Outline

1. Introduction to Bit-Vector Logic
2. Syntax
3. Semantics
4. Decision procedures for Bit-Vector Logic
   - Flattening Bit-Vector Logic
   - Incremental Flattening

Decision Procedures for System-Level Software

What kind of logic do we need for system-level software?

- We need bit-vector logic – with bit-wise operators, arithmetic overflow
- We want to scale to large programs – must verify large formulas
- Examples of program analysis tools that generate bit-vector formulas:
  - CBMC
  - SATABS
  - F-Soft (NEC)
  - SATURN (Stanford, Alex Aiken)
  - EXE (Stanford, Dawson Engler, David Dill)
  - Variants of those developed at IBM, Microsoft

Bit-Vector Logic: Syntax

```
formula : formula ∨ formula | ¬formula | atom
atom : term rel term | Boolean-Identifier | term[constant]
rel : = | <
term : term op term | identifier | ~ term | constant | atom?term:term |
      term[ constant : constant ] | ext( term )
op : + | − | · | / | < < | >> | & | | ⊕ | ◦
```

- ∼x: bit-wise negation of x
- ext(x): sign- or zero-extension of x
- x << d: left shift with distance d
- x ◦ y: concatenation of x and y

Decisions

```
(x > y) ⇐⇒ (x − y > 0)
```

Valid over \( \mathbb{R}/\mathbb{N} \), but not over the bit-vectors.
(Many compilers have this sort of bug)
Width and Encoding

- The meaning depends on the width and encoding of the variables.
- Typical encodings:
  - Binary encoding
    \[ (x)_U := \sum_{i=0}^{l-1} a_i \cdot 2^i \]
  - Two’s complement
    \[ (x)_S := -2^{n-1} \cdot a_{n-1} + \sum_{i=0}^{l-2} a_i \cdot 2^i \]
- But maybe also fixed-point, floating-point, . . .

Examples

\[(11001000)_U = 200 \]
\[(11001000)_S = -128 + 64 + 8 = -56 \]
\[(01100100)_S = 100 \]

Width and Encoding

Notation to clarify width and encoding:

\[ x[32] \]

Width in bits

U: unsigned binary
S: signed two’s complement

Lambda-Notation for Bit-Vectors

\[ \lambda \] expressions are functions without a name

Examples:
- The vector of length \( l \) that consists of zeros:
  \[ \lambda i \in \{0, \ldots, l-1\}.0 \]
- A function that inverts (flips all bits in) a bit-vector:
  \[ \text{bv-invert}(x) := \lambda i \in \{0, \ldots, l-1\}.\neg x_i \]
- A bit-wise OR:
  \[ \text{bv-or}(x, y) := \lambda i \in \{0, \ldots, l-1\}.(x_i \lor y_i) \]

Example

\[(x[10] \circ y[5])[14] \iff x[9] \]

- This is translated as follows:
  \[ x[9] = x_9 \]
  \[ (x \circ y) = \lambda i \cdot (i < 5)?y_i : x_{i-5} \]
  \[ (x \circ y)[14] = (\lambda i \cdot (i < 5)?y_i : x_{i-5})(14) \]
  - Final result:
    \[ (\lambda i \cdot (i < 5)?y_i : x_{i-5})(14) \iff x_9 \]
**Semantics for Arithmetic Expressions**

What is the output of the following program?

```c
unsigned char number = 200;
number = number + 100;
printf("Sum: %d\n", number);
```

On most architectures, this is 44!

\[
\begin{align*}
11001000 &= 200 \\
01100100 &= 100 \\
\end{align*}
\]

\[
\begin{align*}
00101100 &= 44
\end{align*}
\]

\[ \Rightarrow \text{Bit-vector arithmetic uses modular arithmetic!} \]

**Semantics for Relational Operators**

Semantics for \(<\), \(\le\), \(\ge\), and so on:

\[
\begin{align*}
& a_U < b_U \iff (a)_U < (b)_U \\
& a_S < b_S \iff (a)_S < (b)_S
\end{align*}
\]

Mixed encodings:

\[
\begin{align*}
& a_U < b_S \iff (a)_U < (b)_S \\
& a_S < b_U \iff (a)_S < (b)_U
\end{align*}
\]

Note that most compilers don’t support comparisons with mixed encodings.

**Complexity**

- Satisfiability is undecidable for an unbounded width, even without arithmetic.

- It is NP-complete otherwise.

**A Simple Decision Procedure**

- Transform Bit-Vector Logic to Propositional Logic
- Most commonly used decision procedure
- Also called ‘bit-blasting’

**Bit-vector Flattening**

We denote the new Boolean variable for bit \(i\) of term \(t\) by \(\mu(t)_i\).

What constraints do we generate for a given term?

- This is easy for the bit-wise operators.

Example for \(a|g|b\):

\[
\bigwedge_{i=0}^{l-1} (\mu(t)_i = (a_i \lor b_i))
\]

(read \(x = y\) over bits as \(x \iff y\))

- We can transform this into CNF using Tseitin’s method.
Flattening Bit-Vector Arithmetic

How to flatten $a + b$?

$\rightarrow$ we can build a circuit that adds them!

Full Adder

$s \equiv (a + b + i) \mod 2 \equiv a \oplus b \oplus i$

$o \equiv (a + b + i) \div 2 \equiv a \cdot b + a \cdot i + b \cdot i$

The full adder in CNF:

$$(a \lor b \lor \neg o) \land (a \lor \neg b \lor i \lor \neg o) \land \neg a \lor b \lor \neg i \lor o \land \neg a \lor \neg b \lor o$$

Multipliers

- Multipliers result in very hard formulas
- Example:
  $$a \cdot b = c \land b \cdot a \neq c \land x < y \land x > y$$

CNF: About 11000 variables, unsolvable for current SAT solvers

- Similar problems with division, modulo
- Q: Why is this hard?
- Q: How do we fix this?

Incremental Flattening

- Idea: add 'easy' parts of the formula first
- Only add hard parts when needed
- $\varphi_f$ only gets stronger – use an incremental SAT solver

Incomplete Assignments

- Hey: initially, we only have the skeleton!
  How do we know what terms are inconsistent with the current assignment if the variables aren’t even in $\varphi_f$?

- Solution: guess some values for the missing variables.
  If you guess right, it’s good.

- Ideas:
  - All zeros
  - Sign extension for signed bit-vectors
  - Try to propagate constants $(a = b + 1)$