# Impact of FPGA Architecture on Resource Sharing in High-Level Synthesis

Stefan Hadjis[1], Andrew Canis[1], Jason Anderson[1], Jongsok Choi[1], Kevin Nam[1], Stephen Brown[1], and Tomasz Czajkowski[‡]

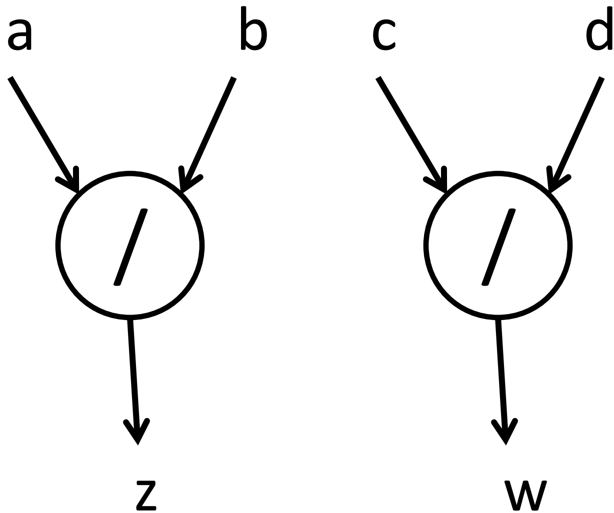
[1]ECE Department, University of Toronto, Toronto, ON, Canada
[‡] Altera Toronto Technology Centre, Toronto, ON, Canada

# Introduction

- High Level Synthesis (HLS) tools are capable of targeting multiple devices and logic element architectures
- E.g. LegUp HLS Tool (www.legup.org)
  - C → Verilog synthesis
  - Targets Cyclone II (4-LUT) and Stratix IV (Adaptive LUT)
- How should HLS be adapted for different target architectures?
- We modify the **Binding** Phase of HLS, in which operations in the high-level circuit specification (C) are assigned to specific functional units in the hardware

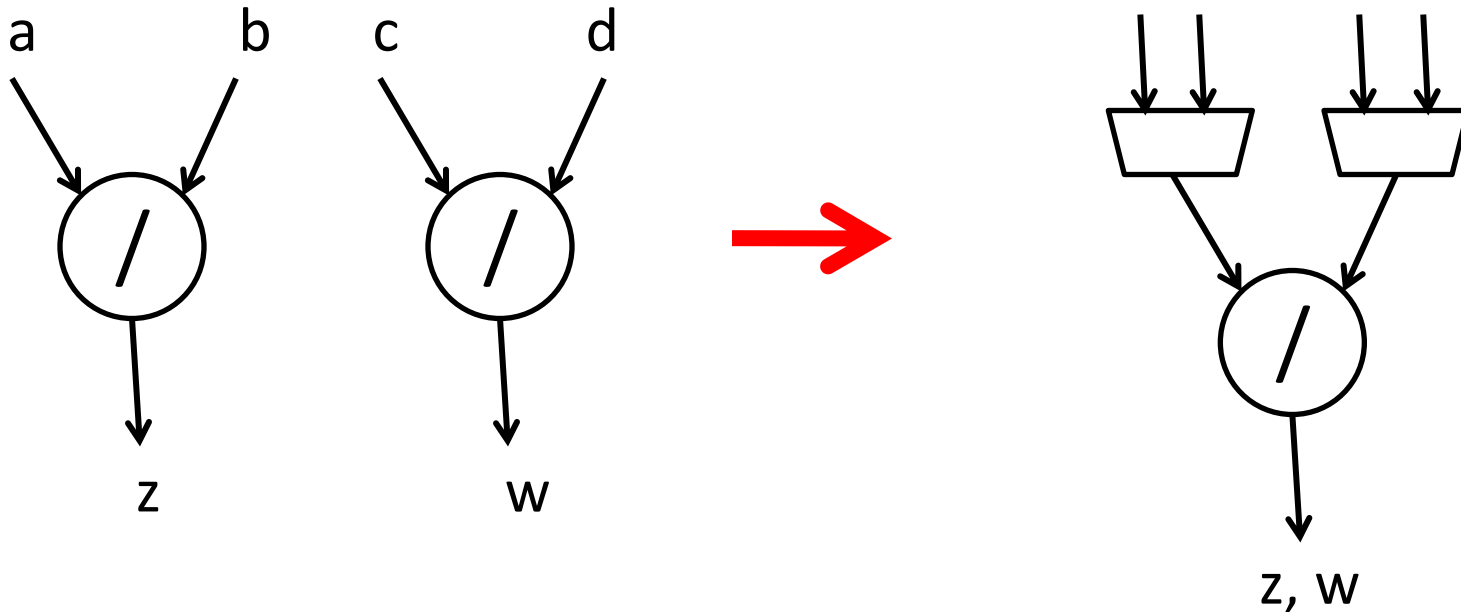
# Resource Sharing in High Level Synthesis

- **Resource Sharing** is an area-reduction optimization in binding which involves assigning multiple operations to the same hardware unit
- E.g. consider a C program which performs division twice:



# Resource Sharing in High Level Synthesis

- **Resource Sharing** is an area-reduction optimization in binding which involves assigning multiple operations to the same hardware unit
- E.g. consider a C program which performs division twice:



# Resource Sharing in High Level Synthesis

- **Resource Sharing** is an area-reduction optimization in binding which involves assigning multiple operations to the same hardware unit
- Different resource sharing tradeoffs exist depending on the target architecture

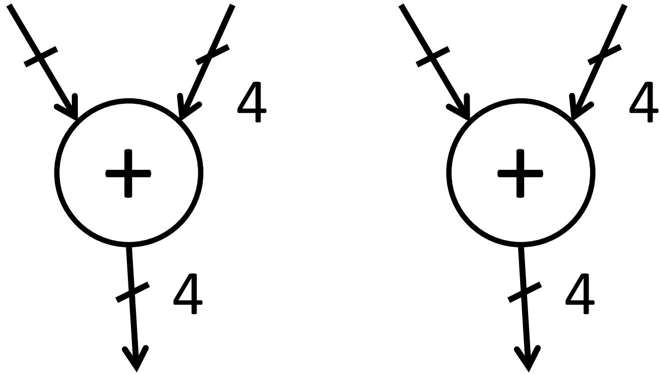
# Example: 4 Bit Adder

# Example: 4 Bit Adder

- Consider a C program which performs two additions

# Example: 4 Bit Adder

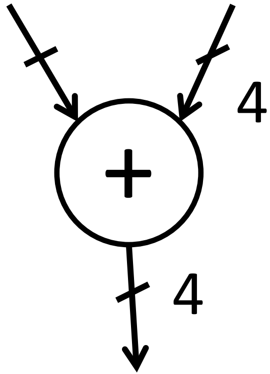
- Consider a C program which performs two additions
- Which hardware implementation is preferred?

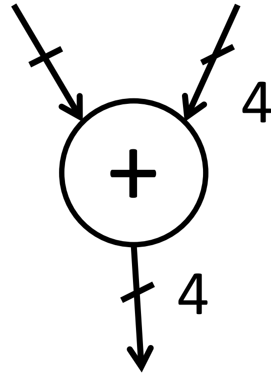



**METHOD 1: NOT SHARING**

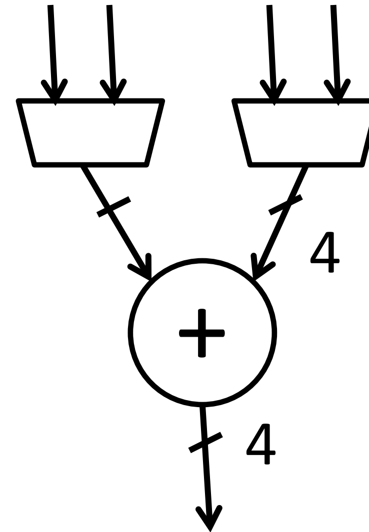
# Example: 4 Bit Adder

- Consider a C program which performs two additions
- Which hardware implementation is preferred?




**VS.**

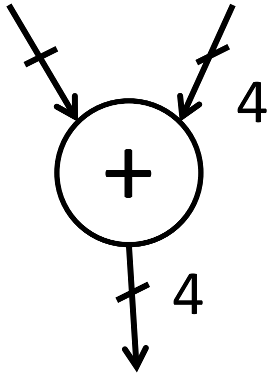


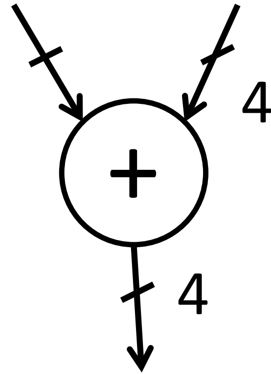

**METHOD 1: NOT SHARING**

**METHOD 2: SHARING**

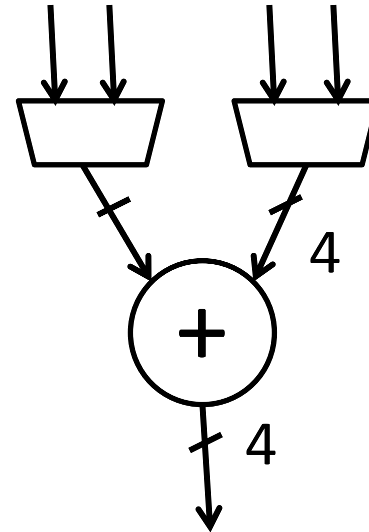
# Example: 4 Bit Adder

- Consider a C program which performs two additions
- Which hardware implementation is preferred?




VS.

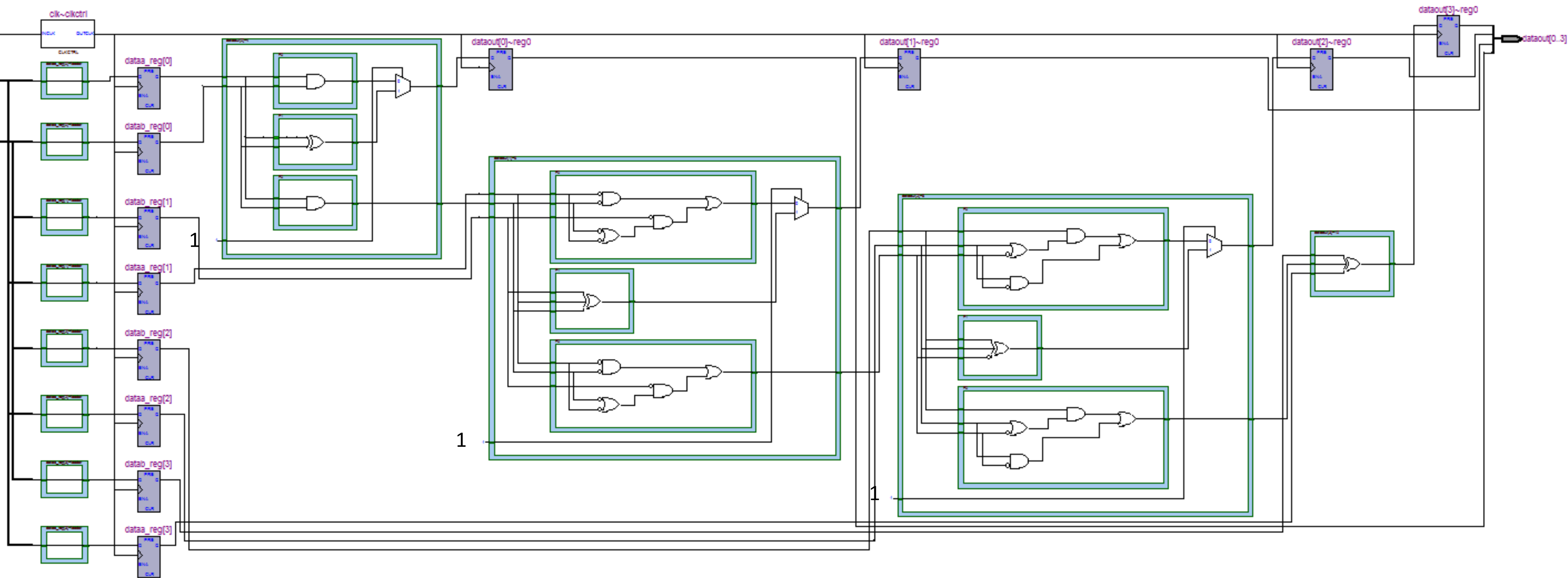



**METHOD 1: NOT SHARING**

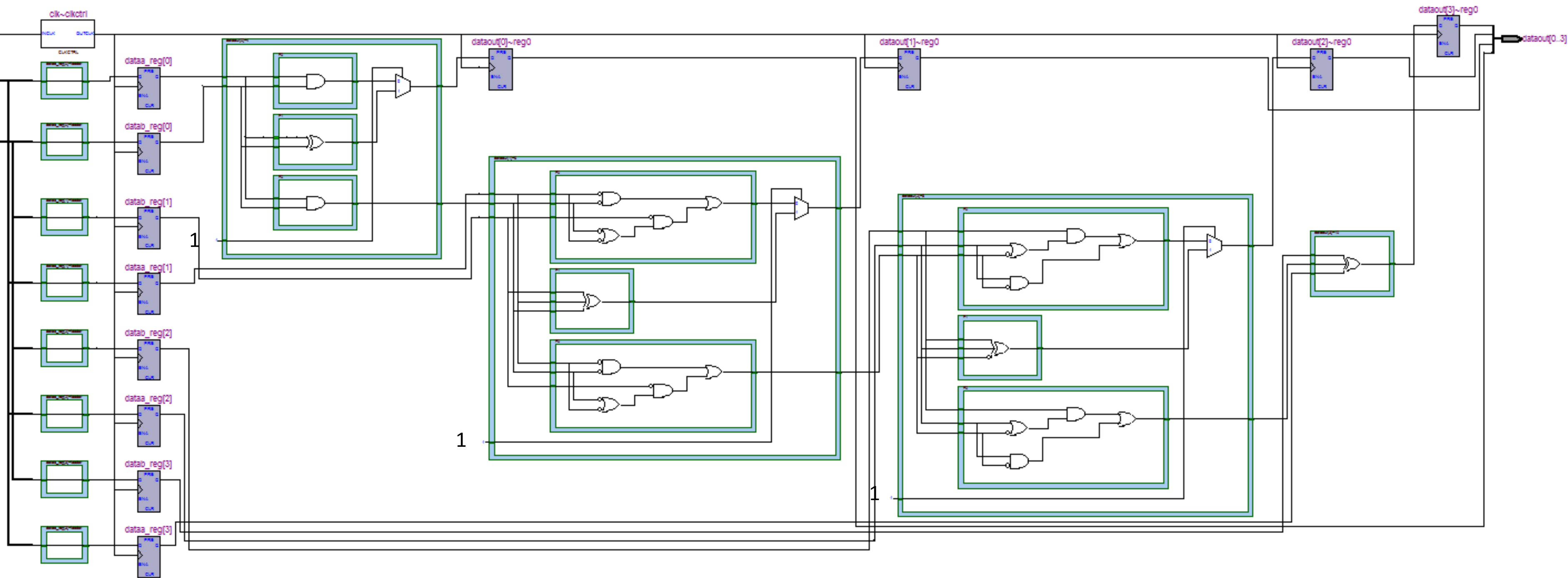
**METHOD 2: SHARING**

- The answer depends on the architecture

# METHOD 1: NOT SHARING (a single 4-bit adder)

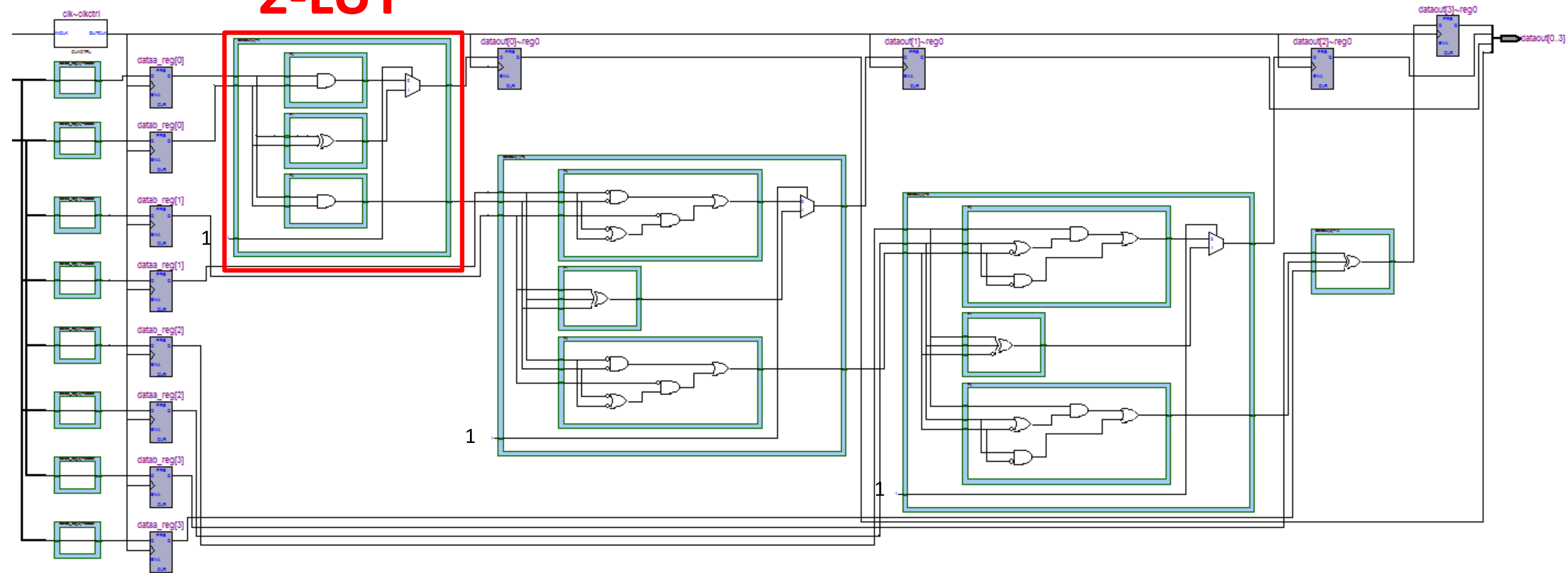


# METHOD 1: NOT SHARING (a single 4-bit adder)



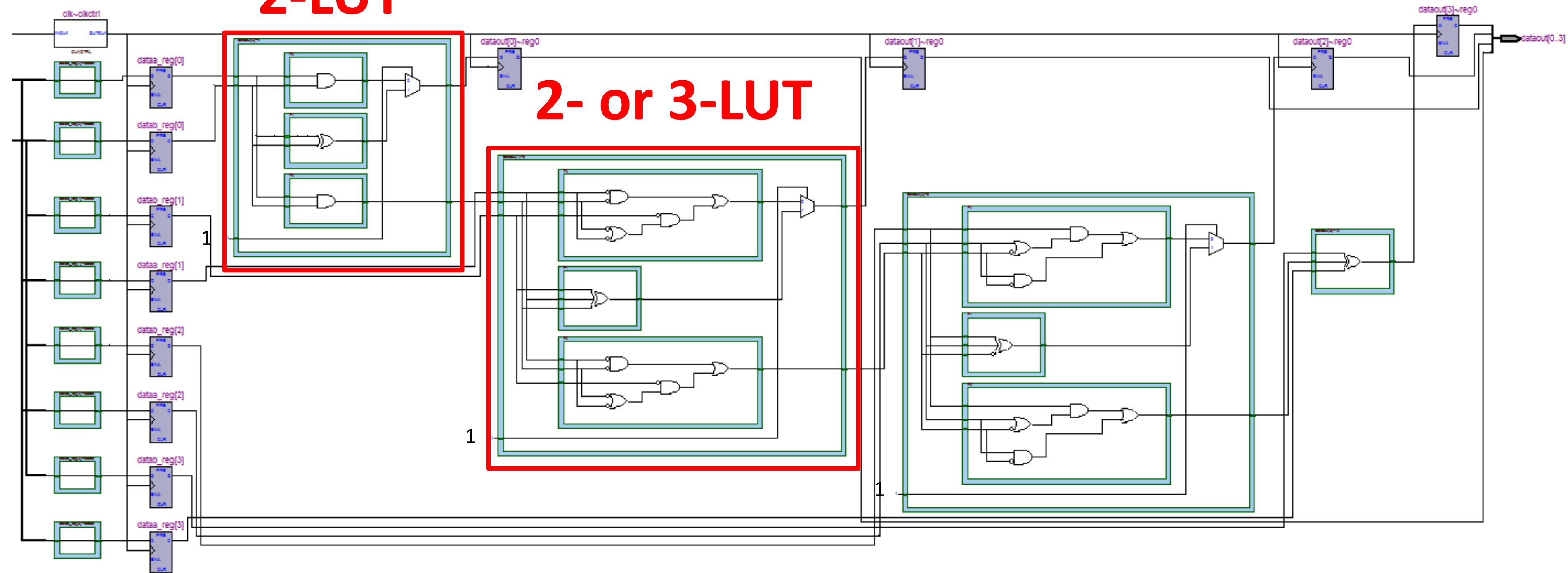

- Ripple Carry implementation

# METHOD 1: NOT SHARING (a single 4-bit adder)



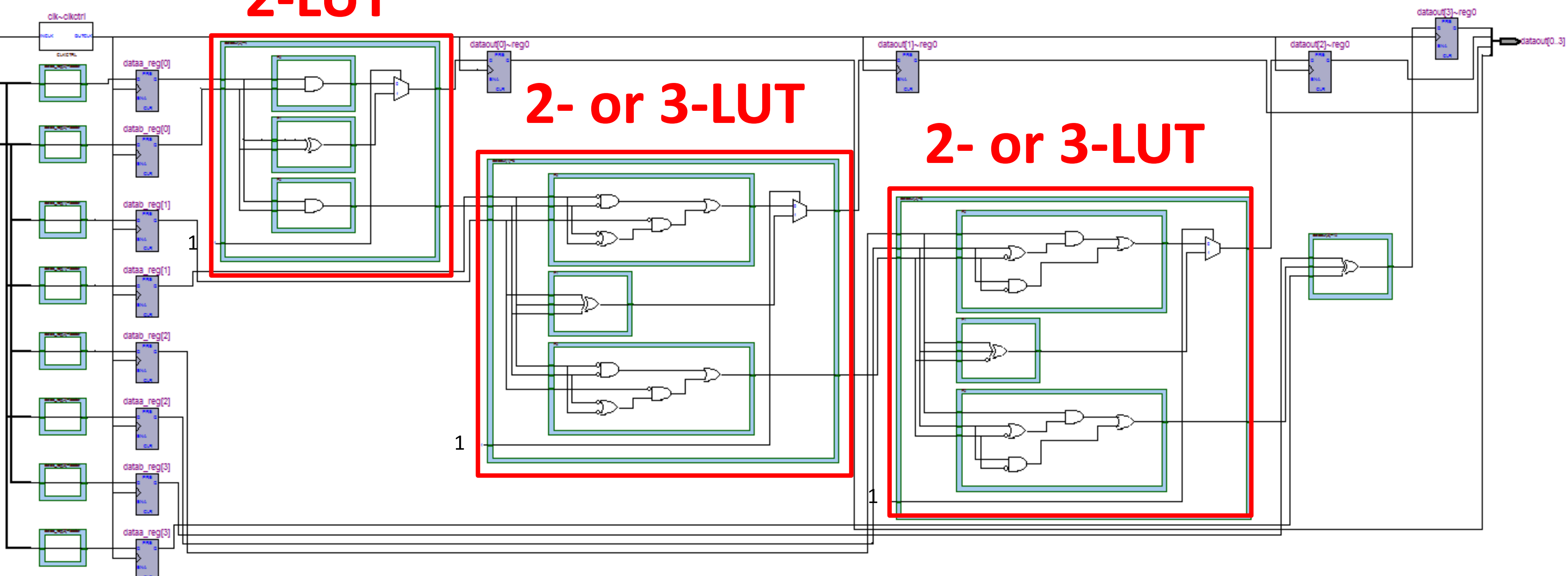

- Ripple Carry implementation

# METHOD 1: NOT SHARING (a single 4-bit adder)



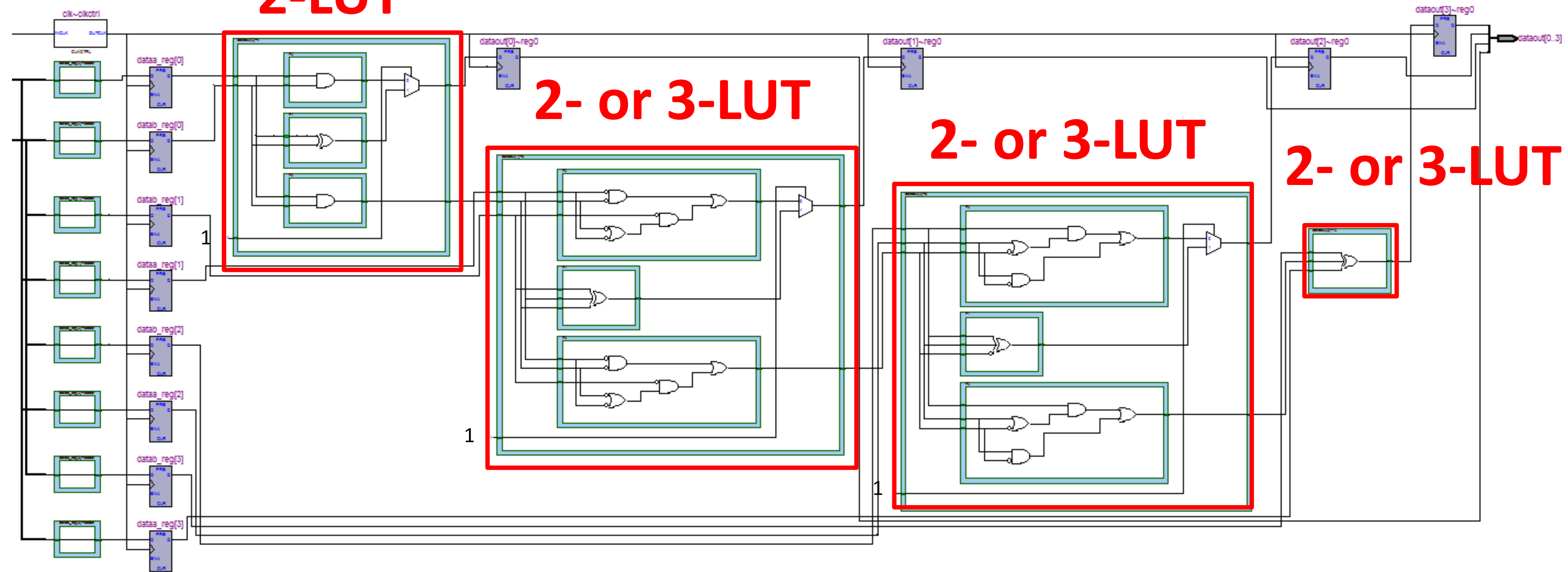

- Ripple Carry implementation

# METHOD 1: NOT SHARING (a single 4-bit adder)




- Ripple Carry implementation

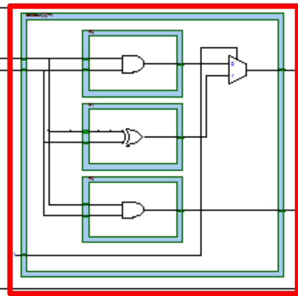
# METHOD 1: NOT SHARING (a single 4-bit adder)



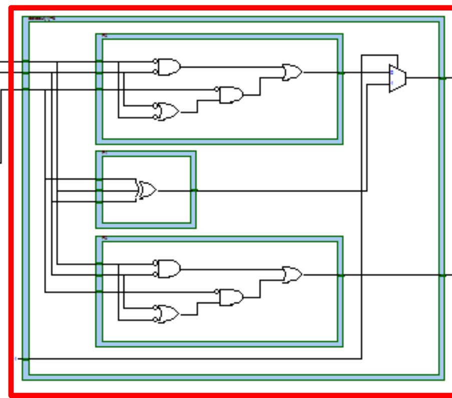

- Ripple Carry implementation

# METHOD 1: NOT SHARING (a single 4-bit adder)

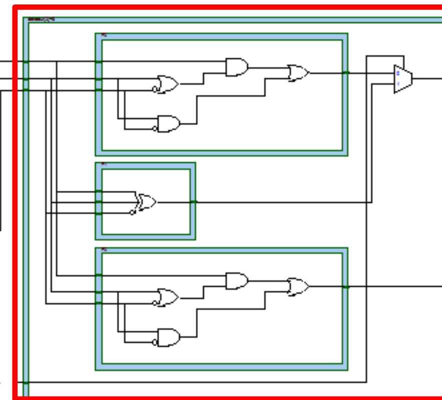
2-LUT



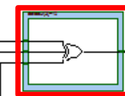
2- or 3-LUT



2- or 3-LUT

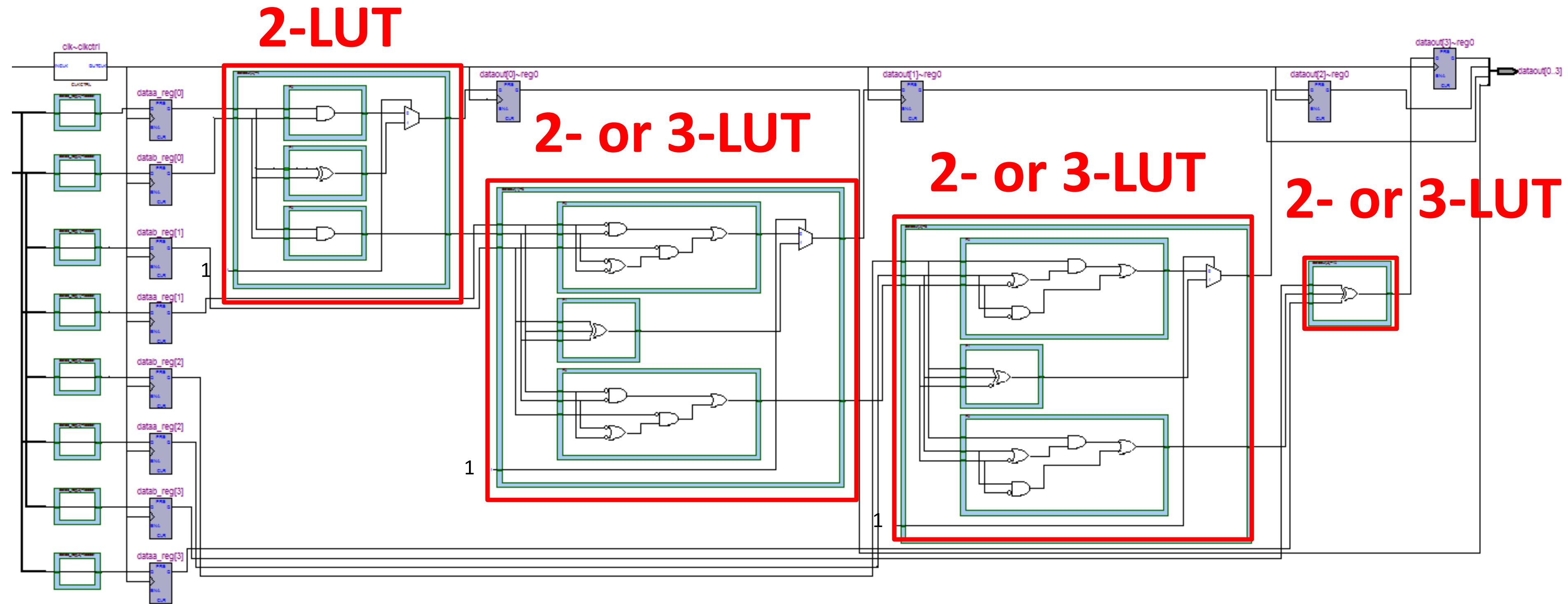


2- or 3-LUT



- Ripple Carry implementation: Four LUTs, all with 2-3 inputs

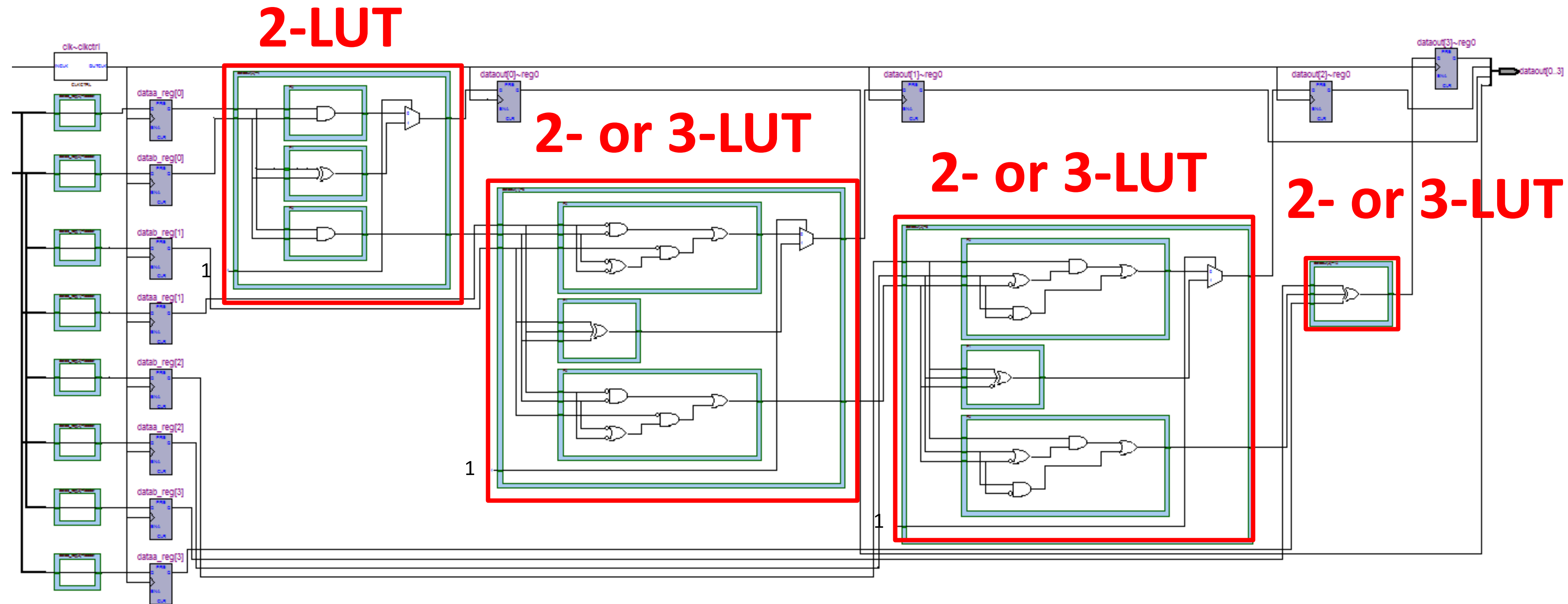
# METHOD 1: NOT SHARING (a single 4-bit adder)




- Ripple Carry implementation: Four LUTs, all with 2-3 inputs

| | |
|---|---|
| **Cyclone II** | **4 LEs** |
| **Stratix IV** | **2 ALMs** |

# METHOD 1: NOT SHARING (a single 4-bit adder)



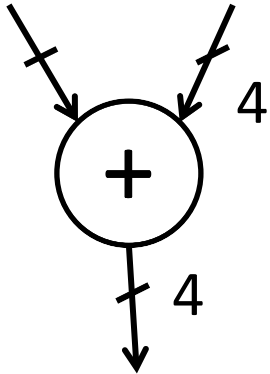

- Ripple Carry implementation: Four LUTs, all with 2-3 inputs

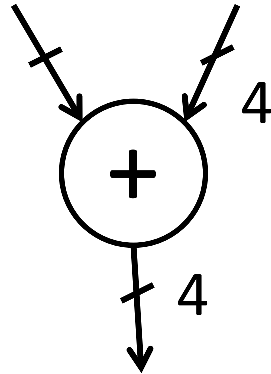
For <u>TWO</u> adders:

| **Cyclone II** | **8 LEs** |
|---|---|
| **Stratix IV** | **4 ALMs** |

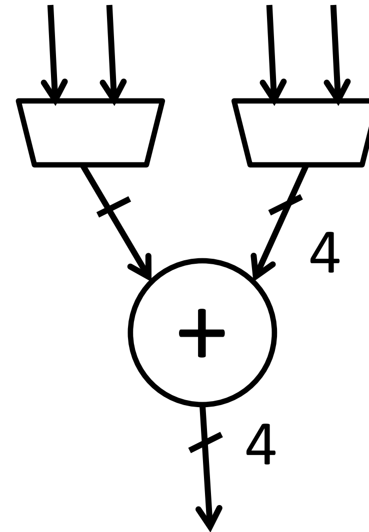
# Example: 4 Bit Adder

- Consider a C program which performs two additions
- Which hardware implementation is preferred?




**VS.**



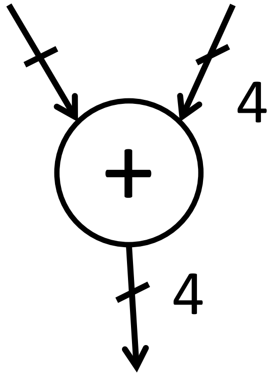

**METHOD 1: NOT SHARING**

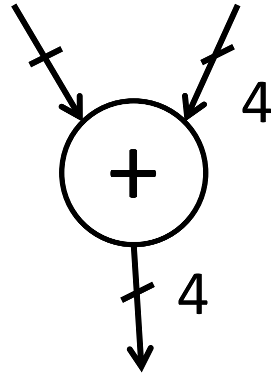
**METHOD 2: SHARING**

| | |
|---|---|
| **Cyclone II** | **8 LEs** |
| **Stratix IV** | **4 ALMs** |

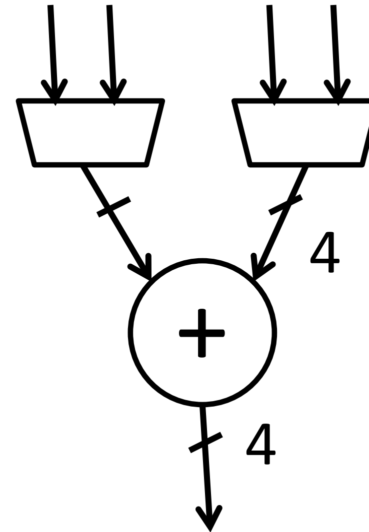
# Example: 4 Bit Adder

- Consider a C program which performs two additions
- Which hardware implementation is preferred?




**VS.**



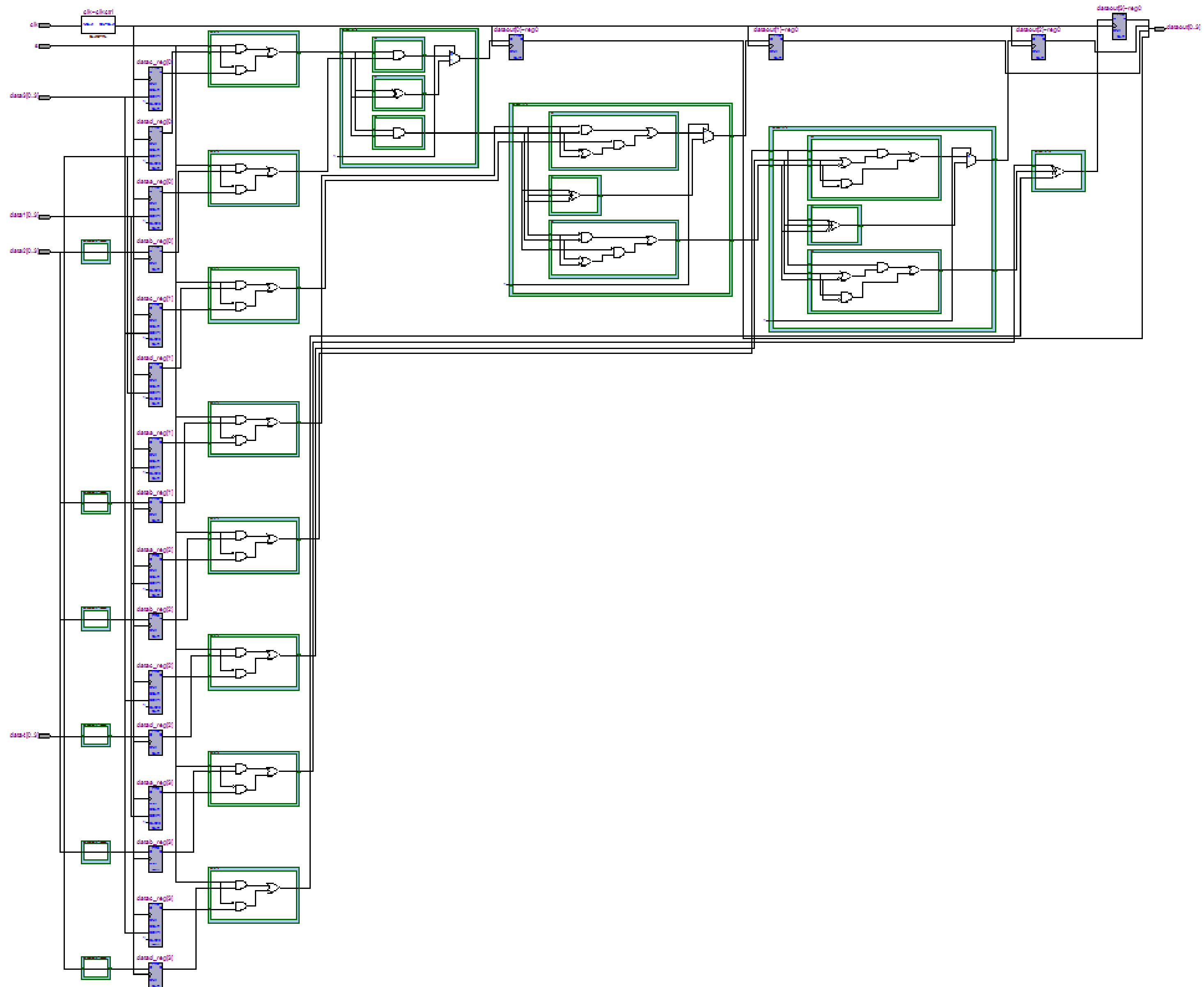

**METHOD 1: NOT SHARING**

**METHOD 2: SHARING**

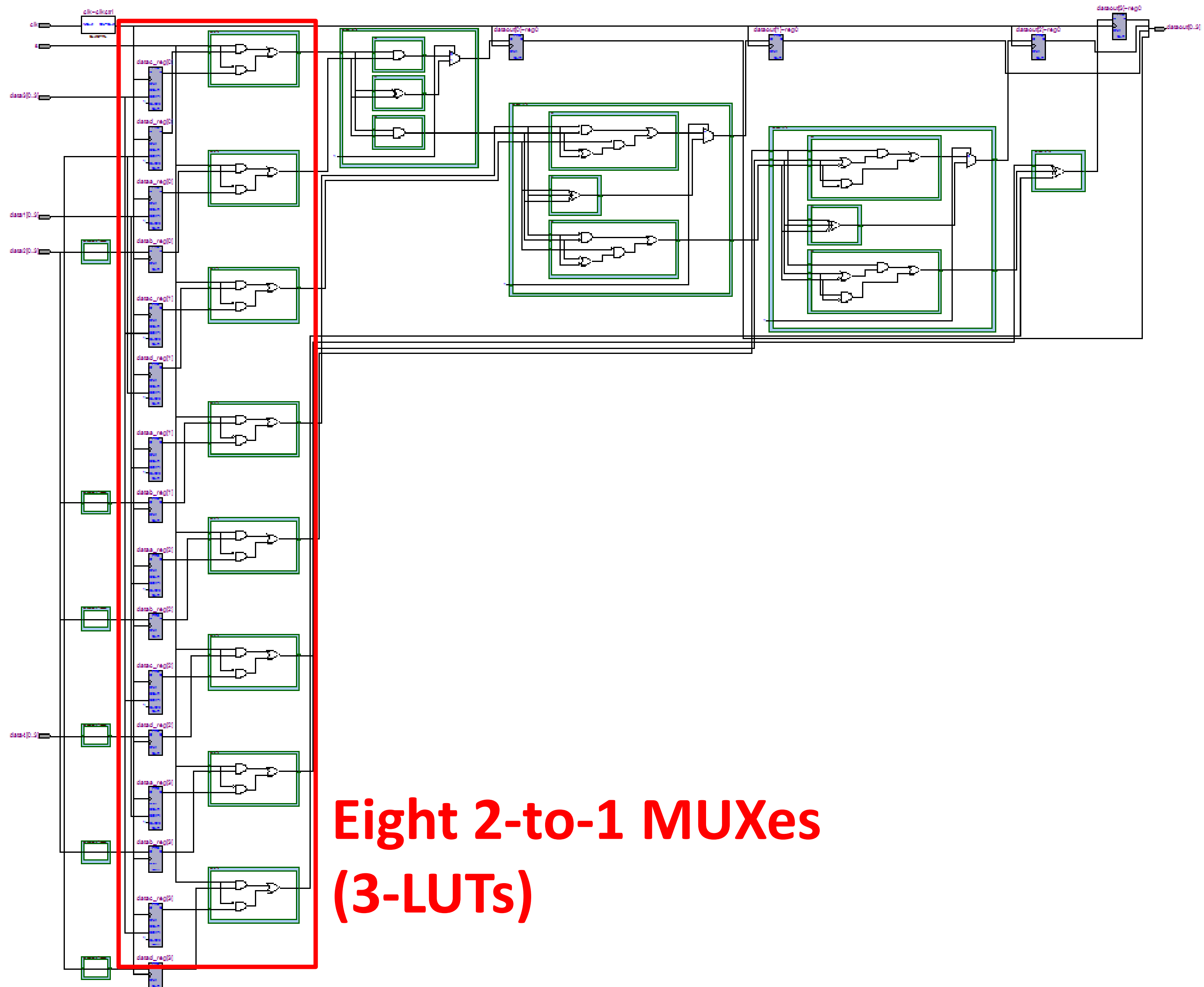
| Cyclone II | 8 LEs |
|---|---|
| Stratix IV | 4 ALMs |

| Cyclone II | ?? LEs |
|---|---|

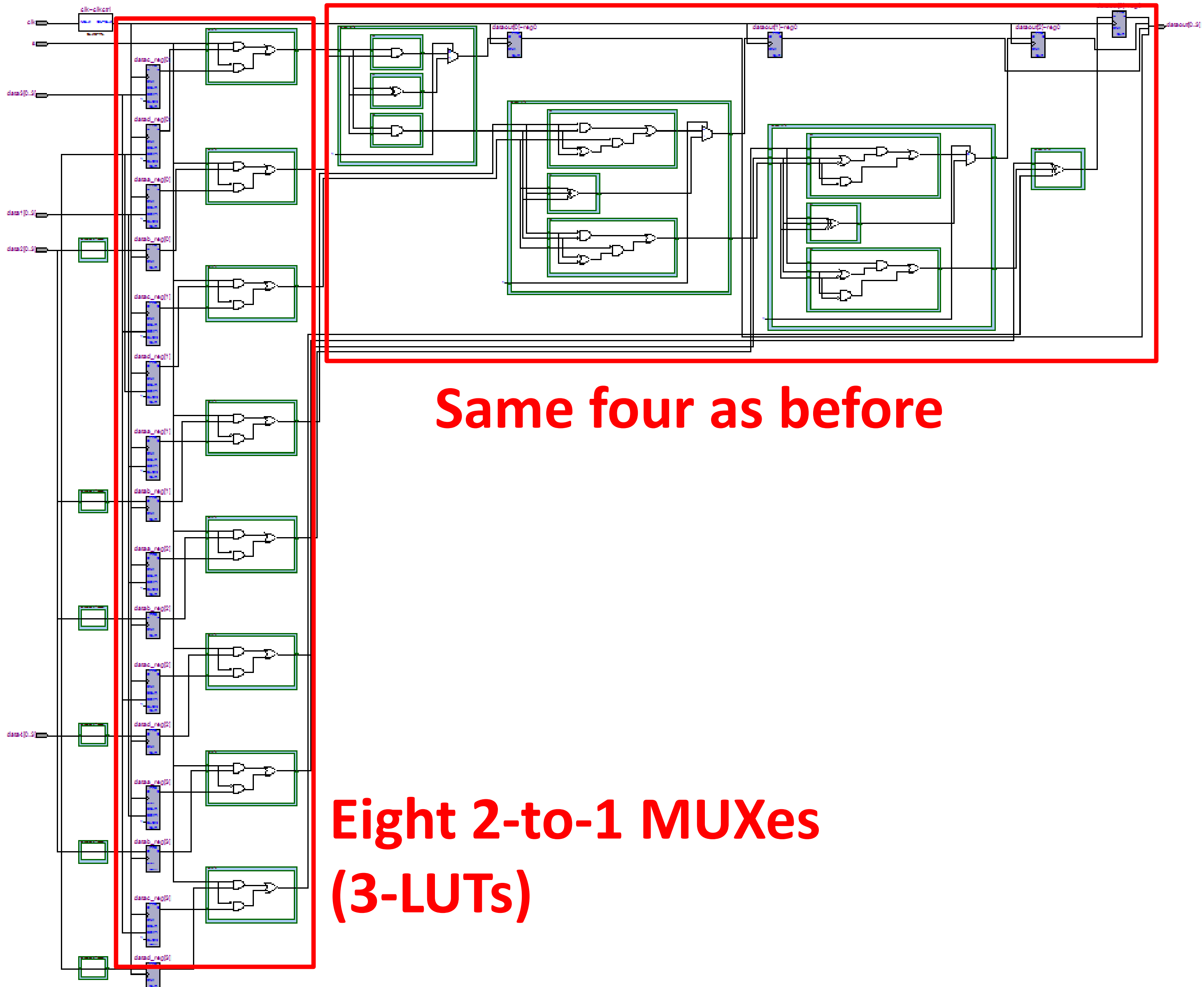
# METHOD 2: SHARING (Cyclone II)



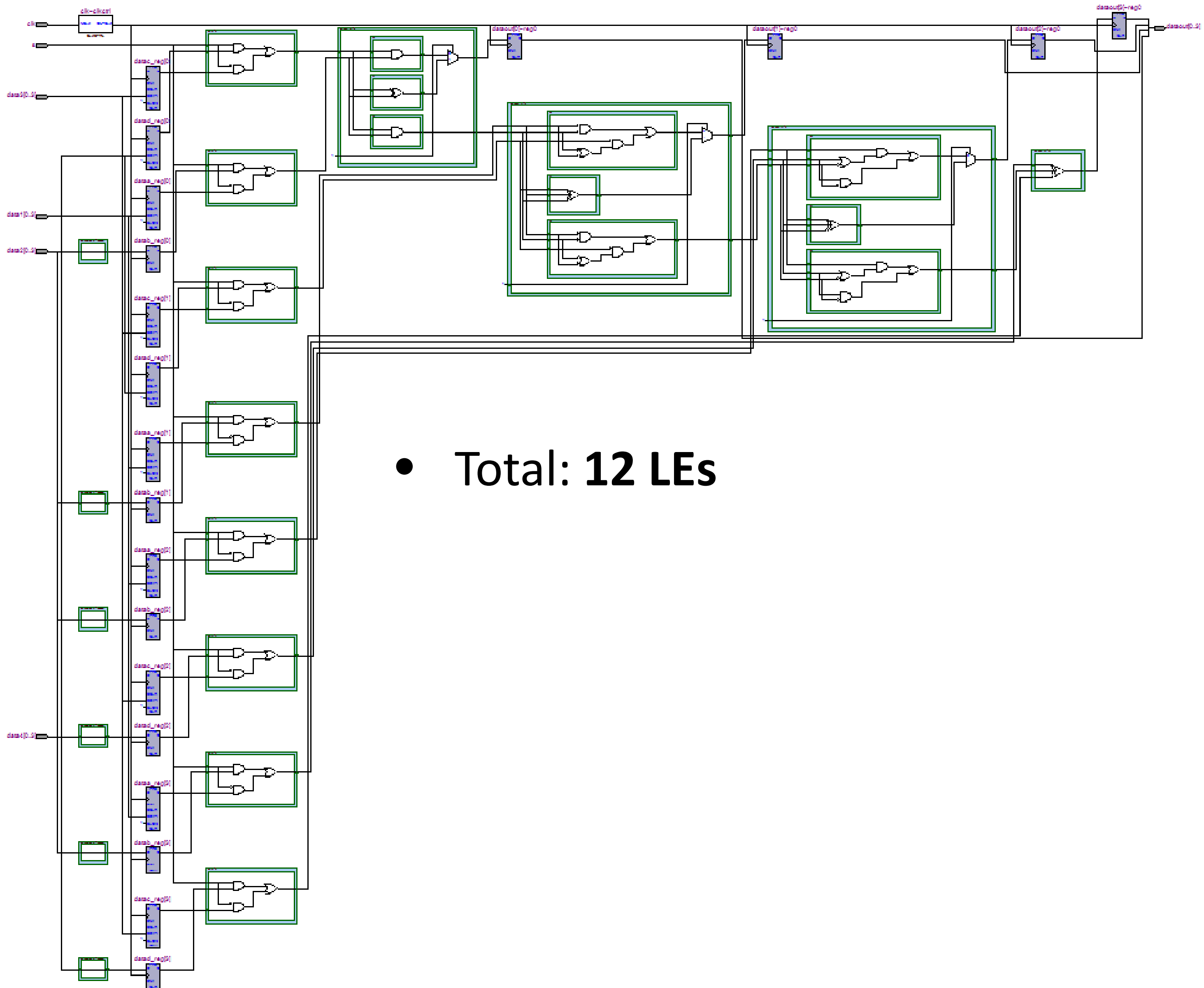
# METHOD 2: SHARING (Cyclone II)



# METHOD 2: SHARING (Cyclone II)



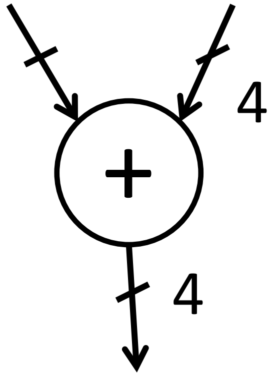
# METHOD 2: SHARING (Cyclone II)

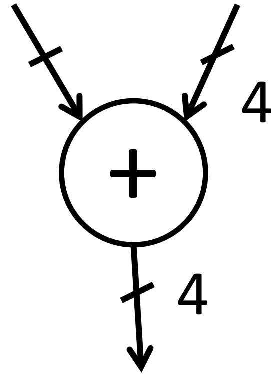


- Total: **12 LEs**

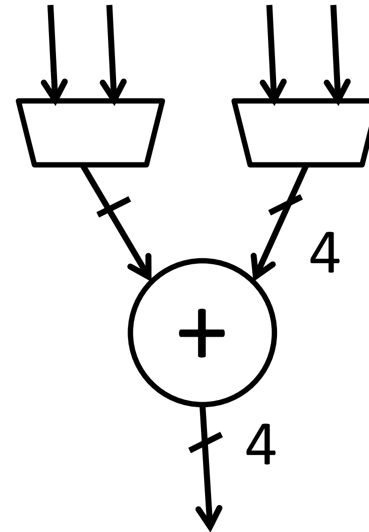
# Example: 4 Bit Adder

- Consider a C program which performs two additions
- Which hardware implementation is preferred?




**VS.**




**METHOD 1: NOT SHARING**

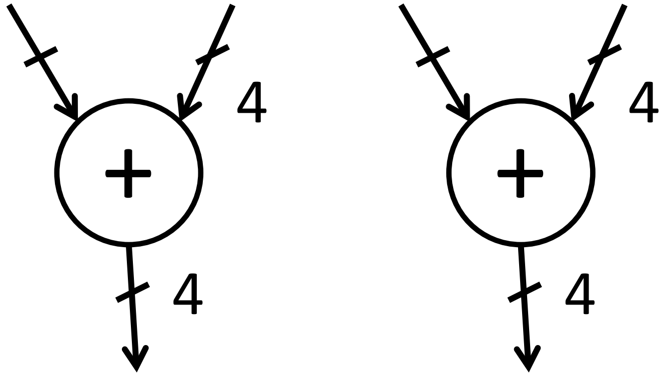
**METHOD 2: SHARING**

| | |
|---|---|
| **Cyclone II** | **8 LEs** |
| **Stratix IV** | **4 ALMs** |

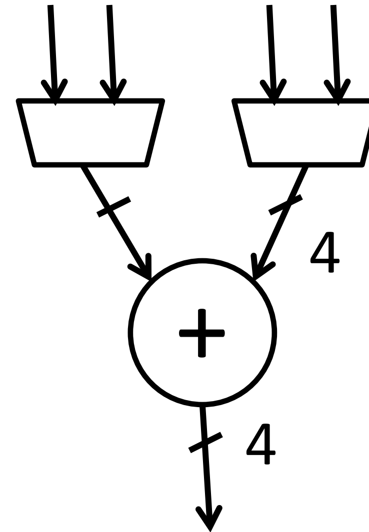
| | |
|---|---|
| **Cyclone II** | **12 LEs** |

# Example: 4 Bit Adder

- Consider a C program which performs two additions
- Which hardware implementation is preferred?




**VS.**




**METHOD 1: NOT SHARING**

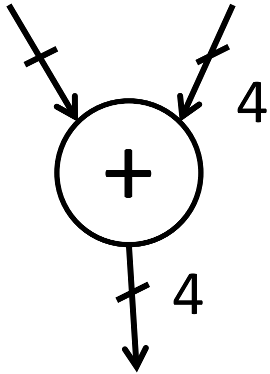
~~**METHOD 2: SHARING**~~

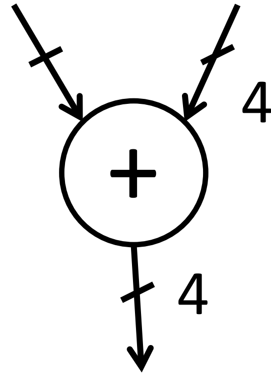
| Cyclone II | 8 LEs |
| --- | --- |
| Stratix IV | 4 ALMs |

| Cyclone II | 12 LEs |
| --- | --- |

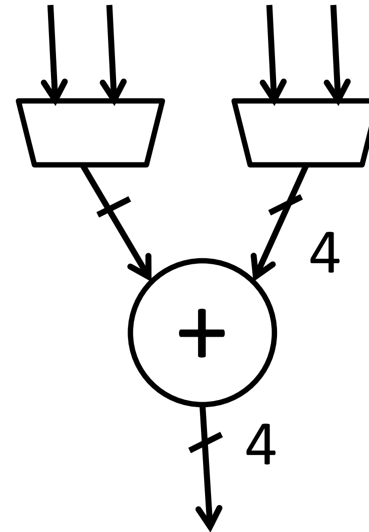
# Example: 4 Bit Adder

- Consider a C program which performs two additions
- Which hardware implementation is preferred?




**VS.**



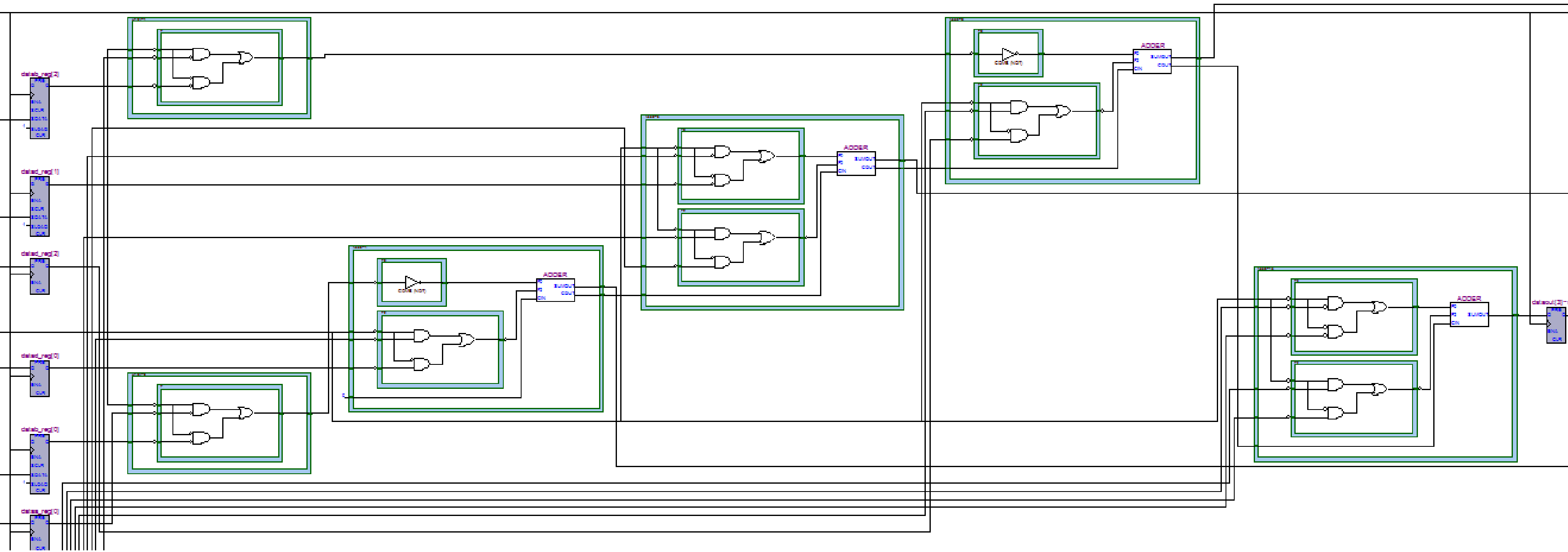

**METHOD 1: NOT SHARING**

**METHOD 2: SHARING**

| | |
|---|---|
| **Cyclone II** | **8 LEs** |
| **Stratix IV** | **4 ALMs** |

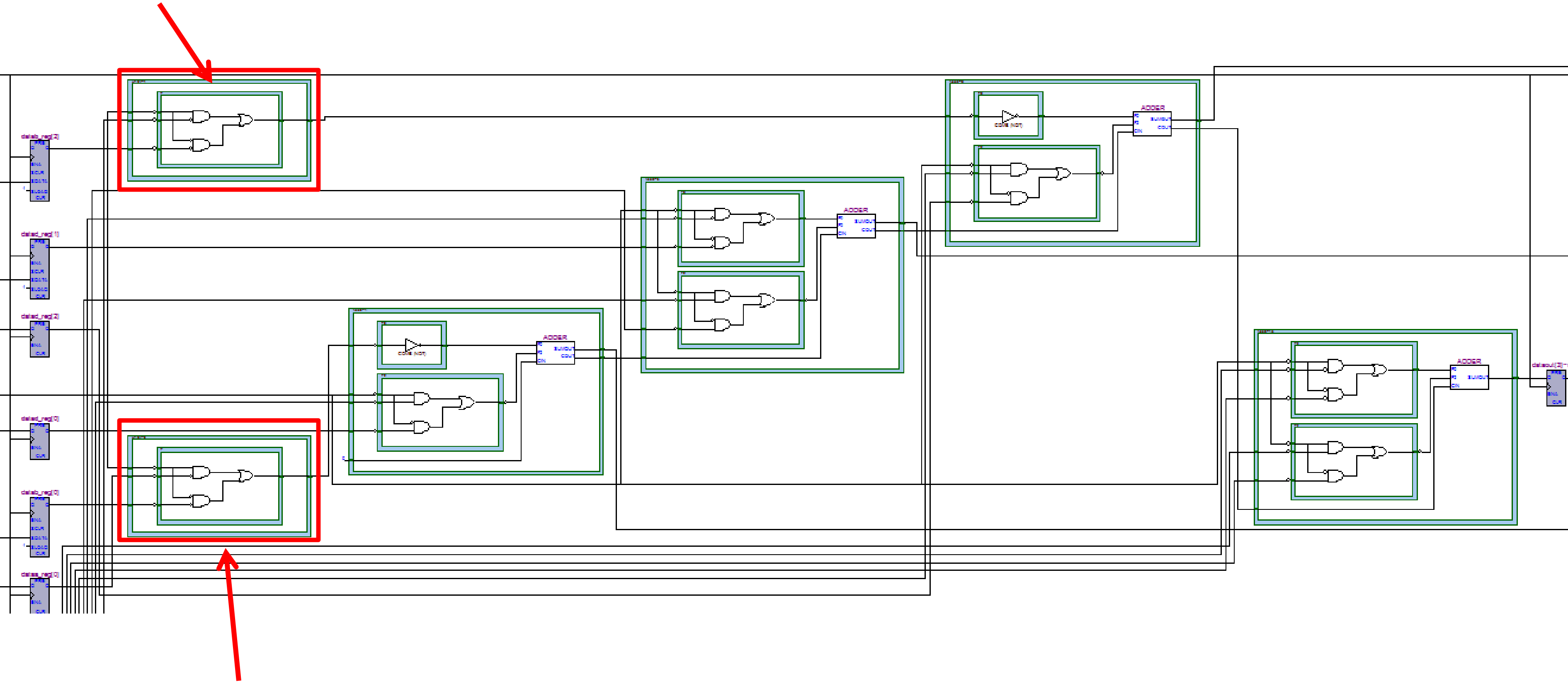
| | |
|---|---|
| **Cyclone II** | **12 LEs** |
| **Stratix IV** | **?? ALMs** |

# METHOD 2: SHARING (Stratix IV)



# METHOD 2: SHARING (Stratix IV)

MUX (3-LUT)

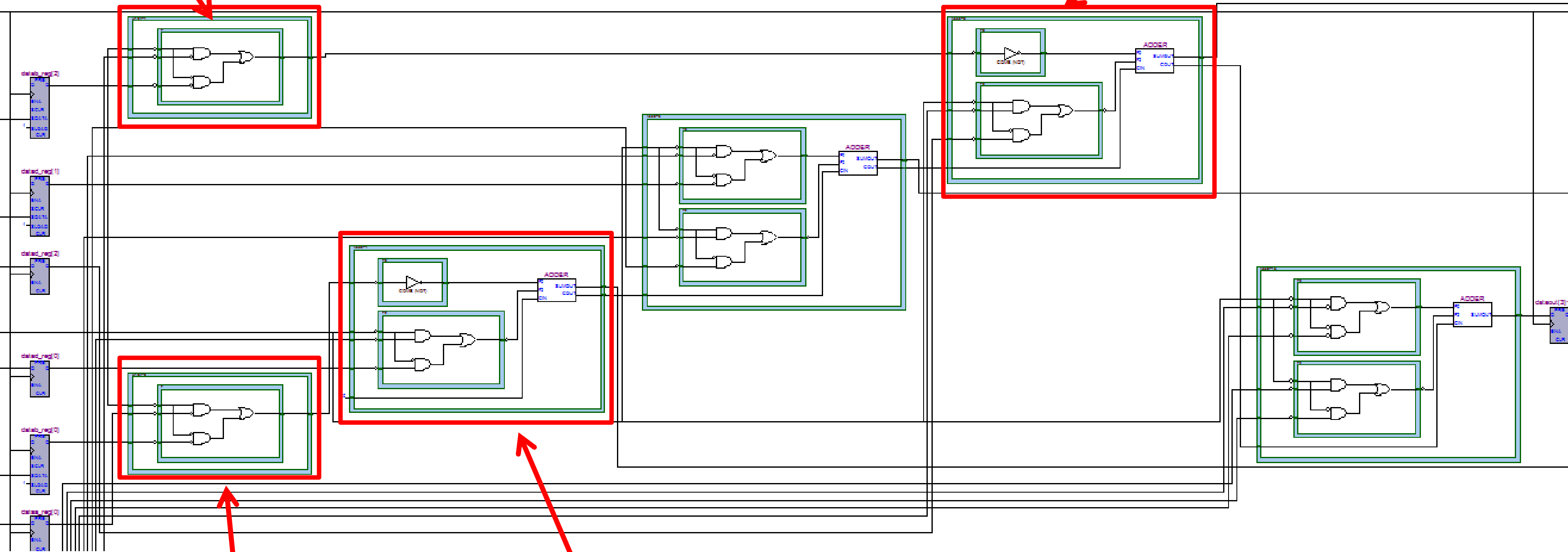


MUX (3-LUT)

# METHOD 2: SHARING (Stratix IV)

MUX (3-LUT)

MUX + adder (4-LUT)



MUX (3-LUT)

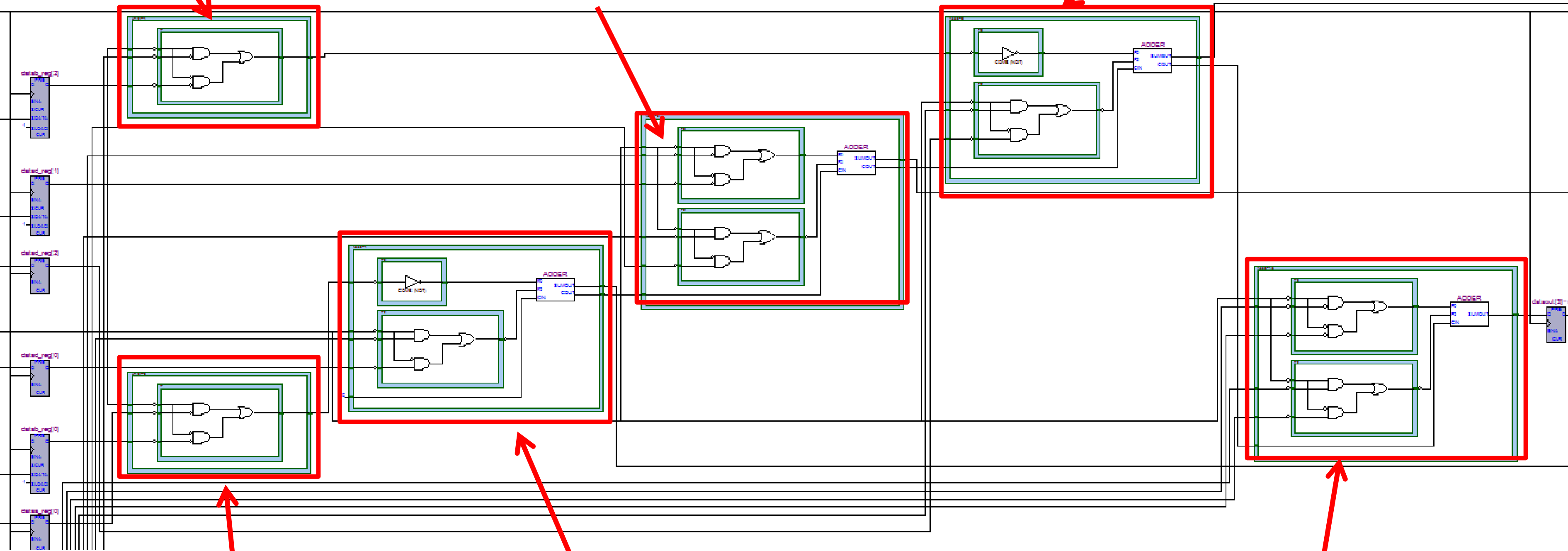
MUX + adder (4-LUT)

# METHOD 2: SHARING (Stratix IV)

MUX (3-LUT)

2 MUXes + adder (5-LUT)

MUX + adder (4-LUT)

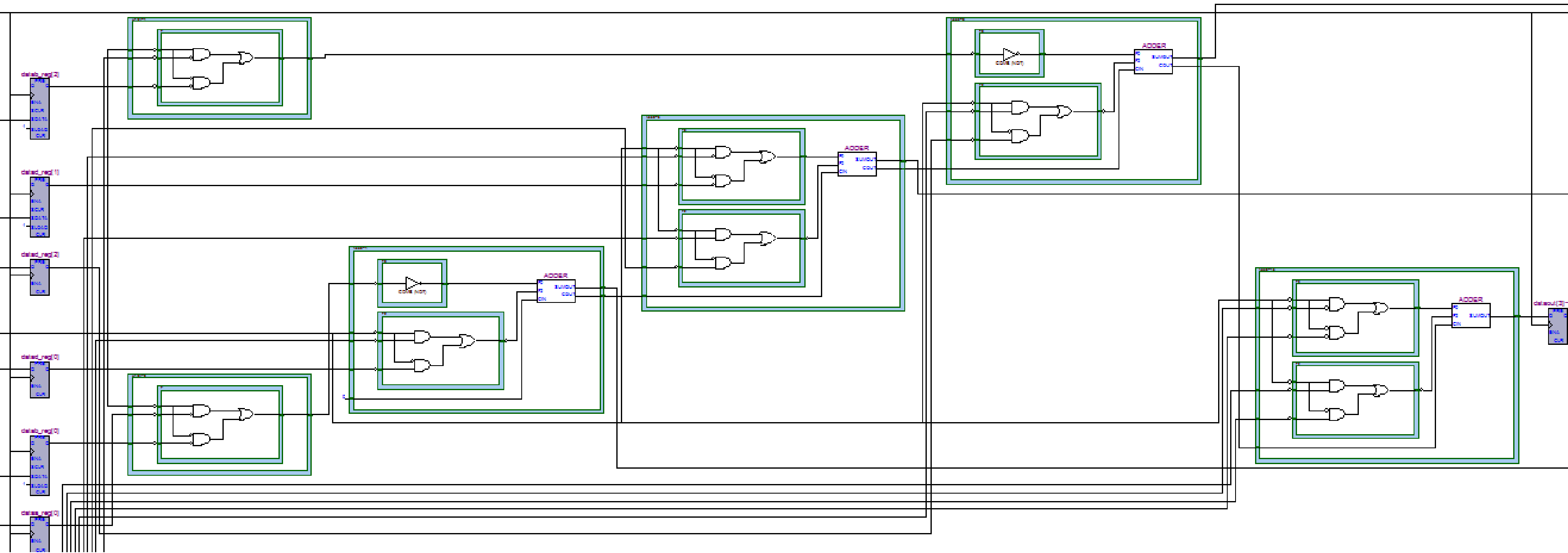


MUX (3-LUT)

MUX + adder (4-LUT)

2 MUXes + adder (5-LUT)

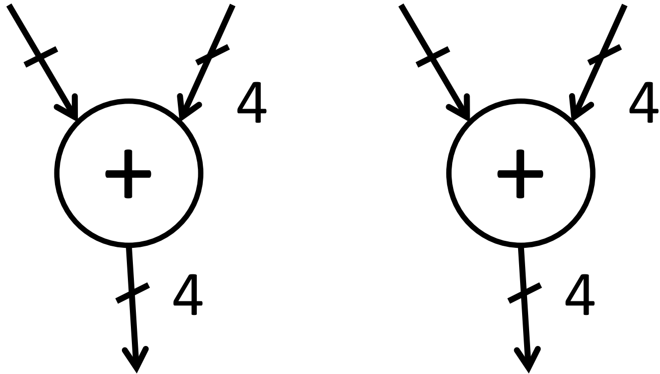
# METHOD 2: SHARING (Stratix IV)



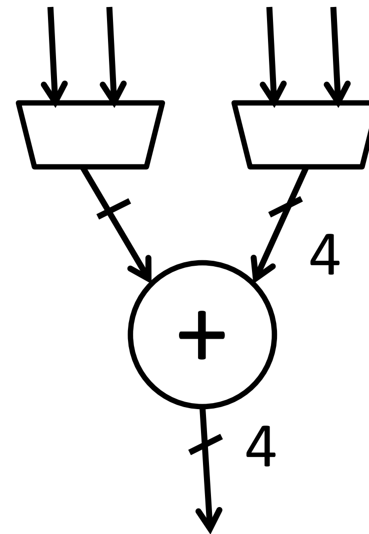

- Two 3-LUTs, Two 4-LUTs, Two 5-LUTs
- Quartus II maps this to only **3 ALMs:** One with two 4-LUTs, and Two with a 3-LUT and a 5-LUT

# Example: 4 Bit Adder

- Consider a C program which performs two additions
- Which hardware implementation is preferred?




VS.




**METHOD 1: NOT SHARING**

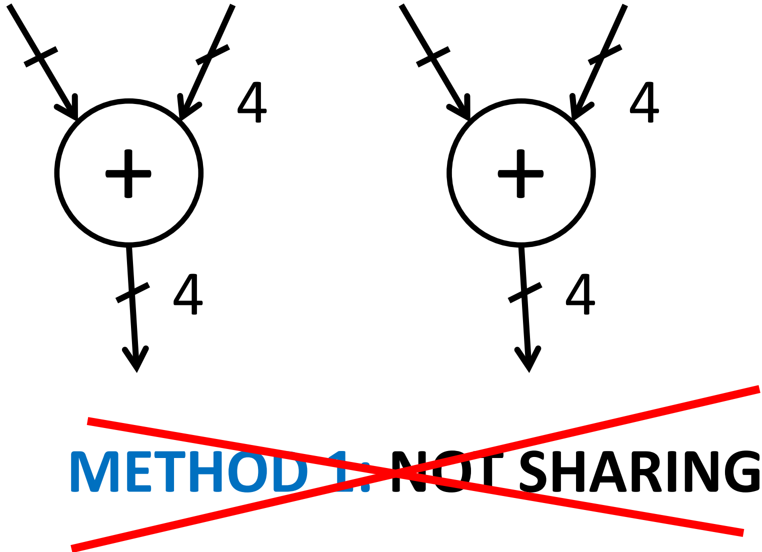
**METHOD 2: SHARING**

| | |
|---|---|
| **Cyclone II** | **8 LEs** |
| **Stratix IV** | **4 ALMs** |

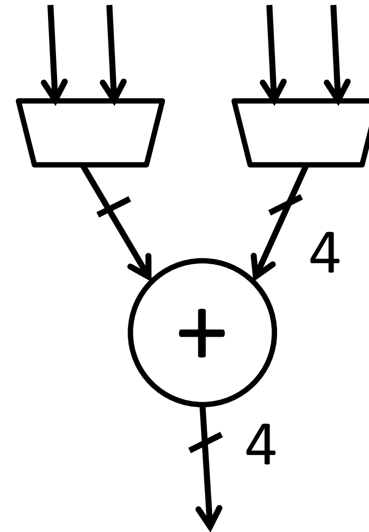
| | |
|---|---|
| **Cyclone II** | **12 LEs** |
| **Stratix IV** | **3 ALMs** |

# Example: 4 Bit Adder

- Consider a C program which performs two additions
- Which hardware implementation is preferred?




**VS.**




**METHOD 2: SHARING**

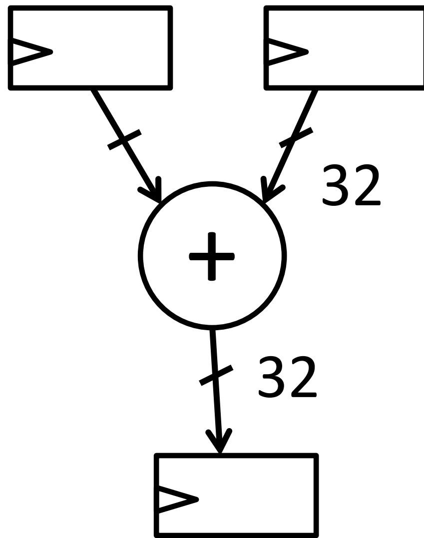
| | |
|---|---|
| **Cyclone II** | **8 LEs** |
| **Stratix IV** | **4 ALMs** |

| | |
|---|---|
| **Cyclone II** | **12 LEs** |
| **Stratix IV** | **3 ALMs** |

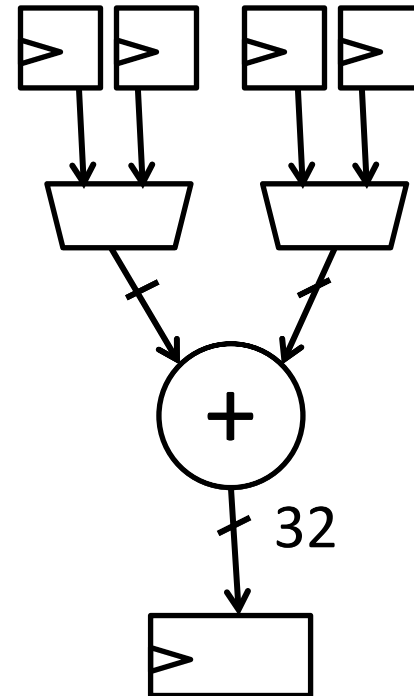
# Sharing Single Operators

# Evaluating Area of Single Operators

- This procedure was performed for all operators (**32-bit**) and on both architectures

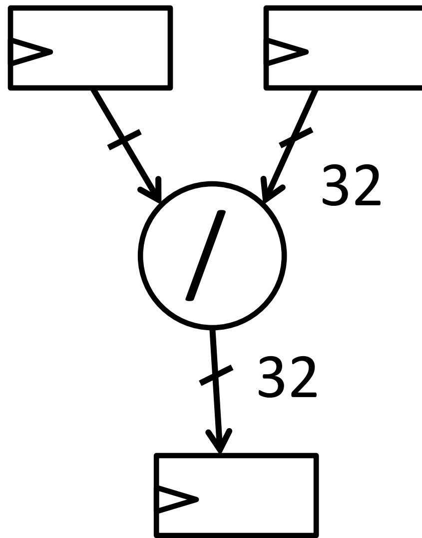



**VS.**

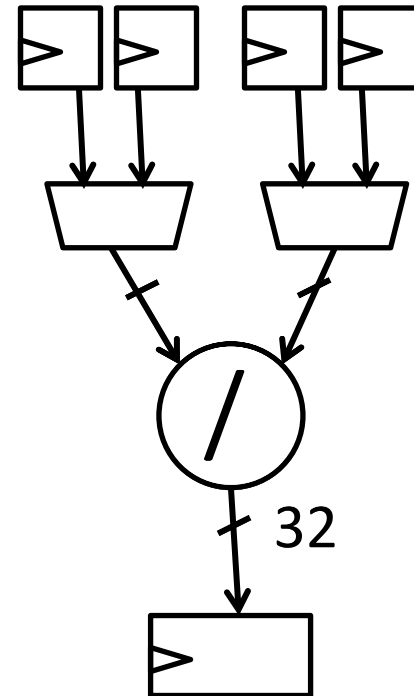


# Evaluating Area of Single Operators

- This procedure was performed for all operators (**32-bit**) and on both architectures

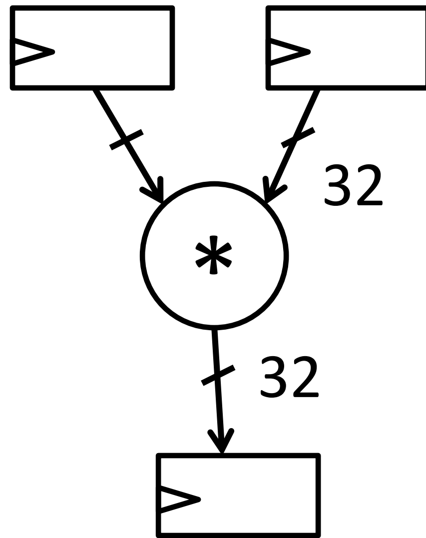



**VS.**

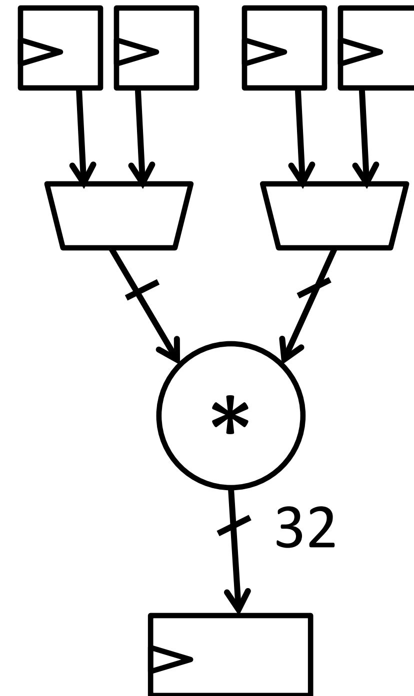


# Evaluating Area of Single Operators

- This procedure was performed for all operators (**32-bit**) and on both architectures

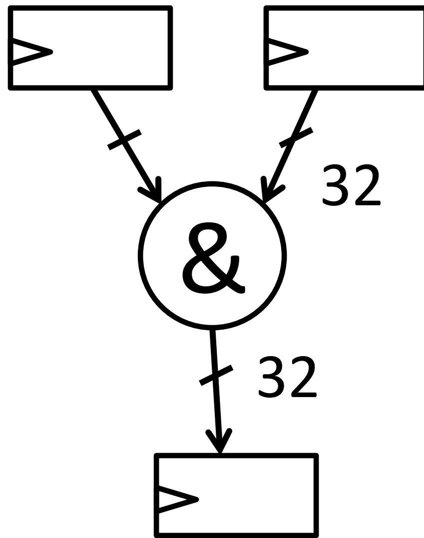



VS.

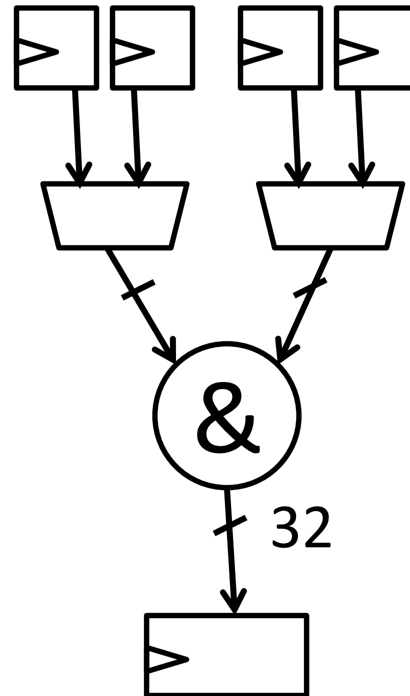


# Evaluating Area of Single Operators

- This procedure was performed for all operators (**32-bit**) and on both architectures




**VS.**



# Evaluating Area of Single Operators

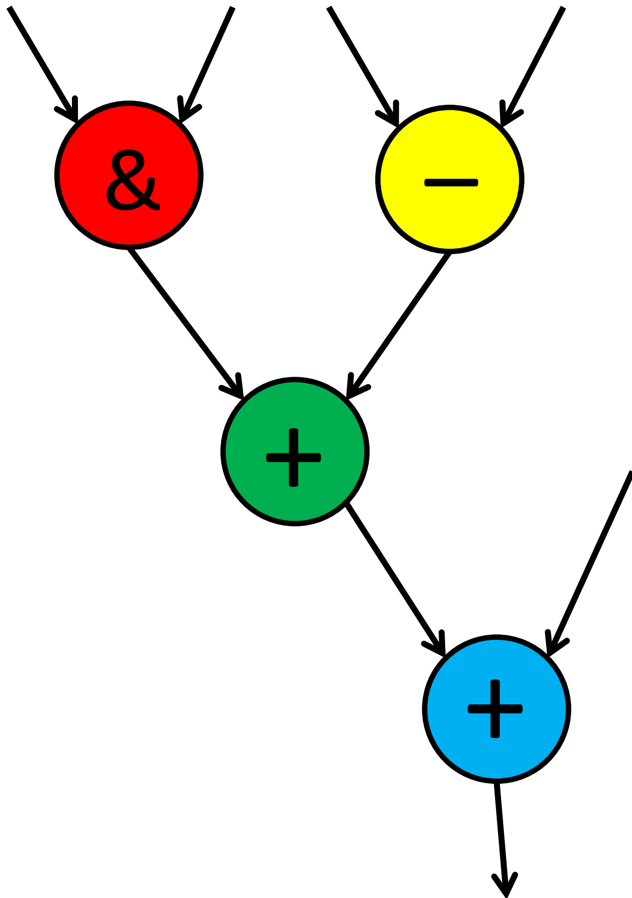
- Isolated operators which reduce area when shared:

| **Cyclone II** | **Stratix IV** |
|---|---|
| Div/Mod | Div/Mod |
| Multipliers | Multipliers |
| Barrel Shifters | Barrel Shifters |
| | **Add/Subtract** |
| | **Bitwise Operations (OR, XOR, AND)** |

# Sharing Composite Operators

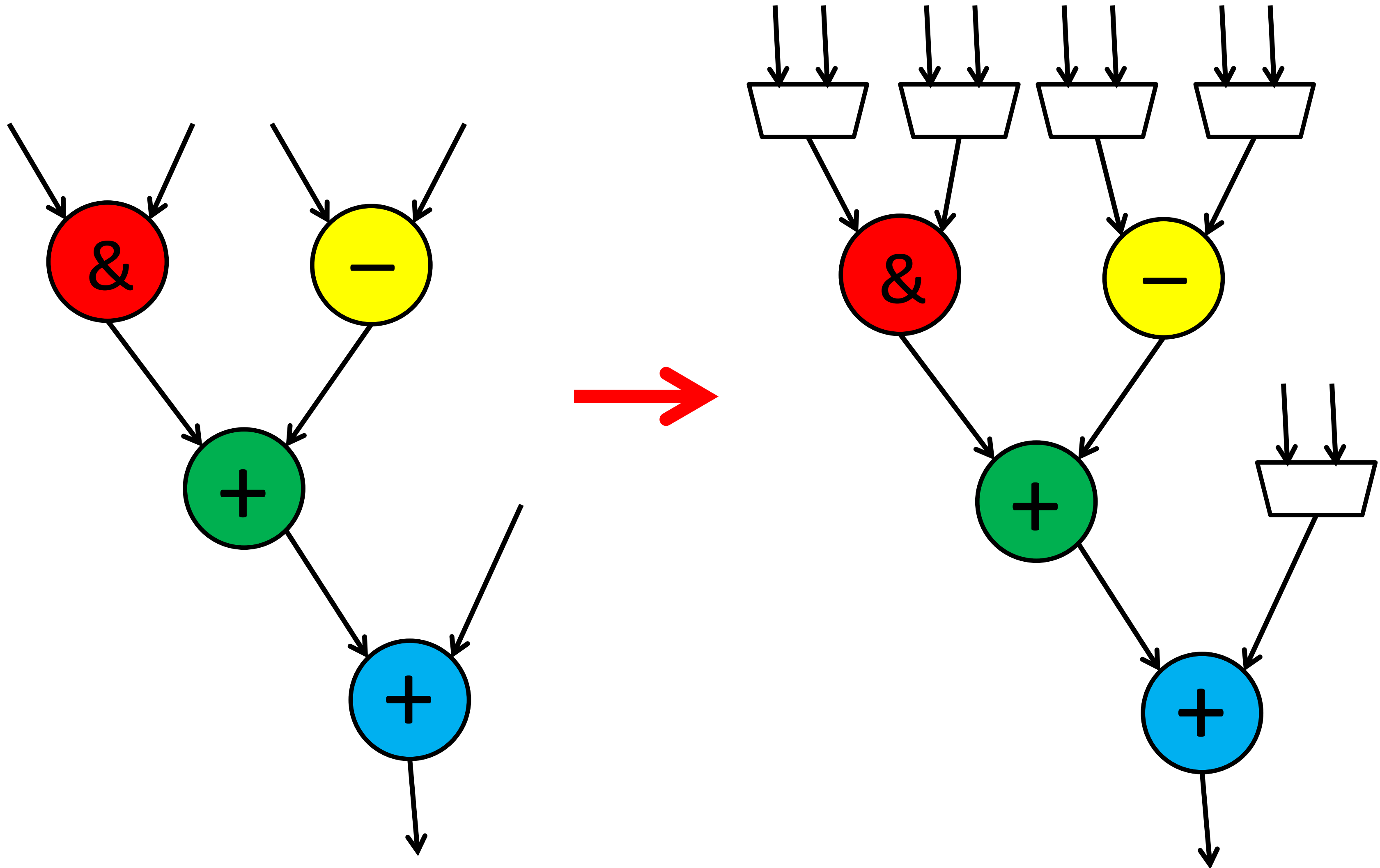
# Sharing Computational Patterns

- The focus of this work is on sharing **patterns** of smaller operators



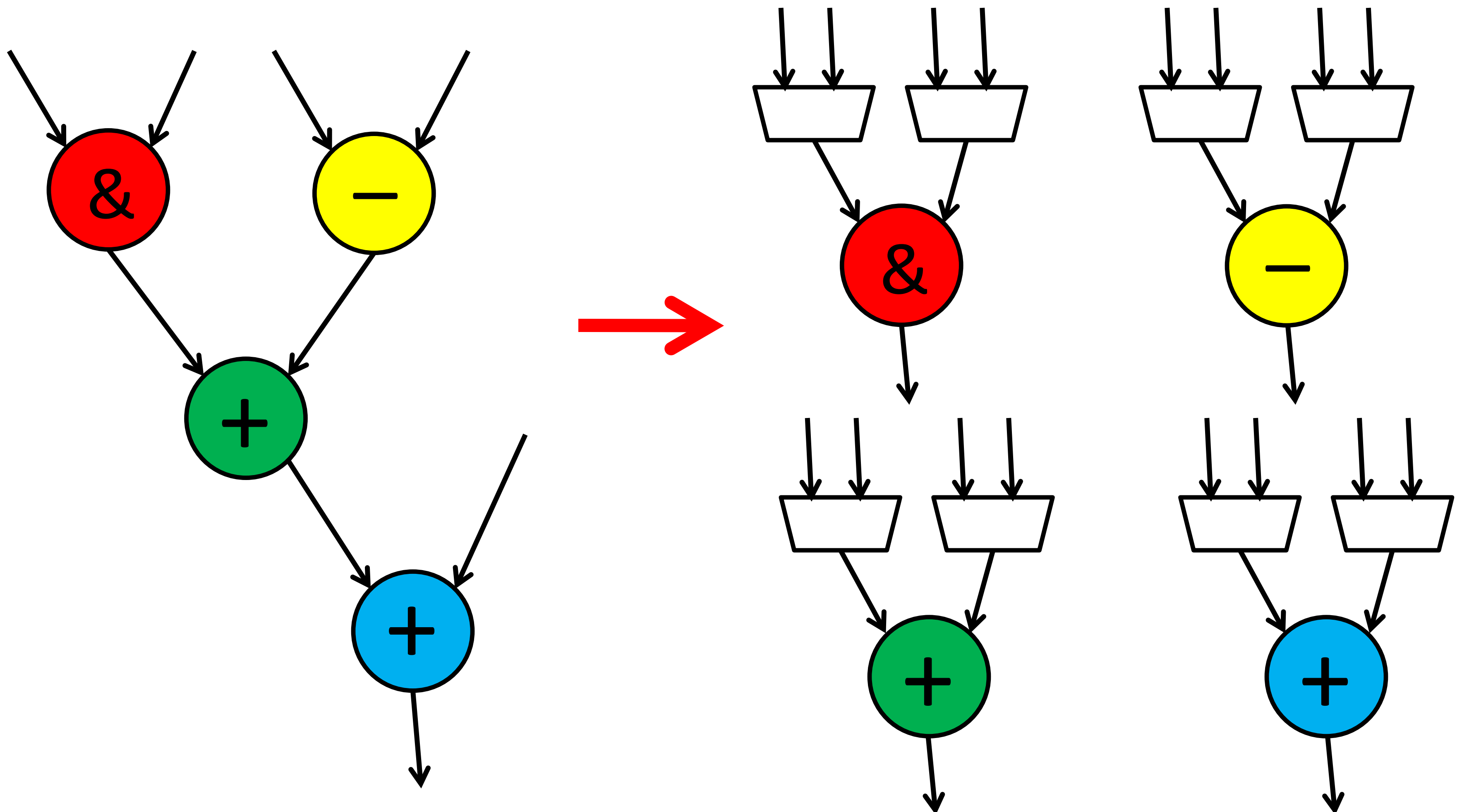
# Sharing Computational Patterns

- The focus of this work is on sharing **patterns** of smaller operators



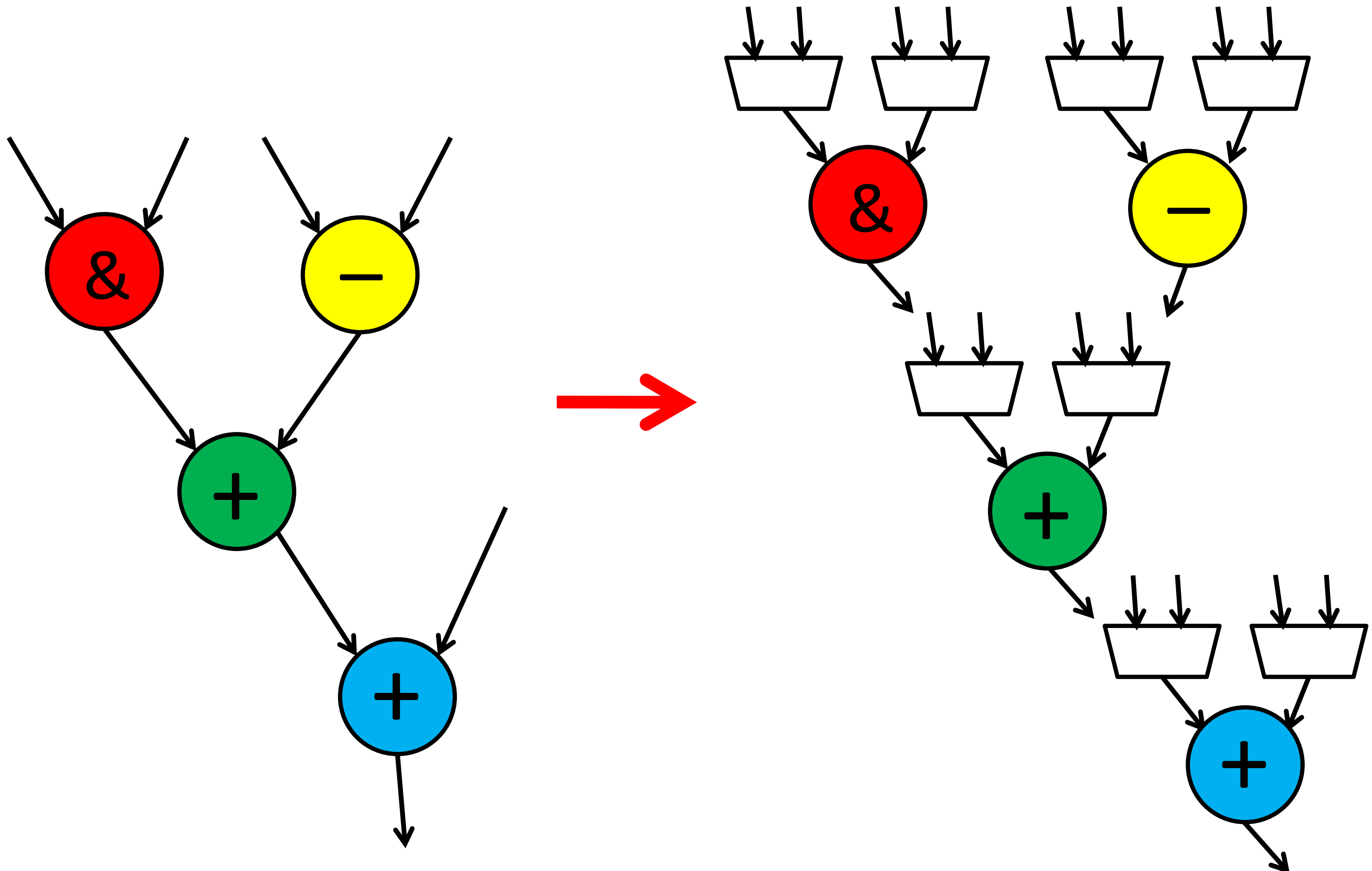
# Sharing Computational Patterns

- The focus of this work is on sharing **patterns** of smaller operators



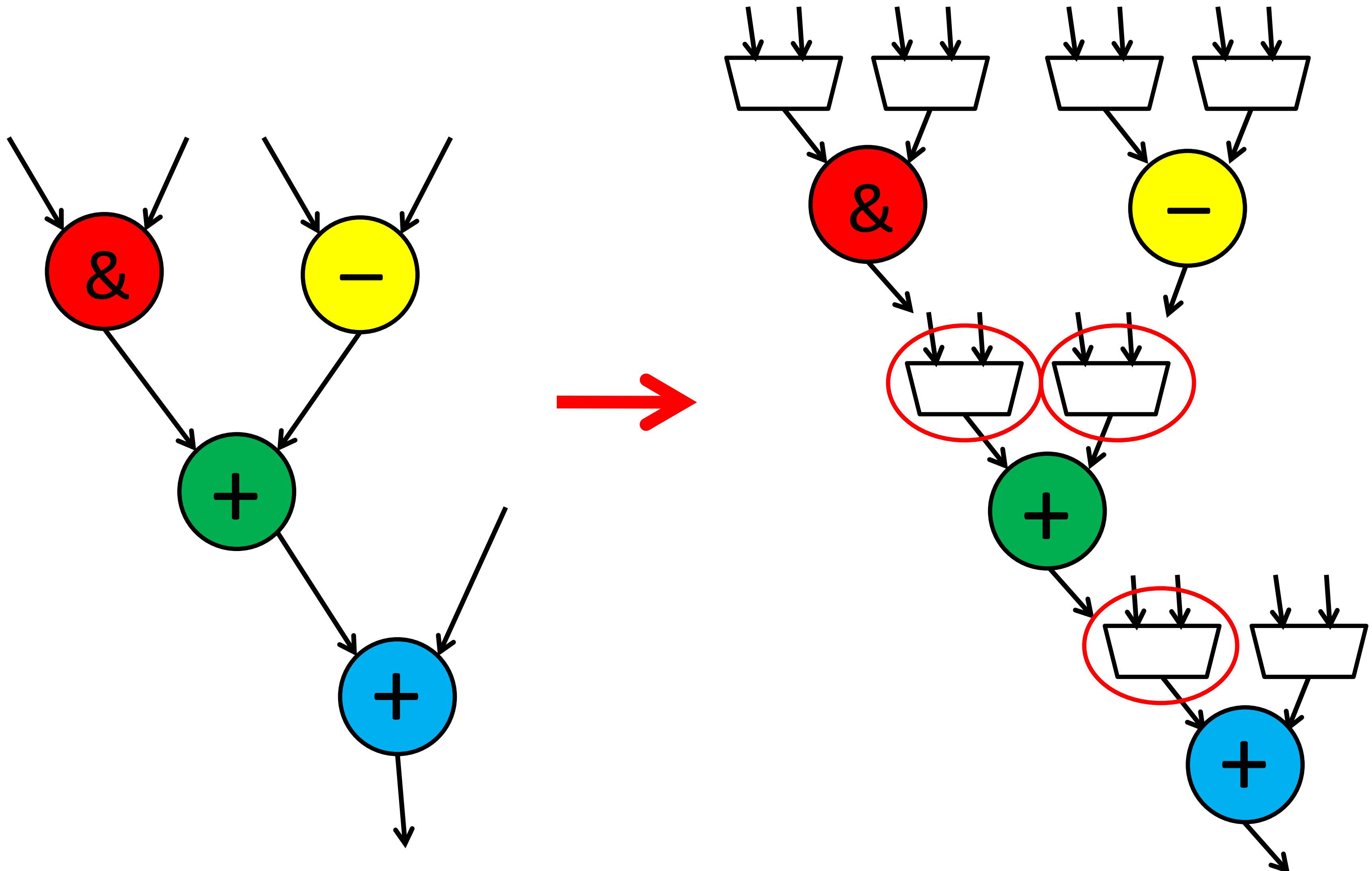
# Sharing Computational Patterns

- The focus of this work is on sharing **patterns** of smaller operators



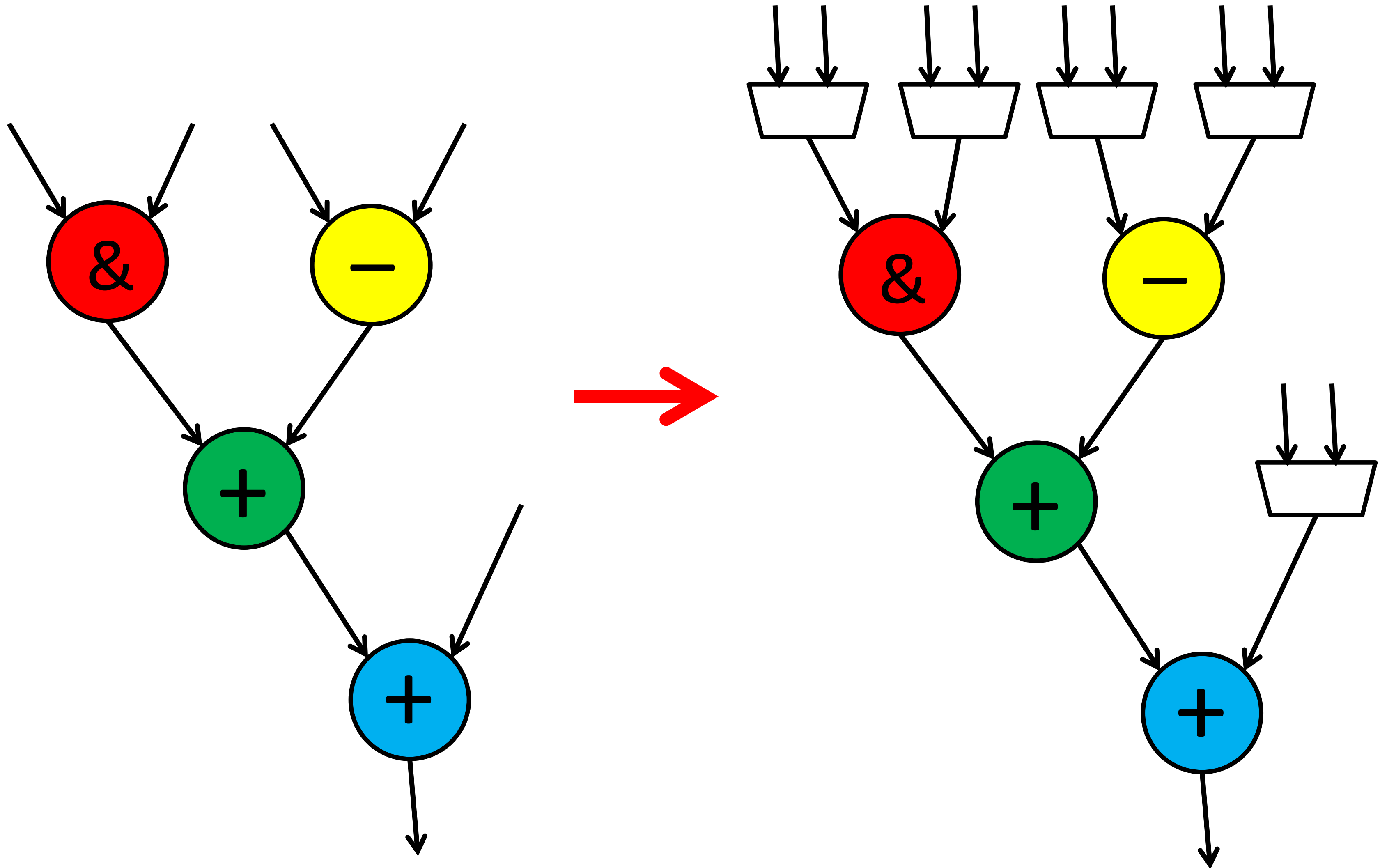
# Sharing Computational Patterns

- The focus of this work is on sharing **patterns** of smaller operators



# Sharing Computational Patterns

- The focus of this work is on sharing **patterns** of smaller operators



# Pattern Sharing in LegUp

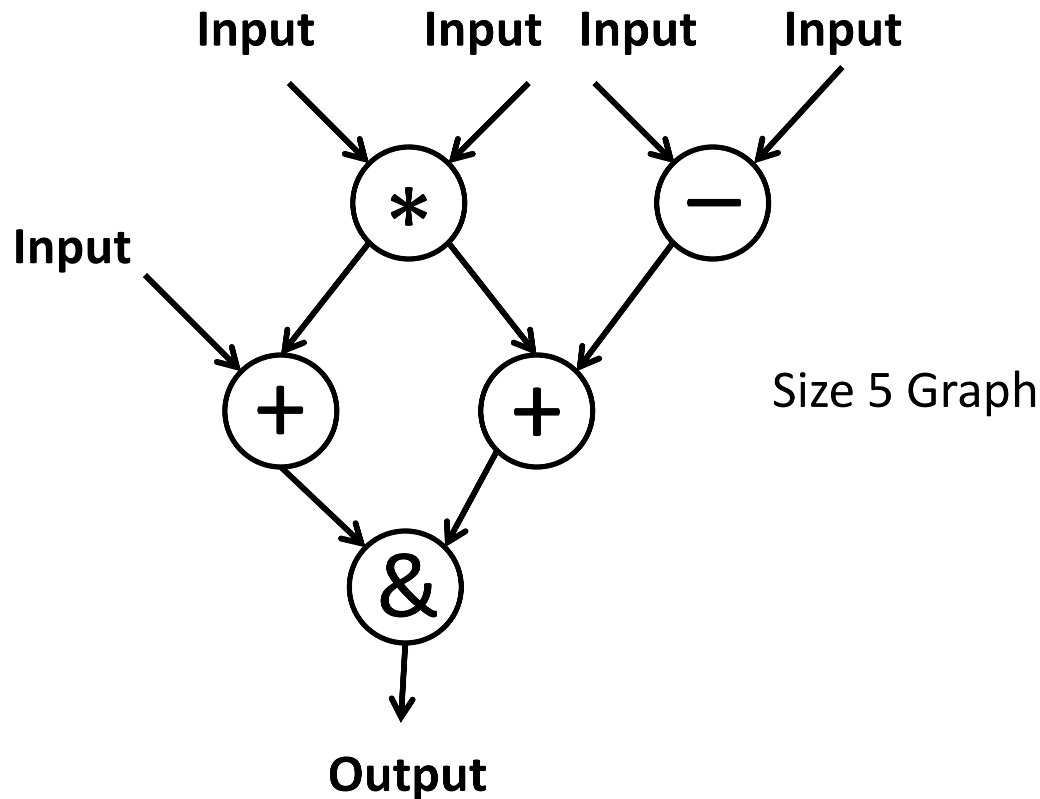
# Pattern Sharing Algorithm

LegUp’s Pattern Sharing Algorithm:

1. Find all **computational patterns** in the software program (up to pattern size of 10)
2. **Group together** patterns which can be implemented using the same hardware
3. **Select pairs** of equivalent patterns from step 2 to be implemented using the same hardware

# Pattern Representation

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

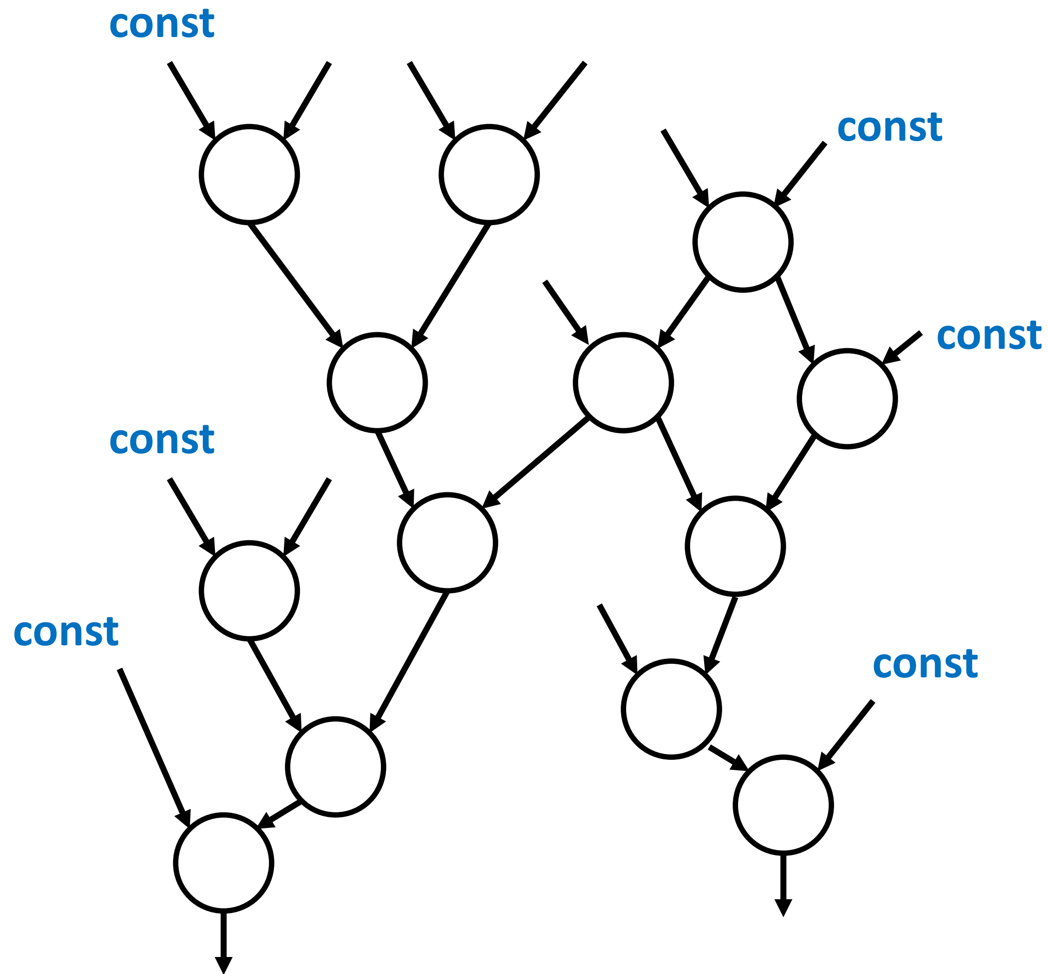


# 1. Finding all Computational Patterns

- LegUp uses a Data Flow Graph (DFG) to represent each compiled C Program
- The first step of the pattern sharing algorithm is to find all **subgraphs** of this DFG which are candidates for sharing:

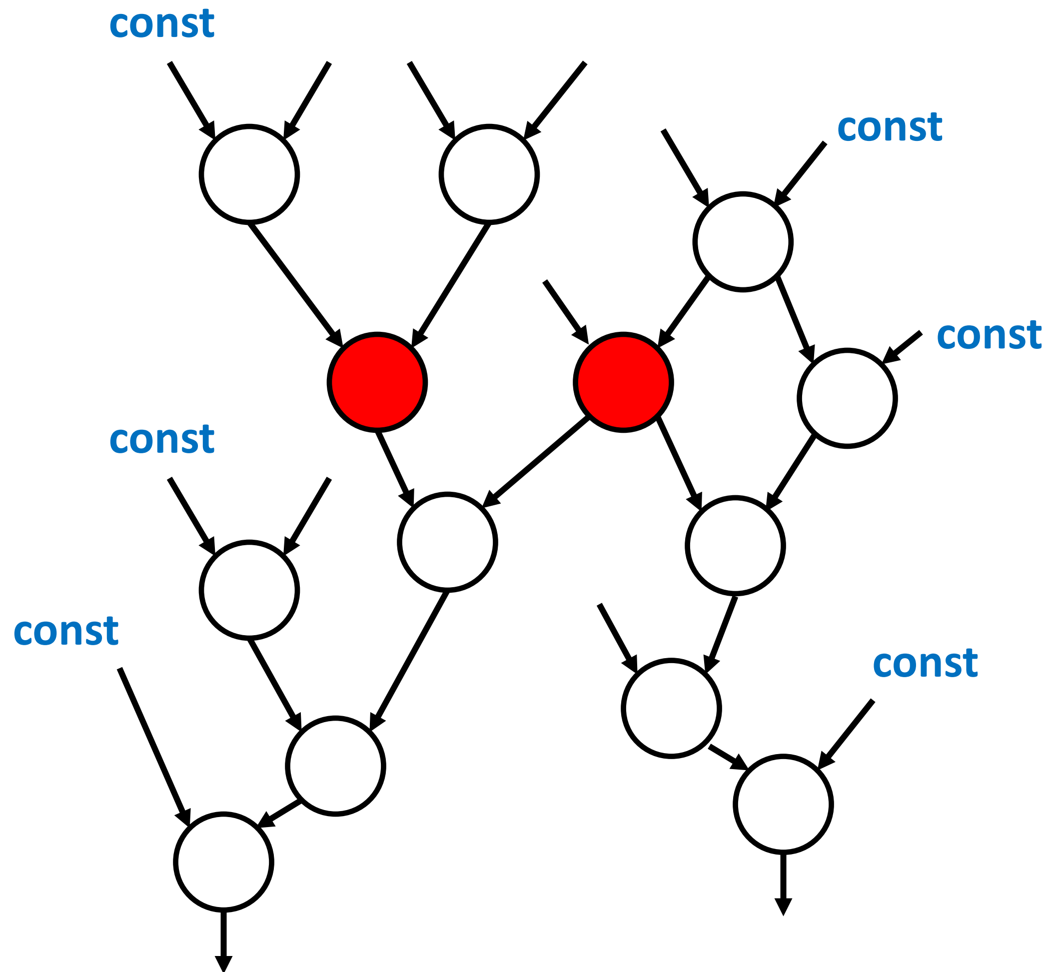
# 1. Finding all Computational Patterns

Consider the following DFG produced by LegUp:

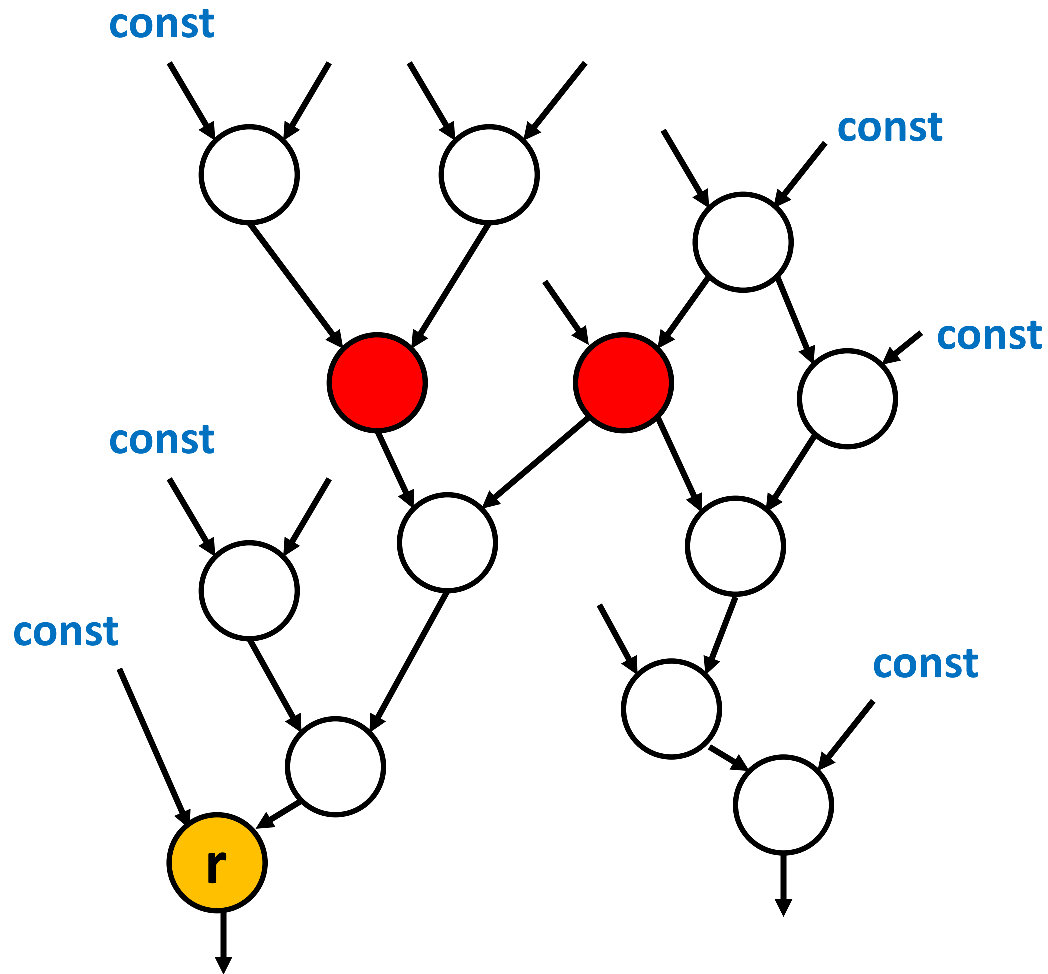


# 1. Finding all Computational Patterns

RED = "Invalid"
(e.g. Branch instruction)



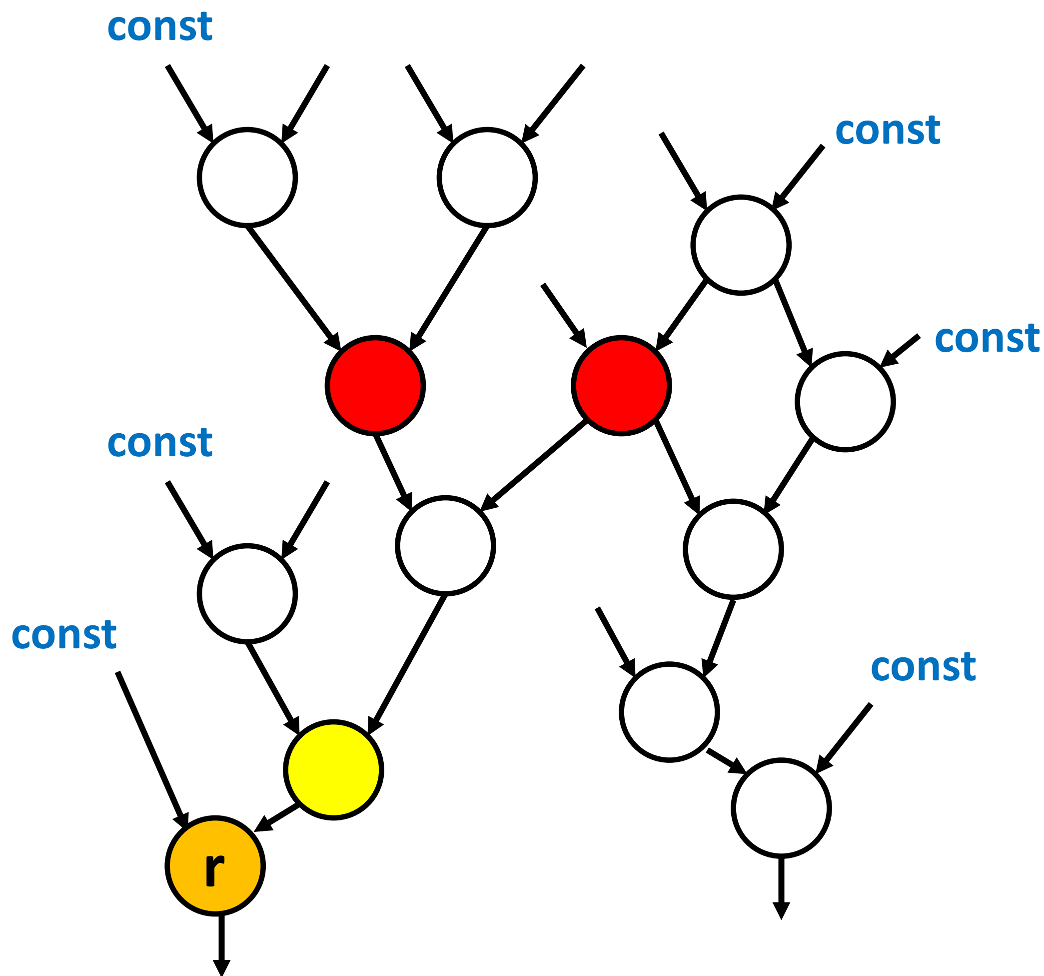
# 1. Finding all Computational Patterns




Size: 1

# 1. Finding all Computational Patterns

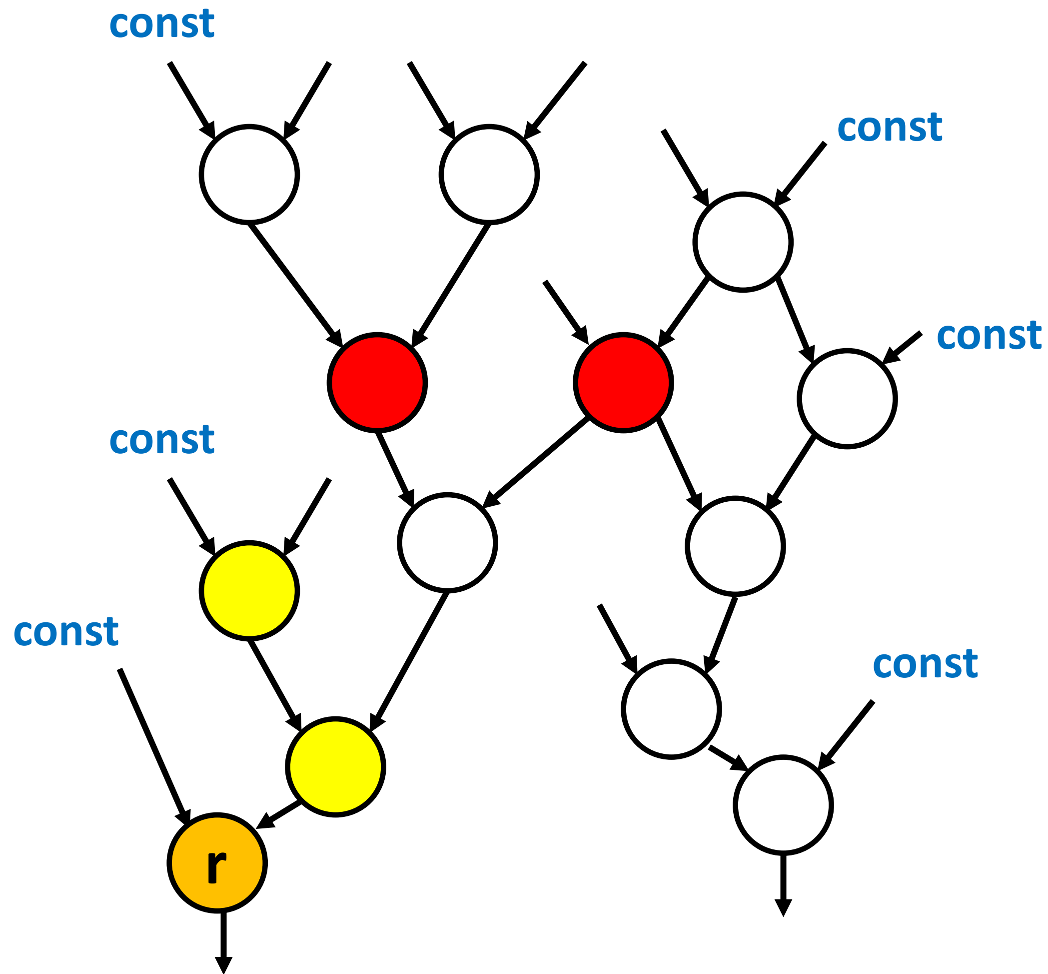
Perform a BFS of all the predecessors of **r** to find all subgraphs rooted at **r**




Size: 2

# 1. Finding all Computational Patterns

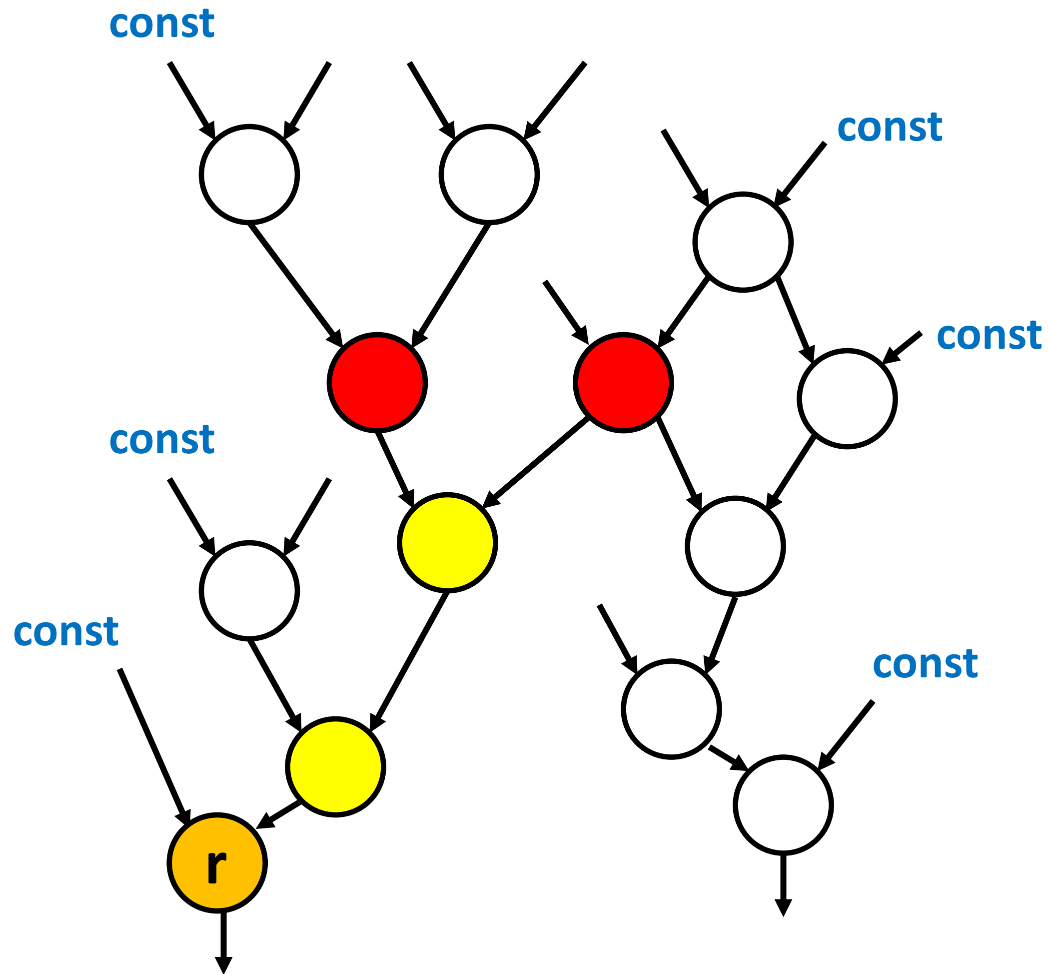
Perform a BFS of all the predecessors of **r** to find all subgraphs rooted at **r**




Size: 3

# 1. Finding all Computational Patterns

Perform a BFS of all the predecessors of **r** to find all subgraphs rooted at **r**

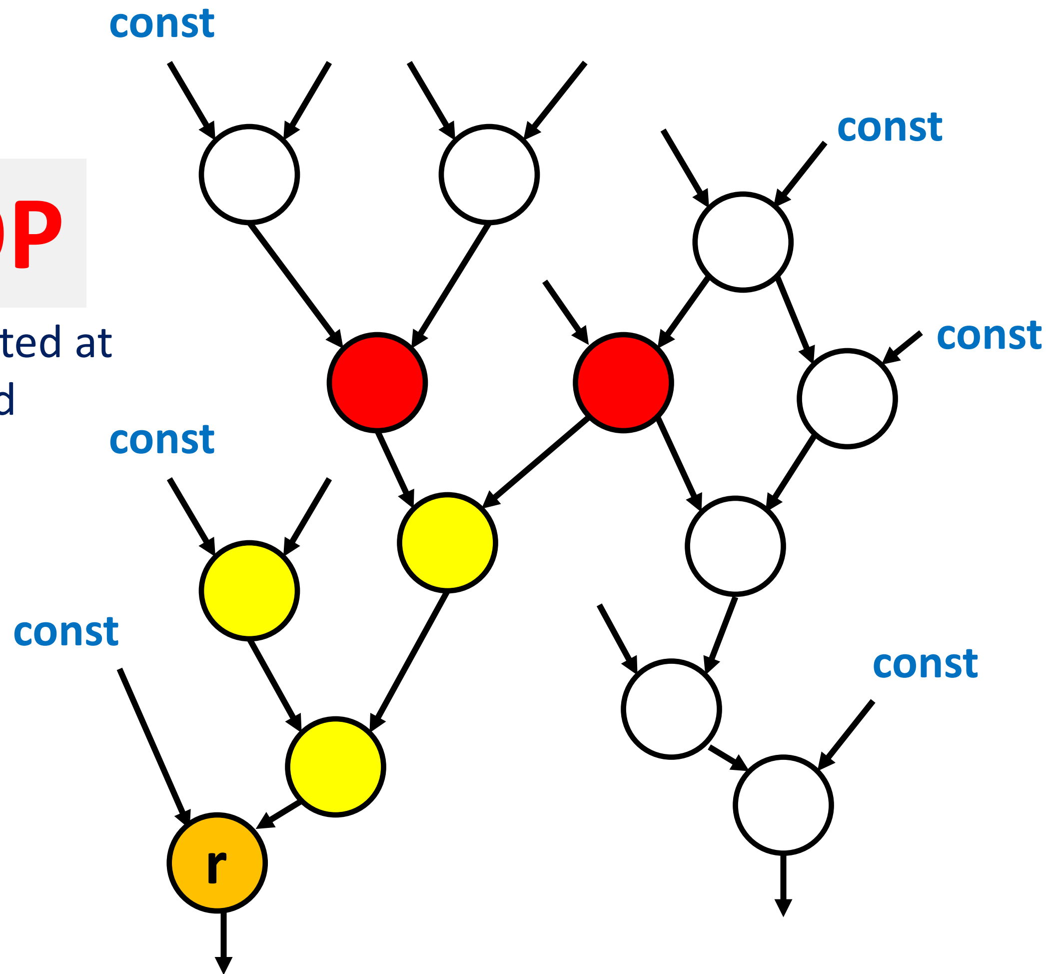



Size: 3

# 1. Finding all Computational Patterns

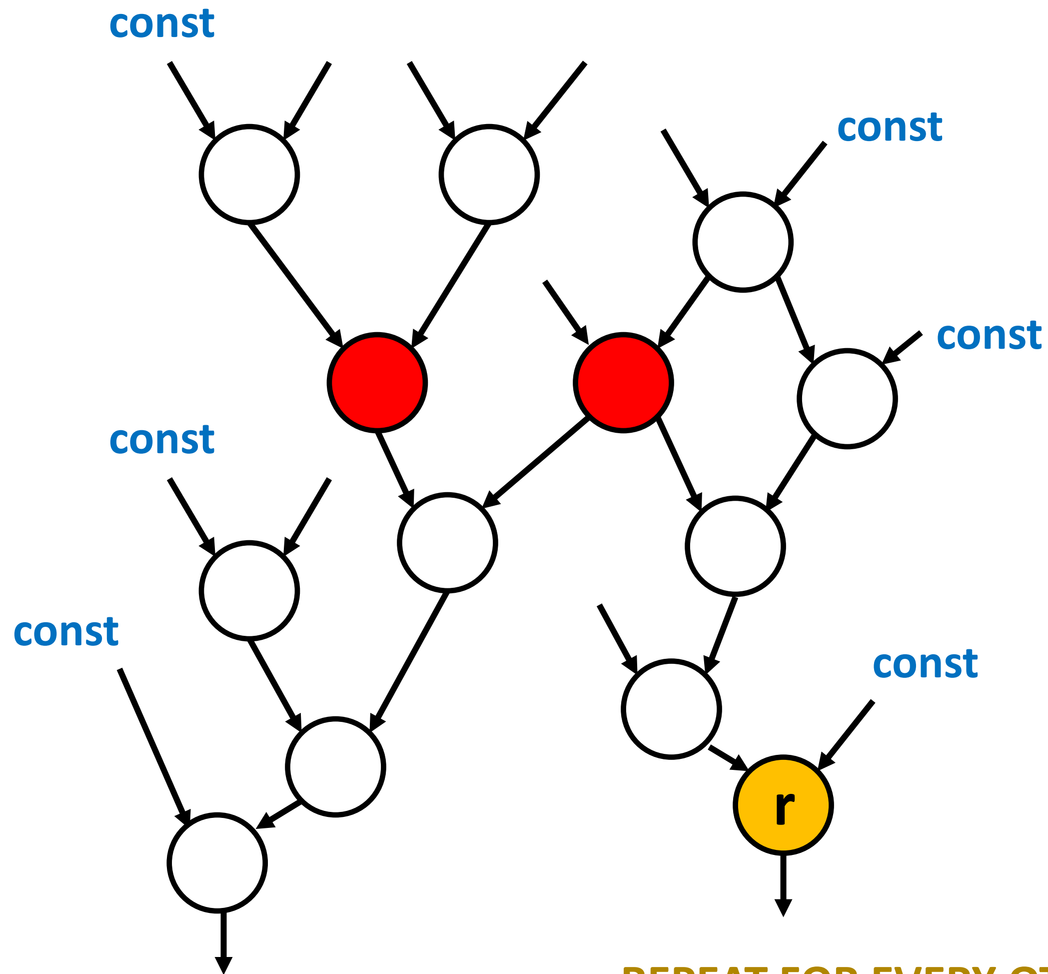
**STOP**

All subgraphs rooted at **r** have been found




Size: 4

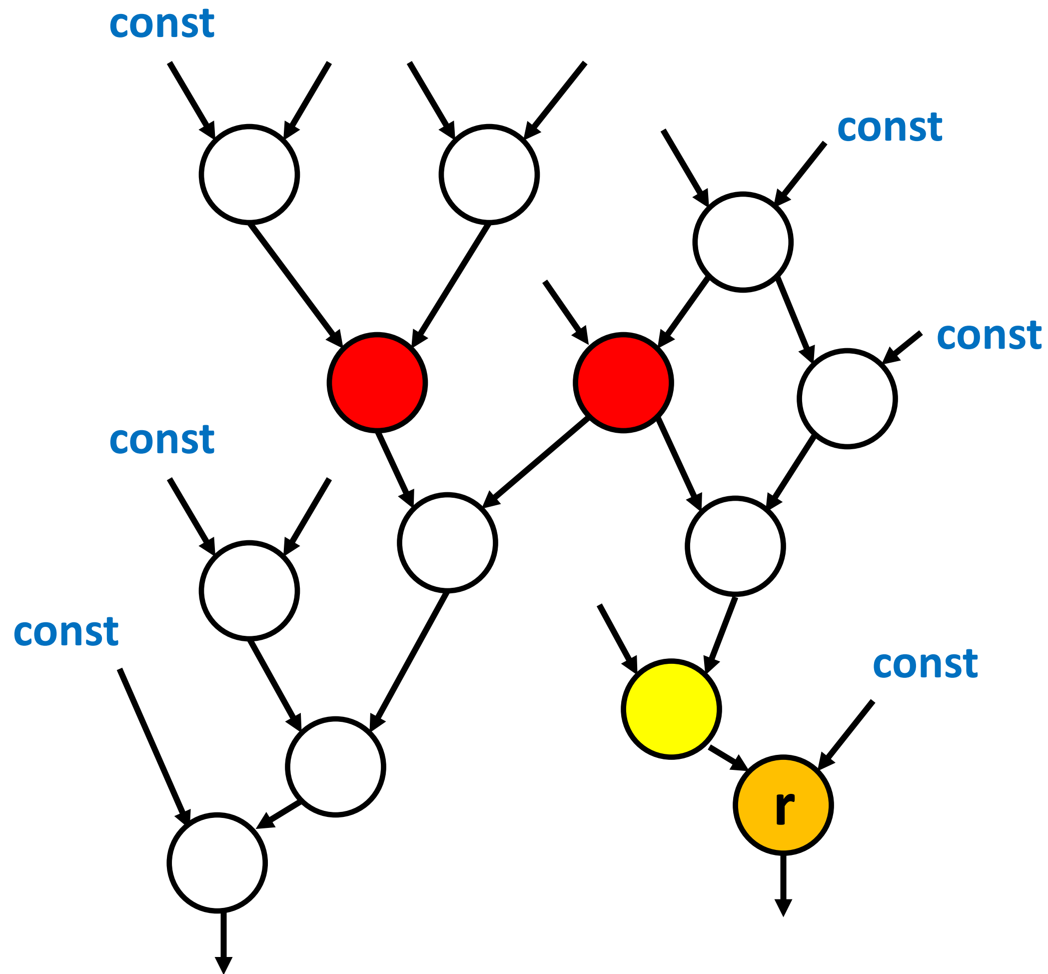
# 1. Finding all Computational Patterns




Size: 1

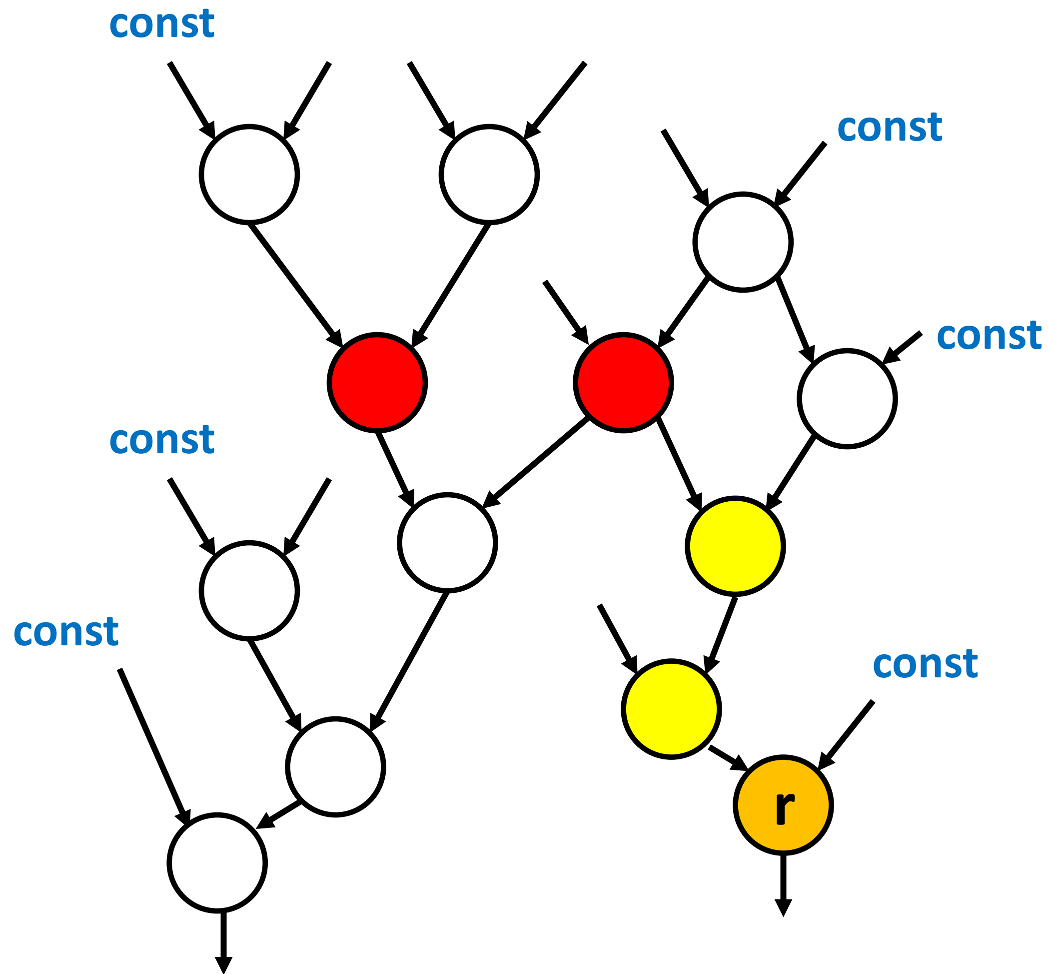
REPEAT FOR EVERY OTHER NODE

# 1. Finding all Computational Patterns



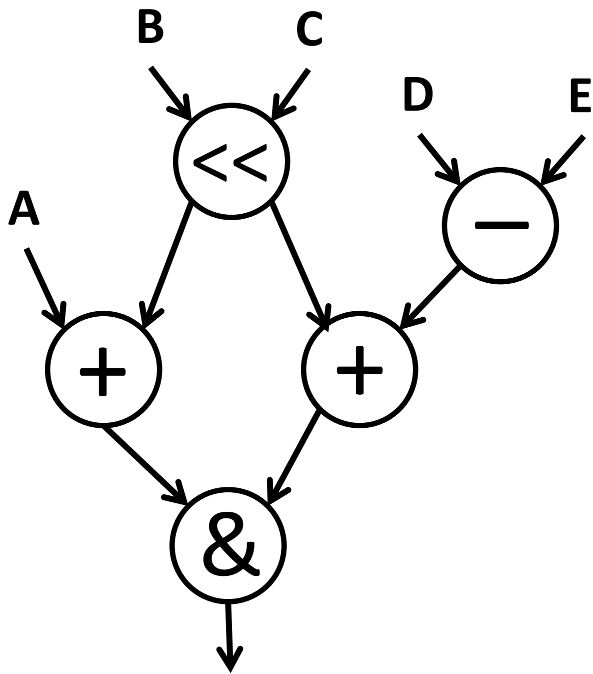

Size: 2

# 1. Finding all Computational Patterns

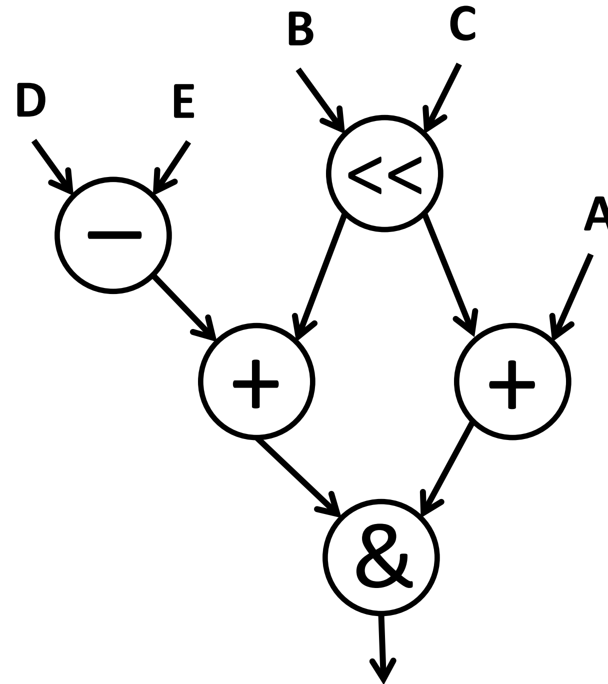



Size: 3

# 2. Sorting Patterns by Isomorphic Equivalence



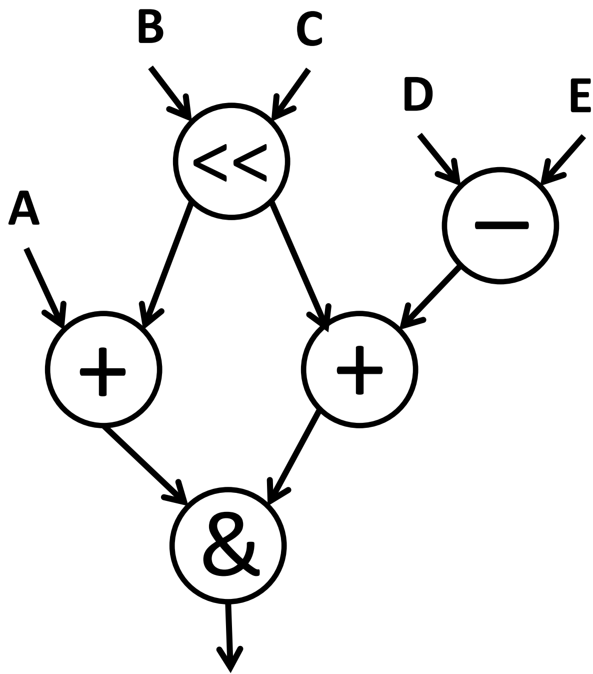

a) A Graph with a re-converging path



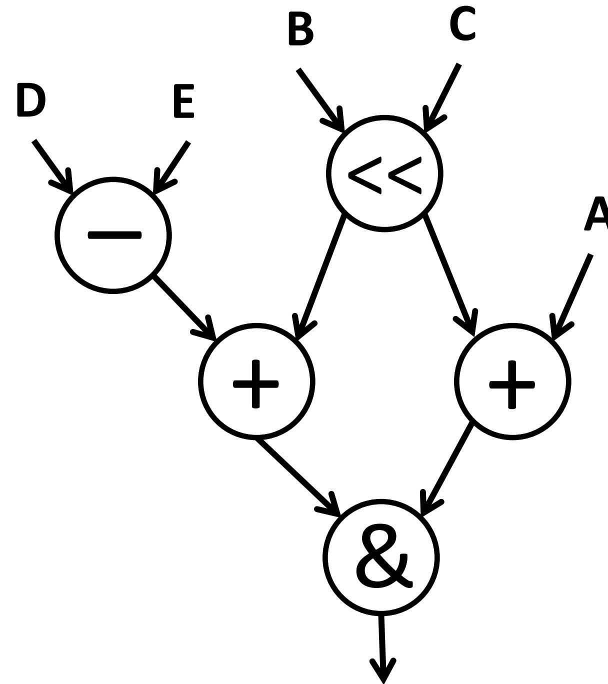

b) This graph can be implemented with the same hardware as (a) but is topologically different due to commutativity

# 2. Sorting Patterns by Isomorphic Equivalence

As opposed to just topological




a) A Graph with a re-converging path



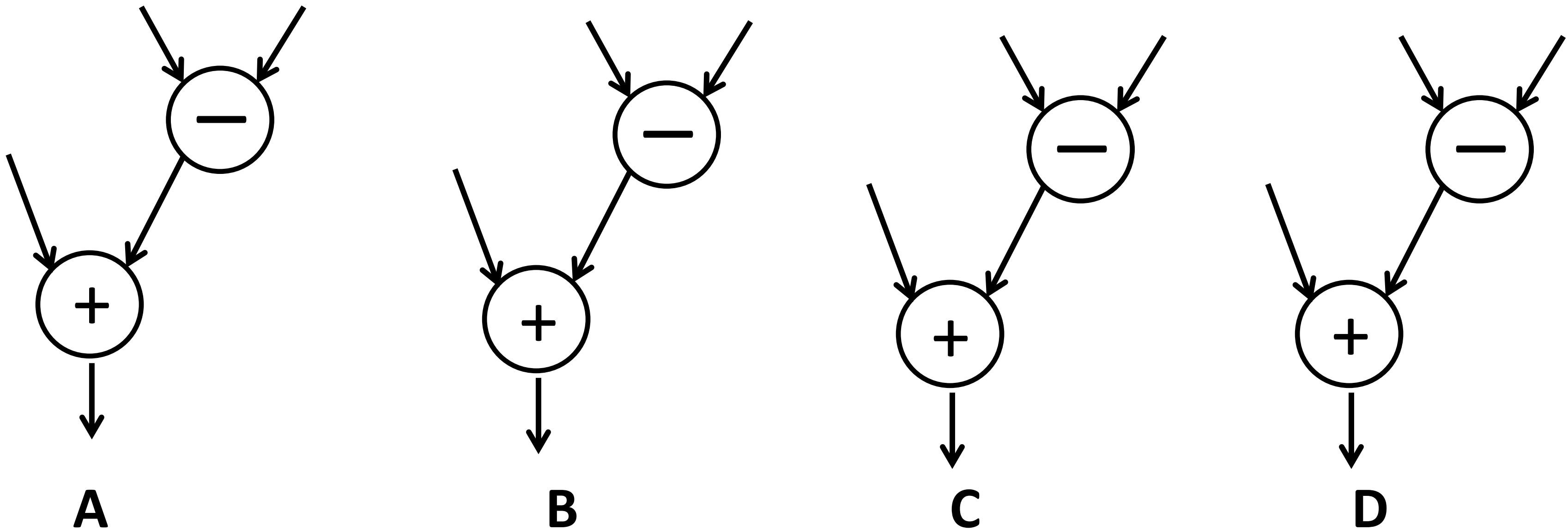

b) This graph can be implemented with the same hardware as (a) but is topologically different due to commutativity

# 3. Decide which Pattern Instances to Share

- So far, steps 1 and 2 have provided sets of equivalent pattern graphs

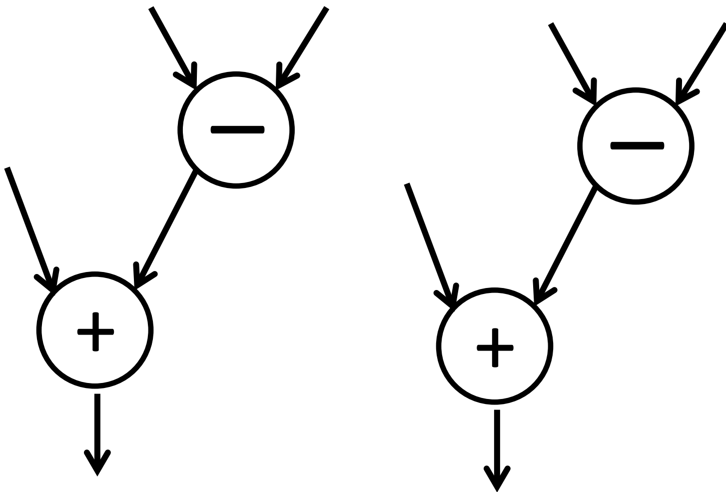
# 3. Decide which Pattern Instances to Share

- So far, steps 1 and 2 have provided sets of equivalent pattern graphs
- For example, we may have found 4 graphs for this pattern:



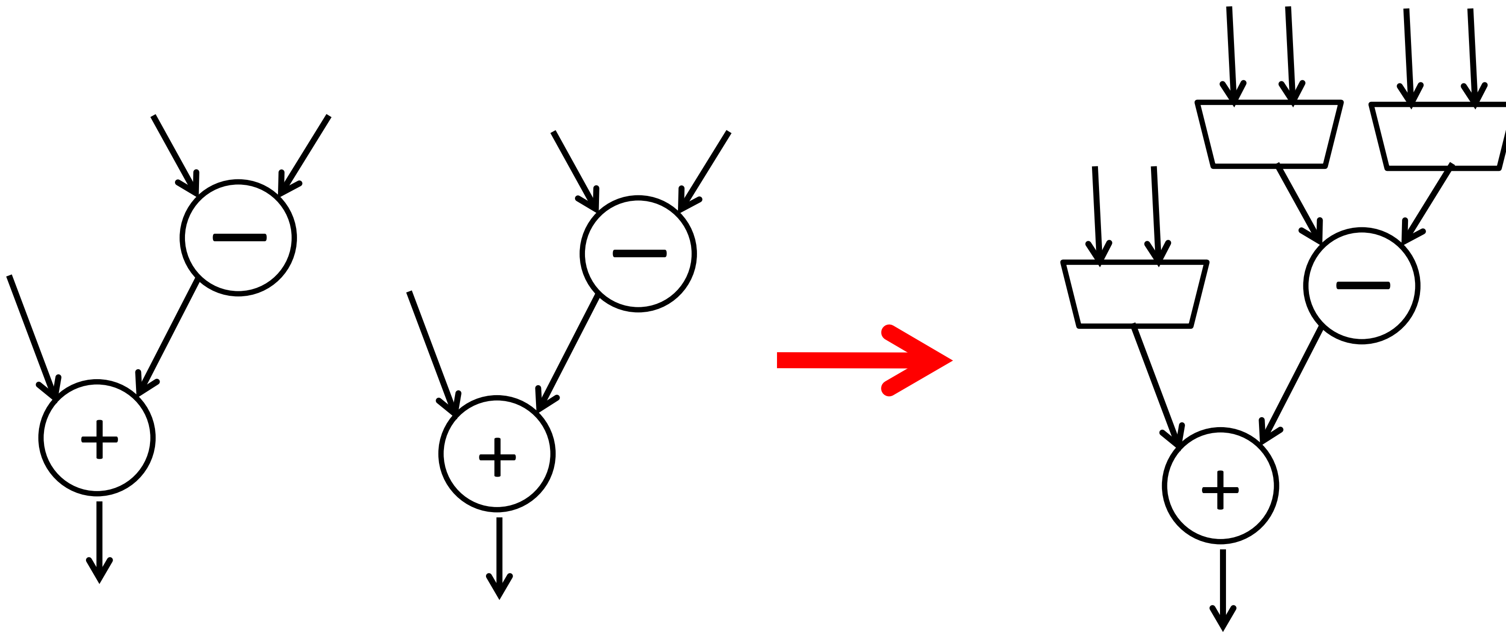
# 3. Decide which Pattern Instances to Share

- So far, steps 1 and 2 have provided sets of equivalent pattern graphs
- For example, we may have found 4 graphs for this pattern:



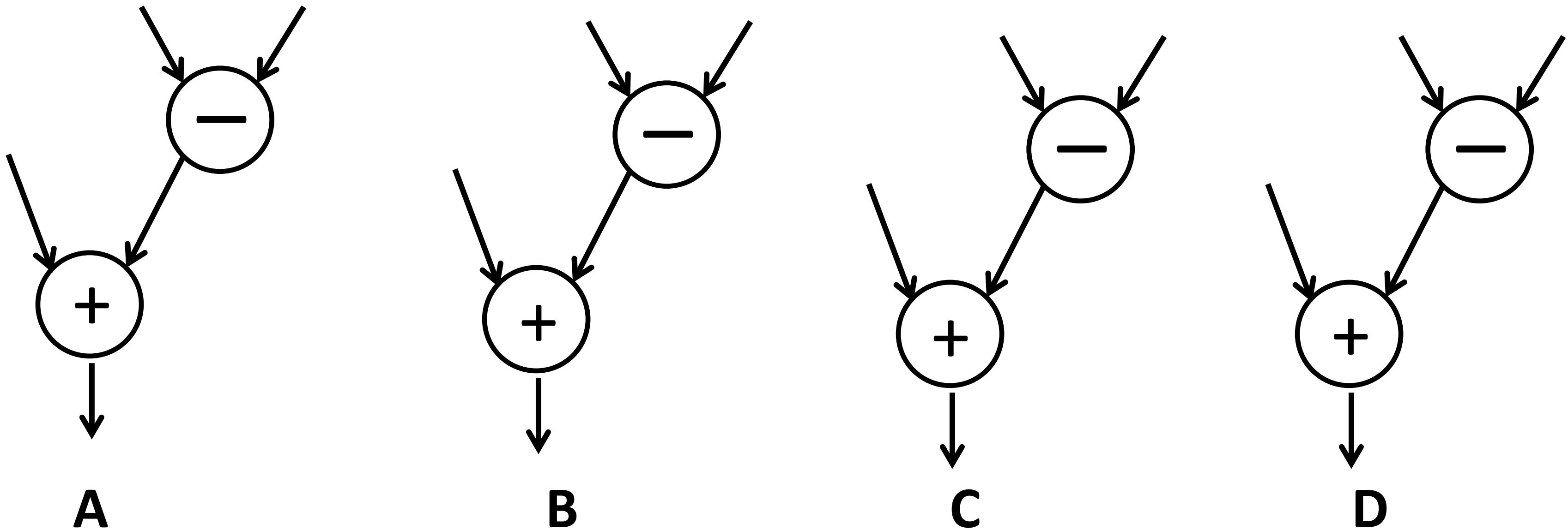
# 3. Decide which Pattern Instances to Share

- So far, steps 1 and 2 have provided sets of equivalent pattern graphs
- For example, we may have found 4 graphs for this pattern:



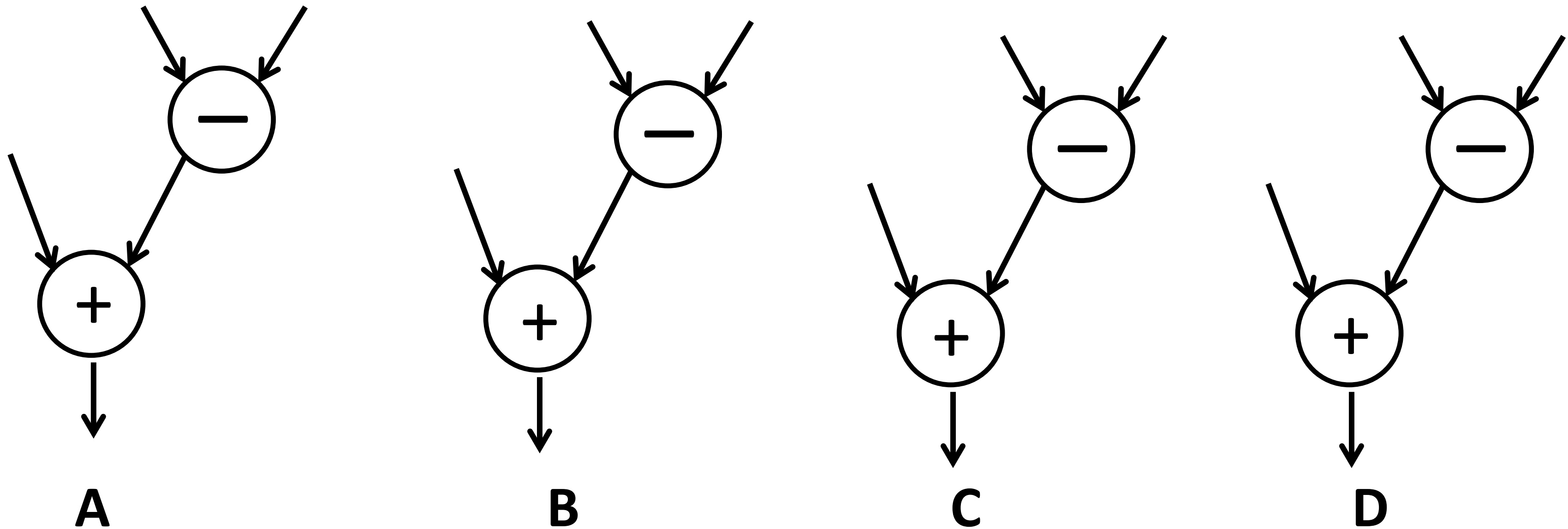
# 3. Decide which Pattern Instances to Share

- So far, steps 1 and 2 have provided sets of equivalent pattern graphs
- For example, we may have found 4 graphs for this pattern:



# 3. Decide which Pattern Instances to Share

- So far, steps 1 and 2 have provided sets of equivalent pattern graphs
- For example, we may have found 4 graphs for this pattern:

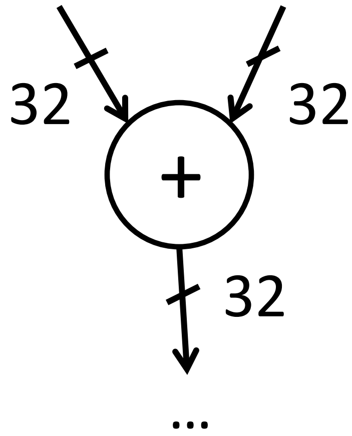


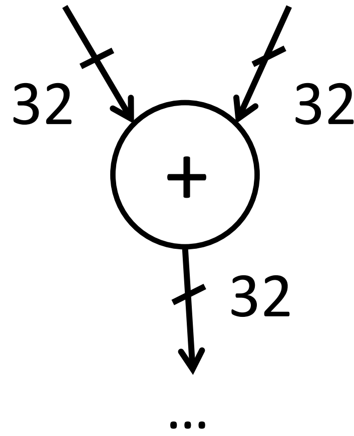

- Our goal is to <u>split these 4 into pairs</u> (create groups of 2) so that each hardware unit will implement two patterns

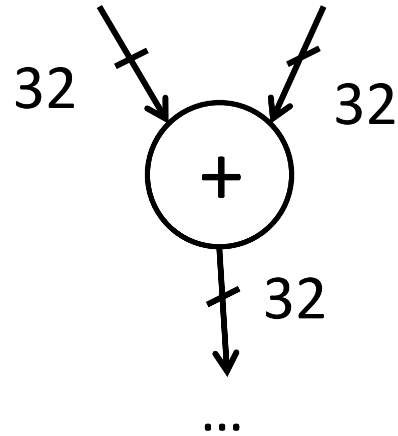
# 3. Decide which Pattern Instances to Share

- But which combination of pairs is best?
- Consider the *bit widths* of the operators

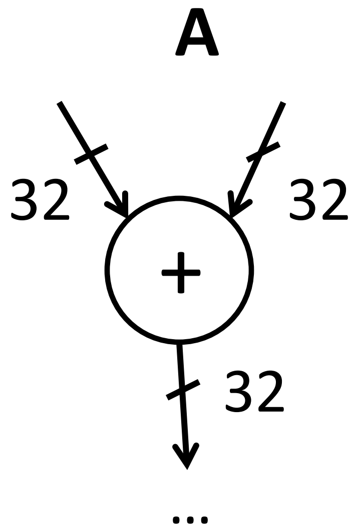
# Operator Bit Widths

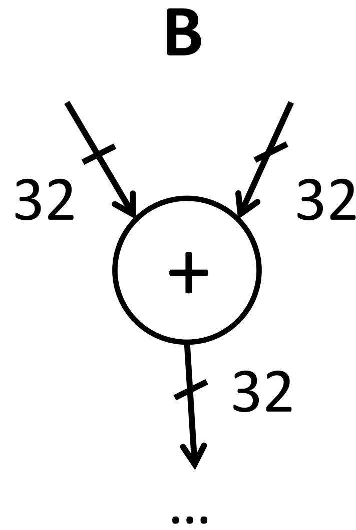


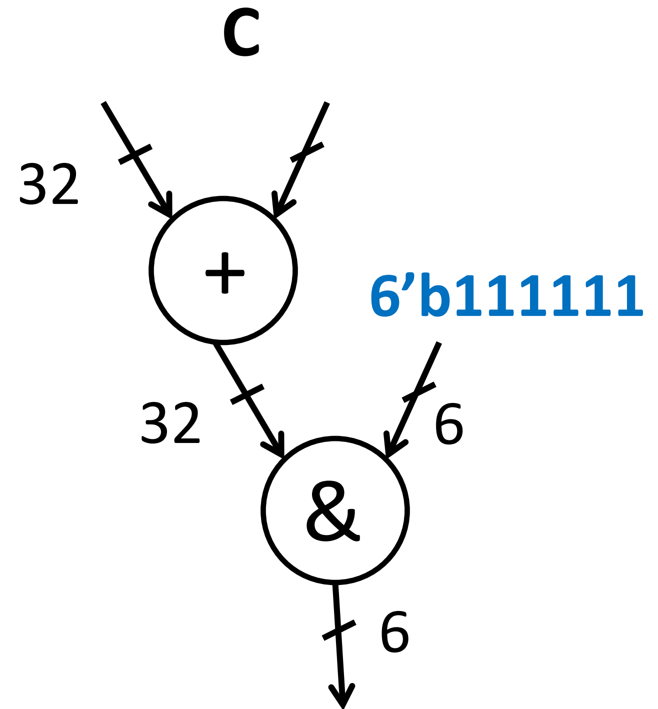



# Operator Bit Widths

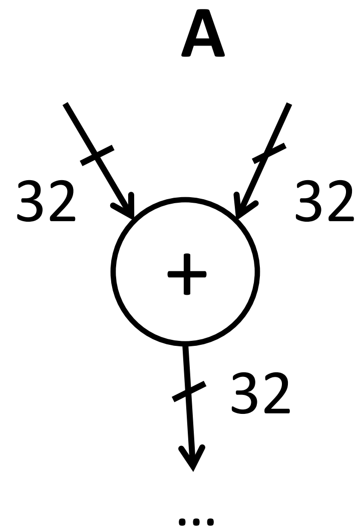


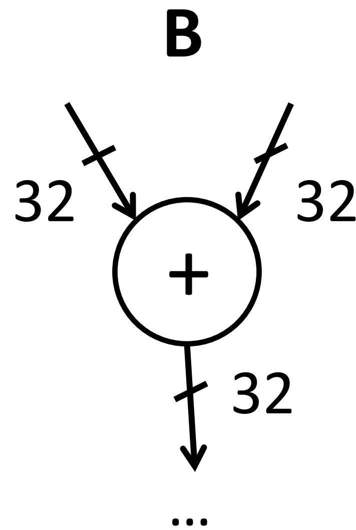



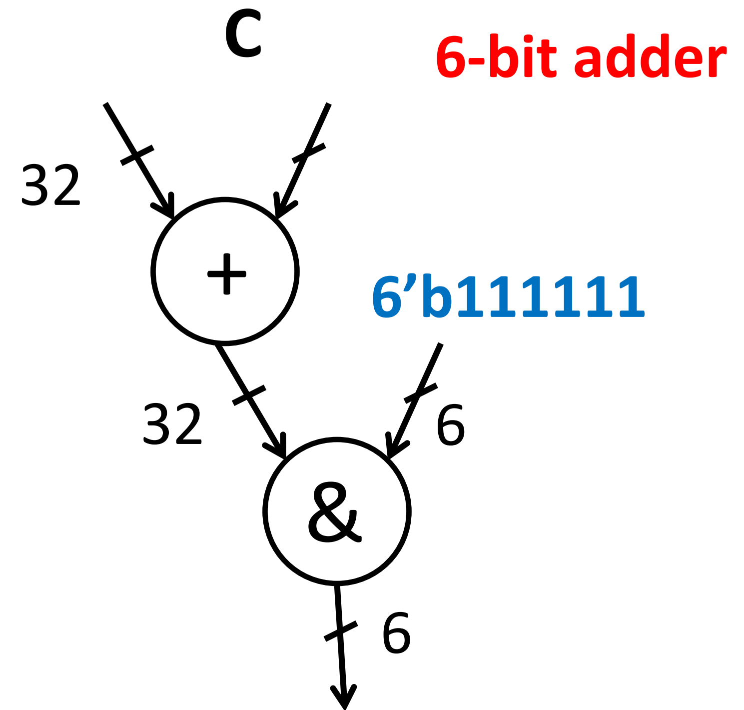




**Mask first six bits**

# Operator Bit Widths

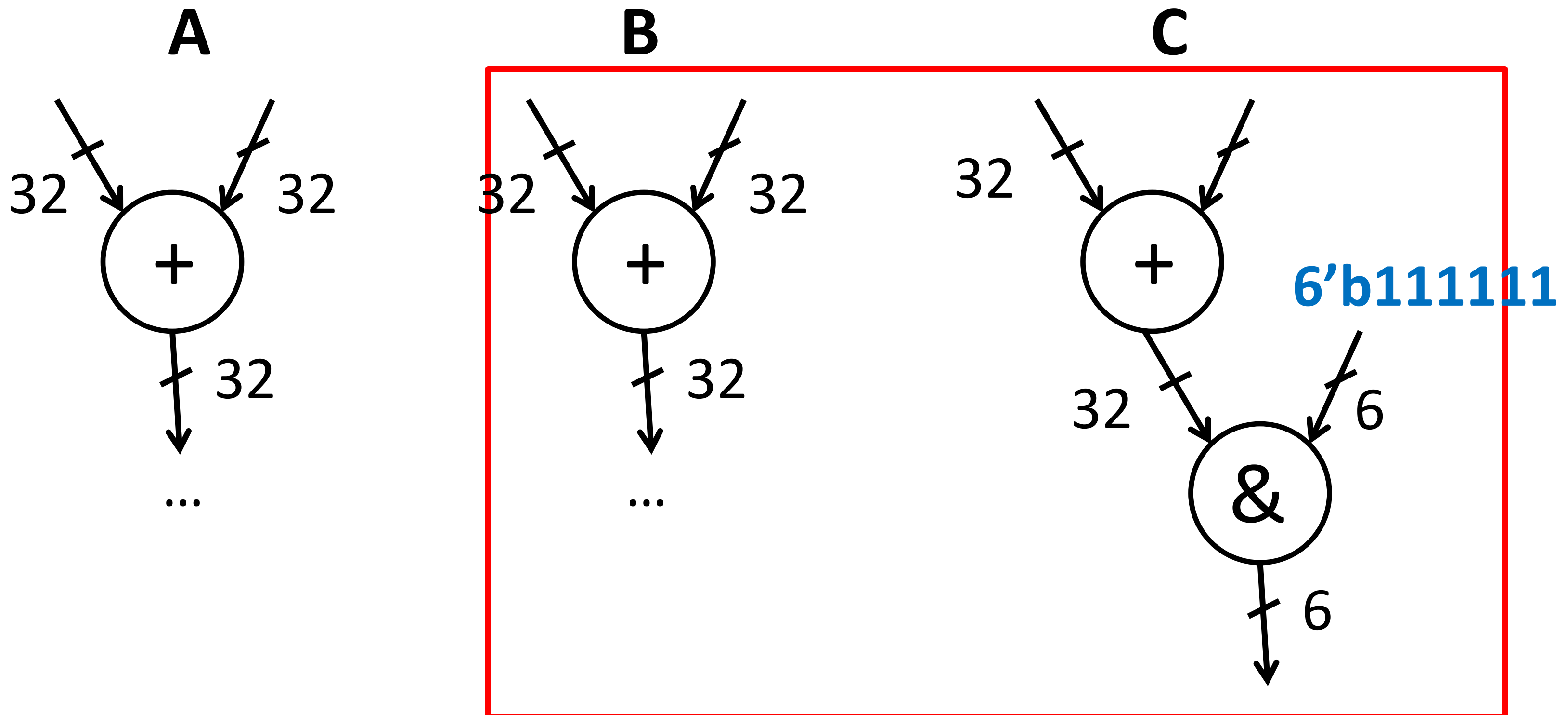


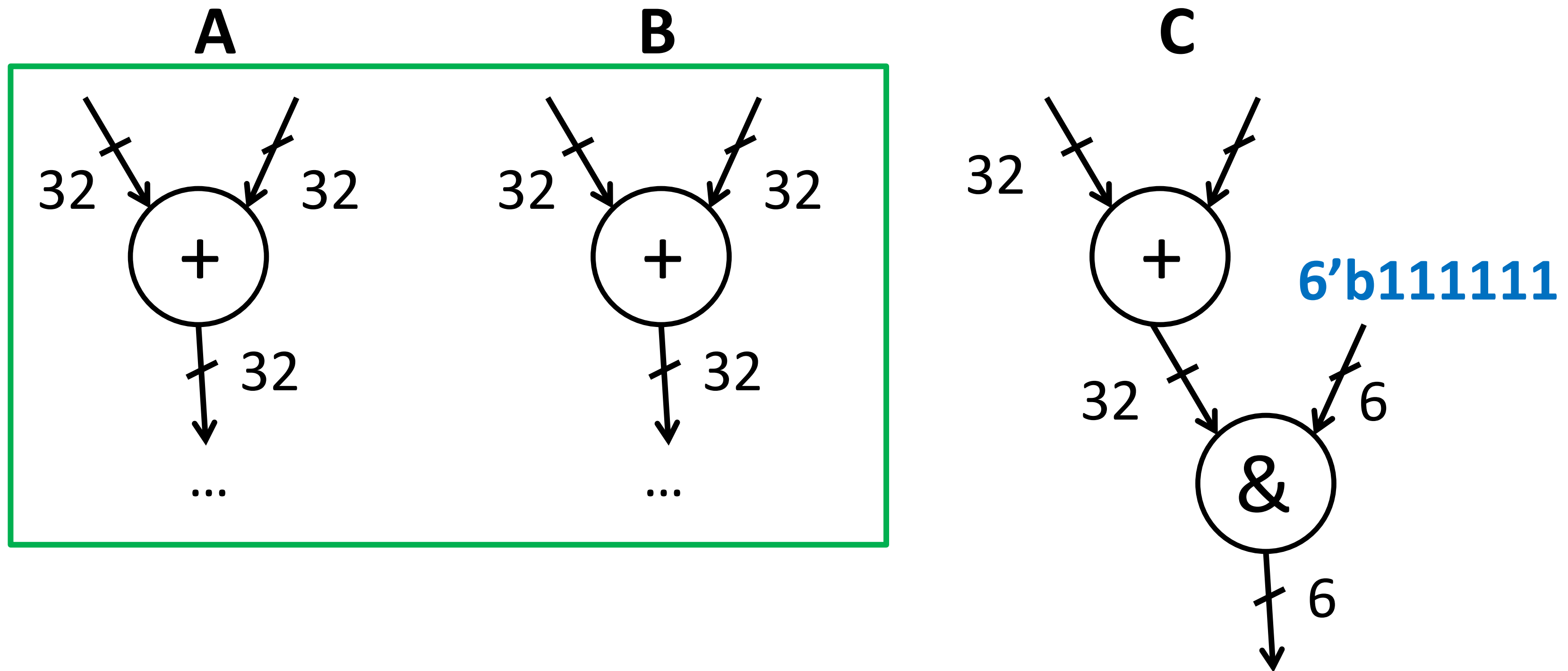

- Adder C would be optimized by synthesis tools because only 6 outputs bits are needed
- Sharing adder C with A or B would *force* a 6-bit addition to be implemented using a 32-bit adder

# Operator Bit Widths




- Adder C would be optimized by synthesis tools because only 6 outputs bits are needed
- Sharing adder C with A or B would *force* a 6-bit addition to be implemented using a 32-bit adder

# Operator Bit Widths




- Adder C would be optimized by synthesis tools because only 6 outputs bits are needed
- Sharing adder C with A or B would *force* a 6-bit addition to be implemented using a 32-bit adder

# 3. Decide which Pattern Instances to Share

- **Cost function** for sharing two pattern graphs:

Given two pattern graphs P1 and P2 with nodes n1 and n2 respectively,

Sharing Cost of P1, P2= $\sum_{n1 \in P1, n2 \in P2} |width(n1) - width(n2)|$

# 3. Decide which Pattern Instances to Share

- **Cost function** for sharing two pattern graphs:

Given two pattern graphs P1 and P2 with nodes n1 and n2 respectively,

Sharing Cost of P1, P2= $\sum_{n1 \in P1, n2 \in P2} |width(n1) - width(n2)|$

- Cost is then adjusted based on **preferential sharing conditions**

# 3. Decide which Pattern Instances to Share

- **Cost function** for sharing two pattern graphs:

Given two pattern graphs P1 and P2 with nodes n1 and n2 respectively,

Sharing Cost of P1, P2= $\sum_{n1 \in P1, n2 \in P2} |width(n1) - width(n2)|$

- Cost is then adjusted based on **preferential sharing conditions**
- For each graph a **greedy algorithm** selects its sharing-partner with the lowest cost

# 3. Decide which Pattern Instances to Share

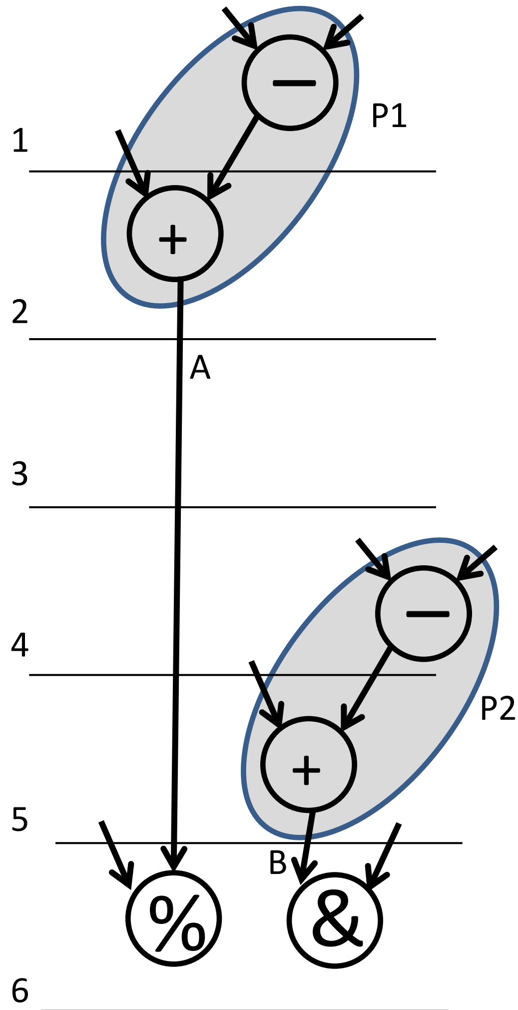
- **Cost function** for sharing two pattern graphs:

Given two pattern graphs P1 and P2 with nodes n1 and n2 respectively,

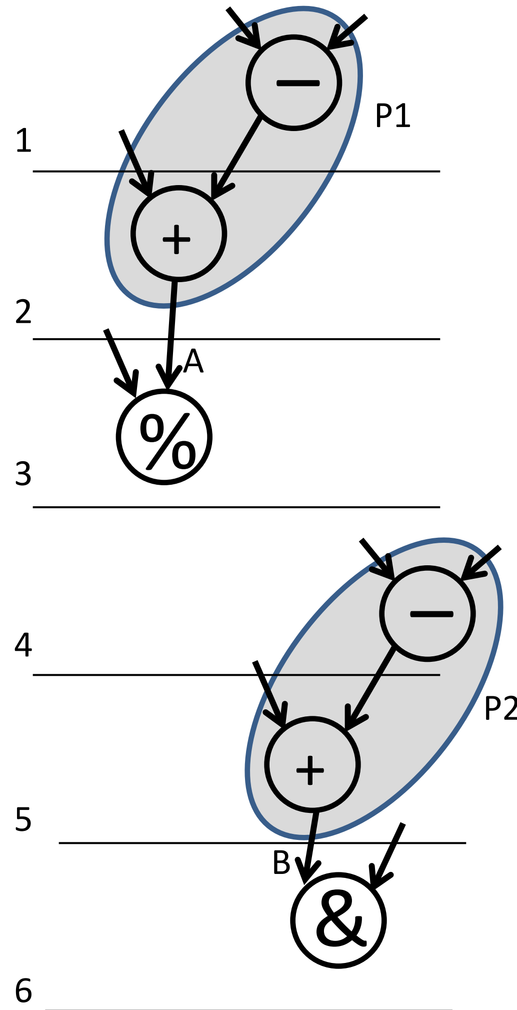
Sharing Cost of P1, P2= $\sum_{n1 \in P1, n2 \in P2} |width(n1) - width(n2)|$

- Cost is then adjusted based on **preferential sharing conditions**
- For each graph a **greedy algorithm** selects its sharing-partner with the lowest cost
- Once pairs are determined, the *Binding* phase of LegUp implements pairs with the same hardware

# Independent Variable Lifetimes



a) Values A,B have overlapping lifetimes

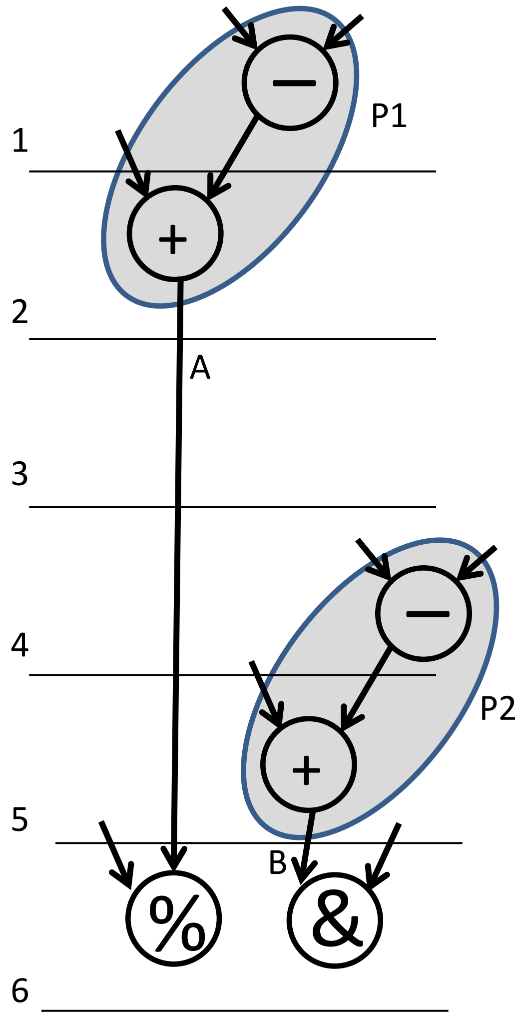


b) Values A,B have non-overlapping lifetimes

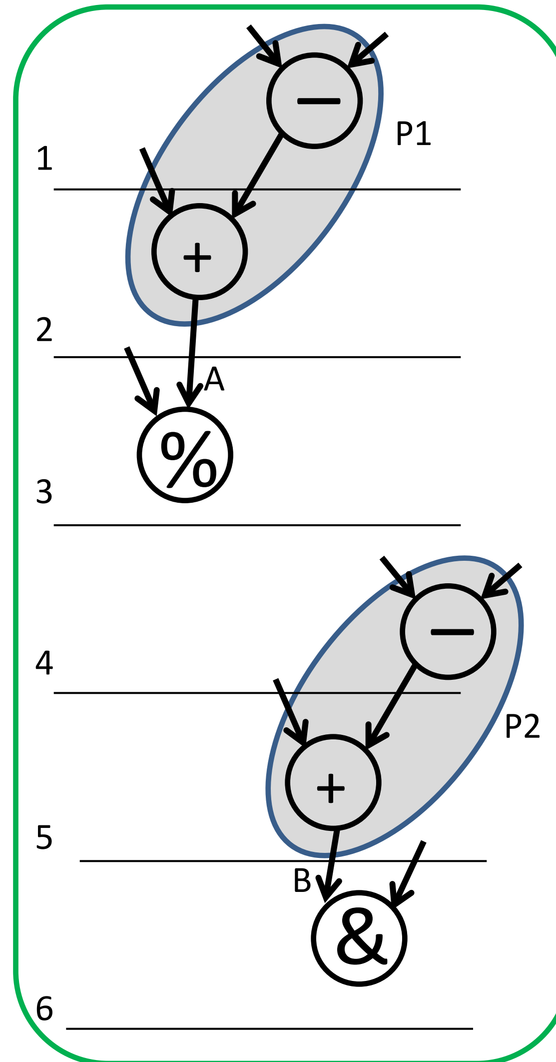
Prefer to share patterns with non-overlapping lifetimes

– Saves registers.

# Independent Variable Lifetimes




a) Values A,B have overlapping lifetimes

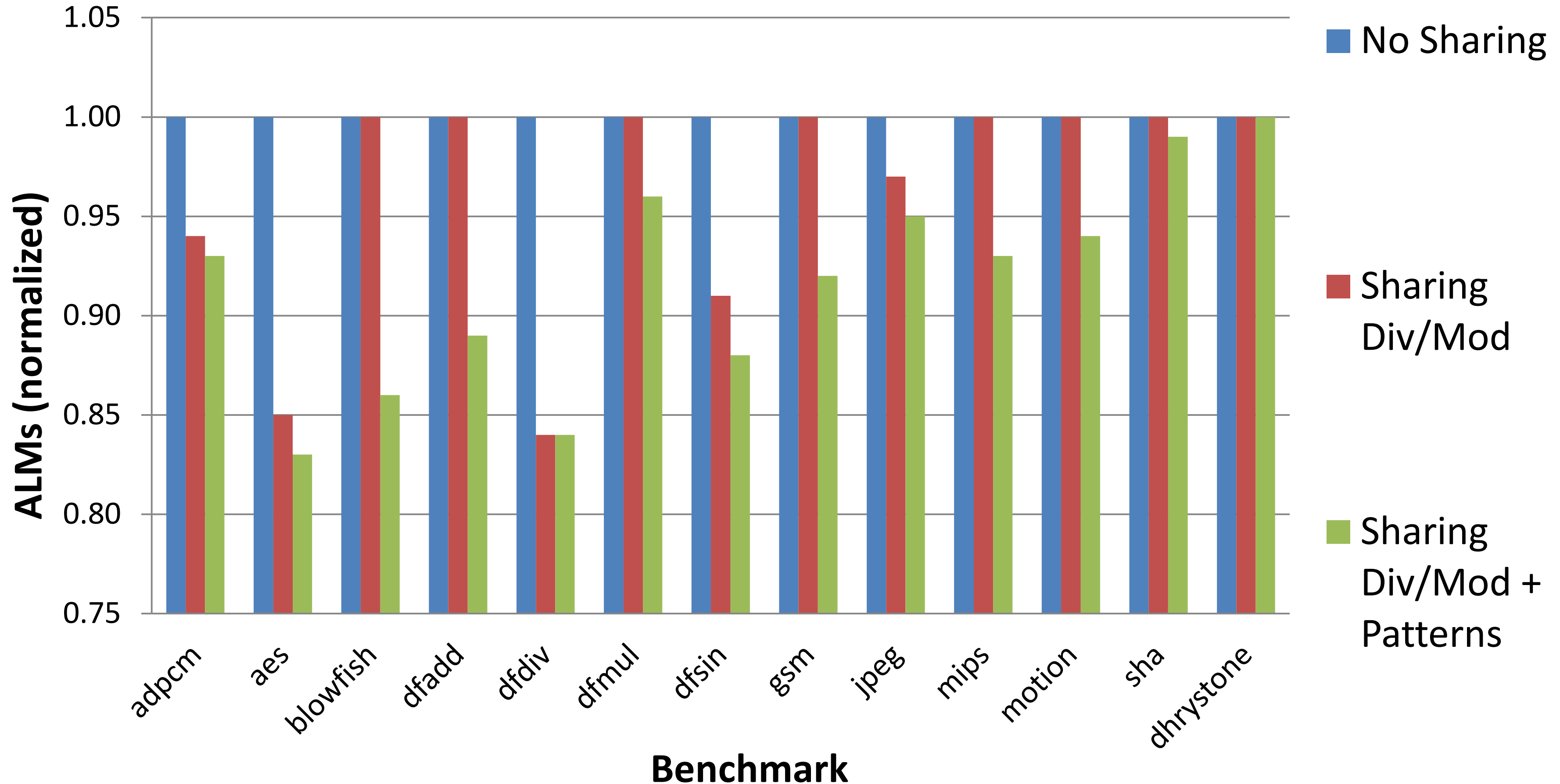



b) Values A,B have non-overlapping lifetimes

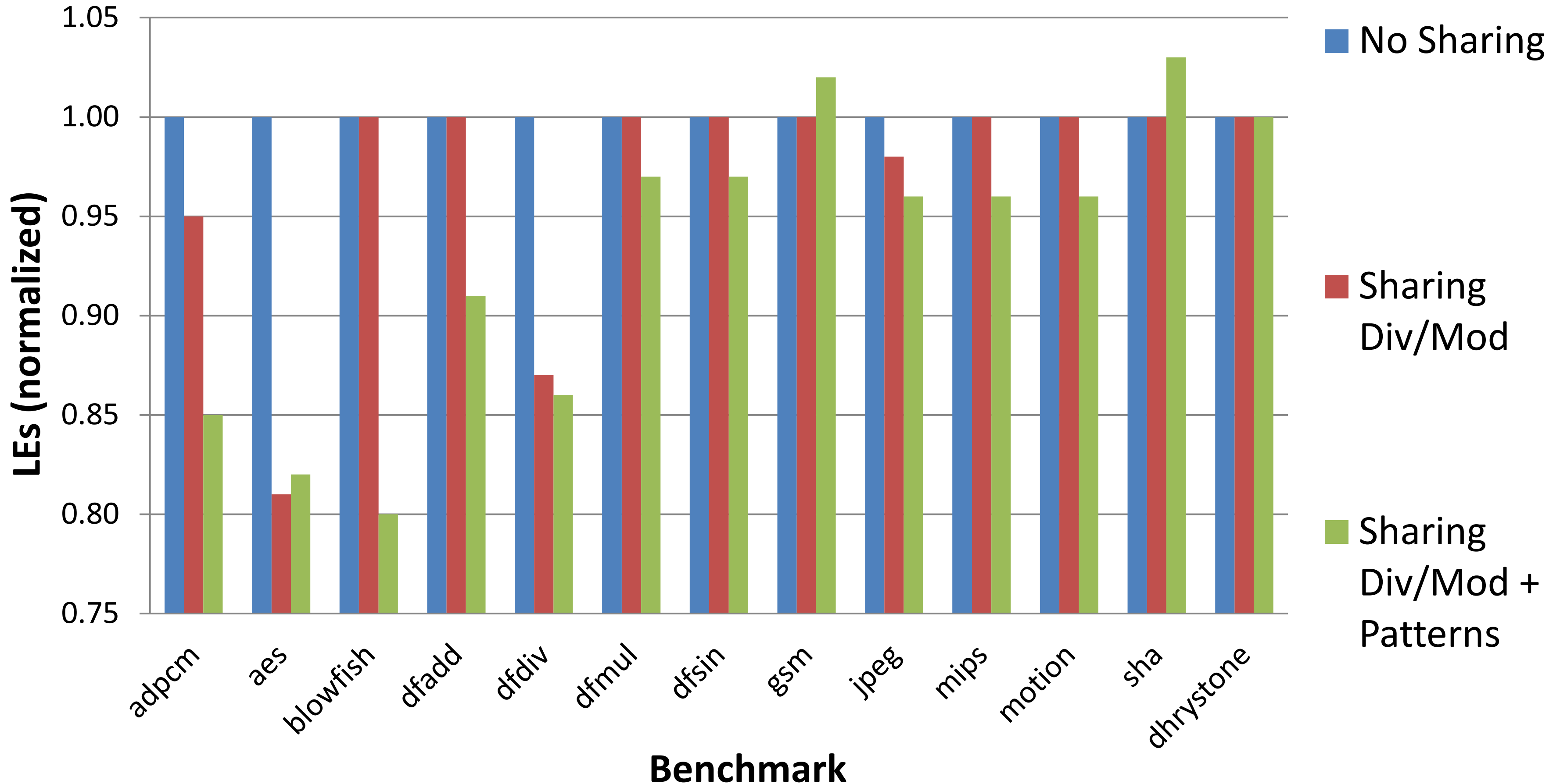
Prefer to share patterns with non-overlapping lifetimes

- Saves registers.

# Results




- 4% area reduction (geomean) for sharing dividers/modulus
- An additional **4.9%** reduction from sharing patterns
- 12% improvement when using LUT-based multipliers




- 3% area reduction (geomean) for sharing dividers/modulus
- An additional **4.2%** reduction from sharing patterns
- 16% improvement when using LUT-based multipliers

# Summary

- FPGA logic architecture has significant impact on resource sharing
- Pattern sharing can provide >10% area reduction
- Future work: alter scheduling to favor creation of certain patterns
  - Provide more sharing opportunities

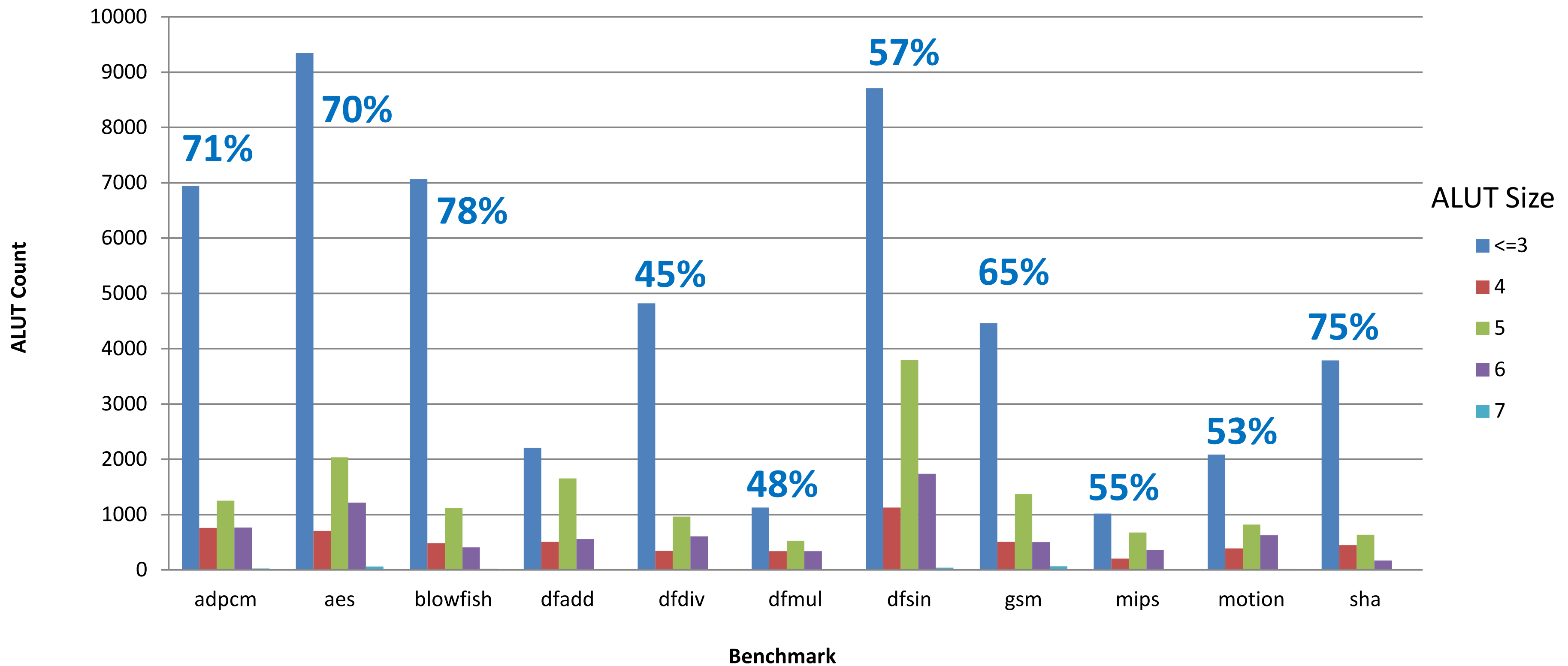
# Summary

- Questions?

# Extra Slides

# Motivation

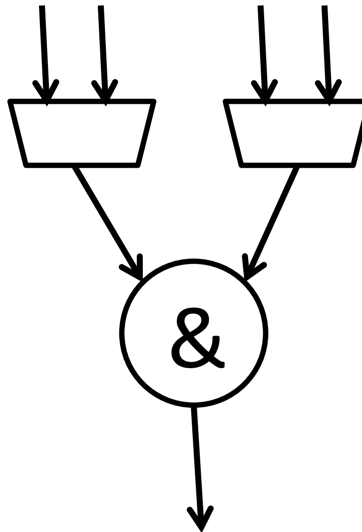
- Circuits created by LegUp use mostly 2 and 3 input LUTs




**Average: 62%**

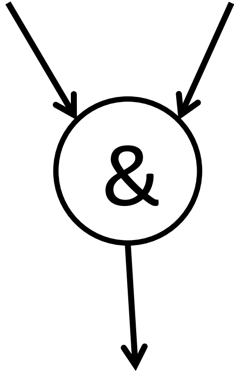
# Example – Sharing a Bitwise AND

This seems like a bad idea:



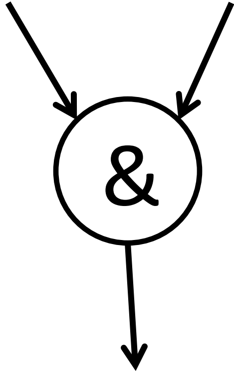
# Example – Sharing a Bitwise AND

Consider a Bitwise AND:



# Example – Sharing a Bitwise AND

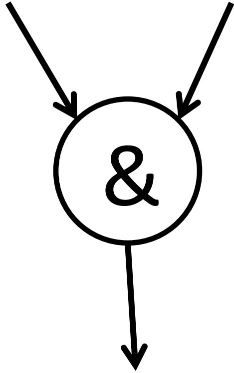
Consider a Bitwise AND:




2 Input LUT

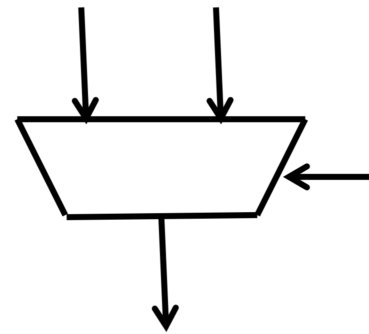
# Example – Sharing a Bitwise AND

Consider a Bitwise AND:




2 Input LUT

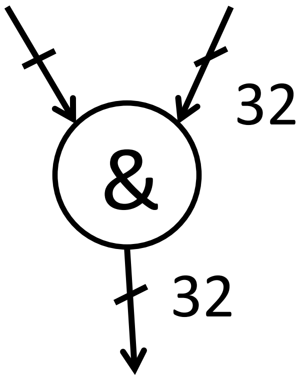
And a 2-to-1 MUX:



3-input LUT

# Example – Sharing a Bitwise AND

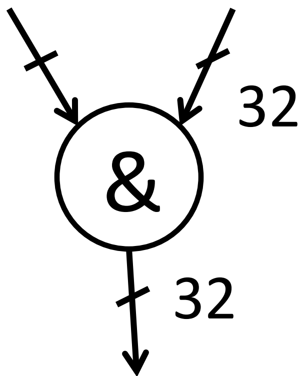
Consider a 32-bit Bitwise AND



# Example – Sharing a Bitwise AND

Consider a 32-bit Bitwise AND

- Requires 32 LUTs for 32 output bits

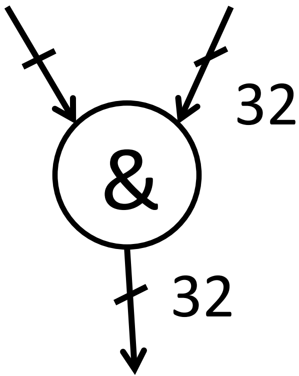



32 LUTs

# Example – Sharing a Bitwise AND

Consider a 32-bit Bitwise AND

- Requires 32 LUTs for 32 output bits

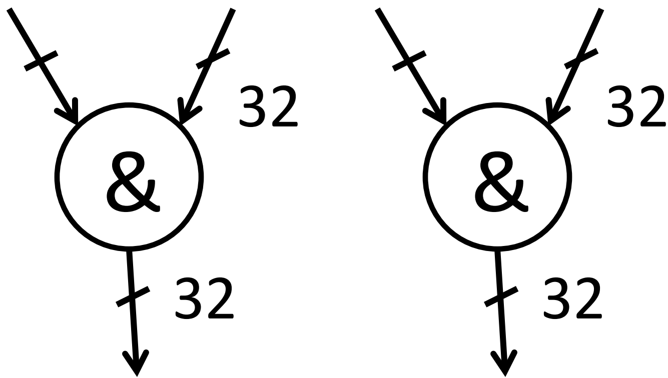



32 LUTs
(all 2-input LUTs)

# Example – Sharing a Bitwise AND

Consider a 32-bit Bitwise AND

- Requires 32 LUTs for 32 output bits

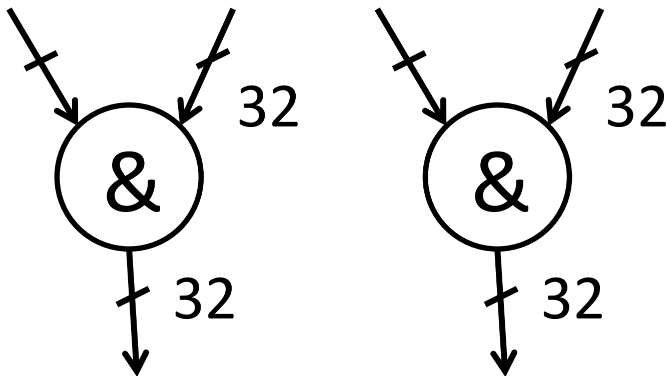


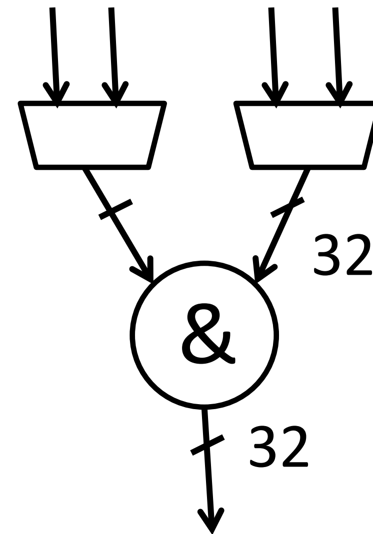

**64** LUTs
(all 2-input LUTs)

# Example – Sharing a Bitwise AND

Consider a 32-bit Bitwise AND

- Requires 32 LUTs for 32 output bits




**64** LUTs
(all 2-input LUTs)

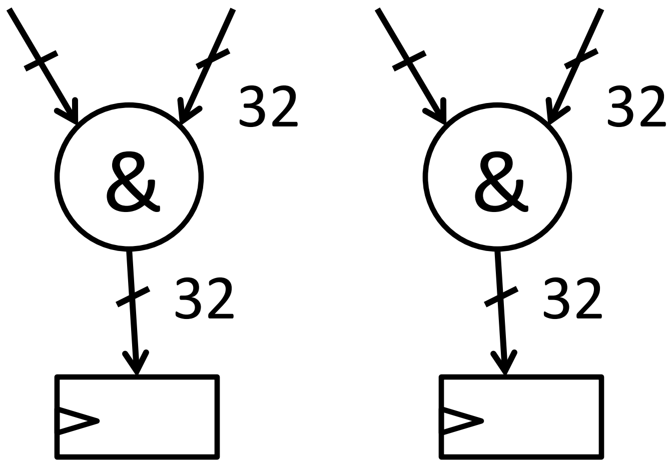
**32** LUTs
(**5**-input LUTs)

# Sharing Single Operations

- In the example of bitwise operations, we can reduce the number of LUTs by half at the expense of increasing their size
- However, if a circuits contains mostly small LUTs, ALMs are being under-utilized and can incorporate these larger logic functions
- Therefore, sharing even small operations reduces ALUT and ALM usage

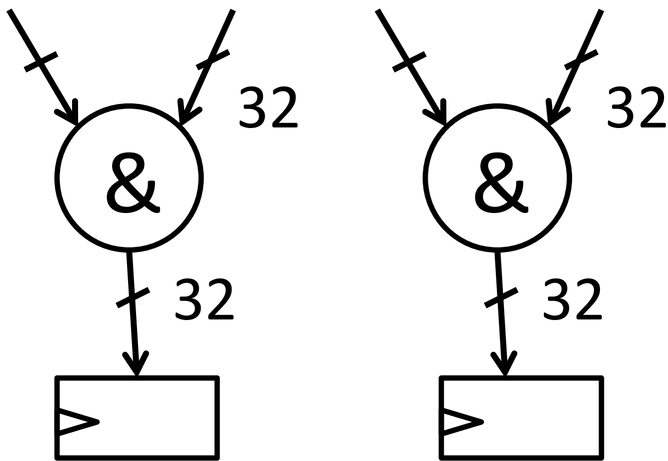
# Variable Liveness Analysis

- Consider next if each bitwise AND had its output stored in a register:



# Variable Liveness Analysis

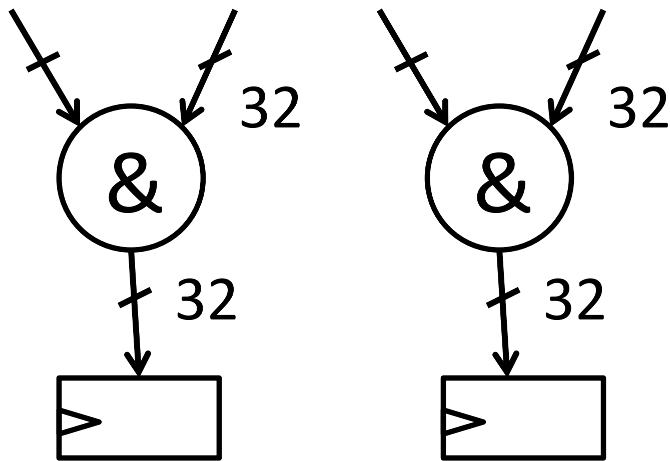
- Consider next if each bitwise AND had its output stored in a register:

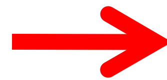


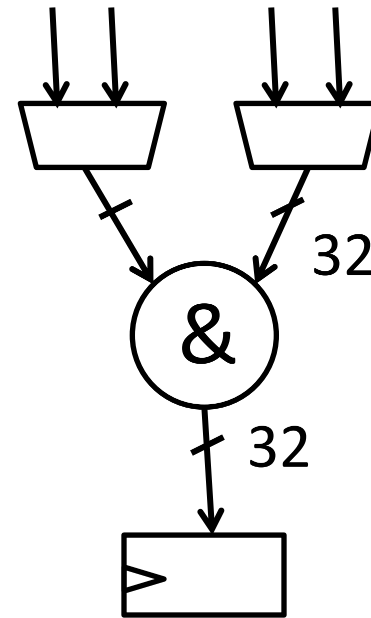

64 Registers

# Variable Liveness Analysis

- Consider next if each bitwise AND had its output stored in a register:



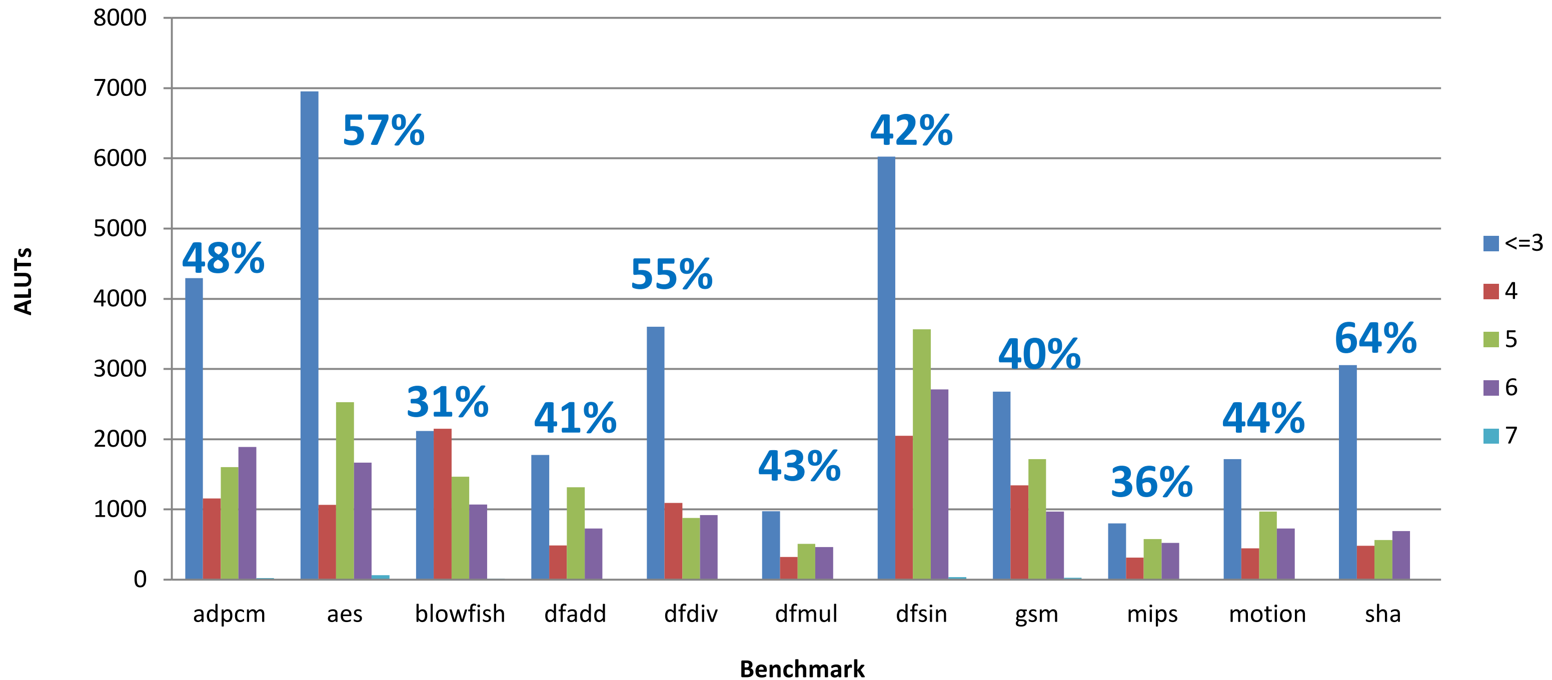






64 Registers

**32** Registers

(if lifetimes are independent)




**Average: 45%**

**(was 62%)**

**Table 5: Cyclone II resource sharing area results. Values in the table are LEs. Values in parentheses represent ratios relative to the no sharing case.**

| | Multiplication Using Embedded Mults | | | Multiplication Using LUT-Based Multipliers | | | |
|---|---|---|---|---|---|---|---|
| **Benchmark** | **No Sharing** | **Sharing Div/Mod** | **Sharing Div/Mod + Patterns** | **No Sharing** | **Sharing Div/Mod** | **Sharing Div/Mod + Mult** | **Sharing Div/Mod + Mult + Patterns** |
| **adpcm** | 22541 | 21476 (0.95) | 19049 (0.85) | 46702 | 45696 (0.98) | 23802 (0.51) | 24933 (0.53) |
| **aes** | 18923 | 15418 (0.81) | 15477 (0.82) | 18923 | 15418 (0.81) | 15418 (0.81) | 15342 (0.81) |
| **blowfish** | 11571 | 11571 (1.00) | 9306 (0.80) | 11571 | 11571 (1.00) | 11571 (1.00) | 9306 (0.80) |
| **dfadd** | 7012 | 7012 (1.00) | 6364 (0.91) | 7012 | 7012 (1.00) | 7012 (1.00) | 6258 (0.89) |
| **dfdiv** | 15286 | 13267 (0.87) | 13195 (0.86) | 22404 | 20421 (0.91) | 19217 (0.86) | 19151 (0.85) |
| **dfmul** | 3903 | 3903 (1.00) | 3797 (0.97) | 8669 | 8669 (1.00) | 8669 (1.00) | 8613 (0.99) |
| **dfsin** | 27860 | 27982 (1.00) | 26996 (0.97) | 40353 | 38449 (0.95) | 37277 (0.92) | 36407 (0.90) |
| **gsm** | 10479 | 10479 (1.00) | 10659 (1.02) | 18203 | 18203 (1.00) | 13584 (0.75) | 13762 (0.76) |
| **jpeg** | 35792 | 34981 (0.98) | 34316 (0.96) | 49218 | 48388 (0.98) | 38755 (0.79) | 38273 (0.78) |
| **mips** | 3103 | 3103 (1.00) | 2986 (0.96) | 5732 | 5732 (1.00) | 4377 (0.76) | 4114 (0.72) |
| **motion** | 4049 | 4049 (1.00) | 3897 (0.96) | 4049 | 4049 (1.00) | 4036 (1.00) | 4228 (1.04) |
| **sha** | 11932 | 11932 (1.00) | 12307 (1.03) | 11932 | 11932 (1.00) | 12069 (1.01) | 12449 (1.04) |
| **dhrystone** | 5277 | 5277 (1.00) | 5277 (1.00) | 5291 | 5291 (1.00) | 5351 (1.01) | 5351 (1.01) |
| **Geomean:** | **10419.82** | **10093.65** | **9677.25** | **13921.41** | **13515.54** | **12034.45** | **11752.99** |
| **Ratio:** | **1.00** | **0.97** | **0.93** | **1.00** | **0.97** | **0.86** | **0.84** |
| **Ratio:** | | **1.00** | **0.96** | | **1.00** | **0.89** | **0.87** |
| **Ratio:** | | | | | | **1.00** | **0.98** |

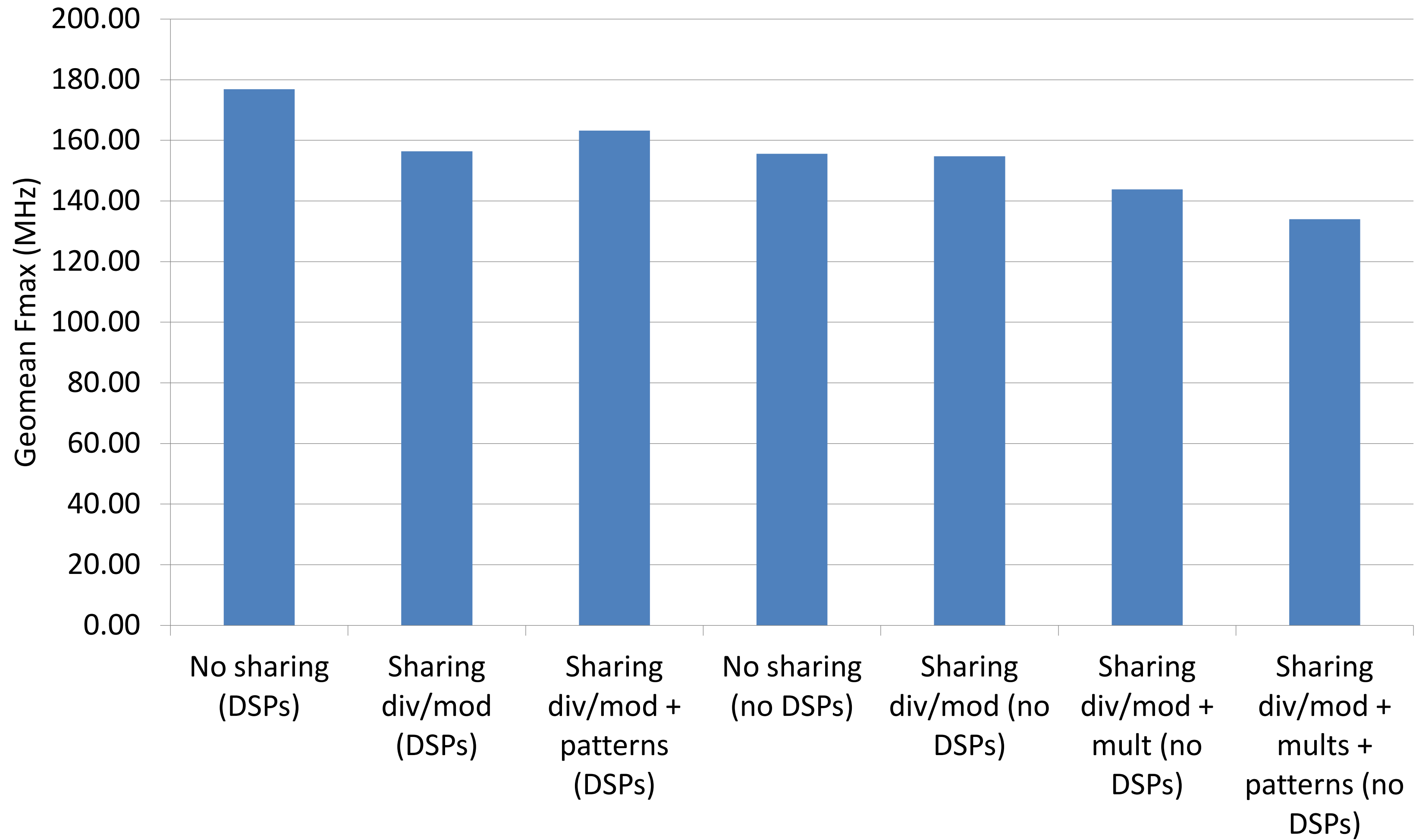
**Table 6: Stratix IV resource sharing area results. Values in the table are ALMs. Values in parentheses represent ratios relative to the no sharing case.**

| | Multiplication Using DSP Blocks | | | Multiplication Using LUT-Based Multipliers | | | |
|---|---|---|---|---|---|---|---|
| **Benchmark** | **No Sharing** | **Sharing Div/Mod** | **Sharing Div/Mod + Patterns** | **No Sharing** | **Sharing Div/Mod** | **Sharing Div/Mod + Mult** | **Sharing Div/Mod + Mult + Patterns** |
| **adpcm** | 8585 | 8064 (0.94) | 7943 (0.93) | 18951 | 18438 (0.97) | 11909 (0.63) | 11722 (0.62) |
| **aes** | 9582 | 8136 (0.85) | 7929 (0.83) | 9582 | 8136 (0.85) | 8136 (0.85) | 7929 (0.83) |
| **blowfish** | 6082 | 6082 (1.00) | 5215 (0.86) | 6082 | 6082 (1.00) | 6082 (1.00) | 5215 (0.86) |
| **dfadd** | 3327 | 3327 (1.00) | 2966 (0.89) | 3327 | 3327 (1.00) | 3327 (1.00) | 2966 (0.89) |
| **dfdiv** | 7043 | 5949 (0.84) | 5915 (0.84) | 9352 | 8277 (0.89) | 8203 (0.88) | 8204 (0.88) |
| **dfmul** | 1893 | 1893 (1.00) | 1824 (0.96) | 3170 | 3170 (1.00) | 3170 (1.00) | 3105 (0.98) |
| **dfsin** | 12630 | 11529 (0.91) | 11094 (0.88) | 16631 | 15418 (0.93) | 15523 (0.93) | 15129 (0.91) |
| **gsm** | 4914 | 4914 (1.00) | 4537 (0.92) | 7630 | 7630 (1.00) | 6252 (0.82) | 6043 (0.79) |
| **jpeg** | 17148 | 16703 (0.97) | 16246 (0.95) | 22349 | 21853 (0.98) | 19592 (0.88) | 19127 (0.86) |
| **mips** | 1610 | 1610 (1.00) | 1493 (0.93) | 2471 | 2471 (1.00) | 2299 (0.93) | 2210 (0.89) |
| **motion** | 1988 | 1988 (1.00) | 1878 (0.94) | 1988 | 1988 (1.00) | 1982 (1.00) | 1930 (0.97) |
| **sha** | 5909 | 5909 (1.00) | 5856 (0.99) | 5909 | 5909 (1.00) | 5947 (1.01) | 5917 (1.00) |
| **dhrystone** | 2598 | 2598 (1.00) | 2598 (1.00) | 2602 | 2602 (1.00) | 2607 (1.00) | 2607 (1.00) |
| **Geomean:** | **4980.59** | **4788.06** | **4558.11** | **6273.87** | **6078.47** | **5709.30** | **5499.92** |
| **Ratio:** | **1.00** | **0.96** | **0.92** | **1.00** | **0.97** | **0.91** | **0.88** |
| **Ratio:** | | **1.00** | **0.95** | | **1.00** | **0.94** | **0.90** |
| **Ratio:** | | | | | | **1.00** | **0.96** |

# Pattern Sharing Conclusions

- The most frequently occurring patterns in 13 HLS Benchmarks (CHStone Benchmark suite and dhrystone) were analyzed
- Benefits of pattern sharing improve as pattern size increases, but **LUT-underutilization** is the major factor
  - allows MUXes to be incorporated into the same LUTs as the operator
  - sharing is thus more advantageous if **registers** are present in patterns as they prevent an efficient mapping of operators into LUTs

# Stratix IV Speed Performance



# Cyclone II Speed Performance