defn (*project*)

a hierarchy of directories containing all of the files and other data needed for an FPGA build

NOTE: projects give you a nice GUI-based environment that allows you to manipulate things using the mouse rather than the keyboard.

defn (*wizard*)

A "wizard" in computing is something that guides the user through a sequence of steps to accomplish some process.

For example, Vivado has a wizard to walk a user through setting up a FPGA project.

<u>Defn</u> (synthesis)

the process of converting a high level description into a netlist

Note:

1) description usually is RTL code
2) you will need a constraint file

note
feedback

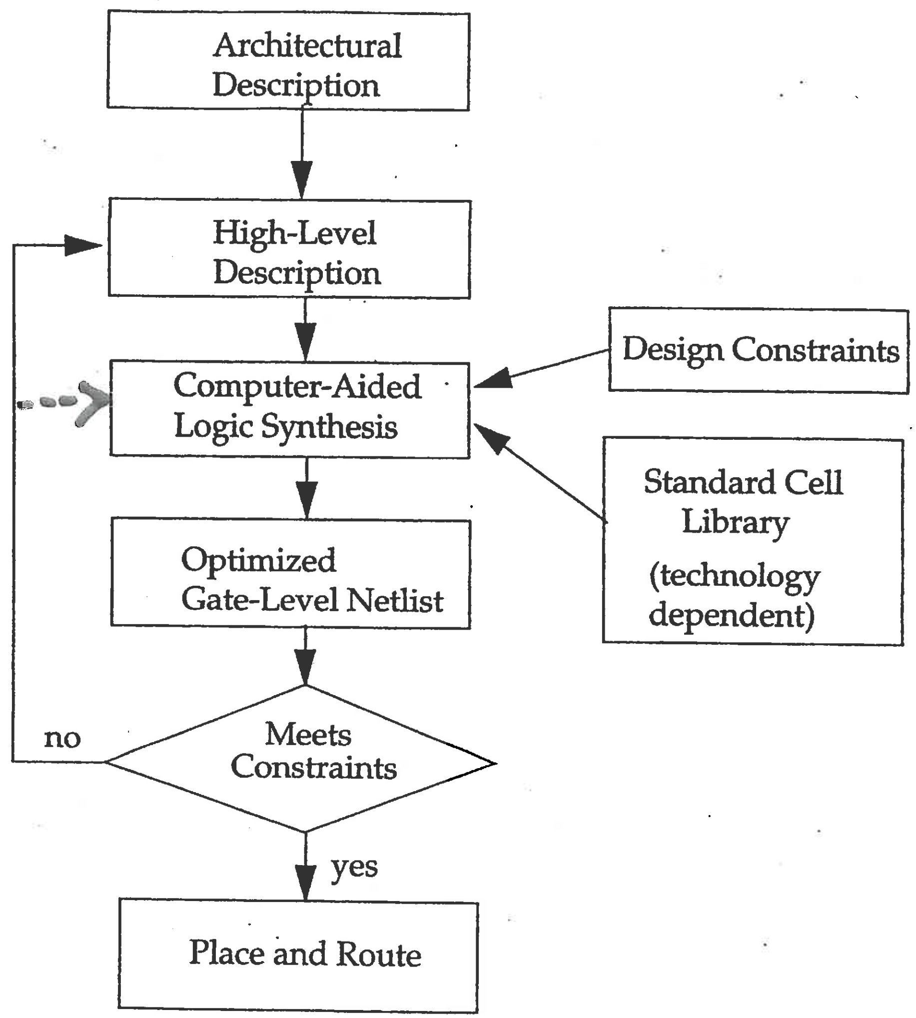


Fig 14-2 *Basic Computer-Aided Logic Synthesis Process*

Table 14-1

Verilog HDL Constructs for Logic Synthesis

Construct Type	Keyword or Description	Notes
ports	input, inout, output	
parameters	parameter	
module definition	module	
signals and variables	wire, reg, tri	Vectors are allowed
instantiation	module instances, primitive gate instances	E.g., mymux m1(out, i0, i1, s); E.g., nand (out, a, b);
functions and tasks	function, task	Timing constructs ignored
procedural	always, if, then, else, case, casex, casez	initial is not supported
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed
data flow	assign	Delay information is ignored
loops	for, while, forever,	while and forever loops must contain @(posedge clk) or @(negedge clk)

All delays are ignored

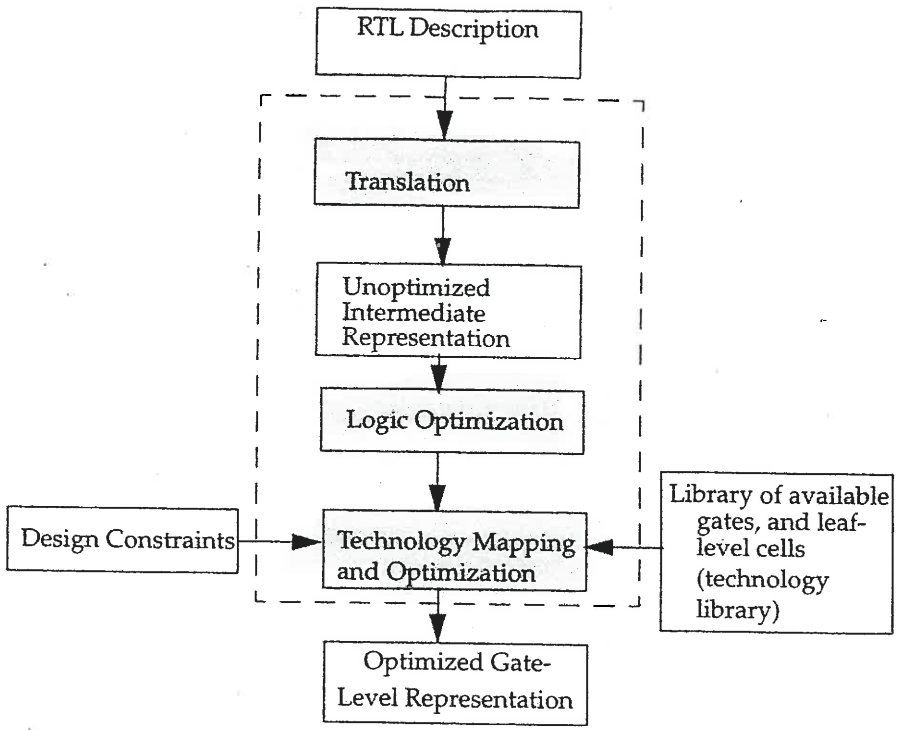
$\Downarrow$

simulation results $\neq$ synthesized results

---

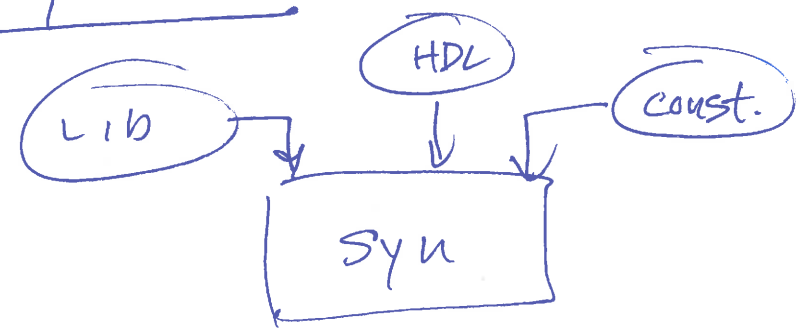
what does 'x' mean to synthesizer??

example 14-4



Logic Synthesis Flow from RTL to Gates

ASIC synthesis

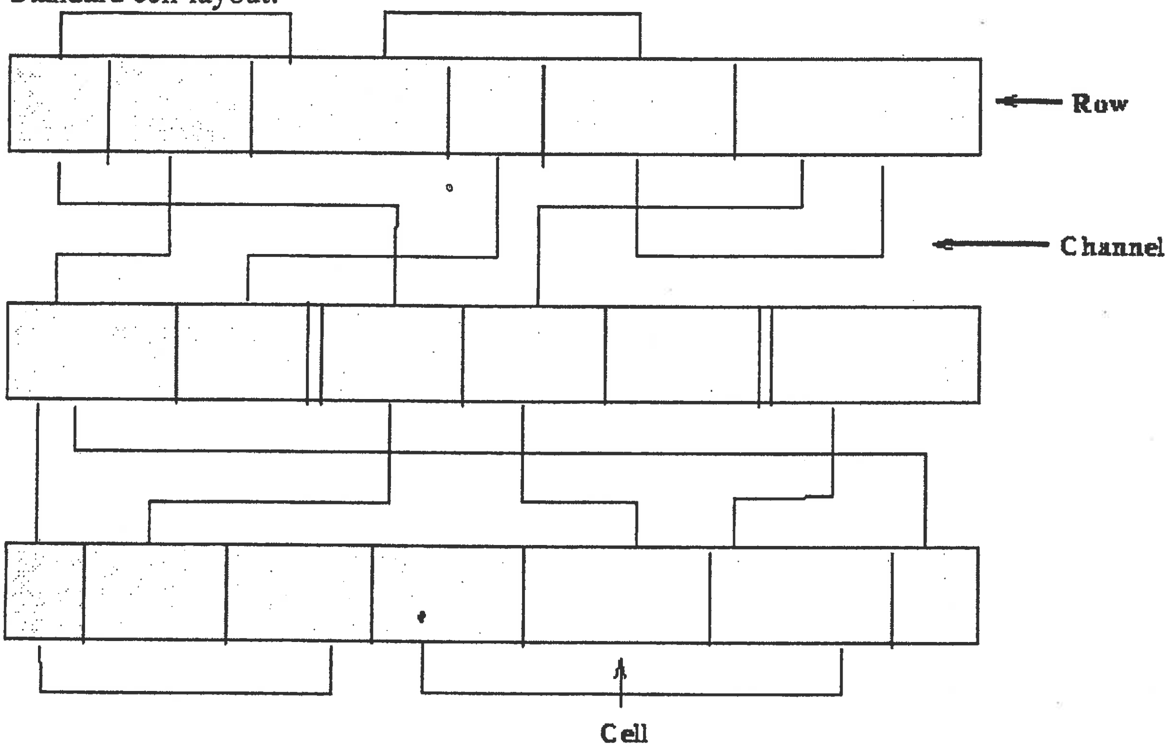


ASIC Library contain standard cells (gates)
and macro (muxes, counters, ...)

each standard cell/macro has

- function info
- layout info
- timing info
- power consumption info

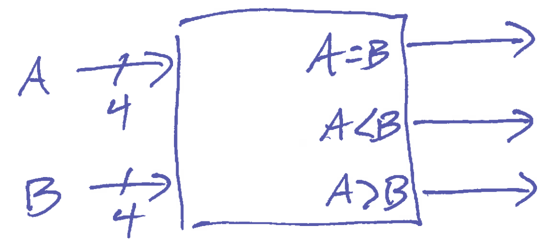
Standard Cell Design Style. In this methodology, the designer is provided with a predesigned library of cell layouts, called standard cells. These cells may be simple logic gates, such as NANDs, NORs, or complex modules like adders and flip flops. The cells are constrained to be of equal height, but they can be of varying width. Standard cells are typically placed in rows with cells butting against each other. This allows one to run common signals such as power and ground through the cells. In a typical design using standard cells, a desired function is realized by drawing the required cells from the standard cell library and describing their connectivity. The figure below shows a section of a standard cell layout.
Standard cell layout.



The standard cell network is then fed into an automatic place and route system. The place and route system first places the cells in rows and routes the connections among then in the area between the rows. The space between two adjacent rows is called a *channel*. The active area of the device is limited to the rows. The channel area contains the overhead due to the wiring. A good place and route system attempts to produce as compact a layout as possible, by keeping the channel area to a minimum. In a standard cell layout, the channel area can be as high as 70% of the total area of the device. Hence, there is great need to build better place and route tools that could bring down the wiring overhead.

<u>e.g.</u>:

specification (comparator)



propagation delay as small as possible (t_{pd})

Observation:

Area and ~~speed~~ t_{pd} are inversely proportional

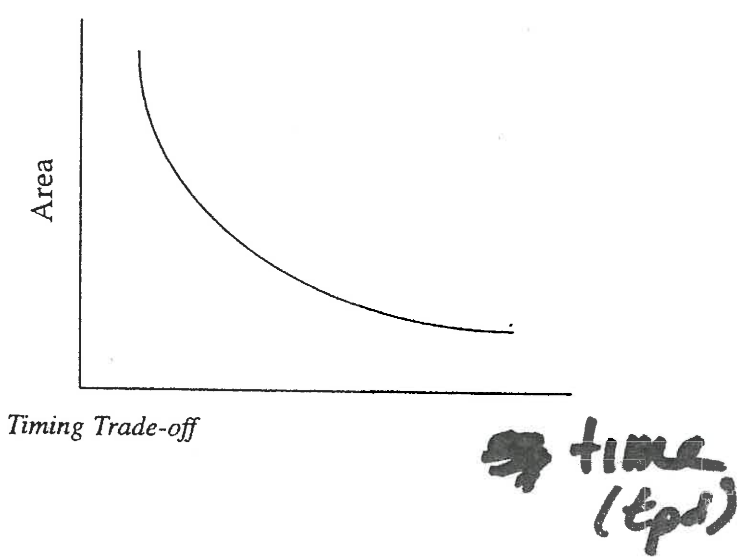


Figure 14-5 Area vs. Timing Trade-off

high speed ⇒ Large area

Example 14-1 *RTL for Magnitude Comparator*

```
//Module magnitude comparator
module magnitude_comparator(A_gt_B, A_lt_B, A_eq_B, A, B);

//Comparison output
output A_gt_B, A_lt_B, A_eq_B;

//4-bits numbers input
input [3:0] A, B;

assign A_gt_B = (A > B); //A greater than B
assign A_lt_B = (A < B); //A less than B
assign A_eq_B = (A == B); //A equal to B

endmodule
```

Notice that the RTL description is very concise.

```
//Library cells for abc_100 technology

VNAND//2-input nand gate

VAND//2-input and gate
VNOR//2-input nor gate
VOR//2-input or gate
VNOT//not gate
VBUF//buffer
NDFF//Negative edge triggered D flipflop
PDFF//Positive edge triggered D flipflop
```

(pg 317)

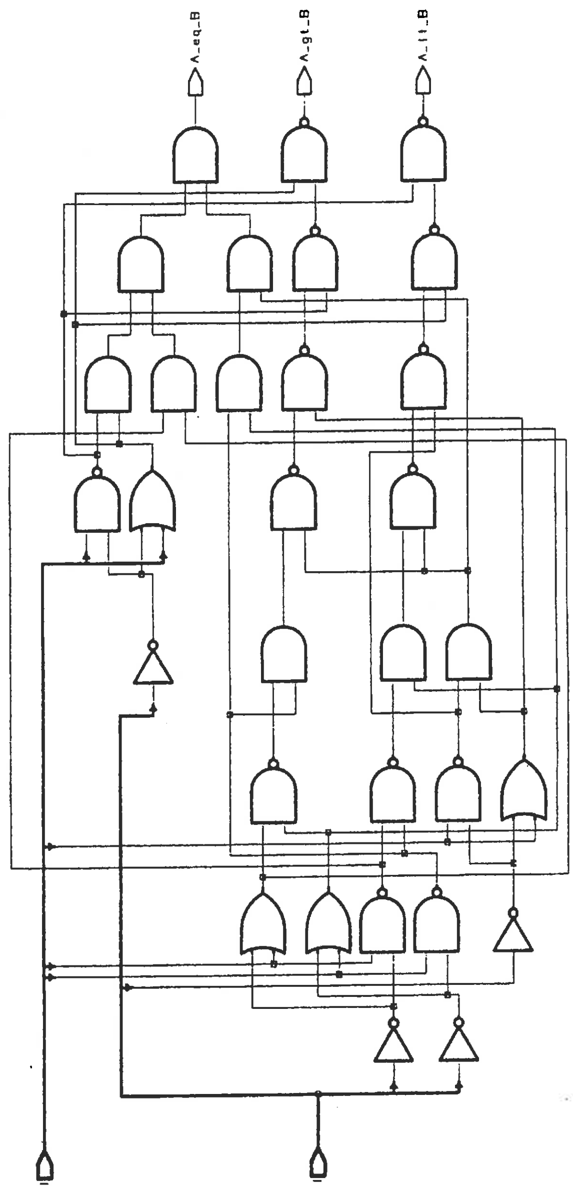


Figure 14-6 Gate-Level Schematic for the Magnitude Comparator

Example 14-3 *Stimulus for Magnitude Comparator*

```
module stimulus;

reg [3:0] A, B;
wire A_GT_B, A_LT_B, A_EQ_B;

//Instantiate the magnitude comparator
magnitude_comparator MC(A_GT_B, A_LT_B, A_EQ_B, A, B);

initial
  $monitor($time," A = %b, B = %b, A_GT_B = %b, A_LT_B = %b, A_EQ_B = %b",
        A, B, A_GT_B, A_LT_B, A_EQ_B);

//stimulate the magnitude comparator.

initial
begin
  A = 4'b1010; B = 4'b1001;
  # 10 A = 4'b1110; B = 4'b1111;
  # 10 A = 4'b0000; B = 4'b0000;
  # 10 A = 4'b1000; B = 4'b1100;
  # 10 A = 4'b0110; B = 4'b1110;
  # 10 A = 4'b1110; B = 4'b1110;
end

endmodule
```

The gate-level Verilog description produced by the logic synthesis tool for the circuit is shown in Example 14-2. Ports are connected *by name*.

Example 14-2 Gate-Level Description for the Magnitude Comparator

```
module magnitude_comparator ( A_gt_B, A_lt_B, A_eq_B, A, B );
input  [3:0] A;
input  [3:0] B;
output A_gt_B, A_lt_B, A_eq_B;
    wire n60, n61, n62, n50, n63, n51, n64, n52, n65, n40, n53,
         n41, n54, n42, n55, n43, n56, n44, n57, n45, n58, n46,
         n59, n47, n48, n49, n38, n39;
    VAND U7 ( .in0(n48), .in1(n49), .out(n38) );
    VAND U8 ( .in0(n51), .in1(n52), .out(n50) );
    VAND U9 ( .in0(n54), .in1(n55), .out(n53) );
    VNOT U30 ( .in(A[2]), .out(n62) );
    VNOT U31 ( .in(A[1]), .out(n59) );
    VNOT U32 ( .in(A[0]), .out(n60) );
    VNAND U20 ( .in0(B[2]), .in1(n62), .out(n45) );
    VNAND U21 ( .in0(n61), .in1(n45), .out(n63) );
    VNAND U22 ( .in0(n63), .in1(n42), .out(n41) );
    VAND U10 ( .in0(n55), .in1(n52), .out(n47) );
    VOR U23 ( .in0(n60), .in1(B[0]), .out(n57) );
    VAND U11 ( .in0(n56), .in1(n57), .out(n49) );
    VNAND U24 ( .in0(n57), .in1(n52), .out(n54) );
    VAND U12 ( .in0(n40), .in1(n42), .out(n48) );
    VNAND U25 ( .in0(n53), .in1(n44), .out(n64) );
    VOR U13 ( .in0(n58), .in1(B[3]), .out(n42) );
    VOR U26 ( .in0(n62), .in1(B[2]), .out(n46) );
    VNAND U14 ( .in0(B[3]), .in1(n58), .out(n40) );
    VNAND U27 ( .in0(n64), .in1(n46), .out(n65) );
    VNAND U15 ( .in0(B[1]), .in1(n59), .out(n55) );
    VNAND U28 ( .in0(n65), .in1(n40), .out(n43) );
    VOR U16 ( .in0(n59), .in1(B[1]), .out(n52) );
    VNOT U29 ( .in(A[3]), .out(n58) );
    VNAND U17 ( .in0(B[0]), .in1(n60), .out(n56) );
    VNAND U18 ( .in0(n56), .in1(n55), .out(n51) );
    VNAND U19 ( .in0(n50), .in1(n44), .out(n61) );
    VAND U2 ( .in0(n38), .in1(n39), .out(A_eq_B) );
    VNAND U3 ( .in0(n40), .in1(n41), .out(A_lt_B) );
    VNAND U4 ( .in0(n42), .in1(n43), .out(A_gt_B) );
    VAND U5 ( .in0(n45), .in1(n46), .out(n44) );
    VAND U6 ( .in0(n47), .in1(n44), .out(n39) );
endmodule
```

Example 14-4 *Simulation Library*

```
//Simulation Library abc_100.v. Extremely simple. No timing checks.

module VAND (out, in0, in1);
input in0;
input in1;
output out;

//timing information, rise/fall and min:typ:max
specify
(in0 => out) = (0.260604:0.513000:0.955206, 0.255524:0.503000:0.936586);
(in1 => out) = (0.260604:0.513000:0.955206, 0.255524:0.503000:0.936586);
endspecify

//instantiate a Verilog HDL primitive
and (out, in0, in1);
endmodule
...
```

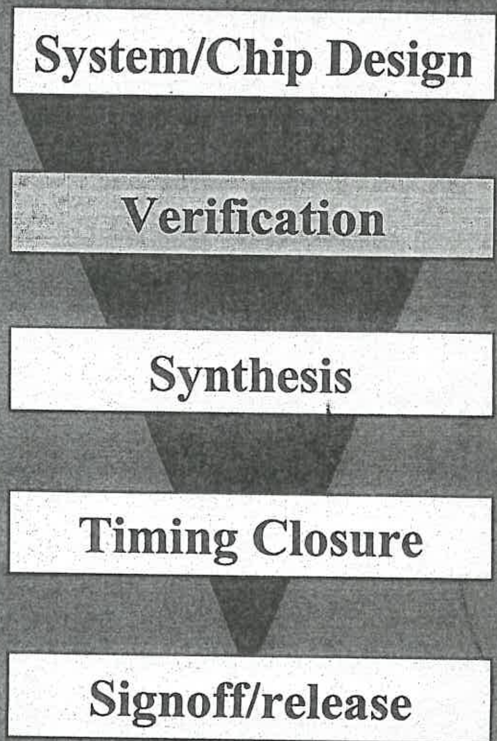
Timing verification

The gate-level netlist is typically checked for timing by use of *timing simulation* or by a *static timing verifier*. If any timing constraints are violated, the designer must either redesign part of the RTL or make trade-offs in design constraints for logic synthesis. The entire flow is iterated until timing requirements are met. Details of static timing verifiers are beyond the scope of this book. Timing simulation is discussed in Chapter 10, *Timing and Delays*.

(pg 323)

[redacted]

Design checkout or proof



- What is Verification?

  Proof the design meets the functional intent.

- What is timing closure?

  Proof the design meets the timing restrictions.

- What is test?

  Proof the design is manufactured without flaws.

EDA vendors claim 70% of design effort is now verification

Managing complexity

One guiding principle

★ Always clearly separate the datapath from the control path ★

1. Structure the datapath

from the design spec determine what functional units (muxes, adders, etc.) to use

this is where design tradeoffs are done

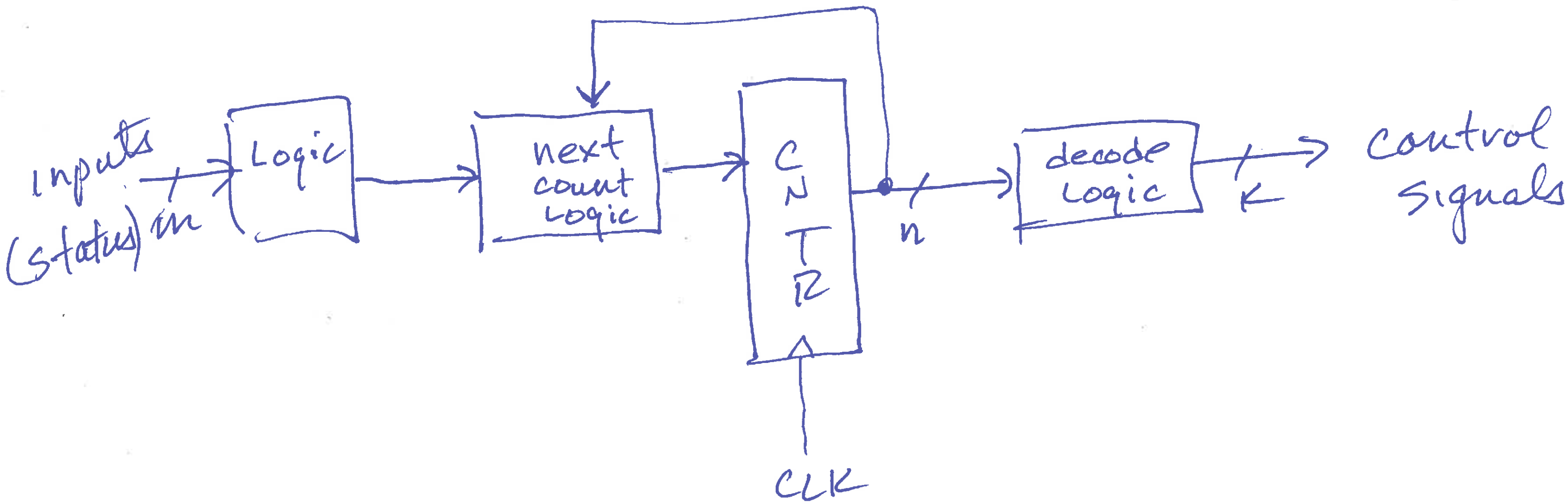
2. identify control points

functional units have control/status signals

called control points

③ Determine a control strategy

- FSM
- counter



4 Determine a reset strategy

i.e. what gets initialized and what is the initial value?

- FSMs have an initial state
- counter have an " count value
- decoders may have a defined output polarity
- FFs (registers) need initial state

Don't forget the reset signal itself (logic 0?)

5 Greenwood's 1st law

ALWAYS DESIGN BEFORE coding