Functional Verification of Arithmetic Circuits

Maciej Ciesielski

Department of Electrical & Computer Engineering
University of Massachusetts, Amherst
ciesiel@ecs.umass.edu

Outline

- Introduction
  - Hardware verification methods, focus on **arithmetic verification**
- Basics
  - Boolean techniques: BDD
  - Word-level canonical: BMD, TED
  - Equivalence checking, SAT
- Bit-vector and word-level techniques
  - SMT, ILP models
- Computer algebra methods
  - Arithmetic bit level
  - Data-flow based approach
  - Other algebraic methods
- Extended bibliography

Arithmetic Verification
Part I

Basics

Canonical Diagrams, SAT

Hardware Verification

- Variety of formal techniques
  - Model checking, property checking
  - Equivalence checking
  - Theorem proving
- Solution methods
  - Canonical diagrams (Boolean, word-level)
  - SAT (satisfiability)
  - SMT (satisfiability modulo theories)
  - Integer Linear Programming (ILP) methods
  - Computer Algebra approach

Formal Verification Techniques

- Theorem proving,
  - Deductive reasoning with axioms, rules to prove correctness
  - Term-rewriting, no guarantee it will terminate
  - Complex, heavy user interaction and domain knowledge
  - Systems: ACL, PVS, HOL,
- Model checking
  - Automatic technique to prove correctness of concurrent systems
  - Use temporal logic specification, CTL, etc. to describe properties
  - Practical tools become available, popular in industry
- Equivalence checking
  - Check if two designs are equivalent
  - Solved for combinational circuits
    - Except arithmetic circuits and datapaths
    - Difficult problem for sequential systems
- Functional verification (our focus: arithmetic circuits)
  - Special case of equivalence checking and property checking

Functional Verification

- Determined by *functional specification*
  - Input-output (I/O) relationship
  - Our focus: combinational integer arithmetic circuits
- How is functional specification given?
  - By writing a formula that describes I/O relationship
    - Easy for logic circuits (write a Boolean formula)
    - What about arithmetic circuits?
    - Different ways to provide “specification”
  - By providing reference design with desired function
    - e.g. standard “text-book” multiplier
    - Checking equivalence with the reference design

Combinational Equivalence Checking

- Functional Approach
  - Transform output functions of combinational circuits into a unique (*canonical*) representation
  - Two circuits are equivalent if their representations are identical
  - Efficient canonical representations:
    - BDD, BMD, TED.
- Structural
  - Identify structurally similar internal points
  - Prove internal points (cut-points) equivalent

Canonical Representations

- Boolean Representations ($f: B \rightarrow B$)
  - BDDs, ZBDDs, etc.
- Moment Diagrams ($f: B \rightarrow Z$)
  - BMDs, K*BMDs, etc.
  - Canonical DAGs for Polynomials ($f: Z \rightarrow Z$)
  - Taylor Expansion Diagrams (TEDs)
  - Horner Decision Diagrams (HDDs)
- Arithmetic verification needs representation for $f: Z_{2^m} \rightarrow Z_{2^m}$
  - Modular arithmetic

Binary Decision Diagrams (BDD)

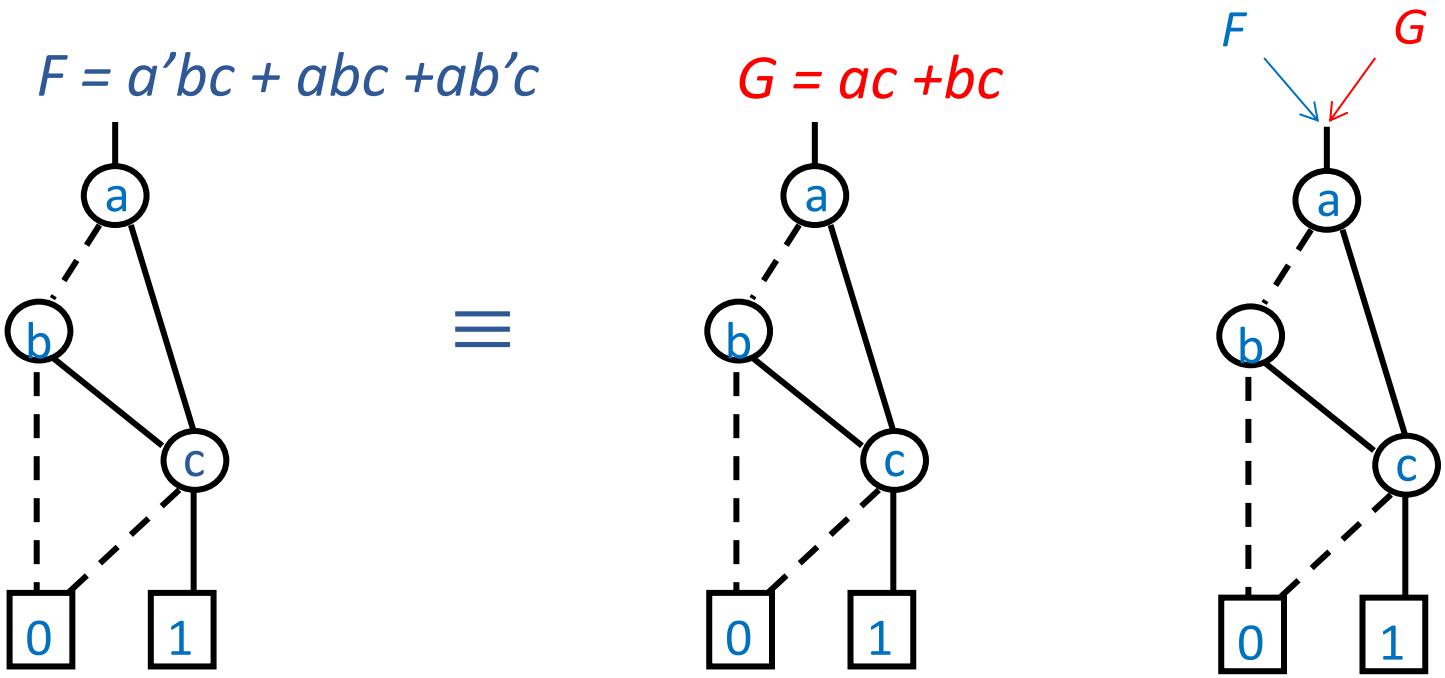
- Based on recursive Shannon expansion [Bryant DAC'85]

$$f = x f_x + x' f_{x'}$$

- Compact data structure for Boolean logic
  - can represents sets of objects (states) encoded as Boolean functions
- Canonical representation
  - Reduced, ordered BDDs (ROBDD) are canonical
  - Essential for verification
    - Equivalence checking
    - SAT

Application to Verification - EC

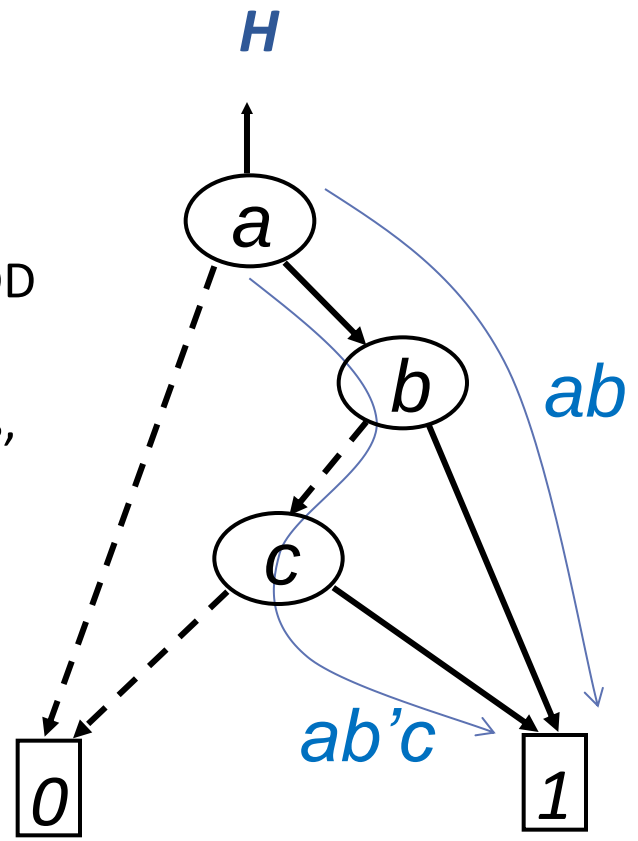
- Equivalence Checking (EC) of *combinational* circuits
- *Canonicity* property of BDDs:
  - if *F* and *G* are equivalent, their BDDs are identical (for the *same ordering* of variables)



Application to Verification - SAT

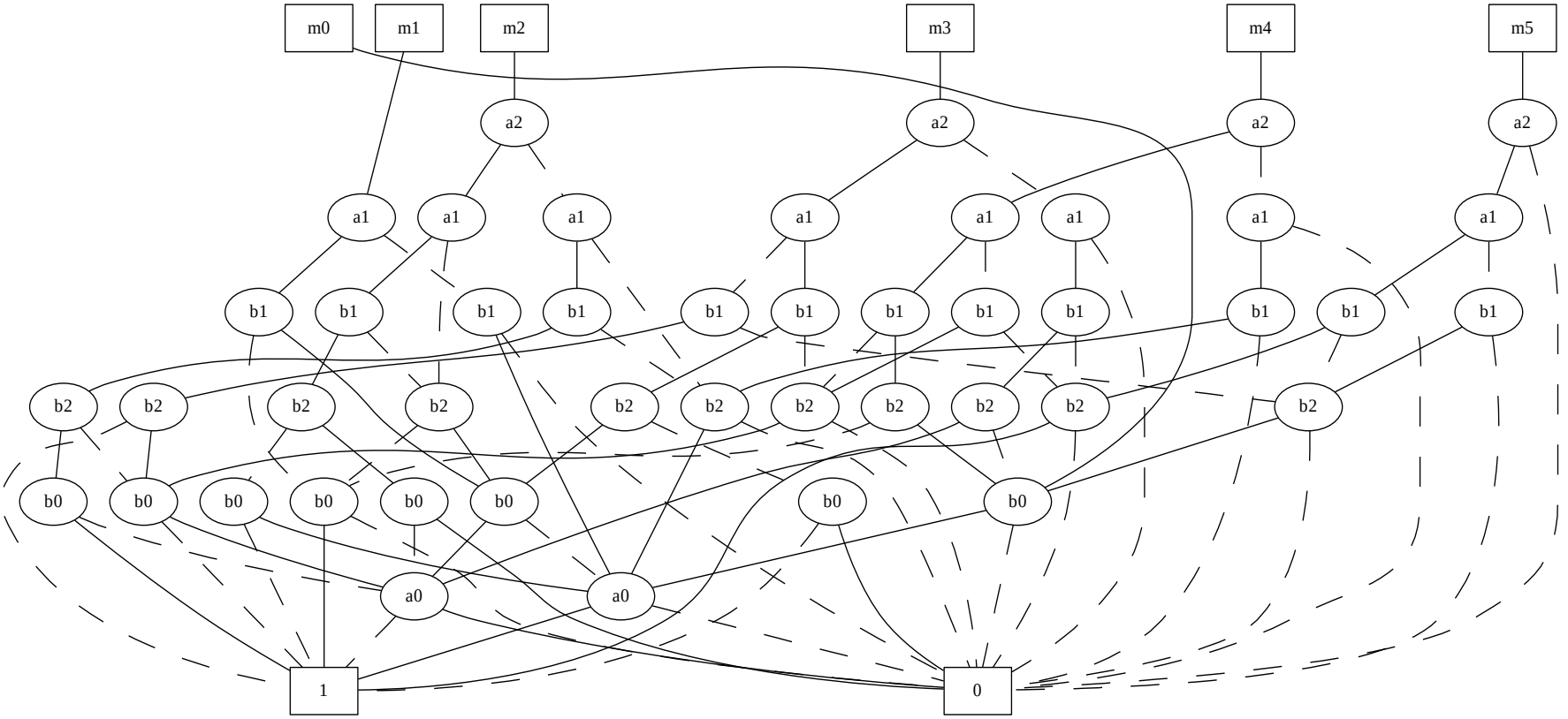
- General SAT
  - Find a set of satisfying assignments
- Functional test generation
  - SAT, Boolean *satisfiability* analysis
  - to test for *H* = 1 (0), find a path in the BDD to terminal **1** (0)
  - the path, expressed in function variables, gives a satisfying solution (test vector)

Problem: size explosion



Large BDDs

- Maps: $B \rightarrow B$, very low-grain
  - Can be prohibitively large for arithmetic circuits (*multipliers*, etc.)



Partitioned BDDs

- Circuits for which BDD can be constructed
  - Represent multiple-output circuits as shared BDDs
  - BDDs must be identical (with same variable order)

- Circuits whose BDDs are too large
  - Cannot construct BDDs, memory problem
  - Use partitioned BDD method
    - decompose circuit into smaller pieces, each as BDD
    - check equivalence of internal points (*cut-point* method)

Word-level Canonical Diagrams - BMD

- **BMD** for 4-bit Multiplier (bit-level) [Bryant TCAD'95]
  - Map: *B → Z (binary to integers)*
- Devised for word-level operations, arithmetic designs
- Based on modified Shannon expansion (*positive Davio*)

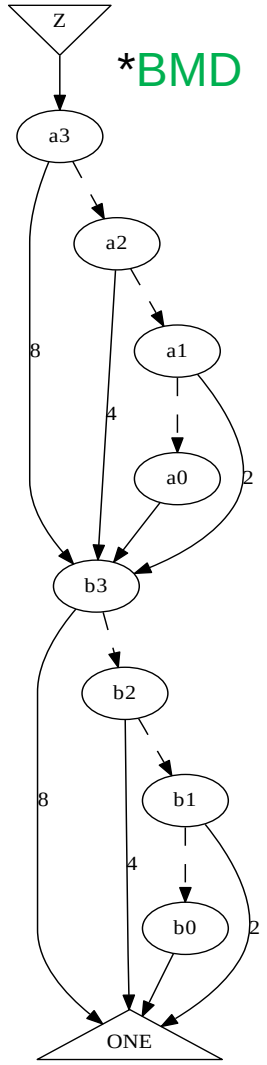
$$f = x f_x + x' f_{x'} = x f_x + (1-x) f_{x'}$$

$$= f_{x'} + x (f_x - f_{x'}) = f_{x'} + x f_{\Delta x}$$

where $f_{x'} = f_{x=0}$ is *zero moment*

$f_{\Delta x} = (f_x - f_{x'})$ is *first moment,* first derivative

- Additive and multiplicative weights on edges (*BMD)



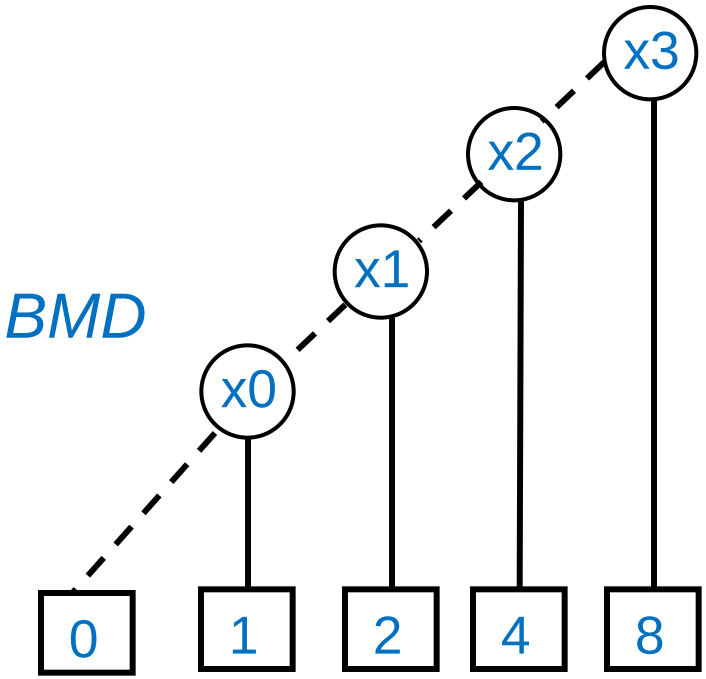
*BMD - Construction

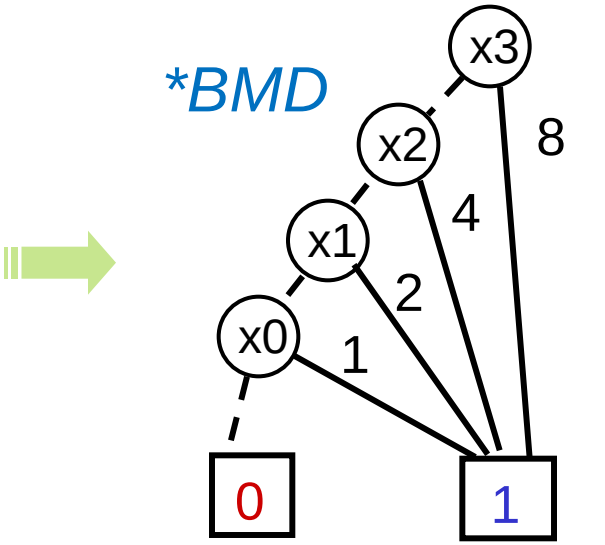
- Unsigned integer: $X = 8x_3 + 4x_2 + 2x_1 + x_0$

$X_{x3=1} = 8 + 4x_2 + 2x_1 + x_0$

$X_{x3=0} = 4x_2 + 2x_1 + x_0$

$X_{\Delta x3} = 8$

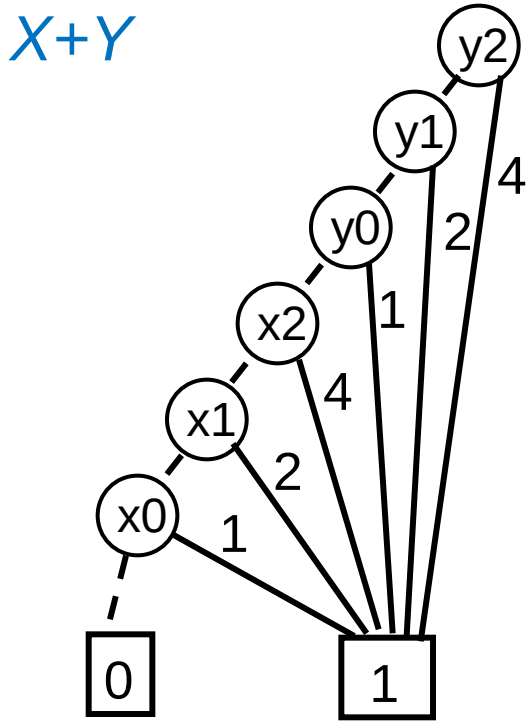


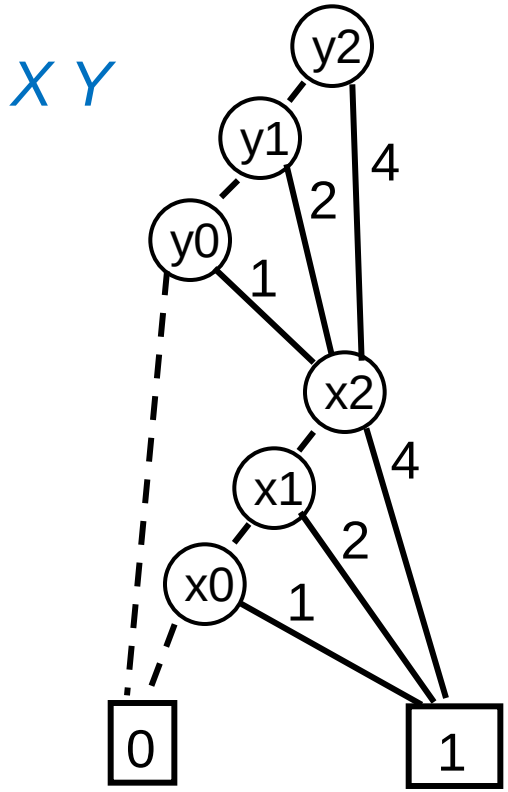


Multiplicative edges

*BMD – Word-Level Representation

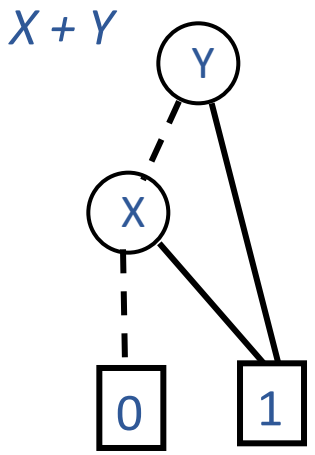
- Efficiently modeling symbolic word-level operators

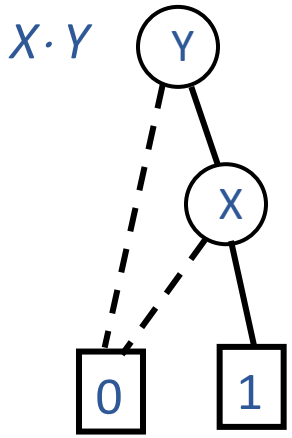




Taylor Expansion Diagram (TED)

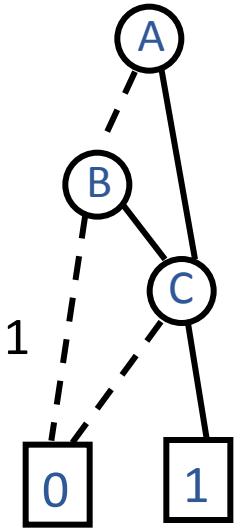
- Canonical representation of multi-variate polynomials of arbitrary degree [Ciesielski-TComp'06]
  - *f*: *Integer* $\rightarrow$ *Integer*
  - More *word-level* than BMD
  - When input are Boolean: TED $\rightarrow$ BMD
- TED is *not* a decision diagram
  - Cannot solve SAT
  - Too high-grain
  - Cannot express output bits as function of word-level inputs



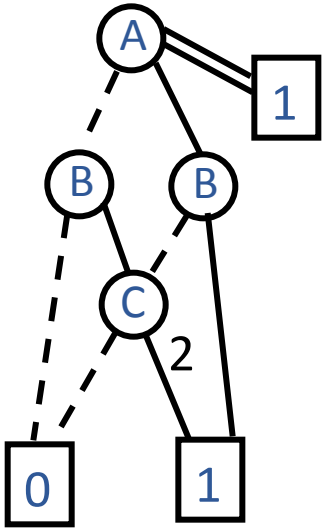


TED – a few Examples

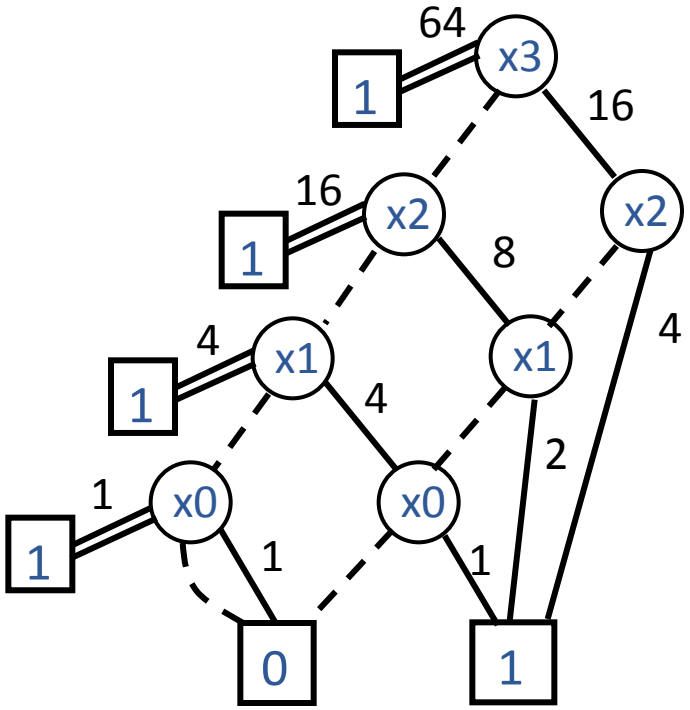
AC+BC +1
= A(B+C)+1



A^2+AB +2BC



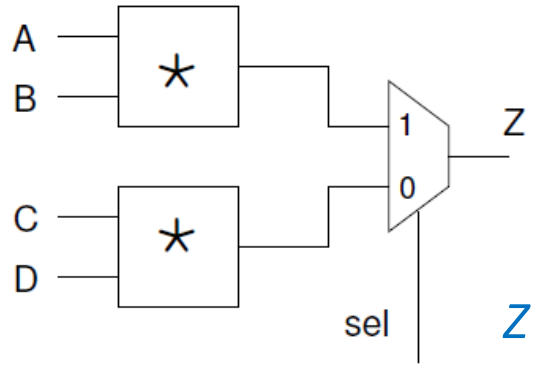
$$X^2 = (8x_3 + 4x_2 + 2x_1 + x_0)^2$$



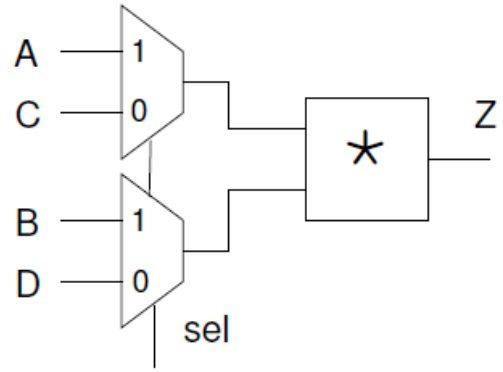
Useful for finding factored forms

TED – Application to EC

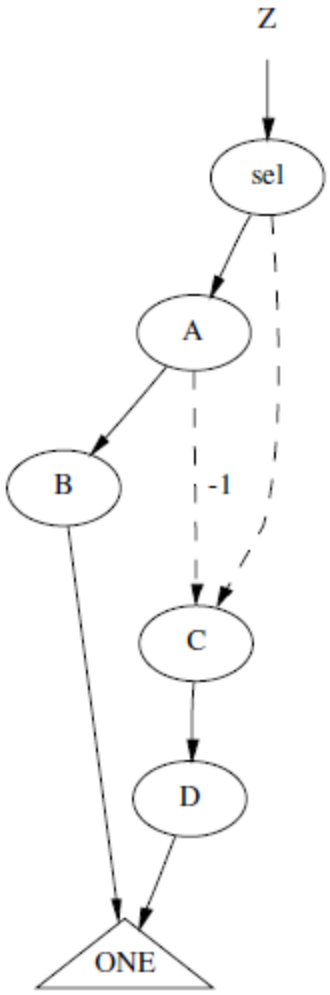
- Resource sharing
  - TED can prove their equivalence



$Z = sel(A*B) + (1-sel)(C*D)$



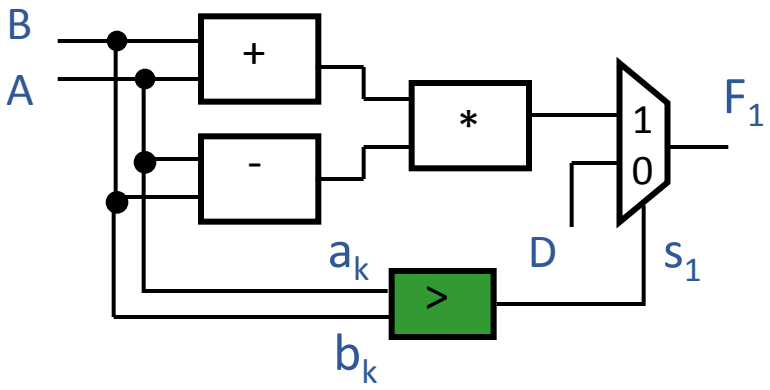
$= sel(A*B - C*D) + CD$

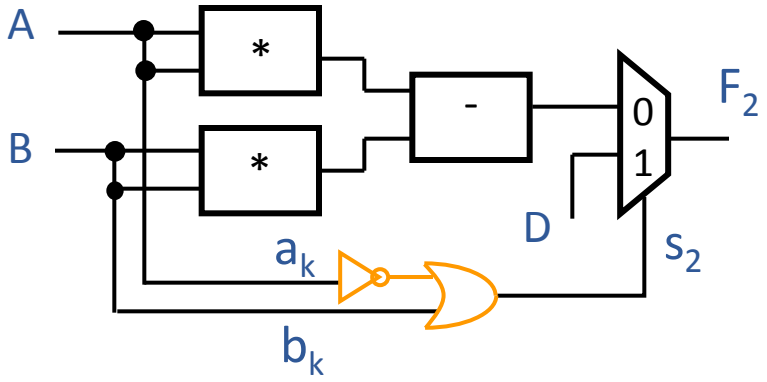


Applications to RTL Verification

- Equivalence checking with TEDs
  - word-level and Boolean variables

$A = [a_{n-1}, \ldots, a_k, \ldots, a_0] = [A_{hi}, a_k, A_{lo}]$, $B = [b_{n-1}, \ldots, b_k, \ldots, b_0] = [B_{hi}, b_k, B_{lo}]$





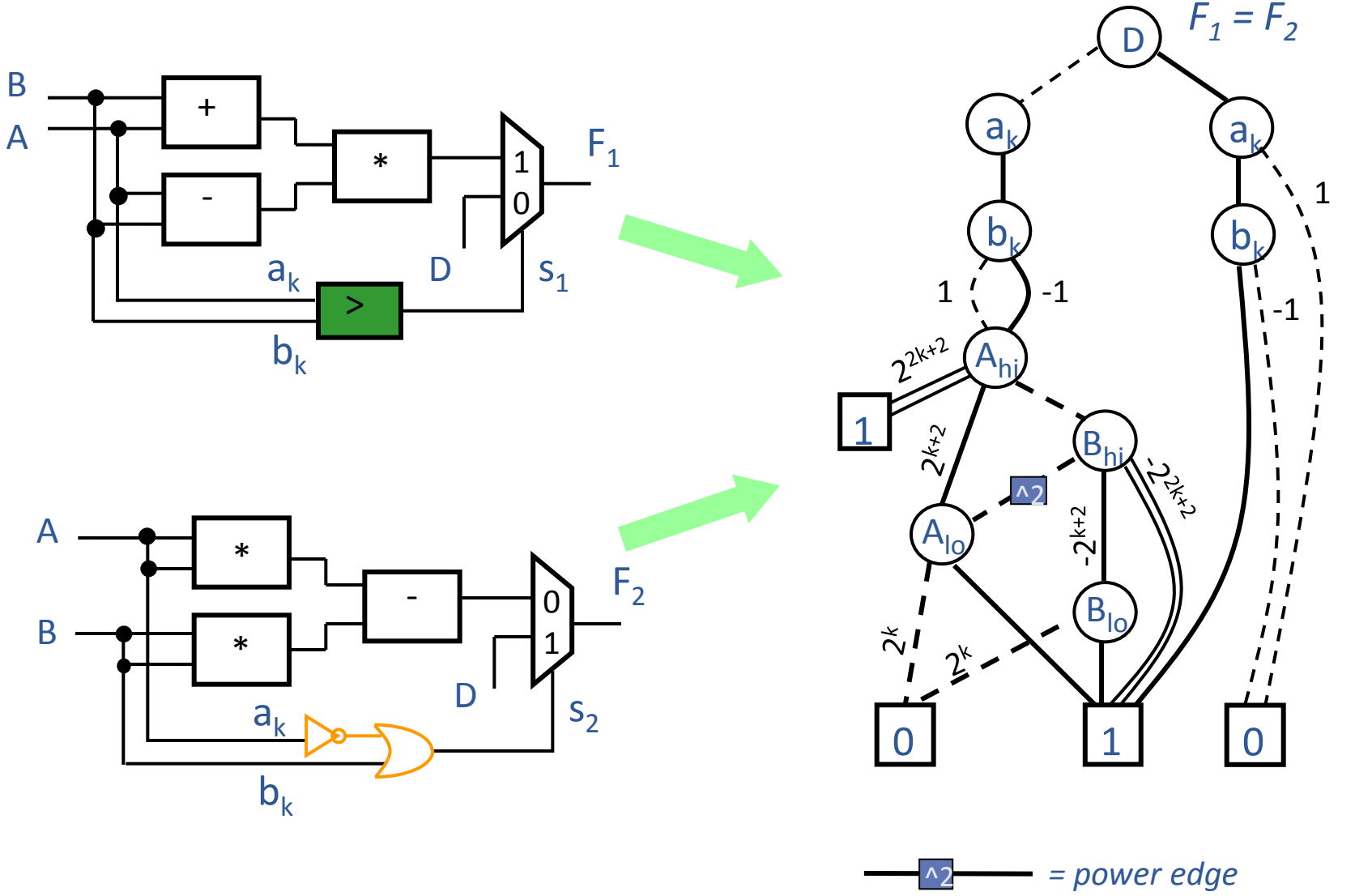
$F_1 = s_1(A+B)(A-B) + (1-s_1)D$

$s_1 = (a_k > b_k) = a_k\,(1-b_k)$

$F_2 = (1-s_2)\,(A^2-B^2) + s_2\,D$

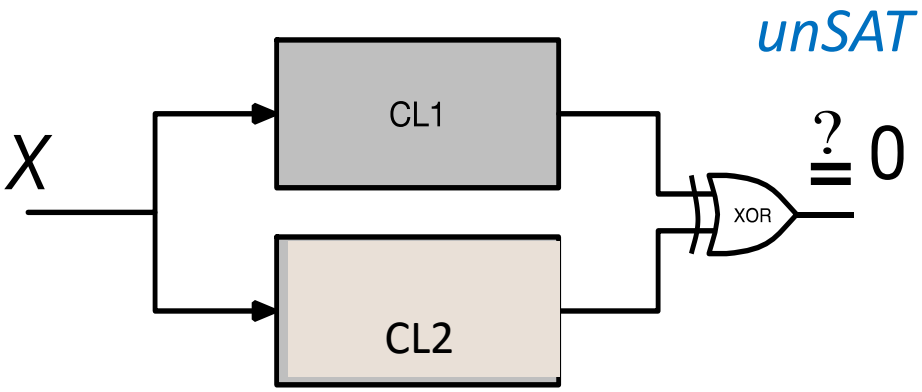
$s_2 = a_k' \vee b_k = 1 - a_k + a_k\,b_k$

RTL Equivalence Checking



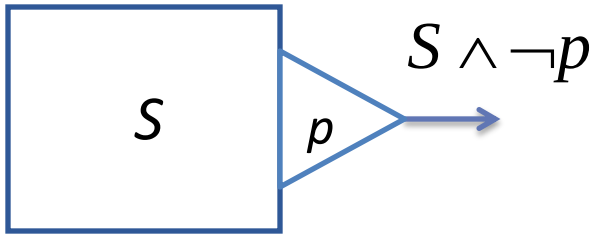
Equivalence Checking with SAT

- Equivalence checking using SAT [GRASP, zChaff, MiniSAT]
  - Create a "miter" at the outputs
  - Check for unSAT (if *always* evaluates to 0)
  - The most popular way to solve equivalence checking (EC)



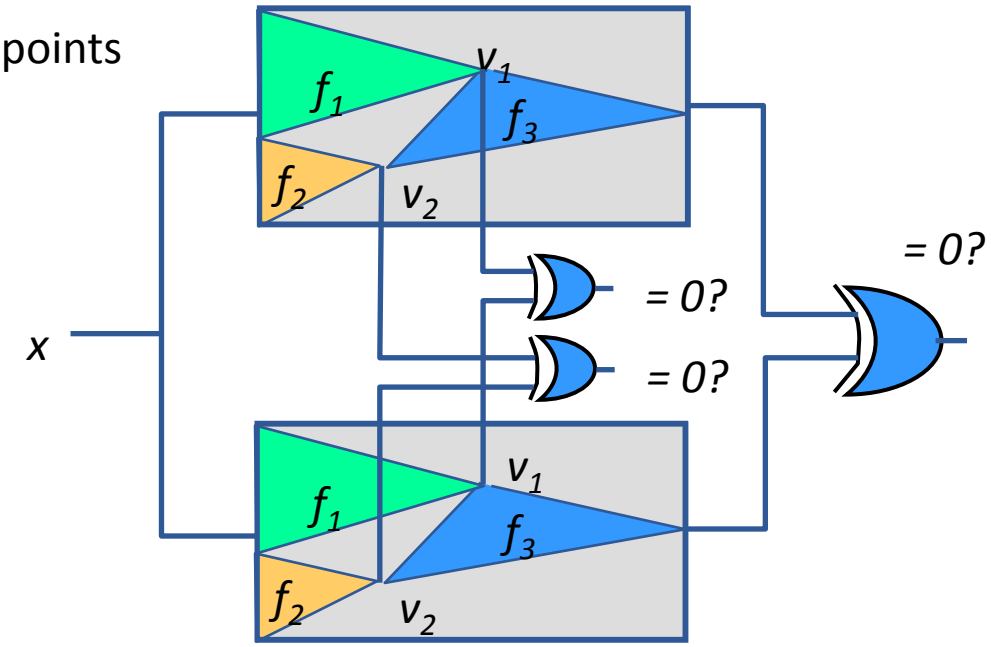
Property Checking using SAT

- Same concept can be applied to property checking
  - Need to conjunct the system spec (*S*) with the complement of the property (*p*)
  - Invoke a SAT solver
    - unSAT if system *S* satisfies property *p*



Miter for Cut-point based EC

- Use *cut-points* to partition the Miter
- Use SAT to solve the problem: is the output of Miter unSAT ?
- Cut-point guessing
  - Compute signature with random simulation
  - Sort signatures + select cut-points
  - Iteratively verify and refine cut-points
  - Verify outputs



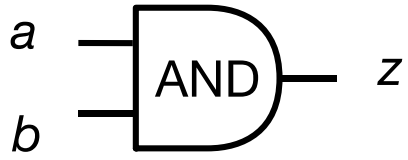
Boolean Satisfiability (SAT)

- Well known *Constraint Satisfaction* Problem.
- Given a propositional formula Ψ, determine if there exist a variable assignment such as Ψ evaluates to true.
  - If it exist, Ψ is called *satisfiable*
  - If not, Ψ is called *unsatifiable*
  - SAT problems are hard (NP complete)
- Most SAT solvers uses Conjunctive Normal Form (CNF) to represent the propositional formula
  - Conjunction of clauses
  - Each clause is a disjunction of literals

$$(a + b + \overline{c})(\overline{a} + c)(a + \overline{b} + c)$$

CNF for Gate-level Circuits

- Converting gate-level circuit into CNF formula



$$(a \vee \bar{z}) \wedge (b \vee \bar{z}) \wedge (\bar{a} \vee \bar{b} \vee z)$$

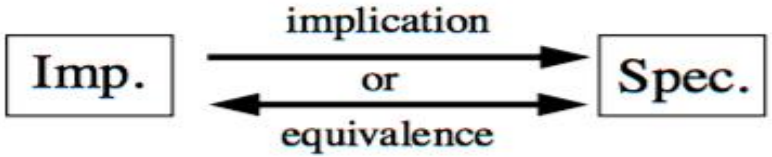
$$CNF = (a + \bar{d})(b + \bar{d})(\bar{a} + \bar{b} + d)(c + g)(d + g)(\bar{c} + \bar{d} + \bar{g})$$
$$(\bar{g} + f)(\bar{e} + f)(g + e + \bar{f})$$

Theorem Provers

Introduction

Theorem Proving

Prove that an implementation satisfies a specification by mathematical reasoning



Implementation and specification expressed as *formulas* in *a formal logic*

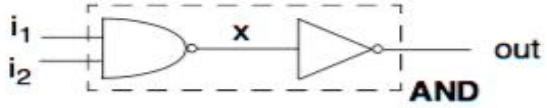
Required relationship (logical equivalence/logical implication) described as *a theorem* to be proven within the context of a proof calculus

A proof system:

A set of axioms and inference rules (simplification, rewriting, induction, etc.)

Theorem Prover: Example

Example 1: Logic AND

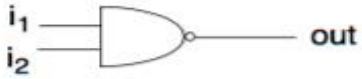


AND Specification:

AND_SPEC $(i_1, i_2, out) := out = i_1 \wedge i_2$

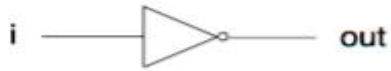
NAND specification:

NAND $(i_1, i_2, out) := out = \neg(i_1 \wedge i_2)$



NOT specification:

NOT $(i, out) := out = \neg i$



AND Implementation:

AND_IMPL $(i_1, i_2, out) := \exists x.$ NAND $(i_1, i_2, x) \wedge$ NOT (x, out)

Theorem Prover: Example

Logic AND (cont'd)

Proof Goal:

$\forall\, i_1, i_2, out.\ AND\_IMPL(i_1,i_2,out) \Rightarrow ANDSPEC(i_1,i_2,out)$

Proof (forward)

$AND\_IMP(i_1,i_2,out)$ {from above circuit diagram}

$\vdash \exists x.NAND\ (i_1,i_2,x) \wedge NOT\ (x,out)$ {by def. of AND impl}

$\vdash NAND\ (i_1,i_2,x) \wedge NOT(x,out)$ {strip off "$\exists x$."}

$\vdash NAND\ (i_1,i_2,x)$ {left conjunct of line 3}

$\vdash x = \neg(i_1 \wedge i_2)$ {by def. of NAND}

$\vdash NOT\ (x,out)$ {right conjunct of line 3}

$\vdash out = \neg x$ {by def. of NOT}

$\vdash out = \neg(\neg(i_1 \wedge i_2)$ {substitution, line 5 into 7}

$\vdash out = (i_1 \wedge i_2)$ {simplify, $\neg\neg\ t = t$}

$\vdash AND\ (i_1,i_2,out)$ {by def. of AND spec}

$\vdash AND\_IMPL\ (i_1,i_2,out) \Rightarrow AND\_SPEC\ (i_1,i_2,out)$

Q.E.D.

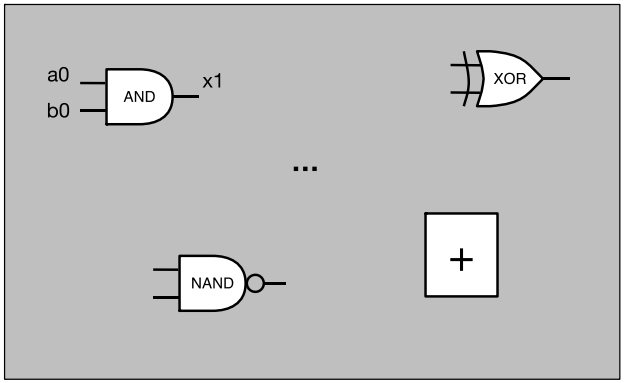
Arithmetic Verification
Part II

Word-level Models

SMT, ILP methods

SMT for bit-vector Operation

- Bit-vector operations
  - variables extended to bit-vectors



SMT-LIB

$$\left\{\begin{array}{l} assert\ (bvadd\ (x_1)\ (not\ (a_0 b_0))\ ) = \#b000...0 \\ \ldots\ \ldots \\ assert\ (\ldots\ \ldots) = \#b000...0 \end{array}\right\}\ gate\ netlist$$

← $X_1 = AND\ (a_0\ ,\ b_0)$
$X_1 - a_0 b_0 = 0$

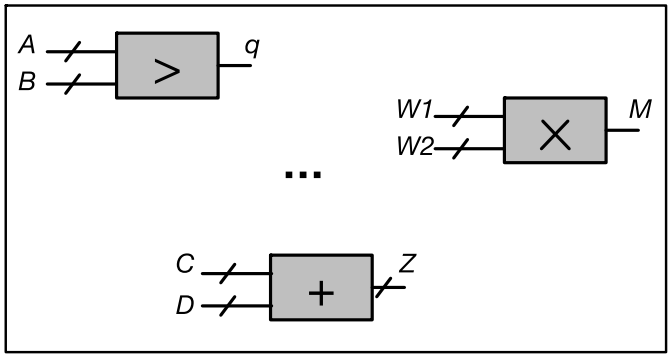
$$assert(xor(a_0,b_0) \vee xor(a_1,b_1) \vee \ldots xor(a_n,b_n)\ ) = 0$$ ← $A = B$

$$assert(xor(a_0,b_0) \vee xor(a_1,b_1) \vee \ldots xor(a_n,b_n)\ ) = 1$$ ← $A \neq B$

$$A = \sum_{i=0}^{i=n-1} 2_i a_i,\ B = \sum_{i=0}^{i=n-1} 2_i b_i\ ,\ \ldots$$

SMT for bit-vector Operation

- Bit-vector operation
  - variables extended to bit-vectors



SMT-LIB

$$assert(bvadd(\sum_{i=0}^{i=n} 2^i \cdot z_i \ (not(\sum_{i=0}^{i=n-1} 2_i a_i))(not(\sum_{i=0}^{i=n-1} 2_i b_i))\ ) = \#b000...0$$ ← Z – (A+B)

$$assert(bvadd(M(not\ (bvmul\ (\sum_{i=0}^{i=n-1} 2_i w1_i)(\sum_{i=0}^{i=n-1} 2_i w2_i))\ ) = \#b000...0$$ ← M – (A*B)

$$assert(\ (A < B) \wedge (q = 0)\ ) \vee (\ (A > B) \wedge (q = 1)\ ))$$ ← A > B

$$Z = \sum_{i=0}^{i=n} 2^i z_i,\ A = \sum_{i=0}^{i=n-1} 2_i a_i,\ B = \sum_{i=0}^{i=n-1} 2_i b_i\ ,\ ...$$

SMT vs SAT

- SMT extends SAT solving by adding extensions (theories)
  - *Core* theory (Boolean)
  - *Ints* theory
  - *Reals* theory
  - *Reals_Ints* theory
  - *ArraysEx* theory
  - *Fixed_Size_BitVectors* theory, etc.
- Properties
  - Decidable: An effective procedure exists to check if a formula is a member of a theory *T*
  - Often *Quantifier-free*
- An SMT solver can solve a SAT problem, but not vice-versa
- Application:
  - Software Model Checking, RTL Design, Analog

DPLL(T)

- In a nutshell

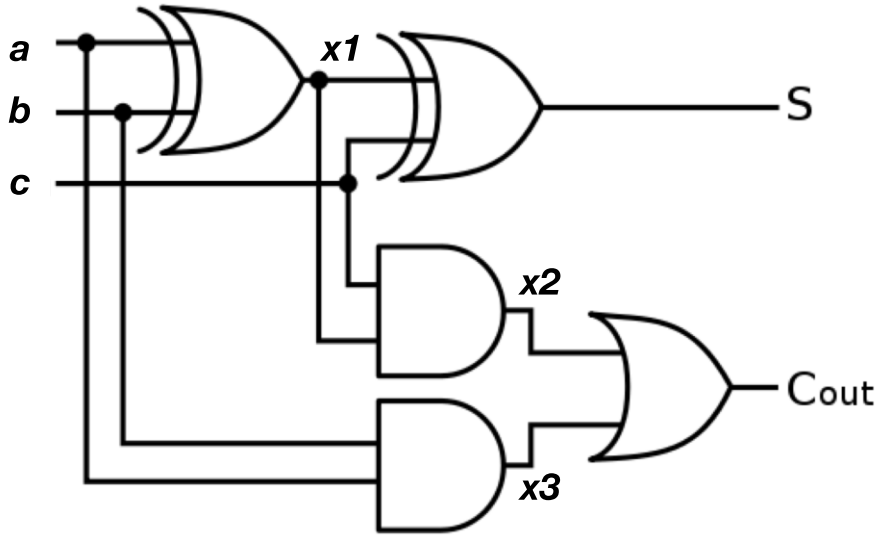
$$DPLL(T) = DPLL(X) + Tsolver$$

  - DPLL(X)
    - Very similar to a SAT solver, enumerates Boolean model
    - Not allowed: pure literal, blocked clause elimination, ...
    - Required: incremental addition of clauses
    - Desirable: partial model detection
  - T-Solver
    - Checks consistency of conjunctions of literals
    - Computes theory propagations
    - Produces explanations of inconsistency / T-propagation
    - Should be incremental and backtractable

SMT modeling without Reference

- Full adder using SMT: $a+b+c = 2C_{out}+S$
  - {+, -, 2} are bit-vector operations

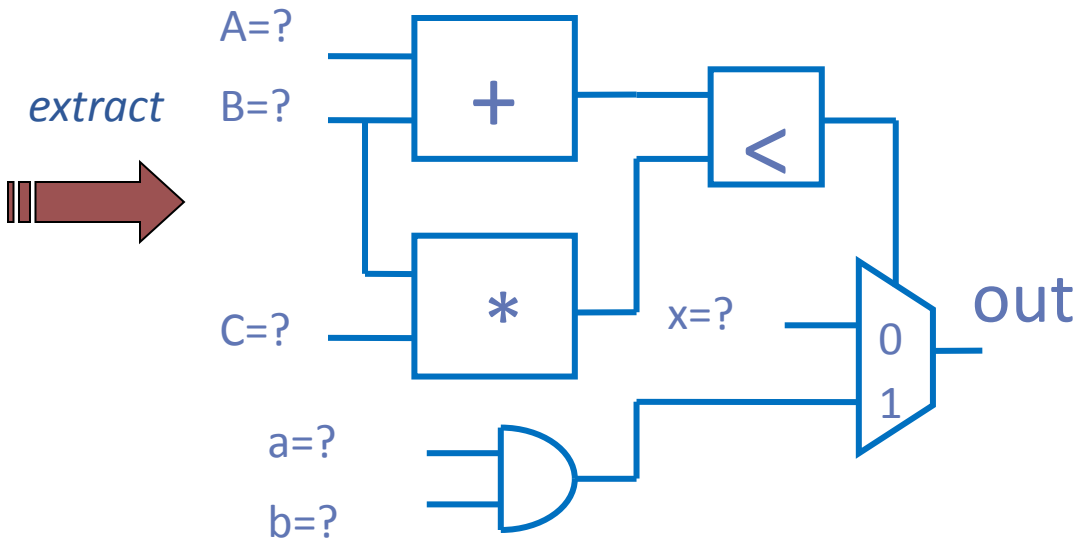
$$(a+b-2ab-x_1)=0 \quad \text{XOR}$$
$$\wedge\ (x_1+c-2x_1c-S)=0 \quad \text{XOR}$$
$$\wedge\ (x_1c-x_2)=0 \quad \text{AND}$$
$$\wedge\ (ab-x_3)=0 \quad \text{AND}$$
$$\wedge\ (2C_{out}+S-a-b-c)=0 \quad F_{spec}$$



Functional Test Generation

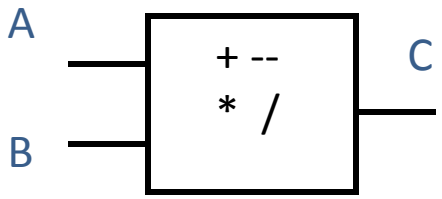
- Deterministic test pattern generation
  - Formulate a SAT problem for a complex combinational design
  - Solve SAT: find a set of satisfying assignment

```
modulo example(
  A, B, C, x, a, b, out
);
input [?:0] A,B,C;
input a,b,x;
output out;
assign sel = (A+B)<(B*C);
assign out = (sel)? (a&b) : x;
endmodule
```

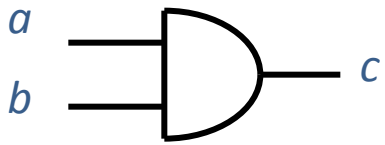


Types of Operators

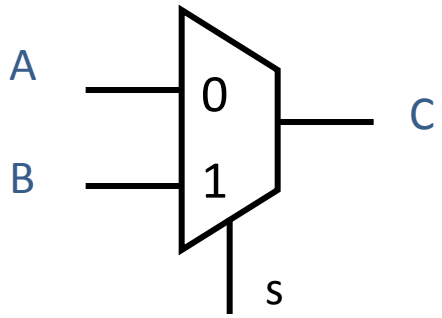
- Arithmetic blocks
  (symbolic, word-level operators)
  - ADD, SUB
  - MULT, DIV



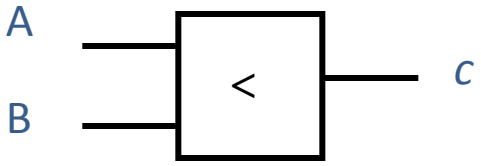
- Boolean logic (bit-level)
  - logic gates



- Mixed-level blocks
  – MUX



  - comparators
  - shifters, etc

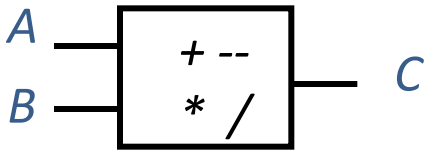


Modeling Arithmetic Datapaths

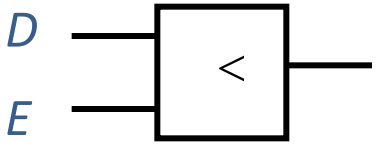
- Map entire design to CNF (miniSAT, GRASP, zCHAFF,…)
  - Any generic CNF-based solver can be used
  - Representation is large, structural information is lost
- Map Boolean logic onto CNF, arithmetic operators onto linear equations (HSAT)
  - Inconsistent domains, explicit backtracking needed
- Represent both domains in a unified format (LPSAT)
  - Solve Mixed Integer Linear Program (MILP)
  - Scalable with design size
  - Constraint propagation *implicitly* passed to MILP solver
- First, assume infinite precision
  - No overflows, arbitrarily large bit-width
- ATPG is also used

Arithmetic and Mixed Operators

n = number of bits

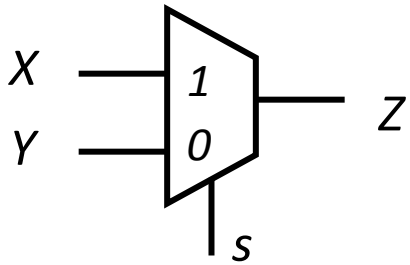


$$\begin{cases} C = A + B \\ A, B \leq 2^n - 1 \end{cases}$$



$s = \{0,1\}$

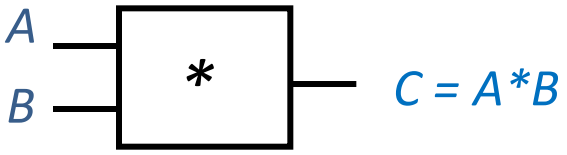
$$\begin{cases} D - E - L(1-s) < 0 \\ D - E + L\,s \geq 0 \end{cases} \qquad D, E \leq 2^n - 1$$

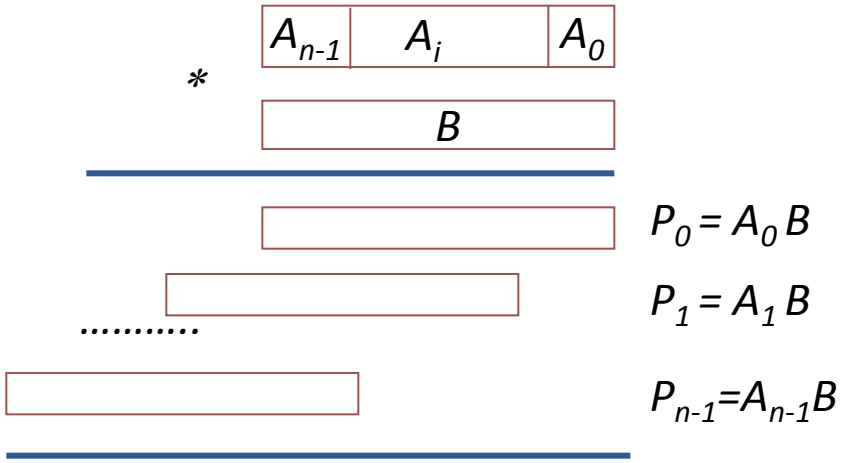


$X, Y \leq 2^n - 1$ $\qquad s = \{0,1\}$

$$\begin{cases} Z - X - L(1-s) \leq 0 \\ X - Z - L(1-s) \leq 0 \\ Z - Y - L\,s \leq 0 \\ Y - Z - L\,s \leq 0 \end{cases} \qquad L = 2^n - 1$$

LP: Linearizing the Multiplier





- Expand operand A

  $A = A_0 + 2\,A_1 + \ldots + 2^{n-1}\,A_{n-1}$

- Keep operand *B* as one variable
- Represent result in terms of partial products P_i

$$C = P_0 + 2\,P_1 + \ldots + 2^{n-1}\,P_{n-1}$$

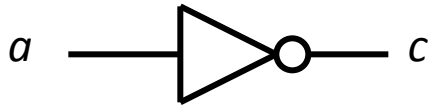
for $i = 1, \ldots, n-1$:

$$\begin{cases} P_i - L\,A_i \le 0 \\ P_i - B + L(1-A_i) \ge 0 \\ 0 \le P_i \le B \end{cases}$$

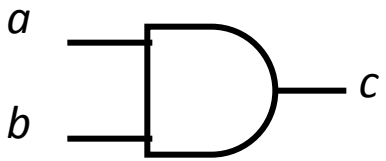
where $L = 2^n - 1$

LP Modeling of Boolean Logic

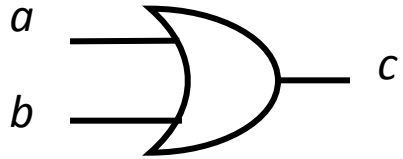
$a, b, c = \{0, 1\}$



$$c = 1 - a$$



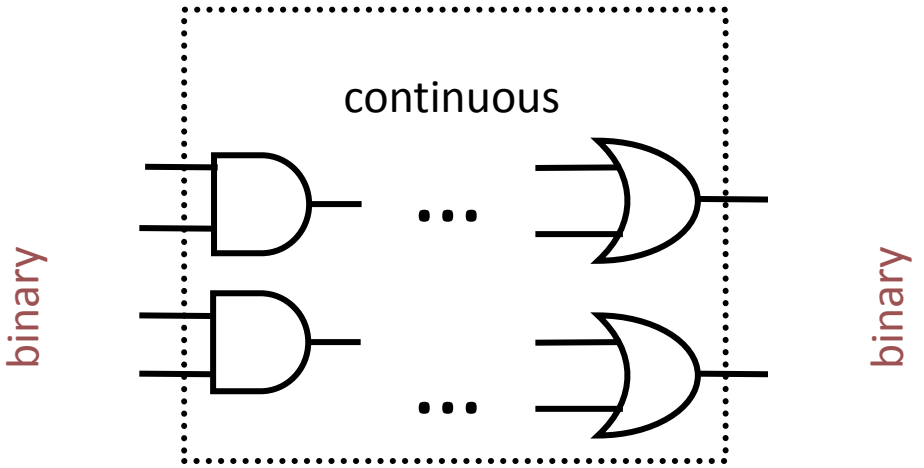
$$\begin{cases} c \leq a \\ c \leq b \\ c \geq a+b-1 \\ c \geq 0 \end{cases}$$



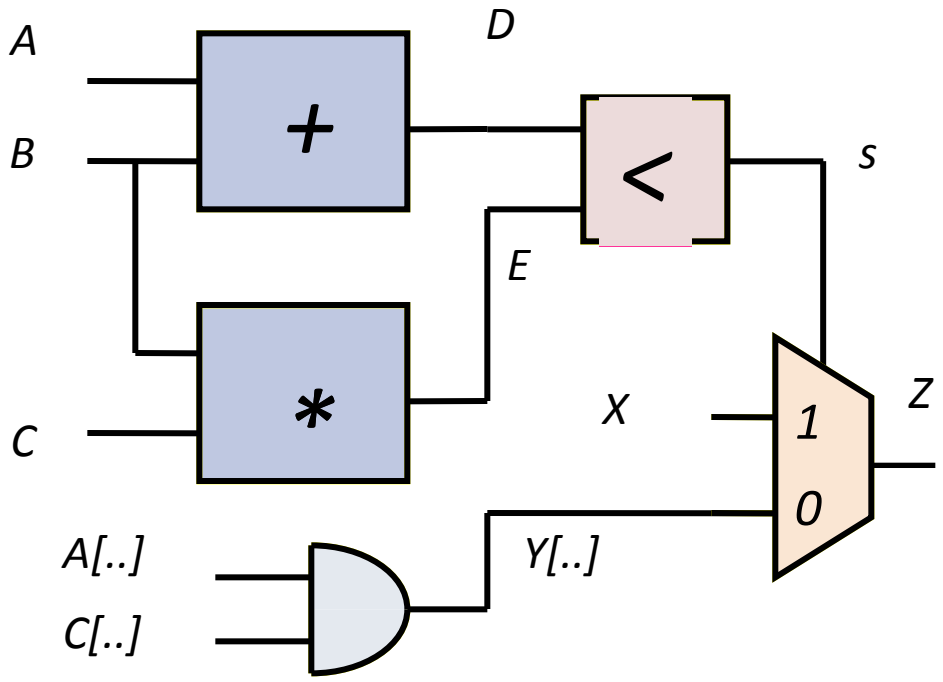
$$\begin{cases} c \geq a \\ c \geq b \\ c \leq a + b \\ c \leq 1 \end{cases}$$

MILP Solvers - Efficiency Issues

- Efficiency depends on the number of integer variables
  - Only IO signals defined as *binary* variables
  - All internal signals left as *continuous,* automatically adjusted to integer
- Implicit branch & bound, backtracking
  - Impose ordering of variables to branch on
  - Put decision variables first



LPSAT - Example



$D = A + B$

$E = B * C$ (linearized)

$0 \leq A,B,C \leq 2^n-1$

$D - E - L(1-s) < 0 \quad D - E + L\,s \geq 0$

$Z - X - L(1-s) \leq 0$

$X - Z - L(1-s) \leq 0$

$Z - Y - L\,s \leq 0$

$Y - Z - L\,s \leq 0$

A[..] , C[..] ,Y[..] = bit *vectors*

s = *decision* variable (0,1)

A,B,C,D,E,X,Z = *continuous* variables

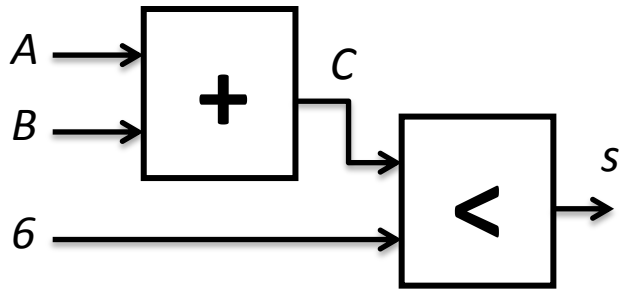
$Y[k] \leq A[k],\ C[k]$

$Y[k] \geq A[k] + C[k] - 1$

$Y[k] \geq 0$

Improving ILP Modeling

- ILP model for RTL design

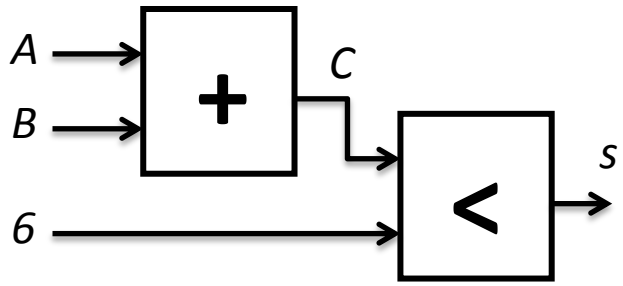


$$A + B - C = 0$$
$$C - 6 - 2^n \cdot (1 - s) \leq -1$$
$$C - 6 + 2^n s \geq 0$$
$$A \leq 7$$
$$B \leq 7$$
$$s \leq 1$$

- Solve the problem for *A=5, B=3* in *n*=3 bits
  - ILP solution (incorrect):
    - C = 8; s = (8 < 6) = 0
  - Correct solution in 3 bits:
    - C = 8 mod 2^3 = 0; s = (0 < 6) = 1
  - Note: adding constraint C ≤ 7 will make it infeasible
  - Need to properly model modulo semantics
    - Add a slack variable δ to adjust the variable size

Modeling Modulo Semantic

- Correctly models modulo semantics [Brinkman VLSI'02]



$$A+B-C=0$$

$$C-6-2^n(1-s)\leq -1$$

$$C-6+2^n s\geq 0$$

$$A\leq 7$$

$$B\leq 7$$

$$s=\{0,1\}$$

$$A_{[n]}:\{A\leq 2^n-1$$

$$-A_{[n]}:\begin{cases}A\leq 2^n-1\\ -2^n+A\end{cases}$$

$$A+B:\begin{cases}A+B-2^n\cdot\delta\\ A+B-2^n\cdot\delta\leq 2^n-1\\ \delta=\{0,1\}\end{cases}$$

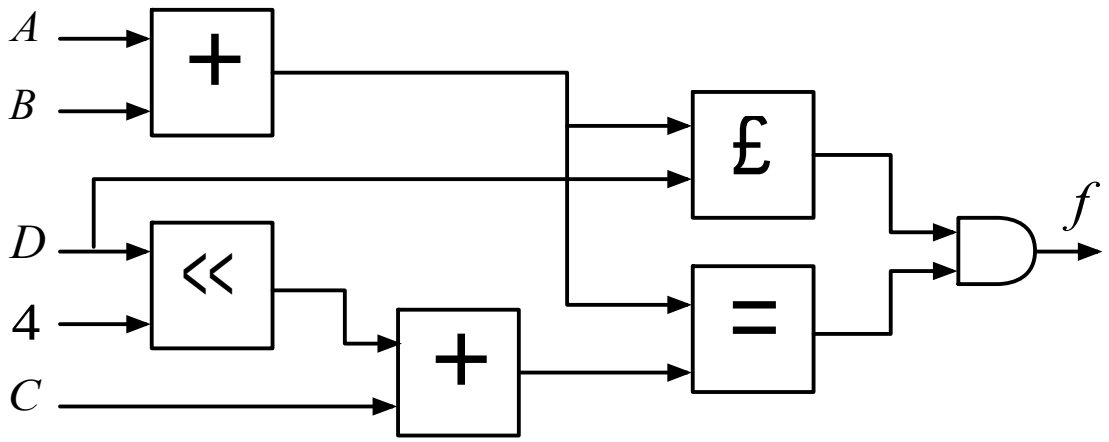
$$A\circ B:\{2^n\cdot A+B$$

Now the solution is *correct*:

C = 0; δ = 1; s = (0 < 6) = 1

ILP Model for RTL Design

- Complete RTL model



$$t_1 = t_2 \wedge t_1 \le D \wedge t_1 = A + B \wedge t_2 = C + t_3 \wedge t_3 = D << 4$$

$$A: \quad A \le 2^n - 1$$

$$A = \sum_{i=0}^{n-1} 2^i \cdot a^i \qquad -A = -2^{n-1} \cdot a_{n-1} + \sum_{i=0}^{i=n-2} 2_i \cdot a_i$$

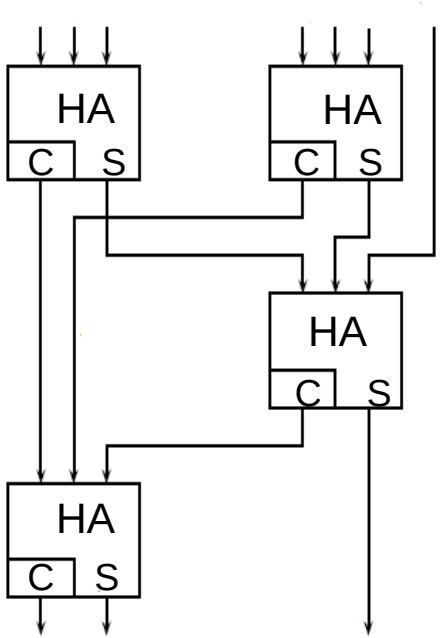
$$A + B = \sum_{i=0}^{i=n-1} 2_i \cdot a_i + \sum_{i=0}^{i=n-1} 2_i \cdot b_i$$

Arithmetic Verification
Part III
Algebraic Approach

Arithmetic Verification

- Functional verification of arithmetic circuits
  - Verify function implemented by arithmetic circuit
  - Use algebraic approach

- Why is it important
  - Arithmetic circuits are difficult to verify on bit-level
  - Avoid “bit blasting” (flattening to bit-level)

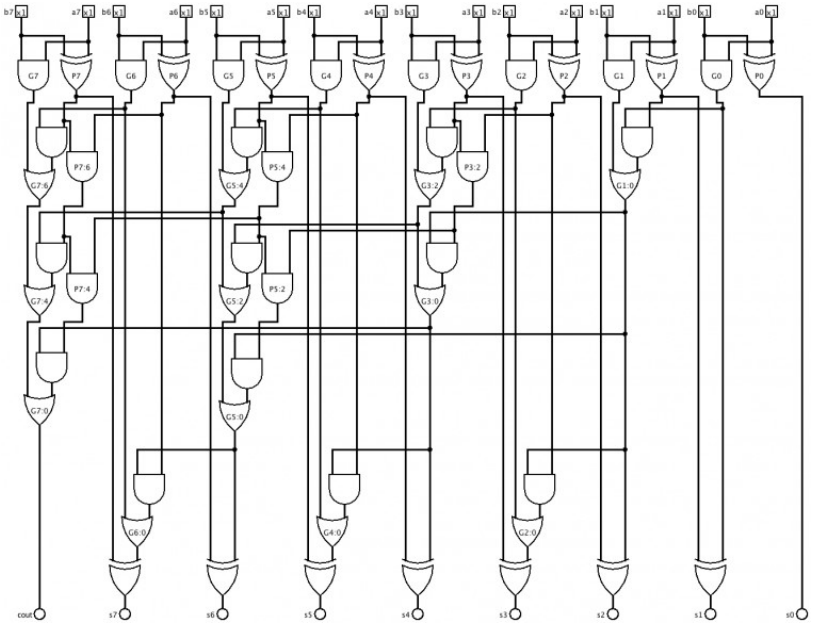


Arithmetic Verification

- We should be able to answer questions:
  - Does the circuit meet the specification ?
  - *What function* does this circuit implement ?
  - If it does not meet the specification
    - Demonstrate error, show bug trace
    - Find the bug (debugging, difficult)

- How to approach it:

  with **Computer Algebra**



Computer Algebra Methods

- Arithmetic Bit-level (ABL) representation

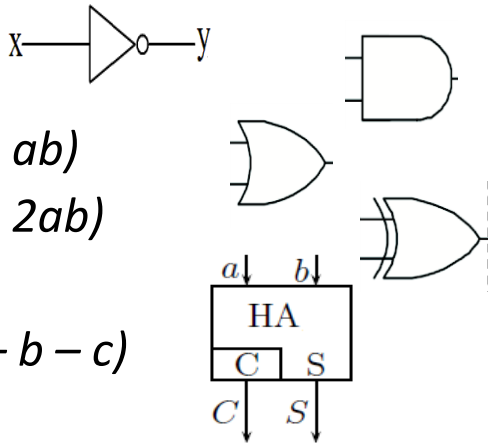
  [Wienand'08, Pavlenko'11]

- Also applied to Galois Fields (GF) [Kalla'14, Tcomp'15]
  - Circuit specification F_{spec} and implementation B represented by polynomials
  - Check if implementation B satisfies specification F_{spec}
  - Done by reducing F_{spec} modulo B

- Methods differ in ways they accomplish the reduction

Computer Algebra - Basics

- Represent specification and implementation as polynomials, F_{spec} and B in Z_2^n.
- Example: multiplier $Z = X*Y$
  - $F_{spec} = Z - X * Y$
  - B = set of polynomials in Z_2^n representing circuit elements:
    - $y = INV(x)$: $(y + 1 - x)$
    - $q_1 = AND(a, b)$: $(q_1 - a\,b)$
    - $q_2 = OR(a, b)$: $(q_2 - a - b + ab)$
    - $q_3 = XOR(a, b)$: $(q_3 - a - b + 2ab)$
    - $\{C,S\} = HA(a, b)$: $(2C + S - a - b)$
    - $\{C,S\} = FA(a, b, c)$: $(2C + S - a - b - c)$



  - Each satisfies the local function if *poly = 0*

Computer Algebra - reduction

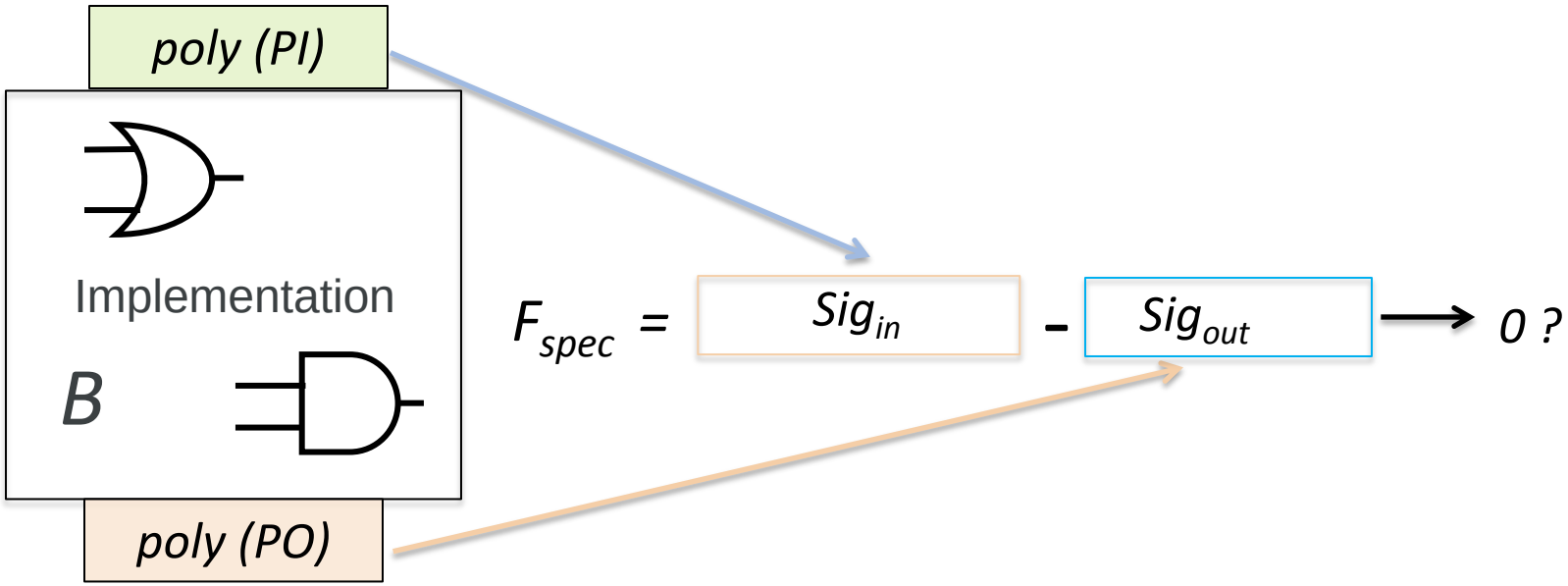
- Goal: reduce F_{spec} modulo B:

$$F_{spec} \xrightarrow{B}_{+} r$$

- Systematic methods exist to perform this reduction
  - If $r = 0$, the circuit is correct
  - If $r \neq 0$, circuit *may* still be correct but B needs to be a canonical basis (*Groebner basis*) to determine if $r = 0$
  - Groebner basis
    - Difficult to compute, computationally complex
  - In gate-level circuit, B is already Groebner basis
    - But it must also include polynomials $\langle x^2 - x \rangle$ for all *Boolean* signals x in the circuit (i.e., $x = 0, 1$)

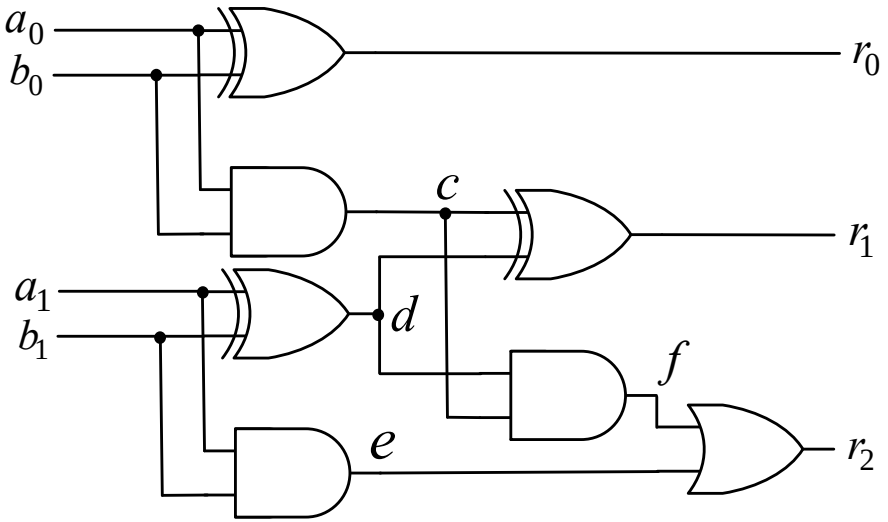
Reducing F_{spec} modulo B

- Reduce F_{spec} modulo B *by a series of polynomial divisions*



Example (STABLE)

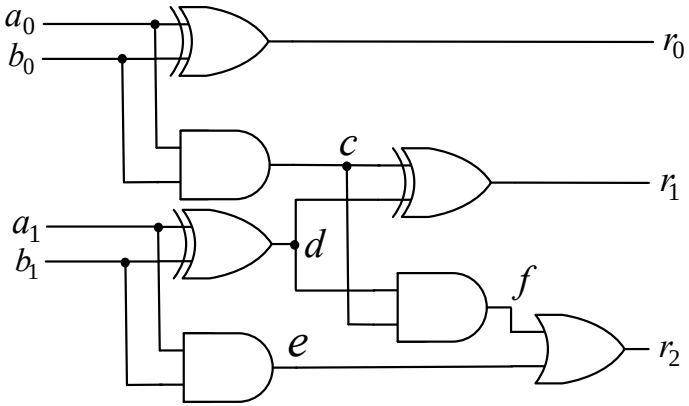
- Example: 2-bit adder
  - *Fspec* = $a_0 + b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 - r_0$
  - *B* = list of polynomials describing gates



$$\begin{cases} g_1 = r_0 - (a_0 + b_0 - 2a_0b_0) \\ g_2 = c - (a_0b_0) \\ g_3 = d - (a_1 + b_1 - 2a_1b_1) \\ g_4 = r_1 - (c + d - 2cd) \\ g_5 = f - (cd) \\ g_6 = e - (a_1b_1) \\ g_7 = r_2 - (e + f - ef) \end{cases}$$

Polynomial Division (1)

- Divide polynomial F_{spec} =



$$
\begin{cases}
g_1 = r_0 - (a_0 + b_0 - 2a_0b_0) \\
g_2 = c - (a_0b_0) \\
g_3 = d - (a_1 + b_1 - 2a_1b_1) \\
g_4 = r_1 - (c + d - 2cd) \\
g_5 = f - (cd) \\
g_6 = e - (a_1b_1) \\
g_7 = r_2 - (e + f - ef)
\end{cases}
$$

$a_0 + b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 - r_0$

$= -(a_0 + b_0 - 2a_0b_0) + r_0 + b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 - r_0$

$= 2a_0b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1$

$= -2(a_0b_0) + 2c + 2a_0b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1$

$= 2c + 2a_1 + 2b_1 - 4r_2 - 2r_1$

$= -2(a1 + b1 - 2a_1b_1) + 2d + 2c + 2a_1 + 2b_1 - 4r_2 - 2r_1$

$= 4a_1b_1 + 2d + 2c - 4r_2 - 2r_1$

$= -2(c + d - 2cd) + 2r_1 + 4a_1b_1 + 2d + 2c - 4r_2 - 2r_1$

$= 4cd + 4a_1b_1 - 4r_2$

$= -4(cd) + 4f + 2cd + 4a_1b_1 - 4r_2$

$= 4f + 4a_1b_1 - 4r_2$

$= -4(a_1b_1) + 4e + 4f + 4a_1b_1 - 4r_2$

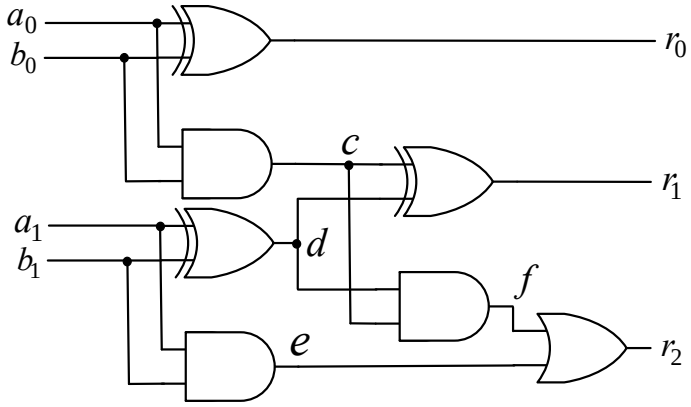
$= 4e + 4f - 4r_2$

$= -4(e + f - ef) + 4r_2 + 4e + 4f - 4r_2$

$= 4ef$

Polynomial Division (2)

- Continue dividing polynomial *4ef*



$$\begin{cases} g_1 = r_0 - (a_0 + b_0 - 2a_0b_0) \\ g_2 = c - (a_0b_0) \\ g_3 = d - (a_1 + b_1 - 2a_1b_1) \\ g_4 = r_1 - (c + d - 2cd) \\ g_5 = f - (cd) \\ g_6 = e - (a_1b_1) \\ g_7 = r_2 - (e + f - ef) \end{cases}$$

4ef

= 4e(cd)

= 4(a_1b_1)(cd)

= 4(a_1b_1) (a_0b_0) ($a_1 + b_1 - 2a_1b_1$)

= 4(a_1b_1) ($a_1 + b_1 - 2a_1b_1$) (a_0b_0)

= 4($a_1b_1a_1 + a_1b_1b_1 - 2a_1b_1a_1b_1$) ($a_0b_0$)

= 4(0) (a_0b_0)

= 4ef = 0

This means that F_{spec} mod $B = 0$, hence the circuit correctly implements a 2-bit adder.

Example Verification Formulation

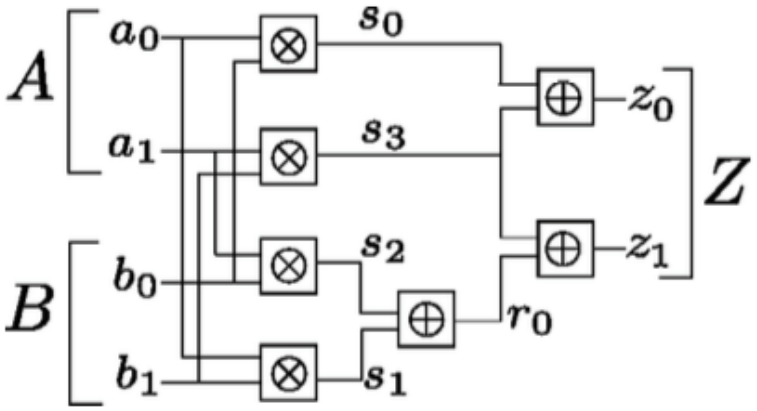


Figure: A GF multiplier over $\mathbb{F}_{2^2}$

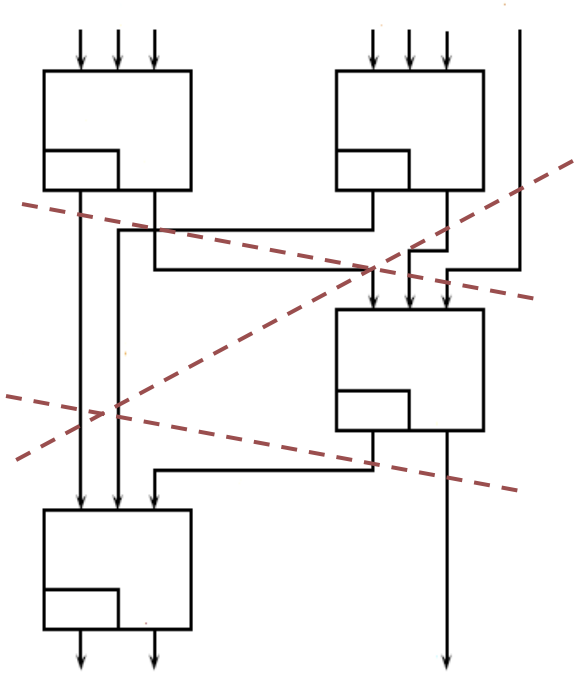
Ideal $J = \langle f_1, \ldots, f_{10} \rangle$

$$z_0 = s_0 \oplus s_3; \mapsto f_1 : z_0 + s_0 + s_3$$
$$z_1 = r_0 \oplus s_3; \mapsto f_2 : z_1 + r_0 + s_3$$
$$\vdots$$
$$s_0 = a_0 \wedge b_0; \mapsto f_7 : s_0 + a_0 \cdot b_0$$
$$A = a_0 + a_1\alpha; \mapsto f_8 : A + a_0 + a_1\alpha$$
$$B = b_0 + b_1\alpha; \mapsto f_9 : B + b_0 + b_1\alpha$$
$$Z = z_0 + z_1\alpha; \mapsto f_{10} : Z + z_0 + z_1\alpha$$

Ideal $J_0 = \langle z_0^2 - z_0, s_0^2 - s_0, \ldots, A^{2^k} - A, B^{2^k} - B, Z^{2^k} - Z \rangle$

Verification problem: Check if $f \xrightarrow{GB(J+J_0)}_+ 0$?

Data Flow Approach



- Treat computation as *flow of data*

  [Basith-FMCAD'11, Ciesielski-HVC'13,]

- Based on observation that :
  - In an arithmetic circuit an *integer flow* across the circuit is the same at any point (cut) in the circuit
    - *Cut*: set of signals separating *PIs* from *POs*
  - Write equations to represent the flow for a cut
  - Functional correctness can be done by proving that the *flow at PIs = flow at the POs*.

- Define *input* and *output signatures* of the network

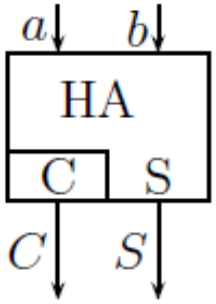
- Prove functionality by checking if input signature *can be transformed into* output signature (or vice-versa)
  - ABL networks (HA based, linear)
  - Gate-level networks (nonlinear)

Arithmetic Network Model

- Represent design as network of HAs and FAs (if possible)

Half-Adder (HA)

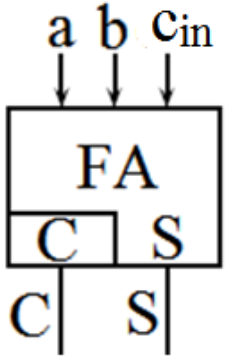
- Binary inputs (*a, b*)
- Binary outputs (*S, C*)



$a + b = 2C + S$

Full-Adder (FA)

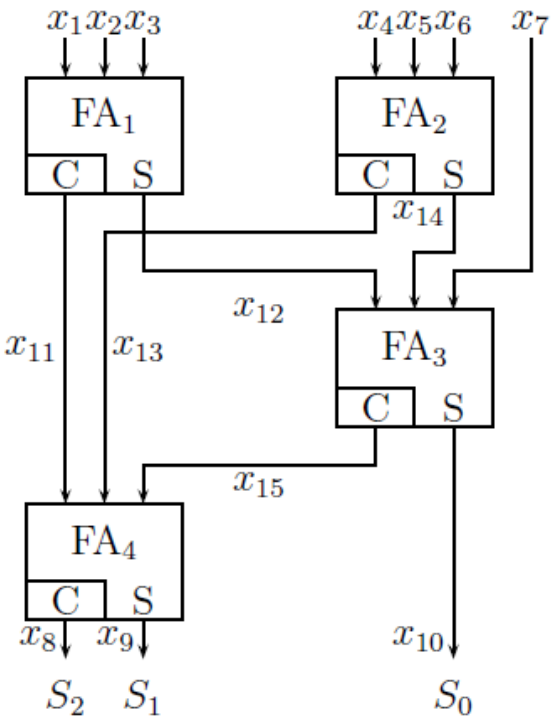
- Binary inputs (*a, b , c_{in}*)
- Binary outputs (*S, C*)



$a + b + c_{in} = 2C + S$

Input & Output Signatures

- *Input signature*: Functionality provided by user (*spec*)



$Sig_{In} = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$

Transform Sig_{in} into Sig_{out} to *verify* function

Or

Transform Sig_{out} into Sig_{in} to *extract* function

$Sig_{out} = 4S_2 + 2S_1 + S_0$

Output signature: Binary encoding of outputs

Data Flow Model – Basic Concept

- Sig_{in} transformed into Sig_{out} by a series of rewriting steps *(~Gaussian elimination)*

$a + b + c = 2\,C + S$

- Replace input part of the *FA* equation by the output part

$Sig_{in} = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$ *(PI)*

$\rightarrow (2\,x_{11} + x_{12}) + x_4 + x_5 + x_6 + x_7$

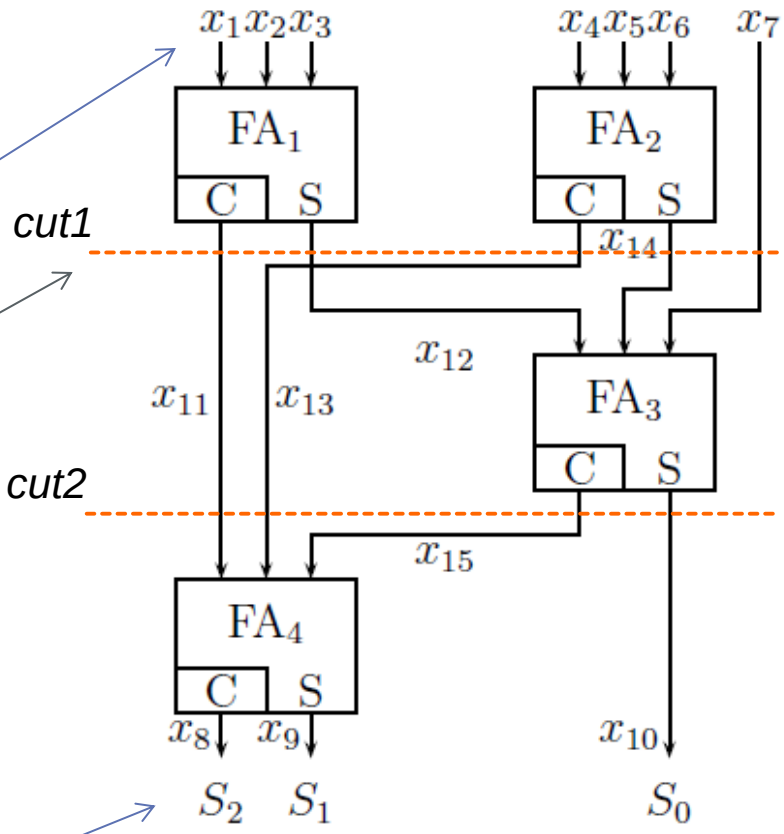
$\rightarrow (2\,x_{11} + x_{12}) + (2\,x_{13} + x_{14}) + x_7$

Continue:

$2\,x_{11} + 2\,x_{13} + x_{12} + x_{14} + x_7$

$\rightarrow 2\,x_{11} + 2\,x_{13} + (2\,x_{15} + x_{10})$

$\rightarrow 2\,(2\,x_8 + x_9) + x_{10} = 4S_2 + 2S_2 + S_0$ *(PO)*



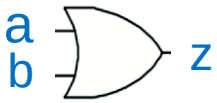
- This proves that circuit implements a 7-3 compactor

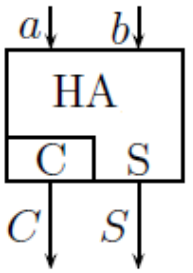
Gate-level Arithmetic Circuits

- Functional correctness can be shown by
  - Forward rewriting (*PI* $\rightarrow$ *PO*) or
  - Backward rewriting (*PO* $\rightarrow$ *PI*)

- Algebraic model:

$$\neg a = 1 - a$$
$$a \wedge b = a \cdot b$$
$$a \vee b = a + b - a \cdot b$$
$$a \oplus b = a + b - 2a \cdot b$$





- Forward rewriting
  - Polynomial division (reduction),
  - Replacing input expression by outputs
  - HA: *(a + b) / (a + b - 2C - S) = 2C + S*

equation: *a + b = 2C + S*

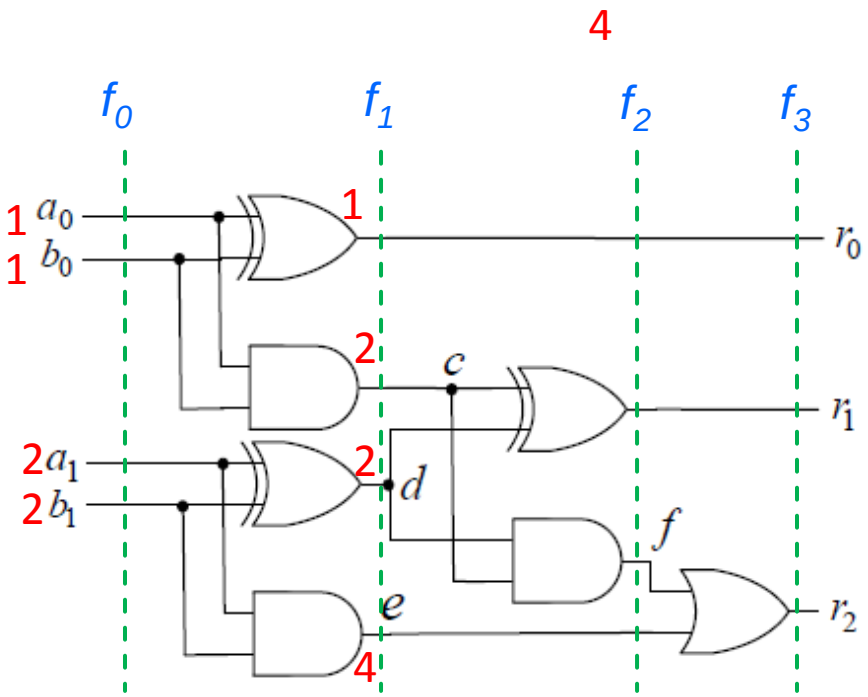
polynomial: *(a + b - 2C - S)*

- Backward rewriting
  - Replacing gate output by the expression in its inputs (expansion)
  - e.g., OR gate: *z = a + b - a*b*

Forward Rewriting

Example: 2-bit adder

- Problem: *RE ≠ Ø*
  - $\langle x^2\text{-}x \rangle$ was not used
  - Can we reduce it to zero?



$$f_0 = 2b_1 + 2a_1 + b_0 + a_0$$

$$f_1 = \underbrace{4e - 4a_1b_1}_{} + \underbrace{2d - 2(a_1 + b_1 - 2a_1b_1)}_{} + 2a_1 + 2b_1$$
$$\underbrace{+2c - 2a_0b_0}_{} + \underbrace{r_0 - (a_0 + b_0 - 2a_0b_0)}_{} + a_0 + b_0$$
$$= \underline{4e + 2d + 2c + r_0}$$

$$f_2 = \underbrace{4f - 4dc}_{} + \underbrace{2r_1 - 2(d + c - 2dc)}_{} + 4e + 2d + 2c + r_0$$
$$= \underline{4e + 4f + 2r_1 + r_0}$$

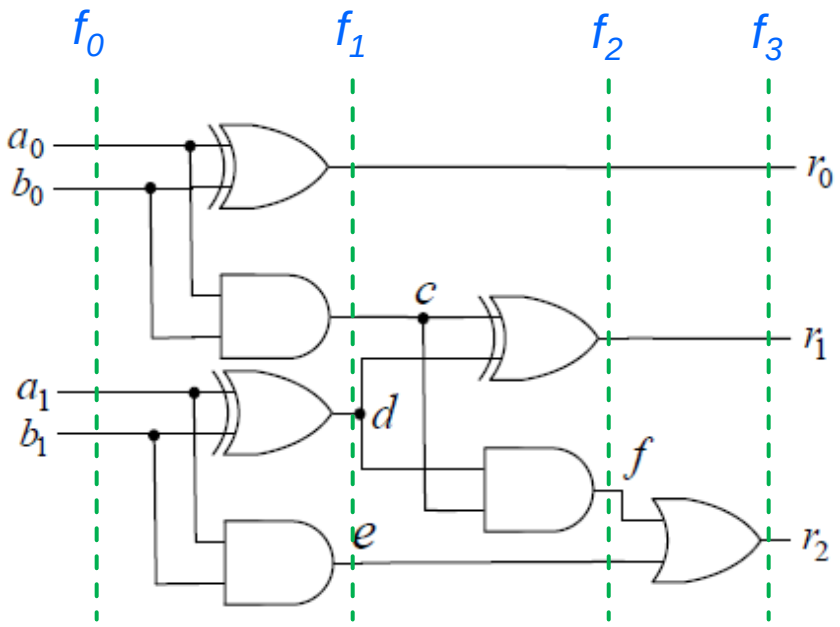
$$f_3 = \underbrace{4r_2 - 4(e + f - ef)}_{} + 4e + 4f + 2r_1 + r_0$$
$$= \underline{4r_2 + \boxed{4ef} + 2r_1 + r_0}$$

- Problem: *RE = 4ef, but … it reduces to 0*

$$4ef = 4(a_1b_1)(dc)$$
$$= 4(a_1b_1)((a_1 + b_1 - 2a_1b_1)(a_0\, b_0))$$
$$= 4(a_1b_1)(a_1\, a_0\, b_0 + b_1\, a_0\, b_0 - 2a_1b_1\, a_0\, b_0)$$
$$= 4(a_1b_1a_1\, a_0\, b_0 + a_1b_1b_1\, a_0\, b_0 - 2a_1b_1a_1b_1\, a_0\, b_0)$$
$$= 4(a_1b_1\, a_0\, b_0 + a_1b_1\, a_0\, b_0 - 2a_1b_1\, a_0\, b_0)$$
$$= 0$$

Backward Rewriting

- Replace gate output by its equation
  - Backward *symbolic simulation*
  - No *RE* ! It will *never* be generated
  - But … the expression can explode



$$f_3 = 4r_2 + 2r_1 + r_0$$

$$f_2 = 4(f + e - ef) + 2r_1 + r_0$$
$$= 4f + 4e - 4ef + 2r_1 + r_0$$

$$f_1 = 4e + 4(\mathrm{cd}) - 4\,\mathrm{e}(cd) + 2(\mathrm{c} + d - 2cd) + r_0$$
$$= 4e + 2c + 2d + r_0 - 4ecd$$

$$f_0 = 4(a_1b_1) + 2(a_0b_0) + 2(a_1 + b_1 - 2a_1b_1)$$
$$+ (a_0 + b_0 - 2a_0b_0)$$
$$- 4(a_1b_1)(a_0b_0)(a_1 + b_1 - 2a_1b_1)$$

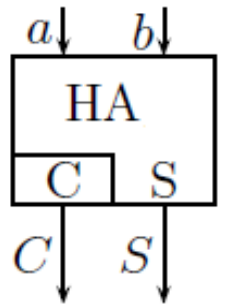
$$= 2a_1 + 2b_1 + a_0 + b_0$$

It matches the specification:

$\rightarrow$ *circuit is correct*

Function Extraction - summary

- Important features
  - Backward rewriting = **function extraction !**



$a + b = 2C + S$

- Different than standard symbolic simulation
  - Proof of functional correctness is done by propagating cut expressions for all the signals, from POs, rather than from individual outputs
  - Cancellation happen during the process (example: HA)
    - Consider *cut = 2C+S*, with *C = ab; S = a + b -2ab*
    - If done separately: first replace *C = ab,* then replace *a, b* by other signals, up to certain level, so that cut =

      $2\,f(x)\,g(y) + f(x) + f(y) - 2ab$

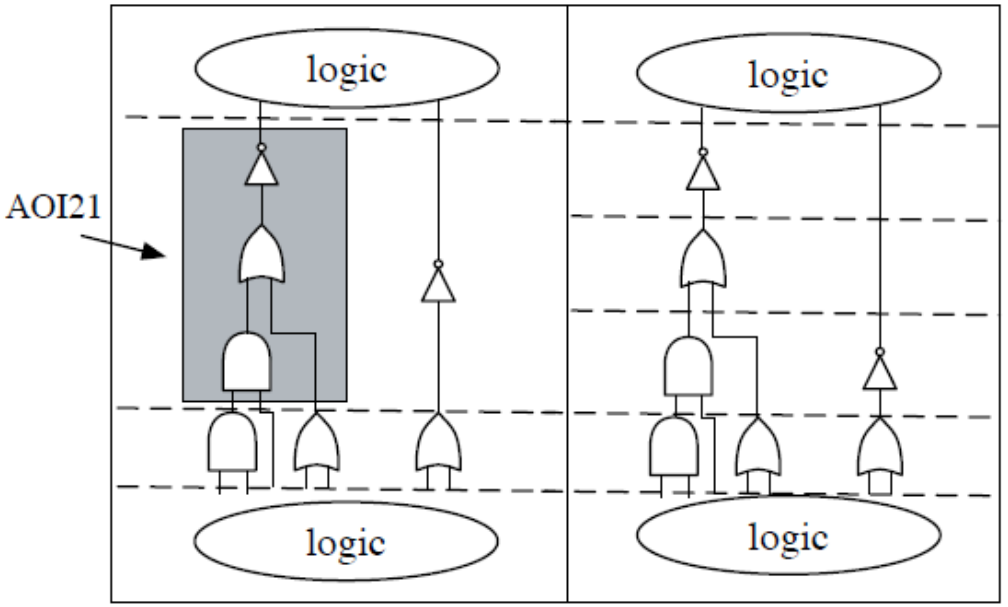
      It cannot be simplified until a, b are substituted.
    - But if S is replaced immediately after C, then

      $cut = 2ab + a + b - 2ab = a + b$
    - proving that $2C+S = a + b$

Backward Rewriting - issues

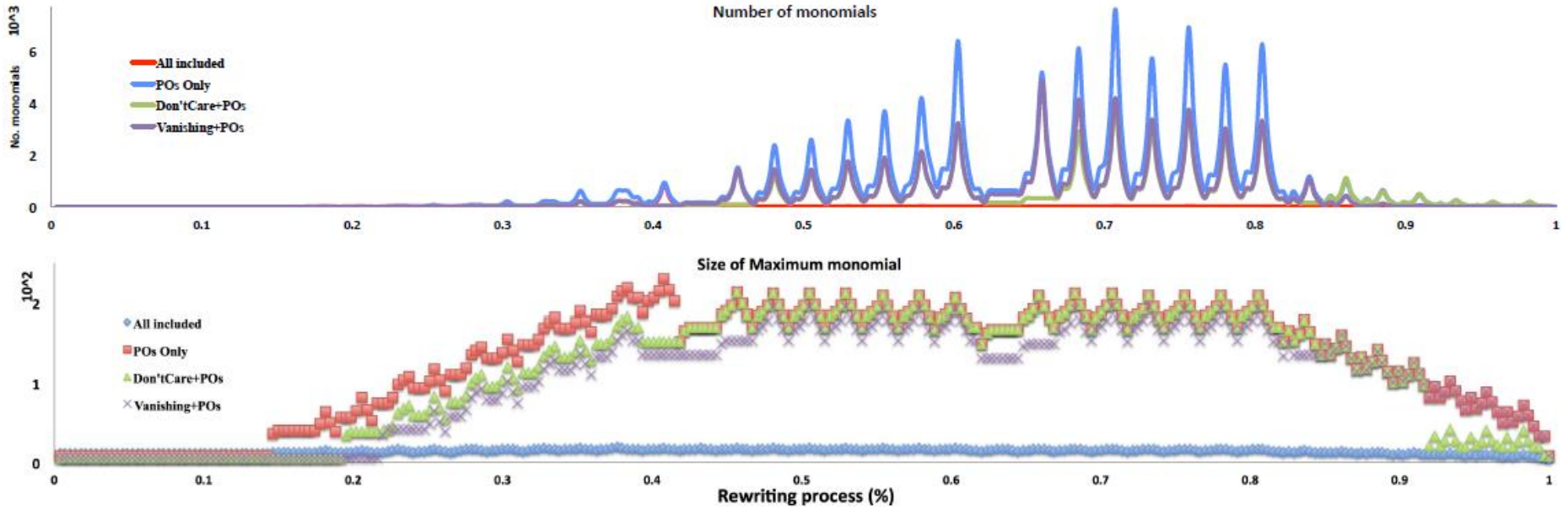
- No residual expression !
  - But the cut expression can explode (*fat belly* issue)
  - Choice and ordering of cuts during rewriting affects performance

- Issues:
  - Minimize the “fat belly”, the size of largest expression (memory)
  - Handling complex gates
  - Provide cuts in AOI gate



Experimental Results

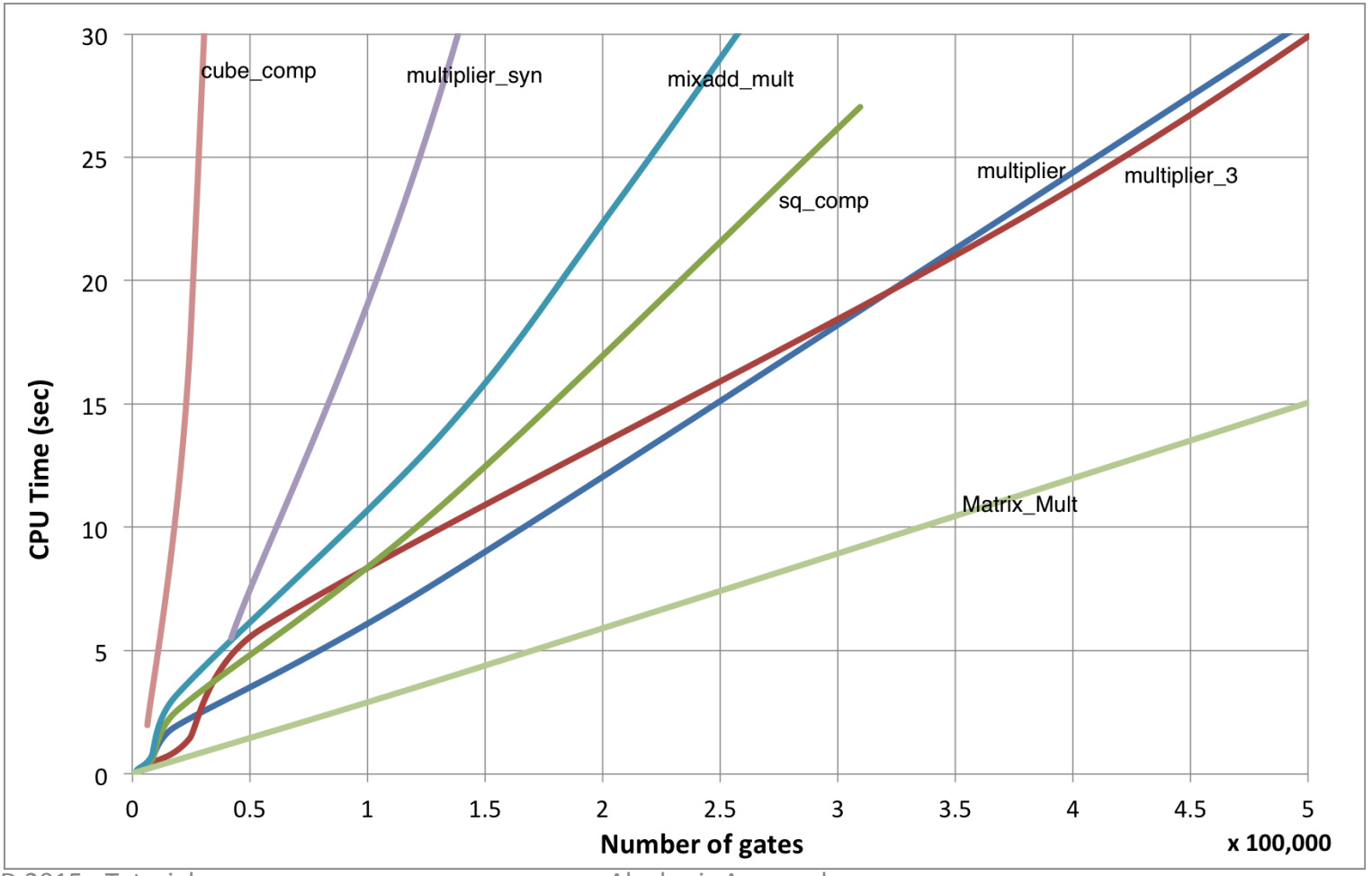
- Effect of ordering of cuts and including some useful redundancy (*vanishing polynomials*) on the size of cut expressions



4-bit serial squarer

Experimental Results

- Performance of the backward rewriting process
  - Combinational and sequential (bounded model) circuits



Experimental Results

Performance for original and *synthesized* designs

Benchmark		64-Bit			128-Bit			256-Bit		
Name	Function	# Gates	CPU [sec]	Mem	# Gates	CPU [sec]	Mem	# Gates	CPU [sec]	Mem
adder	$F = A + B$ (Wallace)	445	0.01	1.5 MB	893	0.05	3.5 MB	1.8K	0.10	5.7 MB
adder_syn	$F = A + B$ (Wallace)	559	0.06	3.5 MB	1.1K	0.15	5.3 MB	1.3K	0.22	5.5 MB
shift_Add	$F = A + A/2 + A/4 + A/8$	1.9K	0.09	3.6 MB	3.8K	0.20	9.8 MB	7.7K	0.44	18.2 MB
shift_Add_syn	$F = A + A/2 + A/4 + A/8$	1.2K	0.08	4.6 MB	2.5K	0.28	9.1 MB	4.6K	1.39	15.4 MB
multiplier	$F = A \cdot B$ (CSA Array)	32K	1.89	72 MB	129K	7.78	129 MB	521K	32.26	1.15 GB
multiplier_syn	$F = A \cdot B$ (CSA Array)	42K	5.50	76 MB	164K	39.64	299 MB	663K	285.22	1.25 GB
mixAddMult	$F = A \cdot (B + C)$	33K	3.17	18 MB	131K	13.77	306 MB	525K	70.18	1.18 GB
mixAddMult_syn	$F = A \cdot (B + C)$	39K	5.03	80 MB	161K	34.32	302 MB	650K	209.31	1.12 GB
multiplier_3	$F = A_1B_1 + A_2B_2 + A_3B_3$	98K	5.88	75 MB	393K	23.32	392 MB	1,571K	-	MO
sq_comp	$F = A^2 + 2 \cdot A + 1$	33K	2.56	18 MB	132K	10.96	285 MB	527K	48.84	1.13 GB
cube_comp	$F = 1 + A + A^2 + A^3$	99K	192.8	416 MB	395K	2052.85	2.3 GB	1,576K	TO	-
Matrix_Mult	$F = A[3 \times 3] \cdot B[3 \times 1]$	293K	18.82	621 MB	1,176K	77.09	2.5 GB	4,712K	-	MO

Multiplier design - comparisons with other methods

multiplier_synthesized												*multiplier_3*	
Statistics		Function-Extraction		[23]	SAT [sec]			SMT [sec]			Commercial		
Size	#Gates	CPU [sec]	Mem	[sec]	lingeling	minisat_blbd	ABC	Boolector	Z3	CVC4	Formality	Our	Formality
4	86	0.01	2.2 MB	0.45	0.00	0.00	0.01	0.00	0.03	0.09	0.81	0.02	2.34
8	481	0.04	2.9 MB	1.72	4.40	62.75	11.66	7.18	16.55	42.63	3.19	0.07	21.51
12	1.2K	0.08	4.3 MB	5.21	TO	1615.47	UD	2030.19	TO	TO	108.1	0.17	150.65
16	2.1K	0.14	6.1 MB	7.34	TO	TO	UD	TO	TO	TO	111.2	0.33	798.24
64	41.4K	5.50	76 MB	TO	TO	TO	UD	TO	TO	TO	675.4	5.88	TO
128	164K	39.64	299 MB	TO	TO	TO	UD	TO	TO	TO	TO	23.32	TO
256	663K	285.22	1.25 GB	TO	TO	TO	UD	TO	TO	TO	TO	97.60	TO
512*	2,091K	130.22	4.44 GB	TO	TO	TO	UD	TO	TO	TO	TO	MO	TO

Summary and Conclusions

- Forward rewriting is simple to implement
  - No memory explosion, but ..
  - Generates RE to be verified (typically using backward rewriting)
  - Need to compute weights (not easy for non-linear ckts)
- Backward rewriting is more reliable (no RE)
  - But … may explode
  - Can be used for function extraction:
    - the computed signature gives the specification
- Combining rewriting in both directions
  - Can be used for debugging
  - The difference between computed signatures tells which gate should be replaced *[isvlsi-2015]*
- Solving the problem for highly bit-optimized circuits
  - Remains difficult
  - Very large fat belly, memory explosion problem

Arithmetic Verification

Bibliography

Algebraic

[Basith-FMCAD'11] B., Mohamed Abdul, T. Ahmad, A. Rossi, and M. Ciesielski. "Algebraic approach to arithmetic design verification." FMCAD 2011.

[Ciesielski-HVC'13] M. Ciesielski, W. Brown, and A. Rossi."Arithmetic Bit-level Verification using Network Flow Model." HVC, 2013. 327-343.

[Ciesielski-ISVLSI'14] Ciesielski, M., Brown, W., Liu, D., and Rossi, A. "Function extraction from arithmetic bit-level circuits." ISVLSI, 2014.

[Ciesielski-DAC'15] "Verification of Gate-level Arithmetic Circuits by Function Extraction." DAC,2015

[Wienand-CAV'08] Wienand, Oliver, et al. "An algebraic approach for proving data correctness in arithmetic data paths." CAV, 2008.

[Pavlenko-DATE'11] Pavlenko, E., Wedler, M., Stoffel, D., Kunz, W., Dreyer, A., Seelisch, F., and Greuel, G. M. "STABLE: A new QF-BV SMT solver for hard verification problems combining Boolean reasoning with computer algebra." DATE, 2011.

[Marx-ICCAD'13] Marx, O., Wedler, M., Stoffel, D., Kunz, W., and Dreyer, A. "Proof logging for computer algebra based SMT solving." ICCAD,2013

Algebraic, cont'd

[Shekhar-ICCAD'05] Shekhar, N., Kalla, P., Enescu, F., and Gopalakrishnan, S. "Equivalence verification of polynomial datapaths with fixed-size bit-vectors using finite ring algebra." ICCAD,2005

[Shekhar-ICCD'05] Shekhar, N., Kalla, P., Enescu, F., and Gopalakrishnan, S. "Exploiting vanishing polynomials for equivalence verification of fixed-size arithmetic datapaths." ICCD, 2005.

[Pruss-DAC'14] Pruss, T., Kalla, P., and Enescu, F. "Equivalence verification of large galois field arithmetic circuits using word-level abstraction via gröbner bases." DAC,2014

[Lv-TCAD'14] Lv, J., Kalla, P., and Enescu, F. "Efficient gröbner basis reductions for formal verification of galois field arithmetic circuits." Trans. on CAD, 2014.

[Lv-DATE'12] Lv, J., Kalla, P., and Enescu, F. "Efficient gröbner basis reductions for formal verification of galois field multipliers." DATE,2012.

[Sun-DATE'15] Sun, X., Kalla, P., Pruss, T., & Enescu, F. (2015, March). Formal verification of sequential Galois field arithmetic circuits using algebraic geometry. DATE, 2015.

[Kroening-2008] Kroening, Daniel, and Ofer Strichman. Decision procedures: an algorithmic point of view. Springer Science & Business Media, 2008.

Algebraic, cont'd

[Cox-1992] Cox, David, John Little, and Donal O'shea. Ideals, varieties, and algorithms. Vol. 3. New York: Springer, 1992.

[Tim-TCAD'15] Tim Pruss, Priyank Kalla and Florian Enescu. Accepted, to appear in IEEE Trans. on CAD, 2015

ILP

[Brinkmann-VLSID'02], Raik, and Rolf Drechsler. "RTL-datapath verification using integer linear programming." VLSID,2002

[Zeng-DATE'01] Zeng, Z., Kalla, P., and Ciesielski, M."LPSAT: a unified approach to RTL satisfiability." DATE, 2001.

[Fallah-DAC'1998] Fallah, F., Devadas, S., and Keutzer, K. "Functional vector generation for HDL models using linear programming and 3-satisfiability." DAC,1998.

[Huan-TCAD'01] Huan, C. Y., & Cheng, K. T. (2001). Using word-level ATPG and modular arithmetic constraint-solving techniques for assertion property checking. Trans. on CAD, 20(3), 381-391.

Abstraction

[Johannsen-CAV'01] Johannsen, P. "BOOSTER: Speeding up RTL property checking of digital designs by word-level abstraction." CAV, 2001.

[Jain-TCAD'08] "Word-level predicate-abstraction and refinement techniques for verifying RTL verilog." Trans. on CAD, 2008.

[Kroening-ICCAD'07] Kroening, D., and Seshia, S. A. "Formal verification at higher levels of abstraction." ICCAD, 2007.

[Andraus-DAC'04] Andraus, Z. S., and Sakallah, K. A. "Automatic abstraction and verification of verilog models." DAC, 2004.

[Brady-MEMOCODE'10] Brady, B., Bryant, R. E., Seshia, S., and O'leary, J. W. "ATLAS: automatic term-level abstraction of RTL designs." MEMOCODE, 2010.

[Andraus-ASPDAC'06] Andraus, Z. S., Liffiton, M. H., and Sakallah, K. A. "Refinement strategies for verification methods based on datapath abstraction." ASP-DAC, 2006.

Theorem Provers

[Sawada-FMCAD'11] Sawada, J., Sandon, P., Paruthi, V., Baumgartner, J., Case, M., and Mony, H. "Hybrid verification of a hardware modular reduction engine." FMCAD, 2011.

[Sawada-FMCAD'06] Sawada, J., and Reeber, E. ”ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool." FMCAD, 2006.

[Russinoff-IMACS'05] Russinoff, D., Kaufmann, M., Smith, E., and Sumners, R. "Formal verification of floating-point RTL at AMD using the ACL2 theorem prover." IMACS, 2005.

[Harrison-2006] Harrison, J. "Floating-point verification using theorem proving." Formal Methods for Hardware Verification, 2006. 211-242.

[Brock-FMCAD'1996] Brock, B., Kaufmann, M., and Moore, J. S. “ACL2 theorems about commercial microprocessors." FMCAD, 1996.

[Hartmanis-2006] Hartmanis, A. C. D. H. J., Henzinger, T., Leighton, J. H. N. J. T., and Nivat, M. "Texts in Theoretical Computer Science An EATCS Series." (2006). Springer.

[Vasudevan-TCAD’07] Vasudevan, S., Viswanath, V., Sumners, R. W., & Abraham, J. Automatic verification of arithmetic circuits in RTL using stepwise refinement of term rewriting systems. Trans on CAD. 56(10), 1401-1414.

[Kapur-FMSD’1998] Kapur, D., & Subramaniam, M. (1998). Mechanical verification of adder circuits using rewrite rule laboratory. *Formal Methods in System Design*, *13*(2), 127-158.

Industry, floating-point

[Aharoni-ARITH'05] Aharoni, M., Asaf, S., Maharik, R., Nehama, I., Nikulshin, I., and Ziv, A. "Solving constraints on the invisible bits of the intermediate result for floating-point verification." Computer Arithmetic, 2005.

[Jacobi-DATE'05] Jacobi, C., Weber, K., Paruthi, V., and Baumgartner, J. “Automatic formal verification of fused-multiply-add FPUs." DATE, 2005.

[Krautz-DAC'14] Krautz, U., Paruthi, V., Arunagiri, A., Kumar, S., Pujar, S., and Babinsky, T. "Automatic Verification of Floating Point Units." DAC, 2014.

[Guralnik-TC'11] Guralnik, E., Aharoni, M., Birnbaum, A. J., and Koyfman, A. "Simulation-based verification of floating-point division." Trans. on Computers, 2011.

ABC, Simulation graph

[Brayton-CAV'10] Brayton, R., and Mishchenko, A. "ABC: An academic industrial-strength verification tool." CAV, 2010.

[Mishchenko-2010]. Mishchenko, A. "ABC: A system for sequential synthesis and verification." URL http://www. eecs. berkeley.edu/alanmi/abc

[Soeken-FMCAD'15] Soeken, M., Sterin, B., Drechsler, R., and Brayton, R. "Simulation Graphs for Reverse Engineering." FMCAD, 2015.

Canonical Diagrams

[Bryant-TC'1986] Bryant, R. E. "Graph-based algorithms for boolean function manipulation." TC, 100.8 (1986): 677-691.

[Bryant-DAC'1995] Bryant, R. E., and Chen, Y. A. "Verification of arithmetic circuits with binary moment diagrams." DAC, 1995.

[Chen-ICCAD'1997] Chen, Y. A., and Bryant, R. E. "PHDD: An efficient graph representation for floating point circuit verification." ICCAD, 1997.

[Drechsler-ISMVL'1997]. Drechsler, R., Keim, M., and Becker, B. "Sympathy-MV: Fast Exact Minimization of Fixed Polarity Multi-Valued Linear Expressions." ISMVL, 1997.

[Ciesielski-TCAD'06] Ciesielski, M., Kalla, P., and Askar, S. "Taylor expansion diagrams: A canonical representation for verification of data flow designs." Trans on Computers, 55.9 (2006): 1188-1201.

[Ciesielski] Ciesielski, M., Gomez-Prado, D., Ren, Q., Guillot, J., & Boutillon, E. “Optimization of data-flow computations using canonical TED representation”. Trans on CAD. 28(9), 1321-1333. 2009

SAT/SMT

[Zhang-ICCAD'01] Zhang, L., Madigan, C. F., Moskewicz, M. H., and Malik, S. "Efficient conflict driven learning in a boolean satisfiability solver." ICCAD, 2001.

[Biere-2009] Biere, A., Heule, M., and van Maaren, H. “Handbook of satisfiability.” Vol. 185, 2009.

[Aloul-IWLS'02] Aloul, F. A., Mneimneh, M. N., and Sakallah, K. A. "ZBDD-Based Backtrack Search SAT Solver." IWLS. 2002.

[Alizadeh-TCAD'10] Alizadeh, B., and Fujita, M. "Modular datapath optimization and verification based on modular-HED." Trans. on CAD, 29.9 (2010): 1422-1435.

[Mishchenko-2010]. Mishchenko, A. "ABC: A system for sequential synthesis and verification." URL http://www. eecs. berkeley.edu/alanmi/abc

[Sörensson-SAT’09] Sörensson, N., & Eén, N. (2009). MiniSat 2.1 and MiniSat++ 1.0—SAT race 2008 editions. SAT, 31.

[Brummayer-2009] Brummayer, R., & Biere, A. (2009). Boolector: An efficient SMT solver for bit-vectors and arrays. In Tools and Algorithms for the Construction and Analysis of Systems (pp. 174-177).

SAT/SMT, cont'd

[Barrett-CAV'11] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., ... & Tinelli, C. "CVC4". CAV, 2011.

[De Moura-2008]De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In Tools and Algorithms for the Construction and Analysis of Systems (pp. 337-340).

[Silva-ICCAD'1997] Silva, J. P. M., & Sakallah, K. A. (1997, January). GRASP—a new search algorithm for satisfiability. ICCAD, 1996.

[Biere-2010] Biere, Armin. "Lingeling, plingeling, picosat and precosat at SAT race 2010." FMV Report Series Technical Report 10.1 (2010).