ASIC/FPGA Chip Design

Verilog for Synthesis

Mahdi Shabany

Department of Electrical Engineering
Sharif University of technology

ASIC/FPGA Design Flow

Specifications
RTL Coding
Test Bench
Simulation Pass?
NO
Yes
Standard Cells
Synthesis
Timing Constraints
Pre-Layout Timing Alanysis Pass?
NO
Yes
APR
Back Annotation
Post-Layout Timing Alanysis Pass?
NO
Yes
Logic verification
Tapeout

1. HDL Coding **2. Simulation** **3. Synthesis** **4. Placement & routing** **5. Timing Analysis & Verification**

Front-End **Back-End**

❑ In this course we learn all the above steps in detail for

- ASIC Platform
- FPGA Platform

© M. Shabany, ASIC/FPGA Chip Design

Synthesis

Specifications
RTL Coding
Test Bench
Simulation Pass?
NO
Yes
Standard Cells
Synthesis
Timing Constraints
Pre-Layout Timing Alanysis Pass?
NO
Yes
APR
Back Annotation
Post-Layout Timing Alanysis Pass?
NO
Yes
Logic verification
Tapeout

- Synthesis tool:
  - Analyzes a piece of Verilog code and converts it into optimized logic gates
  - This conversion is done according to the **"language semantics"**
  - ➡ We have to learn these language semantics, i.e., Verilog code.

Synthesis

- Why using synthesis tools?

  - It is an important tool to improve designers' productivity to meet today's design complexity.

  - If a designer can design 150 gates a day, it will take 6666 man's day to design a 1-million gate design, or almost 2 years for 10 designers! This is assuming a linear grow of complexity when design get bigger.

Synthesis in Different Levels

- Synthesis can be done in different levels:
  - High-level Synthesis
    - To convert an algorithm-level description to an RTL code
  - RTL Synthesis
    - To convert an RTL code to a gate-level netlist
  - Logic Synthesis
    - To convert the gate-level description to a specific logic library

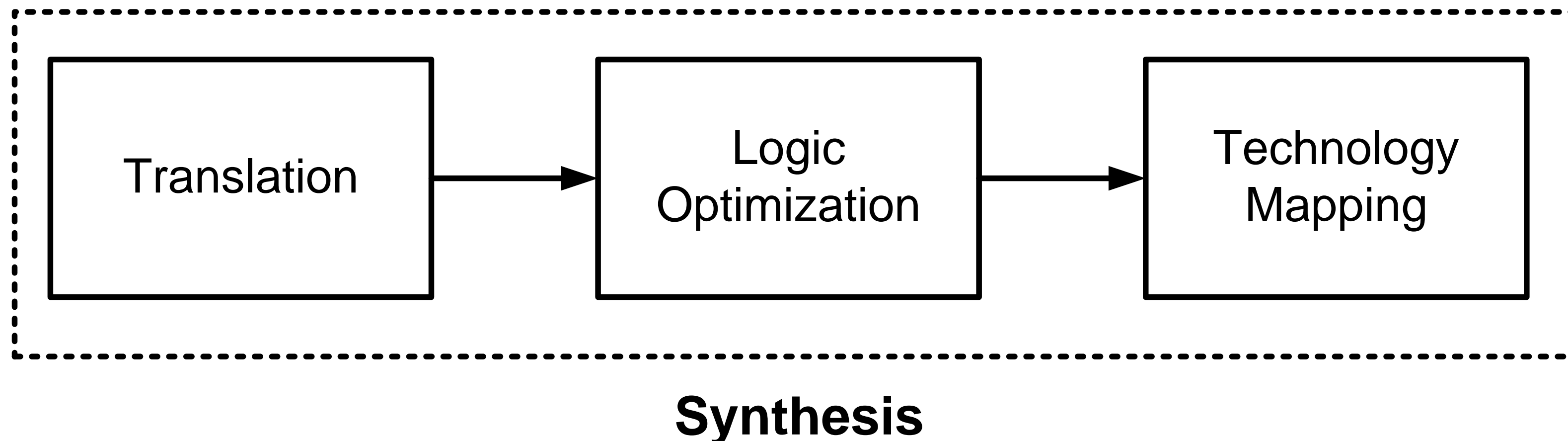
Synthesis

- Synthesis tool: (RTL & Logic Synthesis)
  - **Input:**
    - HDL Code
    - "Technology library" file → Standard cells (known by transistor size, 90nm)
      - Basic gates (AND, OR, NOR, …)
      - Macro cells (Adders, Muxes, Memory, Flip-flops, …)
    - Constraint file (Timing, area, power, loading requirement, optimization Alg.)
  - **Output:**
    - A gate-level "Netlist" of the design
    - Timing files (.sdf)

This process is done using various optimization algorithms

Synthesis

- **Synthesis** = Translation + Logic Optimization + Technology Mapping
  - **Translation:** going from RTL to Boolean function
  - **Logic Optimization :** Optimizing and minimizing Boolean function
  - **Technology Mapping (TM):** Map the Boolean function to the target library



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
```

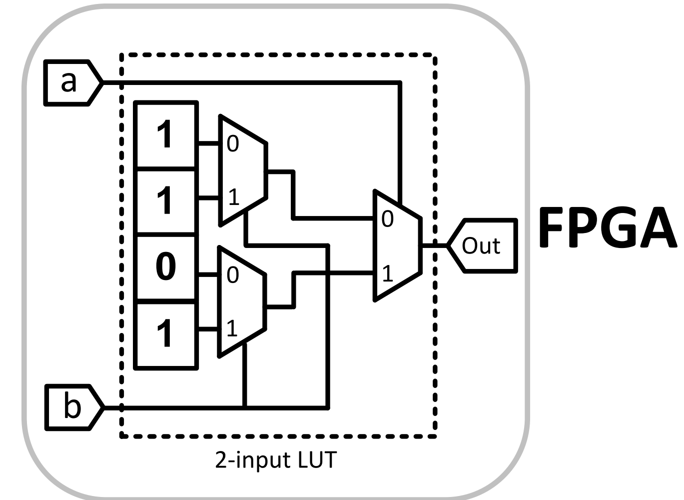
Translation

$$out = \overline{a}b + ab + \overline{a}\,\overline{b}$$

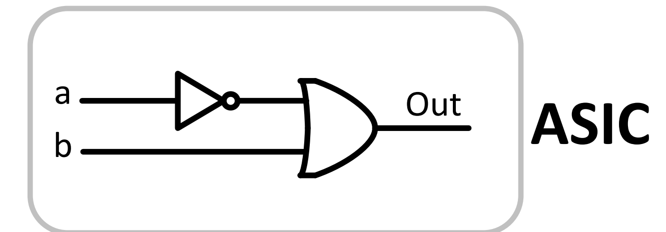
Logic Optimization

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

Technology Mapping



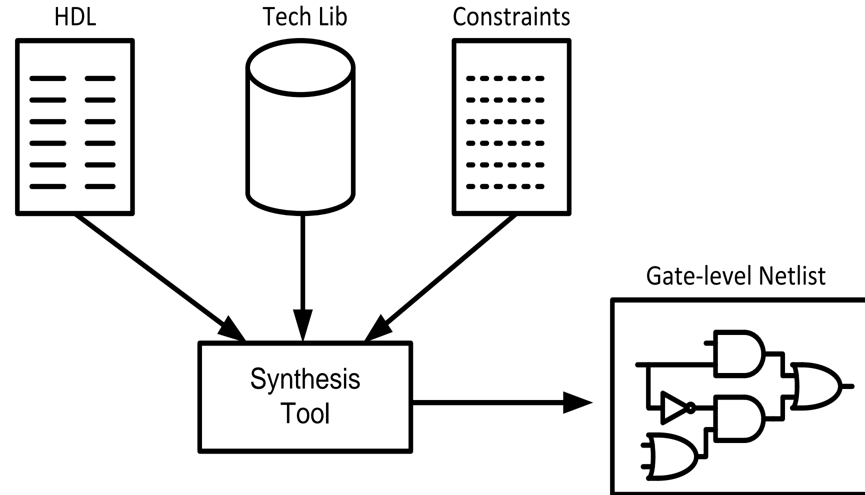
FPGA



ASIC

Synthesis Tools

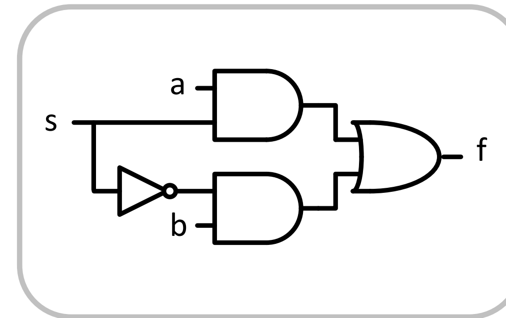
- Synthesis tool:

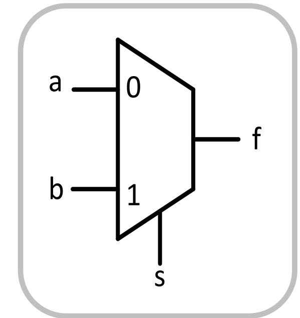


❖ **Example:** A 2-to-1 Multiplexer (2x1-MUX)

```
If (s==0)
   f = a;
else
   f = b;
```

Synthesis Tool





Verilog code
(has to comply with certain structures)

Synthesized gate-level

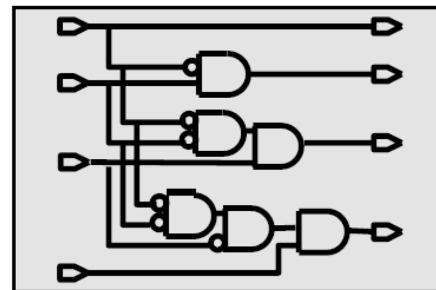
Schematic

Synthesis

```
residue = 16'h0000;
if (high_bits == 2'b10)
   residue = state_table[index];
else
   state_table[index] = 16'h0000;
```

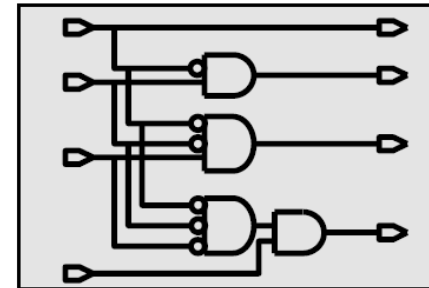
HDL Source

Translate (read)



Generic Boolean (GTECH)

Optimize + Map (compile)



Target Technology

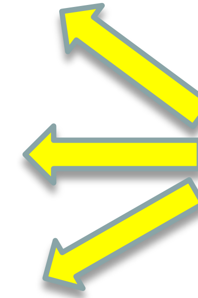
Synthesis is Constraint-Driven

You should specify your constraints

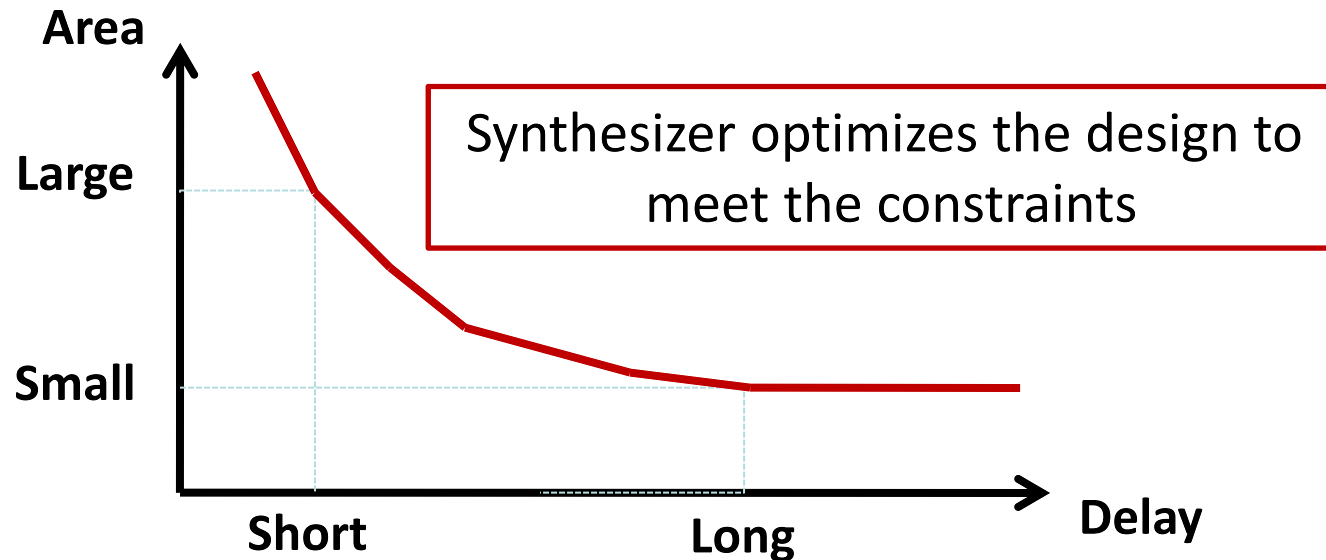
(Delay) سرعت

(Area) مساحت

(Power Consumption) توان مصرفی



Synthesis Constraint



Synthesizer optimizes the design to meet the constraints

Synthesis is Constraint-Driven

- Synthesis process takes some time

- Synthesis time is a function of the target critical path
  - Clock period: 10nsec → synthesis time: 10 minutes
  - Clock period: 5 nsec → synthesis time: 40 minutes

Synthesis is Constraint-Driven

- The designer guides the synthesis tool by providing **design constraints:**
  - Timing requirements (max. expected clock frequency)
  - Area requirements
  - Maximum power consumption
- The synthesis tool uses this information and tries to generate the smallest possible design that will satisfy the timing requirements
- Without any constraints specified, the synthesis tool will generate a non-optimal netlist, which might not satisfy the designer's requirements

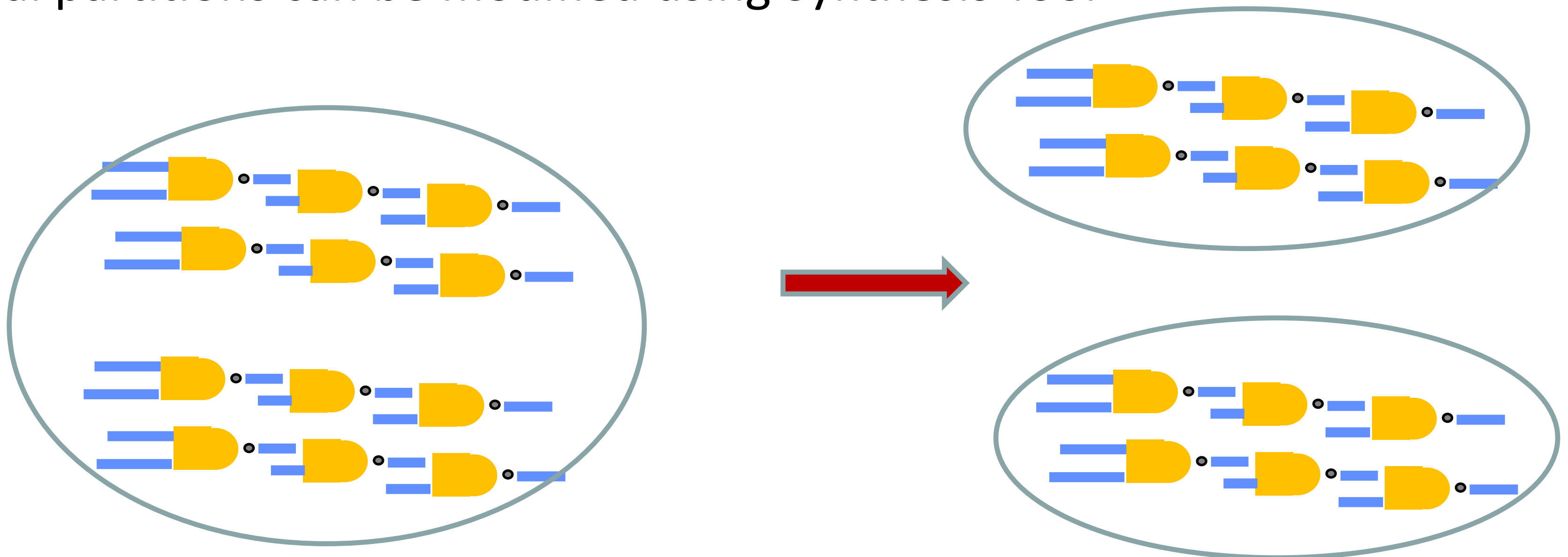
Synthesis Tools

- Commercial Synthesis Tools:

Vendor Name	Product Name	Platform
Altera	Quartus II	FPGA
Xilinx	ISE	FPGA
Mentor Graphics	Modelsim, Precision	FPGA/ASIC
Synopsys	Design Compiler	ASIC
Synplicity	Synplify	ASIC
Cadence	Ambit	ASIC

Divide and Conquer for Optimal Synthesis

- To achieve the best synthesis result, the design is better to be partitioned into smaller parts.
- **Partitioning:** the process of dividing complex designs into smaller parts
- Ideally, all partitions would be planned prior to writing any HDL:
  - Initial partitions are defined by the HDL
  - Initial partitions can be modified using Synthesis Tool



Partitioning

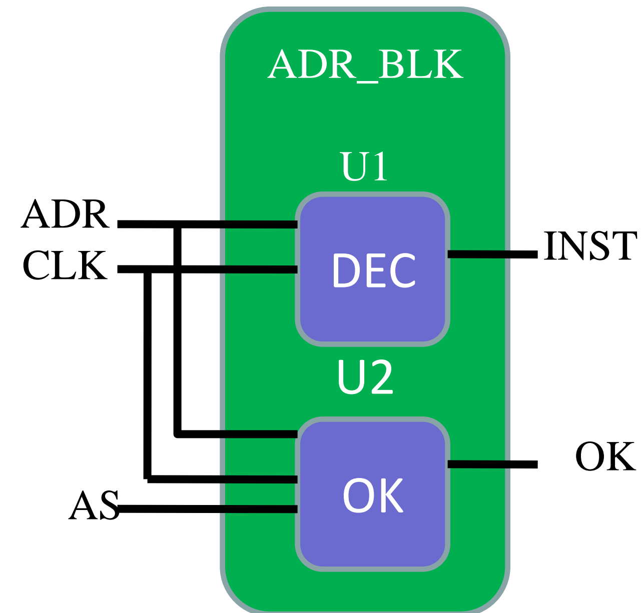
- Partition a design into different modules based on the functionality
  - Pros:
    - Separation of the cores that have different functionality
    - More manageability of smaller modules
    - Easier managements of a design implementation by a team
    - Focus to write optimized HDL code for each module
    - Possibility of reusing smaller IPs/Block in other designs
  - Cons:
    - Routing congestion or increased die size due to more signaling b/w modules

- Not too small not too large modules (~10Kgates)

Partitioning in Verilog

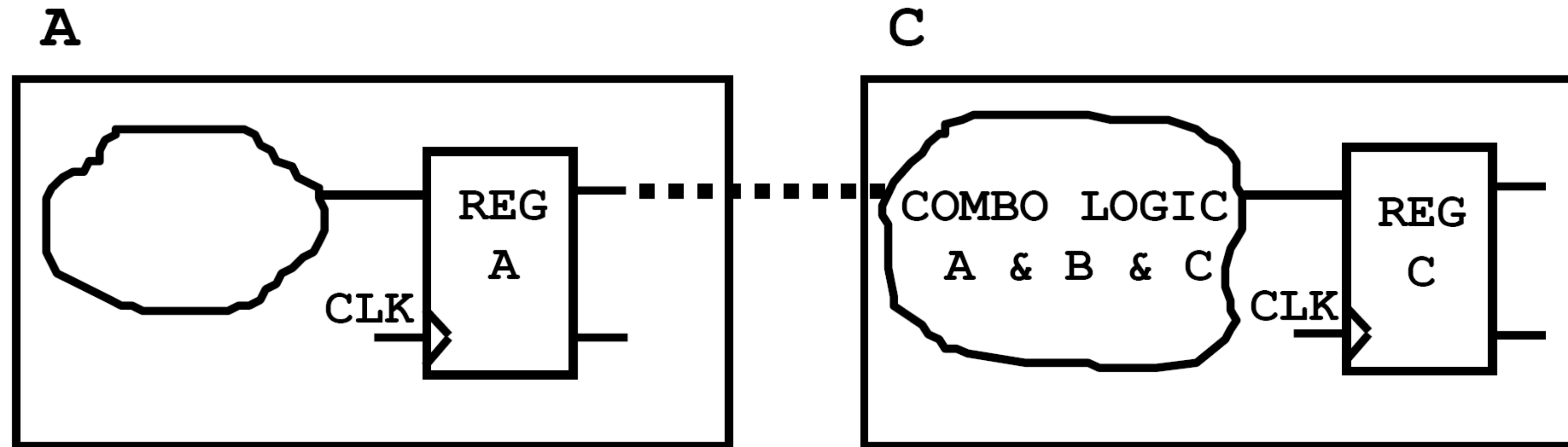
- module statement defines hierarchical blocks (partitions):
  - Instantiation of an entity or module creates a new level of hierarchy
- Inference of Arithmetic Circuits (+, -, *) can create a new level of hierarchy
- Always statements do not create hierarchy

```
module ADR_BLK (….
DEC U1 (ADR, CLK, INST);
OK  U2 (ADR, CLK, AS, OK);
endmodule;
```



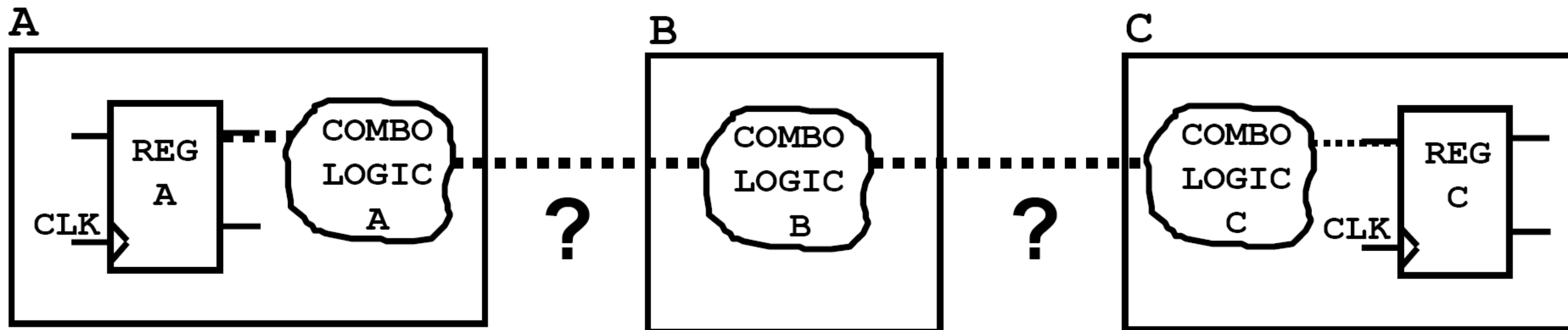
Good Partitioning (Partition at Register Boundaries)

- Try to design so the hierarchy boundaries follow register outputs.
- Related combinational logics in the middle are merged into the same block
  - Combinational optimization techniques can still be fully exploited

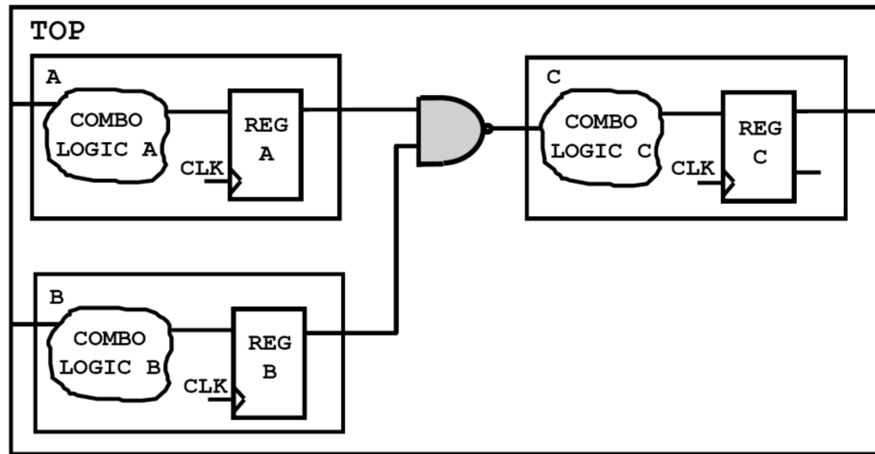


Poor Partitioning (Partition at Combinational Logic)

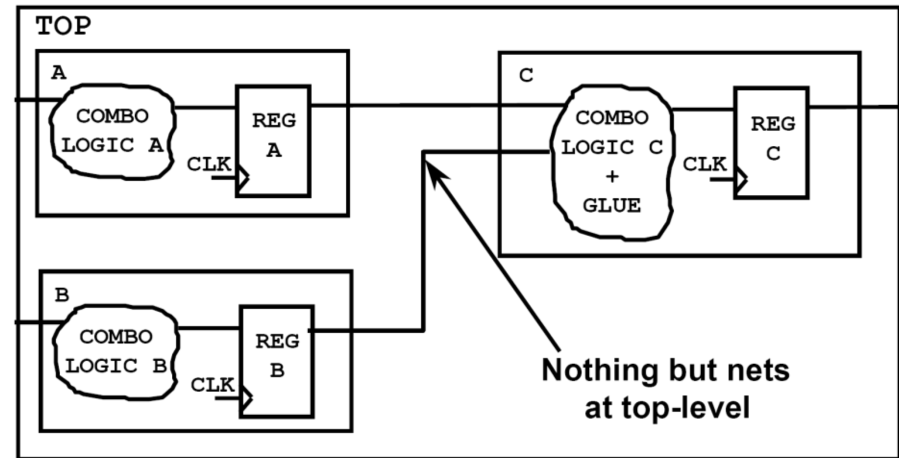
- Try not to break the Comb. Logics into several hierarchies
- Synthesis tool must preserve port definitions
- Logic optimization does not cross block boundaries
  - Adjacent blocks of combinational logic cannot be merged
- Path from REG A to REG C may be larger and slower than necessary!



Good Partitioning (Avoid Glue Logic)



Poor Partitioning



Good Partitioning

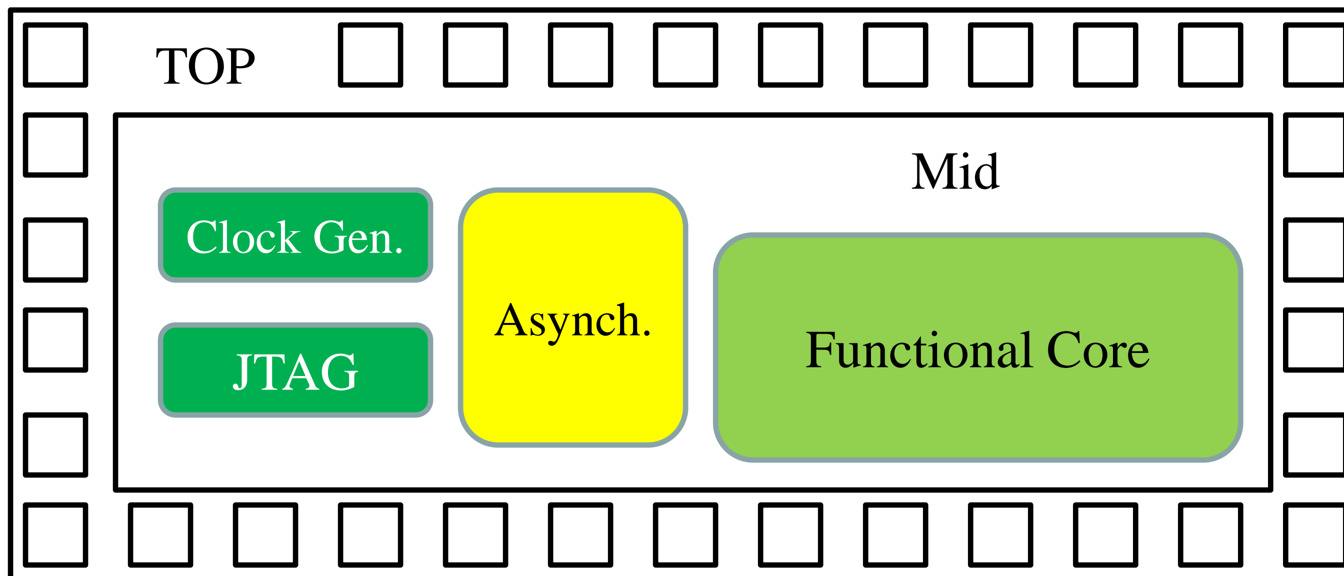
- The NAND gate at the top-level serves only to "glue" the instantiated cells.
- Optimization is limited because the glue logic cannot be "absorbed"
- Additional compile needed at top-level

- The glue logic can now be optimized with other logic
- Top-level design is only a structural netlist, it does not need to be compiled

Good Partitioning

در جزءبندی، Core Logic، کلاکها، پدها، JTAG را از هم جدا کنید.

سلسله مراتب طرح:
- Top_level
- Mid_level
- Functional Core

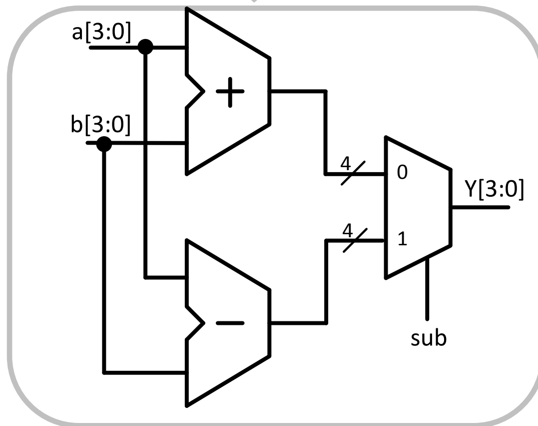


HDL for Synthesis

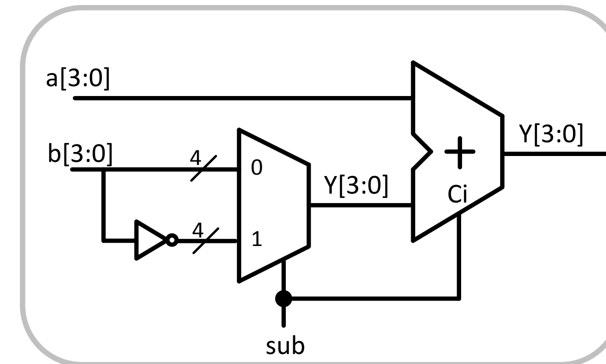
- "Bad" HDL code does not allow efficient optimization during synthesis
  - Garbage in, garbage out!
- Logic synthesizer doesn't do magic! designer has to take some responsibility in coding.

❖ 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}
```



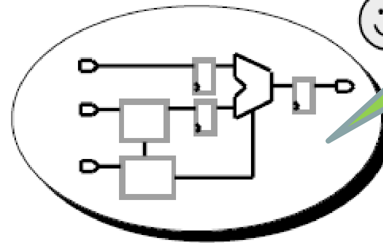
More efficient

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

HDL for Synthesis (General Guidelines)

Yes!



برای نوشتن یک برنامه کارامد لازم است توپولوژی مناسبی برای سخت افزاری که قرار است پیاده سازی شود، ارائه کنیم

```
after 20 ns and
    2 clock cycles
    OUTPUT <= IN1 + RAM1;
    wait 20 ns;
    ...
```

No!

در نوشتن کد سخت فزار خود از مدلهایی که در شبیه سازی از آنها استفاده می شود مانند اعمال تاخیر سیگنالها بپرهیزید.

Synthesizable Constructs

- Not all Verilog constructs are synthesizable because:
  - Does not make sense in hardware (e.g. $display, initial block)
  - Not possible to achieve (e.g. delay control like #10)
  - Not support by design flow (e.g. use of tran in P&R)
  - Too difficult or too abstract for the synthesis software (e.g. A / B)

Synthesizable Constructs

- Ports (input, output, inout)
- Parameter
- module
- wire, reg, tri
- function, task
- always, if, else, case
- assign
- for, while

Non-Synthesizable Constructs

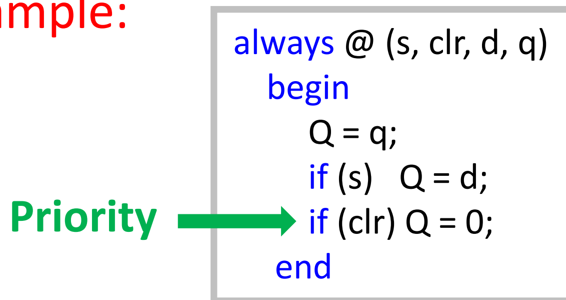
Non-Synthesizable Constructs

- Initial (used only in testbenches)
- real (real type data type)
- time (time data type)
- assign for reg data types
- comparison to X and Z are ignored (e.g., a == 1’bz)
- delays “#” are ignored by synthesis tools as if it is not there

HDL for Synthesis (Priority logic)

- The order in which assignments are written in an always block may affect the logic that is synthesized.

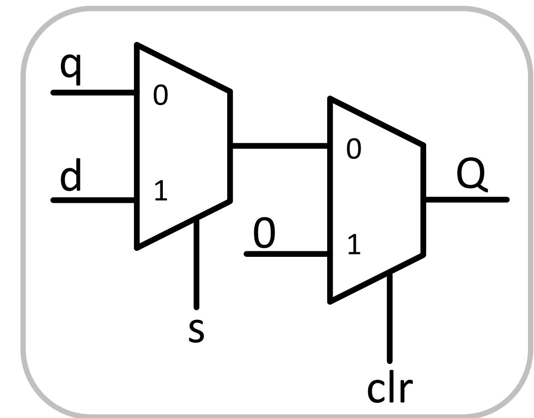
❖ Example:



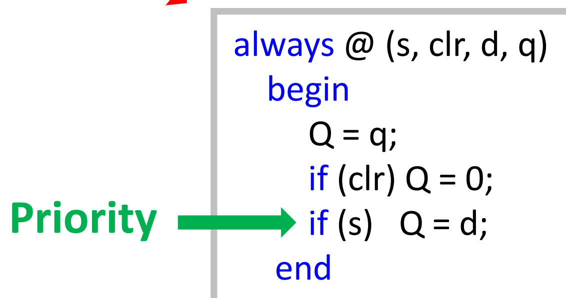
```
always @ (s, clr, d, q)
   begin
      Q = q;
      if (s)   Q = d;
      if (clr) Q = 0;
   end
```

Priority → if (clr) Q = 0;

s	clr	Q
0	0	q
0	1	0
1	0	d
1	1	0



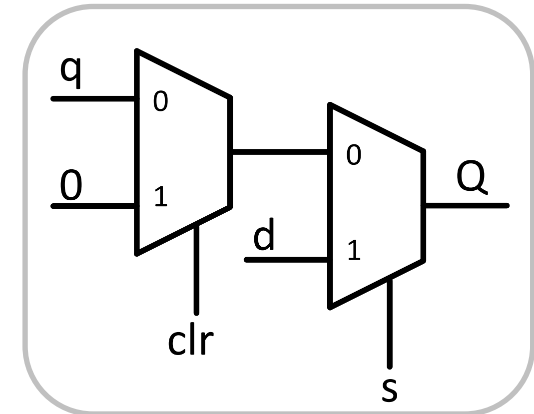
Different



```
always @ (s, clr, d, q)
   begin
      Q = q;
      if (clr) Q = 0;
      if (s)   Q = d;
   end
```

Priority → if (s) Q = d;

s	clr	Q
0	0	q
0	1	0
1	0	d
1	1	d



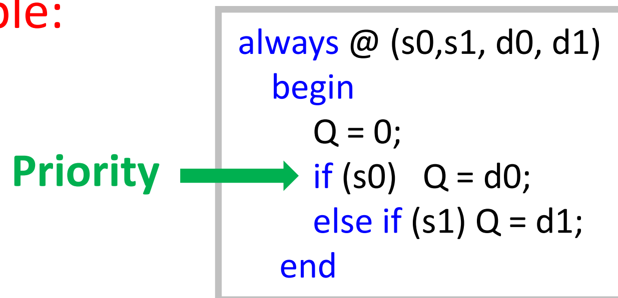
Non of the above infer latch, why?

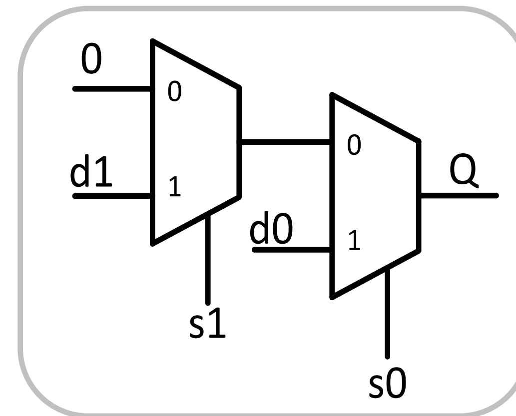
HDL for Synthesis (Priority logic)

- The order in which assignments are written in an always block may affect the logic that is synthesized.

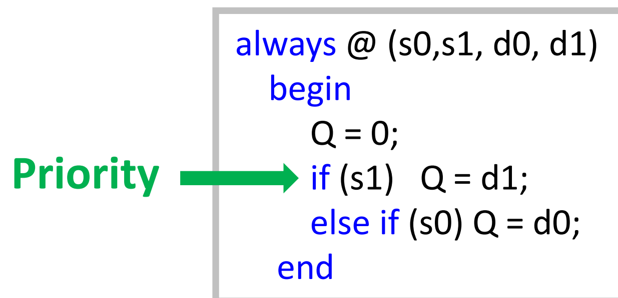
❖ Example:

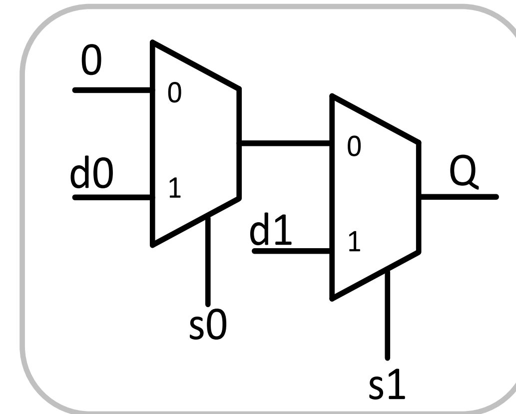
Priority → **Different**


```
always @ (s0,s1, d0, d1)
   begin
      Q = 0;
      if (s0)   Q = d0;
      else if (s1) Q = d1;
   end
```



Priority →


```
always @ (s0,s1, d0, d1)
   begin
      Q = 0;
      if (s1)   Q = d1;
      else if (s0) Q = d0;
   end
```



Non of the above infer latch, why?

HDL for Synthesis (Priority logic) : Poor Coding

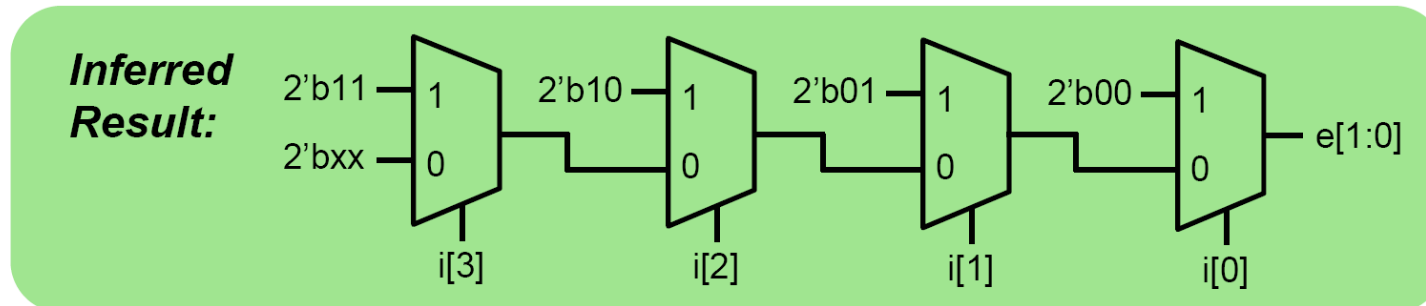
Intent: if more than one input is 1, the result is a don't-care.

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Code: if i[0] is 1, the result is 00 regardless of the other inputs. *i[0] takes the highest priority.*

1st Priority → 2nd Priority → 3rd Priority → 4th Priority

```
if (i[0]) e = 2'b00;
else if (i[1]) e = 2'b01;
else if (i[2]) e = 2'b10;
else if (i[3]) e = 2'b11;
else e = 2'bxx;
end
```



- `if-else` and `case` statements are interpreted very literally! Beware of unintended priority logic.

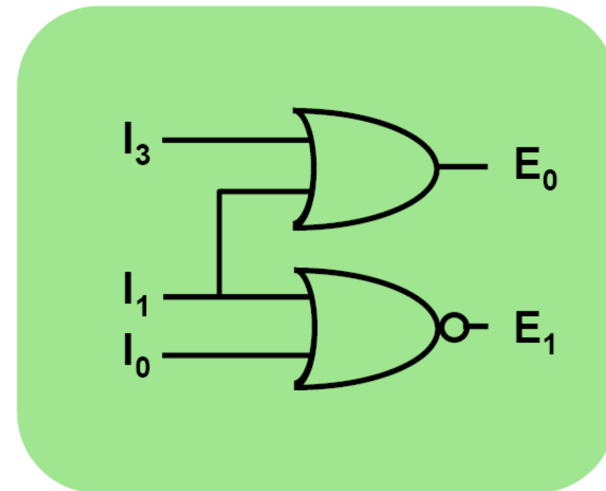
HDL for Synthesis (Priority logic) : Good Coding

- Make sure that `if-else` and `case` statements are *parallel*
  - If mutually exclusive conditions are chosen for each branch...
  - ...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module binary_encoder(i, e);
  input  [3:0]  i;
  output  [1:0]  e;
  reg e;

  always @(i)
  begin
    if  (i == 4'b0001)  e = 2'b00;
    else if  (i == 4'b0010)  e = 2'b01;
    else if  (i == 4'b0100)  e = 2'b10;
    else if  (i == 4'b1000)  e = 2'b11;
    else e = 2'bxx;
  end
endmodule
```

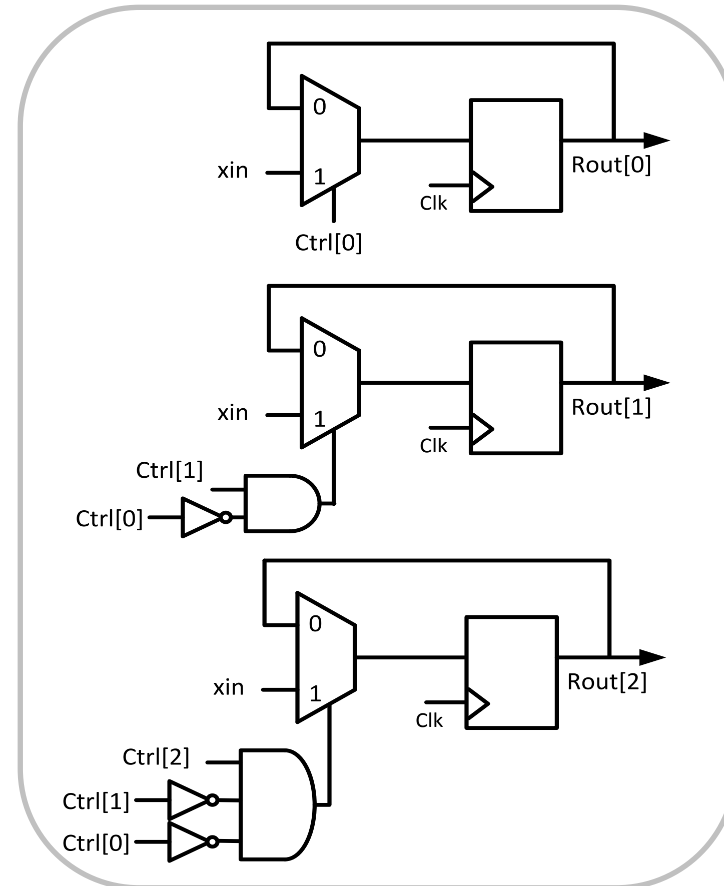
Minimized Result:



Flatten Logic Structure

- Applies to the logic that is chained due to the priority encoding
- Synthesis and layout tools are smart enough to duplicate logic to reduce fan-out, but they are not smart enough to break up logic structures that are coded in a serial Fashion

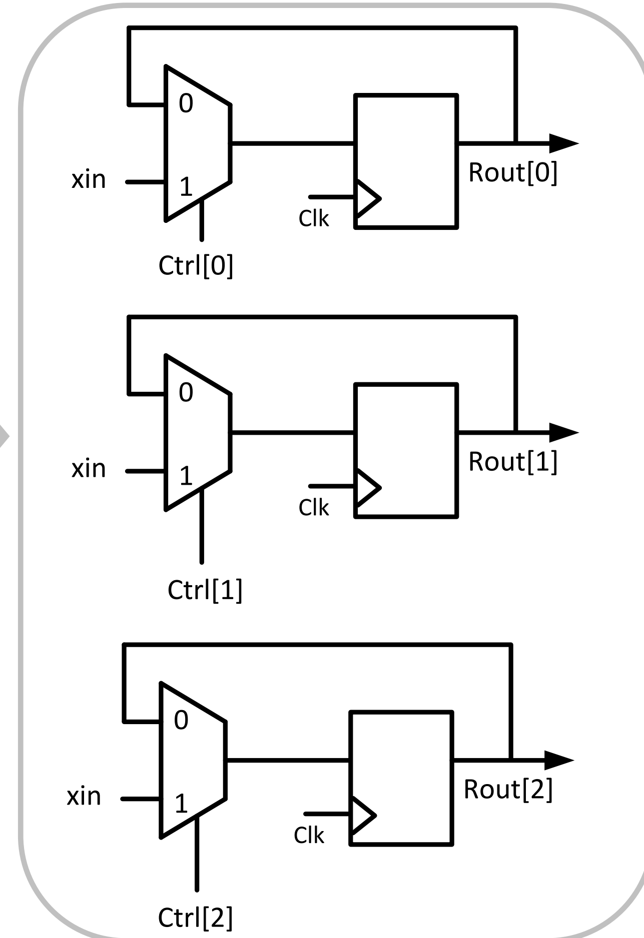
```
module regwrite(
output reg [2:0] Rout,
input clk, in,
input [2:0] Ctrl);
always @ (posedge clk)
if      (Ctrl[0]) Rout[0] <= in;
else if (Ctrl[1]) Rout[1] <= in;
else if (Ctrl[2]) Rout[2] <= in;
endmodule
```



Flatten Logic Structure

- Flatten the logic (when conditions are mutually exclusive)
  - No priority logic (each register is controlled independently)
  - Less logic delay

```
module regwrite(
output reg [2:0] rout,
input clk, in,
input [2:0] ctrl);
always @(posedge clk)
 if(ctrl[0]) rout[0] <= in;
 if (ctrl[1]) rout[1] <= in;
 if (ctrl[2]) rout[2] <= in;
endmodule
```



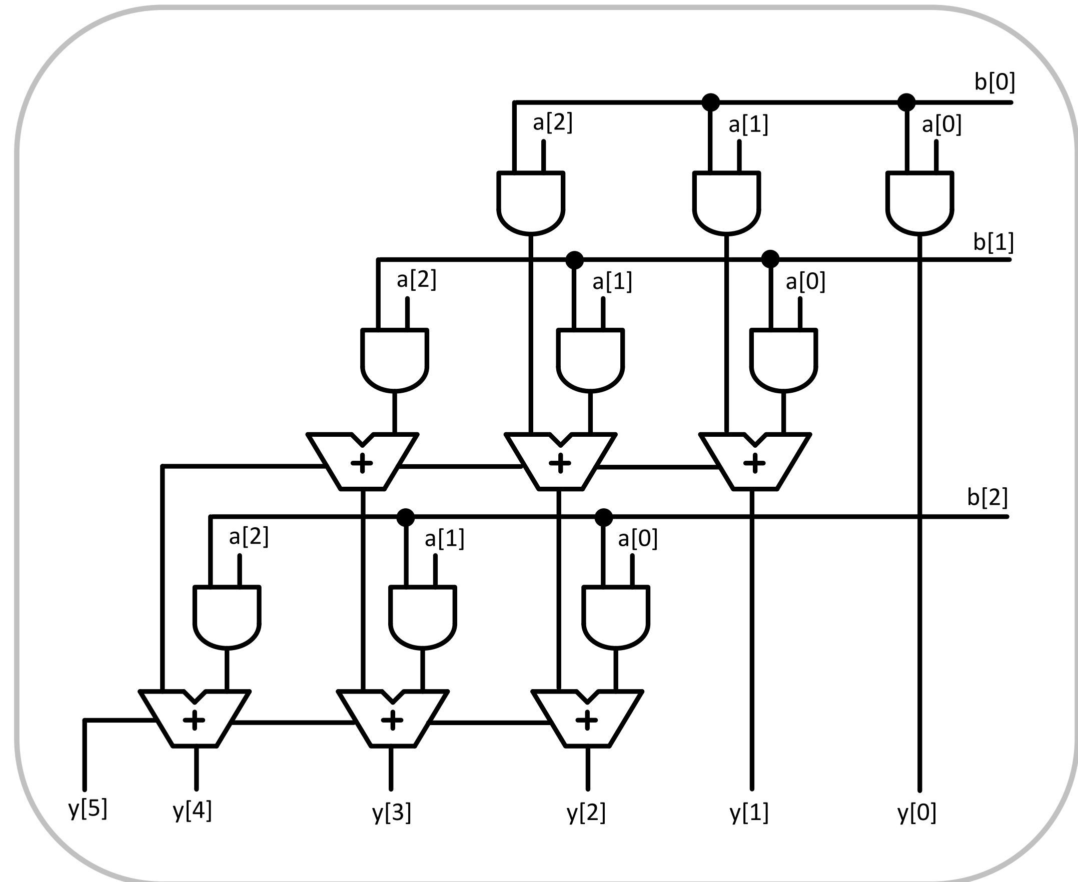
HDL for Synthesis

- ❑ It is possible to derive gate-level implementation of "+", "-", "*" operations and write them in Verilog. However, it is better to use just "+, "-", "*" and let the synthesis tool to decide which block to use to meet the performance

❖ Example:

```
module multi3x3 (a, b, y)
input [2:0] a, b;
output [5:0] y;
assign y = a * b;
```

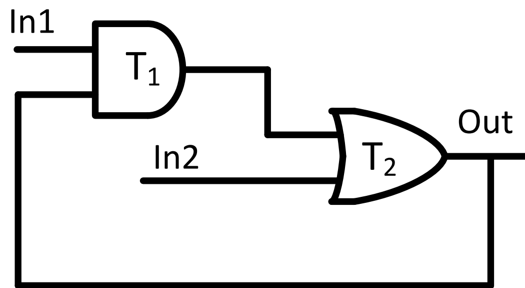
More efficient



- ❑ Unsigned operations
- ❑ Performance:
  - ➢ Power
  - ➢ Area
  - ➢ Speed

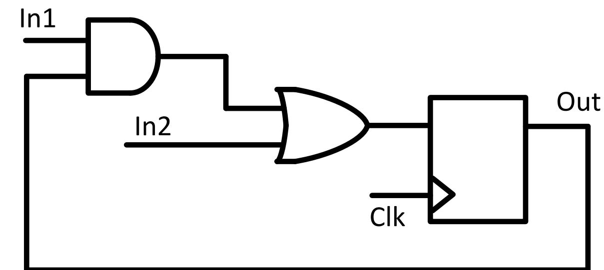
Design for Synthesis (No Timing Loop)

- Do not use timing loops in the circuit
- **Timing loop:** when an output of a combinational logic loops back to its input
  - Results in oscillation
  - Complicated timing analysis
  - Timing glitches
- **Solution:** Add a flip-flop on the feedback path




```
always @ (*)
  Out = (In1 & Out) | In2;
```

Oscillates with $f=1/(T_1+T_2)$
(Not desired)




```
always @ (posedge gated)
  Out <= (In1 & Out) | In2;
```

No Oscillation (Desired)

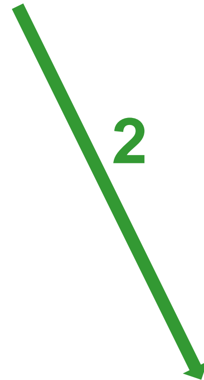
Latch Inference in Combinational Logic

- When realizing combinational logic with always block using if-else or case constructs care has to be taken to avoid latch inference after synthesis
- The latch is inferred when "incomplete" if-else or case statements are declared
- This latch is "unwanted" as the logic is combinational not sequential
- To avoid latch inference make sure to specify all possible cases "explicitly"
- Two practical approaches to avoid latch inference:
  - **For if-else construct:**
    1. Initialize the variable before the if-else construct
    2. Use else to explicitly list all possible cases
  - **For case constructs:**
    1. Use default to make sure no case is missed!
- If there is some logic path through the always block that does not assign a value to the output a latch is inferred
- Do NOT let it up to the synthesis tool to act in unspecified cases and do specify all cases explicitly.

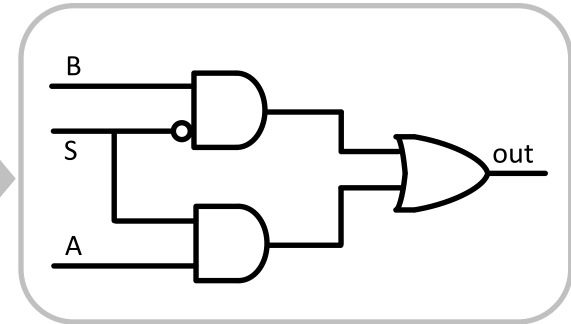
Avoid Latch Inference in If-else Statements

❖Example:

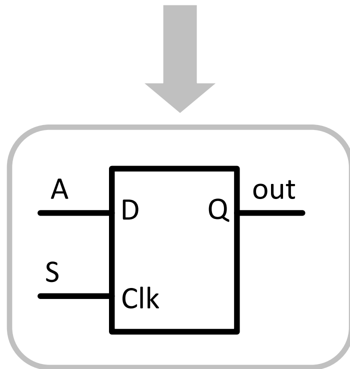
```
module DUT (A, B, S, out);
input A, B, S;
output reg out;
always @(*)
begin
   if (S==1)
       out = A;
end
endmodule
```




```
module DUT (A, B, S, out);
input A, B, S;
output reg out;
always @(*)
Begin
   out = B;
   if (S==1)
       out = A;
end
endmodule
```

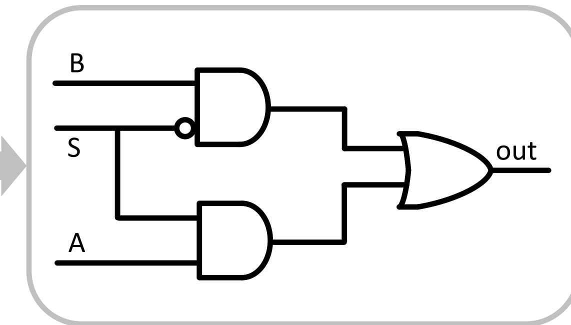


No Latch



Latch Inference

```
module DUT (A, B, S, out);
input A, B, S;
output reg out;
always @(*)
begin
   if (S==1)
       out = A;
   else
       out =B;
end
endmodule
```



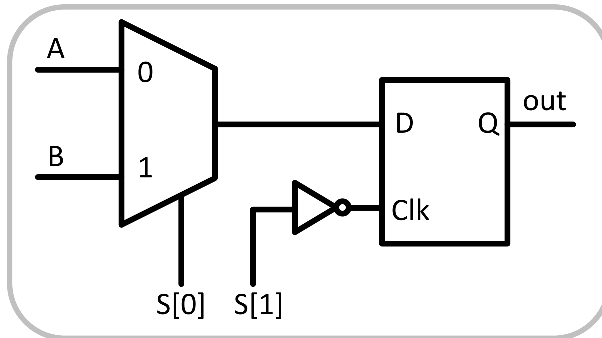
No Latch

Avoid Latch Inference in Case Statements

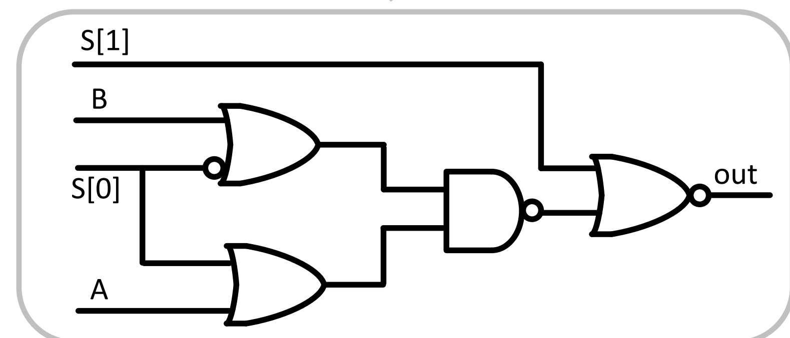
❖ Example:

```
module DUT (A, B, S, out);
input A, B;
Input [1:0] S;
output reg out;

always @(A, B, S)
begin
   case (S)
       2'b00: out = A;
       2'b01: out = B;
   endcase
end
endmodule
```

```
module DUT (A, B, S, out);
input A, B;
Input [1:0] S;
output reg out;
always @(A, B, S)
begin
   case (S)
       2'b00: out = A;
       2'b01: out = B;
       default: out = 1'b0;
   endcase
end
endmodule
```



Latch Inference



No Latch

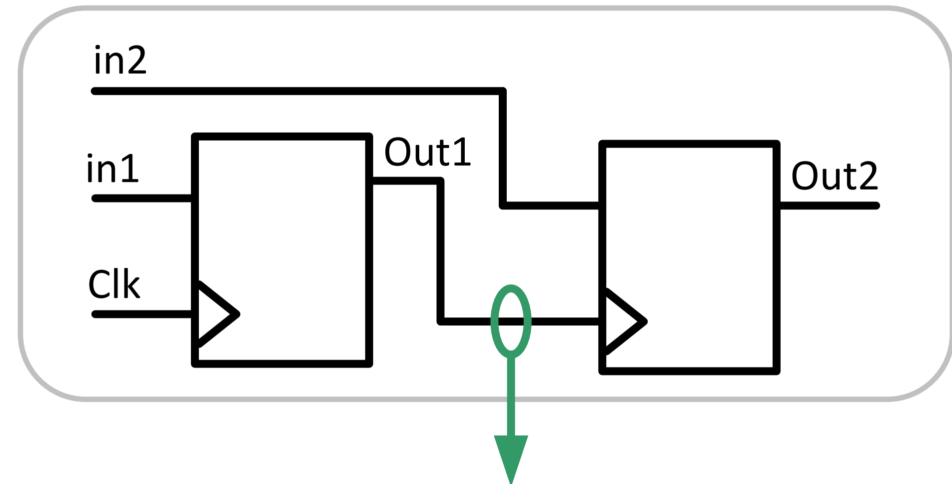
Clock

- Clock is the most important signal in the design (golden)
- Why is it different from other signals?
  - It is a global signal, i.e., it is routed across all modules in the design
- Treat clock as a golden signal
  - No buffering should be done on clock during coding and synthesis
    - Clock buffering to fix the clock skew is done during clock tree synthesis (part of APR in ASIC flow, which is done automatically)
  - No "clock gating":
    - Clock should be directly connected to flip-flops without any logic gating
    - Otherwise, it results in clock skew in the design (undesired!)

Clock (No Internally Generated Clock)

- Do not use internally generated clocks

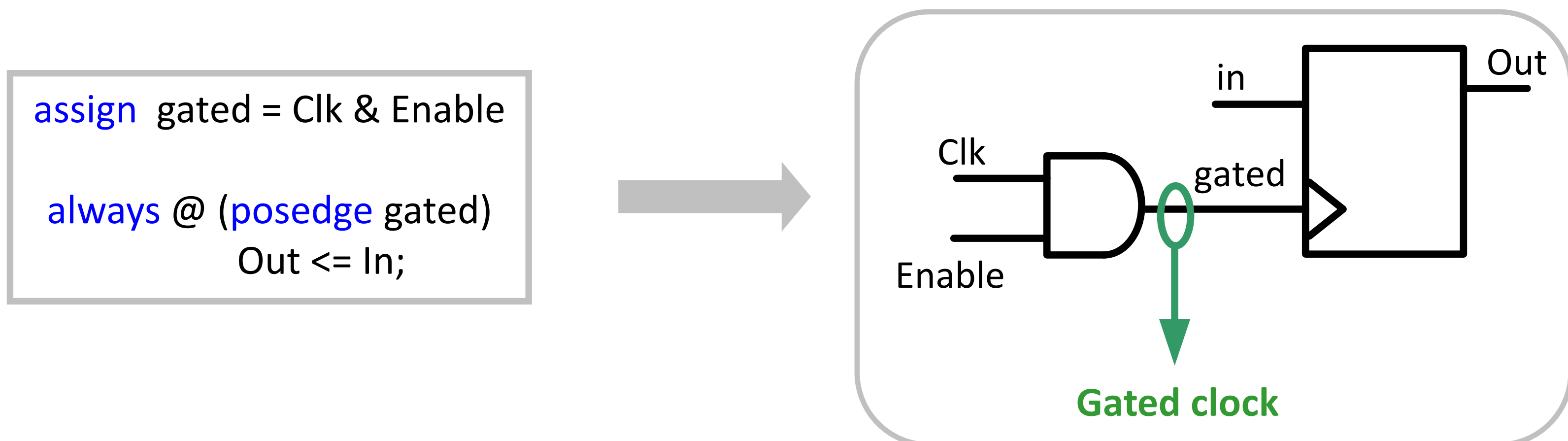
```
always @ (posedge Clk)
        Out1 <= In1;

always @ (posedge Out1)
        Out2 <= In2;
```



- Complicates the timing analysis
  - Setup time
  - Hold time
- Difficult to deal with during synthesis

Clock (No Gating)

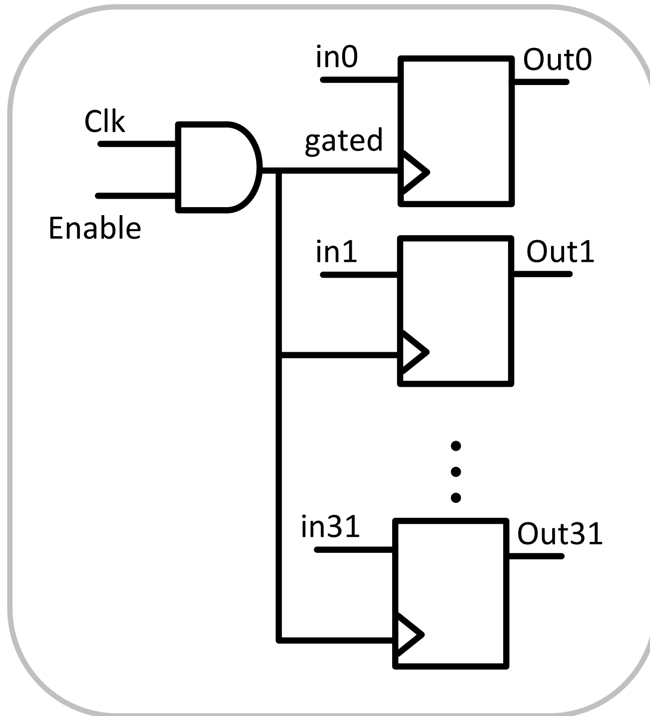
- **Gated clock**: clock that is enabled by an enable signal
- Applications:
  - Used for power saving to switch off part of the chip in fraction of time
- Avoid clock gating as much as possible
  - Because results in clock skew
  - Not a golden signal anymore!



© M. Shabany, ASIC/FPGA Chip Design

Clock (No Gating)

❑ If clock gating is used avoid large fan-outs



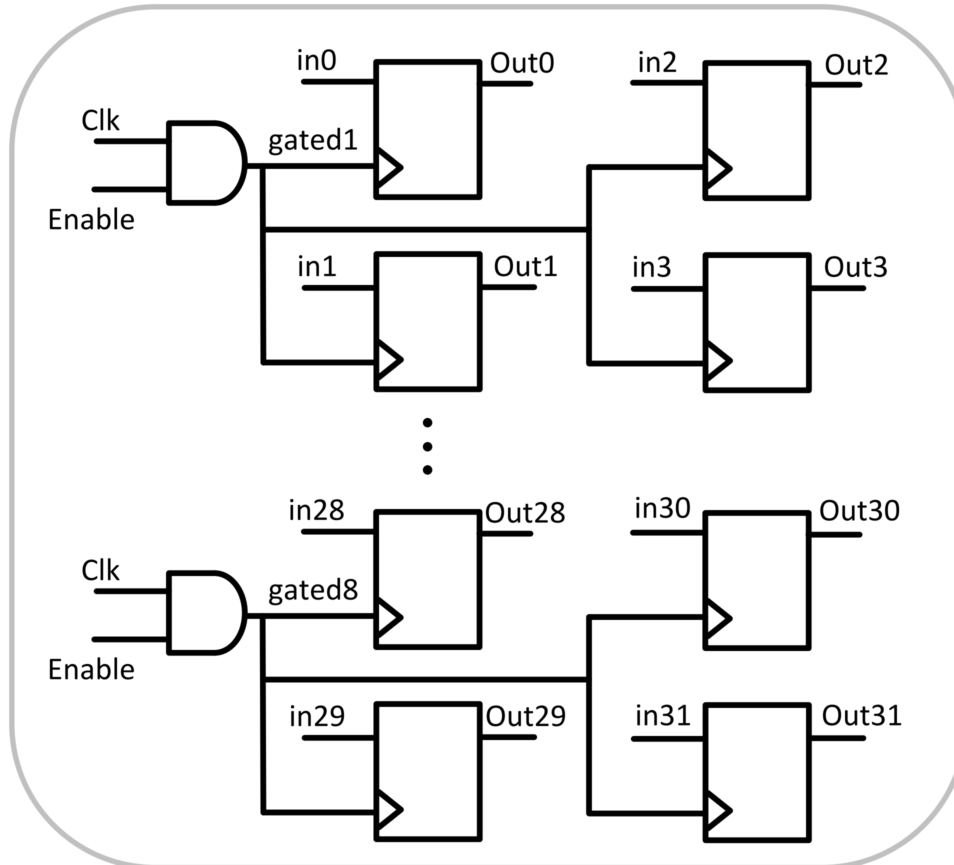

```
assign gated = Clk & Enable;

always @ (posedge gated)
        Out[31:0] <= In[31:0];
```

❑ Large fan-out (deriving 32 flip flops)

❑ Large delay ⟹ high clock skew

Clock (No Gating)

- Low Fan-out Alternative:



Explicit "and" instantiation

```
and U1 (gated1, Clk, Enable);
and U2 (gated2, Clk, Enable);
  ⋮
and U8 (gated8, Clk, Enable);


always @ (posedge gated1)
        Out[3:0] <= In[3:0];
always @ (posedge gated2)
        Out[7:4] <= In[7:4];
  ⋮
always @ (posedge gated8)
        Out[31:28] <= In[31:28];
```

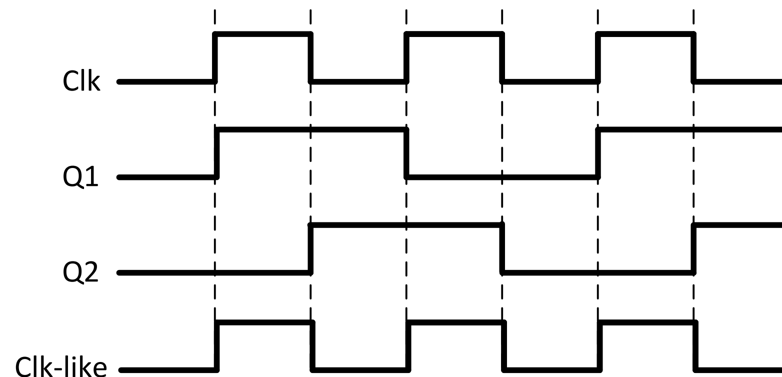
© M. Shabany, ASIC/FPGA Chip Design

Clock

- Do not use clock as an input or selector (results in an inefficient clock tree)

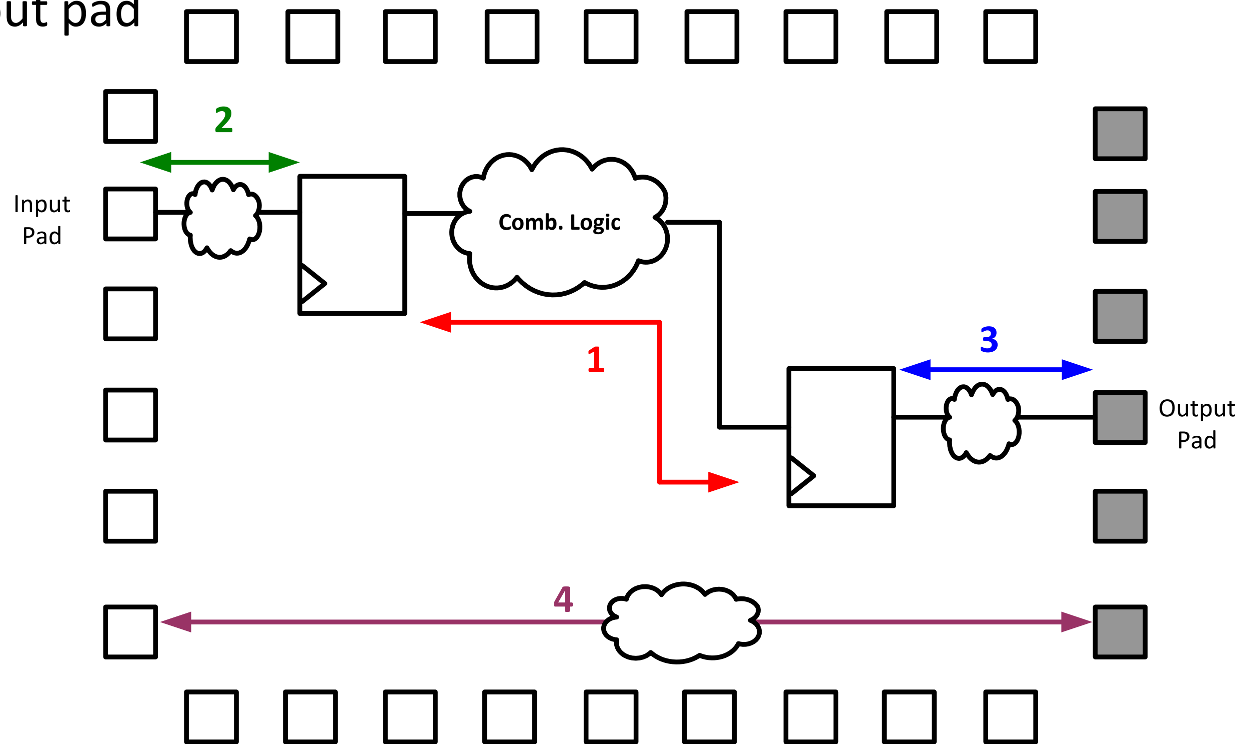
A
0
B
1
Clk

A
0
B
1
Q1
posedge Clk
Clk-like
Different from Clock
Q2
negedge Clk



Architectural Techniques : Critical Path

- Critical path in any design is the longest path between
  1. Any two internal latches/flip-flops
  2. An input pad and an internal latch
  3. An internal latch and an output pad
  4. An input pad and an output pad



Use FFs right after/before input/out pads to avoid the last three cases (off-chip and packaging delay)

The maximum delay between any two sequential elements in a design will determine the max clock speed

© M. Shabany, ASIC/FPGA Chip Design

Digital Design Metrics

- Three primary physical characteristics of a digital design:
  - **Speed**
    - Throughput
    - Latency
    - Timing
  - **Area**
  - **Power**

Digital Design Metrics

- **Speed**
  - **Throughput** :
    - The amount of data that is processed per clock cycle (bits per second)
  - **Latency**
    - The time between data input and processed data output (clock cycle)
  - **Timing**
    - The logic delays between sequential elements (clock period)
    - When a design does not meet the timing it means the delay of the critical path is greater than the target clock period

Maximum Clock Frequency: Critical Path

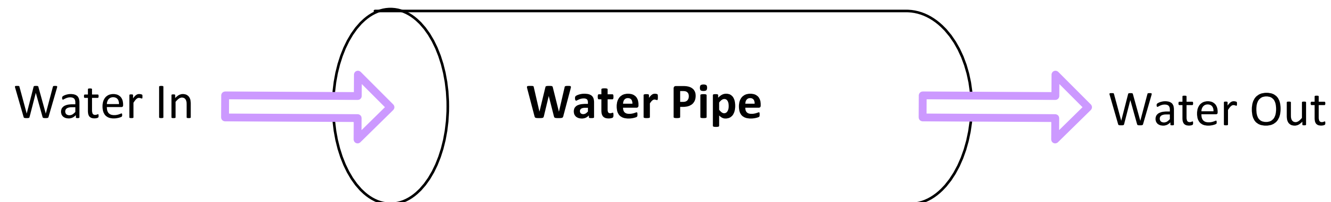
- Maximum Clock Frequency:

$$F_{max} = \frac{1}{T_{clk-q} + T_{logic} + T_{setup} + T_{routing} - T_{skew}}$$

- $T_{clk\text{-}q}$: time from clock arrival until data arrives at Q
- T_{logic} : propagation delay through logic between flip-flops
- $T_{routing}$: routing delay between flip-flops
- T_{setup} : minimum time data must arrive at D before the next rising edge of clock
- T_{skew} : propagation delay of clock between the launch flip-flop and the capture flip-flop.

Pipelining (to Improve Throughput)

- **Pipelining:**
  - Comes from the idea of a water pipe: continue sending water without waiting the water in the pipe to be out
  - Used to reduce the critical path of the design



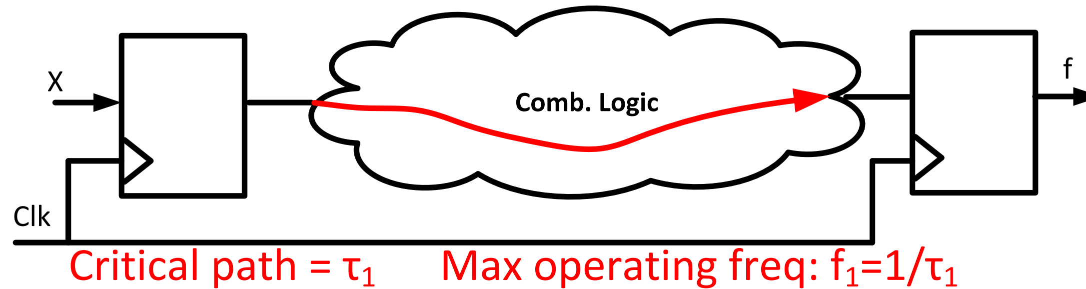
- **Advantageous**:
  - Reduction in the critical path
  - Higher throughput (number of computed results in a give time)
  - Increases the clock speed (or sampling speed)
  - Reduces the power consumption at same speed

Architectural Techniques :Pipelining

- **Pipelining:**
  - Very similar to the assembly line in the auto industry
  - The beauty of a pipelined design is that new data can begin processing before the prior data has finished, much like cars are processed on an assembly line.

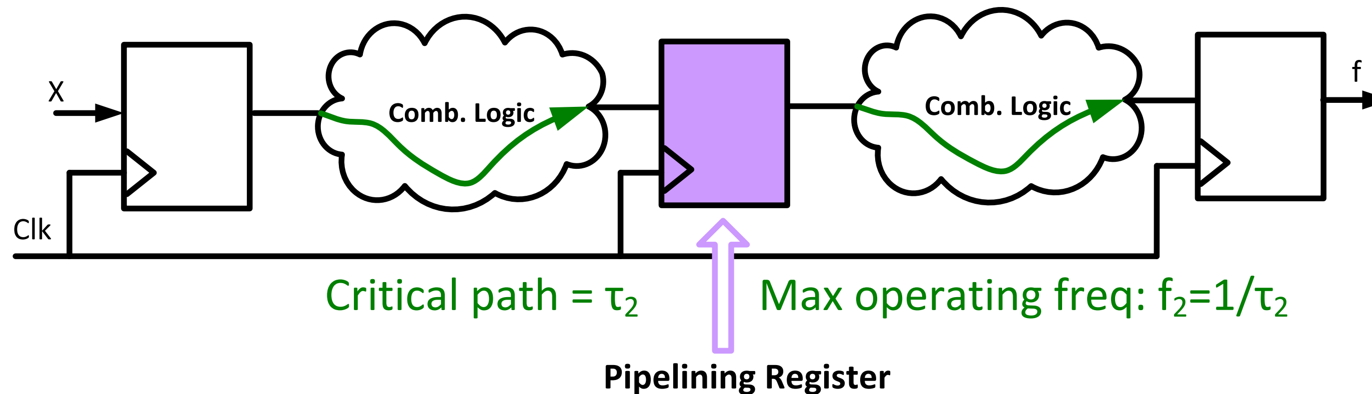
Architectural Techniques :Pipelining

- **Original System: (**Critical path = τ_1 Max operating freq: $f_1=1/\tau_1$**)**



2 cycles later

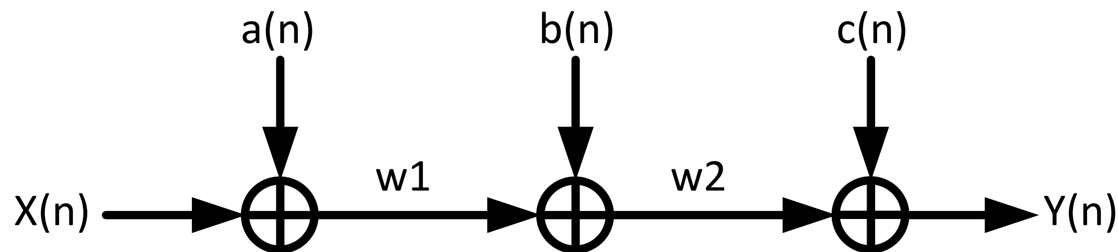
- **Pipelined version: (**Critical path = τ_2 Max operating freq: $f_2=1/\tau_2$**)**
  - Smaller Critical Path → higher throughput ($\tau_2<\tau_1$ → $f_2>f_1$)
  - Longer latency



3 cycles later

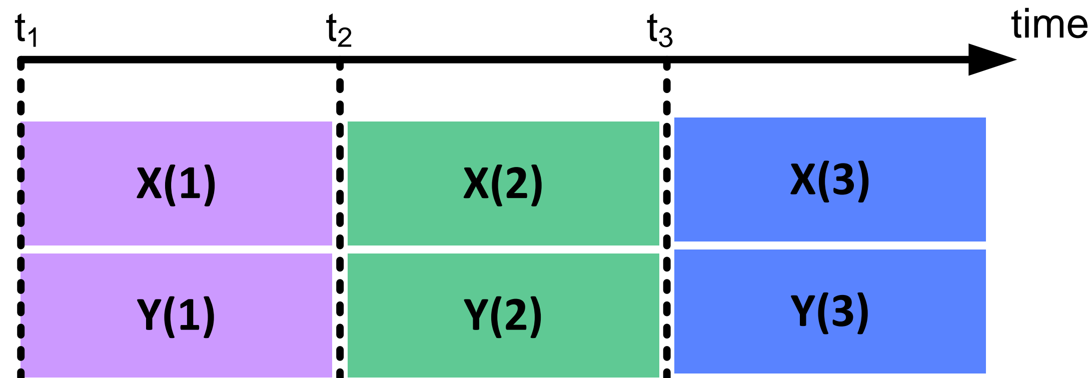
Architectural Techniques : Pipeline depth

- Pipeline depth: 0 (No Pipeline)
  - Critical path: 3 Adders



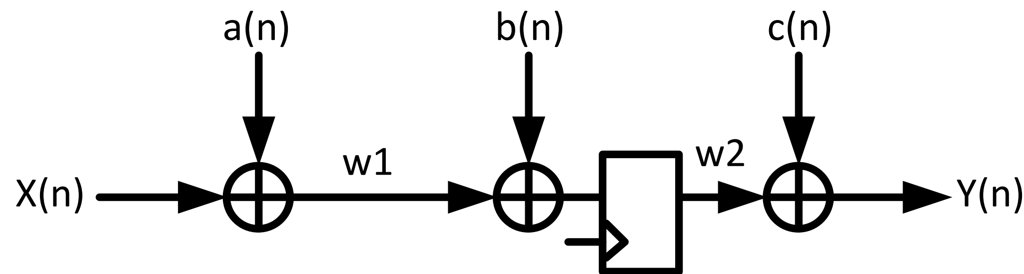

```
wire  w1, w2;
assign  w1 = X + a;
assign  w2 = w1 + b;
assign  Y = w2 + c;
```

- Latency : 0



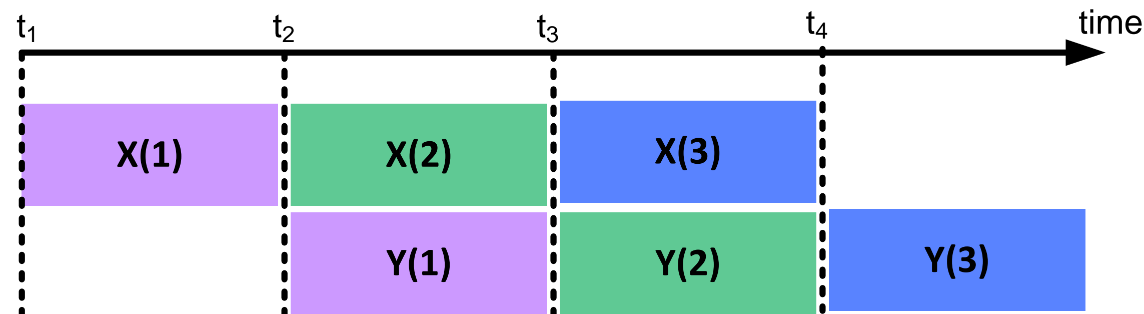
Architectural Techniques : Pipeline depth

- Pipeline depth: 1 (One Pipeline register Added)
  - Critical path: 2 Adders



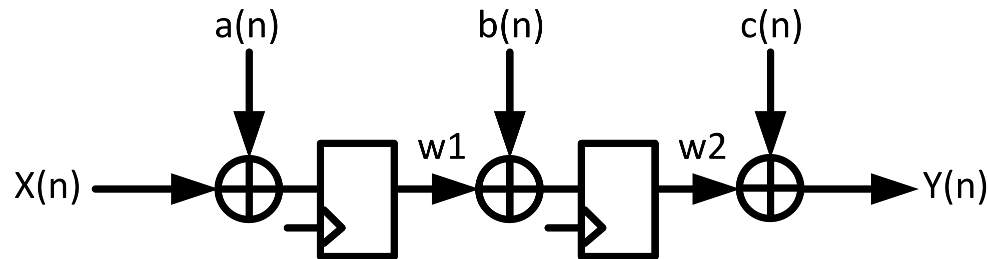

```
wire  w1;
reg   w2;
assign  w1 = X + a;
assign  Y = w2 + c;

always @(posedge Clk)
   w2 <= w1 + b;
```

- Latency : 1



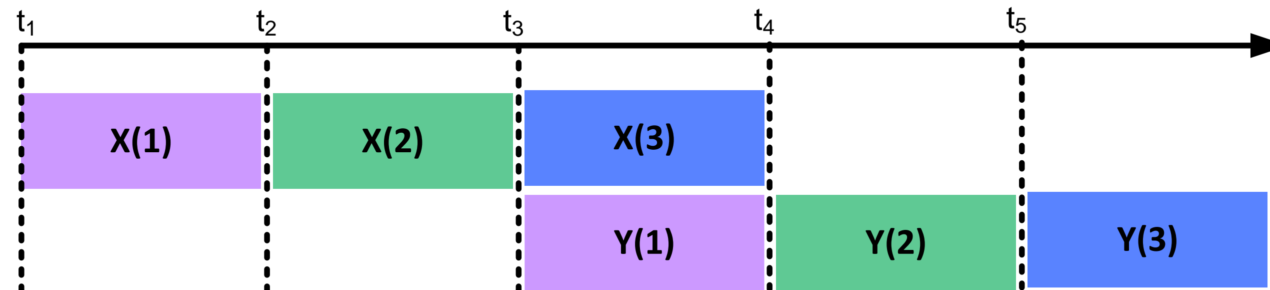
Architectural Techniques : Pipeline depth

- Pipeline depth: 2 (One Pipeline register Added)
  - Critical path: 1 Adder




```
reg   w1, w2;
assign  Y = w2 + c;

always @(posedge Clk)
  begin
     w1 <= X + a;
     w2 <= w1 + b;
  end
```

- Latency : 2

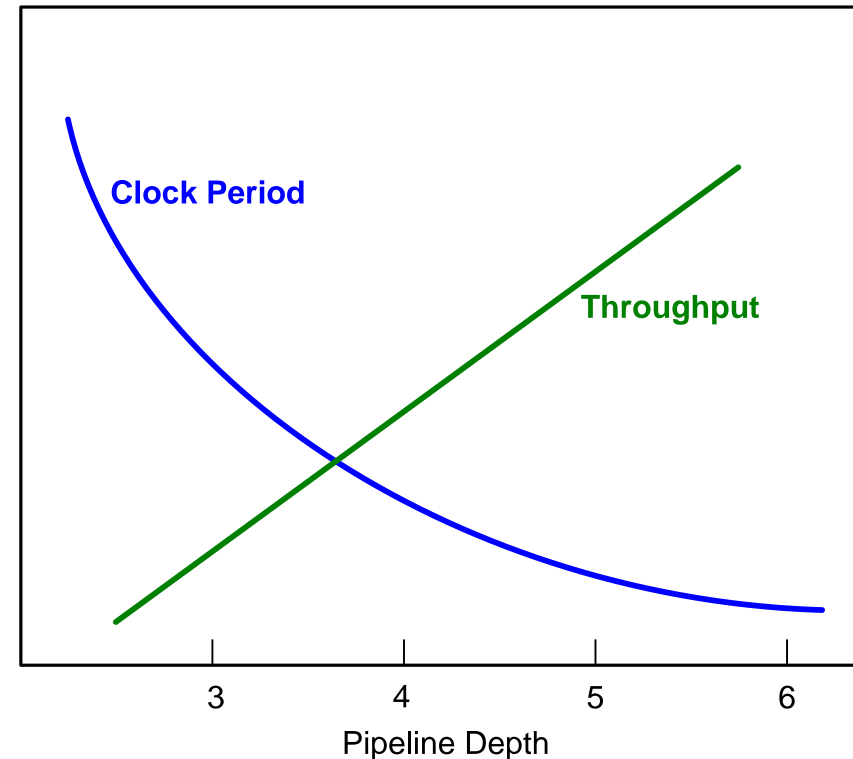


Architectural Techniques :Pipelining

- Clock period and throughput as a function of pipeline depth:

  - Clock period : $\tau_{\text{Clk}} \propto \frac{1}{n}$
  - Throughput: $\text{T} \propto n$

Adding register layers improves timing by dividing the critical path into two paths of smaller delay



Architectural Techniques : Pipelining

- **General Rule:**
  - Pipelining latches can only be placed across **feed-forward cutsets** of the circuit.

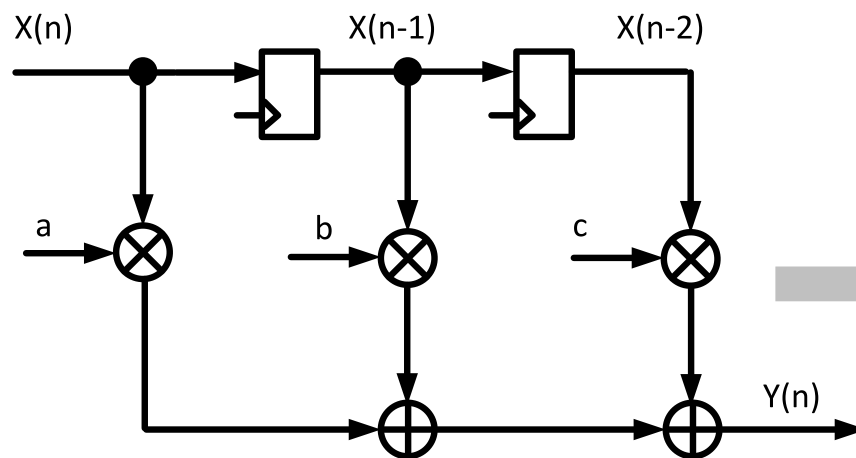
- **Cutset:**
  - A set of paths of a circuit such that if these paths are removed, the circuit becomes disjoint (i.e., two separate pieces)

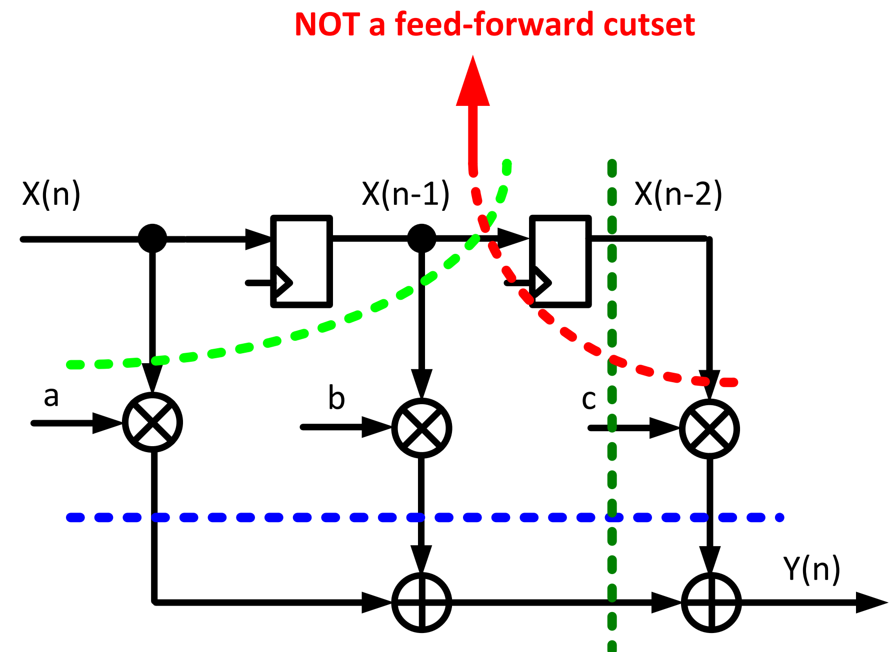
- **Feed-Forward Cutset:**
  - A cutset is called feed-forward cutset if the data moves in the forward direction on all the paths of the cutset

Architectural Techniques : Pipelining

❖ **Example:**

- FIR Filter
- Three feed-forward cutsets are shown

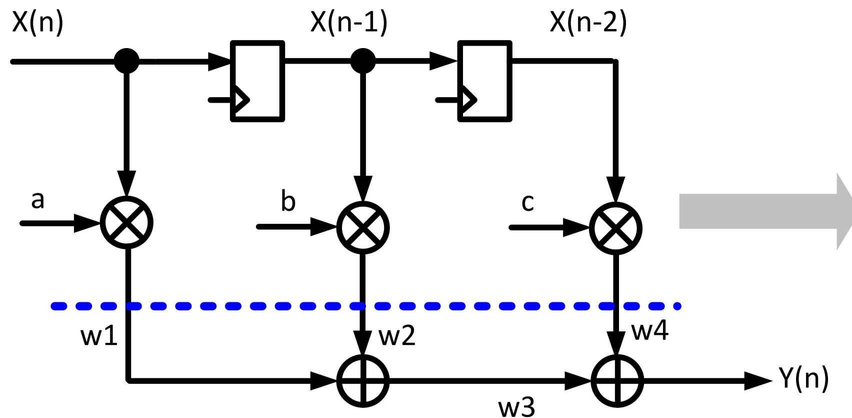


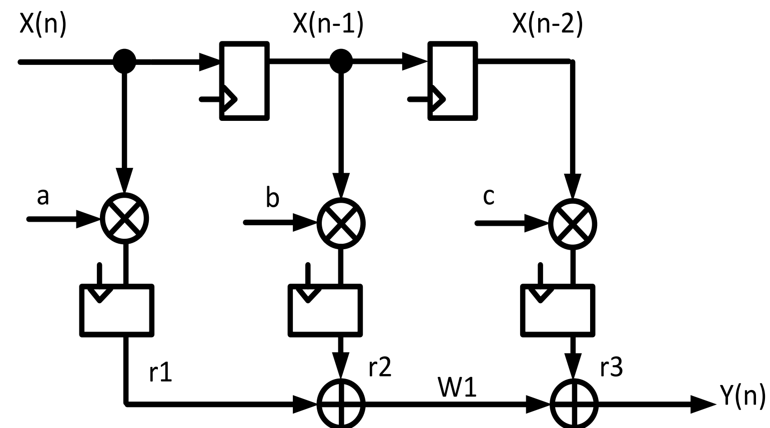


Architectural Techniques : Pipelining

Critical Path: 1M+2A

Critical Path: 2A

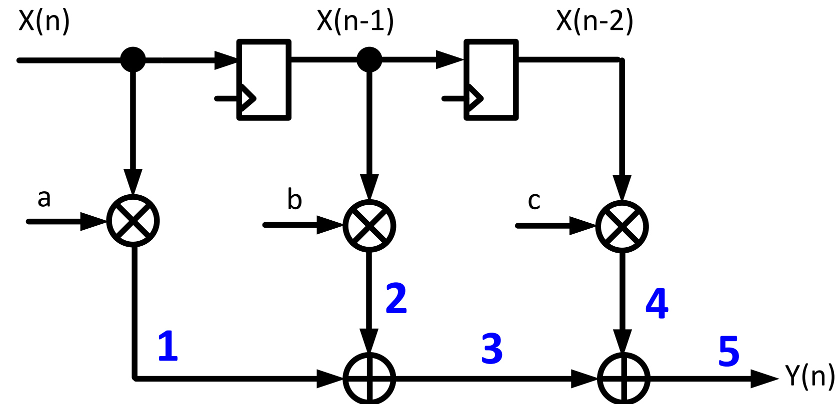





```
assign  w1 = a*Xn;
assign  w2 = b*Xn_1 ;
assign  w3 = w1 + w2;
assign  w4 = c*Xn_2;
assign  Y = w3 + w4;
always @(posedge Clk)
  begin
     Xn_1 <= Xn;
     Xn_2 <= Xn_1;
   end
```

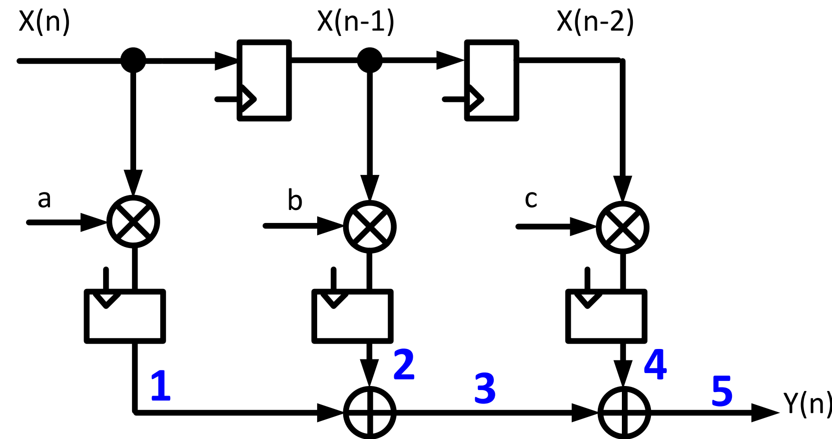
```
assign  Y = r3 + w1;
assign  w1 = r1 + r2;
always @(posedge Clk)
  begin
     Xn_1 <= Xn;
     Xn_2 <= Xn_1;
     r1 <= a*Xn;
     r2 <= b*Xn_1;
     r3 <= c*Xn_2;
   end
```

Architectural Techniques : Pipelining



Clock	Input	1	2	3	4	5	Output
0	X(0)	aX(0)	-	aX(0)	-	aX(0)	Y(0)
1	X(1)	aX(1)	bX(0)	aX(1)+bX(0)	-	aX(1)+bX(0)	Y(1)
2	X(2)	aX(2)	bX(1)	aX(2)+bX(1)	cX(0)	aX(2)+bX(1)+cX(0)	Y(2)
3	X(3)	aX(3)	bX(2)	aX(3)+bX(2)	cX(1)	aX(3)+bX(2)+cX(1)	Y(3)

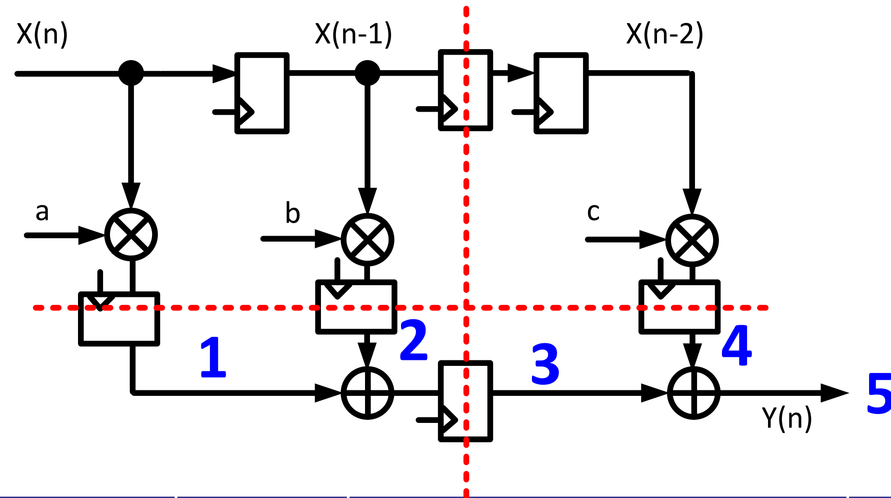
Architectural Techniques : Pipelining



Clock	Input	1	2	3	4	5	Output
0	X(0)	-	-	-	-	-	-
1	X(1)	aX(0)	-	aX(0)	-	aX(0)	Y(0)
2	X(2)	aX(1)	bX(0)	aX(1)+bX(0)	-	aX(1)+bX(0)	Y(1)
3	X(3)	aX(2)	bX(1)	aX(2)+bX(1)	cX(0)	aX(2)+bX(1)+cX(0)	Y(2)

Architectural Techniques : Pipelining

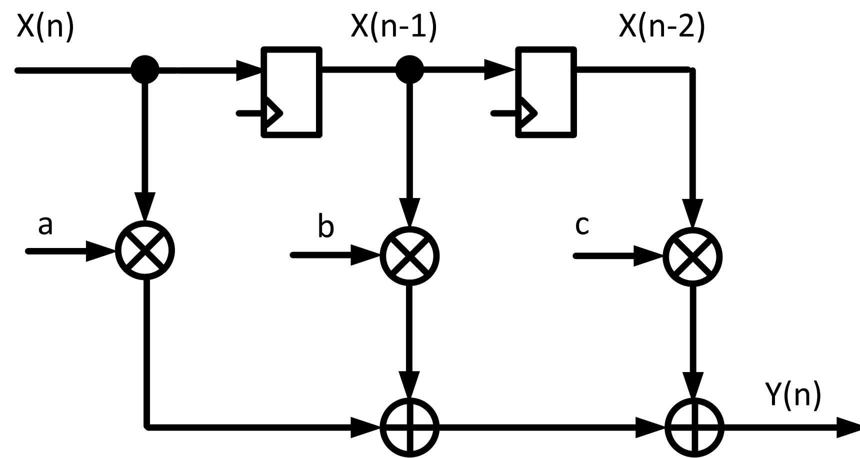
- Even more pipelining



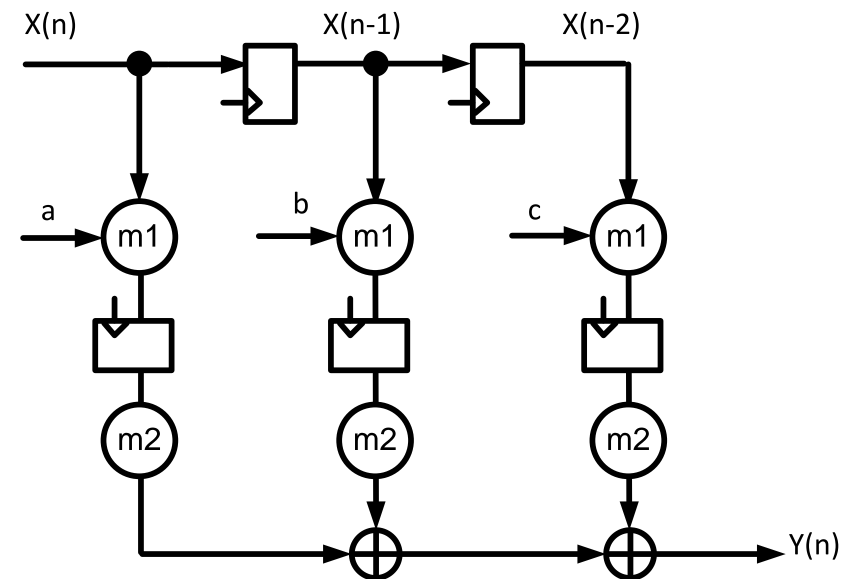
Clock	Input	1	2	3	4	5	Output
0	X(0)	-	-	-	-	-	-
1	X(1)	aX(0)	-	-	-	-	-
2	X(2)	aX(1)	bX(0)	aX(0)	-	aX(0)	Y(0)
3	X(3)	aX(2)	bX(1)	aX(1)+bX(0)	-	aX(1)+bX(0)	Y(1)
4	X(3)	aX(2)	bX(1)	aX(2)+bX(1)	cX(0)	aX(2)+bX(1)+cX(0)	Y(2)

Architectural Techniques : Fine-Grain Pipelining

- Pipelining at the operation level
  - Break the multiplier into two parts



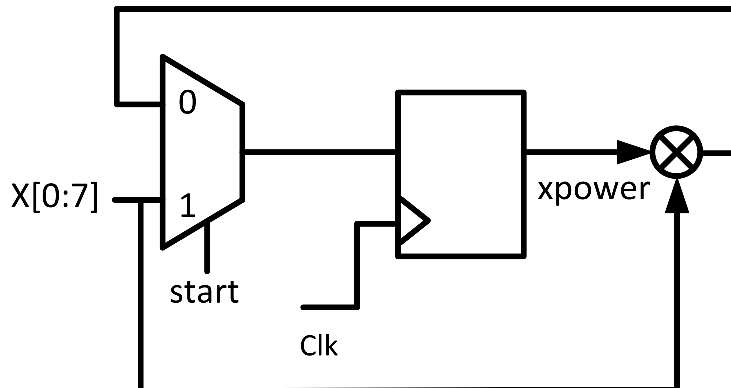
Fine-Grain Pipelining



Unrolling the Loop Using Pipelining

- Calculation of X^3
  - Throughput = 8/3, or 2.7 bits/clock
  - Latency = 3 clocks
  - Timing = One multiplier in the critical path

- Iterative implementation:
- No new computations can begin until the previous computation has completed

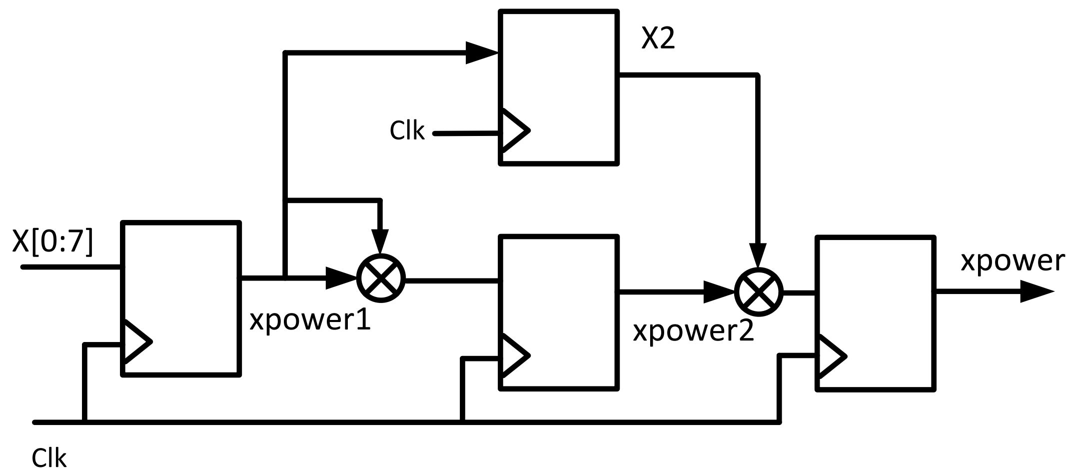



```
module power3(
output reg [7:0] X3,
output finished,
input [7:0] X,
input clk, start);
reg [7:0] ncount;
reg [7:0] Xpower, Xin;
assign finished = (ncount == 0);
        always@(posedge clk)
        if (start) begin
        XPower <= X; Xin<=X;
        ncount <= 2;
        X3 <= XPower;
        end
        else if(!finished) begin
        ncount <= ncount - 1;
        XPower <= XPower * Xin;
        End
endmodule
```

Unrolling the Loop Using Pipelining

- Calculation of X^3
  - Throughput = 8/1, or 8 bits/clock (3X improvement)
  - Latency = 3 clocks
  - Timing = One multiplier in the critical path
- **Penalty**: More Area

Unrolling an algorithm with n iterative loops increases throughput by a factor of n



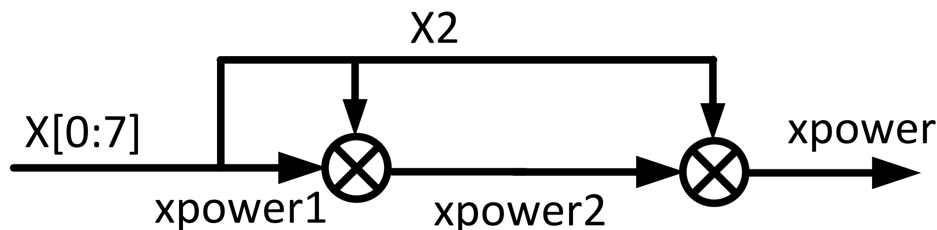

```
module power3(
output reg [7:0] XPower,
input clk,
input [7:0] X);
reg [7:0] XPower1, XPower2;
reg [7:0] X2;
always @(posedge clk) begin
// Pipeline stage 1
XPower1 <= X;
// Pipeline stage 2
XPower2 <= XPower1 * XPower1 ;
X2 <= XPower1 ;
// Pipeline stage 3
XPower <= XPower2 * X2;
end
endmodule
```

Removing Pipeline Registers (to Improve Latency)

- Calculation of X^3
  - Throughput = 8 bits/clock (3X improvement)
  - Latency = 0 clocks
  - Timing = Two multipliers in the critical path

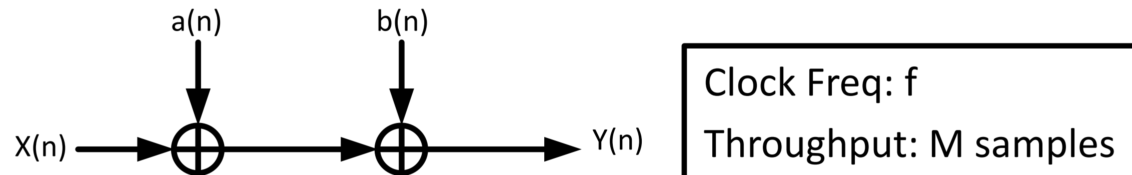
Latency can be reduced by removing pipeline registers

```
module power3(
Output [7:0] XPower,
input [7:0] X);
reg [7:0] XPower1, XPower2;
reg [7:0] X1, X2;
always @(*)
   XPower1 = X;
always @(*)
begin
   X2 = XPower1;
   XPower2 = XPower1*XPower1;
end

assign XPower = XPower2 * X2;

endmodule
```



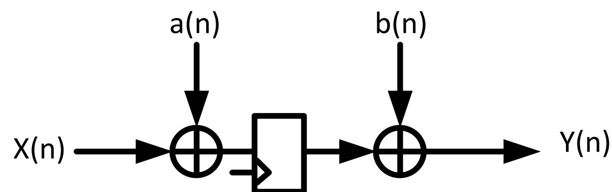
Architectural Techniques : Parallel Processing

- In parallel processing the same hardware is duplicated to
  - Increases the throughput without changing the critical path
  - Increases the silicon area



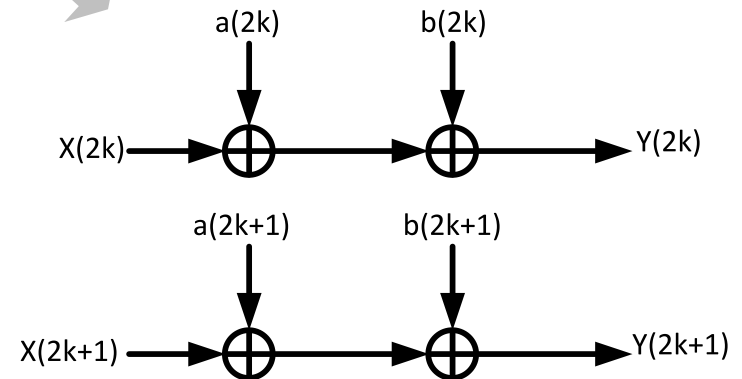
Clock Freq: f
Throughput: M samples

Pipelining



Clock Freq: 2f
Throughput: 2M samples

Parallel Processing



Clock Freq: f
Throughput: 2M samples

Architectural Techniques : Parallel Processing

- Parallel processing for a 3-tap FIR filter
  - Both have the same critical path (M+2A)

Parallel Factor:3

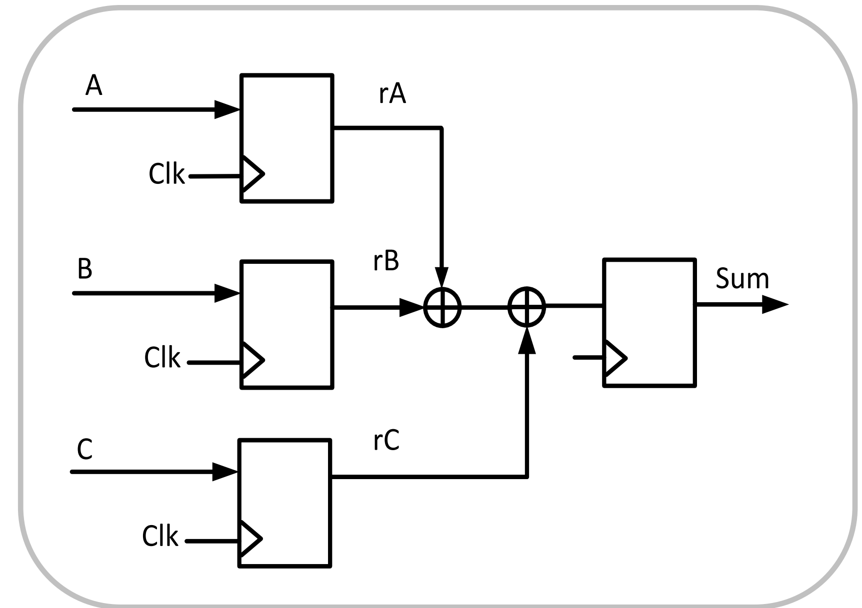
© M. Shabany, ASIC/FPGA Chip Design

Register Balancing (to Improve Timing)

❑ Redistribute logic evenly between registers to minimize the worst-case delay between any two registers

➢ b/c clock is limited by only the worst-case delay

```
module adder(
output reg [7:0] Sum,
input [7:0] A, B, C,
input clk);
reg [7:0] rA, rB, rC;
always @(posedge clk) begin
rA <= A;
rB <= B;
rC <= C;
Sum <= rA + rB + rC;
end
endmodule
```

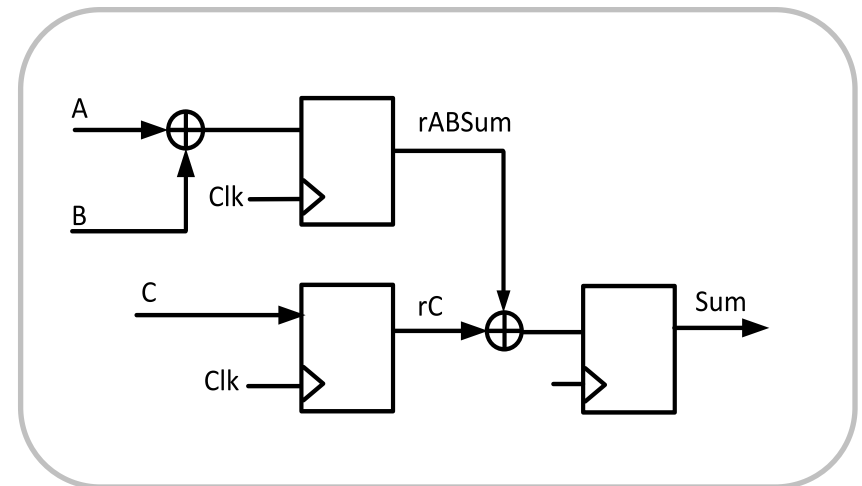


Register Balancing (to Improve Timing)

❑ Redistribute logic evenly between registers to minimize the worst-case delay between any two registers

➢b/c clock is limited by only the worst-case delay

```
module adder(
output reg [7:0] Sum,
input [7:0] A, B, C,
input clk);
reg [7:0] rABSum, rC;
always @(posedge clk) begin
rABSum <= A + B;
rC <= C;
Sum <= rABSum + rC;
end
endmodule
```



Speed-related Techniques: Summary

- **Throughput-related:**
  - A high-throughput architecture is one that maximizes the number of bits per second that can be processed by a design.
  - Unrolling an iterative loop increases throughput.
  - The penalty for unrolling an iterative loop is an increase in area.

- **Latency-related:**
  - A low-latency architecture is one that minimizes the delay from the input of a module to the output.
  - Latency can be reduced by removing pipeline registers.
  - The penalty for removing pipeline registers is an increase in combinatorial delay between registers.

Speed-related Techniques: Summary

- **Timing-related:**
  - Timing refers to the clock speed of a design. A design meets timing when the maximum delay between any two sequential elements is smaller than the minimum clock period.
  - Adding register layers improves timing by dividing the critical path into two paths of smaller delay.
  - Separating a logic function into a number of smaller functions that can be evaluated in parallel reduces the delay to the longest of the substructures.
  - By removing priority encodings where they are not needed, the logic structure is flattened, and the path delay is reduced.
  - Register balancing improves timing by moving combinatorial logic from the critical path to an adjacent path.

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

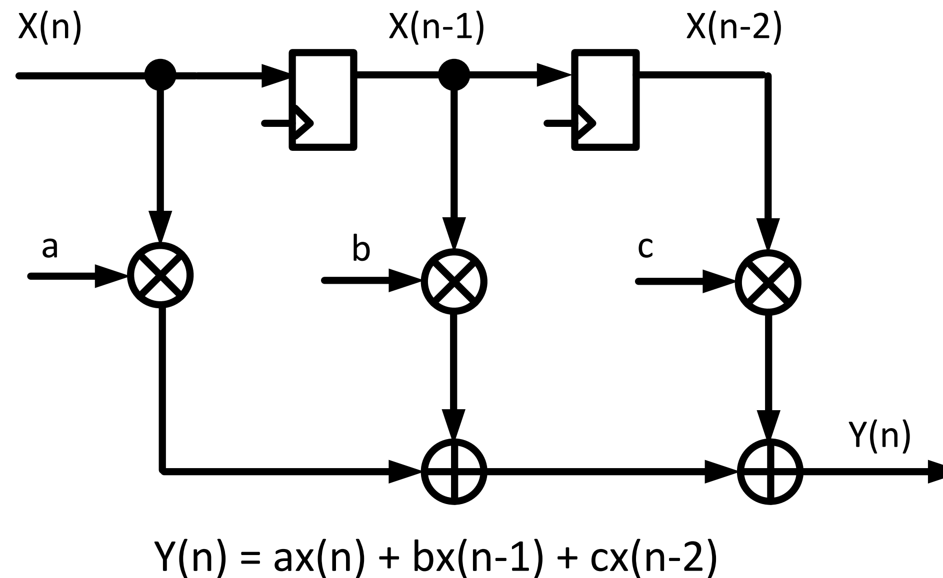
- Rolling Up the Pipeline:
  - Using series of shift-and-add operations
  - Smaller critical path
  - Less area due to the simple operations and sharing
  - One product every 8 clock cycles! (low throughput)

```
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



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

Clk

State	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
X0	0	X[0]	X[0]	X[0]	X[0]	X[1]	X[1]	X[1]	X[1]	X[2]	X[2]	X[2]	X[2]	X[3]	X[3]	X[3]
X1	0	0	0	0	0	X[0]	X[0]	X[0]	X[0]	X[1]	X[1]	X[1]	X[1]	X[2]	X[2]	X[2]
X2	0	0	0	0	0	0	0	0	0	X[0]	X[0]	X[0]	X[0]	X[1]	X[1]	X[1]
multout	0	AX0	0	0	0	AX1	BX0	0	0	AX2	BX1	CX0	0	AX3	BX2	CX1
accum	0	0	AX0	AX0	0	0	AX1	AX1+BX2	0	0	AX2	AX2+BX1	0	0	AX3	AX3+BX2
accumsum	0	0	AX0	AX0	AX0	0	AX1	AX1+BX2	AX1+BX2	0	AX2	AX2+BX1	AX2+BX1 +CX0	0	AX3	AX3+BX2
out	0	0	0	0	0	AX0	AX0	AX0	AX0	AX1+BX2	AX1+BX2	AX1+BX2	AX1+BX2	AX2+BX1 +CX0	AX2+BX1 +CX0	AX2+BX1 +CX0

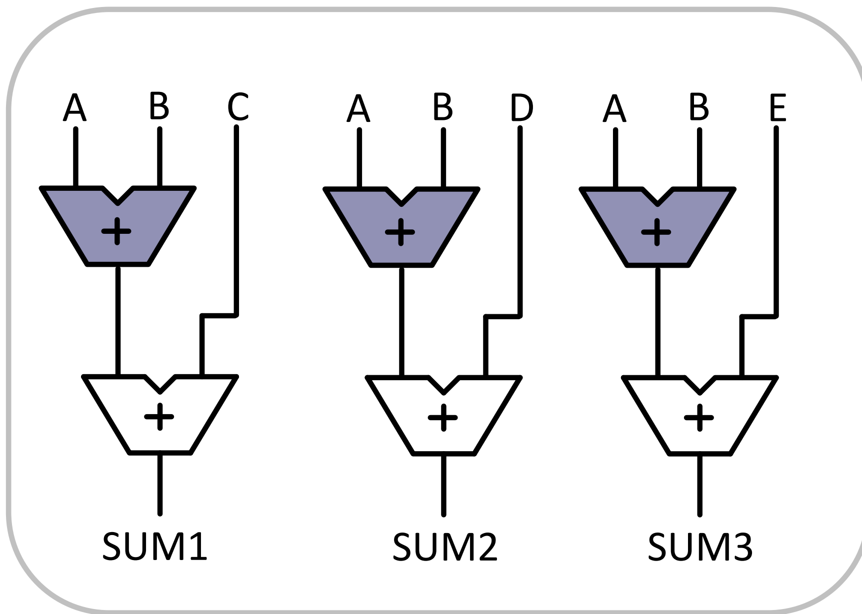
© M. Shabany, ASIC/FPGA Chip Design

Resource Sharing: Area Reduction Technique

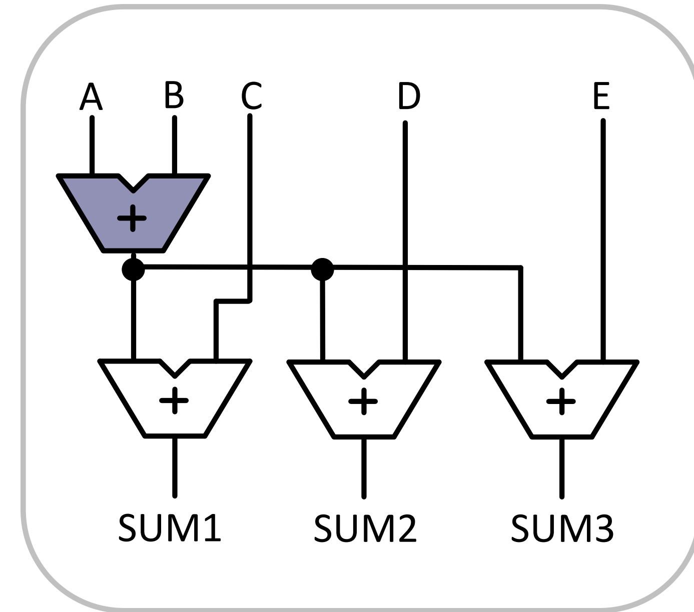
- HDL coding style can force a specific topology to be synthesized

```
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;
```

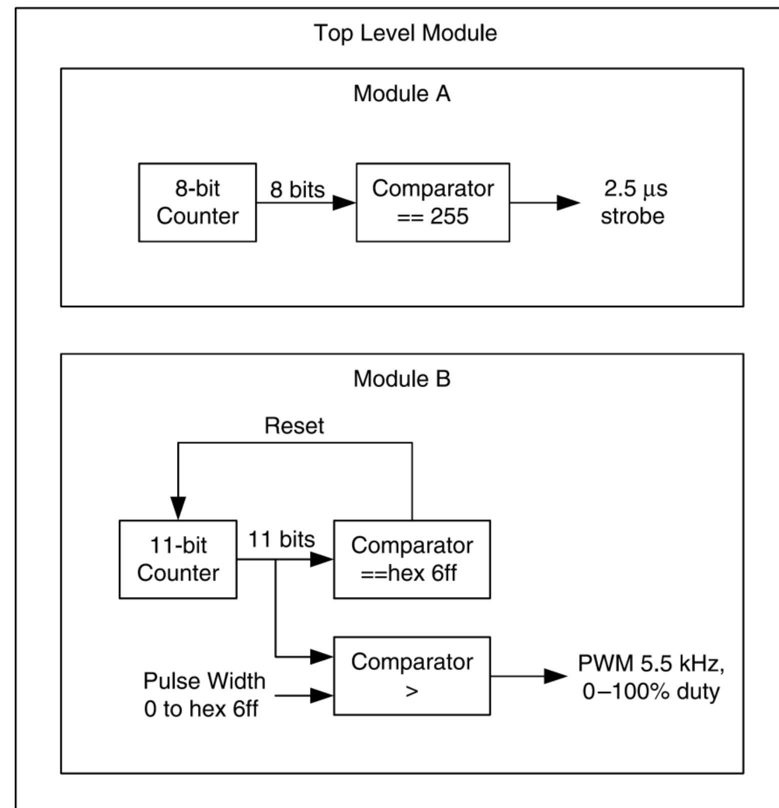


Sharing



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

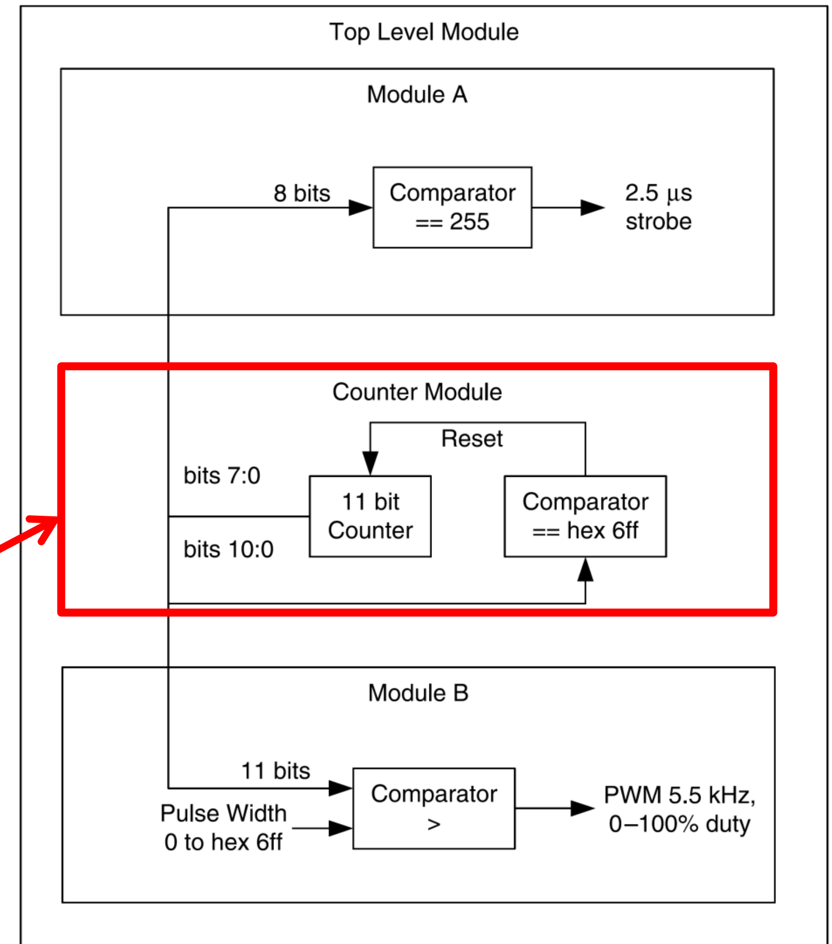


Resource Sharing: Area Reduction Technique

❑ 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



© M. Shabany, ASIC/FPGA Chip Design

Area Techniques: Reset Issues

- Some predefined modules are optimized area-wise in all vendors
- These modules have a specific constraint, e.g., only support synchronous reset, or only support reset to zero
- If the same functionality is used but violating these constraints, the function can not be realizable using predefined optimized modules
- In this case, the function is realized using general LUTs, thus a lot more area

Area Techniques: Reset Issues

- Multiply-Accumulate (MAC) module can be realized using built-in DSP blocks

```
module dsp(
output reg [15:0] oDat,
Input Reset, Clk,
input [7:0] Dat1, Dat2);
reg [15:0] multfactor;
always @(posedge Clk or negedge Reset)
if(!Reset) begin
multfactor <= 0;
oDat <= 0;
end
else begin
multfactor <= (Dat1 * Dat2);
oDat <= multfactor + oDat;
end
endmodule
```

Resources	Amount
Total Pins	34
Combination LUTs	0
Dedicated Logic Registers	0
Total Register	0
9-bit DSP Block	4

Area Techniques: Reset Issues

- Multiply-Accumulate (MAC) module can **NOT** be realized using built-in DSP blocks
- Because the DSP block is customized only for asynchronous reset.

```
module dsp(
output reg [15:0] oDat,
Input Reset, Clk,
input [7:0] Dat1, Dat2);
reg [15:0] multfactor;
always @(posedge Clk)
if(!Reset) begin
multfactor <= 0;
oDat <= 0;
end
else begin
multfactor <= (Dat1 * Dat2);
oDat <= multfactor + oDat;
end
endmodule
```

Resources	Amount
Total Pins	34
Combination LUTs	16
Dedicated Logic Registers	32
Total Register	32
9-bit DSP Block	1

Area Techniques: Reset Issues

- Multiply-Accumulate (MAC) module can **NOT** be realized using built-in DSP blocks
- Because asynchronous reset can be done only to zero not a non-zero value

```
module dsp(
output reg [15:0] oDat,
Input Reset, Clk,
input [7:0] Dat1, Dat2);
reg [15:0] multfactor;
always @(posedge Clk or negedge Reset)
if(!Reset) begin
multfactor <= 16'hffff;
oDat <= 16'hffff;
end
else begin
multfactor <= (Dat1 * Dat2);
oDat <= multfactor + oDat;
end
endmodule
```

Resources	Amount
Total Pins	34
Combination LUTs	48
Dedicated Logic Registers	32
Total Register	32
9-bit DSP Block	1

RAM Reset Issues

- Some vendors have either synchronous or asynchronous Block RAMs (BRAMs)

```
module resetckt(
output reg [15:0] oDat,
input iReset, iClk, iWrEn,
input [7:0] iAddr, oAddr,
input [15:0] iDat);
reg [15:0] memdat [0:255];
always @(posedge iClk)
if(!iReset)
oDat <= 0;
else begin
if(iWrEn)
memdat[iAddr] <= iDat;
oDat <= memdat[oAddr];
end
endmodule
```

Resources	Amount
Total Pins	51
Combination LUTs	1652
Dedicated Logic Registers	4152
Total Register	4152
Total Block Memory Bits	0

RAM Reset Issues

- Some vendors have either synchronous or asynchronous Block RAMs (BRAMs)

```
module resetckt(
output reg [15:0] oDat,
input iReset, iClk, iWrEn,
input [7:0] iAddr, oAddr,
input [15:0] iDat);
reg [15:0] memdat [0:255];
always @(posedge iClk or negedge iReset)
if(!iReset)
oDat <= 0;
else begin
if(iWrEn)
memdat[iAddr] <= iDat;
oDat <= memdat[oAddr];
end
endmodule
```

Resources	Amount
Total Pins	51
Combination LUTs	17
Dedicated Logic Registers	1
Total Register	1
Total Block Memory Bits	4096

Summary: An optimized FPGA resource will not be used if an incompatible reset is assigned to it. The function will be implemented with generic elements and will occupy more area.

Power Reduction Techniques:

- In CMOS technology, dynamic power consumption is related to charging and discharging parasitic capacitances on gates and metal traces.

$$P = C.f.V^2$$

- **C**: Capacitance
  - Related to the number of gates that are toggling at any given time and the lengths of the routes connecting the gates
- **f**: frequency (related to the clock frequency)
- **V**: voltage (usually fixed in FPGAs)

All of the power-reduction techniques ultimately aim at reducing one of these three components.

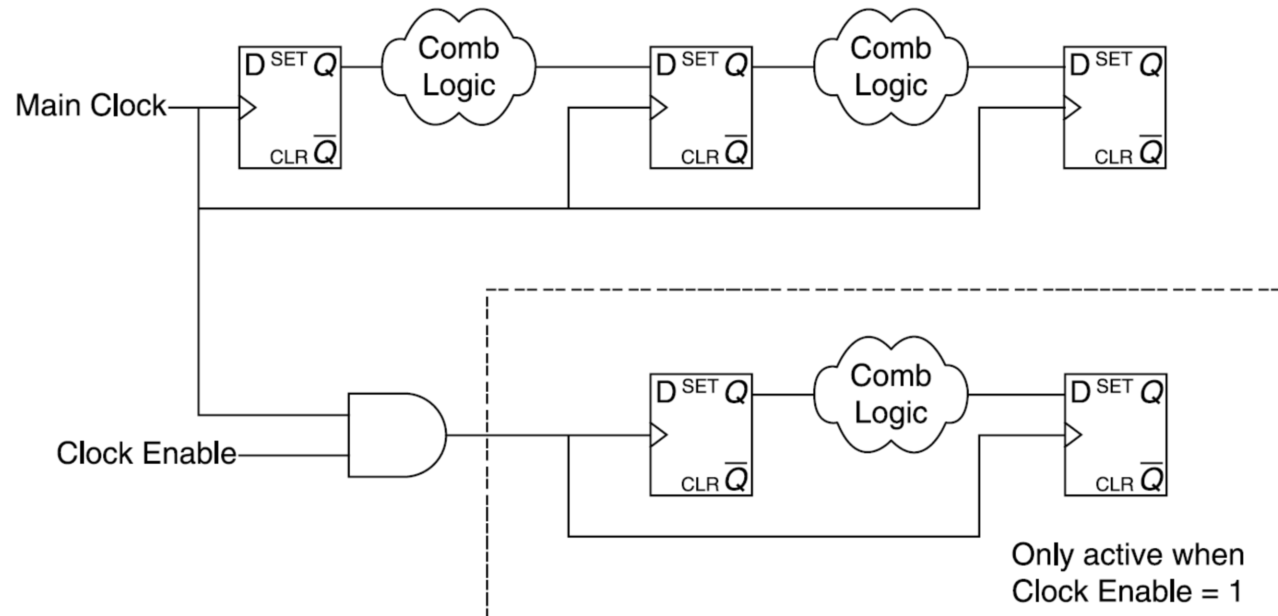
Power Reduction Techniques: Clock Issues

- The most effective power reduction technique related to the clock is to dynamically disable the clock in specific regions that do not need to be active in specific times

- To do this use
  - Clock enable pin on Flip-flops (available in most modern devices)
  - Global clock mux
  - Clock gating (not preferred)

By gating portions of circuitry, the designer reduces the dynamic power dissipation proportional to the amount of logic (capacitance C) and the average toggling frequency of the corresponding gates (frequency f). Note that the clock tree dissipates a lot of power!

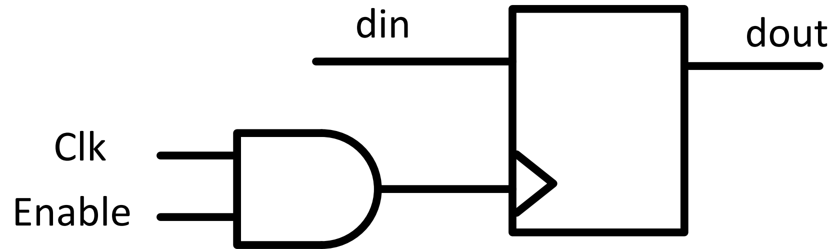
Power Reduction Techniques: Clock Issues



- ❑ When a clock is gated, the new net on the clock generates a new clock domain. This new clock net requires a low-skew path to all FFs in its domain.
- ❑ For ASIC, these low-skew lines can be built in the custom clock tree, but it is problematic for FPGA b/c of the limited number/fixed layout of low-skew lines.

Power Reduction Techniques: Gated Clock

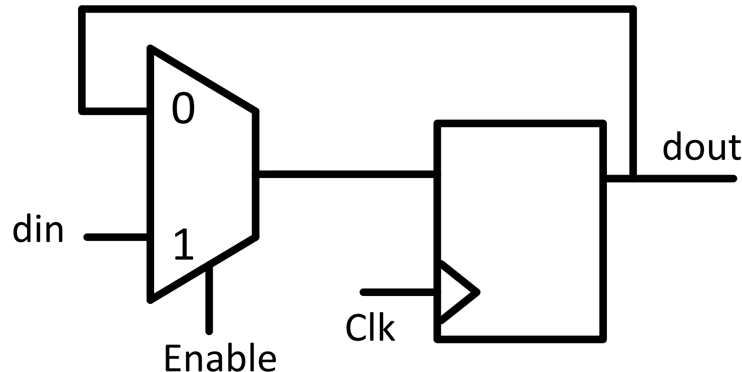
- Some synthesis tools have an option called “fix gated clocks”.
- This feature will automatically move the gating operation off of the clock line and into the data path.




```
module clockstest(
output reg dout,
input Clk, Enable,
input Din);
wire gated_clock = Clk & Enable;
always @(posedge gated_clock)
dout <= din;
endmodule
```

Gated clock removed

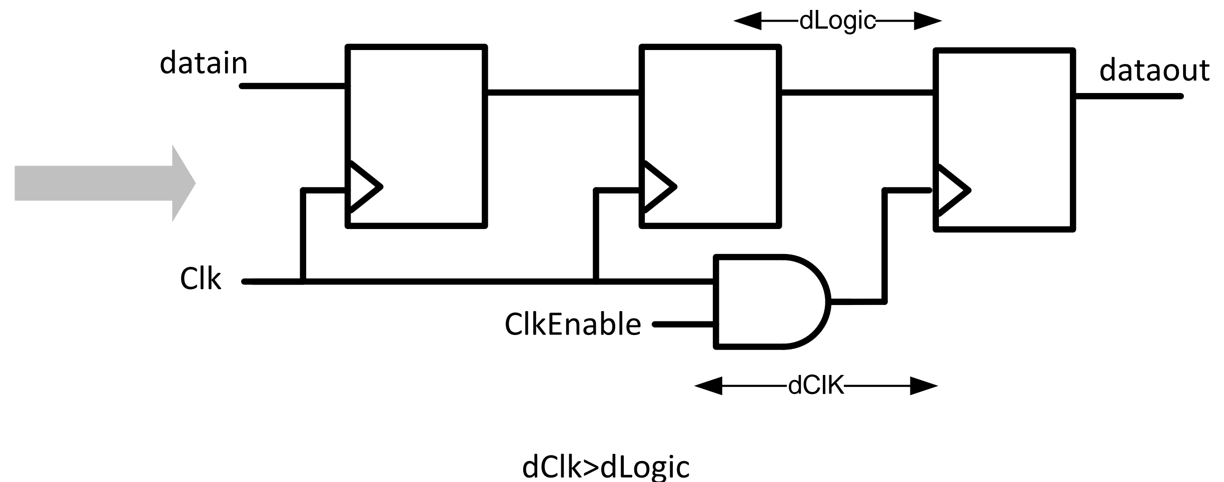
MUX adds delay to the data path



Power Reduction Techniques: Clock Issues (skew)

- ❑ Hold time violation in FPGAs is mainly because of the excessive delay on the clock
- ❑ Hold time violation b/c of low logic delay is rare due to the built-in delays of the logic blocks and routing resources

```
module clockgating(
output dataout,
input clk, datain,
input ClockEnable);
reg ff0, ff1, ff2;
wire clk1;
assign clk1 = clk & ClockEnable;
assign dataout = ff2;
always @(posedge clk)
ff0 <= datain;
always @(posedge clk)
ff1 <= ff0;
always @(posedge clk1)
ff2 <= ff1;
endmodule
```

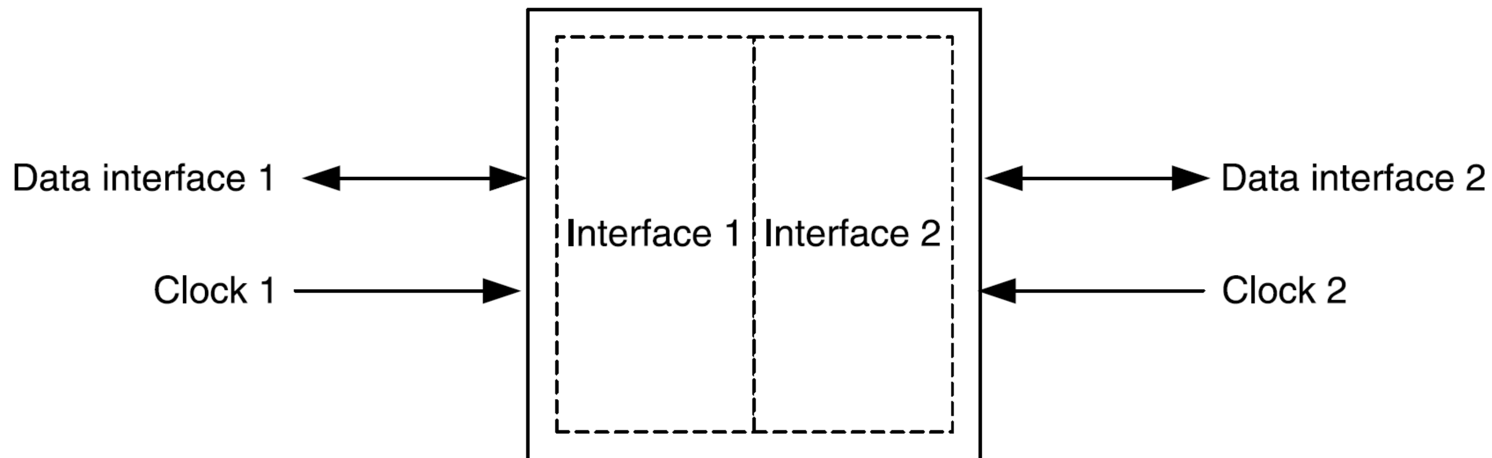


Clock Domain

- A clock domain is a section of logic where all synchronous elements (flip-flops, synchronous RAMs, pipelined multipliers) are clocked by the same net.

- If all flip-flops are clocked by a single global clock input to the FPGA, then there is one clock domain.

- Two clock domains for different interfaces:



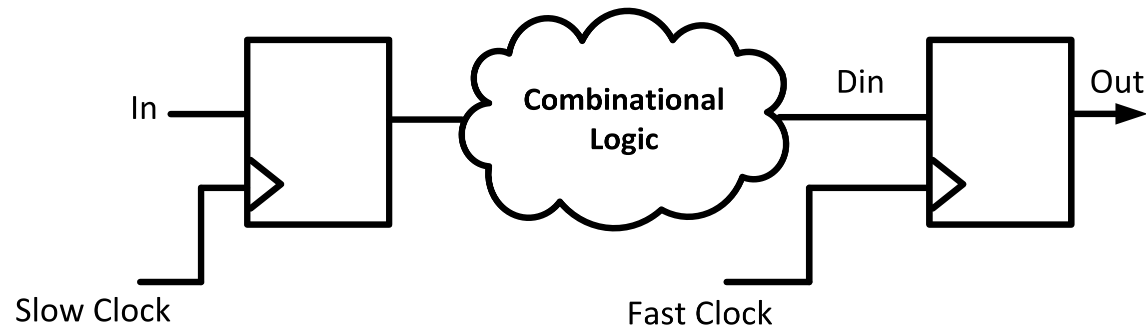
Crossing Clock Domains

- **Crossing clock domains**: passing signals between multiple clock domains
- Clock domain crossing can be a major source of problem b/c:

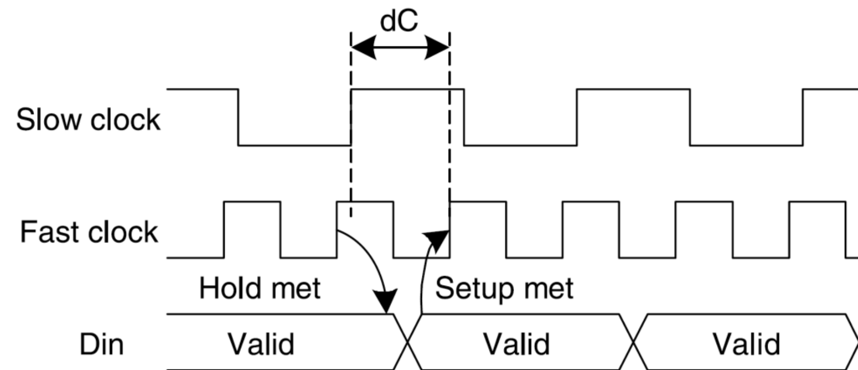
1. If there are two clock domains that are asynchronous, then failures are often related to the relative timing between the clock edges.
2. Problems will vary from technology to technology. Higher speed technologies with smaller setup and hold constraints will have statistically fewer problems
3. EDA tools typically do not detect these problems. Static timing analysis tools analyze timing based on individual clock zones
4. Cross-clock domain failures are difficult to detect and debug if they are not understood. It is very important that all inter-clock interfaces are well defined and handled before any implementation takes place.

Crossing Clock Domains : Issues

- Two clock domains
- Consider a **fine** case, which works properly

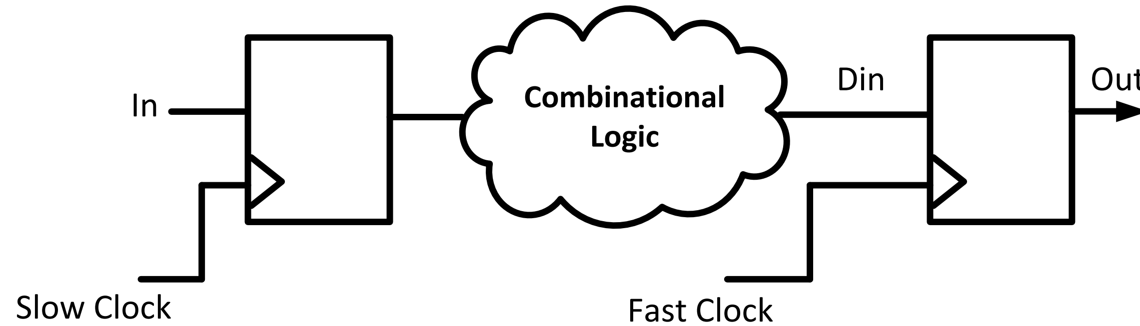


- Phase difference is such that both hold time and setup time constraints are met

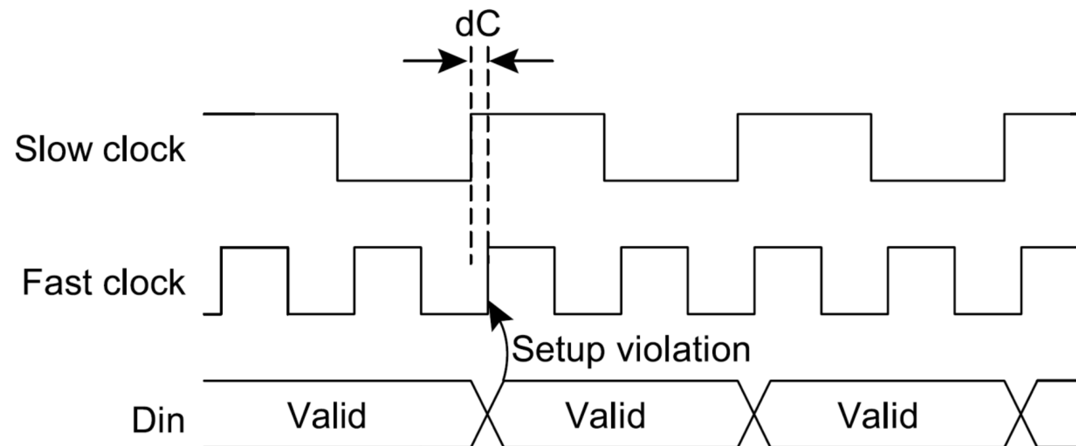


Crossing Clock Domains : Issues

- Consider a **problematic** case



- Phase difference is such that the setup time for the second FF is violated



Crossing Clock Domains: What happens with violation?

- A timing violation occurs when the data input to a flip-flop transitions within a window around the active clock edge as defined by the setup and hold times

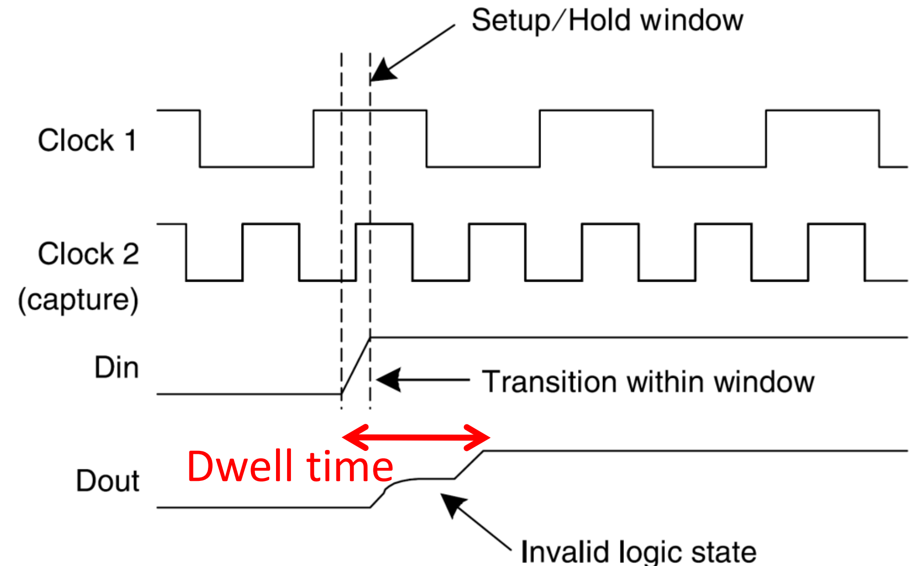
- This timing violation exists b/c if the setup and hold times are violated, a node within the flip-flop can become suspended at a voltage that is not valid for either a logic-0 or logic-1.

- Rather than saturating at a high or low voltage, the transistors may dwell at an intermediate voltage before settling on a valid level (which may or may not be the correct level).

This is called Metastability

Crossing Clock Domains: Metastability

- Data transition is not allowed within the setup/hold condition window
- The output may remain metastable for the entire clock period (functional failure)
- In an RTL simulation, no setup and hold violations occur and thus no signal will ever go metastable.
- Even harder to verify in the gate-level simulations.



- Undefined dwell time before the signal settles
- This dwell time adds to the path's propagation delay

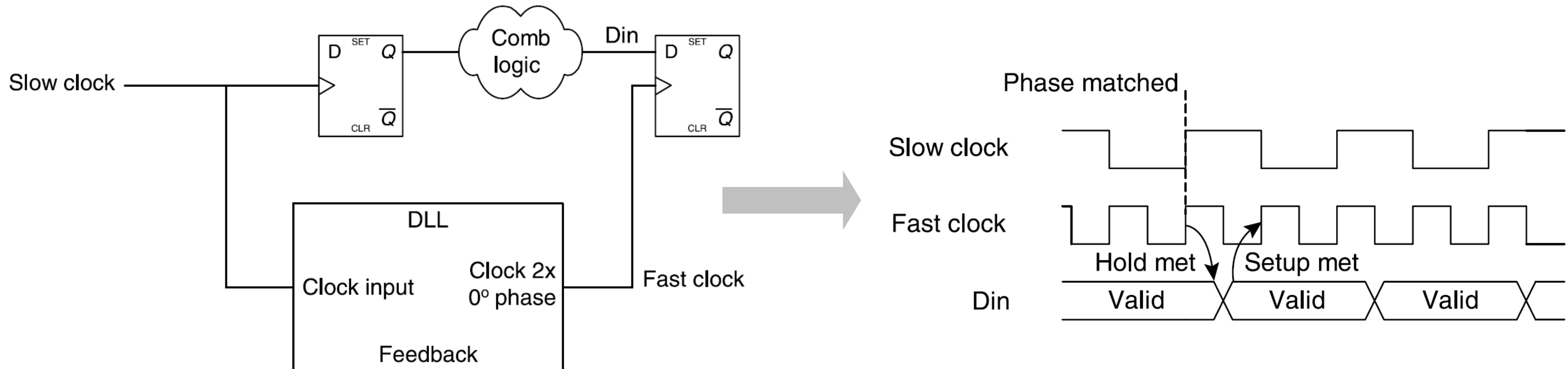
How to Avoid Metastability?

- To resolve the metastability problem, three main solutions may be used:
  - **Case I: the period of one clock is a multiple of the other**
    - Using Phase Locked Loop (PLL) or Delay Locked Loop (DLL)
  - **Case II: Otherwise**
    - Double Flopping (pass a single-bit signal b/w asynchronous clock domains)
    - Using a FIFO structure (pass multi-bit signals b/w asynchronous clocks)

Avoid Metastability Using PLL or DLL

- DLL adjusts the phase of the faster (capture) clock domain to match that of the slower (transmitting) clock domain.
- The total amount of time available for data to pass between the two domains is always at its maximum possible value.

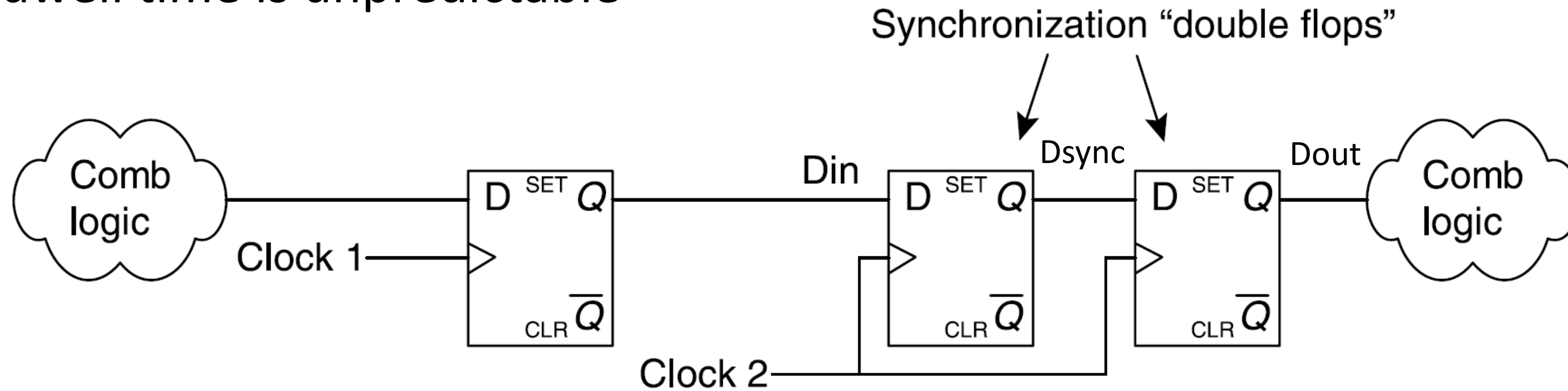
Condition: Propagation delay b/w FFs less than the period of the fast clock

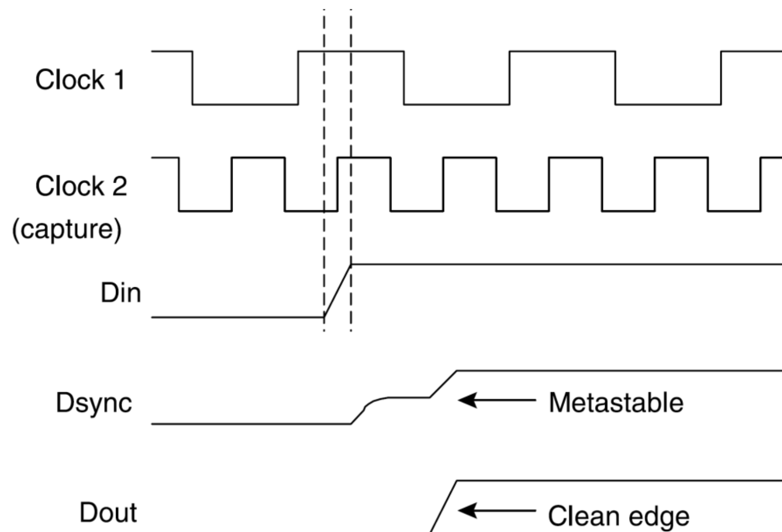


Phase adjustment in general not possible

Avoid Metastability by Double Flopping

- The dwell time before the signal settles adds to the path's propagation delay
- This dwell time is unpredictable



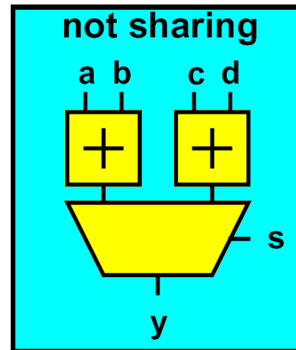


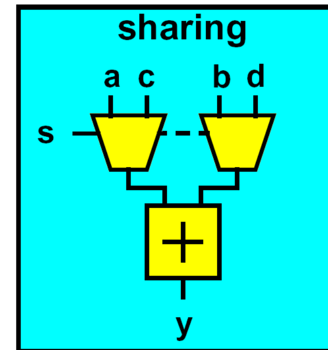
By adding no logic between the two FFs, we maximize the amount of time provided to the signal to settle.

Used to pass a single bit

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?

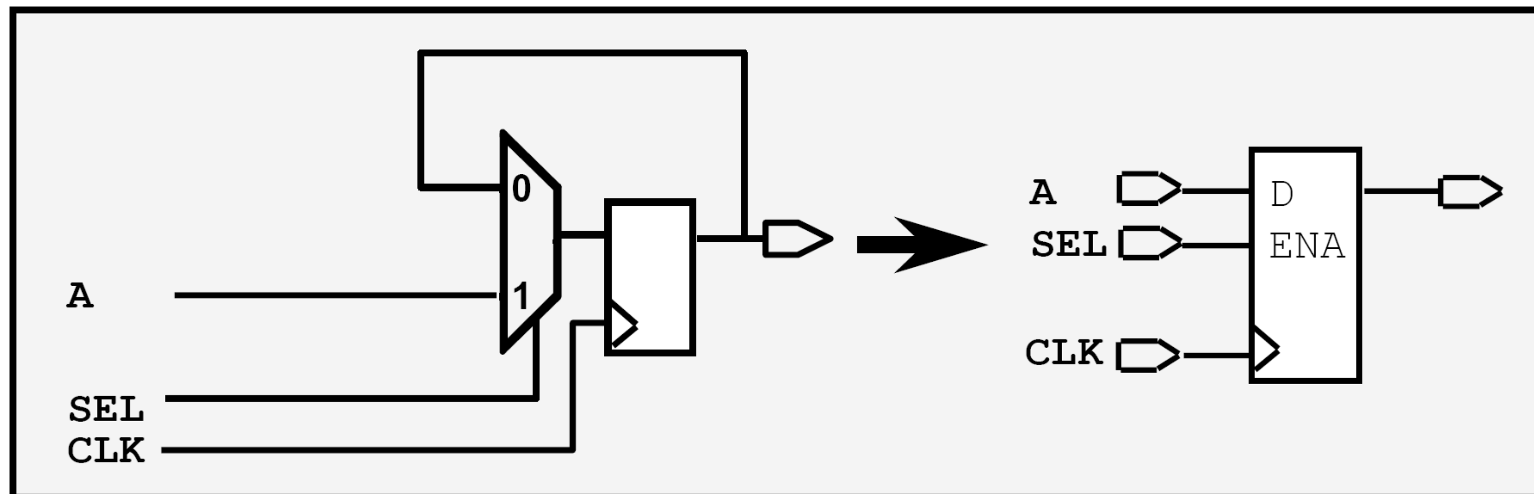
You can write your RTL code to do its own resource sharing!

Synthesis Tool Optimization (Target Library)

The process of using gates from the target library to generate a design that meets timing and area goals.

Synthesis Tool Optimization (Target Library)

- The process by which Synthesis tool maps to sequential cells from the technology library:
  - Tries to save speed and area by using a more complex sequential cell
  - May be slower but will reduce area



Synthesis in Design Corners

- Library standard cells are typically characterized under “normal” temperature and voltage
- Vendors allow for synthesis of circuits, which will not operate under “nominal” conditions by embedding operating condition models in the technology libraries