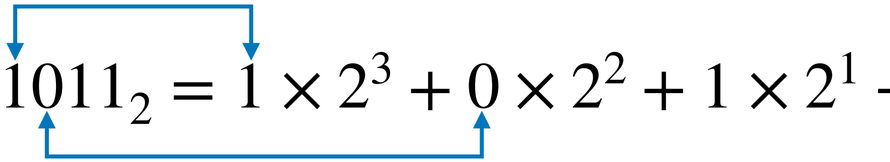
Bit-Vector Arithmetic

Ming-Hsien Tsai, FLOLAC 2025

Based on Daniel Kroening and Ofer Strichman: Decision Procedures - An Algorithmic Point of View

Bit-Vector (BV) Arithmetic

- A computer system uses bit-vectors to encode information, for example, numbers
  - 32-bit unsigned integers, 64-bit signed integers, etc.
  - The integer values represented by bit-vectors:
    - 4-bit unsigned integer $1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$
    - 4-bit signed integer $1011_2 = 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 - 1 \times 2^3 = -5$



- The semantics of computer operations such as addition no longer matches what we are used to when reasoning about unbounded types, for example, the natural numbers

Bit-Accurate Programs

A cryptographic program computing
$r \equiv a \times b \pmod{p_{256}}$

```
...
xor     %r9,%r9
mulx    0x80(%rsi),%rcx,%rbp
adcx    %rcx,%r10
adox    %rbp,%r11
mulx    0x88(%rsi),%rcx,%rbp
adcx    %rcx,%r11
adox    %rbp,%r12
mulx    0x90(%rsi),%rcx,%rbp
adcx    %rcx,%r12
adox    %rbp,%r13
...
```

The 256-bit integer a is represented by 4 64-bit registers a_0, a_1, a_2, a_3, i.e., $a = a_3 2^{192} + a_2 2^{128} + a_1 2^{64} + a_0$

The computation of $a \times b$ must be performed by multiplying $a_i's$ and $b_j's$ separately

Carries has to be processed carefully

Cryptographers tend to omit carries that must be zero due to performance consideration

The may make mistakes in estimating carries

Unsigned and Signed Interpretations

MSB: Most Significant Bit

LSB: Least Significant Bit

- Consider the n-bit bit-vector $(b_{n-1}\cdots b_2 b_1 b_0)_2$
  - Unsigned (binary encoding)

MSB → $b_{n-1}2^{n-1} + \cdots + b_2x^2 + b_1x^1 + b_0x^0$ ← LSB

  - Signed (two's complement interpretation)

sign bit → $b_{n-2}x^{n-2} + \cdots + b_2x^2 + b_1x^1 + b_0x^0 - b_{n-1}2^{n-1}$

  - invert bits, plus 1, interpret as unsigned, negate the result

$$(1011)_2 \overset{inv}{\rightarrow} (0100)_2 \overset{+1}{\rightarrow} (0101)_2 \overset{unsigned}{\rightarrow} 5 \overset{negate}{\rightarrow} -5$$

Simple Bit-Vector Operations

	BV	Unsigned	Signed
	1011	11	−5
+	0101	5	5
	0000	0	0

	BV	Unsigned	Signed
	0101	5	5
−	1011	11	−5
	1010	10	−6

	BV	Unsigned	Signed
	1011	11	−5
·	0101	5	5
	1011		
+	11		
	0111	7	7

Motivation #1

- Consider the following formula which obviously holds over the integers:
  - $(x - y > 0) \Leftrightarrow (x > y)$
- Does the formula still hold if x and y are interpreted as finite-width bit vectors?

Motivation #2

- Consider the following C program:

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

- Note: unsigned char occupies 8 bits
- What is the output?

Special Interpretations

- Modern SAT solvers uses integers to represent literals, for example
  - x_3 may be represented as 3, and in this case
  - $\neg x_3$ is represented as -3

Two's Complement

```
unsigned variable_index(unsigned literal) {
  if (literal < 0) return -literal;
  else return literal;
}
```

$$(b_{n-1}\cdots b_2 b_1 b_0)_2$$

b_0: sign bit

```
unsigned variable_index(unsigned literal) {
  return literal >> 1;
}
```

Bit Vector Syntax and Semantics

Syntax

$$
\begin{aligned}
\mathit{formula} &: \ \mathit{formula} \wedge \mathit{formula} \mid \neg \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &: \ \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean{-}Identifier} \mid \mathit{term}[\mathit{constant}] \\
\mathit{rel} &: < \mid = \\
\mathit{term} &: \ \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \sim \mathit{term} \mid \mathit{constant} \mid \mathit{atom}?\mathit{term} : \mathit{term} \mid \\
&\quad \mathit{term}[\mathit{constant} : \mathit{constant}] \mid \mathit{ext}(\mathit{term}) \\
\mathit{op} &: \ + \mid - \mid \cdot \mid / \mid \ll \mid \gg \mid \& \mid \ \mid \mid \oplus \mid \circ
\end{aligned}
$$

- $\sim$: bitwise negation
- _?_ : _: case-split
- $\ll$: left shit
- $\gg$: right shift
- $\oplus$: bitwise XOR
- $\circ$: concatenation

Defined Operators

- $f_1 \vee f_2 \triangleq \neg(\neg f_1 \wedge \neg f_2)$
- $t_1 \neq t_2 \triangleq \neg(t_1 = t_2)$
- $t_1 \leq t_2 \triangleq t_1 < t_2 \vee t_1 = t_2$
- $t_1 > t_2 \triangleq \neg(t_1 \leq t_2)$
- $t_1 \geq t_2 \triangleq t_1 > t_2 \vee t_1 = t_2$

λ-Notation

- λ-notation: a lambda expression for a bit vector with l bits has the form:
  - $\lambda i \in \{0,\ldots,l-1\}\,.f(i)$, where
  - $f(i)$ is an expression that denotes the value of the i-th bit
- Examples:
  - $\lambda i \in \{0,\ldots,7\}.0$ denotes $(00000000)_2$
  - $\lambda i \in \{0,\ldots,7\}\,.\begin{cases}0\text{: } i \text{ is even}\\ 1\text{: otherwise}\end{cases}$ denotes $(10101010)_2$
- Exercise: write a λ-expression that denotes the bitwise negation of the vector x

the domain may be omitted if the length is clear from the context

Semantics

- $b : \{0,\ldots,l-1\} \rightarrow \{0,1\}$: a bit vector b is a vector of bits with a given length l
- $bvec_l$: the set of all 2^l bit vectors of length l
- b_i: the i-th bit of the bit vector b is
- The meaning of a bit-vector formula obviously depends on the width of the bit-vector variables in it
  - $x \neq y \wedge x \neq z \wedge y \neq z$, where
  - x, y, z are bit vectors of arbitrary bit lengths

Explicit Bit Length

- Add indices in square brackets to the operator and operands in order to denote the bit width
- Examples:
  - $a_{[32]} \cdot_{[32]} b_{[32]}$: denotes the multiplication of a and b where the result and the operands are 32 bits wide
  - $a_{[8]} \circ_{[24]} b_{[16]}$: denotes the concatenation of a and b and is in total 24 bits wide
- The subscript may be omitted when the width is clear from the context

Semantics: Bitwise Operators

- Unary operator $\sim$: bitwise negation
  - $\sim a \doteq \lambda i \,.\, \neg a_i$
- The binary bitwise operators take two l-bit bit vectors as arguments and return an l-bit bit vector
  - $\star \in \{\ \&\ ,\ |\ ,\ \oplus\ \},\ \ \star_{[l]} : (bvec_l \times bvec_l) \to bvec_l$
  - $a \,\&\, b \doteq \lambda i \,.\, (a_i \wedge b_i)$
  - $a \mid b \doteq \lambda i \,.\, (a_i \vee b_i)$
  - $a \oplus b \doteq \lambda i \,.\, (a_i \vee b_i) \wedge (\neg a_i \vee \neg b_i)$

Binary Encoding for Nonnegative integers

- Let x denote a natural number, and $b \in bvec_l$ a bit vector
- b is called a binary encoding of x iff $x = \langle b \rangle_U$, where $\langle b \rangle_U$ is defined as follows:
  - $\langle \cdot \rangle_U : bvec_l \to \{0,\ldots,2^l - 1\}$
  - $\langle b \rangle_U \doteq \sum_{i=0}^{l-1} b_i \cdot 2^i$
- b_0: least significant bit (LSB) of b
- b_{l-1}: most significant bit (MSB) of b

Two's Complement for Integers

- Let x denote a natural number, and $b \in bvec_l$ a bit vector
- b is called a two's complement of x iff $x = \langle b \rangle_S$, where $\langle b \rangle_S$ is defined as follows:
  - $\langle \cdot \rangle_S : bvec_l \to \{-2^{l-1}, \ldots, 0, \ldots, 2^{l-1} - 1\}$
  - $\langle b \rangle_S \doteq -2^{l-1} \cdot b_{l-1} + \sum_{i=0}^{l-2} b_i \cdot 2^i$
- b_{l-1}: sign bit of b

Encoding-Relevant Operators

- The following bit vector operators have meanings depending on whether a binary encoding or a two's complement encoding is used for the operands
  - $<, >, \leq, \geq$
  - $\cdot, /$
  - $\gg$
- Use the subscript U for a binary encoding (unsigned) and the subscript S for a two's complement encoding (signed)
- May omit this subscript if the encoding is clear from the context, or if the meaning of the operator does not depend on the encoding (this is the case for most operators)

Semantics: Addition and Subtraction

- Arithmetic on bit vectors has a wraparound effect
- This corresponds to a modulo operation
- Semantics of bit vector addition and subtraction:
  - $a_{[l]} +_U b_{[l]} = c_l \Leftrightarrow \langle a \rangle_U + \langle b \rangle_U \equiv \langle c \rangle_U \pmod{2^l}$
  - $a_{[l]} -_U b_{[l]} = c_l \Leftrightarrow \langle a \rangle_U - \langle b \rangle_U \equiv \langle c \rangle_U \pmod{2^l}$
  - $a_{[l]} +_S b_{[l]} = c_l \Leftrightarrow \langle a \rangle_S + \langle b \rangle_S \equiv \langle c \rangle_S \pmod{2^l}$
  - $a_{[l]} -_S b_{[l]} = c_l \Leftrightarrow \langle a \rangle_S - \langle b \rangle_S \equiv \langle c \rangle_S \pmod{2^l}$

$$1011 +_U 0111 = 0010$$
$$\Leftrightarrow \langle 1011 \rangle_U + \langle 0111 \rangle_U \equiv \langle 0010 \rangle_U \pmod{2^4}$$
$$\Leftrightarrow 11 + 7 \equiv 2 \pmod{16}$$

$$0110 +_S 0100 = 1010$$
$$\Leftrightarrow \langle 0110 \rangle_S + \langle 0100 \rangle_S \equiv \langle 1010 \rangle_S \pmod{2^4}$$
$$\Leftrightarrow 6 + 4 \equiv -6 \pmod{16}$$

Semantics: Mixed-Type Addition/Subtraction and Unary Minus

- Note that $a +_U b = a +_S b$ and $a -_U b = a -_S b$
- Semantics for mixed-type expressions can be defined easily
  - $a_{[l]U} +_U b_{[l]S} = c_{[l]U} \Leftrightarrow \langle a \rangle + \langle b \rangle_S = \langle c \rangle \pmod{2^l}$
- Unary minus
  - $-a_{[l]} = b_l \Leftrightarrow -\langle a \rangle_S \equiv \langle b \rangle_S \pmod{2^l}$

Semantics: Relational Operators

- Semantics:
  - $a_{[l]U} < b_{[l]U} \Leftrightarrow \langle a \rangle_U < \langle b \rangle_U$
  - $a_{[l]S} < b_{[l]S} \Leftrightarrow \langle a \rangle_S < \langle b \rangle_S$
  - $a_{[l]U} < b_{[l]S} \Leftrightarrow \langle a \rangle_U < \langle b \rangle_S$
  - $a_{[l]S} < b_{[l]U} \Leftrightarrow \langle a \rangle_S < \langle b \rangle_U$
- Note that ANSI-C compilers do not implement the relational operators on operands with mixed encodings the way they are formalized above
- Instead, the signed operand is converted to an unsigned operand

Semantics: Multiplication and Division

- Semantics
  - $a_{[l]} \cdot_U b_{[l]} = c_l \Leftrightarrow \langle a \rangle_U \cdot \langle b \rangle_U \equiv \langle c \rangle_U \pmod{2^l}$
  - $a_{[l]} /_U b_{[l]} = c_l \Leftrightarrow \langle a \rangle_U / \langle b \rangle_U \equiv \langle c \rangle_U \pmod{2^l}$
  - $a_{[l]} \cdot_S b_{[l]} = c_l \Leftrightarrow \langle a \rangle_S \cdot \langle b \rangle_S \equiv \langle c \rangle_S \pmod{2^l}$
  - $a_{[l]} /_S b_{[l]} = c_l \Leftrightarrow \langle a \rangle_S / \langle b \rangle_S \equiv \langle c \rangle_S \pmod{2^l}$
- The semantics of multiplication is independent of whether the arguments are interpreted as unsigned or two's complement
- This does not hold in the case of division

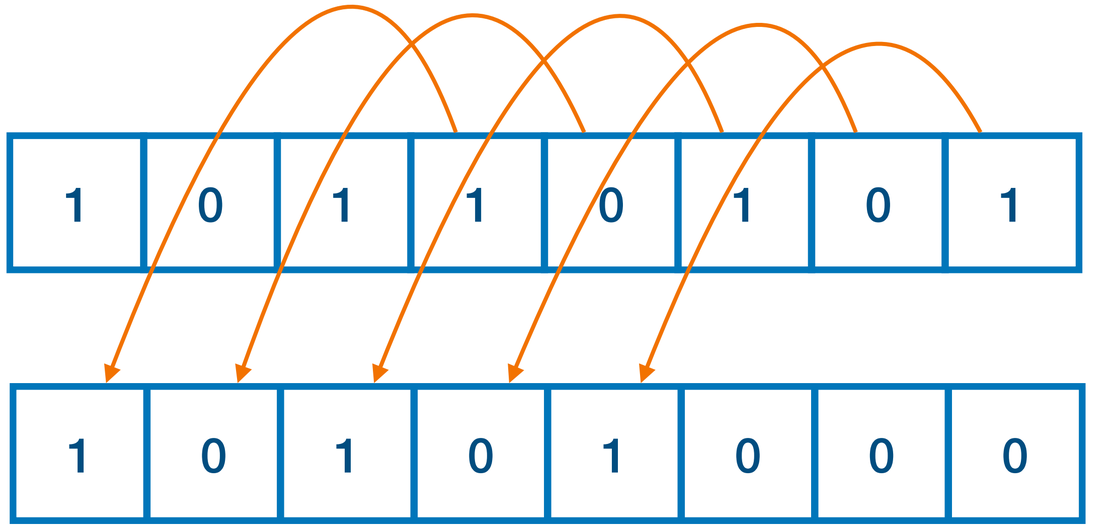
Semantics: Extension Operator

- Extension operator: convert a bit vector to a bit vector with more bits
  - Unsigned case: zero extension
  - Signed case: sign extension
- Let $l \leq m$
  - Zero extension: $ext_{[m]U}(a_{[l]}) = b_{[m]U} \Leftrightarrow \langle a \rangle_U = \langle b \rangle_U$
  - Sign extension: $ext_{[m]S}(a_{[l]}) = b_{[m]S} \Leftrightarrow \langle a \rangle_S = \langle b \rangle_S$

Semantics: Shifting Operators

- Left shift operator $\ll$
  - $a_{[l]} \ll b_U = \lambda i \in \{0,\ldots,l-1\} . \begin{cases} a_{i-\langle b \rangle_U} & : i \geq \langle b \rangle_U \\ 0 & : \text{otherwise} \end{cases}$
    - b_U is called the shift distance

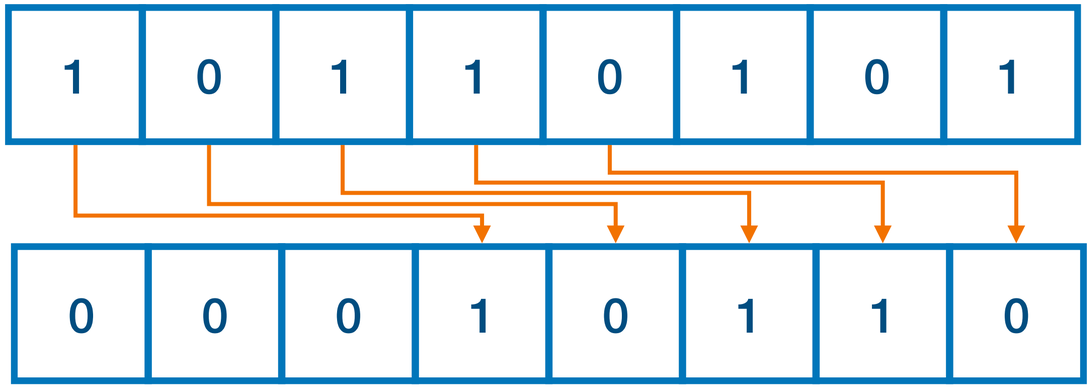
$10110101 \ll 00000011_U$



Semantics: Logical Right Shift

- Right shift operator $\gg$
  - Unsigned case: logical right shift
    - $a_{[l]U} \gg b_U = \lambda i \in \{0,\ldots,l-1\} . \begin{cases} a_{i+\langle b \rangle_U} & : i < l - \langle b \rangle_U \\ 0 & : \text{otherwise} \end{cases}$

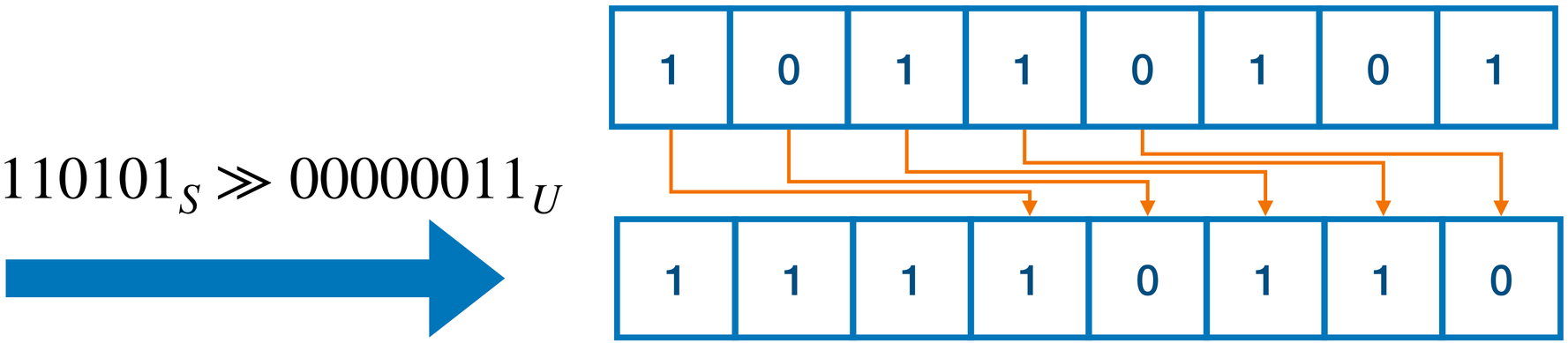
$10110101_U \gg 00000011_U$



Semantics: Arithmetic Right Shift

- Right shift operator $\gg$
  - Signed case: arithmetic right shift
    - $$a_{[l]S} \gg b_U = \lambda i \in \{0,\ldots,l-1\} . \begin{cases} a_{i+\langle b \rangle_U} & : i < l - \langle b \rangle_U \\ a_{l-1} & : \text{otherwise} \end{cases}$$

$$10110101_S \gg 00000011_U$$



Deciding Bit-Vector Arithmetic

BV-Flattening

- The most commonly used decision procedure for bit-vector arithmetic is called **flattening** (or called **bit-blasting**)
- For a given bit-vector arithmetic formula φ, the algorithm BV-Flattening computes an equisatisfiable propositional formula $\mathfrak{B}$, which is then passed to a SAT solver

```
1.  function BV-Flattening
2.    𝔅 := e(φ);
3.    for each t_[l] ∈ T(φ) do
4.      for each i ∈ {0,...,l − 1} do
5.        set e(t)_i to a new Boolean variable;
6.    for each a ∈ At(φ) do
7.      𝔅 := 𝔅 ∧ BV-Constraint(e, a);
8.    for each t_[l] ∈ T(φ) do
9.      𝔅 := 𝔅 ∧ BV-Constraint(e, t);
10.   return 𝔅;
```

BV-Flattening Step 1: Propositional Skeleton

- Let $At(\varphi)$ denote the set of atoms in φ
- The first step replaces the atoms in φ with new Boolean Variables
  - An atom $a \in At(\varphi)$ becomes its propositional encoder $e(a)$
  - The resulting formula is denoted by $e(\varphi)$

$\varphi := x_{[8]U} +_U y_{[8]U} = z_{[8]U} \wedge x_{[8]U} \leq y_{[8]U}$

$e(\varphi) := e(x_{[8]U} +_U y_{[8]U} = z_{[8]U}) \wedge e(x_{[8]U} \leq y_{[8]U})$

```
1.  function BV-Flattening
2.    𝔅 := e(φ);
3.    for each t_[l] ∈ T(φ) do
4.      for each i ∈ {0,...,l − 1} do
5.        set e(t)_i to a new Boolean variable;
6.    for each a ∈ At(φ) do
7.      𝔅 := 𝔅 ∧ BV-Constraint(e, a);
8.    for each t_[l] ∈ T(φ) do
9.      𝔅 := 𝔅 ∧ BV-Constraint(e, t);
10.   return 𝔅;
```

BV-Flattening Step 2: Bit-Vector Terms

- Let $T(\varphi)$ denote the set of terms in φ
- Step 2 assigns each bit-vector term $t \in T(\varphi)$ a vector $e(t)$ of new Boolean variables
  - $e(t)_i$ denotes the variable for the bit with index i of the term t

$e(\varphi) := e(x_{[8]U} +_U y_{[8]U} = z_{[8]U}) \wedge e(x_{[8]U} \leq y_{[8]U})$

$e(x_{[8]U}) := x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0$ $\quad e(x_{[8]U})_5 = x_5$

...

```
1.  function BV-Flattening
2.    𝔅 := e(φ);
3.    for each t_[l] ∈ T(φ) do
4.      for each i ∈ {0,...,l − 1} do
5.        set e(t)_i to a new Boolean variable;
6.    for each a ∈ At(φ) do
7.      𝔅 := 𝔅 ∧ BV-Constraint(e, a);
8.    for each t_[l] ∈ T(φ) do
9.      𝔅 := 𝔅 ∧ BV-Constraint(e, t);
10.   return 𝔅;
```

BV-Flattening Step 3: Constraints

- Step 3 iterates over the terms and atoms of φ and computes a constraint for each of them
- Constraints for terms
  - Bit vector variable: $True$
  - Boolean variable: $True$
  - Bit vector constant $C_{[l]}$:
    - $\bigwedge_{i=0}^{l-1} (C_i \Leftrightarrow e(t)_i)$

```
1.  function BV-Flattening
2.    𝔅 := e(φ);
3.    for each t_[l] ∈ T(φ) do
4.      for each i ∈ {0,...,l − 1} do
5.        set e(t)_i to a new Boolean variable;
6.    for each a ∈ At(φ) do
7.      𝔅 := 𝔅 ∧ BV-Constraint(e, a);
8.    for each t_[l] ∈ T(φ) do
9.      𝔅 := 𝔅 ∧ BV-Constraint(e, t);
10.   return 𝔅;
```

$e(1011) = b_3 b_2 b_1 b_0$

BV-Constraint(e, 1011) $= b3 \Leftrightarrow 1 \wedge b2 \Leftrightarrow 0 \wedge b1 \Leftrightarrow 1 \wedge b0 \Leftrightarrow 1$

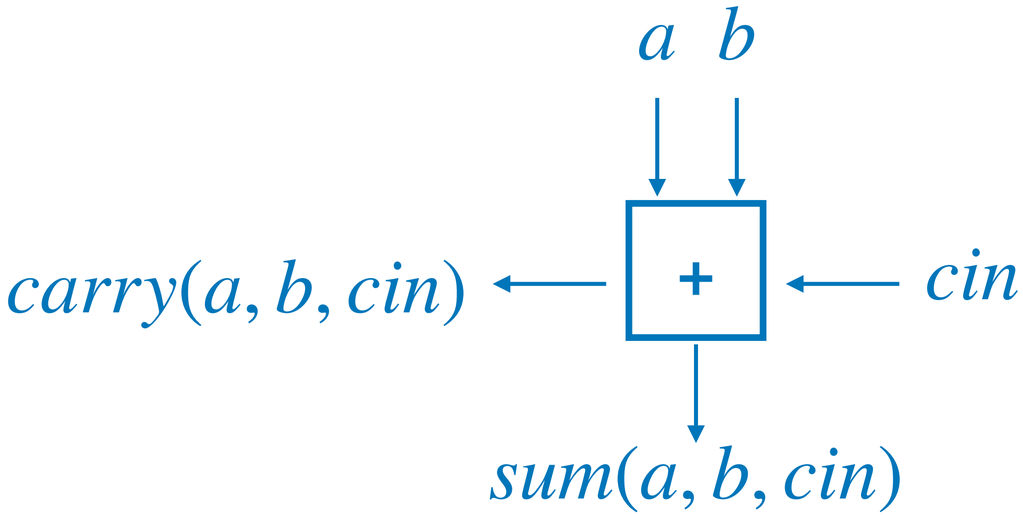
Constraints for Atoms: Bitwise Operators

- If $t \triangleq \sim_{[l]} a$, then BV-Constraint$(e, t) := \bigwedge_{i=0}^{l-1} ((\neg e(a)_i) \Leftrightarrow e(t)_i)$
- If $t \triangleq a \,\&_l\, b$, then BV-Constraint$(e, t) := \bigwedge_{i=0}^{l-1} ((e(a)_i \wedge e(b)_i) \Leftrightarrow e(t)_i)$
- If $t \triangleq a \,|_l\, b$, then BV-Constraint$(e, t) := \bigwedge_{i=0}^{l-1} ((e(a)_i \vee e(b)_i) \Leftrightarrow e(t)_i)$
- If $t \triangleq a \oplus_l b$, then BV-Constraint$(e, t) := \bigwedge_{i=0}^{l-1} (((e(a)_i \vee e(b)_i) \wedge (\neg e(a)_i \vee \neg e(b)_i)) \Leftrightarrow e(t)_i)$

Constraints for Atoms: Addition #1

- One-bit Full Adder -

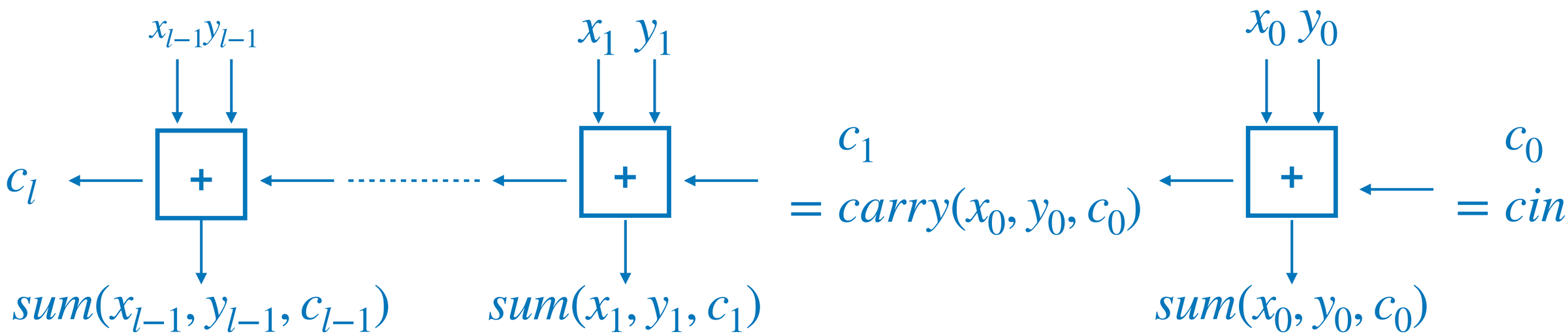
- A one-bit full adder is defined using the two functions *carry* and *sum*
  - $sum(a, b, cin) \doteq (a \oplus b) \oplus cin$
  - $carry(a, b, cin) \doteq (a \wedge b) \vee ((a \oplus b) \wedge cin)$



Constraints for Atoms: Addition #2

- Carry Bits -

- Let x and y denote two l-bit bit vectors and cin a single bit
- The carry bits c_0 to c_l are defined recursively:
  - $c_i \doteq \begin{cases} cin & : i = 0 \\ carry(x_{i-1}, y_{i-1}, c_{i-1}) & : otherwise \end{cases}$



Constraints for Atoms: Addition #3

- Adder -

- An l-bit adder maps two l-bit bit vectors x, y, and a carry-in bit cin to their sum and a carry-out bit
- Let c_i denote the i-th carry bit as in the previous slide
- The function add is defined as the follows
- $$add(x, y, cin) \doteq \langle result, cout \rangle$$
$$result_i \doteq sum(x_i, y_i, c_i) \quad for\ i \in \{0, \ldots, l-1\}$$
$$cout \doteq c_l$$

Constraints for Atoms: Addition #4

- Constraint for Addition -

- The constraint for $t \triangleq a_{[l]} + b_{[l]}$ is given by the add function with $cin = 0$
  - $\bigwedge_{i=0}^{l-1} (add(a, b, 0) \,.\, result_i \Leftrightarrow e(t)_i)$
- Lemma: the above constraint holds if and only if $\langle a \rangle_U + \langle b \rangle_U \equiv \langle e(t) \rangle_U \pmod{2^l}$
  - Prove by induction on l

Constraints for Atoms: Subtraction

- Observe that $\langle (\sim b) + 1 \rangle_S = -\langle b \rangle_S \pmod{2^l}$
- The constraint for $t \triangleq a_{[l]} - b_{[l]}$ is then also given by the *add* function
  - $\bigwedge_{i=0}^{l-1} (add(a, \sim b, 1) \,.\, result_i \Leftrightarrow e(t)_i)$

Constraints for Atoms: Relational Operators

- Constraint for $t \triangleq a =_{[l]} b$: $\bigwedge_{i=0}^{l-1} ((a_i = b_i) \Leftrightarrow e(t)_i)$

- Constraint for $a < b$ is the constraint for $a - b < 0$:
  - Unsigned case: $\langle a \rangle_U < \langle b \rangle_U \Leftrightarrow \neg add(a, \sim b, 1)\,.\,cout$
  - Signed case: $\langle a \rangle_S < \langle b \rangle_S \Leftrightarrow (a_{l-1} \Leftrightarrow b_{l-1}) \oplus add(a, \sim b, 1)\,.\,cout$

$\neg add(a, \sim b, 1)\,.\,cout$: need a borrow in $a - b$

Signed cases:

$a \triangleq 0 - - \cdots -$
$b \triangleq 0 - - \cdots -$

$a \triangleq 1 - - \cdots -$
$b \triangleq 1 - - \cdots -$

$a \triangleq 0 - - \cdots -$
$b \triangleq 1 - - \cdots -$

$a \triangleq 1 - - \cdots -$
$b \triangleq 0 - - \cdots -$

Constraints for Atoms: Shifting Operators

Idea: the following results are the same

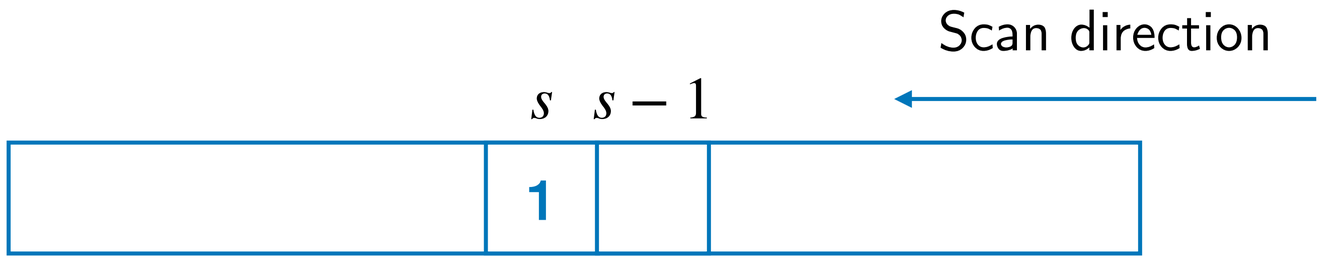
- shift by 6
- shift by 2 and then shift by 4

- Additional restriction is put on shifting operators
  - Consider $a_{[l]} \star b_{[n]}$ for $\star \in \{ \ll, \gg \}$, $l = 2^n$ must hold
- The left shifter ls is split into n stages for $s \in \{-1, \ldots, n-1\}$
  - $ls(a_{[l]}, b_{[n]U}, -1) \doteq a$
  - $$ls(a_{[l]}, b_{[n]U}, s) \doteq \lambda i \in \{0, \ldots, l-1\} . \begin{cases} (ls(a, b, s-1))_{i-2^s} & : i \geq 2^s \wedge b_s \\ (ls(a, b, s-1))_i & : \neg b_s \\ 0 & : otherwise \end{cases}$$

Constraints for Atoms: Shifting Operators (cont'd)

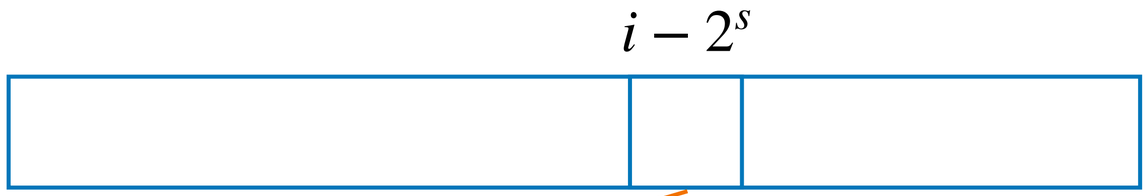
$b \triangleq b_{n-1} \cdots b_1 b_0$

$(\langle b \rangle_U = b_{n-1} 2^{n-1} + \cdots + b_1 2^1 + b_0 2^0)$



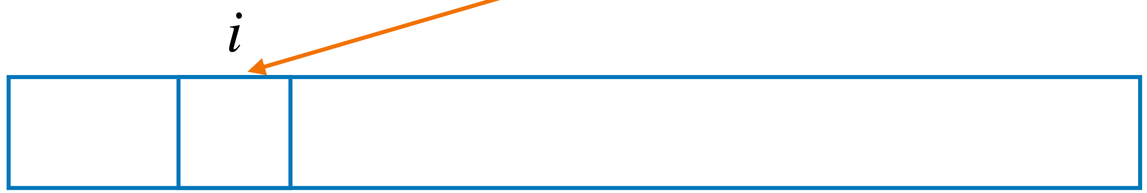
$ls(a, b, s-1)$

(a shifted by $b_{s-1} \cdots b_1 b_0$)



$ls(a, b, s)$

(under b_s)



$\neg b_s$: the bit does not contribute to the shift

b_s: the bit contributes 2^s shifts

Constraints for Atoms: Multiplication

- Constraint for multiplication is based on circuit design, which uses the shift-and-add idea
  - $mul(a, b, -1) \doteq 0$
  - $mul(a, b, s) \doteq mul(a, b, s-1) + (b_s ? (a \ll s) : 0)$
- A division $t \triangleq a/_U b$ is implemented by adding two constraints:
  - $b \neq 0 \Rightarrow e(t) \cdot b + r = a$
  - $b \neq 0 \Rightarrow r < b$

Verification Conditions

Overflow Checking

- Verification conditions are formulas that specify program correctness
- For example, we may write a bit-vector formula φ to specify the presence of overflows in a program
- If φ is unsatisfiable, then the program won't overflow

```
uint32_t add(uint32_t x, uint32_t y) {
  return x + y;
}
```

$$\varphi = ?$$

Hoare Logic Specifications

- Consider a Hoare triple $\{P\}\ C\ \{Q\}$
- We may compute the weakest precondition $wp(C, Q)$ of Q w.r.t. C such that $P \Rightarrow wp(C, Q)$ is valid if and only if $\{P\}\ C\ \{Q\}$ is valid
- We may also generate a verification condition $\varphi = P \wedge \varphi_C \wedge \neg Q$ such that φ is unsatisfiable if and only if $\{P\}\ C\ \{Q\}$ is valid
  - φ_C is a formula describing the semantics of the program C
  - C may be transformed into static single assignment (SSA) form in order to get φ_C

Static Single Assignment

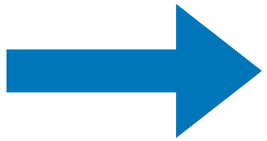
- A program is in SSA if it satisfies the following two conditions
  - Every variable is defined before it's use
  - Every variable is assigned at most once
- To transform a program into SSA, we may
  - rename variables by adding indices, and
  - use ϕ-functions at the join of control flows

Static Single Assignment (cont'd)

variable types and declarations are omitted for simplicity

```
function foo(x, y) {
  x = x + x;
  y = x + y;
  x = 2 * x;
  z = x + y;
  return z;
}
```

SSA



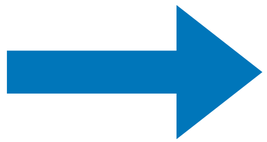
```
function foo(x0, y0) {
  x1 = x0 + x0;
  y1 = x1 + y0;
  x2 = 2 * x1;
  z1 = x2 + y1;
  return z1;
}
```

Static Single Assignment (cont'd)

```
function bar(x) {
  y = 1;
  count = 0;
loop:
  if (count < 3) {
    y = y * x;
    goto loop;
  }
  return y;
}
```

SSA

(partially done)



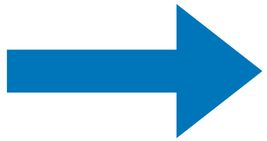
```
function bar(x0) {
  y1 = 1;
  count1 = 0;
loop:
  if (count < 3) {
    y = y * x;
    count = count + 1;
    goto loop;
  }
  return y;
}
```

Static Single Assignment (cont'd)

```
function bar(x) {
  y = 1;
  count = 0;
loop:
  if (count < 3) {
    y = y * x;
    goto loop;
  }
  return y;
}
```

SSA

(partially done)

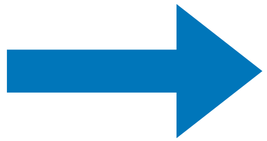


```
function bar(x0) {
  y1 = 1;
  count1 = 0;
loop:
  y2 = ϕ(y1, ...);
  count2 = ϕ(count1, ...);
  if (count < 3) {
    y = y * x;
    count = count + 1;
    goto loop;
  }
  return y;
}
```

Static Single Assignment (cont'd)

```
function bar(x) {
  y = 1;
  count = 0;
loop:
  if (count < 3) {
    y = y * x;
    goto loop;
  }
  return y;
}
```

SSA

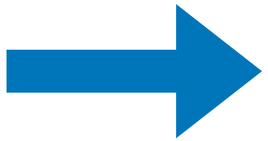


```
function bar(x0) {
  y1 = 1;
  count1 = 0;
loop:
  y2 = ϕ(y1, y3);
  count2 = ϕ(count1, count3);
  if (count2 < 3) {
    y3 = y2 * x0;
    count3 = count2 + 1;
    goto loop;
  }
  return y2;
}
```

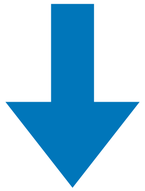
Generate Verification Conditions

```
{ true }
if (x > y)
  r = x;
else
  r = y;
{ r ≥ x ∧ r ≥ y }
```

SSA



```
{ true }
if (x0 > y0)
  r1 = x0;
else
  r2 = y0;
r3 = ϕ(r1, r2)
{ r3 ≥ x0 ∧ r3 ≥ y0 }
```



P	$\wedge\, \varphi_{\mathbf{C}}$	$\wedge\, \neg Q$
true	$\wedge\, (r1 = x0 \wedge r2 = y0 \wedge r3 = \phi(r1,r2))$	$\wedge\, \neg(r3 \geq x0 \wedge r3 \geq y0)$
true	$\wedge\, (r1 = x0 \wedge r2 = y0 \wedge (x0 > y0 \Rightarrow r3 = r1) \wedge (\neg(x0 > y0) \Rightarrow r3 = r2))$	$\wedge\, \neg(r3 \geq x0 \wedge r3 \geq y0)$

Performance and Improvements

Performance of Flattening

- The size of the formula generated by BV-Constraint may be large for some operators such as multiplication
- In addition to the sheer size of these formulas, their symmetry and connectivity is a burden on the decision heuristic of state-of-the-art propositional SAT solvers
- As a consequence, formulas with multipliers are often very hard

Size of the constraint for an n-bit multiplier after Tseitin's transformation

n	# of Vars	# of Clauses
8	313	1,001
16	1,265	4,177
24	2,857	9,529
32	5,089	17,057
64	20,417	68,929

Improvements: Rewriting and Incremental Bit Flattening

- Formula rewriting
  - $a \cdot (b \cdot c) = d \wedge (a \cdot b) \cdot c \neq d$
  - The bit-blasting results of $a \cdot (b \cdot c)$ and $(a \cdot b) \cdot c$ are totally different
- Incremental bit flattening
  - $a \cdot b = c \wedge b \cdot a \neq c \wedge x < y \wedge x > y$
  - There are two inconsistent subformulas: $a \cdot b = c \wedge b \cdot a \neq c$, $x < y \wedge x > y$
  - Flattening $x < y \wedge x > y$ is much easier than flattening the former one

Improvements: Incremental Bit Flattening

Procedure of incremental bit flattening

1. Make the propositional skeleton of φ
2. If SAT solving returns unsat, return unsat
3. Otherwise, find atoms that are inconsistent with the satisfying assignment
4. If there is no such atoms, return sat
5. Otherwise, select "easy" inconsistent atoms and add constraints for flattening them
6. Go to Step 2

$\varphi \triangleq a \cdot b = c \wedge b \cdot a \neq c \wedge x < y \wedge x > y$

1. $e(\varphi) = e(a \cdot b = c) \wedge e(b \cdot a \neq c) \wedge e(x < y) \wedge e(x > y)$
2. Find satisfying assignment:
   - $\{e(a \cdot b = c) \mapsto 1,\ e(b \cdot a \neq c) \mapsto 1,\ e(x < y) \mapsto 1,\ e(x > y) \mapsto 1\}$
3. Inconsistent atoms:
   - $\{e(a \cdot b = c),\ e(b \cdot a \neq c),\ e(x < y),\ e(x > y)\}$
4. The set of inconsistent atoms is not empty
5. Select $\{x < y,\ x > y\}$ for flattening
6. The SAT solving now has enough information to conclude unsat

Improvements: Abstraction with Uninterpreted Functions

- Abstraction with uninterpreted functions
  - This technique is particularly effective when one is checking the equivalence of two models

```
int32_t power3_1(int32_t in0) {
  int32_t out_a0 = in0;
  int32_t out_a1 = out_a0 * in0;
  int32_t out_a2 = out_a1 * in0;
  return out_a2;
}
```

```
int32_t power3_2(int32_t in0) {
  int32_t out_b0 = (in0 * in0) * in0;
 return out_b0;
}
```

Improvements: Abstraction with Uninterpreted Functions (cont'd)

```
int32_t power3_1(int32_t in0) {
  int32_t out_a0 = in0;
  int32_t out_a1 = out_a0 * in0;
  int32_t out_a2 = out_a1 * in0;
  return out_a2;
}
```

```
int32_t power3_2(int32_t in0) {
  int32_t out_b0 = (in0 * in0) * in0;
 return out_b0;
}
```

$$
\begin{aligned}
& out\_a0_{[32]} = in0_{[32]} \\
\wedge\ & out\_a1_{[32]} = out\_a0_{[32]} \cdot_S in0_{[32]} \\
\wedge\ & out\_a2_{[32]} = out\_a1_{[32]} \cdot_S in0_{[32]} \\
\wedge\ & out\_b0_{[32]} = (in0_{[32]} \cdot_S in0_{[32]}) \cdot_S in0_{[32]} \\
\wedge\ & \neg(out\_a2_{[32]} = out\_b0_{[32]})
\end{aligned}
$$

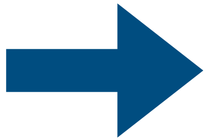
Checking such bit-vector formula is time consuming

Instead, we can replace $\cdot_S$ with an uninterpreted function

Improvements: Abstraction with Uninterpreted Functions (cont'd)

Original Formula

$$
\begin{aligned}
& out\_a0_{[32]} = in0_{[32]} \\
\wedge\ & out\_a1_{[32]} = out\_a0_{[32]} \cdot_S in0_{[32]} \\
\wedge\ & out\_a2_{[32]} = out\_a1_{[32]} \cdot_S in0_{[32]} \\
\wedge\ & out\_b0_{[32]} = (in0_{[32]} \cdot_S in0_{[32]}) \cdot_S in0_{[32]} \\
\wedge\ & \neg(out\_a2_{[32]} = out\_b0_{[32]})
\end{aligned}
$$



Abstract Formula

$$
\begin{aligned}
& out\_a0_{[32]} = in0_{[32]} \\
\wedge\ & out\_a1_{[32]} = G(out\_a0_{[32]}, in0_{[32]}) \\
\wedge\ & out\_a2_{[32]} = G(out\_a1_{[32]}, in0_{[32]}) \\
\wedge\ & out\_b0_{[32]} = G(G(in0_{[32]}, in0_{[32]}), in0_{[32]}) \\
\wedge\ & \neg(out\_a2_{[32]} = out\_b0_{[32]})
\end{aligned}
$$

The abstract formula is now easier to check without bit flattening

- If the abstracts formula is unsatisfiable, then the original formula is also unsatisfiable
- Otherwise, the original formula has to be checked