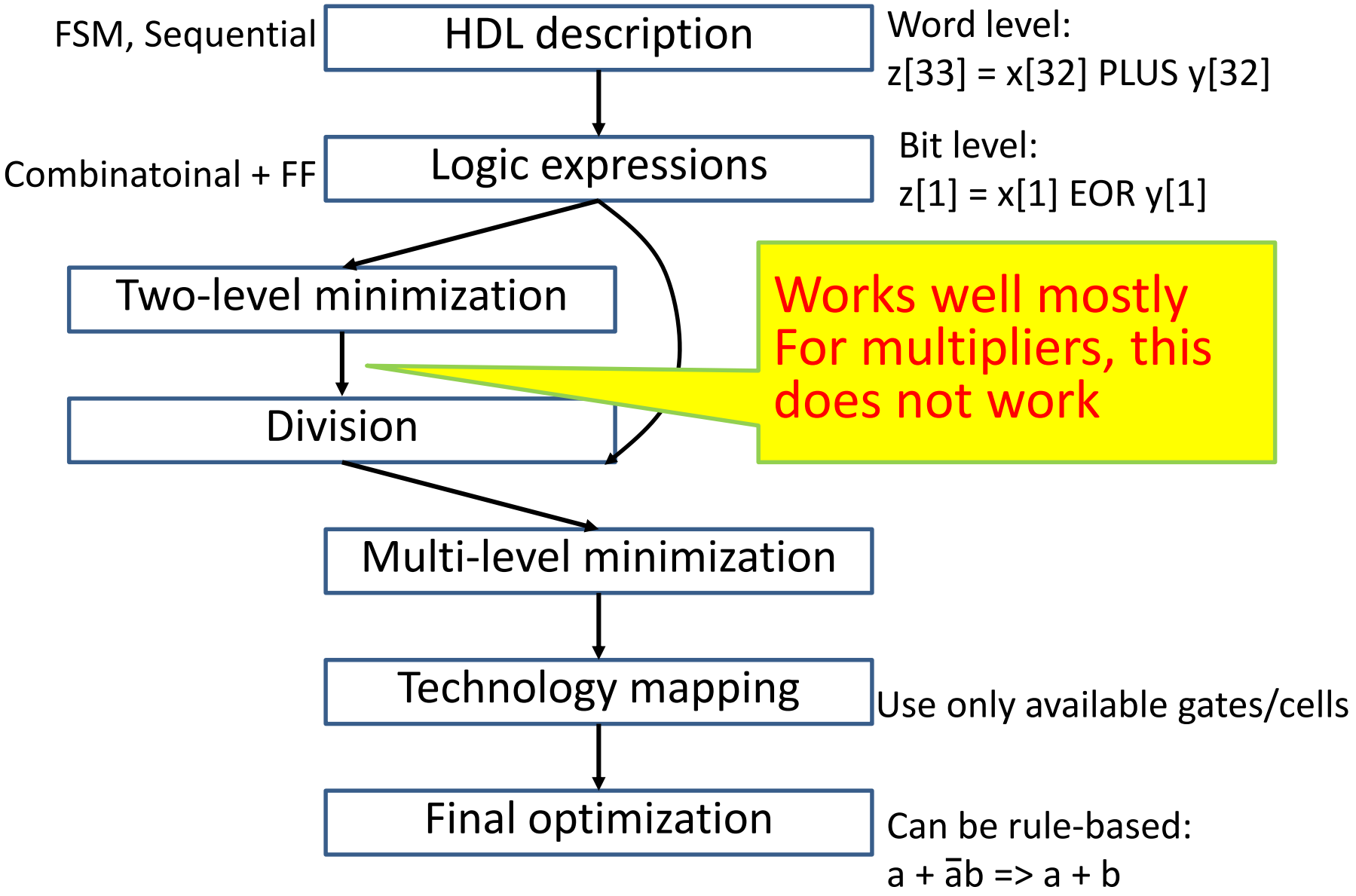
Basic and Advanced Researches in Logic Synthesis and their Industrial Contributions

Masahiro Fujita
VLSI Design and Education Center
University of Tokyo

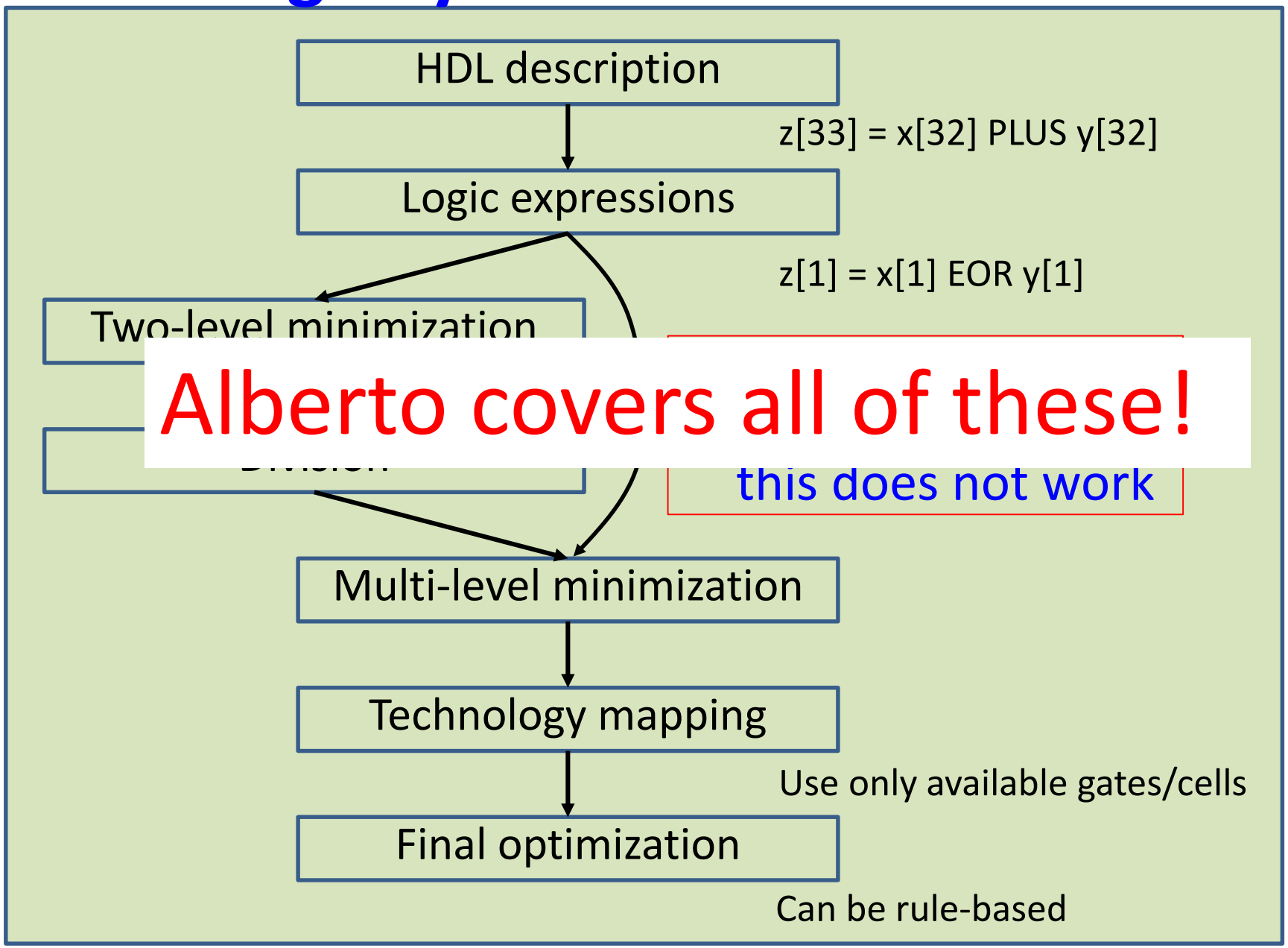
Outline

- Logic synthesis flow
  - Automatic process except for complication arithmetic circuits
  - Two-level logic minimization
    - Unate recursive paradigm by case splitting
  - Multi-level logic optimization
    - How to deal with don't cares coming from the topology
  - Synthesis from FSM
    - Various sequential optimization techniques
- Partial logic synthesis
  - Engineering change order and logic debugging
- Discussing hardware design flow
  - Importance on logic synthesis
- Application of partial logic synthesis to automatic synthesis of parallel/distributed computing
  - Solved by SAT solvers with implicit and exhaustive search
  - Use human induction to generalized the solutions

Logic synthesis flow

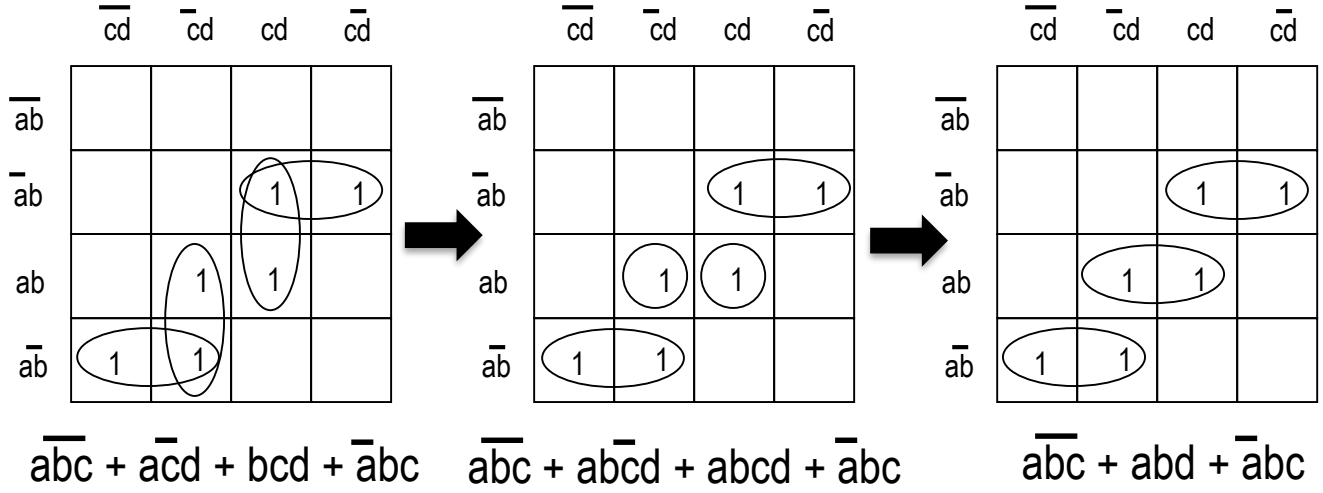


Logic synthesis flow



Two level minimization

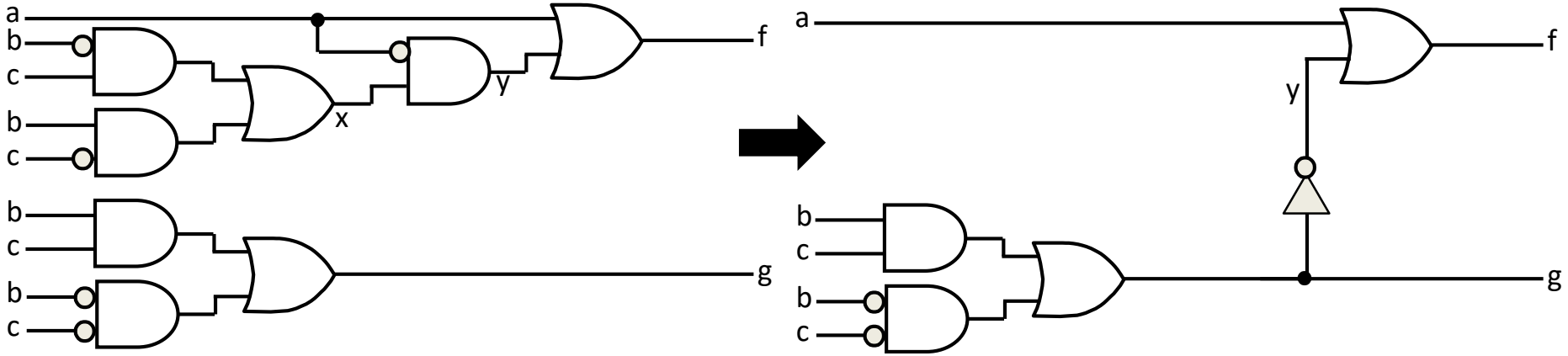
- Human can do with Karnaugh map up to 4 variables
- Espresso2 algorithm
  - Based on iteration of redundancy removal, reduce, and expand



  - How to implement these operations
    - Unate functions are easy to analyze
    - Based on unate recursive paradigm by case splitting
  - Logic expressions with more than 1,000 variables can be minimized

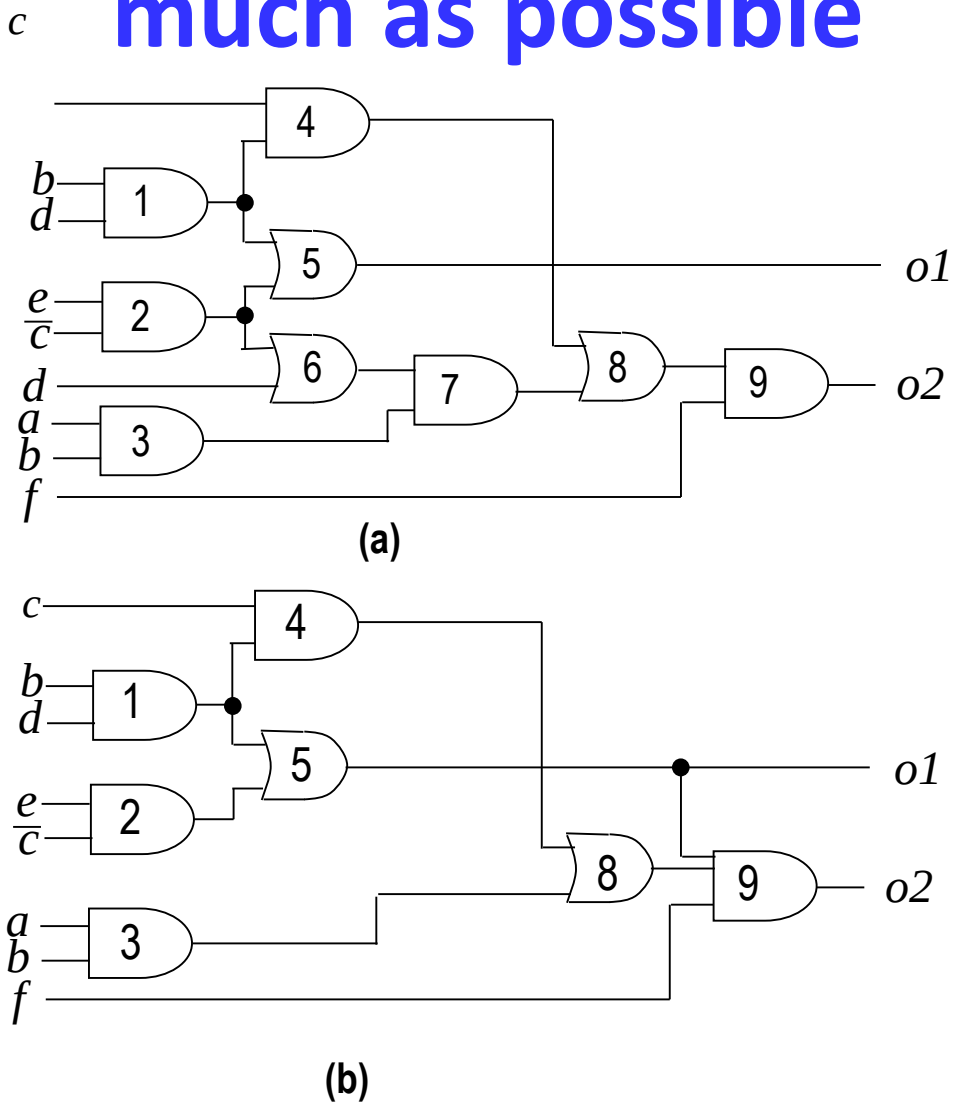
Multi level logic minimization

- Repetition of local transformations
  - Global transformation is too computation intensive
- How to check if each transformation is valid
  - Do not use don't care: Not good in quality
  - Use local don't care: Good and efficient mostly
  - Use global don't care: Too much computation

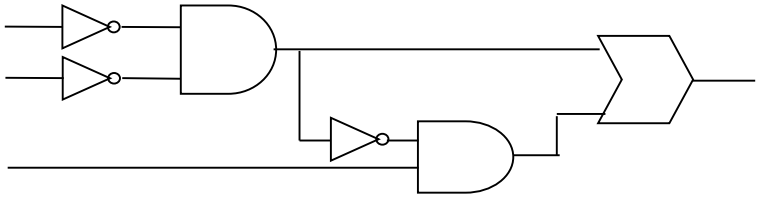


- Not work well for complicated arithmetic circuits
  - Multipliers synthesized from truth tables can be over 100 times larger than manual designs!

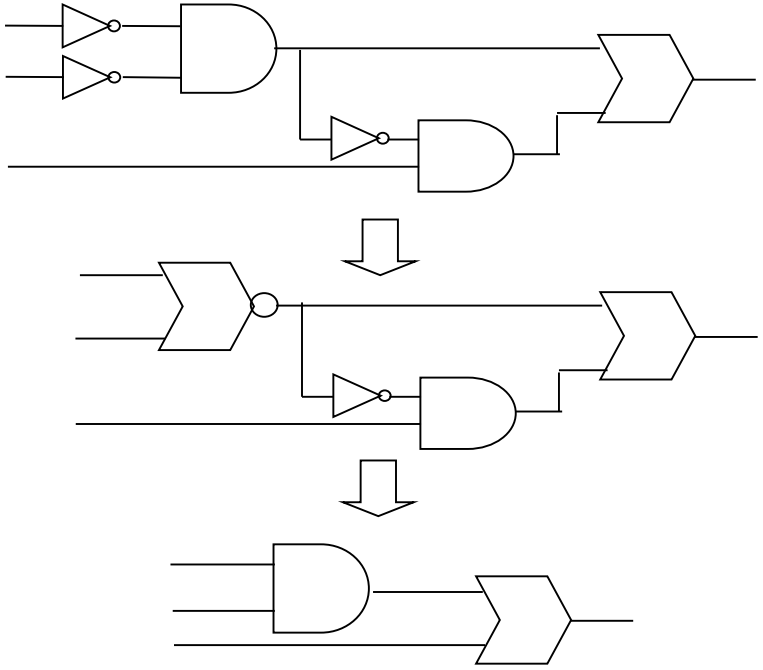
Apply logic minimization methods as much as possible



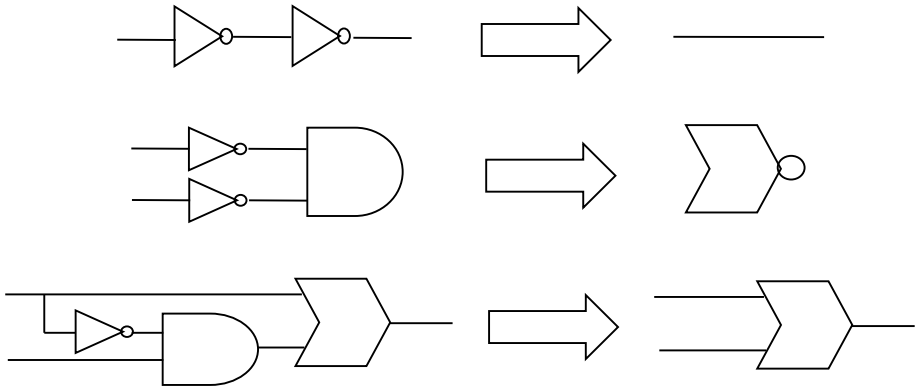
Rule based optimization



Target circuit



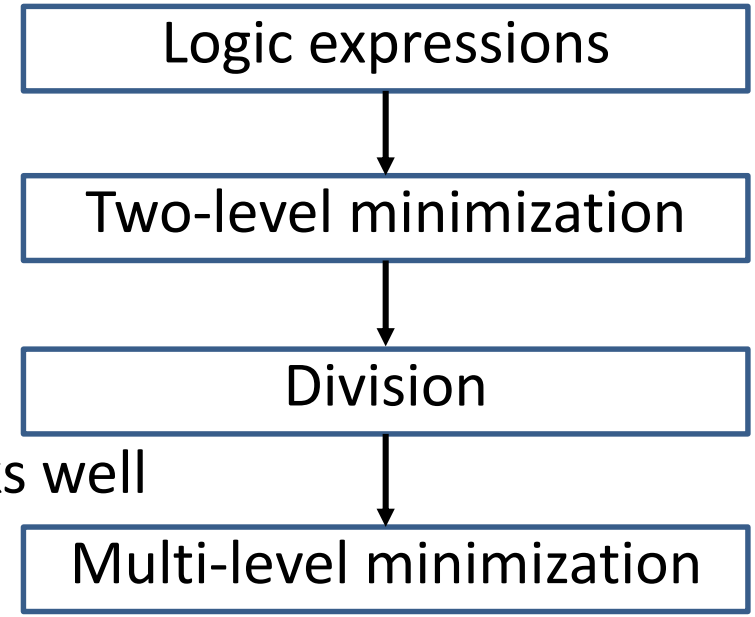
Example of optimization



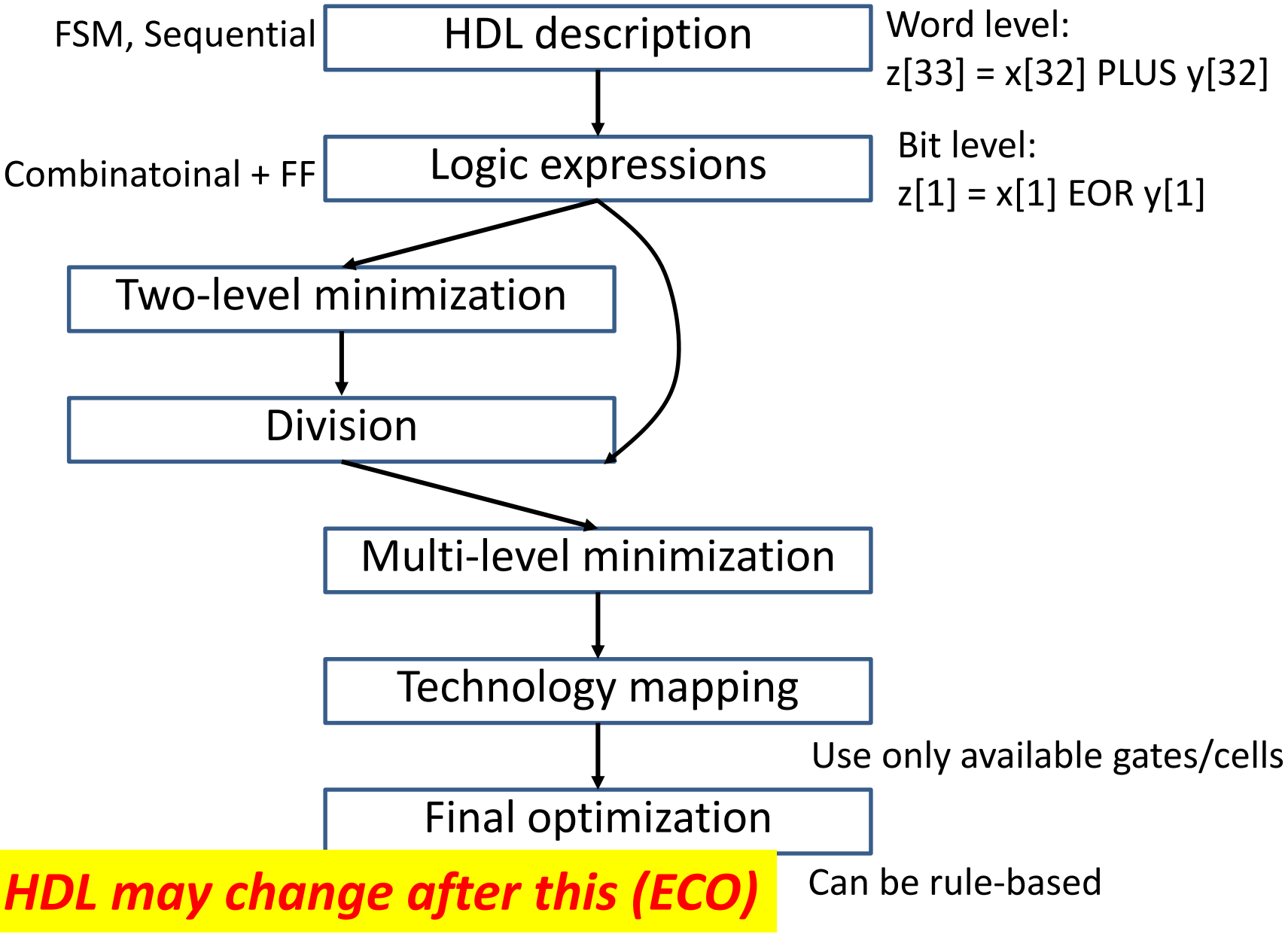
Rules

Synthesis of combinational multipliers

- Area minimum implementation
  - Array multipliers with ripple carry adders
  - For 8bit by 8bit multipliers, 430 gates implementation
  - Exists in design libraries
- Synthesis from truth table
  - 65,536 rows in truth table
  - Generated one has 40,000 gates!
  - No redundancy!
  - No multi-level minimization works well
  - Still a research topic!
  - Cannot find good “intermediate logic” automatically
  - Practically maybe OK (use the one in the library)

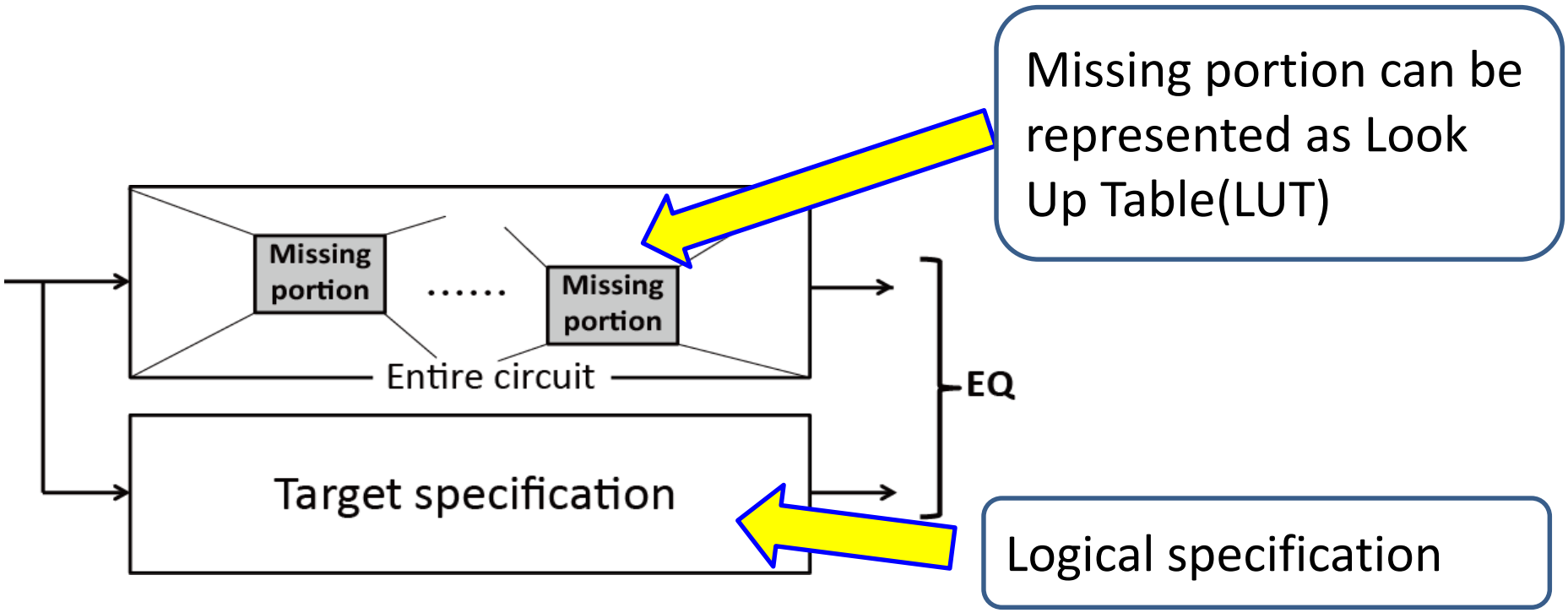


Real synthesis



Partial logic synthesis (my research)

- Find out appropriate circuits for the missing portions
  - Entire circuit must become logically equivalent to the specification which is given separately



Engineering Change Order: After implementation, specification changes

Logic debugging

LUT (Look up Table)

- Any logic function with m-inputs
  - MUX with m-control inputs
  - 2^m variables for truth table values
- $p_0, p_1, \ldots, p_{2^m-1}$ represent values of truth tables
- By changing those values, any logic function with m-input can be represented

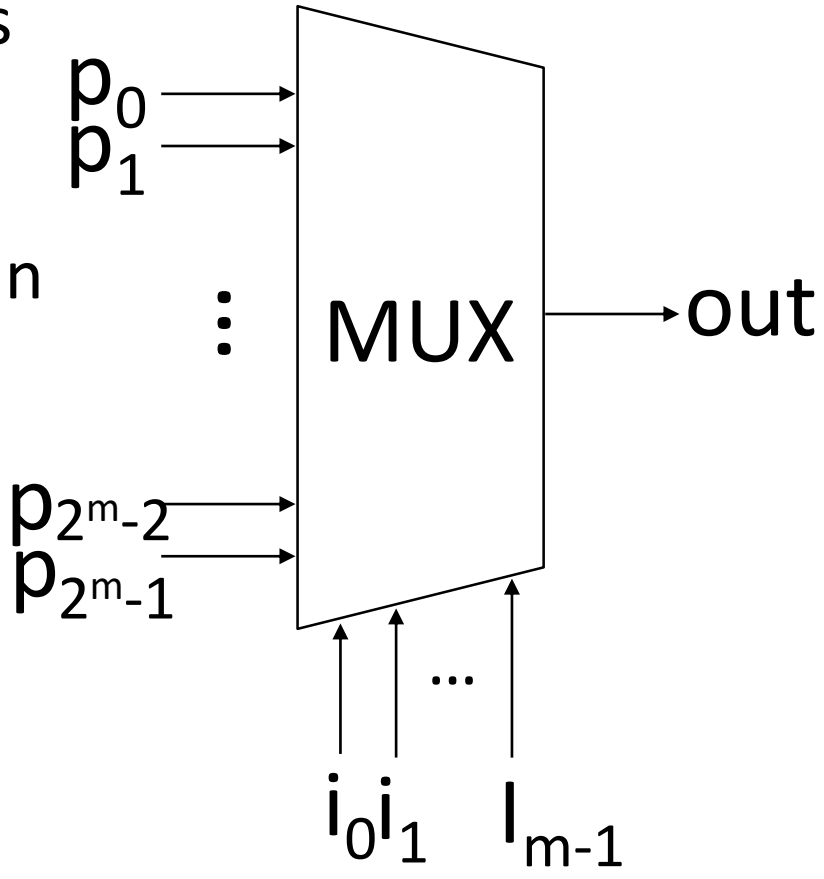
If $i_0\, i_{1\ldots} i_{2^m-1} = 00\ldots0$ then out = p_0 AND

If $i_0\, i_{1\ldots} i_{2^m-1} = 10\ldots0$ then out = p_1 AND

…

If $i_0\, i_{1\ldots} i_{2^m-1} = 11\ldots1$ then out = p_{2^m-1}

- Only one of $p_0, p_1, \ldots, p_{2^m-1}$ is connected to out



Problem formulation

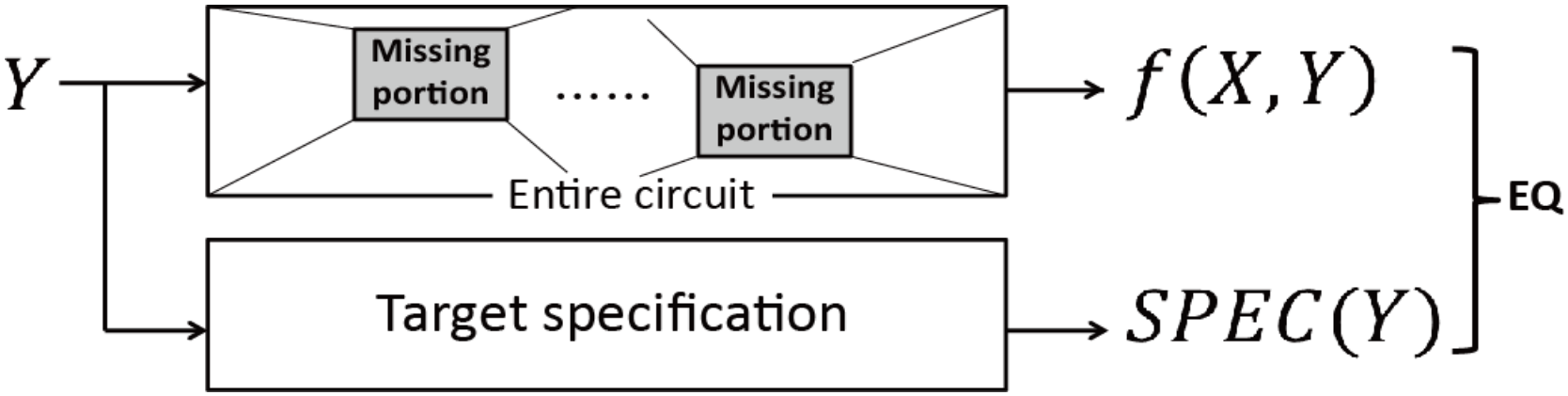
- Partial synthesis problems can be formulated as:

"Under appropriate programs for LUTs (existentially quantified), circuit behaves correctly for all possible input values (universally quantified)"

$$\exists X \forall Y . f(X,Y) = SPEC(Y)$$

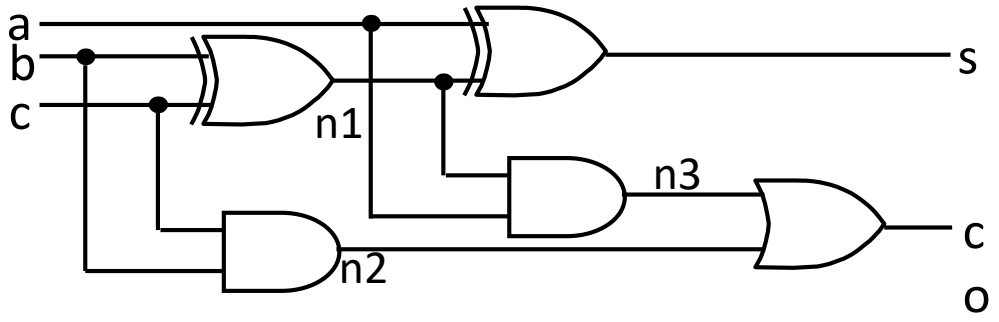
X : configurations of LUTs, Y : inputs value of the circuit

f : output value of target circuit, $SPEC$: output value of specification

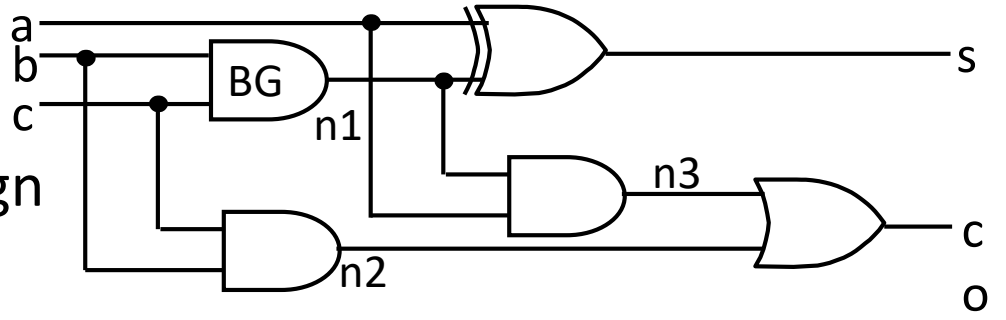


A buggy design for a 1-bit full adder

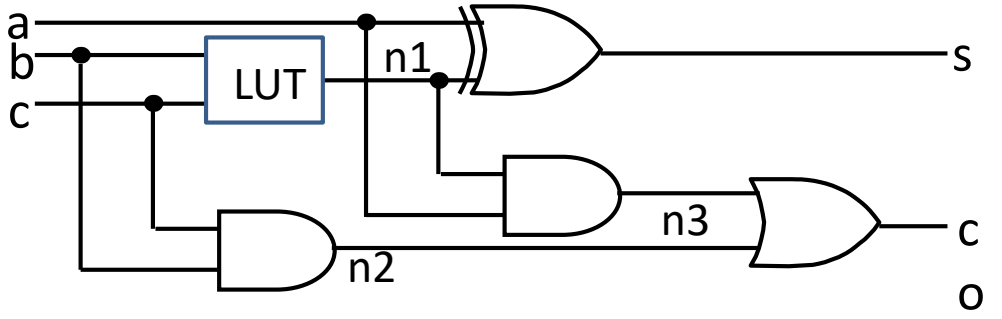
- Specification



- An example buggy design



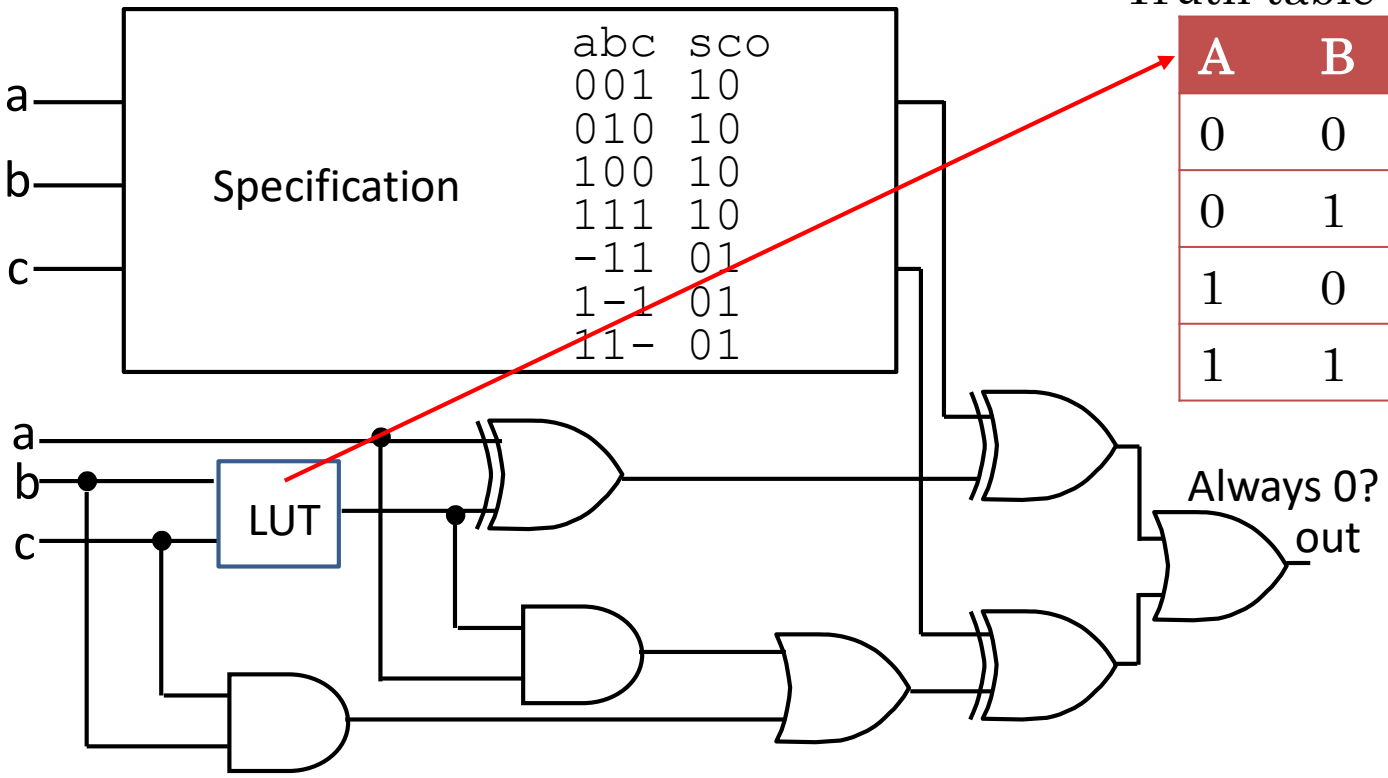
- Buggy design with LUT



Miter generation

- Specification in SOP
- Target in netlist with LUT
- If out is always 0 (UNSAT), the target is a correct one
- If SAT, there is a counter example generated by SAT solver

$$\exists X0, X1, X2, X3.\ \forall A, B, C.$$
$$\mathrm{Spec}(A, B, C) = \mathrm{Circuit}(X0, X1, X2, X3, A, B, C)$$



Truth table for LUT

A	B	D
0	0	X0
0	1	X1
1	0	X2
1	1	X3

Step 1

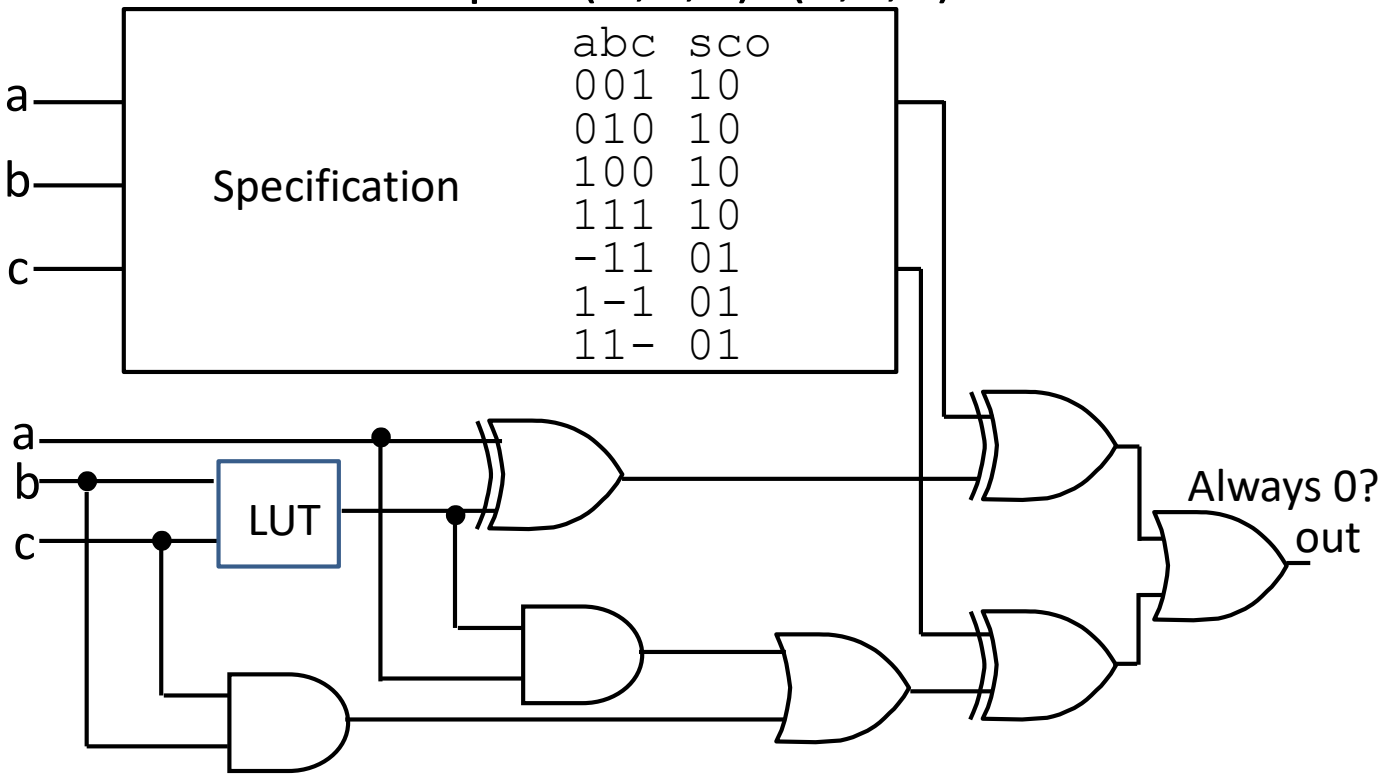
- In the beginning, we do not know how to program LUT
- Just need a counter example, and so solve the following SAT prob.

$$\exists X0, X1, X2, X3.\ \exists A, B, C.\ \mathrm{Spec}(A, B, C) = \mathrm{Circuit}(X0, X1, X2, X3, A, B, C)$$

Instead of

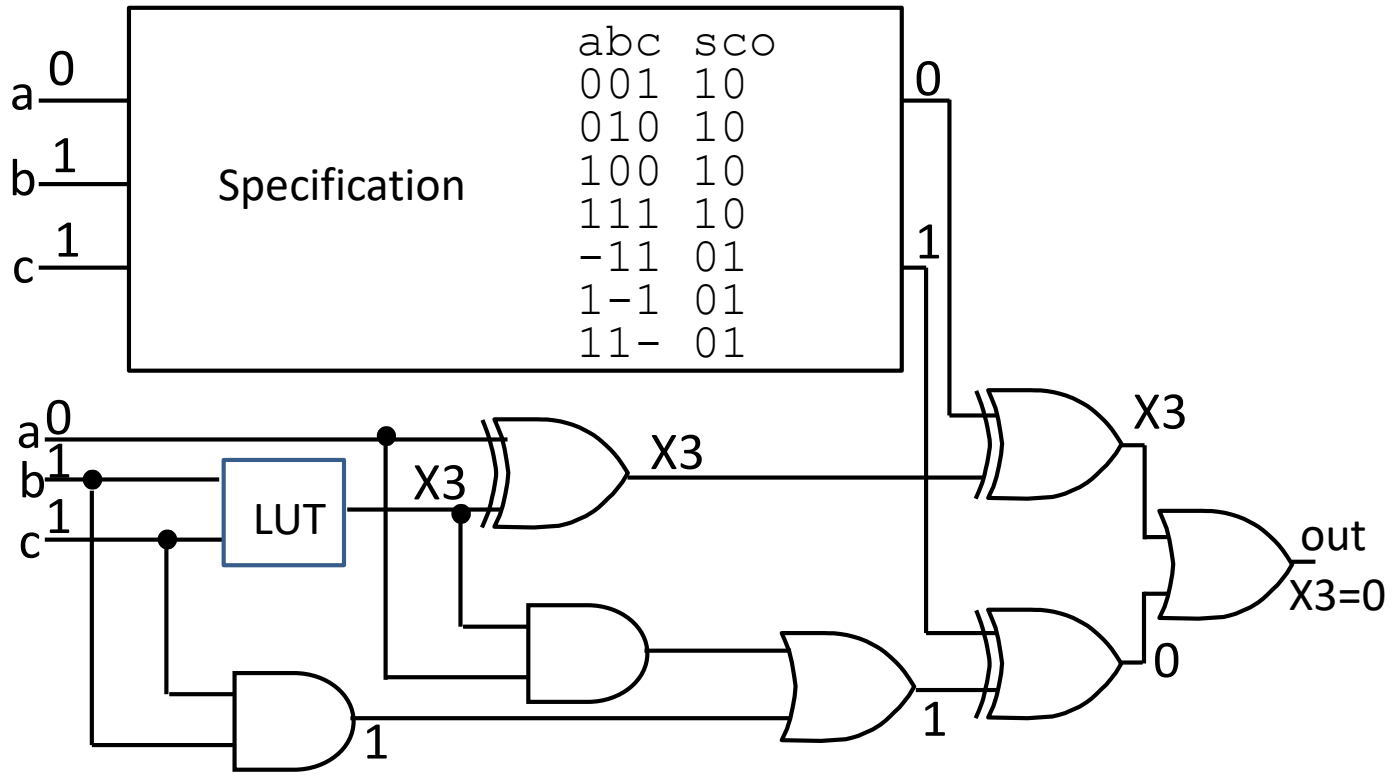
$$\exists X0, X1, X2, X3.\ \forall A, B, C.\ \mathrm{Spec}(A, B, C) = \mathrm{Circuit}(X0, X1, X2, X3, A, B, C)$$

- Then get a counter example: (A,B,C)=(0,1,1)



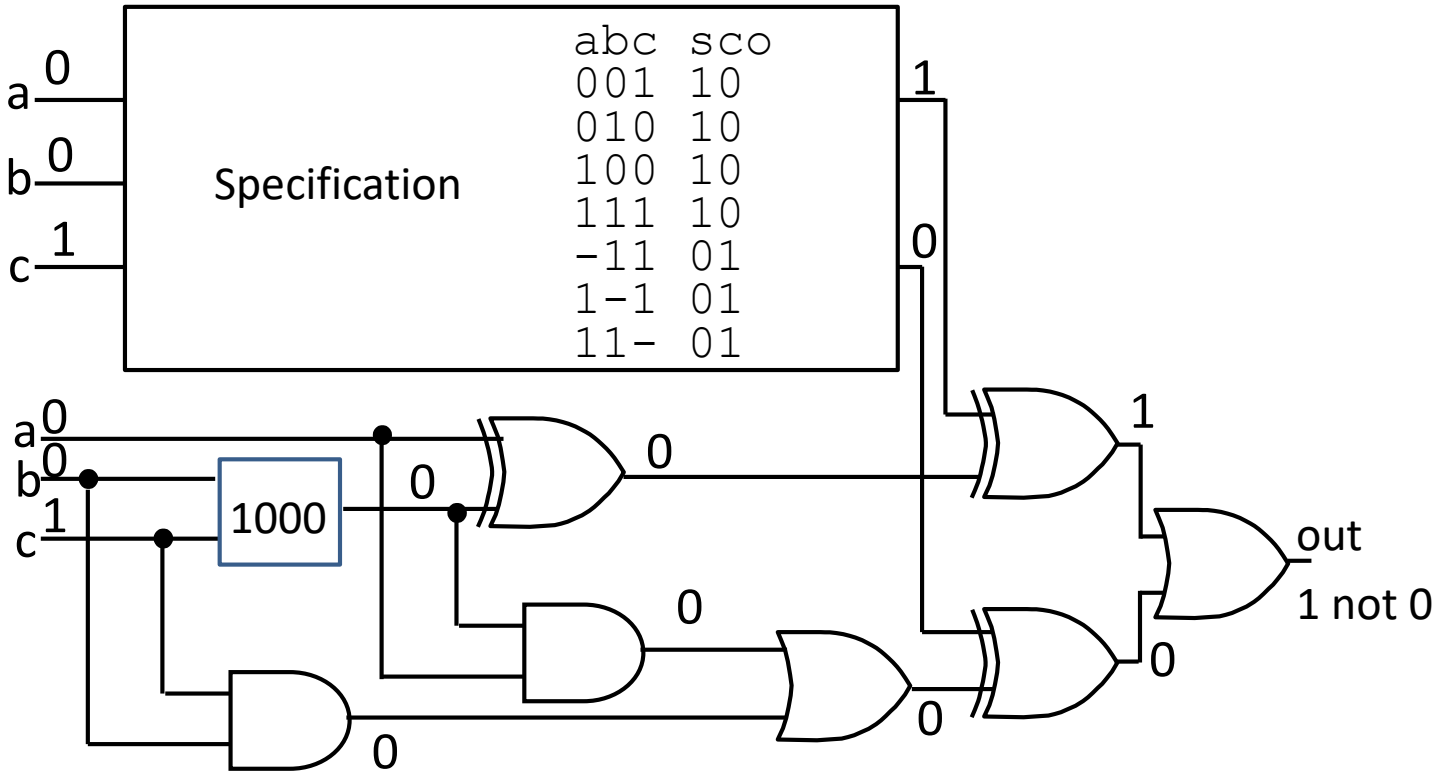
Step 2

- Get the function for LUT (X1,X2,X3,X4) under which out is 0 when (A,B,C)=(0,1,1)
  - X3 must be 0
- SAT solver returns a solution example
  - (X0,X1,X2,X3)=(1,0,0,0)



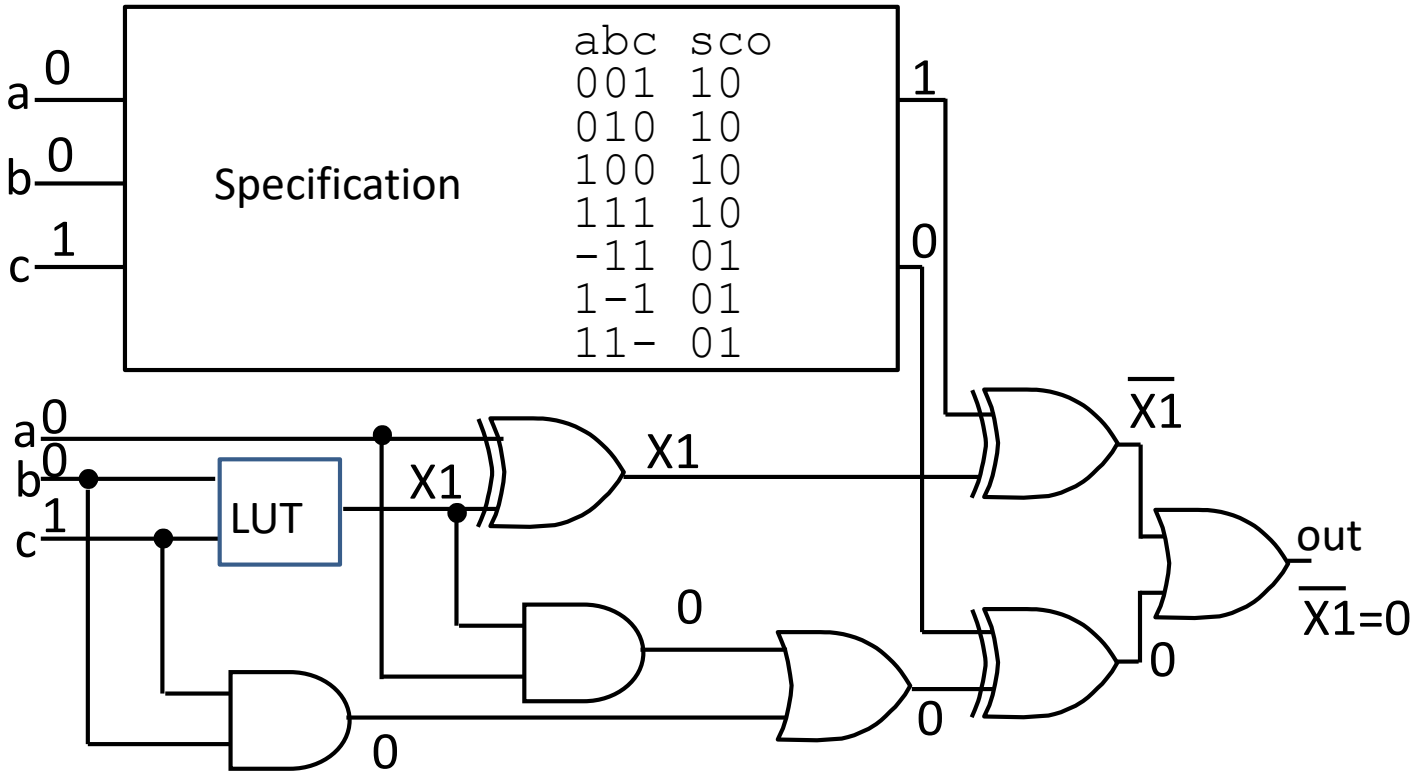
Step 3

- Program the LUT with (X1,X2,X3,X4)=(1,0,0,0)
- Create a miter and check the equivalence
  - If UNSAT, current (X1,X2,X3,X4) is a correct function for LUT
- Unfortunately SAT, and returns a counter example
  - (A,B,C)=(0,0,1)



Step 4

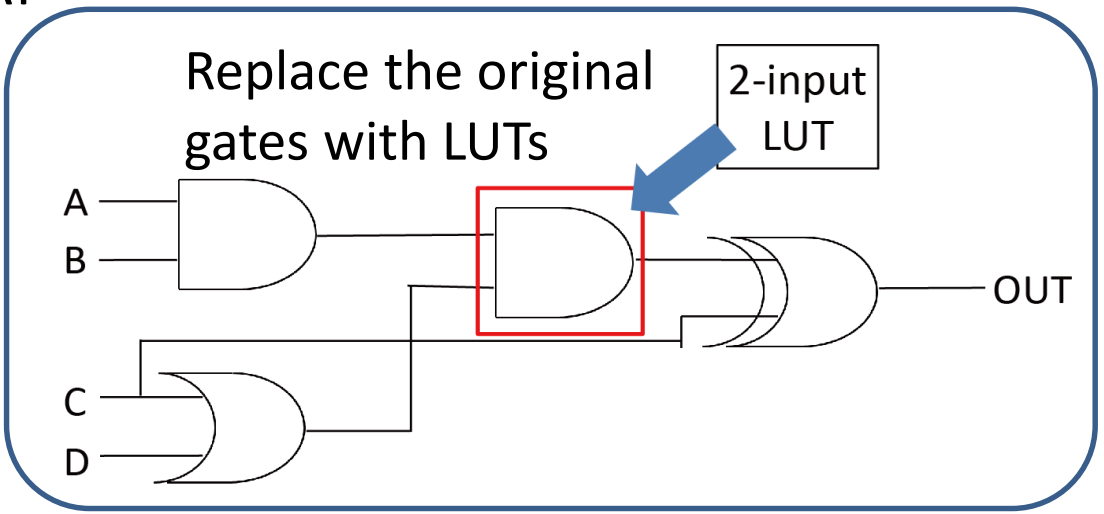
- When the inputs (A,B,C)=(0,1,1) and (A,B,C)=(0,0,1), out must be 0
  - X1 must be 1 and X3 must be 0
- If SAT returns a solution: (X0,X1,X2,X3)=(0,1,1,0), finish
  - If SAT returns other solutions, just continue the steps



How large circuits can be processed?

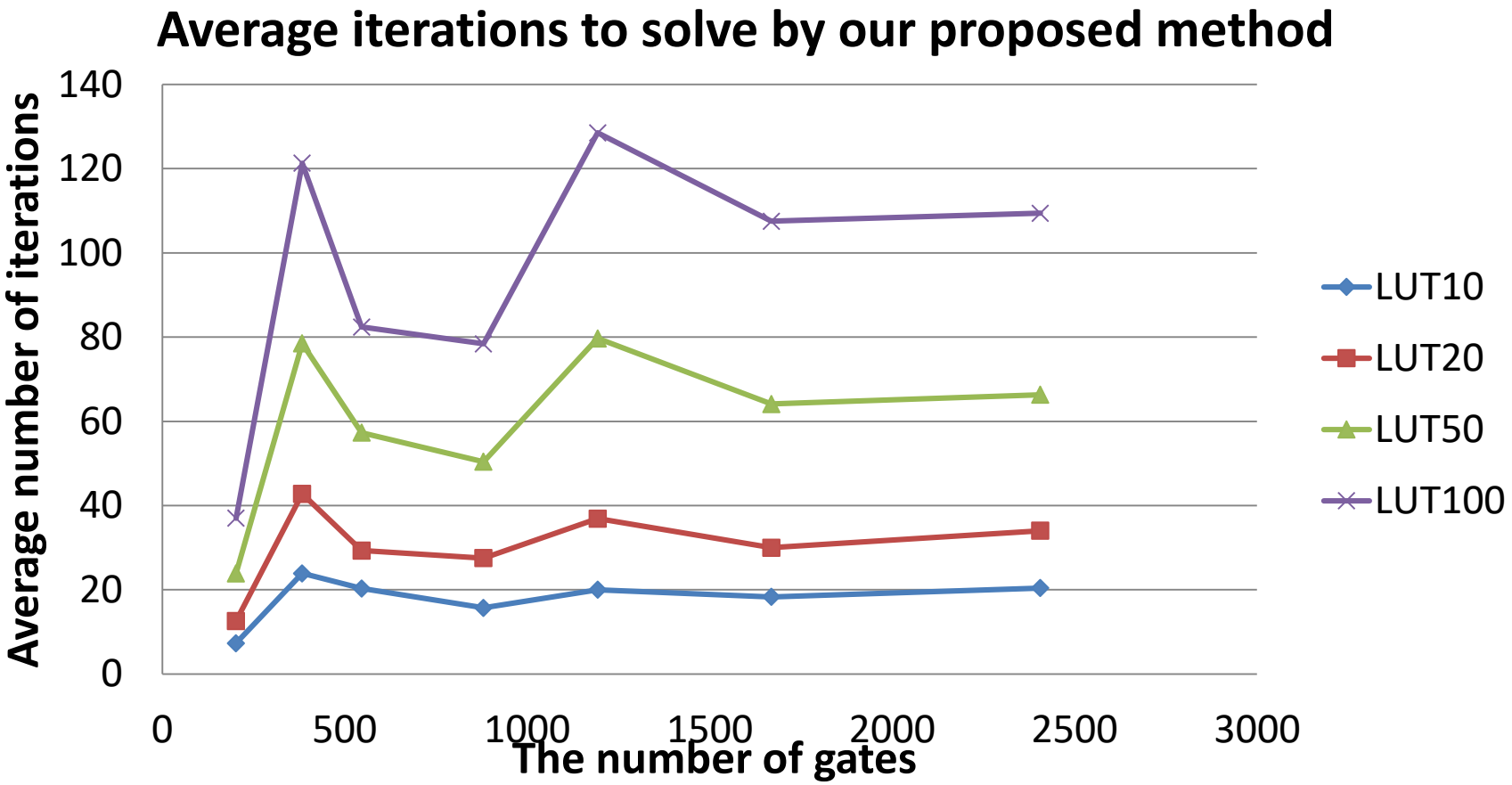
Experiment

- Replaced 10, 20, 50 and 100 original 2-input gates picked up randomly with
- Used the original circuits as specification
- Target circuit
  - ISCAS 85/89 benchmark
  - SAT solver : Pico SAT



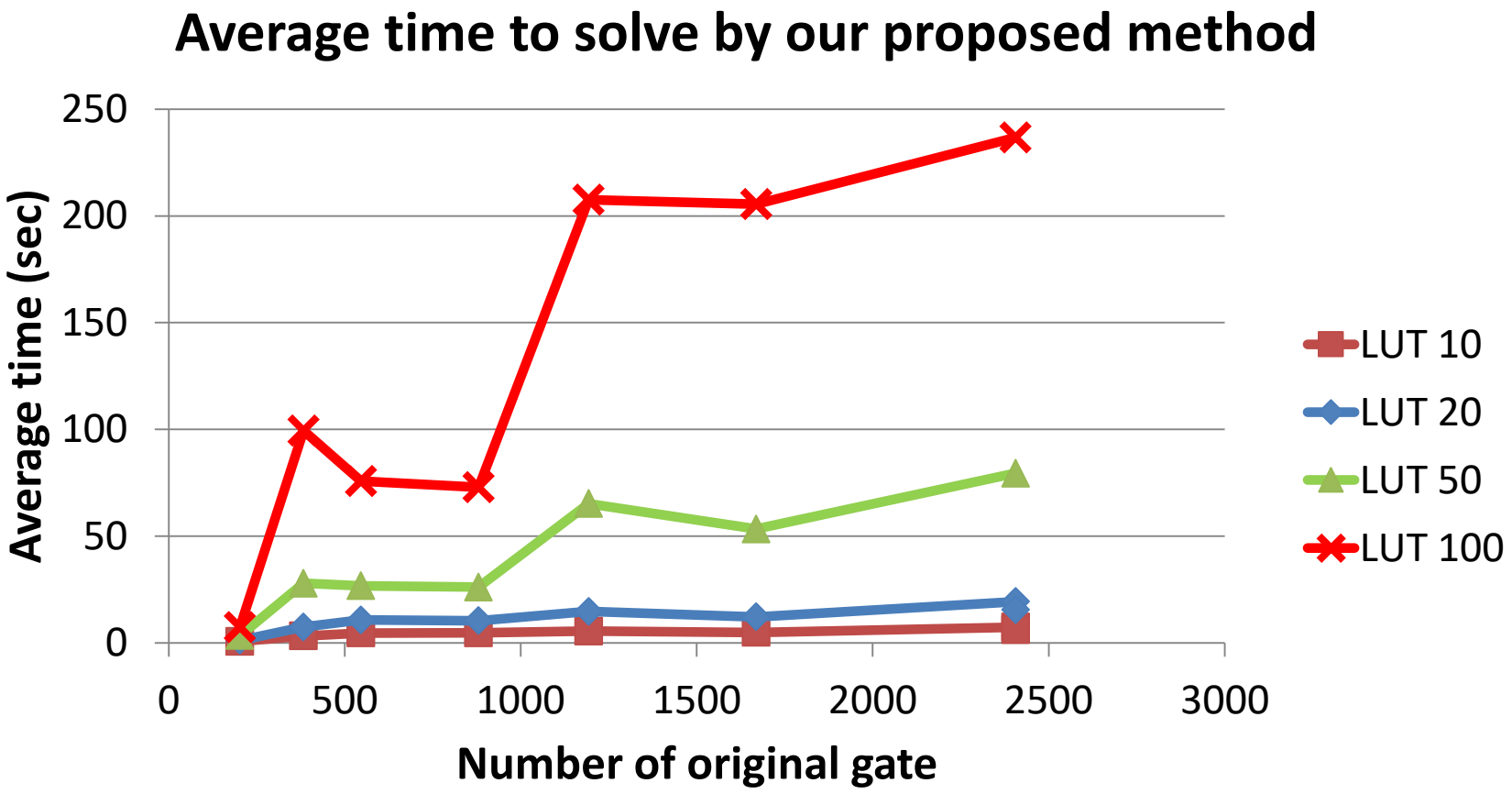
Experimental results (1)

- Number of iterations is surprisingly small
- Number of iterations increases more rapidly with the increase of number of LUTs than size of circuits

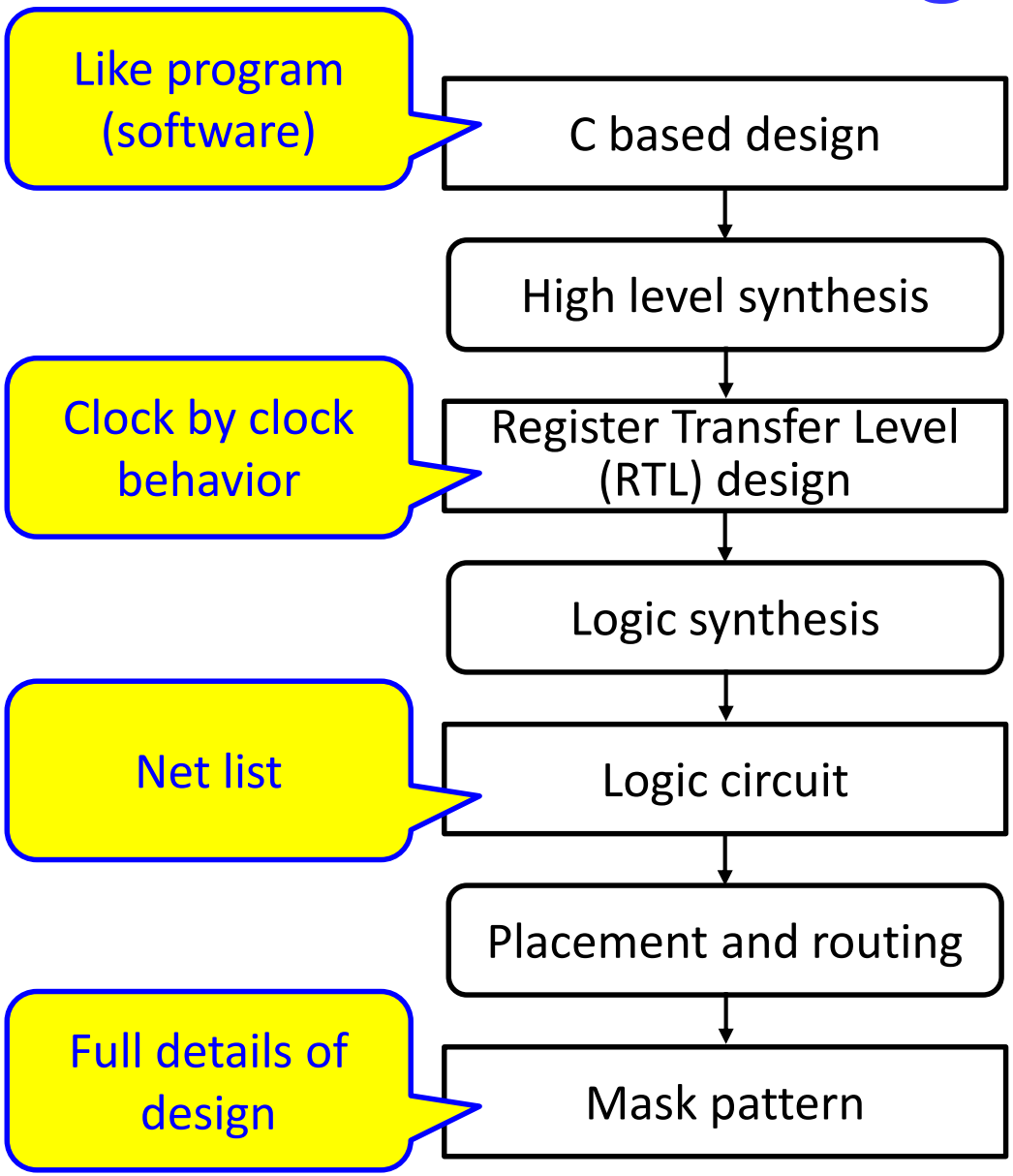


Experimental results (2)

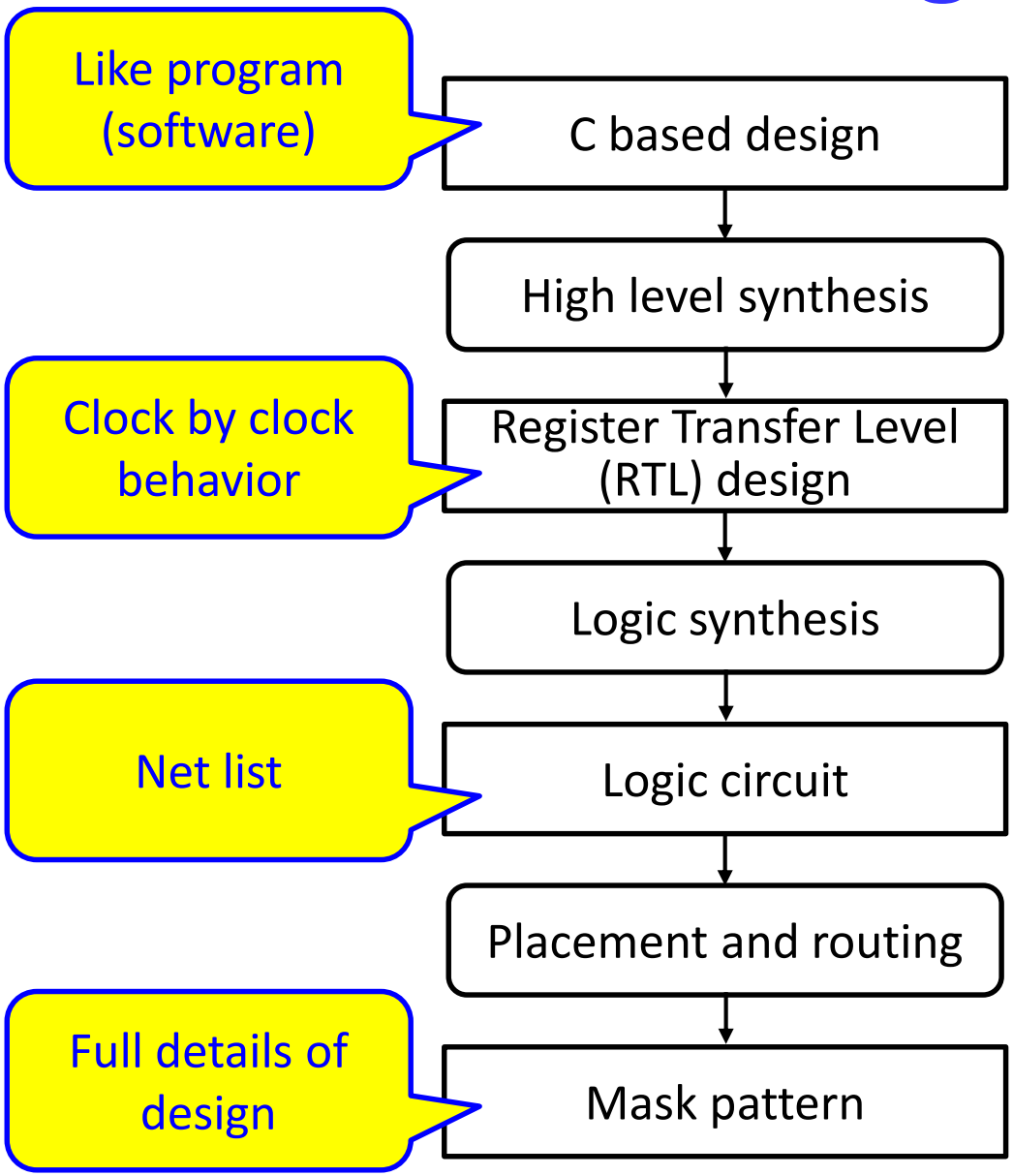
- For circuits with 2,000 gates and 100 LUTs it took several minutes to finish



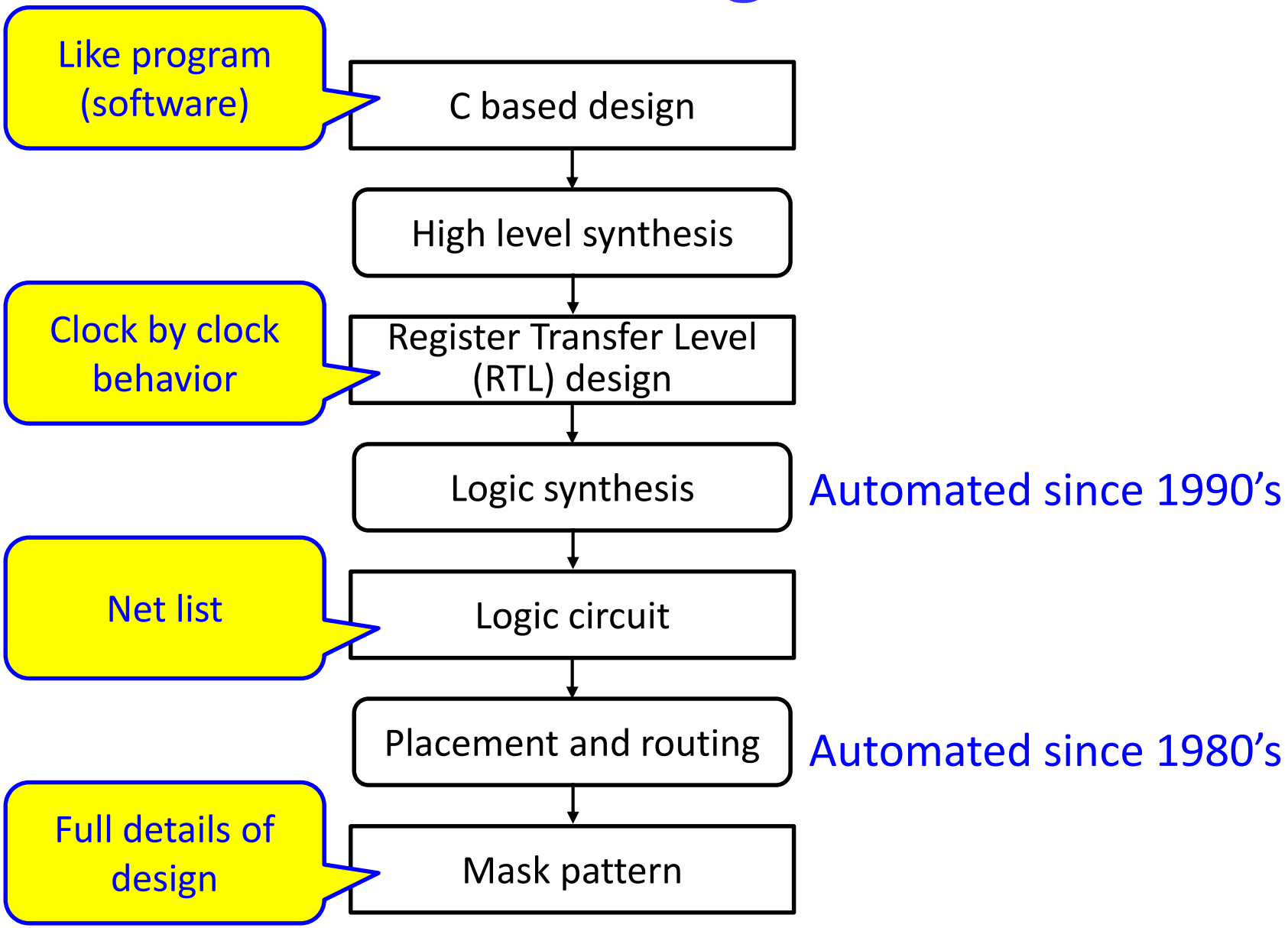
Hardware design flow



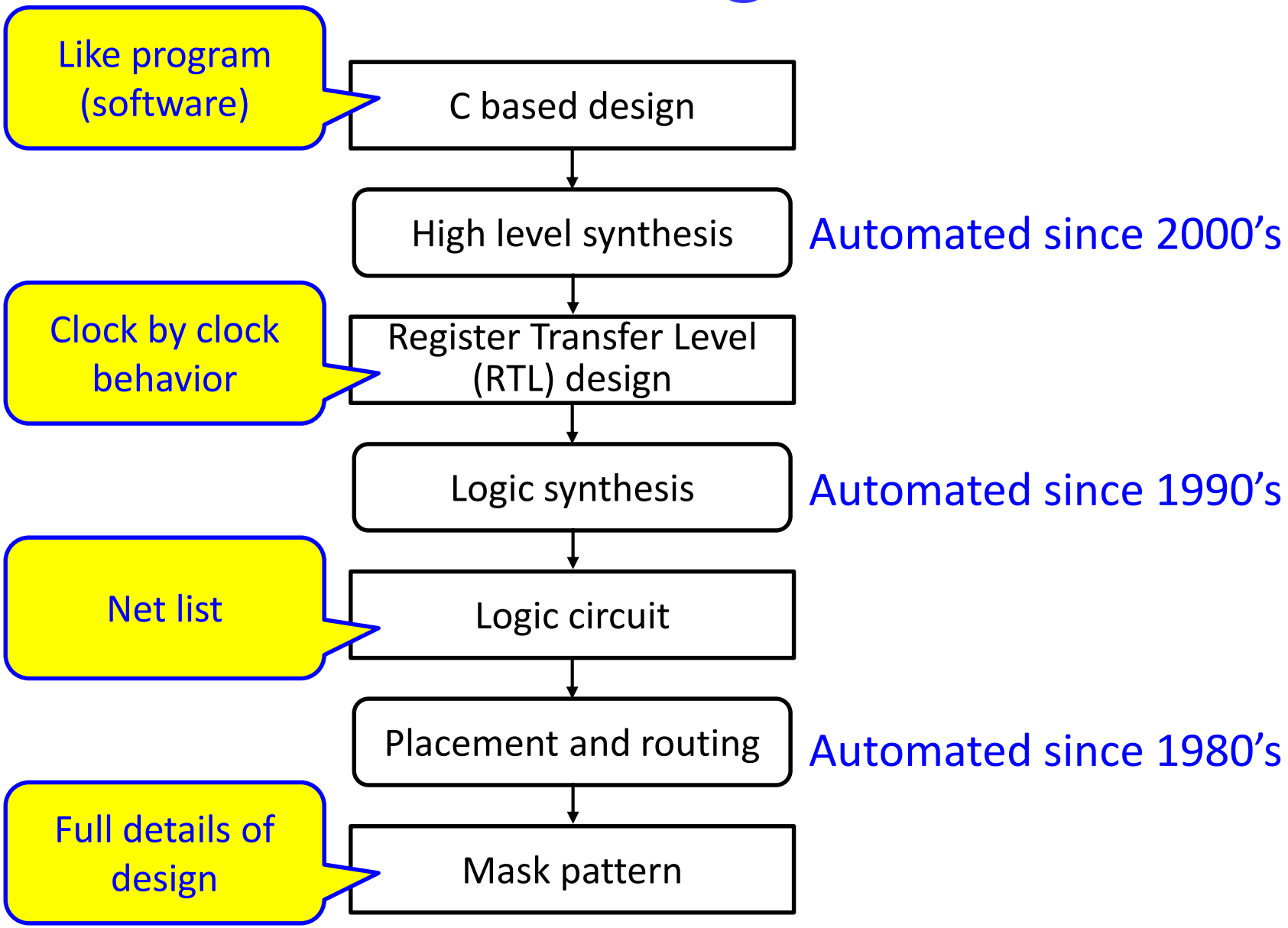
Hardware design flow



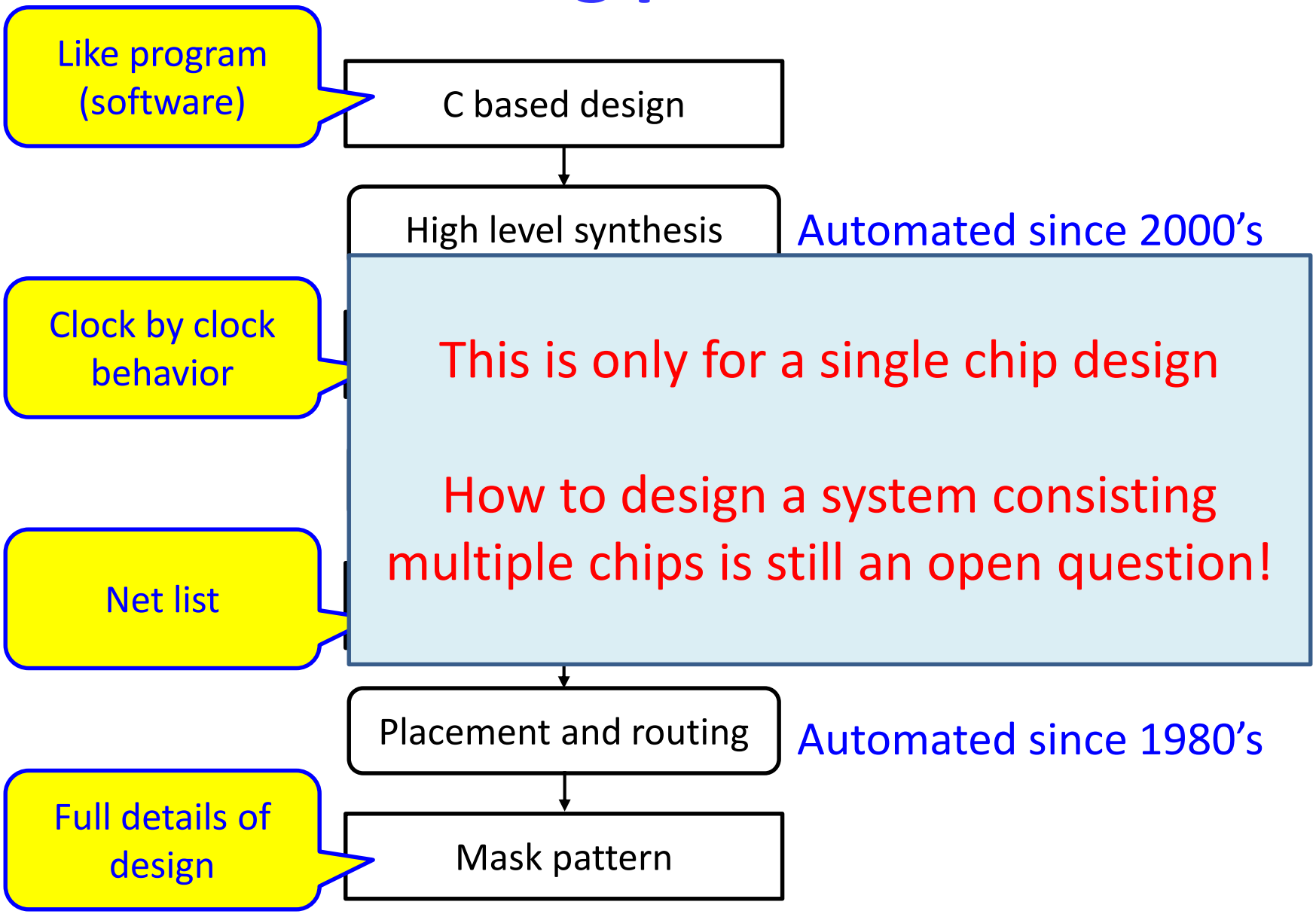
Hardware design flow



Hardware design flow

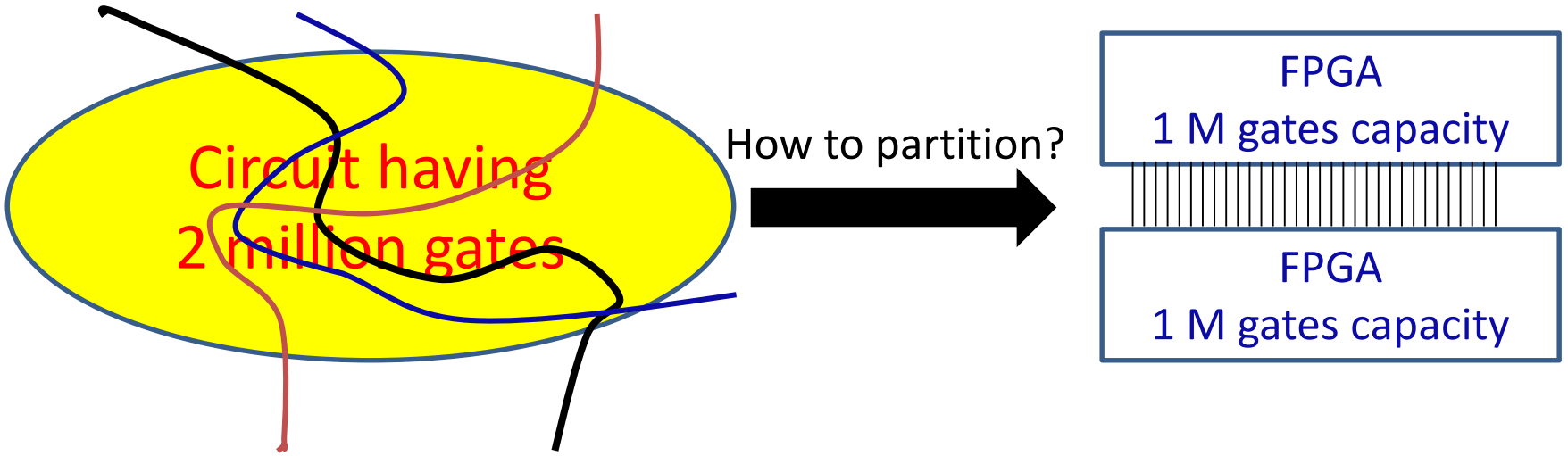


Remaining problem



Automatic partitioning in gate level practically no way to work!

- Ex: Given 2 million gate circuit should be partitioned into two FPGA chips
  - Each FPGA has 1 million gate capacity
  - Partitioning itself is straightforward
- But, how many signals will cross the chip boundary?



- Do need partitioning in algorithm level, not gate level!

Multi-chip synthesis example: Weight sum = Matrix-vector product

$$I^i_{stim} = c\sum_{j=1}^{N} W_{ij} I^j_s$$

$\leftarrow O(N^2)$

Large number of calculation and data communication

$$\begin{pmatrix} I^1_{stim} \\ I^2_{stim} \\ I^3_{stim} \\ I^4_{stim} \\ \vdots \\ I^N_{stim} \end{pmatrix} = \begin{pmatrix} w11 & w12 & w13 & w14 & & w1N \\ w21 & w22 & w23 & w24 & & w2N \\ w31 & w32 & w33 & w34 & & w3N \\ w41 & w42 & w43 & w44 & & w4N \\ & & & & \ddots & \vdots \\ wN1 & wN2 & wN3 & wN4 & \dots & wNN \end{pmatrix} \cdot \begin{pmatrix} I^1_s \\ I^2_s \\ I^3_s \\ I^4_s \\ \vdots \\ I^N_s \end{pmatrix}$$

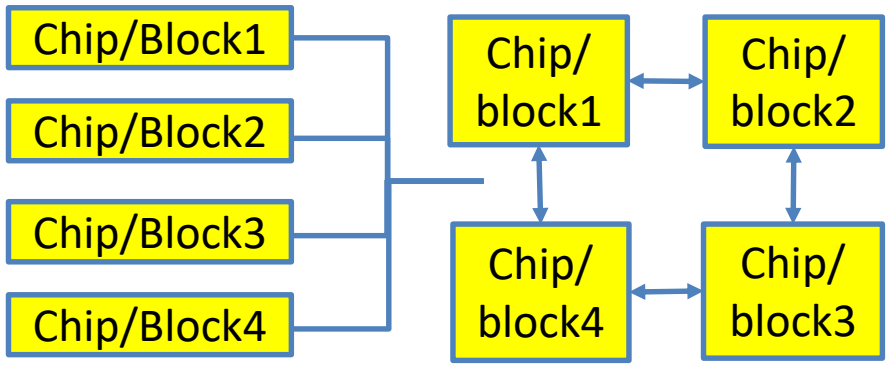
↑Assume dense matrix

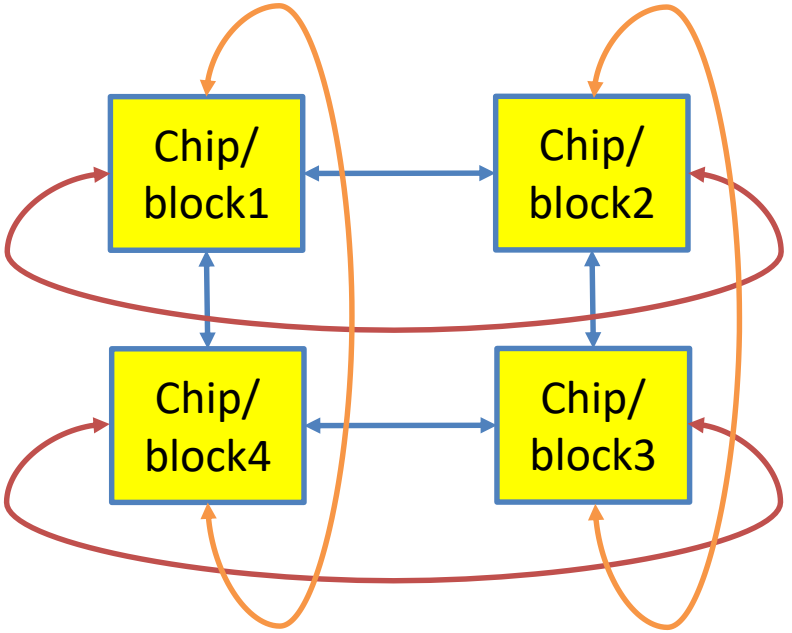
(in general can be very sparse, not discussed here)

- Need good algorithm/template for efficient computation
  - Especially for multiple chips/blocks architecture

Extension to multiple chips

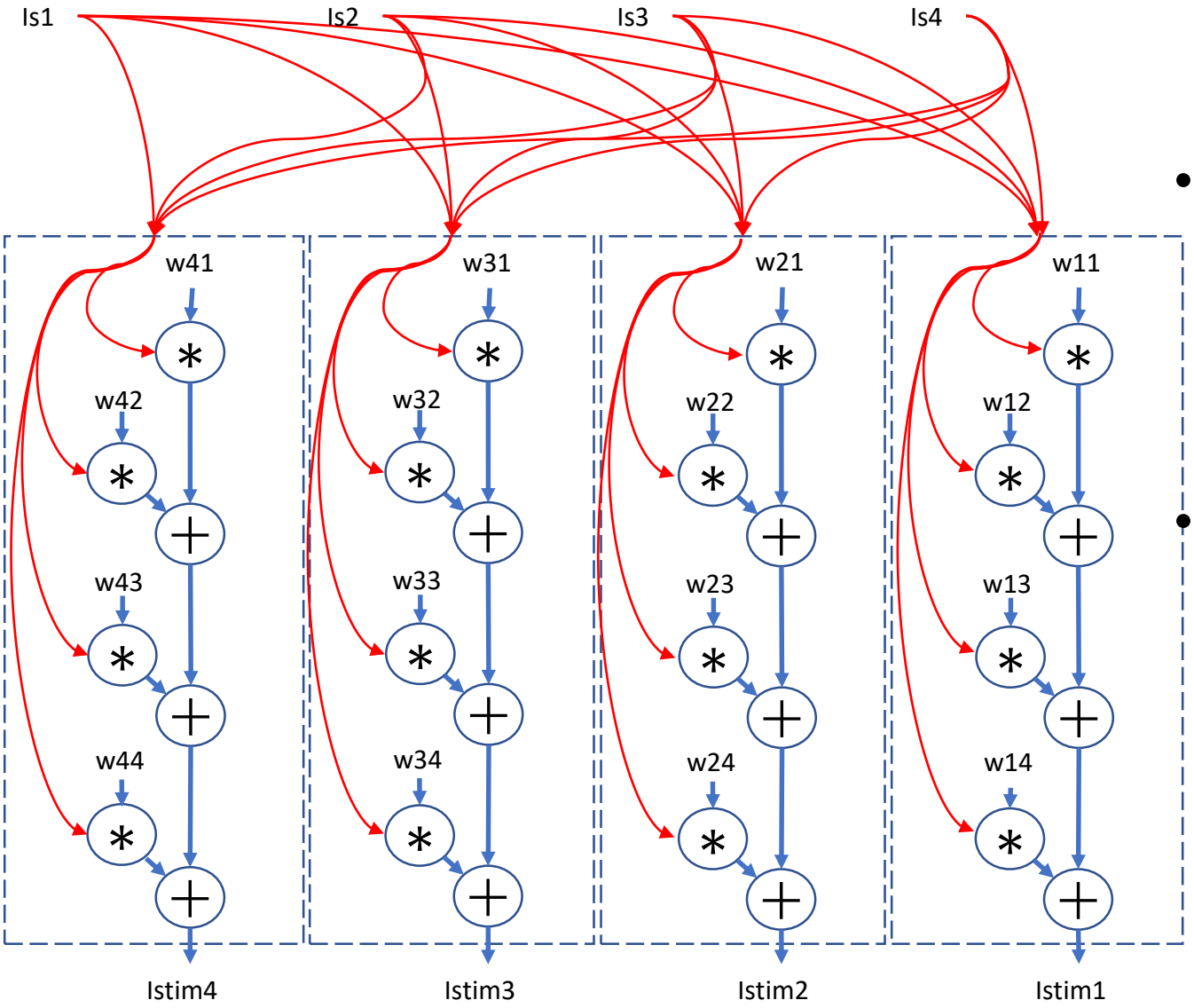
- Weighted sum is memory and computation resource consuming
- Communication latency can easily become bottleneck
- Easily implementable network topology for multiple chips
  - Common Bus
    - One pair of communication
  - Ring
    - Only with neighbors but all pairs at the same time
  - Mesh (2D, 3D, 4D, 5D, 6Dtorus)





Will show that ***ring is sufficient*** for maximum speed up

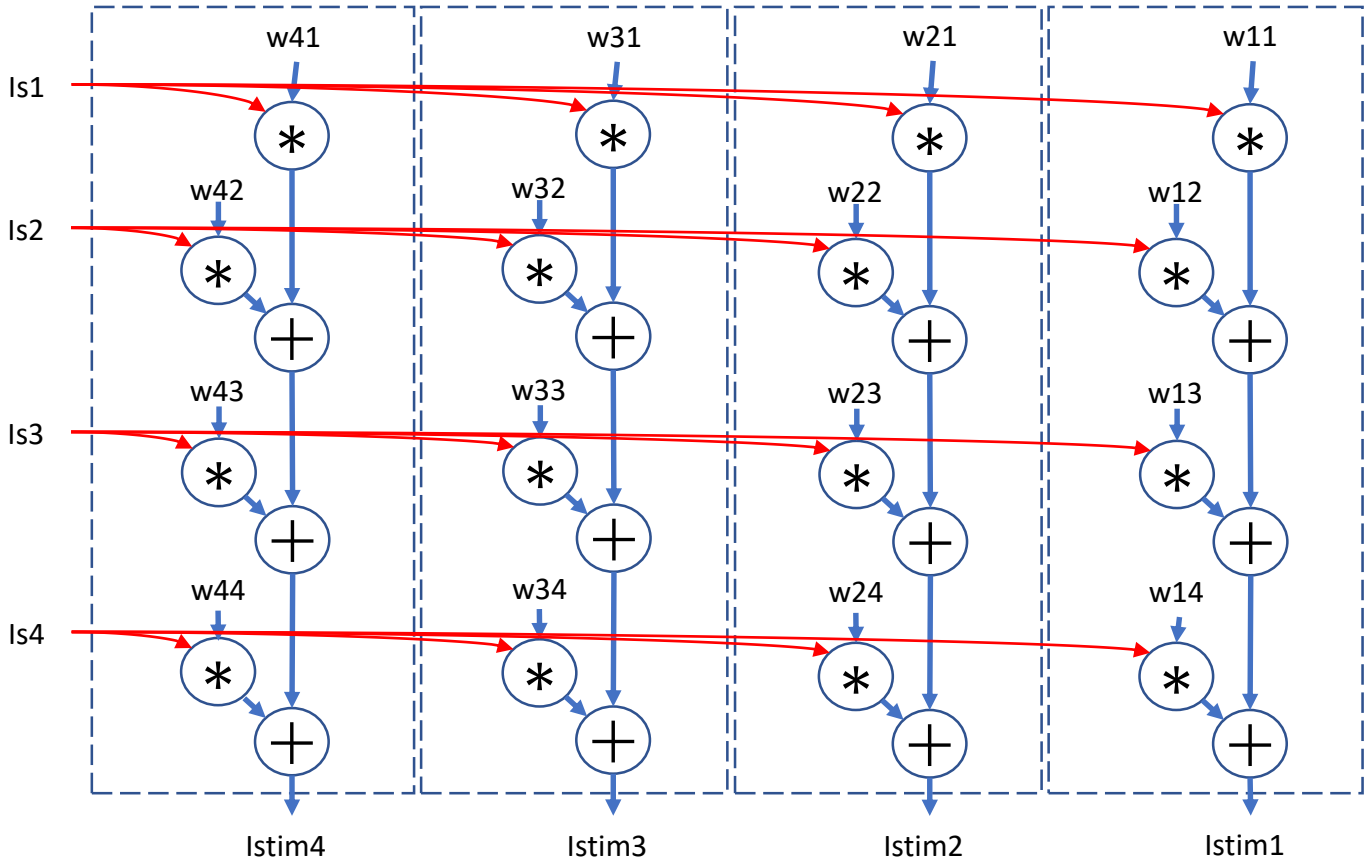
Method 1



- Send all vector elements to every node initially
- The communication may become overhead of calculation
- Need lots of storage

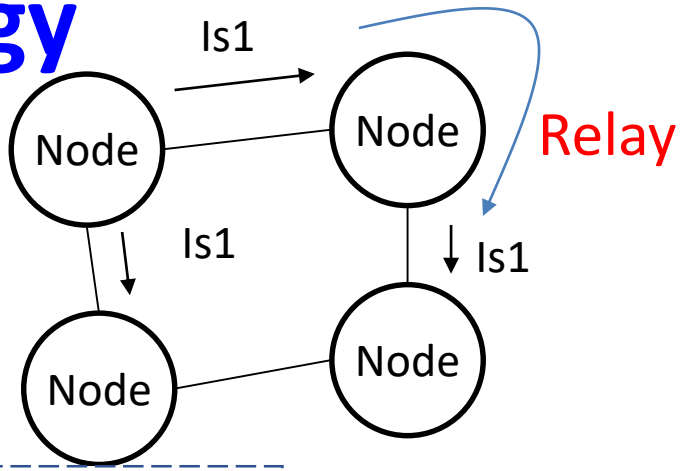
Method 2

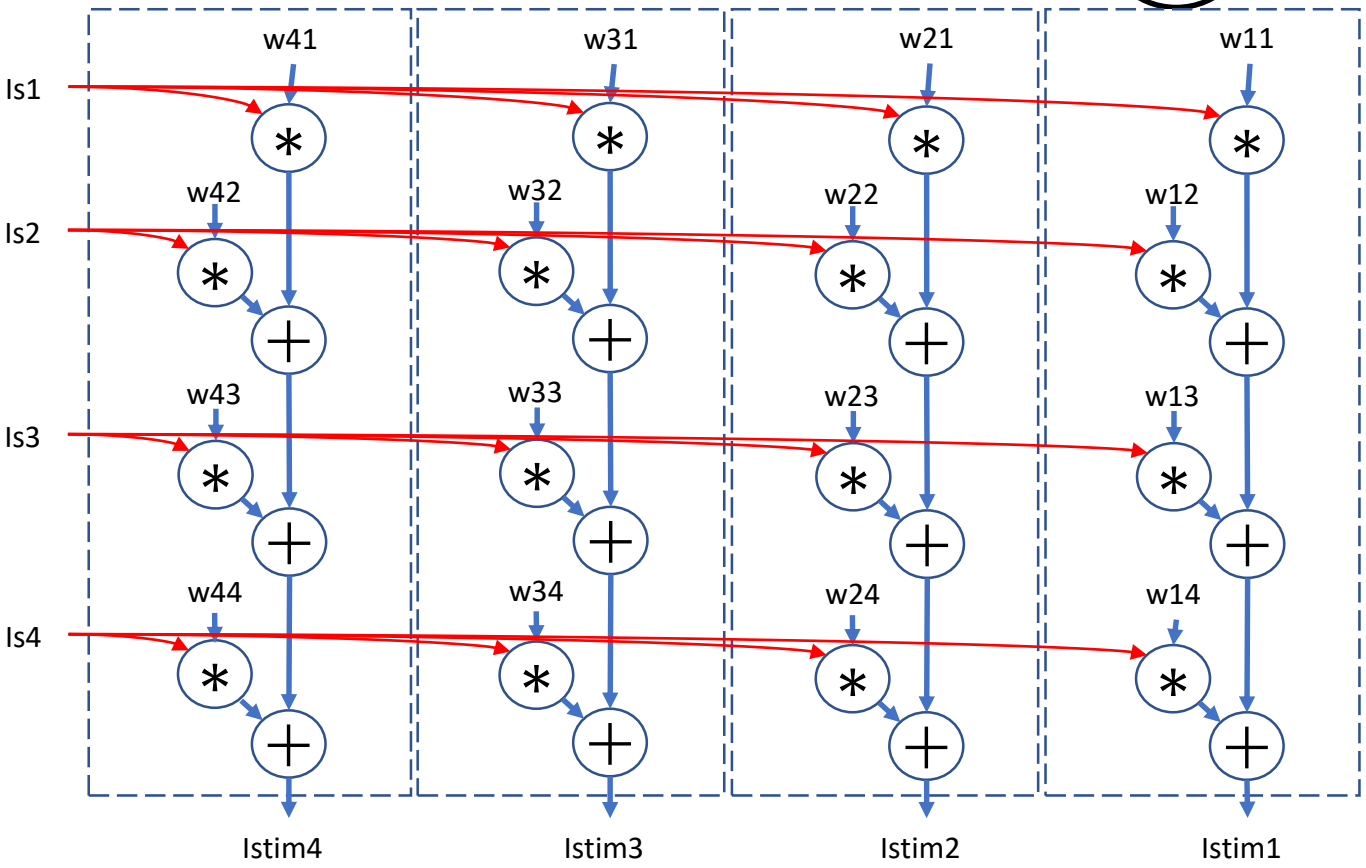
- Broadcast one of the vector elements to in every cycle
- More efficient than method 1, if multiplication and communication can be executed simultaneously
- If the topology is NOT bus, communications may need relaying



Method 2 in ring topology

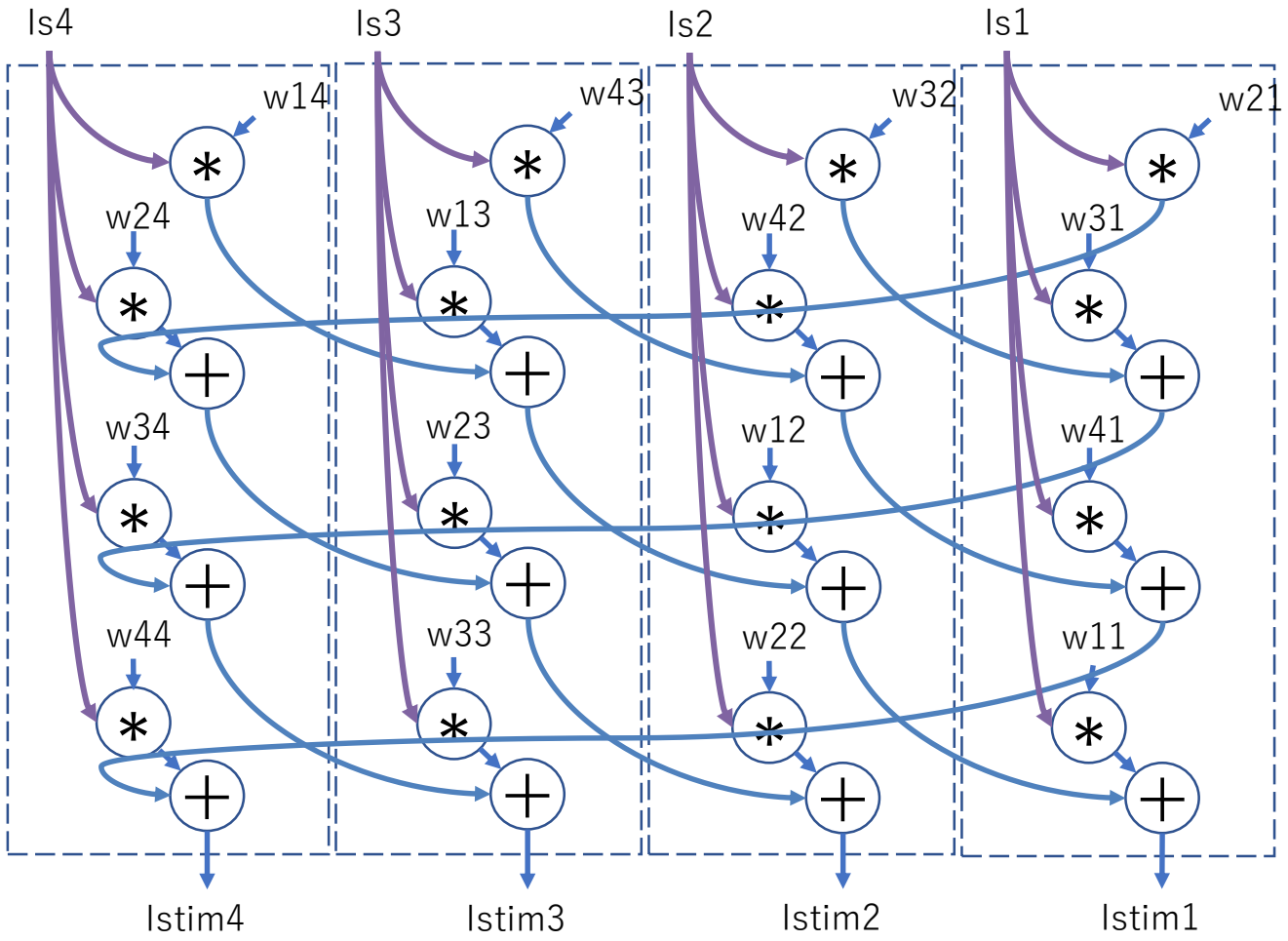
- Ring connection is easy to scale up
- Communication is not between adjacent nodes and needs relaying





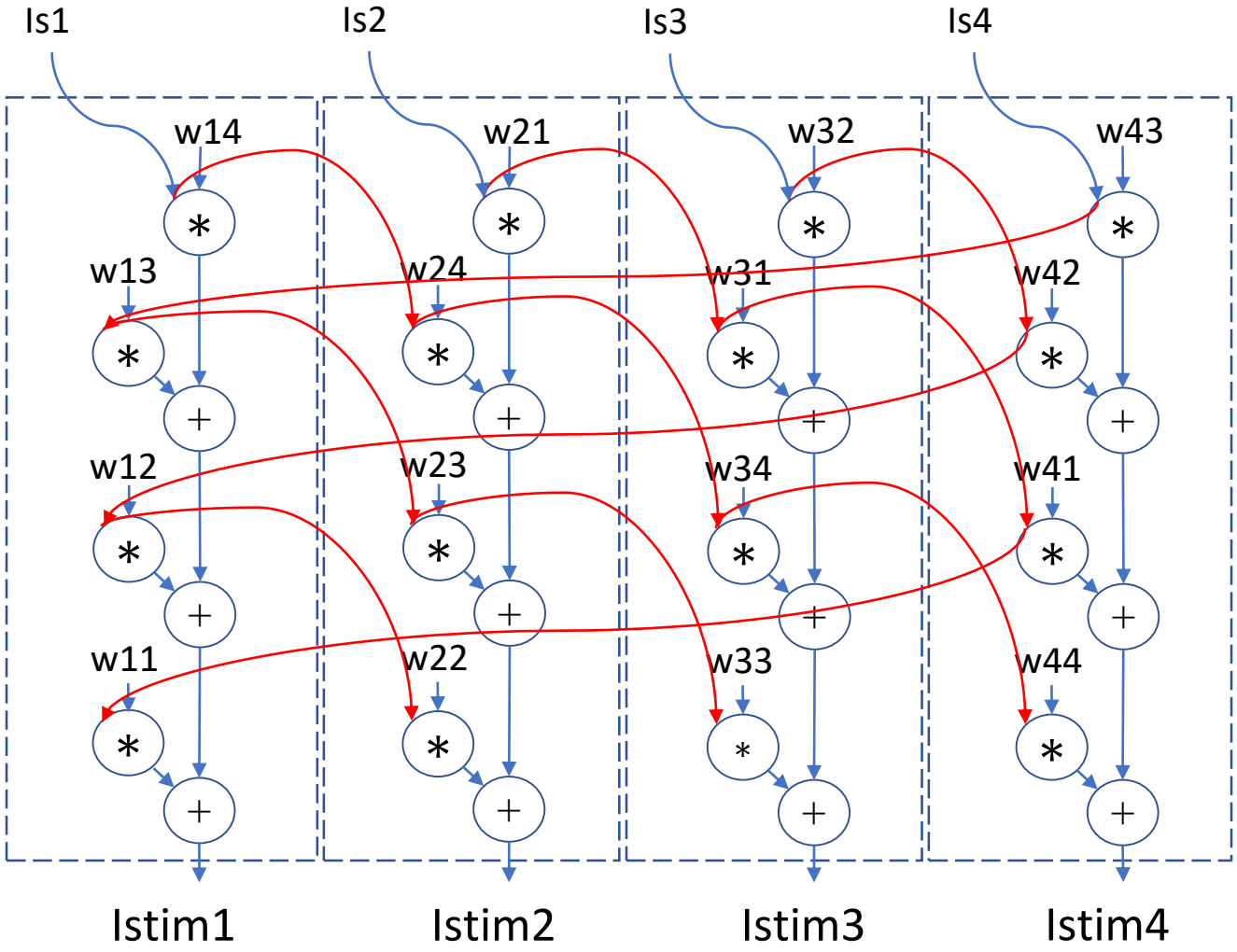
Method 3

- Communicate partial products among nodes by cycle
- No communication overhead!



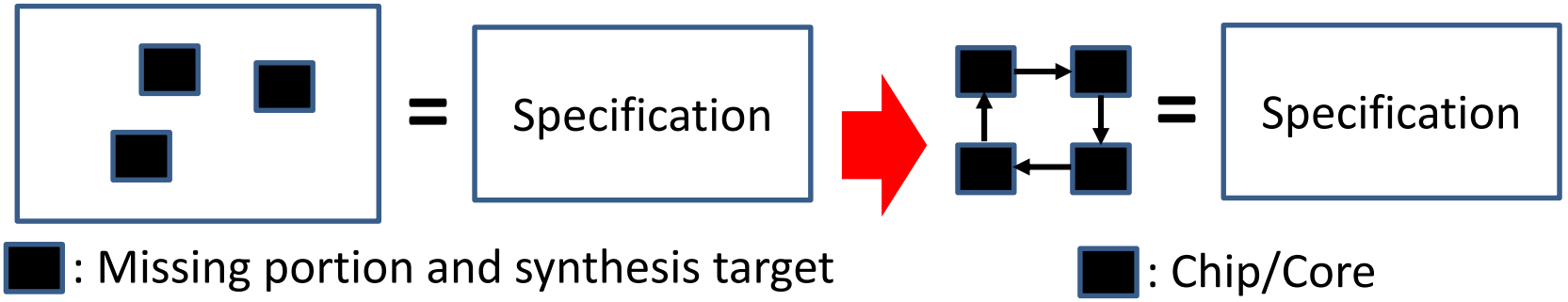
Method 4

- Communicate vector elements among nodes by cycle
- No communication overhead!



Automatic synthesis of parallel/distributed computing with partial logic synthesis

- Automatic synthesis of parallel/distributed computing can be formulated as partial logic synthesis problem
  - Solved by SAT solvers with implicit and exhaustive search
  - Work only for small instances of the problems
- Use human induction to generalized the solutions
  - Generalized solution can be formally verified



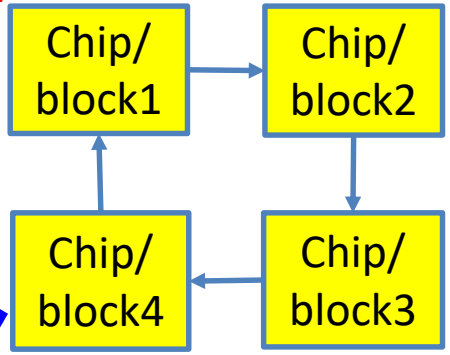
Can we automatically transform the computations?

Template based synthesis

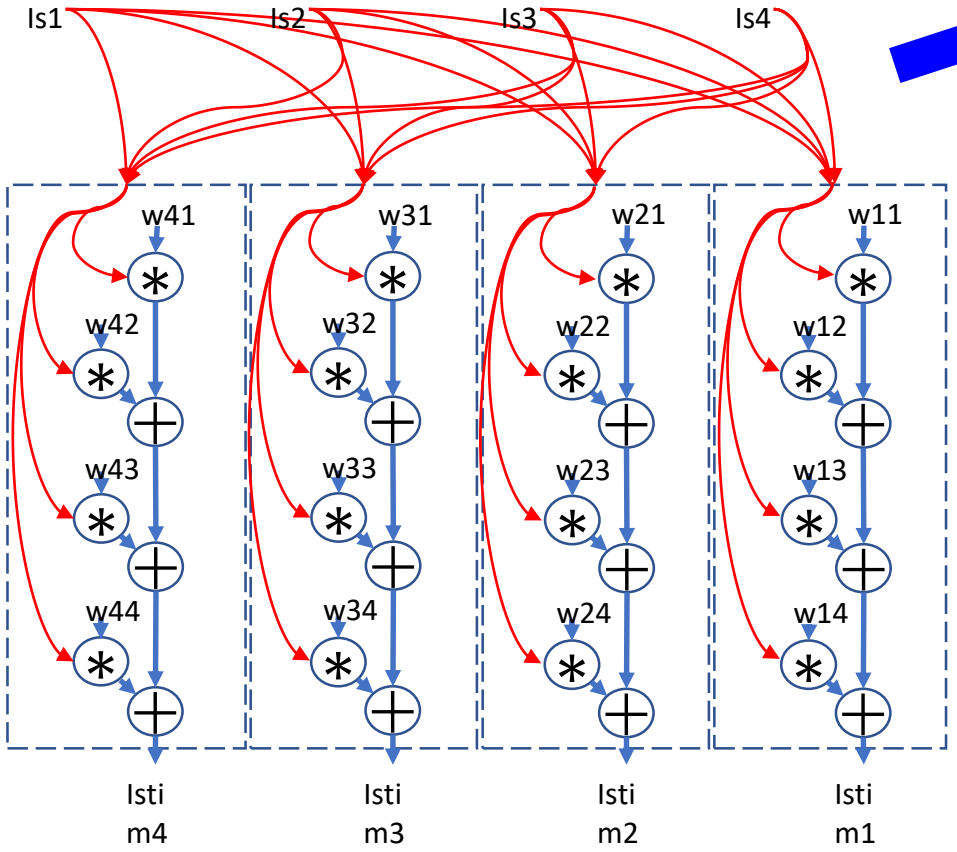
4X4 weighted sum

$$\begin{pmatrix} I_{stim}1 \\ I_{stim}2 \\ I_{stim}3 \\ I_{stim}4 \end{pmatrix} = \begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix} \cdot \begin{pmatrix} I_s1 \\ I_s2 \\ I_s3 \\ I_s4 \end{pmatrix}$$

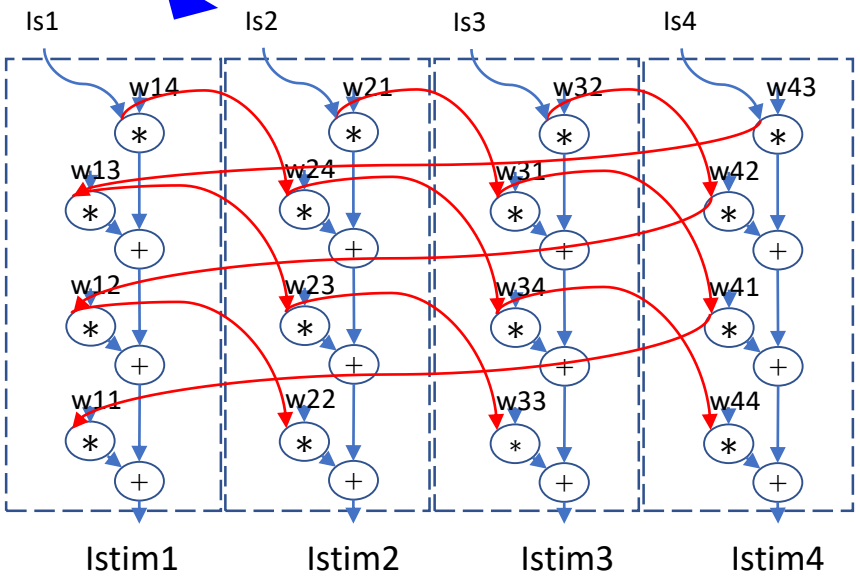
Template for one input stream and one output stream from library



Automatic identification of portions to be transformed

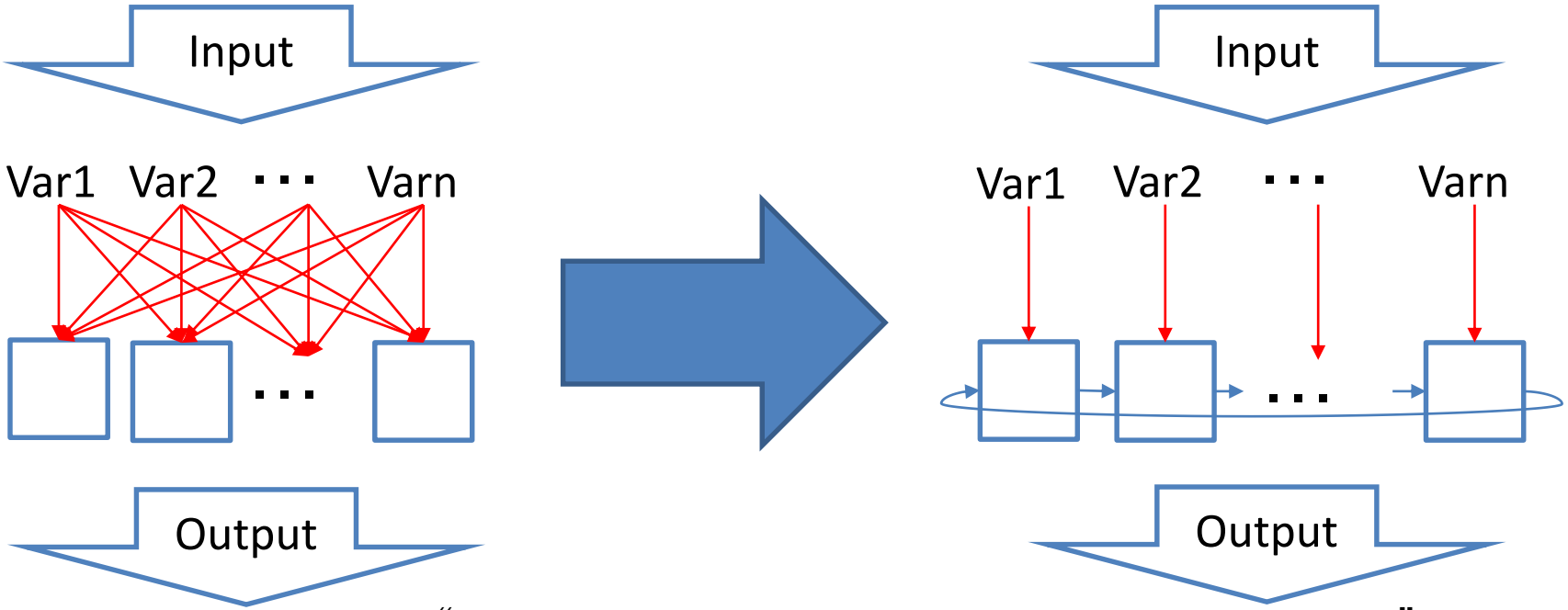


After automatic refinement



Use of template to generate regular structures and less communications

- Problem: Decompose an algorithm into a set of blocks which communicate less
- With templates, structural constraints can be added



Q. Wang, Y. Kimura , M. Fujita, "Template based synthesis for high performance computing," 2017 25th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), UAE, Oct 2017 to appear.

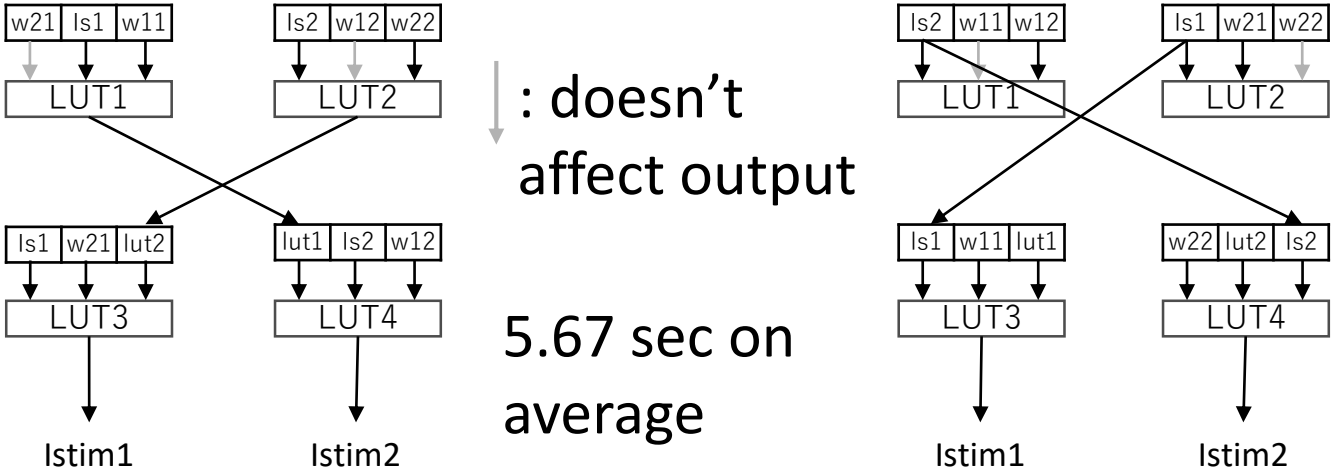
Synthesis example

$$\begin{pmatrix} I_{stim}1 \\ I_{stim}2 \end{pmatrix} = \begin{pmatrix} w11 & w12 \\ w21 & w22 \end{pmatrix} \cdot \begin{pmatrix} I_s1 \\ I_s2 \end{pmatrix}$$

2 × 2 Matrix Vector Product

with

2 cores connected mutually



- Correct dataflow was derived with 1 bit variables
- May get different types of solution as shown before

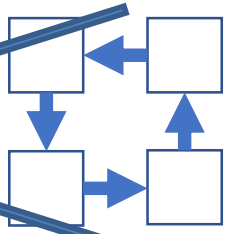
Learning additional constraints for larger problems

$$\begin{pmatrix} I_{stim}1 \\ I_{stim}2 \\ I_{stim}3 \\ I_{stim}4 \end{pmatrix} = \begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix} \begin{pmatrix} I_s1 \\ I_s2 \\ I_s3 \\ I_s4 \end{pmatrix}$$

4 × 4 Matrix Vector Product

Cannot solve

with



4 cores connected by ring

Make Small Instance

$$\begin{pmatrix} I_{stim}1 \\ I_{stim}2 \end{pmatrix} = \begin{pmatrix} w11 & w12 \\ w21 & w22 \end{pmatrix} \cdot \begin{pmatrix} I_s1 \\ I_s2 \end{pmatrix}$$

2 × 2 Matrix Vector Product

with

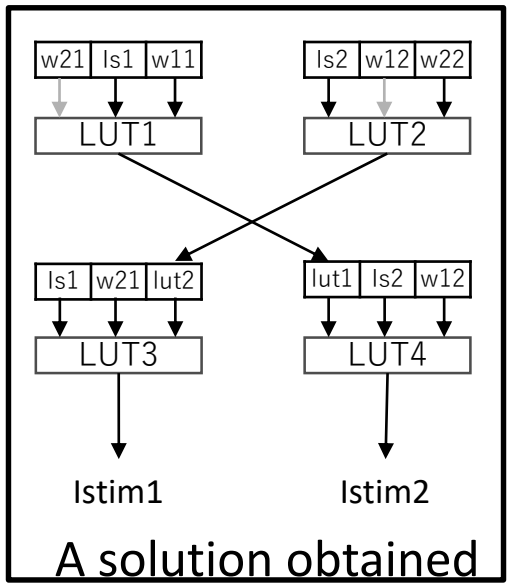
2 cores connected mutually

Easy to solve

Add Constraints on MUXs and LUTs

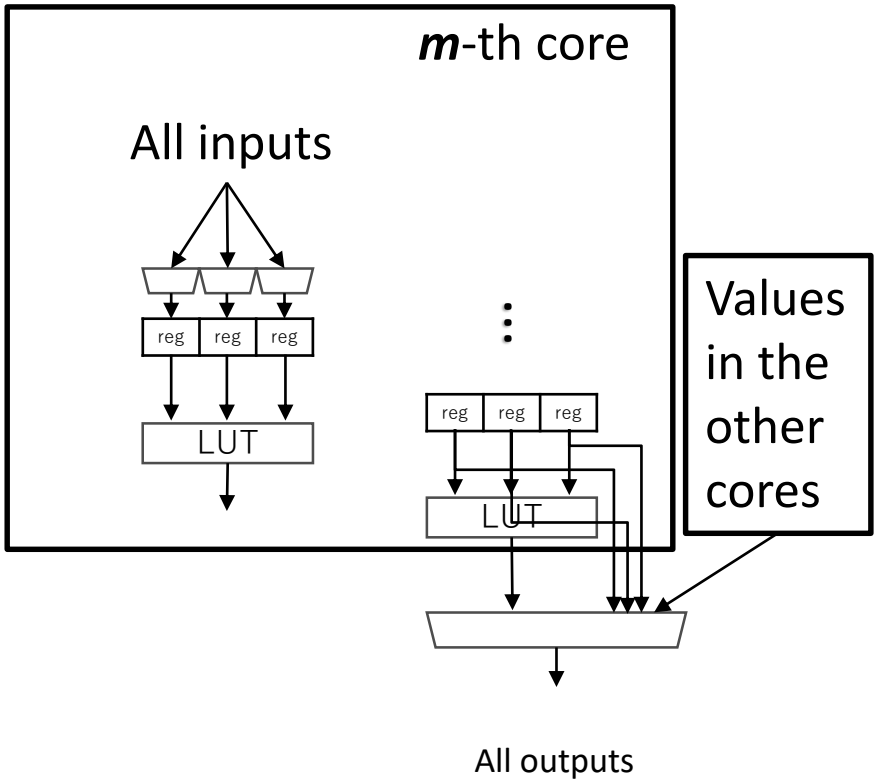
Analysis of solution obtained (1)

$$\begin{pmatrix} I_{stim}1 \\ I_{stim}2 \end{pmatrix} = \begin{pmatrix} w11 & w12 \\ w21 & w22 \end{pmatrix} \cdot \begin{pmatrix} I_s1 \\ I_s2 \end{pmatrix}$$ with



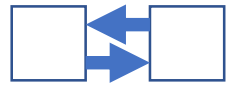
Template

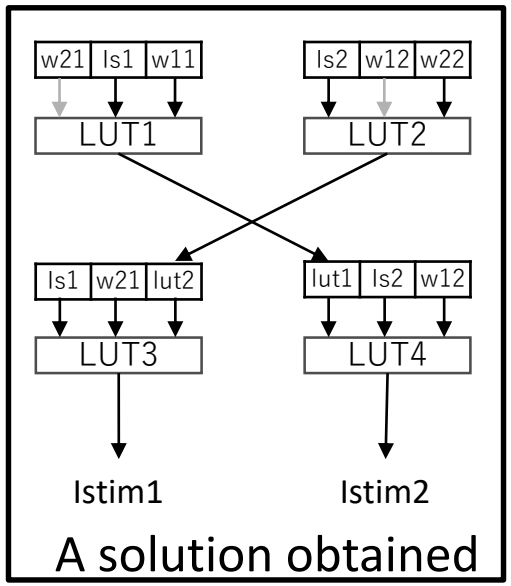
Synthesis



Analysis of solution obtained (2)

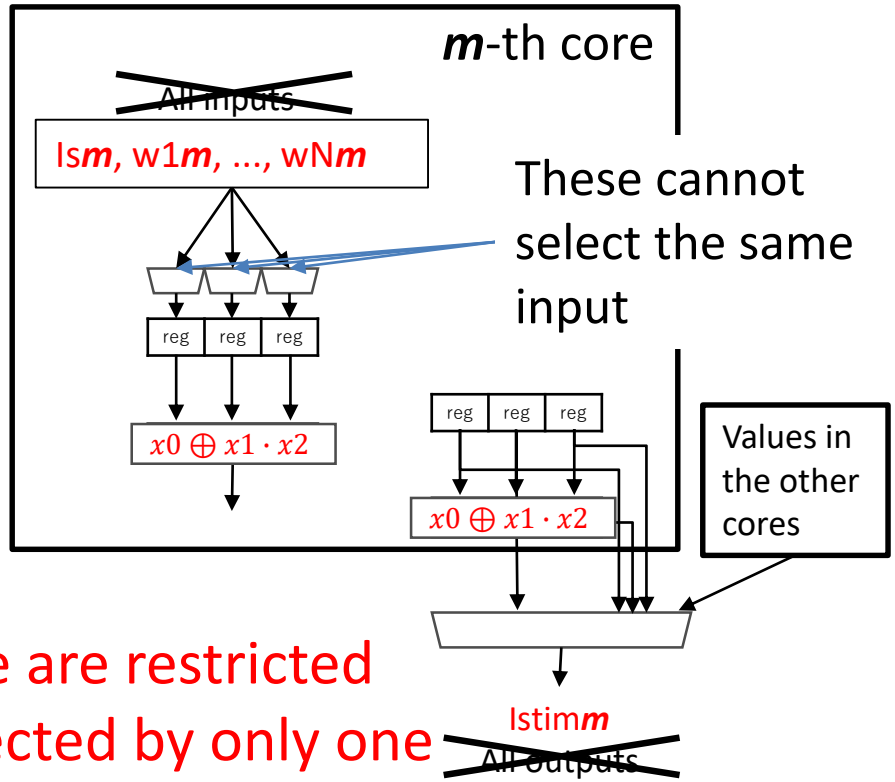
$$\begin{pmatrix} I_{stim}1 \\ I_{stim}2 \end{pmatrix} = \begin{pmatrix} w11 & w12 \\ w21 & w22 \end{pmatrix} \cdot \begin{pmatrix} I_s1 \\ I_s2 \end{pmatrix} \text{ with}$$





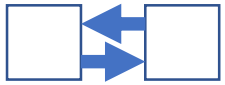
Add Constraints

Template

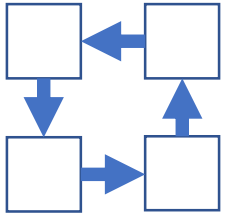


1. Inputs and output of each core are restricted
2. Each primary input can be selected by only one register
3. Functions of all LUTs are fixed to $x0 \oplus x1 \cdot x2$

Synthesis with Additional Constraints

$$\begin{pmatrix} I_{stim}1 \\ I_{stim}2 \end{pmatrix} = \begin{pmatrix} w11 & w12 \\ w21 & w22 \end{pmatrix} \cdot \begin{pmatrix} I_s1 \\ I_s2 \end{pmatrix}$$ with 

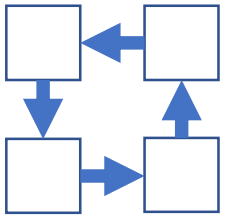
5.67sec → Additional Constraints → 0.16sec

$$\begin{pmatrix} I_{stim}1 \\ I_{stim}2 \\ I_{stim}3 \\ I_{stim}4 \end{pmatrix} = \begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix} \cdot \begin{pmatrix} I_s1 \\ I_s2 \\ I_s3 \\ I_s4 \end{pmatrix}$$ with 

Infeasible → Additional Constraints → 53.1sec

Further Example

Synthesis for

$$\begin{pmatrix} I_{stim}1 \\ \dots \\ I_{stim}32 \end{pmatrix} = \begin{pmatrix} w(1,1) & \dots & w(1,32) \\ \dots & \dots & \dots \\ w(32,1) & \dots & w(32,32) \end{pmatrix} \cdot \begin{pmatrix} I_s1 \\ \dots \\ I_s32 \end{pmatrix}$$ with 

32 × 32 Matrix Vector Product

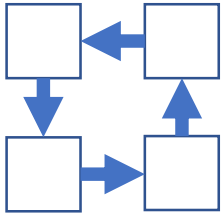
Infeasible

As explained before → Make Small Instance

2 × 2 Matrix Vector Product

with 

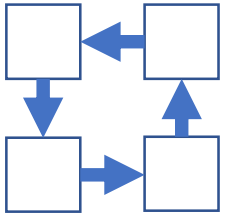
4 × 4 Matrix Vector Product

with 

→ **4 × 4** Matrix Vector Product

with 

→ **8 × 8** Matrix Vector Product

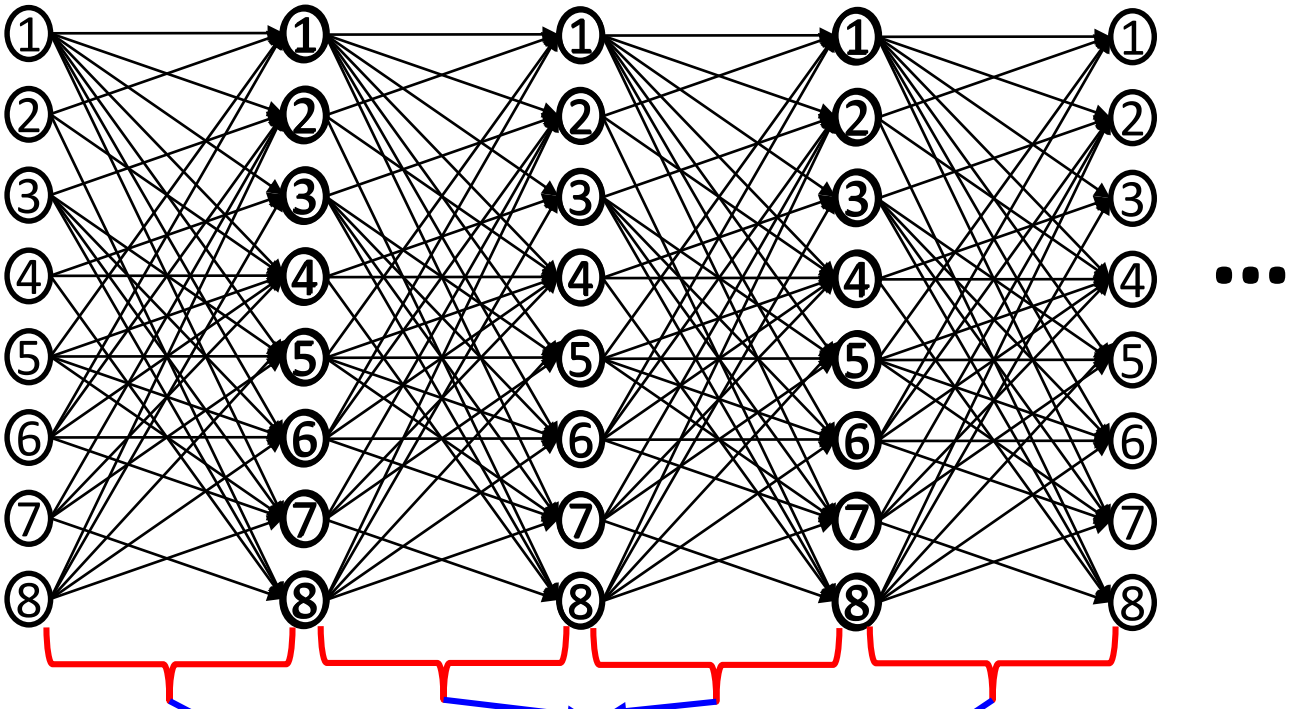
with 

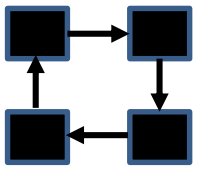
Application : Deep learning

- Each layer is processed one by one with M cores connected through one way ring communication

Connections are sparse and not like to perform multiplication by 0

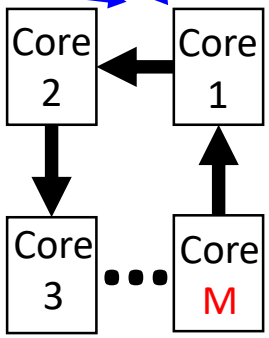
Overall computation should be accelerated by M times





= Specification

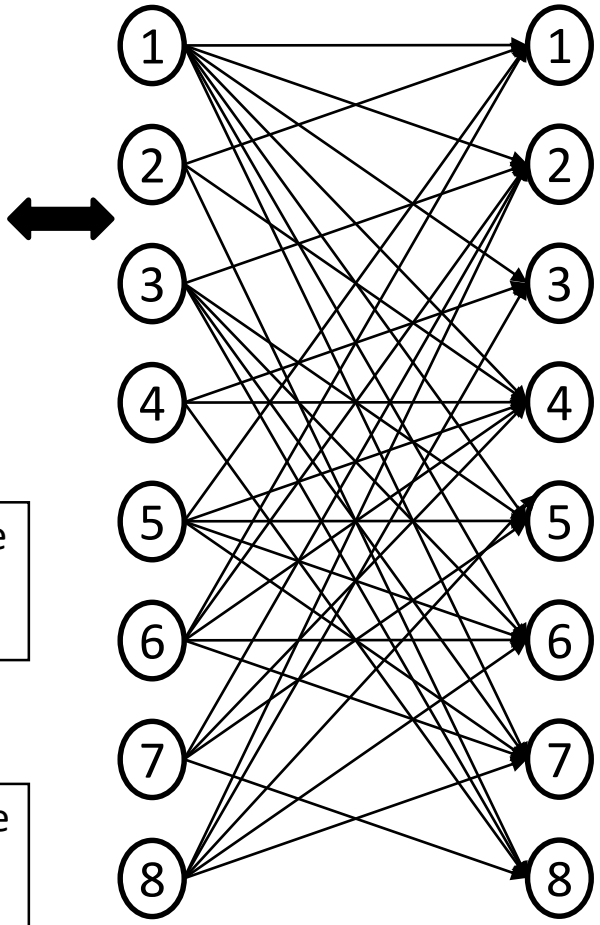
: Chip/Core



Target: Matrix-vector products (Lots of MAC)

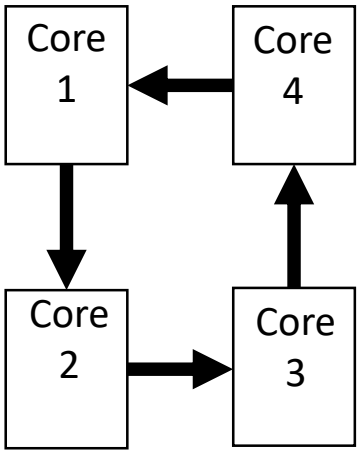
Sparse matrix is also OK to be compiled

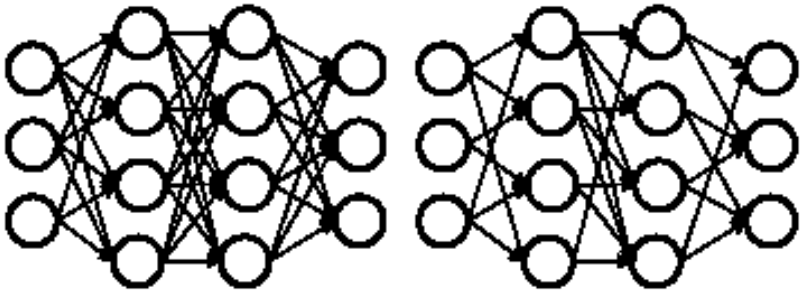
$$\begin{pmatrix} y1 \\ y2 \\ y3 \\ y4 \\ y5 \\ y6 \\ y7 \\ y8 \end{pmatrix} = \begin{pmatrix} w11 & w12 & w13 & w14 & w15 & w16 & w17 & 0 \\ w21 & 0 & 0 & w24 & 0 & 0 & 0 & w28 \\ 0 & w32 & 0 & 0 & w35 & w36 & w37 & w38 \\ 0 & 0 & w43 & w44 & 0 & 0 & 0 & w48 \\ w51 & 0 & 0 & w54 & w55 & w56 & w57 & 0 \\ w61 & w62 & 0 & w64 & 0 & w66 & w67 & 0 \\ 0 & w72 & 0 & w74 & w75 & 0 & 0 & w78 \\ 0 & w82 & w83 & 0 & w85 & w86 & w87 & 0 \end{pmatrix} \cdot \begin{pmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \\ x8 \end{pmatrix}$$



- 8*8-27=37
- 37/4=9.25 => 10 cycles
- Our method generate a scheduling with 10 cycles

on top of





(a) Full connection (b) Partial connection

Scheduling is very complicated and hard to understand at all !

Overview of Additional Constraints

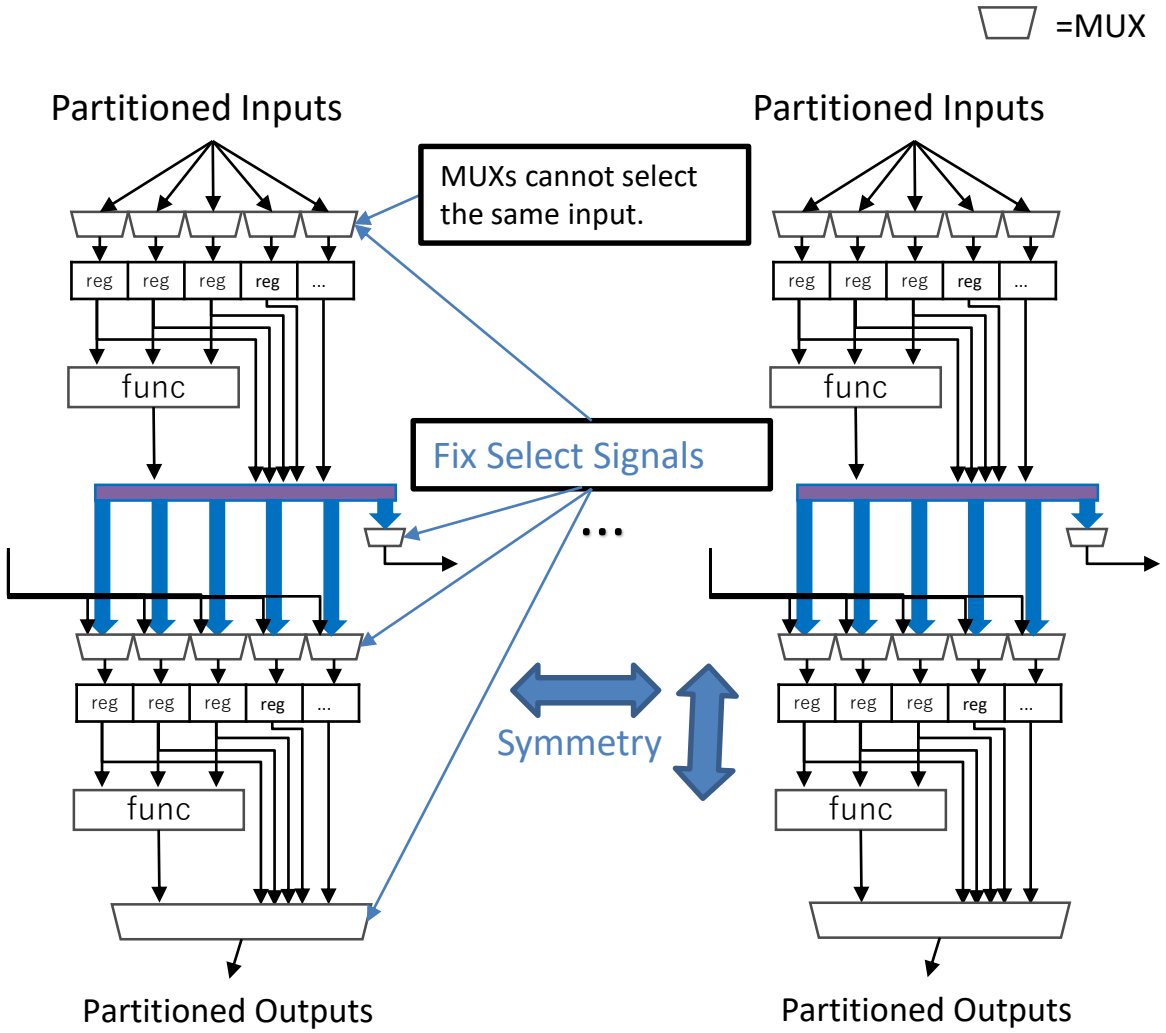
As explained before,

1. Partition Inputs/outputs
2. Each input can be used only once
3. Fix LUT function

In addition to these,

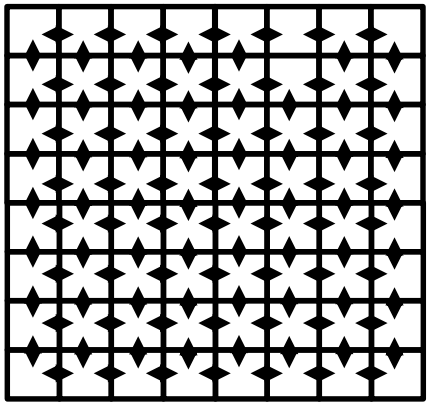
4. Fix some select signals of MUXs
5. Impose symmetry among cores and some cycles

Dataflow for 32 × 32 Matrix Vector Product was synthesized in 460sec

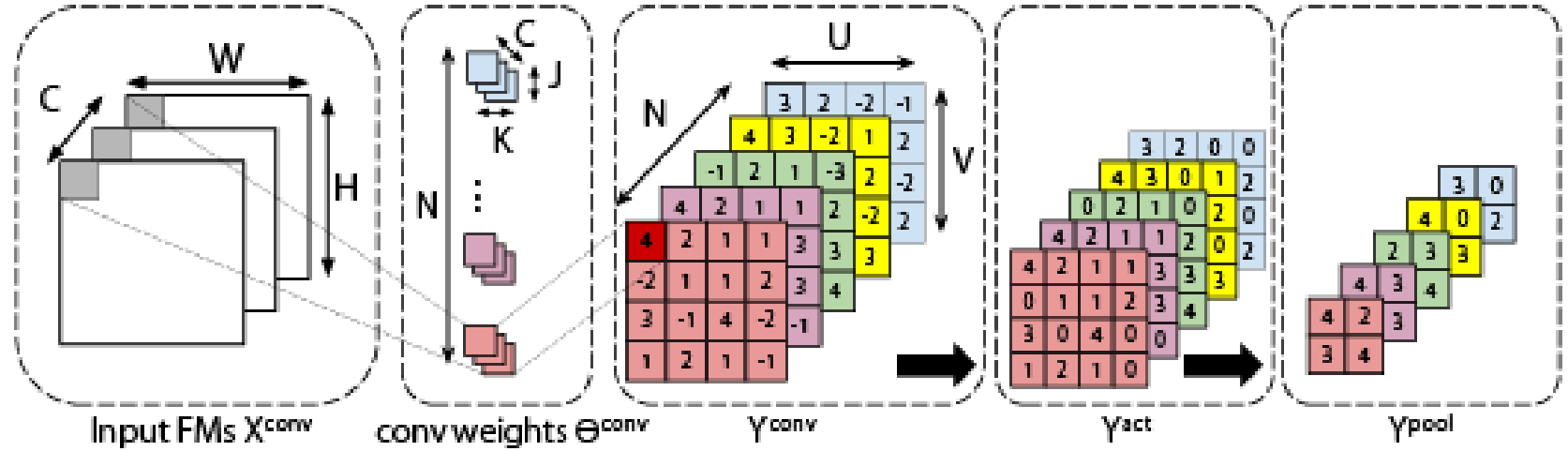


Application example: 1st layer of CNN for image classification

- Implement the 1st layer of the CNN
  - Typically used in image recognition/classification
  - Realize on mesh-architecture "optimally"

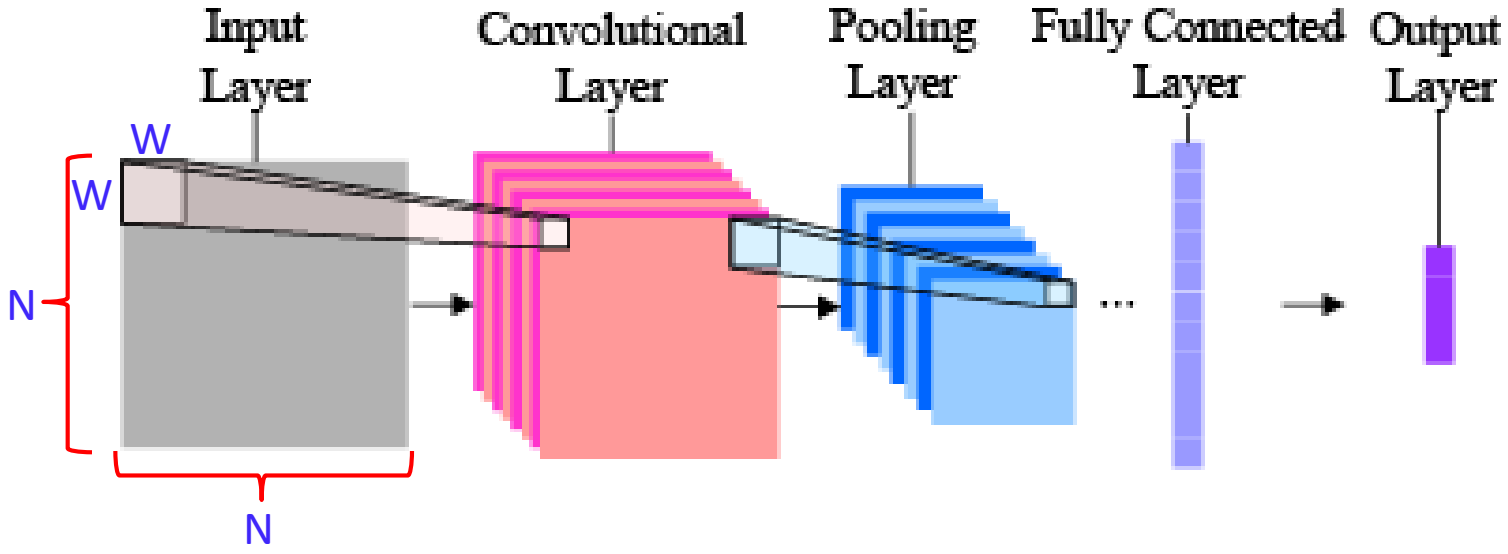


Implement this part on mesh-architectures only with 4 neighbor communication



N^2 parallel computation on NxN mesh

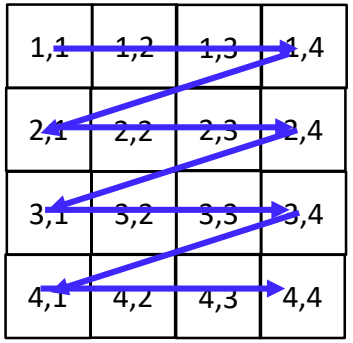
- NxN images on NxN mesh architecture (N^2 MAC units)
- Window size: WxW (N^2 such widows)
- In total $N^2 \cdot W^2$ MAC operations
- Theoretical optimum= $N^2 \cdot W^2 / N^2 = W^2$ cycles for all
- Typical numbers: N=128, W=4, and so all computations should finish in W^2=16 cycles



The key

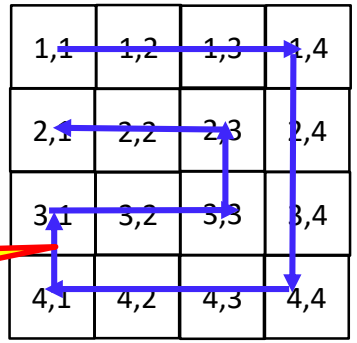
- Change the order of computation in convolution
  - Original:

    (1,1)→(1,2) →(1,3) →(1,4) →(2,1) →(2,2) →(2,3) →(2,4)→(3,1) →(3,2) →(3,3) →(3,4) →(4,1) →(4,2) →(4,3) →(4,4)



  - Proposed, for example:

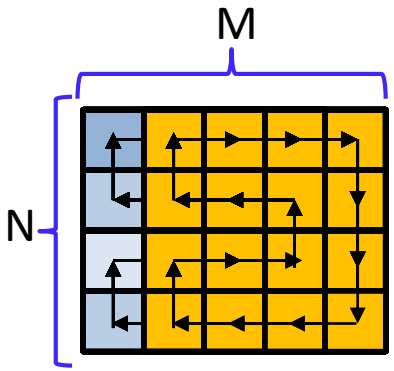
    (1,1)→(1,2) →(1,3) →(1,4) →(2,4) →(3,4) →(4,4) →(4,3)→(4,2) →(4,1) →(3,1) →(3,2) →(3,3) →(2,3) →(2,2) →(2,1)



This is a ring communication

Convolutional NN on mesh architecture

- Realizing theoretical optimum on mesh-architectures
- Mesh has N*M MAC units connected only to 4 neighbors
- There are around N*M windows
- With window size W, takes W^2 cycles for all computations
- Joint work with Dr. Alan Mishchenko of UCB
- We have no plan to apply for patent regarding to this algorithm!

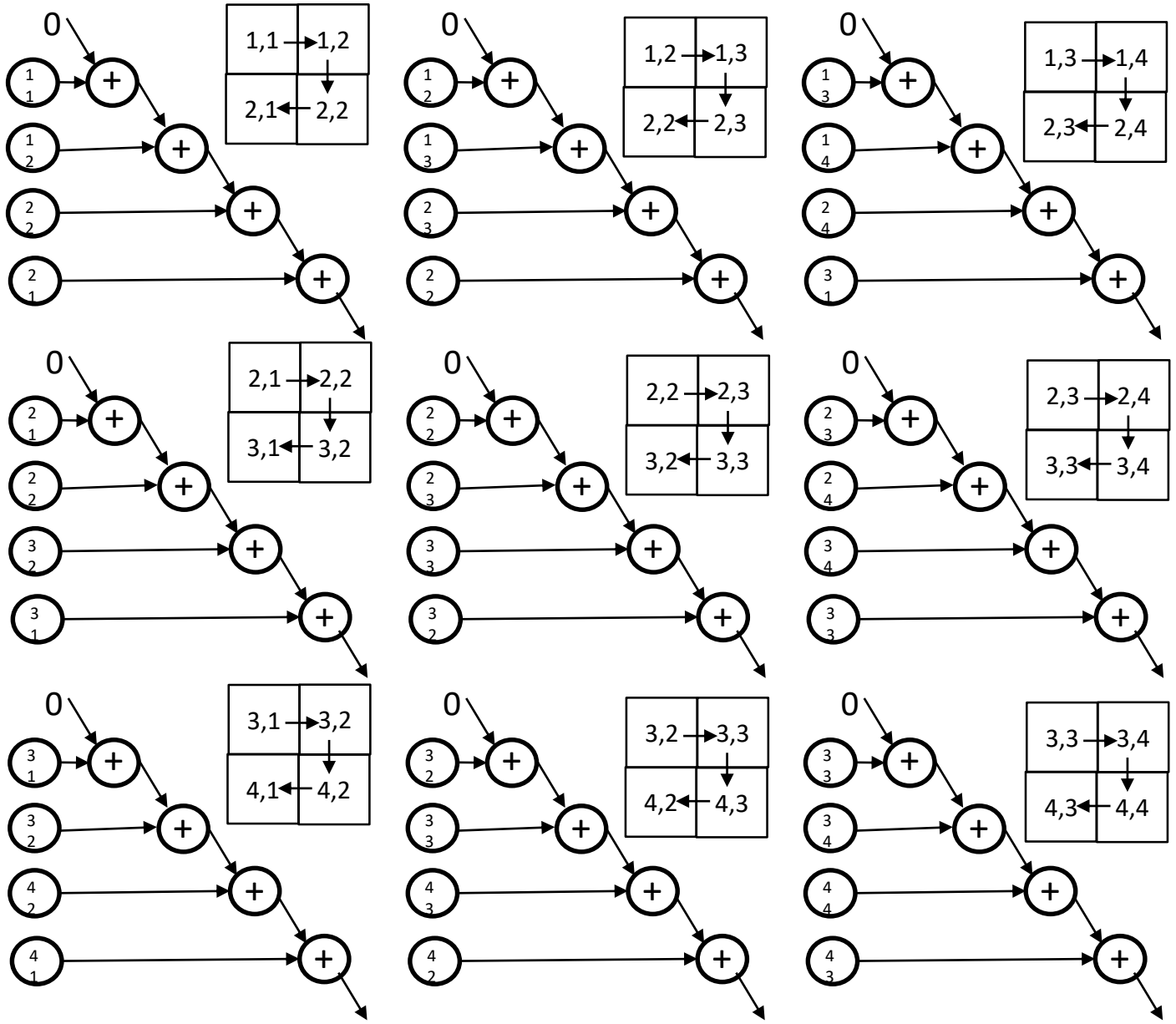


Typical numbers:
N=M=128, W=4
N=M=1024, W=10

DFG for automatic synthesis

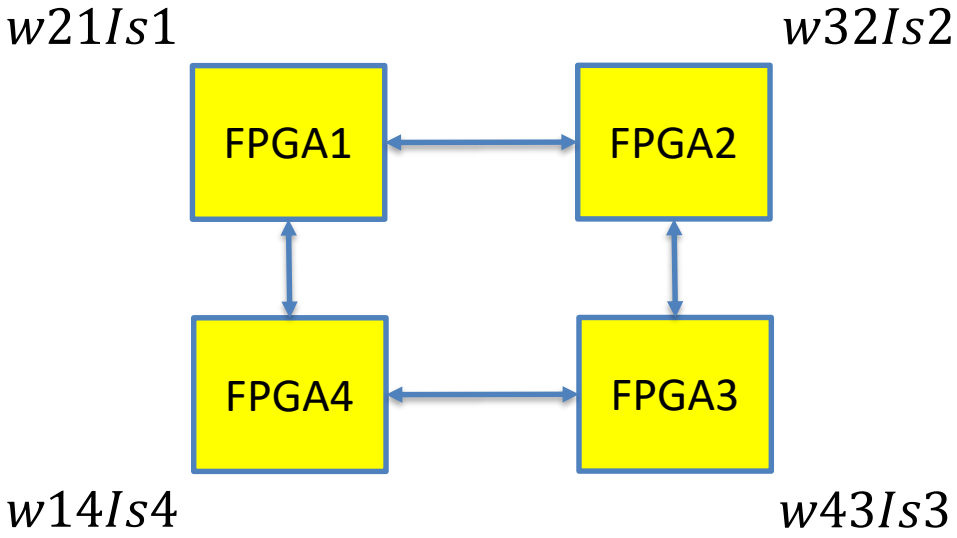
N=M=4, W=2

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4



Communication algorithm (1)

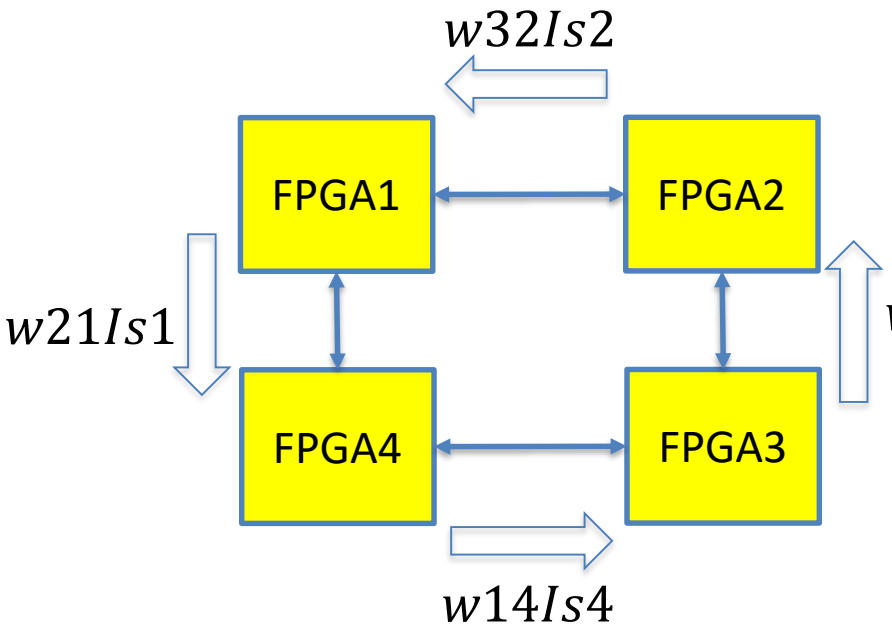
- N=4 (step1)
- All communications are in the same direction



$$\begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix}$$

Communication algorithm (2)

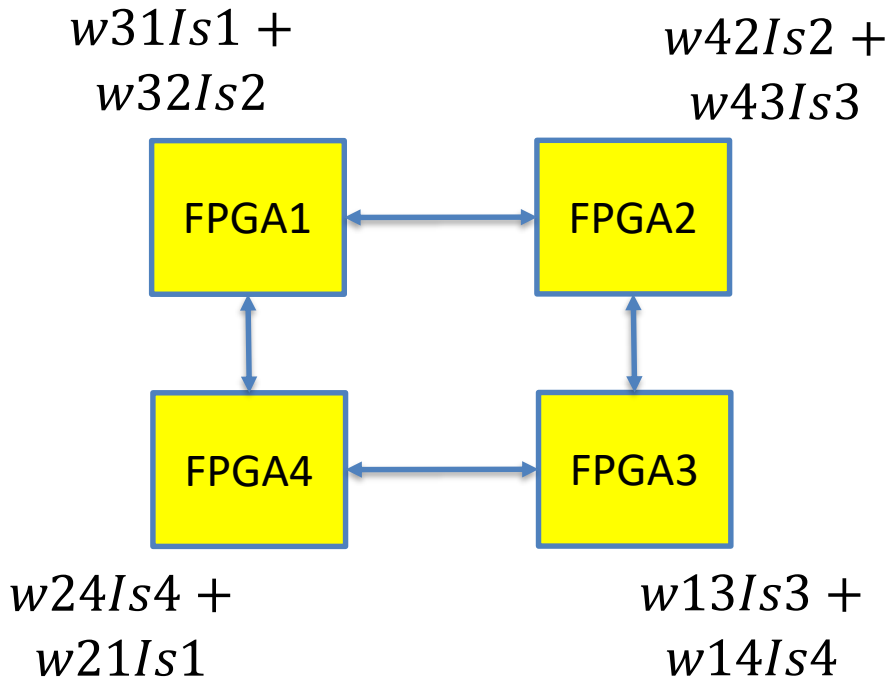
- N=4 (step2)
- All communications are in the same direction



$$\begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix}$$

Communication algorithm (3)

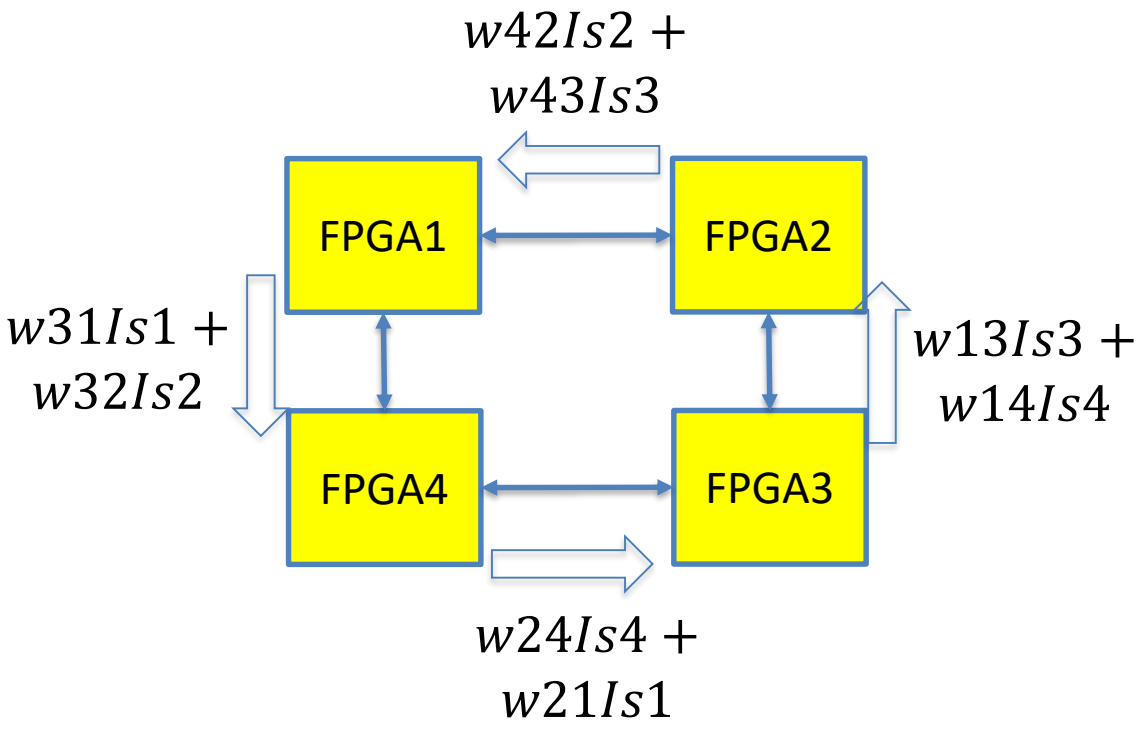
- N=4 (step3)
- All communications are in the same direction



$$\begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix}$$

Communication algorithm (4)

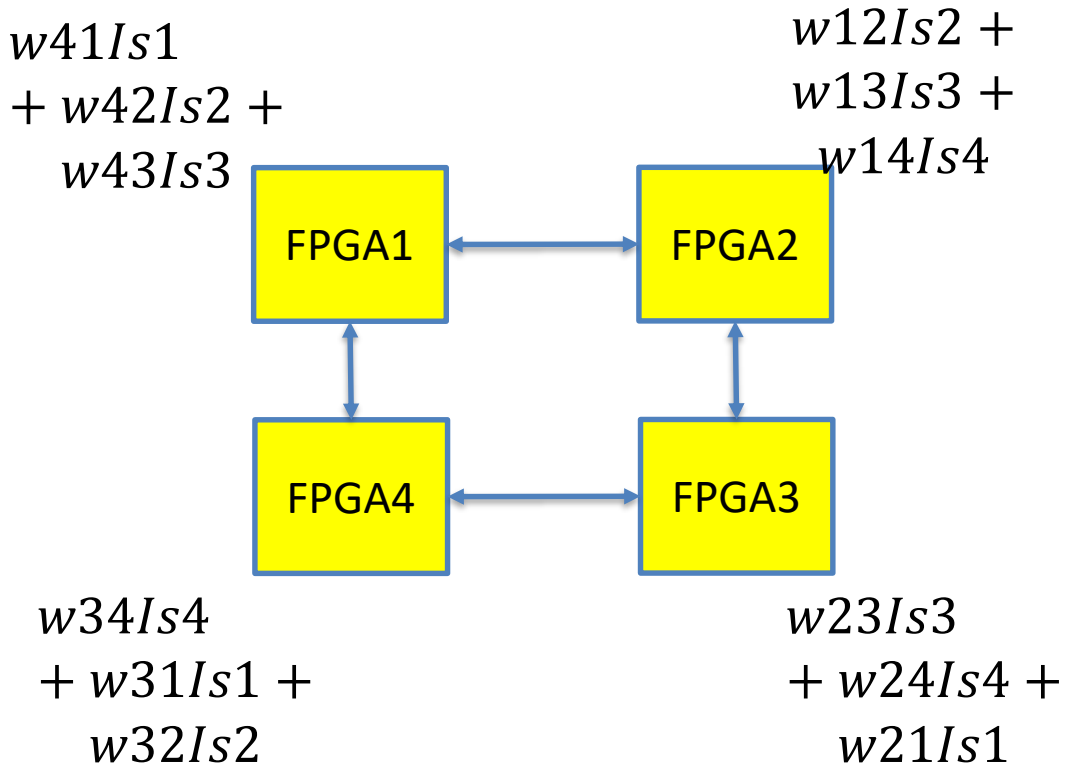
- N=4 (step4)
- All communications are in the same direction



$$\begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix}$$

Communication algorithm (5)

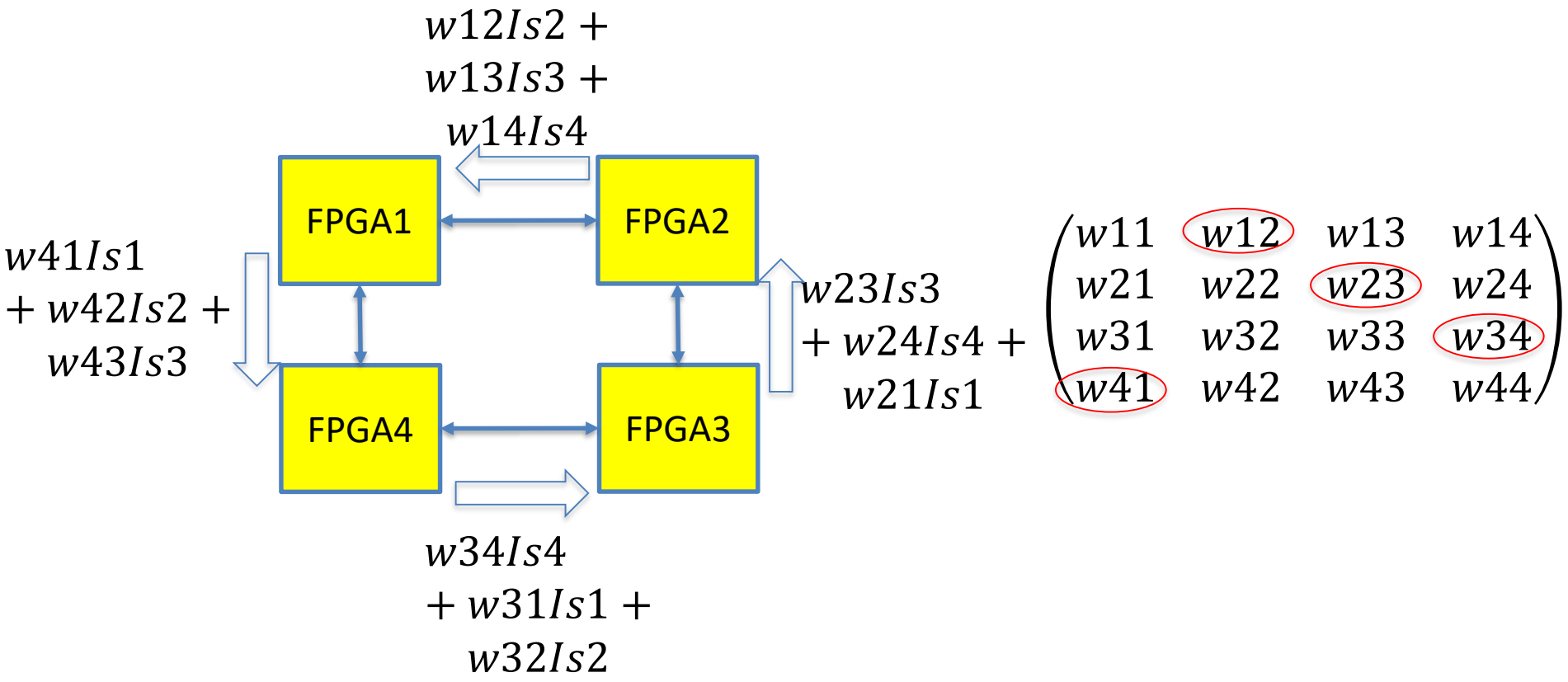
- N=4 (step5)
- All communications are in the same direction



$$\begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix}$$

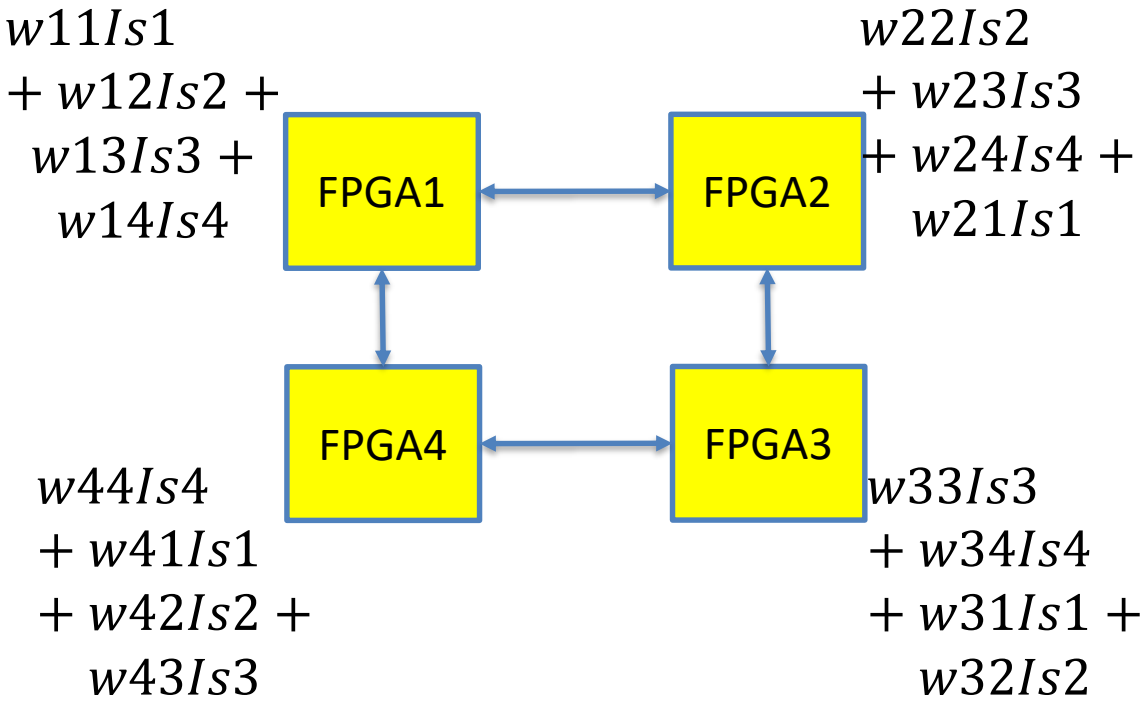
Communication algorithm (6)

- N=4 (step6)
- All communications are in the same direction



Communication algorithm (7)

- N=4 (step7)
- All communications are in the same direction

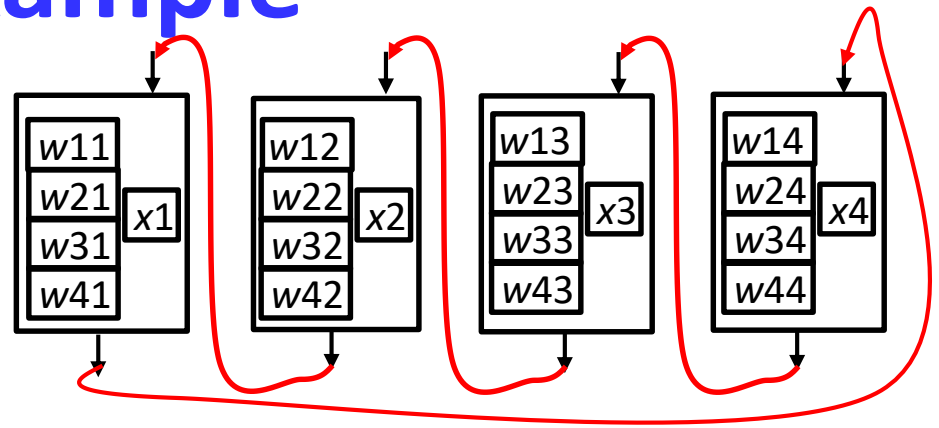


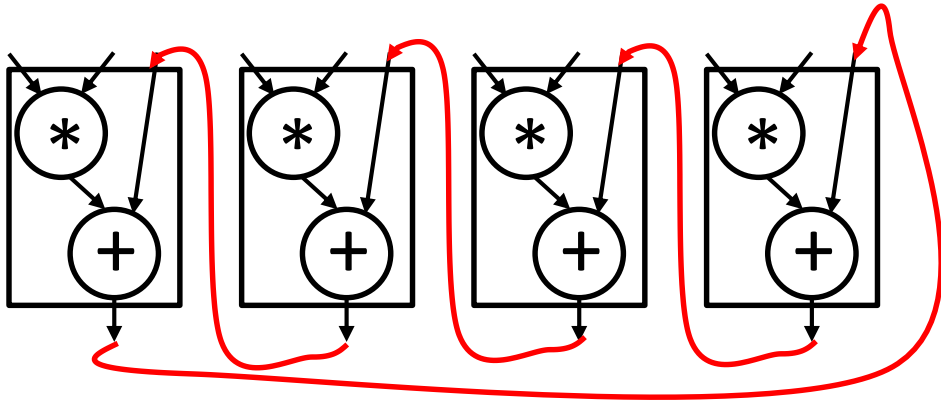
$$\begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix}$$

Template example

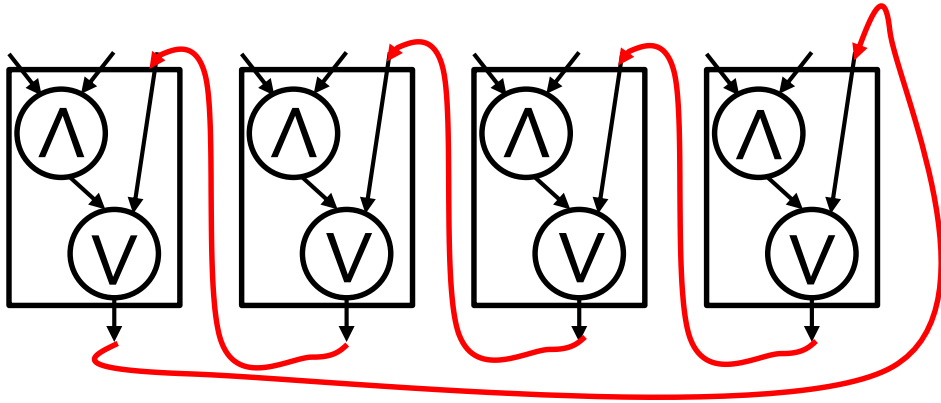
$$\begin{pmatrix} y1 \\ y2 \\ y3 \\ y4 \end{pmatrix} = \begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix} \cdot \begin{pmatrix} x1 \\ x2 \\ x3 \\ x4 \end{pmatrix}$$

- Can be solved in less than one second
- Even with a single FPGA chip implementation, clock speed increases by 90% or more!
  - Cycle time after P&R
    - Original DFG: 2.98ns (335.1MHz)
    - Synthesized DFG: 1.62ns (616.1MHz)





Automatic abstraction $* \Rightarrow \wedge \quad + \Rightarrow \vee$



Proposed processing flow