



[졸업작품전 - 작품]

VeriLaygo : Automated Layout Generator for custom circuits

- 이동현 : 융합전자공학부 2019071476
- 한진규 : 융합전자공학부 2019029061



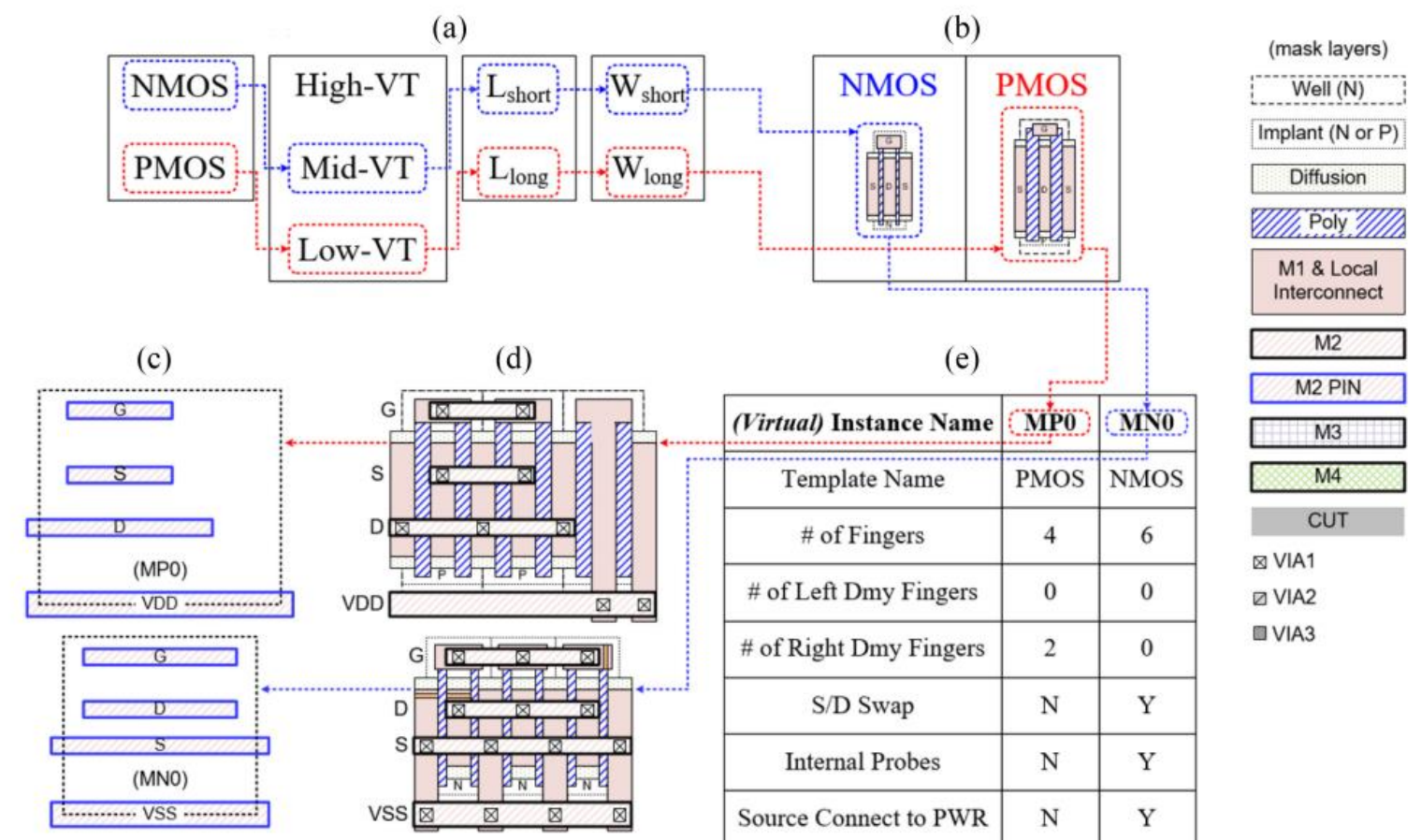
What is VeriLaygo?

Verilog

- + Behavioral Design을 통해 구체적인 공정 사양과 분리된 하드웨어 동작 설계
- + 모듈화에 용이, 하드웨어의 계층적 구조를 간결하게 표현가능
- + 설계자 입장에서 높은 추상화 수준에서 설계 가능
- Layout 수준에서의 customizing에는 적합하지 못함

Laygo

- + 공정에 따라 다른 템플릿들을 활용 가능한 layout 생성 자동화 프레임워크



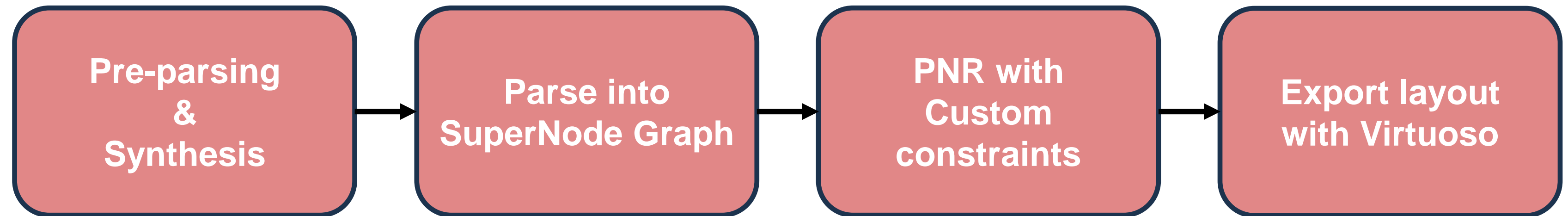
What is VeriLaygo?

Verilog + Laygo

From Behavioral Design to Custom Layouts

- ❑ Verilog의 Behavioral Design을 통한 high-level에서의 설계의 장점
- ❑ Layout 수준에서의 여러 custom constraint를 적용할 수 있도록 하여 다양한 공정에 대한 custom layout를 자동 생성할 수 있는 워크플로우

VeriLaygo Workflow



- Pre-parse custom syntax
- Synthesis via yosys

- Hierarchical graph structure
- Save to JSON file

- Placement using Force-Directed placement algorithm
- Apply custom constraints for placement
- Routing using Laygo

- Export layout to Virtuoso using Laygo export

VeriLaygo Workflow

1) Pre-parsing

❑ Behavioral Verilog code를 먼저 parsing 하여 layout에 적용할 옵션을 나타내는 특수문법 추가

- Number of rows
- Symmetry matching
- Critical net identification

```
module fulladder_1bit(  
    input wire A;  
    input wire B;  
    input wire Cin;  
    output reg S;  
    output reg Cout;  
  
    \nrows=2  
    \sym=HA1/HA2  
);  
  
    wire sum1, carry1, carry2;  
  
    halfadder_1bit HA1(.A(A), .B(B), .S(sum1), .Cout(carry1));  
    halfadder_1bit HA2(.A(sum1), .B(Cin), .S(S), .Cout(carry2));  
    assign Cout = carry1 | carry2;  
  
endmodule
```

VeriLaygo Workflow

1) Synthesis

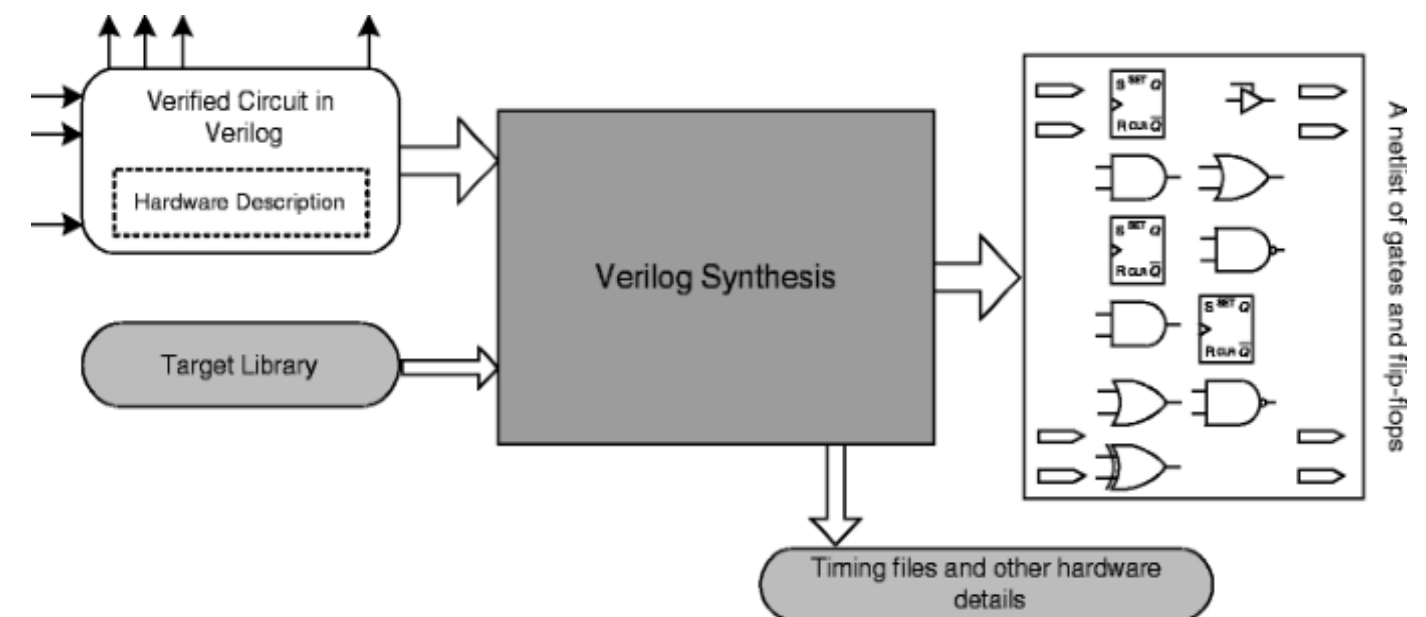
- ❑ Open-source synthesis tool Yosys 활용
- ❑ 타겟 공정에 대한 standard cell library 파일과 parsing 된 Verilog code를 입력
- ❑ 입력된 lib파일 기반으로 게이트들과 netlist 추출

```
read_verilog halfadder_1bit.v
read_verilog fulladder_1bit.v
read_verilog fulladder_2bit.v
hierarchy -top fulladder_2bit
synth
dfflibmap -liberty mycells.lib
abc -liberty mycells.lib
write_verilog fulladder_2bit_post_pynth.v
```

Multiple
Behavioral level
Verilog codes

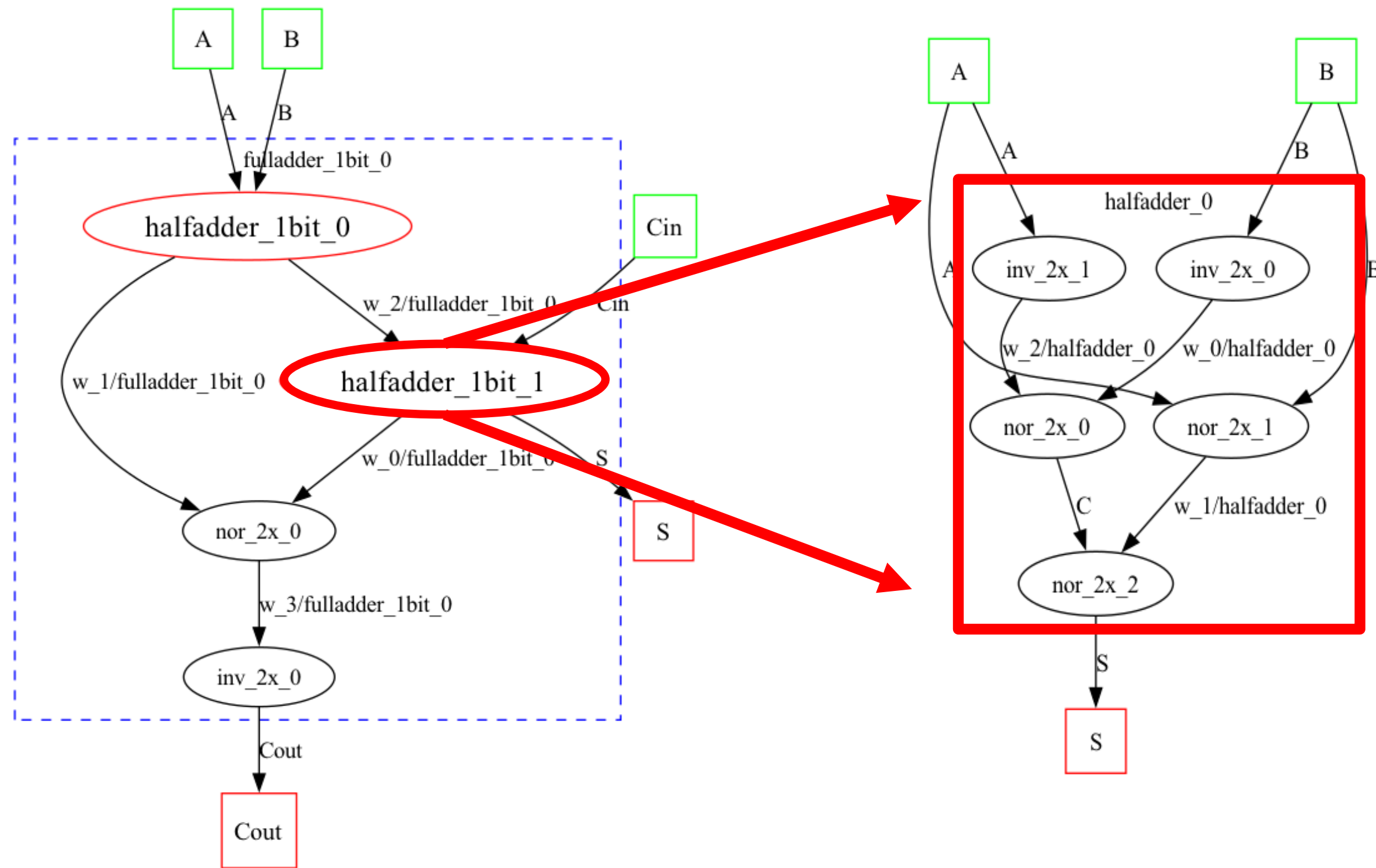
Standard cell library

Synthesized output



VeriLaygo Workflow

2) Parse instances into SuperNode Graph



Node

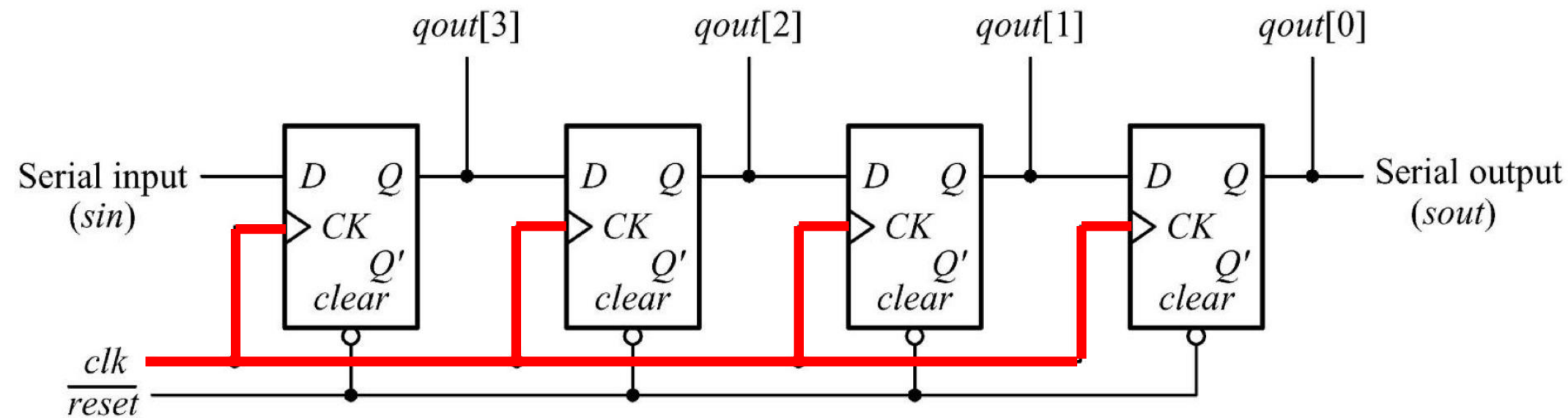
- predefined cells (leaf nodes)
- e.g. inv_2x, nor_2x, dff_2x, etc.

SuperNode

- Modules from verilog files
- e.g. fulladder_1bit, halfadder_1bit

VeriLaygo Workflow

2) Parse instances into SuperNode Graph



❑ Critical net identification (represented in red)

- 4bit shift register의 clock 신호를 critical net으로 지정
- Critical net과 연결된 instance들은 placement 단계에서 수평적으로 배치

VeriLaygo Workflow

3) Placement with Force-Directed placement

물리 시뮬레이션 기반 placement optimization

□ Basic forces

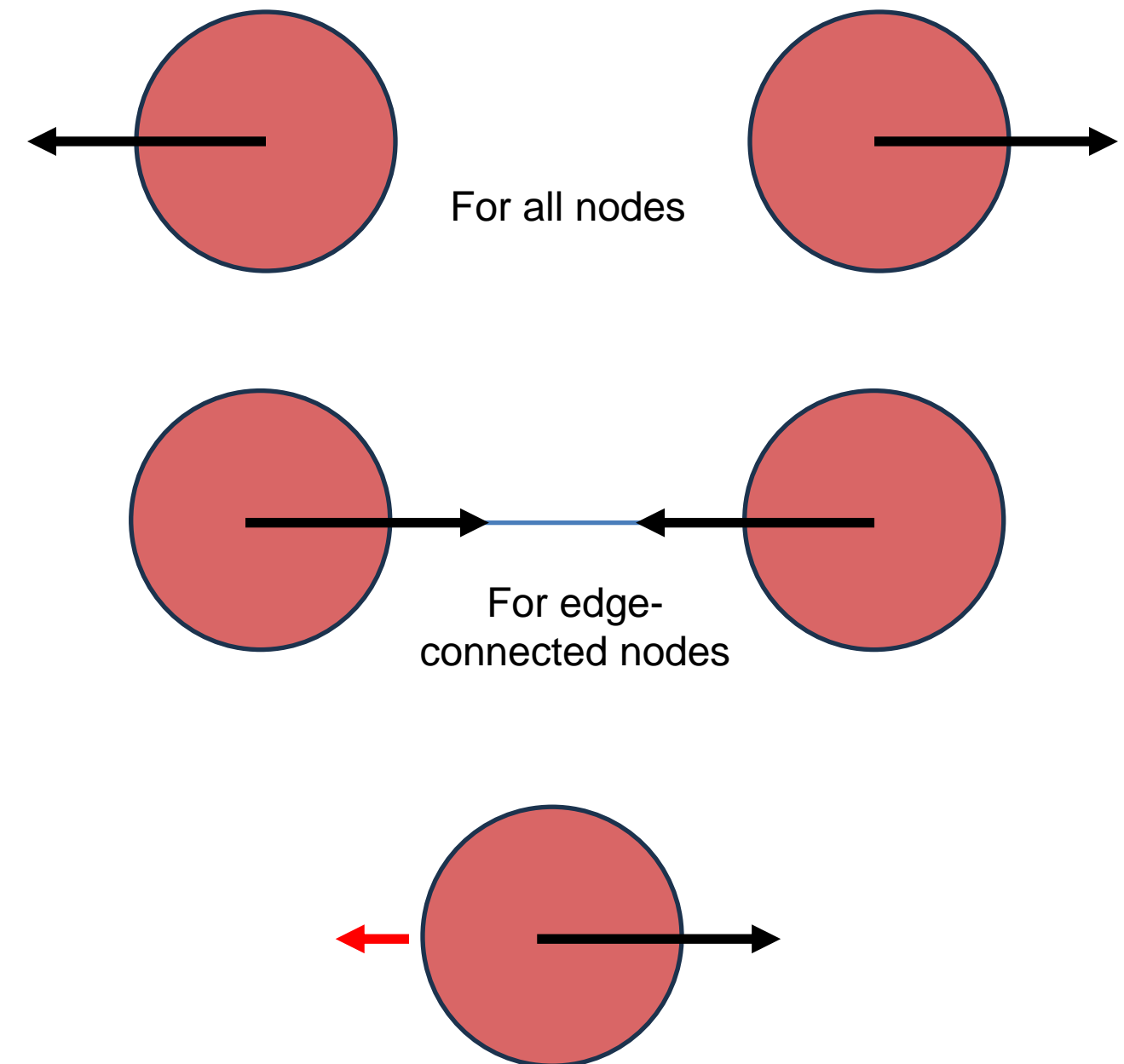
- Coulombic force

모든 Node 간의 척력

- Spring force

Edge로 연결된 Node 간의 인력

- Friction



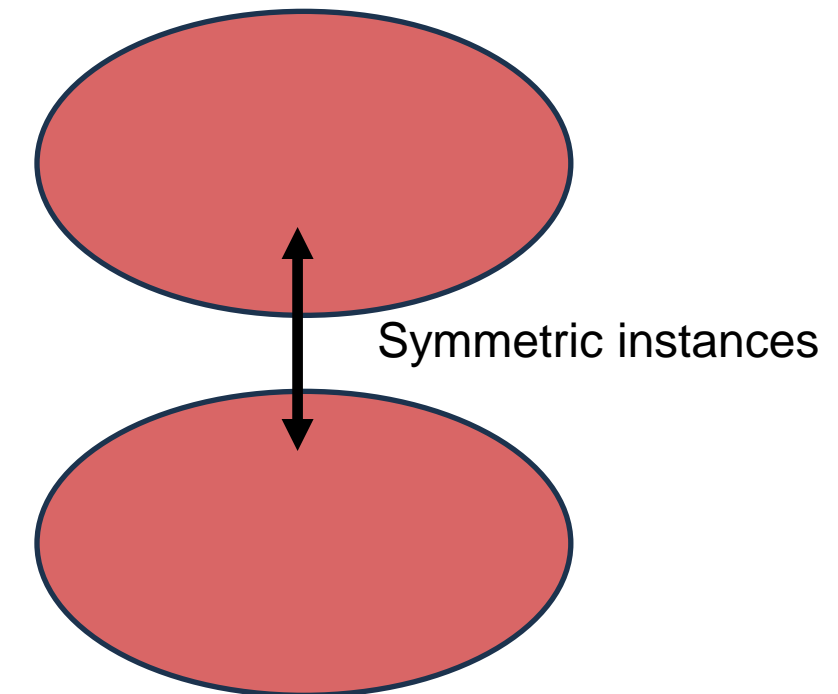
VeriLaygo Workflow

3) Placement with Force-Directed placement

❑ Custom constraints

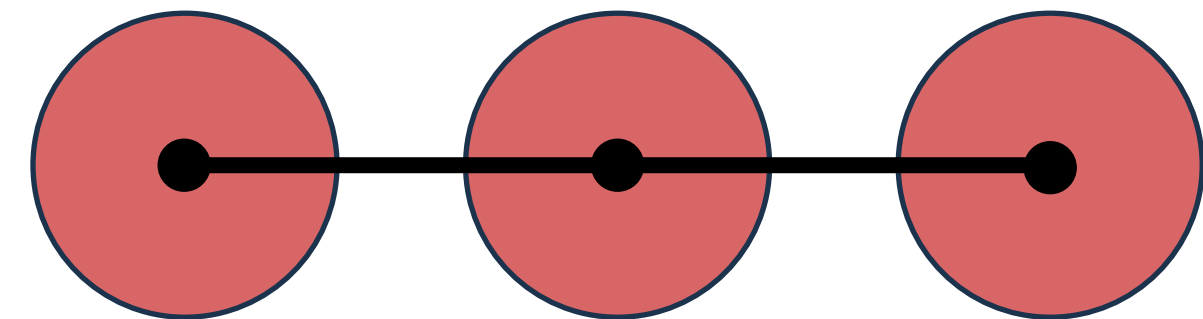
- Symmetry matching

동일한 instance들을 대칭적 위치에 배치



- Critical nets

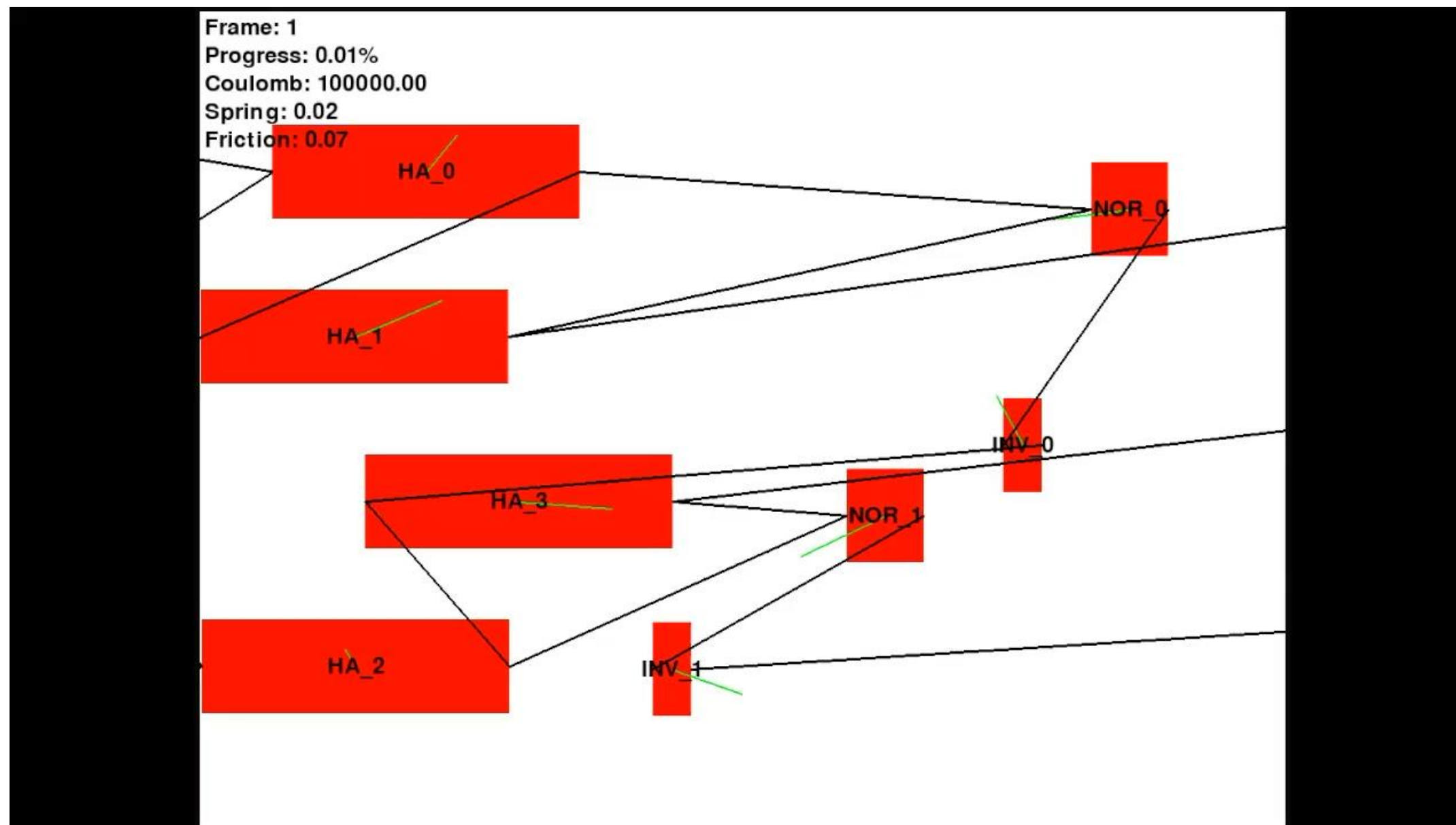
Critical net과 연결된 instance들을 수평적으로 결합



Critical net connected instances

VeriLaygo Workflow

3) Placement using Force-Directed placement



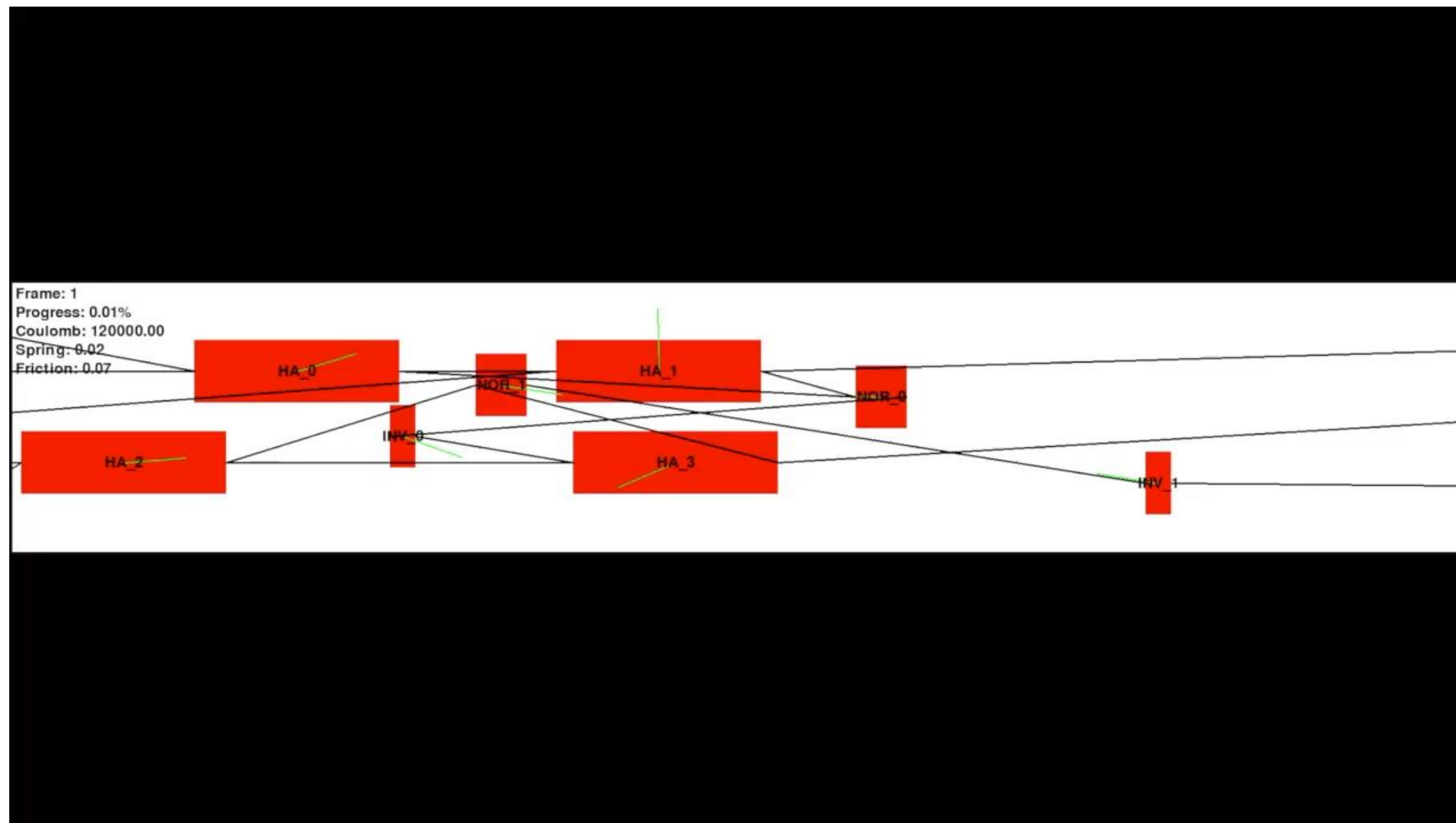
□ Placement of 2bit full adder

□ Constraints

- number of rows = 4
- symmetry matching

VeriLaygo Workflow

3) Placement using Force-Directed placement



□ Placement of 2bit full adder

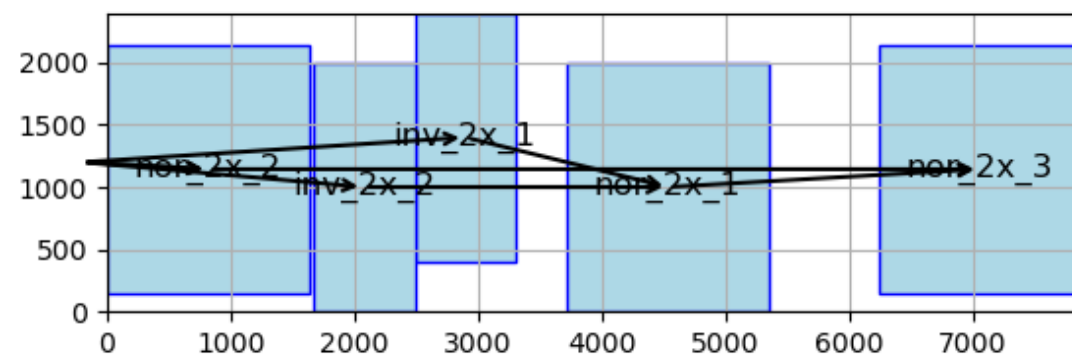
□ Constraints

- number of rows = 2
- symmetry matching

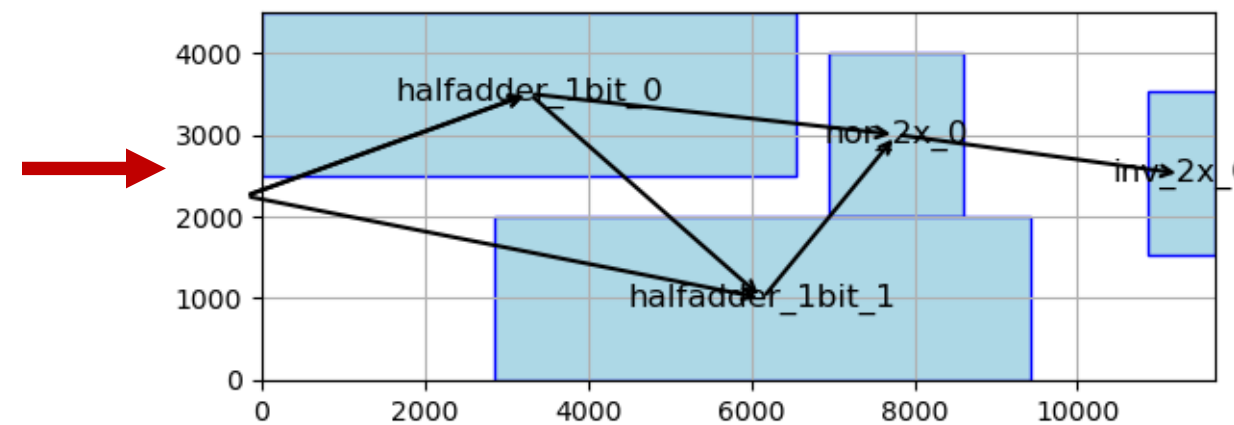
VeriLaygo Workflow

3) PNR with Laygo

- 가장 depth가 깊은 SuperNode부터 placement 진행
- Placement가 완료된 width, height 정보는 상위모듈로 전달되어 재귀적 placement 진행
- 이후 Laygo를 활용하여 Routing 진행

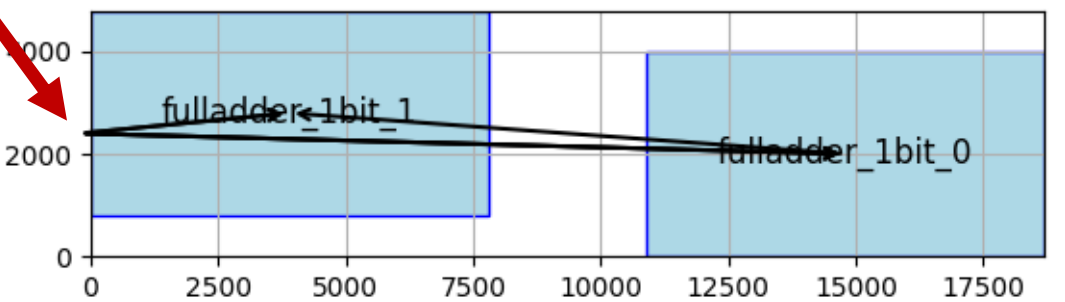
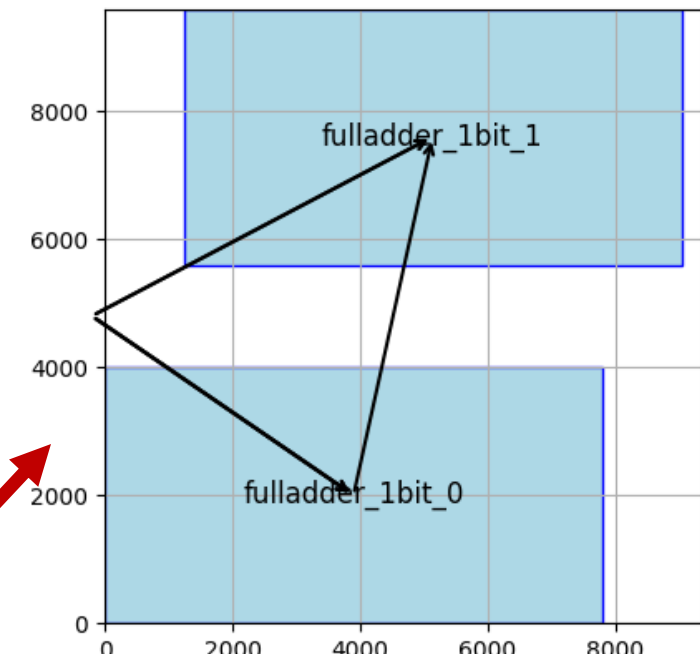


□ halfadder_1bit / row = 1



□ fulladder_1bit / row = 2

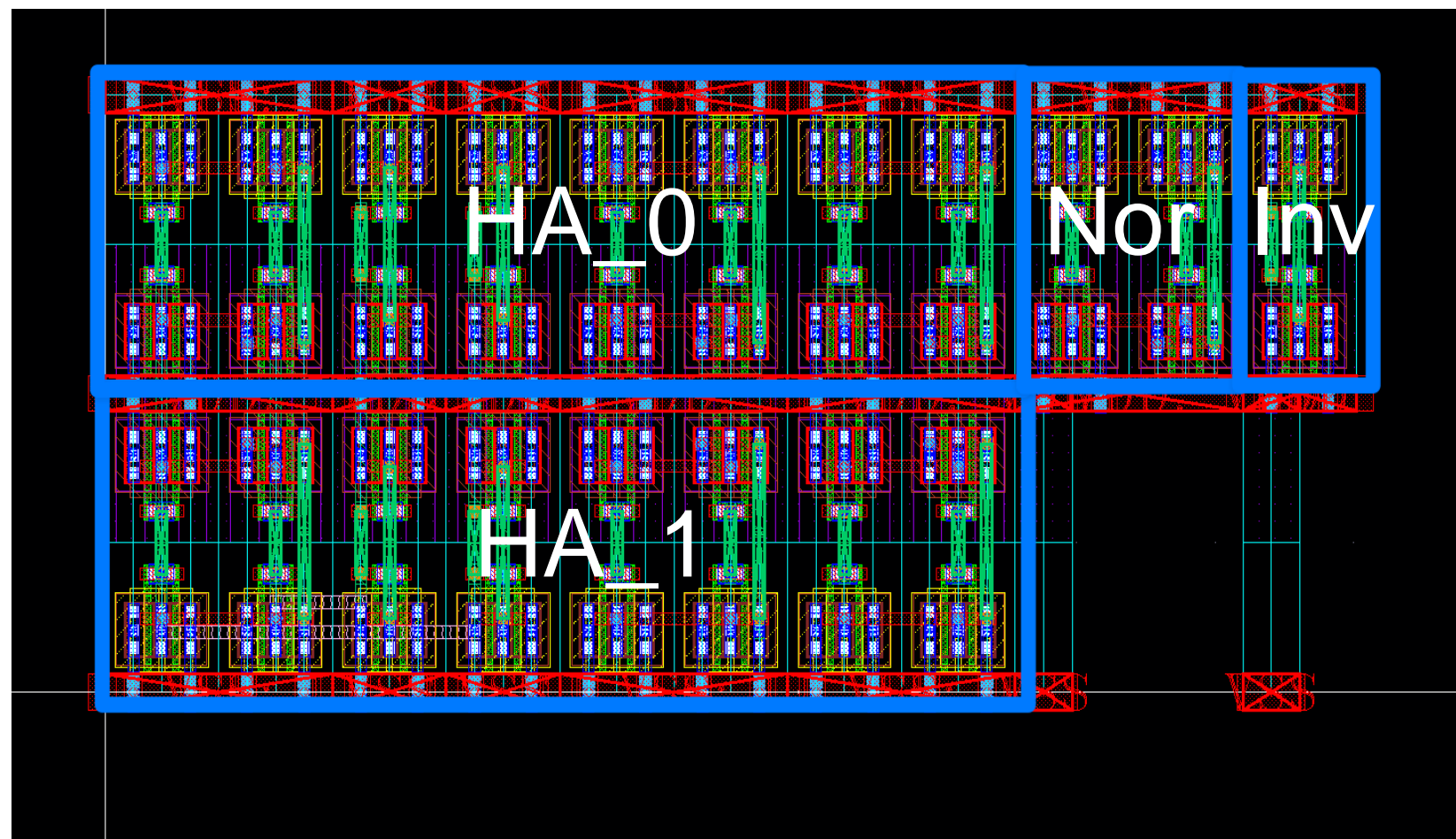
□ fulladder_2bit / row = 2



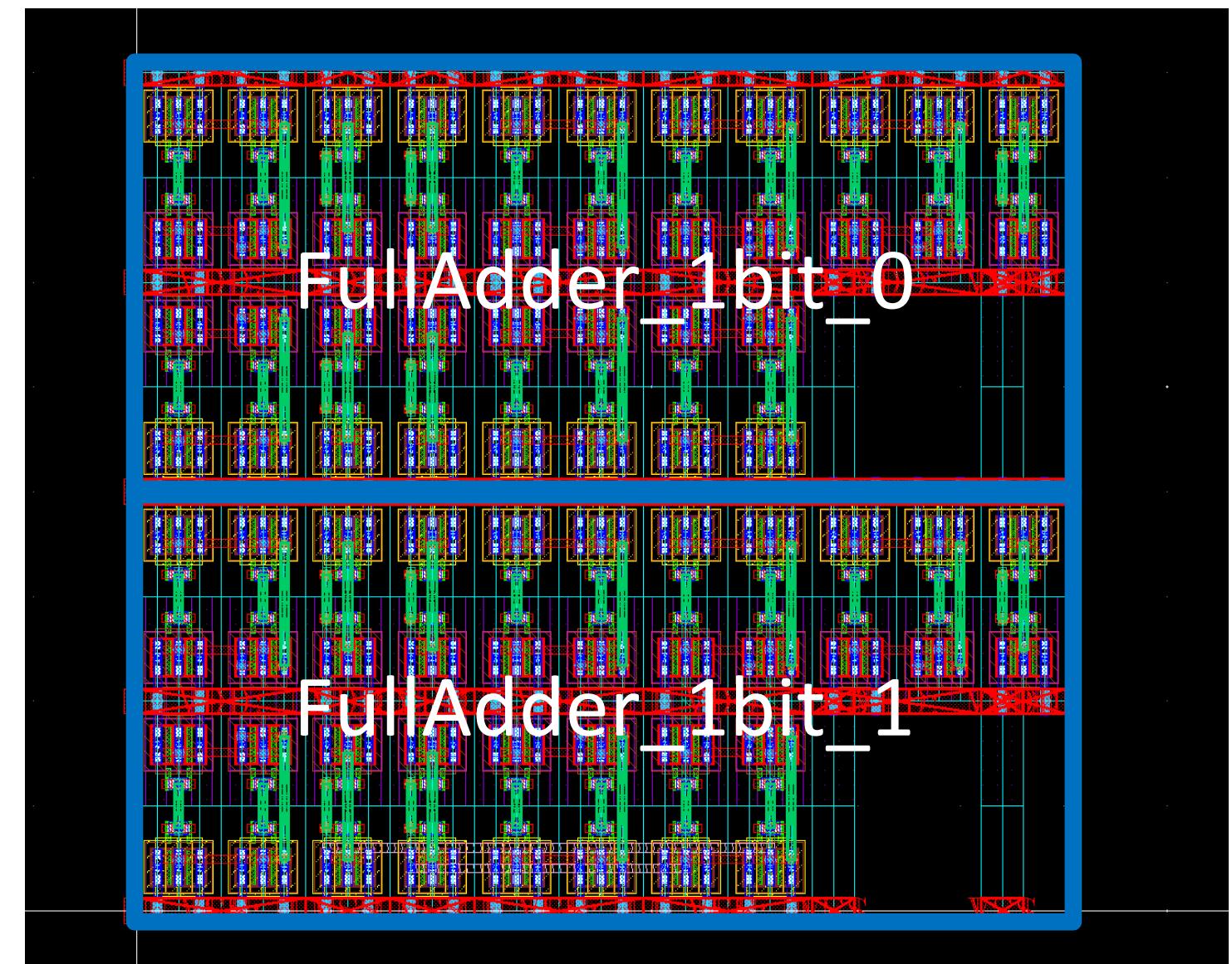
□ fulladder_2bit / row = 1

VeriLaygo Workflow

4) Export Layout to Virtuoso



1bit full adder with 2 rows



2bit full adder with 2 rows

Conclusion

- ❑ Verilog의 문법에 layout에 대한 옵션을 추가하는 방식으로 **high-level과 low-level 설계를 모두 고려**할 수 있는 워크플로우 구현
- ❑ Verilog code에 layout의 요구사항을 추가하여 **custom circuit의 손쉬운 생성** 가능
- ❑ Placement 단계에서 회로의 특성에 맞는 custom force를 정의하여 추가적인 최적화 적용 가능

Appendix I

Data structure of graphs

mycells.json
+
logic_generated_templates.json
+
fulladder_2bit_dict.json



❑ fulladder_2bit_structural.json

- Contains all modules + predefined cells
- Dictionary containing width, depth, pins, sub_blocks, input, output, wires, etc.
- Netname overwriting for identification of Critical Nets

```
{
  "top": "fulladder_2bit",
  "cellname": "fulladder_2bit",
  "libname": "verilog_generated",
  "name": "fulladder_2bit_0",
  "hier_name": "fulladder_2bit_0",
  "width": 31160,
  "depth": 0,
  "input": [
    "A[0]",
    "A[1]",
    "B[0]",
    "B[1]",
    "Cin"
  ],
  "output": [
    "Cout",
    "S[0]",
    "S[1]"
  ],
  "wire": [
    "A[0]",
    "A[1]",
    "B[0]",
    "B[1]",
    "Cin",
    "Cout",
    "S[0]",
    "S[1]",
    "w_0/fulladder_2bit_0"
  ],
}
```

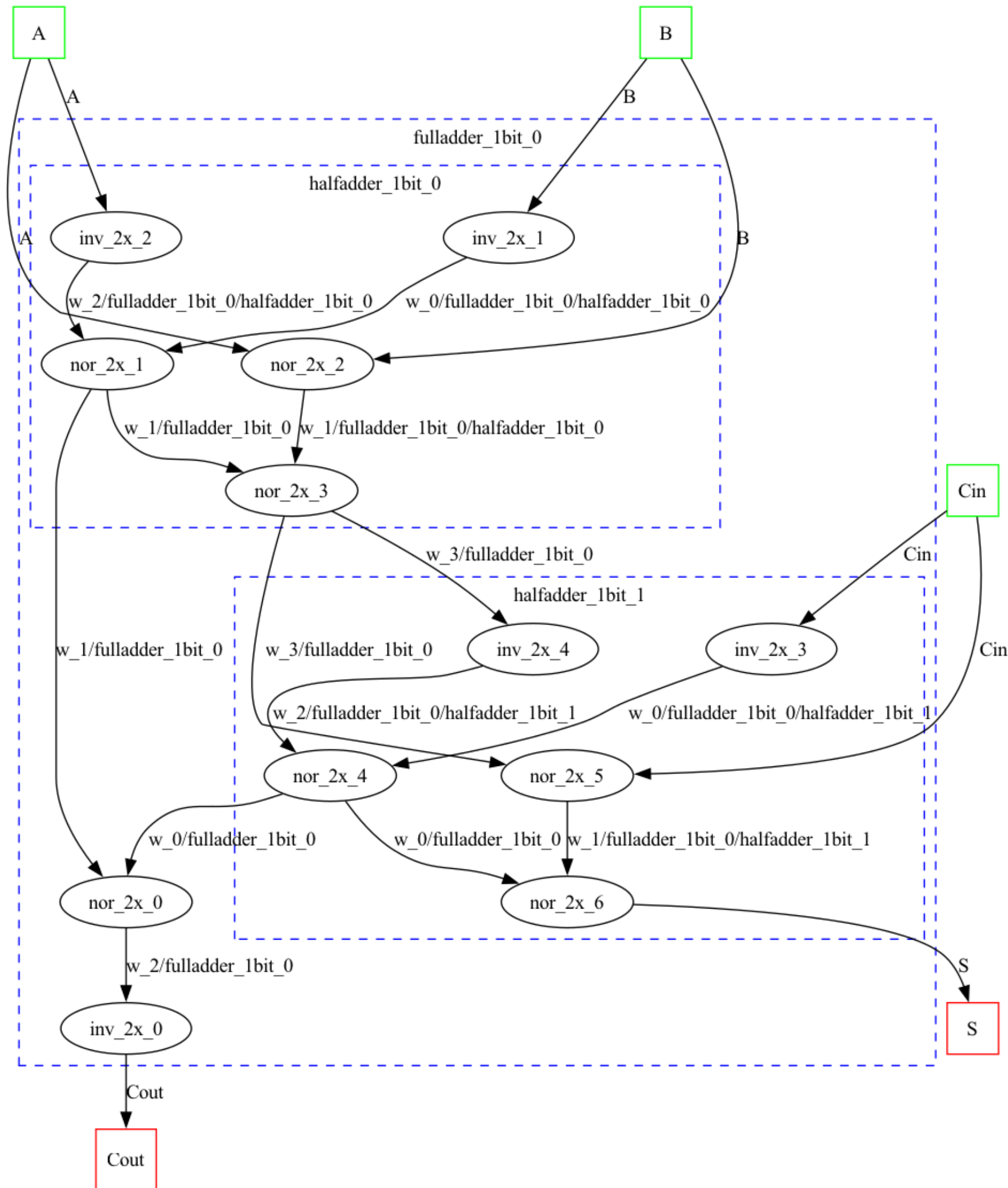
```
"pins": {
  "A": {
    "netname": "A[0]",
    "direction": "input"
  },
  "B": {
    "netname": "B[0]",
    "direction": "input"
  },
  "Cin": {
    "netname": "Cin",
    "direction": "input"
  },
  "Cout": {
    "netname": "w_0/fulladder_2bit_0",
    "direction": "output"
  },
  "S": {
    "netname": "S[0]",
    "direction": "output"
  }
},
"sub_blocks": [
  {
    "cellname": "nor_2x",
    "libname": "logic_generated",
    "name": "nor_2x_0",
    "hier_name": "fulladder_2bit_0/fulladder_1bit_0/nor_2x_0",
    "width": 1640,
    "depth": 2,
    "input": [
      "A",
      "B"
    ],
  }
]
```

Appendix II

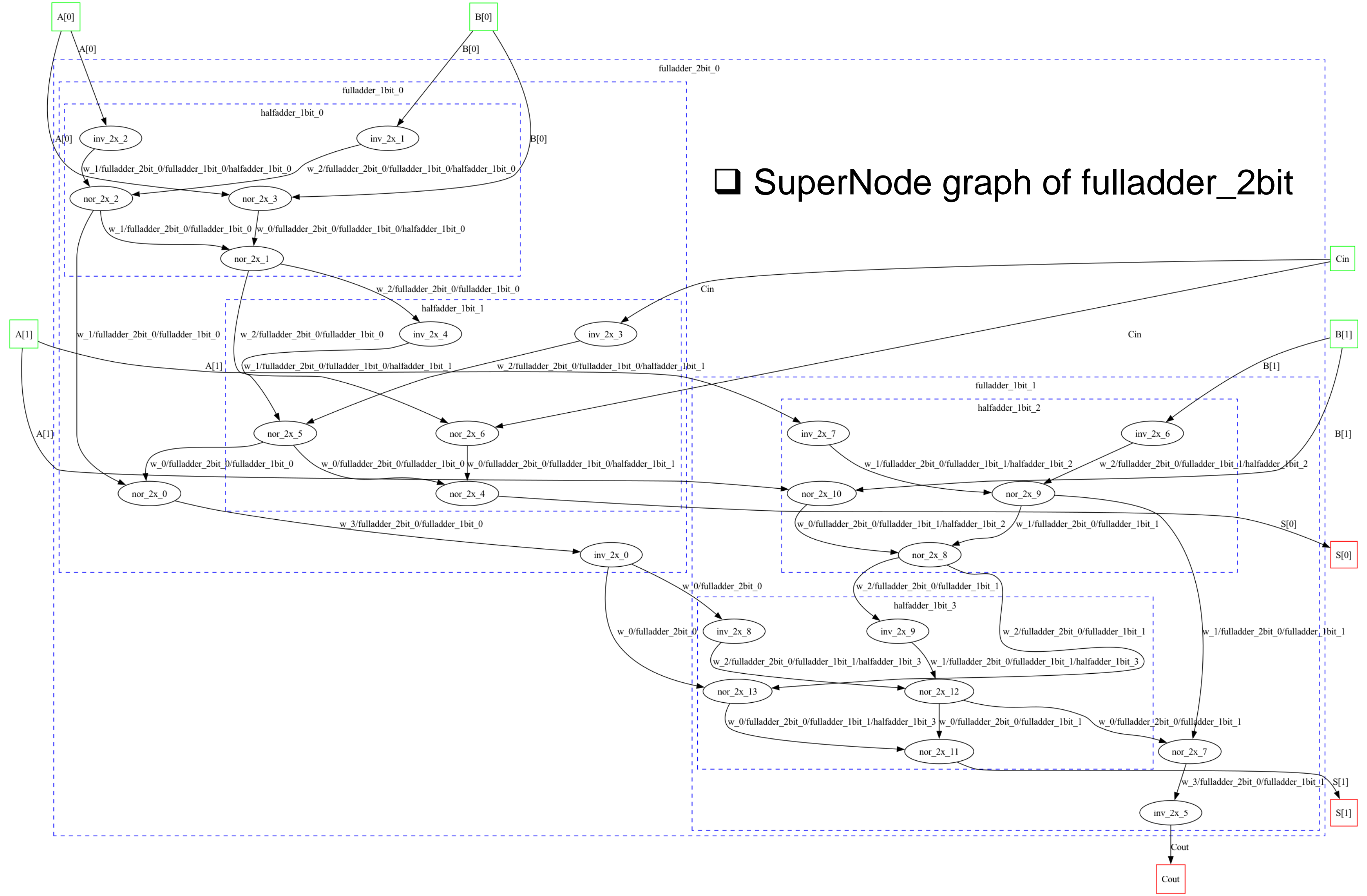
SuperNode graphs

❑ SuperNode graph of fulladder_1bit

- Stores the layout in a hierarchical graph
- Supernodes are presented in blue boxes
- Input/output presented in green/red
- Named wires for identification



❑ SuperNode graph of fulladder_2bit

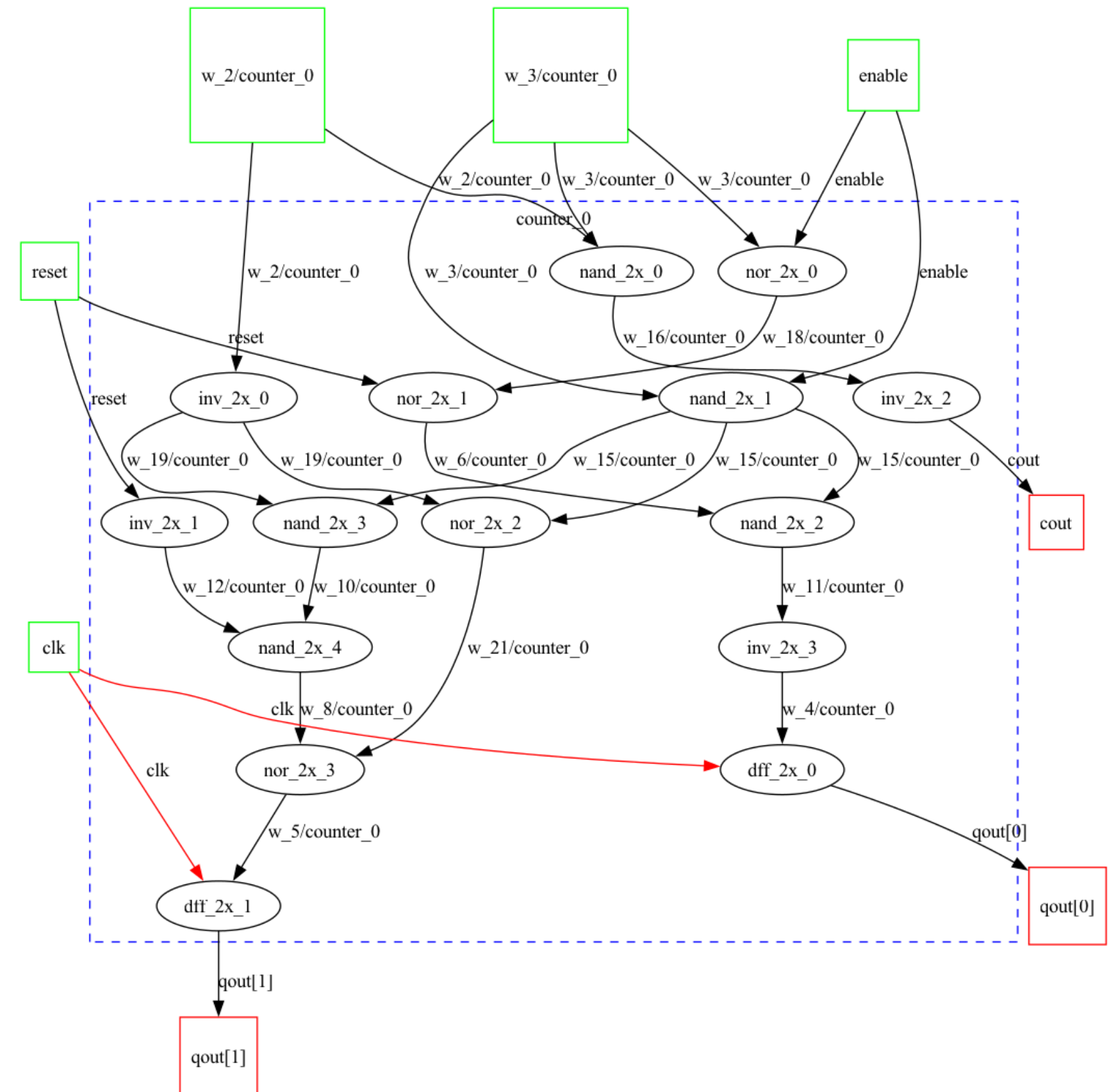


Appendix III

Critical net identification

❑ Critical net identification (represented in red)

- Identifies the critical net (“clk”) of 2bit counter
- Nodes connected to the critical net are placed in a single row



Appendix IV

Number of rows for each layout

❑ Can support multi-row placement by controlling the width and height of the box

- Width = $\text{int}(\text{supernode.width} / \text{n_rows} * \text{enlarge_factor})$
- Height = $\text{int}(\text{supernode.height} * \text{n_rows} * \text{enlarge_factor})$

```
def init_force_layout(graph, supernode, n_rows=1):  
  
    factor = 1.2  
    supernode_width = supernode.width  
    supernode_height = int((2000 * n_rows) * factor)  
    nodes = []  
    edges = []  
    fixed_nodes = []  
    for sub_node in supernode.sub_nodes:  
        width = sub_node.width  
        height = 2000  
        node = ForceNode(sub_node.name, width, height)  
        nodes.append(node)  
    for edge in graph.edge_list:  
        in_nodes = []  
        out_nodes = []
```


Appendix V

Time Complexity

- ❑ SuperNode Graph generation → $O(n^2)$
- ❑ Force-Directed Graph Placement → $O((\log n)^3)$
 - Avg number of supernodes = $O(\log n)$
 - Avg number of sub_nodes in supernode = $O(\log n)$
 - Calculation forces of each node = $O(n^2)$
 - n = number of nodes for Force-Directed placement