# Configuration **Bitstream Reduction** for SRAM-based FPGAs by Enumerating LUT Input Permutations

Ameer Abdelhadi Guy Lemieux

The University of
British Columbia

# Modern FPGAs are HUGE!

- Altera Stratix IV “4SGX530” device (45nm)
  - 200k 6-LUTs in one die

- FPGAs keep getting bigger
  - Moore’s Law
    - 2x every 18-24 months
    - 28nm devices announced (Stratix V, Virtex 7)
  - More than Moore
    - Xilinx 1,200k 6-LUTs using stacked silicon

# Configuration Bits

- Growing **bitstream storage** & **load time**
  - Altera Stratix IV (4SGX, largest device)
    - **4 seconds** to boot up using serial EEPROM
    - **35 seconds** to configure using USB 2.0
    - **20 MByte** bitstream
- **Multiple configurations** per FPGA
  - Even more bitstream storage

# Bitstream Compression

- Obvious solution: **bitstream compression**
  - Many methods in research
    - Architecture-aware: switch blocks (eg, avoid 5:1 muxes), logic blocks (eg, ULMs, coarse-grained) , readback, frame reordering, wildcard techniques
    - Architecture-agnostic: general compression (eg: huffman, LZ, arithmetic), “don’t care”
  - **Not yet applied in practice**

- Can we do better?

# Main Idea

- **Bitstream Reduction**
  - **Throw away** redundant bits
    - Do not store them in configuration EEPROM
    - Do not transmit them
  - **Regenerate missing bits** (in the FPGA)
    - Done on-the-fly, during bitstream load
    - Does not reduce # SRAM bits inside FPGA

- This **is not** bitstream compression
  - Can still compress **after** reduction

# Main Idea

- Where to find redundant bits?
  - LUT configuration bits (this paper)
  - Interconnect (future work)
- Eg, 4-input LUT:
  - 16 config bits ($2^k$)
  - 65,536 configurations ($2^{(2^k)}$)
  - 222 unique (npn-equivalent) 4-input functions
  - Theoretically, 8 config bits is sufficient

# Equivalence Classes

- What is **npn**-equivalent ?

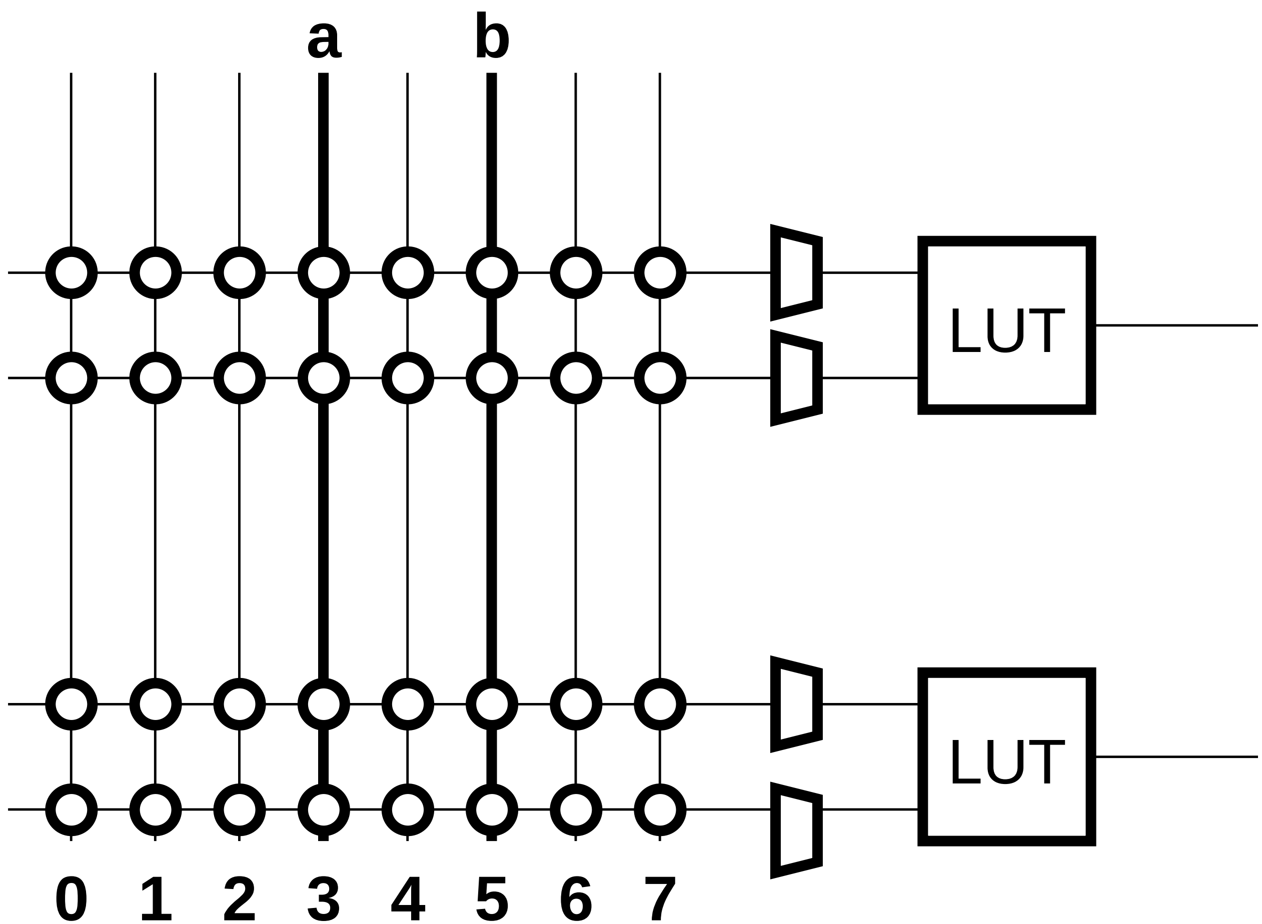
  First n = input negation (inversion),

  **p = input permutations**,

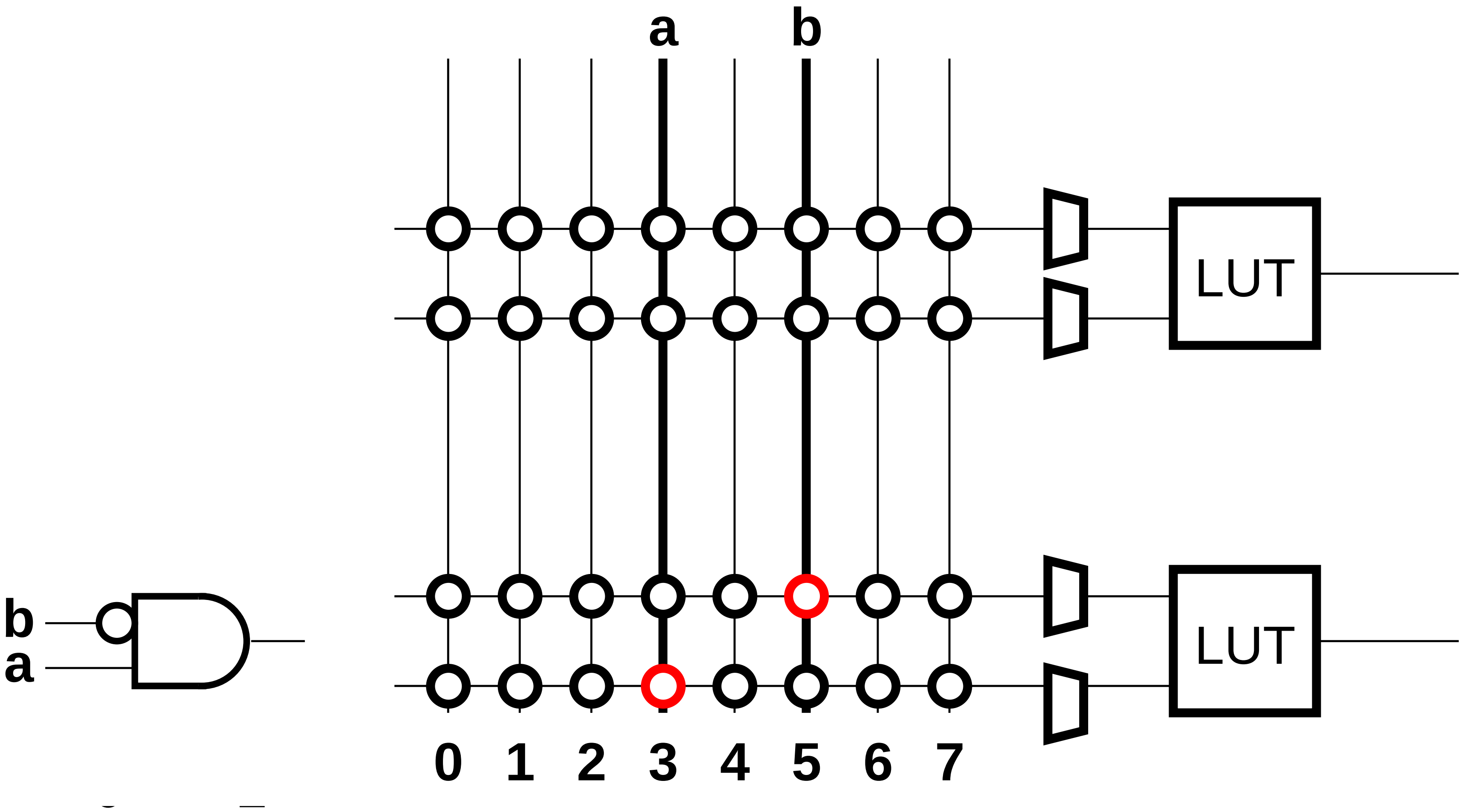
  Second n = output negation

- Eg, 4-input LUT
  - 4! = 24 input permutations for the same function
  - 2^k = 16 input inversions
  - 2^1 = 2 output inversions

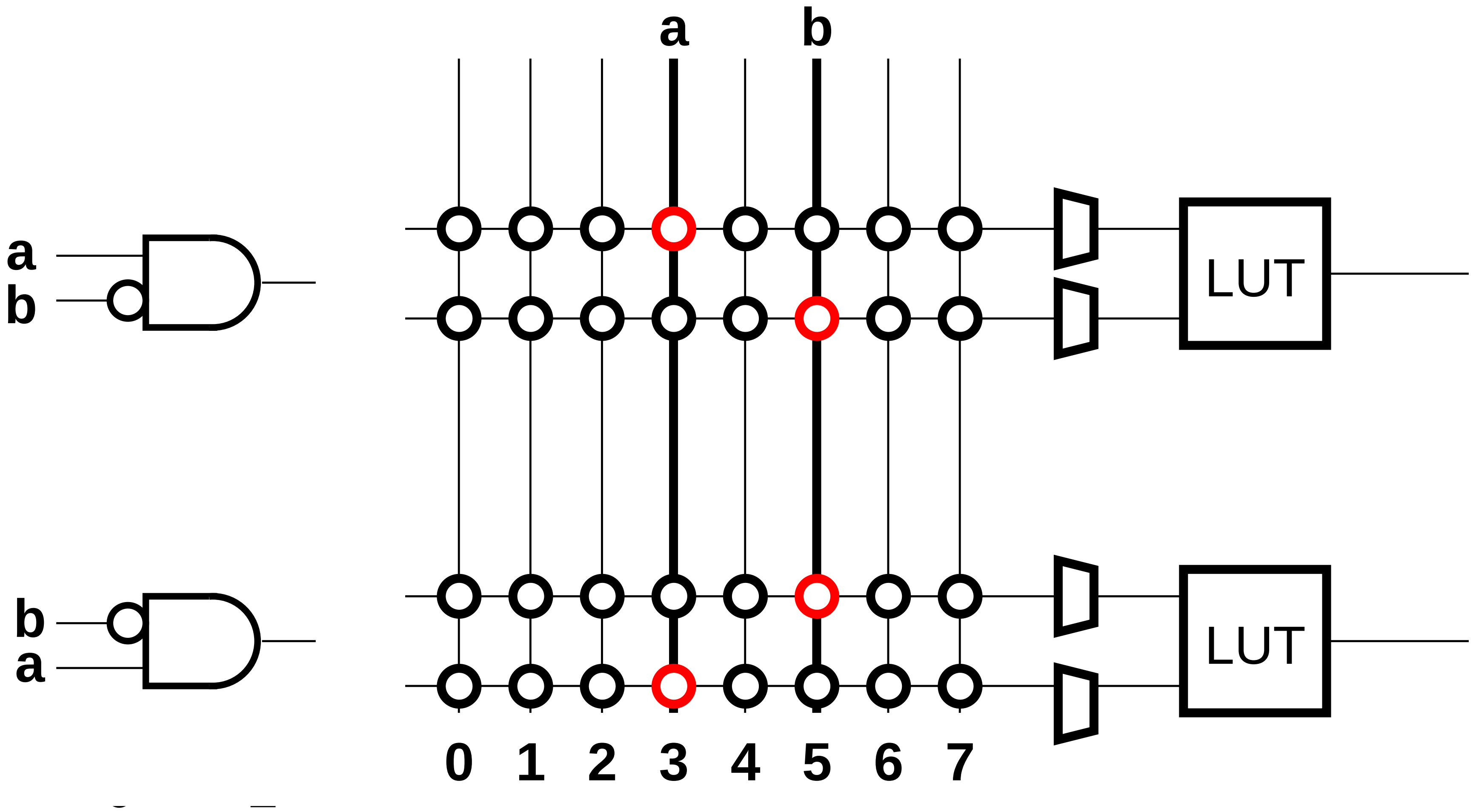
# Example: LUT input ordering



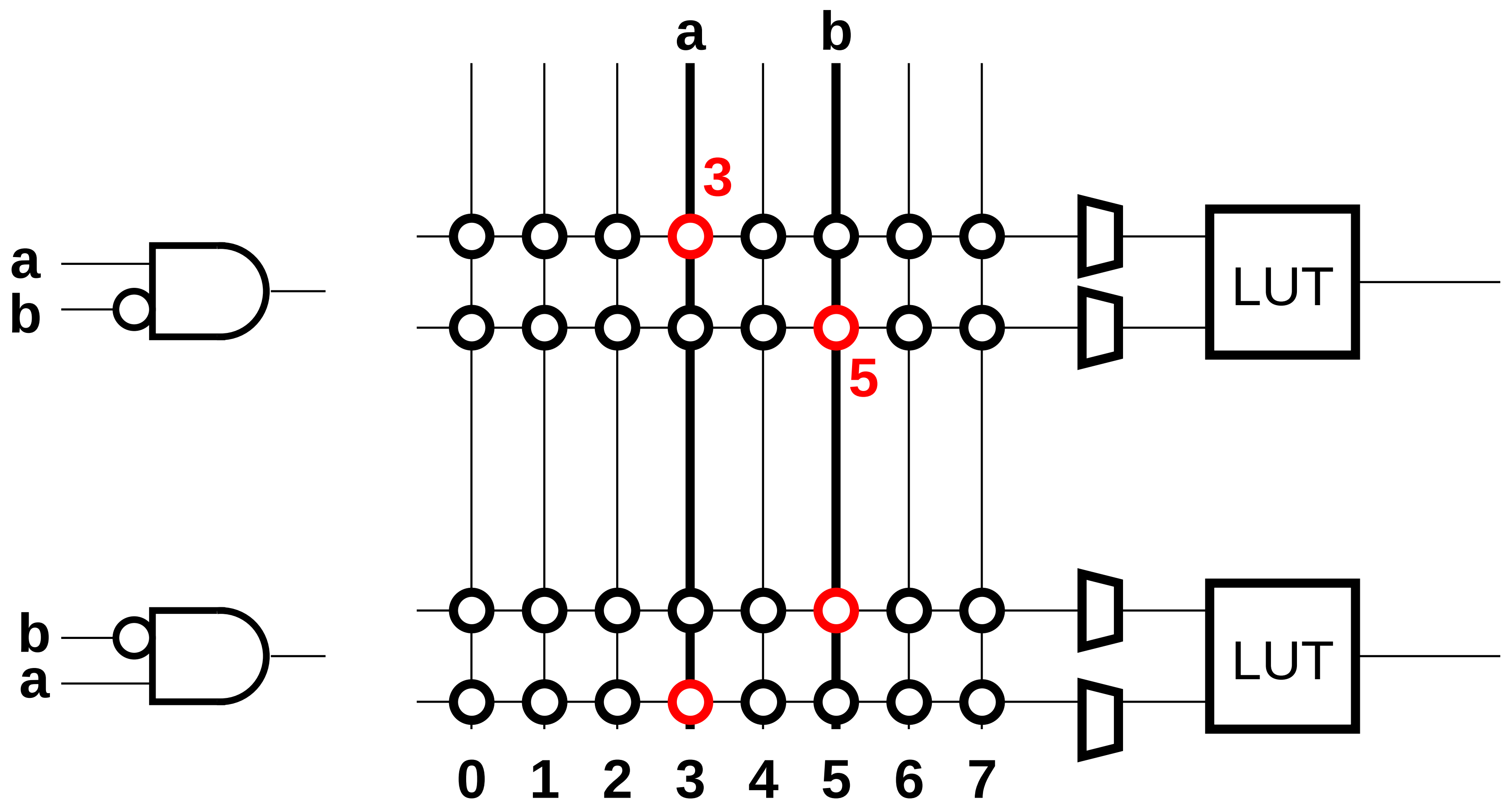
# Example: LUT input ordering



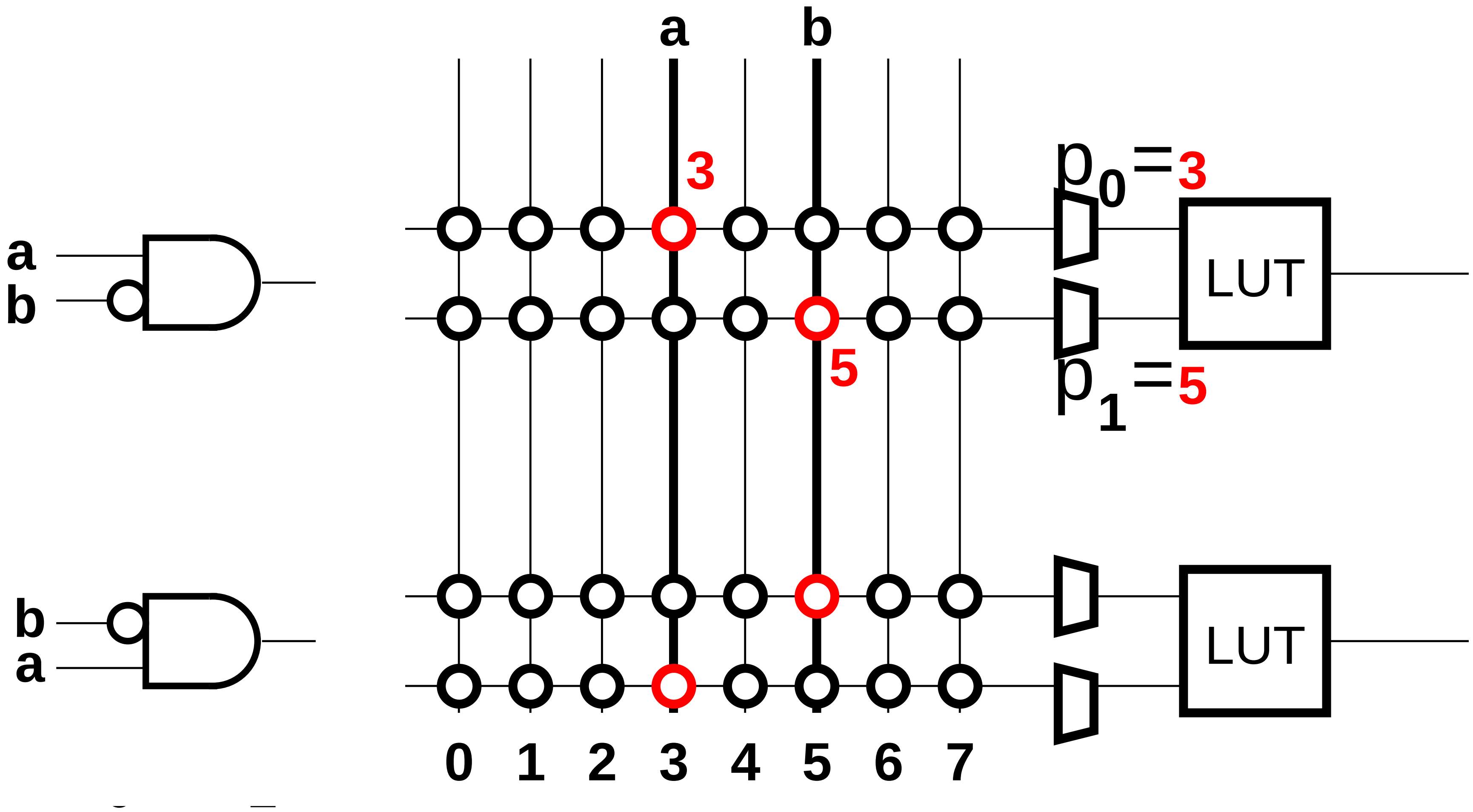
# Example: LUT input ordering



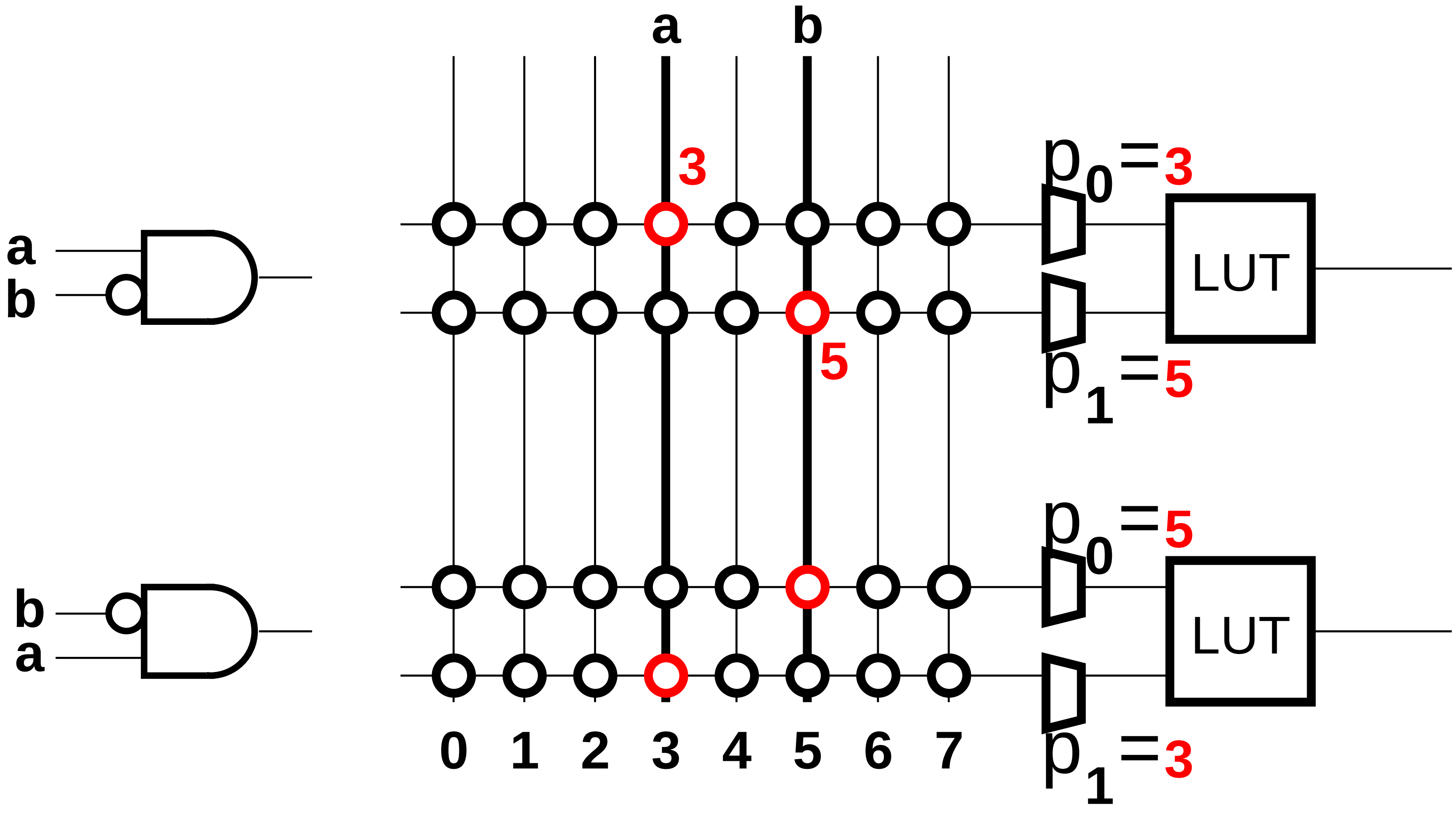
# Example: LUT input ordering



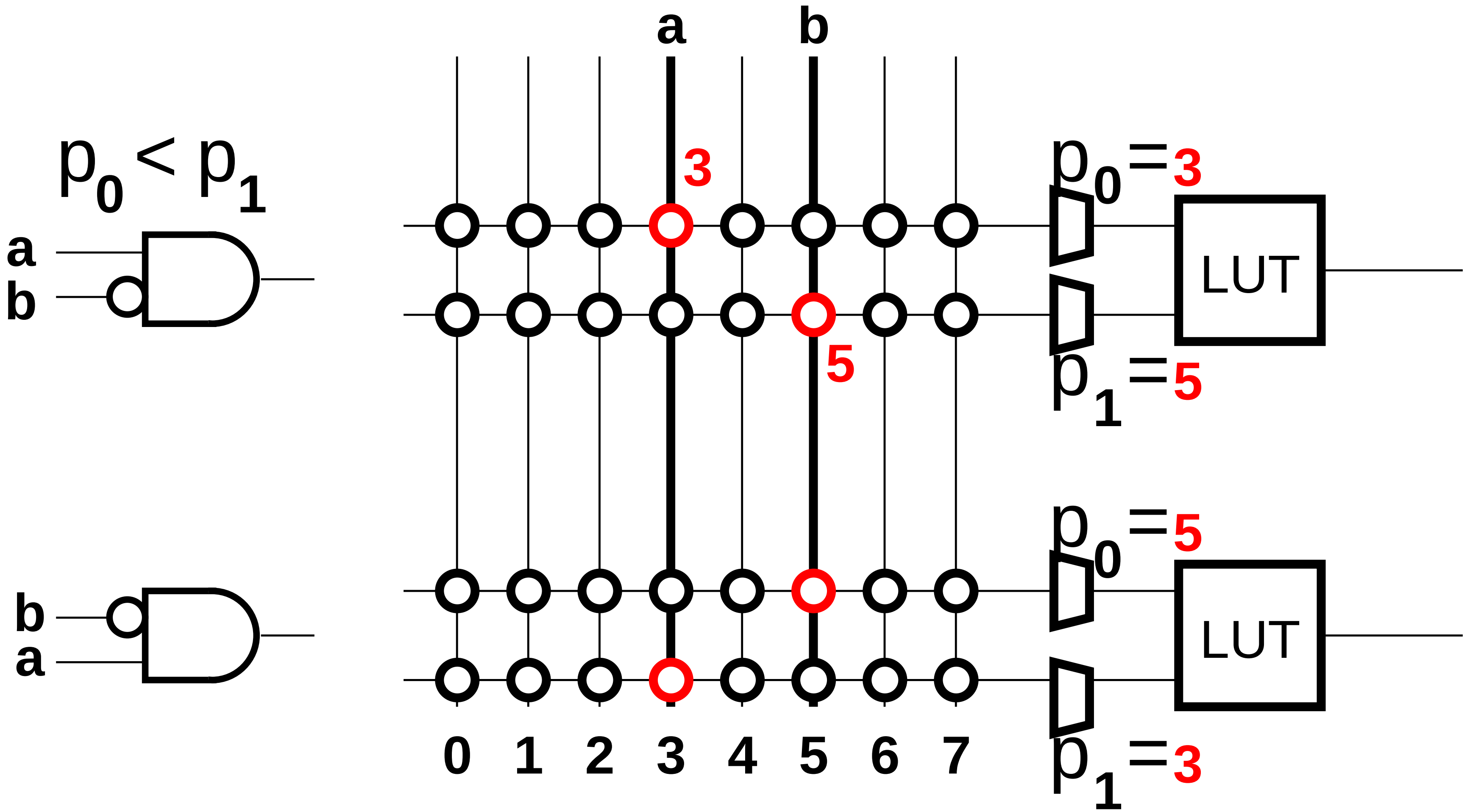
# Example: LUT input ordering



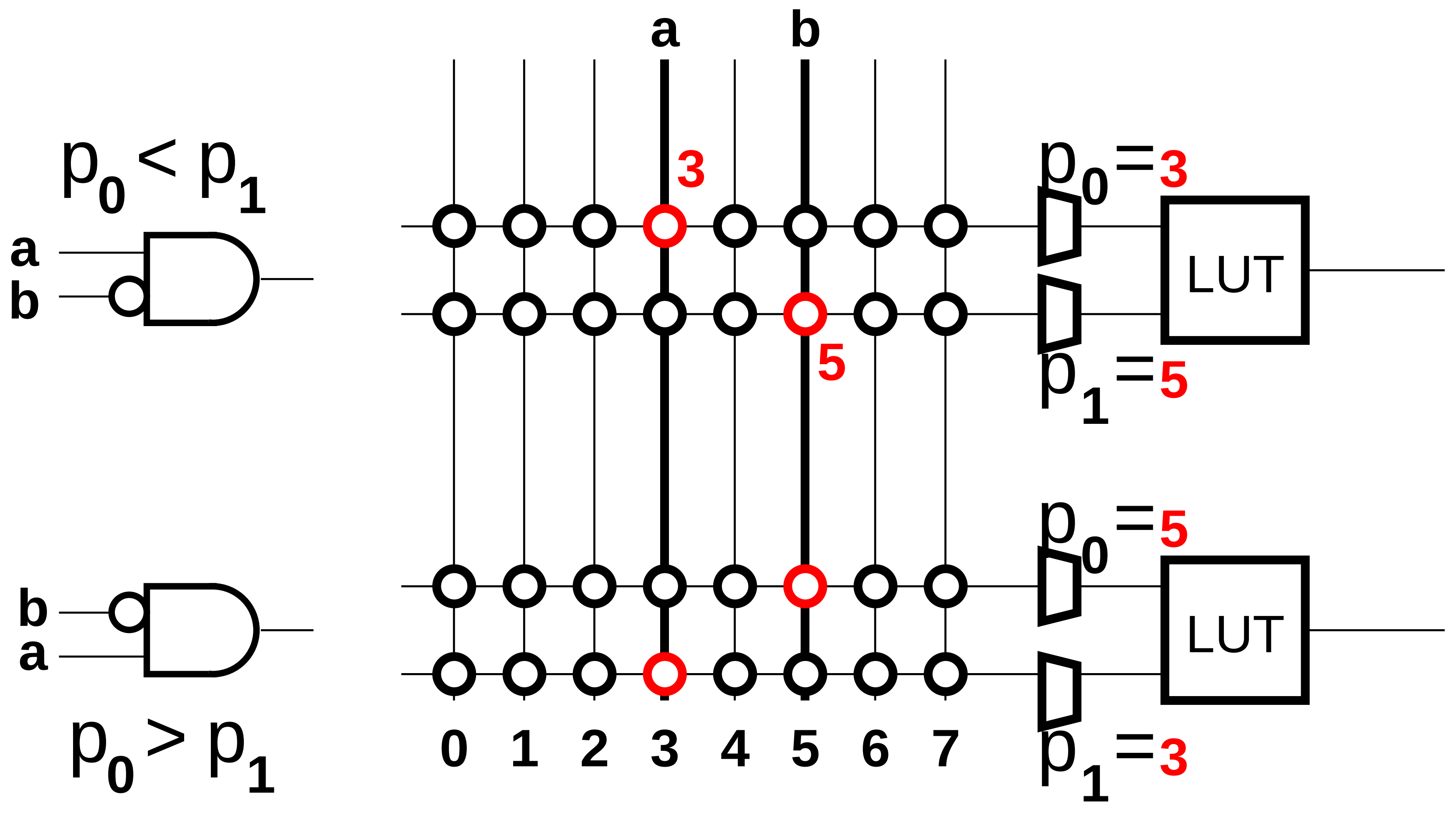
# Example: LUT input ordering



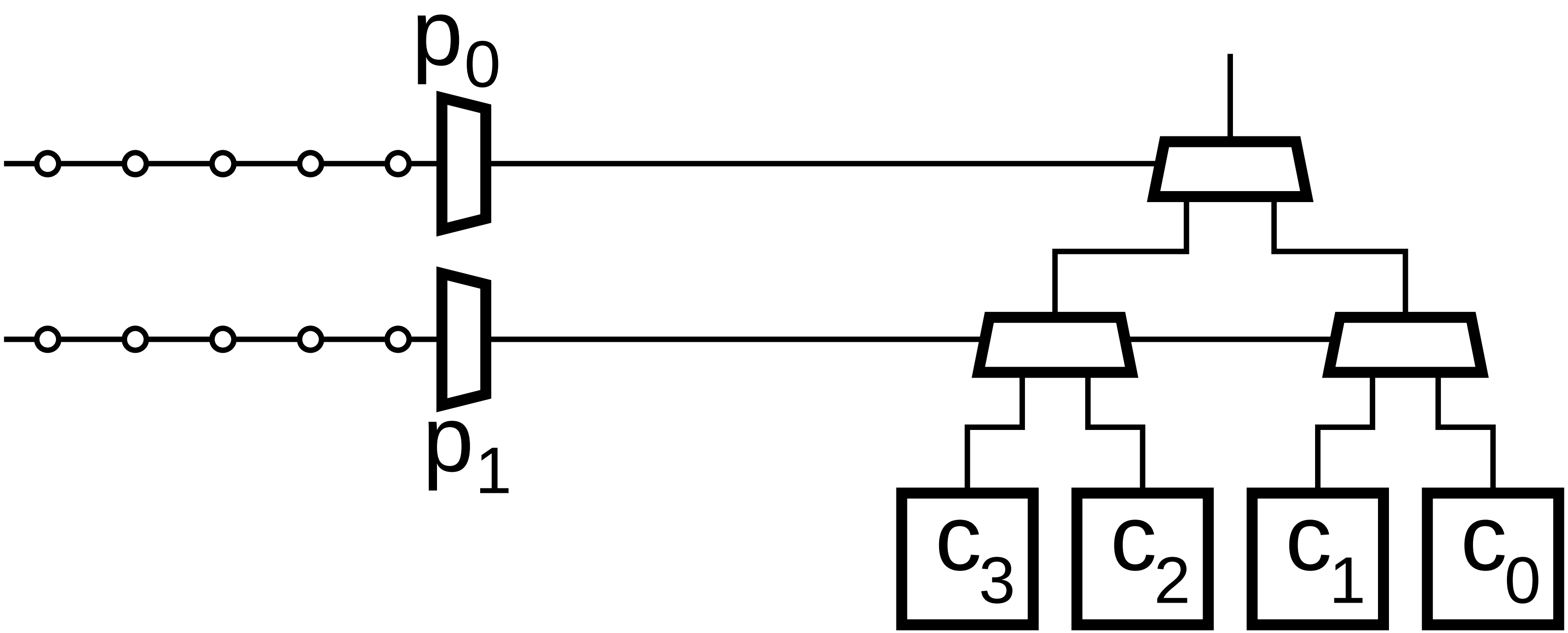
# Example: LUT input ordering



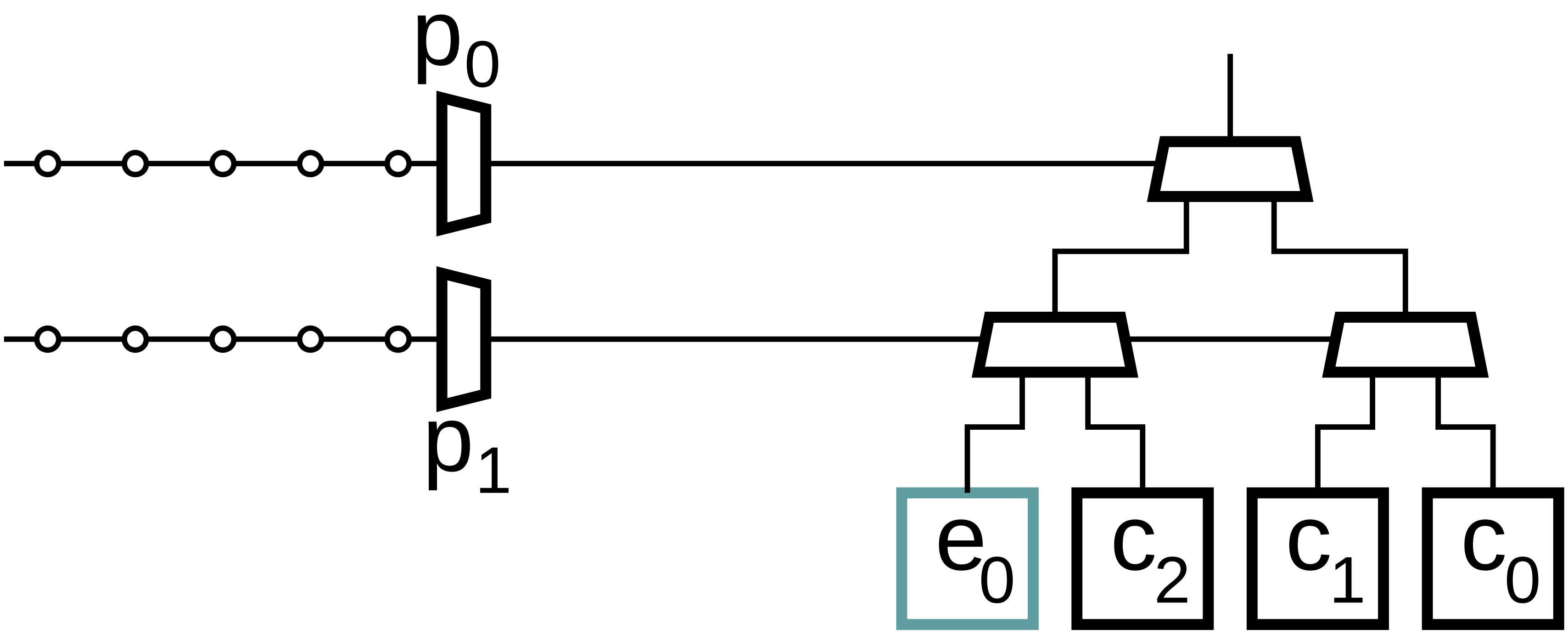
# Example: LUT input ordering



# Example 2-LUT

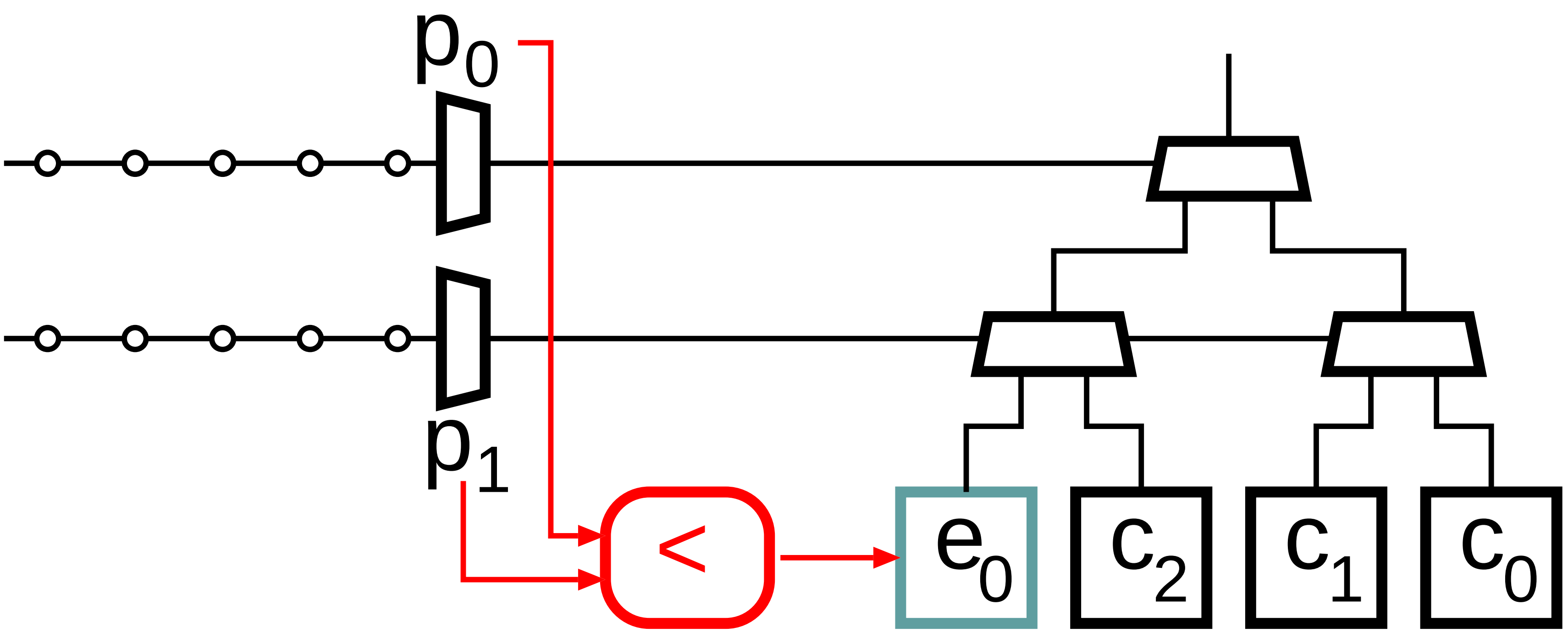


# Example 2-LUT




Bit $e_0$
-- removed from bitstream
-- regenerated at load-time
-- storage bit still exists in LUT

# Example 2-LUT




Regeneration circuit
-- shared by all LUTs
-- regenerates missing bits

$p_0 < p_1$

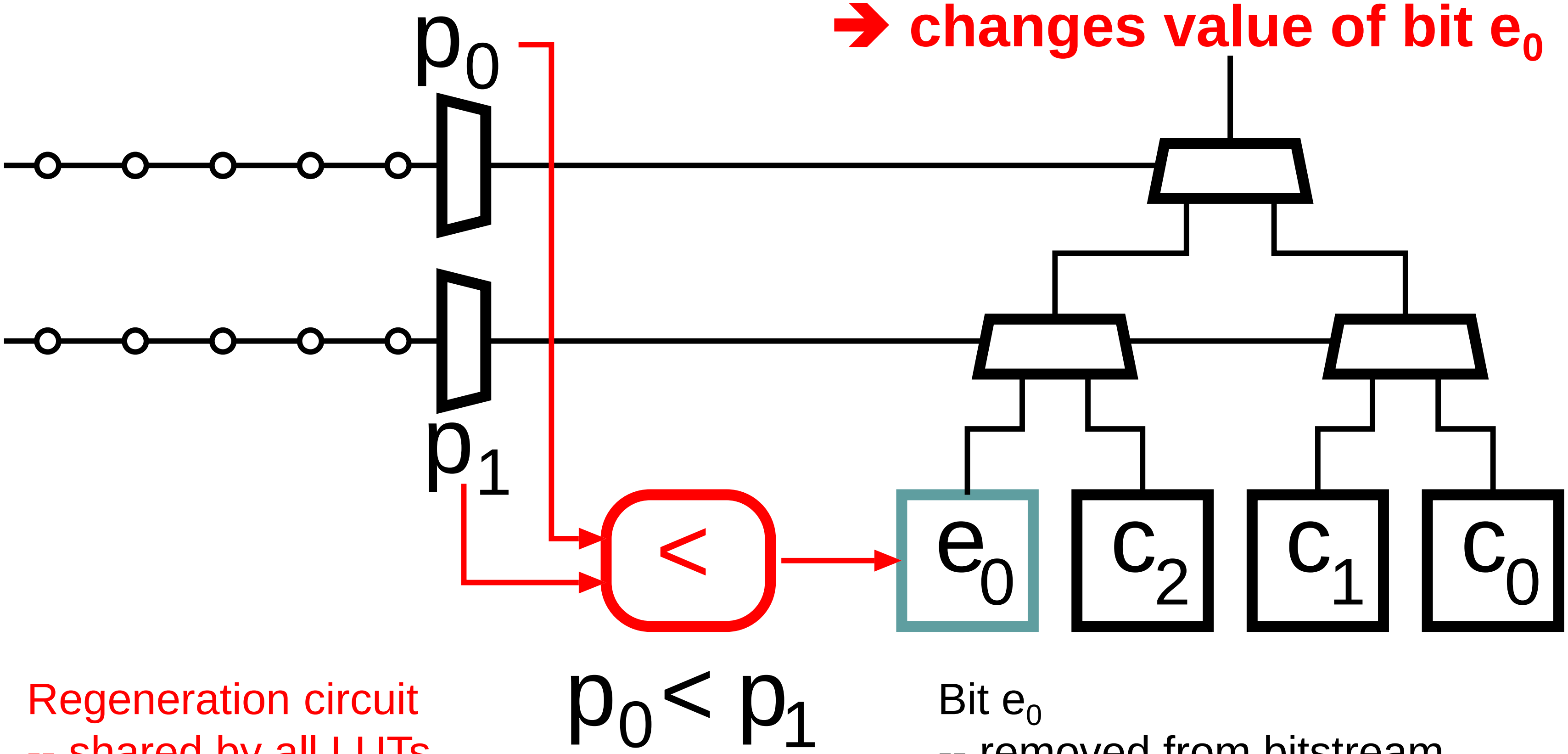
Bit $e_0$
-- removed from bitstream
-- regenerated at load-time
-- storage bit still exists in LUT

# Example 2-LUT

**Problem:**

**changing order of inputs ➔ reorders LUT contents**

**➔ changes value of bit $e_0$**




Regeneration circuit
-- shared by all LUTs
-- regenerates missing bits

Bit $e_0$
-- removed from bitstream
-- regenerated at load-time
-- storage bit still exists in LUT

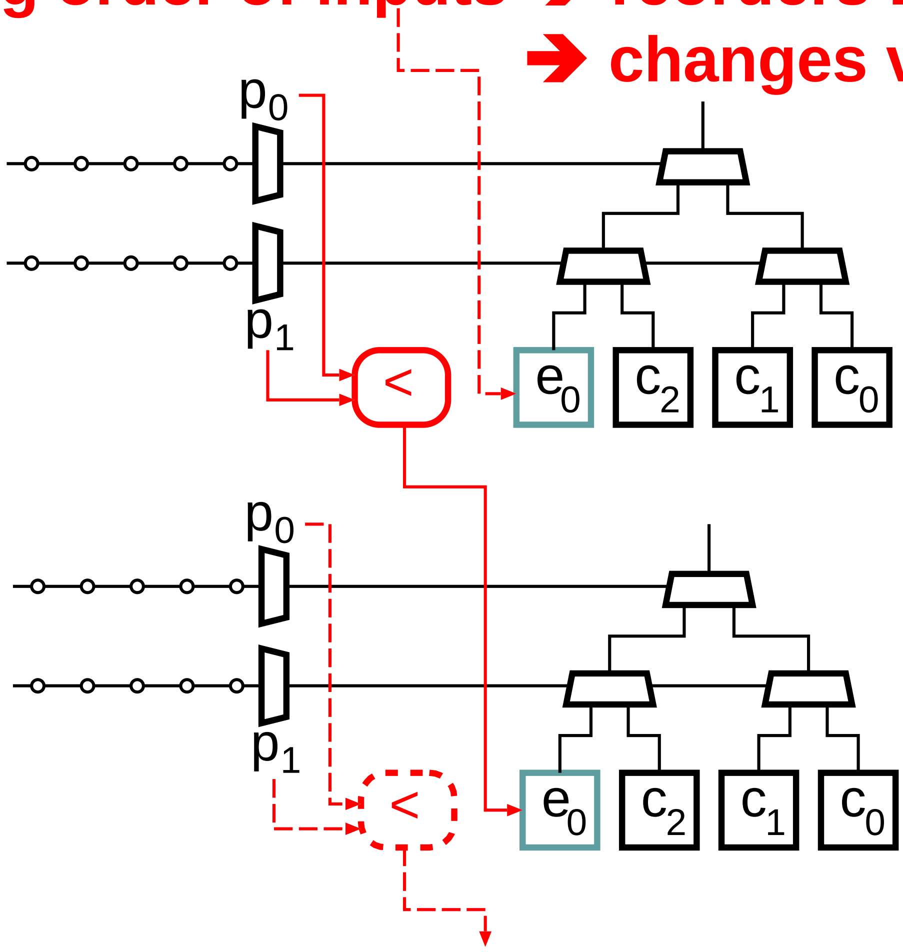
# Configuration Chaining

**Problem:**

**changing order of inputs ➔ reorders LUT contents**

**➔ changes value of bit $e_0$**

**Solution:**



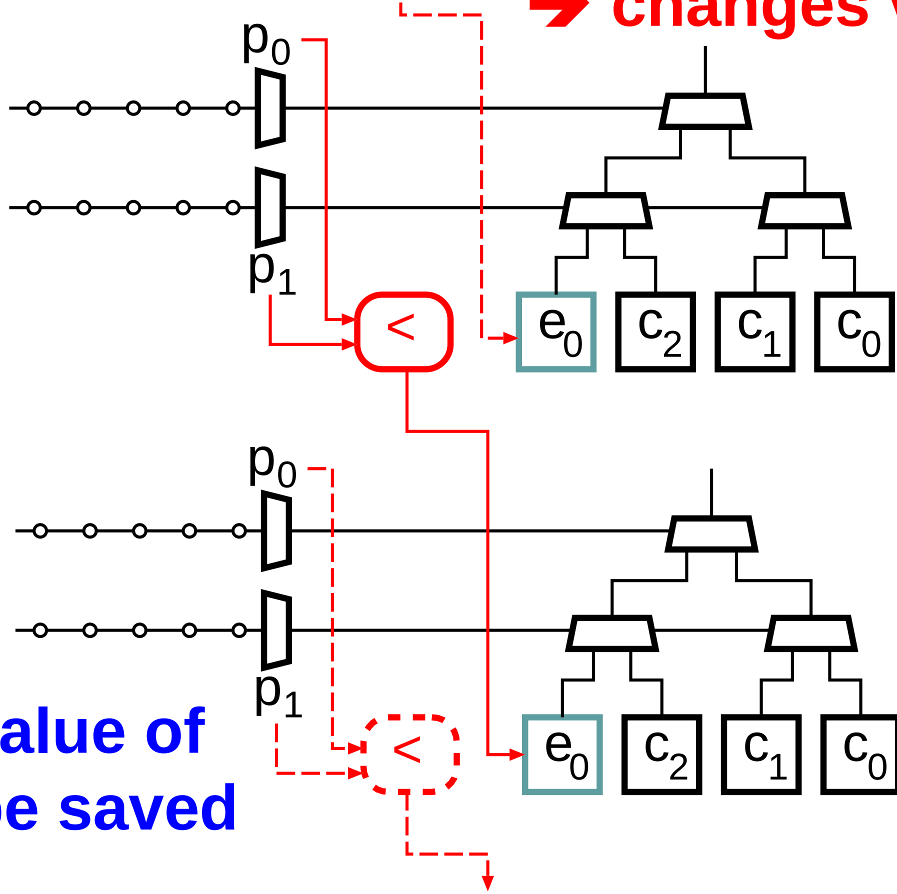
# Configuration Chaining

**Problem:**

**changing order of inputs ➔ reorders LUT contents**

**➔ changes value of bit $e_0$**

**Solution:**




**1. Copy value of bit $e_0$ to be saved**

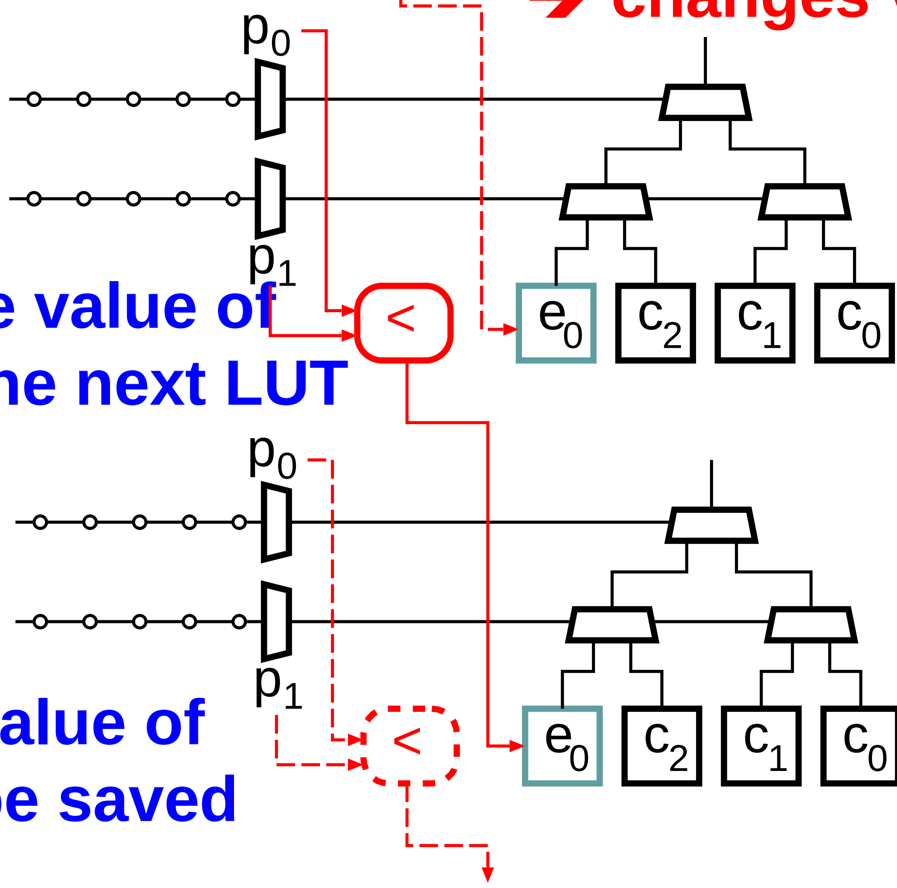
# Configuration Chaining

**Problem:**

**changing order of inputs ➔ reorders LUT contents**

**➔ changes value of bit $e_0$**

**Solution:**




**2. Encode value of bit $e_0$ in the next LUT**

**1. Copy value of bit $e_0$ to be saved**

# General Approach

Produce a LUT chain

- Input ordering of next LUT generates removed bits for current LUT

Approach

1. Enumerate # of input permutations of LUT

   **k=2 → 2! = 2** **k=3 → 3! = 6** **k=4 → 4! = 24**

2. Define number of bits ($p_k$) to be replaced

   **k=2 → $\log_2(2)=1$** **k=3 → $\log_2(6)=2$** **k=4 → $\log_2(24)=4$**

3. For each LUT

   a. Copy P = $p_k$ bits from the current LUT

   b. Re-order next LUT inputs according to value of P

   c. Adjust next LUT configuration bits to match new input ordering

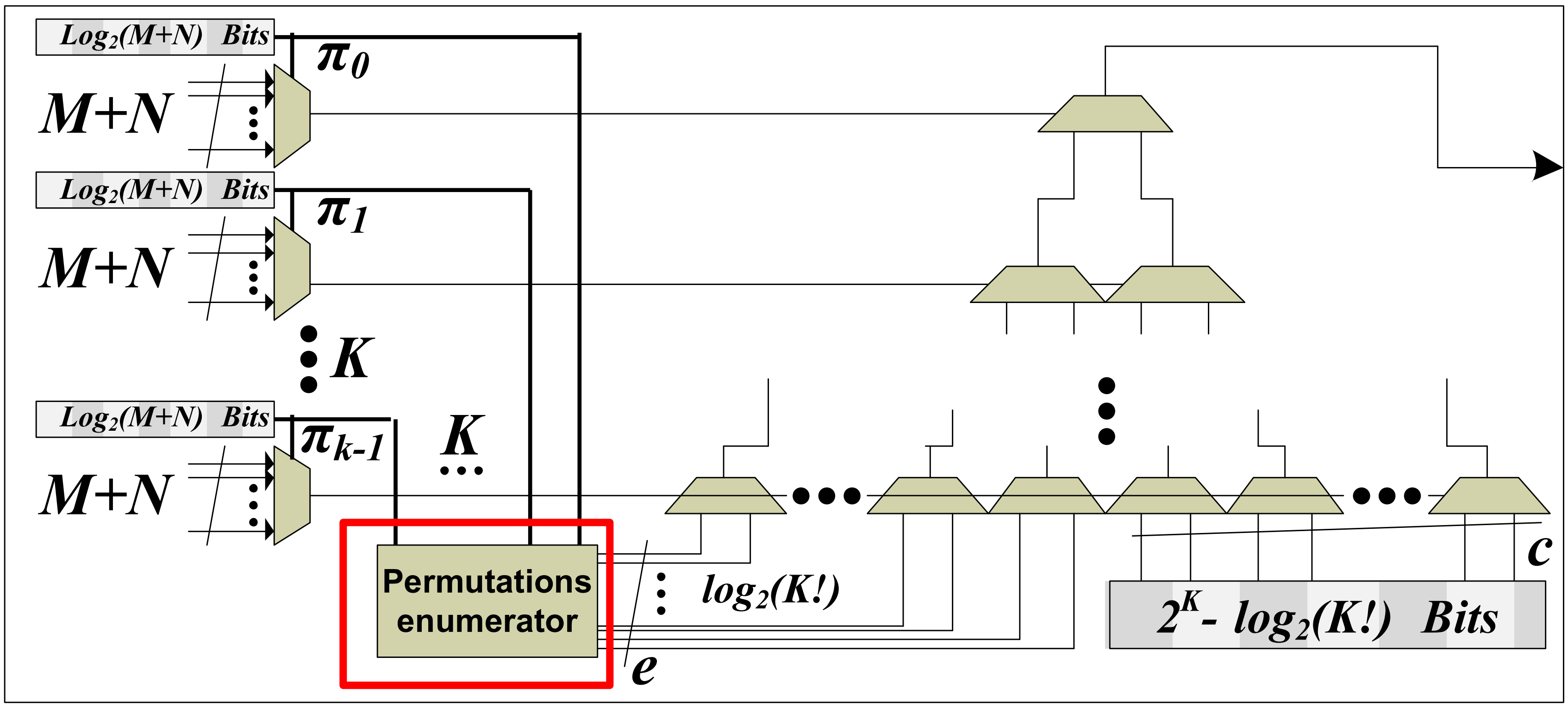
   d. Remove $p_k$ configuration bits from current LUT

# LUT Savings

| k | # LUT bits | # bits Saved = Log2(k!) | % saved |
|---|---|---|---|
| 2 | 4 | 1 | 25% |
| 3 | 8 | 2 | 25% |
| 4 | 16 | 4 | 25% |
| 5 | 32 | 6 | 19% |
| 6 | 64 | 9 | 14% |
| 7 | 128 | 12 | 9.4% |
| 8 | 256 | 15 | 5.9% |

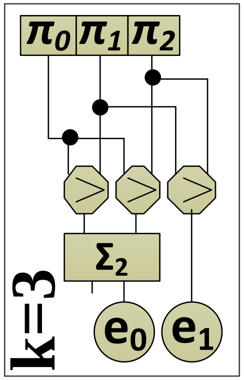
**How complex is bit regeneration?**

# General Structure



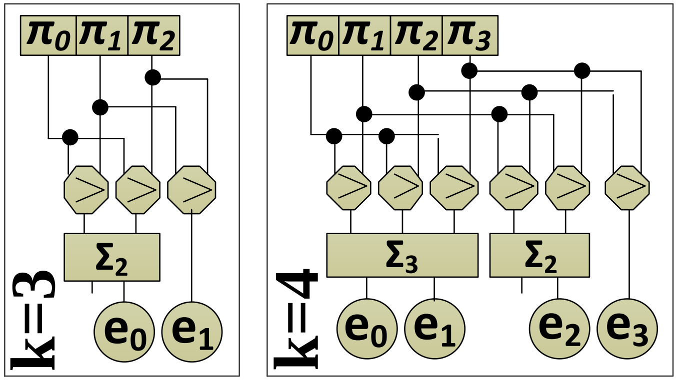
# Regeneration Circuits (k=3,4,5)

- 3 x 6b comparators
- 2b (half) adder
- Saves 2 bits (25%)



# Regeneration Circuits (k=3,4,5)

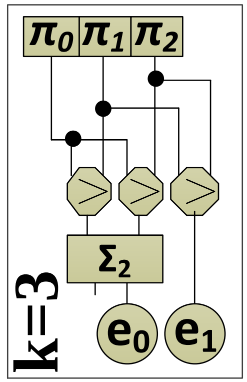
- 3 x 6b comparators
- 2b (half) adder
- Saves 2 bits (25%)

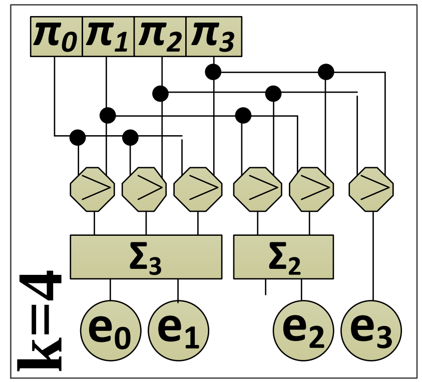



- 6 x 6b comparators
- 3b (full) adder
- 2b (half) adder
- Saves 4 bits (25%)

# Regeneration Circuits (k=3,4,5)

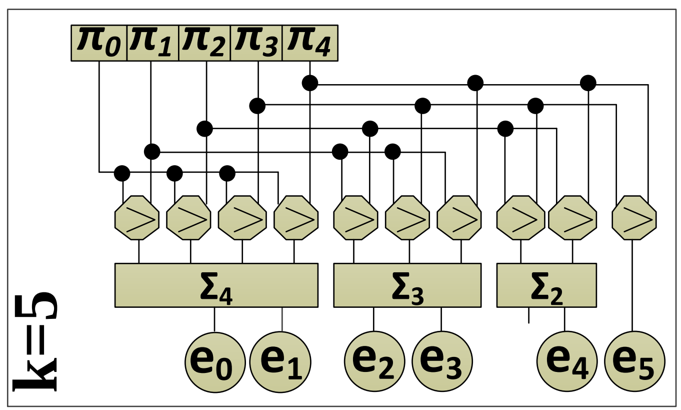
- 3 x 6b comparators
- 2b (half) adder
- Saves 2 bits (25%)



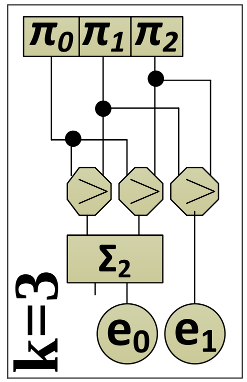

- 6 x 6b comparators
- 3b (full) adder
- 2b (half) adder
- Saves 4 bits (25%)

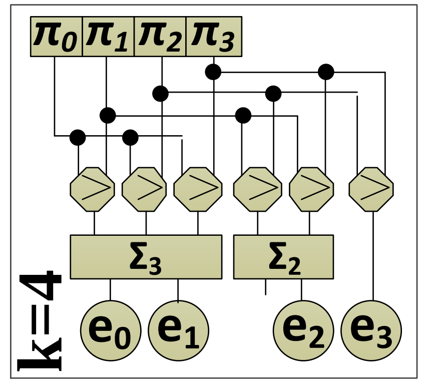
- 10 x 6b comparators
- 2 x 3b (full) adder
- 3 x 2b (half) adder
- Saves 6 bits (25%)



# Regeneration Circuits (k=3,4,5)

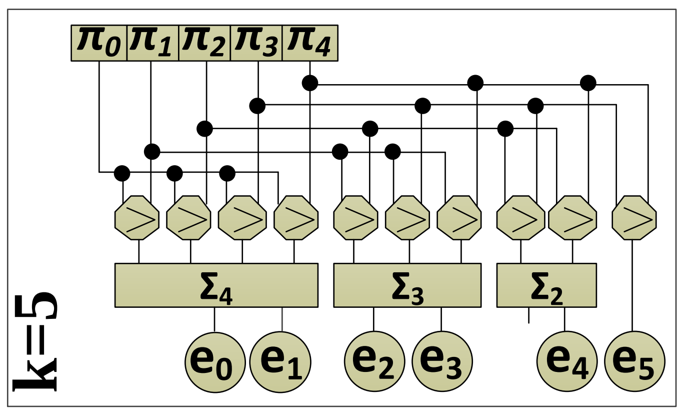
- 3 x 6b comparators
- 2b (half) adder
- Saves 2 bits (25%)




- 6 x 6b comparators
- 3b (full) adder
- 2b (half) adder
- Saves 4 bits (25%)

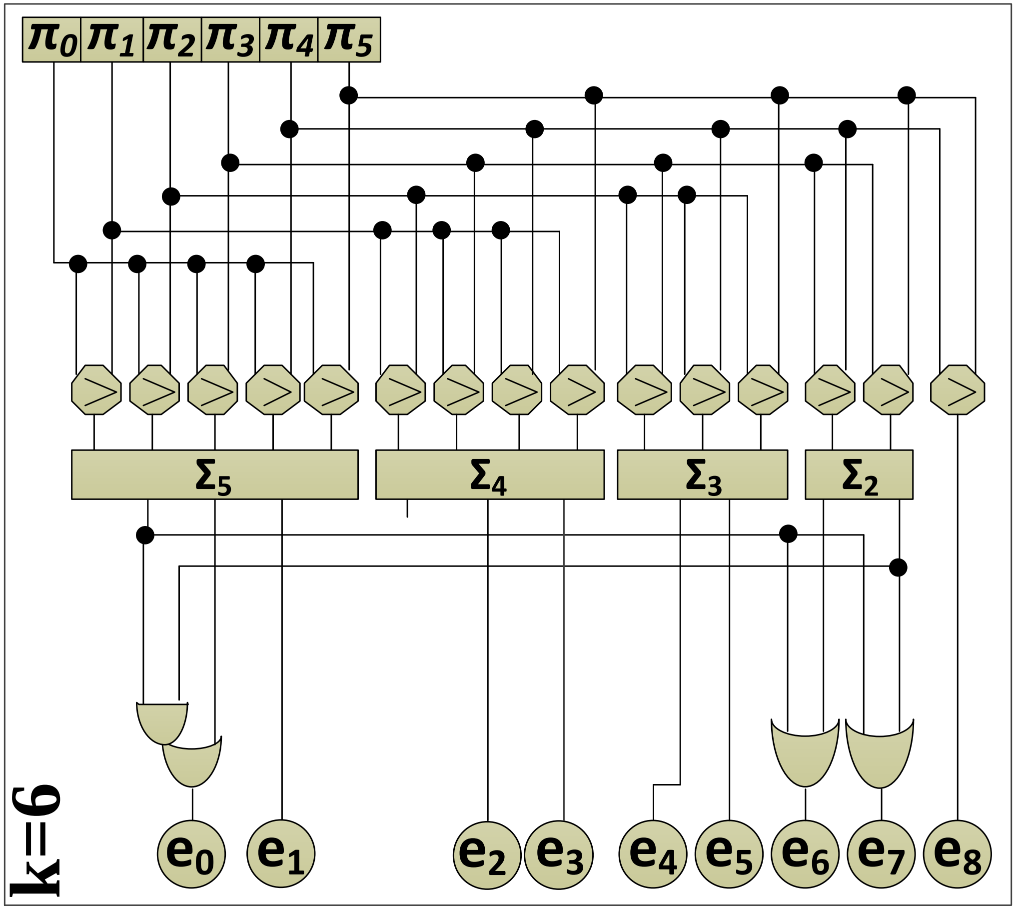
- 10 x 6b comparators
- 2 x 3b (full) adder
- 3 x 2b (half) adder
- Saves 6 bits (25%)



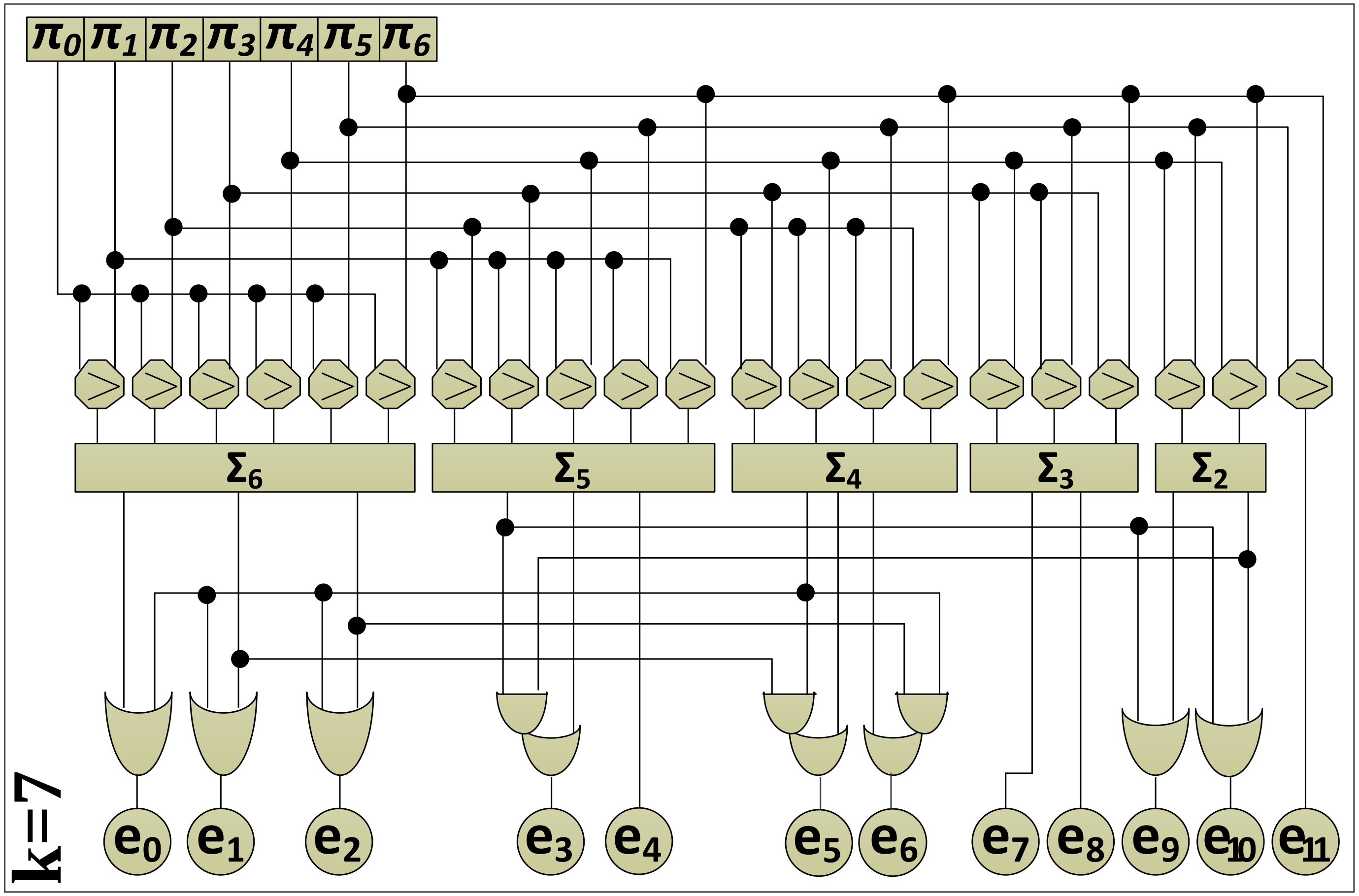

**Note: area cost is only once per device**

(shared across all LUTs in entire FPGA)

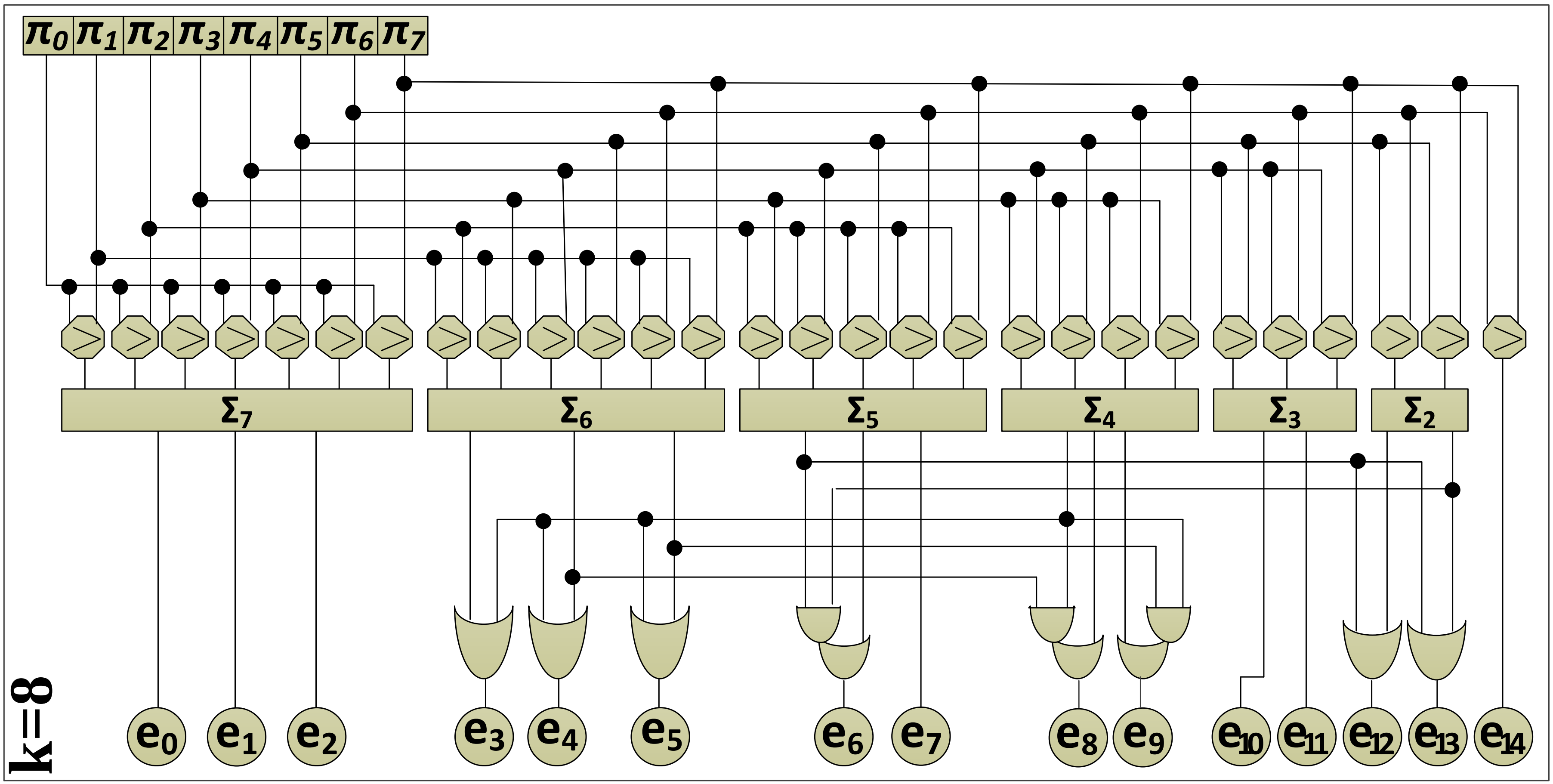
# Regeneration Circuits (k=6)



# Regeneration Circuits (k=7)



# Regeneration Circuits (k=8)



# Regeneration Circuits

All designs are freely available for download.

# Cost of Regeneration – Summary

| k | # LUT bits | # bits Saved = Log2(k!) | | # Compare (6 bit compare) | # Full Adders | # Half Adders | # 2-input Gates |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 1 | | **1** | **0** | **1** | **0** |
| 3 | 8 | 2 | | **3** | **0** | **2** | **0** |
| 4 | 16 | 4 | | **6** | **1** | **1** | **0** |
| 5 | 32 | 6 | | **10** | **2** | **3** | **0** |
| 6 | 64 | 9 | | **15** | **4** | **4** | **4** |
| 7 | 128 | 12 | | **21** | **7** | **5** | **11** |
| 8 | 256 | 15 | | **28** | **11** | **5** | **11** |

**Note: area cost is only once per device**

(shared across all LUTs in entire FPGA)

# Implementation

- 65nm standard cell implementation
  – Use minimum-size drive strength (not timing critical)

| | k=3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|
| Area ($\mu m^2$) | 64 | 129 | 220 | 333 | 502 | 702 |
| Transistors | 198 | 404 | 694 | 1,058 | 1,576 | 2,208 |

- Regeneration circuits are very small
  – Shared across entire device

# Future Work

| k | # LUT bits | # bits Saved = Log2(k!) | % saved |
|---|---|---|---|
| 2 | 4 | 1 | 25% |
| 3 | 8 | 2 | 25% |
| 4 | 16 | 4 | 25% |
| 5 | 32 | 6 | **19%** |
| 6 | 64 | 9 | **14%** |
| 7 | 128 | 12 | **9.4%** |
| 8 | 256 | 14 | **5.5%** |

**Can we reach 25% savings with 5+ inputs ?**

# Future Work

| k | # LUT bits | # bits Saved = Log2(k!) | % saved |
|---|---|---|---|
| 2 | 4 | 1 | 25% |
| 3 | 8 | 2 | 25% |
| 4 | 16 | 4 | 25% |
| 5 | 32 | 6 | **19%** |
| 6 | 64 | 9 | **14%** |
| 7 | 128 | 12 | **9.4%** |
| 8 | 256 | 14 | **5.5%** |

**Can we reach 25% savings with 5+ inputs ?**

**Eg, combine two 4-LUTs to make one 5-LUT**

Thank you!