InterSynth User Manual

Clifford Wolf
http://www.clifford.at/

April 2012

1 Introduction

InterSynth is a command line tool implemented in C++ that generates (synthesizes) interconnects for heterogeneous coarse-grained reconfigurable logic circuits. Its input consists of descriptions of cells (only the interfaces, not the implementations) and example netlists utilizing this cells. These example netlists are then used by InterSynth to generate an interconnect that can implement the given example netlists and other circuits that are similar to these example netlists. This way it is possible to use InterSynth to create interconnects for a domain of applications using examples from this domain to describe it.

InterSynth can generate Verilog HDL files that implement the interconnect and instantiate the cells. It can also generate the configuration bitstreams for implementing netlists on a previously generated interconnect.

The topology used by InterSynth for the interconnects is a set of parallel trees. The rationale behind this design decision as well as a description of the algorithm used by InterSynth is given by the paper "*Example-Driven Interconnect Synthesis for Heterogeneous Coarse-Grain Reconfigurable Logic; Clifford Wolf, Johann Glaser, Florian Schupfer, Jan Haase, Christoph Grimm; IEEE [FIXME]*"[1]. In this manual we try to keep the focus on how to use InterSynth to actually implement interconnects.

This manual is structured in three parts: in this first section a brief overview of the InterSynth work flow is given using a simple example. In the second section all commands are explained in detail and in the third section usage examples for different scenarios are given.

1.1 Building and Installing

InterSynth is written in C++ and is using a simple `Makefile` to manage the build process. On UNIX/Linux machines it can be built with the command "`make`" and installed with "`make install`".

[1] FIXME: Hyperlink to paper here

InterSynth can be built using GCC or Clang. When using GCC the optional OpenMP support can be used for more performance on SMP systems. This options can be set in the header of the `Makefile` before running `make`.

1.2 A simple example

Usually InterSynth is used in one of two modes. Either it can be used to generate the interconnect itself (this is done before the chip is produced) or to generate a configuration for a previously generated interconnect (after the chip is produced or for testing the to-be produced chip).

Let's consider a simple example utilizing simple AND, XOR and NOT gates. First we need to describe the cell types and the connection types involved. In this example we only use simple 1-bit wide connections but most real-world examples involve one or more word-wide connection types.

```
conntype bin 1 2 1
topology bin 4 2
```

The `conntype` command declares a connection type. The four arguments are:

1. The name of the newly declared connection type (`bin`).
2. The number of bits used by this connection type (1).
3. The number of parallel trees for the interconnect (2).
4. The routing cost of this connection type (1).

The `topology` command is used to declare the topology that should be used for this connection type. InterSynth is always using tree topologies by the degree[2] of the interconnect tree can be set for each level of the tree using this command. In this example a maximum of four cells are grouped together on the lowest level of the interconnect tree and all layers above that implement a binary tree. The degree 0 specifies the last level. I.e. the command "`topology bin 3 3 3 0`" would create a tree that has a maximum depth of 4 with a degree of 3 for the first three levels.

Next the cell types must be declared. This is done using the "`celltype`" command:

```
celltype AND2X1 bin A bin B bin *Y
celltype XOR2X1 bin A bin B bin *Y
celltype INVX1 bin A bin *Y
```

[2]Maximum number of child nodes per node in the interconnect tree

The first argument for specifies the name of the new cell type. The remaining arguments are pairs of connection types (in this example always "`bin`") and port names. Output ports must be prefixed with an asterisk (`*Y`). The asterisk is only used when specifying the port in the `celltype` command, not when referring to the port in other commands.

There is more to say about the `celltype` command but for this simple example we don't need any of the additional features that are described in the later sections of this manual.

So far we have no means of communicating with the reconfigurable circuit as we have not defined any inputs or outputs. As far as InterSynth is concerned the input and output ports are just additional cell types:

```
celltype !INa bin *X
celltype !INb bin *X
celltype !INc bin *X
celltype !OUT bin X
```

In this example we are using different cell types for each individual input cell. This way it is possible to distinguish between the input cells in the netlists. If this is not required (e.g. because the inputs are interchangeable), a single cell type can be used for all or a group of inputs.

The exclamation mark in front of the input and output cell names indicate that this cells don't have an actual implementation that can be instantiated. The exclamation mark is not part of the cell type name and is therefore only used in the `celltype` command and not when referring to the cell type in other commands.

The commands `input` and `output` can be used to implement a wide range of connections between the interior of the HDL module generated by InterSynth with the outside world. In this example we only need the simplest version of this commands:

```
input A 1 INa X
input B 1 INb X
input C 1 INc X
output Y 1 OUT X
```

The first argument to `input` or `output` declares the name that should be used for the input or output signal in the generated HDL module. The 2nd argument specifies the bit width of the signal and the 3rd and 4th argument specify the cell type and cell port the signal should be connected to.

The only thing missing now are the example netlists. They are declared using the commands "`netlist`" and "`node`". E.g. an implementation of a 3-input AND would look like this:

```
netlist demo_and
node INa INa X a
node INb INb X b
node INc INc X c
node OUT OUT X F
node a1 AND2X1 A a B b Y tmp
node a2 AND2X1 A tmp B c Y F
```

The `netlist` command marks the beginning of a new netlist. Its argument is the name of the new netlist.

The `node` command instantiates a cell type. The first argument is the name of the new node. The 2nd argument is the cell type to be instantiated. All remaining arguments are pairs of cell ports and net names. Note that nets are not declared explicitly in this file format.

Similarly a 3-input AND and a 3-input OR can be implemented like this:

```
netlist demo_xor
node INa INa X a
node INb INb X b
node INc INc X c
node OUT OUT X F
node x1 XOR2X1 A a B b Y tmp
node x2 XOR2X1 A tmp B c Y F

netlist demo_or
node INa INa X a
node INb INb X b
node INc INc X c
node OUT OUT X F
node i1 INVX1 A a Y not_a
node x1 XOR2X1 A not_a B b Y ab_eq
node x2 XOR2X1 A not_a B c Y ac_eq
node a1 AND2X1 A not_a B ab_eq Y tmp
node a2 AND2X1 A tmp B ac_eq Y F_inv
node i2 INVX1 A F_inv Y F
```

The `presilicon` command can be used to create the actual interconnect from the example netlists. It takes two arguments. The first one is the number of iterations the algorithm should run (the recommended value is 6) and the second argument specifies a limit for the number of permutations without improvement after which the algorithm accepts the last best result in each of the many passes in the algorithm (bigger is better but takes longer, try 20 if you are unsure).

Usually the synthesis results are better if InterSynth's internal state is "shuffled" (set to a random state) before optimizing. This can be done using the "`shuffle cells`" and "`shuffle nodes`" commands.

Finally two files must be generated from the synthesis results. First a Verilog HDL file that contains the synthesised module and second a file that contains all the data InterSynth needs to generate configurations for the synthesised module. This can be done using the commands "`write_data`" and "`write_verilog`".

Thus the commands for synthesising the interconnect and saving the results are:

```
shuffle cells
shuffle nodes
presilicon 6 20
write_data chipdata.txt baseconfig cellmapping
write_verilog interconn.v name INTERCONN
```

The additional arguments for `write_data` select the data sets that should be written. The `name` argument to `write_verilog` specifies the Verilog name for the synthesised module (in this case "`INTERCONN`").

If all the commands quoted above are stored in a file `input.txt`, InterSynth can be called with this filename as argument in order to execute all commands within that file:

```
$ intersynth input.txt
```

It is also possible to specify InterSynth commands directly on the command line using the "`+`" prefix. So if e.g. the `conntype`, `topology`, `celltype`, `input` and `output` statements are in a file `basedata.txt` and the netlists are stored in `netlist_and.txt`, `netlist_xor.txt` and `netlist_or.txt`, the same job can be performed using the following command:

```
$ intersynth basedata.txt                                    \
       netlist_and.txt netlist_xor.txt netlist_or.txt        \
       +"shuffle cells" +"shuffle nodes" +"presilicon 6 20" \
       +"write_data chipdata.txt baseconfig cellmapping"     \
       +"write_verilog interconn.v name INTERCONN"
```

It is the recommended procedure to either have the data commands and the actual synthesis commands in different files or load the data from files and specify the synthesis commands on the command line.

The generated Verilog module for this example has the following interface:

```
module INTERCONN(bitdata, A, B, C, Y);
    input [57:0] bitdata;
    input A;
    input B;
    input C;
    output Y;
endmodule
```

The ports A, B, C and Y are the ports that have been specified in the InterSynth input. The input port "`bitdata`" contains the configuration for the multiplexers within the interconnect. Note that the generated module does not implement any shift registers for the configuration. This must be done outside of the generated module and the whole configuration must be applied as one large input to the interconnect.

The implementation of this module does only contain basic Verilog statements and instantiations of the modules `AND2X1`, `XOR2X1` and `INVX1`. It does not depend on any additional cell libraries and thus can be used with any synthesis tool that excepts Verilog input.

The command "`postsilicon`" (same arguments as presilicon) can now be used to create the interconnect configuration for a single netlist. Only one netlist may be loaded when the "`postsilicon`" command is used. The command "`write_bitdata`" can then be used to save the configuration data generated by InterSynth:

```
$ intersynth chipdata.txt netlist_or.txt   \
       +"shuffle nodes" +"postsilicon 6 20" \
       +"write_bitdata bitdata_or.txt"
```

Note that the "`shuffle cells`" command must not be used in conjunction with the "`postsilicon`" command as it would randomize the part of InterSynth's state that is loaded from `chipdata.txt`.

The generated file `bitdata_or.txt` contains a lot of debug information and the following line with the actual configuration bits:

```
bitdata bits 57:0 10010011001100000000100000010100110001001000000010001001000
```

When this bit pattern is applied to the `bitdata` input of the generated Verilog module, it implements a three input OR gate.

It is recommended not to start InterSynth manually from the command line but instead use (shell) script to integrate it with the ASIC work flow in order to guarantee reproducibility of the results.

2 Command reference

This section contains a complete reference for all InterSynth commands.

InterSynth has two different modes of operation. In the *setup mode* connection and cell types, interconnect topology and input/output ports are set up and netlists are read in. In the *synthesis mode* synthesis jobs are performed.

InterSynth automatically switches from setup mode to synthesis mode when the first synthesis mode command is executed. Once InterSynth has been switched to synthesis mode it is not possible to switch it back to synthesis mode. Thus the commands must be passed in the correct order.

2.1 verbose

Enable verbose mode. In this mode all commands are written to the output before being executed.

2.2 load [<filename> ..]

Load commands from one or more file. This command first executes all setup mode commands from all input files and then all synthesis mode commands. Therefore it is better to pass all files to one single `load` command than to use a separate `load` command per file.

A list of file names passed as command line arguments are processed using an implicit `load` command.

2.3 stats

Print some general information. Useful for profiling resource usage of synthesis results.

This command switches to synthesis mode.

2.4 conntype <name> <bit-width> [<trees> <cost>]

Declare a connection type with the specified name and bit-width. When interconnect tree should be generated for this connection type, the number of trees and the routing cost for this connection type must be specified as 3rd and 4th parameter. If unsure, use the bit-width as routing cost.

When the 3rd and 4th parameters are omitted a simple "big multiplexers" interconnect is generated instead of interconnect trees.

This command can only be processed in setup mode.

2.5 celltype [!]<name> [{ <conntype> [@|*]<port> | cfg:<bit-width> <cfgport> } ..]

Declare a cell type with the specified name. When the cell is not to be instantiated in the generated Verilog HDL code (e.g. for input/output cells),

the name must be prefixed with an exclamation mark. When no exclamation mark is put in from of the cell type name this name is also used as the name of the HDL module to be instantiated.

All following parameters declare cell ports. The "`<conntype> [@|*]<port>`" syntax can be used to declare a cell port that is exposed to the interconnect. Output ports must be declared with an asterisk prefixing the port name. Usually InterSynth does not create Data paths that allow an output signal from one cell to be directly fed back to an input of the same cell. Inputs that may be used as feedback inputs must be declared using the at-sign prefix.

The "`cfg:<bit-width> <cfgport>`" syntax can be used to declare cell inputs that are driven directly from additional bits embedded in InterSynth's config data. This is e.g. useful for cell that are reconfigurable by themselves (e.g. a combined Adder/Subtracter).

This command can only be processed in setup mode.

2.6 input <name> [*]<bit-width> [<celltype> { <port> | .<signal> } ..]

Declare an input port on the generated HDL module with the specified name and bit-width and connect it to cells.

In the simple variant (without the asterisk prefix for the bit-width) this command simply creates an input port with the specified bit-width and connects it to the specified ports of the cell types. With the "`<celltype> <port>`" syntax the connection goes to the signal within the interconnect that is connected to the cell port (this is used for input/output cells). With the "`<celltype> <port>`" syntax the connection goes directly to a port on the cell instances. This is e.g. be used to distribute a clock signal to the cells. In both cases the same single input signal is connected to all instances of the specified cell types.

With the asterisk prefix for the bit-width, a separate signal is routed to each cell of the specified cell type. I.e. when there are N cells and bit-width is set to M, the generated input port will be $N \times M$ bits wide. In this mode only one cell type and port may be specified.

This command can only be processed in setup mode.

2.7 output <name> [*]<bit-width> [<celltype> { <port> | .<signal> } ..]

Declare an output port on the generated HDL module with the specified name and bit-width and connect it to cells.

The syntax is identical to the `input` command. Unless the cells involved have tri-state outputs, the user must ensure that an output port is only

driven by one cell either by using the asterisk prefix for the bit-width or by making sure that there is only one instance of the specified cell type.

This command can only be processed in setup mode.

2.8 topology <conntype> [<width> ..]

Set the topology for all trees for the specified connection type. The width arguments specify the degree (maximum number of child nodes per node) for each layer of the interconnect tree. I.e. the first number specifies the maximum number of cells that should be connected to the same 1st level switch. The second number specifies the maximum number of 1st level switches that should be connected to the same 2nd level switch and so on.

InterSynth behaves as if the last number is repeated indefinitely. I.e. "`topology foobar 4`" declares that the interconnect trees for the connection type `foobar` should use a degree of 4 on all levels.

The width 0 can be used to specify the last level of the interconnect tree. It yields a (potentially huge) switch that connects all switches of the second last level.

A degree of 4 is a good starting point for finding the optimal topology by experimentation.

This command can only be processed in setup mode.

2.9 setcellcount <celltype> <num>

Usually the number of cells per cell type is determined by the example netlists. This command can be used to manually set this number.

In most cases this command is only used in the chip description generated by the `presilicon` command (as opposed to being part of a user-generated input).

This command can only be processed in setup mode.

2.10 setswitchsizes <conntype> <tree> [<size-up> <size-down> ..]

Usually the number of connections from a switch to its parent switch (size-up) and the number of connections from that parent switch to the switch (size-down) is determined by the `presilicon` command. This command can be used to set this information manually. (But a re-run of `presilicon` will overwrite the manually set value.)

In most cases this command is only used in the chip description generated by the `presilicon` command (as opposed to being part of a user-generated input).

This command switches to synthesis mode.

2.11 mapcell <celltype> <num> <conntype> [<pos> ..]

Usually the mapping of cells to leaves in the interconnect tree is determined by the `presilicon` command. This command can be used to set this information manually. (But a re-run of `presilicon` will overwrite the manually set value.)

In most cases this command is only used in the chip description generated by the `presilicon` command (as opposed to being part of a user-generated input).

This command switches to synthesis mode.

2.12 headroom cellup <celltype> [abs <num>] [rel <num>] [min <num>]

When the number of cells per cell type is determined from the example netlists, the smallest number necessary to still be able to implement all example netlists is chosen. This command can be used to specify how many cells should be added atop of the number determined from the example netlists.

The option "`abs <num>`" simply adds *num* cells to the original number of cells (*absolute increment*).

The option "`rel <num>`" multiplies the original number of cells by *num* (*relative increment*).

The option "`min <num>`" sets the number of cells to *num* if the original number of cells is smaller.

When more than one of this options is given, all the operations are performed in the given order. The final result is then rounded up to the next integer.

This command can only be processed in setup mode.

2.13 headroom switchup <conntype> [abs <num>] [rel <num>] [min <num>] [depth <num>] [up | down]

This command is simmular to the "`headroom cellup`" command but incrases the number of connections between the switches for the interconnect for the specified connection type.

The option "`abs <num>`" simply adds *num* cells to the original number of connections (*absolute increment*).

The option "`rel <num>`" multiplies the original number of connections by *num* (*relative increment*).

The option "`min <num>`" sets the number of connections to *num* if the original number of connections is smaller.

The option "`depth <num>`" adds the level within the interconnect multiplied by *num* to the number of connections.

The option `up` limits the effect of this command to the connections from switches to their parent switches. The option `down` limits the effect of this command to the connections from the parent switches to the children. When neither `up` nor `down` is specified this command effects both numbers.

This command can only be processed in setup mode.

2.14 netlist <name>

This command marks the begin of a netlist with the given name.

2.15 node <name> <celltype> [{ <port> <net> | <cfgport> <value> } ..]

This command adds a node (with the given name and cell type) to the current netlist. The remaining parameters specify the connections of cell ports to nets and the assignments of configuration values to cell ports declared using the "`cfg:<bit-width> <cfgport>`" syntax. Configuration values up to 64 bit can be specified in decimal, octal (with a leading "0") and hexadecimal (with a leading "0x"). Configuration values of any size can be specified in binary with a leading single quote.

This command can only be processed in setup mode.

2.16 mapnode <name> <cellnum>

This command maps a node in the current netlist to the specified cell. This is something the synthesis commands do and usually there is no need to do it manually. Therefore this is a command one will usually only see in InterSynth output files.

This command switches to synthesis mode.

2.17 shuffle seed <integer>

Re-initialize the internal random number generator using the given seed value.

2.18 shuffle cells [<conntype> [<tree>]]

Shuffles the cell to tree leaf mappings using the internal random number generator.

The optional `<conntype>` and `<tree>` arguments can be used to only shuffle the mappings for the specified conntype or only the specified tree within the conntype.

The `shuffle cells` command must not be used when generating a configuration for an existing interconnect using the `presilicon` command.

This command switches to synthesis mode.

2.19 shuffle nodes [type <celltype>] [netlist <netlist>]

Shuffles the node to cell mappings using the internal random number generator.

The optional arguments can be used to limit the shuffling to the specified celltype and/or netlist.

This command switches to synthesis mode.

2.20 presilicon [keepcells] <iterations> <hill-limit>

This command creates the interconnect from the loaded example netlists.

The optional `keepcells` argument disables the optimization for cell to leaf mapping. This can be used to investigate the level of optimization that can be achieved using this optimization by comparing the used interconnect resources with and without this option. Usually this option is used in conjunction with the `shuffle cells` command.

The `<iterations>` and `<hill-limit>` arguments configure how hard InterSynth tries to find an optimal solution. For both options a larger value causes the optimizer to use more CPU cycles but potentially yield a better result. Recommended values are 6 for `<iterations>` and 20 for `<hill-limit>`.

This command switches to synthesis mode.

2.21 postsilicon <iterations> <hill-limit> [<retries>]

This command creates an interconnect configuration for an existing interconnect (that has been generated using `presilicon` earlier).

The `<iterations>` and `<hill-limit>` arguments have the same meaning as for the `presilicon` command.

The optional `<retries>` argument causes InterSynth to restart the synthesis with a different random initial state for the given number of times before reporting a failed synthesis to the user. Without this option no retries are attempted when the synthesis fails.

This command switches to synthesis mode.

2.22 write_data { <filename> | - } [baseconfig] [cellmapping] [nodemapping] [allnetlists | netlist <name> ..]

This command writes the internal state of InterSynth to a file (or stdout if "-" is passed as filename). The syntax used for this files is the InterSynth command syntax. I.e. this data can be loaded into InterSynth by executing the files using the `load` command.

The optional arguments specify which part of the internal state should be written to the file. When none of this options are passed, all available data is written to the file.

This command switches to synthesis mode.

2.23 write_verilog { <filename> | - } [name <name>] [truth] [config]

This command writes the Verilog HDL code for the generated interconnect to the specified file.

The optional `name <name>` argument can be used to set the name of the generated Verilog module.

The optional `truth` argument generates (as additional Verilog module) a simple test case that attempts to generate the truth table for the generated module (only makes sense for example with a limited number of input pins and no internal state). This option is used to validate that InterSynth is working correctly in some of the original test cases. In real world applications there is most likely no use for this option.

The optional `config` argument generates (as additional Verilog module) a configuration core that can be used to configure the interconnect to any of the loaded example netlists. This may be useful for creating simple test cases when setting up a work flow that includes InterSynth.

This command switches to synthesis mode.

2.24 write_bitdata { <filename> | - } [netlist <netlist>]

This command creates a file containing the configuration bit-streams for all loaded netlists (or only the one specified using the `netlist <netlist>` option.

A work flow that includes InterSynth usually contains an application specific script that can convert the file generated by this command into an application specific bit-stream format.

The file generated by `write_bitdata` contains a lot of debug information. The relevant lines for generating a configuration bit-stream are the ones

starting with the `bitdata` keyword. E.g.:

```
bitdata size 503
bitdata bits   502:439  11111111111111111111111111111011000000000000000000000000000000100
bitdata bits   438:375  11111111111111111111111111111010000000000000000000000000000000010
bitdata bits   374:311  11111111111111111111111111111111111111111111111111111111111111101
bitdata bits   310:247  11111111111111111111111111111011000000000000000000000000000000100
bitdata bits   246:183  00000000000000000000000000000010111111111111111111111111111111111
bitdata bits   182:119  11110000001001010010110011001110000000001010100110011000000000000
bitdata bits   118:55   00000000000000001001110010010110001100111011011101011010111010111110
bitdata bits    54:0    01101000010001101010110100001010000100100100001110010000
```

The `bitdata size` record simple specifies the length of the bit-stream.

The `bitdata bits` records specify the actual bit data. Each of this records contain a fraction of the entire bit-stream in binary notation. The first argument gives the bit indices specified by this record in Verilog vector notation and the second argument specifies the actual bit data (as big endian binary string).

This command switches to synthesis mode. But as this command use useless before either the `presilicon` or the `postsilicon` command has been executed, InterSynth should be already in synthesis mode when this command is executed.

2.25 write_tikz { <filename> | - } <netlist> <conntype> <tree>

This command writes a LaTeXTikZ file that visualizes the generated interconnect and how it implements the given netlist, connection type and tree.

Again, this command switches to synthesis mode but should only be used after the `presilicon` or the `postsilicon` command has been executed anyways.

2.26 write_graphviz { <filename> | - } [<netlist> ..]

This command dumps a loaded netlist as graphviz `.dot` file. It is useful e.g. for debugging automatically generated net lists.

This command switches to synthesis mode.

2.27 write_html { <filename> | - }

This command write out an HTML file that visualizes the generated interconnect and how it is used by the different example netlists. This is particularly useful to analyze how InterSynth detects and common groups of cells in the given example netlists.

This command switches to synthesis mode.

3 Examples

FIXME