power_sink
Data Sheet

Content

Table of Contents

Figures

1 Introduction

This component implements chains of FFs, SRLs and BRAMs that toggle at a given pattern. This allows draining much power for testing purposes.

The number of elements to toggle ins configurable.

1.1 Purpose

The document here describes this very generic firmware developed for all kinds of projects.

1.2 Scope

This document provides a detailed overview of the firmware interface and specifies the user interface.

1.3 Definitions, acronyms, and abbreviations

This document is based on the “IEEE Recommended Practice for Software Requirements Specifications” [1].

FPGA	Field Programmable Gate Array. Programmable logic device.
FF	Flip-Flop
SRL	Shift register implemented in LUTs
BRAM	Block RAM

1.4 References

[1] IEEE Std 830-1998, Recommended Practice for Software Requirements Specifications.

2 IP Description

2.1 Vivado Component

The *power_sink* feature is loaded into a System On Chip (SOC) as a AXI slave component. Different parameters can be configured in a simple GUI.

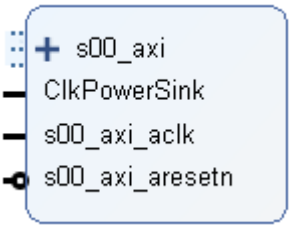

Figure 1: Component Overview

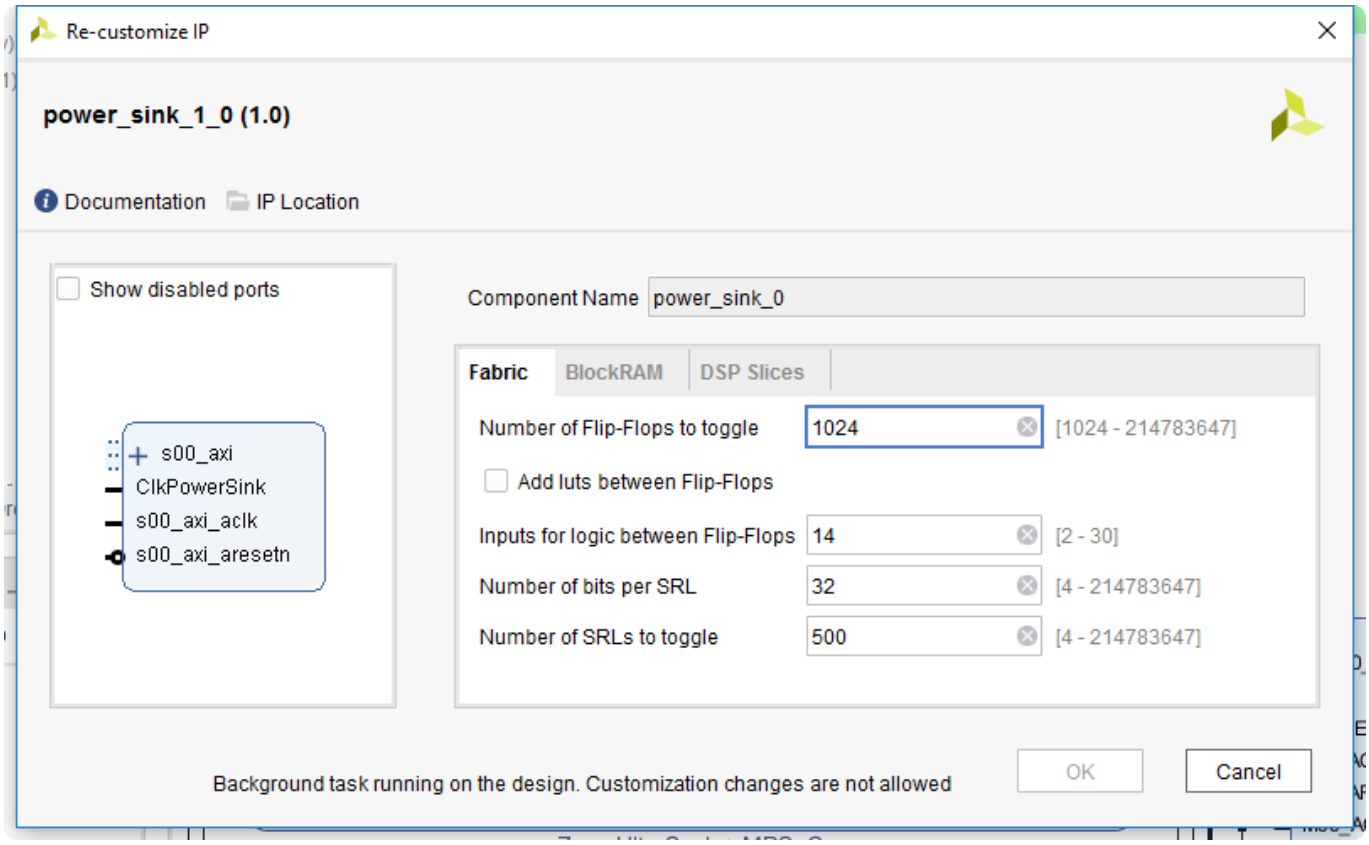

Figure 2: Component Configuration GUI for FPGA fabric

For the FPGA fabric, a chain of FFs with optional logic between them and SRLs can be configured separately. For more details about the FF chain and the logic in between, see detailed description later in this document.

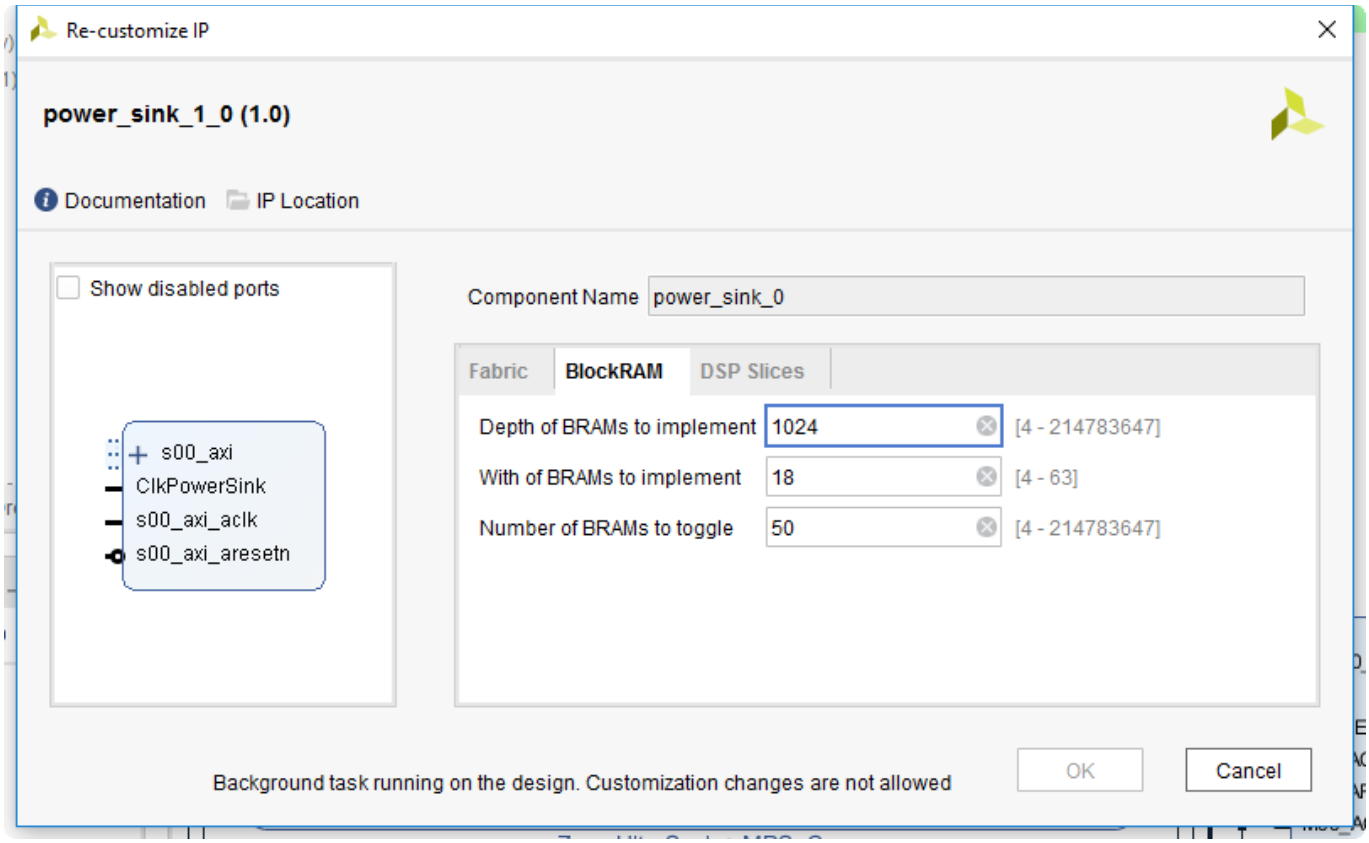

Figure 3: Component Configuration GUI for BRAMs

For BRAMs the settings are pretty self-explaining.

Implement IP

The implmenentation run can take a very long time in Vivado. For test purpose there is also a Switch in the Configuration GUI: “Implement IP” to basically overwrite the implementation of the logic blocks etc, to save compilation time, when other parts are in focus during tests. Only the AXI interface remains with empty registers.

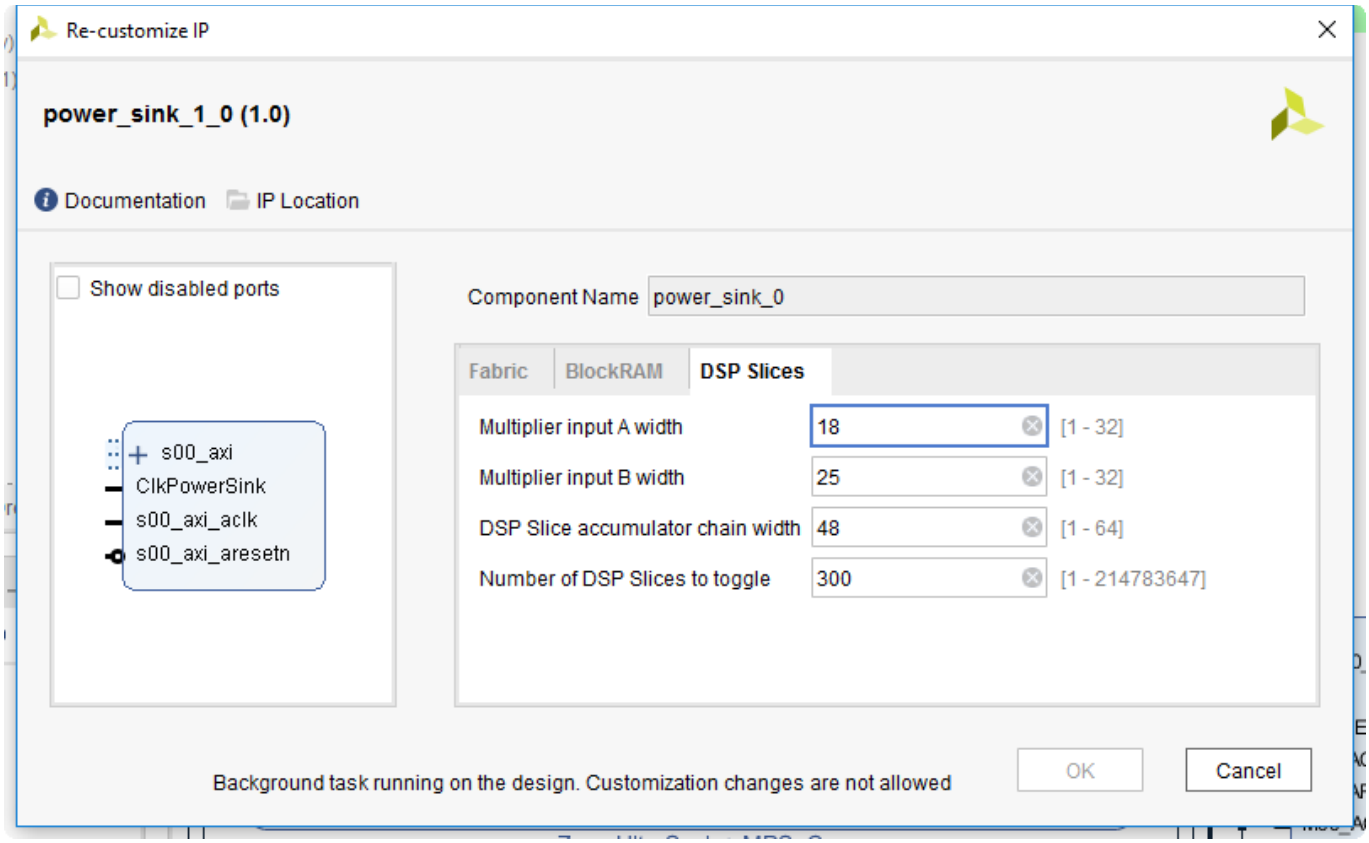

Figure 4: Component Configuration GUI for DSP Slices

For DSP slices, a chain of multiply add operations is implemented. Width of multiplier inputs and the accumulator chain are configurable to match different DSP slice architectures.

2.2 Architecture

2.2.1 Pattern Generation

The pattern is generated by a 32-bit shift register. The initial content of the shift register can be modified by the user to have some control over the toggle-rate.

A pattern of 0xAAAAAAAA or 0x55555555 leads to maximum toggle rate (all values toggle every clock cycle). The minimum toggle rate can be selected by using the pattern 0x0000FFFF or 0xFFFF0000 which leads to bits toggling only every 16th cycle.

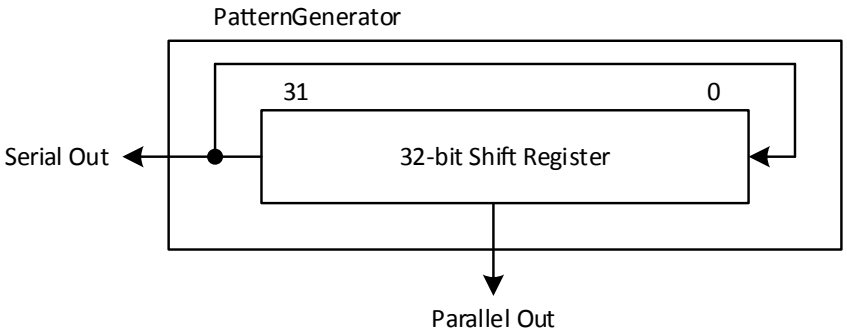

Figure 5: Pattern Generator

2.2.2 FF Implementation

For the FF toggling, a chain of FFs (with attributes that prevent optimization) is implemented. It is fed by the serial output of the pattern generator. Its output is fed to an AXI register only to prevent optimization (but reading this AXI register does not make any sense to the user).

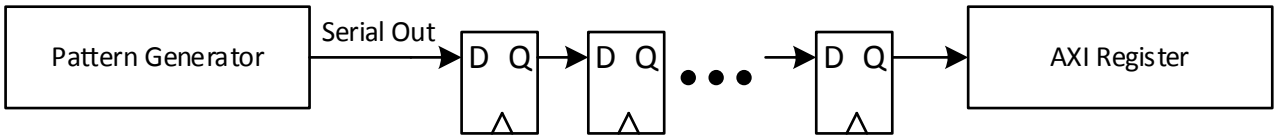

Figure 6: FF Implementation

Because real designs do not only contains FFs but also LUTs, it is possible to connect the FFS through logic. In steady operation case (if the user is not changing the pattern), the LUTs will output exactly the same thing as if there was only a wire (i.e. the chain of FFs still behave as if there was no logic). This is achieved by ANDing together FFs that are spaced by exactly the number of bits in the pattern. As a result, all inputs always have the same value.

The logic between FFs is only there to increase power consumption by using additional LUTs and, even more important, by utilizing some interconnect. A pure FF chain will not utilize much interconnect and hence is way less power-hungry than a real application.

The number of inputs to the logic between the FFs (and hence the number of LUTs and the amount of interconnect to use) is configurable.

The figure below shows the logic implemented. For simpler drawing, a pattern of only 5 bits (instead of 32 as implemented) is shown. The picture shows a logic containing two logic inputs for each stage.

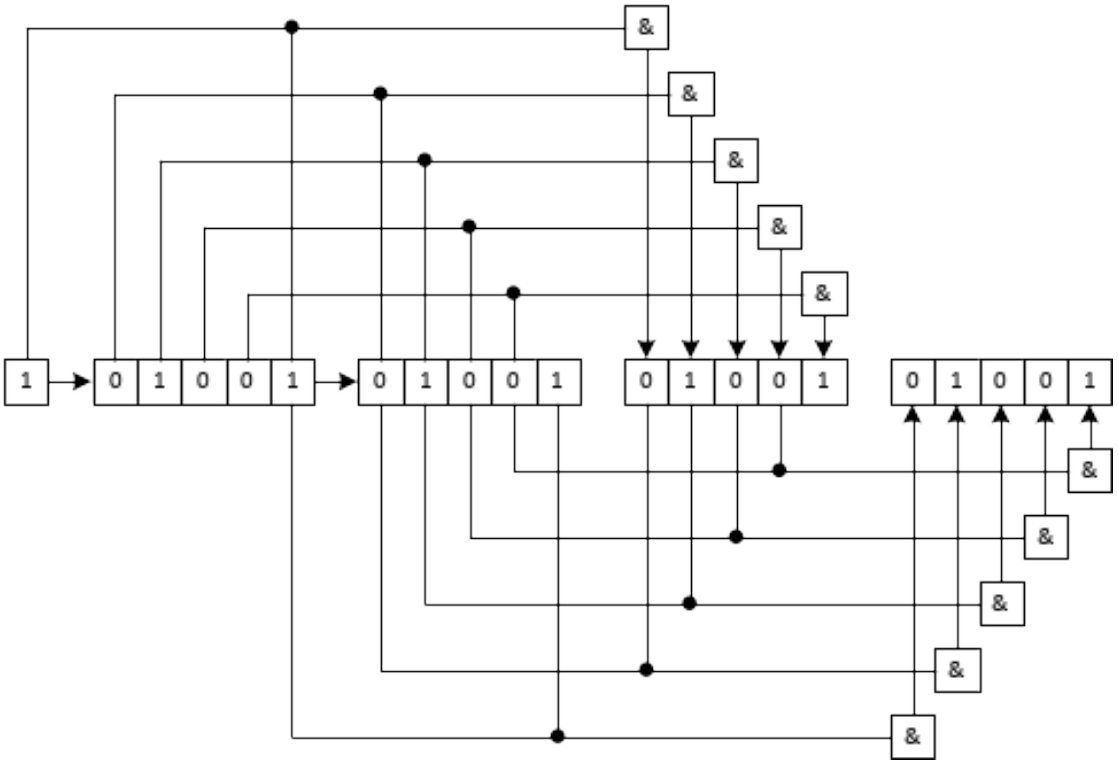

Figure 7: Logic between FFs

The first few registers are still implemented as normal chain without logic. All other FFs are fed by AND gates. The picture below shows the state of the chain after one clock cycle. It is obvious that the logic acts exactly the same way as a shift register would do.

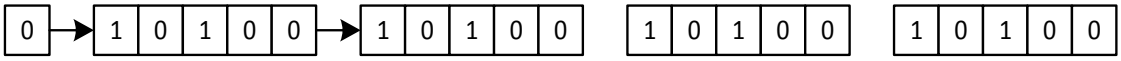

Figure 8: Logic between FFs (next clock cycle)

2.2.3 SRL Implementation

The implementation for SRLs is the same as for FFs, just with SRLs and attributes to enforce SRL implementation.

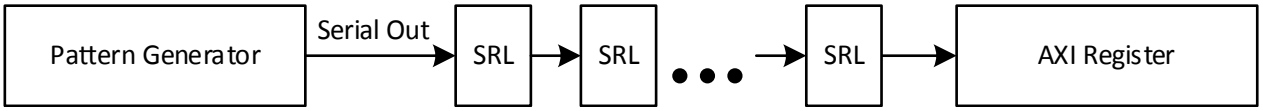

Figure 9: SRL Implementation

2.2.4 BRAM Implementation

For the BRAMs, the pattern generator is duplicated. The second pattern generator is initialized with the inverted version of the requested pattern. This results in the same number of toggling events.

An address counter counts through all addresses available and the pattern is written/read constantly. The address counter is distributed to all the BRAMs in the chain in pipelined fashion to prevent timing problems.

Both ports of the BRAMs are used. The addresses for both ports loop over the whole address range of the RAMs but the addresses are shifted by half the address range.

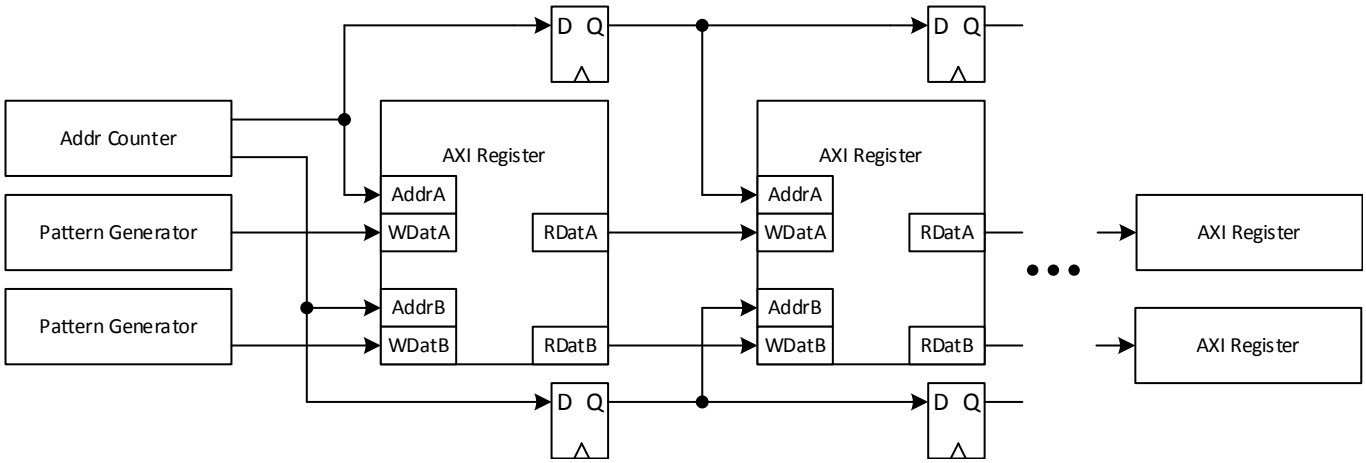

Figure 10: BRAM Implementation

2.2.5 DSP Implementation

For DSP Slices a chain of multiply/add operations is implanted. The implementation is written in a way to match the Xilinx DSP slice architecture that was not changed for many years now, so it is assumed it will stay like that in future.

Inputs A/B are both toggling between two patterns (A1/A2 resp. B1/B2) every two cycles. This leads to the following calculations happening in an endless loop:

1. A1 x B1
2. A2 x B1
3. A2 x B2
4. A1 x B2

Worst case power is consumed if the Patterns +1/-1 are used (in this case all bits toggle every cycle).

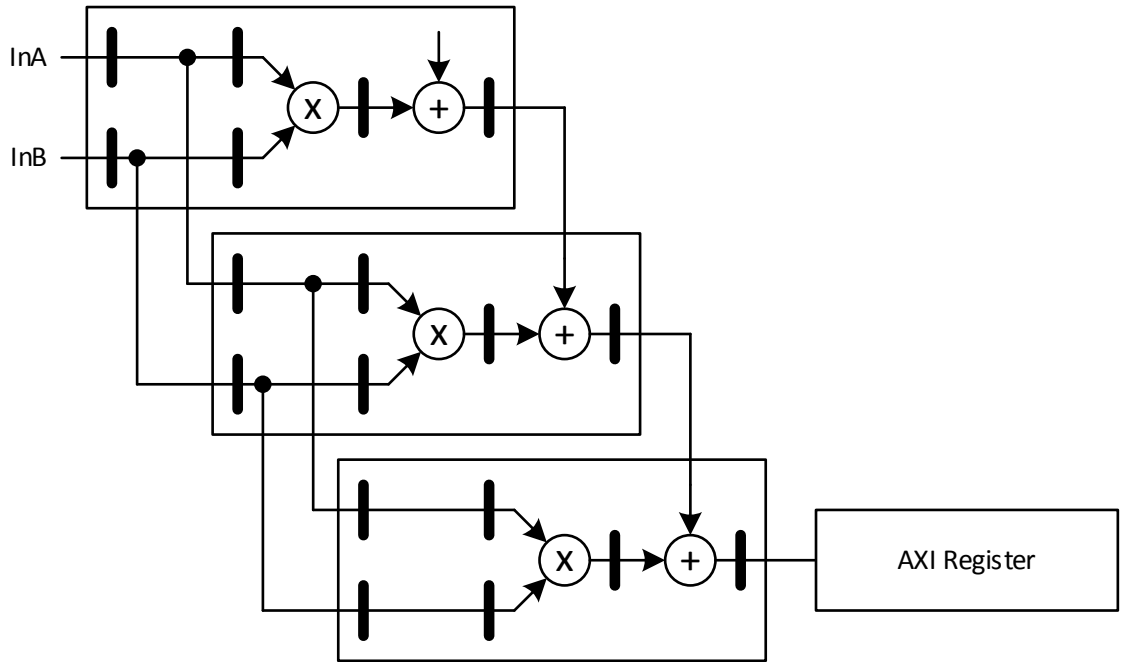

Figure 11: DSP Implementation

Because DSP slices are connected directly, they must be in one DSP slice column on the device. To allow using multiple columns, every 50 DSP slices some pipelining in fabric is added (not shown on the figure).

As a result, DSP slices can be distributed over multiple columns but only in groups of 50 consecutive slices.

3 Register Bank Description

3.1 Configuration

The following short notations for component parameters are used:

N SPI Transfer size in bits

M $\log_2(Depth_{FIFO})$

Address offset	R/W	Bit	Name	Description
0x00	R/W	0	EnaFf	Enable FF chain (reset value = 1) Enabled if EnaFf = 1 and EnaGlobal = 1
0x04	R/W	0	EnaSrl	Enable SRL chain (reset value = 1) Enabled if EnaSrl = 1 and EnaGlobal = 1
0x08	R/W	0	EnaBram	Enable BRAM chain (reset value = 1) Enabled if EnaBram = 1 and EnaGlobal = 1
0x0C	R/W	0	EnaDsp	Enable DSP chain (reset value = 1) Enabled if EnaDsp = 1 and EnaDsp = 1
0x10	R/W	0	EnaGlobal	Enable/Disable all logic
0x20	RW	31:0	PatternFf	Pattern for the FF chain
0x24	RW	31:0	PatternSrl	Pattern for the SRL chain
0x28	RW	31:0	PatternBram	Pattern for the BRAM chain
0x30	RW	31:0	PatternDspA1	Pattern A1 for the DSP chain (lower N bits are used)
0x34	RW	31:0	PatternDspA2	Pattern A2 for the DSP chain (lower N bits are used)
0x38	RW	31:0	PatternDspB1	Pattern B1 for the DSP chain (lower N bits are used)
0x3C	RW	31:0	PatternDspB2	Pattern B2 for the DSP chain (lower N bits are used)

Table 1: Registers

Normally direct access to the registers is not required because a low-level driver for the IP-Core exists.

4 Developer Information

4.1 Generate Fast Transients

Fast transients in power consumption are the most critical situation for the power supplies.

If the Core is enabled the first time, the power consumption will not increase immediately because all the FFs, SRLs and BRAMs are initialized with zeros. So first let the core run for a while to fill all cells with the pattern, then disable it and after a while re-enable it to generate transients of maximum steepness.

Use the global enable to enable all chains at the same time to generate one quick transient.

4.2 Packaging

To simplify re-packaging of the IP-Core after changes and avoid trouble with the Vivado GUI, a packaging script was written. To re-package the IP-Core, follow the steps below:

1. Open Vivado
2. In the TCL console, navigate to the “scripts” directory (using “cd <path>”)
3. Execute “source ./package.tcl”