Designing Relocatable Partitions with Vivado Design Suite

Björn Gottschall

Vivado 2016.4

01.06.2017

Dr.-Ing. Thomas B. Preußer

Prof. Dr. Akash Kumar

Processor Design » Computer Science » TU-Dresden

Contents

1 Introduction

The options to reconfigurate a FPGA in a running system is giving a lot of opportunities to dynamically adjust the hardware as required. Partial reconfiguration is splitting a FPGA into partitions or modules which can be individually and independently reconfigurated at runtime, which increases the flexibility of dynamic hardware modules even more and gives a better usage of logic resources.

The designing and implementation of partial reconfigurable partitions can be quite time consuming because every bitstream has to be build for each partition. Relocation is giving the option to parse a bitstream for different partitions than it was build for, so a bitstream has to be build only one time for one partition.
Other benefits of relocatable partitions are that older bitstreams stay compatible with new static implementations. As long as the interface for each partition will not be touched, the static design can be completly exchanged. Also with the right design choices it will be easy to allow a unprivileged user to reprogram only his partitions without interfering with static logic or other partitions.

To follow this guide it is necessary to read [UG909] *Partial Reconfiguration User Guide* and [XAPP1222] *Isolation Design Flow for Xilinx 7 Series FPGAs*.

2 Considerations for Relocatable Partitions

Before designing a FPGA Layout for relocatable partial reconfigurations, there are some details which have to be considered.

Correct height of the PR Regions

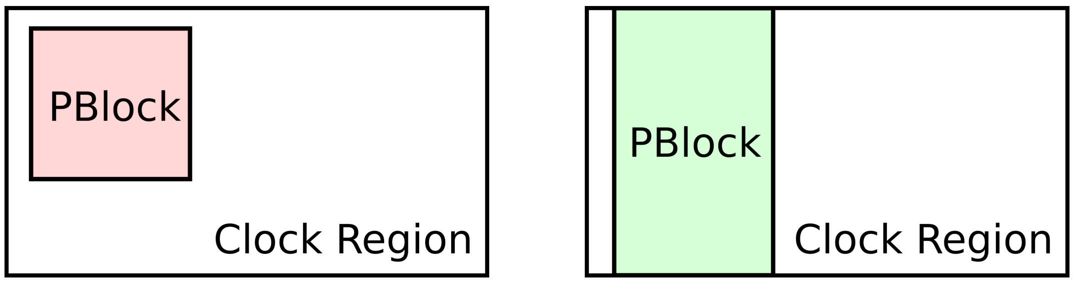

Figure 2.1: wrong pblock placement (left) and correct pblock placement (right)

A PR Region is represented through a cell which is assigned to a pblock. The pblock should fit into one frame which is at least 50 CLBs high and aligned to the clock region in the 7 Series FPGAs [Frame-UG702].
The smallest reconfigurable part of a 7 Series FPGA is a 50 CLBs high and one CLB wide frame. Making a pblock smaller in height, would result in a bitstream which always reconfigures a whole frame in its clock region. If a smaller pblock is required make sure that no logic and routing is placed in the frame next to the pblock which could be destroyed through reconfiguration of relocated bitstreams.

Equal Arrangement of resources

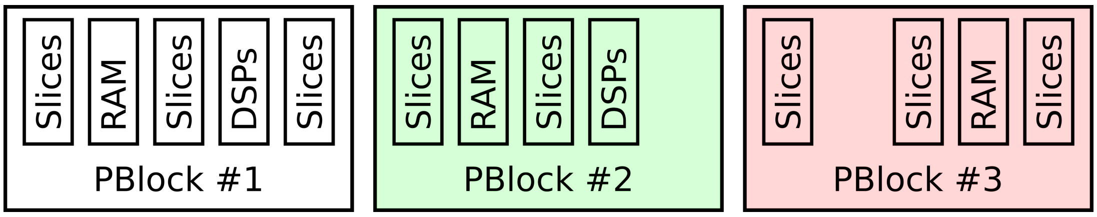

Figure 2.2: In comparison to the first pblock only the second one is compatible with relocation.

For the definition of pblocks for relocatable partitions the most important detail is the arrangement of resources in each pblock. Best practice is to define completely equal pblocks. That means each pblock has absolutely the same amount of resources of any type at the same location relative to the pblock position.
The bitstreams of smaller partitions can be used for relocation into bigger partitions like seen in Figure 2.2. The pblock #2 fits into pblock #1 as long as there are no interfaces on the missing resources. However pblock #3 is not compatible to pblock #1 due to the incompatible arrangement of resources. Relocation doesn't has to work both ways, as seen on pblock #1 which is not compatible with pblock #2 because of missing resources.

Exclude unwanted Routing

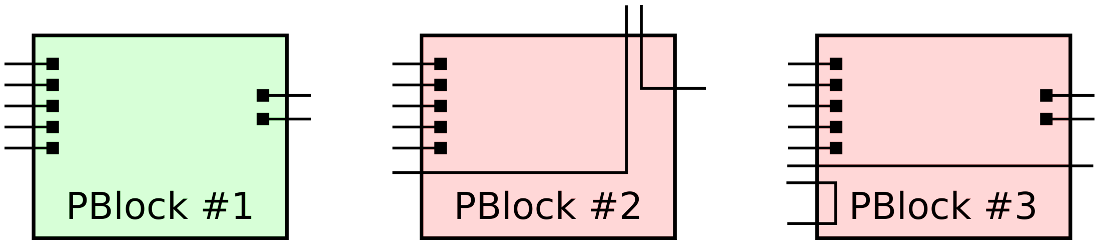

Figure 2.3: Pblock #2 and #3 have got global routes passing the partitions. PBlock #1 has only interface routing.

All routes which do not belong to the partitions have to be excluded. That includes pass-through global routing and routing from other partitions. The only routing allowed inside a partition is the internal routing of the partition itself and the routing defining the interface between the static and reconfigurable partition. A relocated partition must not be in position to destroy any routing from the static logic or other partitions.
In Figure 2.3 pblock #1 has only interface routing while pblock #2 and #3 have pass through routes not belonging the their partitions.

3 Design and Layout

The basic idea of relocatable partitions is to design a hardware layout where every partial reconfiguration (PR) can be exchanged with each other without damaging the static design and the partitions. This guide will assume a base design with a static design always present on the FPGA and multiple Partial Reconfigurations (PR). All PRs are connected to the static design through an interface for exchanging data. Shared resources like external ports have to be managed through the static design because they are not relocatable in the PRs. All PRs must share the same interface definition.

3.1 Hardware Design

In the normal PR Workflow a direct connection from PR to the static design would be sufficient because Vivado takes care of all timing constraints. Through relocation the implemented PR can be placed in every PR Partition with varying signal paths from static to PR. Leaving the responsibility of timing to the PR Designer would destroy the benefit to not have to check every possible implementation. Also relocation allows to change the static design without having to implement the PRs again, which would change the timing behaviour every time.
To solve these problems there has to be placed a register buffer in front of every PR Partition to offer the best timings for the PR Designers.

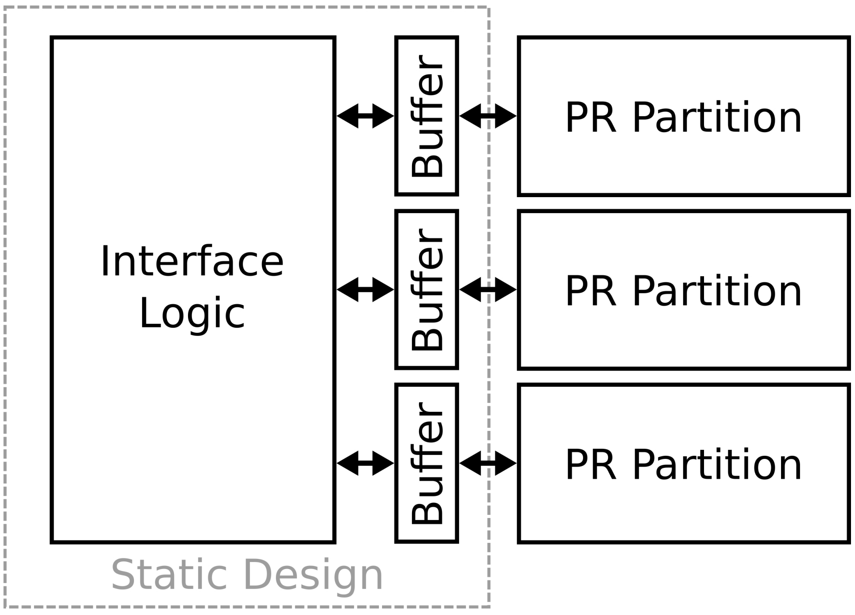

Figure 3.1: Example Cell Layout for Relocatable Partitions

Another advantage of placing a buffer in front of the PR Partitions is a more flexible static design. Every partition interface will be absolutely fixed in location and routing on the PR and static side. By placing the Static Interface in the buffers in front of the PRs the rest of the static design is free from fixed constraints.

For designing the Relocatable Partitions it is highly recommended to fill the PR Partitions with the most simple design, for example a loopback buffer which is simply throwing back what is coming in.
To fix the interface later, it must be clearly defined by a start and end point which will be primitive leaf cells (LUT or REG). The best way to define these interface cells is to add another buffer, but this time with LUT cells.

Declaring a LUT1 primitive component in Vivado is easy:

```
component LUT1
    generic (
        INIT: bit_vector(1 downto 0) := "10"
    );
    port (
        O : out std_logic;
        I0 : in std_logic
    );
end component;
```

The interface LUT buffer has to be added on the Static and PR side with no logic inbetween. Contrary to the register buffers these LUT buffers will not interfere on interfaces with flow control.

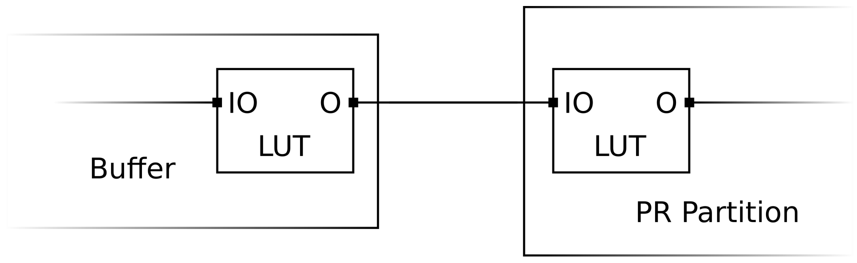

Figure 3.2: Placement of LUT buffers between Static Design and PR Partitions

The likelihood of getting a valid relocatable partition layout at the end, depends on the size of the interface and its layout which will be done in Section 4.2. Since this whole project is about to force the router and placer to do something they were not meant for, they sometimes reject to implement the generated constraints at the end. Placing the interface on only one column, the limit of around 135 signals reduces the probability of a successful implementation. But that depends highly on the placement of the interface and the layout of the PR Partition and needs further examinations.

3.2 Partition Layout

Before starting drawing pblocks in the synthesized design, care should be taken that throughout the implementation process the *Isolation Design Flow* (IDF) is used and all top level cell interfaces have to cross the fences between the pblocks. It is also not recommended to insert any global logic outside isolated cells without the control over placement and routing.

The first cell to position is the interface logic of the static design. By using a 7 Series FPGA it is typically the cell with the processing system and all external ports, so it needs a pblock covering all needed resources. Special care must be taken with clock signals. The pblock of the cell which drives clock signals needs to cover the BUFGCTRL resources. This pblock for the interface logic must be set as isolated. To allow the global routing of clock signals from and to isolated cells these components must be marked with a special isolation property:

```
set_property HD.ISOLATED_EXEMPT true [get_cells -hierarchical -filter
    {PRIMITIVE_TYPE =~ CLK.gclk.*}]
```

Next the PR Partitions with their buffers will be placed. It is not quite easy to find regions with the correct properties for relocatable partitions because the buffers also need special placement constraints. In this guide all PR Partitions will be fully compatible, but as long as the properties from chapter two are all fulfilled the relocation will work. Additional to fully equal PR Partitions in respect to resources and their relative placement, the buffers in front of the PR partitions need to have the same properties.

The easiest way to define those partitions is to find multiple regions with the mentioned properties, make them as high as a clock region and as wide as possible. These regions are the PR Partitions. The buffers will be placed on the interface side of the PRs. Resizing the PR pblocks some columns down on the interface side will give some space for the buffers with the exact same relocation properties. Leaving some space between all pblocks for the fences is important for a cleaner routing result at the end.

The procedure verify_relocateable_pblocks 💬 will check the given pblocks if they are compatible with each other.

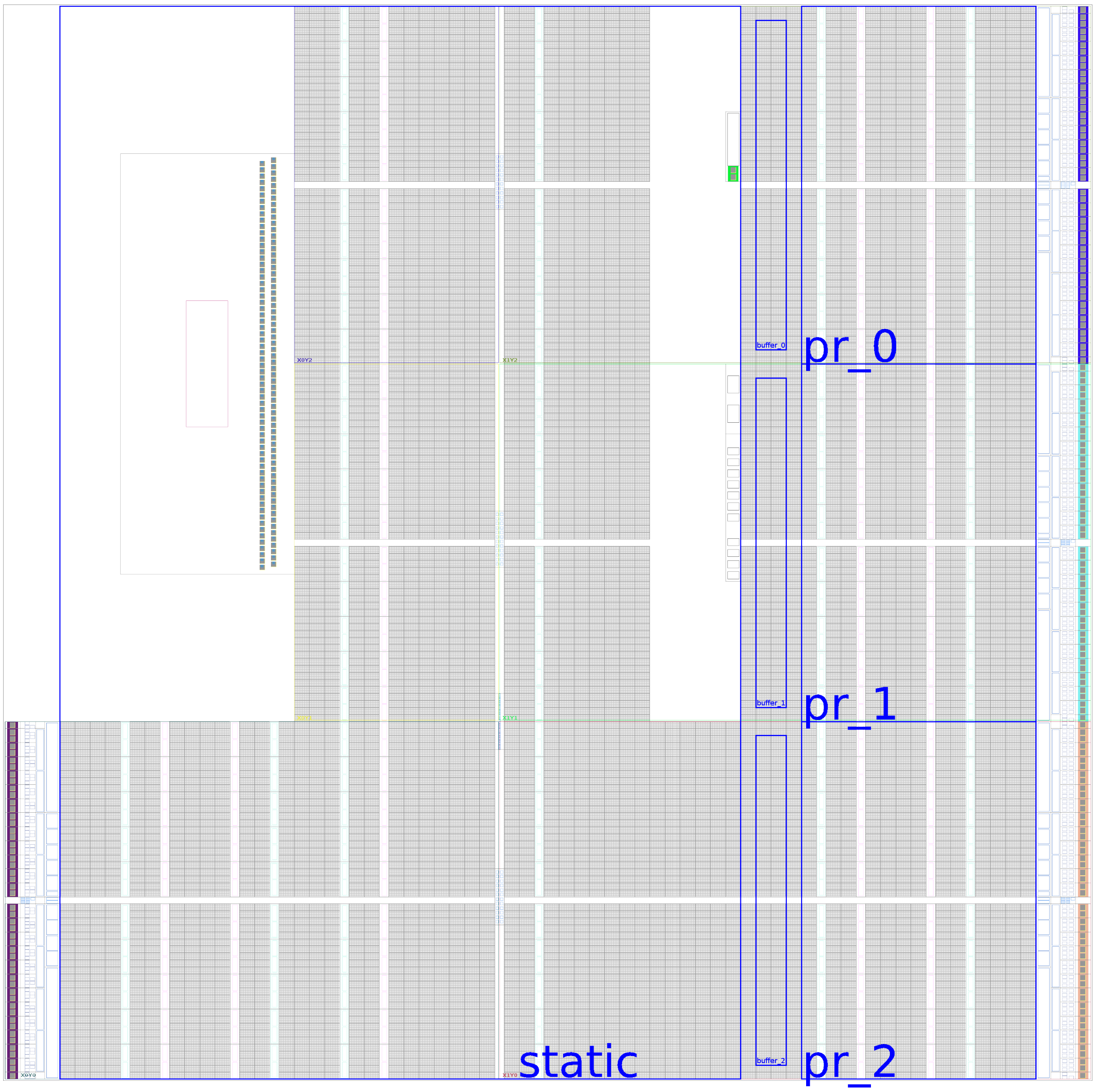

Figure 3.3: Example pblock layout for 3 relocatable partitions on a ZedBoard

3.3 ZedBoard Example

The project in the *zedboard_example* folder is a full relocatable partitions example for the ZedBoard. Starting with the interface logic it is easy to use a block design for the processing system connected to 3 DMA engines. Each DMA engine has a slave and master AXI Stream interface which will be used for communicating with the PR Partitions. In this example a datawidth of 32 bit is used, which produces 78 signals per PR Interface.

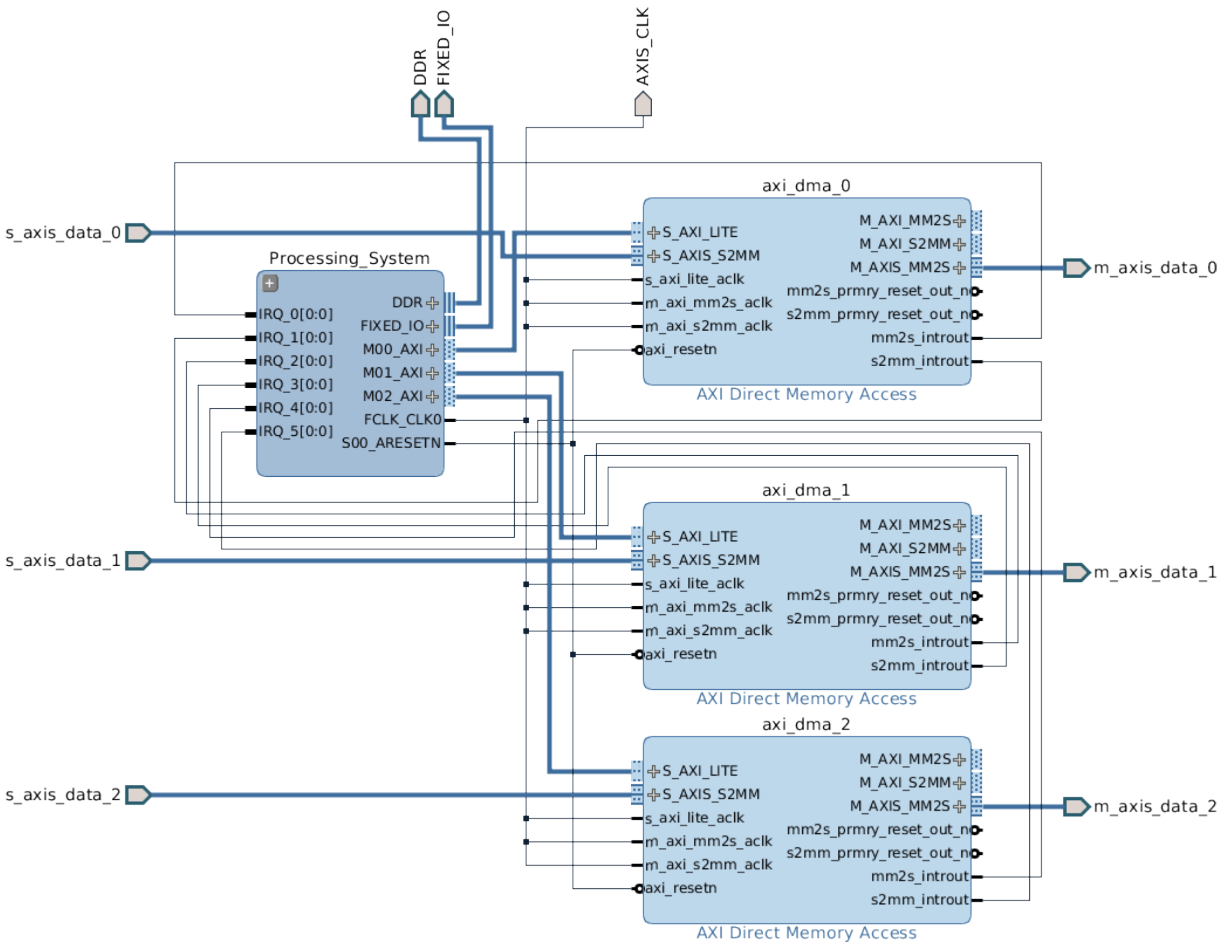

Figure 3.4: ZedBoard Block Design representing the Interface Logic with the Processing System and 3 DMA Engines

The rest of the hardware design will be designed through VHDL, therefor the AXIS and AXIS clock ports need to be external ports in the Block Design.

In this example an implementation of the buffer and LUT buffer for the AXIS Interface is provided in *axis_buffer.vhd* and *axis_lut_buffer.vhd*. The placement and isolation needs single top level cells under the top wrapper, that is why it is not possible to build the complete hierarchy with these components inside the top level wrapper.

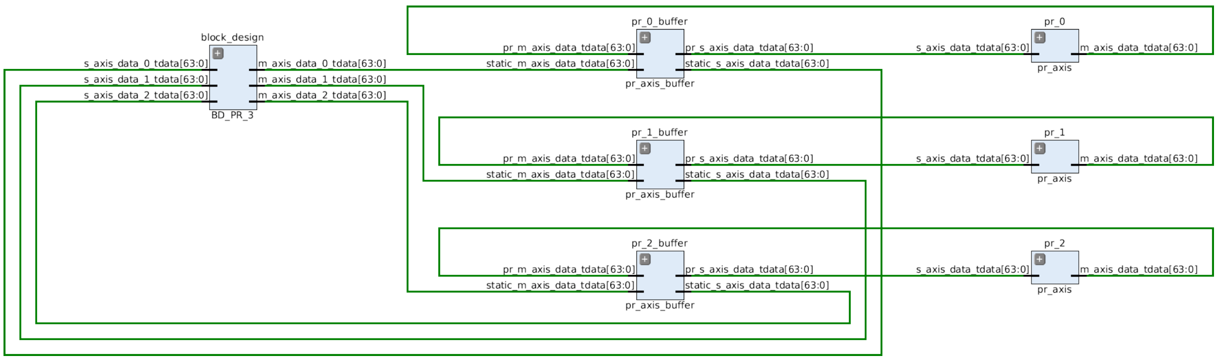
block_design
s_axis_data_0_tdata[63:0]
s_axis_data_1_tdata[63:0]
s_axis_data_2_tdata[63:0]
m_axis_data_0_tdata[63:0]
m_axis_data_1_tdata[63:0]
m_axis_data_2_tdata[63:0]
BD_PR_3
pr_0_buffer
pr_1_buffer
pr_2_buffer
pr_m_axis_data_tdata[63:0]
static_m_axis_data_tdata[63:0]
pr_s_axis_data_tdata[63:0]
static_s_axis_data_tdata[63:0]
pr_axis_buffer
pr_0
pr_1
pr_2
s_axis_data_tdata[63:0]
m_axis_data_tdata[63:0]
pr_axis

Figure 3.5: Reduced schematic of the full design. Block Design is connected to the PR Buffer Cells, which are connected to their PR Cells (only AXIS Data Nets)

Figure 3.5 is a reduced schematic of the top level cells inside the top wrapper. All these cells will be isolated from each other having their own pblock. The Block Design on the left (from Figure 3.4) combines the interface logic and connects to the PR Buffer Cells which are directly placed in front of the PR Cells. Every net from the PR Cells goes through this buffer to the interface logic.

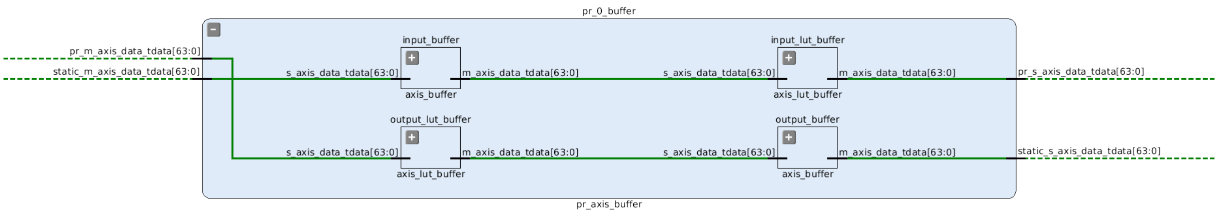

Figure 3.6: Reduced schematic of a PR Buffer Cell (only AXIS Data Nets)

The PR Buffer Cell from Figure 3.6 consist of 2 AXIS Register Buffers and 2 AXIS LUT Buffers. The names of the nets was chosen in respect to the cells which they connect to. So "pr_s" is the Slave PR Interface and "static_s" the Slave Static Interface.

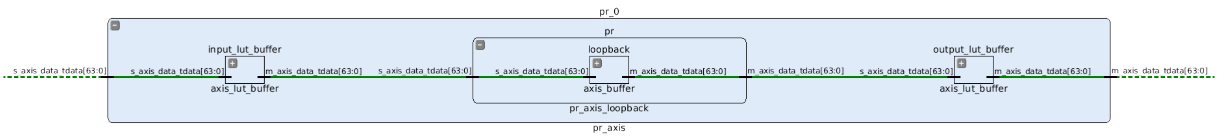

Figure 3.7: Reduced schematic of a PR Cell with a Loopback Interface (only AXIS Data Nets)

The PR Cell from Figure 3.7 consists of 2 AXIS LUT Buffers and the true PR Cell itself. In reality the whole PR Cell gets reconfigured but the only changing part in the PR Partition will be the nested PR Cell inside. This nested PR Cell consists of a AXIS Register Buffer representing the Loopback Interface.

As seen on the figures the AXIS LUT Buffer is the last and first cell of a net going from the static part to the PR Partition. This is absolutely important for fixing the interface later because the starting and ending cells of fixed nets have to be well defined and choosing LUT Buffers for this will not change the timing behaviour for flow controlled Interfaces like AXIS.

The last point before starting the relocation process is to define the physical FPGA layout with pblocks. Like mentioned before every top level cell in the top wrapper need their own pblock. Nested pblocks are not allowed in the PR and IDF Workflow. The design of the pblock is described in Section 3.2 and can be seen in the wrapper constraint set of the example project or Figure 3.3.

Every top level cell needs the ability to be isolated for the Isolation Design Flow. The scripts from the next chapter will take care of the PR Partitions but not of the rest. Attention should be given to the clock ports, they need a special isolation flag. For this exmaple the isolation constraints look like this:

```
set_property HD.ISOLATED true [get_cells pr_0]
set_property HD.ISOLATED true [get_cells pr_1]
set_property HD.ISOLATED true [get_cells pr_2]
set_property HD.ISOLATED true [get_cells pr_0_buffer]
set_property HD.ISOLATED true [get_cells pr_1_buffer]
set_property HD.ISOLATED true [get_cells pr_2_buffer]

set_property HD.ISOLATED true [get_cells block_design]
set_property HD.ISOLATED_EXEMPT true [get_cells -hierarchical -filter
    {PRIMITIVE_TYPE =~ CLK.gclk.*}]
```

It is a good idea to implement the just created design to check for errors and compatibility to the Isolation Design Flow. If no errors or critical warnings appear the created design is ready for the next chapter.

4 Relocation

The created design is now ready for designing the relocatable partitions. The main idea is to define the interface to one PR Partition. This interface will be fixed in all details and then copied to all other PR Partitions. The process of copying the interface requires that all resources are at the same relative positions which is now assured through the previous chapter.

The relocation process is heavily scripted and should not be done manually because there will be thousands of constraints generated. The required procedures are included in *procs.tcl* and documented in chapter 5. To follow this guide this script should be copied to the current project folder and sourced in the Vivado Console:

```
source -notrace procs.tcl
```

4.1 Preparation

The current constraint set with all pblocks and isolated cells should be saved in a new constraint set in which all changes will be made. This line will create a new constraint set out of the current with a new file reloc.xdc as target constraint file:

```
save_constraints_force reloc reloc.xdc
```

The design needs to be synthesized and opened for the placement of the interface:

```
synthesize
open_synth
```

Now all information can be gathered for placing the interface:

```
set pr_base [get_cells pr_0]
set pr_cells [get_cells {pr_1 pr_2}]
```

The pr_base is the PR Partition on which the interface will be defined and from where the interface will be copied to the other PR Partitions in pr_cells. And that is basically the preparations. If the design is made the same way as in chapter 3 the scripts can manage the rest.

4.2 Placing the Interface

The interface between the Base PR Partition and its buffer is defined through the LUT Buffers and the routes between them. The LUT cells can be easily retrieved through the start and end cells of the boundary nets.

```
set boundary_nets [get_boundary_nets $pr_base]
set intf_start [get_nets_start_cells $boundary_nets]
set intf_end [get_nets_end_cells $boundary_nets]
set pr_intf_start [get_cells $intf_start -filter NAME=~$pr_base/*]
set pr_intf_end [get_cells $intf_end -filter NAME=~$pr_base/*]
set buffer_intf_start [get_cells $intf_start -filter NAME!~$pr_base/*]
set buffer_intf_end [get_cells $intf_end -filter NAME!~$pr_base/*]
```

The get_boundary_nets procedure retrieves all boundary nets of the Base PR Cell which are all nets between the PR Buffer and the PR Partition. The procedures get_nets_start_cells and get_nets_end_cells return the start and end cells of the boundary nets, which are the LUT Buffers from the PR Buffer and PR Partiton. The cells intf_start and intf_end have to be split in PR Cells and PR Buffer Cells for placement. This can easily be done with the filter parameter from get_cells.
The relations between the interface cells are the following:

- pr_intf_start → buffer_intf_end
- buffer_intf_start → pr_intf_end

Since the interface from the PR Buffer to the PR Partition is the same, the assumption is made that the same LUT Buffers were used and the LUT Cells are named equally. This will allow to map the interface lists and place the LUT Cells towards each other. Here is an example relation of the LUT cells:

- pr_0_buffer/input/LUT_BUFFER → pr_0/input/LUT_BUFFER
- pr_0/output/LUT_BUFFER → pr_0_buffer/output/LUT_BUFFER

For this relation the procedure match_lists can be used to map the Interface Cells of the PR Cell to the PR Buffer. If the hierarchy or naming is different a custom mapping method has to be used.

```
match_lists $pr_intf_start buffer_intf_end
match_lists $buffer_intf_start pr_intf_end
```

This procedure tries to sort list two to match list one. A third parameter is the strip_filter which defines how much from each element is compared. The standard filter stips off the first element of the cell, so that from the example above the "pr_0" and "pr_0_buffer" part is removed for comparison. This procedure alters list two to match list one and returns 0 if everything matched.

Since all interface cells are found and matched, the lists can be put together for placement. Attention must be paid so that the order does not get destroyed.

```
set pr_intf [concat $pr_intf_start $pr_intf_end]
set buffer_intf [concat $buffer_intf_end $buffer_intf_start]
```

When thinking about the placement of the Interface Cells it is important to find a layout that works at the end. Sadly there is no exact recipe for this and the outcome can only be seen at the very end.
In this section the interface will be placed naively at the top of one column. A different approach can be seen in the *pr_script.tcl* script or Section 4.5.

It is absolutely important to not use all LUT resources in one column, leaving more space for the router to do its job. A more even distribution of the LUT resources can also help. Bigger interfaces with more than 135 signals should use a multi column layout for the interface because one column will mostly fail.

Before getting the LUT resources out of the slices, the pblocks have to be retrieved from the PR Cell and PR Buffer Cell.

```
set pr_pblock [get_pblocks -of_objects $pr_base]
set buffer_pblock [get_pblocks -of_objects $buffer_intf]
```

One slice has 4 LUT6 resources, so 4 LUT buffers can be placed in one slice. It is recommended to use one column or row of slices to place the interface, but only if the interface is not too big. The next lines retrieve the slices on the left side of the PR Partition and the slices on the right side of the PR Buffer. An example for all four sides can be found in script *pr_script.tcl*.

```
#Slice Range e.g. SLICE_X0Y0:SLICE_X10Y10
set pr_slice_range [get_range_from_pblock $pr_pblock SLICE]
set buffer_slice_range [get_range_from_pblock $buffer_pblock SLICE]

#Slice List e.g. SLICE_X0Y0 SLICE_X1Y0....
set pr_slices [get_list_from_range $pr_slice_range]
set buffer_slices [get_list_from_range $buffer_slice_range]

#Left and Right Column
set pr_col [lindex [get_cols $pr_slices] 0]
set buffer_col [lindex [get_cols $buffer_slices] end]

#Filter Columns from Slices
set pr_intf_slices [filter_tiles $pr_slices SLICE $pr_col]
set buffer_intf_slices [filter_tiles $buffer_slices SLICE $buffer_col]
```

To retrieve the LUT6 resources from the just filtered slices get_bels_lut6 can be used. The order of the returned list will be used for placement later. If a different placement (starting from top or bottom) is wanted the lists can be sorted with lsort.

```
set pr_luts [get_bels_lut6 [get_sites $pr_intf_slices]]
set buffer_luts [get_bels_lut6 [get_sites $buffer_intf_slices]]
```

Now that the interface cells and LUT resources are known they can be placed using place_cells. This procedure will place the given cells on the given BELs regarding the order of the lists. This generates the BEL (Basic Element of Logic) and LOC (Location) constraint for all LUT Cells and sets automatically the IS_BEL_FIXED and IS_LOX_FIXED property.

```
place_cells $pr_intf $pr_luts
place_cells $buffer_intf $buffer_luts
```

In the process of implementation opt_design will optimize the design and remove everything that is not necessary in the opinion of vivado. This involves nearly all LUTs which are placed right now. To prevent the optimizations of the added LUTs the property DONT_TOUCH must be true.

```
set_property DONT_TOUCH true $pr_intf
set_property DONT_TOUCH true $buffer_intf
```

Before creating all routes between the LUTs the PartPin LOCs need to be assigned to the PR Partition Pins. The procedure fix_plocs will place the PartPin LOC to the interconnect tile from the fixed cell (in this case the LUT) of a Partition Pin.

```
fix_plocs [get_pins ${pr_base}/*]
```

All created constraints can be viewed and checked in the device view of the opened design as seen in Figure 4.1.

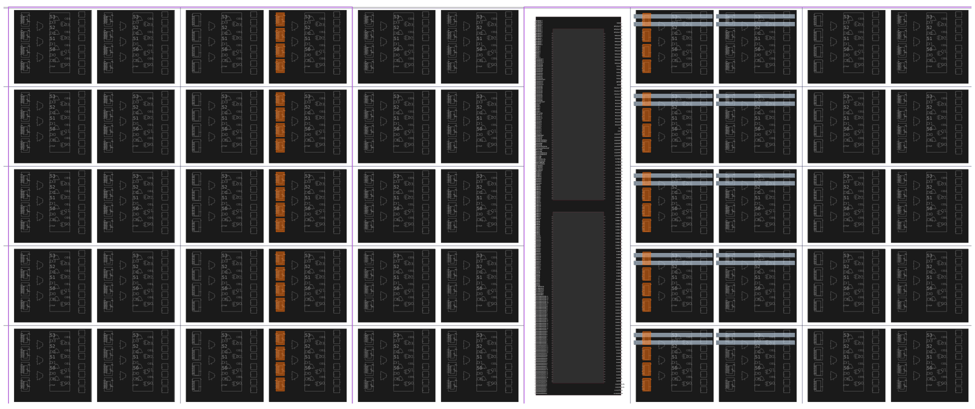

Figure 4.1: Device view of a placed Interface with PartPin LOCs

Before implementing the placed interface, the Base PR Partition must be isolated and the just created constraints have to be saved to the current constraint set.

```
set_isolated ${pr_base}
save_constraints_force "" ${pr_base}_intf.xdc
```

4.3 Generate and Fix Routing

In this section the routing will be generated and fixed. The script for that is only some lines long but it is absolutely important that special attention is given to verify that the generated routing is compatible to relocation.

After implementing the design with the placed interface from the Base PR Partition the routing will be created. Since using the Isolate Design Flow the routes of the other PR Partitions can be ignored, they will not touch the Base PR Partition. More important is the routing between the Interface Logic, the Base PR Buffer and the Base PR Partition.

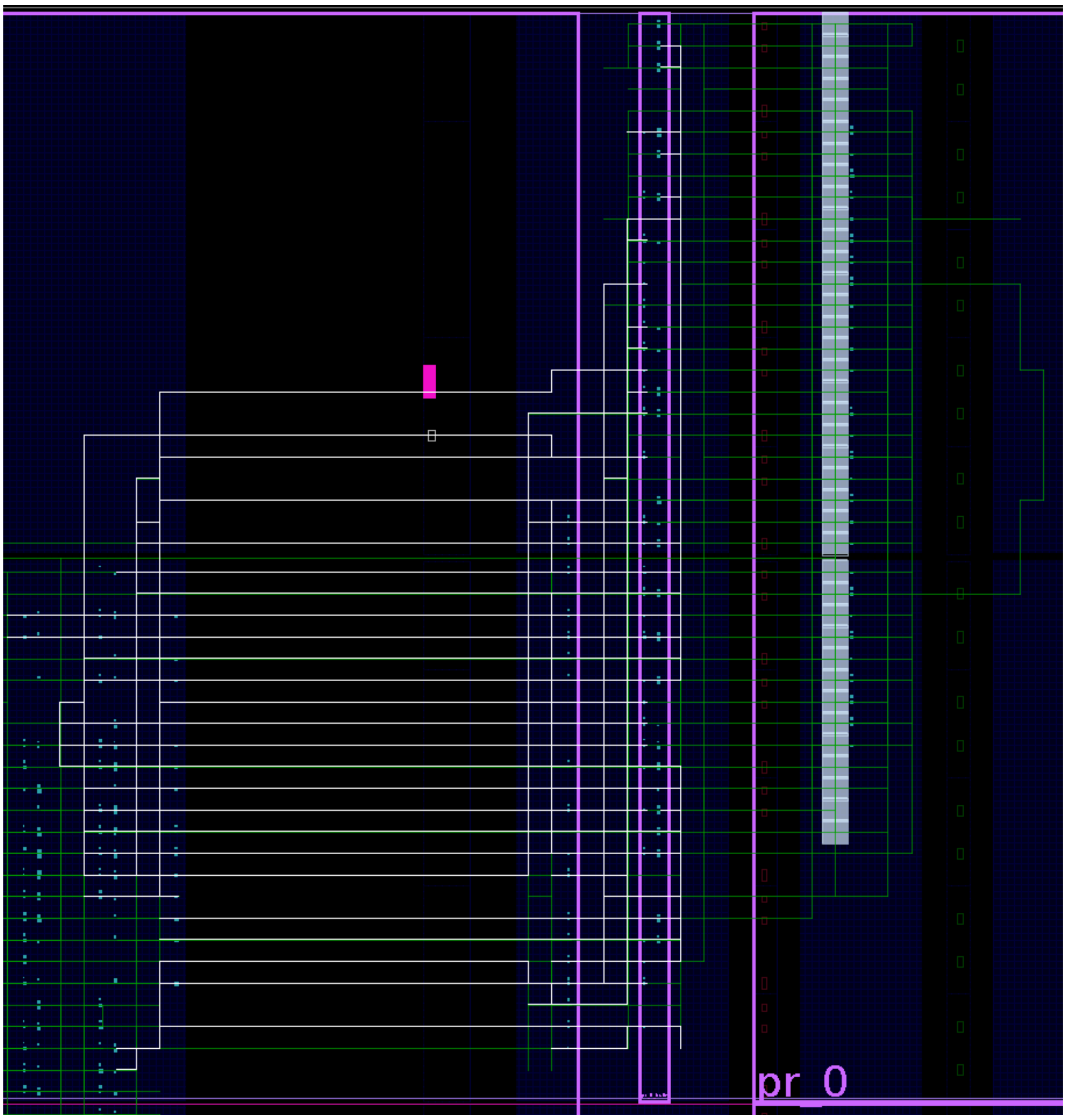

Figure 4.2: A routed Interface for the Base PR Partition with selected routes between the Interface Logic and the PR Buffer.

For relocation of a PR Partition it is important that only the PR related routes touch the PR pblock. The PR Partition is marked as isolated, which means that all routing inside the PR is contained and not leaving the partition. This behaviour is the same as in the PR Workflow. The only routing that is allowed to leave the PR Partiton is the interface. In the Isolation Design Flow the routing between two isolated cells is called trusted routing which traverses the fence between the pblocks of the cells. The fence is an empty region not covered by any pblocks. Inside the fence the routing is not allowed to use any routing resources, so the routes are absolutely straight between the cells and not touching any interconnect tiles inside the fence. These rules also yield to a maximum fence width of 6 slices. The router will fail if isolated cells are more than 6 slices apart from each other.

Trusted Routing from the Isolation Design Flow would be perfect for relocatable PR Partitions. It guarantees that only interface routing crosses a fence without the use of extra routing resources outside the pblocks. In theory this means that the routes from the Interface Logic would cross the fence and then end up in the PR Buffer, but as seen in Figure 4.4 that is not the reality. The "trusted" routing generated by the IDF with the placed PR Interface is far away from the promised routes and that is because of the constraints that were added.
The real trusted routing would be generated if the placer and router are absolutely free in choice of placement and routing. But then the generated interface would be perfect for the current implementation and not for every PR Partition later. So why using IDF if there are no trusted routes? Even if the routing is not trusted after constraining the interface, the routes look much better than without IDF. It can be said that the router and placer tries to follow the IDF rules.
The most "destroying" constraint is the PartPin LOCs constraint which is not necessary for IDF but for PR. Adding them after the generation of the routes was not possible because the routes have some unknown dependecies on the partition pins in the PR Workflow.

To verify that the routing does comply with all rules for relocatable PR Partitions the generated routing to the base PR Partition has to be checked manually. Most of the time the routing should be fine to continue. In some rare cases the routing must be fixed to not cross the PR Partition.
To highlight the important routes these two lines can be used to mark the boundary nets inside the implemented device view:

```
select_objects [get_boundary_nets interface_logic]
select_objects [get_boundary_nets ${pr_base}]
```

If no routes between the cells touch unrelated cells the PR Interface can be fixed with 3 lines:

```
set boundary_nets [get_boundary_nets $pr_base]
fix_lut_pins $boundary_nets
fix_routes $boundary_nets
```

The fix_lut_pins is solving a missing feature in the Vivado GUI. On a fixed route the starting and ending point needs to be known and fixed. If these cells are registers nothing needs to be done because the input and output pins of a specific register are always known (D and Q). On LUT Cells the input and output pins can vary (A1, A2...). The property LOCK_PINS on a primitive LUT cell is defining the input and output pin. This procedure is getting all LUT cells to the given nets and fixes the pins that are used in the implemented design.
The fix_routes procedure does basically the same as the GUI but for all given nets at the same time. It is going through the nets, finds the parent if necessary (only for alias nets) and fixes the implemented route to their already fixed start and endpoints. This generates the FIXED_ROUTE constraint.

```
save_constraints_force "" ${pr_base}_routes.xdc
```

The generated constraint needs to be saved for implementing the fixed interface again. This implementation run will verify if the router and placer accepts the new constraints. If it does not it sometimes helps to unfix the affected routes, route them again and fix them.

```
refix_routes [get_failed_nets]
```

In the next section the fixed PR Interace will be copied to all other PR Partitions.

4.4 Copy PR Interface

The last step for designing relocatable PR Partitions is to copy the generated and fixed PR Interface from the Base PR Partition to all other compatible PR Partitions. It is important to open the synthesized design for this step because the other PR Partitions have no constraints and are optimized which may result in missing nets/cells.

```
relocate_cells $pr_base $pr_cells
```

The relocate_cells procedure is doing all the work for relocating the PR Interface to other cells. The first step is to clean up the nets and cells of the target PRs from all constraints. Then the procedure gathers all information about nets and cells of the base and target PRs and compares them. If the target PR Cell does not match it will not copy anything to it. Since the PR Partitions are absolutely the same in resources and relative locations

there are only two properties which have to be handled in a special way. These are the LOC and PARTPIN_LOCS properties because they point to specific slice and interconnect tiles. The PARTPIN_LOCS property is commented out in relocate_cells because the exact same job will do fix_plocs afterwards. At first all interface cells will be relocated. The properties LOCK_PINS, BEL, IS_LOC_FIXED, IS_BEL_FIXED, DONT_TOUCH can be copied without anything else to do. The LOC property needs to be relocated relative the the new pblock location. This is done through copy_object_properties with a special property parsing procedure relocate_loc_property which is doing the calculations.
After relocation the PartPin Locs have to be placed for the new PR Cells:

```
foreach cell $pr_cells {
        fix_plocs [get_pins_no_clk ${cell}/*]
}
```

Now all PRs can be set as reconfigurable to finish the relocation process and get real partial reconfigurable partitions in the design

```
set_reconfigurable $pr_base
set_reconfigurable $pr_cells
```

Saving the new constraints:

```
save_constraints_force "" pr_others.xdc
```

The last implementation of the design will verify if everything is working. Sometimes the router is not able to place or route some resources inside the PRs. Fortunately that is a rare case and only happens if the interface is too big for the chosen interface layout, but if it happens it can be helpful to resize the PRs. Making them a little bit smaller or bigger (only if possible) can resolve these problems. Otherwise there is no other way than starting all over again with another placement of the interface. Therefor scripting the whole process is very helpful and an exmaple can be found in *pr_script.tcl*

If everything goes well a relocatable partitions design was found and can be used to implement PRs. As long as the fixed interface to all PRs stays in place, the static design and every PR can be modified and they will always be compatible to each other.

4.5 ZedBoard Example

Starting directly after Section 3.3, the PR Interface will be placed in a special way. The placement will be on the left side but starting in the middle. This will give a cleaner look of the interface. But at first the synthesized design needs to be opened.

```
open_synth
save_constraints_force reloc isol.xdc
```

Then setting the Base PR Partition, retrieving the boundary nets and their starting and ending cells:

```
set pr_base [get_cells pr_0]
set boundary_nets [get_boundary_nets $pr_base]
set intf_start [get_nets_start_cells $boundary_nets]
set intf_end [get_nets_end_cells $boundary_nets]

set pr_in_cells [get_cells $intf_end -filter NAME=~$pr_base/*]
set pr_out_cells [get_cells $intf_start -filter NAME=~$pr_base/*]
set buf_out_cells [get_cells $intf_end -filter NAME!~$pr_base/*]
set buf_in_cells [get_cells $intf_start -filter NAME!~$pr_base/*]
```

These lists have to be put into the correct order that every LUT cell has their matching counterpart. This is done through the naming of the LUT Entities which is the same in the PR Buffer and PR Partition.

```
match_lists $pr_in_cells buf_in_cells
match_lists $pr_out_cells buf_out_cells
```

The interface will be on the left side of the PR Partition. Therefor all slices in the far left column of the PR Partition and all slices on the far right column of the PR Buffer need to be gathered.

```
set pr_pblock [get_pblocks pr_0]
set buf_pblock [get_pblocks pr_0_buffer]

set pr_range [get_range_from_pblock $pr_pblock SLICE]
set buf_range [get_range_from_pblock $buf_pblock SLICE]
set pr_slices [get_list_from_range $pr_range]
set buf_slices [get_list_from_range $buf_range]

#Interface on the left
set pr_pos [lindex [get_cols $pr_slices] 0]
set buf_pos [lindex [get_cols $buf_slices] end]

set pr_slices [filter_tiles $pr_slices SLICE ${pr_pos}]
set buf_slices [filter_tiles $buf_slices SLICE ${buf_pos}]
```

Since the placement of the inteface should start from the middle the retrieved slices will be cut in half and the buttom part will be sorted in reverse.

```
set slice_count [llength $pr_slices]
set pr_in_slices [get_sites [lrange $pr_slices 0 [expr $slice_count /
    2] ]]
set pr_out_slices [lsort [get_sites [lrange $pr_slices [expr (
    $slice_count / 2) + 1] $slice_count]]]

set slice_count [llength $buf_slices]
set buf_in_slices [get_sites [lrange $buf_slices 0 [expr $slice_count
    / 2]]]
set buf_out_slices [lsort [get_sites [lrange $buf_slices [expr (
    $slice_count / 2) + 1] $slice_count]]]
```

To place LUT Cells the LUT6 resources of the slices need to be retrieved.

```
set pr_in_bels [get_bels_lut6 $pr_in_slices]
set pr_out_bels [get_bels_lut6 $pr_out_slices]

set buf_in_bels [get_bels_lut6 $buf_in_slices]
set buf_out_bels [get_bels_lut6 $buf_out_slices]
```

Now the placement can start with the LUT Cells and the LUT6 resources. Afterwards the cells have to be marked with DONT_TOUCH to prevent opt_design from removing them.

```
place_cells $buf_in_cells $buf_in_bels
place_cells $buf_out_cells $buf_out_bels
place_cells $pr_in_cells $pr_in_bels
place_cells $pr_out_cells $pr_out_bels

set_property DONT_TOUCH 1 $buf_in_cells
set_property DONT_TOUCH 1 $buf_out_cells
set_property DONT_TOUCH 1 $pr_in_cells
set_property DONT_TOUCH 1 $pr_out_cells
```

As last step for the PR Interface the PartPin LOCs need to be placed.

```
fix_plocs [get_pins ${pr_base}/*]
```

Saving the generated constraints to a new file and synthesize the design to view the updated device view.

```
save_constraints_force "" ${pr_base}_intf.xdc
synthesize
open_synth
```

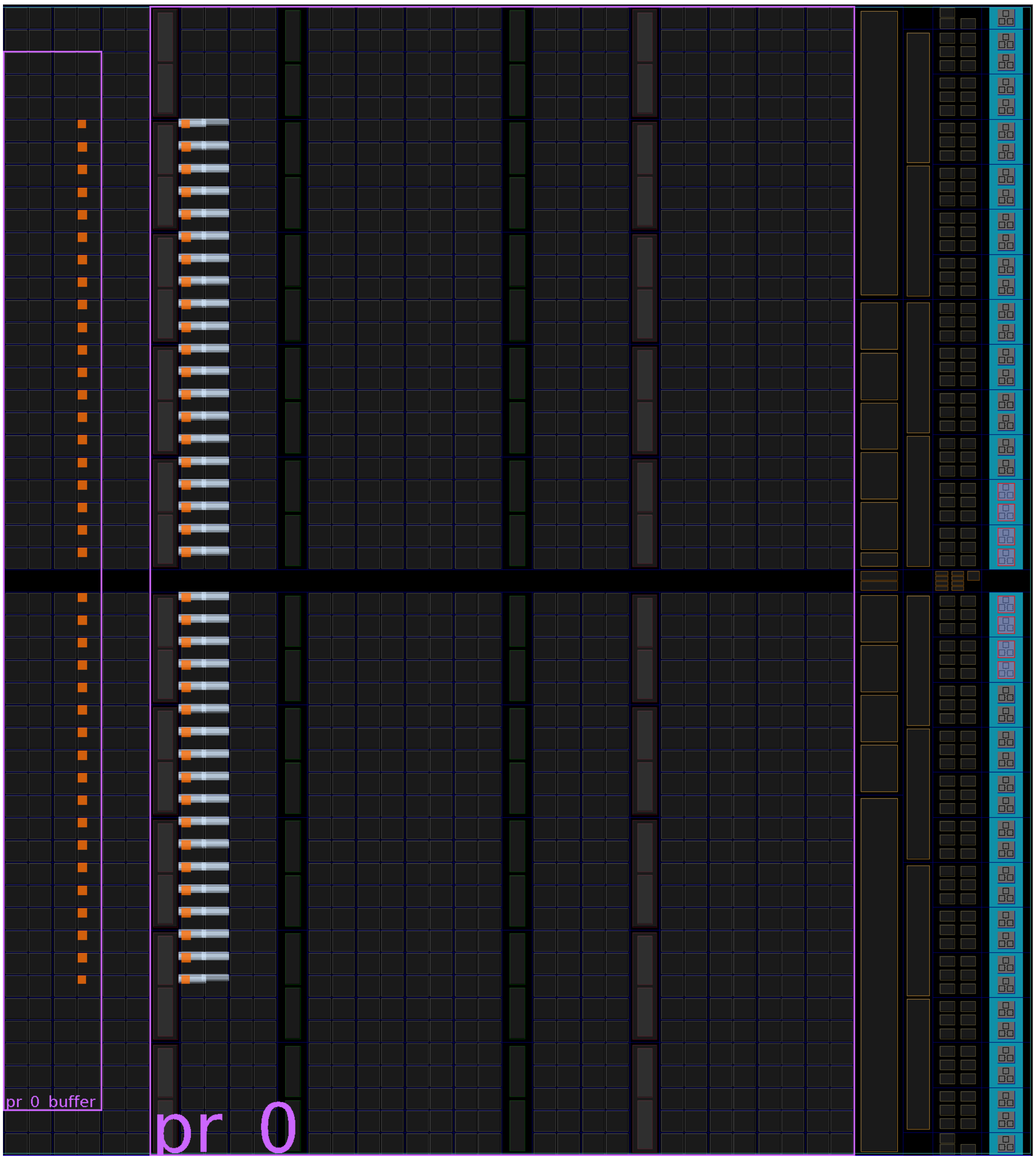

Figure 4.3: A completely placed Interface in the synthesized device view.

In the next step the routing will be generated. Therefor the design must be implemented with the interface constraints.

```
implement
open_impl
```

In this example the routing is compatible with the rules of relocatable PR Partitions. If the exmaple was changed read Section 4.3. The generated routing can be fixed with the next lines.

```
set boundary_nets [get_boundary_nets ${pr_base}]
fix_lut_pins $boundary_nets
fix_routes $boundary_nets
```

Now the interface is complete and the new constraints can be saved to the constraint set. Implementing the design again to make sure that the placer and router accept the constraints. After this the synthesized design needs to be opened for the next step.

```
save_constraints_force "" ${pr_base}_routing.xdc
implement
open_synth
```

The created and fixed interface needs to be copied from the Base PR Partition to all other PRs which are compatible. Afterwards create the PartPin LOCs to the copied interfaces.

```
relocate_cells $pr_base [get_cells {pr_1 pr_2}]
fix_plocs [get_pins pr_1/*]
fix_plocs [get_pins pr_2/*]
save_constraints_force "" pr_others.xdc
```

The last step is to set the PRs as reconfigurable.

```
set_reconfigurable $pr_base
set_reconfigurable [get_cells {pr_1 pr_2}]
save_constraints_force "" isol.xdc
```

After the successful implementation of the generated constraints the design is ready for implementing partial reconfigurations. These reconfigurations will be fully relocatable by rewriting the frame addresses inside the bitstreams.

The implemented design with the relocated interfaces can be seen in Figure 4.4.

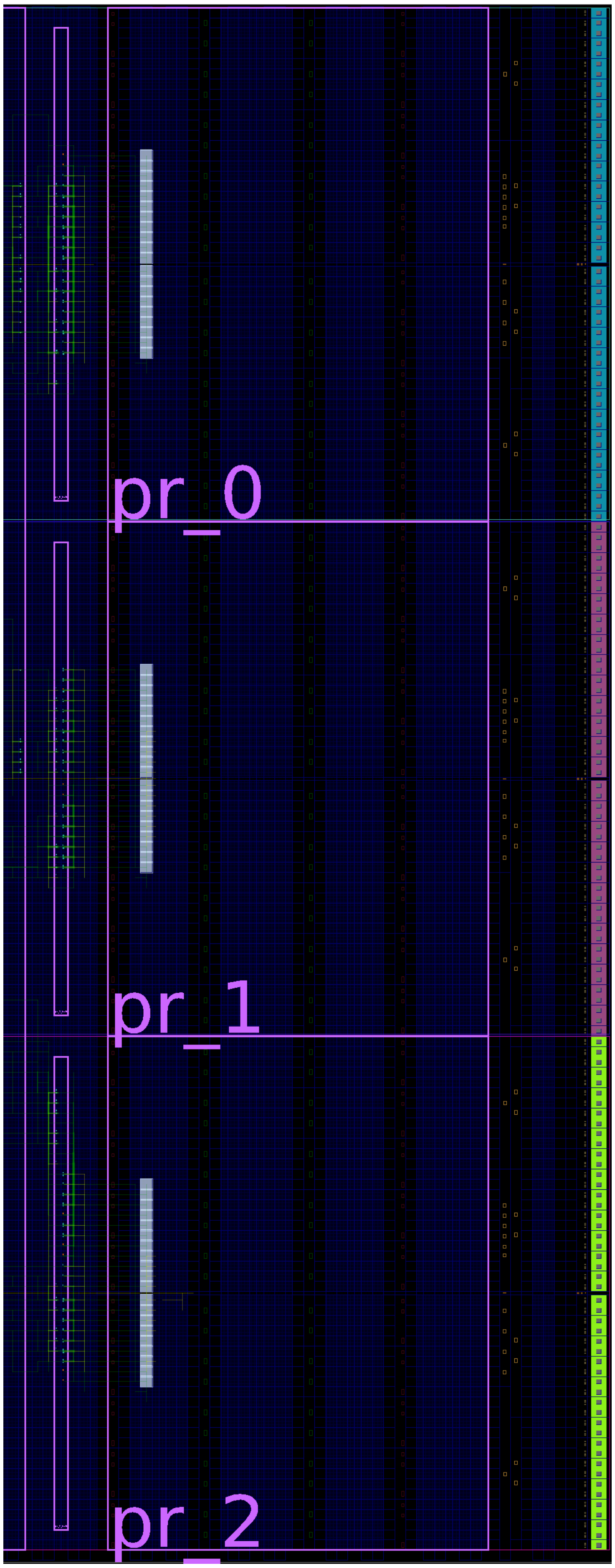

Figure 4.4: A finished Design with 3 fully relocatable Partitions

5 Procedures Documentation

The functionality of Vivado gets extended by sourcing the file *procs.tcl* shipped with this guide. This chapter is a short summary for the most important procedures.

Procedure	Parameters	Description
save_constraints_force	(o) constraint_set (o) target_file	Saves Constraints to a given constraint set. If no constraint set was passed the current constraint set is used. Target constraint file will be created if necessary.
verify_relocateable_pblocks	pblocks	Checks all given pblocks if they are compatible with each other, outputs the resources of each pblock while checking. Returns 0 on success. Does not check relative location from resources over different types.
get_nets_start_cells	of_nets	Returns all start cells of the given nets.
	of_nets	Returns all end cells of the given nets.
get_interconnects	from	Retrieves all interconnects from given tiles or sites.
fix_routes	of_nets (o) state	Fixes all given nets, the start and end cells. Retrieves parent net if necessary. State 0 will unfix the given nets.
fix_lut_pins	of_nets (o) state	Fixes all connected LUT Pins of the given nets. State 0 will unfix the LUT Pins of the given nets.
fix_plocs	of_pins (o) state (o) col_offset (o) row_offset	Place PARTPIN_LOCS to the given pins with optional column and row offset. If state is 0 it will reset the PARTPIN_LOCS property.

Parameters: (o) optional, (r) reference

Procedure	Parameters	Description
get_range_from_pblock	pblock (o) type	Retrieves the resource ranges from a pblock. Optional retrieves only a specific resource type.
filter_tiles	tiles (o) regex_tile (o) regex_col (o) regex_row	Filters a list of tiles with regular expressions for tile name, column and row
get_list_from_range	ranges	Transforms range expressions to lists. Does respect INT_L and INT_R relations. e.g. SLICE_X0Y0:SLICE_X10Y10
get_cols	tiles	Returns the columns of the given tiles e.g. SLICE_X5Y9 → 5
get_rows	tiles	Returns the rows of the given tiles e.g. SLICE_X5Y9 → 9
sort_properties	file	Sorts the special properties inside a file in the current constraint set BEL > LOC > LOCK_PINS > FIXED_ROUTE
synthesize	(o) run (o) force	Synthesize given run, returns 0 on success and 1 on error. Force will synthesize even if not necessary.
implement	(o) run (o) force	Implement given run, returns 0 on success and 1 on error. Force will imeplement even if not necessary.
get_bels_lut6	of_objects	Retrieves all LUT6 resources of the given objects.
place_cells	cells bells	Places the given cells on the given BELs by the order of the lists.
get_boundary_nets	of_cells	Retrieves boundary nets of the given cells.
set_isolated	cells	Sets the given cells to isolated
set_reconfigurable	cells	Sets the given cells to reconfigurable
match_lists	list1 (r) list2 (o) strip_filter	Sorts list2 to match list1. The strip_filter will cut off the root of each element for comparison. Useful for matching cells and nets from differently named parents.

Parameters: (o) optional, (r) reference

Procedure	Parameters	Description
reset_properties	properties objects	Will reset multiple properties of the given objects. Ignores if they are not present and has special handling for properties like BEL and LOC to really remove them.
relocate_cleanup	cells	Cleans the given cells for relocation. Attention: Also cleans properties of boundary nets!
relocate_cells	base_cell cells	relocates the interface of base_cell to all cells. Detailed explanation in chapter 4.
copy_object_properties	properties from to (o) parse_prop	Copies given properties from one object to another and optionally parses the property before. Used for relocation. parse_prop
relocate_partpin_property	(r) from (r) to (r) property (r) value	Can be used for copy_object_properties. Parses PARTPIN_LOCS property for relocation and transforms the partpin location of the partpin relative to the pblocks of the from and to objects. Currently not used, fix_plocs is doing the job.
relocate_loc_property	(r) from (r) to (r) property (r) value	Can be used for copy_object_properties. Parses LOC property for relocation and transforms the LOC property relative to the pblocks of the from and to objects.

Parameters: (o) optional, (r) reference

Bibliography

References

[Frame-UG702] UG702: Partial Reconfiguration User Guide
Page 96: "Base regions in 7 series FPGAs are 50 CLBs high by 1 CLB wide."
https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug702.pdf