Resource Sharing in Stratix IV

Stefan Hadjis

August 25th 2011

Binding in High Level Synthesis

- Binding reduces circuit area by sharing functional units among multiple inputs
- Intuitively it makes sense to share larger operations or chains of smaller operations

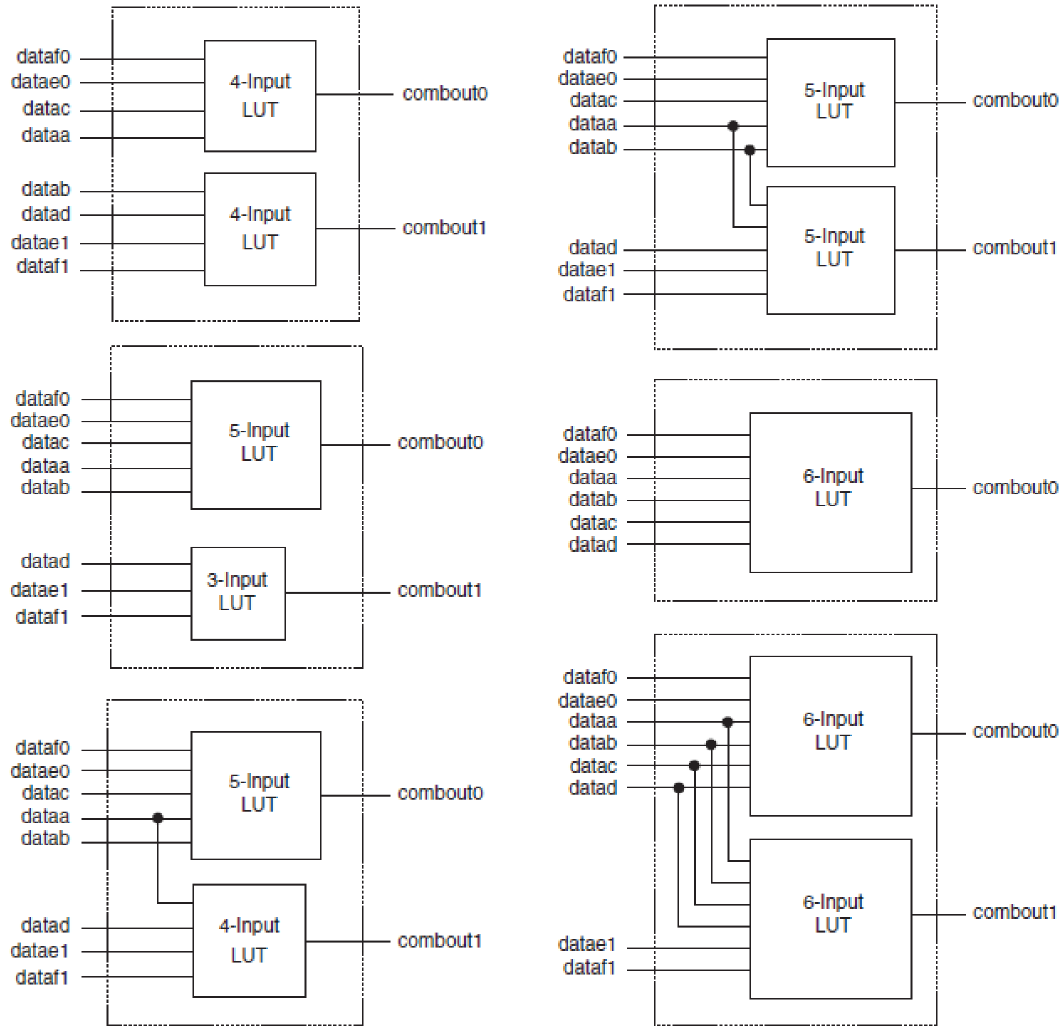
Binding in High Level Synthesis

- Binding reduces circuit area by sharing functional units among multiple inputs
- Intuitively it makes sense to share larger operations or chains of smaller operations

<u>Original LegUp Release</u>

- In Cyclone II Architecture, sharing is beneficial for **dividers** and **remainders** (mod), which are large operations
- E.g. if a circuit needs to perform division 10 times, it is cheapest to have one divider with 10-to-1 muxing on its inputs

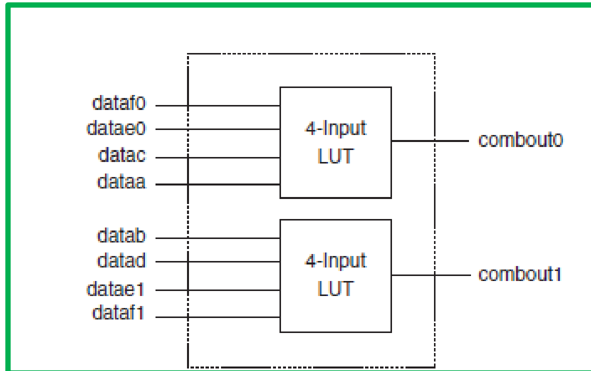
Stratix IV, Adaptive Logic Modules (ALM)

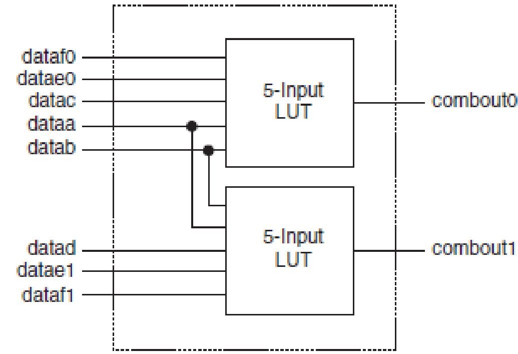


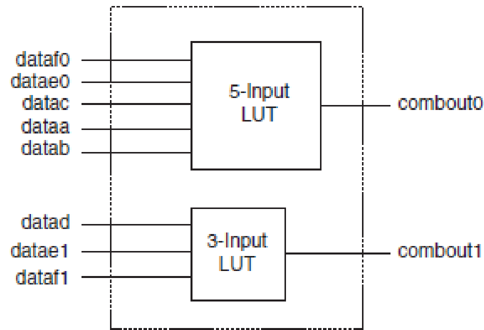
Each ALM contains 2 Adaptive LUTs (ALUTs) which can implement a function of between 4 and 7 inputs

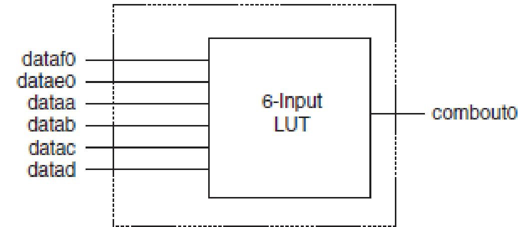
Stratix IV, Adaptive Logic Modules (ALM)

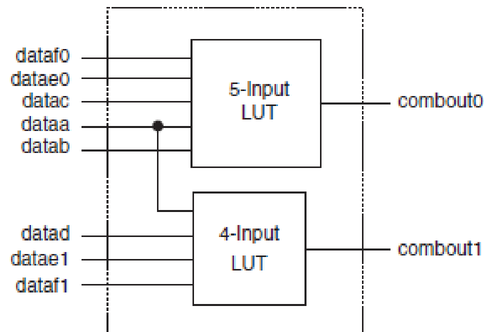
Cyclone II

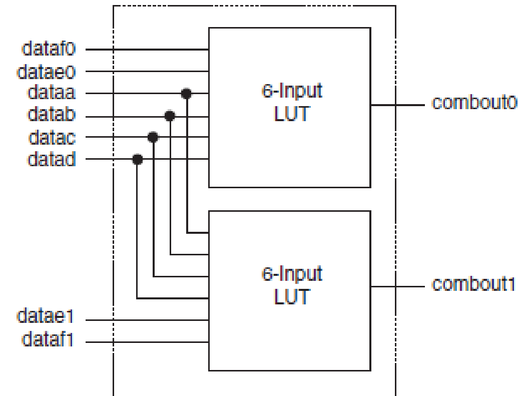












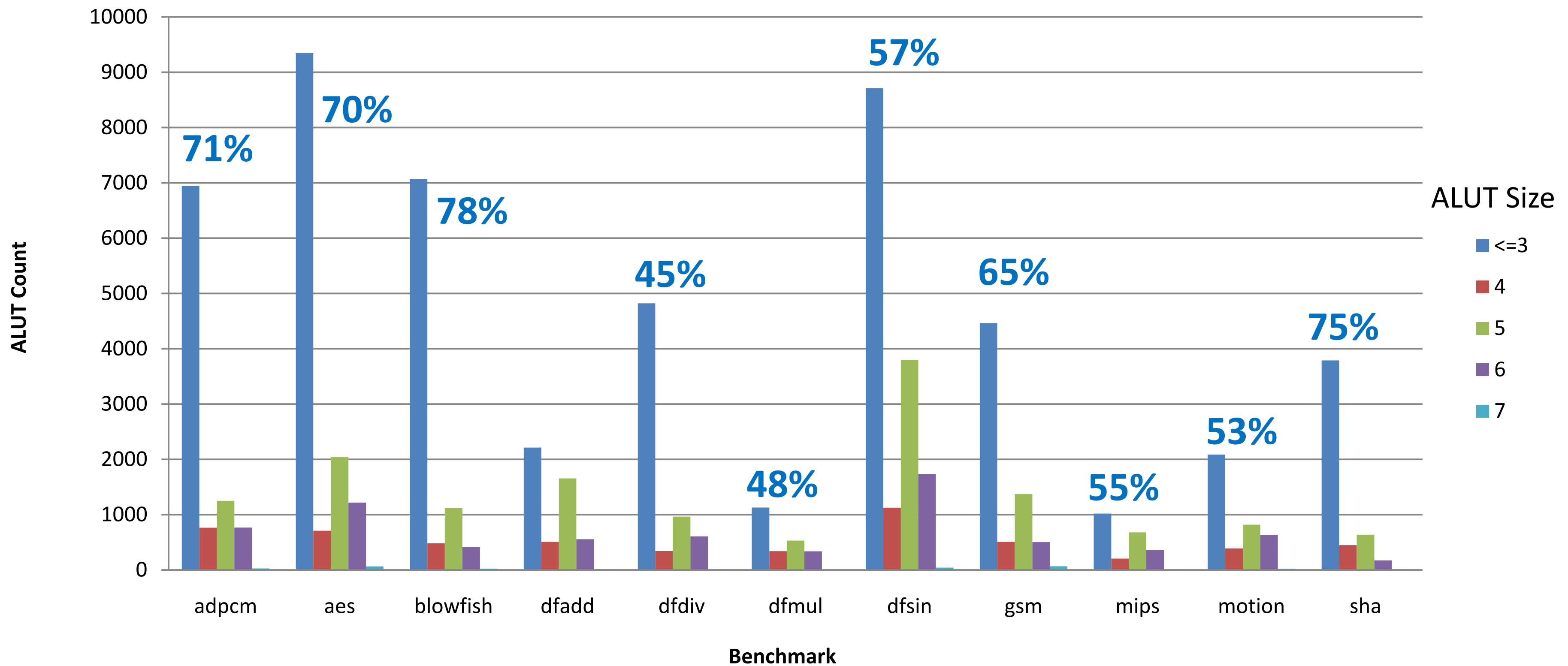
Each ALM contains 2 Adaptive LUTs (ALUTs) which can implement a function of between 4 and 7 inputs

Binding in Stratix IV

- All of the circuits created by LegUp tend to use mostly 2 and 3 input functions (LUTs)

Binding in Stratix IV

- All of the circuits created by LegUp tend to use mostly 2 and 3 input functions (LUTs)



Average: 62%

Binding in Stratix IV

- All of the circuits created by LegUp tend to use mostly 2 and 3 input functions (LUTs)
- For example, a bitwise AND of two 32-bit numbers requires 32 ALUTs, but each is only size 2 (2 input bits, 1 output bit)

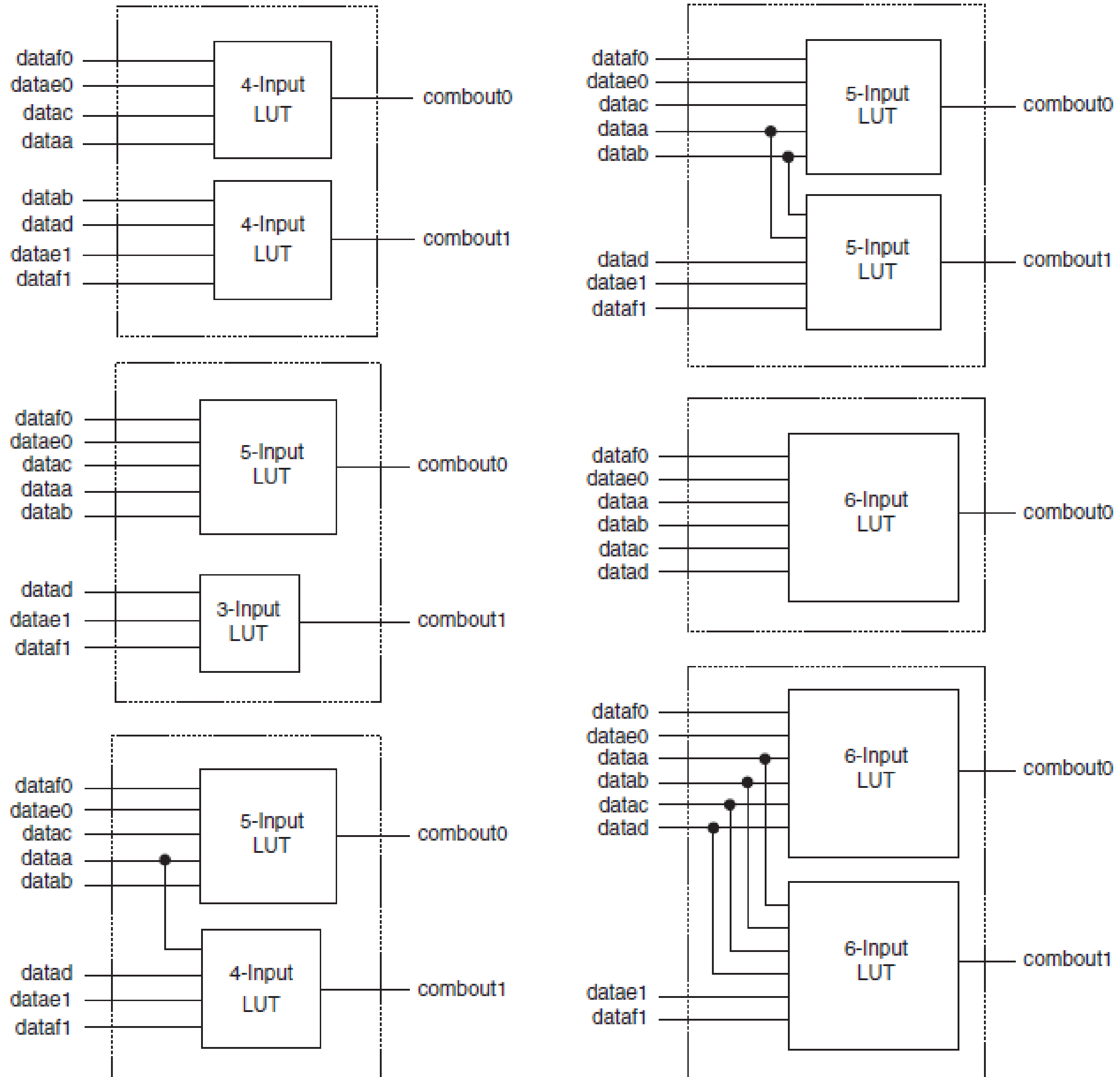
ALM Example

- Consider two circuits:

Circuit 1: Implemented using 100 **3-input LUTs**

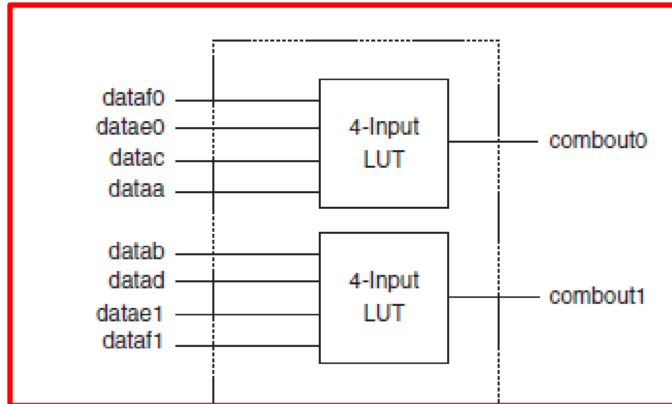
Circuit 2: Implemented using 45 **3-input LUTs** and 45 **5-input LUTs**

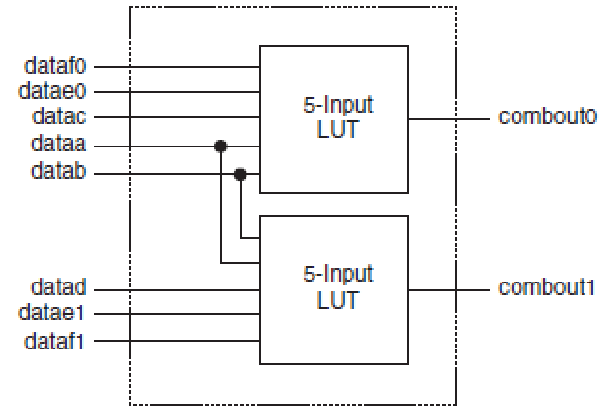
Adaptive Logic Modules (ALM)

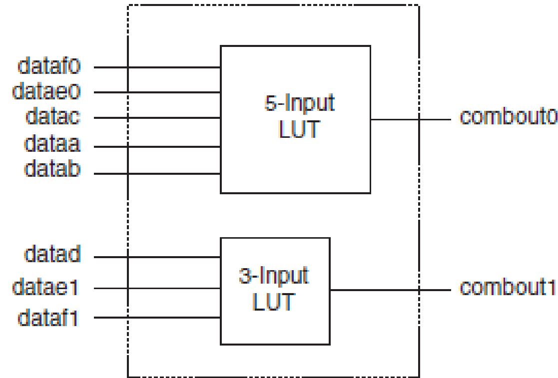


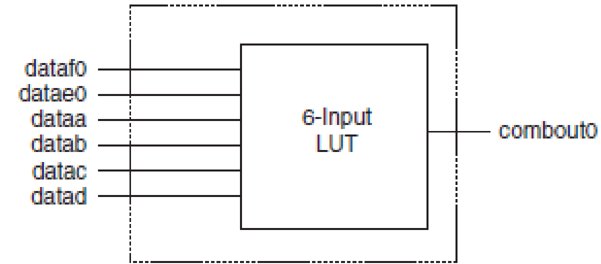
Adaptive Logic Modules (ALM)

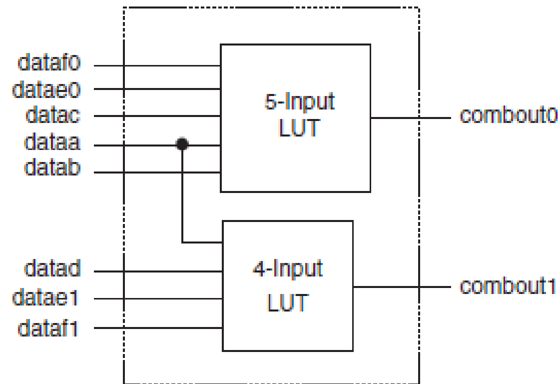
50

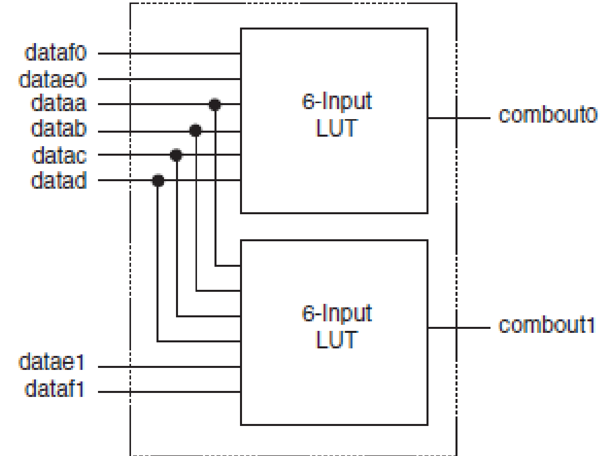






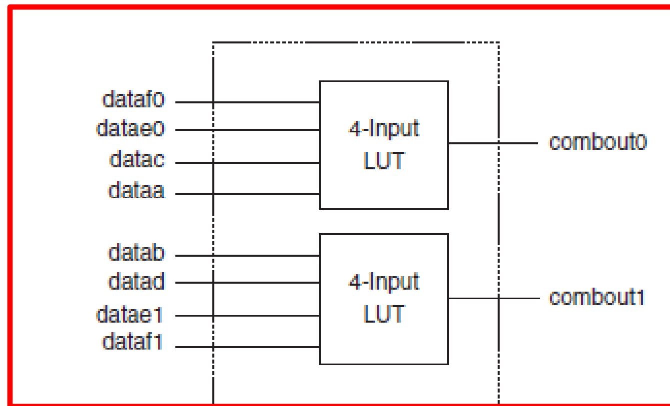




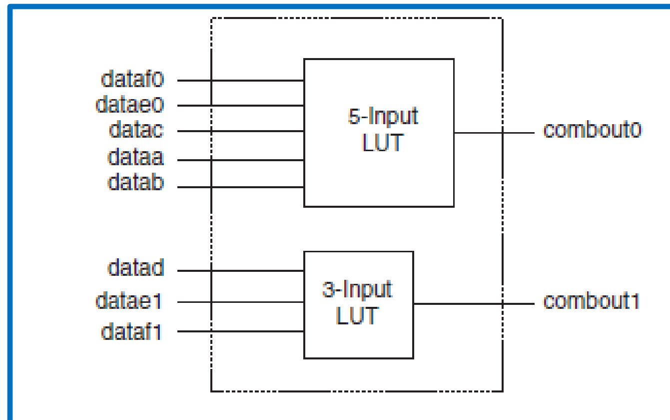


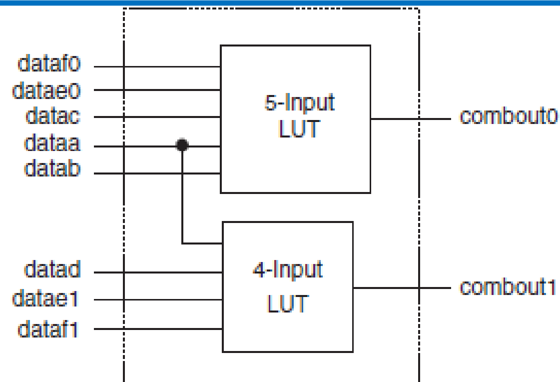
Adaptive Logic Modules (ALM)

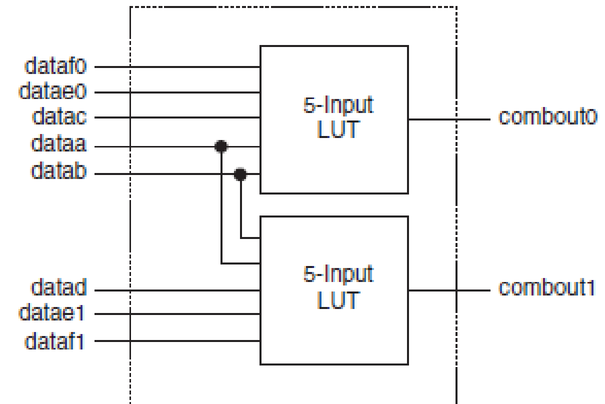
50

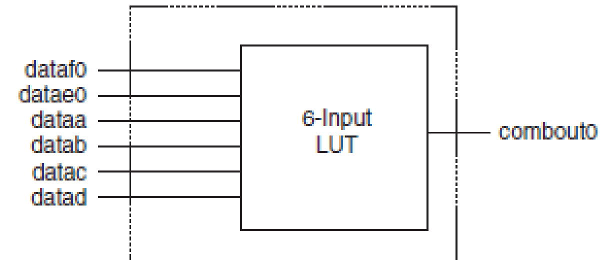


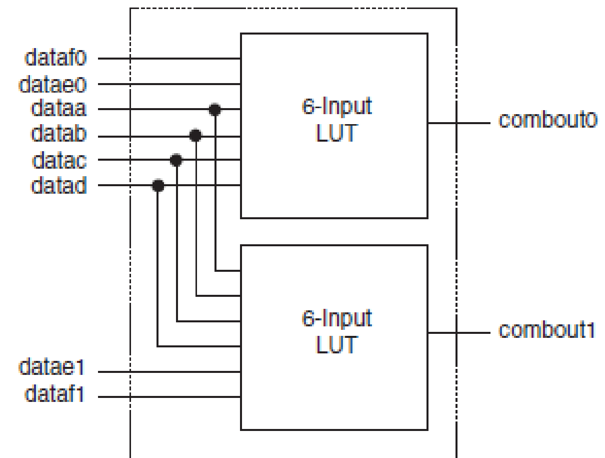
45











ALM Example

- Consider two circuits:

Circuit 1: Implemented using 100 **3-input LUTs**

Circuit 2: Implemented using 45 **3-input LUTs** and 45 **5-input LUTs**

ALM Example

- Consider two circuits:

Circuit 1: Implemented using 100 **3-input LUTs**

→ Requires **50** ALMs

Circuit 2: Implemented using 45 **3-input LUTs** and 45 **5-input LUTs**

ALM Example

- Consider two circuits:

Circuit 1: Implemented using 100 **3-input LUTs**

→ Requires **50** ALMs

Circuit 2: Implemented using 45 **3-input LUTs** and 45 **5-input LUTs**

→ Requires **45** ALMs, even though the circuit contains more logic

Sharing Single Operations

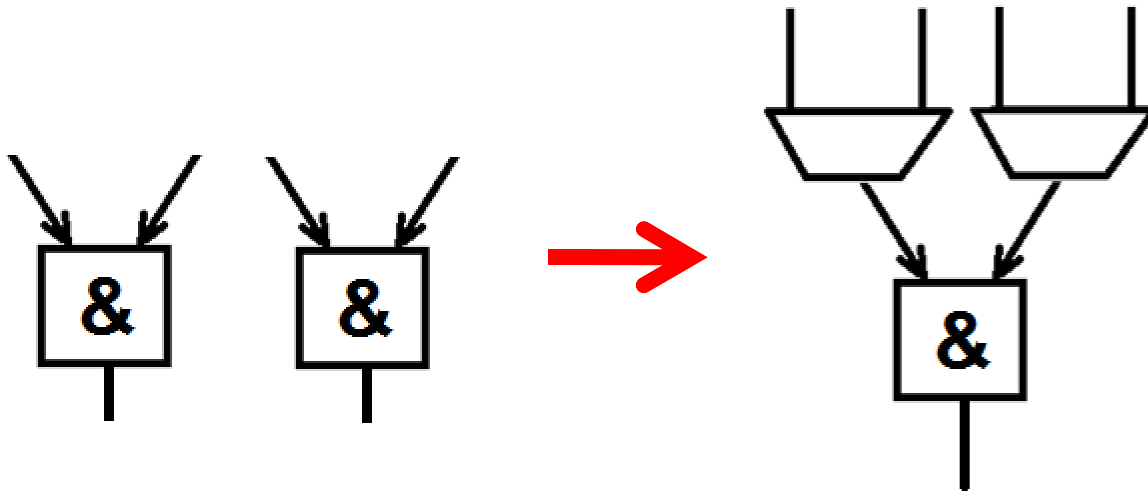
- Given that LegUp-generated circuits contain mostly 2-3 input functions therefore, the number of ALMs can be reduced by packing many “smaller LUTs” into fewer “larger LUTs”

Example – Sharing Bitwise Operations

- A 1-bit **2-to-1 multiplexor** requires a 3-input LUT (2 inputs and select), whereas a 1-bit **Bitwise AND** requires only a 2-input LUT

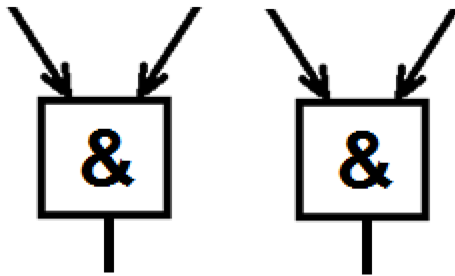
Example – Sharing Bitwise Operations

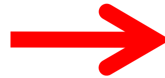
- A 1-bit **2-to-1 multiplexor** requires a 3-input LUT (2 inputs and select), whereas a 1-bit **Bitwise AND** requires only a 2-input LUT
- Intuitively therefore, it does not make sense to do this:

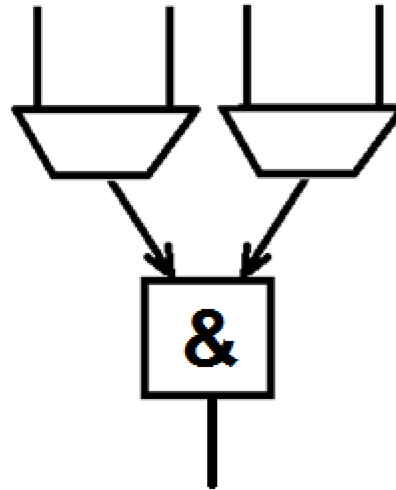


Example – Sharing Bitwise Operations

- A 1-bit **2-to-1 multiplexor** requires a 3-input LUT (2 inputs and select), whereas a 1-bit **Bitwise AND** requires only a 2-input LUT
- Intuitively therefore, it does not make sense to do this:







Two 2-input LUTs

One 5-input LUT

Example – Sharing Bitwise Operations

- A 1-bit **2-to-1 multiplexor** requires a 3-input LUT (2 inputs and select), whereas a 1-bit **Bitwise AND** requires only a 2-input LUT
- However, sharing a bitwise operation uses half the LUTs – one 5-input LUT versus two 2-input LUTs
- If the circuits therefore contain many more small LUTs than large LUTs, sharing bitwise operations reduces ALMs

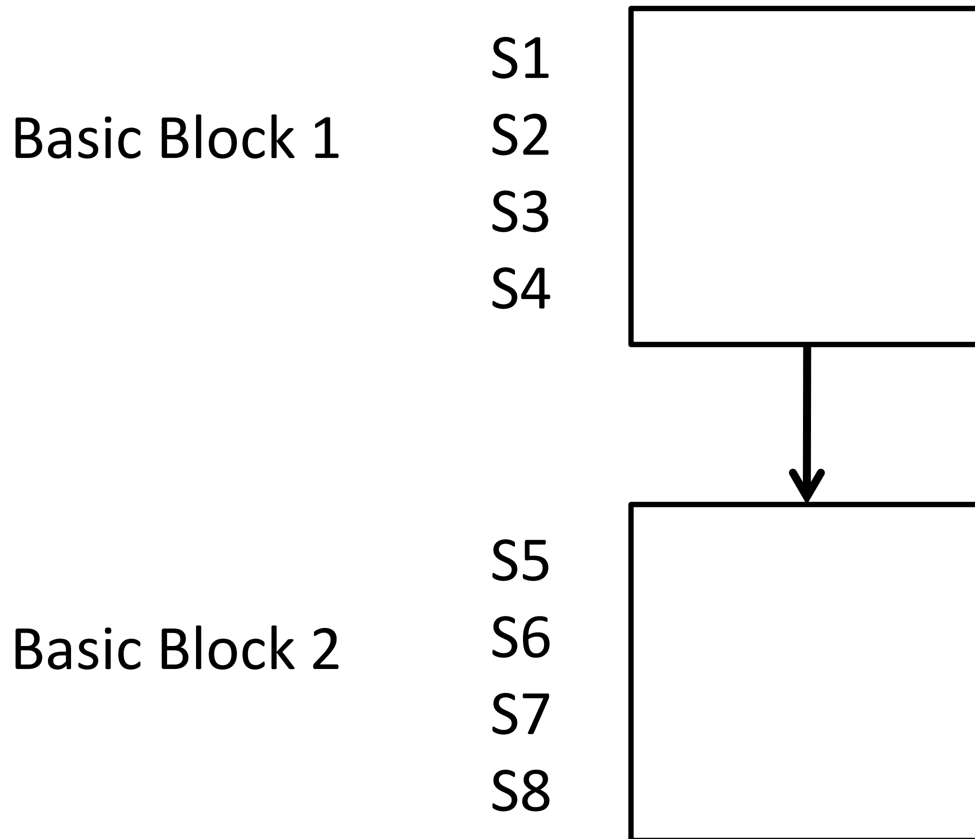
Example – Sharing Bitwise Operations

- A 1-bit **2-to-1 multiplexor** requires a 3-input LUT (2 inputs and select), whereas a 1-bit **Bitwise AND** requires only a 2-input LUT
- However, sharing a bitwise operation uses <u>half the LUTs</u> – one 5-input LUT versus two 2-input LUTs
- If the circuits therefore contain many more small LUTs than large LUTs, sharing bitwise operations reduces ALMs
- Therefore, sharing even small operations reduces ALUT and ALM usage

Live Variable Analysis

- Next, if the output of each shared operation was originally connected to a register, then using **Live Variable Analysis,** operations whose live intervals do not overlap can be shared to save registers

Live Variable Analysis



Live Variable Analysis

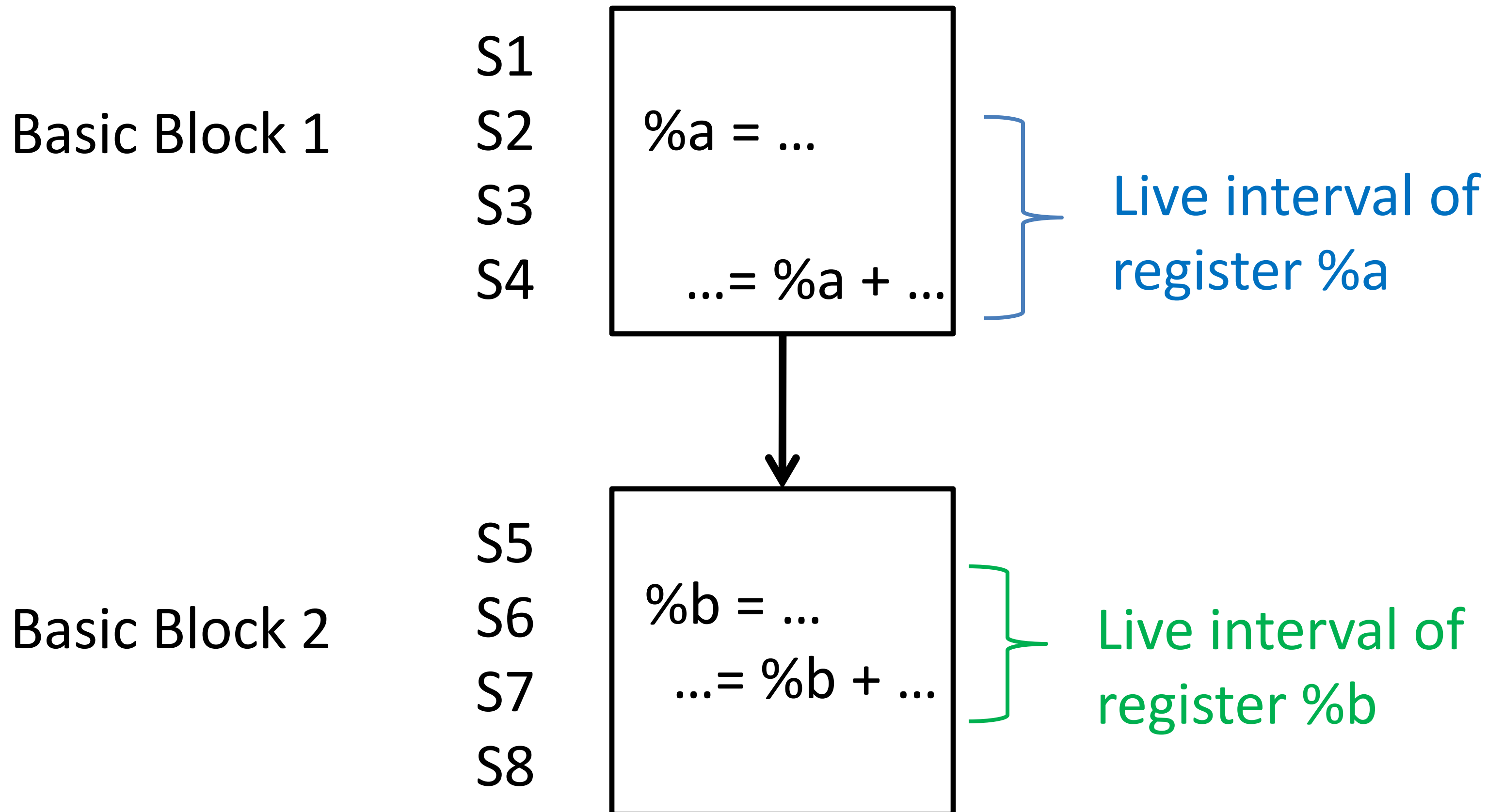
Basic Block 1

S1
S2 %a = …
S3
S4 …= %a + …

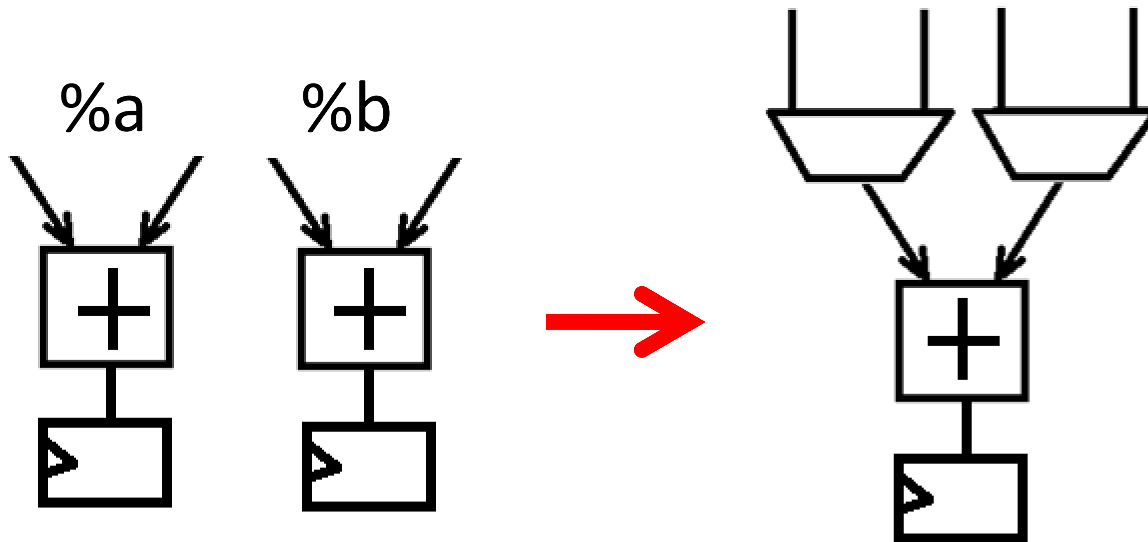
Basic Block 2

S5
S6 %b = …
S7 …= %b + …
S8

Live Variable Analysis



Live Variable Analysis



E.g. for a 32-bit add, save 32 registers

Live Variable Analysis

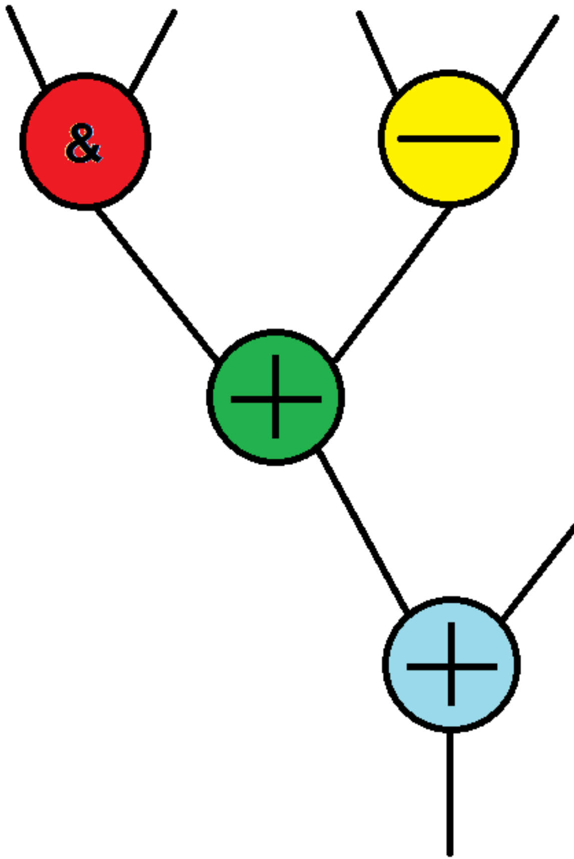
- Created an LLVM pass which calculates post-scheduling liveness information for each register
- Used to map every instruction to a set of other instructions with which it is independent
- This represents a graph, where nodes are instructions and edges represent independence
- Common instructions (e.g. all adders) are then paired and pairs are shared to save registers

Altera Resource Sharing

- Quartus II has a Resource Sharing synthesis option
- Shares adders, subtractors and multipliers
- However, none of the circuits tested were affected by this option

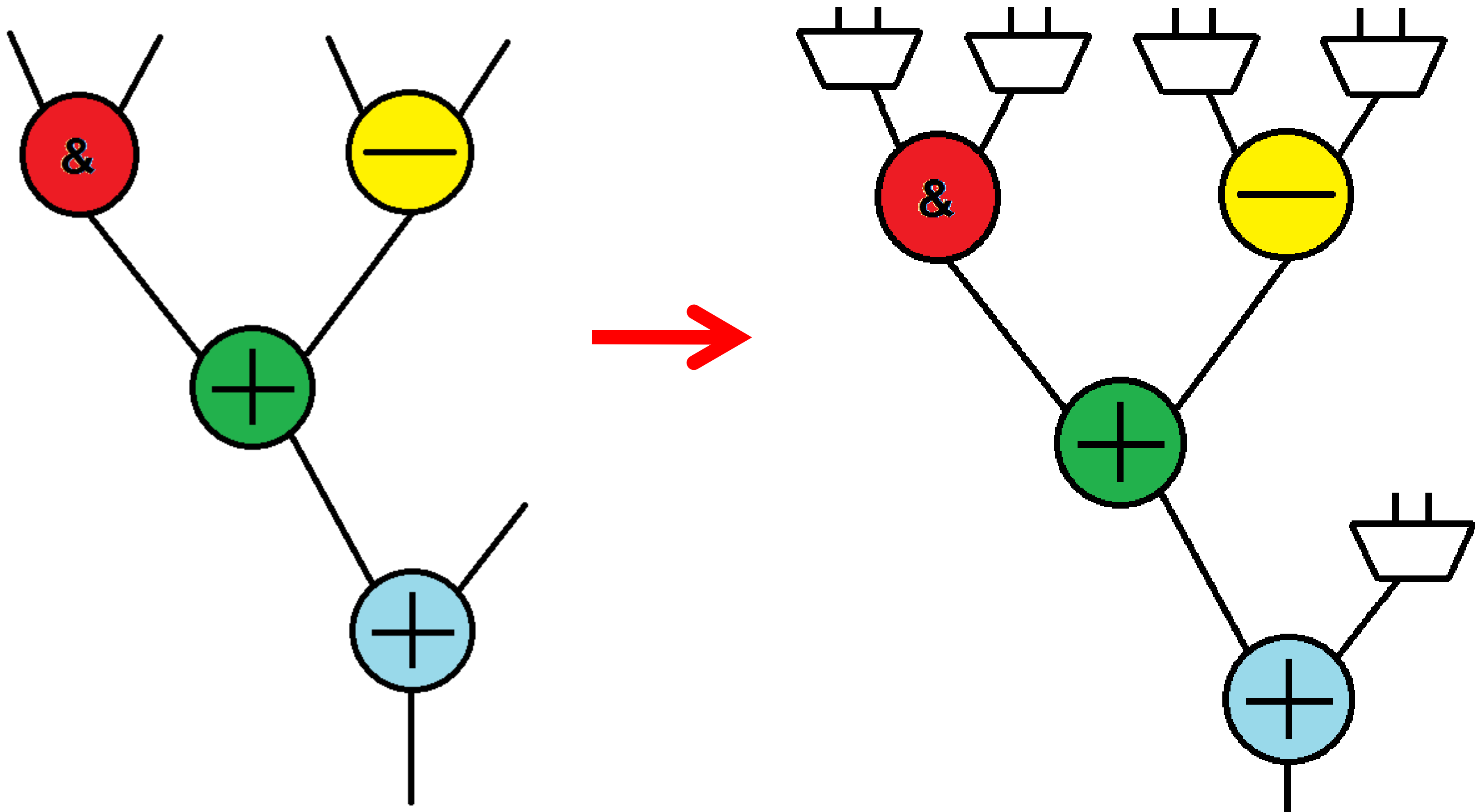
Sharing Chains of Operations

- So far, ALUTs and Registers were saved by sharing single operations
- Consider:



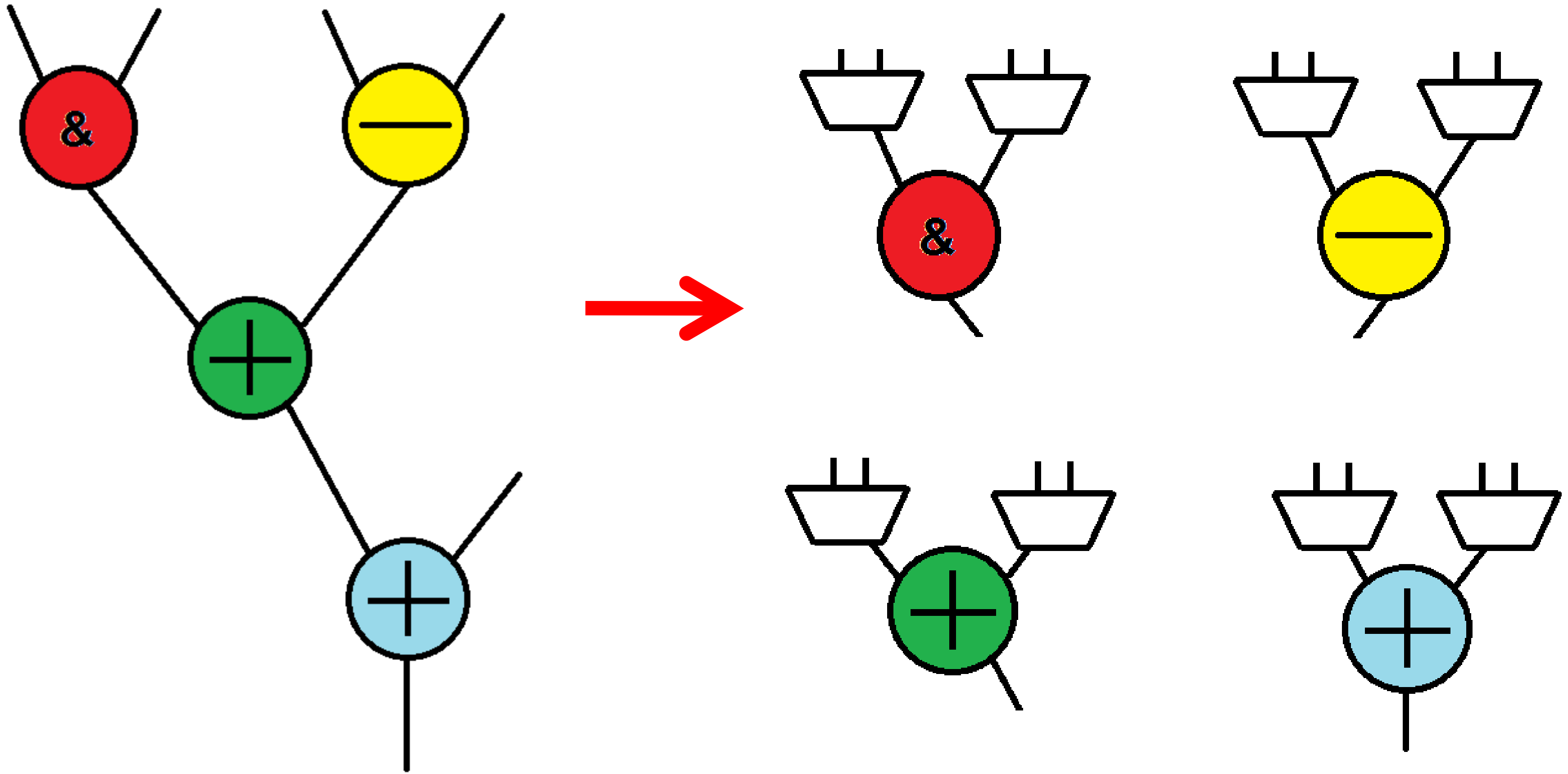
Sharing Chains of Operations

- So far, ALUTs and Registers were saved by sharing single operations
- Consider:



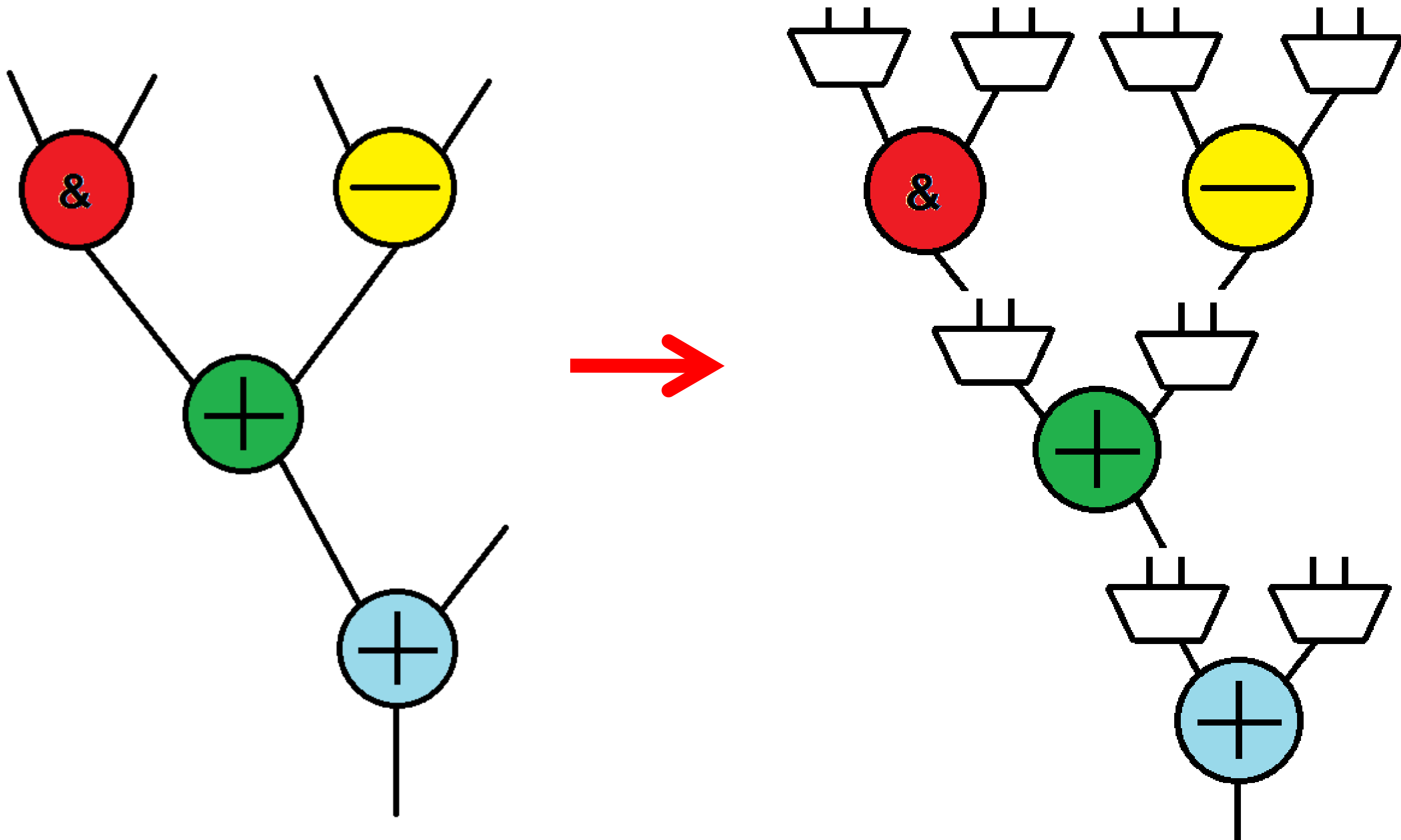
Sharing Chains of Operations

- So far, ALUTs and Registers were saved by sharing single operations
- Consider:



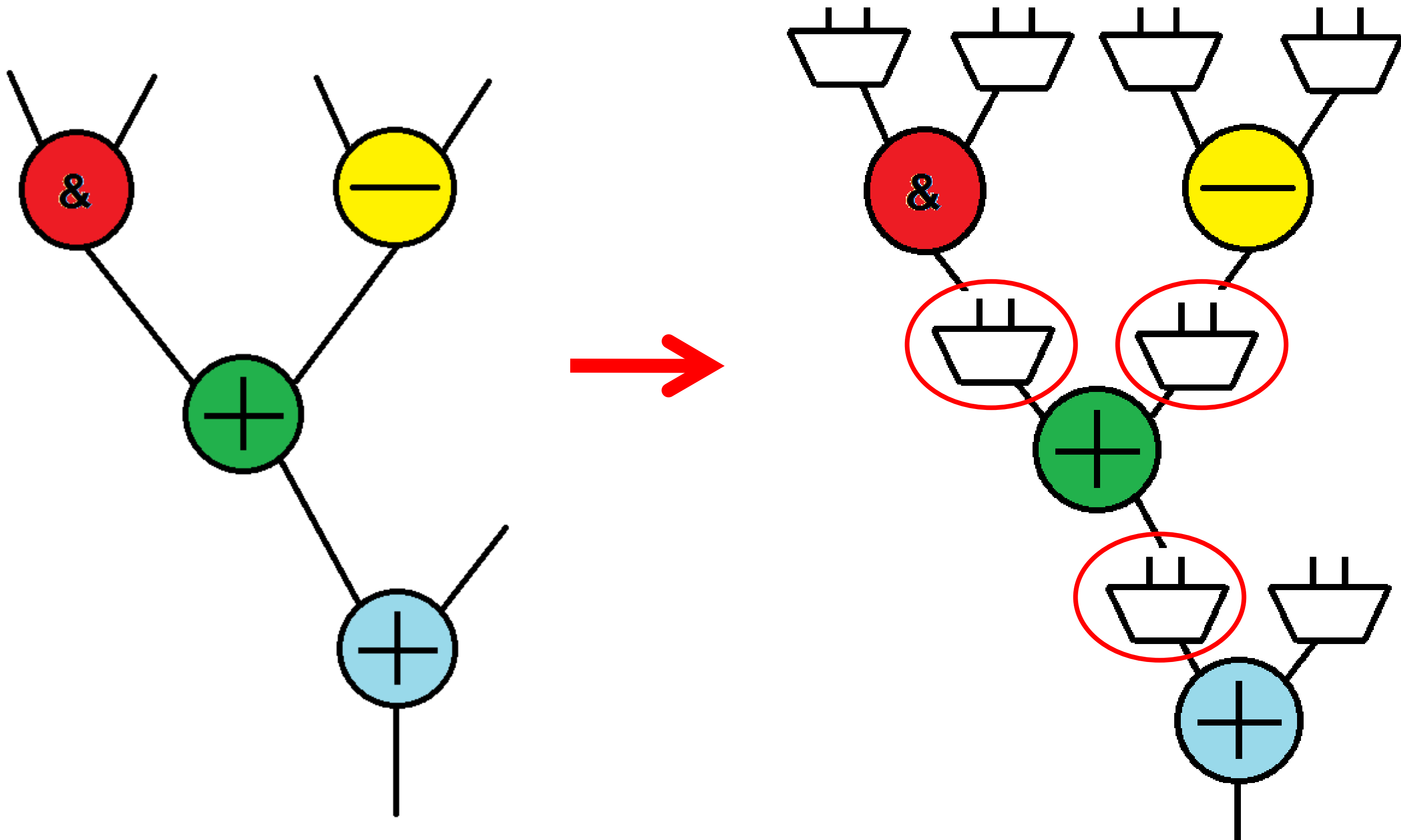
Sharing Chains of Operations

- So far, ALUTs and Registers were saved by sharing single operations
- Consider:



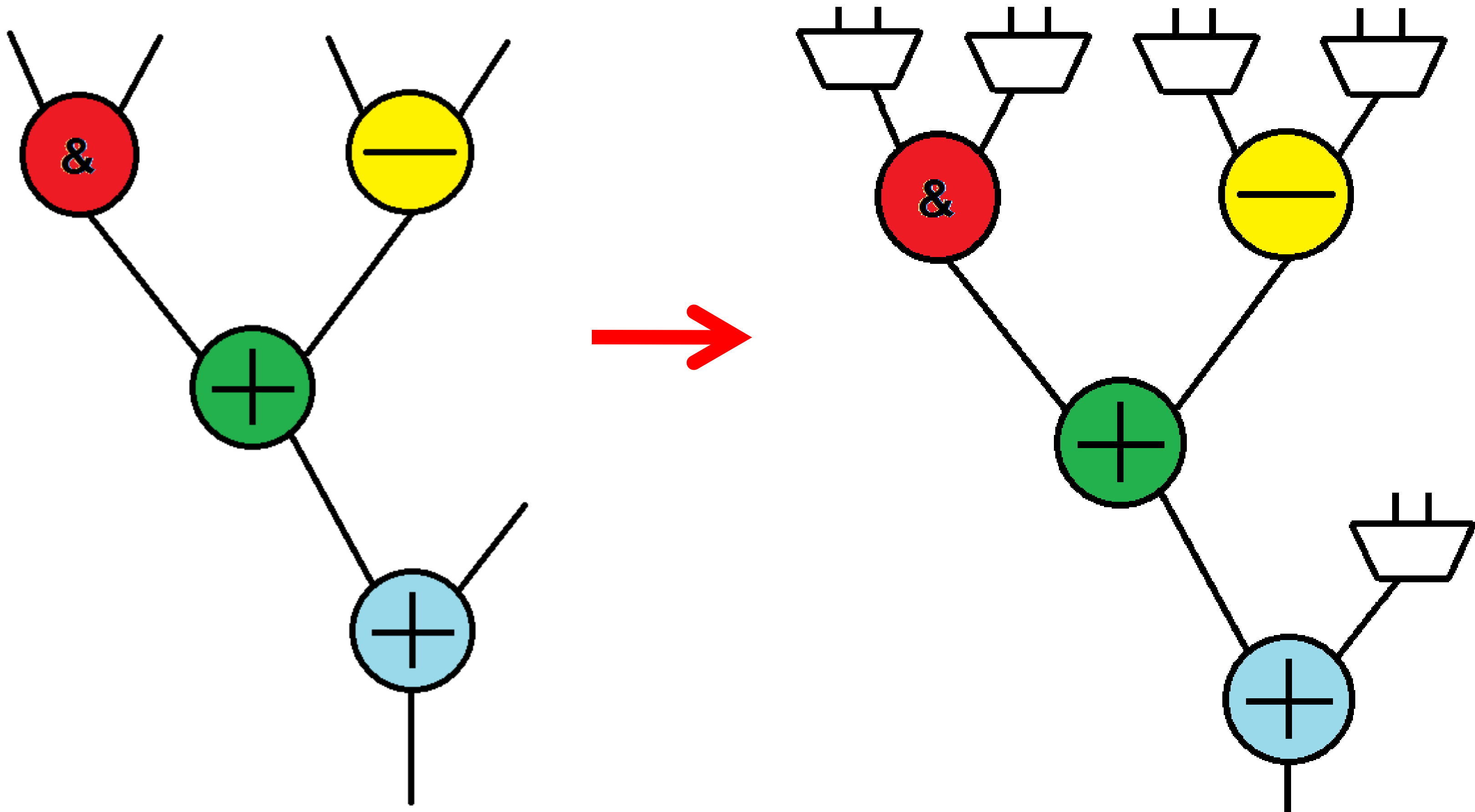
Sharing Chains of Operations

- So far, ALUTs and Registers were saved by sharing single operations
- Consider:



Sharing Chains of Operations

- So far, ALUTs and Registers were saved by sharing single operations
- Consider:



Sharing Chains of Operations

- So far, ALUTs and Registers were saved by sharing single operations
- By sharing chains instead of only single operations, the amount of multiplexing is reduced and ALUTs decrease further

Sharing Chains of Operations

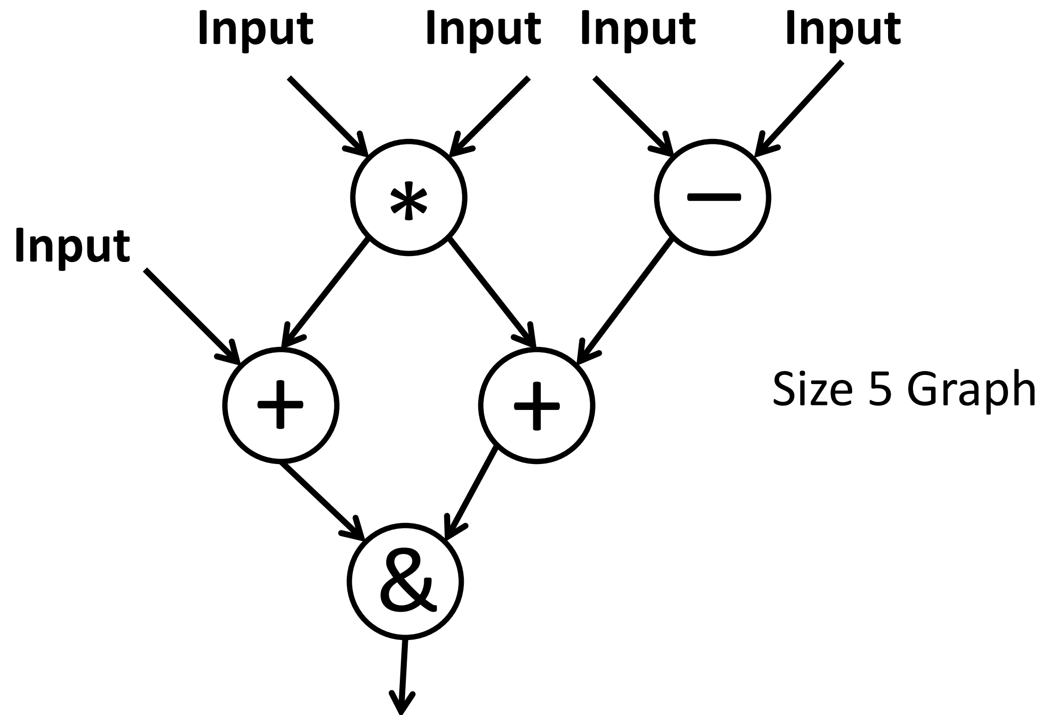
- So far, ALUTs and Registers were saved by sharing single operations
- By sharing chains instead of only single operations, the amount of multiplexing is reduced and ALUTs decrease further
- Intuitively, as larger patterns are considered, their frequency of occurrence will decrease

Sharing Chains of Operations

- So far, ALUTs and Registers were saved by sharing single operations
- By sharing chains instead of only single operations, the amount of multiplexing is reduced and ALUTs decrease further
- Intuitively, as larger patterns are considered, their frequency of occurrence will decrease
- Therefore, an iterative algorithm is used to find patterns of all sizes

Sharing Chains of Operations

- Computational patterns are represented as **Directed Graphs**, with a single output ("root") node:



- Graph traversal algorithms are then used to compare graphs and group together similar patterns for sharing

Sharing Algorithm

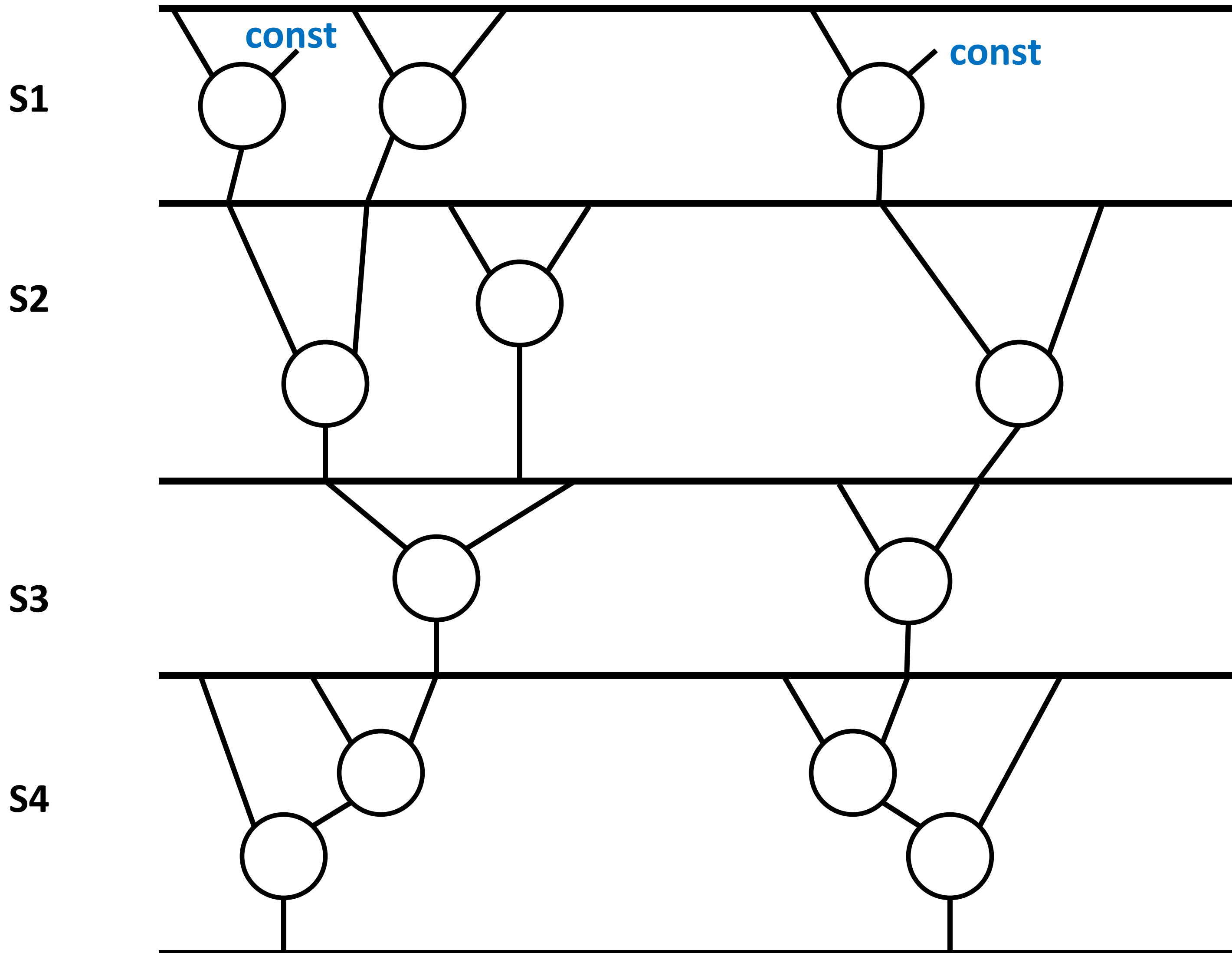
S1

S2

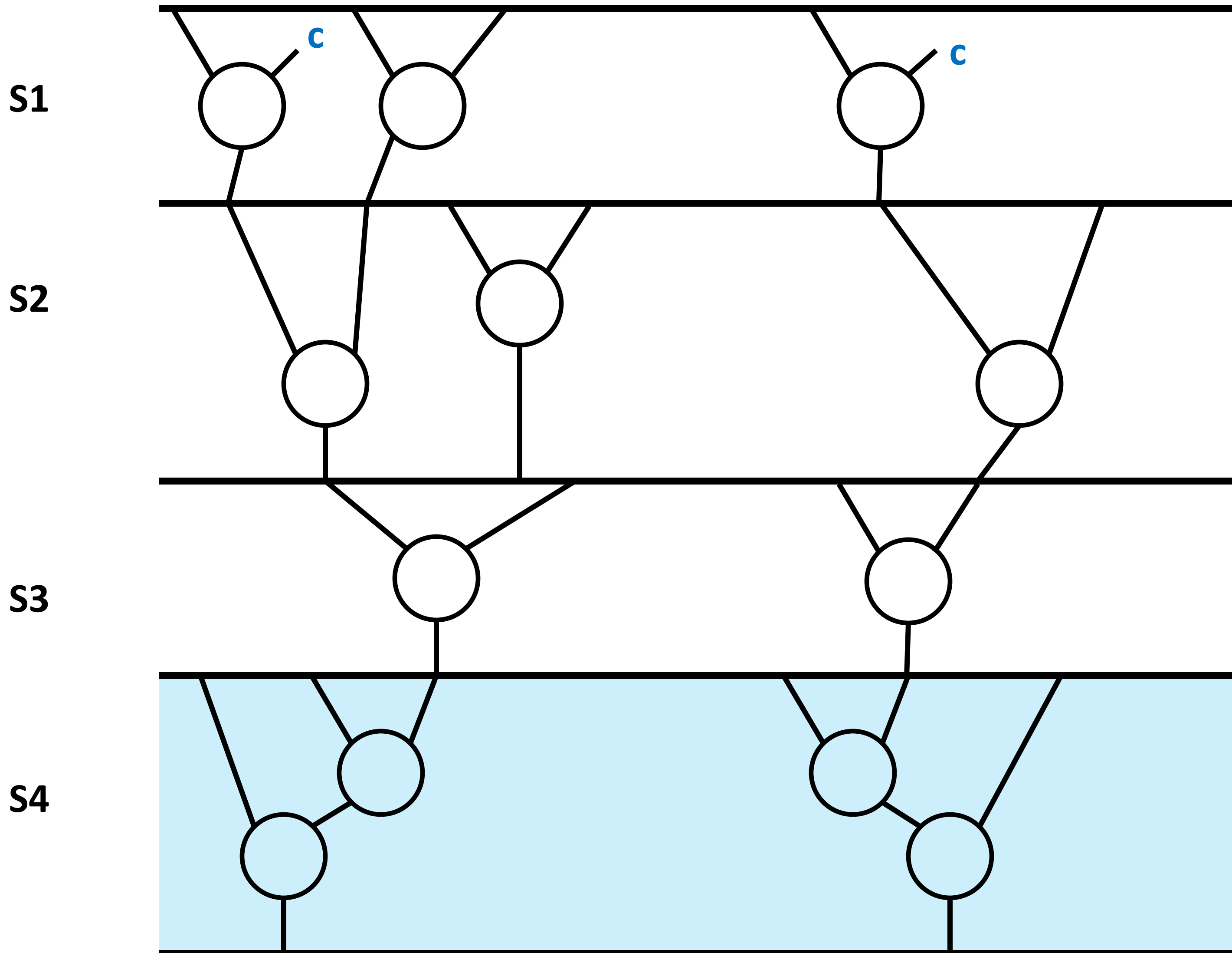
S3

S4

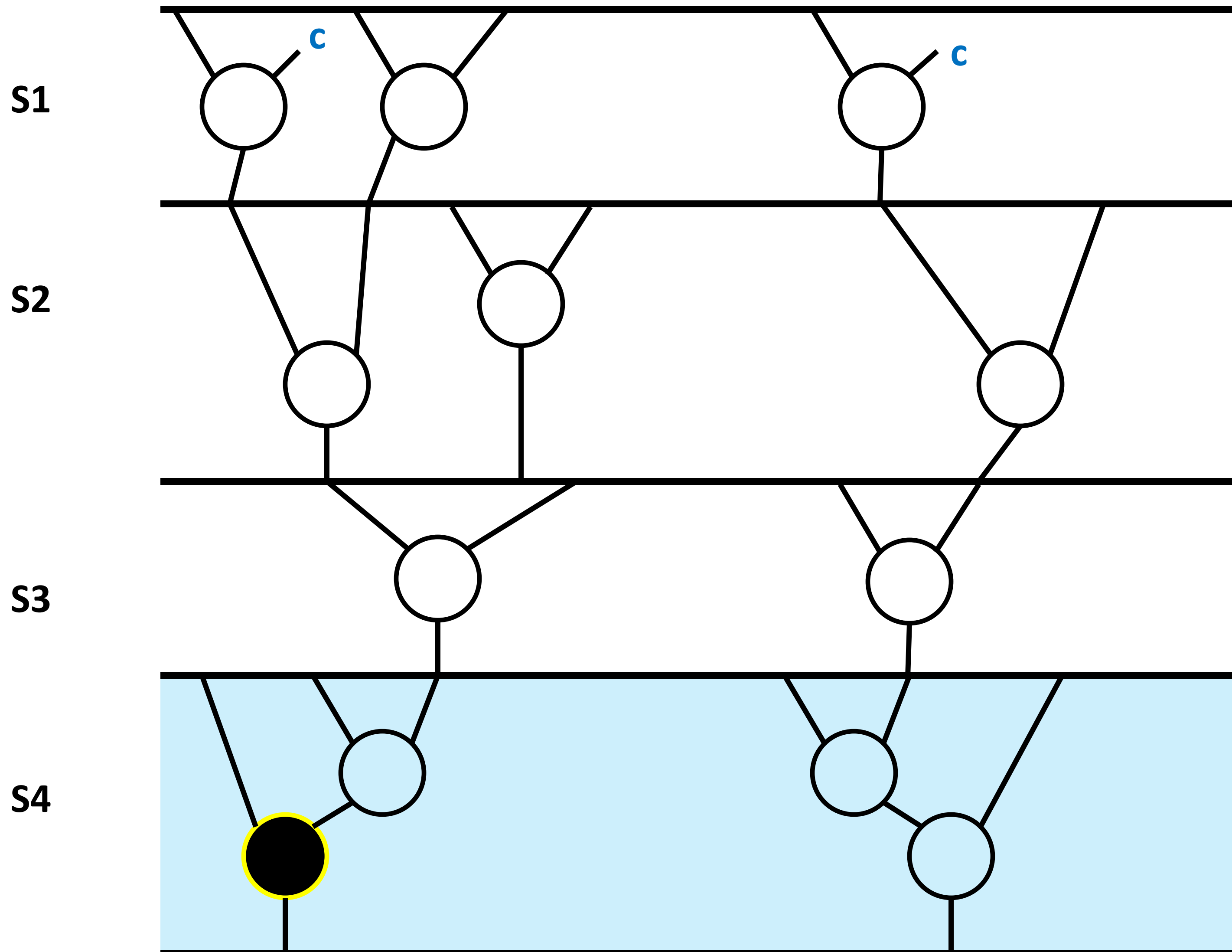
Sharing Algorithm



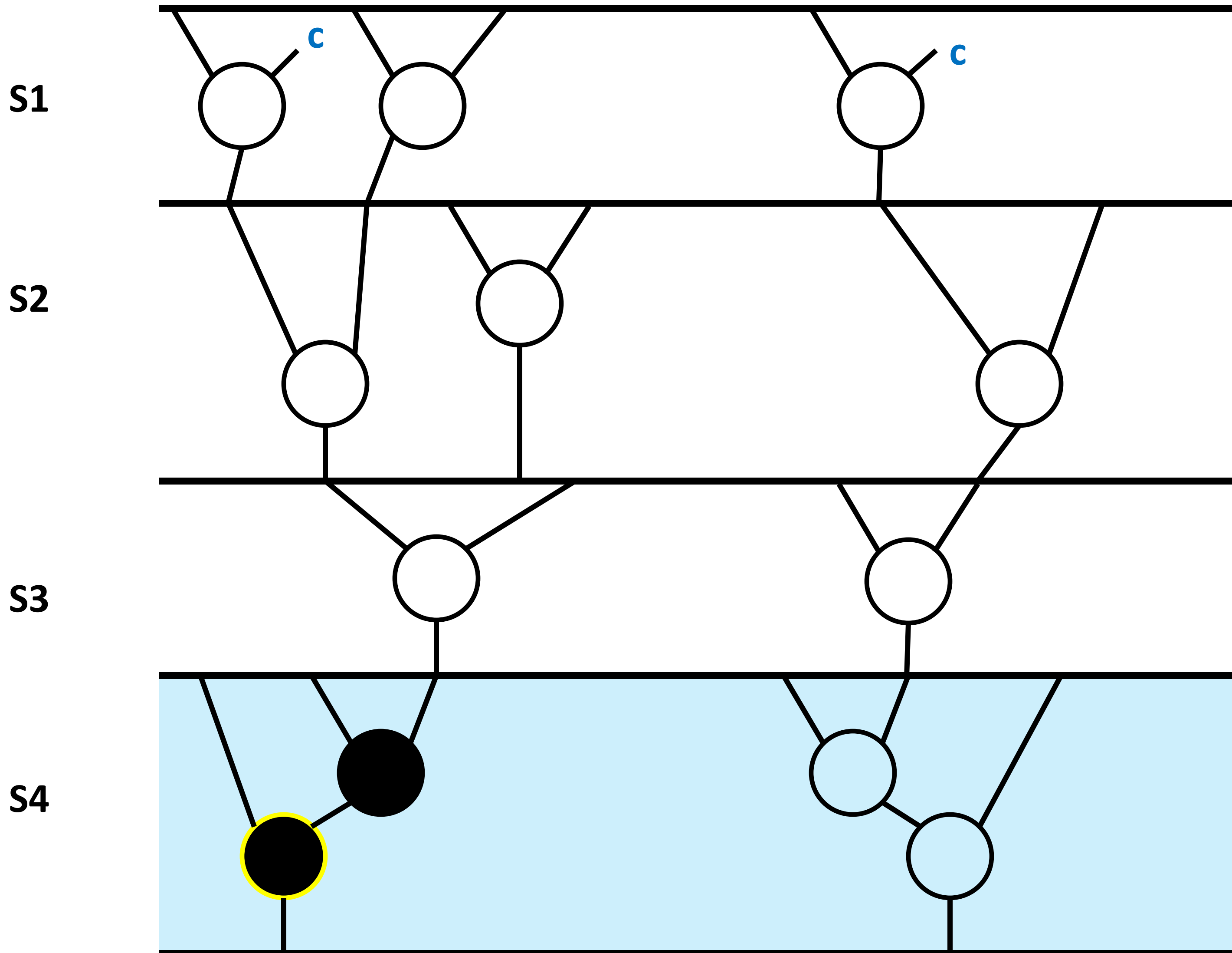
Sharing Algorithm



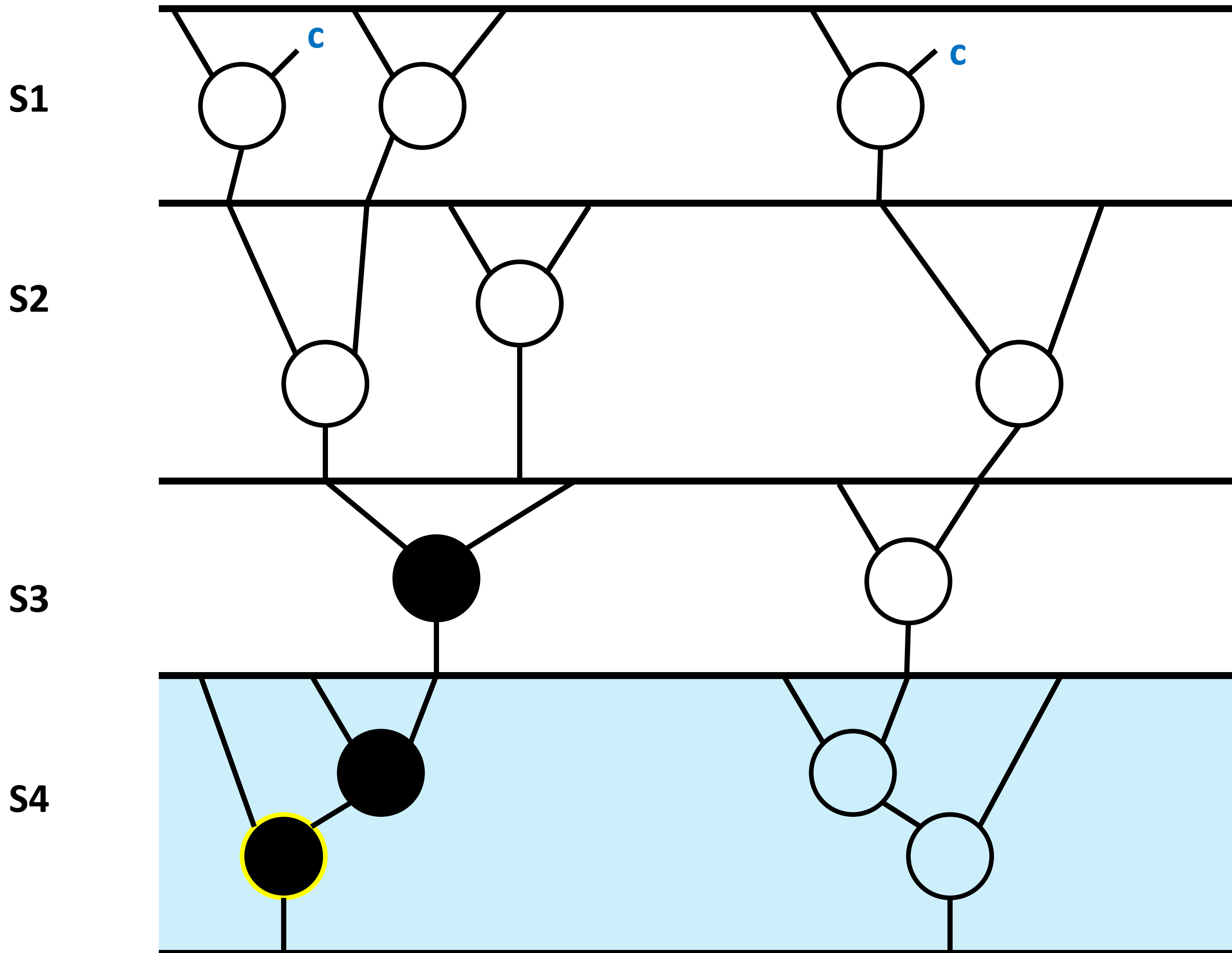
Sharing Algorithm



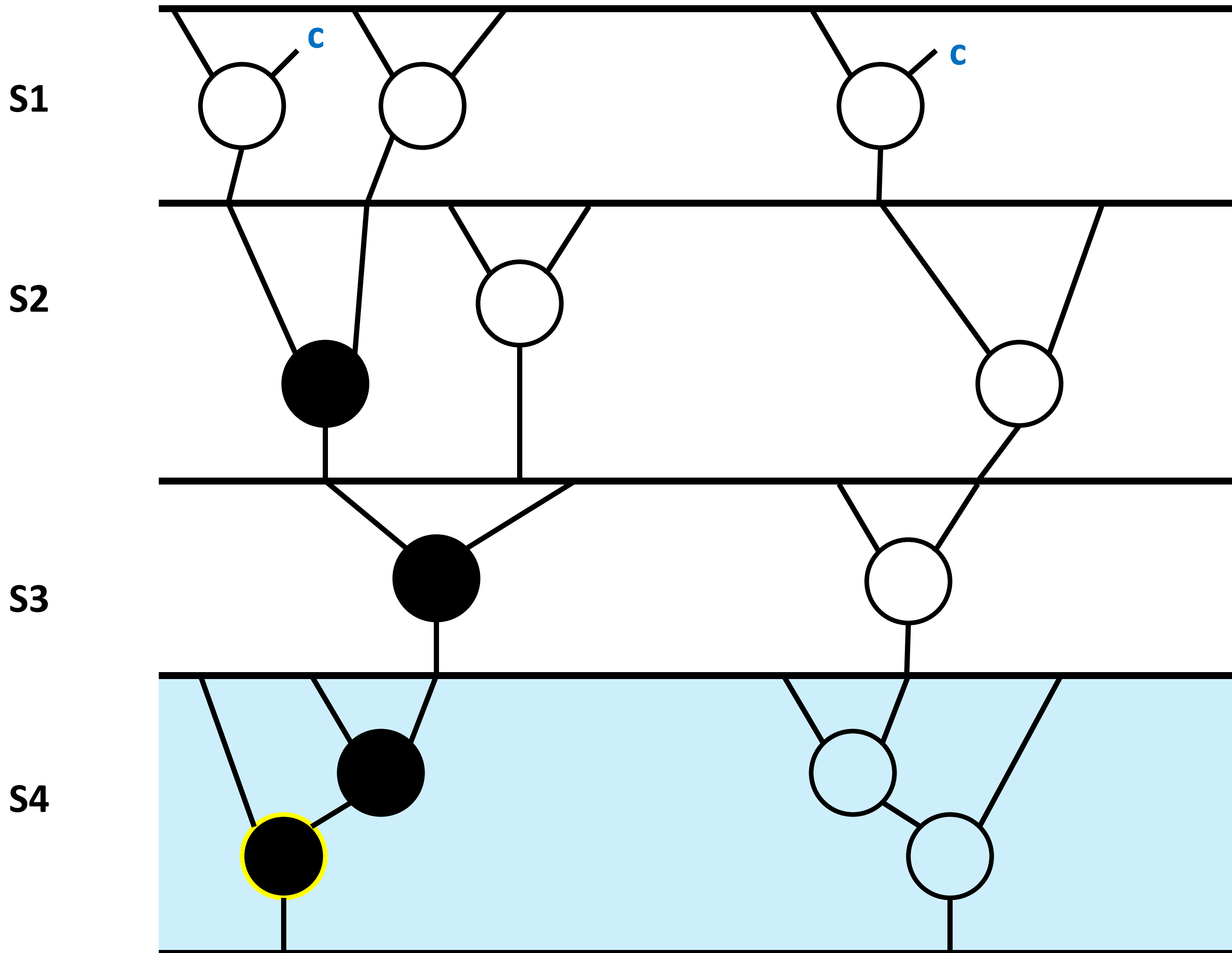
Sharing Algorithm



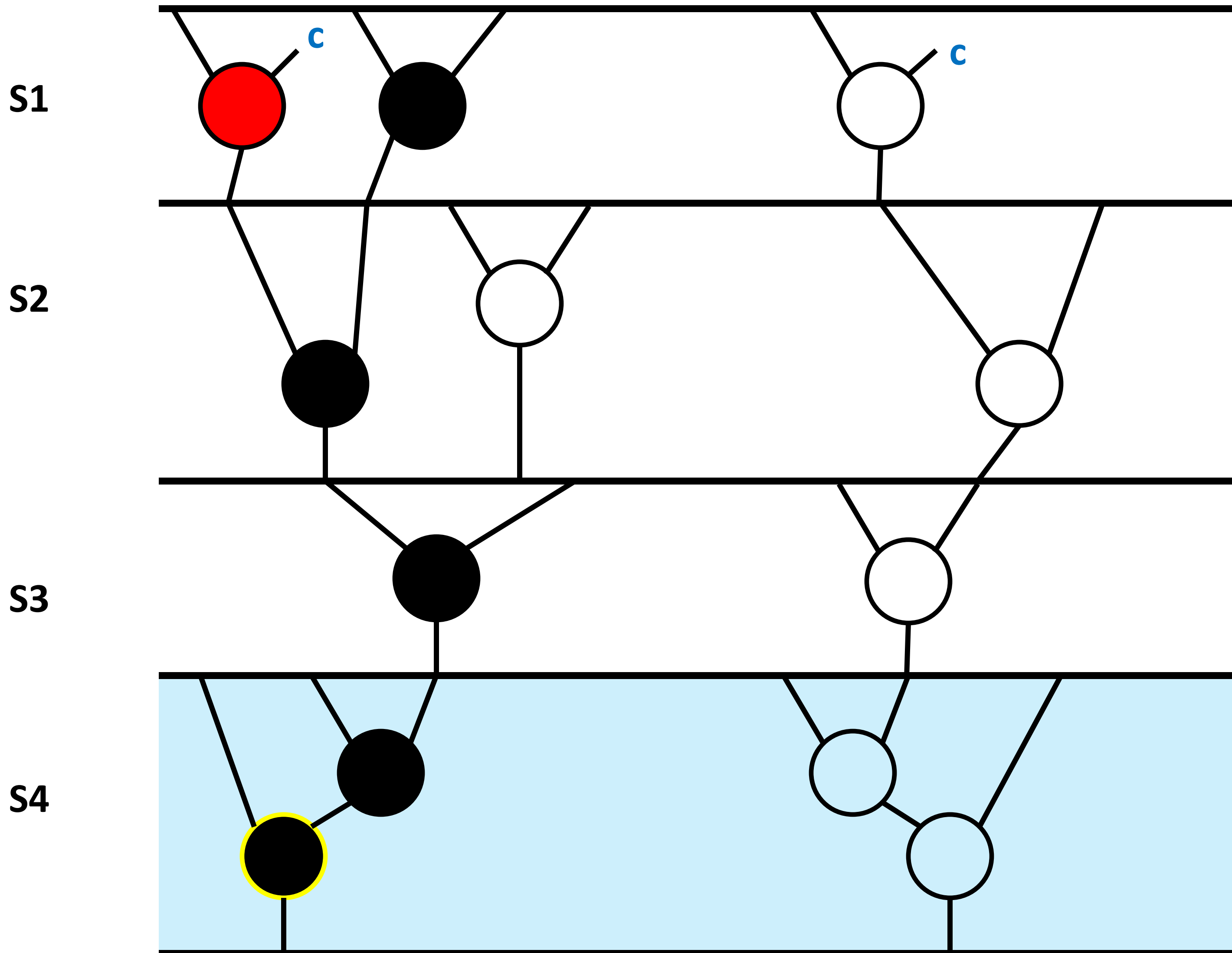
Sharing Algorithm



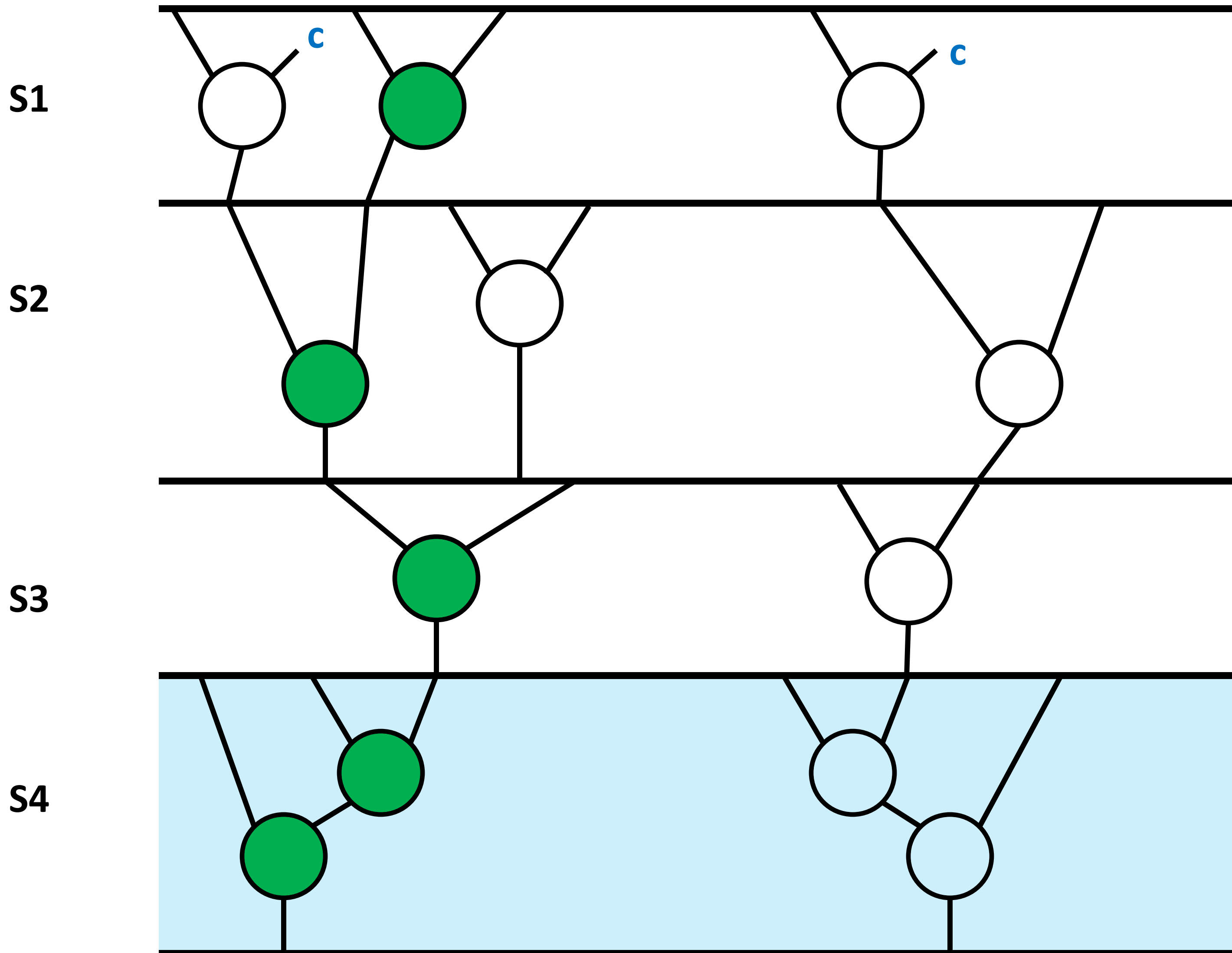
Sharing Algorithm



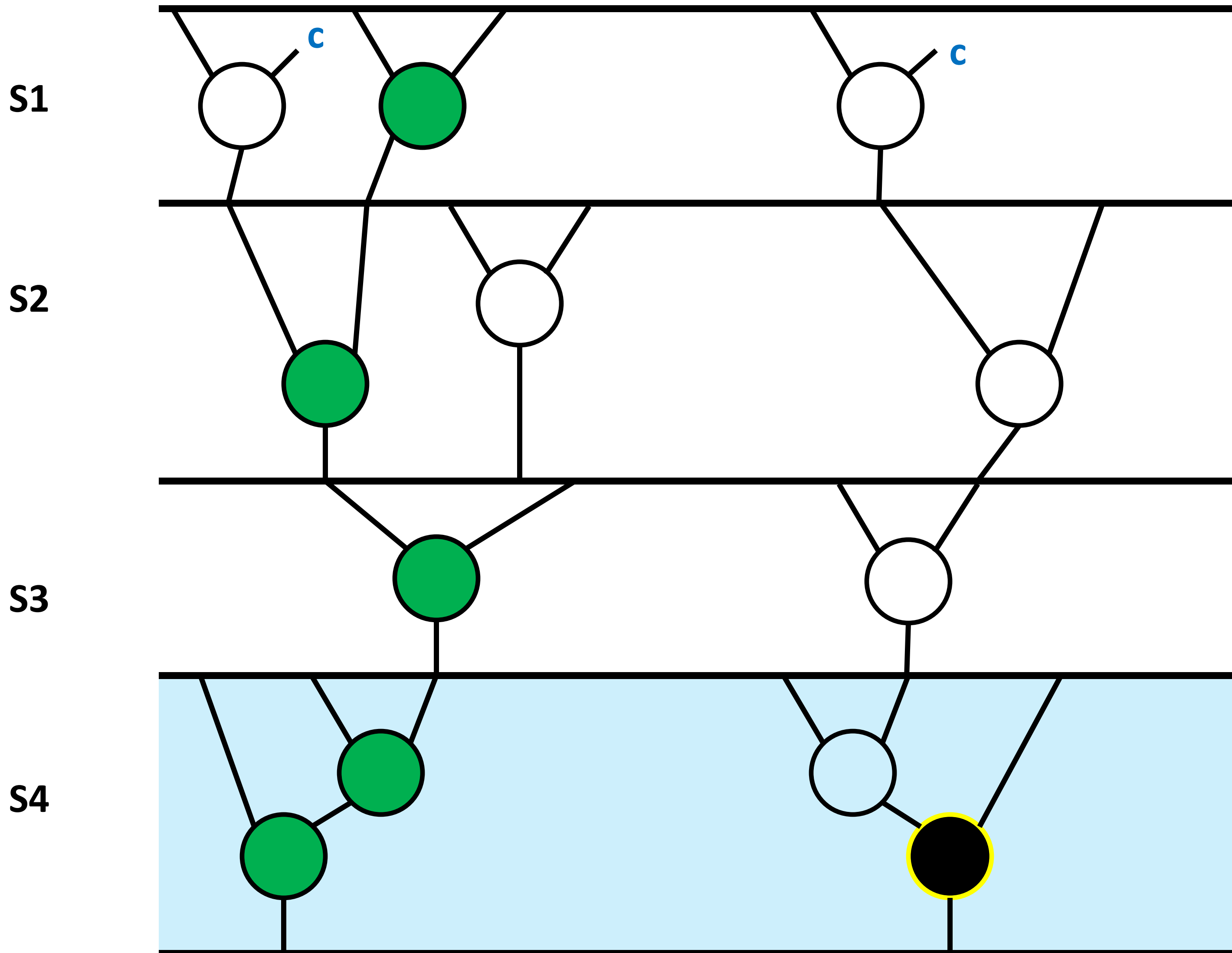
Sharing Algorithm



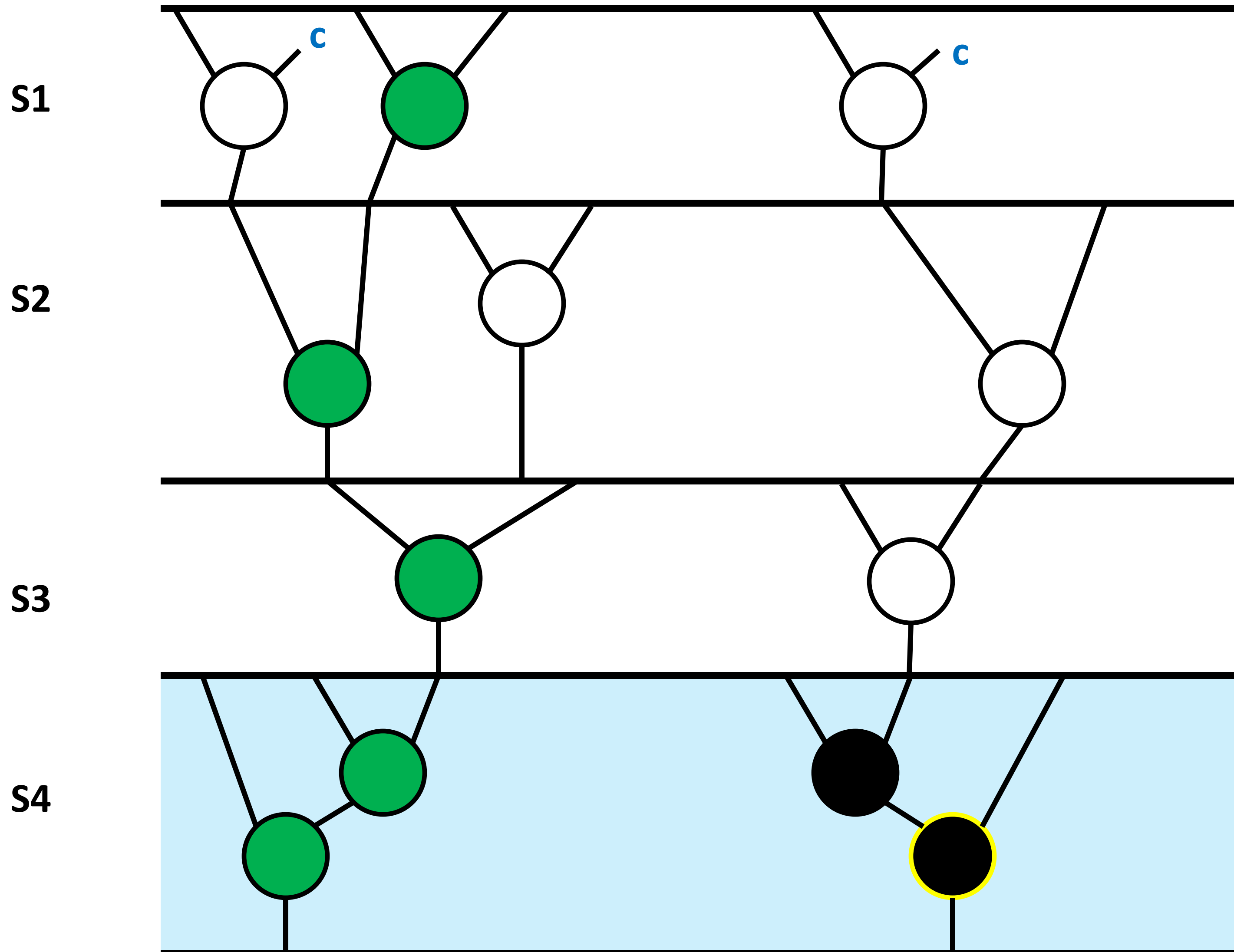
Sharing Algorithm



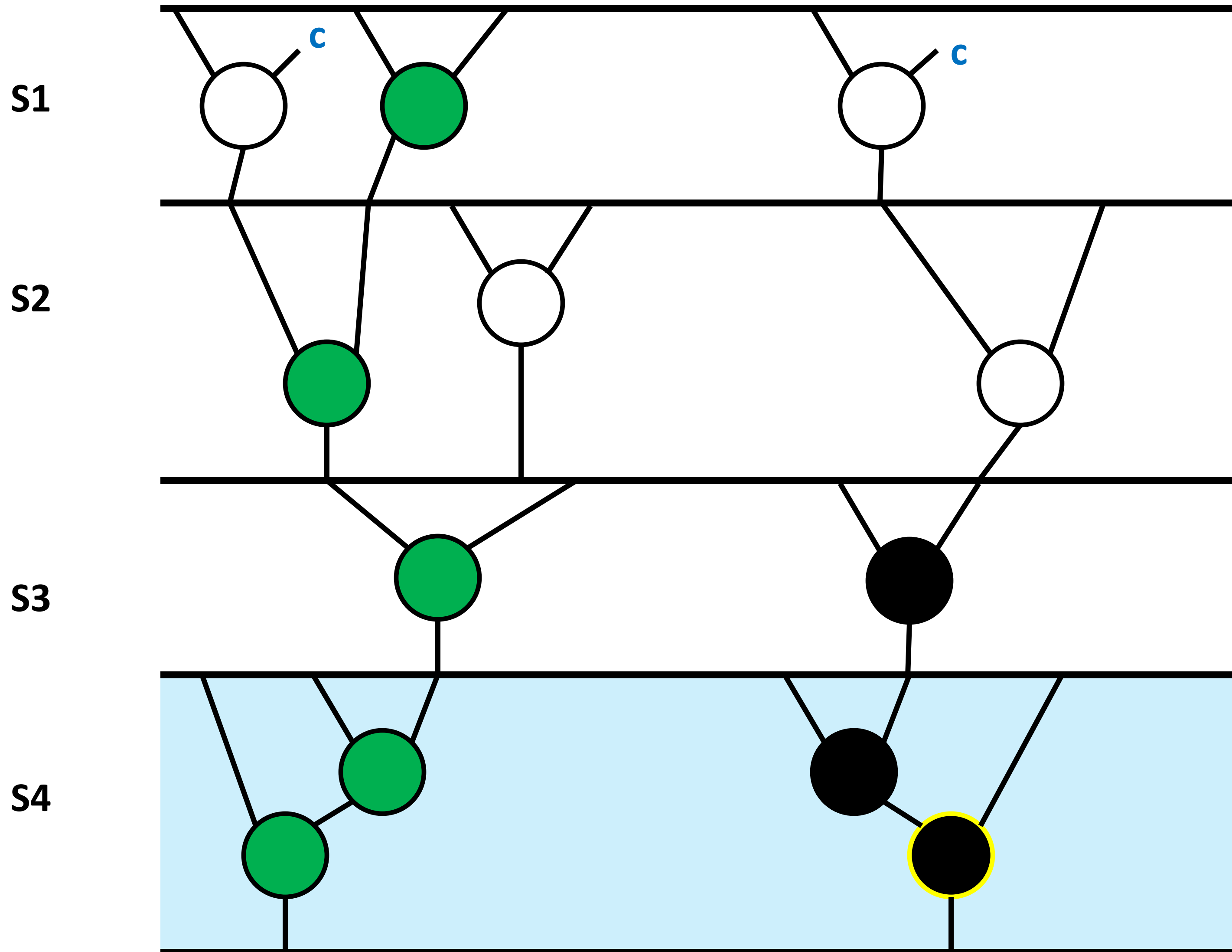
Sharing Algorithm



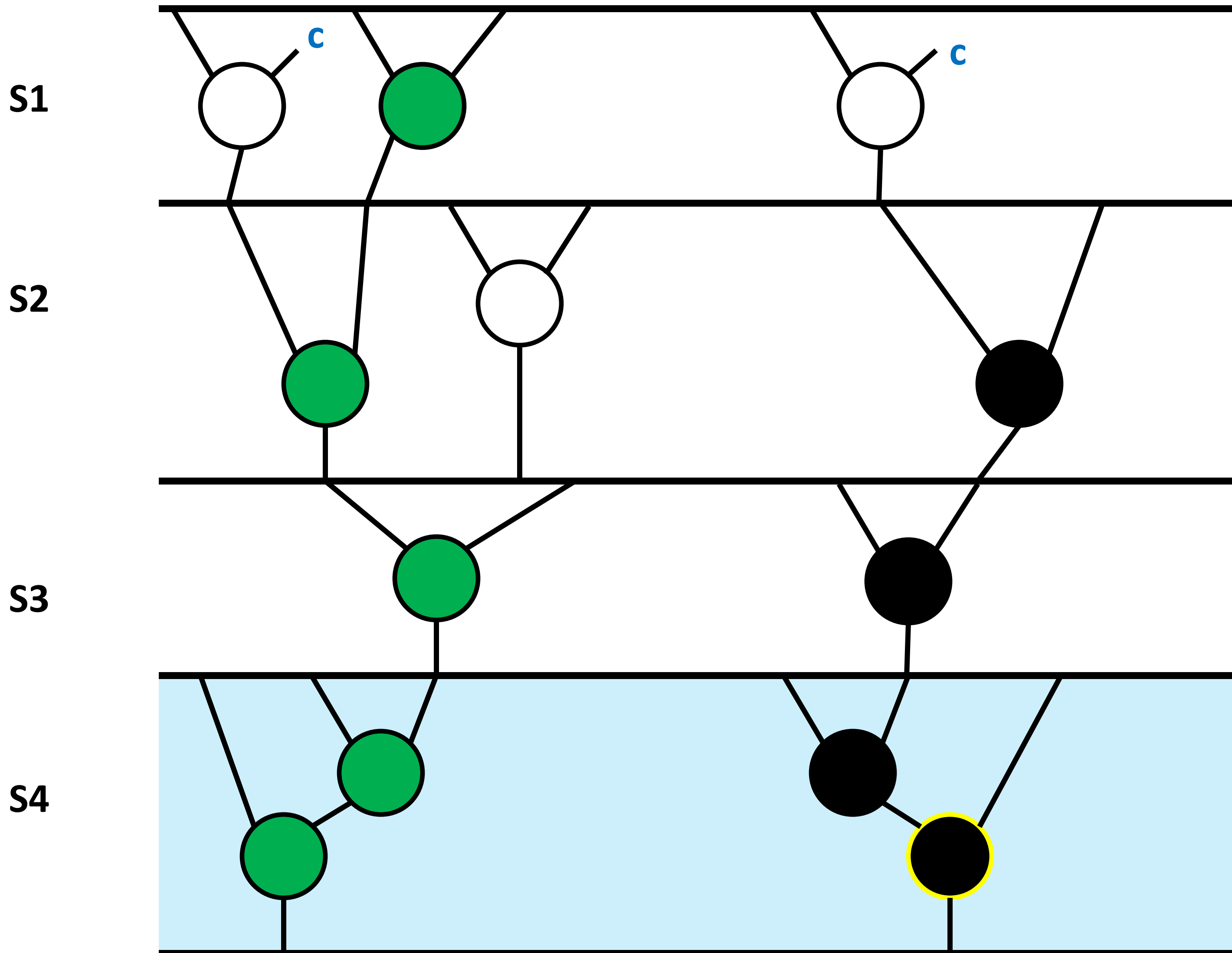
Sharing Algorithm



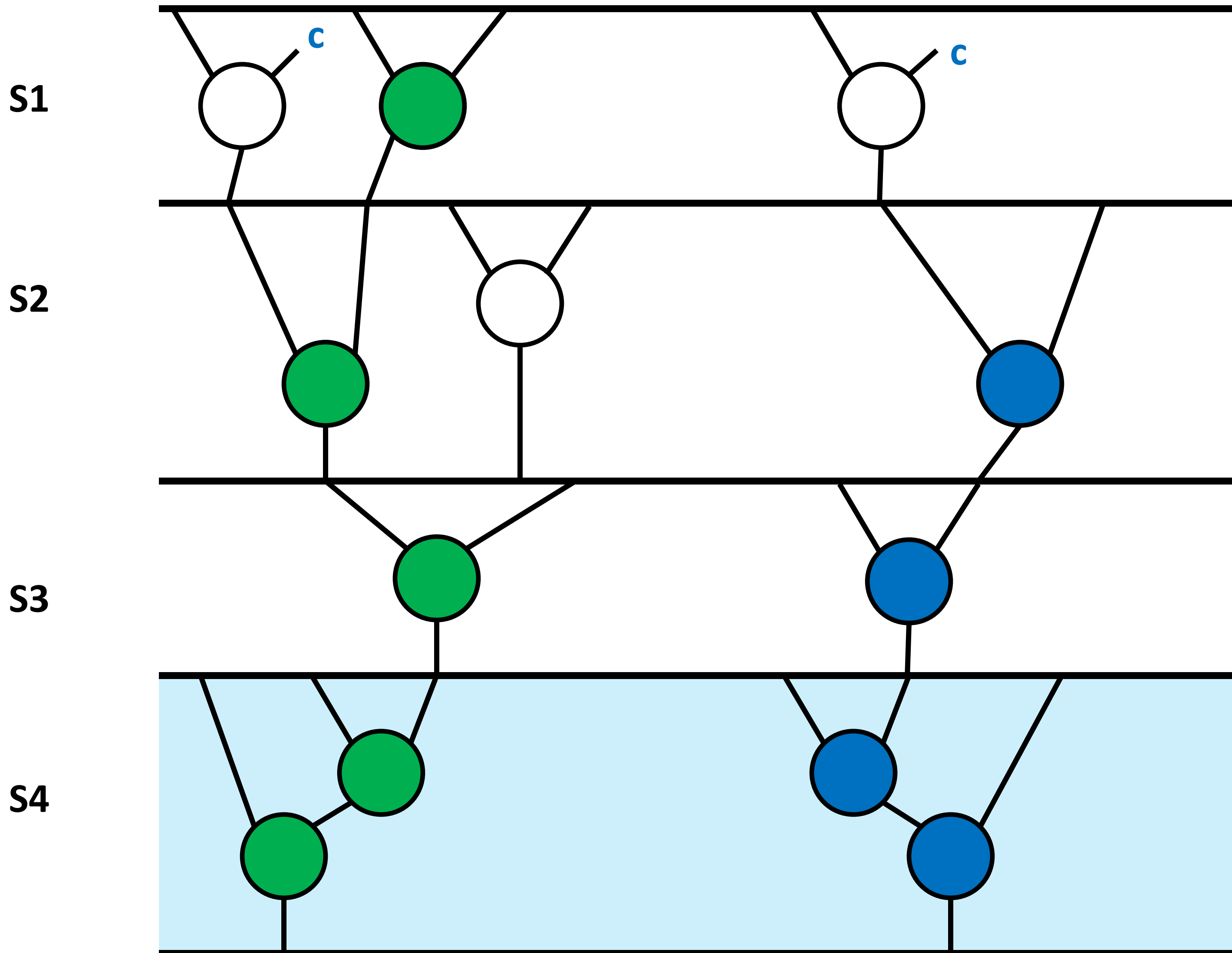
Sharing Algorithm



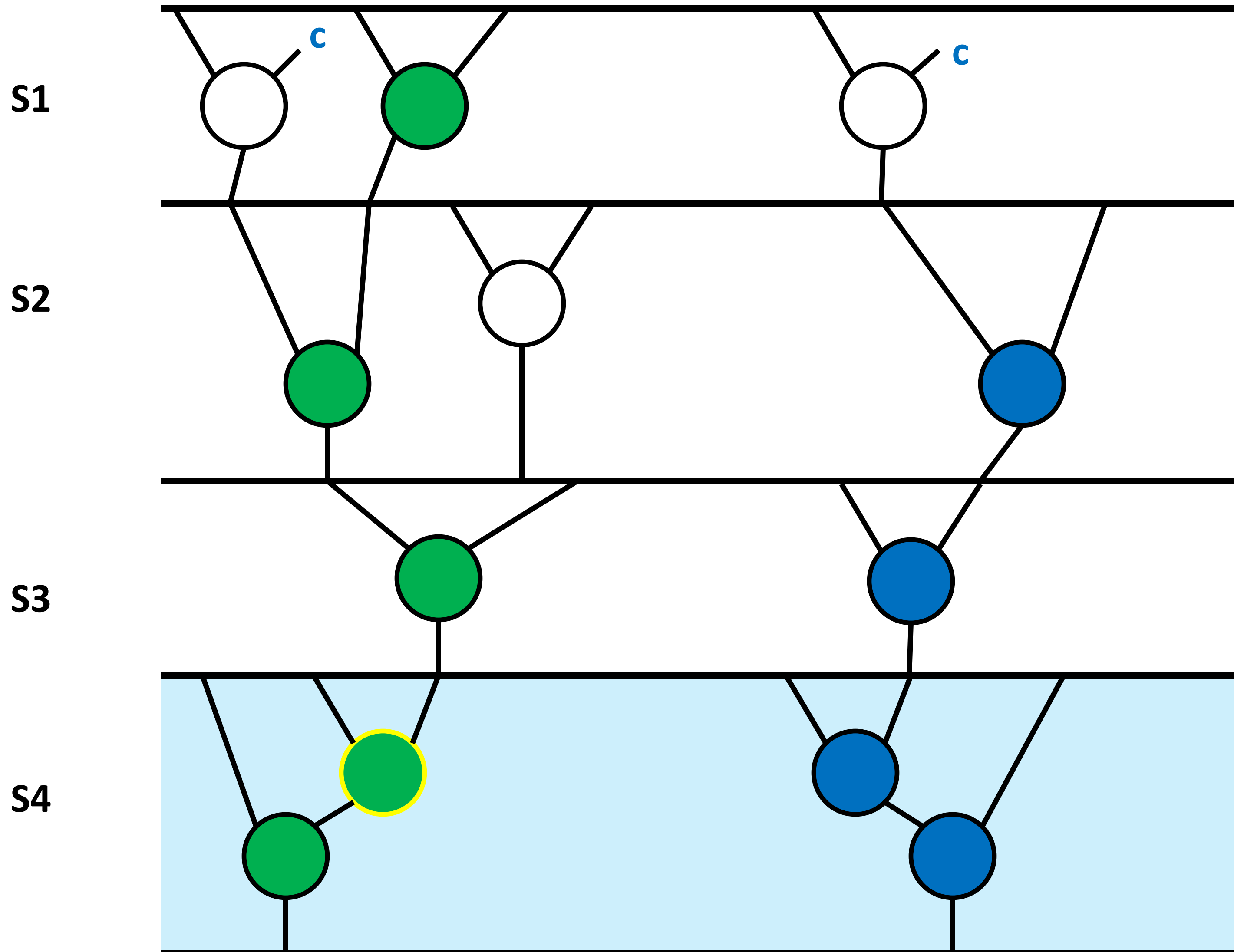
Sharing Algorithm



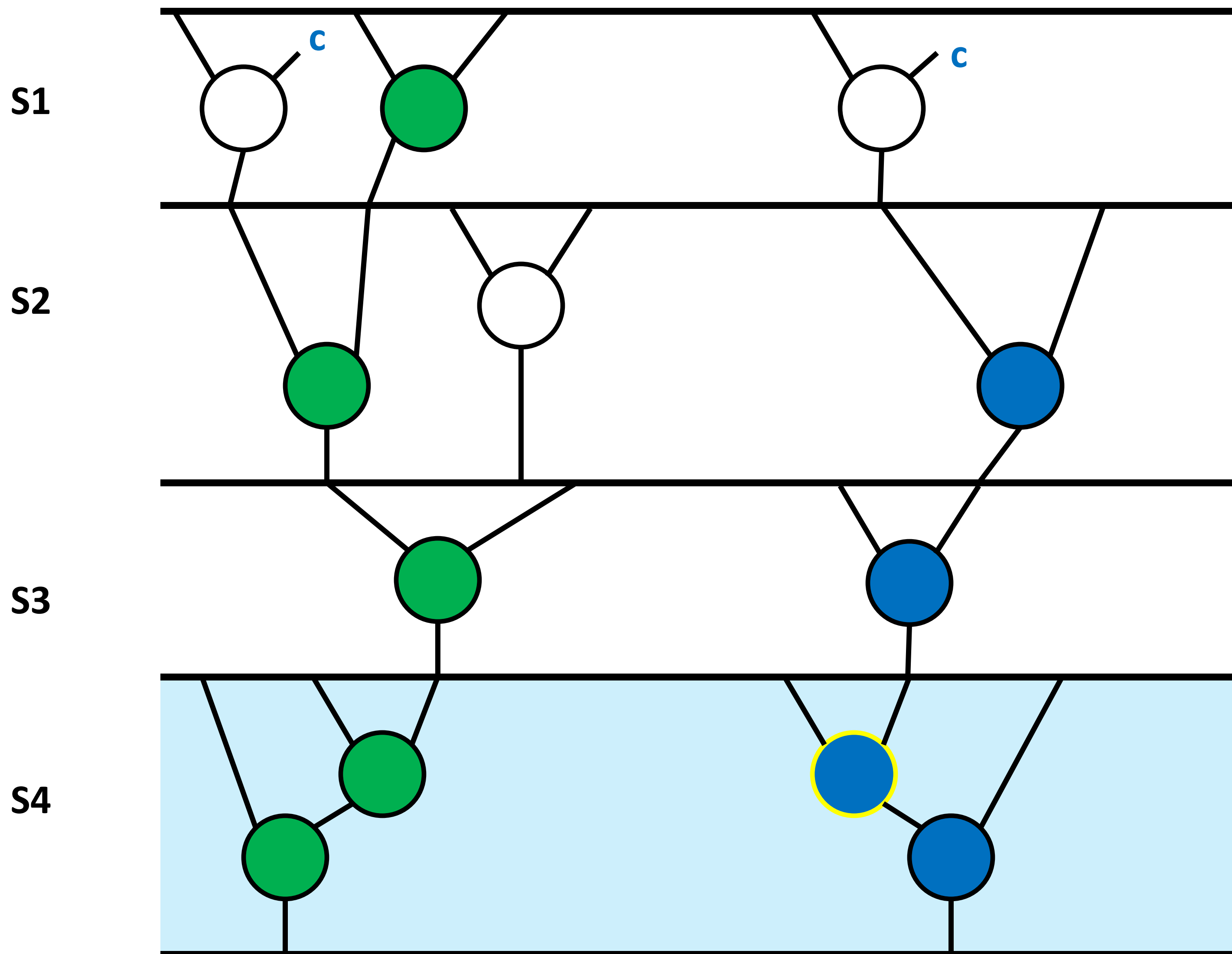
Sharing Algorithm



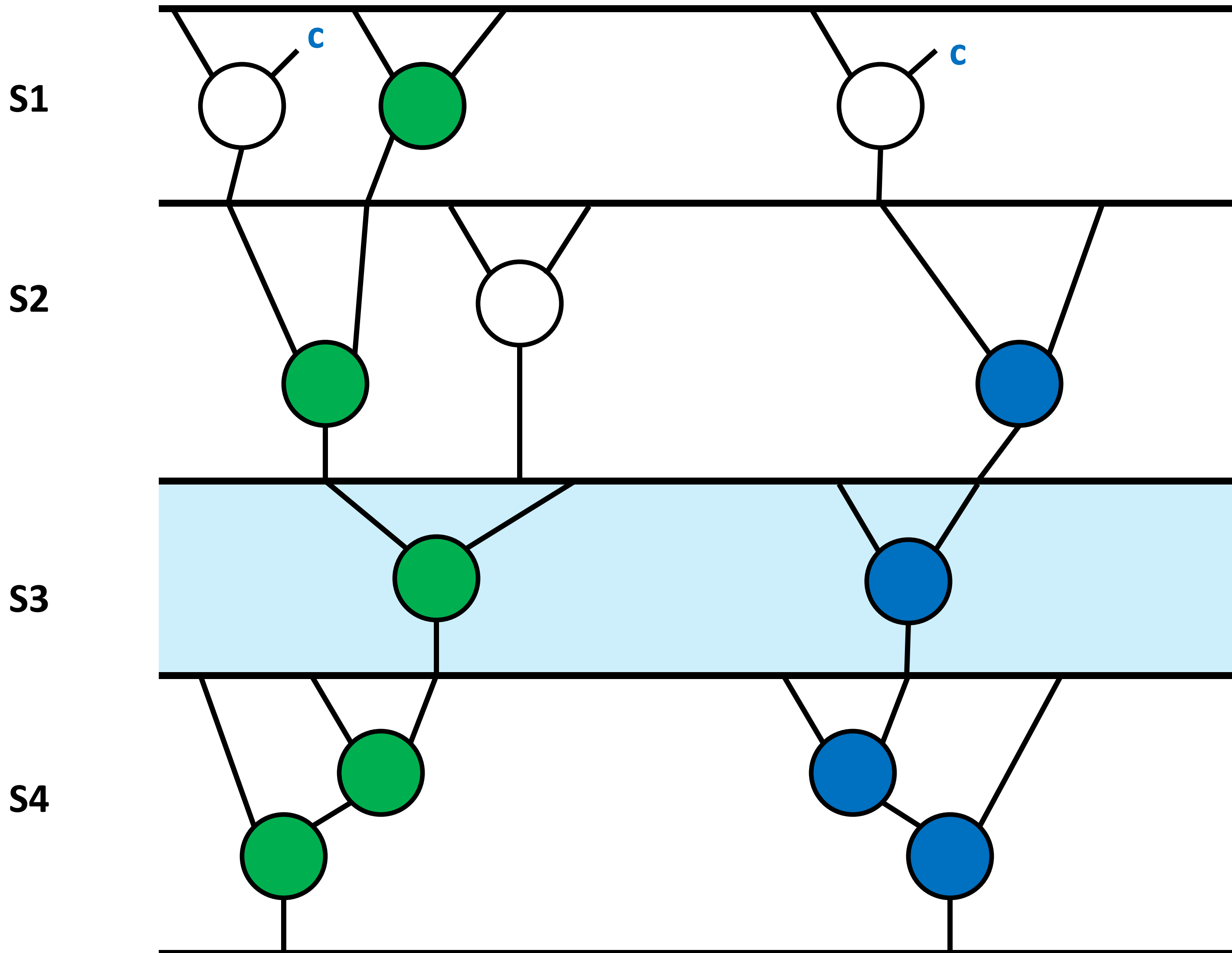
Sharing Algorithm



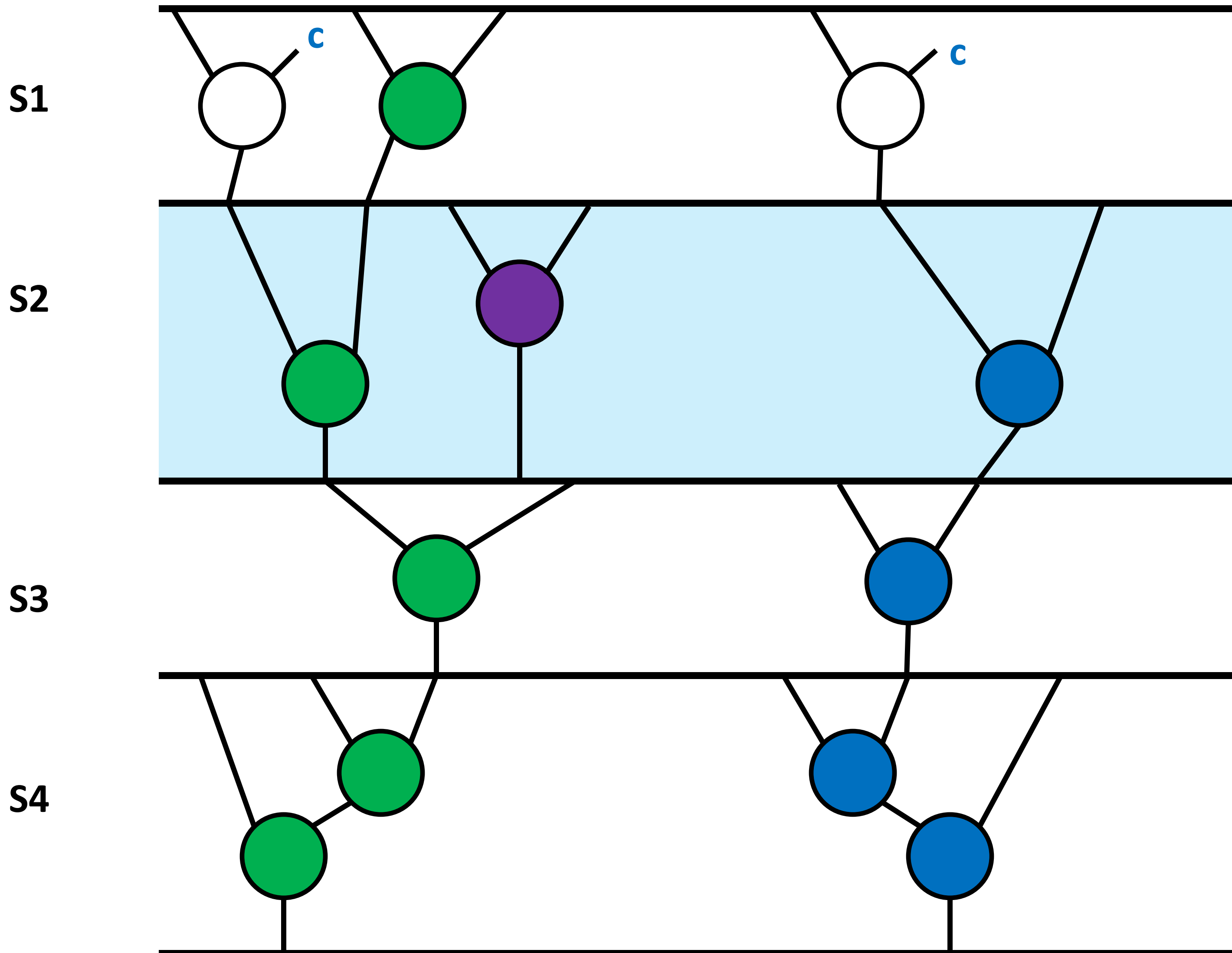
Sharing Algorithm



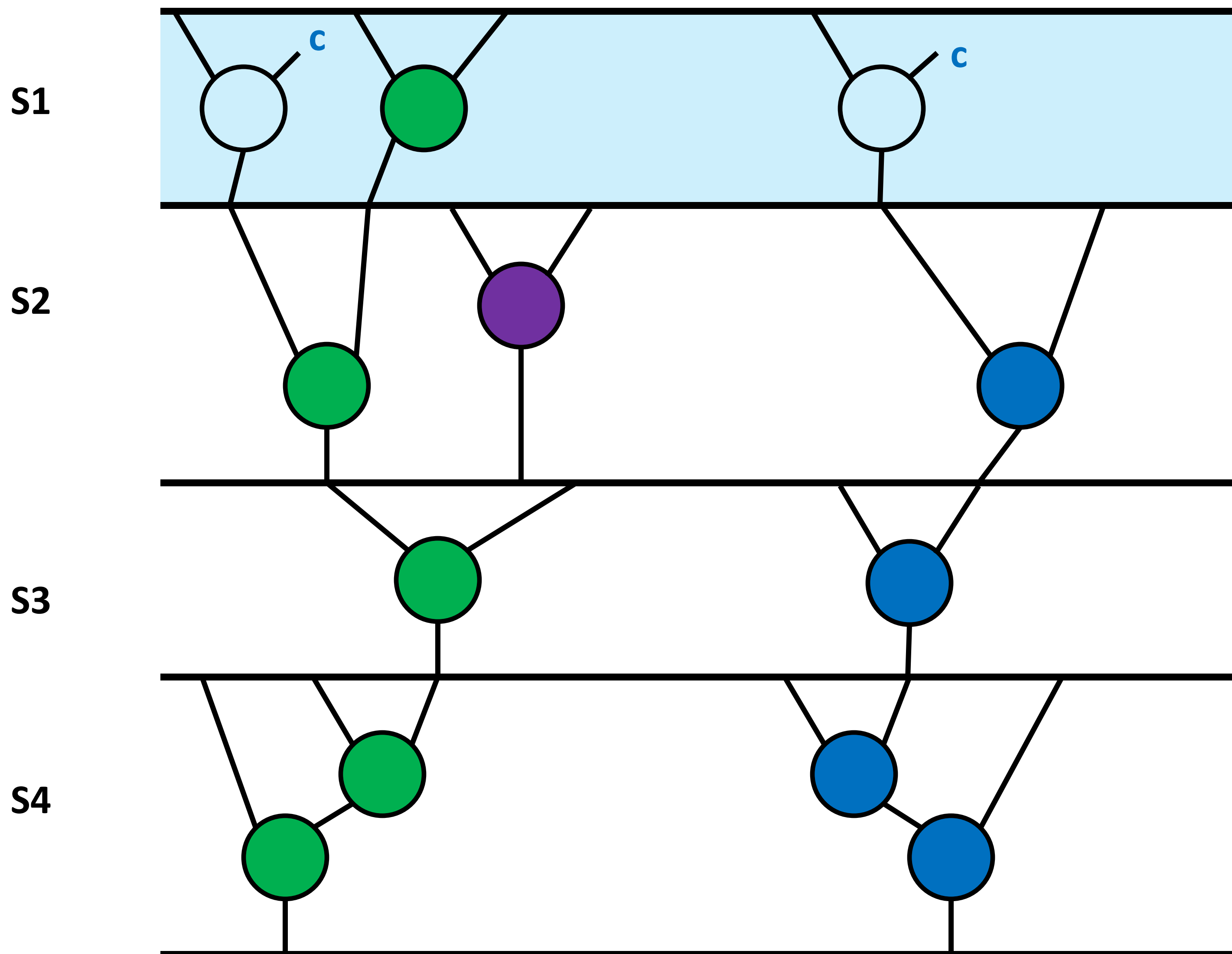
Sharing Algorithm



Sharing Algorithm

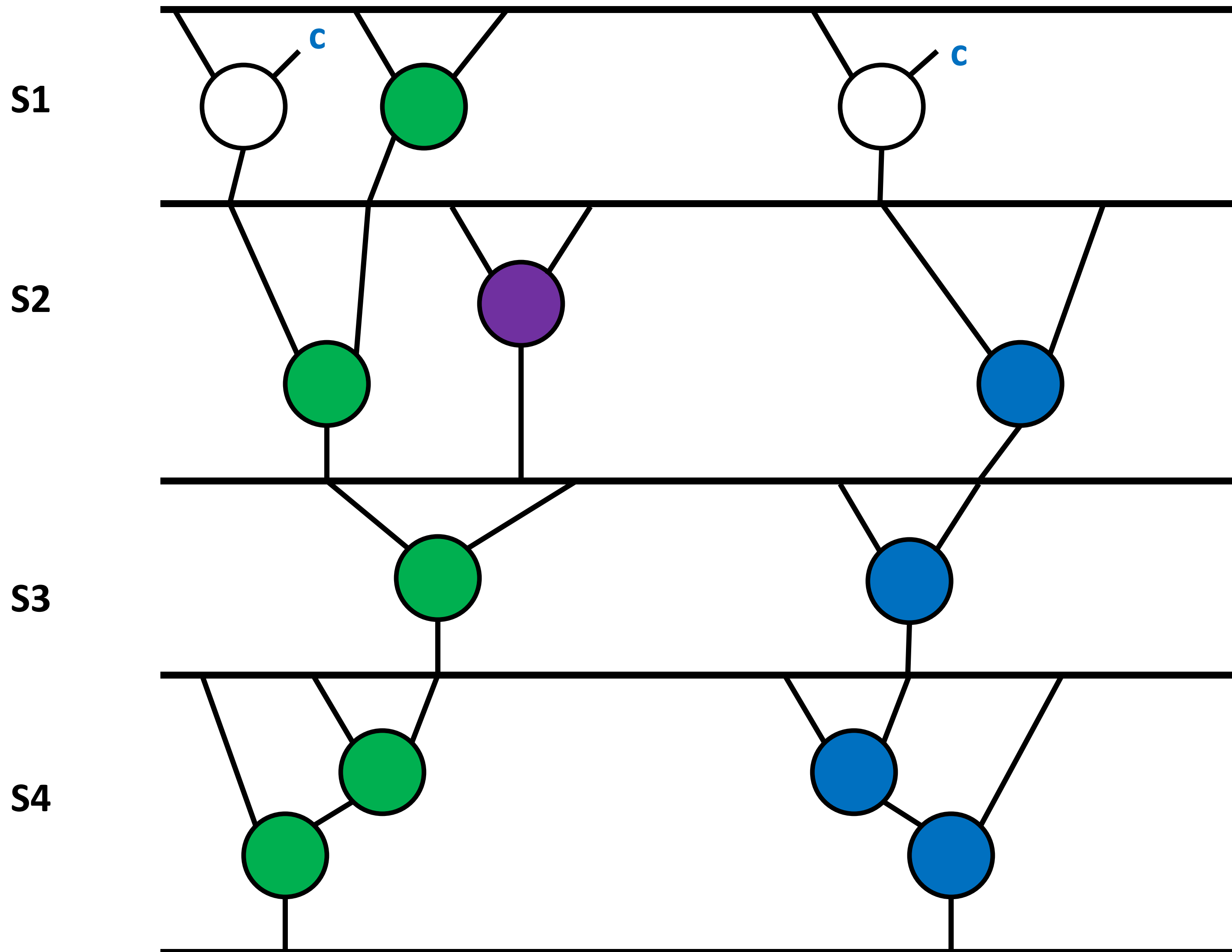


Sharing Algorithm



Sharing Algorithm

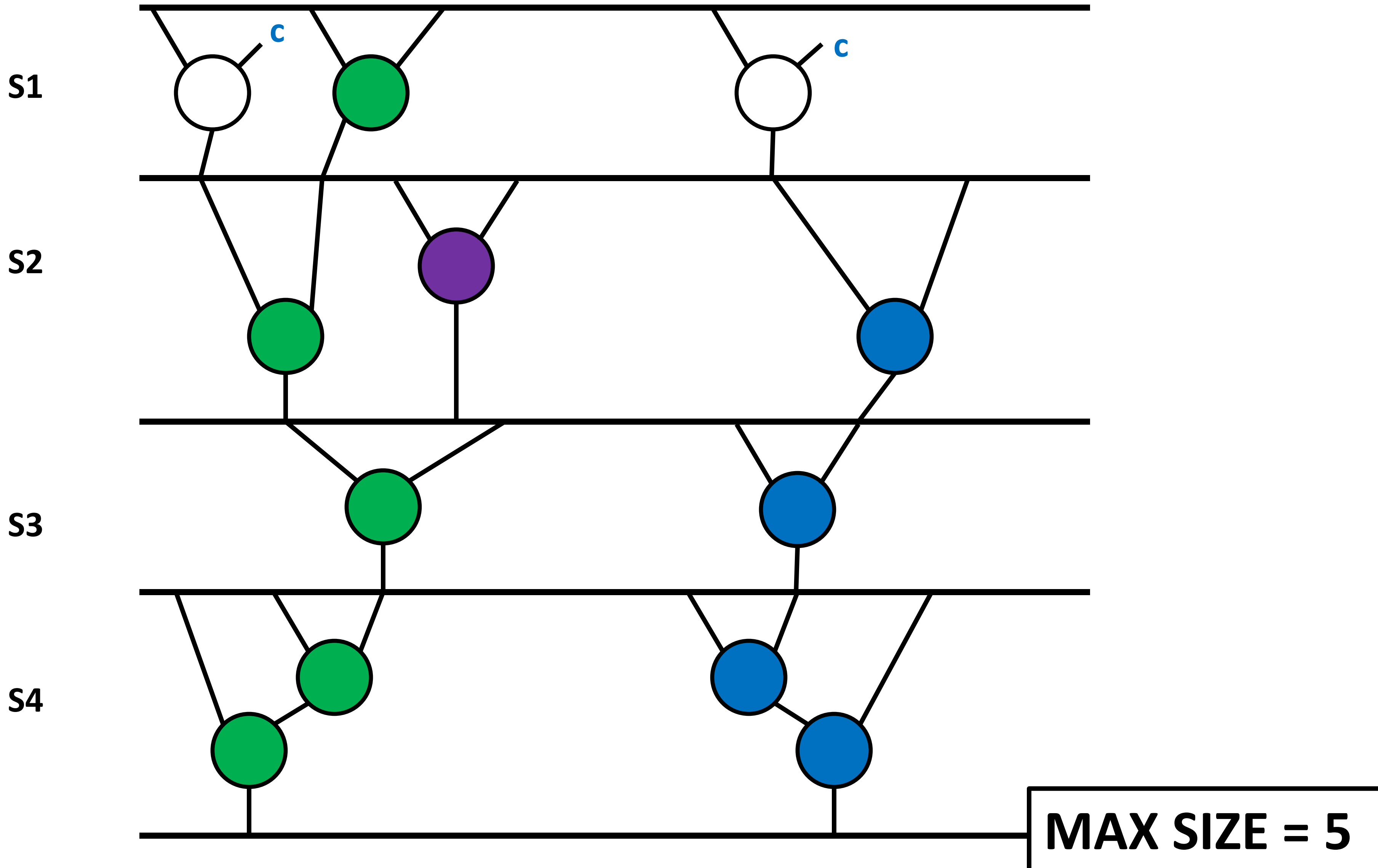
Number of Graphs found: 3



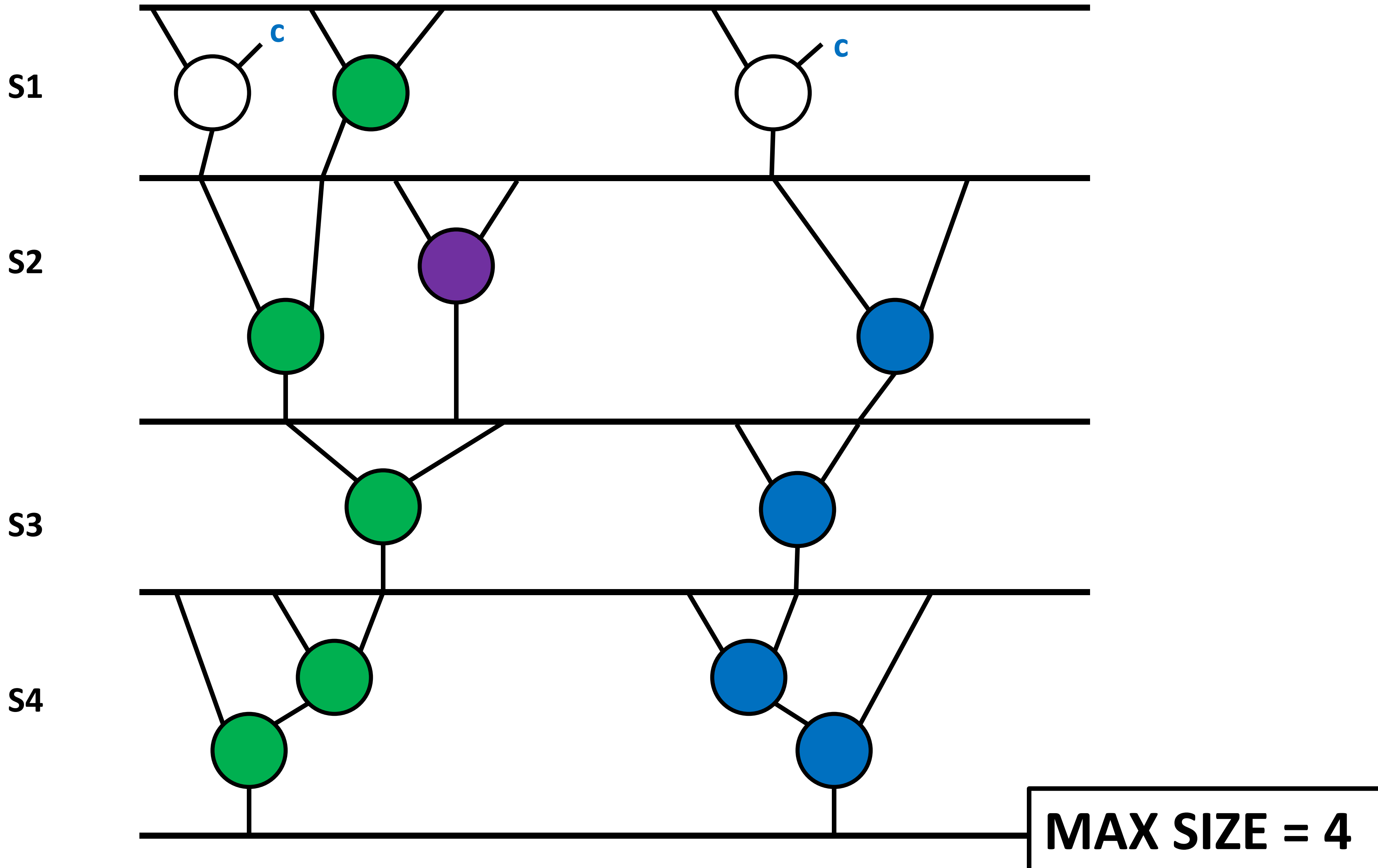
Sharing Algorithm

- The animation illustrates how graphs are <u>found</u>
- However, the three graphs all occurred with frequency 1 and hence no sharing could take place

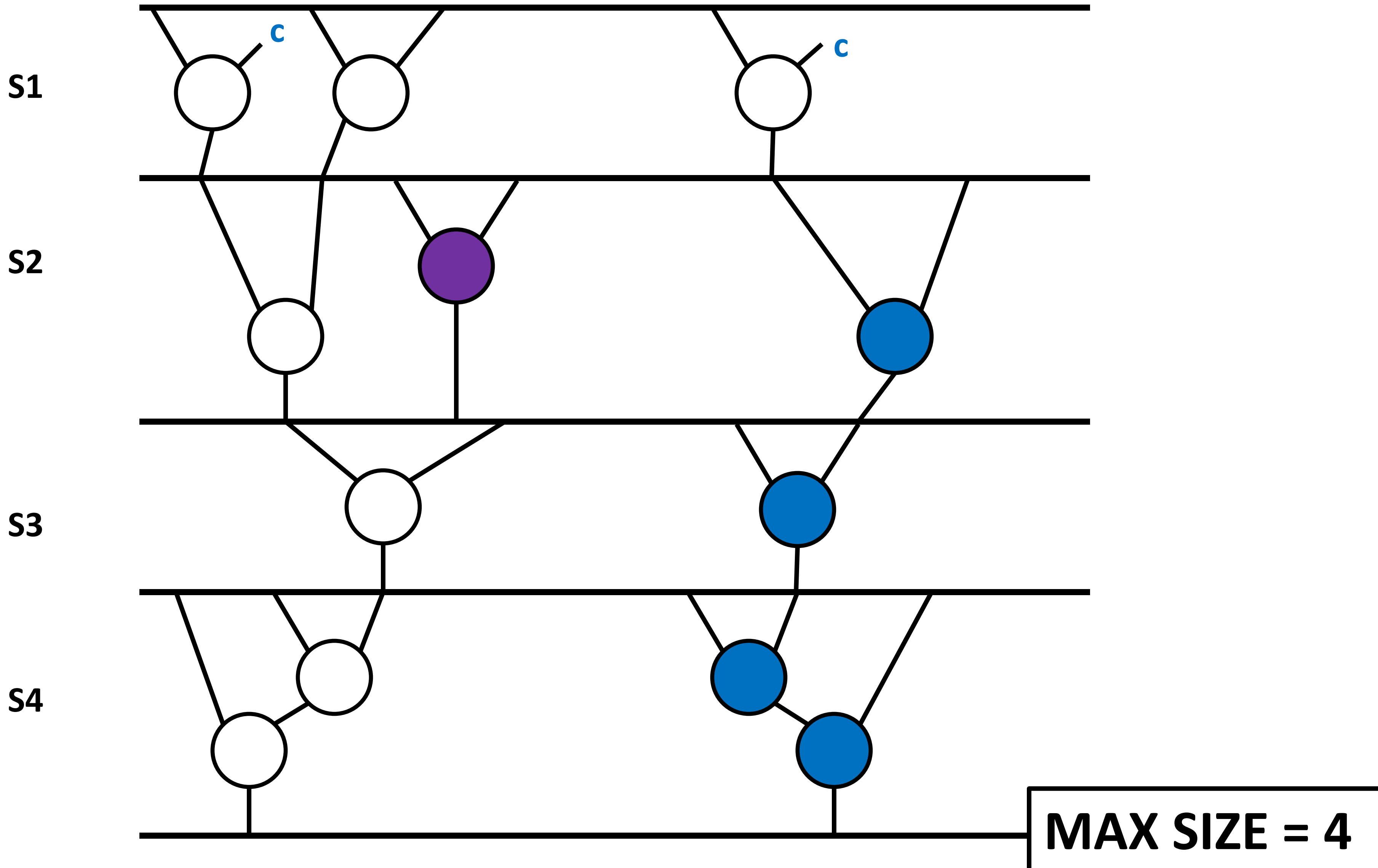
Sharing Algorithm



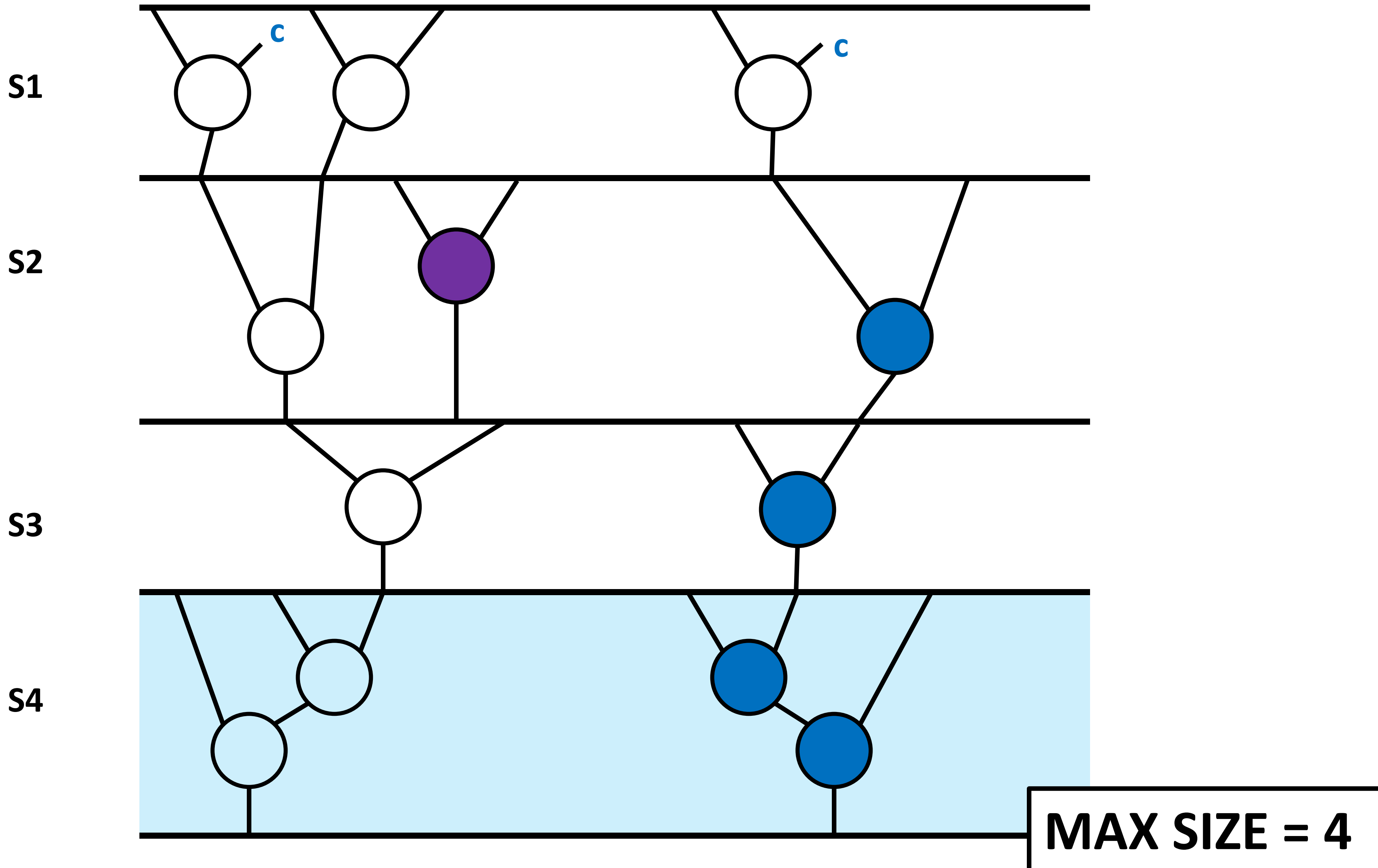
Sharing Algorithm



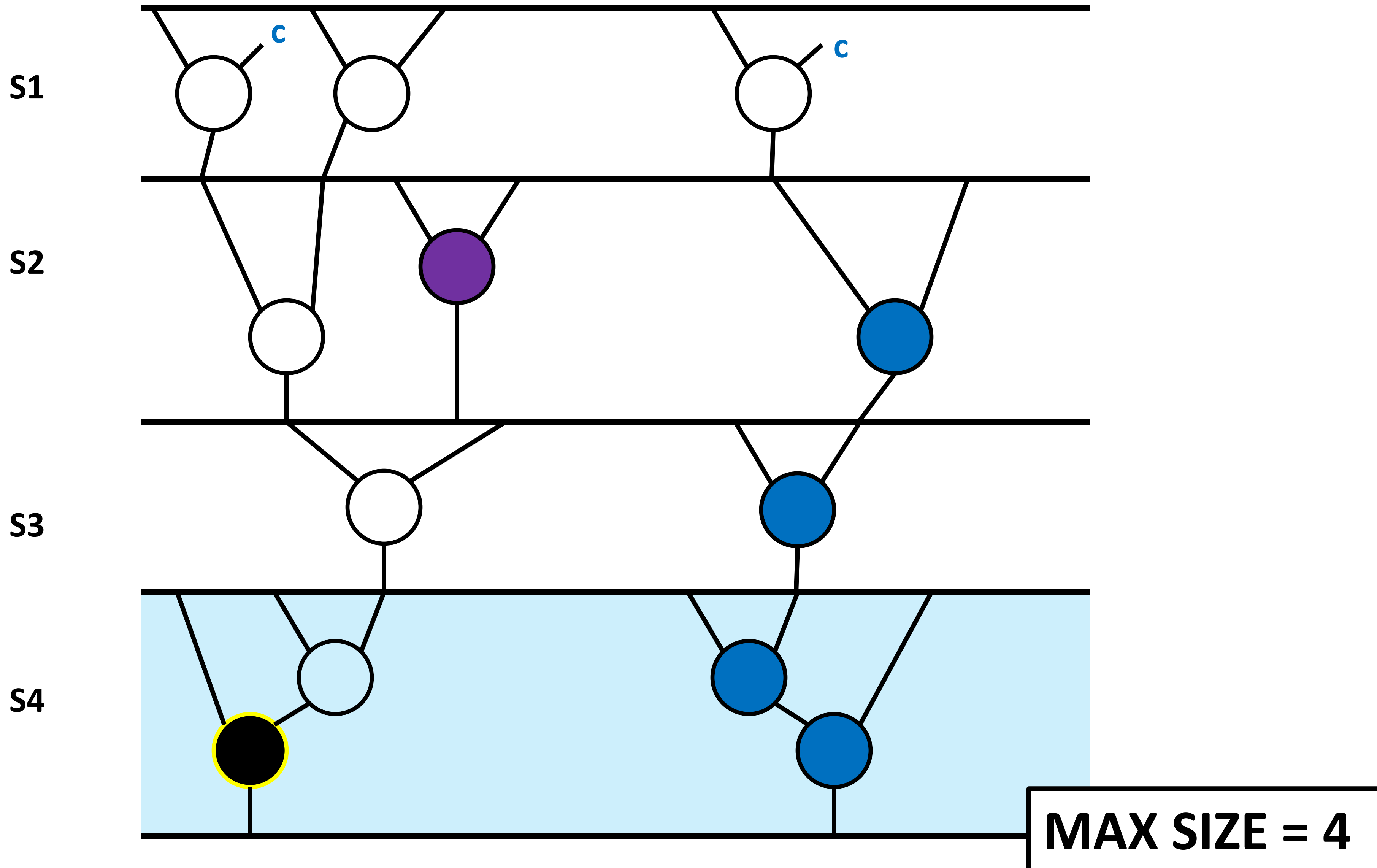
Sharing Algorithm



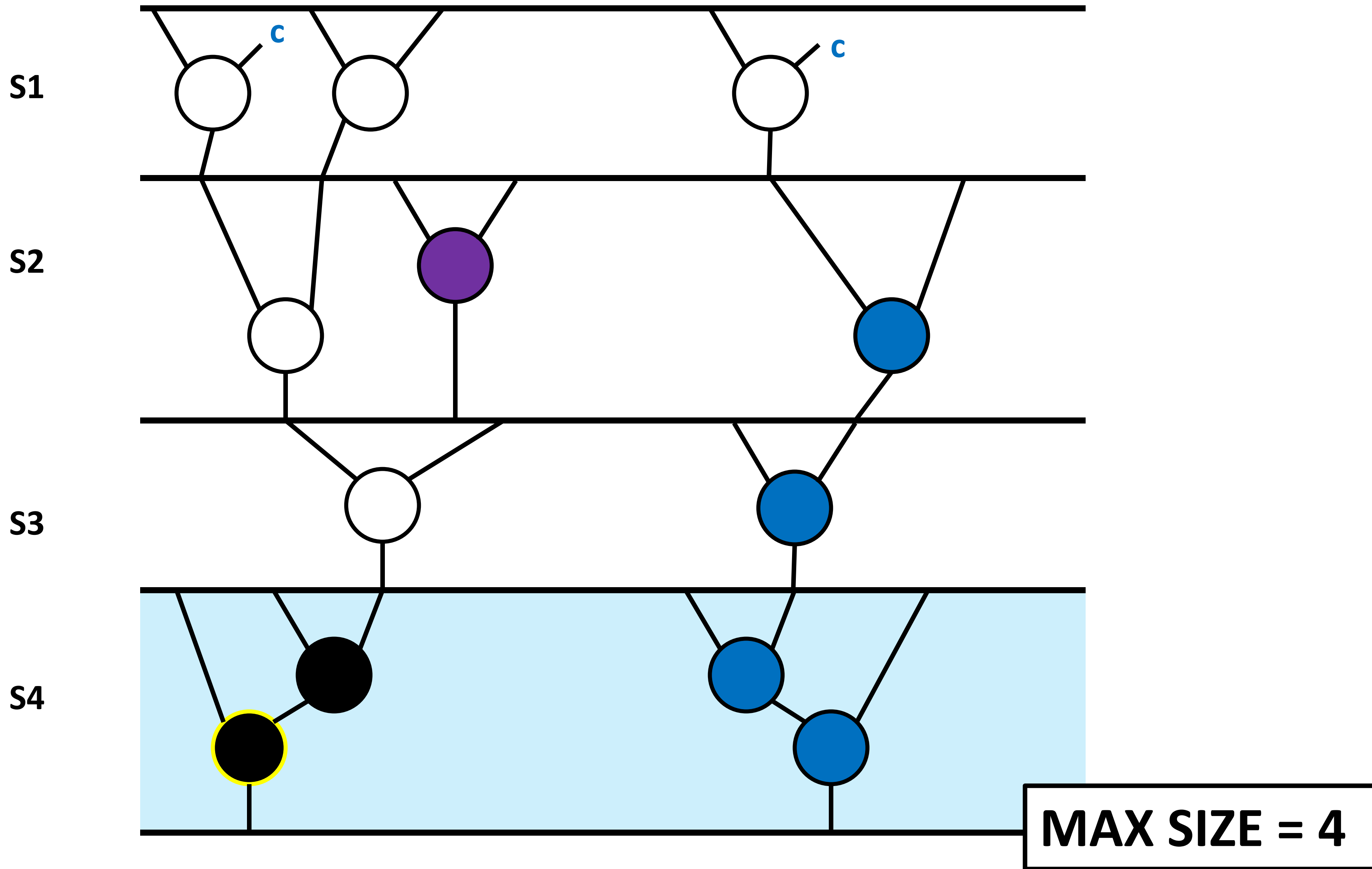
Sharing Algorithm



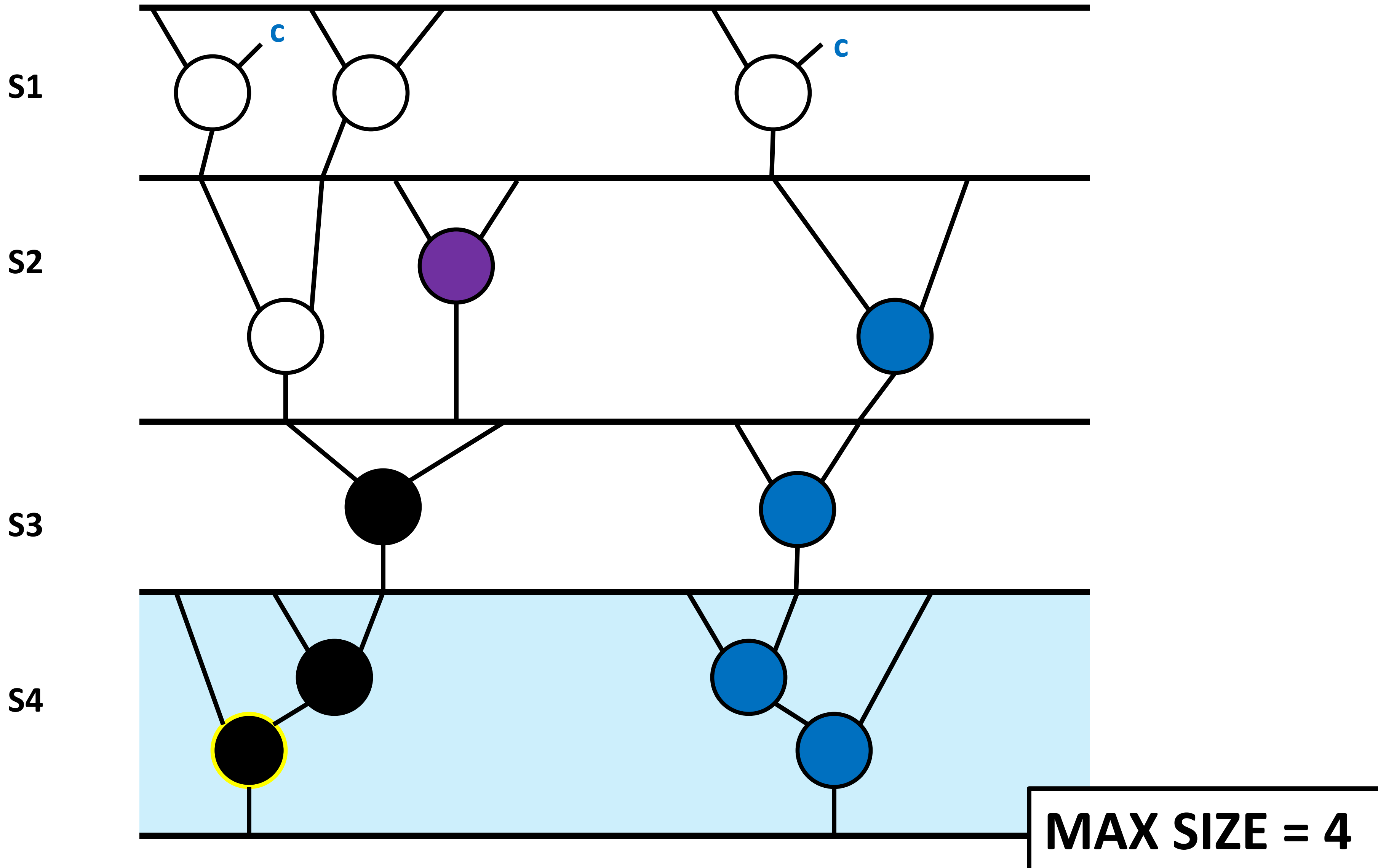
Sharing Algorithm



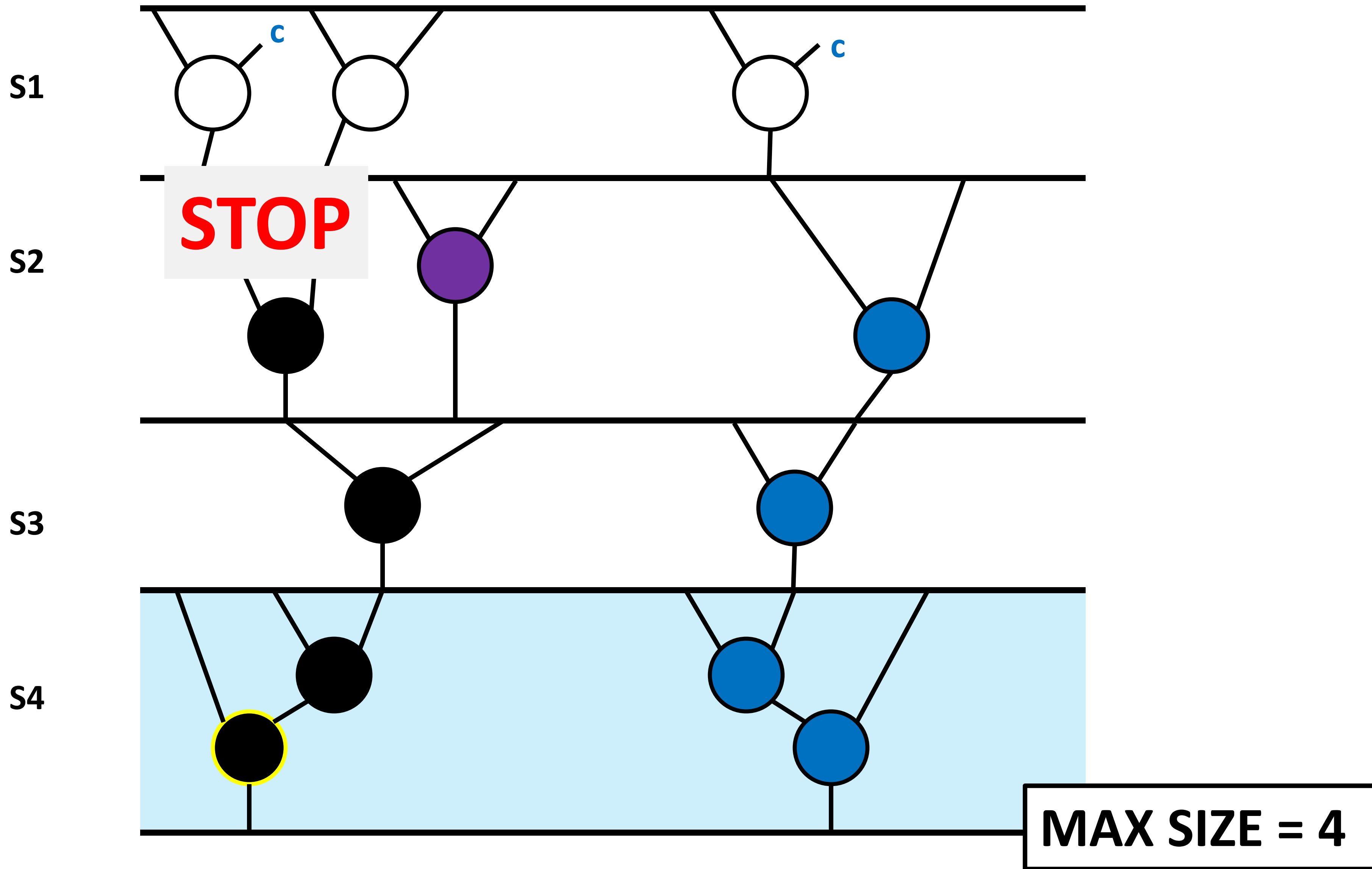
Sharing Algorithm



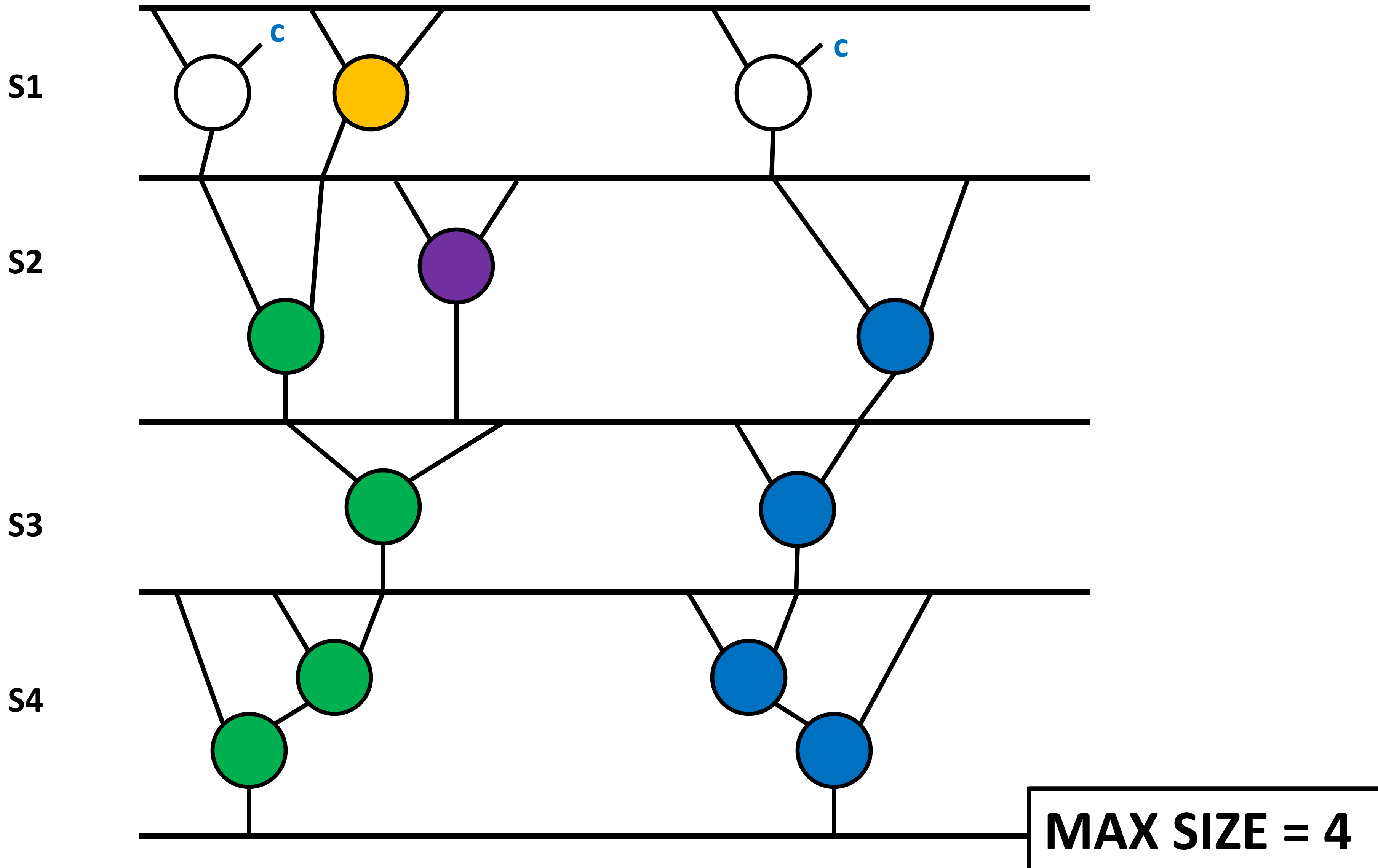
Sharing Algorithm



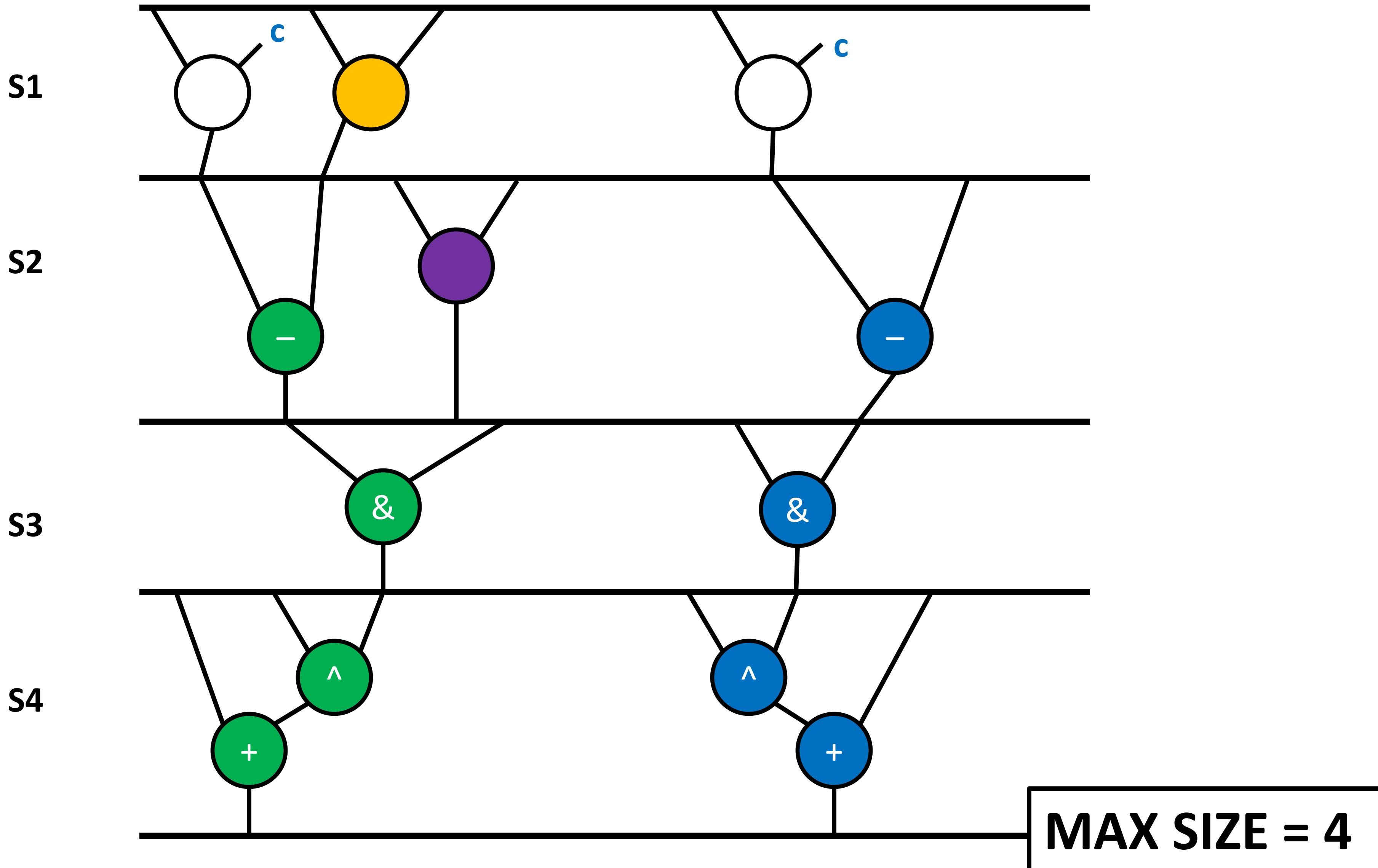
Sharing Algorithm



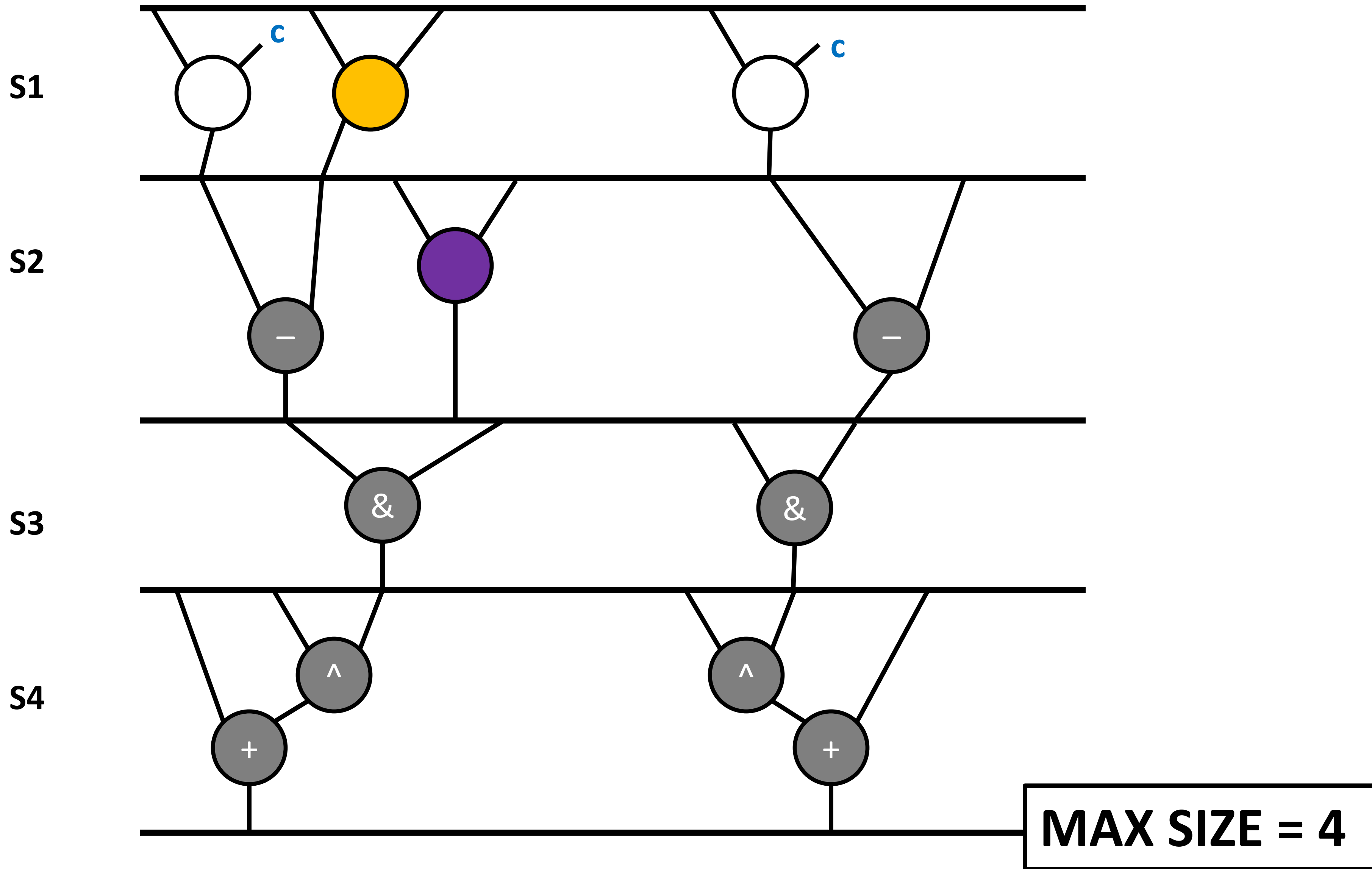
Sharing Algorithm



Sharing Algorithm



Sharing Algorithm

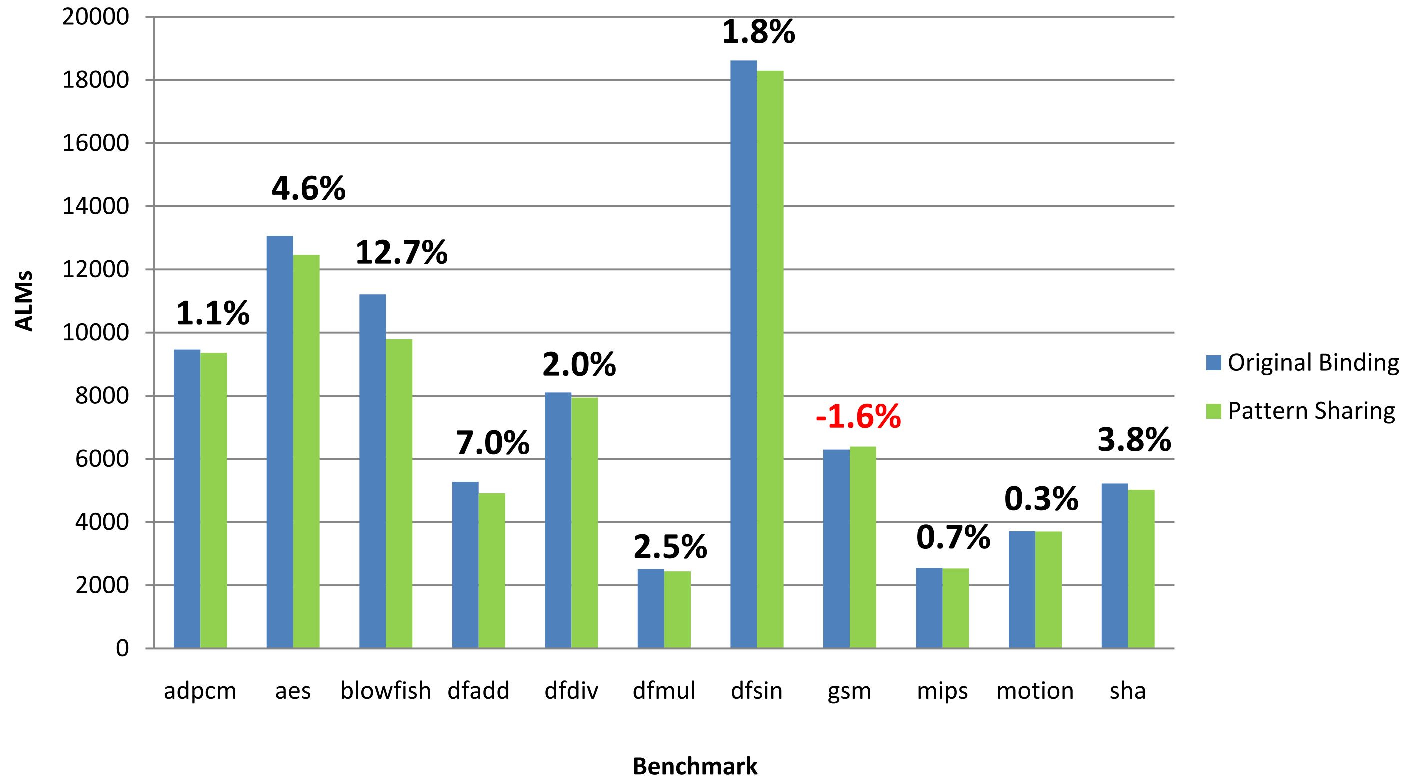


Sharing Algorithm

- These graphs still can’t be shared however because their live intervals overlap, therefore this is the last check in the algorithm
- Graphs are then shared in LegUp Binding.cpp

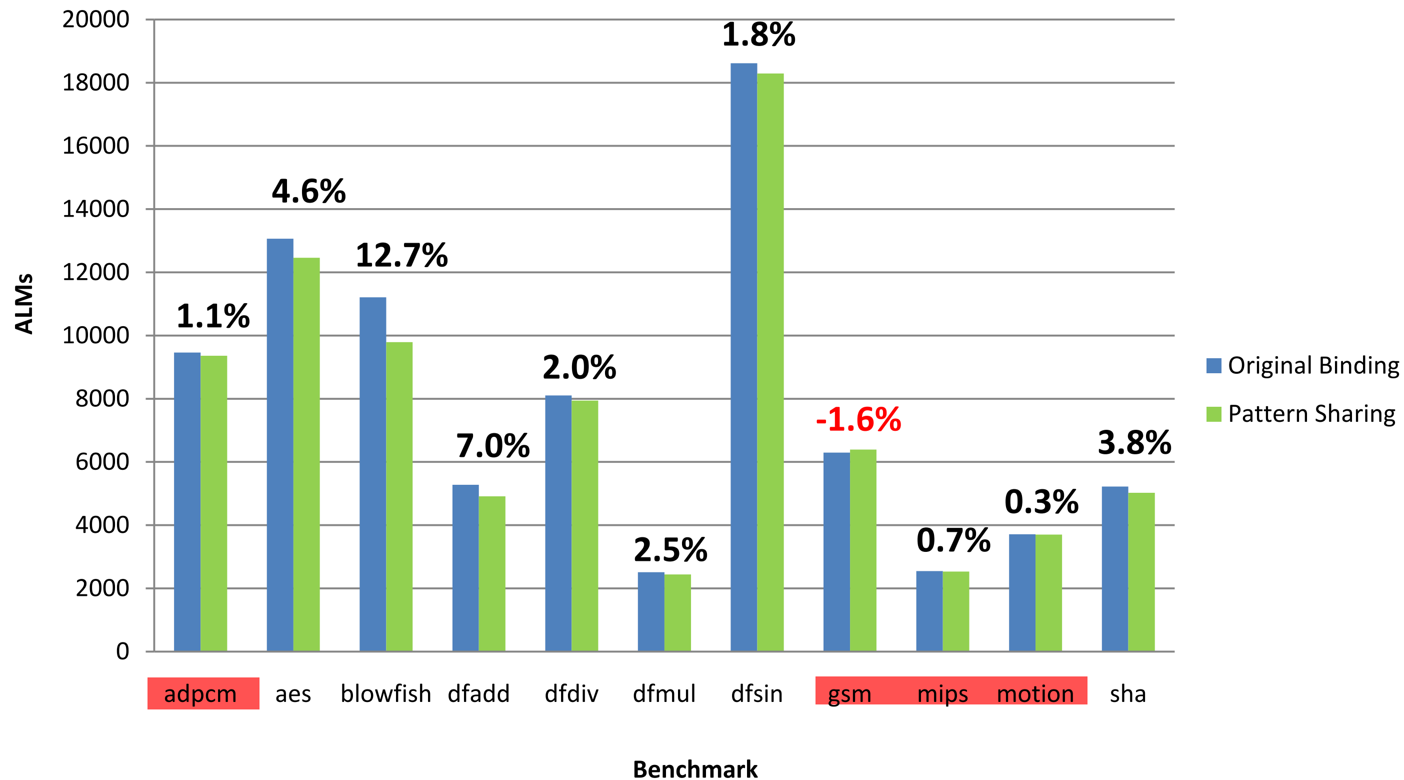
Results

ALMs

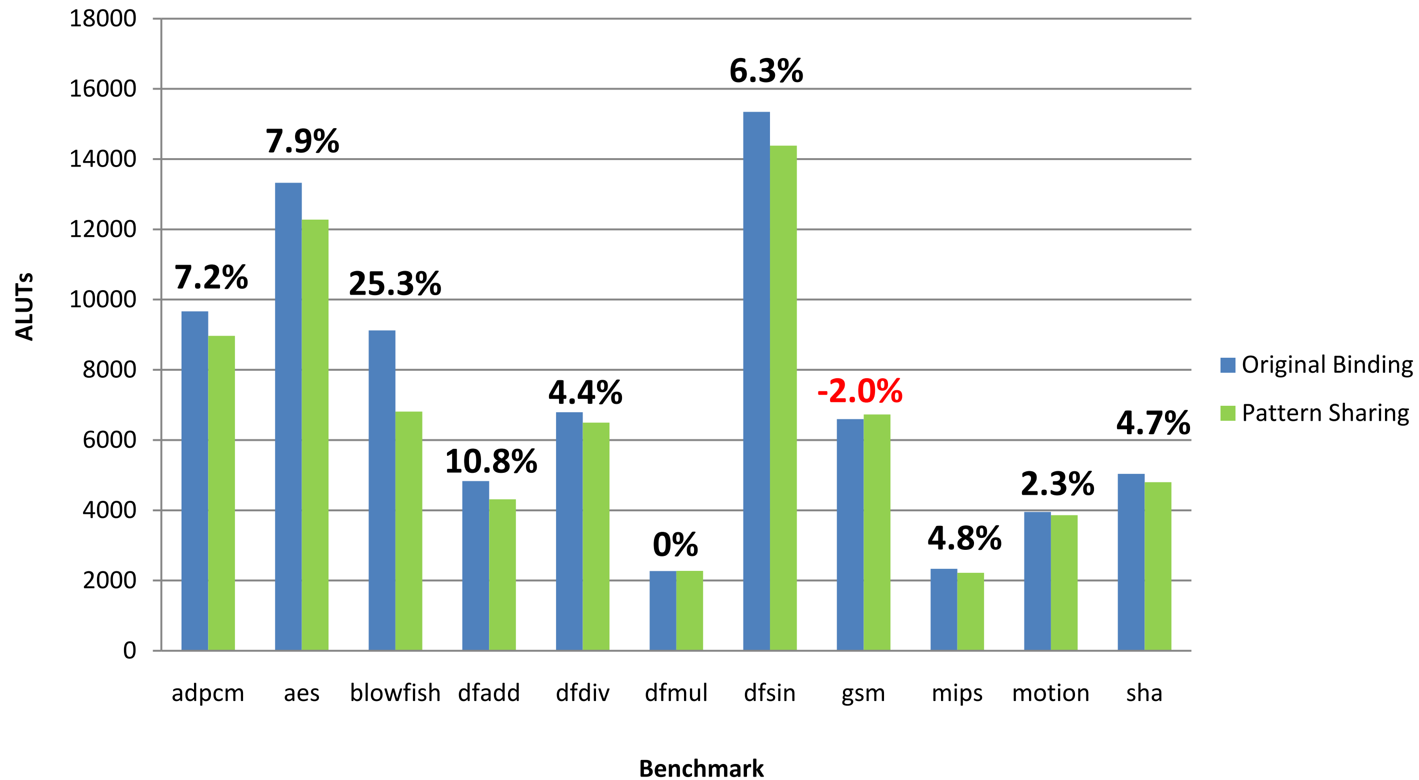


Average: 3.2%

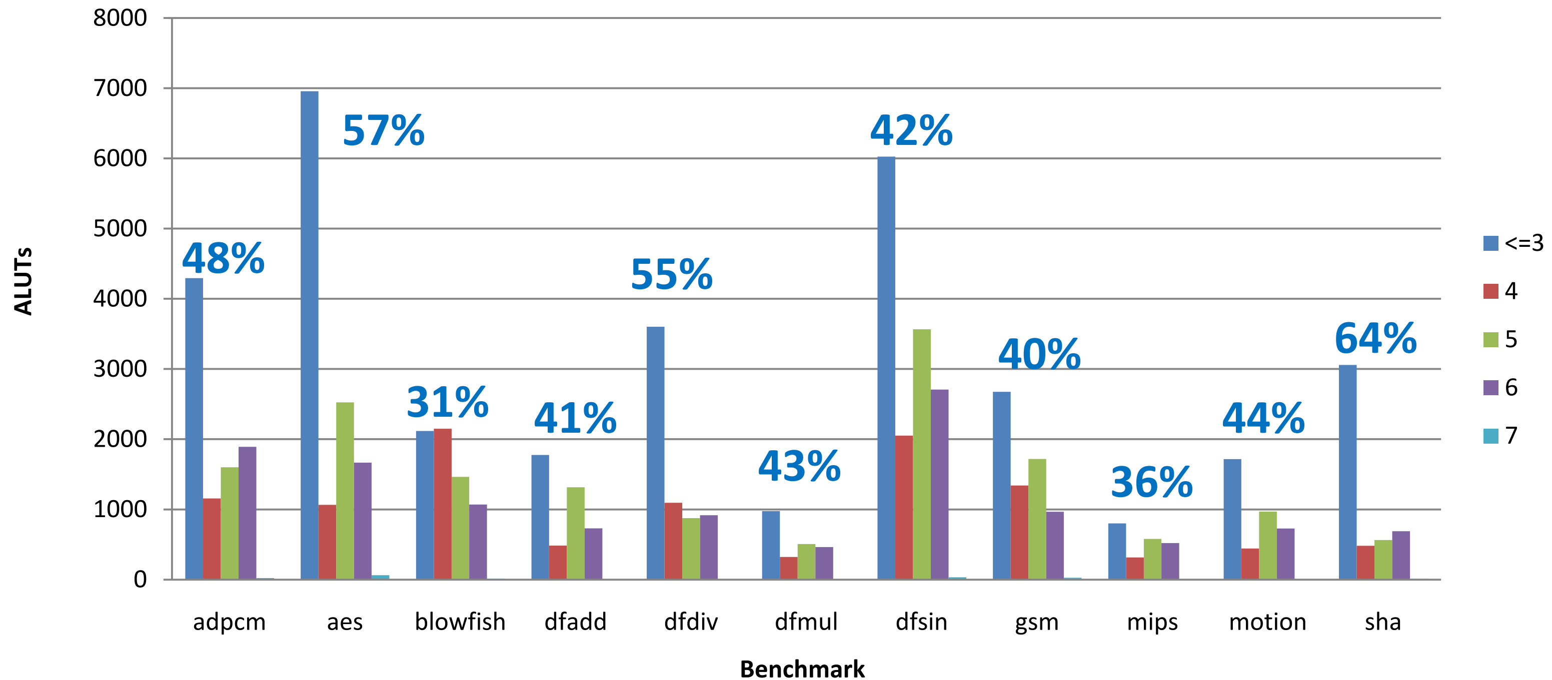
ALMs



Average: 3.2%



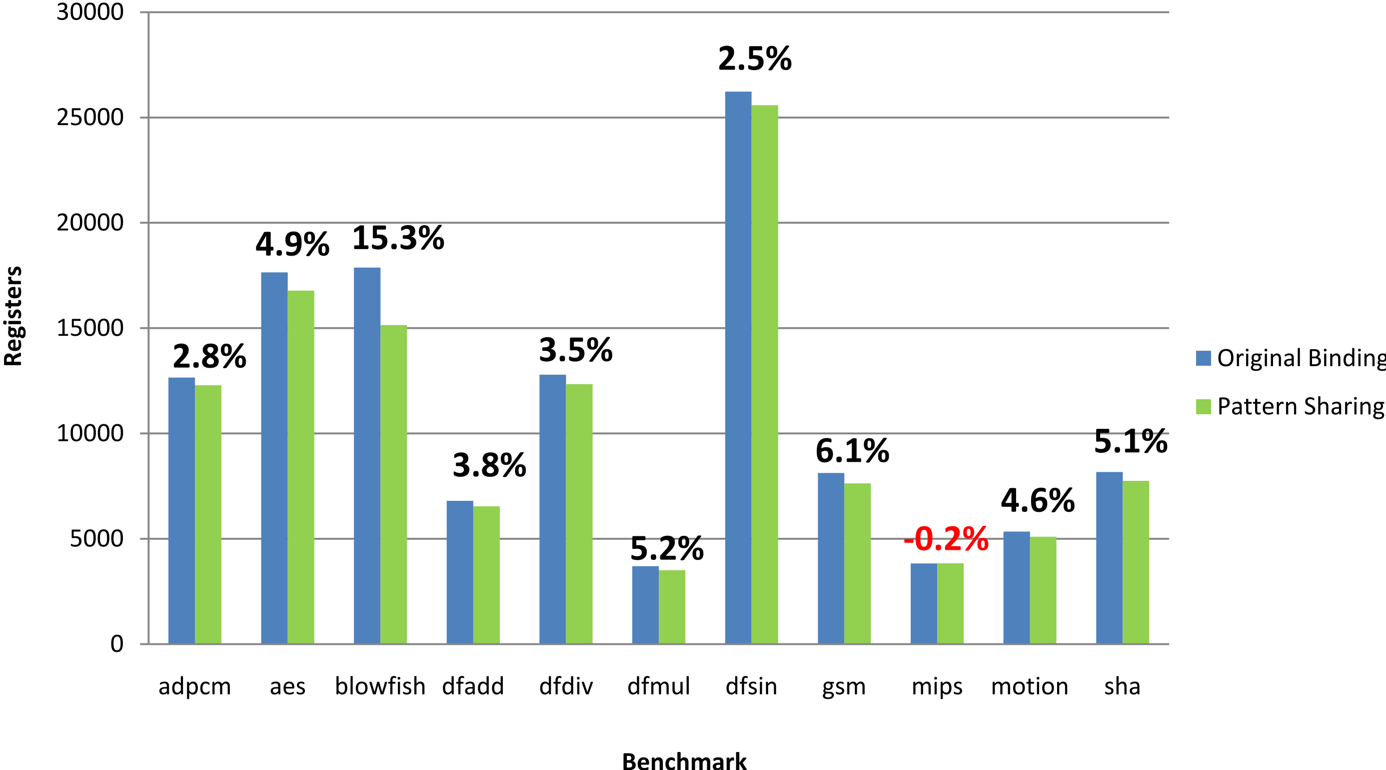
Average: 6.5%



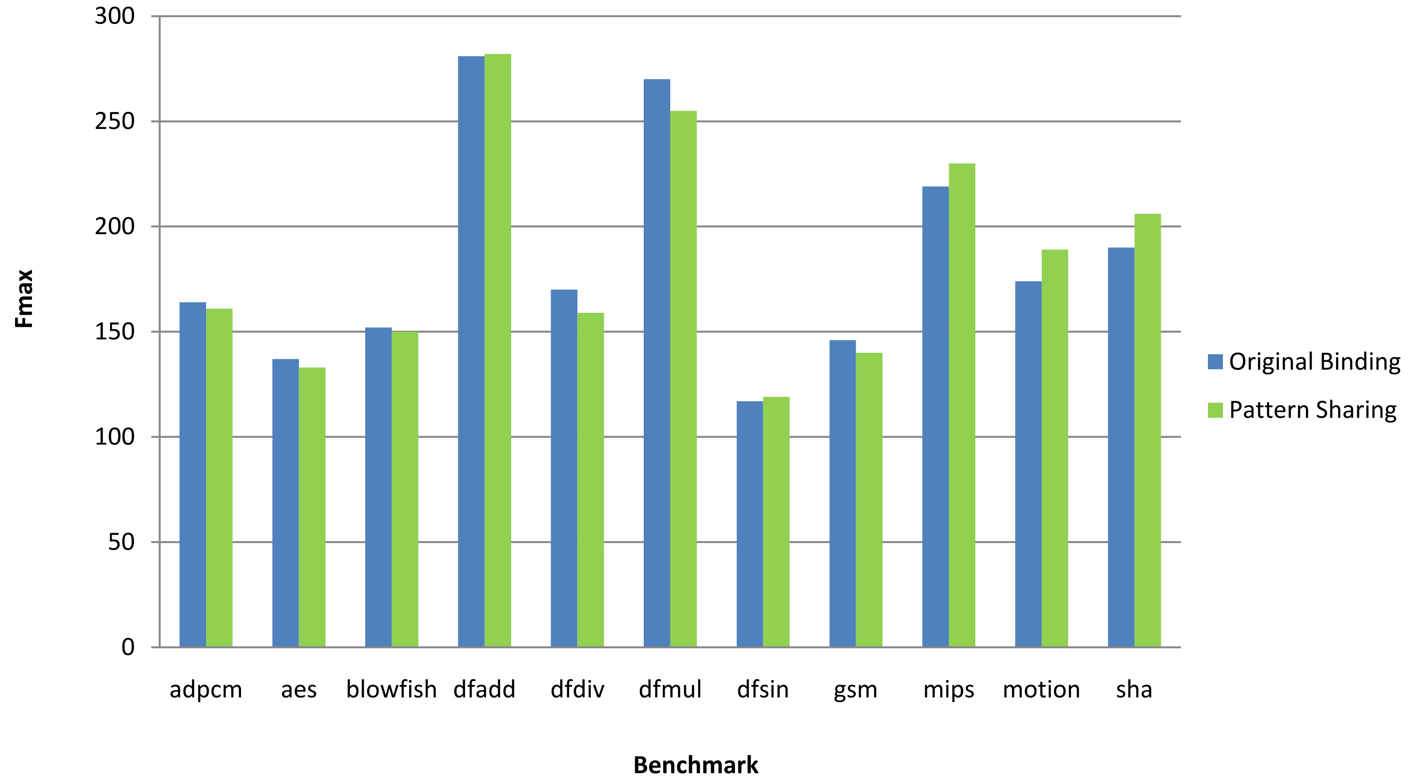
Average: 45%

(was 62%)

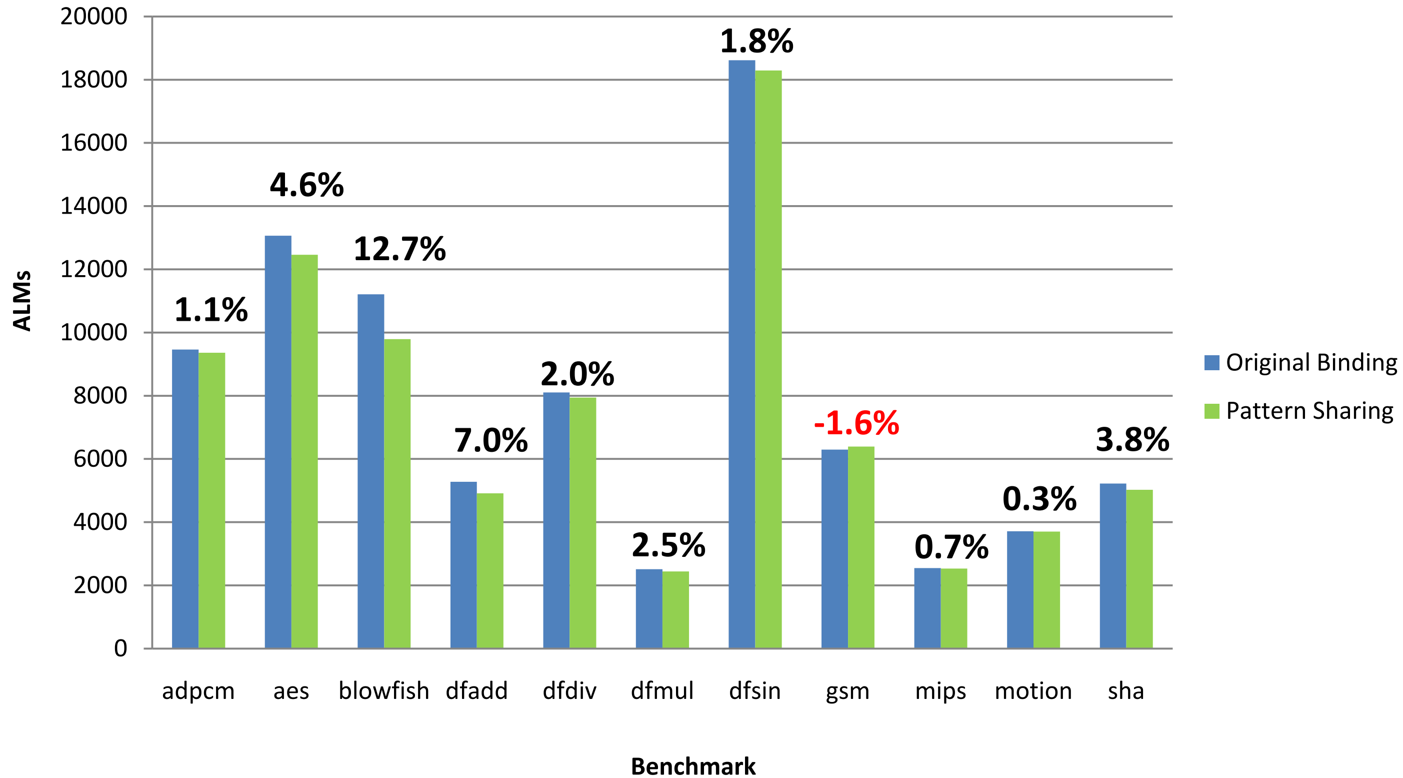
Registers



Average: 4.9%

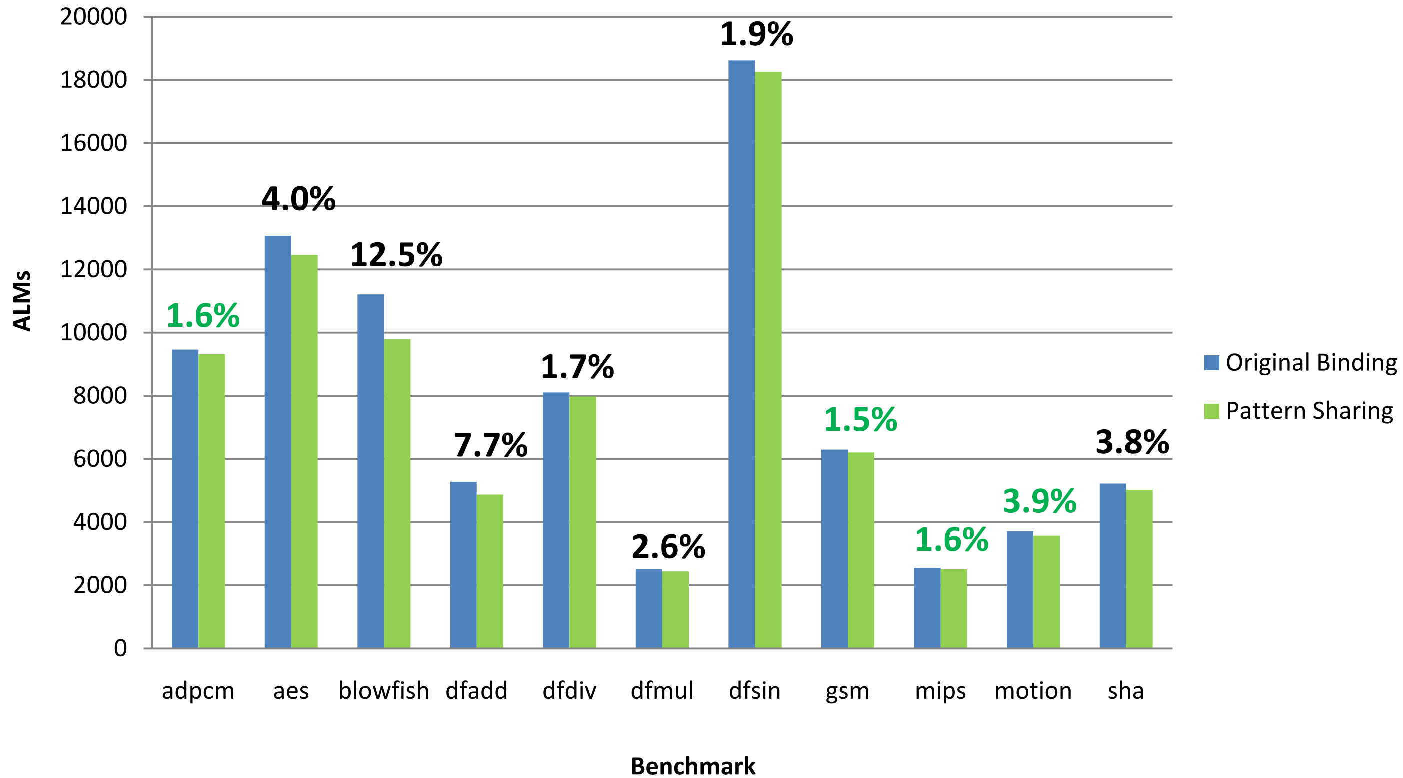

Fmax
300
250
200
150
100
50
0
Fmax
adpcm
aes
blowfish
dfadd
dfdiv
dfmul
dfsin
gsm
mips
motion
sha
Benchmark
Original Binding
Pattern Sharing

ALMs

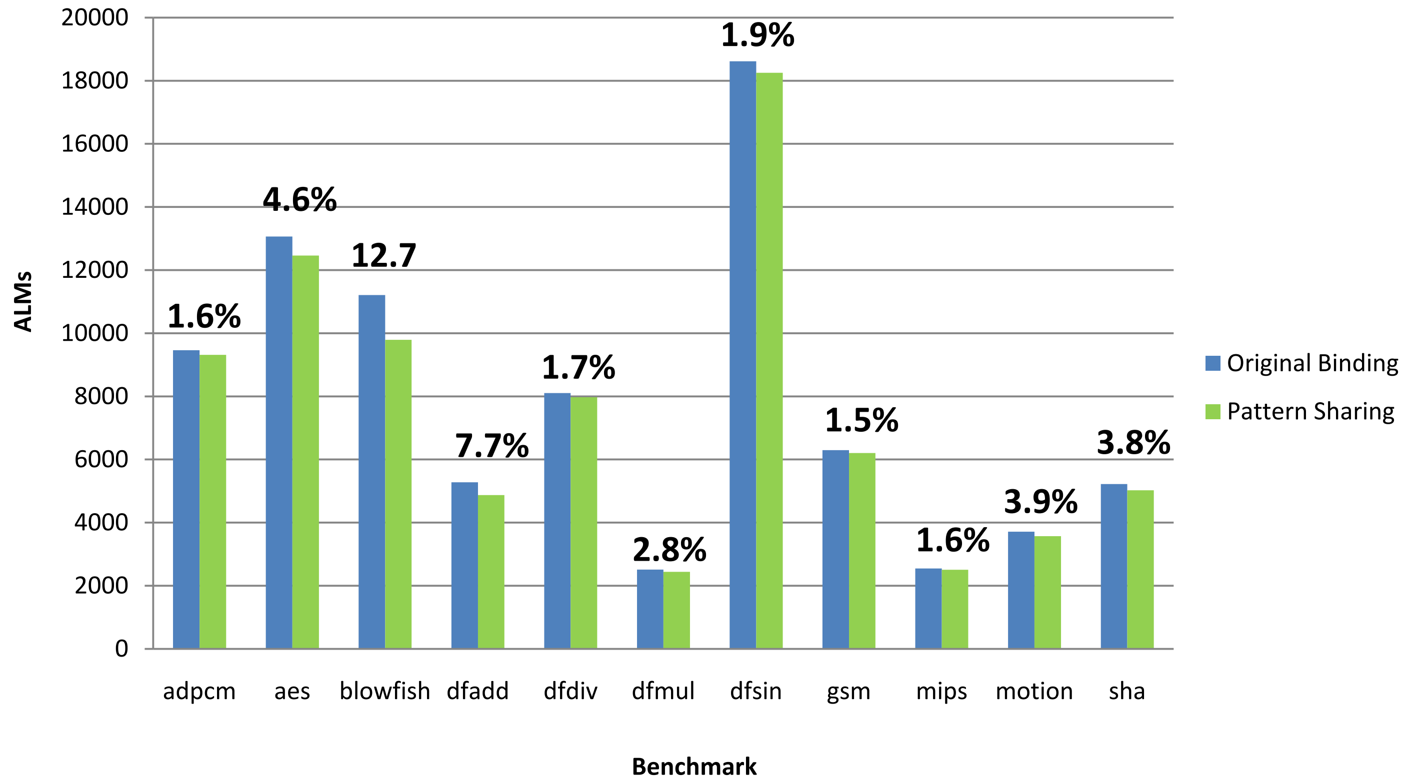


Average: 3.2%

ALMs



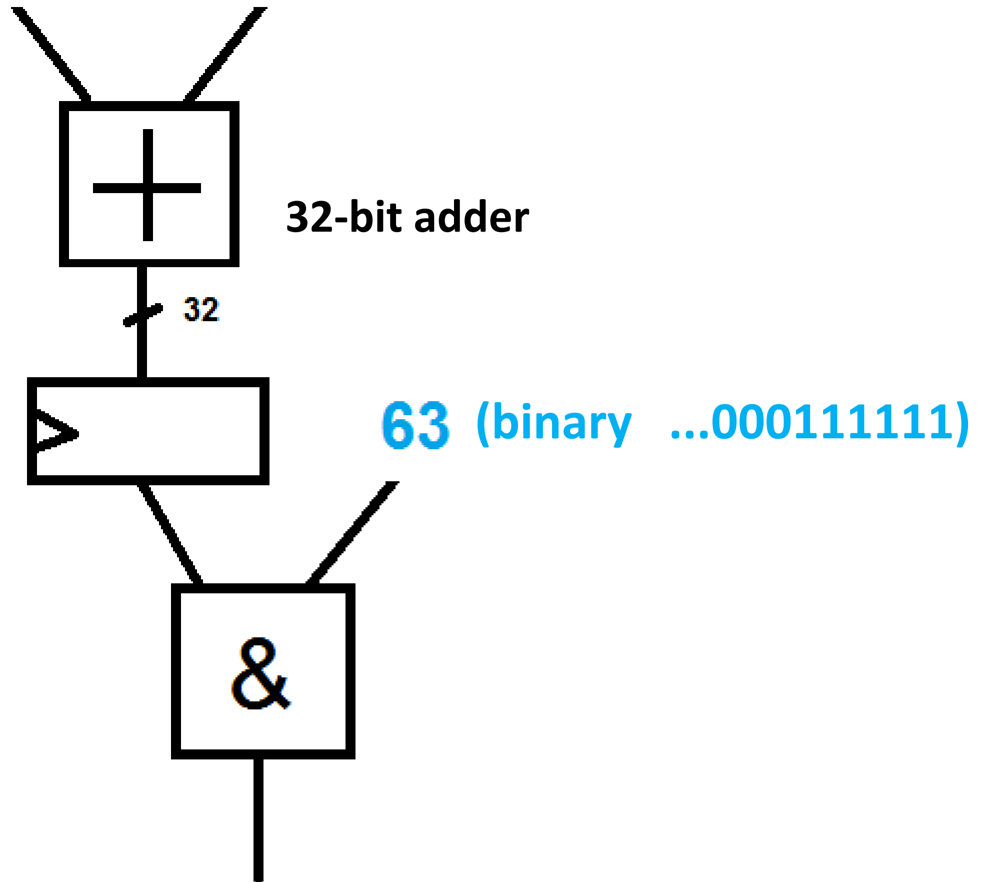
Average: 3.9%



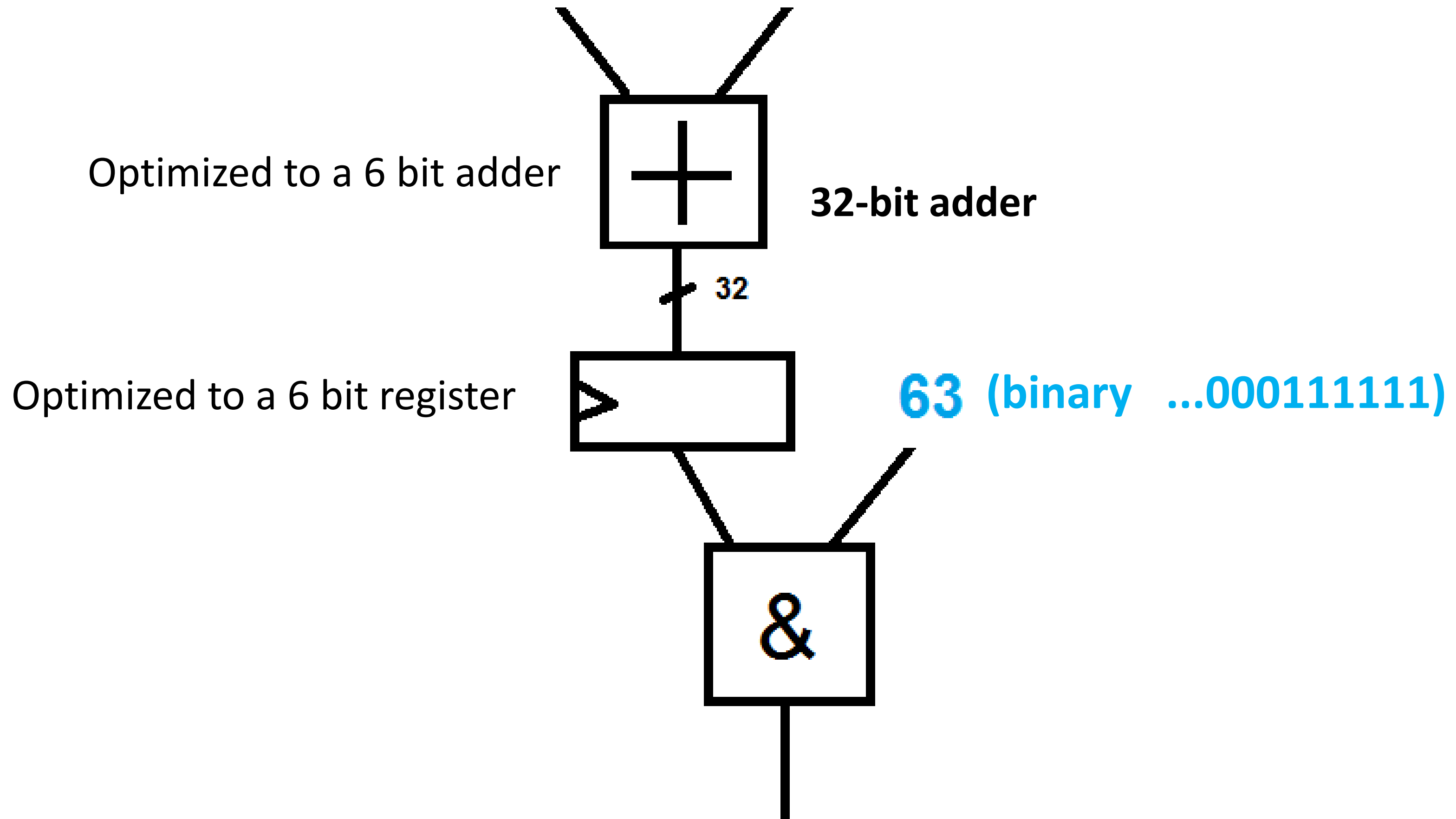
Average: 4.0%

Current Work – Intelligently Pairing Patterns

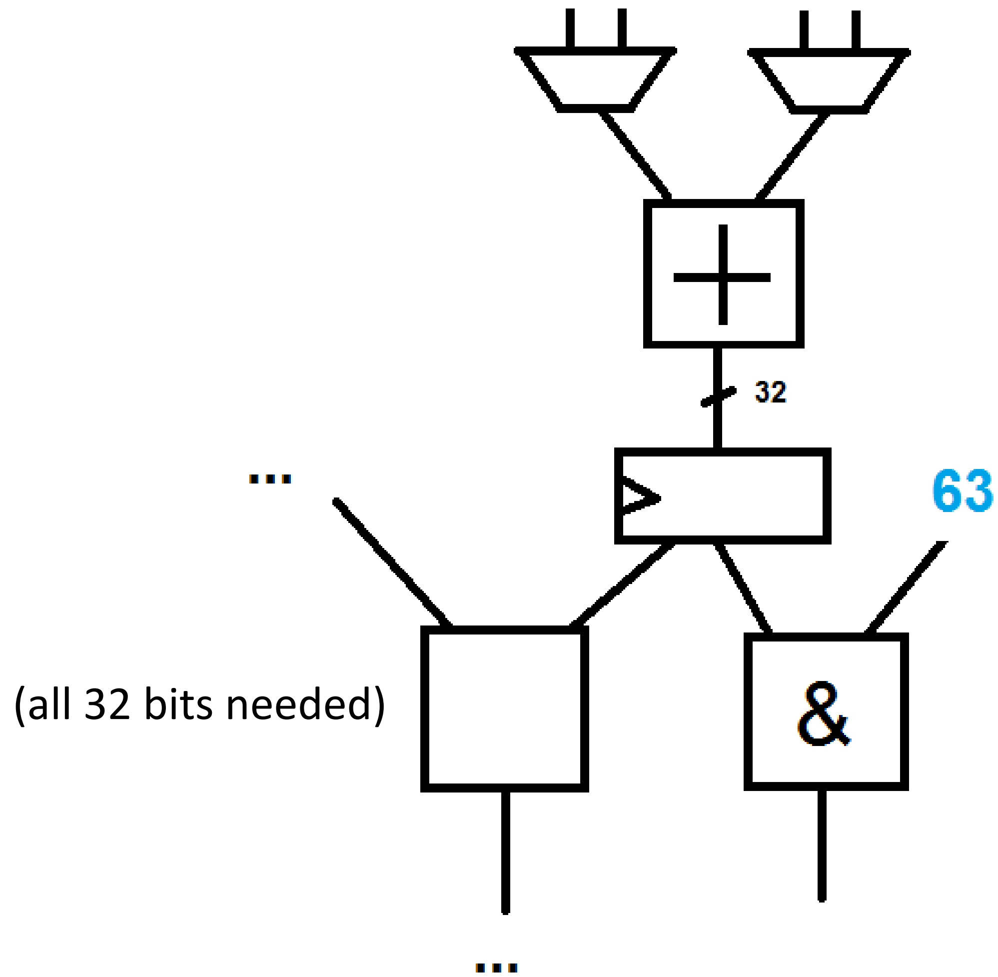
Current Work



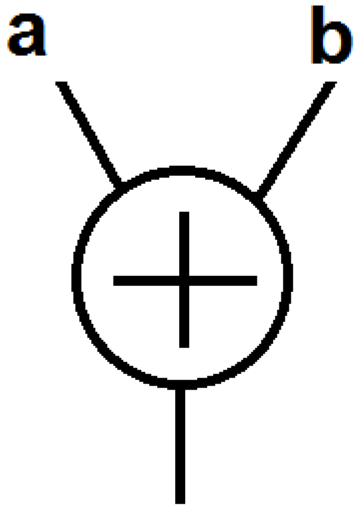
Current Work

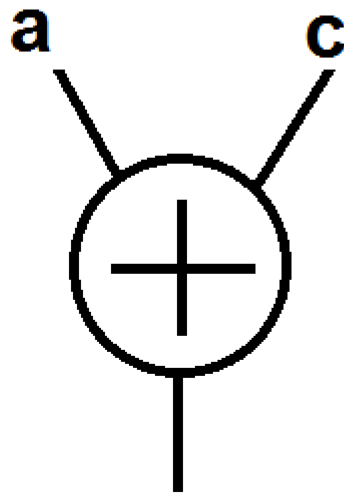


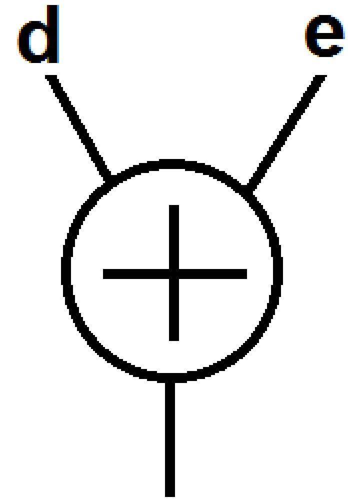
Current Work



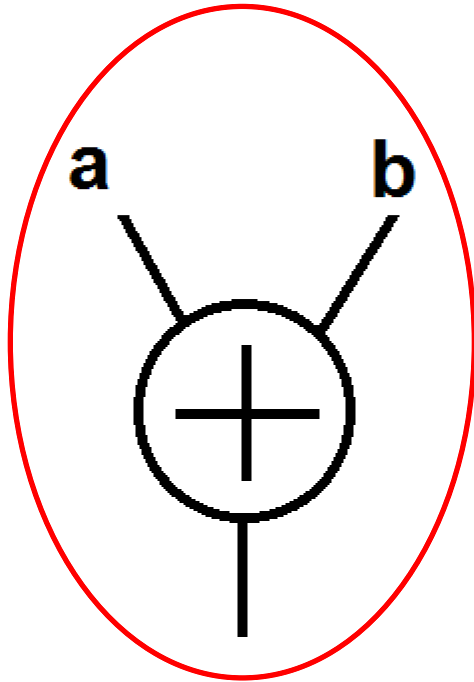
Current Work

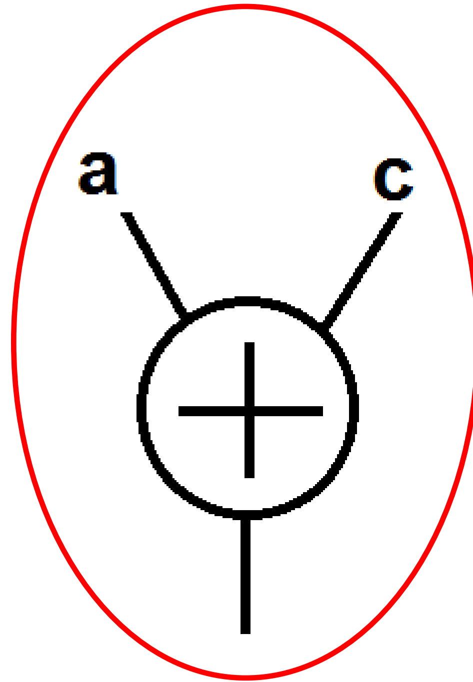


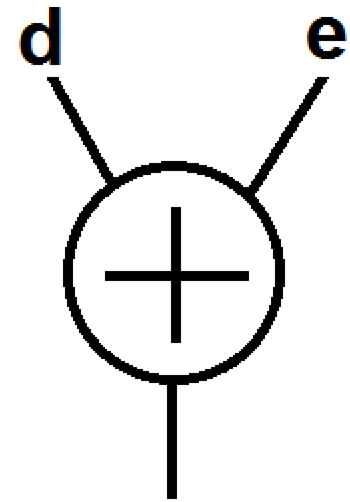




Current Work







Current Work