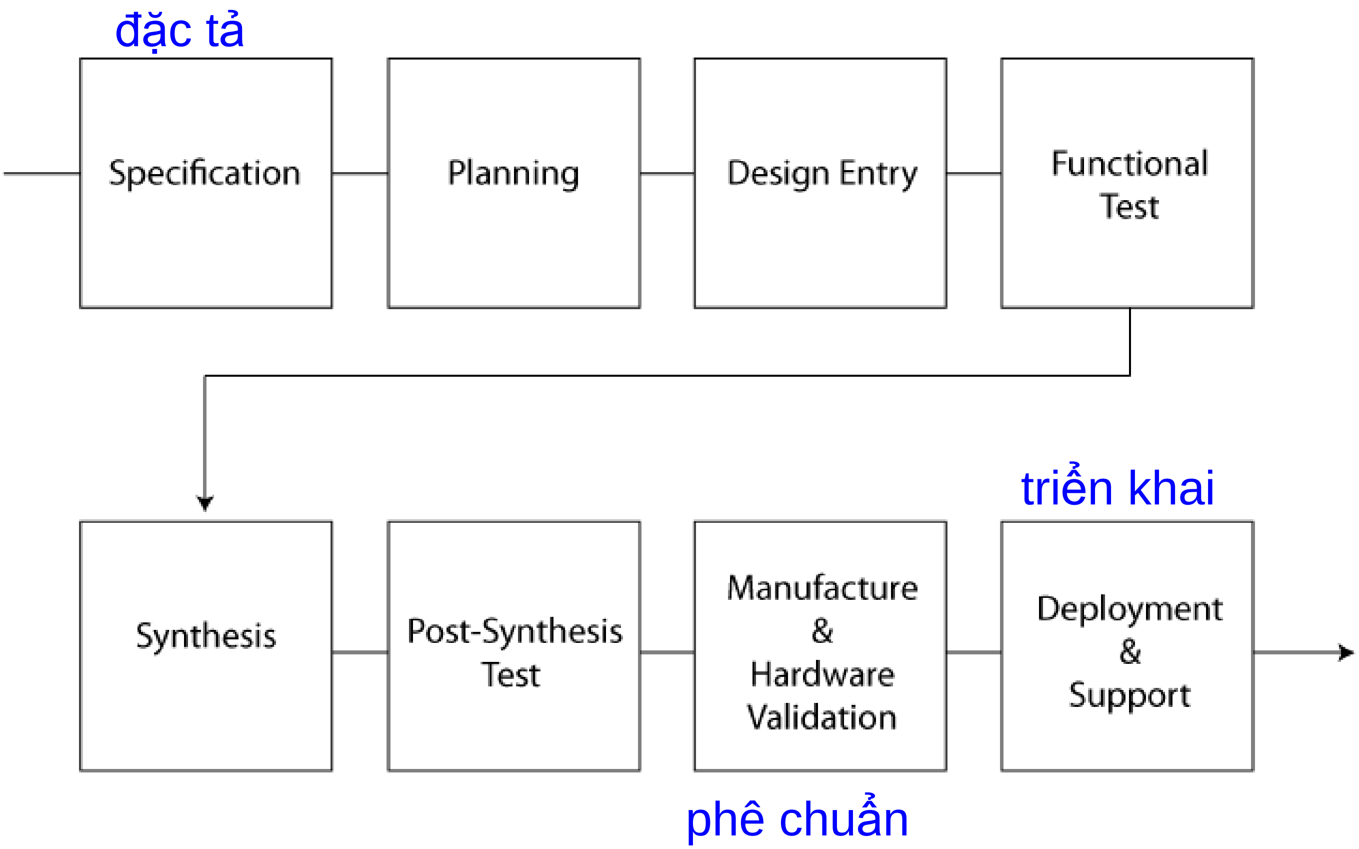
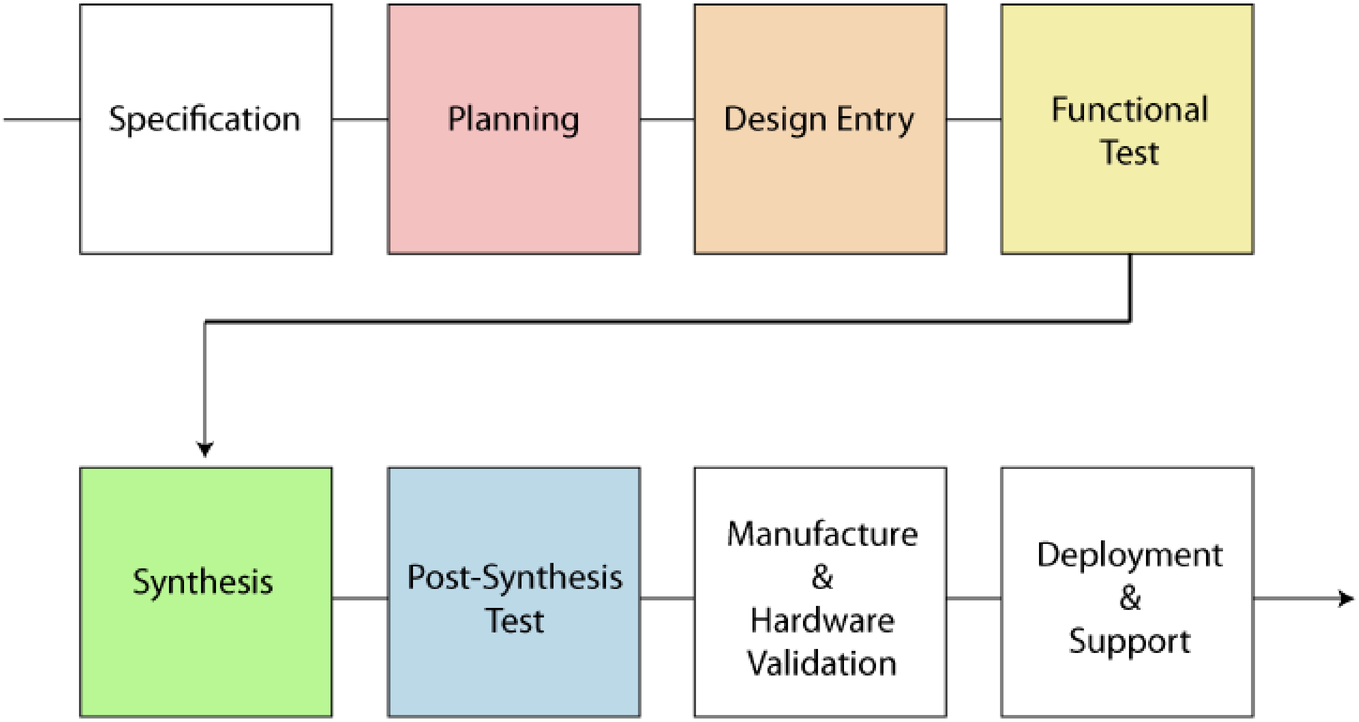
Overview

OVERVIEW OF HDLS AND THEIR ROLE IN HARDWARE DESIGN

Digital Design Process



Description Phase



Specification

Planning

Design Entry

Functional Test

Synthesis

Post-Synthesis Test

Manufacture & Hardware Validation

Deployment & Support

Implementation Phase

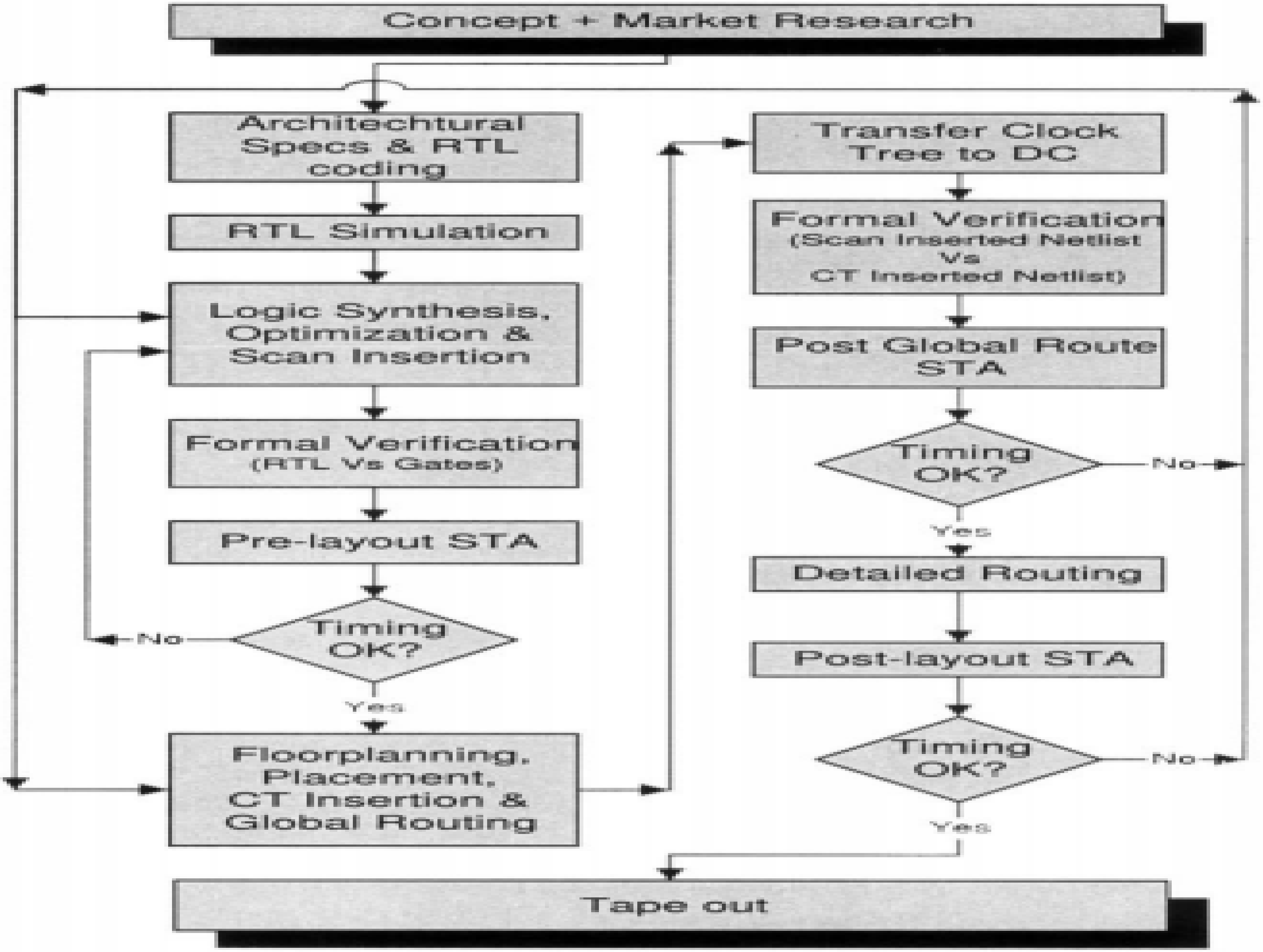

Concept + Market Research
Architechtural Specs & RTL coding
RTL Simulation
Logic Synthesis, Optimization & Scan Insertion
Formal Verification (RTL Vs Gates)
Pre-layout STA
Timing OK?
No
Yes
Floorplanning, Placement, CT Insertion & Global Routing
Transfer Clock Tree to DC
Formal Verification (Scan Inserted Netlist Vs CT Inserted Netlist)
Post Global Route STA
Timing OK?
No
Yes
Detailed Routing
Post-layout STA
Timing OK?
No
Yes
Tape out

Figure 1-1. Traditional ASIC Design Flow

HDL Overview

- Hardware description languages (HDLs)
  - văn bản
    Textual descriptions of digital logic
  - Description languages, not Programming languages
  - Allow modeling and simulating the functional behavior and timing of digital hardware
  - Synthesis tools take an HDL description and generate a technology-specific netlist (real hardware representation)
- Two main HDLs used by industry
  - Verilog (C-based, industry-driven)
  - VHDL (Ada-based, defense/industry/university-driven)

  - Other options available: SystemC

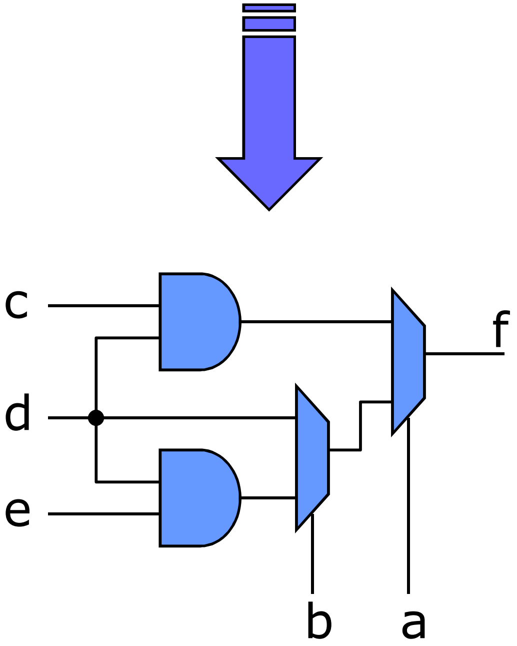
Describing Hardware, not Software!

- Hardware is created during synthesis
  - Even if `a` is true, still performs `d&e`
  - HDLs are inherently parallel

vốn đã

- Learn to understand how descriptions translated to hardware

```
if (a) f = c & d;
else if (b) f = d;
else f = d & e;
```



OK, but why not just use Schematic Capture?

chụp sơ đồ

Why Use an HDL?

- More and more transistors can fit on a chip
  - Allows larger designs!
  - Work at transistor/gate level for large designs: extremely difficult and extremely expensive.
  - Many designs need to go to production quickly

đoạn tóm tắt

- Abstract large hardware designs
  - Describe what you need the hardware to do
  - Tools then design the hardware for you
- BIG CAVEAT cảnh báo
  - Good descriptions → Good hardware
  - Bad descriptions → BAD hardware!

Why Use an HDL?

- Simplified & faster design process
- Explore larger solution space
  - Smaller, faster, lower power
  - Throughput vs. latency độ trễ
  - Examine more design tradeoffs đánh đổi
- giảm bớt Lessen the time spent gỡ lỗi debugging the design
  - Design errors still possible, but in fewer places
  - Generally easier to find and fix
- Can reuse design to target different technologies
  - Don’t manually change all transistors for rule change

Other Important HDL Features

di động

- Are highly portable (text)

tự tạo tài liệu

- Are self-documenting (when commented well)
- Describe multiple levels of abstraction
- Provides many descriptive styles
  - Structural
  - Register Transfer Level (RTL)
  - Behavioral
- Serve as input for synthesis tools

HDL Overview

- HDLs may LOOK like software, but they’re not!
  - NOT a program
  - Doesn’t “run” on anything
    - Though we do *simulate* them on computers
  - This is an important distinction to remember

- Also use HDLs to test the hardware you create
  - Some special HDL code can be used more like software, but is only for simulation purposes, not for synthesis

sự phân chia

HDL Dichotomy – Sim VS Synth

- HDL code can be divided into two major categories

- Synthesizable Code

  biến thành

  - Can be converted into real hardware

  giới hạn

  - Bound by the same limitations as hardware
  - Can be both simulated and synthesized

- Non-Synthesizable Code
  - Only meant for simulation
  - Can represent behaviors that are too difficult or too expensive to create in real hardware
  - Used to test the Synthesizable Code you design

HDL Dichotomy – Sim VS Synth

- Example:
- Synthesizable: A 3-input AND gate
- Non-synthesizable: A 3-input AND gate that has a delay of 5 ns on Sundays and 10 ns on Weekends
- Synthesizable: A 32-bit output bus
- Non-synthesizable: printf("Hello World")

HDL Dichotomy – Sim VS Synth

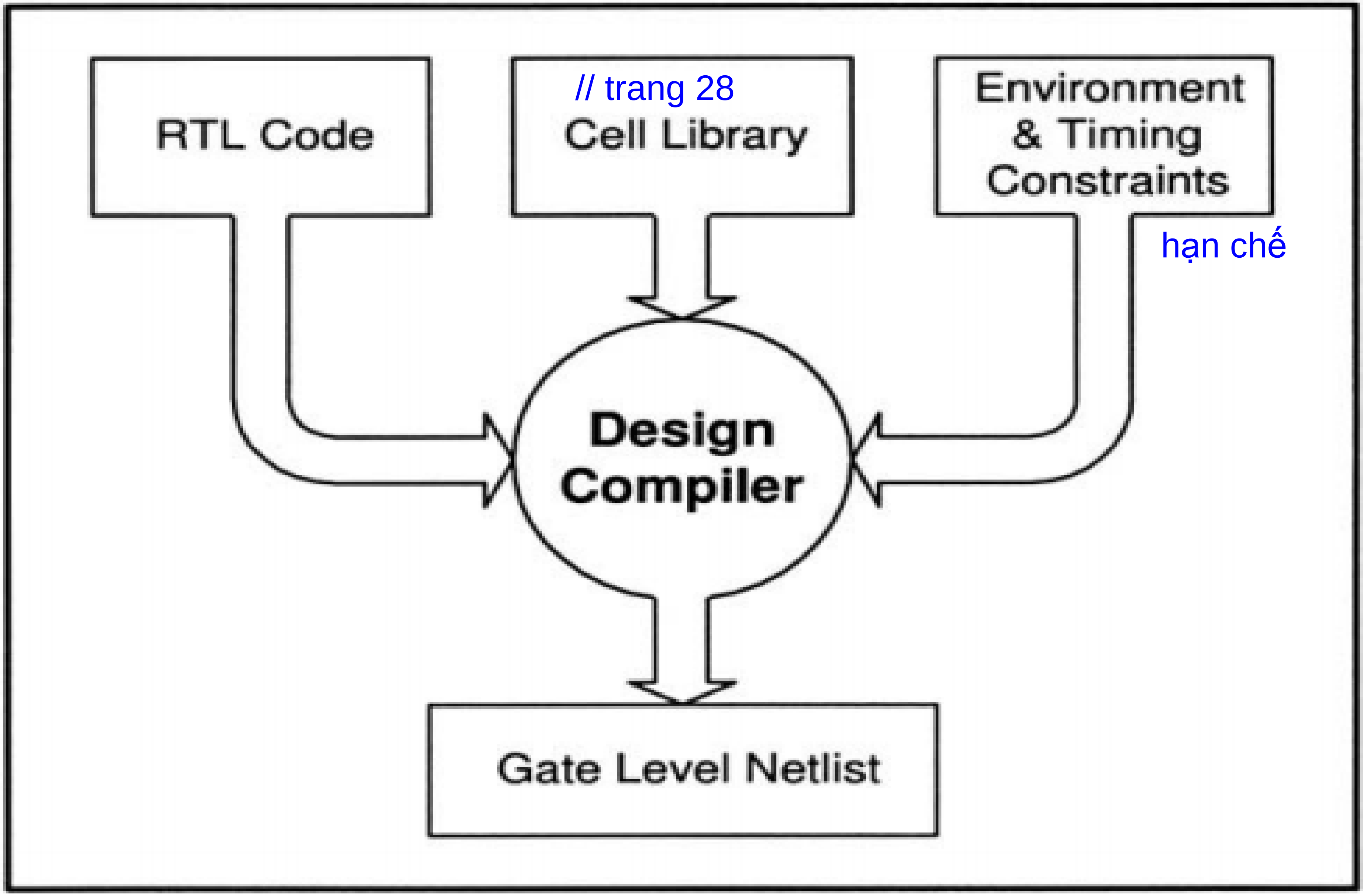
- Unfortunately, it is not always so obvious to determine what is and isn't synthesizable.
    - Some things are non-synthesizable according to the Verilog Synthesis Standard
    - Some things are synthesizable only by certain synthesis tools
    - Some things are synthesizable only when targeting certain hardware

khả năng

- Rely on your knowledge of hardware capabilities
- This will become clearer with time & practice
- Generally you can assume that we are talking about Synthesizable code unless otherwise stated

trừ khi có quy định khác

HDL Design Flow

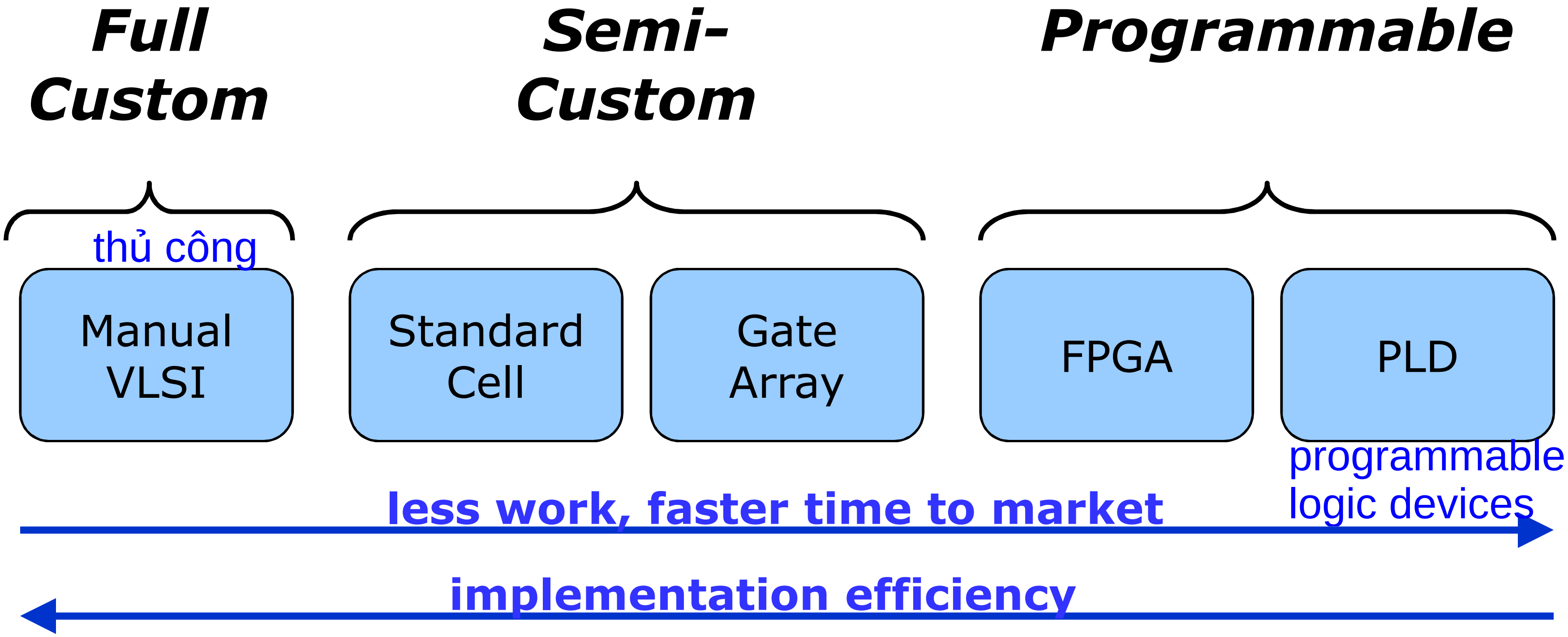


Overview

HARDWARE IMPLEMENTATIONS OVERVIEW

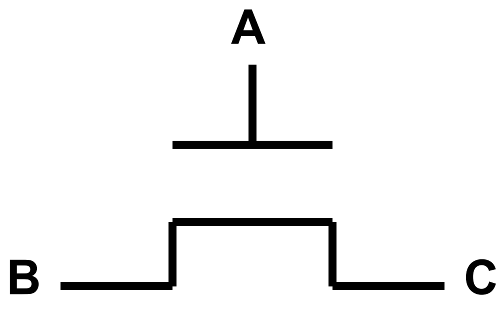
Hardware Implementations

- HDLs can be compiled to semi-custom and programmable hardware implementations
- We will focus on Standard Cells in this course

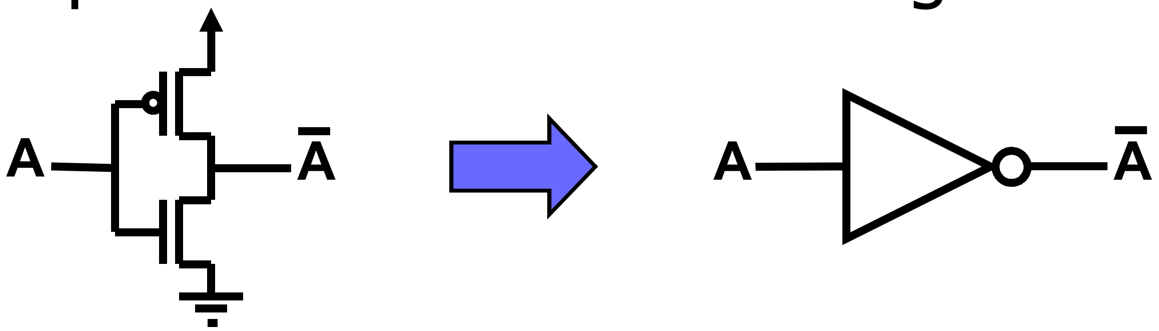

Full Custom
Semi-Custom
Programmable
thủ công
Manual VLSI
Standard Cell
Gate Array
FPGA
PLD
programmable logic devices
less work, faster time to market
implementation efficiency

Hardware Building Blocks

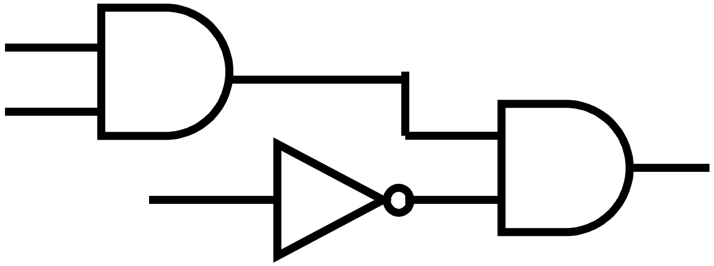
- Transistors are switches



- Use multiple transistors to make a gate



- Use multiple gates to make a circuit

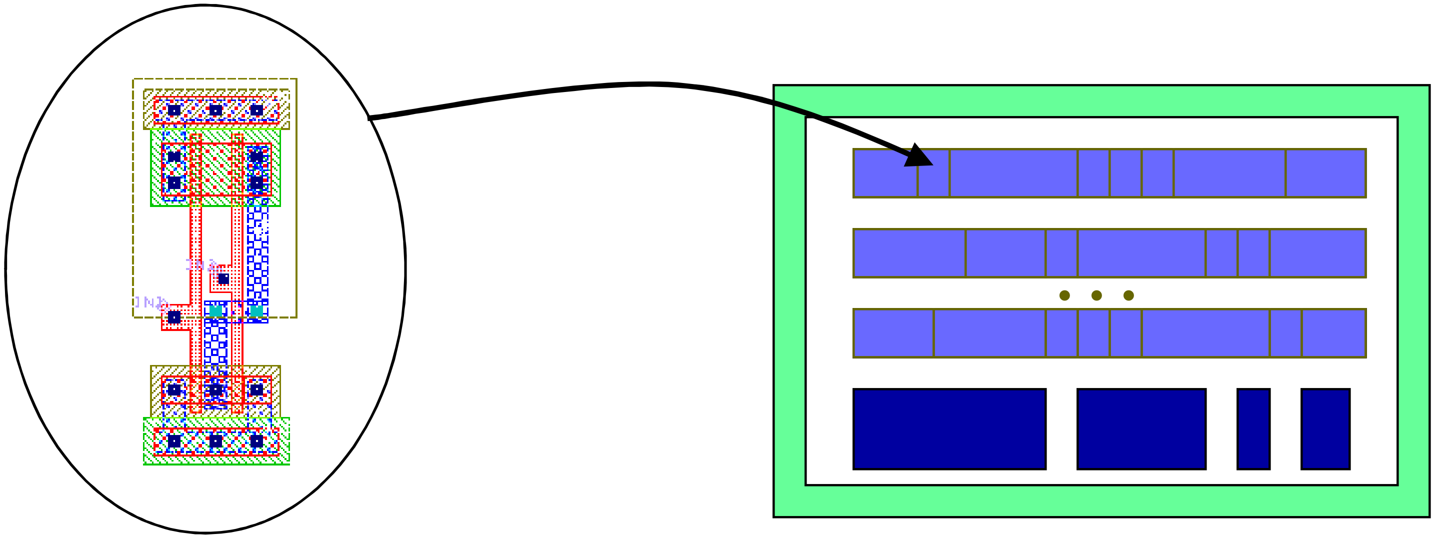


Standard Cells

- Library of common gates and structures (cells)

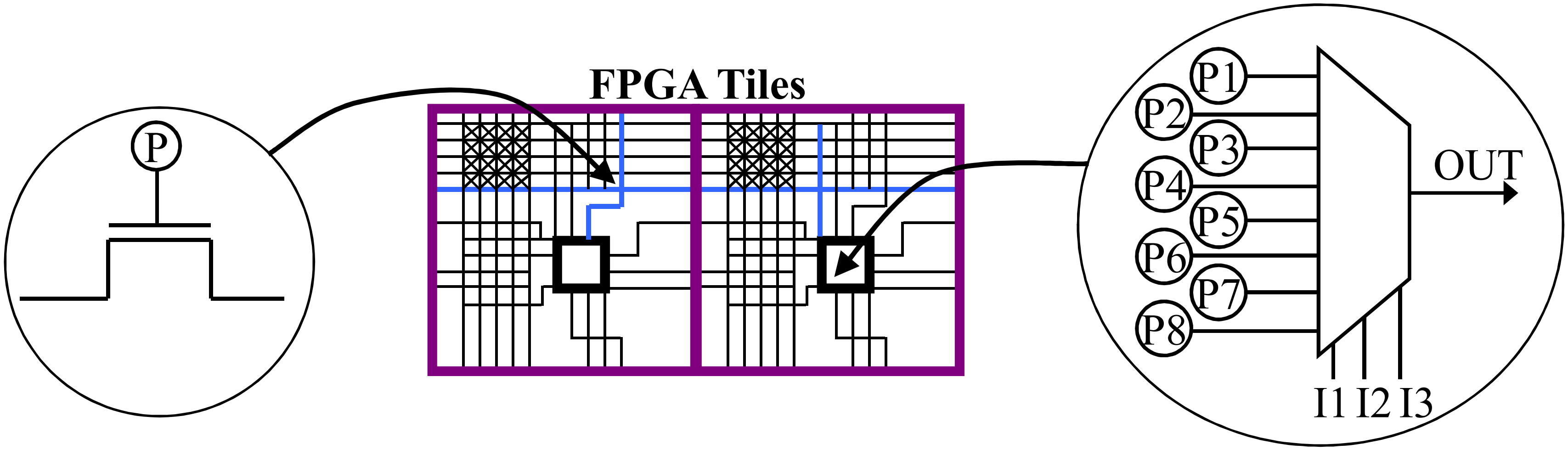
phân tích

- Decompose hardware in terms of these cells
- Arrange the cells on the chip
- Connect them using metal wiring



FPGAs

- “Programmable” hardware
- Use small memories as truth tables of functions
- Decompose circuit into these blocks
- Connect using programmable routing
- SRAM or flash memory bits control functionality



Introduction to Verilog

- In this class, we will use the Verilog HDL
  - Widely used in academia and industry
- VHDL is another common HDL
  - Also widely used by both academia and industry

- Verilog HDL is the most common in IC industry
  - Easier to use in many ways = better for teaching
  - C - like syntax
- History

  độc quyền

  - Developed as proprietry language in 1985

  miền

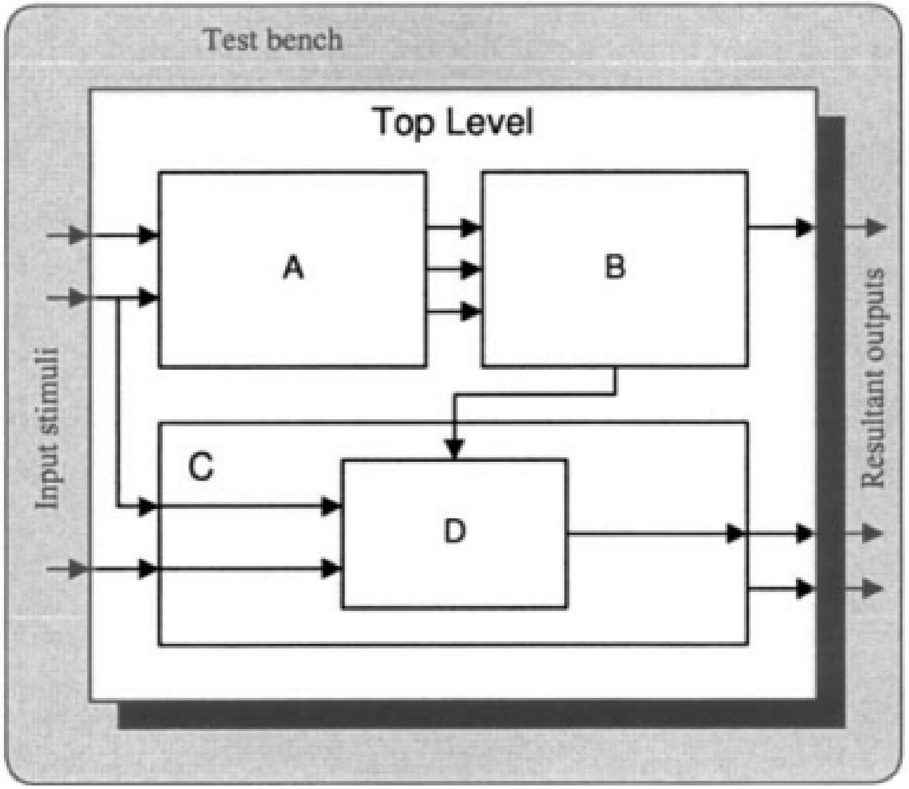
  - Opened as public domain spec in 1990
    - Due to losing market share to VHDL
  - Became IEEE standard in 1995

Introduction to Verilog

- Many principles we will discuss apply to any HDL
- Once you can “think hardware”, you should be able to use any HDL fairly quickly

Introduction to Verilog

- Verilog constructs are use defined *keywords*
  - Examples: and, or, wire, input output
- One important construct is the *module*
  - Modules have inputs and outputs
  - Modules can be built up of Verilog primatives or of user defined submodules.



Example: Simple Circuit Diagram

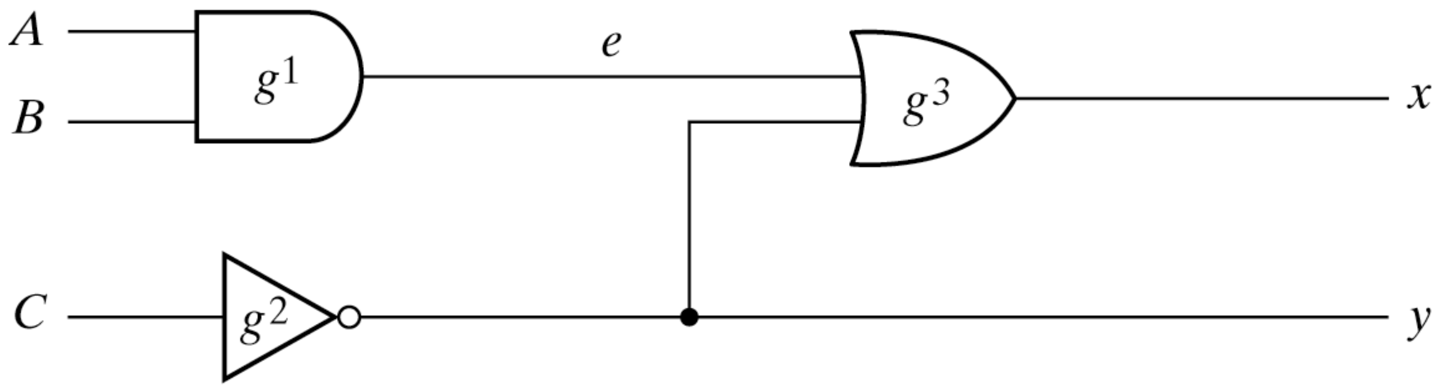


Fig. 3-37 Circuit to Demonstrate HDL

Example: Simple Circuit HDL

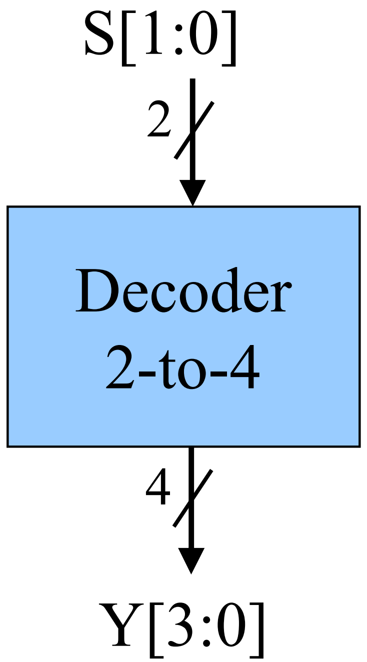
```
module smpl_circuit(A,B,C,x,y);
input A,B,C; output x,y; wire e;
and g1(e,A,B);
not g2(y, C);
or      g3(x,e,y);
endmodule
```

- The module starts with **module** keyword and finishes with **endmodule**.
- Internal signals are named with **wire**.
- Comments follow **//**
- **input** and **output** are ports. These are placed at the start of the module definition.
- Each statement ends with a semicolon, except **endmodule**.

Verilog Modules

- In Verilog, the basic unit of design is a module.

```
module decoder_2_to_4 (S, Y) ;

input [1:0] S ;
output [3:0] Y ;

assign Y =      (S == 2'b00) ? 4'b0001 :
                (S == 2'b01) ? 4'b0010 :
                (S == 2'b10) ? 4'b0100 :
                (S == 2'b11) ? 4'b1000 ;
endmodule
```



Verilog Modules

module name

port names of module

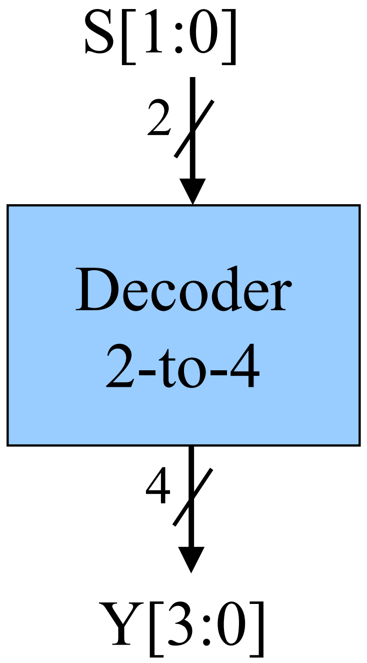
```
module decoder_2_to_4 (S, Y) ;

input [1:0] S ;
output [3:0] Y ;

assign Y =      (S == 2'b00) ? 4'b0001 :
                (S == 2'b01) ? 4'b0010 :
                (S == 2'b10) ? 4'b0100 :
                (S == 2'b11) ? 4'b1000 ;
endmodule
```

port types

port sizes

module contents (what it "does")

keywords underlined



Circuit to code

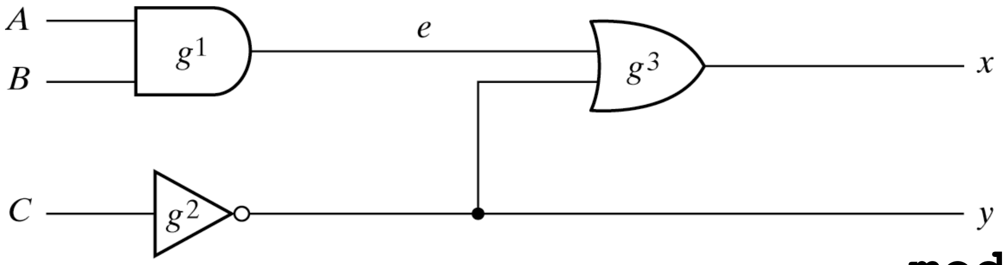


Fig. 3-37 Circuit to Demonstrate HDL

```
module
smpl_circuit(A,B,C,x,y
);
  input A,B,C;
  output x,y;
  wire e;
  and g1(e,A,B);
  not g2(y, C);
  or g3(x,e,y);
endmodule
```

Declaring A Module

- Name the module
  - Can’t use Verilog keywords (see Appendix C.1) as module, port or signal names
  - Choose a *descriptive* module name

giao diện

- List the port names (module interface)
  - Choose *descriptive* port names
  - Declare the type and size of ports

- Declare any internal signals
- Write the internals of the module (functionality)

Declaring A Module

- Good HDL code is self-commenting

```
module xyz123 (A, B) ;
        input [3:0] A ;
        output B ;

        //module contents
endmodule
```

BAD

```
module doomsday_machine (crystals, earthquake) ;
        input [3:0] crystals ;
        output earthquake ;

        //module contents
endmodule
```

GOOD

Declaring Ports

- Only the ports are accessible from outside the module!
- A signal is attached to every port

- Declare type of port
  - **input**
  - **output** 2 chiều
  - **inout** (bidirectional)
- Scalar (single bit) - don’t specify a size
  - **input** cin;
- Vector (multiple bits) - specify size using range
  - Range is MSB to LSB (left to right)
  - Don’t have to include zero if you don’t want to… (**D[2:1]**)
  - **output** [7:0] OUT; ← most common to use high:low
  - **input** [0:4] IN;

Verilog Module Styles

- Modules can be specified different ways

  cơ bản
  - *Structural* – connect primitives and modules
  - *RTL* – use continuous assignments to specify combinational logic
  - *Behavioral* – use initial and always blocks to describe the behavior of the circuit, not its implementation
- A single module can (and often does) use more than one method.

- We will cover structural first, then RTL, and finally behavioral.
- First, a quick example of the differences…

Structural Verilog

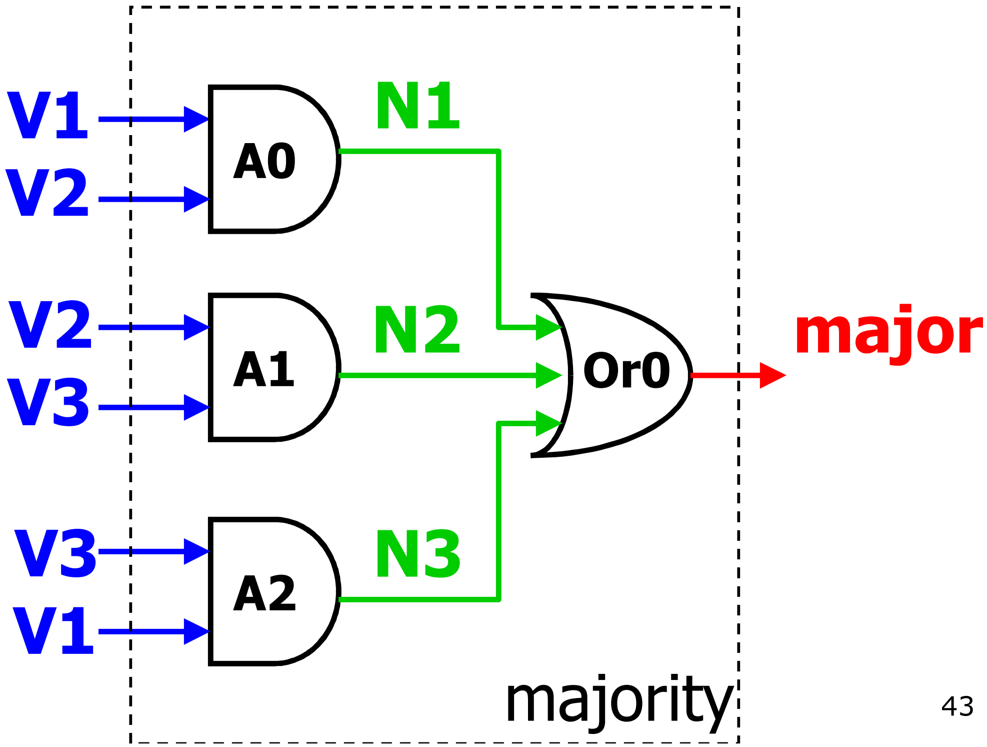
- Text description of a schematic sơ đồ
- Build up a circuit from gates/flip-flops
  - Gates are primitives (part of the language)
  - No flip-flop primitive
- Structural design
  - Create module interface
  - Instantiate the gates in the circuit
  - Declare the internal wires needed to connect gates
  - Put the names of the wires in the correct port locations of the gates to make connections
    - For primitives, outputs always come first

máy dò đa số

Structural Example: Majority Detector

- Structural models specify interconnections of primitives and modules.
- Synthesis tools may still optimize your design!

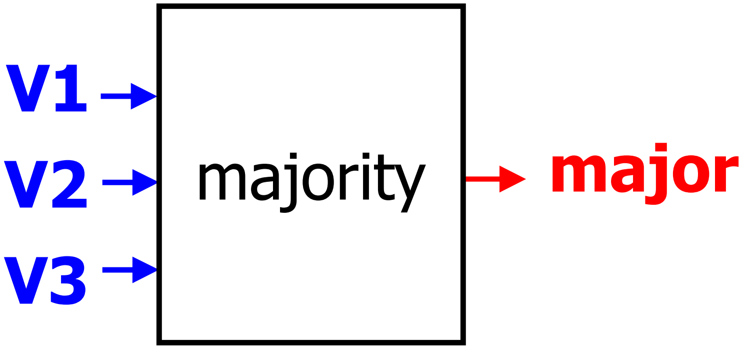
```
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

wire N1, N2, N3;

and A0 (N1, V1, V2),
    A1 (N2, V2, V3),
    A2 (N3, V3, V1);

or  Or0 (major, N1, N2, N3);

endmodule
```



RTL Example: Majority Detector

- RTL models use continuous assignment statements to assign Boolean expressions to signals.
- If an input value changes, the value of the assignment is immediately updated. This is combinational *hardware,* not *software*.

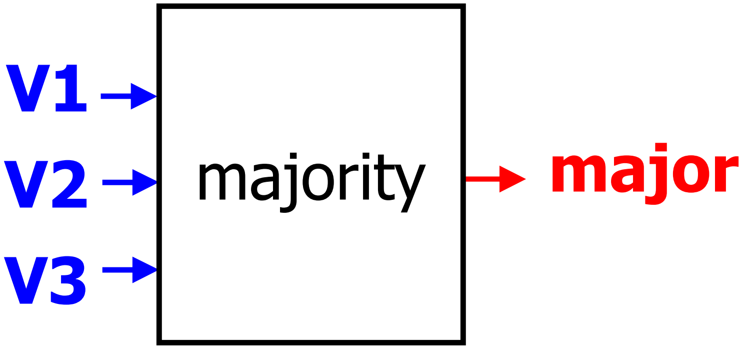
```
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

assign major = (V1 & V2)
             | (V2 & V3)
             | (V1 & V3);
endmodule
```



Behavioral Example: Majority Detector

- Behavior models specify what the logic does, not how to do it (simulation versus hardware)
  - Tools try to figure out what hardware is implied by the described behavior – Not all behaviors can synthesize!

```
module majority (major, V1, V2, V3) ;

output reg major ;
input V1, V2, V3 ;

always @(V1, V2, V3) begin
    if ((V1 && V2) || (V2 && V3)
        || (V1 && V3)) major = 1;
    else major = 0;
end

endmodule
```



What do you think the synthesis tool will make of this?

Syntax For Structural Verilog

- First declare the interface to the module
  - Module keyword, module name
  - Port names/types/sizes
- Next, declare any internal wires using “wire”
  - wire [3:0] partial_sum;

tạo

- Then *instantiate* the gate primitives/submodules
  - Indicate which signal is on which port

Declaration vs. Instantiation

- Declaration is when you define a module
  - Its interface
  - Its functionality

xor_2 is **declared** here

```
module xor_2 (Y, A, B) ;
input  A, B ;
output Y ;
assign Y = A ^ B;
endmodule
```

- Instantiation is when you "insert" the module into your design

chèn

xor_3 is **declared** here

```
module xor_3 (F, C,D,E) ;
input  C, D, E ;
wire G;
xor_2 x1 (G, C, D) ;
xor_2 x2 (F, G, E) ;
endmodule
```

xor_2 is **instantiated** twice here (two copies)

Structural Basics: Primitives

- Build design up from the gate level
  - Flip-flops usually constructed using Behavioral Verilog
- Verilog provides a set of gate primitives
  - **and, nand, or, nor, xor, xnor, not, buf, bufif1…**
  - Combinational building blocks for structural design
  - Each primitive is a “black box”
- Can also model at the transistor level
  - Most people don’t; we won’t

Primitives

- No declarations - can only be instantiated
- **Output port appears before input ports**
- Optionally specify: instance name and/or delay

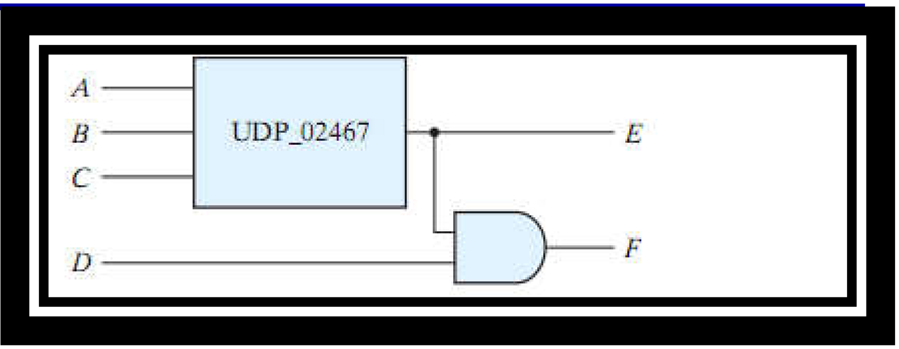
```
and N25 (Z, A, B, C);     // name specified
and #10 (Z, A, B, X),
        (X, C, D, E);     // 2 gates, delay specified
and #10 N30 (Z, A, B); // name and delay specified
```

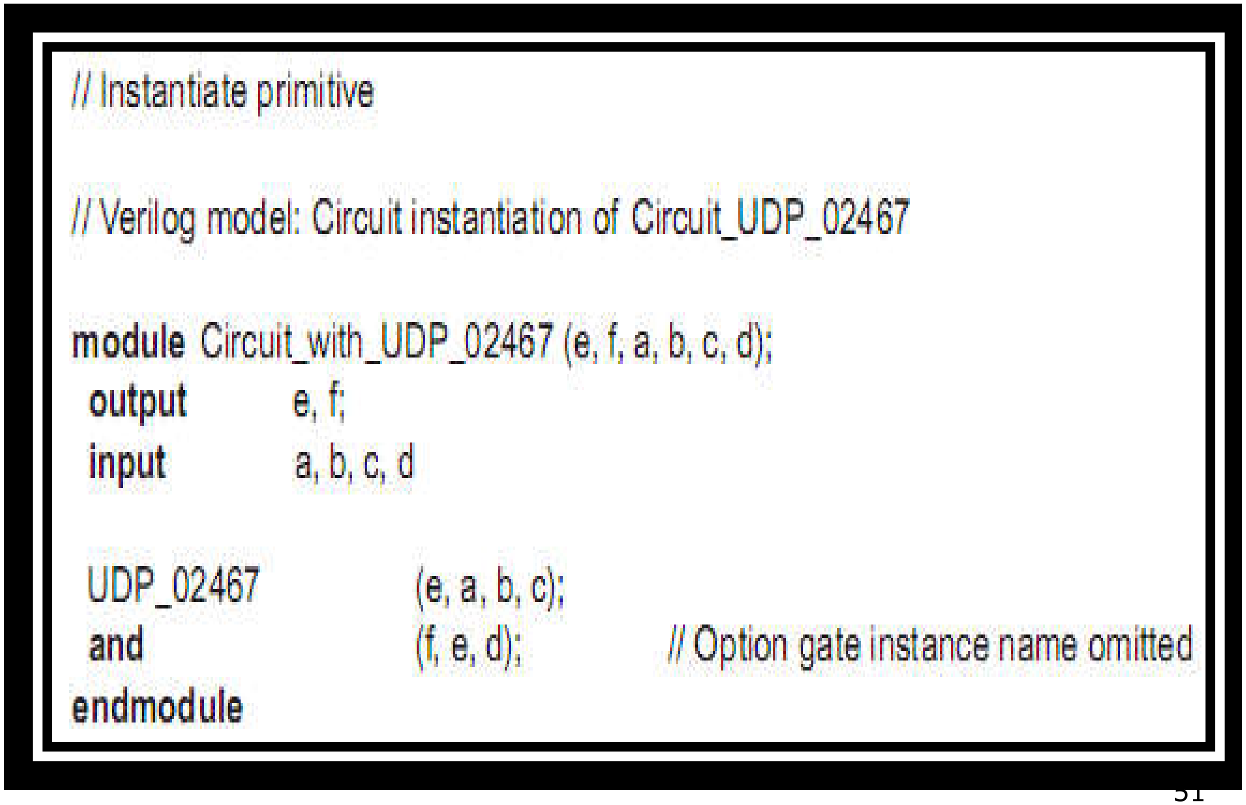
User Defined Primitives

- It is declared with the keyword **primitive** , followed by a name and port list.
- There can be only one **output**, and it must be listed first in the port list and declared with keyword output .
- There can be any number of inputs. The order in which they are listed in the **input** declaration must conform to the order in which they are given values in the table that follows.
- The truth table is enclosed within the keywords **table** and **endtable**.
- The values of the inputs are listed in order, ending with a colon (:). The output is always the last entry in a row and is followed by a semicolon (;).
- The declaration of a UDP ends with the keyword **endprimitive**.

User Defined Primitives

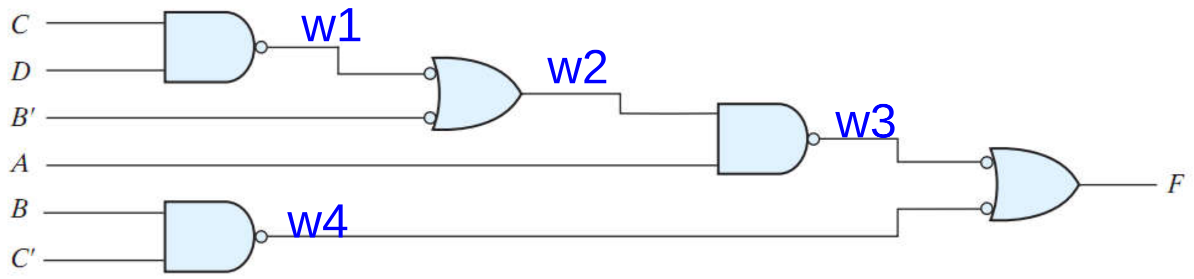
```
// Verilog model: User-defined Primitive
primitive UDP_02467 (D, A, B, C);
  output D;
  input   A, B, C;
//Truth table for D 5 f (A, B, C) 5 Σ(0, 2, 4, 6, 7);
  table
//      A     B     C     :     D       // Column header comment
        0     0     0     :     1;
        0     0     1     :     0;
        0     1     0     :     1;
        0     1     1     :     0;
        1     0     0     :     1;
        1     0     1     :     0;
        1     1     0     :     1;
        1     1     1     :     1;
  endtable
endprimitive
```





```
// Instantiate primitive

// Verilog model: Circuit instantiation of Circuit_UDP_02467

module Circuit_with_UDP_02467 (e, f, a, b, c, d);
  output        e, f;
  input         a, b, c, d

  UDP_02467            (e, a, b, c);
  and                  (f, e, d);          // Option gate instance name omitted
endmodule
```

Examples



(b) NAND gates

FIGURE 3.20
Implementing $F = A(CD + B) + BC'$

assign F = (((C ~& D) ~& B') ~& A) ~& (B ~& C')

```
module Fig_3_20b_gates (F, A, B, B_Bar,
C, C_bar, D);
          output F;
        input  A, B, B_bar, C, C_bar, D;
        wire  w1, w2, w3, w4;
        nand   (w1, C, D);
        not   (w1_bar, w1);
      or   (w2, w1_bar, B);
      nand   (w3, w2, A);
      not   (w3_bar, w3);
      nand   (w4, B, C_bar);
      not   (w4_bar, w4);
      or    (F, w3_bar, w4_bar);
      endmodule
```

```
module Fig_3_20b_CA (F, A, B, C, C_bar,
D);
          output F;
        input  A, B, B_bar, C, C_bar, D;
        wire  w2 = !w1;
        wire  w3 = !B_bar;
       wire   w4, w5, w5_bar, w6, w6_bar;
       assign  w1 = !(C && D);
        assign w4 = w2 || w3;
        assign w5 = !(w4 && A);
        assign w5_bar = !w5;
        assign w6 = !(B && C_bar);
        assign w6_bar = !w6;
assign F = w5_bar || w6_bar;
       endmodule
```

Examples

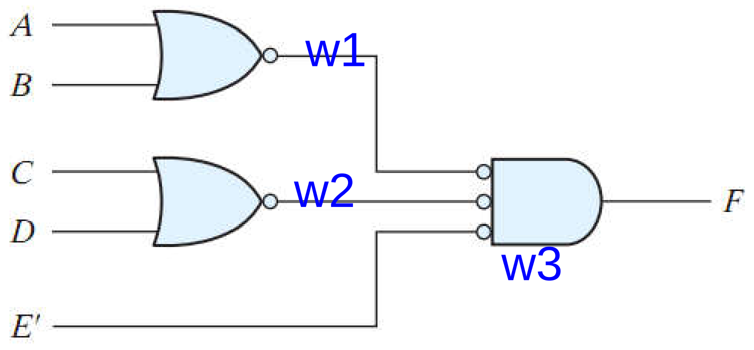


FIGURE 3.24
Implementing $F = (A + B)(C + D)E$

```
module Fig_3_24_gates (F, A, A_bar, B,
B_bar, C, D_bar);
        output F;
        input  A, A_bar, B, B_bar, C, D_bar
       wire  w1, w2, w3;
        not  (w1_bar, w1);
        not  (w2_bar, w2);
        not   (w3, E_bar);
        nor   (w1, A, B);
        nor  (W2, C, D);
        and  (F, w1_bar, w2_bar, w3);
      endmodule
```

```
module Fig_3_24_CA (F, A, B, C, D,
E_bar);
    output F;
    input  A, B, C, D, E_bar;
    wire  w1, w2, w1_bar, w2_bar, w3~~_bar~~;
    assign  w1 = !(A || B);
    assign  w1_bar = !w1;
   assign  w2 = !(C || D);
   assign w2_bar = !w2;
    assign w3 = !E_bar;
    assign F = w1_bar && w2_bar && w3;
endmodule
```

Adding Delays

- To simulate a circuits real world behaviour it is important that propogation delays are included.

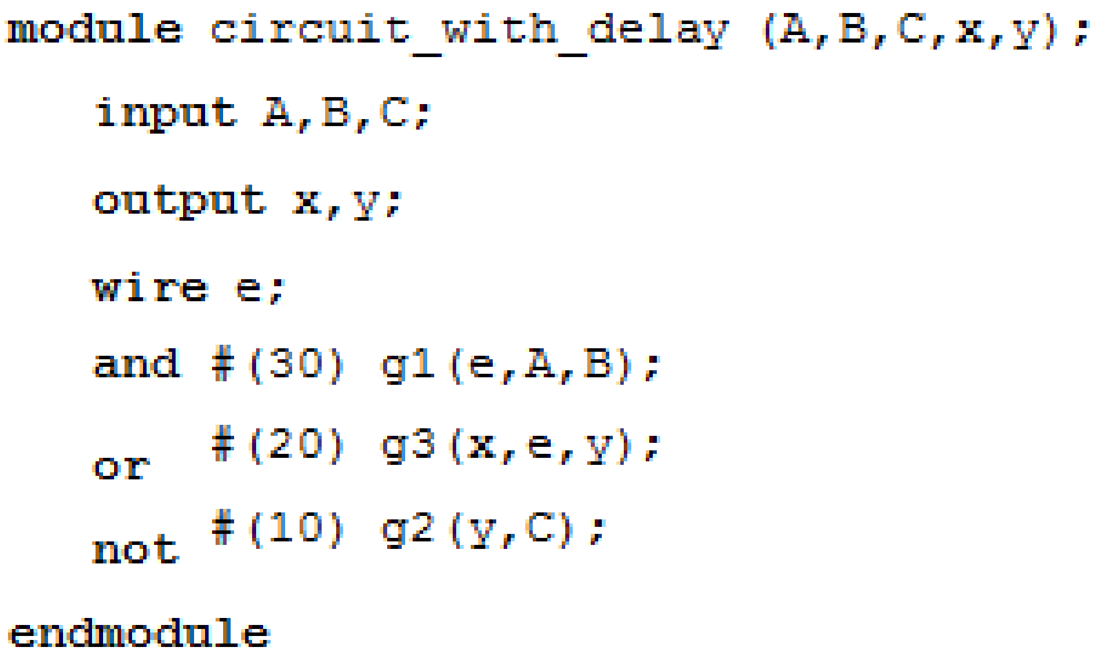
  lan truyền

- The units of time for the simulation can be specified with **`timescale.`**
  - Default is 1ns with precision of 100ps

    chính xác

- Component delays are specified as #(delay)

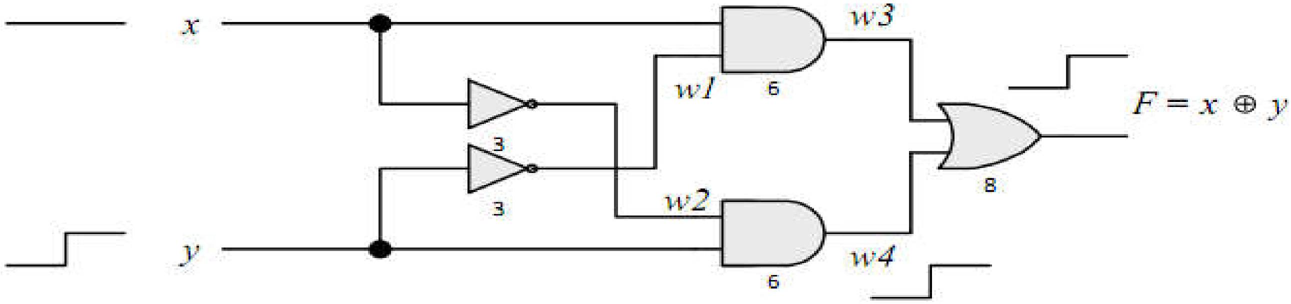
Simple Circuit with Delay



```
module circuit_with_delay (A,B,C,x,y);
   input A,B,C;
   output x,y;
   wire e;
   and #(30) g1(e,A,B);
   or  #(20) g3(x,e,y);
   not #(10) g2(y,C);
endmodule
```

Simple Circuit with Delay

Initially, with xy = 00, w1 = w2 = 1, w3 = w4 = 0 and F = 0. w1 should change to 0 **3ns** after xy changes to 01. w4 should change to 1 **6ns** after xy changes to 01. F should change from 0 to 1 **8ns** after w4 changes from 0 to 1, i.e., 14 ns after xy changes from 00 to 01.



```
`timescale 1ns/1ps
module Prob_3_33 (F, x, y);
wire w1, w2, w3, w4;

and #6 (w3, x, w1);
not #3 (w1, y);
and #6 (w4, y, w1);
not #3 (w2, y);
or  #8 (F, w3, w4);
endmodule
```

```
module t_Prob_3_33 ();
  reg x, y;
 wire F;

 Prob_3_33 M0 (F, x, y);
  initial #200 $finish;
 initial  begin
   x = 0;
   y = 0;
   #20 y = 1;
 end
endmodule
```

?

Example: Combinational Gray code

$$S_2' = \overline{Rst}\ S_2\ S_0 + \overline{Rst}\ S_1\ \overline{S_0}$$

$$S_1' = \overline{Rst}\ \overline{S_2}\ S_0 + \overline{Rst}\ S_1\ \overline{S_0}$$

$$S_0' = \overline{Rst}\ \overline{S_2}\ \overline{S_1} + \overline{Rst}\ S_2\ S_1$$

Using the primitives, **and**, **or**, & **not**

Review Questions

- What are some advantages of using HDLs, instead of schematic capture?
- What are some ways in which HDLs differ from thông thường conventional programming languages?
  - How are they similar?
- What are the different styles of Verilog coding?

Vector Declaration and Access

- Declaring a vector

  wire [7:0] my_wire; // an 8-bit vector

  wire [15:0] wire_a, wire_b; // two 16-bit vectors

- Accessing a vector

  or my_or (out, my_wire[6], my_wire[3]); //Single bit

  //Can select multiple bits from a vector

  add_4bit a4 (sum, wire_a[3:0], wire_b[15:12]);

  //If no range specified, use the full vector

  add_16bit a16 (sum, wire_a, wire_b);

nối nhau

Concatenating Vectors

- Can "build" vectors using smaller vectors and/or scalar values
- Use the {} operator
- Example

becomes
8-bit vector:
$a_1\ a_0\ b_1\ b_0\ c_1\ c_0\ d\ a_2$

```
module concatenate(out, a, b, c, d);
    input [2:0] a;
    input [1:0] b, c;
    input d;
    output [9:0] out;

    assign out = {a[1:0],b,c,d,a[2]};

endmodule
```

Arrays Of Instances

- Need several instances with similar connections?
- Can create an array of instances!
- Works for both primitives and modules

optional

- Syntax:
  <type> <delay> <array name> [<range>] (<ports>);

- Example:

```
      wire [7:0] a, b, out;
      and #4 AND8 [7:0] (out, a, b);
```

Notice that the brackets are in a different place for arrays!

Simple Array Example [1]

- Make an array of instances with each instance having same connections

```
module array_of_xor (y, a, b);
    input [3:0] a,b;
    output [3:0] y;
    xor X3 (y[3], a[3], b[3]);    // instantiates 4 xor gates
    xor X2 (y[2], a[2], b[2]);
    xor X1 (y[1], a[1], b[1]);
    xor X0 (y[0], a[0], b[0]);
endmodule
```

Simple Array Example [1]

- Make an array of instances with each instance having same connections

```
module array_of_xor (y, a, b);
    input [3:0] a,b;
    output [3:0] y;

    xor X_ALL [3:0] (y, a, b);

 // xor X_ALL [3:0] (y[3:0], a[3:0], b[3:0]);  OK too


endmodule
```

Simple Array Example [2]

- Make an array of instances with each instance having same connections

```
module array_of_flops (q, data_in, clk, set, rst);
    input [7:0] data_in;         // one per flip-flop
    input clk, set, rst;         // shared signals
    output [7:0] q;              // one per flip-flop
    /* instantiate 8 flip-flops to form an 8-bit register */
    flip_flop R7(q[7], data_in[7], clk, set, rst);
    flip_flop R6(q[6], data_in[6], clk, set, rst);
    flip_flop R5(q[5], data_in[5], clk, set, rst);
    flip_flop R4(q[4], data_in[4], clk, set, rst);
    flip_flop R3(q[3], data_in[3], clk, set, rst);
    flip_flop R2(q[2], data_in[2], clk, set, rst);
    flip_flop R1(q[1], data_in[1], clk, set, rst);
    flip_flop R0(q[0], data_in[0], clk, set, rst);
endmodule
```

Simple Array Example [2]

- Make an array of instances with each instance having same connections

```
module array_of_flops (q, data_in, clk, set, rst);
    input [7:0] data_in;          // one per flip-flop
    input clk, set, rst;          // shared signals
    output [7:0] q;               // one per flip-flop
    /* instantiate 8 flip-flops to form an 8-bit register */

    flip_flop my_flops [7:0] (q, data_in, clk, set, rst);

endmodule
```

clk, set, rst are being broadcast to all modules in the array

Complex Array Example

- Use an array to build a multi-bit adder
  - Caution: carry chain!

```
module 4_bit_adder(sum, c_out, a, b, c_in);
    input [3:0] a, b;
    input c_in;
    output [3:0] sum;
    output c_out;
    wire [3:1] c;

    Add_full M3(sum[3], c_out, a[3], b[3], c[3]);
    Add_full M2(sum[2], c[3], a[2], b[2], c[2]);
    Add_full M1(sum[1], c[2], a[1], b[1], c[1]);
    Add_full M0(sum[0], c[1], a[0], b[0], c_in);
endmodule
```

the carry signals don't line up the way the other signals do!

Complex Array Example

- Use an array to build a multi-bit adder
  - Caution: carry chain!

```
module 4_bit_adder(sum, c_out, a, b, c_in);
    input [3:0] a, b;
    input c_in;
    output [3:0] sum;
    output c_out;
    wire [3:1] c;

    Add_full M[3:0](sum, {c_out, c[3:1]}, a, b, {c[3:1], c_in});
endmodule
```

use concatenation to form vectors!

Four-Value Logic

- A single bit can have one of FOUR possible values
  - 0 Numeric 0, logical FALSE
  - 1 Numeric 1, logical TRUE

    không rõ ràng
  - x Unknown or ambiguous value
  - z No value (high impedence)

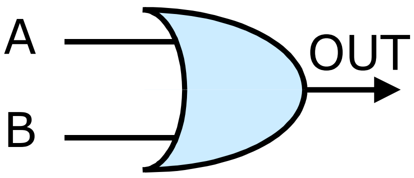
    trở kháng
- Why **x**?

  thiếu khởi tạo
  - Could be a conflict, could be lack of initialization
- Why **z**?
  - Nothing driving the signal
  - Tri-states

The x and z Values

- IN SIMULATION
  - Can detect x or z using special comparison operators
  - **x** is useful to see:
    - Uninitialized signals tín hiệu không khởi tạo
    - Conflicting drivers to a wire mâu thuẫn tới dây nối
    - Undefined behavior hoạt động không xác định
- IN REAL HARDWARE (i.e. in synthesis)
  - Cannot detect **x** or **z** as logical values
  - No actual '**x**' – electrically just isn't 0, 1, or **z**
  - Except for some uninitialized signals, x is bad!
    - Multiple strong conflicting drivers => short circuit
    - Weak signals => circuit can't operate, unexpected results
  - **z** means nothing is driving a net (tri-state)

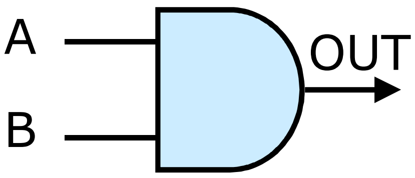
Resolving 4-Value Logic (Boolean Algebra)



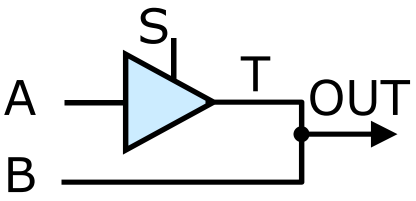
A	B	OUT
0	0	0
0	1	1
1	1	1
0	x	x
0	z	x
1	x	1
1	z	1

1

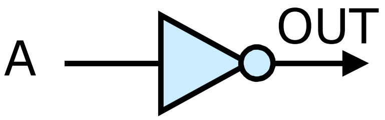
zzz



A	B	OUT
0	0	0
0	1	0
1	1	1
0	x	0
0	z	0
1	x	x
1	z	x



S	A	T	B	OUT
0	0	z	z	z
0	1	z	x	x
0	x	z	1	1
0	z	z	0	0
1	0	0	1	x
1	0	0	z	0
1	1	1	z	1
1	x	x	z	x
1	z	x	0	x



A	0	1	x	z
OUT	1	0	x	x

Representing Numbers

- Representation: <size>’<base><number> (8’d17)
  - size => number of **BITS** (regardless of base used)
  - base => base the given number is specified in
  - number => the actual value in the given base
- Can use different bases
  - Decimal (d or D) – default if no base specified!
  - Hex (h or H)
  - Octal (o or O)
  - Binary (b or B)
- Size defaults to 32 bits if unspecified
  - You should specify the size explicitly! rõ ràng
  - Why create 32-bit register if you only need 5 bits?
  - May cause compilation errors on some compilers

Number Examples

Number	Decimal value	Actual Bits
4’d3	3	0011
8’ha	10	00001010
8’o26	22	00010110
5’b111	7	00111
8’b0101_1101	93	01011101
8’bx1101	N/A	xxxx1101
-8’d6	-6	11111010

EXCEL

=DEC2BIN(-6) bù =DEC2BIN(5)

Numbers with MSB of ***x*** *or* ***z*** *extended with that value*

MSB: most significant bit

Verilog Data Types

- Two basic categories
  - Nets
    - Nets directly represent physical connections
    - Nets must be driven by something
  - Reg variables
    - Variables in behavioral Verilog
- Structural/RTL Verilog only uses Nets
- **The "reg" data type doesn't necessarily imply a physical register in synthesis…**
  - Will see this later when we discuss behavioral Verilog

Net Types

- Wire: most common, establishes connections
- Tri: indicates will be output of a tri-state
  - Same functionality as "wire", used for tri-state buses
- supply0, supply1: ground & power connections
  - Can imply this by saying "0" or "1" instead
  - xor xorgate(out, a, 1'b1);
- wand, wor, triand, trior, tri0, tri1, trireg
  - Nets with special behavior (open-drain, pull-ups, etc.)
  - Not used in this course

kéo

- See Appendix A in the text for more detail

Metaslide – SystemVerilog Slides

- This course focuses on the Verilog-2001 Standard

tập lớn

- A superset of the Verilog-2001 Standard exists, known as SystemVerilog-2009, with many new features

- We are starting to integrate teaching a few SystemVerilog features into the course with colored slides like this

Datatypes – SystemVerilog

- Nets
  - Can only be assigned values in Structural and RTL
  - Can resolve multiple drivers based on drive strength when dealing with tri-state logic
- Variables
  - Can only be assigned values in Behavioral
  - “reg” variable is unfortunately names
- New Variable in SV: “logic”
  - Can be assigned in *any* type of Verilog style
  - Only caveat: cannot resolve multiple drivers

cảnh báo

thứ bậc

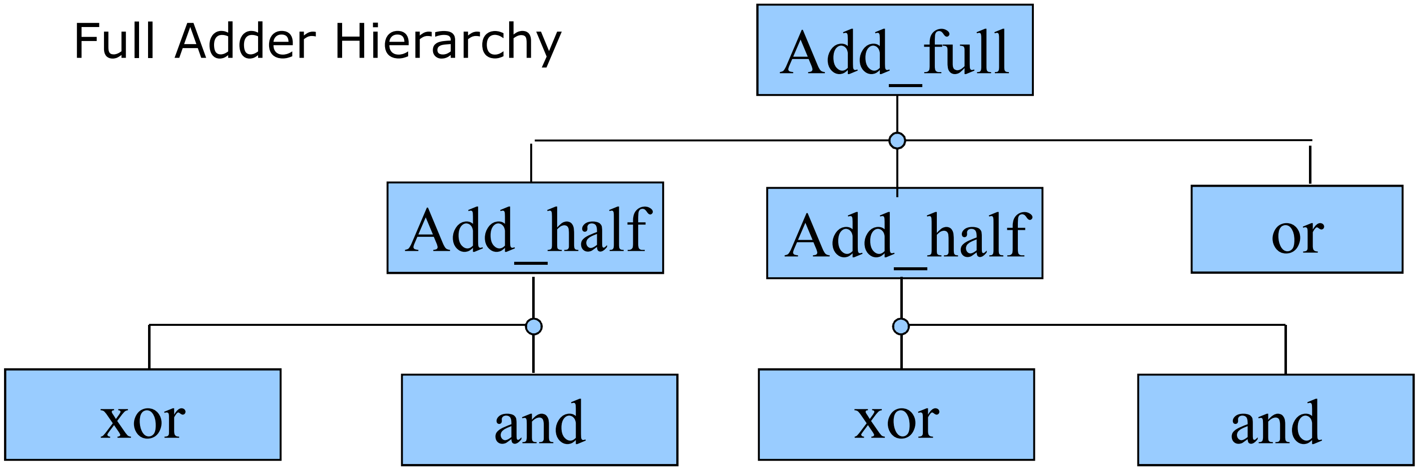
Hierarchy

- Any Verilog design you create will be a module.
  - This includes testbenches!
- Every Verilog design is a single top-level module which may or may not contain sub-modules
- Key Concepts:
  - Interface (“black box” representation)
    - Module name, ports
  - Definition
    - Describe functionality of the block
    - Includes interface
  - Instantiation
    - Use the module inside another module

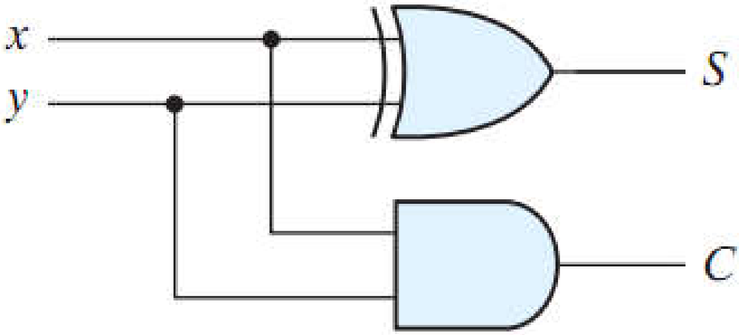
Hierarchy

- Build up a module from smaller pieces
  - Primitives and other modules
  - Can mix/match Verilog coding models (structural, etc.)
- Design: typically top-down
- Verification: typically bottom-up

Full Adder Hierarchy

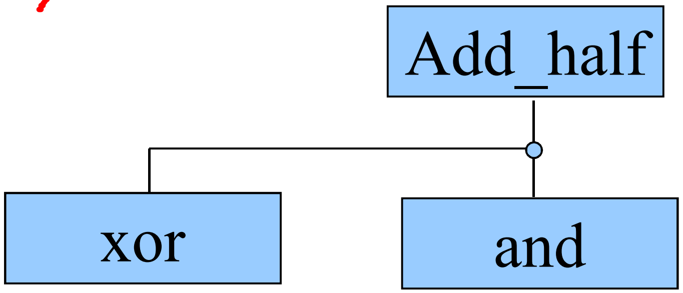


Add_half Module



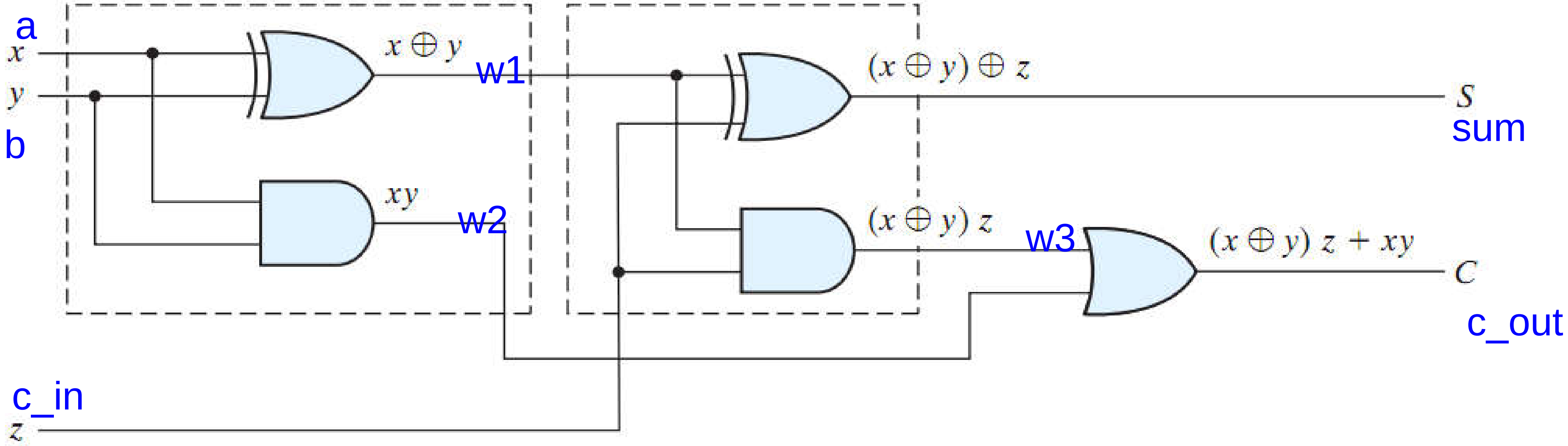
(b) $S = x \oplus y$
$C = xy$

$a = x$
$b = y$

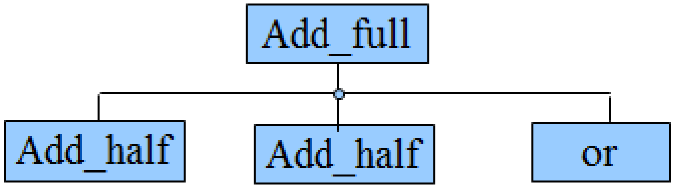


```
module Add_half(c_out, sum, a, b);
    output sum, c_out;
    input a, b;

    xor sum_bit(sum, a, b);
    and carry_bit(c_out, a, b);
endmodule
```

Add_full Module



```
module Add_full(c_out, sum, a, b, c_in) ;
    output sum, c_out;
    input a, b, c_in;
    wire w1, w2, w3;

               **
    Add_half AH1(.sum(w1), .c_out(w2), .a(a), .b(b));
    Add_half AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));
    or carry_bit(c_out, w2, w3);
endmodule
```



slide 75

Example: Gray code counter

Implement: module gray_counter(out, clk, rst);

Use: module dff(q,d,clk);
module comb_gray_code(ns, rst, out) ;

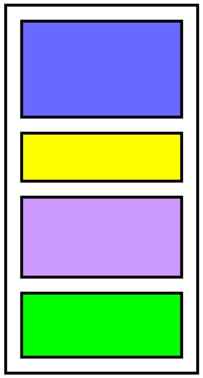
```
module gray_counter(out, clk, rst);
    output [2:0] out;
    input clk, rst;
    wire [2:0] ns;








endmodule
```

Solution: Gray code counter

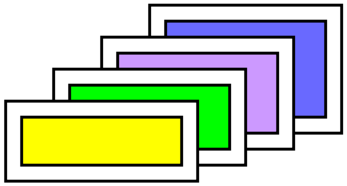
```
module gray_counter(out, clk, rst);
    output [2:0] out;
    input clk, rst;
    wire [2:0] ns;

    comb_gray_code CGC(ns, rst, out) ;
    dff DFF0(out[0], ns[0], clk) ;
    dff DFF1(out[1], ns[1], clk) ;
    dff DFF2(out[2], ns[2], clk) ;

endmodule
```

Hierarchy And Source Code

- Can have all modules in a single file
  - Module order doesn't matter!
  - Good for small designs
  - Not so good for bigger ones
  - Not so good for module reuse (cut & paste)



- Can break up modules into multiple files
  - Helps with organization
  - Lets you find a specific module easily
  - Great for module reuse (add file to project)



Structural Verilog: Connections

vị trí ẩn

- “Positional” or “Implicit” port connections
  - Used for primitives (first port is output, others inputs)
  - Gets confusing for large #s of ports
- Can specify the connecting ports by name
  - .<*port name*>(<*signal name*>)
  - Ex: `freeze_ray fr(.coolant(liquid_O2), .beam(target));`
  - Helps avoid “misconnections”
  - Don’t have to memorize or look up port order
  - Easier to read cao cấp
  - I think this way is superior. *Use it*.

Connections Examples

- Variables – defined in upper level module
  - wire [3:2] X; wire W_n; wire [3:0] word;
- By position
  - **module** dec_2_4_en (A, E_n, D);
  - dec_2_4_en DX (X[3:2], W_n, word);
- By name
  - **module** dec_2_4_en (A, E_n, D);
  - dec_2_4_en DX (.E_n(W_n), .A(X[3:2]), .D(word));

- In both cases,
  A = X[3:2],   E_n = W_n,   D = word

Empty Port Connections

- Example: module dec_2_4_en(A, E_n, D); trở kháng
  - dec_2_4_en DX (X[3:2], , word); // E_n is high impedence (z)
  - dec_2_4_en DX (.A(X[3:2]), .E_n(W_n)); // Output D[3:0] unused.
- General rules
  - Empty input ports => high impedance state (z)
  - Empty output ports => output not used
- **Specify all input ports anyway!**
  - Usually don't want z as input
  - Clearer to understand & find problems if specified
  - Unconnected inputs generate warnings in most synthesis tools.
  - Better to tie unused input to 0 or 1.

Example: 4-Entry Register File

```
module regfile_4(output [15:0] out, input [1:0] addr,
                                 input [15:0] data, input wr, clk, rst);
```

The design should use the following modules:

```
   // 2-to-4 line decoder with enable
   decoder_2_to_4(output [3:0] D, input [1:0] A, input en);

   // 16-bit register with enable, clock, and reset
   register_16bit(output [15:0] q, input [15:0] d, input en, clk, rst);

   // 16-bit 4-to-1 multiplexer
   mux_4_to_1_16bit(output [15:0] out, input [15:0] in0, in1, in2, in3,
                                              input [1:0] sel);
```

Interface: 4-Entry Register File

```
module regfile_4(output [15:0] out, input [1:0] addr,
                  input [15:0] data, input wr, clk, rst);

  wire [3:0] wr_en;
  wire [15:0] q0, q1, q2, q3;

  // decoder_2_to_4(output [3:0] D, input [1:0] A, input en);

  // register_16bit(output [15:0] q, input [15:0] d, input en, clk, rst);

  // mux_4_to_1_16bit(output [15:0] out, input [15:0] in0, in1, in2, in3,
                                              input [1:0] sel);

endmodule
```