FPGA partial configuration using ICAP

CEA Leti - LSCO

1 Introduction

This document aims to explain how to use the ICAP (Internal Configuration Access Port) module in FPGAs to perform partial reconfiguration. ICAP is a hardware block inside the FPGA fabric that provides access to the internal configuration memory. It allows both read and write operations; however, this document focuses exclusively on the write capabilities of the module to change a portion of the FPGA's configuration at runtime.

It is important to note that full configuration of the FPGA cannot be done using ICAP, as this would overwrite the logic responsible for controlling the ICAP itself, effectively disabling the module during the process.

While the ICAP module has some architectural differences in UltraScale and UltraScale+ FPGA families, its core functionality remains largely the same, with added capabilities. This document specifically details the use of the ICAPE2 primitive found in 7 Series FPGAs, including how to instantiate and operate it for partial reconfiguration. To implement and test the ICAP-based partial reconfiguration flow, this work uses the Artix-7 7 Series FPGA.

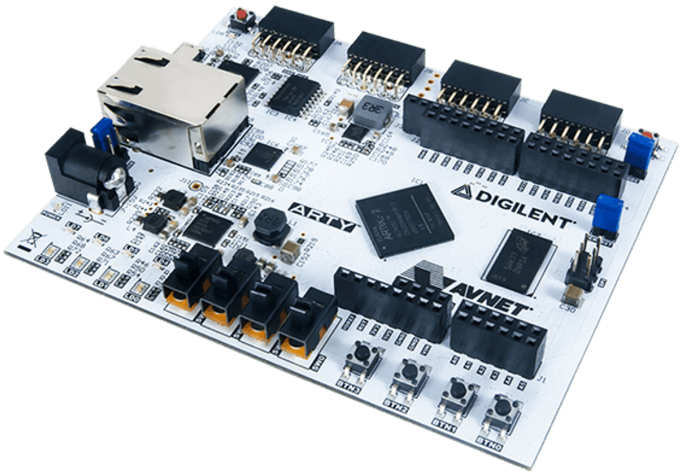

Figure 1: Arty A7-100T

2 ICAP Overview

Access to the FPGA's configuration functions from within the FPGA fabric is provided by this design element. It facilitates reading from and writing commands and data to the configuration logic of the FPGA array.

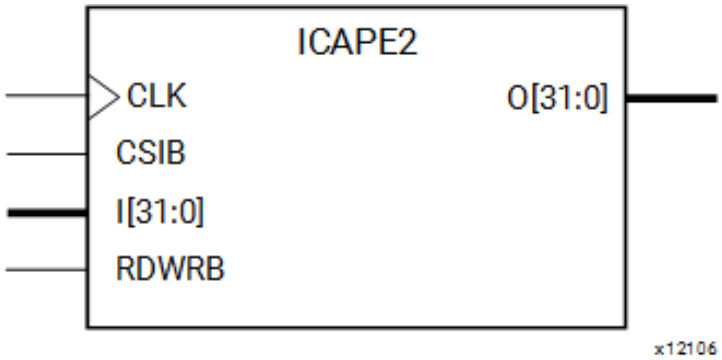

Figure 2: ICAPE2

Signal	Direction	Width	Description
`CLK`	Input	1	Clock input
`CSIB`	Input	1	Active-low enable signal for the ICAP interface. When low, enables configuration access.
`I[31:0]`	Input	32	Configuration data input bus. Carries data to the configuration logic.
`O[31:0]`	Output	32	Configuration data output bus. Carries data read from the configuration logic.
`RDWRB`	Input	1	Read/Write select input. When low, the interface performs a write operation; when high, it performs a read operation.

Table 1: Description of the ICAP I/O

3 Writing data in configuration logic

3.1 Continuous Data Loading via ICAPE2

In applications where an uninterrupted stream of configuration data can be provided, continuous data loading ensures bitstream delivery through the ICAPE2 interface.

After FPGA power-up and startup completion, the configuration logic within the fabric must initiate the write operation as follows:

1. Set `RDWRB` = 0 to indicate a write operation.
2. Assert `CSIB` = 0 to enable the ICAPE2 port.
3. Ensure that `RDWRB` is already Low before asserting `CSIB`, as asserting `CSIB` while `RDWRB` is High triggers an ABORT on the next clock edge.

Once `CSIB` is asserted, configuration data is presented on the `I[31:0]` input bus and sampled on each rising edge of the `CLK` signal. Initially, only the lower byte (`I[7:0]`) is sampled to detect the bus width through a synchronization word search. After the synchronization word is identified, the appropriate bus width (8, 16, or 32 bits) is used for further configuration.

The configuration process continues, byte-by-byte (or word-by-word depending on the width), until the entire bitstream is loaded. Upon completion:

- The FPGA enters its startup sequence, triggered by a startup command within the bitstream.
- The `DONE` pin is driven High by the device during startup.
- The controlling logic must continue generating `CLK` pulses during this phase. A minimum of 64 additional clock cycles is recommended to ensure reliable startup.

After startup concludes, both `CSIB` and `RDWRB` can either be deasserted or left asserted. Importantly, at this stage the ICAPE2 interface is inactive, so toggling `RDWRB` no longer causes an ABORT.

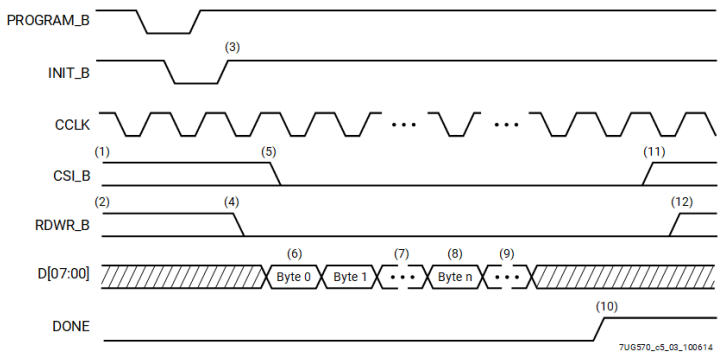

Figure 3: Continuous Data Loading

- **Signal order is critical:** `RDWRB` must be Low *before* asserting `CSIB`.
- **Tying control signals:** If no readback is required, `RDWRB` can be tied Low; similarly, if only one device uses ICAP, `CSIB` can be tied Low.
- **Bus width support:** ICAPE2 supports x8, x16, or x32 data bus widths, consistent with SelectMAP.

3.2 Non-Continuous Data Loading via ICAPE2

In scenarios where the configuration controller cannot deliver a continuous stream of configuration data, **non-continuous data loading** is employed. This is particularly relevant when the controller must intermittently pause to fetch additional data.

Pausing Configuration

Configuration can be paused using one of the following techniques:

- **Free-Running CCLK Method:** Pause is implemented by deasserting `CSIB` while maintaining a running clock.
- **Controlled CCLK Method:** Pause is implemented by halting the clock signal (`CLK`) while keeping `CSIB` asserted.

Operation in Free-Running CCLK Mode

In this method, `CLK` continues to toggle while `CSIB` is toggled to control the loading process. The steps are as follows:

1. Set `RDWRB` = 0 (write mode). This configures the `I[31:0]` data bus as input for configuration data.
2. Wait for `INIT_B` to go High, indicating readiness for configuration.
3. Present data on `I[31:0]`; data is latched on the rising edge of `CLK`.
4. To pause configuration:
   - Deassert `CSIB`. Data on `I[31:0]` is ignored while `CSIB` is High.
   - When ready to resume, reassert `CSIB`.
5. Continue presenting data on the rising edge of `CLK`.

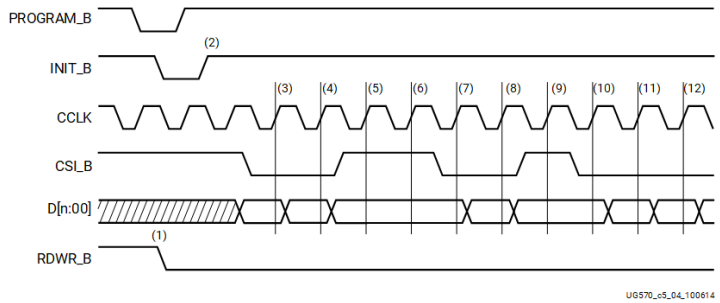

Figure 4: Non-Continuous SelectMAP Data Loading

- `RDWRB` should remain Low throughout the configuration unless readback is required.
- Avoid toggling `RDWRB` after asserting `CSIB`, as this may trigger an ABORT condition.
- Each valid configuration byte is sampled on a rising clock edge when `CSIB` is asserted.
- Data bus widths of x8, x16, and x32 are supported by ICAPE2.

4 Creating a Vivado project compatible with partial configuration

In Vivado, there is a feature called Dynamic Function eXchange (DFX) that allows designers to change or update specific parts of an FPGA design while the rest of the device keeps running normally. This means you can fix bugs, add new features, or switch functions on the fly without stopping the whole system. It helps keep the device working smoothly and saves time since you don't need to reprogram the entire FPGA every time you want to make a change.

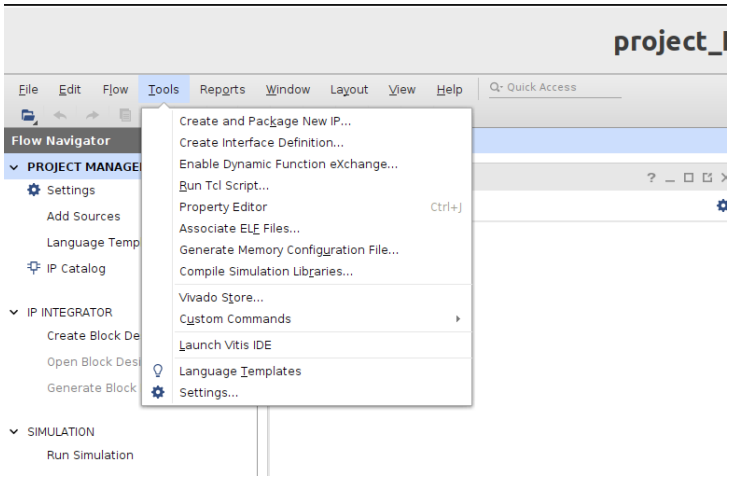

Figure 5: DFX Vivado

Important: Once the Dynamic Function eXchange (DFX) feature is enabled in a Vivado project, it cannot be disabled.

A *Dynamic Function eXchange Wizard* appears in the top right corner of the Vivado Project Manager. This wizard helps to define the **Reconfigurable Partitions** and the associated **Reconfigurable Modules** in the project.

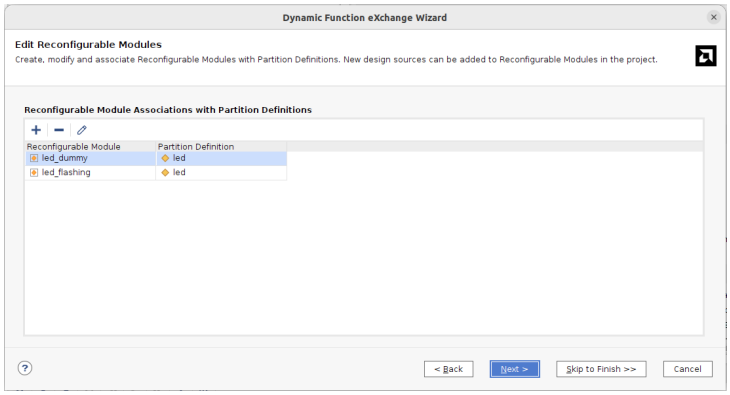

Figure 6: DFX Wizard – Reconfigurable Modules page

Here, two Reconfigurable Modules are selected: `led_flashing`, which creates a running LED light pattern using the board's LEDs, and `led_dummy`, which simply turns on four LEDs permanently.

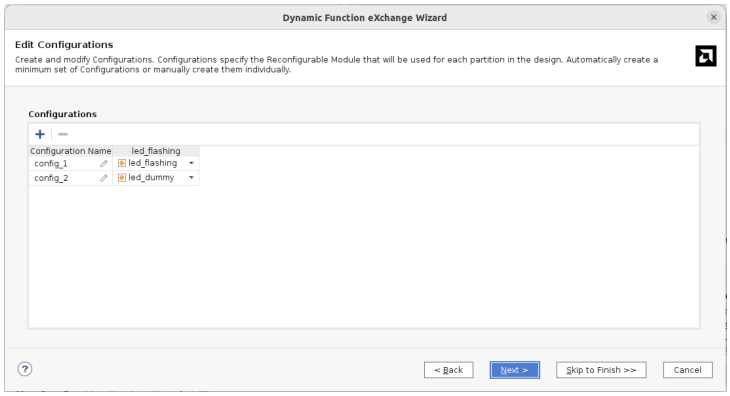

Figure 7: DFX Wizard – Configurations page

This page allows modification of the number of configurations available for the partition. In this example, there are two configurations, but for designs with multiple partitions, more complex combinations of configurations can be created and managed here.

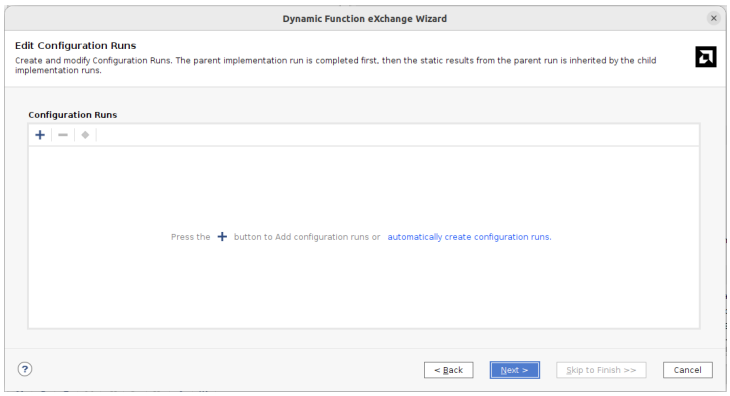

Figure 8: DFX Wizard – Configuration Runs page

Next, the wizard proceeds to the Configuration Runs page. It is important to select *Automatically create configuration runs*, as this automatically creates a parent implementation for the static design and the first configuration, and a child implementation for the second configuration. This setup ensures consistent routing between the configurations, which is critical for partial reconfiguration to work correctly.

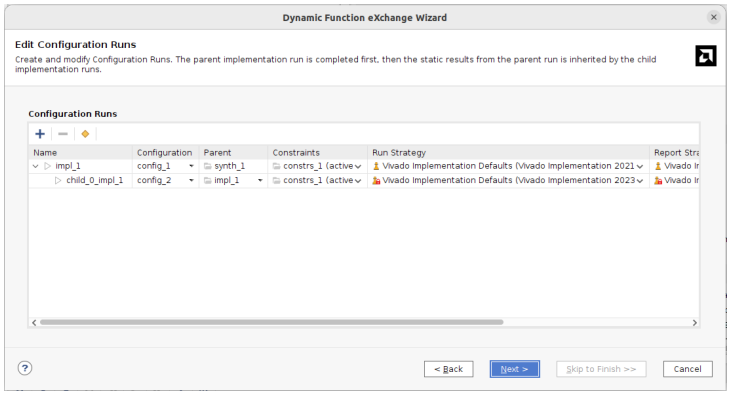

Figure 9: DFX Wizard – Child implementation created

The child implementation for the second configuration is created here, guaranteeing consistent routing between the static and reconfigurable logic.

5 Choosing the Reconfigurable Partition on the FPGA

Before launching implementation, a configurable Pblock must be selected. A special tool is available when the synthesized design is open.

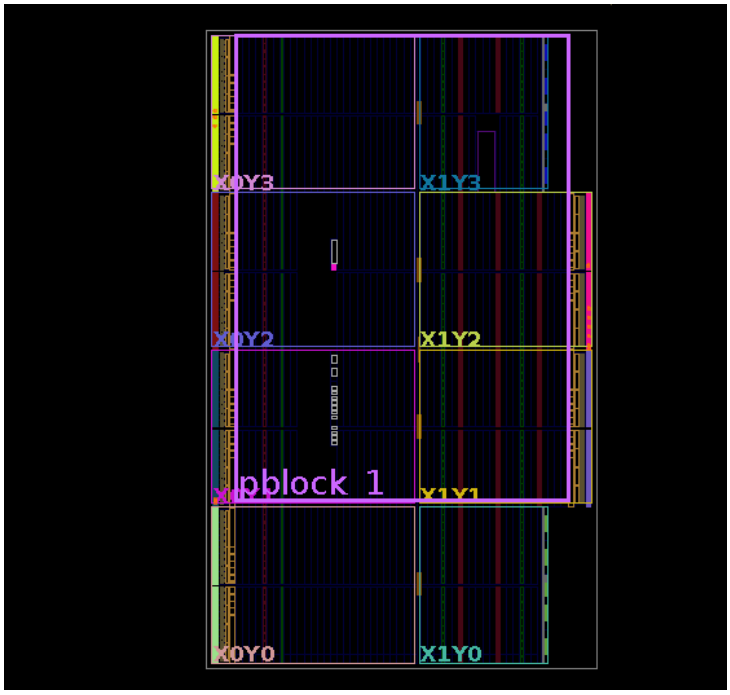

Figure 10: Pblock selection

Constraints of this created Pblock are automatically generated and written in the XDC file. These constraints define the area of the reconfigurable partition.

```
create_pblock pblock_1
add_cells_to_pblock [get_pblocks pblock_1] [get_cells -quiet [list led_flashing]]
resize_pblock [get_pblocks pblock_1] -add {SLICE_X0Y199:SLICE_X89Y50}
resize_pblock [get_pblocks pblock_1] -add {DSP48_X0Y20:DSP48_X2Y79}
resize_pblock [get_pblocks pblock_1] -add {PCIE_X0Y0:PCIE_X0Y0}
resize_pblock [get_pblocks pblock_1] -add {RAMB18_X0Y79:RAMB18_X3Y20}
resize_pblock [get_pblocks pblock_1] -add {RAMB36_X0Y39:RAMB36_X3Y10}
```

Figure 11: Pblock Constraints

Then, the Reconfigurable Module must be linked to the created Pblock, as shown in the figure below.

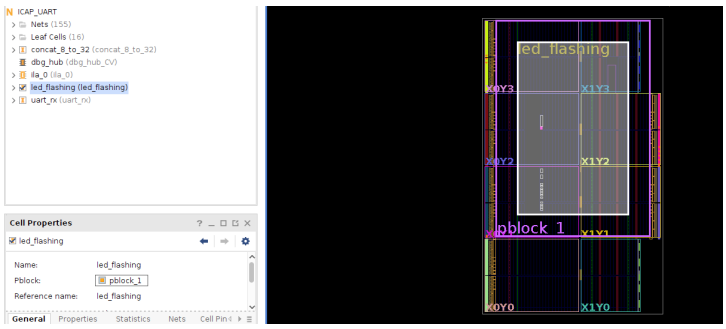

Figure 12: Reconfigurable Module linked to Pblock

The static portion of the digital circuit is not routed through the reconfigurable Pblock, as shown in the implemented design below.

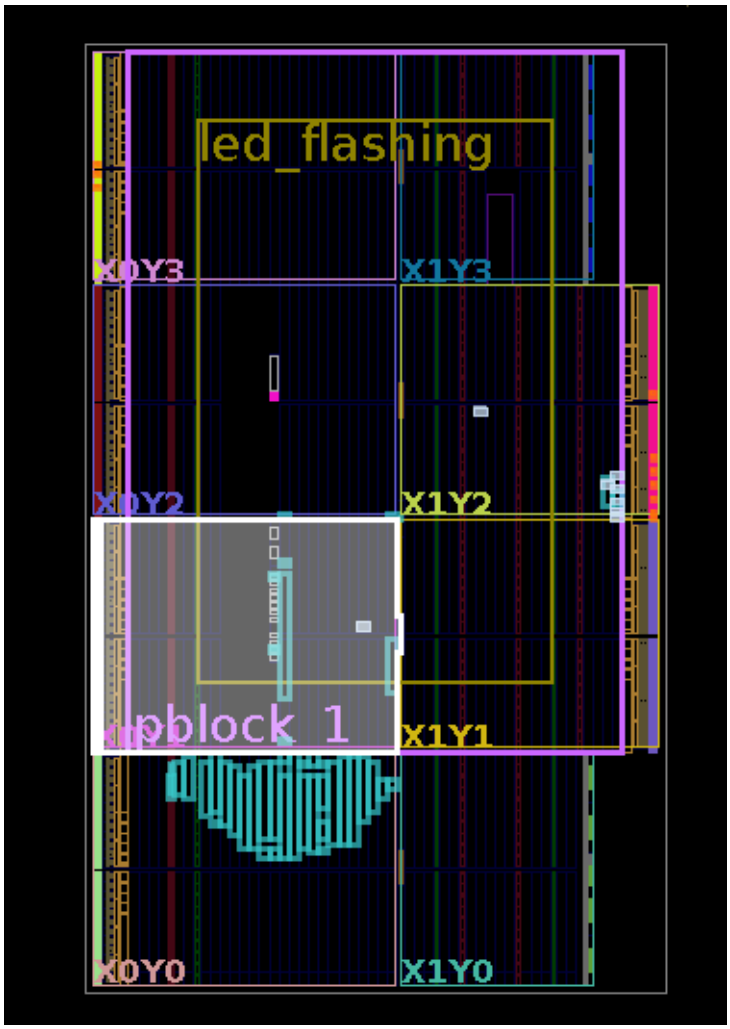

Figure 13: Implemented Design

5.1 Special Constraints Added to Reconfigurable Modules

RESET_AFTER_RECONFIG

The `RESET_AFTER_RECONFIG` feature ensures that, after a Reconfigurable Module (RM) is updated during partial reconfiguration, all logic inside that region is cleanly reset to a known initial state—just like after a full device configuration. While the static design and other dynamic regions continue operating, the reconfigured region receives a dedicated reset (via GSR) to safely initialize registers and flip-flops. This helps prevent unexpected behavior. In UltraScale and UltraScale+ devices, this reset is applied automatically. In 7 series FPGAs, however, the reconfigurable Pblock must be properly aligned to reconfiguration frames to use this feature.

```
set_property RESET_AFTER_RECONFIG true [get_pblocks pblock_1]
```

SNAPPING_MODE

The `SNAPPING_MODE ON` property is applied to Reconfigurable Partition Pblocks in DFX designs to automatically adjust their size and shape to meet floorplanning requirements. When enabled, it aligns the Pblock boundaries to legal programmable unit (PU) ranges used by the implementation tools. This ensures that the resulting Pblock complies with DFX constraints, especially for frame alignment and routing legality. The adjusted Pblock ranges are derived internally and are not directly written to the XDC file—only the `SNAPPING_MODE` property is saved.

This means that even if you manually make the Pblock as large as possible, there is still a physical limit on the size of the reconfigurable design, depending on the target FPGA architecture.

```
set_property SNAPPING_MODE ON [get_pblocks pblock_1]
```

References

[1] Xilinx, *7 Series FPGAs Configuration User Guide (UG470)*, v2023.2, December 2023. https://docs.amd.com/v/u/en-US/ug470_7Series_Config

[2] Xilinx, *Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)*, v2023.2, December 2023. https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration