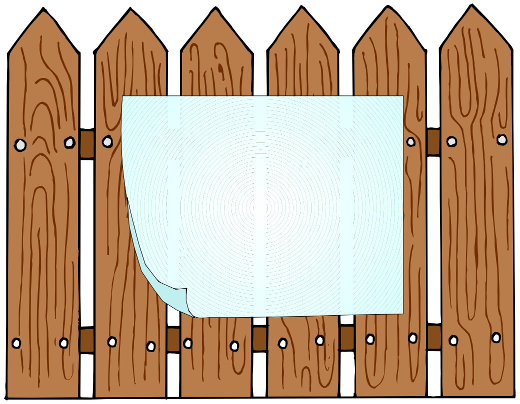
Other combinational circuit topics



- Today we'll mention several miscellaneous but important circuit topics.
  - There are a few additional gates that are often used in logic design aside from the AND, OR and NOT gates that we've already seen.
  - Up to now we haven't talked about how much time it takes for gates to operate.
  - It's possible to design hardware using special programming languages.

NAND

- The NAND of two inputs x and y is the complement of their product.

  $(xy)'$

- We know from DeMorgan's Law that this can also be written as:

  $x' + y'$

- There are two equivalent logic gate symbols for NAND, corresponding to the two equivalent expressions.
- The NAND gate is universal, since every function can be implemented using only NAND gates!

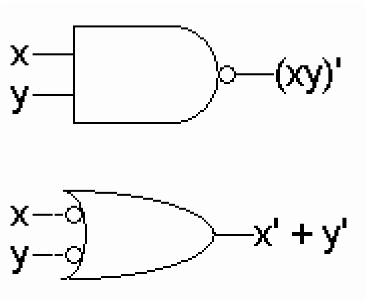
Operation: NAND
(NOT-AND)

Expressions: $(xy)' = x' + y'$

Truth table:

x	y	(xy)'
0	0	1
0	1	1
1	0	1
1	1	0

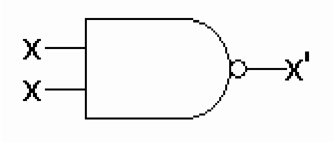
Logic gates:



NAND gates are universal

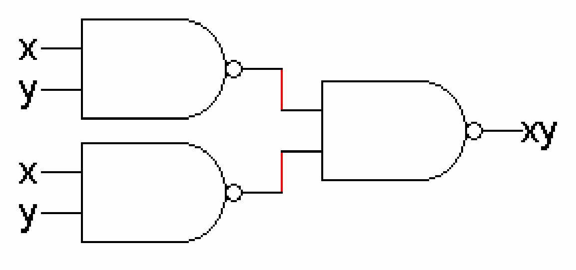
- Here is how you can use NAND to implement the other basic logic gates.

NOT



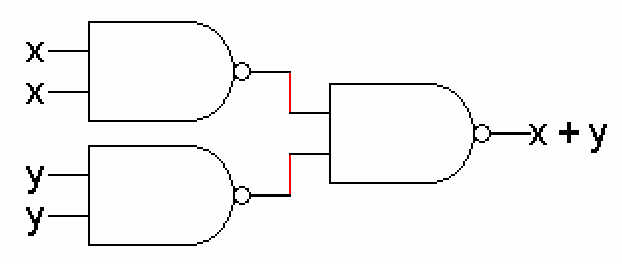
$(xx)' = x'$

AND



$((xy)'\,(xy)')' = xy$

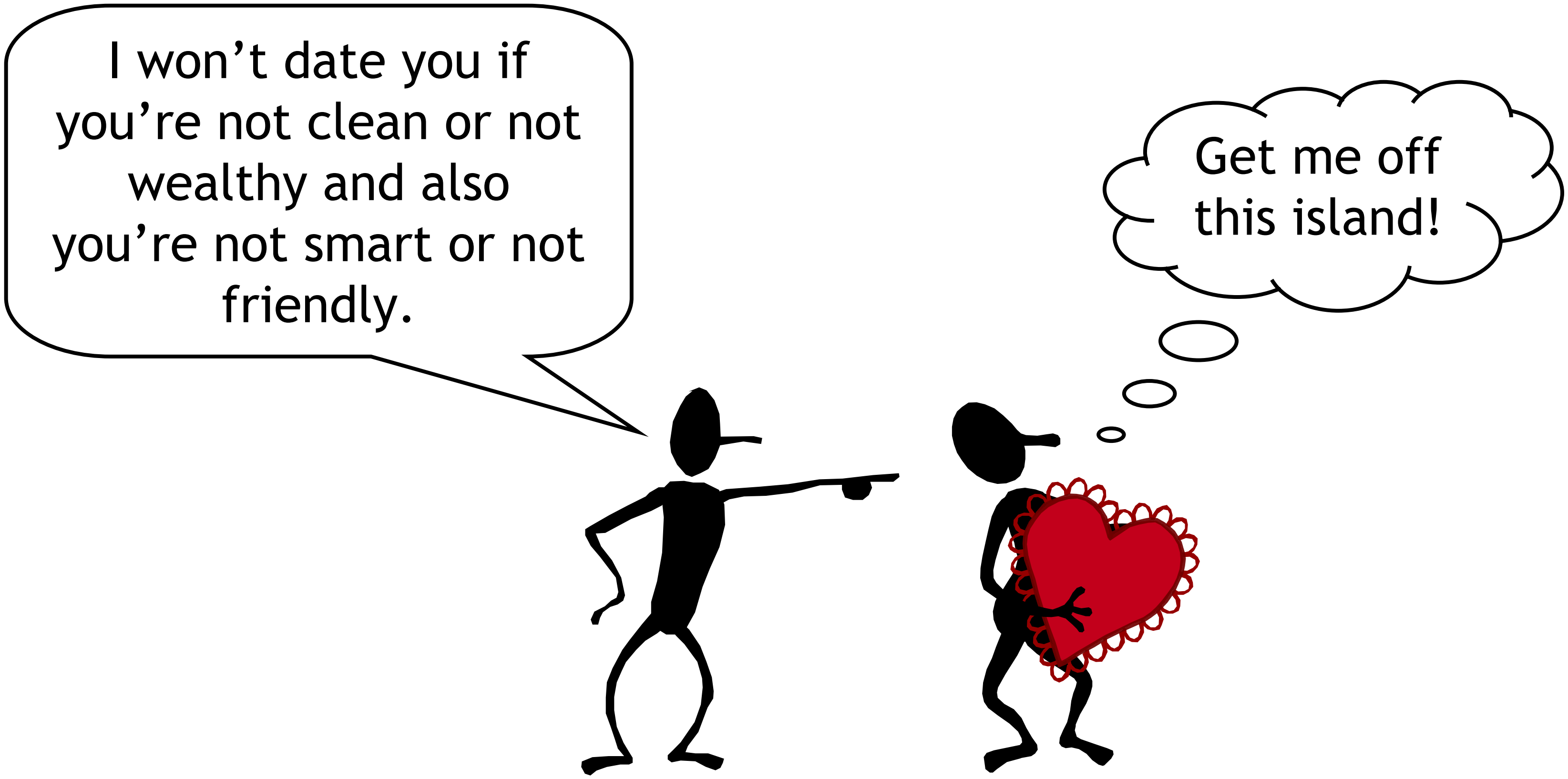
OR



$$\begin{aligned} ((xx)'\,(yy)')' &= (x'\,y')' \\ &= x + y \end{aligned}$$

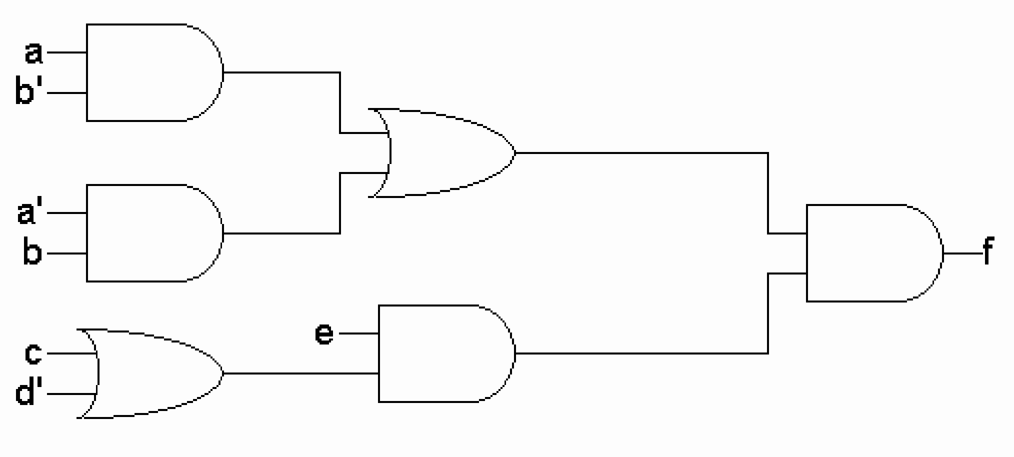
- If you know how to convert the primitive gates to NAND gates, then you can convert any circuit to one that includes only NANDs.

But NAND gates are weird…

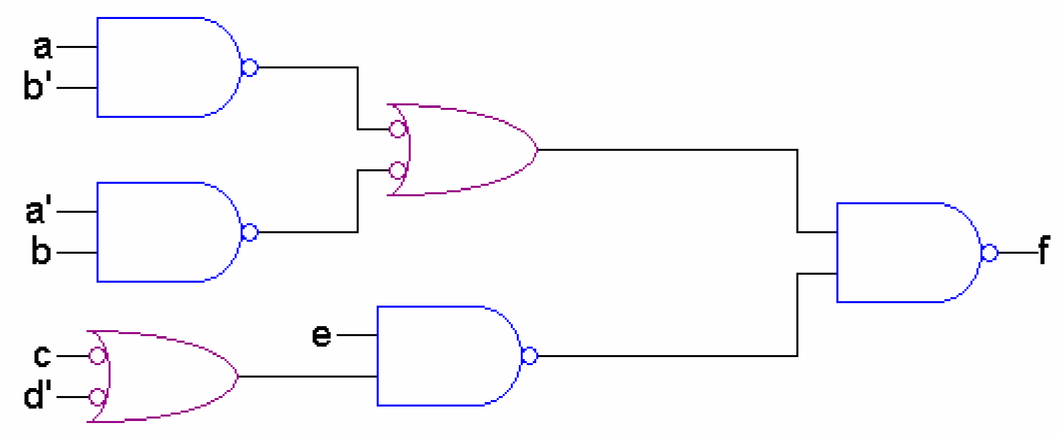


Making NAND circuits I

- The easiest way for humans to make a NAND-only circuit is to start with a regular diagram designed with primitive gates.

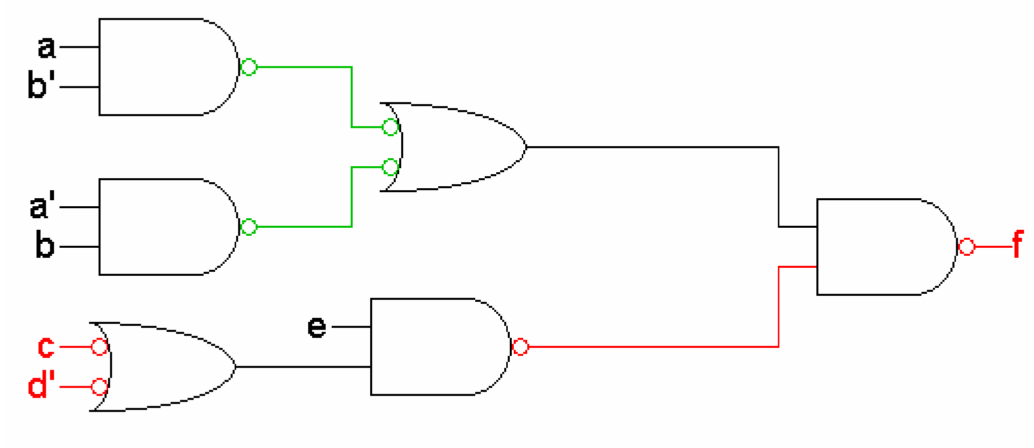


- First convert all AND and OR gates to NAND gates. This is easy if you use the AND-NOT and NOT-OR symbols respectively.

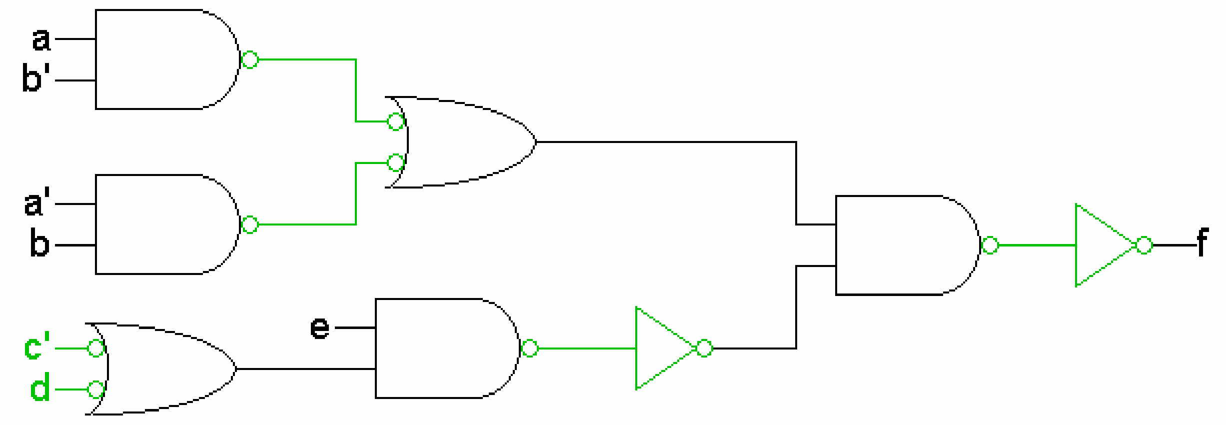


Making NAND circuits II

- Then make sure you added inverters to lines *in pairs*, as shown in green.



- We must add inverters or complement the input variables for the red lines, to ensure that the original function is not modified ((x')' = x).



NOR

- The NOR of two inputs x and y is the complement of their sum.

$$(x + y)'$$

- This is equivalent to:

$$x'y'$$

- There are two equivalent logic gate symbols for NOR, based on the two equivalent expressions.
- NOR gates are also universal.
  - All of the primitive operations can be defined in terms of NOR.
  - Any Boolean function can be implemented with a NOR-only diagram.

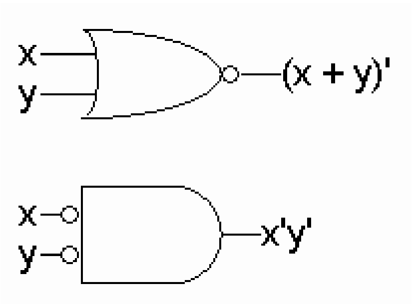
Operation: NOR (NOT-OR)

Expressions: $(x + y)' = x'y'$

Truth table:

x	y	(x + y)’
0	0	1
0	1	0
1	0	0
1	1	0

Logic gates:



XOR

- The XOR of two inputs x and y is true when exactly one of the inputs is true—in other words, xy=01 or xy=10.

$$x \oplus y = x'y + xy'$$

- Another way to think about this is that $x \oplus y$ is true when x and y are different.
- This corresponds more closely to the normal English usage of "or," as in "eat your meat or you won't get any pudding."

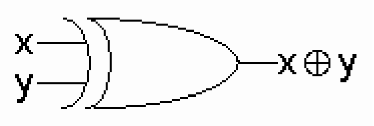
Operation: XOR (eXclusive OR)

Expression: $x \oplus y$

Truth table:

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Logic gate:



Interesting XOR properties

- There are several fascinating properties of XOR that you can prove using Boolean algebra, starting from the definition $x \oplus y = x'y + xy'$

$x \oplus 0 = x$	$x \oplus 1 = x'$
$x \oplus x = 0$	$x \oplus x' = 1$
$x \oplus (y \oplus z) = (x \oplus y) \oplus z$	Associative
$x \oplus y = y \oplus x$	Commutative

- We will meet the first two laws in this table again next week.
  - The exclusive or of any value and zero is that value.
  - The XOR of any value and one is the complement of that value.

More XOR tidbits

- XOR can be extended to an arbitrary number of arguments.
- In general, the XOR function is true when an *odd* number of its inputs are true.
  - For instance, you can simplify an XOR of three inputs to the expression and truth table on the right.
  - The output is true only when either 1 or 3 of the inputs are true.
- XOR is especially useful in building adders as we'll see soon, and error detection and correction circuits.

$$x \oplus (y \oplus z) =$$
$$x'y'z + x'yz' + xyz + xy'z'$$

x	y	z	$x \oplus y \oplus z$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

XNOR

- Finally, the XNOR of two inputs x and y is true when x and y are the same—in other words, when xy=00 or xy=11.

$$(x \oplus y)' = x'y' + xy$$

- Notice that the XNOR function is the complement of the XOR.

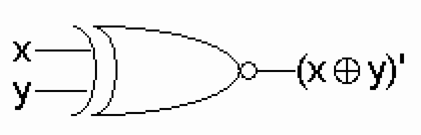
Operation: XNOR (eXclusive NOR)

Expression: $(x \oplus y)'$

Truth table:

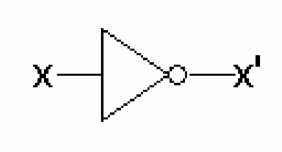
x	y	$(x \oplus y)'$
0	0	1
0	1	0
1	0	0
1	1	1

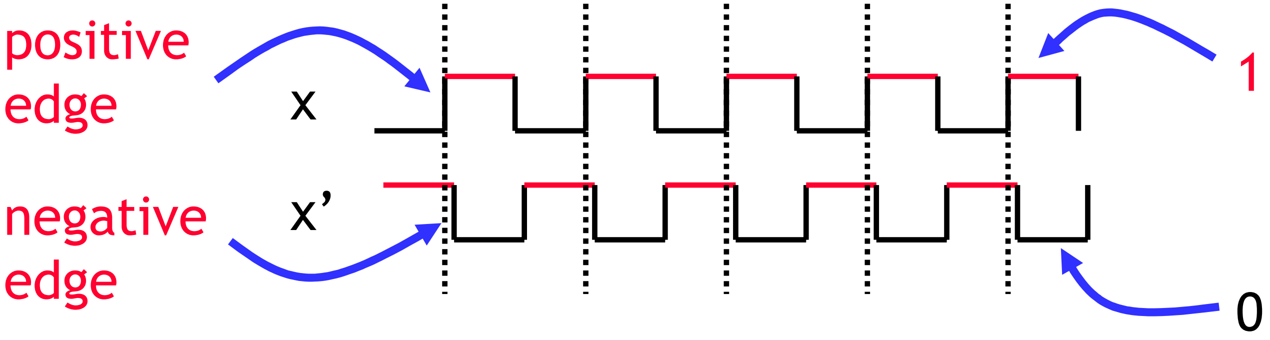
Logic gate:



Delays and timing diagrams

- Although we haven’t said anything up to now, gates take time!
- The time it takes for a gate’s outputs to change in response to an input change is the propagation delay. The time is mainly required to apply or drain voltages in the lower-level transistors.
- Timing diagrams are used to show delays.

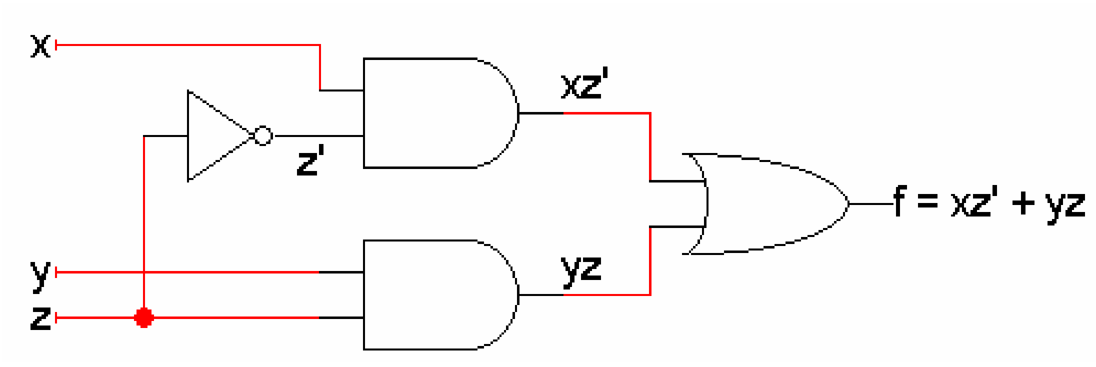


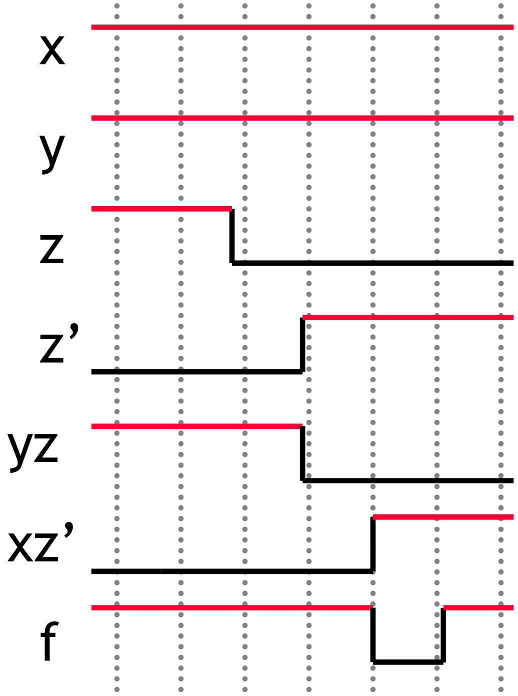


- This example, for an inverter, shows how signal values, on the vertical axis, change with time on the horizontal axis.
  - When the input becomes 1, the output becomes 0, and vice versa
  - There is a slight delay before the outputs changes

Hazards

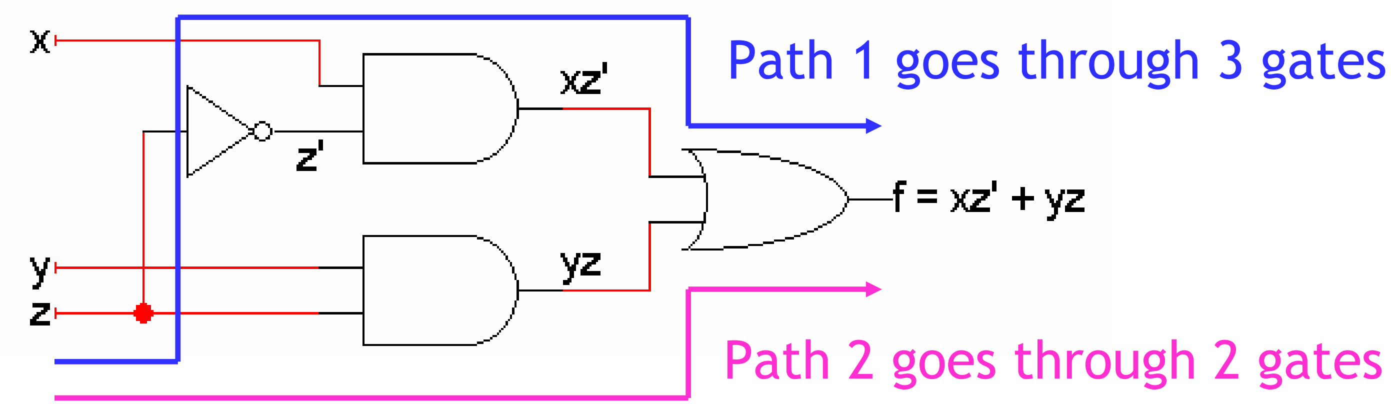
- In the circuit below, the output f should remain 1 if the inputs change from xyz = 111 to xyz = 110.
- But what really happens? There is a glitch, and f becomes 0 temporarily!
  - z' and yz change right after z changes.
  - But *xz'* has *not* changed yet, so f then (incorrectly) changes to 0.
  - Only after xz' changes does f go back to 1.





Hazards are a difficult problem

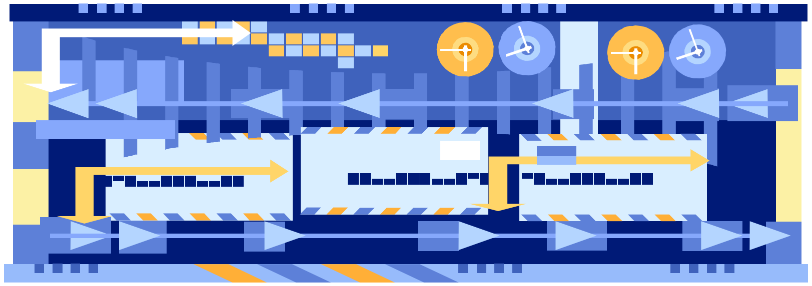
- Part of the problem here is that there are multiple paths from the inputs (z) to the outputs, and some paths are longer than others.
- Hazards can be very difficult to detect and prevent in general.



Path 1 goes through 3 gates

Path 2 goes through 2 gates

Hardware description languages

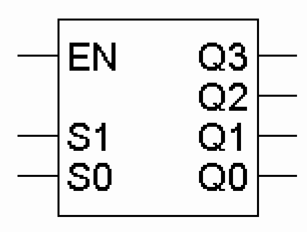


- It’s possible to describe hardware textually instead of graphically, using a hardware description language that’s a little like C or Java.
- These languages provide functionality like that of LogicWorks.
  - Built-in parts libraries provide basic gates and devices.
  - You can create your own parts and build circuits hierarchically.
  - Devices are connected together to make a complete circuit.
  - A simulator allows you to test the resulting design.
- We’ll present a brief introduction to VHDL (Very High-speed Integrated Circuit Hardware Description Language), with small examples to give you the flavor of the language.

Entities

- An entity in VHDL describes the inputs and outputs of a device. It’s like the block symbols we often use, or a function header in C or Java.
- For instance, a 2-to-4 decoder has three inputs and four outputs, each of which is a single bit with a VHDL type std_logic.

```
ENTITY Decoder4 IS
    PORT (en, s1, s0: IN std_logic;
          q3, q2, q1, q0: OUT std_logic);
END Decoder4;
```



A basic decoder implementation

- An architecture describes the implementation of a device, similar to how a function body in C or Java implements some task.
- Here is a simple 2-to-4 decoder implementation in VHDL.

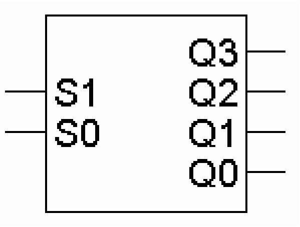
```
ARCHITECTURE Dataflow OF Decoder4 IS
BEGIN
    q3 <= en AND s1 AND s0;
    q2 <= en AND s1 AND (NOT s0);
    q1 <= en AND (NOT s1) AND s0;
    q0 <= en AND (NOT s1) AND (NOT s0);
END Dataflow;
```

- Remember that these signals were already declared before, in an entity.

Vector types

- You can also group bits into vectors or arrays.
  - The 2-to-4 decoder entity below has a one-bit input EN and a two-bit input S, which consists of bits S(1) and S(0).
  - There is a "single" output Q, which consists of bits Q(0) to Q(3).

```
ENTITY Decoder4b IS
    PORT (s: IN std_logic_vector(1 DOWNTO 0);
          q: OUT std_logic_vector(3 DOWNTO 0));
END Decoder4b;
```



An alternative decoder implementation

- A behavioral specification expresses *what*, rather than *how*.
- The alternative decoder implementation below uses a case-like statement to describe the four-bit output Q in terms of the two-bit input S.
- This can be automatically translated into a circuit—and we didn't write a single Boolean expression!

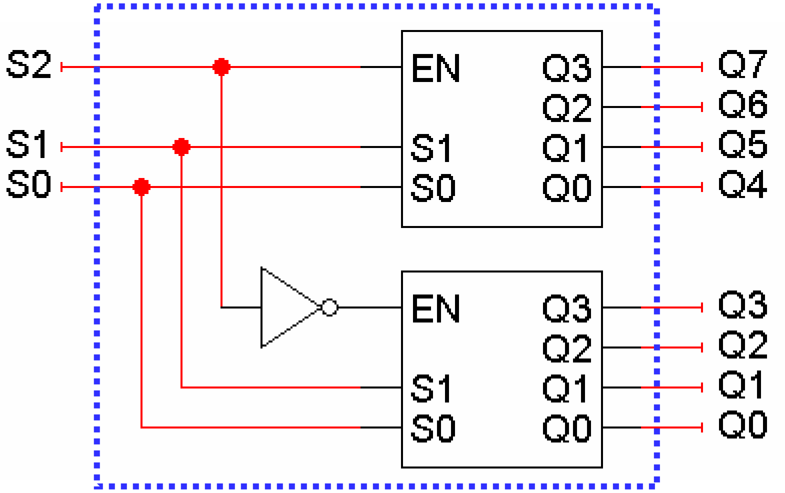
```
ARCHITECTURE Behavioral OF Decoder4b IS
BEGIN
    q <= "0001" WHEN s = "00" ELSE
         "0010" WHEN s = "01" ELSE
         "0100" WHEN s = "10" ELSE
         "1000";
END Behavioral;
```

Top-down design

- VHDL systems can translate behavioral descriptions into circuits.
  - This makes it pretty easy to build a complex device.
  - The disadvantage is that a compiler-generated circuit may not be as efficient as one you design by hand.
- A lot of VHDL designers use a top-down methodology.
  - Behavioral specifications are written first, to get everything up and running.
  - Then individual entities can be slowly refined with more efficient, lower-level descriptions.

A 3-to-8 decoder

- Let’s combine two 2-to-4 decoders to make a 3-to-8 decoder in VHDL.



- First, a VHDL entity declaration is shown below.

```
ENTITY Decoder8 IS
    PORT (s2, s1, s0: IN std_logic;
          q7, q6, q5, q4, q3, q2, q1, q0: OUT std_logic);
END Decoder8;
```

Composing devices in VHDL

- Here is the VHDL architecture for the 3-to-8 decoder.
  - We reuse our 2-to-4 decoder by declaring it as a component.
  - A port map specifies the inputs and outputs for a component. It’s like a function call in C or Java.
  - An internal signal E2 represents the negation of input S2.
- The outputs Q0-Q7 are taken directly from the 2-to-4 decoders.

```
ARCHITECTURE Dataflow OF Decoder8 IS
  COMPONENT Decoder4
    PORT (en, s1, s0: IN std_logic;
          q3, q2, q1, q0: OUT std_logic);
  END COMPONENT;
  SIGNAL e2: std_logic;
BEGIN
  e2 <= NOT s2;
  block1: Decoder4 PORT MAP (s2, s1, s0, q7, q6, q5, q4);
  block0: Decoder4 PORT MAP (e2, s1, s0, q3, q2, q1, q0);
END Dataflow;
```

Modularity

- Modularity is an important design principle in building any large system, whether it's a program or a circuit.
- A VHDL entity describes a circuit's inputs and outputs, while there are many ways to specify the implementation.
  - A behavioral description expresses what the circuit does at a higher level, without giving implementation details.
  - A dataflow description supplies an actual low-level design, perhaps using Boolean expressions or other subdevices.
- High-level languages provide similar features.
  - Headers and interfaces specify the inputs and outputs of a function or method, but not how it works.
  - Implementations with actual code are usually stored in separate files, and there can be many implementations of one function.
- These features help make VHDL much better than LogicWorks for large-scale circuit design involving many people.

Summary

- There are several useful additional logic gates.
  - NAND and NOR are universal gates which can replace all others.
  - XOR implements the "odd" function, and XNOR is its complement.
- Gates and circuits all have propagation delays.
  - Delays can be shown explicitly on timing diagrams.
  - Sometimes delays can lead to undesirable hazards.
- Hardware description languages like VHDL are another way to specify and design circuits, applying programming language ideas to hardware.
- Next week we'll talk about arithmetic circuits for addition, subtraction, and multiplication.