Multi-Clock Design: Interfacing with a PS/2 Keyboard

As you've researched the way in which a PS/2 keyboard communicates, you've likely encountered a description of the two signals that your keyboard generates:

- A single PS/2 **data signal**, which the keyboard uses to send *serial* data, and
- A single PS/2 ***pseudo*-clock signal**, with which the keyboard notifies your Basys board that new data is ready.

In order to interface with a PS/2 device, we'll need to understand how each of these two signals works- and how we'll be able to use them in our design!

The PS/2 *Psuedo*-Clock

The *pseudo*-clock signal provided by your PS/2 keyboard works like a much slower version of the clock on your Basys board: it's designed to toggle at a pre-defined interval- and it can be used to keep time within your circuit. The general shape of the provided *pseudo*-clock should be very familiar to you: it looks very much akin to our typical system clock:

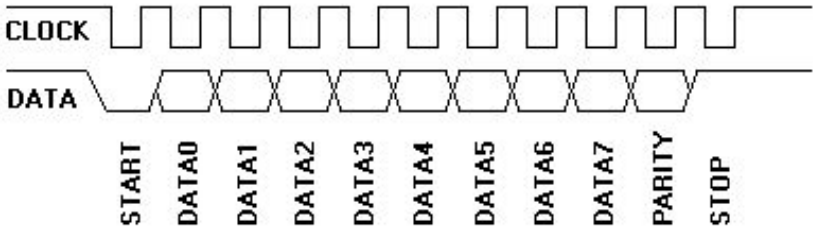


image credit: http://bit.ly/VDxqyU, CC ShareAlike

Unfortunately, the PS/2 *pseudo*-clock has one significant downside: it's only transmitted when at least one keyboard key is being pressed! Since the keyboard clock is only sent sometimes, we don't consider it a true clock- hence the name *pseudo*-clock. Many other guides found around the internet don't make this distinction, so you'll often see the name given as simple the *PS/2 clock*.

Despite this limitation, the keyboard clock is far from useless. If we were only designing a *keyboard interface*, this clock signal might be the only clock we need. If you *only* care about recognizing various keypresses- and don't need to keep track of anything that's happening *between* keypresses, you could use the keyboard's clock as your system clock.

Common Clock and the Basys Board's Clock

If we're trying to design something that contains more than just a basic keyboard interface, we'll almost definitely need our circuit to operate between keypresses. Since our *synchronous circuits* require a clock signal to work, we'll need a clock that operates even when the user isn't touching the keyboard- a *true* clock, like the one on our Basys boards.

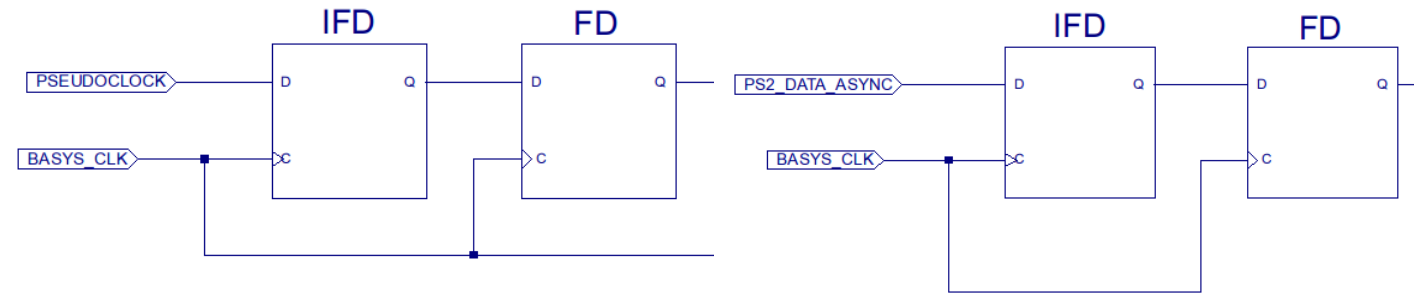
Faced with this issue, you may be tempted to hook *some* portions of your circuit to the keyboard's *pseudo*-clock, and others to the Basys board's clock- but this violates the *common clock assumption*. Recall that we've designed our synchronous circuits so every register changes value at the same time- a model that greatly simplifies the task of ensuring proper timing throughtout the circuit.

Instead, we'll have to choose *one* and *only one* signal to act as our system clock. Since we've already determined that the keyboard's *pseudo*-clock isn't suitable, we'll need to power each one of our circuits off of the Basys board's clock.

PS/2 Signal Conditioning

Since we've decided to use the *true* clock on the Basys board as our system clock, we'll need to take special care when handling the keyboard inputs. The keyboard doesn't know anything about our system clock- so it can't ensure that its values change on the *rising edges* of our system clock. In other words, the signals we receive from the keyboard aren't in sync with our system clock- they're *asynchronous*.

Like all other *asynchronous* inputs, we have to take special care to make sure that these inputs don't violate the *setup* or *hold* times of our internal synchronous components. Luckily, there's nothing all that special about the two PS/2 signals- they're nice and slow, compared to the clock on our Basys board's clock- so we can use a standard synchronizer to make those inputs *synchronous*:



Notice the two *Xilinx* schematic components we've chosen for our synchronizer above:

- one `IFD`- an *input* D-FF, and
- one `FD`- a normal D-FF.

The `IFD` components are a special variant of D-FF designed to have very short *setup* and *hold times*- and which are slightly better at recovering from any violations that occur. These *input* flip-flops are ideal for use in synchronizers- but, due to limitations inside of the FPGA, they can only be used for signals which are directly connected to the FPGA's inputs.

PS/2 Data Availability

Once we've synchronized our two PS/2 signals, we're almost ready to start processing PS/2 data! Now that we have a *synchronous* version of our PS/2 data input, we can safely[1] read from that data line at any time. There's just one minor hitch- if we're not using the *pseudo*-clock as our system clock, how will we know when each bit of the PS/2 signal is ready?

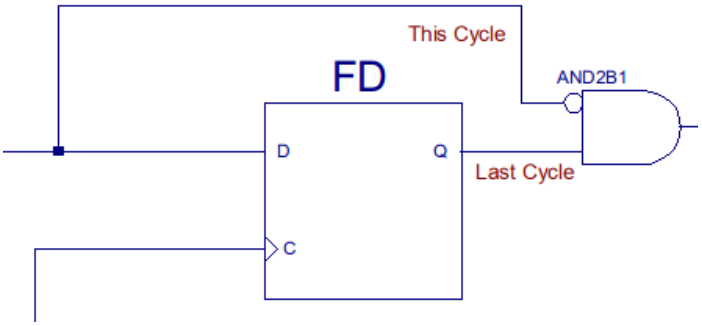
Fortunately, we can still look at the value of the PS/2 *pseudo*-clock. The PS/2 protocol specification tells us that each new bit will be available on our PS/2 data input "at the falling edge of our *pseudo*-clock". If we avoid thinking of the PS/2 *pseudo*-clock as a clock, then we might re-phrase that statement as follows: we know that a new bit is available on the PS/2 data input when the PS/2 *pseudo*-clock changes from `'1'` to `'0'`.

1 Well, as safely as we'll ever be able to. There's always a *very* tiny probability that a violation occurs which our synchronizer isn't able to correct- but that probability is so small that we won't worry about it in this class.

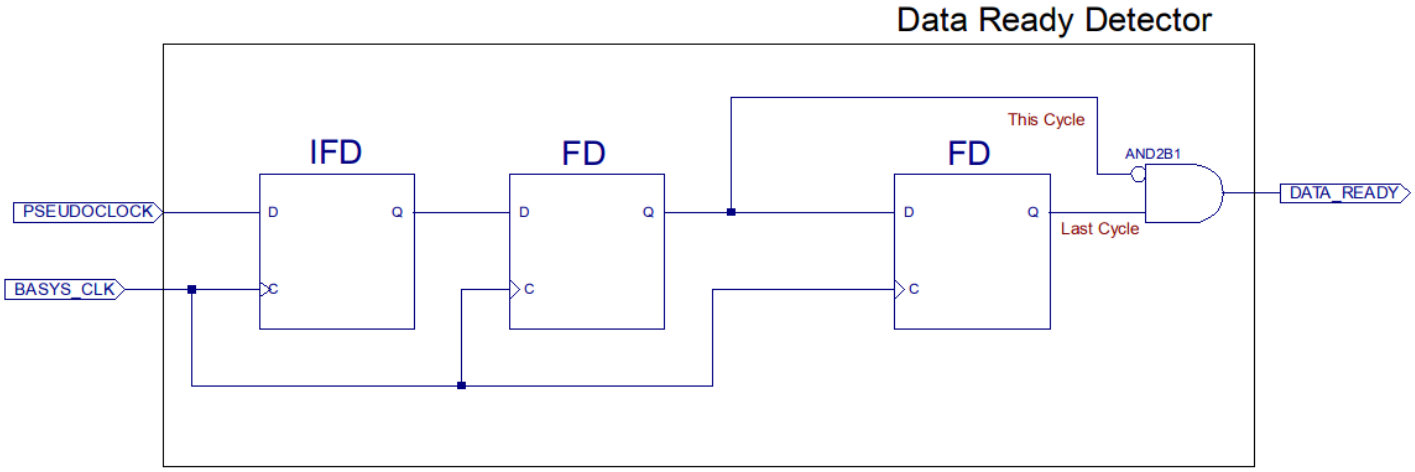
To simplify the process of reading in data from the PS/2 port, we can design a circuit that detects when a new data bit is available. We're looking to design a circuit that outputs '1' when the following conditions are met:

1. The value of the *pseudo*-clock is currently `'0'`; and
2. The value on the *pseudo*-clock was `'1'` a short time ago- say, during the previous clock cycle.

Since we know that a *flip-flop* is capable of delaying a signal by a single cycle, it's pretty easy to develop a circuit which detects the conditions above:



Combined with our synchronizer from the previous section, we have a *data ready* circuit that looks like this:



The PS/2 keyboard sends a clock-like signal, which indicates when a bit is ready on the PS/2 data line. This 'ready detector' converts that signal into a single-cycle pulse, which indicates when you can take in each bit of the pattern.

Using the Data Ready Signal

We now have everything we need to start processing the PS/2 data! Our *data ready* signal will be particularly useful as we start working with the Xilinx tools. We can use that signal in a variety of ways: we could very easily control the *operation* of a multi-function register, or the operation of an ALU- we could even use that signal as an input to a Finite State Machine!

You'll very likely want to explore the various components which come packaged with the *Xilinx* tools. Several of the built-in multi-function registers feature a single-bit *opeation enable* called `CE`- which is perfect for interfacing with a signal like *data ready*!