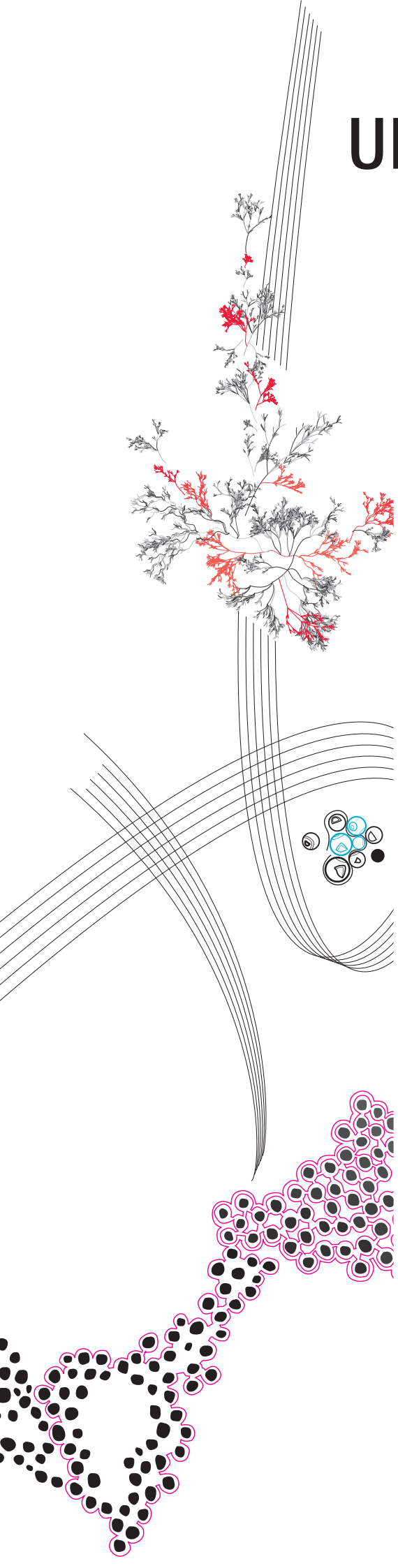


UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Design Reconstruction for Partial Reconfigurable FPGA Systems

Jeroen ter Haar

M.Sc. Thesis

September 2021

Supervisors:
dr.ing. D.M. Ziener
Ali Asghar
dr.ir. A.B.J. Kokkeler

Computer Architecture for Embedded Systems
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Contents

CHAPTER 1

INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are general-purpose chips with a large number of programmable cells that can be programmed to form any logic circuit. Its functionality can be altered after manufacturing, hence the name field-programmable. The reconfigurable nature of FPGAs is useful for prototyping applications and beneficial for systems that are susceptible for changes or future updates. The configuration of an FPGA is generally specified using a hardware description language. After that, the configuration is translated which represents an electronic circuit to be mapped onto the fabric of the FPGA.

Partial Reconfiguration (PR) is a feature that allows modification of certain predefined regions of the fabric of the FPGA. Here the fabric is divided into a static region and one or more dynamic regions. During runtime, the dynamic regions can be reconfigured while the remaining design continues to function without interruption. Hardware resources can be virtualized in a time-shared fashion. Increasing the logic density of the chip and larger designs may be implemented on smaller chips. *Dynamic Partial Reconfiguration* (DPR) extends the design flexibility even further by allowing hardware modules to be loaded during run-time on demand.

The use of DPR introduces several design and implementation challenges. Depending on the tools used, PR designs require additional design steps such as partitioning, floorplanning and constraints that have to be applied on the design. There exist various commercial and academic frameworks or tools to assist the designer in this process. While the major vendor tools do have support and implement the DPR design flow, they do not come without limitations. One such limitation is the lack of more advanced reconfiguration styles such as slot- and grid-style [Int20; Xil20], where the partial region can host multiple reconfigurable modules at the same time. Module relocation is another missing feature, allowing the same modules to be reused in more than one partial region.

Certain tools from the academic community such as [BKT12; Zam+18] overcome these limitations or extend the vendor tool flow. Adding new features or provide an automated and stand-alone framework that implements the whole system In general, the commercial tools have a dependent design flow for building PR systems. Where dependent means that the static system and reconfigurable modules are developed in a single project. The integrity and compatibility between static design and partial modules are kept in this way. Furthermore, it provides verification of the complete system by means of a timing analysis and (post-implementation) simulation. Independent design flow found in some of the academic tools allows the static system and reconfigurable modules to be designed independently from each other. This can save implementation time, but a drawback of this decoupling is that the design is difficult

to test as a whole. Although the reconfigurable modules could be tested and simulated on their correct behavior, for the static system this is not possible by default. Moreover, with different modules and configurations, interface mismatches and timing-related bugs are likely to occur. With only in-circuit testing, it can be hard to point out where the real error originates.

1.1 Problem Description

Verification of FPGA systems can be of a challenge, in particular, that of designs using PR. The academic tools that implement a grid-style reconfiguration architecture do not have the ability to perform a timing simulation. Reason for this limitation is that the design was split into static and dynamic parts. A full placed and routed design (static including partial modules) does not exist, thus a valid timing analysis can not be performed. For such tools, functional testing and verification can only be done in-circuit. This method of testing requires that the target hardware is known and present and this might not always be possible or practical. In-circuit test can only be done in a late stage of the design. If bugs arise in this late stage, such as timing violations, additional time must be spent to resolve these issues. Timing-related bugs can be hard to detect. This is mainly because they occur on-chip and are dependent on the given clock and data signals. In the case of PR, the timing can also be module configuration dependent. Testing the whole design before deployment into the field would be of added value. Even if a module is successfully placed on the fabric during reconfiguration, there is no guarantee that the system behaves and functions as expected.

Each time when a new module is developed, the system as a whole should be tested again. As the number of variants of a module increases, also the number of configurations and the number of tests that have to be performed increases. In-circuit testing done manually can be cumbersome, inaccurate, time-consuming and error-prone, especially when grid-style reconfiguration architecture is used. Therefore, automating the verification of systems that uses partial reconfiguration will be of added value for the designers.

In this work, an automated tool for verification and design reconstruction is proposed for PR systems. The verification part checks if a given set of modules is compatible with each other and the static system. A fully placed and routed netlist is obtained by merging reconfigurable modules back into the static design. Since place and route constraints must be kept, the merging is done on a netlist level. The so-called design reconstruction takes care of this merging process. The logic of the modules that belong to a specific configuration is placed into the partial region of the FPGA fabric and interface nets are reconnected. Finally, we end up with a fully placed and routed design on which a timing analysis and functional simulation can be performed.

1.2 Thesis Outline

This report is organized in chapters and sections and has the following structure. First, the necessary background information is provided in Chapter 2. The main topics are FPGA architecture, design and verification flow and DPR. The related work is described Chapter 3 summarizes the work found in the academic literature. In Chapter 4 a method is presented to overcome some of the mentioned limitations in the previous chapters. Next, the realization of the tool that implements the proposed functionalities is described in Chapter 5. Some examples are presented in Chapter 6 which also provides a way to check the correctness of the design reconstruction. Finally, the conclusion and some possible future research directions are discussed.

CHAPTER 2

BACKGROUND

This chapter provides the reader with the necessary background information required for this work. In Section 2.1, the basic architecture of FPGAs is described. The general design flow and design verification is described in Section 2.2. Section 2.3 describes the feature partial reconfiguration together with the use cases. After that, the previous work and the current literature on verification and timing analysis of partial reconfiguration systems is discussed. The bitstream format and the possible interactions with this binary file is described in Section 2.5. The last section Section 2.6 gives a short introduction on TCL scripts since it will be used extensively in Chapter 5.

2.1 FPGA Architecture

FPGAs are composed of a large number of logic elements and interconnects on a programmable fabric. This programmable fabric allows for combinations in logic elements to be made, providing the flexibility to implement almost any algorithm. The fabric is a structure of a grid-like array of tiles (see Figure 2.1.1).

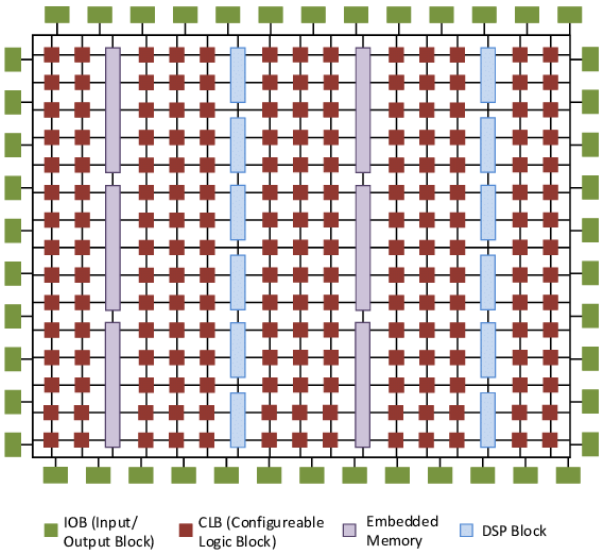


Figure 2.1.1: A simplified representation of the typical internal architecture of the FPGA fabric where some of the basic components can be identified.

Tiles are arranged by identical resource types. In general, they span the whole vertical direction and one or more columns in the horizontal direction. Apart from the

tiles, the fabric is furthermore divided into separate clock regions. This allows for a more even distribution of the clock signal on the fabric. A list of common tile types that can be found in most FPGA architectures include:

- *Interconnect* (INT) tiles provide connections between the logic blocks.
- *Configurable Logic Block* (CLB) tiles include digital logic elements that implements the user logic.
- *Input/Output Block* (IOB) tiles used for the communication outside the chip.
- *Clock Management Tile* (CMT) that to provide clock frequency synthesis.
- *Digital Signal Processor* (DSP) tiles that contain hardware multipliers and accumulators to enhance the speed and efficiency of applications that are using digital signal processing.
- *Block RAM* (BRAM) tiles to provide on-chip storage for data.

FPGAs are equipped with prefabricated routing resources. The INT tiles are the primary routing resource on the FPGA fabric. It consists of a switch box[1] and wires. The switch box allows wires to switch between vertical and horizontal wires (Figure 2.1.2). Switch boxes that connect tracks in the same direction are called planar switch boxes, while switch boxes that allow connections to other directions are called Wilton switch boxes [DÉ18]. They are commonly used since they provide routing flexibility. The INT tiles contains wires of different lengths. Single-length wires are intended for short connections to adjacent CLBs. Double-length wires that spans two CLBs. Long wires can reach several CLBs.

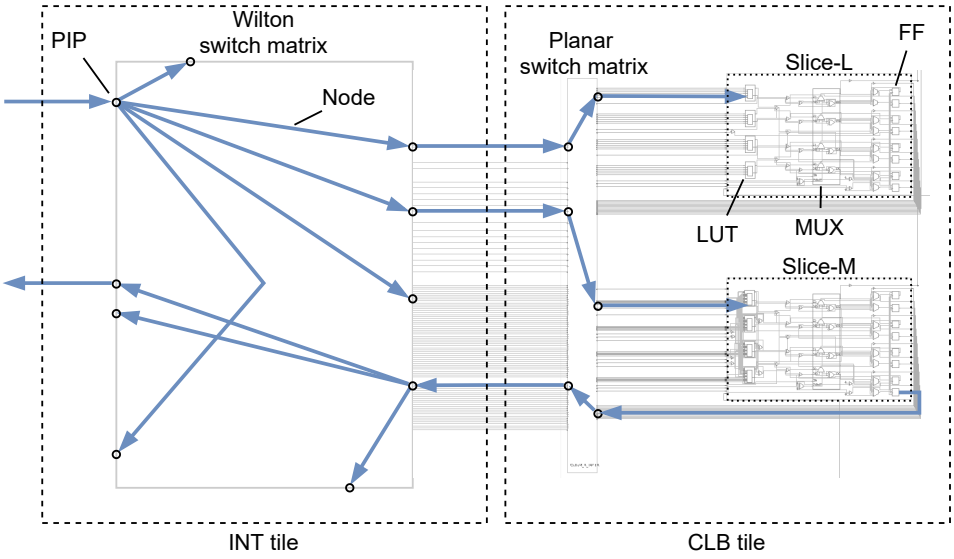


Figure 2.1.2: The topology of CLB and INT tiles in a Xilinx 7-Series FPGA.

CLBs form the primary resource for any combinatorial or sequential function. For example, the Xilinx 7-Series CLBs[2] contain a pair of identical slices, arranged symmetrically [Xil18a; Xil16]. Those slices contain the *Basic Element*s (BELs) such as *Look-up tables* (LUTs), *Multiplexer*s (MUXs) and *Flip-Flops* (FFs). Each slice has four 6-input LUTs, 8 FFs and a 4-bit carry chain. The carry chain logic is intended for

[1] Commonly called a switch matrix.

[2] The focus in this work will be on the Xilinx FPGA architecture, but is comparable to other FPGA manufactures and architectures.

the implementation of fast arithmetic functions. Some slices (such as SLICEM) have additional memory capabilities and can be configured as synchronous RAM cells.

The basic building block of the FPGA is the LUT which are available in the majority of FPGA architectures. LUTs are the primary building block to implement any Boolean logic function. Basically, a LUT is a multiplexer where k inputs are compared with 2^k SRAM cells (Figure 2.1.3). A truth table is stored into the SRAM cells and can represent any Boolean function. LUTs with k inputs can implement 2^{2^k} different functions. For example, the 7-Series FPGA implement a 6-input LUT. With 6 inputs we can form $2^{64} = 4096$ logic functions. The inputs of the LUT are permutable, the same function can be achieved by swapping the inputs of the LUT. This swapping property gives the router more freedom to find a shorter path.

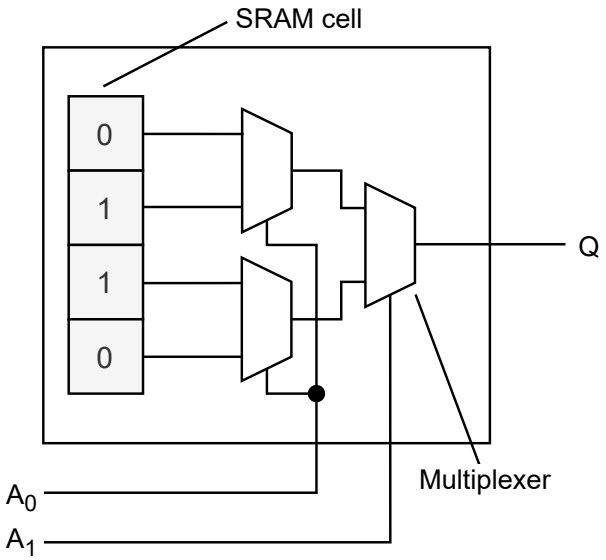


Figure 2.1.3: Example of a 2-input LUT. Here the LUT will behave as a XOR-gate with the provided SRAM configuration.

2.1.1 Modeling Routing Resources

The routing resources on the fabric of an FPGA can be modelled as a directed *Routing Resource Graph* (RRG). Consider RRG G, where $G = (V, E)$. Each vertex $v_i \in V$ corresponds to an electrical wire segment (or pin). Each edge $e_{i,j} \in E$ represents the (programmable) connection between two vertices[3]. Figure 2.1.4 shows an example of a RRG. We can furthermore define a *net* $N_i = (s_i, n_{i,1}, ..., (n_{i,2}, t_{i,3}), ..., t_{i,k})$ as a signal route in G [MB14]. Each net N_i starts with a source pin $s_i \in V$ and ends in one or more sink pins $t_{i,j} \in V$. Intermediate nodes are defined by $n_{i,j} \in V$. In essence, the set of nodes in N_i forms a routing tree, which that defines all paths from the source to all sinks.

From the Xilinx FPGA design perspective, a net consists of interconnected pins, ports and wires [Xil18b]. Nets can be grouped to form buses. The signals declared in the HDL design are converted to a netlist during the Place & Route phase of the design. In the post-synthesis design, a net connects a starting point to an end point.

[3]Although for this work we do not (re-)route the FPGA, additional knowledge on routing and routing resources of FPGAs was required and gained by reading the available literature.

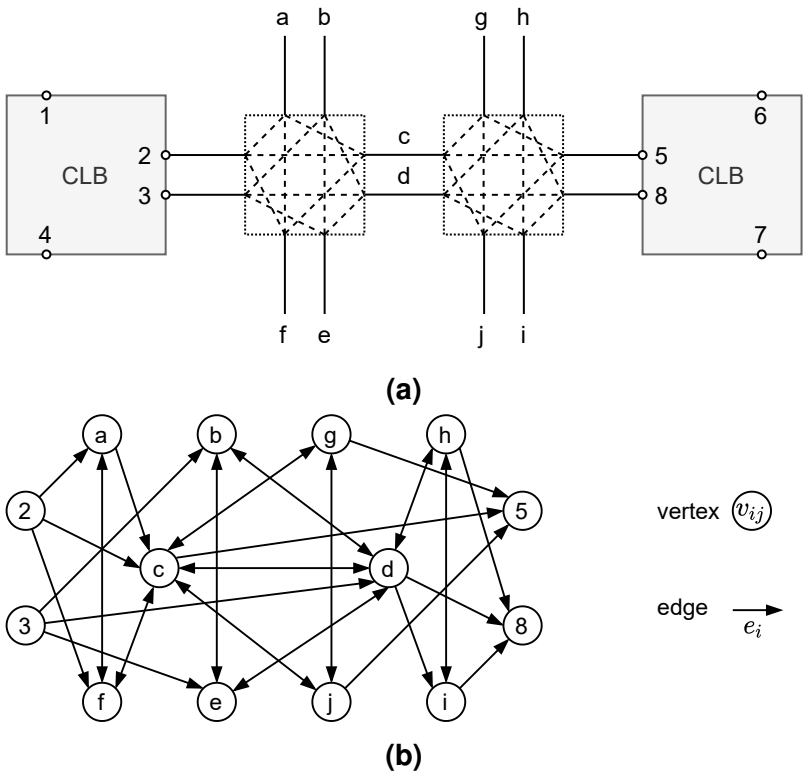


Figure 2.1.4: Route resource graph of FPGA. From the fabric level (**a**) to a graph model (**b**).

Those endpoints are the input and output pins of logical components (such as LUTs, flipflops, DSPs etc.). Moreover, we can distinguish two kinds of nets, logical and physical nets (Figure 2.1.5[4]). A logical net forms a network of connected cell pins in the RTL schematic. The physical net describes the physical connections between site pins on the chip. During a place operation in the design step, the design is mapped onto the routing resources of the target FPGA chip. This mapping effectively creates the physical net.

In the device view of an open design from the Vivado Design Suite, we can query the properties for each net. This is done by the `get_nets <net_name>` and `get_property <net>` commands. Physical nets have the additional property called *ROUTE* which specifies the physical structure of the route. The route is stored as a directed routing string, represented by a tree structure. Branches in the route string are represented by curly braces ({}). An example of a directed route string is shown in Figure 2.1.6.

The route shown in Figure 2.1.6 can be represented as $\{n_1\ n_2\ n_3\ \{\ n_7\ n_8\ \}\ n_4\ n_5\ n_6\}$. Where n_1 to n_8 represent the wires of the route. Another valid representation could be $\{n_1\ n_2\ n_3\ \{\ n_4\ n_5\ n_6\ \}\ n_7\ n_8\}$. By default, the route strings in Vivado are formatted using relative wires. Relative route strings are smaller in size since the tile information is omitted. However, without the tile information, the route string is ambiguous. Wires with the same name may be repeated several times. The Xilinx Vivado tool accepts an absolute route string where each node is formatted tile/wire combination like `tilename_x<?>_y<?>/wire`. Where each tile is distinguished by a combination of the tile type and the x- and y-coordinates and `<?>` is an integer (an example would be

[4]From the tutorial "Build a Basic Router" [LK18]

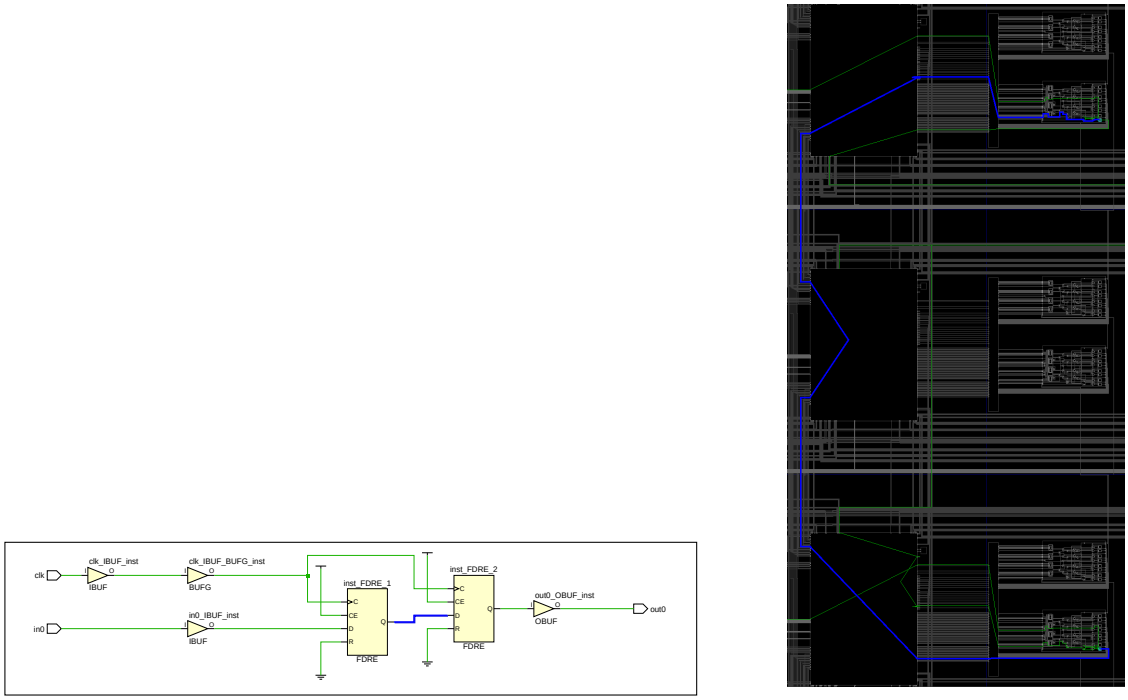


Figure 2.1.5: Example that shows the difference between a logical and physical net. This is the output result after synthesis and implementation phase of Listing A.0.4. Where (**a**) is the resulting RTL schematic and (**b**) the corresponding implementation in the device view of Vivado. The blue line is the intermediate signal `i_out` between the two flipflops in the VHDL source code.

```
1  get_property ROUTE [get_nets i_out]
2  # { CLBLM_M_AQ CLBLM_LOGIC_OUTS4 NR1BEG0 NR1BEG0 BYP_ALT1 BYP1 CLBLM_M_AX }
3
4  get_nodes -of_objects [get_nets i_out]
5  # INT_R_X71Y74/BYP_ALT1 INT_R_X71Y74/BYP1 CLBLM_R_X71Y74/CLBLM_M_AX INT_R_X71Y73/NR1BEG0
   ↪  INT_R_X71Y72/NR1BEG0 CLBLM_R_X71Y72/CLBLM_LOGIC_OUTS4 CLBLM_R_X71Y72/CLBLM_M_AQ
6
7  get_pips -of_objects  [get_nets i_out] -downhill
8  # CLBLM_R_X71Y72/CLBLM_R.CLBLM_M_AQ->CLBLM_LOGIC_OUTS4 INT_R_X71Y72/INT_R.LOGIC_OUTS4->>NR1BEG0
   ↪  INT_R_X71Y73/INT_R.NR1END0->>NR1BEG0 INT_R_X71Y74/INT_R.NR1END0->>BYP_ALT1
   ↪  INT_R_X71Y74/INT_R.BYP_ALT1->>BYP1 CLBLM_R_X71Y74/CLBLM_R.CLBLM_BYP1->CLBLM_M_AX
9
10 get_absolute_routestring_from_nets_dict $nets
11 # i_out {\{ CLBLM_R_X71Y72/CLBLM_M_AQ CLBLM_R_X71Y72/CLBLM_LOGIC_OUTS4 INT_R_X71Y72/NR1BEG0
   ↪  INT_R_X71Y73/NR1BEG0 INT_R_X71Y74/BYP_ALT1 INT_R_X71Y74/BYP1 CLBLM_R_X71Y74/CLBLM_M_AX \}}
```

Listing 2.1.1: Example TCL script shows how to get the route information from a net. These commands are entered into the TCL console of Vivado on an open design. In this example, net `i_out` is the intermediate signal between the two FDRE instances.

node INT_R_X41Y35/SS2BEG2). An INT tile is is associated to each CLB (see Figure 2.1.2). The INT tile consist of a Wilton *Switch Matrix* (SM) where each input has multiple mappings possible to the output nodes. The Input node can send its signal to various outgoing nodes (called downhill nodes). The connection between each input and an output node of the SM is controlled by a *Programmable Interconnect Point* (PIP). These PIPs are programmable (or configurable) interconnects and is achieved

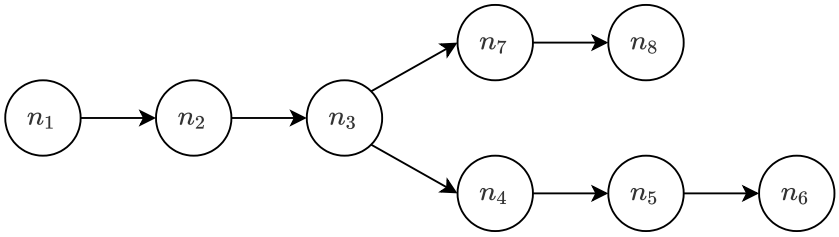


Figure 2.1.6: Example route string

by turning on or off a CMOS transistor. When turned on, the input signal passes from the input node to the corresponding output node. Between the CLB and SM there exist another switchbox. This planar SM is not user configurable.

Wires are the metal interconnects on the fabric in a single tile. A node is a collection of wires that can span multiple tiles. Nodes and wires are defined and named by their cardinal direction on the fabric. They are formatted by concatenating property fields into a single string as shown in Equation (2.1.1):

$$wire = \langle cardinal \rangle \langle displacement \rangle \langle direction \rangle \langle index \rangle \tag{2.1.1}$$

Where we define:

$cardinal = \in \{NN, NL, NR, NE, NW, EE, \ldots, WW, \ldots, SS, \ldots\}$.
The cardinal direction of the wire (or node). This includes north, east, south, west, and intercardinal directions. This is denoted by two characters, where, for example, NN means north direction.

$displacement = \in \mathbb{N}$. This is the length of the wire, roughly the number of tiles it skips.

$direction = \{BEG, END\}$. Begin or end, refers to the begin and end port of the switchbox.

$index = \in \mathbb{N}_0$. This is the index for identical nodes in the same direction. Wires that have an identical direction and properties are grouped and indexed by this number.

There exist also nodes that have different formatting, for example:

- Wires starting or ending in a CLB (e.g. CLBLM_LOGIC_OUTS1, CLBLM_M_AX).
- Planer SM wires (e.g IMUX_L1).
- Bypass wires in the switchbox (e.g. BYP_ALT1, BYP1).
- Long vertical nodes spanning multiple tiles (e.g LV_L0).

2.2 Design and verification flow

The design and verification flow for FPGA designs is shown in Figure 2.2.1. In general, it starts by having a design specification (or idea) written in a HDL language (e.g. VHDL or Verilog). The HDL sources are then modelled into an abstract digital circuit which is called the RTL description of the design. Next step is synthesis, where the HDL code is translated to the available design primitives. Design primitives are the actual gates, registers, LUTs etc. that are present on as available hardware resources of the target device. The implementation phase consists of two steps. In the place step, the location of the hardware is decided, effectively mapping the design onto the chip. Followed by the route step, which decides which logic should be connected using the programmable routing fabric. After this, a so-called bitstream file is generated, a binary file containing all the instructions to configure the FPGA.

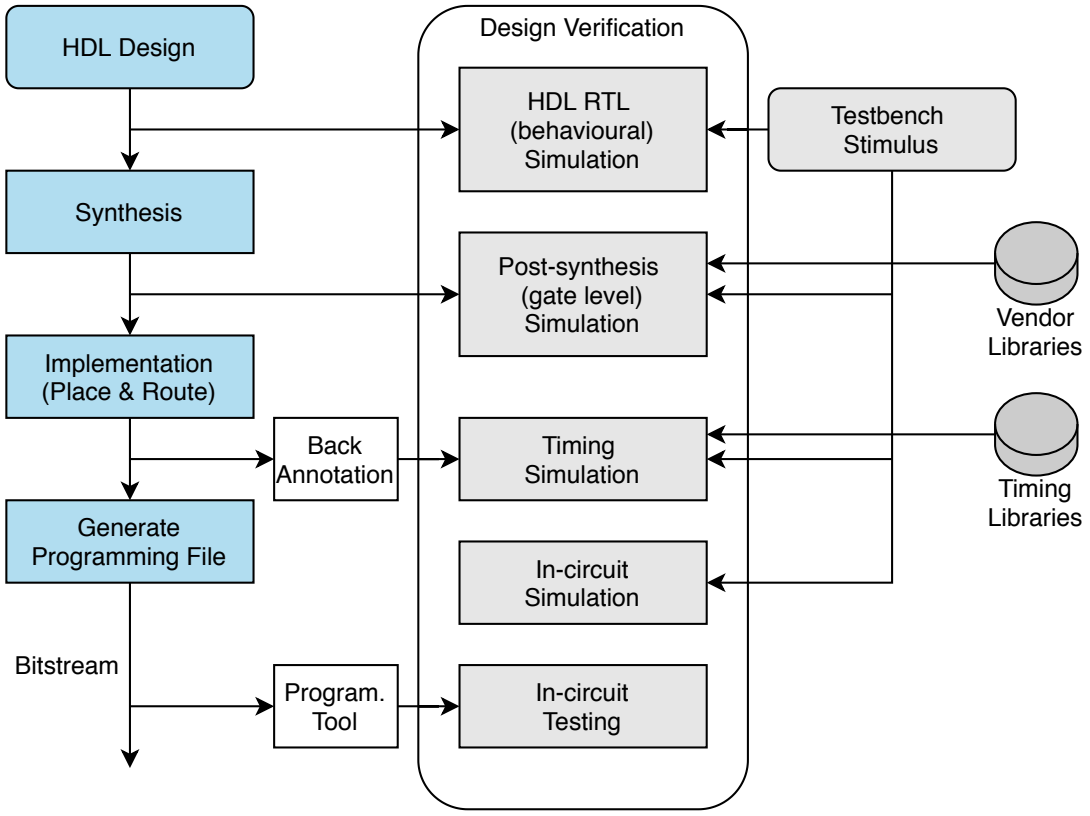


Figure 2.2.1: The general design flow (in blue) and verification flow (grey) for FPGA systems (figure from [DSC12]).

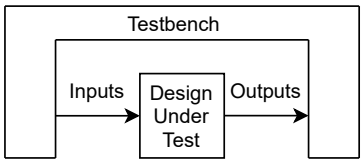


Figure 2.2.2: Test bench for DUT

Different methods of verification are possible during each step of the design phase. Verification is required to ensure that the design behaves correctly and as intended

by the designer. A test bench is often used when working with HDL languages such as VHDL and Verilog. With a test bench you apply input signals to the design as if it is connected to the real world (Figure 2.2.2). The output is captured by the test bench and compared with the reference output. Additionally, most simulators provide a graphical waveform viewer to capture and observe the output signals in time. Note that when using any of the HLS languages (e.g. C++), this is often carried out by a co-simulation where the hardware test bench is often automatically generated by the HLS tool. We can distinguish five types of (in-software) verification methods by means of simulation[Xil19a]:

- *Behavioral Simulation* is performed on the RTL and verifies only the logic without any delay information.
- *Post-Synthesis Functional Simulation* is performed after synthesis and ensures that any optimizations have not affected the functionality of the design.
- *Post-Synthesis Timing Simulation* is performed on an unrouted design and includes only estimated time delays about the routing and components of the FPGA being used.
- *Post-Implementation Functional Simulation* is performed after the design has been placed and routed. This verification is useful for determining if any physical optimizations during implementation have affected the functionality of the design.
- *Post-Implementation Timing Simulation* is used for detecting whether or not the design can operate at the specified clock speed using accurate time delays. This is the closest possible way to emulate the design on the device. Making it possible to detect asynchronous path timing errors. The netlist is annotated with timing information using a SDF file, in which all circuit delays are defined.

Note that the timing information for items 1, 2, and 4 in the list above are ignored. Additional vendor libraries for device specific (timing) information are required to do any of the post-synthesis and timing analysis.

The final verification of the design is the in-circuit testing. This is done on the actual hardware, the circuit board itself. For example by interaction with the board itself (e.g. buttons, LEDs, measuring voltages, etc.) or via a serial data interface for debugging.

2.2.1 Timing Analysis

Timing analysis is one of the techniques to verify the timing requirements of a digital design. These requirements are, for example, the clock speed on which the design must be able to operate. Apart from any geometric requirements, the design must also meet the timing constraints, e.g. the setup and hold constraints. The optimization process that meets these requirements is called *timing closure*. Violations in timing constraints lead to glitches at the output which results in undefined behavior of the design.

Delays in electronic circuits are mainly due to the on and off switching time of transistors. Charging and discharging of (parasitic) capacitors present in each transistor takes time, increasing the turn on and off time of the transistor (see Figure 2.2.3). The time delay by signal propagation in wires plays a less significant role in this. However, the routing architecture of FPGAs consists of wires and switches that are used to connect those wires. The type and quantity of switches attached to each routing wire, such as pass transistors, multiplexers, buffers, increase the overall wire delay. As well

as the size of transistors, the topology of the interconnection of the switches and the wire width and spacing [SR01].

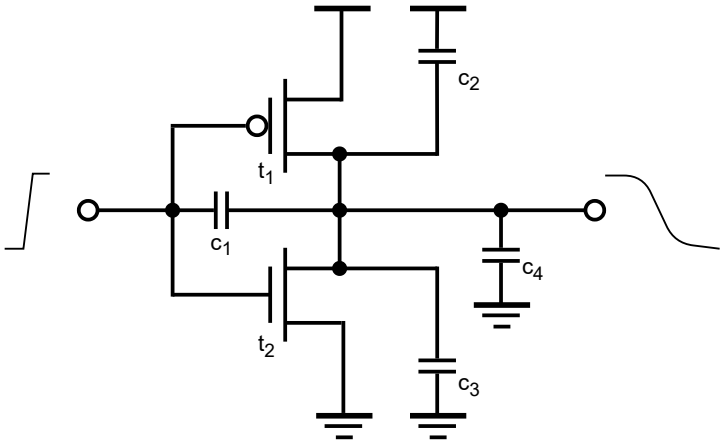


Figure 2.2.3: Parasitic capacitance present in a CMOS inverter circuit. Capacitors together with resistors form RC circuits (resistors not drawn) which take time to charge and discharge, increasing the turn on and turn off time of the transistors (MOSFETS).

Digital circuits are analysed on certain delay properties. Combinational logic is characterized by propagations delay and contamination delay. The propagation delay is the length of time from when the input changes until the output has reached its final value (Figure 2.2.4). Contamination delay is the minimum time when the output can change its value when the input changes. For synchronous logic, i.e. logic that requires a clock signal, we can define the setup and hold timing properties. These timing properties are required to check for proper propagation of data through sequential logic (or cells) by validating if the data is stable around the active edge of the clock. Setup time is defined as the minimum time period before the active edge of the clock where the input data must remain stable. Similarly, the hold time is the minimum time the input data must remain stable after the clock edge. The active edge of the clock for sequential logic is the rising or falling edge of the clock where data capture takes place. We can furthermore define t_{ccq} as the amount of time required for an initial change in output Q and t_{pcq} as the clock to output Q propagation delay of a flipflop.

We can define two methods for verification namely: *Static Timing Analysis* (STA) and simulation based analysis [BC09]. STA is performed statically, meaning that it does not depend on data input values. Whereas for simulation bases timing analysis, stimulus is applied on the inputs of the design under test. The output behavior is observed and verified, the time is then advanced and new input data is applied. The behavior is again observed and verified. Simulation-based timing analysis is only complete and exhaustive when all possible test vectors are used as stimulus. For large designs with millions of gates this is a very slow method, making it difficult to verify through simulation. On the other hand, static timing analysis provides a faster and simpler method for checking if paths have any timing violation. STA can be used to optimize the design by finding the worst or critical time paths. Timing-driven placement uses STA to identify critical nets to improve signal propagation. This is achieved by either minimizing the *Worst Negative Slack* (WNS) or the *Total Negative Slack* (TNS).

All static timing analysis is done on paths. A path is a net that starts at a clocked element (e.g. a flipflop), going through any number of combinatorial elements and

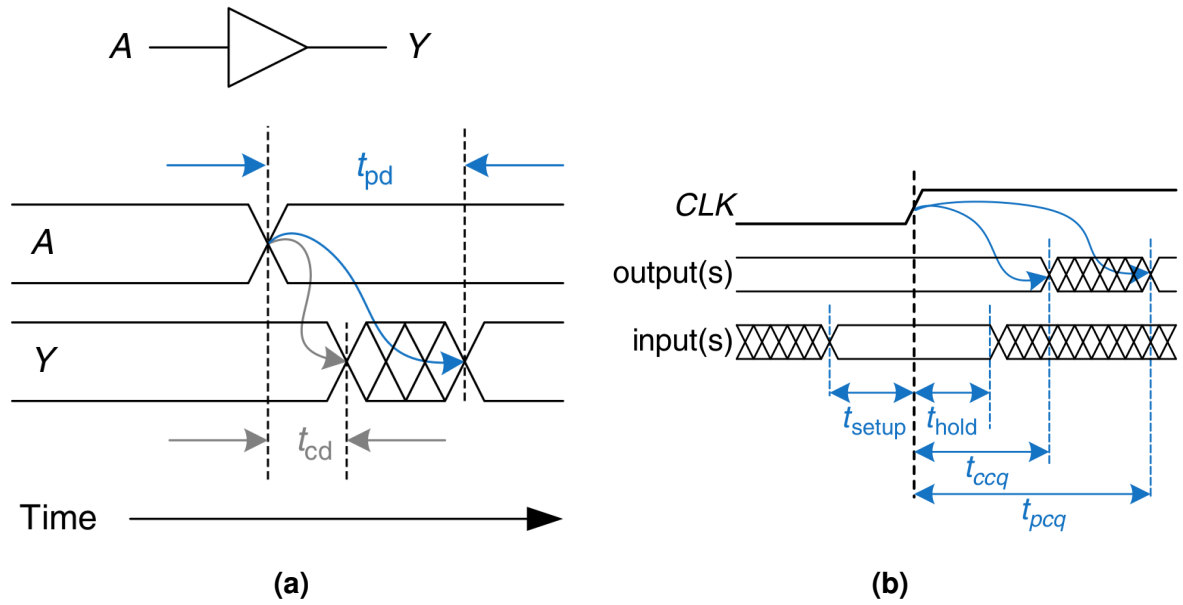


Figure 2.2.4: Timing properties displayed in the wave form where we have: For logic elements (**a**), propagation delay t_{pd} and contamination delay t_{cd}. For sequential logic (**b**), the setup time t_{setup} and hold time t_{hold}. Figures from [HH07].

ends at a clocked element. For example, in Figure 2.1.5a signal `i_out` (blue line) is path. Paths themselves may have multiple segments as they can pass through different levels of hierarchy in the design. The critical path is the signal path that has the longest propagation delay. This path determines the highest clock speed possible for the design. All static timing analysis is conducted on paths to determine the overall circuit delays.

A key metric for STA is the timing slack for a given timing point. This timing slack is defined as the difference between the requested arrival time and the actual arriving time. The *slack* value is an indicator of whether the timing constraint for node v has been satisfied. A positive value means that the timing is met, i.e. there is some *slack*. Negative slack indicates a timing violation, there exist a signal that arrives after its required time. The *timing slack* of a node v is defined as follows [Kah+11]:

$$slack(v) = RAT(v) - AAT(v) \tag{2.2.1}$$

Where:

$RAT =$ Required Arrival Time

$AAT =$ Actual Arrival Time

The WNS is defined as:

$$WNS = \min_{\tau \epsilon T}(slack(\tau)) \tag{2.2.2}$$

Where T is the set of all timing endpoints. The TNS is defined as:

$$TNS = \sum_{\tau \epsilon T, slack(\tau) < 0} slack(\tau) \tag{2.2.3}$$

2.3 Dynamic Partial Reconfiguration

FPGAs have two modes of operation, configuration mode and user mode. After power-up, an (SRAM-based) FPGA goes into its configuration mode. In this mode, there exist several mechanisms to configure an FPGA. The Common and most widely used method is the JTAG interface. This interface can also be used for testing the device and handle multiple devices. Xilinx FPGAs offer various configuration methods such as SelectMap, *Internal Configuration Access Port* (ICAP), *Processor Configuration Access Port* (PCAP) or via a serial interface. To make the configuration persistent after a power-cycle, the program is stored onto a flash memory chip near the FPGA. During the start of the system, the configuration is loaded into the *Static Random-Access Memory* (SRAM) memory chip to initialize the FPGA. Three configuration methods can be classified as follows:

- Full configuration: The configuration is loaded during start-up (or during development) of the FPGA.
- Dynamic reconfiguration: During operation, the FPGA is put into a configuration mode to update its entire configuration.
- Dynamic Partial Reconfiguration (DPR): The FPGA keeps on performing its task, but a portion of the fabric is reconfigured.

Additionally, there exist FPGAs that implement the feature of DPR. DPR allows you to reconfigure a portion of the FPGA, while the remaining design continues to function without interruption. The fabric is divided into a static region and one or more dynamic regions (or partial regions). At any point in time, during run-time of the design, pre-compiled partial bitstreams can be loaded to alter the behavior of the system. Reconfiguration is done by a PR controller. The PR controller can be present on the programmable logic itself or externally via PCAP interface. In this work, a Xilinx Zynq 7000 chip is used. The Zynq SoC integrates the hardware programmability of an FPGA with a dual-core ARM processor. Here the processor system can issue a reconfiguration operation.

2.3.1 Terminology

The commonly used PR terminology that is used throughout this work is described next. DPRS are FPGA systems that use PR that are decoupled into a static and one or more dynamic parts. This decoupling is called the *partitioning phase* where the design (the project) is split into two parts, static and dynamic. Here *static* means it does not change during runtime of the design and *dynamic* refers the part where the behavior that can be altered during runtime. The designer determines which part of the FPGA design must be made dynamic and defines the architecture for the communication interface. Furthermore, in order to determine the number of resources required to host each reconfigurable module, *resource budgeting* is carried out (i.e. the number of LUTs, DSPs or BRAMs etc.). At last, the *floorplanning* step determines the location of each reconfigurable region on the FPGA fabric. To enforce routing constraints in the design a *blocker* or *blocker macro* is applied. A blocker is used to occupy all routing resources in order to force the (vendor) router not to use the resources. During the

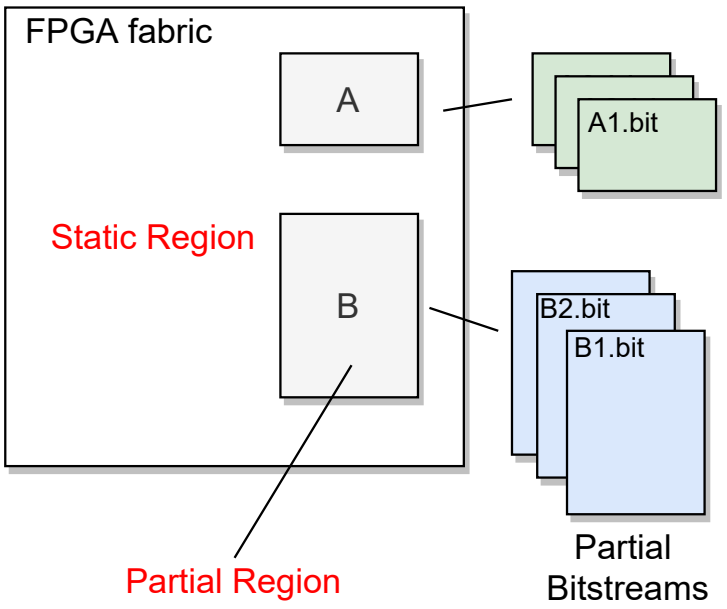


Figure 2.3.1: Concept of an FPGA system using partial reconfiguration. Multiple partial regions can be defined on which multiple variants of partial bitstreams can be loaded.

implementation of a module, the blocker is located around the partial region. Forcing the router only to use the routing resources inside the partition. Holes in the border are left open for interfacing. These holes are called *tunnels* and are wires in the border of the blocker excluded from blocking. Tunnels allow for communication in and out of the partial region (see Figure 2.3.2).

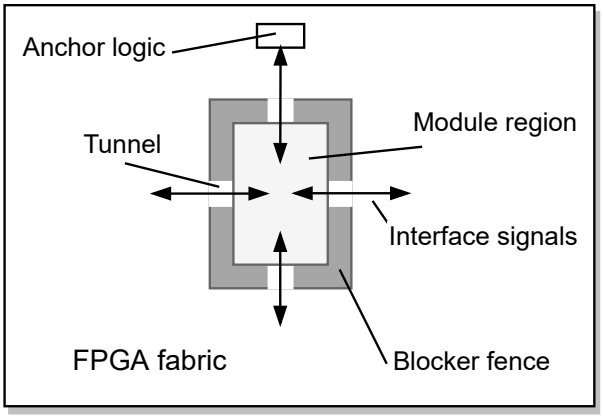


Figure 2.3.2: The blocker function is applied here to isolate a partial module. Tunnels are unblocked wires that module can use for interfacing. Anchor logic is used to tie-off the interface signals.

PR requires specials demands on communication architecture. Additional bus-based architectures or methods using (LUTs) proxy logic have been proposed and used in the literature. Using LUTs as anchor logic is a common method nowadays since it has the least logic overhead. The LUT input (or output) is used as a termination point for the interface signals.

The implementation of a reconfigurable module depends on the location on the

fabric. Behaviorally identical modules can have different functional implementations. Modules with function-equivalent implementations can, depending on where the module is placed and routed on the fabric, also differ in *module footprint*. We have seen in Section 2.1 that the fabric is divided into columns. An implemented module must follow those resource constraints. Columns with identical resource types give room for the feature called *module relocation*, where an implemented module can be reused on different partitions of the fabric. Columns with different resource types result into a different module footprint, which are most often not interchangeable with other partitions.

The partial region must be large enough to host the largest module. This might lead to low utilization of smaller modules and an increase of the internal fragmentation. Unused fabric space or area of a module is called *internal fragmentation*. Therefore, there are reconfiguration styles that provide smaller (more optimal) slots, to lower this internal fragmentation. As smaller slots will result into lower internal fragmentation. The reconfigurable area can be categorized in different reconfiguration styles (Figure 2.3.3). The first variant is single island style, where only one module is loaded in one specific partial area. Multi-island variants, such as slot-based and grid-style, allows for one or more modules to be loaded at the same time. For grid-style, the modules can have any arbitrary shape or size. Furthermore, adjacent modules directly communicate with each other. No additional overhead logic or routing resources is required for a direct module-to-module communication. Additionally, a fourth reconfiguration style can be defined: fine-grained reconfiguration style [Zam+18]. This granularity allows individual reconfiguration of components such as changing the truth table of a LUTs. Fine-grained reconfiguration on LUTs is comparable with the concept of *Tunable Look-Up Table* (TLUT) functions [BA09]. Having the advantage to be faster than performing full module swap. However, its usability is limited to only small and some specific cases. Some examples for this reconfiguration style could be: in circuit switching without any additional logic overhead, clock tree switching to changing the clock frequency, conditional logic switching (e.g. exchange an OR-gate for an AND-gate).

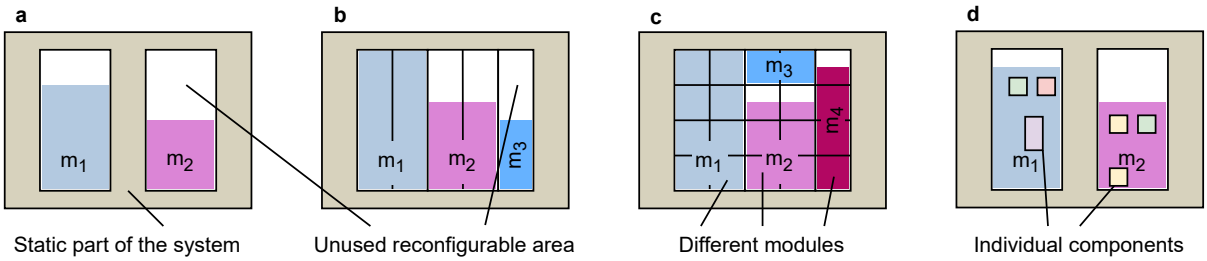


Figure 2.3.3: Different reconfiguration styles: island-style (**a**), slot-style (**b**), mesh or grid-style (**c**), fine-grained-style (**d**). Figures **a**,**b**,**c** from [Koc13].

Note that although we have different reconfiguration styles, the chip used might be limited in its PR granularity, see Table 2.3.1.

2.3.2 Benefits and Applications for DPR

DPR can be used for a wide variety of applications. To quantify some of the benefits, its prime use is to swap functions on-demand while the system is operational. More adaptive designs can be created in order to increase the functionality on demand.

Table 2.3.1: Xilinx PR Granularity

Architecture	PR Granularity	Circuit Relocation	PR Primitive
Xilinx Zynq	One clock region high	Very difficult	ICAP/PCAP
Xilinx Ultrascale	One CLB	Very difficult	ICAP/MCAP

With time-sharing FPGA (hardware) resources, more functionally can be implemented on smaller devices. Using fewer resources and thus more efficient in terms of silicon usage, can result in less power consumption. Additionally, loading functions only when needed can also lead to power reduction. Furthermore, the design might be able to work with a smaller sized FPGAs, reducing the cost of the system even further. Another benefit would be the reduced configuration time. Instead of writing a full bitstream, a smaller partial bitstream can be loaded.

A wide range of applications using DPR can be found the literature. These can be grouped based on the specific features of DPR being used such as adaptability, overhead reduction, reliability improvement, and hardware computing. To mention some of the fields of applications and use cases, for example:

- *Video processing* - in [KL10], an adaptable video de-blocking filter using DPR is proposed. This de-blocking removes artifacts that are created by block-based transforms, motion estimation and quantization operations. The ability to adapt to applications' needs is used to support different resolutions and frame rates dynamically.
- *Image processing* - in [Zam+18] different reconfigurable image processing filters (such as dilate, erode and Sobel filters) can be loaded on-demand during runtime.
- *Database accelerating* - using configurable hardware to accelerate database operations. In [DZT13; Ves19] present such implementations for using DPR to accelerate SQL database queries in hardware. The data from the database is transferred to the FPGA. Basic SQL operators are then executed in hardware resulting in an impressive speedup of the database query.
- *Side-Channel Protection* - counter measure against side-channel attacks on cryptographic implementations using DPR. For example in [Sas+15; Het+19], DPR is used to create different power profiles to make side-channel attacks on power lines more difficult.
- *Software Defined Radio (SDR)* - In [Hos+18], five wireless communication systems are implemented on a Zynq FPGA. It shows to be effective in saving area and power.
- *Real-time systems* - DPR can be used to schedule hardware with the concept of scheduling tasks in a way that real-time systems can benefit from DPR [Pez+17].
- *Neural networks* - in [You+20] the power consumption of the neural network is reduced by reducing the number of bits that represent the parameters of the neural network. In [IAZ21],DPR is used to optimize throughput and accuracy.

2.3.3 Verification of Partial Reconfigurable Systems

A common method for verifying hardware design functionality is using simulation. In general, the more details included in the simulation, the more accurate the simulation will be. However, this more detailed model leads to a decrease in simulation speed and is often more time-consuming for the designers to trace the root cause of simulation failures. Therefore we can say that verification productivity decreases with an increasing simulation accuracy [GD14] (Figure 2.3.4).

Figure 2.3.4 illustrates the productivity and accuracy trade-off. Productivity is defined as the simulation throughput, the number of simulated cycles per elapsed second. On top of the graph, we have the high-level languages which are capable of modeling and simulating hardware designs. Even though the simulation is not cycle-accurate, it is accurate enough to verify the hardware architecture. In the middle, the RTL-level of simulation, which is the common method used for verification. At the bottom, we have the timing simulation. This is most of the time performed on the design netlist annotated with the timing information.

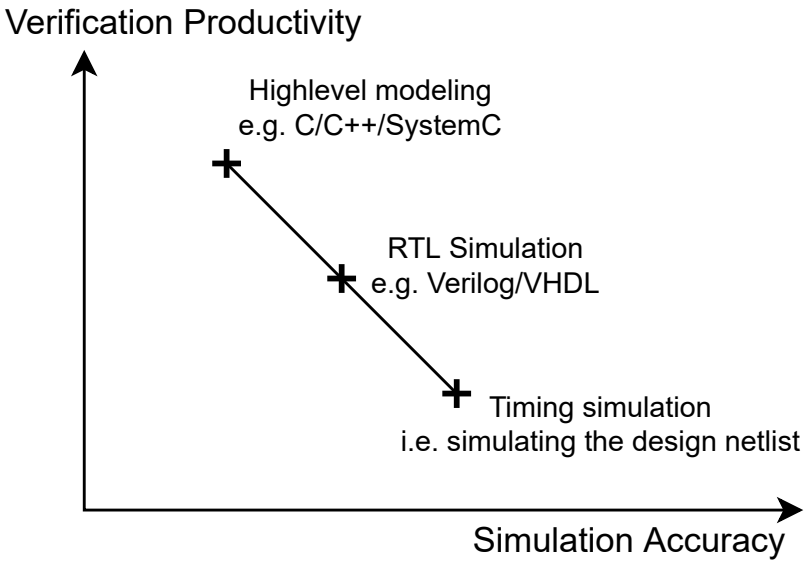


Figure 2.3.4: The simulation accuracy and verification productivity tradeoff (for static designs). Figure from [GD14].

There has been some limited work in the academic community on verification and simulating PR systems. Mainly because the major vendor tools do have basic support for simulation when you use their tool flow. Since PR is closely associated with the targeted FPGA architecture, fully modeling it requires modeling of low-level architectural details. Some papers present methods to verify if the correct interface connections are used. Or by automating design steps such as floorplanning, generate the (partial) bitstreams, mistakes can be found. One important challenge for functional verification is to verify the different stages of the reconfiguration process itself. Simulation of the actual reconfiguration process is not always fully supported by the major vendor tools.

The Intel Quartus Prime software can simulate PR designs [Int20] and also generate the gate-level PR simulation models for each module. It is possible to use the behavioral *Register Transfer Level* (RTL) or the gate-level PR simulation model for

simulation of the PR personas[5]. Simulation of PR persona replacement transition is done by using simulation multiplexers and a simulation wrapper (see Figure 2.3.5). The simulation multiplexers are used to change which persona drives the logic inside the PR region during the simulation. The resulting change and intermediate effect can then be observed in the reconfigurable partition.

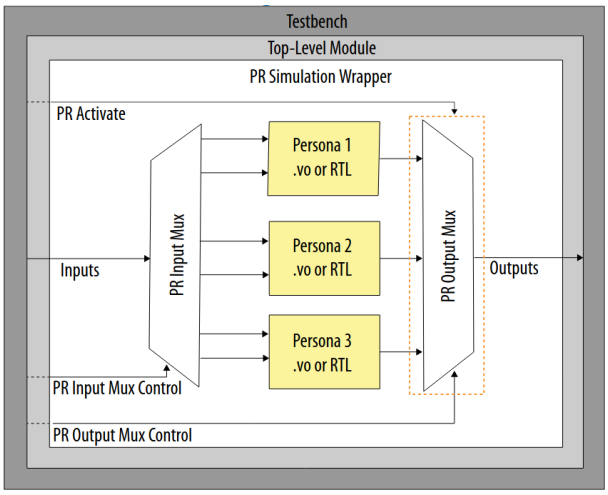


Figure 2.3.5: Simulation of PR persona switching (from [Int20]).

From Vivado, the configurations of PR designs can use the standard simulation, timing analysis, and verification techniques. However, the partial reconfiguration process itself can not be simulated [Xil20]. The stages of the reconfiguration process is described and categorized in [GD11a]. Divided into three stages, *BEFORE*, *DURING* and *AFTER*:

- *BEFORE* reconfiguration is the time between the request and the first configuration byte written.
- *DURING* reconfiguration is the time interval when the configuration is being written.
- *AFTER* reconfiguration is the last stage, this is the time after the last byte written and until the module is activated.

For each stage, various bugs and errors that can occur have been mentioned in [GD11a].

There exist some academic frameworks that are capable of modeling the partial reconfiguration process. In [GD14] the analysis challenges in verifying *Dynamically Reconfigurable Systems* (DRS) designs are stated. Furthermore, a simulation-only layer to emulate the behavior of the target FPGA is proposed. The simulation-only layer is an approach for the functional verification of DRS designs. There exist MUX-based methods such as [LSC97; Int20], but those methods fail to provide the accuracy required to verify the design undergoing reconfiguration. Mainly because they swap modules instantaneously or assume a compile-time defined reconfiguration delays. Simulating the reconfiguration process and the bitstream traffic improves the accuracy of functional verification.

[5]Intel calls the reconfigurable modules *personas*.

To verify the reconfiguration process, the ReSim library is presented by Gong and Diessel in [GD14]. It provides the designer assistance in verifying implementation-related bugs, such as timing violation errors in the placed and routed design, and short or open circuits, if any, caused by partial reconfiguration. This library uses a simulation layer to model the physical layer of the partial run-time reconfiguration systems. The configuration port and configuration memory are emulated in this work. A simulation bitstream (SimB) is used to transfer configuration data from storage to the configuration port. The design flow of using ReSim takes a functional specification and a set of reconfiguration strategies as input. These strategies include the name, size and connectivity of the partial region and *Reconfigurable Module* (RM). The reconfiguration strategies are described in a *Tool Command Language* (TCL) script. Based on that script, ReSim can generate the simulation-only artifacts. ReSim models the three stages of the reconfiguration process and is thereby capable of simulating a design undergoing partial reconfiguration. By accurately simulating the synchronization, isolation and initialization mechanisms of the BEFORE, DURING and AFTER reconfiguration, timing errors were detected in their case-study design. ReSim lets the designer make use of the ”x” value injection. The ”x” injection can be changed to any design- or test-specific error sequence. The chosen injection values will propagate through the system from which erroneous cycles can be detected.

In [HKT13] a cycle-accurate simulation framework is presented. It extends the idea of the ReSim [GD11b], but uses real bitstreams instead of simulation-only bitstreams. This framework operates on the RTL-level using *Very high-speed integrated circuits program HDL* (VHDL). The timing information is extracted from an actual FPGA and provides cycle-accurate simulation. The provided reconfiguration controller uses real bitstreams to control and simulate the reconfiguration process. This framework supports island- and slot-based reconfiguration styles as well as the more advanced features such as module relocation. The simulation framework is capable of detecting and covering most of the common bugs described in [GD11a]. This includes the bugs or errors that typically can occur during the different stages of the reconfiguration process.

There are other techniques to assist the (pre-)verification of PR designs. For example in [AMM18a] a technique is prosed to verify connections of the RMs using *Assertion Based Verification* (ABV). It can verify the RTL designs after being modified to match the DPR technique. The connections are modeled using *System Verilog Assertion* (SVA) properties. Where an assertion is a statement of the design that is expected to be true, SVA is a language construct providing a way to write the rules that constraint the design specification. The assertions can then be used for formal verification (or RTL simulation). When a property fails during verification, the root cause can be found without much extra effort. Assertions can be synthesized on the FPGA and used for runtime verification of DPR systems. Issues appear when there is a mismatch in the number of ports between different modes of the RM. In this paper, we use the connectivity verification approach to verify the changes in the interfaces of the RM. When the design is synthesized, the netlist of the design is traversed to extract the connections of the RMs from the original design. The port connections are verified for every mode of each RM. The proposed methodology verifies that the ports of all the RMs of the design are properly connected. Ahmed et al. [AMM20] uses previous work to demonstrate this on a *Software Defined Radio* (SDR) system. The effectiveness of applying these approaches in the design cycle is shown and three functional verification

approaches are presented for DPR to verify:

- the port connections of the RMs,
- the dedicated logic added for DPR activities,
- *Clock Domain Crossing* (CDC) signals in the designs.

Likewise in [AMM18b] uses the same assertion-based verification technique to detect bugs in the design They are able to identify output isolation errors, reset activation sequence errors, and issues waiting for running computations on a module before reconfiguring it.

In [AMM18c] a method is addressed for the issues that can occur during CDC. If a signal crosses a clock domain and it does not remain steady during setup and hold time, the receiving register can become metastable. Its output may settle at a random, undetermined value that is different from the RTL simulation. Meta-stability can cause functional errors in the design. The method proposed here first runs a sanity check on the number of ports used. After that, a configuration mode is picked for the DRS design that generates a RTL file for that mode. The utility generates the RTL design for every mode and a script to run Questa CDC tool from Mentor Graphics[6] to perform the analysis on the design. Steps are repeated for all possible configuration modes of the design. A report is generated from the CDC analysis. All is done on the RTL-level.

On the topic *Static verification and Design Reconstruction*. In the academic PR tools, static timing analysis of the whole system is not possible, at least not directly. However, Zamacola et al. [Zam+18] do offer a solution for this which they call *design reconstruction*. The reconstruction is capable of merging a module back into the implemented design of the static system. Essentially making the design as it is during run-time. Using their framework and the exported data during the implementation, a project that used island reconfiguration style can be reconstructed. Their framework is limited to island-style only, fine-grained reconfiguration style is not supported.

2.4 Design Checkpoints

A *Design Checkpoint* (DCP) is a file used by Vivado. It represents a snapshot of a design at any stage of the design process. At any point in time during the compilation process, the designer can save a snapshot of the design state to a file which is referred as a *design checkpoint*. The design checkpoint saves the intermediate state of the design flow. Four states in the design flow can be classified: linked design, post-synthesis, post-placement and post-routing. The linked design checkpoint does not have a netlist, while the other three do.

A checkpoint is an archive file containing a collection of files that hold the netlist and constraints of the design. The contents of the file can be viewed using any ordinary archiving software (e.g. 7-Zip). After extracting the archive, we end up with a collection of files:

- `dcp.xml` is an XML text file containing which version of Vivado is used, which part (device) and what the top entity is. Furthermore, it contains a list of files that are in the DCP archive.

[6] https://trias-mikro.de/wp-content/uploads/2018/07/Datenblatt-Questa-CDCand-Formal-Technologies.pdf

- `top.edf` specifies the design netlist. The file is formatted using the *Electronic Design Interchange Format* (EDIF) specification[7].
- `top.incr` contains timing-related information.
- `top.rda` contains a list of keywords and values separated with binary operators, further usage is not known.
- `top.shape` starts with the text: ”Xilinx New Shape Database” and contains some readable ASCII text. Contents and usage is unknown.
- `top.sta`, `top.wdf`, `top.xbdc` and `top.xn` for these files, the contents and usage is unknown.
- `top.xdef` is the Xilinx Design Exchange Format file.
- `top_late.xdc` is a design constraint file.
- `top_stub.v` contains the top entity of the design, i.e. the ports of the top entity. This is a Verilog source file.
- `top_stub.vhdl` contains the top entity of the design, equal to `top_stub.v` but then in VHDL.

Checkpoints are of interest because they allow custom-developed CAD tools (e.g. [LK18; WN14]) to interact with the design. For STA we required a full place and route design. A possible idea would to is merging multiple *Design Check Point* (DCP) files into one. From that point, a timing analysis can be performed. The incremental compile flow [Xil21, p. 122] of Vivado, the logic, placement and routing of multiple designs can be placed into a single design. However, it was found not useful (not intended by Xilinx) in the contents of merging multiple implemented designs into a single design.

The RapidWright [LK18] framework was reviewed and considered for this work. RapidWright is an open-source framework written in Java that complements the Vivado. Offering various additional features to customize and modify the FPGA design implementations. The topics of interest are *A Pre-Implemented Module Flow*[8] and the *Lightweight Timing Model* [Mai+19].

During this research, it was found that the lightweight timing model was not implemented for 7-series FPGAs. That is to say, the timing information has to be complemented for those devices. Furthermore, all the necessary functions required for this work are present in the Vivado Design Suite. Therefore, any additional tools are not necessary. Although with RapidWright offers better debug capabilities and more abstraction can be applied using object-oriented programing style, with Vivado we directly interact with the open design and inspect the state graphically.

2.5 Bitstream Format

Xilinx FPGAs are configured by a binary file called bitstream. It contains the information of the hardware logic, routing and initial values for on-chip memory. The file is a

[7] https://www.rulabinsky.com/cavd/text/chapd.html
[8] From https://www.rapidwright.io/docs/PreImplemented_Modules_Part_I.html

set of commands that are executed in sequence during the configuration of the FPGA. Those commands are instructions that not only hold the chip configuration, but also describe the configuration process itself. Split into three parts: a header, the configuration data and a footer (Figure 2.5.1a). The *SYNC* word is used to allow the configuration logic to align at a 32-bit word boundary. Furthermore, the header contains information about the origin, device, encryption and content of the entire bitstream. The body holds the configuration data, which is arranged in data frames (Figure 2.5.1b). Those frames are tiled over the device and are the smallest addressable segments of the FPGA configuration memory [Xil18a]. They configure the resources of the FPGA (the CLBs, IOs, BRAMs etc.). The footer finalizes the chip configuration and takes care of the start-up sequence of the device. The *DESYNC* command releases the configuration logic.

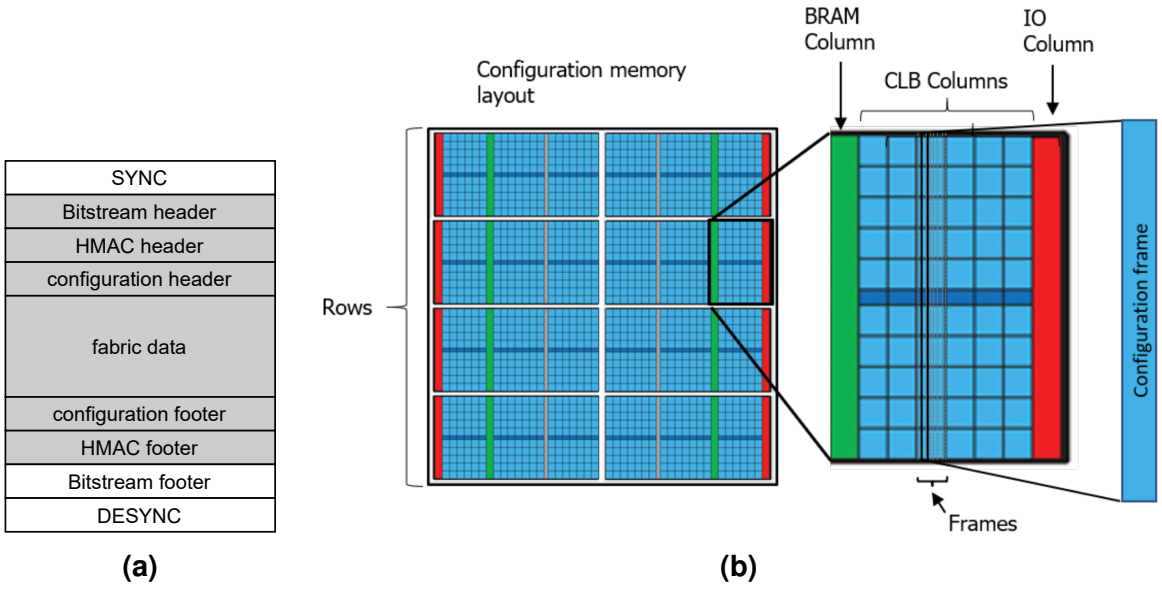


Figure 2.5.1: (**a**) Bitstream file structure. Shaded parts can be encrypted. (**b** Representation of the configuration memory layout arranged in data frames, taken from [Gio+19]

The bitstream format is publicly documented. However, the mapping of configuration bits (LUTs, PIPs etc.) is not. In the academic community, there exist various literature and tools that analyze and reverse engineer bitstream files. Most work is done to be able to modify the contents of LUTs or the interconnect (PIP) configuration [Yu+19; MD20].

The program BitMan [DHK17] is able to modify Xilinx bitstreams. This tool is written in ANSI C and its knowledge of bitstreams was acquired by reverse engineering. BitMan has support for geometric operations such as cutting, relocation, duplication and a number of low-level modifications on the contents of LUTs and BRAMs. For DPRS this tool is useful since it can extract partial bitstreams from a full bitstream. These partial bitstreams can be directly loaded using the available configuration port (e.g. ICAP) of the device. Furthermore, since the tool is able to modify the address information fields inside the bitstream, it is possible to perform module relocation on the bitstream level. Module stitching is another feature that the tool is capable of. This stitching property connects the interface of module tiles directly to each other on the bitstream level. We will use this stitching and merging feature of BitMan to verify the correctness of our work in Chapter 5.

2.6 TCL scripts

This work makes use of TCL scripts, therefore some basic background information is provided in this section. Vivado integrates TCL version 8.5 (whereas ISE 13.4 is using 8.4) [Xil19b] and is equipped with its own binary version of the TCL interpreter and shell. TCL is also pronounced as *tickle*. The GUI of Vivado includes a TCL console where commands can be directly executed. Almost every action from the GUI can also be performed with a corresponding TCL command. This allows working in project mode and non-project mode. In non-project mode, Vivado is purely controlled by TCL commands or scripts. All Vivado get_* commands (e.g `get_nets` for querying all nets in a design) returns a collection of data, see for example Listing 2.1.1. Basically, these collections are specialized wrappers around the 'list' and 'dict' data structures found in the TCL framework. Collections are limited in the number of elements they list when converted to a string representation. This limit can be adjusted with the following command:

```
set_param tcl.collectionResultDisplayLimit 0; 0=disable the limit.
```

Listing 2.6.1 shows a few basic TCL commands [Whe11; Tcl]. Variables are declared and initialized with the `set` command. Using the `$` operator (or the `set` command without the value argument) the value can be retrieved again. In TCL variables do not have a type, everything is considered as a string. Values that variables hold can be interpreted as numeric to perform mathematical operations. A group of elements can be handled as a list and various list operations are supported by the TCL interpreter. Furthermore, a key-valued lists are supported. These list are dictionaries and are declared using the *dict* keyword.

Procedures can be created using the `proc` command. This command replaces any existing procedure with the same name. TCL files can be arranged using namespaces. A namespace is a collection of commands and variables. This ensures that commands and variables don't interfere with each other. By default, everything is in the global namespace. Using the namespace *eval* command, a new namespace is created.

```tcl
#!/usr/bin/tclsh

# Variable declaration
set e 2.7182
puts $e;       # prints 2.7182
puts [set e];  # prints 2.7182

# List example
set alist {4 8 15 16 23}
set blist [list 4 8 15 16 23]; # another list initialization method
lappend alist 42
puts [lindex $alist 0]; # list index are zero-based, prints 4
puts [llength $alist];  # prints 6

# Dictionary example
set d [dict create]
dict append d key1 val1
dict append d key2 val2
puts [dict get $d key1]; # prints 'val1'

# Create a namespace
namespace eval example {
  namespace export example_proc
  variable x 1

  proc example_proc {} {
     variable x
     incr x
     puts $x
  }
}

# Calls the example_proc in the example namespace
example::example_proc; # prints 2
```

Listing 2.6.1: TCL example script showing basic commands. This scripts can be executed using the `tclsh <script.tcl>` command in a shell.

CHAPTER 3

RELATED WORK

This chapter gives an overview of the DPR tools of the leading FPGA vendors and the related academic tools found in the literature. A number of those tools have been selected for comparison on their features and (active) development status.

3.1 An Overview of DPR Tools

The major FPGA vendors do have support for DPR. Intel Altera supports this for their Cyclone, Arria, and Stratix devices with the Quartus Prime tool [Int20]. The PR design flow of Intel requires initial planning where the design is set up with one or more partitions and the placement in the floorplan. In the floorplan view, you define the static region, the PR place regions and routable regions for interfacing. The interface planner is used to create periphery floorplan assignments in the design. The next step is adding the PR controller to the project. The *personas* (how Intel names reconfigurable modules) are to be defined next. After that, the base revision for the design, as well as PR implementation revisions for each persona is created. The Intel PR flow works with project revisions to organize several versions in a single project. At this stage, the base revisions can be compiled together with an export of the static region. The last step is to generate the PR bitstream files and program the FPGA.

For Xilinx, designers can use PlanAhead for the ISE Design Suite or their latest software the Vivado Design Suite. In the Vivado IDE, the partial reconfiguration design flow is to be used for the Virtex, Zynq and UltraScale devices. Projects using PR have to be created with the option partial reconfiguration enabled [Xil20; Xil19c]. The designer then has to define the number of partitions in the project. To add and manage the RM and the RTL sources, the Partial Reconfiguration Wizard is used. At this point, the project can be synthesized. Each RM is assigned to a *Physical Block* (Pblock) by default. Floorplanning can be carried out to adjust and move the Pblocks in the device view. After passing the PR-specific checks, the implementation can be run to place and route all RM configurations and the static design. When the implementation has finished, running PR Verify is recommended to ensure consistency between static and reconfigurable part. The last step is to generate the (partial) bitstream files.

Both tool flows are similar and comparable to each other. However, the vendor tools do not come without limitations. For example, the partial region can only host a single module at a time. They do not have support for slot- or grid-style reconfigurable styles. This island-only style can lead to a non-optimal use of fabric area. Furthermore, the vendor tools work with a dependent design flow for building reconfigurable systems. The advantage of having a single project for the complete reconfigurable system is

that the configuration can be checked for integrity. A disadvantage is that, for example, when the static design requires a change, all of the reconfigurable modules need to be re-implemented, up to the generating of new partial bitstreams. For large designs, this is undesirable. This is not the case for an independent design flow. Here the static system and modules are created independently from each other. Another limitation is having no support for module relocation. When the same module is used (or loaded) at various locations on the fabric it is called module relocation. Module relocation reduces the storage space requirements of the reconfigurable modules and also reduces the number of variants required for a specific module.

To overcome some of these limitations, there exist several academic tools. As we will see later, design verification can be a challenge for such tools. The following sections are devoted to the academic tools for the Xilinx platform.

3.1.1 GoAhead

The GOAHEAD tool is the successor of ReCoBus-builder tool [BKT12; Bec+13] supporting more features and newer devices. This PR tool targets the Xilinx FPGAs for implementing run-time reconfigurable systems. The tool is designed for usability by abstracting most of the low-level details from the design engineer. Originally developed for the Xilinx ISE IDE, where it is able to interact through XDL with the Xilinx tools. Support for XDL was dropped in Vivado, but GoAhead has been adapted to emit TCL code instead. GoAhead is a stand-alone .NET C# application and provides a graphical user interface and a scripting interface. This scripting capability allows automating the implementation process in a single shot batch job. The tool provides floorplanning capabilities, communication interface generation, and constraints generation required for the implementation phase.

The tool flow of GoAhead is depicted in Figure 3.1.1b. In the planning phase, the interface specifications are defined and resource budgeting is performed to calculate the minimal size of the partial area. The designer has to take care of partitioning the design into a static and dynamic part. In Figure 3.1.1a the block view of the FPGA device is shown where each tile resource type has a distinct color. The block view is stored into device descriptions files (**.binFPGA*), a custom GoAhead format that comes along with the tool. After the planning phase, GoAhead is used to floorplan the design (Figure 3.1.1b). The definition of the reconfigurable areas can be selected manually or by using the custom script commands of the tool. From the selection, GoAhead generates the design templates (RTL sources) and constraint files. The RTL sources have to be added to the Vivado project. The constraints files are executed at different stages of the design. Using goa script files is a preferred method for building reconfigurable systems with GoAhead.

To create a project with GoAhead (Vivado) using scripts, the following has to be done.

- Define the static.goa file, which holds the commands:
  - Floorplan the partial area
  - Generate interface constraints
  - Generate connection primitives in VHDL format
  - Generate placement constraints
  - Generate blocker macro
- module.goa file

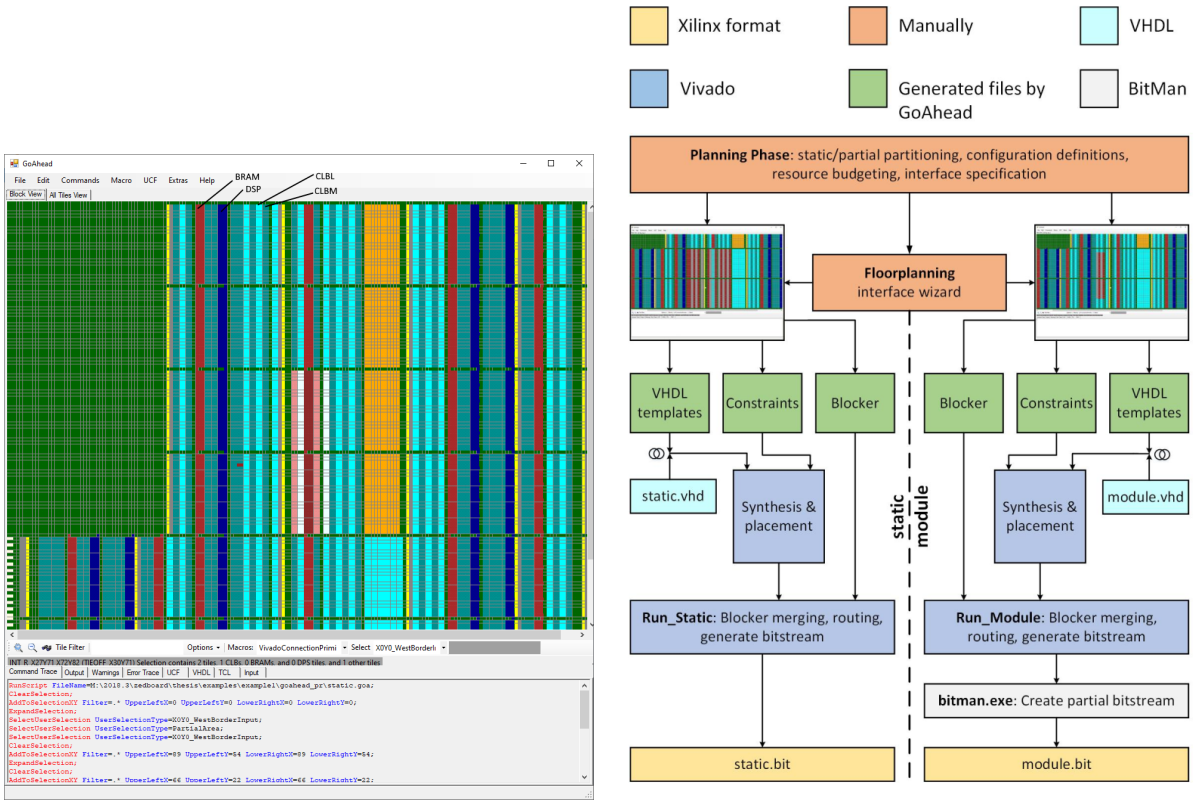


Figure 3.1.1: (**a**) The GUI of GoAhead. (**b**) The design flow for building reconfigurable systems with the GoAhead tool. The design of the static system and modules is completely separated. Floorplanning is done via de GUI of GoAhead. The corresponding constraints and VHDL templates are generated by the tool. After that, Vivado applies these files during the implementation phase of the design. Figure from [Hog19].

  - Floorplan the module area
  - Generate interface constraints
  - Generate connection primitives in VHDL format
  - Generate blocker macro
  - Generate placement constraints
- build.tcl script file for the static design and for each module.

Running the goa scripts generates all the RTL sources and constrains scripts. The RTL sources must be included in the FPGA project. For the constraints scripts, they are executed in specific order after synthesis.

3.1.2 IMPRESS

IMPRESS is an open-source automated tool for implementing reconfigurable systems [Zam+18; Zam+19; Zam+20] intended for the Xilinx Zynq SoC FPGAs. The framework extends the Vivado reconfiguration flow capabilities by including a library written in TCL language. It thereby overcomes some of the limitations of the Xilinx design flow, such as the inability of stacking multiple *Reconfigurable Partitions* (RPs) within a clock region [Zam+18]. The design flow for the IMPRESS framework is shown in Figure 3.1.2a. Different types of granularities are supported in the same reconfigurable

system (Figure 3.1.2b). For *Coarse-grain* reconfiguration style, multiple RPs can exist in the design. Each RP will hold only a single module. Exchanging modules during run-time adapts the behavior of the system. The *Medium-grain* style has the property that multiple RMs can exist in the same RP. Similar to the grid-style variant from Figure 2.3.3(c), this variant allows direct module-to-module communication. Useful, for example, in systolic array networks. The *Fine-grain* type has the capability to reconfigure low-level components. For example, by changing the equation of a LUT, the logic behavior can be altered during the run-time of the design.

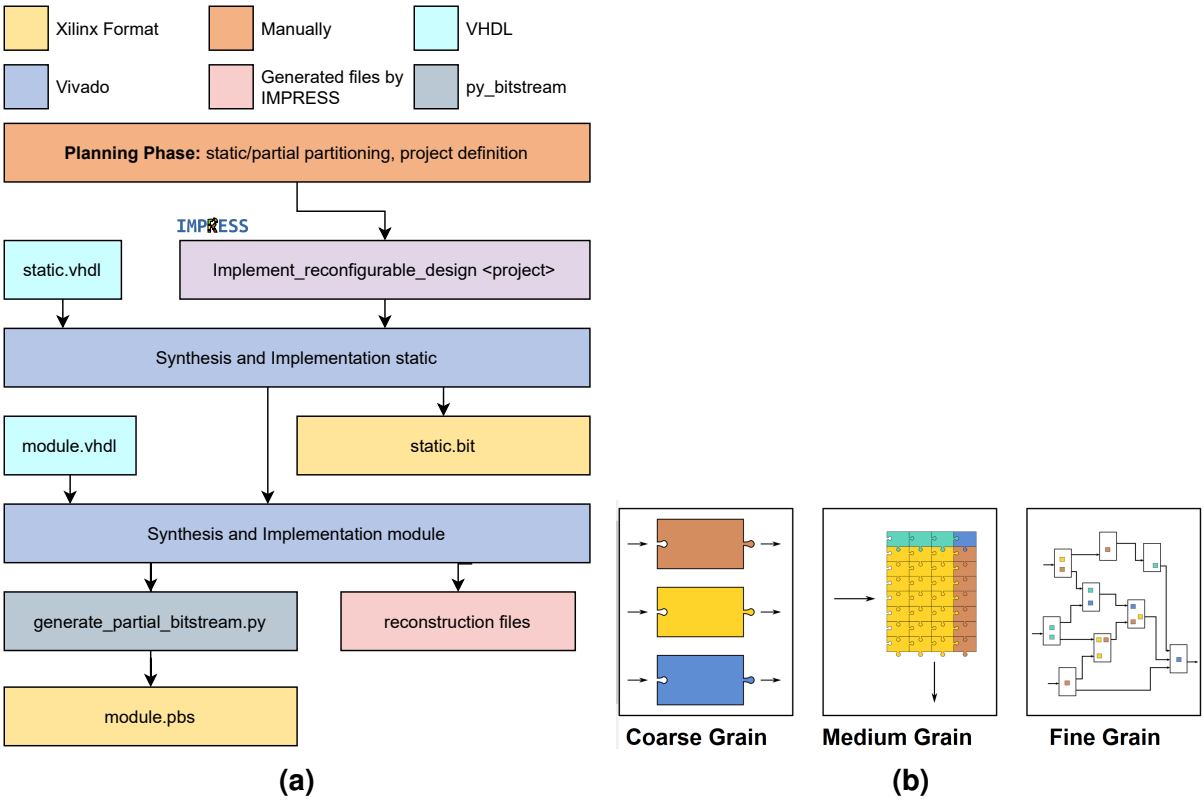


Figure 3.1.2: (**a**) The design flow for the IMPRESS framework. (**b**) IMPRESS and Multi-grain Reconfiguration (from https://des-cei.github.io/tools/impress).

Ease of use has a strong focus for the IMPRESS framework as mentioned by the authors. By providing a project-style architecture, it allows validation before any implementation is performed. This can be a time-saver since errors can be found in an early design stage. The number of manual (or intermediate) steps that the designer has to perform are kept minimal. After defining the system specification, the tool automatically performs all the implementation steps and (partial) bitstream generation. Three straightforward text files are used to define the system specification. Having a profound understanding of the FPGA fabric and architecture is therefore not required. The following specification files are setup during design time:

- *project_info file* is the starting point for setting up the reconfigurable system. Divided into three sections:
  - General settings, such as project name and FPGA chip to be used.
  - Source locations of the static system.

  - Source locations of the reconfigurable modules and the partition group definition per module.

- *virtual_architecture* defines the architecture. E.g. the number of partitions in the static design, the partition size, location, and reference to the interface file.

- *interface* file defines the local and global net interfaces.

The listings in Appendix A A.0.3, A.0.2 and A.0.1 show an example configuration. This configuration was used for further clarification, understanding, and reference. The decoupling of the implementation into a static system and reconfigurable modules (also called the partition phase) is a manual task. This is done on the source level of the project. When the project files have been created and the design is partitioned, a call to the IMPRESS framework can be issued in the TCL console from Vivado:

```
source /home/jeroen/git/impress/design_time/reconfiguration_tool/IMPRESS.tcl
implement_reconfigurable_design /home/jeroen/git/impress/examples/coarse_grain/project_info
```

Listing 3.1.1: Implement the example reconfigurable design

The first command makes the IMPRESS library available in Vivado by evaluating all the source files specified in the file given by the argument. The second implements the system previously defined in the project file. During synthesis and implementation, the design checkpoints are stored in the *project_name* directory near the project specification input files. The generated bitstream files are stored in a subfolder. These files include the full bitstream of the static system and, for each module, a *Partial Bitstream* (PB) file. Additionally, an info file and corresponding DCP file can be reviewed to find potential issues that might have occurred during the implementation phase.

Regarding run-time management, the generated PBs are intended (and prepared) for the run-time application. This application runs on one of the two ARM processor cores of the Zynq FPGA, called the *Processing Subsystem* (PS). The PS has full read and write access to the FPGA configuration memory through the PCAP interface. This enables modifications to the circuit structure and functionality during operation of the *Programmable Logic* (PL) part of the FPGA.

The following steps are performed when issuing the commands from Listing 3.1.1:

1. Parsing the project and virtual architecture files.
2. Setup of the Vivado project and create the output folder structure.
3. Static system generation.
4. Reconfigurable module generation.

Steps 1 and 2 do not require much further clarification. We continue with the system implementation flow: static system and reconfigurable module generation. After parsing the project input files the static system is synthesized.

- *obtain_interface_info* saves the interface information of all the reconfigurable partition groups to a TCL variable.

- *obtain_xilinx_and_custom_pblocks_format* takes a Pblock and saves its properties (e.g the size) to two formats: the Xilinx format and the custom format for the MORA PBS extractor tool.

- *obtain_max_DSP_and_RAM_for_reconfigurable_partitions* determines the number of DSP and *Random-Access Memory* (RAM) components (number of tiles) for each reconfigurable partition.

The RPs are arranged by their compatible footprint as well as with their compatible interface. IMPRESS makes use of all inputs of the LUTs. Observing the implementation of the modules with respect to interfacing. The user can specify which border is an input or an output. The number of usable tiles, that is the width, can be defined by a suffix after specifying the cardinal direction. For example, defining *SOUTH_0:3* for the interface allows that four tiles are to be used on the southern border. Additional standalone signal routing is possible. For example, a commonly used asynchronous reset signal brings the FPGA system to its initial state. In GoAhead, it is possible to configure the interface for all four borders separately. For each border, the designer can adjust the width, the width in the number of bits the interface must have. Furthermore, the border can be bidirectional. Meaning it has separate input and output bus[1]. Note that we do not mean a bidirectional bus here (declared with `INOUT` keyword in VHDL).

3.1.2.1 Design reconstruction

The term *design reconstruction* refers to the operation on the previously decoupled designs reconstructing into a fully implemented and single design. This implemented design is identical to the run-time design when all partial bitstreams are loaded. Reconstruction is performed on the netlist level of the design since the place and route constraint must be kept for all designs. IMPRESS framework offers support for design reconstruction. A method is included to merge a module into the static system at design-time. For this to work, the reconfigurable regions must be the same for static system and module, therefore the design reconstruction is only available for coarse-grained reconfigurable systems. The provided method (a TCL proc) requires additional information, which is output during the build and implementation phase of the reconfigurable system. This additional information is stored in two text files and includes:

- For each module, the clock, reset and interface nets and its route string property.
- Placement constraints of certain cells (e.g. the LOCK_PINS etc.).

The command listed in Listing 3.1.2 merges the *div* reconfigurable module inside the reconfigurable partition of the static design. After completion, the result is a fully implemented design, which can be validated using the simulator or by running a timing analysis.

```
1 insert_reconfigurable_module pblock_reconf_part_0 <path to
  ↪  IMPRESS>/examples/coarse_grain/impress_build/DESIGN_RECONSTRUCTION/group1_div.dcp
```

Listing 3.1.2: Example command for design reconstruction with the IMPRESS framework.

[1] For example the `STD_LOGIC_VECTOR()`

3.1.3 TedTCL

TedTCL (TED: Tcl/Tk EDA Development) is an extension library to ease the use of the TCL *Application Programming Interface* (API) of Vivado [Ves19; Vai+20]. Written in TCL, it implements identical features for partial reconfiguration that are available in GoAhead. This library is an extension for Vivado. TedTCL uses predefined connectors that form the fixed wire assignment for a signal bundle. The connectors must use the identical wires for the static system and modules to be able to communicate with each other. For preventing wires routed through the reconfigurable region, TedTCL uses a blocker (Figure 2.3.2). The blocker functions are using physical nets (the GROUND or POWER nets) to prohibit using the wires from the interconnect tiles. The blocker blocks all wires that are crossing the border of the Pblock, this results in a fence around the module region. According to [Ves19], the performance of the blocker from TedTCL outperforms the one from GoAhead, mainly because of the direct internal representation of wires to block. TedTCL blocks only the wires that go across a tile, resulting in a fence of only a single tile wide around the partial region. TedTCL features extended support for clock routing. At the moment of writing, this feature has not been compared extensively with the clock routing capabilities of GoAhead. GoAhead can generate scripts for reconnecting the clocks to/in the partial area. This can be achieved by adding the *ConnectClockPins* command in the GoAhead scripts. The command generates a TCL script containing the statements to reconnect all the clock pins found in the partial area to a predefined clock net. For example, the clockpin input from all the flip-flops that each site contains.

3.2 Comparison and Differences between Frameworks

Starting with Xilinx Vivado, they have built-in support for PR, but island-style only. Frameworks such as [Gli+19] completely automate the generation of the partial bitstreams by compiling and configuring RMs onto the PR region, without the use of vendor tools. Dreams [OdR12] has the disadvantage of that modules cannot communicate if they are located nonadjacent. If modules are located nonadjacent, we can configure the slots in between modules such that they connect these modules. Those modules are called bypass modules.

Various academic tools were compared on relevant properties on the design flow of implementation PR systems. In particular, the GoAhead tool and IMPRESS framework. They offer more or less the same functionality and there is still active development on them. For each property in Table 3.2.1:

- Module relocation, the possibility to re-use modules in different partial regions.
- Style, the reconfiguration style, e.g. island or any of the other styles.
- Partition Interface, this is how the communication is defined between static system and partial modules.
- Routing Isolation, how the routing is constrained not to use certain regions.
- Automated flow, is the PR flow automated, or does it require additional user steps.
- Design reconstruction, can the design be reconstructed to a full design.
- System architecture description, Does the tool work with a system configuration description.

Tool / Feature	Xilinx Vivado[Xil18b]	GoAhead[BKT12]	IMPRESS[Zam+18]	TedTCL[Ves19]	CoPR[VF14]	DREAMS[OdR12]	RePaBit[RFG16]
FPGAs	V7,Zynq, UltraScale	V4...V7,S6, Zynq, UltraScale	Zynq	Zynq,UltraScale	Zynq	V5,S6	Zynq
Module relocation	No	Yes	Yes	Yes	No	Yes	Yes
Reconfiguration styles	Island	All	All	Island	Island	All	Island
Partition interface	Proxy logic	LUT binding in HDL	Virtual	Virtual	(Xilinx) Proxy logic	Virtual	Bus macro
Routing isolation	Xilinx Internal	Blocker	Blocker	Dynamic Blocker	Xilinx Internal	Custom router	Isolation Design Flow
Independent design flow	No	Yes	Yes	?[1]	No	Yes	No
Automated tool flow	Yes	No[2]	Yes	?	Yes	Yes	Yes
Design reconstruction	Build-in	Not yet	Yes[3]	No	No	No	No
System architecture description	Yes	No	Yes	No	Yes	Yes	Yes

Table 3.2.1: Comparison of various DPR tools

3.2.1 GoAhead and IMPRESS

GoAhead has a built-in model of the FPGA fabric interconnections (or a graphical device view), TedTCL and IMPRESS do not offer such functionality. The model has the advantage that tile selections can be done graphically and the internal PIP connections can be shown. A downside is that the chip family database (and possibly the GUI) has to be maintained separately. Another major difference is that the generated TCL constraints scripts from GoAhead must be tied into the build process of the reconfigurable system. They must be executed at the right time during synthesis and implementation flow within Vivado.

Since GoAhead exists for a long while, it has support for more (older) devices. Over time, it is extended to support more devices and additional features that have been implemented. The custom scripting interface gives access to all commands and macros and gives the ability to replay the commands later. To make use of all commands, it is required to have an in-depth technical knowledge of GoAhead and the FPGA chip to be used. Furthermore, since the tool flow for modules and static design is decoupled, the designer must keep track of the consistency and integrity between modules and static design. Especially for the correct placement of the interface. For example, changing the partition size requires the designer to adjust the location (coordinates) of the interface location and tunnels as well as the size of the blocker macro. Changing a setting or parameter can result in changes required elsewhere, something the designer needs to keep in mind.

Regarding floorplanning, it can be considered more effortless for the IMPRESS framework since it automates most steps. The designer does not have to know the details of the (GoAhead) commands or have an understanding of the FPGA structure. The project files from IMPRESS are of a similar approach as floorplanning phase of GoAhead. Main difference is that GoAhead generates TCL scripts based on predefined *.goa* configuration files for the static system and for each module. These must be executed at the right time during the build of the project. Whereas IMPRESS takes care of this by executing all the necessary commands directly in Vivado when reading the project system architecture files.

Currently, the build process builds all modules and the static system. Rebuilding a single module is not possible with IMPRESS framework Whereas GoAhead lets you build a single module by running the template script for each module and perform an implementation run.

GoAhead supports a number of devices, but not all features might have been implemented for all devices. IMPRESS only supports the Zynq series FPGAs, but the

[1] Independent or automated flow is not mentioned in the literature.
[2] Not fully automated, manual and individual steps have to be performed.
[3] This framework only supports design reconstruction for island-style PR.

tool could be extended to support more devices.

Regarding interface architecture, the IMPRESS framework uses fixed nodes in the device which are shared between static and reconfigurable regions. This virtual interface has the advantage of having no overhead in the RTL source code. In GoAhead, these so-called interface connection primitives need to be declared and instantiated in the VHDL source code. Note that for both tools there is no (logic) overhead in the final design.

For GoAhead, each interface located the (cardinal) border. The number of signals must be specified, as well as the number of tiles (CLBs) to use and reserve for the communication. Both must relate and follow each other. Increasing the number of signals also means adjusting the number of tiles to reserve for the connection macros. Furthermore, it must match with the number of LUTs and LUT-pins available for each CLB.

GoAhead is developed with Visual Studio adding extended debug capabilities to the application (e.g. breakpoints and intermediate views). For TCL based frameworks such as TedTCL and IMPRESS, those debugging capabilities are limited and in general, do not go without a code change (e.g. adding print and assert statements.). On the other hand, GoAhead has no direct access to an open FPGA design in Vivado. It can only interact indirectly via script files. This makes it impossible to make some specific design changes. GoAhead requires the device files to be available for each device to retrieve the fabric information, whereas IMPRESS can just query this in Vivado.

For comparison, example1 (from Section 6.1) is used. This example has one module and one partition. RTL source code is identical, the static system The partition has a size of 4×50 tiles, the shape is a rectangle placed from tile X34Y99 to X38Y50. The designs were built consecutively on the same desktop computer. GoAhead took about 31 minutes and 47 seconds to complete the build[6]. The IMPRESS framework completed in 8 minutes and 52 seconds. The difference in time is mainly due to the number of nodes that are blocked. The static blocker has 32750 nodes and the module blocker has 175040 nodes blocked. 14301 nodes are blocked by IMPRESS, only single fence nodes.

GoAhead blocks all wires around the reconfigurable partition. A GND net is created and all nodes to be blocked are assigned to that net (Listing 3.2.2). This ensures that the vendor router algorithm does not use these nodes (or wires).

The IMPRESS framework uses a similar approach for blocking nodes. There a blocking net is constructed by a set of nodes which they call a fence. The same fence net is used for the static design and for the reconfigurable module. Nodes are assigned to the net by using the `FIXED_ROUTE` property to prohibit the router to use these nodes (Listing 3.2.1). They are terminated by using the input buffer (IBUF) and output buffer (OBUF) primitives (see Figure 3.2.1).

The blocking algorithm from IMPRESS is faster than the one used in GoAhead as we have seen above. GoAhead blocks a lot more wires and this takes considerable more time to assign the blocking net and route the design. The fence created by GoAhead is thicker. Especially for the module, a whole lot more wires are blocked. At this point, it is not known if this is done by design or that it was required at some point. It could also be the case that the algorithm is less optimal in certain situations.

The partial bitstreams are generated after a successful build of the module by the IMPRESS framework. GoAhead relies on BitMan tool to do this. The designer has to

[6]This is without generating the partial bitstream.

command the BITMAN tool to slice out the module bitstream files for each module.

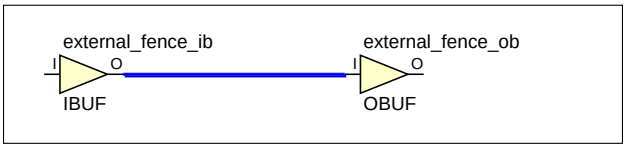


Figure 3.2.1: IMPRESS blocker net in the RTL schematic.

```
# IMPRESS:
# <skipped some code >
# Add the FIXED_ROUTE constraint
set froute [list]
  foreach node $nodes {
    lappend froute GAP $node
  }
set_property FIXED_ROUTE [list $froute] $net

# Final route string:
get_property FIXED_ROUTE [get_nets external_fence]
 { IOB_IBUF1 GAP INT_L_X38Y54/EE4BEG3 GAP INT_L_X38Y64/NE6BEG0 GAP INT_L_X38Y64/NE6BEG1 GAP
 ↪  INT_L_X38Y64/NE6BEG2 GAP ... }
```

Listing 3.2.1: Blocker script of IMPRESS (partly listed).

```
# GoAhead
create_net blocker_net_BlockSelection
create_pin -direction OUT gnd_for_BlockSelection/G
connect_net -net blocker_net_BlockSelection -objects [get_pins gnd_for_BlockSelection/G]
set_property ROUTE "( \ { INT_L_X34Y99/LVB_L12 INT_L_X34Y99/WW4BEG2 } \ { ... } ... )" [get_nets
↪  blocker_net_BlockSelection]
```

Listing 3.2.2: Blocker script of GoAhead (partly listed).

3.3 Module Stitching and Rapid Overlay

For this work, there was a need to modify the routing on a low-level (netlist) basis of the design (as we will see later in Chapter 5). Tools that are capable of interaction with the netlist such as RapidSmith [NTH18] and RapidWright [LK18] have been reviewed and considered for this work. Those tools allow the designer to build customized FPGA implementations and operate on a netlist level of the design. However, installation of additional software is required, whereas Vivado already has most of all the necessary functionality.

In [YKL15], a method is presented for module stitching without invoking the router. This so-called stitching operation is also called *zipping*, connects adjacent modules directly, without logic overhead. The long place-and-route process is a growing concern and the vendor tools try to accelerate the compilation using different techniques. Overlays are a way to solve reconfigurable computing problems. Those overlay architectures act as pre-compiled circuits. They propose a tool called ROB (Rapid Overlay

Builder) that effectively builds those overlays from any logic design. This tool addresses the issue of long place and route times that overlay architecture can have. It can place and route any logic design into a set of adjacent modules.

CHAPTER 4

PROPOSITION

As we have seen in Chapter 3, the frameworks that implement their own PR flow can have limited test capabilities. Table 3.2.1 shows that timing analysis is not supported by GoAhead. The IMPRESS framework has better test and verification support than GoAhead. The automated design flow checks the system configuration for potential mistakes. Interface alignment issues can not occur, because, during implementation, the border nodes from the static system are used again for the modules. Furthermore, it has support design reconstruction, capable of building a fully placed and routed design. However, this design reconstruction is only supported for island reconfiguration style, fine-grained reconfiguration style is not supported. Currently, GoAhead only generates the constraints files, but can not verify the configuration or to perform timing analysis. Therefore, the main contribution of this work is to extend the GoAhead tool flow with design reconstruction capabilities. This enables timing analysis for designs generated with the GoAhead tool flow. It will have support for more fine-grained architectures, such as the grid-style variant.

The in this work proposed method (or tool) allows reconstructing the design for a given module configuration. It is a contribution for designs generated for the GoAhead tool flow, by adding configuration validation and timing analysis. This reconstruction overcomes some of the limitations on the testability of slot- and grid-style reconfigurable architectures as mentioned before. The size of the reconfigurable region can be of size $M \times N$, where M and N are of arbitrary size. To build a grid-style design, the GoAhead tool is used to generate the design constraints for the reconfigurable FPGA system. Figure 4.0.1 shows a general overview of the design flow together with the GoAhead tool flow.

The static system is modified to create a fully placed and routed netlist by combining and merging the logic and routing configuration of multiple designs into one. During this reconstruction, we can verify the correctness of a given set of modules and detect interface connection errors. With the fully placed and routed netlist we can apply timing analysis and timing simulation.

The full design must be identical to that of the run-time design, so some important physical constraints must remain intact. The logic must be restored in the exact same place on the fabric. Routing constraints must be kept, i.e. the signals must use identical wires on the fabric. Route and placement constraints are device-dependent. Therefore we operate on the post-implemented design which is the gate-level netlist.

The tool will aid the user to find mistakes that would otherwise leave the design in a non-functional state. It will be able to detect interface connection errors. GoAhead will generate the design constraints for the reconfigurable system. After completion of the implementation phase in Vivado, a merge script can be executed. The script uses

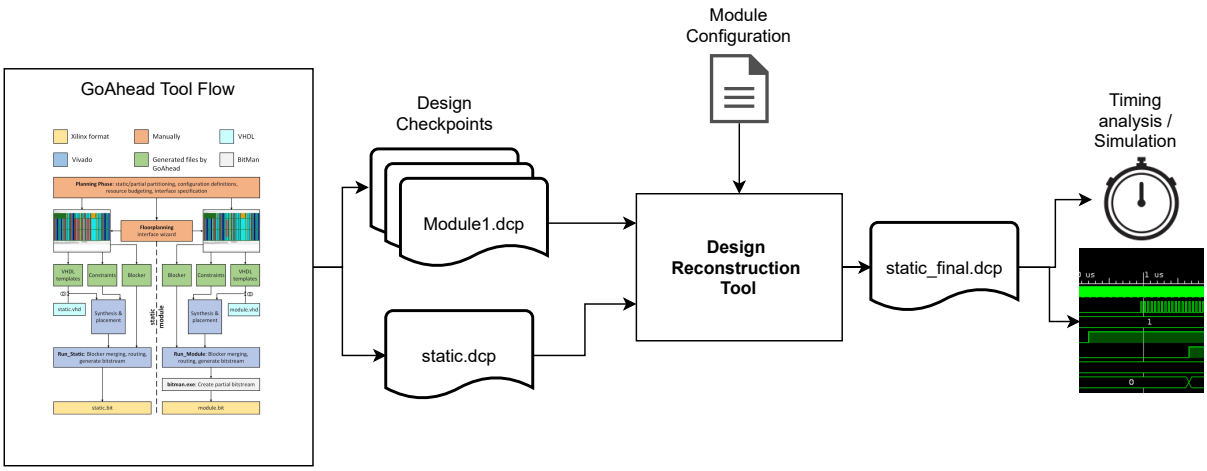


Figure 4.0.1: Overview of the proposed method.

the design checkpoints and reconstructs the design with the following properties:

- The user must be able to provide the interface mapping, module configurations.
- All configurations with all variants can be restored.
- Checks if valid interface mapping, e.g. signal width matches.
- For each module, if the signals match up with the dedicated wires, pin layout should be equivalent.
- The design should be reconstructed in such a way that it is functional and performance-wise identical as the design in run-time.
- Automatic timing analysis for each design.

TCL scripting and Vivado was chosen mainly because:

- Direct interaction with an open design.
- Has the knowledge of the fabric of the used device.
- Able to place and route the design manually and visually inspect the result.
- Report the timing properties.
- Equipped with a simulator.

4.1 Merging Modules and Variants

A variant of a module has the same behavior, but different hardware implementation. As each variant has a different hardware layout, the timing (and other properties) may vary. To verify the complete design, all possible (useful) module configurations have to be tested. Furthermore, modules may have other placements constraints. The footprint of a module is such a constraint. The module might not fit into a given location based on the hardware requirements (e.g. tiles that contain DSPs or BRAMs blocks etc.).

4.2 Features

Figure 4.2.1 shows a correctly placed module. To mention some of the properties that the design reconstruction tool can check. For example, in Figure 4.2.2a the number of

inputs or outputs signals of the static design, does not match with the (reconfigurable) module interface. We can detect possible alignment errors that may occur as shown in Figure 4.2.2b, where a different wire is expected on the fabric level. Another verification feature is the detection of open circuits, or to check that the correct interface is used or present (Figure 4.2.2c). In that figure, the input signal is expected at the west border of the module, but the signal is applied at the north border. Furthermore, able to detect pin or wire swaps (Figure 4.2.2d). This can happen if, for example, the synthesis tool tries to optimize the design, resulting in pin or wire swaps near or at the border.

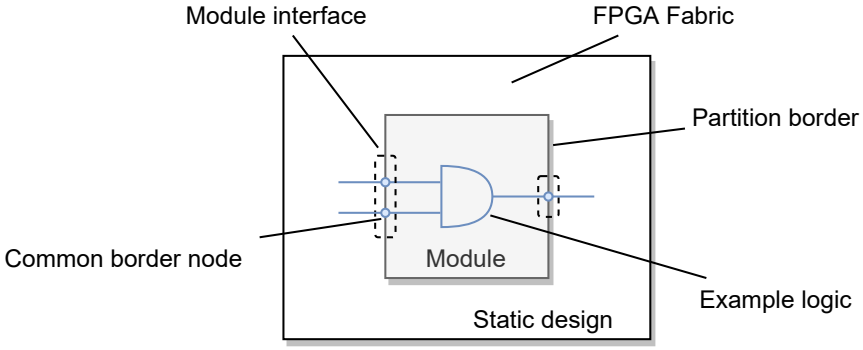


Figure 4.2.1: Example of a correctly placed module on the FPGA fabric.

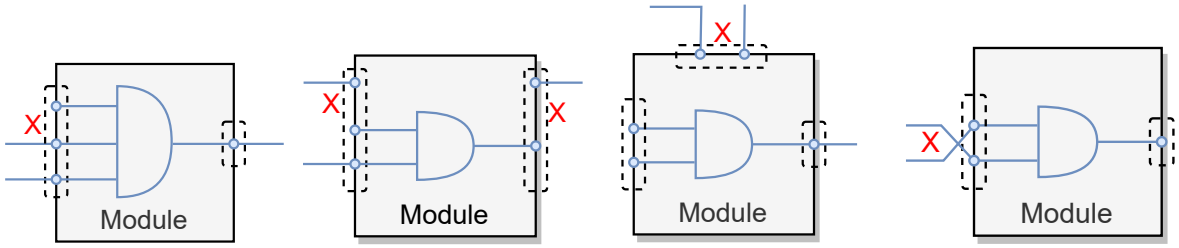


Figure 4.2.2: Problems that might occur when incompatible modules are placed: (**a**) the number of provided interface signals mismatches, (**b**) wrong wire alignment on the interface border, (**c**) the module expects input from the West border, but the North border is provided and (**d**) the wrong pins plugged at the border.

Regarding the placement of the module itself, misplacement can be detected in advance. To ensure that no overlap occurs and the module is placed in the correct location of the fabric. For example in Figure 4.2.3. After the placement of Module 2, the configuration of Module 1 is overwritten, leaving the design in an undefined state. It could also happen that the module is placed in (a part of) the static region.

For PR, the design is split into a static and one or more dynamic parts. Although we are capable of testing reconfigurable modules individually, the decoupling makes it impossible to do a timing analysis on the whole system. Furthermore, the placement constraints that have been put on the design can affect the timing of the system as well. For example, the output signals are routed alongside the partial region, the length of the signal paths increases, affecting the timing properties of that signal. After reconstruction, we end up with a fully placed and routed netlist.

Timing analysis (or timing simulation) checks if the design can operate at the specified clock frequency. In case of timing violations, changes have to be made to the de-

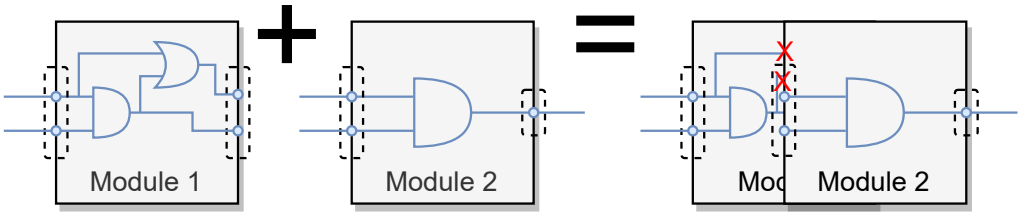


Figure 4.2.3: Overlap of modules after placement. The configuration of Module 1 is overwritten by Module 2.

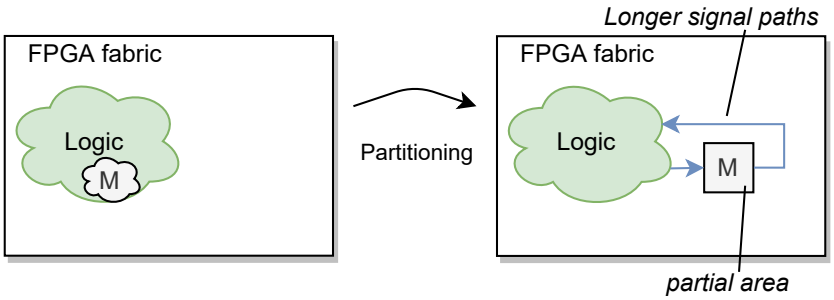


Figure 4.2.4: Longer signal paths, different timing properties.

sign, such as choosing a different module configuration or lower the clock frequency. Different module configurations may lead to different operating frequencies, therefore we could find a more optimal module placement concerning the clock frequency.

In the final stage, we end up with a full bitstream[1] that can also be used as an initial configuration.

[1] Can additionally be performed on bitstream level with BitMan[DHK17]

CHAPTER 5

IMPLEMENTATION

This section describes the implementation for the *Design Reconstruction Tool* (DRT) as proposed in the previous chapter. The DRT takes care of merging modules into the static design. It is intended for the designs created with the GoAhead tool and uses the TCL scripting of Vivado. The starting point is the DCP files that are the output of the synthesis tool. They contain a snapshot of the design which includes the netlist, constraints, and implementation results. The DCP files are generated in conjunction with the constraints files from the GoAhead tool. In total we have:

- TCL library: *drt.tcl* (namespace eval drt).
- Input DCP files: modules and static design.
- A mapping and module configuration definition.

5.1 Design Reconstruction

Regarding the design reconstruction, there are some points of attention. Starting with the nets, they can only make use of the already existing nodes. Any other modifications must be prevented at all times. All nets should be properly terminated and not left floating, but as we see later, this is not always possible.

The connection logic outside the module partition needs to be removed. This logic is not present during runtime and can cause issues in the implementation of the static part of the design. Furthermore, we must check that no logic gets unplaced during the placement of the module logic. Unplaced logic can be a sign of logic overlap in the design and therefore it might not function as intended.

During run-time reconfiguration we can have antennas, but during design time this is not allowed. The design will not pass the design rule checks DRC (unless we lower the rule check).

The following subsections describes the sequence of the reconstruction step-by-step:

- Prepare the designs (5.1.1).
- Validate the user input, check existence of input files and mapping script (5.1.2).
- Preserve anchor logic, before placing modules (5.1.3).
- Place the modules into the static design (5.1.4).
- Reconnect all interface nets (5.1.5).
- Restore clock logic (5.1.6).
- Restore anchor logic s2p (5.1.7).
- Final route of the design and bitstream generation (5.1.8).

After performing these steps, we can do a timing analysis and run a simulation. Since we use Vivado in non-project mode, we must enter the commands manually in the console. The DRT includes a script to launch the simulator on non-project mode. It takes a DCP file and a simulation script to set up and run the simulation. Those commands and steps are described in Section 5.2.

5.1.1 Prepare the Designs

During the implementation phase, the design is optimized. One such optimization is flattening the hierarchy of the design. This flatting will, for example, coalesce certain nets into a single net. Those nets will get a different name other than used in the HDL source. Although it is not a strict requirement for the design reconstruction, adding annotations to the interface signals and keeping the design hierarchy makes it easier to do the signal mapping[1]. Without the signal annotating, the implemented design has to be opened and searched for the interface nets by hand.

In Vivado there exist a number of attributes to constraint the HDL design [Xil18c]. The Save (S) constraint is a mapping constraint. During mapping, the nets are absorbed into logic blocks and some elements are optimized away. The *S* attribute prevents such optimizations in order to preserve access to nets in the post-synthesis netlist. The Keep (KEEP) constraint preserves signals in the netlist. Both *S* and *KEEP* constraints are applied to a signal in the VHDL source code, an example is shown in Listing 5.1.1. To disable any flatting we can use *flatten_hierarchy none* with the synthesis command (Listing 5.1.2).

```
1 -- attribute_declaration
2 attribute s : string;
3 attribute keep : string;
4
5 -- attribute_assignment
6 attribute s of x0y0_s2p_w : signal is "true";
7 attribute keep of x0y0_s2p_w : signal is "true";
```

Listing 5.1.1: Signal names to 'keep' by defining the attributes in VHDL source.

```
1 # Synthesize design
2 synth_design -part xc7z020clg484-1 -top top -keep_equivalent_registers -flatten_hierarchy none
```

Listing 5.1.2: To keep the (signal) hierarchy, add the *flatten_hierarchy* switch to the synthesize command.

5.1.2 Validate User Input

The interface definition for each module configuration must be provided beforehand. For certain designs (e.g. when there is a one-to-one relation), we could deduce this from the module implementation. Regarding grid-style architecture, this deduction is

[1] Note that this can lead to a less optimal design

not always possible since the same module can be placed at various locations and using a different interface. For each configuration the user has to define:

- The runtime module configuration.
- The interface mapping.
- The name of the clock net.
- Indicate the unused interfaces (or slots).

The module mapping configuration is defined in a TCL script file (see Listing 5.1.3). It contains a list of modules to be placed, together with the interface mapping. The mapping defines which input is mapped to which output. It is possible to map each signal like $x[0] = y[0], x[1] = y[1], ..., x[n] = y[n]$ manually. However, Vivado allows the use of the * wildcard in the index field. If all the index numbers of the signals line up with their index, manual unrolling is not required.

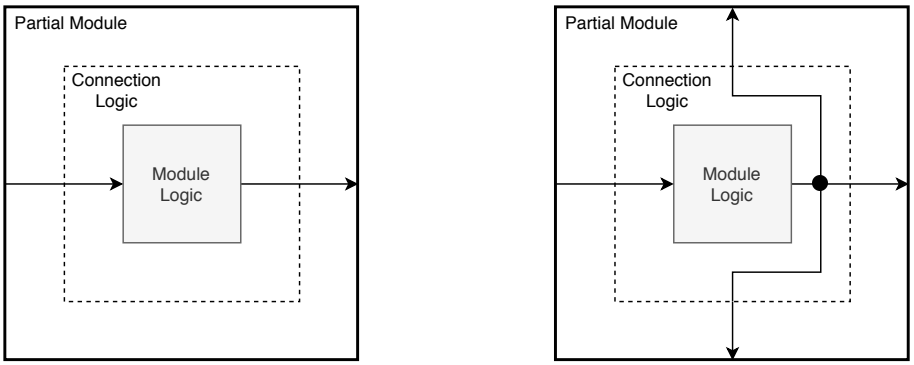
```
# Define a list of modules
set modules [list module1 module2 ...]

# Define the interface mapping
set mapping {
  {x0y0_s2p_w[*]} {module1/inst_ConnectionPrimitiveWestInput/x0y0_s2p_w[*]}
  {module1/inst_ConnectionPrimitiveWestOutput/x0y0_p2s_w[*]} {x0y0_p2s_w[*]}
  ...
}

# Define the unused s2p and p2s nets
set unused_s2p [list]
set unused_p2s [list]

# Define the clock net of the module instance
set internal_module_clk "clk"

# Define the clock net outside the partial area of the module
set external_module_clk "fclk_clk0"

# Assign the config, do not change
set modules_config [list {*}$modules]
set mapping_config $mapping
```

Listing 5.1.3: Example of the mapping script file. Here were define the list of modules, the interface definition, and the name of the clock net. The script is sourced during the execution of the DRT.

For designs generated with GoAhead, we can identify two hierarchical levels: the connection logic and the module logic (Figure 5.1.1). The connection logic can be seen as the switch box for routing in- and output signals outside the partial module. The module logic contains the functional implementation for the partial module. The number of in- and output ports from the connection logic can differ from the module Logic. In Figure 5.1.1b the output of the module logic is connected to 3 outputs ports of the partial module. The connection logic net name is used in the mapping script.

5.1.3 Preserve Routing and Anchor Logic

Some of the anchor logic might be required to get design closure. This is the case when, for example, not all the ports from the partial area of the static design are used. The anchor logic is removed when turning the partial area into a black box. The black



(a) **(b)**

Figure 5.1.1: Partial module connection logic examples. (**a**) Connection logic with 1 input and 1 output connection. Connection logic with 1 input and 3 output connections. (**b**) Here the single output of the module logic is present on the 3 outputs of the partial module.

box cell has all of its logic removed and signals are chopped off at the border of the partial area. Those signals are now defined to be antennas and therefore will not pass the DRC. The partition nets are connected to anchor LUTs in the partial area. Those LUTs are (leaf) cells and need to be restored as well. For each LUT cell, we have to preserve the routing, pins, and the configuration (equation) of the anchor LUT.

Physical constraints have to be added to the implemented designs. These are the lock and fix route constraints to prevent the vendor tool to make any changes or do optimizations. In particular, the *LOCK_PINS* constraint needs to be applied to all LUTs. This property specifies the mapping between logical LUT inputs and the physical LUT inputs [Xil18d, p. 150]. An example is shown in Listing 5.1.4. After placement, we have seen that pin swaps at the LUTs can occur (see Figure 5.1.2), which, as observed, can not always be resolved later on.

```
# lock the pins on the LUT
set_property LOCK_PINS {I0:A3 I1:A2 I2:A1 I3:A4 I4:A5} [get_cells $lut_cell]

# fix the route string
set_property IS_ROUTE_FIXED 1 [get_nets -hierarchical]
```

Listing 5.1.4: Example on how to apply the design constraints.

Reason to believe that this is a kind of optimization that happens in the background, e.g. using the fastest A6 input of a 6LUT or the A5 input for a 5LUT [Xil18b, p. 260]. During the implementation of the design, LUT pins are mapped in order from highest to lowest by default. The highest logical pin is mapped to the highest physical pin.

Furthermore, the *IS_ROUTE_FIXED* property is applied to each net. This prevents any changes in routing that could otherwise affect the timing and behavior of the final design.

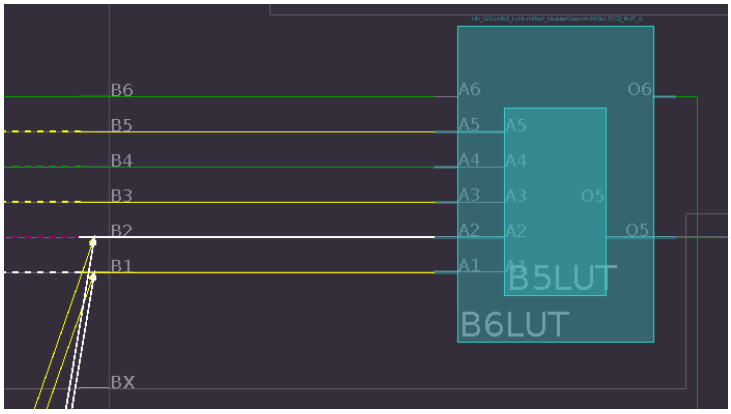


Figure 5.1.2: Input pins A1 and A2 are swapped. This can happen if the net has a different end point in the route string than the physical pin assignment of the LUT itself.

5.1.4 Place the Modules

We can add logic from another design into a cell of static design with the `read_checkpoint -cell <the cell>` command. This command reads a DCP file containing the implemented design and populates the given cell with the netlist from the checkpoint. The command on its own is not suitable for merging multiple modules in the partial area. It can only be used once on an empty blackbox. Therefore, in conjunction, the `add_cells_to_pblock <pblock> [<cells>...]` is used. First, an empty cell instance is created with the `create_cell` command. Normally, this command expects the `-reference <arg>` to be a type from the library cell. The Library Cell contains all the cells that are applicable for the current device of the design. It was found that Vivado creates custom cells with the `-reference` argument that have a unique name (i.e. not found in the Library Cell). This way, we can create an empty cell for each module that we have. For each module, a Pblock is created. The empty cell is added to this Pblock. After that, the content of the module cell is added to this cell by reading the checkpoint.

When all modules have been placed, the design is inspected on the existence of any unplaced logic cells. The behavior of Vivado is that cells placed on the fabric will be unplaced when new cells are placed on the same position. Using the `get_cells filter {STATUS=="UNPLACED"}` command, we are able to find such cells and report this to the user. Note that if any unplaced cells are found, the tool can not continue and the user has to manually resolve this.

5.1.5 Reconnect the Interface Nets

After module placement, the interface nets from the modules are placed on top of the interface nets of the static system. The nets have to be fused since this is not performed by the vendor tool by default. This is also called *module stitching*, where the nets are stitched together. Communication can only happen when two interface signals share a common node at the border of the reconfigurable region. In case when the grid-style reconfigurable architecture is used, module-to-module communication happens via the common node at the border of the modules. Therefore we have to identify the node (or the physical wire) that crosses the interface border. If no such node is present, the module will not function as intended. Figure 5.1.3 show this

overlap in routing.

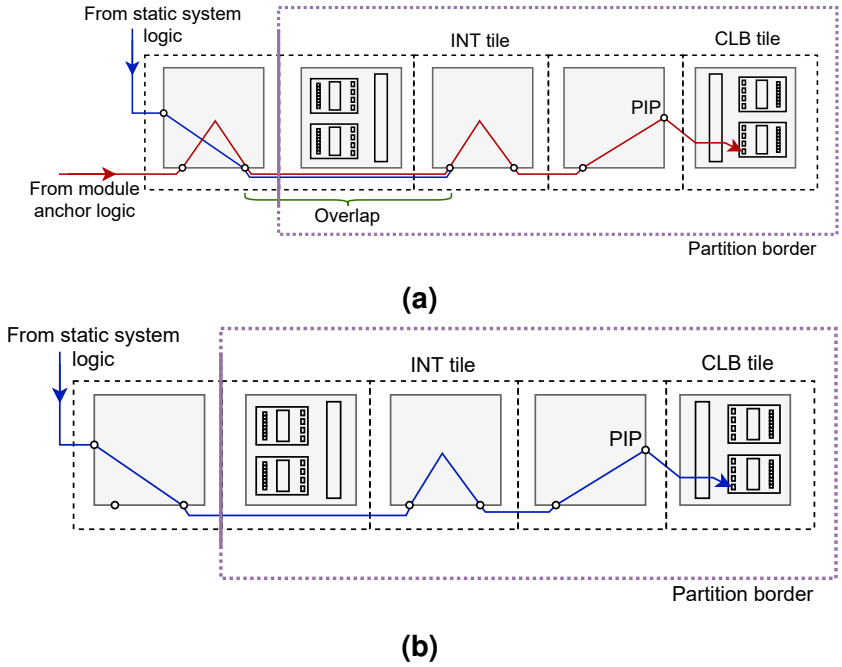


Figure 5.1.3: Example route that has a wire that overlaps in both nets.

To find the common border node, we have to inspect the routing of the interface nets. The route can be acquired by querying the `ROUTE` property using the `get_property ROUTE [get_nets <net name>]` command for the net specified. As we have seen in Section 2.1.1, the command returns a list of nodes that forms the route on the fabric. By default, the route string is in a relative form. The node names are not proceeded by their tile name. For routing a design, this is not a problem. As long as there is a defined begin and endpoint and there is only one way to go from one node to another. However, with this ambiguity, we can not find the common node border node in the two interface nets to be fused. In essence, we want to have the index of the node in the route string. By including the tile information in the route string we can find the index. There exist several methods to obtain the tile information and translate the route in an absolute route string:

- Using GoAhead [BKT12], we could extract this information from the included model of the fabric. This information must be exported during the generation of the constraints scripts. Either the complete absolute net or the index in both nets could be of use.
- Using RapidWright [LK18] which could work in the same way as GoAhead. The included model of the fabric is comparable to that of GoAhead tool, thereby it is able to retrieve an absolute route string. This requires processing in an external tool.
- Via `find_routing_path` command from Vivado, we can find a routing path between two or more nodes. This methods works if the tile information is present for begin and end node and only reliable for a single node hop. Multiple hops can result in a different route.
- Using the cardinal information of the nodes we can resolve the tile information. As we have seen in Section 2.1.1, we deduce the next tile from the node name. For example in Listing 5.1.5. From third node, INT_L_X20Y25/EE4BEG3, the displacement is four

into to the east direction (EE4). Adding 4 to the x-coordinate, we end up at node INT_L_X24Y25/EE4BEG3. We need at least one adjacent node that has the tile information to resolve the next node in the list.

- Via `get_nodes <net>` we get the list of nodes of the corresponding net. These nodes can be used in the route string. By default the list returned by the `get_nodes` command include tile information. However, the list is not in the order (the direction) of the nodes in the route string (when using `get_property ROUTE [get_nets net]`)[2]. Making this option not useful to resolve all nodes. It is only suitable for the unique nodes, commonly found at the begin and end of the route string.
- Via the `get_pips` command has been found to be the best method and is used currently. An example output of this command is shown in Listing 2.1.1. The PIP list has the same order as the route string. From the PIP name, we can extract the required tile information.

```
1 get_property ROUTE <net name>
2
3   { CLBLL_LL_D CLBLL_LOGIC_OUTS15 EE4BEG3 EE4BEG3 SE2BEG3 NR1BEG3 ... IMUX2 CLBLL_LL_A2 }
4
5 set_property ROUTE { CLBLL_L_X20Y25/CLBLL_LL_D CLBLL_L_X20Y25/CLBLL_LOGIC_OUTS15 INT_L_X20Y25/EE4BEG3
↪   INT_L_X24Y25/EE4BEG3 INT_L_X28Y25/SE2BEG3 INT_R_X29Y24/NR1BEG3 ... IMUX2 CLBLL_LL_A2} <net name>
```

Listing 5.1.5: Example assigning nodes to the ROUTE property of a net. Vivado permits intermixing nodes with and without tile information.

For each interface signal, we can now find the index of the common border node in both nets. There should be at least one common node, but it is likely to have multiple nodes if the net is fairly straight. In case we have multiple nodes, we choose the first node (in front of an open brace). When we have module-to-module nets, the last common node in the lowest nested branch is chosen.

The next step is to combine the route of two nets into one. Nets to travel from static system to the partial area are called *s2p* nets. The s2p nets have an output driver in the static system and or more inputs in the reconfigurable region. For those s2p nets, the tail of the static net and the head of the partial net needs to be removed. After that, the nets can be combined and applied to the driving net. In Vivado, it is not possible to create a net alias from TCL command. Therefore, we clear the route of the trailing net such that we do not get conflicts in overlapping nodes. The same is done for signals that are leaving the partial area. Nets that go from partial area to static system are called *p2s* nets. In case the grid-style architecture is used, we can also define module-to-module nets *m2m*.

For modules that do not make use of any logic, but only use the routing resources, a number of additional steps have to be carried out. These wire-only modules arise if, for example, in the HDL the data is shifted. Certain shift operations such as division or the shift rows operation from the AES algorithm as we see later in Section 6.2, only requires wires on the fabric. The routing resources of those nets need to be added to the source module first. After that, the final net combination can be made with the destination net.

[2]Observed with Vivado 2018.3, but this could be different in later versions.

5.1.6 Restore Clock Logic

Clock signals are required for the correct operation of the modules in the partial region. To ensure that the clock signal is available for each module, it must be connected and distributed on the reconfigurable region. Likewise, for the interface signals, these clock resources and networks must be allocated and properly terminated in the reconfigurable region. GoAhead provides the `ConnectClockPins` command (Listing 5.1.6). The command is executed on all tiles that are selected in the block view of GoAhead. Logic elements on the fabric that require a clock signal (i.e. have a clock input pin) are instantiated by the generated TCL statements. For the CLBs, the output registers are instantiated as flipflops. Each clock pin of the register is connected to the global clock net. During the implementation phase, this script is executed. All registers that are present in the slices of the CLBs are instantiated as FDRE cells. The clock input (CK) of each FDRE flipflop is connected to the clock tree that spans two columns of CLBs. In the middle of the column, the clock net is connected to the global clock network. Figure 5.1.4 shows how restoration looks on the fabric level.

```
ConnectClockPins
  ClockPin=CK
  BELs=[A-D]FF
  ClockNetName=clk_IBUF_BUFG
  FileName=connect-clockpins.tcl
  Append=False
  CreateBackupFile=False;
```

Listing 5.1.6: GoAhead command to connect the clock pins

```
create_cell -reference FDRE  SLICE_X51Y99_DFF
place_cell SLICE_X51Y99_DFF SLICE_X51Y99/DFF
create_pin -direction IN SLICE_X51Y99_DFF/C
connect_net -hier -net clk_IBUF_BUFG -objects {SLICE_X51Y99_DFF/C}
create_cell -reference FDRE  SLICE_X51Y99_CFF
place_cell SLICE_X51Y99_CFF SLICE_X51Y99/CFF
create_pin -direction IN SLICE_X51Y99_CFF/C
connect_net -hier -net clk_IBUF_BUFG -objects {SLICE_X51Y99_CFF/C}
create_cell -reference FDRE  SLICE_X51Y99_BFF
place_cell SLICE_X51Y99_BFF SLICE_X51Y99/BFF
create_pin -direction IN SLICE_X51Y99_BFF/C
connect_net -hier -net clk_IBUF_BUFG -objects {SLICE_X51Y99_BFF/C}
create_cell -reference FDRE  SLICE_X51Y99_AFF
place_cell SLICE_X51Y99_AFF SLICE_X51Y99/AFF
create_pin -direction IN SLICE_X51Y99_AFF/C
connect_net -hier -net clk_IBUF_BUFG -objects {SLICE_X51Y99_AFF/C}
```

Listing 5.1.7: Example script to connect the clock pins in the partial area of the static design. Here all flipflops of SLICE_X51Y99 are instantiated and connected to the global clock net.

The placed clock logic and routing from the GoAhead script causes a physical overlap of BELs on the fabric with that of the module logic. The `update_design -cell <cell name> -black_box` command removes all cells in the cell specified. It does however not remove the clock net generated from the TCL command from GoAhead, because it does not belong to that hierarchy. The parent property is not set of those

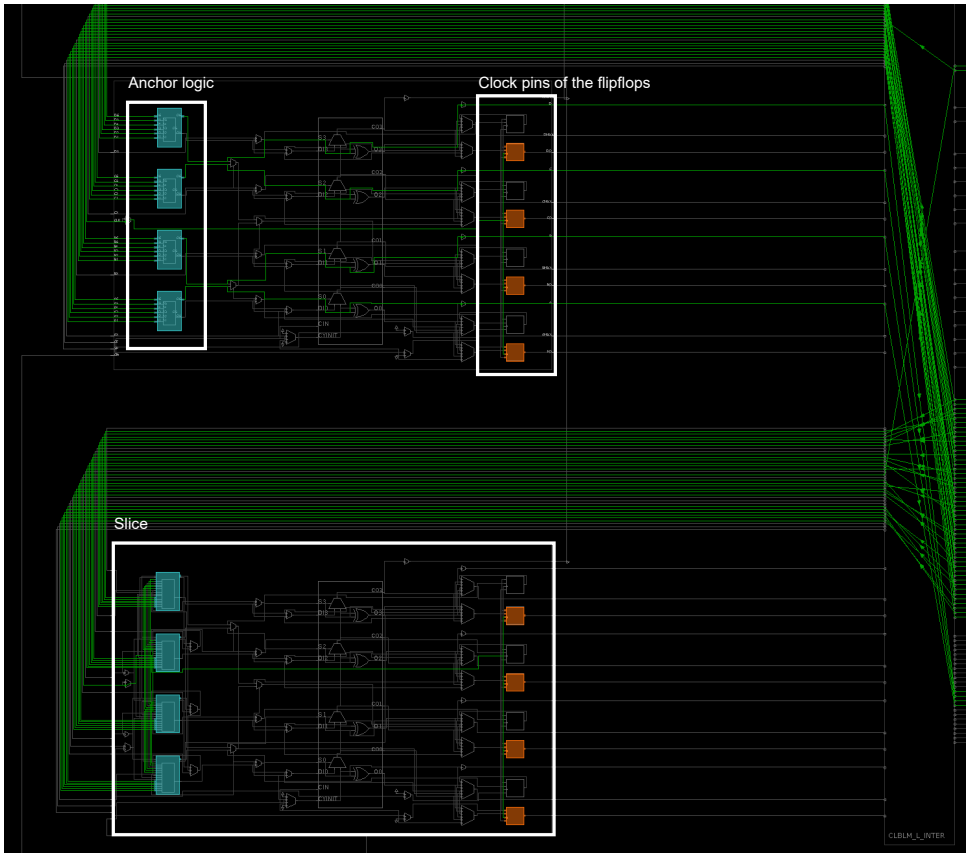


Figure 5.1.4: The device view of the reconnected clock pins of each flipflop in each slice and the restored anchor logic.

cells and can not be altered the current state of the design. To restore the clock nets and logic after the placement of a module, the unplaced FDRE cells need to be removed first from the design. These unplaced cells can be found by querying all cells that have the property `STATUS==UNPLACED`. From the mapping script, the user must specify the name of the module clock nets. The clock nets and pins are then merged with the clock net of the static design.

5.1.7 Restore Anchor Logic

For certain configurations of modules, not all communication interfaces are required. The reconfigurable region is cleared before any logic is placed. This cuts all communication signals (nets) at the border of the partition. Those nets have floating nodes and are marked as `ANTENNAS`. Antennas are nets that contain nodes not properly terminated with a sink pin [BBS17]. Floating nodes can also occur in a branch of the specified net. By running a DRC check or running the `report_route_status` command, Vivado will list all the nets that have a problem including antennas. Vivado is able to remove such antenna nets, but this removal leads to unwanted optimizations in the route. Therefore we restore as much as possible of the original anchor logic to prevent any floating nets. Depending on the design, if the interface is not used, we could just clear the entire route to the partition.

5.1.8 Finalize the Design

The intermediate signals used in the HDL source for the interface signals need to be removed since they are not required anymore. Furthermore, if a module turns into a blackbox, it needs to be removed as well. After that, a final `route_design` command is issued. If no errors occurred at this point, we know that the provided module configuration is valid. We should be able to write a DCP file and a final bitstream file. On this checkpoint, we can perform a timing analysis and run a functional simulation. This is described in Section 5.2.

5.2 Timing Analysis and Simulation

After a successful reconstruction, we have obtained the DCP file on which we can perform timing analysis. By running the `report_timing_summary`, Vivado reports if there are any timing violations in the design. A clock constraint is required to perform any timing analysis on paths in the design. To create clock constraint of 100 MHz, we issue the following command:

```
create_clock -name clk -period 10 -waveform {2.5 5} [get_ports clk]
```

Furthermore, a script is provided with the DRT to invoke the simulator. This script writes a *Standard Delay Format* (SDF) file, which contains all the timing-related information for each cell. After that, the top-level unit is translated into executable code and linked to the simulation kernel to create an executable simulation snapshot. Next, the simulator loads a simulation snapshot to affect the interactive simulation environment.

CHAPTER 6

EXAMPLES

This chapter provides a few example implementations for design reconstruction. The first example is a minimal working example. The last example is a case study that uses the grid-style partial reconfiguration architecture.

6.1 Example 1: Minimal Working Example

6.1.1 Implementation and results

Division is one of the four basic arithmetic operations that is the hardest to implement in hardware. Mainly because addition, subtraction and multiplication are well defined and give exact answers. The result of a division between integers will generally be a rational number, which in many cases can not be represented in binary. This example implements 16-bit signed division in VHDL using the division operator (/). Information on which division algorithm is implemented when using the division operator is not published by Xilinx. Furthermore, depending on the operand size and the chip used, different (more optimal) algorithms could be chosen for the implementation. Listing A.0.6 shows the HDL source code for this example.

If at any point in time an error is detected in the configuration (e.g. the specified interface nets) or a common border is not found or a (potential) overlap is detected, the reconstruction stops, and the error is reported to the user. After a successful run, the design reconstruction, we can visually inspect the result in the device view. The nets outside the partial region appear to be identical (Figure 6.1.2). We can also observe that the correct border nodes (wires) are chosen by the tool, highlighted in the device view in Figure 6.1.1. Looking closer at the logic inside the partial area (purple rectangle, PBlock), additional clock logic can be seen. After removing the clock logic cells and routing, the PBs shows up as binary identical (Figure 6.1.5).

After we have obtained a fully placed and routed design, we can check the design for timing-related issues. The maximum clock frequency for the design can be calculated as follows:

$$f_{max} = \frac{1}{|t_{ACP} - t_{WNS}|} \tag{6.1.1}$$

Where we define:

t_{ACP} = The Actual Clock Period.

t_{WNS} = The Worst Negative Slack.

The WNS can be retrieved by running the command *report_timing_summary* from Vivado. The ACP is set in the XDC file of the design. For Example 1, using a 100

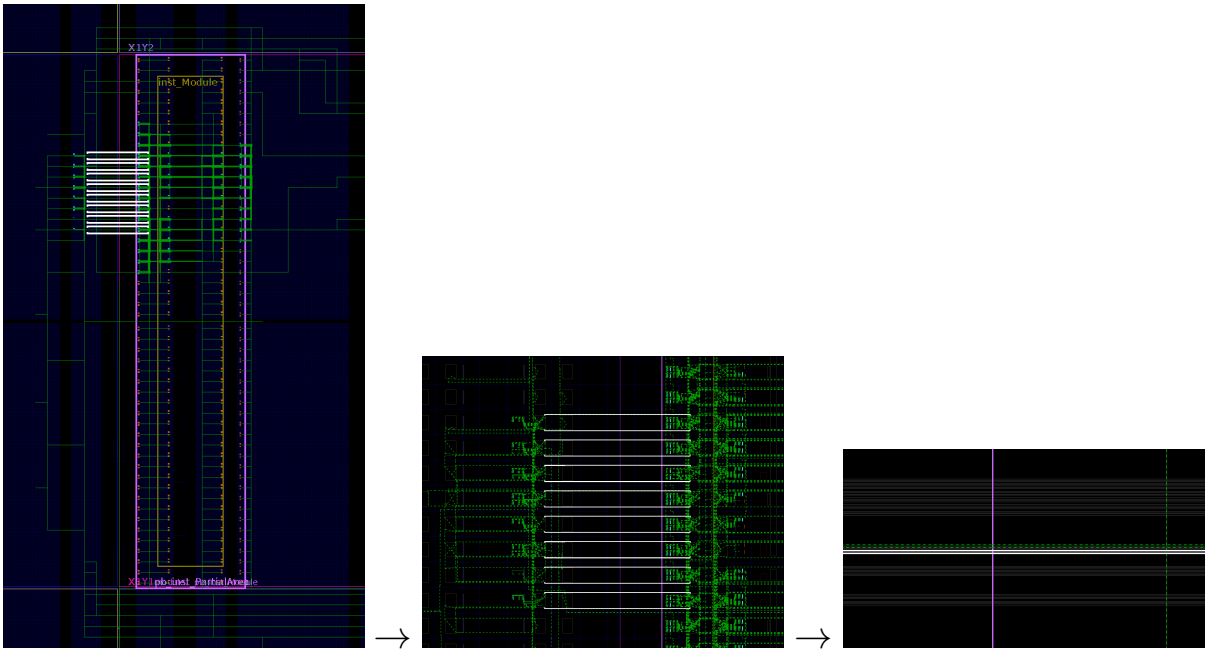


Figure 6.1.1: Border nodes used for interfacing, from left to right zoomed in on the border.

MHz clock, the slack is negative. This means the design can not run on the frequency specified. We get that the maximum operation frequency is[1]:

$$f_{max} = \frac{1}{|10ns - (-44.589ns)|} \approx 18.3MHz \tag{6.1.2}$$

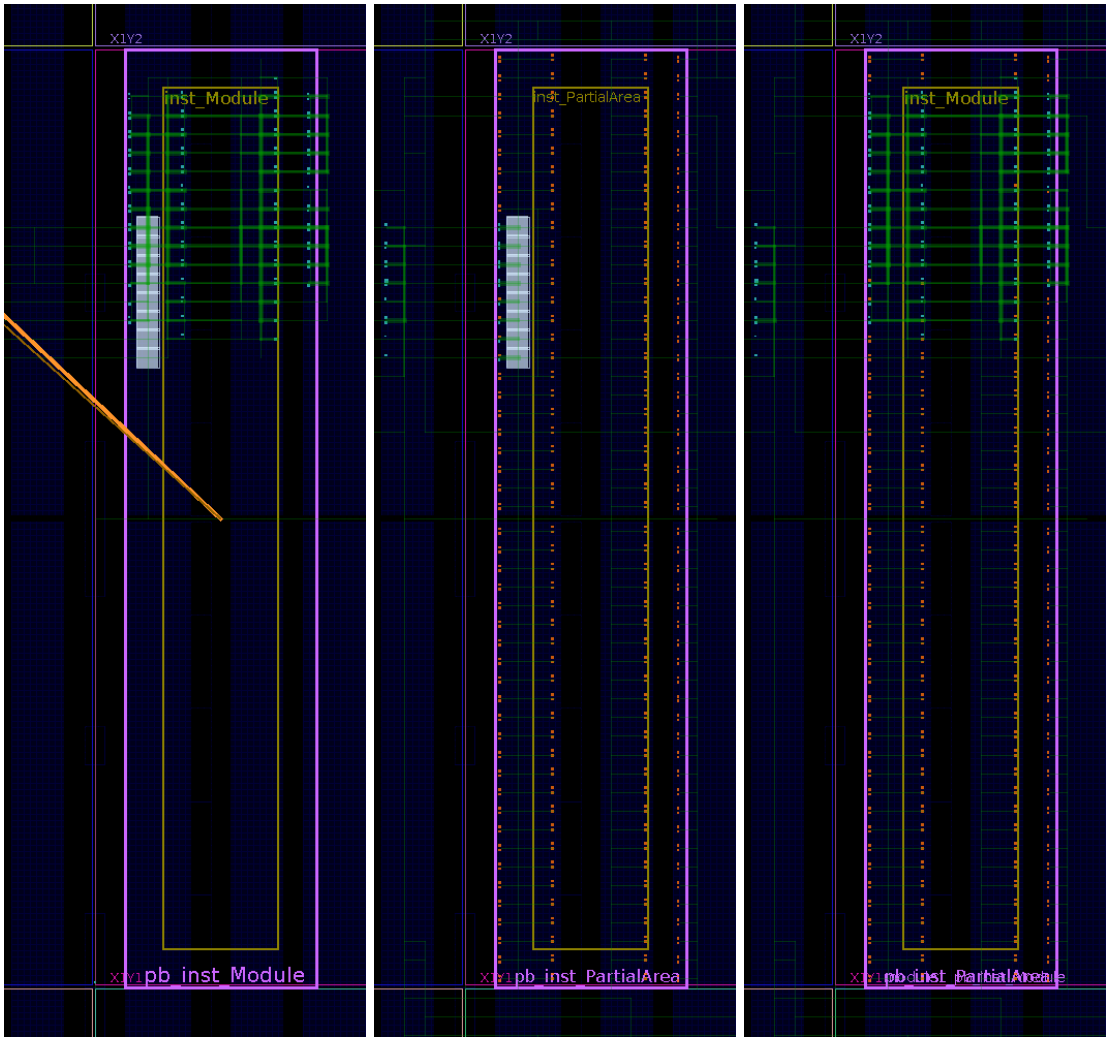
Since we run in a non-project mode, we have to do the timing analysis in a manual fashion by entering commands in the TCL console of Vivado. Listing 6.1.1 shows how to get the timing information of the design by means of a TCL script. The listing shows how to add the clock constraint (if not already present) with the `create_clock` command and how to query the timing paths. This gives a quick pass/fail test if the design can operate on the given clock frequency. The output of the timing analysis for 100 MHz is shown in Listing 6.1.2, where it shows that the timing constraints are not met.

Moreover, we can simulate the design with a graphical simulator for functional testing. Vivado has supports for multiple simulators. For this work, we use the integrated Vivado Simulator.

The static system contains a state machine that applies the values to the inputs *a* and *b* of the module. At each clock edge, input *a* of the module remains at the maximum value of 32767. Input *b* toggles between arbitrary values 3, 333 and 1. Figure 6.1.4 shows the simulation waveform of example 1. In the begin of the waveform, we have a input clock of 10 MHz and at the end, the clock speed is 100 MHz. It clearly shows that on 100 MHz the output value is wrong.

To verify that the design reconstruction is correct, we can compare the bitstream files of the static system and module. A placed module must have an identical partial

[1] We do not include any input or output delays here, only the paths involving the clock.



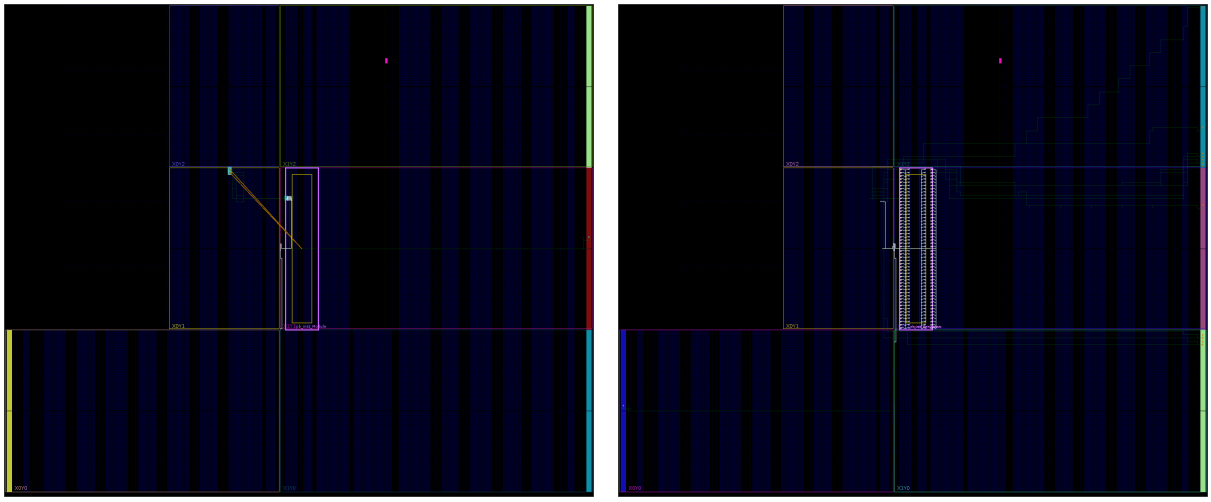
(a) **(b)** **(c)**

Figure 6.1.2: Visual inspection of the reconfigurable partition in the device view. (**a**) The module to be placed. (**b**) The empty slot of the static system. (**c**) The final implemented design.

bitstream as the partial bitstream of a module. They must be binary equivalent. The BITMAN tool is used to extract a partial bitstream from the (full) bitstream of the static system. After that, the extracted bitstream is compared with the partial bitstream of the module. If the bitstream files are identical, we are sure the reconstructed module is equal to the module in run-time.

Another thing we can do is place the PB into the static bitstream and compare the static system on equality. According to Dang Pham et al., the generated full bitstream from Vivado is different from BITMAN. The contents and FAR value should be the same, but there are some differences in the commands. Furthermore, the timestamp and Vivado version is different. Printing the CLB configuration with BITMAN with the *-c* switch does not show any differences. Unfortunately, we can not compare the static system with this method[2]. The method works for the partial bitstream since we slice out the same region from the full bitstream and executing the commands consecutive

[2]Another method would be to read back the configuration from the FPGA, but this probably leads to the same result



(a) Module 1 **(b)** Combined

Figure 6.1.3: Clock nets

```
# Create a clock object
create_clock -name clk -period 10 -waveform {2.5 5} [get_ports clk]

# Get the period of the clock (ACP, in ns)
set period [get_property PERIOD [get_clocks clk]]
# 10.000

# Get the timing slack (WNS)
set slack [get_property SLACK [get_timing_paths]]
# -44.589

# The max. clock frequency (in Hz):
set max_clock [expr { 1 / ($period - $slack) * 1000000000}]
# 18318708.897396915

report_timing_summary
```

Listing 6.1.1: Script to get the timing related information of the design. The output is shown below the command.

```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints
    -------      -------  ---------------------  -------------------      -------      -------  ---------------------
    -44.589     -708.992                     16                   27        0.112        0.000                      0


THS Failing Endpoints  THS Total Endpoints     WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
---------------------  -------------------     --------     --------  ----------------------  --------------------
                    0                   27        2.000        0.000                       0                 1252

Timing constraints are not met.
```

Listing 6.1.2: The output of timing summary report of example 1 for a 100 MHz clock (partially displayed).

gives the same timestamp.

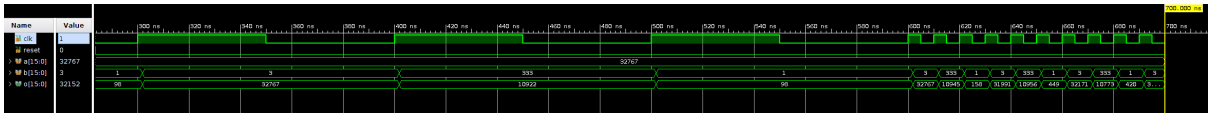


Figure 6.1.4: The waveform from the Vivado simulator.

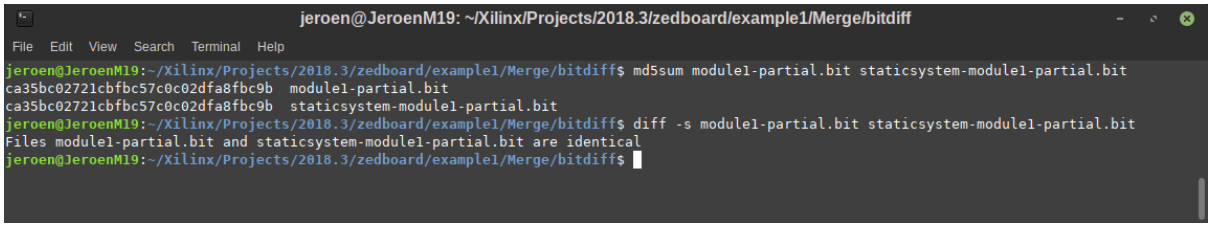


Figure 6.1.5: Bitstream file comparison using the `md5sum` and `diff` command, they show up as identical.

6.2 Example 2: Case Study AES encryption

For this example, the *Advanced Encryption Standard* (AES) encryption algorithm is implemented on an FPGA system using the grid-style architecture. The goal is to prevent side-channel attacks and can be achieved by shuffling modules during runtime. This results in different power profiles, thereby making side-channel attacks more difficult. However, the operating frequency of the design could not be determined, it had to be guessed by lowering the clock speed.

6.2.1 Background on AES encryption

The AES is a fast and secure block cipher widely used for the encryption of data. It is a symmetric encryption algorithm where the same key is used to encrypt en decrypt the data. AES is known for its speed and security. Speed from the fact that it requires less computational power compared to asymmetric encryption algorithm. Secure, because of a sophistical block cipher algorithm where the data is encrypted on a block basis. AES uses a fixed block size of 128 bits and key size of 128, 192 and 256 bits. It operates on a 4x4 column-major order matrix of bytes. The key size determines the number of rounds required to move the data to be encrypted through the cipher algorithm. A 128 bits key requires 10 rounds, 192 bits requires 12 rounds and 14 rounds when 256-bit key is being used.

The AES algorithm shown in Figure 6.2.1 is an iterative algorithm. Firstly, a bitwise XOR operation is performed between the encryption key and the plain text input data. This operation gives the initial round key (note that RoundKey(0) is the starting key). After that, a repeated sequence of four operation steps are performed (Figure 6.2.2):

- **SubBytes**
  This byte substitution operation replaces each of the 16 bytes in the state matrix (input) with a fixed byte from a lookup table called the S-box. The S-box effectively maps an 8-bit input, to an 8-bit output. This ensures that the cipher has a non-linear component.

- **ShiftRows**
  The ShiftRows shifts the bytes in the static matrix by a certain offset. The first

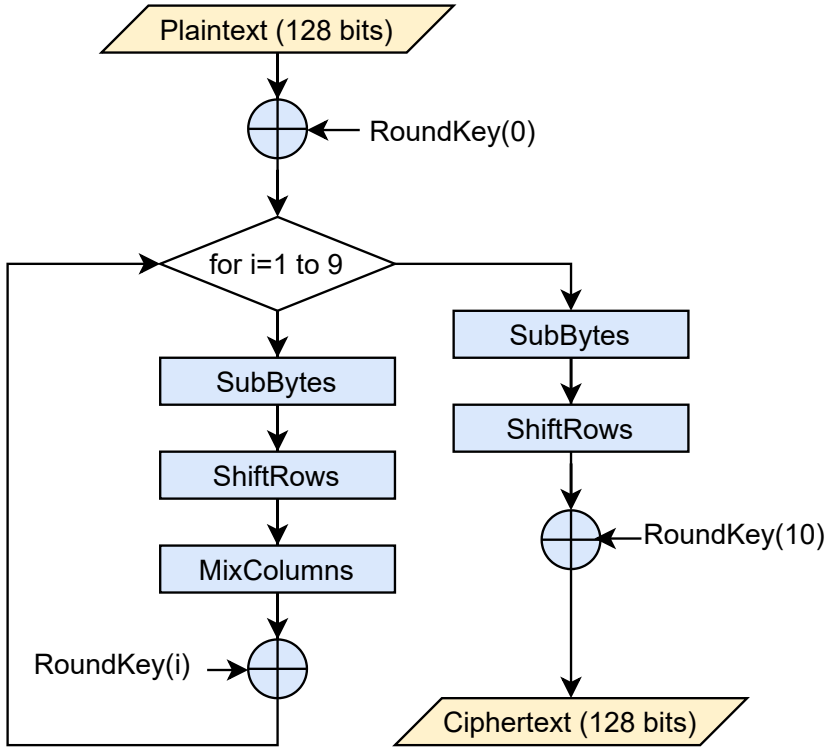
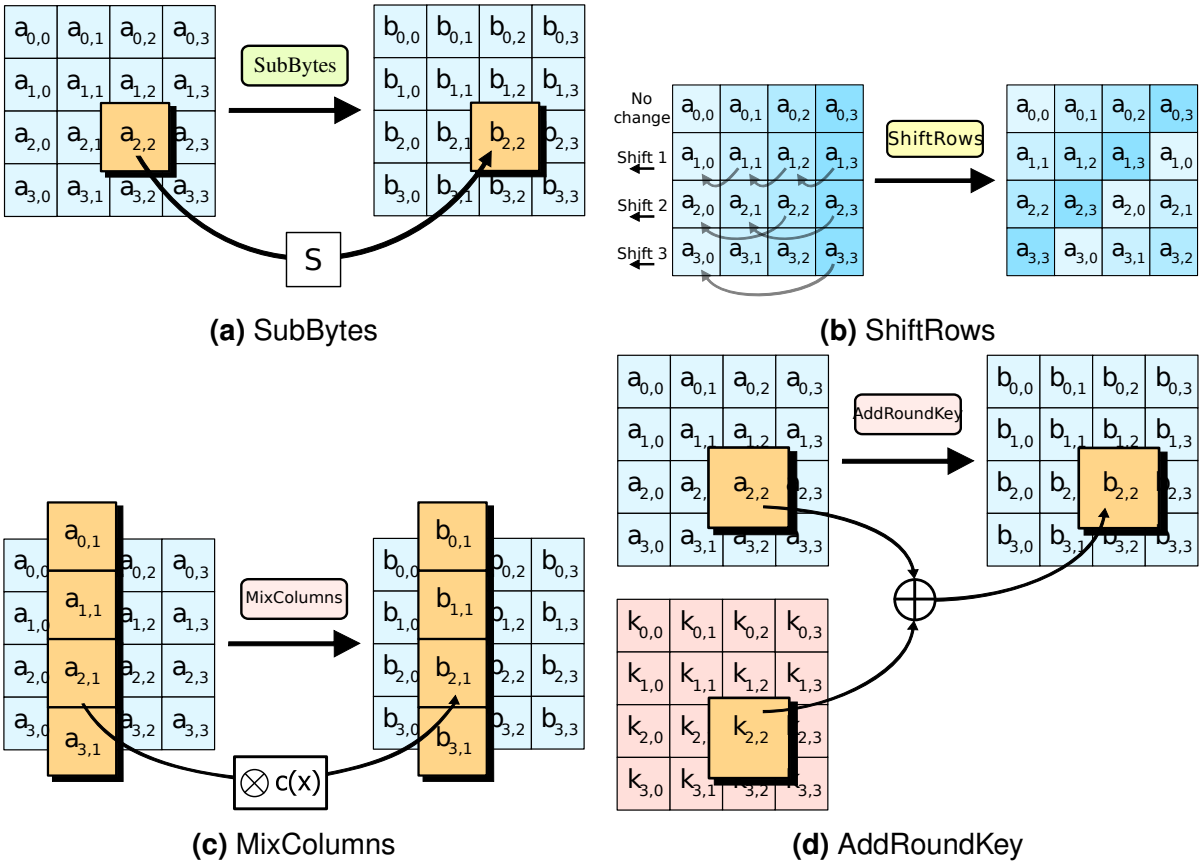


Figure 6.2.1: The AES encryption algorithm when a 128-bit key is used.



(a) SubBytes **(b)** ShiftRows

(c) MixColumns **(d)** AddRoundKey

Figure 6.2.2: AES encryption operation steps for each round (figures from [Wik]).

row is not changed. For the second row, each byte is shifted one to the left. For the third row, each byte is shifted by two to the left and three to the left for the fourth row.

- **MixColumns**
  In the Mix Columns step, each column of the state matrix is multiplied by a fixed polynomial represented by the matrix in Equation (6.2.1).

- **AddRoundKey**
  In the final step for each round, the current state matrix is XORed with the Round-Key and is the input of the next iteration. It is identical to the very first operation.

$$p = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \tag{6.2.1}$$

For a 128-bit key, the above steps are repeated 9 times. The last round does not have the MixColumns operation. After finishing all rounds, the 128-bit input data is encrypted. For decryption, the sequence from Figure 6.2.1 is inverted.

Searching for weaknesses in AES algorithm, there exist some types of attacks aimed at the physical implementation. These types of attacks are, for example the timing and power side-channel attacks. Koeune et al. describes a timing attack method. Certain logic operations take more time to execute and differs on the input. Measuring the time for each operation, an attacker can recover the key by working backwards. Another attack would be looking at the power consumption of AES [LBC16] and see Figure 6.2.3. Differential power analysis is a common power analysis attack that is very effective against implementations of block ciphers like AES. The basic idea is to analyze the power consumption of the device for different cipher texts during encryption or decryption.

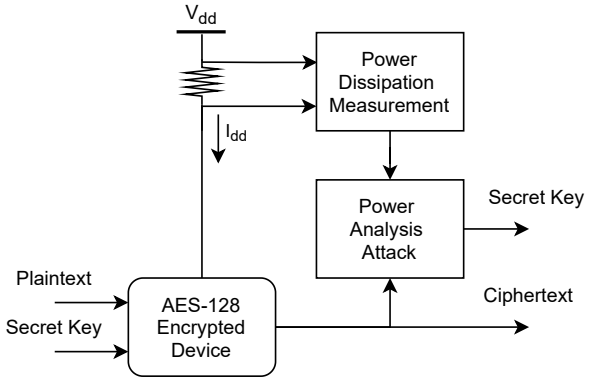


Figure 6.2.3: Power analysis based side-channel attack on AES-128.

There exist some countermeasures to overcome some physical attacks on the AES implementation. One method could be to switch the key frequently enough such that the attacker can not learn enough about the key in time. Another way would be to make the power consumption independent from the processed input values. This can be achieved by making the power consumption more random (or equal) for each clock

cycle. An example for achieving power side-channel protection is by using partial reconfiguration [Sas+15; Het+19].

Leakages can often be found on power traces during transitions of the combinatorial circuit after a change of a driving register. A common target on side-channel attacks on symmetric block ciphers is the output of the non-linear substitution layer of the algorithm.

In [Sas+15] the authors use the idea of random S-Box decomposition realized by configurable lookup tables. The standard S-Box is split into two random mappings and the correct S-box output is never stored onto a register. Using DPR, the dynamic decomposition was applied for the S-Boxes, including pre-charging of registers[3]. After running the power traces, they were not able to detect any first-order[4] side-channel leakage.

The following subsection describes another countermeasure to prevent power side-channel attacks.

6.2.2 Implementation and Results

The starting point for this example is the implementation and continuation of the case study from “Framework for Fine-Grained Partial Reconfiguration on FPGAs” [Hog19]. The idea is to use random module configurations and module implementation variants, resulting into a more random power profile to make side-channel attacks more difficult. To have multiple modules in the same partial region, the grid-style reconfiguration architecture was used. Along with the use of DPR, the physical implementation of the encryption algorithm can be altered during runtime. The design is implemented on the ZedBoard, a development board for the Xilinx Zynq 7000 SoC. Three operations of the AES encryption algorithm were relocated to the partial region: the SubBytes, ShiftRows and the MixColumns. The static system controls the sequence and the number of rounds of the algorithm. As mentioned before, timing analysis could not be performed on grid-style reconfiguration systems. Using the proposed reconstruction tool, this can be done now. Two configurations are tested. Each configuration implements a variant of the SubBytes, ShiftRows and MixColumns operation, see Figure 6.2.4.

The static system acts as the testbench for the AES implementation. A TCL script is executed during simulation. This script sets up the wave from signals and provides the stimulus to the inputs of the top entity. Those inputs are mapped to real inputs of the FPGA chip. This simulates the switches and buttons on the Zedboard. For the AES algorithm, the last 4 bits of input data is applied via switches present on the ZedBoard. The last 4 bits of the AES are output on four LEDs of the ZedBoard. These outputs can now also be observed in the simulator (Figure 6.2.5b and Figure 6.2.6b). Figure 6.2.5a and Figure 6.2.6a shows that the timing constraints are met for this design. Two input values are used for testing the designs[5]. For input value 6 we get:

- input: 3243F6A8885A308D313198A2E0370736
- key: 2B7E151628AED2A6ABF7158809CF4F3C
- output: 481A0AAC6942EA7F426d3A5BD4B25fAD

[3]Consecutive round inputs stored into registers can cause leakage. By expanding the single register into two registers the leakage is avoided.

[4]When only one probe is used on the design.

[5]The output was checked using http://aes.online-domain-tools.com

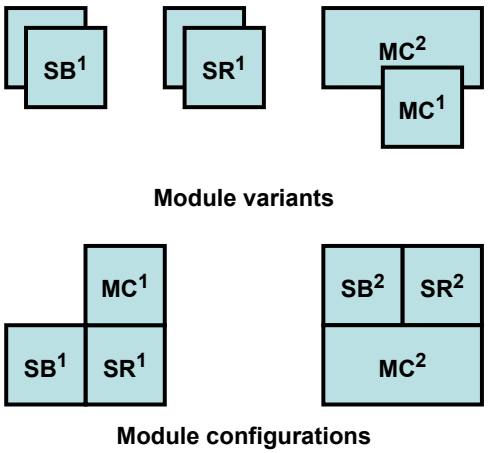


Figure 6.2.4: Different AES configurations and module variants

And for input value 1 we get:

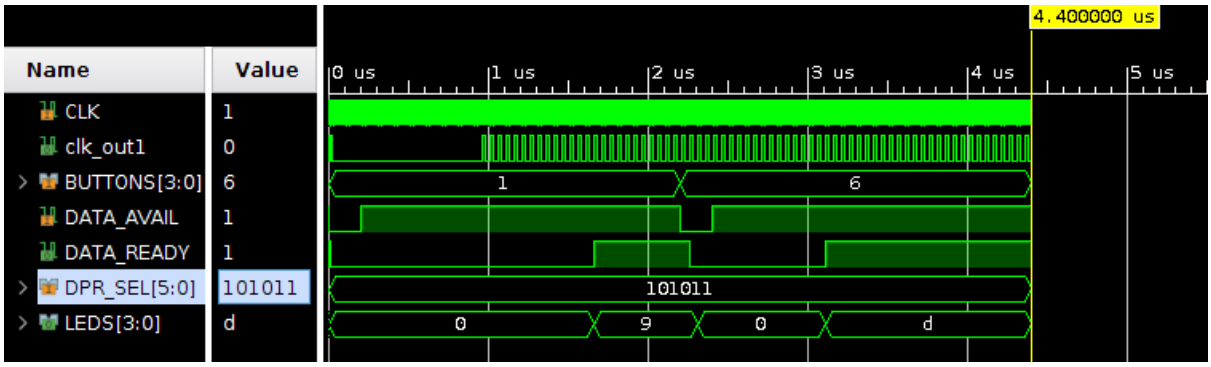
- input: 3243F6A8885A308D313198A2E0370731
- key: 2B7E151628AED2A6ABF7158809CF4F3C
- output: F91914CD01924B124C2EC316B4B35A79

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	26.468 ns	Worst Hold Slack (WHS):	0.150 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	666	Total Number of Endpoints:	666	Total Number of Endpoints:	344

All user specified timing constraints are met.

(a)



(b)

Figure 6.2.5: Test results of AES configuration 1, (**a**) shows the timing report and (**b**) the waveform from the Vivado simulator.

Both waveforms show the correct output on the LEDs. For the maximum clock frequency we get for configuration 1:

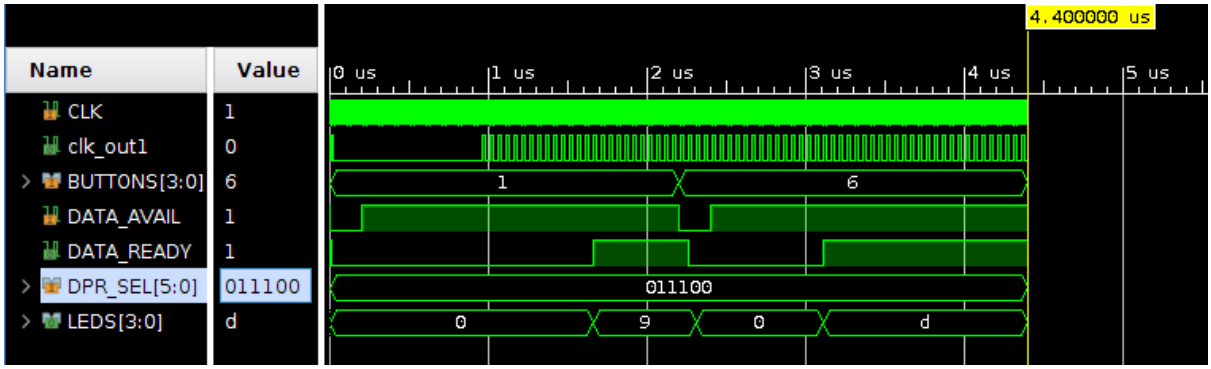
$$f_{max} = \frac{1}{|10ns - (26.468ns)|} \approx 31MHz \tag{6.2.2}$$

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	21.671 ns	Worst Hold Slack (WHS):	0.150 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	666	Total Number of Endpoints:	666	Total Number of Endpoints:	344

All user specified timing constraints are met.

(a)



(b)

Figure 6.2.6: Test results of AES configuration 1, (**a**) shows the timing report and (**b**) the waveform from the Vivado simulator.

And for configuration 2 we get:

$$f_{max} = \frac{1}{|10ns - (21.671ns)|} \approx 27MHz \tag{6.2.3}$$

CHAPTER 7

CONCLUSION AND RECOMMENDATION

7.1 Conclusions

We have seen in Chapter 6 that the tool can merge and stitch modules back into the reconfigurable partition of the static system. With the resulting fully implemented design, we are able to verify a run-time PR configuration for grid-style reconfiguration styles. Timing analysis can be performed and the static system can be functionally simulated. Furthermore, during reconstruction, the interface configuration between static design and modules can be checked for consistency and correct module placement on the fabric can be monitored. The design can be tested without having the hardware. The whole reconstruction process is automated, including the simulation.

7.2 Recommendations

To mention some of the recommendations and future work, one of the limitations is that we can not *move logic around*, i.e. module relocation on the netlist level. The module must be built with the correct footprint and location (same coordinates) on the fabric since module relocation on the netlist level is not implemented in the tool. The tool could be extended to do this relocation on the netlist level or use an academic tool capable of performing this operation.

We are unable to restore all interfaces when they are unused since we can not leave nets floating as antennas. As long as the nets are not clock paths, it will not affect the timing of the design. Otherwise, another method must be found to allow these antennas in the design, e.g. lowering the DRC from Vivado.

For simplicity, a TCL script is used to set up the simulation and provide the input values. However, by doing this we have to verify the output from the simulator manually. A testbench written in HDL code could be used for this.

The reconstruction tool is designed to work with the design checkpoints generated with the GoAhead tool. Therefore it relies on the interface names used in the designs, but the tool could be extended to work with other frameworks.

Moreover, GoAhead could be extended with a system configuration specification file for grid-style reconfigurable systems. A number of commands to set up a reconfigurable system with GoAhead have to be repeated for each slot. Some of them have no relation or restrictions to other commands, therefore missing certain validations in the design step. An example validation could be checking if a specified region (a selection of tiles) on the FPGA is valid for a certain configuration. Valid in the sense that the region of the correct size, position and that it does not overlap with other regions

on the FPGA fabric. Examples for the regions to be defined are the partial area, the interfaces in and outside the partial area, the size of the blocker fence, the size and position of tunnels. To make the setup of a (grid-style) reconfigurable system more simple and effortless, we could combine some of the commands in a single command. Combining commands makes the configuration less error-prone.

Furthermore, the reconfiguration time is directly proportional to the size of the bitstream. The smaller the bitstream, the lesser time it requires to reconfigure it. Therefore, regarding the fine-grained reconfiguration style, one could find out what the real impact is on bitstream storage and reconfiguration time compared to island-style.

APPENDIX A

```
reconfigurable_partition_group
  partition_group_name = group1
  interface = ${local}/interface
  reconfigurable_partition
    partition_name = reconf_part_0
    pblock = X40Y9:X45Y40
  end_reconfigurable_partition
end_reconfigurable_partition_group
```

Listing A.0.1: Example file *virtual_architecture.*. For each partition, the size, location and interface is declared here.

```
local_nets
  input1 NORTH
  input2 NORTH
  output1 SOUTH
end_local_nets

global_nets
  clk 0
  reset 1
end_global_nets
```

Listing A.0.2: Example file *interface*. The inputs to the module are only permitted to used the North border and outputs only the South border.

```
project_variables
  project_name = IMPRESS_build
  directory = /example1/impressproject
  fpga_chip = xc7z020clg400-1
end_project_variables

static_system
  sources = /example1/output/static_system/build/synth_checkpoint
  virtual_architecture = /example1/impressproject/virtual_architecture
end_static_system

reconfigurable_modules
  reconfigurable_module
    module_name = add
    sources = /example1/hdl_src/modules/add.vhd
    partition_group_name = group1
  end_reconfigurable_module
  reconfigurable_module
    module_name = multiply
    sources = /example1/hdl_src/modules/multiply.vhd
    partition_group_name = group1
  end_reconfigurable_module
  reconfigurable_module
    module_name = substract
    sources = /example1/hdl_src/modules/substract.vhd
    partition_group_name = group1
  end_reconfigurable_module
  reconfigurable_module
    module_name = div
    sources = /example1/hdl_src/modules/div.vhd
    partition_group_name = group1
  end_reconfigurable_module
  virtual_architecture = /example1/impressproject/virtual_architecture
end_reconfigurable_modules
```

Listing A.0.3: Basic IMPRESS project configuration example. The *project_info* file defines the system specification. Four reconfigurable modules are declared. Modules are grouped by the same partition, the partial area.

```vhdl
library UNISIM;
use UNISIM.vcomponents.all;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity top is
  port (clk       : in std_logic;
        in0       : in std_logic;
        out0      : out std_logic
        );
end top;

architecture Behavioral of top is
  signal i_out: std_logic;
begin

-- source: ug953
inst_FDRE_1 : FDRE
generic map(
  INIT => '0') -- Initial value of register ('0'or'1')
port map(
  Q => i_out,   -- Data output
  C => clk,     -- Clock input
  CE => '1',    -- Clock enable input
  R => '0',     -- Synchronous reset input
  D => in0      -- Data input
);

inst_FDRE_2 : FDRE
generic map(
  INIT => '0') -- Initial value of register ('0'or'1')
port map(
  Q => out0,    -- Data output
  C => clk,     -- Clock input
  CE => '1',    -- Clock enable input
  R => '0',     -- Synchronous reset input
  D => i_out    -- Data input
);

end Behavioral;

-- xdc:
-- set_property PACKAGE_PIN Y9 [get_ports {clk}];
-- set_property PACKAGE_PIN K19 [get_ports {out0}];
-- set_property PACKAGE_PIN M22 [get_ports {in0}];
```

Listing A.0.4: Example design for netlist.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top is
  port (
    outp   : out std_logic_vector (7 downto 0);
    inp    : in std_logic_vector (7 downto 0);
    clk    : in std_logic;
    reset  : in std_logic;
    btn    : in std_logic
  );
end top;

architecture structural of top is

component PartialArea is
  port (
    x0y0_s2p_w : in std_logic_vector(31 downto 0);
    x0y0_p2s_w : out std_logic_vector(15 downto 0)
  );
end component PartialArea;

attribute DONT_TOUCH : string;
attribute DONT_TOUCH of inst_PartialArea: label is "TRUE";

-- state machine
type state_type is (s0, s1, s2);
signal state : state_type;

-- intermediate signals
signal a : signed(15 downto 0);
signal b : signed(15 downto 0);

signal x0y0_s2p_w : std_logic_vector(31 downto 0);
signal x0y0_p2s_w : std_logic_vector(15 downto 0);

-- attribute_declaration
attribute s : string;
attribute keep : string;

-- attribute_assignment
attribute s of x0y0_s2p_w : signal is "true";
attribute s of x0y0_p2s_w : signal is "true";
attribute keep of x0y0_s2p_w : signal is "true";
attribute keep of x0y0_p2s_w : signal is "true";
begin

  inst_PartialArea : PartialArea
  port map (
    x0y0_s2p_w => x0y0_s2p_w,
    x0y0_p2s_w => x0y0_p2s_w
  );

  x0y0_s2p_w <= std_logic_vector(a) & std_logic_vector(b);


  -- output select
  process(clk, reset, btn)
  begin
    if rising_edge(clk) then
      if btn = '1' then
        outp <= x0y0_p2s_w(15 downto 8);
      else
        outp <= x0y0_p2s_w(7 downto 0);
      end if;
    end if;
  end process;


  -- state machine
  process(clk, reset)
  begin
    if reset = '1' then
      state <= s0;
    elsif rising_edge(clk) then
      case state is
        when s0 => state <= s1;
        when s1 => state <= s2;
        when s2 => state <= s0;
        when others => state <= s0;
      end case;
    end if;
  end process;


  -- provide input data
  process(state, inp)
  begin
    case state is
      when s0 =>
        a <= to_signed(32767, a'length);
        b <= to_signed(3, b'length);
      when s1 =>
        a <= to_signed(32767, a'length);
        b <= to_signed(333, b'length);
      when s2 =>
```

```
98          a <= to_signed(32767, a'length);
99          b <= resize(signed(inp), 16);
100       when others =>
101         a <= to_signed(1, a'length);
102         b <= to_signed(1, b'length);
103     end case;
104   end process;
105
106
107 end structural;
```

Listing A.0.5: Implementation of the static system of example 1.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity div is
6    port (
7      clk : in std_logic;
8      a   : in std_logic_vector(15 downto 0);
9      b   : in std_logic_vector(15 downto 0);
10     o   : out std_logic_vector(15 downto 0)
11   );
12 end div;
13
14 architecture behavioural of div is
15 begin
16
17   process(clk, a, b)
18   begin
19     if rising_edge(clk) then
20       o <= std_logic_vector(signed(a) / signed(b));
21     end if;
22   end process;
23
24 end behavioural;
```

Listing A.0.6: Implementation of the reconfigurable module of example 1.

APPENDIX B

```
# Read sources
read_vhdl [glob sources/*]
read_vhdl [glob goahead/sources/*]

# Synthesize design
synth_design -part xc7z020clg484-1 -top top -keep_equivalent_registers -flatten_hierarchy none
#synth_design -part xc7z020clg484-1 -top top
write_checkpoint -force checkpoints/synthesis.dcp

# Optimize synthesized design
opt_design -sweep
write_checkpoint -force checkpoints/optimize-design.dcp

# Placement constraints
source goahead/tcl/module-placement-constraints.tcl
write_checkpoint -force checkpoints/placement-constraints.dcp

# Interface constraints
source goahead/tcl/module-interface-constraints.tcl
write_checkpoint -force checkpoints/interface-constraints.dcp

# Place
place_design
write_checkpoint -force checkpoints/place-design.dcp

# Insert blocker
source goahead/tcl/module-blocker.tcl
write_checkpoint -force checkpoints/blocker.dcp

# Route design
route_design -nets [get_nets -hierarchical -filter {TYPE != "GROUND"}]
write_checkpoint -force checkpoints/route-with-blocker.dcp

# Remove blocker
route_design -unroute -physical_nets

# Reroute other physical nets
route_design -physical_nets
write_checkpoint -force checkpoints/route-without-blocker.dcp

# Generate bitstream for BitMan
set_property BITSTREAM.General.UnconstrainedPins {Allow} [current_design]
set_property BITSTREAM.GENERAL.CRC DISABLE [current_design]
write_bitstream -force module.bit
```

Listing B.0.1: Example build script

```
# Description: Static system
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
# Global variables
#-------------------------------------------------------------------------------
Set Variable=ProjectDir Value=M:\2018.3\zedboard\thesis\examples\example1\goahead_pr;
Set Variable=ZedBoard Value=%GOAHEAD_HOME%/Devices/xc7z020clg484.binFPGA;
Set Variable=GoAheadSources Value=%ProjectDir%/static/goahead/sources;
Set Variable=GoAheadTcl Value=%ProjectDir%/static/goahead/tcl;
Set Variable=EntityName Value=PartialArea;
Set Variable=InstanceName Value=inst_PartialArea;

#-------------------------------------------------------------------------------
# Load device
#-------------------------------------------------------------------------------
OpenBinFPGA FileName=%ZedBoard%;

#-------------------------------------------------------------------------------
# Floorplan the partial area
#-------------------------------------------------------------------------------
ClearSelection;
AddBlockToSelection UpperLeftTile=INT_L_X34Y99 LowerRightTile=INT_L_X38Y50;
ExpandSelection;
StoreCurrentSelectionAs UserSelectionType=PartialArea;

#-------------------------------------------------------------------------------
# Connect clock to the Partial Area
#-------------------------------------------------------------------------------
ConnectClockPins
  ClockPin=C
  BELs=[A-D]FF
  ClockNetName=clk_IBUF_BUFG
  FileName=%GoAheadTcl%/static-connect-clockpins.tcl
  Append=False
  CreateBackupFile=False;

#-------------------------------------------------------------------------------
# Generate interface constraints
#-------------------------------------------------------------------------------
# Slot (0,0) Border - West - Input
ClearSelection;
AddBlockToSelection UpperLeftTile=INT_L_X34Y90 LowerRightTile=INT_L_X34Y83;
StoreCurrentSelectionAs UserSelectionType=X0Y0_WestBorderInput;
PrintInterfaceConstraintsForSelection
  FileName=%GoAheadTcl%/static-interface-constraints.tcl
  Append=False
  CreateBackupFile=False
  SignalPrefix=x0y0
  InstanceName=%InstanceName%
  Border=West
  NumberOfSignals=32
  PreventWiresFromBlocking=True
  InterfaceSpecs=In:2-4:s2p_w;

InstantiateConnectionPrimitives
  LibraryElementName=LUTConnectionPrimitive
  InstanceName=inst_x0y0_w_in
  NumberOfPrimitives=32;

AnnotateSignalNamesToConnectionPrimitives
  InstantiationFilter=inst_x0y0_w_in.*
  InputMappingKind=external
  OutputMappingKind=internal
  SignalPrefix=x0y0
  InputSignalName=s2p_w
  OutputSignalName=dummy_w_in
  LookupTableInputPort=3;

# Slot (0,0) Border - West - Output
ClearSelection;
AddBlockToSelection UpperLeftTile=INT_L_X34Y90 LowerRightTile=INT_L_X34Y83;
StoreCurrentSelectionAs UserSelectionType=X0Y0_WestBorderOutput;
PrintInterfaceConstraintsForSelection
  FileName=%GoAheadTcl%/static-interface-constraints.tcl
  Append=True
  CreateBackupFile=False
  SignalPrefix=x0y0
  InstanceName=%InstanceName%
  Border=West
  NumberOfSignals=16
  PreventWiresFromBlocking=True
  InterfaceSpecs=Out:2-4:p2s_w;

InstantiateConnectionPrimitives
  LibraryElementName=LUTConnectionPrimitive
  InstanceName=inst_x0y0_w_out
  NumberOfPrimitives=16;

AnnotateSignalNamesToConnectionPrimitives
  InstantiationFilter=inst_x0y0_w_out.*
  InputMappingKind=internal
  OutputMappingKind=external
  SignalPrefix=x0y0
  InputSignalName=1
  OutputSignalName=p2s_w
  LookupTableInputPort=3;

#-------------------------------------------------------------------------------
# Generate partial area module in VHDL
#-------------------------------------------------------------------------------
PrintVHDLWrapper
  InstantiationFilter=.*
  EntityName=%EntityName%
  FileName=%GoAheadSources%/PartialArea.vhd
  Append=False
```

```
  CreateBackupFile=False;

#-------------------------------------------------------------------------------------
# Generate placement constraints
#-------------------------------------------------------------------------------------
ClearSelection;
SelectUserSelection UserSelectionType=PartialArea;

PrintAreaConstraint
  InstanceName=%InstanceName%
  AddResources=True
  FileName=%GoAheadTcl%/static-placement-constraints.tcl
  Append=False
  CreateBackupFile=False;

PrintExcludePlacementProperty
  InstanceName=%InstanceName%
  FileName=%GoAheadTcl%/static-placement-constraints.tcl
  Append=True
  CreateBackupFile=False;

#-------------------------------------------------------------------------------------
# Generate blocker macro
#-------------------------------------------------------------------------------------
ClearSelection;
SelectUserSelection UserSelectionType=PartialArea;

BlockWiresInSelection
  PortsToUnblockRegex=.*(1|2)(BEG|END).*;
BlockLUTInputPortsInSelection
  InputPortsRegex=.*(L|M)*_(A|B|C|D)4;

BlockSelection
  PrintUnblockedPorts=False
  Prefix=blocker_net
  BlockWithEndPips=True
  NetlistContainerName=default_netlist_container;

#SaveAsBlocker
#  NetlistContainerNames=default_netlist_container
#  FileName=%GoAheadTcl%/static-blocker.tcl
#  CreateBackupFile=False;

SaveAsBlocker
  NetlistContainerNames=default_netlist_container
  FileName=%GoAheadTcl%/static-blocker.tcl;
```

Listing B.0.2: Goa script static

```
# Description: Module - 1 Location - 1
#-------------------------------------------------------------------------------------

#-------------------------------------------------------------------------------------
# Global variables
#-------------------------------------------------------------------------------------
Set Variable=ProjectDir Value=M:\2018.3\zedboard\thesis\examples\example1\goahead_pr;
Set Variable=MODULE_NAME Value=module1;
Set Variable=ZedBoard Value=%GOAHEAD_HOME%/Devices/xc7z020clg484.binFPGA;
Set Variable=GoAheadSources Value=%ProjectDir%/%MODULE_NAME%/goahead/sources;
Set Variable=GoAheadTcl Value=%ProjectDir%/%MODULE_NAME%/goahead/tcl;

#-------------------------------------------------------------------------------------
# Load device
#-------------------------------------------------------------------------------------
OpenBinFPGA FileName=%ZedBoard%;

#-------------------------------------------------------------------------------------
# Floorplan the module area
#-------------------------------------------------------------------------------------
# 5 columns module
ClearSelection;
AddBlockToSelection UpperLeftTile=INT_L_X34Y99 LowerRightTile=INT_L_X38Y50;
ExpandSelection;
StoreCurrentSelectionAs UserSelectionType=ModuleArea;

#-------------------------------------------------------------------------------------
# Generate interface constraints
#-------------------------------------------------------------------------------------
# Slot (0,0) Border - West - Input
ClearSelection;
AddBlockToSelection UpperLeftTile=INT_L_X34Y90 LowerRightTile=INT_L_X34Y83;
StoreCurrentSelectionAs UserSelectionType=X0Y0_WestBorderInput;
PrintInterfaceConstraintsForSelection
  FileName=%GoAheadTcl%/module-interface-constraints.tcl
  Append=False
  CreateBackupFile=False
  SignalPrefix=x0y0
  InstanceName=inst_ConnectionPrimitiveWestInput
  Border=West
  NumberOfSignals=32
  PreventWiresFromBlocking=True
  InterfaceSpecs=In:2-4:s2p_w;

InstantiateConnectionPrimitives
  LibraryElementName=LUTConnectionPrimitive
  InstanceName=inst_x0y0_w_in
  NumberOfPrimitives=32;

AnnotateSignalNamesToConnectionPrimitives
  InstantiationFilter=inst_x0y0_w_in.*
  InputMappingKind=internal
  OutputMappingKind=external
```

```
  SignalPrefix=x0y0
  InputSignalName=1
  OutputSignalName=s2p_w
  LookupTableInputPort=3;

# Slot (0,0) Border - West - Output
ClearSelection;
AddBlockToSelection UpperLeftTile=INT_L_X34Y90 LowerRightTile=INT_L_X34Y83;
StoreCurrentSelectionAs UserSelectionType=X0Y0_WestBorderOutput;
PrintInterfaceConstraintsForSelection
  FileName=%GoAheadTcl%/module-interface-constraints.tcl
  Append=True
  CreateBackupFile=False
  SignalPrefix=x0y0
  InstanceName=inst_ConnectionPrimitiveWestOutput
  Border=West
  NumberOfSignals=16
  PreventWiresFromBlocking=True
  InterfaceSpecs=Out:2-4:p2s_w;

InstantiateConnectionPrimitives
  LibraryElementName=LUTConnectionPrimitive
  InstanceName=inst_x0y0_w_out
  NumberOfPrimitives=16;

AnnotateSignalNamesToConnectionPrimitives
  InstantiationFilter=inst_x0y0_w_out.*
  InputMappingKind=external
  OutputMappingKind=internal
  SignalPrefix=x0y0
  InputSignalName=p2s_w
  OutputSignalName=dummy_w
  LookupTableInputPort=3;

#-------------------------------------------------------------------------------------
# Generate connection primitives in VHDL format
#-------------------------------------------------------------------------------------
# connection primitives that are connected to the input signals of the module
PrintVHDLWrapper
  InstantiationFilter=inst_x0y0_w_in.*
  EntityName=ConnectionPrimitiveWestInput
  FileName=%GoAheadSources%/ConnectionPrimitive.vhd
  Append=False
  CreateBackupFile=False;

# connection primitives that are connected to the output signals of the module
PrintVHDLWrapper
  InstantiationFilter=inst_x0y0_w_out.*
  EntityName=ConnectionPrimitiveWestOutput
  FileName=%GoAheadSources%/ConnectionPrimitive.vhd
  Append=True
  CreateBackupFile=False;

#-------------------------------------------------------------------------------------
# Tunnels
#-------------------------------------------------------------------------------------
ClearSelection;
AddBlockToSelection UpperLeftTile=INT_R_X29Y90 LowerRightTile=INT_R_X33Y83;
StoreCurrentSelectionAs UserSelectionType=IOWestTunnel;
DoNotBlockDoubleEast;
DoNotBlockQuadEast;
DoNotBlockDoubleWest;
DoNotBlockQuadWest;

#-------------------------------------------------------------------------------------
# Generate blocker macro
#-------------------------------------------------------------------------------------
ClearSelection;
AddBlockToSelection UpperLeftTile=INT_R_X29Y109 LowerRightTile=INT_L_X42Y40;
SelectUserSelection UserSelectionType=ModuleArea;
StoreCurrentSelectionAs UserSelectionType=BlockerArea;

BlockSelection
  PrintUnblockedPorts=False
  Prefix=blocker_net
  BlockWithEndPips=True
  NetlistContainerName=default_netlist_container;

#SaveAsBlocker
#  NetlistContainerNames=default_netlist_container
#  FileName=%GoAheadTcl%/module-blocker.tcl
#  CreateBackupFile=False;

SaveAsBlocker
  NetlistContainerNames=default_netlist_container
  FileName=%GoAheadTcl%/module-blocker.tcl;

#-------------------------------------------------------------------------------------
# Generate placement constraints
#-------------------------------------------------------------------------------------
# module area
ClearSelection;
SelectUserSelection UserSelectionType=ModuleArea;

PrintAreaConstraint
  InstanceName=inst_Module
  AddResources=True
  FileName=%GoAheadTcl%/module-placement-constraints.tcl
  Append=False
  CreateBackupFile=False;

PrintExcludePlacementProperty
  InstanceName=inst_Module
  FileName=%GoAheadTcl%/module-placement-constraints.tcl
  Append=True
  CreateBackupFile=False;
```

```
160
161 # connection primitives connected to the input interface of the module
162 ClearSelection;
163 AddBlockToSelection UpperLeftTile=INT_L_X26Y99 LowerRightTile=INT_L_X26Y92;
164 ExpandSelection;
165 StoreCurrentSelectionAs UserSelectionType=ConnectionPrimitiveWestInput;
166
167 PrintAreaConstraint
168   InstanceName=inst_ConnectionPrimitiveWestInput
169   AddResources=True
170   FileName=%GoAheadTcl%/module-placement-constraints.tcl
171   Append=True
172   CreateBackupFile=False;
173
174 # connection primitives connected to the output interface of the module
175 ClearSelection;
176 AddBlockToSelection UpperLeftTile=INT_L_X26Y98 LowerRightTile=INT_L_X26Y91;
177 ExpandSelection;
178 StoreCurrentSelectionAs UserSelectionType=ConnectionPrimitiveWestOutput;
179
180 PrintAreaConstraint
181   InstanceName=inst_ConnectionPrimitiveWestOutput
182   AddResources=True
183   FileName=%GoAheadTcl%/module-placement-constraints.tcl
184   Append=True
185   CreateBackupFile=False;
```

Listing B.0.3: Goa script module

References

[Int20] Intel. *UG-20136 - Intel Quartus Prime Pro Edition User Guide: Partial Reconfiguration*. Dec. 2020.

[Xil20] Xilinx. *UG909, Vivado Design Suite User Guide Partial Reconfiguration*. Jan. 2020.

[BKT12] C. Beckhoff, D. Koch, and J. Torresen. “Go Ahead: A Partial Reconfiguration Framework” (Apr. 2012), pp. 37–44.

[Zam+18] R. Zamacola et al. “IMPRESS: Automated Tool for the Implementation of Highly Flexible Partial Reconfigurable Systems with Xilinx Vivado” (Dec. 2018), pp. 1–8.

[DÉ18] M. Darvishi and Québec). Département de génie électrique (2001-) École polytechnique (Montréal. *Characterization of Interconnection Delays in FPGAs Due to Single Event Upsets and Mitigation*. Thèse de doctorat. École polytechnique de Montréal, 2018.

[Xil18a] Xilinx. *UG470, 7 Series FPGAs Configuration*. Aug. 2018.

[Xil16] Xilinx. *UG474, 7 Series FPGAs Configurable Logic Block*. Sept. 2016.

[MB14] Yehdhih Ould Mohammed Moctar and Philip Brisk. “Parallel FPGA routing based on the operator formulation” (2014), pp. 1–6.

[Xil18b] Xilinx. *UG912, Vivado Design Suite Properties Reference Guide*. June 2018.

[LK18] C. Lavin and A. Kaviani. “RapidWright: Enabling Custom Crafted Implementations for FPGAs” (2018), pp. 133–140.

[DSC12] J. Deschamps, G. Sutter, and E. Cantó. *Guide to FPGA Implementation of Arithmetic Functions*. Vol. 149. Springer, Apr. 2012.

[Xil19a] Xilinx. *UG900, Vivado Design Suite User Guide Logic Simulation*. Oct. 2019.

[SR01] Mike Sheng and Jonathan Rose. “Mixing Buffers and Pass Transistors in FPGA Routing Architectures” (2001), pp. 75–84.

[HH07] David Harris and Sarah Harris. *Digital Design and Computer Architecture*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[BC09] J. Bhasker and Rakesh Chadha. *Static Timing Analysis for Nanometer Designs: A Practical Approach*. 1st. Springer Publishing Company, Incorporated, 2009.

[Kah+11] Andrew B. Kahng et al. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. 1st. Springer Publishing Company, Incorporated, 2011.

[BA09] K. Bruneel and F. Abouelella. “TMAP : A Reconfigurability-Aware Technology Mapper” (2009).

[Koc13] D. Koch. *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer, 2013.

[KL10] R. Khraisha and J. Lee. “A scalable H.264/AVC deblocking filter architecture using dynamic partial reconfiguration” (2010), pp. 1566–1569.

[DZT13] C. Dennl, D. Ziener, and J. Teich. “Acceleration of SQL Restrictions and Aggregations through FPGA-Based Dynamic Partial Reconfiguration” (2013), pp. 25–28.

[Ves19] M. Vesper. “Dynamic Streamprocessing pipelines on FPGAs targeting database acceleration”. Mar. 2019.

[Sas+15] P. Sasdrich et al. *Achieving side-channel protection with dynamic logic reconfiguration on modern FPGAs*. 2015.

[Het+19] B. Hettwer et al. *Securing Cryptographic Circuits by Exploiting Implementation Diversity and Partial Reconfiguration on FPGAs*. 2019.

[Hos+18] S. Hosny et al. *A Software Defined Radio Transceiver Based on Dynamic Partial Reconfiguration*. 2018.

[Pez+17] L. Pezzarossa et al. “Can real-time systems benefit from dynamic partial reconfiguration?” (2017), pp. 1–6.

[You+20] E. Youssef et al. “Energy Adaptive Convolution Neural Network Using Dynamic Partial Reconfiguration” (2020), pp. 325–328.

[IAZ21] Hasan Irmak, Nikolaos Alachiotis, and Daniel Ziener. “Increasing Flexibility of FPGA-based CNN Accelerators with Dynamic Partial Reconfiguration” (2021), pp. 1–6.

[GD14] L. Gong and O. Diessel. “Simulation-Based Functional Verification of Dynamically Reconfigurable Systems”. *ACM Trans. Embed. Comput. Syst.* 13.4 (Mar. 2014).

[GD11a] L. Gong and O. Diessel. “Modeling Dynamically Reconfigurable Systems for Simulation-Based Functional Verification” (2011), pp. 9–16.

[LSC97] W. Luk, N. Shirazi, and P. Y. K. Cheung. “Compilation tools for run-time reconfigurable designs” (1997), pp. 56–65.

[GD11b] L. Gong and O. Diessel. “ReSim: A reusable library for RTL simulation of dynamic partial reconfiguration” (2011), pp. 1–8.

[HKT13] S. G. Hansen, D. Koch, and J. Torresen. “Simulation framework for cycle-accurate RTL modeling of partial run-time reconfiguration in VHDL” (2013), pp. 1–8.

[AMM18a] I. Ahmed, H. Mostafa, and A. N. Mohieldin. “On the Functional Verification of Dynamic Partial Reconfiguration” (2018), pp. 1126–1129.

[AMM20] Islam Ahmed, Ahmed Nader Mohieldin, and Hassan Mostafa. “Functional Verification of Dynamic Partial Reconfiguration for Software-Defined Radio”. *Journal of Circuits, Systems and Computers* (2020).

[AMM18b] I. Ahmed, H. Mostafa, and A. N. Mohieldin. “Dynamic partial reconfiguration verification using assertion based verification” (2018), pp. 1–2.

[AMM18c] I. Ahmed, H. Mostafa, and A. N. Mohieldin. “Automatic Clock Domain Crossing Verification Flow For Dynamic Partial Reconfiguration” (2018), pp. 1122–1125.

[WN14] B. White and B. Nelson. “Tincr - A custom CAD tool framework for Vivado” (Dec. 2014), pp. 1–6.

[Xil21] Xilinx. *UG904, Vivado Design Suite User Guide - Implementation*. Feb. 2021.

[Mai+19] P. Maidee et al. “An Open-Source Lightweight Timing Model for RapidWright” (Dec. 2019), pp. 171–178.

[Gio+19] Raffaele Giordano et al. “Custom Scrubbing for Robust Configuration Hardening in Xilinx FPGAs”. *Instruments* 3.4 (2019).

[Yu+19] H. Yu et al. “FPGA reverse engineering in Vivado design suite based on X-ray project” (2019), pp. 239–240.

[MD20] Michail Moraitis and Elena Dubrova. “Interconnect-Aware Bitstream Modification” (2020).

[DHK17] K. Dang Pham, E. Horta, and D. Koch. “BITMAN: A tool and API for FPGA bitstream manipulations” (2017), pp. 894–897.

[Xil19b] Xilinx. *UG835, Tcl command reference guide*. 2019.

[Whe11] Bert Wheeler. *Tcl/Tk 8.5 Programming Cookbook*. Feb. 2011.

[Tcl] Tcl/Tk. *Tcl/Tk documentation*. URL: https://www.tcl.tk/man/tcl8.5/.

[Xil19c] Xilinx. *UG947, Vivado Design Suite Tutorial Partial Reconfiguration*. June 2019.

[Bec+13] C. Beckhoff et al. *Building partial systems with GoAhead*. 2013.

[Hog19] Tom Hogenkamp. “Framework for Fine-Grained Partial Reconfiguration on FPGAs”. Oct. 2019.

[Zam+19] R. Zamacola et al. “Automated Tool and Runtime Support for Fine-Grain Reconfiguration in Highly Flexible Reconfigurable Systems” (2019), pp. 307–307.

[Zam+20] R. Zamacola et al. “An Integrated Approach and Tool Support for the Design of FPGA-Based Multi-Grain Reconfigurable Systems”. *IEEE Access* 8 (2020), pp. 202133–202152.

[Vai+20] Anuj Vaishnav et al. “FOS: A Modular FPGA Operating System for Dynamic Workloads”. *ACM Trans. Reconfigurable Technol. Syst.* 13.4 (Sept. 2020).

[Gli+19] D. Glick et al. “Maverick: A Stand-Alone CAD Flow for Partially Reconfigurable FPGA Modules” (2019), pp. 9–16.

[OdR12] A. Otero, E. de la Torre, and T. Riesgo. “Dreams: A tool for the design of dynamically reconfigurable embedded and modular systems” (2012), pp. 1–8.

[VF14] K. Vipin and S. A. Fahmy. “Automated Partial Reconfiguration Design for Adaptive Systems with CoPR for Zynq” (2014), pp. 202–205.

[RFG16] J. Rettkowski, K. Friesen, and D. Göhringer. “RePaBit: Automated generation of relocatable partial bitstreams for Xilinx Zynq FPGAs” (2016), pp. 1–8.

[NTH18] B. Nelson, T. Townsend, and T. Haroldsen. *RAPIDSMITH2, A Library for Low-level Manipulation of Vivado Designs at the Cell/BEL Level*. Feb. 2018.

[YKL15] M. X. Yue, D. Koch, and G. G. F. Lemieux. “Rapid Overlay Builder for Xilinx FPGAs” (2015), pp. 17–20.

[Xil18c] Xilinx. *UG901, Vivado Design Suite User Guide Synthesis*. Dec. 2018.

[Xil18d] Xilinx. *UG903, Using Constraints*. Dec. 2018.

[BBS17] S. Berrima, Y. Blaquière, and Y. Savaria. “Sub-ps resolution programmable delays implemented in a Xilinx FPGA” (2017), pp. 918–921.

[Wik] Wikipedia. *Advanced Encryption Standard*. URL: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[Koe+99] Francois Koeune et al. “A timing attack against Rijndael” (1999).

[LBC16] Owen Lo, William Buchanan, and Douglas Carson. “Power analysis attacks on the AES-128 S-box using differential power analysis (DPA) and correlation power analysis (CPA)”. *Journal of Cyber Security Technology* 1 (Sept. 2016), pp. 1–20.

List of Figures

List of Tables

Acronyms

ABV *Assertion Based Verification*. 24
ACP *Actual Clock Period*. 55
AES *Advanced Encryption Standard*. 51, 59, 61, 62
API *Application Programming Interface*. 36
BEL *Basic Element*. 9, 52
BRAM *Block RAM*. 9, 18, 27, 42
CDC *Clock Domain Crossing*. 25
CLB *Configurable Logic Block*. 9, 12, 13, 27, 38, 52, 57
CMT *Clock Management Tile*. 9
DCP *Design Check Point*. 25, 26, 34, 45, 46, 49, 54
DPR *Dynamic Partial Reconfiguration*. 5, 7, 18, 20, 21, 24, 25, 30, 62
DPRS *Dynamically Partially Reconfigurable System*. 18, 27
DRC *Design Rule Check*. 45, 48, 53, 65
DRS *Dynamically Reconfigurable Systems*. 23, 25
DSP *Digital Signal Processor*. 9, 11, 18, 35, 42
EDIF *Electronic Design Interchange Format*. 26
FDRE *Flip-flop with clock enable and synchronous reset*. 52
FF *Flip-Flop*. 9
FPGA *Field Programmable Gate Array*. 5–11, 14, 15, 18, 21–24, 26, 27, 30–35, 37–39, 41, 57, 59, 62, 65, 66, 82
HDL *Hardware Description Language*. 10, 14, 46, 51, 54, 55, 65
HLS *High-Level Synthesis*. 15
ICAP *Internal Configuration Access Port*. 18, 27
INT *Interconnect*. 9, 12
IOB *Input/Output Block*. 9
JTAG *Joint Test Action Group*. 18
LUT *Look-Up Table*. 9–11, 14, 18–20, 27, 33, 35, 38, 48
MUX *Multiplexer*. 9
PB *Partial Bitstream*. 34, 55, 57
Pblock *Physical Block*. 30, 34, 36, 49
PCAP *Processor Configuration Access Port*. 18, 34
PIP *Programmable Interconnect Point*. 12, 27, 37, 51
PL *Programmable Logic*. 34
PR *Partial Reconfiguration*. 5, 6, 18, 19, 22–25, 30, 31, 36, 41, 43, 65
PS *Processing Subsystem*. 34
RAM *Random-Access Memory*. 10, 35
RM *Reconfigurable Module*. 24, 25, 30, 33, 36
RP *Reconfigurable Partition*. 32, 33, 35
RRG *Routing Resource Graph*. 10
RTL *Register Transfer Level*. 11, 14, 15, 22, 24, 25, 30–32, 38
SDF *Standard Delay Format*. 15, 54
SDR *Software Defined Radio*. 24

SM *Switch Matrix*. 12, 13
SoC *System on Chip*. 18, 62
SRAM *Static Random-Access Memory*. 10, 18
STA *Static Timing Analysis*. 16, 17, 26
SVA *System Verilog Assertion*. 24
TCL *Tool Command Language*. 8, 24, 28, 31, 32, 34–38, 42, 45, 47, 51, 52, 56, 62, 65
TLUT *Tunable Look-Up Table*. 20
TNS *Total Negative Slack*. 16, 17
VHDL *Very high-speed integrated circuits program HDL*. 14, 15, 24, 26, 35, 38, 46, 55
WNS *Worst Negative Slack*. 16, 17, 55