Automated Reasoning 2020

Lecture 20: Theory of bitvectors

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2020-11-05

Finite width

There are no true integers in computers.

Numbers are stored in fixed-width bitvectors.

Example 20.1

$$(x - y \geq 0) \quad \not\equiv \quad x \geq y$$

We need precise reasoning for fixed-with bitvectors.

Topic 20.1

Theory of bitvectors

Sorts and variables

- Bitvector sorts

```
(_ BitVec num)
```

- Declaration of bitvectors

```
(declare-fun x3 () (_ BitVec 32))
```

- Declaring bitvector constants

```
(bv 1 4)
(bv #b0010001 7)
```

Operators on bitvectors

- Bitwise operators
- Vector operators
- Arithmetic operators

Arithmetic operations preserve vector length.

`concat` and `extract` operators are used to change bit lengths.

Bitwise operators

- `bvand: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
  - It takes two vectors of same length, and returns a vector that is bitwise and of inputs.
- similarly `bvor`, `bvxor`, `bvnot` are defined

Vector operators

- `concat: ((_ BitVec m) (_ BitVec n)) (_ BitVec m+n)`
- `(_ extract i j): ((_ BitVec m)) (_ BitVec i-j)`
- `(_ rotate_left i): ((_ BitVec n)) (_ BitVec n)`
- `(_ rotate_right i): ((_ BitVec n)) (_ BitVec n)`
- `(_ bvshl i): ((_ BitVec n)) (_ BitVec n)` shift left

Exercise 20.1

- `(concat 101 10)` =
- `((_ extract 1 4) 10110)` =
- `((_ rotate_left 2) 10110)` =
- `((_ rotate_right 2) 10110)` =
- `((_ bvshl 2) 10110)` =

Signed/unsigned interpration of bitvectors

Bitvectors as number. Let b be a bitvector of length n.

There are two well known ways to interpret b.

- Unsigned number:

$$u\_num(b) := 2^{n-1}b[n-1] + \cdots + 2^0 b[0]$$

- Singed number:

$$s\_num(b) := -2^{n-1}b[n-1] + 2^{n-2}b[n-2] + \cdots + 2^0 b[0]$$

In signed number, the highest bit indicate sign.

Sign aware vector operators

There are two kinds of shift right

- `bvlshr`
  - Pads 0 while shifting
- `bvashr`
  - Padded bits are copy of the highest bit

Two kinds of extension on the left size of bitvector

- `zero_extend`
- `sign_extend`

Exercise 20.2

- `((_ bvlshr 1) 10110)` =
- `((_ bvashr 1) 10110)` =
- `((_ zero_extend 2) 10110)` =
- `((_ sign_extend 2) 10110)` =

Exercise 20.3

a. Prove `sign_extend` preserves s_num value, b. Prove `zero_extend` preserves u_num value

Arithmetic Operators

Addition, subtraction, and multiplication need not know the interpretation

- `bvadd: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
- `bvmul: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
- `bvsub: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`

Definition 20.1

Let a and b be bitvectors of size n.

`u_num( (bvadd a b) ) = (u_num(a) + u_num(b)) mod` 2^n
`u_num( (bvsub a b) ) = (u_num(a) - u_num(b)) mod` 2^n
`u_num( (bvmul a b) ) = (u_num(a) * u_num(b)) mod` 2^n

Theorem 20.1

Let a and b be bitvectors of size n.

`s_num( (bvadd a b) ) mod` 2^n $=$ `(s_num(a) + s_num(b)) mod` 2^n
`s_num( (bvsub a b) ) mod` 2^n $=$ `(s_num(a) - s_num(b)) mod` 2^n
`s_num( (bvmul a b) ) mod` 2^n $=$ `(s_num(a) * s_num(b)) mod` 2^n

Arithmetic Operators II

Computing negative of a variable and division needs to be aware of the interpretation

- `bvneg: ((_ BitVec n)) (_ BitVec n)`

- `bvsrem: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
- `bvsdiv: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`

- `bvurem: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
- `bvudiv: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`

Signed Arithmetic comparators

- `bvslt : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvsgt : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvsle : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvsge : ((_ BitVec n) (_ BitVec n)) Bool`

Definition 20.2

Let a and b be bitvectors of size n.

`(bvslt a b)` $\Leftrightarrow$ `s_num(a) < s_num(b)`
`(bvsgt a b)` $\Leftrightarrow$ `s_num(a) > s_num(b)`
`(bvsle a b)` $\Leftrightarrow$ `s_num(a)` $\leq$ `s_num(b)`
`(bvsge a b)` $\Leftrightarrow$ `s_num(a)` $\geq$ `s_num(b)`

Unsigned Arithmetic comparators

- `bvult : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvugt : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvule : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvuge : ((_ BitVec n) (_ BitVec n)) Bool`

Definition 20.3

Let a and b be bitvectors of size n.

`(bvult a b)` $\Leftrightarrow$ `u_num(a) < u_num(b)`
`(bvugt a b)` $\Leftrightarrow$ `u_num(a) > u_num(b)`
`(bvule a b)` $\Leftrightarrow$ `u_num(a)` $\leq$ `u_num(b)`
`(bvuge a b)` $\Leftrightarrow$ `u_num(a)` $\geq$ `u_num(b)`

Topic 20.2

Solving bitvector formulas

BV theory solving

bitvector theory solvers work in the following two stages

- term rewriting
- bit blasting (SAT encoding)

Term rewriting to deal with high level structure and
bit blasting for unstructured boolean reasoning.

Term rewriting

Here are some rewriting phases that are helpful

- bvToBool: lift boolean statements to bitvector statements
  - For example:
    $(x[i:i]) \wedge ite(c, x[1], 0[1]) = 1[1]$ is translated to $(x[i:i] = 1[1]) \wedge ite(c, x[1] = 1[1], \bot)$
- ackermannize
  - For example: $\phi(f(x), f(y))$ is translated to $\phi(x', y') \wedge (x = y \Rightarrow x' = y')$
- algebraic pattern detection
  - For example: $x * x + x$ translated to $x * (x + 1)$

Refactoring isomorphic circuits

We may use information learned by term rewriting in clause learning

For example, refactoring isomorphic circuits

Consider the following formula

$$(x_0 = 2 * y_0 + y_1 \vee x_0 = 2 * y_1 + y_2 \vee x_0 = 2 * y_2 + y_0) \wedge \phi$$

We may observe that x_0 is assigned by expressions.

We recognize the pattern and rewrite the formula

$$\forall x, x'.\ f(x, x') = 2 * x + x' \wedge (x_0 = f(y_0, y_1) \vee x_0 = f(y_1, y_2) \vee x_0 = f(y_2, y_0)) \wedge \phi$$

How is it helpful?

Refactoring isomorphic circuits(contd.)

This information can be used multiple ways

- Learned clause reuse
  - Each application of f will produce isomorphic clauses after bit blasting.
  - A clause learned one set of clause can be translated to another learned clause for the isomorphic clause sets.
  - For example, if the clause learning detects first bit of x_0 and y_1 are equal since $x_0 = f(y_0, y_1)$. We may similar clauses to the other applications of f.
  - Minimize the encoding: the rewritten formula will have far less number of clauses

Topic 20.3

Bit blasting

Bit blasting: Translating to clauses

If high-level reasoning does not result in any answer.

In bit blasting, we convert every BV arithmetic expressions to Boolean clauses

Let us see a few such translations

Translating comparison

Consider `(bvult a b)`, where a and b are bitvectors of size n.

We can encode the bitvector formula into the following propositional formula.

$cmp(a, b, n) := (\neg a[n-1] \wedge b[n-1]) \vee (a[n-1] = b[n-1] \wedge cmp(a, b, n-1))$
$cmp(a, b, 0) := \bot$

Exercise 20.4

a. encode `(bvule a b)`
b. encode `(bvugt a b)`
c. encode `(bvslt a b)`

Translating addition

Consider `sum = (bvadd a b)`, where a and b are bitvectors of size n.

We can encode addition as follows

$carry[-1] = \bot$

$carry[i] := (a[i] \wedge b[i]) \vee ((a[i] \oplus b[i]) \wedge carry[i-1])$

$sum[i] \Leftrightarrow a[i] \oplus b[i] \oplus carry[i-1]$

Translating multiplication

Consider `res = (bvmul a b)`, where a and b are bitvectors of size n.

We can encode the multiplication as follows
$mul(a, b, -1) := b$
$mul(a, b, s) :=$ `(bvadd mul(a,b,s-1) (ite b[s] (bvshl a s) 0) )`

Exercise 20.5
What is the multiplier encoding in Z3, Boolector, and CBMC?

Translating division

Consider `d = (bvudiv a b)` and `r = (bvurem a b)`, where a and b are bitvectors of size n.

We can encode the unsigned division as follows

$$b \neq 0 \Rightarrow (mul(d, b, n) + r = a) \wedge r < b$$

Exercise 20.6

encode `d = (bvsdiv a b)` and `r = (bvsrem a b)`

Topic 20.4

Lazy bit blasting

Eagar bit blasting may hurt!

There are no ways to encode multiplications efficiently.

Example 20.2

The following formula is unsatisfiable for a simple reason.

$$a = b * c \wedge a < b \wedge a > b$$

But, the eager bit blasting blows up the encoding due to multiplication and may loose structure.

Bit blast on demand!

1. We treat bitvector operators as uninterpreted functions/relations, with other functional proprieties, e. g., `bvslt` is antisymmetric.

2. If the formula remains satisfiable, we find the operations that are violated by the current assignment.

3. If no such violation found, the formula is satisfiable.

4. We add bit blasted encoding of the violated operations and goto 2.

Commentary: We need not add encodings of all the violated operations immediately. As long as we add at least one in each iteration, the procedure is sound and terminates.

Topic 20.5

Modular arithmetic

Can we use linear arithmetic solver?

If there are not too many "bitwise operations" in input formulas,

we can translate the formulas into integer linear arithmetic constraints.

We need to be aware of the modular operations, also called overflows.

Encoding into linear arithmetic

We translate the bitvectors to integer constraints in the following steps.

1. Remove bitwise operations
2. Remove division
3. Remove constant multiplications
4. Replace additions with overflow aware additions

Naturally, if the input in not linear, we can not encode in the integer linear arithmetic.

Remove bitwise operations

- Translating `(bvnot b)`

$$-b + 1$$

- Translating `x = (bvand b 1)`

$$(b = 2y + x \wedge y < 2^{n-1} \wedge 0 \leq x \leq 1)$$

Exercise 20.7

a. Provide encoding for `bvshl`.
b. Provide encoding for `bvand` with arbitrary constants.
c. Provide encoding for `bvor` with arbitrary constants.

Remove division by constants

Translating `x = (bvudiv b k)`

$$b = x * k$$

Remove bitvector multiplication by constants

- For small constants repeat addition
- For large constants `(bvmul b c)` translates to

$$b * c - x2^n$$

with supporting constraints $b * c - x2^n < 2^n \wedge x < c - 1$

Removing bitvector addition

We replace addition `(bvadd s t)` as follows

$$ite(s + t < 2^n, s + t, s + t - 2^n)$$

Such a translation is not always helpful! It may end up introducing a large number of case splits.

End of Lecture 20