Fast Three-Valued Abstract Bit-Vector Arithmetic *

Jan Onderka[1][0000−0003−2069−8584] and Stefan Ratschan[2][0000−0003−1710−1513]

[1] Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic
onderjan@fit.cvut.cz

[2] The Czech Academy of Sciences, Institute of Computer Science, Prague, Czech Republic
stefan.ratschan@cs.cas.cz

Abstract. Abstraction is one of the most important approaches for reducing the number of states in formal verification. An important abstraction technique is the usage of three-valued logic, extensible to bit-vectors. The best abstract bit-vector results for movement and logical operations can be computed quickly. However, for widely-used arithmetic operations, efficient algorithms for computation of the best possible output have not been known up to now.

In this paper, we present new efficient polynomial-time algorithms for abstract addition and multiplication with three-valued bit-vector inputs. These algorithms produce the best possible three-valued bit-vector output and remain fast even with 32-bit inputs.

To obtain the algorithms, we devise a novel modular extreme-finding technique via reformulation of the problem using pseudo-Boolean modular inequalities. Using the introduced technique, we construct an algorithm for abstract addition that computes its result in linear time, as well as a worst-case quadratic-time algorithm for abstract multiplication. Finally, we experimentally evaluate the performance of the algorithms, confirming their practical efficiency.

Keywords: Formal verification · Three-valued abstraction · Computer arithmetics · Addition and multiplication · Pseudo-Boolean modular inequality

1 Introduction

In traditional microprocessors, the core operations are bitwise logical operations and fixed-point wrap-around arithmetic. Behaviour of programs in machine code can be formally verified by model checking, enumerating all possible system

* This work was supported by the Czech Technical University (CTU) grant No. SGS20/211/OHK3/3T/18 and institutional financing of the Institute of Computer Science (RVO:67985807).

states and transitions (*state space*) and then verifying their properties. Unfortunately, naïve exhaustive enumeration of states quickly leads to prohibitively large state spaces (*state space explosion*), making verification infeasible.

State space explosion may be mitigated by a variety of techniques. One of them is *abstraction*, where a more efficient state space structure preserving certain properties of the original is constructed [3, p. 17]. Typically, the formal verification requirement is that it should be impossible to prove anything not provable in the original state space (*soundness for true*), while allowing *overapproximation*, leading to the possibility of a false counterexample.

For machine-code model checking, three-valued abstract representation of bits was introduced in [7] where each abstract bit can have value "zero", "one", or "perhaps one, perhaps zero" (unknown). Using this abstraction, bit and bit-vector movement operations may be performed directly on abstract bits. Each movement operation produces a single abstract result, avoiding state space explosion. The caveat is that overapproximation is incurred as relationships between unknown values are lost.

Three-valued representation was further augmented in [11] via bitwise logic operations (AND, OR, NOT...) with a single abstract result, further reducing state space explosion severity. However, other operations still required *instantiation* of the unknown values to enumerate all concrete input possibilities, treating each arising output possibility as distinct. This would lead not only to output computation time increasing exponentially based on the number of unknown bits, but also to potential creation of multiple new states and the possibility of severe state space explosion. For example, an operation with two 32-bit inputs and a 32-bit output could require up to 2^{64} concrete operation computations and could produce up to 2^{32} new states.

The necessity of instantiation when encountering arithmetic operations had severely reduced usefulness of a microcontroller machine-code model checker with three-valued abstraction developed by one of the authors [8]. This prompted our research in performing arbitrary operations without instantiation, with emphasis on fast computation of results of arithmetic operations.

1.1 Our Contribution

In this paper, we formulate the *forward operation problem*, where an arbitrary operation performed on three-valued abstract bit-vector inputs results in a single three-valued abstract bit-vector output which preserves soundness of model checking. While the best possible output can always be found in worst-case time exponential in the number of three-valued input bits, this is slow for 8-bit binary operations and infeasible for higher powers of two.

To aid with construction of polynomial-time worst-case algorithms, we devise a novel *modular extreme-search* technique. Using this technique, we find a linear-time algorithm for abstract addition and a worst-case quadratic-time algorithm for abstract multiplication.

Our results will allow model checkers that use the three-valued abstraction technique to compute the state space faster and to manage its size by only performing instantiation when necessary, reducing the risk of state space explosion.

2 Related Work

Many-valued logics have been extensively studied on their own, including Kleene logic [6] used for three-valued model checking [11]. In [10], three-valued logic was used for static program analysis of 8-bit microcontroller programs. Binary decision diagrams (BDDs) were used to compress input-output relationships for arbitrary abstract operations. This resulted in high generation times and storage usage, making the technique infeasible to use with 16-bit or 32-bit operands. These restrictions are not present in our approach where we produce the abstract operation results purely algorithmically, but precomputation may still be useful for abstract operations with no known worst-case polynomial-time algorithms.

In addition to machine-code analysis and verification, multivalued logics are also widely used for register-transfer level digital logic simulation. The IEEE 1164 standard [5] introduces nine logic values, out of which '0' (zero), '1' (one), and 'X' (unknown) directly correspond to three-valued abstraction. For easy differentiation between concrete values and abstract values, we will use the IEEE 1164 notation in this paper, using single quotes to represent an abstract bit as well as double quotes to represent an abstract bit-vector (tuple of abstract bits), e.g. "0X1" means ('0', 'X', '1'). While we primarily consider microprocessor machine-code model checking as our use case, we note that the presented algorithms also might be useful for simulation, automated test pattern generation, and formal verification of digital circuits containing adders and multipliers.

In [14], it was proposed that instantiation may be performed based only on interesting variables. For example, if a status flag "zero" is of interest, a tuple of values "XX" from which the flag is computed should be replaced by the possibilities {"00", "1X", "X1"}. This leads to lesser state space explosion compared to naïve instantiation, but is not relevant for our discussion as we discuss avoiding instantiation entirely during operation resolution.

In the paper, we define certain pseudo-Boolean functions and search for their global extremes. This is also called pseudo-Boolean optimization [2]. Problems in this field are often NP-hard. However, pseudo-Boolean functions for addition and multiplication that we will use in this paper have special forms that will allow us to resolve the corresponding problems in polynomial time without having to resort to advanced pseudo-Boolean optimization techniques.

3 Basic Definitions

Let us consider a binary concrete operation which produces a single M-bit output for each combination of two N-bit operands, i.e. $r : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{B}^M$. We define

the *forward operation problem* as the problem of producing a single abstract bit-vector output given supplied abstract inputs, preserving soundness. The output is not pre-restricted (the operation computation moves only *forward*). To preserve soundness, the abstract output must contain all possible concrete outputs that would be generated by first performing instantiation, receiving a set of concrete possibilities, and then performing the operation on each possibility.

To easily formalize this requirement, we first formalize three-valued abstraction using sets. Each three-valued abstract bit value ('0','1','X') identifies all possible values the corresponding concrete bit can take. We define the abstract bit as a subset of $\mathbb{B} = \{0, 1\}$ and the abstract bit values as

$$\text{`0'} \stackrel{\text{def}}{=} \{0\}, \text{`1'} \stackrel{\text{def}}{=} \{1\}, \text{`X'} \stackrel{\text{def}}{=} \{0, 1\}. \tag{1}$$

This formalization corresponds exactly to the meaning of 'X' as "possibly 0, possibly 1". Even though $\emptyset$ is also a subset of $\mathbb{B}$, it is never assigned to any abstract bit as there is always at least a single output possibility.

If an abstract bit is either '0' or '1', we consider it *known*; if it is 'X', we consider it *unknown*. For ease of representation in equations, we also introduce an alternative math-style notation $\hat{X} \stackrel{\text{def}}{=} \{0, 1\}$.

Next, we define abstract bit-vectors as tuples of abstract bits. For clarity, we use hat symbols to denote abstract bit-vectors and abstract operations. We use zero-based indexing for simplicity of representation and correspondence to typical implementations, i.e. $\hat{a}_0$ means the lowest bit of abstract bit-vector $\hat{a}$. We denote slices of the bit-vectors by indexing via two dots between endpoints, i.e. $\hat{a}_{0..2}$ means the three lowest bits of abstract bit-vector $\hat{a}$. In case the slice reaches higher than the most significant bit of an abstract bit-vector, we assume it to be padded with '0', consistent with interpretation as an unsigned number.

3.1 Abstract Bit Encodings

In implementations of algorithms, a single abstract bit may be represented by various *encodings*. First, we formalize a *zeros-ones* encoding of abstract bit $\hat{a}_i$ using concrete bits $a_i^0 \in \mathbb{B}$, $a_i^1 \in \mathbb{B}$ via

$$a_i^0 = 1 \iff 0 \in \hat{a}_i,\ a_i^1 = 1 \iff 1 \in \hat{a}_i, \tag{2}$$

which straightforwardly extends to bit-vectors a^0, a^1. Assuming $\hat{a}$ has $A \in \mathbb{N}_0$ bits, $\hat{a} \in (2^{\mathbb{B}})^A$, while $a^0 \in \mathbb{B}^A$, $a^1 \in \mathbb{B}^A$, i.e. they are concrete bit-vectors.

We also formalize a *mask-value* encoding: the mask bit $a_i^{\text{m}} = 1$ exactly when the abstract bit is unknown. When the abstract bit is known, the value bit a_i^{v} corresponds to the abstract value (0 for '0', 1 for '1'), as previously used in [11]. For simplicity, we further require $a_i^{\text{v}} = 0$ if $a_i^{\text{m}} = 1$. We formalize the encoding of abstract bit $\hat{a}_i$ using concrete bits $a_i^{\text{m}} \in \mathbb{B}$, $a_i^{\text{v}} \in \mathbb{B}$ via

$$a_i^{\text{m}} = 1 \iff 0 \in \hat{a}_i \wedge 1 \in \hat{a}_i,\ a_i^{\text{v}} = 1 \iff 0 \notin \hat{a}_i \wedge 1 \in \hat{a}_i, \tag{3}$$

which, again, straightforwardly extends to bit-vectors $a^{\mathrm{m}} \in \mathbb{B}^A$ and $a^{\mathrm{v}} \in \mathbb{B}^A$. We note that the encodings can be quickly converted via

$$\begin{aligned} a_i^0 = 1 &\iff a_i^{\mathrm{m}} = 1 \vee a_i^{\mathrm{v}} = 0,\ a_i^1 = 1 \iff a_i^{\mathrm{m}} = 1 \vee a_i^{\mathrm{v}} = 1, \\ a_i^{\mathrm{m}} = 1 &\iff a_i^0 = 1 \wedge a_i^1 = 1,\ a_i^{\mathrm{v}} = 1 \iff a_i^0 = 0 \wedge a_i^1 = 1. \end{aligned} \tag{4}$$

We note that when interpreting each concrete possibility in abstract bit-vector $\hat{a}$ as an unsigned binary number, a^{v} corresponds to the minimum, while a^1 corresponds to the maximum. For conciseness and intuitiveness, we will not explicitly note the conversions in the presented algorithms. Furthermore, where usage of arbitrary encoding is possible, we will write the hat-notated abstract bit-vector, e.g. $\hat{a}$.

3.2 Abstract Transformers

We borrow the notions defined in this subsection from abstract interpretation [4,12], adapting them for the purposes of this paper.

The set of concrete bit-vector possibilities given by a tuple containing A abstract bits, $\hat{a} \in (2^{\mathbb{B}})^A$, is given by a *concretization function* $\gamma : (2^{\mathbb{B}})^A \to 2^{(\mathbb{B}^A)}$,

$$\gamma(\hat{a}) \stackrel{\mathrm{def}}{=} \{a \in \mathbb{B}^A \mid \forall i \in \{0, \ldots, A-1\} \,.\, a_i \in \hat{a}_i\}. \tag{5}$$

Conversely, the transformation of a set of bit-vector possibilities $C \in 2^{(\mathbb{B}^A)}$ to a single abstract bit-vector $\hat{a} \in (2^{\mathbb{B}})^A$ is determined by an *abstraction function* $\alpha : 2^{(\mathbb{B}^A)} \to (2^{\mathbb{B}})^A$ which, to prevent underapproximation and to ensure soundness of model checking, must fulfill $C \subseteq \gamma(\alpha(C))$.

An abstract operation $\hat{r} : (2^{\mathbb{B}})^N \times (2^{\mathbb{B}})^N \to (2^{\mathbb{B}})^M$ corresponding to concrete operation $r : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{B}^M$ is an *approximate abstract transformer* if it overapproximates r, that is,

$$\forall \hat{a} \in (2^{\mathbb{B}})^N, \hat{b} \in (2^{\mathbb{B}})^N \,.\, \{r(a,b) \mid a \in \gamma(\hat{a}), b \in \gamma(\hat{b})\} \subseteq \gamma(\hat{r}(\hat{a}, \hat{b})). \tag{6}$$

The number of concrete possibilities $|\gamma(\alpha(C))|$ should be minimized to prevent unnecessary overapproximation. For three-valued bit-vectors, the best abstraction function α^{best} is uniquely given by

$$\forall i \in \{0, \ldots, A-1\} \,.\, (\alpha^{\mathrm{best}}(C))_i \stackrel{\mathrm{def}}{=} \{c_i \in \mathbb{B} \mid c \in C\}. \tag{7}$$

By using α^{best} to perform the abstraction on the minimal set of concrete results from Equation 6, we obtain the *best abstract transformer* for arbitrary concrete operation r, i.e. an approximate abstract transformer resulting in the least amount of overapproximation, uniquely given as

$$\hat{r}_k^{\mathrm{best}}(\hat{a}, \hat{b}) = \alpha^{\mathrm{best}}(\{r_k(a,b) \mid a \in \gamma(\hat{a}), b \in \gamma(\hat{b})\}). \tag{8}$$

We note that when no input abstract bit is $\emptyset$, there is at least one concrete result $r(a,b)$ and no output abstract bit can be $\emptyset$. Thus, three-valued representation is truly sufficient.

3.3 Algorithm Complexity Considerations

We will assume that the presented algorithms are implemented on a general-purpose processor that operates on binary machine words and can compute bitwise operations, bit shifts, addition and subtraction in constant time. Every bit-vector used fits in a machine word. This is a reasonable assumption, as it is likely that the processor used for verification will have machine word size equal to or greater than the processor that runs the program under consideration.

We also assume that the ratio of M to N is bounded, allowing us to express the presented algorithm time complexities using only N. Memory complexity is not an issue as the presented algorithms use only a fixed amount of temporary variables in addition to the inputs and outputs.

3.4 Naïve Universal Abstract Algorithm

Equation 8 immediately suggests a naïve algorithm for computing $\hat{r}^{\text{best}}$ for any given $\hat{a}, \hat{b}$: enumerating all $a, b \in 2^{(\mathbb{B}^N)}$, filtering out the ones that do not satisfy $a \in \gamma(\hat{a}) \ \wedge \ b \in \gamma(\hat{b})$, and marking the results of $r(a, b)$, which is easily done in the zeros-ones encoding. This naïve algorithm has a running time of $\Theta(2^{2^N})$.

Average-case computation time can be improved by only enumerating unknown input bits, but worst-case time is still exponential. Even for 8-bit binary operations, the worst-case input combination (all bits unknown) would require 2^{16} concrete operation computations. For 32-bit binary operations, it would require 2^{64} computations, which is infeasible. Finding worst-case polynomial-time algorithms for common operations is therefore of significant interest.

4 Formal Problem Statement

Theorem 1. *The best abstract transformer of abstract bit-vector addition is computable in linear time.*

Theorem 2. *The best abstract transformer of abstract bit-vector multiplication is computable in worst-case quadratic time.*

In Section 5, we will introduce a novel *modular extreme-finding technique* which will will use a basis for finding fast best abstract transformer algorithms. Using this technique, we will prove Theorems 1 and 2 by constructing corresponding algorithms in Sections 6 and 7, respectively. We will experimentally evaluate the presented algorithms to demonstrate their practical efficiency in Section 8.

5 Modular Extreme-Finding Technique

The concrete operation function r may be replaced by a pseudo-Boolean function $h : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{N}_0$ where the output of r is the output of h written in base 2. Surely, that fulfills

$$\begin{aligned} &\forall a \in \mathbb{B}^N, b \in \mathbb{B}^N, \forall k \in \{0, \ldots, M-1\} \ . \\ &\qquad r_k(a, b) = 1 \iff (h(a, b) \bmod 2^{k+1}) \geq 2^k. \end{aligned} \tag{9}$$

The best abstract transformer definition in Equation 8 is then equivalent to

$$\begin{aligned}\forall k \in \{0,\dots,M-1\}\,.\\ (0 \in \hat{r}_k^{\text{best}} \iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b})\,.\,(h(a,b) \bmod 2^{k+1}) < 2^k) \wedge \\ (1 \in \hat{r}_k^{\text{best}} \iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b})\,.\,(h(a,b) \bmod 2^{k+1}) \geq 2^k).\end{aligned} \tag{10}$$

The forward operation problem is therefore transformed into a problem of solving certain modular inequalities, which is possible in polynomial time for certain operations. We will later show that these include addition and multiplication.

If the inequalities were not modular, it would suffice to find the global minimum and maximum (extremes) of h. Furthermore, the modular inequalities in Equation 10 can be thought of as alternating intervals of length 2^k. Intuitively, if it was possible to move from the global minimum to the global maximum in steps of at most 2^k by using different values of $a \in \hat{a}, b \in \hat{b}$ in $h(a,b)$, it would suffice to find the global extremes and determine whether they are in the same 2^k interval. If they were, only one of the modular inequalities would be satisfied, resulting in known r_k (either '0' or '1'). If they were not, each modular inequality would be satisfied by some a, b, resulting in $r_k = \hat{X}$.

We will now formally prove that our reasoning for this *modular extreme-finding method* is indeed correct.

Lemma 1. *Consider a sequence of integers $t = (t_0, t_1, \dots, t_{T-1})$ that fulfills*

$$\forall n \in [0, T-2]\,.\,|t_{n+1} - t_n| \leq 2^k. \tag{11}$$

Then,

$$\begin{aligned}&\exists v \in [\min t, \max t]\,.\,(v \bmod 2^{k+1}) < 2^k \iff \\ &\exists n \in [0, T-1]\,.\,(t_n \bmod 2^{k+1}) < 2^k.\end{aligned} \tag{12}$$

Proof. As the sequence t is a subset of range $[\min t, \max t]$, the backward direction is trivial. The forward direction trivially holds if v is contained in t. If it is not, it is definitely contained in some range (v^-, v^+), where v^-, v^+ are successive values in the sequence t. Since $|v^+ - v^-| \leq 2^k$, $(v^- \bmod 2^{k+1}) < 2^k$, and $(v^+ \bmod 2^{k+1}) < 2^k$, the value v in range (v^-, v^+) definitely must also fulfill $(v \bmod 2^{k+1}) < 2^k$. □

Theorem 3. *Consider a pseudo-Boolean function $f : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{Z}$, two inputs $\hat{a}, \hat{b} \in (2^{\mathbb{B}})^N$, and a sequence $p = (p_0, p_1, \dots, p_{P-1})$ where each element is a pair $(a,b) \in (\gamma(\hat{a}), \gamma(\hat{b}))$, that fulfill*

$$\begin{aligned}&\forall n \in [0, P-2]\,.\,|f(p_{n+1}) - f(p_n)| \leq 2^k,\\ &f(p_0) = \min_{\substack{a \in \gamma(\hat{a})\\ b \in \gamma(\hat{b})}} f(a,b),\\ &f(p_{P-1}) = \max_{\substack{a \in \gamma(\hat{a})\\ b \in \gamma(\hat{b})}} f(a,b).\end{aligned} \tag{13}$$

Then,

$$\begin{aligned}\forall C \in \mathbb{Z} \,.\, (\exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \,.\, ((f(a,b)+C) \bmod 2^{k+1}) < 2^k \\ \iff \exists n \in [0, P-1] \,.\, ((f(p_n)+C) \bmod 2^{k+1}) < 2^k).\end{aligned} \tag{14}$$

Proof. Since each element of p is a pair $(a,b) \in (\gamma(\hat{a}), \gamma(\hat{b}))$, the backward direction is trivial. For the forward direction, use Lemma 1 to convert the sequence $(f(p_n)+C)_{n=0}^{P-1}$ to range $[f(p_0)+C, f(p_{P-1})+C]$ and rewrite the forward direction as

$$\begin{aligned}&\forall C \in \mathbb{Z} \,.\, (\exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \,.\, ((f(a,b)+C) \bmod 2^{k+1}) < 2^k \implies \\ &\exists v \in \left[\min_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} (f(a,b)+C) \,,\, \max_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} (f(a,b)+C) \right] \,.\, (v \bmod 2^{k+1}) < 2^k).\end{aligned} \tag{15}$$

The implication clearly holds, completing the proof. □

While Theorem 3 forms a basis for the modular extreme-finding method, there are two problems. First, finding global extremes of a pseudo-Boolean function is not generally trivial. Second, the *step condition*, that is, the absence of a step longer than 2^k in h, must be ensured. Otherwise, one of the inequality intervals could be "jumped over". For non-trivial operators, steps longer than 2^k surely are present in h for some k. However, instead of h, it is possible to use a tuple of functions $(h_k)_{k=0}^{M-1}$ where each one fulfills Equation 10 for a given k exactly when h does. This is definitely true if each h_k is congruent with h modulo 2^{k+1}.

Fast best abstract transformer algorithms can now be formed based on finding extremes of h_k, provided that h_k changes by at most 2^k when exactly one bit of input changes its value, which implies that a sequence p with properties required by Theorem 3 exists. For ease of expression of the algorithms, we define a function which discards bits of a number x below bit k (or, equivalently, performs integer division by 2^k),

$$\zeta_k(x) = \left\lfloor \frac{x}{2^k} \right\rfloor. \tag{16}$$

For conciseness, given inputs $\hat{a} \in (2^{\mathbb{B}})^N, \hat{b} \in (2^{\mathbb{B}})^N$, we also define

$$h_k^{\min} \stackrel{\text{def}}{=} \min_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} h_k(a,b), h_k^{\max} \stackrel{\text{def}}{=} \max_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} h_k(a,b), \tag{17}$$

Equation 10 then can be reformulated as follows: if $\zeta_k(h_k^{\min}) \neq \zeta_k(h_k^{\max})$, both inequalities are definitely fulfilled (as each one must be fulfilled by some element of the sequence) and output bit k is unknown. Otherwise, only one inequality is fulfilled, the output bit k is known and its value corresponds to $\zeta_k(h_k^{\min}) \bmod 2$. This forms the basis of Algorithm 1, which provides a general blueprint for fast abstract algorithms. Proper extreme-finding for the considered operation must

be added to the algorithm, denoted by $(\dots)$ in the algorithm pseudocode. We will devise extreme-finding for fast abstract addition and multiplication operations in the rest of the paper.

Algorithm 1 Modular extreme-finding abstract algorithm blueprint

function MODULAR_ALGORITHM_BLUEPRINT($\hat{a}, \hat{b}$)
  for $k \in \{0, \dots, M-1\}$ **do**
    $h_k^{\min} \leftarrow (\dots)$ ▷ Compute extremes of h_k
    $h_k^{\max} \leftarrow (\dots)$
    if $\zeta_k(h_k^{\min}) \neq \zeta_k(h_k^{\max})$ **then**
      $c_k \leftarrow \hat{X}$ ▷ Set result bit unknown
    else
      $c_k^{\mathrm{m}} \leftarrow 0, c_k^{\mathrm{v}} \leftarrow \zeta_k(h_k^{\min}) \bmod 2$ ▷ Set value
    end if
  end for
  return $\hat{c}$
end function

6 Fast Abstract Addition

To express fast abstract addition using the modular extreme-finding technique, we first define a function expressing the unsigned value of a concrete bit-vector a with an arbitrary number of bits A,

$$\Phi(a) \stackrel{\text{def}}{=} \sum_{i=0}^{A-1} 2^i a_i. \tag{18}$$

Pseudo-Boolean addition is then defined simply as

$$h^+(a, b) \stackrel{\text{def}}{=} \Phi(a) + \Phi(b). \tag{19}$$

To fulfill the step condition, we define

$$h_k^+(a, b) = \Phi(a_{0..k}) + \Phi(b_{0..k}). \tag{20}$$

This is congruent with h^+ modulo 2^{k+1}. The step condition is trivially fulfilled for every function h_k^+ in $(h_k^+)_{k=0}^{M-1}$, as changing the value of a single bit of a or b changes the result of h_k^+ by at most 2^k. We note that this is due to h^+ having a special form where only single-bit summands with power-of-2 coefficients are present. Finding the global extremes is trivial as each summand only contains a single abstract bit. Recalling Subsection 3.1, the extremes can be obtained as

$$\begin{aligned} h_k^{+,\min} &\leftarrow \Phi(a_{0..k}^{\mathrm{v}}) + \Phi(b_{0..k}^{\mathrm{v}}), \\ h_k^{+,\max} &\leftarrow \Phi(a_{0..k}^{1}) + \Phi(b_{0..k}^{1}). \end{aligned} \tag{21}$$

The best abstract transformer for addition is obtained by combining Equation 21 with Algorithm 1. Time complexity is trivially $\Theta(N)$, proving Theorem 1. Similar reasoning can be used to obtain fast best abstract transformers for subtraction and general summation, only changing computation of $h_k^{\min}$ and $h_k^{\max}$.

For further understanding, we will show how fast abstract addition behaves for "X0" + "11":

$$\begin{aligned}
&k = 0 : \text{“0” + “1”},\ 1 = \zeta_0(0+1) = \zeta_0(0+1) = 1 \rightarrow r_0 = \text{‘1’},\\
&k = 1 : \text{“X0” + “11”},\ 1 = \zeta_1(0+3) \neq \zeta_1(2+3) = 2 \rightarrow r_1 = \text{‘X’},\\
&k = 2 : \text{“0X0” + “011”},\ 0 = \zeta_2(0+3) \neq \zeta_2(2+3) = 1 \rightarrow r_2 = \text{‘X’},\\
&k > 2 : \zeta_k(h_k^{+,\min}) = \zeta_k(h_k^{+,\max}) = 0 \rightarrow r_k = \text{‘0’}.
\end{aligned} \tag{22}$$

For $M = 2$, the result is "XX1". For $M > 2$, the result is padded by '0' to the left, preserving the unsigned value of the output. For $M < 2$, the addition is modular. This fully corresponds to behaviour of concrete binary addition.

7 Fast Abstract Multiplication

Multiplication is typically implemented on microprocessors with three different input signedness combinations: unsigned × unsigned, signed × unsigned, and signed × signed, with signed variables using two's complement encoding. It is a well-known fact that the signed-unsigned and signed multiplication can be converted to unsigned multiplication by extending the signed multiplicand widths to product width using an arithmetic shift right. This could pose problems when the leading significant bit is 'X', but it can be split beforehand into two cases, '0' and '1'. This allows us to only consider unsigned multiplication in this section, signed multiplication only incurring a constant-time slowdown.

7.1 Obtaining a Best Abstract Transformer

Abstract multiplication could be resolved similarly to abstract addition by rewriting multiplication as addition of a sequence of shifted summands (long multiplication) and performing fast abstract summation. However, this does not result in a best abstract transformer. The shortest counterexample is "11" · "X1". Here, the unknown bit b_1 is added twice before influencing r_2, once as a summand in the computation of r_2 and once as a carryover from r_1:

$$\begin{array}{cccc}
(2^3) & (2^2) & (2^1) & (2^0) \\
 & & 1 & 1 \\
\cdot & b_1 & 1 & \\
\hline
(b_1) & (b_1) & b_1 & 1 \\
 & b_1 & 1 & \\
\hline\hline
b_1 & \mathbf{2b_1} & 1 + b_1 & 1
\end{array}$$

In fast abstract summation, the summand b_1 is treated as distinct for each output bit computation, resulting in unnecessary overapproximation of multiplication.

Instead, to obtain a fast best abstract transformer for multiplication, we apply the modular extreme-finding technique to multiplication itself, without intermediate conversion to summation. Fulfilling the maximum 2^k step condition is not as easy as previously. The multiplication output function h^* is defined as

$$h^*(a,b) \stackrel{\text{def}}{=} \Phi(a) \cdot \Phi(b) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 2^{i+j} a_i b_j. \tag{23}$$

One could try to use congruences to remove some summands from h_k^* while keeping all remaining summands positive. This would result in

$$h_k(a,b) = \sum_{i=0}^{k} \sum_{j=0}^{k-i} 2^{i+j} a_i b_j. \tag{24}$$

Changing a single bit a_i would change the result by $\sum_{j=0}^{k-i} 2^{i+j} b_j$. This sums to at most $2^{k+1} - 1$ and thus does not always fulfill the maximum 2^k step condition. However, the sign of the summand $2^k a_i b_{k-i}$ can be flipped due to congruence modulo 2^{k+1}, after which the change of result from a single bit flip is always in the interval $[-2^k, 2^k - 1]$. Therefore, to fulfill the maximum 2^k step condition, we define $h_k^* : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{Z}$ as

$$h_k^*(a,b) \stackrel{\text{def}}{=} \left(-\sum_{i=0}^{k} 2^k a_i b_{k-i} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i b_j \right). \tag{25}$$

For more insight into this definition, we will return to the counterexample to the previous approach, "11" · "X1", which resulted in unnecessary overapproximation for $k = 2$. Writing h_2^* computation as standard addition similarly to the previously shown long multiplication, the carryover of b_1 is counteracted by the summand $-2^2 b_1$:

(2^3)	(2^2)	(2^1)	(2^0)
	(b_1)	b_1	1
	$\mathbf{-b_1}$	1	
	$\mathbf{0}$	$1 + b_1$	1

It is apparent that $\zeta_2(h_2^{\min}) = \zeta_k(h_2^{\max}) = 0$ and unnecessary overapproximation is not incurred. Using that line of thinking, the definition of h_k^* in Equation 25 can be intuitively regarded as ensuring that the carryover of an unknown bit into the k-th column is neutralized by a corresponding k-th column summand. Consequently, if the unknown bit can appear only in both of them simultaneously, no unnecessary overapproximation is incurred.

While the maximum 2^k step condition is fulfilled in Equation 25, extreme-finding is much more complicated than for addition, becoming heavily dependent on abstract input bit pairs of form $(\hat{a}_i, \hat{b}_{k-i})$ where $0 \leq i \leq k$. Such pairs result in a summand $-2^k a_i b_{k-i}$ in h_k^*. When multiplication is rewritten using long

multiplication as previously, this summand is present in the k-th column. We therefore name such pairs *k-th column pairs* for conciseness.

In Subsection 7.2, we show that if at most one k-th column pair where $\hat{a}_i = \hat{b}_{k-i} = \hat{X}$ (*double-unknown pair*) exists, extremes of h_k^* can be found easily. In Subsection 7.3, we prove that if at least two double-unknown pairs exist, $r_k = \hat{X}$. Taken together, this yields a best abstract transformer algorithm for multiplication. In Subsection 7.4, we discuss implementation considerations of the algorithm with emphasis on reducing computation time. Finally, in Subsection 7.5, we present the final algorithm.

7.2 At Most One Double-Unknown k-th Column Pair

An extreme is given by values $a \in \hat{a}, b \in \hat{b}$ for which the value $h_k^*(a, b)$ is minimal or maximal (Equation 17). We will show that such a, b can be found successively when at most one double-unknown k-th column pair is present.

First, for single-unknown k-th column pairs where $\hat{a}_i = \hat{X}$, $\hat{b}_{k-i} \neq \hat{X}$, we note that in Equation 25, the difference between h_k^* when $a_i = 1$ and when $a_i = 0$ is

$$h_k^*(a, b \mid a_i = 1) - h_k^*(a, b \mid a_i = 0) = -2^k b_{k-i} + \sum_{j=0}^{k-i-1} 2^{i+j} b_j. \tag{26}$$

Since the result of the sum over j must be in the interval $[0, 2^k - 1]$, the direction of the change (negative or non-negative) is uniquely given by the value of b_{k-i}, which is known. It is therefore sufficient to ensure $a_i^{\min} \leftarrow b_{k-i}$ when minimizing and $a_i^{\min} \leftarrow 1 - b_{k-i}$ when maximizing. Similar reasoning can be applied to single-unknown k-th column pairs where $\hat{a}_i \neq \hat{X}$, $\hat{b}_{k-i} = \hat{X}$.

After assigning values to all unknown bits in single-unknown k-th column pairs, the only still-unknown bits are the ones in the only double-unknown k-th column pair present. In case such a pair $\hat{a}_i = \hat{X}$, $\hat{b}_j = \hat{X}, j = k - i$ is present, the difference between h_k^* when a_i and b_j are set to arbitrary values and when they are set to 0 is

$$\begin{aligned} &h_k^*(a, b) - h_k^*(a, b \mid a_i = 0, b_j = 0) = \\ &- 2^k a_i b_j + 2^i a_i \left(\sum_{z=0}^{j-1} 2^z b_z \right) + 2^j b_j \left(\sum_{z=0}^{i-1} 2^z a_z \right). \end{aligned} \tag{27}$$

When minimizing, it is clearly undesirable to choose $a_i^{\min} \neq b_j^{\min}$. Considering that the change should not be positive, $a_i^{\min} = b_j^{\min} = 1$ should be chosen if and only if

$$2^i \left(\sum_{z=0}^{j-1} 2^z b_z \right) + 2^j \left(\sum_{z=0}^{i-1} 2^z a_z \right) \leq 2^k. \tag{28}$$

When maximizing, it is clearly undesirable to choose $a_i^{\max} = b_j^{\max}$. That said, $a_i^{\max} = 1, b_j^{\max} = 0$ should be chosen if and only if

$$2^j \left(\sum_{z=0}^{i-1} 2^z a_z \right) \leq 2^i \left(\sum_{z=0}^{j-1} 2^z b_z \right). \tag{29}$$

Of course, the choice is arbitrary when both possible choices result in the same change. After the case of the only double-unknown k-th column pair present is resolved, there are no further unknown bits and thus, the values of h_k^* extremes can be computed as

$$\begin{aligned} h_k^{*,\min} &= \left(-\sum_{i=0}^{k} 2^k a_i^{\min} b_{k-i}^{\min} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i^{\min} b_j^{\min} \right), \\ h_k^{*,\max} &= \left(-\sum_{i=0}^{k} 2^k a_i^{\max} b_{k-i}^{\max} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i^{\max} b_j^{\max} \right). \end{aligned} \tag{30}$$

7.3 Multiple Double-Unknown k-th Column Pairs

Lemma 2. *Consider a sequence of integers $t = (t_0, t_1, \ldots, t_{T-1})$ that fulfills*

$$\forall n \in [0, T-2] \,.\, |t_{n+1} - t_n| \leq 2^k, t_0 + 2^k \leq t_{T-1}. \tag{31}$$

Then,

$$\exists n \in [0, T-1] \,.\, (t_n \bmod 2^{k+1}) < 2^k. \tag{32}$$

Proof. Use Lemma 1 to transform the claim to equivalent

$$\exists v \in [\min t, \max t] \,.\, (v \bmod 2^{k+1}) < 2^k. \tag{33}$$

Since $[t_1, t_1 + 2^k] \subseteq [\min t, \max t]$, such claim is implied by

$$\exists v \in [t_0, t_0 + 2^k] \,.\, (v \bmod 2^{k+1}) < 2^k. \tag{34}$$

As $[t_0, t_0 + 2^k] \bmod 2^{k+1}$ has $2^k + 1$ elements and there are only 2^k elements that do not fulfill $(v \bmod 2^{k+1}) < 2^k$, Equation 34 holds due to the pigeonhole principle. □

Corollary 1. *Given a sequence of integers $(t_0, t_1, \ldots, t_{T-1})$ that fulfills Lemma 2 and an arbitrary integer $C \in \mathbb{Z}$, the lemma also holds for sequence $(t_0 + C, t_1 + C, \ldots, t_{T-1} + C)$.*

Theorem 4. *Let $\hat{r}_k^{*,best}$ be the best abstract transformer of multiplication. Let $\hat{a}$ and $\hat{b}$ be such that there are p_1, p_2, q_1, q_2 in $\{0, \ldots, k\}$ where*

$$\begin{aligned} p_1 \neq p_2, p_1 + q_2 = k, p_2 + q_1 = k, \\ \hat{a}_{p_1} = \hat{X}, \hat{a}_{p_2} = \hat{X}, \hat{b}_{q_1} = \hat{X}, \hat{b}_{q_2} = \hat{X}. \end{aligned} \tag{35}$$

Then $\hat{r}_k^{best,*}(\hat{a}, \hat{b}) = \hat{X}$.

Proof. For an abstract bit-vector $\hat{c}$ with positions of unknown bits $u_1, \ldots, u_n$, denote the concrete bit-vector $c \in \gamma(\hat{c})$ for which $\forall i \in \{1, \ldots, n\} \;.\; c_{u_i} = s_i$ by $\gamma_{s_1,\ldots,s_n}(\hat{c})$. Let $\Phi_{s_1,\ldots,s_n}(\hat{c}) \stackrel{\text{def}}{=} \Phi(\gamma_{s_1,\ldots,s_n}(\hat{c}))$.

Now, without loss of generality, assume $\hat{a}$ only has unknown values in positions p_1 and p_2 and $\hat{b}$ only has unknown positions q_1, q_2 and $p_1 < p_2, q_1 < q_2$. Then, for $s_1, s_2, t_1, t_2 \in \mathbb{B}$, using $h(a, b) = \Phi(a) \cdot \Phi(b)$,

$$h(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b})) = (2^{p_1} s_1 + 2^{p_2} s_2 + \Phi_{00}(\hat{a})) \cdot (2^{q_1} t_1 + 2^{q_2} t_2 + \Phi_{00}(\hat{b})). \quad (36)$$

Define $A \stackrel{\text{def}}{=} \Phi_{00}(\hat{a})$ and $B \stackrel{\text{def}}{=} \Phi_{00}(\hat{b})$ and let them be indexable similarly to bit-vectors, i.e. $A_{0..z} = (A \mod 2^{z+1})$, $A_z = \zeta_z(A_{0..z})$. Define

$$\begin{aligned} &h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b})) \stackrel{\text{def}}{=} \\ &\quad 2^{p_1+q_1} s_1 t_1 + 2^{p_1+q_2} s_1 t_2 + 2^{q_1} t_1 A_{0..p_2-1} + 2^{p_1} s_1 B_{0..q_2-1} + \\ &\quad 2^{p_2+q_1} s_2 t_1 + 2^{p_2+q_2} s_2 t_2 + 2^{q_2} t_2 A_{0..p_1-1} + 2^{p_2} s_2 B_{0..q_1-1} + AB. \end{aligned} \quad (37)$$

As $A_{p_1} = A_{p_2} = B_{q_1} = B_{q_2} = 0$, h_k^{proof} and h are congruent modulo 2^{k+1}. Define

$$D(s_1, s_2, t_1, t_2) \stackrel{\text{def}}{=} h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b})) \;-\; h_k^{\text{proof}}(\gamma_{00}(\hat{a}), \gamma_{00}(\hat{b})). \quad (38)$$

As $p_1 + q_2 = k$ and $p_2 + q_1 = k$,

$$\begin{aligned} D(s_1, s_2, t_1, t_2) = 2^{p_1+q_1} s_1 t_1 + 2^k s_1 t_2 + 2^{q_1} t_1 A_{0..p_2-1} + 2^{p_1} s_1 B_{0..q_2-1} + \\ 2^k s_2 t_1 + 2^{p_2+q_2} s_2 t_2 + 2^{q_2} t_2 A_{0..p_1-1} + 2^{p_2} s_2 B_{0..q_1-1}. \end{aligned} \quad (39)$$

Set s_1, s_2, t_1, t_2 to specific chosen values and obtain

$$\begin{aligned} D(1,1,0,0) &= D(1,0,0,0) + D(0,1,0,0), \\ D(0,0,1,1) &= D(0,0,1,0) + D(0,0,0,1), \\ D(1,0,0,1) &= 2^k + D(1,0,0,0) + D(0,0,0,1). \end{aligned} \quad (40)$$

Inspecting the various summands, note that

$$\begin{aligned} D(1,0,0,0) \in [0, 2^k - 1],\; D(0,1,0,0) \in [0, 2^k - 1], \\ D(0,0,1,0) \in [0, 2^k - 1],\; D(0,0,0,1) \in [0, 2^k - 1], \\ D(1,1,0,0) - D(1,0,0,0) \in [0, 2^k - 1], \\ D(0,0,1,1) - D(0,0,1,0) \in [0, 2^k - 1]. \end{aligned} \quad (41)$$

Recalling Equation 10, the best abstract transformer can be obtained as

$$\begin{aligned} 0 \in \hat{r}_k^{\text{best}} &\iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \;.\; (h_k^{\text{proof}}(a, b) \bmod 2^{k+1}) < 2^k, \\ 1 \in \hat{r}_k^{\text{best}} &\iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \;.\; ((h_k^{\text{proof}}(a, b) + 2^k) \bmod 2^{k+1}) < 2^k. \end{aligned} \quad (42)$$

Constructing a sequence of $h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b}))$ that fulfills the conditions of Lemma 2 then implies that both inequalities can be fulfilled due to Corollary 1,

which will complete the proof. Furthermore, as $D(s_1, s_2, t_1, t_2)$ only differs from $h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b}))$ by the absence of summand AB that does not depend on the choice of s_1, s_2, t_1, t_2, it suffices to construct a sequence of $D(s_1, s_2, t_1, t_2)$ that fulfills Lemma 2 as well.

There is at least a 2^k step between $D(0,0,0,0)$ and $D(1,0,0,1)$. They will form the first and the last elements of the sequence, respectively. It remains to choose the elements in their midst so that there is at most 2^k step between successive elements.

Case 1. $D(0,1,0,0) \geq D(0,0,0,1)$. Considering Equations 40 and 41, a qualifying sequence is

$$(D(0,0,0,0), D(1,0,0,0), D(1,1,0,0), D(1,0,0,1)). \tag{43}$$

Case 2. $D(0,1,0,0) < D(0,0,0,1)$. Using Equation 39, rewrite the case condition to

$$2^{p_2-p_1} D(1,0,0,0) < 2^{q_2-q_1} D(0,0,1,0). \tag{44}$$

As $p_1 + q_2 = k, p_2 + q_1 = k$, it also holds that $q_2 - q_1 = p_2 - p_1$. Rewrite the case condition further to

$$2^{p_2-p_1} D(1,0,0,0) < 2^{p_2-p_1} D(0,0,1,0). \tag{45}$$

Therefore, $D(1,0,0,0) < D(0,0,1,0)$. Considering Equations 40 and 41, a qualifying sequence is

$$(D(0,0,0,0), D(0,0,1,0), D(0,0,1,1), D(1,0,0,1)). \tag{46}$$

This completes the proof. □

7.4 Implementation Considerations

There are some considerations to be taken into account for an efficient implementation of the fast multiplication algorithm.

The first question is how to detect the positions of single-unknown and double-unknown k-th column pairs. As such pairs have the form $2^k a_i b_{k-i}$, it is necessary to perform a bit reversal of one of the bit-vectors before bitwise logic operations can be used for position detection. Fortunately, it suffices to perform the reversal only once at the start of the computation. Defining the bit reversal of the first z bits of b as $\lambda(b, z) = (b_{z-1-i})_{i=0}^{z-1}$, when the machine word size $W \geq k+1$, reversal of the first $k+1$ bits (i.e. the bits in $b_{0..k}$) may be performed as

$$\lambda(b, k+1) = ((b_{k-i})_{i=0}^{k}) = ((b_{W-1-i})_{i=W-k-1}^{W-1}) = \lambda(b, W)_{W-k-1..W-1}. \tag{47}$$

It is thus possible to precompute $\lambda(b, W)$ and, for each k, obtain $\lambda(b, k+1)$ via a right shift through $W - k - 1$ bits, which can be performed in constant time. Furthermore, power-of-two bit reversals can be performed in logarithmic time

on standard architectures [1, p. 33-35], which makes computation of $\lambda(b, W)$ even more efficient.

The second problem is finding out whether multiple double-unknown k-th column pairs exist, and if there is only a single one, what is its position. While that can be determined trivially in linear time, a *find-first-set* algorithm can also be used, which can be implemented in logarithmic time on standard architectures [1, p. 9] and also is typically implemented as a constant-time instruction on modern processors.

The third problem, computation of h_k^* extremes in Equation 30, is not as easily mitigated. This is chiefly due to removal of summands with coefficients above 2^k due to 2^{k+1} congruence. While typical processors contain a single-cycle multiplication operation, we have not found an efficient way to use it for computation of Equation 25. To understand why this is problematic, computation of h_k^* with 3-bit operands and $k = 2$ can be visualised as

$$\begin{array}{ccccc}
(2^4) & (2^3) & (2^2) & (2^1) & (2^0) \\
 & & a_2 & a_1 & a_0 \\
\cdot & & b_2 & b_1 & b_0 \\
\hline
 & & (-a_2b_0) & a_1b_0 & a_0b_0 \\
 & \cancel{a_2b_1} & (-a_1b_1) & a_0b_1 & \\
\cancel{a_2b_2} & \cancel{a_1b_2} & (-a_0b_2) & & \\
\hline
 & & & & \dots
\end{array}$$

The striked-out operands are removed due to 2^{k+1} congruence, while the k-th column pair summands are subtracted instead of adding them. These changes could be performed via some modifications of traditional multiplier implementation (resulting in a custom processor instruction), but are problematic when only traditional instructions can be performed in constant time. Instead, we propose computation of h_k^* via

$$h_k^*(a, b) = \sum_{i=0}^{k} a_i \left(-2^k b_{k-i} + 2^i \Phi(b_{0..k-i-1})\right). \tag{48}$$

As each summand over i can be computed in constant time on standard architectures, $h_k^*(a, b)$ can be computed in linear time. Modified multiplication techniques with lesser time complexity such as Karatsuba multiplication or Schönhage–Strassen algorithm [13] could also be considered, but they are unlikely to improve practical computation time when N corresponds to the word size of normal microprocessors, i.e. $N \leq 64$.

7.5 Fast Abstract Multiplication Algorithm

Applying the previously discussed improvements directly leads to Algorithm 2. For conciseness, in the algorithm description, bitwise operations are denoted by the corresponding logical operation symbol, shorter operands have high zeros added implicitly, and the bits of $a^{\min}, a^{\max}, b^{\min}, b^{\max}$ above k are not used, so there is no need to mask them to zero.

Algorithm 2 Fast abstract multiplication algorithm

function FAST_ABSTRACT_MULTIPLICATION($\hat{a}, \hat{b}$)
  $a^{\mathrm{v}}_{\mathrm{rev}} \leftarrow \lambda(b^{\mathrm{v}}, W)$ ▷ Compute machine-word reversals for word size W
  $b^{\mathrm{v}}_{\mathrm{rev}} \leftarrow \lambda(b^{\mathrm{v}}, W)$
  $a^{\mathrm{m}}_{\mathrm{rev}} \leftarrow \lambda(a^{\mathrm{m}}, W)$
  $b^{\mathrm{m}}_{\mathrm{rev}} \leftarrow \lambda(b^{\mathrm{m}}, W)$
  for $k \in \{0, \ldots, M\}$ **do**
    $s_{\mathrm{a}} \leftarrow a^{\mathrm{m}} \wedge \neg b^{\mathrm{m}}_{\mathrm{rev},W-k-1..W-1}$ ▷ Single-unknown k-th c. pairs, 'X' in a
    $a^{\min} \leftarrow a^{\mathrm{v}} \vee (s_{\mathrm{a}} \wedge b^{\mathrm{v}}_{\mathrm{rev},W-k-1..W-1})$ ▷ Minimize such pairs
    $a^{\max} \leftarrow a^{\mathrm{v}} \vee (s_{\mathrm{a}} \wedge \neg b^{\mathrm{v}}_{\mathrm{rev},W-k-1..W-1})$ ▷ Maximize such pairs
    $s_{\mathrm{b}} \leftarrow b^{\mathrm{m}} \wedge \neg a^{\mathrm{m}}_{\mathrm{rev},W-k-1..W-1}$ ▷ Single-unknown k-th c. pairs, 'X' in b
    $b^{\min} \leftarrow b^{\mathrm{v}} \vee (s_{\mathrm{b}} \wedge a^{\mathrm{v}}_{\mathrm{rev},W-k-1..W-1})$ ▷ Minimize such pairs
    $b^{\max} \leftarrow b^{\mathrm{v}} \vee (s_{\mathrm{b}} \wedge \neg a^{\mathrm{v}}_{\mathrm{rev},W-k-1..W-1})$ ▷ Maximize such pairs
    $d \leftarrow a^{\mathrm{m}} \wedge b^{\mathrm{m}}_{\mathrm{rev},W-k-1..W-1}$ ▷ Double-unknown k-th column pairs
    if $\Phi(d) \neq 0$ **then** ▷ At least one double-unknown 2^k pair
      $i \leftarrow$ FIND_FIRST_SET(d)
      if $\Phi(d) \neq 2^i$ **then** ▷ At least two double-unknown k-th col. pairs
        $c_k \leftarrow \hat{X}$ ▷ Theorem 4
        continue
      end if
      $j \leftarrow k - i$ ▷ Resolve singular double-unknown k-th column pair
      if $2^i\Phi(b^{\min}_{0..j-1}) + 2^j\Phi(a^{\min}_{0..i-1}) \leq 2^k$ **then** ▷ Equation 28
        $a^{\min}_i \leftarrow 1$
        $b^{\min}_j \leftarrow 1$
      end if
      if $2^j\Phi(a^{\max}_{0..i-1}) \leq 2^i\Phi(b^{\max}_{0..j-1})$ **then** ▷ Equation 29
        $a^{\max}_i \leftarrow 1$
      else
        $b^{\max}_j \leftarrow 1$
      end if
    end if
    $h^{*,\min}_k \leftarrow 0$ ▷ Computed $a^{\min}, b^{\min}$, compute minimum of h^*_k
    $h^{*,\max}_k \leftarrow 0$ ▷ Computed $a^{\max}, b^{\max}$, compute maximum of h^*_k
    for $i \in \{0, \ldots, k\}$ **do** ▷ Compute each row separately
      if $a^{\min}_i = 1$ **then**
        $h^{*,\min}_k \leftarrow h^{*,\min}_k - (2^k b^{\min}_{k-i}) + (2^i\Phi(b^{\min}_{0..k-i-1}))$
      end if
      if $a^{\max}_i = 1$ **then**
        $h^{*,\max}_k \leftarrow h^{*,\max}_k - (2^k b^{\max}_{k-i}) + (2^i\Phi(b^{\max}_{0..k-i-1}))$
      end if
    end for
    if $\zeta_k(h^{*,\min}_k) \neq \zeta_k(h^{*,\max}_k)$ **then**
      $c_k \leftarrow \hat{X}$ ▷ Set result bit unknown

43: **else**
44: $c_k^{\mathrm{m}} \leftarrow 0, c_k^{\mathrm{v}} \leftarrow \zeta_k(h_k^{*,\min}) \bmod 2$ ▷ Set value
45: **end if**
46: **end for**
47: **return** $\hat{c}$
48: **end function**

Upon inspection, it is clear that the computation complexity is dominated by computation of $h_k^{\min}, h_k^{\max}$ and the worst-case time complexity is $\Theta(N^2)$, proving Theorem 2. Since the loops depend on M which does not change when signed multiplication is considered (only N does), signed multiplication is expected to incur at most a factor-of-4 slowdown when $2N$ fits machine word size, the possible slowdown occurring due to possible splitting of most significant bits of multiplicands (discussed at the start of Section 7).

8 Experimental Evaluation

We implemented the naïve universal algorithm, the fast abstract addition algorithm, and the fast abstract multiplication algorithm in the C++ programming language, without any parallelization techniques used. In addition to successfully checking equivalence of naïve and fast algorithm outputs for $N \leq 9$, we measured the performance of algorithms with random inputs. The implementation and measurement scripts are available in the accompanying artifact [9].

To ensure result trustworthiness, random inputs are uniformly distributed and generated using a C++ standard library Mersenne twister before the measurement. The computed outputs are assigned to a volatile variable to prevent their removal due to compile-time optimization. Each measurement is taken 20 times and corrected sample standard deviation is visualised.

The program was compiled with `GCC 9.3.0`, in 64-bit mode and with maximum speed optimization level `-O3`. It was ran on the conference-supplied virtual machine on a x86-64 desktop system with an AMD Ryzen 1500X processor.

8.1 Visualisation and Interpretation

We measured the CPU time taken to compute outputs for 10^6 random input combinations for all algorithms for $N \leq 8$, visualising the time elapsed in Figure 1. As expected, the naïve algorithm exhibits exponential dependency on N and the fast addition algorithm seems to be always better than the naïve one. The fast multiplication algorithm dominates the naïve one for $N \geq 6$. The computation time of the naïve algorithm makes its usage for $N \geq 16$ infeasible even if more performant hardware and parallelization techniques were used.

For the fast algorithms, we also measured and visualised the results up to $N = 32$ in Figure 2. Fast addition is extremely quick for all reasonable input sizes and fast multiplication remains quick enough even for $N = 32$. Fast multiplication results do not seem to exhibit a noticeable quadratic dependency.

We consider it plausible that as N rises, so does the chance that there are multiple double-unknown k-th column pairs for an output bit and it is set to 'X' quickly, counteracting the worst-case quadratic computation time.

Finally, we fixed $N = 32$, changing the independent variable to the number of unknown bits in each input, visualising the measurements in Figure 3. As expected, the fast multiplication algorithm exhibits a prominent peak with the easiest instances being all-unknown, as almost all output bits will be quickly set to 'X' due to multiple double-unknown k-th column pairs. Even at the peak around $N = 6$, the throughput is still above one hundred thousands computations per second, which should be enough for model checking usage.

In summary, while the naïve algorithm is infeasible for usage even with 16-bit inputs, the fast algorithms remain quick enough even for 32-bit inputs.

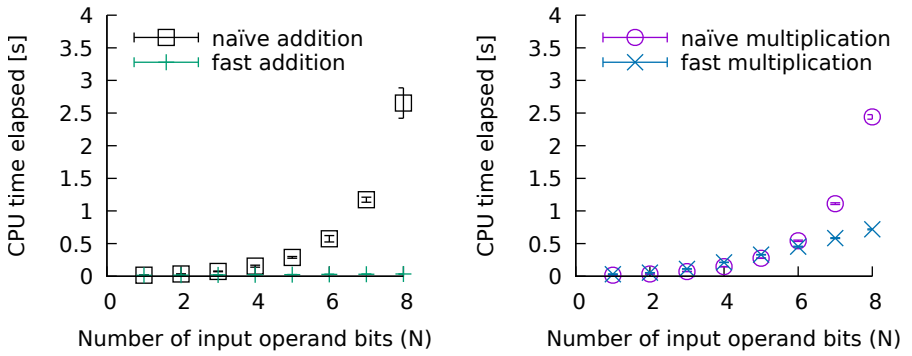

Fig. 1: Measured computation times for 10^6 random abstract input combinations.

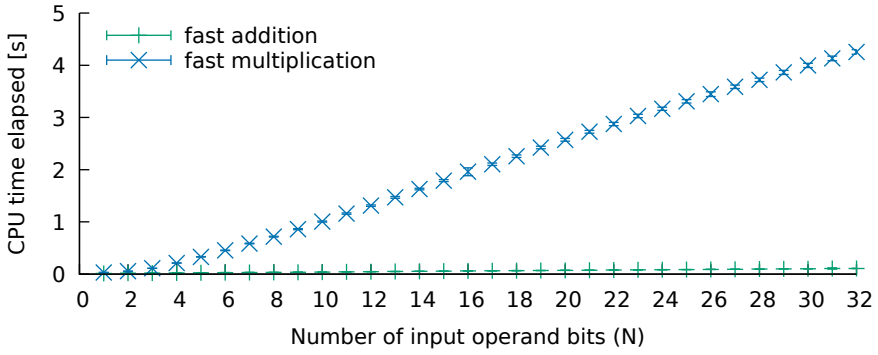

Fig. 2: Measured computation time for 10^6 random abstract input combinations, fast algorithms only.

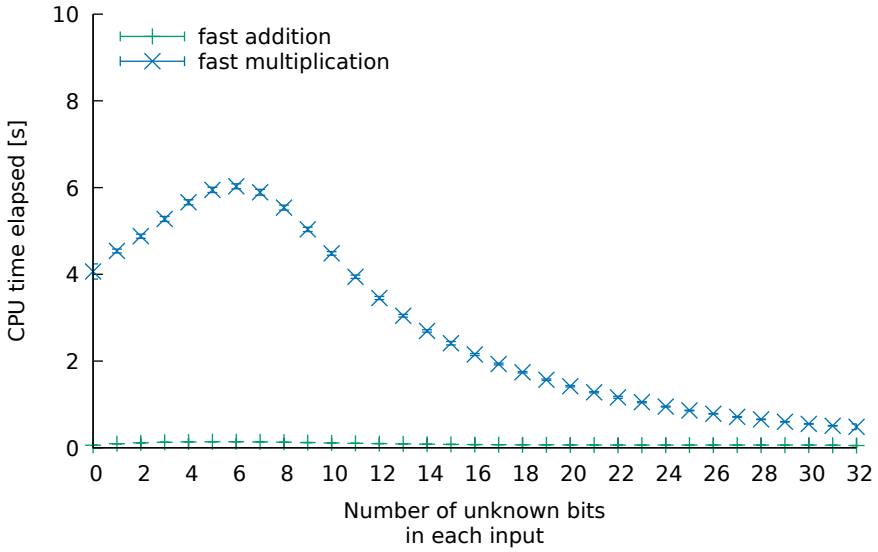

Fig. 3: Measured computation times for 10^6 random abstract input combinations with fixed $N = 32$, while the number of unknown bits in each input varies.

9 Conclusion

We devised a new *modular extreme-finding technique* for construction of fast algorithms which compute the best permissible three-valued abstract bit-vector result of concrete operations with three-valued abstract bit-vector inputs when the output is not restricted otherwise (*forward operation problem*). Using the introduced technique, we presented a linear-time algorithm for abstract addition and a worst-case quadratic algorithm for abstract multiplication. We implemented the algorithms and evaluated them experimentally, showing that their speed is sufficient even for 32-bit operations, for which naïve algorithms are infeasibly slow. As such, they may be used to improve the speed of model checkers which use three-valued abstraction.

There are various research paths that could further the results of this paper. Lesser-used operations still remain to be inspected, most notably the division and remainder operations. Composing multiple abstract operations into one could also potentially reduce overapproximation. Most interestingly, the forward operation problem could be augmented with pre-restrictions on outputs, which would allow not only fast generation of the state space in forward fashion, but its fast pruning as well, allowing fast verification via state space refinement. Furthermore, verification of hardware containing adders and multipliers could be improved as well, e.g. by augmenting Boolean satisfiability solvers with algorithms that narrow the search space when such a structure is found.

References

1. Arndt, J.: Bit wizardry, pp. 2–101. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10/cj6w5g
2. Boros, E., Hammer, P.L.: Pseudo-Boolean optimization. Discrete Applied Mathematics **123**(1), 155–225 (2002). https://doi.org/10/bbpz24
3. Clarke, E.M., Henzinger, T.A., Veith, H.: Introduction to Model Checking, pp. 1–26. Springer International Publishing, Cham (2018). https://doi.org/10/gvdh
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 269–282. POPL '79, Association for Computing Machinery, New York, NY, USA (1979). https://doi.org/10/bnxf32
5. Institute of Electrical and Electronics Engineers: IEEE standard multivalue logic system for VHDL model interoperability (std_logic_1164). IEEE Std 1164-1993 pp. 1–24 (1993). https://doi.org/10/bhz6s9
6. Kleene, S.C.: On notation for ordinal numbers. The Journal of Symbolic Logic **3**(4), 150–155 (1938). https://doi.org/10/drz5bh
7. Noll, T., Schlich, B.: Delayed nondeterminism in model checking embedded systems assembly code. In: Yorav, K. (ed.) Hardware and Software: Verification and Testing. pp. 185–201. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10/b6jbwx
8. Onderka, J.: Deadline Verification Using Model Checking. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology (2020), `http://hdl.handle.net/10467/87989`
9. Onderka, J.: Operation checker for fast three-valued abstract bit-vector arithmetic (Nov 2021). https://doi.org/10/g33p, companion artifact to this paper
10. Regehr, J., Reid, A.: Hoist: A system for automatically deriving static analyzers for embedded systems. SIGOPS Oper. Syst. Rev. **38**(5), 133–143 (Oct 2004). https://doi.org/10/fjcx9w
11. Reinbacher, T., Horauer, M., Schlich, B.: Using 3-valued memory representation for state space reduction in embedded assembly code model checking. In: 2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems. pp. 114–119 (2009). https://doi.org/10/dhg4ww
12. Reps, T., Thakur, A.: Automating abstract interpretation. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 3–40. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). https://doi.org/10/g498
13. Skiena, S.S.: Introduction to Algorithm Design, pp. 3–30. Springer London, London (2008). https://doi.org/10/cfjnv2
14. Yamane, S., Konoshita, R., Kato, T.: Model checking of embedded assembly program based on simulation. IEICE Transactions on Information and Systems **E100.D**(8), 1819–1826 (2017). https://doi.org/10/gbsnzd