Bit Vector Arithmetic

Nabarun Deka   Prathamesh Patil

18th May, 2021

Bit Vector Arithmetic

- A computer uses bit vectors to store information such as numbers. When we perform operations on these numbers, the computer does so on the bit vectors, which can lead to errors.
- For example, suppose we are working with 2 bit long bit vectors. We can represent the integer 3 as 11 and 1 as 01 in bit vector form. However when we add 3 and 1, we get 00 which is the representation of 0, since the extra bit is truncated.
- To make sense of such inaccuracies, we study bit vector arithmetic.

Syntax

Grammar

formula: formula $\wedge$ formula | $\neg$ formula | atom
atom: term rel term | Boolean Variable | term[constant]
rel : $<$ | $=$
term : term op term | Bit Vector | $\sim$ term | constant Bit Vector |
atom?term:term | term[constant: constant] | ext(term)
op : $+ | - | \cdot | / | << | >> | \& | \,|\, | \otimes | \circ$

- The grammar only mentions $\wedge$ and $\neg$ as boolean operators. We can create the rest, like $\vee$, using these two.
- Similarly, the grammar only allows us $<$ and $=$ as relational operators, but we can have the rest by taking Boolean combinations of the given operators.
- For example, we can have $a \leq b$ by writing $\neg(b < a)$.

Semantics

- **Bit Vector:** A bit vector b is a sequence of bits, which can have the value 0 or 1. The sequence has length l. The ith bit of b is denoted by b_i. The set of all bit vectors of length l is denoted by $bvec_l$.
- Each bit vector is associated with a **type**, which is the length of the bit vector and whether it is signed or unsigned.

Semantics

- **Binary Encoding (Unsigned):** A bit vector b of length l is a binary encoding of a natural number x iff

$$x = \langle b \rangle_U = \sum_{i=0}^{l-1} b_i \cdot 2^i$$

- **Two's Complement (Signed):** A bit vector b of length l is the two's complement of an integer x iff

$$x = \langle b \rangle_S = \sum_{i=0}^{l-2} b_i \cdot 2^i - b_{l-1} \cdot 2^{l-1}$$

- The difference is in the interpretation of the bit vector. For example, if $b = 1100$, then $\langle b \rangle_U = 8 + 4 = 12$ and $\langle b \rangle_S = -8 + 4 = -4$.

Semantics

- Observe that, for a given bit vector b of length l, $\langle b \rangle_U = \langle b \rangle_S \mod 2^l$.
- This is because, by definition,

$$\langle b \rangle_U = \sum_{i=0}^{l-1} b_i \cdot 2^i - 2^l \mod 2^l$$

which evaluates to

$$\langle b \rangle_U = \sum_{i=0}^{l-2} b_i \cdot 2^i - b_{l-1} \cdot 2^{l-1} \mod 2^l = \langle b \rangle_S \mod 2^l$$

Bitwise Operators

Bitwise operators are operators like & and | which operate on given bit vectors one bit at a time.

- **Bitwise And &:** Suppose a, b and c are bit vectors of length l. Then $c = a \& b$ iff
$$c_i = a_i \wedge b_i$$
- **Bitwise Or |:** Suppose a, b and c are bit vectors of length l. Then $c = a \mid b$ iff
$$c_i = a_i \vee b_i$$
- **Bitwise Negation $\sim$:** Suppose a and b are bit vectors of length l. Then $a = \sim b$ iff
$$a_i = \neg b_i$$
- **Bitwise XOR $\otimes$:** Suppose a, b and c are bit vectors of length l. Then $c = a \otimes b$ iff
$$c_i = (a_i \vee b_i) \wedge (\neg a_i \vee \neg b_i)$$

Arithmetic Operators

Arithmetic operators are operators like $+$ and $\cdot$ which operate on the numbers represented by given bit vectors.

- **Addition $+$:** Suppose a, b and c are bit vectors of length l. Then

$$a +_U b = c \iff \langle a \rangle_U + \langle b \rangle_U = \langle c \rangle_U \mod 2^l$$

$$a +_S b = c \iff \langle a \rangle_S + \langle b \rangle_S = \langle c \rangle_S \mod 2^l$$

- Observe that

$$\langle a \rangle_S + \langle b \rangle_S = \langle c \rangle_S \mod 2^l \iff \langle a \rangle_U + \langle b \rangle_U = \langle c \rangle_U \mod 2^l$$

  Thus, we may drop the subscript on the $+$ symbol.
- We define subtraction similarly to addition. Due to above observation, we need not write the subscript for $-$ symbol either.

Arithmetic Operators

- **Multiplication $\cdot$:** Suppose a, b and c are bit vectors of length l. Then

$$a \cdot_U b = c \iff \langle a \rangle_U \cdot \langle b \rangle_U = \langle c \rangle_U \mod 2^l$$

$$a \cdot_S b = c \iff \langle a \rangle_S \cdot \langle b \rangle_S = \langle c \rangle_S \mod 2^l$$

- Again, we may drop the subscript on the $\cdot$ symbol since

$$\langle a \rangle_S \cdot \langle b \rangle_S = \langle c \rangle_S \mod 2^l \iff \langle a \rangle_U \cdot \langle b \rangle_U = \langle c \rangle_U \mod 2^l$$

Arithmetic Operators

- **Division $/$:** Suppose a, b and c are bit vectors of length l. Then

$$a /_U b = c \iff \mathit{floor}(\langle a\rangle_U / \langle b\rangle_U) = \langle c\rangle_U \mod 2^l$$

$$a /_S b = c \iff \mathit{floor}(\langle a\rangle_S / \langle b\rangle_S) = \langle c\rangle_S \mod 2^l$$

- This definition runs into trouble when the vector b represents 0, since there can be no such c if $\langle b\rangle_U = 0$ or $\langle b\rangle_S = 0$. A possible solution may be to fix the quotient, that is c, to a bit vector constant when $\langle b\rangle_U = 0$ or $\langle b\rangle_S = 0$.

Relational Operators

Relational operators, like $<$ and $=$, compare the numbers represented by given bit vectors.

- **Equality** $=$ **:** Suppose a and b are bit vectors of length l. Then, $a = b$ iff

  $$\langle a \rangle_U = \langle b \rangle_U$$

  Clearly, $\langle a \rangle_U = \langle b \rangle_U$ is equivalent to $\langle a \rangle_S = \langle b \rangle_S$
- **Less than** $<$ **:** Suppose a and b are bit vectors of length l. Then $a < b$ if the number represented by a (according to the encoding corresponding to its type) is less than the number represented by b.
- For example, the signed bit vector $b = 1100$ is less than unsigned b since $\langle b \rangle_S = -4 < 12 = \langle b \rangle_U$.

The other relational operators, such as $\leq$, are interpreted similarly.

Shifting Operators

Shifting operators shift the bits in a given bit vector to the left or to the right.

- **Left Shift** $<<$ **:** Suppose a and c are bit vectors with length l and b is an unsigned bit vector such that $\langle b \rangle_U \leq l$. Then $a << b = c$ iff $c_i = a_{i - \langle b \rangle_U}$ when $i \geq \langle b \rangle_U$ and $c_i = 0$ otherwise.
- **Right Shift for unsigned bit vectors** $>>$ **:** Suppose a and c are unsigned bit vectors with length l and b is an unsigned bit vector such that $\langle b \rangle_U \leq l$. Then $a >> b = c$ iff $c_i = a_{i + \langle b \rangle_U}$ when $i < l - \langle b \rangle_U$ and $c_i = 0$ otherwise.
- **Right Shift for signed bit vectors** $>>$ **:** Suppose a and c are signed bit vectors with length l and b is an unsigned bit vector such that $\langle b \rangle_U \leq l$. Then $a >> b = c$ iff $c_i = a_{i + \langle b \rangle_U}$ when $i < l - \langle b \rangle_U$ and $c_i = a_{l-1}$ otherwise.

Other Operators

- **Extension Operator ext() :**Suppose a is a bit vector of length l and b is a bit vector of length m. Let $l \leq m$. Then,

$$ext_{[m]U}(a) = b \iff \langle a \rangle_U = \langle b \rangle_U$$

  Here, the subscript $[m]U$ on the operator indicates that the resulting bit vector has length m and is unsigned. We may switch U with S to get a signed bit vector.

- **Concatenation** $\circ$ **:**Suppose a is a bit vector of length l and b is a bit vector of length m and c is a bit vector of length $l + m$, then $c = a \circ b$ iff the first l bits of c form the bit vector a and the next m bits form the bit vector b.
- **Case Split atom?term:term :** Suppose a and b are bit vectors with some fixed length and c is an atom in bit vector arithmetic. Then $c?a : b$ evaluates to a if c is true and b otherwise.

Incremental Bit Flattening

- Depending on the size of the formula to check, the operators used in the formula and the length of the terms in the formula, directly applying bit flattening can end up being resource consuming and slow. This is due to the sheer number of constraints that need to be considered.
- On top of that, just creating a constraint corresponding to an arithmetic operation in a given formula can take a very long time, depending on the length of the bit vectors involved.
- Comparitively, constraints corresponding to bitwise and relational operations are more manageable, both when it comes to creating the constraints and solving them.
- For example, consider $a \cdot b = c \wedge b \cdot a \neq c \wedge x < y \wedge x > y$. This formula is clearly unsatisfiable, since we have $a \cdot b = c \wedge \neg(a \cdot b = c)$ and $x < y \wedge x > y$. Bit flattening, however, may take a lot of time to decide this formula, since it will create and check constraints for the arithmetic operators as well as the relational operators.

Incremental Bit Flattening

- Thus, we may only introduce the constraints corresponding to bitwise and relational operators only, and omit all other operations. We may then check if these constraints, along with the propositional skeleton are satisfiable or not using a propositional SAT solver.
- If the solver outputs UNSAT, then the original formula itslef must be unsatisfiable, since adding new constraints to a formula which is already unsatisfiable will not help.
- If the solver outputs SAT, then either the original formula is satisfiable OR it is unsatisfiable, but some of the omitted constraints are needed to show this.
- Thus, we may check if the satisfying assignment obtained is consistent with the omitted constraints. If it is, then the formula is satisfiable. If not, we can add some of the omitted constraints to our propositional formula and check satisfiability again.
- In the worst case, we end up checking satisfiablility of all the constraints, along with the propositional skeleton.

References

- Sections 6.1, 6.2, 6.3 : Bit Vector Arithmetic, Deciding Bit Vector Arithmetic with Flattening, Incremental Bit Flattening in 'Decision Procedures An Algorithmic Point of View' by Daniel Kroening and Ofer Strichman