Correct high level synthesis of triple modular redundant user circuits for FPGAs

Michael Bernardi[1] Ediz Cetin[2] Oliver Diessel[3]

[1] University of New South Wales, Australia
`mrmbernardi@gmail.com`
[2] Macquarie University
`ediz.cetin@mq.edu.au`
[3] University of New South Wales, Australia
`o.diessel@unsw.edu.au`

Technical Report
UNSW-CSE-TR-201804
October 2018

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

This report outlines the HLS tool *TLegUp*, an extension of *LegUp* that produces circuits with TMR for use in space-based applications. We cover the background and motivation behind the project and the need to show that the software produces correct output in order to verify the current implementation and provide a framework for future development. A three-tiered approach for validating correctness is described and the top two tiers are applied to *TLegUp*. Finally, this report explores results of the validation and future goals and considerations for the project.

This work is substantially based on the 4th year Bachelor of Engineering thesis by Michael Bernardi.

Acknowledgements

The authors acknowledge and thank the contributions of Ganghee Lee, Dimitris Agiakatsikas and Tong Wu for advice in carrying out this study.

Abbreviations

DFG Data Flow Graph

DFS Depth First Search

FPGA Field Programmable Gate Array

HLS High Level Synthesis

LLVM Low Level Virtual Machine (an open-source compiler framework)

LLVM IR LLVM Intermediate Representation

MER Module-based Error Recovery

RTL Register-Transfer Level

SEU Single Event Upset

SRAM Static Random Access Memory

TMR Triple Modular Redundancy

1 Introduction

SRAM-based Field-Programmable Gate Arrays (FPGAs) are attractive components for space-based applications since they are inexpensive, easily reconfigured, and have low power usage coupled with high performance. In space, however, SRAM FPGAs are vulnerable to errors known as single-event upsets (SEUs) [OCG+09]. SEUs are the result of charged particles interfering with the circuit hardware. They have the capacity to change the output of a circuit by flipping bits in internal memory or by creating voltage transients that then propagate as errors through the logic.

SEUs can affect FPGAs in two different ways depending on whether they cause errors in user-defined circuitry or in configuration memory. SEUs have temporary effects on cycle-free user-defined ciruitry. After an upset, the circuit will again produce correct output if provided with fresh error-free input. If the upset affects configuration memory it alters the function of the circuit. An altered circuit will continue to produce erroneous output until the correct configuration is restored.

A preferred method of mitigating the risk of SEUs is implementing circuits with triple modular redundancy (TMR), and then periodically rewriting their configuration memory [Car00]. TMR involves triplicating the logic into three identical modules and then selecting the output using a majority vote. This allows an error occurring in user-defined logic to be masked provided that there are no simultaneous errors in the other two modules. Rewriting configuration memory is then necessary to correct any errors resulting from altered configuration memory.

TLegUp is a program that generates circuit designs with TMR from high level *C* code using a technique called high-level synthesis (HLS) [LAW+17]. The circuits generated are represented by *Verilog* register-transfer level (RTL) code. While there are other tools that use HLS to generate RTL code, and tools that add TMR to circuits described in RTL code, there are no tools that can do both at the same time, and this is the basis for *TLegUp*. It is intended to be a productivity tool that allows users to generate reliable circuit designs for space applications directly from *C* code with no intermediate steps.

Although this report is focused primarily on the verification of *TLegUp*, we also cover relevant information on the motivations and design of the software. Section 2 explains background concepts such as SEUs and HLS in further detail. Section 3 elaborates on the internal design of *TLegUp*. Sections 5 through 7 cover the verification process. Finally, section 8 outlines further work and considerations for the project and concludes the report.

2 Background

2.1 The Effects of SEUs

SEUs are the result of physical interactions with charged particles changing the state of circuit hardware. These particles can flip memory bits directly or create transients in user logic, both of which affect the output. New input to the circuit naturally overwrites existing errors, but only after they have already been propagated.

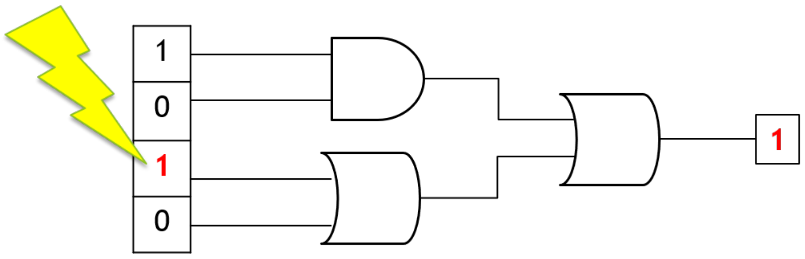


Figure 2.1: Diagram depicting a charged particle affecting a register and altering the output of subsequent logic.

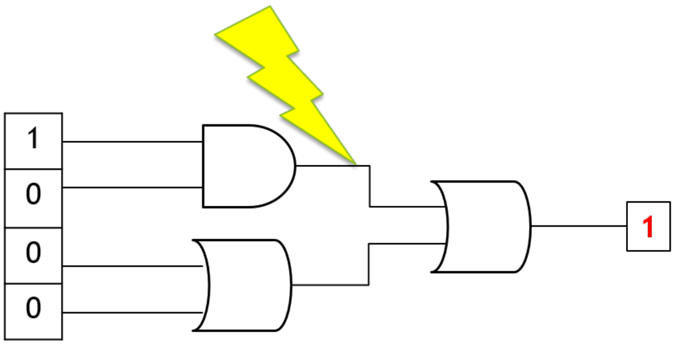


Figure 2.2: Diagram depicting a charged particle creating a voltage transient in a signal in the user logic and thus affecting the output of the logic.

The third way that an SEU can affect an SRAM-based FPGA is by corrupting memory which the FPGA uses to store configuration data. This results in the logic of the user circuit itself being modified. The function of the logic gates, the routing of signals, or any other property of the user-defined logic could be altered. Changes to the configuration bits persist until they are overwritten, and, unlike in the case of registers, configuration bits are not overwritten during normal operation of the FPGA.

2.2 Triple Modular Redundancy

TMR refers to the triplication of circuits into three discrete units, with redundancy added by having the three circuits vote on the output with a majority vote. For each bit of output from three modules x, y and z, the voting circuit evaluates $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$.

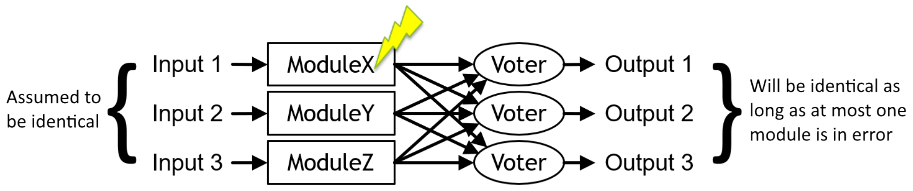


Figure 2.3: Diagram depicting a TMR system. In this case, ModuleX has been affected by an SEU and the voters will set all three outputs to that of Modules Y and Z.

In our case, we increase the reliability further by partitioning the circuit into a number of discrete partitions. Triplicating these partitions individually and voting on the output before it is passed on to another partition means that a failure requires errors in at least two of the three modules of the same partition. With many smaller partitions, the likelihood of this occurring is reduced [LAW$^+$17].

This technique alone is sufficient to handle SEUs that flip user registers or create transients in user logic, as it masks the errors before they reach the output while new input clears the original errors. However, this technique is not sufficient to deal with SEUs that affect configuration memory, nor will it correct errors that occur within feedback cycles in the circuitry.

To handle errors in configuration memory, the configuration bits may be periodically rewritten from a more robust memory source. This is known as *scrubbing*. Alternatively, the majority voters can include circuitry to detect repeat errors coming from one module and signal that it needs to be reconfigured. This is known as *Module-based Error Recovery* (MER) [THNCD17].

To handle errors in feedback cycles, a voter must be inserted somewhere in the cycle. Without such a voter, an error in a feedback cycle can persist in the state of the cycle and propagate from the cycle. We call these voters *synchronisation voters* as they serve to synchronise the state of a feedback cycle with that of its neighbouring modules.

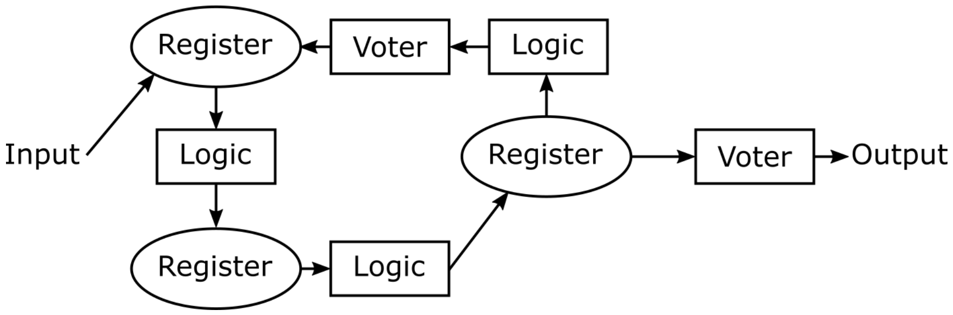


Figure 2.4: Diagram depicting a feedback cycle. The synchronisation voter depicted at the top centre of the cycle is required to prevent errors from propagating around the cycle indefinitely.

2.3 High Level Synthesis

HLS refers to the generation of circuit designs from high-level programming languages, such as *C*. It is typically accomplished by compiling code into a ‘formal model’, then running scheduling, and binding algorithms on that model [CGMT09]. Scheduling determines when each operation should take place in hardware, while binding assigns hardware resources to the operations. After these processes occur, the output is usually emitted as RTL code. RTL code describes circuits as registers and the combinational logic between them, which is a useful intermediate level of abstraction for FPGA circuit design.

2.4 Previous work

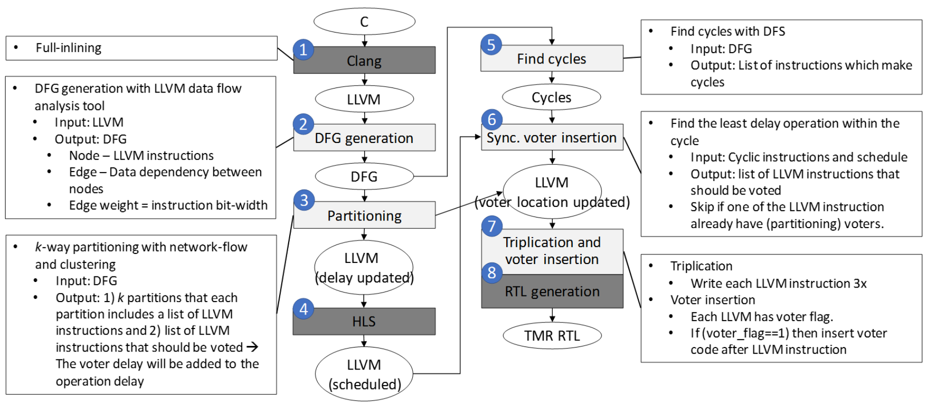
TLegUp [LAW+17] is a modified version of the open source HLS tool *LegUp* [CCA+11]. *LegUp* takes *C* code as input and outputs *Verilog* RTL code. The formal model used in *LegUp* is provided by *LLVM* [LLV], an open-source compiler framework. *LLVM* was chosen because the compiler architecture is easily extendible and mirrors the traditional steps of HLS. Our modifications to *LegUp* add processes that partition the circuit and determine where to add voters. The generation phase is modified to triplicate the design and add the voting circuits in the previously identified locations.

2.5 Motivation

The motivation behind *TLegUp* is to create a tool that uses HLS to generate TMR circuits. This will increase the productivity of users who otherwise would have to use multiple tools and be skilled at hardware design to accomplish this. Using HLS to create TMR circuits also holds the promise of producing such circuits more reliably and quickly, as well as producing more efficient or faster circuits than those produced by running a TMR tool after an HLS tool. By combining both processes, the voter insertion process is more flexible and informed, and operations can be scheduled more optimally to better suit the addition of voters.

3 TLegUp Design Flow

This section summarises each of the steps *TLegUp* takes to produce circuit design. We deal with version 1.0 of the software, which partitions a circuit at the LLVM instruction level. Later versions differ in that they are able to partition the algorithms at the function level rather than the instruction level. We provide a basic overview of the steps in the order they are executed. Refer to the *TLegUp* design flow document [Lee17] for further detail.



C
1 Clang
LLVM
2 DFG generation
DFG
3 Partitioning
LLVM (delay updated)
4 HLS
LLVM (scheduled)
5 Find cycles
Cycles
6 Sync. voter insertion
LLVM (voter location updated)
7 Triplication and voter insertion
8 RTL generation
TMR RTL
• Full-inlining
• DFG generation with LLVM data flow analysis tool
• Input: LLVM
• Output: DFG
• Node – LLVM instructions
• Edge – Data dependency between nodes
• Edge weight = instruction bit-width
• k-way partitioning with network-flow and clustering
• Input: DFG
• Output: 1) k partitions that each partition includes a list of LLVM instructions and 2) list of LLVM instructions that should be voted → The voter delay will be added to the operation delay
• Find cycles with DFS
• Input: DFG
• Output: List of instructions which make cycles
• Find the least delay operation within the cycle
• Input: Cyclic instructions and schedule
• Output: list of LLVM instructions that should be voted
• Skip if one of the LLVM instruction already have (partitioning) voters.
• Triplication
• Write each LLVM instruction 3x
• Voter insertion
• Each LLVM has voter flag.
• If (voter_flag==1) then insert voter code after LLVM instruction

Figure 3.1: TLegUp design flow diagram. Grey blocks represent original LegUp processes. [Lee17]

3.1 Compilation to LLVM

Firstly, the *C* code is compiled to an *LLVM* instruction listing using the *clang* compiler. This is an original *LegUp* step. This is done because *LLVM* operations are simple enough to directly correspond to hardware operations [CCA+11].

3.2 Data Flow Graph Generation

Next, a data flow graph is created from the *LLVM IR*. This is a step added in *TLegUp*. A data flow graph represents the *LLVM IR* in a graph form, in which individual nodes are operations and directed edges represent the transfer of inputs and outputs between operations. This is done because a data flow graph is a more useful representation of the code for partitioning and finding feedback cycles.

3.3 Partitioning

The graph is then partitioned into roughly equally-sized partitions, using number of operations as a metric. This is a *TLegUp* step. The number of partitions is specified by the user. In order to accomplish this, a network flow algorithm is used to find the minimum cut between partitions [LW98]. The parameter for cut size is the number of data bits, so the algorithm minimizes the size of the data connections between partitions. Operations that output from a partition are flagged for partition voter insertion, and the delay of these operations is increased to be consistent with the added delay of a voter.

3.4 HLS

With the delays of partition outputs updated, HLS algorithms are run. The original *LegUp* scheduling and binding algorithms are used. Scheduling is done with a simple 'as soon as possible' algorithm that schedules operations as soon as their dependencies are available. The binding algorithm then looks for large components, such as hardware dividers, to share between operations.

3.5 Feedback Cycle Identification

TLegUp must identify feedback cycles because there is a possibility for errors to persist in and propagate from these cycles if they do not contain a voter somewhere in the cycle (see Figure 2.4). A depth first search is run on the data flow graph to identify strongly-connected components, which are sub-graphs that contain a path from any node to any other node.

3.6 Synchronisation Voter Insertion

After the cycles have been identified, *TLegUp* must decide where best to insert a synchronisation voter in the cycle. This is a complex problem as cycles can share edges with other cycles, and the best edge to insert a voter on in order to minimize circuit delay and hardware resources may not be obvious. The current algorithm inserts the voter so as to minimise the increase to the critical path length.

3.7 RTL Generation

With all the HLS processes complete and the locations to insert voters determined, the *Verilog* output can be generated. This step relies on the original *LegUp* algorithm, but with some modifications. The modifications write the resultant RTL code block for each *LLVM* operation three times and insert a predefined voter circuit for instructions that have previously been flagged as requiring a voter. Voters are also inserted automatically in key locations, such as at the outputs of the top module and outputs from the memory bus.

4 Report Scope

This section describes the overall scope of this document.

4.1 Problem Statement

While *TLegUp* has performed well in preliminary runtime testing, little work has been done to verify that the processes and algorithms it uses are correct and will produce correct output in all cases. Undertaking such work would aid in finding limitations of the current implementation and correcting any mistakes in it. It would also aid in developing robust documentation and providing a framework for future development. Additionally, after this verification is complete, more work can be directed to various improvements in partitioning and other optimisations.

4.2 Verification of Correctness

A formal proof of each and every detail of the program would be too large an undertaking to be reasonable. Instead, we have opted to argue for correctness using a three-tiered approach, where each tier represents an argument at a certain level of abstraction. The top tier presents a case for the overall design flow of the program to be correct. It argues that each of the overarching abstract steps taken by *TLegUp* are necessary and sufficient for generating a correct circuit with TMR. The middle tier shows that the algorithms used to implement these steps satisfy the requirements of the steps outlined in the top tier, and that they are able to execute properly given input from previous steps and produce the correct output for subsequent steps. The bottom tier is required to conclude that the code implements the algorithms presented at the middle level correctly.

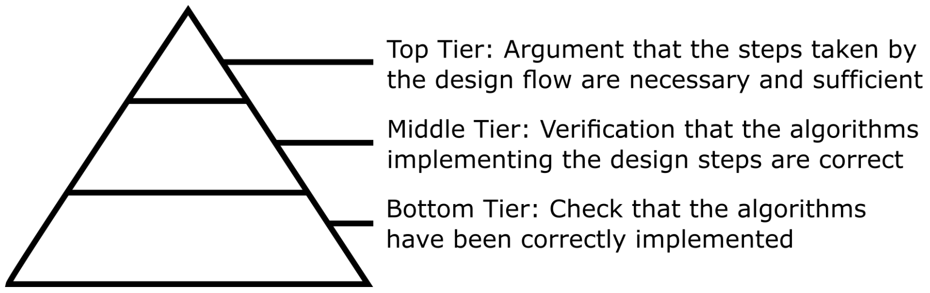

Top Tier: Argument that the steps taken by the design flow are necessary and sufficient
Middle Tier: Verification that the algorithms implementing the design steps are correct
Bottom Tier: Check that the algorithms have been correctly implemented

Figure 4.1: Three-tiered verification approach, with lower tiers expanding on the tiers above.

5 Top Tier - High-level Argument for Correctness

This section presents the argument for the top tier of the correctness proof for TLegUp.

5.1 TLegUp Goal

TLegUp is an extension of *LegUp*, a high-level synthesis (HLS) tool. The goal of *TLegUp* is to use *LegUp* to generate circuits with triple modular redundancy (TMR) from *C* programs. This process will produce better output compared to existing techniques where synthesis and triplication are isolated processes.

5.2 Definition of an Argument of Correctness

This section provides an argument of correctness for the processes used in *TLegUp.* It is different from a formal proof of correctness in that it does not seek to rigorously and formally prove all the details of the implementation. Rather, it seeks to provide an argument, from an abstract level, that the steps *TLegUp* takes produce a correct TMR circuit. To accomplish this, we explore what the necessary and sufficient requirements of TMR and HLS are, show that the steps *TLegUp* takes meet those requirements, and finally, show that these steps are consistent and do not conflict with each other.

5.3 Triple Modular Redundancy

Triple modular redundancy requires circuits to be triplicated and majority voters to be inserted at the outputs to mask errors in at most one of the copies. Starting with an existing circuit, simple triplication is trivial and is achieved through a process of copying the original circuit and inserting a majority voter at each output. This is sufficient for basic TMR. However, to increase reliability, we partition the circuit and apply TMR to each partition individually. This requires implementing a partitioning process. We must also consider feedback loops within the circuit. Each such cycle needs to contain at least one synchronisation voter to ensure that errors will not persist within and propagate from them. Further, these voters serve to synchronise the state of cycles in a module that has had its configuration rewritten to correct a configuration error. In order to insert voters into these cycles, another process must therefore identify where they occur.

Thus, we outline three necessary and sufficient processes to perform on a circuit to satisfy our partitioned implementation of TMR: partitioning, feedback cycle identification, and triplication with voter insertion. The final step relies on the completion of the first two.

5.4 High Level Synthesis

The goal of high-level synthesis is to create a circuit from a high-level language. In the case of *LegUp*, the high-level language is *C* and the circuit representation is Register Transfer Level (RTL) *Verilog.* The high level language is translated into a RISC-like instruction listing, and then the HLS is divided into three steps:

allocation, scheduling and binding. Allocation quantifies the hardware resources available on the target device. Scheduling determines when instructions should take place in hardware. Binding assigns the available registers and hardware resources to specific instructions. After these three steps are complete, a generation process builds the circuit representation from the information generated from the previous steps.

In our case, *LegUp* uses *clang* to generate *LLVM IR*, which is a low level intermediate representation. After allocation, *LegUp* then adds scheduling and binding information to the generated *LLVM* instruction listing. Finally, it uses the *LLVM* listing and the added information to generate *Verilog*.

Thus we identify five necessary processes to perform on the high-level code to generate a circuit representation: compilation, allocation, scheduling, binding, and generation. Each of these steps depends on the previous ones and must be completed in this order. As we know that *LegUp* and other HLS solutions work with these processes, we can conclude that they are sufficient.

5.5 Using HLS with TMR

We modify *LegUp* to produce TMR circuits by performing the TMR steps during the existing HLS process. The partitioning and feedback cycle identification steps are performed on a graph that is derived from the *LLVM* instruction listing rather than on a circuit representation, and they are done after the initial *clang* compilation. This is achieved by creating a data flow graph (DFG) from the instruction listing in order to partition it into roughly equally sized partitions, using a minimum cut algorithm on the graph. Following that, feedback cycles are identified as strongly connected components (SCCs) in the DFG using a depth first search originating at each node. The *LLVM* instructions at the locations where voters will be inserted are flagged. The information generated from these processes is independent from the existing allocation, scheduling, and binding steps. This means *LegUp* has been modified to perform them without affecting the correctness of its original HLS steps.

The final required step is triplication and voter insertion. This is done during the generation phase after all the prerequisite steps for both TMR and HLS have been completed. Triplication is performed by copying the existing output from *LegUp* while voter insertion is performed by adding predefined *Verilog* code representing a voting circuit at the output of the flagged *LLVM* instructions. Triplication itself has no effect on the functioning of the circuit, as each one of the three modules has the same components as the original circuit without TMR. On the other hand, voter insertion intends to modify the functioning of the circuit, and for it to be correct it must be shown that it does not result in a different output of the circuit compared to the original circuit without TMR. A voting circuit evaluates $xy \vee yz \vee xz$, where x, y, and z are its three inputs. Provided the overall circuit is error-free, a voting circuit's inputs are all equal, and its output takes the value of the inputs. This results in no change to the output. In the case where inputs are not equal, the voting circuit outputs the majority input.

5.6 Conclusion

This section shows that the steps added to *TLegUp* satisfy the requirements needed to generate a correct TMR circuit. They also do not conflict with the existing *LegUp* processes. Therefore, it is shown that the processes used by *TLegUp* correctly produce a TMR circuit from high-level code that is functionally similar to the non-TMR circuit produced by *LegUp*.

6 Middle Tier - Algorithmic Verification

This section describes and applies a verification method to the algorithms described in the high level argument.

6.1 Background

To complete the verification at this level, various existing verification techniques were examined. Many involved unit testing or other automated tools for dynamic analysis. Interest in these techniques was low as *TLegUp* had already undergone dynamic tests. On top of this, because these tests rely on running code, when a test is failed it is not possible to determine if the failure is due to a problem in the design of the algorithms, or a problem in their implementation. Further still, dynamic testing only catches errors which are handled by the tests, it does not uncover error conditions that have not been previously thought of. Reasoning about all possible error conditions and creating tests to catch them is too labour intensive for our needs. For these reasons, we restricted ourselves to static analysis.

Static analysis techniques were suggested in communications with Prof. Fethi Rabhi and A/Prof. Gerwin Klein of UNSW. These techniques generally involve rigorous mathematical formalisation of all components into a model, followed by a formal logical verification of the model based on some specified conditions, either by hand or with automated tools. Again, these techniques were considered to be far too labour intensive. There is also difficulty in being sure that the model and the specified conditions are congruent with the software being verified. For these reasons, we again restricted ourselves to informal techniques.

There are a number of techniques described in the literature that meet our restrictions, yet they often describe processes to be applied during construction rather than techniques to verify software that already exists. Therefore we describe a new method based on the existing ones that is suitable for application to our project.

6.2 Verification Method

We used the following method to verify that the algorithms implemented in the *TLegUp* design flow are correct. The method takes influence from Fagan inspections [Fag76] and a software design technique known as 'design by contract' [Mey92]. Both these methods are intended to be applied during the creation of software, but our method is applied to software after it has been written. Fagan's methods influence the process we use to review, including the roles of each person involved in the review. Design by contract gives us the idea of modularising the code and specifying requirements for those modules.

Firstly, formal requirements are described for each algorithm. These take the form of preconditions (input conditions) and postconditions (output conditions). These conditions are defined in terms of the data processed by an algorithm and must cover all assumptions made by it. The conditions describe the data at a reasonably high level. The requirements are decided upon collaboratively by all people involved in the review.

Secondly, pseudo-code is generated for each algorithm. The pseudo-code is generated by the author of the software. It should be a representation of the implementation code. It's the author's task to educate the other review members on how the pseudo-code works, and the purpose of its statements.

Thirdly, the pseudo-code is verified by ensuring that each algorithm meets each of its output requirements, and that the output requirements are congruent with the input requirements of the next algorithm. In our case, *TLegUp* uses some standard algorithms (e.g. network flow) that have already been reasoned about, so we can use some of the properties of those to simplify this phase of the procedure. This stage of the review is performed by an independent reviewer. In order to minimise bias, the author has no input. Any errors brought to light by this stage should be resolved by a moderator, independent again from the reviewer and author. Resolution requires changes to the pseudo-code and therefore it must be verified again.

The now verified pseudo-code can form the basis of the final verification phase, which will consist of verifying that the implemented code matches the pseudo-code. The reliability of our verification method depends on how robust the set of requirements are. It is necessary to make sure requirements cover all assumptions made by an algorithm.

In our specific case, the algorithms we intend to apply this method to are those described in the *TLegUp* design flow. Namely, they are: DFG generation, DFG partitioning, cycle finding, synchronisation voter insertion, and triplication.

6.3 Verification

DFG Generation

The purpose of this step is to generate a network flow graph compatible for use in the network flow algorithm described by Yang and Wong [YW96]. In our case, the graph represents the data flow of the compiled *LLVM* code, with nodes representing instructions and directed edges representing data dependencies. Nodes weights are proportional to the amount of logic assigned to each instruction whilst edge weights are proportional to the bit width of the data represented by the edge.

Requirements

Input

1. The input is a weighted and directed graph.
2. Every node represents an LLVM instruction.
3. Edges represent data dependencies between nodes.

Output

In the following section, 'X is connected to Y' means that X has an edge directed towards Y.

1. The output must be a weighted and directed graph.
2. For every node V_i in the input graph, there is a node V_o in the output graph.

3. For every node V_o, there are two additional bridging nodes N_1 and N_2 in the output graph.
4. V_o is connected to N_1 with an edge weight of infinity.
5. N_1 is connected to N_2 with an edge weight proportional to the bit-width of the data leaving V_o.
6. N_2 is connected with edge weights of infinity to all nodes in the output graph that correspond to children of V_i.
7. All nodes in the output graph that correspond to children of V_i are connected to N_1 with edge weights of infinity.
8. N_2 is connected to V_o with an edge weight of infinity.
9. No other edges are present.

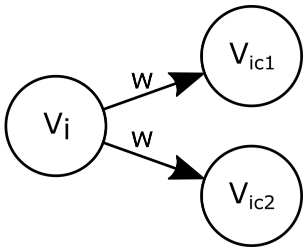


Figure 6.1: An example of the form of the input.

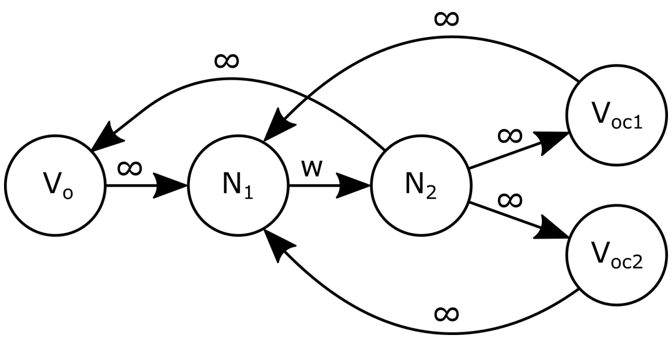


Figure 6.2: Given the input example, the output is of this form.

Pseudo-code

```
1: Initialise a weighted and directed graph with three times as many nodes
   as the input graph such that each input node corresponds to three nodes in
   the output graph; $V_i$ in the input has $V_o$, $N_1$, and $N_2$ in the output graph.
2: Set all edge weights to 0 (representing no edge)
3: for each $V_i$ and corresponding $V_o$, $N_1$ and $N_2$ do
4:     Set the edge weight from $V_o$ to $N_1$ to INF
5:     Set the edge weight from $N_1$ to $N_2$ to the output data width in bits of
       $V_i$
6:     for each child of $V_i$ called $V_{ic}$ do
7:         Get the corresponding node of $V_{ic}$ called $V_{oc}$
8:         Set the edge weight from $N_2$ to $V_{oc}$ to INF
9:         Set the edge weight from $V_{oc}$ to $N_1$ to INF
10:    end for
11:    Set the edge weight from $N_2$ to $V_o$ to INF
12: end for
```

Validation This is the first algorithm encountered in *TLegUp* that is not in the standard *LegUp* flow, and therefore all its input is sourced from *LegUp* algorithms. As *LegUp* algorithms are not in the scope of this validation, it is assumed that the input requirements are met.

Output requirement 1 is a requirement on the data type of the graph. As the graph data structure is explicitly weighted and directed, this requirement is satisfied.

Output requirements 2 to 8 regard the structure of the graph, specifically where the edges are and what their weights are. Each of these requirements is matched by a statement in the pseudo-code that creates the edge according to the requirements. However, it must be shown that once a requirement is satisfied by a statement it is not unsatisfied by subsequent statements before the algorithm terminates. Since each statement creates a unique edge that is not created before or after, and since no edges are ever removed, this requirement is met.

Output requirement 9 is met because there is no statement that creates an edge not mentioned by a requirement.

DFG Partitioning

This algorithm partitions the graph into n balanced partitions, and in doing so attempts to minimise the width of the data connections between partitions. The method behind it is explained by Yang and Wong [YW96]. They describe an algorithm which takes an input graph, a ratio, and an error value, and produces two partitions of the input graph, with the weight of one to the other being approximately equal to the ratio. The error value specifies a tolerance by which the weight of each partition can differ from the desired weight. We can apply their algorithm repeatedly to create n partitions. In our case, edge weight is proportional to the data width of the connection while node weight is proportional to the amount of logic used by the instruction at a node. The weight of a cut is the sum of the weights of all edges cut.

We strictly require that the algorithm produces the desired number of partitions. However, we place no strict requirements on balancing or minimising the width of the data connections because these results are dependent on the properties of the input graph. Some input graphs are naturally impossible to balance. Consequently, we only require that the algorithm aims to balance partitions and minimise the width of data connections. Although not strict, these requirements are important for producing good results with input graphs that can be well balanced with minimal data connections between partitions.

Requirements

Input

This algorithm only requires the output generated in the previous step with one additional requirement:

1. There are enough nodes in order to create the desired number of partitions.

Output

1. Graph must be partitioned into n partitions that balance the amount of logic allocated to each partition as well as the total bit-width of wires connecting the partitions.

Pseudo-code

```
1: Choose any two nodes, $s$ and $t$, in the flow graph
2: Find the minimum cut between $s$ and $t$
3: Mark all nodes that can be reached from $s$ without crossing the cut as
   belonging to $p_1$ (partition one), and all other nodes as $p_2$ (partition two)
4: if $p_1$ is larger than the total graph weight divided by $n$ plus some margin
    then
5:     Move one node with an edge on the cut from $p_1$ to $p_2$ such that the
       weight of the cut is increased by as little as possible
6:     Collapse all nodes in $p_2$ down to $t$
7:     go to line 2
8: else if $p_1$ is smaller than the total graph weight divided by $n$ minus some
   margin then
9:     Move one node with an edge on the cut from $p_2$ to $p_1$ such that the
       weight of the cut is increased by as little as possible
10:    Collapse all nodes in $p_1$ to $s$
11:    go to line 2
12: else                                ▷ $p_1$ is a partition with the target weight
13:    Push $p_1$ to a list of output partitions
14:    if $n > 2$ then
15:        Modify the flow graph to exclude $p_1$
16:        $n \leftarrow n - 1$
17:        go to line 1
18:    else
19:        Push $p_2$ to the list of output partitions
20:        return
21:    end if
22: end if
```

Validation We make the assumption that the margin used in line 4 is sufficiently large such that the operation at line 5 never makes p_1 smaller than the target weight minus the margin, and that the operation at line 9 never makes p_1 larger than the target weight plus the margin. This is always possible when the margin is larger than the weight of the largest node moved by those operations. In the case where the margin is too small, the algorithm would not be able to complete as the subsequent merge operation would result in a partition that is either too large or too small but cannot be reduced or increased in size respectively. We must assume that the margin is appropriate because it is a user-defined value that our algorithm has no control over. Larger margins have implications for how well balanced the resulting partition is, but do not affect the validity. On the contrary, values that are too small invalidate the algorithm.

Similarly, we can only assume that input requirement 1 is met because it is a function of n, the number of nodes in the graph (a function of the circuit design), and the margin. Again, these are user-defined parameters that our algorithm has no control over.

With the aforementioned assumptions, the algorithm will satisfy output requirement 1 because each iteration will successfully create a partition of the target weight until n partitions have been created.

The algorithm balances the weight of partitions because the algorithm only outputs partitions that satisfy the weight requirement, and all partitions are subject to the same weight requirement.

The algorithm minimises the data bits connecting partitions because the partition boundaries are defined by a minimum cut algorithm. The algorithm first starts with the minimum possible cut, and if a satisfactory partition isn't found, it merges a node and finds the next minimum cut. By merging in such a way as to increase the minimum cut the least, the algorithm sorts through possible partitions in order of increasing cut size thus finding partitions with smaller cut weights first. Since cut weights are proportional to data bits, the algorithm aims to minimise the amount of interconnected data bits between partitions.

Cycle Finding

LegUp provides us with information on the location of cycles in the program flow using a concept known as 'basic blocks' [BAS]. It creates two lists, a list of cycles of basic blocks and a list of every location where a basic block jumps back to itself or a previous basic block, known as the back edge list. This information alone is insufficient for us as we are interested in cycles in the data flow graph, not the program flow. The distinction is important as it is possible for a basic block in a cycle to be comprised of instructions with no data dependencies, and therefore be part of a program flow cycle whilst not being part of a data flow cycle. To find data flow cycles we use a depth first search [Tar72] to find a path from an instruction to itself in the data flow graph. Depth first is chosen over breadth first because we are interested in all paths and must search all instructions in the design anyway, and in this capacity, depth first is far more memory efficient.

The DFS algorithm has a run time of $M \times \mathbf{O}(N + E)$ where M is the number of instructions we search from, N is the number of instructions in the entire circuit design, and E is the number of edges between instructions in the

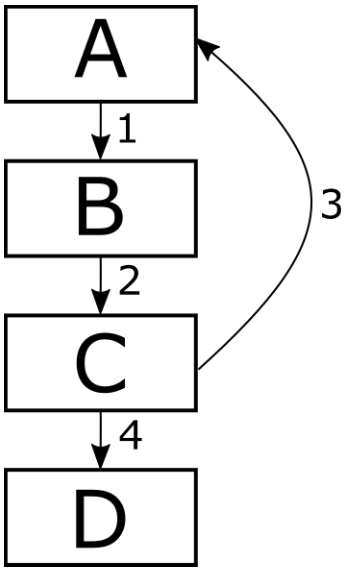


Figure 6.3: This diagram represents a program with 4 basic blocks (A, B, C, and D) and the edges between them (numbered 1, 2, 3, and 4). The list of cycles of basic blocks would contain {A, B, C} and {D} as two distinct elements. The back edge list would contain edge 3.

entire circuit design. Simply running the DFS from all instructions will generate many duplicate cycles and take a length of time on the order of hours on typical hardware. Therefore we choose to reduce the number of instructions we start the search from, thus reducing the value of M. We do this through two techniques.

The first technique is to use the program cycle information to restrict the basic blocks that we start searches from. Instructions in basic blocks that are not in cycles will never contain data flow cycles as they are only traversed once. Therefore we can restrict starting points for the DFS to instructions in basic blocks in the cycle list that aren't singletons and all blocks that have a back edge pointing to themselves. It is important to exclude singletons as they are present in the cycle list even if they don't form a cycle (see figure 6.3). We must also include blocks that have a back edge pointing to themselves in order to find the singletons which form cycles with themselves.

The second technique is only initiating searches from phi instructions [PHI]. The reasoning is that phi instructions are closely related to the concept of branching, so we assume that we find a phi instruction somewhere in every cycle.

Requirements

Input

1. Input is a list of all cycles of basic blocks

Output

1. The output is a list of lists of instructions that form cycles.

Pseudo-code

```
 1: for each cycle of basic blocks do
 2:     if the current cycle has more than one basic block then
 3:         for each basic block in the current cycle do
 4:             run FindSCC with the current block as the argument
 5:         end for
 6:     end if
 7: end for
 8: for each back edge do
 9:     if the edge is a loop then
10:         run FindSCC with the node as the argument
11:     end if
12: end for
13: procedure FindSCC(basic block)
14:     for each phi instruction in the block do
15:         DFS for a path to the current phi instruction
16:         if DFS has found one or more cycles then
17:             Add the found cycles to the list of cycles if they are not already
                 present
18:         end if
19:     end for
20: end procedure
```

Validation For this step, the input data is constructed entirely by the *LLVM* library, and therefore is not subject to our validation. However, we must still show that the input data is congruent with a list of all basic blocks that contain instructions that belong to a cycle, as per the input requirement. We can show that filtering the basic block cycle list and the back edge list satisfies this. We filter the basic block cycle list by excluding all singleton cycles on line 2. This is because such elements do not represent true cycles; *LLVM* includes single basic blocks which don't contain any cycles in the list in this manner. So far, this covers all blocks which contain instructions that form a cycle with other blocks, but we're still missing blocks that contain an internal cycle. To find those blocks, we filter the back edge list on line 9 to only inspect blocks which have an internal back edge. With these two filters, the two data sets satisfy our input requirement.

To find all cycles and satisfy the output requirement, the algorithm must run *FindSCC* on at least one instruction in each cycle in each basic block it considers. We find that the assumption about phi instructions is problematic. The *LLVM* compiler need not emit a phi instruction for every looping structure. In fact, a simple for loop doesn't contain a phi instruction by default when compiled by the *clang* compiler [ben15]. We have verified this to be the case with *LLVM* version 6.0.0; the latest version at the time of writing. This means that if we were to have a similar for loop alone in a basic block, *TLegUp* would never find the cycle. Even though the back edge list would indicate that there is a loop in the block, the block wouldn't be searched if it doesn't contain any phi instructions. Ultimately, this algorithm does not satisfy the output requirement.

There are better algorithms for cycle finding, such as that described by Johnson [BJ75].

It is recommended to switch to Johnson's algorithm for a number of reasons. Its primary purpose is cycle finding in directed graphs, so it is designed for this application and has been tested and validated before. It is also far more efficient, and is able to find every cycle in a graph with one traversal of the graph without generating any duplicates. For this reason there is no need for dangerous optimisations like restricting the amount of instructions searched from.

Synchronisation Voter Insertion

This algorithm decides where to insert voters in each cycle found. The choice is non-trivial, as it has implications on circuit area and performance.

In our chosen implementation, we use a simple heuristic that finds the combinatorial path with the lowest latency and sets a flag to insert a voter after the final instruction of the path. A combinatorial path is a sequence of instructions without any registers in between them. This allows the circuit to keep clock periods as low as possible.

Requirements

Input
This algorithm relies only on the cycle list produced by the previous algorithm.

Output

1. Each cycle specified in the input contains a synchronisation voter.

Pseudo-code

```
1: for each cycle do
2:     Mark the instruction at the end of the path with the lowest latency
3: end for
```

Validation The validation of the output is trivial as the algorithm always inserts a voter when given a cycle.

Triplication

The purpose of this algorithm is to create a *Verilog* RTL file that properly represents the triplicated circuit. We use the usual HLS methods, but whenever we encounter an instruction that is flagged for voter insertion, we insert a predefined voter circuit. The main circuit design is defined to reside within a module we instantiate three copies of. In order to connect each voter to its equivalents in the neighbouring two modules (its sibling voters), we add input and output ports to each module; two inputs and one output for each voter. Connecting the voters is a matter of iterating through each voter in the circuit and assigning wires from its input and output ports within its module to the input and output ports of its corresponding sibling voters. This is handled at the global 'top module' in the *Verilog* code.

Requirements

Input
The input to this algorithm is the usual list of LLVM instructions in the circuit that is usually handled by *LegUp*, but with one additional requirement for *TLegUp*.

1. Each instruction has a flag denoting whether or not a voter should be inserted on its output.

Output

1. There are three identical modules of the main circuit created, each consisting of n partitions.
2. Every instruction with a voter flag is followed by a voter circuit.
3. Every voter is connected to the corresponding voters in its sibling modules.

Pseudo-code

```
 1: procedure WriteMainModule
 2:     for each instruction do
 3:         Normal LegUp Code
 4:         if current instruction is flagged for voter insertion then
 5:             Write a voter circuit after the current instruction
 6:             Write unique input and output ports for the voter
 7:         end if
 8:     end for
 9: end procedure
10: procedure WriteTopModule
11:     Normal LegUp Code
12:     Instantiate 3 main modules
13:     for each voter in all three modules do
14:         Connect the output of the current voter to its two sibling voters
15:         Connect the inputs of the current voter to its two sibling voters
16:     end for
17: end procedure
```

Validation The input requirement is met by the partitioning and synchronisation voter insertion algorithms, which set flags on instructions that should be followed by a voter.

Output requirement 1 is met by line 12 of the pseudo-code, which creates exactly three instances of the main module.

Output requirement 2 is met by line 6 of the pseudo-code, which creates a voter circuit on the outputs of each flagged instruction.

Output requirement 3 is met by lines 14 and 15, which handle each and every voter in all three modules.

6.4 Conclusion

In conclusion, this tier of the verification has investigated *TLegUp* at the algorithm level. While not using a formal method, we have synthesised an appropriate informal method which provides a useful level of insight with an acceptable level of labour. By this method we have verified that many of the algorithms correctly implement the design steps, although we have uncovered an issue in the cycle finding algorithm.

7 Bottom Tier - Implementation Verification

This level of verification has not been completed, and so this section describes considerations for this level rather than presenting a method and implementation of it.

7.1 Considerations

This level deals with many implementation-specific issues regarding the use of the *clang* compiler and *LegUp* software. For example, changes must be made to *LegUp* data types in order to support voter flags. Retrieving the required information for the aforementioned algorithms may require calling *LLVM* functions that require specially constructed arguments. These issues necessitate a further set of requirements which apply only to this level. The relevant software components should be given a closer look to determine what the requirements are, and whether they conflict with the previously verified algorithms. Care must be taken when modifying existing *LegUp* code such that verification of those modifications can be constrained. Otherwise there is the issue of the task of verifying the modifications becoming the task of verifying the original *LegUp* code at large.

7.2 Possible Method

The first step should involve identifying all the implementation-specific details that the code must handle and verifying that they are congruent with the previous levels. Then the modifications to the *LegUp* code that handle such issues should be inspected to make sure they meet the requirements.

With all the implementation-specific parts of the code verified, we can focus on verifying code that implements the algorithms from the middle tier. The written code should be plainly analogous to the pseudo-code. In places where it is not, a short justification can be written, or the code could be found to be unverified. We can also employ the use of specific dynamic tests to once again verify that our implementation meets the requirements specified in the previous stage.

8 Conclusion

This section documents the future work still required on this project, as well as final comments on this report.

8.1 Future Work

Verification

At this stage, only the upper two tiers of verification have been completed. To completely verify *TLegUp*, the verification of the bottom tier needs to be completed along with a verification of *LegUp* itself. Such a task would be considerably labour intensive. In addition, since beginning this report a new version of *TLegUp* has been developed. This requires new verification for all parts where the verification in this report no longer applies.

Documentation

Currently, documentation for *TLegUp* is quite sparse. This presents a difficulty for new developers learning the architecture of the software. The work in this report has documented many concepts of the code that were previously undocumented. However, further work is required to produce comprehensive and clearly written documentation in a more suitable format.

Optimization and Improvements

There are a number of areas of improvement for this *TLegUp* version:

- The cycle finding method has a large potential for optimisation via use of Johnson's algorithm.
- Partitioning could be improved to produce a more balanced result. Currently partitioning is balanced by number of operations. Operations can have different area, so this results in unbalanced partitioning.
- Currently, all functions are inlined, which uses a lot of circuit area. It should be possible for multiple calls of the same function to share circuit hardware.
- Circuit area estimation could be improved by taking into account routing area.
- The algorithm that places voters inside feedback cycles could be improved to take into account other feedback cycles to make a better placement. For example it could decide to place one voter to serve two cycles if those cycles share a data path.

Many of these issues are dealt with in the subsequent FLP (function-level partitioning) version of the software which is able to partition at the function level rather than at the instruction level. This allows more balanced partitioning and addresses the circuit area concerns. Johnson's algorithm has also been implemented in this version as well. The issue regarding voter placement in feedback cycles remains outstanding.

8.2 Final Comments

The verification of *TLegUp* has been quite useful in documenting and verifying the assumptions in the design, as well as proving that some parts of the code do not meet our requirements. It has also highlighted numerous areas for improvement. It is anticipated that continuing to verify this project will be greatly valuable in identifying further areas for improvement.

Bibliography

[BAS] LLVM: Basic block. `http://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html`. Accessed: 2018-05-06.

[ben15] benstopics. How to create loop object on the llvm? `https://stackoverflow.com/a/34517113`, 2015.

[BJ75] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4:77–84, March 1975.

[Car00] Carl Carmichael. *Correcting Single-Event Upsets Through Virtex Partial Configuration.* Xilinx Application Note, June 2000.

[CCA$^+$11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[CGMT09] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[Fag76] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 15(3):182–211, September 1976.

[LAW$^+$17] Ganghee Lee, Dimitris Agiakatsikas, Tong Wu, Ediz Cetin, and Oliver Diessel. TLegUp: A TMR code generation tool for SRAM-based FPGA applications using HLS. In *Proc. of IEEE Symposium of Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[Lee17] Ganghee Lee. TLegUp design flow. `http://lp14partition.unsw.wikispaces.net/file/view/TLegUp_ILP_designflow.pdf/620788125/TLegUp_ILP_designflow.pdf`, 2017.

[LLV] The LLVM compiler infrastructure project. `https://llvm.org`. Accessed: 2017-10-17.

[LW98] Huiqun Liu and DF Wong. Network-flow-based multiway partitioning with area and pin constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):50–59, 1998.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[OCG$^+$09] Patrick S Ostler, Michael P Caffrey, Derrick S Gibelyou, Paul S Graham, Keith S Morgan, Brian H Pratt, Heather M Quinn, and Michael J Wirthlin. SRAM FPGA reliability analysis for harsh radiation environments. *IEEE Transactions on Nuclear Science*, 56(6):3519–3526, 2009.

[PHI] LLVM language reference manual LLVM 7 documentation. `https://llvm.org/docs/LangRef.html#phi-instruction`. Accessed: 2018-05-09.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[THNCD17] Nguyen T. H. Nguyen, Ediz Cetin, and Oliver Diessel. Improving reliability of FPGA-based systems by scheduling checks for configuration memory errors. *Submitted to IEEE Transactions on Aerospace and Electronic Systems*, 2017.

[YW96] Hannah Honghua Yang and D. F. Wong. Balanced partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1533–1540, Dec 1996.