Power and Timing Optimization Using Multibit Flip-Flop

Sheng-Wei Yang, Jhih-Wei Hsu, Yu-Hsuan Cheng, Cindy Chin-Fang Shen

Synopsys, Inc.

0. Revision History

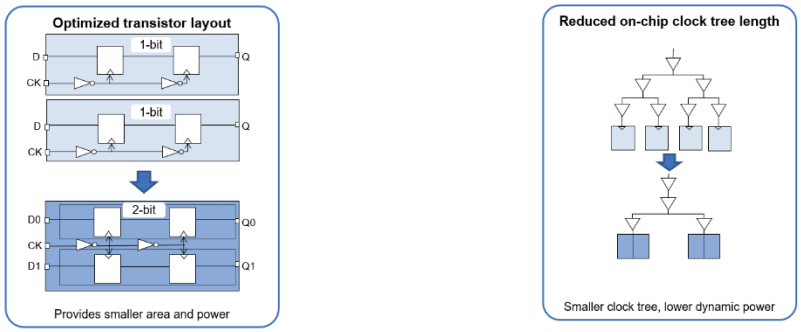
2025-08-20	-	Removed scan chain constraints. Update Fusion Compiler binary version.
2025-08-13	-	Revise data output format. Added OPERATION req. Change CPU to 16 cores
2025-08-01	-	Revise data input format. Align legality information with testcase.
2025-07-08	-	Fixed missing attribute in appended program format rule
2025-06-24	-	Appended program format rule
2025-05-29	-	Revised submit format and file content
2025-05-08	-	Revised testcase access procedure
2025-04-18	-	Added SDC as file input
2025-02-28	-	Added all registers as file input
2025-02-21	-	Second revision
2025-02-14	-	First revision
2025-02-07	-	Initial version

1. Introduction

With advances in modern technology nodes, minimizing power and area has become one of the most crucial challenges in the semiconductor industry. A widely adopted approach to address this challenge is to optimize the number and types of flip-flops, which are fundamental components in modern digital designs.

Substituting multiple single-bit flip-flops with a single multibit flip-flop is referred to as "multibit flip-flop banking." Since a multibit flip-flop requires less space than the individual flip-flops it replaces, more area can be saved. Fig. 1(a) illustrates this concept. Furthermore, each single-bit flip-flop has separate power, ground, and clock pin connections. Replacing them with a single multibit flip-flop helps streamline and reduce power, ground, and clock net routing complexity. Figure 1(b) provides an example. However, optimizing timing, power, and area has become increasingly complex for modern designs. For certain timing-critical nets, the initial approach of multibit flip-flop banking may degrade the timing, hindering the overall optimization goal. Therefore, in some cases, it becomes necessary to split a multibit flip-flop into several single-bit flip-flops to optimize timing-critical nets better. This technique is commonly known as "multibit flip-flop debanking."

This contest is an extension of the 2024 ICCAD CAD Contest Problem B. The contest's objective is to simulate multibit flip-flop banking and debanking decisions in virtual designs. Contestants must optimize timing, power, and area simultaneously while ensuring your program runs efficiently (i.e., with a short enough runtime to avoid penalties).



(a) (b)

Figure 1(a): A basic transistor layout featuring two single-bit flip-flops, along with one multi-bit flip-flop.

Figure 1(b): A diagram showing how the dynamic power consumption of a design is reduced by shortening the on-chip clock tree length, achieved by replacing four one-bit flip-flops with two two-bit flip-flops.

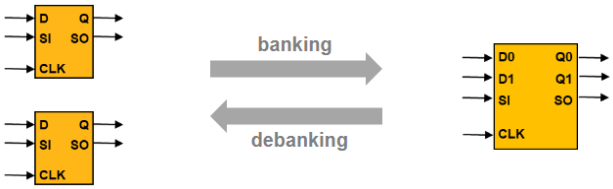


Figure 2: A simplified diagram of how two single-bit flip-flops can be banked into one two-bit flip-flop, and debanked vice versa. In this diagram, the upper left D pin of the single-bit flip-flop is mapped to the D0 pin of the two-bit flip-flop; the lower left D pin of the single-bit flip-flop is mapped to the D1 pin of the two-bit flip-flop.

2. Contest Objective

The goal is to develop a banking and debanking algorithm that produces a placement result optimized for timing, power, and area. Sequential cells are not allowed to overlap with any other cell in the placement result. The quality of the result is estimated by the cost function, which is the weighted summation of these three cost metrics: timing, power, and area, as follows:

$$\alpha \cdot TNS + \sum_{\forall i \in FF} \left(\beta \cdot Power(i) + \gamma \cdot Area(i)\right)$$

where i represents each flip-flop instance in the design, TNS denotes the total negative slack of all flip-flops, $Power(i)$ refers to the power consumption of the flip-flop, and $Area(i)$ represents the area cost of the flip-flop. α, β and γ indicate the weights for timing, power, and area, respectively. The values will be provided for each test case.

3. Problem Formulation and Input/Output Format

Given:

1. Weight factors for cost metrics
2. Timing slack and delay information
3. Power information
4. Area information
5. Cell placement result with flip-flop cells

Output:

1. A list of pin mapping between each input flip-flop pins and the output flip-flop pins
2. A flip-flop placement solution with its Verilog of netlist connections.

Note that flip-flops are not allowed to overlap with any other cell. Additionally, each flip-flop must be placed on the defined sites of the placement rows, while ensuring that the maximum placement utilization ratio is satisfied.

3.1 Format of Input Data

This section illustrates syntaxes for data input. Most of the syntaxes are floating point numbers. Contestants are expected to handle the data range within the limits of DBL_MAX.

There are two parts of files for each testcase:

- The first part is a file including weight factors for cost metrics, timing slack and delay information, power information, area information, and a list of all flip-flops. The syntaxes are described in Sections 3.1.1 to 3.1.4.
- The second part is design files containing cell placement results with flip-flops cells in LEF and DEF formats described in Section 3.1.5.

3.1.1 Weight factors for cost metrics

α, β, and γ values are given out as Alpha, Beta, and Gamma, respectively.

Syntax

Alpha <alphaValue>

Beta <betaValue>

Gamma <gammaValue>

Example

Alpha 1

Beta 5

Gamma 5

3.1.2 Timing slack and delay information

The design's timing slack and delay information will be reported using Fusion Compiler's report_qor command. The keyword TNS stands for the total negative slack value calculated across the current design. The larger the value means the worse the total negative slack is for this design.

Syntax

TNS <Total Negative Slack>

Example

TNS 0.13

3.1.3 Power information

The power information of the design will be reported using Fusion Compiler's report_power command. The keyword. TPO stands for total power value calculated across the current design. The larger the value means the worse the total power consumption it requires to run this design.

Syntax

TPO <Total Power>

Example

TPO 2.76

3.1.4 Area information

The design's area information will be reported using Fusion Compiler's report_qor command. The keyword Area value refers to the cell area in the design. A larger value indicates a larger total cell area required to tape out the design.

Syntax

Area <Cell Area>

Example

Area 19.58

3.1.5 Cell placement result with flip-flops cells

The cell placement results, including flip-flop cells, will be provided in LEF/DEF format. A Synopsys Design Constraint (SDC) file will be included to represent timing information. For the detail format of LEF DEF, please refer to LEF/DEF Language Reference[5]. For the detail format of Synopsys Design Constraints (SDC), please refer to Synopsys Timing Constraints Manager [6], or some other tutorial documents regarding the usage of SDC. [7] [8]

3.2 Format of Output Data

The output contains a cell placement result with flip-flop cells in the same format as the input data. Additionally, contestants need to provide a list detailing the changed cells and their corresponding pin mappings. Only the resultant cells and their mappings should be output, excluding intermittent cells or any combinational gates. The number of cell instances may vary based on the banking and debanking methods used to generate the output. All sequential instances must be placed on-site, with the lower-left corner of each cell aligning with the lower-left corner of a placement site, ensuring that no flip-flop overlaps with any other cell.

For each test case, three parts of the output file are expected:

- The first part is a file listing the pin mappings between each input flip-flop's pins and the output flip-flop's pins using the syntaxes described in section 3.2.1.
- The second part is a list of all OPERATIONs performed. The details are described in 3.2.2.
- The third part is the design files containing the flip-flop placement solution described in section 3.2.3.

3.2.1 A list of pin mapping between each input flip-flop pins and the output flip-flop pins

This file contains all the information regarding the mapping from the original input to the output. The keyword CellInst refers to the total number of flip-flop cell instances in the output design. For each flip-flop cell instance in the input design with a valid data connection, we expect a 1-to-1 mapping of flip-flop pins between the input and output designs.

Syntax

CellInst <InstCount>

<originalCellPinFullName> map <resultCellPinFullNameName>

Example

CellInst 2

C1/D map C6/D

C1/Q map C6/Q

C1/CLK map C6/CLK

C2/D map C5/D1

C2/Q map C5/Q1

C2/CLK map C5/CLK

C3/D map C5/D0

C3/Q map C5/Q0

C3/CLK map C5/CLK

3.2.2 A list of Operation Log Format

In addition to the pin mapping information specified in Section 3.2.1, contestants are required to provide a detailed operation log that documents the sequence of multibit flip-flop banking and debanking operations performed during optimization. This operation log enables proper validation of the transformation process and ensures the correctness of the final result.

The operation log must be included as part of the .list output file, following the pin mapping section described in Section 3.2.1.

Syntax

OPERATION <OperationCount>

<operation_1>

<operation_2>

...

<operation_n>

Where:

- <OperationCount> represents the total number of operation steps performed
- An operation step may consist of one or more atomic operations separated by semicolons.
- Each atomic operation can be one of the following types:
  - create_multibit: Banking operation that combines multiple single-bit flip-flops into a single multibit flip-flop
  - split_multibit: Debanking operation that splits a multibit flip-flop into multiple single-bit flip-flops
  - change_name: Renaming operation for flip-flop instances
  - size_cell: Library cell substitution operation
- Each operation step represents one banking, debanking, sizing, or renaming move.
- Each operation step must either contain a single atomic operation of type change_name or size_cell, or multiple atomic operations of type create_multibit or split_multibit separated by semicolons.

Operation Types

1. **create_multibit**: Combines multiple single-bit flip-flops into a larger multibit flip-flop

Syntax:

```
   create_multibit { {inputFF1_name inputFF1_lib inputFF1_width}
{inputFF2_name inputFF2_lib inputFF2_width} ... {outputFF_name outputFF_lib
outputFF_width} }
```

Parameters:

- Input flip-flops: List of {instance_name, library_name, bit_width} for all input flip-flops.
    - bit_width of every input flip-flop should be 1.
    - instance_name must be full name, including all hierarchy from top to bottom instance.
- Output flip-flop: Single {instance_name, library_name, bit_width} for the resulting multibit flip-flop.
    - instance_name must be full name, including all hierarchy from top to bottom instance.
    - The new instance_name must not duplicate with any existing instance_name in the design.
    - The hierarchy must not be changed.
- The output bit_width must equal the sum of all input bit_widths.
- The cell order of the output flip-flops represents the bits from lowest to the highest of the input flip-flop. Please order the output exactly in accordance to the intended pin order.

2. **split_multibit**: Splits a multibit flip-flop into multiple single-bit flip-flops

Syntax:

```
split_multibit { {inputFF_name inputFF_lib inputFF_width} {outputFF1_name outputFF1_lib outputFF1_width} {outputFF2_name outputFF2_lib outputFF2_width} ... }
```

Parameters:

- Input flip-flop: Single {instance_name, library_name, bit_width} for the multibit flip-flop to be split.
    - instance_name must be full name, including all hierarchy from top to bottom instance.
- Output flip-flops: List of {instance_name, library_name, bit_width} for all resulting flip-flops.
    - bit_width of every output flip-flop should be 1.
    - instance_name must be full name, including all hierarchy from top to bottom instance.
    - The new instance_name must not duplicate with any existing instance_name in the design.
    - The hierarchy must not be changed.
- The input bit_width must equal the sum of all output bit_widths.
- The cell order of the output flip-flops represents the bits from lowest to the highest of the input flip-flop. Please order the output exactly in accordance to the intended pin order.

3. **change_name**: Renames a flip-flop instance

Syntax:

```
change_name {old_instance_name new_instance_name}
```

Parameters:

- Input old_instance_name: Original instance name.
    - old_instance_name must be full name, including all hierarchy from top to bottom instance.
- Output new_instance_name: New instance name to replace the original instance name.

    - The new_instance_name must not duplicate with any existing instance_name in the design.
    - new_instance_name must be full name, including all hierarchy from top to bottom instance.
- The hierarchy should not be changed. Only the bottom instance_name is allowed to be changed.

For example:

"change_name {hier1/hier2/A hier1/hier2/B}" is allowed.

"change_name {hier1/hier2/A hier1/B}" is forbidden.

"change_name {hier1/hier2/A hier1/hier3/B}" is forbidden.

4. **size_cell**: Changes the library cell type of a flip-flop instance

Syntax:

```
size_cell {instance_name old_library_name new_library_name}
```

Parameters:

- Input instance_name: Flip-flop instance_name.
    - instance_name must be full name, including all hierarchy from top to bottom instance.
- Input old_library_name: The flip-flop instance's original library name.
- Output new_library_name: New library name to replace the original instance name.
    - The new_library_name must be one of the existing library_names.

Important Constraints

1. **Clock Domain Consistency**: All flip-flops involved in a banking or debanking operation must belong to the same clock domain

2.. **Dummy Instance Naming**: When intermediate operations require temporary instances, use the naming convention `dummy_<index>` with library name `dummy_lib_cell`. A dummy cell must be a single-bit flip-flop. Each `dummy_<index>` must be unique within the .list file. Dummy cells only exist as intermediate instances during an OPERATION and must not remain at the end of any OPERATION line.

3. **Single-bit Width**: All single-bit flip-flops must have bit_width = 1

4. **Functional Equivalence**: All D and Q pin connections must remain functionally equivalent after transformations

5. **Cell order represents the pin order**: The order of the parameters in create_multibit and split_multibit represents the pin order.

6. There will be not scan chain in this contest.

Example 1: Banking Two 4-bit Flip-flops into One 8-bit Flip-flop

Given:

- 4-bit flip-flop `hier1/A` with library `libA`
- 4-bit flip-flop `hier1/B` with library `libB`
- Target: 8-bit flip-flop `hier1/C` with library `libC`

Pin Mapping (Sequential), followed by operation log. Please note that the operation is written in one line:

```
CellInst 1

hier1/A/Q0 map hier1/C/Q0

hier1/A/Q1 map hier1/C/Q1

hier1/A/Q2 map hier1/C/Q2

hier1/A/Q3 map hier1/C/Q3

hier1/B/Q0 map hier1/C/Q4

hier1/B/Q1 map hier1/C/Q5

hier1/B/Q2 map hier1/C/Q6

hier1/B/Q3 map hier1/C/Q7

OPERATION 1

split_multibit { {hier1/A libA 4} {dummy_0 dummy_lib_cell 1} {dummy_1
dummy_lib_cell 1} {dummy_2 dummy_lib_cell 1} {dummy_3 dummy_lib_cell 1} };
split_multibit { {hier1/B libB 4} {dummy_4 dummy_lib_cell 1} {dummy_5
dummy_lib_cell 1} {dummy_6 dummy_lib_cell 1} {dummy_7 dummy_lib_cell 1} };
create_multibit { {dummy_0 dummy_lib_cell 1} {dummy_1 dummy_lib_cell 1}
{dummy_2 dummy_lib_cell 1} {dummy_3 dummy_lib_cell 1} {dummy_4 dummy_lib_cell
1} {dummy_5 dummy_lib_cell 1} {dummy_6 dummy_lib_cell 1} {dummy_7
dummy_lib_cell 1} {hier1/C libC 8} }
```

Example 2: Banking with Interleaved Pin Mapping

Pin Mapping (Interleaved), followed by operation log. Please note that the operation is written in one line:

```
CellInst 1

hier1/A/Q0 map hier1/C/Q0

hier1/A/Q1 map hier1/C/Q2

hier1/A/Q2 map hier1/C/Q4
```

```
hier1/A/Q3 map hier1/C/Q6

hier1/B/Q0 map hier1/C/Q1

hier1/B/Q1 map hier1/C/Q3

hier1/B/Q2 map hier1/C/Q5

hier1/B/Q3 map hier1/C/Q7

OPERATION 1

split_multibit { {hier1/A libA 4} {dummy_0 dummy_lib_cell 1} {dummy_1
dummy_lib_cell 1} {dummy_2 dummy_lib_cell 1} {dummy_3 dummy_lib_cell 1} };
split_multibit { {hier1/B libB 4} {dummy_4 dummy_lib_cell 1} {dummy_5
dummy_lib_cell 1} {dummy_6 dummy_lib_cell 1} {dummy_7 dummy_lib_cell 1} };
create_multibit { {dummy_0 dummy_lib_cell 1} {dummy_4 dummy_lib_cell 1}
{dummy_1 dummy_lib_cell 1} {dummy_5 dummy_lib_cell 1} {dummy_2 dummy_lib_cell
1} {dummy_6 dummy_lib_cell 1} {dummy_3 dummy_lib_cell 1} {dummy_7
dummy_lib_cell 1} {hier1/C libC 8} }
```

Example 3: Multi-step Optimization Process

Pin Mapping (Reversed), followed by operation log:

```
CellInst 1

hier1/A/Q0 map hier1/C/Q7

hier1/A/Q1 map hier1/C/Q6

hier1/A/Q2 map hier1/C/Q5

hier1/A/Q3 map hier1/C/Q4

hier1/B/Q0 map hier1/C/Q3

hier1/B/Q1 map hier1/C/Q2

hier1/B/Q2 map hier1/C/Q1

hier1/B/Q3 map hier1/C/Q0

OPERATION 3

split_multibit { {hier1/A libA 4} {hier1/D libD 1} {hier1/E libD 1} {hier1/F
libD 1} {hier1/G libD 1} }

split_multibit { {hier1/B libC 4} {hier1/H libD 1} {hier1/I libE 1} {hier1/J
libF 1} {hier1/K libG 1} }

create_multibit { {hier1/K libG 1} {hier1/J libF 1} {hier1/I libE 1} {hier1/H
libD 1} {hier1/G libD 1} {hier1/F libD 1} {hier1/E libD 1} {hier1/D libD 1}
{hier1/C libC 8} }
```

Validation Requirements

The operation log will be validated to ensure:

1. **Conservation of Connectivity**: Total number of flip-flop bits is preserved throughout all operations

2. **Library Compatibility**: All referenced library cells must exist in the provided technology libraries, except for dummy cell.

3. **Clock Domain Constraints**: Banking and debanking operations must respect clock constraints

Integration with Output Files

The complete .list file format now includes both pin mapping and operation log:

```
CellInst <InstCount>
<pin_mapping_1>
<pin_mapping_2>
...
<pin_mapping_n>

OPERATION <OperationCount>
<operation_1>
<operation_2>
...
<operation_m>
```

This operation log format ensures full traceability of the optimization process and enables comprehensive validation of the banking and debanking decisions made by contestant algorithms.

3.2.3 A flip-flop placement solution

A flip-flop placement solution is expected to be presented in the format of LEF DEF and Verilog. For the detail format of LEF DEF, please refer to LEF/DEF Language Reference[5].

4. Example

We take the following circuit as a sample input:

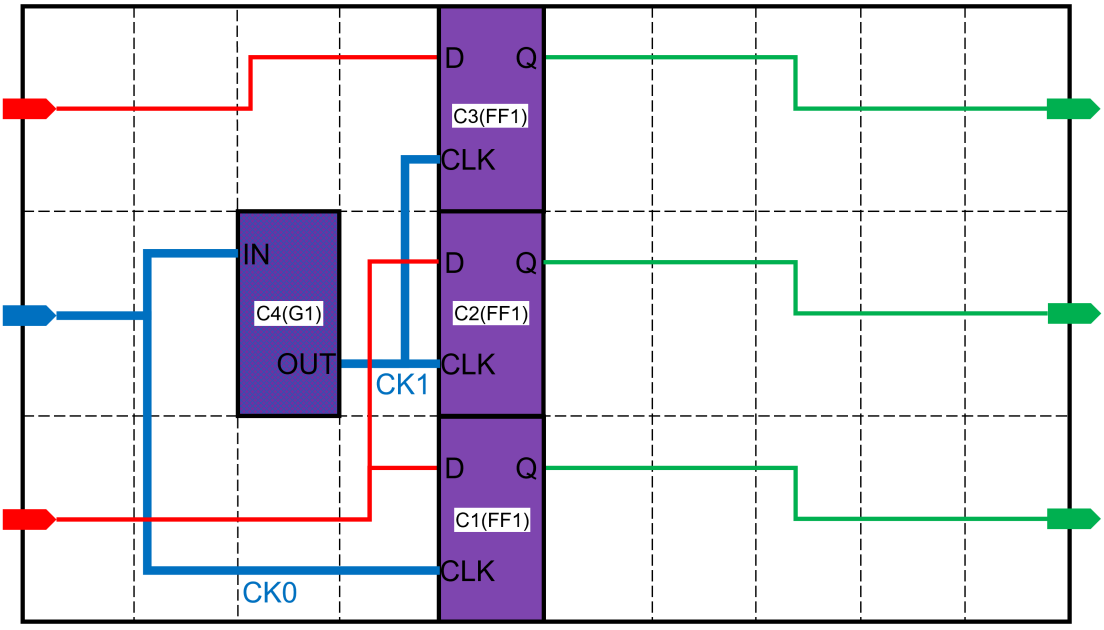


Figure 3: A visualization of the example data input circuit. CK0 and CK1 are two different clock nets.

Input File 1: Syntax Data:

Alpha 1

Beta 5

Gamma 5

TNS 0.13

TPO 2.76

Area 19.58

Input File 2:

Cell placement result with flip-flops cells (in LEF DEF)

A sample output:

Based on the given input, here's an example of output:

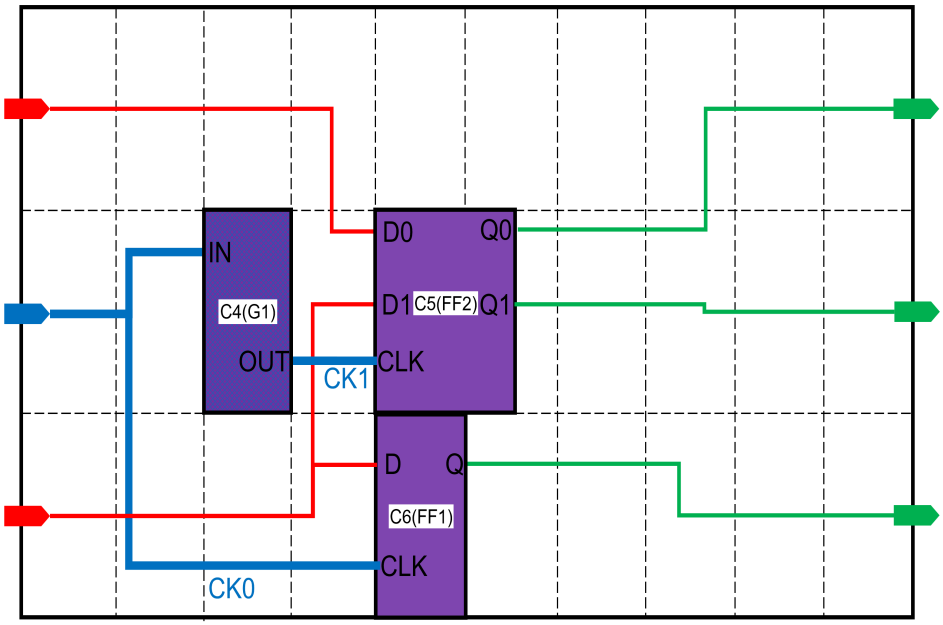


Figure 4: A visualization of the example data output circuit. CK0 and CK1 remain two different clock nets.

Output File 1: Syntax Data

CellInst 2

C1/D map C6/D

C1/Q map C6/Q

C1/CLK map C6/CLK

C2/D map C5/D1

C2/Q map C5/Q1

C2/CLK map C5/CLK

C3/D map C5/D0

C3/Q map C5/Q0

C3/CLK map C5/CLK

OPERATION 2

change_name {C1 C6}

create_multibit { {C3 FF1 1} {C2 FF1 1} {C5 FF2 2} }

Output File 2:

A flip-flop placement solution (outputFileName.def and outputFileName.v)

5. Evaluation

The Fusion Compiler used for alpha and beta evaluation is W-2024.09-SP3 20240824 build. The expected output must satisfy the following constraints. The satisfiability of these constraints would be evaluated via the latest release of Synopsys Fusion Compiler.

Failure to meet any of these constraints will result in no score:

A. All sequential instances must be placed on-site and within the die region.
B. Sequential instances must not overlap with any other cell.
C. Nets connected to the flip-flops must remain functionally equivalent to the data input after banking or debanking. Specifically, all D pin and Q pin connections must remain functionally equivalent, and all CLK pins must be connected to the same clock net. The result should not leave any open or short nets.
D. A list of banking and debanking mapping needs to be provided

- Banking Criteria:
  Contestants are allowed to perform banking by clustering several lower-bit flip-flops to one higher-bit flip-flop. However, all D pin and Q pin connections need to remain functionally equivalent after banking, and all CLK pins need to be connected to the same clock net.
- Debanking Criteria:
  Contestants are allowed to perform debanking by split a higher-bit flip-flop to several lower-bit flip-flops. However, all D pin and Q pin connections need to remain functionally equivalent after debanking, and all CLK pins need to remain connected to the same clock net.

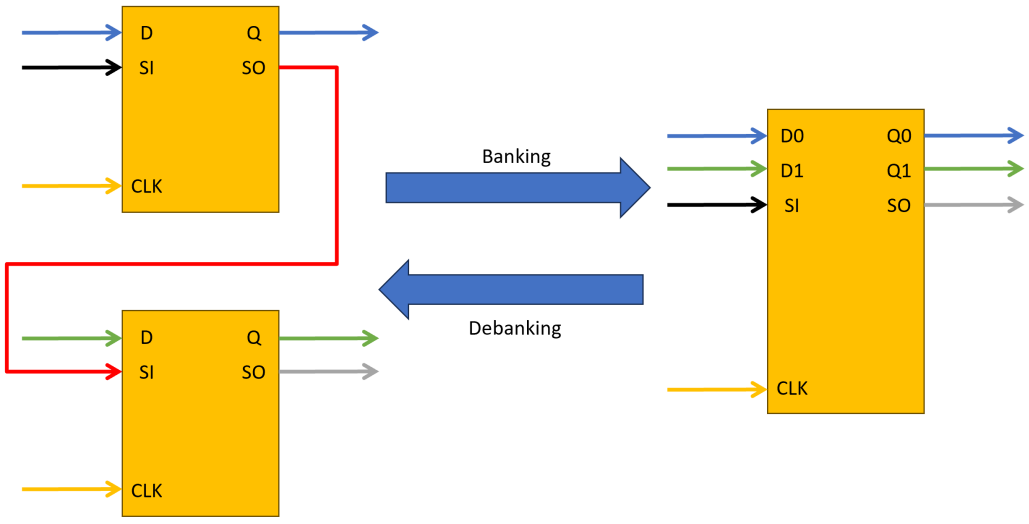


Figure 5: A banking and debanking example involving design for testability (DFT) pins.

Figure 5 provides an example to illustrate the criteria. A scan chain is formed starting from the Scan-Input (SI) pin of the upper-left flip-flop. The Scan-Output (SO) pin of the upper-left flip-flop is then

connected to the SI pin of the bottom-left flip-flop, and subsequently to the SO pin of the bottom-left flip-flop. Therefore, when banking the two single-bit flip-flops, the banked multibit flip-flop should map the SI pin of the upper-left flip-flop to the SI pin of the multibit flip-flop, and the SO pin of the bottom-left flip-flop to the SO pin of the multibit flip-flop. When debanking from the multibit flip-flop, a scan chain is formed from the SI to SO pins of the multibit flip-flop. Therefore, when debanking the multibit flip-flop into single-bit flip-flops, the SO pin of the debanked single-bit flip-flop in the upper-left corner should be connected to the SI pin of the debanked single-bit flip-flop in the bottom-left corner.

5.1 Overview of Score Function

The quality of your solution is evaluated by considering both the optimization of timing, power, and area, as well as the efficiency of your code. The formula for calculating the final score is:

$$\boldsymbol{Final\ score} = Initial\ score\ \times (1.0 + runtime\ factor\ bounded)$$

"$Initial\ score$" reflects the optimization of timing, power, and area. It is calculated as:

$$Initial\ score = \alpha \cdot TNS + \sum_{\forall i \in FF} \big(\beta \cdot Power(i) + \gamma \cdot Area(i)\big)$$

TNS indicates how well the timing of your design is optimized. It is reported using the command "***report_qor***" from Fusion Compiler

$Power$ stands for total power consumption of your design. It is reported using the command "***report_power***" from Fusion Compiler.

$Area$ stands for total area of cells. It is reported using the command "***report_qor***" from Fusion Compiler.

"$runtime\ factor\ bounded$" reveals the efficiency of your code. To prevent large changes in the final score, the "$runtime\ factor$" is bounded within a ±10% range as follows:

$$runtime\ factor\ bounded = max\big(-0.1, min(0.1, runtime\ factor)\big)$$

"$runtime\ factor$" adjusts the score based on the performance of your code (i.e., how fast or slow it runs compared to an expected reference time). It is estimated as follows:

$$runtime\ factor = \ 0.02 \times \log_2\left(\frac{elapse\ time\ of\ test\ binary}{median\ elapse\ time}\right)$$

where

- elapsed time of test binary is the runtime took by your code
- median elapsed time is the reference time based on the median execution time of all contest submissions.

Specifically, if the binary is 2x faster or slower than the expected runtime, the score will increase or decrease by 2% of the initial score. If the binary runs 4x faster or slower, the score will adjust by ±4%.

Key Considerations:

- CPU Cores: You have 16 CPU cores available for your program.

- Time Limit: The evaluation machine is limited to 60 minutes for each test case. The hidden cases will be in the same scale as public cases.

5.2 Program Requirements

The contestant's submitted program should be able to execute as follows

```
./cadb_0000_alpha
    -weight <weightFile>
    -lib <libFile1> <libFile2> ...
    -lef <lefFile1> <lefFile2> ...
    -db <dbFile1> <dbFile2> ...
    -tf <tfFile1> <tfFile2> ...
    -sdc <sdcFile1> <sdcFile2> ...
    -v <verilogFile1> <verilogFile2> ...
    -def <defFile1> <defFile2> ...
    -out <outputName>
```

Notes:

1. Separate multiple files for each parameter with a space.

2. The order of parameters is flexible, but each parameter must be followed by its corresponding files or values.

This program format rule is recommended for alpha testing and will be enforced for beta and final evaluation. Contestants submitting alpha testing, if failed to abide by the requested program format rule, are still eligible for evaluation as long as a **README** file with detailed description on how to run the submission is provided along with the submission. Failure to follow the program format will not be eligible for scoring in beta and final submission.

outputName represents the expected output file name and will be the last parameter when executing the contestant's program. Contestants are expected to output their results as follows:

```
outputName.list outputName.v outputName.def
```

`outputName.list` represents the output syntax data, including that covered in section 3.2.1.

outputName.def and outputName.v represent the flip-flop placement solution and its netlist connection covered in section 3.2.2. Please follow the naming convention by prefixing your binary as cadb_0000*, where 0000 is the placeholder of contestants serial number. Please replace 0000 with your registered team number.

Please follow the naming convention by prefixing your binary as cadb_0000*, where 0000 is the placeholder of contestants serial number. Please replace 0000 with your registered team number.

The *_alpha stands for the alpha testing stage. For each stage please specify with different postfixes (*_beta, *_final).

Please note that contestants should follow the naming description; otherwise, the score will be annulled in beta and final submission.

Contestants will have access to the machine provided by ICCAD. They must upload their binary along with the necessary modules to run the binary. Contestants should note that the score will be marked as a failure if the public account cannot run its binary for any reason.

6. Access to Testcase

The testcases are generated based on Synopsys technology. Therefore, contestants are required to register at Synopsys SolvNetPlus website to receive the testcases:

Please sign-up to the contest first. Access to the testcase will be distributed after 5/18 and will take up to 7 - 10 work days.

7. References

[1] 2024 ICCAD CAD Contest Problem B: https://www.iccad-contest.org/2024/Problems.html

[2] Ya-Chu Chang, Tung-Wei Lin, Iris Hui-Ru Jiang, and Gi-Joon Nam. 2019. Graceful Register Clustering by Effective Mean Shift Algorithm for Power and Timing Balancing. In Proceedings of the 2019 International Symposium on Physical Design (ISPD '19).

[3] Meng-Yun Liu, Yu-Cheng Lai, Wai-Kei Mak, and Ting-Chi Wang. 2022. Generation of Mixed-Driving Multi-Bit Flip-Flops for Power Optimization. In Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22).

[4] Gang Wu, Yue Xu, Dean Wu, Manoj Ragupathy, Yu-yen Mo, and Chris Chu. 2016. Flip-flop clustering by weighted K-means algorithm. In Proceedings of the 53rd Annual Design Automation Conference (DAC '16)

[5] LEF/DEF Language Reference https://www.ispd.cc/contests/18/lefdefref.pdf

[6] Synopsys Timing Constraints Manager https://www.synopsys.com/verification/static-and-formal-verification/timing-constraints-manager.html

[7] Synopsys® Design Constraints File (.sdc) Definition https://www.intel.com/content/www/us/en/programmable/quartushelp/current/index.htm#reference/glossary/def_sdc.htm

[8] Synopsys Design Constraints (SDC) Basics https://www.vlsi-expert.com/2011/02/synopsys-design-constraints-sdc-basics.html

[9] Synopsys' Open-Source Liberty Format to Incorporate On-Chip Variation Extensions https://news.synopsys.com/home?item=123415