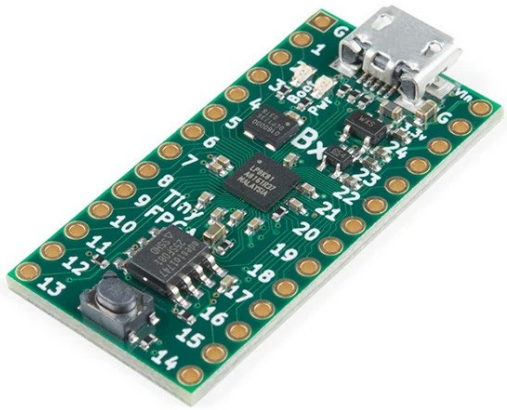
Introduction to TinyFPGA BX
and RTL synthesis in Verilog

v1.05



The term FPGA[1] (Field Programmable Gate Array) groups a class of devices that are used to build complex digital circuits. Although programmable, FPGAs are very different from microcontrollers: the source code does not define a program to execute but a logic circuit to implement. Each logical element of the project “consumes” a physical part of the FPGA, which is functionally configured to make that particular element. Furthermore, while in a microcontroller the instructions are executed one at a time, in an FPGA all the defined elements are active at the same time, so FPGAs are particularly useful to realize fast and highly parallel operations. Any digital system can be built with FPGAs, including microcontrollers with custom peripherals.

Most FPGAs are manufactured by Altera and Xilinx (which together control about 80% of the market). There are also some smaller companies like Lattice Semiconductor, which produces (among others) the iCE40 and ECP5 families. These FPGAs are very popular among developers and hobbyists because they can be programmed using completely free applications. The FPGA iCE40 in particular is the basis of the TinyFPGA BX board, which has a low cost, and is therefore perfect to take the first steps in the world of programmable logic: just put it on a breadboard and interface it with LEDs, switches, encoders and digital sensors.

The TinyFPGA BX provides developers with the power and flexibility of a FPGA with 7680 cells in a small form factor, perfect for breadboards and homemade PCBs. There are currently no other small low-priced FPGA boards with as many logic resources as this one. The number of available cells is enough to implement a small 32-bit RISC-V CPU! (PicoRV32)[2].

The board can be programmed using a digital design language (Verilog/VHDL[3], Migen[4], Chisel[5]) or a schematic input tool like Icestudio[6], which allows you to graphically design the digital circuit.

1 https://en.wikipedia.org/wiki/Field-programmable_gate_array
2 https://github.com/mattvenn/TinyFPGA-BX/tree/master/examples/picosoc
3 http://www.fpga4fun.com/HDLtutorials.html
4 https://m-labs.hk/migen/manual/introduction.html
5 https://chisel.eecs.berkeley.edu/
6 https://github.com/FPGAwars/icestudio

Once the desired design is obtained, it is transferred to the TinyFPGA BX module through a USB connection.

These are the specifications of the TinyFPGA BX board:

- Small size (height 1.4 inches, width 0.7 inches)
- High quality PCB (4 layers with dedicated power planes)
- Programming via full-speed USB 2.0 interface (12 Mbit/s)
- Lattice FPGA iCE40[7] LP8K-CM81 (low power 8K gate - package “81 ucBGA”):
  - 7,680 logical cells (with four-input association tables)
  - 128 Kbit RAM (16KB)
  - 1 PLL circuit (note that the iCE40 LP8K provides two PLLs but only one is available in the tiny 4x4mm “81 ucBGA” package used on the TinyFPGA BX board)
  - 41 user I/O (the iCE40 LP8K provides 178 user I/O, but the “81 ucBGA” package makes only 63 available, of which only 41 are connected to the TinyFPGA BX pins)
- 8 Mbit SPI Flash (for storing the bootloader, the logic circuit and the user data)
- On-board LDO regulators for 3.3V (300mA) and 1.2V (150mA)
- Low-power 16 MHz MEMS oscillator:
  - Only 1.3 mA when active
  - Stability 50 ppm
- Open-source USB bootloader[8]

NOTE: the 16MHz clock of the MEMS oscillator cannot be modified, but we can generate a Verilog logic module to scale up this value with the PLL (within some limits)[9] and also scale it down using dividers.

When powered on, the TinyFPGA BX loads the USB bootloader from the SPI flash, appearing on the host computer as a device with a virtual serial port. The programming software automatically detects this device and uses the serial interface to program the logic circuit on the board. Once the design has been loaded into the SPI flash, the board reboots and directly loads the implemented logic circuit. To update the design simply press the reset button and the bootloader will be activated again. The bootloader also includes metadata stored in the SPI flash, in open JSON format. These metadata contains:

- A unique ID for each card: no matter what serial port the operating system will assigns to the card, you are always sure to program the right one.
- A human-readable name assigned to the board (and FPGA).
- Information on where to store the user program and any optional data.
- An editable URL, for retrieving bootloader and firmware updates.

For technical details and source code see the GitHub repository of the TinyFPGA BX[10].

7 https://en.wikipedia.org/wiki/ICE_%28FPGA%29
8 https://github.com/tinyfpga/TinyFPGA-Bootloader
9 https://discourse.tinyfpga.com/t/i-i-am-using-tinyfpga-bx-for-the-first-time-i-have-questions/1080/2
10 https://github.com/tinyfpga/TinyFPGA-BX

1 Updating the bootloader

First you need to install the "*tinyprog*" program, check that the board is recognized by the PC and eventually update the bootloader to the latest version. The instructions below are for Arch Linux and Manjaro; for other Linux distributions just install the equivalent packages, while for Windows and MacOS see the reference guide[11]:

- Verify that the user belongs to the "*uucp*" group (if not, add the user to this group and login again). Note that on some Linux distributions you may need to belong to the "*dialup"* group instead of "*uucp*".

- Install "*tinyprog-1.0.23*" (or later) from the official repositories of your Linux distribution.

- Connect the TinyFPGA BX to the PC through a suitable USB cable (beware that the tiny USB connector is quite delicate). It is recommended, if possible, to insert the board onto a breadboard (after soldering the headers) to ensure greater stability: the board alone is very light and the rigidity of the cable could easily move it, bringing it in contact with any nearby metal parts, damaging both the TinyFPGA and the computer to which it is connected.

- In the virtual terminal, type "*dmesg*" (or "*sudo dmesg*" if necessary). The last lines should display the following messages (with some variations depending on the Linux kernel used):

  […] usb 2-1: new full-speed USB device number 2 using xhci_hcd
  […] usb 2-1: New USB device found, idVendor=1d50, idProduct=6130, bcdDevice= 0.00
  […] usb 2-1: New USB device strings: Mfr=0, Product=0, SerialNumber=0
  […] cdc_acm 2-1:1.0: ttyACM0: USB ACM device
  […] usbcore: registered new interface driver cdc_acm
  […] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters

- Type "*lsusb*" and look for the line containing the values 1d50:6130 (*idVendor:idProduct*, see above). The descriptive string reported should be "*OpenMoko, Inc*".

- Type "*tinyprog -l*". The following lines should be displayed:

  TinyProg CLI
  Using device id 1d50:6130
  Only one board with active bootloader, using it.
  Boards with active bootloaders:

    /dev/ttyACM0: TinyFPGA BX 1.0.0
    UUID: 626e3f5d-09a8-468f-a17c-6313e22488f9
    FPGA: ice40lp8k-cm81

  Note that the UUID is specific to each board, while the 1.0.0 value shown specify the revision of the board, not the bootloader. To check the bootloader version you need to print the metadata stored in the SPI flash, using the "*tinyprog -m*" command:

11 https://tinyfpga.com/bx/guide.html

```
[
 {
  "boardmeta": {
   "name": "TinyFPGA BX",
   "fpga": "ice40lp8k-cm81",
   "hver": "1.0.0",
   "uuid": "626e3f5d-09a8-468f-a17c-6313e22488f9"
  },
  "bootmeta": {
   "bootloader": "TinyFPGA USB Bootloader",
   "bver": "1.0.1",
   "update": "https://tinyfpga.com/update/tinyfpga-bx",
   "addrmap": {
    "bootloader": "0x000a0-0x28000",
    "userimage": "0x28000-0x50000",
    "userdata": "0x50000-0x100000"
   }
  },
  "port": "/dev/ttyACM0"
 }
]
```


- If the term "*bver*" in the "*bootmeta*" section does not appear (or if it is less than 1.0.1) you need to update the bootloader (an internet connection is required). Type the command "*tinyprog --update-bootloader*" (note the double dash). If the bootloader is already updated, the string "All connected and active boards are up to date!" will be printed as last line, otherwise the bootloader will be downloaded and updated[12].

- Note that if you have more than one TinyFPGA BX connected to your PC, the tinyprog program will report data from all boards and will perform an update of all bootloaders that need it. To force the addressing of a specific board, just specify its UUID with the "*--id*" option (the first 2 or 4 characters are enough if they can distinguish one board from another).

- If the bootloader is corrupted (resulting in a bricked board) you can restore it using the SPI interface on the bottom side of the TinyFPGA BX[13].

12 https://github.com/tinyfpga/TinyFPGA-Bootloader/tree/master/programmer
13 https://discourse.tinyfpga.com/t/bx-bootloader-bricked/852/2

2 TinyFPGA BX Quick Start

There are several solutions for programming the iCE40 FPGA on the TinyFPGA BX:

- Lattice's proprietary program *iCEcube2*: available only in 32-bit version for Windows and Linux, it allows programming the FPGA in Verilog and VHDL; the latest release is from December 2020, so the installation on new 64-bit Linux distributions could be complicated due to old 32-bit dependencies required; moreover it requires a (free) license from Lattice Semiconductor, which is tied to the MAC address of the network card and must be periodically renewed (it expires after some time);

- the open-source Apio environment: currently it allows programming only in Verilog; the 0.8 series has several issues so I recommend using an older version;

- my makefile (available in https://gitlab.com/flaviocappelli/fpga-code), together with the necessary open-source simulation and synthesis tools (see the repository); this makefile also provides post-synthesis and C++ simulations (not available in Apio);

- the *Icestudio* schematic editor (which is still based on Apio); on Linux it is provided as AppImage package;

For simplicity, in this paper we will consider Apio. It uses the IceStorm and PrjTrellis projects, which aspire to document the bitstream format of all iCE40 and ECP5 FPGAs, while providing a fully open-source environment for their programming.

NOTE: Old examples on Internet describe using Apio with the Atom editor (via the Apio-IDE extension). But the development of Apio-IDE has stopped long ago and Apio > 0.4.0b5 is no longer compatible with this extension. We will therefore use Apio standalone, running its commands from the terminal. To load a simple example in the TinyFPGA BX proceed with the following steps:

- Install Apio, in particular a version between 0.5.4 and 0.7.6: if such a version is not available for the system is use (or not working), use a VM (virtual machine) with an environment capable of supporting these versions.

- Type "*apio install --all*" (note the double dash) to download all the necessary dependencies (which will be installed in the "*.apio*" folder, created in the user's home directory); note that an active Internet connection is required.

- Go to the folder "*~/.apio/packages/examples/TinyFPGA-BX/blinkSOS*".

- Connect the TinyFPGA BX to the system (and the VM if necessary) and type "*apio upload*".

The example will be compiled (generating the "hardware.asc", "hardware.json" and "hardware.bin" files), then loaded into the board. If successful, the word "SUCCESS" will appear green in the terminal and the LED on the board will start blinking, emitting a simple SOS sequence.

Next we'll look at the main commands of the Apio environment and will learn how to create, synthesize, and simulate a simple project in Verilog. For the moment, it is useful to observe the structure of the "*.apio/package/*" folder[14], which in the case of Apio < 0.8.0 is the following:

14 https://github.com/FPGAwars/apio

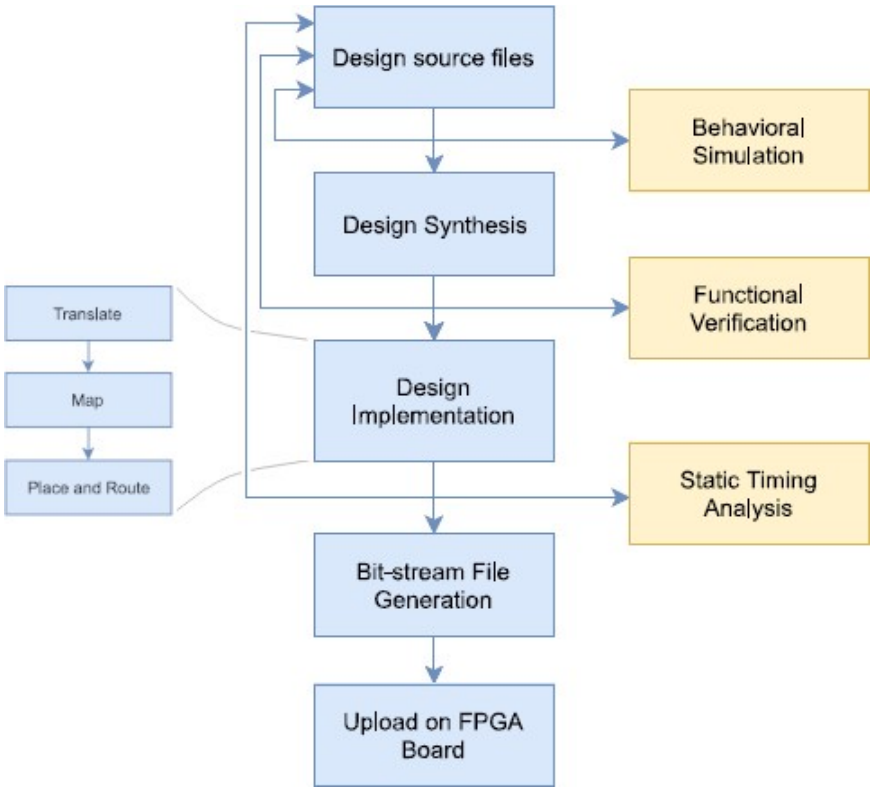
- examples - examples and templates for several boards, including the TinyFPGA BX
- toolchain-ecp5 - programming tool for Lattice ECP5 FPGA (from the Trellis project[15])
- toolchain-ice40 - programming tool for Lattice iCE40 FPGA (from the IceStorm project[16])
- toolchain-iverilog - tool for synthesis and simulation in Verilog (from the Icarus project[17])
- toolchain-verilator - C++ simulator for Verilog (from the Verilator project[18])
- toolchain-yosys - FPGA synthesis tool (from the Yosys projects[19])
- tool-... - other irrelevant subdirectories

Inside each one of these subdirectories you'll find the "*bin*" folder, that contains the executables of the individual packages. Normally Apio calls the required executables based on the workflow, but these can also be invoked directly if needed (e.g. "*.apio/package/toolchain-ice40/bin/icepll*" is very useful for generating the Verilog module to scale up the 16MHz clock). Note that in Windows Apio also creates the following two folders:

- drivers - FPGA Windows driver (required for programming)
- gtkwave - Verilog simulation viewer (from GTKWave1 project[20])

On Linux and MacOS, no drivers are required and the GTKWave application must be manually installed using the package management tools of the operating system in use.

The synthesis of a logic system on FPGA follows the following flow:

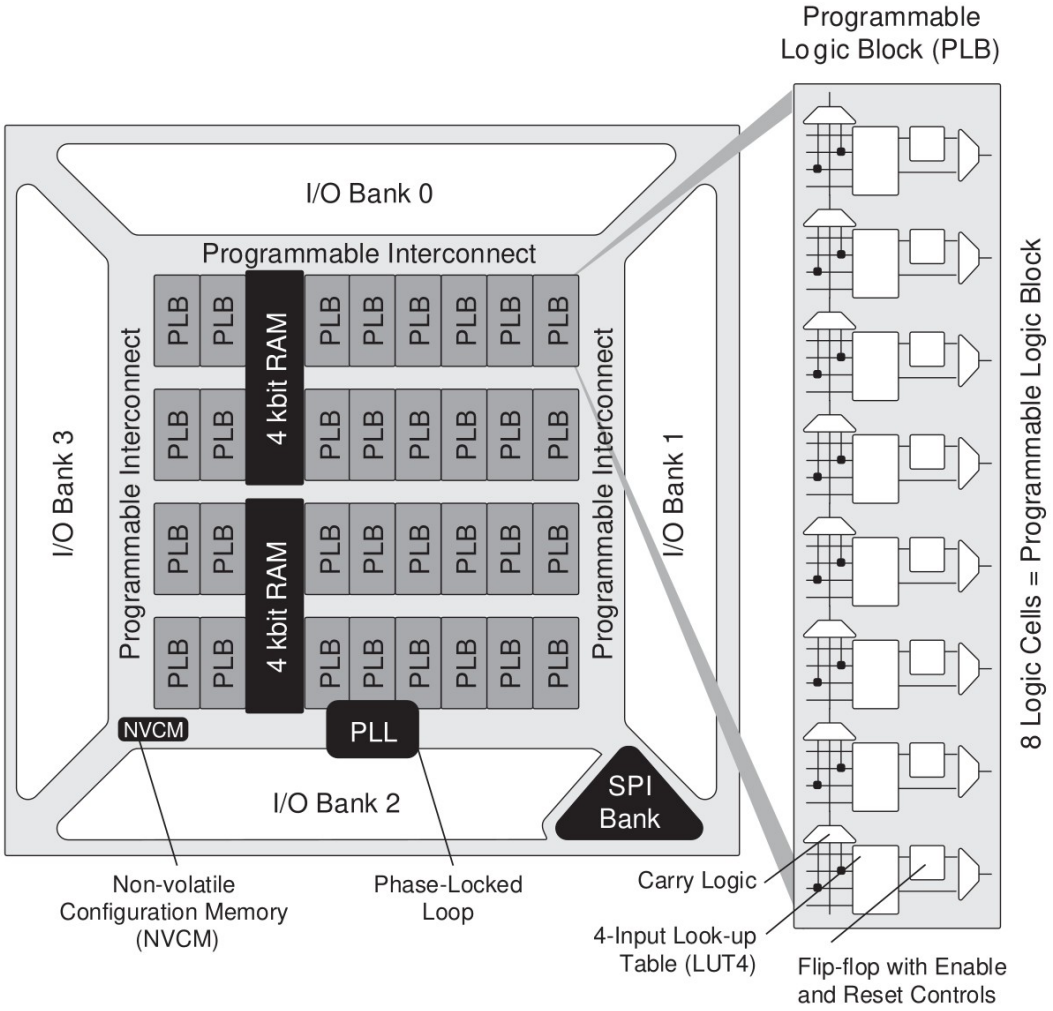


The tools listed above intervene in one or more phases of this flow, as we will see later.

15 https://github.com/YosysHQ/prjtrellis
16 https://github.com/YosysHQ/icestorm
17 https://github.com/steveicarus/iverilog
18 https://www.veripool.org/wiki/verilator
19 https://github.com/YosysHQ/yosys
20 http://gtkwave.sourceforge.net/

3 Architecture of iCE40 FPGA

iCE40 devices (the number refers to the 40nm design technology) are built as an array of programmable logic blocks (PLB): a PLB is a block of 8 logic cells and each logic cell consists of a four-input lookup table (called 4-LUT or LUT4) with the output connected to a D-type flip-flop. Within a PLB, each logic cell is connected to the next and previous cells by carry logic, to improve the performance of constructs such as adders and subtractors. Interspersed with the PLBs are RAM blocks, each of 4Kbits. The number of RAM blocks depends on the device (for the iCE40 LP8K, used on the TinyFPGA BX, there are 32 blocks for a total of 128Kbit of RAM). All blocks are connected horizontally and vertically via routing channels which are configured by the loaded design in a suitable bitstream format. In the FPGA there are also one or two PLLs, an SPI bank dedicated to the acquisition of the initial configuration and optionally a non-volatile configuration memory (NVCM).



At the edges of the device, divided into a maximum of four banks, there are the PIO cells. While the PLBs contain the building blocks for logic, arithmetic, and register functions, the PIO cells contain flexible I/O buffers capable of supporting several standard interfaces. On some iCE40 devices, each bank has its own power pin (VCCIO_x, with x=0,1,2,3), which allows different voltages to be used for the I/O banks, so that the FPGA can simultaneously support multiple interface standards (with voltage levels always between 1.8V and 3.3V). However, on the TinyFPGA BX all four VCCIO_x pins are connected to the 3.3V power supply, so all the I/O pins of the board work with the 3.3V logic (LVCMOS33 standard).

Cell structure

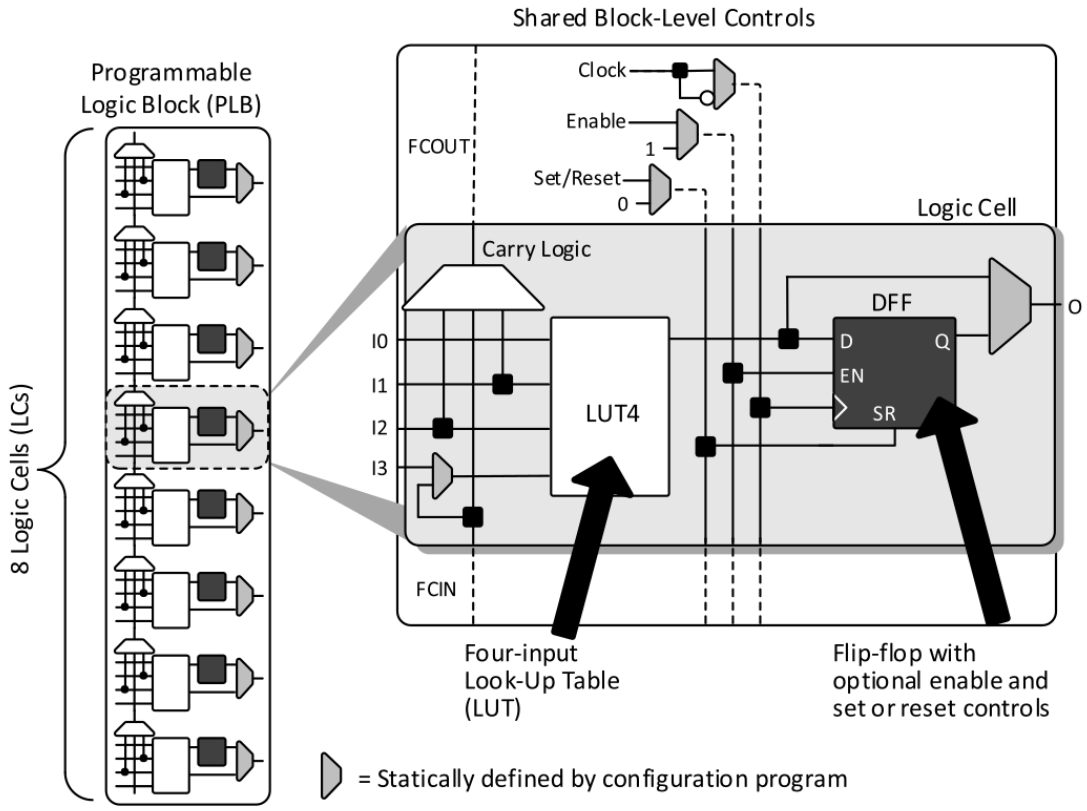
We saw briefly that the core of the iCE40 FPGA consists of a large amount of programmable logic blocks, each consisting of eight interconnected cells. Each cell includes several logic elements:

- A look-up table, capable of implementing any combinational logic function, with a maximum of 4 inputs (LUT4). Basically, the table behaves like a 16x1 ROM. By combining multiple LUT4s in cascade, larger logic functions can be created.

- A D-type flip-flop, equipped with set/reset and enable input, used to implement sequential logic functions. Each flip-flop is also connected to a global reset signal (not shown in the figure below) that is automatically activated immediately after the FPGA configuration.

- A “Carry Logic” signal whose purpose is to optimize (in terms of cells used and propagation time) the implementation of some basic arithmetic functions (such as adders, subtractors and comparators), binary counters and extensive cascaded logic functions. Note that the iCE40 does not have dedicated hardware multipliers (some FPGAs, such as the ECP5, have them).

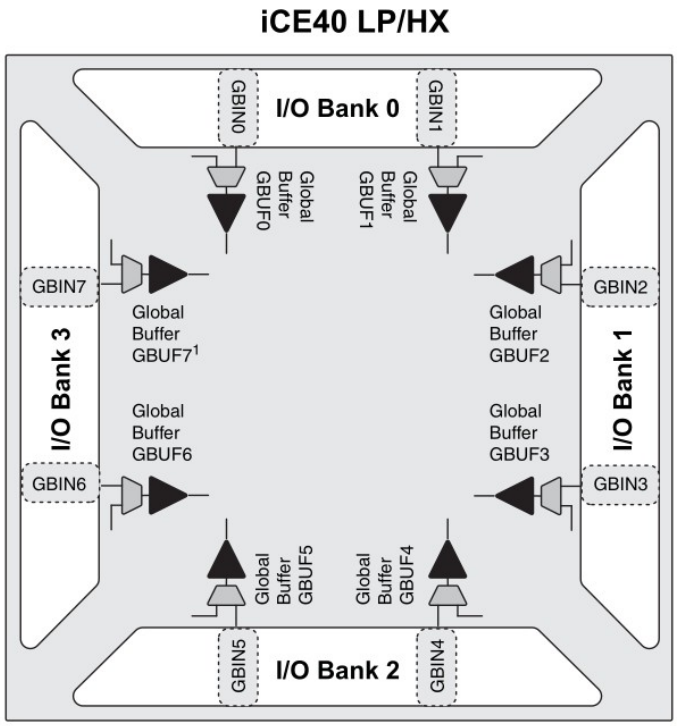
- Several 2-input multiplexers (represented in light gray in the following figure), in which the selected input is statically defined by the loaded configuration.



As you can see in the figure above, the set/reset signal, the enable signal and the clock are shared among all cells of the same PLB.

Global Buffer

The FPGA iCE40 internally provides eight lines with high "*fan-out*"[21] and low "skew"[22] called "*Global Buffer*" (GBUFx, with x=0..7). These lines are mainly designed to distribute clock and other signals (set/reset, enable, etc) that must be able to drive many logic inputs with the minimum possible delay. In each I/O bank there are specialized inputs for two Global Buffers:



1. GBUF7 and its associted PIO are best for direct differential clock inputs.

The following can be applied to the inputs of each GBUFx Global Buffers:

- the output of a PLL;
- the corresponding "Global Buffer Input" (GBINx/Gx) available on the I/O bank;
- an internal interconnection;
- a PIO cell.

The associated GBINx/Gx pin represents the best way to drive a Global Buffer from an external source, and should therefore be preferred (if possible) over the other PIO cells.

The optimal use of a Global Buffer is not necessarily automatic: although the FPGA resource allocation program tries to optimize the allocation of the Global Buffers according to the fan-out used, the developer could in some cases make better choices. It is always possible to force the use of a Global Buffer through the primitives "SB_GB"[23] or through the "SB_GB_IO". For details, please refer to the provided link and the document "*Lattice iCE Technology Library*"[24].

21 https://en.wikipedia.org/wiki/Fan-out
22 https://it.wikipedia.org/wiki/Skew_%28elettronica%29
23 https://www.mjoldfield.com/atelier/2018/02/ice40-blinky-icestick.html
24 http://www.latticesemi.com/view_document?document_id=52046

Programming

iCE40 devices are SRAM-based FPGAs. SRAM memory cells are volatile, meaning that once power is removed, the device configuration is lost and must be reloaded the next time it is powered up. This approach is useful in prototyping because it allows us to reprogram the device as many times as we want, but it also requires that the configuration information are stored elsewhere and read each time power is applied to the FPGA. For this reason, the iCE40 devices usually have beside an external flash memory to store such configuration data, or a microcontroller capable of transferring it to the FPGA. Some iCE40 devices also posses an internal non-volatile OTP (One Time Programmable) memory called NVCM. Using the NVCM eliminates the need for an external flash memory (or microcontroller), allows the FPGA to instantly configure itself and improves security by making reverse engineering more difficult, but prevents any design upgrades or reuse of the FPGA for other projects. The iCE40 LP8K of the TinyFPGA BX includes the NVCM memory but it is better to forget about this feature!

Loading the FPGA configuration from the outside can be done in two different ways, both based on the SPI interface:

- *Master SPI Configuration Mode*: the operating configuration is read from an external flash memory by the FPGA after the power-up phase;

- *Slave SPI Configuration Mode*: The configuration is transferred from a microcontroller, DSP or normal CPU at appropriate time (in the meantime the FPGA waits).

The standard configuration sequence is shown below:

1. at the beginning of the configuration all SRAM is cleared, pull-ups enabled, I/O pins set to high impedance;

2. as the download continues, the SRAM bits are gradually replaced with the received values (new bits take effect immediately);

3. At the end of the transfer, the I/O pins are released to the loaded design, except for the SPI pins which remain available to the FPGA for additional 49 clock cycles before becoming normal I/O for the application.

For more details on the sequence and configuration methods please refer to the technical note "*iCE40 Programming and Configuration*"[25].

The iCE40 FPGA can also be programmed directly by the Linux kernel[26]. A special driver has been created to allow the integration of such FPGA into some *Embedded Linux* boards.

iCE40 devices (like most FPGAs) are designed to be programmed using a hardware description language (HDL). The details of the binary bitstream format (which defines how the FPGA's internal elements are connected and interact with each other) are generally not made public by FPGA manufacturers, so developers are forced to use tools provided by the manufacturer (that have limitations and sometimes are even old and bugged). For example, for iCE40 devices, Lattice Semiconductor provides the ICEcube2 program for Windows and Linux (but not MacOS) that needs a license.

25 http://www.latticesemi.com/-/media/LatticeSemi/Documents/ApplicationNotes/IK/FPGA-TN-02001-3-2-iCE40-Programming-Configuration.ashx?document_id=46502

26 https://hackaday.com/2017/04/13/lattice-ice40-fpga-configured-by-linux-kernel/

Fortunately, for the iCE40, there is also a fully open-source toolchain constantly updated, consisting of “Yosys” (Verilog synthesis front-end), “NextPNR” (bitstream generation in text format) and “IcePack” (conversion from bitstream in text format to bitstream in binary format)[27]. IcePack is part of the IceStorm project[28], which aspire to describe the binary bitstream format of all iCE40 FPGAs.

It has been demonstrated how it is possible, with this toolchain, to build a fully functional RISC-V SoC on the iCE40 HX8K. The toolchain supports several iCE40 devices, including the LP8K (fitted on the TinyFPGA BX) and is fully integrated in the Apio environment (which has the advantage of making the toolchain very simple to use). Alternatively, the entire toolchain can be used through the “*make*” command[29] or the “*CMake*” build system[30].

Conclusions

The FPGA iCE40 is a low-end, low-cost device that can be used for prototyping and small-volume productions. For large volumes (millions of pieces) FPGAs are never convenient because they’ll always have a much higher cost than any other integrated circuit of the same logic complexity. As a matter of fact, with the same number of gates and flip-flops available for the application, an FPGA needs much more space and logic dedicated to the programmable interconnections, which considerably increases its price (there are FPGAs which exceed 100000$). It is therefore common practice, for large volumes, to use an FPGA in the development phase and then carry out the true production on ASICs (Application-Specific Integrated Circuit). Note that, for small volumes, ASICs are never convenient, due to the design cost of the production process, which although not recurring is typically very high (millions of $)[31].

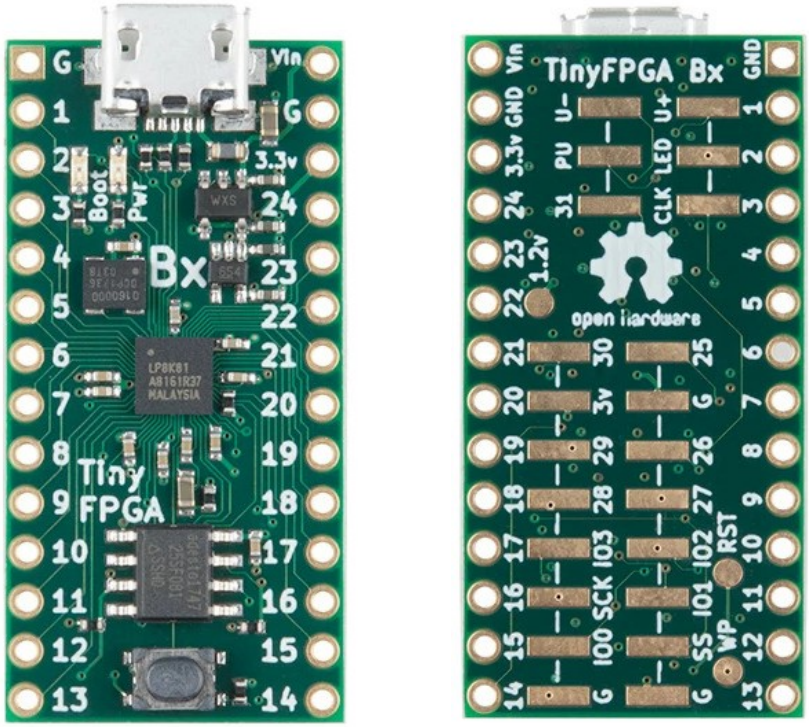
To simplify the design and make the iCE40 series less expensive, Lattice Semiconductor chose to implement the logic functions using LUT4 tables. Compared to the expensive LUT6-based FPGAs (such as the Xilinx 7 series or the Altera Stratix), LUT4-based FPGAs may need more cells to implement the same logic functions, especially if they are complex. For example, a logic function with seven inputs needs only two LUT6s but requires two or more LUT4s. It has been seen that, on average, a complex design implemented with LUT4 tables requires 15% to 20% more cells than the implementation with LUT6 tables. However, LUT4s are smaller and require less energy: miniaturization is increasing the number of LUT4s per die and, thanks to more efficient mapping algorithms capable of better exploiting LUT4s, LUT6 tables are gradually losing their original advantage[32].

Despite its simple structure, the iCE40 is an excellent FPGA and its low cost makes it ideal as a tool to start learning the world of programmable logic and to implement logic control circuits, small custom microcontrollers and simple game consoles.

27 https://hackaday.com/2015/12/29/32c3-a-free-and-open-source-verilog-to-bitstream-flow-for-ice40-fpgas/
28 https://github.com/YosysHQ/icestorm
29 https://github.com/nesl/ice40_examples
30 https://www.walknsqualk.com/post/015_fpga_design_p2/
31 https://en.wikipedia.org/wiki/Application-specific_integrated_circuit
32 https://people.eecs.berkeley.edu/~alanmi/publications/2018/fpga18_s44.pdf

4 Board TinyFPGA BX in detail

Below are given some essential information for the correct use of the TinyFPGA BX. As you can see in the following pictures, the board provides part of signals and supply voltages on the side headers, and the rest on SMD pads on the back. Here we will consider the use of the board with a breadboard, so we will mainly examine the signals and supply voltages provided on the headers. If necessary, you can always solder (carefully) a flat cable on the required SMD pads, to bring the related signals to the breadboard.



The TinyFPGA BX can be powered directly through the USB connector (+5V) or by applying a voltage from +3.5V to +5.5V on the “Vin” silkscreened pin (see right side of the USB connector).

A stabilized voltage of +3.3V is generated on board, and used to power the I/O sections of the FPGA. Such voltage is also available on the “3.3v” silkscreened pin (2nd pin after Vin). The TinyFPGA BX documentation states to draw not more than 100mA from this pin, probably to keep the dropout of the MIC5504-3.3YM5-TR regulator below 200mV. But providing at least +3.7V on the “Vin” input pin it should be possible to draw much more than 100mA, depending on the current used by the FPGA I/O ports.

The board also generates a stabilized voltage of +1.2V, required to power the FPGA core. This voltage is available on the circular “1.2v” silkscreened SMD pad (near the “open hardware” logo). The maximum current that can be drawn from this pin is 100mA. Indeed, the MIC5365-1.2YC5-TR regulator can supply a maximum of 150mA and 50mA are reserved for the FPGA. On the schematic this pad is indicated with “TEST TP3”.

Ground connections are available on two header pins and three SMD pads on the back, all silkscreened with “G”.

The 16MHz clock generated by the oscillator DSC6001CI2A-016.0000T is applied to pin B2 of the FPGA (IOL_2B); it is also provided on the “CLK” silkscreened SMD pad.

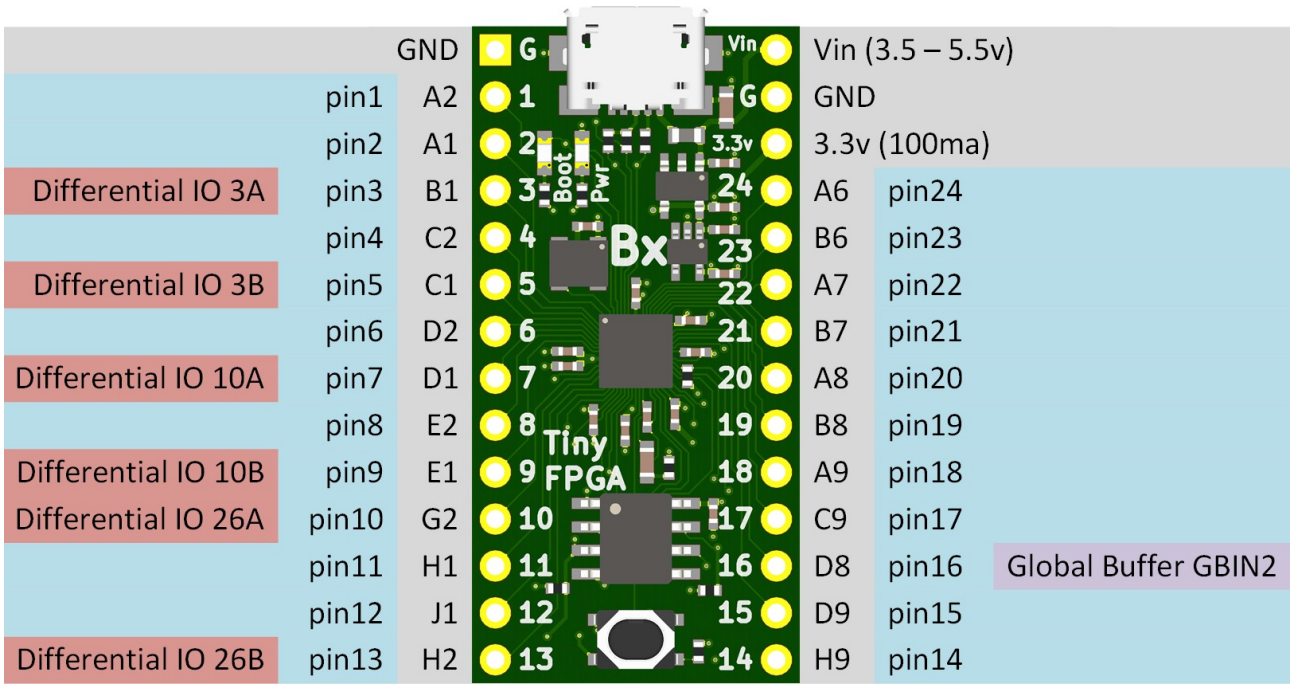
There are two LEDs on the board, one connected directly to the +3.3V and used to indicate the presence of power, the other connected to pin B3 of the FPGA (IOT_218) and used for both the bootloader and the user's design. This signal is routed on the "LED" silkscreened SMD pad.

On the back of the board there is a reset signal (SMD circular pad silkscreened "RST"), but from the schematic ("TP2 TEST") we can see that this signal, normally high (+3.3V), is made low (GND) only by the SW1 button, that is used to restart the bootloader after a design has been loaded. It is therefore not a true reset signal that can be used by external logic, as it is not temporarily active after power is supplied and is not filtered by debounce logic. However, it is possible to generate a true reset signal with the FPGA[33].

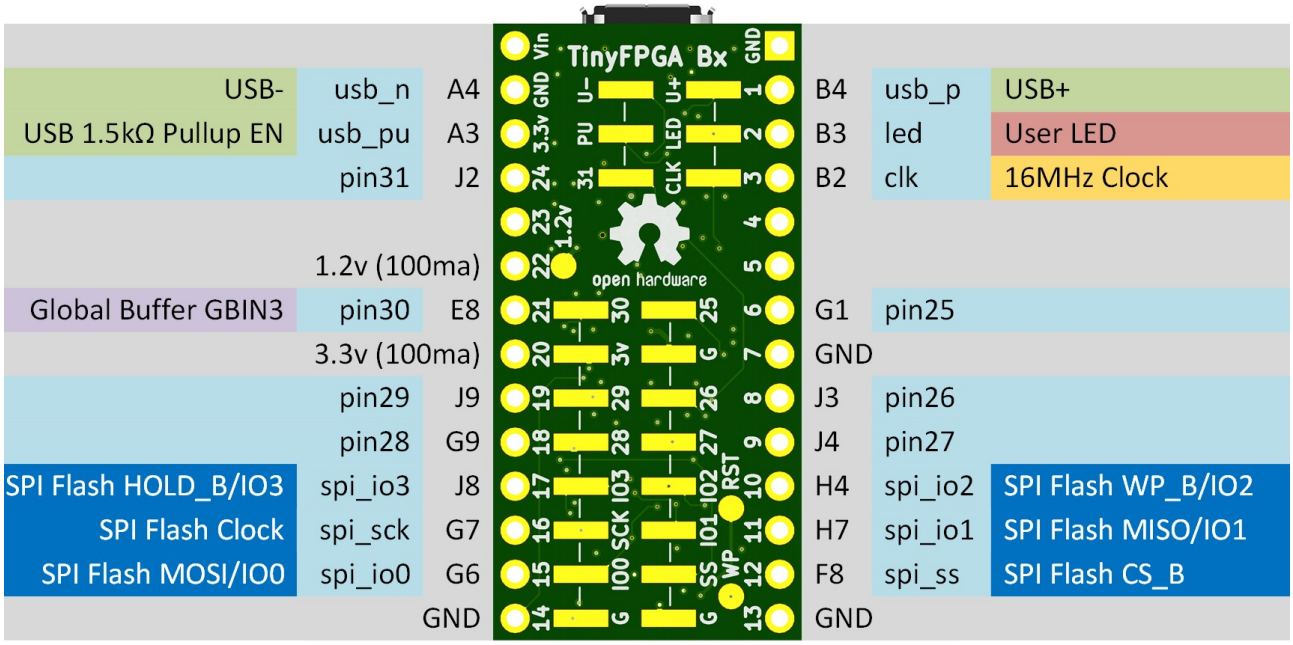
As you can see from the following pictures, there are 24 free FPGA I/O on the headers, while other 7 are available on the back, for a total of 31 usable I/O. On the back side of the TinyFPGA BX there are also 10 I/Os already used (USB_N, USB_PU, USB_P, LED, SPI_IO3, SPI_CLK, SPI_IO0, SPI_IO2, SPI_IO1, SPI_SS) for a total of 41 I/Os.

Although the TinyFPGA BX can be powered at +5V, the I/O pins work at 3.3V and tolerate a maximum voltage of 3.5V (LVCMOS33 standard, see the FPGA datasheet). Never connect the TinyFPGA BX with 5V logic without a proper interface circuit, nor apply +5V directly to the board inputs, otherwise a serious damage will be caused to the FPGA. In addition, note that the FPGA I/O pins can withstand a maximum current of 8mA.

For interfacing, discrete solutions are possible for both 3.3V → 5V[34] and 5V → 3.3V[35] but the best approach is to use dedicated ICs, such as the 74LVCH1T45 or 74LVCH8T245, which allow bi-directional conversion at maximum speed and low load for the I/O ports.



33 https://stackoverflow.com/questions/38030768/icestick-yosys-using-the-global-set-reset-gsr
34 https://next-hack.com/index.php/2020/02/15/how-to-interface-a-3-3v-output-to-a-5v-input/
35 https://next-hack.com/index.php/2017/09/15/how-to-interface-a-5v-output-to-a-3-3v-input/



Both the bootloader and the application circuit are saved on the SPI flash AT25SF081-SSHD-B. This memory guarantees 100,000 write/erase cycles and a retention life of 20 years. The power drain of the SPI flash, while reading, is only 4mA, which is reduced to 20uA in standby. The maximum operating frequency of such memory is about 100MHz. For the operation of the SPI flash, please refer to the related datasheet.

Bootloader behavior

The bootloader behaves differently depending on what is connected to the USB port:

- If the board is connected to a host via USB data cable, the bootloader remains active waiting for the board to be programmed by "*tinyprog*". It is possible to force the exit from the bootloader and load an application image already present in the SPI flash, using the "*tinyprog* -b" command.

- If the card is simply powered via USB (e.g. from a powerbank or using a cable without the data lines) or via the "Vin" pin, the bootloader times out after one second, loading the application image from the SPI flash.

Pressing the reset button will make the card to always reactivate the bootloader, but it will remain in the bootloader (waiting for a new firmware) only if it is connected to a host with a USB data cable.

Pull-up

On some pins of the iCE40 FPGA (and therefore of the TinyFPGA BX) we can enable pull-up resistors. There are two different ways to do this[36]: by editing the .pcf file (adding "*-pullup yes*" to "*set_io*") or by using the "*SB_IO*" directive. For the latter case, please refer to the link provided and to the document "*Lattice iCE Technology Library*"[37]. Note that such pull-ups are static, they cannot be changed after the FPGA configuration is loaded. Also, only I/O banks 0,1,2 have the pull-up resistors (on bank 3 the enabling is ignored).

36 https://discourse.tinyfpga.com/t/internal-pullup-in-bx/800
37 http://www.latticesemi.com/view_document?document_id=52046

5 Simulation and programming of the TinyFPGA BX

These are the main steps required to create and simulate a design for the iCE40 FPGA (in particular for the TinyFPGA BX) using the Apio environment:

- Create a clean folder for the project and configure the Apio environment for the TinyFPGA:

  apio init --board TinyFPGA-BX

  Note: with the command "*apio boards --list*" you can see all supported boards.

- Copy the "*TinyFPGA-BX-pins.pcf*" file, found in the blinkSOS example directory, to the folder you just created (it discloses the pins of the TinyFPGA BX to the Verilog code).

- Create one or more "**.v*" project files with the required Verilog code.

- Verify, using Icarus Verilog, that there are no rough errors in the code:

  apio verify

- Verify, using Verilator, the correctness of the code (more formal check):

  apio lint

- Create a "*testbench*" module to simulate the project with Icarus Verilog and GTKWave. Note that the testbench file must end with "*_tb.v*" otherwise it will be considered part of the design to be synthesized; also, the testbench must instantiate a single module, hierarchically superior to all (it is not possible to instantiate more than one module in the testbench).

- Start the simulation using the command:

  apio sim

  At the end of the simulation GTKWave will be started (see below).

- Generate the timings analysis file "hardware.rpt":

  apio time

  Visual inspection of such a file is essential to verify that the signal propagation time through the synthesized logic is compatible with the clock duration.

- Synthesize the project using the IceStorm Tools:

  apio build -v

  The -v option displays the build process and reports the amount of FPGA resources used (links, cells, LUT4, flip-flops, carry units, memory blocks, global buffers, etc) first from "*yosys*" (not optimized) and then from "*nextpnr-ice40*" (optimized and mapped on the iCE40 L8K). Note: the useful information must be searched among lot of displayed lines!

- Connect the TinyFPGA BX to the PC and upload the bitstream code through tinyprog:

  apio upload

- Delete unnecessary files generated by the previous commands:

  apio clean

Notes about the simulation

At first GTKWave starts without displaying any information. The reason is that a project can have hundreds of modules and thousands of different signals, so it is up to the developer to decide which ones to display. For this purpose, in GTKWave:

- click on the "*testbench*" module in the top left window (under SST);
- in the window below, the signals defined in that module will be highlighted; drag the ones you want into the "*Waves*" window (on the right);
- expand the GTKWave application to the desired size, possibly until the entire available area is covered;
- in the toolbar click on the "*Zoom Fit*" button;
- add the signals of any other modules of the project that you want to examine (expand the "testbench" module and any sub-modules);
- on the keyboard type CTRL-S (or click on the menu item "*File*" → "*Write Save File*") to save a file "**_tb.gtkw*" with the settings: this way, if the simulation is restarted, the GTKWave application will automatically show the chosen signals with the saved time scale.

NOTE: if you change the signals names or signal sizes (if they are vectors) you will need to add them again to the "*Waves*" widget.

If signals are not propagated between modules in the simulation, check that their instantiation is correct. The classic function call format should never be used:

```
module_template MODNAME (       /* WRONG INSTANTIATION */
        wire1,
        wire2,
        ...
);
```

All arguments in Verilog must be provided with the following format:

```
module_template MODNAME (       /* CORRECT INSTANTIATION */
        .arg1( wire1 ),
        .arg2( wire2 ),
        ...
);
```

6 Brief notes about the hardware synthesis in Verilog

Verilog is a somewhat ambiguous language, born to bring C developers into the world of hardware synthesis. It has gone through several revisions (the latest is the IEEE 1364-2005) which have incremented the number of linguistic constructs. People approaching Verilog and the world of hardware synthesis are initially confused by the fact that some operations can be performed in different ways, some constructs are obsolete or not recommended, some terms used in the description of the language are confusing. Purpose of this section is to clarify common perplexities that arise when learning Verilog. Note that there is also another language used in hardware synthesis, VHDL: it is more formal (but also more verbose), and is particularly popular in military, aerospace and mission-critical environments. The two languages are practically equivalent and no one has a real dominance over the other. We will not talk about VHDL here.

In the next few pages I have annotated some observations on topics that I personally found unclear during my approach to Verilog. The following sections are not intended to be a Verilog course, so it is recommended that you read an introduction to Verilog 2005 before continuing.

“Behavioral”, “RTL” and “Gate Level” models

Such terms are often encountered learning Verilog and VHDL, so let's see what they mean:

- ”*Behavioral*” model means code that describes the behavior of the system at high-level, suitable for simulation, but not for synthesis. This code can be used as first step to describe the design as a whole, before the detailed “*RTL*” code of the individual modules is created. But it is also used for the “*testbench*”, i.e. the environment surrounding the project in a simulation (the part that simulates the stimuli and allows the results to be represented in an understandable way). The output of the testbench is usually made up of time diagrams, or higher level visual representation, depending on the design (for example, if the project imposes a video output on VGA or HDMI, the testbench could even simulate a monitor).

- With “*RTL*” (Register Transfer Level) model we mean synthesizable code, usually split into modules, capable of describing the behavior of the system in terms of medium-level combinational and sequential logic (registers, mux/demux, rom, ram, etc) . The RTL code is always simulated to verify the correctness of the project (*pre-synthesis simulation*), but it is also used as input for the synthesis to generate the “*Gate Level*” code (see below). In the RTL, explicit delays are not considered, and the timings are determined by clock edges. “Latch” structures (triggered by levels instead of edges) are generally avoided, the registers are synchronized with one or more clocks, and it is assumed that the propagation of the combinational logic stabilizes within a single clock cycle. The constructs that can be used in the RTL description are a specific subset of what Verilog or VHDL make available, since a big part of these languages is dedicated exclusively to the behavioral simulation (Verilog and VHDL were born as tools for simulation and only later were applied to the synthesis).

- “*Gate Level*” model means code that describes the system in terms of flip-flops, logic gates and other basic constructs that can be directly mapped on the elements provided by FPGAs and ASICs. The “Gate Level” model has lower level than the RTL one: indeed, by the synthesis of the RTL code we practically obtain “Gate Level” code. The simulation of this code (*post-synthesis simulation*) is essential to verify that the synthesis has been carried out correctly, but generally it takes much more time than the RTL simulation (because operations that were previously performed with a single instruction, for example a

multiplication between numbers, are now split in the simulation of tens, hundreds or thousands of elementary components).

“Always” and “initial” blocks

A block in Verilog encapsulates a section of code, that is simulated and synthesized independently of the code in other blocks. In practice, each block is synthesized on different hardware and simulated in an independent process, parallel to the others. All blocks in Verilog fall into two broad categories: repetitive and non-repetitive. The repetitive ones are defined by:

- *always @ (<sensitivity list>)*
  block_of_statements;

- *initial forever*
  block_of_statements;

- *initial while(1)*
  block_of_statements;

- *initial begin forever*
  block_of_statements;
  end

Regarding the simulation, the four definitions are fully equivalent, but only “*always*” is accepted in the synthesis, so this construct must be used within the modules that must be synthesized. Note that “*forever*” and “*while(1)*” (which express an infinite loop) can also be used in the body of an “*if*” condition but only in simulation, because it is not possible to “unroll” an infinite loop and transform it into a group of parallel logic circuits (while this can be done for a finite loop, within some limits). Always remember that Verilog code represents logic hardware and not a program that is executed!

Non-repetitive blocks are performed only once. They are defined by:

- *initial begin*
  block_of_statements;
  end

To terminate the simulation after some time, you must use the keyword “*$finish*” at the end of the non-repetitive block: it causes the interruption of all blocks, including the repetitive ones. The testbench usually has a non-repetitive block containing one or more delays (*#N*) before the keyword “*$finish*”: the sum of all delays will determine the duration of the simulation.

Type “reg” and “wire”

All variables and registers assigned inside an “*always*” (or “*initial*”) blocks must have the “*reg*” type; all variables and signals assigned outside the “*always*” (or “*initial*”) blocks must have the “*wire*” type. Note that the “*reg*” keyword does not necessarily involve the instantiation of a hardware “register”. Inputs and outputs without a type definition are automatically considered “*wire*” type.

Assignments and events

Assignment in Verilog can be of two types:

- "*Non-blocking assignment*" (<=): characterizes the assignments that are all performed simultaneously when the event chosen as trigger (or events chosen as trigger) occurs. It is normally used with the "*reg*" type to describe sequential logic, for example:

```
reg ff1, ff2;
always @ (posedge clk) begin
        ff1 <= ff2;
        ff2 <= ff1;
end
```

  At the trigger (positive edge of the clock) the value of ff2 is "captured" by ff1; at the same time ff1 is "captured" by ff2. The two assignments are then executed "in parallel". This happens because ff1 and ff2 are both "reg" types and they are assigned with a non-blocking assignments, so they can be automatically implemented as outputs of two D-type flip-flops. As stated above, it is possible to specify more trigger events:

```
always @ (posedge clk, negedge reset) begin
        if (~reset) begin
                ff1 <= 1'b0;
                ff2 <= 1'b0;
        end else begin
                ff1 <= ff2;
                ff2 <= ff1;
        end
end
```

  Now, the positive edge of the clock and the negative edge of the reset both cause the execution of the block, The "*if(~reset)*" statement discriminates the reset phase (reset of flip-flops) from normal operation. Basically we added an asynchronous reset (active low, because the reset is performed when "*~reset*" is true, i.e. when "*reset*" is false, i.e. at 0V).

- "*Blocking assignment*" (=): normally used with the "*wire*" type to define combinational logic, it blocks the execution (or interpretation) of the next instruction until "completion". In hardware and simulation terms this means that the result of the next instruction may depend on the current one. For example, the sequence:

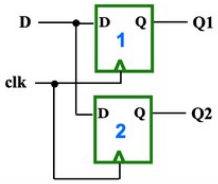
```
c = a & b;
e = c ^ d;
```

  is translated to: *e = (a & b) ^ d*. Blocking assignments can therefore be seen as performed "in series", especially if there is a dependency relationship between the expressions (otherwise they are performed in parallel). At the circuit level, blocking assignments define an (almost) instantaneous and continuous propagation of the values through the logic gates.

Note that the two types of assignments, when used on sequential logic, generates in many cases completely different circuits, as illustrated in the following example:

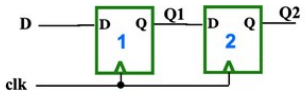
Blocking Statement:

```
module block (
   input D, clk,
   output reg Q1, Q2
);

   always @ (posedge clk)
   begin
      Q1 = D;
      Q2 = Q1;
   end
endmodule
```



Non-Blocking Statement:

```
module block (
   input D, clk,
   output reg Q1, Q2
);

   always @ (posedge clk)
   begin
      Q1 <= D;
      Q2 <= Q1;
   end
endmodule
```



So be very careful: a simple distraction can turn a non-blocking assignment into a blocking one (just forget the '<') messing up the circuit. As general rule, never use the blocking assignment inside the "always" blocks that have rising and falling edges as parameters of the "sensitivity list".

The "*sensitivity list*" may also consist of a list of signals without the "*posedge*" or "*negedge*" specifiers. In this case the "*always*" block defines some combinational or asynchronous sequential logic (it must be considered as a continuously active block). For example the following code implements a simple latch with double output "q" and "q_n" (q negated):

```
reg q;
always @ (enable, d) begin
        if (enable) begin
                q    <= d;
                q_n <= ~d;
        end
end
```

(note that using latches is not recommended, you should always prefer the synchronous logic to the asynchronous one). The next example defines a 4-input multiplexer (a,b,c,d → y):

```
reg y;
wire [1:0] sel;          // input selection
wire a, b, c, d;
always @ (sel, a, b, c, d) begin
        case (sel)
                2'b00 : y = a;
                2'b01 : y = b;
                2'b10 : y = c;
                2'b11 : y = d;
        endcase
end
```

As mentioned above, all the variables assigned within a block must always be defined as "reg" type (this type does not necessarily imply the instantiation of a hardware register).

Note that both blocking and non-blocking assignments can be used to model blocks with combinational logic, but the blocking one is preferred because it is much more readable (and with some bugged synthesis tools even more efficient). Following this rule, it only takes an instant to distinguish a block with memory from one with only combinational logic.

Statement "assign"

Another way to define combinational logic (without latches) is with the "*assign*" statement, for example:

```
assign {c_out, sum} = a + b + c_in;
```

It must be used outside the "*always*" and "*initial*" blocks. All "assign" statements are to be interpreted as being executed in parallel, unless there are dependency relationships.

Some basic rules

There are some mandatory rules to follow when using assignments[38]:

- Never mix blocking and non-blocking assignments on the same variable.
- Never assign the same variable in two different blocks as this causes a conflict (two different entities trying to impose their value on the same physical resource). The rule obviously applies to both assignments. If a resource has to depend on two different blocks, we must use two different variables (one for each block) and define with the keyword "assign" a third variable external to the blocks (of type "wire"), whose value is given by a suitable combinational logic operating on the other two variables, for example:

```
always @ (posedge clk1)
        cnt1 <= cnt1 + 1;

always @ (posedge clk2)
        cnt2 <= cnt2 – 1;

assign out = cnt1 ^ cnt2;
```

- Note that reading in a block a variable that is modified in another block does not create issues if the two actions take place at different times (considering the propagation delays), for example one on the positive edge and the other on the negative edge of the same clock. If this condition cannot be guaranteed (for example because the two blocks are activated by asynchronous events) and there is a risk that the variable is read and modified at the same time, then the project design must be re-engineered.
- Note that you should never activate the same block on both positive and negative edges of the same signal. This is because FPGAs generally do not have "*Double Edge Triggered*" flip-flops (i.e. memory elements active on both edges). Usually the synthesis tools recognize this inconsistency but it is always good to avoid such condition.

38 http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA.pdf

Violation of these rules can lead to errors in synthesis, malfunctions, or a different behavior between the simulation and the synthesis.

Register initialization

Initialization of registers (flip-flops) at declaration time (or by *initial begin...end* block) is possible, if the synthesis targets an SRAM-based FPGA. For example, the following declaration tells the compiler that the value of the register at power-up must be 5:

```
reg [3:0] count = 4‘b0101;
```

Unfortunately this practice is not portable[39]. It works perfectly on iCE40 and ECP5 FPGAs with the open-source tools described above, while it does not work with iCEcube2, which ignores the specified values and always forces the initialization to 0. Among FPGA manufacturers the issue is somewhat debated: for Xilinx such initialization is even recommended, while for others it should be avoided! Personally, I believe that the best practice is that one which allows maximum code portability, i.e. the use of a reset signal for all registers and control variables:

```
always @ (posedge clk, posedge reset) begin
        if (reset) begin
                // Reset value goes in here.
                count <= 4‘b0101;
                ...
        end else begin
                // Code executed when the reset is deasserted.
                ...
        end
end
```

Besides the fact that some FPGA manufacturer’s synthesis tools don't accept it (or worse, ignore it), there are other reasons why you should avoid the initialization in the declaration:

- In an FPGA, the initialization values become part of the bitstream file and are loaded during the FPGA “configuration”. This practice is only possible for SRAM-based FPGAs: other technologies (such as ASICs) do not support the power-up initialization.

- Such initialization can be performed only immediately after loading the configuration (i.e. at the FPGA power up). Instead, an explicit reset input for each module allows us to bring the system back to a reset state after various events: power-on, push button (with anti-bounce logic), voltage drops, internal errors, etc.

- Mixing the initialization in the declaration with the one by explicit reset can create issues,, especially in big projects: there is a risk that a reset signal (generated after the power-on) will resets only a part of the logic, creating two desynchronized logic sections!

In summary: it is good thing to reserve the initialization in the declaration for very special cases and instead provide a reset signal in the sequential logic, explicitly implementing the appropriate initialization. To avoid undefined states and signals from appearing in the simulation, just simulate the activation of the reset for a few clock cycles at the beginning of the simulation.

39 https://stackoverflow.com/questions/10005411/assign-a-synthesizable-initial-value-to-a-reg-in-verilog

Synchronous and asynchronous reset

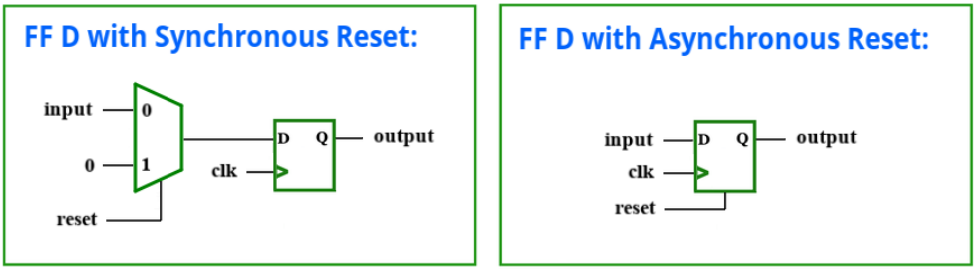
As mentioned above, the reset realizes a fundamental component of the design that allows us to put flip-flops, registers and other memory elements in a well-defined state. There are two type of reset: *synchronous* and *asynchronous*. As an example, let's look at the Verilog code of a D-type flip-flop with synchronous (left) and asynchronous (right) reset:

```
always @ (posedge clk) begin
        if (reset) begin
                q <= 1'b0;
        end else begin
                q <= d;
        end
end
```

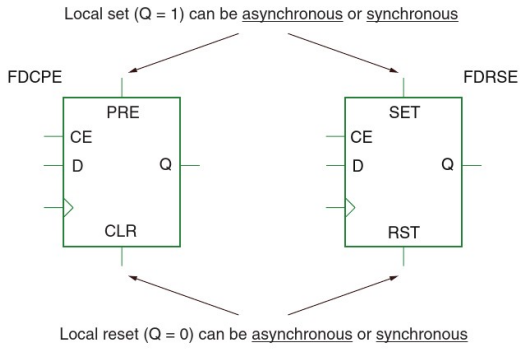
```
always @ (posedge clk, posedge reset) begin
        if (reset) begin
                q <= 1'b0;
        end else begin
                q <= d;
        end
end
```

As you can see, the code differs only for the arguments in the "s*ensitivity list*" of the "*always*" block: by definition the **synchronous reset** must act only on the edges of the clock, so the activation of the block must take place only on those edges; instead, the **asynchronous reset**, must act independently from the clock, for this reason it must be present in the "*sensitivity list*" of the block. Both examples above implement the "*active high*" reset; to obtain the "*active low*" reset just replace "*posedge reset*" with "*negedge reset*" and "*if(reset)*" with "i*f(~reset)*".

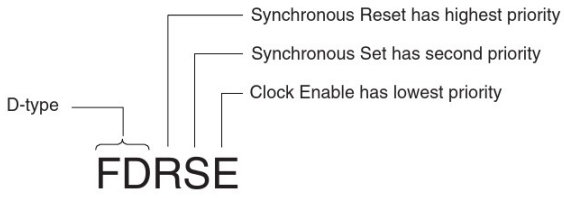
Note that although the Verilog code of the two implementations is very similar, the logic generated in the FPGA might be significantly different:



It would therefore seem that the synchronous reset, as a signal acting on input D of the flip-flops, consumes some combinational logic of the FPGA, causing also an increment of propagation times. Actually, this is not necessarily true: the final hardware configuration depends on both the type of basic elements that the FPGA makes available, and the order in which the various signals are specified in the Verilog code. To understand this concept, a premise is required. Many FPGA families (including the iCE40) can instantiate two types of flip-flops, which have the same controls, but differ in the type of initialization (asynchronous or synchronous):



FDCPE flip-flops have asynchronous inputs called CLR (*clear*) and PRE (*preset*) which can act at any time. Vice versa, in the FDRSE the inputs SET and RST (*reset*) are synchronous, i.e. they act only in correspondence of the selected clock edge (see below). When multiple controls are used, each one follows the assigned priority, indicated by the abbreviation itself:



The generated circuit is optimized if (and only if) the control order of the signals in the code follows the above scheme!

Before delving into some easy-to-understand practical examples, we need to look in more detail at the subtypes of flip-flop that can be instantiated. As mentioned, they strongly depend on the used FPGA family. For example, for the iCE40 we have the following basic types[40]:

SB_DFF	D flip-flop, clock active on positive edge
SB_DFFE	D flip-flop, clock active on positive edge, Clock Enable input
SB_DFFSR	D flip-flop, clock active on positive edge, Synchronous Reset input
SB_DFFR	D flip-flop, clock active on positive edge, Asynchronous Reset input (clear)
SB_DFFSS	D flip-flop, clock active on positive edge, Synchronous Set input
SB_DFFS	D flip-flop, clock active on positive edge, Asynchronous Set input (preset)
SB_DFFESR	D flip-flop, clock active on positive edge, Clock Enable & Synchronous Reset inputs
SB_DFFER	D flip-flop, clock active on positive edge, Clock Enable & Asynchronous Reset inputs
SB_DFFESS	D flip-flop, clock active on positive edge, Clock Enable & Synchronous Set inputs
SB_DFFES	D flip-flop, clock active on positive edge, Clock Enable & Asynchronous Set inputs
SB_DFFN	D flip-flop, clock active on negative edge
SB_DFFNE	D flip-flop, clock active on negative edge, Clock Enable input
SB_DFFNSR	D flip-flop, clock active on negative edge, Synchronous Reset input
SB_DFFNR	D flip-flop, clock active on negative edge, Asynchronous Reset input (clear)
SB_DFFNSS	D flip-flop, clock active on negative edge, Synchronous Set input
SB_DFFNS	D flip-flop, clock active on negative edge, Asynchronous Set input (preset)
SB_DFFNESR	D flip-flop, clock active on negative edge, Clock Enable & Synchronous Reset inputs
SB_DFFNER	D flip-flop, clock active on negative edge, Clock Enable & Asynchronous Reset inputs
SB_DFFNESS	D flip-flop, clock active on negative edge, Clock Enable & Synchronous Set inputs
SB_DFFNES	D flip-flop, clock active on negative edge, Clock Enable & Asynchronous Set inputs

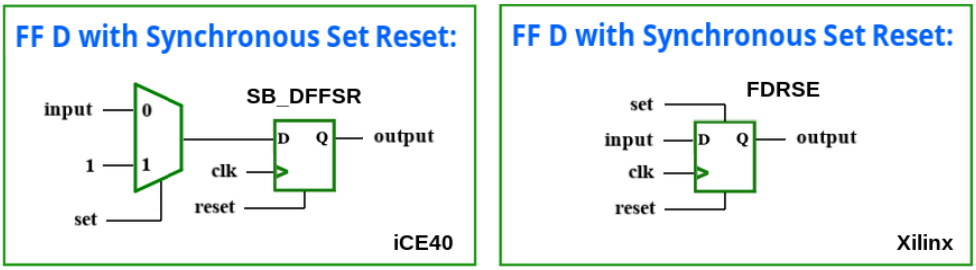
So, the two reset examples seen above, in the case of the iCE40, generate the following circuits:

Therefore in this case, for a synchronous type reset, no additional FPGA logic is used (nor propagation delays introduced). Let's see another example:

40 http://www.latticesemi.com/view_document?document_id=52046

```
always @ (posedge clk) begin
        if (reset)
                q <= 1'b0;
        else if (set)
                q <= 1'b1;
        else
                q <= d;
end
```

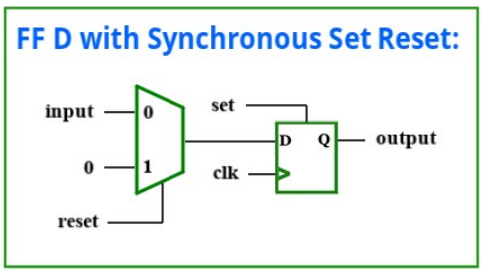
Note that, in the above list, there are no flip-flops with both Set/Reset (or Clear/Preset) inputs. Indeed, the iCE40 does not have this type of elements. Xilinx FPGAs, on the other hand, can instantiate this kind of flip-flops, so the generated circuit will be different in the two families:



What happens if for distraction, in the previous code, we swap the order of the set and reset signals?

```
always @ (posedge clk) begin
        if (set)
                q <= 1'b1;
        else if (reset)
                q <= 1'b0;
        else
                q <= d;
end
```

In that case the specified priority differs from the natural priority (denoted by FDRSE), and the synthesis program has no way of knowing that this is the result of carelessness! It will assume that this behavior is what the designer wants, and will use additional logic to satisfy this requirement. In practice the following circuit will be generated, even for the Xilinx FPGA:



Similar considerations can be made for the use of Clock Enable in relation to set/reset. Furthermore, such considerations can be extended (with necessary modifications) to the asynchronous set/reset (preset/clear). Note that you should not mix asynchronous reset with synchronous set (or vice versa) in the same block, as no FPGA on the market is able to instantiate elements that support both at the same time (in which case the unwanted use of additional logic levels is a certainty!).

So, we must pay attention to the code we write, in order to avoid the introduction of useless logic (which would increase the propagation times of some signals, causing possible violations of the project timings). For this purpose, we must study the basic elements of the FPGA, and perform some synthesis tests, observing how the resources of the FPGA are used.

Returning to the reset, it is natural to ask which is the best, if the synchronous or the asynchronous one. There is no simple answer: both have advantages and disadvantages, but also supporters and detractors! It is worth to examine both solutions and decide case-by-case which one to adopt:

- All FPGAs have FDCPE elements (classic flip-flops with Preset and Clear) but not necessarily flip-flops with synchronous set/reset. In the absence of FDRSE elements, the use of **synchronous reset** consumes combinational logic of the FPGA as it acts on the D input of the flip-flops. To reduce the use of this logic, it is sometimes possible to limit the reset to only some registers as long as this reset is kept active for a certain number of clock cycles. For example, in a "*shift register*" we can make the synchronous reset act only on the first flip flop and use a few clock cycles to propagate the reset state to the following flip-flops. In a complex project, the synchronous reset may require hundreds or thousands of clock cycles (and it is easy to lose track of which module is the most critical). Furthermore, care must be taken not to generate unexpected logical combinations, for example the involuntary use of the enable in flip-flops[41]. To summarize, the synchronous reset can be slower than the asynchronous reset and consume additional logic, it requires the clock to be constantly active and it is more devious to manage at the HDL code level.

- The **asynchronous reset**, as mentioned, is natively available: it operates on the reset (or clear) input of the FDCPE flip-flops and does not consume combinational logic (unless the priority of the signals is changed, as previously described). It also does not require the clock to be active from the beginning, acting immediately in all modules that implement it.

  Unfortunately, this type of reset has a critical issue that, if left untreated, is a source of huge problems: the "*de-assert edge*". If the asynchronous reset is released (i.e. returned to the inactive state) near a clock transition, the output of some flip-flops may end up in a *metastable state* (see below) causing the reset to be released on different clock edges for the different part of the design. This would nullify the reset operation, making the circuit malfunctioning in totally random ways (and the debugging difficult).

  As example, consider a state machine: if the flip-flops containing the status bits were to enter a metastable state, the state machine would most likely acquire an invalid initial state. Such inconveniences can happen more frequently than we think in real circuits: it is therefore mandatory to synchronize the "*de-assert*" edge of the reset signal with the clock, using a suitable "*reset synchronizer*"[42]. Another necessary precaution with the asynchronous reset is to filter *spurious glitches* that would cause unwanted resets: we can act on the physical circuit making it sufficiently immune to disturbances or implement a digital filtering technique to verify that the reset is active for a certain number of clock cycles before propagating it to all modules.

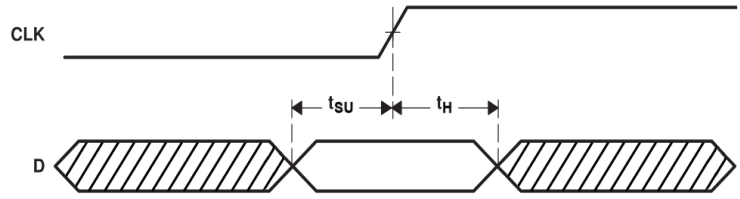
The choice of using a synchronous or asynchronous reset depends on the project, but also on individual preferences. Personally, I prefer the "active high" asynchronous reset in the definition of the modules (because it is easier to handle), and the creation of a "*master reset*" driven by a synchronizer (to eliminate the de-assert problem) and possibly a glitch filter.

41 http://www.sunburst-design.com/papers/CummingsSNUG2003Boston_Resets.pdf
42 https://www.youtube.com/watch?v=Wdzo_DpKKGA

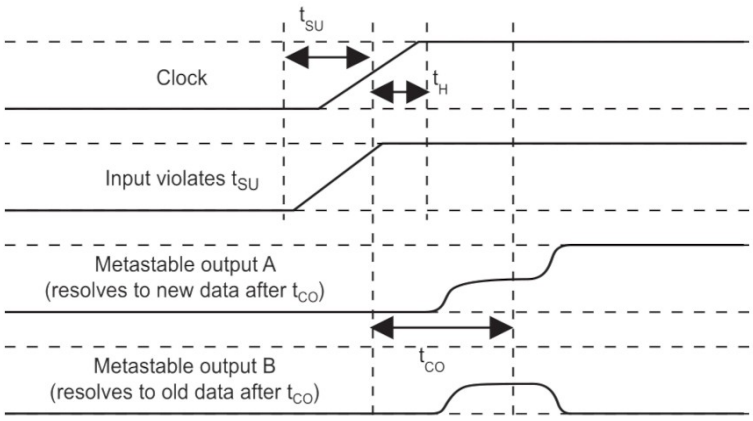
In any case, we should be consistent, i.e. do not mix the two types of reset within the same system (especially within the same module or block), unless it is desired.

Cross Domain Clock (CDC)

As seen above, to prevent an asynchronous reset signal from forcing the flip-flops into a *metastable state*, the reset must be passed through a synchronizer circuit. Actually, this is required for any signal coming from the outside, and on the interface between signals belonging to different clock domains (i.e. controlled by two clocks not linked by a constant phase relationship). Indeed, in such situations, it is not known when a rising or falling edge will occur in the input signal and if the edge occurs while the signal is acquired by a memory device, such device might enter a *metastable state*. More precisely, for physical reasons related to the operation of the flip-flop, in order for a logic signal to be correctly stored, it must remain stable at least for a certain time t_{SU} (Set-Up Time) before the arrival of the clock edge, and be kept equally stable at least for a certain time t_H (Hold Time) after the edge:



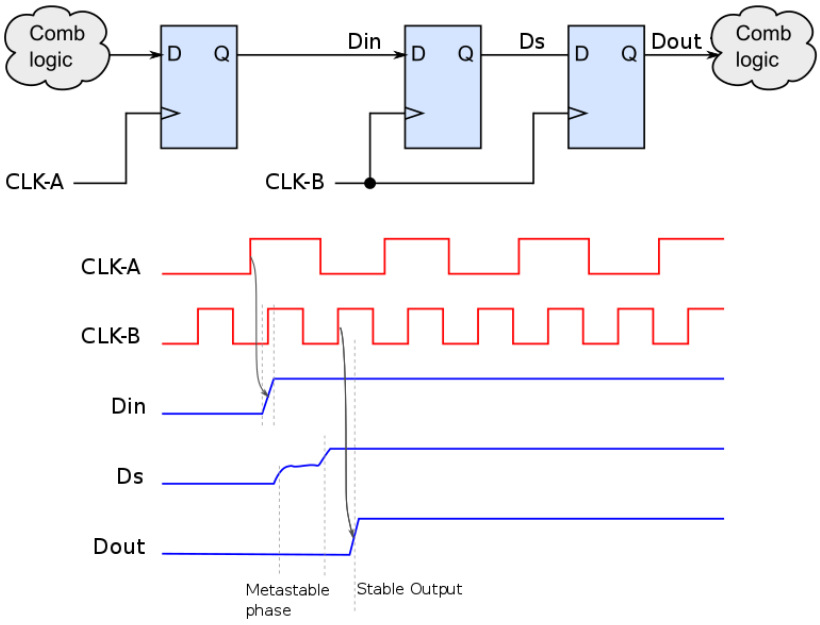
If this condition is violated, the acquired signal will assume intermediate values (metastability), possibly oscillating, and will take some time to settle upwards or downwards:



In the figure, we can see how the time required for the settling can be even higher than t_{CO} (Clock to Output delay: delay from the clock so that the output stabilizes in normal operating conditions), determining a variation on the output beyond the allowed time limit, thus creating potential difficulties for the downstream logic.

To avoid this inconvenience there are various techniques that can be adopted, depending on whether we are considering a single signal or multiple signals belonging to a bus, and whether they are slow or fast with respect to the target domain clock.

The simplest case is that of a single slow signal (in relation to the target domain clock), as in the case of a logic level coming from a button. In this case, we'll use a circuit consisting of two cascaded flip-flops, named "*two flip-flop synchronizer*" (this circuit is slightly different from the synchronizer required for the asynchronous reset, see the above linked video):



Suppose “Din” is a signal coming from a domain controlled by the “CLK-A” clock. For a second domain, controlled by the “CLK-B” clock, “Din” is an asynchronous signal. There is a good chance that, when sampling “Din”, the output “Ds” may enter a metastable state. But it is also very likely that, in a time shorter than the period of the “CLK-B” clock, the output “Ds” converges to a stable logic value and can therefore be correctly sampled by the second flip-flop (on the next clock edge) , and returned to the “Dout” output.

So, the necessary condition for this circuit to work is that the metastability condition resolves within a time slightly less than the “CLK-B” period (since t_{SU} must be taken into account). If this does not happen, an additional stage can be added. However, even with only two flip-flops the probability that the “Dout” output is also metastable is very close to zero.

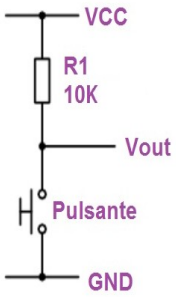
The Verilog implementation of the circuit is given by:

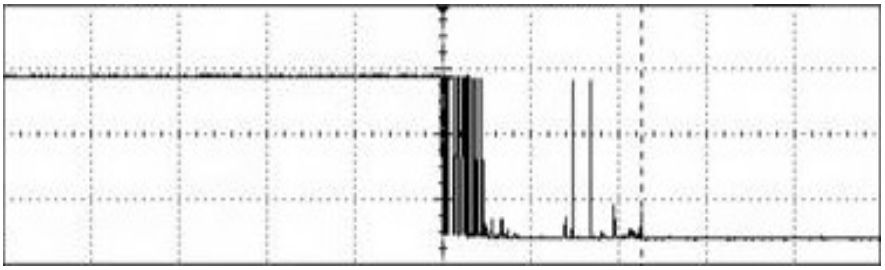
```
reg Dout, Ds;
always @ (posedge clk_b, posedge reset) begin
        if (reset)
                {Dout, Ds} <= 2'b00;
        else
                {Dout, Ds} <= {Ds, Din};
end
```

As for the other situations (bus, or fast signals), there are several techniques, all rather complex to be covered here. Please refer to the many articles and examples available on the Internet.

Debouncing

Closing or opening a mechanical contact is never instantaneous and decisive. However fast it may be, it is always affected by the rebound phenomena, short moments in which the state of the contact oscillates between open and closed. This situation is common to buttons, switches, keypads and relay contacts. The following figure shows an example of a bounce in a button:





The width of the pulses, the number of commutations and the time interval required for the signal to stabilize (which can range from a few microseconds to tens of milliseconds) depend on the type of mechanical device. It is obvious that such a signal is not only asynchronous, but must also be filtered so that the digital system registers a single commutation and not a train of pulses.

There are different filtering methods: use of RC networks, special integrated circuits (like the MAX6816 which also offers ESD protection) and in microcontrollers the use of software routines that check the contact status after a certain time from the first switching. Here we are talking about FPGAs, so we are interested in analyzing how to solve the problem through HDL synthesis.

First of all, the FPGA input signal coming from the button (connected to a pull-up resistor), being asynchronous in nature, must pass through a synchronizer. We call such input signal "*button_in*". Note that it is normally high, so the flip-flops of the synchronizer must be initialized to 1:

```
wire button_in;
reg button_sync, meta;

always @ (posedge clk, posedge reset) begin
        if (reset)
                {button_sync, meta} <= 2'b11;
        else
                {button_sync, meta} <= {meta, button_in};
end
```

The output signal from the synchronizer (here called "*button_sync*") must be now filtered to eliminate bounces. There are several solutions to this task, but the simplest way is probably to use a simple D-type flip-flop and a binary counter. Conceptually, the flip-flop is used to detect a bounce in the signal, while the counter is used to keep track of the time elapsed since the last bounce: after a well defined amount of time has elapsed without a bounce, we can assume that the signal has stabilized. Practically:

- at each positive edge of the clock the "button_sync" signal is stored in the flip-flop;
- on the same positive clock edge, the value of "button_sync" stored in the previous cycle (here called "button_sync_old"), is compared with the current value of "button_sync":
    - if the values of "button_sync" and "button_sync_old" are different, we can assume that a bounce has occurred: in this case the counter must be reset (to start from scratch in keeping track of the passage of time);

- if the values of “button_sync” and “button_sync_old” are the same, we can assume that no bounces have occurred (they are mechanical in nature, thus very slow: it is very unlikely that a double bounce will occur during one clock cycle); in this case the counter is incremented to take into account the passage of time;

- when the counter reaches the maximum preset value, it means that no bounces have occurred for the considered time interval: in this case we can consider the signal stable, and the stored value of “button_sync_old” as the filtered signal (“button_debounced”); moreover, the counter increment in this case is not necessary.

For a classic manually triggered button we can assume its output stable after 15ms from the last bounce. On the TinyFPGA BX we have a 16MHz clock and to wait 15ms we need to count 240,000 pulses: so, 18 bits are required for the counter (2^18 => 262,144).

Hence, we have the following Verilog code for the debounce section:

```
reg button_sync_old, button_debounced;
reg [17:0] counter;

// FF.
always @ (posedge clk, posedge reset) begin
	if (reset)
		button_sync_old <= 1'b1;
	else
		button_sync_old <= button_sync;
end

// Counter.
always @ (posedge clk, posedge reset) begin
	if (reset) begin
		button_debounced <= 1'b1;
		counter <= 18'b0;
	end else if (button_sync_old != button_sync)
		counter <= 18'b0;
	else if (counter == 240000)
		button_debounced <= button_sync_old;
	else
		counter <= counter + 1'b1;
end
```

Note that because the signal is active low, both “button_sync_old” and “button_debounced” must be initialized to 1.

And for the pull-up resistor we have two possibilities: physically connect a resistor between the button and the +3.3V voltage output from the corresponding pin of the TinyFPGA BX (be careful not to connect it to +5V), or use the internal pull-up resistor that the iCE40 provides on the I/O pins of banks 0, 1 and 2 (remember that bank 3 has no pull-up resistors).

The second option allows us to simplify the circuit as much as possible and eliminates the risk of using the +5V by mistake (which would damage the FPGA). As seen previously, two different strategies can be used to enable a pull-up resistor on an I/O pin:

- edit the .pcf file (adding "*-pullup yes*" to "*set_io*");
- use the "*SB_IO*" directive.

The first one is trivial. For the second, assuming that the button is connected to the "PIN_1" silkscreened pin of the TinyFPGA BX (which has pull-up, because it is connected to the FPGA's pin A2 that belongs to bank 0), just add the following code to the main Verilog module:

```
`ifndef VERILATOR
        SB_IO #(
                .PIN_TYPE( 6'b0000_01 ),        // Configure pin as simple input.
                .PULLUP( 1'b1 )                 // Enable pull-up resistor.
        ) enable_button_in_pullup (
                .PACKAGE_PIN( PIN_1 ),          // TinyFPGA BX pin name.
                .D_IN_0( button_in )            // Input signal name.
        );
`else
        assign button_in = PIN_1;               // Bypass SB_IO on Verilator.
`endif
```

Note that Verilator (which is run from "apio lint") must be bypassed because it display errors in the SB_IO directive, even though it is correct (found with versions 3.992 and 4.102).

Also note that, in the FPGA iCE40 LP family, the pull-up resistor value is fixed at 100Kohm. For some other families, such as the iCE40 UL, it is instead possible to change this value to 10Kohm using the Verilog attribute "(* PULLUP_RESISTOR = "10K" *)".

Helpful tips

Similarly to software development, hardware synthesis is profoundly affected by the quality of the code. For example, the number of instantiated logic elements and the way they are connected determine the propagation times of the signals through the components of the design and thus the performance of the overall circuit. To write good Verilog (or VHDL) code you need a good background in the field of logical networks and years of experience that cannot be summarized in these few pages. However, here are some useful tips:

- Study your FPGA and carry out some simple tests to verify that the datasheet is correct (this also allows you to get familiar with the FPGA before dealing with a project). If possible, use more than one tools and check which one synthesizes the best logic.

- Build the project one small piece at a time. In code development, don't change too many things at once and do continuous simulations (and tests on the real FPGA). If you introduce too many changes at once, and what has been achieved so far stops working, it becomes difficult to determine which change is responsible for the failure.

- Always prefer synchronous logic to the asynchronous one: the latter is less reliable and more difficult to design correctly. The opposite is true for the reset.

- Keep the same reset style throughout the design (i.e. don't mix synchronous and asynchronous reset).

- Possibly use a single clock signal throughout the project: the use of dividers or other logic to generate other clock domains is a very bad practice that causes non-homogeneous switching of sequential logic, jitter, fan-out problems (as global buffer lines are limited) and high propagation times[43]. The correct methodology is to use a single clock for all the flip-flops and registers of the FPGA, and exploit the “clock enable” inputs (that these components have) to obtain what is desired, enclosing the HDL code in conditional “if” instructions by appropriate logic[44].

- When you have complex combinational logic, always evaluate how many levels of lookup table (LUT) are required: if the propagation time through this logic becomes comparable with the clock period (or worse, if it is greater), break the logic path into two or more parts, interposing one or more registers in between (pipelining). Repeat the propagation time analysis as you build your circuit, in order to not discover timing issues too late.

- If possible, always use the same clock edge throughout the design (for example, the positive one). Using both edges may halve the maximum acceptable propagation time for the signals.

- Prefer D-type flip-flops to other types (which involve the use of additional logic).

- Always synchronize the asynchronous signals to the clock using appropriate synchronizers.

- If you are porting the design to different FPGA architectures, consider the characteristics of each family to verify that they meet the design time constraints.

- Use meaningful names in the code and be consistent in the style employed.

- Do not blindly use the available library components, understand how they fit into the project and whether or not they need to be modified.

- Take advantage of advanced components that the particular FPGA provides (multipliers, DSP units, etc.) but also maintain a corresponding generic version: this will be useful for simulation and possibly to port the design to those FPGAs that do not have such components.

Conclusion

This document ends here, its purpose is only to facilitate people approaching the fascinating world of FPGAs for the first time. For further details, please refer to the specific books and the countless examples on the Internet.

Have fun!

43 https://electronics.stackexchange.com/questions/222972/advantage-of-clock-enable-over-clock-division
44 https://www.fpga4student.com/2017/08/how-to-generate-clock-enable-signal.html

7 References

Reference Guide
https://tinyfpga.com/bx/guide.html

Schematics/KiCad PCBs, bootloaders, examples
https://github.com/tinyfpga/TinyFPGA-BX
https://github.com/mattvenn/TinyFPGA-BX
NOTE: The two repositories differ only for some files in the examples/picosoc directory.

Forum:
https://discourse.tinyfpga.com/

FPGA iCE40LP8K-CM81:
http://www.latticesemi.com/Products/FPGAandCPLD/iCE40.aspx

Logic libraries:
http://www.latticesemi.com/solutionsearch?qprod=9e00ea3e2f6e43b2bb3f9430fc476b47&qiptype=3614c818569f4eecb0602ba20a521a41,6da9534f318a4969a6b5e7dc9081bdba

Apio User's Guide
https://apiodoc.readthedocs.io/en/stable/source/user_guide/index.html

Verilog Testbench
https://www.fpgatutorial.com/how-to-write-a-basic-verilog-testbench/

Other development tools:
https://f4pga.org/
https://github.com/FPGAwars/icestudio
https://github.com/rochus-keller/QtcVerilog
https://github.com/YosysHQ/oss-cad-suite-build

RISC-V PicoRV32
https://github.com/YosysHQ/picorv32
https://libraries.io/github/kionix/picorv32

Index