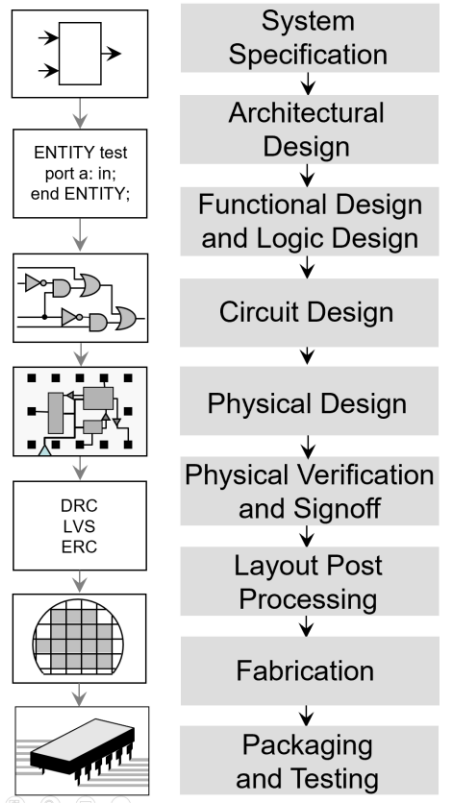
Chip Level Integration

David King

IEEE EPS @ EPS

How are chips designed?

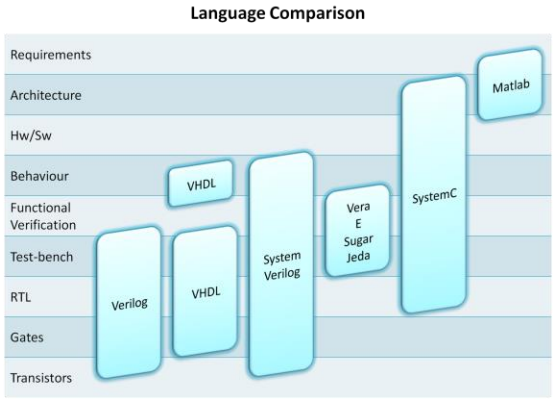
- Today we'll be focusing in on
  - Functional Design and Logic Design
  - Physical Design

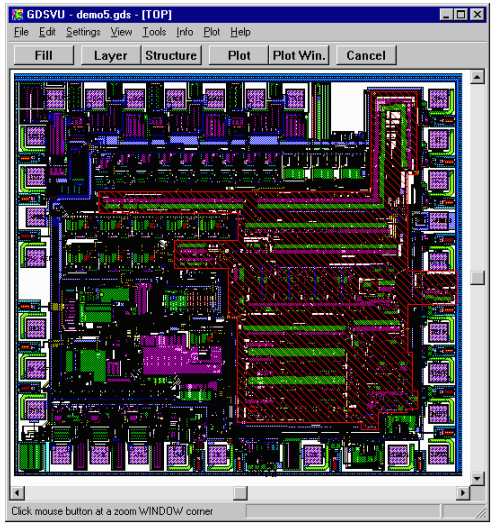


Hardware Description Language

We need a language that knows how to model digital circuits.

- How can we describe hardware behavior without fully implementing it
- How can we describe hardware behavior without overwhelming ourselves



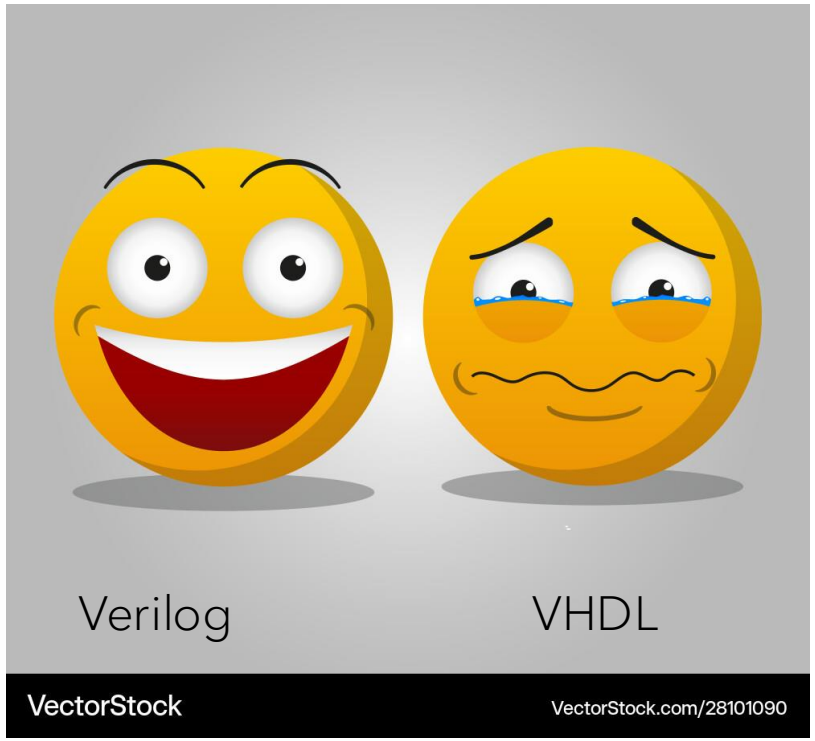


Today we're using Verilog

We're going to do a short primer on Verilog

We will not be doing a deep dive

This section is to help you understand how HDL differs from a typical programming language.



This is my personal opinion

Data Types

- Verilog uses for valued logic
- Reasons for "X":
  - Uninitialized value
  - Conflicting drivers / assignments

Value	Meaning
0	Logic zero, "low"
1	Logic one, "high"
Z or ?	High impedance (tri-state buses)
X	Unknown value

Numeric Constants

By default, constant values are specific with specific width and radix.

All values by default are unsigned.

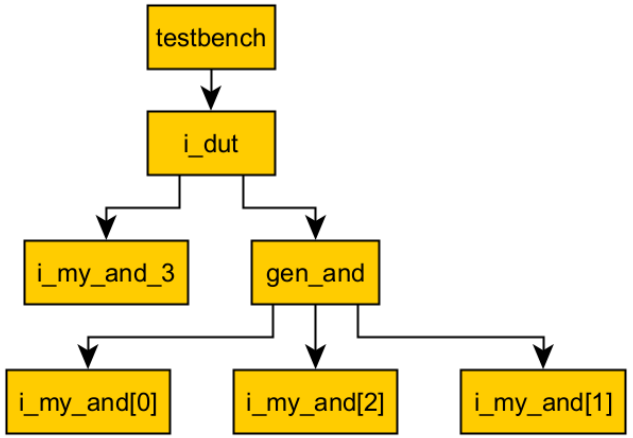
Value	Description
123	Default: Decimal, unspecified width
‘d123	‘d = Decimal radix
‘h7B	‘h = Hexadecimal radix
‘o173	‘o = Octal radix
‘b111_1011	‘b = binary radix, “_” are ignored
‘hxx	Can include X, Z, or ? In non-decimal constants
16’d5	16 bit constant with a decimal assignment of 5.
11’h1	11-bit constant with a hexadecimal assignment

Modules

Modules are the primary way we abstract logical primitives and other modules.

All modules declare directional ports, which are used for inter-module communication.

```
module mux2(input a,b,sel, output z,zbar);
        assign z = sel ? b : a;assign zbar = ~z;
endmodule
```



Wires and Registers

We need to declare all named wires (nets)

We can also produce buses, indexed collections of wires.

If you want a register just replace **wire** with **reg**

Net	Description
wire a, b, z;	Three 1-bit busses
wire [31:0] memdata;	A 32-bit bus
wire [7:0] b1, b2, b3, b4	Four 8-bit busses
wire [W-1:0] input;	Parameterized bus

Wire	Reg
Use it if you want a continuous assignment	• Holds a value across procedural assignments • Used for sequential logic

Wires vs Registers

Feature	reg	wire
Purpose	Holds values assigned within procedural blocks (always, initial)	Used for continuous assignments and connecting module outputs/inputs
Continuous Assignment	Not allowed	Allowed using assign statements
Procedural Assignment	Can be assigned in always or initial blocks	Cannot be assigned in procedural blocks
Non-blocking Assignments	Allowed (for sequential logic)	Not allowed
Blocking Assignments	Allowed (for combinational logic)	Not allowed
Synthesis	May synthesize to flip-flops, latches, or combinational logic depending on use	Synthesizes to combinational connections only

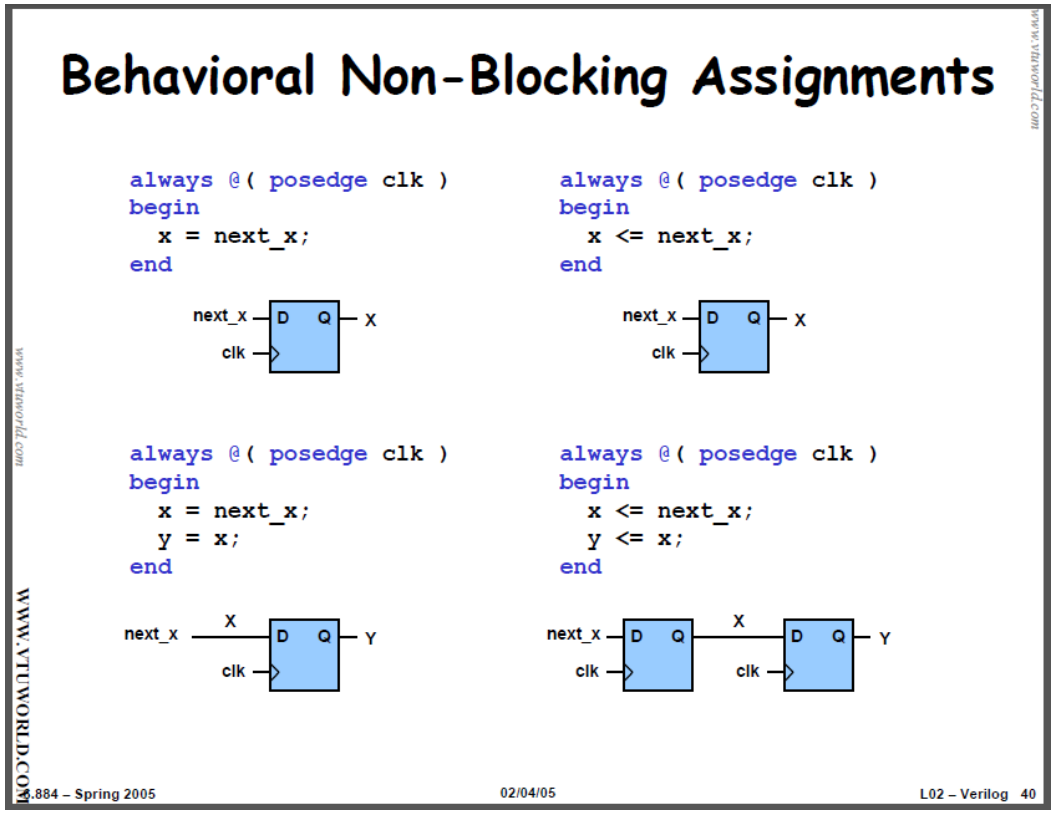
Assignment Types

Blocking: Values assigned sequentially in order

- Used in combinatoric logic

Non-Blocking: Values assigned in parallel

- Used in sequential logic



Sensitivity Lists

There is a keyword called **always**

Ensures that logic executes appropriately to signal change in a module.

```
always @(posedge clk or negedge reset)
begin
  if (!reset)
    q <= 0;
  else
    q <= d;
end
```

Operators

Logical operators similar to other programming languages

Verilog-Specific:

- Concatenation
- Replication

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Abstraction and Hierarchy

Verilog is capable of operating on different levels of abstraction

Modules are instantiated hierarchically allowing for another form of abstraction

Structural Level

```
module full_adder (input a, input b, input cin, output sum, output cout);
    assign sum = a ^ b ^ cin;       // XOR for sum
    assign cout = (a & b) | (cin & (a ^ b)); // Carry-out calculation
endmodule

module adder_structural (input [3:0] a, input [3:0] b, output [4:0] sum);
    wire c1, c2, c3;

    // Instantiate 1-bit adders to build the 4-bit adder
    full_adder fa0 (a[0], b[0], 1'b0, sum[0], c1);
    full_adder fa1 (a[1], b[1], c1, sum[1], c2);
    full_adder fa2 (a[2], b[2], c2, sum[2], c3);
    full_adder fa3 (a[3], b[3], c3, sum[3], sum[4]);
endmodule
```

Dataflow Level

```
module adder_dataflow (input [3:0] a, input [3:0] b, output [4:0] sum);
    assign sum = a + b;  // Dataflow description of addition
endmodule
```

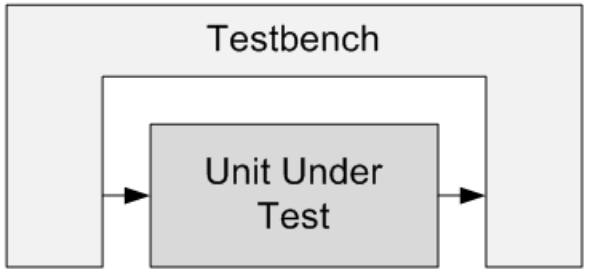
Test Benches and Verification

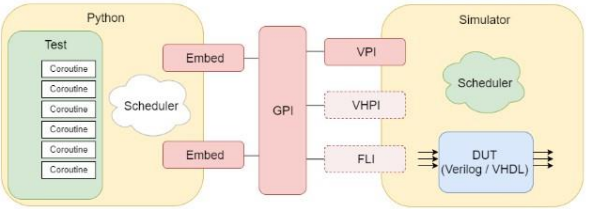
Hardware verification is crucial in ensuring your behavioral model is accurate.

Pass inputs, get outputs, compare with expected outputs.

Add randomness to check all cases.

Test bench can be made in the HDL or in python through CoCoTB





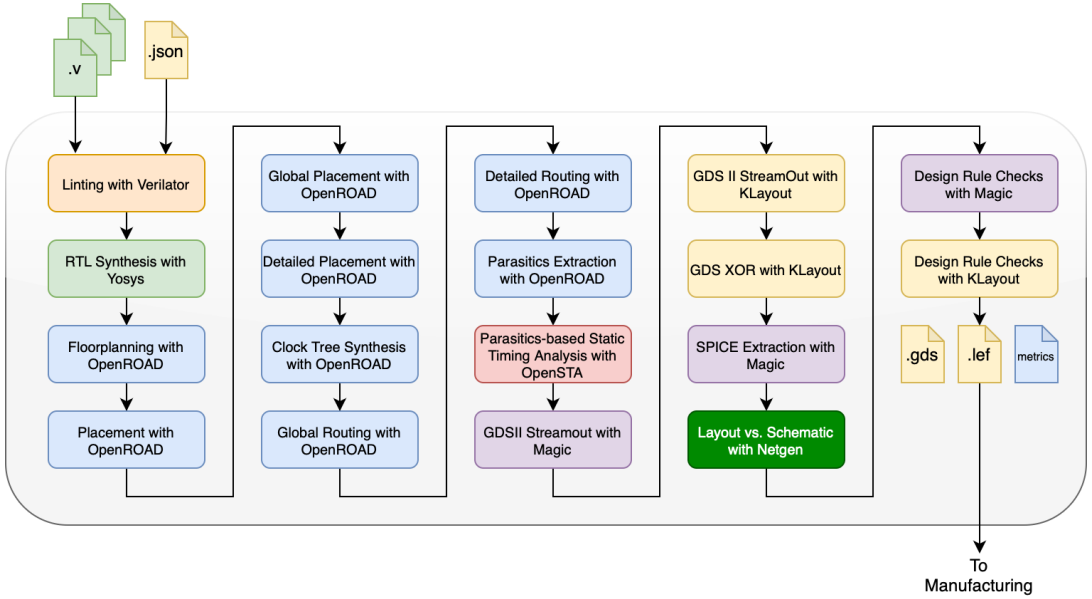
OpenLANE

OpenLANE is an RTL to GDSII flow made up of many open-source tools.

RTL: Register transfer level

GDSII: Graphic Design System 2

It turns hardware models into photomasks, which is called hardening.



Stage 1 - Linting

Objective: Check HDL syntax for errors.

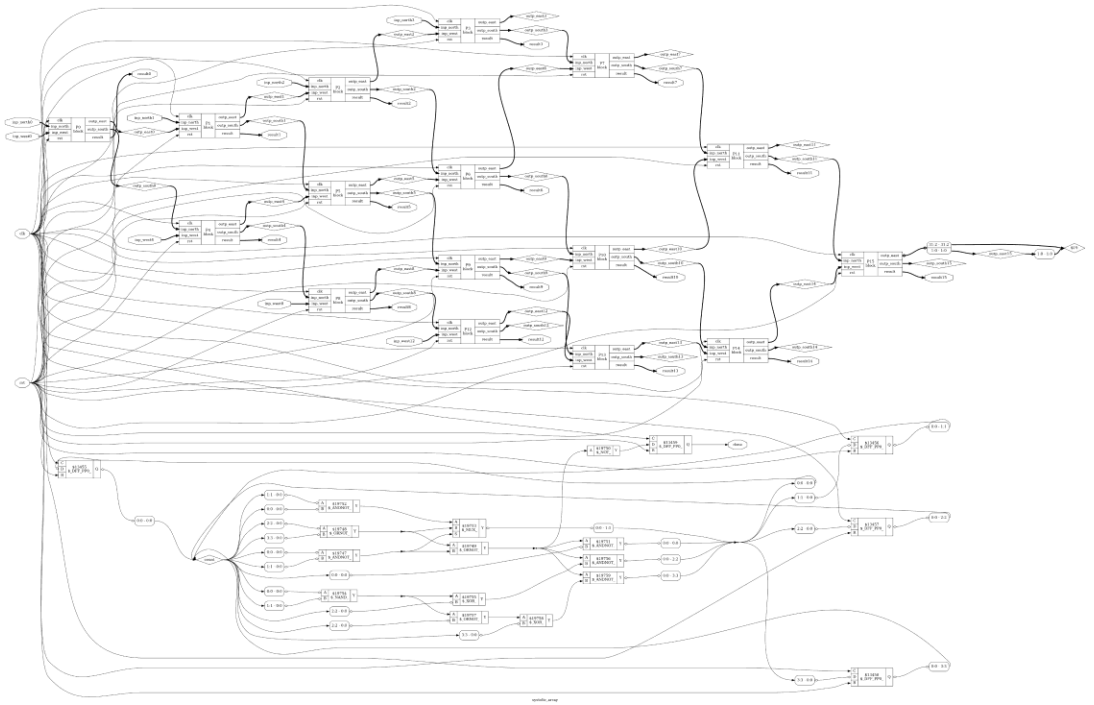
- Ensures code follows best practices
- Ideally this shouldn't be a stage, all HDL must be verified before you even think of hardening.

Icarus Verilog

Stage 2 - Synthesis

Objectives:

1. Create netlist
2. Do static timing analysis on created netlist



Stage 2 – Synthesis - Yosys

We want to bring the abstraction down as much as possible.

Acts as starting point for defining:

1. Static Timing Analysis
2. Floor plan
3. Placement
4. Routing

Behavioral:

```
assign Out  = S? B:A;
```

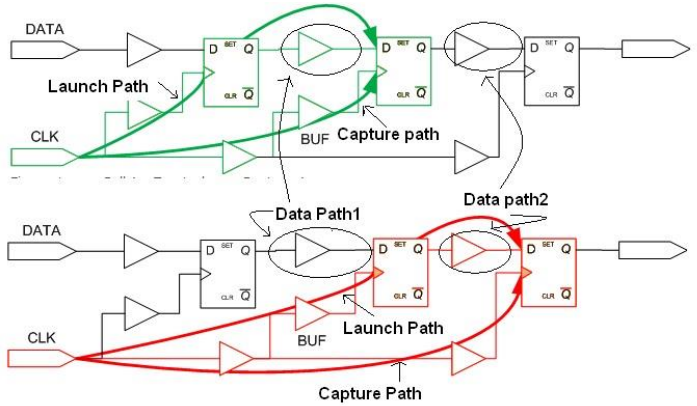
Netlist/Structural:

```
module mux2to1 (
                input S, A, B,
                output Out
);
                wire S_, Ans_, Bns;
                not (S_,S);
                and (AnS_, A, S_);
                and (BnS, B, S);
                or (Out, AnS_, BnS);
endmodule
```

Stage 2 – Synthesis - OpenSTA

Takes Verilog Netlist and performs Static Timing Analysis.

- We don't know where anything is
- We must make estimations with wire load models
  - Global WLM
  - Hierarchical WLM
  - Fan-Out WLM



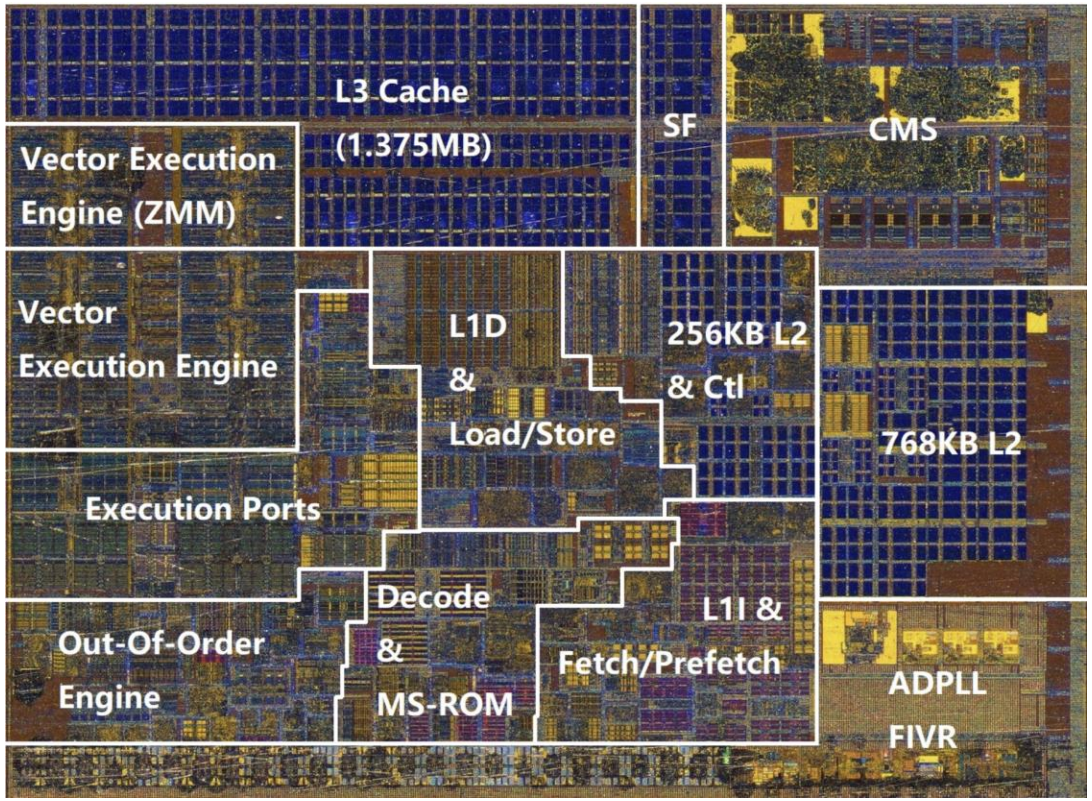
Two Types:

- Single Corner
- Multi Corner

Stage 3 – Floor Planning

Objectives:

1. Define chip dimensions
2. Define placement rows and route tracks
3. Place input output ports.
4. Generate power distribution network
5. Insert well tap and end cap cells



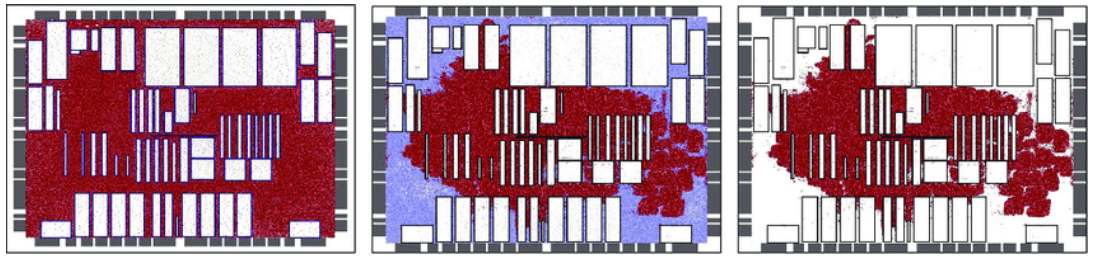
Building Blocks

Standard Cells:

- Smallest functional blocks used to create digital circuits (inverters, AND, OR, flip-flops)
- They are predefined in the standard cell library or PDK.

Macros:

- Pre-designed blocks with fixed functionality that are already hardened.

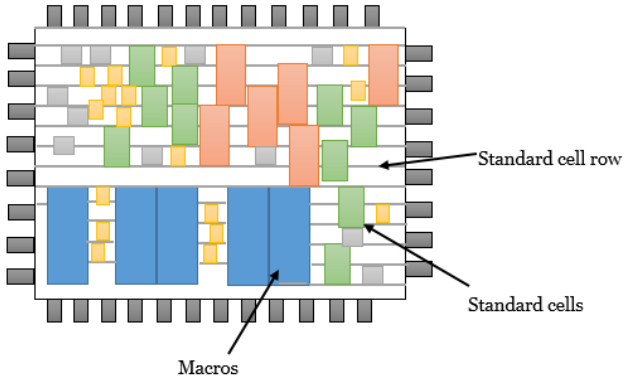


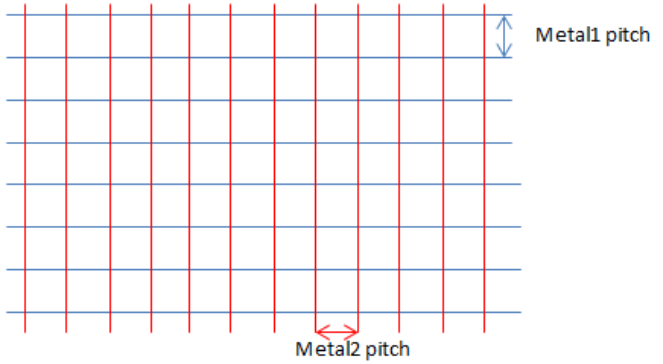
Stage 3 – Floor Planning – Row and Track Creation

Placement Rows: Defines size and possible locations of macros and standard cells

Routing Tracks: Defines grid at which interconnects can be placed

Typically defined in the PDK.



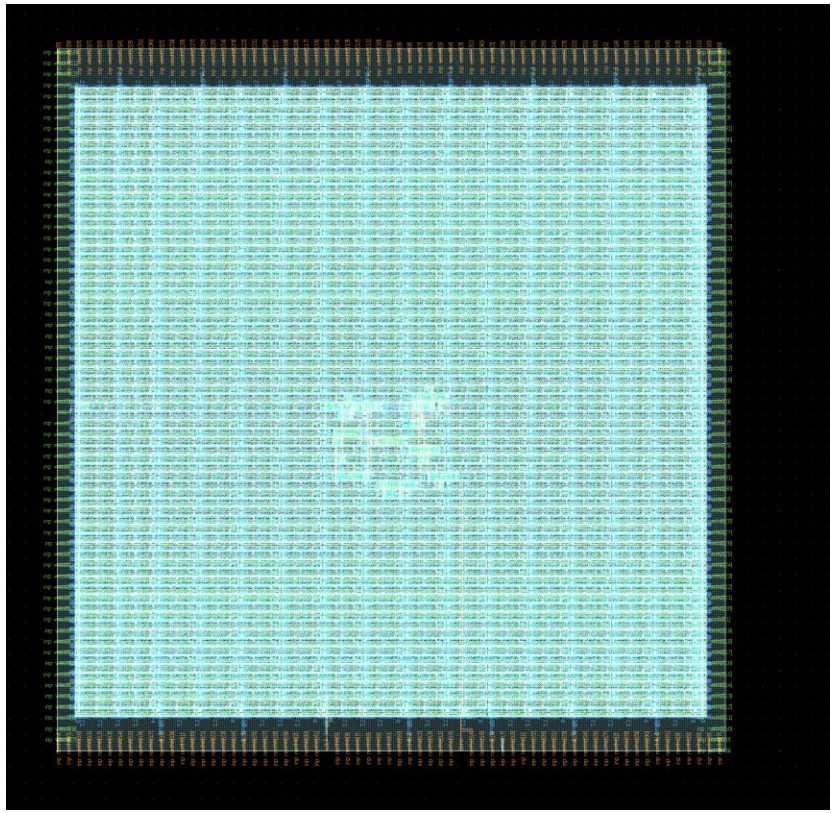


Stage 3 – Floor Planning – Place IO

Space around edges of CORE_AREA is reserved for IO

Typically a floor planning utility randomly assigns IO

If you want specific locations, use a DEF file to define this.

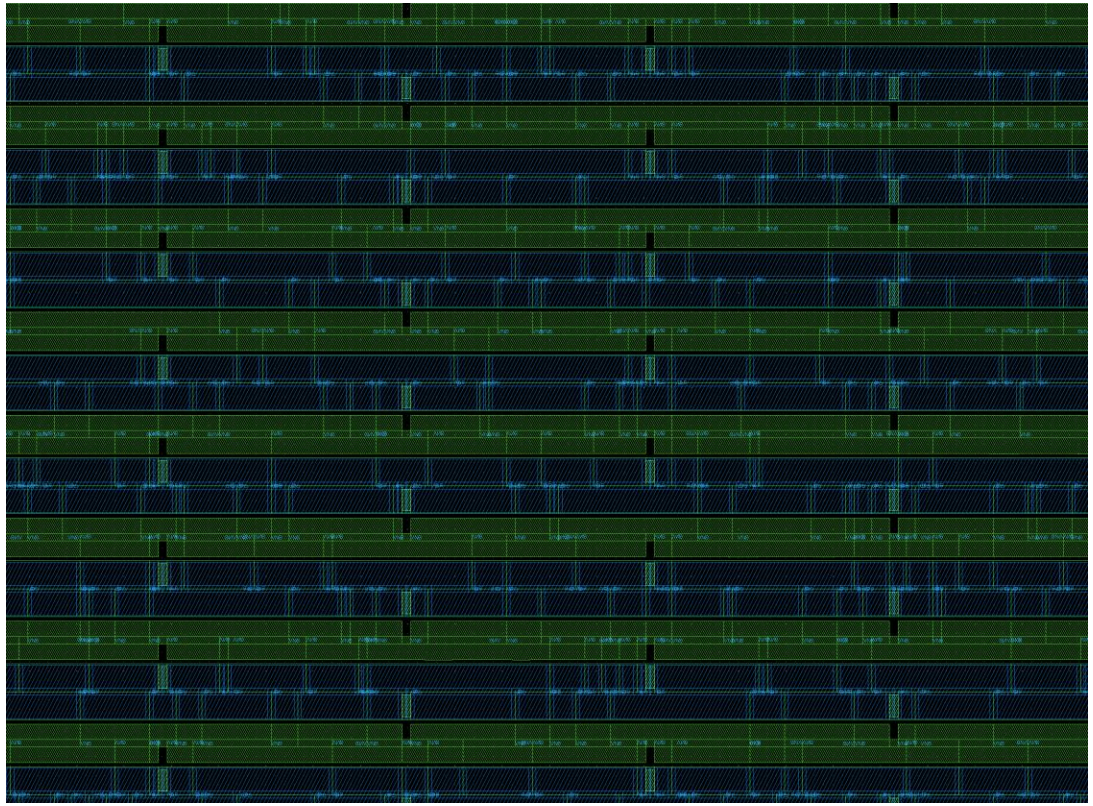


Stage 3 – Floor Planning – Power Distribution Network

Aligned with the placement rows

Wider pitch to allow for more current carrying capacity

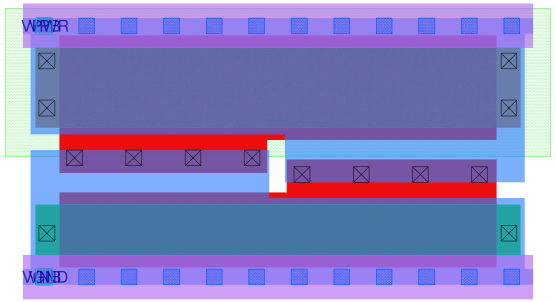
Diodes placed between V_dd and V_ss to prevent ESD.

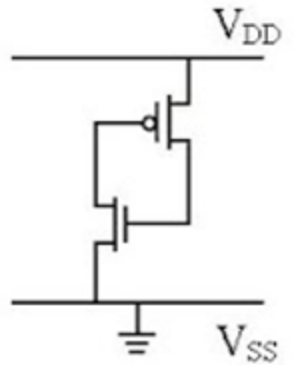


Stage 3 – Floor Planning – Well Tap and Cap Cells

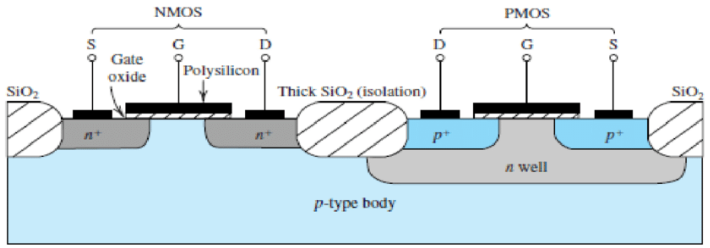
DeCap Cells:

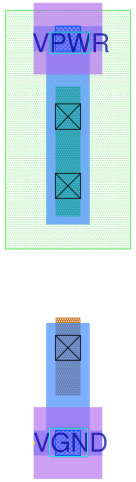
- Electrically stabilize power rails
- Fill in empty space maintaining density

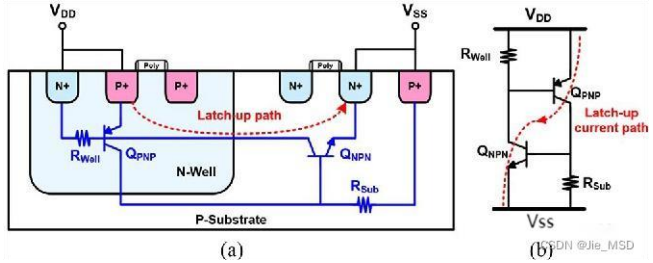




Well Tap Cell:





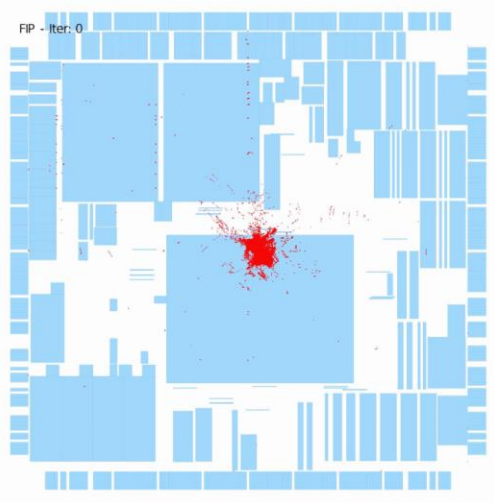


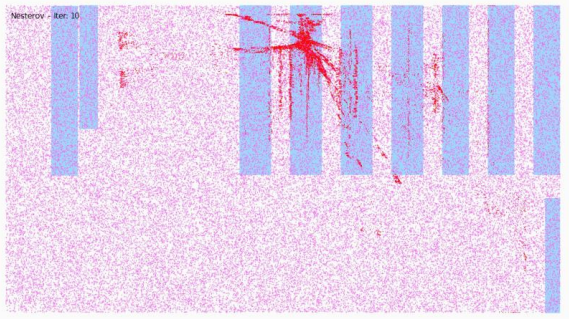
Stage 4 - Placement

Objectives:

1. Find optimal location for standard cells while considering:
    1. Area, timing, and power optimizations
    2. Route Feasibility
    3. Minimal timing DRCs
    4. Minimal cell density and pin density

Typically use Nesterov's method to find optimal solution. Too bad I have no idea how it works.

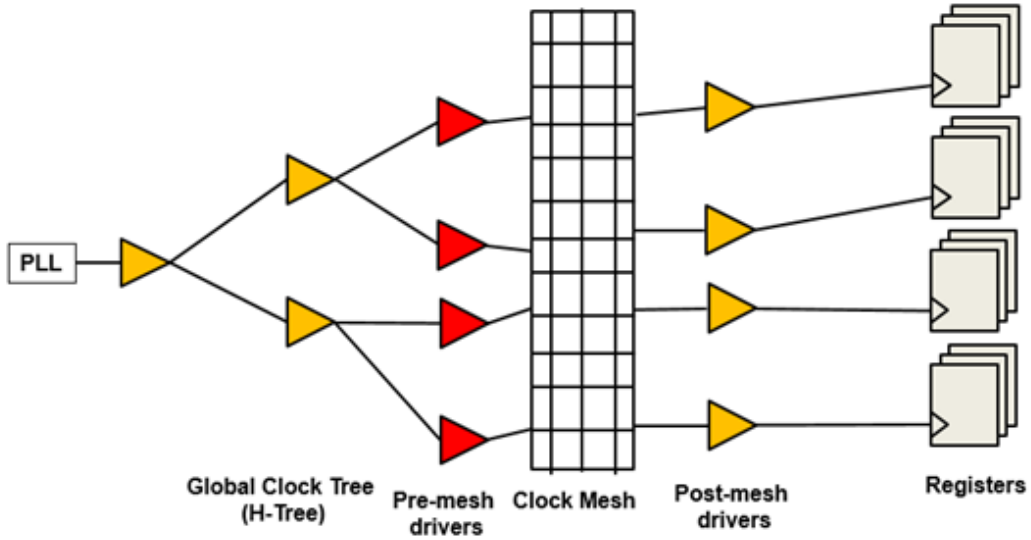




Stage 5 – Clock Tree Synthesis

Objective: Design and optimize the clock distribution network to achieve:

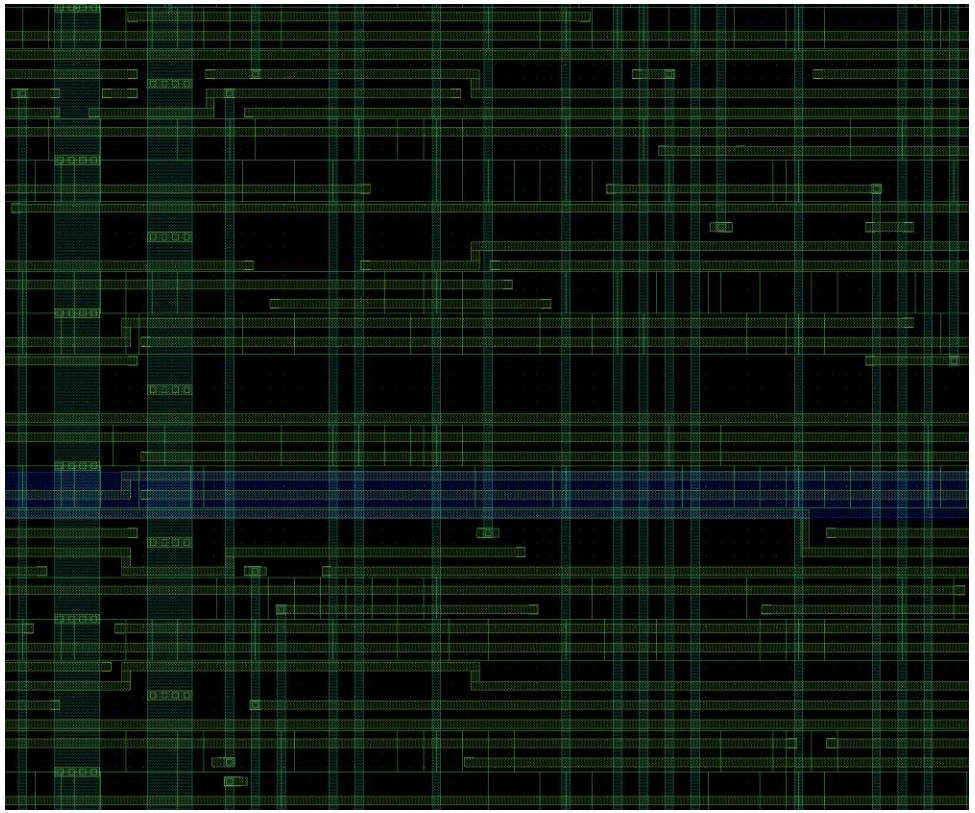
1. Minimal area usage by clock repeaters while maintaining low clock skew
2. Acceptable clock latency and smooth clock transition times
3. Efficient power consumption



Stage 6 - Routing

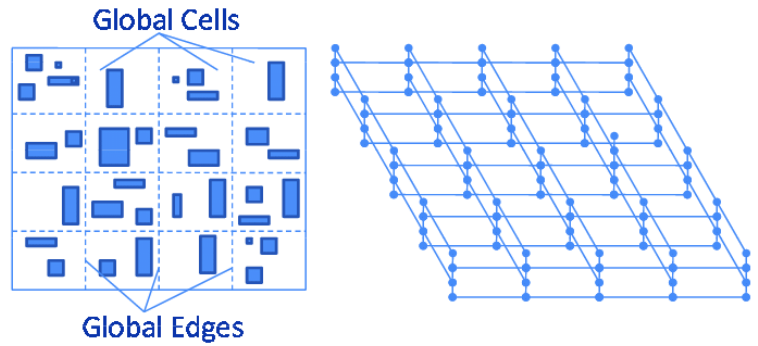
Objectives:

1. Create routing network to connect standard cells
2. Perform SPEF extraction

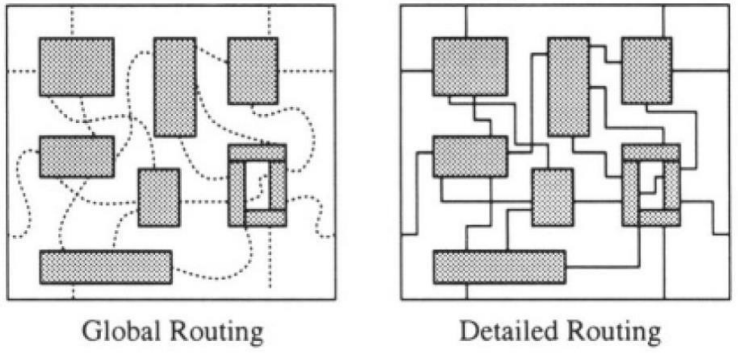


Stage 6 – Routing – Routing

Typically use Steiner trees for fast route algorithm.



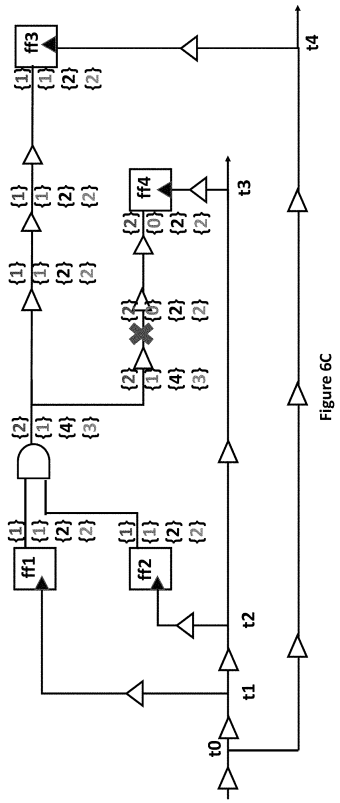
Global & Detailed Routing



Stage 6 – Routing – SPEF Extraction

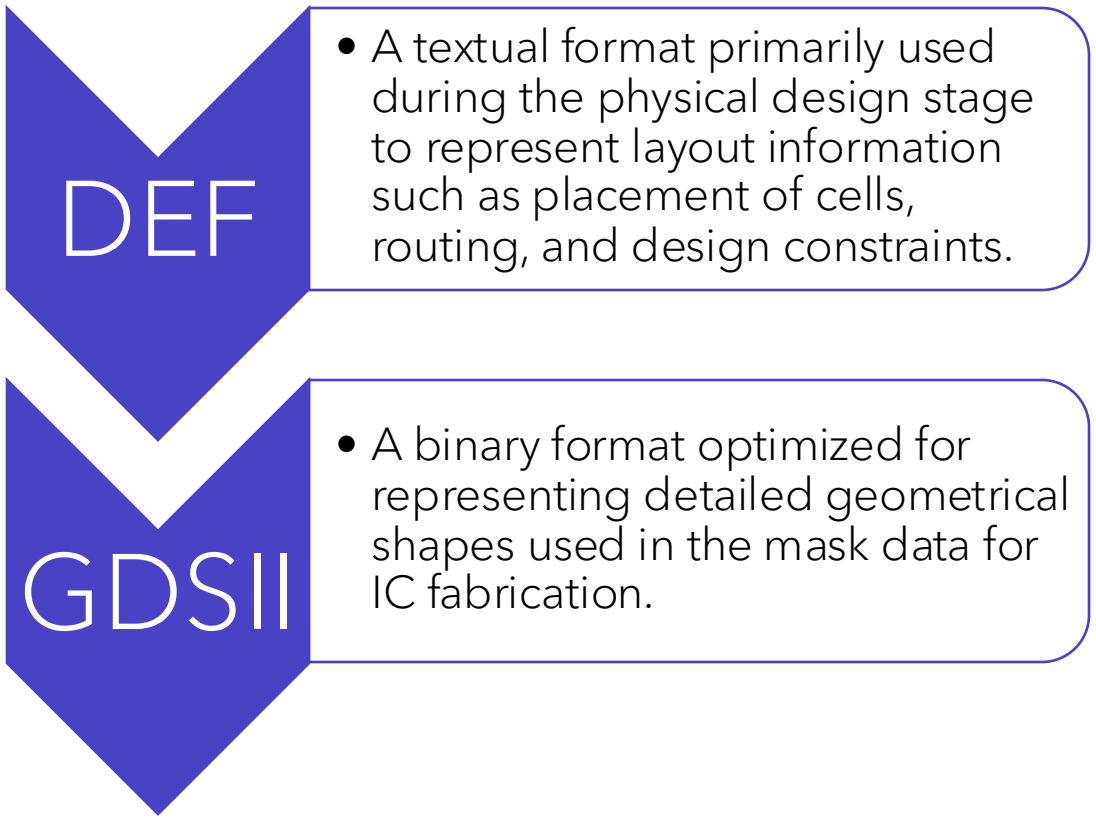
Stands for Standard Parasitic Exchange Format

Once parasitics are determined, multi-corner STA is performed.



Stage 7 – Tape Out

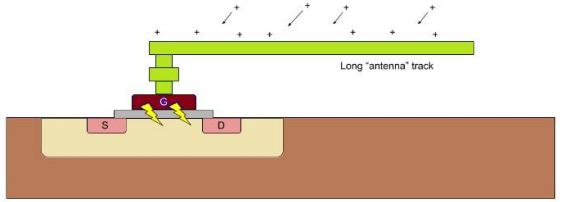
Objective: Convert routed def to GDSII layout.

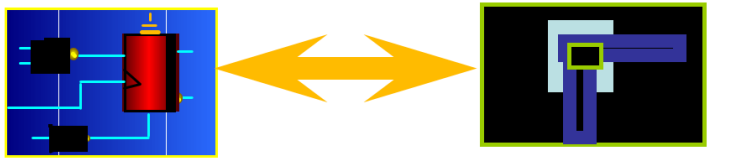


Stage 8 - Signoff

Objectives:

1. Perform DRC checks:
    1. Are the cells or macros too far or too close together?
    2. Is the wire pitch and density too high or too low?
    3. Are layers aligned and overlapped within limit.
2. Perform Antenna checks:
    1. Are there any long metal wires where charge can build up?
3. Perform LVS checks
    1. Does the physical netlist match the HDL netlist?





netlist physical layout