RAMAIAH
SKILL ACADEMY

Optimization

Vasudev Murthy

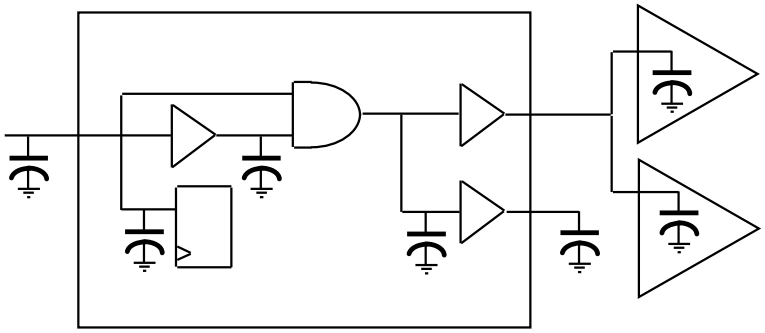
External Interfaces

- A design comprises of many sub-blocks
- Many teams work on the same design at different times
- Sub-block should not only meets it's specifications but also that of the top block

- The output port is assumed to have 0 load
  - Leads to wrong timing analysis
  - A inferior driver chosen for driving the output port
- The input port is assumed to have infinite drive strength
  - Leads to incorrect timing analysis
  - Port may not be capable of physically driving the connected gates

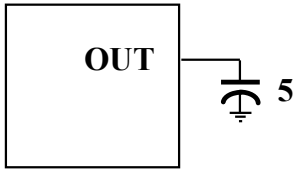
Modelling Ports

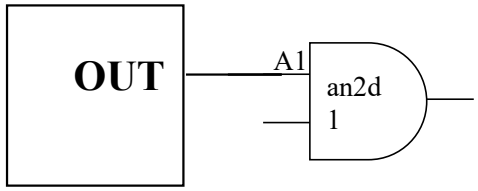
- The load at the output port has to be appropriately modeled for system to work efficiently
- In order to accurately calculate the timing of an output circuit, the total capacitance driven by the output cells need to be known



Modelling Ports

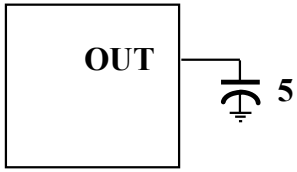
- '*Set_load*' specifies the capacitive load on ports
  - A constant value can be specified
  - Load can be specified as a function of input capacitance of the cell which is being driven by the output port
- A specific capacitance value can be specified
- A relative capacitance value can be specified



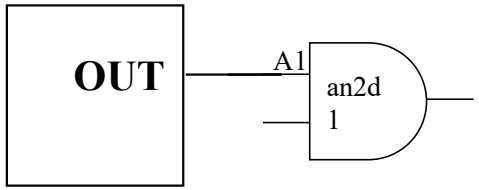


Output Capacitance

- set_load 5 [get_ports OUT]
  - Specifies a capacitive load of 5 units on output port ‘OUT’
  - An ad-hoc approach
  - The value specified may not be consistent with library parameters

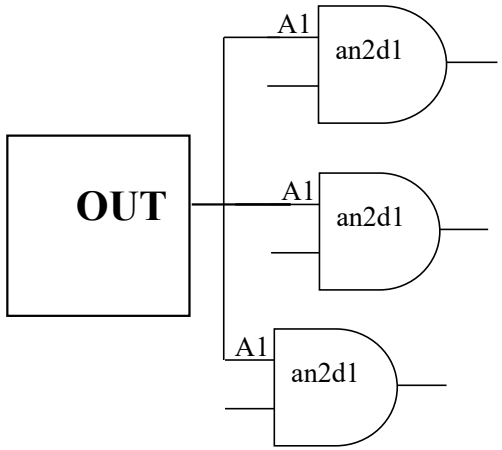


- set_load [load_of tcb013ghplvtwc/an2d1/A1] [get_ports OUT]
  - Specifies the capacitance of pin A1 of cell an2d1 in library tcb013ghplvtwc as the load on port ‘OUT’

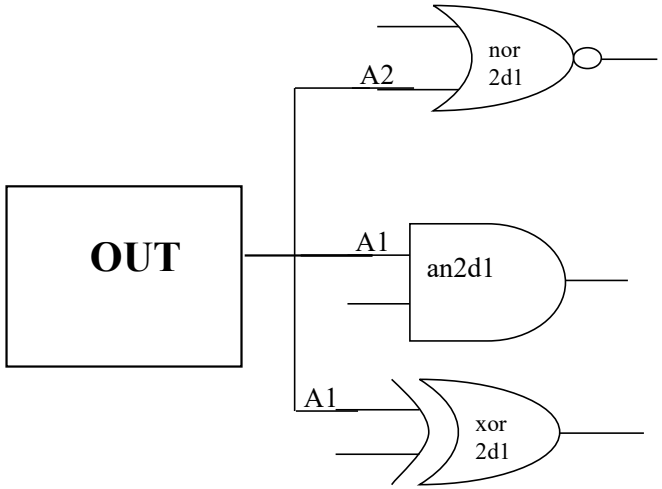


Output Capacitance

set_load {3 * [load_of tcb013ghplvtwc/an2d1/A1]} [get_ports OUT]



set_load { [load_of tcb013ghplvtwc/an2d1/A1] + [load_of tcb013ghplvtwc/nor2d1/A2] + [load_of tcb013ghplvtwc/xor2d1/A1]} [get_ports OUT]



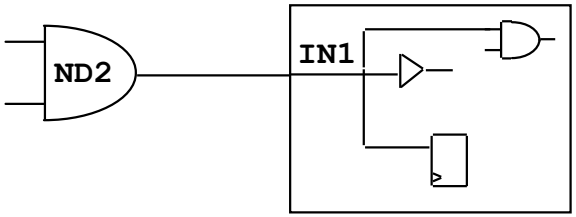
An accurate capacitance value can be specified

Fanout of Port

- The delay of the output port and the gate which is driving the port depends on the fanout of the port
- Fanout of output port is typically unknown
- Needs to be modeled for correct analysis
- set_fanout_load specifies the fanout on output port
  - Just a number which is related to fanout_load attributes, of input pin of gates driven by output port
  - set_fanout_load 6 [get_ports OUT]
  - Has no explicit physical relevance

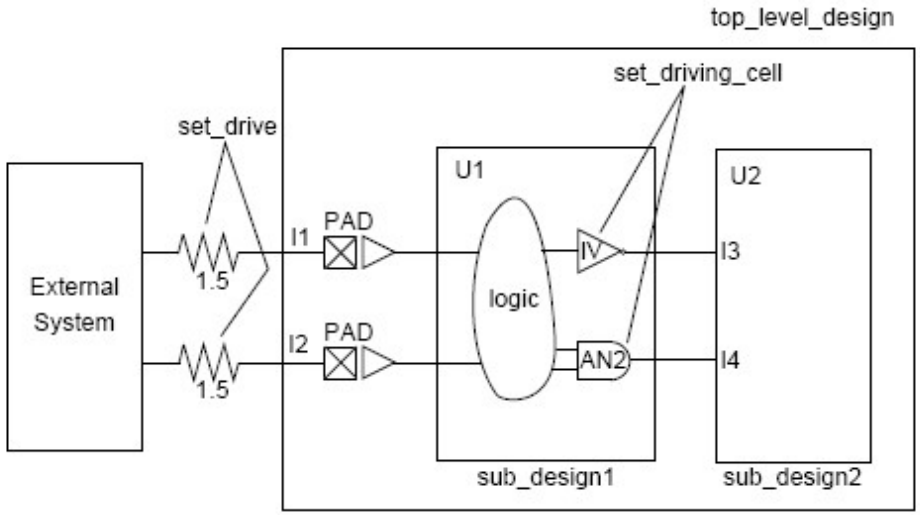
Input Port

- The drive resistance at the input port has to be accurately modeled
- By default 0 drive resistance assumed which leads to erroneous timing calculations



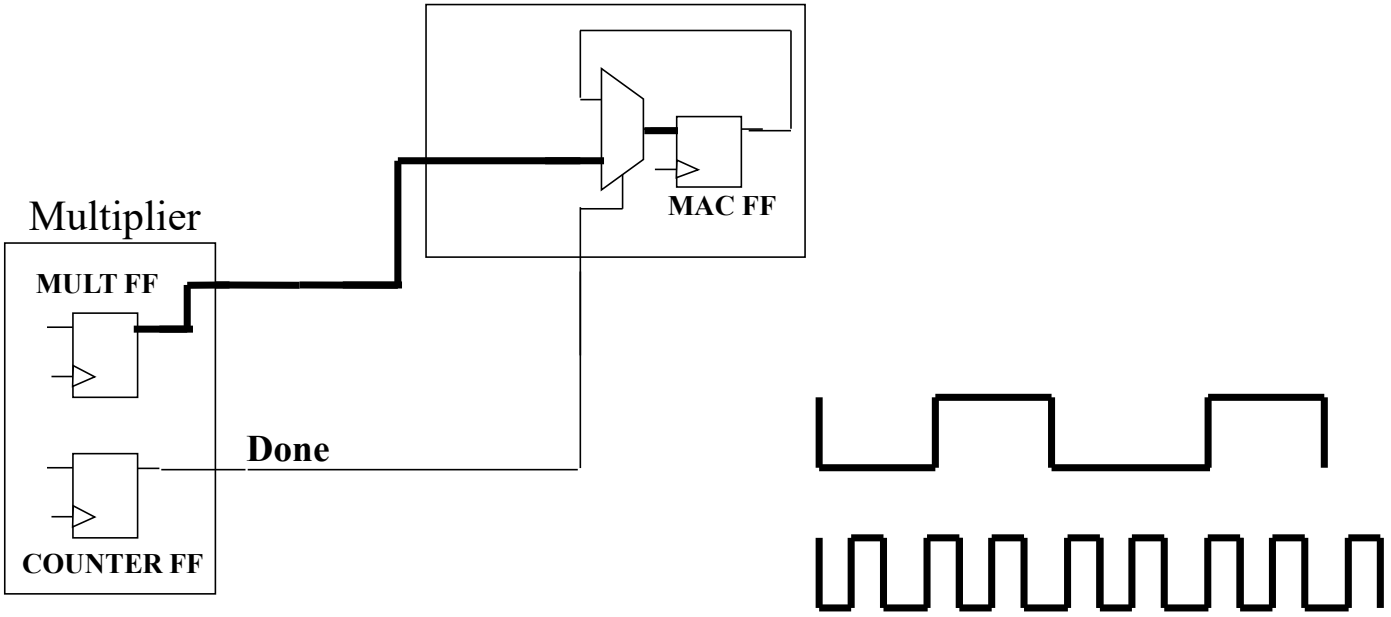
- '*set_drive*' specifies the drive resistance at input port
  - Either constant value can be specified
  - Or relative value of driving cell can be specified

Input Drive Resistance



- A specific input drive resistance can be specified
  - set_drive 1.5 [get_ports I1, I2]
- A relative drive resistance can be specified
  - set_driving_cell tcb013ghplvtwc/IV [get_ports I3]

Multi Cycle Path



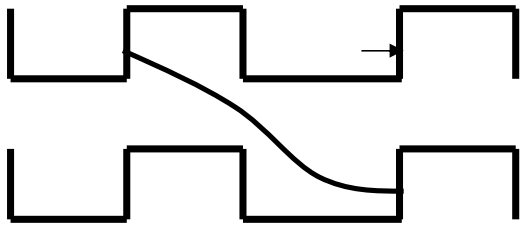
- By default every path in a design is assumed to be a single cycle path
  - Data is allowed to travel from start point to end point within one clock cycle

Multi Cycle Path

- There can be certain paths which consume more than one clock cycle
- A path in which data logically (functionally), takes more than one clock cycle to travel from start to end point is known as MultiCycle path
- The flip-flop which is supposed to capture data is physically stopped from doing so by some control signal
- By definition all paths are single cycle paths
- MultiCycle paths are due to architecture and functional issues which need to be taken care of
- A MultiCycle path should be explicitly set so that wrong timing analysis and unnecessary optimizations are not carried out

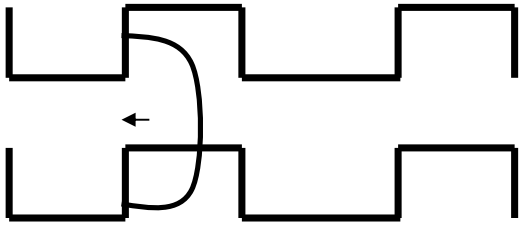
Setup - Hold

- Setup is checked one clock cycle after launch



(1)

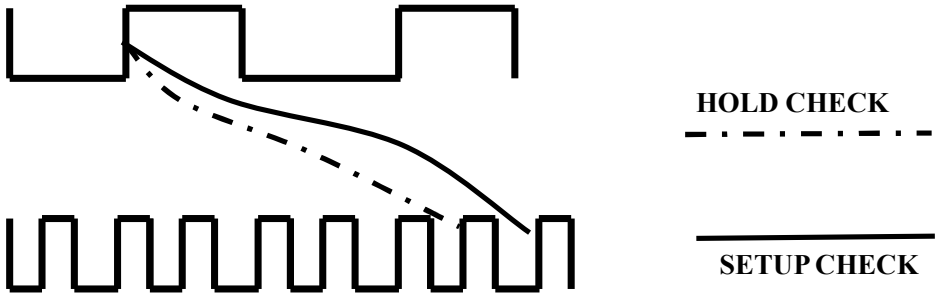
- Hold is checked immediately after launch



(2)

- From 1, and 2 it can be deduced that hold is checked one clock cycle before setup

Hold in MCP



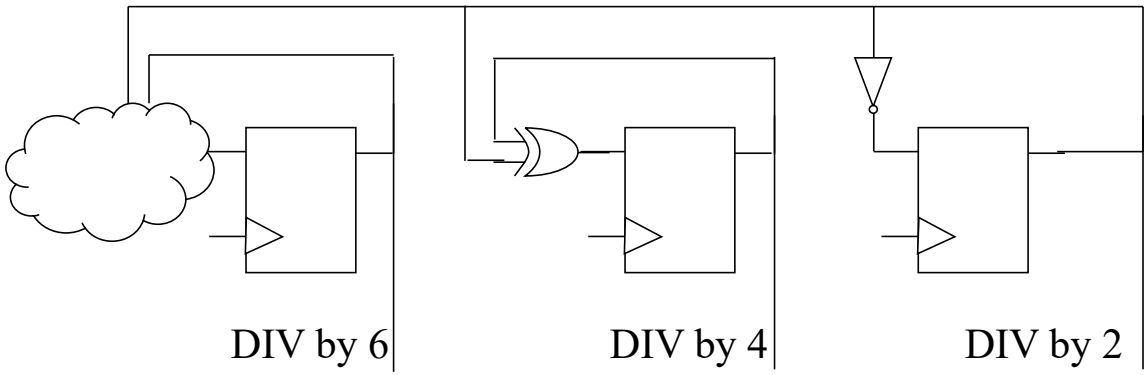
- During analysis of MultiCycle paths setup check will be pushed by the number of clock cycles specified – say ‘x’
- Simultaneously hold checks of MultiCycle paths, are unnecessarily pushed forward
- Hold check should be done at the same instant as launch
  - set_multicycle_path –setup 4 –from mult_reg –to mac_reg
  - set_multicycle_path –hold 3 –from mult_reg –to mac_reg

Generated Clocks

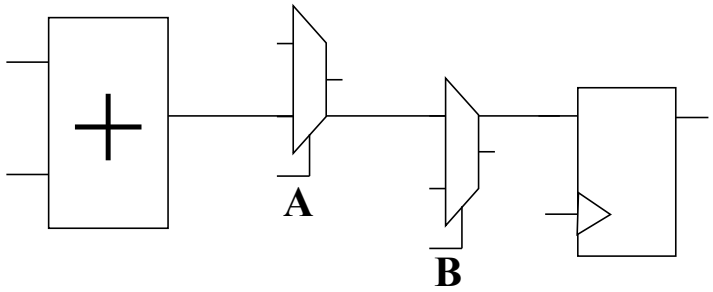
- Commonly designs have more than one clocks
- Clock can be from oscillator / PLL
- Clock is internally generated by several means
- Clock generation is logical in nature and generated clocks are not recognized as clocks by default by the tool
- Generated clocks have to be explicitly defined

Generated Clocks

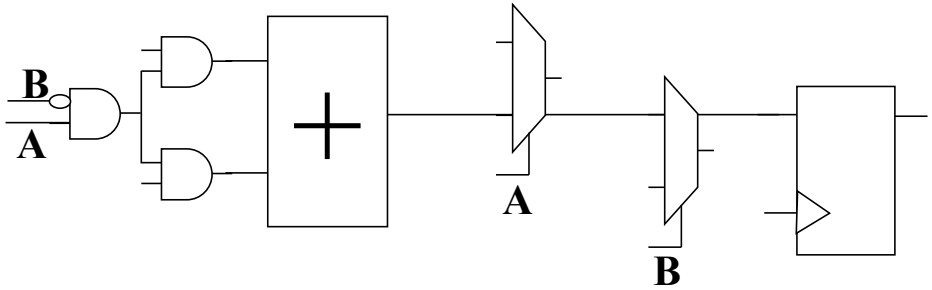
- Generated clocks have a source clock
- The child clocks are related to the master by some factor
  - create_generated_clock –source clock –divide_by 2 –name div_by_2_clock [get_pins counter_reg[0]/Q]



Operand Isolation

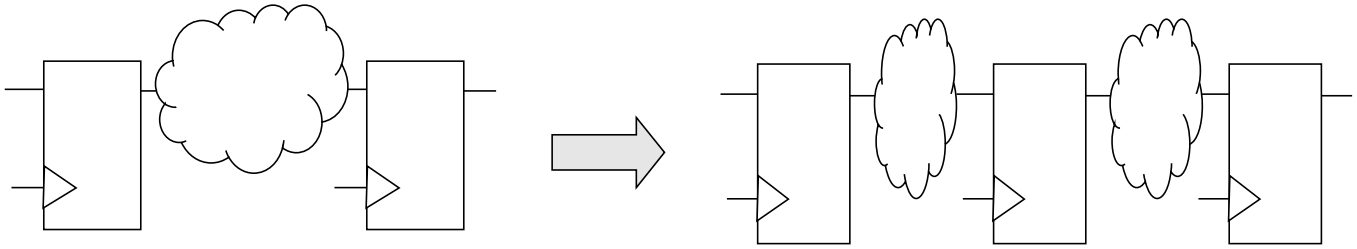


- Most designs are datapath intensive
- Computations in the data path contributes to a large portion of dynamic power dissipation
- Unnecessary computations need to be eliminated



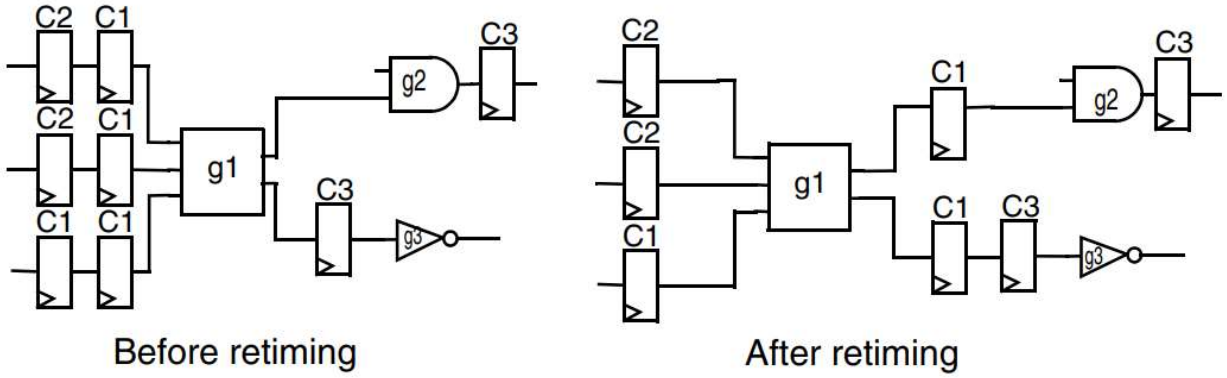
Pipelining

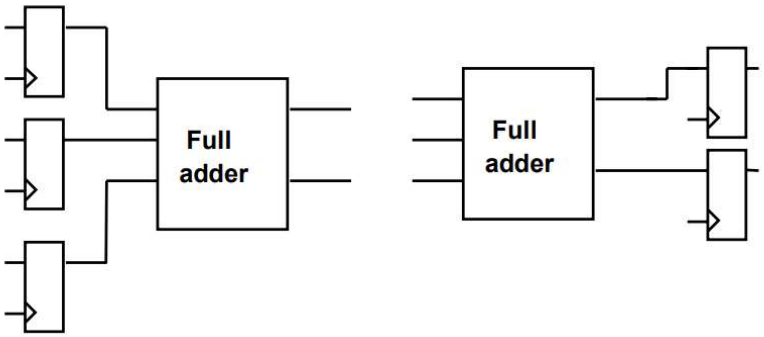
- The main objective of synthesis being to generate a netlist which works at the maximum possible speed there exists a number of techniques to increase the clock frequency



- One of the most common is pipelining
  - Involves introduction of flip-flops at strategic places
  - Increase the overall latency of the design
  - Increases the hardware and power though insignificantly
    - *optimize_registers*
    - *balance_registers*

Pipelining


C2
C1
C2
C1
C1
C1
g1
g2
C3
C3
g3
C2
C2
C1
g1
C1
C3
C1
g2
C3
g3
Before retiming
After retiming

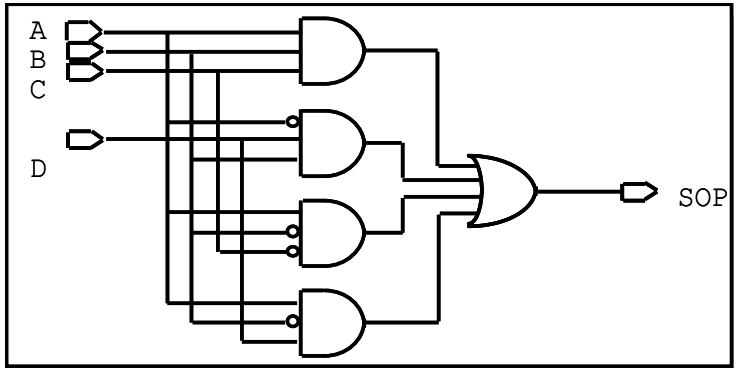

Full adder
Full adder

Opimization

- There are mainly two kinds of optimization
- Two Level Optimization
  - Optimization of combinational logic circuits modeled by two level SOP / POS expression forms
  - A representation for a multiple-output Boolean function is found that is optimum
  - Carried out using K Maps, ESPRESSO, Quine-McCluskey algorithms
  - Suitable for PLDs and PLAs
  - Area is reduced though delay may not be reduced
  - Called as flattening

Flattening

- Flattening is the process of the reduction of combinational logic paths to a two-level SOP form
  - Does not collapse design hierarchy
  - Useful for balancing of delays
  - May be area-intensive
  - May or may not reduce delay (based on size of product term)



Multi-level Opimization

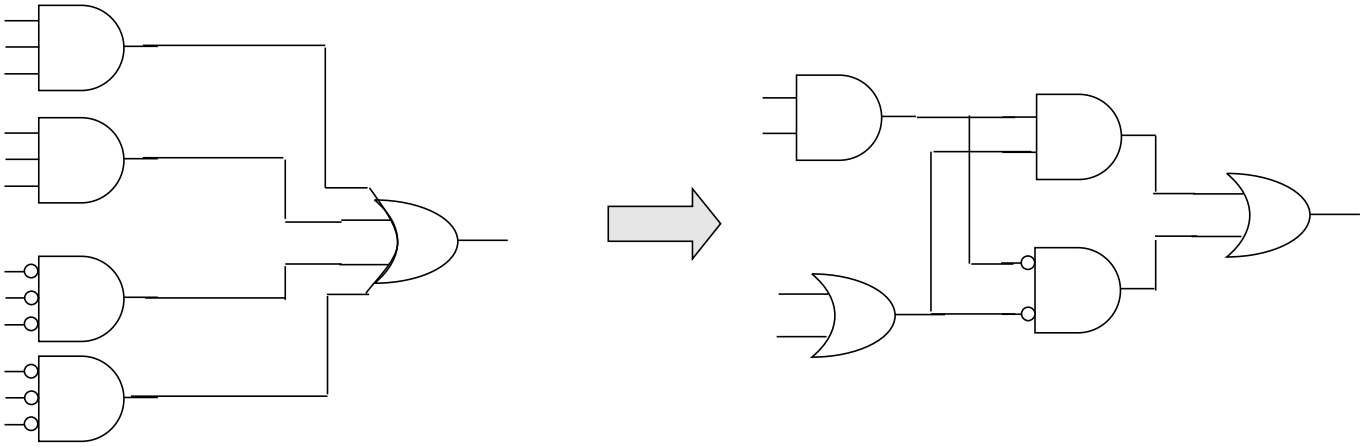
- Transform two level logic to multi level logic
- The use of intermediate terms to create a multilevel implementation of a design that satisfies almost all constraints
- Known as ‘Structuring’
- The key to all the multi-level optimizations is to identify the divisor of the boolean functions
  - F = D Q + R (Divisor * Quotient + Remainder)
  - Example

    F = AD + BCD + E

    F = (A + B) [(A + C) D] + E

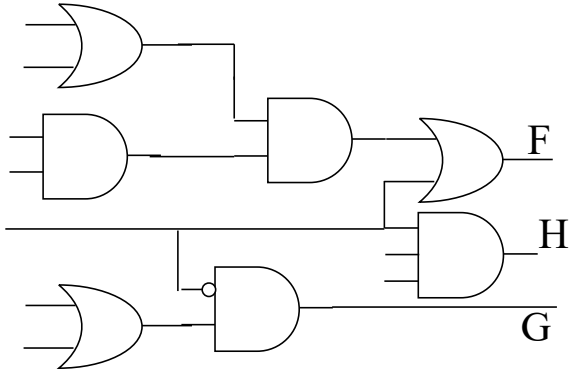
Decomposition

- A single Boolean expression is replaced with a set of new expressions
  - F = A B C + A B D + A' C' D' + B' C'D'
  - F = X Y + X'Y'
  - Where X = A B , Y = C + D

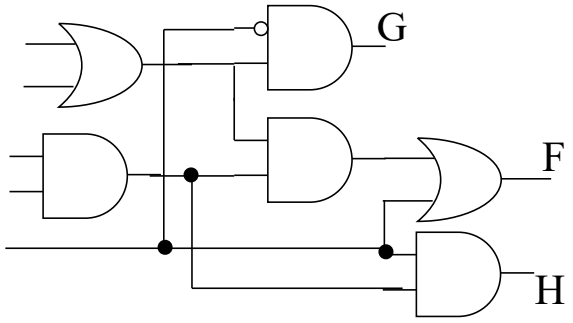


Extraction

- The common intermediate sub-functions of many Boolean functions are factored out
  - F = (A + B) C D + E
  - G = (A + B) E’
  - H = C D E

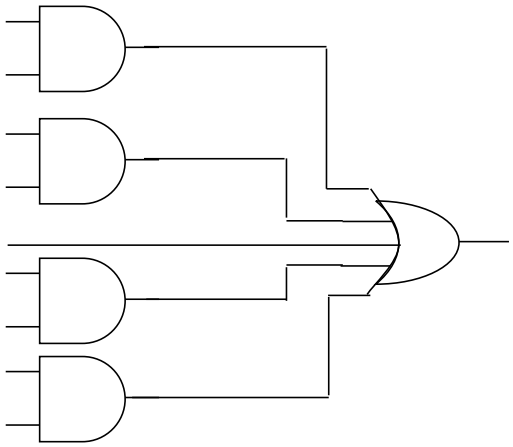


- Re-written as
  - F = X Y + E
  - G = X E’
  - H = Y E
  - X = A + B
  - Y = C D

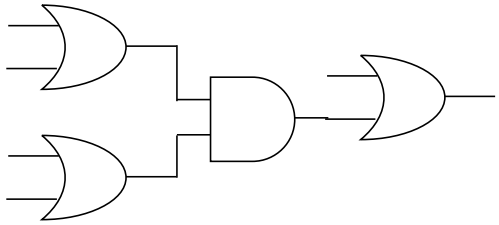


Factoring

- A two level expression re-expressed in multi-level form
- F = AC + AD + BC + BD + E
  - 2 levels of logic
  - 5 Gates
  - 1, 3 input Gate



- F = (A + B) (C + D) + E
  - 3 levels of logic
  - 4, 2 input Gates

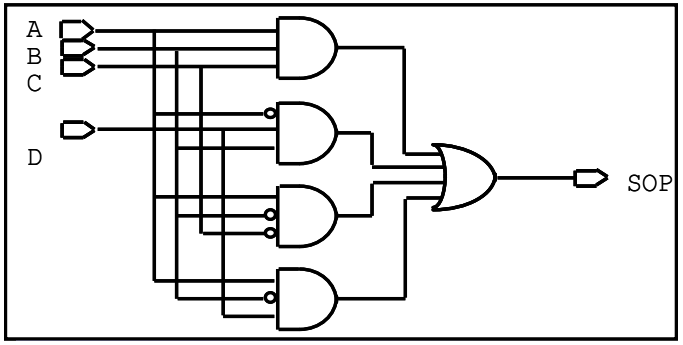


Other Multi-level Opimization

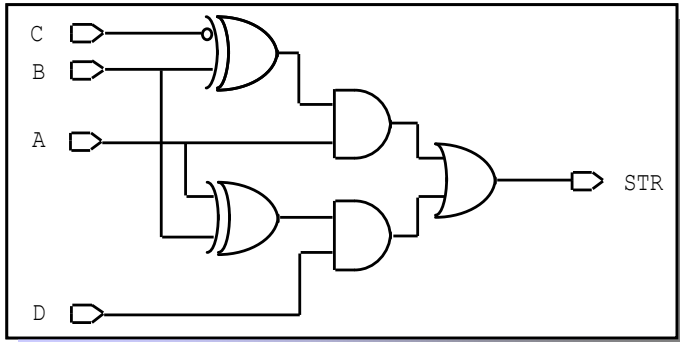
- Substitution
  - Given two functions express one in terms of the other
    - Given F = A + BC, G = A + B
    - Re-written as F = G (A + C)
- Collapsing
  - The opposite of substitution
  - Given a function break it into sub functions
    - F = G (A + C)
    - F = (A +B) (A + C)
    - F = A + AB + AC + BC
    - F = A + BC

Flattening vs Structuring

- Removes intermediate structures
- Done independent of constraints
- Highly Area intensive
- Limited by kinds of cells available in the library
- Two – level optimization



- Creates intermediate structures to implement design
- Is Constraint based
- Improves area as well as speed
- Default optimization strategy
- Multi - level optimization



Path Groups

- During synthesis the design is considered to be a collection of groups
- Each clock creates a group for itself
- All input -> reg paths and reg -> reg paths are grouped
- All reg -> output paths are grouped
- All input -> output paths (if any) are grouped
- Any group which is not constrained by clock is named as ‘default’ and the remaining groups are named after the clock which constrains them
- The so formed groups are then individually checked for meeting the constraints

Synthesis Cost Funcions

- For synthesis to happen the cost of optimization is calculated based on the targeted constraints and obtained results
- Cost = $\Sigma\ \Delta$ max_delay + Δ min_delay + Δ max_area +. .
- The max delay costs can be calculated as
  - max_delay cost : Σ $weight_i$ * $worst\_slack_i$
    - Where i denotes a particular path group
    - By default weight is 1
    - The worst slack per path group is calculated as :

      max (0, actual path delay – max delay)

Group and Weight

- A particular path group can be allocated a higher priority by increasing the 'weight' of that group
- Particular cells can be grouped for better optimization
- Cells grouped thus can be assigned a particular weight for higher optimization
  - group_path –from [all_inputs] –to a_reg*/D –name one_g –weight 4
- A group is created of all paths from all inputs to flip-flops with name 'a'
- The group so created is named as 'one_g' and allotted a weight of 4

Critical Range

- Synthesis process by default tries to minimize the delay in the path with maximum violation
  - Known as Worst Negative Slack optimization
  - May not be efficient for large designs with multiple paths in a group having comparable timing violations

- Alternative is to optimize a number of paths together so as to achieve better quality of results
  - Known as Total Negative Slack optimization
  - A ‘critical range’ is set

    set_critical_range 0.35

    All paths whose slacks fall within 0.35 time units from the slack of critical path are optimized

Strategies for Compiling Designs

- Compile approaches that must be avoided are
  - Capturing the entire design in one large HDL file, reading that file in to DC, specifying the following constraint
    - **max_delay 0 -from all_inputs( ) –to all_outputs( )**
  - Dividing the design in to too many hierarchal sub blocks
    - This is extreme strategy of first one
  - This method is not recommended for ‘2’ reasons
    - Managing design with several sub blocks is difficult
    - Optimization across boundaries is not effective

Steps in optimizing

- Design is written in HDL and user has limited idea of timing requirements and wish to attain fastest possible design
- Simple strategy to realize the optimal design is to start with medium effort compile, specify no constraints before compile step
- This gives the feel for timing/area performance of your block
- Next specify your approximate timing
- Analyze your results using command

  report_constraints –all_violators –verbose
- DC lists all the paths that fail to meet timing requirements in design

Optimizing (contd)

- If few paths violate timing recompile and analyze the results
- If a number of paths violated by a large margin then meeting timing is difficult or impossible
- Next recompile the design after reading in source code and specifying optimization constraints
- Optimization works best when you set achievable targets or go about achieving your goals in step by step process

Optimizing (contd)

- In general when timing is not met one or more of these can be followed
  - Re-assess the code and consider alternate design partitioning
  - Modify the constraints to set more realistic values
  - Identify any functional false paths or multi-cycle paths and specify them
  - Specify point-point delay timing constraints for asynchronous paths
  - For synchronous paths that violate timing use the group_path command to create separate path groups and assign weights to different path groups

Optimizing (contd)

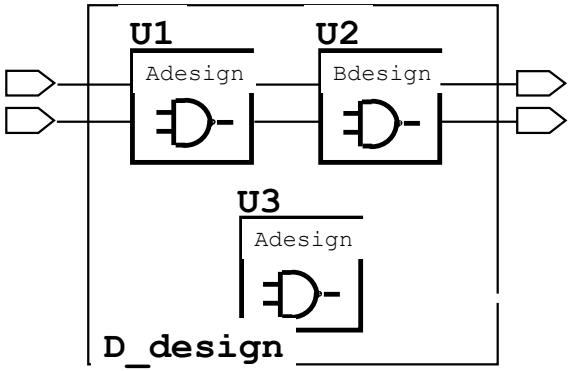
- In general when timing is not met one or more of these can be followed
  - Modify the constraints to set more realistic values
  - Identify any functional false paths or multi-cycle paths and specify them
  - Specify point-point delay timing constraints for asynchronous paths
  - For synchronous paths that violate timing use the group_path command to create separate path groups and assign weights to different path groups
  - compile_default_critical_range variable can be used to ensure that DC not only work on worst violator
    - Example: when this variable is set to the value of 2 , DC looks at all paths within 2ns of possible violation of timing
  - Re-assess the code and consider alternate design partitioning

Synthesis Strategy and Budget

- Overall design goals for timing, area, and power should be documented before macros are designed or selected.
- In particular, the overall chip synthesis methodology needs to be planned very early in the chip design process.
- Use bottom-up synthesis approach
  - Each macro should have its own synthesis script to ensure the internal timing of the macro can be met in the target library.
  - Chip -level synthesis then consists solely of connecting the macros and realizing output drive buffers to meet actual wire load and fanout
- The macro should appear at the top level as don't_touch

dont_touch

- set_dont_touch can be assigned to design objects
  - It prevents modification of that design object
  - Caution: If placed on an unmapped design, that design will remain unmapped
- In scenarios of multiple instantiation, set_dont_touch on the design Adesign
  - Prevents any further optimization of instances U1 and U3



Top Down Approach

- Compile as large a design as possible
  - Synthesis will take care of timing between interfaces automatically!
- Limitations:
  - How much “coding” is done at one time
  - Compile run time
    - Compiling 200k - 1000k gates at one time can
    - Result in an overnight compile (or longer)
  - Floorplaning/Place & Route issues

Characterize

- characterize simply captures a design's surroundings and places the actual constraints on the design.
  - characterize should only be used when all blocks are compiled
    - characterize uses existing gates to determine I/O delays, driving cells, and output loads
    - It is useless to characterize if the surroundings are GTECH!
  - characterize should NOT be used to derive design budgets of top level modules
    - All blocks must already be constrained and compiled before characterize can be used

Pros and Cons of top down

RAMAIAH SKILL ACADEMY

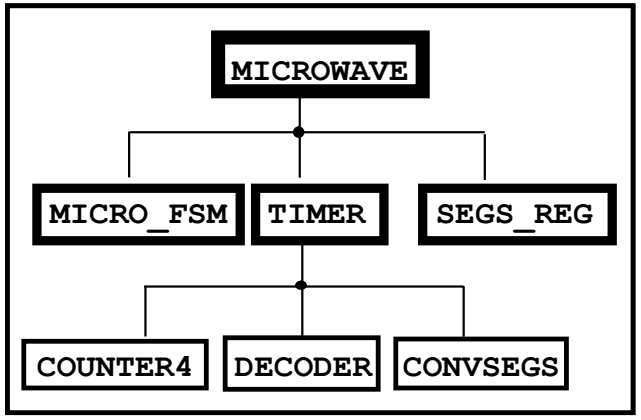
- Advantages
  - “push-button" approach
  - Intermodule dependencies are taken care of automatically
  - Fewer man-hours spent driving the tool
  - Most productive approach when practical
- Disadvantages
  - Can be memory and CPU intensive
  - Long compile run times with large designs or complex, “pushing the technology” constraints

Bottom Up

- Large, complex designs should be partitioned into smaller, easier-to-compile blocks
  - Individual blocks can then be compiled independently, *in parallel*
- What are some of the considerations when using a Bottom-Up methodology?
  - Constrain and compile sub blocks independently.
  - Make sure all sub blocks meet their initial constraints
  - Read in the entire compiled design and apply top-level constraints
  - Check constraint report: if design passes, you are done!
- How can we avoid integration problems when the sub-blocks are connected together?

Budgeting

- Design Budgeting
  - Develop time, load, and drive budgets for each block to meet
  - Compile each block to meet that budget



  - Integration at top level should cause no problems if:
    - All blocks compiled and met their budget
    - Budget was accurate and sufficiently constrained the design

PADs

- Pads are the interfaces between the core (logic) and the external world (package)
- Pads can be inserted at three places
  - During physical synthesis
  - During logic Synthesis – hand instantiation in netlist
  - In the RTL code
- Of all the methods instantiating pads in the code is the best as it gives a broader insight to the synthesis tool to pick more appropriate cells
- IO designer suggests the suitable IOs based on the load being driven by the ports / specifications etc
- Separate IOs are available for logic and clocks
  - PDDDGZ is an input pad with PAD as its input and C as output

```
module PDDDGZ (PAD, C);
  input PAD;
  output C;
```

IO PADs

- PAD connects to the port of the design while C connects to the Core of the design
- Rename the input ports of your design by appending '_p' to their names
- Instantiate the pads and connect the ports to the pads
- Simultaneously connect the core to the output of the pads

Before Pads

```
module ff (
output reg Q,
input clk, data);

always@(posedge clk)
Q <= data;

endmodule
```

After Pads

```
module ff (
output Q_p,
input clk_p, data_p);

wire clk, data;
reg Q;

PDDSDGZ    ck  ( .PAD(clk_p), .C(clk));
PDDDGZ     dt    (.PAD(data_p), .C(data));
PDO02CDG  qp   (.I(Q), PAD(Q_p));

always@(posedge clk)
  Q <= data;

endmodule
```