On the Complexity of Fixed-Size Bit-Vector Logics with Binary Encoded Bit-Width

Gergely Kovácsznai, Andreas Fröhlich, Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract

Bit-precise reasoning is important for many practical applications of Satisfiability Modulo Theories (SMT). In recent years, efficient approaches for solving fixed-size bit-vector formulas have been developed. From the theoretical point of view, only few results on the complexity of fixed-size bit-vector logics have been published. Some of these results only hold if unary encoding on the bit-width of bit-vectors is used.

In previous work [30], we have already shown that binary encoding adds more expressiveness to bit-vector logics, e.g. it makes fixed-size bit-vector logic without uninterpreted functions NExpTime-complete.

In this paper, we revisit our complexity results for fixed-size bit-vector logics with binary encoded bit-width and go into more detail when specifying the underlying logics and presenting our proofs.

We also extend our previous work by analyzing commonly used bit-vector operations showing how they can be represented by a minimal set of so-called base operations. We further give a better insight in where the additional expressiveness of binary encoding comes from by proposing some new complexity results for bit-vector logics with a restricted set of base operations or certain conditions on the bit-width.

1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification.

Examples of state-of-the-art SMT solvers with support for bit-precise reasoning are Booletor [6], MathSAT [9], STP [23], Z3 [17], and Yices [19].

Syntax and semantics of fixed-size bit-vector logics do not differ much in the literature [16, 5, 10, 20]. Concrete formats for specifying bit-vector problems also exist, e.g. the SMT-LIB format [3] or the BTOR format [7].

Working with non-fixed-size bit-vectors has been considered for instance in [5, 1], and more recently in [11], but will not be further discussed in this paper. Most industrial applications (and examples in the SMT-LIB) have fixed bit-width.

We investigate the complexity of solving fixed-size bit-vector formulas. Some papers propose such complexity results, e.g. in [4] the authors consider quantifier-free bit-vector logic and give an argument for NP-hardness of its satisfiability problem. In [10], a sublogic of the previous one is claimed to be NP-complete. Interestingly, in [11] there is a claim about the full quantifier-free bit-vector logic being NP-complete, however the proposed decision procedure confirms this claim only if the bit-widths of the bit-vectors in the input formula are written/encoded in unary form. In [44, 43], the quantified case is addressed, and the satisfiability of this logic with

---

*This work is partially supported by FWF, NFN Grant S11408-N23 (RiSE).
uninterpreted functions is proved to be \textit{NExpTime-complete}. However, the proof, similarly to the decision procedure in [11], only holds if we assume unary encoded bit-widths.

A workshop version of our paper already appeared as [30]. Apart from this, we are not aware of any work that investigates how the particular encoding of the bit-widths in the input affects complexity (as an exception, see [14, Page 239, Footnote 3]). In practice, a more natural and exponentially more succinct \textit{logarithmic} encoding is used, such as in the SMT-LIB, the BTOR, and the Z3 format. We investigate how complexity varies if we consider either a unary or a logarithmic, actually without loss of generality, \textit{binary encoding}.

The present paper extends our previous work in several ways. After giving a motivation for the use of binary encoded bit-vector logics, we will specify various fixed-size bit-vector logics in detail. While our previous paper was referring to the common syntax and semantics used in other works, e.g. [16, 4, 5, 20, 10, 7], we now want to provide self-contained descriptions for the bit-vector logics we are considering. Therefore, we will introduce syntax and semantics for bit-vector logics including a set of base operations and give examples of how to reduce other common operations to elements of this set. We will show that slicing, concatenation, shift, and multiplication all are equally powerful, and while it is enough to include one to the set of base operations, we will later prove that it is at the same time also necessary in order not to lose expressiveness.

After these preliminary definitions, we will propose reworked versions of our complexity proofs for most of the introduced bit-vector logics as already proposed in [30]. Although our previous proofs of course still hold, we will make use of our newly introduced concepts and definitions to present those proofs in a clearer, easier-to-read, way.

Finally, we will again look at certain restrictions on the bit-vector logics with binary encoded bit-width that will cause the logics to stay in lower complexity classes. While we already considered such a restriction on the bit-width in [30], we will now also show that for quantified formulas, a bit-width restriction for the universal variables, and for quantifier free formulas, removing slicing from the set of base operations, will have a similar effect on bit-vector logics with binary encoded bit-width. Also, we give further benchmark problems that are not bit-width bounded and can directly be constructed out of our definition of bit-vector operations. These benchmark problems can also be used to experimentally verify that our proposed translations for the various operations are correct.

The appendix contains examples that make some definitions and proofs easier to understand.

## 2 Motivation

In practice, state-of-the-art bit-vector solvers rely on rewriting and bit-blasting. The latter is defined as the process of translating a bit-vector resp. word-level description into a bit-level circuit, as in hardware synthesis. The result can then be checked by a (propositional) SAT solver. We give an example why bit-blasting is not polynomial in general. Consider checking commutativity of bit-vector addition in SMT2 syntax for two bit-vectors of size one million:

```smt
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(assert (distinct (bvadd x y) (bvadd y x)))
```

Written to a file, this formula can be encoded with 138 bytes. Using a version of our SMT solver Boolector with all rewriting optimizations switched off (except for structural hashing), bit-blasting produces a circuit of size 103 MB encoded in the actually rather compact AIGER.
format. Tseitin transformation results in a CNF in DIMACS format of size 1 GB (actually 1040 MB). A bit-width of 10 million bits can be represented by two more bytes in the original SMT2 format but could not be bit-blasted anymore with our tool-flow (due to integer overflow).

As this example shows, checking bit-vector logics through bit-blasting cannot be considered to be a polynomial reduction, which also disqualifies bit-blasting as a sound way to prove that the decision problem for (quantifier-free) bit-vector logics is in NP. We show that deciding bit-vector logics, even without quantifiers, is much harder. It turns out to be NExpTime-complete.

Informally speaking, we show that moving from unary to binary encoding for bit-widths increases complexity exponentially and that binary encoding has at least as much expressive power as quantification. However, we give a sufficient condition for bit-vector problems to remain in the “lower” complexity class, when moving from unary to binary encoding. We call this bit-width bounded problems. For such problems it does not matter whether bit-width is encoded unary or binary. The same effect happens when we only use a certain subset of bit-vector operations in a formula.

3 Preliminaries

3.1 Fixed-Size Bit-Vector Logics

In this section, we specify the bit-vector logics on which we focus our investigations. First, we are going to define bit-vector terms, together with the set of base bit-vector operations, namely bitwise negation, bitwise and, equality, and slicing. In Sec. 3.2 we will show how other common bit-vector operations can be expressed by the use of these base ones. In the subsequent definition, we also address the use of uninterpreted functions.

Definition 1 (Term). A bit-vector term \( t \) of bit-width \( n \) (\( n \in \mathbb{N}, n \geq 1 \)) is denoted by \( t^{[n]} \). A term is defined inductively as follows:

<table>
<thead>
<tr>
<th>Term Type</th>
<th>Condition</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit-vector constant:</td>
<td>( c^{[n]} ) ( c \in \mathbb{N}, 0 \leq c &lt; 2^n )</td>
<td>( n )</td>
</tr>
<tr>
<td>bit-vector variable:</td>
<td>( x^{[n]} ) ( x ) is an identifier</td>
<td>( n )</td>
</tr>
<tr>
<td>bitwise negation:</td>
<td>( \sim t^{[n]} ) ( t^{[n]} ) is a term</td>
<td>( n )</td>
</tr>
<tr>
<td>bitwise and:</td>
<td>( (t_1^{[n]} &amp; t_2^{[n]}) ) ( t_1^{[n]} ) and ( t_2^{[n]} ) are terms</td>
<td>( n )</td>
</tr>
<tr>
<td>equality(^1)</td>
<td>( (t_1^{[n]} = t_2^{[n]}) ) ( t_1^{[n]} ) and ( t_2^{[n]} ) are terms</td>
<td>1</td>
</tr>
<tr>
<td>slice:</td>
<td>( t^{[n]}[i:j] ) ( t^{[n]} ) is a term, ( n &gt; i \geq j \geq 0 )</td>
<td>( i - j + 1 )</td>
</tr>
<tr>
<td>uninterpreted function:</td>
<td>( f^{[n]}(t_1^{[n_1]}, \ldots, t_k^{[n_k]}) ) ( f ) is an identifier, ( t_1^{[n_1]}, \ldots, t_k^{[n_k]} ) are terms</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Let us emphasize that, in a term, bit-widths are specified explicitly only for bit-vector constants, bit-vector variables, and uninterpreted functions. In any other cases the bit-width is implicit, i.e., it can calculated from the bit-widths of the operands of operators.

The terms defined so far only use what we called base operations. We also want to allow the use of other operations. All operations we did not explicitely use to define bit-vector terms

\(^1\) Consider the bit-vector operation equality as \( \texttt{bvcomp} \) in the SMT-LIB logic QF_BV, which is also referred as “bitwise equality”.

3


so far will be considered as non-base operations. Bit-vector terms can also consist of non-base operations:

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$op(t_1^{[n_1]}, \ldots, t_k^{[n_k]}, i_1, \ldots, i_l)$</td>
<td>$op$ is a bit-vector operation,</td>
<td>$f(op, n_1, \ldots, n_k, i_1, \ldots, i_l)$, for some function $f$</td>
</tr>
<tr>
<td>$t_1^{[n_1]}, \ldots, t_k^{[n_k]}$ are terms, $i_1, \ldots, i_l \in \mathbb{N}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In Sec. 3.2 we will show that most non-base operations can be expressed (polynomially) by base operations. Next, we define bit-vector formulas, addressing quantification as well.

**Definition 2 (Formula).** $Q, t^{[1]}$ is a bit-vector formula where $t^{[1]}$ is a bit-vector term, $Q$ is a quantifier prefix $Q_1 x_1^{[n_1]} Q_2 x_2^{[n_2]} \ldots Q_k x_k^{[n_k]}$, each $Q_i \in \{\forall, \exists\}$, each $x_i^{[n_i]}$ is a bit-vector variable, and $t$ does not contain any variable other than $x_1, \ldots, x_k$. We call $t$ the matrix of the formula.

We assume that variables and uninterpreted functions are identified by their unique names. In a formula, therefore, each variable and each uninterpreted function must be used in a consistent way, regarding its bit-width and the bit-widths of its arguments.

In the following, we may omit explicit bit-widths and parentheses if they can be concluded from the context. If only existential quantifiers appear in a formula, we may omit the quantifier prefix and refer such a formula as a quantifier-free one.

In literature, most of the approaches distinguish between a bit-vector level and a Boolean level within a bit-vector formula, by allowing only relational operators at the lowest Boolean level [15] [4] [8] [11] [20]. By considering our bitwise operations ($\sim$, $\&$) in the Boolean case (i.e., for bit-width 1) as logical connectives ($\neg$, $\land$) the same separation of a Boolean level and a bit-vector level can be made in any bit-vector formula. Notice, however, that our sole relational operator, equality, can occur not only at the lowest Boolean level, but even below that, due to Definition 1 which allows equalities to be nested. Notice furthermore that not only equalities can occur at the lowest Boolean level, but any terms of bit-width 1. In order to be compatible with the above-mentioned two-level approaches, we introduce a normal form for bit-vector formulas as follows. We call a bit-vector formula flat if all the logical connectives are applied only to equalities and equalities are not nested. Any bit-vector formula can be flattened by exhaustively applying the following two rewriting rules:

1. If $t^{[1]}$ is a term at the lowest Boolean level and is not an equality, then rewrite it to $(t^{[1]} = 1^{[1]})$.
2. If the equality $(t_1^{[n_1]} = t_2^{[n_2]})$ is nested in another equality, then rewrite it to $x^{[1]}$ and conjunct $(x^{[1]} \iff (t_1^{[n_1]} = t_2^{[n_2]}))$ with the matrix of the formula, where $x$ is a new existential variable, quantified within the quantifier scopes of all the variables occurring in $t_1$ and $t_2$.

In the rest of the paper, we assume only flat bit-vector formulas.

A bit-vector formula can be skolemized in the standard way, i.e., by replacing the existential variables with new uninterpreted functions. A model for a bit-vector formula $\phi$ is a mapping of functions to all the uninterpreted functions in the Skolemized form of $\phi$; by $[\phi]_\sigma$, we mean the interpretation of $\phi$ under the model $\sigma$. A bit-vector formula $\phi$ is satisfiable iff there exists a model $\sigma$ for $\phi$ such that $[\phi]_\sigma = 1$.

---

2 We intend also to use the logical connectives $\lor$, $\Rightarrow$ (implication), and $\Leftrightarrow$ (bi-implication).
In this paper, investigate the following 4 bit-vector logics, which we denote in the common way:

- **QF_BV**: quantifier-free bit-vector formulas without uninterpreted functions;
- **QF_UFBV**: quantifier-free bit-vector formulas allowing uninterpreted functions;
- **BV**: bit-vector formulas allowing quantification, but no uninterpreted functions;
- **UFBV**: bit-vector formulas allowing quantification and uninterpreted functions.

In order to distinguish between logics that use unary resp. binary numbers in formulas, we introduce notations in the following definition. Notice that in bit-vector formulas a number can only be of 3 sorts (c.f. Definition 1): a) a bit-width, b) the value \( c \) of a bit-vector constant, or c) the index \( i \) or \( j \) of a slicing.

**Definition 3** (Logic with unary resp. binary encoding). Given a bit-vector logic \( B \), let \( B_1 \) resp. \( B_2 \) denote the logic \( B \) using unary resp. binary encoding on all the numbers in formulas.

In Section 4, we are going to investigate the complexity of the decision problem for \( QF_BV_1 \), \( QF_UFBV_1 \), \( BV_1 \), \( UFBV_1 \), \( QF_BV_2 \), \( QF_UFBV_2 \), \( BV_2 \), and \( UFBV_2 \).

Given a bit-width \( n \), we are going to refer to the ceiling of its logarithm to base 2 and the nearest power of 2 (which is not less than \( n \)) quite frequently. For the sake of simplicity, let us introduce the following notations:

\[
\hat{L}_n := \lceil \log_2 n \rceil \\
\hat{n} := 2^{\hat{L}_n}
\]

We now want to define the size of bit-vector formulas.

**Definition 4** (Size). Given a bit-vector logic \( B \) and a formula \( \phi \in B \), where \( \phi := Q_1 x_1^{[n_1]} P_1 Q_2 x_2^{[n_2]} P_2 \ldots Q_k x_k^{[n_k]} P_k t^{[l]} \). The size of \( \phi \) is defined as \( |x_1^{[n_1]}| + \ldots + |x_k^{[n_k]}| + |t^{[l]}| \).

The expression \( |t^{[n]}| \) denotes the size of a term \( t^{[n]} \) and is defined as follows:

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit-vector constant: ( [c^{[n]}] )</td>
<td>( enc_B(c) + enc_B(n) )</td>
</tr>
<tr>
<td>bit-vector variable: ( [v^{[n]}] )</td>
<td>( 1 + enc_B(n) )</td>
</tr>
<tr>
<td>uninterpreted function: ( [f^{[n]}(t_1^{[n_1]}, \ldots, t_k^{[n_k]})] )</td>
<td>( 1 +</td>
</tr>
<tr>
<td>operation: ( [op(t_1^{[n_1]}, \ldots, t_k^{[n_k]}, i_1, \ldots, i_l)] )</td>
<td>( n + 1 ), if ( B ) uses unary encoding ( L(n + 1) + 1 ), if ( B ) uses binary encoding</td>
</tr>
<tr>
<td>natural number: ( enc_B(n) )</td>
<td>( enc_B(i_1) + \ldots + enc_B(i_l) )</td>
</tr>
</tbody>
</table>

Note that *operation* in our definition for the size of a bit-vector formula includes non-base operations as well as base operations.
3.2 Syntactic Sugar

In the subsequent sections, we are going to show how to define common bit-vector operations by the use of the so-called base operations (cf. Definition 1). For each operation, we give a translation which is polynomial in the formula size. Since, as we have proposed, even the case when formulas may contain binary numbers is addressed, we need to propose translations that are logarithmic in bit-width. For instance, in Sec. 3.2.7 we will give a translation of multiplication \( t_1[n] \cdot t_2[n] \) to the base operations, by increasing the formula size by only a factor of \( \log_2 n \).

For proposing a translation, we will use the following form:

\[
\text{term}_1 \downarrow \text{term}_2 \\
\text{where} \\
\text{formula}_1 \\
\vdots \\
\text{formula}_n
\]

By this description, we mean to replace a term \( \text{term}_1 \) in any formula \( F \) with \( \text{term}_2 \), and simultaneously to add \( \text{formula}_1, \ldots, \text{formula}_n \) to \( F \) (i.e. to conjunct \( \text{formula}_1 \land \cdots \land \text{formula}_n \) with it). We call \( \text{formula}_1, \ldots, \text{formula}_n \) the assertions in the definition.

It is important to specify in advance that all variables that do not occur in \( \text{term}_1 \), but do occur in any of \( \text{term}_2, \text{formula}_1, \ldots, \text{formula}_n \), are assumed to be new existential variables to \( F \), quantified within the quantifier scopes of all the variables occurring in \( \text{term}_1 \).

3.2.1 Bitwise Operations

In the following table, we show how to express two non-base bitwise operations. Other bitwise operations (e.g. nand, nor, xnor) can be expressed in a similar fashion.

<table>
<thead>
<tr>
<th>term</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1[n] \mid t_2[n] )</td>
<td>( n )</td>
</tr>
<tr>
<td>( \uparrow (\neg t_1 \land \neg t_2) )</td>
<td></td>
</tr>
<tr>
<td>( t_1[n] \oplus t_2[n] )</td>
<td>( n )</td>
</tr>
<tr>
<td>( \uparrow (ts_1 \land \neg ts_2) \mid (\neg ts_1 \land ts_2) ) where ( ts_1[n] = t_1 ) ( ts_2[n] = t_2 )</td>
<td></td>
</tr>
</tbody>
</table>

The newly introduced variables \( ts_1 \) and \( ts_2 \) can be considered as Tseitin variables for the terms \( t_1 \) and \( t_2 \), respectively. In each subsequent translation, we will introduce such Tseitin variables whenever any of the operands occurs more than once during the translation, in order to avoid exponential blow-up.

\(^3\) Base operation in SMT-LIB.
3.2.2 Indexing and Concatenation

Indexing is about reading one single bit from a bit-vector and can naturally be expressed by slicing. The concatenation of two bit-vectors can be characterized by the fact that slicing applied to the result yields the original bit-vectors.

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t[n][i])</td>
<td>(0 \leq i &lt; n)</td>
<td>1</td>
</tr>
<tr>
<td>(t^{}<em>{1}[m] \circ t^{}</em>{2}[n])</td>
<td>(m + n)</td>
<td></td>
</tr>
</tbody>
</table>

### Concatenation

\[t^{}_{1}[m] \circ t^{}_{2}[n]\]

where

\[t^{}_{1} = x[n + m - 1 : n]\]
\[t^{}_{2} = x[n - 1 : 0]\]

3.2.3 Extensions

Zero extension and sign extension are common operations, like in SMT-LIB or in \([20]\). We are also going to refer to them as unsigned extension and signed extension, respectively. The operation \(\text{ite}\) denotes functional if-then-else, as it is detailed in Sec. 3.2.4.

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{ext}^{}_{u}(t[n], m))</td>
<td>(m \geq 0)</td>
<td>(m + n)</td>
</tr>
<tr>
<td>(\text{ext}^{}_{s}(t[n], m))</td>
<td>(m \geq 0)</td>
<td>(m + n)</td>
</tr>
</tbody>
</table>

### Zero Extension

\[\text{ext}^{}_{u}(t[n], m) = \begin{cases} t & \text{if } m = 0 \\ 0[m] \circ t & \text{otherwise} \end{cases}\]

### Sign Extension

\[\text{ext}^{}_{s}(t[n], m) = \begin{cases} t & \text{if } m = 0 \\ \text{ite} \left( \begin{array}{l} t_s[n-1] \\ \sim 0[m] \circ t_s \\ 0[m] \circ t_s \end{array} \right) & \text{otherwise} \end{cases}\]

where

\[t_s[n] = t\]

3.2.4 Helper Operations

In this section, we introduce a few helper operations which we are going to use in the subsequent sections in order to make complicated translations (like the ones in Sec. 3.2.6 or Sec. 3.2.7) easier to formalize.

**Functional if-then-else.** This is a common operation, e.g. used in SMT-LIB or in Boolector \([7]\). Its first operand \(t_1\) must be a Boolean term.
if-then-else: \[ \text{ite} \left( t_1[1], t_2[n], t_3[n] \right) \]

where

- \( ts_1[1] = t_1 \)
- \( ts_1 \Rightarrow x = t_2 \)
- \( \neg ts_1 \Rightarrow x = t_3 \)

Self-concatenation. This operation receives a bit-vector term \( t^{(2^m)} \) and concatenates it with itself until a bit-vector of bit-width \( 2^n \) has been constructed. In the table below, we propose a \((n - m)\)-step approach for this operation.

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{selfconcat} \left( t^{(2^m)}, 2^n \right)</td>
<td>( m \leq n )</td>
<td>( 2^n )</td>
</tr>
</tbody>
</table>

\[ \uparrow \quad x_n \quad \text{where} \quad x_m^{(2^n)} = t \]
\[ x_i^{(2^1)} = x_{i-1} \circ x_{i-1} \quad \text{for all} \quad m < i \leq n \]

Binary magic numbers. The next helper operation produces specific bit-vector constants, the so-called binary magic numbers [21] (see also [42, Chpt. 7]). A binary magic number can be written in the following form:

\[
\begin{array}{c}
0 \ldots 0 \\
2^m
\end{array}
\begin{array}{c}
1 \ldots 1 \\
2^m
\end{array}
\begin{array}{c}
0 \ldots 0 \\
2^m
\end{array}
\begin{array}{c}
1 \ldots 1 \\
2^m
\end{array}
\]

Such binary numbers, as bit-vectors, will be used several purposes in the rest of the paper, e.g. for expressing multiplication in Sec. 3.2.7 or in the proof of our central theorem in Sec. 4.2.1. Notice that a binary magic number can be produced by self-concatenating the bit-vector \( \begin{array}{c} 0 \ldots 0 \\
2^m
\end{array} \begin{array}{c} 1 \ldots 1 \\
2^m
\end{array} \begin{array}{c} 0 \ldots 0 \\
2^m
\end{array} \begin{array}{c} 1 \ldots 1 \\
2^m
\end{array} \).

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{binmagic} \left( 2^m, 2^n \right)</td>
<td>( m &lt; n )</td>
<td>( 2^n )</td>
</tr>
</tbody>
</table>

\[ \uparrow \quad \text{selfconcat} \left( \left( 0^{(2^m)} \circ 0^{(2^m)} \right), 2^n \right) \]

Remark 5. While the previous approach requires logarithmic many bit-vector base operations, there is a different, arithmetic-based (c.f. Sec. 3.2.7), approach that uses only a constant number of operations.

As known in literature [28], binary magic numbers can arithmetically be calculated in the following way (actually as sum of a geometric series):

\[
\text{binmagic} \left( 2^m, 2^n \right) := \frac{2^{(2^n)} - 1}{2^{(2^m)} + 1}
\]
In order to reformulate this definition in terms of bit-vectors, a) the numerator can be written as \(~0[2^n]\), b) \(2^m\) as \(1 \ll 2^m\), and c) the result as a bit-vector variable \(b[2^n]\):

\[
b = \sim0[2^n] / u ((1 \ll 2^m) + 1)
\]

This definition requires the use of unsigned division, which, however, can easily be eliminated by rewriting the above equation to

\[
b \cdot ((1 \ll 2^m) + 1) = \sim0[2^n]
\]

**Multiplication** in this equation can then be eliminated by rewriting it as follows:

\[
(b \ll 2^m) + b = \sim0[2^n]
\]

In this equation, there are only two non-base operations: **shift left by constant** and **addition.** As detailed in Sec. 3.2.6 and Sec. 3.2.7, both can be expressed by constant many base operations.

**Remark 6.** A third approach for generating the binary magic numbers only uses **shifts by 1**, bitwise operations and equality. This will be of importance in Thm. 32, because **shifts by 1** can be expressed using **addition**.

For \(0 \leq m < n\), add the following equation to the formula:

\[
b'_m[2^n] = \left( \bigwedge_{0 \leq i < m} b_i[2^n] \right) \oplus b_m[2^n]
\]

Consider all bit-vectors \(b_0[2^n], \ldots, b_m[2^n]\) as a matrix \(B^{2^n \times m}\) and \(b'_0[2^n], \ldots, b'_m[2^n]\) as a matrix \(B'^{2^n \times m}\). If each row of \(B\) is interpreted as a number \(0 \leq c < 2^m\) in binary representation, the corresponding row of \(B'\) is equal to \(c + 1\).

Now, again for \(0 \leq m < n\), add another constraint:

\[
b'_m[2^n] = b_m[2^n] \ll 1[2^n]
\]

Together with the previous \(n\) equations, those \(n\) constraints force the rows of \(B\) to represent an enumeration of all binary numbers \(0 \leq c < 2^m\), starting with 0. Therefore, the columns, i.e. the individual bit-vectors \(b_0[2^n], \ldots, b_n[2^n]\), exactly define the binary magic numbers: **binmagic**(\(2^m, 2^n\)) := \(b_m[2^n]\).

Of course, all \(b'_m[2^n], 0 \leq m < n\), can be eliminated and the two sets of constraints can be replaced by a single set of constraints:

\[
\left( \bigwedge_{0 \leq i < m} b_i[2^n] \right) \oplus b_m[2^n] = b_m[2^n] \ll 1[2^n]
\]

**Expand.** When proposing a translation for multiplication in Sec. 3.2.7 we will apply a special type of operation which we call **expanding** a bit-vector \(t[2^m]\) to the bit-width \(2^n\), where \(m \leq n\). The operation is about “multiplying” each bit of \(t\) into a bit group of size \(2^{n-m}\). The resulting bit-vector can be visualized as follows:

\[
\underbrace{t[2^m - 1] \ldots t[2^m - 1]}_{2^{n-m}} \underbrace{t[2^m - 2] \ldots t[2^m - 2]}_{2^{n-m}} \ldots \underbrace{t[0] \ldots t[0]}_{2^{n-m}}
\]
To be precise, this operation can be expressed in the following form:

\[ \text{selfconcat}(t[2^m - 1], 2^{n-m}) \circ \ldots \circ \text{selfconcat}(t[0], 2^{n-m}) \]

However, the translation above is linear (i.e. not logarithmic) in the bit-width \(2^m\). Therefore, let us propose a logarithmic translation, which is based on the generalization of a bit-vector operation called \textit{half-shuffle} [42, Chpt. 7]. This generalized variant produces the following bit-vector:

\[
\begin{array}{cccc}
0 & \ldots & 0 & t[2^m - 1] \\
2^{n-m-1} & \ldots & 2^{n-m-1} & t[2^m - 2] \\
& \ldots & 2^{n-m-1} & \ldots \\
& \ldots & 0 & t[0] \\
\end{array}
\]

In the initialization step, we apply zero extension to \(t\). Then, in \(m\) steps, we shuffle smaller and smaller bit groups in our bit-vector. In the 1st step, the two halves (i.e. \(2^{m-1}\)-bit groups) are shuffled. In the 2nd step, the halves of all the previously shuffled halves (i.e. \(2^{m-2}\)-bit groups), and so on. In the last step, we shuffle single bits, and this is how to put each bit at its destination.

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{halfshuffle} (l^{[2^m]}, 2^n)</td>
<td>(m \leq n)</td>
<td>(2^n)</td>
</tr>
</tbody>
</table>

As it can be seen in the table above, in the \(i\)th step we a) shift our current bit-vector left by the constant \(2^{n-i} - 2^{m-i}\), b) merge it with the original bit-vector, by using bitwise or, c) and we mask some unnecessary bit groups out, by using a binary magic number. For an example see Appendix A.

Now, it is quite easy to express \textit{expanding} by the use of half-shuffling in the first step, as it can be seen in the table below. In the next \(n - m\) steps, we copy larger and larger non-zero bit groups, by using left shift and bitwise or.

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{expand} (l^{[2^n]}, 2^n)</td>
<td>(m \leq n)</td>
<td>(2^n)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
x_m & & & \\
\text{where} & \text{ext}_u(t, 2^n - 2^m) & \text{binmagic}(2^{n-i}, 2^n) & \\
\end{array}
\]

3.2.5 Relational Operations

Apart from our base relational operation, \textit{equality}, other relational operations are also commonly used (e.g. \textit{less than} and \textit{greater than} equal).

Let us first express the \textit{unsigned less than} operation, by using the following simple idea: \(t_1^{[n]}\) is less than \(t_2^{[n]}\) if there is a bit position \(i\) such that \(t_1[i] = 0\) and \(t_2[i] = 1\), and, furthermore, \(t_1[n-1 : i + 1] = t_2[n-1 : i + 1]\). In the table below, we are proposing a \(L_n\)-step approach.
for finding such an $i$. In each step, the two current bit-vectors are halved. If their higher-order halves are equal then we continue the process with their lower-order halves; otherwise we continue with their higher-order halves. In order to make the approach easier to read, let us introduce the following notations for the higher-order half and the lower-order half of a bit-vector of bit-width $2^k$, $k > 0$:

$$hi(t[2^k]) := t[2^k - 1 : 2^{k-1}]$$

$$lo(t[2^k]) := t[2^{k-1} - 1 : 0]$$

<table>
<thead>
<tr>
<th>term</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1[n] &lt;_u t_2[n]$</td>
<td>1</td>
</tr>
<tr>
<td>$\uparrow \neg x_{Ln} \land y_{Ln}$</td>
<td></td>
</tr>
<tr>
<td>where</td>
<td></td>
</tr>
<tr>
<td>$x_0[n] = ext_u(t_1, \hat{n} - n)$</td>
<td></td>
</tr>
<tr>
<td>$y_0[n] = ext_u(t_2, \hat{n} - n)$</td>
<td></td>
</tr>
</tbody>
</table>
| $x_i[\hat{n}/2] = ite\left(\begin{array}{l}
hi(x_{i-1}) = hi(y_{i-1}), \\
lo(x_{i-1}), \\
hi(x_{i-1})
\end{array}\right)$ for all $0 < i \leq Ln$ | |
| $y_i[\hat{n}/2] = ite\left(\begin{array}{l}
hi(x_{i-1}) = hi(y_{i-1}), \\
lo(y_{i-1}), \\
hi(y_{i-1})
\end{array}\right)$ | |

When checking signed less than, also the special case when $t_1$ is negative and $t_2$ is positive must be considered. If they have the same polarity (even if both are negative), unsigned less than can be used.

<table>
<thead>
<tr>
<th>term</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1[n] &lt;_s t_2[n]$</td>
<td>1</td>
</tr>
</tbody>
</table>
| $\uparrow (ts_1[n - 1] \land \neg ts_2[n - 1]) \lor$
| $(ts_1[n - 1] = ts_2[n - 1] \land ts_1 <_u ts_2)$ | |
| where | |
| $ts_1[n] = t_1$ | |
| $ts_2[n] = t_2$ | |

Having these two relations and equality in hand, other common relational operations, like unsigned or signed less than or equal, greater than, and greater than or equal, can easily be expressed.

### 3.2.6 Shifts

Shifts are usually considered as base operations, e.g. in SMT-LIB, in Boolector [7], and in [20]. We show that they can be (logarithmically) translated to our base operations.

**Shift by constant.** First, we show how to express logical (left or right) shift by a constant on the right-hand side. Let us note that there are approaches [8] where shifts only by constants are addressed. Such shifts can be expressed by the use of slicing (and concatenation), as follows.
Complexity of Bit-Vector Logics with Binary Encoded Bit-Width
Kovácsnai, Fröhlich, and Biere

Shift by any term. A shift by any term can be expressed by the use of shifts by constants. This can be done by an approach called barrel shift. Given the two operands $t_1$ and $t_2$ of bit-width $n$, the shift can be done in $L_n$ steps. In the $i$th step, we check the $i$th bit of $t_2$, and if it is 1 then we shift $t_1$ by $2^i$.

Of course, this approach only needs to be applied if the value of $t_2$ is less than $n$. Otherwise the result of the shift must be a constant 0 bit-vector.

In the following table, we show how to do left shift; logical (i.e. unsigned) right shift can be done in a similar fashion. We also propose a translation for arithmetic (i.e. signed) right shift.

<table>
<thead>
<tr>
<th>Shift left: $t_1[n] \ll c[n]$</th>
<th>bit-width $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$ $\begin{cases} t &amp; \text{if } c = 0 \ t[n - c - 1 : 0] \circ 0[c] &amp; \text{if } 0 &lt; c &lt; n \ 0[n] &amp; \text{otherwise} \end{cases}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical shift right: $t_1[n] \gg u c[n]$</th>
<th>bit-width $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$ $\begin{cases} t &amp; \text{if } c = 0 \ 0[c] \circ t[n - 1 : c] &amp; \text{if } 0 &lt; c &lt; n \ 0[n] &amp; \text{otherwise} \end{cases}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Left shift: $t_1[n] \ll t_2[n]$</th>
<th>bit-width $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$ $\text{ite} \left( t_{s2} \ll u n, x_{i\ln}, 0[n] \right)$</td>
<td></td>
</tr>
<tr>
<td>where $t_{s2}[n] = t_2$</td>
<td></td>
</tr>
<tr>
<td>$x_0[n] = t_1$</td>
<td></td>
</tr>
<tr>
<td>$x_i[n] = \text{ite} \left( t_{s2}[i - 1], x_{i - 1} \ll 2^{i - 1}, x_{i - 1} \right)$ for all $0 &lt; i \leq L_n$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical shift right: $t_1[n] \gg u t_2[n]$</th>
<th>bit-width $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$ $t_{s1}[n - 1], \neg (\neg t_{s1} \gg u t_{s2}), t_{s1} \gg u t_{s2}$</td>
<td></td>
</tr>
<tr>
<td>where $t_{s1}[n] = t_1$</td>
<td></td>
</tr>
<tr>
<td>$t_{s2}[n] = t_2$</td>
<td></td>
</tr>
</tbody>
</table>
### 3.2.7 Arithmetic Operations

In this section, we focus on the four basic arithmetic operations, and we address also some commonly used variants (e.g. signed modulo).

Furthermore, we are going to propose how to detect overflow for these arithmetic operations. Detecting overflow means that the result of an arithmetic operation is too large (or too small) to be correctly represented in the resulting bit-vector. In the subseqent definitions of overflow detection operations, the symbol “…” may appear among the assertions. This means that some assertions are omitted, since they are the same as in the definition of the corresponding arithmetic operation.

#### Addition

Addition can be expressed by the use of constant many base operations. The $i$th bit of the sum of two bit-vectors can be calculated by performing $\text{xor}$ on their $i$th bits and the $i$th carry-in bit. Calculating the carry-in bit-vector is straightforward if we have the carry-out bit-vector in hand. A carry-out bit has to be set to 1 if and only if at least two of the three bits on which the aforementioned $\text{xor}$ is performed are set to 1.

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1[n] + t_2[n]$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

where

- $ts_1[n] = t_1$
- $ts_2[n] = t_2$
- $cin[n] = cout \ll 1$
- $cout[n] = (ts_1 \& ts_2) \mid (ts_1 \& cin) \mid (ts_2 \& cin)$

Detecting overflow for addition of unsigned integers is extremely easy by using our approach: we only need to check whether the most significant bit of the carry-out bit-vector is 1. In the signed case, an overflow occurs if and only if the operands have the same sign and the sum has the opposite sign [42, Chpt. 2-12]. As already mentioned, in the subsequent two definitions the symbol “…” denotes the same assertions as in the definition of addition (i.e. the ones that define $cin$ and $cout$).

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{overflow}_u(t_1[n] + t_2[n])$</td>
<td>1</td>
</tr>
</tbody>
</table>

where

- $cout[n-1]$

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{overflow}_s(t_1[n] + t_2[n])$</td>
<td>1</td>
</tr>
</tbody>
</table>

where

- $ts_1[n-1] \wedge ts_2[n-1] \wedge \neg \text{sign}_\text{sum}$
- $\neg ts_1[n-1] \wedge \neg ts_2[n-1] \wedge \text{sign}_\text{sum}$

$\text{sign}_\text{sum}[1] = (ts_1 \oplus ts_2 \oplus cin)[n-1]$
**Subtraction.** Subtraction can naturally be expressed by addition, however, we need a unary minus operation, which computes the two’s complement of a bit-vector.

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary minus</td>
<td>$-t[n]$</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
</tr>
<tr>
<td>$t \sim t + 1$</td>
<td></td>
</tr>
<tr>
<td>subtraction:</td>
<td>$t_1[n] - t_2[n]$</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
</tr>
<tr>
<td>$t \sim t_1 + -t_2$</td>
<td></td>
</tr>
</tbody>
</table>

**Overflow detection** for subtraction of unsigned integers corresponds to checking whether the first operand is less than the second operand. In the signed case, addition overflow detection can be applied. Finally, the operation unary minus only cause overflow in one case: if the operand is the least (negative) $n$-bit integer, i.e. it is represented by the bit sequence 10...0.

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned subtraction overflow:</td>
<td>$\text{overflow}_u (t_1[n] - t_2[n])$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
<tr>
<td>$t \sim t_1 \lessdot t_2$</td>
<td></td>
</tr>
<tr>
<td>unary minus overflow:</td>
<td>$\text{overflow}_s (-t[n])$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
<tr>
<td>$t \sim ts[n-1] \land ts[n-2:0] = 0[n-1]$ where $ts[n] = t$</td>
<td></td>
</tr>
<tr>
<td>signed subtraction overflow:</td>
<td>$\text{overflow}_s (t_1[n] - t_2[n])$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
<tr>
<td>$t \sim \text{overflow}_s (-ts_2) \lor \text{overflow}_s (t_1 + -ts_2)$ where $ts_2[n] = t_2$</td>
<td></td>
</tr>
</tbody>
</table>

**Multiplication.** Multiplication can be encoded by simulating the common "shift and add" algorithm for (unsigned or signed) integers. In a first step, one of the operands is multiplied independently by each digit of the other operand. Using base 2, this multiplication by a single digit can be expressed by a logical and-operation. Afterwards the results of the single-digit multiplications are shifted by the offset of the corresponding digit and finally added to give the result of the full multiplication.

While this approach is straightforward in a naive implementation, we have to ensure only logarithmic many operations in the bit-width are used in our encoding. To achieve this, we generate bit-vectors of quadratic size $\hat{n}^2$ out of our original operands of size $n$, applying self-concat to the first one and expand to the second one (cf. Sec. 3.2.4). Using bitwise and on the two new vectors, we directly get the results of all single-digit multiplications in one step. More precisely, the resulting bit-vector consists of $\hat{n}$ groups of $\hat{n}$ bits, each group representing the result of one single-digit multiplication. To add all $\hat{n}$ partial results, a binary addition algorithm is used. Iteratively pairs of neighbouring groups are shifted relative to each others’ offsets and then added to form one new group. The number of groups therefore is halved in each step, resulting in the final sum after $\log_2(\hat{n}) = \ln$ steps. For a detailed example, see also
Overflow detection for multiplication of \(n\)-bit integers is simple if we have access to the “higher-order” \(n\) bits of the product [42, Chpt. 2-12]. In our approach for multiplication, these bits are located within the slice \(x_{\text{Ln}}[2n - 1 : n]\). In the unsigned case, all these bits should be 0, otherwise overflow occurs.

The signed case is a little bit more complicated. First, the product has to be “corrected” in the following way [42, Chpt. 8-1]: if \(t_2\) is negative then \(2^n \cdot t_1\) needs to be subtracted from the product, which is actually the same as subtracting \(t_1\) from the “higher-order” \(n\) bits of the product. Analogously, if \(t_1\) is negative then \(t_2\) must be subtracted. Finally, we have to check whether the corrected “higher-order” \(n\) bits all equal to the sign of the product, otherwise overflow occurs.

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-width</th>
</tr>
</thead>
</table>
| \( t_1^{[n]} \cdot t_2^{[n]} \) | \( n \)

Overflow detection for multiplication of \(n\)-bit integers is simple if we have access to the “higher-order” \(n\) bits of the product [42, Chpt. 2-12]. In our approach for multiplication, these bits are located within the slice \(x_{\text{Ln}}[2n - 1 : n]\). In the unsigned case, all these bits should be 0, otherwise overflow occurs.

The signed case is a little bit more complicated. First, the product has to be “corrected” in the following way [42, Chpt. 8-1]: if \(t_2\) is negative then \(2^n \cdot t_1\) needs to be subtracted from the product, which is actually the same as subtracting \(t_1\) from the “higher-order” \(n\) bits of the product. Analogously, if \(t_1\) is negative then \(t_2\) must be subtracted. Finally, we have to check whether the corrected “higher-order” \(n\) bits all equal to the sign of the product, otherwise overflow occurs.

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-width</th>
</tr>
</thead>
</table>
| \( \text{overflow}_u(t_1^{[n]} \cdot t_2^{[n]}) \) | \( 1 \)

Overflow detection for multiplication of \(n\)-bit integers is simple if we have access to the “higher-order” \(n\) bits of the product [42, Chpt. 2-12]. In our approach for multiplication, these bits are located within the slice \(x_{\text{Ln}}[2n - 1 : n]\). In the unsigned case, all these bits should be 0, otherwise overflow occurs.

The signed case is a little bit more complicated. First, the product has to be “corrected” in the following way [42, Chpt. 8-1]: if \(t_2\) is negative then \(2^n \cdot t_1\) needs to be subtracted from the product, which is actually the same as subtracting \(t_1\) from the “higher-order” \(n\) bits of the product. Analogously, if \(t_1\) is negative then \(t_2\) must be subtracted. Finally, we have to check whether the corrected “higher-order” \(n\) bits all equal to the sign of the product, otherwise overflow occurs.

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-width</th>
</tr>
</thead>
</table>
| \( \text{overflow}_s(t_1^{[n]} \cdot t_2^{[n]}) \) | \( 1 \)

Overflow detection for multiplication of \(n\)-bit integers is simple if we have access to the “higher-order” \(n\) bits of the product [42, Chpt. 2-12]. In our approach for multiplication, these bits are located within the slice \(x_{\text{Ln}}[2n - 1 : n]\). In the unsigned case, all these bits should be 0, otherwise overflow occurs.

The signed case is a little bit more complicated. First, the product has to be “corrected” in the following way [42, Chpt. 8-1]: if \(t_2\) is negative then \(2^n \cdot t_1\) needs to be subtracted from the product, which is actually the same as subtracting \(t_1\) from the “higher-order” \(n\) bits of the product. Analogously, if \(t_1\) is negative then \(t_2\) must be subtracted. Finally, we have to check whether the corrected “higher-order” \(n\) bits all equal to the sign of the product, otherwise overflow occurs.

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-width</th>
</tr>
</thead>
</table>
| \( \text{overflow}_s(t_1^{[n]} \cdot t_2^{[n]}) \) | \( 1 \)

Division. The operations unsigned division and unsigned remainder are usually considered as base operations, e.g. in SMT-LIB, in Boolector [7], or in [20]. In the table below, we propose an approach for expressing them by the use of multiplication, addition, and unsigned less than. Notice that it is necessary to check whether (unsigned) overflow occurs when applying
multiplication and addition. In the definition of *unsigned remainder*, the assertions (denoted by “...”) are the same as in the definition of *unsigned division*.

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1[n] /_{\text{u}} t_2[n] )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

\[ \uparrow \ d \\
\text{where} \\
\begin{align*}
t_{s2}[n] &= t_2 \\
t_1 &= t_{s2} \cdot d[n] + r[n] \\
r &<_{\text{u}} t_{s2} \\
\neg \text{overflow}_u(t_{s2} \cdot d) \\
\neg \text{overflow}_u(t_{s2} \cdot d + r)
\end{align*} \]

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1[n] %_{\text{u}} t_2[n] )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

\[ \uparrow \ r \\
\text{where} \\
\ldots \]

*Signed division* and *signed remainder* can be expressed by applying their unsigned counterparts to the absolute value of the operands (c.f. SMT-LIB, or even [20], where the translation of these 2 operations is faulty). In the case of *signed division*, if the sign of the two operands is not the same, then the result of the unsigned division has to be negated. The same has to be done in the case of *signed remainder*, if the first operand is negative. In the corresponding definitions, the assertions are the same.

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1[n] /_{s} t_2[n] )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

\[ \uparrow \ \text{ite} \left( \begin{array}{c}
ts_{s1}[n - 1] = t_{s2}[n - 1] \\
\text{abs}_{t_1} /_{u} \text{abs}_{t_2} \\
-(\text{abs}_{t_1} /_{u} \text{abs}_{t_2})
\end{array} \right) \\
\text{where} \\
t_{s1}[n] = t_1 \\
t_{s2}[n] = t_2 \\
\text{abs}_{t_1}[n] = \text{ite}(t_{s1}[n - 1], -t_{s1}, t_{s1}) \\
\text{abs}_{t_2}[n] = \text{ite}(t_{s2}[n - 1], -t_{s2}, t_{s2})
\]

<table>
<thead>
<tr>
<th>Term</th>
<th>Bit-Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1[n] %_{\text{u}} t_2[n] )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

\[ \uparrow \ \text{ite} \left( \begin{array}{c}
ts_{s1}[n - 1] \\
-(\text{abs}_{t_1} \%_{u} \text{abs}_{t_2}) \\
\text{abs}_{t_1} \%_{u} \text{abs}_{t_2}
\end{array} \right) \\
\text{where} \\
\ldots \]

*Signed modulo* is another operation which is usually considered, e.g. in SMT-LIB or in Boolector [7]. While the sign of the remainder follows the first operand, the sign of the modulus follows the second operand (c.f. SMT-LIB).
Complexity of Bit-Vector Logics with Binary Encoded Bit-Width
Kovásznai, Fröhlich, and Biere

Among the various division operations, only signed division can cause an overflow (we do not address division by 0 here, which is extremely easy to check). Such an overflow can occur only if the value of the second operand is $-1$ [42, Chpt. 2-12], i.e. when we are about applying unary minus to the first operand.

3.3 QBF, DQBF, EPR, and QFPAbit

In this subsection, we will describe some logics that we will later use to derive hardness proofs for the complexity results of certain bit-vector classes. Particularly, this subsection deals with Quantified Boolean Formulas (QBF), Dependency Quantified Boolean Formulas (DQBF), Effectively Propositional Logic (EPR), and quantifier-free Presburger arithmetic with bitwise operations (QFPAbit). We will briefly introduce each of the mentioned classes and provide known complexity-results.

3.3.1 QBF and DQBF

Let $V$ be a set of propositional variables and let $\phi$ denote a Boolean Formula over the variables of $V$. The class of QBF, is the class of formulas obtained by adding quantifiers to Boolean Formulas. Each QBF $\psi$ can be written in Prenex Normal Form, i.e. as

$$\psi = Q \cdot \phi = \exists Q_0 \forall Q_1 \exists Q_2 \forall Q_3 \ldots \exists Q_{m-2} \forall Q_{m-1} \exists Q_m \cdot \phi, \quad m = 2n, \quad n \in \mathbb{N}$$

with $Q$ being the quantifier prefix and $\phi$ being a Boolean Formula which is also called the matrix of $\psi$. $Q_i$ is representing a set of variables and is called a quantifier scope. A variable $x \in Q_i$ depends on a variable $y \in Q_j$ iff $i > j$. This defines a total order on the variable dependencies of $\psi$. The decision problem for QBF is PSPACE-complete [35].

Instead of adding totally ordered quantifiers to a formula, it is also possible to extend Boolean Formulas by Henkin quantifiers [25]. Henkin quantifiers specify variable dependencies explicitly instead of using implicit dependencies defined by the quantifier order. This allows to define more general dependency constraints only requiring a partial order. Adding Henkin quantifiers to Boolean Formulas, results in the class of DQBF, as first defined in [37]. Again a DQBF $\psi'$ can always be expressed in Prenex Normal Form, i.e. as
\[ \psi' = Q' \cdot \phi = \forall u_1, \ldots, u_m \exists v_1(v_1, \ldots, v_{1,m_1}), \ldots, v_n(u_n, \ldots, u_{n,m_n}) \cdot \phi \]

with quantifier prefix \( Q' \) and matrix \( \phi \) over the variables \( V := U \cup E \), consisting of universal variables \( U = \{u_1, \ldots, u_m\} \) and existential variables \( E = \{e_1, \ldots, e_n\} \), with \( u_{i,j} \in U \) for all \( i \in \{1, \ldots, m\} \). In DQBF, existential variables can always be placed after all universal variables in the quantifier prefix, since the dependencies of a certain variable are explicitly given and not implicitly defined by the order of the prefix (in contrast to QBF).

The more general quantifier order makes DQBF more powerful than QBF and allows more succinct encodings. The decision problem for DQBF is \( \text{NExpTime}\)-complete \[37, 36\].

3.3.2 EPR

EPR, also known as the Bernays-Schönfinkel class, is a decidable fragment of first-order logic. It corresponds to the set of first-order formulas which contain a) no function symbol of arity greater than 0, and b) no existential quantifier within the scope of a universal quantifier.

Consequently, EPR clauses can be defined as follows. Let a set of (universally quantified) variables, a set of constant symbols (i.e. function symbols of arity 0), and a set of predicate symbols be given. Moreover, each predicate symbol is associated with a non-negative integer which we call its arity. A term is either a variable or a constant symbol. An atom is an expression of the form \( p(t_1, \ldots, t_n) \) where \( p \) is a predicate symbol of arity \( n \) and each \( t_i \) is a term. A literal is either an atom or the negation of an atom. A clause is a disjunction of literals. The decision problem for EPR is \( \text{NExpTime}\)-complete \[32\].

3.3.3 QFPAbit

By QFPAbit, we denote quantifier-free Presburger arithmetic with bitwise operations \[38\]. This is an expansion of quantifier-free Presburger arithmetic and bitwise operations are performed on the two’s complement encoding of the integers.

Given a set of variables \( V \). With \( x \in V \) and \( c \in \mathbb{Z} \), QFPAbit-terms \( T \) resp. QFPAbit-formulas \( F \) can be defined by

\[
T := c \mid x \mid (T + T) \mid (T \cdot T) \mid (T \lor T) \mid (T \land T)
\]

\[
F := T < 0 \mid T = T \mid \neg F \mid (F \lor F) \mid (F \land F)
\]

In \[38, 41\], some other relational operations and multiplication by constants are also permitted. It is straightforward to check that our definition only leads to polynomially larger QFPAbit-formulas (c.f. \[41, 40\]). Deciding QFPAbit is \( \text{PSpace}\)-complete \[40\].

4 Complexity

In this section, we discuss the complexity of deciding the bit-vector logics defined so far. We first summarize our results, and then give more detailed proofs for the new non-trivial ones.

Notice that all the base operations (c.f. Def. 1) can be bit-blasted polynomially in the bit-widths of their operands. Consequently the same holds for all the operations we have discussed in Sec. 3.2. This fact plays an important role in some of our propositions, e.g. in Prop. 7 or Prop. 9. All results of this section are also summarized in Tab. 1.
4.1 Logics With Unary Encoding

First, we consider unary encoding on the bit-widths in formulas. Without uninterpreted functions nor quantification, i.e., for $QF_{BV1}$, the following complexity result can be shown (for partial results and related work see also [4] and [10]):

Proposition 7. $QF_{BV1}$ is $NP$-complete.$^6$

Proof. By bit-blasting, $QF_{BV1}$ can be polynomially reduced to Boolean formulas, for which the satisfiability problem (SAT) is $NP$-complete. The other direction follows from the fact that Boolean formulas are actually $QF_{BV1}$ formulas with terms of bit-width 1. □

Adding uninterpreted functions to $QF_{BV1}$ does not increase complexity:

Proposition 8. $QF_{UFBV1}$ is $NP$-complete.

Proof. In a quantifier-free formula, uninterpreted functions can be eliminated by replacing each occurrence with a new bit-vector variable and adding (at most quadratic many) Ackermann constraints (see e.g. [31, Chapter 3.3.1]). Therefore, $QF_{UFBV1}$ can be polynomially translated to $QF_{BV1}$. The other direction follows from the fact that $QF_{BV1} \subseteq QF_{UFBV1}$. □

Adding quantifiers to $QF_{BV1}$ yields the following complexity (see also [14]):

Proposition 9. $BV1$ is $PSPACE$-complete.

Proof. By bit-blasting, $BV1$ can be polynomially reduced to Quantified Boolean Formulas (QBF), which is $PSPACE$-complete. Hardness follows from the fact that QBF $\subseteq BV1$ (following the same argument as in Prop. 7). □

Adding quantifiers to $QF_{UFBV1}$ increases complexity exponentially:

Proposition 10 (see [43]). $UFBV1$ is $NEXPTIME$-complete.

Proof. Effectively Propositional Logic (EPR), being $NEXPTIME$-complete, can be polynomially reduced to $UFBV1$ [43, Theorem 7]. For completing the other direction, apply the reduction in [43, Theorem 7] combined with the bit-blasting of the bit-vector operations. □

---

$^6$This kind of result is often called unary $NP$-completeness [24].
4.2 Logics With Binary Encoding

Our main contribution is to give complexity results for the more common logarithmic (actually without loss of generality) binary encoding. Even without uninterpreted functions nor quantification, i.e., for QF_BV_2, we obtain the same complexity as for UFBV_1.

**Proposition 11.** QF_BV_2 is \textit{NExpTime}-complete.

**Proof.** It is obvious that QF_BV_2 \in NExpTime, since a QF_BV_2 formula can be translated exponentially to QF_BV_1 \in NP (Prop. 7), by a simple unary re-encoding of all the numbers in the formula. The proof that QF_BV_2 is \textit{NExpTime}-hard is more complex and given in Sec. 4.2.1.

Adding uninterpreted functions to QF_BV_2, does not increase complexity.

**Proposition 12.** QF_UFBV_2 is \textit{NExpTime}-complete.

**Proof.** Again, new existential variables together with Ackermann constraints can be used in the same way as in the proof for Prop. 8.

However, adding quantifiers to QF_UFBV_2 increases complexity exponentially:

**Proposition 13.** UFBV_2 is 2-\textit{NExpTime}-complete.

**Proof.** Similarly to the proof of Prop. 11 a UFBV_2 formula can be exponentially translated to UFBV_1 \in \textit{NExpTime} (Prop. 10), simply by re-encoding all the numbers to unary. It is more difficult to prove that UFBV_2 is 2-\textit{NExpTime}-hard, which we show in Sec. 4.2.2.

Notice that deciding QF_BV_2 has the same complexity as UFBV_1. Thus, starting with QF_BV_1, re-encoding numbers to binary gives the same expressive power, in a precise complexity theoretical sense, as introducing both quantification and uninterpreted functions. This shows that it is important to differentiate between unary and binary encoding of numbers in bit-vector logics. Our results prove that binary encoding is at least as expressive as quantification, while only the latter has been considered in [44, 43].

4.2.1 QF_BV_2 is \textit{NExpTime}-hard

In order to prove that QF_BV_2 is \textit{NExpTime}-hard, we pick a \textit{NExpTime}-hard problem and, then, reduce it to QF_BV_2. Let us choose the satisfiability problem of DQBF (c.f. Section 3.3.1), which has been shown to be \textit{NExpTime}-complete [2].

**Theorem 14.** DQBF can be (polynomially) reduced to QF_BV_2.

**Proof.** The basic idea is to use bit-vector logic to encode function tables in an exponentially more succinct way, which then allows to characterize independence of an existential variable from a particular universal variable polynomially.

More precisely, we will use binary magic numbers, as constructed in Section 3.2.4, to create a certain set of fully-specified exponential-size bit-vectors by using a polynomial expression, due to binary encoding. We will then formally point out the well-known fact that those bit-vectors correspond exactly to the set of all assignments. We can then use a polynomial-size bit-vector formula for cofactoring Skolem-functions in order to express independency constraints.

First, we describe the reduction (c.f. an example in appendix D), then show that the reduction is polynomial, and, finally, that it is correct.
The reduction. Given a DQBF formula \( \phi := Q.m \) consisting of a quantifier prefix \( Q \) and a Boolean, without loss of generality, CNF formula \( m \) called the matrix of \( \phi \). Let \( u_0, \ldots, u_{k-1} \) denote all the universal variables that occur in \( \phi \). Translate \( \phi \) to a QF\_BV2 formula \( \Phi \) by eliminating the quantifier prefix and translating the matrix as follows:

**Step 1.** Replace all Boolean constants 0 and 1 with 0\(2^k\) resp. \(\neg 1\)\(2^k\) and all logical connectives with corresponding bitwise bit-vector operations (\(\lor, \land, \neg\) with \(\_\), \&\(\lor\), resp.).

Let \( \Phi' \) denote the formula generated so far. Extend it to the formula \( \Phi' = \neg 0\)\(2^k\).

**Step 2.** For each \( u_i \),

1. translate (all the occurrences of) \( u_i \) to a new bit-vector variable \( U_i^{2^k} \);
2. in order to assign the binary magic number to \( U_i \), add the following equation (i.e., conjunct it with the current formula):

   \[
   U_i = \text{binmagic}(2^i, 2^k)
   \]

**Step 3.** For each existential variable \( e \) depending on universals \( \text{Deps}(e) \subseteq \{u_0, \ldots, u_{k-1}\} \),

1. translate (all the occurrences of) \( e \) to a new bit-vector variable \( E^{2^k} \);
2. for each \( u_i \notin \text{Deps}(e) \), add the following equation:

   \[
   (E \land U_i) = ((E \Rightarrow u^{2^i}) \land U_i)
   \]  

As it is going to be detailed in the rest of the proof, the above equations enforce the corresponding bits of \( E^{2^k} \) to satisfy the dependency scheme of \( \phi \). More precisely, Eqn. (1) makes sure that the positive and negative cofactors of the Skolem-function representing \( e \) with respect to an independent variable \( u_i \) have the same value.

Polynomiality. Let us recall that all the numbers are encoded binary in the formula \( \Phi \), thus exponential bit-widths and constants \(2^k\) resp. \(2^i\) are encoded into linear many \((k\) resp. \(i\)) bits. We show now that each reduction step results only in polynomial growth of the formula size.

**Step 1** may introduce additional bit-vector constants to the formula. Their bit-width is \(2^k\), therefore, the resulting formula is bounded quadratically in the input size. **Step 2** adds \( k \) variables \( U_i^{2^k} \) for the original universal variables, as well as \( k \) equations as restrictions. The bit-widths of added variables and constants is \(2^k\). Thus the size of the added constraints is bounded quadratically in the input size. **Step 3** adds one bit-vector variable \( E^{2^k} \) and at most \( k \) constraints for each existential variable. Thus the size is bounded cubically in the input size.

Correctness. We show that the original \( \phi \) and the result \( \Phi \) of the translation are equisatisfiable. Consider one bit-vector variable \( U_i \) introduced in Step 2. In the following, we formalize the well-known fact that the combination of all the \( U_i \)s corresponds exactly to all possible assignments to the universal variables of \( \phi \). By construction, all bits of \( U_i \) are fixed to some constant value. Additionally, for every bit-vector index \( b_m \in [0, 2^k - 1] \), there exists a bit-vector index \( b_n \in [0, 2^k - 1] \) such that

\[
U_i[b_m] \neq U_i[b_n] \quad \text{and} \quad U_j[b_n] = U_j[b_n], \forall j \neq i.
\]
Actually, let us define $b_n$ in the following way (considering the 0th bit the least significant):

$$b_n := \begin{cases} 
    b_m - 2^i & \text{if } U_i[b_m] = 0 \\
    b_m + 2^i & \text{if } U_i[b_m] = 1
\end{cases}$$

By defining $b_n$ this way, Eqs. (2a) and (2b) both hold, which can be seen as follows. Let $R(c, l)$ be the bit-vector of length $l$ with each bit set to the Boolean constant $c$. Eqs. (2a) holds, since, due to construction, $U_i$ consists of several $(2^{k-1} - 1)$ concatenated bit-vector fragments $0 \ldots 01 \ldots 1 = R(0, 2^i)R(1, 2^i)$ (with both $2^i$ zeros and $2^i$ ones). Therefore, it is easy to see that $U_i[b_m] \neq U_i[b_m - 2^i]$ (resp. $U_i[b_m] \neq U_i[b_m + 2^i]$) holds if $U_i[b_m] = 0$ (resp. $U_i[b_m] = 1$). With a similar argument, we can show that Eqn. (2b) holds: $U_j[b_m] = U_j[b_m - 2^i]$ (resp. $U_j[b_m] = U_j[b_m + 2^i]$) if $U_j[b_m] = 0$ (resp. $U_j[b_m] = 1$), since $b_m - 2^i$ (resp. $b_m + 2^i$) is located either still in the same half or already in a concatenated copy of a $R(0, 2^j)R(1, 2^j)$ fragment, if $j \neq i$.

Now, consider all possible assignments to the universal variables of our original DQBF-formula $\phi$. For a given assignment $\alpha \in \{0, 1\}^k$, the existence of such a previously defined $b_n$ for every $U_i$ and $b_m$ allows us to iteratively find a $b_n$ such that $(U_0[b_n], \ldots, U_{k-1}[b_n]) = \alpha$. Thus, we have a bijective mapping from the universal assignments $\alpha$ of $\phi$ to a bit-vector index $b_n$ in $\Phi$.

In Step 3, we first replace each existential variable $e$ with a new bit-vector variable $E$ that can take $2^{(2^u)}$ different values. The value of each individual bit $E[b_n]$ corresponds to the value that $e$ takes under a given assignment $\alpha \in \{0, 1\}^k$ to the universal variables in $\phi$. Note that, without any further restriction, there is no connection between the different bits in $E$ and therefore the vector represents an arbitrary Skolem-function for an existential variable $e$. It may have different values for all universal assignments and thus would allow $e$ to depend on all universals.

If, however, $e$ does not depend on a universal variable $u_i$, we add the constraint of Eqn. (1). In DQBF, independence can be formalized in the following way: $e$ does not depend on $u_i$ if $e$ has to take the same value in the case of all pairs of universal assignments $\alpha, \beta \in \{0, 1\}^k$ where $\alpha[j] = \beta[j]$ for all $j \neq i$. Exactly this is enforced by our constraint. We have already shown that for $\alpha$ we have a corresponding bit-vector index $b_\alpha$, and we have defined how we can construct a bit-vector index $b_\beta$ for $\beta$. Our constraint for independence ensures that $E[b_\alpha] = E[b_\beta]$.

Step 1 ensures that all logical connectives and all Boolean constants are consistent for each bit-vector index, i.e. for each universal assignment, and that the matrix of $\phi$ evaluates to 1 for each universal assignment.

\[\square\]

### 4.2.2 UFBV2 is 2-NExpTime-hard

In order to prove that UFBV2 is 2-NExpTime-hard, we pick a 2-NExpTime-hard problem and, then, reduce it to UFBV2. We can find such a problem among the so-called domino tiling problems \[13\]. Let us first define what a domino system is, and then we specify a 2-NExpTime-hard problem on this kind of systems.

**Definition 15** (Domino System). A domino system is a tuple $\langle T, H, V, n \rangle$, where

- $T$ is a finite set of tile types, in our case, $T = \{0, k - 1\}$, where $k \geq 1$;
- $H, V \subseteq T \times T$ are the horizontal and vertical matching conditions, respectively;
- $n \geq 1$, encoded unary.
Let us note that the above definition differs (but not substantially) from the classical one in [13], in the sense that we use sub-sequential natural numbers for identifying tiles, as it is common in recent papers. Similarly to [33] and [34], the size factor $n$, encoded unary, is part of the input. However, while a start tile $\alpha$ and a terminal tile $\omega$ is usually used, in our case the starting tile is denoted by 0 and the terminal tile by $k - 1$, without loss of generality.

There are different domino tiling problems examined in the literature. In [13], a classical tiling problem is introduced, namely the square tiling problem, which can be defined as follows.

**Definition 16 (Square Tiling).** Given a domino system $\langle T, H, V, n \rangle$, an $f(n)$-square tiling is a mapping $\lambda : [0, f(n) - 1] \times [0, f(n) - 1] \mapsto T$ such that

- the first row starts with the start tile: $\lambda(0, 0) = 0$
- the last row ends with the terminal tile: $\lambda(f(n) - 1, f(n) - 1) = k - 1$
- all horizontal matching conditions hold: $(\lambda(i, j), \lambda(i, j + 1)) \in H \forall i < f(n), j < f(n) - 1$
- all vertical matching conditions hold: $(\lambda(i, j), \lambda(i + 1, j)) \in V \forall i < f(n) - 1, j < f(n)$

In [13], a general theorem on the complexity of domino tiling problems is proved:

**Theorem 17 (from [13]).** The $f(n)$-square tiling problem is NTime($f(n)$)-complete.

Since, for completing our proof on UFBV2, we need a 2-NExpTime-hard problem, let us emphasize the following easy corollary:

**Corollary 18.** The $2^{(2^n)}$-square tiling problem is 2-NExpTime-complete.

**Theorem 19.** The $2^{(2^n)}$-square tiling problem can be (polynomially) reduced to UFBV2.

**Proof.** Given a domino system $\langle T = [0, k - 1], H, V, n \rangle$, let us introduce the following notations which we intend to use in the resulting UFBV2 formula.

- Represent each tile in $T$ with the corresponding bit-vector of bit-width $Lk$.
- Represent the horizontal and vertical matching conditions with the uninterpreted functions (predicates) $h[1](t_1[Lk], t_2[Lk])$ and $v[1](t_1[Lk], t_2[Lk])$, respectively.
- Represent the tiling with an uninterpreted function $\lambda[Lk](i[2^n], j[2^n])$. $\lambda$ returns the tile in the cell at the row index $i$ and column index $j$. Notice that the bit-width of $i$ and $j$ is exponential in the size of the domino system, but due to binary encoding it can be represented polynomially.

The resulting UFBV2 formula is the following:

$$
\lambda(0, 0) = 0 \land \lambda(2^{(2^n)} - 1, 2^{(2^n)} - 1) = k - 1 \land \bigwedge_{(t_1, t_2) \in H} h(t_1, t_2) \land \bigwedge_{(t_1, t_2) \in V} v(t_1, t_2)
\land \forall i, j \left( \begin{array}{c}
(j \neq 2^{(2^n)} - 1 \Rightarrow h(\lambda(i, j), \lambda(i, j + 1)) ) \\
(i \neq 2^{(2^n)} - 1 \Rightarrow v(\lambda(i, j), \lambda(i + 1, j)) )
\end{array} \right)
$$

This formula contains four kinds of constants. Three can be encoded directly $(0[2^n], 0[Lk], (k - 1)[Lk])$. However, the constant $2^{(2^n)} - 1$ has to be treated in a special way in order to avoid double exponential size, namely in the following form: $\sim 0[2^n]$. The size of the resulting formula, due to binary encoding of the bit-width, is polynomial in the size of the domino system. \qed
5 Restricted Bit-Vector Logics

In this section, we will deal with different restrictions that affect the complexity of various bit-vector logics and propose some more complexity results. An overview of our results can be found in Table 2.

<table>
<thead>
<tr>
<th>encoding</th>
<th>unary</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>uninterpreted functions</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>uninterpreted functions</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 2: Completeness results for various restricted (resp. extended) bit-vector logics

5.1 Bit-Width Bounded Problems

We are now going to introduce a sufficient condition for bit-vector problems to remain in the “lower” complexity class, when re-encoding bit-width from unary to binary. This condition tries to capture the bounded nature of bit-widths in certain bit-vector problems.

In any bit-vector formula, there has to be at least one term with explicit specification of its bit-width. In the logics we are dealing with, only a variable, a constant, or an uninterpreted function can have explicit bit-width. Given a formula \( \phi \), let us denote the maximal explicit bit-width in \( \phi \) with \( \text{max}_{bw}(\phi) \). Furthermore, let \( \text{size}_{bw}(\phi) \) denote the number of terms with explicit bit-width in \( \phi \).

**Definition 20 (Bit-Width Bounded Formula Set).** An infinite set \( S \) of bit-vector formulas is (polynomially) bit-width bounded, if there exists a polynomial function \( p : \mathbb{N} \rightarrow \mathbb{N} \) such that \( \forall \phi \in S. \max_{bw}(\phi) \leq p(\text{size}_{bw}(\phi)) \).

**Proposition 21.** Given a bit-width bounded set \( S \) of formulas with binary encoded bit-width, any \( \phi \in S \) grows polynomially when re-encoding the bit-widths to unary.

**Proof.** Let \( \phi' \) denote the formula obtained through re-encoding bit-widths in \( \phi \) to unary. For the size of \( \phi' \), the following upper bound can be shown: \( |\phi'| \leq \text{size}_{bw}(\phi) \cdot \text{max}_{bw}(\phi) + c \). Notice that \( \text{size}_{bw}(\phi) \cdot \text{max}_{bw}(\phi) \) is an upper bound on the sum over the sizes of all the terms with explicit bit-width in \( \phi' \). The constant \( c \) represents the size of the rest of the formula. Since \( S \) is bit-width bounded, it holds that

\[
|\phi'| \leq \text{size}_{bw}(\phi) \cdot \text{max}_{bw}(\phi) + c \leq \text{size}_{bw}(\phi) \cdot p(\text{size}_{bw}(\phi)) + c \leq |\phi| \cdot p(|\phi|) + c
\]

where \( p \) is a polynomial function. Therefore, the size of \( \phi' \) is polynomial in the size of \( \phi \).

By applying this proposition to the logics of Sec. 3 we get:

**Corollary 22.** Let us assume a bit-width bounded set \( S \) of bit-vector formulas. If \( S \subseteq \text{QF}_\text{BV}_2 \) (and even if \( S \subseteq \text{QF}_\text{UFBV}_2 \)), then \( S \in \text{NP} \). If \( S \subseteq \text{BV}_2 \), then \( S \in \text{PSPACE} \). If \( S \subseteq \text{UFBV}_2 \), then \( S \in \text{NExpTime} \).
5.2 Benchmark Problems

In this section, we discuss concrete SMT-LIB benchmark problems in respect of whether they are bit-width bounded. Since, in SMT-LIB bit-widths are encoded logarithmically and quantification on bit-vectors is not (yet) addressed, we have picked benchmarks from QF_BV, which can be considered as QF_BV2 formulas.

First, consider the benchmark family QF_BV/brummayerbiere2/umulov2bw, which represent instances of an unsigned multiplication overflow detection equivalence checking problem, and is parameterized by the bit-width of unsigned multiplicands (b). We show that the set of these benchmarks, with b ∈ N, is bit-width bounded, and therefore is in NP. This problem checks that a certain (unsigned) overflow detection unit, defined in [39], gives the same result as the following condition: if the b/2 most significant bits of the multiplicands are zero, then no overflow occurs. It requires 2 · (b − 2) variables and a fixed number of constants to formalize the overflow detection unit, as detailed in [39]. The rest of the formula contains only a fixed number of variables and constants. The maximal bit-width in the formula is b. Therefore, the (maximal explicit) bit-width is linearly bounded in the number of variables and constants.

The benchmark family QF_BV/brummayerbiere3/mulhsb represents instances of computing the high-order half of product problem, parameterized by the bit-width of unsigned multiplicands (b). In this problem, the high-order b/2 bits of the product are computed following an algorithm detailed in [42, Page 132]. The maximal bit-width is b and the number of variables and constants to formalize this problem is fixed, i.e. independent of b. Therefore, the (maximal explicit) bit-width is not bounded in the number of variables and constants.

The family QF_BV/bruttomesso/lfsr/lfsr represents the behaviour of a linear feedback shift register [10]. Since, by construction, the bit-width (b) and the number (n) of registers do not correlate, and only n variables are used, this benchmark problem is not bit-width bounded.

A last family of benchmark problems was directly derived from the first part of this paper. In Sec. 3.2, we proposed ways to formalize various bit-vector operations by a set of base operations. To further backup our claims, we encoded most of the proposed translations into SMT-LIB benchmarks, proving that our operations produce correct results for all inputs of a given bit-width. Since all of our translations use at most logarithmic many steps in the size of the bit-width, the bit-width is at least exponential in the formula size. The benchmark family therefore is not bit-width bounded.

5.3 Restricted Quantifiers and Non-Recursive Macros

5.3.1 Restricting the Bit-Width of Universal Variables

As already discussed in Sec. 4 there is no completeness result for BV2. However, we will now show that a completeness result can be obtained when a certain restriction to the bit-width of universal variables is applied.

For a given formula φ ∈ BV2, let maxbw(∃) (φ) (resp. maxbw(∀) (φ)) denote the maximal bit-width of all the existentially (resp. universally) quantified variables. Let us consider all φ ∈ BV2 with maxbw(∀) (φ) ≤ log(max_bw(∃) (φ)). We will refer to this subset by BV2log.

This logic is of special practical interest, because it can be used to express quantification over array indices if arrays are expressed as bit-vectors.

Proposition 23. BV2log is NExpTime-complete.

Proof. It is easy to see that BV2log is NExpTime-hard since it is an extension of QF_BV2, which is already NExpTime-hard (Prop. [11]).
Let $\phi \in \text{BV}^2_{\log}$ be a formula of size $|\phi| = n$ containing $k \leq n$ universal variables. Also, let $\text{max}_{\text{bw}}(\phi)$ and $\text{size}_{\text{bw}}(\phi)$ be defined in the same way as it was done in Sec. 5.1. This implies $\text{max}_{\text{bw}}(\phi) = \text{max}_{\text{bw}(1)}(\phi) \leq 2^n$ and $\text{size}_{\text{bw}}(\phi) \leq n$. In order to prove that $\text{BV}^2_{\log} \in \text{NExpTime}$, we will now give a non-deterministic decision procedure that runs in exponential time of $n$.

First, all universal variables are eliminated by universal expansion. This produces a quantifier free formula $\phi'$ with $\text{max}_{\text{bw}}(\phi') = \text{max}_{\text{bw}(1)}(\phi') \leq 2^n$ and $\text{size}_{\text{bw}}(\phi') \leq \text{size}_{\text{bw}}(\phi) \cdot 2^{(n^2)}$

In a second step, a unary re-encoding is applied to $\phi'$ (similar to Prop. 11), resulting in $\phi'' \in \text{QF}_1$. The size of $\phi''$ is bounded by

$$|\phi''| \leq \text{max}_{\text{bw}}(\phi') \cdot \text{size}_{\text{bw}}(\phi') + c \leq 2^n \cdot 2^{(n^2)} + c$$

Therefore, $\phi''$ is still only exponential in the size of $\phi$. Together with $\text{QF}_1 \in \text{NP}$ (Prop. 7), this gives $\text{BV}^2_{\log} \in \text{NExpTime}$.

**Remark 24.** We can look at Prop. 23 from the perspective of Sec. 5.1. From this point of view, the set of exponentially large formulas $\phi' \in \text{QF}_2$ that is generated out of all possible $\phi \in \text{BV}^2_{\log}$ during the first step of our proof can be considered to be bit-width bounded. Together with Corr. 22, this gives an alternative way to see that $\text{BV}^2_{\log} \in \text{NExpTime}$.

In the same way, we can also define $\text{UFBV}^2_{\log}$ by adding uninterpreted functions to $\text{BV}^2_{\log}$ (or by adding the restriction on the bit-width of universal variables to $\text{UFBV}^2$). As one might expect, this does not increase complexity:

**Proposition 25.** $\text{UFBV}^2_{\log}$ is $\text{NExpTime}$-complete.

**Proof.** All arguments used in Prop. 23 still hold. In a last step, uninterpreted functions can then be replaced by existential bit-vector variables and the introduction of Ackermann constraints (as in Prop. 8).

### 5.3.2 Non-Recursive Macros

A very similar class is obtained when non-recursive macros are added to our logics. For example, SMT2 allows the usage of non-recursive macros via the keywords `define-fun` and `let`. In the general case, allowing macros might increase the complexity of a given class. For instance, Boolean Formulas extended by non-recursive macros equal to the class of Boolean Programs or Nested Boolean Functions (NBF), which is known to be $\text{PSPACE}$-complete [15] [12].

Extending $\text{QF}_2$ by macros, however, does not give additional expressiveness. Let the subscript $\text{macro}$ denote the fact that additionally non-recursive macros can be used in our logic.

**Proposition 26.** $\text{QF}_2^{\text{macro}}$ is $\text{NExpTime}$-complete.

**Proof.** Again, $\text{NExpTime}$-hardness is trivial since $\text{QF}_2 \subset \text{QF}_2^{\text{macro}}$. Inclusion can be shown in a very similar way as it is done in 23.

Let $\phi \in \text{QF}_2^{\text{macro}}$ be a formula of size $n$ containing arbitrary nested macros. We now inductively expand all macros in $\phi$. Each macro is replaced by an instance of the corresponding term in its definition. Since the definition of all macros is also part of the input, the size of
each term is also bounded by $n$. As we are dealing with nested macros, the instantiated term can again contain other macros. We continue expanding those macros until no more macros are part of the formula. This will finally happen because all macros are non-recursive and therefore the number of expansion steps is also bounded by $n$.

The result is a formula $\phi' \in \text{QF}_\text{BV2}$ with
\[
\begin{align*}
\max_{bw} (\phi') &= \max_{bw} (\phi) \leq 2^n \\
\size_{bw} (\phi') &\leq n^n = 2^{n \log n}
\end{align*}
\]

We now apply a unary re-encoding to $\phi'$, yielding $\phi'' \in \text{QF}_\text{BV1}$. The size of $\phi''$ is bounded by
\[
|\phi''| \leq \max_{bw} (\phi') \cdot \size_{bw} (\phi') + c \leq 2^n \cdot 2^{n \log n} + c
\]

which is only exponential in the size of $\phi$. This gives $\text{QF}_\text{BV2}_{macro} \in \text{NExpTime}$.

\textbf{Proposition 27.} $\text{QF}_\text{UFBV2}_{macro}$ is $\text{NExpTime}$-complete.

\textit{Proof.} All arguments used in Prop. 26 still hold. In a following step, uninterpreted functions are replaced by new existential bit-vector variables and Ackermann constraints.

\textbf{5.3.3 Logics with Unary Encoded Bit-Width}

Note that all proofs we presented in this section only hold for the binary case. It is easy to see that $\text{BV1}_{log}$ (resp. $\text{UFBV1}_{log}$) is $\text{PSPACE}$-complete (resp. $\text{NExpTime}$-complete) since it is a superset of $\text{QBF}$ (resp. $\text{DQBF}$), and a subset of $\text{BV1}$ (resp. $\text{UFBV1}$).

In the same way, adding macros increases expressiveness and makes $\text{QF}_\text{BV1}_{macro}$ (resp. $\text{QF}_\text{UFBV1}_{macro}$) $\text{PSPACE}$-complete (resp. $\text{NExpTime}$-complete). This can be seen, for example, by translating macros to quantifiers and vice versa in the same way as it is done in [12].

\textbf{5.4 Logics Without Slicing}

In Sec. 3 we have defined a set of base operations of our bit-vector logics and described a way to express all other common operations by this set. We then looked at the complexity of these logics and gave several completeness results in Sec. 4.

Now we will argue that this set of base operations is minimal in a certain way for the binary case. We will show that as soon as we remove slicing from the set of given operations, i.e. only consider bit-vector logics with bitwise operations and equality, binary encoding does not give additional power over unary encoding for the quantifier-free case. For unary encoded formulas on the other hand, slicing does of course not offer any additional expressiveness since it can simply be bit-blasted polynomially.

We will now look at the binary case and denote those logics by $\text{QF}_\text{BV2}_{bw}$, and $\text{QF}_\text{UFBV2}_{bw}$, with the subscript $bw$ representing the fact that the logics only contain bitwise operations and equality.

\textbf{Proposition 28.} $\text{QF}_\text{BV2}_{bw}$ is $\text{NP}$-complete

\textit{Proof.} Since Boolean Formulas still are a subset of $\text{QF}_\text{BV2}_{bw}$, $\text{NP}$-hardness follows directly.

To show that $\text{QF}_\text{BV2}_{bw} \in \text{NP}$, we give a reduction from $\text{QF}_\text{BV2}_{bw}$ to a set of bit-width bounded formulas. The claim then follows directly from Corr. 22.
Given a formula \( \phi \in QF_{BV2} \). Without loss of generality, assume that \( \phi \) does not contain any bit-vector constants. We now construct a formula \( \phi' \) by reducing the bit-widths of all bit-vector terms in \( \phi \). Each term \( t^{[n]} \) in \( \phi \) with an explicit bit-width \( n \) is replaced by a term \( t^{[n']} \), with \( n' := \min\{n, \text{size}_{bw}(\phi)\} \). Apart from this, \( \phi' \) is exactly the same as \( \phi \). As a consequence, \( \text{max}_{bw}(\phi') \leq \text{size}_{bw}(\phi) = \text{size}_{bw}(\phi') \). The set of formulas constructed in this way, therefore, is bit-width bounded according to Def. [20].

To complete our proof we now have to show that the proposed reduction is sound, i.e. out of every satisfying assignment to the bit-vector variables \( v_1^{[n_1]}, \ldots, v_k^{[n_k]} \) for \( \phi \) we can construct a satisfying assignment to \( v_1^{[n_1']}, \ldots, v_k^{[n_k']} \) for \( \phi' \) and vice versa.

Let \( \alpha \) be an assignment and \( F \) a formula. We use the notation \( \alpha(F) \) to donate the evaluation of \( F \) under \( \alpha \), with \( \alpha(F) \in \{0, 1\} \) and \( \alpha(F) = 1 \) representing the fact that \( \alpha \) is satisfying for \( F \).

It is easy to see that whenever we have a satisfying assignment \( \alpha' \) for \( \phi' \), we can construct a satisfying assignment \( \alpha \) for \( \phi \). This can be done, for example, by simply setting all additional bit positions of all bit-vector variables to the same value as the most significant bit of the corresponding original vector, i.e. by performing a signed extension. Since all equalities and inequalities will still evaluate to the same value under the extended assignment, \( \alpha(F) = \alpha'(F') \) for all atomic formulas \( F \) (resp. \( F' \)) of \( \phi \) (resp. \( \phi' \)). As a direct consequence, \( \alpha(\phi) = \alpha'(\phi') = 1 \).

The other direction needs slightly more reasoning. Given \( \alpha \), with \( \alpha(\phi) = 1 \), we need to construct \( \alpha' \), with \( \alpha'(\phi') = 1 \). Again, we want to ensure that \( \alpha'(F') = \alpha(F) \) for all atomic formulas \( F \) (resp. \( F' \)) in \( \phi \) (resp. \( \phi' \)).

For each \( F \) with \( \alpha(F) = 0 \), we therefore look for a bit-index which is a witness for its evaluation and select it. If \( \alpha(F) = 1 \), we select an arbitrary bit-index. We then mark the selected bit-index in all bit-vector variables contained in \( F \), as well as in all other bit-vector variables with the same bit-width. Having done this for all atomic formulas, we end up with a set \( M(v_i^{[n_i]}) \) of marked bit indices for each bit-vector, with

\[
|M(v_i^{[n_i]})| \leq \min\{n_i, \text{size}_{bw}(\phi)\} \quad \forall i \in \{1, \ldots, k\}
\]

\[
M(v_i^{[n_i]}) = M(v_j^{[n_j]}) \quad \forall i, j \in \{1, \ldots, k\} \text{ with } n_i = n_j
\]

and we know the marked indices contain a witness for the evaluation of each atomic formula.

We now mark arbitrary further indices for all bit-vector variables, again selecting the same indices in bit-vector variables with the same bit-width, until \( |M(v_i^{[n_i]})| = \min\{n_i, \text{size}_{bw}(\phi)\} \forall i \in \{1, \ldots, k\} \).

Finally, we can directly construct \( \alpha' \) using the marked indices and get \( \alpha'(\phi') = \alpha(\phi) = 1 \) because of the fact that we included a witness for every atomic formula in our marking process.

Note that we only had to choose a specific witness for the case that \( \alpha(F) = 0 \). For \( \alpha(F) = 1 \), we were able to choose an arbitrary bit-index because every satisfied atomic formula will trivially still be satisfied when only a subset of all bit-indices is considered.

\[\square\]

**Remark 29.** A similar proof can be found in [26, 27]. While the focus of [26, 27] is on improving the practical efficiency of SMT-solvers by reducing the bit-width of a given formula before bit-blasting, the author does not investigate its influence on the complexity of a given problem class. In fact, the author claims that bit-vector theories with concatenation are NP-complete. As we have already shown, this only holds if a unary encoding for the bit-widths is used. At the same time, unary encoding leads to the fact that the given class of formulas remains NP-complete, independent of whether a reduction of the bit-width is possible. However, the arguments on bit-width reduction given in [26, 27] still hold for binary encoded bit-vector formulas when no
concatenation is used. Our proof directly relates to the complexity of the problem class while sometimes using similar arguments as [26, 27] to remain self-contained.

Not surprisingly, adding uninterpreted functions to this class, does not affect complexity:

**Proposition 30.** $\text{QF}_{\text{UFBV}}^{2}_{\text{bw}}$ is NP-complete

**Proof.** NP-hardness of $\text{QF}_{\text{UFBV}}^{2}_{\text{bw}}$ directly follows from the NP-hardness of $\text{QF}_{\text{BV}}^{2}_{\text{bw}}$. For inclusion, we can again apply the same steps as in our inclusion proofs for Prop. 8 and Prop. 12. Replacing all uninterpreted functions by bit-vector variables and adding quadratic many Ackermann constraints, yields a $\text{QF}_{\text{BV}}^{2}_{\text{bw}}$ formula, proving that $\text{QF}_{\text{UFBV}}^{2}_{\text{bw}} \in \text{NP}$.

### 5.5 The Power of Non-Base Operations

As we have just shown, the decision problem of $\text{QF}_{\text{BV}}^{2}_{\text{bw}}$ and $\text{QF}_{\text{UFBV}}^{2}_{\text{bw}}$ is NP-complete. As a direct consequence of Theorem 14, adding slicing to the set of operations makes the decision problem NExpTime-complete, since we have shown how to express all other operations by this set of base operations.

On the other hand, we could also use the alternative approach in Remark 5 for constructing binary magic numbers, instead of the standard concatenation-based (and, hence, slicing-based) one. This shows that also the non-base operation shift by constant is sufficient to prove NExpTime-completeness together with bitwise operations and equality, without applying slicing at all.

In this section, we will show that the same holds for the non-base operation multiplication together with the set of bitwise operations and equality.

This means all the aforementioned operations, slicing, concatenation, shifts (by constant), and multiplication, can be considered equally powerful from a complexity theoretical point of view.

To prove this, we will propose a way to express shifts by constant using multiplication. While we have also shown NExpTime-completeness using only bitwise operations and equality, in combination with concatenation or shifts by constant, we also want to give a constructive approach. We will therefore show how slicing can be expressed by concatenation, and how a formula containing concatenation can be reduced to a formula using shifts by constant instead.

In Figure 5.5 a graphical representation of our claims can be seen. The vertex shift$_c$ represents the (left and logical right) shift operations by a constant, all the other ones are self-explanatory.

The solid edges refer to the translations that we have already proposed in Section 3.2. Although we have not explicitly shown so far that shift$_c$ can be expressed by shift, this is trivial, since shift is more general than shift$_c$. Also, we have not explicitly shown how shift$_c$, shift, and mult can be expressed by slice. But again, this is trivial since we can just succinctly apply the proposed translations until the formula only contains base operations.

In the rest of this section, we propose the translations corresponding to the dashed edges and the single dotted edge. The dashed edge between concat and shift$_c$ takes a slightly special role, which we will discuss in Sec. 5.5.2. The dotted edge between mult and add occurs in logics which allow both quantification and uninterpreted functions (e.g. in UFBV2). We have not found any evidence yet whether it can occur in more restrictive logics. Sec. 5.5.4 will cover this in more detail.

---

7 Consequently, even the several variants of division have the same power.
5.5.1 Slice → Concatenation.

Slicing can naturally be expressed by concatenation. Still, we have to deal with some boundary cases due to the limitation that any operand of concatenation must not be of bit-width 0 by definition. In the table below, we first distinguish between two cases: whether the 2nd index of the slicing does or does not take the least possible value (0). Within both cases, we distinguish between two more cases: whether the 1st index does or does not take the greatest possible value (n - 1).

<table>
<thead>
<tr>
<th>Term</th>
<th>Condition</th>
<th>Bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>slice:</td>
<td>( t^{[n]}[i : j] )</td>
<td>( n &gt; i \geq j \geq 0 )</td>
</tr>
<tr>
<td>( \uparrow x[i-j+1] )</td>
<td>( t = x )</td>
<td>if ( i = n - 1 )</td>
</tr>
<tr>
<td>( t = y_1^{[n-i-1]} \odot x )</td>
<td>if ( i \neq n - 1 )</td>
<td>otherwise</td>
</tr>
<tr>
<td>( t = x \odot y_2[j] )</td>
<td>( t = y_1^{[n-i-1]} \odot x \odot y_2[j] )</td>
<td>otherwise</td>
</tr>
</tbody>
</table>

5.5.2 Concatenation → Shift by Constant.

As already mentioned, this translation takes a special role. Given two terms \( t_1^{[m_1]}, t_2^{[m_2]} \), concatenation produces a new term of bit-width \( m_1 + m_2 \). This change in bit-width (which also occurs for slicing) cannot be captured by only applying rewriting rules using shifts (or any other operation that only produces outputs with the same bit-width as the input). However, due to the NExpTime-completeness of both classes, we can find a reduction from bit-vector formulas containing only concatenation, bitwise operations and equality to bit-vector formulas containing only shifts by constant, bitwise operations and equality.

Given a formula \( F \) with bit-vector variables \( v_1^{[n_1]}, \ldots, v_l^{[n_l]} \) and a set of concatenations \( t_1^{[m_{j,1}]} \odot t_2^{[m_{j,2}]} \). Construct a new formula \( F' \) by copying \( F \) and replacing all bit-vector variables by extended bit-vectors \( v_1^{[n_{M}], \ldots, v_{l}^{[n_{M}]}}, \) with \( n_{M} := \max\{\max_{i} \{n_{i}\}, \max_{j} \{m_{j,1} + m_{j,2}\}\} \). This ensures that all bit-vectors have the same bit-width and every bit-vector and every result of a concatenation can take (at least) all values it could take in the original formula.

In a second step, add the constraints \( \bigwedge_{k} (v_k^{[n_M]} = (v_k^{[n_M]} \ll (n_M - n_k)) \gg u (n_M - n_k)) \) to \( F' \). This restricts the bit-vectors to exactly the same set of values that they could take in...
the original formula $F$.

Finally, for each concatenation $t_1^{m_{j,1}} t_2^{m_{j,2}}$ in $F$, replace the corresponding concatenation in $F'$ by $(t_1^{m_{j,1}} \ll m_{j,2})$ | $t_2^{m_{j,2}}$. This step removes all concatenations using shifts by constant (and bitwise or) to express its effect.

5.5.3 Shift by Constant → Multiplication.

Left shift by a constant $c$ can be easily expressed by multiplication: $t^{c} \ll c^{n} \leftarrow t \cdot 2^c$. Of course, this only holds if $c < n$; otherwise the result of the shift is 0. Till the end of this section, let us assume that $c < n$.

In a similar fashion, (logical) right shift by a constant $c$ can be expressed by division: $t^{n} \gg c^{n} \leftarrow t / 2^c$. As we showed in Section 3.2.7, division can be expressed by multiplying the quotient $d$ with the divisor $2^c$ and then adding the remainder $r$, as follows:

$$t = d \cdot 2^c + r$$  \hspace{1cm} (3)

We also need to make some restrictions. First of all, $r$ must not exceed the divisor. Furthermore, overflow is also needed to be checked. Fortunately, since now we are about dividing with $2^c$, overflow can be avoided by forcing $d$ not to exceed $2^{n-c}$.

$$r < 2^c \text{ and } d < 2^{n-c}$$  \hspace{1cm} (4)

Notice that all the $c$ least significant bits of $d \cdot 2^c$ and all the $n - c$ most significant bits of $r$ are 0, therefore the addition in Eqn. (3) can be replaced with bitwise xor, or rather with bitwise or.

$$t \leftarrow (d \cdot 2^c) \mathbin{\&} r$$

The inequalities in Eqn. (4) use constants of form $2^i$ on the right-hand side. This kind of inequality can simply be evaluated by checking whether all the $n - i$ most significant bits of the bit-vector on the left-hand side are 0. We can do this by masking the bit-vector with $\overline{1 \ldots 1 0 \ldots 0}$, and checking whether the result is 0. The required mask is represented by the constant $(2^{n-i} - 1) \cdot 2^i$. Thus, Eqn. (4) can be reformulated as follows:

$$r \& ((2^{n-c} - 1) \cdot 2^c) = 0$$  \hspace{1cm} and

$$d \& ((2^c - 1) \cdot 2^{n-c}) = 0$$  \hspace{1cm} (5)

In order to construct exponential numbers of form $2^i$ resp. $2^i - 1$ in a polynomial way, we are going to introduce two helper operations $\text{pow2}(i)$ resp. $\text{pow2min1}(i)$. But let us first introduce a set of variables $p_j$. Each variable $p_j$ represents the number $2^{(2^j)}$:

$$p_0 = 2$$

$$p_j = p_{j-1} \cdot p_{j-1} \quad \text{where } j > 0$$

Each variable $pm_j$ represents the number $2^{(2^j)} - 1$:

$$pm_0 = 1$$

$$pm_j = (pm_{j-1} \cdot p_{j-1}) | pm_{j-1} \quad \text{where } j > 0$$
Considering the individual bits of a bit-vector constant $c^{[n]}$, the helper operation $\text{pow}2(c)$ selects some of the $p_j$s and multiplies them with each other. The other helper operation $\text{pow}2\text{min1}(c)$ selects and concatenates some of the $pm_j$s, in a similar fashion as it was proposed in Sec. 5.5.2 (where left shift is now done by multiplying with a $p_j$).

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{pow}2(c^{[n]})$</td>
<td>$c &lt; n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$\uparrow x_0^{[n]}$</td>
<td>$p_{\text{max}}$ if $c[0]$</td>
<td></td>
</tr>
<tr>
<td>where</td>
<td>$1$ otherwise</td>
<td></td>
</tr>
<tr>
<td>$x_t^{[n]} = {x_{i-1} \cdot p_i$ if $c[i]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for all $0 &lt; i \leq L_{\text{max}}$</td>
<td>$x_{i-1}$ otherwise</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{pow}2\text{min1}(c^{[n]})$</td>
<td>$c &lt; n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$\uparrow x_0^{[n]}$</td>
<td>$p_{\text{max}}$ if $c[0]$</td>
<td></td>
</tr>
<tr>
<td>where</td>
<td>$0$ otherwise</td>
<td></td>
</tr>
<tr>
<td>$x_t^{[n]} = {(x_{i-1} \cdot p_i) \mid pm_i$ if $c[i]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for all $0 &lt; i \leq L_{\text{max}}$</td>
<td>$x_{i-1}$ otherwise</td>
<td></td>
</tr>
</tbody>
</table>

We can use these helper operations to construct the four exponential numbers in Eqn. (5). In the following table, we can finally see the translation of shifts to bitwise operations, equality, and multiplication.

<table>
<thead>
<tr>
<th>term</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t^{[n]} \ll c^{[n]}$</td>
<td>$n$</td>
</tr>
<tr>
<td>$\uparrow {0^{[n]}$ if $c \geq n$</td>
<td></td>
</tr>
<tr>
<td>$t \cdot \text{pow}2(c)$ otherwise</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>term</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t^{[n]} \gg_a c^{[n]}$</td>
<td>$n$</td>
</tr>
<tr>
<td>$\uparrow {0^{[n]}$ if $c \geq n$</td>
<td></td>
</tr>
<tr>
<td>$d$ otherwise</td>
<td></td>
</tr>
<tr>
<td>where</td>
<td></td>
</tr>
<tr>
<td>$t = (d^{[n]} \cdot \text{pow}2(c)) \mid r^{[n]}$</td>
<td></td>
</tr>
<tr>
<td>$r &amp; (\text{pow}2\text{min1}(n - c) \cdot \text{pow}2(c)) = 0^{[n]}$</td>
<td></td>
</tr>
<tr>
<td>$d &amp; (\text{pow}2\text{min1}(c) \cdot \text{pow}2(n - c)) = 0^{[n]}$</td>
<td></td>
</tr>
</tbody>
</table>

Notice that the translation of left shift requires to generate the $p_j$s for all $0 \leq j < L_{\text{c}}$. Furthermore, the translation of logical right shift requires to generate both the $p_j$s and $pm_j$s for all $0 \leq j < L_{\text{max}}(c, n - c)$. Therefore, only logarithmic many such variables (and corresponding equations) have to be added to our original formula.
5.5.4 Multiplication → Addition.

As a last operation, we now want to look at addition. In Sec. 4.2.2, we proved that UFBV is 2-NEXPTime-hard. The only bit-vector operations we used to do so, were addition, bitwise negation and equality. This shows that for UFBV, even addition is equally powerful as slicing, concatenation, shifts (by constant) and multiplication. A simple way to express multiplication by addition is using quantification and uninterpreted functions as it is done in Peano arithmetic:

1. \( \forall x^n, (f^n(x^n, 0^n) = 0^n) \)
2. \( \forall x^n, y^n, (f^n(x^n, y^n + 1^n) = f^n(x^n, y^n) + x^n) \)

Adding these two axioms as constraints to the formula, allows us to replace any multiplication \( t_1^n \cdot t_2^n \) by an instance of the uninterpreted function \( f^n(t_1^n, t_2^n) \).

Now, considering QF_BV2 again, we cannot use universally quantified axioms to express multiplication by addition. While adding addition to QF_BV2₂ is still increases expressiveness, the resulting logic is less powerful than QF_BV₂. This can be seen by the following proposition:

**Proposition 31.** QF_BV₂₂⁺ is PSPACE-complete.

**Proof.** In Thm. 32, we will give a (polynomial) reduction from QBF to QF_BV₂₂⁺. This implies PSPACE-hardness of QF_BV₂₂⁺. In Prop. 33, we will then show that QF_BV₂₂⁺ ∈ PSPACE by giving a translation from QF_BV₂₂⁺ to (polynomial sized) Sequential Circuits.

**Theorem 32.** QBF can be (polynomially) reduced to QF_BV₂₂⁺.

**Proof.** To show PSPACE-hardness of QF_BV₂₂⁺, we will propose a polynomial reduction from QBF similar to the one from DQBF to QF_BV₂ given in Thm. 1.

For our reduction, we will again use the binary magic numbers. Note that in Rem. 6, we gave a construction for the binary magic numbers which uses only shifts by 1, bitwise operations and equality.

It is easy to see that shift by 1 can be expressed by addition:

<table>
<thead>
<tr>
<th>shift by 1:</th>
<th>term</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t^n \ll 1^n )</td>
<td>( t^n \ll 1^n )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Let \( \phi := Q.m \) denote a QBF formula with quantifier prefix \( Q \) and matrix \( m \). Since \( \phi \) is a QBF formula (in contrast to DQBF), we know that \( Q \) defines a total order on the universal variables. We now assume the universal variables \( u_0, \ldots, u_{k-1} \) of \( \phi \) are ordered according to their appearance in \( Q \), with \( u_0 \) (resp. \( u_{k-1} \)) being the innermost (resp. outermost) variable.

Translate \( \phi \) to a QF_BV₂₂⁺ formula \( \Phi \) by eliminating the quantifier prefix and translating the matrix as follows:

**Step 1.** Replace Boolean constants 0 and 1 with \( 0[2^k] \) resp. \( 1[2^k] \) and logical connectives with corresponding bitwise bit-vector operations \( (\land, \lor, \neg \land \lor, \land, \lor \neg) \).

Let \( \Phi' \) denote the formula generated so far. Extend it to the formula \( (\Phi' = \neg 0[2^k]) \).

---

---
Step 2. For each $u_i$,

1. translate (all the occurrences of) $u_i$ to a new bit-vector variable $U_i^{[2^k]}$;
2. in order to assign the binary magic number to $U_i$, add the following equation (i.e., conjunct it with the current formula):

$$U_i = binmagic(2^i, 2^k)$$

Step 3. For each existential variable $e$ depending on universals $Deps(e) = \{u_i, \ldots, u_k\}$, with $u_i$ being the innermost universal variable that $e$ depends on,

1. translate (all the occurrences of) $e$ to a new bit-vector variable $E^{[2^k]}$;
2. if $Deps(e) = \emptyset$ add the following equation:

$$E \& \sim 1 = (E \ll 1)$$

otherwise, if $i \neq 0$ add the two equations:

$$U_i' = \sim ((U_i \ll 1) \oplus U_i)$$
$$E \& U_i' = ((E \ll 1) \& U_i')$$

First of all, note that again shift by 1 can be expressed by addition. Eqn. (7) creates a bit-vector $U_i'$ for which each bit equals to 1 if and only if the corresponding universal variable changes its value from one assignment to the next. Of course, Eqn. (7) does not have to be added multiple times, if several existential variables depend on the same universals. Eqn. (8) (resp. Eqn. (6)) ensures that the corresponding bits of $E^{[2^k]}$ satisfy the dependency scheme of $\phi$ by only allowing the value of $e$ to change if an outer universal variable takes a different value. If $i = 0$, i.e. if $e$ depends on all universal variables, Eqn. (7) evaluates to $U_0' = 0$, and as a consequence Eqn. (8) simplifies to true. Because of this no constraints need to be added for $i = 0$.

Proposition 33. QF_BV_{bw+} can be (polynomially) reduced to Sequential Circuits.

Proof. In [41, 40], the authors give a translation from QFPAbit to Sequential Circuits. We can adopt their approach in order to construct a translation for QF_BV_{bw+}. The only difference between QFPAbit and QF_BV_{bw+} is the fact that bit-vectors of arbitrary, non-fixed, size are allowed in QFPAbit while all bit-vectors contained in QF_BV_{bw+} have a fixed bit-width. Given $\phi \in QF_BV_{bw+}$. We first translate $\phi$ to $\phi' \in QF_BV_{bw+}$ so that $\phi'$ only contains atoms $t_1[n] = t_2[n], x[n] = c[n]$, and $z[n] = x[n] + y[n]$, where $x[n], y[n], z[n]$ are bit-vector variables, $c[n]$ is a bit-vector constant, and $t_1[n], t_2[n]$ are bit-vector terms only containing bit-vector variables and bitwise operations (c.f. [41, 40]).

In a second step, we now encode each atom in $\phi'$ separately to a Sequential Circuit. In general, this can be done in the same way as it was proposed in [41, 40]. However, a slight modification is needed to represent the fact that we are dealing with fixed-size bit-vectors. Let $n$ be the bit-width of all bit-vectors in the current formula atom. Note that all bit-vectors inside a single atom have the same bit-width. We now additionally implement a counter for each Sequential Circuit. The counter initially is set to 0 and is incremented by 1 in each clock cycle up to a value of $n$. When the counter reaches a value of $n$, it does not change anymore.
and the output of the Sequential Circuit is set to the same value as the output in the previous cycle. A counter like this can be realized with \( \log(n) \) flip-flops, i.e. polynomially in the size of \( \phi' \), even with binary encoding for the bit-width \( n \). We also need to make sure the input streams for all variables start with the least significant bit.

Finally, once the individual circuits have been constructed, we combine the outputs by logical gates representing the Boolean structure of \( \phi' \). Again, this step is straightforward and equals to the original approach in [41, 40]. Due to adding counters to our Sequential Circuits in step 2, we ensured that for every input stream \( x_i \), only the first \( n_i \) bits of \( x_i \) influence the result of the whole circuit. This directly gives an assignment to all variables \( x_i^{[n_i]} \), so that \( \phi' \) is satisfied.

6 Conclusion

We discussed the complexity of deciding various quantified and quantifier-free fixed-size bit-vector logics. In contrast to existing literature, where usually it is not distinguished between unary or binary encoding of the bit-width, we argued that it is important to make this distinction. Our new results apply to the actually much more natural binary encoding as it is also used in standard formats, e.g. in the SMT-LIB format.

We proved that deciding \( \text{QF}_{\text{BV}}2 \) is \( \text{NExpTime} \)-complete, which is the same complexity as for deciding \( \text{UFBV1} \). This shows that binary encoding for bit-widths has at least as much expressive power as quantification does. We also proved that \( \text{UFBV2} \) is \( 2-\text{NExpTime} \)-complete. So far we know complexity results for \( \text{QF}_{\text{BV}}1 \), \( \text{QF}_{\text{UFBV}}1 \), \( \text{BV1} \), \( \text{UFBV1} \), \( \text{QF}_{\text{BV}}2 \), \( \text{QF}_{\text{UFBV}}2 \), and \( \text{UFBV2} \). The complexity of deciding \( \text{BV2} \) remains unclear.

While it is easy to show \( \text{ExpSpace} \)-inclusion for \( \text{BV2} \) by bit-blasting to an exponential-size QBF, and \( \text{NExpTime} \)-hardness follows directly from \( \text{QF}_{\text{BV2}} \subset \text{BV2} \), it is not clear whether \( \text{QF}_{\text{BV2}} \) is complete for any of these classes. However, we showed that as soon as we restrict the bit-width of universal variables to be logarithmic in the bit-width of the existential variables, we are able to prove \( \text{NExpTime} \)-completeness for the resulting logic \( \text{QF}_{\text{BV2}_{\log}} \).

We also introduced the concept of bit-width bounded problems and proved that for certain sets of formulas the increase of complexity that comes with a binary encoding can be avoided. We then gave examples of benchmark problems that do (resp. do not) fulfill this condition, i.e. are bit-width bounded (resp. not bit-width bounded). As future work, it might be interesting to consider our results in the context of parametrized complexity [18].

Another contribution of our paper was to make clear which operations are necessary for a bit-vector logic to express all other common operations. Therefore, we showed how most common bit-vector operations can be logarithmically expressed by a certain set of base operations. In particular, we gave this translation for all operations used in our hardness proofs. We then proved that this set of base operations is minimal in the sense that removing slicing from \( \text{QF}_{\text{BV2}} \) makes the resulting logic \( \text{QF}_{\text{BV2}_{bw}} \) \( \text{NP} \)-complete. Allowing addition results in an intermediate logic that is \( \text{PSPACE} \)-complete.

Our theoretical results give an argument for using more powerful solving techniques when dealing with bit-vector logics. Currently the most common approach used in state-of-the-art SMT solvers for bit-vectors is based on simple rewriting, bit-blasting, and SAT solving. We have shown this can possibly produce exponentially larger formulas when a logarithmic encoding is used as an input. Possible candidates are techniques used in EPR and/or (D)QBF solvers (see e.g. [22, 29]).
References


Complexity of Bit-Vector Logics with Binary Encoded Bit-Width  
Kovácszni, Fröhlich, and Biere

A Example: Half-Shuffle and Expand applied to a bit-vector as described in Sec. 3.2.4

\[
\text{halfshuffle} \left( \begin{array}{c}
\ell^{[4]} \\
1101, 16
\end{array} \right)
\]

\[ \downarrow x_2^{[16]} \]

First, zero extension is applied to the original vector:

\[ x_0^{[16]} = \text{ext}_4 (1101, 12) = 0000 0000 0000 1101 \]

Now, in two iterations, the bits of \( \ell^{[4]} \) are separated and moved to the distinct partitions of the extended vector:

\[
x_1^{[16]} = (x_0^{[16]} | (x_0^{[16]} \ll 6)) \& \text{binmagic} (2, 16) =
\]
\[
= (0000 0000 0000 1101 | 0000 0011 0100 0000) \& 0011 0011 0011 0011 =
\]
\[
= 0000 0011 0000 0001
\]

\[
x_2^{[16]} = (x_1^{[16]} | (x_1^{[16]} \ll 3)) \& \text{binmagic} (1, 16) =
\]
\[
= (0000 0011 0000 0001 | 0001 1000 0000 1000) \& 0101 0101 0101 0101 =
\]
\[
= 0001 0001 0000 0001
\]

The result now can be used for example in expand:

\[
\text{expand} \left( \begin{array}{c}
\ell^{[4]} \\
1101, 16
\end{array} \right)
\]

\[ \downarrow x_2^{[16]} \]

First, the result of half-shuffle is used:

\[ x_0'{}^{[16]} = \text{halfshuffle} (1101, 16) = 0001 0001 0000 0001 \]

Then, the single bits in each partition are copied several times:

\[
x_1'{}^{[16]} = x_0'{}^{[16]} | (x_0'{}^{[16]} \ll 1) =
\]
\[
= 0001 0001 0000 0001 | (0010 0010 0000 0010) = 0011 0011 0000 0011
\]

\[
x_2'{}^{[16]} = x_1'{}^{[16]} | (x_1'{}^{[16]} \ll 2) =
\]
\[
= 0011 0011 0000 0011 | (1100 1100 0000 1100) = 1111 1111 0000 1111
\]

with \( x_2' \) being the final result of expanding \( \ell^{[4]} = 1101 \) to bit-width 16.
B Example: Multiplication of two bit-vectors using the approach described in Sec. 3.2.7

First, both vectors are transformed by selfconcat resp. expand to quadratic size in order to generate all single digit multiplications in one step by using bitwise-and:

\[ x^{[16]} = \text{selfconcat}(0011, 16) \land \text{expand}(0101, 16) = 0011 0011 0011 0011 \land 0000 1111 0000 1111 = 0000 0011 0000 0011 \g{4} \g{4} \g{4} \g{4} \]

\[ t_1^{[4]} \cdot t_2^{[4]} \]  
\[ \Downarrow \]  
\[ y_1^{[16]} [3 : 0] \]

Now, the neighbouring groups have to be shifted to their relative offset and are added:

\[ g_1^{[4]}, g_2^{[4]}, g_3^{[4]}, \text{ and } g_4^{[4]} \text{ are the groups of bits representing the bit-vector which would result from the single digit multiplication of } t_1^{[4]} = 0011 \text{ with the single bits of } t_1^{[4]} = 0101. \]

After the last step, only one group of bits is remaining and the lowest bits of the bit-vector \( y_1^{[16]} = 0000 0000 0000 1111 \) correspond to the solution of the multiplication, i.e. \( 0011 \cdot 0101 = y_1^{[16]} [3 : 0] = 1111. \)

Further examples for multiplication or for other operations can easily be generated by feeding our benchmark family of bit-vector operations encoded in SMT-LIB format to a SMT-solver.

C Example: A reduction of DQBF to QF_BV2

Consider the following DQBF formula:

\[ \forall u_0, u_1, u_2 \exists x(u_0), y(u_1, u_2). (x \lor y \lor \neg u_0 \lor \neg u_1) \land (x \lor \neg y \lor u_0 \lor \neg u_1 \lor u_2) \land (x \lor \neg y \lor \neg u_0 \lor u_1 \lor u_2) \land (\neg x \lor y \lor \neg u_0 \lor \neg u_2) \land (\neg x \lor \neg y \lor u_0 \lor u_1 \lor \neg u_2) \land \]

...
This DQBF formula is unsatisfiable. Let us note that by adding one more dependency for \( y \), or even by making \( x \) and \( y \) dependent on all \( u_i \)s, the resulting QBF formula becomes satisfiable.

Using the reduction in Sec. 4.2.1, this formula is translated to the following QF\_BV2 formula:

\[
\big((X \mid Y \mid \neg U_0 \mid \neg U_1) \& (X \mid Y \mid U_0 \mid \neg U_1 \mid U_2)\& (X \mid Y \mid U_0 \mid \neg U_1 \mid U_2)\& (\neg X \mid Y \mid U_0 \mid \neg U_1 \mid U_2)\& (\neg X \mid Y \mid U_0 \mid \neg U_1 \mid U_2)\big) = \neg \emptyset^{[8]} \wedge
\]

\[
\bigwedge_{i \in \{0,1,2\}} \left( (U_i \ll (1 \ll i)) + U_i \right) = \neg \emptyset^{[8]} \wedge
\]

\[
(X \& U_1) = ((X \gg_u (1 \ll 1)) \& U_1) \wedge
\]

\[
(X \& U_2) = ((X \gg_u (1 \ll 2)) \& U_2) \wedge
\]

\[
(Y \& U_0) = ((Y \gg_u (1 \ll 0)) \& U_0)
\]

In the following, let us show that this formula is also unsatisfiable. Note that \( U_0^{[3]} = 55_{16}^{[8]} = 01010101_{16}^{[8]} \), \( U_1^{[3]} = 33_{16}^{[8]} = 00110111_{16}^{[8]} \), and \( U_2^{[3]} = 0F_{16}^{[8]} = 00011111_{16}^{[8]} \), where “\( 16 \)” resp. “\( 2 \)” denotes hexadecimal resp. binary encoding of the binary magic numbers.

In the following, let us show that the formula \( \emptyset \) is also unsatisfiable. First, we show how the bits of \( X \) get restricted by the constraints introduced above. Let us denote the originally unrestricted bits of \( X \) with \( x_7, x_6, \ldots, x_0 \). Since the bit-vectors

\[
(X \& U_1) = (0,0,x_5,x_4,0,0,x_1,x_0)
\]

and

\[
((X \gg_u (1 \ll 1)) \& U_1) = (0,0,x_7,x_6,0,0,x_3,x_2)
\]

are forced to be equal, some bits of \( X \) should coincide, as follows:

\[
X := (x_5,x_4,x_5,x_4,x_1,x_0,x_1,x_0)
\]

Furthermore, considering also the equation of

\[
(X \& U_2) = (0,0,0,0,x_3,x_2,x_1,x_0)
\]

and

\[
((X \gg_u (1 \ll 2)) \& U_2) = (0,0,0,0,x_7,x_6,x_5,x_4)
\]

results in

\[
X := (x_1,x_0,x_1,x_0,x_1,x_0,x_1,x_0)
\]

In a similar fashion, the bits of \( Y \) are constrained as follows:

\[
Y := (y_6,y_5,y_4,y_4,y_2,y_0,y_0)
\]

In order to show that the formula \( \emptyset \) is unsatisfiable, let us evaluate the “clauses” in the formula:

\[
(X \mid Y \mid \neg U_0 \mid \neg U_1) = (1,1,1,x_0 \lor y_4,1,1,1,x_0 \lor y_0)
\]

\[
(X \mid \neg Y \mid U_0 \mid \neg U_1 \mid \neg U_2) = (1,1,1,1,1,1,x_1 \lor \neg y_0,1)
\]

\[
(X \mid \neg Y \mid U_0 \mid \neg U_1 \mid U_2) = (1,1,1,x_0 \lor \neg y_4,1,1,1,1)
\]

\[
(\neg X \mid Y \mid \neg U_0 \mid \neg U_2) = (1,1,1,1,1,\neg x_0 \lor y_2,1,\neg x_0 \lor y_0)
\]

\[
(\neg X \mid Y \mid U_0 \mid U_1 \mid \neg U_2) = (1,1,1,1,\neg x_1 \lor \neg y_2,1,1,1)
\]
By applying bitwise and to them, we get the bit-vector represented by the formula (9):

\[
\Phi' = \begin{pmatrix}
1 \\
1 \\
1 \\
(x_0 \lor \neg y_1) \land (x_0 \lor y_1) \\

\neg x_1 \lor \neg y_2 \\

\neg x_0 \lor y_2 \\
x_1 \lor \neg y_0 \\

(x_0 \lor y_0) \land (\neg x_0 \lor y_0)
\end{pmatrix} = \begin{pmatrix}
1 \\
1 \\
1 \\
x_0 \\
\neg x_1 \lor \neg y_2 \\
\neg x_0 \lor y_2 \\
x_1 \lor \neg y_0 \\
y_0
\end{pmatrix}
\]

In order to check if \( \Phi' = 0 \)\[^8\] is satisfiable, it is sufficient to try to satisfy the set of the above (propositional) clauses. It is easy to see that this clause set is unsatisfiable, since by unit propagation \( x_1 \) and \( y_2 \) must be 1, which contradicts with the clause \( \neg x_1 \lor \neg y_2 \).

D Example: A reduction of QBF to QF_BV2bw+

Consider the following QBF formula:

\[
\exists x \forall u_2 \exists y \forall u_1 \exists z . (u_2 \lor u_1 \lor \neg z) \land \\
(u_2 \lor \neg x \lor y) \land \\
(u_0 \lor \neg x \lor \neg z) \land \\
(u_1 \lor \neg y \lor z) \land \\
(u_0 \lor \neg u_1 \lor z)
\]

This QBF formula is satisfiable.

\[
(U_2 \mid U_1 \mid \neg Z) \& (U_2 \mid \neg X \mid Y) \& (U_0 \mid \neg X \mid \neg Z) \& \\
(U_1 \mid \neg Y \mid Z) \& (U_0 \mid \neg U_1 \mid \neg Z) = 0\[^8\] \land \\
\bigwedge_{i \in \{0,1,2\}} (U_i \ll 1) = \left( \bigwedge_{0 \leq j < i} U_j \right) \oplus U_i \land \\
(X \& \neg 1) = (X \ll 1) \land \\
(U_2' = \neg ((U_2 \ll 1) \oplus U_2)) \land ((Y \& U_2') = ((Y \ll 1) \& U_2'))
\]

In the following, let us show that this formula is also satisfiable. As in the previous example, we have \( U_0^3 = 01010101 \)\[^8\], \( U_1^3 = 00110011 \)\[^8\], and \( U_2^3 = 00001111 \)\[^8\]. However, this time the binary magic numbers were created in a different way to ensure that only addition and bitwise operations are used.

First, we show how the bits of \( X \) get restricted by the constraints introduced above. Let us denote the originally unrestricted bits of \( X \) with \( x_7, x_6, \ldots, x_0 \). Since the bit-vectors

\[
(X \& \neg 1) = (x_7, x_6, x_5, x_4, x_3, x_2, x_1, 0)
\]

and

\[
(X \ll 1) = (x_6, x_5, x_4, x_3, x_2, x_1, x_0, 0)
\]
all bits of \( X \) are forced to be equal:

\[
X := (x_0, x_0, x_0, x_0, x_0, x_0)
\]

Similarly we get the constraints on \( Y \):

\[
U'_2 := \sim ((U_2 \ll 1) \oplus U_2) = 11101110
\]

and therefore

\[
\begin{align*}
(Y & \& U'_2) = (y_7, y_6, y_5, 0, y_3, y_2, y_1, 0) \\
(Y \ll 1 & \& U'_2) = (y_6, y_5, y_4, 0, y_2, y_1, y_0, 0)
\end{align*}
\]

which are forced to be equal and again put restrictions on the individual bits of \( Y \):

\[
Y := (y_4, y_4, y_4, y_0, y_0, y_0, y_0)
\]

Finally, \( Z \) is not restricted in any way since the \( u_0 \) is the innermost universal variable that \( z \) depends on, i.e. \( z \) depends on all universal variables.

\[
Z := (z_7, z_6, z_5, z_4, z_3, z_2, z_1, z_0)
\]

In order to show that the formula \([10]\) is unsatisfiable, let us evaluate the “clauses” in the formula:

\[
\begin{align*}
(U_2 | U_1 | \sim Z) &= (\neg z_7, \neg z_6, 1, 1, 1, 1, 1, 1) \\
(U_2 | \sim X | Y) &= (\neg x_0 \lor y_4, \neg x_0 \lor y_4, \neg x_0 \lor y_4, \neg x_0 \lor y_4, 1, 1, 1, 1) \\
(U_0 | \sim X | \sim Z) &= (\neg x_0 \lor \neg z_7, 1, \neg x_0 \lor \neg z_5, 1, \neg x_0 \lor \neg z_3, 1, \neg x_0 \lor \neg z_1, 1) \\
(U_1 | \sim Y | Z) &= (\neg y_4 \lor z_7, \neg y_4 \lor z_6, 1, 1, \neg y_0 \lor z_4, \neg y_0 \lor z_3, 1, 1) \\
(U_0 | \sim U_1 | Z) &= (1, 1, z_5, 1, 1, 1, z_1, 1)
\end{align*}
\]

By applying \textit{bitwise and} to them, we get the bit-vector represented by the formula \([10]\):

\[
\Phi' = \begin{pmatrix}
\neg z_7 \land (\neg x_0 \lor y_4) \land (\neg x_0 \lor \neg z_7) \land (\neg y_4 \lor z_7) \\
\neg z_6 \land (\neg x_0 \lor y_4) \land (\neg y_4 \lor z_6) \\
(\neg x_0 \lor y_4) \land z_5 \\
\neg x_0 \lor y_4 \\
(\neg x_0 \lor \neg z_3) \land (\neg y_0 \lor z_4) \\
\neg y_0 \lor z_3 \\
(\neg x_0 \lor \neg z_1) \land z_1 \\
1
\end{pmatrix} = \begin{pmatrix}
\neg z_7 \land \neg y_4 \\
\neg z_6 \land \neg y_4 \\
z_5 \\
\neg x_0 \\
\neg y_0 \lor z_4 \\
\neg y_0 \lor z_3 \\
z_1 \\
1
\end{pmatrix}
\]

\( \Phi' = \sim 0^{[8]} \) can easily be satisfied, e.g. by setting

\[
\begin{align*}
z_7 &= z_6 = y_4 = y_0 = x_0 = 0 \\
z_5 &= z_1 = 1
\end{align*}
\]

Therefore,

\[
\begin{align*}
U_0 &= 01010101_2^{[8]}, & U_1 &= 00110011_2^{[8]}, & U_2 &= 00001111_2^{[8]}, \\
X &= 00000000_2^{[8]}, & Y &= 00000000_2^{[8]}, & Z &= 00110111_2^{[8]}
\end{align*}
\]

is a possible solution of the bit-vector formula.