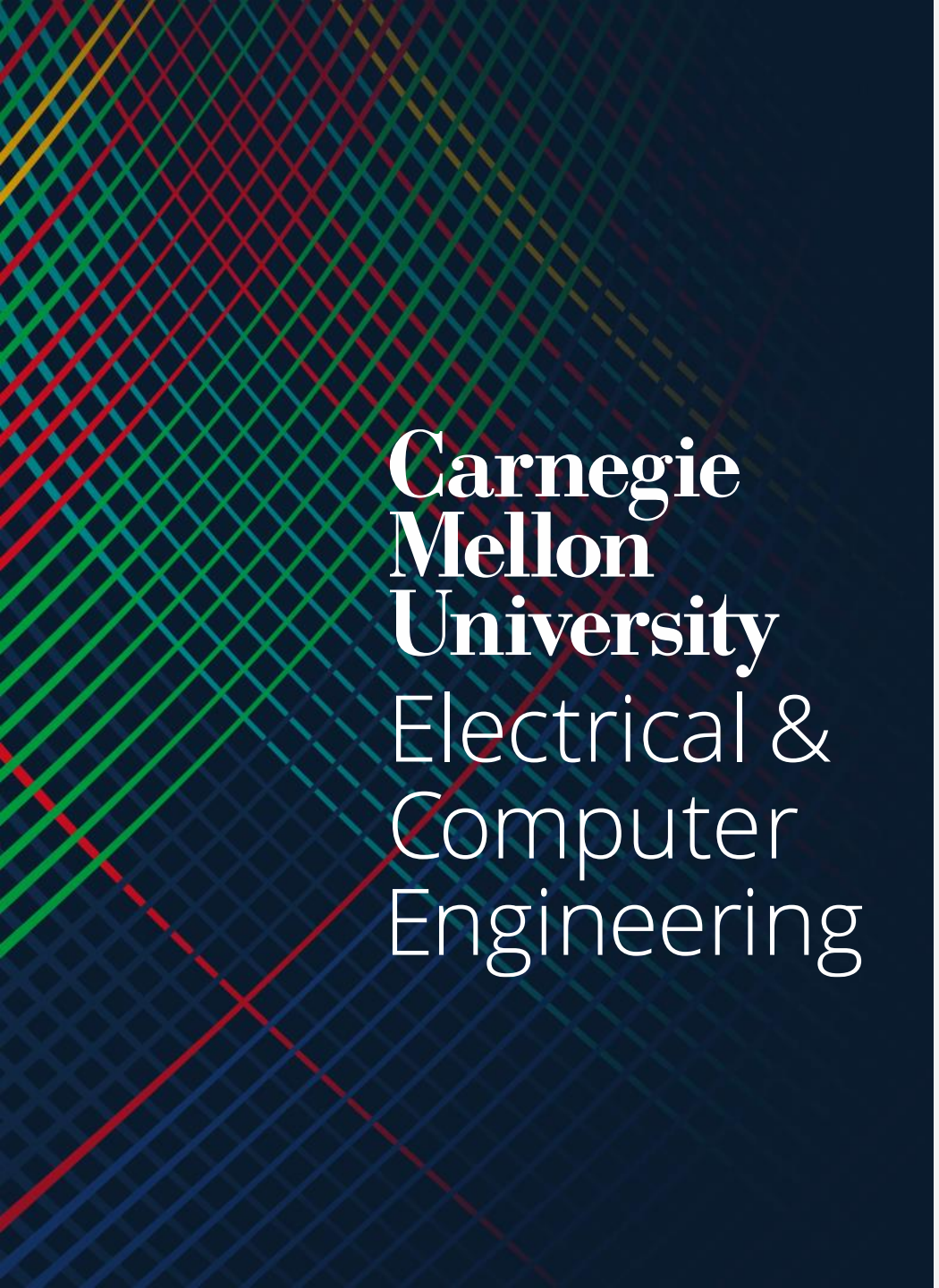


Obfuscation and Security for Digital Integrated Systems

Joe Sweeney, Larry Pileggi

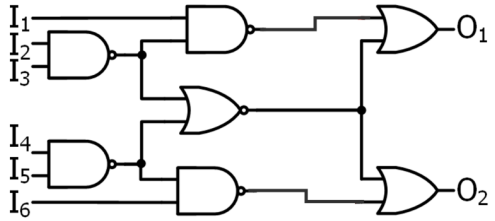
Integrated Circuit Fabrication Flow

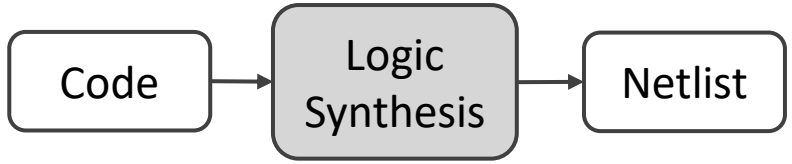
```
module aes_128(clk, start, state, key, out, out_valid);
    input          clk;
    input          start;
    input  [127:0] state, key;
    output [127:0] out;
    output         out_valid;
    reg    [127:0] s0, k0;
    wire   [127:0] s1, s2, s3, s4, s5, s6, s7, s8, s9,
                   k1, k2, k3, k4, k5, k6, k7, k8, k9,
                   k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b, k8b,
k9b;

    reg start_r;
    always @(posedge clk) begin
      start_r <= start;
    end

    wire start_posedge = start & ~start_r;
    reg [4:0] validCounter;

    always @ (posedge clk) begin
      if(start_posedge) begin
        s0 <= state ^ key;
        k0 <= key;
        validCounter <= 21;
      end
      else if(~out_valid) begin
        validCounter <= validCounter - 1;
      end
    end

    assign out_valid = (validCounter == 0);
```

Code

Integrated Circuit Fabrication Flow

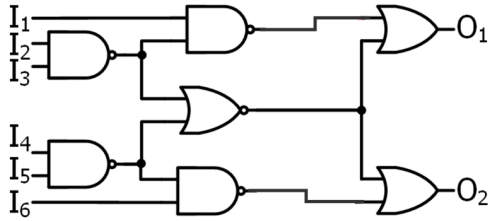
```
module aes_128(clk, start, state, key, out, out_valid);
    input           clk;
    input           start;
    input  [127:0] state, key;
    output [127:0] out;
    output          out_valid;
    reg    [127:0] s0, k0;
    wire   [127:0] s1, s2, s3, s4, s5, s6, s7, s8, s9,
                   k1, k2, k3, k4, k5, k6, k7, k8, k9,
                   k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b, k8b,
k9b;

    reg start_r;
    always @(posedge clk) begin
      start_r <= start;
    end

    wire start_posedge = start & ~start_r;
    reg [4:0] validCounter;

    always @ (posedge clk) begin
      if(start_posedge) begin
        s0 <= state ^ key;
        k0 <= key;
        validCounter <= 21;
      end
      else if(~out_valid) begin
        validCounter <= validCounter - 1;
      end
    end

    assign out_valid = (validCounter == 0);
```

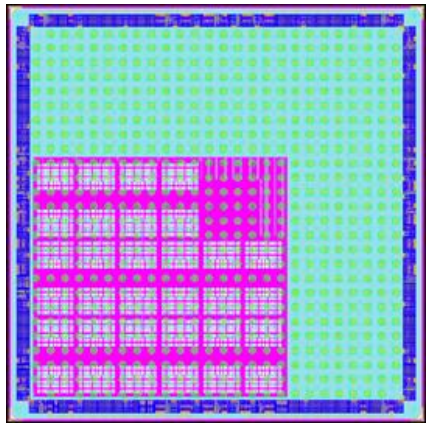




Integrated Circuit Fabrication Flow

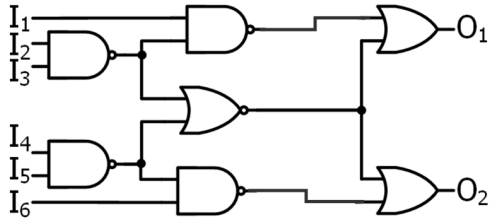
```
module aes_128(clk, start, state, key, out, out_valid);
    input           clk;
    input           start;
    input  [127:0] state, key;
    output [127:0] out;
    output          out_valid;
    reg    [127:0] s0, k0;
    wire   [127:0] s1, s2, s3, s4, s5, s6, s7, s8, s9,
                   k1, k2, k3, k4, k5, k6, k7, k8, k9,
                   k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b, k8b,
k9b;

    reg start_r;
    always @(posedge clk) begin
      start_r <= start;
    end

    wire start_posedge = start & ~start_r;
    reg [4:0] validCounter;

    always @ (posedge clk) begin
      if(start_posedge) begin
        s0 <= state ^ key;
        k0 <= key;
        validCounter <= 21;
      end
      else if(~out_valid) begin
        validCounter <= validCounter - 1;
      end
    end

    assign out_valid = (validCounter == 0);
```

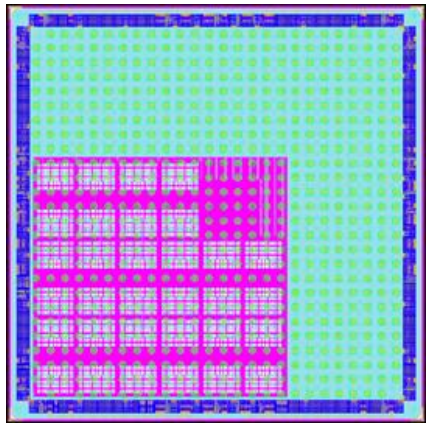


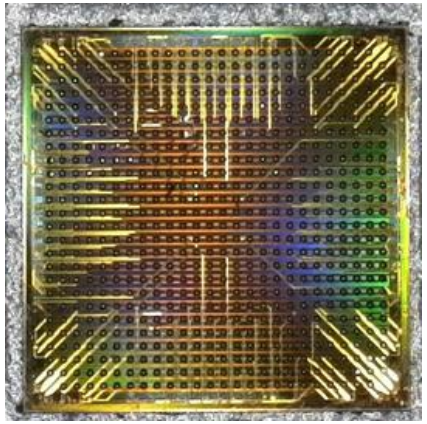


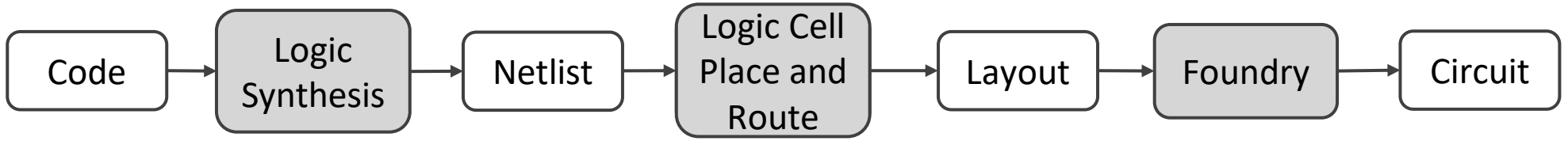
Integrated Circuit Fabrication Flow

```
module aes_128(clk, start, state, key, out, out_valid);
    input           clk;
    input           start;
    input  [127:0] state, key;
    output [127:0] out;
    output          out_valid;
    reg    [127:0] s0, k0;
    wire   [127:0] s1, s2, s3, s4, s5, s6, s7, s8, s9,
                   k1, k2, k3, k4, k5, k6, k7, k8, k9,
                   k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b, k8b,
k9b;

    reg start_r;
    always @(posedge clk) begin
      start_r <= start;
    end

    wire start_posedge = start & ~start_r;
    reg [4:0] validCounter;

    always @ (posedge clk) begin
      if(start_posedge) begin
        s0 <= state ^ key;
        k0 <= key;
        validCounter <= 21;
      end
      else if(~out_valid) begin
        validCounter <= validCounter - 1;
      end
    end

    assign out_valid = (validCounter == 0);
```

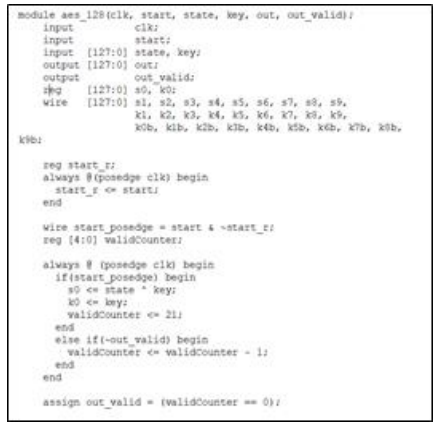


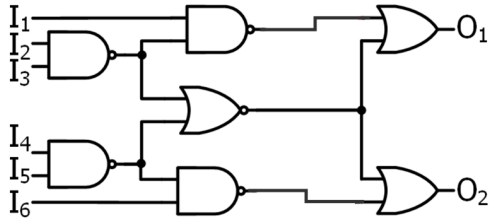


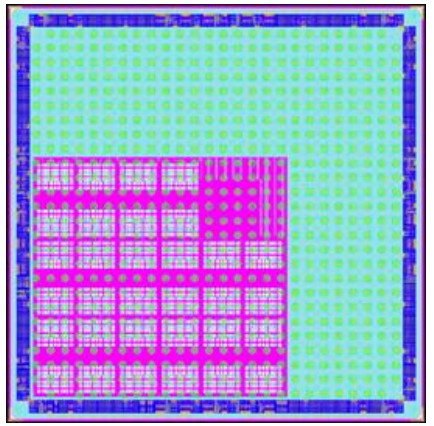


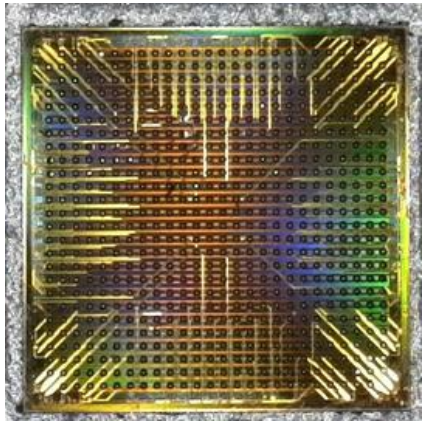


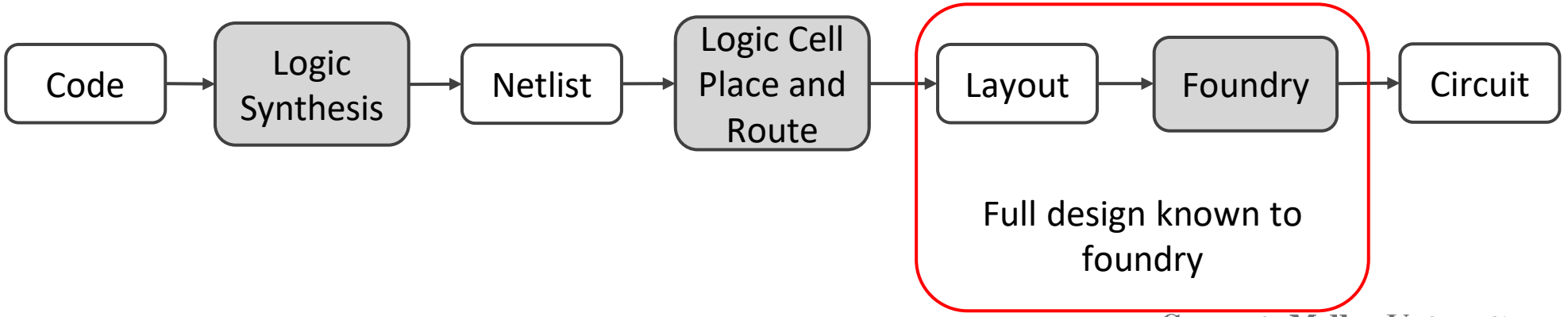
Integrated Circuit Fabrication Flow



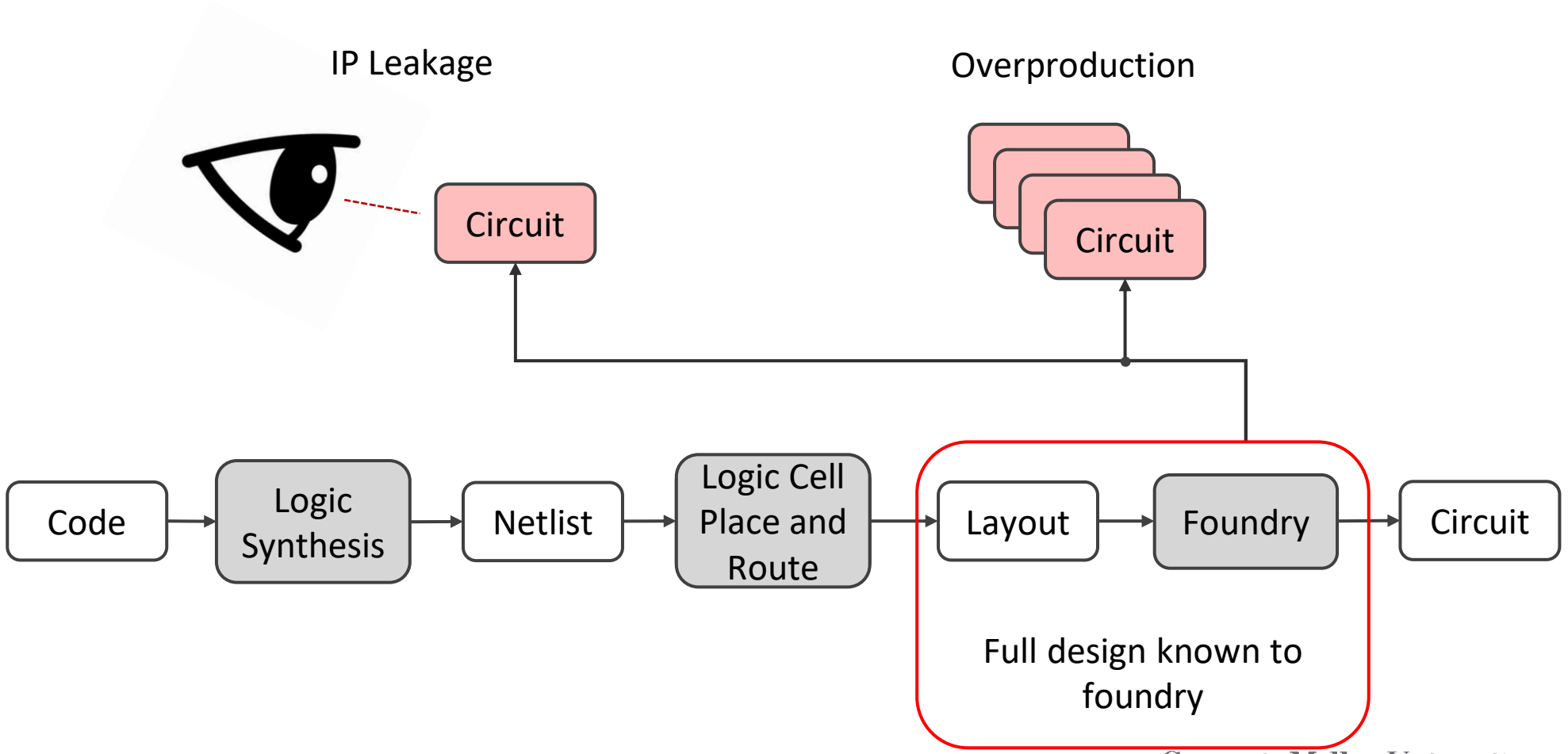




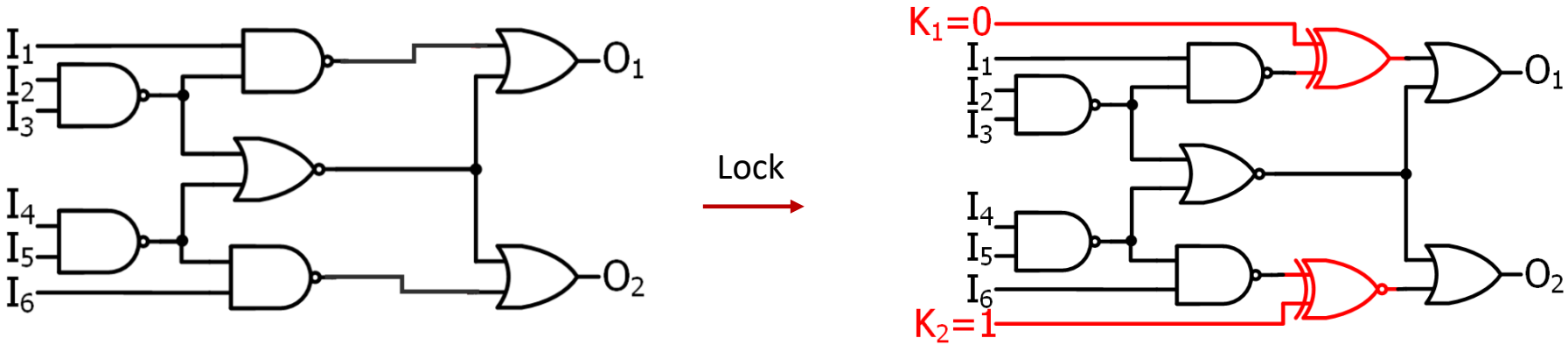




Integrated Circuit Fabrication Flow



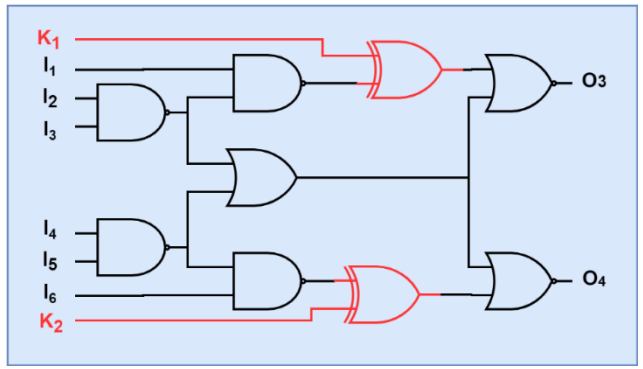
Logic Locking Prevents Unauthorized Use



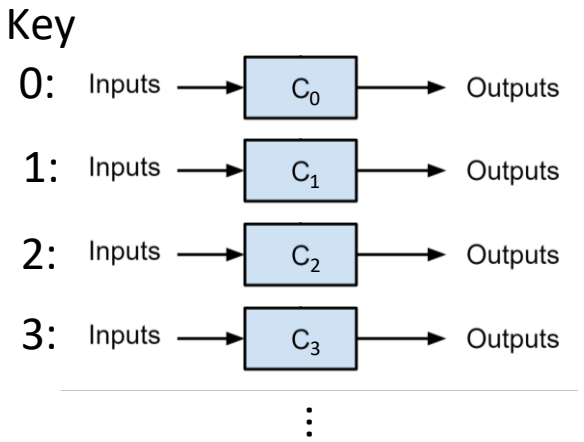
Logic locking adds programmable key elements to circuit

Logic Locking Prevents Unauthorized Use

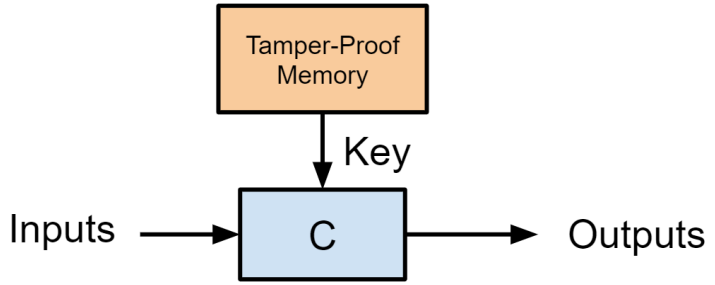
Manufacture locked circuit



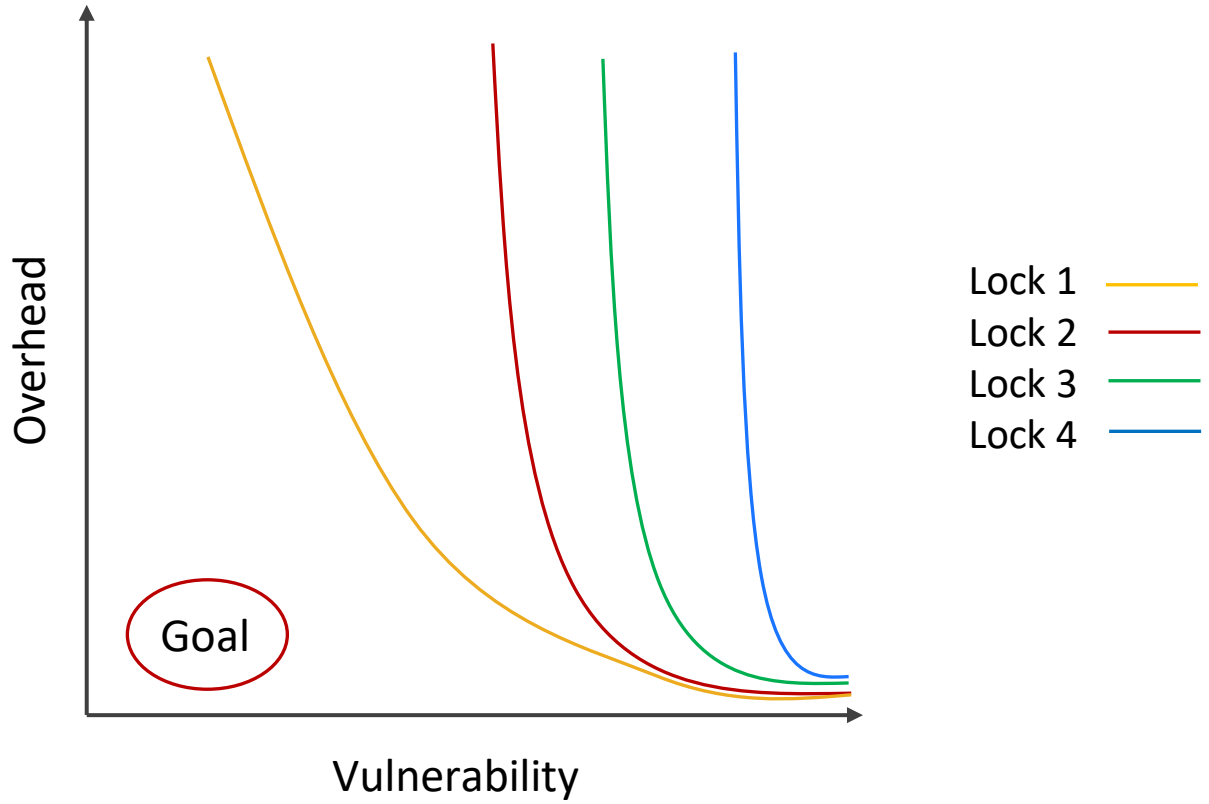
Each key creates different functionality



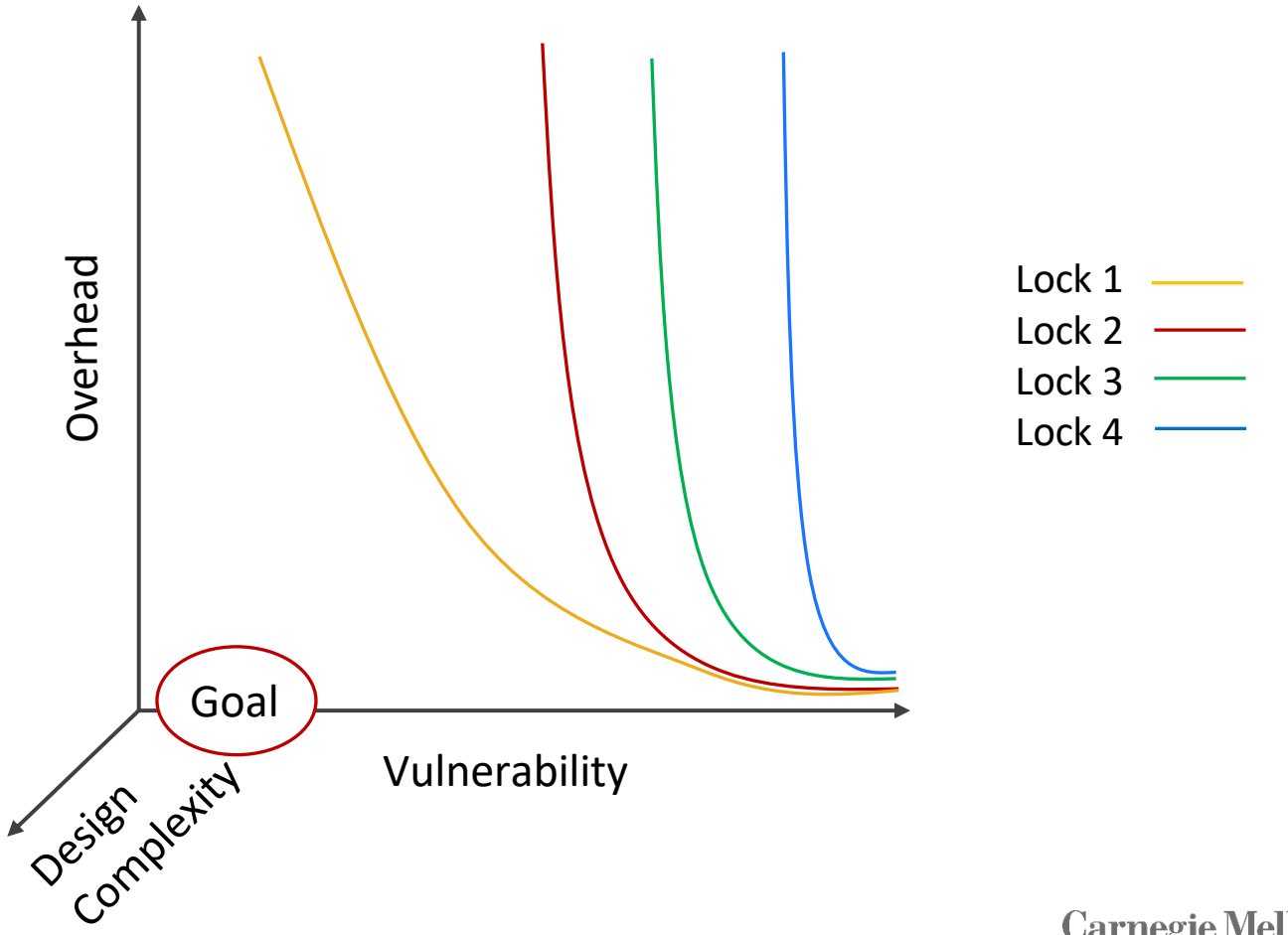
Restore correct functionality with key stored in memory



Tradeoff of Logic Locking

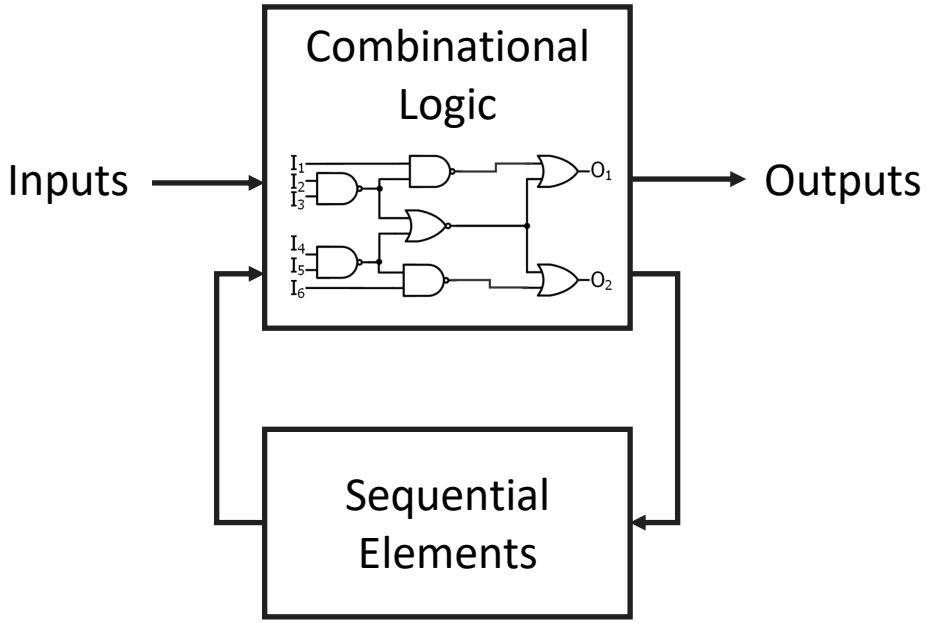


Tradeoff of Logic Locking



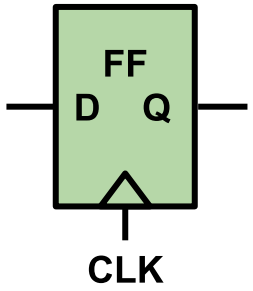
Lock Sequential Elements

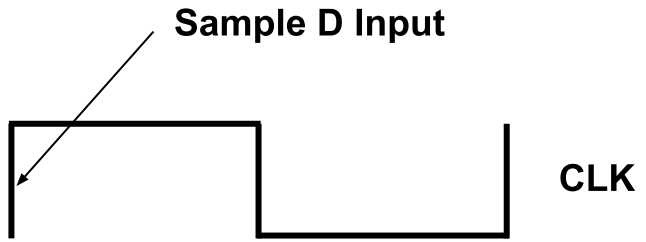
- Circuits made of combinational and sequential elements (memory)
- Current locking schemes target combinational logic
- Our technique manipulates the sequential elements



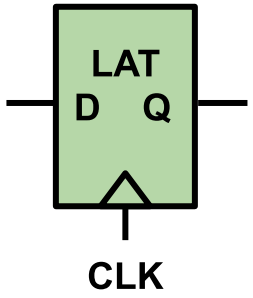
Sequential Element Functionality

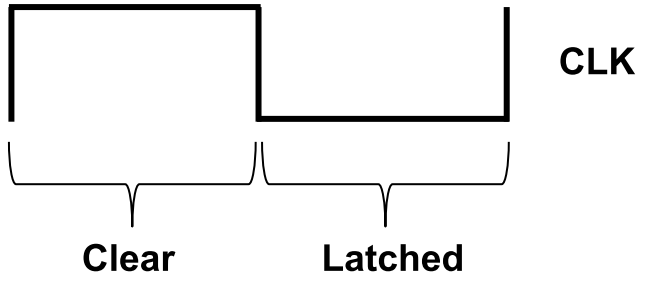
Flip-Flop
(Edge-sensitive)





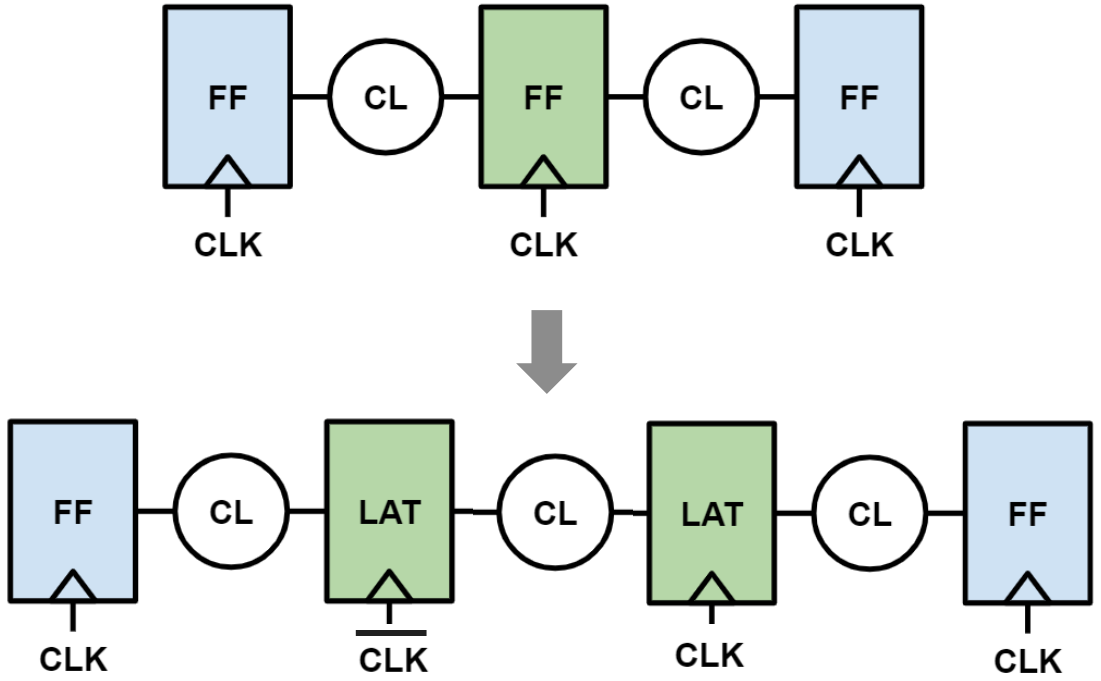
Latch
(Level-sensitive)





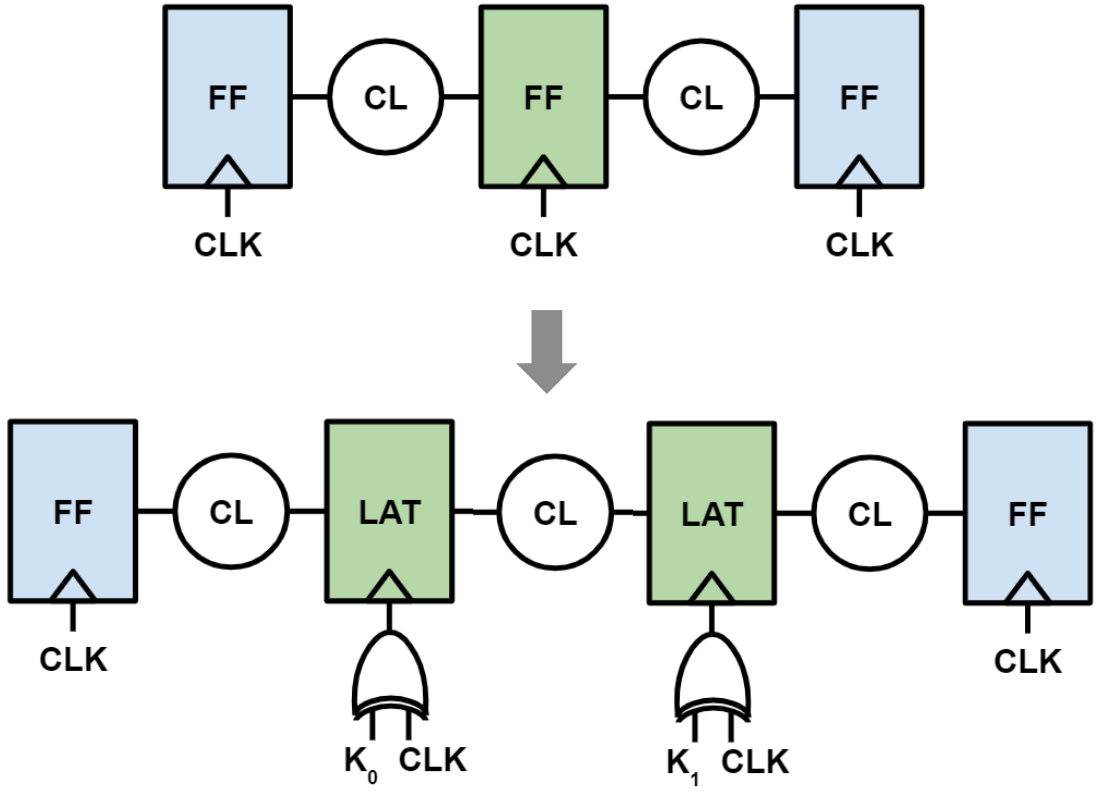
Latch-Based Logic Locking (LBLL)

- Flip-flop can be broken into latches
- Allows flexible signal arrival times with cycle stealing
- Latch-based logic commonly used in high-speed design

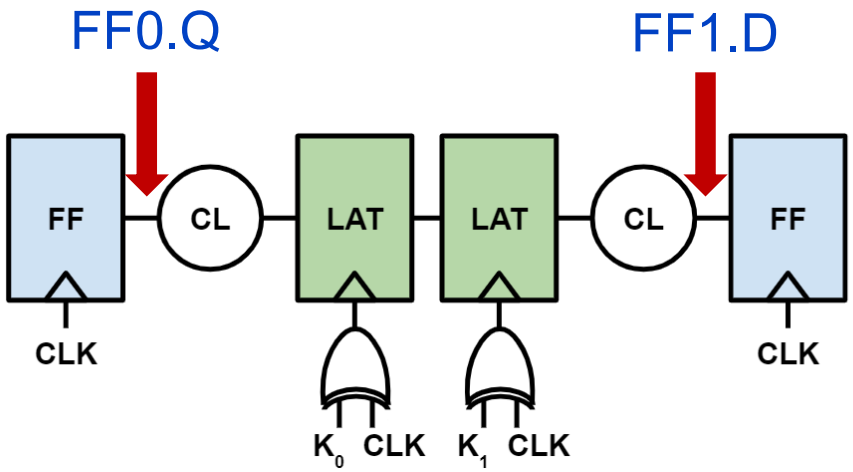


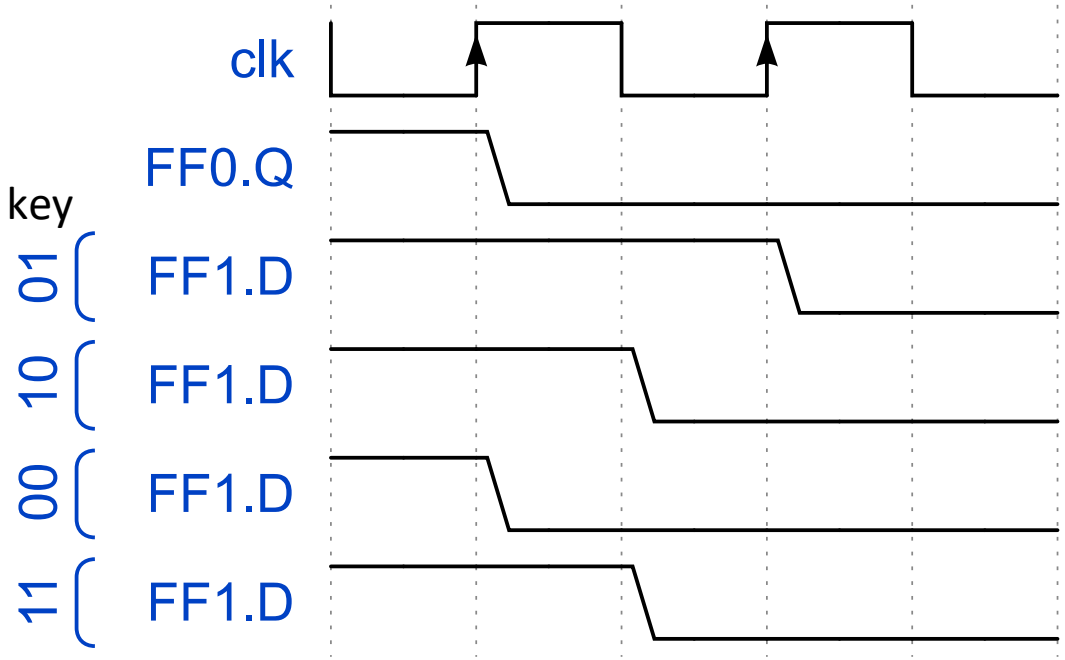
Latch-Based Logic Locking (LBLL)

- LBLL repurposes this design style
- LBLL keys the latch clock phase for obfuscation
- Circuit manipulation not in critical path



LBLL Programmable Path Delay

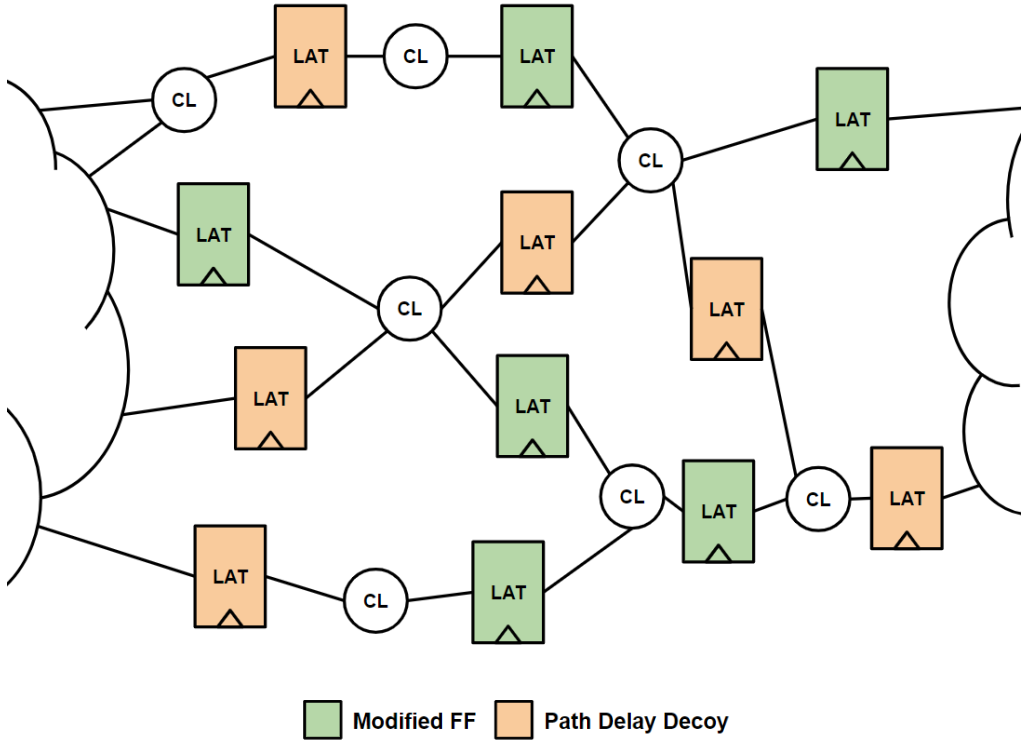




Key inputs modulate signal propagation

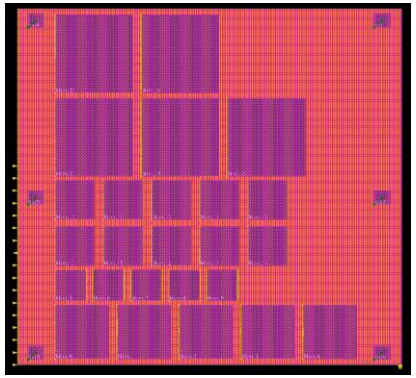
LBLL Conceptual Diagram

- Portion of original flops converted to latches
- Delay decoys are held clear with correct key
- Resistant to all known attacks

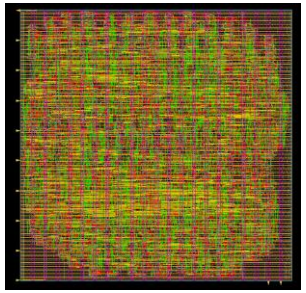


Performance, Power, Area Overhead

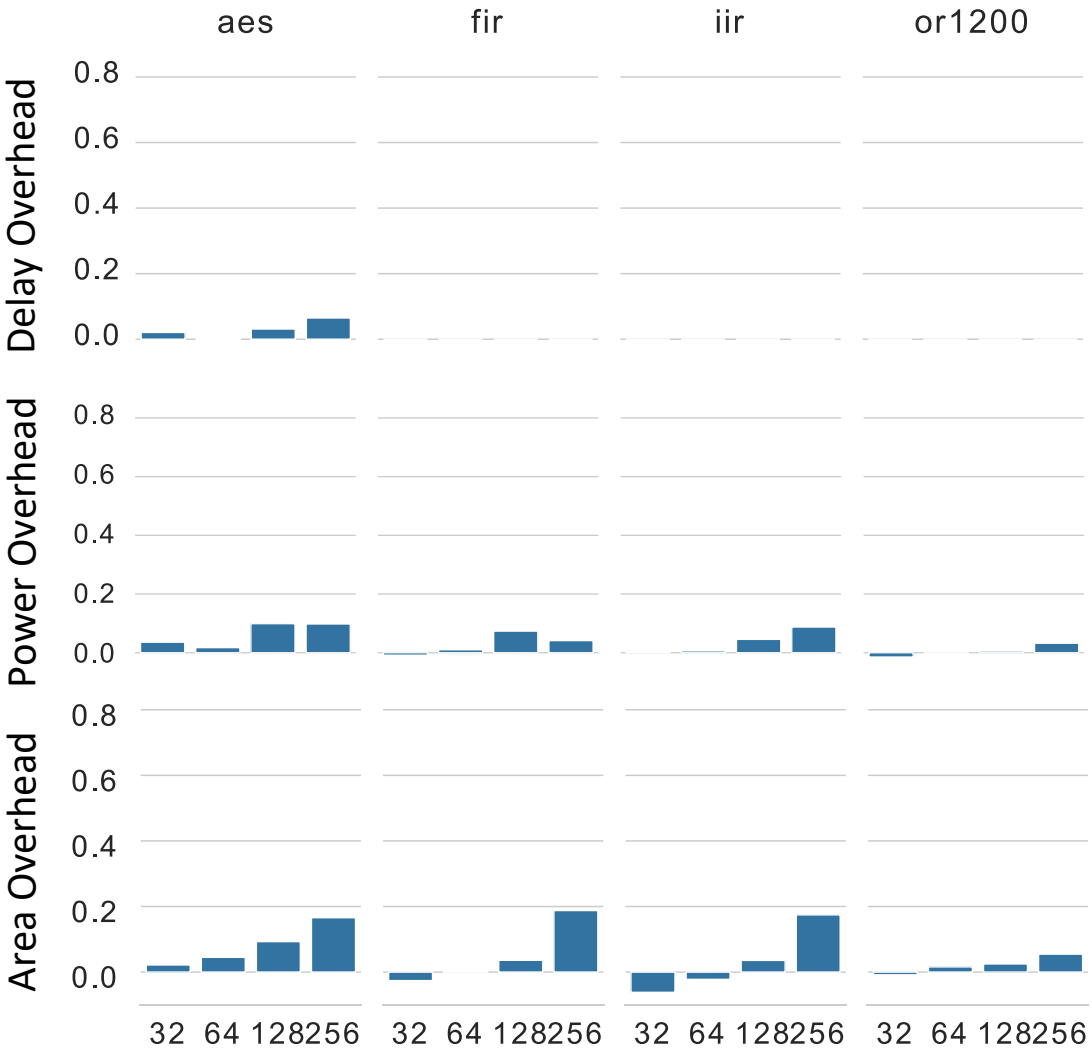
- Manufactured circuits in modern technology (22nm)
- Negligible delay overhead
- Small power and area increase



Top Level

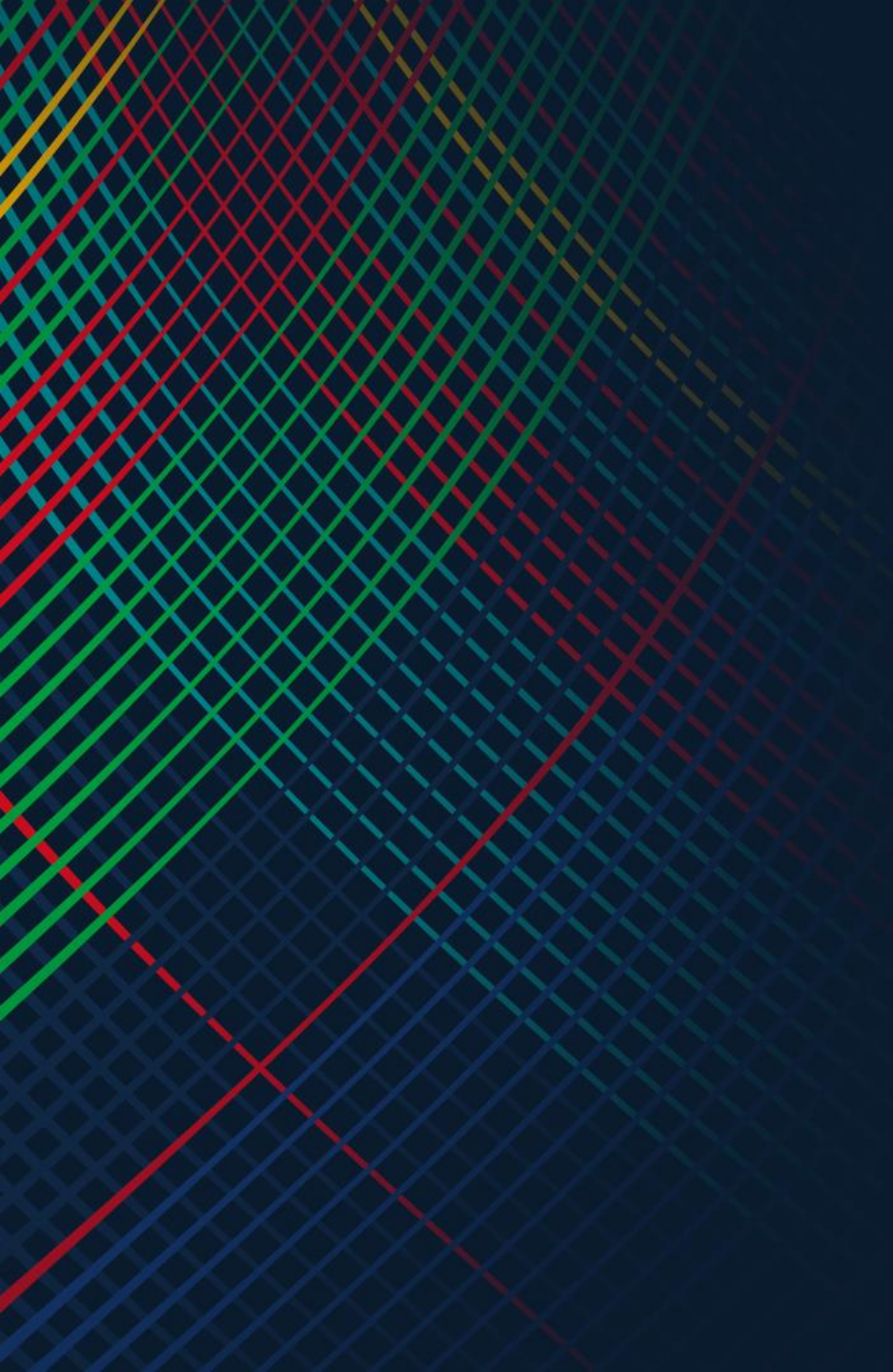


IIR, 256b



Summary

- Logic locking protects a design during manufacture
- Techniques must balance overhead with vulnerability
- LBLL provides a low overhead locking solution



Thanks!