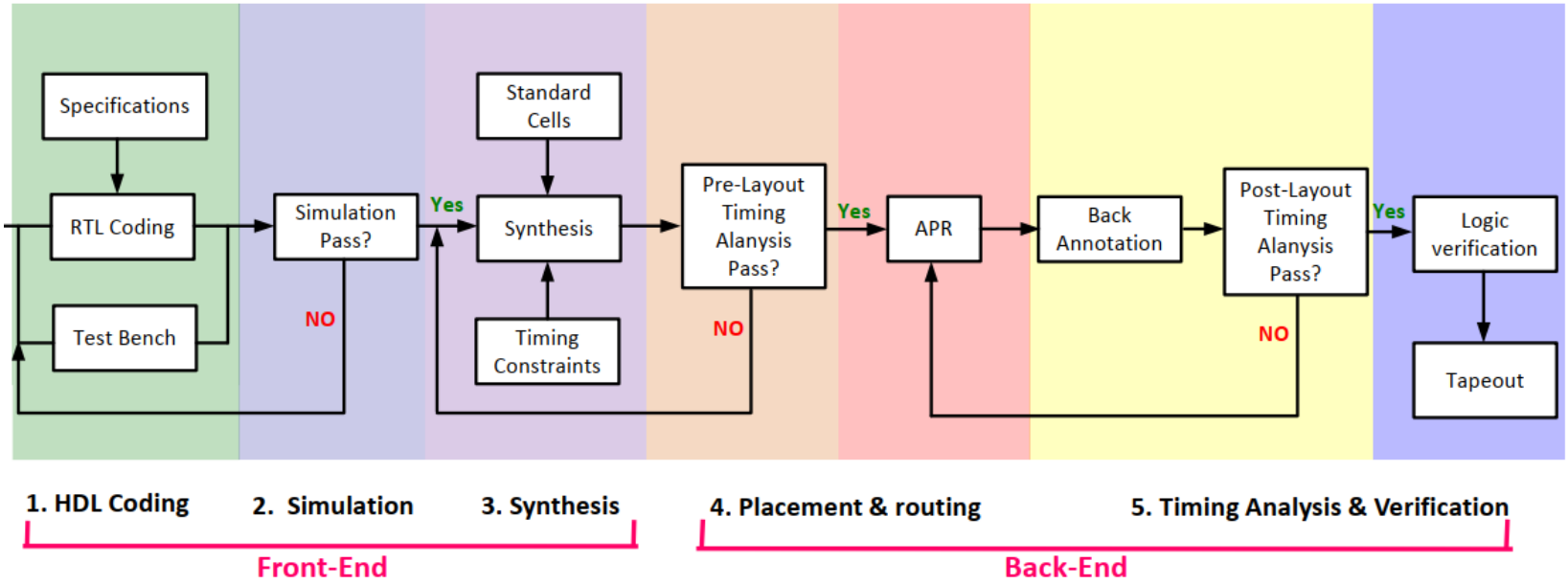
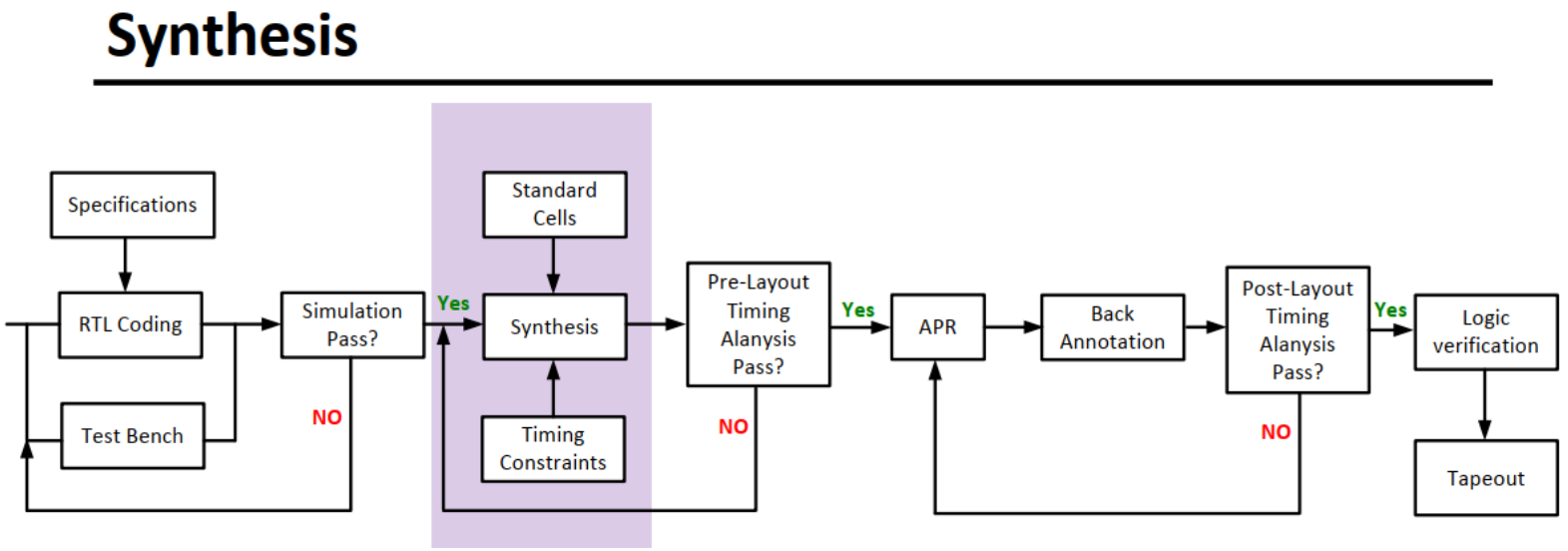
Chapter 7: ASIC/FPGA Chip Design
Verilog for Synthesis

Assoc. Prof. Truong Ngoc Son

ASIC / FPGA Design Flow



Synthesis



- 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

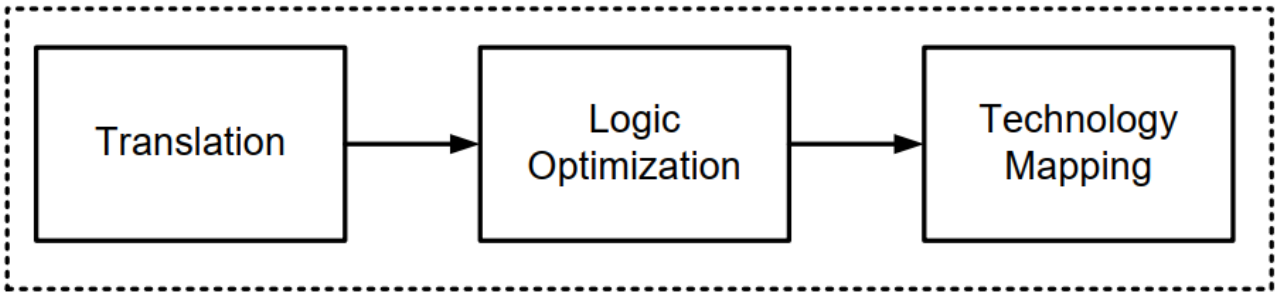
Synthesis

- 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

❑ **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
```

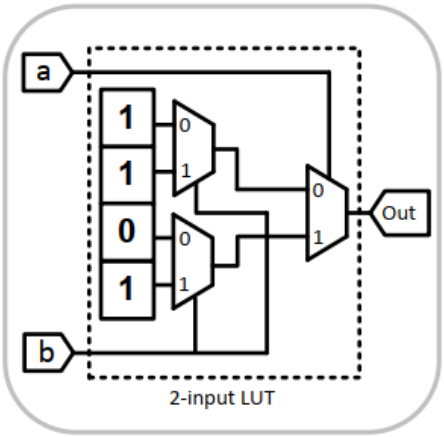
Translation

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

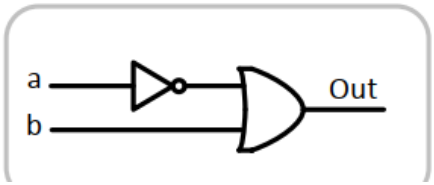
Logic Optimization

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

Technology Mapping



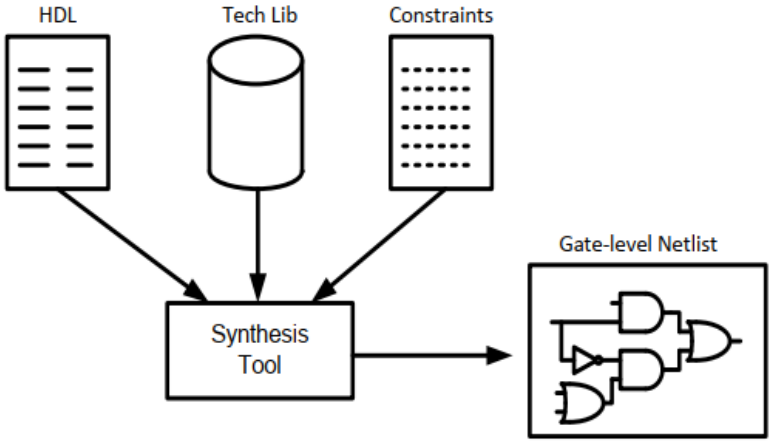
FPGA



ASIC

Synthesis Tools

❑ Synthesis tool:

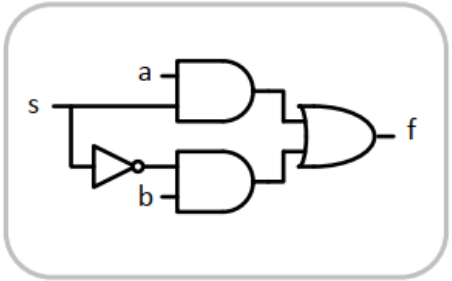


❖ **Example:** A 2-to-1 Multiplexer (2x1-MUX)

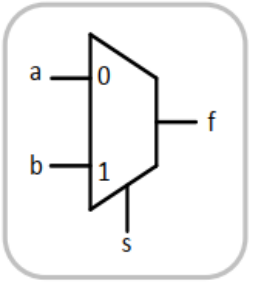
```
If (s==0)
   f = a;
else
   f = b;
```

Verilog code
(has to comply with certain structures)

Synthesis Tool



Synthesized gate-level



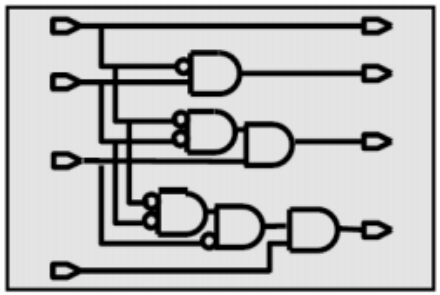
Schematic

Synthesis

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

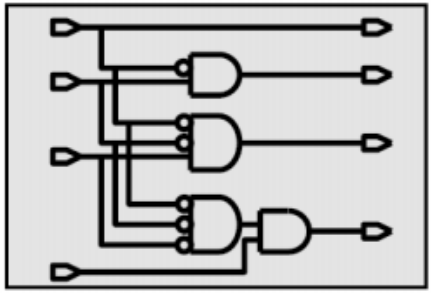
HDL Source

Translate (read)



Generic Boolean
(GTECH)

Optimize + Map
(compile)



Target Technology

Synthesis is Constraint-Driven

- 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

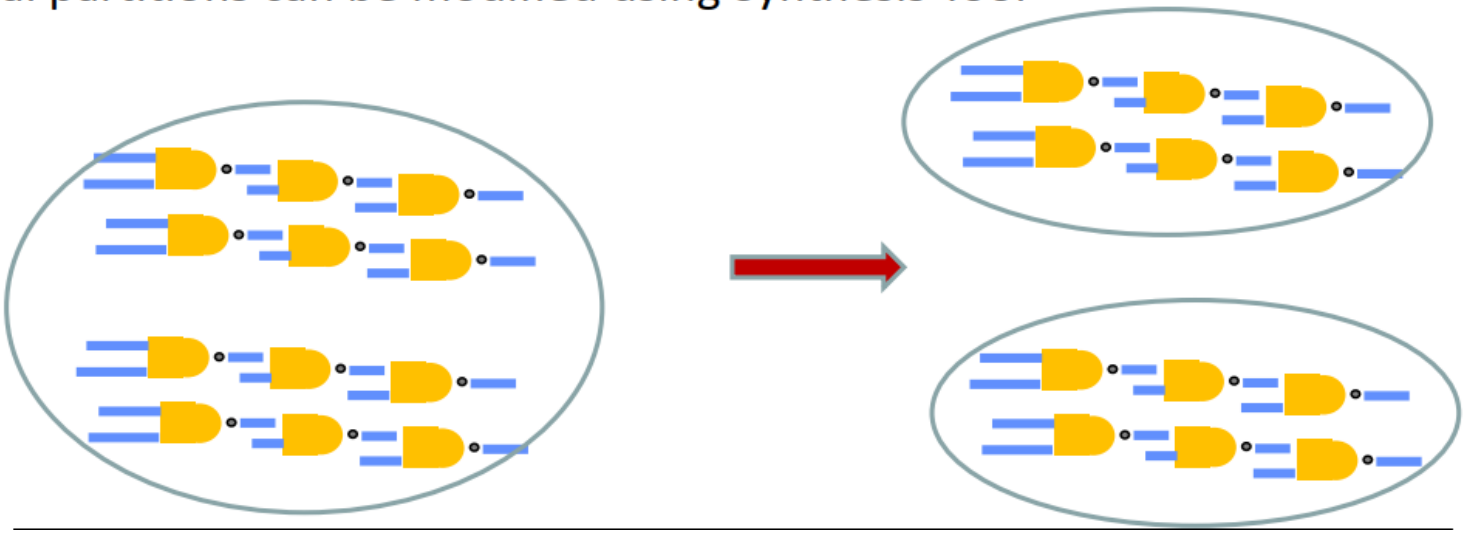
Synthesis Tools

- 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

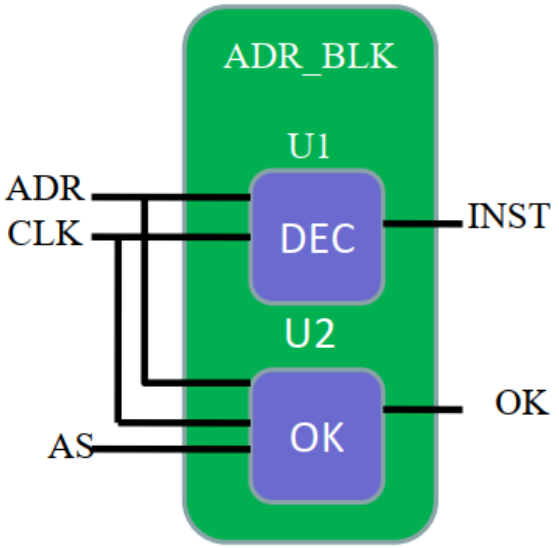
- Partition a design into different modules based on the functionality
  - Pros:
    - Separation of the cores that have different functionality
    - More manageability of smaller modules
    - Easier managements of a design implementation by a team
    - Focus to write optimized HDL code for each module
    - 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

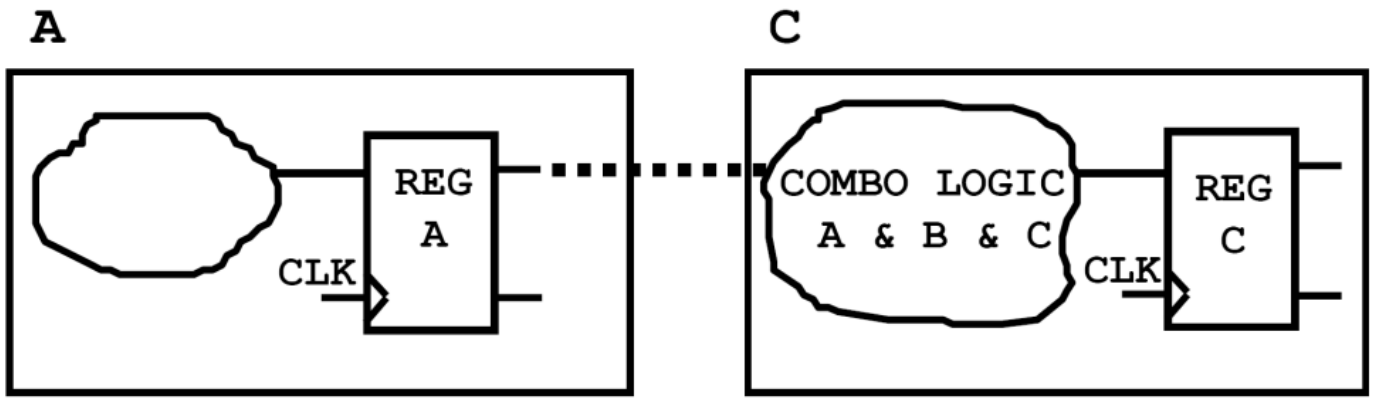
- module statement defines hierarchical blocks (partitions):
  - Instantiation of an entity or module creates a new level of hierarchy
- Inference of Arithmetic Circuits (+, -, *) can create a new level of hierarchy
- Always statements do not create hierarchy

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



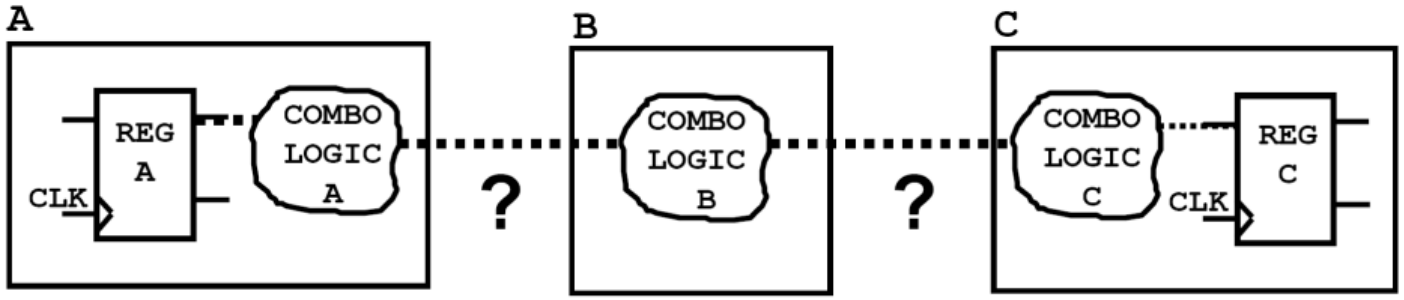
Good Partitioning (Partition at Register Boundaries)

- 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

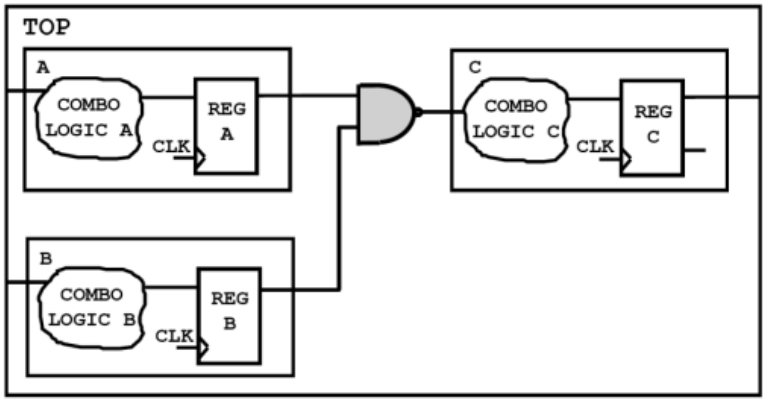


Poor Partitioning (Partition at Combinational Logic)

- 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!

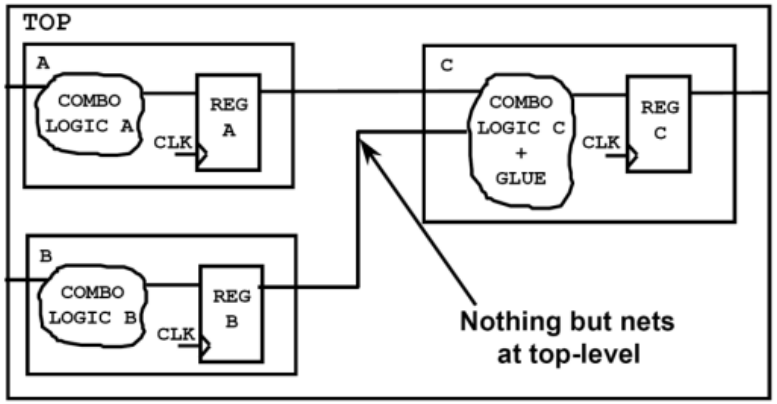


Good Partitioning (Avoid Glue Logic)



Poor Partitioning

- 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



Good Partitioning

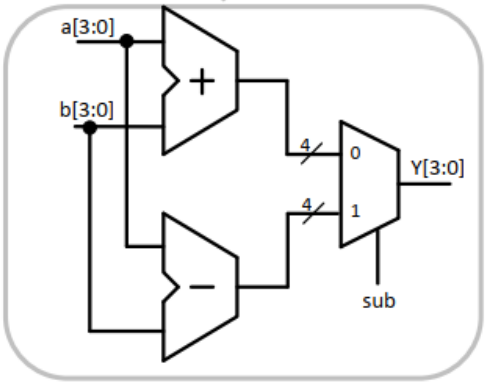
- The glue logic can now be optimized with other logic
- Top-level design is only a structural netlist, it does not need to be compiled

HDL for Synthesis

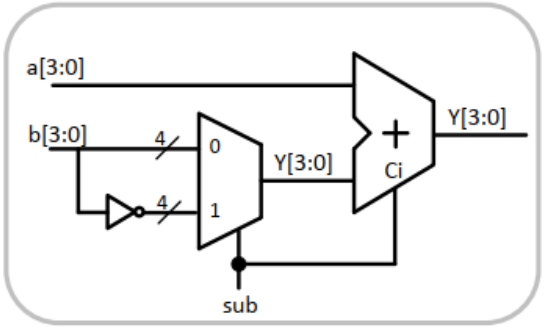
- "Bad" HDL code does not allow efficient optimization during synthesis
  - Garbage in, garbage out!
- Logic synthesizer doesn't do magic! designer has to take some responsibility in coding.

❖ 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}
```

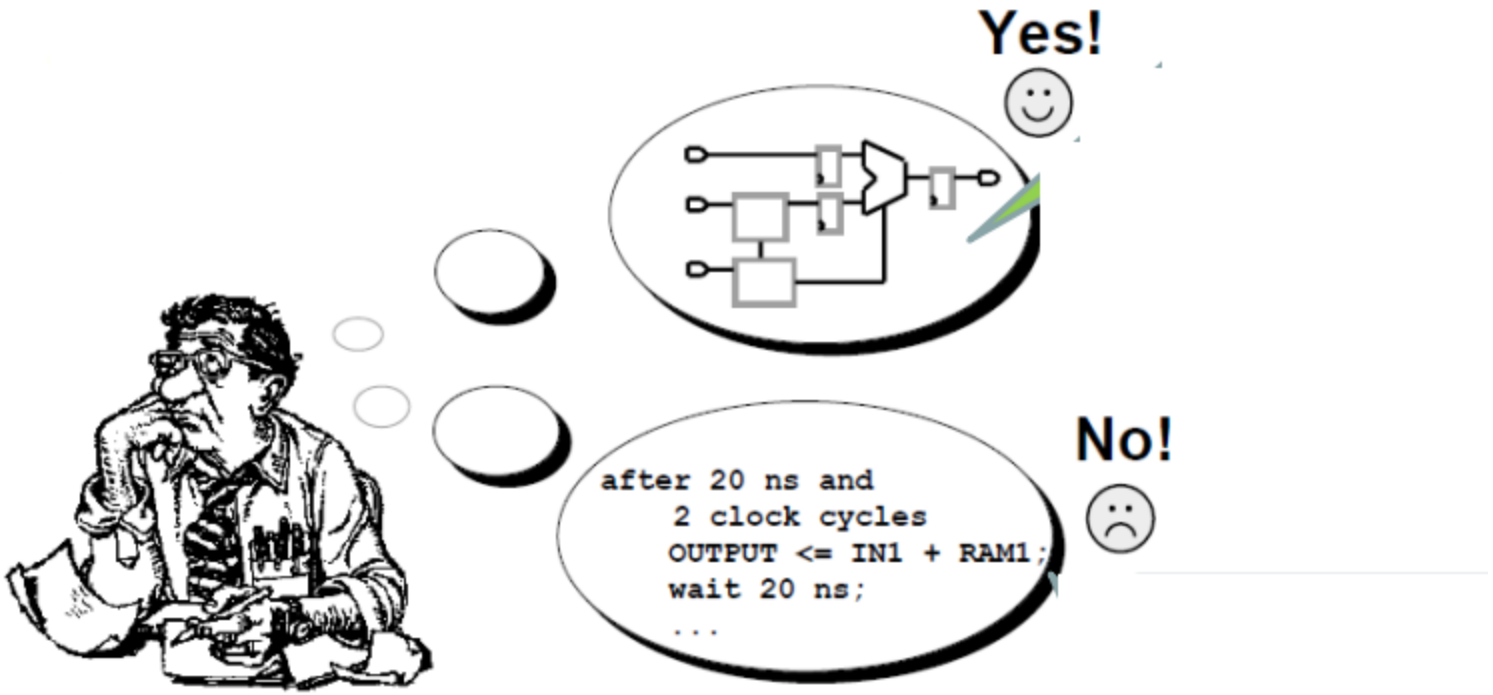


More efficient

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)



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

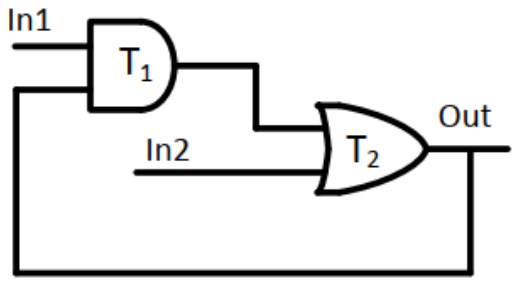
Non-Synthesizable Constructs

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

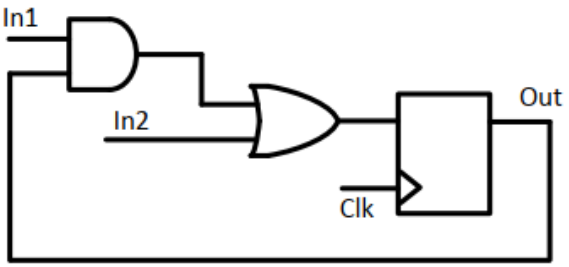
Design for Synthesis (No Timing Loop)

- 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.

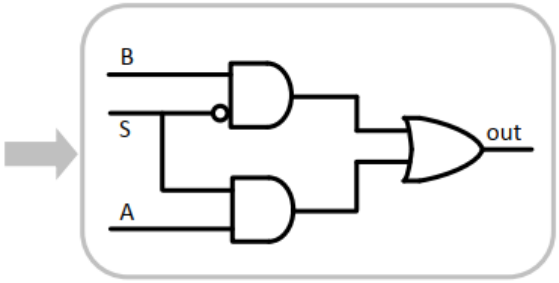
Avoid Latch Inference in If-else Statements

❖Example:

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

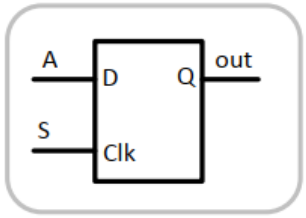
1

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



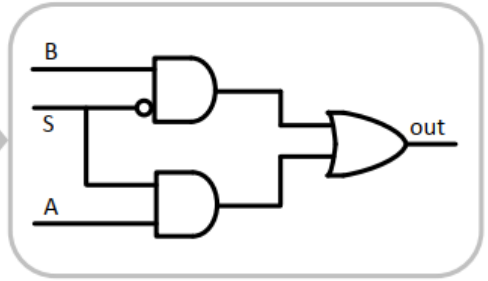
No Latch

2



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

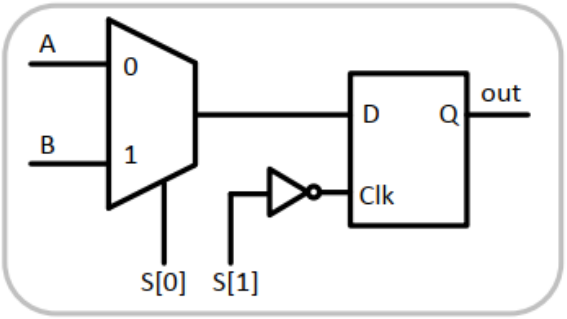
Avoid Latch Inference in Case Statements

❖Example:

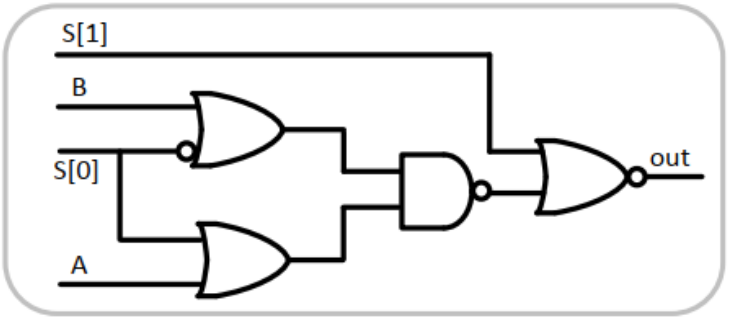
```
module DUT (A, B, S, out);
input A, B;
Input [1:0] S;
output reg out;

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

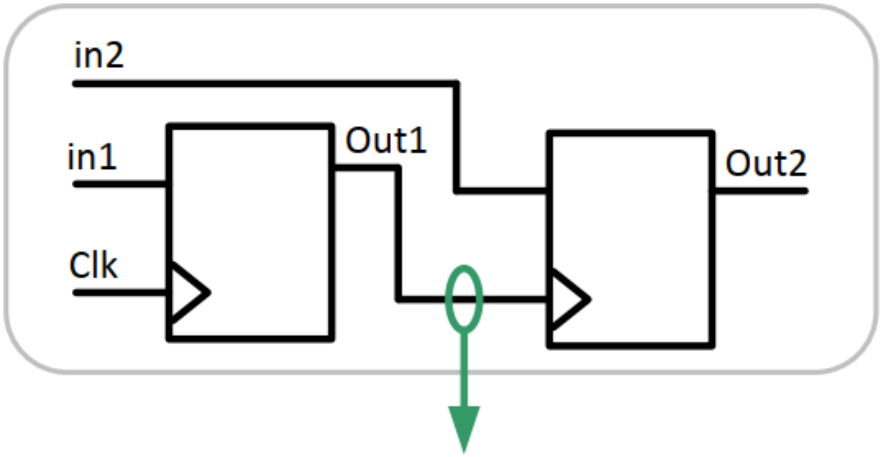
- Clock is the most important signal in the design (golden)
- Why is it different from other signals?
  - It is a global signal, i.e., it is routed across all modules in the design
- Treat clock as a golden signal
  - No buffering should be done on clock during coding and synthesis
    - Clock buffering 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

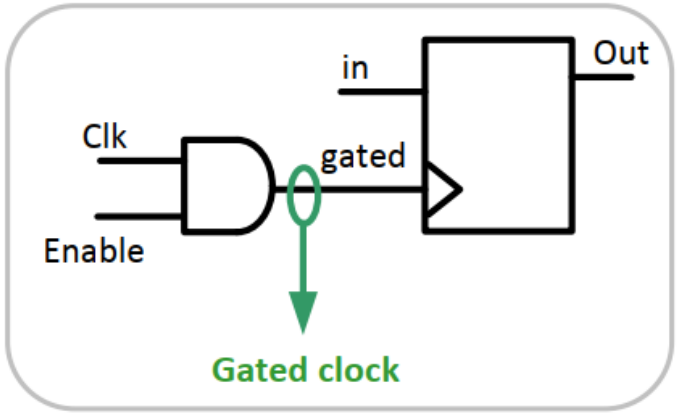
Clock (No Gating)

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!

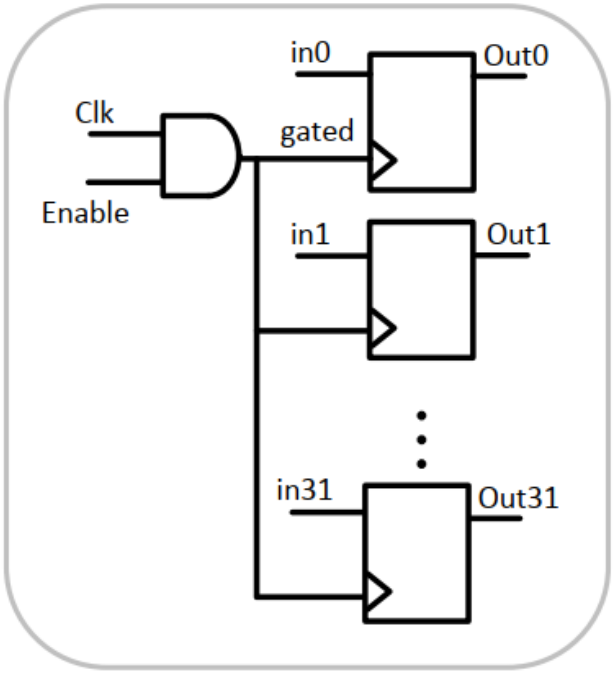
```
assign  gated = Clk & Enable

always @ (posedge gated)
                Out <= In;
```



Clock (No Gating)

- If clock gating is used avoid large fan-outs



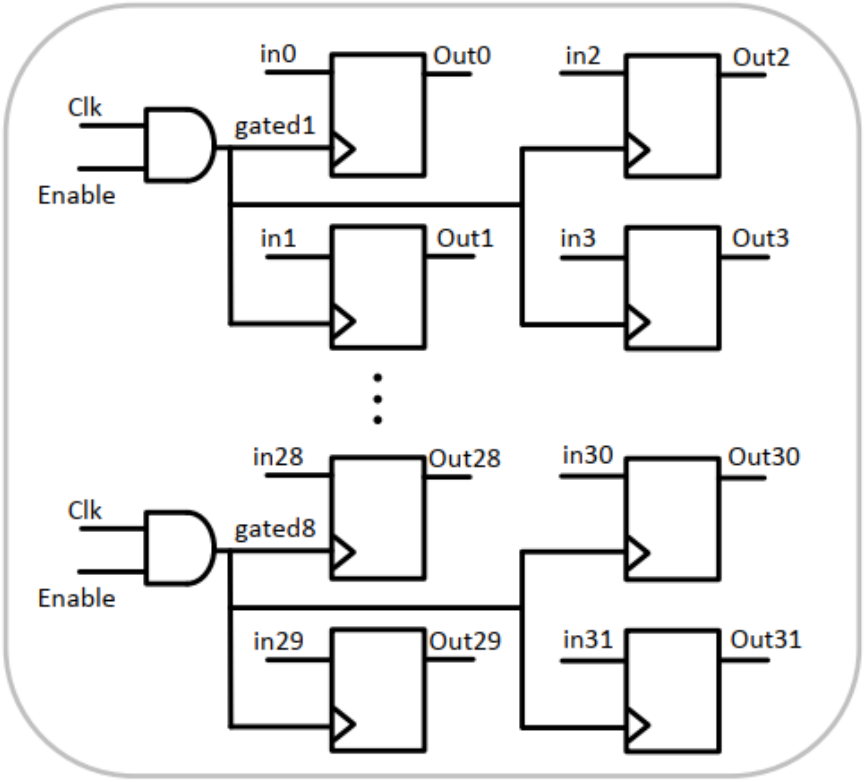

```
assign gated = Clk & Enable;

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:



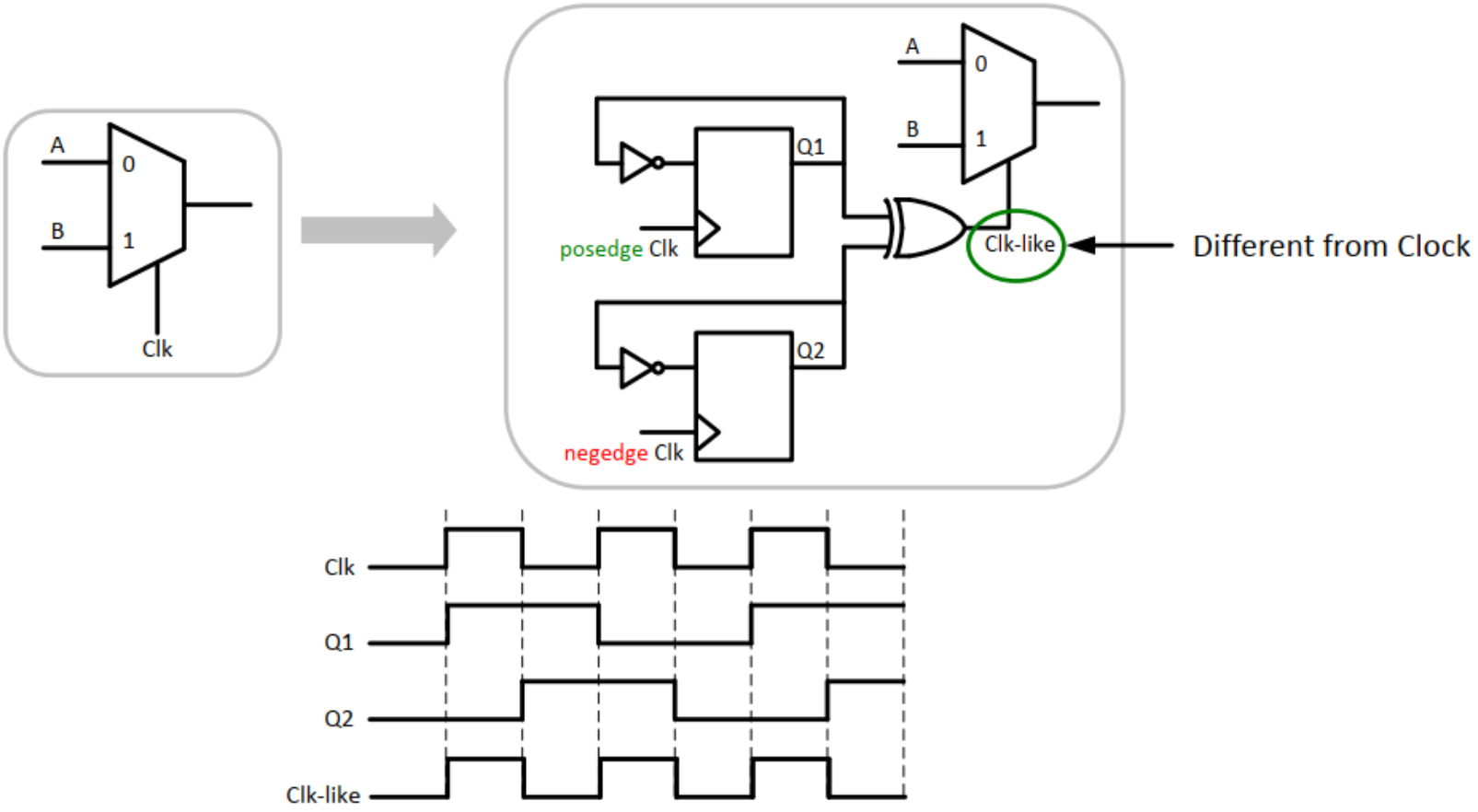
Explicit "and" instantiation

```
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];
```

Clock

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

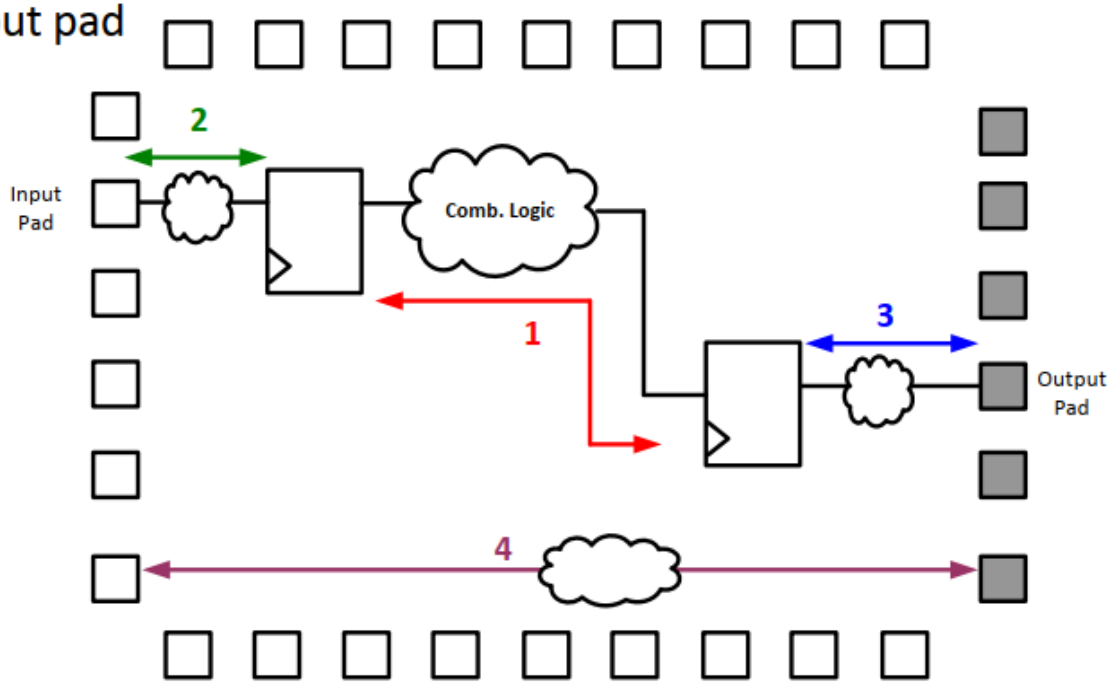


Architectural Techniques : Critical Path

- 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



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

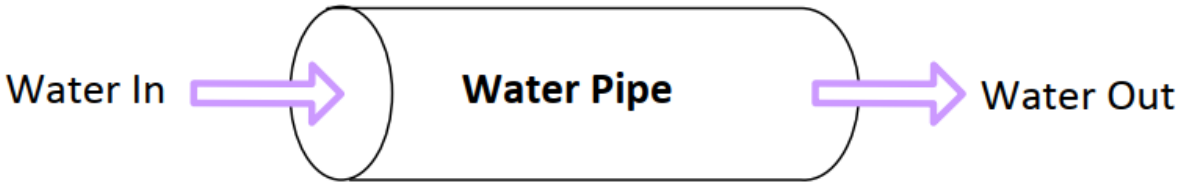
- Maximum Clock Frequency:

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

- T_{clk-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.