CAL POLY
SAN LUIS OBISPO

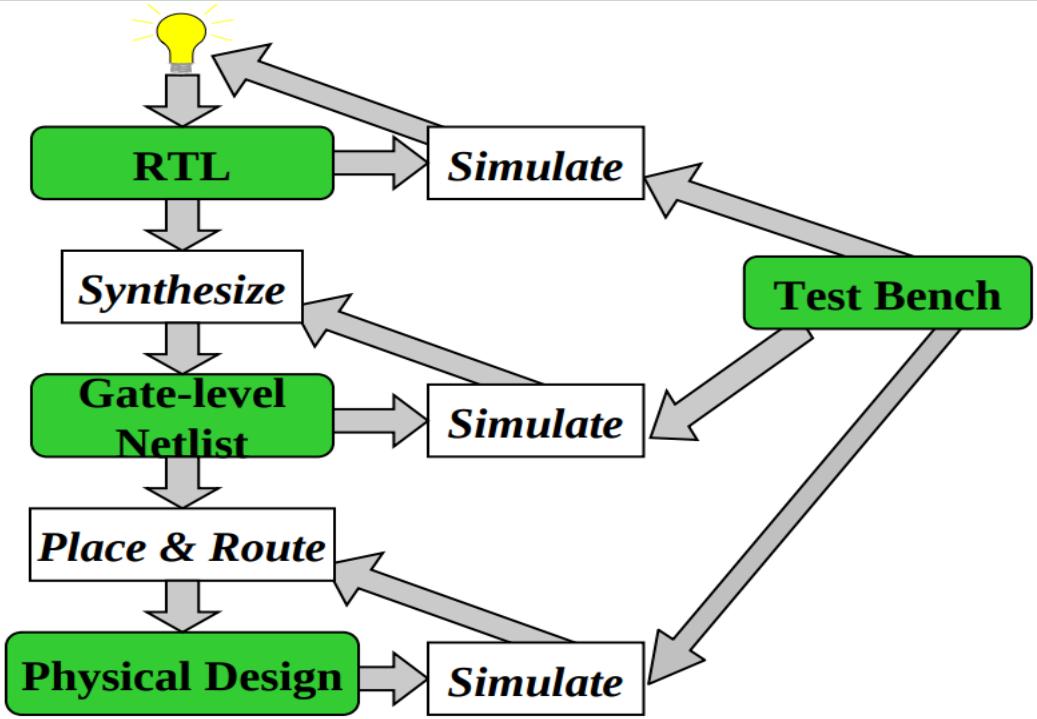
EE 531: ADVANCED VLSI DESIGN

Logic Synthesis

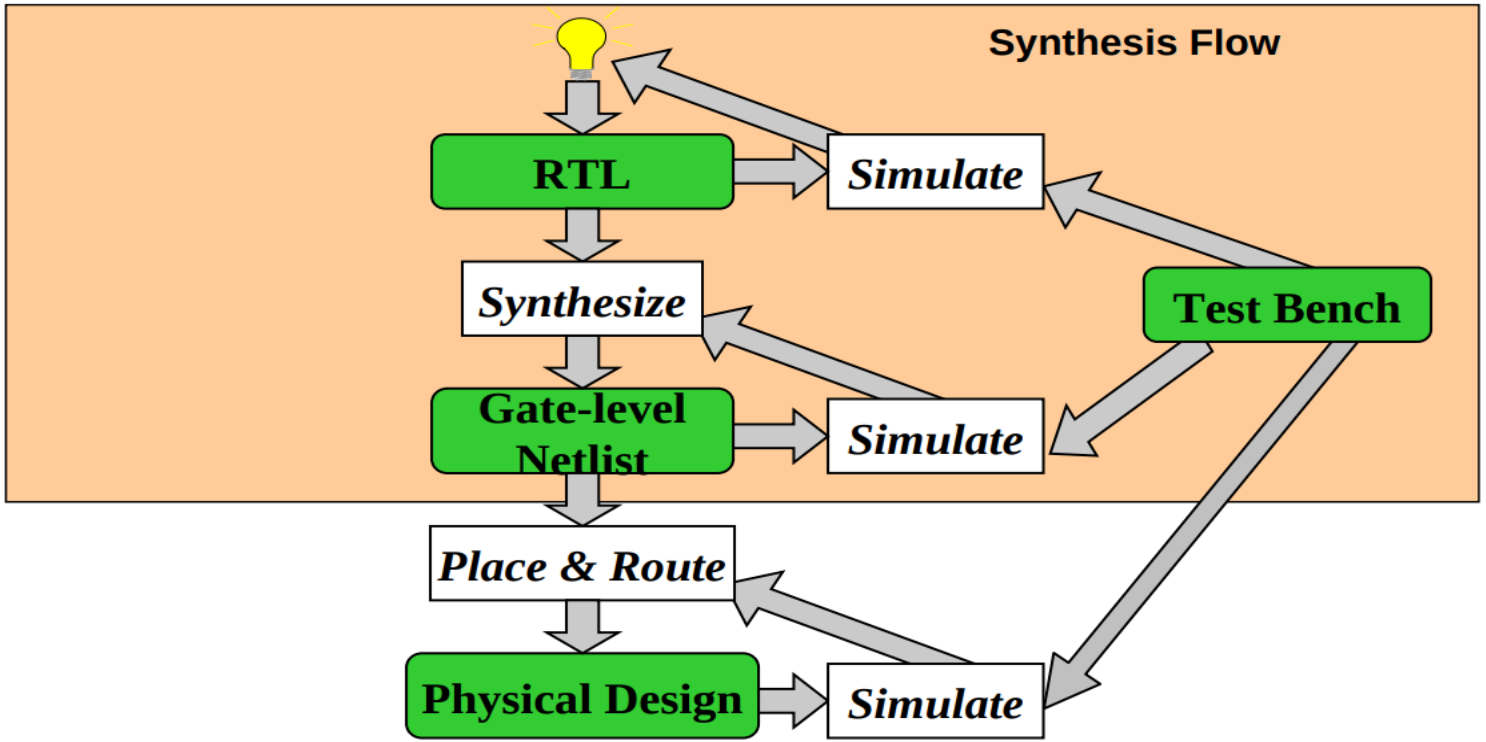
Nishith N. Chakraborty

January, 2025

ASIC/SOC DESIGN FLOW



ASIC/SOC DESIGN FLOW



RTL SYNTHESIS DESIGN STEPS

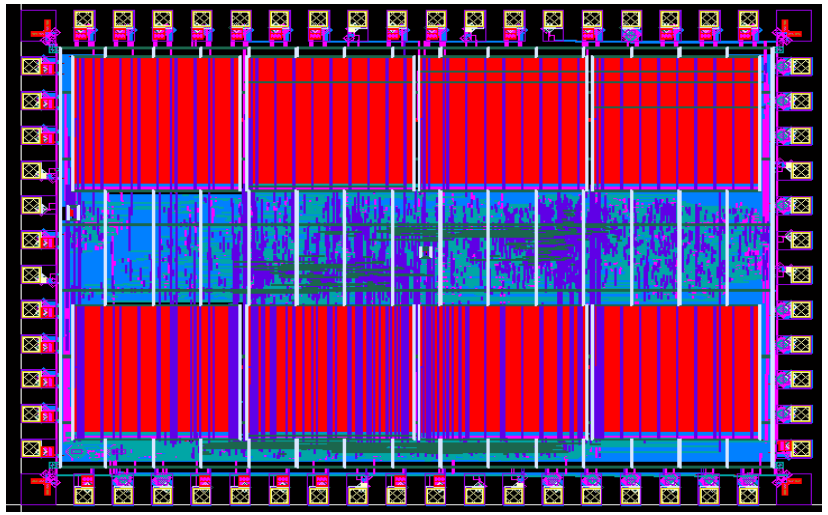
- Code design in HDL such as VHDL or Verilog
  - Can use ‘gedit’ on Linux
- Simulation/verification of HDL description
  - QuestaSim (Mentor Graphics), ModelSim or Vivado (Xilinx)
  - Use test bench, Verilog or VHDL
- Synthesis of HDL description
  - Use Design Vision, Genus, Yosys
  - Output of synthesis is a Verilog gate level netlist
  - Netlist built from standard cells
- Gate level netlist should be simulated using same test bench designed for RTL verification
  - QuestaSim or Vivado useful here as well

CODING HARDWARE

- Previous weeks: combinational and sequential logic with SystemVerilog
- Verilog used to describe hardware components in code
- Use structural Verilog (or VHDL) as much as possible

CODING HARDWARE

```
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------
-----------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity SRAD_Top is
    Port ( clk      : in std_logic;
           ExtRST   : in std_logic;
           NewFrm   : in std_logic;
           DataIn   : in std_logic_vector(7 downto 0);
           Addr_R   : out std_logic_vector (13 downto 0);
           Dsp_Addr                    : out std_logic_vector (13
downto 0);
           DataOut  : out std_logic_vector(7 downto 0);
           InAdMux  : out std_logic;
           DisAdMux: out std_logic;
           Dsp_WE   : out std_logic;
           VGA_Ena  : out std_logic;
           SRAD_Clk: out std_logic
                    );
end SRAD_Top;

architecture Structure of SRAD_Top is

COMPONENT Antilog
    Port ( D       : in std_logic_vector(7 downto 0);
```

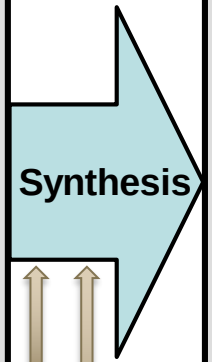


WHAT IS LOGIC SYNTHESIS?

- Synthesis is the process that converts RTL into a technology-specific gate-level netlist, optimized for a set of pre-defined constraints.
- You start with:
  - A behavioral RTL design
  - A standard cell library
  - A set of design constraints
- You finish with:
  - A gate-level netlist, mapped to the standard cell library
  - For FPGAs: LUTs, flip-flops, and RAM blocks
  - Hopefully, it's also efficient in terms of speed, area, power, etc.

LOGIC SYNTHESIS: EXAMPLE

```
module counter(
     input clk, rstn, load,
     input  [1:0] in,
     output reg [1:0] out);
always @(posedge clk)
  if (!rstn) out <= 2'b0;
  else if (load) out <= in;
  else out <= out + 1;
endmodule
```



Standard Cell Library

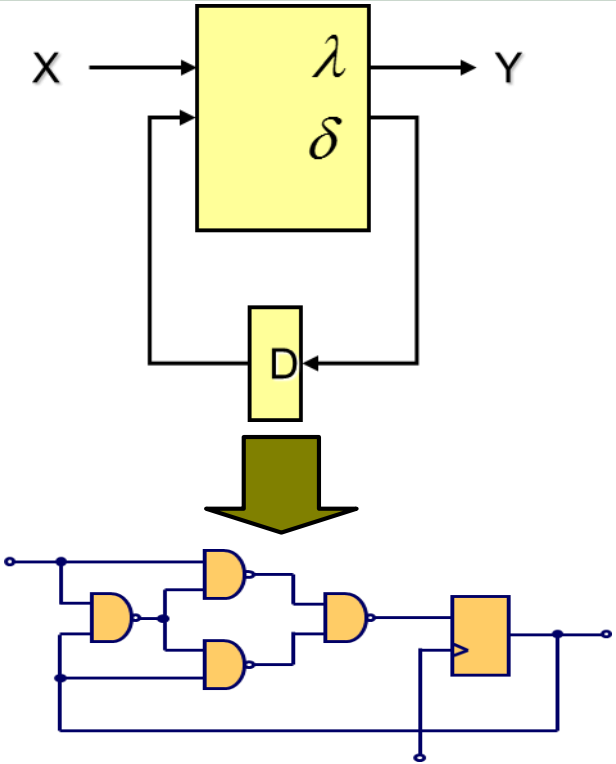
Design Constraints

```
module counter ( clk, rstn, load,
  in, out ); input [1:0] in;
  output [1:0]
  out; input
  clk, rstn,
  load;
  wire   N6, N7, n5, n6, n7, n8;

  FFPQ1 out_reg_1 (.D(N7),.CK(clk),.Q(out[1]));
  FFPQ1 out_reg_0 (.D(N6),.CK(clk),.Q(out[0]));
  NAN2D1 U8 (.A1(out[0]),.A2(n5),.Z(n8));
  NAN2D1 U9 (.A1(n5),.A2(n7),.Z(n6));
  INVD1 U10 (.A(load),.Z(n5));
  OA211D1 U11 (.A1(in[0]),.A2(n5),.B(rstn),.C(n8),.Z(N6));
  OA211D1 U12 (.A1(in[1]),.A2(n5),.B(rstn),.C(n6),.Z(N7));
  EXNOR2D1 U13 (.A1(out[1]),.A2(out[0]),.Z(n7));
endmodule
```

WHAT IS LOGIC SYNTHESIS?

- Given: Finite-State Machine F(X, Y, Z, λ, δ), where:
  - X: Input
  - Y: Output
  - Z: Set of internal states
  - λ: $X \times Z \rightarrow Z$ (next state function)
  - δ: $X \times Z \rightarrow Y$ (output function)
- Target: Circuit C(G, W) where:
  - G: set of circuit components

    G = {Boolean gates, flip-flops, etc.}
  - W: set of wires connecting G



MOTIVATION

Why perform logic synthesis?

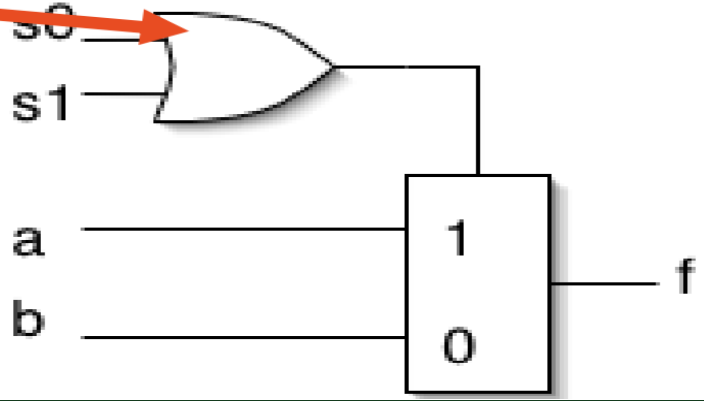
- Automatically manages many details of the design process
- Fewer bugs
- Improves productivity
- Abstracts the design data (HDL description) from any particular implementation technology
- Designs can be re-synthesized targeting different chip technologies;
  - E.g.: first implement in FPGA then later in ASIC
- In some cases, leads to a more optimal design than could be achieved by manual means (e.g.: logic optimization)

Why not logic synthesis?

- May lead to less than optimal designs in some cases

LOGIC SYNTHESIS: ANOTHER SIMPLE EXAMPLE

```
module foo (a,b,s0,s1,f);
 input [3:0] a;
 input [3:0] b;
 input s0,s1;
 output [3:0] f;
 reg f;

 always @ (a or b or s0 or s1)
   if (!s0 && s1 || s0)
       f=a;
   else
       f=b;
endmodule
```



REGISTER EXAMPLE

```
// Simple SystemVerilog 4-bit register

module reg4 (input d0, d1, d2, d3, en, clk,
             output q0, q1, q2, q3);
  logic q0_tmp, q1_tmp, q2_tmp, q3_tmp;

  always_ff @(posedge clk) begin
    if (en) begin
      q0_tmp <= d0;
      q1_tmp <= d1;
      q2_tmp <= d2;
      q3_tmp <= d3;
    end
  end

  assign #2 q0 = q0_tmp;
  assign #2 q1 = q1_tmp;
  assign #2 q2 = q2_tmp;
  assign #2 q3 = q3_tmp;
endmodule
```

REGISTER EXAMPLE

```
// Simple SystemVerilog 4-bit register

module reg4 (input d0, d1, d2, d3, en, clk,
             output q0, q1, q2, q3);
  logic q0_tmp, q1_tmp, q2_tmp, q3_tmp;

  always_ff @(posedge clk) begin
    if (en) begin
      q0_tmp <= d0;
      q1_tmp <= d1;
      q2_tmp <= d2;
      q3_tmp <= d3;
    end
  end

  assign #2 q0 = q0_tmp;
  assign #2 q1 = q1_tmp;
  assign #2 q2 = q2_tmp;
  assign #2 q3 = q3_tmp;
endmodule
```

Will this synthesize?

REGISTER EXAMPLE

```
// Simple SystemVerilog 4-bit register

module reg4 (input d0, d1, d2, d3, en, clk,
             output q0, q1, q2, q3);
  logic q0_tmp, q1_tmp, q2_tmp, q3_tmp;

  always_ff @(posedge clk) begin
    if (en) begin
      q0_tmp <= d0;
      q1_tmp <= d1;
      q2_tmp <= d2;
      q3_tmp <= d3;
    end
  end

  assign #2 q0 = q0_tmp;
  assign #2 q1 = q1_tmp;
  assign #2 q2 = q2_tmp;
  assign #2 q3 = q3_tmp;
endmodule
```

Will this synthesize?

Yes, will ignore '#2'

GOALS OF LOGIC SYNTHESIS

- **Minimize area**
  - In terms of literal count, cell count, register count, etc.
- **Minimize power**
  - In terms of switching activity in individual gates, deactivated circuit blocks, etc.
- **Maximize performance**
  - In terms of maximal clock frequency of synchronous systems, throughput for asynchronous systems
- **Any combination of the above**
  - Combined with different weights
  - Formulated as a constraint problem
    - "Minimize area for a clock speed > 300MHz"

GOALS OF LOGIC SYNTHESIS

- **More global objectives**
  - **Feedback from layout**
    - **Actual physical sizes, delays, placement and routing**

HOW DOES IT WORK?

Variety of general and ad-hoc (special case) methods:

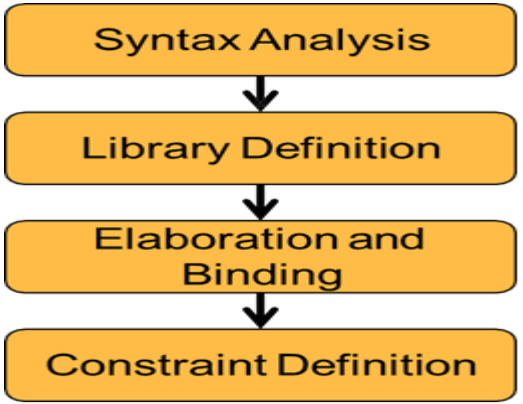
- **Instantiation:**
  - Maintains a library of primitive modules (AND, OR, etc.) and user defined modules
- **"Macro expansion"/substitution:**
  - A large set of language operators (+, -, Boolean operators, etc.) and constructs (if-else, case) expand into special circuits
- **Inference:**
  - Special patterns are detected in the language description and treated specially
  - e.g.,: inferring memory blocks from variable declaration and read/write statements, FSM detection and generation from always@(posedge clk) blocks

HOW DOES IT WORK? (CONTINUED)

- **Logic optimization:**
  - Boolean operations are grouped and optimized with logic minimization techniques
- **Structural reorganization:**
  - Advanced techniques including sharing of operators, and retiming of circuits (moving FFs), and others

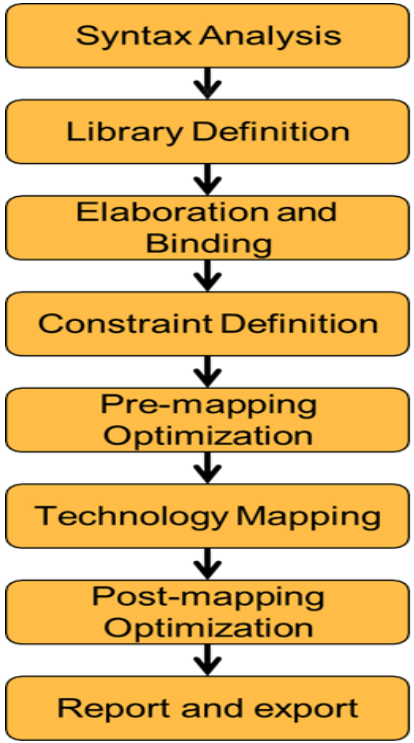
BASIC SYNTHESIS FLOW

- **Syntax Analysis:**
  - Read in HDL files and check for syntax errors.
- **Library Definition:**
  - Provide standard cells and IP Libraries.
- **Elaboration and Binding:**
  - Convert RTL into Boolean structure.
  - State reduction, encoding, register infering.
  - Bind all leaf cells to provided libraries.
- **Constraint Definition:**
  - Define clock frequency and other design constraints.



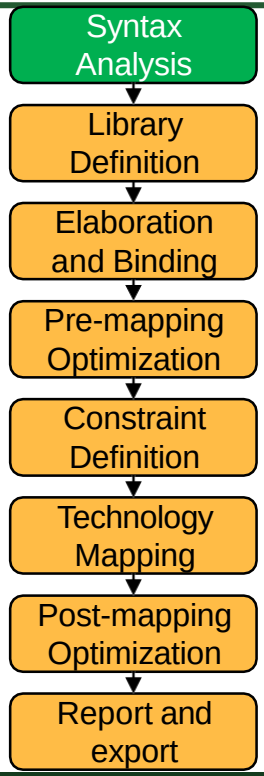
BASIC SYNTHESIS FLOW (CONTINUED)

- **Pre-mapping Optimization:**
  - Map to generic cells and perform additional heuristics.
- **Technology Mapping:**
  - Map generic logic to technology libraries.
- **Post-mapping Optimization:**
  - Iterate over design, changing gate sizes, Boolean literals, architectural approaches to try and meet constraints.
- **Report and export:**
  - Report final results with an emphasis on timing reports.
  - Export netlist and other results for further use.

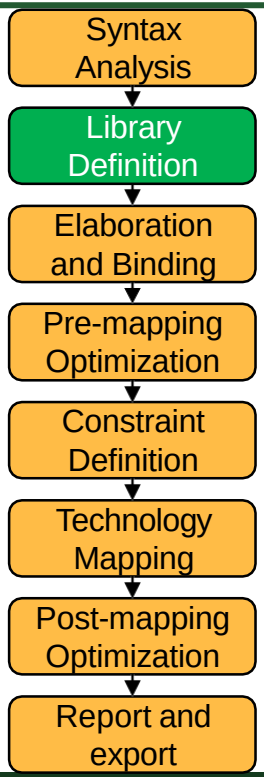


COMPILATION IN THE SYNTHESIS FLOW

- Before starting to synthesize, we need to check the syntax for correctness.
- Synthesis vs. Compilation:
  - Compiler
    - Recognizes all possible constructs in a formally defined program language
    - Translates them to a machine language representation of execution process
  - Synthesis
    - Recognizes a target dependent subset of a hardware description language
    - Maps to collection of concrete hardware resources
    - Iterative tool in the design flow



LIBRARY DEFINITION



IT'S ALL ABOUT THE STANDARD CELLS…

- The library definition stage tells the synthesizer where to look for leaf cells for binding and the target library for technology mapping.
  - We can provide a list of paths to search for libraries in:

```
set_db init_lib_search_path "/design/data/my_fab/digital/lib/"
```

- And we have to provide the name of a specific library, usually characterized for a single corner:
  - We also need to provide .lib files for IPs, such as memory macros, I/Os, and others.

```
read_libs "TT1V25C.lib"
```

- Make sure you understand all the warnings about the libs that the synthesizer spits out, even though you probably can't fix them.

STANDARD CELL LIBRARY: DEFINITION

- A standard cell library is a collection of well defined and appropriately characterized logic gates that can be used to implement a digital design.
- Similar to LEGO, standard cells must meet predefined specifications to be flawlessly manipulated by synthesis, place, and route algorithms.
- Therefore, a standard cell library is delivered with a collection of files that provide all the information needed by the various EDA tools.

REASONING FOR STANDARD CELLS

- Limit implementation effort by reusing limited library of cells
- Cells need to be designed and verified only once
  - Reuse many times: amortization of design cost
- Trade-offs vs. full custom design
  - Reduced integration density
  - Lower performance
  - Can’t “fine-tune” a design at a low level
  - Options limited by quality of available library

LIBRARY CONSIDERATIONS

- Many options when determining library composition
- Design of a standard cell library can take long time
  - Difference from full custom: amortization over a large number of designs due to reuse
- Which is better?
  - A small library with limited fan-in and fan-out
  - A large library with many versions of gates
    - Different fan-ins
    - Different sizing for different loads

LIBRARY CONSIDERATIONS

- Fan-out and wiring parasitics are unknown at the beginning of the design process
- The design process can be simplified by ensuring each library element can drive a large fan-out
  - Oversized cells that fit many conditions
  - Trade-offs:
    - Further degradation of performance
    - Further waste of area

USING THE LIBRARY

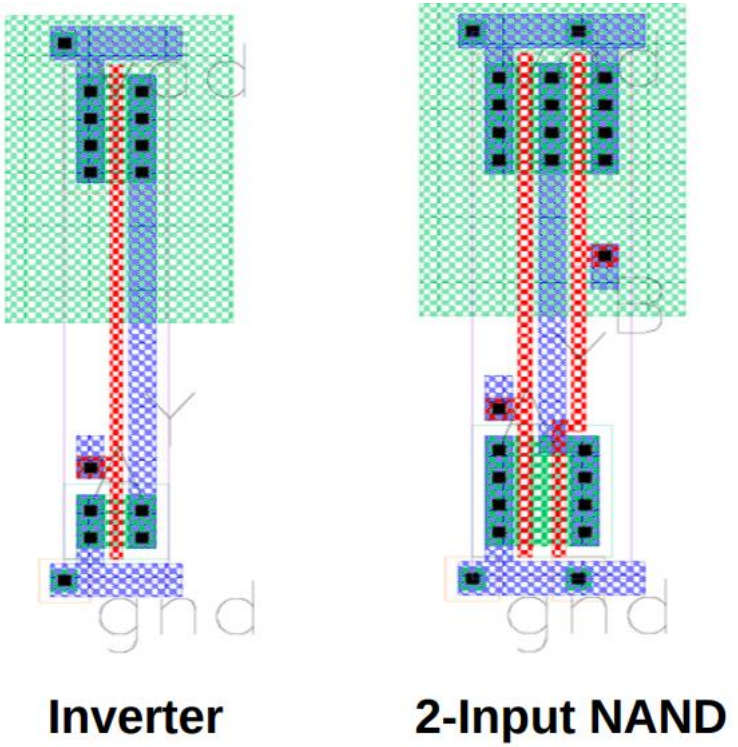
- A design can be captured at schematic level where it is composed of only the cells in the library
- Usually, a design is generated from a high-level hardware description language (VHDL, Verilog, etc)
- The layout is then automatically generated
- Synthesis tool must choose the right cells for given speed, power, and area requirements

STANDARD CELL COMPONENTS

- Standard cells are building blocks for overall design
- Each cell in the standard cell library consists of more than just the layout
- Essential components of most standard cells:
  - Layout
  - Abstract view
  - Schematic
  - HDL representation (functional description)
  - Timing information

STANDARD CELL LAYOUTS

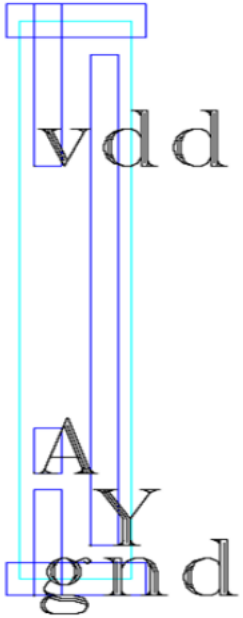
- Major component of a standard cell is layout
- Layouts of all cells tend to be the same height or a multiple of that height
- These layout cells are used for final layout
- On right: Inverter and NAND layouts for a 0.18um library



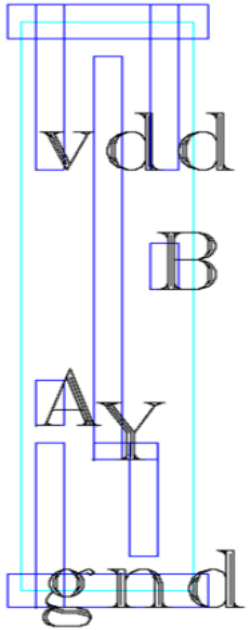
Inverter 2-Input NAND

STANDARD CELL ABSTRACT

- When CAD tools route, only concern is metal
- An abstract view is a layout with only the metal layers
- The abstract view is used by tools for routing
- Used for LEF description
- Examples on the right



Inverter



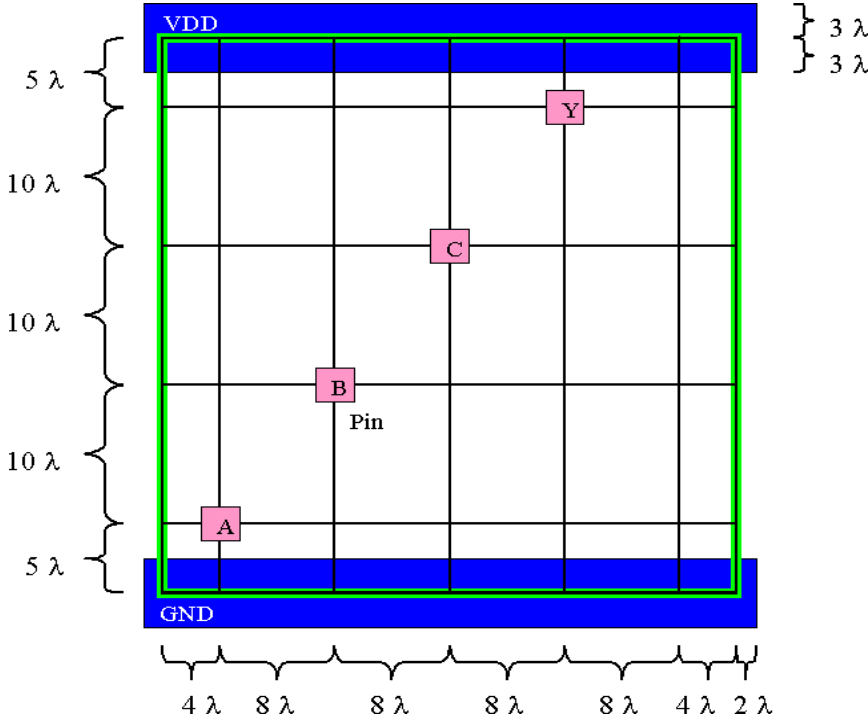
2-Input NAND

THE ROUTING GRID

- Every standard cell must include conventions to determine dimension and placement of cell features
- Some standard cell libraries use a conservative grid to give more flexibility in cell design
- These grid rules must be inline with design rules

ROUTING GRID EXAMPLE

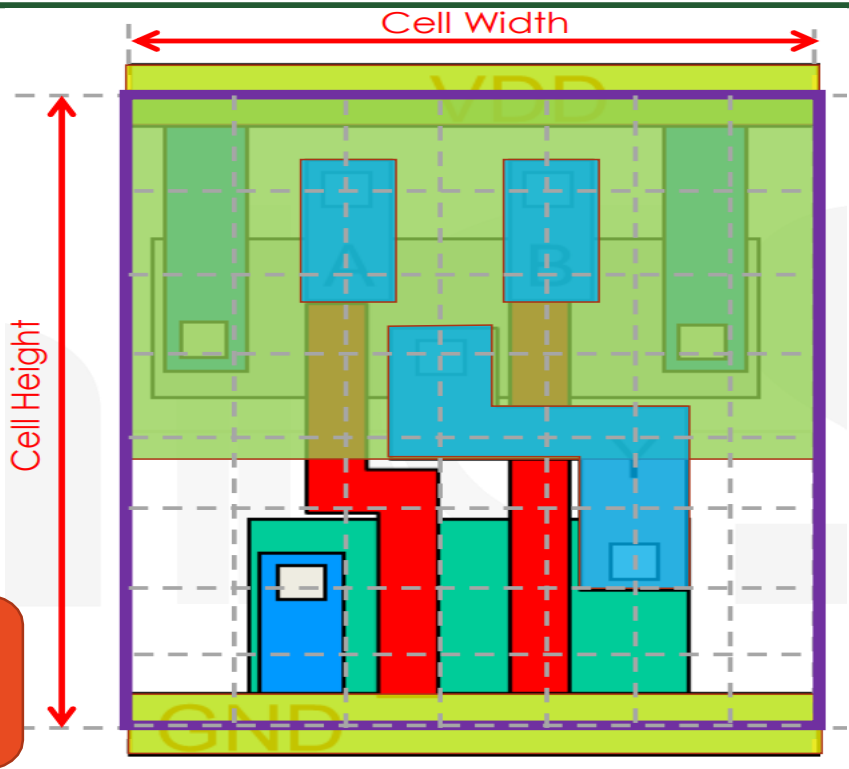
- Metal spacings are via to via
- Vias can be placed on two adj. grid points
- Routing made easier by wider grid points
- Illustration of grid rules to the right
- Any customized standard cell should follow these rules



DETAILED EXAMPLE: NAND GATE

Pay attention to:

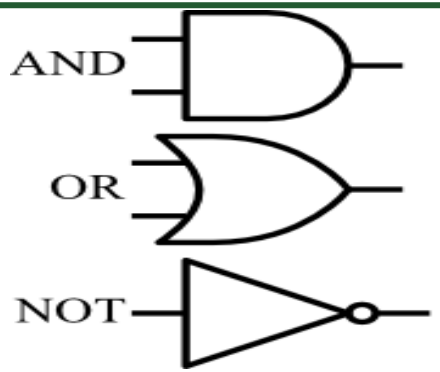
- Cell height
- Cell width
- Voltage rails
- Well definition
- Pin Placement
- PR Boundary
- Metal layers

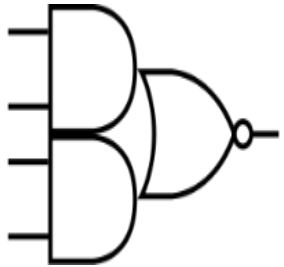


Ideally, Standard Cells should be routed entirely in M1 !

WHAT CELLS ARE IN A STANDARD CELL LIBRARY?

- **Combinational logic cells (NAND, NOR, INV, etc.):**
  - Variety of drive strengths for all cells.
  - Complex cells (AOI, OAI, etc.)
  - Fan-In <= 4
  - ECO Cells
- **Buffers/Inverters**
  - Larger variety of drive strengths.
  - "Clock cells" with balanced rise and fall delays.
  - Delay cells
  - Level Shifters

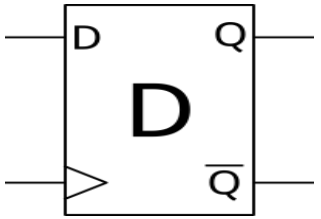




AND-OR INVERT (AOI)

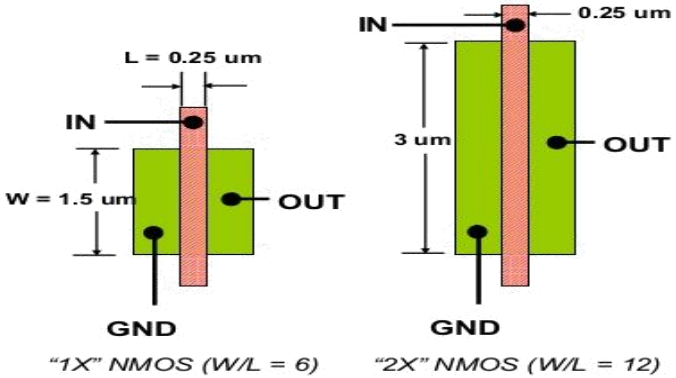
WHAT CELLS ARE IN A STANDARD CELL LIBRARY?

- **Sequential Cells:**
  - Many types of flip flops: pos/negedge, set/reset, Q/QB, enable
  - Latches
  - Integrated Clock Gating cells
  - Scan enabled cells for ATPG (Automatic Test Pattern Generation).
- **Physical Cells:**
  - Fillers, Tap cells, Antennas, DeCaps, EndCaps, Tie Cells



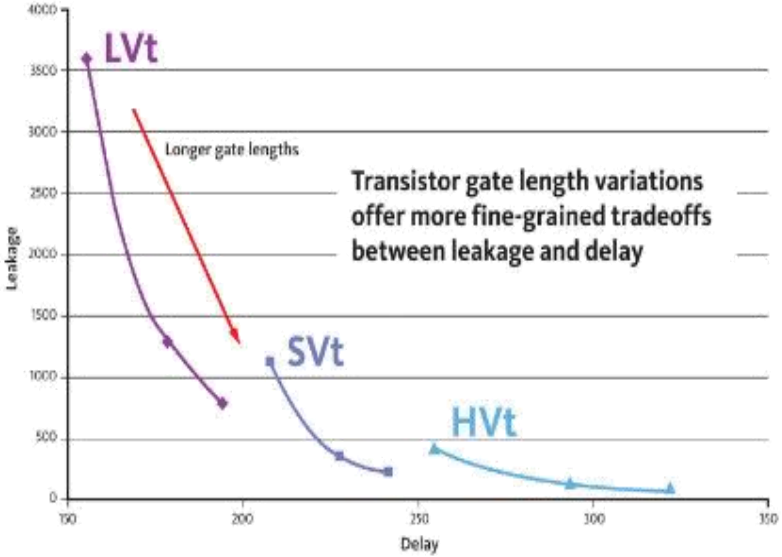
MULTIPLE DRIVE STRENGTHS

- **Multiple Drive Strength**
  - Each cell will have various sized output stages.
  - Larger output stage → better at driving fanouts/loads.
  - Smaller drive strength → less area, leakage, input cap.
  - Often called X2, X3, or D2, D3, etc.



MULTIPLE VTS

- **Multiple Threshold (MT-CMOS)**
  - A single additional mask can provide more or less doping in a transistor channel, shifting the threshold voltage.
  - Most libraries provide equivalent cells with three or more VTs: SVT, HVT, LVT
    - This enables tradeoff between speed vs. leakage.
  - All threshold varieties have same footprint and therefore can be swapped without any placement/routing iterations.

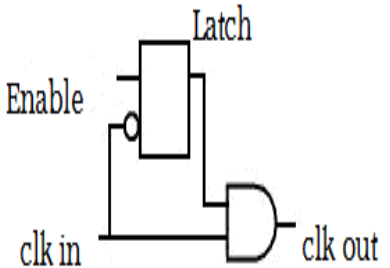


CLOCK CELLS

- General standard cells are optimized for speed.
  - That doesn't mean they're balanced…

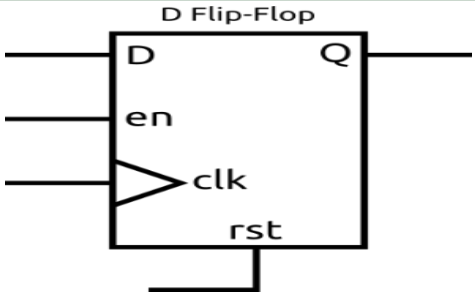
$$\min t_{pd} = \min\left(\frac{t_{p,LH} + t_{p,HL}}{2}\right) \not\Rightarrow t_{p,LH} = t_{p,HL}$$

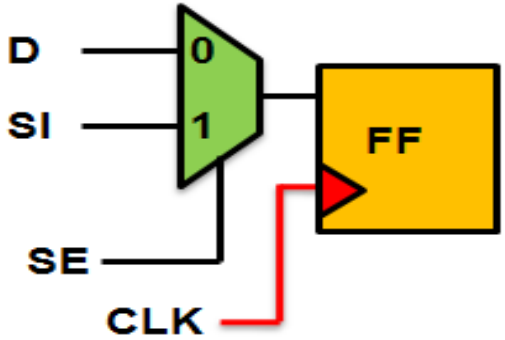
- This isn't good for clock nets…
  - Unbalanced rising/falling delays will result in unwanted skew.
  - Special "clock cells" are designed with balanced rising/falling delays to minimize skew.
  - These cells are usually less optimal for data and so should not be used.
- In general, only buffers/inverters should be used on clock nets
  - But sometimes, we need gating logic.
  - Special cells, such as integrated clock gates, provide logic for the clock networks.



SEQUENTIAL CELLS

- **Flip Flops and Latches, including**
  - Positive/Negative Edge Triggered
  - Synchronous/Asynchronous Reset/Set
  - Q/QB Outputs
  - Enable
  - Scan
  - etc.



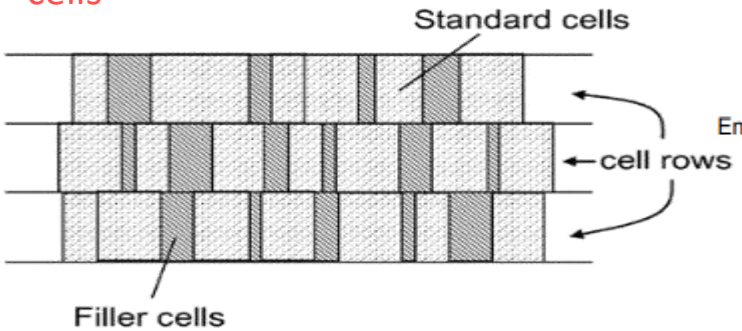


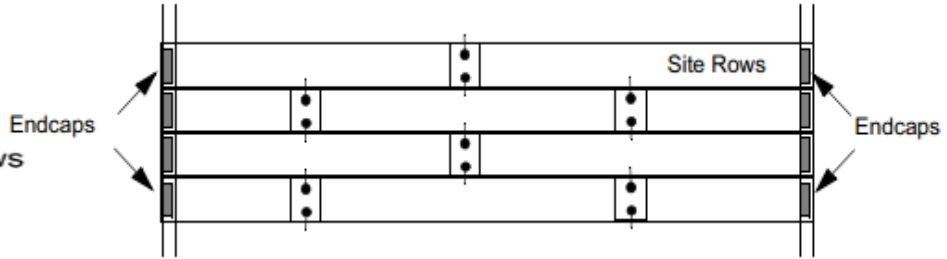
LEVEL SHIFTERS

- Level shifter cells are placed between voltage domains to pass signals from one voltage to another.
- HL (high-to-low) shifter
  - Requires only one voltage
  - Single height cell
- LH (low-to-high) shifter
  - Needs 2 voltages
  - Often double height

FILLER CELLS

- Filler cells Must be inserted in empty areas in rows
  - Ensure well and diffusion mask continuity
  - Ensure density rules on bottom layers
  - Provide dummy poly for scaled technologies
  - Sometimes, special cells are needed at the boundaries of rows - "End Caps"
  - Other fillers may include MOSCAPs between VDD and GND for voltage stability - "DeCAP cells"



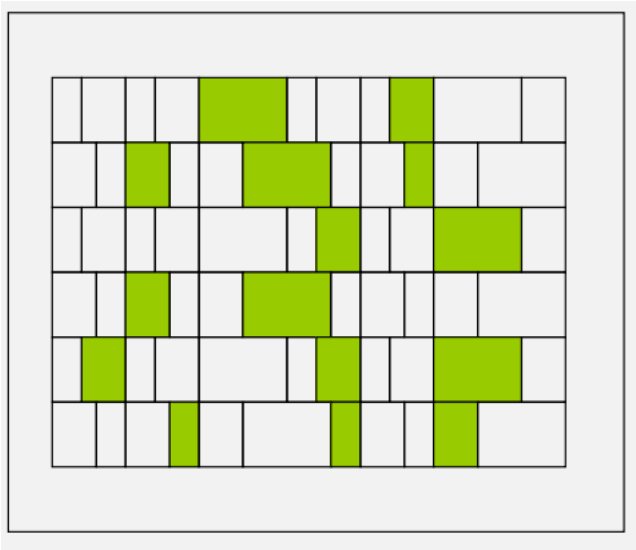


ENGINEERING CHANGE ORDER (ECO) CELLS

- An Engineering Change Order (ECO) is a very late change in the design.
  - ECOs usually are done after place and route.
  - However, re-spins of a chip are often done without recreating all-masks. This is known as a "Metal-Fix".
- ECOs usually require small changes in logic.
  - How can we do this after placement?
  - Or worse – after tapeout???

ENGINEERING CHANGE ORDER (ECO) CELLS

- Solution – Spare (Bonus) Cells!
  - Cells without functionality
  - Cells are added during design (fill)
  - In case of problems (after processing) new metal and via mask → cells get their wanted functionality
  - Cell combinations can create more complex functions
  - Ex. AND,NAND,NOR,XOR,FF,MUX,INV,..
- Special standard cells are used to differentiate from real cells.



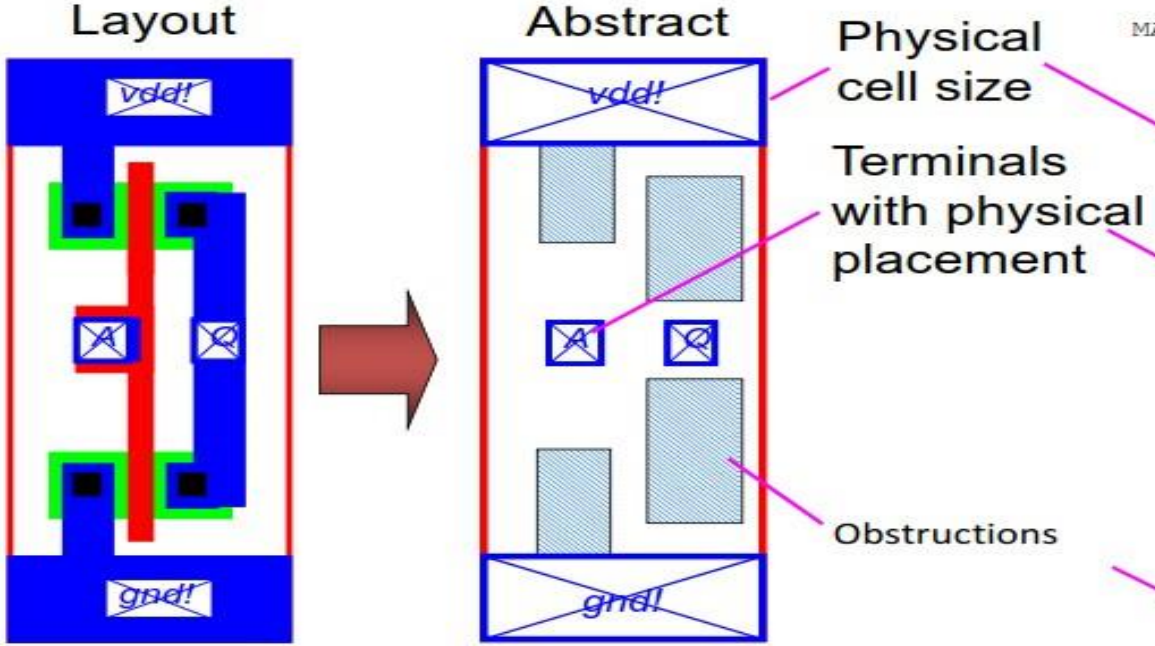
LIBRARY EXCHANGE FORMAT (LEF)

- Placement tools must have a standard cell definition describing dimensions and port locations
- Typically, this information is included in a Library Exchange Format (LEF) file
- The LEF file also contains capacitance and resistance information for each metal layer
- Cadence Abstract Generator can be used to generate such LEF files if needed
- Any custom blocks to be used must be added to the LEF description

LEF (CONTINUED)

- Abstract description of the layout for P&R
  - Readable ASCII Format.
  - Contains detailed PIN information for connecting.
  - Does not include front-end of the line (poly, diffusion, etc.) data.
- Abstract views only contain the following:
  - Outline of the cell (size and shape)
  - Pin locations and layer (usually on M1)
  - Metal blockages
    - Areas in a cell where metal of a certain layer is being used, but is not a pin

EXAMPLE LEF DESCRIPTION



LEF

```
MACRO IV
    CLASS CORE ;
    FOREIGN IV 0.000 0.000 ;
    ORIGIN 0.00 0.00 ;
    SIZE 3.00 BY 12.00 ;
    SYMMETRY x y  ;
    SITE CORE ;
    PIN A
        DIRECTION INPUT ;
        ANTENNASIZE 1.4 ;
        PORT
        LAYER metal1 ;
        RECT  0.50 5.00 1.00 5.50
    ;
        END
    END A

    OBS
        LAYER metal1 ;
        RECT  1.90 6.50 2.60 7.20
    ;
        RECT  0.40 4.90 1.00 5.60
    ;
```

TECHNOLOGY LEF

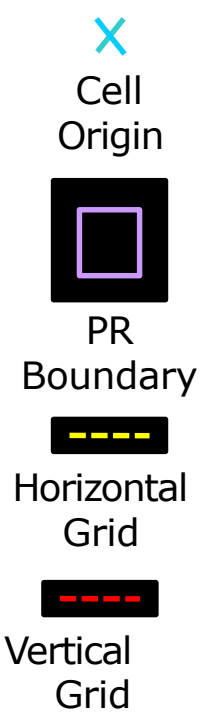
- Technology LEF Files contain (simplified) information about the technology for use by the placer and router:
  - Layers
    - Name, such as M1, M2, etc.
    - Layer type, such as routing, cut (via)
    - Electrical properties (R, C)
    - Design Rules
    - Antenna data
    - Preferred routing direction
  - SITE (x and y grid of the library)
    - CORE sites are minimum standard cell size
    - Can have site for double height cells!
    - IOs have special SITE.
  - Via definitions
  - Units
  - Grids for layout and routing

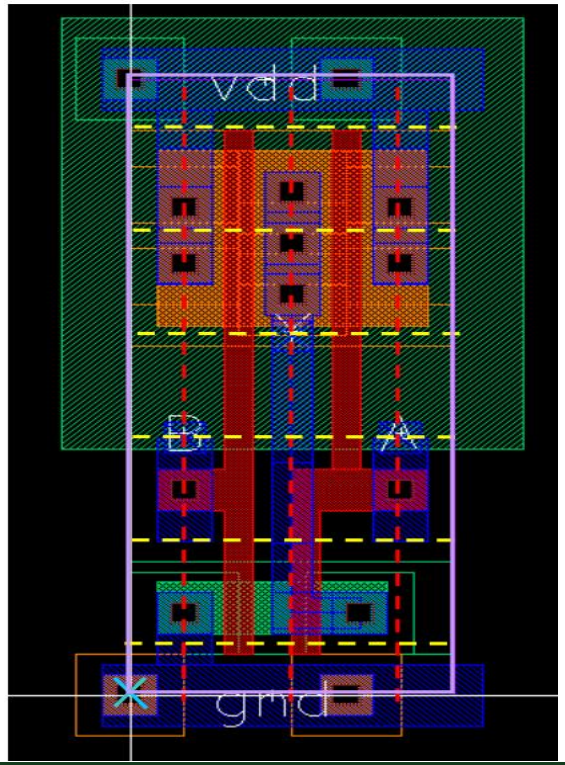
```
SITE CORE
  CLASS CORE;
  SIZE 0.2 X 12.0;
END CORE

LAYER MET1
  TYPE ROUTING ;
  PITCH 3.5 ;
  WIDTH 1.2 ;
  SPACING 1.4 ;
  DIRECTION HORIZONTAL ;
  RESISTANCE RPERSQ .7E-01 ;
  CAPACITANCE CPERSQDIST .46E-04 ;
END MET1

LAYER VIA
  TYPE CUT ;
END VIA
```

TECHNOLOGY LEF (CONTINUED)

- Cells must fit into a predefined grid
  - The minimum Height X Width is called a SITE.
  - Must be a multiple of the minimum X-grid unit and row height.
  - Cells can be double-height, for example.
- Pins should coincide with routing tracks
  - This enables easy connection of higher metals to the cell.

```
SITE CORE
  CLASS CORE;
  SYMMETRY X Y;
  SIZE 0.2 X 12.0;
END CORE
```





TIMING MODEL: .LIB DESCRIPTION

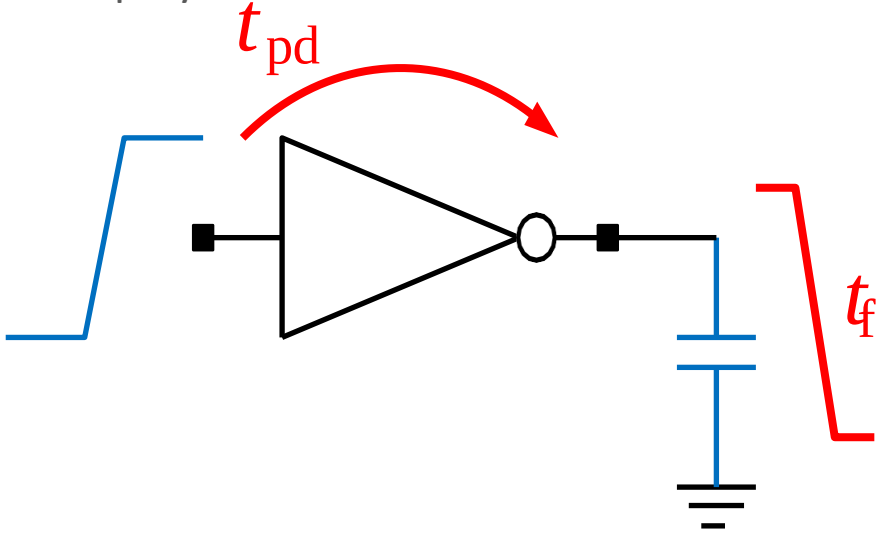
- The LIB file is also important (synthesis & routing):

```
...
cell (INVX1) {
  cell_footprint : inv;
area : 16;
  cell_leakage_power : 0.0221741;
  pin(A)  {
    direction : input;
    capacitance : 0.00932456;
    rise_capacitance : 0.00932196;
    fall_capacitance : 0.00932456;
  }
  pin(Y)  {
    direction : output;
    capacitance : 0;
    rise_capacitance : 0;
    fall_capacitance : 0;
    max_capacitance : 0.503808;
    function : "(!A)";
...
```

```
...
 cell_fall(delay_template_5x5) {
        index_1 ("0.005, 0.0125, 0.025,
0.075, 0.15");
        index_2 ("0.06, 0.18, 0.42, 0.6,
1.2");
        values ( \
          "0.030906, 0.037434, 0.038584,
0.039088, 0.030318", \
          "0.04464, 0.057551, 0.073142,
0.077841, 0.081003", \
          "0.064368, 0.091076, 0.11557,
0.126352, 0.144944", \
          "0.139135, 0.174422, 0.232659,
0.261317, 0.321043", \
          "0.249412, 0.28434, 0.357694,
0.406534, 0.51187");
      }
...
```

LIBERTY (.LIB): INTRODUCTION

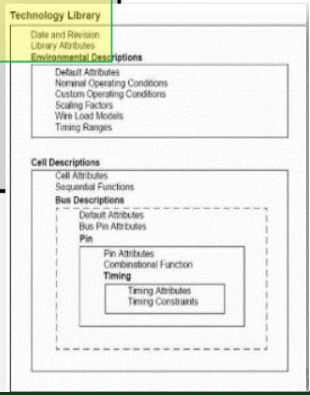
- How do we know the delay through a gate in a logic path?
  - Running SPICE is way too complex.
  - Instead, create a timing model that will simplify the calculation.
- Goal:
  - For every timing arc, calculate:
    - Propagation Delay (t_{pd})
    - Output transition (t_{rise}, t_{fall})
  - Based on:
    - Input net transition (t_{rise}, t_{fall})
    - Output Load Capacitance (C_{load})



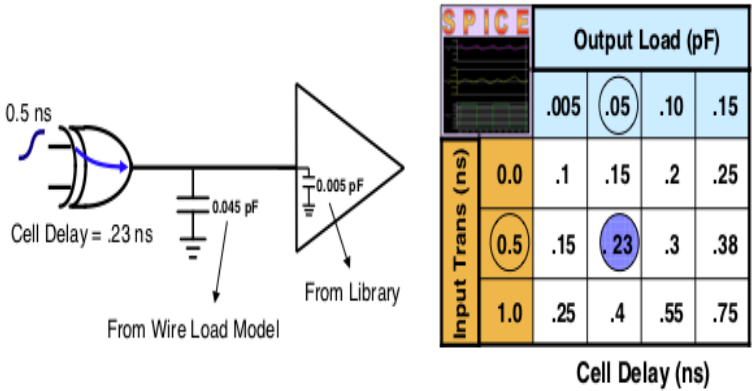
LIBERTY (.LIB): GENERAL

- Timing data of standard cells is provided in the Liberty format.
  - Library:
    - General information common to all cells in the library.
    - For example, operating conditions, wire load models, look-up tables
  - Cell:
    - Specific information about each standard cell.
    - For example, function, area.
  - Pin:
    - Timing, power, capacitance, leakage, functionality, etc. characteristics of each pin in each cell.

```
library (nameoflibrary) {
... /* Library level simple and complex attributes */

/* Cell definitions */
cell (cell_name) {
   ... /* cell level simple attributes */

   /* pin groups within the cell */
   pin(pin_name) {
       ... /* pin level simple attributes */

       /* timing group within the pin level */
       timing(){
            ... /* timing level simple attributes */ }
       ... /* additional timing groups */

   } /* end of pin */
   ... /* more pin descriptions */
} /* end of cell */
... /* more cells */

} /* end of library */
```



LIBERTY (.LIB): TIMING MODELS



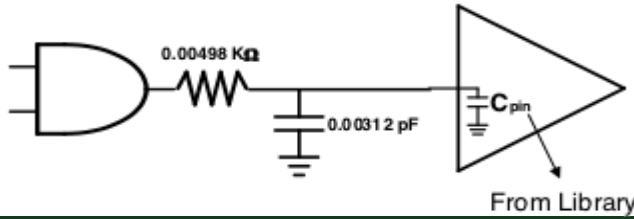

```
lu_table_template(delay_template_5x5) {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  index_1 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
  index_2 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
}
```

```
cell (INVX1) {
  pin(Y) {
    timing() {
      cell_rise(delay_template_5x5) {
        values ( \
          "0.147955, 0.218038, 0.359898, 0.922746, 1.76604", \
          "0.224384, 0.292903, 0.430394, 0.991288, 1.83116", \
          "0.365378, 0.448722, 0.584275, 1.13597, 1.97017", \
          "0.462096, 0.551586, 0.70164, 1.24437, 2.08131", \
          "0.756459, 0.874246, 1.05713, 1.62898, 2.44989"); }
```

LIBERTY (.LIB): WIRE LOAD MODELS

- How do you estimate the parasitics (RC) of a net before placement and routing?
- Wire Load Models estimate the parasitics based on the fanout of a net.

Net Fanout	Resistance KΩ	Capacitance pF
1	0.00498	0.00312
2	0.01295	0.00812
3	0.02092	0.01312
4	0.02888	0.01811



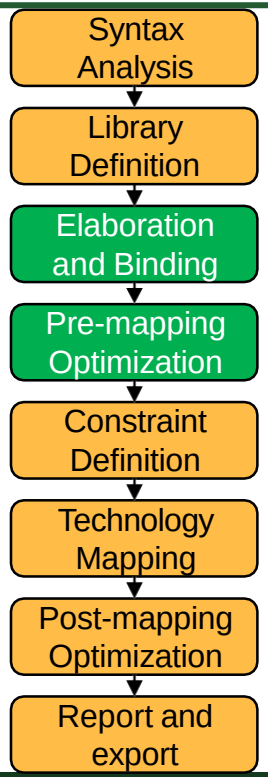

```
library (myLib) {
  wire_load("WLM1")
    resistance: 0.0006 ; // R per unit length
    capacitance: 0.0001 ; // C per unit length
    area : 0.1 ; // Area per unit length
    slope : 1.5 ; // Used for linear extrapolation
    fanout_length(1, 0.002) ; // for fo=1, Lwire=0.002
    fanout_length(2, 0.006) ; // for fo=2, Lwire=0.006
    fanout_length(3, 0.009) ; // for fo=3, Lwire=0.009
    fanout_length(4, 0.015) ; // for fo=4, Lwire=0.015
    fanout_length(5, 0.020) ; // for fo=5, Lwire=0.020
    fanout_length(6, 0.028) ; // for fo=6, Lwire=0.028
  }
} /* end of library */
```

PHYSICAL-AWARE SYNTHESIS

- Due to the lack of accuracy, wireload models lead to very poor correlation between synthesis and post-layout in nanometer technologies.
- Instead, use physical information during synthesis
  - Synopsys calls this “Topographical Mode”
  - Cadence calls this “Physical Synthesis”
- Physical-Aware Synthesis basically runs placement inside the synthesizer to obtain more accurate parasitic estimation:
  - Without a floorplan, just using .lef files
  - After first iterations, import a floorplan .def to the synthesizer.

BOOLEAN MINIMIZATION

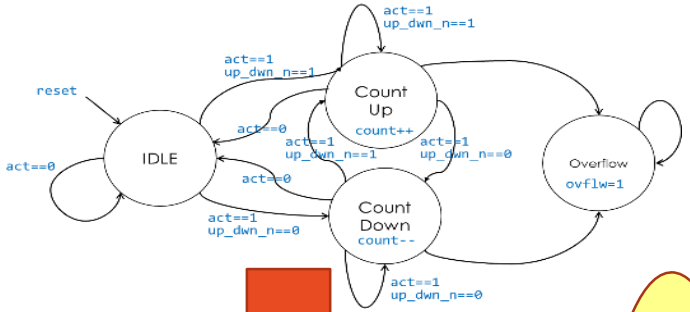
- Mapping to Generics and Libs
- Basics of Boolean Minimization:
  - BDDs, Two-Level Logic, Espresso

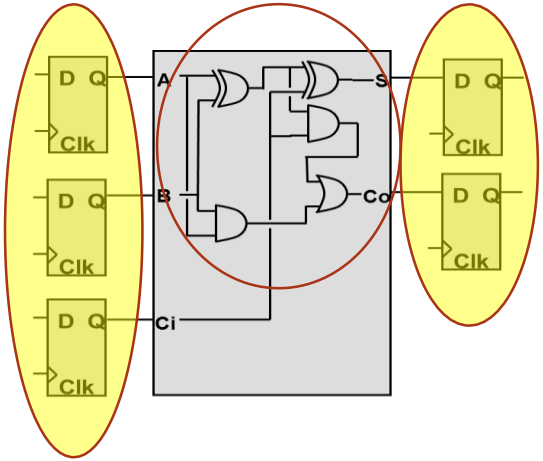


ELABORATION AND BINDING

- During the next step of logic synthesis, the tool:
  - Compiles the RTL into a Boolean data structure (elaboration)
  - Binds the non-Boolean modules to leaf cells (binding), and
  - Optimizes the Boolean logic (minimization).
- The resulting design is mapped to generic, technology independent logic gates.
- This is the core of synthesis and has been a very central subject of research in computer science since the eighties.

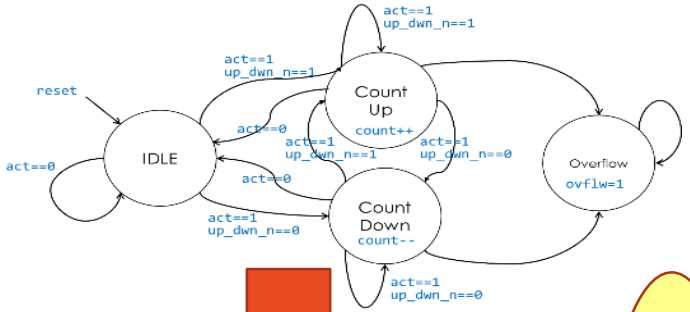
ELABORATION ILLUSTRATED





Inferred Registers

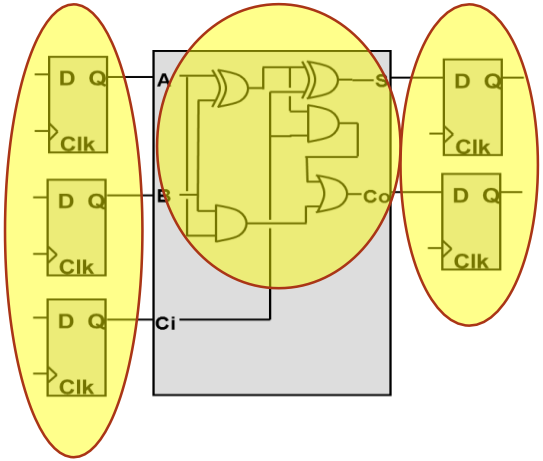
ELABORATION ILLUSTRATED



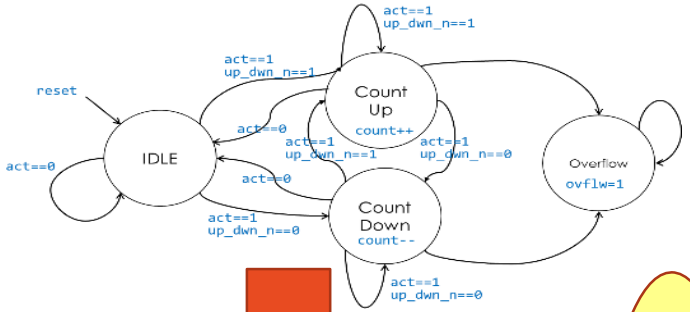
x_1 x_2 x_3	$f(x_1x_2x_3)$
0 0 0	1
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	0
1 0 1	0
1 1 0	1

Boolean Logic

Inferred Registers



ELABORATION ILLUSTRATED

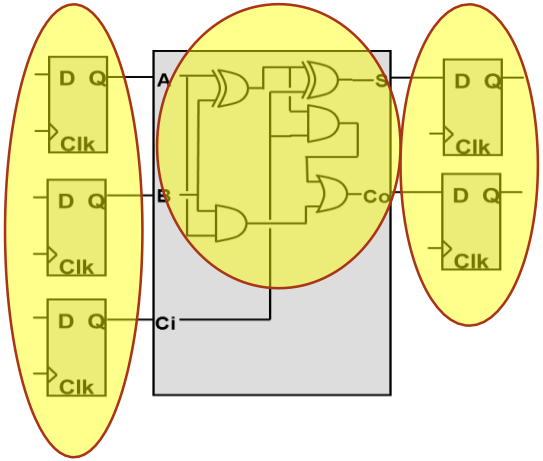


x_1 x_2 x_3	$f(x_1x_2x_3)$
0 0 0	1
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	0
1 0 1	0
1 1 0	1

Boolean Logic

F_1=ACB'+DEF'+A'BCF+…
F_2=C'B'+D'GH'+A'FG'+…
…

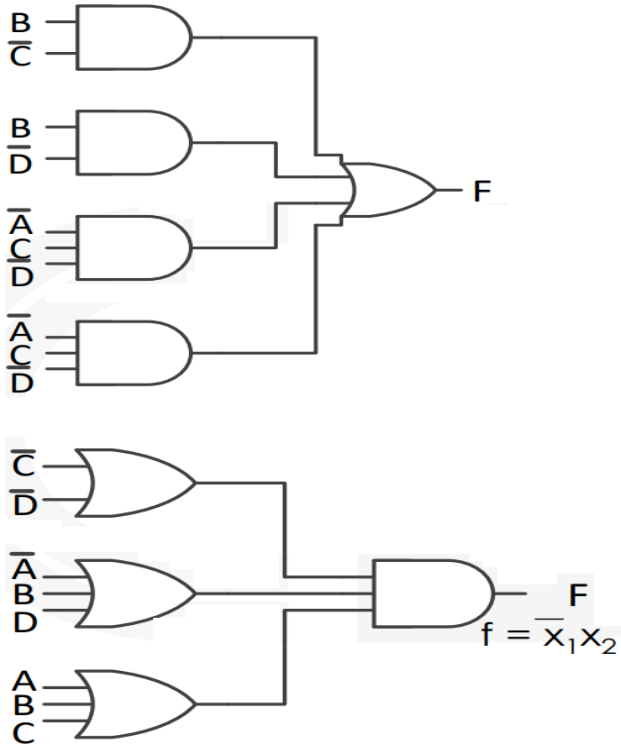
Two-level Logic



Inferred Registers

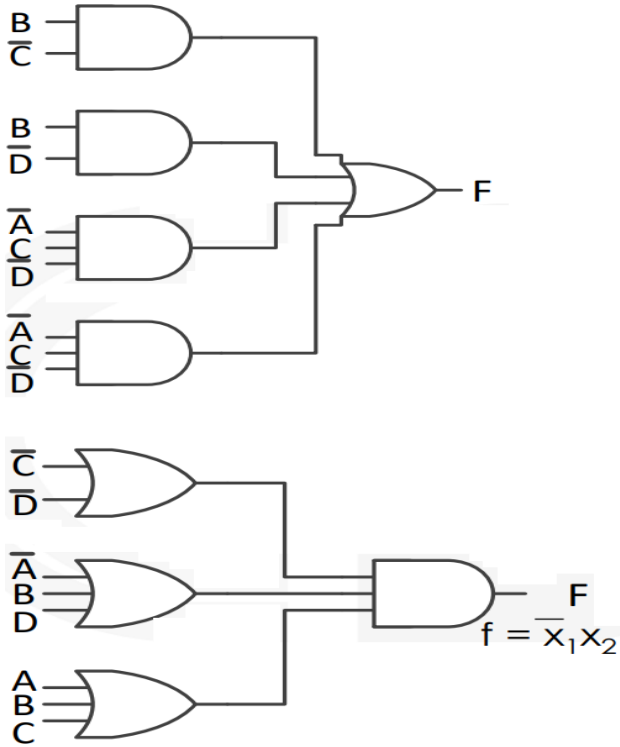
TWO-LEVEL LOGIC

- During elaboration, primary inputs and outputs (ports) are defined and sequential elements (flip-flops, latches) are inferred.
- This results in a set of combinational logic clouds with:
  - Input ports and register outputs are inputs to the logic
  - Output ports and register inputs are the outputs of the logic
  - The outputs can be described as Boolean functions of the inputs.
  - The goal of Boolean minimization is to reduce the number of literals in the output functions.



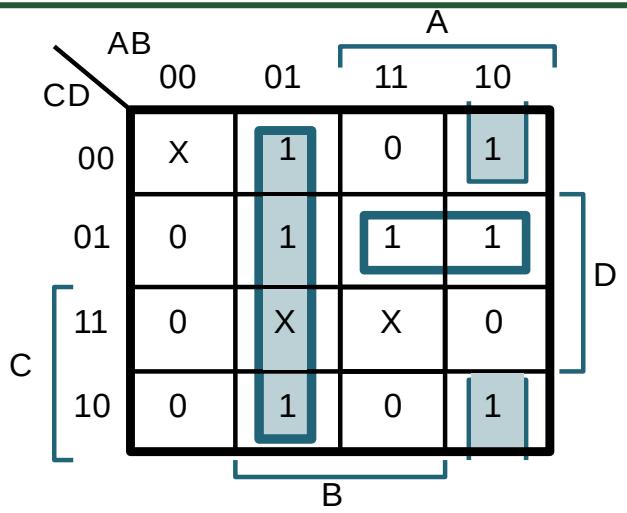
TWO-LEVEL LOGIC

- Many different data structures are used to represent the Boolean functions:
  - Truth tables, cubes, Binary Decision Diagrams, equations, etc.
  - A lot of the research was developed upon SOP or POS representation, which is better known as "Two-Level Logic"



TWO-LEVEL LOGIC MINIMIZATION

- In previous years, we learned about Karnaugh maps:
  - For n inputs, the map contains 2^n entries
  - Objective is to find the minimum prime cover
- However…
  - Difficult to automate (NP-complete)
  - Number of cells is exponential (<6 variables)
- A different approach is the Quine-McCluskey method
  - Easy to implement in software
  - BUT computational complexity too high
- Some Berkeley students fell asleep while solving a Quine-McCluskey exercise.
- They needed a shot of **Espresso**.



ESPRESSO HEURISTIC MINIMIZER

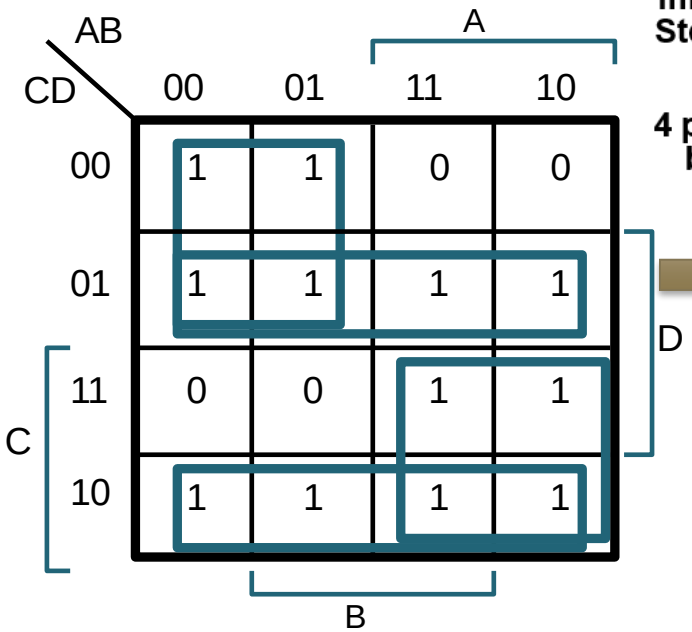
Start with an SOP solution.

- Reduce
  - The cubes in the cover are reduced in size.
- Expand
  - Make each cube as large as possible without covering a point in the OFF-set.
  - Increases the number of literals (worse solution)
- Irredundant
  - Throw out *redundant* cubes.
  - Remove smaller cubes whose points are covered by larger cubes.
- In general, the new cover will be different from the initial cover.
  - "expand" and "irredundant" steps can possibly find out a new way to cover the points in the ON-set.
  - Hopefully, the new cover will be smaller.

```
ESPRESSO(F) {
  do {
      reduce(F);
      expand(F);
      irredundant(F);
  } while (fewer terms in F);
  verify(F);
}
```

ESPRESSO EXAMPLE

Starting SOP Form:



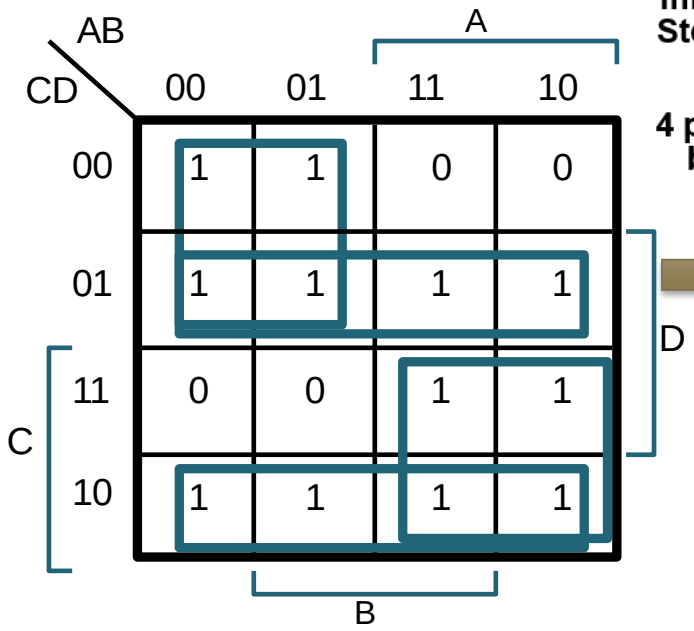
Initial Set of Primes found by Steps1 and 2 of the Espresso Method

4 primes, irredundant cover, but not a minimal cover!

$$f = \overline{A}\,\overline{C} + \overline{C}D + AC + C\overline{D}$$

ESPRESSO EXAMPLE

Starting SOP Form:



$$f = \bar{A}\bar{C} + \bar{C}D + AC + C\bar{D}$$

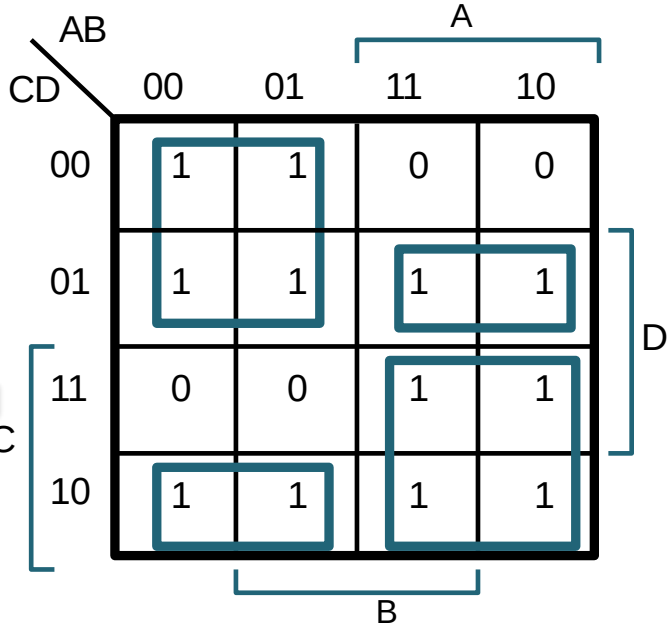
Initial Set of Primes found by Steps1 and 2 of the Espresso Method

4 primes, irredundant cover, but not a minimal cover!

Result of **REDUCE**: Shrink primes while still covering the ON-set

Choice of order in which to perform shrink is important

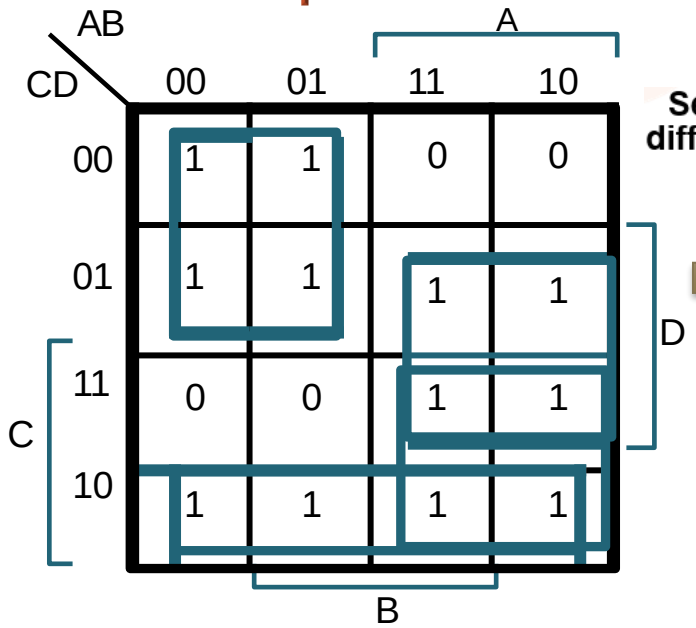
Reduce:



$$f = \bar{A}\bar{C} + A\bar{C}D + AC + \bar{A}C\bar{D}$$

ESPRESSO EXAMPLE

Expand:

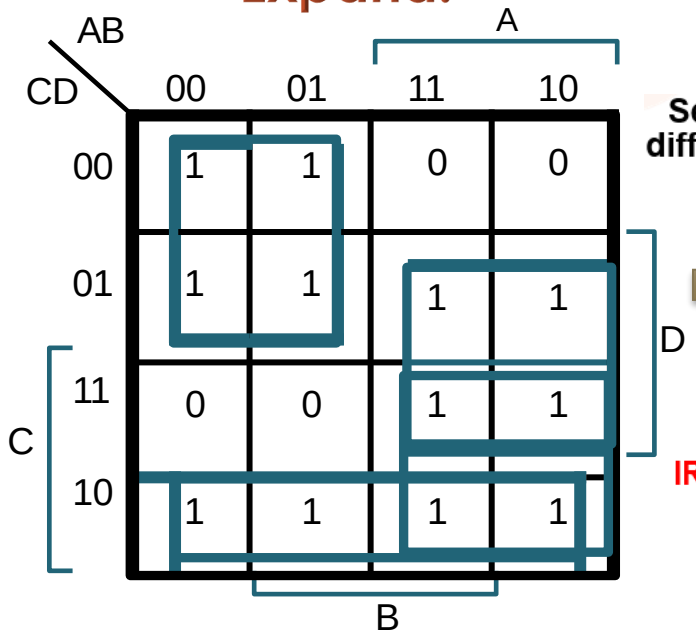


Second EXPAND generates a different set of prime implicants

$$f = \overline{A}\overline{C} + AD + AC + C\overline{D}$$

ESPRESSO EXAMPLE

Expand:



$$f = \overline{A}\overline{C} + AD + AC + C\overline{D}$$

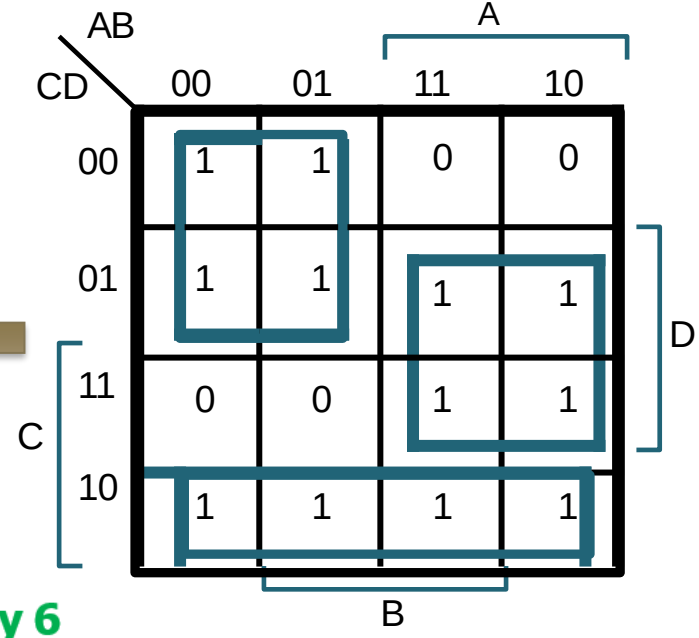
Second **EXPAND** generates a different set of prime implicants

IRREDUNDANT COVER found by final step of espresso

Only three prime implicants!

Only 6 literals!

Irredundant:



$$f = \overline{A}\overline{C} + AD + C\overline{D}$$

MULTI-LEVEL LOGIC MINIMIZATION

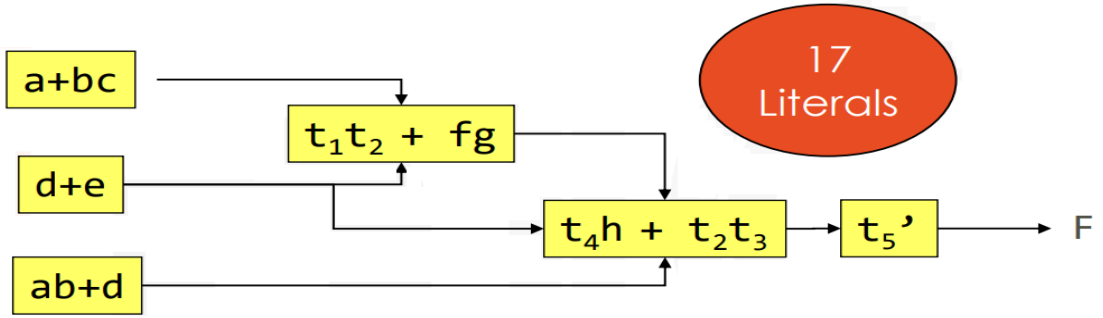
- Two-level logic minimization has been widely researched and many famous methods have come out of it.
  - However, often it is better and/or more practical to use many levels of logic (remember logical effort?).
- Therefore, a whole new optimization regime, known as multi-level logic minimization was developed.
  - We will not cover multi-level minimization in this course, however, you should be aware that the output of logic minimization will generally be multi-level and not two-level.

MULTI-LEVEL LOGIC MINIMIZATION

For example:

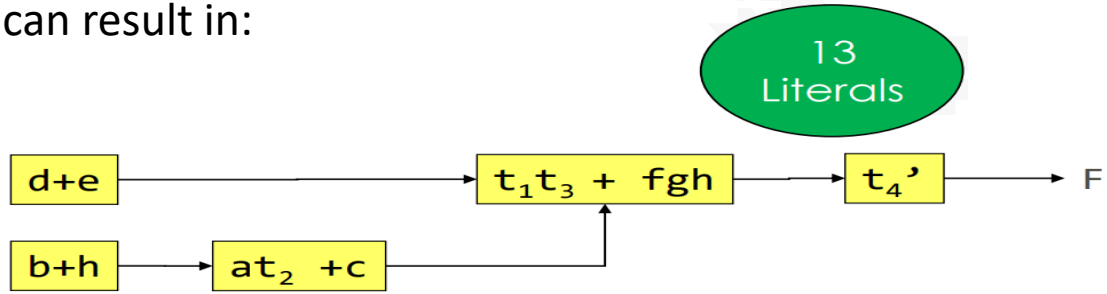
- Given the following logic set:

$t_1 = a + bc;$
$t_2 = d + e;$
$t_3 = ab + d;$
$t_4 = t_1t_2 + fg;$
$t_5 = t_4h + t_2t_3;$
$F = t_5';$



- Multi-level Logic Minimization can result in:

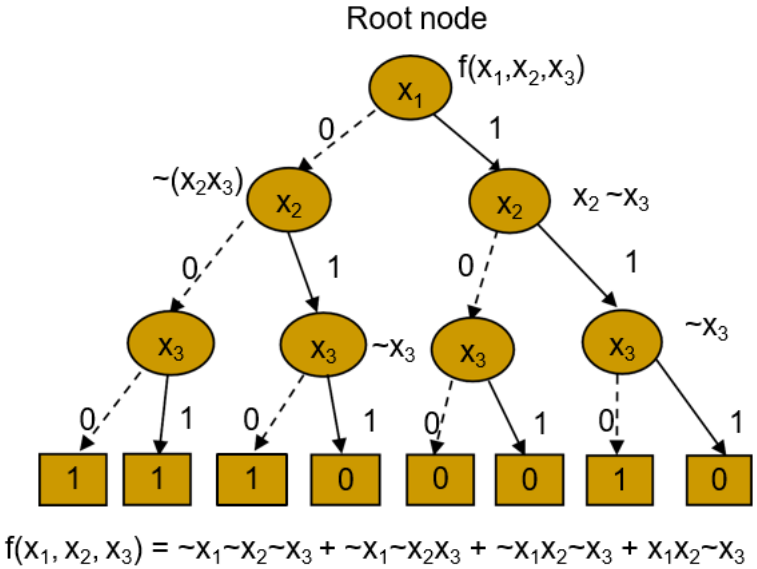
$t_1 = d + e;$
$t_2 = b + h;$
$t_3 = at_2 + c;$
$t_4 = t_1t_3 + fgh;$
$F = t_4';$



BINARY DECISION DIAGRAMS (BDD)

- BDDs are DAGs (directed acyclic graph) that represent the truth table of a given function

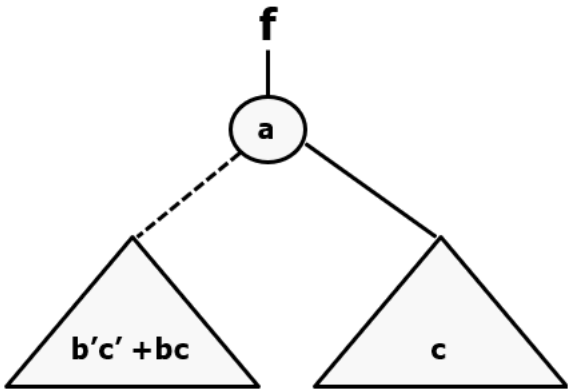
x_1 x_2 x_3	$f(x_1x_2x_3)$
0 0 0	1
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	0
1 0 1	0
1 1 0	1



BINARY DECISION DIAGRAMS (BDD)

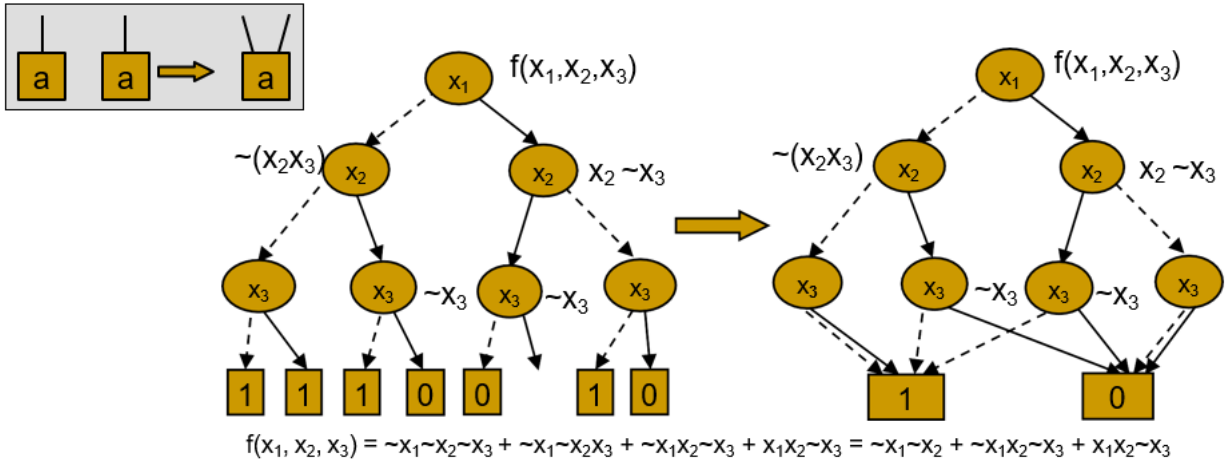
- The Shannon Expansion of a function relates the function to its cofactors:
  - Given a Boolean function: $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$
  - Positive cofactor: $f_i^{\,1} = f(x_1, x_2, \ldots, 1, \ldots, x_n)$
  - Negative cofactor: $f_i^{\,0} = f(x_1, x_2, \ldots, 0, \ldots, x_n)$
- Shannon's expansion theorem states that:
  - $f = x_i' \, f_i^{\,0} + x_i \, f_i^{\,1}$
  - $f = (x_i + f_i^{\,0})(x_i' + f_i^{\,1})$
- This leads to the formation of a BDD:
  - Example:

$$
\begin{aligned}
f &= ac + bc + a'b'c' \\
&= a' (b'c' + bc) + a (c + bc) \\
&= a' (b'c' + bc) + a (c)
\end{aligned}
$$



REDUCED ORDERED BDD (ROBDD)

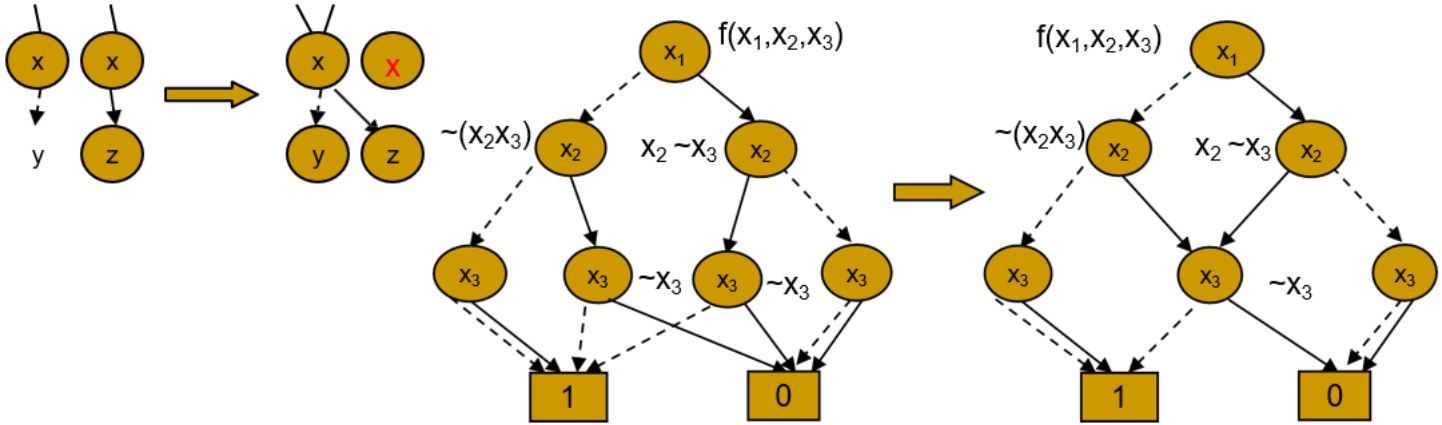
- BDDs can get very big.
- So let's see if we can provide a reduced representation.
- Reduction Rule 1: Merge equivalent leaves



$f(x_1, x_2, x_3) = \sim x_1 \sim x_2 \sim x_3 + \sim x_1 \sim x_2 x_3 + \sim x_1 x_2 \sim x_3 + x_1 x_2 \sim x_3 = \sim x_1 \sim x_2 + \sim x_1 x_2 \sim x_3 + x_1 x_2 \sim x_3$

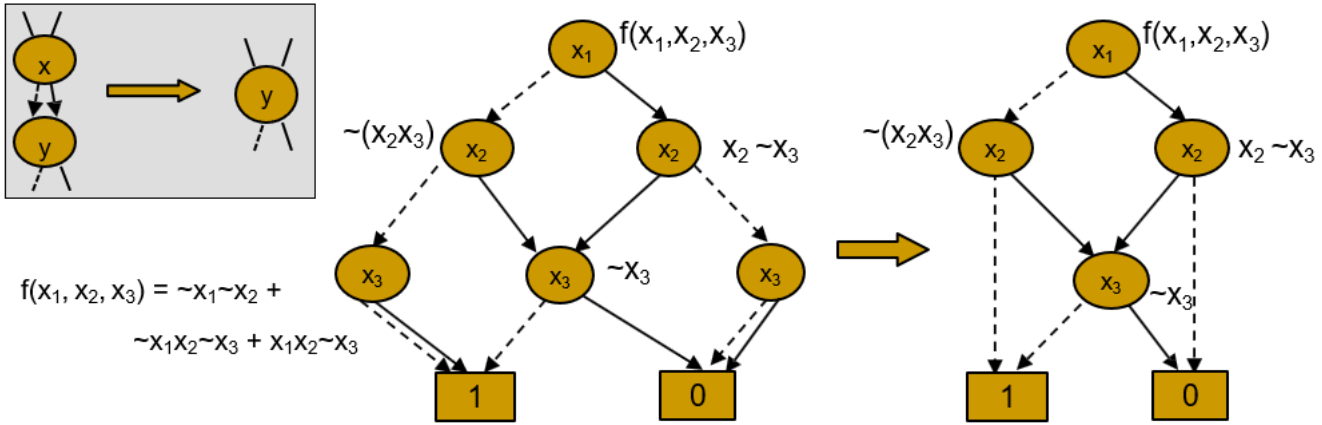
REDUCED ORDERED BDD (ROBDD)

- BDDs can get very big.
- So let's see if we can provide a reduced representation.
- Reduction Rule 2: Merge isomorphic nodes



REDUCED ORDERED BDD (ROBDD)

- BDDs can get very big.
- So let's see if we can provide a reduced representation.
- Reduction Rule 3: Eliminate Redundant Tests



$f(x_1, x_2, x_3) = \sim x_1 \sim x_2 + \sim x_1 x_2 \sim x_3 + x_1 x_2 \sim x_3$

BDD CONTINUED

- Some benefits of BDDs:
  - Check for tautology is trivial.
    - BDD is a constant 1.
- Complementation.
  - Given a BDD for a function f, the BDD for f' can be obtained by interchanging the terminal nodes.
- Equivalence check.
  - Two functions f and g are equivalent if their BDDs (under the same variable ordering) are the same.
- An Important Point:
  - The size of a BDD can vary drastically if the order in which the variables are expanded is changed.
  - The number of nodes in the BDD can be exponential in the number of variables in the worst case, even after reduction.

Thank you!