# Solution Efficiencies for Dynamic Function eXchange Using Abstract Shells

WP533 (v1.0) June 22, 2021

## Abstract

Dynamic Function eXchange (DFX) enables great flexibility within Xilinx® silicon, empowering you to load applications on demand, deliver updates to deployed systems, and reduce power consumption. Platform designs allow for collaboration between groups, where one group can focus on infrastructure and another on hardware acceleration. However, DFX has fundamental flow requirements that lead to longer Vivado® Design Suite compile times and expose challenges with multi-user environments. The abstract shell flow removes some of these barriers, creating a more efficient path through the Vivado tools for DFX. Abstract shells open new possibilities for collaboration and compile-time efficiency for DFX platform solutions.

# Introduction

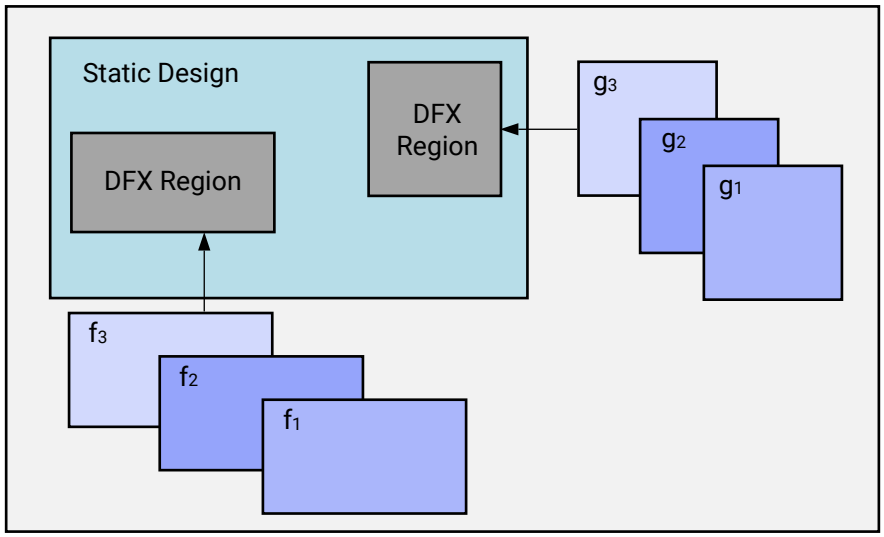
Dynamic Function eXchange is a powerful capability within Xilinx FPGAs and adaptive SoCs that expands the flexibility of the silicon. Functions can be uploaded on the fly, delivering new functionality to one part of the device while the rest remains operational, expanding the effective usage of the device while reducing downtime. This time-multiplexed silicon usage approach allows you to do more with less, providing a critical advantage for designers looking to save cost, power, and time.

DFX also enables multi-user environments, allowing multiple groups or multiple companies to share the programmable logic space. One group (call them the primary user) creates the infrastructure of the design, from board-level considerations like memory access and communication links to safety and security details and run-time managers. The primary user locks down this static platform and leaves one or more empty work spaces for other groups (secondary users) to fill.

However, DFX requires some upfront considerations:

- DFX designs require a specific design structure and layout to match the chip-within-a-chip paradigm. This more rigid structure can incur a penalty in the form of a longer design processing time.
- Accelerated functions can be delivered to work spaces that are within platforms on-premises or in the cloud. The standard Vivado DFX flow requires that the complete locked static image be present to provide the context to place and route these accelerated functions. This requirement could expose proprietary design information owned by the primary user in multi-user scenarios.

*Figure 1:* **Concepts of Dynamic Function eXchange**




DFX enables new possibilities, but these two operating conditions might not be desirable for all systems or designs. An enhanced approach is needed to alleviate these concerns.

# Abstract Shells to the Rescue

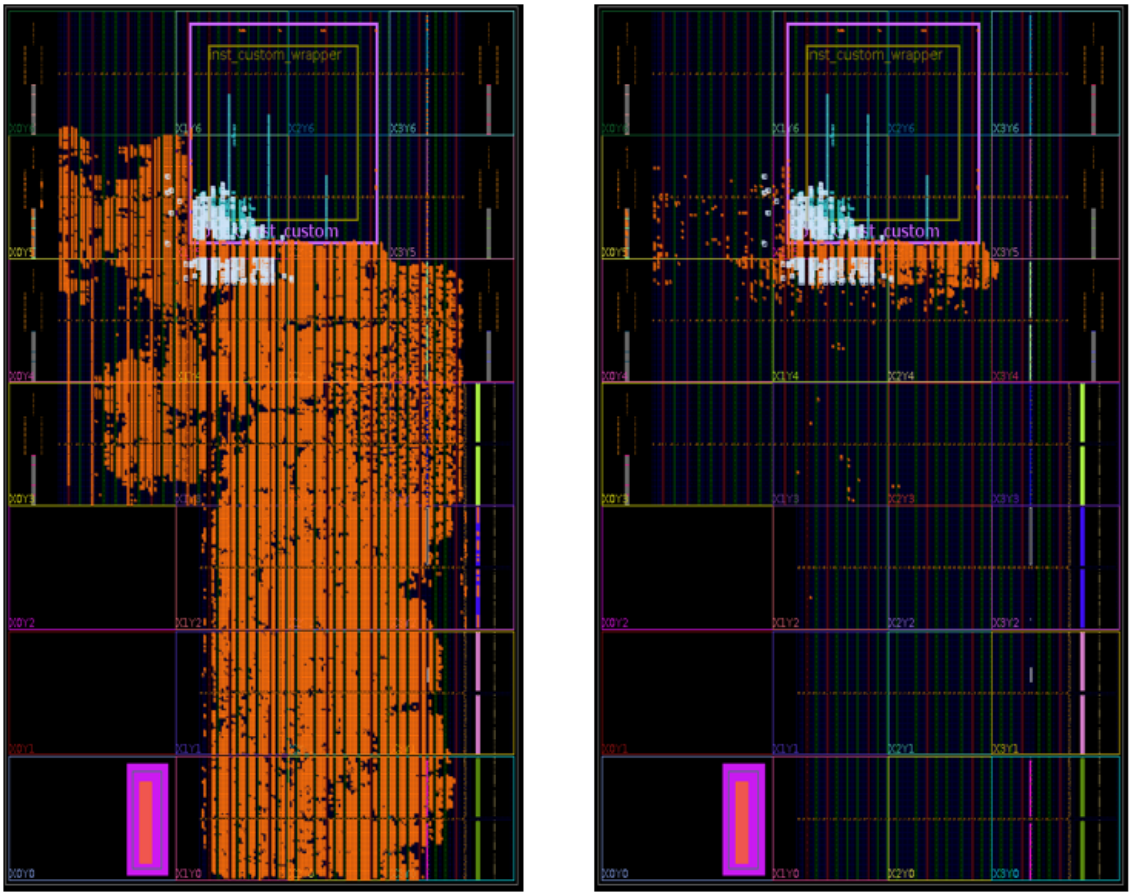
The abstract shell feature was released in the Vivado Design Suite 2020.2 for all UltraScale+™ devices and provides a few key advantages over the standard DFX flow.

### What are Abstract Shells?

The standard DFX flow in Vivado tools requires multiple passes through the implementation tools. The first pass establishes the place and route results for the static design along with one reconfigurable module (RM) per reconfigurable partition (RP). This static design image is the platform that is configured into the target device to remain operational during subsequent partial reconfiguration events. Each successive pass through implementation uses the locked static image as the framework to implement all remaining RMs. You are not permitted to modify the static design to ensure consistent context, but the static design must be opened in memory to allow the Vivado tools to understand the working design environment. Even if a target reconfigurable region is very small, the full static design image must be opened, which takes time and memory.

The abstract shell flow creates a trimmed down version of static design for a given RP. Fundamentally, an abstract shell is the minimal design image needed to provide the context for implementing a new RM and generating a partial bitstream for that module. The logical design of the shell contains the boundary interface to the module instantiation. The Pblock of the partition, including the expanded routing region, is captured as part of the design constraints, as are any clocking and boundary timing requirements. All logical and physical resource usage within the region is included to ensure the RM does not try to use anything already consumed by the static design. A partial bitstream is built for a physical region of the device and contains programming information for both static and dynamic parts of the design. It is critical that the static parts of these bitstreams remain absolutely identical.

*Figure 2:* **Abstraction of the Full Static Design (Left) Using an Abstract Shell (Right)**




The optimized abstract shell provides the following key benefits:

- The shell checkpoints can be significantly smaller than a full shell. This lowers the compile time and memory use for new RMs.
- For designs with more than one RP, runs can be set up to implement all RMs in parallel, because full design configurations are not needed.
- For scenarios where multiple users are involved, design security is a benefit. Because the vast majority of the static design is removed, the vast majority of proprietary design information is not visible in the abstract shell.
- Any licensed IP in the static design are not included in their entirety in the abstract shell. This means that license checks are bypassed during abstract runs.

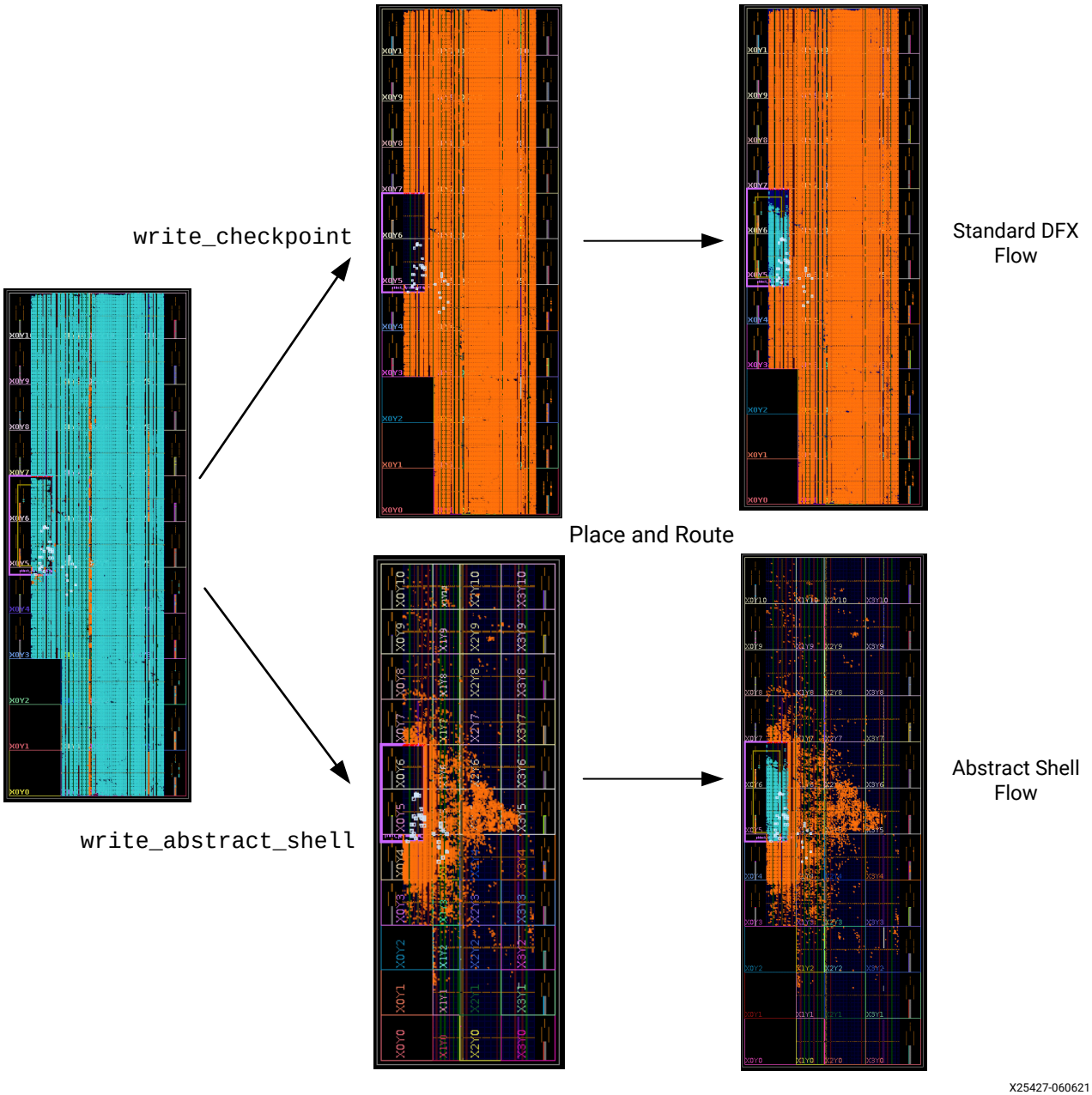
# Abstract Shell Design Flow

This section compares the abstract shell flow and the default DFX flow, examining the underlying commands to produce and then use abstract shells. With the abstract shell flow, the implementation of the parent configuration is absolutely identical to the standard DFX flow where you implement the static design and then lock down those results. The flow does not diverge until the static-only design result is saved. The `write_abstract_shell` command is used to black box the target partition, trim away unneeded static, lock the remaining design, and

validate the result using `pr_verify`. This command must be called for any RP that is to become an abstract shell, because each RP has a unique footprint and connection to the static design. Finally, the implementation of the remaining RMs matches the standard flow, but now the focus is on a single RP for each child run. Instead of building new full-design configurations to implement (or reuse) multiple RMs per design, each RM is implemented in its own run in its own abstract shell, leading to greater efficiency.

***Note:*** In the Vivado Design Suite, the abstract shell flow is supported in non-project `Tcl` mode.

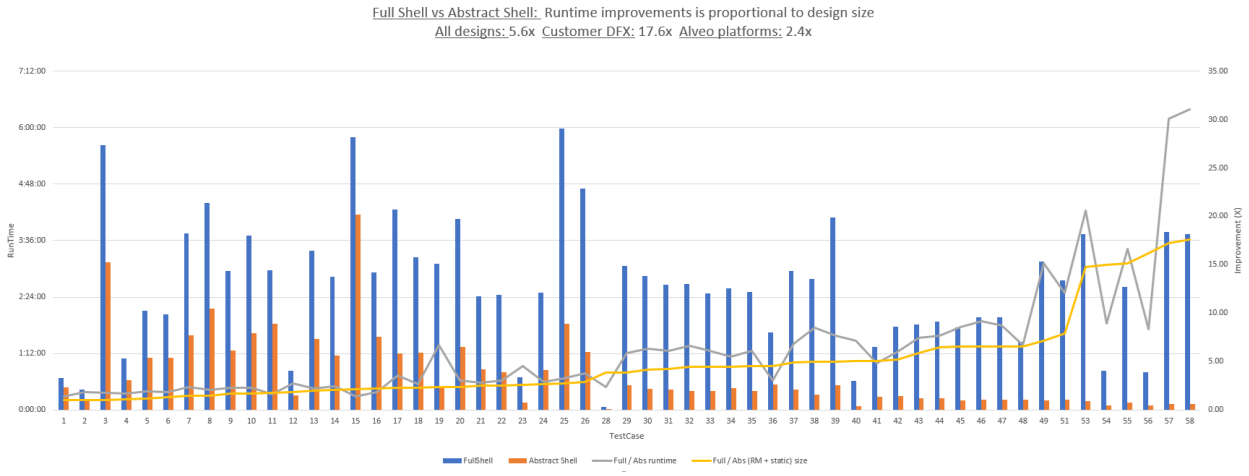
*Figure 3:* **Design Flow for the Abstract Shell Solution**



The parent design configuration can be used to create a full device programming image. With the initial RMs included in any RPs, each creates partial bitstreams. Abstract shell runs can generate additional partial bitstreams for the functions they implement. In single-user environments, within a single company or design group, additional full design configurations can be created for bitstream generation by linking routed static and dynamic checkpoints. Any combination of static and reconfigurable images can be rebuilt to create any full or partial bitstreams needed for the target system. There is no limitation compared to a standard DFX flow in this regard.

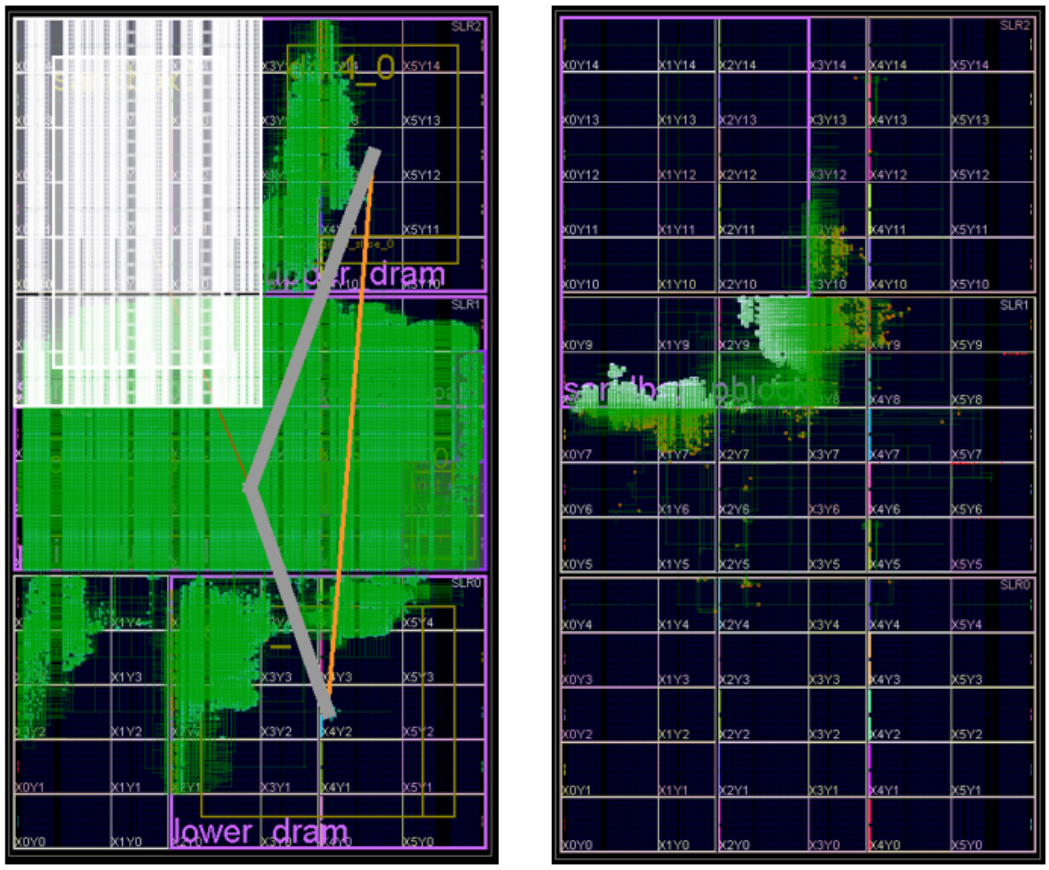
Abstract shell compile times are faster in nearly every scenario. How much faster depends on the structure of the design. Designs with very large dynamic regions and minimal static produce modest gains given that very little static is removed to create the shell. For example, Xilinx Alveo™ platform designs, which contain very little static logic, can be compiled more than twice as fast using the abstract shell flow. This can add up to big savings when you consider the static platform is rarely revisited; the primary usage is to build new RMs. For designs with smaller dynamic regions and larger static regions, the gains are much greater—across a varied design suite, improvements that can be 5, 10, or more times faster than the standard DFX flow. Although the creation of the abstract shell takes time, it is a step that is not done often. The following figure shows the compile-time savings (gray line) using the abstract shell across a variety of UltraScale+ device designs.

*Figure 4:* **Compile-time Savings**




The following image is a closer look at a specific sample design. This design has a dynamic region that covers about a quarter of the device, with a static region that is not much more than a single super logic region (SLR) in the three SLR Virtex® UltraScale+™ FPGA (VU9P). After the static design is implemented, new RMs can take 1.5 hours to compile, partly due to the 266 MB static design checkpoint. Using the abstract shell flow, the design checkpoint of the shell shrinks to only 10 MB, memory usage is cut by a third, and compile time is cut by two thirds.

*Figure 5:* **Example of a Full Shell (Left) Compared to an Abstract Shell (Right)**




# Abstract Shell for Multi-User Environments

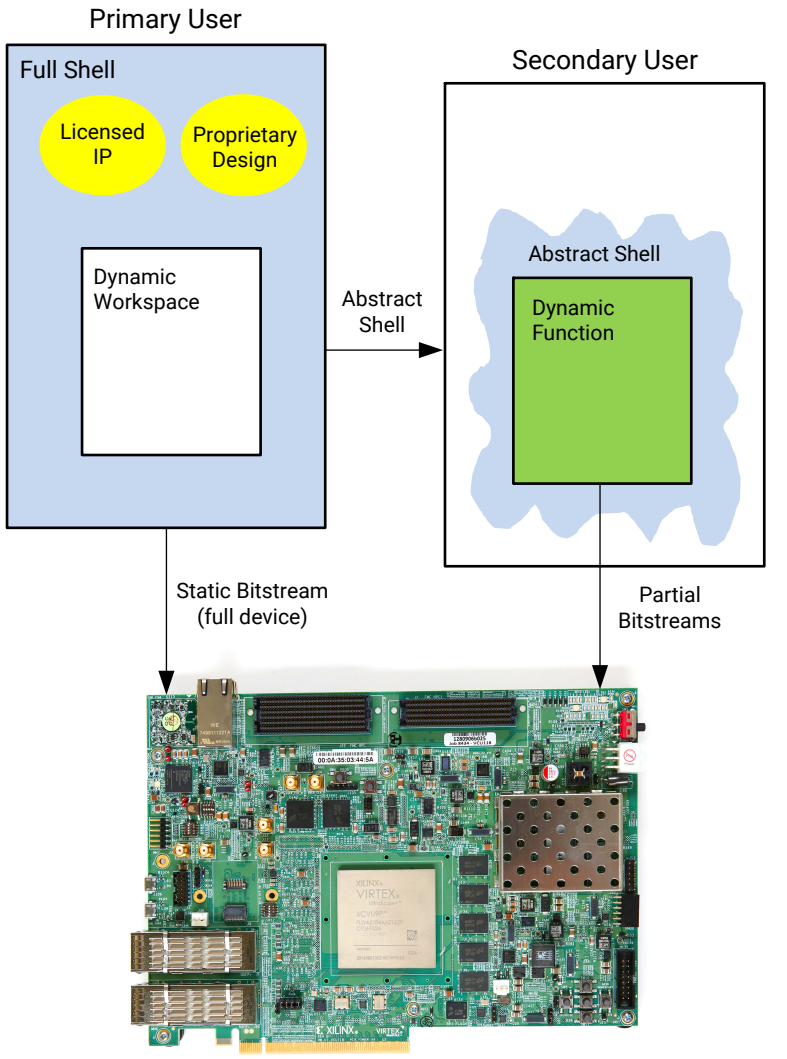
Using the abstract shell flow is a tremendous benefit when employing DFX for multi-user environments. DFX enables you to build a solution with a dynamic application space within a Xilinx device. You can build a platform that can be passed to another user to later fill in the space with any number of applications to be swapped on the fly. The platform built by the primary user locks down key details that must not change including memory interfaces, communication channels, and design safety and security features. The secondary user inserts their functionality into this locked context to be accelerated in hardware. In this scenario, using the standard DFX design flow, the primary user shares the full static design checkpoint with the secondary user, potentially exposing design secrets. The secondary user also needs licenses for any IP contained within the static design, even though they are not implementing that part of the design.

With the abstract shell flow, the bulk of the static design is stripped away, hiding proprietary design information. Some fragments might still remain based on connectivity with the target dynamic region, which means that the primary user needs to examine the contents of the abstract shell before distributing it. However, in nearly all cases, there is not enough information to reverse engineer the functionality. For the same reason, IP license checks are bypassed for any IP in the static shell, because it is impossible to use that IP other than how it is implemented by the primary user.

**IMPORTANT!** *Because the IP functionality is passed to the secondary user within the static design bitstream, it is the responsibility of the primary user to review the terms and conditions of any IP to ensure they have redistribution rights for that intellectual property.*

When programming the device in a single-user environment, the standard DFX approach is still good. Any full design image can be assembled by linking static and reconfigurable routed checkpoints then calling `write_bitstream` to generate a full device BIT file. However, for multi-user environments, the primary user must supply an initial full-device bitstream to initially program the Xilinx device. This image can have any sort of default application in the dynamic region, from a functionless gray box, to a *hello world*, to a test application that exercises some fundamental features of the system. Then, the secondary user partially reconfigures the device to load in their accelerated functions using partial bitstreams generated from abstract shell runs.

*Figure 6:* **Programming Flow for Multi-User Environments**




This multi-user environment example shows all the benefits of DFX:

- Hardware acceleration of applications with on-the-fly dynamic function swapping
- Fast compilation through the Vivado DFX development flow
- Removal of any sensitive design information from the static platform

It is critical that all full and partial bitstreams remain in sync for any DFX methodology, including the abstract shell. Even though partial bitstreams can be generated from within the abstract shell environment, they are compatible with the original static design used to create the shell. This can be confirmed by calling `pr_verify` to compare the abstract shell alone versus the abstract shell with any routed RM, or by reassembling different full-design configurations and running `pr_verify` again.

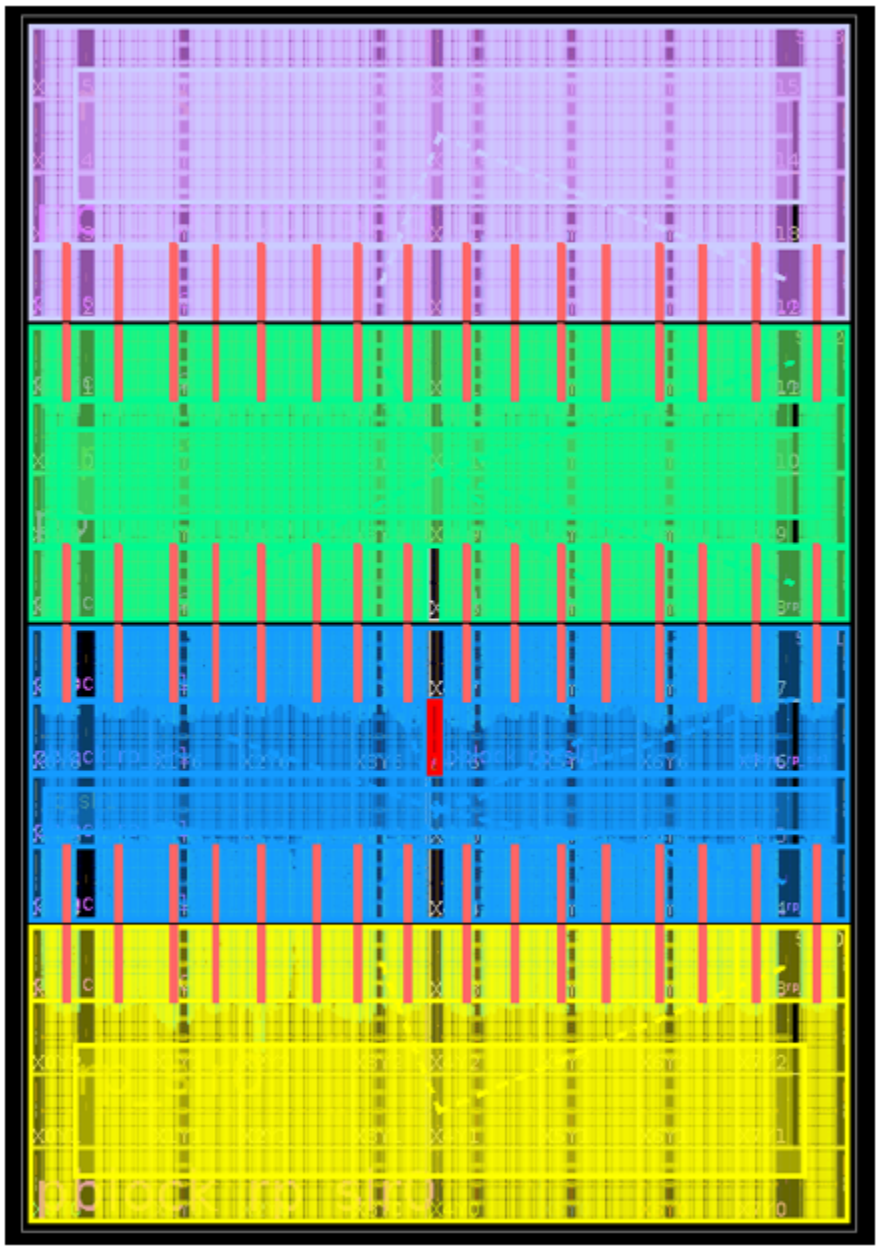
# Abstract Shell for Flat Designs

The abstract shell flow can even help designs not using dynamic reconfiguration. Large FPGA designs can experience long compile times through place and route due to the sheer volume of information to be considered during this process. For these designs, especially in larger multi-SLR Virtex UltraScale+ devices, DFX can be used to establish an in-context hierarchical design (HD) flow solution. This reduces the granularity of design iterations by segmenting the design into smaller, more manageable pieces. You set up the hierarchy to have a thin top-level wrapper and multiple RPs. When a minor iteration is required within one portion of the design, only that RM must be re-implemented. The abstract shell flow further enhances this approach by carving away the rest of the design, focusing attention on the modified module, significantly reducing compile time.

This in-context HD approach requires a segmented and floorplanned design. Resource utilization or performance limited designs are not good candidates for this methodology because applying DFX prevents logic optimization across boundaries and restricts placement within the target Pblocks. However, if your design can be divided into independent building blocks that occupy their own regions of the device, DFX with the abstract shell flow can greatly improve productivity when making small design changes.

In the following image, a four-SLR VU13P is divided into four reconfigurable partitions, one per SLR. Each RP can be extracted as an abstract shell, allowing each to be implemented independently. These abstract shells can be shared with different team members or simply run in parallel. When each region is complete, the RM-level checkpoint is linked with the routed-and-locked top-level checkpoint before bitstream generation is done.

*Figure 7:* **Improving Productivity with Segmented Dynamic Regions**



# Conclusion

Abstract shells for DFX is a powerful tool that can significantly improve design processing time and enhance design security. Examine your design structure and goals to see if abstract shells can provide benefits for your environment. For more information on abstract shell and DFX, consult the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909). A tutorial design is included as *Lab 9* within the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947). These documents and more are available on the DFX page.

# References

These documents provide supplemental material useful with this white paper:

1. *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909)
2. *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947)

# Revision History

The following table shows the revision history for this document.

| Section | Revision Summary |
|---|---|
| **6/22/2021 Version 1.0** | |
| Initial release. | N/A |

# Please Read: Important Legal Notices

OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

### Copyright