Bit-Vector Logic: Syntax

\[\begin{align*}
  \text{formula} & : \text{formula} \lor \text{formula} | \neg \text{formula} | \text{atom} \\
  \text{atom} & : \text{term} \ \text{rel} \ \text{term} | \text{Boolean-Identifier} | \text{term}[ \text{constant} ] \\
  \text{rel} & : = \ | < \\
  \text{term} & : \text{term} \ \text{op} \ \text{term} | \text{identifier} | \sim \ \text{term} | \text{constant} | \text{atom} ? \text{term} : \text{term} | \\
  \text{atom} ? \text{term} : \text{term} & \\
  \text{term}[ \text{constant} : \text{constant} ] & | \text{ext}(\text{term}) \\
  \text{op} & : + \ | - \ | \cdot \ | / \ | \ll \ | \gg \ | \& \ | \mid \ | \oplus \ | \circ
\end{align*}\]
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
A simple decision procedure

- Transform Bit-Vector Logic to Propositional Logic
- Most commonly used decision procedure
- Also called 'bit-blasting'
A simple decision procedure

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

### Bit-Vector Flattening

1. Convert propositional part as before
2. Add a *Boolean variable for each bit* of each sub-expression (term)
3. Add *constraint* for each sub-expression

We denote the new Boolean variable for \( i \) of term \( t \) by \( t_i \)
What constraints do we generate for a given term?
What constraints do we generate for a given atom

- This is easy for the bit-wise operators.
- Example for $t = a \mid b$

$$
\bigwedge_{i=0}^{l-1} t_i = (a_i \lor b_i)
$$

What about $x = y$
Flattening bit-vector arithmetic

How to flatten $s=a+b$
Flattening bit-vector arithmetic

How to flatten $s=a+b$

→ we can build a circuit that adds them!

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

**Full Adder**

\[
\begin{align*}
  s & \equiv (a + b + i) \mod 2 \equiv a \oplus b \oplus i \\
  o & \equiv (a + b + i) \div 2 \equiv a \cdot b + a \cdot i + b \cdot i
\end{align*}
\]

The full adder in CNF:

\[
(a \lor b \lor \neg o) \land (a \lor \neg b \lor i \lor \neg o) \land (a \lor \neg b \lor \neg i \lor o) \land
(\neg a \lor b \lor i \lor \neg o) \land (\neg a \lor \neg b \lor \neg i \lor o) \land (\neg a \lor \neg b \lor o)
\]
Ok, this is good for one bit! How about more?
Flattening bit-vector arithmetic

Ok, this is good for one bit! How about more?

8-Bit ripple carry adder (RCA)

- Also called carry chain adder
- Adds $l$ variables
- Add: $10^* l$ clauses 6 for o, 4 for s
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
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: How do we fix this?
Incremental flattening

\[ \varphi_f := \varphi_{sk}, \ F := \emptyset \]

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding

\( I \): set of terms that are inconsistent with the current assignment
Incremental flattening

\[ \varphi_f := \varphi_{sk}, \ F := \emptyset \]

Is \( \varphi_f \) SAT?

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding

\( I \): set of terms that are inconsistent with the current assignment
Incremental flattening

\[ \varphi_f := \varphi_{sk}, F := \emptyset \]

Is \( \varphi_f \) SAT?

No!

UNSAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding

\( I \): set of terms that are inconsistent with the current assignment
Incremental flattening

\[ \varphi_f := \varphi_{sk}, F := \emptyset \]

Is \( \varphi_f \) SAT?

- Yes! compute \( I \)
- No! UNSAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding

\( I \): set of terms that are inconsistent with the current assignment
\( \varphi_f := \varphi_{sk}, \, F := \emptyset \)

- Is \( \varphi_f \) SAT?
  - Yes! compute \( I \)
  - No! UNSAT

- \( I = \emptyset \)
- SAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding

\( I \): set of terms that are inconsistent with the current assignment
Incremental flattening

\[ \varphi_f := \varphi_{sk}, \quad F := \emptyset \]

Pick \( F' \subseteq (I \setminus F) \)

\[ F := F \cup F' \]

\[ \varphi_f := \varphi_f \land \text{CONSTRAINT}(F) \]

Is \( \varphi_f \) SAT?

Yes!

compute \( I \)

\( I \neq \emptyset \)

No!

UNSAT

\( I = \emptyset \)

SAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding

\( I \): set of terms that are inconsistent with the current assignment
Incremental flattening

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