THE UNIVERSITY OF TEXAS AT DALLAS

# CS4341 Digital Logic & Computer Design

## Lecture Notes 19

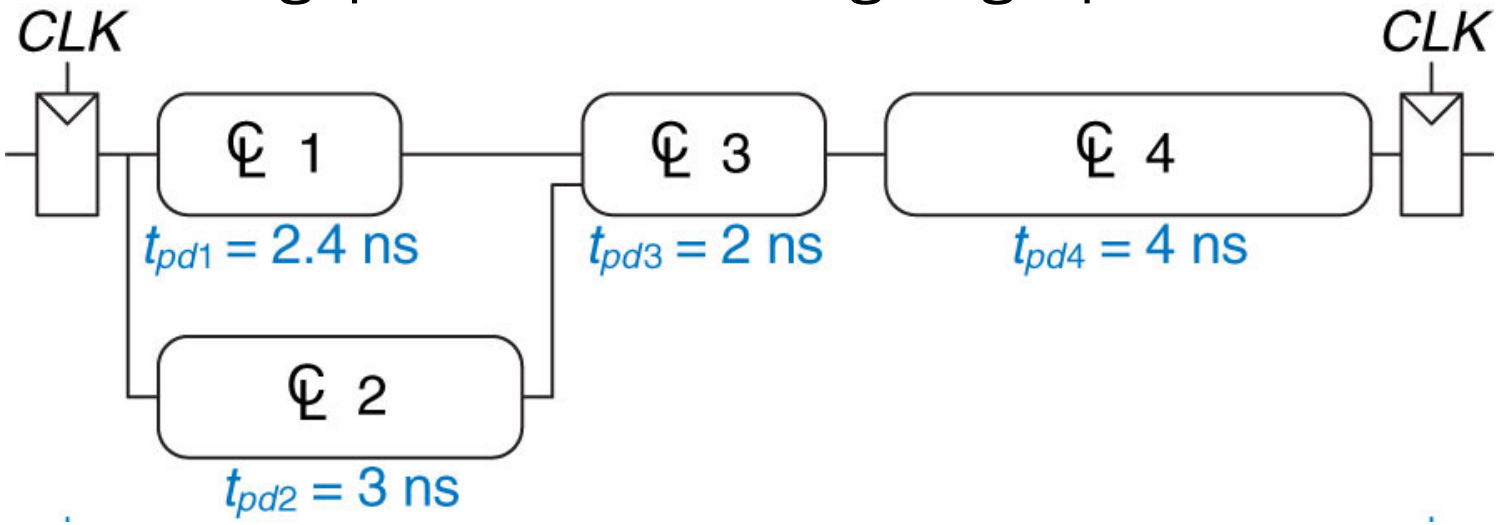
**Omar Hamdy**
Assistant Professor
Department of Computer Science

# Review: Parallelism

- In digital circuits, if a task has latency L, in a single process system, the throughput is 1/L
- In a spatially parallel system with N copies of the hardware, the throughput is N/L
- In pipelined system, if the task is broken into N equal stages, then throughput is N/L. Otherwise, the throughput is $1/L_1$, where $L_1$ is the longest step.

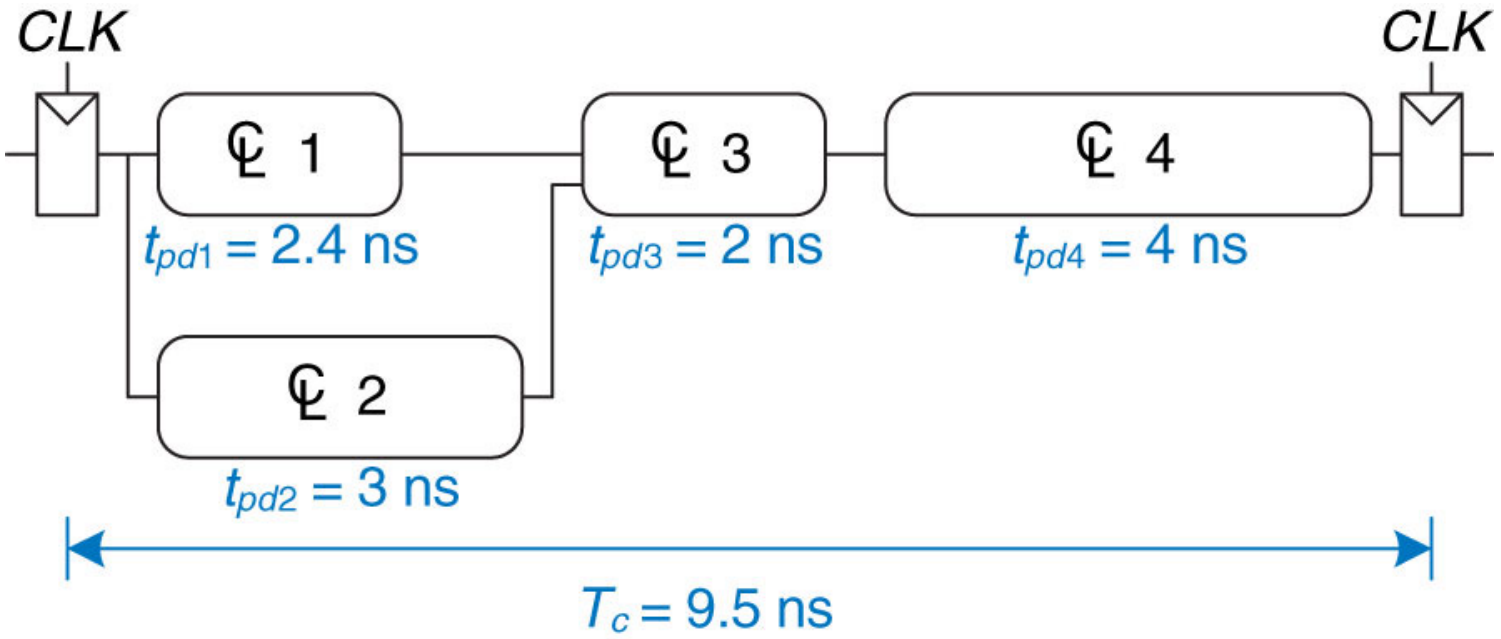
# Pipelining Implementation

- In pipelining, the combinational circuit is divided into smaller stages by insertion of registers.
- Registers are important to prevent tokens from catching up with one another.
- Calculate the throughput of the following single process circuit



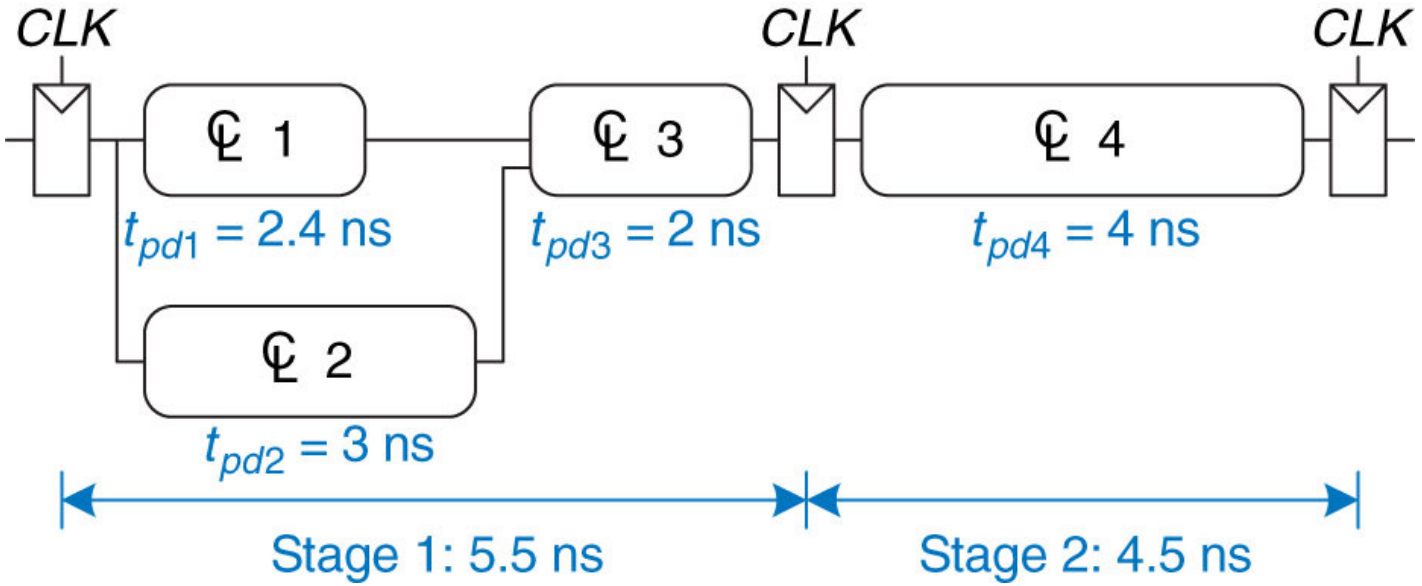
# Throughput Calculation – No Pipelining

- First, we calculate the latency, which is Tc in this circuit (why)?
  - $T_c = t_{pcq} + t_{pd} + t_{setup}$
  - $t_{pd}$ = sum of all $t_{pd}$ on the critical path (CL 2, 3 and 4) = 3 + 2 + 4 = 9 ns
  - Assume $t_{pcq}$ = .3 ns and $t_{setup}$ = 0.2 ns
  - $T_c$ = 9 + .5 = 9.5 ns ➔ throughput $1/T_c$ = 105 MHz



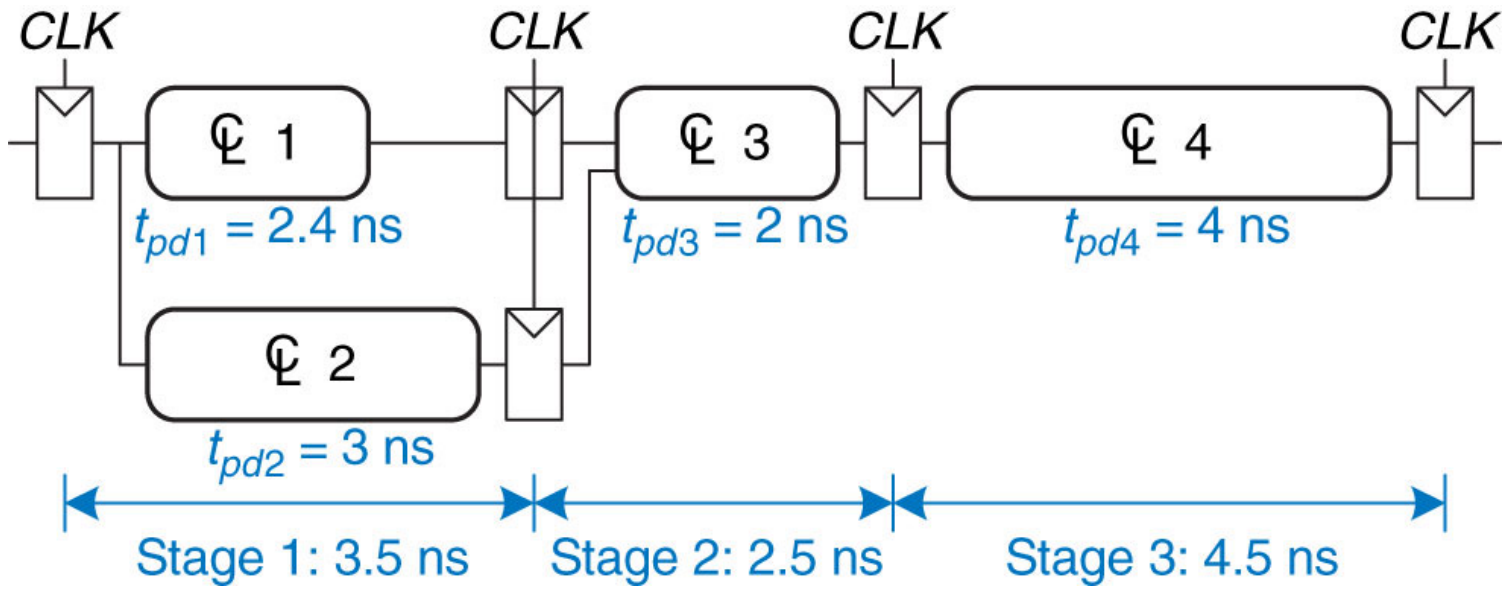
# Throughput Calculation – 2 Stage Pipeline

- First, we calculate the latency as $T_c$ in each stage.
  - $T_{c1}$ = 5 + .5 = 5.5 ns
  - $T_{c2}$ = 4 + .5 = 4.5 ns
  - System clock has to follow the slower $T_c$, hence, Circuit $T_c$ = 5.5 ns
  - Therefore, throughput $1/T_c$ = 182 MHz. However, latency L is now 5.5 x 2 = 11 ns (why?)



# Throughput Calculation – 3 Stage Pipeline

- Again, we calculate the latency as $T_c$ in each stage.
  - $T_{c1}$ = 3 + .5 = 3.5 ns
  - $T_{c2}$ = 2 + .5 = 2.5 ns
  - $T_{c2}$ = 4 + .5 = 4.5 ns
  - Therefore, throughput $1/T_c$ = 1/4.5ns = 222 MHz. However, latency L = 13.5 ns



# Hardware Descriptive Language

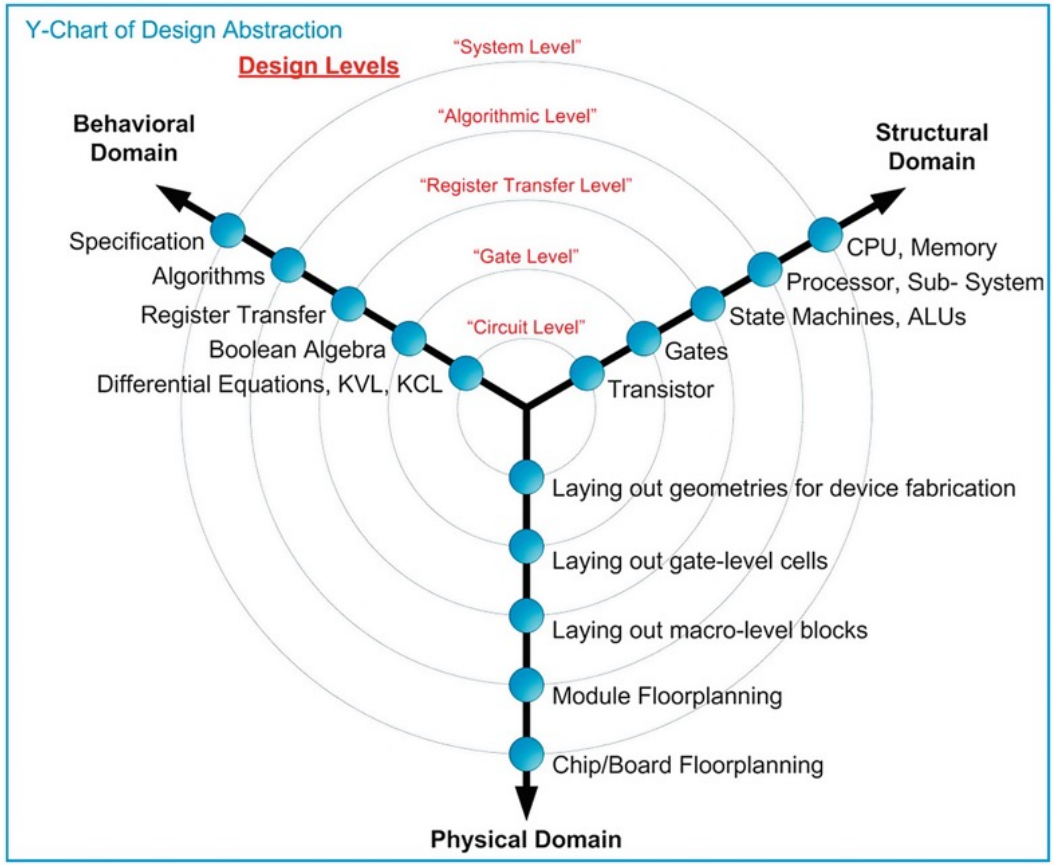
- The process of designing a digital system end-to-end including function simplification, FSM translation and gate selection is very challenging.
- Instead, the idea was to work at a higher-level of abstraction, and focus only on the functional design, then use a computer-aided design (CAD) tool to optimize and produce the actual gate blueprints.
- The CAD tool language is known as Hardware Descriptive Language (HDL)
- The two leading hardware descriptive languages are VHDL and Verilog
- Both VHDL and Verilog are built on similar principles, but they differ in the syntax.

# How to Learn the Language

- Both VHDL and Verilog are vast languages with lots of syntax. It will be impractical to try to learn and master the entire language before using it.
- Instead, you follow a gradual approach in looking at small parts of the language that are relevant to your domain or work, look at examples, then apply to similar problems you are trying to solve.
- Remember, HDL is not a program intended to run and produce outputs. Instead, it is just a helping tool to verify designs (simulate) and to synthesize.
- For the purpose of this class lectures, VHDL will be used as the reference language.
- VHDL stands for: Very High Speed Integrated Circuits Descriptive Language

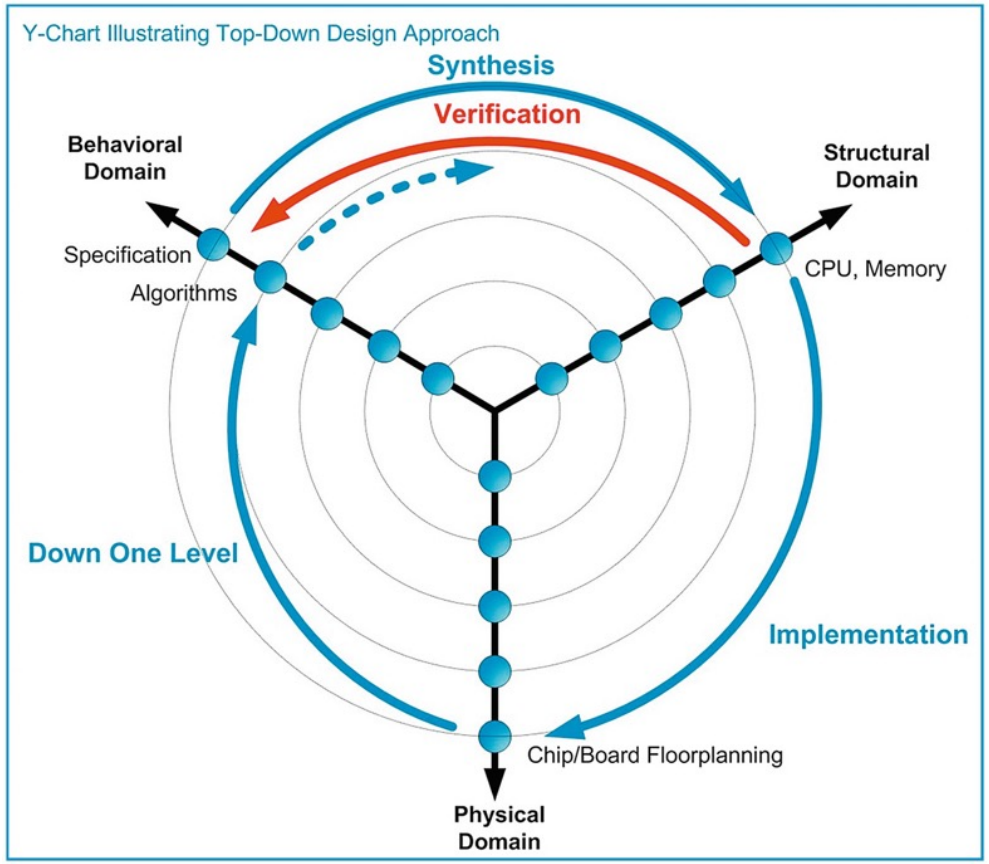
# Domains of Description

- One famous model used for hierarchal abstraction of the hardware design is the Gajski-K Y-chart



# Domains of Description

- VHDL deals only with the behavioral and the structural domains



# HDL – Entity and Architecture

- Entity and Architecture are the two most used modules in HDL languages.
  - Entity: Is a block of hardware with input and outputs.
  - Architecture: The description of the internal build (functionality) of an entity.
  - There are two common architectural models:
    - Behavioral: it describes what an entity or a module does.
    - Structural: it describes how an entity or a module is built from simpler pieces

# VHDL – "Hello World" Example

- The following example defines an entity which computes the Boolean function of $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$. The entity has 3 inputs and 1 output.
- The example illustrates the behavioral architecture of this entity

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity simplefunction is
  port (a, b, c: in STD_LOGIC;
        y: out STD_LOGIC);
end;
architecture synth of simplefunction is
  begin
   y <= ((not a) and (not b) and (not c)) or
        (a and (not b) and (not c)) or
        (a and (not b) and c);
end;
```

The library `use` clause is a must, because the types are not built in the language itself. Other libraries include `IEEE.STD_LOGIC_UNSIGNED` and `IEEE.STD_LOGIC_SIGNED`

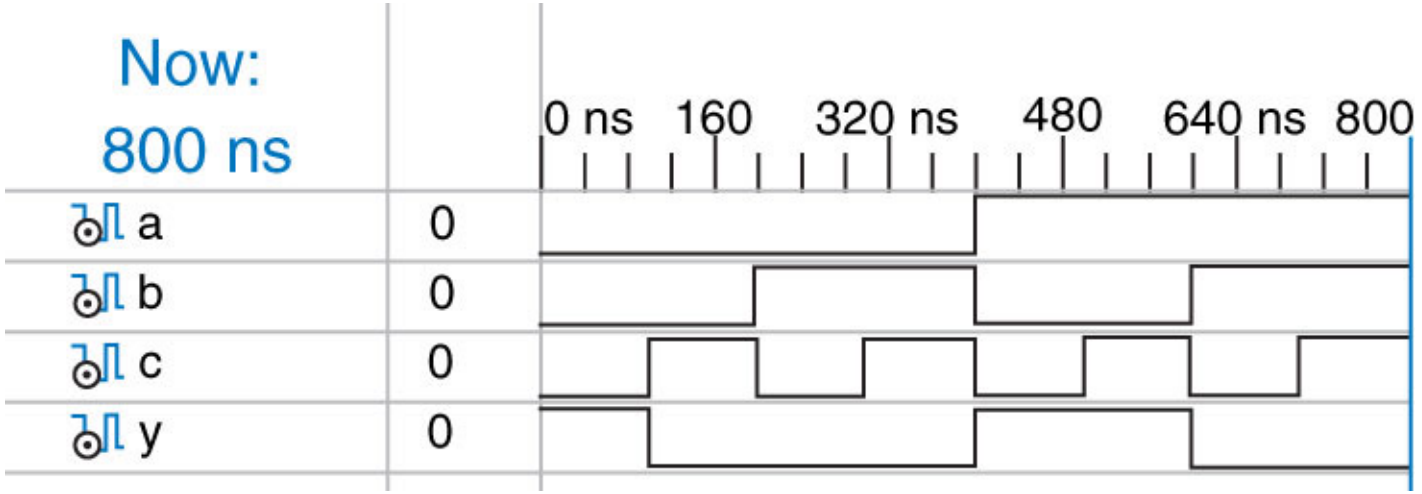
The `entity` declaration lists the module name and its inputs and outputs

The `architecture` body defines what the module does.

- VHDL signals, such as inputs and outputs, must have a type declaration of STD_LOGIC type.
- STD_LOGIC signals can be '0' or '1', floating and undefined
- VHDL lacks a good default order of operations, so Boolean equations should be parenthesized.

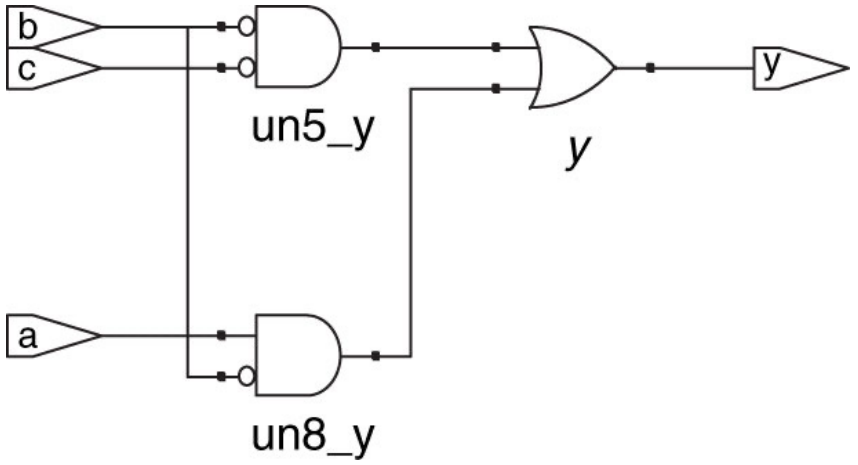
# Simulation Use of HDL

- Simulation is a key benefit of HDL, because testing and verifying a design for correctness can be very difficult to accomplish in the lab.
- Also, system defects (bugs) after manufacturing can sometimes be very expensive. Example, Pentium processor floating point division bug (FDIV).
- Simulating the previous example would produce the following



# Synthesis Use of HDL

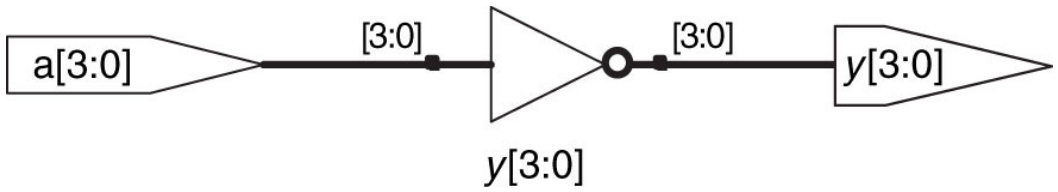
- Logic synthesis transforms HDL code into a netlist describing the hardware (e.g., the logic gates and the wires connecting them).
- The logic synthesizer might perform optimizations to reduce the amount of hardware required.
- The netlist may be a text file, or it may be drawn as a schematic
- Not all of the language commands can be synthesized (for example print command)
- Synthesis output of the previous example would produce the following



# VHDL – C. Logic: Bitwise Operators

- Bitwise operators act on single-bit signals or on multi-bit busses
- STD_LOGIC_VECTOR (3 downto 0) means 4-bit bus where 3 is the msb.
- The vector could be declared in any order or using any count. Example: STD_LOGIC_VECTOR (1 to 4). However, you need to keep consistency

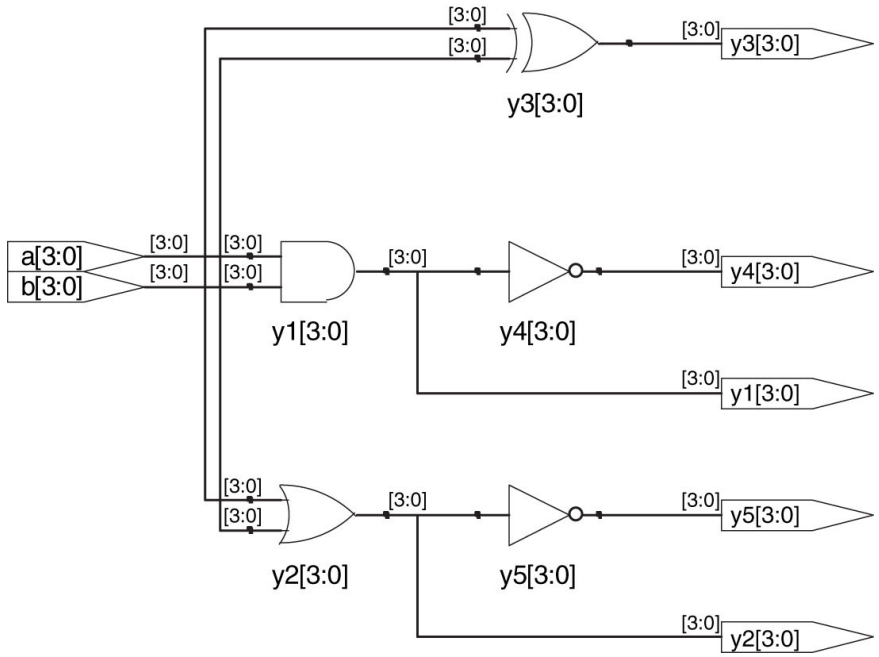
```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
  port (a: in STD_LOGIC_VECTOR (3 downto 0);
        y: out STD_LOGIC_VECTOR (3 downto 0));
  end;
architecture synth of inv is
begin
  y <= not a;
end;
```



# VHDL – C. Logic: Bitwise Operators

- Unlike normal programming languages, HDL introduces concurrent signal assignment.
- In any assignment statement out1 <= in1 op in2, if the input signals (operands) change at anytime, the output signal is updated accordingly.

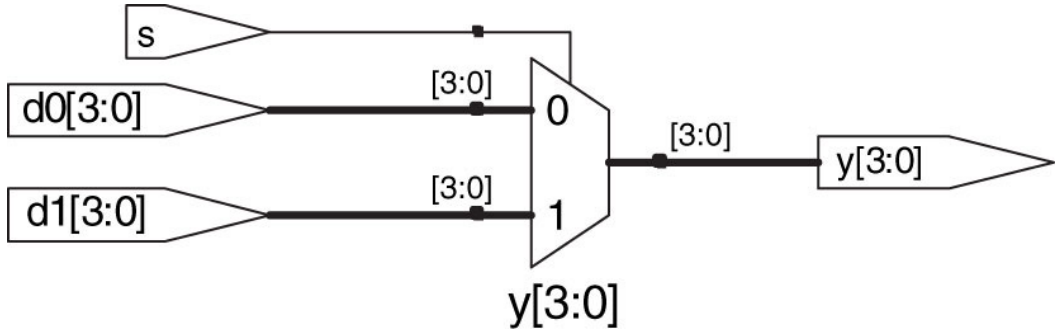
```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity gates is
  port (a, b: in STD_LOGIC_VECTOR (3 downto 0);
          y1, y2, y3, y4,
          y5: out STD_LOGIC_VECTOR (3 downto 0));
end;
architecture synth of gates is
begin
— — Five different two-input logic gates
— —acting on 4 bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```



# VHDL – C. Logic: Conditional Assignment

- Conditional assignment is when HDL selects the output from among alternatives based on an input called the condition. What does that remind you of?

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity mux2 is
  port (d0, d1: in STD_LOGIC_VECTOR (3 downto 0);
        s: in STD_LOGIC;
        y: out STD_LOGIC_VECTOR (3 downto 0));
end;
architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

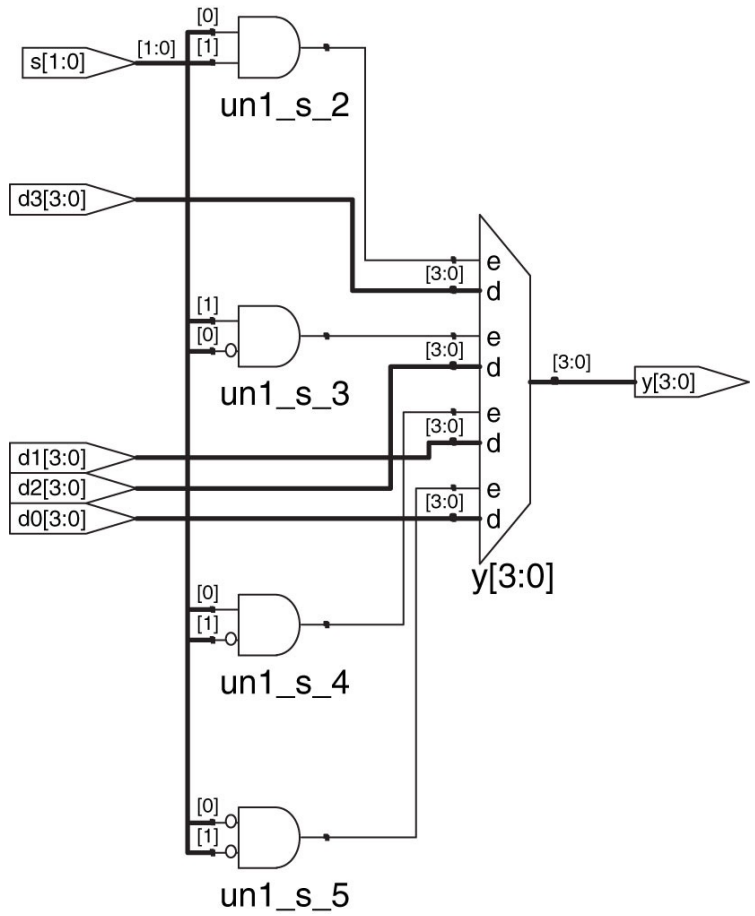


# VHDL – C. Logic: 4x1 MUX

- In additional to conditional assignment, *select* signal assignment statements is a better option when selecting from one of several possibilities

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity mux4 is
  port (d0, d1,
        d2, d3: in STD_LOGIC_VECTOR (3 downto 0);
        s: in STD_LOGIC_VECTOR (1 downto 0);
        y: out STD_LOGIC_VECTOR (3 downto 0));
end;
architecture synthl of mux4 is
begin
  y <= d0 when s = “00” else
  d1 when s = “01” else
  d2 when s = “10” else
  d3;
end;
```

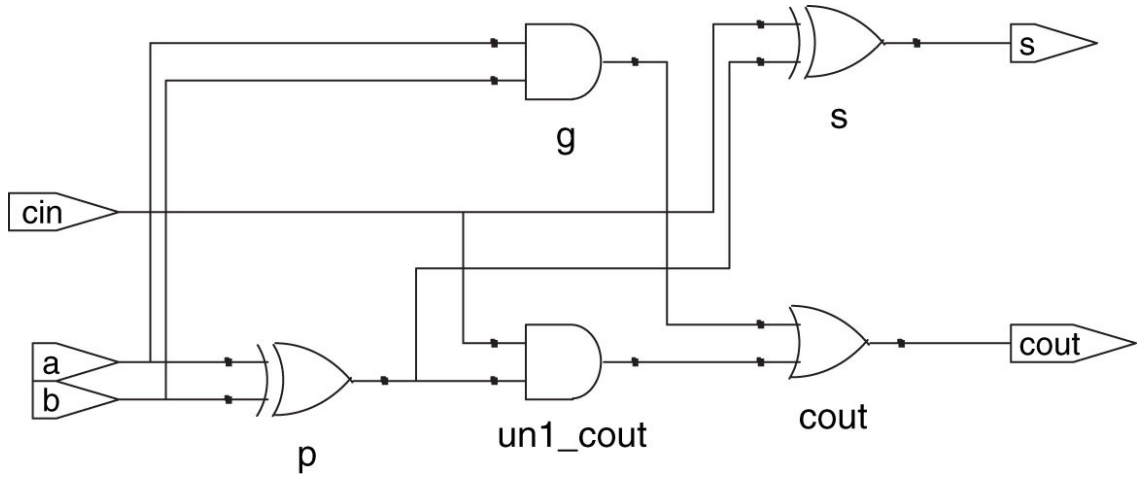
```
begin
  with s select y <=
  d0 when “00”,
  d1 when “01”,
  d2 when “10”,
  d3 when others;
end;
```



# VHDL – C. Logic: Internal Variables

- Internal variables are useful when breaking statements into intermediate steps.
- Internal variables are neither input nor output signals

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity fulladder is
  port(a, b, cin: in STD_LOGIC;
       s, cout: out STD_LOGIC);
end;
architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;
  s <= p xor cin;
  cout <= g or (p and cin);
end;
```



# To Do List

- Review lecture notes
- Try all the HDL examples
- Study chapter 4