University of Bremen

JOHANNES KEPLER UNIVERSITY LINZ

Divider Verification Using Symbolic Computer Algebra and Delayed Don't Care Optimization

Alexander Konrad[1], Christoph Scholl[1], Alireza Mahzoon[2], Daniel Große[3], Rolf Drechsler[2]

[1]University of Freiburg, Germany
[2]University of Bremen, Germany
[3]Johannes Kepler University Linz, Austria

Verification of Arithmetic Circuits, Motivation

- **Circuit design containing arithmetic not only by processor vendors, but also by suppliers of special-purpose hardware**
- **Fully automatic formal verification of arithmetic circuits needed**

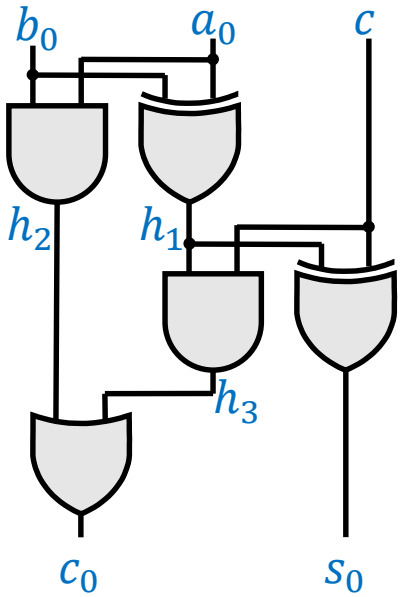
- **Great progress for gate-level multipliers during last years based on Symbolic Computer Algebra**
- **Situation for fully automatic formal verification of gate-level dividers not such beneficial.**

Symbolic Computer Algebra (SCA)

- **Symbolic Computer Algebra to verify integer arithmetic:**
  - **Exposition can be simplified by considering replacements of variables by gate polynomials („backward rewriting“)**
  - **Based on the fact that the polynomial for a pseudo-Boolean function $f: \{0,1\}^n \to Z$ is unique (up to reordering of terms)**
  - **Illustrated by a simple example: …**

SCA – Illustration of Backward Rewriting by Example

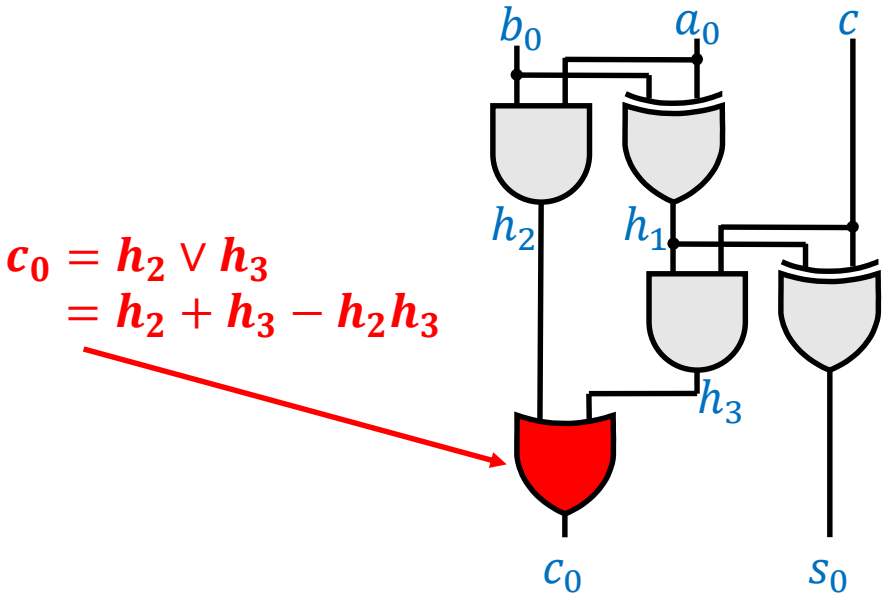
- **Full-Adder with specification $2c_0 + s_0 = a_0 + b_0 + c$:**



- **Start with output word $2c_0 + s_0$**

SCA – Illustration of Backward Rewriting by Example

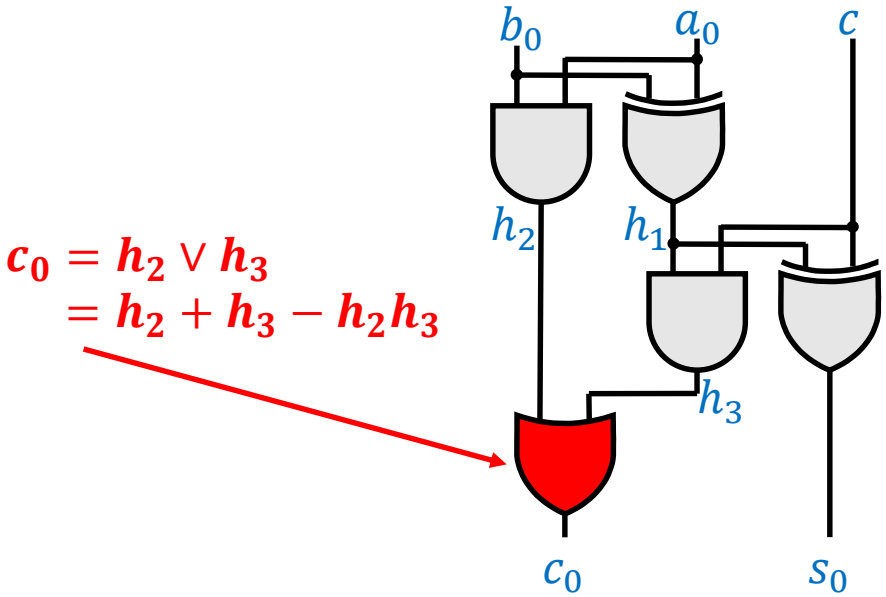
- **Full-Adder with specification $2c_0 + s_0 = a_0 + b_0 + c$:**



- **Start with output word $2c_0 + s_0$**

SCA – Illustration of Backward Rewriting by Example

- **Full-Adder with specification $2c_0 + s_0 = a_0 + b_0 + c$:**

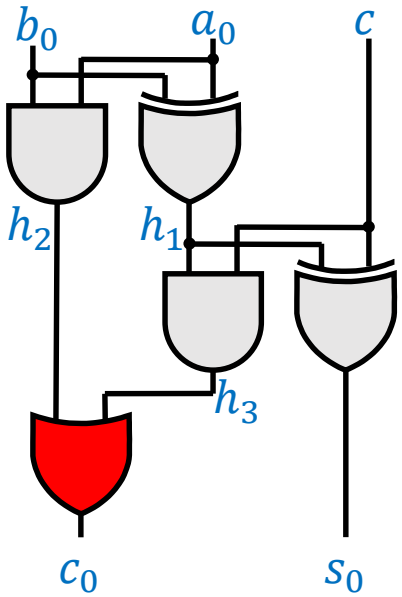


- **Start with output word $2c_0 + s_0$**

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

SCA – Illustration of Backward Rewriting by Example

- **Full-Adder with specification** $2c_0 + s_0 = a_0 + b_0 + c$:



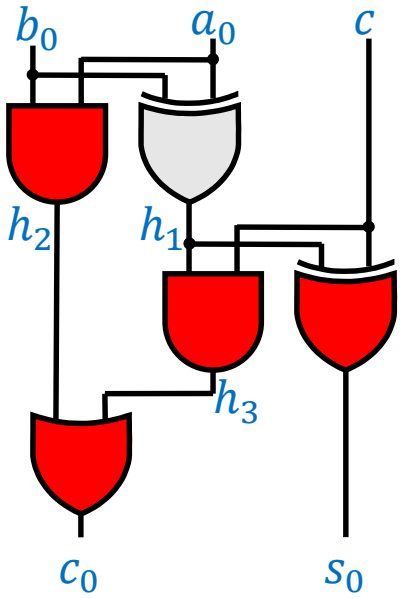
- **Start with output word** $2c_0 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow \ldots$

SCA – Illustration of Backward Rewriting by Example

- **Full-Adder with specification** $2c_0 + s_0 = a_0 + b_0 + c$:



- **Start with output word** $2c_0 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

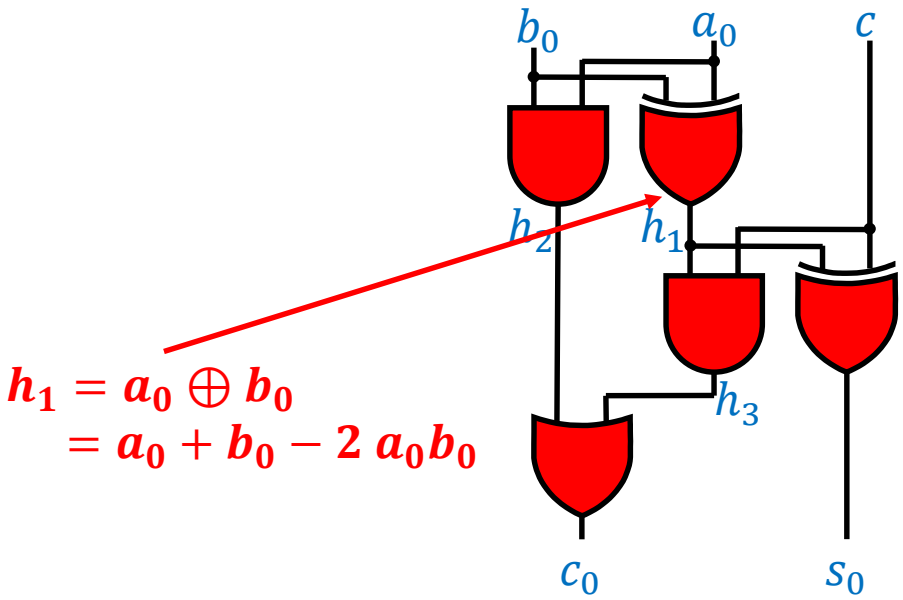
$\Rightarrow 2h_2 + 2ch_1 - 2ch_1h_2 + s_0$

$\Rightarrow 2h_2 - 2ch_1h_2 + c + h_1$

$\Rightarrow 2a_0b_0 - 2a_0b_0ch_1 + c + h_1$

SCA – Illustration of Backward Rewriting by Example

- **Full-Adder with specification** $2c_0 + s_0 = a_0 + b_0 + c$:



- **Start with output word** $2c_0 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

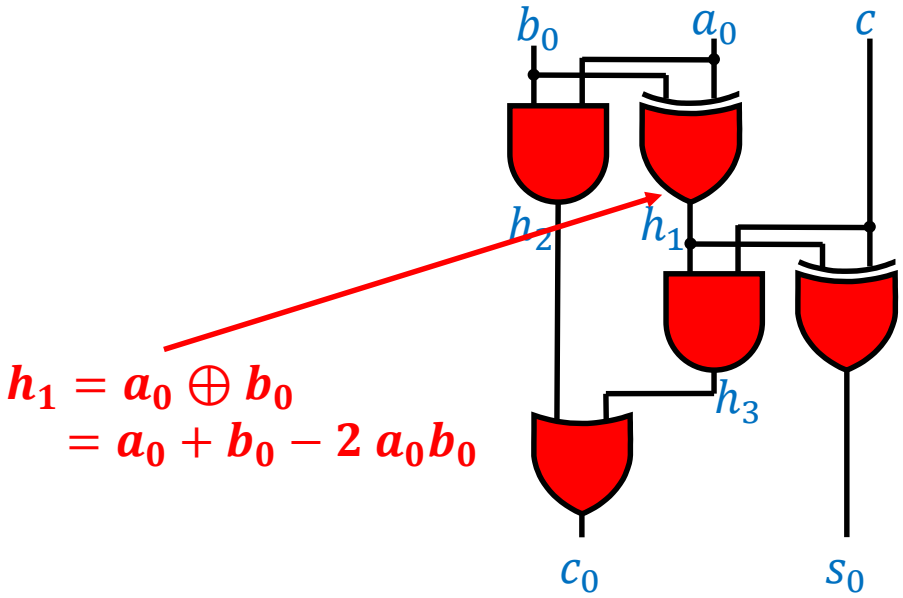
$\Rightarrow 2h_2 + 2ch_1 - 2ch_1h_2 + s_0$

$\Rightarrow 2h_2 - 2ch_1h_2 + c + h_1$

$\Rightarrow 2a_0b_0 - 2a_0b_0ch_1 + c + h_1$

SCA – Illustration of Backward Rewriting by Example

- **Full-Adder with specification** $2c_0 + s_0 = a_0 + b_0 + c$:



- **Start with output word** $2c_0 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2ch_1 - 2ch_1h_2 + s_0$

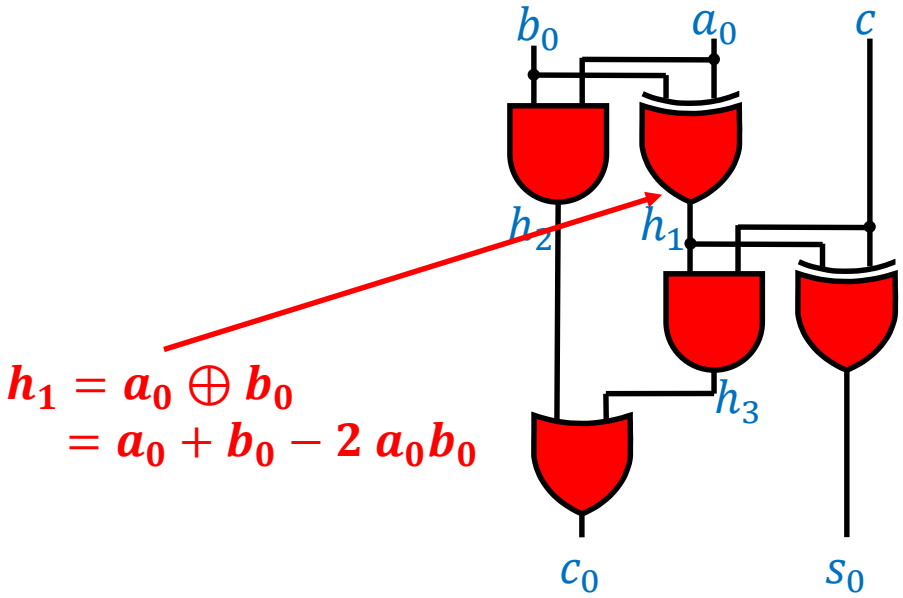
$\Rightarrow 2h_2 - 2ch_1h_2 + c + h_1$

$\Rightarrow 2a_0b_0 - 2a_0b_0ch_1 + c + h_1$

$\Rightarrow 2a_0b_0 - 2a_0^2b_0c - 2a_0b_0^2c$
$+4a_0^2b_0^2c + c + a_0 + b_0 - 2a_0b_0$

SCA – Illustration of Backward Rewriting by Example

- **Full-Adder with specification** $2c_0 + s_0 = a_0 + b_0 + c$:



- **Start with output word** $2c_0 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2ch_1 - 2ch_1h_2 + s_0$

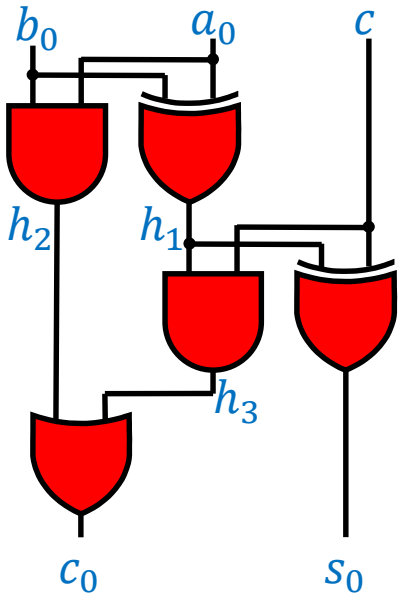
$\Rightarrow 2h_2 - 2ch_1h_2 + c + h_1$

$\Rightarrow 2a_0b_0 - 2a_0b_0ch_1 + c + h_1$

$\Rightarrow a_0 + b_0 + c$

SCA – Illustration of Backward Rewriting by Example

- **Full-Adder with specification** $2c_0 + s_0 = a_0 + b_0 + c$:



- **Or: Start with**

$$2c_0 + s_0 - a_0 - b_0 - c$$

$\Rightarrow \ldots$

$\Rightarrow \ldots$

$\Rightarrow \ldots$

$\Rightarrow \ldots$

$\Rightarrow a_0 + b_o + c - a_0 - b_0 - c = 0$

Integer Division

- **Given:**
  - **Dividend** $(0\, r_{2n-3}^{(0)} \ldots r_0^0)$ **with value** $R^{(0)} = \sum_{i=0}^{2n-3} r_i^{(0)}\, 2^i$
  - **Divisor** $(0\, d_{n-2}\, \ldots d_0)$ **with value** $D = \sum_{i=0}^{n-2} d_i 2^i$
  - **Input constraint** $0 \leq R^{(0)} < D \cdot 2^{n-1}$.
- **Compute:**
  - **Quotient** $(q_{n-1} \ldots q_0)$ **with value** $Q = \sum_{i=0}^{n-1} q_i 2^i$
  - **Remainder** $(r_{n-1} \ldots r_0)$ **with value** $R = \sum_{i=0}^{n-2} r_i 2^i - r_{n-1} 2^{n-1}$
  - **With**
    - **(vc1)** $R^{(0)} = Q \cdot D + R$
    - **(vc2)** $0 \leq R < D$.

Division algorithms

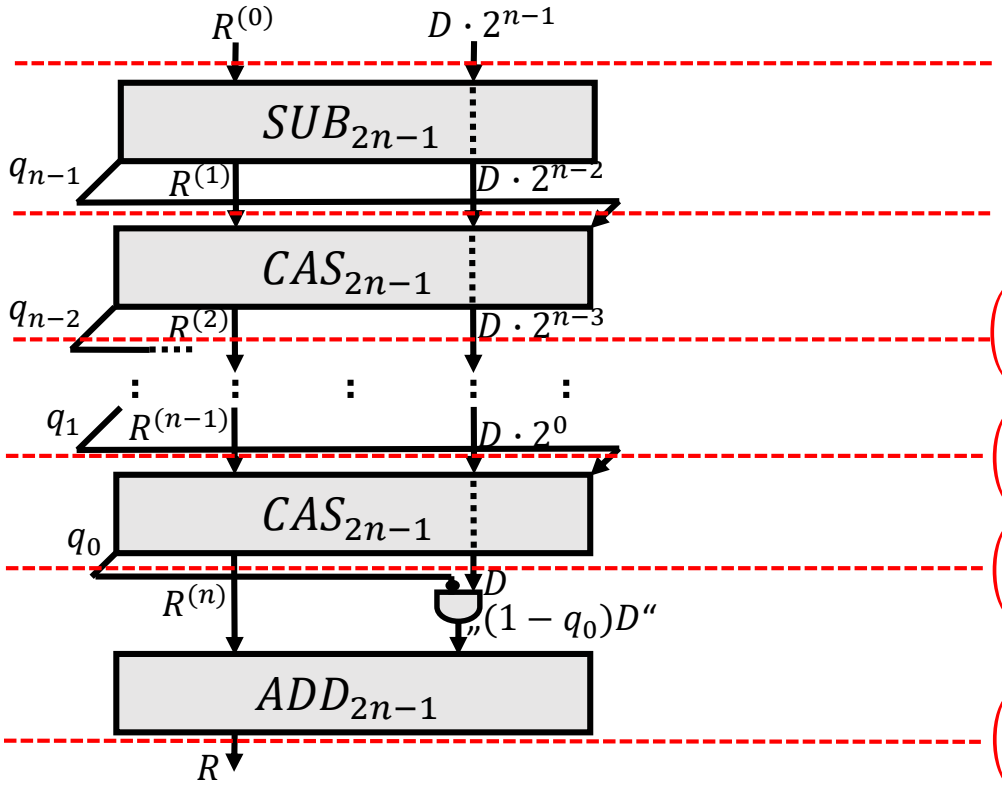
- **Most simple: Restoring division**
  - **Division by “school method“**
  - **Computing partial remainders $R^{(j)}$ and quotient bits q_{n-j} iteratively**

Division algorithms

- **Most simple: Restoring division**
  - **Division by "school method"**
  - **Computing partial remainders $R^{(j)}$ and quotient bits q_{n-j} iteratively**
- **Improved by: Non-restoring division**
  - **Combines back addition and subtraction into single addition**
  - **Relation of new and previous partial remainders derived from algorithm:**

$$R^{(j)} = R^{(j-1)} + (1 - 2q_{n-j+1})(D \cdot 2^{n-j})$$

Circuit for Non-Restoring Division



0

$R^{(1)} = R^{(0)} - D \cdot 2^{n-1}$

$2^{n-1} \cdot D + R^{(1)} - R^{(0)}$

$R^{(2)} = R^{(1)} + (1 - 2q_{n-1})D \cdot 2^{n-2}$

$\left(\sum_{i=n-1}^{n-1} q_i 2^i + 2^{n-2}\right) \cdot D + R^{(2)} - R^{(0)}$

$\left(\sum_{i=2}^{n-1} q_i 2^i + 2^1\right) \cdot D + R^{(n-1)} - R^{(0)}$

$R^{(n)} = R^{(n-1)} + (1 - 2q_1)D \cdot 2^0$

$\left(\sum_{i=1}^{n-1} q_i 2^i + 2^0\right) \cdot D + R^{(n)} - R^{(0)}$

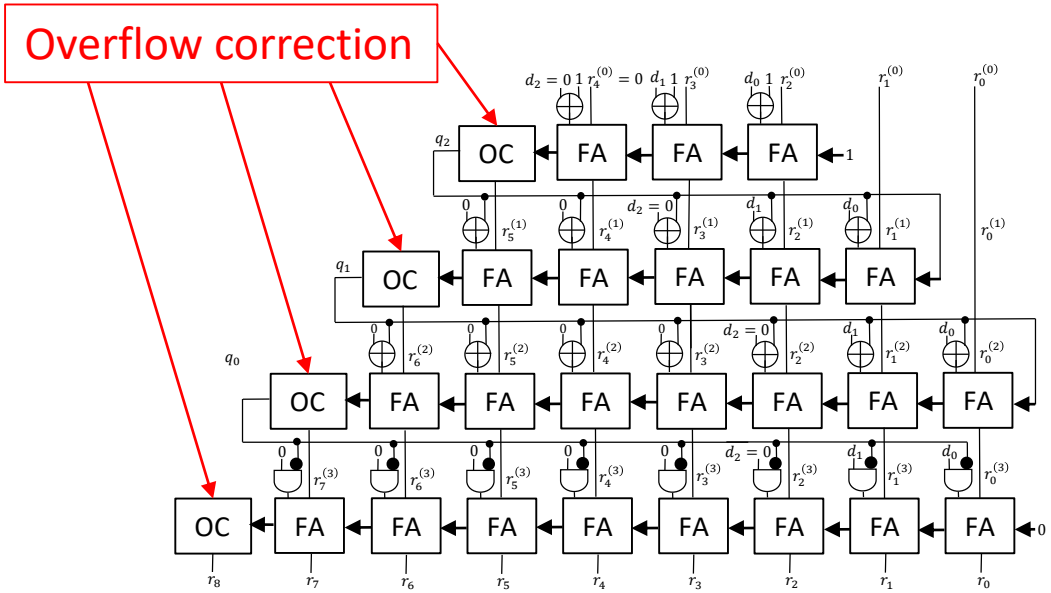
$R = R^{(n)} + (1 - q_0)D$

$\left(\sum_{i=0}^{n-1} q_i 2^i\right) \cdot D + R - R^{(0)}$

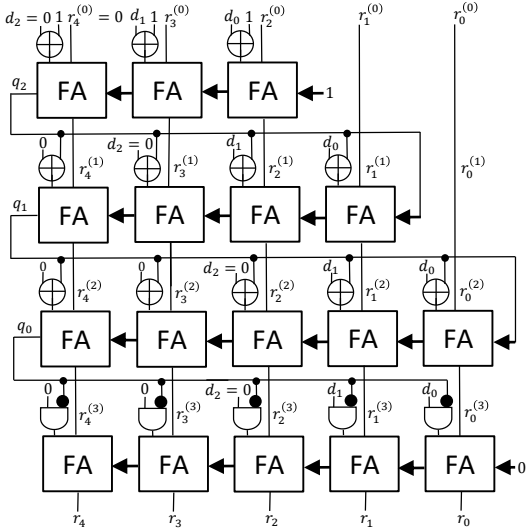
Sizes of Polynomials

- **Real gate level implementations lead to polynomials with exponential sizes on cuts between stages!**
- **Why?**

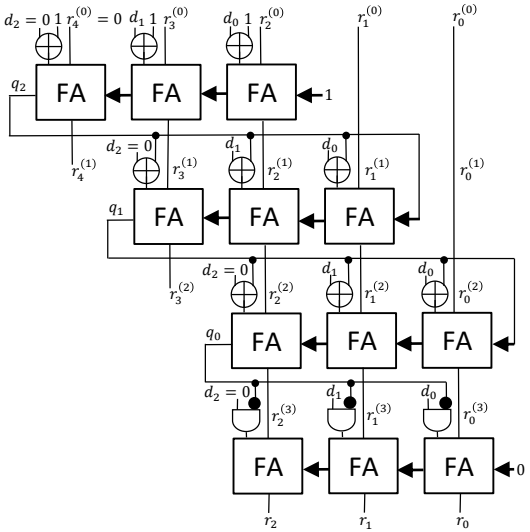
“Clean“ adder / subtractor stages



Omitted overflow bits



Omitted leading bits



Sizes of Polynomials

- Real gate level implementations lead to polynomials with exponential sizes on cuts between stages!
- Why?
  - Real gate level implementations omit overflow bits as well as leading bits of adders / subtractors
  - Still correct due to size restrictions for partial remainders derived from input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$ and the division algorithm

Sizes of Polynomials

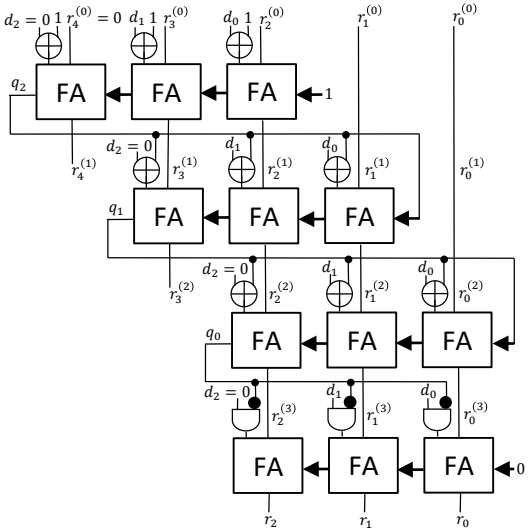
- Real gate level implementations lead to polynomials with exponential sizes on cuts between stages!
- Why?
  - Real gate level implementations omit overflow bits as well as leading bits of adders / subtractors
  - Still correct due to size restrictions for partial remainders derived from input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$ and the division algorithm
  - Restrictions resulting from input constraint not visible for backward rewriting!

Sizes of Polynomials

- Real gate level implementations lead to polynomials with exponential sizes on cuts between stages!
- Why?
  - Real gate level implementations omit overflow bits as well as leading bits of adders / subtractors
  - Still correct due to size restrictions for partial remainders derived from input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$ and the division algorithm
  - Restrictions resulting from input constraint not visible for backward rewriting!
  - Final polynomial after rewriting can even differ from 0 for correct implementation → "only needs to be 0 under the input constraint"

Previous work: Introducing Don’t Care Optimization

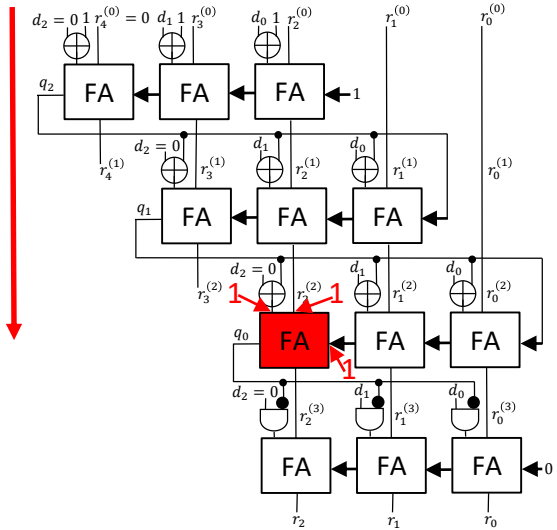
- **„Verifying Dividers Using Symbolic Computer Algebra and Don't Care Optimization“**
  - **Scholl, Konrad, Mahzoon, Große & Drechsler at Design, Automation and Test in Europe Conference 2021 [Scholl et al., DATE‘21]**



Previous work: Introducing Don't Care Optimization

- **Input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$ together with divider design implies „satisfiability don't cares" at boundaries of atomic blocks:**

$0 \leq R^{(0)} < D \cdot 2^{n-1}$



Example:

Input constraint implies that value combination (1, 1, 1) at the inputs of the atomic block marked in red cannot occur

⇒ satisfiablity don't care

Previous work: Introducing Don’t Care Optimization

- **Make use of satisfibility don‘t cares (obtained by „forward information propagation“) to optimize intermediate polynomials during backward rewriting.**

Previous work: Introducing Don't Care Optimization

- **Make use of satisfibility don't cares (obtained by „forward information propagation“) to optimize intermediate polynomials during backward rewriting.**
- $p = 1 - x_1 - x_2 - x_3 + 2x_1x_2 + 2x_1x_3 + 2x_2x_3 - 4x_1x_2x_3$
  - **Satisfiability don't care cubes** $\neg x_1 x_2 x_3$, $x_1 \neg x_2 \neg x_3$

Previous work: Introducing Don't Care Optimization

- **Make use of satisfibility don't cares (obtained by „forward information propagation“) to optimize intermediate polynomials during backward rewriting.**
- $p = 1 - x_1 - x_2 - x_3 + 2x_1x_2 + 2x_1x_3 + 2x_2x_3 - 4x_1x_2x_3$
  - **Satisfiability don't care cubes** $\neg x_1 x_2 x_3$, $x_1 \neg x_2 \neg x_3$
  - **Choose integer variable** v_1 **for** $\neg x_1 x_2 x_3$, **add** $v_1(1 - x_1)x_2x_3$

$$p = 1 - x_1 - x_2 - x_3 + 2x_1x_2 + 2x_1x_3 + (2 + v_1)x_2x_3 + (-v_1 - 4)x_1x_2x_3$$

Previous work: Introducing Don't Care Optimization

- **Make use of satisfibility don't cares (obtained by „forward information propagation“) to optimize intermediate polynomials during backward rewriting.**
- $p = 1 - x_1 - x_2 - x_3 + 2x_1x_2 + 2x_1x_3 + 2x_2x_3 - 4x_1x_2x_3$
  - **Satisfiability don't care cubes** $\neg x_1 x_2 x_3$, $x_1 \neg x_2 \neg x_3$
  - **Choose integer variable** v_2 **for** $x_1 \neg x_2 \neg x_3$, **add** $v_2 x_1 (1 - x_2)(1 - x_3)$

$$p = 1 + (v_2 - 1)x_1 - x_2 - x_3 + (2 - v_2)x_1x_2 + (2 - v_2)x_1x_3 + (2 + v_1)x_2x_3 + (v_2 - v_1 - 4)x_1x_2x_3$$

Previous work: Introducing Don't Care Optimization

- **Make use of satisfibility don't cares (obtained by „forward information propagation“) to optimize intermediate polynomials during backward rewriting.**
- $p = 1 - x_1 - x_2 - x_3 + 2x_1x_2 + 2x_1x_3 + 2x_2x_3 - 4x_1x_2x_3$
  - **Satisfiability don't care cubes** $\neg x_1 x_2 x_3$, $x_1 \neg x_2 \neg x_3$
  - **Choose integer variable** v_2 **for** $x_1 \neg x_2 \neg x_3$, **add** $v_2 x_1 (1 - x_2)(1 - x_3)$

$$p = 1 + (v_2 - 1)x_1 - x_2 - x_3 + (2 - v_2)x_1x_2 + (2 - v_2)x_1x_3 + (2 + v_1)x_2x_3 + (v_2 - v_1 - 4)x_1x_2x_3$$

  - **Remaining task: Choose** v_1 **and** v_2 **such that the coefficient of a maximum number of terms is 0.**

Previous work: Introducing Don't Care Optimization

⇒ **Optimization problem**

- $1 + (v_2 - 1)x_1 - x_2 - x_3 + (2 - v_2)x_1x_2 + (2 - v_2)x_1x_3 + (2 + v_1)x_2x_3 + (v_2 - v_1 - 4)x_1x_2x_3$
- **Equation system:**

  $v_2 - 1 = 0$

  $2 - v_2 = 0$

  $2 - v_2 = 0$

  $2 + v_1 = 0$

  $v_2 - v_1 - 4 = 0$

- **Maximize the number of satisfied equations!**

⇒ **Reduced to integer linear programming (ILP).**

Previous work: Introducing Don't Care Optimization

⇒ **Optimization problem**

- $1 + (v_2 - 1)x_1 - x_2 - x_3 + (2 - v_2)x_1x_2 + (2 - v_2)x_1x_3 + (2 + v_1)x_2x_3 + (v_2 - v_1 - 4)x_1x_2x_3$
- **Equation system:**

  $v_2 - 1 = 0$

  $2 - v_2 = 0$

  $2 - v_2 = 0$

  $2 + v_1 = 0$

  $v_2 - v_1 - 4 = 0$
- **Maximize the number of satisfied equations!**

⇒ **Reduced to integer linear programming (ILP).**

- **Here:** $v_1 = -2, v_2 = 2$.

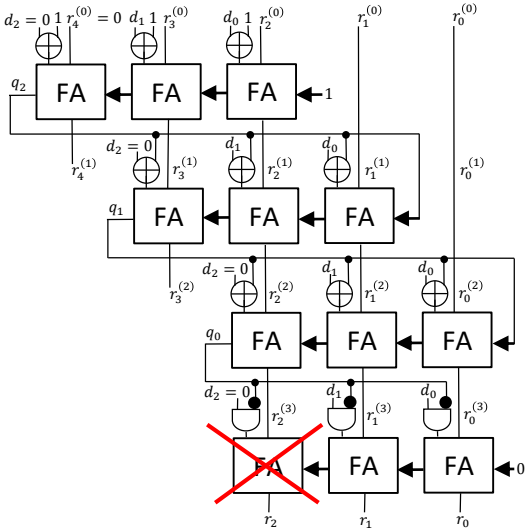
⇒ **Result:** $P(x_1, x_2, x_3) = 1 + x_1 - x_2 - x_3$

Previous work: Introducing Don’t Care Optimization

- **Don‘t Care Optimization used as part of backtracking approach**
- **Successfully avoided exponential polynomials during backward rewriting for real gate level implementations shown before**

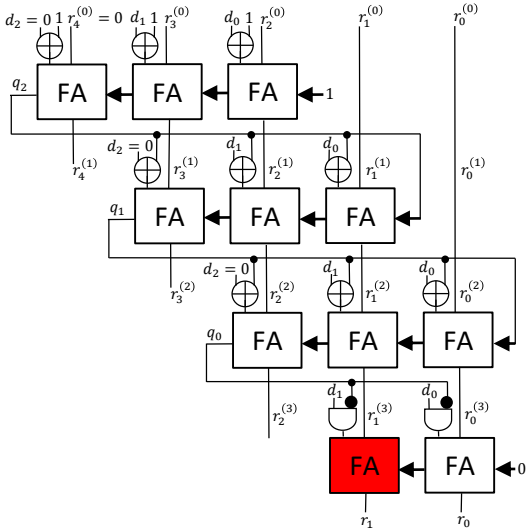
Further optimized implementation

- In correct dividers the remainder is non-negative, i.e. $r_{n-1} = 0$
- Computation of r_{n-1} can be omitted



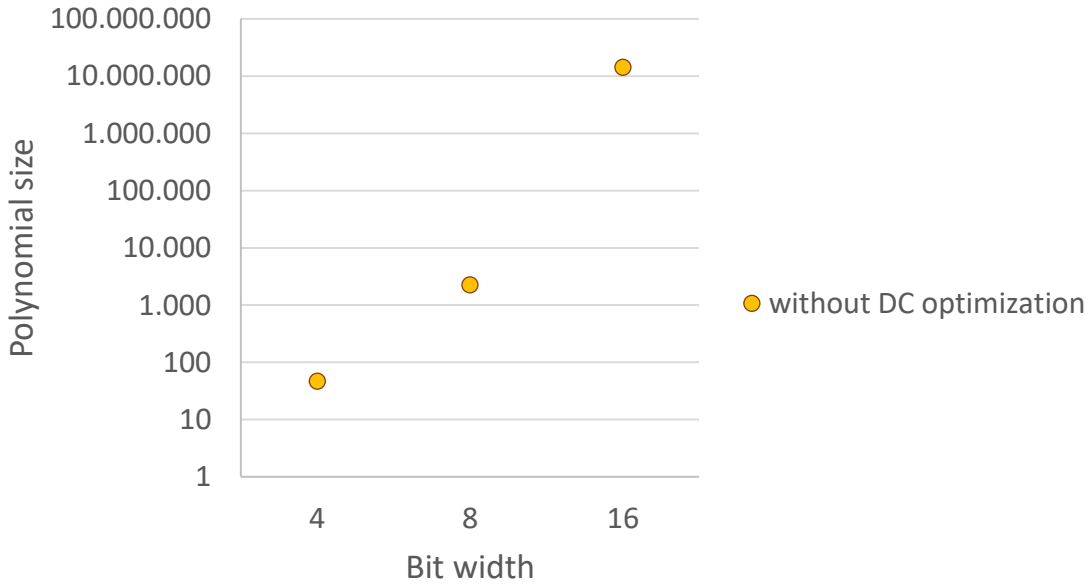
Further optimized implementation

- But we lost possibilities to optimize polynomial
- Remaining don‘t cares not strong enough to avoid exponential polynomials („blowup“ already in last stage)



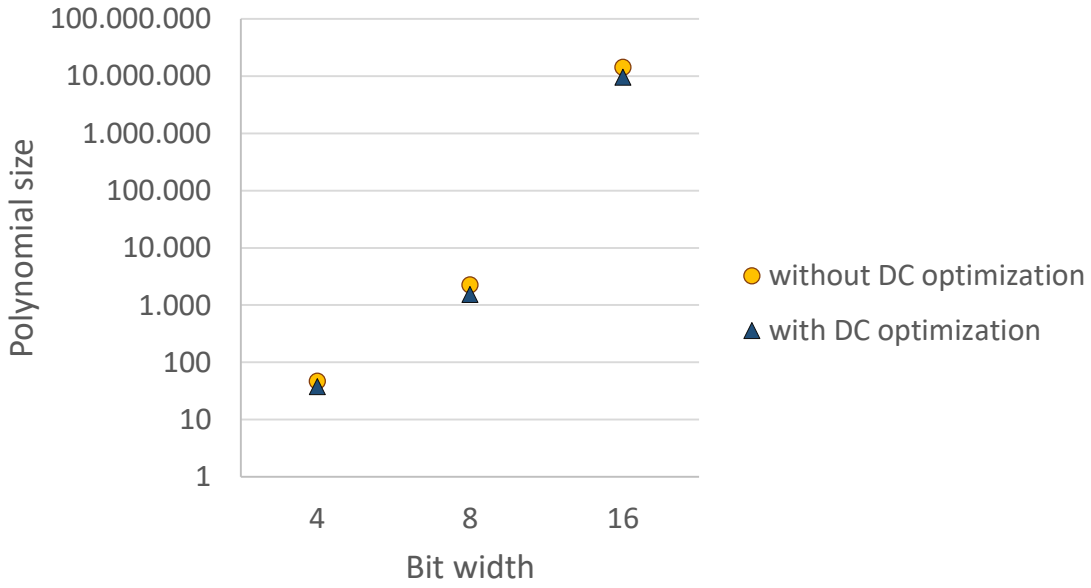
Further optimized implementation

- **Shown: polynomial size after last stage has been replaced**



Further optimized implementation

- **Shown: polynomial size after last stage has been replaced**



New idea: Extended Atomic Blocks (EABs)

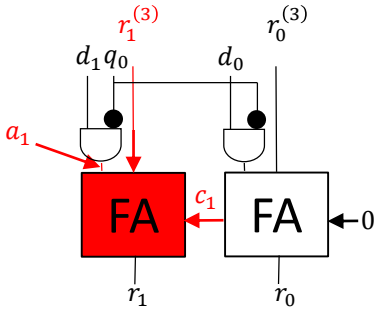
- **Extend atomic blocks with remaining gates to fanout-free cones**
- **Purpose: find more and better don‘t cares**

New idea: Extended Atomic Blocks (EABs)

- **Extend atomic blocks with remaining gates to fanout-free cones**
- **Purpose: find more and better don‘t cares**

2 DCs at FA

a_1	$r_1^{(3)}$	c_1
0	0	1
1	0	0

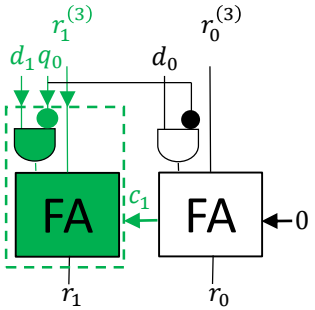


New idea: Extended Atomic Blocks (EABs)

- **Extend atomic blocks with remaining gates to fanout-free cones**
- **Purpose: find more and better don‘t cares**

2 DCs at FA

a_1	$r_1^{(3)}$	c_1
0	0	1
1	0	0

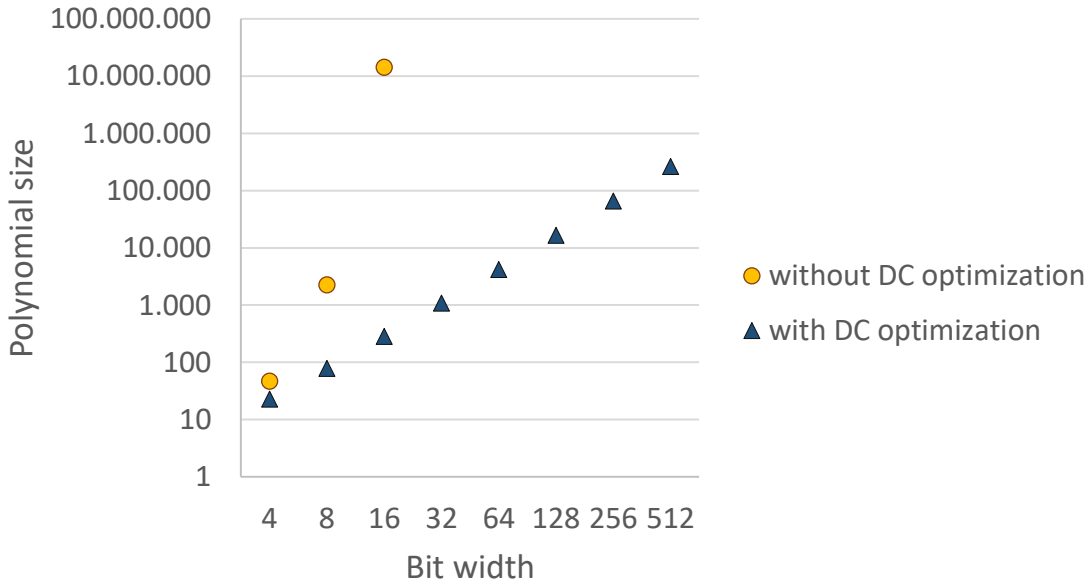


9 DCs at extended block

d_1	q_0	$r_1^{(3)}$	c_1
0	0	0	0
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	1	0	1
1	1	1	1

Further optimized implementation

- **Shown: polynomial size after last stage has been replaced**



New method: Delayed Don’t Care Optimization

- **Applying Don’t care optimization immediately only gives us a “local minimum” in the context of backward rewriting**
- **We want to apply optimizations more targeted**
  - **minimize future polynomial sizes during backward rewriting**
  - **Delay optimization to take future rewriting steps into account**

    → **Lookahead achieves optimizations in a more global context**

Delayed Don‘t Care Optimization (DDCO)

New method: Delayed Don’t Care Optimization

- **Polynomial $p = x_1x_2x_3x_4 - x_2x_3x_4$ with DC $(x_1, x_2, x_3) = (0, 1, 1)$**
- **Next replacement will be $x_4 = x_2 \cdot x_3$**

New method: Delayed Don't Care Optimization

- Polynomial $p = x_1x_2x_3x_4 - x_2x_3x_4$ with DC $(x_1, x_2, x_3) = (0, 1, 1)$
- Next replacement will be $x_4 = x_2 \cdot x_3$
- Option 1:
  - Adding DC leads to $q_1 = x_1x_2x_3x_4 - x_2x_3x_4 + v_1x_2x_3 - v_1x_1x_2x_3$
  - Optimal solution is $v_1 = 0$
  - After replacement of x_4 : $q_1' = x_1x_2x_3 - x_2x_3$

New method: Delayed Don't Care Optimization

- Polynomial $p = x_1x_2x_3x_4 - x_2x_3x_4$ with DC $(x_1, x_2, x_3) = (0, 1, 1)$
- Next replacement will be $x_4 = x_2 \cdot x_3$
- Option 1:
  - Adding DC leads to $q_1 = x_1x_2x_3x_4 - x_2x_3x_4 + v_1x_2x_3 - v_1x_1x_2x_3$
  - Optimal solution is $v_1 = 0$
  - After replacement of x_4 : $q_1' = x_1x_2x_3 - x_2x_3$
- Option 2:
  - Adding DC leads to $q_2 = x_1x_2x_3x_4 - x_2x_3x_4 + v_1x_2x_3 - v_1x_1x_2x_3$
  - After replacement of x_4 : $q_2' = (1 - v_1)x_1x_2x_3 + (v_1 - 1)x_2x_3$
  - Optimal solution is $v_1 = 1$, resulting in $q_2' = 0$

Experimental results

- **Three types of divider benchmarks:**
  - **Non-restoring dividers from [DATE‘21]: *non-restoring*$_1$**
  - **Further optimized non-restoring dividers: *non-restoring*$_2$**
  - **Restoring dividers: *restoring***

Experimental results

- **Three types of divider benchmarks:**
  - **Non-restoring dividers from [DATE‘21]: *non-restoring*$_1$**
  - **Further optimized non-restoring dividers: *non-restoring*$_2$**
  - **Restoring dividers: *restoring***
- **Six experiments:**
  - **SAT**
  - **Equivalence checking of ABC**
  - **Commercial tool**
  - **[DATE‘21]**
  - **[DATE‘21] + EABs**
  - **Our new tool = [DATE‘21] + EABs + DDCO (instead of backtracking)**

Experimental Results: run times for *non-restoring*$_1$

n	#gates	SAT	ABC	Commercial	[DATE'21]	[DATE'21] + EABs	Our new tool = [DATE'21] + EABs + DDCO
4	100	0.22 s	0.01 s	1.23 s	0.15 s	0.44 s	0.23 s
8	404	68.58 s	17.65 s	1.33 s	0.39 s	1.21 s	0.94 s
16	1,588	> 1 day	> 1 day	165.87 s	1.59 s	3.26 s	1.87 s
32	6,260	> 1 day	> 1 day	> 1 day	5.06 s	12.10 s	6.78 s
64	24,820	> 1 day	> 1 day	> 1 day	21.88 s	96.15 s	28.24 s
128	98,804	> 1 day	> 1 day	> 1 day	114.73 s	1,434.11 s	153.71 s
256	394,228	> 1 day	> 1 day	> 1 day	825.11 s	13,656.97 s	1,985.05 s
512	1,574,900	> 1 day	> 1 day	> 1 day	9,183.28 s	> 62 GB	27,370.60 s

Experimental Results: run times for *non-restoring*$_2$

n	#gates	SAT	ABC	Commercial	[DATE'21]	[DATE'21] + EABs	Our new tool = [DATE'21] + EABs + DDCO
4	96	0.23 s	0.01 s	1.21 s	0.17 s	0.26 s	0.23 s
8	400	31.83 s	16.78 s	1.86 s	2,486.89 s	0.99 s	0.95 s
16	1,584	> 1 day	> 1 day	108.23 s	> 62 GB	2.68 s	2.17 s
32	6,256	> 1 day	> 1 day	> 1 day	> 62 GB	9.36 s	7.25 s
64	24,816	> 1 day	> 1 day	> 1 day	> 62 GB	49.41 s	26.87 s
128	98,800	> 1 day	> 1 day	> 1 day	> 62 GB	340.85 s	149.75 s
256	394,224	> 1 day	> 1 day	> 1 day	> 62 GB	7,341.86 s	1,691.72 s
512	1,574,896	> 1 day	> 1 day	> 1 day	> 62 GB	> 62 GB	27,351.10 s

Experimental Results: run times for *restoring*

n	#gates	SAT	ABC	Commercial	[DATE'21]	[DATE'21] + EABs	Our new tool = [DATE'21] + EABs + DDCO
4	140	0.27 s	0.01 s	1.21 s	2.59 s	0.47 s	0.38 s
8	700	14.88 s	14.27 s	1.49 s	> 62 GB	1.77 s	1.42 s
16	3,068	> 1 day	> 1 day	16.39 s	> 62 GB	8.41 s	6.63 s
32	12,796	> 1 day	> 1 day	53,277.73 s	> 62 GB	65.99 s	29.02 s
64	52,220	> 1 day	> 1 day	> 1 day	> 62 GB	885.71 s	193.40 s
128	210,940	> 1 day	> 1 day	> 1 day	> 62 GB	> 62 GB	2,244.24 s
256	847,868	> 1 day	> 1 day	> 1 day	> 62 GB	> 62 GB	33,359.30 s
512	3,399,676	> 1 day	> 1 day	> 1 day	> 62 GB	> 62 GB	> 1 day

Conclusions and Future Work

- **Forward information propagation is crucial for successful verification of real gate level dividers**
- **Advancing Don’t Care Optimization by using Extended Atomic Blocks and Delayed Don’t Care Optimization**

- **In the future: Verification of other divider architectures as well as other arithmetic circuits**

Restoring Division

Restoring Division:

- For $j = 1$ to n do:
  - $R^{(j)} = R^{(j-1)} - D \cdot 2^{n-j}$
  - $q_{n-j} = \begin{cases} 0, \text{if } R^{(j)} < 0 \\ 1, \text{if } R^{(j)} \geq 0 \end{cases}$
  - If $R^{(j)} < 0$ then $R^{(j)} = R^{(j)} + D \cdot 2^{n-j} = R^{(j-1)}$
  - Final remainder: $R = R^{(n)}$

Non-Restoring Division

Non-Restoring Division:

- $R^{(1)} = R^{(0)} - \left(D \cdot 2^{n-1}\right)$
- $q_{n-1} = \begin{cases} 0, \text{if } R^{(1)} < 0 \\ 1, \text{if } R^{(1)} \geq 0 \end{cases}$
- For $j = 2$ to n do:
  - $R^{(j)} = \begin{cases} R^{(j-1)} + D \cdot 2^{n-j}, \text{if } q_{n-j+1} = 0 \\ R^{(j-1)} - D \cdot 2^{n-j}, \text{if } q_{n-j+1} = 1 \end{cases}$
  - $q_{n-j} = \begin{cases} 0, \text{if } R^{(j)} < 0 \\ 1, \text{if } R^{(j)} \geq 0 \end{cases}$
- **Final remainder:** $R = R^{(n)} + (1 - q_0)D$

Non-Restoring Division

Non-Restoring Division:

- $R^{(1)} = R^{(0)} - \left(D \cdot 2^{n-1}\right)$
- $q_{n-1} = \begin{cases} 0, \text{if } R^{(1)} < 0 \\ 1, \text{if } R^{(1)} \geq 0 \end{cases}$
- For $j = 2$ to n do:
  - $R^{(j)} = \begin{cases} R^{(j-1)} + D \cdot 2^{n-j}, \text{if } q_{n-j+1} = 0 \\ R^{(j-1)} - D \cdot 2^{n-j}, \text{if } q_{n-j+1} = 1 \end{cases}$
  - $q_{n-j} = \begin{cases} 0, \text{if } R^{(j)} < 0 \\ 1, \text{if } R^{(j)} \geq 0 \end{cases}$
- **Final remainder:** $R = R^{(n)} + (1 - q_0)D$

This implies

$$R^{(j)} = R^{(j-1)} - q_{n-j+1}\left(D \cdot 2^{n-j}\right) + \left(1 - q_{n-j+1}\right)\left(D \cdot 2^{n-j}\right)$$

$$= R^{(j-1)} + (1 - 2q_{n-j+1})(D \cdot 2^{n-j})$$

Another problem: Exponential backtrackings

- **Extended Atomic Blocks lead to more occurrences of DCs**

Another problem: Exponential backtrackings

- **Extended Atomic Blocks lead to more occurrences of DCs**
- **[Scholl et al., DATE‘21] used backtracking approach**
  - **If during backward rewriting DC optimization is applicable, only save backtrack point and continue rewriting**

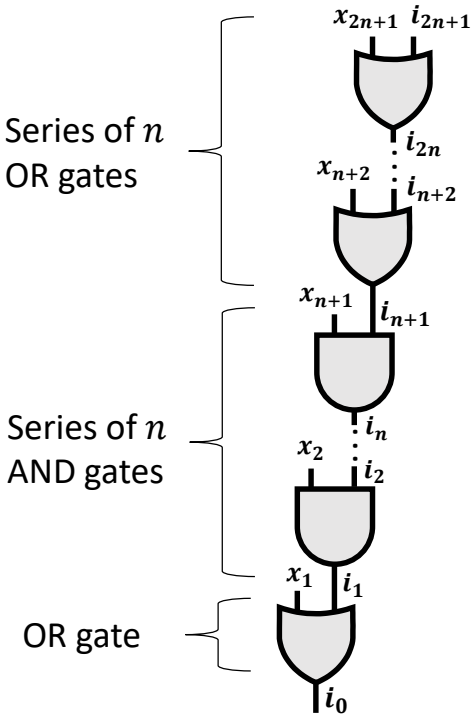
Another problem: Exponential backtrackings

- Extended Atomic Blocks lead to more occurrences of DCs
- [Scholl et al., DATE‘21] used backtracking approach
  - If during backward rewriting DC optimization is applicable, only save backtrack point and continue rewriting
  - Only if polynomial exceeds pre-defined threshold, backtrack and apply DC optimization, continue rewriting process

Another problem: Exponential backtrackings

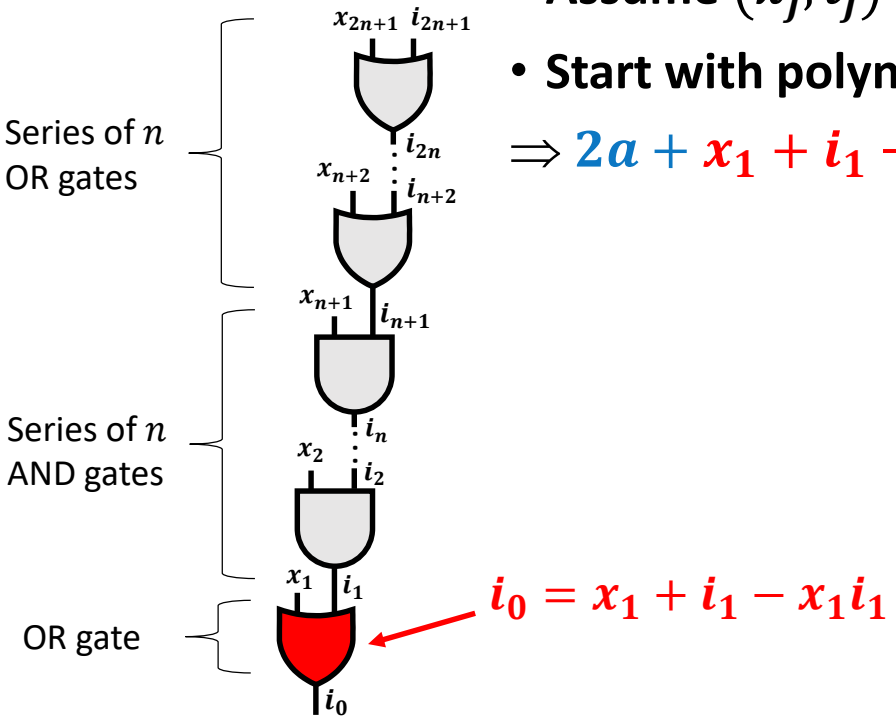
- Extended Atomic Blocks lead to more occurrences of DCs
- [Scholl et al., DATE‘21] used backtracking approach
  - If during backward rewriting DC optimization is applicable, only save backtrack point and continue rewriting
  - Only if polynomial exceeds pre-defined threshold, backtrack and apply DC optimization, continue rewriting process
  - More DCs→ more backtrack points and potentially more backtracks
  - In worst case this can lead to exponential amount of backtrackings like seen in the following example…

Another problem: Exponential backtrackings



- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

Another problem: Exponential backtrackings

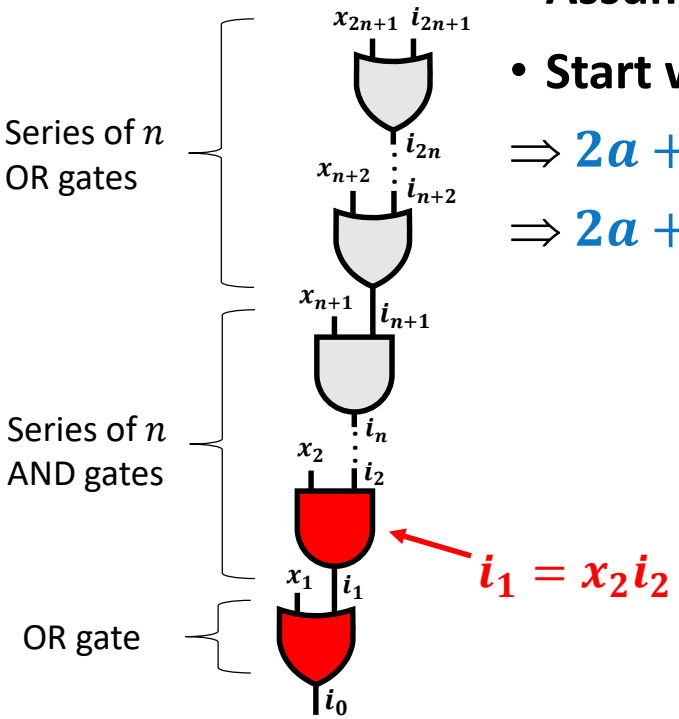


- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$ ← backtrack points

$i_0 = x_1 + i_1 - x_1 i_1$

Another problem: Exponential backtrackings



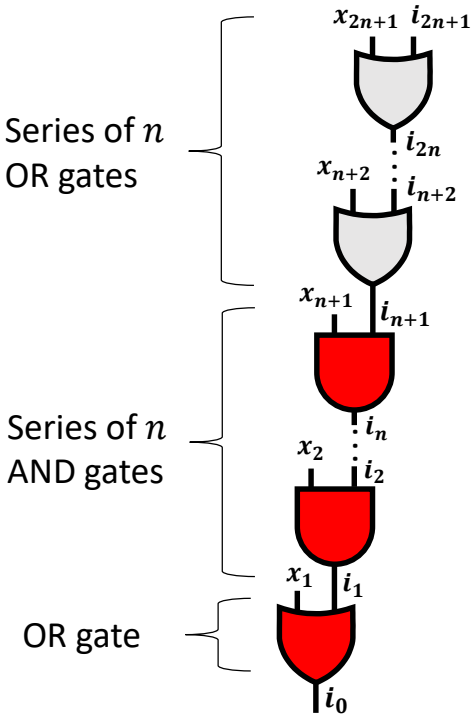
- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$

$\Rightarrow 2a + x_1 + x_2 i_2 - x_1 x_2 i_2$

backtrack points

Another problem: Exponential backtrackings



- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$

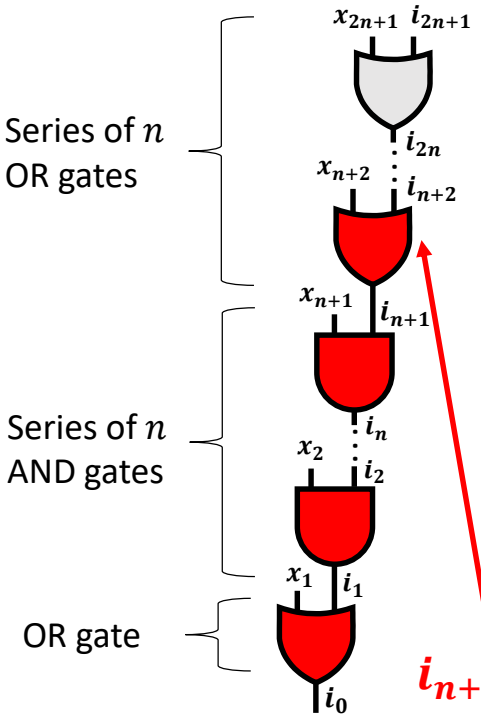
$\Rightarrow 2a + x_1 + x_2 i_2 - x_1 x_2 i_2$

$\Rightarrow$ … (replacing all AND gates)

$\Rightarrow 2a + x_1 + x_2 \dots x_{n+1} i_{n+1} - x_1 x_2 \dots x_{n+1} i_{n+1}$

backtrack points

Another problem: Exponential backtrackings



- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$

$\Rightarrow 2a + x_1 + x_2 i_2 - x_1 x_2 i_2$

$\Rightarrow$ … (replacing all AND gates)

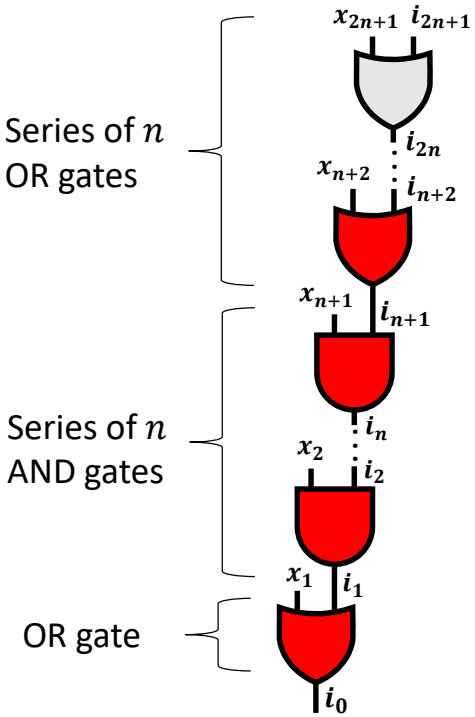
$\Rightarrow 2a + x_1 + x_2 \dots x_{n+1} i_{n+1} - x_1 x_2 \dots x_{n+1} i_{n+1}$

$\Rightarrow 2a + x_1 + x_2 \dots x_{n+2} + x_2 \dots x_{n+1} i_{n+2} - x_2 \dots x_{n+2} i_{n+2}$
$- x_1 \dots x_{n+2} - x_1 \dots x_{n+1} i_{n+2} + x_1 \dots x_{n+2} i_{n+2}$

backtrack points

$i_{n+1} = x_{n+2} + i_{n+2} - x_{n+2} i_{n+2}$

Another problem: Exponential backtrackings



- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$

$\Rightarrow 2a + x_1 + x_2 i_2 - x_1 x_2 i_2$

$\Rightarrow$ … (replacing all AND gates)

$\Rightarrow 2a + x_1 + x_2 \dots x_{n+1} i_{n+1} - x_1 x_2 \dots x_{n+1} i_{n+1}$

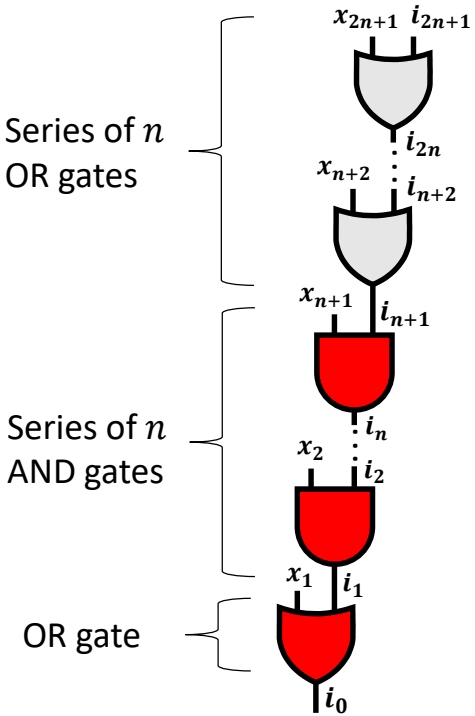
$\Rightarrow 2a + x_1 + x_2 \dots x_{n+2} + x_2 \dots x_{n+1} i_{n+2} - x_2 \dots x_{n+2} i_{n+2}$
$-x_1 \dots x_{n+2} - x_1 \dots x_{n+1} i_{n+2} + x_1 \dots x_{n+2} i_{n+2}$

$\Rightarrow$ … (replacing all OR gates leads to exponential growth)

backtrack points

backtracking starts!

Another problem: Exponential backtrackings



- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$

$\Rightarrow 2a + x_1 + x_2 i_2 - x_1 x_2 i_2$

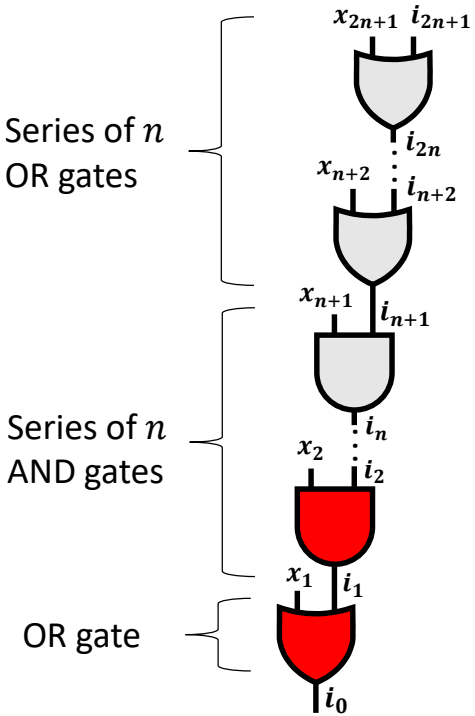
$\Rightarrow$ … (replacing all AND gates)

$\Rightarrow 2a + x_1 + x_2 \dots x_{n+1} i_{n+1} - x_1 x_2 \dots x_{n+1} i_{n+1}$

backtrack points

But DC optimization does not change the polynomial here!

Another problem: Exponential backtrackings



- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$

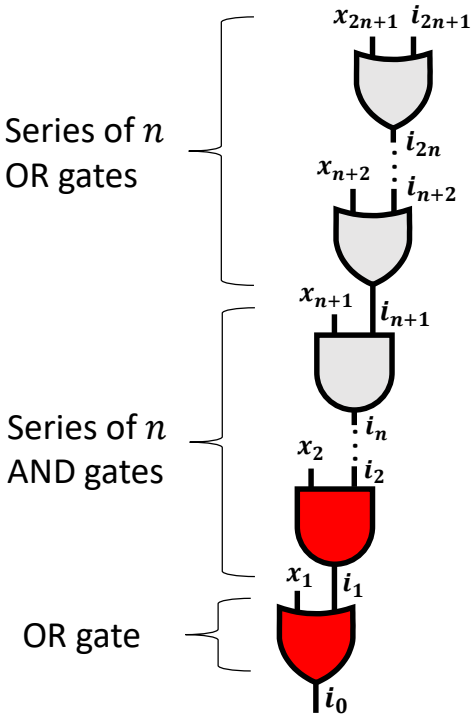
$\Rightarrow 2a + x_1 + x_2 i_2 - x_1 x_2 i_2$

$\Rightarrow$ … (replacing all AND gates)

backtrack points

But DC optimization does not change the polynomial here!

Another problem: Exponential backtrackings



- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

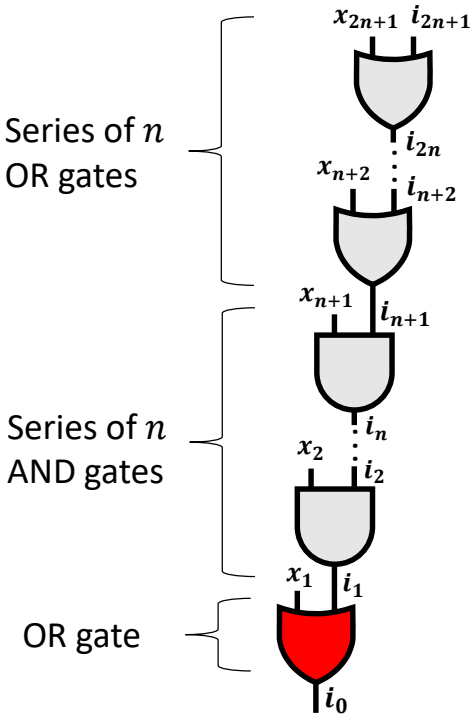
$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$

$\Rightarrow 2a + x_1 + x_2 i_2 - x_1 x_2 i_2$

backtrack points

But DC optimization does not change the polynomial here!

Another problem: Exponential backtrackings

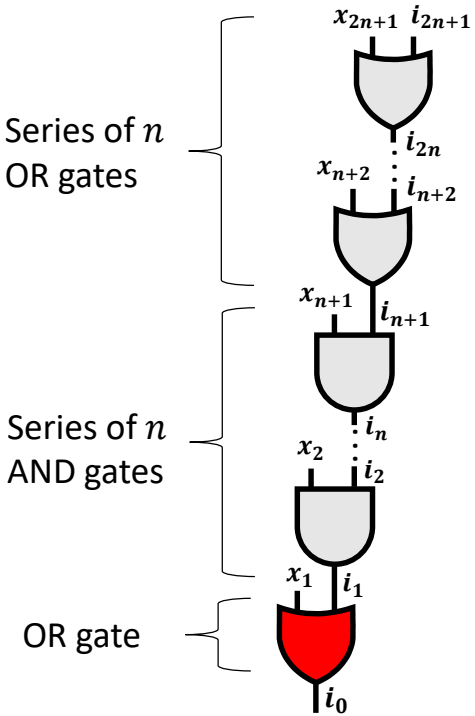


- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$ ← backtrack points

Here DC optimization is finally helpful!
Replacing the OR by a constant 1.

Another problem: Exponential backtrackings



- **Assume $(x_j, i_j) = (0, 0)$ is DC for $j = 1, \dots, n+1$**
- **Start with polynomial $SP^{init} = 2a + i_0$**

$\Rightarrow 2a + x_1 + i_1 - x_1 i_1$ ← backtrack points

Adding DC monomials:

- $2a + (1 - v_1)x_1 + (1 - v_1)i_1 - (v_1 - 1)x_1 i_1 + v_1$
- $v_1 = 1$ obviously best solution
- Resulting polynomial: $2a + 1$

Here DC optimization is finally helpful!
Replacing the OR by a constant 1.

New method: Delayed Don’t Care Optimization

- **We want to use every DC only once (to avoid previous problem)**
- **Additionally apply optimizations in a more global context**
  - **Considering future backward rewriting steps as well**

Delayed Don‘t Care Optimization (DDCO)