Further Steps Down The Wrong Path : Improving the Bit-Blasting of Multiplication

Martin Brain[1]

[1]City, University of London, Northampton Square, London, EC1V 0HB, United Kingdom

Abstract

"Bit-blasting" : reducing terms in the theory of bit-vectors to formulae in propositional logic, is a popular and effective technique. For logical operations, comparisons and even bit-vector addition, it produces circuits or CNF clauses that are linear in the size of the bit-vector formula and unit propagation gives effective bit-wise reasoning. However bit-blasting is highly limited when it comes to multiplication. The formulae produced are at least one order of magnitude larger than other terms and introduce significant difficulty into the SAT problem. Even basic awareness of modular (2^n) arithmetic and the vast body of arithmetic, algebraic and cryptographic theorems on it, make it clear that bit-blasting is the wrong path for handling multiplication. In this work-in-progress paper we sketch two further steps along this wrong path, compacting multiplication by constants and showing the existence of incremental encodings of multiplication. It is hoped that these will not only get us closer to "the best that can be achieved given the limitations" but also that they might eventually connect to less limited, algebraic techniques.

1. Introduction

> No matter how far down the wrong road you've gone, turn back.
>
> – Turkish Proverb

"Bit-blasting" is a technique for reducing first-order terms and predicates in the theory of bit-vectors[1] to propositional logic. Every bit in each bit-vector term is represented by a propositional literal and clauses are added that link them. Often additional proposition variables, so called intermediate variables, are used to control the size and complexity of the encoding. This reduction allows first-order (or "word-level") equations to be solved using a SAT solver (or "bit-level" reasoning). Thus the immense (and on-going) improvements in SAT reasoning can be leveraged.

Bit-blasting is often regarded as characteristic of SMT as a whole. Even though it is just one component in some modern solvers, and there are solvers such as COLIBRI [1] and iSAT3 [2, 3] that support the theory of bit-vectors without bit-blasting. It is easy to explain, easy to visualise and works very well[2] It is not uncommon to find people who mistakenly believe that *all* of SMT is a variant on this technique.

SMT'21: 19th International Workshop on Satisfiability Modulo Theories July 18 - 19, 2021

[1]Or anything that can be reduced to these, such as uninterpreted functions and sorts, arrays, floating-point, integers, etc.

[2]... for applications that people try to solve with SMT solvers. Any statement of how well particular techniques work must be taken with the caveat that applications, benchmarks and solvers are all co-evolved.

Section 2 discusses the current state-of-the-art and explains why multiplication (and other "non-linear" operations) remain and likely will always be, cumbersome and difficult to handle via bit-blasting. Having tried to convince the reader that this is the wrong path, Sections 3 and 4 outline our work-in-progress on improving the performance and reducing the overheads of bit-blasting multiplication. No performance data is given as we only have prototype implementations and a number of algorithmic questions remain open. There is also a question over what would be appropriate benchmarks. Although it is easy to come up with synthetic benchmarks (factoring, associativity, etc.) it is not clear how relevant they are. As there has been a co-evolution of solvers and benchmarks, there is a shortage of graduated and calibrated non-linear bit-vector problems.

2. State-of-the-Art Bit-Blasting

Bit-blasting implementations tend to convert each term or predicate independently[3]. For each operator there is a small section of program which produces the bit-blasting. We will refer to the program and the output interchangeably as their *encoding* or the *circuit*. Some implementations produce CNF directly, but it is more common to produce a propositional circuit form. These may be And-Inverter Graphs (AIG) or they may allow other logical gates, with XOR being particularly useful. Bit-level constant propagation and rewriting is applied to this, either on-the-fly or after the whole expression has been converted. If circuits are produced an encoding, such as Plaisted-Greenbaum [5], is used to convert the intermediate form to propositional CNF[4]. Finally this is passed to a SAT solver, normally a CDCL solver but there are some evidence that other solver algorithms might be more effective [6].

The encodings used for each operator can be evaluated in terms of a number of metrics. The easiest but least predictive of performance is the size: either the number of clauses or the number of intermediate variables produced. More useful in the small scale is whether the encoding is *propagation complete* (with respect to unit propagation) [7, 8]. Given a partial assignment of the input and output bits[5], an encoding is propagation complete (with respect to unit propagation) when if unit propagation will infer all of the bits that are logically entailed.

Table 1 shows the current best known encodings for each of the bit-vector operators of length n. Operations that move and duplicate bits in a fixed pattern are effectively free because they can be encoded by simply renaming propositional literals. Likewise negation can be encoded by changing the polarity of the literals input. Operations that are 'bit-parallel' can be encoded with a circuit for each bit. So `bvand` needs one AND gate (3 clauses) per bit and `bvxor` needs one XOR gate (4 clauses) per bit. The non-standard bit-vector if-then-else (`bvite`) is bit-parallel but shows an interesting effect. The smallest encoding is not propagation complete. This happens with more complex operations, including multiplication. Smaller encodings do not

[3] Experience from [4] suggests that handling compound expressions would be an interesting source of improvements.

[4] Which, of course, is then pre-processed by the SAT solver. We will follow the conventions of the literature and ignore this important step.

[5] Whether assignments to intermediate variables are considered in propagation completeness is a subtle but significant point that is often ignored in the literature.

Operators	CNF	Aux. Vars.	Prop. Comp.
`concat`, `extract`, `repeat`	0	0	✓
`rotate_left`, `rotate_right`	0	0	✓
`zero_extend`, `sign_extend`	0	0	✓
`bvnot`	0	0	✓
`bvand`, `bvor`, `bvnand`, `bvnor`	$3n$	0	✓
`bvxor`, `bvxnor`	$4n$	0	✓
`bvite`	$4n$ / $6n$	0	$4n$× / $6n$ ✓
`bvcomp`	$4n + (n + 1)$	n	✓
`bvult`, `bvule`, `bvugt`, `bvuge`, `bvslt`, `bvsle`, `bvsgt`, `bvsge`	$6n$	n	✓
`bvneg`	$7n$	n	✓
`bvadd`, `bvsub`	$14n$	n	✓
`bvshl`, `bvlshr`, `bvashr`	$O(n \log(n))$	$O(n \log(n))$	×
`bvmul`, `bvudiv`, `bvurem`, `bvsdiv`, `bvsmod`	$O(n^2)$	$O(n^2)$	×

Table 1
The state-of-the-art in bit-vector encodings for bit-vectors of length n. Results are given in term of clauses required (CNF), auxiliary variables introduced (Aux. Vars.) and propagation completeness (Prop. Comp.)

necessarily propagate as well and encodings that propagate well are not necessarily small. Thus it is important to be clear about which metrics we are aiming to improve.

Most of the arithmetic operations make use of additional intermediate or auxiliary variables. For example in comparisions it is useful to have 'the previous m bits match' propositions. Likewise addition can be built from full-adders and intermediate 'carry' propositions.

The first operations for which no propagation complete encodings are known are shifts. The author believes that they can be created, but will likely increase the size of the encoding by a constant factor. Note that shifts by a constant amount are a special case and are effectively zero cost, like other static bit shuffling.

There are a very large number of possible encodings for multiplication. Without more useful metrics and ways of comparing them (a work-in-progress) it is hard to explore this space in a meaningful way. The most obvious (and popular) encoding is based on the shift-add algorithm:

```
bv multiplier_encoding(bv lhs, bv rhs) {
  int n = lhs.length();
  bv intermediate[n];

  intermediate[0] = and(repeat(n,lhs[0]), rhs);
  for (int i = 1; i < n; ++i) {
    intermediate[i] = add(intermediate[i-1],
                          lshift(and(repeat(n,lhs[i]), rhs), i));
  }
```

```
  return intermediate[n-1];
}
```

Here `bv` are data types that store a vector of propositional variables, one for each bit of the bit-vector. `repeat`, and, `lshift` and `plus` are functions that generate the appropriate encoding.

The intermediate variables need $n*n$ propositional variables, plus $n*n-1$ auxiliary variables from the additions. There are $n*3*n$ clauses from the and operations and $(n-1)*14*n$ clauses from the additions. If $n=32$ then this is $2,016$ variables and $16,960$ clauses. If $n=64$ then this is $8,128$ variables and $68,736$ clauses for a single multiplier!

Knuth [9] proposes use the use of Dadda multiplication reduction step so that it only uses $O(n^2)$ full adders rather than $O(n\log(n))$. Bitwuzla [10] uses a Wallace tree reduction step for this reason. In the author's experience this does give a reduction in the number of clauses and variables used but comes with a performance penalty.

Limits of Multiplier Encodings How much could this be improved? A number of results suggest that there are likely to be fundamental limits. Schönhage and Strassen [11] conjecture that the lower complexity bound for multiplication of n bit numbers is $n\log(n)$. As unit propagation is worst-case linear *in the size of the set of the clauses* and is sufficient to compute multiplication if the input bits are fixed, it seems implausible that there is an encoding that has less than $O(n\log(n))$ clauses and auxiliary variables. Recent work on $n\log(n)$ multipliers [12] suggests that the coefficients involved will be undesirable for most bit-blasting applications.

Reversing this argument, we can obtain a size bound on a propagation complete encoding of multiplication. Given $r=p*q$ with p and q prime, a propagation complete encoding could be used to factor r in at most n calls to unit-propagation by incrementally calling unit-propagation and then fixing bits of p if they are not set by the encoding. This would give a factoring algorithm that would run in $O(m^2)$ steps where m is the size of the set of clauses. As the best known algorithms for this task are pseudo-polynomial, the existence of polynomial-sized propagation complete multipliers seems unlikely. Our experience in [7, 8] supports this conjecture as minimal propagation complete encodings seem to show exponential growth in size.

So, having shown that improved encodings for multiplication[6] are likely to still be large and will only propagate better when very large, we conclude that algebraic techniques (such as incremental computation of Groebner bases) are much more likely to be effective and suggest future researchers pursue those.

In the rest of the paper we will ignore this advice.

3. Multiplication by Constant

For most bit-vector operators, constant propagation and some minor rewriting (`(bvand a a)` $\rightarrow$ a) during bit-blasting is enough produce optimised encodings when one operand is constant. When using the shift-add multiplier above, normalising symmetry so that `lhs` is the constant will reduce the number of additions to the number of `1`s in `lhs` minus one. For program analysis

[6] In the naïve encoding, divide and remainder are both even more expensive. However rewriting them using the defining identity $a/b*b+r=a$ reduces them to basically multiplication (if you ignore divide by zero...).

(apart from cryptography) this is generally sufficient as most constants are small. However for floating-point multiplication by constant, we need to consider more varied bit-vector constants.

Consider `127 * x`. The shift-add multiplier with constant propagation would give:

```
127 * x = x«6 + x«5 + x«4 + x«3 + x«2 + x«1 + x
```

However 127 is a difference between powers of 2, so we can compute it as:

```
127 * x = (128 - 1)*x = x«7 - x
```

So we have dropped from 6 additions down to just one. By treating contiguous sections of `1`s in this way we can reduce the number of additions to the number of parity changes in the number. This is a significant saving in the average case but the worst-case, `01010101...`, is still no better than the shift-add approach above.

This inspires our next trick; if the constant contains repeated patterns then we can use sharing of expressions to reduce the number of additions. For example:

```
0x5555 * x = (0x55«16 + 0x55) * x = (0x55 * x) « 16 + (0x55 * x).
```

Iterating this we can compute multiplication by `0x5555` using just 3 additions, rather than the 8 required previously. By combining these two techniques we can achieve an asymptotic (and significant) improvement over the shift-add encoding.

Unfortunately, it is not immediately obvious how to best combine these two; consider `0b10111011 * x`. If we use shift-add multiplication with constant rewriting then this will require 5 additions. Using subtraction of powers of two to create regions of `1`s we can compute it with 4 additions:

```
0b10111011 * x  =  x«7 + (0b111 * x)«3 + (0b11 *x)
                =  x«7 + (x«3 - x)«3 + (x«2 - x)
```

However using the repeated pattern trick first requires just 3 additions, a 40% reduction from our starting encoding:

```
0b10111011 * x  =  (0b1011 * x) « 4 + (0b1011 * x)
    0b1011 * x  =  x«3 + (x«2 - x)
```

Before developing an algorithm, it is worth considering the context. It is relatively rare for floating-point multiplications by constants to be encountered on their own. Weighted sums, vector and matrix multiplication and evaluating neural networks will often multiply *the same number* by several different constants. It is possible and desirable to share the same sub-terms between multiplications by different constants. Unfortunately [13] suggests that this problem is NP-complete, bringing us full circle and needing a SAT solver to compute the best encoding for a SAT solver!

4. Multiplication Using Polynomial Interpolation

A very common idiom in verification is "The full problem is very difficult or impossible, but how much of it do we actually need to do?". One way this is realised is via approximations. If the full bit-blasting is large and adds complexity to the SAT solver, why not use an approximation and refine as needed? Bryant et al. [14] proposes this idea but practical implementations run into problems due to the lack of incremental encodings of multiplication. Using a variant of the Toom-Cook multiplication algorithm [15, 16], with non-determinism rather than Gaussian elimination, it is possilbe to create a series of encodings that over-approximate multiplication and allow for incremental tightening of the approximation.

Consider multiplying two 16-bit numbers[7] `c = a * b`. We write each input as concatenations of 4-bit components, `a = a3:a2:a1:a0` and `b = b3:b2:b1:b0`. Rather than computing the multiplication directly, let p and q be polynomials in x such that $p(16) = \mathtt{a}$ and $q(16) = \mathtt{b}$:

$$\begin{aligned} p(x) &= \mathtt{a3}x^3 + \mathtt{a2}x^2 + \mathtt{a1}x + \mathtt{a0} \\ q(x) &= \mathtt{b3}x^3 + \mathtt{b2}x^2 + \mathtt{b1}x + \mathtt{b0} \end{aligned}$$

we will refer to these as *component polynomials*. If we define r to be the product of the component polynomials:

$$\begin{aligned} r(x) &= p(x) * q(x) \\ &= \mathtt{d6}x^6 + \mathtt{d5}x^5 + \mathtt{d4}x^4 + \mathtt{d3}x^3 + \mathtt{d2}x^2 + \mathtt{d1}x + \mathtt{d0} \end{aligned}$$

then:

$$\mathtt{a} * \mathtt{b} = p(16) * q(16) = r(16) = \mathtt{c}$$

By computing the coefficients of r we can evaluate the polynomial and obtain `c`:

$$\mathtt{c} = \mathtt{d6} \ll 24 + \mathtt{d5} \ll 20 + \mathtt{d4} \ll 16 + \mathtt{d3} \ll 12 + \mathtt{d2} \ll 8 + \mathtt{d1} \ll 4 + \mathtt{d0}$$

We could compute the coefficients of r directly but if we allow them to be non-deterministic, we can constrain the value of r on *any* 7 points and by polynomial interpolation, the coefficients will have the unique, correct value. Critically, we can add these 7 constraints incrementally during the solving process, with each reducing the degree of over-approximation. By picking our evaluation points to be small positive and negative powers of two, the size of the multiplications we perform are greatly reduced:

$$\begin{aligned} r(0) &= p(0) * q(0) \\ &= \mathtt{a0} * \mathtt{b0} \\ r(1) &= p(1) * q(1) \\ &= ((\mathtt{a2} + \mathtt{a0}) + (\mathtt{a3} + \mathtt{a1})) * ((\mathtt{b2} + \mathtt{b0}) + (\mathtt{b3} + \mathtt{b1})) \\ r(-1) &= p(-1) * q(-1) \end{aligned}$$

[7] For simplicity of explanation we will omit the extensions and computation of necessary bit-widths. This turns out to be a significantly more fiddly process than might be expected

$$
\begin{aligned}
&= ((\mathtt{a2} + \mathtt{a0}) - (\mathtt{a3} + \mathtt{a1})) * ((\mathtt{b2} + \mathtt{b0}) - (\mathtt{b3} + \mathtt{b1})) \\
r(2) &= p(2) * q(2) \\
&= ((\mathtt{a2} \ll 2 + \mathtt{a0}) + (\mathtt{a3} \ll 1 + \mathtt{a1}) \ll 1) * \\
&\quad ((\mathtt{b2} \ll 2 + \mathtt{b0}) + (\mathtt{b3} \ll 1 + \mathtt{b1}) \ll 1) \\
r(-2) &= p(-2) * q(-2) \\
&= ((\mathtt{a2} \ll 2 + \mathtt{a0}) - (\mathtt{a3} \ll 1 + \mathtt{a1}) \ll 1) * \\
&\quad ((\mathtt{b2} \ll 2 + \mathtt{b0}) - (\mathtt{b3} \ll 1 + \mathtt{b1}) \ll 1) \\
&\ldots
\end{aligned}
$$

This produces correct results but computing 7 ($= 2 * (n/components) - 1$) small multiplications instead of one large multiplications is not necessarily a saving. Extensive experimentation [8] suggests that it is a net reduction in clauses and variables for 64-bit multipliers and the 24/48 and 53/106 bit multipliers needed for `float32` and `float64`. But this is dependant on which evaluation points are used, the number of components, etc.

However, there remain a number of open questions about this approach:

- What is the best strategy for picking the component size? Longer components mean less constraints but make the multiplications in them larger. Small components mean more constraints and can reduce the size of the multiplication but evaluation at larger powers of 2 will negate this. Optimising for cases where we can use existing small propagation complete multipliers [7, 8] would seem promising.
- Is it possible to use the component polynomial of $\mathtt{c}$? What is the relationship between this order 7 polynomial and r?
- As presented we are computing a $2n$ bit output. For bit-vector multiplication we only need the low n bits, for floating-point we need the high $n + 1$ bits. Is it possible to find interpolation theorems specialised for these cases? Which evaluation points should be used and in what order for these different use-cases?
- Applying this recursively at non-zero evaluation points may give a way to incrementally expand the width of the multiplier, allowing this to be used for bit-blasting the theory of integers.

5. Conclusion

By blindly ignoring the right, algebraic path and continuing to try to step-wise improve the bit-blasting of multiplication we have found ourselves with a number of algebraic questions about the behaviour of polynomials over modulo 2^n arithmetic. The author considers this an entertainingly ironic place to finish a paper.

[8] Many thanks to Kevin Stefanov for his work on these.

Acknowledgments

The author would like to thank Kevin Stefanov for his work on a prototype of the Toom-Cook multiplier and Florian Schanda and the anonymous referees of SMT21 for their myriad of suggestions.

References

[1] B. Marre, B. Blanc, P. Mouy, Z. Chihani, F. Vedrine, F. Bobot, COLIBRI, SMT-COMP 2020 System Descriptions (2020).

[2] K. Scheibler, F. Neubauer, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, D. Fehrer, B. Becker, Accurate ICP-based Floating-point Reasoning, in: Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design, FMCAD '16, FMCAD Inc, Austin, TX, 2016, pp. 177–184. URL: http://dl.acm.org/citation.cfm?id=3077629.3077660.

[3] F. Neubauer, K. Scheibler, B. Becker, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, D. Fehrer, Accurate Dead Code Detection in Embedded C Code by Arithmetic Constraint Solving, in: E. Ábrahám, J. H. Davenport, P. Fontaine (Eds.), Proceedings of the 1st Workshop on Satisfiability Checking and Symbolic Computation, volume 1804 of *CEUR*, 2016, pp. 32–38.

[4] M. Brain, F. Schanda, Y. Sun, Building Better Bit-Blasting for Floating-Point Problems, in: T. s Vojnar, L. Zhang (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, volume 11427 of *LNCS*, Heidelberg, Germany, 2019, pp. 79–98. doi:10.1007/978-3-030-17462-0_5.

[5] D. Plaisted, S. Greenbaum, A Structure-Preserving Clause Form Translation, Journal of Symbolic Computation 2 (1986).

[6] V. Liew, P. Beame, J. Devriendt, J. Elffers, J. Nordström, Verifying Properties of Bit-vector Multiplication Using Cutting Planes Reasoning, in: Proceedings of FMCAD., 2020.

[7] M. Brain, L. Hadarean, D. Kroening, R. Martins, Automatic Generation of Propagation Complete SAT Encodings, in: B. Jobstmann, M. K. R. Leino (Eds.), Verification, Model Checking, and Abstract Interpretation, volume 9583 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2016, pp. 536–556. doi:10.1007/978-3-662-49122-5_26.

[8] M. Brain, L. Hadarean, D. Kroening, R. Martins, Stronger, Better, Faster: Optimally Propagating SAT Encodings, 2015. Presentation only paper at SMT15.

[9] D. E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability, 1st ed., Addison-Wesley Professional, 2015.

[10] A. Niemetz, M. Preiner, Bitwuzla at the SMT-COMP 2020, CoRR abs/2006.01621 (2020). URL: https://arxiv.org/abs/2006.01621. arXiv:2006.01621.

[11] A. Schönhage, V. Strassen, Schnelle multiplikation großer zahlen, Computing 7 (1971) 281–292.

[12] D. Harvey, J. van der Hoeven, Integer Multiplication in Time $o(n \log n)$, Annals of Mathematics 193 (2021) 563–617. doi:10.4007/annals.2021.193.2.4.

[13] Y. Voronenko, M. Püschel, Multiplierless multiple constant multiplication, ACM Transactions on Algorithms 3 (2007).

[14] R. Bryant, D. Kroening, J. Ouaknine, S. Seshia, O. Strichman, B. Brady, Deciding Bit-Vector Arithmetic with Abstraction, volume 4424 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2007, pp. 358–372. doi:10.1007/978-3-540-71209-1_28.

[15] A. L. Toom, The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers, Soviet Mathematics - Doklady (1963) 714–716.

[16] S. A. Cook, On the Minimum Computation Time of Functions, Ph.D. thesis, Harvard University, 1966.