Introduction to FPGA design

J. Serrano
CERN, Geneva, Switzerland

Abstract
This paper presents an introduction to digital hardware design using Field Programmable Gate Arrays (FPGAs). After a historical introduction and a quick overview of digital design, the internal structure of a generic FPGA is discussed. We then describe the design flow, i.e., the steps needed to go from design idea to actual working hardware. Digital signal processing is an important area where FPGAs have found many applications in recent years. Therefore a complete section is devoted to this subject. The paper finishes with a discussion of important peripheral concepts essential for success in any project involving FPGAs.

1 Historical introduction

Digital electronics is concerned with circuits which represent information using a finite set of output states [1]. Most of the applications use in fact just two states, which are often labelled '0' and '1'. Behind this choice is the fact that the whole Boolean formalism then becomes available for the solution of logic problems, and also that arithmetic using binary representations of numbers is a very mature field.

Different mappings between the two states and the corresponding output voltages or currents define different logic families. For example, the Transistor–Transistor Logic (TTL) family defines an output as logic '1' if its voltage is above a certain threshold (typically 2.4 V). For the same family, if we set the input threshold for logic '1' as 2 V, we will have a margin of 0.4 V which will allow us to interconnect TTL chips inside a design without the risk of misinterpretation of logic states. This complete preservation of information even in the presence of moderate amounts of noise is what has driven a steady shift of paradigm from analog to digital in many applications. Here we see as well another reason for the choice of binary logic: from a purely electrical point of view, having only two different values for the voltages or currents used to represent states is the safest choice in terms of design margins.

Historically, TTL chips from the 74 series fuelled an initial wave of digital system designs in the 1970s. From this seed, we shall focus on the separate branches that evolved to satisfy the demand for programmability of different logic functions. By programmability, we mean the ability of a designer to affect the logic behaviour of a chip after it has been produced in the factory.

A first improvement in the direction of programmability came with the introduction of gate arrays, which were nothing else than a chip filled with NAND gates that the designer could interconnect as needed to generate any logic function he desired. This interconnection had to happen at the chip design stage, i.e., *before* production, but it was already a convenient improvement over designing everything from scratch. We had to wait until the introduction of Programmable Logic Arrays (PLAs) in the 1980s to have a really programmable solution. These were two-level AND-OR structures with user-programmable connections. Programmable Array Logic (PAL) devices were an improvement in performance and cost over the PLA structure. Today, these devices are collectively called Programmable Logic Devices (PLDs).

The next stage in sophistication resulted in Complex PLDs (CPLDs), which were nothing else than a collection of multiple PLDs with programmable interconnections. FPGAs, in turn, contain a much larger number of simpler blocks with the attendant increase in interconnect logic, which in fact dominates the entire chip.

2 Basics of digital design

A typical logic design inside an FPGA is made of combinatorial logic blocks sandwiched in between arrays of flip-flops, as depicted in Fig. 1. A combinatorial block is any digital sub-circuit in which the current state of the outputs only depends, within the electrical propagation time, on the current state of the inputs. To this group belong all the well-known basic logic functions such as the two-input AND, OR and any combination of them. It should be noted, that logic functions of arbitrary complexity can be derived from these basic blocks. Multiplexers, encoders and decoders are all examples of combinatorial blocks. The input in Fig. 1 might be made of many bits. The circuit is also supplied with a clock, which is a simple square wave oscillating at a certain fixed frequency. The two flip-flops in the circuit, which might be flip-flop blocks in the case of a multi-bit input, are fed with the same clock and propagate the signals from their D inputs to their Q outputs every time there is a rising edge in the clock signal. Apart from that very specific instant in time, D is disconnected from Q.

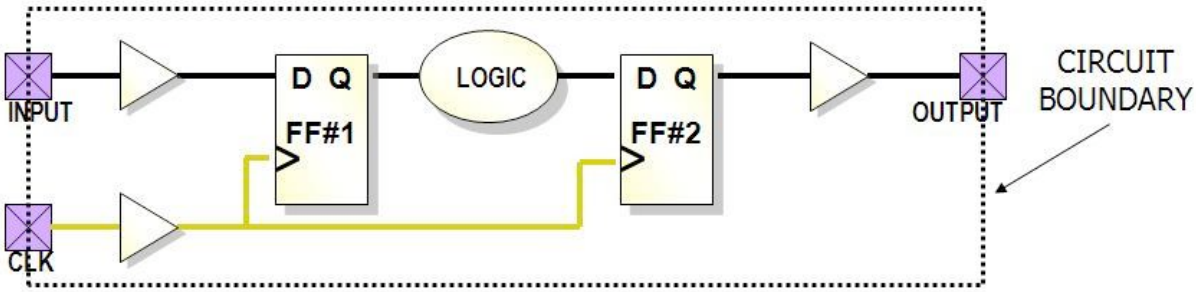

Fig. 1: A typical digital design

The structure of the circuit is thus very simple, and its application as a template covers the vast majority of digital design requirements in standard applications, such as the control of particle accelerators. The designer only needs to ensure that the worst-case propagation delay between any of the inputs to the combinatorial logic block and any of its outputs is less than one clock period. If that condition is satisfied, the inputs to the second stage of flip-flops will be stable by the time the next clock edge reaches them. As we shall see later, the process of ensuring this timing closure is nowadays automated, so the designer need only be concerned with the specification of the logic behaviour of the circuit.

The main merit of this design strategy, called *synchronous design*, is that the analysis of the possible timing failures and race conditions is greatly simplified. One might design otherwise, feeding for example the clock input of a flip-flop from the output of a combinatorial clock, in what is called a 'gated clock' circuit, and that design might give good results during simulation. But then, slight changes in the different delays of the signals might result in circuit malfunction. Slight changes of delays in the synchronous design paradigm can be easily accommodated by taking some safety margin in the maximum allowed time for any combinatorial signal path. This simple recipe contrasts with the 'many-body problem' of examining effects of delay changes in asynchronous designs.

3 FPGA structure

A typical modern FPGA (see Fig. 2) provides the designer with programmable logic blocks that contain the pool of combinatorial blocks and flip-flops to be used in the design. In addition, vendors acknowledge the fact that logic is often used in conjunction with memory, and typically include variable amounts of static Random Access Memory (RAM) inside their chips. Clock conditioning has also become commonplace, and support in the form of Delay Locked Loops (DLLs) and Phase Locked Loops (PLLs) is also provided inside the same silicon chip. Finally, an FPGA chip does not lead a solitary life isolated from the rest of the world. It needs to be easily interfaced to other chips or external signals. In order to make this interfacing easier, FPGA vendors have invested a great deal of effort in enhancing the flexibility of the input/output blocks behind the chip pads. Each pad can serve as an input, an output, or both. The list of electrical standards supported is extensive, and novel techniques for maximizing bandwidth, such as clocking data in using both edges of the clock, are widely supported.

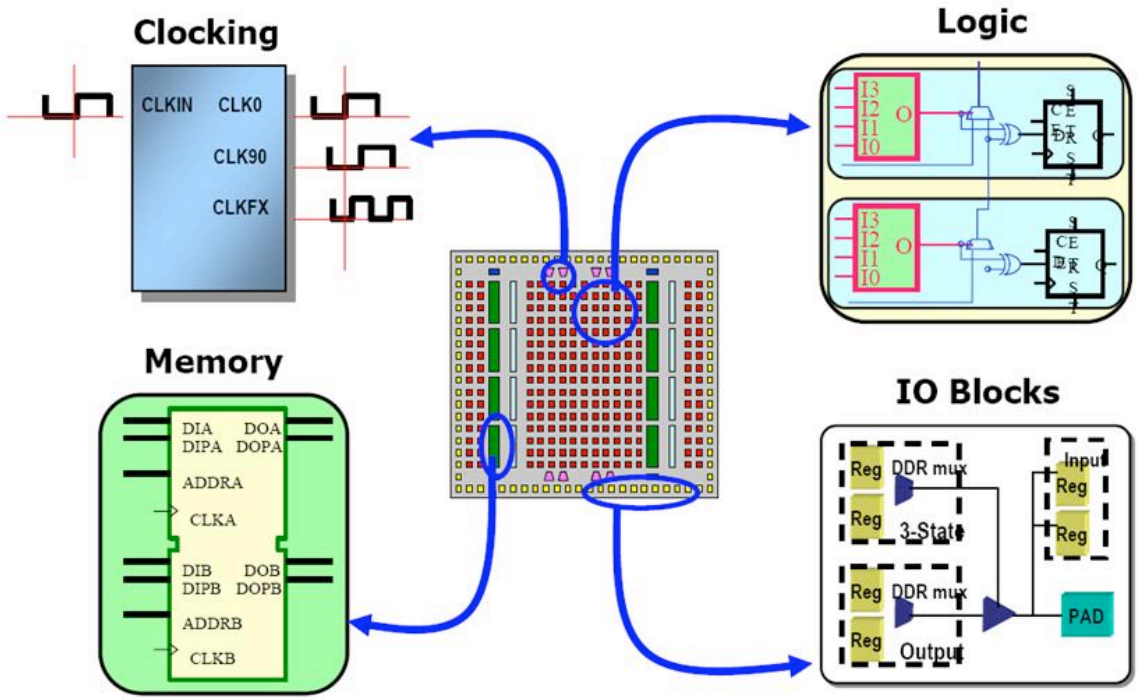

Fig. 2: Internal structure of a generic FPGA (courtesy Xilinx, Inc.)

All the components shown in Fig. 2, however, typically account for less than 20% of the silicon inside an FPGA chip. What is not shown is the large amounts of programmable interconnect and the auxiliary circuits which 'program' the generic blocks to become a well-defined piece of logic. This silicon inefficiency is the price to pay for programmability, and is also the reason why FPGAs have traditionally been more successful in high-end, low-volume applications in the past, with Application-Specific Integrated Circuits (ASICs) taking a leading role for high-volume applications. With Moore's law, however, the line between high-end and low-end applications is continuously shifting, and FPGAs are more and more used in domains which used to be dominated by ASICs and Digital Signal Processors (DSPs).

To overcome the silicon inefficiency problem, FPGA vendors often include hardwired Intellectual Property (IP) cores inside the chips for functions identified as recurrent in many designs. These non-programmable blocks include general-purpose processors, high-speed serial interfaces, arithmetic blocks and Ethernet Medium Access Control (MAC) units.

4 Design flows

The designer facing a design problem must go through a series of steps between initial ideas and final hardware. This series of steps is commonly referred to as the 'design flow'. First, after all the requirements have been spelled out, a proper digital design phase must be carried out. It should be stressed that the tools supplied by the different FPGA vendors to target their chips do not help the designer in this phase. They only enter the scene once the designer is ready to translate a given design into working hardware.

The most common flow nowadays used in the design of FPGAs involves the following subsequent phases:

- Design entry. This step consists in transforming the design ideas into some form of computerized representation. This is most commonly accomplished using Hardware Description Languages (HDLs). The two most popular HDLs are Verilog and the Very High Speed Integrated Circuit HDL (VHDL) [2]. It should be noted that an HDL, as its name implies, is only a tool to *describe* a design that pre-existed in the mind, notes, and sketches of a designer. It is not a tool to design electronic circuits. Another point to note is that HDLs differ from conventional software programming languages in the sense that they don't support the concept of sequential execution of statements in the code. This is easy to understand if one considers the alternative schematic representation of an HDL file: what one sees in the upper part of the schematic cannot be said to happen before or after what one sees in the lower part.
- Synthesis. The synthesis tool receives HDL and a choice of FPGA vendor and model. From these two pieces of information, it generates a netlist which uses the primitives proposed by the vendor in order to satisfy the logic behaviour specified in the HDL files. Most synthesis tools go through additional steps such as logic optimization, register load balancing, and other techniques to enhance timing performance, so the resulting netlist can be regarded as a very efficient implementation of the HDL design.
- Place and route. The placer takes the synthesized netlist and chooses a place for each of the primitives inside the chip. The router's task is then to interconnect all these primitives together satisfying the timing constraints. The most obvious constraint for a design is the frequency of the system clock, but there are more involved constraints one can impose on a design using the software packages supported by the vendors.
- Bit stream generation. FPGAs are typically configured at power-up time from some sort of external permanent storage device, typically a flash memory. Once the place and route process is finished, the resulting choices for the configuration of each programmable element in the FPGA chip, be it logic or interconnect, must be stored in a file to program the flash.

Of these four phases, only the first one is human-labour intensive. Somebody has to type in the HDL code, which can be tedious and error-prone for complicated designs involving, for example, lots of digital signal processing. This is the reason for the appearance, in recent years, of alternative flows which include a preliminary phase in which the user can draw blocks at a higher level of abstraction and rely on the software tool for the generation of the HDL. Some of these tools also include the capability of simulating blocks which will become HDLs with other blocks which provide stimuli and processing to make the simulation output easier to interpret. The concept of hardware co-simulation is also becoming widely used. In co-simulation, stimuli are sent to a running FPGA hosting the design to be tested and the outputs of the design are sent back to a computer for display (typically through a Joint Test Action Group (JTAG), or Ethernet connection). The advantage of co-simulation is that one is testing the real system, therefore suppressing all possible misinterpretations present in a pure simulator. In other cases, co-simulation may be the only way to simulate a complex design in a reasonable amount of time.

5 Digital signal processing using FPGAs

Clearly the main advantage of FPGAs over conventional DSPs to perform digital signal processing is their capability to exploit parallelism, i.e., replication of hardware functions that operate concurrently in different parts of the chip. Figure 3 shows how a Finite Impulse Response (FIR) filter could be implemented in both platforms. While the DSP needs 256 clock ticks to calculate an output sample, the FPGA generates a new sample for every clock cycle. Even if DSP chips can be clocked faster than FPGAs, the difference is in no case larger than a factor of 10. If one adds that many such filters can exist concurrently and interact inside the same FPGA, it is easy to see that DSPs are no match for FPGAs in high-performance signal processing applications [3].

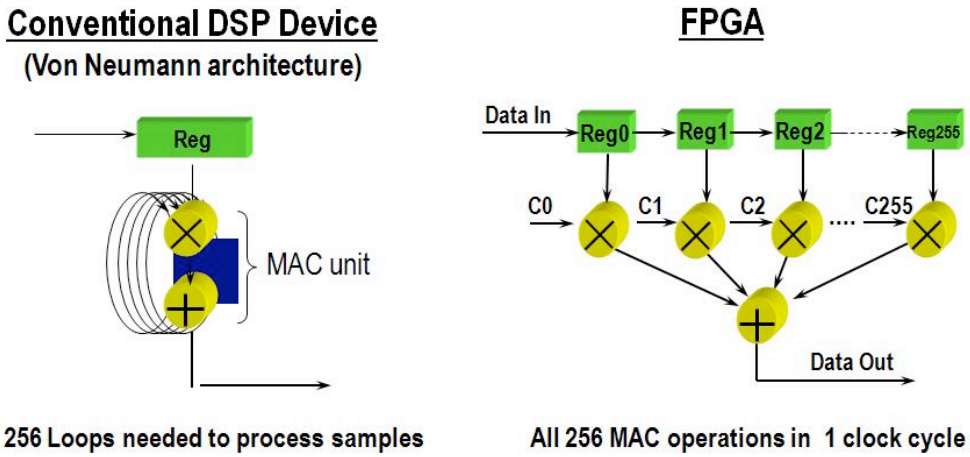

Fig. 3: FIR filter comparison between DSP and FPGA

Another advantage of FPGAs is the flexibility for trading off between area and speed until very late in the design cycle. Figure 4 shows three different implementations of a sum of products, from a full expansion using more silicon to a minimalist implementation which takes more clock cycles to generate a result.

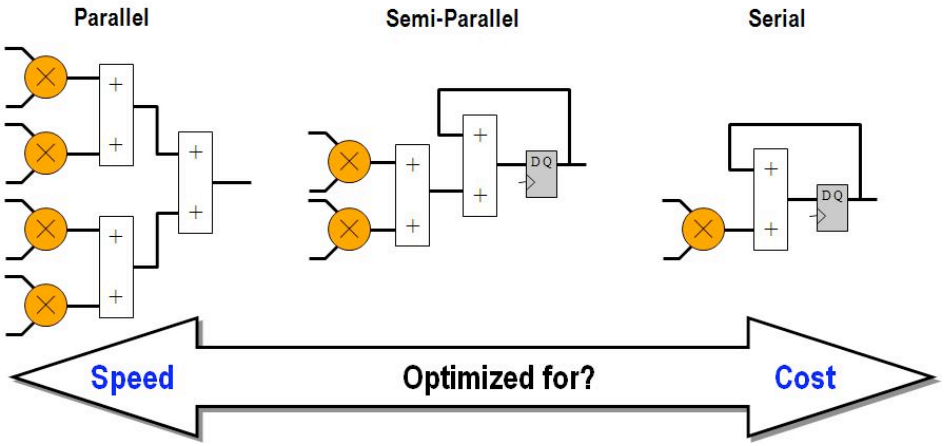

Fig. 4: Illustrating the speed/area trade-off in FPGAs

In this section, we give a basic introduction to fixed-point arithmetic and then touch briefly upon two interesting techniques for processing fixed-point signals: distributed arithmetic and the COordinate Rotation DIgital Computer (CORDIC) algorithm. The interested reader will find more details in the references at the end of this document.

5.1 Fixed-point arithmetic

In FPGA design, one typically uses a two's complement fixed-point representation of numbers. Floating point design is perfectly feasible, but the high-performance applications typically targeted by FPGAs can very often be served adequately by using enough bits in a fixed-point representation. This is another advantage of FPGAs: the possibility to tailor the bus widths in different parts of the design to satisfy the final precision requirements. Figure 5 shows an example of fixed-point two's complement representation, where we have taken three of the bits for the integer part and five for the fractional part. In reality, as we shall see, an adder or any other arithmetic circuit does not know about our decision on how many bits to interpret as fractional. This is purely an interpretational issue, so for all practical purposes, one can think of fixed-point arithmetic as integer arithmetic.

digit worth								decimal value
$-(2^2)$	2^1	2^0 •	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	
-4	**2**	**1** •	**0.5**	**0.25**	**0.125**	**0.0625**	**0.03125**	
0	0	0 •	0	0	0	0	1	0.03125
0	0	0 •	0	0	0	1	0	0.0625
1	0	1 •	0	0	0	0	0	-3.0
1	1	0 •	0	0	1	1	1	-1.78125
1	1	1 •	1	1	1	1	1	-0.03125

Fig. 5: Fixed-point two's complement representation of signed numbers using three integer bits and five fractional bits

As an introductory example, let us see how one can make a circuit that performs simple addition or subtraction using logic gates. This will be a combinatorial circuit with nine inputs (say four per input and one for controlling if we add or subtract) and five outputs (the final result). Notice that we need to have one extra bit in the output because the addition/subtraction of two 4-bit numbers can result in a 5-bit number.

We start with the full adder circuit of Fig. 6. It is easy to see that this circuit made of AND, OR and XOR gates takes two bits and a carry (maybe from a preceding stage) and generates a proper sum and carry out.

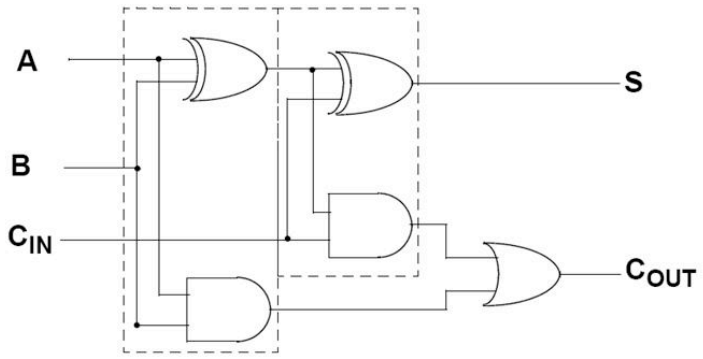

Fig. 6: A full adder circuit

Out of many of these Full Added (FA) cells, one can build the circuit of Fig. 7, which takes two 4-bit signed numbers and adds them together if the Control signal is '0'. Otherwise, a subtraction is performed. Although these days one would generate this circuit with a simple HDL statement, it is enlightening to understand what the underlying hardware looks like. More complicated blocks to multiply, divide, take a square root, etc. can be synthesized using these basic blocks and the reader is referred to the specialized literature for details [3].

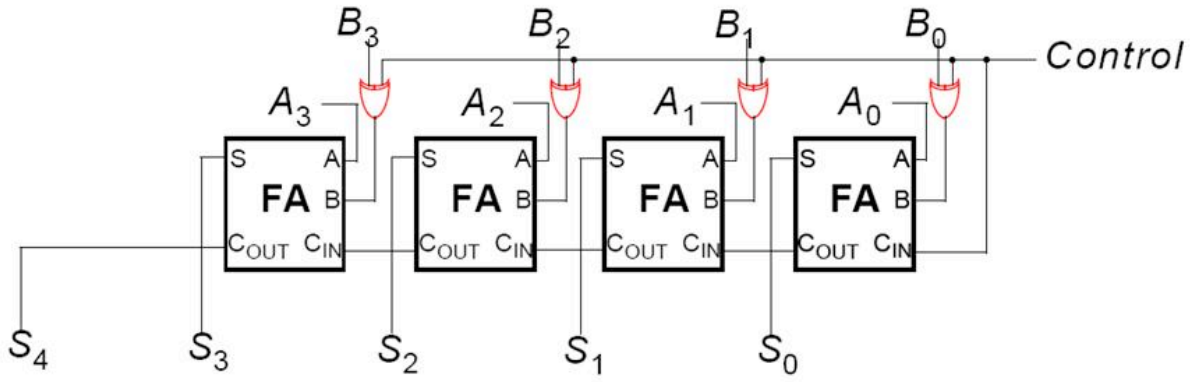

Fig. 7: 4-bit full add/subtract

One question that arises immediately is what to do with this arithmetic bus that gets wider and wider as we cascade more and more operations one after the other. At some point, the bus width will become inconveniently large. We will be wasting bits with unnecessary information, and our timing constraints could be compromised as the combinatorial paths traverse more layers of logic before hitting the next flip-flop. One example could be a feedback system that generates an analog signal towards a Radio Frequency (RF) cavity using a 16-bit Digital to Analog Converter (DAC) fed by an FPGA. There would be no point in keeping an internal representation of say 50 bits and then collapse it all at the very end to feed the DAC with only 16 bits. The solution to this problem is to control the width after each operation by judiciously choosing a suitable number of bits to represent the intermediate results. Figure 8 shows two ways of doing this: truncation and rounding. In truncation, some of the fractional bits are taken out before feeding the result to the next stage. In rounding, a '1' is added to the most significant bit (of the ones to be taken out) before truncation.

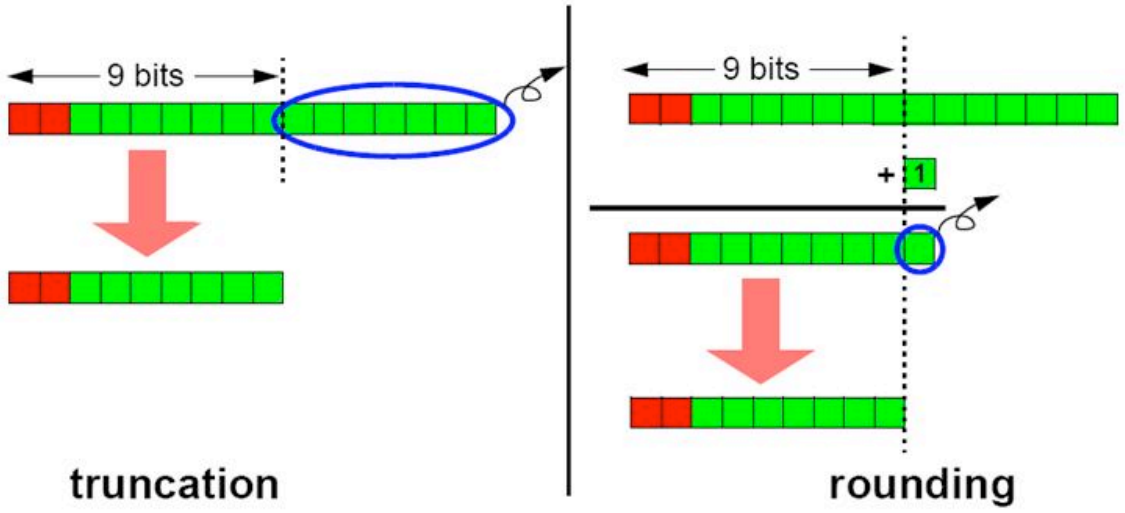

Fig. 8: Truncation vs. rounding in fixed-point representation

Notice that in two's complement, truncation is a biased operation. The output of truncation will always be a smaller number than the input. If an unbiased scheme is needed, then rounding should be used at the expense of the extra addition involved. The loss in precision incurred by taking some of the bits out can be studied statistically by modelling rounding or truncation as a source of white noise with an amplitude dependent on the number of bits eliminated [4].

5.2 Distributed arithmetic

Digital signal processing is all about sums of products. For example, if a generic filter is fed with an input sequence $x[n]$, we can write its output as

$$y = \sum_{n=0}^{N-1} c[n] \cdot x[n] \tag{1}$$

where $c[n]$ are the filter coefficients. If these coefficients are constant, and assuming the input signal to be B bits wide, we can rearrange the terms in a sequence that will end up suggesting an alternative hardware implementation. We begin by re-writing Eq. (1) as

$$y = \sum_{n=0}^{N-1} \left(c[n] \cdot \sum_{b=0}^{B-1} x_b[n] \cdot 2^b \right) \tag{2}$$

where $x_b[n]$ is bit number b of $x[n]$, i.e., either '0' or '1'. After rearranging:

$$y = \sum_{b=0}^{B-1} 2^b \cdot \left(\sum_{n=0}^{N-1} c[n] \cdot x_b[n] \right) \tag{3}$$

and the term in parentheses can be implemented as a Look Up Table (LUT) with N inputs, as suggested in Fig. 9.

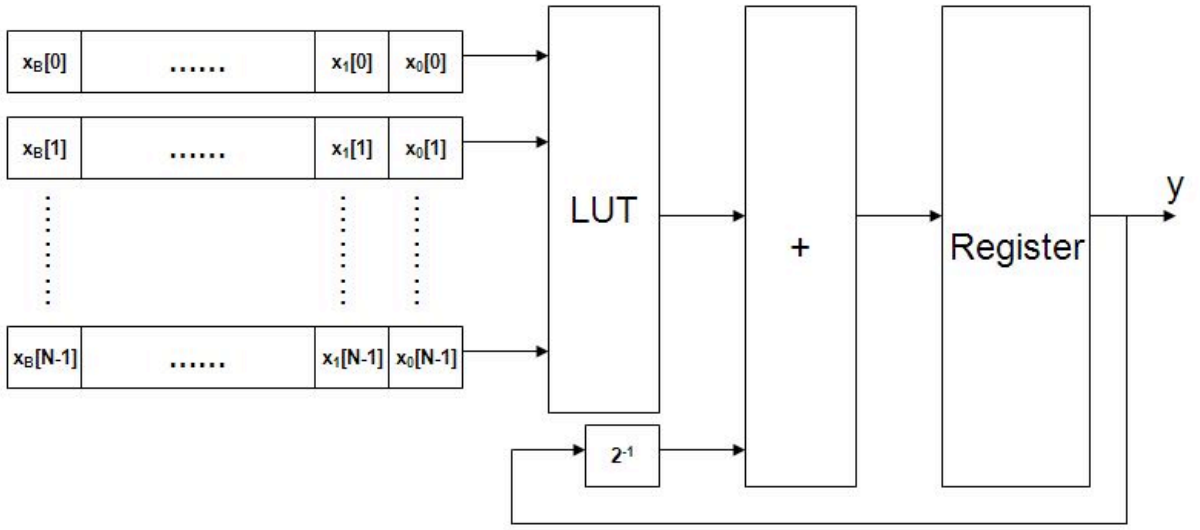

Fig. 9: Distributed arithmetic implementation of a filter

The filter implemented in this way has no need of hardware multipliers, and generates a result every B ticks, independent of the filter length N. By increasing the number of LUTs and replicating hardware, one can trade off latency versus area. The extreme case would be full parallelization: replicating B times to get one output sample per clock tick.

5.3 The CORDIC algorithm

The CORDIC is a circuit that iteratively rotates an input vector $(x^{(1)}, y^{(1)})$ and generates an output vector $(x^{(2)}, y^{(2)})$, where x and y are the Cartesian coordinates of the vectors. There are two modes of operation. In rotation mode, an angle accumulator is set to the desired rotation angle, and the CORDIC approximates that angle by performing elementary rotations of decreasing angles. The output is the input vector rotated by the specified angle. In vectoring mode, the CORDIC block rotates the input vector using the same table of decreasing angles until the resulting vector is aligned with the

horizontal axis. In this mode, the result is the angle accumulated throughout the whole rotation process. The trick in the CORDIC algorithm is to constrain the set of angles to those whose tangent can be expressed as 2^{-i}, i being the iteration index. Then the rotation operations for these angles do not need any specific multipliers, since a multiplication by 2^{-i} is just a right-shift by i places. This produces a very efficient hardware implementation in terms of area and speed. Each iteration generates roughly an extra bit of precision in the result. Among the several things one can calculate with a CORDIC, we can highlight the following:

– Magnitude of a vector: it is found on the x of the output vector after operating the CORDIC in vectoring mode.

– Sine and cosine of an angle: found by feeding an input vector with $x = 1$, $y = 0$ and setting the CORDIC to work in rotation mode with the specified angle.

More uses of the CORDIC as well as a detailed description on its internal features and ways to accelerate it can be found in Ref. [5].

6 FPGAs in real-world designs

This section is devoted to design aspects which are encountered in real projects. The FPGA designer will find none of these problems while simulating behavioural HDL in a computer, but they are paramount for the success of any FPGA project.

6.1 Performance-boosting techniques

We have already discussed the benefits of synchronous design. The place-and-route tool will analyse the timing constraints and optimize the placement and routing in such a way as to meet these constraints. But what if it cannot? If the delays due to propagation through individual gates are already higher than the specified clock period, there is nothing the tool can do to meet the specification. Remember, the tool's degrees of freedom are just related to where to place blocks and how to interconnect them. The interconnect delay will never be less than 0, and it has to be added to gate propagation delays which are fully determined by the synthesized netlist. So if the gate propagation delays already exceed the specified clock period, it's mission impossible for the place-and-route tool.

Delays in modern designs can be as much as 90% due to routing and 10% due to logic. The routing bit is due to long routes and capacitive loading on the nets. Many synthesis tools automatically insert buffers in some nets to provide more current to drive the capacitive loads, therefore decreasing routing delay, as depicted in Fig. 10.

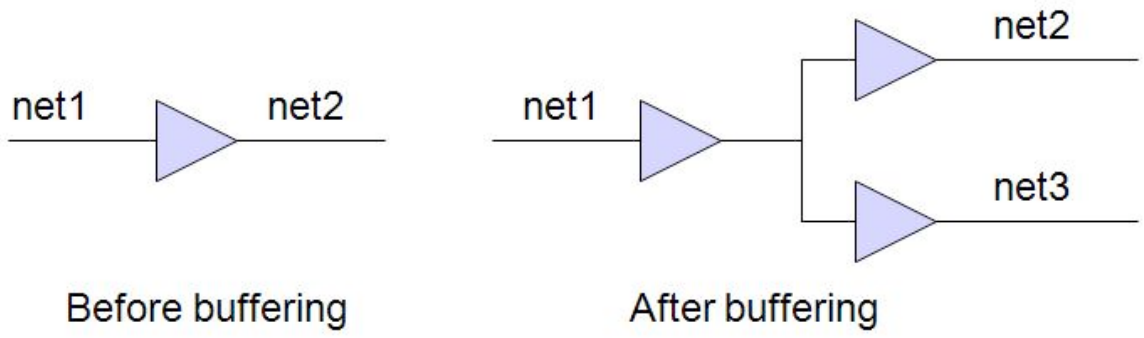

Fig. 10: Automatic buffer insertion example

The automatic replication of registers is another useful technique. This can be set as an option for those synthesis tools that support it, or it can be done by hand at the HDL level. Figure 11 illustrates the principle. The nets coming out of the flip-flop after the producer are going to four different destinations, potentially covering great lengths inside the chip. After the flip-flop is duplicated, each of the outputs only has to serve two destinations, so the timing constraints become easier. If there were combinatorial logic after the first flip-flop, it would also be replicated. The HDL specification is therefore fully respected.

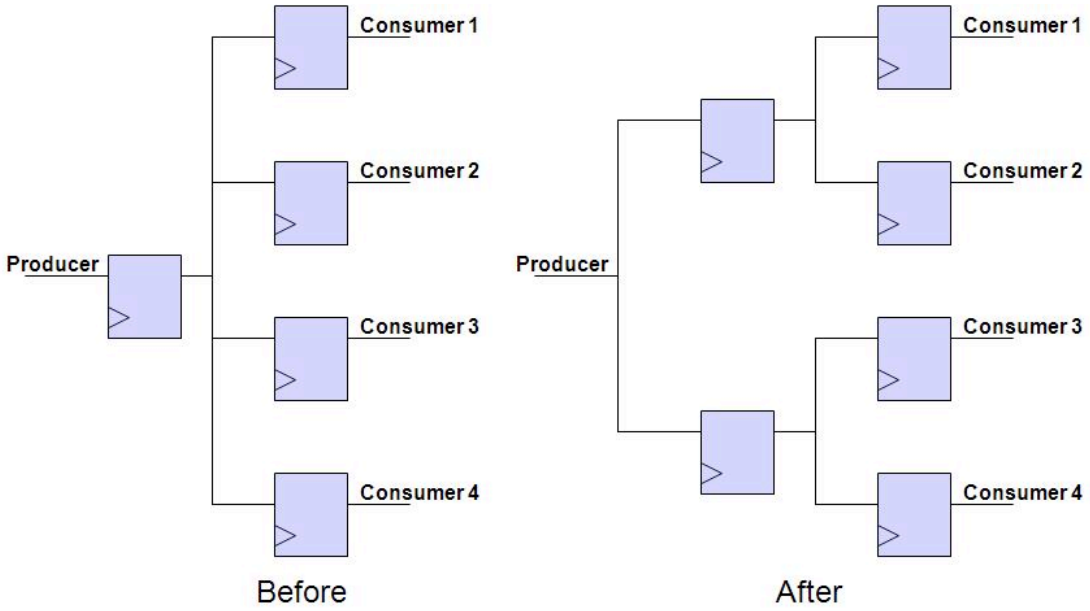

Fig. 11: Automatic replication of registers

Another problem case concerns long combinatorial delays between flip-flop stages. As we said earlier, there is nothing that the place-and-route tool can do in this case. The solution must come from the synthesis tool or the designer. Retiming — also known as register balancing — is a technique that can be used in these cases. Figure 12 shows how it works. Some of the combinatorial logic is passed to the next stage in the pipeline so that the maximum delay in each stage remains within the specifications.

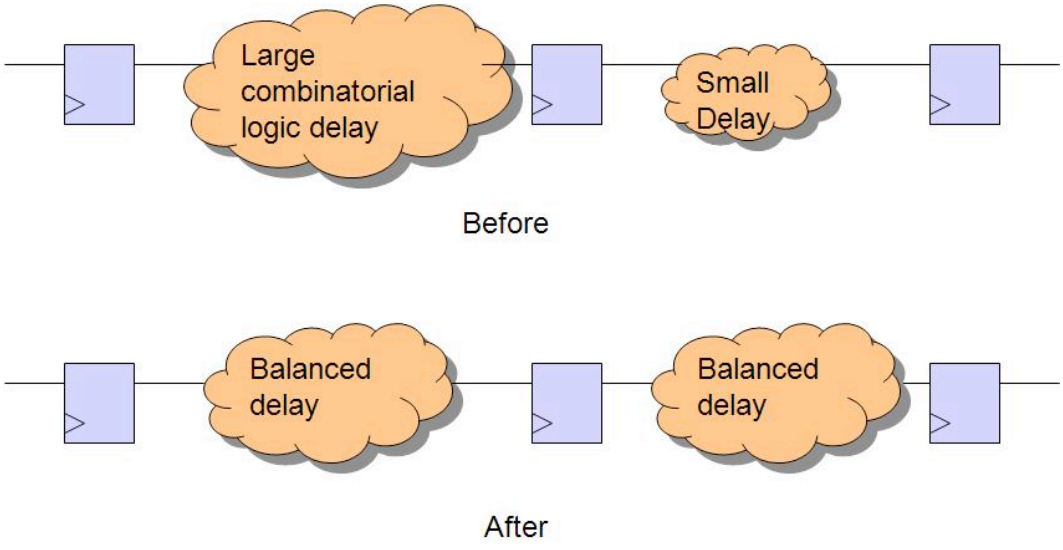

Fig. 12: Retiming

If retiming is not possible, one can always try pipelining, provided the circuit is not sensitive to the latency in the extra number of clock cycles. The principle is explained in Fig. 13. It consists in breaking up the large combinatorial delay by inserting flip-flop stages after intermediate results. In this case, it is better to modify the original design rather than using the synthesis tool, since it could lead to an incoherency between HDL sources and final hardware.

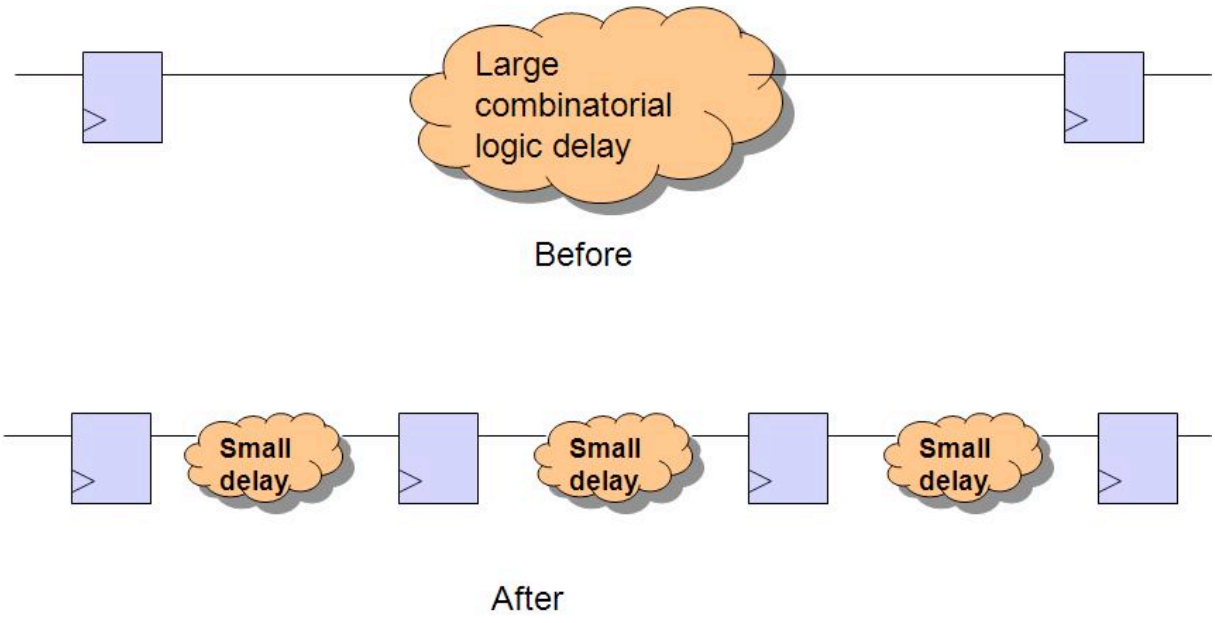

Fig. 13: Pipelining

Finally, time-multiplexing in conjunction with hardware replication can also be a powerful tool to prevent timing pathologies. The principle, depicted in Fig. 14, consists in splitting a data path in two, making each branch work at half the speed, and recombining the results at the end to regenerate a data flow at the design clock frequency.

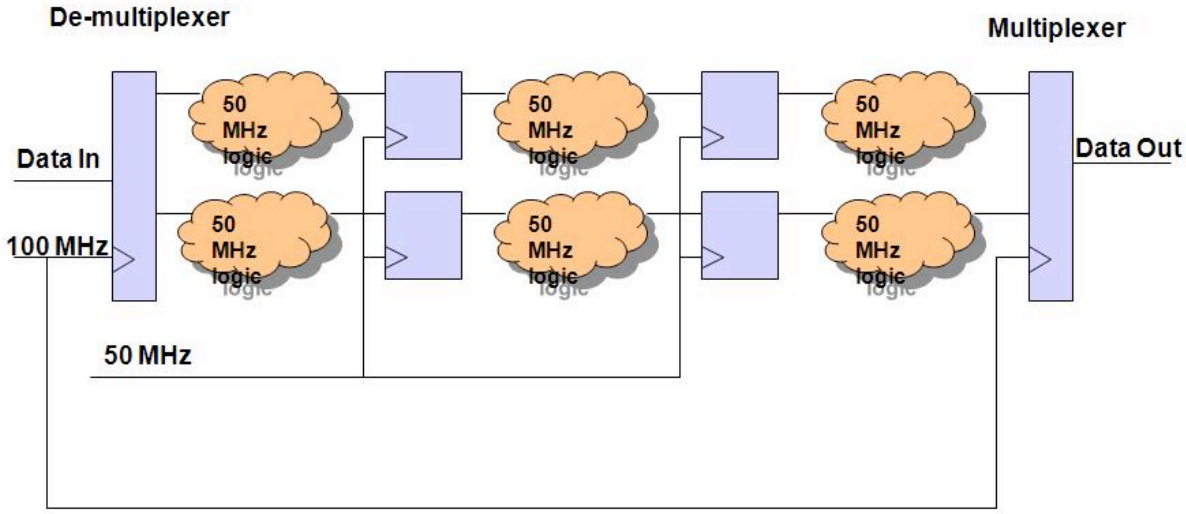

Fig. 14: Time-multiplexing

As an example of how these tools can be used in practical cases, let us examine a performance problem that arose in the phase filter of a PLL used to track bunch frequency in CERN's PS. Figure 15 shows the original filter design, a first-order Infinite Impulse Response (IIR) filter implementing the transfer function $y[n+1] = ay[n] + bx[n]$. Signal y goes back to the output flip-flop through a multiplier

and an adder, and these combinatorial delays are not compatible with the clock frequency. What can we do?

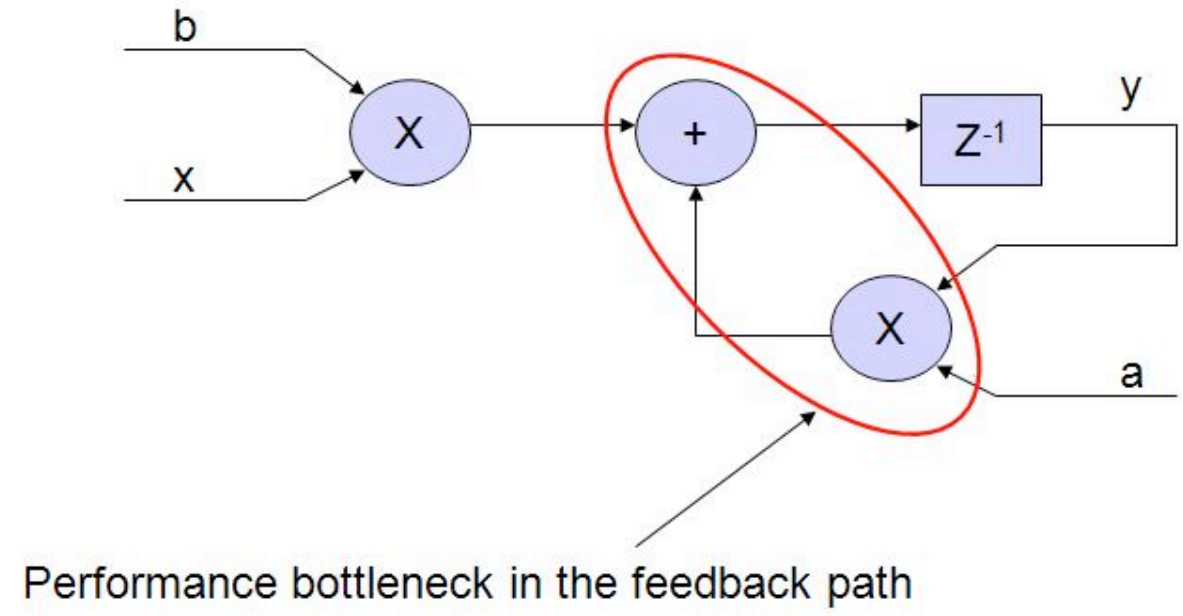

Fig. 15: A simple IIR filter with a performance problem

We can calculate $y[n+2] = ay[n+1] + bx[n+1] = a^2y[n] + abx[n] + bx[n+1]$, and see what the resulting direct implementation would look like in Fig. 16.

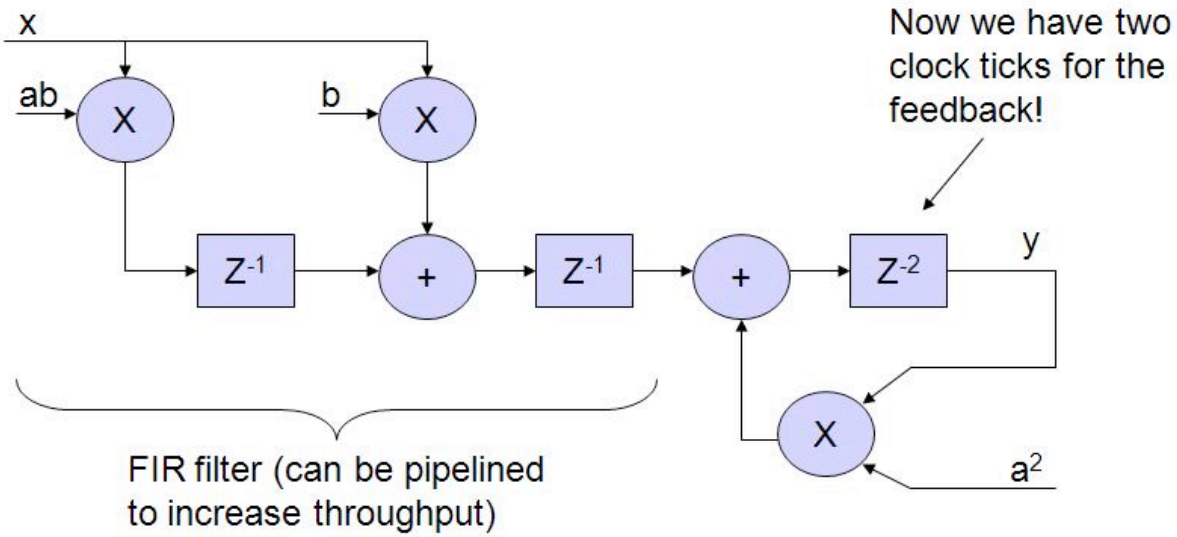

Fig. 16: Look-ahead scheme for IIR

The circuit now looks much more favourable for timing improvements. The leftmost part looks like an FIR and can be pipelined as much as necessary. The second part now contains two flip-flops in series in the feedback path, which can be used for retiming. The technique we used is called 'look-ahead' and is very common for boosting the speed of digital circuits.

6.2 Powering FPGAs

FPGAs are typically powered from various supply rails. They need different voltages for the internal logic, the Input/Output (I/O) circuitry, and some analog blocks like PLLs. Typical specifications

include a ±5% tolerance on the actual value of the voltage and monotonic ramping of the supplies during power-up. While it is not proven that ramping in a non-monotonic way would not work, FPGAs are not tested that way after manufacturing, so it is better to guarantee a monotonic ramp in order to avoid surprises. Devices also specify a minimum and a maximum ramping time for the voltage rails. Again, this is just how they are tested after production, and it is very wise to follow these guidelines.

An important aspect to bear in mind concerns in-rush current at power-up due to the decoupling capacitors on the power supply rails. If C is the total capacitance, $I_c = C*\Delta V/\Delta T$, so one might want to slow the ramping process down using a soft-start circuit in order to avoid the kick-in of protection mechanisms in regulators, which could in turn compromise monotonicity.

Sequencing of supply voltages, i.e., making one available, then another one and so on, was a required practice in old technologies, and nowadays it is only recommended. It seems sensible that the I/O stages get power only after the internal logic is properly configured. A Supply Voltage Supervisor (SVS) chip can be used to control the process. Sequencing is also good to make sure that the main (typically 5 V) rail feeding the regulators is well established (i.e., all capacitors charged) before they begin requesting current from it. Otherwise the 5 V protection could trip and spikes could appear in the output of the regulators.

The design of a proper bypassing network using capacitors is also a critical issue. A decoupling network should look like a short to ground for all the frequencies of power supply noise we want to reject. At high frequencies, like the ones of interest for this discussion, a capacitor chip can be modelled as an equivalent RLC circuit to take into account the various imperfections in its design. The parasitic inductance dominates at high frequencies, and is (almost) exclusively determined by the package type of the capacitor. The global frequency response presents a downward slope at low frequencies whose value depends on the capacitance, and an upward slope at high frequencies whose value depends on the parasitic inductance. The minimum of the curve thus depends on the capacitance value, and can be made arbitrarily wide by selecting a suitable set of capacitor values and placing them in parallel. High-value capacitors take care of low-frequency perturbations and can be placed relatively far away from the chip, while low values of capacitance (typically 10 nF), can be placed close to the chip — ideally below it — to take care of the fast perturbations. Reference [6] can be consulted for further details.

6.3 Interfacing to the outside world

Modern FPGAs have very versatile I/O blocks which make them easy to interface to other chips. In this section, we look in particular at issues which could appear when interfacing to Analog to Digital Converters (ADCs) or DACs.

Whenever a design deals with high-speed, high-pin-count parallel busses, as is the case often when interfacing FPGAs and ADCs/DACs, there is potential for noise problems. This is because the I/O drivers in the FPGAs commute state all at the same time, creating large current surges in the Power Distribution System (PDS). The PDS should be well decoupled using the appropriate mix of capacitors as discussed above, but it cannot filter all the noise at all frequencies. In addition, sampling many bits at a high frequency can pose synchronization problems. If the clock edge is very close to the transition of any of the data bits, a problem known as metastability — to be explained later — can arise. It is therefore desirable to avoid simultaneous fast-switching of large busses if possible. One example where this is possible is in the sampling of high frequency, low bandwidth analog signals. According to sampling theory, there is no need to sample them in their main Nyquist zone, i.e., with at least twice their frequency. It is sufficient to sample them at least faster than twice their bandwidth – which can be significantly slower. This can be a solution for systems where latency is more or less a secondary concern, but it might not be possible for feedback systems. Another possibility for mitigating noise problems is to choose ADC and DAC chips which use differential signalling for the

data bits and the clock. Currents in differential signals go in through one line and out of the other, without any net demand on the PDS. The use of differential signalling also creates negligible ground bounce. Ground bounce is caused by the fact that the impedance between the ground pins and the ground plane is not exactly zero. This can cause a perceived change in ground level, as seen by the digital chip, when it consumes significant current. This impedance has a resistive component but also a very important inductive component which will create voltage jumps as a function of d*I*/d*t*. Therefore, another way to improve the noise problem is to feed the digital outputs of the ADC with the lowest realistic supply voltage. Current generation ADCs can accept digital supplies in the range 2.0–2.5 V. In addition, current can be reduced by placing resistors in series with the ADC outputs.

Another aspect to bear in mind when designing FPGA systems is the state of I/O drivers during power-up. Most FPGA chips provide a possibility, through external jumpers, of selecting whether the I/Os will be pulled-up or tri-stated during the power-up process. The tri-state option lets the designer control the power-up state of each pin through external pull-up or pull-down resistors. This is important if glitch-free operation is requested during startup.

6.4 Clock domains and metastability

Oftentimes a designer is faced with an input signal that is not synchronized with the system clock, i.e., its rising and falling edges do not maintain a constant delay with respect to the rising edge of the clock signal. Let us imagine for example that we have a card where an FPGA is clocked by an on-board 100 MHz oscillator, and the card is fed with an external input representing the revolution frequency of a synchrotron. Let us also assume that the revolution tick has to be fed to two different state machines inside the FPGA, and that the correct functioning of this design relies on both state machines detecting the revolution tick during the *exact* same period of the system clock, which is used as the clock for the state machines.

A naïve design might split the revolution tick signal in two before feeding it to the state machines. The problem with this solution is that the revolution tick might eventually, after going through several layers of combinatorial logic, find itself at the D inputs of two different flip-flops inside the FPGA. But because the propagation delays of the revolution signal going through the two paths are different, one flip-flop might already clock it in as ‘1’ while the other still sees a ‘0’.

A less naïve designer would then propose to feed the revolution tick to the D input of a flip-flop to begin with, and only then split it in two. Indeed, the rate of failures would go down, but every now and then we would still see an incoherency between the two state machines. The culprit is an effect known as ‘metastability’ which afflicts flip-flops when a transition at their D input occurs too close in time to the rising edge in the clock input. In that case, their Q output hesitates until it finally settles to one of the two possible output values. The resolution time can be arbitrarily long as we push the two edges closer and closer in time. In our second design, from time to time the Q output of the synchronizing flip-flop will go metastable, with a resolution time such that — on the next system clock tick — one of the two subsequent flip-flops will already see a ‘1’ when the other one still sees a ‘0’.

While it seems that this could become a never-ending story, in fact, for all practical purposes, the circuit in Fig. 17 will solve the problem.

Now, for typical system clock and asynchronous input frequencies, the chances that the second flip-flop goes metastable after the first one did the same, one system clock tick earlier, are vanishingly small. One can easily design a circuit that will fail on average once every million years.

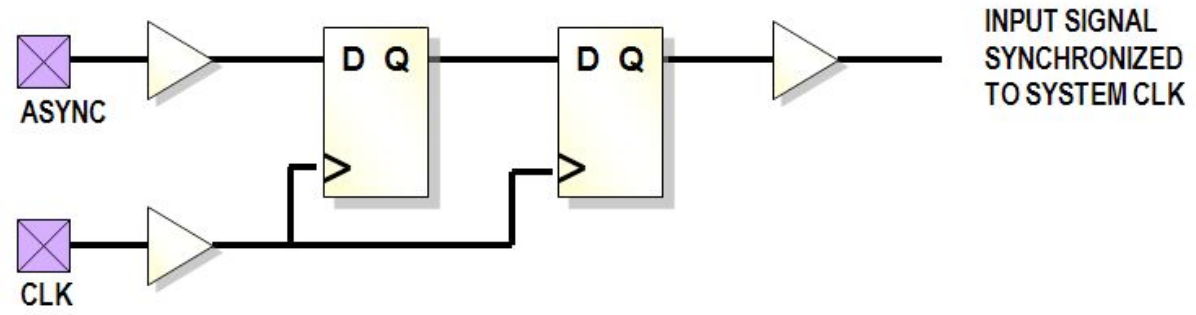

Fig. 17: Two-flip-flop synchronizer

The discussion above applies equally well to any design with more than one clock domain where data must be transferred from one domain to the other. By clock domain, we mean a part of the design which is clocked by a single common clock. Let us imagine that a designer needs to transfer a 16-bit number from one domain to another. Inserting a synchronizer for each bit would not help, since different flip-flops will see different set-up times of the data with respect to their clock. Indeed, these set-up times vary with time in a random way! The solution is to design the emitter block so that it asserts a data strobe when the data are ready, and holds the data stable for a suitable amount of time. The receiver circuit can then sample the strobe with a two-flip-flop synchronizer and clock the data in once it senses a '1' on the strobe line. A variation on this circuit includes a handshake whereby the receiver sends back an acknowledge line to the emitter, which in turn synchronizes it into its clock domain. Once the emitter senses a '1' on the acknowledge line, it knows it can change the state of the data lines to prepare the next transfer.

Sometimes the above scheme will not work because data comes in bursts at a speed which makes it impossible to perform the full handshake. In those cases, a First-In-First-Out (FIFO) block is needed with each of its sides clocked by a different clock.

6.5 Safe design

There is at least one asynchronous signal in almost all designs: the external reset. In many cases, it is very important to handle this signal properly in order to guarantee coherency of different parts of a design. If we return to the example of the two different state machines within an FPGA, both running this time off the same system clock, and we require that they both 'wake up' during the same clock tick after the de-assertion of the reset signal, we find ourselves with a need to treat the reset signal as we treated the revolution tick above, i.e., we need to feed it to a synchronizer before using it in the design.

The best reset strategy, not always possible, is to synchronize the reset to the system clock and then use it as a *synchronous* reset. This means that the reset line is treated as any other synchronous signal. It will enter some combinatorial block and affect its output, which will then be fed to the D input of a flip-flop. Chances are that the output of that combinatorial logic block will go to a predefined 'reset' state irrespective of other inputs if the reset line is active, but there is really nothing special about the reset line in this case from a topological point of view.

Things change if, for some reason like saving resources, the designer wants to use the asynchronous reset input present in all flip-flops. There is still a certain guarantee of coherency if the reset fed to these inputs has been properly synchronized in advance, but this will greatly depend on the clock period and the delay to reach each asynchronous reset input. Typical place-and-route tools will not include these paths in their timing analysis because they do not go from Q outputs to D inputs. It is really best if the asynchronous reset can be avoided altogether.

Another important topic in safe design is that of complete state coverage in state machines. If a state machine has five different states, it will need at least three signal lines to represent its current

state. But with three lines one can have eight different states, three of which will be illegal. It is the designer's responsibility to detect these states and take the state machine to a safe state if it goes to one of them. Now, how can a state machine go to an illegal state if no combination of inputs and states is supposed to take it there? The state vector is made of three lines, and each of these lines is — we can assume this without loss of generality — fed from the Q output of a flip-flop. High-energy particles crossing the FPGA can induce what is called a Single-Event Upset (SEU), flipping the state of one of the flip-flops to a new one, which might make the new three-bit combination illegal. With process geometries shrinking, one no longer needs a high-energy accelerator to produce SEUs. Atmospheric particles will produce them at a rate high enough to make it a major concern for safety-critical systems.

Sometimes in high-energy accelerator applications, an FPGA must live in a highly radioactive environment. This is a completely different game. Here are three techniques designers use frequently to make their designs more robust under such adverse conditions:

- Antifuse technology. An antifuse is an element that creates a short circuit when overheated, i.e., exactly the opposite of what a fuse does. FPGAs based on antifuse technology are inalterable at least as far as the configuration memory is concerned. But we know that, among the RAM bits of a typical FPGA, the vast majority of them are configuration bits, so antifuse technology is a major improvement in terms of resistance to radiation. The negative side is the price of antifuse chips and also their lower densities. These devices are roughly one generation behind in terms of silicon processes.
- Scrubbing. As we said, most of the FPGA RAM bits that are susceptible of being affected by a SEU are in fact configuration bits. One can read the configuration stream back repeatedly to check if there has been any corruption, and then take corrective action if needed. The device that scans the configuration bits must itself be more robust than the FPGA being checked. It is typically antifuse-based.
- Triple Mode Redundancy (TMR). This technique consists in replicating the same piece of logic three times and adding a set of voters to detect whether there was a SEU in one of the blocks. Figure 18 shows the principle for the simple example of a counter. If a mismatch is found, the losing counter will be informed and correct its value accordingly.

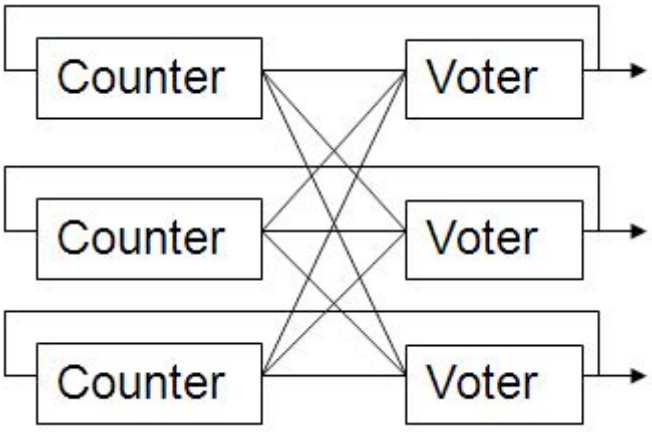

Fig. 18: TMR with state feedback

It is assumed that the probabilities to have a double upset that will affect two counters at the same time are negligible, but this may not be a very realistic assumption with the ever-diminishing process geometries. A single high-energy particle can indeed affect more than one transistor around the impact area. This is why some vendors are developing software that automatically generates TMR logic and places the different parts of the ensemble in different, geographically distant, areas of the

chip. Notice also that our FPGA now contains three counter outputs instead of one. With only one output, a SEU in one of the output transistors could defeat the whole TMR scheme. If the counter value is to be used outside the FPGA, decision logic must be implemented outside using a radiation-hard scheme in order to work out the current value of the counter.

For unmatched reliability, radiation-hard antifuse products are available. Every flip-flop in these devices is TMRed in silicon, with feedback TMR built in. Needless to say, these devices are extremely expensive and reserved for the most demanding applications in terms of radiation hardness. The extra logic needed for the TMR scheme and the state feedback are also a problem if excellent timing performance is required.

References

[1] J.F. Wakerly, *Digital Design: Principles and Practices*, 4th ed. (Prentice Hall, Upper Saddle River, NJ, 2006).

[2] A. Rushton, *VHDL for Logic Synthesis*, 2nd ed. (John Wiley & Sons, Chichester, 1998).

[3] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 3rd ed. (Springer, Berlin, 2007).

[4] J.G. Proakis and D.K. Manolakis, *Digital Signal Processing*, 4th ed. (Prentice Hall, Upper Saddle River, NJ, 2006).

[5] R. Andraka, A survey of CORDIC algorithms for FPGAs, Proc. 1998 ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays, Feb. 22–24, Monterrey, CA, USA, pp. 191–200. URL: http://www.andraka.com/files/crdcsrvy.pdf.

[6] M. Alexander, Power Supply Distribution (PDS) Design: Using bypass/decoupling capacitors, Xilinx application note 623. URL:
http://www.xilinx.com/support/documentation/application_notes/xapp623.pdf.