
GoAhead Tutorial Version 1.0

Document Revision

March 31, 2012

Contents

1	Introduction	3
1.1	Reconfiguration Style	3
1.1.1	Single Island Style	3
1.1.2	Multi Island Style	3
1.1.3	One-Dimensional Slot Style	3
1.1.4	Two-Dimensional Grid Style	4
2	The Tool GOAHEAD	5
2.1	Installation	6
2.1.1	Installing Cygwin	6
2.1.2	Setting Environment Variables and Xilinx ISE Version	6
2.1.3	Installing New Devices	6
2.1.4	Updating GOAHEAD	7
3	A Reconfigurable Hello World Example System	8
3.1	Implementation of the Static System	9
3.2	Generating Implementation Constraints in GOAHEAD	10
3.2.1	Defining the Reconfigurable Island	10
3.2.2	Adding a Bus Macro in GOAHEAD	12
3.2.3	Adding a Bus Macro in Xilinx ISE	13
3.2.4	Generating Physical Place & Route Constraints	16
3.3	Routing and Bitstream Generation	18
3.3.1	Running STATIC.BAT and Options	18
3.3.2	Unbind Macros	18
3.3.3	XDLConverter	20
3.3.4	Merge Blocker	20
3.3.5	Routing the Design	20
3.3.6	Timing Analysis	20
3.3.7	Bitstream Generation	20
3.4	static_constraints.goa	20
3.5	Resulting Static System	21
4	Implementing the Partial Modules	23
4.1	Creating an Empty Partial Module Project	23
4.2	Partial Module Implementation Constraints	26
4.2.1	Clock Network Constraint and Bus Macro Placement	26
4.2.2	Generating the Physical Partial Place & Route Constraints	27
4.2.3	Blocking the Center Clock Column	28
4.3	Routing and Partial Bitstream Generation	29
4.4	Important Spartan-6 Specific Topics	33
4.4.1	Spartan-6 Clock Resources	33
4.5	Further Topics	36
4.5.1	Timing Constraints	36
4.5.2	Multiple Islands and Module Relocation	36
4.5.3	Routing Optimizations	36
4.5.4	Zero Logic Overhead Integration (no Bus Macro)	37
4.5.5	Bus Macro Placement and Wire Allocation	38
4.5.6	I/O Bars for Slot-based Module Placement	39
4.5.7	Partial Bitstream Generation	39

1 Introduction

Partial run-time reconfiguration is the process of swapping parts of a reconfigurable system while allowing for continuing operation of the rest of the system. The parts of the system that are always present are called *static* while reconfigurable parts will be called *partial* or *dynamic*. To ensure reconfiguration without interference with the rest of the system, we have to ensure 1) that a partial module uses only resources (logic & routing) that are not used by other parts of the system and 2) that identical physical wire resources are used to connect partial modules. In addition, we have to 3) activate all used clock trees used by partial modules and 4) constrain the timing.

This is mostly everything to consider and there is nothing mystic behind applying partial reconfiguration! The real difficulty stems from a lack in the design tools that, for example, provide no constraints on binding a signal to specific wires on the fabric. For example, there is no constraint to bind a signal in a top level design that is responsible for the communication with a partial module.

1.1 Reconfiguration Style

Like in most practical implementations, there is no golden way to implement a reconfigurable system in a generic way and different applications and systems might demand different techniques to fulfill the system requirements best.

1.1.1 Single Island Style

Using *single island style* reconfiguration allows to swap modules exclusively in one particular reconfigurable region on the FPGA. This is useful in systems where one function is executed mutually exclusive to another one. Let us consider an SSL encrypted connection as example. Then an accelerator module responsible for the asymmetric key exchange could be replaced by a symmetric cipher which is used for the actual data transfer using the same reconfigurable island on FPGA. Note that multiple island may exist in a system but each island is bound to host its own individual set of modules. This configuration style is the only one supported by the FPGA vendor tools from Xilinx or Altera.

1.1.2 Multi Island Style

If a system provides multiple identical islands such that the same modules can be placed in different islands, we call this *multi island style*. This is, for example, useful for modules that accelerate an interface protocol in a multi-port system. Then it might be possible to partially configure an accelerator (e.g., for encryption or error correction) to the system with respect to the currently required standard for each port individually. The advantage of the multi island style is that only one physical implementation step is required. In general, physical implementation means that a module is placed and routed.

In order to implement a system providing multiple islands for hosting the same set of modules, the islands have to be *floorplanned* such that each island provides exactly the same resource primitives (e.g., look-up tables or block RAMs) at the same relative position within all islands. In this case we say that islands provide the same *resource footprint*. In addition, interfaces to the static system have to be located at exactly the same relative position within each island. Which means that the *connection footprint* is identical.

1.1.3 One-Dimensional Slot Style

Using any island reconfiguration style does not permit to directly replace a large module by multiple smaller ones which might result in costly *internal fragmentation*. Internal fragmentation is a waste of resources that stems from different modules requiring different number of resources. In other words, the largest module defines the size of an island which might then not be effectively utilized

when hosting another tiny module. As a solution to improve resource utilization, one reconfigurable area can be divided and organized into a set of adjacent one-dimensional aligned *resource slots*. Modules can then take one or more slots (depending on their resource requirements)

An example for the 1D slot style approach would be a network packet inspection system. Here, different inspection accelerator modules might be placed into a reconfigurable region with respect to the present protocol load of the traffic. Consequently, the system can adapt to different workload scenarios by loading different numbers of accelerator modules to the system.

However, different sized modules reduce internal fragmentation but can come along with *external fragmentation*. External fragmentation arises by loading and terminating modules over time. Then free area will in general not be aligned in one continuous region but fragmented in multiple smaller resource slots. In other words, counting the number of free resource slots alone is not sufficient to determine if a module can be placed or not. The resources are typically required as a set of continuous available free resource slots and the resource footprint has to match the module footprint.

1.1.4 Two-Dimensional Grid Style

Even making the resource slots as narrow as possible (i.e. one CLB column wide on Xilinx FPGAs), can still result in unlikely large resource slots. In particular when using dedicated primitives like block RAMs or multipliers, someone would not spend a larger column of these resources if the modules might only require a few of these primitives. In this case, we divide the slots another time such that the resource slots are organized in a two-dimensional grid. This has the advantage of further reducing internal fragmentation, but makes the system implementation and the module placement at run-time more challenging.

2 The Tool GOAHEAD

The tool GOAHEAD provides floorplanning capabilities, communication infrastructure generation (required to integrate reconfigurable modules), and constraints generation required for the place and route implementation phase. GOAHEAD provides an intuitive graphical user interface as well as a scripting interface. All GUI actions are recorded by corresponding script commands. Consequently, there is no need to learn the GOAHEAD scripting language. However, it is recommended to use the GUI to create scripts. This removes error sources and ensures reproducible results. When calling GOAHEAD without parameters, it will start with the GUI. As an alternative, it is possible to call the tool in batch mode:

```
goahead my_script.goa
```

This allows to run the whole implementation process including the generation of all bitstreams as a single shot batch job.

GOAHEAD is not restricted to work together with a particular synthesis tool. However, throughout this document, we will demonstrate how GOAHEAD can interact with the Xilinx XST synthesis tool.

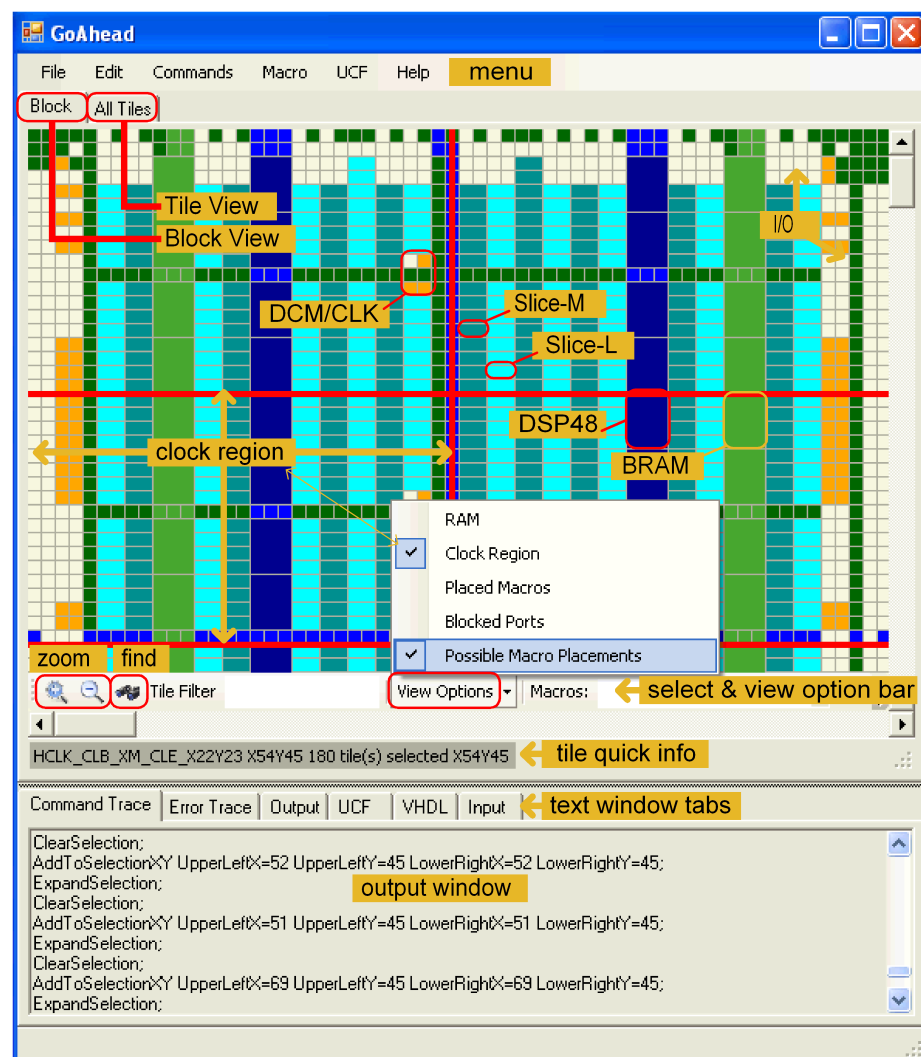


Figure 1: GOAHEAD GUI showing a Spartan-6 LX16 FPGA. All GUI actions will be recorded in the Command Trace output window.

2.1 Installation

Download the tool, run the installation program and follow the instructions. Depending on your windows system, the installation of a .net framework might be required. Note that GOAHEAD has been written in C#. Linux users can also use GOAHEAD with the help of MONO. GOAHEAD is working with recent MONO versions.

2.1.1 Installing Cygwin

We utilize the Cygwin environment to run some powerful bash commands under Windows (e.g., grep or sed) as Windows alone is still very limited to do advanced batch operations. For our needs, the portable version is sufficient and it is freely available from sourceforge:

sourceforge.net/projects/cygwinportable/

Users having an EDK installation could use their already existing Cygwin environment that can be found under (or something similar):

`C:\YourXilinxPath\13.2\ISE_DS\EDK\gnuwin\bin`

2.1.2 Setting Environment Variables and Xilinx ISE Version

The entire latest Version of the Xilinx Vendor tools is not necessarily the most stable one. Furthermore, different versions have sometimes different behavior for different device families. Therefore, it is recommended to install new versions of the Xilinx vendor tools in parallel to the existing installations rather than upgrading a particular ISE Version. Consequently, multiple different versions might be installed in parallel on the same machine. Under Windows, we recommend to switch among the different versions using a batch file. If your installations have followed the Xilinx suggestions (i.e. each versions is installed in a corresponding path `YourXilinxPath\version\`), you could use the following batch file to set the environment:

```
set XILINX=YourXilinxPath\%1\ISE_DS\ISE
set PATH=%XILINX%\bin\nt;YourCygwinPath\bin;YourGoAheadPath\%PATH%
```

Note that we also include the path to Cygwin and to GOAHEAD . If you store this file as `SetEnv.bat`, you can set ISE version 12.3 by calling `'SetEnv.bat 12.3'` in a DOS box. After this, you can start the currently selected ISE version by calling `ise` in the command line window. The same holds respectively for GOAHEAD , which can then be called with `goahead`.

Depending on your ISE installation, you might also set your license server in that file:

```
LM_LICENSE_FILE=port@server_address
```

2.1.3 Installing New Devices

The GOAHEAD installation contains a library providing various FPGA device descriptions in the internal `.binFPGA` format. In a few cases, these descriptions might fail to be opened in your GOAHEAD installation. This is caused by compatibility problems among different .net framework versions. In this case, or if a required device is not in the library, it is possible to use the Xilinx command line tool XDL to create device descriptions in XDL format. Here are some XDL examples:

```
xdl -report -pips -all_conns xq5v1x30t xq5v1x30t.xdl
xdl -report -pips -all_conns xc6slx16 xc6slx16.xdl
xdl -report -pips -all_conns xc6slx45 xc6slx45.xdl
xdl -report -pips -all_conns xc6vlx130t xc6vlx130t.xdl
```

These examples require a proper installation of Xilinx ISE, including the setting of the `XILINX` environment variable and the device libraries. With the help of the `partgen` command line tool, it is possible to determine the installed devices:

```
partgen -i                // list all installed devices
partgen -arch Spartan6    // list only Spartan6 devices
```

After generating an XDL description, open it in GoAhead with FPGA -> Open XDL. Next, save this description in the internal GOAHEAD format for further usage with FPGA -> Save Raw. As an alternative, a GOAHEAD script (.goa) can be used for running the process from the command line:

```
ReadXDL FileName=C:\My_XDL_devices\xc6slx16.xdl;
```

```
SaveFPGA FileName=c:\My_GOA_devices\xc6slx16csg324-3.binFPGA;
```



TIP XDL descriptions can be unlikely huge and up to several gigabytes for larger devices (e.g., 42 GB for the largest Virtex-7 device, XC7V2000t). The generation of these files as well as reading the XDL device descriptions into GOAHEAD can take several hours. However, this process can be done as an overnight batch process; and after reading in the XDL description into GOAHEAD, it can be stored in our own internal format which is orders of magnitude more compact and faster. The process is supported by GOAHEAD. Go to the menu File -> PartGen to generate device descriptions for certain FPGA families.

2.1.4 Updating GoAHEAD

Updating Goahead can be accomplished by simply overwriting the goahead.exe executable. However, different versions of GOAHEAD might use slightly different internal data structures that are not compatible among multiple versions of the tool. Consequently, the device descriptions in the internal binary format (i.e., your *.binFPGA files) have to be generated again.

For the same reason, the file storing the preferences should be deleted. It is stored under %appdata%\GoAhead and is called preferences.bin. Deleting this file will restore the GOAHEAD default settings.



ISSUE In a few installations of the Xilinx ISE tools, we have not been able to start more than one instance of the FPGA editor. In this case, it doesn't matter if the GUI version or the command line version of the FPGA editor is running. As GoAhead (and in particular the routing scripts) use the FPGA editor, all running instances might have to be closed if your system is affected by this.

3 A Reconfigurable Hello World Example System

The following sections describe the implementation of a tiny system using simple island style reconfiguration. The system will replace a module with AND/NAND gates by a corresponding one with OR/NOR gates. We will firstly implement the static system and then, completely independent from this, the two reconfigurable modules.

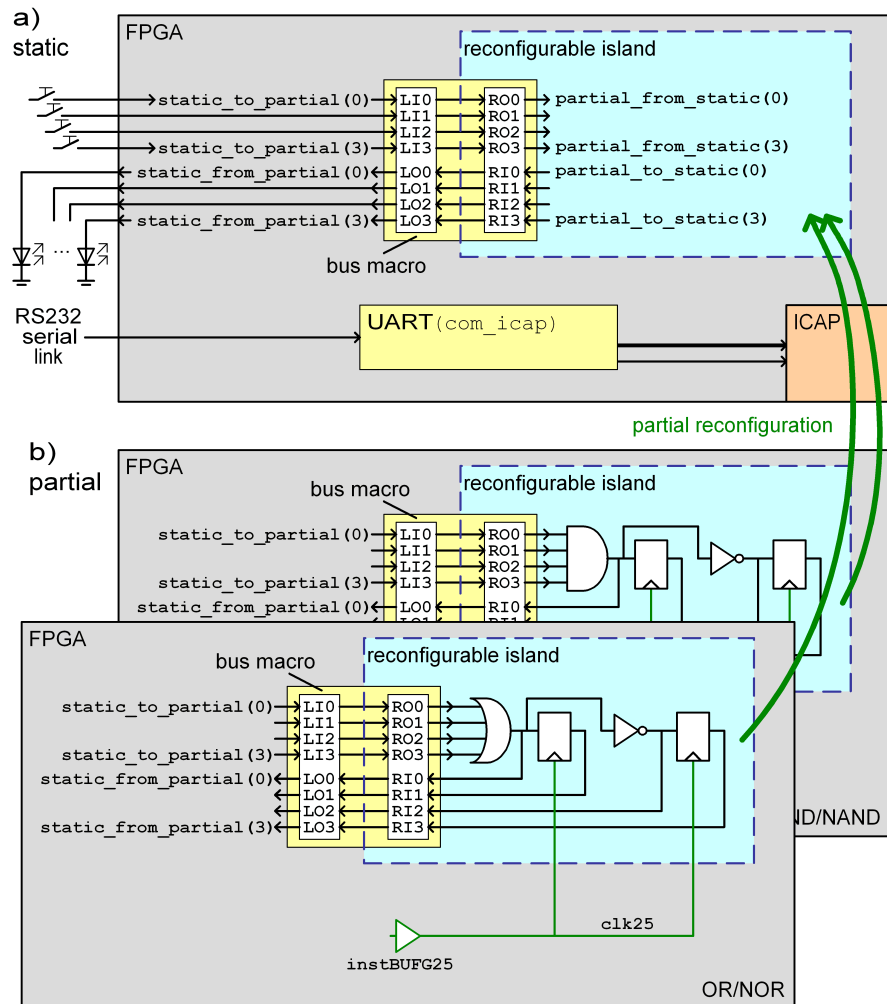


Figure 2: Reconfigurable "Hello World" system. The system consists of a static system a) which comprises a reconfigurable island to host modules that are implemented independently in b).

As shown in Figure 2, the static system comprises a reconfigurable island for hosting two different reconfigurable modules. In order to guarantee that always the same wires are used for the communication between the static system and the reconfigurable island, a *bus-macro* will be used. Such a macro is nothing else as two look-up tables (in this example placed left and right in two consecutive CLB columns) and a set of wires between them. To be more precise, the bus macro consists of two slices with four LUTs each for bridging four signals from left-to-right as well as for bridging four signals from right-to-left. Except any clock signals (which are routed via dedicated clock networks), all signals connecting a partial module (i.e., the top level port signals of the reconfigurable modules) have to be bridged using a bus macro. In Section 4.5.4, we will demonstrate that constraining a signal to a particular wire on the FPGA fabric can be accomplished more efficient without bus macros.

3.1 Implementation of the Static System

The archive with the "Hello World" tutorial contains two directories for the static system and the partially reconfigurable system. At the beginning, we open the project in the static directory in ISE (the provided project was created with version 13.1, but runs with all 13.x versions). The project contains only the following five design files:

- `static.vhd` The static system follows mostly the system as illustrated in Figure 2a), except that there is no reconfigurable island or bus macro defined. In other words, except the UART interface to the ICAP port, this system is currently not implemented to support partial runtime reconfiguration. The push buttons are connected through a multiplexer to the LEDs on the board. As can be seen in the code, this has been done to share the LEDs for displaying some state additional information. We used multiplexing, controlled by an extra input, as some boards, including the SP601, provide only a few LEDs. This might be removed on boards with more LEDs.
- `com_icap.vhd` A tiny UART that is connected with some glue logic to the internal ICAP port. Note that we found that partial reconfiguration on Spartan-6 FPGAs is only supported using the internal ICAP reconfiguration port.
- `testXOR.vhd` This is a test module for test purpose. By changing the look-up table function of this single LUT module inside the FPGA-editor, a partial bitstream can be easily generated using the `bitgen -r` option. Such a bitstream was used to test partial reconfiguration via ICAP.
- `static.ucf` The user constraints file contains only the top-level signal to I/O pin assignments, a clock constraint, and placement information for the testXOR test module.
- `PAR_stress.vhd` A dummy module used later to fill up the FPGA. This is used to put more stress on the placement and routing process. This can be done by increasing the value of `NoOfRegs => .`. A value of `NoOfRegs => 700` will result in almost full logic utilization on a Spartan-6 LX16.

It is recommended to test this design on the board in order to ensure that UCF signal constraints are specified correctly. On the SP601 board, we multiplexed the LEDs. There, depending on switch `GPIO_SWITCH(0)`, we select between passing through the push buttons to the output LEDs or to display some internal states. The latter selection can be identified by LED3 that is blinking. Furthermore, it is recommended to test the serial transmission by sending the provided test file to the COM port. For this reason, open a command shell and change to the static project directory; then we type the following:

```
MODE                                // to figure out the available com ports
MODE COMi:115200,N,8,2             // set com port-i to 115.2 kboud
copy com_test.hex COMi             // send test data to the com port (e.g., COM5)
```

Before downloading the test file, ensure that the state signals are connected to the LED (LED3 is blinking). During test file transfer, LED0 should shortly light up. If not, the com port or baud-rate might be wrong. If LED0 starts blinking after the test, data got clipped, stuck, or corrupted. Note that the checksum is hard encoded for `com_test.hex`. Consequently, LED0 will blink after transferring any other file to the UART. For using the binary mode, the file `com_test.bin` has to be sent to the com port. Note that the default UART provides both modes simultaneously.



ISSUE The CP2103 USB-to-UART bridge of the SP601 or the corresponding driver caused interruptions on larger data on some systems. Sometimes using Cygwin or changing to HEX instead of binary data transfer helped. However sometimes even this failed and we used an FTDI cable instead as shown in Figure 3. With DIP switch 3 = on (which corresponds to signal `GPIO_SWITCH(2)='1'`) the UART is connected to FTDI otherwise data is accepted from the USB bridge of the SP601. Note that using this approach required to use the HEX transfer mode in our setup.

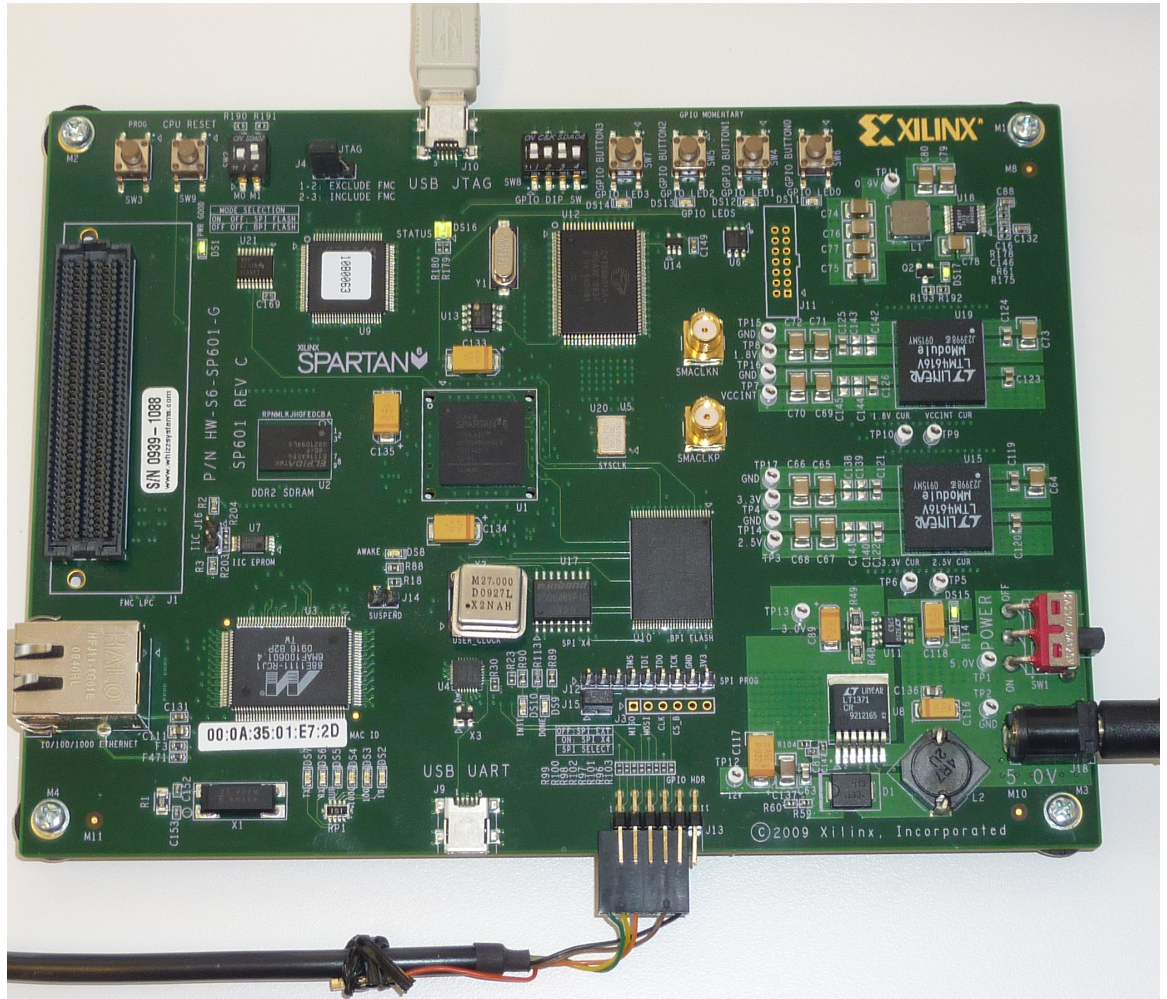


Figure 3: USB-UART replacement using an FTDI cable on the SP601 development board. The red power cable has been removed from the FTDI plug and the cable is plugged into J13 such that the yellow cable connects to pin 1; consequently, the green wire is not connected at all.

3.2 Generating Implementation Constraints in GOAHEAD

We will now add a reconfigurable island and a bus macro to the static design. We have to ensure that no static logic and static routing is used inside the reconfigurable island¹. This avoids resource conflicts with the partial modules hosted in the reconfigurable island at run-time.

For the following steps, we have to start GoAhead and open the device description of the used device target FPGA. In case of the SP601 board, we use the command `FPGA -> Open Raw` and select the file `xc6s1x16csg324-3.binFPGA` from the devices library. Refer to Section 2.1.3 if this causes problems.

3.2.1 Defining the Reconfigurable Island

After loading the device description in GOAHEAD, we see an empty device in the Block view. We will not use the more detailed All Tiles view in this tutorial. We select the reconfigurable island

¹We will learn later that there are good reasons to relax this constraint (e.g., for crossing a static signal through the reconfigurable island).

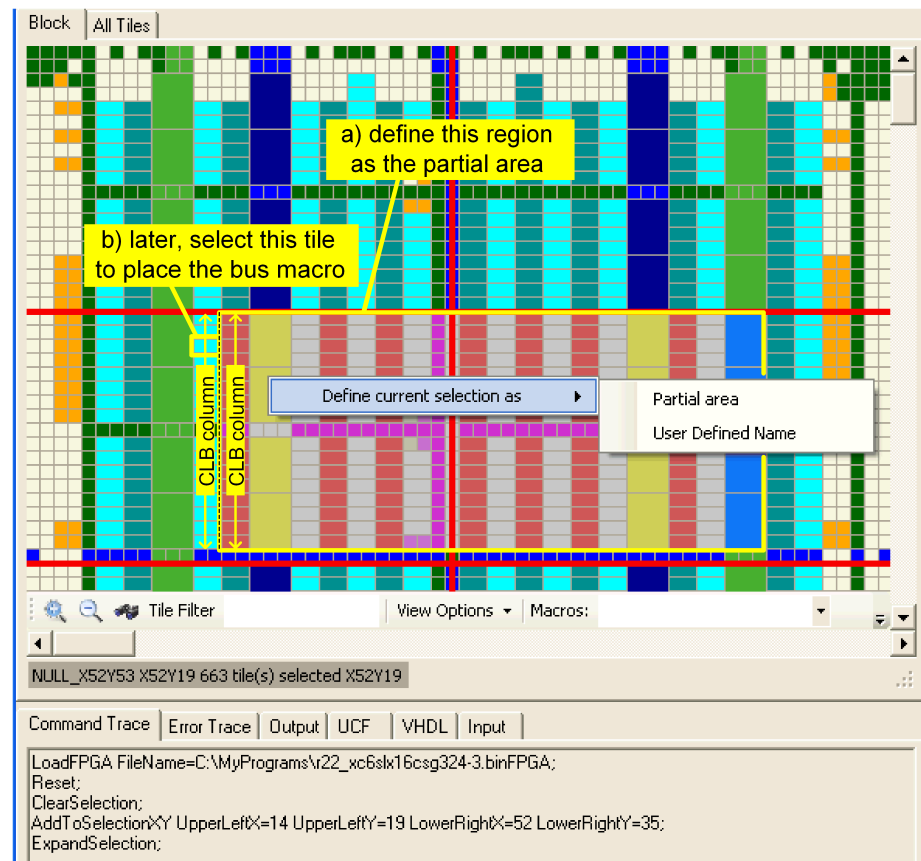


Figure 4: FPGA floorplanning. a) in the beginning, we define the reconfigurable island as the `Partial` area; b) later, we place the bus macro, as described in Section 3.2.2 at the marked tile.

as shown in Figure 4. We select an area that spans the height of a clock region. Clock regions are 16 CLBs, 4 BRAMs, or 4 DSP blocks high on Spartan-6 devices. The borders of the clock regions are marked in the GOAHEAD GUI, if activated in the View Options.

As all I/O pins are located at the four edges of the chip, it is recommended not to place a reconfigurable island too close the edges of the fabric. For the same reason, it is not recommended (but possible) to use to clock regions at the very top or bottom of a Spartan-6 FPGA. Our reconfigurable region can contain arbitrary resources including logic (i.e., CLBs, BRAMs, or DSP48 tiles). However, for placing the bus macro in the next step, we have to ensure that CLBs are located at both sides of the left border of the reconfigurable island.

For defining the partial area, we use the context menu (right mouse button) and execute `Define current selection as` -> `Partial area`, as shown in Figure 4. This results in a selection of resources that we can refer to by the symbolic name `PartialArea`.

It is recommended to play a bit around to learn the usage of the `<SHIFT>` and `<CTRL>` key to select arbitrary areas in the block view.

When locking into the command trace, we will find the following commands that we could copy and paste into a GOA script file:

```
1 ClearSelection;
2 AddToSelectionXY UpperLeftX=14 UpperLeftY=19 LowerRightX=52 LowerRightY=35;
3 ExpandSelection; # ensures to select logic and routing tiles together
4 StoreCurrentSelectionAs UserSelectionType=PartialArea;
```

These commands store a selection under the symbolic name `PartialArea`. This allows us to re-select this selection anytime later with the command `SelectUserSelection UserSelectionType=PartialArea;`. This command is available directly via the `select & view` option bar. Note that any arbitrary selection (e.g., non-rectangular selections) can be stored in this way.

3.2.2 Adding a Bus Macro in GOAHEAD

We have now to announce to GOAHEAD which kind of macro we will use (or macros, as multiple different macros can be used). This could be done by adding the macro to the Library Manager which is opened from the menu via `Macro -> Macro Library`. However, the same functionality is also available in the Macro Placer that is located under the Macro menu. Inside the Macro Placer, we use `Add XDL Macro` or `Add binary Macro` to open macro descriptions from the macro library provided with GOAHEAD. It is possible to generate own XDL Macros and convert them inside the Macro Placer or Library Manager with `Save Macro` to an internal GOAHEAD macro description that is faster to process by the tool.

For this tutorial on a Spartan-6 FPGA, we open the following file from the GOAHEAD Macros directory:

`BM_S6_L4_R4_single.xdl`



TIP XDL Macros can also be loaded using the `AddLibraryElementFromXDL` command in a start-up GOA script (see your command trace after loading a macro for an example). In addition, all present macros in the Library Manager can be stored (with `Save Library`) in one file that can be re-opened after restarting GOAHEAD using `Load Library`.

We now activate in the `View Options` the option `Possible Macro Placements`. This will highlight all possible placement positions (i.e., macro reference points or anchors) of the current macro that is selected in the top down selection menu right beside the `View Options`.

If the macro is loaded to GOAHEAD and if the Macro Placer dialog is open, we can define a macros' placement position by clicking on the target reference tile. This fills the fields `Anchor Tile` and `Anchor Slice` in the Macro Placer. For placing a bus macro, we select a tile left outside the reconfigurable area as shown in Figure 4.

Next, we specify the instance name of the macro as `instBusMacro` inside the Macro Placer dialog. This name will be used for the VHDL and UCF template generation. We can compare our settings with Figure 5.

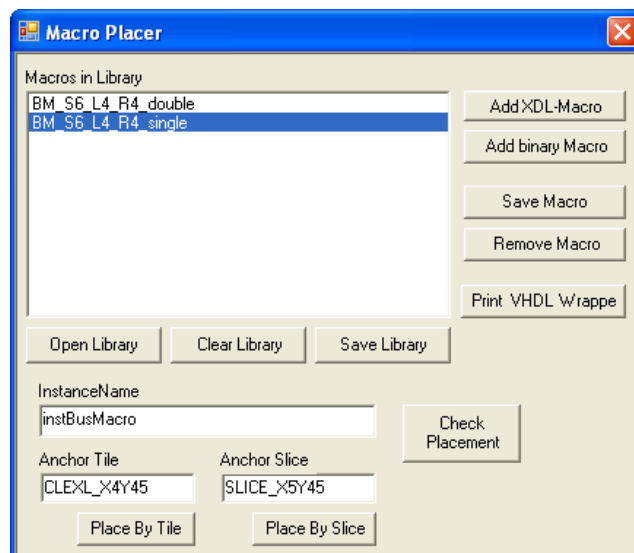


Figure 5: Bus macro instantiation using the Macro Placer.

GOAHEAD can check if the selected macro of the library can be placed at the specified position by clicking the **Check Placement** button. The placement itself is accomplished by either clicking **Place by Tile** or **Place by Slice**. Both commands perform the same action and differ only in the coordinate system used to specify the macro reference point. While the tile grid is more intuitive to use, specifying slice coordinates only has had a longer tradition.

The whole process results into the following log in the **Command Trace** that we can copy to our GOA script (if we use **Place by Slice**):

```

5  AddLibraryElementFromXDL XDLMacro=C:\WORK\GoAheadMacros\BM_S6_L4_R4_double.xdl↵
    ParseXDLPortStatements=True;
6  AddLibraryElementFromXDL XDLMacro=C:\WORK\GoAheadMacros\BM_S6_L4_R4_single.xdl↵
    ParseXDLPortStatements=True;
7  ClearSelection;
8  AddToSelectionXY UpperLeftX=12 UpperLeftY=21 LowerRightX=12 LowerRightY=21;
9  ExpandSelection;
10 AddSingleMacroInstantiationBySlice MacroName=BM_S6_L4_R4_single InstanceName=↵
    instBusMacro SliceName=SLICE_X5Y45;

```

3.2.3 Adding a Bus Macro in Xilinx ISE

GOAHEAD can generate VHDL instantiation templates for all macros instantiated within the tool. Before we start this process, we select the VHDL text window tab in the GOAHEAD GUI. Next, within the **Macro Placer**, we select the **BM_S6_L4_R4_single** macro and click on **Print VHDL Wrapper** in order to print the VHDL declaration of the used bus macro into the output window:

```

1  component BM_S6_L4_R4_single is Port (
2    RH0 : in STD_LOGIC;
3    RH1 : in STD_LOGIC;
4    RH2 : in STD_LOGIC;
5    RH3 : in STD_LOGIC;
6    LH0 : in STD_LOGIC;
7    LH1 : in STD_LOGIC;
8    LH2 : in STD_LOGIC;
9    LH3 : in STD_LOGIC;
10   RI0 : in STD_LOGIC;
11   RI1 : in STD_LOGIC;
12   RI2 : in STD_LOGIC;
13   RI3 : in STD_LOGIC;
14   RO0 : out STD_LOGIC;
15   RO1 : out STD_LOGIC;
16   RO2 : out STD_LOGIC;
17   RO3 : out STD_LOGIC;
18   LI0 : in STD_LOGIC;
19   LI1 : in STD_LOGIC;
20   LI2 : in STD_LOGIC;
21   LI3 : in STD_LOGIC;
22   LO0 : out STD_LOGIC;
23   LO1 : out STD_LOGIC;
24   LO2 : out STD_LOGIC;
25   LO3 : out STD_LOGIC );
26 end component BM_S6_L4_R4_single;

```

We use the mouse context menu (right button) to copy & paste this output from the VHDL text box to the architecture declaration block of our static top implementation file `static.vhd`.

After this, we close the Macro Placer and open the Macro Instantiations dialog which is located in the GOAHEAD Macro menu. The new window provides a table containing information about our instantiated bus macro. Now, we click once on Print Macro Instantiations button and we can see a macro instantiation template in the VHDL output window that corresponds to the macro declaration from the previous step.

However, there are currently no signals connected to this instantiation. We can add signals by adding Port Mappings. A port mapping has the form:

MacroPortPrefix_a:{Signal_a|0|1|open}[,MacroPortPrefix_b:{Signal_b|0|1|open}]...

This means that we can specify the beginning of a macro port name (e.g., RH) in order to set the port to a constant '1' value by adding the port mapping RH:1. We can add more mappings by separating them with commas. If we connect a port prefix to a signal (e.g., LI:static_to_partial), vector connections will be generated for this signal. We will connect the LH and RH inputs to '1' and the remaining ports according to the signal names listed in Figure 2. The corresponding port mapping is:

RH:1,LH:1,LI:static_to_partial,L0:static_from_partial,RI:partial_to_static,
R0:partial_from_static

The Macro Instantiations window should look as follows:

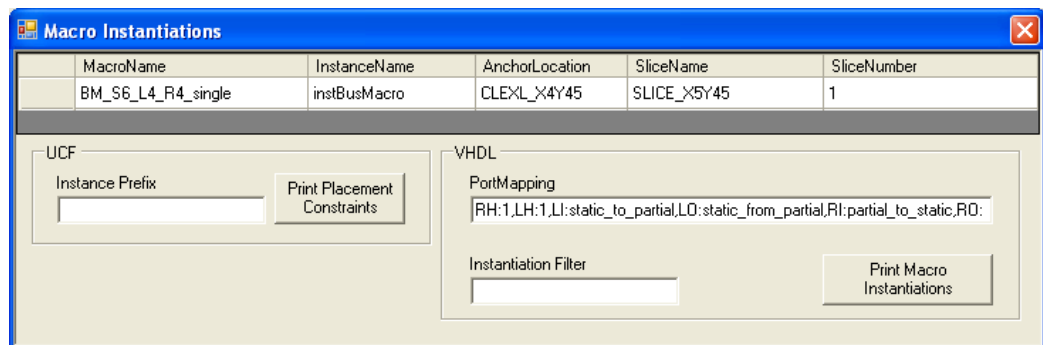


Figure 6: Macro instantiation window containing a port mapping string.

After clicking Print Macro Instantiations, the VHDL output window contains:

```

310 instBusMacro : BM_S6_L4_R4_single
311 Port Map (
312   RH0 => '1',
313   RH1 => '1',
314   RH2 => '1',
315   RH3 => '1',
316   LH0 => '1',
317   LH1 => '1',
318   LH2 => '1',
319   LH3 => '1',
320   RI0 => partial_to_static(0),
321   RI1 => partial_to_static(1),
322   RI2 => partial_to_static(2),
323   RI3 => partial_to_static(3),
324   RO0 => partial_from_static(0),
325   RO1 => partial_from_static(1),
326   RO2 => partial_from_static(2),
327   RO3 => partial_from_static(3),
328   LI0 => static_to_partial(0),
329   LI1 => static_to_partial(1),

```

```

330 LI2 => static_to_partial(2),
331 LI3 => static_to_partial(3),
332 LO0 => static_from_partial(0),
333 LO1 => static_from_partial(1),
334 LO2 => static_from_partial(2),
335 LO3 => static_from_partial(3) );

```

We use the mouse context menu in order to copy this instantiation template at the end of our top implementation file (static.vhd). Next, we have to add the declarations of the new signals to static.vhd:

```

27 signal partial_from_static : std_logic_vector(3 downto 0) := (others=>'1');
28 signal partial_to_static : std_logic_vector(3 downto 0) := (others=>'1');
29 signal static_from_partial : std_logic_vector(3 downto 0) := (others=>'1');
30 signal static_to_partial : std_logic_vector(3 downto 0) := (others=>'1');
31
32 attribute keep of partial_from_static : signal is "true";
33 attribute keep of partial_to_static : signal is "true";
34 attribute keep of static_from_partial : signal is "true";
35 attribute keep of static_to_partial : signal is "true";

```

Note that we initialized all signals to '1'. This is done as each Slice, BRAM, or DSP48 primitive input can be set to '1' without additional logic (which is not the case for the default '0' value). In addition, a keep attribute is set for each signal. Together with the map -u option (do not truncate unconnected signals), this allows to implement a module without the need of external signals. As the last modification to our static design, we connect the bus macro with the push button and LED signals:

```

336 — LocalLEDs <= GPIO.BUTTON; —uncomment this line after adding the bus macro
337 LocalLEDs <= static_from_partial;
338 static_to_partial <= GPIO.BUTTON;

```



TIP Port mappings can be used among multiple macros at the same time. In this case, the vectorization of signals is continued over all macros that match the port prefix. Furthermore, it is possible to optionally filter the involved macro instance names using regular expressions in the Instantiation Filter mask.



ISSUE Despite that macros can contain vectorized entities, single bit signals have been used for the entity signals of all provided macros in the library. This is a bit cumbersome but helped us to remove crashes in the Xilinx ISE tool chain. However, the instantiation templates are connected more convenient to signal vectors.



ISSUE Several Xilinx ISE Versions have had problems when instantiating NMC hard-macros in a sub-hierarchy of a design. In order to overcome this issue, we highly recommend to instantiate macros only at the top level of a design.

The important Command Trace entries that have been logged by GOAHEAD are:

```

11 PrintWrapperForMacro MacroName=BM_S6_L4_R4_single
12 PrintVHDLMacroInstantiation PortMapping=RH:1,LH:1,LI:static_to_partial,LO:↔
    static_from_partial,RI:partial_to_static,RO:partial_from_static

```

3.2.4 Generating Physical Place & Route Constraints

By instantiating a macro in GOAHEAD, its resources are announced to the tool and can be considered when generating implementation constraints. This is used to generate UCF placement constraints for the `static.ucf` file of our current ISE project.

For the next steps, we select the UCF text window tab in the GOAHEAD GUI. At first, we generate the macro placement constraint by clicking `Print Placement Constraints` in the Macro Instantiations Window. This will generate the following output in the UCF output window:

```
30 INST "instBusMacro" LOC = "SLICE_X5Y45"; # generated_by_GoAhead
```

Now, we will generate placement prohibit constraints for the reconfigurable island in order to prevent the static system from instantiating resources inside this region. We start this by clearing the present selection via the menu `Edit -> Clear Selection`. Next, we select all resources of our reconfigurable island by selecting the `PartialArea` group in the drop down menu of the select and view option bar (see also Figure 7a). With this selection, we run from the menu the `UCF -> Prohibit` command. We leave the `Exclude used Slices` checkbox activated. Otherwise, the bus macro slices would get prohibited which would prevent us to place the macro. The result is a list of UCF constraints containing primitives that are not allowed to be used by the placer.

```
31 CONFIG PROHIBIT = "SLICE_X6Y47"; # generated_by_GoAhead
32 CONFIG PROHIBIT = "SLICE_X7Y47"; # generated_by_GoAhead
33 CONFIG PROHIBIT = "SLICE_X6Y46"; # generated_by_GoAhead
34 ...
```

Next, we use the mouse to select the constraints from the UCF output window and copy & paste them at the end of `static.ucf`. After this, we run up to the ISE placement which is done by MAP. Note that opposed to earlier versions of ISE, the placement is now done during the map phase. Despite that the route tool is called *par* for place and route, it only provides routing functionality.

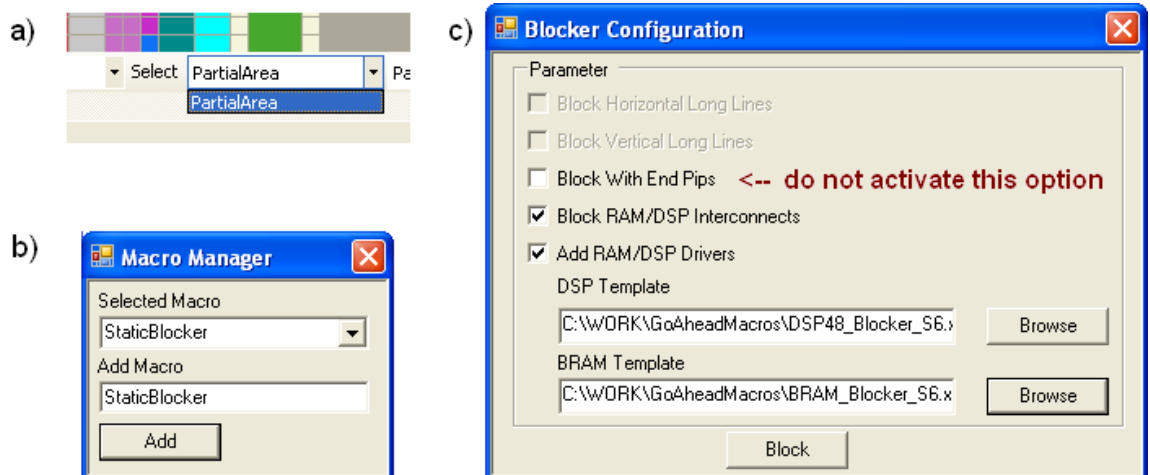


Figure 7: a) `PartialArea` selection. b) Adding a `StaticBlocker` macro. c) Blocker settings.

Note that prohibit statements have no impact on the router. For prohibiting the router from using resources within the reconfigurable island, GOAHEAD can generate *blocker macros* that occupy all routing resources within a definable region. We start the blocker macro generation by adding a new macro with the name `StaticBlocker` inside the Macro Manager (see Figure 7b)). The Macro Manager is opened from the menu with `Macro -> Macro Manager`. After this, we ensure

that the PartialArea is still selected (Figure 7a)). Next, we run from the menu **Commands -> Block Selection**. As our reconfigurable island contains BRAMs and DSP48 primitives, we have to provide the corresponding RAM and DSP connection primitives for Spartan-6 FPGAs (see Figure 7c)). Depending on the size of the selection, the blocker generation may take a few seconds or even minutes. Finally, we generate the blocker macro (in the XDL language) via the menu with **Macro -> Generate XDL Code**. In the opened dialog (see Figure 8), we keep the default settings and specify the output file as `StaticBlocker.xdl` that has to be stored in our static project directory.

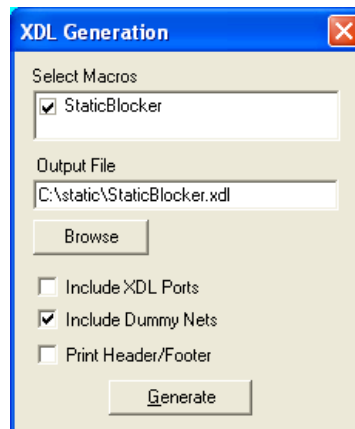


Figure 8: XDL generation options for the StaticBlocker.



ISSUE We disabled the "Block With End Pips" option in the blocker configuration, which is allowed, whenever using the bus-macro design methodology. Not activating this switch 1) permits the router to go into the reconfigurable region but will 2) remove any option to leave the region, which consequently will remove the first option so that we can still be sure that no unintended routing crosses the reconfigurable region. However, in this tutorial, the reconfigurable region includes some primitives to drive global clock nets that we should not block (otherwise the global clock nets will not get completely routed. This particular driver primitives can only be blocked with the "Block With End Pips" option. Disabling "Block With End Pips" causes actually the wanted side effect to allow the clock drivers. As one alternative to this, we could remove the clock primitive tile from the selection. In this example, it is `REG_V_HCLK_X29Y45`. Use the **Tile Search** or type `REG_V_HCLK<ENTER>` into the **Tile Filter** to find that tile (see also Section 4.4.1).

The following GOAHEAD commands have been executed (and logged in the Command Trace) when generating the place & route constraints:

```

13 PrintLocationConstraintsForPlacedMacros ;
14 ClearSelection ;
15 SelectUserSelection UserSelectionType=PartialArea ;
16 PrintProhibitStatementsForSelection ExcludeUsedSlices=True FileName= Append=<
    True ;
17 AddMacro MacroName=StaticBlocker ;
18 SelectMacro MacroName=StaticBlocker ;
19 BlockSelection SliceNumber=0 BlockArcsInRAMTiles=True BlockWithEndPips=False <
    AddDSPBRAMDriver=True DSPTemplate=C:\WORK\GoAheadMacros\DSP48_Blocker.S6.<
    xdl BRAMTemplate=C:\WORK\GoAheadMacros\BRAM_Blocker.S6.xdl <
    PrintUnblockedPorts=False ;
20 GenerateXDL FileName=C:\WORK\GoAhead_Tutorials\HelloWorld\static\StaticBlocker<
    .xdl MacroNames=StaticBlocker IncludePorts=False IncludeDummyNets=True <
    IncludeModuleHeader=False IncludeModuleFooter=False ;

```

3.3 Routing and Bitstream Generation

After running the synthesis and technology mapping in Xilinx ISE (or any other third party synthesis tool), we use the script `static.bat` to accomplish the routing and bitstream generation. The following sections describe these automatically performed steps in the order of execution and as they are illustrated in Figure 9.

3.3.1 Running STATIC.BAT and Options

As some parameters, like for example the installation paths or the used device change only seldom, these parameters have been defined using variables within `static.bat`. The definitions are located in the beginning of the script files and specify the path to the Cygwin binaries and the GOAHEAD device description file of the used FPGA:

```
1 SET cygpath="C:\EDA\CygwinPortable\App\Cygwin\bin"  
2 SET binFPGApath="c:\MyPrograms\%GoAheadVersion%\xc6slx16csg324-3.binFPGA"
```

These variables have to be adjusted according to the local installation before running the script. The routing script is called as follows:

```
>static InputProject [Blocker_XDL_FILE BUFGInstanceName] [Blocker_XDL_FILE ] ...
```

The options are:

InputProject:	this is typically the prefix of the <code>MyProject.map.ncd</code> file. Note to specify the prefix without the <code>_map.ncd</code> at the end.
Blocker_XDL_FILE:	a blocker file generated by GOAHEAD to constrain the routing
BUFGInstanceName:	an optional instance name used to connect blocker primitives to a global clock net (see Section 3.3.4 for more documentation)

For this tutorial, we open a DOS box, change to the static project directory, ensure that the environment variables are set (see Section 2.1.2), and run the script:

```
>static top StaticBlocker.xdl instBUFG25
```

3.3.2 Unbind Macros

The input NCD-file might contain hard macros that have been introduced to the design by the instantiation of NMC-files in the HDL design (e.g. by using a provided bus-macro). Such hard macros can contain routing information which is in most cases essential to be preserved for the final routing step. However, by converting a design from NCD to XDL and again back to NCD format, routing information of hard macros gets lost. By unbinding all hard macros, this conversion is possible without losing the routing information. Note that GOAHEAD requires a conversion of the placed design to XDL for adding routing constraints to the design.



ISSUE Writing back the NCD file within the FPGA-Editor script will throw errors indicating that signals are unrouted. This is correct as signals get routed later and these errors can be safely ignored at this point.

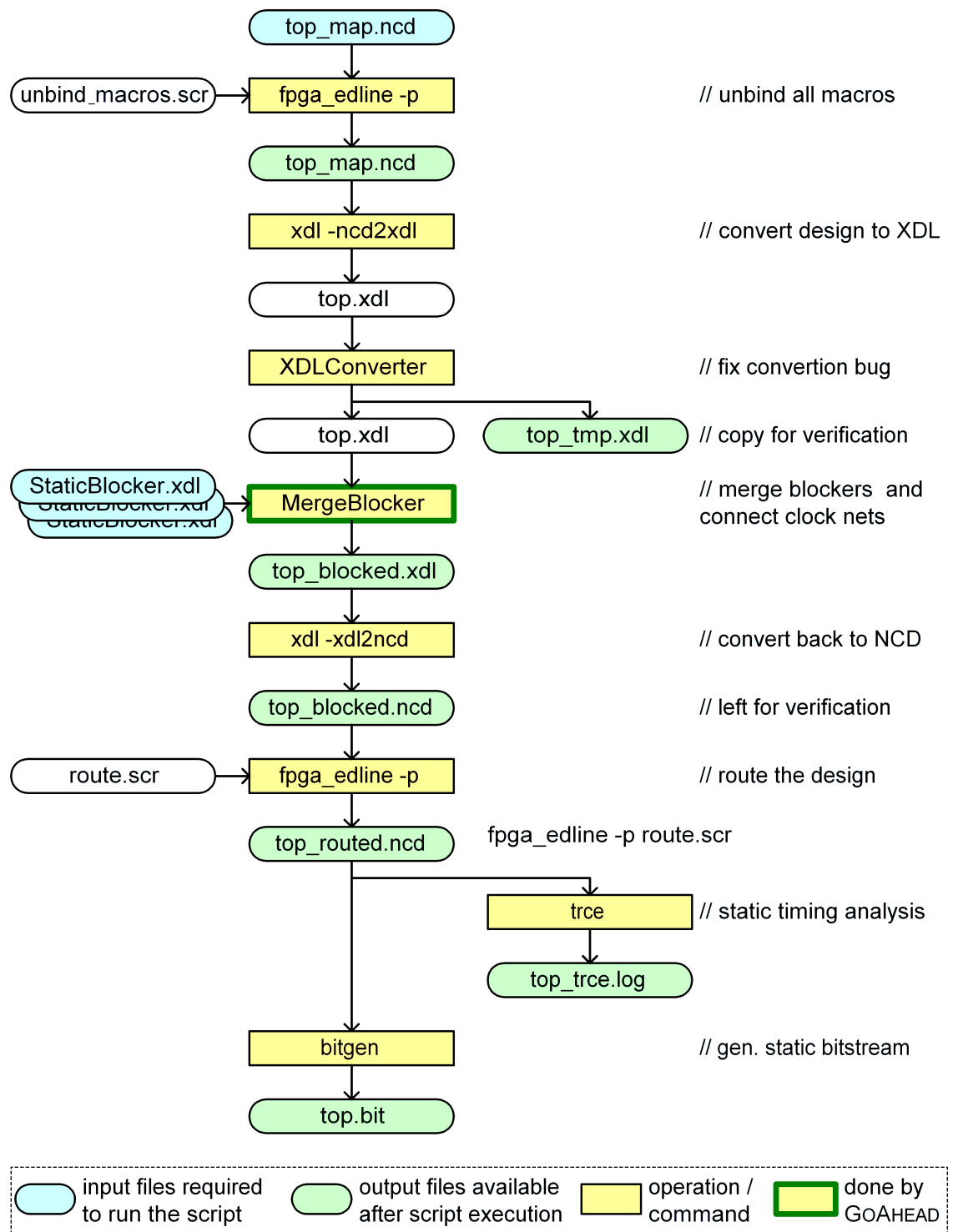


Figure 9: Static routing flow. The example is for `>static top StaticBlocker.xdl`.

3.3.3 XDLConverter

When converting a fully routed and functional design given as an NCD-file to XDL and back to NCD-format, some I/O pin settings get corrupted and prevent further usage of the NCD-file, including bitstream generation. This problem is fixed by the command line tool:

```
XDLConverter inputfile.xdl outputfile.xdl
```



TIP After converting the outputfile to NCD (`textttdxl -xdl2ncd outputfile.xdl outputfile.ncd`), a bitstream can be generated that would be identical to one generated from the original input NCD file. In this case, do not forget to provide the original physical constraints file (PCF) to `bitgen`. This file contains some specific configurations (e.g., for I/O pins) required for proper bitstream generation.

3.3.4 Merge Blocker

The static batch file generates a GOA script on the fly (`append_blocker.goa`) and executes it using the GOAHEAD command line mode. The script merges the blocker that has been generated in Section 3.2.4 into the present placed design. The blocker will occupy routing resources that are then prohibited to be used by the router. In addition, the blocker contains primitives, including LUTs, BRAMs, or DSP48 blocks. These primitives can be optionally connected to a global clock net. This ensures that a global clock network is activated inside a partial region. Because the name of a clock net may not be preserved, the clock net is specified by its dedicated BUFG driver instance (i.e. the clock signal driven by a BUFG.O pin). Note that GOAHEAD will automatically wildcard a prefix before the BUFGInstanceName. This is useful if the BUFG instance is instantiated in a sub-hierarchy.



ADVANCED Note that multiple blockers can be included and connected to different clock nets. This can be used to activate multiple clock nets within a reconfigurable region. For activating a clock network, a blocking primitive (e.g., a slice or a BRAM) connected to the corresponding clock net is required in the top and the bottom half of a clock region.

3.3.5 Routing the Design

After including all blockers, we convert the design back to the NCD format and route it in the FPGA editor. This is carried out on the command line with a script (`route.scr`) that is generated on-the-fly. In order to allow a nicer inspection of the result (`InputProject_routed.ncd`) in the FPGA editor, we delete all blocker nets.



ISSUE Writing back the NCD file within the FPGA-Editor script will throw several errors. These errors are caused by the blockers consisting of truncated signal paths that are, however, not used. Consequently, these errors can be safely ignored.

3.3.6 Timing Analysis

The batch files calls the Xilinx timing analyzer and writes the report in the file `InputProject_trce.log`. The achieved clock frequency is printed to the console.

3.3.7 Bitstream Generation

Finally, we generate the bitstream of the static design with:

```
bitgen InputProject_routed.ncd InputProject.bit -d -w -g Binary:Yes
```

This bitstream can be loaded to the FPGA board using Impact or the Digilent Adept tool.

3.4 static_constraints.goa

By copying GOAHEAD commands from the command trace into a text file, we can generate a GOA script file that can be executed within the GUI (`Commands -> Run Script`) or on the command line

(goahead static_constraints.goa). All GOAHEAD commands are human readable and easy to modify. The script that performs all actions of this tutorial is provided in the static project directory (note that the provided file includes some file definitions in the beginning):

```

1  # Open FPGA device description
2  LoadFPGA FileName=C:\MyPrograms\r22_xc6slx16csg324-3.binFPGA;
3  Reset;
4  # Define the Reconfigurable Region
5  ClearSelection;
6  AddToSelectionXY UpperLeftX=14 UpperLeftY=19 LowerRightX=52 LowerRightY=35;
7  ExpandSelection;
8  StoreCurrentSelectionAs UserSelectionType=PartialArea;
9  # Adding a Bus Macro
10 AddLibraryElementFromXDL XDLMacro=C:\WORK\GoAheadMacros\BM_S6_L4_R4_single.xdl↵
    ParseXDLPortStatements=True;
11 ClearSelection;
12 AddToSelectionXY UpperLeftX=12 UpperLeftY=21 LowerRightX=12 LowerRightY=21;
13 ExpandSelection;
14 AddSingleMacroInstantiationBySlice MacroName=BM_S6_L4_R4_single InstanceName=↵
    instBusMacro SliceName=SLICE_X5Y45;
15 # Generate VHDL Templates
16 PrintWrapperForMacro MacroName=BM_S6_L4_R4_single;
17 PrintVHDLMacroInstantiation PortMapping=RH:1,LH:1,LI:static_to_partial,LO:↵
    static_from_partial,RI:partial_to_static,RO:partial_from_static;
18 # Generate Placement Constraints
19 PrintLocationConstraintsForPlacedMacros;
20 ClearSelection;
21 SelectUserSelection UserSelectionType=PartialArea;
22 PrintProhibitStatementsForSelection ExcludeUsedSlices=True FileName= Append=↵
    True;
23 # Generate a Blocker Macro for constraining the routing
24 AddMacro MacroName=StaticBlocker;
25 SelectMacro MacroName=StaticBlocker;
26 BlockSelection SliceNumber=0 BlockArcsInRAMTiles=True BlockWithEndPips=False ↵
    AddDSPBRAMDriver=True DSPTemplate=C:\WORK\GoAheadMacros\DSP48_Blocker_S6.↵
    xdl BRAMTemplate=C:\WORK\GoAheadMacros\BRAM_Blocker_S6.xdl ↵
    PrintUnblockedPorts=False;
27 GenerateXDL FileName=C:\WORK\GoAhead_Tutorials\HelloWorld\static\StaticBlocker↵
    .xdl MacroNames=StaticBlocker IncludePorts=False IncludeDummyNets=True ↵
    IncludeModuleHeader=False IncludeModuleFooter=False;

```

As a third option, we can run the script in the GOAHEAD-script debugger. The debugger is located under Commands -> Single Step Script. See Figure 10 for a screen shot of the debugger with the opened textttstatic_constraints.goa script file.

As can be seen, the additional steps that have to be carried out among a static only implementation of a system can be accomplished by running two straight scripts: static_constraints.goa and static.bat. Note that the script contains absolute paths that have to be adjusted with respect to the local GOAHEAD installation.

3.5 Resulting Static System

As shown in the FPGA Editor screen shot in Figure 11, the result of the static implementation is a system that provides a reconfigurable island. This island provides access to the global c1k25 clock network. Furthermore, communication between the static and partial part of the system can be carried out with the help of a bus macro. For opening the routed design in the FPGA editor, go to your DOS box used to run static.bat and type fpga_editor top_routed.ncd.

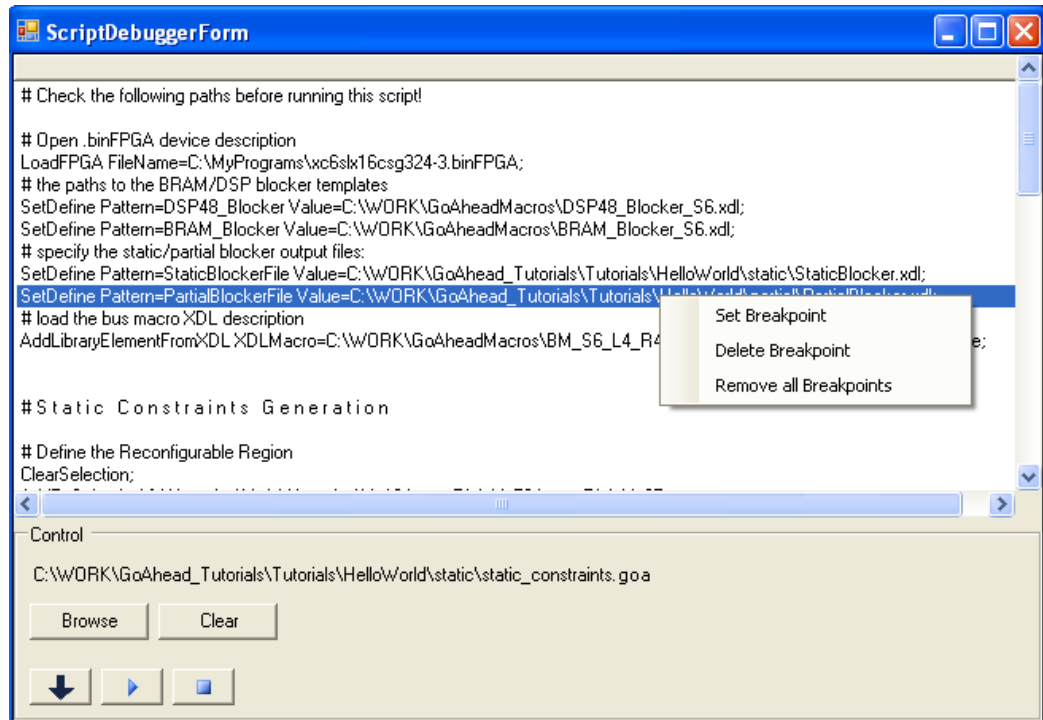


Figure 10: GOA script debugger: add a GOA script (button **Browse**), set or remove breakpoints in the context menu (right mouse button on a text line), and run the script until the next break point, execute the next command, or jump to the start of the script (with the three buttons in the bottom).

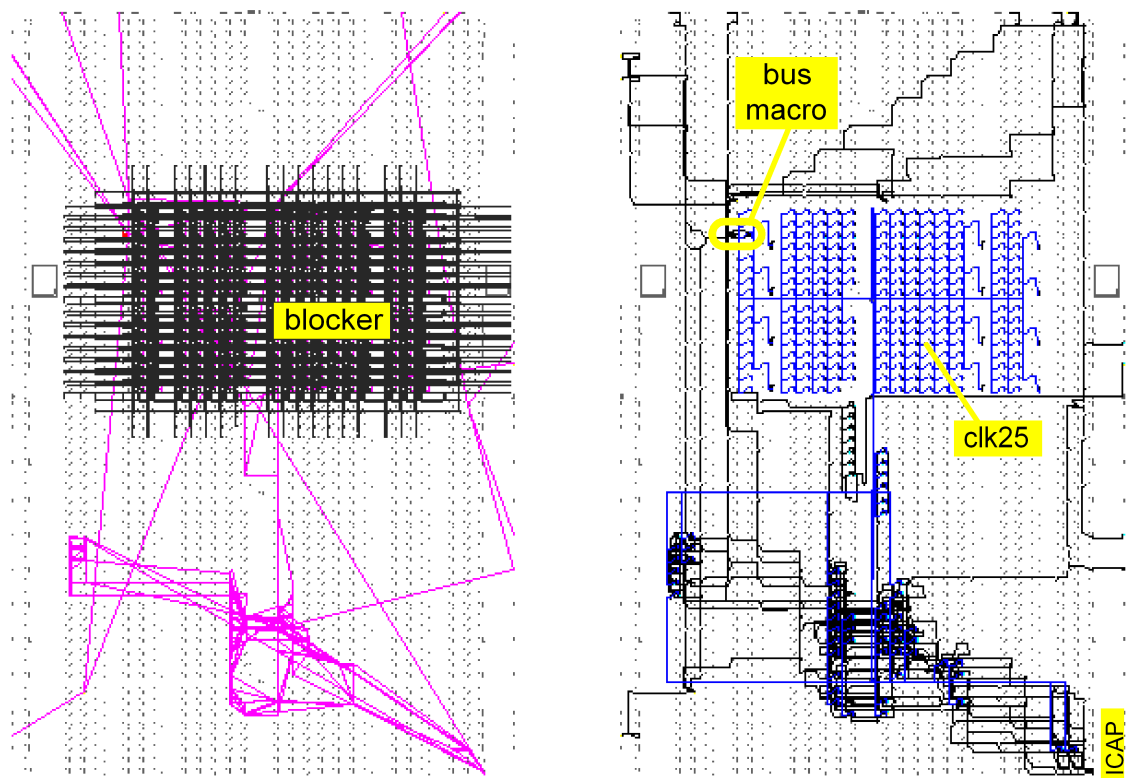


Figure 11: left: static design before routing, and right: after routing (with deleted blocker).

4 Implementing the Partial Modules

As depicted in Figure 2b), the partial modules will be implemented independently from the static system.

4.1 Creating an Empty Partial Module Project

For implementing the partial modules, we start with a new empty ISE project and select the same FPGA as has been used before for the static system. We name the project `partial`. After this, we add two new (empty) source files to the project (see also Figure 12):

`partial.vhd` : a VHDL module file containing all logic. As a partial module has typically only internal signals, the top level entity will only contain a dummy clock input pin (named "CLKin" in the example).

`partial.ucf` : an implementation constraints file

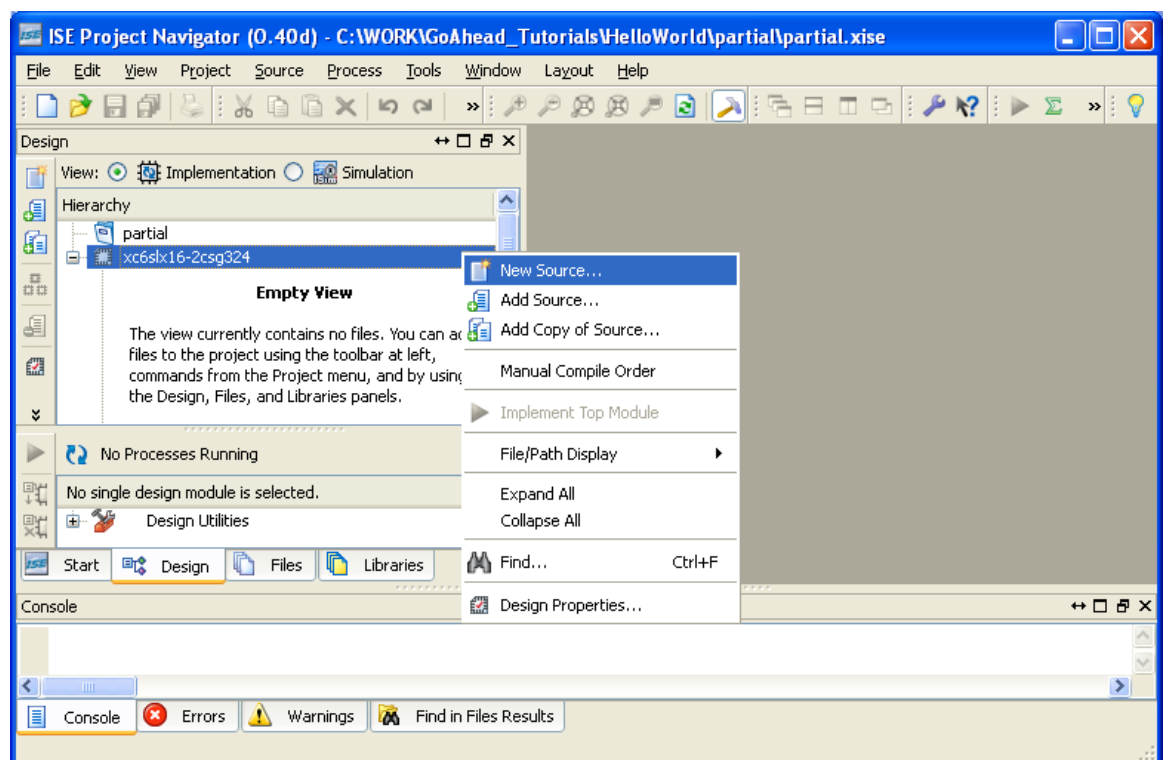


Figure 12: Adding `partial.vhd` and `partial.ucf` to an empty ISE project.

Next, we copy the bus macro NMC file (`BM_S6_I4_R4_single.nmc`) to the partial project directory. This file is located in the static project directory and in the macro library directory, which is provided with GOAHEAD.

After this, we copy from the static implementation file (`static.vhd`) the bus macro component declaration, the declaration of the bus macro connecting signals (e.g., `static_to_partial`), including the keep attribute statements, and the bus macro instantiation to our new partial implementation file (`partial.vhd`). In addition, we copy the component instantiation of the BUFG clock driver to our `partial.vhd` (the BUFG declaration is provided by Xilinx in the UNISIM library, search for UNISIM in `partial.vhd`). Here we keep the `clk25` output signal and change the input signal name to `CLKin`. `CLKin` has to be declared in the top level entity. By deactivating the "ADD I/O Buffers" synthesize

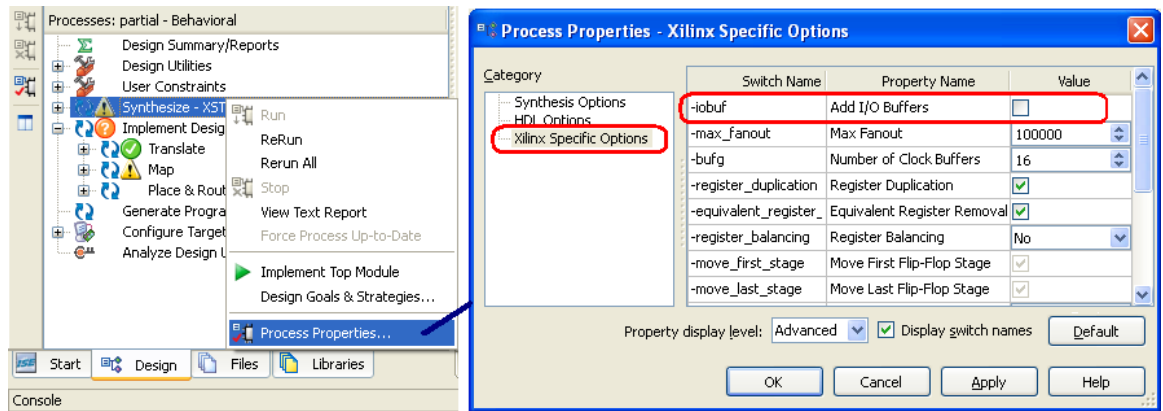


Figure 13: Synthesize options for the partial module implementation (available by clicking with the right mouse button on Synthesize - XST in the processes window).

option, as shown in Figure 13, no routing towards the BUFG primitive will be generated. With this option, an *internal primitive input* (here a BUFG input) will act as the top level input connection. This fits best the partial module design, because the routing to the BUFG is part of the static design.

As the final VHDL code, we implement the module logic according to Figure 2b) at the end of the file `static.vhd`. The solution directory provides a code example in `partial.vhd`. This file also shows how to use a generate statement in order to select between different partial modules. The final `partial.vhd` file should look as follows:

```

1  -- HELLO WORLD -- Partial
2
3
4  library IEEE, UNISIM;
5  use IEEE.STD_LOGIC_1164.ALL;
6  use UNISIM.VComponents.all;
7
8  entity partial is
9      Port ( CLKin : in  STD_LOGIC );
10 end partial;
11
12 architecture Behavioral of partial is
13
14 component BM_S6_L4_R4_single is
15     Port ( LH0 : in  STD_LOGIC;
16           LH1 : in  STD_LOGIC;
17           LH2 : in  STD_LOGIC;
18           LH3 : in  STD_LOGIC;
19           RH0 : in  STD_LOGIC;
20           RH1 : in  STD_LOGIC;
21           RH2 : in  STD_LOGIC;
22           RH3 : in  STD_LOGIC;
23           LI0 : in  STD_LOGIC;
24           LI1 : in  STD_LOGIC;
25           LI2 : in  STD_LOGIC;
26           LI3 : in  STD_LOGIC;
27           LO0 : out STD_LOGIC;
28           LO1 : out STD_LOGIC;
29           LO2 : out STD_LOGIC;
30           LO3 : out STD_LOGIC;
31           RI0 : in  STD_LOGIC;
32           RI1 : in  STD_LOGIC;

```

```

33         RI2 : in  STD_LOGIC;
34         RI3 : in  STD_LOGIC;
35         RO0 : out STD_LOGIC;
36         RO1 : out STD_LOGIC;
37         RO2 : out STD_LOGIC;
38         RO3 : out STD_LOGIC );
39 end component BM_S6_L4_R4_single;
40
41 attribute KEEP : string;
42
43 signal partial_from_static : std_logic_vector(3 downto 0) := (others=>'1');
44 signal partial_to_static : std_logic_vector(3 downto 0) := (others=>'1');
45 signal static_from_partial : std_logic_vector(3 downto 0) := (others=>'1');
46 signal static_to_partial : std_logic_vector(3 downto 0) := (others=>'1');
47
48 attribute keep of partial_from_static : signal is "true";
49 attribute keep of partial_to_static : signal is "true";
50 attribute keep of static_from_partial : signal is "true";
51 attribute keep of static_to_partial : signal is "true";
52
53 signal CLK25 : std_logic := '1';
54
55 begin
56
57 instBUFG25 : BUFG
58     port map ( I => CLKin,
59                O => CLK25 );
60
61 instBusMacro : BM_S6_L4_R4_single
62 Port Map (
63     RH0 => '1',
64     RH1 => '1',
65     RH2 => '1',
66     RH3 => '1',
67     LH0 => '1',
68     LH1 => '1',
69     LH2 => '1',
70     LH3 => '1',
71     RI0 => partial_to_static(0),
72     RI1 => partial_to_static(1),
73     RI2 => partial_to_static(2),
74     RI3 => partial_to_static(3),
75     RO0 => partial_from_static(0),
76     RO1 => partial_from_static(1),
77     RO2 => partial_from_static(2),
78     RO3 => partial_from_static(3),
79     LI0 => static_to_partial(0),
80     LI1 => static_to_partial(1),
81     LI2 => static_to_partial(2),
82     LI3 => static_to_partial(3),
83     LO0 => static_from_partial(0),
84     LO1 => static_from_partial(1),
85     LO2 => static_from_partial(2),
86     LO3 => static_from_partial(3) );
87
88 ----- the actual module logic: the OR/NOR Gate Module -----
89
90 partial_to_static(0) <= partial_from_static(0) OR partial_from_static(1) OR ↔
    partial_from_static(2) OR partial_from_static(3);

```

```

91 partial_to_static(2) <= NOT (partial_from_static(0) OR partial_from_static(1) ←
    OR partial_from_static(2) OR partial_from_static(3));
92
93 process (clk25)
94 begin
95     if clk25 'event and clk25='1' then
96         partial_to_static(1) <= partial_from_static(0) OR partial_from_static(1) ←
            OR partial_from_static(2) OR partial_from_static(3);
97         partial_to_static(3) <= NOT (partial_from_static(0) OR partial_from_static(1) ←
            (1) OR partial_from_static(2) OR partial_from_static(3));
98     end if;
99 end process;
100
101 end Behavioral;

```

Most of the lines that have been copied from the static implementation file have been generated by GOAHEAD throughout Section 3.1. Consequently, the partial module implementation can be carried out *before and independent to* the implementation of the static system. GOAHEAD will assist this by providing implementation and constraint templates. The main things that have to be done beforehand for both, the static system and the partial modules is the partial region allocation and the placement of the bus macros. For the latter one, it is required that the relative bus macro placement at the static to partial border is identical for all islands that will host the same set of partial modules at run-time.

The listing of `partial.vhd` gives the implementation of the OR/NOR module according to Figure 2b). Note that this time, we leave the two signals located in the static part of the system (`static_to_partial` and `static_from_partial`) unconnected. These signals will be set to '1' because of the initial value specification in the signal declarations. Again, only a logical '1' level can be created at any logic primitive input without extra implementation cost (in terms of LUTs).

4.2 Partial Module Implementation Constraints

After the VHDL implementation, we will add some implementation constraints to our `partial.ucf` file, as described in the following sections.

4.2.1 Clock Network Constraint and Bus Macro Placement

The first two entries can be copied from the static system (`static.ucf`):

```

1 INST "instBUFG25" LOC = "BUFGMUX_X2Y2"; # GCLK1
2 INST "instBusMacro" LOC = "SLICE_X5Y45"; # generated_by_GoAhead

```

The first line specifies the BUFG clock driver primitive of the FPGA fabric that will drive the `clk25` net that is connected to this primitive instantiation. By specifying the BUFG primitive, we basically constrain which global clock network will be used for `clk25`. Note that there are 16 global clock networks and corresponding BUFG primitives available on Spartan-6 FPGAs. All these networks are basically identical and the corresponding BUFG primitive names are listed in Table 1. The names are also available in GOAHEAD under File -> Device Info. However, we have to use a BUFG constrain in order to guarantee that the partial modules connect to the same global clock network that has been activated during the implementation of the static system (see Figure 11 for an example of an activated clock network).

The second line in the UCF file listing specifies the placement of the bus macro. For this tutorial, we keep the exact placement position of the macro. However, as demonstrated later in Section ??, it is sufficient to ensure that the relative placement position of a macro within its reconfigurable area is maintained.

clock net	Spartan-6 BUFG primitive	Virtex5 and Virtex-6	
		bottom	top
GCLK_0	BUFGMUX_X2Y1	BUFGCTRL_X0Y0	BUFGCTRL_X0Y16
GCLK_1	BUFGMUX_X2Y2	BUFGCTRL_X0Y1	BUFGCTRL_X0Y17
GCLK_2	BUFGMUX_X2Y3	BUFGCTRL_X0Y2	BUFGCTRL_X0Y18
GCLK_3	BUFGMUX_X2Y4	BUFGCTRL_X0Y3	BUFGCTRL_X0Y19
GCLK_4	BUFGMUX_X3Y5	BUFGCTRL_X0Y4	BUFGCTRL_X0Y20
GCLK_5	BUFGMUX_X3Y6	BUFGCTRL_X0Y5	BUFGCTRL_X0Y21
GCLK_6	BUFGMUX_X3Y7	BUFGCTRL_X0Y6	BUFGCTRL_X0Y22
GCLK_7	BUFGMUX_X3Y8	BUFGCTRL_X0Y7	BUFGCTRL_X0Y23
GCLK_8	BUFGMUX_X2Y9	BUFGCTRL_X0Y8	BUFGCTRL_X0Y24
GCLK_9	BUFGMUX_X2Y10	BUFGCTRL_X0Y9	BUFGCTRL_X0Y25
GCLK_10	BUFGMUX_X2Y11	BUFGCTRL_X0Y10	BUFGCTRL_X0Y26
GCLK_11	BUFGMUX_X2Y12	BUFGCTRL_X0Y11	BUFGCTRL_X0Y27
GCLK_12	BUFGMUX_X3Y13	BUFGCTRL_X0Y12	BUFGCTRL_X0Y28
GCLK_13	BUFGMUX_X3Y14	BUFGCTRL_X0Y13	BUFGCTRL_X0Y29
GCLK_14	BUFGMUX_X3Y15	BUFGCTRL_X0Y14	BUFGCTRL_X0Y30
GCLK_15	BUFGMUX_X3Y16	BUFGCTRL_X0Y15	BUFGCTRL_X0Y31

Table 1: BUFG clock driver primitive names for Spartan-6, Virtex-5 and Virtex-6 FPGAs.

4.2.2 Generating the Physical Partial Place & Route Constraints

For our Hello World module, these constraints would be sufficient in most cases. However, to ensure correct physical implementation of more complex modules, GOAHEAD helps us to generate prohibit constraints and blocker macros. However, as opposed to the prohibit constraints used for the static system, we have now to prohibit everything around the reconfigurable island. This is required to prevent the partial modules from taking resources that might be used by the static part of the system.

We go now back to GOAHEAD and select the PartialArea (as has been done in Section 3.2.4). If the tool got closed meantime, use your GOA script to define the reconfigurable island (which we called PartialArea). Then, we open the UCF output window tab and clear it (with the mouse context menu Clear Window). Next, we invert the selection via the menu with Edit -> Invert Selection (see also Figure 14).

After this, we generate the prohibit statements with UCF -> Prohibit. Again, we will exclude the used slices from getting prohibited. Note that this step requires a bus macro being placed in GOAHEAD (on order leave the internal macro routing unblocked). We append the generated prohibit statements to partial.ucf. This can be done by copy and paste from the UCF output window or by selecting the partial UCF file (ensure that Append is activated).

With the same selection, we continue by generating a blocker macro. Therefore, we add a new macro in the Macro Manager (located under the Macro menu). We give this macro the name PartialBlocker. After this, we open the blocker dialog with Commands -> Block Selection. We use the same settings as being used before for generating the static blocker (see Figure 7c) for an example). Remember to specify the DSP and BRAM template files (DSP48_Blocker_S6.xdl and BRAM_Blocker_S6.xdl). The blocker generation can take a few minutes; but however, this process can be accelerated by defining a blocker fence around the reconfigurable island rather than by blocking the whole remaining FPGA. Such a fence should be at least four tiles wide or height when using Spartan-6 FPGAs. The GOAHEAD commands for the partial modules are appended to static_constraints.goa (in the static solution directory). There is an additional file partial_constraints.goa (in the partial solution directory) that gives an example of how to define a fence.

As the last step in GOAHEAD, we store the PartialBlocker in the project directory of our partial system via the menu with Macro -> Generate XDL Code. We give the blocker the filename PartialBlocker.xdl. The partial system constraint generation produced additional Command Trace entries that we should append to our GOA script:

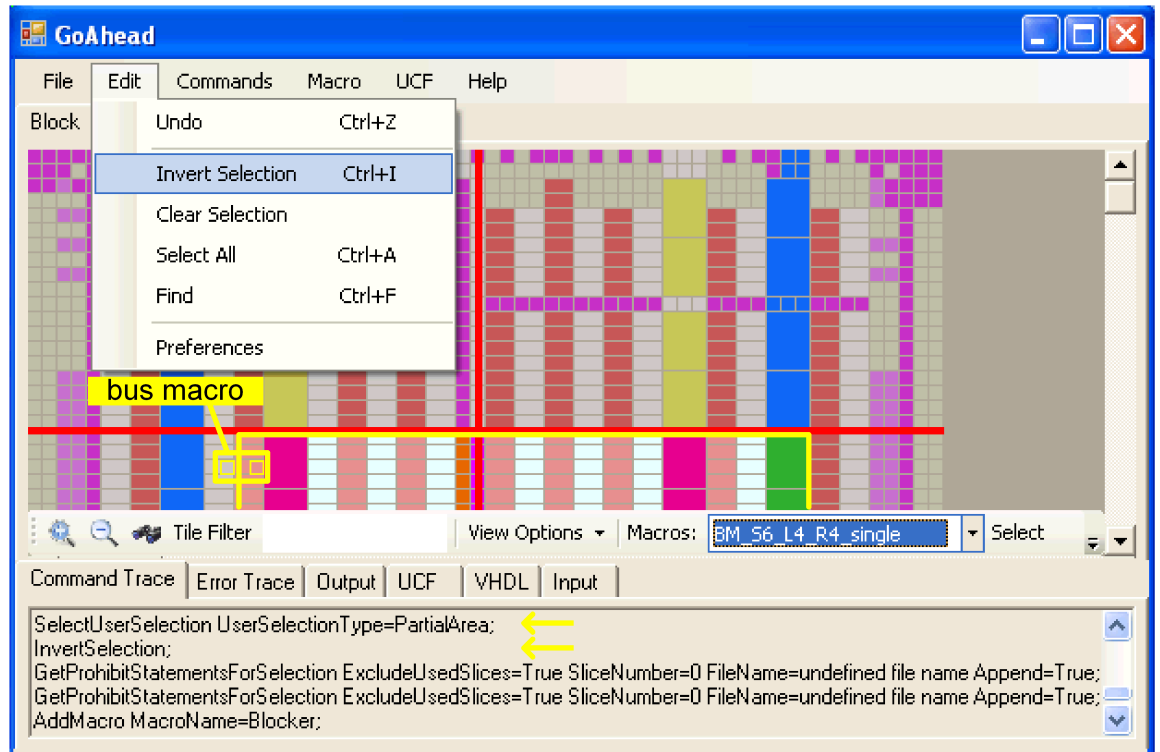


Figure 14: Inverting the selection in GOAHEAD to generate prohibit statements and a blocker macro around the reconfigurable region.

```

28 # Generate Prohibit Constraints for the Partial Modules
29 ClearSelection;
30 SelectUserSelection UserSelectionType=PartialArea;
31 InvertSelection;
32 PrintProhibitStatementsForSelection ExcludeUsedSlices=True FileName= Append=↵
   True;
33 # Generate the Partial Blocker
34 AddMacro MacroName=PartialBlocker;
35 SelectMacro MacroName=PartialBlocker;
36 BlockSelection SliceNumber=0 BlockArcsInRAMTiles=True BlockWithEndPips=False ↵
   AddDSPBRAMDriver=True DSPTemplate=C:\WORK\GoAheadMacros\DSP48_Blocker_S6.↵
   xdl BRAMTemplate=C:\WORK\GoAheadMacros\BRAM_Blocker_S6.xdl ↵
   PrintUnblockedPorts=False;
37 GenerateXDL FileName=C:\WORK\GoAhead_Tutorials\HelloWorld\partial\↵
   PartialBlocker.xdl MacroNames=PartialBlocker IncludePorts=False ↵
   IncludeDummyNets=True IncludeModuleHeader=False IncludeModuleFooter=False;

```

4.2.3 Blocking the Center Clock Column

On Spartan-6 FPGAs, global clock resources (BUFG, DCM, or PLL primitives) share their configuration data with logic primitives. To be more precise, in the center CLB column, some CLBs are replaced with global clock primitives. This has not only to be considered when relocating modules where the resource footprint of the modules must match the resource footprint of the FPGA primitives, this impacts the global clock configuration in general. For a sake of simplicity, we will not use the center CLB column for the partial module implementation at this point. However, GOAHEAD

allows to use the center CLB column as covered in Section ??.

To prohibit the center CLB column of the partial region, we simply select this column, as shown in Figure ??. Then, we generate the placement prohibit constraints with UCF -> Prohibit and append the result to `partial.ucf`. Next, we go to the Macro Manager and add (and select) a further macro with the name `ClockBlocker`. Then we generate the blocker with Commands -> Block Selection Finally, we generate the XDL code that will prohibit the wires of the clock column with Macro -> Generate XDL Code. We give this blocker the name `ClockBlocker.xdl`. The

Instead of generating an additional blocker, we could have added the clock column blocker directly into the partial blocker.

4.3 Routing and Partial Bitstream Generation

With the prohibit and the placement constraints for a particular partial module, we run the logic synthesis and placement (identical to what has been done during the implementation of the static system). Consequently, we run the implementation until map has finished.

The placed netlist has then to be converted to XDL where we add the `PartialBlocker`, the result has to be converted back to the NCD format before routing the partial module in the FPGA editor. These steps are carried out by the batch script `partial.bat` that is very similar to `static.bat`.

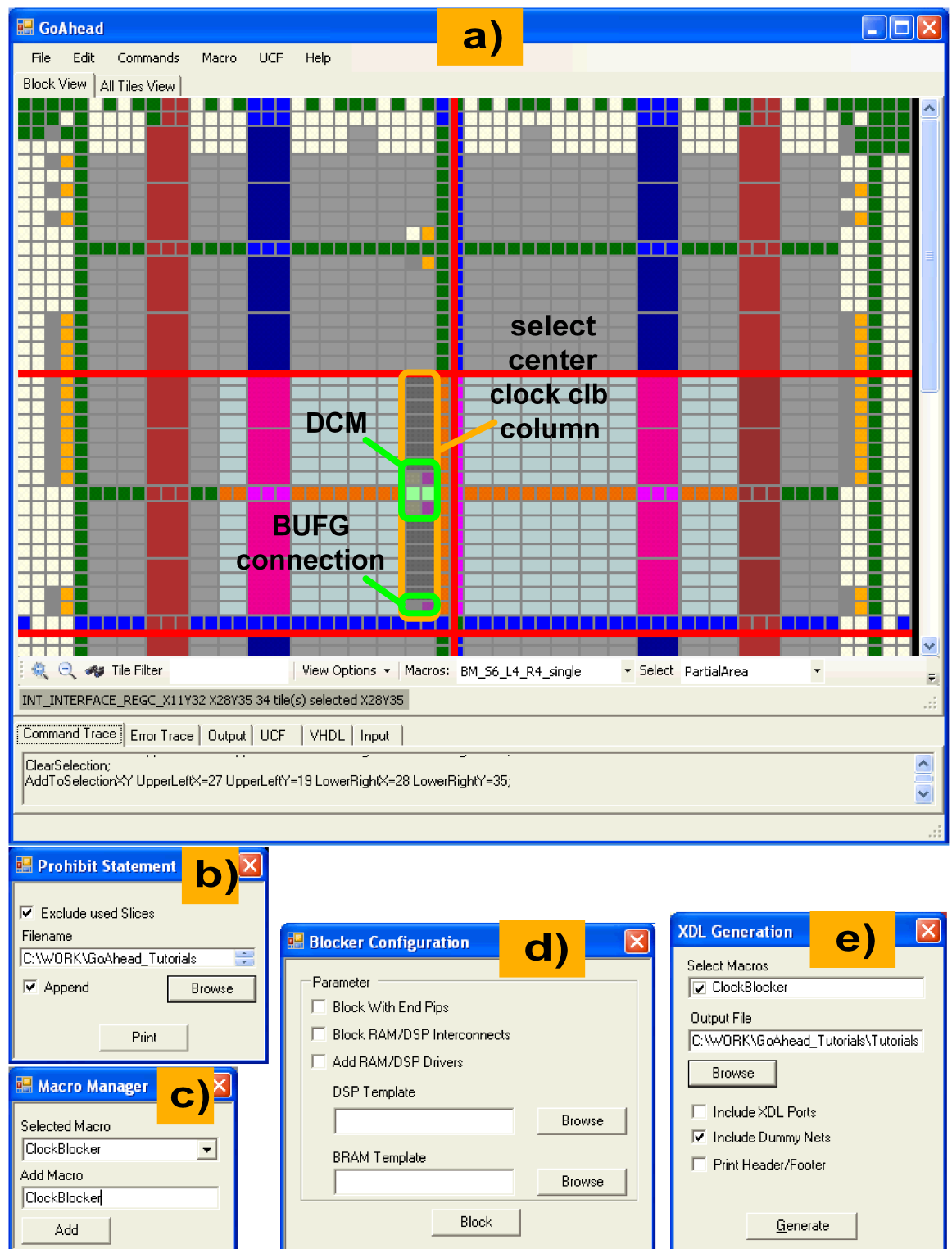


Figure 15: Clock Blocker Generation. a) selection, b) UCF Prohibit generation, c) adding the ClockBlocker in the Macro Manager, d) Blocker generation (**leave all fields and options empty**), e) generation of the XDL output file (ClockBlocker.xdl).

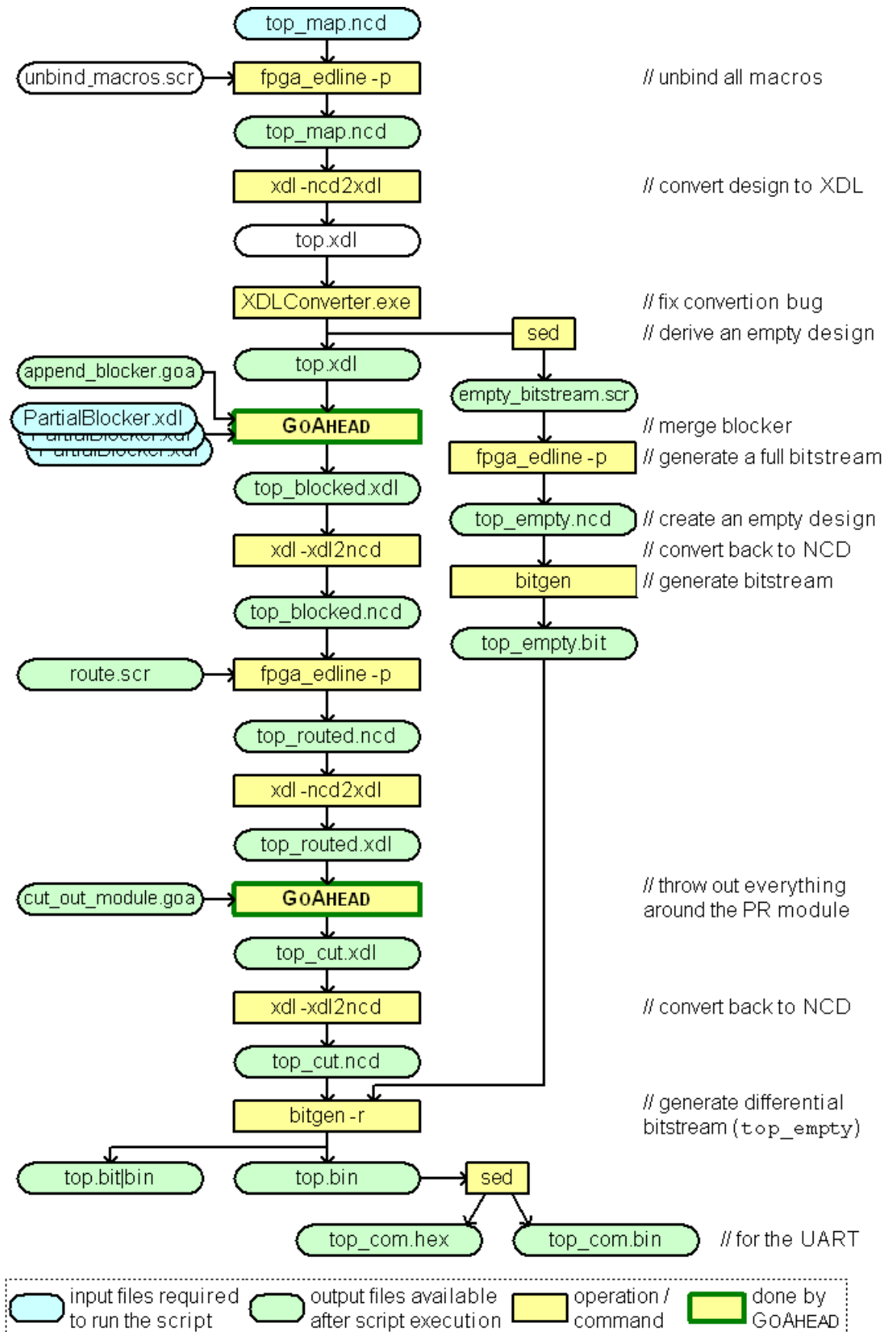


Figure 16: Partial routing and bitstream generation flow. The example is for >partial top PartialBlocker.xdl.

The script `partial.bat` is visualized in Figure 16. Note that the partial blocker macro is typically not required to be connected to any clock network. This might be required when implementing hierarchical reconfigurable modules (i.e. a PR module containing itself a PR module).

The routing script for the partial module implementation is called as follows:

```
>partial InputProject [Blocker_XDL_FILE BUFGInstanceName] [Blocker_XDL_FILE ] ...
```

The options are identical to the ones used for `static.bat`:

InputProject:	this is typically the prefix of the <code>MyProject_map.ncd</code> file (MyProject). Note to specify the prefix without the <code>_map.ncd</code> at the end.
Blocker_XDL_FILE:	a blocker file generated by GOAHEAD to constrain the routing
BUFGInstanceName:	an optional instance name used to connect blocker primitives to a global clock net (see Section 3.3.4 for more documentation)

In order to generate the partial module bitfile, the script generates an empty design (i.e. a design without any logic and routing resources). After this, we create a configuration bitfile for this empty design. Then we take the routed module, convert it to XDL and use GOAHEAD to cut out all logic and routing belonging to the reconfigurable module. For example, the partial design contains a BUFG clock driver, that will however, not be part of the partial module (and cut away) because the clock infrastructure is already implemented in the static part of the system. For the module cutting process, we have to set two variables that are located in the beginning of `partial.bat`. They define the device description of the FPGA that was used to implement the partial module and to specify the region of the partial module, like for example:

```
SET BinFPGA_Device_Description=C:\MyPrograms\xc6slx16csg324-3.binFPGA
SET PartialArea=UpperLeftX=14 UpperLeftY=19 LowerRightX=52 LowerRightY=35
```

Note that both settings can be copied from the GOAHEAD command trace.

The remaining partial module is converted back to NCD in order to generate the partial module bitfile using the `bitgen -r` option:

```
bitgen %1_cut.ncd -d -w -g ActiveReconfig:Yes -g CRC:Disable
-g binary:Yes -r %1_empty.bit
```

Remember that partial reconfiguration on Spartan-6 FPGAs has to be done using ICAP. Hence, the option `-g ActiveReconfig:Yes` is mandatory in order to permit the static part of the system to write internally to the ICAP port.

As the last action of the partial routing and bitstream generation process, the script will generate two configuration files that are compatible with the provided UART in the static design. The files support either faster binary transmission mode (`yourdesign_com.bin`) or the more compatible ASCII hex mode (`yourdesign_com.bin`). They can be sent to the board using simple copy commands, as described in more detail in Section 3.1 on Page 9. Note that all these configuration files get overwritten when starting the script (e.g., for implementing a different partial module).



TIP There are several more possibilities to generate partial bitfiles including to perform to cut out the module on the bitstream level or to use GOAHEAD to delete selectively the partial module in order to produce the reference bitstream for the `bitgen -r` option.



ISSUE Adding blockers and cutting modules will throw some error messages by the FPGA editor that we can ignore. Note that GOAHEAD can link modules into a design such that the final netlist passes all design rule checks.

4.4 Important Spartan-6 Specific Topics

The partial Hello World example should work fine, but however, there are a few more issues that have to be followed to implement more complex partially reconfigurable modules.

4.4.1 Spartan-6 Clock Resources

We have already learned that by specifying a BUFG placement constraint for a clock driver primitive, we are able to bind a clock signal to a specific global clock tree. However, as compared to previous Virtex FPGA architectures from Xilinx, clock resources are now embedded directly into the logic fabric rather than being separated from the fabric.

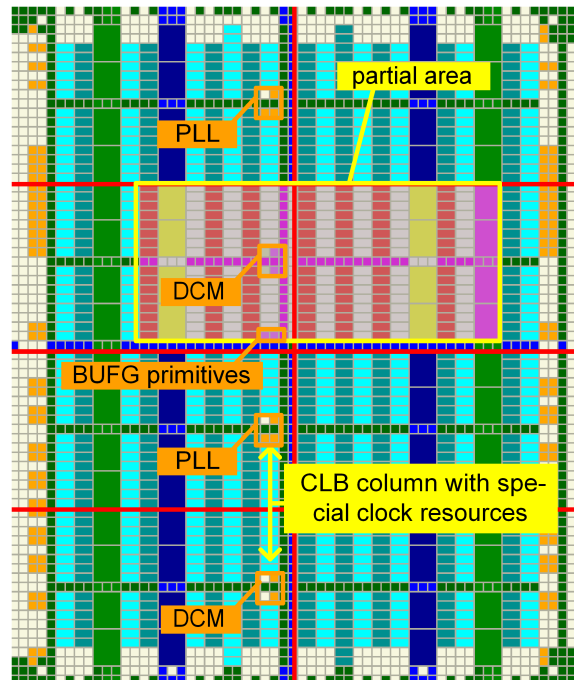


Figure 17: Clock resources of a Spartan-6 XC6SLX16 FPGA.

As shown in Figure 17, various resources dedicated to clock generation and distribution are obstacles in the otherwise homogeneous FPGA fabric. The obstacles are not only visible in GOAHEAD or the FPGA editor but are also present in the configuration bitstream. This has impact on the module placement as the resource footprint has to match the resource footprint of the partial modules and it has impact on the partial bitstream generation.

In the easiest case, the CLB column providing the dedicated clock resources might be left to the static design by creating adjusted prohibit statements and blocker macros. Remember that it is possible to select another (not necessary connected) region to a selection, by holding down the <CTRL> key for the second selection. Note that in this case it is still possible to use resources located left and right beside this column for the same module as there exist routing resources that can hop over this column. Then, it might be required to add further implementation constraints to the UCF files in order to prevent that too many signals cross the forbidden CLB column. Or in other words, it might be required to partition a module into two parts such that the routing between the two partitions is feasible. This is supported in GOAHEAD via the command `UCF -> AREA`. The result will be printed into the UCF output text window.

As a (more advanced) alternative, we can use the logic resources of the CLB column containing the special clock resources also for implementing partial modules. In this case, we have to consider

that the BUFG, DCM and PLL primitives are in almost all cases instantiated in the static design. Therefore, we have to use the configuration that corresponds to the static design and not to the partial design for the clock primitives when generating partial bitstreams. For example, in our static Hello World design, we used two BUFG instances, one for the input clock that was divided and one for the entire 25 MHz system clock. However, the partial system instantiates only the BUFG driver for the 25 MHz system clock. Consequently, when generating a partial module bitstream following the flow described in Section 4, the system will hang up during run-time reconfiguration as there is no input clock BUFG instance in the partial design. Note that there are cases where a partial module has no I/O pin connections at all.

GOAHEAD can extract the BUFG configuration from the static system and merge it into a partial module if necessary. The following GOA script cuts out the BUFG tile from the static design:

```
1 LoadFPGA FileName=C:\MyPrograms\xc6slx16csg324-3.binFPGA;
2 AddToSelectionXY UpperLeftX=30 UpperLeftY=36 LowerRightX=30 LowerRightY=36;
3 CutOffFromDesign XDLInFile=top_routed.xdl XDLOutFile=top_clk_cut.xdl ←
   KeepSelection=True RemoveModules=True;
```

The selection of the BUFG tile can be determined using the Tile Filter in the GOAHEAD GUI. As depicted in Figure 18, only the BUFG tile will be displayed if we filter for the prefix CLKC. We can click on that tile and can then copy the `AddToSelectionXY` command from the Command Trace window to the GOA script.

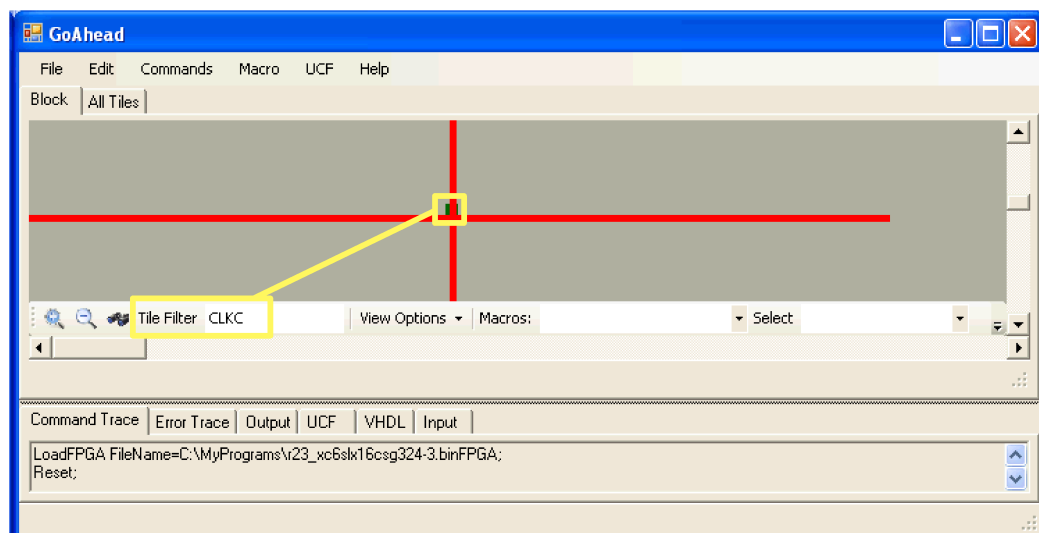


Figure 18: Identifying the tile containing the BUFG primitives using the tile filter.

So far, we merged the extracted tile manually, a command for this will come shortly.



ISSUE Xilinx has not released documentation about how PLLs or DCMs behave during partial reconfiguration. Despite that we have been successful in reconfiguring CLB columns containing these primitives, there might be unknown sideeffects. Therefore, we tried not to use PLLs and DCMs inside reconfigurable islands (e.g., by using prohibit statements) or used them for less critical parts. For example, temporal clock glitches supplied to a video output module might be acceptable but the same would be crucial for a DDR memory controller clock.



ADVANCED The dedicated clock resources are obstacles in the otherwise more homogeneous FPGA fabric and have to be considered in the case partial modules should be relocated to different regions of a static system or migrated among different static systems. However, as the local interconnect routing of the clock resource tiles is identical to the routing of the logic, BRAM, and DSP48 tiles, it is directly possible to relocate modules from an island containing clock resources to another island. Note that this implies that no further local interconnect to be used by the dedicated clock resources. If local routing is required, follow the instructions in Section 4.5.3. The opposite (i.e., relocating a module to an island containing clock tiles) is possible if prohibit constraints are generated for tiles that might have to be placed on a clock tile.

4.5 Further Topics

The Hello World tutorial is intended to teach how GOAHEAD can be used together with the Xilinx tools to easily implement reconfigurable systems. The following sections cover more advanced topics that are for users that have understood the basic principles in the design of reconfigurable systems.

4.5.1 Timing Constraints

By setting the clock frequency to a conservative low value of 25 MHz in our static test design, we don't have to worry about any latency in the design. However, in general, we want to achieve a much higher clock speed which then requires the definition of proper timing constraints.

As a general rule, it is recommended to place registers at all input and output signals of both the static system and the reconfigurable modules. If handshaking is required for backpressure flow control (i.e. a module or pipeline stage can signal a previous stage to stall operation), small FIFOs or registers can be used to prevent idle bubbles due to the handshaking.

With the TPSYNC timing constraint, we can assign the available time slack arbitrarily to the static system or to the partial modules. In other words we can constrain how much time is allowed for a signal (and the corresponding combinatory path) to be spent in the static part of the system and how much for implementing a partial module. With this constraint we can provide a full interface specification that includes the binding of interface signals to wires as well as the corresponding timing.

4.5.2 Multiple Islands and Module Relocation

4.5.3 Routing Optimizations

See also our paper "Routing optimizations for component-based system design and partial run-time reconfiguration on FPGAs" During the static system implementation, we used blocking to ensure that no static routing used resources within the reconfigurable region. Consequently, the reconfigurable region might become a significant obstacle that could negatively impact timing or even result in unroutable situations. This is in particular to be considered, when implementing reconfigurable systems using Spartan-6 FPGAs. These devices have no long lines (i.e. wires spanning the full height or width of the fabric) for hopping over a reconfigurable region. These routing resources are only available in Virtex series FPGAs and for crossing a reconfigurable region, we would have to route paths around it.

To overcome this issue, we can drill tunnels into the blocker (i.e., selectively allowing wires by not blocking them). We only have to ensure, that these tunnels do not interfere with other tunnels used for the communication, as described in Section `refchap:ZeroLogicOverheadIntegration`. For horizontal tunnels, this means that we do not use the same CLB columns for the partial-module communication and for crossing the reconfigurable region. Tunnels are generated by selecting the possible CLB rows within the partial region and by applying the command `MarkPortsInSelectionAsUsed PortName=` before blocking or the command `RemoveArcs PortName=` after blocking. We could then, release for example all or a subset of horizontal quad-lines by specifying `PortName=WW4B1` and `PortName=EE4B` in two commands. Note that for narrow modules (that are not more than four CLB columns wide) this has no impact on the module routing as these resources would be left unused elsewhere.

While the selective tunnel drilling allows static routing to cross the partial area, we have to ensure that partial modules 1) do not use resources that are allocated to static routing and 2) that the partial modules contain the routing information for the static fraction (all the way down to the partial bitstreams). Both issues are solved with the `AddArc` command (e.g., `AddArc From=WW4E1 To=WW4B1;`). With this command, we block the involved wire resources, and by leaving them in the final module, we will speculatively continue the routing path regardless if it will be used by the static system or not. in the

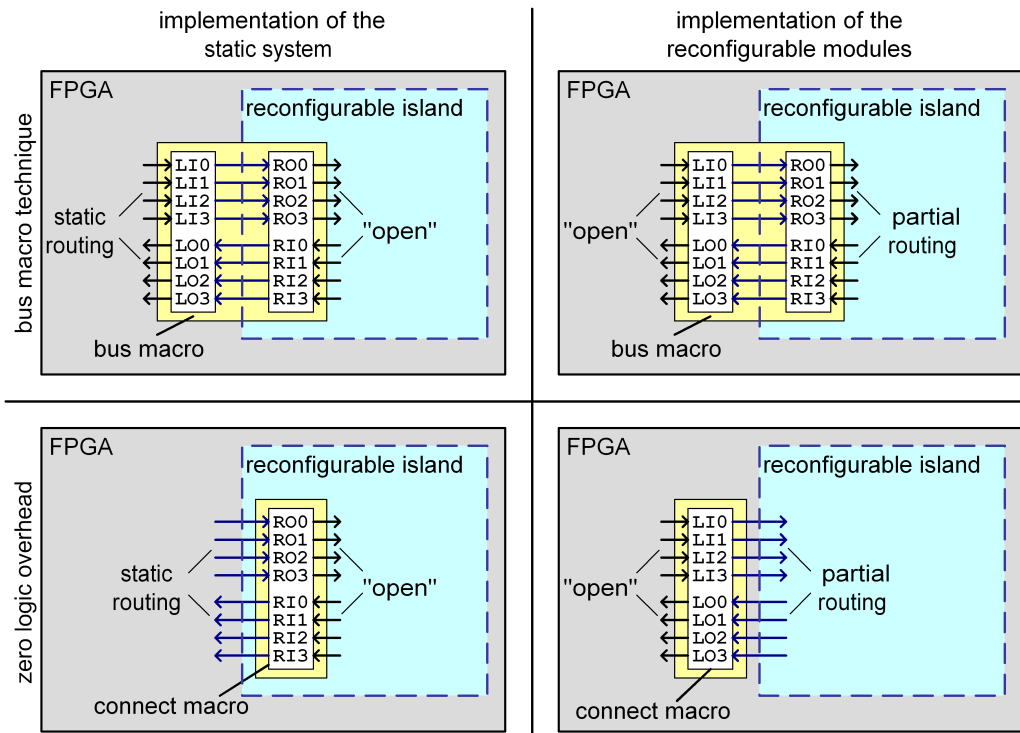


Figure 19: Comparison of the bus macro approach with the zero overhead technique.

4.5.4 Zero Logic Overhead Integration (no Bus Macro)

In the Hello World system, we used *bus macros* for routing the signals that cross between the reconfigurable modules and the static system. However, bus macros can comprise a significant logic overhead that might in some examples exceed the logic implementation cost of a reconfigurable module itself. Furthermore, bus macros add additional latency which might impact performance.

Note that this drawback is still existent in the more recent Xilinx design flow that uses *proxy logic*. The idea of that design flow is to place some LUTs into the reconfigurable region that act as anchor points (called proxy logic) for signals that cross between the static and the partial part. During the implementation of the static system, the routing to these anchor points will be generated. Based on that initial static routing step, incremental place and route will then be used for each reconfigurable module individually. However, this flow has several drawbacks. For example modules cannot be relocated from one island to another one because the proxy routing is typically different in all islands. Another problem is that all reconfigurable modules have to be rerouted (for all module/placement permutations) if the static system changes. This is because the routing to the proxy logic cannot be safely preserved.

The drawbacks of the bus macro and proxy logic approaches can be prevented using the GoAhead *zero logic overhead integration* flow. The idea of this technique is derived from the bus macro approach. As shown in Figure 19 for the bus macro approach, only one half of a bus macro is actually used during an implementation step (either when implementing the static system or a reconfigurable module). As an alternative, a *connection macro* can be instantiated as a placeholder for the entire missing part of the system during an implementation step. As shown in the bottom half of figure 19, the partial system is substituted by a connection macro placeholder during the static system implementation. Respectively, the static system is replaced by connection macro placeholder when implementing a reconfigurable module.

At first glance, this seems to be identical to the recent proxy logic approach from Xilinx. However,

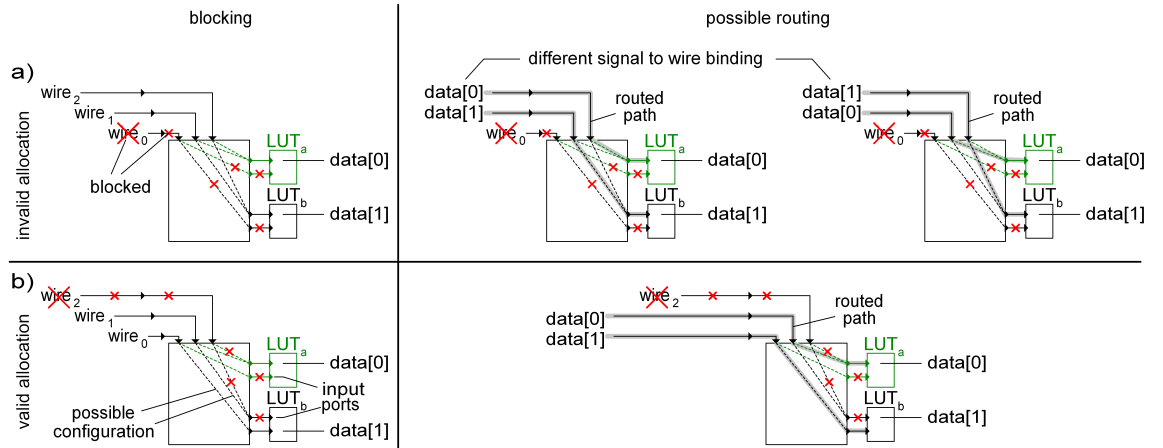


Figure 20: Wire allocation for the zero overhead integration technique. a) allocating $wire_1$ and $wire_2$ results in an unpredictable routing, while b) allocating $wire_0$ and $wire_1$ results in a unique routing.

the trick is that GOAHEAD can generate blockers that leave only one dedicated routing path to and from the connection macro. Consequently this binds a signal to a definable wire that crosses between the static system and the reconfigurable modules. This means that, for example, we can exactly constrain which wire will be used during the static implementation by generating a blocker that leaves only one possibility to reach the connection macro. In addition, GOAHEAD can create a compatible blocker to be used for the module implementation that releases exactly the same set of wires the cross the reconfigurable island.

This technique has to consider some rules. First of all, by blocking, we can only select the set of wires that are allowed for routing (i.e. wire allocation) but this does not necessary ensure a particular binding of a logical signal to a physical wire. However, the binding is achieved by allocating wires such that *only one unique routing path* can be used to reach the connection macro (see Figure 20). As a consequence, not all wires within a CLB can be used at the same time to implement the routing between the static part and the partial part of a system. However, the GOAHEAD package includes a special designed connection macro (`Connect4_S6_double`) that can connect up to four input and four output signals (using horizontal double line wires) at the same time. Hence, the connection macro can connect the same number of signals than the bus macro. Note that the wires can be arbitrarily selected for a connection at the left or right border of a module. Moreover, with the provided connection macro, the interfaces can be arranged in such a way that multiple reconfigurable modules can be dasychained together, as described in Section 4.5.6.

Blocking design flow and corner cases

4.5.5 Bus Macro Placement and Wire Allocation

So far, we have not discussed *how many* wires should be connected per CLB (configurable logic block) and how to select *where* the wires should be connected. To answer this question, we have to look closer at the Spartan-6 architecture. Each CLB comprises two slices with four 6-input look-up tables (LUTs) each. Each LUT can alternatively be split into two 5-input LUTs when inputs are shared (which is used for the provided bus macros and connection macros) Furthermore, there exists carry chain logic in each CLB that chains four of the LUTs within a CLB before continuing the chain in the next CLB above. This means that the carry chain traverses in upwards direction with four LUTs per CLB. This logic is used to implement fast carry ripple operations (e.g., for adders or counters).

With this information, we know that in most cases it is best to connect four signals per CLB. This is considered within the provided bus macros and connection macros by allowing the connection

of four input signals and four output signals. In addition, the macros should be placed such that the LSBs of a signal vector are located in the bottom of a reconfigurable module and the MSBs respectively in the bottom. This helps to reduce vertical routing overhead within the module. Note that the permutations of wires within a CLB have only minor impact on the routing as there exists at least one connection from each local wire to each look-up table.

In Section ??, we will discuss a more advanced system where we connect two times four input signals together with four output signals within the CLBs. This will be used to connect the signals of a reconfigurable instruction set extension.

4.5.6 I/O Bars for Slot-based Module Placement

4.5.7 Partial Bitstream Generation

In previous versions of the Xilinx bitstream generation tools, there was an option for generating a partial bitstream for a definable region of the FPGA fabric (a bounding box). This option is unfortunately not available for recent FPGA families. However, there is an option to generate differential bitstreams using the `bitgen -r` switch. The command is called as follows:

```
>bitgen -d -w -g ActiveReconfig:Yes -g CRC:Disable to_file.ncd -r from_file.bit
    partial_diff.bit
```

This command will create a partial bitstream that contains the differences to switch from the `from_file` to the `to_file`. The `to_file` is always in the NCD netlist format and the `from_file` is always a bitstream (*.bit). Consequently, before running this command, `bitgen` has to be called to generate the `from_file`. The used `bitgen` options are:

```
-d:    do not run the design rule check DRC
-w:    overwrite existing output files
-g ActiveReconfig:Yes: keep FPGA running during reconfiguration
-g CRC:Disable:    do not check the CRC after configuration
```

In most of the cases, we could use the CRC. As illustrated in Figure 21, there are several ways to generate partial bitstreams using `bitgen -r`. Note that by changing the `from_file` and the `to_file`, we will generate a blanking bitstream that removes the partial module from the system (if loaded to the FPGA). It is recommended to use module blanking rather than overwriting one module directly by another one. This reduces power consumption during reconfiguration. For more details see our paper "Short-Circuits on FPGAs caused by Partial Runtime Reconfiguration". Blanking can be done quickly using compressed bitstreams. In these bitstreams, identical frames are automatically duplicated by the FPGA configuration logic.

In the `bitgen` call options, we specified not to use the design rule check DRC (-d). However, this is only required when cutting out modules using the methods illustrated in Figure 21b) and Figure 21c). In that examples, there will be input and output antennas (i.e., signals that have either only an input or only a driver, but not both). These antennas are created by cutting out the module and consequently by cutting the signal paths crossing the static to partial border (for example the internal wires of a bus macro). Consequently, the DRC can be ignored. As an alternative, GOAHEAD can create netlists at any configuration state by merging modules on the netlist level into a static system. In other words, GOAHEAD can cut out modules directly from a design and store them as partial netlists (using the GOAHEAD command `CutOffFromDesign`). These netlists can be used to create incremental systems such that the bitstream generation method from Figure 21a) can be used. In this case, partial bitstreams can be generated that pass the `bitgen` DRC check.

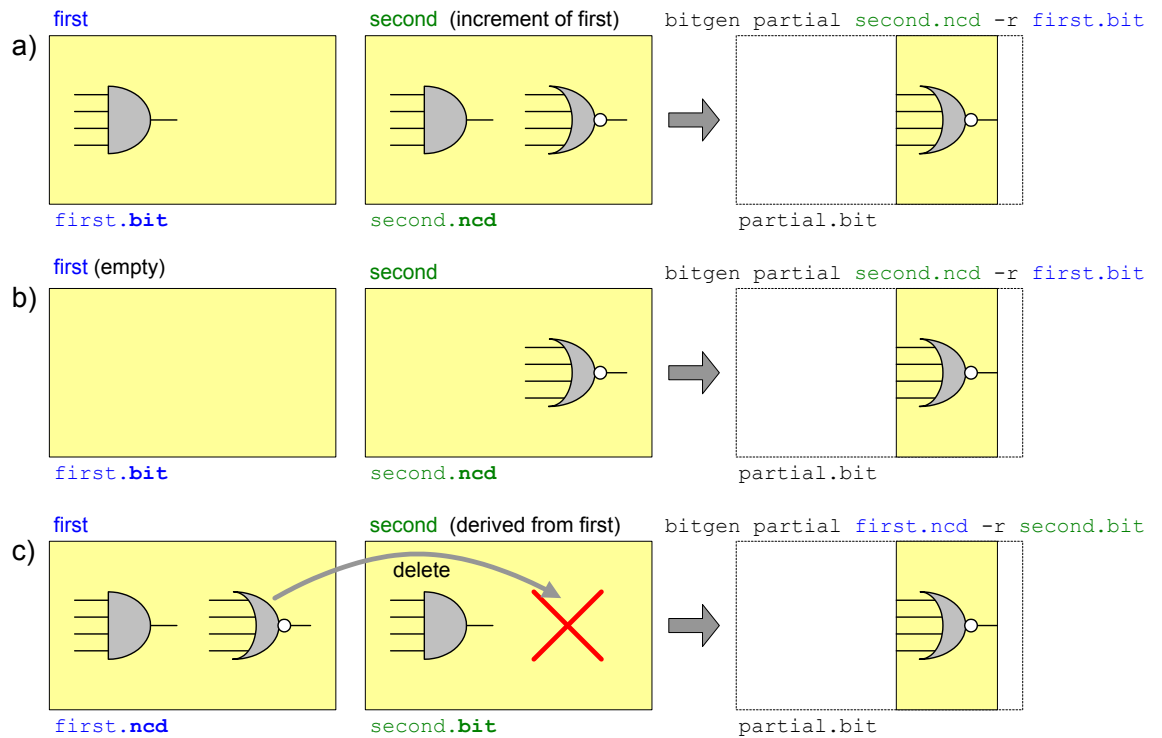


Figure 21: Partial bitstream generation using the differential bitstream option (`bitgen -r`). a) in the case the partial part is implemented as an increment to another (typically static) system. b) if the partial module is completely separated. c) as an alternative to b), we can delete the parts that should be added to the bitstream. Note, by changing `first` with `second`, a blank