Lecture 7-2 Design For Area Minimization

Outlines

- Analysis of area cost
- Algorithm: Complexity reduction
- Architecture: resource sharing

ANALYSIS OF AREA COSTS

Synthesis

- **Synthesis** = Translation + Logic Optimization + Technology Mapping

```
always @ (a, b)
        case ({a,b})
            2'b00:  out  =  1;
            2'b01:  out  =  1;
            2'b11:  out  =  1;
            default: out = 0;
        endcase
```

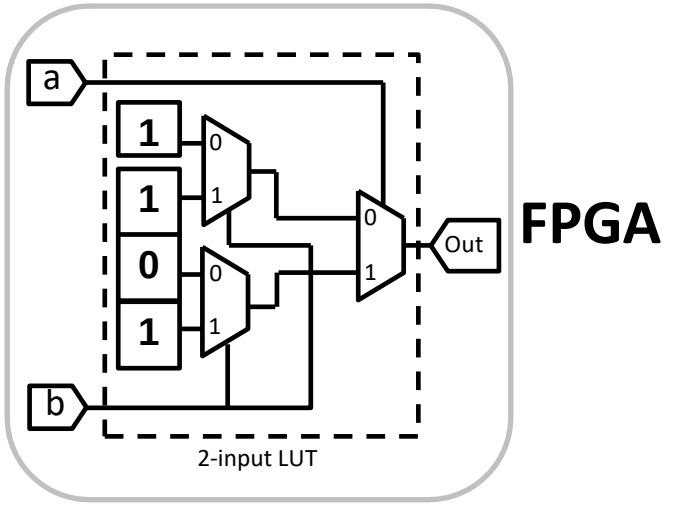
Translation

$$out = \bar{a}b + ab + \bar{a}\bar{b}$$

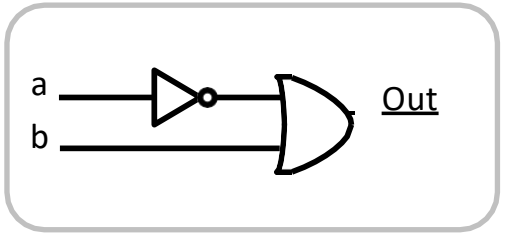
Logic Optimization

$$out = \bar{a} + b$$

Technology Mapping



FPGA



ASIC

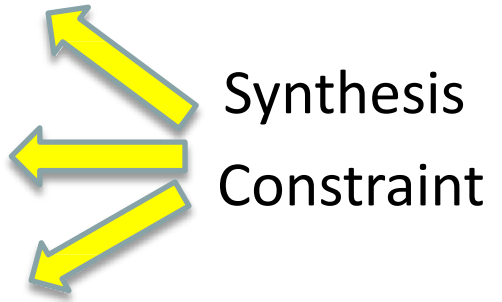
Synthesis is Constraint-Driven

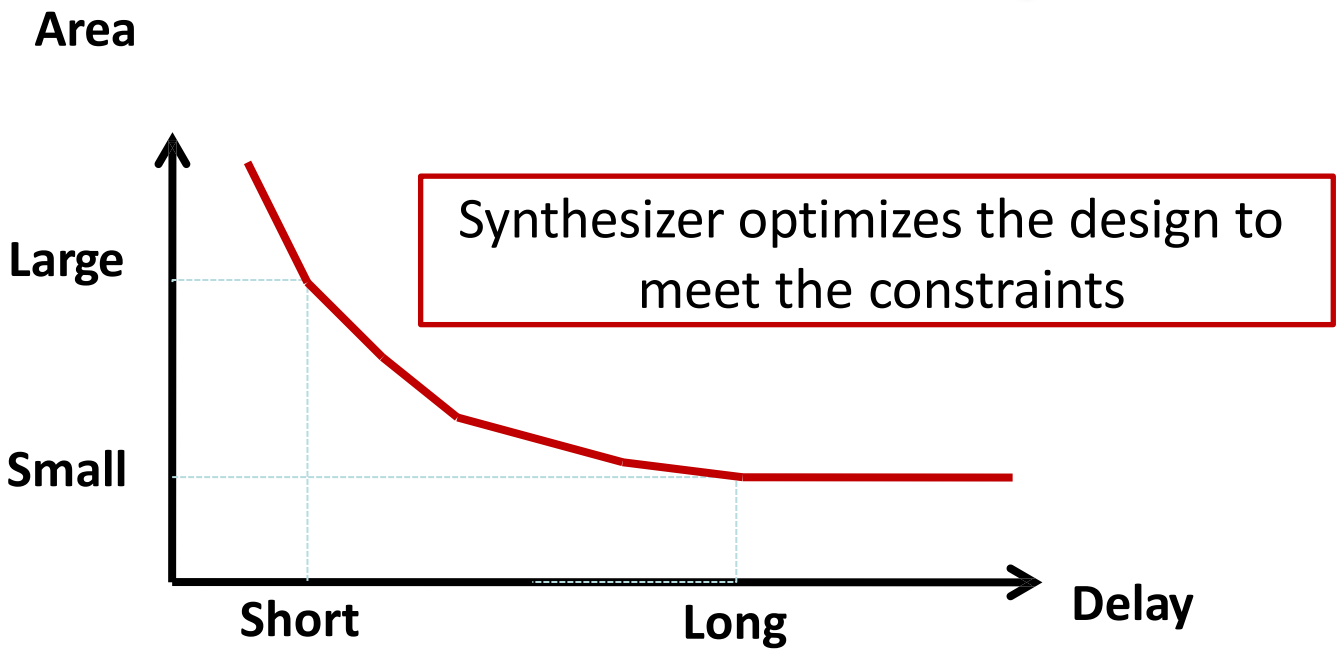
You should specify your constraints

(Delay)

(Area)

(Power Consumption)



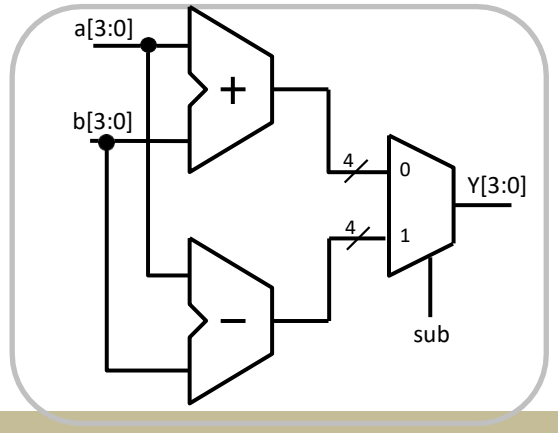


HDL for Synthesis

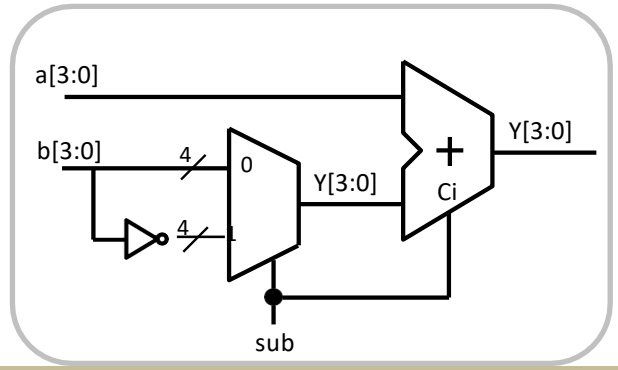
- "Bad" HDL code does not allow efficient optimization during synthesis
  - Garbage in, garbage out!
- Logic synthesizer doesn't do magic! designer has to take some responsibility in coding.

❖ Example:

```
input sub;
input [3:0] a,b;
output [3:0] y;
assign y = sub ? (a-b) : (a+b)
```




```
input sub;
input [3:0] a,b;
output [3:0] y;
wire [3:0] tmp;
assign tmp = sub ? ~b : b;
assign y = a + tmp + {3'b0,sub}
```



More efficient

HDL for Synthesis (General Guidelines)

- Think Hardware:
  - Write HDL hardware descriptions
    - Think of the topology implied by the code
  - Do not write HDL simulation models
    - No explicit delays
    - No file I/O
- Think RTL:
  - Writing in an RTL coding style means describing:
    - Register architecture
    - Circuit topology
    - Functionality between registers
  - Synthesis tool optimizes logic between registers:
    - It does not optimize the register placement

How to Read You Area Reports

```
****************************************
Report : area
Design : prime
Version: 2003.06
Date   : Sat Oct  4 11:38:08 2003
****************************************

Library(s) Used:

    XXXXX

Number of ports:                5
Number of nets:                 9
Number of cells:                5
Number of references:           4

Combinational area:          7.000000
Noncombinational area:       0.000000
Net Interconnect area:      undefined  (Wire load has zero
     net area)

Total cell area:             7.000000
Total area:                 undefined
```

Noncombinational part: DFF, SRAM, latch

Left part: area report
Right part: timing report

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : prime
Version: 2003.06
Date   : Sat Oct  4 11:38:08 2003
****************************************

Operating Conditions:
Wire Load Model Mode: enclosed

  Startpoint: in[2] (input port)
  Endpoint: isprime (output port)
  Path Group: (none)
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  prime              2K_5LM                XXXXX

  Point                                    Incr       Path
  ------------------------------------------------------------
    -
  input external delay                     0.000      0.000 r
  in[2] (in)                               0.000      0.000 r
  U4/Y (EX210)                             0.191      0.191 f
  U5/Y (BF051)                             0.116      0.307 r
  U1/Y (BF052)                             0.168      0.475 f
  isprime (out)                            0.000      0.475 f
  data arrival time                                   0.475
  ------------------------------------------------------------
    -
  (Path is unconstrained)
```

Area Estimation

- 1 DFF, 1 adder, 1 2:1 MUX:
  - 7 gate counts
- Multiplier: depends on architecture
  - NxN => N^2 * 7
- Estimate data flow part first
  - How many adder/multiplier have been used
- Control part (FSM)
  - Extra 5 to 10% area cost

Where your area goes?

- Use Synopsys Design Compiler to analyze this
  - Not covered in this course
- Code review
  - Sequential part: Count DFF numbers generated in each always_ff
    - Unique output variables in the Verilog commands
    - Avoid unnecessary pipeline or DFF storages
  - Data path part
    - Count adder/multiplier and other and/or logic
  - Controller part
    - Simplify your FSM
- Rule of thumb
  - Controller: take 5 to 10% of area
  - Data path takes 40% to 70%, memory or buffers takes the rest

Complexity Reduction

- Algorithm optimization
  - Application dependent
  - Something more like used in C program
  - E.g. assign y = a * b;
  - If b = 4'h2;
    - assign y = a << 1;

AREA OPTIMIZATION

Concept

- Folding (or rolling up the pipeline)
  - Time sharing the same logic to save area
- Algorithm change

Area-related Techniques:

- Area is the second primary factors of a digital design
  - A topology that targets area is one that reuses the logic resources to the greatest extent possible, often at the expense of throughput (speed).
  - This requires a recursive data flow, where the output of one stage is fed back to the input for similar processing.

Area-related Techniques: Rolling Up the Pipeline

- Opposite to the unrolling the loop to increase throughput
  - Unrolling the loop achieved by adding more registers to hold intermediate values, i.e., more area
  - Thus to reduce the area the reversed action should be done (i.e., Sharing)

- Resource Sharing is used where there are functional blocks that can be used in other areas of the design or even in different modules
  - Sharing logic resources requires special control circuitry to determine which elements are input to the particular structure

Rolling Up the Pipeline

- Calculation of P = A*B
  - A: a normal integer with the fixed point just to the right of the LSB (8 bits)
  - B: a fractional number with a fixed point just to the left of the MSB (8 bits)
  - P: the product, which requires only 8-bits
- One implementation alternative:
  - Critical path: one multiplier (complex itself)
  - One product every clock cycle (high throughput)

```
module mult8(
output [7:0] P,
input [7:0] A,
input [7:0] B,
input clk);
reg [15:0] prod16;
assign P= prod16[15:8];
always @(posedge clk)
prod16 <= A * B;
endmodule
```

Rolling Up the Pipeline : Resource Sharing

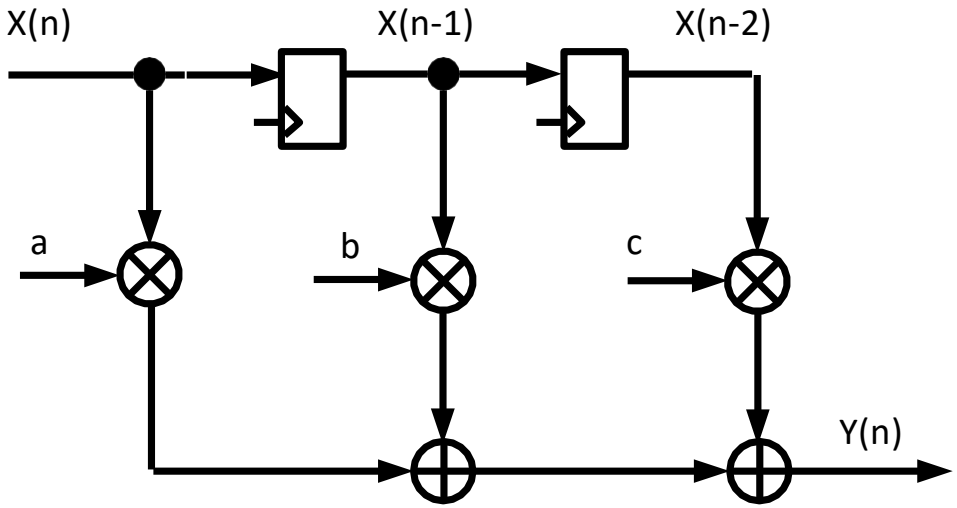
- Rolling Up the Pipeline:
  - Using series of shift-and-add operations
  - Smaller critical path
  - Less area due to the simple operations and sharing
  - One product every 8 clock cycles! (low throughput)

```
module mult8(output done,output reg [7:0] product,
input [7:0] A, input [7:0] B, input clk, input start);
reg [4:0] multcounter; // number of shift/adds
reg [7:0] shiftB; // shift register for B
reg [7:0] shiftA; // shift register for A
wire adden; // enable addition
assign adden = shiftB[7] & !done;
assign done = multcounter[3];
always @(posedge clk) begin
if(start) multcounter <= 0;
else if(!done) multcounter <= multcounter + 1;
// shift register for B
if(start) shiftB <= B;
else shiftB[7:0] <= {shiftB[6:0], 1'b0};
```

```
// shift register for A
if(start) shiftA <= A;
else shiftA[7:0] <= {shiftA[7], shiftA[7:1]};
// calculate multiplication
if(start) product <= 0;
else if(adden) product <= product + shiftA;
end
endmodule
```

Resource Sharing: Area Reduction Technique

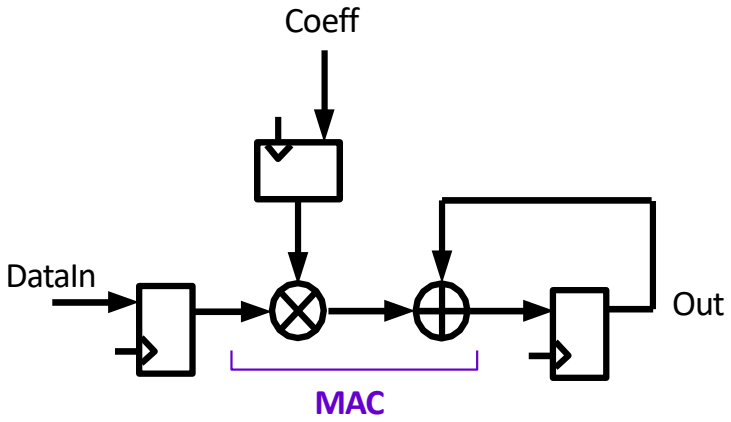
- Back to FIR Filter:
  - Three multiplications, two adders, two registers



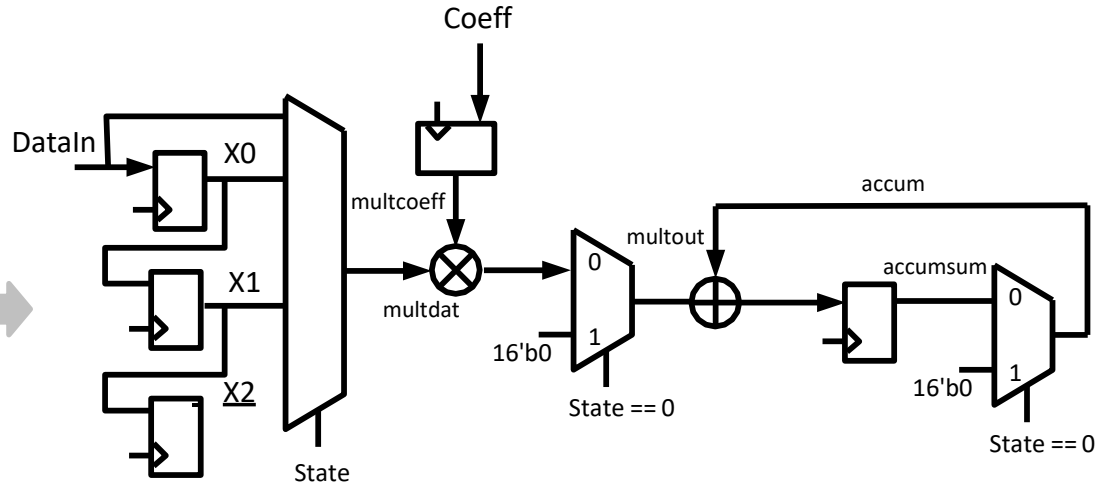
$Y(n) = ax(n) + bx(n-1) + cx(n-2)$

Resource Sharing: Area Reduction Technique

- Sharing the Multiply-Accumulate (MAC) to reduce area:
  - One multiplication, one adder, one register
  - Requires some control logic to determine which input is inserted (FSM)



DataIn=X[0], 0,0,0, X[1], 0,0,0,X[2], ...
Coeff= a, b, c, 0,a, b, c, 0, ...



Resource Sharing: Area Reduction Technique

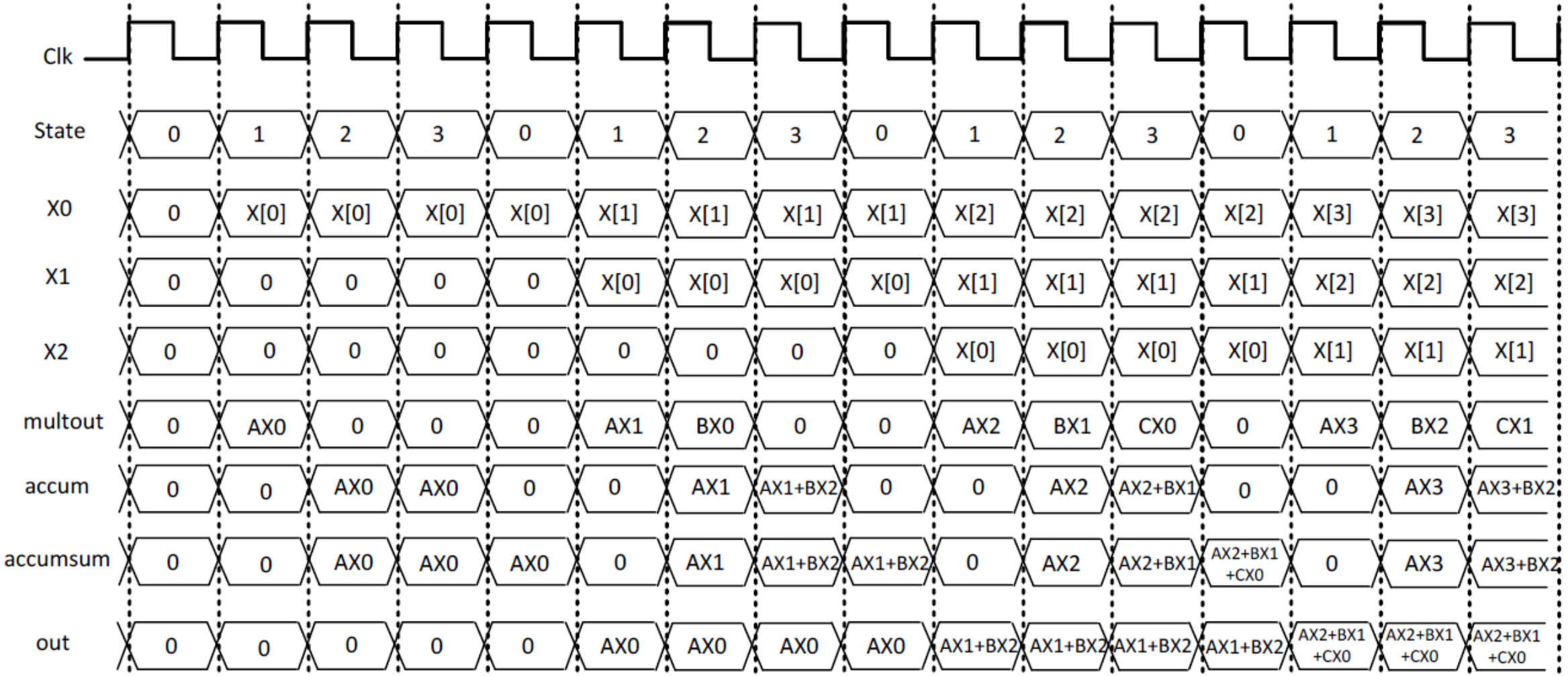
```
module sharing(
output reg [15:0] Out,
input clk,
input [7:0] datain, // X[0]
input [7:0] coeffA, coeffB, coeffC); // coeffs for low pass filter
// define input/output samples
reg [7:0] X0, X1, X2;
reg [2:0] state; // holds state for sequencing through mults
wire [15:0] accum; // accumulates multiplier products
reg [15:0] accumsum;
wire [15:0] multout; // multiplier product
reg [7:0] multdat;
reg [7:0] multcoeff;

assign multout =(state==0)?16'b0:multcoeff * multdat;
// clearing and loading accumulator
assign accum = (state==0)?16'b0:accumsum;

always @(posedge clk)
accumsum <= accum + multout;
```

```
always @ (posedge clk) begin
case(state)
      0: begin // load new data
                X0 <= datain; X1 <= X0; X2 <= X1;
                multdat <= datain; multcoeff <= coeffA;
                state <= 1;
                 Out <= accumsum;
      end
      1: begin // A*X[0] is done, load B*X[1]
                multdat <= X1; multcoeff <= coeffB;
                state <= 2;
       end
       2: begin // B*X[1] is done, load C*X[2]
                multdat <= X2; multcoeff <= coeffC;
                state <= 3;
        end
        3: begin // C*X[2] is done, load output
                state <= 0;
         end
         default
                state <= 0;
        endcase
end
endmodule
```

Resource Sharing: Area Reduction Technique

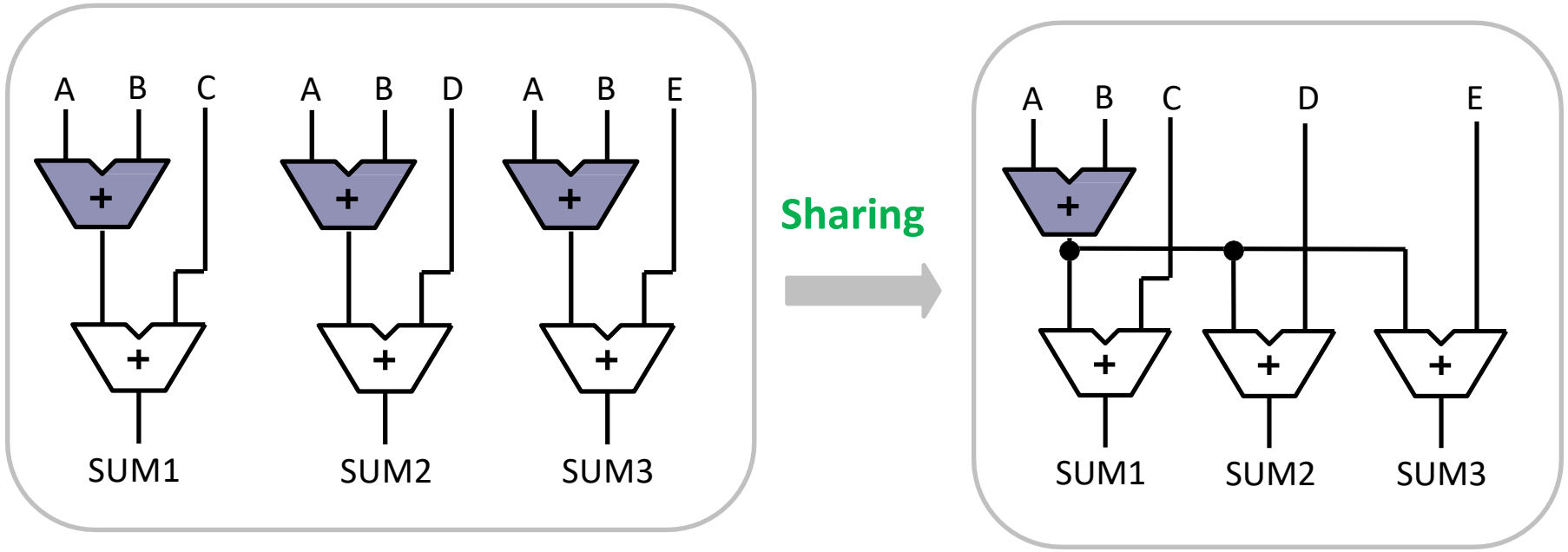


Resource Sharing: Area Reduction Technique

- HDL coding style can force a specific topology to be synthesized

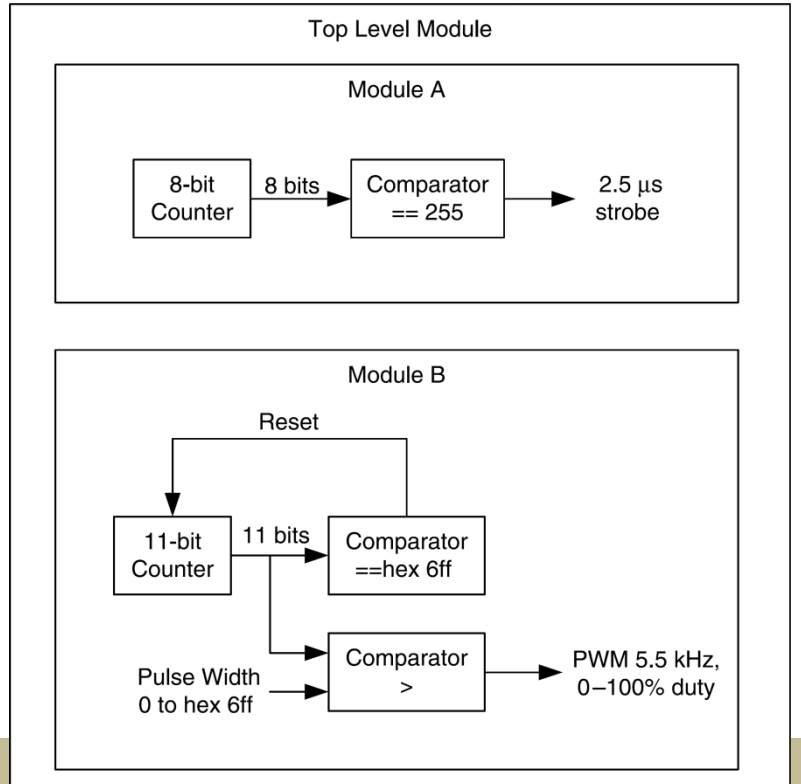
```
SUM1 <= A+B+C;
SUM2 <= A+B+D;
SUM3 <= A+B+E;
```

```
assign tmp = A+B;
SUM1 <= tmp +C;
SUM2 <= tmp +D;
SUM3 <= tmp +E;
```



Resource Sharing: Area Reduction Technique

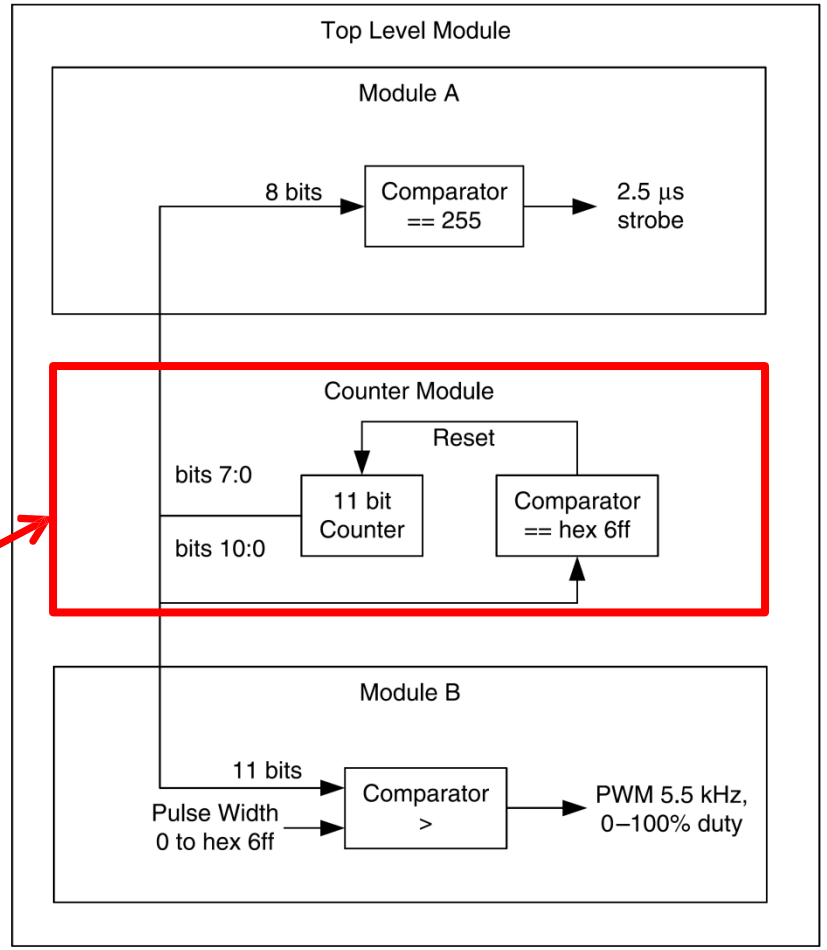
- Let's assume we have two counters controlling different design sections
  - CounterA, a free running 8-bit counter
  - CounterB, an 11-bit counter, counting from 0 to 1666 and resets to zero



Resource Sharing: Area Reduction Technique

- Resource-shared version of these two counters

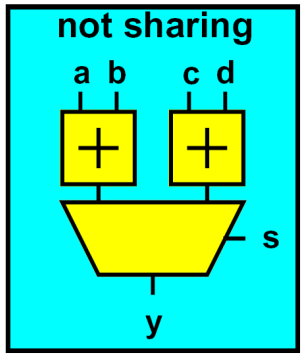
For compact designs where area is the primary requirement, search for resources that have similar counterparts in other modules that can be brought to a global point in the hierarchy and shared between multiple functional areas.



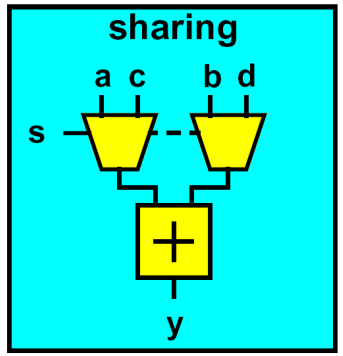
Shared Part

Architectural Techniques : Resource Sharing

More than one RTL statement can utilize an expensive computational resource.




```
always @(a or b or c
         or d or s)
  begin
    if (s)
      y = a + b;
    else
      y = c + d;
  end
```




```
always @(a or b or c
         or d or s)
  begin: newscope
    reg tmp1, tmp2;
    tmp1 = s ? a : c;
    tmp2 = s ? b : d;
    y = tmp1 + tmp2;
  end
```

Some synthesis tools automatically share resources.

What if your synthesis tool does not automatically share resources?

You can write your RTL code to do its own resource sharing!