal{L}2 \to \mathcal{L}2$, has the following specification.

[1] In [Zan] an implementation of such a declaration is given as rewrite rules for μCRL, for which it is proved that its evaluation takes $\mathcal{O}(\#s)$ steps.

Specification 4.11 For $b, c \in \{0,1\}$ and $s \in \mathcal{L}\mathit{2}$, with $s \neq [\,]$:

$$vn2 \cdot (incdec \cdot b \cdot c \cdot s) = vn2 \cdot s + b - c$$

□

Using functions *inc* and *dec*, we may choose the following declaration for function *incdec*:

Declaration 4.12 For $c \in \{0,1\}$ and $s \in \mathcal{L}\mathit{2}$, with $s \neq [\,]$:

$$\begin{aligned} incdec \cdot 0 \cdot c \cdot s &= dec \cdot c \cdot s \\ incdec \cdot 1 \cdot c \cdot s &= inc \cdot (1 - c) \cdot s \end{aligned}$$

□

Now we may choose the following declaration for the case $s = [\,b\,]$ of *adc2*, and analogous for the case $t = [\,c\,]$:

$$\begin{aligned} adc2 \cdot d \cdot [\,b\,] \cdot t &= incdec \cdot d \cdot b \cdot t \\ adc2 \cdot d \cdot s \cdot [\,c\,] &= incdec \cdot d \cdot c \cdot s \end{aligned}$$

These declarations for function *adc2* make the base case of declaration 4.13 obsolete. Hence, we have obtained the following declaration for function *adc2*.

Declaration 4.13 For $b, c, d \in \{0,1\}$ and $s, t \in \mathcal{L}\mathit{2}$, with $s \neq [\,]$ and $t \neq [\,]$:

$$\begin{aligned} adc2 \cdot d \cdot [\,b\,] \cdot t &= incdec \cdot d \cdot b \cdot t \\ adc2 \cdot d \cdot s \cdot [\,c\,] &= incdec \cdot d \cdot c \cdot s \\ adc2 \cdot d \cdot (s \triangleleft b) \cdot (t \triangleleft c) &= adc2 \cdot (h \;\mathbf{div}\; 2) \cdot s \cdot t \triangleleft h \;\mathbf{mod}\; 2 \\ &\quad \mathbf{whr}\; k = b + c + d \;\mathbf{end} \end{aligned}$$

□

4.3 Carry-save addition

We construct declarations for functions that perform carry-save addition on lists of trits and lists of binary pairs.

4.3.1 Ternary addition

Functions *add23*, of type $\mathcal{L}\mathit{3} \to \mathcal{L}\mathit{2} \to \mathcal{L}\mathit{3}$, and *adc23*, of type $\{0,1\} \to \mathcal{L}\mathit{3} \to \mathcal{L}\mathit{2} \to \mathcal{L}\mathit{3}$, have the following specification.

Specification 4.14 For $d \in \{0,1\}$, $s \in \mathcal{L}\mathit{3}$ and $t \in \mathcal{L}\mathit{2}$, with $s \neq [\,]$ and $t \neq [\,]$:

$$\begin{aligned} vn23 \cdot (add23 \cdot s \cdot t) &= add \cdot (vn23 \cdot s) \cdot (vn2 \cdot t) \\ vn23 \cdot (adc23 \cdot d \cdot s \cdot t) &= adc \cdot d \cdot (vn23 \cdot s) \cdot (vn2 \cdot t) \end{aligned}$$

□

From declaration 4.4 of *add*, we may choose the following declaration for function *add23*.

Declaration 4.15 For $s \in \mathcal{L}3$ and $t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$add23 \cdot s \cdot t = adc23 \cdot 0 \cdot s \cdot t$$

□

Analogous to the construction of declaration 4.8 of *add2*, we can construct the following partial declaration for function *adc23*, using declaration 4.6 of *adc*.

Declaration 4.16 For $b \in \{0..2\}$, $c, d \in \{0,1\}$, $s \in \mathcal{L}3$ and $t \in \mathcal{L}2$, with $s \neq [\,]$, $t \neq [\,]$ and $\#s = \#t$:

$$\begin{array}{lcl} adc23 \cdot d \cdot [\,b\,] \cdot [\,c\,] & = & [\,h \textbf{ div } 2 + h \textbf{ mod } 2, h \textbf{ mod } 2 + d\,] \\ & & \textbf{whr } h = b + c \textbf{ end} \\ adc23 \cdot d \cdot (s \triangleleft b) \cdot (t \triangleleft c) & = & adc23 \cdot (h \textbf{ div } 2) \cdot s \cdot t \triangleleft h \textbf{ mod } 2 + d \\ & & \textbf{whr } h = b + c \textbf{ end} \end{array}$$

□

To be able to convert the result of this declaration to a binary integer, we introduce function *c23to2*, of type $\{0,1\} \to \mathcal{L}3 \to \mathcal{L}2$, with the following specification.

Specification 4.17 For $c \in \{0,1\}$ and $s \in \mathcal{L}3$, with $s \neq [\,]$:

$$vn2 \cdot (c23to2 \cdot c \cdot s) = vn23 \cdot s + c$$

□

Analogous to the construction of declaration 3.4 of *inc*, we can construct the following declaration for function *c23to2*.

Declaration 4.18 For $b \in \{0..2\}$, $c \in \{0,1\}$ and $s \in \mathcal{L}3$, with $s \neq [\,]$:

$$\begin{array}{lcll} c23to2 \cdot c \cdot [\,b\,] & = & [\,c - h \textbf{ div } 2, h \textbf{ mod } 2\,] & \textbf{whr } h = b + c \textbf{ end} \\ c23to2 \cdot c \cdot (s \triangleleft b) & = & c23to2 \cdot (h \textbf{ div } 2) \cdot s \triangleleft h \textbf{ mod } 2 & \textbf{whr } h = b + c \textbf{ end} \end{array}$$

□

4.3.2 Binary paired addition

In subsection 2.3.2 we have shown how we can represent trits by pairs of bits. We now use function *c2pto3* of that section, together with functions *add23* to *adc23* to implement addition of a binary integer to a binary paired integer. We introduce functions *add2p*, of type $\mathcal{L}2p \to \mathcal{L}2 \to \mathcal{L}2p$, and *adc2p*, of type $\{0,1\} \to \mathcal{L}2p \to \mathcal{L}2 \to \mathcal{L}2p$, with the following specification.

Specification 4.19 For $e \in \{0,1\}$, $s \in \mathcal{L}2p$ and $t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$\begin{array}{lcl} vn2p \cdot (add2p \cdot s \cdot t) & = & vn23 \cdot (add23 \cdot (c2pto3 \cdot s) \cdot t) \\ vn2p \cdot (adc2p \cdot e \cdot s \cdot t) & = & vn23 \cdot (adc23 \cdot e \cdot (c2pto3 \cdot s) \cdot t) \end{array}$$

□

From declaration 4.15 of *add23*, we may choose the following declaration for function *add2p*.

Declaration 4.20 For $s \in \mathcal{L}2p$ and $t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$add2p \cdot s \cdot t = adc2p \cdot 0 \cdot s \cdot t$$

□

For the construction of a declaration for function *adc2p*, we assume $\#s = \#t$. We derive a declaration by induction on the length of s and t. For the base case $s = \langle b, c \rangle$ and $t = [d]$, with $b, c, d \in \{0, 1\}$, we derive as follows:

$$\begin{array}{cl}
 & vn2p \cdot (adc2p \cdot e \cdot [\langle b,c \rangle] \cdot [d]) \\
= & \quad \{ \text{ specification 4.19 of } adc2p\text{, definition 2.10 of } c2pto3 \ \} \\
 & vn23 \cdot (adc23 \cdot e \cdot [b+c] \cdot [d]) \\
= & \quad \{ \text{ definition 4.16 of } adc23\text{, with } h = b+c+d \ \} \\
 & vn23 \cdot [h \ \mathbf{div}\ 2 + h \ \mathbf{mod}\ 2, h \ \mathbf{mod}\ 2 + e] \\
= & \quad \{ \text{ definition 2.9 of } vn23 \text{ (twice) } \} \\
 & 2 * -(h \ \mathbf{div}\ 2 + h \ \mathbf{mod}\ 2) + h \ \mathbf{mod}\ 2 + e \\
= & \quad \{ \text{ declaration 2.11 of } vn2p \text{ (twice), from } h \ \mathbf{div}\ 2, h \ \mathbf{mod}\ 2, e \in \{0,1\} \ \} \\
 & vn2p \cdot [\langle h \ \mathbf{div}\ 2, h \ \mathbf{mod}\ 2 \rangle, \langle h \ \mathbf{mod}\ 2, e \rangle]
\end{array}$$

For the lists $s \triangleleft \langle b, c \rangle$ and $t \triangleleft d$, we derive:

$$\begin{array}{cl}
 & vn2p \cdot (adc2p \cdot e \cdot (s \triangleleft \langle b,c \rangle) \cdot (t \triangleleft d)) \\
= & \quad \{ \text{ specification 4.19 of } adc2p\text{, definition 2.10 of } c2pto3 \ \} \\
 & vn23 \cdot (adc23 \cdot e \cdot (c2pto3 \cdot s \triangleleft b + c) \cdot (t \triangleleft d)) \\
= & \quad \{ \text{ definition 4.16 of } adc23\text{, with } h = b+c+d \ \} \\
 & vn23 \cdot (adc23 \cdot (h \ \mathbf{div}\ 2) \cdot (c2pto3 \cdot s) \cdot t \triangleleft h \ \mathbf{mod}\ 2 + e) \\
= & \quad \{ \text{ definition 2.9 of } vn23 \ \} \\
 & 2 * vn23 \cdot (adc23 \cdot (h \ \mathbf{div}\ 2) \cdot (c2pto3 \cdot s) \cdot t) + h \ \mathbf{mod}\ 2 + e \\
= & \quad \{ \text{ specification 4.19 of } adc2p\text{, from induction hypothesis } \} \\
 & 2 * vn2p \cdot (adc2p \cdot (h \ \mathbf{div}\ 2) \cdot s \cdot t) + h \ \mathbf{mod}\ 2 + e \\
= & \quad \{ \text{ declaration 2.11 of } vn2p\text{, from } h \ \mathbf{mod}\ 2, e \in \{0,1\} \ \} \\
 & vn2p \cdot (adc2p \cdot (h \ \mathbf{div}\ 2) \cdot s \cdot t \triangleleft \langle h \ \mathbf{mod}\ 2, e \rangle)
\end{array}$$

Hence, we may choose the following partial declaration for function *adc2p*.

Declaration 4.21 For $b, c, d, e \in \{0, 1\}$, $s \in \mathcal{L}2p$ and $t \in \mathcal{L}2$, with $s \neq [\,]$, $t \neq [\,]$ and $\#s = \#t$:

$$\begin{array}{lcl}
adc2p \cdot e \cdot [\langle b,c \rangle] \cdot [d] & = & [\langle h \ \mathbf{div}\ 2, h \ \mathbf{mod}\ 2 \rangle, \langle h \ \mathbf{mod}\ 2, e \rangle] \\
 & & \mathbf{whr}\ h = b+c+d \ \mathbf{end} \\
adc2p \cdot e \cdot (s \triangleleft \langle b,c \rangle) \cdot (t \triangleleft d) & = & adc2p \cdot (h \ \mathbf{div}\ 2) \cdot s \cdot t \triangleleft \langle h \ \mathbf{mod}\ 2, e \rangle \\
 & & \mathbf{whr}\ h = b+c+d \ \mathbf{end}
\end{array}$$

□

Evaluation of this declaration takes $\mathcal{O}(\#s)$ steps, but we can implement it in hardware with $\mathcal{O}(1)$ propagation delay. We will do this in chapter 7.

To be able to convert the result of this declaration to a binary integer, we introduce function *c2pto2*, of type $\{0, 1\} \rightarrow \mathcal{L}2p \rightarrow \mathcal{L}2$, with the following specification.

Specification 4.22 For $d \in \{0,1\}$ and $s \in \mathcal{L}2p$, with $s \neq []$:

$$vn2\cdot(c2pto2\cdot d\cdot s) = vn2p\cdot s + d$$

□

Analogous to the construction of declaration 3.4 of *inc*, we can construct the following declaration for function *c2pto2*.

Declaration 4.23 For $b, c, d \in \{0,1\}$ and $s \in \mathcal{L}2p$, with $s \neq []$:

$$\begin{aligned} c2pto2\cdot d\cdot[\langle b,c\rangle] &= [b+c-h \textbf{ div } 2, h \textbf{ mod } 2] \\ &\quad \textbf{whr } h = b+c+d \textbf{ end} \\ c2pto2\cdot d\cdot(s \triangleleft \langle b,c\rangle) &= c2pto2\cdot(h \textbf{ div } 2)\cdot s \triangleleft h \textbf{ mod } 2 \\ &\quad \textbf{whr } h = b+c+d \textbf{ end} \end{aligned}$$

□

4.4 Subtraction

In this section we introduce functions that subtract a binary integer from a binary integer, a ternary integer and a binary paired integer, respectively. To construct declarations for these functions, we greatly benefit from the already derived functions in this chapter.

4.4.1 Binary subtraction

Function *subt2*, of type $\mathcal{L}2 \to \mathcal{L}2 \to \mathcal{L}2$, which subtracts a binary integer from a binary integer, has the following specification.

Specification 4.24 For $s, t \in \mathcal{L}2$, with $s \neq []$ and $t \neq []$:

$$vn2\cdot(subt2\cdot s\cdot t) = vn2\cdot s - vn2\cdot t$$

□

To construct a declaration for *subt2*, we derive as follows, for non-empty binary lists s, t:

$$\begin{aligned} & vn2\cdot(subt2\cdot s\cdot t) \\ = & \quad \{\text{ specification 4.24 of } subt2 \} \\ & vn2\cdot s - vn2\cdot t \\ = & \quad \{\text{ property 3.2 of } cmpl \} \\ & vn2\cdot s + vn2\cdot(cmpl\cdot t) + 1 \\ = & \quad \{\text{ specification 4.7 of } adc2 \} \\ & vn2\cdot(adc2\cdot 1\cdot s\cdot(cmpl\cdot t)) \end{aligned}$$

Hence, we may choose the following declaration for function *subt2*.

Declaration 4.25 For $s, t \in \mathcal{L}2$, with $s \neq []$ and $t \neq []$:

$$subt2\cdot s\cdot t = adc2\cdot 1\cdot s\cdot(cmpl\cdot t)$$

□

4.4.2 Carry-save subtraction

Function *subt23*, of type $\mathcal{L}3 \rightarrow \mathcal{L}2 \rightarrow \mathcal{L}3$, which subtracts a binary integer from a ternary integer, has the following specification.

Specification 4.26 For $s \in \mathcal{L}3$ and $t \in \mathcal{L}2$, with $s \neq []$ and $t \neq []$:

$$vn23 \cdot (subt23 \cdot s \cdot t) = vn23 \cdot s - vn2 \cdot t$$

□

Analogous to the previous derivation, we may choose the following declaration for function *subt23*.

Declaration 4.27 For $s \in \mathcal{L}3$ and $t \in \mathcal{L}2$, with $s \neq []$ and $t \neq []$:

$$subt23 \cdot s \cdot t = adc23 \cdot 1 \cdot s \cdot (cmpl \cdot t)$$

□

Function *subt2p*, of type $\mathcal{L}2p \rightarrow \mathcal{L}2 \rightarrow \mathcal{L}2p$, which subtracts a binary integer from a binary paired integer, has the following specification.

Specification 4.28 For $s \in \mathcal{L}2p$ and $t \in \mathcal{L}2$, with $s \neq []$ and $t \neq []$:

$$vn2p \cdot (subt2p \cdot s \cdot t) = vn2p \cdot s - vn2 \cdot t$$

□

Analogous to the derivation at the beginning of this section, we may choose the following declaration for function *subt2p*.

Declaration 4.29 For $s \in \mathcal{L}2p$ and $t \in \mathcal{L}2$, with $s \neq []$ and $t \neq []$:

$$subt2p \cdot s \cdot t = adc2p \cdot 1 \cdot s \cdot (cmpl \cdot t)$$

□

Chapter 5

Multiplication

In this chapter we treat multiplication. In the first section we investigate the recursive structure of the multiplication operation. Then we derive declarations for a function that performs multiplication of two binary integers. After that we try to make the operation more efficient by means of carry-save addition and Booth recoding.

5.1 Integer multiplication

Function *mul*, of type $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, performs multiplication of two integers. This function has the following specification.

Specification 5.1 For $x, y \in \mathbb{Z}$:

$$mul{\cdot}x{\cdot}y = x * y$$

□

For the construction of a declaration of this function, we assume $x, y \in \mathbb{Z}$, $b \in \{0, 1\}$ and $n \in \mathbb{N}$, with $-2^n \leq y < 2^n$. We derive by induction on n. For the base case $n = 0$, we have $mul{\cdot}x{\cdot}(-b) = x * (-b)$, and this is $-(b * x)$. For the inductive case $n + 1$, we have $mul{\cdot}x{\cdot}(2 * y + b) = x * (2 * y + b)$, and this is $2 * x * y + b * x$. This expression can be interpreted as $(2 * x) * y + b * x$ and also as $2 * (x * y) + b * x$. In both cases, we may apply specification 5.1 of *mul*, from induction hypothesis. For the first case, we then obtain the following declaration.

Declaration 5.2 For $b \in \{0, 1\}$ and $x, y \in \mathbb{Z}$:

$$\begin{aligned} mul{\cdot}x{\cdot}(-b) &= -(b * x) \\ mul{\cdot}x{\cdot}(2 * y + b) &= mul{\cdot}(2 * x){\cdot}y + b * x \end{aligned}$$

□

For the second case, we obtain the following declaration.

Declaration 5.3 For $b \in \{0, 1\}$ and $x, y \in \mathbb{Z}$:

$$\begin{aligned} mul{\cdot}x{\cdot}(-b) &= -(b * x) \\ mul{\cdot}x{\cdot}(2 * y + b) &= 2 * mul{\cdot}x{\cdot}y + b * x \end{aligned}$$

□

In both declarations, parameters x and y are called the multiplicand and the multiplier, respectively.

In the inductive case of the above declarations, $b*x$ is added to $2*x*y$ *after* $2*x*y$ is evaluated. This means that all additions are postponed until function *mul* is completely unfolded. For large numbers, this requires a considerable amount of space. Therefore, we derive an alternative declaration for function *mul*, in which the addition *can* be performed immediately. For this purpose, we generalize function *mul* to *gmul*, of type $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, that has the following specification.

Specification 5.4 For $x, y, z \in \mathbb{Z}$:

$$gmul \cdot z \cdot x \cdot y = x * y + z$$

□

Then we may choose the following declaration for function *mul*.

Declaration 5.5 For $x, y \in \mathbb{Z}$:

$$mul \cdot x \cdot y = gmul \cdot 0 \cdot x \cdot y$$

□

We derive a declaration for *gmul* by induction on n. We assume $z \in \mathbb{Z}$. For the base case $n = 0$, we have $gmul \cdot z \cdot x \cdot (-b) = z + x * (-b)$, and this is $z - b * x$. For the inductive case $n+1$, we have $gmul \cdot z \cdot x \cdot (2*y+b) = x*(2*y+b)+z$, and this is $2*x*y+z+b*x$. Like the previous derivation, this expression can be interpreted as $(2*x)*y+z+b*x$ and as $2*(x*y)+z+b*x$. In the first case, we may apply specification 5.4 of *gmul*, from induction hypothesis. We then obtain the following declaration for function *gmul*.

Declaration 5.6 For $x, y, z \in \mathbb{Z}$ and $b \in \{0, 1\}$:

$$\begin{aligned} gmul \cdot z \cdot x \cdot (-b) &= z - b * x \\ gmul \cdot z \cdot x \cdot (2*y+b) &= gmul \cdot (z + b*x) \cdot (2*x) \cdot y \end{aligned}$$

□

For the second case, we derive as follows:

$$\begin{aligned} & 2*(x*y)+z+b*x \\ = \quad & \{ \bullet\ h, k\text{: } 2*h+k = z+b*x \} \\ & 2*(x*y)+2*h+k \\ = \quad & \{ \text{algebra} \} \\ & 2*(x*y+h)+k \\ = \quad & \{ \text{specification 5.4 of } gmul\text{, from induction hypothesis} \} \\ & 2*gmul \cdot h \cdot x \cdot y + k \end{aligned}$$

One possible choice for h and k is $h = l \textbf{ div } 2$ and $k = l \textbf{ mod } 2$, with $l = z + b * x$, because $2*h+k = 2*(l \textbf{ div } 2) + l \textbf{ mod } 2$. There are also other possibilities, as we will see in the next sections. Now we may choose the following alternative declaration for function *gmul*.

Declaration 5.7 For $b \in \{0,1\}$ and $x, y, z \in \mathbb{Z}$:

$$\begin{array}{ll} gmul \cdot z \cdot x \cdot (-b) & = z - b * x \\ gmul \cdot z \cdot x \cdot (2 * y + b) & = 2 * gmul \cdot h \cdot x \cdot y + k \\ & \quad \textbf{whr } 2 * h + k = z + b * x \textbf{ end} \end{array}$$

□

In the above declarations of *gmul*, parameters x, y and z are called the multiplicand, the multiplier and the partial product, respectively.

5.1.1 On efficiency

We have derived 4 different declarations that perform multiplication of two integers. The first one is declaration 5.2 of *mul*, the second is declaration 5.3 of *mul*, the third and fourth are declaration 5.5 of *mul*, together with declaration 5.6 of *gmul* and declaration 5.7 of *gmul*, respectively. Now which of the 4 alternatives is the most efficient?

We already know that the third and the fourth alternative can be more *space* efficient than the first two alternatives. We try to find out which of the 4 alternatives is the most *time* efficient. For our purpose of hardware implementations, the most time expensive operations that are used in the alternatives are addition, subtraction and negation, where one of the arguments is the multiplicand. Evaluation of $mul \cdot x \cdot y$ requires the same total amount of such operations for each of the 4 alternatives. The cost of the operations can be expressed in terms of the absolute values of its arguments. Therefore, the alternative that keeps the absolute values of these arguments smaller than the other alternatives, is the most time efficient. From the declarations of the 4 alternatives, it can easily be seen that the fourth alternative is the one we are looking for. Hence, we may say that the fourth alternative — declaration 5.5 of *mul* and 5.7 of *gmul* — is the most efficient alternative.

5.2 Binary multiplication

We introduce function *mul2*, of type $\mathcal{L}2 \to \mathcal{L}2 \to \mathcal{L}2$, and function *gmul2*, of type $\mathcal{L}2 \to \mathcal{L}2 \to \mathcal{L}2 \to \mathcal{L}2$, with the following specification.

Specification 5.8 For $s, t, u \in \mathcal{L}2$, with $s \neq []$, $t \neq []$ and $u \neq []$:

$$\begin{array}{ll} vn2 \cdot (mul2 \cdot s \cdot t) & = mul \cdot (vn2 \cdot s) \cdot (vn2 \cdot t) \\ vn2 \cdot (gmul2 \cdot u \cdot s \cdot t) & = gmul \cdot (vn2 \cdot u) \cdot (vn2 \cdot s) \cdot (vn2 \cdot t) \end{array}$$

□

We use declaration 5.5 of *mul* and declaration 5.7 of *gmul* to construct declarations for these functions. Hence, we may choose the following declaration for function *mul2*.

Declaration 5.9 For $s, t \in \mathcal{L}2$, with $s \neq []$ and $t \neq []$:

$$mul2 \cdot s \cdot t = gmul2 \cdot [\,0\,] \cdot s \cdot t$$

□

For the construction of a declaration for *gmul2*, we need functions *addbmul2* and *subtbmul2*, both of type $\mathcal{L}2 \to \{0,1\} \to \mathcal{L}2 \to \mathcal{L}2$, with the following specification.

Specification 5.10 For $b \in \{0,1\}$ and $s,t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$\begin{aligned} vn2\cdot(addbmul2\cdot s\cdot b\cdot t) &= vn2\cdot s + b * vn2\cdot t \\ vn2\cdot(subtbmul2\cdot s\cdot b\cdot t) &= vn2\cdot s - b * vn2\cdot t \end{aligned}$$

□

Using specifications 4.1, 4.7 and 4.24 of functions *add*, *add2* and *subt2*, respectively, we may choose the following declaration for these functions.

Declaration 5.11 For $s,t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$\begin{aligned} addbmul2\cdot s\cdot 0\cdot t &= s \\ addbmul2\cdot s\cdot 1\cdot t &= add2\cdot s\cdot t \\ subtbmul2\cdot s\cdot 0\cdot t &= s \\ subtbmul2\cdot s\cdot 1\cdot t &= subt2\cdot s\cdot t \end{aligned}$$

□

We derive a declaration for *gmul2* by induction on the length of t. For the base case $t = [\,b\,]$, with $b \in \{0,1\}$, we derive:

$$\begin{aligned} & vn2\cdot(gmul2\cdot u\cdot s\cdot[\,b\,]) \\ = & \quad \{ \text{ specification 5.8 of } gmul2\text{, definition 2.3 of } vn2 \} \\ & gmul\cdot(vn2\cdot u)\cdot(vn2\cdot s)\cdot(-b) \\ = & \quad \{ \text{ declaration 5.7 of } gmul \} \\ & vn2\cdot u - b * vn2\cdot s \\ = & \quad \{ \text{ specification 5.10 of } subtbmul2 \} \\ & vn2\cdot(subtbmul2\cdot u\cdot b\cdot s) \end{aligned}$$

For the list $t \triangleleft b$, we derive:

$$\begin{aligned} & vn2\cdot(gmul2\cdot u\cdot s\cdot(t \triangleleft b)) \\ = & \quad \{ \text{ specification 5.8 of } gmul2\text{, definition 2.3 of } vn2 \} \\ & gmul\cdot(vn2\cdot u)\cdot(vn2\cdot s)\cdot(2 * vn2\cdot t + b) \\ = & \quad \{ \text{ declaration 5.7 of } gmul\text{, with } 2 * h + k = vn2\cdot u + b * vn2\cdot s \} \\ & 2 * gmul\cdot h\cdot(vn2\cdot s)\cdot(vn2\cdot t) + k \\ = & \quad \{ \bullet\ v, c\text{: } h = vn2\cdot v \text{ and } k = c\text{, with } c \in \{0,1\},\ v \in \mathcal{L}2 \text{ and } v \neq [\,] \} \\ & 2 * gmul\cdot(vn2\cdot v)\cdot(vn2\cdot s)\cdot(vn2\cdot t) + c \\ = & \quad \{ \text{ specification 5.8 of } gmul2\text{, from induction hypothesis } \} \\ & 2 * vn2\cdot(gmul2\cdot v\cdot s\cdot t) + c \\ = & \quad \{ \text{ definition 2.3 of } vn2\text{, from } c \in \{0,1\} \} \\ & vn2\cdot(gmul2\cdot v\cdot s\cdot t \triangleleft c) \end{aligned}$$

We have to find $c \in \{0,1\}$ and $v \in \mathcal{L}2$, with $v \neq [\,]$, that satisfy $2 * vn2\cdot v + c = vn2\cdot u + b * vn2\cdot s$. From definition 2.3 of *vn2* and specification 5.10 of *addbmul2*, we may rewrite this equation to $vn2\cdot(v \triangleleft c) = vn2\cdot(addbmul2\cdot u\cdot b\cdot s)$. Then we may choose $v \triangleleft c = addbmul2\cdot u\cdot b\cdot s$, if $\#(addbmul2\cdot u\cdot b\cdot s) > 1$. To satisfy this requirement, we introduce function *f2*, of type $\mathcal{L}2 \to \mathcal{L}2$, with the following specification.

Specification 5.12 For $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$vn2\cdot(f2\cdot s) = vn2\cdot s \;\wedge\; \#(f2\cdot s) > 1$$

□

From property 8 of *vn2*, we may choose the following declaration for function *f2*.

Declaration 5.13 For $b \in \{0,1\}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$\begin{aligned} f2\cdot[b] &= [b,b] \\ f2\cdot(s \triangleleft b) &= s \triangleleft b \end{aligned}$$

□

Then we may choose $v \triangleleft c = f2\cdot(addbmul2\cdot u\cdot b\cdot s)$.

Hence, we may now choose the following declaration for function *gmul2*.

Declaration 5.14 For $b \in \{0,1\}$ and $s,t,u \in \mathcal{L}2$, with $s \neq [\,]$, $t \neq [\,]$ and $u \neq [\,]$:

$$\begin{aligned} gmul2\cdot u\cdot s\cdot[b] &= subtbmul2\cdot u\cdot b\cdot s \\ gmul2\cdot u\cdot s\cdot(t \triangleleft b) &= gmul2\cdot v\cdot s\cdot t \triangleleft c \\ &\quad \textbf{whr } v \triangleleft c = f2\cdot(addbmul2\cdot u\cdot b\cdot s) \textbf{ end} \end{aligned}$$

□

Using declarations 5.11 and 5.13 of *addbmul2*, *subtbmul2* and *f2*, evaluation of this declaration takes $\mathcal{O}(\#t) * \mathcal{O}(\#s)$ steps.

The construction of alternative declarations for *mul2* and *gmul2* using declarations 5.2 and 5.3 of *mul* and 5.6 of *gmul* is not considered here, because this is analogous to the above derivation and because it amounts to less efficient declarations for binary multiplication. Also, there are ways to make the derived declaration for binary multiplication even more efficient. One way is to speed up the additions and subtractions by means of carry-save addition. Another way is to reduce the number of additions and subtractions. These optimizations are the topic of the next two sections.

Remark 5.15 There are various other ways to find $c \in \{0,1\}$ and $v \in \mathcal{L}2$, with $v \neq [\,]$, in the recursive case of declaration 5.14 of *gmul2*, such that $2 * vn2\cdot v + c = vn2\cdot u + b * vn2\cdot s$. We may for instance choose $v = hlvq\cdot w$ and $c = hlvr\cdot w$, with $w = addbmul2\cdot u\cdot b\cdot s$, from $2 * vn2\cdot v + c = 2 * vn2\cdot w \textbf{ div } 2 + vn2\cdot w \textbf{ mod } 2$ and specifications 3.10 and 3.12 of *hlvq* and *hlvr*. □

5.3 Multiplication with carry-save addition

In this section we try to speed up the multiplication operation by replacing the additions and subtractions by their carry-save equivalents. For this purpose, we replace functions *add2* and *subt2* by functions *add2p* and *subt2p*. Then it is necessary to replace functions *addbmul2* and *subtbmul2* by functions *addbmul2p* and *subtbmul2p*, both of type $\mathcal{L}2p \to \{0,1\} \to \mathcal{L}2 \to \mathcal{L}2p$, with the following specification.

Specification 5.16 For $b \in \{0,1\}$, $s \in \mathcal{L}2p$ and $t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$\begin{aligned} vn2p\cdot(addbmul2p\cdot s\cdot b\cdot t) &= vn2p\cdot s + b * vn2\cdot t \\ vn2p\cdot(subtbmul2p\cdot s\cdot b\cdot t) &= vn2p\cdot s - b * vn2\cdot t \end{aligned}$$

□

From specifications 4.1, 4.19 and 4.28 of functions *add*, *add2p* and *subt2p*, respectively, we may choose the following declarations for functions *addbmul2p* and *subtbmul2p*.

Declaration 5.17 For $s \in \mathcal{L}2p$ and $t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$\begin{aligned} addbmul2p\cdot s\cdot 0\cdot t &= s \\ addbmul2p\cdot s\cdot 1\cdot t &= add2p\cdot s\cdot t \\ subtbmul2p\cdot s\cdot 0\cdot t &= s \\ subtbmul2p\cdot s\cdot 1\cdot t &= subt2p\cdot s\cdot t \end{aligned}$$

□

We now replace function *gmul2* by function *gmul2p*, of type $\mathcal{L}2p \to \mathcal{L}2 \to \mathcal{L}2 \to \mathcal{L}2p$, that uses functions *addbmul2p* and *subtbmul2p*. Function *gmul2p* has the following specification.

Specification 5.18 For $s, t \in \mathcal{L}2$ and $u \in \mathcal{L}2p$, with $s \neq [\,]$, $t \neq [\,]$ and $u \neq [\,]$:

$$vn2p\cdot(gmul2p\cdot u\cdot s\cdot t) = gmul\cdot(vn2p\cdot u)\cdot(vn2\cdot s)\cdot(vn2\cdot t)$$

□

From specification 4.22 of *c2pto2*, we may choose the following alternative declaration for function *mul2*.

Declaration 5.19 For $s, t \in \mathcal{L}2$, with $s \neq [\,]$ and $t \neq [\,]$:

$$mul2\cdot s\cdot t = c2pto2\cdot(gmul2p\cdot[\langle 0,0\rangle]\cdot s\cdot t)$$

□

The construction of a declaration for function *gmul2p* is analogous to the construction of declaration 5.14 of function *gmul2*. Hence, we give its declaration immediately.

Declaration 5.20 For $b \in \{0,1\}$, $s, t \in \mathcal{L}2$ and $u \in \mathcal{L}2p$, with $s \neq [\,]$, $t \neq [\,]$ and $u \neq [\,]$:

$$\begin{aligned} gmul2p\cdot u\cdot s\cdot[b] &= subtbmul2p\cdot u\cdot b\cdot s \\ gmul2p\cdot u\cdot s\cdot(t \triangleleft b) &= gmul2p\cdot v\cdot s\cdot t \triangleleft c \\ &\quad \textbf{whr } v \triangleleft c = f2p\cdot(addbmul2p\cdot u\cdot b\cdot s) \textbf{ end} \end{aligned}$$

□

Here, function *f2p*, of type $\mathcal{L}2p \to \mathcal{L}2p$, has the following specification.

Specification 5.21 For $s \in \mathcal{L}2p$, with $s \neq [\,]$:

$$vn2p\cdot(f2p\cdot s) = vn2p\cdot s \;\wedge\; \#(f2p\cdot s) > 1$$

□

Analogous to declaration 5.13 of *f2*, we may choose the following declaration for function *f2p*.

Declaration 5.22 For $b, c \in \{0,1\}$ and $s \in \mathcal{L}2p$, with $s \neq [\,]$:

$$\begin{aligned} f2p\cdot[\langle b,c\rangle] &= [\langle b,c\rangle, \langle b,c\rangle] \\ f2p\cdot(s \triangleleft \langle b,c\rangle) &= s \triangleleft \langle b,c\rangle \end{aligned}$$

□

Like declaration 5.14 of *gmul2*, evaluation of this declaration takes $\mathcal{O}(\#t) * \mathcal{O}(\#s)$ steps. In chapter 7 we implement this declaration in hardware, with $\mathcal{O}(\#t) + \mathcal{O}(\#s)$ propagation delay, however.

5.4 Booth multiplication

In this section we try to make the multiplication operation more efficient by reducing the number of additions and subtractions. We do this by means of so-called Booth recoding, which we explain first.

5.4.1 Booth recoding

For successive bits b and c in binary list s, $b-c$ and $c-b$ are more often equal to 0 than not, when $b=c$ occurs more often than $b \neq c$. We also have $vn2\cdot(s \lhd b \lhd c) = 2 * vn2\cdot(s \lhd b) + b + (c-b)$. We use these observations to implement a function that converts a binary list to a signed binary list. For this purpose, we introduce function br, of type $\{0,1\} \to \mathcal{L}2 \to \mathcal{L}s2$, with the following specification.

Specification 5.23 For $c \in \{0,1\}$ and $s \in \mathcal{L}2$, with $s \neq []$ and $n = \#s$:

$$v2s2\cdot(br\cdot c\cdot s) = vn2\cdot s + c \ \wedge \ (\forall i : 0 \leq i < n-1 : (br\cdot c\cdot s)\cdot i = s\cdot(i+1) - s\cdot i)$$

□

We derive a declaration for function br by induction on n. For the base case $n=1$, we have for the first conjunct $v2s2\cdot(br\cdot c\cdot[b]) = vn2\cdot[b] + c$, and this is $v2s2\cdot[c-b]$, from definitions 2.3 and 2.12 of $vn2$ and $v2s2$, respectively. Because the second conjuct holds trivially, we may choose $br\cdot c\cdot[b] = [c-b]$. For the inductive case $n+1$, we derive for the first conjunct:

$$\begin{array}{ll}
 & v2s2\cdot(br\cdot c\cdot(s \lhd b)) \\
= & \quad \{ \text{ specification 5.23 of } br\text{, definition 2.3 of } vn2 \ \} \\
 & 2 * vn2\cdot s + b + c \\
= & \quad \{ \text{ algebra, to introduce } c-b \ \} \\
 & 2 * (vn2\cdot s + b) + (c-b) \\
= & \quad \{ \text{ specification 5.23 of } br\text{, from induction hypothesis } \} \\
 & 2 * v2s2\cdot(br\cdot b\cdot s) + (c-b) \\
= & \quad \{ \text{ definition 2.12 of } v2s2\text{, from } c-b \in \{-1,0,1\} \ \} \\
 & v2s2\cdot(br\cdot b\cdot s \lhd (c-b))
\end{array}$$

Hence, we choose $br\cdot c\cdot(s \lhd b) = br\cdot b\cdot s \lhd (c-b)$. We have to verify this choice for the second conjunct. For this purpose, we derive as follows:

$$\begin{array}{ll}
 & (\forall i : 0 \leq i < n : (br\cdot c\cdot(s \lhd b))\cdot i = (s \lhd b)\cdot(i+1) - (s \lhd b)\cdot i) \\
\equiv & \quad \{ \text{ declaration of } br \ \} \\
 & (\forall i : 0 \leq i < n : (br\cdot b\cdot s \lhd (c-b))\cdot i = (s \lhd b)\cdot(i+1) - (s \lhd b)\cdot i) \\
\equiv & \quad \{ \text{ property of } \lhd \text{ (twice), from } i < n \text{ and } \#(br\cdot b\cdot s) = n \ \} \\
 & (\forall i : 0 \leq i < n : (br\cdot b\cdot s)\cdot i = (s \lhd b)\cdot(i+1) - s\cdot i) \\
\equiv & \quad \{ \text{ split off } i = n-1\text{, property of } \lhd \text{ (twice) } \} \\
 & (\forall i : 0 \leq i < n-1 : (br\cdot b\cdot s)\cdot i = s\cdot(i+1) - s\cdot i) \ \wedge \\
 & (br\cdot b\cdot s)\cdot(n-1) = b - s\cdot(n-1) \\
\equiv & \quad \{ \text{ induction hypothesis } \} \\
 & (br\cdot b\cdot s)\cdot(n-1) = b - s\cdot(n-1)
\end{array}$$

From the declaration of *br*, it can easily be seen that this holds. Hence, the following declaration for function *br* is correct.

Declaration 5.24 For $b, c \in \{0,1\}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$\begin{aligned} br\cdot c\cdot[\,b\,] &= [\,c-b\,] \\ br\cdot c\cdot(s \triangleleft b) &= br\cdot b\cdot s \triangleleft (c-b) \end{aligned}$$

□

From this declaration and its specification, it can be seen that $br\cdot 0\cdot s$ has exactly $m = (\#i : 0 \leq i < n-1 : s\cdot i = s\cdot(i+1)) + 1 - s\cdot(n-1)$ occurrences of 0. If m is greater than the number of occurrences of 0 in s, we can use function *br* to obtain a list that has more occurrences of 0 than s. We will use this to reduce the number of additions and subtractions in the multiplication operation. This has been applied for the first time by Andrew D. Booth ([Boo]), hence the name Booth recoding.

5.4.2 Binary multiplication with Booth recoding

We introduce function *gmulbr2*, of type $\mathcal{L}2 \rightarrow \mathcal{L}2 \rightarrow \{0,1\} \rightarrow \mathcal{L}2 \rightarrow \mathcal{L}2$, with the following specification.

Specification 5.25 For $c \in \{0,1\}$ and $s, t, u \in \mathcal{L}2$, with $s \neq [\,]$, $t \neq [\,]$ and $u \neq [\,]$:

$$vn2\cdot(gmulbr2\cdot u\cdot s\cdot c\cdot t) = gmul\cdot(vn2\cdot u)\cdot(vn2\cdot s)\cdot(v2s2\cdot(br\cdot c\cdot t))$$

□

Because $vn2\cdot t = v2s2\cdot(br\cdot 0\cdot t)$, we may choose the following alternative declaration for function *gmul2*.

Declaration 5.26 For $s, t, u \in \mathcal{L}2$, with $s \neq [\,]$, $t \neq [\,]$ and $u \neq [\,]$:

$$gmul2\cdot u\cdot s\cdot t = gmulbr2\cdot u\cdot s\cdot 0\cdot t$$

□

We construct a declaration for function *gmulbr2* by induction on the length of t. For the base case, we derive:

$$\begin{aligned} & vn2\cdot(gmulbr2\cdot u\cdot s\cdot c\cdot[\,b\,]) \\ = & \quad \{ \text{ specification 5.25 of } gmulbr2 \ \} \\ & gmul\cdot(vn2\cdot u)\cdot(vn2\cdot s)\cdot(v2s2\cdot(br\cdot c\cdot[\,b\,])) \\ = & \quad \{ \text{ declaration 5.24 of } br\text{, definition 2.12 of } v2s2 \ \} \\ & gmul\cdot(vn2\cdot u)\cdot(vn2\cdot s)\cdot(c-b) \\ = & \quad \{ \text{ specification 5.4 of } gmul\text{, algebra } \} \\ & vn2\cdot u + (c-b) * vn2\cdot s \\ = & \quad \{ \bullet \text{ specification 5.27 of } asbmul2 \ \} \\ & vn2\cdot(asbmul2\cdot u\cdot c\cdot b\cdot s) \end{aligned}$$

Here, function *asbmul2*, of type $\mathcal{L}2 \rightarrow \{0,1\} \rightarrow \{0,1\} \rightarrow \mathcal{L}2 \rightarrow \mathcal{L}2$, has the following specification.

Specification 5.27 For $b, c \in \{0,1\}$ and $s, t \in \mathcal{L}2$, with $s \neq []$ and $t \neq []$:

$$vn2\cdot(asbmul2\cdot s\cdot b\cdot c\cdot t) = vn2\cdot s + (b-c) * vn2\cdot t$$

□

From specification 5.10 of *addbmul2* and *subtbmul2*, we may choose the following declaration for this function.

Declaration 5.28 For $c \in \{0,1\}$ and $s, t \in \mathcal{L}2$, with $s \neq []$ and $t \neq []$:

$$\begin{aligned} asbmul2\cdot s\cdot 0\cdot c\cdot t &= subtbmul2\cdot s\cdot c\cdot t \\ asbmul2\cdot s\cdot 1\cdot c\cdot t &= addbmul2\cdot s\cdot (1-c)\cdot t \end{aligned}$$

□

For the inductive case of function *gmulbr2*, we now derive:

$$\begin{aligned}
& vn2\cdot(gmulbr2\cdot u\cdot s\cdot c\cdot(t \lhd b)) \\
= \quad & \{ \text{ specification 5.25 of } gmulbr2 \ \} \\
& gmul\cdot(vn2\cdot u)\cdot(vn2\cdot s)\cdot(v2s2\cdot(br\cdot c\cdot(t \lhd b))) \\
= \quad & \{ \text{ declaration 5.24 of } br\text{, definition 2.12 of } v2s2 \ \} \\
& gmul\cdot(vn2\cdot u)\cdot(vn2\cdot s)\cdot(2 * v2s2\cdot(br\cdot b\cdot t) + (c-b)) \\
= \quad & \{ \bullet \text{ declaration 5.7 of } gmul\text{, with } 2*h+k = vn2\cdot u + (c-b) * vn2\cdot s \ \} \\
& 2 * gmul\cdot h\cdot(vn2\cdot s)\cdot(v2s2\cdot(br\cdot b\cdot t)) + k \\
= \quad & \{ \bullet\ v, d\text{: } h = vn2\cdot v \text{ and } k = d\text{, with } d \in \{0,1\}, v \in \mathcal{L}2 \text{ and } v \neq [\,] \ \} \\
& 2 * gmul\cdot(vn2\cdot v)\cdot(vn2\cdot s)\cdot(v2s2\cdot(br\cdot b\cdot t)) + d \\
= \quad & \{ \text{ specification 5.25 of } gmulbr2\text{, from induction hypothesis } \} \\
& 2 * vn2\cdot(gmulbr2\cdot v\cdot s\cdot b\cdot t) + d \\
= \quad & \{ \text{ definition 2.3 of } vn2\text{, from } d \in \{0,1\} \ \} \\
& vn2\cdot(gmulbr2\cdot v\cdot s\cdot b\cdot t \lhd d)
\end{aligned}$$

Because we have only used that b is an integer in the derivation of $gmul\cdot z\cdot x\cdot(2*y+b)$, we were allowed to apply declaration 5.7 of *gmul*, although $c-b$ may not be in $\{0,1\}$. We also have to find $d \in \{0,1\}$ and $v \in \mathcal{L}2$, with $v \neq []$, that satisfy $2 * vn2\cdot v + d = vn2\cdot u + (c-b) * vn2\cdot s$. Analogous to the construction of declaration 5.14 of *gmul2*, we may choose $v \lhd d = f2\cdot(asbmul2\cdot u\cdot c\cdot b\cdot s)$. Then we may choose the following declaration for function *gmulbr2*.

Declaration 5.29 For $b, c \in \{0,1\}$ and $s, t, u \in \mathcal{L}2$, with $s \neq []$, $t \neq []$ and $u \neq []$:

$$\begin{aligned} gmulbr2\cdot u\cdot s\cdot c\cdot[\,b\,] &= asbmul2\cdot u\cdot c\cdot b\cdot s \\ gmulbr2\cdot u\cdot s\cdot c\cdot(t \lhd b) &= gmulbr2\cdot v\cdot s\cdot b\cdot t \lhd d \\ & \quad \textbf{whr } v \lhd d = f2\cdot(asbmul2\cdot u\cdot c\cdot b\cdot s) \textbf{ end} \end{aligned}$$

□

From this declaration, it can be seen that parameter b is treated in the same way in the base case and the inductive case. This was the reason for Booth to use this kind of multiplication. It can also be seen that declaration 5.26 of function *gmul2* requires less additions and subtractions than declaration 5.14 of *gmul2*, if $br\cdot 0\cdot t$ contains more 0's than t. The determination if this is the case requires $\mathcal{O}(\#t)$ steps, however. Without this determination, it is not clear which declaration of *gmul2* is the most efficient.

Remark 5.30 Analogous to remark 5.15, we may also choose $v = hlvq \cdot w$ and $c = hlvr \cdot w$, with $w = asbmul2 \cdot u \cdot c \cdot b \cdot s$ for v and c in the inductive case of declaration 5.29 of *gmulbr2*. Table 5.1 lists the value of v for all values of b and c using declarations 5.28 and 5.11 of *asbmul2*, *addbmul2* and *subtbmul2*. Note that v is the new partial product of the recursive call.

b	c	v
0	0	$hlvq \cdot u$
0	1	$hlvq \cdot (add2 \cdot u \cdot s)$
1	0	$hlvq \cdot (subt2 \cdot u \cdot s)$
1	1	$hlvq \cdot u$

Table 5.1: Value of v in the inductive case of function *gmulbr2*

Table 5.1 is analogous to table 3.13 of [Omo]. According to [Omo], this is analogous to table 3.16(a) of [Omo], and hence to rule (1) to (4) of page 238 of [Boo]. □

Chapter 6

Division

In this chapter we treat the last and most complex of the four basic arithmetic operations, namely division. We first investigate the recursive structure of this operation. After that, we derive declarations for a function that performs division on binary integers.

6.1 Integer division

In this section, we treat the division of an integer by a positive number. We use a positive denominator, because division with a negative denominator is hardly used in practice and its meaning is ambiguous. For constant $B \in \mathbb{N}^+$, function *dm*, of type $\mathbb{Z} \rightarrow \langle \mathbb{Z}, \mathbb{N} \rangle$, has the following specification.

Specification 6.1 For $x \in \mathbb{Z}$:

$$dm \cdot x = \langle q, r \rangle \textbf{ whr } q, r\text{: } x = q * B + r \ \wedge \ 0 \leq r < B \textbf{ end}$$

□

To find a declaration for this function, we have to find $\langle q, r \rangle$, with:

$$x = q * B + r \ \wedge \ 0 \leq r < B \qquad (1)$$

We derive a solution for this equation by induction on $n \in \mathbb{N}$, with $-2^n \leq x < 2^n$. For the base case $n = 0$, a bit b exists, such that $x = -b$. If $b = 0$, then $\langle q, r \rangle = \langle 0, 0 \rangle$ is a solution of (1). If $b = 1$, then $\langle q, r \rangle = \langle -1, B - 1 \rangle$ is a solution of (1). For the integer $2 * x + b$, we have $-2^{n+1} \leq 2 * x + b < 2^{n+1}$. Then we may assume as an induction hypothesis that we have $\langle h, k \rangle = dm \cdot x$, with $x = h * B + k \ \wedge \ 0 \leq k < B$. We now derive as follows:

$$\begin{array}{ll} & 2 * x + b \\ = & \quad \{ \ x = h * B + k, \text{ from induction hypothesis } \} \\ & 2 * (h * B + k) + b \\ = & \quad \{ \text{ algebra } \} \\ & (2 * h) * B + (2 * k + b) \\ = & \quad \{ \ l = 2 * k + b \ \} \end{array}$$

$$\begin{array}{ll} & (2*h)*B+l \\ = & \quad \{ \text{ case distinction, algebra } \} \\ & (2*h)*B+l \qquad\qquad , \text{ if } l < B \\ & (2*h+1)*B+(l-B) \ , \text{ if } B \leq l \end{array}$$

From induction hypothesis, we have $0 \leq k < B$, and hence $0 \leq l < 2*B$. If $l < B$, then $\langle q,r \rangle = \langle 2*h,l \rangle$ is a solution of (1). If $B \leq l$, then $\langle q,r \rangle = \langle 2*h+1, l-B \rangle$ is a solution of (1).

Hardware implementations of these solutions have a disadvantage, i.e. the comparison of two binary integers in general takes $\mathcal{O}(p)$ steps, where p is the maximum of the length of the lists. The comparison of a binary integer with 0 only takes $\mathcal{O}(1)$ steps, however. Therefore, using $m = l - B$, we replace $l < B$, $B \leq l$ and $l - B$ by $m < 0$, $0 \leq m$ and m, respectively. Then we obtain the following declaration for function *dm*.

Declaration 6.2 For $b \in \{0,1\}$ and $x \in \mathbb{Z}$:

$$\begin{array}{lll} dm{\cdot}0 & = & \langle 0,0 \rangle \\ dm{\cdot}(-1) & = & \langle -1, B-1 \rangle \\ dm{\cdot}(2*x+b) & = & \textbf{if } m < 0 \rightarrow \langle 2*h,l \rangle \\ & & [] \ \ 0 \leq m \rightarrow \langle 2*h+1,m \rangle \\ & & \textbf{fi whr } \langle h,k \rangle = dm{\cdot}x \ \& \ l = 2*k+b \ \& \ m = l-B \textbf{ end} \end{array}$$

□

This declaration is called *restoring* division. This name is inspired by a possible implementation of the declaration as a sequential circuit. In each cycle of this implementation, an integer representation is subtracted from another integer representation and the result is stored in a register that represents the remainder. When the value of this remainder is negative, the old value of the register is *restored.*

As another possibility, we *weaken* function *dm* to *gdm*, of type $\mathbb{Z} \rightarrow \langle \mathbb{Z},\mathbb{Z} \rangle$, with the following specification.

Specification 6.3 For $x \in \mathbb{Z}$:

$$gdm{\cdot}x = \langle q,r \rangle \textbf{ whr } q,r\text{: } x = q*B+r \ \wedge \ -B \leq r < B \textbf{ end}$$

□

Thus we obtain the following alternative declaration for function *dm*.

Declaration 6.4 For $x \in \mathbb{Z}$:

$$\begin{array}{ll} dm{\cdot}x = & \textbf{if } r < 0 \rightarrow \langle q-1, r+B \rangle \\ & [] \ \ 0 \leq r \rightarrow \langle q,r \rangle \\ & \textbf{fi whr } \langle q,r \rangle = gdm{\cdot}x \textbf{ end} \end{array}$$

□

For the construction of a declaration for function *gdm*, we have to find $\langle q,r \rangle$, with:

$$x = q*B+r \ \wedge \ -B \leq r < B \qquad (2)$$

We derive a solution for this equation by induction on n. For the base $n = 0$, $\langle q,r \rangle = \langle 0,-b \rangle$ is a solution of (2). For the inductive case $n+1$, we assume as an induction hypothesis that we have $\langle h,k \rangle = gdm{\cdot}x$, with $x = h*B+k \ \wedge \ -B \leq k < B$. Then we derive:

$$
\begin{array}{ll}
 & 2*x+b \\
= & \quad \{\ x = h*B+k, \text{ from induction hypothesis } \} \\
 & 2*(h*B+k)+b \\
= & \quad \{\ \text{algebra}\ \} \\
 & (2*h)*B+(2*k+b) \\
= & \quad \{\ l = 2*k+b\ \} \\
 & (2*h)*B+l \\
= & \quad \{\ \text{case distinction, algebra}\ \} \\
 & (2*h-1)*B+(l+B)\ ,\ \text{if } l<0 \\
 & (2*h+1)*B+(l-B)\ ,\ \text{if } 0 \le l
\end{array}
$$

From induction hypothesis, we have $-B \le k < B$, and hence $2*(-B) \le l < 2*B$. If $l < 0$, then $\langle q,r \rangle = \langle 2*h-1, l+B \rangle$ is a solution of (2). If $0 \le l$, then $\langle q,r \rangle = \langle 2*h+1, l-B \rangle$ is a solution of (2). Hence, we obtain the following declaration for function *gdm*.

Declaration 6.5 For $b \in \{0,1\}$ and $x \in \mathbb{Z}$:

$$
\begin{array}{lll}
gdm\cdot(-b) & = & \langle 0, -b \rangle \\
gdm\cdot(2*x+b) & = & \textbf{if } l<0 \rightarrow \langle 2*h-1, l+B \rangle \\
 & & [\!]\ \ 0 \le l \rightarrow \langle 2*h+1, l-B \rangle \\
 & & \textbf{fi whr } \langle h,k \rangle = gdm\cdot x\ \&\ l = 2*k+b \textbf{ end}
\end{array}
$$

□

From this declaration it can be seen that the quotient is always 0 or odd. The declaration is called *non-restoring* division, because in implementations of this declaration as a sequential circuit the register representing the remainder never needs to be restored.

6.2 Binary division

For constant $B2 \in \mathcal{L}2$, with $vn2\cdot B2 = B$, function *dm2*, of type $\mathcal{L}2 \rightarrow \langle \mathcal{L}2, \mathcal{L}2 \rangle$, has the following specification.

Specification 6.6 For $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$dm2\cdot s = \langle t,u \rangle \textbf{ whr } t,u\text{: } \langle vn2\cdot t, vn2\cdot u \rangle = dm\cdot(vn2\cdot s) \textbf{ end}$$

□

To construct a declaration for this function, we use the restoring and the non-restoring version of *dm*, respectively.

6.2.1 Restoring division

We construct a declaration for function *dm2* by induction on the length of s. For the base case $s = [0]$, we have $dm\cdot(vn2\cdot[0]) = \langle vn2\cdot[0], vn2\cdot[0] \rangle$, from definition 2.3 of *vn2* and declaration 6.2 of *dm*. Then we obtain $dm2\cdot[0] = \langle [0],[0] \rangle$, from specification 6.6 of *dm2*. For the other base case $s = [1]$, we have $dm\cdot(vn2\cdot[1]) = \langle vn2\cdot[1], vn2\cdot(dec\cdot 1\cdot B2) \rangle$, from definition 2.3 of *vn2*, declaration 6.2 of *dm* and specification 3.5 of *dec*. So we obtain $dm2\cdot[1] = \langle [1], dec\cdot 1\cdot B2 \rangle$. For the list $s \triangleleft b$, with

$b \in \{0,1\}$, we assume as induction hypothesis that we have $\langle t,u \rangle = \mathit{dm2}\cdot s$, with $\langle \mathit{vn2}\cdot t, \mathit{vn2}\cdot u \rangle = \mathit{dm}\cdot(\mathit{vn2}\cdot s)$. Now, from definition 2.3 of *vn2* and declaration 6.2 of *dm*, we have:

$$\begin{aligned} \mathit{dm}\cdot(\mathit{vn2}\cdot(s \lhd b)) = \ & \textbf{if}\ m < 0 \rightarrow \langle 2*h, l \rangle \\ & []\ 0 \leq m \rightarrow \langle 2*h+1, m \rangle \\ & \textbf{fi}\ \textbf{whr}\ \langle h,k \rangle = \mathit{dm}\cdot(\mathit{vn2}\cdot s)\ \&\ l = 2*k+b \\ & \quad \&\ m = l - B\ \textbf{end} \end{aligned} \tag{3}$$

From the induction hypothesis, we have $h = \mathit{vn2}\cdot t$ and $k = \mathit{vn2}\cdot u$. Hence, we have $l = \mathit{vn2}\cdot v$, with $v = u \lhd b$. We also have $m = \mathit{vn2}\cdot w$, with $w = \mathit{subt2}\cdot v\cdot B2$, from $B = \mathit{vn2}\cdot B2$ and specification 4.24 of *subt2*. We use these equalities to perform substitutions in (3), that are listed in table 6.1 .

old	*new*
$m < 0$	$w\cdot 0 = 1$
$0 \leq m$	$w\cdot 0 = 0$
$2*h$	$\mathit{vn2}\cdot(t \lhd 0)$
$2*h+1$	$\mathit{vn2}\cdot(t \lhd 1)$
l	$\mathit{vn2}\cdot v$
m	$\mathit{vn2}\cdot w$

Table 6.1: Substitutions in (3)

From specification 6.6 of *dm2*, we then obtain the following declaration for function *dm2*.

Declaration 6.7 For $b \in \{0,1\}$ and $s \in \mathcal{L}2$, with $s \neq []$:

$$\begin{aligned} \mathit{dm2}\cdot[0] &= \langle [0],[0] \rangle \\ \mathit{dm2}\cdot[1] &= \langle [1], \mathit{dec}\cdot 1\cdot B2 \rangle \\ \mathit{dm2}\cdot(s \lhd b) &= \textbf{if}\ w\cdot 0 = 1 \rightarrow \langle t \lhd 0, v \rangle \\ &\quad []\ w\cdot 0 = 0 \rightarrow \langle t \lhd 1, w \rangle \\ &\quad \textbf{fi}\ \textbf{whr}\ \langle t,u \rangle = \mathit{dm2}\cdot s\ \&\ v = u \lhd b\ \&\ w = \mathit{subt2}\cdot v\cdot B2\ \textbf{end} \end{aligned}$$

□

6.2.2 Non-restoring division

In this subsection we construct a declaration for function *dm2* using declaration 6.4 of *dm*. To start with, we introduce function *gdm2* that divides a binary integer by a binary integer, according to specification 6.3 of *gdm*. We choose the quotient in this function to be a signed unary integer[1] instead of a binary integer, because both the expressions $2*h+1$ and $2*h-1$ in declaration 6.5 of *gdm* can be implemented using function *v2s1* such that their evaluation takes $\mathcal{O}(1)$ steps. The restriction that signed unary integers can only be 0 or odd is not a problem, because the quotient in declaration 6.5 of *gdm* can also be only 0 or odd. Then function *gdm2*, of type $\mathcal{L}2 \rightarrow \langle \mathcal{L}2, \mathcal{L}2 \rangle$, has the following specification.

[1]Signed unary integers are discussed in subsection 2.3.4.

Specification 6.8 For $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$gdm2 \cdot s = \langle t, u \rangle \ \textbf{whr}\ t, u\text{: } \langle v2s1 \cdot t, vn2 \cdot u \rangle = gdm \cdot (vn2 \cdot s)\ \textbf{end}$$

□

We use this specification for the construction of a declaration for function *dm2*. From declaration 6.4 of *dm*, we have:

$$\begin{array}{rl} dm \cdot (vn2 \cdot s) = & \textbf{if}\ r < 0 \rightarrow \langle q-1, r+B \rangle \\ & [\,]\ 0 \leq r \rightarrow \langle q, r \rangle \\ & \textbf{fi}\ \textbf{whr}\ \langle q, r \rangle = gdm \cdot (vn2 \cdot s)\ \textbf{end} \end{array} \qquad (4)$$

From specification 6.8 of *gdm2*, we assume $\langle t, u \rangle = gdm2 \cdot s$, with $\langle v2s1 \cdot t, vn2 \cdot u \rangle = gdm \cdot (vn2 \cdot s)$. Then we have $q = v2s1 \cdot t$ and $r = vn2 \cdot u$. We use these equalities to perform substitutions in (4), that are listed in table 6.2.

old	*new*
$r < 0$	$u \cdot 0 = 1$
$0 \leq r$	$u \cdot 0 = 0$
$q - 1$	$vn2 \cdot (ds1to2 \cdot 1 \cdot t)$
q	$vn2 \cdot (ds1to2 \cdot 0 \cdot t)$
$r + B$	$vn2 \cdot (add2 \cdot u \cdot B2)$
r	$vn2 \cdot u$

Table 6.2: Substitutions in (4)

In this table, function *ds1to2*, of type $\{0,1\} \rightarrow \mathcal{L}2 \rightarrow \mathcal{L}2$, has the following specification.

Specification 6.9 For $c \in \{0,1\}$ and $s \in \mathcal{L}2$:

$$vn2 \cdot (ds1to2 \cdot c \cdot s) = v2s1 \cdot s - c$$

□

We derive a declaration for this function at the end of this section.

Now, from specification 6.6 of *dm2*, we obtain the following declaration for function *dm2*.

Declaration 6.10 For $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$\begin{array}{rl} dm2 \cdot s = & \textbf{if}\ u \cdot 0 = 1 \rightarrow \langle ds1to2 \cdot 1 \cdot t, add2 \cdot u \cdot B2 \rangle \\ & [\,]\ u \cdot 0 = 0 \rightarrow \langle ds1to2 \cdot 0 \cdot t, u \rangle \\ & \textbf{fi}\ \textbf{whr}\ \langle t, u \rangle = gdm2 \cdot s\ \textbf{end} \end{array}$$

□

We derive a declaration for function *gdm2* by induction on the length of s. For the base case $s = [b]$, with $b \in \{0,1\}$ we have $gdm \cdot (vn2 \cdot [b]) = \langle v2s1 \cdot [\,], vn2 \cdot [b] \rangle$, from definition 2.3 of *vn2*, definition 2.13 of *v2s1* and declaration 6.5 of *gdm*. Then we obtain $gdm2 \cdot [b] = \langle [\,], [b] \rangle$, from specification 6.8 of *gdm2*. For the list $s \triangleleft b$, we

assume as induction hypothesis that we have $\langle t,u \rangle = gdm2 \cdot s$, with $\langle v2s1 \cdot t, vn2 \cdot u \rangle = gdm \cdot (vn2 \cdot s)$. From definition 2.3 of *vn2* and declaration 6.5 of *gdm*, we have:

$$\begin{array}{rl} gdm \cdot (vn2 \cdot (s \triangleleft b)) = & \textbf{if } l < 0 \rightarrow \langle 2*h-1, l+B \rangle \\ & [] \;\; 0 \leq l \rightarrow \langle 2*h+1, l-B \rangle \\ & \textbf{fi whr } \langle h,k \rangle = gdm \cdot (vn2 \cdot s) \;\&\; l = 2*k+b \textbf{ end} \end{array} \quad (5)$$

From the induction hypothesis, we have $h = v2s1 \cdot t$ and $k = vn2 \cdot u$. Hence, we also have $l = vn2 \cdot v$, with $v = u \triangleleft b$. We use these equalities to perform substitutions in (5), that are listed in table 6.3.

old	*new*
$l < 0$	$v \cdot 0 = 1$
$0 \leq l$	$v \cdot 0 = 0$
$2*h-1$	$v2s1 \cdot (t \triangleleft 0)$
$2*h+1$	$v2s1 \cdot (t \triangleleft 1)$
$l+B$	$vn2 \cdot (add2 \cdot v \cdot B2)$
$l-B$	$vn2 \cdot (subt2 \cdot v \cdot B2)$

Table 6.3: Substitutions in (5)

From specification 6.8 of *gdm2*, we obtain the following declaration for function *gdm2*.

Declaration 6.11 For $b \in \{0,1\}$ and $s \in \mathcal{L}2$, with $s \neq [\,]$:

$$\begin{array}{rl} gdm2 \cdot [b] = & \langle [\,],[b] \rangle \\ gdm2 \cdot (s \triangleleft b) = & \textbf{if } v \cdot 0 = 1 \rightarrow \langle t \triangleleft 0, add2 \cdot v \cdot B2 \rangle \\ & [] \;\; v \cdot 0 = 0 \rightarrow \langle t \triangleleft 1, subt2 \cdot v \cdot B2 \rangle \\ & \textbf{fi whr } \langle t,u \rangle = gdm2 \cdot s \;\&\; v = u \triangleleft b \textbf{ end} \end{array}$$

□

The only thing that is left to do, is to find a declaration for function *ds1to2*, of which we repeat its specification, for bit c and binary list s:

$$vn2 \cdot (ds1to2 \cdot c \cdot s) = v2s1 \cdot s - c$$

Using induction on the length of s, we can easily derive the following declaration.

Declaration 6.12 For $b, c \in \{0,1\}$ and $s \in \mathcal{L}2$:

$$\begin{array}{rl} ds1to2 \cdot c \cdot [\,] = & [c] \\ ds1to2 \cdot c \cdot (s \triangleleft b) = & ds1to2 \cdot (1-b) \cdot s \triangleleft (1-c) \end{array}$$

□

Evaluation of this declaration requires $\mathcal{O}(\#s)$ steps. Because of the carry-propagation in the recursive case of the declaration, hardware implementations of this declaration have $\mathcal{O}(\#s)$ propagation delay. It can also be seen from this declaration that in the evaluation of $ds1to2 \cdot c \cdot (b \triangleright s)$, b and c are inverted once and every element of s is inverted twice. This brings us to the following conjecture:

$$ds1to2 \cdot c \cdot (b \triangleright s) = (1-b) \triangleright s \triangleleft (1-c) \quad (6)$$

We prove this conjecture by induction on the length of s. For the base case $s = [\,]$, we derive:

$$
\begin{array}{ll}
& ds1to2\cdot c\cdot(b \triangleright [\,]) \\
= & \quad \{\text{ property of } \triangleright \text{ and } \triangleleft \ \} \\
& ds1to2\cdot c\cdot([\,] \triangleleft b) \\
= & \quad \{\text{ declaration 6.12 of } ds1to2 \text{ (twice) } \} \\
& [\,1-b\,] \triangleleft (1-c) \\
= & \quad \{\text{ property of } \triangleright \ \} \\
& (1-b) \triangleright [\,] \triangleleft (1-c)
\end{array}
$$

For the list $s \triangleleft d$, with $d \in \{0,1\}$, we derive:

$$
\begin{array}{ll}
& ds1to2\cdot c\cdot(b \triangleright (s \triangleleft d)) \\
= & \quad \{\text{ property of } \triangleright \text{ and } \triangleleft \ \} \\
& ds1to2\cdot c\cdot((b \triangleright s) \triangleleft d) \\
= & \quad \{\text{ declaration 6.12 of } ds1to2 \ \} \\
& ds1to2\cdot(1-d)\cdot(b \triangleright s) \triangleleft (1-c) \\
= & \quad \{\text{ proposition (6.12), from induction hypothesis } \} \\
& ((1-b) \triangleright s \triangleleft (1-(1-d))) \triangleleft (1-c) \\
= & \quad \{\ (1-(1-d)) = d, \text{ property of } \triangleright \text{ and } \triangleleft \ \} \\
& (1-b) \triangleright (s \triangleleft d) \triangleleft (1-c)
\end{array}
$$

Hence, we obtain the following alternative declaration for function *ds1to2*.

Declaration 6.13 For $s \in \mathcal{L}2$ and $b, c \in \{0,1\}$:

$$
\begin{array}{lcl}
ds1to2\cdot c\cdot[\,] & = & [\,c\,] \\
ds1to2\cdot c\cdot(b \triangleright s) & = & (1-b) \triangleright s \triangleleft (1-c)
\end{array}
$$

□

Evaluation of this declaration requires only $\mathcal{O}(1)$ steps.

Remark 6.14 In definition 2.13 of *v2s1*, each -1 is represented by a 0 and each 1 is represented by a 1. If the rôles of 0 and 1 were interchanged, we would have the following declaration for *v2s1*:

$$
\begin{array}{lcl}
v2s1\cdot[\,] & = & 0 \\
v2s1\cdot(s \triangleleft b) & = & 2*(v2s1\cdot s - b) + 1
\end{array}
$$

Then the expressions $t \triangleleft 0$ and $t \triangleleft 1$ in declaration 6.11 of *gdm2* would have been interchanged. Analogous to declaration 6.12 of *ds1to2*, we would have the following declaration for *ds1to2*:

$$
\begin{array}{lcl}
ds1to2\cdot c\cdot[\,] & = & [\,c\,] \\
ds1to2\cdot c\cdot(s \triangleleft b) & = & ds1to2\cdot b\cdot s \triangleleft (1-c)
\end{array}
$$

Analogous to conjecture (6), we would have obtained the following alternative declaration for *ds1to2*:

$$
\begin{array}{lcl}
ds1to2\cdot c\cdot[\,] & = & [\,c\,] \\
ds1to2\cdot c\cdot(b \triangleright s) & = & b \triangleright cmpl\cdot s \triangleleft (1-c)
\end{array}
$$

Evaluation of this declaration takes $\mathcal{O}(\#s)$ steps, but it can be implemented in hardware with $\mathcal{O}(1)$ propagation delay. □

Chapter 7

Hardware implementations

In this chapter we show how we can implement the most relevant functions of the previous chapters in hardware. We restrict ourselfs to combinatorial circuits, although it is possible to implement functions as sequential circuits as well. First, we provide a formalism as a basis for our hardware implementations. Then, we use this formalism to implement the desired functions in hardware.

7.1 A formalism

In this section we define hardware implementations for the basic elements of functions, of which all parameters and function values are bits, lists of bits, pairs of bits, pairs of lists of bits and lists of pairs of bits. With these implementations, we are able to implement all relevant functions of the previous chapters in hardware. To clarify the implementations, we also introduce a grafical notation that represents the implementations. We mean hardware implementation when we say implementation and grafical notation representing an implementation when we say representation.

We assume that a combinatorial circuit consists of wires and components only. Components are connected by wires on the in- and outputs. A wire carries at most one value at a time from the output of a component to the input of a component. This value is 0 or 1. We also assume $0 = \textit{false}$ and $1 = \textit{true}$ to be able to easily switch between arithmetic and boolean calculus.

7.1.1 Values

We implement a bit b by a wire containing the value of b and we represent this by an arrow, labeled with the name of the bit. We implement a list of bits s by a bundle of wires, where wire i contains the value of $s \cdot i$, with $0 \leq i < \#s$. We represent this bundle by a labeled arrow with a slash. We implement a pair of bits either by two separate wires or by a bundle of wires. We can represent this either by two labeled arrows or by a labeled arrow with a slash. Similarly, we implement a pair of binary lists either by two separate bundles of wires or by a bundle of bundles of wires. We can represent this either by two labeled arrows with a slash, or by a labeled arrow with two slashes. Finally we implement a list s of binary pairs by a bundle of bundles of wires, where bundle i contains the value of the pair $s \cdot i$, with $0 \leq i < \#s$. We represent this by a labeled arrow with two slashes.

7.1.2 Elementary expressions, operators and functions

We implement the expressions $b \wedge c$, $b \vee c$ and $b \not\equiv c$, for bits b, c, by a two-bits and-gate, a two-bits or-gate and a two-bits a xor-gate, respectively. We implement $\neg b$, i.e. $1 - b$, by an inverter. The representations of these expressions are given in figure 7.1.

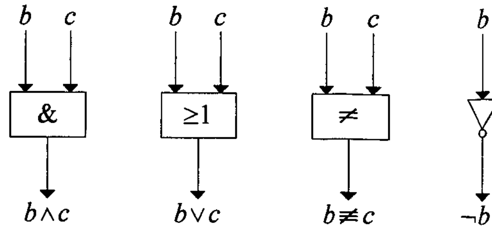

Figure 7.1: Implementations of $\wedge$, $\vee$, $\not\equiv$ and $\neg$

For list s and $0 \leq i < \#s$, we implement $s{\cdot}i$ by selecting the implementation of $s{\cdot}i$ from the implementation of s. We implement $b \triangleright s$ and $s \triangleleft b$, for list s and element b, by adding the implementation of b to the implementation of s such that the implementation of b becomes the most or the least significant element of the resulting implementation, respectively. We implement the inverse of $b \triangleright s$ and $s \triangleleft b$ by splitting off the most or the least significant implementation of b from the implementation of $b \triangleright s$ or $s \triangleleft b$, respectively. We implement $[\,e_0, ..., e_{n-1}\,]$, for elements e_i, with $0 \leq i < n$, by bundling the element implementations, such that element i in the bundle contains the value of e_i. We implement the inverse of this operation by splitting the bundle into the element implementations. Note that we can implement combinations of $\triangleright$, $\triangleleft$ and $[\,\cdot\,]$ in one step by bundling all parameter implementations at the same time. We can implement the inverse of these combinations by splitting the bundle at the same time into the parameter implementations. In figure 7.2, we give representations for these implementations, for bits b, c, binary list s and $0 \leq i < \#s$.

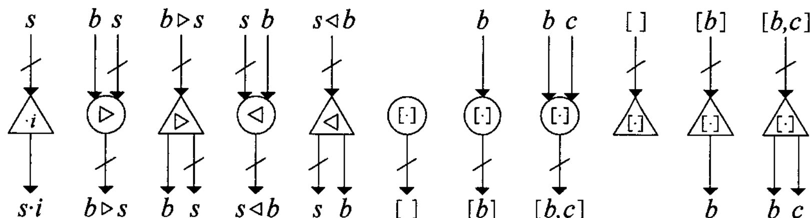

Figure 7.2: Implementations of $s{\cdot}i$, $b \triangleright s$, $s \triangleleft b$, $[\,]$, $[\,b\,]$ and $[\,b, c\,]$

We can implement pairing of two elements by bundling the element implementations. We can implement the inverse of this operation by splitting the bundle into the element implementations. In figure 7.3, we give representations for these implementations, for bits b, c and binary lists s, t.

Values are sometimes used more than once. We implement this by duplicating the implementation of the value. In figure 7.3, we give representations for the duplication of bit b and binary list s. To keep representations clear, we may leave out the duplication of frequently used values. To avoid ambiguity, we have to explicitly label every use of such values.

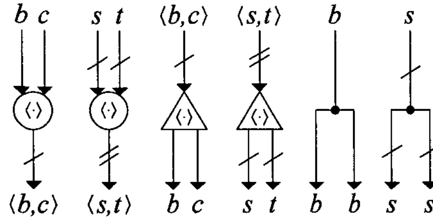

Figure 7.3: Implementations of pairing and duplication

For the implementation of conditional expressions, we introduce function *sel*, of type $\{0,1\} \to \mathcal{L}2 \to \mathcal{L}2 \to \mathcal{L}2$, and *selb*, of type $\{0,1\} \to \{0,1\} \to \{0,1\} \to \{0,1\}$, with the following definitions.

Definition 7.1 For $b, c, d \in \{0,1\}$ and $s, t \in \mathcal{L}2$:

$$\begin{aligned} sel\cdot b\cdot s\cdot t &= \textbf{if } b = 0 \to s \;[]\; b = 1 \to t \textbf{ fi} \\ selb\cdot b\cdot c\cdot d &= \textbf{if } b = 0 \to c \;[]\; b = 1 \to d \textbf{ fi} \end{aligned}$$

□

We implement these functions by a so-called multiplexer. This is represented in figure 7.4.

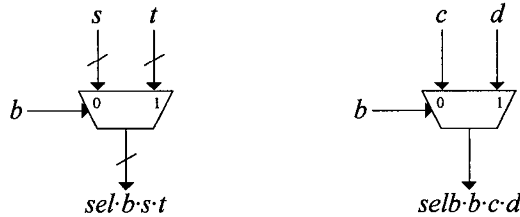

Figure 7.4: Implementations of *sel* and *selb*

7.1.3 Function applications, function definitions, and recursion

We implement a function application by implementing its defining expression and connecting the implementations of its parameters and function value to the implementation of the defining expression. In figure 7.5, we show possible representations of the implementation of an application $f\cdot b\cdot s$ of function f, of type $\{0,1\} \to \mathcal{L}2 \to \langle \mathcal{L}2, \mathcal{L}2 \rangle$.

We implement a function's defining expression by implementing all elementary expressions, operators and functions, and by implementing all remaining function applications. After that, we implement all occurring parameters and connect these at the right way to the implementations of the expressions, operators and functions.

Example 7.2 We introduce function g, of type $\{0,1\} \to \mathcal{L}2 \to \mathcal{L}2$, with the following definition:

$$g\cdot b\cdot s = (b \triangleright s) \triangleleft b$$

We can implement this definition directly into hardware. Figure 7.6 shows the representation of this implementation. □

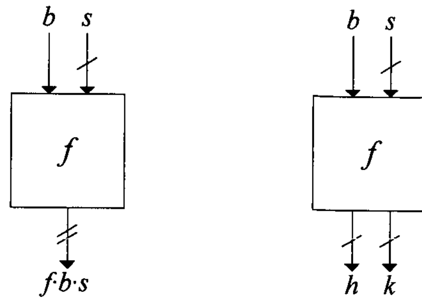

Figure 7.5: Implementations of $f \cdot b \cdot s$, with $\langle h, k \rangle = f \cdot b \cdot s$

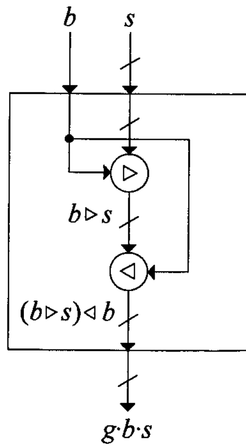

Figure 7.6: Implementation of g

We can implement a function with case distinction on its parameters by implementing all cases separately. We can implement a linearly recursive function by implementing the base cases and the inductive cases separately.

As an example, we repeat definition 3.1 of function *cmpl*, of type $\mathcal{L}2 \to \mathcal{L}2$, with bit b and binary list s:

$$\begin{array}{lcl} cmpl \cdot [\,] & = & [\,] \\ cmpl \cdot (s \lhd b) & = & cmpl \cdot s \lhd (1 - b) \end{array}$$

This definition can be implemented directly. Figure 7.7 shows the representation of the base case and the inductive case of this definition. Note that after unfolding the inductive case, we can see that function *cmpl* is simply a row of $\#s$ inverters, with $\mathcal{O}(1)$ propagation delay.

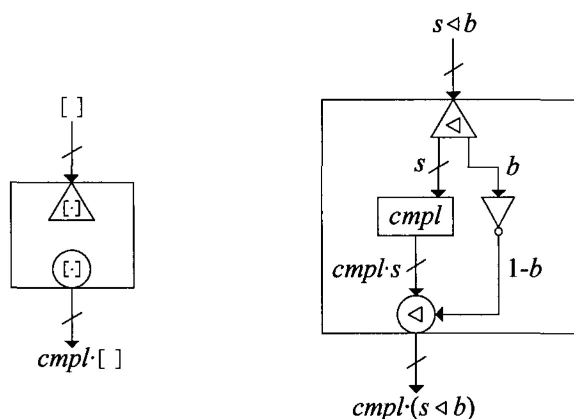

Figure 7.7: Implementation of *cmpl*

7.1.4 Frequently used expressions

We now give implementations and representations for a number of expressions, that are used frequently in the next section.

Inverters

We can simplify the representation of $1 - b$ and $cmpl \cdot s$, for bit b and binary list s. For this purpose, we can remove the symbol ▽ and insert the symbol ∘ between the output of a component providing b or s, and the implementation of b or s, respectively. Similarly, we can remove the symbol ▽ and insert the symbol ∘ between the implementations of b and s and the input of a component that needs $1 - b$ or $cmpl \cdot s$, respectively.

Multiplexers

We can simplify the implementation of $selb \cdot b \cdot c \cdot d$ for some cases of c and d. We have $selb \cdot b \cdot c \cdot d \;=\; (\neg b \Rightarrow c) \wedge (b \Rightarrow d)$. From the rules of implication and double negation, this is $selb \cdot b \cdot c \cdot d = (b \vee c) \wedge (\neg b \vee d)$. Using predicate calculus, we can simplify this property for some special cases of c and d as follows.

Property 7.3 For $b, c, d \in \{0,1\}$, we have:

$$\begin{array}{ll} selb\cdot b\cdot b\cdot d & = b \wedge d \\ selb\cdot b\cdot c\cdot b & = b \vee c \\ selb\cdot b\cdot c\cdot (1-c) & = b \not\equiv c \\ selb\cdot b\cdot (1-d)\cdot d & = \neg(b \not\equiv d) \end{array}$$

□

Half adders and full adders

We now construct implementations for $(b+c)$ **div** 2, $(b+c)$ **mod** 2, $(b+c+d)$ **div** 2 and $(b+c+d)$ **mod** 2 for bits b, c and d. For this purpose, we introduce function ha, of type $\{0,1\} \rightarrow \{0,1\} \rightarrow \langle \{0,1\}, \{0,1\} \rangle$, and function fa, of type $\{0,1\} \rightarrow \{0,1\} \rightarrow \{0,1\} \rightarrow \langle \{0,1\}, \{0,1\} \rangle$, with the following definitions.

Definition 7.4 For $b, c, d \in \{0,1\}$, we have:

$$\begin{array}{ll} ha\cdot b\cdot c & = \langle (b+c) \textbf{ div } 2, (b+c) \textbf{ mod } 2 \rangle \\ fa\cdot b\cdot c\cdot d & = \langle (b+c+d) \textbf{ div } 2, (b+c+d) \textbf{ mod } 2 \rangle \end{array}$$

□

We can implement the expressions $(b+c)$ **div** 2 and $(b+c)$ **mod** 2 by a two bits and-gate and a two bits xor-gate respectively. Hence, we can implement function ha directly in hardware (see figure 7.8).

To construct an implementation for function fa, we derive as follows:

$$\begin{array}{ll} & fa\cdot b\cdot c\cdot d \\ = & \quad \{ \text{ definition 7.4 of } fa \ \} \\ & \langle (b+c+d) \textbf{ div } 2, (b+c+d) \textbf{ mod } 2 \rangle \\ = & \quad \{ \text{ definition of } \textbf{div} \text{ and } \textbf{mod}, \text{ property of } \textbf{div} \text{ and } \textbf{mod} \ \} \\ & \langle (b+c) \textbf{ div } 2 + ((b+c) \textbf{ mod } 2 + d) \textbf{ div } 2, ((b+c) \textbf{ mod } 2 + d) \textbf{ mod } 2 \rangle \\ = & \quad \{ \langle h, k \rangle = ha\cdot b\cdot c \ \} \\ & \langle h + (k+d) \textbf{ div } 2, (k+d) \textbf{ mod } 2 \rangle \\ = & \quad \{ \langle l, m \rangle = ha\cdot k\cdot d \ \} \\ & \langle h+l, m \rangle \end{array}$$

Hence, we have the following the following property for function fa:

$$fa\cdot b\cdot c\cdot d = \langle h+l, m \rangle \textbf{ whr } \langle h, k \rangle = ha\cdot b\cdot c \ \& \ \langle l, m \rangle = ha\cdot k\cdot d \textbf{ end}$$

Because $h + l \leq 1$, we can implement $h + l$ by a two-bits or-gate. In figure 7.8 we give representations of the implementations of functions ha and fa. These implementations are the standard implementations of a *half adder* and a *full adder*, respectively.

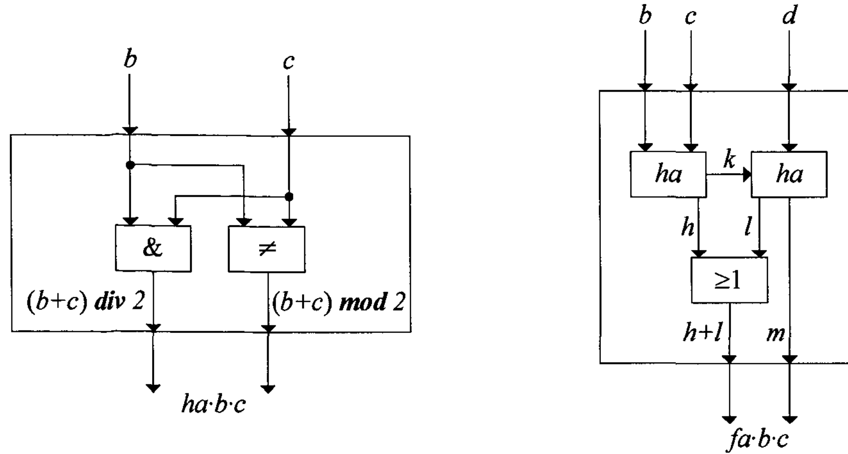

Figure 7.8: Implementations of *ha* and *fa*

7.2 Arithmetic operations

In this section we give hardware implementations for all relevant functions of the previous chapters. We do this by rewriting the declaration of each relevant function, such that it can be implemented using the formalism of the previous section. Also we have to give hardware implementations for all functions occurring in the declaration that are not implemented yet. We will also give representations of most of the implementations.

To increase efficiency of implementations, we keep the following guidelines in mind when we rewrite the declarations:

- minimize case distinction that is not addressed by property 7.3 of *selb*;
- maximize the re-use of expressions, by naming multiply used expressions.

7.2.1 Simple arithmetic operations

We only consider the functions *inc*, *dec* and *neg* of chapter 3. Functions *hlvq* and *hlvr* are trivial.

Increment

We repeat declaration 3.4 of *inc*, for bits b, c and non-empty binary list s:

$$\begin{aligned} inc{\cdot}c{\cdot}[\,b\,] &= [\,c - h \textbf{ div } 2, h \textbf{ mod } 2\,] \quad && \textbf{whr } h = b + c \textbf{ end} \\ inc{\cdot}c{\cdot}(s \triangleleft b) &= inc{\cdot}(h \textbf{ div } 2){\cdot}s \triangleleft h \textbf{ mod } 2 && \textbf{whr } h = b + c \textbf{ end} \end{aligned}$$

We obtain an implementation for $c - h$ **div** 2 by case distinction on c. If $c = 0$, then c suffices, from h **div** $2 = 0$. If $c = 1$, then $1 - h$ **div** 2 suffices. Using definition 7.1 of *selb* and definition 7.4 of *ha*, we obtain for *inc*:

$$\begin{aligned} inc{\cdot}c{\cdot}[\,b\,] &= [\,selb{\cdot}c{\cdot}c{\cdot}(1-k), l\,] && \textbf{whr } \langle\, k, l\,\rangle = ha{\cdot}b{\cdot}c \textbf{ end} \\ inc{\cdot}c{\cdot}(s \triangleleft b) &= inc{\cdot}k{\cdot}s \triangleleft l && \textbf{whr } \langle\, k, l\,\rangle = ha{\cdot}b{\cdot}c \textbf{ end} \end{aligned}$$

We can implement this immediately. From property 7.3 of *selb*, we can implement the expression $selb \cdot c \cdot c \cdot (1 - k)$ with a two-bits and-gate and an inverter. In figure 7.9 we give a representation of this implementation of *inc*.

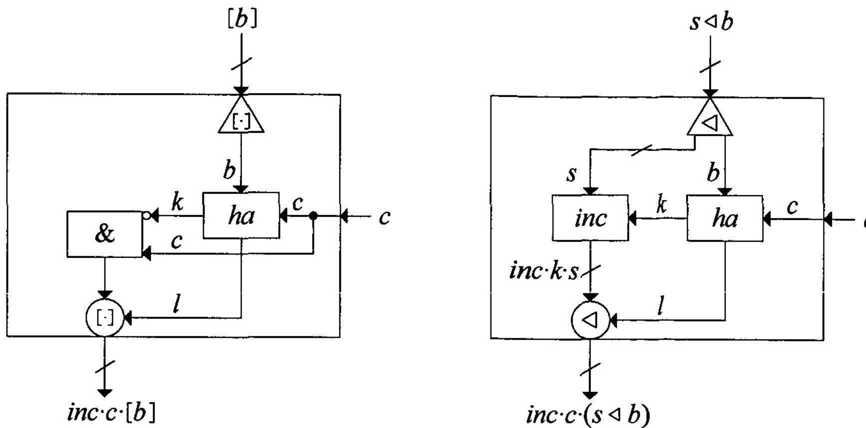

Figure 7.9: Implementation of *inc*

Note that this implementation essentially is a row of $\#s$ half adders, with $n = \#s$. This can be seen from figure 7.10, where we have completely unfolded the recursive declaration.

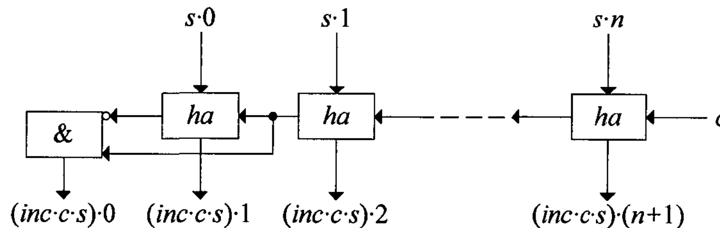

Figure 7.10: Unfolding of *inc*

Decrement

The construction of an implementation of declaration 3.6 of *dec* is analogous to the construction of an implementation of declaration 3.4 of *inc*. Therefore we only consider declaration 3.7 of *dec*. We repeat this declaration, for bit c and non-empty binary list s:

$$dec \cdot c \cdot s = cmpl \cdot (inc \cdot c \cdot (cmpl \cdot s))$$

We can implement this declaration immediately. In figure 7.11 we give a representation of this implementation.

Negation

We repeat declaration 3.9 of function *neg*, for non-empty binary list s:

$$neg \cdot s = inc \cdot 1 \cdot (cmpl \cdot s)$$

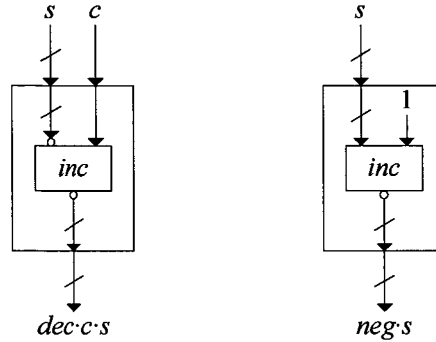

Figure 7.11: Implementation of *dec* and *neg*

We can implement this declaration immediately. In figure 7.11 we give a representation of this implementation.

7.2.2 Addition and subtraction

We consider all functions of chapter 4 concerning binary lists and lists of binary pairs of equal length.

Binary addition

We repeat declaration 4.8 of function *add2*, for bits b, c and non-empty binary lists s, t of equal length:

$$\begin{aligned} add2\cdot[\,b\,]\cdot[\,c\,] &= [\,h \;\mathbf{div}\; 2 + h \;\mathbf{mod}\; 2, h \;\mathbf{mod}\; 2\,] \\ &\quad \mathbf{whr}\; h = b + c \;\mathbf{end} \\ add2\cdot(s \triangleleft b)\cdot(t \triangleleft c) &= inc\cdot(h \;\mathbf{div}\; 2)\cdot(add2\cdot s\cdot t) \triangleleft h \;\mathbf{mod}\; 2 \\ &\quad \mathbf{whr}\; h = b + c \;\mathbf{end} \end{aligned}$$

We obtain an implementation for h **div** $2 + h$ **mod** 2 by case distinction on h **mod** 2. If h **mod** $2 = 0$, then h **div** 2 suffices. If h **mod** $2 = 1$, then h **mod** 2 suffices, from h **div** $2 = 0$. Using definition 7.1 of *selb* and definition 7.4 of *ha*, we obtain for *add2*:

$$\begin{aligned} add2\cdot[\,b\,]\cdot[\,c\,] &= [\,selb\cdot l\cdot k\cdot l, l\,] \qquad \mathbf{whr}\; \langle\, k, l\,\rangle = ha\cdot b\cdot c \;\mathbf{end} \\ add2\cdot(s \triangleleft b)\cdot(t \triangleleft c) &= inc\cdot k\cdot(add2\cdot s\cdot t) \triangleleft l \;\; \mathbf{whr}\; \langle\, k, l\,\rangle = ha\cdot b\cdot c \;\mathbf{end} \end{aligned}$$

We can implement this immediately. The expression $selb\cdot l\cdot k\cdot l$ can be implemented with a two-bits or-gate, from property 7.3 of *selb*. In figure 7.12 we give a representation of the implementation of *add2*.

We repeat declaration 4.10 of function *adc2*, for bits b, c and non-empty binary lists s, t of equal length:

$$\begin{aligned} adc2\cdot d\cdot[\,b\,]\cdot[\,c\,] &= [\,b + c - h \;\mathbf{div}\; 2, h \;\mathbf{mod}\; 2\,] \\ &\quad \mathbf{whr}\; h = b + c + d \;\mathbf{end} \\ adc2\cdot d\cdot(s \triangleleft b)\cdot(t \triangleleft c) &= adc2\cdot(h \;\mathbf{div}\; 2)\cdot s\cdot t \triangleleft h \;\mathbf{mod}\; 2 \\ &\quad \mathbf{whr}\; h = b + c + d \;\mathbf{end} \end{aligned}$$

For the implementation of the expression $b + c - h$ **div** 2, we have $b + c - h$ **div** $2 = 2 * (b + c)$ **div** $2 + (b + c)$ **mod** $2 - h$ **div** 2, from the definition of **div** and **mod**. If

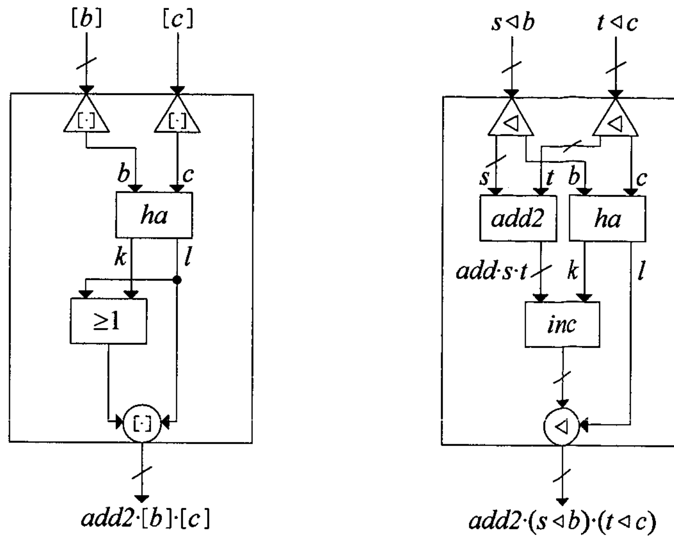

Figure 7.12: Implementation of *add2*

$(b+c)$ **mod** $2=0$, we may simplify this to h **div** 2, from $(b+c)$ **div** $2=h$ **div** 2. If $(b+c)$ **mod** $2=1$, we may simplify this to $1-h$ **div** 2, from $(b+c)$ **div** $2=0$. Using definition 7.1 of *selb* and definition 7.4 of *fa*, we obtain for *adc2*:

$$\begin{aligned}
adc2\cdot d\cdot[\,b\,]\cdot[\,c\,] &= [\,selb\cdot((b+c)\ \mathbf{mod}\ 2)\cdot k\cdot(1-k),l\,]\\
&\quad\ \mathbf{whr}\ \langle\, k,l\,\rangle = fa\cdot b\cdot c\cdot d\ \mathbf{end}\\
adc2\cdot d\cdot(s\lhd b)\cdot(t\lhd c) &= adc2\cdot k\cdot s\cdot t\lhd l\\
&\quad\ \mathbf{whr}\ \langle\, k,l\,\rangle = fa\cdot b\cdot c\cdot d\ \mathbf{end}
\end{aligned}$$

This can be implemented directly into hardware. We can implement the expression $selb\cdot((b+c)\ \mathbf{mod}\ 2)\cdot k\cdot(1-k)$ with two two-bits xor-gates, from property 7.3 of *selb*. In figure 7.13 we give a representation of this implementation. Note that the implementation of *adc2* essentially is a row of $\#s$ full adders.

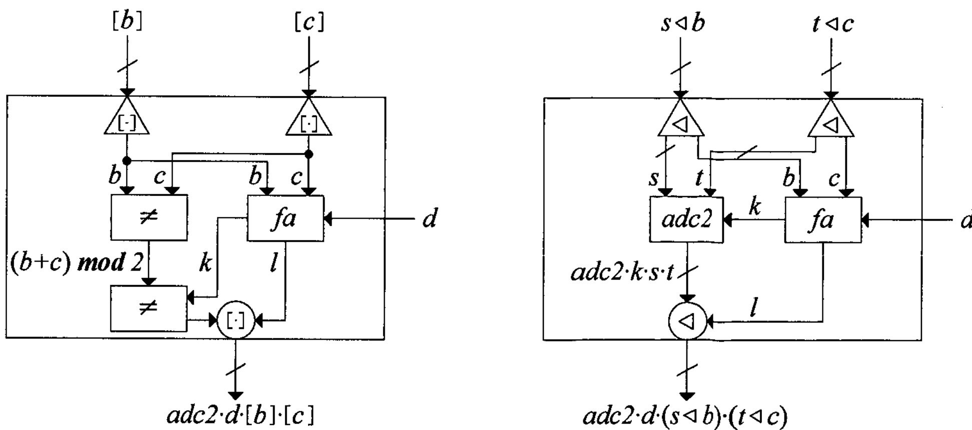

Figure 7.13: Implementation of *adc2*

We repeat declaration 4.9 of *add2* and declaration 4.25 of *subt2*, for non-empty binary lists s, t:

$$add2 \cdot s \cdot t = adc2 \cdot 0 \cdot s \cdot t$$
$$subt2 \cdot s \cdot t = adc2 \cdot 1 \cdot s \cdot (cmpl \cdot t)$$

We can implement these declarations directly. In figure 7.14 we give representations of these implementations.

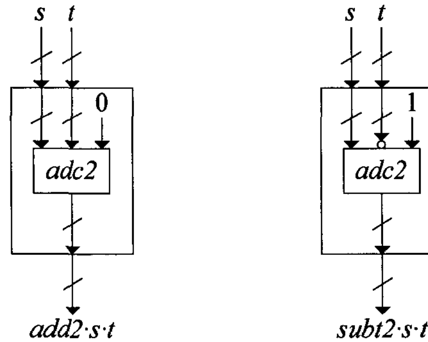

Figure 7.14: Implementations of *add2* and *subt2*

Carry-save addition

We repeat declaration 4.21 of *adc2p*, for bits b, c, d, e and non-empty binary paired lists s, t of equal length:

$$\begin{aligned} adc2p \cdot e \cdot [\langle b,c \rangle] \cdot [d] &= [\langle h \textbf{ div } 2, h \textbf{ mod } 2 \rangle, \langle h \textbf{ mod } 2, e \rangle] \\ &\quad \textbf{whr } h = b + c + d \textbf{ end} \\ adc2p \cdot e \cdot (s \triangleleft \langle b,c \rangle) \cdot (t \triangleleft d) &= adc2p \cdot (h \textbf{ div } 2) \cdot s \cdot t \triangleleft \langle h \textbf{ mod } 2, e \rangle \\ &\quad \textbf{whr } h = b + c + d \textbf{ end} \end{aligned}$$

Using declaration 7.4 of *fa*, we obtain:

$$\begin{aligned} adc2p \cdot e \cdot [\langle b,c \rangle] \cdot [d] &= [\langle k,l \rangle, \langle l,e \rangle] \\ &\quad \textbf{whr } \langle k,l \rangle = fa \cdot b \cdot c \cdot d \textbf{ end} \\ adc2p \cdot e \cdot (s \triangleleft \langle b,c \rangle) \cdot (t \triangleleft d) &= adc2p \cdot k \cdot s \cdot t \triangleleft \langle l,e \rangle \\ &\quad \textbf{whr } \langle k,l \rangle = fa \cdot b \cdot c \cdot d \textbf{ end} \end{aligned}$$

We can implement this directly with $\mathcal{O}(1)$ propagation delay. In figure 7.15 we give a representation of this implementation.

We repeat declaration 4.20 of *add2p* and declaration 4.29 of *subt2p*, for non-empty binary paired lists s, t:

$$add2p \cdot s \cdot t = adc2p \cdot 0 \cdot s \cdot t$$
$$subt2p \cdot s \cdot t = adc2p \cdot 1 \cdot s \cdot (cmpl \cdot t)$$

The implementations of these declarations are analogous to the implementations of declaration 4.9 of *add2* and declaration 4.25 of *subt2*.

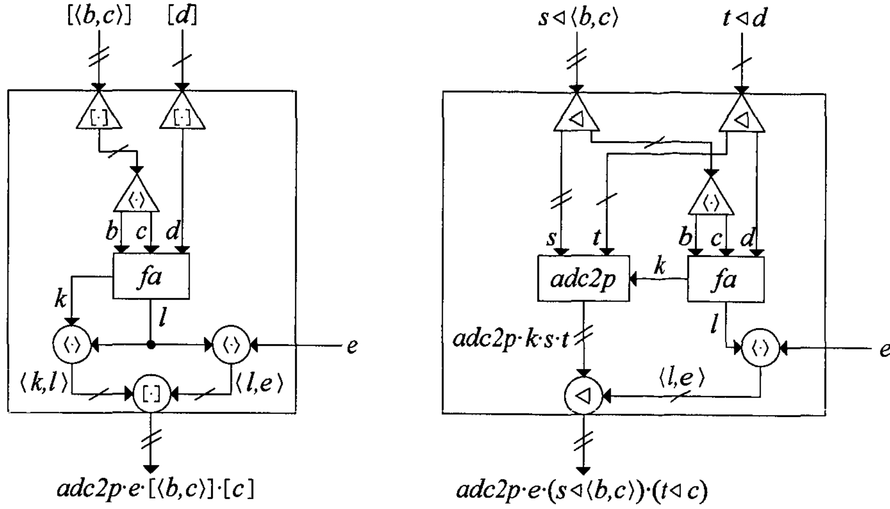

Figure 7.15: Implementation of *adc2p*

Finally, we repeat declaration 4.23 of *c2pto2*, for bits b, c, d and non-empty binary paired list s:

$$\begin{aligned}
c2pto2 \cdot d \cdot [\langle b,c\rangle] &= [b+c-h \ \mathbf{div}\ 2, h \ \mathbf{mod}\ 2] \\
&\quad \mathbf{whr}\ h = b+c+d\ \mathbf{end} \\
c2pto2 \cdot d \cdot (s \triangleleft \langle b,c\rangle) &= c2pto2 \cdot (h \ \mathbf{div}\ 2) \cdot s \triangleleft h \ \mathbf{mod}\ 2 \\
&\quad \mathbf{whr}\ h = b+c+d\ \mathbf{end}
\end{aligned}$$

Except for the bundling of wires, the implementation of this declaration is equal to the implementation of declaration 4.10 of *adc2*.

7.2.3 Multiplication

We give implementations for binary multiplication, multiplication with carry-save addition and Booth multiplication.

Binary multiplication

We repeat declaration 5.9 of *mul2*, declaration 5.11 of *addbmul2* and *subtbmul2* and declaration 5.13 of *f2*, for bit b and non-empty binary lists s, t:

$$\begin{aligned}
mul2 \cdot s \cdot t &= gmul2 \cdot [0] \cdot s \cdot t \\
addbmul2 \cdot s \cdot 0 \cdot t &= s \\
addbmul2 \cdot s \cdot 1 \cdot t &= add2 \cdot s \cdot t \\
subtbmul2 \cdot s \cdot 0 \cdot t &= s \\
subtbmul2 \cdot s \cdot 1 \cdot t &= subt2 \cdot s \cdot t \\
f2 \cdot [b] &= [b,b] \\
f2 \cdot (s \triangleleft b) &= s \triangleleft b
\end{aligned}$$

We can implement the declarations of *mul2* and *f2* immediately. After elimination of the parameter patterns and application of definition 7.1 of *sel*, we obtain for *addbmul2*

and *subtbmul2*:

$$addbmul2\cdot s\cdot b\cdot t \;=\; sel\cdot b\cdot s\cdot(add2\cdot s\cdot t)$$
$$subtbmul2\cdot s\cdot b\cdot t \;=\; sel\cdot b\cdot s\cdot(subt2\cdot s\cdot t)$$

We can implement this directly. From specification 4.7 of *add2* and specification 4.24 of *subt2*, we can also obtain, with u a list of $\#t$ 0's:

$$addbmul2\cdot s\cdot b\cdot t \;=\; add2\cdot s\cdot(sel\cdot b\cdot u\cdot t)$$
$$subtbmul2\cdot s\cdot b\cdot t \;=\; subt2\cdot s\cdot(sel\cdot b\cdot u\cdot t)$$

Now we may replace the multiplexer in the implementation by a row of $\#t$ two bits and-gates, where and-gate i, with $0 \le i < \#t$, has the values of b and $t\cdot i$ on its inputs.

We repeat declaration 5.14 of *gmul2*, for bit b and non-empty binary lists s, t, u:

$$gmul2\cdot u\cdot s\cdot[\,b\,] \;=\; subtbmul2\cdot u\cdot b\cdot s$$
$$gmul2\cdot u\cdot s\cdot(t \lhd b) \;=\; gmul2\cdot v\cdot s\cdot t \lhd c \quad \textbf{whr}\ v \lhd c = f2\cdot(addbmul2\cdot u\cdot b\cdot s)\ \textbf{end}$$

We can implement this directly. In figure 7.16 we give a representation of this implementation.

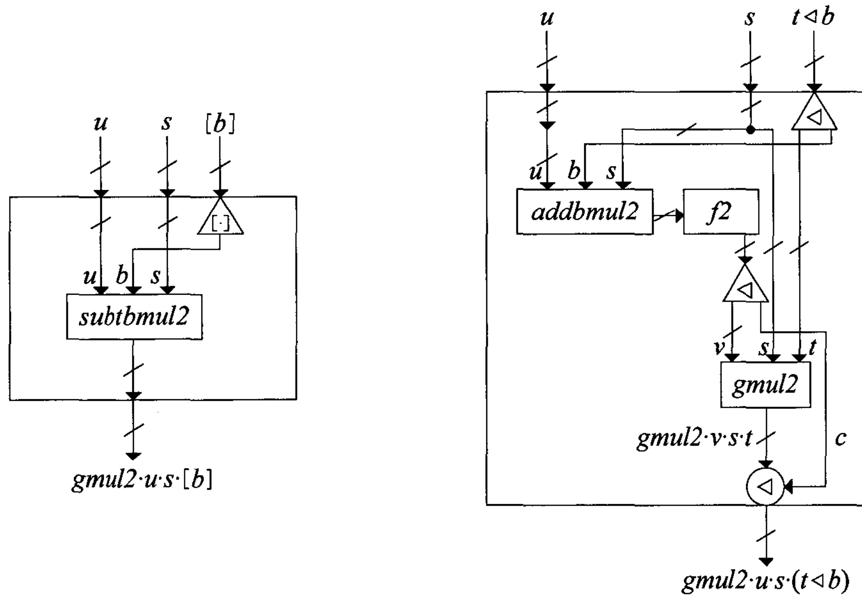

Figure 7.16: Implementation of *gmul2*

Multiplication with carry-save addition

We repeat declaration 5.19 of *mul2*, for non-empty binary lists s, t:

$$mul2\cdot s\cdot t = c2pto2\cdot(gmul2p\cdot[\,\langle\,0,0\,\rangle\,]\cdot s\cdot t)$$

We can implement this immediately.

The construction of implementations for declarations 5.17, 5.22 and 5.20 of functions *addbmul2p*, *subtbmul2p*, *f2p* and *gmul2p* is analogous to the construction of the above implementations for declarations 5.11, 5.13 and 5.14 of functions *addbmul2*, *subtbmul2*, *f2* and *gmul2*.

Booth multiplication

We repeat declaration 5.26 of *gmul2* and 5.28 of *asbmul2*, for bit c and non-empty binary lists s, t, u:

$$\begin{aligned} gmul2 \cdot u \cdot s \cdot t &= gmulbr2 \cdot u \cdot s \cdot 0 \cdot t \\ asbmul2 \cdot s \cdot 0 \cdot c \cdot t &= subtbmul2 \cdot s \cdot c \cdot t \\ asbmul2 \cdot s \cdot 1 \cdot c \cdot t &= addbmul2 \cdot s \cdot (1 - c) \cdot t \end{aligned}$$

We can implement the declaration of *gmul2* immediately. After elimination of the parameter patterns and application of definition 7.1 of *sel*, we obtain for *asbmul2*, for bit b:

$$asbmul2 \cdot s \cdot b \cdot c \cdot t = sel \cdot b \cdot (subtbmul2 \cdot s \cdot c \cdot t) \cdot (addbmul2 \cdot s \cdot (1 - c) \cdot t)$$

We can implement this directly.

We repeat declaration 5.29 of *gmulbr2*, for bits b, c and non-empty binary lists s, t, u:

$$\begin{aligned} gmulbr2 \cdot u \cdot s \cdot c \cdot [\,b\,] &= asbmul2 \cdot u \cdot c \cdot b \cdot s \\ gmulbr2 \cdot u \cdot s \cdot c \cdot (t \triangleleft b) &= gmulbr2 \cdot v \cdot s \cdot b \cdot t \triangleleft d \\ & \ \textbf{whr } v \triangleleft d = f2 \cdot (asbmul2 \cdot u \cdot c \cdot b \cdot s) \textbf{ end} \end{aligned}$$

We can implement this directly. In figure 7.17 we give a representation of this implementation.

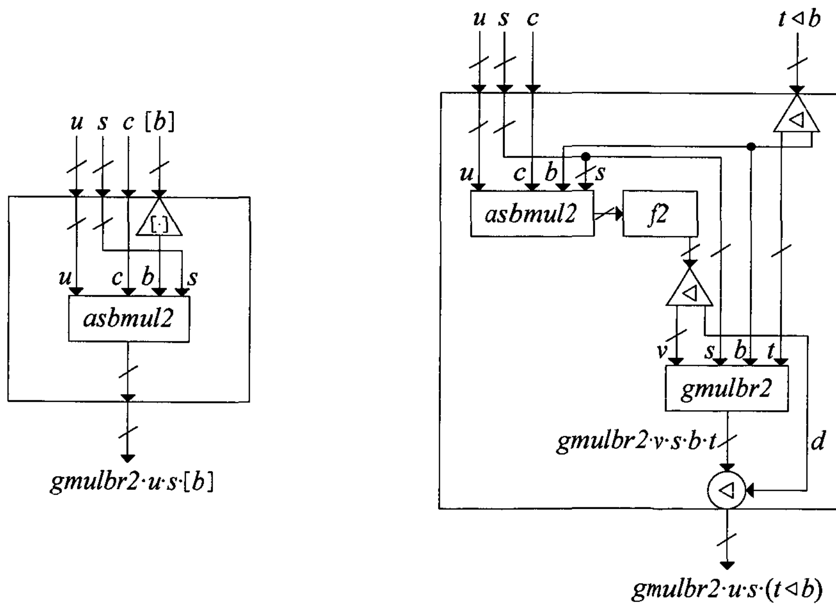

Figure 7.17: Implementation of *gmulbr2*

7.2.4 Division

We give hardware implementations for restoring and non-restoring division, for which we assume constant $B2 \in \mathcal{L}2$, with $vn2 \cdot B2 > 0$.

Restoring division

We repeat declaration 6.7 of *dm2*, for bit b and non-empty binary list s:

$$
\begin{array}{lll}
dm2\cdot[0] & = & \langle [0],[0] \rangle \\
dm2\cdot[1] & = & \langle [1], dec\cdot 1\cdot B2 \rangle \\
dm2\cdot(s \lhd b) & = & \textbf{if } w\cdot 0 = 1 \rightarrow \langle t \lhd 0, v \rangle \\
 & & [\,]\ \ w\cdot 0 = 0 \rightarrow \langle t \lhd 1, w \rangle \\
 & & \textbf{fi whr } \langle t,u \rangle = dm2\cdot s \ \&\ v = u \lhd b \ \&\ w = subt2\cdot v\cdot B2 \textbf{ end}
\end{array}
$$

Minimizing case distinction and maximum the re-use of expressions, we obtain, using definition 7.1 of *sel*:

$$
\begin{array}{lll}
dm2\cdot[b] & = & \langle t, sel\cdot b\cdot t\cdot(dec\cdot b\cdot B2) \rangle \ \textbf{whr } t = [b] \textbf{ end} \\
dm2\cdot(s \lhd b) & = & \langle t \lhd c, sel\cdot c\cdot v\cdot w \rangle \qquad \textbf{whr } \langle t,u \rangle = dm2\cdot s \\
 & & \&\ v = u \lhd b \ \&\ w = subt2\cdot v\cdot B2 \ \&\ c = 1 - w\cdot 0 \textbf{ end}
\end{array}
$$

We can implement this directly. In figure 7.18 we give a representation of this implementation.

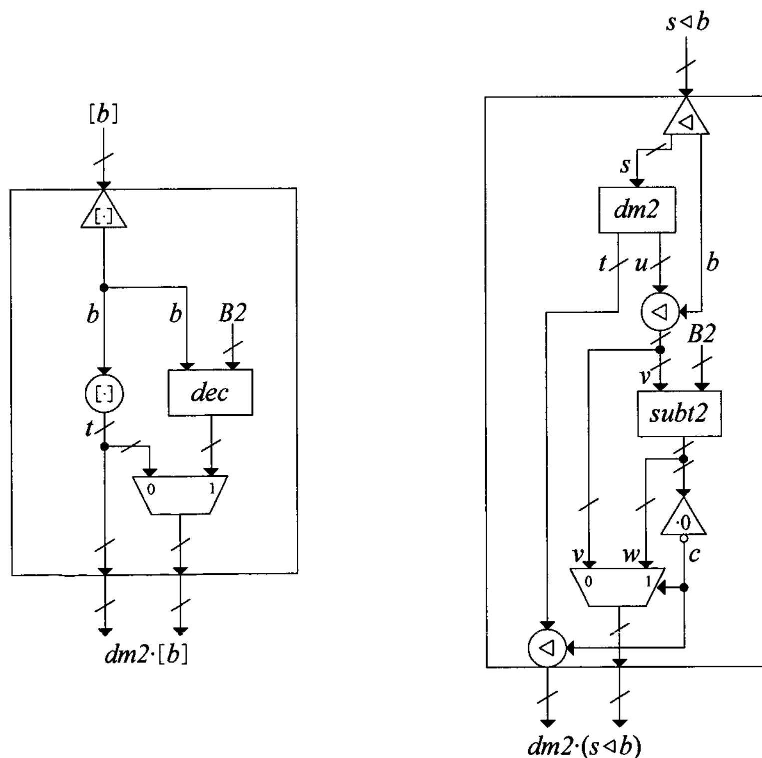

Figure 7.18: Restoring implementation of *dm2*

Non-restoring division

We repeat declaration 6.10 of *dm2* and declaration 6.13 of *ds1to2*, for bits b, c and non-empty binary list s:

$$\begin{aligned}
dm2\cdot s &= \textbf{if } u\cdot 0 = 1 \rightarrow \langle\, ds1to2\cdot 1\cdot t, add2\cdot u\cdot B2\,\rangle \\
&\quad [\!]\ u\cdot 0 = 0 \rightarrow \langle\, ds1to2\cdot 0\cdot t, u\,\rangle \\
&\quad \textbf{fi whr } \langle\, t, u\,\rangle = gdm2\cdot s \textbf{ end} \\
ds1to2\cdot c\cdot[\,] &= [\,c\,] \\
ds1to2\cdot c\cdot(b \rhd s) &= (1-b) \rhd s \lhd (1-c)
\end{aligned}$$

We can implement the declaration of *ds1to2* directly, using only two inverters. Minimizing case distinction and maximum the re-use of expressions, we obtain for *dm2*, using definition 7.1 of *sel*:

$$\begin{aligned}
dm2\cdot s = {} & \langle\, ds1to2\cdot c\cdot t, sel\cdot c\cdot u\cdot(add2\cdot u\cdot B2)\,\rangle \\
& \textbf{whr } \langle\, t, u\,\rangle = gdm2\cdot s \ \&\ c = u\cdot 0 \textbf{ end}
\end{aligned}$$

We can implement this directly. In figure 7.19 we give a representation of this implementation.

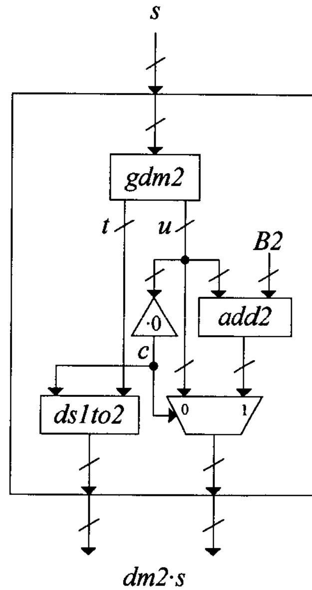

Figure 7.19: Non-restoring implementation of *dm2*

Finally we repeat declaration 6.11 of *gdm2*, for bit b and non-empty binary list s:

$$\begin{aligned}
gdm2\cdot[\,b\,] &= \langle\,[\,], [\,b\,]\,\rangle \\
gdm2\cdot(s \lhd b) &= \textbf{if } v\cdot 0 = 1 \rightarrow \langle\, t \lhd 0, add2\cdot v\cdot B2\,\rangle \\
&\quad [\!]\ v\cdot 0 = 0 \rightarrow \langle\, t \lhd 1, subt2\cdot v\cdot B2\,\rangle \\
&\quad \textbf{fi whr } \langle\, t, u\,\rangle = gdm2\cdot s \ \&\ v = u \lhd b \textbf{ end}
\end{aligned}$$

Minimizing case distinction and maximum the re-use of expressions, we obtain, using definition 7.1 of *sel*, declaration 4.9 of *add2* and declaration 4.25 of *subt2*:

$$\begin{aligned}
gdm2\cdot[\,b\,] &= \langle\,[\,], [\,b\,]\,\rangle \\
gdm2\cdot(s \lhd b) &= \langle\, t \lhd c, adc2\cdot c\cdot v\cdot(sel\cdot c\cdot B2\cdot(cmpl\cdot B2))\,\rangle \\
&\quad \textbf{whr } \langle\, t, u\,\rangle = gdm2\cdot s \ \&\ v = u \lhd b \ \&\ c = 1 - v\cdot 0 \textbf{ end}
\end{aligned}$$

We can implement this directly. In figure 7.20 we give a representation of this implementation.

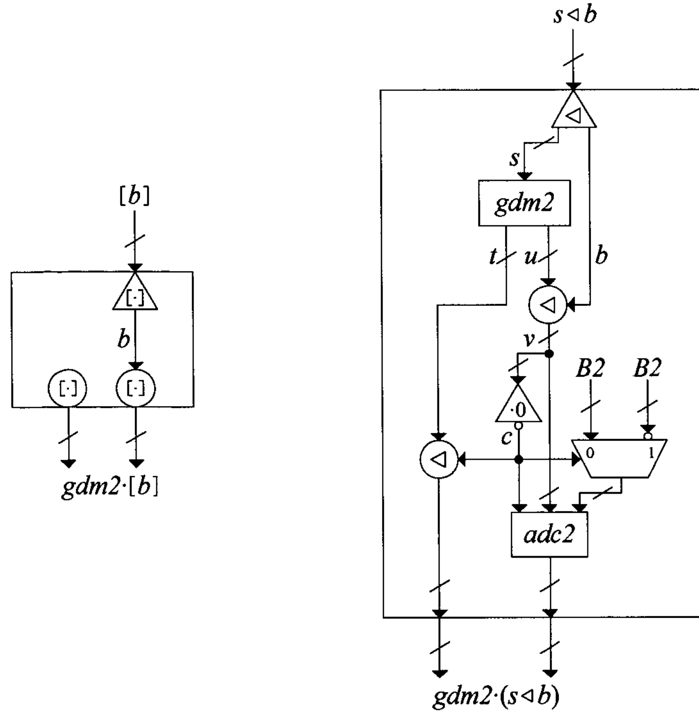

Figure 7.20: Implementation of *gdm2*

Chapter 8

Conclusions and recommendations

In the previous sections, we have constructed combinatorial circuits that perform binary arithmetic. To achieve this, we distinguished three levels of reasoning, i.e. the arithmetic level, the representation level and the implementation level. We derived declarations at the arithmetic and the representation level, and we used these declarations, together with a formalism, to implement combinatorial circuits at the implementation level. This thesis has shown that this approach provides for a good separation of concerns, in which only essential properties of the lower levels have to be taken into account at the higher levels. Due to this, it is possible to use the declarations of this thesis for other purposes than combinatorial circuits. Besides implementations in existing functional programming languages, they can be used for non-functional implementations such as sequential programs, sequential circuits and probably rewrite rules for μCRL. Moreover, I am convinced that this approach can be effectively applied to more complex arithmetic operations, different integer representations and the extension to floating point numbers.

In the past, a number of efforts were made to formally derive implementations for arithmetic operations using a calculational style of functional programming. Hutton et al. ([Hut]) have derived a declaration for the increment of a binary paired natural by a bit. Kloos et al. ([Klo 1], [Klo 2] and [Klo 3]) have derived declarations for carry-propagate addition, carry look-ahead addition and multiplication of binary naturals. Hoogerwoord ([Hoo 2]) has derived declarations for addition of a binary natural to a binary natural and a ternary natural, respectively. Together with Ebergen ([Ebe]), he has derived a circuit implementation for a serial-parallel multiplier of binary naturals. Instead of a using a functional language, Brown and Hutton ([Bro]) have derived a combinatorial circuit that increments a binary natural of length 3 by a bit, using relational calculus. But, according to [Hut], this is difficult to derive fully-formally. This literature differs in the following ways from this thesis:

- In the literature, apart from [Ebe] and [Bro], hardware implementations are given instead of derived formally. In this thesis we have defined a simple formalism, such that we can obtain hardware implementations by further developing the declarations obtained in chapters 3 through 6. This approach is completely different from that of [Ebe] and [Bro].

- We have considered binary arithmetic for integers, whereas in the literature only binary arithmetic for naturals is considered. One could say that we can easily extend the representation of naturals to integers. However, this extension can be non-trivial, e.g. we were able to derive declarations for subtraction and decrement of binary integers, that are completely different from the declarations one would obtain using binary naturals. Also, they were easier to derive.
- In the literature formal derivations are only given at the representation level. This often forces the authors to make design decisions at this level which could have been taken at the arithmetic level. Hence, the derivations at the representation level become more complex than the derivations in this thesis.

The derivation of a function for binary division is shorter and easier than the derivation of a sequential program for binary division; e.g. compare the derivation of declaration 6.2 of function *dm* of chapter 6 to the derivation of a sequential program in section 5.1 of [Kal], which is at least twice as long and complex. To my surprise, I have not found a single functional derivation of a binary division algorithm in the literature.

One could expect that my mental arithmetic skills have improved while doing research for this thesis. Unfortunately, this is not the case. I think there is only one way to really improve these skills, i.e. practise, practise, practise.

Bibliography

[Boo] A.D. Booth
A Signed Binary Multiplication Technique
Quarterly Journal of Mechanics and Applied Mathematics, Volume 4, Part 2, June 1951, pp 236-240.

[Bro] C. Brown, G. Hutton
Categories, Allegories and Circuit Design
Proceedings Symposium on Logic in Computer Science LICS '94, Cat. No. 94CH3464-5, 4-7 July 1994, pp 372-381.

[Ebe] J.C. Ebergen, R.R. Hoogerwoord
A Derivation of a Serial-Parallel Multiplier
CS-90-13, University of Waterloo Computer Science Department, January 1990.

[Hoo 1] R.R. Hoogerwoord
The design of functional programs: a calculational approach
Ph. D. thesis, Eindhoven University of Technology, 1989.

[Hoo 2] R.R. Hoogerwoord
Programming by Calculation
unpublished.

[Hut] G. Hutton, E. Meijer
Deriving Representation Changers Functionally
Journal of Functional Programming, Volume 6, Part 1, January 1996, pp 181-188.

[Kal] A. Kaldewaij
Programming: The Derivation of Algorithms
Prentice Hall, 1990.

[Klo 1] C.D. Kloos, W. Dosch
Transformational Development of Digital Circuit Descriptions: A Case Study
CompEuro87, 1987, pp 217-237.

[Klo 2] C.D. Kloos, W. Dosch
Transformational Development of Circuit Descriptions for Binary Adders
Methods of Programming, LNCS 544, 1991, pp 217-237.

[Klo 3] C.D. Kloos, W. Dosch, B. Möller
Design and Proof of Multipliers by Correctness-Preserving Transformation
CompEuro92 Proceedings 'Computer System and Software Engineering', Cat. No. 91CH3121-1, 4-8 May 1992, pp 238-243.

[Omo] A.R. Omondi
Computer Arithmetic Systems, Algorithms, Architectures and Implementations
Prentice Hall, 1994.

[Zan] H. Zantema
Basic arithmetic by rewriting and its complexity
unpublished.