# RECMICRO

This tool provides an architecture agnostic methodology to reconfigure the design at pipeline stage level or thread level for DSP accelerators and microprocessor-based systems respectively. The tool has combination of python and C++ scripts. The pyhton scripts are mainly to handle text level parsing of netlist and light algorithms. However, the C++ scripts are mainly to handle the complicated graph related algorithms.

Fig. 1. Reconfiguration flow specifying the usage of scripts at every stage for pipeline and thread level reconfiguration

The complete flow and usage of scripts for pipeline and thread level reconfiguration is shown in Fig. 1. The green colour in the flow represents the scripts (or output of the scripts) of our tool. However, the black colour represents the involvement of standard synthesis or PnR tool in the flow (e.g. design compiler and SoC encounter). The flow chart shows the seamless integration and distribution of scripts associated with our tool along with the commercially available EDA tools, hence easier to use.

## Scripts

1. **repipeline.py:** This script is used to perform repipelining in the design. Same script can be used for N-slowing [1] in the microprocessor. The aim of this step is to achieve the desirable maximum throughput in the design [3]. This step is always followed by retiming in the commercial EDA tools (see Fig.1), to achieve higher throughput design.

Inputs:

* ***driven\_wire\_file:***The list of wires connected to outputs of all the flops in the design (generated by netlist2graph.py script)
* ***driving\_port\_file:***The list of ports driving the wires in *driven\_wire\_file* (generated by netlist2graph.py script)
* ***library:*** name of the library from which extra flops have to be added
* ***Input\_verilog\_file:***gate level verilog netlist
* ***std\_cell\_technology:***the db format of the standard cell library
* ***no\_of\_pipeline:*** *number of pipelines to be inserted from the input*
* ***n:***maximum number of threads in thread level reconfiguration
* ***comb\_logic:***if the design to be repipelined is combinational design

Outputs:

* ***dc\_script :***Script to insert ‘*no\_of\_pipeline*’ pipelines from input or ‘*n’* level time multiplexing
* ***output\_verilog\_file:***The verilog file for repipelined design.

1. **netlist2graph.py:** This script converts the input structural gate-level verilog netlist to the graph data structure in the form of associative array as shown in example below.



Fig. 2. Example for netlist to graph conversion

Inputs:

* ***ip\_file:***Flattened gate level netlist
* ***ip\_tech\_file:***Standard cell verilog file
* ***async\_ports****:* list of asynchronous ports (e.g. reset ports, clocks)
* ***async\_ports\_cell:***list of asynchronous ports in standard cells (e.g. CLK, S, R)
* ***register\_iden\_pin:***name of the clock port in flip-flop standard cells (e.g. CLK)

Outputs:

* ***register\_file:***file containing all the flip-flop (instance names) in the design
* ***ip\_port\_file:***file containing all the input ports in the design (excluding async\_ports)
* ***op\_port\_file:***file containing all the output ports in the design
* ***cell\_ip\_port\_info\_file:***file containing the input ports of all the standard cells in the library
* ***cell\_op\_port\_info\_file:***file containing the output ports of all the standard cells in the library
* ***op\_graph\_file:***file containing the graph data structure in the following format:

<wire\_name1>, <wire\_name2> = <port1>,<inst\_name>,<mod\_name>,<port\_2>

* ***driven\_wire\_file:***The list of wires connected to outputs of all the flops in the design
* ***driven\_port\_file:***The list of ports driving the wires in driven\_wire\_file

Run: python netlist2graph.py

**NOTE: Restrict the flip-flops in standard cell library to very basic D flip-flop with and without asynchronous resets. Before running this step, ensure that the throughput requirement of the design is met. If throughput requirement not met, ensure the repipeline.py is run properly.**

1. **graph2skeletongraph.cpp:** converts the graph to skeleton graph which contains the basic information about the flip-flops in the design. This step is essential to reduce the size of the graph database in big designs. All the combinational logic between launching and capturing flop pair is reduced to a graph edge as shown in the example below:



Fig. 3. Example for graph to skeleton graph conversion

Inputs:

* ***ip\_port\_file:*** file containing all the input ports in the design (excluding async\_ports)
* ***op\_port\_file:***file containing all the output ports in the design
* ***register\_file:***file containing all the flip-flop (standard cells) in the design
* ***nl\_graph\_file:***file with the graph database created by netlist2graph.py

Outputs:

* ***skl\_graph\_file:***file containing the skeleton graph with the format:

<node1 >, <node2> = <edge\_weight>, where

<nodei> = input\_wire: flip\_flop\_name: output\_wire

* ***skl\_graph\_node\_file:*** file containing the nodes of the skeleton graph
* ***dummy\_nodes\_file:***file containing all the dummy nodes in the skeleton graph

Run: make graph2skeletongraph

1. **weighskeletongraph.py:** This file is used to fill the <edge\_weight> in skeleton graph created after running *graph2skeletongraph.cpp*. Edge weight is the critical path delay between the launching and capturing flip-flops represented by <node1> and <node2> respectively. This file invokes design compiler (therefore it is mandatory for user to have license of synthesis tools like design compiler to run this step), that populates the *edge\_weight* field by using the file *‘timing.tcl’* generated as an intermediate file for the step.

Fig. 4. Example for weighing the graph

Inputs:

* ***skeleton\_graph\_file:*** file containing the skeleton graph with ‘0’ initialized weights as created by ***graph2skeletongraph.cpp***
* ***input\_verilog\_file:***The gate level netlist of the original design
* ***ip\_port\_file:***file containing all the input ports in the design
* ***op\_port\_file:***file containing all the output ports in the design
* ***register\_file:***file containing all the flip-flop (standard cells) in the design

Outputs:

* ***skeleton\_graph\_file:*** updated input file with the weights

Run: python weighskeletongraph.py

1. ***identify\_pipeline\_linear.cpp:*** This script is used to identify the pipeline registers in the linear design [2]. This script is useful only when it is known that all the flip-flops in the design are pipeline flip-flops



Fig. 5. Example for linear pipeline identification

Inputs:

* ***skl\_graph\_file:***file containing the skeleton graph with the format:

<node1 > , <node2> = <edge\_weight>, where

<nodei> = input\_wire: flip\_flop\_name: output\_wire

* ***skl\_graph\_node\_file:*** file containing the nodes of the skeleton graph
* ***dummy\_nodes\_file:***file containing dummy nodes in the skeleton graph
* ***ip\_port\_file :***file containing all the input ports in the design
* ***op\_port\_file :***file containing all the output ports in the design

Outputs:

* ***registers\_to\_be\_bypassed\_file:*** file containing all the registers to be bypassed
* ***register\_level\_file:*** file containing all the registers and their corresponding pipeline number

**Run: make identifypipeline\_linear**

1. ***identify\_pipeline.cpp:*** This script is used to identify pipeline registers in the design by running the cut-set based algorithms. Post identification it chooses the most optimal set of registers to be bypassed based on random selection of initial register. This script should be used if not all the flip-flops in the design are known to be pipeline registers as shown in the example below:

 Fig. 6. Example for identification for all the pipelines in non-linear kind of architecture

Inputs:

* ***skl\_graph\_file:***file containing the skeleton graph with the format:

<node1 > , <node2> = <edge\_weight>, where

<nodei> = input\_wire: flip\_flop\_name: output\_wire

* ***skl\_graph\_node\_file:*** file containing the nodes of the graph only
* ***dummy\_nodes\_file:***file containing all the dummy nodes in the skeleton graph

Outputs:

* ***set\_of\_cutset\_file***: file containing all the set of cutsets present in the design
* ***loops\_file***: file containing all the loops present in the design.
* ***reg\_to\_be\_bypassed\_file:*** This file contains all the registers computed to be most optimal to be bypassable.
* ***reg\_not\_to\_be\_bypassed\_file:*** This file contains the registers in the design that are not replaced by bypassable registers

Run: make identifypipeline

1. ***bypassable\_reg\_replacement.py:*** This file is used to replace all the identified registers with the bypassable version of the registers as shown in the figure below:



Fig. 7. Bypassable register replacement

Inputs:

* ***replacement\_node\_file:*** The file obtained as a output of identify\_pipeline step. This file contains information about all the flip flops that has to be replaced by bypassable register version.
* ***input\_verilog\_file:*** The verilog file of the balanced retimed design that has to be reconfigured.
* ***std\_cell\_technology:*** The db (or lib) file used as link and target library in synthesis tool.
* ***bypassale\_flops\_file:*** The verilogfile with the basic flop with a multiplexer which combined together form bypassable registers
* ***no\_of\_flops\_per\_gate:*** The number of flip flops driven by a single clock gater.

Outputs:

* ***output\_verilog\_file:*** The final netlist of reconfigured design

1. ***twoslow2oneslow.py:*** This script is used to identify the flip-flops that has to be replaced with bypassable registers while creating reconfigurable thread design.

Input:

* ***graph\_file:*** file containing the graph data structure in the following format:

<wire\_name1>, <wire\_name2> = <port1>,<inst\_name>,<mod\_name>,<port\_2>

* ***reg\_file:*** file containing the instance of all the flip-flops in the design
* ***ip\_port\_file:*** file containing information about all the input ports in the design

Output:

* ***replacement\_node\_file:*** file containing the registers that can be replace by bypassable registers

## References

[1] N. Weaver, Y. Markovskiy, Y. Patel, J. Wawrzynek, “Post-placement C-slow retiming for the Xilinx Virtex FPGA,” in *FPGA*, 2003.

[2] S. Jain, L. Lin., M. Alioto, "Dynamically adaptable pipeline for energy-efficient microarchitectures under wide voltage scaling," *JSSC,* 2018.

[3] S. Jain, L. Lin., M. Alioto, “Automated Design of Reconfigurable Microarchitectures for Accelerators under Wide Voltage Scaling” in Transaction on very large scale integrated circuits *(TVLSI)*

[4] <https://www.eda.ncsu.edu/wiki/NCSU_EDA_Wiki> (open source pdk)