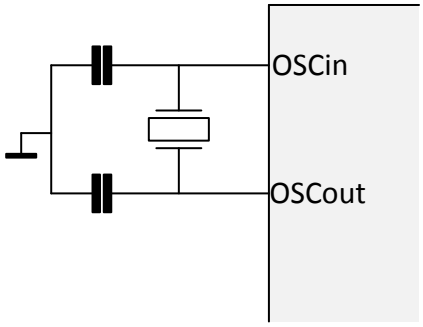


Crystal start-up

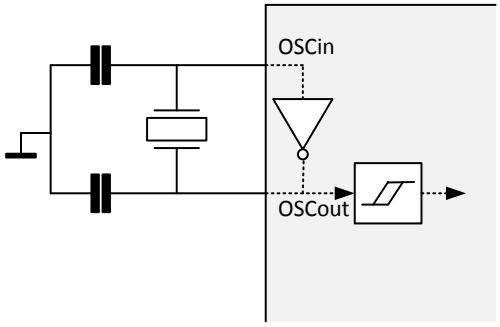
On the world-wide-web you can find a plethora of webpages with articles, code, instructions and examples which tell you how to write Verilog code. All of these have one thing in common: they assume you have a clean clock and a clean reset. However in real life the clean clock and clean reset do not appear out of nowhere. They have to be generated. This module implements a start-up delay for crystal oscillators.

Introduction.

If ever you have seen or worked with electronic schematics you may have encountered the standard circuit for connecting a crystal to a (micro) processor:

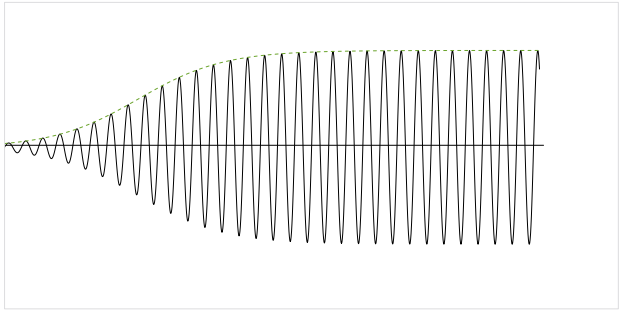


Inside the controller is an inverting amplifier which makes that the circuit start oscillating. This is a slightly more accurate picture:



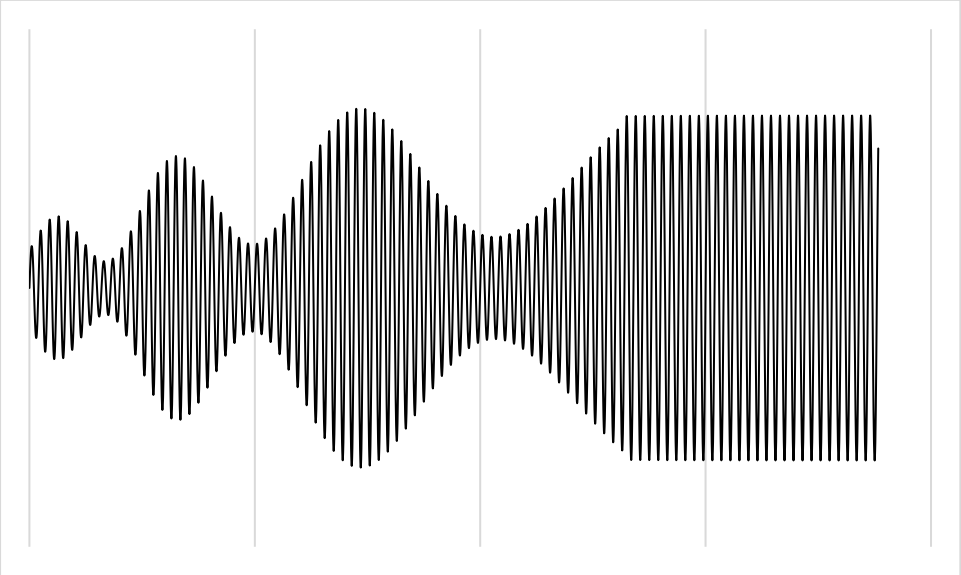
The oscillation is not a nice square wave but a sine signal. In order to use this signal the oscillator output is passed through a Schmitt trigger.

The start-up of the oscillation is depending on electric noise. As a consequence of this, the crystal start-up time is non-deterministic. It depends on the circuit in which the device is placed, the type and amount of decoupling and very much on the way the power comes up[1]. Therefore when power is applied to a crystal oscillator the time before the crystal is producing a regular clock is unknown. Manufactures can specify a start-up time from 1 or 14 milli seconds.
Below is a most common start-up waveform.



[1]Therefore you find a specification for the power ramp up time.

However it is also possible that the start-up oscillations are bit more complex:



Both of the above waveforms can produce pulses which are much shorter than the crystal period. This can (actually: *will*) upset any downstream logic.

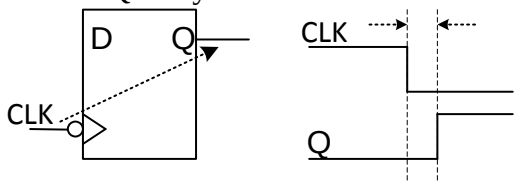
The module **`clock_switch.v`** takes care of all the difficulties of that process. The circuit is designed to cope with irregular clock pulses. It waits for a time and will pass the input clock on to the output after a number of valid clock pulses have been seen.
The crystal is assume to be stable after that delay.

Description

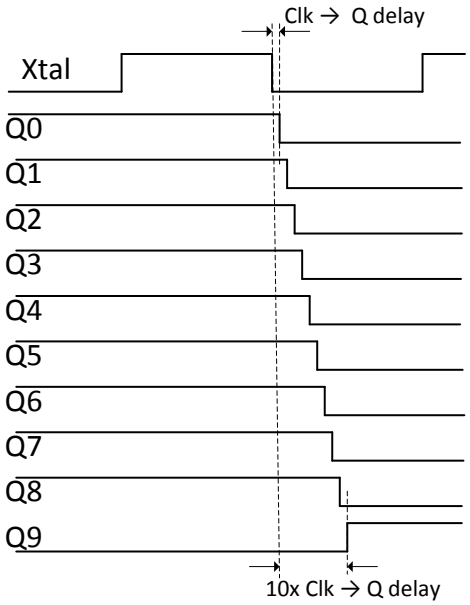
The circuit is deceptively simple and upon seeing it often creates a 'but it is obvious how you do it' reaction. (Look up the term 'hindsight bias'.) The chain of registers are a-synchronous clock dividers. This is what makes the circuit immune to any erratic clock coming in. The earlier stages may or may not toggle but this does not impede the correct function of circuit. The stages at the end will filter out any erratic behaviour.

Hidden danger

The circuit has a hidden danger which can make that the whole circuit is unreliable if no attention is paid to it. As the chain of divide-by-two registers are a-synchronous, the signal incurs a delay at each stage. For the register used in the **`clock_switch.v`** circuit this is the clock-to-Q delay:



As a single delay this is not too bad. However in a long chain of dividers this delay can add up:



The last stage of the delay chain is used to switch off the input to the dividers and to pass the crystal clock to the output. This produces a clean clock *then and only then* when the delay chain is shorter than the low period of the incoming crystal clock.

Basically the supplied circuit will go wrong if:

- The delay chain is very long.
- The Clock-to-Q delay is great.
- The Crystal frequency is high.
- A combination of the above.

Most crystals oscillators work with crystal frequencies are around 20 MHz. This gives a low period of 25 nS. In modern logic the clock-to-Q delay is much better than 1nS. Thus the circuit is likely to be safe for delays chains op to 20 stages (Divide by 1 million). This is NOT the case for the **`clock_switch3.v`** circuit which has a longer intrinsic delay per stage.

Implementation details

There are a number of subtleties in the design which are not immediate obvious.

The basic register element is ***not*** generated from an always @(...) statement. Instead the register is instanced and is expected to come from a logic library. This is for two reasons:

1. A separate module allows the use of a behavioural model which has the clock-to-q delay in the model.
2. More important: in a real circuit it is best that ***you chose your own library cell***[2]. The behaviour and timing of the register is essential for the good function of the circuit. In such cases I advise you to select the gate from the library yourself to make sure it behaves exactly as wanted and not be dependent on the choice of a synthesis tool.

`clock_switch.v`

The **`clock_switch.v`** version is the simplest version. It uses a negative clock edge triggered register with an active low reset. This type of register is present in most libraries. If the no such register exists in the library try to use **`clock_switch2.v.`**

[2] This is a good policy when designing 'abnormal' circuits.

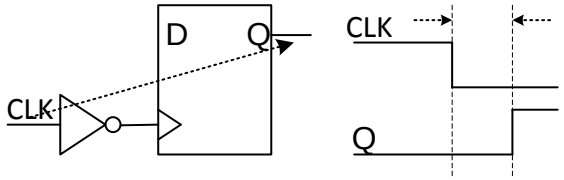
clock_switch2.v

The **clock_switch2.v** is a second version of that circuit which uses a slightly different register. It uses a positive clock edge triggered register with an active low ***set***. If the no such register exists in the library try to use **clock_switch3.v.**

Because the circuit is controlled by a rising clock the clock signal to the chain is stopped by an OR gate (In **clock_switch.v** an AND gate is used). It is details like these which are easily overlooked and can give rise to a circuit which functions *most* of the time.

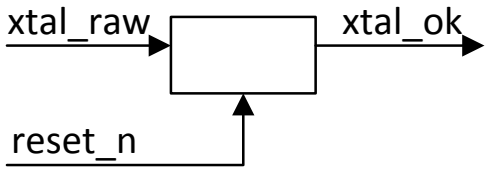
clock_switch3.v

The **clock_switch3.v** is a third version. This version uses a positive clock edge trigger register with an active low reset. This is the most common type of register which is available in any library. However the circuit can NOT be built from this type of register alone. In order for it to work an inverter has to be used in each stage of the divide-by-two chain.

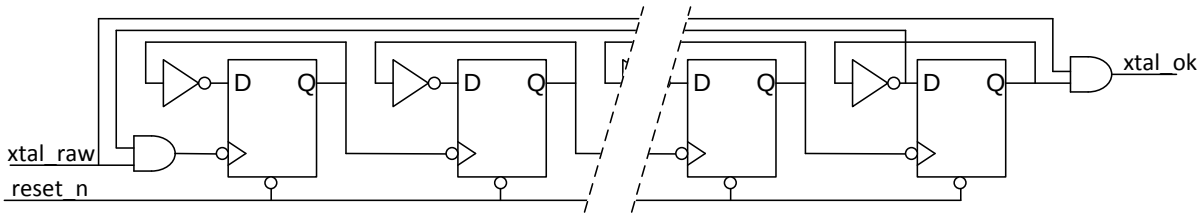


This will increase the delay through the divider chain. As such the user should be even more aware of the timing behaviour.

Diagrams



Principle diagram



clock_switch.v possible implementation

Description

The module has the following ports:

```
input  xtal_raw,     // signal from crystal oscillator pad
input  reset_n_raw,  // signal direct from reset-pad
output xtal_ok       // Clock from crystal which is assumed
                     // to have stabilised    output
```

xtal_raw

The signal from the crystal oscillator. The signal is assumed to be digital (no longer the sine wave as seen on the crystal pins). The signal may show erratic behaviour at the beginning for

a certain period of time. That is, it may have pulses which are low or high for an extremely short period of time (spikes, glitches).

`reset_n`
The raw reset from the reset pin, optionally cleaned up to have no spikes or glitches. This is NOT the system reset which is often generated by the clock from the crystal. See other circuits for that on the web site.

`xtal_ok`
This is the crystal clock which is output after a period of time. The crystal oscillator is assumed to be stable by then. It is low after a reset and thus the first edge out will be a rising edge.

Copyright