The Xilinx Design Language (XDL): Tutorial and Use Cases

Christian Beckhoff, Dirk Koch, and Jim Torresen
Department of Informatics, University of Oslo, Norway
Email: {dirk, christian}@recobus.de, jimtoer@ifi.uio.no

***Abstract*—With the Xilinx Design Language (XDL), the FPGA vendor Xilinx offers a very powerful interface that provides access to virtually all features of their devices. This includes on one side the generation of complete device descriptions containing information about the FPGA primitives and the routing fabric. On the other side, XDL can be used to constrain systems or to directly implement modules or macros for Xilinx FPGAs.**
In this paper, we will provide documentation on the language and reveal several use cases for this language.

I. Introduction

Despite its powerful capabilities, only a few projects in the field of reconfigurable computing use the Xilinx Design Language XDL as an interface to the Xilinx vendor tools. In the following, we give an overview of related projects. Based on their objectives, the projects can be classified into three different groups:

1) *Macro generation:* Several projects use XDL as an interface to create regular structured macros in the field of reconfigurable computing (or to build time to digital converters). Contributing work can be found in [1], [2], [3], and [4]. Macro generation also includes the generation of special *blocker macros* for constraining the routing of designs, see [1] and [2].
2) *Provide an API to XDL:* Other projects provide an API to read, manipulate, and write XDL thus hiding syntactic details from the user, see [5], [6], and [7]. The APIs are intended to be used for CAD tool development.
3) *Documentation:* Xilinx offers a brief documentation in [8], more detailed information on older devices can be found in [9], [10]. This work provides the most recent documentation on XDL.

The three objectives together with their contributing publications are summarized in Table I. Most papers in the fields 1), 2), and 3) provide a only brief overview to XDL and many important details are omitted. In addition, the publications assigned to objective 3) were published years ago and refer to older FPGA families. A comprehensive and recent overview of XDL is missing. The intention of this paper is to close this gap and to provide information needed to implement own macros or to integrate custom tools into the Xilinx vendor tools.

We will continue the paper with presenting XDL fundamentals in Section II. After this, in Section III, we will continue with a presentation of XDL use cases. Then, we will discuss common pitfalls and issues related to XDL in Section IV and finally conclude the paper.

Objective	Contributing Projects
Macro generation	ReCoBus-Builder [1], GoAhead [2], Busmacro Generator [3], and DHHarMa [4]
Provide an API to XDL	RapidSmith [5], Torc [6], and FPGA Analysis Tool [7]
Documentation	Xilinx [8], Hoplite [9], Wire database [10], and this work

TABLE I
All XDL related research projects can be categorized into three groups based on their objective: 1) Macro generation, 2) Provide an API to XDL, and 3) Documentation

II. XDL Features

With XDL, Xilinx provides a human readable view to both 1) the resources available on FPGAs and 2) to FPGA netlists (e.g. complete systems, modules, or hard macros). Although provided in the same language, both views differ in syntax and structure, as revealed in the following.

A. XDL resource descriptions

A resource description for any Xilinx FPGA can be generated with the use of the command line tool `xdl`, e.g. `xdl -report -pips -all_conns xc6slx16`. The sizes of the generated resource descriptions vary from a few megabytes for smaller and older devices up to several gigabytes for recent devices[1].

Xilinx FPGAs consist of an array of *tiles* as depicted in Figure 1. There are several different tile types. Any user logic is implemented in tiles called *Configurable Logic Blocks*. A CLB consists of two slices containing the look-up tables (see Figure 3 for an example of two CLBs).

On its left hand side, each CLB has an *interconnect tile*. An interconnect tile consists of a switch matrix providing access to the routing fabric of the FPGA and is used to connect different CLBs. Although a CLB and its neighboring interconnect tile belong together, they are separated into two distinct tiles within

[1]The size of XDL resource descriptions is huge because the information is printed separately for each tile (e.g., a CLB), despite that there exist only very few different kinds of tiles. In order to reduce the memory requirements in a custom tool, groups of identical tiles can be automatically identified when parsing in an XDL description and replaced by references. For example, after parsing the 5 GB XDL resource description of an xc5vlx110t device (the FPGA of the XUPV5 board) into internal data structures, our tool GoAhead consumes around 190 MB of program memory. Storing the device description in our own format takes 28 MB on disk while preserving virtually all information.

an XDL description[2]. Besides CLBs and interconnect tiles, further tile types are available such as clock tiles, block ram tiles, and I/O tiles with each tile type having its own specific coordinate system.

The tiled structure of an FPGA is reflected in the structure of its XDL resource description as depicted in Listing 1. Each resource description is headed with the device and family of the FPGA. After the header, the tile section follows containg a description for each tile. The tile section starts first with a `tiles` statement holding the the X and the Y dimension of the array of tiles. The remaining part of the tile section is structured hierarchically.

Within the tiles section, each tile of the FPGA is described with a single node, whereat each node provides a unique pair of coordinates for a global X-Y coordinate system. The X-Y coordinates are followed by a tile name and a tile type. The tile name contains again two X-Y coordinates for the tile type specific coordinate system. Hence, a tile can be addressed by two pairs of coordinates. One pair points into the global and a second pair points into a tile type specific coordinate system.

For example the `tile 4 5 INT_X1Y61` in line 16 in Listing 1 describes an interconnect tile with the X-Y coordinates $(4, 5)$ pointing into the global coordinate system and the interconnect tile specific coordinates $(1, 61)$ pointing into the CLB specific coordinate system. The placement of `INT_X1Y61` into both coordinate systems is depicted in Figure 1.

Listing 1. An hierarchical XDL resource description of a Spartan-6 FPGA consisting of a header, a tile section containing 73×62 nodes each describing one tile of the FPGA, and a trailing device summary.

```
 1  # header
 2  (xdl_resource_report v0.2 xc6slx16csg324-3 spartan6
 3  # dimension
 4  (tiles 73 62
 5   ...
 6   # configurable logic block with two slices
 7   (tile 4 6 CLEXL_X1Y61 CLEXL 2
 8     (primitive_site SLICE_X0Y61 SLICEL internal 45
 9      (pinwire A1 input L_A1)
10      ...
11      (primitive_site SLICE_X1Y61 SLICEX internal 43
12      ...
13      (pinwire D output XX_D)
14    ...
15   # interconnect tile
16   (tile 4 5 INT_X1Y61 INT 1
17     ...
18     (wire EE2B0 2
19       (conn CLEXM_X2Y61 CLEXM_EE2M0)
20       (conn INT_BRAM_X3Y61 EE2E0)
21     ...
22     # switch matrxi multiplexers
23     (pip INT_X1Y61 EE2E0 -> EE2B0)
24     (pip INT_X1Y61 EE4E0 -> EE2B0)
25   (pip INT_X1Y61 EL1E_S0 -> LOGICIN_B9)
26    ...
27  # summary
28  (summary tiles=4526 sites=5378 sitedefs=46
29     numpins=157962 numpips=5782505))
```

[2]In older FPGA devices such as Virtex-II and Spartan-3, the CLB and the interconnect tile were merged into a single tile. Since Virtex-4 FPGAs however, interconnect tiles and CLB are split.

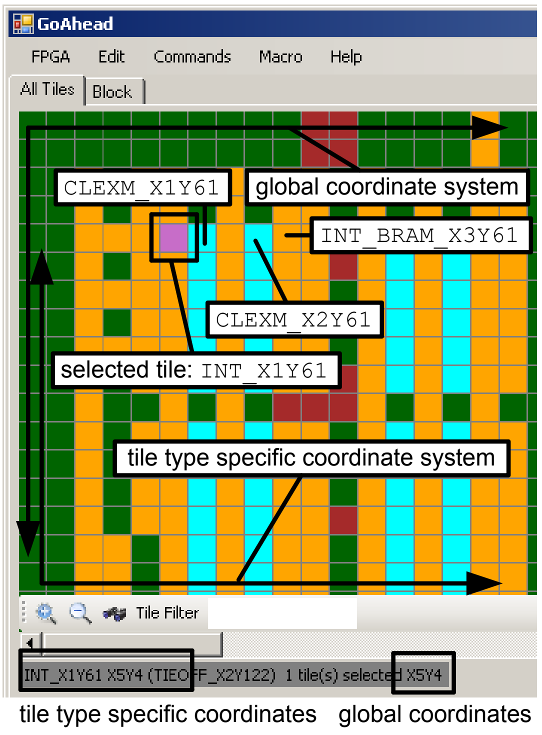


Fig. 1. Spartan-6 FPGA as an array of tiles. Each tile is colored according to its tile type (cyan for CLBs, orange for interconnects tile, green for I/O tiles, and brown for block ram tiles). The interconnect tile `tile 4 5 INT_X1Y61` in the upper left corner of a FPGA is selected. The global and tile type specific coordinates of the selected tile are given in the footer. The global coordinate system originates in the upper left corner while the tile specific coordinate system originates in the lower left corner of the FPGA. The selected interconnect is used to connect `CLEXL_X1Y61` with other tiles, e.g. `CLEXL_X2Y61` and `INT_BRAM_X3Y61`.

The tile `INT_X1Y61` is used to connect `CLEXL_X1Y61` to other CLBs. Therefore, the tile global coordinates of tile `CLEXL_X1Y61` (see line 7 in Listing 1) point directly left of tile `INT_X1Y61`.

Each node in the tile section may contain further hierarchies, e.g. the CLB `CLEXL_X1Y61` contains two slices, in XDL referred to as `primitive sites`. The number of sub nodes at any level of the hierarchy is denoted by the number after the tile type. For example, `CLEXL_X1Y61` of type `CLEXL` contains the two slices i.e. `SLICE_X0Y61` and `SLICE_X1Y61`. In Spartan-6 FPGAs there are three types of slices: `SLICEL`, `SLICEX` and `SLICEM`. Slices of type `SLICEX` only contain look-up tables and flip-flops, while `SLICEL` additionally provide carry-chains. Slices of type `SLICEM` also provide lookup-up tables, flip-flops, and carry-chains, the look-up tables however can be used as shift-registers and distributed memory. The different slices can be investigated with the FPGA-Editor tool. This also includes the naming convention for primitive I/O pins. In line 7 of Listing 1, the CLB `CLEXL_X1Y61` contains one `SLICEX` and one `SLICEL` slice. Each of both slices contains further `pinwire` nodes which describe input and output ports of the slice. Slice `SLICE_X0Y61` e.g. provides the input port `L_A1`.

Note that `SLICE_X1Y61` of type `SLICEX` does not provide a carry-chain, hence it provides less inputs and consequently less `pinwire` statements than `SLICE_X0Y61` of type `SLICEL`.

The interconnect tiles describe the FPGA routing architecture. For example, line 16 in Listing 1 declares an interconnect tile. The routing is described by `connection` statements that specify which neighboring CLBs can be reached from the current interconnect tile. For example, line 18 in Listing 1 states that the wire `EE2B0` connects the interconnect tile `INT_X1Y61` with two other tiles: 1) with the CLB `CLEXM_X2Y61` and 2) with the block ram tile `INT_BRAM_X3Y61` both located east of `INT_X1Y61` (see Figure 1 for the placement of `CLEXM_X2Y61` and `INT_BRAM_X3Y61`).

Wires are named by their direction (e.g., EE for eastwards or NN towards north, as illustrated in Figure 2) and their routing distance in terms of tiles. Furthermore, wires are unidirectional with a begin port denoted by a `B` and an end port denoted by a `E`, see line 23 in Listing 1. As depicted in line 25 in Listing 1, some wires provide an optional tapped port to a CLB in the middle of the way to the end port, e.g. the wire `EE1B0` provides the tapped port `EL1E_S0` and the end port `EE1E0`. The location of the tapped middle port is depicted in Figure 2[3].

Furthermore, an interconnect tile describes the adjacency of the switch matrix by tuples called *programmable interconnect point* (PIP) which specify a configurable connection between a switch matrix input and a switch matrix output. As listed in lines 23 and 24, a switch matrix multiplexer is denoted by multiple PIPs (one for each input) for the same output wire, e.g. the `EE2B0` wire may be driven by `EE2E0` or `EE4E0`. The XDL resource description is closed with a summary comprising statistics on the total number of tiles or pins.

B. XDL netlist descriptions

In addition to FPGA resource descriptions, XDL can also be used to implement a complete design as an XDL netlist description. A user may also convert existing designs into an XDL netlist description using again the command line, e.g. `xdl -ncd2xdl design.ncd design.xdl`. The user may vice versa also convert an XDL netlist into a design with `xdl -xdl2ncd design.xdl design.ncd`. An XDL netlist description is thus the human readable counterpart of the binary *ncd*-format. These two netlist formats describe an implementation after technology mapping of a design to the FPGA primitives (e.g., slices and BRAMs). Depending if place and route has been performed, the netlists can include placement information of primitives as well as the exact routing in terms of switch matrix settings.

Although sharing common syntactic elements, a netlist and a resource description in XDL completely differ in their structure. An XDL netlist description of a bus macro is depicted in Listing 2. The netlist description starts with a header that introduces the design name and the target device followed by the body of the design. Inside a design, there may be several `modules` declared. In Listing 2, a module named `S6BM` is declared. The body of the module comprises 1) ports, 2) instances, and 3) nets.

With the `port` statement a user may declare ports of his design. The port statement `port "LI(0)" "left" "D1"` in line 4 in Listing 2 declares a port with the identifier `LI(0)`. The port is located on an instance called `left`. Here, `left` is a slice instantiated in line 8 of Listing 2. The port `LI(0)` is mapped to the look-up table input port `D1` of that slice. Note that, we do not specify whether the port is an input or an output port, instead the direction of the port is implicitly given by the port itself. In the XDL resource description, the ports `A1` and `D1` are both declared as input ports, consequently `A1` and `D1` are input ports of the module. More information about ports is revealed in the FPGA-Editor.

Listing 2. Spartan-6 bus macro implementation in XDL

```
1  design "S6BusMacro.ncd" xc6slx16cpg196-2 v3.2 ;
2  module "S6BM", "left" cfg "_SYSTEM_MACRO::FALSE";
3  # I/O ports (4 I/Os per direction&side+CLK&reset)
4    port "LI(0)" "left" "D1";
5    port "LI(1)" "left" "A1";
6    ...
7  # component instantiations
8  inst "left" SLICEX,placed CLEXM_X8Y33 SLICE_X11Y33,
9   cfg "A6LUT:left.A6LUT:#LUT:O6=A1 AFFMUX::AX
10       AUSED::0 AFFSRINIT::SRINIT0 AFF:left.AFF:#FF
11       ...
12       D6LUT:left.D6LUT:#LUT:O6=A1 DFFMUX::DX
13       DUSED::0 DFFSRINIT::SRINIT0 DFF:left.DFF:#FF
14       SRUSED::0 SYNC_ATTR::SYNC CLKINV::CLK ";
15 inst "right" SLICEX,placed CLEXL_X9Y33 SLICE_X13Y33,
16  cfg "A6LUT:right.A6LUT:#LUT:O6=A1 ... ";
17
18 # dummy nets for I/O pins
19 net "LI(0)", inpin "left" "D1", ;
20   ...
21 # nets between left and right slice
22 net "l2r_0" ,
23   outpin "left" D ,  inpin "right" AX ,
24   pip CLEXL_X9Y33 CLEXL_LOGICIN_B6 -> XX_AX ,
25   pip CLEXM_X8Y33 X_D -> CLEXM_LOGICOUT9 ,
26   pip INT_X8Y33 LOGICOUT9 -> ER1B0 ,
27   pip INT_X9Y33 ER1E0 -> LOGICIN_B6 ,   ;
28 ...
29 endmodule "S6BM" ;
```

User logic is implemented in slices. Each slice that is used in a module must be instantiated explicitly with e.g. `inst "left" SLICEX, placed CLEXM_X8Y33 SLICE_X11Y33`. An instance declaration starts with the keyword `inst` followed by the unique name of instance, here the instance is called `left`. The instance name is followed by instance *type*, here a slice of type `SLICEX` is used. The succeeding keyword `placed` denotes a fully placed instance. As an alternative, the keyword `unplaced` might be used instead to declare an instance whose placement is not yet defined. If the slice is placed however, after the keyword `placed` the CLB in which the slice is instantiated must succeed. Each CLB contains two slices, therefore we must finally specify the slice we want to instantiate, e.g. `SLICE_X11Y33`.

[3]For other FPGAs, including Virtex-II, Spartan-3, or Virtex-5, the begin, middle and end ports have been named with `BEG`, `MID`, and `END`, as used in Listing 5

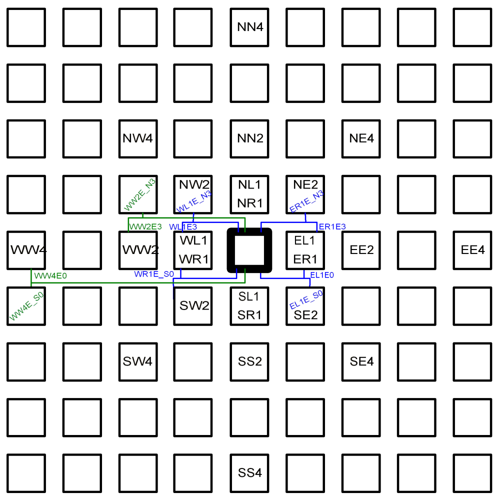


Fig. 2. Spartan 6 routing wires.

Each instantiation contains a configuration (`cfg`) body that allows the user to configure the slice, e.g. to control multiplexers, flip-flops and lookup-tables. For example, the slice instantiation in line 8 in Listing 2 configures one of the four lookup tables using: `D6LUT:left.D6LUT:#LUT:O6=A1`. Here, the look-up table output is directly connected to the input port `A1`. A more complex look-up table function might use all six inputs and boolean operators (@ for XOR, + for OR, $*$ for AND, and $\sim$ for NOT), e.g. `LUT:O6=((A1@A2)+(A3@A4)*(A5+~A6)`.

Instances are connected via nets. A net is declared with the keyword `net` followed by an identifier, e.g. `net "l2r_0"`. Each net has one `outpin` and may have several `inpins` as depicted in line 21 of Listing 2. Each outpin and each inpin must be located on an instance, e.g. `outpin "left" D` assigns the port `D` on instance `left` as a net outport.

The routing from the outpin to the inpins is coded with PIPs coding a particular multiplexer setting in a switch matrix. For example, `pip INT_X8Y33 LOGICOUT9 -> ER1B0` connects look-up table output `LOGICOUT9` with the wire `ER1B0`. Several switch matrix multiplexer setting together form a complete net. Note that it is also possible to completely omit `pip` statements in a net and only to declare an outpin and inpins. Such a net is then fully unrouted.

An XDL description of a module is finally closed with the endmodule statement in line 29 in Listing 2. After presenting syntactic details of the resource and netlist descriptions in XDL, the next Section will present how a user can interface the Xilinx tool chain with XDL.

III. Use Cases

A. Bus Macro Implementation

Before Xilinx has introduced its fourth generation partial design flow [11], macros called *bus macros* have been used to implement the interfaces between the static system and the reconfigurable modules that will be swapped at run-time [12].

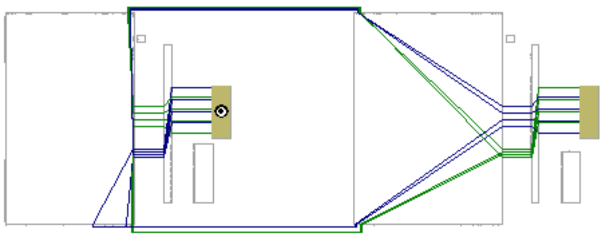


Fig. 3. Spartan 6 bus macro.

The fundamental requirement among static only systems in the physical implementation of a reconfigurable system is the binding of the partial module entity signals to a set of predefined wires on the FPGA fabric that will act as plugs. By further separating the partial resources from the resources used to implement the static system (i.e. defining reconfigurable regions), run-time reconfigurable systems can be implemented.

The recent design flow provided by Xilinx is based on an incremental flow where first the static system is implemented which in particular includes the wires to the reconfigurable modules located in the partial regions. Starting always from this system, partial modules are implemented by incrementally adding the module logic and routing to the system by preserving the initial static system. However, the particular wire resources crossing the reconfigurable regions cannot be constrained in this flow which prevents module relocation. Furthermore, a modification in the static system typically changes the routing to the reconfigurable modules. Consequently, all reconfigurable modules have to be routed again making this flow not scaling towards complex systems with many different modules.

These restrictions can be circumvented by using the bus macro approach [12]. However, despite that implementing such macros is trivial, they are not provided by the vendor for their latest devices. In the following, we will exemplary present how bus macros can be implemented for Xilinx Spartan-6 FPGAs using XDL and the FPGA editor. A screenshot of the macro is shown in Figure 3. It consists of two slices (named `left` and `right`) and two groups with four wires each for implementing signals between `left` and `right` (labeled `l2r`) as well for implementing signals in opposite direction (labeled `l2r` respectively).

The macro is symmetric and the two slices are configured identical as depicted in Figure 4 for `right`. Input signals pass a look-up table in route through mode, are then routed to the entire other slice, and finally connected to a flip-flop in that slice. The figure contains the XDL statements that set the configuration of the slice Note that the macro can be easily adjusted to specific needs. For example, by using the second output of the LUT and the unused flip-flop, up to four more signals could be linked between `left` and `right`.

Listing 2 shows the XDL code of the macro. After a header and the port interface, the instantiation of the two slices for `left` and `right` follows. The slice configurations have been derived using the FPGA editor (see Figure 4).

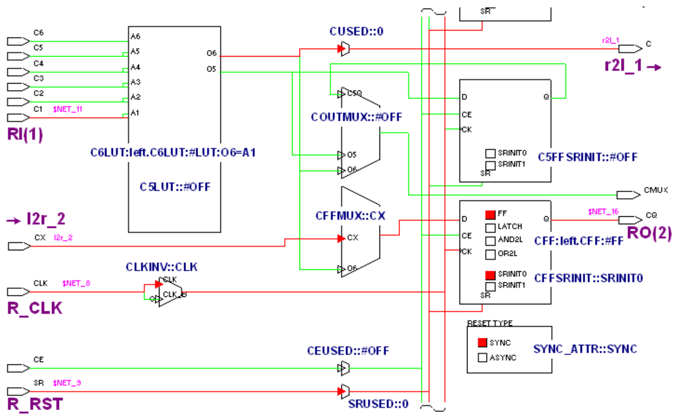


Fig. 4. Spartan 6 bus macro.

As the look-up tables are only used to connect signals and not to evaluate logic functions, they have been set in route through mode. Note that despite the first input is used in each LUT (i.e., `A1`, `B1`, `C1`, and `D1`), the used input might differ if pin swapping is allowed to improve routing. In most cases, this can be accepted but has to be prohibited for the internal routing of the macro. This will automatically be the case if the internal routing is fully specified from an output pin to all inputs and if this net is not connected to any other logic within the system (i.e., the driving output pin is not exported as a port). If not following this rule, the router ignores the specified routing and puts signal paths in an uncontrolled manner.

The entire internal macro routing paths (starting at line 21 in the listing) can be determined by rebuilding a routing database from the FPGA device description as revealed in Section II-A. This is what some tools provide together with a path search function. In GoAhead, for example, there is a function to search for paths between two slice pins and the result can be directly translated into corresponding PIP statements.

As mentioned in the last paragraph, a user can declare ports in XDL modules. However, the support varies between the different ISE versions. While older versions (e.g., ISE 6.3) support the declaration of ports directly in an XDL specification of a macro, more recent ISE versions however, drop all the declared ports and generate a design without any ports. In other words, all recent ISE versions (including version 12.x and 13.1) will ignore the port declarations in the beginning of an XDL macro. As a workaround, a user may generate a FPGA-Editor script as depicted in Listing 3 to attach the ports to the design after the XDL netlist has been converted to ncd-format.

However, for each port to be added in the FPGA-Editor script there must be a dummy net as outlined in line 18 in Listing 2. This net is only required for the conversion of the macro as a used primitive pin cannot be simply unconnected. Within the FPGA editor, these dummy nets are deleted before adding the macro I/O pins. These pins are specified by selecting a primitive pin and applying the `add extpin` command (e.g., the clock input of a slice as listed in line 8 to 12 of the script). Note that it is possible to specify vectorized I/O symbols by using brackets (as shown in line 18).

After saving the macro in the `nmc` format it has to be stored in the project directory of the system and can be instantiated using HDL design methodology. A VHDL code example is given in the appendix. Note that the macro could alternatively be generated completely in the FPGA editor without the need of the XDL tool. A corresponding FPGA editor script is listed in the appendix. However, generating macros directly inside the FPGA editor is more applicable for smaller macros as the internal routing is more difficult to specify.

Listing 3. The FPGA-Editor script used to add a port to a design create from an XDL netlist

```
1  unselect -all
2  select  net "LI(0)"
3  select  net "LI(1)"
4  ...
5  delete
6
7  unselect -all
8  select pin SLICE_X13Y33.CLK
9  add extpin
10 post attr pin SLICE_X13Y33.CLK
11 setattr pin SLICE_X13Y33.CLK
   external_name R_CLK
12 unpost pin "SLICE_X13Y33.CLK"
13
14 unselect -all
15 select pin SLICE_X13Y33.B1
16 add extpin
17 post attr pin SLICE_X13Y33.B1
18 setattr pin SLICE_X13Y33.B1  external_name RI(0)
19 unpost pin "SLICE_X13Y33.B1"
20 ...
```

When using macros in the HDL flow, their placement should be specified in the user constraints file (`ucf`). For macros being more complex than a single primitive (e.g., a slice), the placer will typically fail to find a valid placement position during the `map` phase. In the XDL specification of a macro, one primitive is used as a reference point for the placement of the whole macros and all further primitives are automatically specified as a relative displacement from the reference point. Line 2 in Listing 2 defines the slice primitive `left` by its symbolic name as the macro reference point. In the `ucf` file the placement has to be individually set for each macro with a location constraint specifying the target primitive position for the macro reference point, for example `INST macro_label LOC = SLICE_X23Y35;`.

B. *Routing Constraints*

For implementing a reconfigurable system using the bus macros approach, the macros have to be placed on the border between the static system and a partial module, as depicted in Figure 5. This can be accomplished using the location constraints mentioned in the last paragraph. For further floor-planning there exist `area group` and placement `prohibit` constraints to seperate primitive regions into the static system and regions to host patial modules, but there is no counterpart for routing resources. As a solution, blocker macros can be generated that occupie a definable set of routing rsources. A blocker can be transparently concatenated to a design after the placement step but before the routing step with the

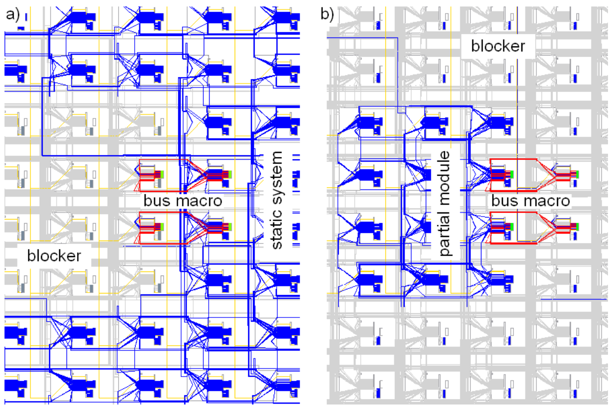


Fig. 5. Constraining the routing in a partial system using blocker macros. a) static system, b) partial module. The clock net is dyed yellow, blockers gray.

help of XDL. A blocker consists of a primitive instantiations acting as drivers. The drivers are the starting point of antenna nets that posses no input pins but contain PIP statements. These antennas will not be touched by the Xilinx router and will consequently prevent the usage of the routing resources included into the antenna set.

The blocker macros can further be used to activate clock drivers within reconfigurable regions during the implementation of a static system. This is achieved by adding PIP statements to the particular clock net as shown in Listing 4. Note that with the output driving a clock net, the primitive and consequently the particular global clock net of the FPGA fabric will be specified (e.g., a `BUFG` clock driver primitive). With this technique, multiple different clock nets can be activated in the partial region. This idea has been derived from [13].

Listing 4. Spartan-6 bus macro implementation in XDL

```
net "clk_100" ,
  outpin "clk_generator/PLL1_CLK_BUFG_INST" O,
  inpin "Hex2Bin_1/HighReg<3>" CLK ,
  ...
# add blocker clock inputs to clock net
  inpin "SLICE_X10Y22" CLK ,
  inpin "SLICE_X10Y21" CLK ,
  ...
```

C. Clock Remapping

In the last paragraph, we presented how blockers can be used to drive one or more clock nets into a reconfigurable region. With the help of the XDL language, it is further possible to remap a reconfigurable module or any other part of a system to another clock without affecting the rest of the system. This can be used in a component-based design flow to migrate a fully placed and routed module from one system to another even if they have been implemented using different global clock networks. A further use case is the adaptation of the operation speed of some modules by swapping between two global clock domains. Consequently, no dedicated clock domain is required to control the clock frequency of a module which permits to apply this approach fine-grained on many individual modules.

For remapping a clock, the clock `input` statements have to be moved to the new clock net (see Listing 4 for an example). In addition, the clock select multiplexer configurations have to be adjusted. This can be done by removing switch matrix settings (e.g., statements looking like `pip INT_X13Y33 GCLK12 -> CLK0`, and slice connection wires (e.g., `pip CLEXL_X13Y33 CLEXL_CLK0 -> XX_CLK,`) from the original clock net. In order to set these connections for the new clock net, the FPGA editor can selectively route the new clock net which will then add the missing PIP statements without changing the rest of the clock tree. As an alternative, the slice connection wire statement can be moved together with the switch matrix setting to the target clock net while adjusting the switch matrix setting (e.g., `GCLK12 -> CLK0` might become `GCLK3 -> CLK0`).

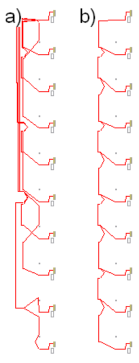
Both the original and the modified system can be analyzed using the Xilinx timing verification tool. Furthermore, partial configuration bitfiles for changing between the two systems can be generated using the differential bitstream option `-r` in the `bitgen` tool. Assuming the existence of the two netlists and full bitstreams of the two systems, the two following commands will generate the partial bitfiles:

```
bitgen change2modified.bit modified.ncd -r start.bit
bitgen change2start.bit start.ncd -r modified.bit
```

D. Homogeneous Place and Route

The Xilinx vendor tools provide no option to generate homogeneously arranged physical implementations of any part of a system. Even if primitives have been regularly placed, the routing will be irregular as shown in Figure 6a). However, there exist applications that require a homogeneously arranged implementation, as shown in Figure 6b). This is in particular useful for implementing on-FPGA communication architectures for reconfigurable systems (e.g. [14]) or for time to digital converters (e.g. [4]). For the latter case, the propagation delay of a signal is used to measure pulses faster than the clock speed. This is achieved by connecting a signal via a delay network to multiple flip-flops which sample on the same clock. Figure 6b) gives an example of a time to digital converter implemented on a Xilinx Virtex-5 FPGA. The homogeneous routing ensures a linear latency increase from flip-flop to flip-flop and a time resolution below 100 ps can be achieved in practice.

While some Xilinx FPGAs, such as Virtex-II devices, allow to directly extend a routing path in a homogeneous manner (e.g., in each switch matrix there exists the PIP `S2END0 -> S2BEG0`), this is not directly supported in the Virtex-5 routing fabric. Searching for such routing paths is non-trivial and can be automated by tools that build up the routing architecture of a Xilinx device using a XDL device description in order to search for homogeneous paths. This feature is provided in the tools ReCoBus-Builder [1] and GoAhead [2] that provide, among traditional shortest path routing, search functions for homogeneous paths. For Virtex-5 FPGAs, a path can be arranged homogeneously with a repeating sequence over a



Listing 5. Homogeneous delay line routing (line 5…9) for b)

```
 1 net "delay_net" ,
 2 outpin S0 C, inpin S1 C3,inpin S2 C5,...
 3 pip CLBLL_X19Y99 L_C->SITE_LOGIC_OUTS10,
 4 pip INT_X19Y99 LOGIC_OUTS10->SR2BEG1,
 5 pip INT_X19Y98 SR2MID1 -> SW2BEG1,
 6 pip INT_X19Y97 SW2MID1 -> SR2BEG1,
 7 pip INT_X19Y96 SR2MID1 -> SW2BEG1,
 8 pip INT_X19Y95 SW2MID1 -> SR2BEG1,
 9 pip INT_X19Y94 SR2MID1 -> SW2BEG1,
10 pip INT_X19Y98 SR2MID1 -> IMUX_B8,
11 pip CLBLL_X19Y98 SITE_IMUX_B8 -> L_C3,
12 pip INT_X19Y97 SW2MID1 -> IMUX_B9,
13 pip CLBLL_X19Y97 SITE_IMUX_B9 -> L_C5,
14 ...
```

Fig. 6. Homogenously routed delay chain for time to digital conversion.

set of two CLBs in each direction. For south direction, for example, the pips of a homogeneous routing path repeat after two CLBs. This can be observed in the lines 5 to 9 of Listing 5 where a PIP repeats in every other CLB (i.e. an identical PIP entry for `INT_X(n)Y(m)` and `INT_X(n)Y(m-2)`). Note that paths could be implement that are identical in each CLB ((i.e. an identical PIP entry for `INT_X(n)Y(m)` and `INT_X(n)Y(m-1)`); but this would require additional resources within each CLB (e.g. a LUT in route-through mode when using a Virtex-5 device). Homogeneous paths can be found by generating all paths up to a certain depth (in terms of hops or PIPs) and filtering out the paths that have repeating PIP entries in consecutive CLBs.

E. Relocation of netlists

Let us assume a fully placed and routed netlist. We can now iterate through the XDL description of this netlist and increment all X-Y coordinates we encounter by a given offset and thus relocate the whole netlist. There are however two pitfalls to consider. Firstly, we may not arbitrary relocate the netlist as we have to consider the underlying resource layout of the FPGA (e.g., the position of block rams and I/O tiles). Secondly, as depicted in Figure 1, there is a global X-Y coordinate system and a plethora of coordinate system for all the different tile types. Hence, for the relocation of a netlist, we may not add the same increment to the X-Y coordinate of each tile type. XDL Netlist relocation is a feature provided in the tool GoAhead.

F. Design merging

In addition to relocating a netlist, a user could also merge two existing designs given as XDL netlists into a single design (in case the target platform provides sufficient resources). For example, a user might want to integrate functionality of two or more different FPGAs into a larger one. In case the two netlist use disjoint resources, their XDL descriptions can be simply concatenated. Let us however assume two fully placed and routed designs as given in Listing 6.

Listing 6. Two fully placed and routed designs. The identifiers for neither the instances not the net names are unique both designs. To merge the designs, we add unique prefixes to each identifier.

```
# design 0
inst "logic_inst_0" "SLICEL", placed
  CLEXM_X5Y74 SLICE_X6Y74,

net "net_0",
  outpin "logic_inst_0" AQ,
  inpin "logic_inst_0" A5,
  ...
# design 1
  inst "logic_inst_0" "SLICEL", placed
    CLEXM_X5Y74 SLICE_X6Y74,

net "net_0",
  outpin "logic_inst_0" AQ,
  inpin "logic_inst_0" A3,
  ...
```

In order to merge any kind of design, we take advantage of the possibility to assign symbolic identifier to instances. We can not simply concatenate the two netlist, as both designs instantiate the slice `SLICE_X6Y74` and further use the same port `AQ` on that instance. In addition, the instance names in both designs are not unique. Furth, the nets in both designs exhibit a resource conflict. In order to merge the two designs, we first prefix each identifier in each design with the design name. Then, we remove any placement information from the instances and any routing information from the nets. We can now concatenate the two netlist and convert the netlist to an completely unplaced and unrouted design. The placement and routing can now be performed by the Xilinx tools.

We could apply the same scheme for inserting a module into an existing designs. If necessary, we first prefix all identifiers to make them unique. Then, we only remove the placement and the routing information from the module to be inserted and append the modules netlist to the netlist of the design. Now, the Xilinx tools only have to place and route the newly integrated module while the larger design remains unchanged.

Merging design was already suggested in [8]. However, the designs were considered to use only disjoint resources.

G. Design analysis

For migrating a static only design to deploy partial runtime configuration, a floorplanning of the system is required [2]. Therefore, we have to determine the resource consumption of the static modules that we plan to migrate to partial modules. The resource consumption for a module can be measured in terms of the number of required look-up tables, block ram or DSP blocks. To determine the number of required look-up tables for a given module name, we can simply count the number of slice instances that contain the given module name.

If we generate the XDL netlist description form an existing design, the slashes inside the instance names denote the hierarchy of the instance. Hence, a custom tool might also deduce a hierarchical design view based on an XDL netlist description. Let us assume, the instances given in Listing 7. We can deduce two modules `mod_0` and `mod_1` instantiated at top level whereat module `mod_0` contains the submodule `mod_01`. In total, the design uses four slices.

Listing 7. The instance names in an XDL netlist can be used to deduce the design hierarchy and the resource consumptions

```
inst "mod_0/logic_inst_0" "SLICEL"
  ...
inst "mod_0/logic_inst_1" "SLICEL"
  ...
inst "mod_0/mod_01/logic_inst_0" "SLICEL"
  ...
inst "mod_1/logic_inst_0" "SLICEL"
  ...
```

Note that we may also extract placement information from the XDL netlist as e.g the tool PlanAhead [15].

IV. Common Pitfalls and Issues

The XDL syntax is highly suitable to act as a programming interface between the Xilinx vendor tools and custom plugins. Unfortunately, there are several issues to follow in order to succeed in the implementation of a running system. The following sections summarize lessons learned by extensively using XDL for implementing custom tool extensions such as our tools ReCoBus-Builder and the GoAhead suite.

A. Avoid Hierarchies for Macros

Macros are instantiated like any other module. However, when instantiating macros in a subhierarchy and depending on the macro and the Xilinx ISE version, some tools (e.g., map) might fail to implement the design. Consequently, all macros should be instantiated in the top-level design file.

B. Different ISE Versions

Many versions or service packs of the Xilinx ISE tools behave slightly different and might demand adaptations in the design flow or might fail in some particular feature. For Virtex-II and Spartan-3 FPGAs, we found the most stable version of the Xilinx ISE tools as the ISE version 10.1 service pack 3. However, when using more complex macros, map might crash. This behavior applies for the Windows and Linux version of the tool and under Linux, more complex macros are accepted.

There is unfortunately no general rule and the entire latest version of the tools are not necessary working more stable. For example, we found problems in the first generation of ISE 12.1. In this version, it was not possible to open a saved design in the FPGA-Editor for further editing it. This is crucial as this tool provides no undo function. Furthermore, the tools behave different for different devices and the support for latest devices was found to take a few releases or service packs to work stable also for more advanced features. For example, in ISE 12.3, we found that macro ports can be specified using vectors for Virtex-5 FPGAs, while the same for Spartan-6 devices let the placer to produce an error during the `map` phase. As a workaround, only bit signals have to be used.

V. Conclusions

With this paper, we provide useful information for designers who want to use the powerful Xilinx Design Language (XDL) for own tools or advanced system manipulations. We support this with plenty of practical examples and use cases. This includes in particular techniques using partial run-time reconfiguration. Hopefully, the XDL support becomes more stable in future versions of the Xilinx vendor tools and that other Vendors provide a similar interface to their tools.

Acknowledgment

This work is supported in part by the Norwegian Research Council under grant 191156V30

References

[1] D. Koch, C. Beckhoff, and J. Teich, “ReCoBus-Builder a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs,” in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL 08)*, Heidelberg, Germany, Sep. 2008, pp. 119–124.

[2] Christian Beckhoff, Dirk Koch and Jim Torresen, “Migrating Static Systems to Partially Reconfigurable Systems on Spartan-6 FPGAs,” *18th Reconfigurable Architectures Workshop RAW 2011,*, May 2011.

[3] Christopher Claus, Bin Zhang, Michael Hübner, Christoph Schmutzler, Jürgen Becker and Walter Stechle, “An XDL-Based Busmacro Generator for Customizable Communication Interfaces for Dynamically and Partially Reconfigurable Systems,” *Workshop on Reconfigurable Computing Education at ISVLSI 2007*, May 2007.

[4] M. K. Sebastian Korf, Dario Cozzi, “Automatic HDL-based generation of homogeneous hard macros for FPGAs,” in *Proceedings of the 19th IEEE Symposium Field-Programmable Custom Computing Machines (FCCM'11)*. IEEE-CS Press, April 2011.

[5] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, “Rapid Prototyping Tools for FPGA Designs: RapidSmith,” in *International Conference on Field-Programmable Technology (FPT'10)*, December 2010.

[6] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, “Torc: Towards an Open-Source Tool Flow,” in *Proceedings of the 19th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, 2011, pp. 41–44.

[7] K. Kepa, F. Morgan, K. Kosciuszkiewicz, L. Braun, M. Hübner, and J. Becker, “FPGA Analysis Tool: High-Level Flows for Low-Level Design Analysis in Reconfigurable Computing,” in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, J. Becker, R. Woods, P. Athanas, and F. Morgan, Eds. Springer Berlin / Heidelberg, 2009, vol. 5453, pp. 62–73.

[8] Xilinx, Inc., *The Xilinx Design Language*, Juli 2000, HTML documentation file supplied with ISE Verion 6.3.

[9] Matthew Scarpino, “The Hoplite Guide To Run-Time Reconfigurable-Computing,” 2008.

[10] Neil Joseph Steiner, “A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs,” Master's thesis, Virginia Tech, 2002.

[11] Xilinx Inc., “Partial Reconfiguration User Guide,” Dec. 2009, rel 11.4.

[12] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, “Invited Paper: Enhanced Architecture, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs,” in *Proceedings of the 16th International Conference on Field Programmable Logic and Application (FPL)*, Aug 2006, pp. 1–6.

[13] Xilinx Inc., *Two Flows for Partial Reconfiguration: Module Based or Difference Based*, May 2002.

[14] D. Koch, C. Beckhoff, and J. Teich, “A Communication Architecture for Complex Runtime Reconfigurable Systems and its Implementation on Spartan-3 FPGAs,” in *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '09. New York, NY, USA: ACM, 2009, pp. 253–256.

[15] X. Inc., *PlanAhead Design Analysis Tool*, 2010.

VI. Appendix

The scripts to generate the Spartan-6 bus macros are embedded as notes into the PDF document (click on the arrows).

- XDL script
- I/O-pin assignment script for the FPGA editor
- Complete macro generation using FPGA editor
- System with VHDL instantiation template