Module 8: Signal Propagation Delays

Revision: September 10, 2007

Overview

This module considers the time-course of logic signals as they pass through logic circuits. Until now, we have not considered the time required for logic signals to propagate through logic gates and along signal wires. Instead, we have been assuming that logic gate outputs change from '0' to '1' or '1' to '0' immediately (i.e., in zero time). Further, we have assumed that in response to input changes, logic circuit outputs either remain constant or change immediately to new values. This simplifying approach was justified because it allowed us to focus on the logical properties of circuits. But now, it is time to examine the behavior of real logic circuits, where voltage levels cannot change immediately.

Before beginning this lab, you should:

- Be familiar with combinational logic circuits of all sorts, from basic SOP and POS circuits through more complex arithmetic and logic designs.
- Be able to design and simulate structural and behavioral circuits using VHDL and/or schematic capture in the Xilinx ISE/WebPack tool;
- Be able to download circuits to the Digilent board;
- Be familiar arithmetic circuits and the bit-slice design method.

After completing this lab, you should:

- Be comfortable in approaching more complex design problems;
- Understand the value of partitioning a design properly;
- Appreciate the utility and trade-offs in top-down vs. modular design methods
- Understand where circuit delays come from
- Be able to analyze a combinational circuit to determine whether its outputs will suffer from logic noise (or "glitches")

This lab exercise requires:

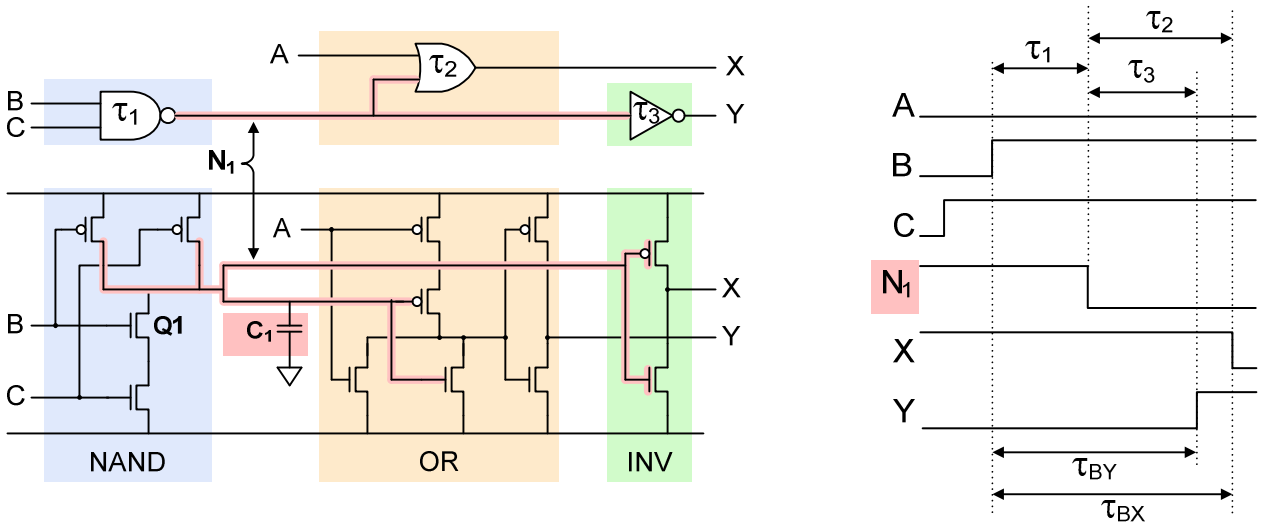
- A windows computer running the Xilinx ISE/WebPack tools

Propagation delays in logic circuits

Electronic signals travel along conductors at about 8cm per nanosecond (the actual speed depends on the conductor material, dimensions, and other external factors). Electronic switches, like the FETs used in logic circuits, typically require up to several hundred picoseconds to turn on and off. When a switch does turn on, it must transfer charge to or from the capacitance at its output node, and again, this takes time. All of these factors contribute to the simple fact that time is required for electric signals to propagate through logic circuits. Restated, time is required to process information in digital circuits. This processing time is divided between the less significant signal transmission time, and the more significant propagation delays associated with switching logic circuits. If not managed properly, propagation delays can result logic circuits that run too slowly to meet their requirements, or that fail altogether.

A simple logic circuit, its equivalent CMOS circuit, and a timing diagram are shown below with a particular intra-gate node (N1) highlighted. Note that if B changes from low to high when C is high as

shown, the circuit node N1 changes from high to low after a time τ_1 has elapsed. The time τ_1 is the "propagation delay" associated with the NAND gate. Referring to the CMOS circuit, the propagation delay τ_1 models transistor Q1 turning on and discharging node N1 from Vdd to GND. Although there is no actual capacitor at the output node, all the signal wires and FET connections associated with the circuit node N1 behave like a single capacitor, and this "parasitic" accumulated capacitance is shown lumped into a single component labeled C_1. As is the case with any capacitor, C_1 cannot transition from Vdd to GND immediately; the propagation delay τ_1 models the time required to discharge this capacitance.



As the C_1 capacitance discharges, the voltage at N_1 decreases below the input switching threshold of the inverter, the inverter drives its output Y to a '1' after the propagation delay τ_3. The propagation delay of the OR gate (τ_2) is longer than the delay for the inverter – in general, different gates will have different propagation delays. Further, since the delay through a given gate depends on the number of other gates and wires that it must drive, different instances of the same type of gate in a given circuit will have different propagation delays as well. In a given digital circuit, a designer is typically interested in the system response time rather than individual gate delays. For this circuit, the system response times T_{BX} and T_{BY} that show the time required for signals X and Y to change in response to a change on signal B are shown at the bottom of the timing diagram.

The amount of time required to drive an output from '0' to '1' (or vice-versa) depends on how much capacitance is present on the output node. In a CMOS circuit, the capacitance on a given output node is determined by how many "downstream" gate inputs are connected to the output node (for example, in the circuit above, node A is driving a single gate input, while node N1 is driving two gate inputs). As a first approximation, it is reasonable to assume a linear relationship between the number of downstream gates driven by an output node and the amount of time required to transition the output node. That is, if an output node connected to 2 downstream gate inputs can transition from '0' to '1' in time X, the same gate driving 4 downstream gate inputs can transition in time 2X.

Different circuit implementation technologies have different typical delays. For example, a circuit implemented in a modern FPGA will typically have delays that are much smaller than a circuit implemented in a five-year-old FPGA, and in turn, both FPGA circuits would have far smaller delays than a similar circuit built from discrete gates. The smallest delay times (on the order of 10's of picoseconds) are available in the most expensive technologies, and these are reserved for "fully custom" chip designs that sell in high-volumes (like Pentium processors), or for designs that require the best performance for specialized applications (like sensitive scientific instruments). Whatever the technology, circuit delays are affected by variations in the manufacturing process, so no two devices

from the same manufacturing line will exhibit exactly the same delay. Further, delays can change when circuits are exposed to different operating environments – both temperature and power supply voltage can greatly alter delays on various circuit nodes.

Circuit delays and CAD tools

When a design is "implemented" (i.e., translated and mapped to a given technology) in a CAD tool like Xilinx's ISE/Webpack, a separate database containing specific information about every component in the design is created. This database contains information that defines the input/output relationships for each component, including the time required for input signal changes to propagate through the component to cause output signal changes. Delay information is typically stored separately for rising-edge transitions (i.e., a 0-to-1 transition) and for falling-edge transitions. Different delay values are used for rising and falling edges to account for the differences in the FETs that are used to drive an output node to '0' or '1'. In a falling transition, nFETs are responsible for driving the output node to '0', while in a rising transition, pFETs are responsible for driving an output node to '1' (see the circuit example above). In CMOS circuits, nFETs can typically pass twice the amount of current as similarly sized pFETs, so driving an output node to '1' typically takes twice as long as driving an output to '0'. Some simpler CAD tools ignore this phenomenon, and use a single number to define "gate delay". This single gate delay number is applied to all inputs for both rising and falling transitions.

In general, the delays encountered in a given circuit cannot be precisely known until the circuit is transformed into its most basic structural representation. The most basic representation depends on the technology that will be used to implement the circuit. When circuits are synthesized to a given device like an FPGA or CPLD, all the "logical" components and interconnections specified in the source file are mapped to particular physical devices in the chip. Once this mapping happens, it is possible to calculate the delays for every circuit node in the design with a high degree of accuracy. Prior to this mapping, it is only possible to estimate the delays. Whether calculated or estimated, all useful logic simulators must accommodate delay values so that designers can simulate the behavior of physical circuits. In fact, it is fair to say that accurate delay modeling is the most important and most useful feature of a simulator. Designers have learned that they must know the effects of all delays on all circuit nodes prior to releasing a design to manufacturing.

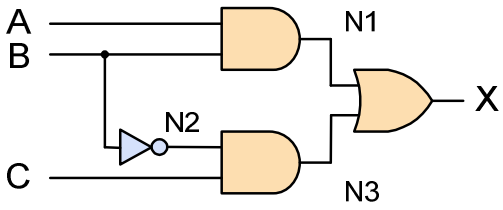
In a modern design flow, a circuit is initially designed without paying much attention to delays. In this early stage, a simulator is used only to check that the circuit logic has been correctly defined. When the design is synthesized to a given technology, the CAD tools can automatically calculate accurate delays for every single circuit node. Then, the circuit can be re-simulated, and the designer can study the circuit's behavior with accurate node delays included. Delay information is typically stored in a file called a "standard delay format", or .sdf file. In a post-synthesis simulation, the .sdf file is used by the simulator along with the circuit definition and the stimulus file to create a highly accurate output.

Many schematic-based CAD tools, and all VHDL tools, allow designers to include delays at the time a circuit is initially specified. These delays are by definition "best guesses", but they are nevertheless useful in studying a given circuit's performance. These delay values can easily be modified to simulate a circuit's behavior under different operating conditions that might arise. For example, best-case or worst-case delays could be used to model circuit performance at different operating temperatures or supply voltages.

In problems and exercises up to this point, the focus has been on creating functionally correct circuits, and the effects of gate delays have been ignored. Going forward, you will come to appreciate that creating a "functionally correct" circuit is the simplest part of solving a given problem. The greater challenge often lies in creating a circuit that will always work in a given physical environment, with all the attendant gate delay and timing issues, and in validating circuit performance through testing.

Circuit delays specified in VHDL source files

The time-behavior of any VHDL assignment statement can be specified using the keyword "after" and a time definition as shown in the example code. In the simplest case, a single delay value is provided to define the time between any input change and a resulting output change. For example, in the second code example for the circuit on the right, Y won't assume a new value until 3ns after A, B, or C changes.



The example above treats the entire circuit as one entity, and assigns a single delay value to the entire circuit. Although this is a simple way to assign delays, a great deal of potentially useful information is hidden. In general, if you are attempting to model delays in a circuit, it is better to assign delays to each logic gate, including those that drive intermediate nodes between the inputs and outputs. Then, more detailed simulations can demonstrate whether circuit delays resulting from individual gates are likely to cause problems.

```
architecture simple of example is begin
   Y <= (A and B) or (not B and C);
end simple;
```

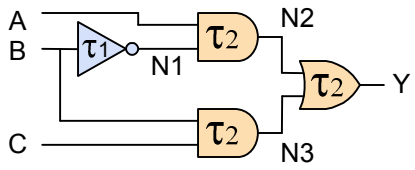
```
architecture simple of example is begin
   Y <= (A and B) or (not B and C) after 3ns;
end simple;
```

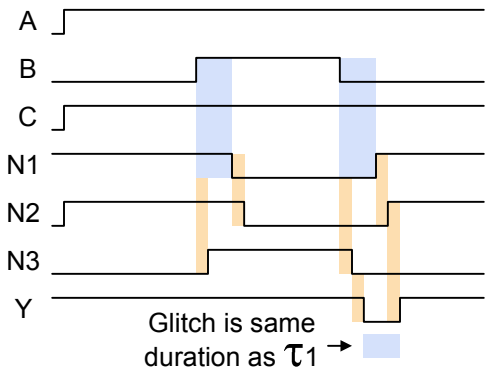
The third architecture example provides a more detailed description by assigning delay values to each individual circuit node. When more complex assignment statements are broken into their constituent parts like this, more detailed (and therefore more useful) delay values can be assigned. Further, when this VHDL code is simulated, every signal node can be examined in the waveform viewer.

```
architecture gates of example is
  signal N1, N2, N3 : std_logic;
begin
  N1 <= (A and B) after 2ns;
  N2 <= not B after 1ns;
  N3 <= (N2 and C) after 2ns;
  X  <= (N1 or N3) after 3ns;
end gates;
```

Glitches

Propagation delays not only limit the speed at which a circuit can operate, they can also cause unexpected and unwanted transitions in outputs. These unwanted transitions, called "glitches", result when an input signal changes state, provided the signal takes two or more paths through a circuit and one path has a longer delay than the other. The increased delay on one path can cause a glitch when the signal paths are recombined at an output gate. Asymmetric path delays commonly arise when an input signal drives an output through two or more paths, with one path containing an inverter and one not. The figures below illustrate a glitch being formed by an inverter. Note the glitch (the 1-0-1 transition on Y) has the same duration as the delay in the inverter.





All logic gates add some delay to logic signals, with the amount of delay determined by their construction and output loading. In the figure to the right, the inverter is shown with a larger delay (identified by time T1) than the other gates (T2). This contrived example uses an over-long inverter delay to clearly show its role in creating an output glitch, but a glitch would appear no matter what the delay time. By carefully studying the timing diagram, it is clear how the inverter delay is related to the output glitch.

Glitches occur when an input is used in two product terms (or two sum terms for a POS equation), and inverted in one term but not in the other. This is illustrated in the figure, the logic equation, and in the K-map to the right. In the K-map, two loops define a minimal logic expression. The B·C term is independent of A; that is, if B and C are both '1', the output will be a '1' regardless of changes on A. Likewise, the term A·B' is independent of C, so that if A and B are '1' and '0', the output is a '1' regardless of how C might change. But note if A is a '1' and C is a '1', the output should always be a '1' regardless of B, but no single term is driving the output independent of B. This is the situation that gives rise to the problem: two different product terms keep the output at a '1' when A and C are both '1' – one when B is a '1' (B·C), and one when B is a '0' (A·B'). Thus, as product terms must recombine at the output to keep the output high, ar glitch.

A \ B C	00	01	11	10
0	0	0	1	0
1	1	1	1	0

Y = AB' + BC

B is a "coupled" variable

A circuit that can glitch can be identified by its schematic, K-map, or logic equation. In a schematic, an input that follows multiple paths to an output gate can create a glitch, if one path has an inverter and one does not. In a K-map, if loops are adjacent but not joined by an "overlapping" loop, then the adjacency not covered by a loop presents the opportunity for a glitch. In example in the K-maps below, only K-map #1 results in a circuit that can glitch.

A B \ C D	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	0	0	1	1
10	0	0	0	0

K-map #1: Possible Glitch

A B \ C D	00	01	11	10
00	0	1	0	0
01	1	1	1	1
11	0	0	1	0
10	0	0	0	0

K-map #2: No Glitch

A B \ C D	00	01	11	10
00	1	0	0	0
01	1	0	1	0
11	0	0	1	0
10	0	0	0	0

K-map #3: No Glitch

A glitch can be identified in a logic equation if two or more terms include the same logic signal, and the signal is inverted in one term but not in another. For this discussion, each pair of terms that contain a single variable that is inverted in one term but not the other are called "coupled terms", the inverted/non-inverted variable the "coupled variable", and the set of all other variables in both terms the "residue". The examples below illustrate.

X = (A + B') (A + C) (B' + C)

No glitch possible
No coupled terms

X = A·B' + A'·C

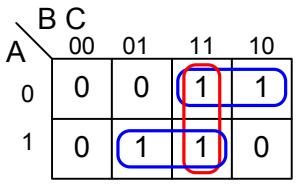
X can glitch when A changes
Both terms are coupled
A is the coupled variable
B' and C are the residue

X = A'·C' + A·B + B'·C

X can glitch when A, B or C changes
A'·C' & A·B are coupled terms
A·B & B'·C are coupled terms
A'·C' & B'·C are coupled terms

In some applications, it may be desirable to remove the glitch so that the output remains steady when a coupled variable changes state. Note that in the solution to Problem 1, the glitch on Y is only possible if B and C are held high. This observation can be generalized: for a glitch to occur, a logic circuit must be "sensitized" to a coupled variable by driving all inputs to appropriate levels so that only the coupled variable can affect the output. In an SOP circuit, this means that all inputs other than the coupled input must be driven to a '1' so that they have no effect on the outputs of the first-level AND gates.

This observation leads directly to the method for removing a glitch from a logic circuit: combine all residue input signals in a new first-level logic gate (i.e., an AND gate for an SOP circuit), and add the new gate to the circuit. For example, in the equation X = A'·B + A·C, the coupled term is A, the residue signals would combine to form the term B·C, and that term would be added to the circuit to form X = A'·B + A·C + B·C. This is shown in the K-map – note that the original equation is minimal (blue loops), and that the glitch-free equation adds a redundant term (red loop).



This is always the case – removing glitches requires a larger circuit with redundant logic. In practice, it is almost always preferable to design minimal circuits and deal with glitches in another manner (discussed in a later module). Perhaps the best lesson is to be aware that in general, whenever an input to a combinational circuit changes, glitches are possible (at least, until proven otherwise).

The loops for the original SOP equation in problem 1 did not overlap, and this is the hallmark of a potential glitch. When a loop for the redundant term is added, every loop overlaps with at least one other, and no glitches are possible.

If non-overlapping (or isolated) loops are located in non-adjacent K-map cells (see K-map #3 above), there are no coupled terms, no coupled variables, and it is not possible to add a loop (or loops) to cause all loops to overlap with at least one other. In such a case, no single input change can cause a glitch. In this type of circuit, two or more inputs might be directed to change state "at the same time", with the desired outcome of having the output remain at some stable state. For example, in the circuit Y = A'·C'·D' + B·C·D from K-map #3 above, it might be desired that all inputs go from '0' to '1' simultaneously, and that in response, the output stay constantly at '1'. In practice, it is impossible to change all inputs simultaneously (at least, on a scale of picoseconds), and as a result, the output will show a glitch-like transition equal in duration to the time difference between input signal changes. Such unwanted transitions cannot be eliminated by adding redundant gates; rather, they must be dealt with by redefining the circuit or with sampling and pipelining (these topics are discussed in a later module). We will not deal further with unwanted output transitions resulting from multiple input changes here.

Most of the glitch discussion so far has focused on SOP circuits, but the same phenomenon is present in POS circuits as well. POS circuits suffer from glitches for the same reasons as SOP circuits (asymmetric path delays for an input arriving at multiple input gates). As you might expect, the conditions required are similar, but not identical to the SOP case.

These simple experiments demonstrate the basic effects of gate delays on digital circuits – namely, output glitches are possible in response to input transitions, provided the input passes through asymmetric circuit path delays in forming the output. In the more general case, any time an input passes through two different circuit branches, and those two branches are recombined at a "downstream" point in the circuit, timing problems like glitches are possible. Again, the lesson is to be aware that signals take time to propagate through logic circuits, and different circuit paths have different delays. And in certain cases, those differential delays can cause problems.

Using CAD-tool Generated Delays

The Xilinx ISE simulator can model delays for circuits that may be programmed into a Xilinx device. The simulator contains a “post-route simulation” feature that automatically generates and includes accurate delays for all circuit nodes. The delays are calculated after the circuit has been mapped to the physical devices in the target chip, and so they are quite accurate. Any source file can be simulated with delays by first “implementing” the design in the Xilinx project navigator, and then by running the simulator in the post-route mode (or, simply run the simulator in the post-route mode, and if the circuit has not yet been implemented, it will be implemented automatically). A brief appendix in the lab project document shows how to run the simulator in post-route mode.