6

Bit Vectors

6.1 Bit-Vector Arithmetic

The design of computer systems is error-prone, and, thus, decision procedures for reasoning about such systems are highly desirable. A computer system uses *bit vectors* to encode information, for example, numbers. Owing to the finite domain of these bit vectors, the semantics of operations such as addition no longer matches what we are used to when reasoning about unbounded types, for example, the natural numbers.

6.1.1 Syntax

The subset of bit-vector arithmetic that we consider is defined by the following grammar:

$$
\begin{aligned}
\mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \neg \mathit{formula} \mid (\,\mathit{formula}\,) \mid \mathit{atom} \\
\mathit{atom} &: \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean\text{-}Identifier} \mid \mathit{term}[\,\mathit{constant}\,] \\
\mathit{rel} &: < \mid = \\
\mathit{term} &: \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \sim \mathit{term} \mid \mathit{constant} \mid \mathit{atom}\,?\,\mathit{term}\,:\,\mathit{term} \mid \\
&\quad\ \mathit{term}[\,\mathit{constant} : \mathit{constant}\,] \mid \mathit{ext}(\,\mathit{term}\,) \\
\mathit{op} &: + \mid - \mid \cdot \mid / \mid << \mid >> \mid \& \mid \mid \mid \oplus \mid \circ
\end{aligned}
$$

As usual, other useful operators such as "$\vee$", "$\neq$", and "$\geq$" can be obtained using Boolean combinations of the operators that appear in the grammar. Most operators have a straightforward meaning, but a few operators are unique to bit-vector arithmetic. The unary operator "$\sim$" denotes bitwise negation. The function *ext* denotes sign and zero extension (the meanings of these operators are explained in Sect. 6.1.3). The ternary operator $c\,?\,a\,:\,b$ is a case-split: the operator evaluates to a if c holds, and to b otherwise. The operators "<<" and ">>" denote left and right shifts, respectively. The operator "$\oplus$" denotes bitwise XOR. The binary operator "$\circ$" denotes concatenation of bit vectors.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_6

Motivation

As an example to describe our motivation, the following formula obviously holds over the integers:

$$(x - y > 0) \iff (x > y)\,. \tag{6.1}$$

If x and y are interpreted as finite-width bit vectors, however, this equivalence no longer holds, owing to possible **overflow** of the subtraction operation. As another example, consider the following small C program:

```
unsigned char number = 200;
number = number + 100;
printf("Sum: %d\n", number);
```

This program may return a surprising result, as most architectures use eight bits to represent variables with type `unsigned char`:

$$\begin{array}{r} \texttt{11001000} = 200 \\ +\,\texttt{01100100} = 100 \\ \hline = \texttt{00101100} = 44 \end{array}$$

When represented with eight bits by a computer, 200 is stored as 11001000. Adding 100 results in an overflow, as the ninth bit of the result is discarded.

The meaning of operators such as "+" is therefore defined by means of *modular* arithmetic. However, the problem of reasoning about bit vectors extends beyond that of overflow and modular arithmetic. For efficiency reasons, programmers use bit-level operators to encode as much information as possible into the number of bits available.

As an example, consider the implementation of a propositional SAT solver. Recall the definition of a *literal* (Definition 1.11): a literal is a variable or its negation. Propositional SAT solvers that operate on formulas in CNF have to store a large number of such literals. We assume that we have numbered the variables that occur in the formula, and denote the variables by $x_1, x_2, \ldots$.

The DIMACS standard for CNF uses signed numbers to encode a literal, e.g., the literal $\neg x_3$ is represented as -3. The fact that we use signed numbers for the encoding avoids the use of one bit vector to store the sign. On the other hand, it reduces the possible number of variables to $2^{31} - 1$ (the index 0 cannot be used any more), but this is still more than sufficient for any practical purpose.

In order to extract the index of a variable, we have to perform a case-split on the sign of the bit vector, for example, as follows:

```
unsigned variable_index(int literal) {
  if(literal < 0)
    return -literal;
  else
    return literal;
}
```

The branch needed to implement the `if` statement in the program above slows down the execution of the program, as it is hard to predict for the branch prediction mechanisms of modern processors. Most SAT solvers therefore use a different encoding: the least significant bit of the bit vector is used to encode the sign of the literal, and the remaining bits encode the variable. The index of the variable can then be extracted by means of a bit-vector right-shift operation:

```
unsigned variable_index(unsigned literal) {
  return literal >> 1;
}
```

Similarly, the sign can be obtained by means of a bitwise AND operation:

```
bool literal_sign(unsigned literal) {
  return literal & 1;
}
```

The bitwise right-shift operation and the bitwise AND are implemented in most microprocessors, and both can be executed efficiently. Such bitwise operators also frequently occur in hardware design. Reasoning about such artifacts requires *bit-vector arithmetic*.

6.1.2 Notation

We use a simple variant of Church's **λ-notation** in order to define vectors easily. A lambda expression for a bit vector with l bits has the form

$$\lambda i \in \{0, \ldots, l-1\}.\, f(i)\,, \tag{6.2}$$

where $f(i)$ is an expression that denotes the value of the i-th bit.

The use of the λ-operator to denote bit vectors is best explained by an example.

Example 6.1. Consider the following expressions:

- The expression
$$\lambda i \in \{0, \ldots, l-1\}.\, 0 \tag{6.3}$$
denotes the l-bit bit vector that consists only of zeros.
- A λ-expression is simply another way of defining a function *without* giving it a name. Thus, instead of defining a function z with
$$z(i) \doteq 0\,, \tag{6.4}$$
we can simply write $\lambda i \in \{0, \ldots, l-1\}.\, 0$ for z.
- The expression
$$\lambda i \in \{0, \ldots, 7\}.\, \begin{cases} 0 : i \text{ is even} \\ 1 : \text{otherwise} \end{cases} \tag{6.5}$$
denotes the bit vector 10101010.

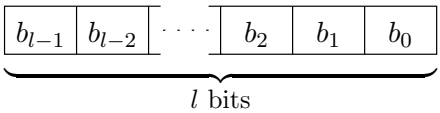

Fig. 6.1. A bit vector b with l bits. The bit number i is denoted by b_i

- The expression
$$\lambda i \in \{0, \ldots, l-1\}. \neg x_i \tag{6.6}$$
denotes the bitwise negation of the vector x.

We omit the domain of i from the lambda expression if the number of bits is clear from the context.

6.1.3 Semantics

We now give a formal definition of the meaning of a bit-vector arithmetic formula. We first clarify what a bit vector is.

Definition 6.2 (bit vector). *A* bit vector b *is a vector of bits with a given length* l *(or dimension):*

$$b : \{0, \ldots, l-1\} \longrightarrow \{0, 1\} . \tag{6.7}$$

$bvec_l$ *The set of all* 2^l *bit vectors of length* l *is denoted by* $bvec_l$*. The* i*-th bit of the bit vector* b *is denoted by* b_i *(Fig. 6.1).*

The meaning of a bit-vector formula obviously depends on the width of the bit-vector variables in it. This applies even if no arithmetic is used. As an example,

$$x \neq y \wedge x \neq z \wedge y \neq z \tag{6.8}$$

is unsatisfiable for bit vectors x, y, and z that are one bit wide, but satisfiable for larger widths.

We sometimes use bit vectors that encode positive numbers only (unsigned bit vectors), and also bit vectors that encode both positive and negative numbers (signed bit vectors). Thus, each expression is associated with a **type**. The type of a bit-vector expression is

1. the width of the expression in bits, and
2. whether it is signed or unsigned.

We restrict the presentation to bit vectors that have a fixed, given length, as bit-vector arithmetic becomes undecidable as soon as arbitrary-width bit vectors are permitted. The width is known in most problems that arise in practice.

In order to clarify the type of an expression, we add indices in square brackets to the operator and operands in order to denote the bit width (this is not to be confused with b_l, which denotes bit l of b). As an example, $a_{[32]} \cdot_{[32]} b_{[32]}$ denotes the multiplication of a and b. Both the result and the operands are 32 bits wide, and the remaining 32 bits of the result are discarded. The expression $a_{[8]} \circ_{[24]} b_{[16]}$ denotes the concatenation of a and b and is in total 24 bits wide. In most cases, the width is clear from the context, and we therefore usually omit the subscript.

Bitwise Operators

The meanings of bitwise operators can be defined through the bit vectors that they yield. The binary bitwise operators take two l-bit bit vectors as arguments and return an l-bit bit vector. As an example, the signature of the bitwise OR operator "|" is

$$|_{[l]} : (bvec_l \times bvec_l) \longrightarrow bvec_l \ . \tag{6.9}$$

Using the λ-notation, the bitwise OR operator is defined as follows:

$$a \,|\, b \doteq \ \lambda i.\, (a_i \vee b_i) \ . \tag{6.10}$$

All the other bitwise operators are defined in a similar manner. In the following, we typically provide both the signature and the definition together.

Arithmetic Operators

The meaning of a bit-vector formula with arithmetic operators depends on the *interpretation* of the bit vectors that it contains. There are many ways to encode numbers using bit vectors. The most commonly used encodings for integers are the **binary encoding** for unsigned integers and **two's complement** for signed integers.

Definition 6.3 (binary encoding). *Let x denote a natural number, and $b \in bvec_l$ a bit vector. We call b a* binary encoding *of x iff*

$$x = \langle b \rangle_U \ , \tag{6.11}$$

where $\langle b \rangle_U$ is defined as follows:

$\langle \cdot \rangle_U$

$$\begin{aligned} \langle \cdot \rangle_U &: bvec_l \longrightarrow \{0, \ldots, 2^l - 1\} \ , \\ \langle b \rangle_U &\doteq \ \textstyle\sum_{i=0}^{l-1} b_i \cdot 2^i . \end{aligned} \tag{6.12}$$

The bit b_0 is called the **least significant bit**, *and the bit b_{l-1} is called the* **most significant bit**.

Binary encoding can be used to represent nonnegative integers only. One way of encoding negative numbers as well is to use one of the bits as a **sign bit**.

A naive way of using a sign bit is to simply negate the number if a designated bit is set, for example, the most significant bit. As an example, 1001 could be interpreted as -1 instead of 1. This encoding is hardly ever used in practice.[1] Instead, most microprocessor architectures implement the **two's complement** encoding.

Definition 6.4 (two's complement). *Let x denote a natural number, and $b \in bvec_l$ a bit vector. We call b the* two's complement *of x iff*

$$x = \langle b \rangle_S \ , \tag{6.13}$$

$\langle\cdot\rangle_S$ *where $\langle b \rangle_S$ is defined as follows:*

$$\begin{aligned} \langle\cdot\rangle_S &: bvec_l \longrightarrow \{-2^{l-1}, \ldots, 2^{l-1} - 1\} \ , \\ \langle b \rangle_S &:= -2^{l-1} \cdot b_{l-1} + \textstyle\sum_{i=0}^{l-2} b_i \cdot 2^i \ . \end{aligned} \tag{6.14}$$

The bit with index $l - 1$ is called the sign bit *of b.*

Example 6.5. Some encodings of integers in binary and two's complement are

$$\begin{aligned} \langle 11001000 \rangle_U &= 200 \ , \\ \langle 11001000 \rangle_S &= -128 + 64 + 8 = -56 \ , \\ \langle 01100100 \rangle_S &= 100 \ . \end{aligned}$$

Note that the meanings of the relational operators ">", "<", "≤", "≥", the multiplicative operators "·", "/", and the right-shift operator ">>" depend on whether a binary encoding or a two's complement encoding is used for the operands, which is why the encoding of the bit vectors is part of the type. We use the subscript U for a binary encoding (unsigned) and the subscript S for a two's complement encoding (signed). We may omit this subscript if the encoding is clear from the context, or if the meaning of the operator does not depend on the encoding (this is the case for most operators).

As suggested by the example at the beginning of this chapter, arithmetic on bit vectors has a wraparound effect: if the number of bits required to represent the result exceeds the number of bits available, the additional bits of the result are discarded, i.e., the result is truncated. This corresponds to a **modulo** operation, where the base is 2^l. We write

$$x = y \mod b \tag{6.15}$$

to denote that x and y are equal modulo b. The use of modulo arithmetic allows a straightforward definition of the interpretation of all arithmetic operators:

[1] The main reason for this is the fact that it makes the implementation of arithmetic operators such as addition more complicated, and that there are two encodings for 0, namely 0 and -0.

- Addition and subtraction:

$$a_{[l]} +_U b_{[l]} = c_{[l]} \iff \langle a\rangle_U + \langle b\rangle_U = \langle c\rangle_U \mod 2^l , \tag{6.16}$$
$$a_{[l]} -_U b_{[l]} = c_{[l]} \iff \langle a\rangle_U - \langle b\rangle_U = \langle c\rangle_U \mod 2^l , \tag{6.17}$$
$$a_{[l]} +_S b_{[l]} = c_{[l]} \iff \langle a\rangle_S + \langle b\rangle_S = \langle c\rangle_S \mod 2^l , \tag{6.18}$$
$$a_{[l]} -_S b_{[l]} = c_{[l]} \iff \langle a\rangle_S - \langle b\rangle_S = \langle c\rangle_S \mod 2^l . \tag{6.19}$$

Note that $a +_U b = a +_S b$ and $a -_U b = a -_S b$ (see Problem 6.7), and thus the U/S subscript can be omitted from the addition and subtraction operands. A semantics for mixed-type expressions is also easily defined, as shown in the following example:

$$a_{[l]U} +_U b_{[l]S} = c_{[l]U} \iff \langle a\rangle + \langle b\rangle_S = \langle c\rangle \mod 2^l . \tag{6.20}$$

- Unary minus:

$$-a_{[l]} = b_{[l]} \iff -\langle a\rangle_S = \langle b\rangle_S \mod 2^l . \tag{6.21}$$

- Relational operators:

$$a_{[l]U} < b_{[l]U} \iff \langle a\rangle_U < \langle b\rangle_U , \tag{6.22}$$
$$a_{[l]S} < b_{[l]S} \iff \langle a\rangle_S < \langle b\rangle_S , \tag{6.23}$$
$$a_{[l]U} < b_{[l]S} \iff \langle a\rangle_U < \langle b\rangle_S , \tag{6.24}$$
$$a_{[l]S} < b_{[l]U} \iff \langle a\rangle_S < \langle b\rangle_U . \tag{6.25}$$

The semantics for the other relational operators such as "$\geq$" follows the same pattern. Note that ANSI-C compilers do not implement the relational operators on operands with mixed encodings the way they are formalized above (see Problem 6.6). Instead, the signed operand is converted to an unsigned operand, which does not preserve the meaning expected by many programmers.

- Multiplication and division:

$$a_{[l]} \cdot_U b_{[l]} = c_{[l]} \iff \langle a\rangle_U \cdot \langle b\rangle_U = \langle c\rangle_U \mod 2^l , \tag{6.26}$$
$$a_{[l]} /_U b_{[l]} = c_{[l]} \iff \langle a\rangle_U / \langle b\rangle_U = \langle c\rangle_U \mod 2^l , \tag{6.27}$$
$$a_{[l]} \cdot_S b_{[l]} = c_{[l]} \iff \langle a\rangle_S \cdot \langle b\rangle_S = \langle c\rangle_S \mod 2^l , \tag{6.28}$$
$$a_{[l]} /_S b_{[l]} = c_{[l]} \iff \langle a\rangle_S / \langle b\rangle_S = \langle c\rangle_S \mod 2^l . \tag{6.29}$$

The semantics of multiplication is independent of whether the arguments are interpreted as unsigned or two's complement (see Problem 6.8), and thus the U/S subscript can be omitted. This does not hold in the case of division.

- The extension operator: converting a bit vector to a bit vector with more bits is called **zero extension** in the case of an unsigned bit vector, and **sign extension** in the case of a signed bit vector. Let $l \leq m$. The value that is encoded does not change:

$$ext_{[m]U}(a_{[l]}) = b_{[m]U} \iff \langle a \rangle_U = \langle b \rangle_U \ , \tag{6.30}$$

$$ext_{[m]S}(a_{[l]}) = b_{[m]S} \iff \langle a \rangle_S = \langle b \rangle_S \ . \tag{6.31}$$

- Shifting: the left-shift operator "<<" takes two operands and shifts the first one to the left as many times as is given by the respective value of the second operand. The width of the left-hand-side operand is called the *width of the shift*, whereas the width of the right-hand-side operator is the *width of the shift distance*. The vector is filled up with zeros from the right:

$$a_{[l]} << b_U = \lambda i \in \{0, \ldots, l-1\}. \begin{cases} a_{i-\langle b \rangle_U} & : i \geq \langle b \rangle_U \\ 0 & : \text{otherwise} \ . \end{cases} \tag{6.32}$$

  See also Problem 6.5. The meaning of the right-shift ">>" operator depends on the encoding of the first operand: if it uses binary encoding (which, recall, is for unsigned bit vectors), zeros are inserted from the left. This is called a **logical right shift**:

$$a_{[l]U} >> b_U = \lambda i \in \{0, \ldots, l-1\}. \begin{cases} a_{i+\langle b \rangle_U} & : i < l - \langle b \rangle_U \\ 0 & : \text{otherwise} \ . \end{cases} \tag{6.33}$$

  If the first operand uses two's complement encoding, the sign bit of a is replicated. This is also called an **arithmetic right shift**:

$$a_{[l]S} >> b_U = \lambda i \in \{0, \ldots, l-1\}. \begin{cases} a_{i+\langle b \rangle_U} & : i < l - \langle b \rangle_U \\ a_{l-1} & : \text{otherwise} \ . \end{cases} \tag{6.34}$$

  The shift operators are rarely defined for a signed shift distance. An option could be to flip the direction of the shift in case b is negative; e.g., a left shift with distance -1 is a right shift with distance 1.

6.2 Deciding Bit-Vector Arithmetic with Flattening

6.2.1 Converting the Skeleton

The most commonly used decision procedure for bit-vector arithmetic is called *flattening*.[2] Algorithm 6.2.1 implements this technique. For a given bit-vector arithmetic formula φ, the algorithm computes an equisatisfiable propositional formula $\mathcal{B}$, which is then passed to a SAT solver.

$\mathcal{B}$

Let $At(\varphi)$ denote the set of atoms in φ. As a first step, the algorithm replaces the atoms in φ with new Boolean variables. We denote the variable that replaces an atom $a \in At(\varphi)$ by $e(a)$, and call this the **propositional encoder**

$At(\varphi)$

of a. The resulting formula is denoted by $e(\varphi)$. We call it the **propositional skeleton** of φ. The propositional skeleton is the expression that is assigned to $\mathcal{B}$ initially.

$e(\varphi)$

Let $T(\varphi)$ denote the set of terms in φ. The algorithm then assigns a vector of new Boolean variables to each bit-vector term in $T(\varphi)$. We use $e(t)$ to denote this vector of variables for a given $t \in T(\varphi)$, and $e(t)_i$ to denote the variable for the bit with index i of the term t. The width of $e(t)$ matches the width of the term t. Note that, so far, we have used e to denote three different, but related things: a propositional encoder of an atom, a propositional formula resulting from replacing all atoms of a formula with their respective propositional encoders, and a propositional encoder of a term.

$T(\varphi)$

$e(t)$

The algorithm then iterates over the terms and atoms of φ, and computes a constraint for each of them. The constraint is returned by the function BV-Constraint, and is added as a conjunct to $\mathcal{B}$.

Algorithm 6.2.1: BV-Flattening

Input: A formula φ in bit-vector arithmetic
Output: An equisatisfiable Boolean formula $\mathcal{B}$

```
1. function BV-Flattening
2.     B:=e(φ);                          ▷ the propositional skeleton of φ
3.     for each t_[l] ∈ T(φ) do
4.         for each i ∈ {0,...,l−1} do
5.                 set e(t)_i to a new Boolean variable;
6.     for each a ∈ At(φ) do
7.         B:=B∧ BV-Constraint(e,a);
8.     for each t_[l] ∈ T(φ) do
9.         B:=B∧ BV-Constraint(e,t);
10.    return B;
```

The constraint that is needed for a particular atom a or term t depends on the atom or term, respectively. In the case of a bit vector or a Boolean variable, no constraint is needed, and BV-Constraint returns true. If t is a bit-vector constant $C_{[l]}$, the following constraint is generated:

$$\bigwedge_{i=0}^{l-1} (C_i \iff e(t)_i) \; . \tag{6.35}$$

Otherwise, t must contain a bit-vector operator. The constraint that is needed depends on this operator. The constraints for the bitwise operators are

[2] In colloquial terms, this technique is sometimes referred to as "*bit-blasting*".

straightforward. As an example, consider bitwise OR, and let $t = a \,|_{[l]}\, b$. The constraint returned by BV-Constraint is

$$\bigwedge_{i=0}^{l-1}((a_i \vee b_i) \iff e(t)_i) \,. \tag{6.36}$$

The constraints for the other bitwise operators follow the same pattern.

6.2.2 Arithmetic Operators

The constraints for the arithmetic operators often follow implementations of these operators as a *circuit*. There is an abundance of literature on how to build efficient circuits for various arithmetic operators. However, experiments with various alternative circuits have shown that the simplest ones usually burden the SAT solver the least. We begin by defining a one-bit adder, also called a **full adder**.

Definition 6.6 (full adder). *A* full adder *is defined using the two functions carry and sum. Both of these functions take three input bits a, b, and cin as arguments. The function carry calculates the carry-out bit of the adder, and the function sum calculates the sum bit:*

$$sum(a, b, cin) \doteq (a \oplus b) \oplus cin \,, \tag{6.37}$$

$$carry(a, b, cin) \doteq (a \wedge b) \vee ((a \oplus b) \wedge cin) \,. \tag{6.38}$$

We can extend this definition to adders for bit vectors of arbitrary length.

Definition 6.7 (carry bits). *Let x and y denote two l-bit bit vectors and cin a single bit. The* carry bits c_0 *to* c_l *are defined recursively as follows:*

$$c_i \doteq \begin{cases} cin & : i = 0 \\ carry(x_{i-1}, y_{i-1}, c_{i-1}) & : otherwise \,. \end{cases} \tag{6.39}$$

Definition 6.8 (adder). *An l-bit* adder *maps two l-bit bit vectors x, y and a carry-in bit cin to their sum and a carry-out bit. Let* c_i *denote the i-th carry bit as in Definition 6.7. The function add is defined using the carry bits* c_i*:*

$$add(x, y, cin) \doteq \langle result, cout \rangle \,, \tag{6.40}$$

$$result_i \doteq sum(x_i, y_i, c_i) \quad for\ i \in \{0, \ldots, l-1\} \,, \tag{6.41}$$

$$cout \doteq c_n \,. \tag{6.42}$$

The circuit equivalent of this construction is called a *ripple carry adder*. One can easily implement the constraint for $t = a + b$ using an adder with $cin = 0$:

$$\bigwedge_{i=0}^{l-1}(add(a, b, 0).result_i \iff e(t)_i) \,. \tag{6.43}$$

One can prove by induction on l that (6.43) holds if and only if $\langle a\rangle_U + \langle b\rangle_U = \langle e(t)\rangle_U \bmod 2^l$, which shows that the constraint complies with the semantics.

Subtraction, where $t = a - b$, is implemented with the same circuit by using the following constraint (recall that $\sim b$ is the bitwise negation of b):

$$\bigwedge_{i=0}^{l-1} (add(a, \sim b, 1).result_i \iff e(t)_i) \; . \tag{6.44}$$

This implementation makes use of the fact that $\langle(\sim b) + 1\rangle_S = -\langle b\rangle_S \bmod 2^l$ (see Problem 6.9).

Relational Operators

The equality $a =_{[l]} b$ is implemented using simply a conjunction:

$$\bigwedge_{i=0}^{l-1} a_i = b_i \iff e(t) \; . \tag{6.45}$$

The relation $a < b$ is transformed into $a - b < 0$, and an adder is built for the subtraction, as described above. Thus, b is negated and the carry-in bit of the adder is set to TRUE. The result of the relation $a < b$ depends on the encoding. In the case of unsigned operands, $a < b$ holds if the carry-out bit *cout* of the adder is FALSE:

$$\langle a\rangle_U < \langle b\rangle_U \iff \neg add(a, \sim b, 1).cout \; . \tag{6.46}$$

In the case of signed operands, $a < b$ holds if and only if $(a_{l-1} = b_{l-1}) \neq cout$:

$$\langle a\rangle_S < \langle b\rangle_S \iff (a_{l-1} \iff b_{l-1}) \oplus add(a,\ b, 1).cout \; . \tag{6.47}$$

Comparisons involving mixed encodings are implemented by extending both operands by one bit, followed by a signed comparison.

Shifts

Recall that we call the width of the left-hand-side operand of a shift (the vector that is to be shifted) the *width of the shift*, whereas the width of the right-hand-side operand is the *width of the shift distance*.

We restrict the left and right shifts as follows: the width l of the shift must be a power of two, and the width of the shift distance n must be $\log_2 l$.

With this restriction, left and right shifts can be implemented by using the following construction, which is called the *barrel shifter*. The shifter is split into n stages. Stage s can shift the operand by 2^s bits or leave it unaltered. The function *ls* is defined recursively for $s \in \{-1, \ldots, n-1\}$:

$$ls(a_{[l]}, b_{[n]U}, -1) \doteq a\ , \tag{6.48}$$

$$ls(a_{[l]}, b_{[n]U}, s) \doteq$$
$$\lambda i \in \{0, \ldots, l-1\}. \begin{cases} (ls(a, b, s-1))_{i-2^s} & : i \geq 2^s \wedge b_s \\ (ls(a, b, s-1))_i & : \neg b_s \\ 0 & : \text{otherwise}\ . \end{cases} \tag{6.49}$$

The barrel shifter construction needs only $O(n \log n)$ logical operators, in contrast to the naive implementation, which requires $O(n^2)$ operators.

Multiplication and Division

Multipliers can be implemented following the most simplistic circuit design, which uses the *shift-and-add* idea. The function *mul* is defined recursively for $s \in \{-1, \ldots, n-1\}$, where n denotes the width of the second operand:

$$mul(a, b, -1) \doteq 0, \tag{6.50}$$
$$mul(a, b, s) \doteq mul(a, b, s-1) + (b_s ? (a << s) : 0)\ . \tag{6.51}$$

A division $a/_U b$ is implemented by adding two constraints:

$$b \neq 0 \implies e(t) \cdot b + r = a\ , \tag{6.52}$$
$$b \neq 0 \implies r < b\ . \tag{6.53}$$

The variable r is a new bit vector of the same width as b, and contains the remainder. The signed-division and modulo operations are done in a similar way.

6.3 Incremental Bit Flattening

6.3.1 Some Operators Are Hard

For some operators, the size of the formula generated by BV-Constraint may be large. As an example, consider the formula for a single multiplier with n bits. The table in Fig. 6.2 shows the number of variables and the number of CNF clauses that are generated from the formula using Tseitin's encoding (see Sect. 1.3).

In addition to the sheer size of these formulas, their symmetry and connectivity is a burden on the decision heuristic of state-of-the-art propositional SAT solvers. As a consequence, formulas with multipliers are often very hard to solve. Similar observations hold for other arithmetic operators such as division and modulo.

As an example, consider the following bit-vector formula:

$$a \cdot b = c\ \wedge\ b \cdot a \neq c\ \wedge\ x < y\ \wedge\ x > y\ . \tag{6.54}$$

n	Number of variables	Number of clauses
8	313	1001
16	1265	4177
24	2857	9529
32	5089	17057
64	20417	68929

Fig. 6.2. The size of the constraint for an n-bit multiplier expression after Tseitin's transformation

When this formula is encoded into CNF, a SAT instance with about 11 000 variables is generated for a width of 32 bits. This formula is obviously unsatisfiable. There are two reasons for this: the first two conjuncts are inconsistent, and independently, the last two conjuncts are inconsistent. The decision heuristics of most SAT solvers (see Chap. 2) are biased towards splitting first on variables that are used frequently, and thus favor decisions on a, b, and c. Consequently, they attempt to show unsatisfiability of the formula on the hard part, which includes the two multipliers. The "easy" part of the formula, which contains only two relational operators, is ignored. Most propositional SAT solvers cannot solve this formula in a reasonable amount of time.

In many cases, it is therefore beneficial to build the flattened formula $\mathcal{B}$ *incrementally*. Algorithm 6.3.1 is a realization of this idea: as before, we start with the propositional skeleton of φ. We then add constraints for the "inexpensive" operators, and omit the constraints for the "expensive" operators. The bitwise operators are typically inexpensive, whereas arithmetic operators are expensive. The encodings with missing constraints can be considered an *abstraction* of φ, and thus the algorithm is an instance of the abstraction–refinement procedure introduced in Sect. 4.4.

The current flattening $\mathcal{B}$ is passed to a propositional SAT solver. If $\mathcal{B}$ is unsatisfiable, so is the original formula φ. Recall the formula (6.54): as soon as the constraints for the second half of the formula are added to $\mathcal{B}$, the encoding becomes unsatisfiable, and we may conclude that (6.54) is unsatisfiable without considering the multipliers.

On the other hand, if $\mathcal{B}$ is satisfiable, one of two cases applies:

1. The original formula φ is unsatisfiable, but one (or more) of the omitted constraints is needed to show this.
2. The original formula φ is satisfiable.

In order to distinguish between these two cases, we can check whether the satisfying assignment produced by the SAT solver satisfies the constraints that we have omitted. As we might have removed variables, the satisfying assignment might have to be extended by setting the missing values to some constant, for example, zero. If this assignment satisfies all constraints, the second case applies, and the algorithm terminates.

Algorithm 6.3.1: INCREMENTAL-BV-FLATTENING

Input: A formula φ in bit-vector logic
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
 1. function INCREMENTAL-BV-FLATTENING(φ)
 2.     B := e(φ);                                  ▷ propositional skeleton of φ
 3.     for each t_[l] ∈ T(φ) do
 4.         for each i ∈ {0, ..., l − 1} do
 5.             set e(t)_i to a new Boolean variable;
 6.     while (TRUE) do
 7.         α := SAT-SOLVER(B);
 8.         if α="Unsatisfiable" then
 9.             return "Unsatisfiable";
10.         else
11.             Let I ⊆ T(φ) be the set of terms that are inconsistent with the
                    satisfying assignment;
12.             if I = ∅ then
13.                 return "Satisfiable";
14.             else
15.                 Select "easy" F' ⊆ I;
16.                 for each t_[l] ∈ F' do
17.                     B:=B ∧ BV-CONSTRAINT(e, t);
```

If this is not so, one or more of the terms for which the constraints were omitted is inconsistent with the assignment provided by the SAT solver. We denote this set of terms by I. The algorithm proceeds by selecting some of these terms, adding their constraints to $\mathcal{B}$, and reiterating. The algorithm terminates, as we strictly add more constraints with each iteration. In the worst case, all constraints from $T(\varphi)$ are added to the encoding.

6.3.2 Abstraction with Uninterpreted Functions

In many cases, omitting constraints for particular operators may result in a flattened formula that is too weak, and thus is satisfied by too many spurious models. On the other hand, the full constraint may burden the SAT solver too much. A compromise between the maximum strength of the full constraint and omitting the constraint altogether is to replace functions over bit vectors by uninterpreted functions (see Sect. 4.2). This technique is particularly effective when one is checking the equivalence of two models.

For example, let $a_1\, op\, b_1$ and $a_2\, op\, b_2$ be two terms, where op is some binary operator (for simplicity, assume that these are the only terms in the input formula that use op). Replace op with a new uninterpreted-function symbol G

to obtain instead $G(a_1, b_1)$ and $G(a_2, b_2)$. The resulting formula is abstract, and does not contain constraints that correspond to the flattening of *op*.

6.4 Fixed-Point Arithmetic

6.4.1 Semantics

Many applications, for example, in scientific computing, require arithmetic on numbers with a fractional part. High-end microprocessors offer support for **floating-point arithmetic** for this purpose. However, fully featured floating-point arithmetic is too heavyweight for many applications, such as control software embedded in vehicles, and computer graphics. In these domains, **fixed-point arithmetic** is a reasonable compromise between accuracy and complexity. Fixed-point arithmetic is also commonly supported by database systems, for example, to represent amounts of currency.

In fixed-point arithmetic, the representation of a number is partitioned into two parts, the *integer part* (also called the *magnitude*) and the *fractional part* (Fig. 6.3). The number of digits in the fractional part is fixed—hence the name "fixed point arithmetic". The number 1.980, for example, is a fixed-point number with a three-digit fractional part.

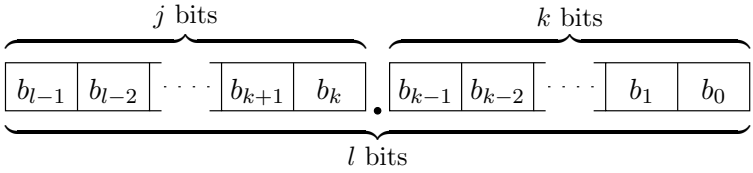

Fig. 6.3. A fixed-point bit vector b with a total of $j + k = l$ bits. The dot is called the radix point. The j bits before the dot represent the magnitude (the integer part), whereas the k bits after the dot represent the fractional part

The same principle can be applied to binary arithmetic, as captured by the following definition. Recall the definition of $\langle\cdot\rangle_S$ (two's complement) from Sect. 6.1.3.

Definition 6.9. *Given two bit vectors M and F with m and f bits, respectively, we define the rational number that is represented by $M.F$ as follows and denote it by $\langle M.F\rangle$:*

$$\langle\cdot\rangle : \{0,1\}^{m+f} \longrightarrow \mathbb{Q}\,,$$
$$\langle M.F\rangle := \frac{\langle M \circ F\rangle_S}{2^f}\,.$$

Example 6.10. Some encodings of rational numbers as fixed-point numbers with base 2 are

$$\begin{aligned}\langle 0.10\rangle &= 0.5\ ,\\ \langle 0.01\rangle &= 0.25\ ,\\ \langle 01.1\rangle &= 1.5\ ,\\ \langle 11111111.1\rangle &= -0.5\ .\end{aligned}$$

Some rational numbers are not precisely representable using fixed-point arithmetic in base 2: they can only be approximated. As an example, for $m = f = 4$, the two numbers that are closest to $1/3$ are

$$\begin{aligned}\langle 0000.0101\rangle &= 0.3125\ ,\\ \langle 0000.0110\rangle &= 0.375\ .\end{aligned}$$

▟

Definition 6.9 gives us the semantics of fixed-point arithmetic. For example, the meaning of addition on bit vectors that encode fixed-point numbers can be defined as follows:

$$\begin{aligned}&a_M.a_F + b_M.b_F = c_M.c_F \iff\\ &\quad \langle a_M.a_F\rangle \cdot 2^f + \langle b_M.b_F\rangle \cdot 2^f = \langle c_M.c_F\rangle \cdot 2^f \mod 2^{m+f}\ .\end{aligned}$$

There are variants of fixed-point arithmetic that implement **saturation** instead of overflow semantics, that is, instead of wrapping around, the result remains at the highest or lowest number that can be represented with the given precision. Both the semantics and the flattening procedure are straightforward for this case.

6.4.2 Flattening

Fixed-point arithmetic can be flattened just as well as arithmetic using binary encoding or two's complement. We assume that the numbers on the left- and right-hand sides of a binary operator have the same numbers of bits, before and after the radix point. If this is not so, missing bits after the radix point can be added by padding the fractional part with zeros from the right. Missing bits before the radix point can be added from the left using sign extension.

The operators are encoded as follows:

- The bitwise operators are encoded exactly as in the case of binary numbers. Addition, subtraction, and the relational operators can also be encoded as in the case of binary numbers.
- Multiplication requires an alignment. The result of a multiplication of two numbers with f_1 and f_2 bits in the fractional part, respectively, is a number with $f_1 + f_2$ bits in the fractional part. Note that, most commonly, fewer bits are needed, and thus, the extra bits of the result have to be rounded off using a separate **rounding** step.

Example 6.11. Addition and subtraction are straight-forward, but note the need for sign-extension in the second sum:

$$\langle 00.1\rangle + \langle 00.1\rangle = \langle 01.0\rangle$$
$$\langle 000.0\rangle + \langle 1.0\rangle = \langle 111.0\rangle$$

The following examples illustrate multiplication without any subsequent rounding:

$$\langle 0.1\rangle \cdot \langle 1.1\rangle = \langle 0.11\rangle$$
$$\langle 1.10\rangle \cdot \langle 1.1\rangle = \langle 10.010\rangle$$

If needed, rounding towards zero, towards the next even number, or towards $\pm\infty$ can be applied in order to reduce the size of the fractional part; see Problem 6.10. ▟

There are many other encodings of numbers, which we do not cover here, e.g., binary-coded decimals (BCDs), or fixed-point formats with sign bit.

6.5 Problems

6.5.1 Semantics

Problem 6.1 (operators that depend on the encoding). Provide an example (with values of operands) that illustrates that the semantics depend on the encoding (signed vs. unsigned) for each of the following three operators: $>$, $/$, and $>>$.

Problem 6.2 (λ-notation). Define the meaning of $a_l \circ b_l$ using the λ-notation.

Problem 6.3 (negation). What is -10000000_S if the operand of the unary minus is a bit-vector constant?

Problem 6.4 (λ-notation). Define the meaning of $a_{[l]U} >>_{[l]U} b_{[m]S}$ and $a_{[l]S} >>_{[l]S} b_{[m]S}$ using modular arithmetic. Prove these definitions to be equivalent to the definition given in Sect. 6.1.3.

Problem 6.5 (shifts in hardware). What semantics of the left shift does the processor in your computer implement? You can use a program to test this, or refer to the specification of the CPU. Formalize the semantics.

Problem 6.6 (relations in hardware). What semantics of the $<$ operator does the processor in your computer implement if a signed integer is compared with an unsigned integer? Try this for the ANSI-C types `int`, `unsigned`, `char`, and `unsigned char`. Formalize the semantics, and specify the vendor and model of the CPU.

Problem 6.7 (two's complement addition). Prove

$$a_{[l]} +_U b_{[l]} = a_{[l]} +_S b_{[l]}. \tag{6.55}$$

Problem 6.8 (two's complement multiplication). Prove

$$a_{[l]} \cdot_U b_{[l]} = a_{[l]} \cdot_S b_{[l]}. \tag{6.56}$$

6.5.2 Bit-Level Encodings of Bit-Vector Arithmetic

Problem 6.9 (negation). Prove $\langle(\sim b) + 1\rangle_S = -\langle b\rangle_S \mod 2^l$.

Problem 6.10 (relational operators). Prove the correctness of the flattening for "<" as given in Sect. 6.2, for:

(a) Unsigned operands
(b) Signed operands
(c) An unsigned and a signed operand

Problem 6.11 (rounding for fixed-point arithmetic). Formally specify the operator for rounding a fixed-point number with a fractional part of size f_1 to a fractional part of size $f_2 < f_1$ for the following cases:

(a) Rounding to zero
(b) Rounding to $-\infty$
(c) Rounding to the nearest even number

Problem 6.12 (flattening fixed-point arithmetic). Provide a flattening for the three rounding operators above.

6.5.3 Using Solvers for Linear Arithmetic

We introduced decision procedures for linear arithmetic in Chap. 5. A restricted subset of bit-vector arithmetic can be translated into linear arithmetic over the integers. As preparation, we perform a number of transformations on the terms contained in a. We write $[\![b]\!]$ for the result of the transformation of any bit-vector arithmetic term b.

$[\![b]\!]$

- Let $b >> d$ denote a bitwise right-shift term that is contained in a, where b is a term and d is a constant. It is replaced by $[\![b]\!]/2^{\langle d\rangle}$, i.e.,

$$[\![b >> d]\!] \doteq [\![b]\!]/2^{\langle d\rangle} . \tag{6.57}$$

  Bitwise left shifts are handled in a similar manner.

- The bitwise negation of a term b is replaced by $-[\![b]\!] - 1$:

$$[\![\sim b]\!] \doteq -[\![b]\!] - 1 \,. \tag{6.58}$$

- A bitwise AND term $b_{[l]}\&1$, where b is any term, is replaced by a new integer variable x subject to the following constraints over x and a second new integer variable σ:

$$0 \leq x \leq 1 \;\wedge\; [\![b]\!] = 2\sigma + x \;\wedge\; 0 \leq \sigma < 2^{l-1} \,. \tag{6.59}$$

  A bitwise AND with other constants can be replaced using shifts. This can be optimized further by joining together groups of adjacent one-bits in the constant on the right-hand side.
- The bitwise OR is replaced with bitwise negation and bitwise AND.

We are now left with addition, subtraction, multiplication by a constant, and division by a constant.

As the next step, the division operators are removed from the constraints. As an example, the constraint $a/_{[32]}3 = b$ becomes $a = b \cdot_{[34]} 3$. Note that the bit width of the multiplication has to be increased in order to take overflow into account. The operands a and b are sign-extended if signed, and zero-extended if unsigned. After this preparation, we can assume the following form of the atoms without loss of generality:

$$c_1 \cdot t_1 +_{[l]} c_2 \cdot t_2 \;\; rel \;\; b \,, \tag{6.60}$$

where rel is one of the relational operators as defined in Sect. 6.1, c_1, c_2, and b are constants, and t_1 and t_2 are bit-vector identifiers with l bits. Sums with more than two addends can be handled in a similar way.

As we can handle additions efficiently, all scalar multiplications $c \cdot_{[l]} a$ with a small constant c are replaced by c additions. For example, $3 \cdot a$ becomes $a + a + a$.

At this point, we are left with predicates of the following form:

$$t_1 +_{[l]} t_2 \;\; rel \;\; b \,. \tag{6.61}$$

Given that t_1 and t_2 are l-bit unsigned vectors, we have $t_1 \in \{0, \ldots, 2^l - 1\}$ and $t_2 \in \{0, \ldots, 2^l - 1\}$, and, thus, $t_1 + t_2 \in \{0, \ldots, 2^{l+1} - 2\}$. Recall that the bit-vector addition in (6.61) will overflow if $t_1 + t_2$ is larger than $2^l - 1$. We use a case split to adjust the value of the sum in the case of an overflow and transform (6.61) into

$$((t_1 + t_2 \leq 2^l - 1) \;?\; t_1 + t_2 \;:\; (t_1 + t_2 - 2^l)) \;\; op \;\; b \,. \tag{6.62}$$

Based on this description, answer the following questions:

Problem 6.13 (translation to integer arithmetic). Translate the following bit-vector formula into a formula over integers:

$$x_{[8]} +_{[8]} 100 \leq 10_{[8]} \,. \tag{6.63}$$

Problem 6.14 (bitwise AND). Give a translation of

$$x_{[32]U} = y_{[32]U} \& 0\text{xffff0000} \tag{6.64}$$

into disjunctive integer linear arithmetic that is more efficient than that suggested by (6.59).

Problem 6.15 (scalar multiplications). Rewriting scalar multiplications $c \cdot_{[l]} a$ into c additions is inefficient if c is large owing to the cost of the subsequent splitting. Suggest an alternative that uses a new variable.

Problem 6.16 (addition without splitting). Can you propose a different translation for addition that does not use case splitting but uses a new integer variable instead?

Problem 6.17 (removing the conditional operator). Our grammar for integer linear arithmetic does not permit the conditional operator. Propose a linear-time method for removing them. Note that the conditional operators may be nested.

6.6 Bibliographic Notes

Tools and Applications

Bit-vector arithmetic was identified as an important logic for verification and equivalence checking in the hardware industry in [263]. The notation we use to annotate the type of the bit-vector expressions is taken from [50].

Early decision procedures for bit-vector arithmetic can be found in tools such as SVC [16] and ICS [113]. ICS used BDDs in order to decide properties of arithmetic operators, whereas SVC was based on a combination of a canonizer and a solver [18]. SVC has been superseded by CVC, and then CVC-Lite [14] and STP, both of which use a propositional SAT solver to decide the satisfiability of a circuit-based flattening of a bit-vector formula. ICS was superseded by Yices, which also uses flattening and a SAT solver.

Bit-vector arithmetic is now primarily used to model the semantics of programming languages. Cogent [75] decides the validity of ANSI-C expressions. ANSI-C expressions are drawn from a fragment of bit-vector arithmetic, extended with pointer logic (see Chap. 8). Cogent and related procedures have many applications besides checking verification conditions. As an example, see [37, 38] for an application of Cogent to database testing. In addition to deciding the validity of ANSI-C expressions, C32SAT [51], developed by Brummayer and Biere, is also able to determine if an expression always has a well-defined meaning according to the ANSI-C standard.

Bounded model checking (BMC) is a common source of bit-vector arithmetic decision problems [33]. BMC was designed originally for synchronous models, as frequently found in the hardware domain, for example. BMC has been adopted in other domains that result in bit-vector formulas, for example, software programs given in ANSI-C [72]. Further applications for decision procedures for bit-vector arithmetic are discussed in Chap. 12.

Translation to Integer Linear Arithmetic

Translations to integer linear arithmetic as described in Sect. 6.5.3 have been used for bit-vector decision problems found in the hardware verification domain. Brinkmann and Drechsler [50] translated a fragment of bit-vector arithmetic into ILP and used the Omega test as a decision procedure for the ILP problem. However, the work in [50] was aimed only at the data paths, and thus did not allow a Boolean part within the original formula. This was mended by Parthasarathy et al. [218] using an incremental encoding similar to the one described in Chap. 3.

IEEE Floating-Point Arithmetic

Decision procedures for IEEE floating-point arithmetic are useful for generating tests for software that uses such arithmetic. A semantics for formulas using IEEE binary floating-point arithmetic is given in the definition of the SMT-LIB FPA theory. IEEE floating-point arithmetic can be flattened into propositional logic by using circuits that implement floating-point units. Beyond flattening, approaches based on incremental refinement [49] and interval arithmetic [45] have been proposed. A theory for SMT solvers is proposed in [47].

State-of-the-Art Solvers

Current state-of-the-art decision procedures for bit-vector arithmetic apply heavy preprocessing to the formula, but ultimately rely on flattening a formula to propositional SAT [54, 189, 136]. The preprocessing is especially beneficial if the formula also contains large arrays, for example, for modeling memories [118, 188], or very expensive bit-vector operators such as multiplication or division. A method for generating encodings that are particularly well-suited for BCP is explained in [46]. The bit-vector category in the 2015 SMT competition was won by Boolector [52], which features an efficient decision procedure for the combination of bit-vector arithmetic with arrays.

6.7 Glossary

The following symbols were used in this chapter:

Symbol	Refers to ...	First used on page ...
$c?a : b$	Case split on condition c	135
λ	Lambda expressions	137
$bvec_l$	Set of bit vectors with l bits	138
$\langle\cdot\rangle_U$	Number encoded by binary encoding	139
$\langle\cdot\rangle_S$	Number encoded by two's complement	140
$A(\varphi)$	Set of atoms in φ	142
$T(\varphi)$	Set of terms in φ	143
c_i	Carry bit i	144
$[\![b]\!]$	Result of translation of bit-vector term b into linear arithmetic	152