universität innsbruck

SS 2024 lecture 10


# Constraint Solving

René Thiemann and Fabian Mitterwallner

based on a previous course by Aart Middeldorp

# Outline

# Outline

## Tightening

- convert $\sum a_i x_i < b$ to $\sum a_i x_i \leq b - 1$
- convert $\sum a_i x_i \leq b$ to $\sum \frac{a_i}{g} x_i \leq \lfloor \frac{b}{g} \rfloor$ where $g = \gcd(a_1, \ldots, a_n)$

## Unit-Cube Test

- check $A\vec{z} \leq \vec{b} - \frac{1}{2} \cdot \begin{pmatrix} |A_{11}| + \ldots + |A_{1n}| \\ \cdots \\ |A_{m1}| + \ldots + |A_{mn}| \end{pmatrix}$ by simplex
- if $\vec{z}$ is solution, then $\lfloor \vec{z} \rceil$ is integer solution of $A\vec{x} \leq \vec{b}$

## Equality Detection

- $A\vec{x} < \vec{b}$ satisfiable $\Longrightarrow$ no implied equality
- $\vec{a}_i \vec{x} < b_i$ part of minimal unsat core $\Longrightarrow$ $\vec{a}_i \vec{x} = b_i$ is implied equation
- reorder equation to $x_j = e_j$, store as part of solution, and eliminate $x_j$ elsewhere

## Diophantine Equation Solver

- return "unsat" whenever $E$ contains $\sum a_i x_i = b$ and $\gcd(a_1, \dots, a_n)$ does not divide $b$
- if $E$ contains equation $\sum a_i x_i = b$ with $|a_k| = 1$ then reorder equation to $x_k = \dots$, store as part of solution and substitute
- otherwise, pick coefficient $a_k$ with minimal absolute value
- rewrite equation as multiples of $a_k$ and remainders: $a_k(x_k + m) + r = 0$
- add equation $x_k = -m + x_t$ to solution for fresh variable $x_t$, and eliminate $x_k$

## Combined Solver

- first tighten, then perform equality detection and deletion
- tighten again, optionally detect new equalities and delete them
- try simplex as unsat criterion
- try unit-cube test as sat criterion
- decide satisfiability via branch-and-bound algorithm
- translate solution to initial set of variables via detected equalities

# Outline

1. **Summary of Previous Lecture**

2. **Bit-Vector Arithmetic**

   Theory  Flattening  Incremental Flattening

3. **Fixed-Point Arithmetic**

4. **Further Reading**

## Example (1) Change of Arithmetic: example formula $x - y > 0 \iff x > y$

- is valid over the integers

**Example** (1) Change of Arithmetic: example formula $x - y > 0 \iff x > y$

- is valid over the integers
- is not valid if $x$ and $y$ are interpreted as finite-width bit vectors (due to overflows)

## Example (1) Change of Arithmetic: example formula $x - y > 0 \iff x > y$

- is valid over the integers
- is not valid if $x$ and $y$ are interpreted as finite-width bit vectors (due to overflows)

## Example (2) Actual Program Behavior, e.g., for C

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

## Example (1) Change of Arithmetic: example formula $x - y > 0 \iff x > y$

- is valid over the integers
- is not valid if $x$ and $y$ are interpreted as finite-width bit vectors (due to overflows)

## Example (2) Actual Program Behavior, e.g., for C

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

prints 44 if variables of type unsigned char are represented by 8 bits

## Example (1) Change of Arithmetic: example formula $x - y > 0 \iff x > y$

- is valid over the integers
- is not valid if $x$ and $y$ are interpreted as finite-width bit vectors (due to overflows)

## Example (2) Actual Program Behavior, e.g., for C

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

prints 44 if variables of type unsigned char are represented by 8 bits:

$$\overset{200}{11001000} + \overset{100}{01100100} = \overset{44}{00101100}$$

**Example (1) Change of Arithmetic: example formula** $x - y > 0 \iff x > y$

- is valid over the integers
- is not valid if $x$ and $y$ are interpreted as finite-width bit vectors (due to overflows)

**Example (2) Actual Program Behavior, e.g., for C**

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

prints 44 if variables of type unsigned char are represented by 8 bits:

$$\overset{200}{11001000} + \overset{100}{01100100} = \overset{44}{00101100}$$

**Example (3) Hardware Verification**

Verify circuits whether they correctly implement bit-vector operations

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$
\begin{aligned}
\mathit{formula} &::= (\neg\,\mathit{formula}) \mid (\mathit{formula} \wedge \mathit{formula}) \mid (\mathit{formula} \vee \mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &::= \top \mid \bot \mid (\mathit{term}\ \mathit{rel}\ \mathit{term}) \\
\mathit{rel} &::= \ = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\mathit{term} &::= (\mathit{term}\ \mathit{binop}\ \mathit{term}) \mid (\mathit{unop}\ \mathit{term}) \mid \mathit{variable} \mid \mathit{constant} \mid \mathit{term}[i:j] \mid \\
&\qquad (\mathit{formula}\ ?\ \mathit{term} : \mathit{term}) \\
\mathit{binop} &::= + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \ \mid\ \mid \hat{}\ \mid\ :: \\
\mathit{unop} &::= \sim \mid -
\end{aligned}
$$

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$
\begin{array}{rcl}
\mathit{formula} & ::= & (\neg\, \mathit{formula}) \mid (\mathit{formula} \wedge \mathit{formula}) \mid (\mathit{formula} \vee \mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} & ::= & \top \mid \bot \mid (\mathit{term}\ \mathit{rel}\ \mathit{term}) \\
\mathit{rel} & ::= & = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\mathit{term} & ::= & (\mathit{term}\ \mathit{binop}\ \mathit{term}) \mid (\mathit{unop}\ \mathit{term}) \mid \mathit{variable} \mid \mathit{constant} \mid \mathit{term}[i:j] \mid \\
 & & (\mathit{formula}\ ?\ \mathit{term} : \mathit{term}) \\
\mathit{binop} & ::= & + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \mid \mid \hat{} \mid :: \\
\mathit{unop} & ::= & \sim \mid -
\end{array}
$$

## Notation

- variables $a_k$ are vectors $a_{k-1} \ldots a_1 a_0$ consisting of $k$ bits

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$
\begin{aligned}
\mathit{formula} &::= (\neg\,\mathit{formula}) \mid (\mathit{formula} \wedge \mathit{formula}) \mid (\mathit{formula} \vee \mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &::= \top \mid \bot \mid (\mathit{term}\ \mathit{rel}\ \mathit{term}) \\
\mathit{rel} &::= \ = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\mathit{term} &::= (\mathit{term}\ \mathit{binop}\ \mathit{term}) \mid (\mathit{unop}\ \mathit{term}) \mid \mathit{variable} \mid \color{red}{\mathit{constant}} \mid \mathit{term}[i:j] \mid \\
&\qquad (\mathit{formula}\ ?\ \mathit{term} : \mathit{term}) \\
\mathit{binop} &::= + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \ | \ \mid \hat{} \mid :: \\
\mathit{unop} &::= \sim \mid -
\end{aligned}
$$

## Notation

- variables $\mathrm{a}_k$ are vectors $a_{k-1} \dots a_1 a_0$ consisting of $k$ bits
- constant $\color{red}{\mathrm{n}_k}$ ($\color{red}{\mathrm{xn}_k}$) is binary representation of decimal (hexadecimal) $n$ in $k$ bits

## Examples

- $0_1 = 0$

## Examples

- $0_1 = 0 \quad 3_2 = 11$

## Examples

- $0_1 = 0 \quad 3_2 = 11 \quad 10_4 = 1010$

## Examples

- $0_1 = 0 \quad 3_2 = 11 \quad 10_4 = 1010 \quad 1024_{32} = 00000000000000000000010000000000$

## Examples

- $0_1 = 0 \quad 3_2 = 11 \quad 10_4 = 1010 \quad 1024_{32} = 00000000000000000000010000000000$
- $x0_4 = 0000$

## Examples

- $0_1 = 0$ $\quad 3_2 = 11$ $\quad 10_4 = 1010$ $\quad 1024_{32} = 00000000000000000000010000000000$
- $x0_4 = 0000$ $\quad xa_4 = 1010$

## Examples

- $0_1 = 0$  $3_2 = 11$  $10_4 = 1010$  $1024_{32} = 00000000000000000000010000000000$
- $\mathtt{x0}_4 = 0000$  $\mathtt{xa}_4 = 1010$  $\mathtt{xb0}_8 = 10110000$

## Examples

- $0_1 = 0$ $\quad 3_2 = 11$ $\quad 10_4 = 1010$ $\quad 1024_{32} = 00000000000000000000010000000000$
- $\mathsf{x0}_4 = 0000$ $\quad \mathsf{xa}_4 = 1010$ $\quad \mathsf{xb0}_8 = 10110000$ $\quad \mathsf{x11cf}_{16} = 0001000111001111$

## Examples

- $0_1 = 0$ $3_2 = 11$ $10_4 = 1010$ $1024_{32} = 00000000000000000000010000000000$
- $\mathtt{x0}_4 = 0000$ $\mathtt{xa}_4 = 1010$ $\mathtt{xb0}_8 = 10110000$ $\mathtt{x11cf}_{16} = 0001000111001111$

## Definition

- unsigned encoding (binary encoding) to represent non-negative integers:

$$\langle a_{k-1} \dots a_1 a_0 \rangle_u = \sum_{i=0}^{k-1} a_i 2^i$$

## Examples

- $0_1 = 0 \quad 3_2 = 11 \quad 10_4 = 1010 \quad 1024_{32} = 00000000000000000000010000000000$
- $\texttt{x0}_4 = 0000 \quad \texttt{xa}_4 = 1010 \quad \texttt{xb0}_8 = 10110000 \quad \texttt{x11cf}_{16} = 0001000111001111$

## Definition

- unsigned encoding (binary encoding) to represent non-negative integers:

$$\langle a_{k-1} \dots a_1 a_0 \rangle_u \; = \; \sum_{i=0}^{k-1} a_i 2^i$$

- two's complement to represent negative integers (signed encoding)

$$\langle a_{k-1} \dots a_1 a_0 \rangle_s \; = \; -a_{k-1} 2^{k-1} + \sum_{i=0}^{k-2} a_k 2^i$$

## Example

$$\langle 0000 \rangle_u = 0 = \langle 0000 \rangle_s$$

## Example

$\langle 0000 \rangle_u = 0 = \langle 0000 \rangle_s \qquad \langle 1010 \rangle_u = 10 \neq -6 = \langle 1010 \rangle_s$

## Example

$\langle 0000 \rangle_u = 0 = \langle 0000 \rangle_s \qquad \langle 1010 \rangle_u = 10 \neq -6 = \langle 1010 \rangle_s \qquad \langle 10110000 \rangle_s = -80$

## Example

$\langle 0000 \rangle_u = 0 = \langle 0000 \rangle_s$ $\quad$ $\langle 1010 \rangle_u = 10 \neq -6 = \langle 1010 \rangle_s$ $\quad$ $\langle 10110000 \rangle_s = -80$

## Example

- $1001 + 0011 = 1100$

**Example**

$\langle 0000 \rangle_u = 0 = \langle 0000 \rangle_s \qquad \langle 1010 \rangle_u = 10 \neq -6 = \langle 1010 \rangle_s \qquad \langle 10110000 \rangle_s = -80$

**Example**

- $1001 + 0011 = 1100$

  unsigned: $9 + 3 = 12$

### Example

$\langle 0000 \rangle_u = 0 = \langle 0000 \rangle_s \qquad \langle 1010 \rangle_u = 10 \neq -6 = \langle 1010 \rangle_s \qquad \langle 10110000 \rangle_s = -80$

### Example

- $1001 + 0011 = 1100$

  unsigned: $9 + 3 = 12$

  signed: $(-7) + 3 = -4$

## Example

$\langle 0000 \rangle_u = 0 = \langle 0000 \rangle_s \qquad \langle 1010 \rangle_u = 10 \neq -6 = \langle 1010 \rangle_s \qquad \langle 10110000 \rangle_s = -80$

## Example

- $1001 + 0011 = 1100$

  unsigned: $9 + 3 = 12$

  signed: $(-7) + 3 = -4$
- $1011 + 0110 = 0001$

## Example

$\langle 0000 \rangle_u = 0 = \langle 0000 \rangle_s \qquad \langle 1010 \rangle_u = 10 \neq -6 = \langle 1010 \rangle_s \qquad \langle 10110000 \rangle_s = -80$

## Example

- $1001 + 0011 = 1100$

  unsigned: $9 + 3 = 12$

  signed: $(-7) + 3 = -4$

- $1011 + 0110 = 0001$

  unsigned: $11 + 6 = 1 \text{ mod } 16$

## Example

$\langle 0000\rangle_u = 0 = \langle 0000\rangle_s$ $\langle 1010\rangle_u = 10 \neq -6 = \langle 1010\rangle_s$ $\langle 10110000\rangle_s = -80$

## Example

- $1001 + 0011 = 1100$

  unsigned: $9 + 3 = 12$

  signed: $(-7) + 3 = -4$

- $1011 + 0110 = 0001$

  unsigned: $11 + 6 = 1 \bmod 16$

  signed: $(-5) + 6 = 1$

## Example

$\langle 0000 \rangle_u = 0 = \langle 0000 \rangle_s$ $\langle 1010 \rangle_u = 10 \neq -6 = \langle 1010 \rangle_s$ $\langle 10110000 \rangle_s = -80$

## Example

- $1001 + 0011 = 1100$

  unsigned: $9 + 3 = 12$

  signed: $(-7) + 3 = -4$

- $1011 + 0110 = 0001$

  unsigned: $11 + 6 = 1 \bmod 16$

  signed: $(-5) + 6 = 1$

## Remark

addition (subtraction, multiplication) is independent of signed/unsigned encoding

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$
\begin{aligned}
\mathit{formula} &::= (\neg\, \mathit{formula}) \mid (\mathit{formula} \wedge \mathit{formula}) \mid (\mathit{formula} \vee \mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &::= \top \mid \bot \mid (\mathit{term}\ \mathit{rel}\ \mathit{term}) \\
\mathit{rel} &::= \ = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\mathit{term} &::= (\mathit{term}\ \mathit{binop}\ \mathit{term}) \mid (\mathit{unop}\ \mathit{term}) \mid \mathit{variable} \mid \mathit{constant} \mid \mathit{term}[i:j] \mid \\
&\quad\ \ (\mathit{formula}\ ?\ \mathit{term} : \mathit{term}) \\
\mathit{binop} &::= + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \texttt{|} \mid \hat{} \mid :: \\
\mathit{unop} &::= \sim \mid -
\end{aligned}
$$

## Notation

- variables $a_k$ are vectors $a_{k-1} \dots a_1 a_0$ consisting of $k$ bits
- constant $n_k$ ($xn_k$) is binary representation of decimal (hexadecimal) $n$ in $k$ bits

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$
\begin{aligned}
\textit{formula} &::= (\neg\, \textit{formula}) \mid (\textit{formula} \wedge \textit{formula}) \mid (\textit{formula} \vee \textit{formula}) \mid \textit{atom} \\
\textit{atom} &::= \top \mid \bot \mid (\textit{term rel term}) \\
\textit{rel} &::= \; = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\textit{term} &::= (\textit{term binop term}) \mid (\textit{unop term}) \mid \textit{variable} \mid \textit{constant} \mid \textit{term}[i:j] \mid \\
&\phantom{::=} (\textit{formula} \; ? \; \textit{term} : \textit{term}) \\
\textit{binop} &::= + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \; | \; \mid \hat{} \mid :: \\
\textit{unop} &::= \sim \mid -
\end{aligned}
$$

## Notation

- variables a$_k$ are vectors $a_{k-1} \dots a_1 a_0$ consisting of $k$ bits
- constant n$_k$ (xn$_k$) is binary representation of decimal (hexadecimal) $n$ in $k$ bits

## Definition (left shift)

$$\mathrm{a}_k \ll \mathrm{b}_k \;=\; \mathrm{c}_k \;\text{ with }\; c_i = \begin{cases} a_{i - \langle \mathrm{b}_k \rangle_u} & \text{if } i - \langle \mathrm{b}_k \rangle_u \geqslant 0 \\ 0 & \text{otherwise} \end{cases}$$

## Definition (left shift)

$$\mathsf{a}_k \ll \mathsf{b}_k \;=\; \mathsf{c}_k \;\text{ with }\; c_i = \begin{cases} a_{i - \langle \mathsf{b}_k \rangle_u} & \text{if } i - \langle \mathsf{b}_k \rangle_u \geqslant 0 \\ 0 & \text{otherwise} \end{cases}$$

## Example

$00101011 \ll 00000100 = 10110000$

## Definition (left shift)

$$\mathsf{a}_k \ll \mathsf{b}_k \;=\; \mathsf{c}_k \quad \text{with} \quad c_i = \begin{cases} a_{i - \langle \mathsf{b}_k \rangle_u} & \text{if } i - \langle \mathsf{b}_k \rangle_u \geqslant 0 \\ 0 & \text{otherwise} \end{cases}$$

## Example

$00101011 \ll 00000100 = 10110000$

## Definition (logical right shift)

$$\mathsf{a}_k \gg_u \mathsf{b}_k \;=\; \mathsf{c}_k \quad \text{with} \quad c_i = \begin{cases} a_{i + \langle \mathsf{b}_k \rangle_u} & \text{if } i + \langle \mathsf{b}_k \rangle_u < k \\ 0 & \text{otherwise} \end{cases}$$

## Definition (left shift)

$$\mathrm{a}_k \ll \mathrm{b}_k \;=\; \mathrm{c}_k \;\text{ with }\; c_i = \begin{cases} a_{i - \langle \mathrm{b}_k \rangle_u} & \text{if } i - \langle \mathrm{b}_k \rangle_u \geqslant 0 \\ 0 & \text{otherwise} \end{cases}$$

## Example

$00101011 \ll 00000100 = 10110000$

## Definition (logical right shift)

$$\mathrm{a}_k \gg_u \mathrm{b}_k \;=\; \mathrm{c}_k \;\text{ with }\; c_i = \begin{cases} a_{i + \langle \mathrm{b}_k \rangle_u} & \text{if } i + \langle \mathrm{b}_k \rangle_u < k \\ 0 & \text{otherwise} \end{cases}$$

## Example

$00101011 \gg_u 00000100 = 00000010$

## Definition (arithmetic right shift)

$$\mathrm{a}_k \gg_s \mathrm{b}_k \;=\; \mathrm{c}_k \;\text{ with }\; c_i = \begin{cases} a_{i+\langle \mathrm{b}_k \rangle_u} & \text{if } i + \langle \mathrm{b}_k \rangle_u < k \\ a_{k-1} & \text{otherwise} \end{cases}$$

## Definition (arithmetic right shift)

$$\mathrm{a}_k \gg_s \mathrm{b}_k \;=\; \mathrm{c}_k \;\text{ with }\; c_i = \begin{cases} a_{i+\langle \mathrm{b}_k \rangle_u} & \text{if } i + \langle \mathrm{b}_k \rangle_u < k \\ a_{k-1} & \text{otherwise} \end{cases}$$

## Example

$00101011 \gg_s 00000100 = 00000010$

## Definition (arithmetic right shift)

$$\mathrm{a}_k \gg_s \mathrm{b}_k \;=\; \mathrm{c}_k \;\text{ with }\; c_i = \begin{cases} a_{i+\langle \mathrm{b}_k \rangle_u} & \text{if } i + \langle \mathrm{b}_k \rangle_u < k \\ a_{k-1} & \text{otherwise} \end{cases}$$

## Example

$00101011 \gg_s 00000100 = 00000010$ $\qquad$ $10101011 \gg_s 00000100 = 11111010$

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$\begin{aligned}
\textit{formula} &::= (\neg\,\textit{formula}) \mid (\textit{formula} \wedge \textit{formula}) \mid (\textit{formula} \vee \textit{formula}) \mid \textit{atom} \\
\textit{atom} &::= \top \mid \bot \mid (\textit{term rel term}) \\
\textit{rel} &::= \; = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\textit{term} &::= (\textit{term binop term}) \mid (\textit{unop term}) \mid \textit{variable} \mid \textit{constant} \mid \textit{term}[i:j] \mid \\
&\phantom{::=}\; (\textit{formula} \;?\; \textit{term} : \textit{term}) \\
\textit{binop} &::= + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \,|\, \mid \hat{}\; \mid :: \\
\textit{unop} &::= \sim \mid -
\end{aligned}$$

## Notation

- variables a$_k$ are vectors $a_{k-1} \dots a_1 a_0$ consisting of $k$ bits
- constant n$_k$ (xn$_k$) is binary representation of decimal (hexadecimal) $n$ in $k$ bits

## Definition (arithmetic right shift)

$$\mathrm{a}_k \gg_s \mathrm{b}_k \;=\; \mathrm{c}_k \;\text{ with }\; c_i = \begin{cases} a_{i+\langle \mathrm{b}_k \rangle_u} & \text{if } i + \langle \mathrm{b}_k \rangle_u < k \\ a_{k-1} & \text{otherwise} \end{cases}$$

## Example

$00101011 \gg_s 00000100 = 00000010$ $\quad$ $10101011 \gg_s 00000100 = 11111010$

## Definitions (bitwise operations)

- $\mathrm{a}_k \;\&\; \mathrm{b}_k \;=\; \mathrm{c}_k$ with $c_i = a_i \wedge b_i$ for all $0 \leqslant i < k$ bitwise and

## Definition (arithmetic right shift)

$$\mathrm{a}_k \gg_s \mathrm{b}_k \;=\; \mathrm{c}_k \;\text{ with }\; c_i = \begin{cases} a_{i+\langle \mathrm{b}_k \rangle_u} & \text{if } i + \langle \mathrm{b}_k \rangle_u < k \\ a_{k-1} & \text{otherwise} \end{cases}$$

## Example

$00101011 \gg_s 00000100 = 00000010$ $\quad$ $10101011 \gg_s 00000100 = 11111010$

## Definitions (bitwise operations)

- $\mathrm{a}_k \;\&\; \mathrm{b}_k \;=\; \mathrm{c}_k$ with $c_i = a_i \wedge b_i$ for all $0 \leqslant i < k$ bitwise and
- $\mathrm{a}_k \;|\; \mathrm{b}_k \;=\; \mathrm{c}_k$ with $c_i = a_i \vee b_i$ for all $0 \leqslant i < k$ bitwise or

## Definition (arithmetic right shift)

$$a_k \gg_s b_k \; = \; c_k \;\text{ with }\; c_i = \begin{cases} a_{i+\langle b_k \rangle_u} & \text{if } i + \langle b_k \rangle_u < k \\ a_{k-1} & \text{otherwise} \end{cases}$$

## Example

$00101011 \gg_s 00000100 = 00000010$ $\qquad$ $10101011 \gg_s 00000100 = 11111010$

## Definitions (bitwise operations)

- $a_k \;\&\; b_k \; = \; c_k$ with $c_i = a_i \wedge b_i$ for all $0 \leqslant i < k$ bitwise and
- $a_k \;|\; b_k \; = \; c_k$ with $c_i = a_i \vee b_i$ for all $0 \leqslant i < k$ bitwise or
- $a_k \;\hat{}\; b_k \; = \; c_k$ with $c_i = a_i \oplus b_i$ for all $0 \leqslant i < k$ bitwise xor

## Definition (arithmetic right shift)

$$a_k \gg_s b_k \;=\; c_k \;\text{ with }\; c_i = \begin{cases} a_{i+\langle b_k\rangle_u} & \text{if } i + \langle b_k\rangle_u < k \\ a_{k-1} & \text{otherwise} \end{cases}$$

## Example

$00101011 \gg_s 00000100 = 00000010$ $\quad$ $10101011 \gg_s 00000100 = 11111010$

## Definitions (bitwise operations)

- $a_k \;\&\; b_k \;=\; c_k$ with $c_i = a_i \wedge b_i$ for all $0 \leqslant i < k$ — bitwise and
- $a_k \mid b_k \;=\; c_k$ with $c_i = a_i \vee b_i$ for all $0 \leqslant i < k$ — bitwise or
- $a_k \;\hat{}\; b_k \;=\; c_k$ with $c_i = a_i \oplus b_i$ for all $0 \leqslant i < k$ — bitwise xor
- $\sim a_k \;=\; c_k$ with $c_i = \neg a_i$ for all $0 \leqslant i < k$ — bitwise complement

## Examples

- $00101011 \,\&\, 01110101 = 00100001$

## Examples

- $00101011 \mathbin{\&} 01110101 = 00100001$
- $00101011 \mid 01110101 = 01111111$

## Examples

- $00101011 \mathbin{\&} 01110101 = 00100001$
- $00101011 \mid 01110101 = 01111111$
- $00101011 \mathbin{\hat{}} 01110101 = 01011110$

## Examples

- $00101011 \mathbin{\&} 01110101 = 00100001$
- $00101011 \mid 01110101 = 01111111$
- $00101011 \mathbin{\hat{}} 01110101 = 01011110$
- $\sim 00101011 = 11010100$

## Examples

- $00101011 \ \& \ 01110101 = 00100001$
- $00101011 \ | \ 01110101 = 01111111$
- $00101011 \ \hat{} \ 01110101 = 01011110$
- $\sim 00101011 = 11010100$

## Definition (negation)

$-\, a_k \;=\; \sim a_k + 1_k$

### Examples

- $00101011 \mathbin{\&} 01110101 = 00100001$
- $00101011 \mid 01110101 = 01111111$
- $00101011 \mathbin{\hat{}} 01110101 = 01011110$
- $\sim 00101011 = 11010100$

### Definition (negation)

$$-a_k = \sim a_k + 1_k$$

### Example

$-0101 = 1011$

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$
\begin{aligned}
\textit{formula} &::= (\neg\, \textit{formula}) \mid (\textit{formula} \wedge \textit{formula}) \mid (\textit{formula} \vee \textit{formula}) \mid \textit{atom} \\
\textit{atom} &::= \top \mid \bot \mid (\textit{term rel term}) \\
\textit{rel} &::= \; = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\textit{term} &::= (\textit{term binop term}) \mid (\textit{unop term}) \mid \textit{variable} \mid \textit{constant} \mid \textit{term}\,[i:j] \mid \\
&\phantom{::=}\ (\textit{formula} \;?\; \textit{term} : \textit{term}) \\
\textit{binop} &::= + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \;|\; \mid \hat{} \mid :: \\
\textit{unop} &::= \sim \mid -
\end{aligned}
$$

## Notation

- variables $\mathrm{a}_k$ are vectors $a_{k-1} \dots a_1 a_0$ consisting of $k$ bits
- constant $\mathrm{n}_k$ ($\mathrm{xn}_k$) is binary representation of decimal (hexadecimal) $n$ in $k$ bits

**Definition (concatenation)**

$$\mathsf{a}_k :: \mathsf{b}_m \;=\; \mathsf{c}_{k+m} \;\text{ with }\; c_i = \begin{cases} b_i & \text{if } i < m \\ a_{i-m} & \text{otherwise} \end{cases}$$

**Definition (concatenation)**

$$\mathrm{a}_k :: \mathrm{b}_m \;=\; \mathrm{c}_{k+m} \;\text{ with }\; c_i = \begin{cases} b_i & \text{if } i < m \\ a_{i-m} & \text{otherwise} \end{cases}$$

**Example**

$1101 :: 0010 = 11010010$

## Definition (concatenation)

$$\mathsf{a}_k :: \mathsf{b}_m \;=\; \mathsf{c}_{k+m} \;\text{ with }\; c_i = \begin{cases} b_i & \text{if } i < m \\ a_{i-m} & \text{otherwise} \end{cases}$$

## Example

$1101 :: 0010 = 11010010$

## Definition (extraction)

$$\mathsf{a}_k[n : m] \;=\; \mathsf{c}_{n-m+1} \;\text{ with }\; c_i = a_{i+m} \text{ for all } 0 \leqslant i \leqslant n-m \qquad \text{if } k > n \geqslant m \geqslant 0$$

## Definition (concatenation)

$$\mathsf{a}_k :: \mathsf{b}_m \;=\; \mathsf{c}_{k+m} \;\text{ with }\; c_i = \begin{cases} b_i & \text{if } i < m \\ a_{i-m} & \text{otherwise} \end{cases}$$

## Example

$1101 :: 0010 = 11010010$

## Definition (extraction)

$$\mathsf{a}_k[n:m] \;=\; \mathsf{c}_{n-m+1} \;\text{ with }\; c_i = a_{i+m} \text{ for all } 0 \leqslant i \leqslant n-m \qquad \text{if } k > n \geqslant m \geqslant 0$$

## Example

$00101011[5:2] = 1010$

**Definition (if-then-else)**

$$(\varphi \; ? \; a_k \; : \; b_k) = \begin{cases} a_k & \text{if } \varphi \text{ holds} \\ b_k & \text{otherwise} \end{cases}$$

## Definition (if-then-else)

$$(\varphi \;?\; a_k : b_k) = \begin{cases} a_k & \text{if } \varphi \text{ holds} \\ b_k & \text{otherwise} \end{cases}$$

## Examples

- $a_4 + b_4 = 3_4$

## Definition (if-then-else)

$$(\varphi \; ? \; a_k \; : \; b_k) \;=\; \begin{cases} a_k & \text{if } \varphi \text{ holds} \\ b_k & \text{otherwise} \end{cases}$$

## Examples

- $a_4 + b_4 = 3_4$

  satisfiable: $v(a_4) = 1_4$ and $v(b_4) = 2_4$

## Definition (if-then-else)

$$(\varphi \mathbin{?} a_k : b_k) = \begin{cases} a_k & \text{if } \varphi \text{ holds} \\ b_k & \text{otherwise} \end{cases}$$

## Examples

- $a_4 + b_4 = 3_4$
  satisfiable: $v(a_4) = 1_4$ and $v(b_4) = 2_4$
- $a_4 >_u a_4 + 2_4$

## Definition (if-then-else)

$$(\varphi \ ? \ a_k : b_k) = \begin{cases} a_k & \text{if } \varphi \text{ holds} \\ b_k & \text{otherwise} \end{cases}$$

## Examples

- $a_4 + b_4 = 3_4$

  satisfiable: $v(a_4) = 1_4$ and $v(b_4) = 2_4$

- $a_4 >_u a_4 + 2_4$

  satisfiable: $v(a_4) = 15_4$

## Definition (if-then-else)

$$(\varphi \;?\; a_k : b_k) = \begin{cases} a_k & \text{if } \varphi \text{ holds} \\ b_k & \text{otherwise} \end{cases}$$

## Examples

- $a_4 + b_4 = 3_4$

  satisfiable: $v(a_4) = 1_4$ and $v(b_4) = 2_4$
- $a_4 >_u a_4 + 2_4$

  satisfiable: $v(a_4) = 15_4$
- $a_4 >_s a_4 + 2_4$

## Definition (if-then-else)

$$(\varphi \mathbin{?} a_k : b_k) = \begin{cases} a_k & \text{if } \varphi \text{ holds} \\ b_k & \text{otherwise} \end{cases}$$

## Examples

- $a_4 + b_4 = 3_4$
  satisfiable: $v(a_4) = 1_4$ and $v(b_4) = 2_4$
- $a_4 >_u a_4 + 2_4$
  satisfiable: $v(a_4) = 15_4$
- $a_4 >_s a_4 + 2_4$
  satisfiable: $v(a_4) = 7_4$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$

  satisfiable: $v(a_4) = -8_4 = 8_4$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$

  satisfiable: $v(a_4) = -8_4 = 8_4$

- $a_4 \ll 2_4 = 12_4$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$

  satisfiable: $v(a_4) = -8_4 = 8_4$

- $a_4 \ll 2_4 = 12_4$

  satisfiable: $v(a_4) = 11_4$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \land \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \land \neg(a_4 = 0_4)$

  satisfiable: $v(a_4) = -8_4 = 8_4$

- $a_4 \ll 2_4 = 12_4$

  satisfiable: $v(a_4) = 11_4$

- $a_4[1:0] :: a_4[3:2] = 2_4 \land b_4[2:0] = 7_3$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$

  satisfiable: $v(a_4) = -8_4 = 8_4$

- $a_4 \ll 2_4 = 12_4$

  satisfiable: $v(a_4) = 11_4$

- $a_4[1:0] :: a_4[3:2] = 2_4 \wedge b_4[2:0] = 7_3$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 15_4$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$

  satisfiable: $v(a_4) = -8_4 = 8_4$

- $a_4 \ll 2_4 = 12_4$

  satisfiable: $v(a_4) = 11_4$

- $a_4[1:0] :: a_4[3:2] = 2_4 \wedge b_4[2:0] = 7_3$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 15_4$

- $a_8 \div_u b_8 = a_8 \gg_u 1_8$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$

  satisfiable: $v(a_4) = -8_4 = 8_4$

- $a_4 \ll 2_4 = 12_4$

  satisfiable: $v(a_4) = 11_4$

- $a_4[1:0] :: a_4[3:2] = 2_4 \wedge b_4[2:0] = 7_3$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 15_4$

- $a_8 \div_u b_8 = a_8 \gg_u 1_8$

  satisfiable: $v(a_8) = 8_8$ and $v(b_8) = 2_8$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$

  satisfiable: $v(a_4) = -8_4 = 8_4$

- $a_4 \ll 2_4 = 12_4$

  satisfiable: $v(a_4) = 11_4$

- $a_4[1:0] :: a_4[3:2] = 2_4 \wedge b_4[2:0] = 7_3$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 15_4$

- $a_8 \div_u b_8 = a_8 \gg_u 1_8$

  satisfiable: $v(a_8) = 8_8$ and $v(b_8) = 2_8$

- $a_8 \,\&\, (a_8 - 1_8) = 0_8$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$

- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$

  satisfiable: $v(a_4) = -8_4 = 8_4$

- $a_4 \ll 2_4 = 12_4$

  satisfiable: $v(a_4) = 11_4$

- $a_4[1:0] :: a_4[3:2] = 2_4 \wedge b_4[2:0] = 7_3$

  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 15_4$

- $a_8 \div_u b_8 = a_8 \gg_u 1_8$

  satisfiable: $v(a_8) = 8_8$ and $v(b_8) = 2_8$

- $a_8 \mathbin{\&} (a_8 - 1_8) = 0_8$

  satisfiable: $v(a_8) = 8_8$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \wedge \neg(a_4 \geqslant_s b_4)$
  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$
- $-a_4 = a_4 \wedge \neg(a_4 = 0_4)$
  satisfiable: $v(a_4) = -8_4 = 8_4$
- $a_4 \ll 2_4 = 12_4$
  satisfiable: $v(a_4) = 11_4$
- $a_4[1:0] :: a_4[3:2] = 2_4 \wedge b_4[2:0] = 7_3$
  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 15_4$
- $a_8 \div_u b_8 = a_8 \gg_u 1_8$
  satisfiable: $v(a_8) = 8_8$ and $v(b_8) = 2_8$
- $a_8 \,\&\, (a_8 - 1_8) = 0_8$
  satisfiable: $v(a_8) = 8_8$ or $v(a_8) = 16_8 = \mathrm{x}10_8$

## Examples (cont'd)

- $a_4 \geqslant_u b_4 \land \neg(a_4 \geqslant_s b_4)$
  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 0_4$
- $-a_4 = a_4 \land \neg(a_4 = 0_4)$
  satisfiable: $v(a_4) = -8_4 = 8_4$
- $a_4 \ll 2_4 = 12_4$
  satisfiable: $v(a_4) = 11_4$
- $a_4[1:0] :: a_4[3:2] = 2_4 \land b_4[2:0] = 7_3$
  satisfiable: $v(a_4) = 8_4$ and $v(b_4) = 15_4$
- $a_8 \div_u b_8 = a_8 \gg_u 1_8$
  satisfiable: $v(a_8) = 8_8$ and $v(b_8) = 2_8$
- $a_8 \ \& \ (a_8 - 1_8) = 0_8$
  satisfiable: $v(a_8) = 8_8$ or $v(a_8) = 16_8 = \text{x}10_8$ or $v(a_8) = \text{x}20_8$ or ...

# Outline

**Theorem**

bit-vector arithmetic is decidable if bit vectors have fixed length

## Theorem

bit-vector arithmetic is decidable if bit vectors have fixed length

## Basic Idea

use $k$ fresh propositional variables to encode bit-vector variable $\mathsf{a}_k$

## Theorem

bit-vector arithmetic is decidable if bit vectors have fixed length

## Basic Idea

use $k$ fresh propositional variables to encode bit-vector variable $a_k$

## Flattening aka Bit Blasting

input: formula $\varphi$ in bit-vector arithmetic

output: equisatisfiable propositional formula $\text{bb}(\varphi)$

## Theorem

bit-vector arithmetic is decidable if bit vectors have fixed length

## Basic Idea

use $k$ fresh propositional variables to encode bit-vector variable $\mathsf{a}_k$

## Flattening aka Bit Blasting

input: formula $\varphi$ in bit-vector arithmetic

output: equisatisfiable propositional formula $\mathrm{bb}(\varphi)$

$$\mathrm{bb}(\varphi \vee \psi) = \mathrm{bb}(\varphi) \vee \mathrm{bb}(\psi)$$

## Theorem

bit-vector arithmetic is decidable if bit vectors have fixed length

## Basic Idea

use $k$ fresh propositional variables to encode bit-vector variable $a_k$

## Flattening aka Bit Blasting

input: formula $\varphi$ in bit-vector arithmetic

output: equisatisfiable propositional formula $\mathrm{bb}(\varphi)$

$$\mathrm{bb}(\varphi \vee \psi) = \mathrm{bb}(\varphi) \vee \mathrm{bb}(\psi) \qquad \mathrm{bb}(\neg\varphi) = \neg\,\mathrm{bb}(\varphi) \qquad \mathrm{bb}(\top) = \top \qquad \mathrm{bb}(\bot) = \bot$$

$$\mathrm{bb}(\varphi \wedge \psi) = \mathrm{bb}(\varphi) \wedge \mathrm{bb}(\psi)$$

## Theorem

bit-vector arithmetic is decidable if bit vectors have fixed length

## Basic Idea

use $k$ fresh propositional variables to encode bit-vector variable $a_k$

## Flattening aka Bit Blasting

input: formula $\varphi$ in bit-vector arithmetic

output: equisatisfiable propositional formula $\mathsf{bb}(\varphi)$

$$\mathsf{bb}(\varphi \vee \psi) = \mathsf{bb}(\varphi) \vee \mathsf{bb}(\psi) \qquad \mathsf{bb}(\neg\varphi) = \neg\,\mathsf{bb}(\varphi) \qquad \mathsf{bb}(\top) = \top \qquad \mathsf{bb}(\bot) = \bot$$

$$\mathsf{bb}(\varphi \wedge \psi) = \mathsf{bb}(\varphi) \wedge \mathsf{bb}(\psi)$$

$$\mathsf{bb}(t_1 \bowtie t_2) = \mathsf{bb_a}(u_1 \bowtie u_2) \wedge \varphi_1 \wedge \varphi_2 \quad \text{if } \mathsf{bb_t}(t_1) = (u_1, \varphi_1) \text{ and } \mathsf{bb_t}(t_2) = (u_2, \varphi_2)$$

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$
\begin{aligned}
\mathit{formula} &::= (\neg\, \mathit{formula}) \mid (\mathit{formula} \wedge \mathit{formula}) \mid (\mathit{formula} \vee \mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &::= \top \mid \bot \mid (\mathit{term}\ \mathit{rel}\ \mathit{term}) \\
\mathit{rel} &::= \; = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\mathit{term} &::= (\mathit{term}\ \mathit{binop}\ \mathit{term}) \mid (\mathit{unop}\ \mathit{term}) \mid \mathit{variable} \mid \mathit{constant} \mid \mathit{term}[i:j] \mid \\
&\phantom{::=}\ (\mathit{formula}\ ?\ \mathit{term} : \mathit{term}) \\
\mathit{binop} &::= + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \;|\; \mid \hat{} \mid :: \\
\mathit{unop} &::= \sim \mid -
\end{aligned}
$$

## Notation

- variables $\mathrm{a}_k$ are vectors $a_{k-1} \dots a_1 a_0$ consisting of $k$ bits
- constant $\mathrm{n}_k$ ($\mathrm{xn}_k$) is binary representation of decimal (hexadecimal) $n$ in $k$ bits

# Flattening aka Bit Blasting (cont'd)

equality

$$\mathsf{bb_a}(\mathsf{a}_k = \mathsf{b}_k) \;=\; (a_{k-1} \leftrightarrow b_{k-1}) \land \cdots \land (a_1 \leftrightarrow b_1) \land (a_0 \leftrightarrow b_0)$$

# Flattening aka Bit Blasting (cont'd)

equality

$$\mathrm{bb_a}(\mathrm{a}_k = \mathrm{b}_k) = (a_{k-1} \leftrightarrow b_{k-1}) \wedge \cdots \wedge (a_1 \leftrightarrow b_1) \wedge (a_0 \leftrightarrow b_0)$$

inequality

$$\mathrm{bb_a}(\mathrm{a}_k \neq \mathrm{b}_k) = (a_{k-1} \oplus b_{k-1}) \vee \cdots \vee (a_1 \oplus b_1) \vee (a_0 \oplus b_0)$$

# Flattening aka Bit Blasting (cont'd)

equality

$$\mathsf{bb_a}(\mathsf{a}_k = \mathsf{b}_k) \;=\; (a_{k-1} \leftrightarrow b_{k-1}) \wedge \cdots \wedge (a_1 \leftrightarrow b_1) \wedge (a_0 \leftrightarrow b_0)$$

inequality

$$\mathsf{bb_a}(\mathsf{a}_k \neq \mathsf{b}_k) \;=\; (a_{k-1} \oplus b_{k-1}) \vee \cdots \vee (a_1 \oplus b_1) \vee (a_0 \oplus b_0)$$

unsigned greater-than-or-equal

$$\mathsf{bb_a}(\mathsf{a}_1 \geqslant_u \mathsf{b}_1) \;=\; b_0 \to a_0$$

$$\mathsf{bb_a}(\mathsf{a}_{k+1} \geqslant_u \mathsf{b}_{k+1}) \;=\; a_k \wedge \neg b_k \vee (a_k \leftrightarrow b_k) \wedge \mathsf{bb_a}(\mathsf{a}[k-1:0] \geqslant_u \mathsf{b}[k-1:0])$$

# Flattening aka Bit Blasting (cont'd)

equality

$$\mathrm{bb_a}(\mathsf{a}_k = \mathsf{b}_k) \;=\; (a_{k-1} \leftrightarrow b_{k-1}) \wedge \cdots \wedge (a_1 \leftrightarrow b_1) \wedge (a_0 \leftrightarrow b_0)$$

inequality

$$\mathrm{bb_a}(\mathsf{a}_k \neq \mathsf{b}_k) \;=\; (a_{k-1} \oplus b_{k-1}) \vee \cdots \vee (a_1 \oplus b_1) \vee (a_0 \oplus b_0)$$

unsigned greater-than-or-equal

$$\mathrm{bb_a}(\mathsf{a}_1 \geqslant_u \mathsf{b}_1) \;=\; b_0 \rightarrow a_0$$

$$\mathrm{bb_a}(\mathsf{a}_{k+1} \geqslant_u \mathsf{b}_{k+1}) \;=\; a_k \wedge \neg b_k \vee (a_k \leftrightarrow b_k) \wedge \mathrm{bb_a}(\mathsf{a}[k-1:0] \geqslant_u \mathsf{b}[k-1:0])$$

unsigned greater-than

$$\mathrm{bb_a}(\mathsf{a}_k >_u \mathsf{b}_k) \;=\; \mathrm{bb_a}(\mathsf{a}_k \geqslant_u \mathsf{b}_k) \wedge \mathrm{bb_a}(\mathsf{a}_k \neq \mathsf{b}_k)$$

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$
\begin{aligned}
\textit{formula} &::= (\neg\, \textit{formula}) \mid (\textit{formula} \wedge \textit{formula}) \mid (\textit{formula} \vee \textit{formula}) \mid \textit{atom} \\
\textit{atom} &::= \top \mid \bot \mid (\textit{term rel term}) \\
\textit{rel} &::= \; = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\textit{term} &::= (\textit{term binop term}) \mid (\textit{unop term}) \mid \textit{variable} \mid \textit{constant} \mid \textit{term}[i:j] \mid \\
&\phantom{::=}\ (\textit{formula} \ ?\ \textit{term} : \textit{term}) \\
\textit{binop} &::= + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \,|\, \mid \hat{} \mid :: \\
\textit{unop} &::= \sim \mid -
\end{aligned}
$$

## Notation

- variables $a_k$ are vectors $a_{k-1} \dots a_1 a_0$ consisting of $k$ bits
- constant $n_k$ ($xn_k$) is binary representation of decimal (hexadecimal) $n$ in $k$ bits

# Flattening aka Bit Blasting (cont'd)

bitwise and

$$\mathsf{bb_t}(\mathsf{a}_k \mathbin{\&} \mathsf{b}_k) = (\mathsf{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow a_i \wedge b_i)$$

# Flattening aka Bit Blasting (cont'd)

bitwise and

$$\mathsf{bb_t}(\mathsf{a}_k \mathbin{\&} \mathsf{b}_k) = (\mathsf{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow a_i \wedge b_i)$$

bitwise or

$$\mathsf{bb_t}(\mathsf{a}_k \mid \mathsf{b}_k) = (\mathsf{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow a_i \vee b_i)$$

# Flattening aka Bit Blasting (cont'd)

bitwise and

$$\mathrm{bb_t}(\mathrm{a}_k \,\&\, \mathrm{b}_k) = (\mathrm{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow a_i \wedge b_i)$$

bitwise or

$$\mathrm{bb_t}(\mathrm{a}_k \mid \mathrm{b}_k) = (\mathrm{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow a_i \vee b_i)$$

bitwise xor

$$\mathrm{bb_t}(\mathrm{a}_k \,\hat{}\, \mathrm{b}_k) = (\mathrm{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow a_i \oplus b_i)$$

## Flattening aka Bit Blasting (cont'd)

bitwise and

$$\mathrm{bb_t}(\mathrm{a}_k \;\&\; \mathrm{b}_k) = (\mathrm{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow a_i \wedge b_i)$$

bitwise or

$$\mathrm{bb_t}(\mathrm{a}_k \mid \mathrm{b}_k) = (\mathrm{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow a_i \vee b_i)$$

bitwise xor

$$\mathrm{bb_t}(\mathrm{a}_k \,\hat{}\, \mathrm{b}_k) = (\mathrm{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow a_i \oplus b_i)$$

bitwise negation

$$\mathrm{bb_t}(\sim \mathrm{a}_k) = (\mathrm{c}_k, \varphi) \quad \text{with} \quad \varphi = \bigwedge_{i=0}^{k-1} (c_i \leftrightarrow \neg a_i)$$

## Definition (Bit-Vector Arithmetic)

formulas are built according to following BNF grammar:

$$
\begin{aligned}
\mathit{formula} &::= (\neg\, \mathit{formula}) \mid (\mathit{formula} \wedge \mathit{formula}) \mid (\mathit{formula} \vee \mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &::= \top \mid \bot \mid (\mathit{term}\ \mathit{rel}\ \mathit{term}) \\
\mathit{rel} &::= \ = \mid \neq \mid \geqslant_u \mid \geqslant_s \mid >_u \mid >_s \\
\mathit{term} &::= (\mathit{term}\ \mathit{binop}\ \mathit{term}) \mid (\mathit{unop}\ \mathit{term}) \mid \mathit{variable} \mid \mathit{constant} \mid \mathit{term}[i:j] \mid \\
&\qquad (\mathit{formula}\ ?\ \mathit{term} : \mathit{term}) \\
\mathit{binop} &::= + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \ | \ \mid \hat{} \mid :: \\
\mathit{unop} &::= \sim \mid -
\end{aligned}
$$

## Notation

- variables a$_k$ are vectors $a_{k-1} \dots a_1 a_0$ consisting of $k$ bits
- constant n$_k$ (xn$_k$) is binary representation of decimal (hexadecimal) $n$ in $k$ bits

# Flattening aka Bit Blasting (cont'd)

concatenation

$$\mathrm{bb}_{\mathrm{t}}(\mathsf{a}_k :: \mathsf{b}_m) = (\mathsf{a}_k \mathsf{b}_m, \top)$$

# Flattening aka Bit Blasting (cont'd)

concatenation

$$\mathrm{bb_t}(\mathrm{a}_k :: \mathrm{b}_m) = (\mathrm{a}_k \mathrm{b}_m, \top)$$

extraction

$$\mathrm{bb_t}(\mathrm{a}_k[n:m]) = (a_n \dots a_m, \top)$$

# Flattening aka Bit Blasting (cont'd)

concatenation

$$\mathsf{bb_t}(\mathsf{a}_k :: \mathsf{b}_m) = (\mathsf{a}_k\mathsf{b}_m, \top)$$

extraction

$$\mathsf{bb_t}(\mathsf{a}_k[n:m]) = (a_n \dots a_m, \top)$$

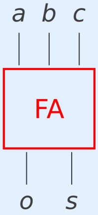
if-then-else

$$\mathsf{bb_t}(\psi \mathrel{?} \mathsf{a}_k : \mathsf{b}_k) = (\mathsf{c}_k, \varphi)$$

$$\text{with} \quad \varphi = (f \leftrightarrow \mathsf{bb}(\psi)) \wedge \bigwedge_{i=0}^{k-1} \big((f \to (c_i \leftrightarrow a_i)) \wedge (\neg f \to (c_i \leftrightarrow b_i))\big)$$

## Remark

addition is flattened using ripple carry adder

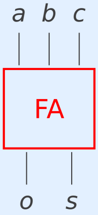



full adder

$$s = a + b + c \mod 2$$
$$o = a + b + c \;\text{div}\; 2$$

## Remark

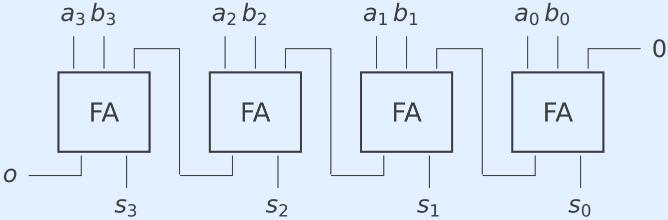
addition is flattened using ripple carry adder




full adder

$$s = a + b + c \mod 2$$
$$o = a + b + c \operatorname{div} 2$$

4 bit ripple carry adder



# Flattening aka Bit Blasting (cont'd)

addition

$$\mathsf{bb_t}(\mathsf{a}_k + \mathsf{b}_k) = (\mathsf{s}_k, \varphi) \quad \text{with}$$

$$\varphi = (s_0 \leftrightarrow a_0 \oplus b_0) \wedge (c_0 \leftrightarrow a_0 \wedge b_0) \wedge \bigwedge_{i=1}^{k-1} \big((s_i \leftrightarrow a_i \oplus b_i \oplus c_{i-1}) \wedge (c_i \leftrightarrow a_i \wedge b_i \vee (a_i \oplus b_i) \wedge c_{i-1})\big)$$

## Flattening aka Bit Blasting (cont'd)

addition

$$\mathrm{bb_t}(\mathrm{a}_k + \mathrm{b}_k) = (\mathrm{s}_k, \varphi) \quad \text{with}$$

$$\varphi = (s_0 \leftrightarrow a_0 \oplus b_0) \wedge (c_0 \leftrightarrow a_0 \wedge b_0) \wedge \bigwedge_{i=1}^{k-1} \big((s_i \leftrightarrow a_i \oplus b_i \oplus c_{i-1}) \wedge (c_i \leftrightarrow a_i \wedge b_i \vee (a_i \oplus b_i) \wedge c_{i-1})\big)$$

unary minus

$$\mathrm{bb_t}(-\mathrm{a}_k) = \mathrm{bb_t}(\sim \mathrm{a}_k + 1_k)$$

# Flattening aka Bit Blasting (cont'd)

addition

$$\mathrm{bb}_t(\mathrm{a}_k + \mathrm{b}_k) = (\mathrm{s}_k, \varphi) \quad \text{with}$$

$$\varphi = (s_0 \leftrightarrow a_0 \oplus b_0) \wedge (c_0 \leftrightarrow a_0 \wedge b_0) \wedge \bigwedge_{i=1}^{k-1} \left((s_i \leftrightarrow a_i \oplus b_i \oplus c_{i-1}) \wedge (c_i \leftrightarrow a_i \wedge b_i \vee (a_i \oplus b_i) \wedge c_{i-1})\right)$$

unary minus

$$\mathrm{bb}_t(-\mathrm{a}_k) = \mathrm{bb}_t({\sim}\, \mathrm{a}_k + 1_k)$$

subtraction

$$\mathrm{bb}_t(\mathrm{a}_k - \mathrm{b}_k) = \mathrm{bb}_t(\mathrm{a}_k + (-\mathrm{b}_k))$$

## Flattening aka Bit Blasting (cont'd)

multiplication

$$\mathsf{bb_t}(\mathsf{a}_k \times \mathsf{b}_k) = \mathsf{bb_t}(\mathsf{mul}(\mathsf{a}_k, \mathsf{b}_k, 0)) \quad \text{with}$$

$$\mathsf{mul}(\mathsf{a}_k, \mathsf{b}_k, i) = \begin{cases} 0_k & \text{if } i = k \\ \mathsf{mul}(\mathsf{a}_k \ll 1_k, \mathsf{b}_k, i+1) + (b_i \mathbin{?} \mathsf{a}_k : 0_k) & \text{if } i < k \end{cases}$$

# Flattening aka Bit Blasting (cont'd)

multiplication

$$\mathsf{bb}_t(\mathsf{a}_k \times \mathsf{b}_k) = \mathsf{bb}_t(\mathsf{mul}(\mathsf{a}_k, \mathsf{b}_k, 0)) \quad \text{with}$$

$$\mathsf{mul}(\mathsf{a}_k, \mathsf{b}_k, i) = \begin{cases} 0_k & \text{if } i = k \\ \mathsf{mul}(\mathsf{a}_k \ll 1_k, \mathsf{b}_k, i+1) + (b_i \; ? \; \mathsf{a}_k : 0_k) & \text{if } i < k \end{cases}$$

unsigned division

$$\mathsf{bb}_t(\mathsf{a}_k \div_u \mathsf{b}_k) = (\mathsf{q}_k, \varphi) \quad \text{with}$$

$$\varphi = \mathsf{bb}(\mathsf{b}_k \neq 0_k \to (\mathsf{q}_k \times \mathsf{b}_k + \mathsf{r}_k = \mathsf{a}_k \wedge \mathsf{b}_k >_u \mathsf{r}_k \wedge \mathsf{a}_k \geqslant_u \mathsf{q}_k))$$

# SMT-LIB 2 Format for BV

bit-vector formula $\mathsf{b}_4 >_u \mathsf{a}_4 + \mathsf{b}_4 \wedge \mathsf{a}_4 \neq 10_4 \wedge \mathsf{a}_4 \,\&\, \mathsf{b}_4 = 8_4$

## SMT-LIB 2 Format for BV

bit-vector formula $b_4 >_u a_4 + b_4 \wedge a_4 \neq 10_4 \wedge a_4 \,\&\, b_4 = 8_4$

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvugt b (bvadd a b)))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

# SMT-LIB 2 Format for BV

bit-vector formula $b_4 >_u a_4 + b_4 \wedge a_4 \neq 10_4 \wedge a_4 \,\&\, b_4 = 8_4$

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvugt b (bvadd a b)))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

- `(_ BitVec k)` is sort of bit vectors of width $k$

# SMT-LIB 2 Format for BV

bit-vector formula $b_4 >_u a_4 + b_4 \wedge a_4 \neq 10_4 \wedge a_4 \mathbin{\&} b_4 = 8_4$

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvugt b (bvadd a b)))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

- `(_ BitVec k)` is sort of bit vectors of width *k*
- `#xa` is constant in hexidecimal notation
- `#b1000` is constant in binary notation

## SMT-LIB 2 Format for BV

bit-vector formula $b_4 >_u a_4 + b_4 \wedge a_4 \neq 10_4 \wedge a_4 \mathbin{\&} b_4 = 8_4$

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvugt b (bvadd a b)))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

- `(_ BitVec k)` is sort of bit vectors of width *k*
- `#xa` is constant in hexidecimal notation
- `#b1000` is constant in binary notation
- `bvadd`, `bvsub`, `bvmul` are arithmetic operators

## SMT-LIB 2 Format for BV

bit-vector formula $b_4 >_u a_4 + b_4 \wedge a_4 \neq 10_4 \wedge a_4 \,\&\, b_4 = 8_4$

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvugt b (bvadd a b)))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

- `(_ BitVec k)` is sort of bit vectors of width $k$
- `#xa` is constant in hexidecimal notation
- `#b1000` is constant in binary notation
- `bvadd`, `bvsub`, `bvmul` are arithmetic operators
- `bvudiv` and `bvsdiv` are unsigned/signed division

# SMT-LIB 2 Format for BV

bit-vector formula $b_4 >_u a_4 + b_4 \wedge a_4 \neq 10_4 \wedge a_4 \,\&\, b_4 = 8_4$

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvugt b (bvadd a b)))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

- `(_ BitVec k)` is sort of bit vectors of width $k$
- `#xa` is constant in hexidecimal notation
- `#b1000` is constant in binary notation
- `bvadd`, `bvsub`, `bvmul` are arithmetic operators
- `bvudiv` and `bvsdiv` are unsigned/signed division
- `bvugt` and `bvsgt` are unsigned/signed greater-than

# SMT-LIB 2 Format for BV

bit-vector formula $b_4 >_u a_4 + b_4 \wedge a_4 \neq 10_4 \wedge a_4 \mathbin{\&} b_4 = 8_4$

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvugt b (bvadd a b)))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

- `(_ BitVec k)` is sort of bit vectors of width *k*
- `#xa` is constant in hexidecimal notation
- `#b1000` is constant in binary notation
- `bvadd`, `bvsub`, `bvmul` are arithmetic operators
- `bvudiv` and `bvsdiv` are unsigned/signed division
- `bvugt` and `bvsgt` are unsigned/signed greater-than
- `bvshl`, `bvlshr`, `bvashr` are shift operators

# SMT-LIB 2 Format for BV

bit-vector formula $b_4 >_u a_4 + b_4 \wedge a_4 \neq 10_4 \wedge a_4 \,\&\, b_4 = 8_4$

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvugt b (bvadd a b)))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

- `(_ BitVec k)` is sort of bit vectors of width $k$
- `#xa` is constant in hexidecimal notation
- `#b1000` is constant in binary notation
- `bvadd`, `bvsub`, `bvmul` are arithmetic operators
- `bvudiv` and `bvsdiv` are unsigned/signed division
- `bvugt` and `bvsgt` are unsigned/signed greater-than
- `bvshl`, `bvlshr`, `bvashr` are shift operators
- `bvand`, `bvor`, `bvnot` are bitwise logical operators

## SMT-LIB 2 Format for BV

bit-vector formula $b_4 >_u a_4 + b_4 \wedge a_4 \neq 10_4 \wedge a_4 \,\&\, b_4 = 8_4$

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvugt b (bvadd a b)))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

- `(_ BitVec k)` is sort of bit vectors of width *k*
- `#xa` is constant in hexidecimal notation
- `#b1000` is constant in binary notation
- `bvadd`, `bvsub`, `bvmul` are arithmetic operators
- `bvudiv` and `bvsdiv` are unsigned/signed division
- `bvugt` and `bvsgt` are unsigned/signed greater-than
- `bvshl`, `bvlshr`, `bvashr` are shift operators
- `bvand`, `bvor`, `bvnot` are bitwise logical operators
- ...

# Outline

## Remark

bit-vector formulas with multiplication (division, remainder) are very hard to solve due to decision heuristics of state-of-the-art SAT solvers

## Remark

bit-vector formulas with multiplication (division, remainder) are very hard to solve due to decision heuristics of state-of-the-art SAT solvers

## Example

bit-vector formula $\varphi$

$$\mathsf{a}_k \times \mathsf{b}_k = \mathsf{c}_k \;\wedge\; \mathsf{b}_k \times \mathsf{a}_k \neq \mathsf{c}_k \;\wedge\; \mathsf{d}_k >_u \mathsf{e}_k \;\wedge\; \mathsf{e}_k >_u \mathsf{d}_k$$

## Remark

bit-vector formulas with multiplication (division, remainder) are very hard to solve due to decision heuristics of state-of-the-art SAT solvers

## Example

bit-vector formula $\varphi$

$$\mathsf{a}_k \times \mathsf{b}_k = \mathsf{c}_k \;\wedge\; \mathsf{b}_k \times \mathsf{a}_k \neq \mathsf{c}_k \;\wedge\; \mathsf{d}_k >_u \mathsf{e}_k \;\wedge\; \mathsf{e}_k >_u \mathsf{d}_k$$

results in propositional CNF formula with about 11 000 variables for $k = 32$

## Remark

bit-vector formulas with multiplication (division, remainder) are very hard to solve due to decision heuristics of state-of-the-art SAT solvers

## Example

bit-vector formula $\varphi$

$$\mathsf{a}_k \times \mathsf{b}_k = \mathsf{c}_k \;\wedge\; \mathsf{b}_k \times \mathsf{a}_k \neq \mathsf{c}_k \;\wedge\; \mathsf{d}_k >_u \mathsf{e}_k \;\wedge\; \mathsf{e}_k >_u \mathsf{d}_k$$

results in propositional CNF formula with about 11000 variables for $k = 32$

- $\varphi$ is unsatisfiable

## Remark

bit-vector formulas with multiplication (division, remainder) are very hard to solve due to decision heuristics of state-of-the-art SAT solvers

## Example

bit-vector formula $\varphi$

$$a_k \times b_k = c_k \;\wedge\; b_k \times a_k \neq c_k \;\wedge\; d_k >_u e_k \;\wedge\; e_k >_u d_k$$

results in propositional CNF formula with about 11000 variables for $k = 32$

- $\varphi$ is unsatisfiable

**Remark**

bit-vector formulas with multiplication (division, remainder) are very hard to solve due to decision heuristics of state-of-the-art SAT solvers

**Example**

bit-vector formula $\varphi$

$$\mathsf{a}_k \times \mathsf{b}_k = \mathsf{c}_k \;\wedge\; \mathsf{b}_k \times \mathsf{a}_k \neq \mathsf{c}_k \;\wedge\; \mathsf{d}_k >_u \mathsf{e}_k \;\wedge\; \mathsf{e}_k >_u \mathsf{d}_k$$

results in propositional CNF formula with about 11000 variables for $k = 32$

- $\varphi$ is unsatisfiable
- most SAT solvers favor decisions on $a$, $b$, $c$ and thus aim to show unsatisfiability of part with multiplications

# Solutions

1. incremental flattening

## Solutions

1. incremental flattening

   - omit $\mathsf{bb}_t(\,\cdot\,)$ constraints for expensive operators from bit-vector formula $\varphi$

## Solutions

1. incremental flattening
   - omit $\mathrm{bb}_t(\,\cdot\,)$ constraints for expensive operators from bit-vector formula $\varphi$
   - if SAT solver returns unsatisfiable then $\varphi$ is unsatisfiable

## Solutions

1. incremental flattening
   - omit $\mathrm{bb}_t(\,\cdot\,)$ constraints for expensive operators from bit-vector formula $\varphi$
   - if SAT solver returns unsatisfiable then $\varphi$ is unsatisfiable
   - if SAT solver returns satisfiable then check whether obtained model satisfies omitted constraints

## Solutions

1. incremental flattening
   - omit $\mathrm{bb}_t(\,\cdot\,)$ constraints for expensive operators from bit-vector formula $\varphi$
   - if SAT solver returns unsatisfiable then $\varphi$ is unsatisfiable
   - if SAT solver returns satisfiable then check whether obtained model satisfies omitted constraints (after extending assignment)

# Solutions

1. incremental flattening
   - omit $\mathrm{bb}_t(\,\cdot\,)$ constraints for expensive operators from bit-vector formula $\varphi$
   - if SAT solver returns unsatisfiable then $\varphi$ is unsatisfiable
   - if SAT solver returns satisfiable then check whether obtained model satisfies omitted constraints (after extending assignment)

     yes $\Longrightarrow$ $\varphi$ is satisfiable

# Solutions

1. incremental flattening

- omit $\mathrm{bb}_t(\,\cdot\,)$ constraints for expensive operators from bit-vector formula $\varphi$
- if SAT solver returns unsatisfiable then $\varphi$ is unsatisfiable
- if SAT solver returns satisfiable then check whether obtained model satisfies omitted constraints (after extending assignment)

  yes $\Longrightarrow$ $\varphi$ is satisfiable

  no $\Longrightarrow$ add constraints for some violated $\mathrm{bb}_t(\,\cdot\,)$ part and repeat

## Solutions

1. incremental flattening
   - omit $\mathrm{bb}_t(\,\cdot\,)$ constraints for expensive operators from bit-vector formula $\varphi$
   - if SAT solver returns unsatisfiable then $\varphi$ is unsatisfiable
   - if SAT solver returns satisfiable then check whether obtained model satisfies omitted constraints (after extending assignment)

     yes $\Longrightarrow$ $\varphi$ is satisfiable

     no $\Longrightarrow$ add constraints for some violated $\mathrm{bb}_t(\,\cdot\,)$ part and repeat

2. use uninterpreted functions to abstract from expensive operators

# Outline

## Remarks

- many applications require arithmetic on rational numbers

## Remarks

- many applications require arithmetic on rational numbers
- high-end microprocessors offer support for floating-point arithmetic

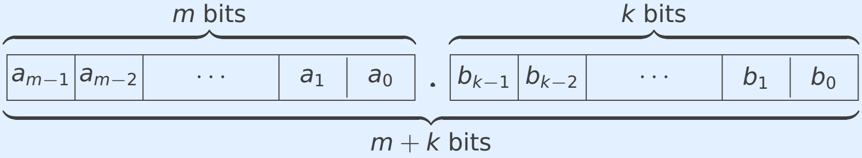
## Remarks

- many applications require arithmetic on rational numbers
- high-end microprocessors offer support for floating-point arithmetic
- fixed-point arithmetic is reasonable compromise between accuracy and complexity of floating-point arithmetic, e.g., used for controllers in vehicles or for currencies in financial software

## Remarks

- many applications require arithmetic on rational numbers
- high-end microprocessors offer support for floating-point arithmetic
- fixed-point arithmetic is reasonable compromise between accuracy and complexity of floating-point arithmetic, e.g., used for controllers in vehicles or for currencies in financial software

## Representation



## Remarks

- many applications require arithmetic on rational numbers
- high-end microprocessors offer support for floating-point arithmetic
- fixed-point arithmetic is reasonable compromise between accuracy and complexity of floating-point arithmetic, e.g., used for controllers in vehicles or for currencies in financial software

## Representation

$m$ bits (integer part), $k$ bits (fractional part), $m + k$ bits in total:

| $m$ bits | | | | | | $k$ bits | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $a_{m-1}$ | $a_{m-2}$ | $\cdots$ | $a_1$ | $a_0$ | . | $b_{k-1}$ | $b_{k-2}$ | $\cdots$ | $b_1$ | $b_0$ |

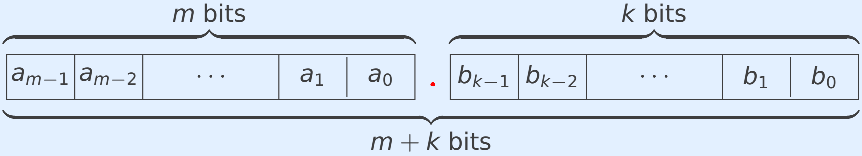
$m + k$ bits

- $a_{m-1}a_{m-2}\dots a_1a_0$ is integer part (magnitude)

## Remarks

- many applications require arithmetic on rational numbers
- high-end microprocessors offer support for floating-point arithmetic
- fixed-point arithmetic is reasonable compromise between accuracy and complexity of floating-point arithmetic, e.g., used for controllers in vehicles or for currencies in financial software

## Representation

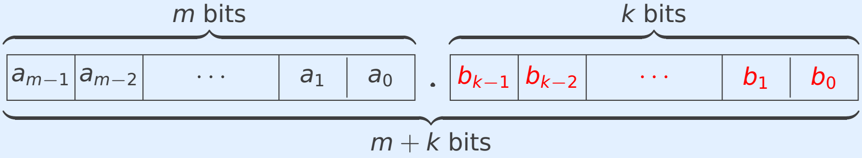



- $a_{m-1}a_{m-2} \dots a_1a_0$ is integer part (magnitude)
- . is radix point

## Remarks

- many applications require arithmetic on rational numbers
- high-end microprocessors offer support for floating-point arithmetic
- fixed-point arithmetic is reasonable compromise between accuracy and complexity of floating-point arithmetic, e.g., used for controllers in vehicles or for currencies in financial software

## Representation




- $a_{m-1}a_{m-2}\ldots a_1a_0$ is integer part (magnitude)
- . is radix point
- $b_{k-1}b_{k-2}\ldots b_1b_0$ is fractional part

## Definition

rational encoding $\langle a_{m-1} \dots a_0 \,.\, b_{k-1} \dots b_0 \rangle_r = \dfrac{\langle a_{m-1} \dots a_0 \, b_{k-1} \dots b_0 \rangle_s}{2^k}$

## Definition

rational encoding $\langle a_{m-1} \dots a_0 \,.\, b_{k-1} \dots b_0 \rangle_r = \dfrac{\langle a_{m-1} \dots a_0 \, b_{k-1} \dots b_0 \rangle_s}{2^k}$

## Examples

- $\langle 0.10 \rangle_r = 0.5$

## Definition

rational encoding $\langle a_{m-1} \dots a_0 \,.\, b_{k-1} \dots b_0 \rangle_r = \dfrac{\langle a_{m-1} \dots a_0 \, b_{k-1} \dots b_0 \rangle_s}{2^k}$

## Examples

- $\langle 0.10 \rangle_r = 0.5$
- $\langle 0.01 \rangle_r = 0.25$

## Definition

rational encoding $\langle a_{m-1} \dots a_0 \,.\, b_{k-1} \dots b_0 \rangle_r = \dfrac{\langle a_{m-1} \dots a_0 \, b_{k-1} \dots b_0 \rangle_s}{2^k}$

## Examples

- $\langle 0.10 \rangle_r = 0.5$
- $\langle 0.01 \rangle_r = 0.25$
- $\langle 01.1 \rangle_r = 1.5$

## Definition

rational encoding $\langle a_{m-1} \dots a_0 \,.\, b_{k-1} \dots b_0 \rangle_r = \dfrac{\langle a_{m-1} \dots a_0 \, b_{k-1} \dots b_0 \rangle_s}{2^k}$

## Examples

- $\langle 0.10 \rangle_r = 0.5$
- $\langle 0.01 \rangle_r = 0.25$
- $\langle 01.1 \rangle_r = 1.5$
- $\langle 11111111.1 \rangle_r = -0.5$

## Definition

rational encoding $\langle a_{m-1} \dots a_0 \,.\, b_{k-1} \dots b_0 \rangle_r = \dfrac{\langle a_{m-1} \dots a_0\, b_{k-1} \dots b_0 \rangle_s}{2^k}$

## Examples

- $\langle 0.10 \rangle_r = 0.5$
- $\langle 0.01 \rangle_r = 0.25$
- $\langle 01.1 \rangle_r = 1.5$
- $\langle 11111111.1 \rangle_r = -0.5$
- $\langle 0000.0101 \rangle_r = 0.3125 < \frac{1}{3} < 0.375 = \langle 0000.0110 \rangle_r$

## Definition

rational encoding $\langle a_{m-1} \dots a_0 \,.\, b_{k-1} \dots b_0 \rangle_r \;=\; \dfrac{\langle a_{m-1} \dots a_0 \, b_{k-1} \dots b_0 \rangle_s}{2^k}$

## Examples

- $\langle 0.10 \rangle_r = 0.5$
- $\langle 0.01 \rangle_r = 0.25$
- $\langle 01.1 \rangle_r = 1.5$
- $\langle 11111111.1 \rangle_r = -0.5$
- $\langle 0000.0101 \rangle_r = 0.3125 < \frac{1}{3} < 0.375 = \langle 0000.0110 \rangle_r$

## Definition (addition)

$\mathsf{a}_m \,.\, \mathsf{b}_k + \mathsf{c}_m \,.\, \mathsf{d}_k \;=\; \mathsf{e}_m \,.\, \mathsf{f}_k \quad\Longleftrightarrow\quad \langle \mathsf{a}_m \,.\, \mathsf{b}_k \rangle_r \cdot 2^k + \langle \mathsf{c}_m \,.\, \mathsf{d}_k \rangle_r \cdot 2^k = \langle \mathsf{e}_m \,.\, \mathsf{f}_k \rangle_r \cdot 2^k \mod 2^{m+k}$

## Example

$\langle 00.1 \rangle_r + \langle 00.1 \rangle_r = \langle 01.0 \rangle_r$

## Example

$\langle 00.1 \rangle_r + \langle 00.1 \rangle_r = \langle 01.0 \rangle_r \qquad \langle 000.0 \rangle_r + \langle 1.0 \rangle_r = \langle 111.0 \rangle_r$

## Example

$\langle 00.1 \rangle_r + \langle 00.1 \rangle_r = \langle 01.0 \rangle_r \qquad \langle 000.0 \rangle_r + \langle 1.0 \rangle_r = \langle 111.0 \rangle_r$

## Remark

sign-extension is needed to ensure that operands have same number of bits

**Example**

$\langle 00.1\rangle_r + \langle 00.1\rangle_r = \langle 01.0\rangle_r$ $\langle 000.0\rangle_r + \langle 1.0\rangle_r = \langle 111.0\rangle_r$

**Remark**

sign-extension is needed to ensure that operands have same number of bits

**Example**

$\langle 0.1\rangle_r \cdot \langle 1.1\rangle_r = \langle 1.11\rangle_r$

**Example**

$\langle 00.1 \rangle_r + \langle 00.1 \rangle_r = \langle 01.0 \rangle_r \qquad \langle 000.0 \rangle_r + \langle 1.0 \rangle_r = \langle 111.0 \rangle_r$

**Remark**

sign-extension is needed to ensure that operands have same number of bits

**Example**

$\langle 0.1 \rangle_r \cdot \langle 1.1 \rangle_r = \langle 1.11 \rangle_r \qquad \langle 1.1 \rangle_r \cdot \langle 1.1 \rangle_r = \langle 0.01 \rangle_r$

**Example**

$\langle 00.1 \rangle_r + \langle 00.1 \rangle_r = \langle 01.0 \rangle_r \qquad \langle 000.0 \rangle_r + \langle 1.0 \rangle_r = \langle 111.0 \rangle_r$

**Remark**

sign-extension is needed to ensure that operands have same number of bits

**Example**

$\langle 0.1 \rangle_r \cdot \langle 1.1 \rangle_r = \langle 1.11 \rangle_r \qquad \langle 1.1 \rangle_r \cdot \langle 1.1 \rangle_r = \langle 0.01 \rangle_r$

**Remark**

rounding steps may be needed to get rid of extra bits in fractional part after multiplication

# Outline

## Kröning and Strichmann

- Chapter 6
- Section A.2.3

## Kröning and Strichmann

- Chapter 6
- Section A.2.3

## Important Concepts

- arithmetic right shift
- bit vector
- bit-vector arithmetic
- bitwise operator
- fixed-point arithmetic
- flattening
- left shift
- logical right shift
- rational encoding $\langle \cdot . \cdot \rangle_r$
- signed encoding $\langle \cdot \rangle_s$
- two's complement
- unsigned encoding $\langle \cdot \rangle_u$