ECE 4514
Digital Design II
Spring 2008

Lecture 18:
Optimizing Area

A *Tools/Methods* Lecture

Patrick Schaumont

Optimization and Backend Verification

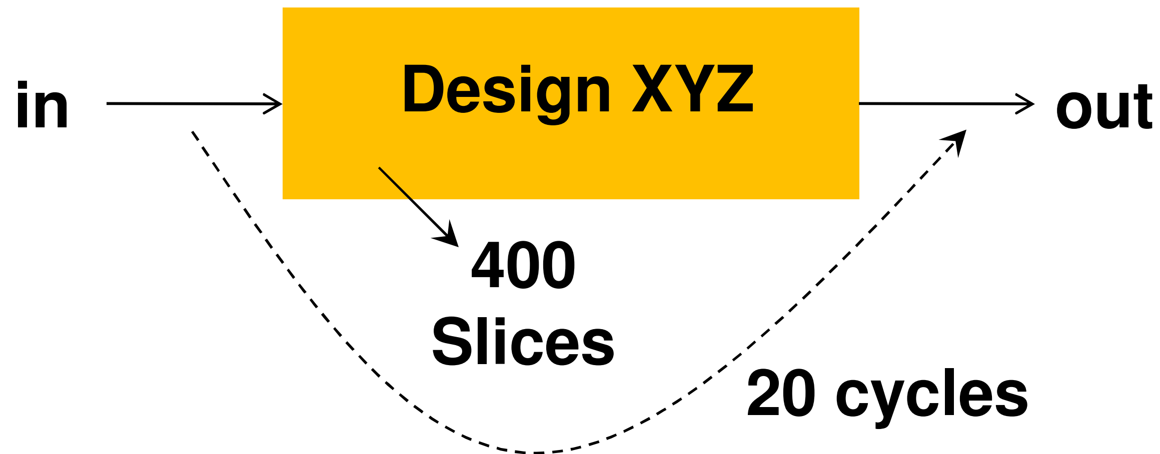
- Previous 4 lectures (Lecture 13-16):
  - How to get Verilog mapped into hardware
- Next 2 lectures:
  - How to *optimize* Verilog for hardware implementation
  - Today (Lecture 17): Optimize for (smaller) area
  - Thursday (Lecture 18): Optimize for (higher) performance

Why do we need optimization?

- A given algorithm can be implemented in many different ways in digital hardware
- Each implementation is characterized by
  - A certain area (In FPGA: Slices)
  - A certain cycle budget (the amount of cycles needed to complete one iteration through the algorithm)
- The application domain and other external requirements provide *constraints* for the area or the performance (=cycle_budget^{-1})
  - 'Create a circuit not larger than can fit in a Spartan3S100 FPGA ...' (area)
  - 'Create an implementation that can complete at least 500 million additions per second ..' (performance)

Area-Delay Product: Generic Optimality Criterion

- Area-Delay Product = Area of a circuit times the cycle budget for that circuit



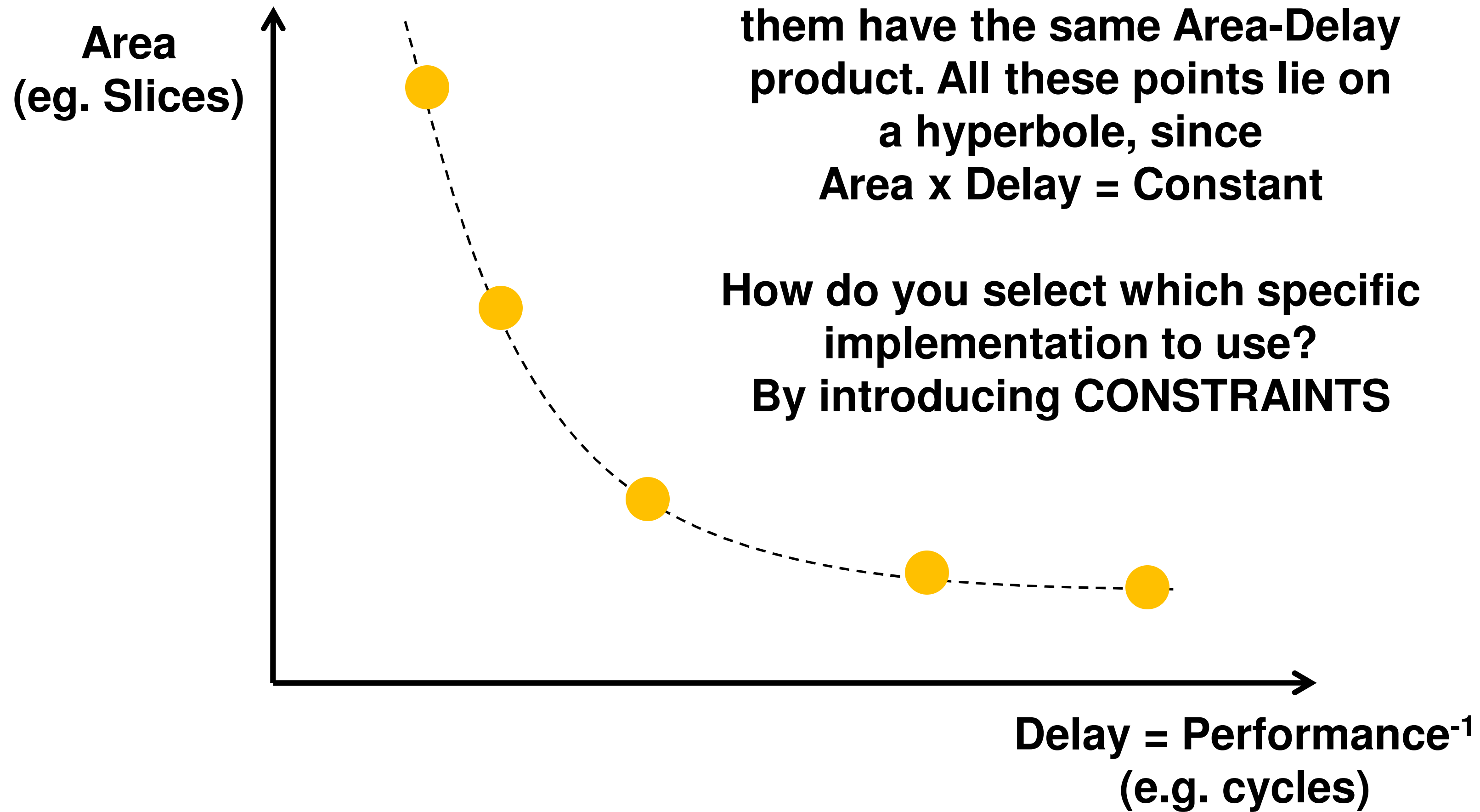
Area-Delay = 8000 Slices.cycles

A *lower* Area-Delay product is better

Area-Delay Product

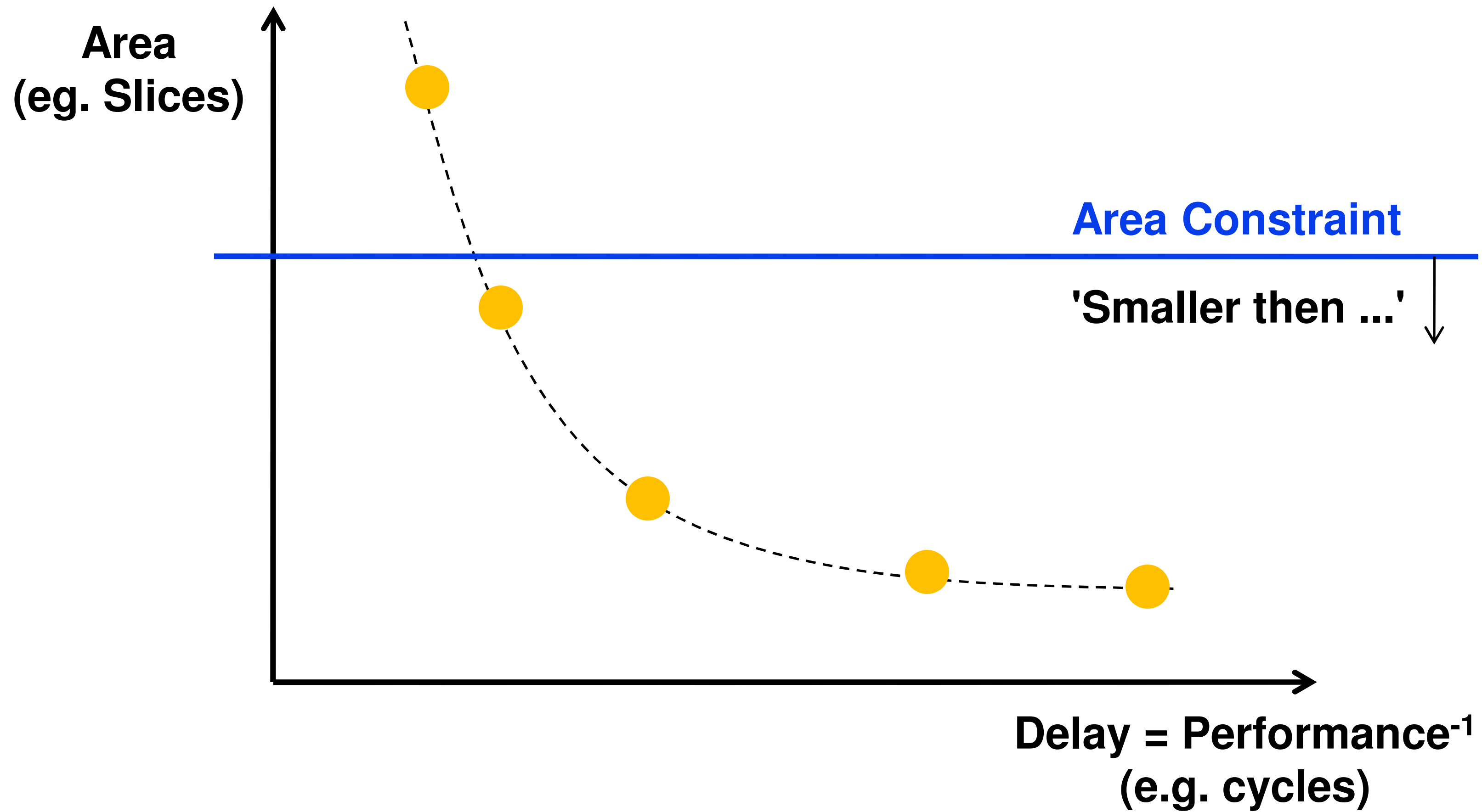
Assume that a given design can be implemented in 5 different ways, and that all of them have the same Area-Delay product. All these points lie on a hyperbole, since
Area x Delay = Constant

How do you select which specific implementation to use?
By introducing CONSTRAINTS



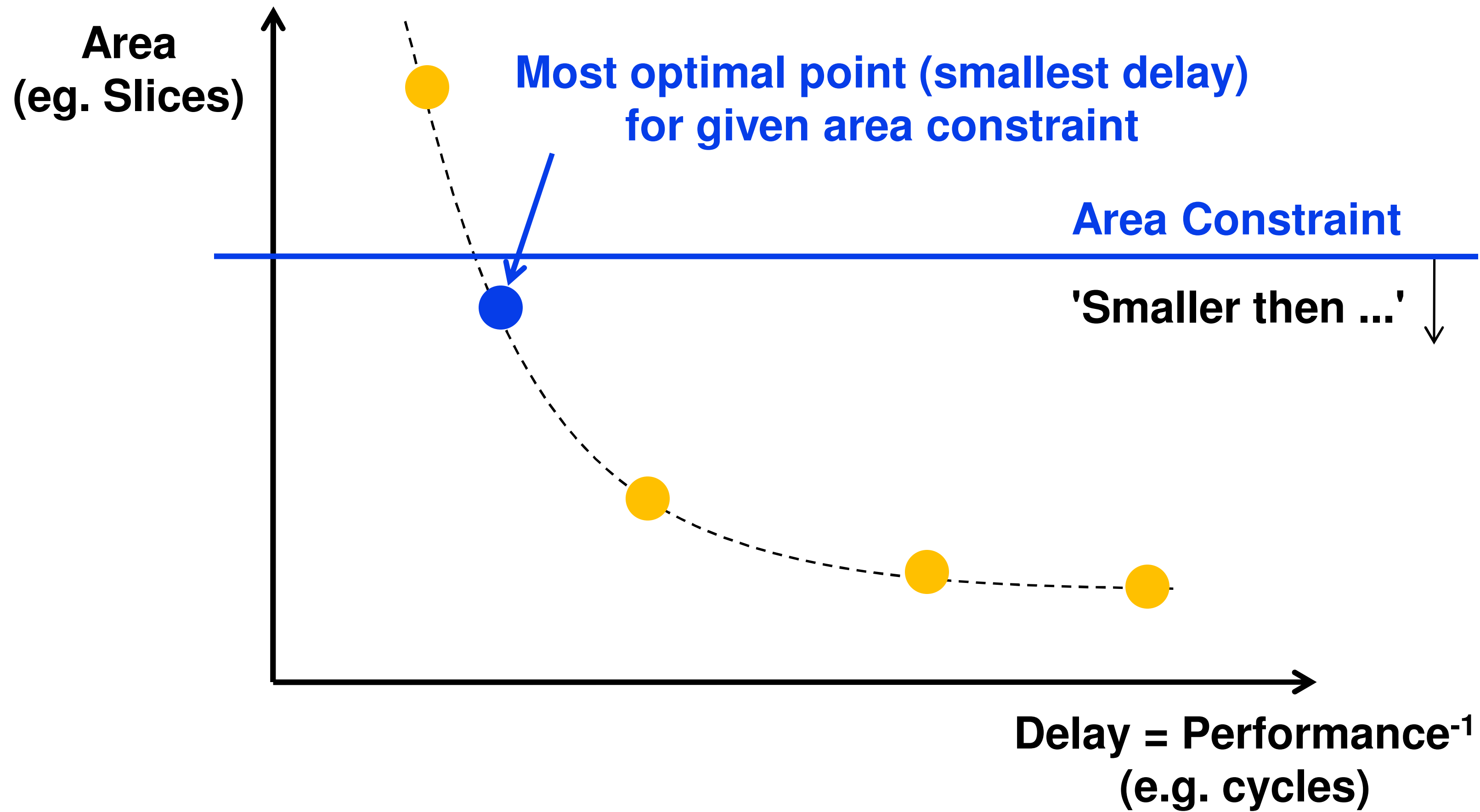
Area-Delay Product

- An Area constraint



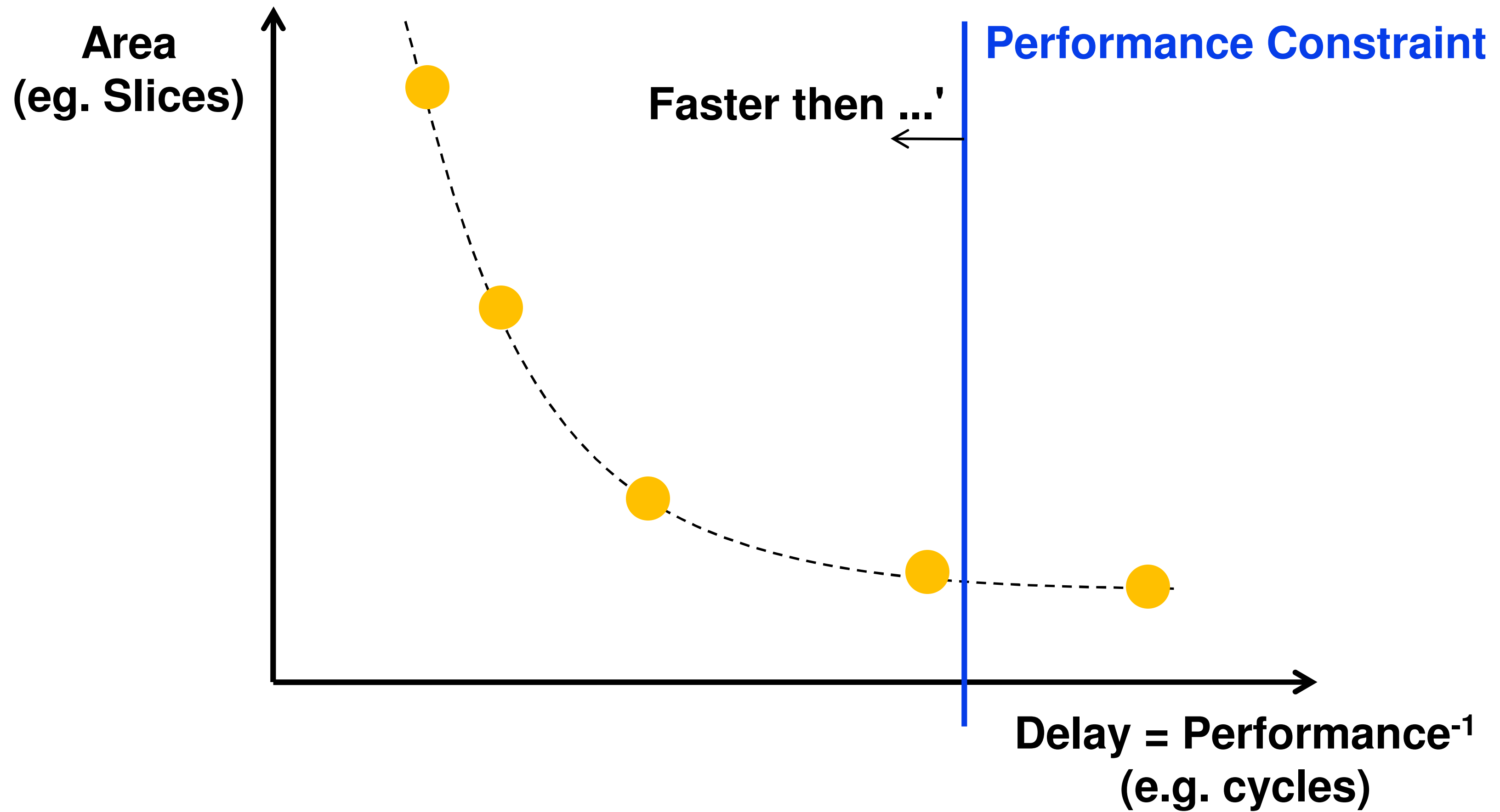
Area-Delay Product

- An Area constraint



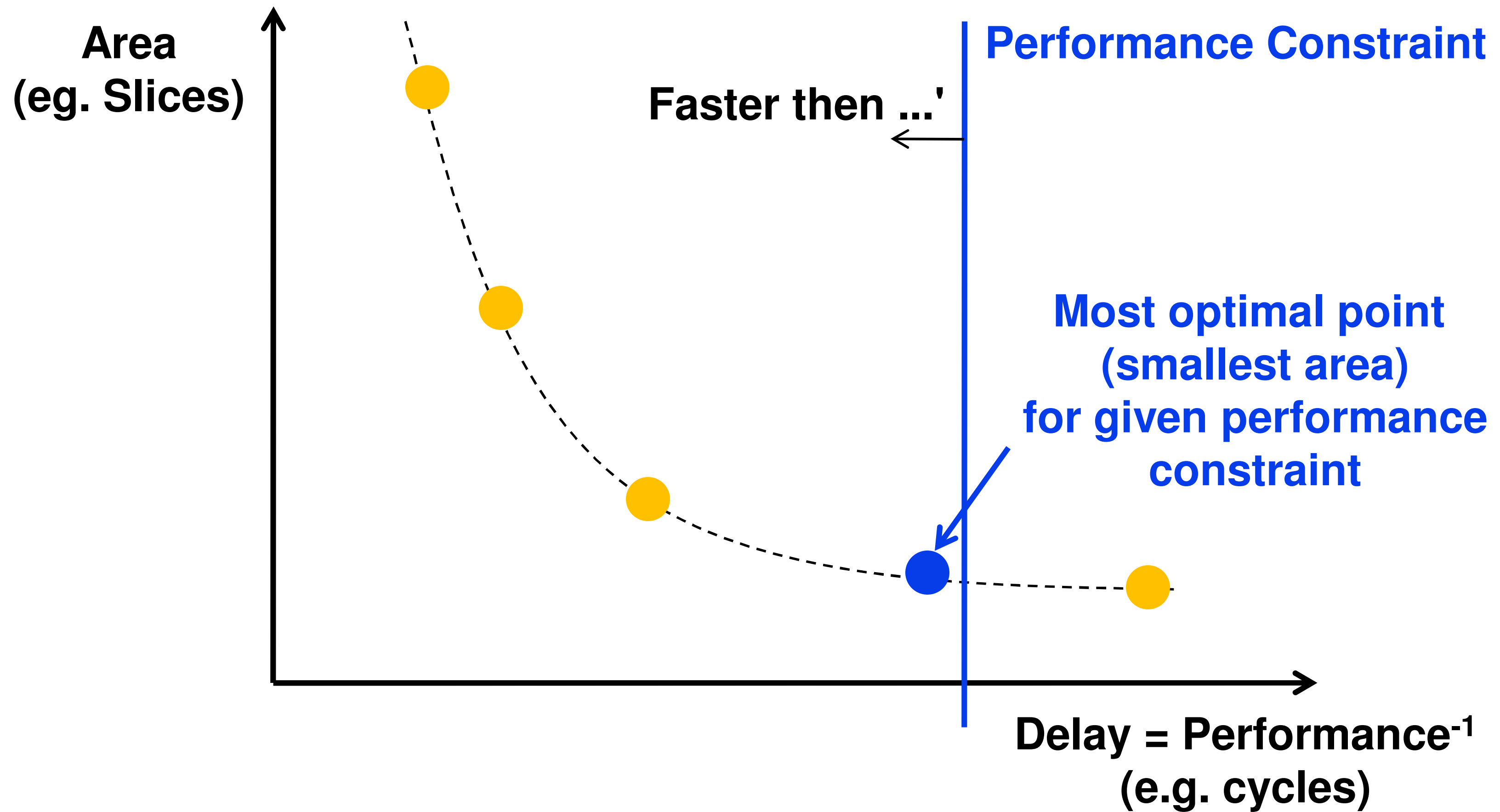
Area-Delay Product

- A Performance Constraint



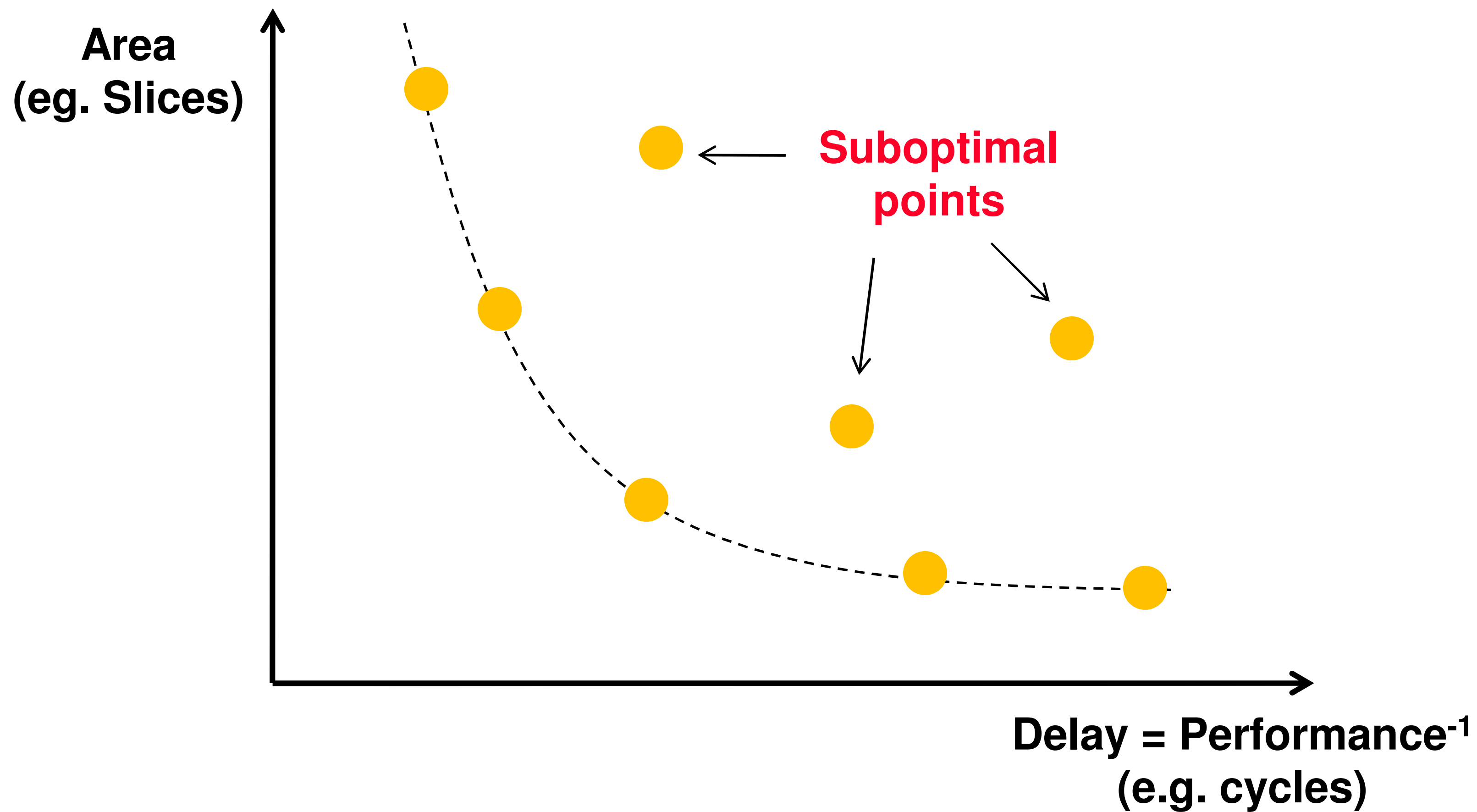
Area-Delay Product

- A Performance Constraint



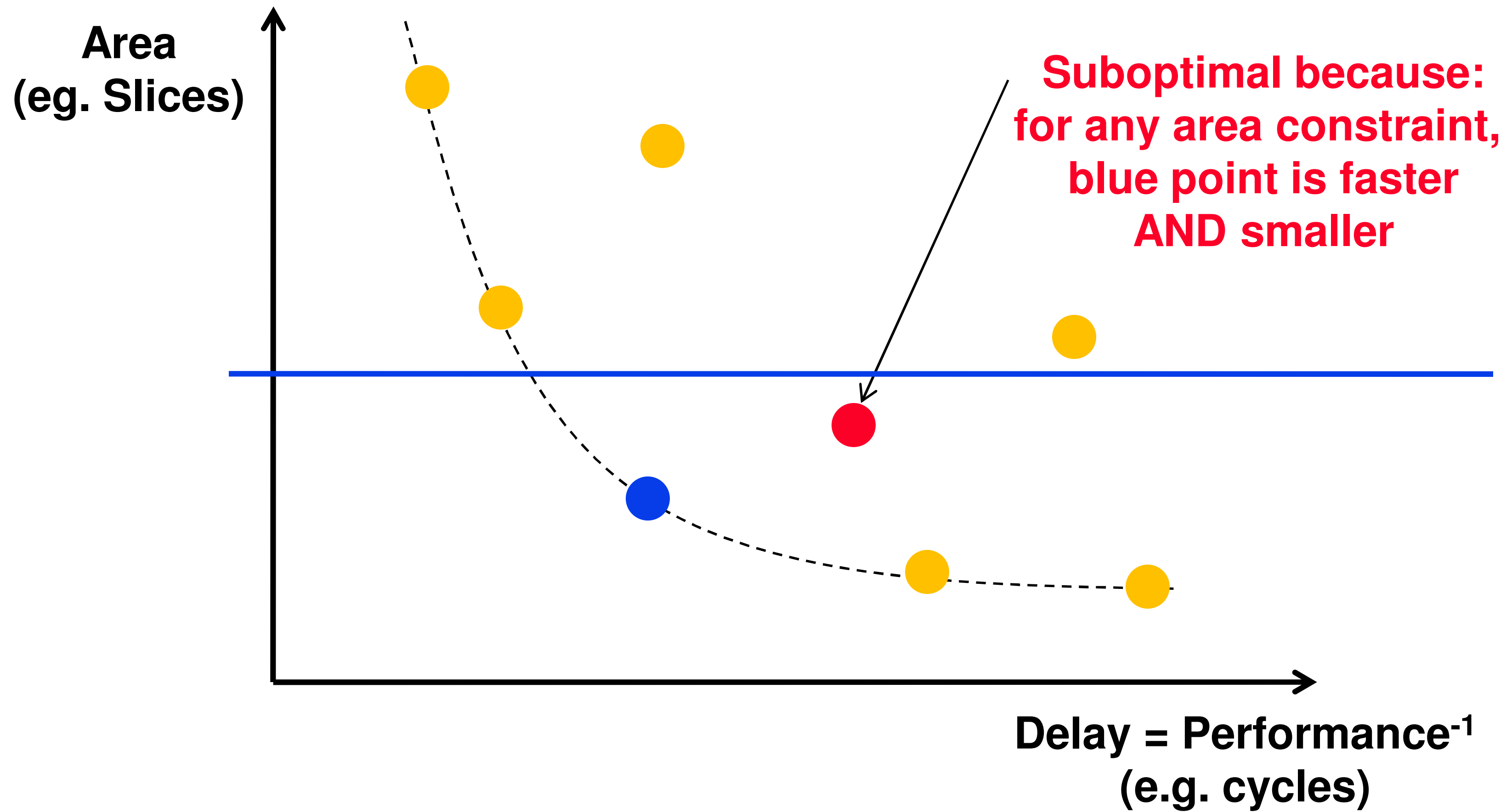
Area-Delay Product

- If no constraint is given, all points with the same area-delay product are equally optimal



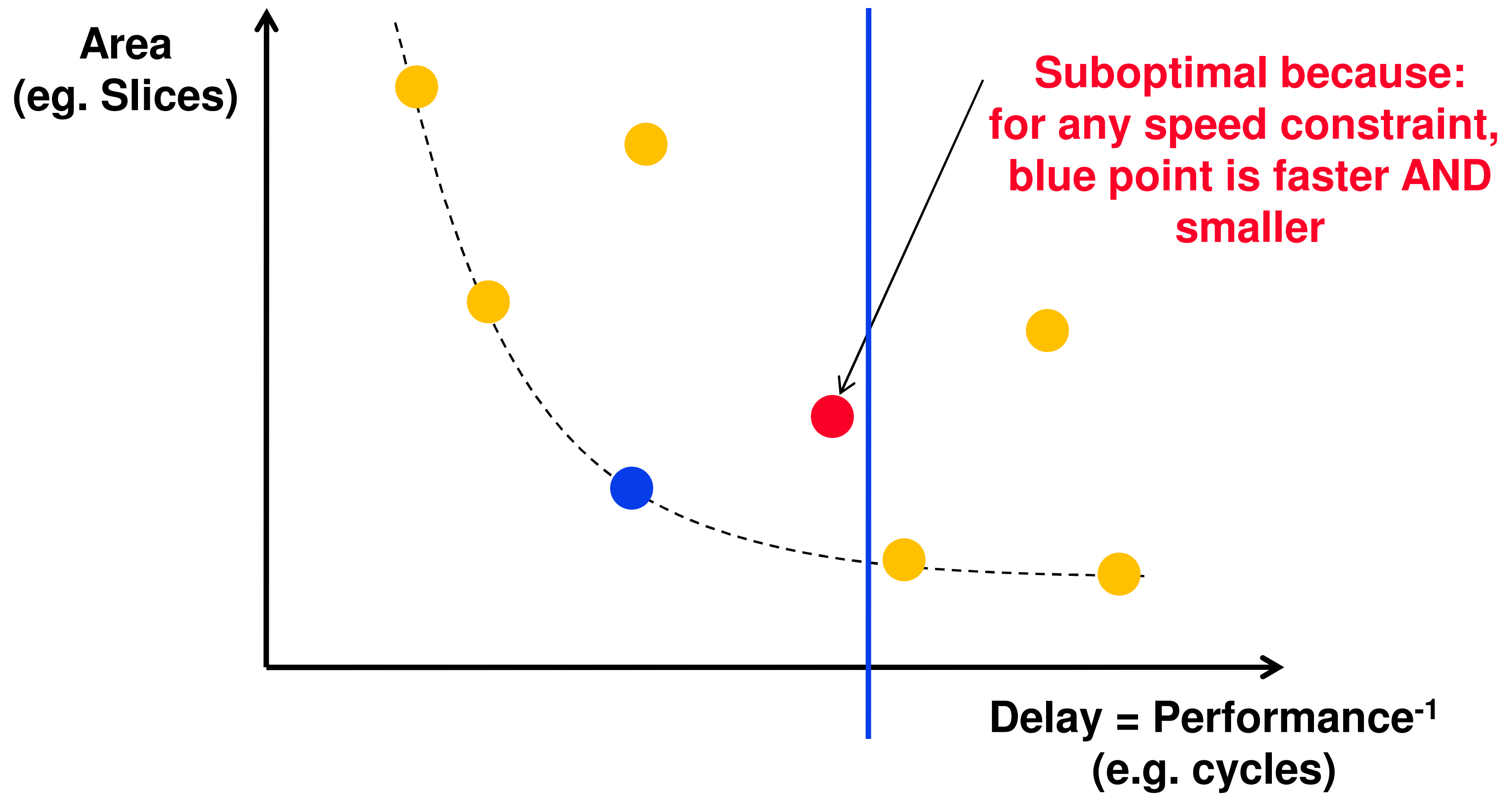
Area-Delay Product

- Sub-optimal points



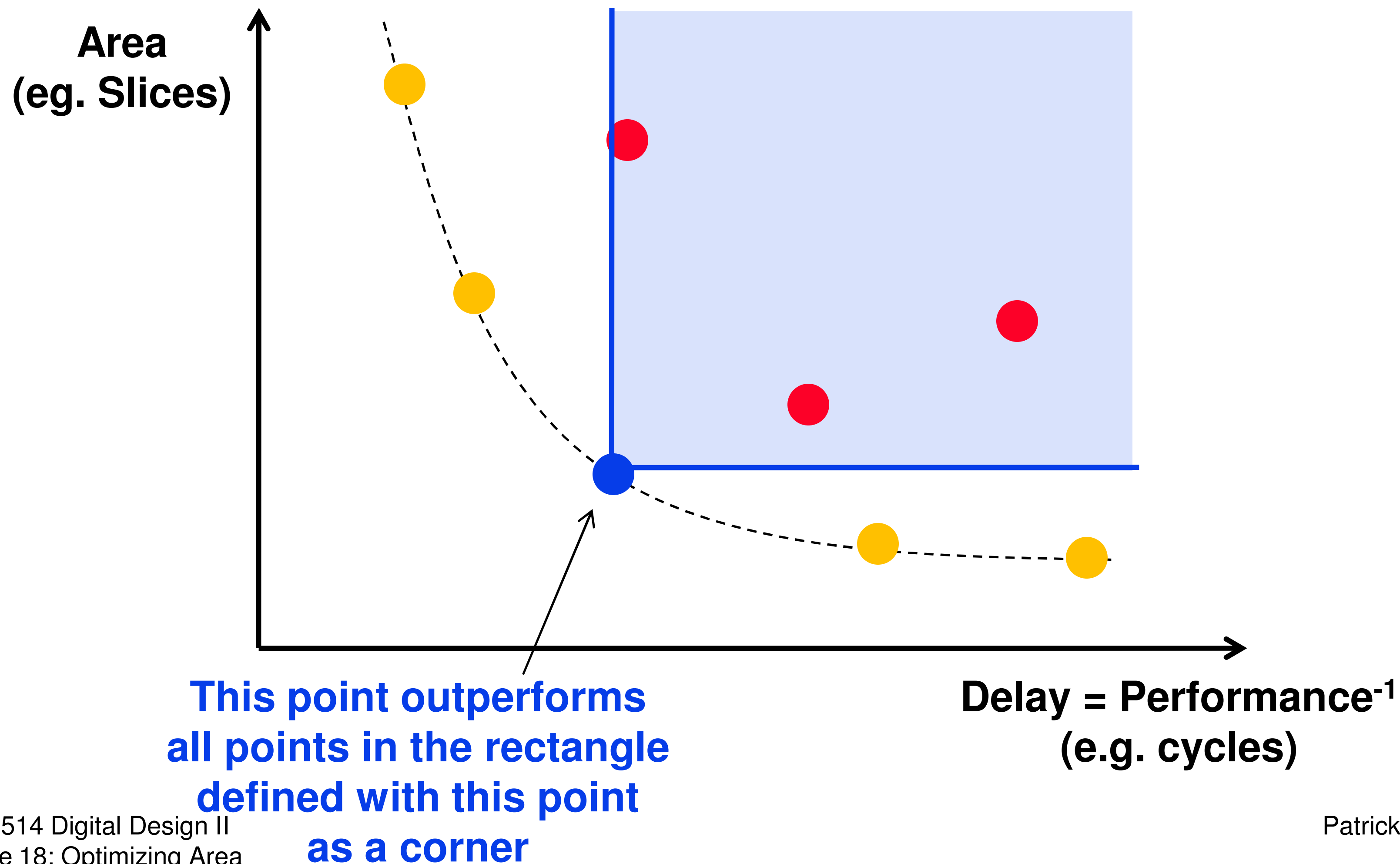
Area-Delay Product

- Sub-optimal points



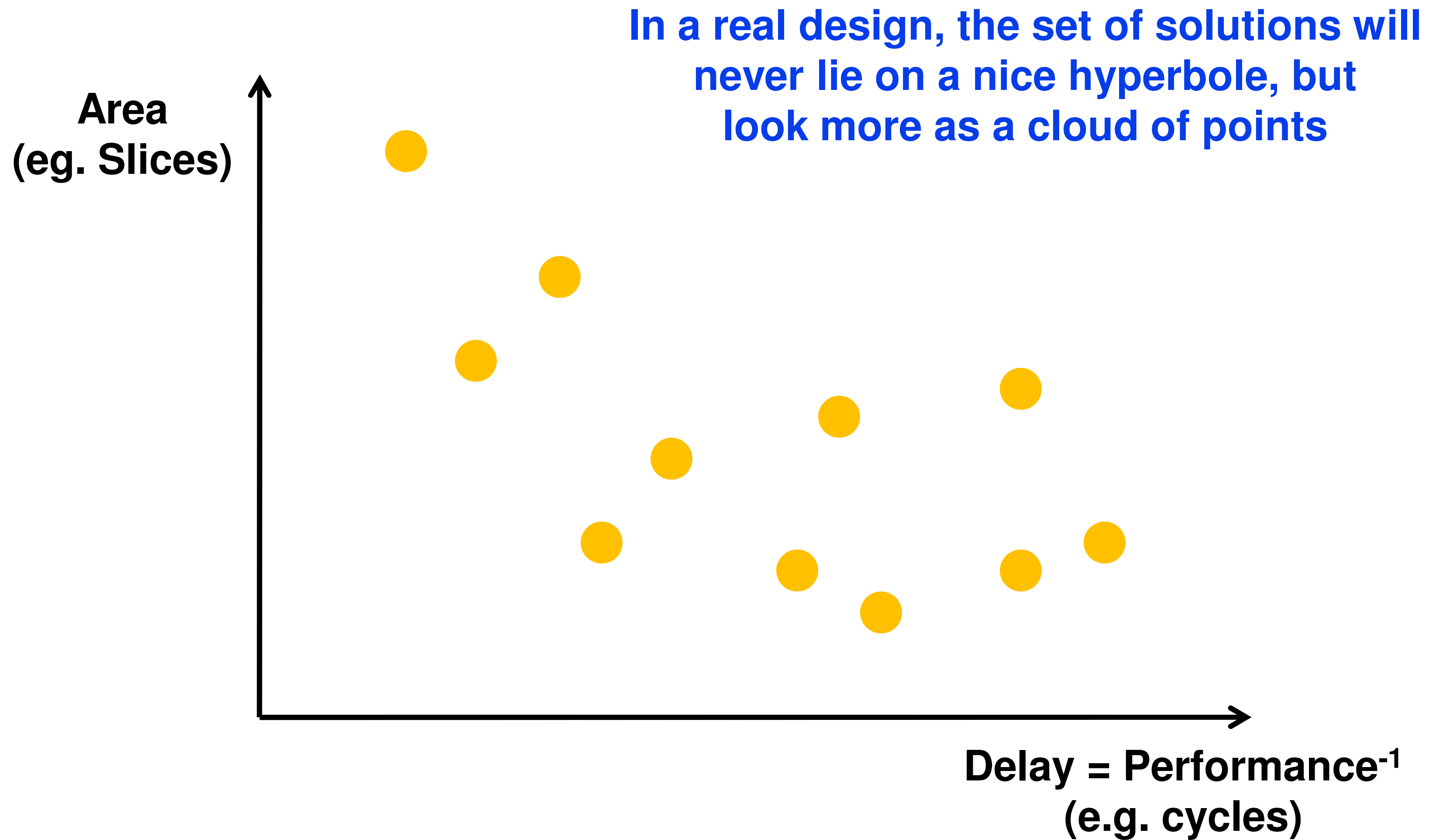
Area-Delay Product

- Optimal points dominate sub-optimal points



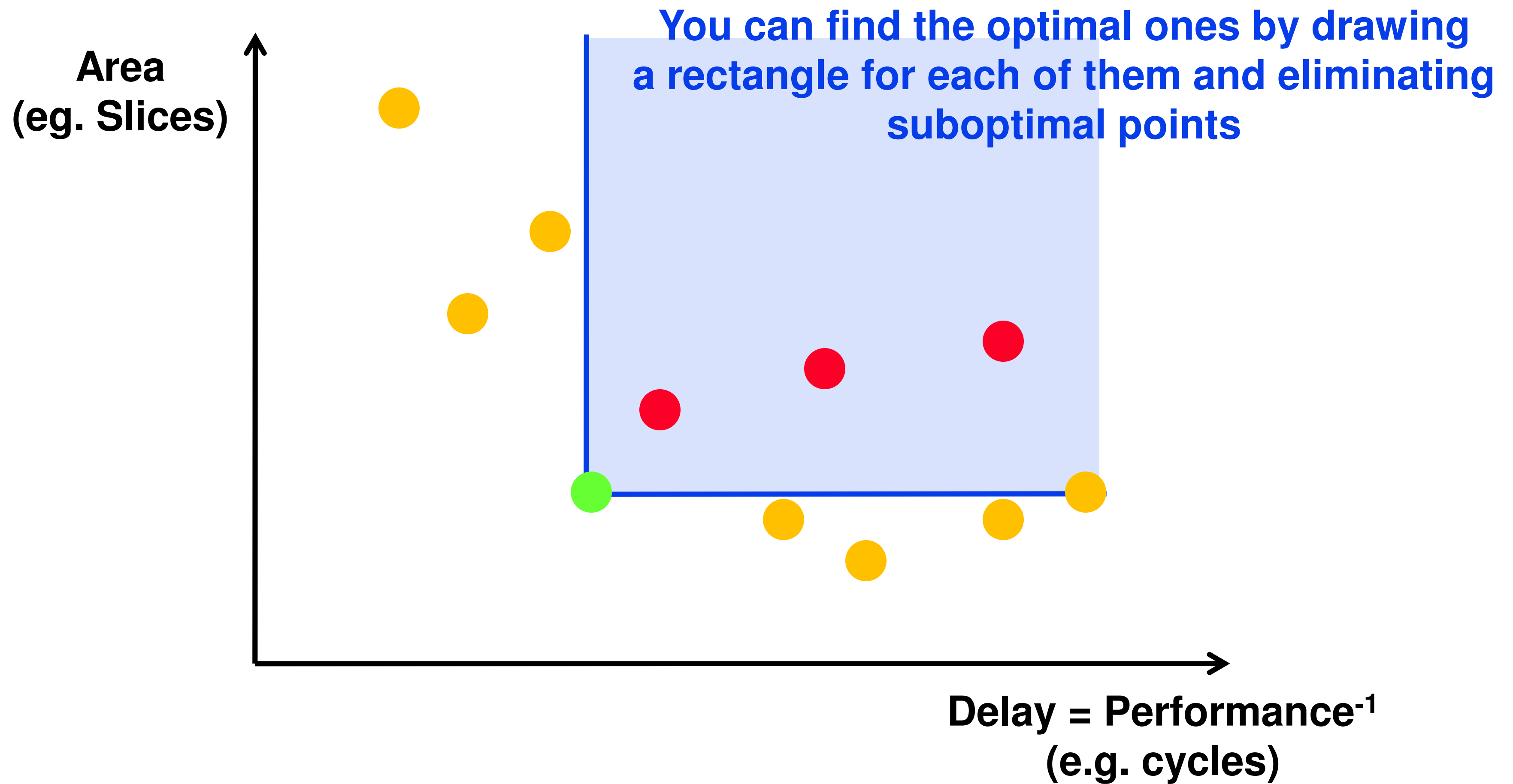
Area-Delay Product

- Finding all the optimal points for a random collection



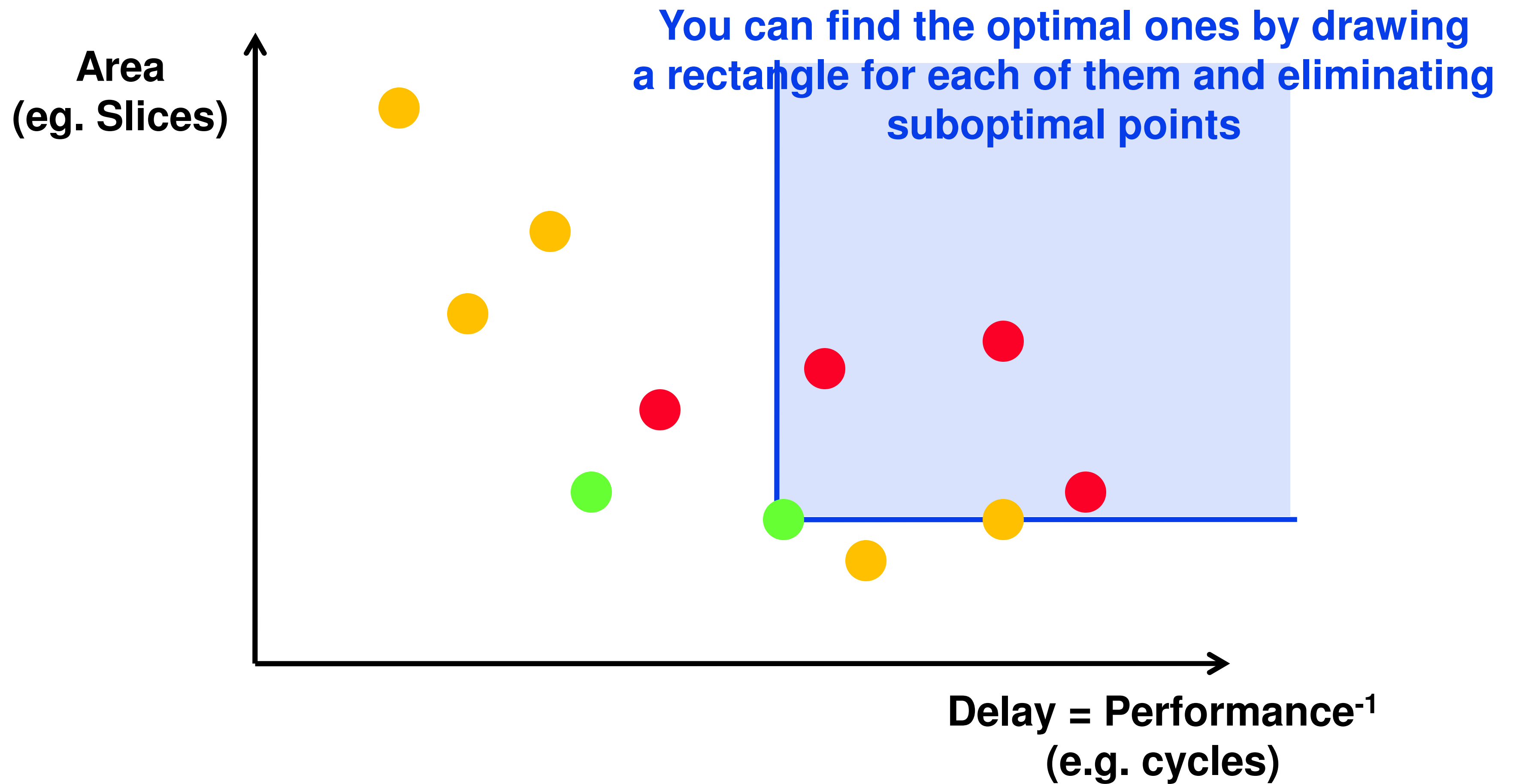
Area-Delay Product

- Finding all the optimal points for a random collection



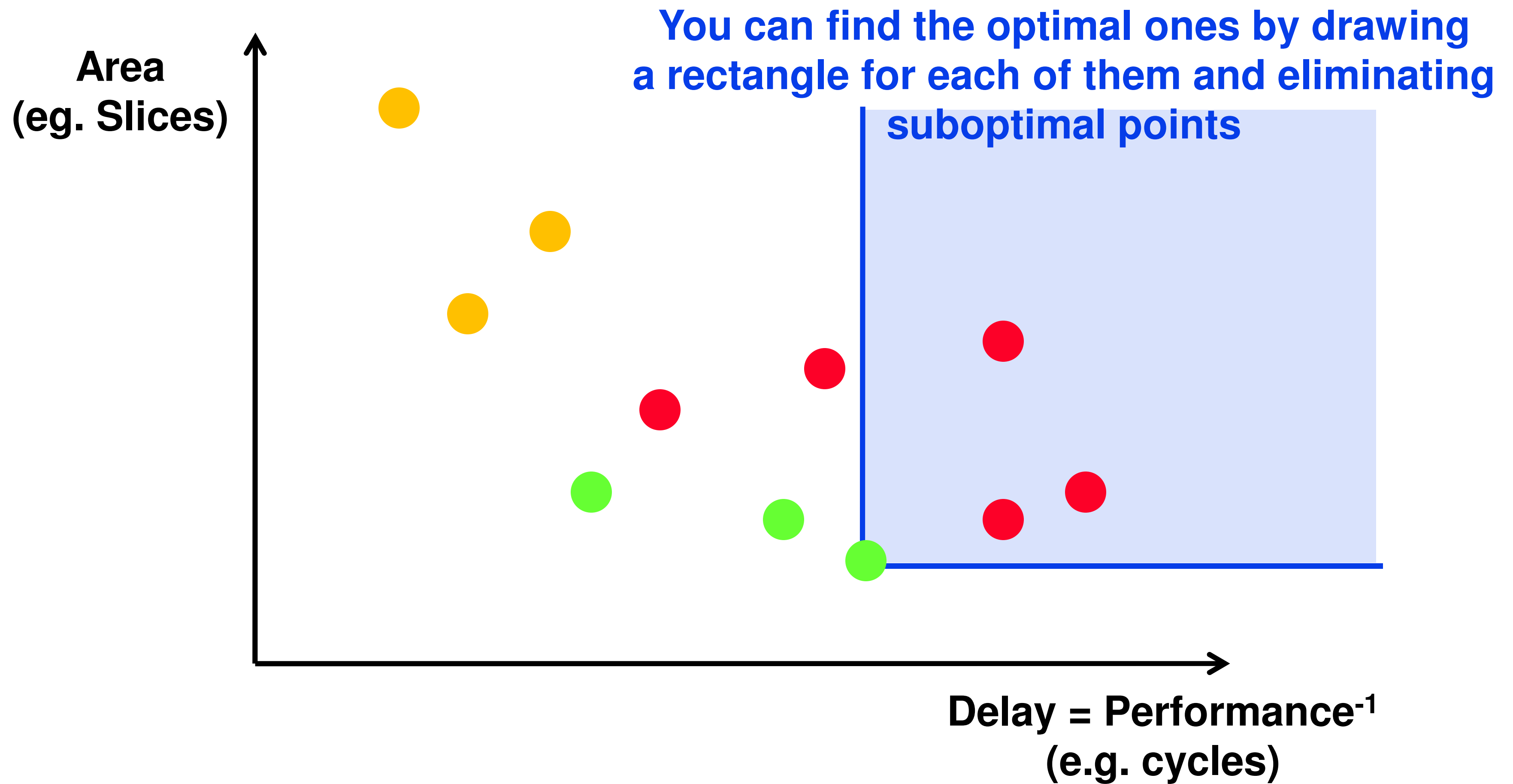
Area-Delay Product

- Finding all the optimal points for a random collection



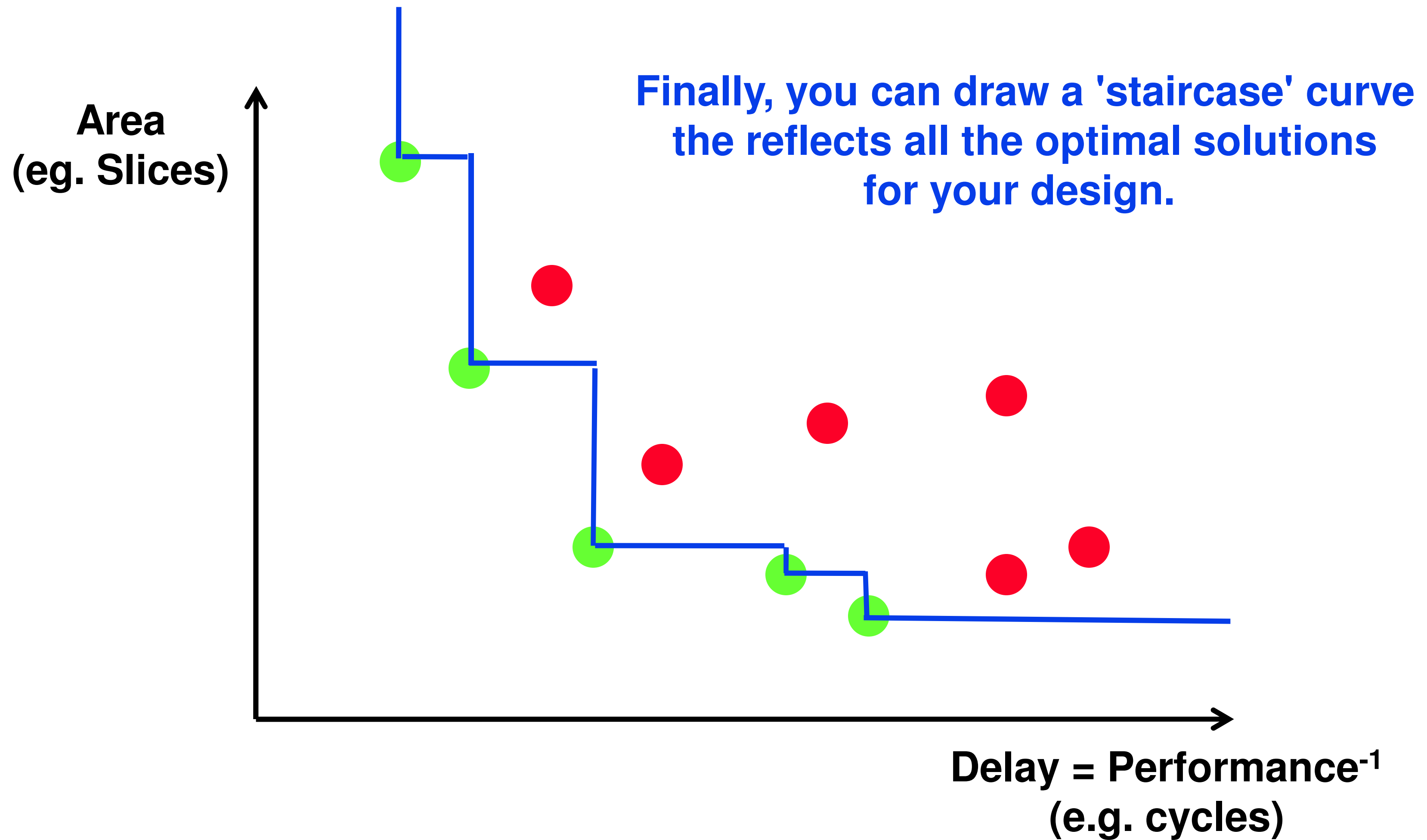
Area-Delay Product

- Finding all the optimal points for a random collection



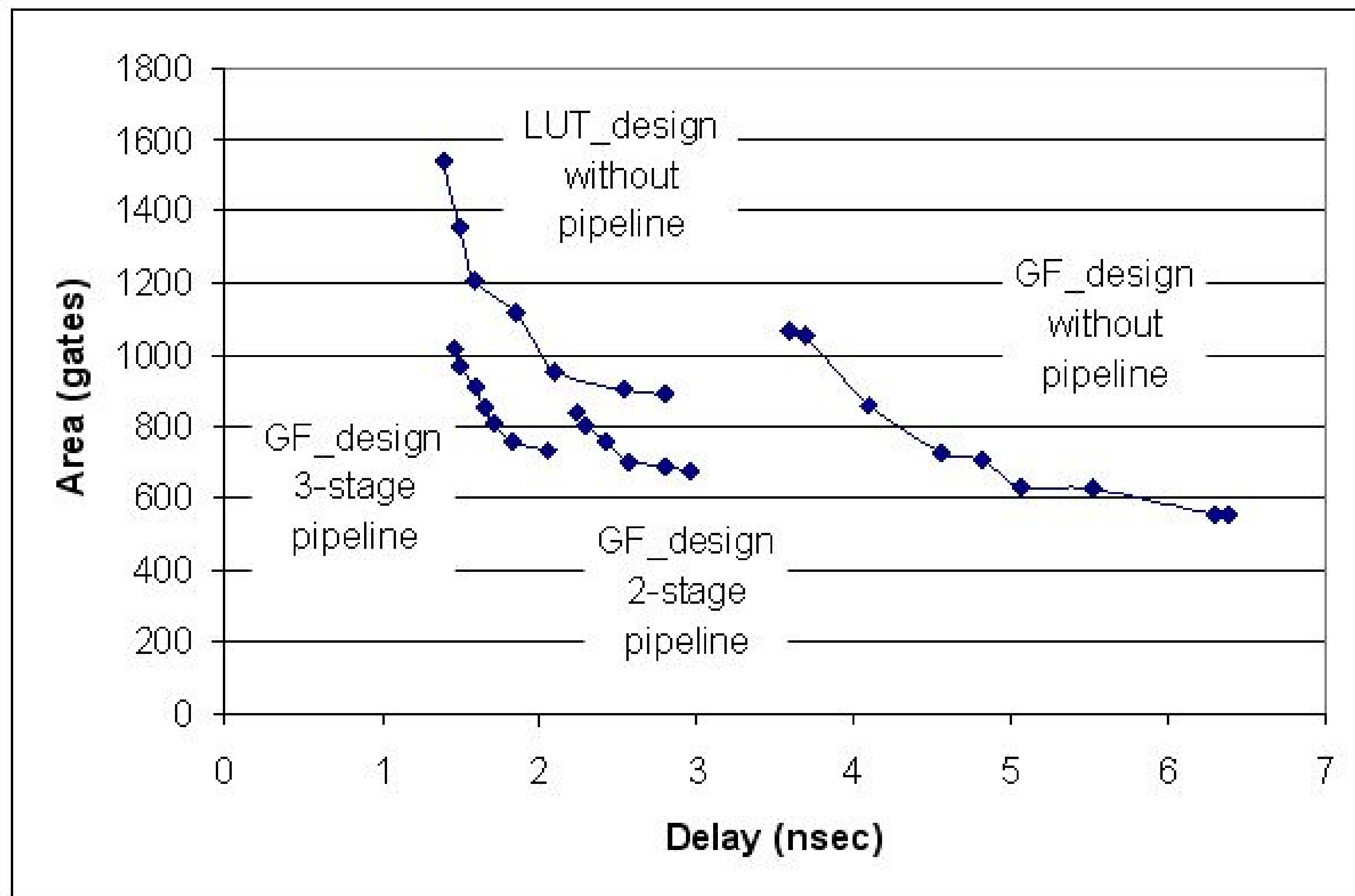
Area-Delay Product

- Finding all the optimal points for a random collection



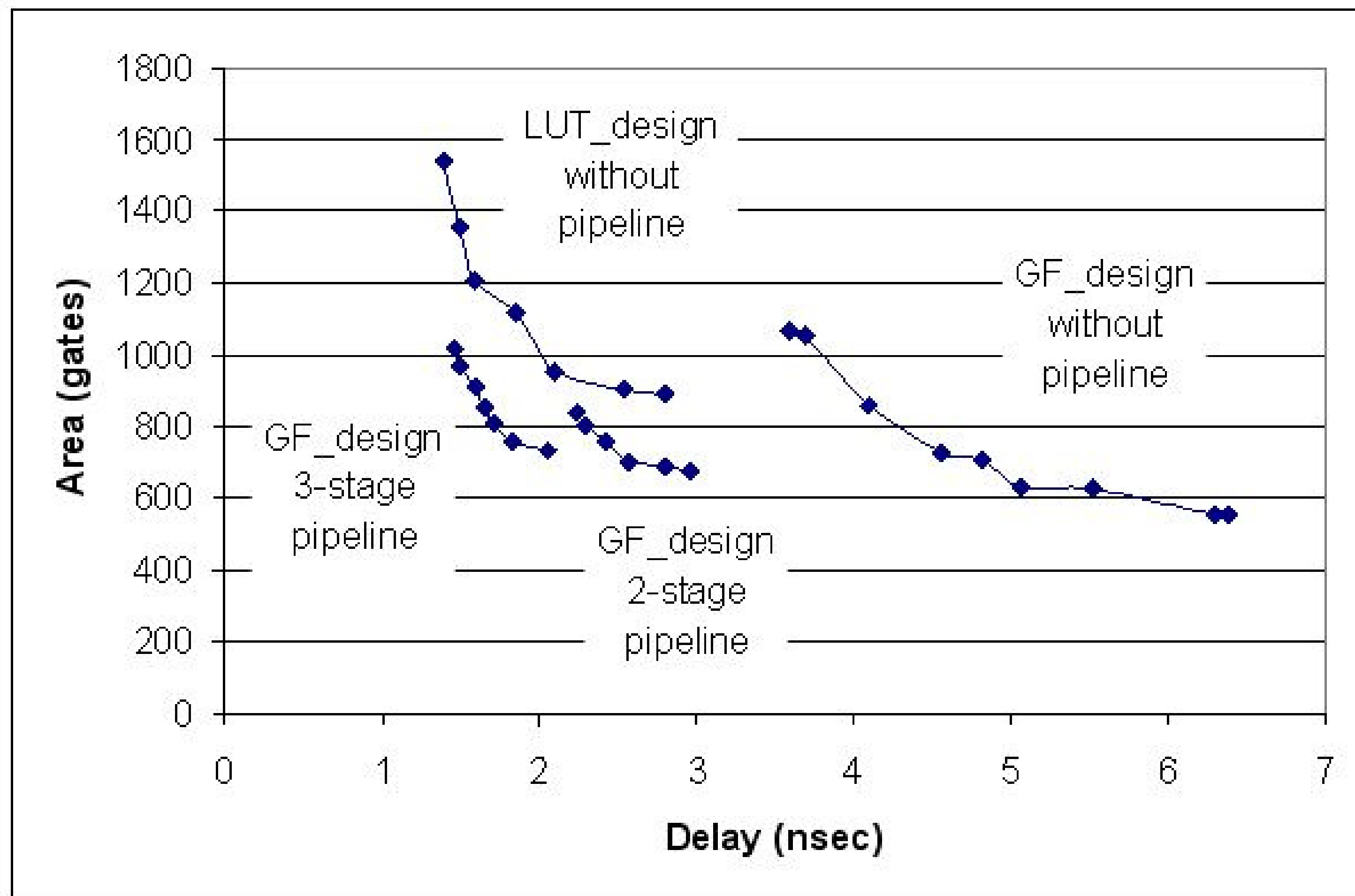
Area-Delay Example

- Area-delay product for an AES Sbox (component of an encryption algorithm) using different architectures



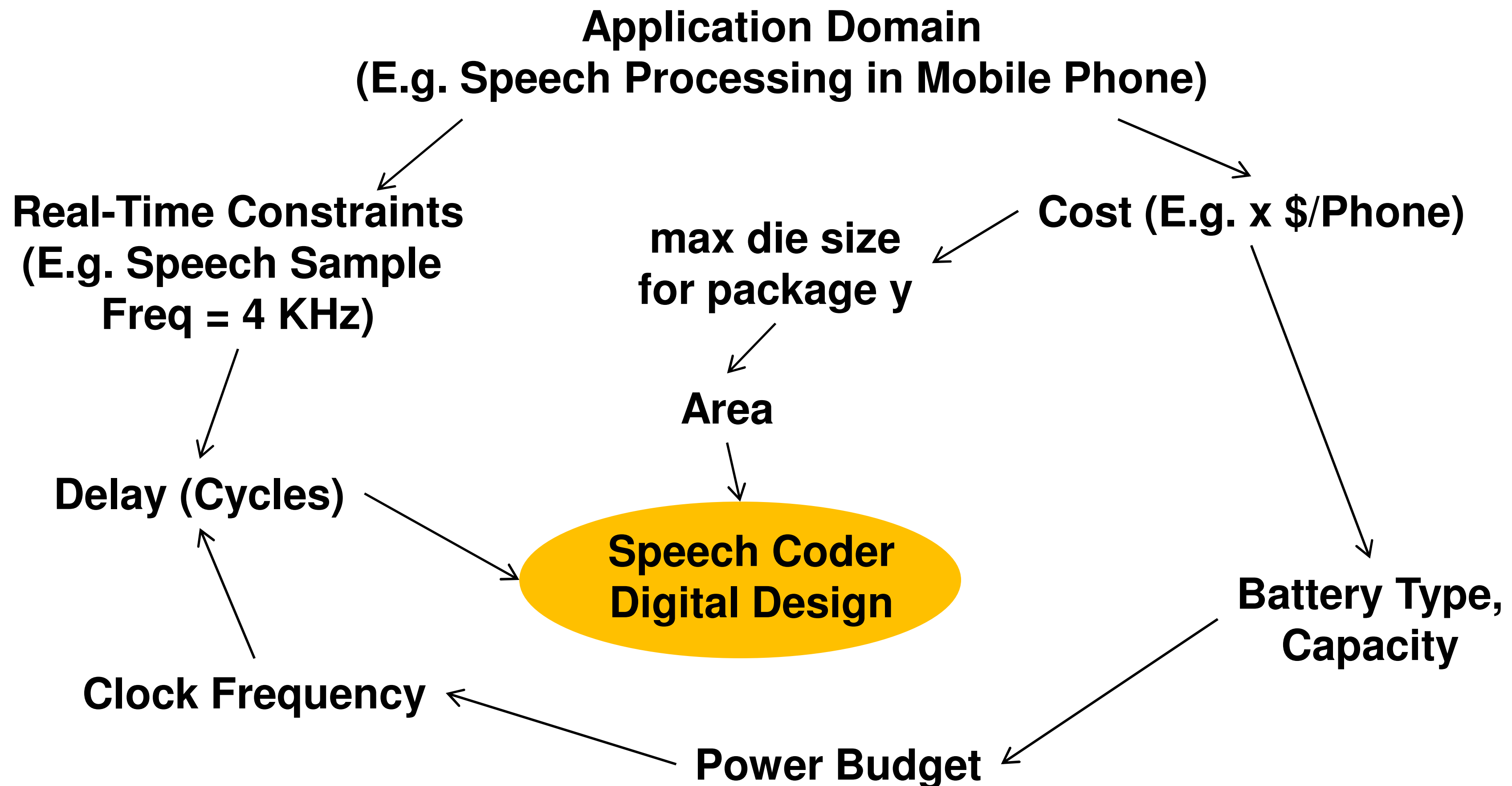
Area-Delay Example

- Area-delay product for an AES Sbox (component of an encryption algorithm) using different architectures



Where do constraints come from?

- Constraints are defined by the application domain or the system that will integrate a given digital design

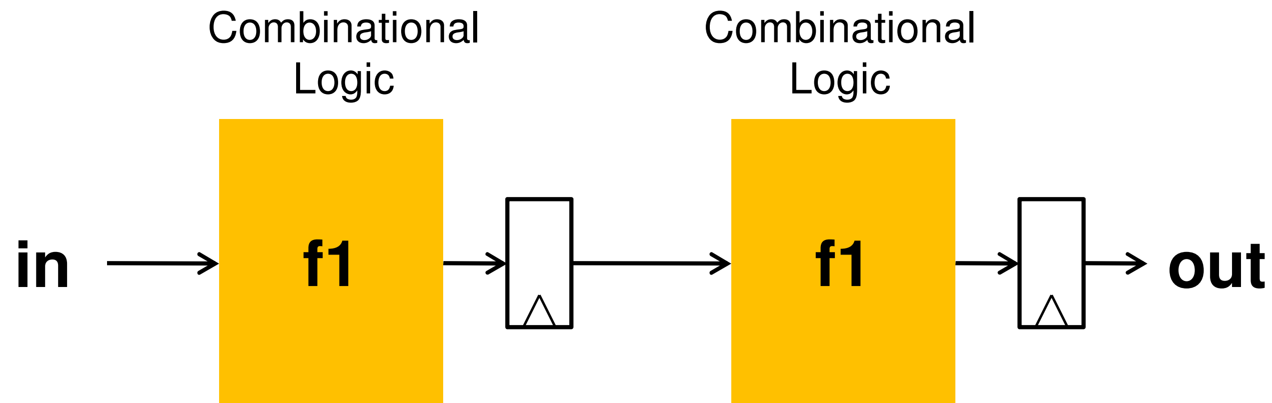


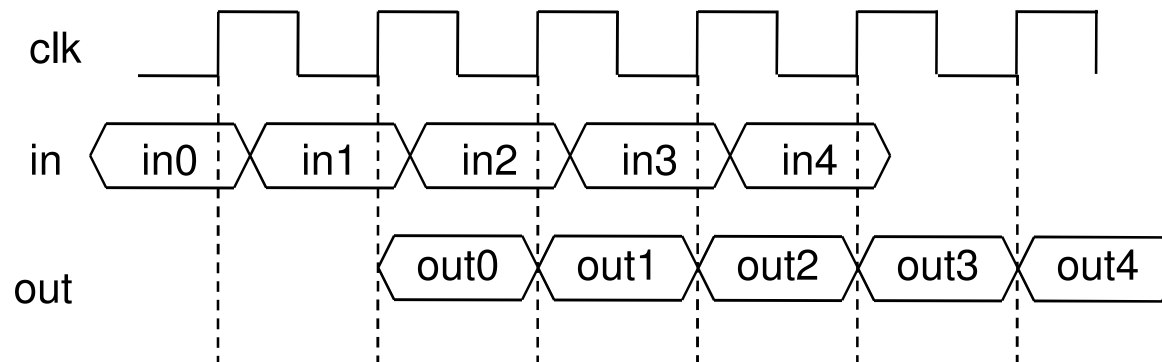
Area Optimization - Overview

- Resource Sharing
- Hardware Sharing Factor
- Examples:
  - An unshared multiplier
  - A shared multiplier
  - Multiplication with a constant

Resource Sharing

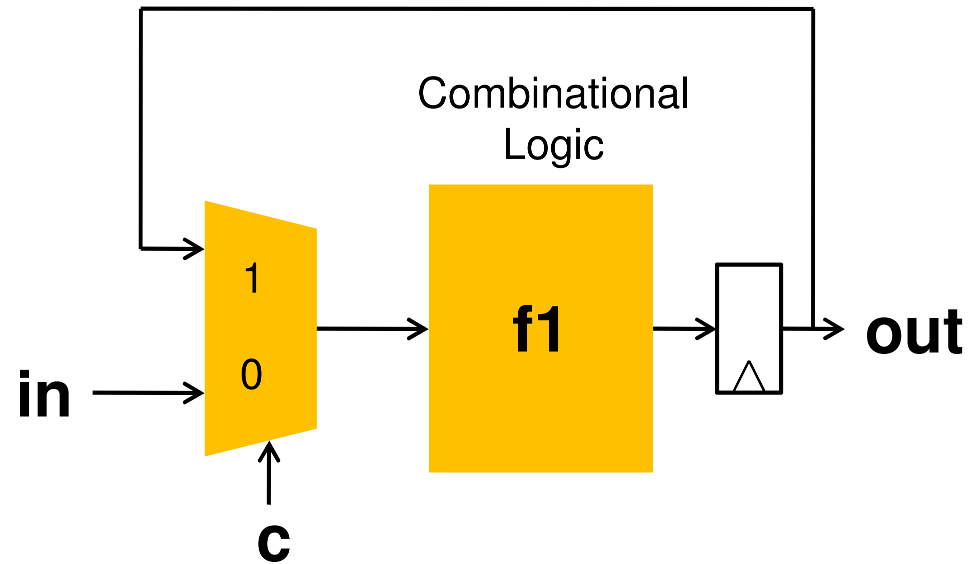
- Unshared case

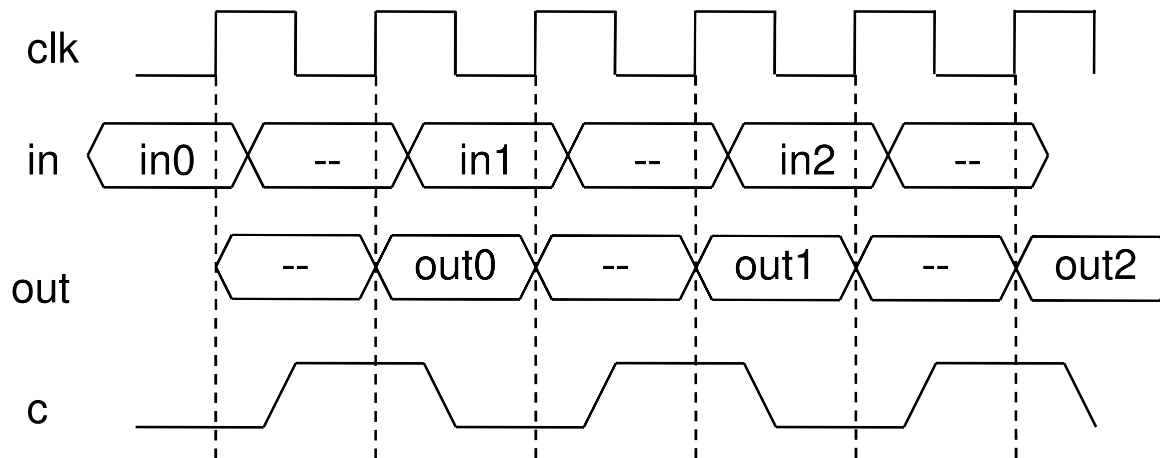




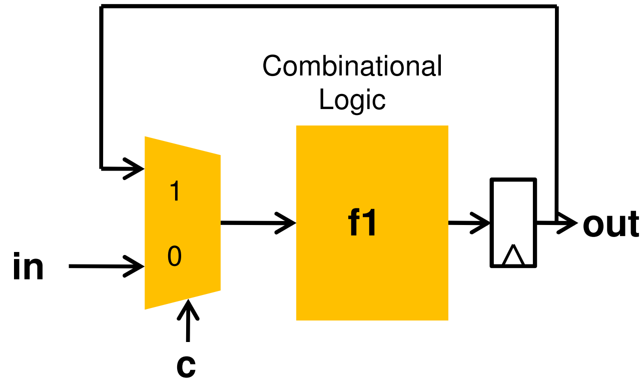
Resource Sharing

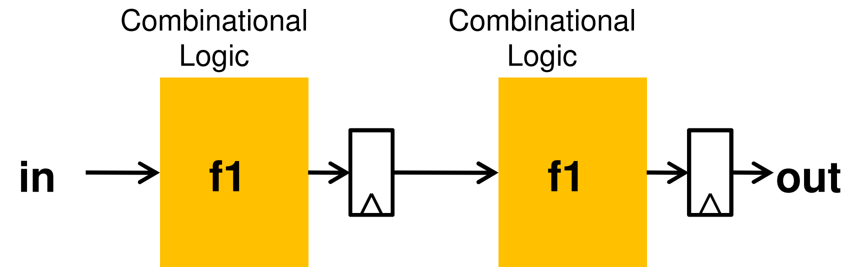
- Shared case





Resource Sharing





AREA 1x function f1, 1 multiplexer, 1 register

AREA 2x function f1, 2 registers

There will be area savings in the datapath when

$$A_{f1} + A_{mux} + A_{reg} < 2A_{f1} + 2A_{reg}$$

the c signal needs to be created as well

or

$$A_{f1} + A_{reg} > Amux$$

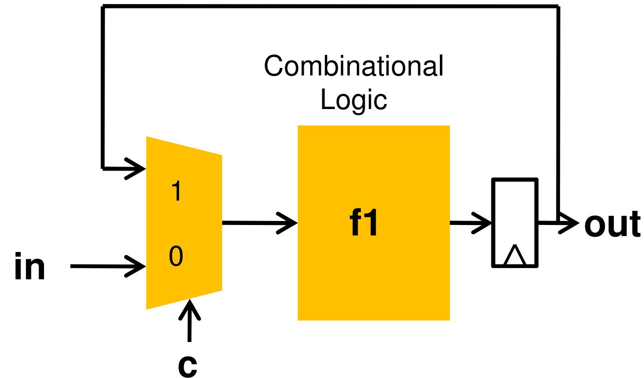
Resource Sharing

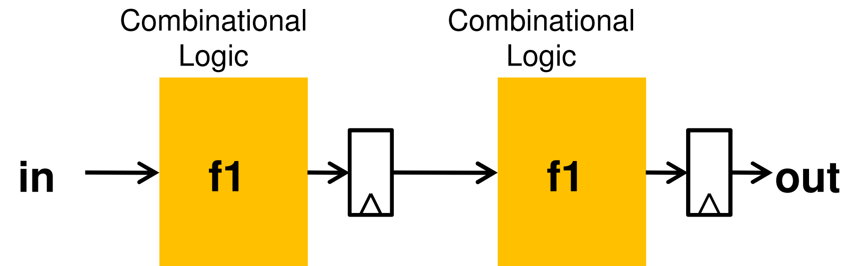
- Registers are relatively big compared to combinational logic (recall: 6 gates for a D flip-flop); while multiplexers are small (3 gates for a mux)
- Therefore, the equation is easy to satisfy

There will be area savings in the datapath when

$$A_{f1} + A_{reg} > Amux$$

Resource Sharing trade-off area & performance

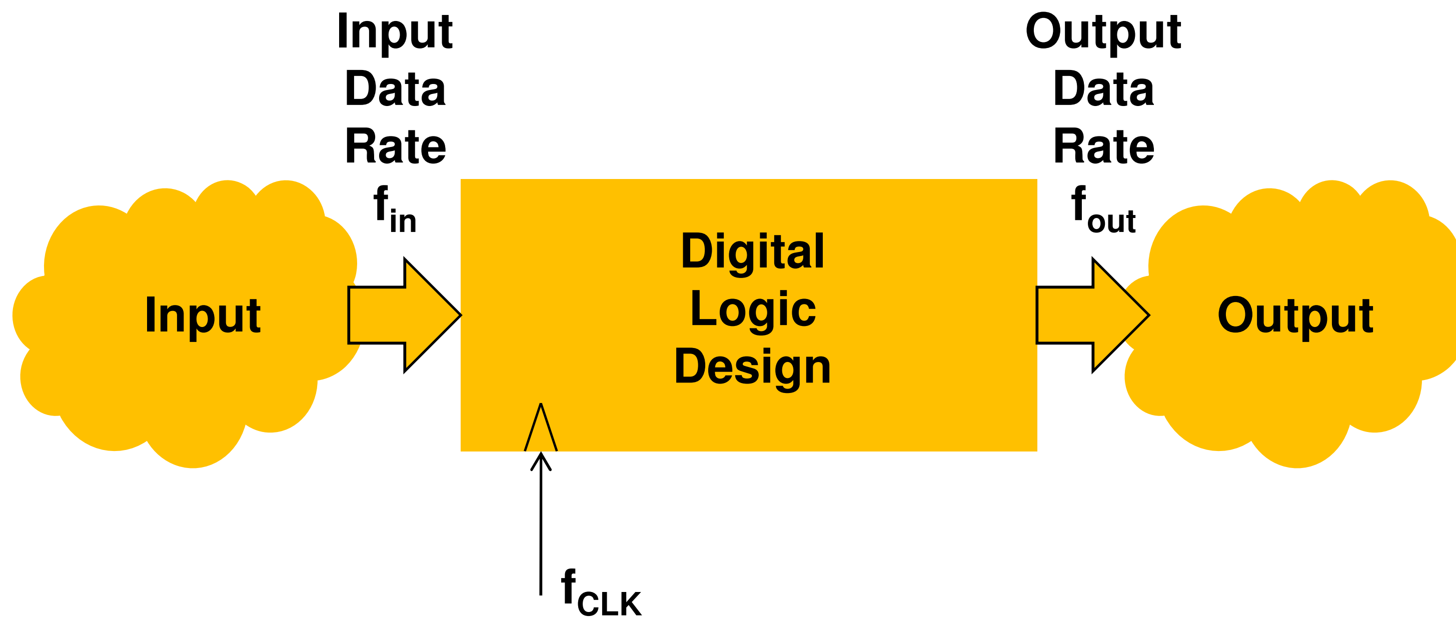




AREA	**1x function f1** **1 multiplexer** **1 register** **Smaller**	**2x function f1** **2 registers**
LATENCY	**Total compute time per input is two cycles**	**Total compute time per input is two cycles**
THROUGHPUT	**1 input each two cycles**	**1 input each cycle** **Faster**

Hardware Sharing Factor

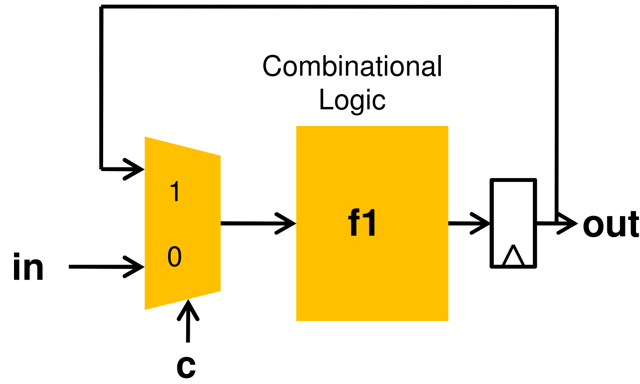
- The Hardware Sharing Factor (HSF) expresses the *potential* amount of resource sharing in a digital design

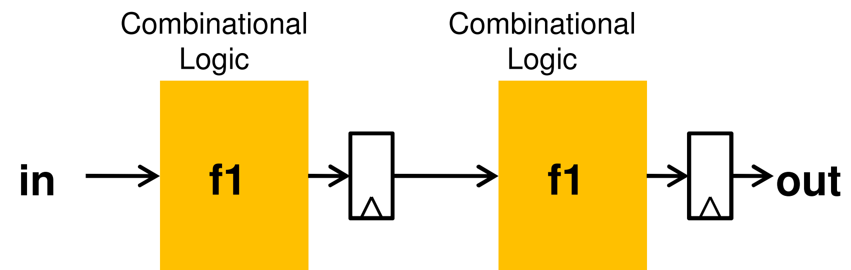


$$HSF = f_{CLK} / \max(f_{in}, f_{out})$$

HSF is the amount of clock cycles available per data item

Hardware Sharing Factor



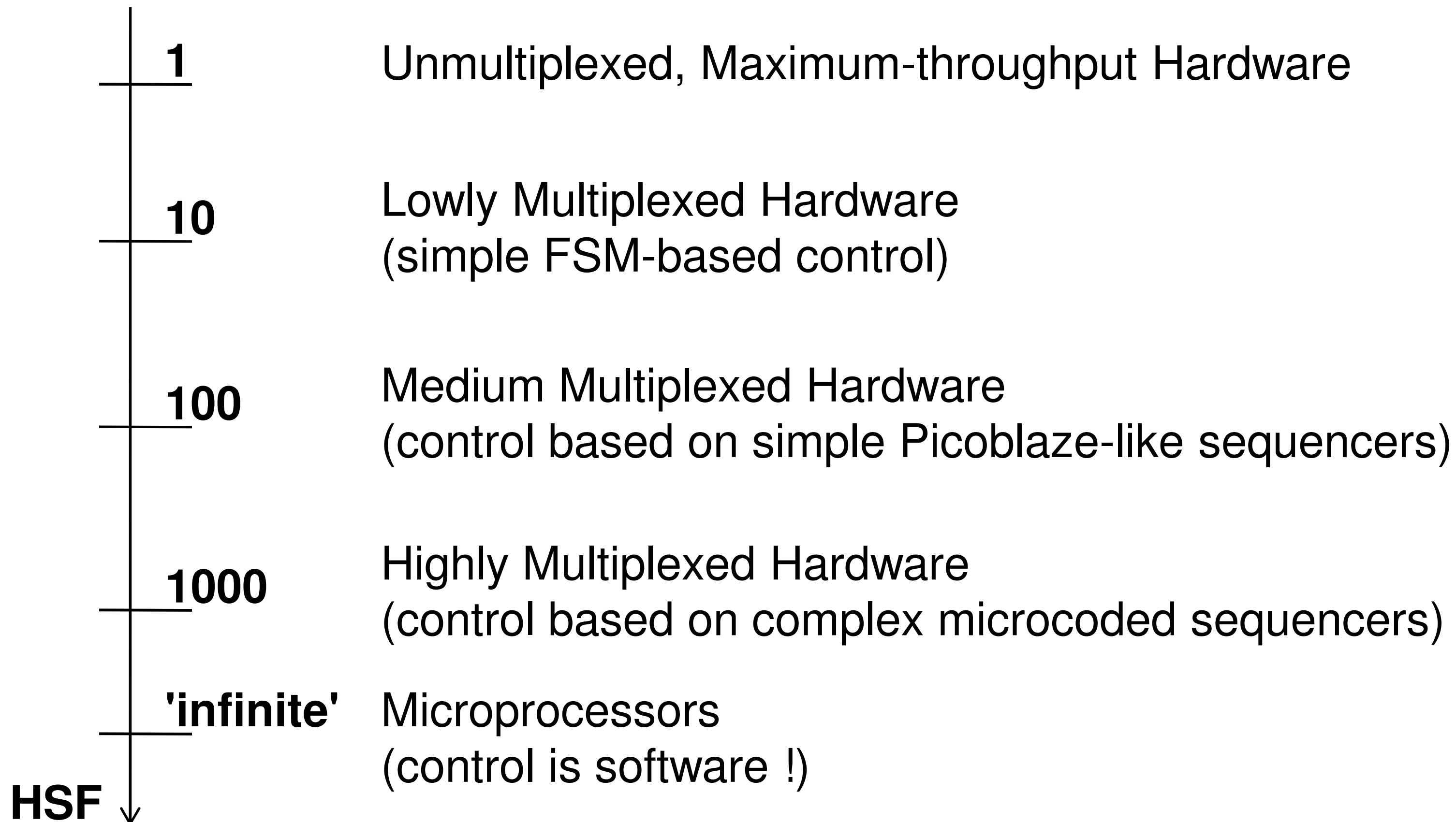


HSF = 2

HSF = 1

Hardware Sharing Factor

- Different styles of design will be reflected with a different HSF

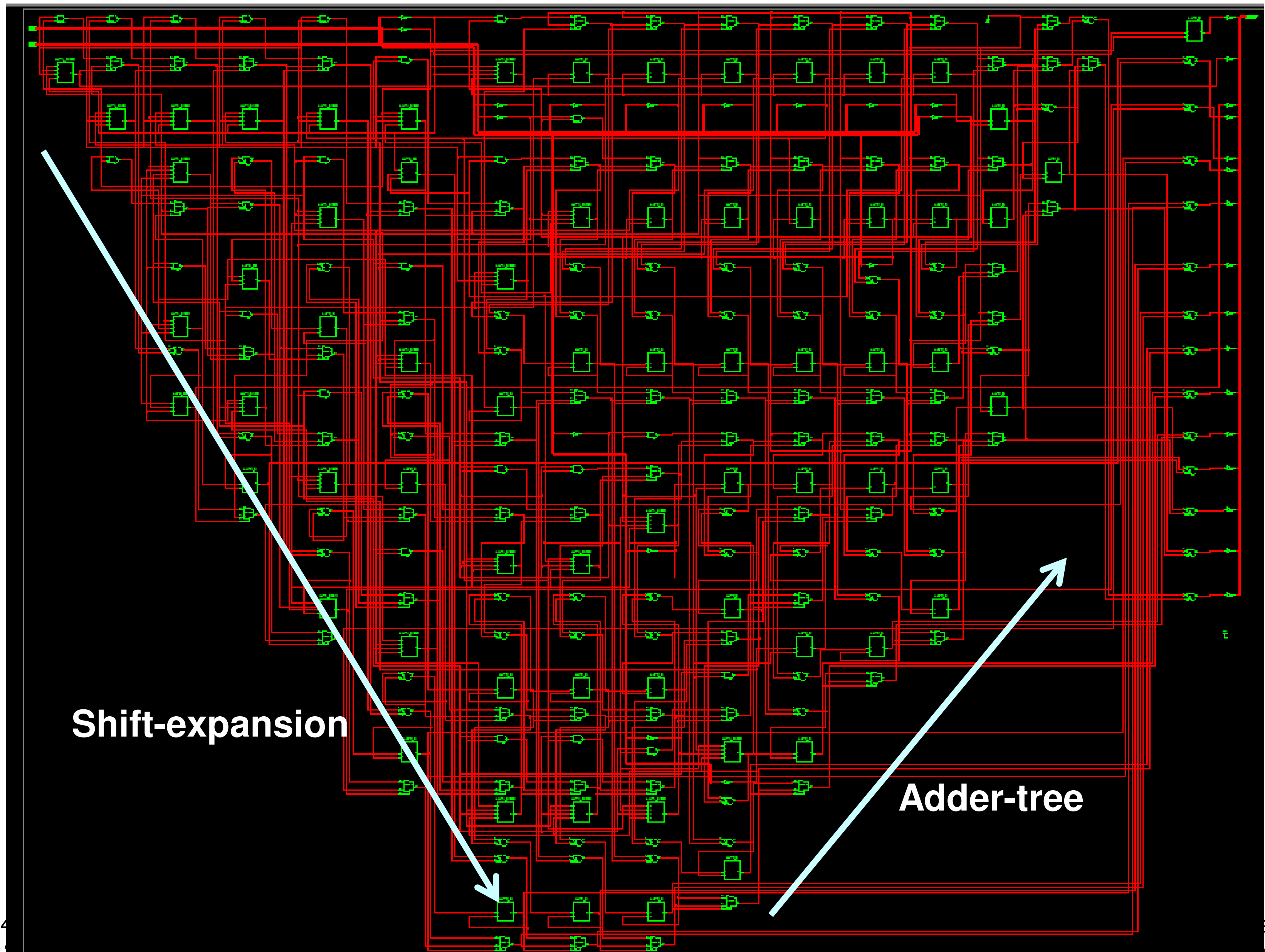


Example: An unshared multiplier

- Synthesize this multiplication into FPGA with multiplication 'expanded' into LUTs (as add-shift)

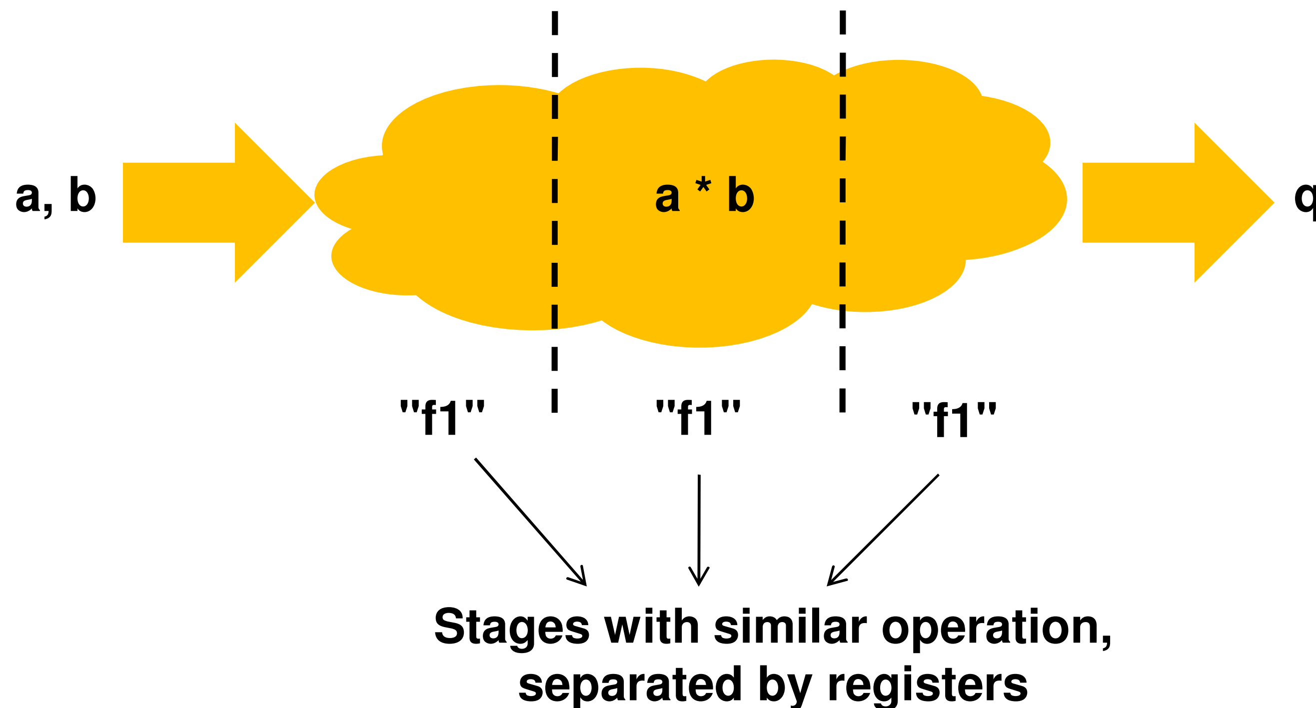
```
(* mult_style = "lut" *)

module syn_ex(q, a, b);
  output [15:0] q;
  input [7:0] a, b;

  assign q = a * b;

endmodule
```

Spartan3: 70 LUTs, 10.6 ns delay



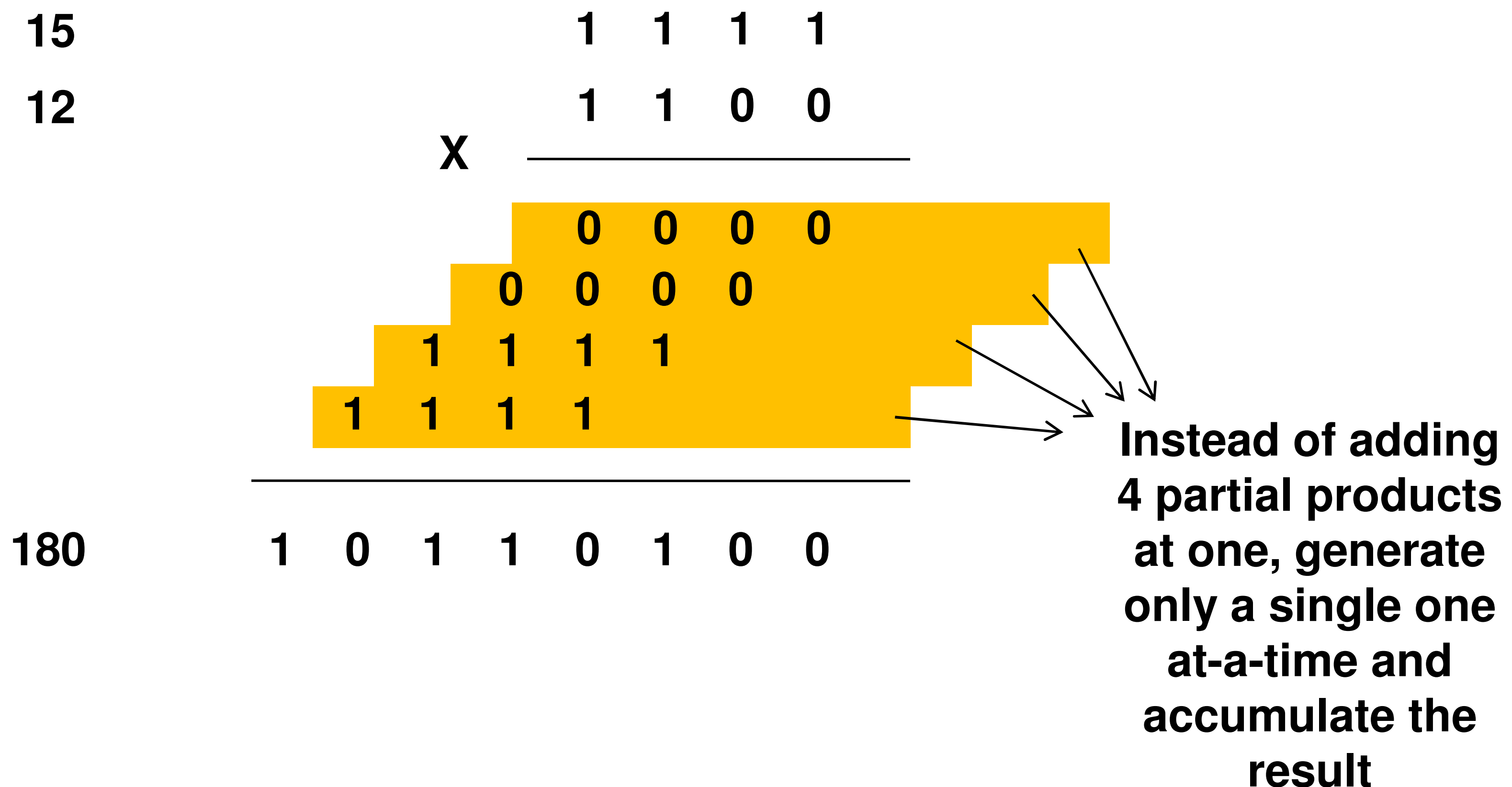
Example: An unshared multiplier

- The idea of a shared implementation is to 'chop' the combinational logic in smaller *similar* pieces, and execute these similar pieces over multiple clock cycles



Example: An unshared multiplier

- For a multiplication, repeated add-shift is the obvious 'repeating' part to be shared.

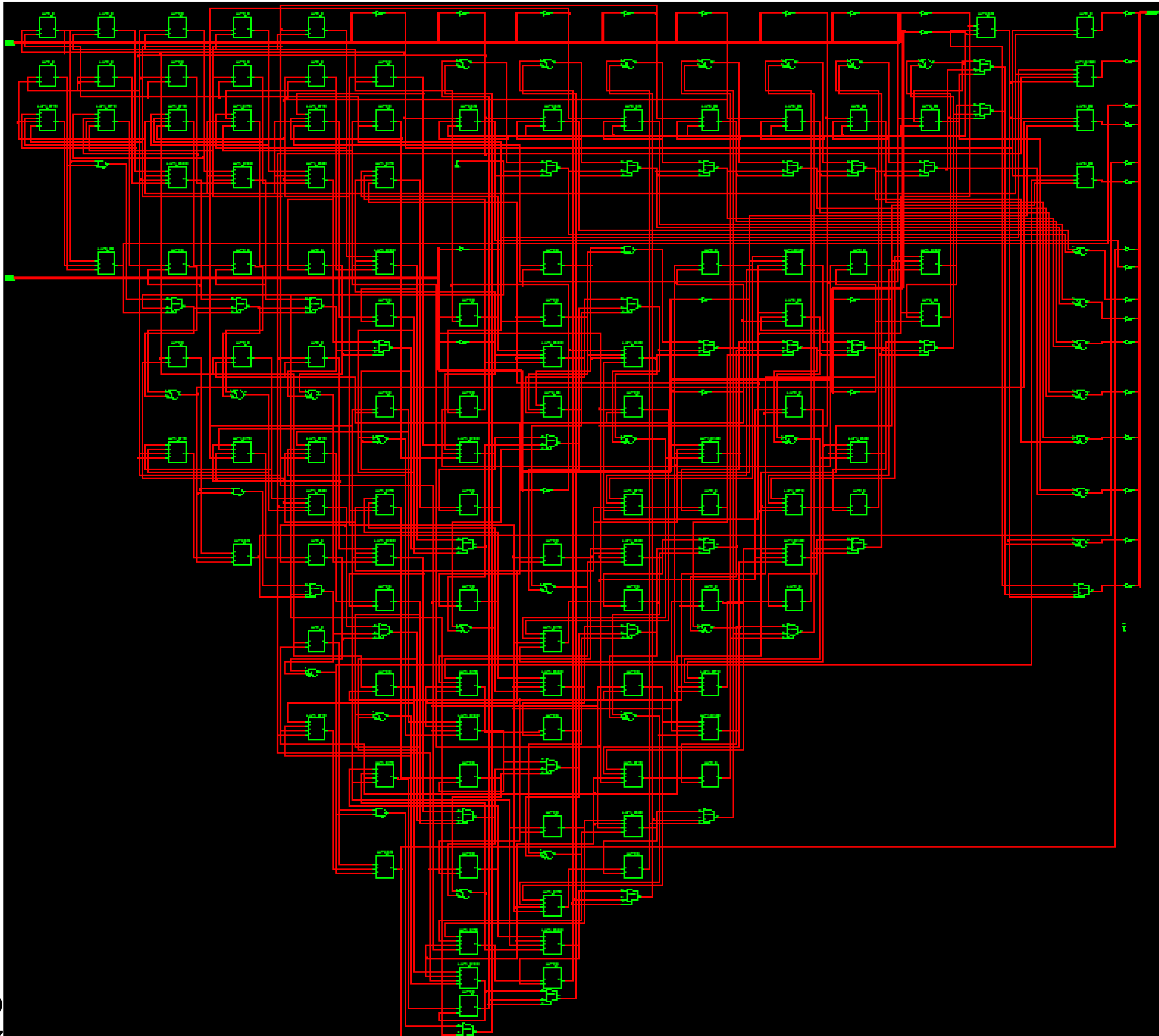


Example: An unshared multiplier

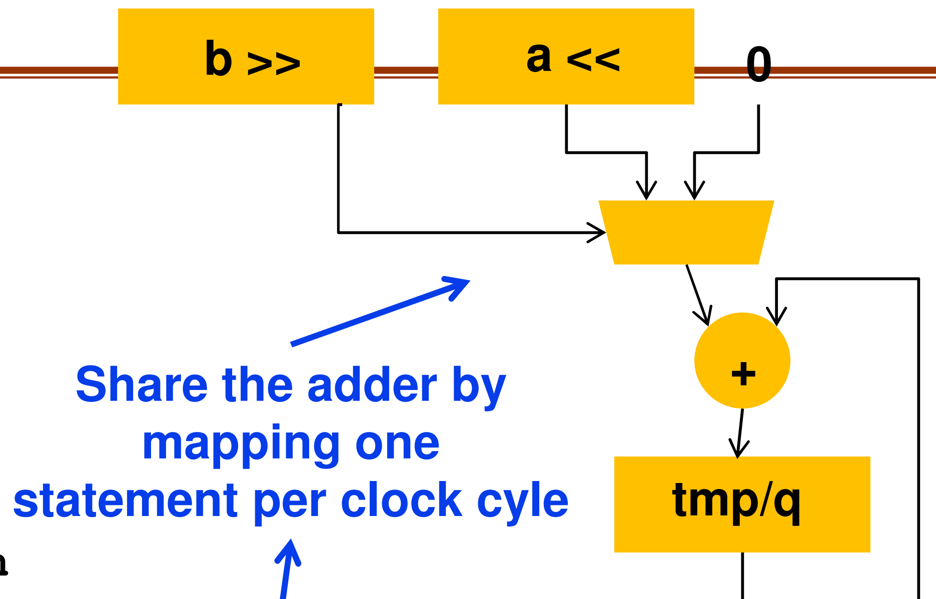
```
module syn_ex3(q, a, b);
  output [15:0] q;
  reg    [15:0] q;
  input  [7:0]  a, b;

  reg [15:0] tmp [7:0];

  always @(a or b) begin
    tmp[0] = b[0] ? a : 15'b0;
    tmp[1] = tmp[0] + (b[1] ? {a, 1'b0} : 15'b0);
    tmp[2] = tmp[1] + (b[2] ? {a, 2'b0} : 15'b0);
    tmp[3] = tmp[2] + (b[3] ? {a, 3'b0} : 15'b0);
    tmp[4] = tmp[3] + (b[4] ? {a, 4'b0} : 15'b0);
    tmp[5] = tmp[4] + (b[5] ? {a, 5'b0} : 15'b0);
    tmp[6] = tmp[5] + (b[6] ? {a, 6'b0} : 15'b0);
    tmp[7] = tmp[6] + (b[7] ? {a, 7'b0} : 15'b0);
    q      = tmp[7];
  end
endmodule
```

8 * 8 bit multiplication written using add-shift

Spartan3: 108 LUTs, 20ns delay ..



Shared multiplier




```
module syn_ex3(q, a, b);
  output [15:0] q;
  reg    [15:0] q;
  input  [7:0]  a, b;

  reg [15:0] tmp [7:0];

  always @(a or b) begin
    tmp[0] = b[0] ? a : 15'b0;
    tmp[1] = tmp[0] + (b[1] ? {a, 1'b0} : 15'b0);
    tmp[2] = tmp[1] + (b[2] ? {a, 2'b0} : 15'b0);
    tmp[3] = tmp[2] + (b[3] ? {a, 3'b0} : 15'b0);
    tmp[4] = tmp[3] + (b[4] ? {a, 4'b0} : 15'b0);
    tmp[5] = tmp[4] + (b[5] ? {a, 5'b0} : 15'b0);
    tmp[6] = tmp[5] + (b[6] ? {a, 6'b0} : 15'b0);
    tmp[7] = tmp[6] + (b[7] ? {a, 7'b0} : 15'b0);
    q      = tmp[7];
  end
endmodule
```

Share the adder by mapping one statement per clock cyle

Shared multiplier

```
module syn_ex4(q, done, a, b, start, clk);
  output [15:0] q;
  reg    [15:0] q;
  output  done;
  input  [ 7:0]  a, b;
  input  start;
  input  clk;
  reg [ 4:0]  ctr;
  reg [15:0]  shiftA;
  reg [ 7:0]  shiftB;
  always @(posedge clk) begin
    ctr    <= (start) ? 5'b0 : ((!done) ? ctr + 1 : ctr);
    q      <= (start) ? 15'b0 :
              (!done) ? (shiftB[0] ? q + shiftA : q) : q;
    shiftB <= (start) ? b : {1'b0, shiftB[7:1]};
    shiftA <= (start) ? a : {shiftA[14:0], 1'b0};
  end
  assign done = ctr[3];
endmodule
```

Shared multiplier

```
module syn_ex4(q, done, a, b, start, clk);
  output [15:0] q;
  reg    [15:0] q;
  output done;
  input  [7:0]  a, b;
  input  start;
  input  clk;
  reg [ 4:0]  ctr;
  reg [15:0]  shiftA;
  reg [ 7:0]  shiftB;
  always @(posedge clk) begin
    ctr    <= (start) ? 5'b0 : ((!done) ? ctr + 1 : ctr);
    q      <= (start) ? 15'b0 :
              (!done) ? (shiftB[0] ? q + shiftA : q) : q;
    shiftB <= (start) ? b : {1'b0, shiftB[7:1]};
    shiftA <= (start) ? a : {shiftA[14:0], 1'b0};
  end
  assign done = ctr[3];
endmodule
```

Controller

Shared multiplier

```
module syn_ex4(q, done, a, b, start, clk);
  output [15:0] q;
  reg    [15:0] q;
  output done;
  input  [7:0]  a, b;
  input  start;
  input  clk;
  reg [ 4:0]  ctr;
  reg [15:0]  shiftA;
  reg [ 7:0]  shiftB;
  always @(posedge clk) begin
    ctr    <= (start) ? 5'b0 : ((!done) ? ctr + 1 : ctr);
    q      <= (start) ? 15'b0 :
              (!done) ? (shiftB[0] ? q + shiftA : q) : q;
    shiftB <= (start) ? b : {1'b0, shiftB[7:1]};
    shiftA <= (start) ? a : {shiftA[14:0], 1'b0};
  end
  assign done = ctr[3];
endmodule
```

B shifts down, A shifts up

Shared multiplier

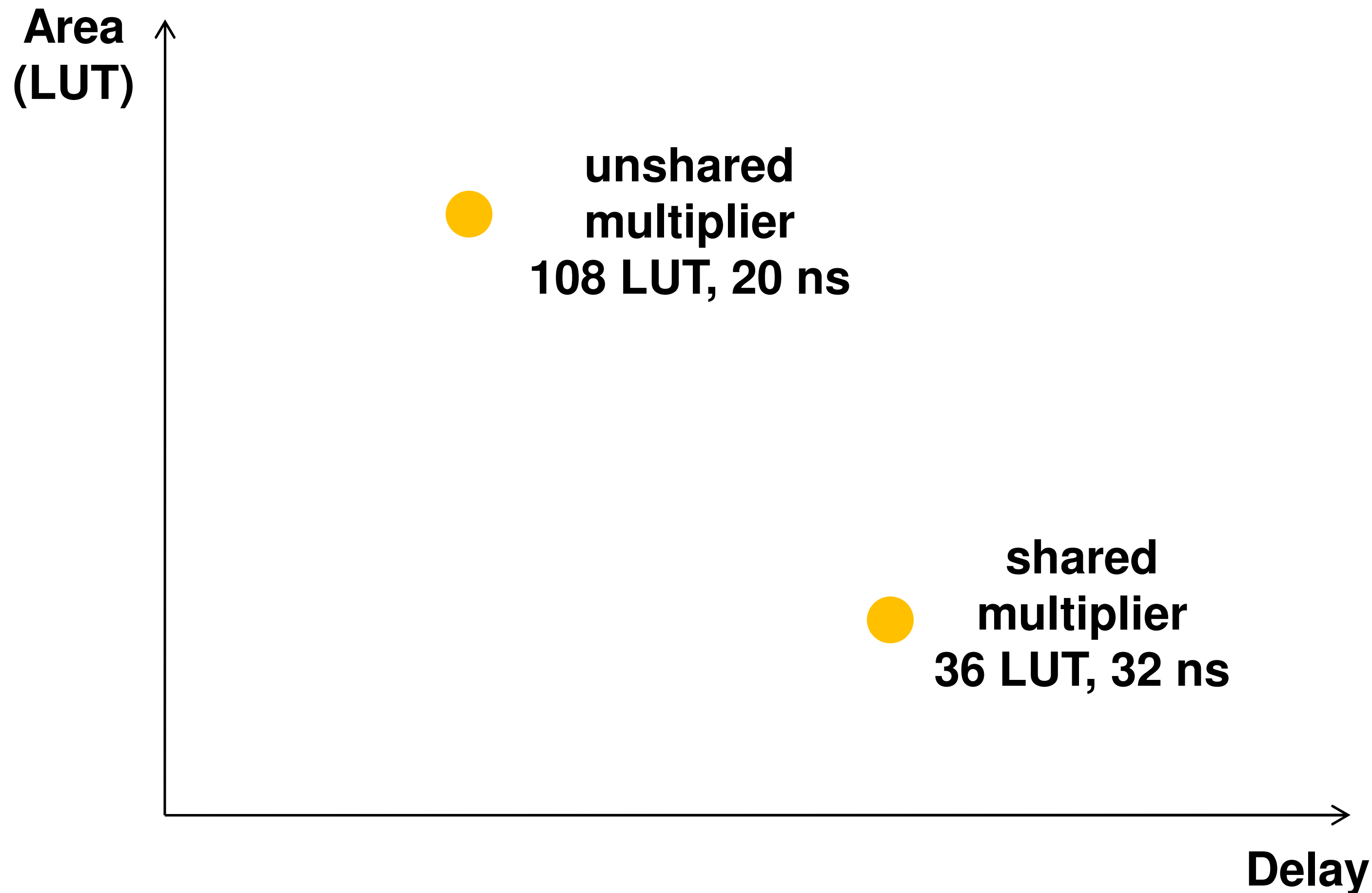
Accumulator

```
module syn_ex4(q, done, a, b, start, clk);
  output [15:0] q;
  reg    [15:0] q;
  output done;
  input  [7:0]  a, b;
  input  start;
  input  clk;
  reg [ 4:0]  ctr;
  reg [15:0]  shiftA;
  reg [ 7:0]  shiftB;
  always @(posedge clk) begin
    ctr    <= (start) ? 5'b0 : ((!done) ? ctr + 1 : ctr);
    q      <= (start) ? 15'b0 :
              (!done) ? (shiftB[0] ? q + shiftA : q) : q;
    shiftB <= (start) ? b : {1'b0, shiftB[7:1]};
    shiftA <= (start) ? a : {shiftA[14:0], 1'b0};
  end
  assign done = ctr[3];
endmodule
```

Spartan3: 36 LUTs, 44 FF, 4.0 ns delay

Full Multiplication still takes 8 * 4 = 32ns

Sharing may reduce area at expense of delay



Other Solutions are possible ...

```
module syn_ex3(q, a, b);
  output [15:0] q;
  reg    [15:0] q;
  input  [7:0]  a, b;

  reg [15:0] tmp [7:0];

  always @(a or b) begin
    tmp[0] = b[0] ? a : 15'b0;
    tmp[1] = tmp[0] + (b[1] ? {a, 1'b0} : 15'b0);
    tmp[2] = tmp[1] + (b[2] ? {a, 2'b0} : 15'b0);
    tmp[3] = tmp[2] + (b[3] ? {a, 3'b0} : 15'b0);
    tmp[4] = tmp[3] + (b[4] ? {a, 4'b0} : 15'b0);
    tmp[5] = tmp[4] + (b[5] ? {a, 5'b0} : 15'b0);
    tmp[6] = tmp[5] + (b[6] ? {a, 6'b0} : 15'b0);
    tmp[7] = tmp[6] + (b[7] ? {a, 7'b0} : 15'b0);
    q      = tmp[7];
  end
endmodule
```

Create an implementation with two adders

Rewrite code to two statements per clock cycle

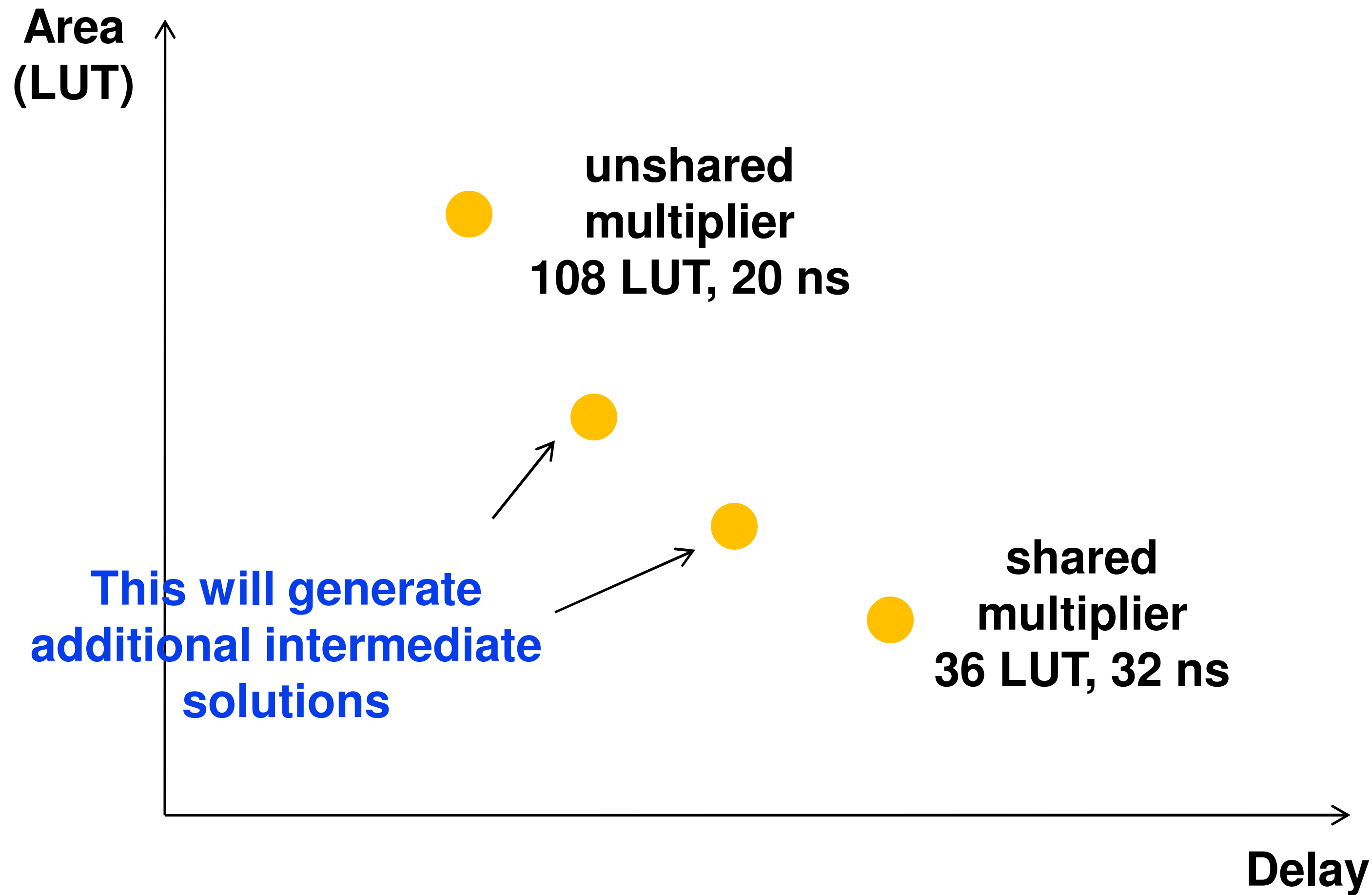
Other Solutions are possible ...

```
module syn_ex3(q, a, b);
  output [15:0] q;
  reg    [15:0] q;
  input  [7:0]  a, b;

  reg [15:0] tmp [7:0];

  always @(a or b) begin
    tmp[0] = b[0] ? a : 15'b0;
    tmp[1] = tmp[0] + (b[1] ? {a, 1'b0} : 15'b0);
    tmp[2] = tmp[1] + (b[2] ? {a, 2'b0} : 15'b0);
    tmp[3] = tmp[2] + (b[3] ? {a, 3'b0} : 15'b0);
    tmp[4] = tmp[3] + (b[4] ? {a, 4'b0} : 15'b0);
    tmp[5] = tmp[4] + (b[5] ? {a, 5'b0} : 15'b0);
    tmp[6] = tmp[5] + (b[6] ? {a, 6'b0} : 15'b0);
    tmp[7] = tmp[6] + (b[7] ? {a, 7'b0} : 15'b0);
    q      = tmp[7];
  end
endmodule
```

Create an implementation with four adders

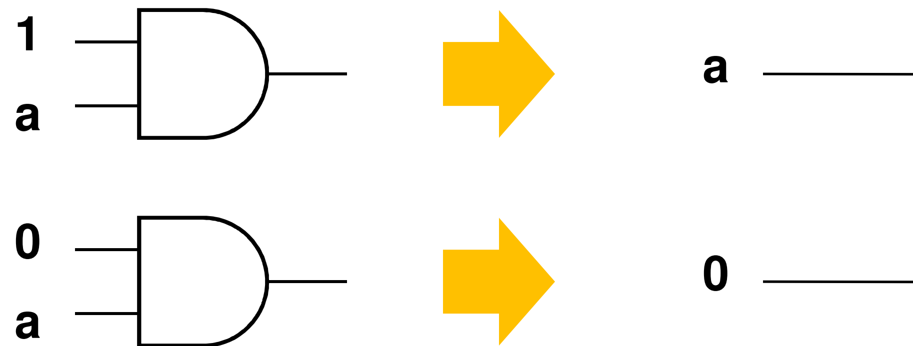
Rewrite code to four statements per clock cycle

Sharing may reduce area at expense of delay



Multiplication with a constant

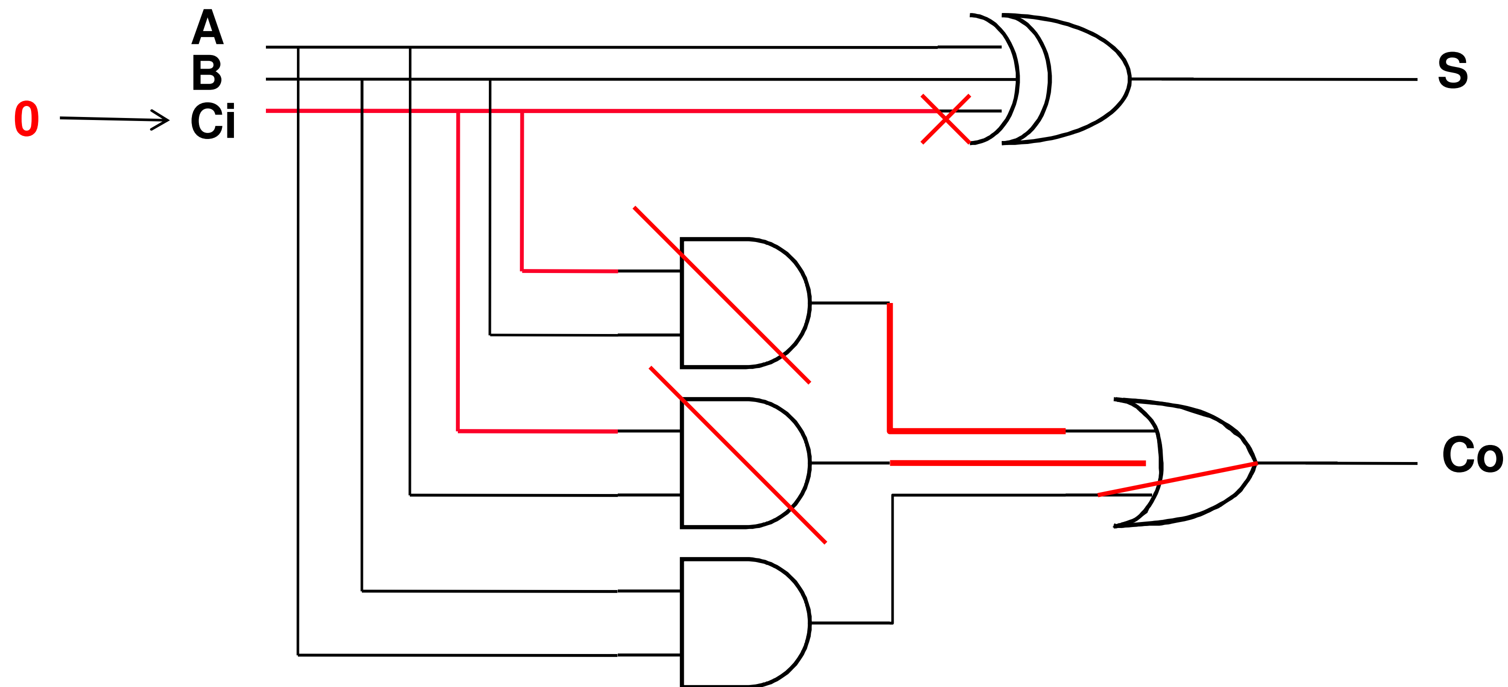
- When constants are used in a digital design, they can be propagated 'into' the implementation



- This can be used to simplify architectures considerably
- For example, a multiplier can be adapted to a more optimal design

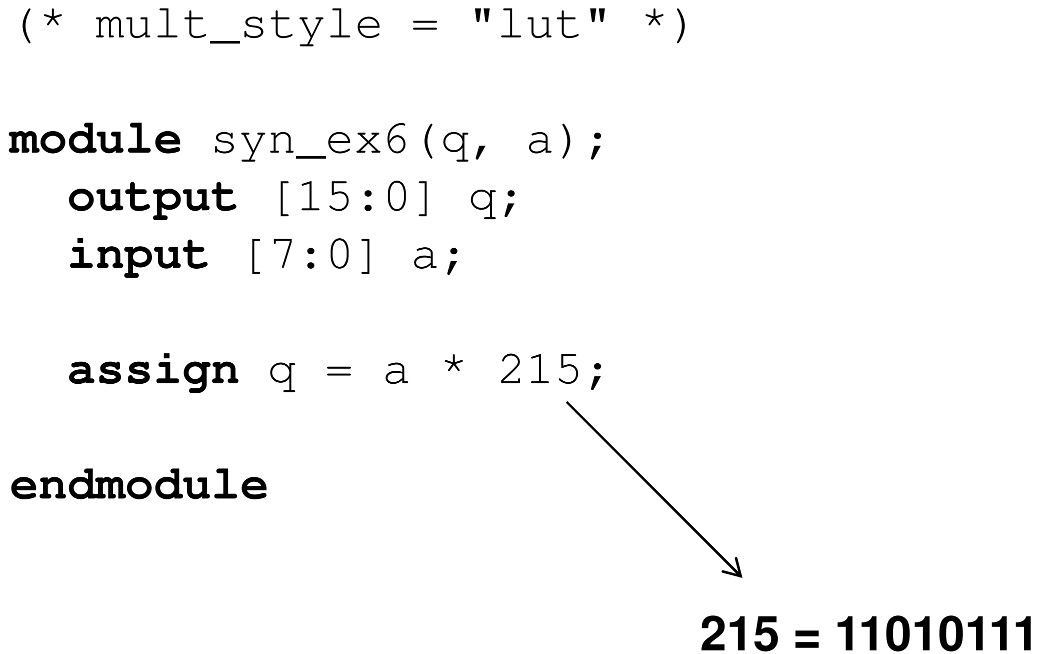
Example Full Adder -> Half Adder

- Propagate '0' into Ci



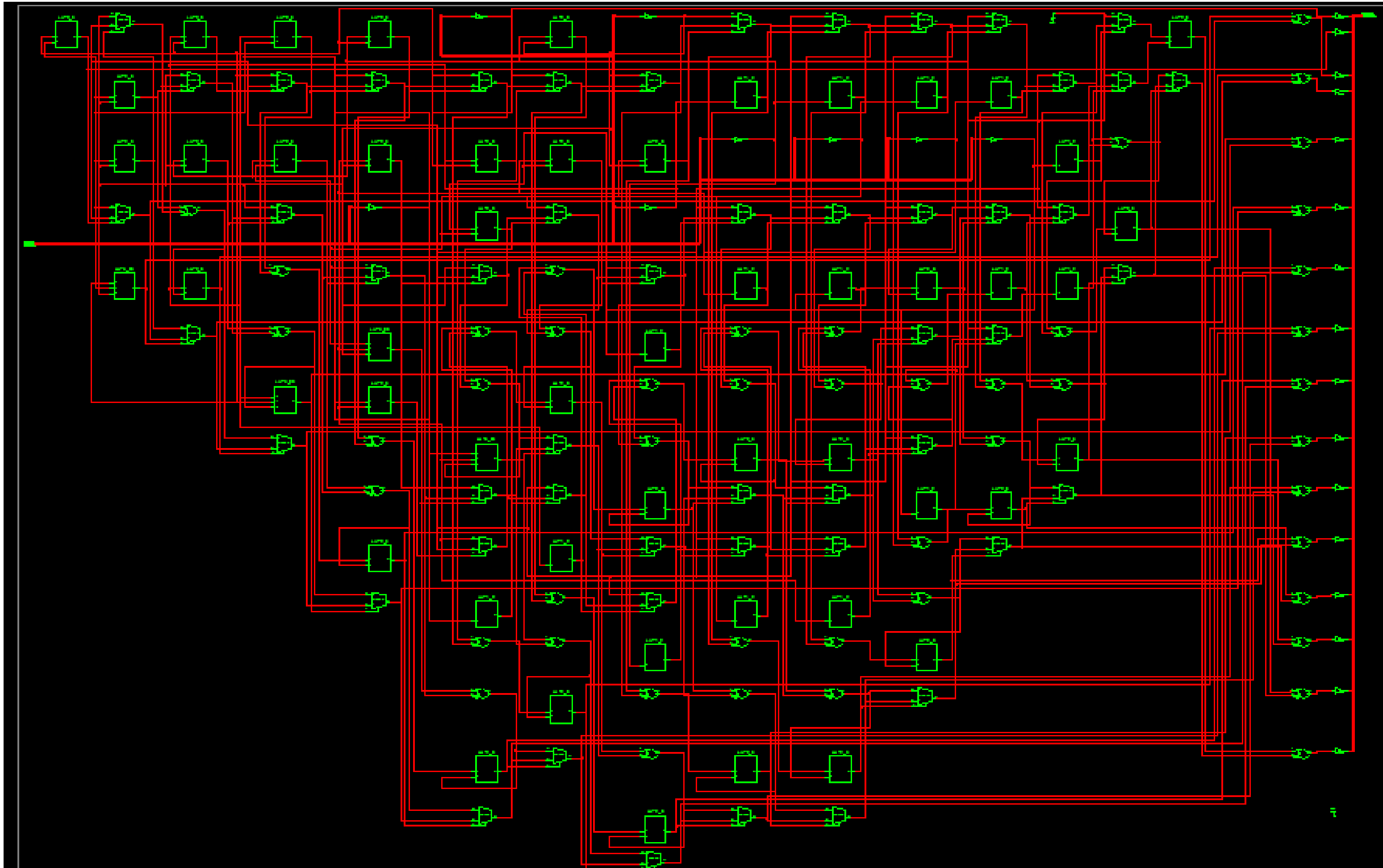
Half Adder: S = A ^ B, Co = A & B

Example: constant multiplication



Spartan3: 36 LUT, 13.5 ns

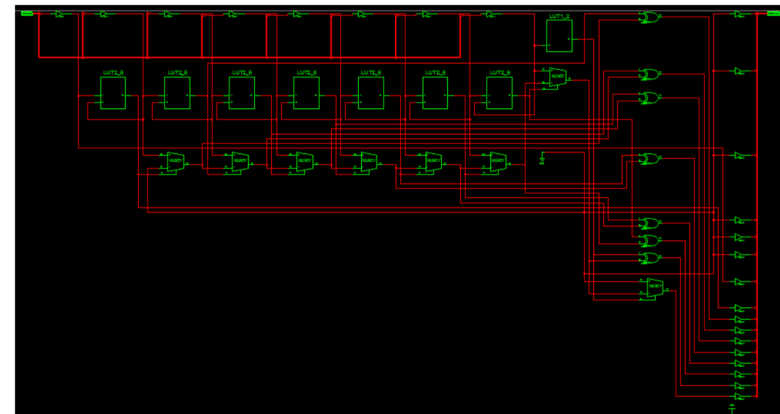
- Cfr: 70 LUT, 10ns for full 8*8 bit



Example: constant multiplication

```
(* mult_style = "lut" *)

module syn_ex6(q, a);
  output [15:0] q;
  input [7:0] a;

  assign q = a * 192;

endmodule
```

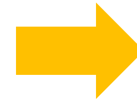
7 LUT, 10ns

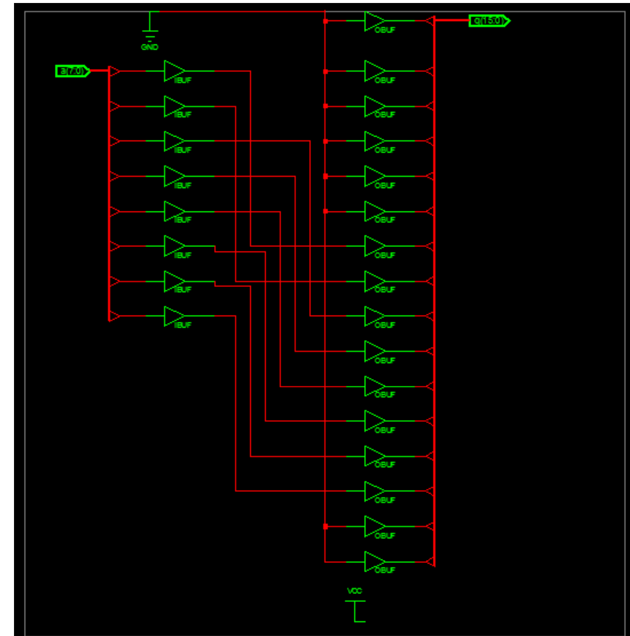


Simple constants (from a bitpatttern-perspective) generate simple multiplier hardware ..

Example: constant multiplication

```
(* mult_style = "lut" *)

module syn_ex6(q, a);
  output [15:0] q;
  input [7:0] a;

  assign q = a * 64;

endmodule
```





When constant is a power of two, multiplication becomes a hardwired shift: no LUT's needed at all!