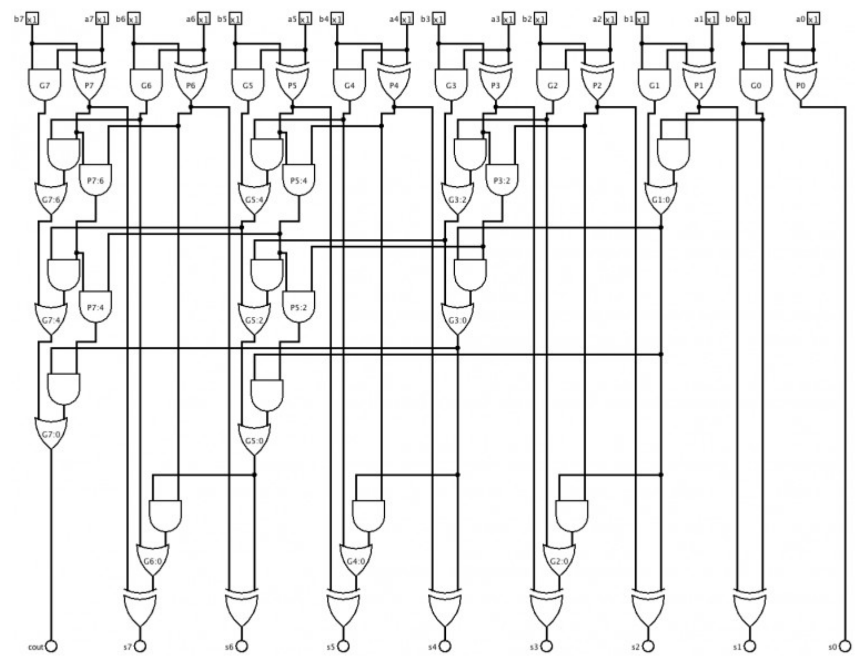
Functional Verification & Abstraction of Arithmetic Circuits

Maciej Ciesielski, Cunxi Yu

University of Massachusetts, Amherst
ciesiel@ecs.umass.edu

Arithmetic Verification

- Functional verification
  - Does the circuit implement the required arithmetic function?
  - *What function* does this circuit implement ?
- Extracting arithmetic function from gate-level implementations
  - Avoid Boolean methods,
    bit-blasting
  - Use Computer Algebra methods



Computer Algebra Approach

- Circuit specification F_{spec} and implementation B represented by pseudo-Boolean polynomials
  - Check if implementation B satisfies specification F_{spec} by reducing F_{spec} modulo B

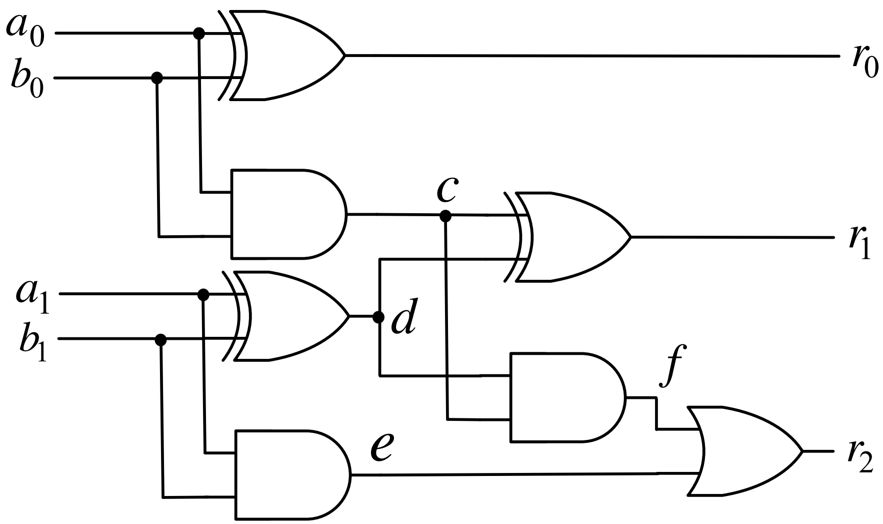
$$F_{spec} \xrightarrow{B}_{+} r$$

  - If $r = 0$, the circuit is correct
  - If $r \neq 0$, circuit *may* still be correct but need canonical *Groebner basis* (*GB*) to determine if $r = 0$
  - Difficult to compute, computationally complex
  - *GB* must include polynomials $<x^2-x>$ (for all *Boolean* signals *x*)

- Methods differ in ways they accomplish reduction
  - Arithmetic Bit-level (ABL) representation [Wienand'08, Pavlenko'11]
  - Also applied to Galois Fields (GF) [Kalla'14, TComp'15]

Computer Algebra Approach: Example

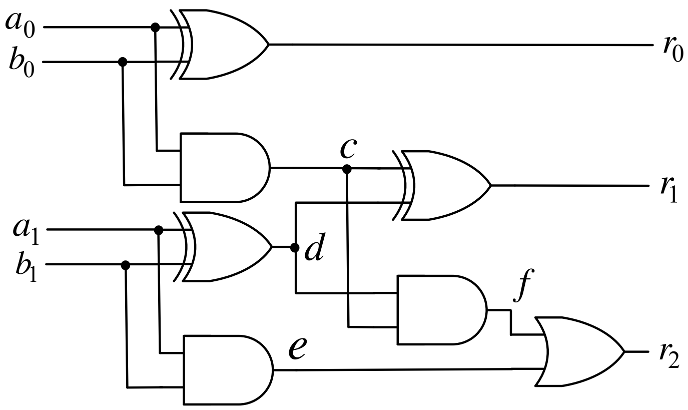
- Example: 2-bit adder
  - $F_{spec} = a_0 + b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 - r_0$
  - B = list of polynomials describing gates
  - Proof of functional correctness done by *polynomial division* (similar to our *forward rewriting*)



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

Polynomial Division (1)

- Divide polynomial F_{spec} = $a_0 + b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 - r_0$



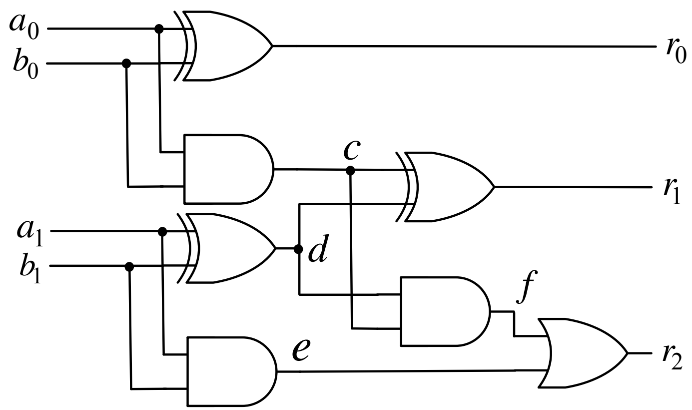
$$= -(a_0 + b_0 - 2a_0b_0) + r_0 + b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 - r_0$$
$$= 2a_0b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1$$
$$= -2(a_0b_0) + 2c + 2a_0b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1$$
$$= 2c + 2a_1 + 2b_1 - 4r_2 - 2r_1$$
$$= -2(a1 + b1 - 2a_1b_1) + 2d + 2c + 2a_1 + 2b_1 - 4r_2 - 2r_1$$
$$= 4a_1b_1 + 2d + 2c - 4r_2 - 2r_1$$
$$= -2(c + d - 2cd) + 2r_1 + 4a_1b_1 + 2d + 2c - 4r_2 - 2r_1$$
$$= 4cd + 4a_1b_1 - 4r_2$$
$$= -4(cd) + 4f + 2cd + 4a_1b_1 - 4r_2$$
$$= 4f + 4a_1b_1 - 4r_2$$
$$= -4(a_1b_1) + 4e + 4f + 4a_1b_1 - 4r_2$$
$$= 4e + 4f - 4r_2$$
$$= -4(e + f - ef) + 4r_2 + 4e + 4f - 4r_2$$
$$= 4ef$$

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

Non-zero residual ! Is circuit correct ?

Polynomial Division (2)

- Continue dividing residual polynomial { *4ef* }



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

$4ef$

$= 4e(cd)$

$= 4(a_1b_1)(cd)$

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

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

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

$= 4(0)\ (a_0b_0)$

$= 4ef = 0$

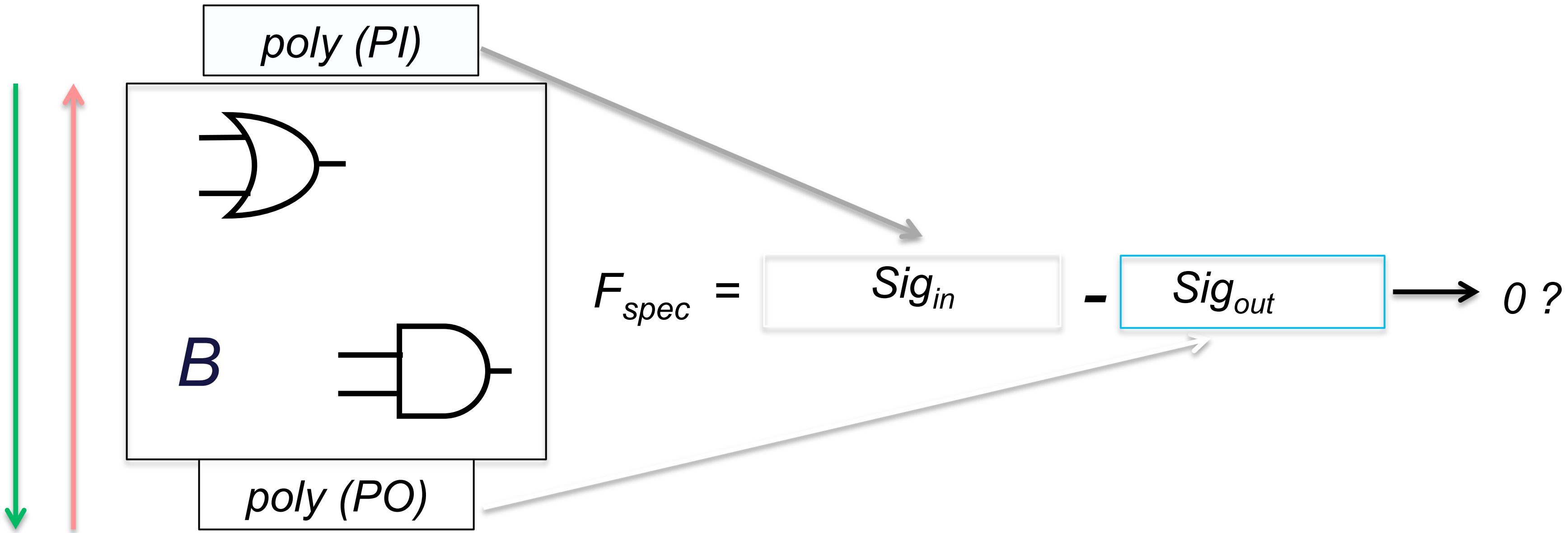
- Hence (F_{spec} mod B) = 0 , the circuit correctly implements a 2-bit adder.
- But many dividing steps are needed

Computer Algebra Approach - summary

- Another way of looking at the problem:

  Instead of reducing F_{spec} modulo *B*, we can

  - Rewrite $Sig_{in} \rightarrow Sig_{out}$ (forward rewriting), or
  - Rewrite $Sig_{out} \rightarrow Sig_{in}$ (backward rewriting)



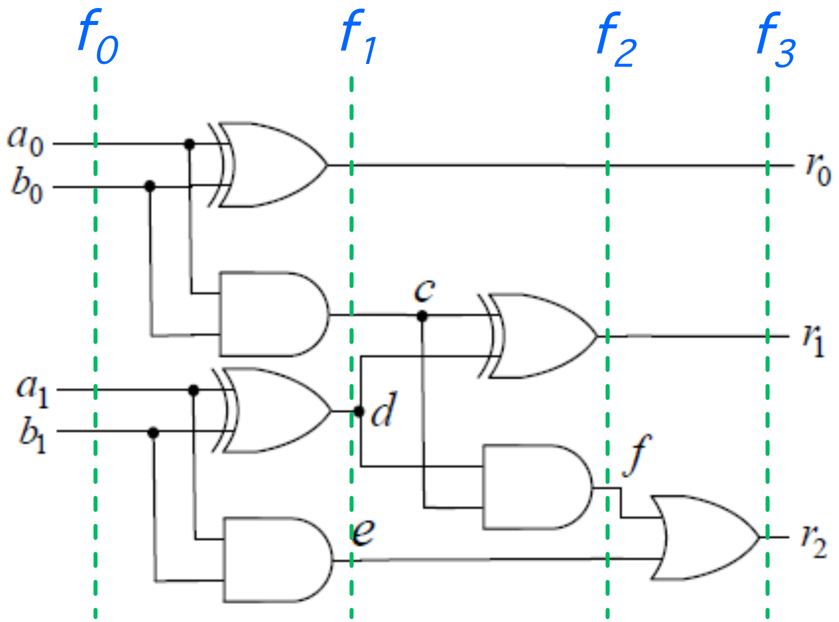
Backward Rewriting

- Replace gate output by its equation
  - Backward *symbolic simulation*
  - No residual expression *(r)* !
  - But … the expression can explode

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

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

$$f_1 = 4e + 4(cd) - 4e(cd) + 2(c + d - 2cd) + r_0$$
$$= 4e + 2c + 2d + r_0 - 4ecd$$



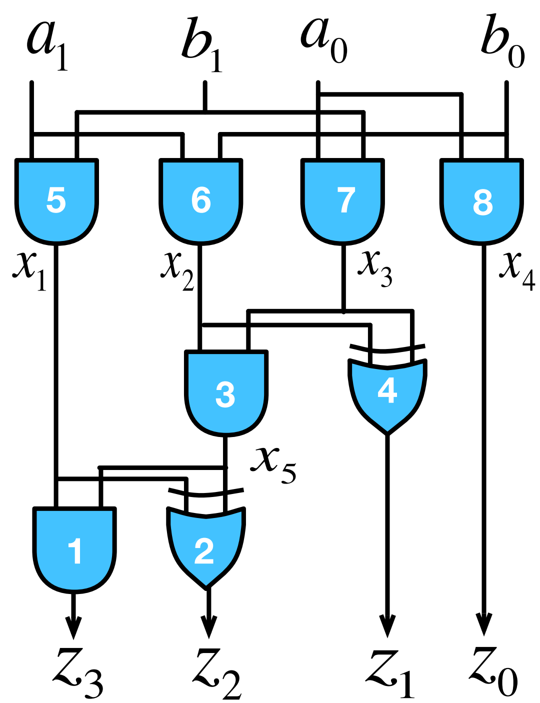
$$f_0 = 4(a_1b_1) + 2(a_0b_0) + 2(a_1 + b_1 - 2a_1b_1) + (a_0 + b_0 - 2a_0b_0) - 4(a_1b_1)(a_0b_0)(a_1 + b_1 - 2a_1b_1)$$

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

It matches the specification:
→ *circuit is correct*

Backward Rewriting - ordering

- How efficient is it ?
  - Simpler than forward rewriting (*no residual*)
  - Cancellations happen during rewriting
  - But expressions may explode



$8z_3^{(1,2)} = 8x_1x_5$ $\quad$ $4z_2^{(1,2)} = 4x_1 + 4x_5 - 8x_1x_5$

$8z_3^{(3,4)} = 8x_1x_2x_3$ $\quad$ $4z_2^{(3,4)} = 4x_1 + 4x_2x_3 - 8x_1x_2x_3$

$8z_3^{(5,6)} = 8a_1b_1a_0b_0$ $\quad$ $4z_2^{(5,6)} = 4a_1b_1 + 4a_1a_0b_1b_0 - 8a_1a_0b_1b_0$

$$F_{spec} = 8z_3 + 4z_2 + 2z_1 + z_0$$

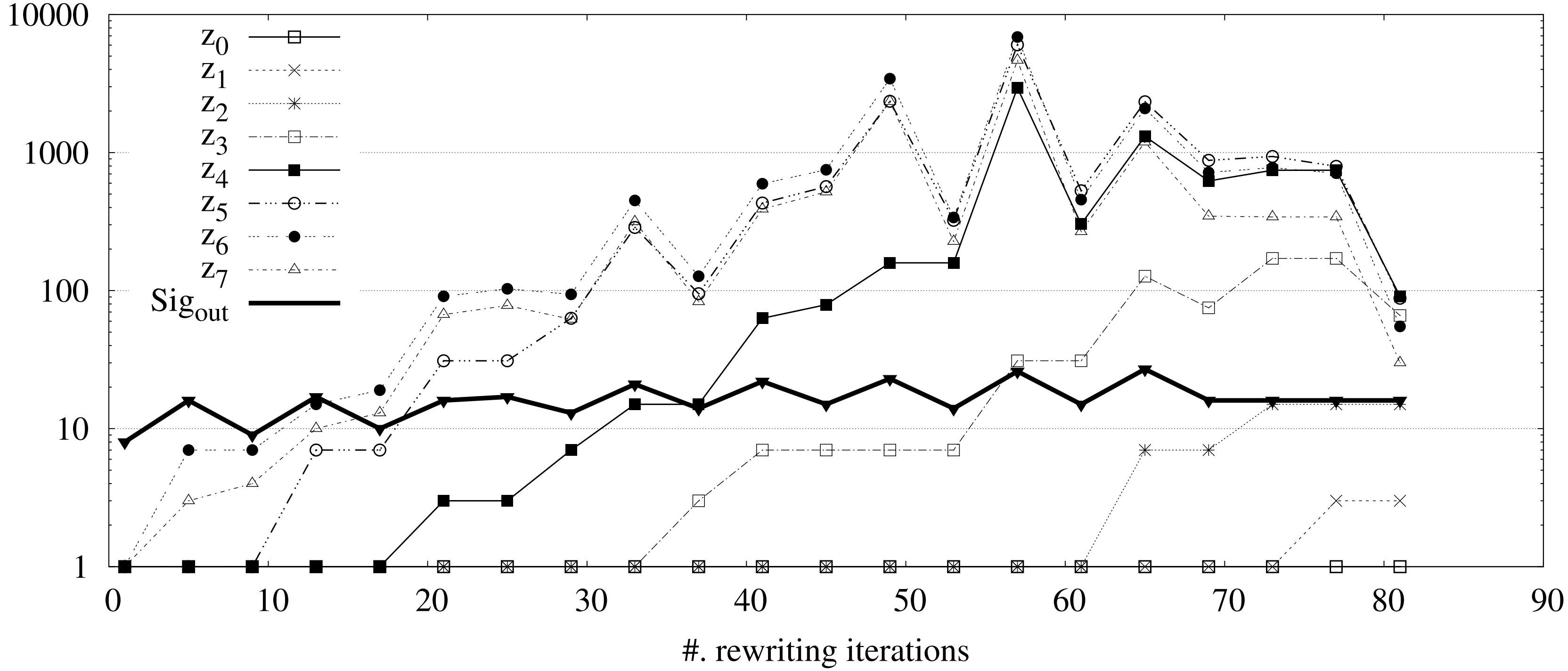
$$F_{spec}^{(1,2)} = 4x_1 + 4x_5 + 2z_1 + z_0$$

$$F_{spec}^{(3,4)} = ...$$

Still Linear !!

Backward Rewriting - Analysis

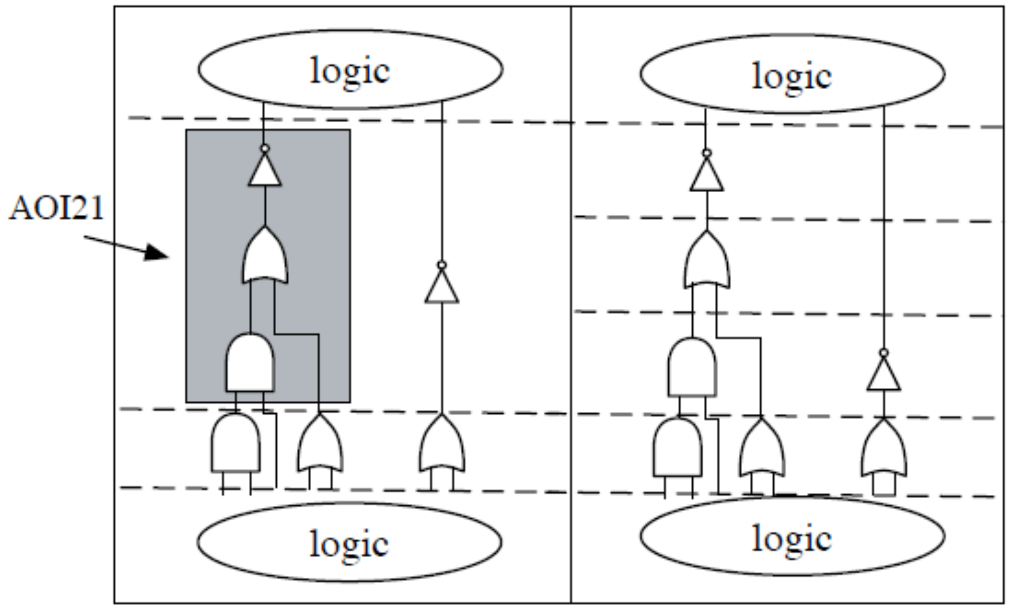
- Example: 4-bit CSA-multiplier
  - Compare size of intermediate expressions
  - Individual bits vs. an entire expression (Sig_{out} → …. → Sig_{in})



Backward Rewriting - Summary

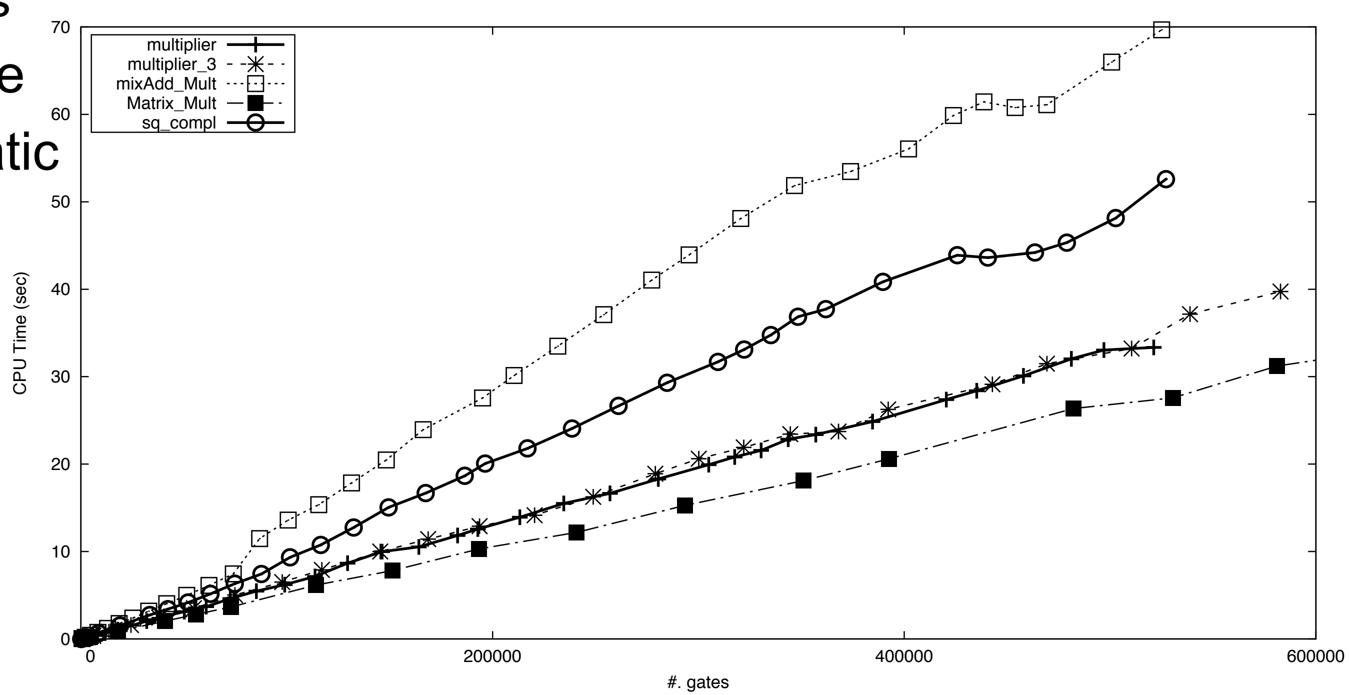
- No residual expression !
  - But the cut expression can explode (*fat belly* issue)
  - Choice and ordering of cuts during rewriting affects performance

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



Backward Rewriting - Results

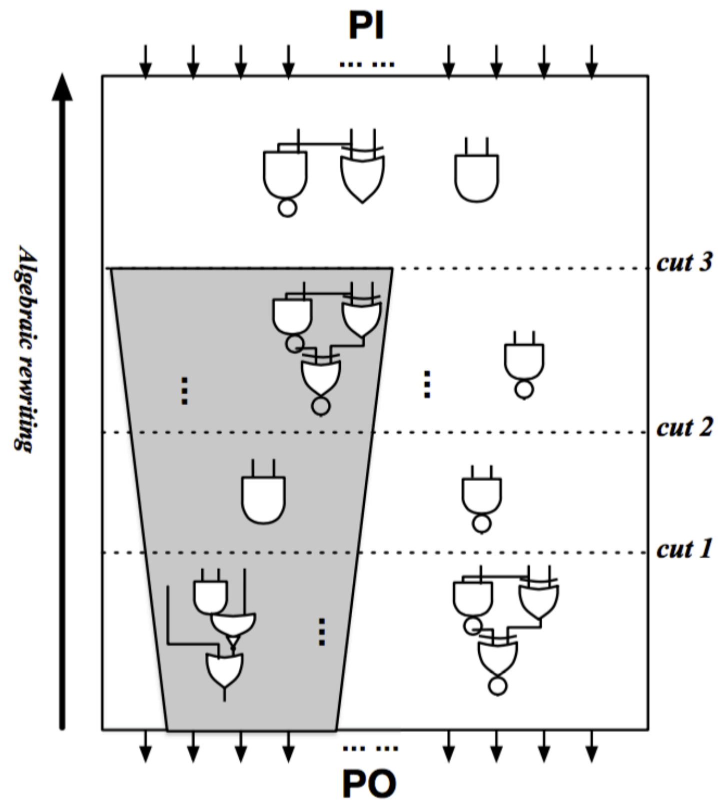
- Performance for original & lightly synthesized designs
  - Synthesis performed by ABC *"resyn"*
  - Verified designs
    - Multipliers, matrix multiplier, *A*B+C*, squarer, etc.
    - Up-to 5 million gates
    - 256+ bit-widths
  - ~Linear CPU time
  - Memory : quadratic in # gates



Functional Abstraction – *Spectral Method*

- Extract arithmetic functions with unknown boundaries
  - Assume that PO boundary is known but no boundary for PIs
  - Backward rewriting

- Spectral method
  - Examine distribution of weights (coefficients) of polynomial terms during rewriting
  - Determine arithmetic function corresponding to a sub-expression
  - based on its coefficients



Arithmetic Spectrum – Adder

- *n*-bit adder

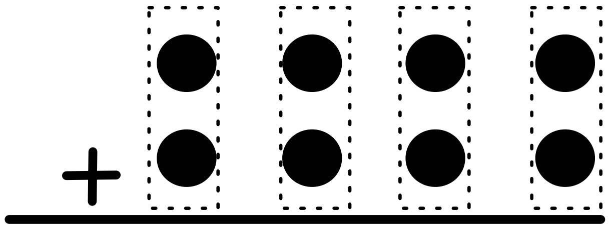
i = bit position of result
C(i) = 2^i, coefficient a bit *i*
N(i) = # terms with coeff *C(i)*

$$A = a_0 + 2a_1 + 4a_2$$

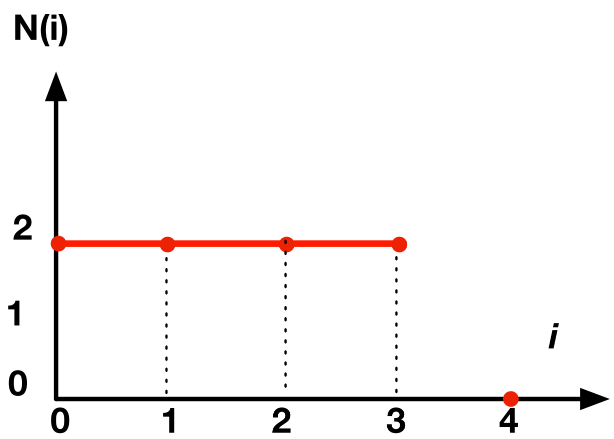
$$B = b_0 + 2b_1 + 4b_2$$

$$A + B = a_0 + 2a_1 + 4a_2 + b_0 + 2b_1 + 4b_2$$

$$N(i) = 2,\ i \in (0,1,2...,n-1)$$



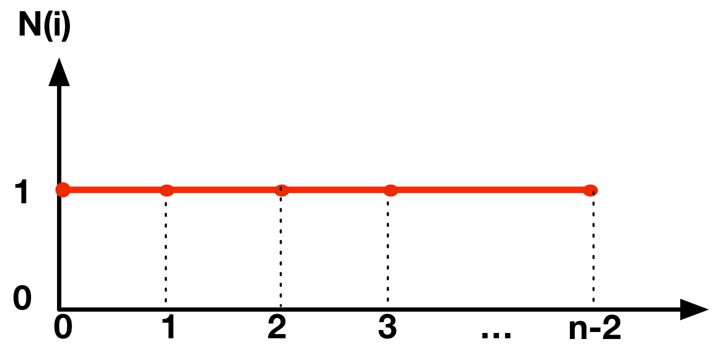
	i=3	*i=2*	*i=1*	*i=0*
N(i)	***2***	***2***	***2***	***2***
C(i)	***8***	***4***	***2***	***1***



Arithmetic Spectrum – Linear Functions

- *n*-bit word *shifting*

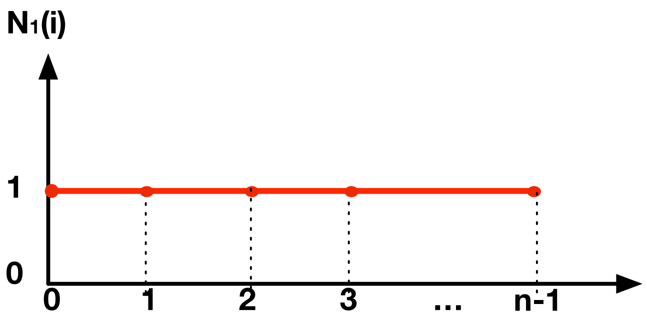
$$N(i) = 1,\ i \in (0,1,2\ldots,n-2)$$

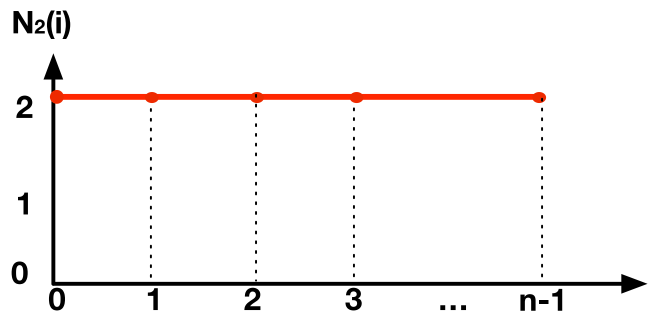


- *Multiplexer*

$$Z_{MUX} = W - 2s \cdot W$$

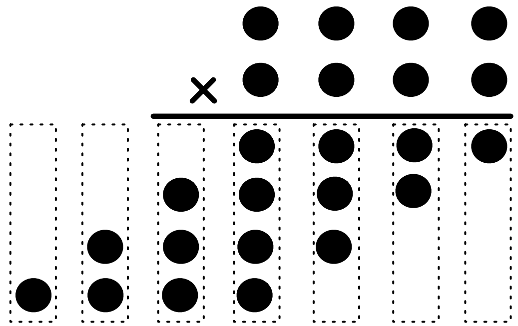
$$N_1(i) = 2, N_2(i) = 1,\ i \in (0,1,2,\ldots,n-1)$$



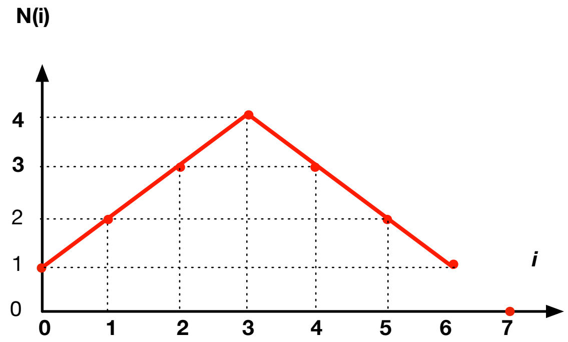


Spectrum – Multiplier *(nonlinear)*

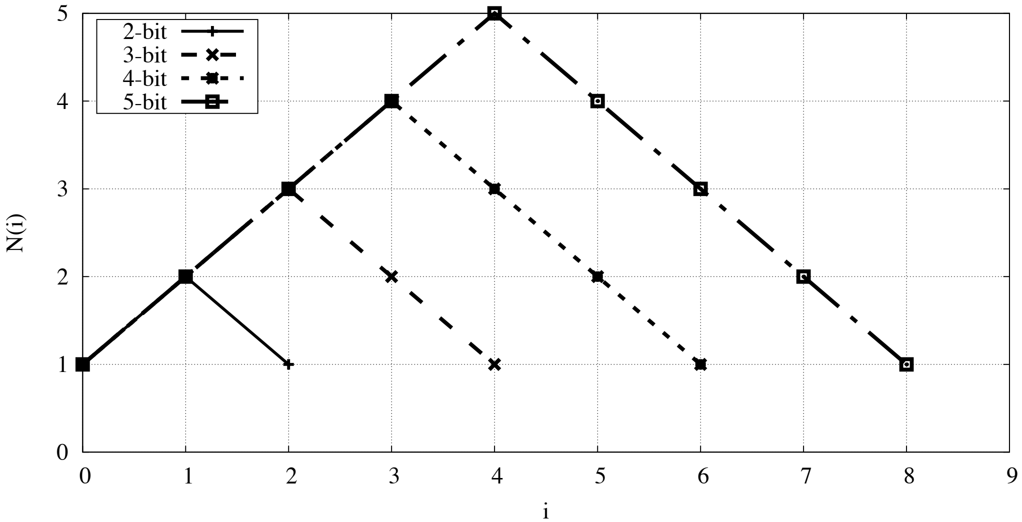
- *Multiplier*



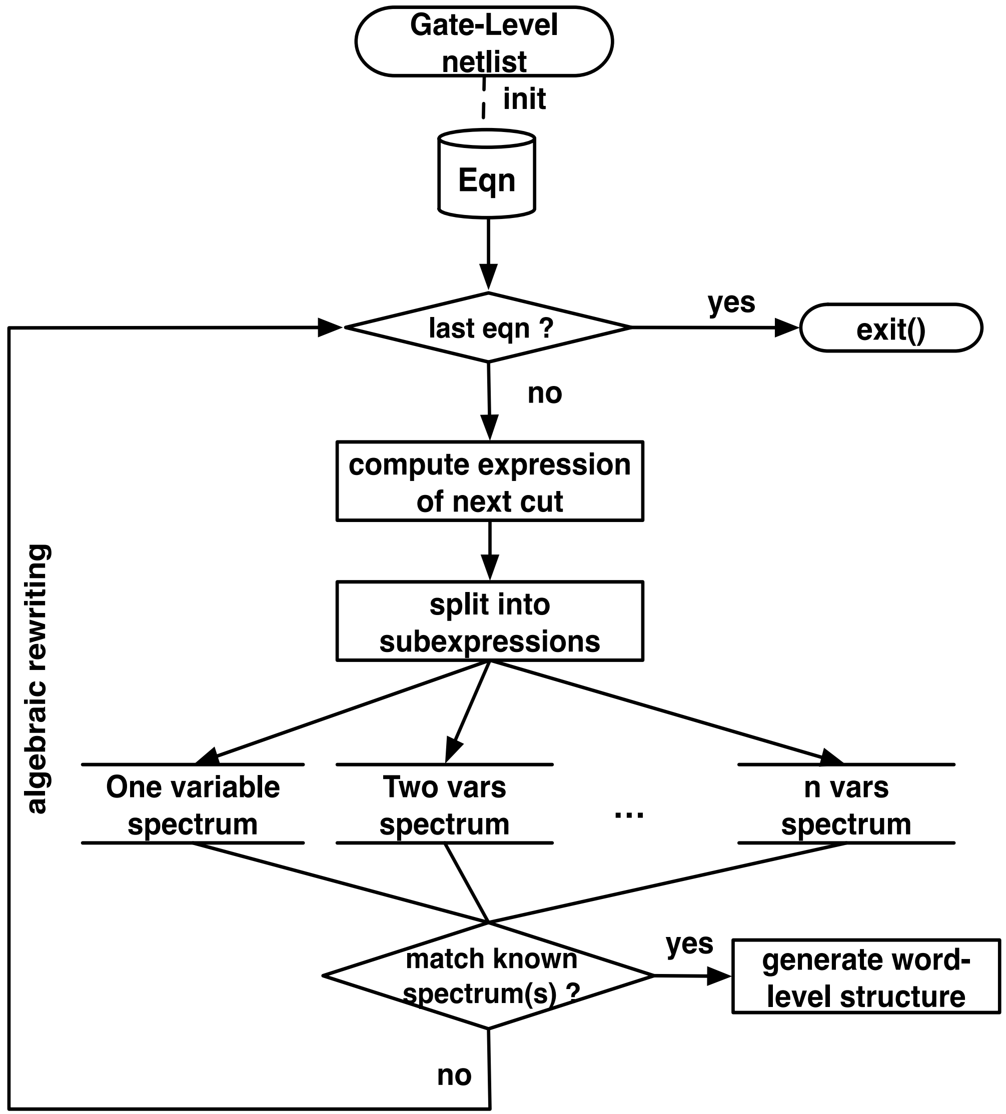
	i=6	i=5	i=4	i=3	i=2	i=1	i=0
N(i)	1	2	3	4	3	2	1
C(i)	64	32	16	8	4	2	1



$$N_i = \begin{cases} i+1 & \text{if } i \leq n/2-1 \\ n-i-1 & \text{if } i \geq n/2 \end{cases}$$



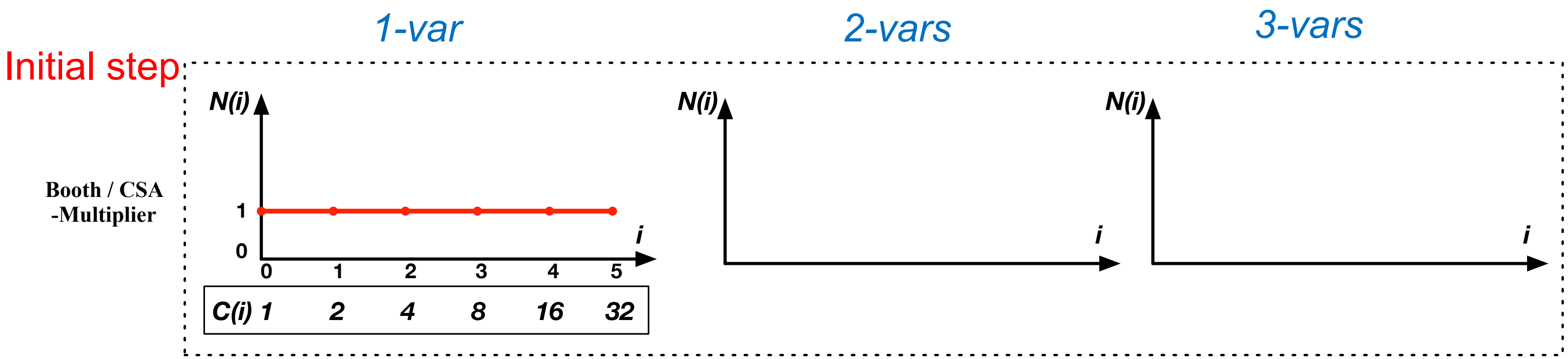
Abstraction – Spectral Method



Other Arithmetic Spectra

- Does the circuit structure affect the spectrum?
  - *No*, it affects rewriting performance, but *not* the spectrum

- Example: 3-bit *Booth m*ultiplier vs. *CSA* multiplier
  - Diagram: single-, double-, triple-variable terms (left to right)



$$F_{spec} = z_0 + 2z_1 + 4z_2 + 8z_3 + 16z_4 + 32z_5$$

Expression with 1-variable terms

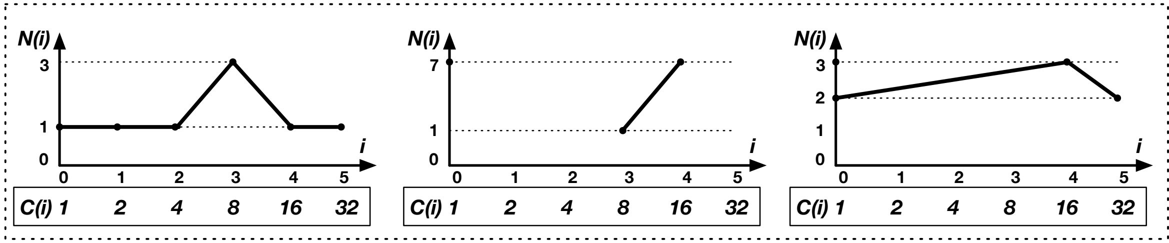
Spectrum – *Booth and CSA Multiplier*

- *3-bit Mults*

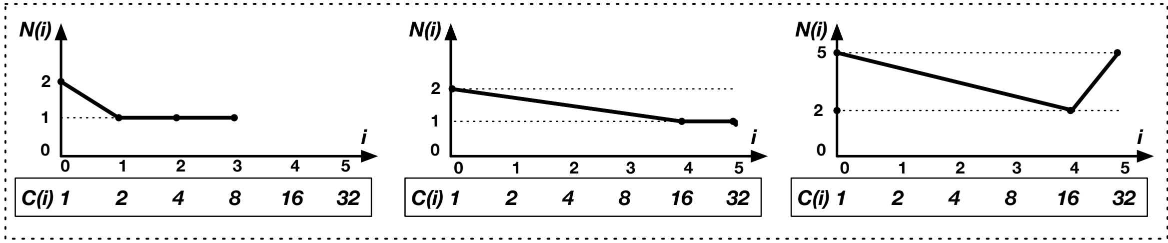
Rewriting progress

20 %

CSA Multiplier

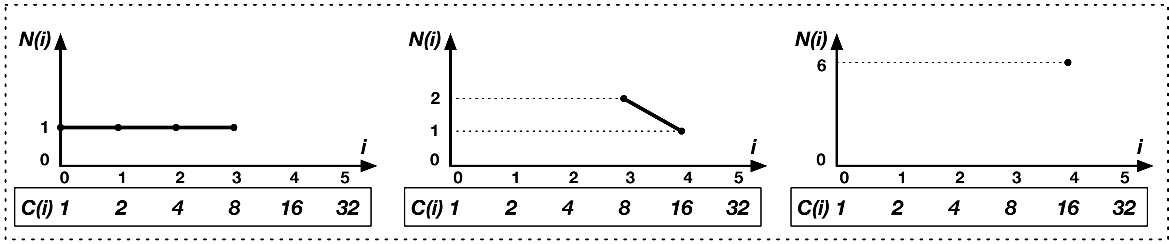


Booth Multiplier

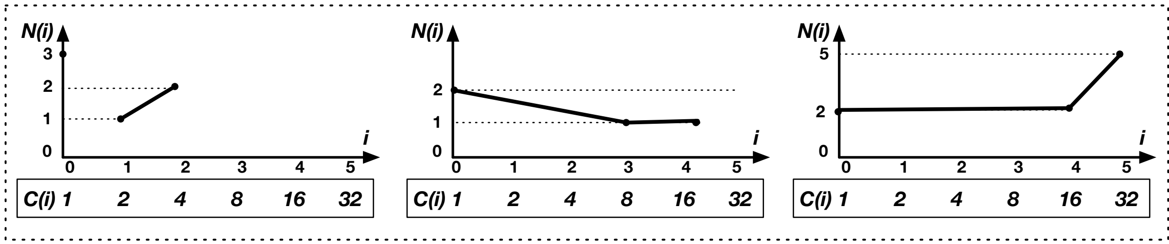


40 %

CSA Multiplier



Booth Multiplier



Spectrum – *Booth and CSA Multiplier*

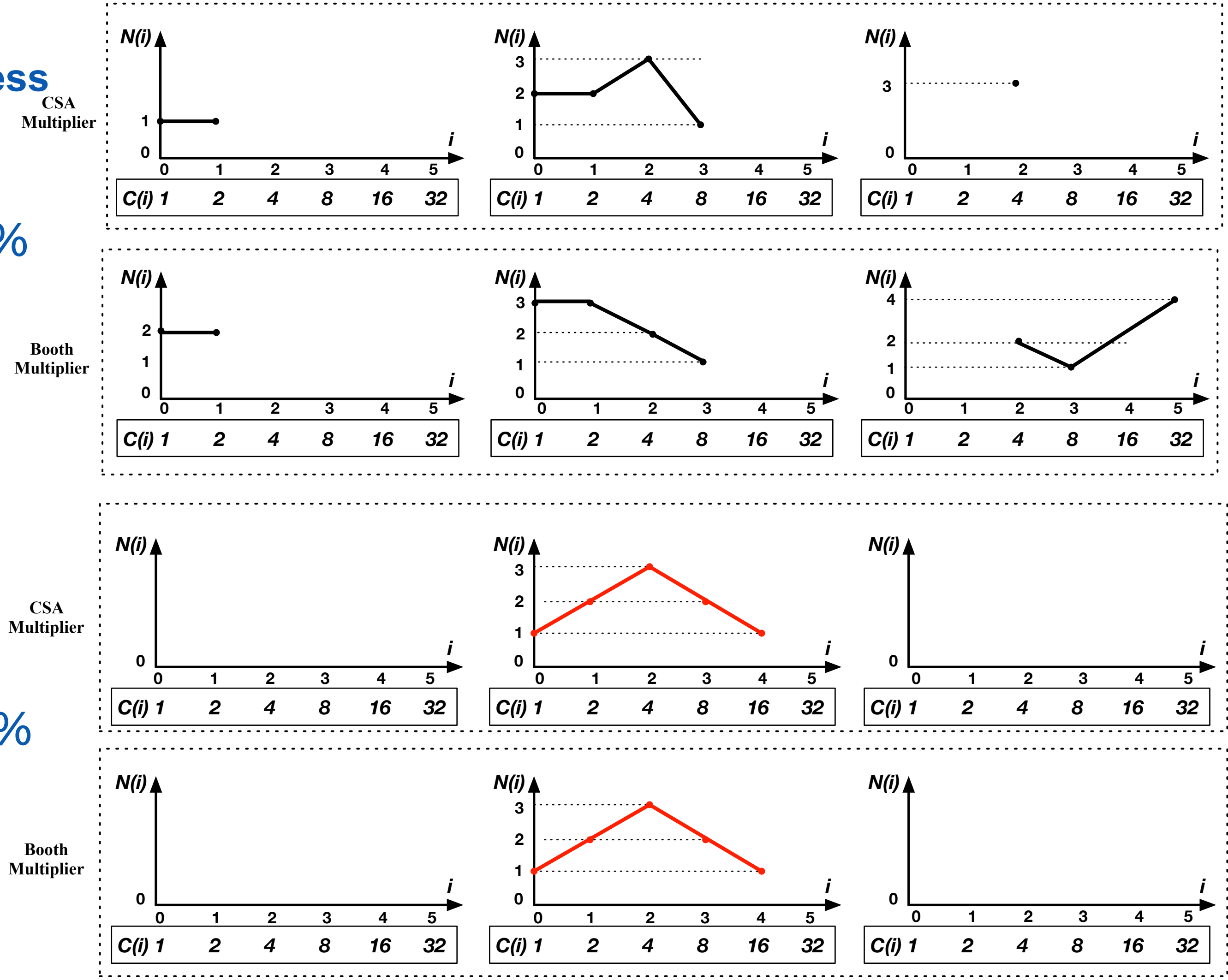
- *3-bit Mults*

Rewriting progress

80 %

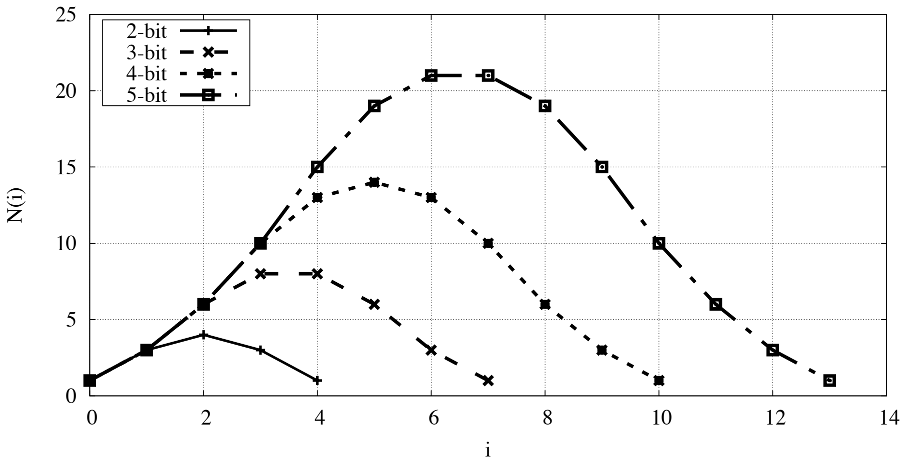
100 %

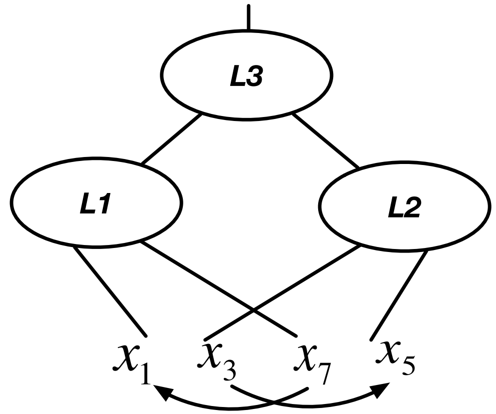
Non-linear word detected !



Spectrum – 3-operand Multiplier

- Spectrum for *arbitrary* arithmetic function ?
  - 3-operand multiplier *A*B*C*
  - *Addition, multiplication* or combination
    - Multiply-Accumulator (MAC)





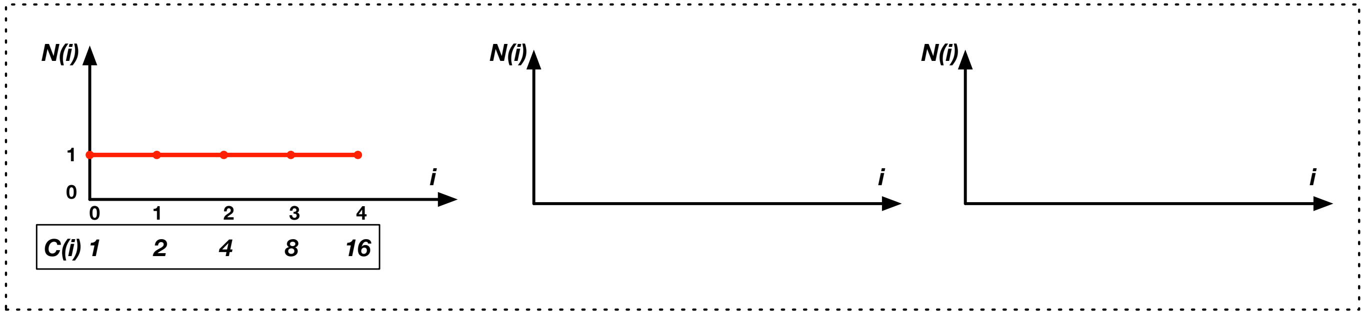
$\ldots + 4\, x_1x_5 + 2x_1x_3 + x_3x_7 + 2x_5x_7 + \ldots$
$= \ldots + (2x_1+x_7)(2x_5+x_3) + \ldots$

- Word abstraction from expression
  - Pairing signals
  - Need topological analysis

Arithmetic Spectrum - MAC

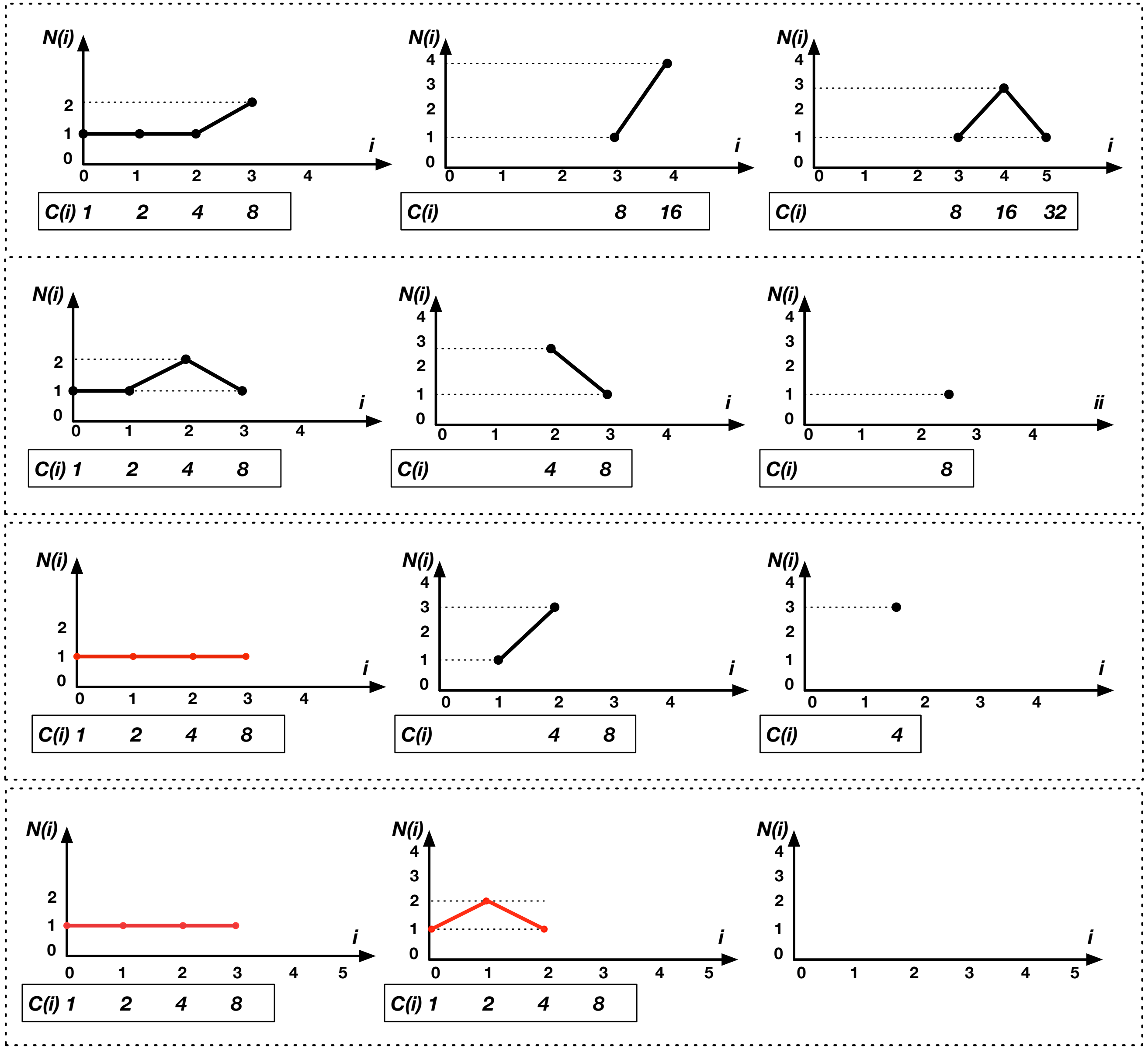
- Multiply Accumulator (*F* = *A* * *B* + *C*)
  - Can we identify the adder and the multiplier ?
    - Adder or multiplier may not exist after synthesis
  - We can tell that there is an *addition* and *a multiplication*
    - Identify the upper boundary of function *F*
  - What we cannot do: identify the adder or multiplier
    - Structural level
- Example : *MAC*
  - 2-bit multiplier following 4-bit adder

Initial step



Arithmetic Spectrum - MAC

- *MAC*

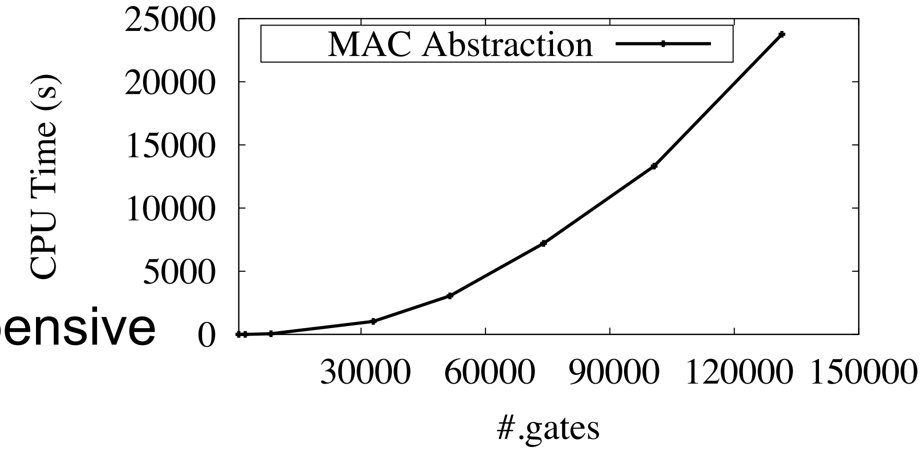


Linear word!

Addition and multiplication detected

Functional Abstraction - Results

- 8- to 128-bit MAC
- Limitations
  - Need to know *output bits*
  - Computing *spectrum* is expensive



size k	*#. gates*	*pre-ordering*	*Addition*	*Multiplication*	*Total*
8	529	0.01 s	0.01 s	0.22 s	0.24 s
16	2089	0.01 s	0.03 s	2.71 s	2.75 s
32	8281	0.03 s	0.11 s	50.7 s	51.00 s
64	32953	0.07 s	0.47 s	1028.9 s	1029 s
80	51432	0.12 s	0.76 s	3049.5 s	3050 s
96	74008	0.15 s	1.27 s	2.0 hrs	2.0 hrs
112	100681	0.21 s	1.62 s	3.7 hrs	3.7 hrs
128	131465	0.27 s	2.23 s	6.6 hrs	6.6 hrs

TABLE I. WORD-ABSTRACTION EVALUATION USING MULTIPLY-ACCUMULATOR S = SECONDS, HRS = HOURS

Thank You !