Matching Multiplications in Bit-Vector formulas

Rahul Jain

TIFR, Mumbai

VMCAI: January 15, 2017

Co-authors: Supratik Chakraborty[1] Ashutosh Gupta[2]

[1]IIT Bombay
[2]TIFR, Mumbai

SMT formula encoding a long multiplication

```
(set-logic QF_BV)
(declare-fun v1 () (_ BitVec 8))
(declare-fun v2 () (_ BitVec 8))
(declare-fun v3 () (_ BitVec 8))
(declare-fun v4 () (_ BitVec 8))
(declare-fun v5 () (_ BitVec 8))
(declare-fun v6 () (_ BitVec 8))
(declare-fun v7 () (_ BitVec 8))
(declare-fun v8 () (_ BitVec 8))
(define-fun e1 () (_ BitVec 16) (concat #b00000000 v1))
(define-fun e2 () (_ BitVec 16) (concat #b00000000 v2))
(define-fun e3 () (_ BitVec 16) (concat #b00000000 v3))
(define-fun e4 () (_ BitVec 16) (concat #b00000000 v4))
(define-fun e5 () (_ BitVec 16) (concat #b00000000 v5))
(define-fun e6 () (_ BitVec 16) (concat #b00000000 v6))
(define-fun e7 () (_ BitVec 16) (concat #b00000000 v7))
(define-fun e8 () (_ BitVec 16) (concat #b00000000 v8))

(assert
(not (=
 (bvmul (concat (concat (concat (concat #b00000000000000000000000000000000 v1) v2) v3) v4)
        (concat (concat (concat (concat #b00000000000000000000000000000000 v5) v6) v7) v8) )

 (bvadd  (concat         (bvmul e1 e5)     #b000000000000000000000000000000000000000000000000)
 (bvadd  (concat (concat #b00000000 (bvmul e1 e6)) #b0000000000000000000000000000000000000000)
 (bvadd  (concat (concat #b0000000000000000 (bvmul e1 e7)) #b00000000000000000000000000000000)
 (bvadd  (concat (concat #b000000000000000000000000 (bvmul e1 e8)) #b000000000000000000000000)
 (bvadd  (concat (concat #b00000000 (bvmul e2 e5)) #b0000000000000000000000000000000000000000)
 (bvadd  (concat (concat #b0000000000000000 (bvmul e2 e6)) #b00000000000000000000000000000000)
 (bvadd  (concat (concat #b000000000000000000000000 (bvmul e2 e7)) #b000000000000000000000000)
 (bvadd  (concat (concat #b00000000000000000000000000000000 (bvmul e2 e8)) #b0000000000000000)
 (bvadd  (concat (concat #b0000000000000000 (bvmul e3 e5)) #b00000000000000000000000000000000)
 (bvadd  (concat (concat #b000000000000000000000000 (bvmul e3 e6)) #b000000000000000000000000)
 (bvadd  (concat (concat #b00000000000000000000000000000000 (bvmul e3 e7)) #b0000000000000000)
 (bvadd  (concat (concat #b0000000000000000000000000000000000000000 (bvmul e3 e8)) #b00000000)
 (bvadd  (concat (concat #b000000000000000000000000 (bvmul e4 e5)) #b000000000000000000000000)
 (bvadd  (concat (concat #b00000000000000000000000000000000 (bvmul e4 e6)) #b0000000000000000)
 (bvadd  (concat (concat #b0000000000000000000000000000000000000000 (bvmul e4 e7)) #b00000000)
         (concat         #b000000000000000000000000000000000000000000000000 (bvmul e4 e8) )

 )))))))))))))))
)))
(check-sat)
```

Operations count:

- multiplications - 17
- additions - 15
- concatenations - 46

Solvers timing out(24 hours):
z3, CVC4, Boolector

SMT formula encoding a long multiplication

```
(set-logic QF_BV)
(declare-fun v1 () (_ BitVec 8))
(declare-fun v2 () (_ BitVec 8))
(declare-fun v3 () (_ BitVec 8))
(declare-fun v4 () (_ BitVec 8))
(declare-fun v5 () (_ BitVec 8))
(declare-fun v6 () (_ BitVec 8))
(declare-fun v7 () (_ BitVec 8))
(declare-fun v8 () (_ BitVec 8))
(define-fun e1 () (_ BitVec 16) (concat #b00000000 v1))
(define-fun e2 () (_ BitVec 16) (concat #b00000000 v2))
(define-fun e3 () (_ BitVec 16) (concat #b00000000 v3))
(define-fun e4 () (_ BitVec 16) (concat #b00000000 v4))
(define-fun e5 () (_ BitVec 16) (concat #b00000000 v5))
(define-fun e6 () (_ BitVec 16) (concat #b00000000 v6))
(define-fun e7 () (_ BitVec 16) (concat #b00000000 v7))
(define-fun e8 () (_ BitVec 16) (concat #b00000000 v8))

(assert
(not (=
 (bvmul (concat (concat (concat (concat #b00000000000000000000000000000000 v1) v2) v3) v4)
        (concat (concat (concat (concat #b00000000000000000000000000000000 v5) v6) v7) v8) )

 (bvadd  (concat         (bvmul e1 e5)     #b000000000000000000000000000000000000000000000000)
 (bvadd  (concat (concat #b00000000 (bvmul e1 e6)) #b0000000000000000000000000000000000000000)
 (bvadd  (concat (concat #b0000000000000000 (bvmul e1 e7)) #b00000000000000000000000000000000)
 (bvadd  (concat (concat #b000000000000000000000000 (bvmul e1 e8)) #b000000000000000000000000)
 (bvadd  (concat (concat #b00000000 (bvmul e2 e5)) #b0000000000000000000000000000000000000000)
 (bvadd  (concat (concat #b0000000000000000 (bvmul e2 e6)) #b00000000000000000000000000000000)
 (bvadd  (concat (concat #b000000000000000000000000 (bvmul e2 e7)) #b000000000000000000000000)
 (bvadd  (concat (concat #b00000000000000000000000000000000 (bvmul e2 e8)) #b0000000000000000)
 (bvadd  (concat (concat #b0000000000000000 (bvmul e3 e5)) #b00000000000000000000000000000000)
 (bvadd  (concat (concat #b000000000000000000000000 (bvmul e3 e6)) #b000000000000000000000000)
 (bvadd  (concat (concat #b00000000000000000000000000000000 (bvmul e3 e7)) #b0000000000000000)
 (bvadd  (concat (concat #b0000000000000000000000000000000000000000 (bvmul e3 e8)) #b00000000)
 (bvadd  (concat (concat #b000000000000000000000000 (bvmul e4 e5)) #b000000000000000000000000)
 (bvadd  (concat (concat #b00000000000000000000000000000000 (bvmul e4 e6)) #b0000000000000000)
 (bvadd  (concat (concat #b0000000000000000000000000000000000000000 (bvmul e4 e7)) #b00000000)
         (concat         #b000000000000000000000000000000000000000000000000 (bvmul e4 e8) )

 )))))))))))))))
)))
(check-sat)
```

Operations count:

- multiplications - 17
- additions - 15
- concatenations - 46

Solvers timing out(24 hours):
z3, CVC4, Boolector

Our time $<$ 1 second

Outline

Introduction

- SMT solvers are important in verification

Introduction

- SMT solvers are important in verification
- Map system properties to formula satisfiability

Introduction

- SMT solvers are important in verification
- Map system properties to formula satisfiability
- Formulas are in various theories

Introduction

- SMT solvers are important in verification
- Map system properties to formula satisfiability
- Formulas are in various theories
- E.g. Core, Int, Bit-Vectors

Introduction

- SMT solvers are important in verification
- Map system properties to formula satisfiability
- Formulas are in various theories
- E.g. Core, Int, Bit-Vectors

Our work deals with Bit-Vectors

Theory of fixed size bit-vectors($\mathcal{T}_{BV}$)

Definition: vector of Boolean values with a given length ℓ:

$$b : \{0, ..., \ell - 1\} \rightarrow \{0, 1\}$$

Theory of fixed size bit-vectors($\mathcal{T}_{BV}$)

Definition: vector of Boolean values with a given length ℓ:

$$b : \{0, ..., \ell - 1\} \rightarrow \{0, 1\}$$

Why are bit-vectors important?

- Computing systems manipulate finite sequences of 0's and 1's

Theory of fixed size bit-vectors($\mathcal{T}_{BV}$)

Definition: vector of Boolean values with a given length ℓ:

$$b : \{0, ..., \ell - 1\} \to \{0, 1\}$$

Why are bit-vectors important?

- Computing systems manipulate finite sequences of 0's and 1's
- Widely used in hardware and software verification

Decision procedures for $\mathcal{T}_{BV}$

- Aim: To find the satisfiability of a bit-vector formula.

Decision procedures for $\mathcal{T}_{BV}$

- Aim: To find the satisfiability of a bit-vector formula.
- Solvers use **heavy** pre-processing based on rewriting rules

Decision procedures for $\mathcal{T}_{BV}$

- Aim: To find the satisfiability of a bit-vector formula.
- Solvers use **heavy** pre-processing based on rewriting rules
- E.g. pre-processing techniques:
  - Normalisation
  - Substitution
  - Disjunctive partitioning

Decision procedures for $\mathcal{T}_{BV}$

- Aim: To find the satisfiability of a bit-vector formula.
- Solvers use **heavy** pre-processing based on rewriting rules
- E.g. pre-processing techniques:
  - Normalisation
  - Substitution
  - Disjunctive partitioning

Sometimes pre-processing techniques solve the bit-vector formula

Decision procedures for $\mathcal{T}_{BV}$

- Aim: To find the satisfiability of a bit-vector formula.
- Solvers use **heavy** pre-processing based on rewriting rules
- E.g. pre-processing techniques:
  - Normalisation
  - Substitution
  - Disjunctive partitioning

Sometimes pre-processing techniques solve the bit-vector formula

Otherwise,

Bit-blasting invoked by the solver

Bit-blasting

- Given a bit-vector formula ϕ

Bit-blasting

- Given a bit-vector formula ϕ
- Converted to an equisatisfiable propositional formula ψ

Bit-blasting

- Given a bit-vector formula ϕ
- Converted to an equisatisfiable propositional formula ψ
- Formula ψ solved using a SAT solver

Bit-blasting

- Given a bit-vector formula ϕ
- Converted to an equisatisfiable propositional formula ψ
- Formula ψ solved using a SAT solver
- SAT is NP complete

Bit-blasting: Bit-level reasoning

Consider the formula:

$$x \oplus y \neq x|y - x\&y$$

where x and y are bit vectors of length ℓ.

Bit-blasting: Bit-level reasoning

Consider the formula:

$$x \oplus y \neq x|y - x\&y$$

where x and y are bit vectors of length ℓ.

- No obvious pre-processing helps the solver
- Solver resorts to bit-blasting

Bit-blasting: Bit-level reasoning

Consider the formula:

$$x \oplus y \neq x|y - x\&y$$

where x and y are bit vectors of length ℓ.

- No obvious pre-processing helps the solver
- Solver resorts to bit-blasting
- As ℓ increases, solving time increases

Pre-processing: Word-level reasoning

Consider the formula:

$$x + y \neq y + x$$

where x and y are bit vectors of length ℓ

Pre-processing: Word-level reasoning

Consider the formula:

$$x + y \neq y + x$$

where x and y are bit vectors of length ℓ

- Pre-processing helps the solver

Pre-processing: Word-level reasoning

Consider the formula:

$$x + y \neq y + x$$

where x and y are bit vectors of length ℓ

- Pre-processing helps the solver
- Commutativity of addition used at the word-level

Pre-processing: Word-level reasoning

Consider the formula:

$$x + y \neq y + x$$

where x and y are bit vectors of length ℓ

- Pre-processing helps the solver
- Commutativity of addition used at the word-level
- As ℓ increases, solving time does not increase

Pre-processing: Word-level reasoning

Consider the formula:

$$x + y \neq y + x$$

where x and y are bit vectors of length ℓ

- Pre-processing helps the solver
- Commutativity of addition used at the word-level
- As ℓ increases, solving time does not increase

No bit-blasting

Our problem: A class of formulas

SMT solvers do not solve some formulas efficiently

Our problem: A class of formulas

SMT solvers do not solve some formulas efficiently

We consider,

- **Bit-vector formulas with multiplication** inspired from hardware verification

Our problem: A class of formulas

SMT solvers do not solve some formulas efficiently

We consider,

- **Bit-vector formulas with multiplication** inspired from hardware verification
- A class of such formulas involving word-level reasoning over multiplication

Why hard?

Why is this class hard for SMT solvers?

Why hard?

Why is this class hard for SMT solvers?

- Failure to identify word-level reasoning during pre-processing

Why hard?

Why is this class hard for SMT solvers?

- Failure to identify word-level reasoning during pre-processing
- Word-level structure is lost after bit-blasting

Why hard?

Why is this class hard for SMT solvers?

- Failure to identify word-level reasoning during pre-processing
- Word-level structure is lost after bit-blasting
- Dramatic blow up in resulting propositional formula

Why hard?

Why is this class hard for SMT solvers?

- Failure to identify word-level reasoning during pre-processing
- Word-level structure is lost after bit-blasting
- Dramatic blow up in resulting propositional formula

Our work - Enable maximal word-level reasoning for the class of problems

Decomposed multiplication

In hardware verification, there are decomposed multiplications

Decomposed multiplication

In hardware verification, there are decomposed multiplications

For example:

$$
\begin{array}{cccc}
 & 2 & 3 & \\
 & 1 & 3 & * \\
\hline
 & 2*3 & 3*3 & \\
2*1 & 3*1 & + & \\
\hline
\end{array}
$$

Hardware implementation: $(2*1)*10^2+(2*3+3*1)*10^1+(3*3)*10^0$

Decomposed multiplication

In hardware verification, there are decomposed multiplications

For example:

$$
\begin{array}{cccc}
 & 2 & 3 & \\
 & 1 & 3 & * \\
\hline
 & 2*3 & 3*3 & \\
2*1 & 3*1 & + & \\
\hline
\end{array}
$$

Hardware implementation: $(2*1)*10^2+(2*3+3*1)*10^1+(3*3)*10^0$

The above decomposed multiplication is an instance of:
Long multiplication

Long multiplication

- Consider bit-vectors v_1, v_2, v_3, v_4

Long multiplication

concat

- Consider bit-vectors v_1, v_2, v_3, v_4
- Let us apply long multiplication on $v_1 \bullet v_2$ and $v_3 \bullet v_4$.

Long multiplication

concat

- Consider bit-vectors v_1, v_2, v_3, v_4
- Let us apply long multiplication on $v_1 \bullet v_2$ and $v_3 \bullet v_4$.
- We obtain the following partial products.

$$
\begin{array}{cccc}
 & v_1 & v_2 & \\
 & v_3 & v_4 & * \\
\hline
 & v_1 * v_4 & v_2 * v_4 & \\
v_1 * v_3 & v_2 * v_3 & + & \\
\hline
\end{array}
$$

E.g. Long multiplication: SMT formula

```
(set-logic QF_BV)
(declare-fun v1 () (_ BitVec 2))
(declare-fun v2 () (_ BitVec 2))
(declare-fun v3 () (_ BitVec 2))
(declare-fun v4 () (_ BitVec 2))

(define-fun e1 () (_ BitVec 4) (concat #b00 v1))
(define-fun e3 () (_ BitVec 4) (concat #b00 v3))
(define-fun e2 () (_ BitVec 4) (concat #b00 v2))
(define-fun e4 () (_ BitVec 4) (concat #b00 v4))

(assert
(not (=
 (bvmul (concat (concat #b0000 v1) v2)
        (concat (concat #b0000 v3) v4))

 (bvadd  (concat (bvmul e1 e3) #b0000)
 (bvadd  (concat (concat #b00 (bvmul e1 e4)) #b00)
 (bvadd  (concat (concat #b00 (bvmul e2 e3)) #b00)
         (concat #b0000 (bvmul e2 e4)))))
 ))

(check-sat)
(exit)
```

E.g. Long multiplication: SMT formula

```
(set-logic QF_BV)
(declare-fun v1 () (_ BitVec 2))
(declare-fun v2 () (_ BitVec 2))
(declare-fun v3 () (_ BitVec 2))
(declare-fun v4 () (_ BitVec 2))

(define-fun e1 () (_ BitVec 4) (concat #b00 v1))
(define-fun e3 () (_ BitVec 4) (concat #b00 v3))
(define-fun e2 () (_ BitVec 4) (concat #b00 v2))
(define-fun e4 () (_ BitVec 4) (concat #b00 v4))

(assert
(not (=
 (bvmul (concat (concat #b0000 v1) v2)
        (concat (concat #b0000 v3) v4))

 (bvadd  (concat (bvmul e1 e3) #b0000)
 (bvadd  (concat (concat #b00 (bvmul e1 e4)) #b00)
 (bvadd  (concat (concat #b00 (bvmul e2 e3)) #b00)
         (concat #b0000 (bvmul e2 e4)))))
 ))

(check-sat)
(exit)
```

Variable Declarations

E.g. Long multiplication: SMT formula

```
(set-logic QF_BV)
(declare-fun v1 () (_ BitVec 2))
(declare-fun v2 () (_ BitVec 2))
(declare-fun v3 () (_ BitVec 2))
(declare-fun v4 () (_ BitVec 2))

(define-fun e1 () (_ BitVec 4) (concat #b00 v1))
(define-fun e3 () (_ BitVec 4) (concat #b00 v3))
(define-fun e2 () (_ BitVec 4) (concat #b00 v2))
(define-fun e4 () (_ BitVec 4) (concat #b00 v4))

(assert
(not (=
 (bvmul (concat (concat #b0000 v1) v2)
        (concat (concat #b0000 v3) v4))

 (bvadd  (concat (bvmul e1 e3) #b0000)
 (bvadd  (concat (concat #b00 (bvmul e1 e4)) #b00)
 (bvadd  (concat (concat #b00 (bvmul e2 e3)) #b00)
         (concat #b0000 (bvmul e2 e4)))))
 ))

(check-sat)
(exit)
```

Variable Declarations

Zero Extended variables

E.g. Long multiplication: SMT formula

```
(set-logic QF_BV)
(declare-fun v1 () (_ BitVec 2))
(declare-fun v2 () (_ BitVec 2))
(declare-fun v3 () (_ BitVec 2))
(declare-fun v4 () (_ BitVec 2))

(define-fun e1 () (_ BitVec 4) (concat #b00 v1))
(define-fun e3 () (_ BitVec 4) (concat #b00 v3))
(define-fun e2 () (_ BitVec 4) (concat #b00 v2))
(define-fun e4 () (_ BitVec 4) (concat #b00 v4))

(assert
(not (=
 (bvmul (concat (concat #b0000 v1) v2)
        (concat (concat #b0000 v3) v4))

 (bvadd  (concat (bvmul e1 e3) #b0000)
 (bvadd  (concat (concat #b00 (bvmul e1 e4)) #b00)
 (bvadd  (concat (concat #b00 (bvmul e2 e3)) #b00)
         (concat #b0000 (bvmul e2 e4)))))
 ))

(check-sat)
(exit)
```

Variable Declarations

Zero Extended variables

Word-level specification

E.g. Long multiplication: SMT formula

```
(set-logic QF_BV)
(declare-fun v1 () (_ BitVec 2))
(declare-fun v2 () (_ BitVec 2))
(declare-fun v3 () (_ BitVec 2))
(declare-fun v4 () (_ BitVec 2))

(define-fun e1 () (_ BitVec 4) (concat #b00 v1))
(define-fun e3 () (_ BitVec 4) (concat #b00 v3))
(define-fun e2 () (_ BitVec 4) (concat #b00 v2))
(define-fun e4 () (_ BitVec 4) (concat #b00 v4))

(assert
(not (=
 (bvmul (concat (concat #b0000 v1) v2)
        (concat (concat #b0000 v3) v4))

 (bvadd  (concat (bvmul e1 e3) #b0000)
 (bvadd  (concat (concat #b00 (bvmul e1 e4)) #b00)
 (bvadd  (concat (concat #b00 (bvmul e2 e3)) #b00)
         (concat #b0000 (bvmul e2 e4)))))
 ))

(check-sat)
(exit)
```

Variable Declarations

Zero Extended variables

Word-level specification

Long implementation

Our work

- Identifies decomposed multipliers hidden inside a formula

Our work

- Identifies decomposed multipliers hidden inside a formula
- Adds assertions encoding equivalence of the decomposed multiplication and the word-level multiplication

Our work

- Identifies decomposed multipliers hidden inside a formula
- Adds assertions encoding equivalence of the decomposed multiplication and the word-level multiplication

For the previous example, we add the assert:

```
(assert
(=
 (bvmul (concat (concat #b0000 v1) v2)
        (concat (concat #b0000 v3) v4))

 (bvadd  (concat (bvmul e1 e3) #b0000)
 (bvadd  (concat (concat #b00 (bvmul e1 e4)) #b00)
 (bvadd  (concat (concat #b00 (bvmul e2 e3)) #b00)
         (concat #b0000 (bvmul e2 e4)))))
 ))

(check-sat)
(exit)
```

Our work

- Identifies decomposed multipliers hidden inside a formula
- Adds assertions encoding equivalence of the decomposed multiplication and the word-level multiplication

For the previous example, we add the assert:

```
(assert
(=
 (bvmul (concat (concat #b0000 v1) v2)
        (concat (concat #b0000 v3) v4))

 (bvadd  (concat (bvmul e1 e3) #b0000)
 (bvadd  (concat (concat #b00 (bvmul e1 e4)) #b00)
 (bvadd  (concat (concat #b00 (bvmul e2 e3)) #b00)
         (concat #b0000 (bvmul e2 e4)))))
 ))

(check-sat)
(exit)
```

New formula is of the form: $\neg(A = B) \wedge (A = B)$

Non-trivial matching

- Multiple sets of operands are possible for the same term t

Non-trivial matching

- Multiple sets of operands are possible for the same term t
- We developed a backtracking based algorithm to capture all

Non-trivial matching

- Multiple sets of operands are possible for the same term t
- We developed a backtracking based algorithm to capture all
- Matching is equivalent to well known hard problem *integer factorization* with restricted input space

Outline

Algorithm overview

Given formula F, for every term t with root as *bvadd*:
We call $\textsc{MatchLong}(t)$

Algorithm overview

Given formula F, for every term t with root as *bvadd*:
We call $\textsc{MatchLong}(t)$

- Checks whether t conforms to the structure of a SMT formula encoding long multiplication

Algorithm overview

Given formula F, for every term t with root as *bvadd*:
We call $\textsc{MatchLong}(t)$

- Checks whether t conforms to the structure of a SMT formula encoding long multiplication
- If yes, extracts the **partial products**, with proper offsets in Λ

Algorithm overview

Given formula F, for every term t with root as *bvadd*:
We call $\textsc{MatchLong}(t)$

- Checks whether t conforms to the structure of a SMT formula encoding long multiplication
- If yes, extracts the **partial products**, with proper offsets in Λ
- $\textsc{MatchLong}(t)$ calls $\textsc{GetMultOperands}(\Lambda)$

Structure of a SMT formula encoding long multiplication

```
(bvadd  (concat (bvmul e1 e3) #b0000)
        (concat #b00 (bvmul e1 e4) #b00)
        (concat #b00 (bvmul e2 e3) #b00)
        (concat #b0000 (bvmul e2 e4)))
```

- Top level operation: *bvadd*

Structure of a SMT formula encoding long multiplication

```
(bvadd  (concat (bvmul e1 e3) #b0000)
        (concat #b00 (bvmul e1 e4) #b00)
        (concat #b00 (bvmul e2 e3) #b00)
        (concat #b0000 (bvmul e2 e4)))
```

- Top level operation: *bvadd*
- Each *bvadd* argument is a *concat* operation

Structure of a SMT formula encoding long multiplication

```
(bvadd  (concat (bvmul e1 e3) #b0000)
        (concat #b00 (bvmul e1 e4) #b00)
        (concat #b00 (bvmul e2 e3) #b00)
        (concat #b0000 (bvmul e2 e4)))
```

- Top level operation: *bvadd*
- Each *bvadd* argument is a *concat* operation
- Each *concat* argument is either a **partial product** or a string of **zeroes**

Algorithm: MatchLong(t)

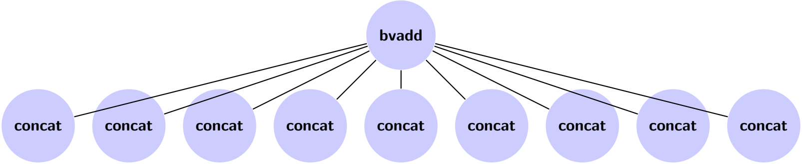
On input t of the form:

```
(bvadd  (concat (bvmul e1 e4) #b00000000)
        (concat #b00 (bvmul e1 e5) #b000000)
        (concat #b00 (bvmul e2 e4) #b000000)
        (concat #b0000 (bvmul e1 e6) #b0000)
        (concat #b0000 (bvmul e2 e5) #b0000)
        (concat #b0000 (bvmul e3 e4) #b0000)
        (concat #b000000 (bvmul e2 e6) #b00)
        (concat #b000000 (bvmul e3 e5) #b00)
        (concat #b00000000 (bvmul e3 e6))
)
```

Algorithm: MatchLong(t)

On input t of the form:

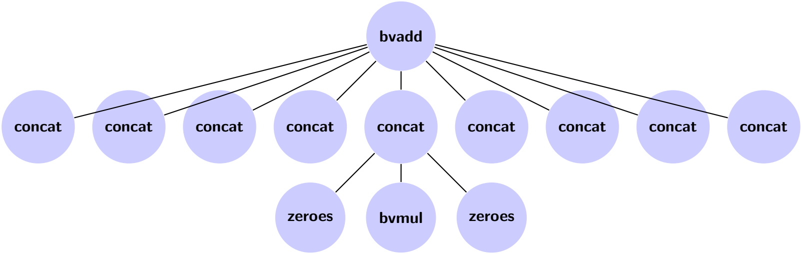
```
(bvadd  (concat (bvmul e1 e4) #b00000000)
        (concat #b00 (bvmul e1 e5) #b000000)
        (concat #b00 (bvmul e2 e4) #b000000)
        (concat #b0000 (bvmul e1 e6) #b0000)
        (concat #b0000 (bvmul e2 e5) #b0000)
        (concat #b0000 (bvmul e3 e4) #b0000)
        (concat #b000000 (bvmul e2 e6) #b00)
        (concat #b000000 (bvmul e3 e5) #b00)
        (concat #b00000000 (bvmul e3 e6))
)
```



Algorithm: MatchLong(t)

On input t of the form:

```
(bvadd  (concat (bvmul e1 e4) #b00000000)
        (concat #b00 (bvmul e1 e5) #b000000)
        (concat #b00 (bvmul e2 e4) #b000000)
        (concat #b0000 (bvmul e1 e6) #b0000)
        (concat #b0000 (bvmul e2 e5) #b0000)
        (concat #b0000 (bvmul e3 e4) #b0000)
        (concat #b000000 (bvmul e2 e6) #b00)
        (concat #b000000 (bvmul e3 e5) #b00)
        (concat #b00000000 (bvmul e3 e6))
)
```



Algorithm: MATCHLONG(t)

On input t of the form:

```
(bvadd  (concat (bvmul e1 e4) #b00000000)
        (concat #b00 (bvmul e1 e5) #b000000)
        (concat #b00 (bvmul e2 e4) #b000000)
        (concat #b0000 (bvmul e1 e6) #b0000)
        (concat #b0000 (bvmul e2 e5) #b0000)
        (concat #b0000 (bvmul e3 e4) #b0000)
        (concat #b000000 (bvmul e2 e6) #b00)
        (concat #b000000 (bvmul e3 e5) #b00)
        (concat #b00000000 (bvmul e3 e6))
)
```

$e_i = 00 \bullet v_i$

- Output: Λ

Offset	8	6	4	2	0
			$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
		$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
	$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
	Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

Algorithm: MatchLong(t)

On input t of the form:

```
(bvadd  (concat (bvmul e1 e4) #b00000000)
        (concat #b00 (bvmul e1 e5) #b000000)
        (concat #b00 (bvmul e2 e4) #b000000)
        (concat #b0000 (bvmul e1 e6) #b0000)
        (concat #b0000 (bvmul e2 e5) #b0000)
        (concat #b0000 (bvmul e3 e4) #b0000)
        (concat #b000000 (bvmul e2 e6) #b00)
        (concat #b000000 (bvmul e3 e5) #b00)
        (concat #b00000000 (bvmul e3 e6))
)
```

$e_i = 00 \bullet v_i$

- Output: Λ

Offset	8	6	4	2	0
			$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
		$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
	$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
	Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- A call to GetMultOperands(Λ) is made.

$\textsc{GetMultOperands}(\Lambda)$

- Checks if Λ can be matched to an instance of long multiplication

GetMultOperands(Λ)

- Checks if Λ can be matched to an instance of long multiplication
- If yes, extract the **operands** and add **assertion** stating **equality** decomposed and word-level multiplication

Input: Λ - the **partial products**, with proper offsets

		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

GETMULTOPERANDS(Λ)

Input: Λ - the **partial products**, with proper offsets

		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

Output: x and y: the two multiplication operands

Example run

Current Λ index:

		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x:
- Current y:

Known partial products removed

Example run

Current Λ index: 4

Λ_4	Λ_3	Λ_2	Λ_1	Λ_0
		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		

- Current x:
- Current y:

Known partial products removed

Example run

Current Λ index: 4

		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1
- Current y: v_4

Known partial products removed

Example run

Current Λ index: 3

Λ_4	Λ_3	Λ_2	Λ_1	Λ_0
		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		

- Current x: v_1
- Current y: v_4

Known partial products removed

Example run

Current Λ index: 3

		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_2
- Current y: v_4 v_5

Known partial products removed

Example run

Current Λ index: 2

		$v_3\,v_4$	$v_2\,v_6$	$v_3\,v_6$
	$v_2\,v_4$	$v_2\,v_5$	$v_3\,v_8$	
$v_1\,v_4$	$v_1\,v_5$	$v_1\,v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_2
- Current y: v_4 v_5

Known partial products removed

Example run

Current Λ index: 2

		$v_3\,v_4$	$v_2\,v_6$	$v_3\,v_6$
	$v_2\,v_4$	$v_2\,v_5$	$v_3\,v_8$	
$v_1\,v_4$	$v_1\,v_5$	$v_1\,v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_2
- Current y: v_4 v_5

Known partial products removed

Example run

Current Λ index: 2

		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_2 v_3
- Current y: v_4 v_5 v_6

Known partial products removed

Example run

Current Λ index: 1

		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_2 v_3
- Current y: v_4 v_5 v_6

Known partial products removed

Example run

Current Λ index: 1

		$v_3\, v_4$	$v_2\, v_6$	$v_3\, v_6$
	$v_2\, v_4$	$v_2\, v_5$	$v_3\, v_8$	
$v_1\, v_4$	$v_1\, v_5$	$v_1\, v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_2 v_3
- Current y: v_4 v_5 v_6

Known partial products removed

Example run

Current Λ index: 0

		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_2 v_3
- Current y: v_4 v_5 v_6

Known partial products removed

Example run

Current Λ index: 0

		$v_3 v_4$	$v_2 v_6$	$v_3 v_6$
	$v_2 v_4$	$v_2 v_5$	$v_3 v_8$	
$v_1 v_4$	$v_1 v_5$	$v_1 v_6$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: $v_1 \; v_2 \; v_3$
- Current y: $v_4 \; v_5 \; v_6$

Known partial products removed

Need for backtracking

- The algorithm may not work always

Need for backtracking

- The algorithm may not work always
- Our modification: backtracking based algorithm

Need for backtracking

- The algorithm may not work always
- Our modification: backtracking based algorithm
- We look at one such example

Example run: Backtracking

Current Λ index:

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x:
- Current y:
- Backtrack allowed:

Example run: Backtracking

Current Λ index: 4

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x:
- Current y:
- Backtrack allowed:

Example run: Backtracking

Current Λ index: 4

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1
- Current y: v_3
- Backtrack allowed: *false*

Example run: Backtracking

Current Λ index: 3

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1
- Current y: v_3
- Backtrack allowed: *false*

Example run: Backtracking

Current Λ index: 3

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 0
- Current y: v_3 v_3
- Backtrack allowed: *false* *true*

Example run: Backtracking

Current Λ index: 2

			$v_1\,v_3$	$v_2\,v_3$
		$v_1\,v_3$		
$v_1\,v_3$	$v_1\,v_3$	$v_2\,v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 0
- Current y: v_3 v_3
- Backtrack allowed: *false* *true*

Example run: Backtracking

Current Λ index: 2

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 0
- Current y: v_3 v_3
- Backtrack allowed: *false* *true*

Example run: Backtracking

Current Λ index: 2

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 0 v_2
- Current y: v_3 v_3 v_3
- Backtrack allowed: *false* *true* *false*

Example run: Backtracking

Current Λ index: 1

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 0 v_2
- Current y: v_3 v_3 v_3
- Backtrack allowed: *false* *true* *false*

Example run: Backtracking

Current Λ index: 1

			$v_1\,v_3$	$v_2\,v_3$
		$v_1\,v_3$		
$v_1\,v_3$	$v_1\,v_3$	$v_2\,v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 0 v_2
- Current y: v_3 v_3 v_3
- Backtrack allowed: *false* *true* *false*

Conflict! Backtrack to last index with backtrack set to true!

Example run: Backtracking

Current Λ index: 3

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1
- Current y: v_3
- Backtrack allowed: *false*

Example run: Backtracking

Current Λ index: 3

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_1
- Current y: v_3 0
- Backtrack allowed: *false* *false*

Example run: Backtracking

Current Λ index: 2

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_1
- Current y: v_3 0
- Backtrack allowed: *false* *false*

Example run: Backtracking

Current Λ index: 2

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_1
- Current y: v_3 0
- Backtrack allowed: *false* *false*

Example run: Backtracking

Current Λ index: 2

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_1 v_2
- Current y: v_3 0 v_3
- Backtrack allowed: *false* *false* *false*

Example run: Backtracking

Current Λ index: 1

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_1 v_2
- Current y: v_3 0 v_3
- Backtrack allowed: *false* *false* *false*

Example run: Backtracking

Current Λ index: 1

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_1 v_2
- Current y: v_3 0 v_3
- Backtrack allowed: *false* *false* *false* *false*

Example run: Backtracking

Current Λ index: 0

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_1 v_2
- Current y: v_3 0 v_3
- Backtrack allowed: *false* *false* *false* *false*

Example run: Backtracking

Current Λ index: 0

			$v_1 v_3$	$v_2 v_3$
		$v_1 v_3$		
$v_1 v_3$	$v_1 v_3$	$v_2 v_3$		
Λ_4	Λ_3	Λ_2	Λ_1	Λ_0

- Current x: v_1 v_1 v_2
- Current y: v_3 0 v_3
- Backtrack allowed: *false* *false* *false* *false* *false*

Outline

Implementation

- Implemented in bit-vector rewrite module of $\mathrm{Z3}$ SMT solver

Implementation

- Implemented in bit-vector rewrite module of $\mathrm{Z3}$ SMT solver
- Various preliminary checks for early exit during identification

Implementation

- Implemented in bit-vector rewrite module of $\mathrm{Z3}$ SMT solver
- Various preliminary checks for early exit during identification
- New formula generated after adding assertions

Implementation

- Implemented in bit-vector rewrite module of $\mathrm{Z3}$ SMT solver
- Various preliminary checks for early exit during identification
- New formula generated after adding assertions
- Can be solved using any solver

Implementation

- Implemented in bit-vector rewrite module of Z3 SMT solver
- Various preliminary checks for early exit during identification
- New formula generated after adding assertions
- Can be solved using any solver
- We compare our tool with Z3, Boolector, CVC4

Benchmarks

- Our experiments include 20 benchmarks

Benchmarks

- Our experiments include 20 benchmarks
- Benchmarks check equivalence of specification and implementation

Benchmarks

- Our experiments include 20 benchmarks
- Benchmarks check equivalence of specification and implementation
- Benchmarks generated by varying:
  - Total bit length of the input bit-vectors
  - Width of each block
  - Assigning specific blocks as equal or setting them to zero

Benchmarks

- Our experiments include 20 benchmarks
- Benchmarks check equivalence of specification and implementation
- Benchmarks generated by varying:
  - Total bit length of the input bit-vectors
  - Width of each block
  - Assigning specific blocks as equal or setting them to zero
- SystemVerilog to SMT formula:
  - Benchmarks written in SystemVerilog
  - Fed to steword, a hardware verification tool
  - steword: Converts SystemVerilog design to SMT formula

Results

Table: Multiplication experiments. Times are in seconds. Bold denotes minimum time.

	SMTSolver			OurSolver			
Benchmark	Z3	Boolector	CVC4	Z3	Boolector	CVC4	Portfolio
base	184.3	42.2	16.54	0.53	43.5	**0.01**	0.01
ex1	2.99	0.7	0.36	0.33	0.8	**0.01**	0.01
ex1_sc	t/o	t/o	t/o	1.75	t/o	**0.01**	0.01
ex2	0.78	0.2	0.08	0.44	0.3	**0.01**	0.01
ex2_sc	t/o	1718	2826	3.15	1519	**0.01**	0.01
ex3	1.38	0.3	0.08	0.46	0.7	**0.01**	0.01
ex3_sc	t/o	1068	t/o	3.45	313.2	**0.01**	0.01
ex4	0.46	0.2	0.03	0.82	0.2	**0.01**	0.01
ex4_sc	287.3	62.8	42.36	303.6	12.8	**0.01**	0.01
sv_assy	t/o	t/o	t/o	0.07	t/o	**0.01**	0.01
mot_base	t/o	t/o	t/o	13.03	1005	**0.01**	0.01
mot_ex1	t/o	t/o	t/o	1581	13.8	**0.01**	0.01
mot_ex2	t/o	t/o	t/o	2231	13.7	**0.01**	0.01
wal_4bit	0.09	0.05	**0.02**	0.09	0.1	0.04	0.02
wal_6bit	2.86	0.6	0.85	**0.28**	0.8	14.36	0.28
wal_8bit	209.8	54.6	225.1	**0.59**	30.0	3471	0.59
wal_10bit	t/o	1523	t/o	**1.03**	98.6	t/o	1.03
wal_12bit	t/o	t/o	t/o	**1.55**	182.3	t/o	1.55
wal_14bit	t/o	t/o	t/o	**2.27**	228.5	t/o	2.27
wal_16bit	t/o	t/o	t/o	**2.95**	481.7	t/o	2.95

Summary

Equivalence of word-level and decomposed multiplication:

- Leading SMT solvers failed to identify word-level reasoning
- Bit-blasting lead to time blow up
- Proposed a new pre-processing heuristic
- Implemented as part of $\mathrm{Z3}$ bit-vector rewriting module
- Added assertions to the input formula
- Reduction in solving time
- Implemented similar pattern matching algorithm for Wallace tree multiplier

Thank you!