2010-03-10

Synchronization Voter Insertion Algorithms for FPGA Designs Using Triple Modular Redundancy

Jonathan Mark Johnson
Brigham Young University - Provo

Synchronization Voter Insertion Algorithms for FPGA Designs Using Triple Modular Redundancy

Jonathan M. Johnson

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Michael J. Wirthlin, Chair
Brad L. Hutchings
Brent E. Nelson

Department of Electrical and Computer Engineering

Brigham Young University

April 2010

ABSTRACT

Synchronization Voter Insertion Algorithms for FPGA

Designs Using Triple Modular Redundancy

Jonathan M. Johnson

Department of Electrical and Computer Engineering

Master of Science

Triple Modular Redundancy (TMR) is a common reliability technique for mitigating single event upsets (SEUs) in FPGA designs operating in radiation environments. For FPGA systems that employ configuration scrubbing, majority voters are needed in all feedback paths to ensure proper synchronization between the TMR replicates. Synchronization voters, however, consume additional resources and impact system timing. This work introduces and contrasts seven algorithms for inserting synchronization voters while automatically performing TMR. The area cost and timing impact of each algorithm on a number of circuit benchmarks is reported. The work demonstrates that one of the algorithms provides the best overall timing performance results with an average 8.5% increase in critical path length over a triplicated design without voters and a 29.6% area increase. Another algorithm provides far better area results (an average 3.4% area increase over a triplicated design without voters) at a slightly higher timing cost (an average 14.9% increase in critical path length over a triplicated design without voters). In addition, this work demonstrates that restricting synchronization voter locations to flip-flop output nets is an effective heuristic for minimizing the timing performance impact of synchronization voter insertion.

Keywords: triple modular redundancy, FPGA, voters, reliability, synchronization, feedback edge set

ACKNOWLEDGMENTS

It is my pleasure to thank those who have helped me and made this thesis possible. I owe many thanks to my advisor, Dr. Michael Wirthlin, for the hours he has spent guiding my research and writing, and for his encouragement to finish this work. I would also like to thank the other members of my graduate committee, Dr. Brent Nelson and Dr. Brad Huthings, for their support and for the knowledge and experience they have helped me obtain.

My family also deserves thanks for supporting me and encouraging me along the way. My parents have been a constant source of strength to help me finish this work. I also owe thanks to my brother Eric for lighting the way before me, and to my other siblings Michelle and Jeffrey for being supportive of my education and research.

Michael Caffrey, Paul Graham, Heather Quinn, and Keith Morgan at Los Alamos National Laboratory also deserve many thanks for their guidance, advice, and the experience they helped me gain. I am very grateful for their role in helping my development as an engineer.

Finally, I would like to thank all of my fellow students who contributed in the form of advice, suggestions, and observations, including Brian Pratt, Nathan Rollins, Chris Lavin, Derrick Gibelyou, Will Howes, and Yubo Li. Their support has made this work possible.

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876 and by the Rocky Mountain NASA Space Grant Consortium.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

SRAM-based FPGAs are an attractive alternative to ASICs for space-based computing missions because their in-orbit reconfigurability enables them to perform various tasks at different stages of a mission. They are often used to implement custom designs that attain application specific performance that would not be possible with software reconfigurable only alternatives. In addition, the use of FPGAs can reduce the overall non-recurring engineering costs involved in devloping a space-based application [1–4].

FPGAs are, however, susceptible to radiation effects in space environments [5]. Radiation induced single event upsets (SEUs) are the major concern for SRAM-based FPGAs used in high radiation environments. An SEU occurs when one of the internal memory cells in an FPGA is upset by a high energy particle. An SRAM-based FPGA contains a large number of internal memory cells, including its configuration memory. This memory controls the FPGA's routing, logic, user flip-flops, internal block memory, and other aspects of the device. The functionality of an FPGA is dependent on the integrity of its configuration memory, and FPGA configuration memories are large targets for single event upsets (SEUs). Mitigation strategies must be employed in order to use SRAM-based FPGAs reliably in environments where SEUs can be encountered.

Several mitigation strategies have been developed for systems that use FPGAs in high radiation environments. The most common strategy is a combination of Triple Modular Redundancy (TMR) [5] and configuration memory scrubbing [6]. The basic concept of TMR is to triplicate a circuit design so that the resulting design consists of three redundant copies of the original, with majority voters inserted at strategic locations to mask errors in any single copy of the circuit. TMR masks failures as they occur. Configuration memory scrubbing continuously configures an FPGA with a golden bitstream stored in a protected memory in order to prevent the buildup of multiple coincident SEUs that could overcome the redundancy of TMR. The effectiveness of this strategy has been demonstrated using both fault-injection and radiation experiments [7–9].

Inserting majority voters is an important step in automated TMR. Majority voters are used to mask errors in any one of the three TMR replicates, and to enable resynchronization of the circuit state after configuration scrubbing [10]. Voters are inserted within *all* feedback paths to ensure that state within logic feedback is updated when the bitstream scrubbing process repairs circuit resources. Identifying good locations for these voters, however, is a difficult problem. Poor synchronization voter locations lead to large area overhead and a significant increase in critical path timing.

Previous work in the area of TMR voter insertion has focused primarily on the reliability impact of voters and has been conducted predominantly with theoretical or manual TMR implementations. For example, Gurzi [11] investigated theoretically optimal partition sizes and voter placement in triplicated logic networks for maximizing overall reliability given a particular voter reliability. He showed that equal partition sizes provide the highest reliability, and that the best number of partitions to use depends upon voter reliability, but his work is not specific to FPGA implementations of TMR. In [12], Kastensmidt et al. showed that partitioning a triplicated FPGA circuit with extra voters can reduce the sensitivity of the circuit to domain crossing events (DCEs), which occur when a single event upset affects the FPGA routing network in such a way that more than one of the TMR replicates is compromised. Manually applied TMR was used to demonstrate this result. In [13], Pratt used an automated algorithm to show that using extra voters to partition a triplicated circuit can be effective at reducing the sensitivity of the circuit to multiple independent upsets (MIUs) within a single scrubbing cycle.

The focus of this work is on the creation and analysis of seven algorithms that automatically select appropriate locations for synchronization voters in triplicated FPGA circuits. In addition to addressing the reliability impact of voter insertion, this work investigates the timing performance and area impact of the voter insertion algorithms on FPGA implementations of automatically applied TMR. It is shown that certain algorithms are better for minimizing the impact of TMR on timing performance while others are more useful for minimizing the area of the resulting circuit. It will also be shown that restricting voters to locations directly after flip-flops is a good heuristic for preserving the timing performance of triplicated FPGA designs.

Although commercial tools for performing automated TMR and inserting synchronization voters exist [14], this is the first published work that demonstrates how to perform automated

synchronization voter insertion. Correctly inserting synchronization voters is one of the most difficult parts of implementing TMR because all feedback paths must be intersected with at least one voter, and performing manual voter insertion can be impractical for complex designs. All of the algorithms presented in this work are available as part of an open source tool created at Brigham Young University that is capable of performing automated TMR on FPGA designs. Information on obtaining and using this tool is available in Appendix A.

This work will first present a background of reliability issues for space-based systems that incorpate SRAM-based FPGAs in Chapter 2. Mitigation techniques will be discussed, including the commonly used technique of combining TMR and configuration memory scrubbing. Next, typical voter insertion issues will be discussed in Chapter 3. Then in Chapter 4, seven algorithms for determining synchronization voter locations in triplicated FPGA designs will be presented. In Chapter 5, experiments that were conducted to compare and evaluate these algorithms will be outlined and the results will be analyzed. The work concludes in Chapter 6.

CHAPTER 2. BACKGROUND

This chapter will summarize radiation effects issues in FPGAs that make techniques such as TMR necessary in FPGA designs for space-based missions. The primary effect that is considered by this work is the single event upset (SEU). This chapter will also discuss common reliability techniques used in systems that use FPGAs in radiation environments. The most commonly used and well understood method is to use TMR in conjunction with a technique called configuration memory scrubbing.

2.1 Radiation Effects in FPGAs

There are several radiation effects that can occur in FPGAs and other CMOS devices. The main effects include total ionizing dose (TID) effects, single event latchups (SELs), and single event upsets (SEUs). TID effects are the changes in electrical parameters of a device due to radiation-induced charge [15]. These effects occur due to exposure to ionizing radiation over time. A single event latchup (SEL) occurs when a charged particle induces a high current state that causes transistor latchup. Such an event can cause permanent device damage. At the very least, it requires power cycling the device to return it to a normal operating state. A single event upset (SEU) occurs when a charged particle strikes an SRAM cell, causing the state of the memory cell to change.

Some FPGA manufacturers guarantee the TID life of their devices as well as SEL immunity. For example, the QPro Virtex-II family of radiation-hardened FPGAs is guaranteed by Xilinx to have a TID life of 200K Rad(Si) and is guaranteed to be latchup immune up to a linear energy transfer (LET) of at least 160 MeV-cm^2/mg [16]. All SRAM-based FPGAs, however, are susceptible to single event upsets (SEUs). SEUs are problematic for FPGAs because their configuration memories contain millions of memory cells, which makes them a large target for SEUs.

The functionality of an SRAM FPGA is dependent on the contents of its configuration memory. FPGAs are typically made up of highly configurable logic blocks containing lookup tables (LUTs) that define logic functions and registers used for sequential logic. A reconfigurable routing network connects the logic blocks in an FPGA in order to implement complex designs. The contents of LUTs, the functionality of registers, and the routing network connections are all stored in SRAM cells in an FPGA's configuration memory. The functionality of an FPGA changes when the contents of its configuration memory change.

An SEU in an FPGA's configuration memory often affects only a single memory cell, but multiple bit upsets (MBUs) can also occur when a charged particle strikes adjacent memory cells. Even a single bit flip can have significant consequences on FPGA functionality. For example, a single bit flip within the memory that controls a LUT changes the logic function implemented by the LUT (this could, for example, cause next state logic in a state machine to be corrupted, causing the state machine to transition into an invalid state; or it could cause output logic to send incorrect results to circuit outputs). A single bit flip could also change a subset of the connections in the FPGA's routing network.

FPGA Block RAMs (BRAMS) and user flip-flops are also sensitive to SEUs. BRAMs are often used as memories or FIFOs in FPGA designs. User flip-flops are the registers in the FPGA that are instantiated in a design for use in state machines, counters, and other sequential logic structures. BRAM and user flip-flop upsets can cause a design to enter invalid states. Although these kinds of upsets are important, the configuration memory (including routing configuration) has a much larger cross section and is more likely to receive SEUs (see Table 2.1).

Table 2.1: Latch types in the Virtex XQVR300FPGA. Repeated from [17].

Latch Type	**Function**	**No. Bits**
CLB	Configuration Logic Blocks	6,144
IOB	Programmable IO Blocks	948
LUT	Look Up Tables	98,304
BRAM	Block RAM	65,536
	Routing & Other Bits	1,579,860

2.2 Mitigation Techniques

Many mitigation techniques have been considered for use in FPGAs (i.e. quadded logic, temporal redundancy, error correcting state machine encodings, etc.), but none have been shown to provide greater reliability than triple modular redundancy (TMR) [18]. TMR is the most commonly used and well understood mitigation technique for space-based missions that incorporate FPGAs, and it is most effective when used in conjunction with a technique called configuration memory scrubbing.

2.2.1 Triple Modular Redundancy

TMR is a well known fault mitigation technique originally proposed by Von Neumann in 1956 [19]. TMR uses redundant hardware to mask circuit faults. A circuit protected by TMR in its most basic form has three redundant copies of the original circuit and a majority voter. A fault in any one of the three replicates of the original circuit does not produce an error at the output because the majority voter selects the correct output from the other two replicates. Triplicated voters are often used to avoid a single point of failure (see Figure 2.1).

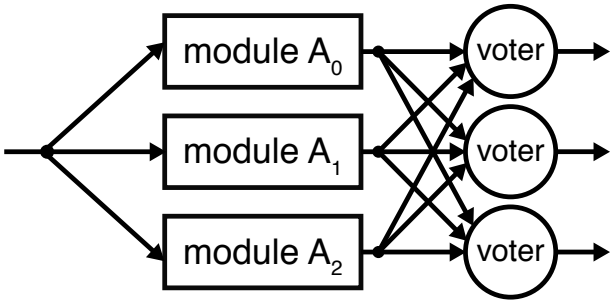


Figure 2.1: A Basic TMR Implementation.

TMR is used extensively to mitigate against radiation induced SEUs in SRAM-based FPGA systems. It has been shown through fault-injection and radiation experiments [7–9] to provide significant improvements in reliability. However, TMR also has significant area and timing performance costs. In an FPGA, TMR increases the size of a circuit by at least 3X and by as much as 6X [20]. FPGA circuits can also suffer a significant decrease in timing performance when TMR is

applied, as will be seen in Chapter 5. Sometimes, a variant of TMR called partial TMR is used to partially triplicate an FPGA circuit when there are insufficient resources on the FPGA to triplicate the whole circuit. When partial TMR is used, triplication can be applied selectively by priority so that the most important parts of the circuit are mitigated [21].

2.2.2 Configuration Memory Scrubbing

Although TMR protects a circuit from single SEUs, it can fail when multiple independent SEUs occur in such a way that two or more of the TMR replicates are affected. When the outputs of two or more of the TMR replicates are in error, majority voters select the incorrect output. Configuration memory scrubbing is a technique that is used to prevent the buildup of multiple independent SEUs.

Configuration memory scrubbing is used in conjuction with TMR to prevent the accumulation of multiple independent SEUs from overcoming the redundancy of TMR [22]. The technique works by continuously reading back the contents of the FPGA's configuration memory and repairing any errors that are found. The amount of time it takes to read and repair the entire configuration memory is called a scrub cycle and is dependent on the size of the FPGA and the implementation of the scrubber. In general, some external hardware is required, such as a protected memory for configuration data storage. A variety of FPGA configuration memory scrubber implementations are possible [23, 24]. When TMR and configuration scrubbing are used together, the effects of errors are masked by TMR as they occur and the errors are corrected as soon as possible by scrubbing.

2.2.3 Reliability Modelling

Reliability modelling techniques can be used to show the effectiveness of TMR and scrubbing. The reliability over time of a non-redundant system, a system using TMR without scrubbing, and a system using TMR with configuration memory scrubbing are compared in Figure 2.2. The reliability of the non-reduntant system is computed as the probability that the system will still be operational at time t given the failure rate λ, and is

$$R_1(t) = e^{-\lambda t}.$$

The reliability of the TMR system without scrubbing is modelled using a simple combinatorial modelling technique. The reliability as derived in [25] is

$$R_2(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t}.$$

The TMR system that uses scrubbing requires the more complex Markov modelling technique. Its reliability is derived in [26] as

$$R_3(t) = \frac{(\mu + 5\lambda)\sinh(\frac{1}{2}t\sqrt{\mu^2 + 10\lambda\mu + \lambda^2})e^{-\frac{1}{2}(\mu+5\lambda)t}}{\sqrt{\mu^2 + 10\lambda\mu + \lambda^2}} + \cosh(\frac{1}{2}t\sqrt{\mu^2 + 10\lambda\mu + \lambda^2})e^{-\frac{1}{2}(\mu+5\lambda)t},$$

where μ is the repair rate of the scrubbing system. The plots in Figure 2.2 are based on the failure rate $\lambda = 0.001$ and the repair rate $\mu = 0.1$, which were chosen to represent a typical space-based computing environment.

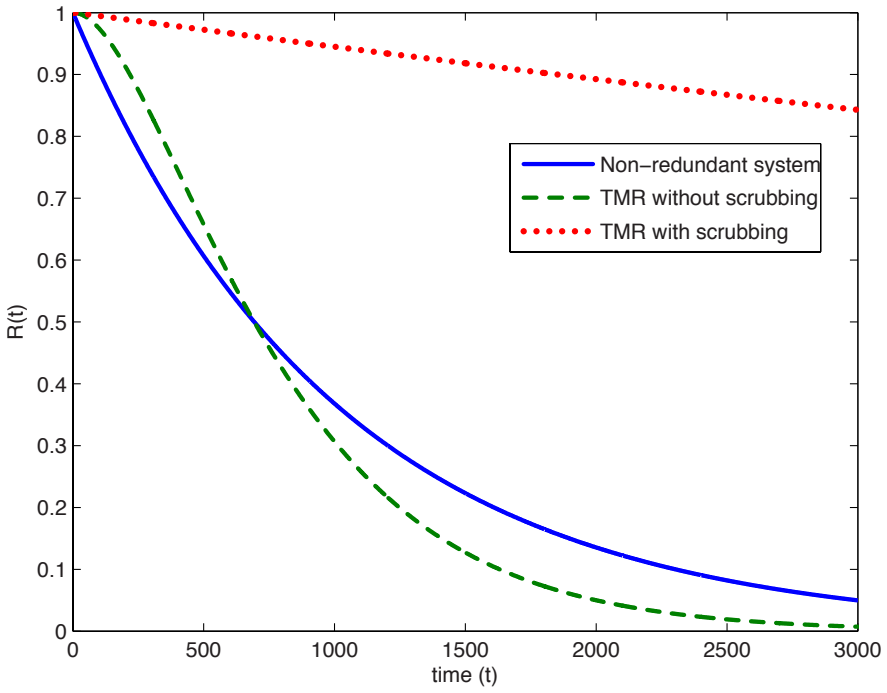


Figure 2.2: Reliability comparison of three systems using $\lambda = 0.001$ and $\mu = 0.1$.

As seen in the figure, a non-redundant system is generally less reliable than a TMR system without repair (i.e. without configuration memory scrubbing) for short mission times. A TMR system with scrubbing is much more reliable than both a non-redundant system and a TMR system without repair. It is interesting to note that for longer mission times the non-redundant system appears more reliable than the TMR system without scrubbing. This is because once the redundancy of TMR has become overcome by multiple unrepaired upsets, the added area of the redundancy and voters actually becomes a reliability weakness.

2.3 Automated TMR

Although TMR is often applied to designs manually, the process is straightforward enough to be implemented by an automated CAD tool. Existing tools for applying TMR to FPGA designs include the Xilinx XTMR tool [5, 14] and the BYU/Los Alamos National Laboratory BL-TMR tool [21]. Using an automated tool can provide several advantages over implementing TMR by hand. For example, inserting voters in the proper places manually is a tedious and error prone process. Using an automated TMR tool allows the mitigation technique to be applied much more quickly.

The process of automated TMR begins with creating three identical copies of the original circuit. First, each component instance is triplicated. Next, each net is triplicated. The nets are then connected such that the connectivity of each of the three replicates matches the connectivity of the original circuit. This is the straightforward part of TMR. Inserting majority voters to mask errors is a more complex process and is the focus of the algorithms presented in this work.

The voter insertion algorithms presented in this work operate within the context of the BL-TMR tool which is capable of applying automated TMR and several other mitigation techniques to FPGA circuits. The tool operates on circuits represented at the post-synthesis netlist level. In this representation, circuits consist of instances of FPGA primitives such as LUTs, flip-flops, and dedicated hardware, and nets that define the connectivity between the primitives. The result of applying TMR using the voter insertion algorithms presented in this work is a new netlist that contains a triplicated version of the original netlist with voters inserted at appropriate locations. After automated TMR, the triplicated netlist follows the traditional FPGA process of technology mapping, placement, and routing, as shown in Figure 2.3.

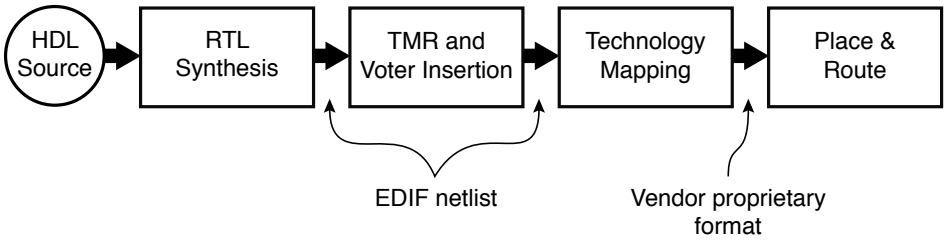


Figure 2.3: TMR toolflow for FPGAs.

2.4 Conclusion

FPGAs are susceptible to various radiation effects when used in space-based computing applications. The primary effect addressed by this work is the single event upset (SEU). A combination of TMR and configuration memory scrubbing is used to mitigate the effects of SEUs in FPGA circuits used in radiation environments. This technique has been shown both in theory and practice to provide a significant reliability improvement. TMR is a technique that is often applied manually, but becomes much easier and less time consuming to use when applied using an automated CAD tool. The voter insertion algorithms presented in this work are implemented as part of such a tool and make the automated application of TMR and insertion of voters a practical alternative to slow and tedious manual TMR implementations.

CHAPTER 3. TMR VOTER INSERTION

Voter insertion is one of the most important steps in applying TMR to a circuit design. It is also the most complex step as voters are inserted in various locations for different reasons, and because of the wiring necessary to insert the voters. This chapter introduces the four main types of voters used in FPGA implementations of TMR. It also describes in general terms how locations for voter insertion are identified. In addition, it describes the process of inserting voters once suitable locations have been identified.

Various reliability concerns motivate voter insertion at different circuit locations. We refer to voters by names that indicate their purpose in a circuit. Voter categories typically used in FPGA implementations of TMR for reliable operation in space-based missions include the following: reducing voters, partitioning voters, clock domain crossing voters, and synchronization voters. Each of these voter categories will be described in detail in the sections that follow.

3.1 Reducing Voters

A reducing voter reduces outputs from the three TMR replicates to a single output. It is a simple majority voter that is generally implemented as a LUT3 primitive. The most common use of reducing voters is at circuit outputs. For example, when reducing voters are not used at circuit outputs, the FPGA outputs are triplicated which requires the use of external voting (and 3 times as many outputs). These requirements can be eliminated by using reducing voters to obtain a single set of untriplicated circuit outputs. Reducing voters are sometimes made necessary when the target FPGA has insufficient I/O resources to allow full triplication of the outputs. In these situations, a reducing voter is used at each circuit output to reduce the outputs from the three TMR replicates to a single output, as shown in Figure 3.1.

Reducing voters are also useful in partial TMR configurations. When partial TMR is used, there are circuit locations where data must flow from a triplicated partition to a non-triplicated

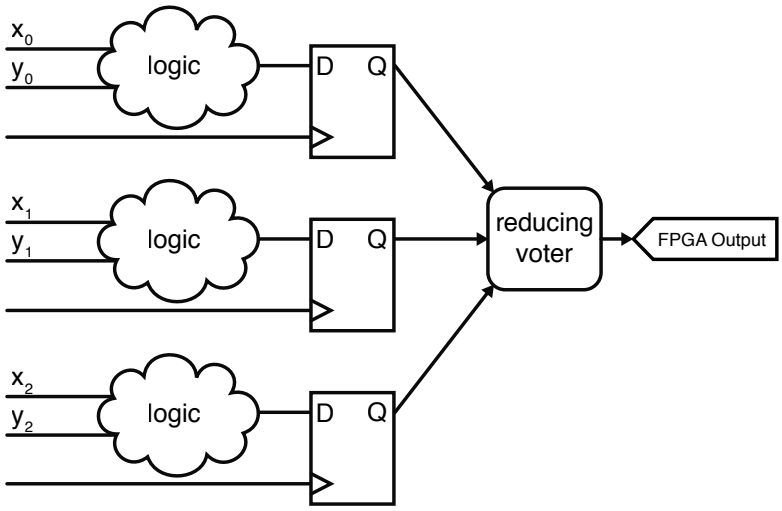


Figure 3.1: Reducing Voter

partition. Reducing voters are used at these locations to provide a single input to the non-triplicated partition.

TMR can also be mixed with duplication with compare (DWC), an error detection technique which uses duplication instead of triplication. In such a configuration, there are circuit locations where data must flow from a triplicated circuit partition to a duplicated partition. At such locations, two reducing voters are used in parallel to reduce outputs from the three TMR replicates to two inputs for the duplicated partition.

3.2 TMR Partitioning Voters

Partitioning voters are used to increase the reliability of a circuit by creating multiple TMR partitions within the design. In a typical TMR system, errors that occur in the configuration memory are discovered and corrected by scrubbing. In a circuit that has voters only at the outputs, errors are masked as long as they occur in only one of the three TMR replicates at a time. If multiple independent upsets occur fast enough such that they accumulate in more than one replicate before being corrected by scrubbing, the redundancy of TMR is overcome. In such a state, TMR voters can receive erroneous signal values on two out of three inputs, making it possible for errors to propagate through the voters to circuit outputs. This is shown in Figure 3.2. In this example, upsets occur in both $\texttt{moduleA}_0$ and $\texttt{moduleB}_0$. This causes each of the three voters to receive erro-

neous values on two out of three inputs, which means that the voters select the incorrect values to propagate.

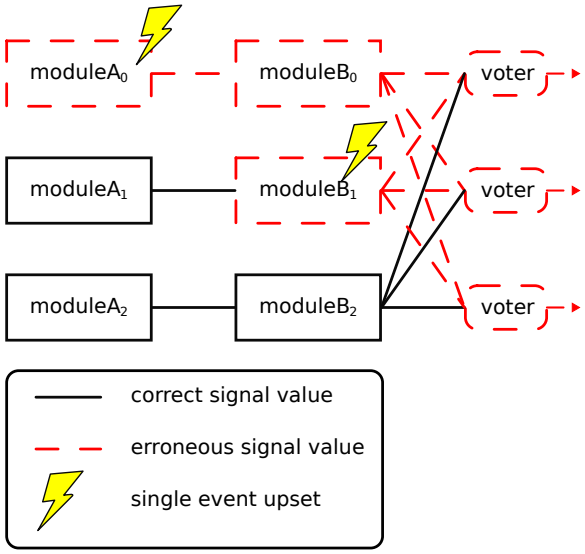


Figure 3.2: Errors in multiple replicates of a single TMR partition

The vulnerability of TMR to multiple independent upsets in separate replicates can be mitigated to a degree by subdividing a circuit into multiple partitions separated with triplicated voters [13]. The added partitions allow the circuit to tolerate more multiple independent upsets. In a TMR system with multiple partitions, each partition can tolerate errors in only one of the TMR replicates. Multiple independent upsets that occur in separate TMR replicates but are separated by partition boundaries (a set of triplicated voters) are called concurrent non-overlapping failures, and they are successfully masked by TMR voters. For example, in Figure 3.3, the same upsets occur as in the previous example, but this time they are in separate partitions because of the added partitioning voters. In the first set of voters, each voter receives only one bad input (each from $\texttt{moduleA}_0$). Each of these voters propagates the correct values received on the other two inputs. Likewise, in the second set of voters, each receives a single bad input from $\texttt{moduleB}_1$. Each propagates the correct values received on the other two inputs.

The probability of multiple independent upsets being in separate partitions increases with the number of partitions in a TMR circuit. The reliability of a circuit can be improved by subdivid-

ing it into smaller and smaller partitions up to the point where the reliability gains from partitioning are overridden by the unreliability of the voters being inserted between the partitions. The optimal placement of partitioning voters for reliability is a difficult issue that is beyond the scope of this work, but it is discussed further in [11] and [12].

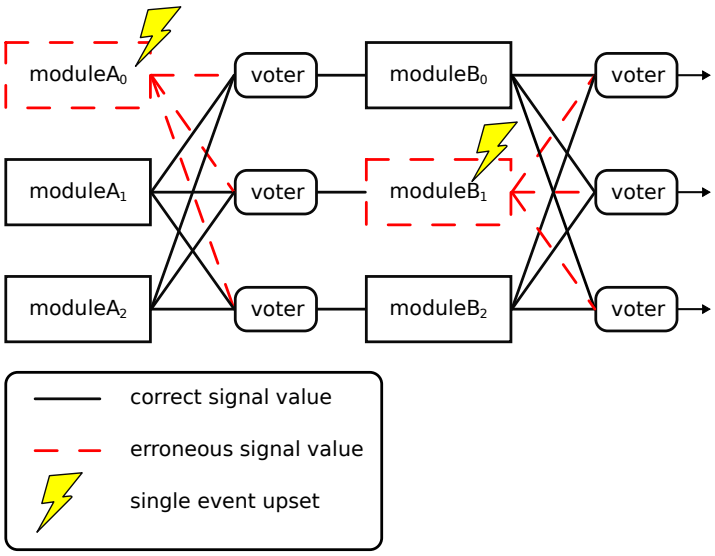


Figure 3.3: Non-overlapping failures masked when TMR partitions are used

Partitioning voters also have a secondary benefit. They decrease the amount of time potentially required to resynchronize the registers of TMR replicates after becoming unsynchronized due to an SEU. When an error affects sequential logic, erroneous values can be propagated through several registers in the affected replicate. When the configuration memory is corrected, it can take several clock cycles for the correct signal values to propagate through all of the affected registers. By using partitioning voters to break up sequential logic, the number of registers that can be affected by a single SEU, and thus the number of clock cycles required for resynchronization, is reduced. This is important because during the time the TMR replicates remain unsynchronized, any additional SEUs in the two yet unaffected replicates would overcome the redundancy of TMR, allowing functional errors to propagate through voters.

As an example, consider the shift register in Figure 3.4. When an SEU affects the indicated location, incorrect signal values propagate through the remainder of the shift register of the affected

replicate. The replicates remain unsynchronized until the configuration memory is corrected via scrubbing and correct values propagate to the end of the shift register. Until this happens, the voters at the end of the shift register mask errors so that they do not reach the rest of the circuit. However, until the registers become resynchronized, the circuit is left vulnerable; additional SEUs in the shift registers of the other two replicates could overcome the redundancy of TMR.

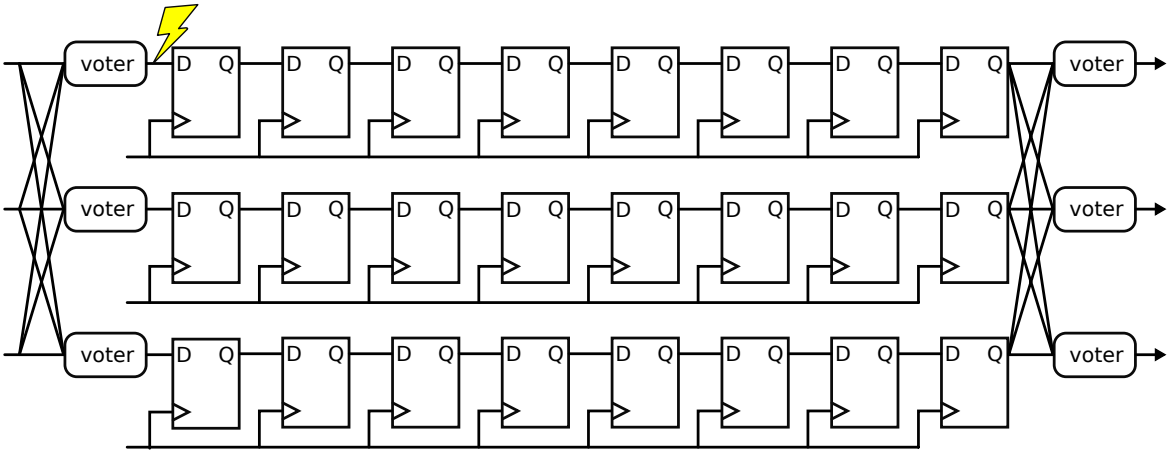


Figure 3.4: An unpartitioned shift register

The time during which the circuit is vulnerable in this manner can be reduced by partitioning the shift register as shown in Figure 3.5. With the shift register partitioned in this manner, only four flip-flops can be affected by an SEU at the indicated location instead of eight. This cuts the time it could potentially take for resynchronization in half.

3.3 Clock Domain Crossing Voters

Special consideration is required when applying TMR to circuits with multiple clock domains because the clock domain crossing synchronizers usually present in such circuits pose a TMR synchronization hazard. Extra voters are needed to mitigate this hazard. This section will describe the issues related to this hazard.

In a circuit with multiple clock domains, clock domain crossing synchronizers are used to reduce metastability effects when a signal from one clock domain enters a second clock domain.

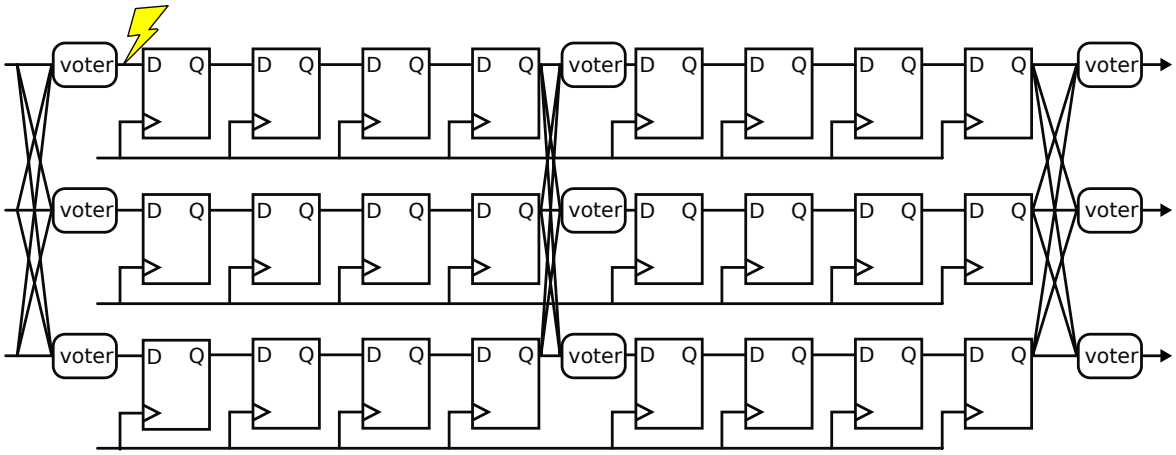


Figure 3.5: A partitioned shift register

A typical clock domain crossing synchronizer consists of a small number of consecutive flip-flops to reduce the probability of a metastable value propagating through the entire synchronizer. Although synchronizers bring the probability of metastable events to an acceptably low value, there can still be sampling uncertainty because a signal arriving from the sending clock domain appears asynchronous to the receiving domain and may cause setup or hold time violations.

The sampling uncertainty inherent in clock domain crossing synchronizers is normally not an issue, but becomes an issue when TMR is used. This is because sampling uncertainty causes the possibility that the outputs of three synchronizers in separate TMR replicates do not propagate outputs from the sending clock domain during the same cycle [27]. Three possibilities can occur:

1. The three synchronizers propagate outputs from the sending clock domain during the same clock cycle,
2. Two of the three synchronizers propagate outputs from the sending clock domain one clock cycle after the other synchronizer, and
3. One of the three synchronizers propagates its output from the sending clock domain one clock cycle after the other two synchronizers.

Figure 3.6 illustrates the third possibility. In the figure, $\texttt{sigA}_0$, $\texttt{sigA}_1$, and $\texttt{sigA}_2$ arrive at the receiving clock domain's synchronizers at a time that causes a setup time violation. This causes the uncertainty seen in the waveforms of $\texttt{sigB}_0$, $\texttt{sigB}_1$, and $\texttt{sigB}_2$, which in turn causes $\texttt{sigC}_1$ to go high one whole clock cycle after $\texttt{sigC}_0$ and $\texttt{sigC}_2$.

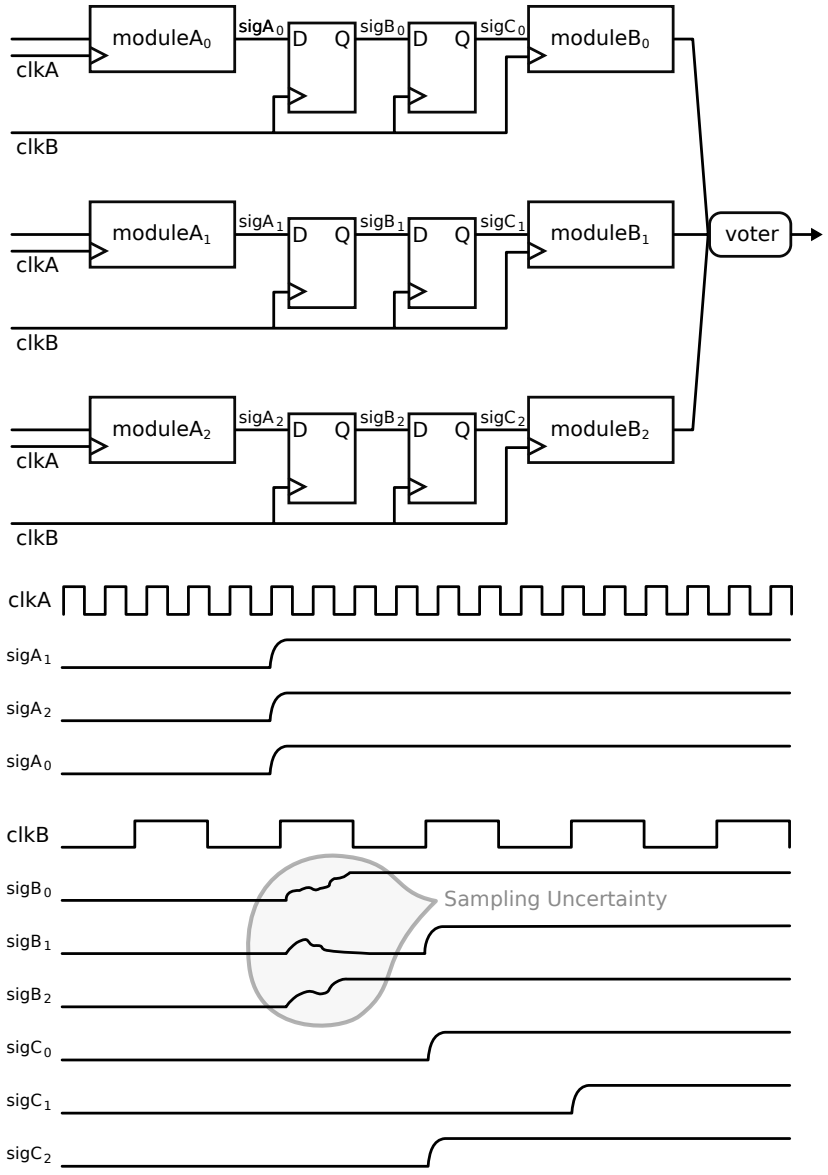


Figure 3.6: Clock domain crossing synchronizer hazard

TMR voters at circuit outputs can mask errors created by a single replicate being unsynchronized with the other two, but such a situation leaves the circuit vulnerable to further errors. With the redundancy created by TMR already being used to correct TMR synchronization errors caused by clock domain synchronizer sampling uncertainty, any SEU in one of the two synchronized replicates could completely overcome the redundancy of TMR, allowing errors to propagate through voters. In fact, this situation leaves the circuit less reliable than if TMR hadn't been used at all.

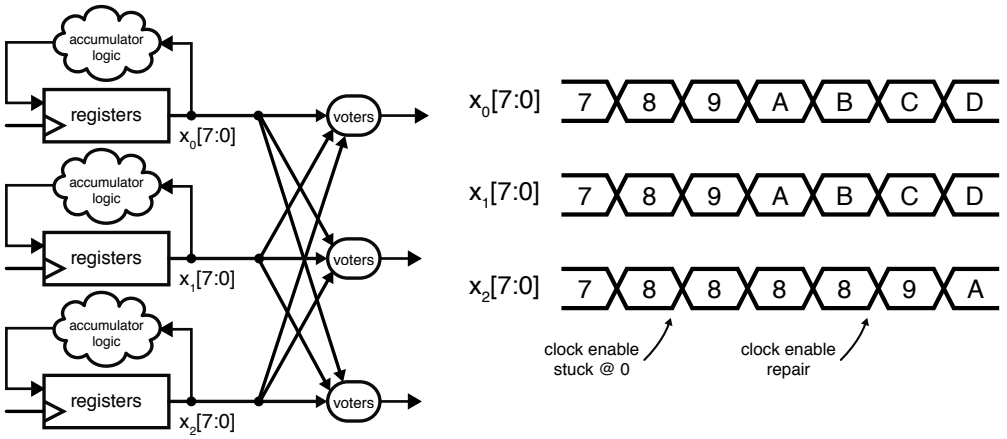
Several strategies for mitigating TMR circuits that have clock domain crossing synchronizers are being investigated. These strategies involve strategically placing additional TMR voters in order to resynchronize TMR domains after clock domain crossing synchronizer outputs. Li demostrates two such strategies in [27].

3.4 Synchronization Voters

The final type of voter is the synchronization voter. Synchronization voters are necessary when configuration memory scrubbing is used with TMR designs that include sequential logic with feedback (almost all designs). The purpose of synchronization voters is to restore correct registered state after FPGA logic problems are repaired by configuration scrubbing. For example, when an SEU affects logic, incorrect signal values may propagate to registers in one of the replicates of the circuit. If the registers are involved in a feedback loop, incorrect values may persist in the loop even after the SEU is corrected by configuration scrubbing. This motivates the use of synchronization voters placed *within* sequential logic feedback loops. Their purpose is to restore correct registered state within feedback loops of a single TMR replicate by using the values from the other two replicates.

The importance of synchronization voters is demonstrated by the example of the simple triplicated counter in Figure 3.7(a). Three copies of a register and accumulator logic are instantiated to provide fault tolerance for any single circuit failure. Voters are placed at the outputs to select the majority result should a failure occur. The synchronization problem that occurs with this circuit is demonstrated by the waveform of Figure 3.7(b).

In this example, a configuration fault forces the clock enable of the third TMR replicate into a stuck-at-0 condition. Because of this fault, the counter does not increment; it remains in the same count state until the clock enable is repaired by scrubbing. Once the counter has been repaired by a configuration scrubbing process, it continues its count sequence from the state in which it was stuck. Although repaired and operating properly, the counter is out of sequence with the other two counters. While the TMR voter circuitry properly ignores the incorrect count value, the reliability of the circuit is reduced because the counters are not synchronized. That is, any additional faults in the other TMR replicates would cause the redundancy of TMR to be overcome, allowing the error



(a) Simple counter with voters *outside* the feedback loop.

(b) A simple counter is susceptible to TMR synchronization problems when SEUs occur within the feedback loop, even after scrubbing has corrected the configuration memory.

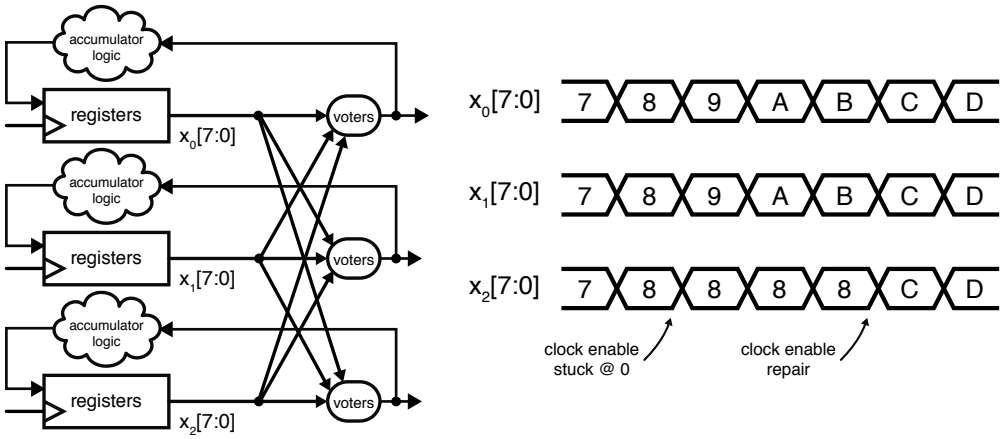
Figure 3.7: A simple triplicated counter.

to propagate to the rest of the circuit. In this state, the circuit is less reliable than if TMR had not been used at all (because of the extra area added by the TMR replicates).

Synchronization voters are voters placed within the feedback of a circuit to provide resynchronization after a fault occurs. Figure 3.8(a) demonstrates the proper use of synchronization voters by placing the voters *within* the feedback loop. Using the voters within the feedback ensures that the proper input value is provided to all of the counters no matter where the fault lies.

The benefits of this technique are illustrated in the counter failure waveform of Figure 3.8(b). As described in the prior example, the third TMR replicate experiences a stuck-at-0 fault on its clock enable input. While this fault is present, the third counter retains the same value and falls out of sequence with the other counters. The voter circuitry masks this faulty value and provides a correct value on the feedback path. Once the configuration fault is repaired by online scrubbing, the proper value is loaded into the third counter and it becomes resynchronized with the other counters. With all three counters synchronized and repaired, the circuit will reliably operate in the presence of another configuration fault.

The placement of synchronization voters is a difficult issue to resolve automatically. There are two constraints that govern the placement of synchronization voters. The first is that all design feedback must be intersected by synchronization voters. The second constraint is that there are



(a) Simple counter with voters *inside* the feedback loop.

(b) Synchronization voters protect the counter from TMR synchronization problems when scrubbing SEUs.

Figure 3.8: A triplicated counter protected by synchronization voters.

certain nets in a netlist representation of a circuit that cannot have voters placed on them because of the FPGA architecture. Within the space left by these two constraints there are many possible synchronization voter configurations. Finding a valid configuration is simple, but determining the best configuration is difficult because the locations of the synchronization voters affect the timing, area, and reliability of the resulting circuit. Heuristic algorithms that attempt to determine good synchronization voter insertion locations are discussed in the next chapter.

3.5 Illegal Voter Locations

One of the constraints that governs voter insertion is that there are certain nets in a netlist representation of a circuit that cannot be cut by voters because of the FPGA architecture. Figure 3.9 illustrates an example of this issue. The figure shows two bits of a simple ripple-carry adder implemented using the dedicated carry chain and arithmetic hardware found in the Virtex FPGA family. The adder in the figure is implemented using logic cells in two different slices. Net A in the figure cannot be cut by voters because this net is implemented by a dedicated route connection within a logic slice. Since there is no reconfigurable routing between a *MULT_AND* primitive and a *MUXCY* primitive, a *MULT_AND* cannot drive a voter and a *MUXCY* cannot receive its input directly from a voter. We refer to locations such as net A as illegal cut locations. Other

illegal cut locations include nets between *MUXCY* and *XORCY* primitives, nets between internal multiplexors that are used to create wider LUTs or multiplexors (i.e. *MUXF5*, *MUXF6*, *MUXF7*, *MUXF8*), and some nets connecting cascaded *DSP48* primitives. Voter insertion algorithms must not create netlists that have voters inserted at illegal cut locations.

In addition to illegal cut locations, there are other locations where inserting voters is legal, but results in an undesirable implementation. For example, net B in Figure 3.9 is implemented using fast dedicated carry chain routing. Adding a voter on this net is legal but will break the high-speed carry chain logic. To add a voter, the output of the *MUXCY* primitive in the lower slice must be routed to a different slice where the voting is performed. The output of this voter would then need to be routed into the CIN input of the upper *MUXCY*, breaking the high-speed carry chain. In addition to avoiding illegal cut locations, the voter insertion algorithms presented in this work avoid dedicated carry chain routing nets in order to preserve timing performance as much as possible.

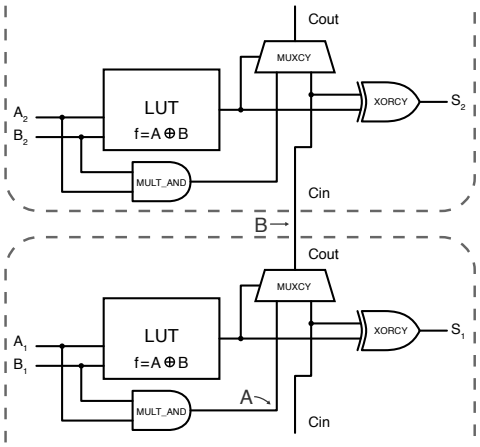


Figure 3.9: Two bits of a ripple-carry adder using FPGA primitives, carry chain, and dedicated arithmetic hardware.

3.6 Voter Insertion

Once voter insertion locations have been determined, the actual insertion of voters into the circuit is a straightforward process. The location of voters in a TMR design is specified in terms

of nets from the original, unmitigated design. When inserting a voter at a net location, the net is split into two pieces and a voter is inserted in the middle. The source of the original net becomes the source of the voter and the sinks of the original net are driven by the voter. This voter insertion occurs in the context of TMR where there are three copies of the source and three copies of each instance. Inserting a voter on a net in the original design involves replacing the three copies of the net in the TMR design with voter nets as described in the following process:

1. Instantiate three voters to perform triple voting on the given net,
2. Identify the three copies of the source of the net and connect these sources to the inputs of each of the three voters, and
3. Connect the output of each voter to the corresponding sinks of the net.

We refer to the process of inserting voters on a net as cutting a net with voters, since the original net is replaced by two sets of triplicated nets: one feeding into the voters and one exiting from them. Figure 3.10 illustrates the basic triplication and voter insertion process.

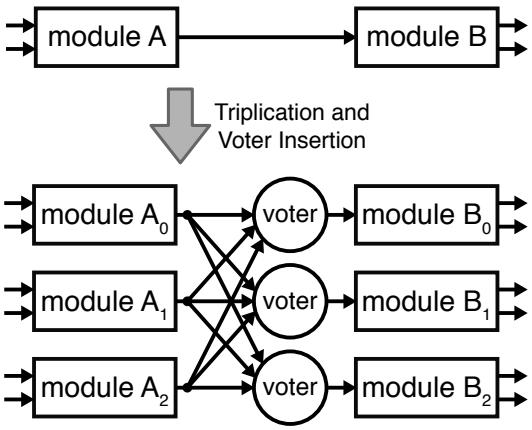


Figure 3.10: The net after Module A is cut with triplicated voters.

3.7 Conclusion

Voters are used in TMR designs for various purposes, including reducing triplicated outputs to a single output, creating multiple TMR partitions, mitigating clock domain synchronization circuitry, and protecting the synchronization of the TMR replicates within sequential logic feedback.

The synchronization voters are difficult to place optimally because there are many possible configurations, and the voter locations have a significant impact on the area and timing performance of the resulting circuit. In addition, there are certain locations in a circuit where voters cannot or should not be place due to FPGA architectural constraints. Once voter insertion locations have been determined, the process of inserting the voters is straightforward and easy to implement in an automated CAD tool.

CHAPTER 4. SYNCHRONIZATION VOTER INSERTION ALGORITHMS

Synchronization voters are essential in FPGA circuits that use TMR because they ensure that the internal state of the TMR replicates are synchronized after configuration scrubbing. Although adding synchronization voters in a design manually is a difficult and error prone process, most implementations of TMR are done by hand. The process of selecting synchronization voter locations and inserting the voters into a circuit can be automated by CAD tools. This chapter will introduce several algorithms that can enable CAD tools to automatically select locations for synchronization voters. All of the algorithms in this chapter are implemented as part of the open source BL-TMR tool. Information on obtaining this tool is available in Appendix A.

Synchronization voter insertion algorithms must determine a set of nets within a design that cuts all feedback in the design. Voters are placed on each of these nets to ensure that synchronization voting occurs within the feedback structures of a design (see Figure 3.8(a)). Determining a set of voter locations that satisfy this constraint is an instance of the feedback edge set (FES) problem. Determining a *minimum* set of voter insertion locations to satisfy the constraint is an instance of the minimum (FES) problem, which is NP-hard [28].

While polynomial time approximation algorithms exist for the minimum FES problem [29, 30], the minimum set of voter insertion locations is not necessarily the best solution for FPGA implementations of TMR. In order to preserve performance, care must be taken to avoid voter insertion locations that would negatively impact timing performance. In addition, existing FES algorithms cannot be applied directly because FPGAs have illegal cut locations. Each of the algorithms in this section solves the FES problem for voter insertion in a way that avoids illegal cut locations (see Figure 3.9 and related discussion).

The goal of the algorithms presented in this chapter is to minimize the area and timing performance impact of synchronization voter insertion by selecting good locations for the voters and using as few voters as possible. Poorly placed voters can adversely affect both the area and

timing performance of a design. For example, when multiple voters are placed within a single timing path (we consider a timing path to be any path from one flip-flop to another), the critical path of the design may be increased more than is necessary. In addition, the locations chosen to intersect the feedback loops of a design affect the total number of voters required. Many of the algorithms in this chapter employ heuristics based on FPGA architecture that attempt to minimize circuit area and timing impact.

This chapter will first present two very simple voter insertion algorithms that solve the problem in a local manner. These will be followed by five algorithms based on strongly connected component (SCC) decomposition that attempt to meet the constraints while using fewer voters and applying timing-based heuristics. The run-time complexity of each algorithm will be given in terms of $|V|$ (the number of nodes in the circuit graph) and $|E|$ (the number of edges in the circuit graph).

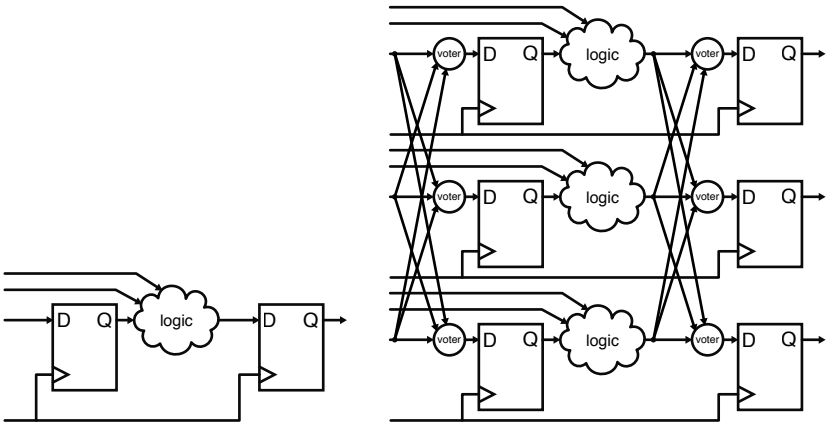
4.1 Simple Algorithms

The algorithms in this section are considered simple because they require only a very simple analysis of the circuit. Although they are simple, they both manage to correctly intersect all of the feedback in a design with voters. In addition, both of these algorithms prevent multiple voters from being placed in a single timing path. One weakness of these algorithms is that they often insert many more voters than are strictly necessary.

4.1.1 Voters Before Every Flip-Flop

The *Voters Before Every Flip-Flop* algorithm places a voter before the data input of every flip-flop in a circuit. For example, the two flip-flops in the circuit of Figure 4.1(a) would be triplicated with voters after each flip-flop as shown in Figure 4.1(b). The algorithm is guaranteed to intersect every cycle with a voter because in standard synchronous circuits, each cycle must have at least one flip-flop. This approach does not insert voters within asynchronous feedback loops. Synchronization of asynchronous feedback loops is beyond the scope of this work. The algorithm also ensures that at most one voter can be placed in a single timing path. This is because a timing path extends from one flip-flop to another flip-flop, and voters are placed only directly before flip-

flops. This reduces the timing impact of the algorithm on the resulting circuit. The algorithm runs in $O(|V|)$ time (each node in the circuit is traversed to find all of the flip-flops).



(a) Original circuit before TMR. (b) The triplicated circuit has voters before each flip-flop.

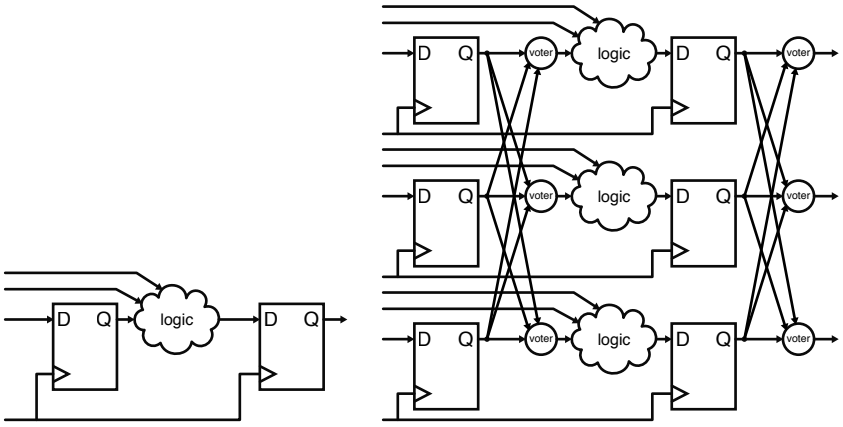
Figure 4.1: *Voters Before Every Flip-Flop* insertion algorithm.

4.1.2 Voters After Every Flip-Flop

The *Voters After Every Flip-Flop* algorithm places a voter *after* the *output* of each flip-flop in a circuit. For example, the two flip-flops in the circuit of Figure 4.2(a) would be triplicated with voters after each flip-flop as shown in Figure 4.2(b). Like the previous algorithm, it is guaranteed to intersect every cycle with a voter, and it inserts at most one voter in a single timing path (reducing the timing impact of the algorithm on the resulting circuit). This algorithm also executes in $O(|V|)$ time.

4.2 Algorithms Based on SCC Decomposition

While the simple algorithms in the previous section satisfy the constraints of synchronization voter insertion, they often insert many more voters than are actually needed. The algorithms that follow are designed to insert fewer voters. They work progressively by indentifying feedback, inserting voters in the feedback, and stopping when there is no feedback left uncut. By inserting



(a) Original circuit before TMR. (b) The triplicated circuit has voters after each flip-flop.

Figure 4.2: *Voters After Every Flip-Flop* insertion algorithm.

fewer voters, these algorithms have the potential to produce circuits with better timing performance and area.

The following five algorithms use analysis of strongly connected components (SCCs) to determine a more efficent voter configuration that cuts all feedback. The SCCs of a graph are the maximal subgraphs in which there is a path from each node to every other node [31]. SCC decomposition is the process of finding all of the SCCs in a graph. The definition of an SCC leads to the following corollaries:

- Each SCC contains at least one cycle,
- No cycle spans more than one SCC,
- There are no cycles outside of the SCCs of a graph,
- Nodes not involved in any cycles will not be found in any SCC.

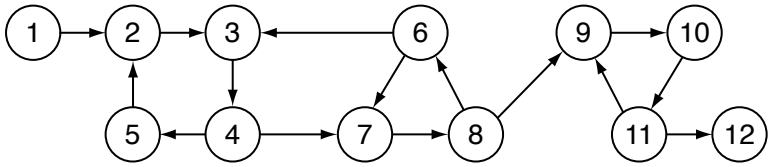
These corollaries suggest that decomposing a graph into SCCs can be a way of simplifying the problem of determining where to place synchronization voters. Since any cycle involves nodes only in a single SCC, each SCC can be treated as a subproblem of the overall synchronization voter insertion problem. Furthermore, graph edges not involved in any of a graph's SCCs need not be considered for synchronization voter insertion.

In order to use SCC decomposition to determine where to insert synchronization voters, the algorithms in this section first generate a directed graph representation of a circuit. Each component instantiation in the circuit netlist becomes a node in the graph. Each net in the netlist becomes a set of edges. For every net in the netlist, the sources and sinks are iterated in a nested fashion and a graph edge is created from each source to every sink. In this manner, the netlist hypergraph representation is converted to a simple directed graph representation.

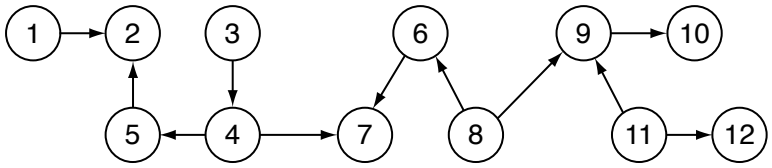
Once a graph representation of the circuit has been created, the algorithms break up the SCCs of the graph into smaller and smaller SCCs by systematically removing edges until all SCCs are dissolved and there are no cycles left in the graph. Edges are removed from the graph representation of the circuit only. Once all of the SCCs are dissolved, voters are inserted in the actual circuit at the locations where edges were removed from the graph representation of the circuit. The process of breaking up SCCs by removing edges is illustrated with the example graph in Figure 4.3(a). This graph contains two SCCs: $\{\{2,3,4,5,6,7,8\}, \{9,10,11\}\}$. The removal of edge $(6,3)$ would break the first SCC into two smaller SCCs, resulting in the SCC decomposition: $\{\{2,3,4,5\}$, $\{6,7,8\}$, $\{9,10,11\}\}$. Removing edge $(10,11)$ would dissolve the third SCC into a feed forward component, giving the SCC decomposition:$\{\{2,3,4,5\}$, $\{6,7,8\}\}$. Additionaly removing edges $(2,3)$ and $(7,8)$ would completely dissolve all of the SCCs in the graph, resulting in the graph shown in Figure 4.3(b). Note that the resulting graph has no feedback loops. Thus, placing synchronization voters in the actual circuit at each of the four locations where edges were removed in the graph representation would break all feedback and ensure proper TMR synchronization.

The algorithms that follow require repeated use of SCC decomposition (several iterations of edge removals are performed and the SCCs are analyzed after each iteration in order to determine what SCCs remain). Several algorithms for SCC decomposition exist, including Kosaraju's algorithm [32] and Tarjan's algorithm [33], both of which run in $O(|V|+|E|)$ time.

The SCC decomposition-based algorithms all have the same basic structure which is summarized with pseudocode in Algorithm 1. The basic structure of the algorithms uses a stack-based method for processing all of the SCCs. To begin, an SCC decomposition of the circuit graph is computed, and all of the SCCs are pushed onto a stack (S). The algorithm iterates over the SCCs in the stack until the stack is empty. During each iteration of the while loop, a single SCC is popped off of the stack for processing. Edges are removed from the SCC to break up the SCC into smaller



(a) Before removing edges.



(b) After dissolving the SCCs by removing selected edges.

Figure 4.3: SCCs can be dissolved by removing edges.

SCCs or single nodes. Any remaining SCCs that result are pushed onto the SCC stack for processing in the next iteration. This process continues until all of the SCCs have been broken into feed forward components. The edge set used to break the feedback of the SCCs indicates the locations of the synchronization voters.

The algorithms that use this structure differ in the manner in which they select edges to remove to dissolve the SCCs into feed forward components. Different edge selection strategies are used to identify feedback edge cutsets that result in, for example, a faster circuit or a fewer number of voters.

4.2.1 Basic SCC Decomposition Algorithm

The *Basic SCC Decomposition Algorithm* is the simplest SCC decomposition based algorithm implemented in this work. The algorithm uses temporary information obtained during the SCC decomposition to completely cut the SCC in a single step. This approach computes SCC decompositions using Kosaraju's algorithm which uses two depth-first searches (DFS). The DFS *back edges* computed during Kosaraju's algorithm are used to remove *all* cycles in the SCC in one pass (removing all DFS back edges from an SCC removes all feedback)[1] This algorithm runs quickly on average, but typically induces poor timing performance in the resulting circuit. The

[1]In some cases, this approach selects edges that correspond to illegal cut locations. When this happens, only the legal voter location edges are removed, an SCC decomposition is recomputed (new SCCs being pushed onto the stack), and the algorithm continues to the next iteration. In the rare case that none of the DFS back edges correspond to legal

Algorithm 1 Basic Structure of SCC Decomposition Algorithms

```
Initialize List L
Initialize Stack S
SCCs ← ComputeSCCDecomposition(graph)
for scc in SCCs do
    S.push(scc)
end for
while S is not empty do
    scc ← S.pop()
    Algorithm specific edge removal
    Add removed edges to List L
    newSCCs ← ComputeSCCDecomposition(scc.nodes())
    for newSCC in newSCCs do
        S.push(newSCC)
    end for
end while
Insert voters on nets corresponding to edges in L
```

runtime complexity is $O(|V|^2 + |V||E|)$, but this is a conservative upper bound. In the best case (when no illegal cuts are encountered), the complexity is $O(|V||E| + |V| + |E|)$. Pseudocode for the algorithm is given in Algorithm 2.

4.2.2 Highest Fanout SCC Decomposition Algorithm

The *Highest Fanout SCC Decomposition Algorithm* uses a heuristic intended to minimize the number of voters used to intersect the cycles of a circuit. The heuristic is based on the intuitive suggestion that a significant amount of feedback can be cut by inserting voters on a single net with high fan-out. Nets with high fan-out are likely to be part of multiple cycles that can all be cut at a single point. At each iteration of the SCC processing while loop, the SCC in question is analyzed to find the node with the highest legal cut fanout. The legal cut output edges from this node are then removed from the graph. In this manner, edge removal is prioritized with high fanout nets. The algorithm runs in $O(|V|^2|E|)$ time, but this is a conservative upper bound. The $|V|^2$ term comes from the fact that each time an SCC is processed by the while loop, each of its nodes must be examined to find the node with the highest fan-out. In practice, the number of times the SCC

voter locations, no edges are removed. The DFS search order is rotated (resulting in a different set of DFS back edges), and the algorithm continues to the next iteration.

Algorithm 2 Basic SCC Decomposition Algorithm

```
Initialize List L
Initialize Stack S
SCCs ← ComputeSCCDecomposition(graph)
for scc in SCCs do
  S.push(scc)
end for
while S is not empty do
  scc ← S.pop()
  if all of scc.backEdges() correspond to legal voter locations then
    Remove scc.backEdges() from graph
    Add scc.backEdges() to L
  else if some of scc.backEdges() correspond to legal voter locations then
    Remove legal edges in scc.backEdges() from graph
    Add legal edges in scc.backEdges() to L
  else if none of scc.backEdges() correspond to legal voter locations then
    Rotate the DFS search order of the graph
  end if
  if not all back edges were removed then
    newSCCs ← ComputeSCCDecomposition(scc.nodes())
    for newSCC in newSCCs do
      S.push(newSCC)
    end for
  end if
end while
Insert voters on nets corresponding to edges in L
```

processing loop executes is far fewer than $|V|$, and the number of nodes in each SCC is generally far fewer than $|V|$. Pseudocode for the algorithm is given in Algorithm 3.

4.2.3 Highest Flip-Flop Fanout SCC Decomposition Algorithm

The *Highest Flip-Flop Fanout SCC Decomposition Algorithm* is similar to the previous algorithm but identifies high fanout nets that originate from flip-flops. This algorithm has two priorities: inserting a small number of voters and reducing the negative impacts of voter insertion on timing performance. When more than one set of voters is inserted in a single timing path (i.e. a path from one register to the next), the voters negatively affect timing performance more than is necessary. For each SCC processed by this algorithm, the flip-flop with the highest legal cut fanout in the SCC is determined. The legal cut output edges from this node are removed. Since a

Algorithm 3 Highest Fanout SCC Decomposition Algorithm

```
Initialize List L
Initialize Stack S
SCCs ← ComputeSCCDecomposition(graph)
for scc in SCCs do
  S.push(scc)
end for
while S is not empty do
  scc ← S.pop()
  highestLegalFanout ← 0
  highestFanoutNode ← null
  for node in scc.nodes() do
    fanout ← ComputeLegalCutFanout(node)
    if fanout > highestLegalFanout then
      highestLegalFanout ← fanout
      highestFanoutNode ← node
    end if
  end for
  Remove output edges of highestFanoutNode from graph
  Add removed edges to List L
  newSCCs ← ComputeSCCDecomposition(scc.nodes())
  for newSCC in newSCCs do
    S.push(newSCC)
  end for
end while
Insert voters on nets corresponding to edges in L
```

timing path consists of the logic from one flip-flop to the next, inserting voters only directly after flip-flop outputs ensures that at most one voter will be inserted per timing path. The runtime of this algorithm is the same as the previous algorithm, $O(|V|^2|E|)$. As with the previous algorithm, this is a conservative upper bound. Pseudocode is given in Algorithm 4. The timing benefits of using this algorithm are demonstrated in the Results chapter.

For an example of the *Highest Flip-Flop Fanout SCC Decomposition Algorithm*, consider Figure 4.4. The figure is a graph representation of a circuit that includes flip-flops that are involved in feedback. The flip-flop nodes in the graph are indicated with gray shading. The initial SCC decomposition performed by the algorithm gives the SCCs $\{\{1,2,4,3\},\{5,7,6\}\}$. The algorithm pushes these SCCs onto a stack and begins processing them with the while loop. The first SCC popped off of the stack is $\{5,7,6\}$. Its only flip-flop node, node 7, is chosen to have its data output

Algorithm 4 Highest Fanout SCC Decomposition Algorithm

```
Initialize List L
Initialize Stack S
SCCs ← ComputeSCCDecomposition(graph)
for scc in SCCs do
    S.push(scc)
end for
while S is not empty do
    scc ← S.pop()
    highestLegalFanout ← 0
    highestFanoutFFNode ← null
    for node in scc.flipFlopNodes() do
        fanout ← ComputeLegalCutFanout(node)
        if fanout > highestLegalFanout then
            highestLegalFanout ← fanout
            highestFanoutFFNode ← node
        end if
    end for
    Remove output edges of highestFanoutFFNode from graph
    Add removed edges to List L
    newSCCs ← ComputeSCCDecomposition(scc.nodes())
    for newSCC in newSCCs do
        S.push(newSCC)
    end for
end while
Insert voters on nets corresponding to edges in L
```

net removed from the graph. In this case, the output net from node 7 is represented by a single edge, $(7,6)$. Edge $(7,6)$ is removed from the graph and an SCC decomposition of the subgraph induced by the nodes $\{5,7,6\}$ is computed. Since the feedback has been removed, no SCCs are found in the subgraph and the while loop continues to the next iteration.

The next iteration pops the SCC $\{1,2,4,3\}$ off of the stack. In this SCC, node 3 is the flip-flop node with the highest fan-out, so it is chosen to have its data output net removed from the graph. Its output net is represented by edges $(3,1)$, $(3,2)$, and $(3,5)$. These edges are removed from the graph (note, however, that this results in only a single voter insertion location) and an SCC decomposition of the subgraph induced by nodes $\{1,2,4,3\}$ is performed. The result of the decomposition is a single remaining SCC: $\{2,4\}$. This SCC is pushed onto the stack and the while loop continues to the next iteration.

In the next iteration, the SCC $\{2,4\}$ is popped off of the stack. Since node 4 is its only flip-flop node, its data output net edges are removed ($(4,2)$ and $(4,6)$). An SCC decomposition of the subgraph induced by $\{2,4\}$ is performed and no SCCs are found. At this point the stack is empty and all of the SCCs have been broken up into feed forward only components. The edges removed by the algorithm result in voters being placed directly after each of nodes 3, 4, and 7. This is sufficient to correctly mitigate the circuit's feedback.

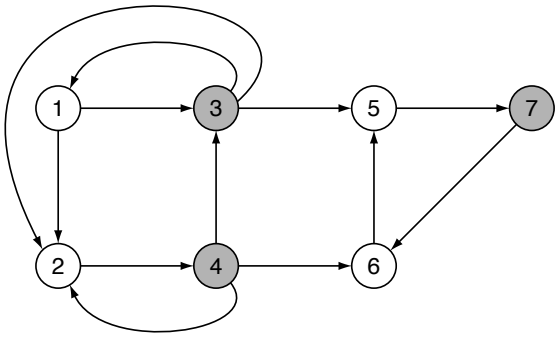


Figure 4.4: Graph representation of a circuit that includes flip-flops involved in feedback.

4.2.4 Highest Fan-in Flip-Flop Input SCC Decomposition Algorithm

The *Highest Fan-in Flip-Flop Input* algorithm uses a heuristic similar to the high fan-out heuristic. It is based on the hypothesis that just as inserting voters after flip-flops with high fan-out can reduce the total number of voters needed to cut all feedback, inserting voters before flip-flops with high fan-in could have a similar effect. In this algorithm, flip-flop fan-in is defined as the number of nets that directly or indirectly feed into the data input of a flip-flop going up to five levels backwards as computed by a depth-limited DFS traversal. For each SCC being processed, this algorithm finds the flip-flop in the SCC with the highest fan-in that also has a data input edge that is a legal voter location and removes its data input edge. The run time of this algorithm is $O(|V|^3 + 2|V|^2|E| + |V||E|^2)$, but this is a conservative upper bound. The extra $|V|$ factor in the dominant term (over the $|V|^2$ of the previous two algorithms) comes from the fact that for each flip-flop node found in each SCC, a depth-limited DFS must be performed to determine the fan-in of

the flip-flop. In practice, the number of nodes traversed in each of these searches is far fewer than $|V|$ because the DFS search is limited to five levels going backwards from the flip-flip. Pseudocode is given in Algorithm 5.

Algorithm 5 Highest Fan-in Flip-Flop Input SCC Decomposition

```
Initialize List L
Initialize Stack S
SCCs ← ComputeSCCDecomposition(graph)
for scc in SCCs do
  S.push(scc)
end for
while S is not empty do
  scc ← S.pop()
  highestFanin ← 0
  highestFaninNode ← null
  for node in scc.flipFlopNodes() do
    fanin ← Compute5LevelFanin(node)
    if fanin > highestFanin and node.dataInputEdge() is a legal voter location then
      highestFanin ← fanin
      highestFaninNode ← node
    end if
  end for
  Remove data input edge of highestFaninNode from graph
  Add removed edge to List L
  newSCCs ← ComputeSCCDecomposition(scc.nodes())
  for newSCC in newSCCs do
    S.push(newSCC)
  end for
end while
Insert voters on nets corresponding to edges in L
```

4.2.5 Highest Fan-in Flip-Flop Output SCC Decomposition Algorithm

This algorithm is very similar to the preceding algorithm and has the same objectives. It is different only in that it inserts voters directly *after* flip-flops with high fan-in instead of directly before. For each SCC being processed, the algorithm finds the flip-flop in the SCC with the highest fan-in and a legal voter location output edge and removes the data output edge. The runtime is the same as that of the previous algorithm, $O(|V|^3 + 2|V|^2|E| + |V||E|^2)$. For the same reasons as the

previous algorithm, this is a conservative upper bound on the complexity. Pseudocode is given in Algorithm 6.

Algorithm 6 Highest Fan-in Flip-Flop Output SCC Decomposition

```
Initialize List L
Initialize Stack S
SCCs ← computeSCCDecomposition(graph)
for scc in SCCs do
  S.push(scc)
end for
while S is not empty do
  scc ← S.pop()
  highestFanin ← 0
  highestFaninNode ← null
  for node in scc.flipFlopNodes() do
    fanin ← Compute5LevelFanin(node)
    if fanin > highestFanin and node.dataOutputEdge() is a legal voter location then
      highestFanin ← fanin
      highestFaninNode ← node
    end if
  end for
  Remove data output edge of highestFaninNode from graph
  Add removed edge to List L
  newSCCs ← ComputeSCCDecomposition(scc.nodes())
  for newSCC in newSCCs do
    S.push(newSCC)
  end for
end while
Insert voters on nets corresponding to edges in L
```

4.3 Conclusion

All of the algorithms in this chapter meet the constraint of inserting voters that intersect all cycles in a circuit while avoiding nets that cannot have voters placed on them due to architectural constraints. Each of the algorithms does so with different priorities. The strengths and weaknesses of each algorithm will become evident in the results presented in the next chapter. All of the algorithms are implemented as part of the open source BL-TMR tool. Information on obtaining this tool is available in Appendix A.

CHAPTER 5. EXPERIMENTAL RESULTS

This chapter will present the results of experiments that were designed to compare the algorithms presented in the preceding chapter in terms of their impact on the timing performance and area of a circuit when applying TMR. It is well known that applying TMR to an FPGA design generally causes poorer timing performance and increases the size of the circuit by at least 3X. A poor voter insertion approach can induce a size increase of well over 3X. The purpose of these experiments is to determine whether some voter insertion strategies are better than others at preserving the timing performance of a circuit and reducing the amount of extra area added by voters when applying TMR.

5.1 Benchmark Designs

A suite of 15 circuit benchmarks including both real-world and synthetic designs was used in the experiments. All of the test designs include some amount of feedback, as synchronization voters are unnecessary in feed forward only designs. The designs were synthesized from VHDL source using Synplify Pro 8.8 synthesis software. The experiments in this chapter were performed on both the Xilinx Virtex and the Xilinx Virtex-5 FPGA architectures (using the xcv1000-fg680-5 part and the xc5vlx110-ff1153-3 part). The benchmark designs are summarized in Table 5.1 with their sizes (in terms of FPGA slices) and critical path lengths for both architectures[1].

The *blowfish* design is a blowfish encrypter. Blowfish is a symmetric block cipher that can be used as a drop-in replacement for other encryption algorithms such as DES. This particular implementation uses a 32-bit key and operates using a feedback loop that greatly reduces the need for parallel encryption circuitry. The large amount of feedback in this design makes it a good candidate for voter insertion experiments.

[1]The Virtex-4 architecture was used for the *ssra_core* benchmark instead of the Virtex architecture because the triplicated design was too large to fit in any of the Virtex parts. Also, the *QPSK* design was not included in the Virtex-5 experiments due to implementation difficulties.

The *DES3* design implements a triple DES encrypter. Triple DES is a block cipher used in cryptography applications. It uses three keys and works by first encrypting data using the first key, decrypting the data with the second key, and finally encrypting the data with the third key. This design was chosen because it is a computationally intensive real world application.

The *QPSK* design is a quadrature phase-shift keying (QPSK) demodulator. QPSK is a digital modulation scheme used in communications applications in which data is encoded using the phase of the carrier signal. This design contains a fair amount of feedback and is another computationally intensive real world application.

The *free6502* design is an FPGA implementation of a simple 8-bit microprocessor that is binary compatible with the 6502 processor. This design is a typical real world FPGA application.

The *T80* design is a CPU core that supports the Z80, 8080, and gameboy instruction sets. It is another good example of a real world FPGA application.

The *MACFIR* design implements a multiply accumulate (MAC) unit using a feedback loop. A MAC unit performs a sum-of-products operation that is useful for computing a convolution sum. Such a design can be used to implement a FIR (finite impulse response) filter for signal processing applications.

The *serial_divide* design is a serial divider that takes a 16-bit dividend and an 8-bit divisor and produces a 16-bit quotient. The feedback necessary for the serial implementation makes it a good candidate for voter insertion experiments.

The *planet*, *s1488*, *s1494*, *s298*, and *tbk* designs are state machine designs from the 1993 International Logic Synthesis Workshop benchmarks. The *tbk* design has the fewest number of states (32) and the *s298* design has the largest number of states (218).

The *Synthetic* design is a design that was crafted to contain both feedback and feed forward logic. It consists of a linear feedback shift register (LFSR) whose output is combined with an input signal using a multiplier and an adder tree. While it is not a typical real world application, it is useful because it contains feedback (making synchronization voters necessary) and uses a large portion of the resources available on the target FPGA device. This is interesting because it results in routing congestion which makes it more difficult for the place and route software to find a routing that meets timing constraints.

The *LFSRs* design is another synthetic design that consists of a large LFSR replicated ten times. It is interesting because it contains a large amount of feedback, and the feedback inherent in an LFSR is of a fairly complex nature, meaning that there are many possible synchronization voter configurations for cutting the feedback.

The *ssra_core* design is a DSP kernel designed by researchers at Los Alamos National Laboratory. It includes a polyphase filter bank as well as FFT and magnitude operations.

Table 5.1: Benchmark test designs with sizes and critical path lengths.

	Virtex		**Virtex-5**	
	Design Size (Slices)	**Critical Path Length**	**Design Size (Slices)**	**Critical Path Length**
blowfish	3416	28.3 ns	355	7.1 ns
des3	658	11.1 ns	430	3.9 ns
qpsk	1041	80.0 ns	-	-
free6502	484	29.6 ns	244	8.1 ns
T80	931	27.8 ns	463	7.4 ns
macfir	658	14.4 ns	341	5.2 ns
serial_divide	40	9.2 ns	32	3.4 ns
planet	144	10.9 ns	72	2.3 ns
s1488	145	9.9 ns	82	2.2 ns
s1494	148	10.4 ns	73	2.2 ns
s298	517	15.8 ns	233	4.0 ns
tbk	155	10.3 ns	76	2.6 ns
synthetic	3061	9.9 ns	1470	2.9 ns
lfsrs	1195	9.0 ns	526	3.3 ns
ssra_core	5393*	6.1 ns*	2785	3.9 ns

5.2 Procedure

The experiments involved applying TMR to each of the test designs using each synchronization voter insertion algorithm. The toolflow used to apply TMR and determine the timing performance and area of each design is shown in Figure 2.3. The BL-TMR tool was used to apply TMR (see Appendix A). The toolflow executes only up to the place and route phase, since at this point the timing performance and area of the resulting circuit can be determined. The number of

voters inserted by each algorithm was recorded in addition to the number of logic slices consumed by the resulting design. The critical path length and area of each design after having TMR applied with each voter insertion algorithm were recorded and compared to the critical path length and area of the original, untriplicated design. In addition, the critical path length and area of each design after having TMR applied with each voter insertion algorithm were also compared to a version of each design that was triplicated without inserting any synchronization voters. Critical path lengths were determined by repeating the place and route process with successively tighter timing constraints until the place and route tool failed to generate a configuration capable of meeting the constraint. Timing constraints were adjusted in 0.1 ns intervals. In this manner, the tightest possible critical path length achievable by the place and route tool was determined for each iteration of each design, including the original untriplicated version.

The experiments were performed on both the Virtex and the Virtex-5 architectures. The Virtex architecture is based on 4-input look up tables while the Virtex-5 architecture is a more modern FPGA architecture based on 6-input look up tables. The results for the two architectures were compared in order to determine whether the effectiveness of the algorithms varies with the FPGA architecture used.

5.3 Timing Results

The critical path length of each algorithm's version of the designs are given in Table 5.2 for the Virtex architecture and in Table 5.3 for the Virtex-5 architecture. The mean values for the critical path are calculated over only 14 of the benchmark designs for the Virtex architecture[2]. The best algorithm's result for each row in the tables is given in bold. A percent increase in critical path length is also given for each design over both the original version and a triplicated version without synchronization voters. The percentages were calculated using the mean critical path length rows of the tables.

The results in Table 5.2 and Table 5.3 show that for the test designs in question, the algorithm that produced the best timing results overall is the *Voters After Every Flip-Flop* algorithm,

[2]The *blowfish* design was excluded from the mean calculations because it did not produce a full row of data. Two of the voter insertion algorithms inserted more voters in this design than could be mapped to the target device. These entries are marked with asterisks in the table.

Table 5.2: Critical path length induced by each voter insertion algorithm using the Virtex architecture.

	Original (Untriplicated)	**TMR w/out voters**	**Voters Before Every FF**	**Voters After Every FF**	**Basic SCC Decomposition**	**Highest Fan-out**	**Highest FF Fan-out**	**Highest Fan-in FF Input**	**Highest Fan-in FF Output**
blowfish	28.3 ns	27.2 ns	35.1 ns	*	43.4 ns	36.5 ns	31.7 ns	35.2 ns	*
des3	11.1 ns	11.0 ns	**13.5 ns**	13.6 ns	17.0 ns	15.0 ns	13.6 ns	13.7 ns	**13.5 ns**
qpsk	80.0 ns	83.7 ns	85.4 ns	85.4 ns	129.3 ns	89.8 ns	**83.9 ns**	84.8 ns	**83.9 ns**
free6502	29.6 ns	30.9 ns	37.5 ns	**31.5 ns**	43.3 ns	39.6 ns	33.1 ns	36.2 ns	32.5 ns
T80	27.8 ns	29.2 ns	33.3 ns	**32.4 ns**	47.4 ns	36.1 ns	33.7 ns	32.7 ns	34.1 ns
macfir	14.4 ns	16.9 ns	18.6 ns	**14.2 ns**	19.4 ns	19.4 ns	19.5 ns	17.2 ns	19.5 ns
serial_divide	9.2 ns	9.5 ns	14.6 ns	**11.6 ns**	15.5 ns	13.9 ns	12.2 ns	14.8 ns	12.2 ns
planet	10.9 ns	10.8 ns	13.2 ns	**12.5 ns**	15.0 ns	12.6 ns	12.6 ns	13.2 ns	12.6 ns
s1488	9.9 ns	10.3 ns	12.6 ns	12.3 ns	14.8 ns	**12.0 ns**	**12.0 ns**	12.8 ns	**12.0 ns**
s1494	10.4 ns	10.7 ns	12.8 ns	12.4 ns	14.8 ns	**12.2 ns**	**12.2 ns**	12.8 ns	**12.2 ns**
s298	15.8 ns	16.2 ns	19.8 ns	19.4 ns	24.7 ns	19.5 ns	**19.1 ns**	20.1 ns	20.1 ns
tbk	10.3 ns	10.6 ns	13.6 ns	13.1 ns	16.9 ns	**12.9 ns**	**12.9 ns**	13.8 ns	**12.9 ns**
synthetic	9.9 ns	10.0 ns	15.9 ns	13.9 ns	**10.0 ns**	10.2 ns	10.4 ns	10.1 ns	10.1 ns
lfsrs	9.0 ns	10.8 ns	13.6 ns	13.9 ns	12.9 ns	13.9 ns	**12.7 ns**	13.5 ns	12.9 ns
ssra_core	6.1 ns	6.5 ns	7.2 ns	7.2 ns	9.2 ns	7.0 ns	7.2 ns	7.0 ns	**6.9 ns**
Mean critical path length	18.17 ns	19.08 ns	22.26 ns	**20.96 ns**	27.87 ns	22.44 ns	21.08 ns	21.62 ns	21.10 ns
% Increase over original	-	5.0%	22.5%	**15.3%**	53.4%	24.5%	16.0%	19.0%	16.1%
% Increase over TMR w/out voters	-	-	16.7%	**9.8%**	46.1%	17.6%	10.5%	13.3%	10.6%

Table 5.3: Critical path length induced by each voter insertion algorithm using the Virtex-5 architecture.

	Original (Untriplicated)	**TMR w/out voters**	**Voters Before Every FF**	**Voters After Every FF**	**Basic SCC Decomposition**	**Highest Fan-out**	**Highest FF Fan-out**	**Highest Fan-in FF Input**	**Highest Fan-in FF Output**
blowfish	7.1 ns	7.8 ns	8.9 ns	**7.0 ns**	9.2 ns	7.6 ns	8.4 ns	8.6 ns	8.0 ns
des3	3.9 ns	3.1 ns	**3.8 ns**	3.9 ns	4.4 ns	4.2 ns	**3.8 ns**	**3.8 ns**	**3.8 ns**
free6502	8.1 ns	9.5 ns	10.9 ns	**8.5 ns**	12.1 ns	10.6 ns	10.0 ns	10.3 ns	10.3 ns
T80	7.4 ns	8.1 ns	8.9 ns	**8.4 ns**	12.2 ns	9.6 ns	8.7 ns	8.5 ns	8.7 ns
macfir	5.2 ns	5.5 ns	5.9 ns	**5.0 ns**	6.2 ns	6.1 ns	6.1 ns	5.6 ns	6.1 ns
serial_divide	3.4 ns	3.5 ns	4.2 ns	**3.5 ns**	4.4 ns	3.8 ns	3.6 ns	4.3 ns	3.6 ns
planet	2.3 ns	2.4 ns	**3.0 ns**	3.1 ns	3.8 ns	3.1 ns	3.1 ns	**3.0 ns**	3.1 ns
s1488	2.2 ns	2.2 ns	2.9 ns	2.9 ns	3.4 ns	**2.8 ns**	**2.8 ns**	2.9 ns	**2.8 ns**
s1494	2.2 ns	2.3 ns	**2.9 ns**	3.0 ns	3.4 ns	3.0 ns	3.0 ns	**2.9 ns**	3.0 ns
s298	4.0 ns	3.9 ns	5.1 ns	4.8 ns	6.4 ns	4.8 ns	**4.7 ns**	5.0 ns	4.9 ns
tbk	2.6 ns	2.7 ns	**3.4 ns**	3.5 ns	4.4 ns	**3.4 ns**	**3.4 ns**	**3.4 ns**	**3.4 ns**
synthetic	2.9 ns	3.1 ns	3.8 ns	3.8 ns	**3.0 ns**	5.4 ns	3.2 ns	4.5 ns	**3.0 ns**
lfsrs	3.3 ns	3.6 ns	4.1 ns	4.2 ns	3.8 ns	4.1 ns	**3.5 ns**	4.2 ns	**3.5 ns**
ssra_core	3.9 ns	4.0 ns	5.0 ns	4.5 ns	6.0 ns	4.6 ns	4.6 ns	4.9 ns	**4.4 ns**
Mean critical path length	4.18 ns	4.41 ns	5.20 ns	**4.72 ns**	5.91 ns	5.22 ns	4.92 ns	5.14 ns	4.90 ns
% Increase over original	-	5.5%	24.4%	**13.0%**	41.4%	25.0%	17.8%	22.9%	17.3%
% Increase over TMR w/out voters	-	-	18.0%	**7.1%**	34.0%	18.5%	11.7%	16.5%	11.2%

which increased the critical path length of the design while adding TMR by only 15.3% over the original design in the Virtex architecture and by only 13.0% in the Virtex-5 architecture. The *Highest Flip-Flop Fanout* and *Highest Fan-in Flip-Flop Output* algorithms also provided very good timing results. It is interesting to note that these three algorithms are the only three that restrict voter placement to locations directly after flip-flops. All of the other algorithms provided notably poorer timing results. The average critical path length increase over a triplicated design without voters for the three algorithms that restrict voter placement to locations directly after flip-flops is only 10.3% for the Virtex architecture and 10.0% for the Virtex-5 architecure. The average for the algorithms that do not restrict voter placement to locations directly after flip-flops is 23.4% for the Virtex architecture and 21.8% for the Virtex-5 architecture. These results suggest that the heuristic of placing voters directy after flip-flops in order to insert at most a single set of voters in any timing path is an effective method of reducing the timing impact of synchronization voter insertion.

5.4 Area Results

The results that pertain to circuit area are given in Table 5.4, Table **??**, and Table 5.5. Table 5.4 gives the number of voters inserted by each algorithm in each design as well as the mean for each algorithm. Table 5.5 and Table 5.6 give the number of slices used by each algorithm's version of each design and the mean number of slices for each algorithm using the Virtex and the Virtex-5 architectures, respectively. As with the timing results, the means for the Virtex architecture are computed over only 14 of the designs[3]

The data in Table 5.5 and Table 5.6 indicate that the algorithms that use SCC decomposition to insert voters only until all of the feedback has been intersected induced comparable and reasonable area increases (an average 263.4% increase for the Virtex architecture and an average 267.6% increase for the Virtex-5 architecture). The *Voters Before Every Flip-Flop* and *Voters After Every Flip-Flop* algorithms, on the other hand, both induced much higher area increases (an average 376.6% increase for the Virtex architecture and an average 400.4% increase for the Virtex-5 architecture). This is because these two algorithms insert many more voters than the algorithms based on SCC decomposition.

[3]The *blowfish* design was again excluded from the mean calculations so that the means can be compared fairly with the critical path length means.

Table 5.4: Number of voters inserted by each voter insertion algorithm.

	Voters Before Every FF	Voters After Every FF	Basic SCC Decomposition	Highest Fan-out	Highest FF Fan-out	Highest Fan-in FF Input	Highest Fan-in FF Output
blowfish	2172	8820*	2448	954	**777**	1359	7158*
des3	321	449	574	435	353	**285**	407
qpsk	1743	1752	1377	165	**96**	117	111
free6502	387	465	402	**237**	264	246	267
T80	732	828	1794	573	**483**	546	645
macfir	4341	4224	**219**	**219**	**219**	**219**	**219**
serial_divide	255	156	114	66	**60**	159	**60**
planet	**18**	21	123	**18**	**18**	**18**	**18**
s1488	**18**	21	99	**18**	**18**	**18**	**18**
s1494	**18**	21	93	**18**	**18**	**18**	**18**
s298	93	96	564	87	**84**	87	87
tbk	198	201	300	**186**	**186**	**186**	**186**
synthetic	13874	13877	722	**290**	326	**290**	**290**
lfsrs	5400	5400	600	**360**	450	450	450
ssra_core	26853	30270	873	684	**636**	**636**	696
Mean number of voters	3875.1	4127.2	561.0	239.7	**229.4**	233.9	248.0

Table 5.5: Number of slices induced by each voter insertion algorithm using the Virtex architecture.

	Original (Untriplicated)	TMR w/out voters	Voters Before Every FF	Voters After Every FF	Basic SCC Decomposition	Highest Fan-out	Highest FF Fan-out	Highest Fan-in FF Input	Highest Fan-in FF Output
blowfish	3416	10293	11255	*	11515	10742	10462	10836	*
des3	658	2056	2251	2312	2387	2295	2242	**2221**	2281
qpsk	1041	3186	4146	3901	4034	3268	**3207**	3240	**3207**
free6502	484	1485	1656	1738	1702	1604	1615	**1574**	1616
T80	931	2831	3168	3304	3744	3111	3079	**3075**	3193
macfir	658	2445	6235	3761	2571	2571	**2566**	2663	**2566**
serial_divide	40	129	300	209	198	166	**164**	235	**164**
planet	144	435	447	443	497	**441**	**441**	447	**441**
s1488	145	438	449	446	491	**444**	**444**	449	**444**
s1494	148	447	**453**	458	489	456	456	**453**	456
s298	517	1551	1614	1594	1849	1600	**1593**	1609	1601
tbk	155	501	595	612	658	603	603	**591**	603
synthetic	3061	12286	**12286**	**12286**	**12286**	**12286**	**12286**	**12286**	**12286**
lfsrs	1195	7429	6585	**6468**	7673	7658	7578	7797	7578
ssra_core	5393	18651	33132	28033	18899	18793	**18745**	18995	18865
Mean number of slices	1040.7	3847.9	5236.9	4683.2	4105.6	3949.7	**3929.9**	3973.9	3950.1
% Increase over original	-	269.7%	403.2%	350.0%	294.5%	279.5%	**277.6%**	281.8%	279.6%
% Increase over TMR w/out voters	-	-	36.1%	21.7%	6.7%	2.6%	**2.1%**	3.3%	2.7%

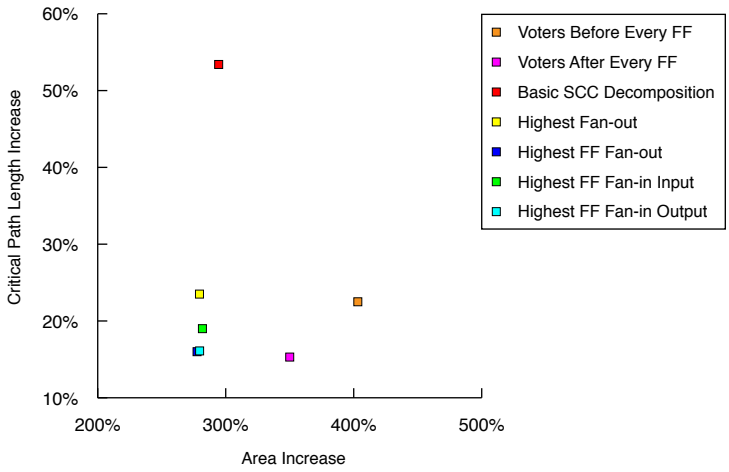
Table 5.6: Number of slices induced by each voter insertion algorithm using the Virtex-5 architecture.

	Original (Unriplicated)	TMR w/out voters	Voters Before Every FF	Voters After Every FF	Basic SCC Decomposition	Highest Fan-out	Highest FF Fan-out	Highest Fan-in FF Input	Highest Fan-in FF Output
blowfish	355	1083	1361	1435	1208	**1030**	1098	1111	1137
des3	430	1242	**1097**	1135	1366	1223	1197	1105	1203
free6502	244	791	998	1006	787	**785**	1006	955	990
T80	463	1504	1672	1718	2157	1603	**1520**	1606	1722
macfir	341	1270	2866	2056	1267	1272	**1213**	1342	**1213**
serial_divide	32	103	214	123	145	118	**107**	150	**107**
planet	72	202	210	198	229	**197**	**197**	210	**197**
s1488	82	204	199	197	211	**187**	**187**	199	**187**
s1494	73	188	197	201	202	**189**	**189**	197	**189**
s298	233	666	750	702	905	**685**	719	704	699
tbk	76	222	**276**	277	333	301	301	345	301
synthetic	1470	5669	8917	8183	5912	**5227**	5953	5323	6543
lfsrs	526	1726	3020	3070	**1959**	2446	2558	2572	2558
ssra_core	2785	10252	15578	14219	10240	10421	10541	10194	**9538**
Mean number of slices	513.0	1794.4	2668.2	2465.7	1922.9	**1834.6**	1913.3	1858.1	1898.9
% Increase over original	-	249.8%	420.1%	380.6%	274.8%	**257.6%**	273.0%	262.2%	270.1%
% Increase over TMR w/out voters	-	-	48.7%	37.4%	7.2%	**2.2%**	6.6%	3.5%	5.8%

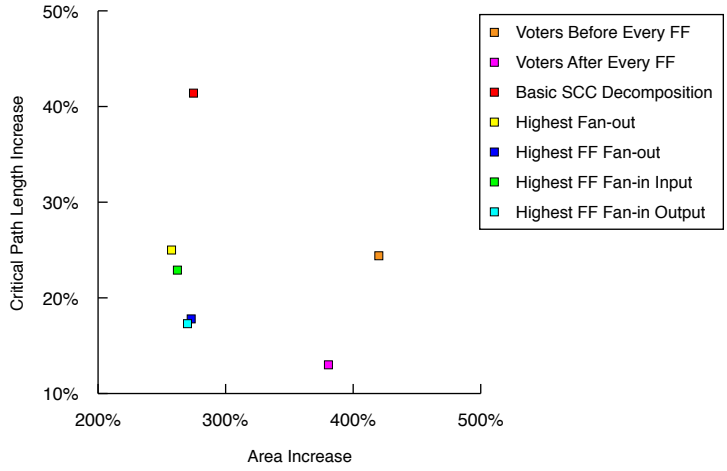
5.5 Analysis

The results of these experiments are demonstrated more clearly in terms of the area/timing performance space in Figure 5.1(a) and Figure 5.1(b). The figures plot the mean percent increase in area (X axis) and critical path length (Y axis) induced by each algorithm for the Virtex and Virtex-5 architectures. As expected, the SCC decomposition algorithms are on the left side of the plot for both architectures, indicating that they induced less of an area increase. Also, the algorithms that restrict voter placement locations to flip-flop outputs are the three lowest algorithms on each plot, indicating that they provide the best timing performance.

One interesting result from these experiments is observed in the number of voters and slices used by the *LFSRs* design in the Virtex architecture. The original design uses 1,195 slices. When TMR is applied without voters, the size of the design increases by about 6.2X to 7,429 slices. Then, when the *Voters Before Every Flip-Flop* and the *Voters After Every Flip-Flop* algorithms are used, each inserts 5,400 voters into the design. However, the algorithms result in an area reduction of 11.4% and 13.0%, respectively, from the number of slices used in the triplicated version without voters. This result is somewhat counterintuitive because the voters are implemented as LUT3 primitives which use resources in the FPGA slices. It appears, however, that the insertion of voters in this particular case prompts the mapping tool to pack more logic into each slice. In



(a) Virtex architecture area/timing results.



(b) Virtex-5 architecture area/timing results.

Figure 5.1: Area/timing performance space of the voter insertion algorithms.

the version of the design without synchronization voters, only 7.1% of the occupied slices in the FPGA are fully utilized (i.e. both of the logic cells in the slice are used). In the *Voters Before Every Flip-Flop* version, 27.5% of the occupied slices are fully utilized, and in the *Voters After Every Flip-Flop* version, 29.8% of the occupied slices are fully utilized. It is possible that the increased logic packing when synchronization voters are inserted is related to the fact that inserting synchronization voters causes the three TMR replicates to be spatially intermixed on the FPGA because each voter requires an input from each TMR replicate. This effect is demonstrated in

Figure 5.5 which shows the layout of the FPGA with the slices color coded by TMR replicate for the three versions of the LFSRs design in question.

Another result that at first appears unexpected is that the *Voters Before Every Flip-Flop* and the *Voters After Every Flip-Flop* algorithms nearly always insert different numbers of voters. This apparent discrepancy can be explained by the circuit structure in Figure 5.3(a). In this figure, two flip-flops receive data from the same source. When the *Voters Before Every Flip-Flop* algorithm is used, only a single set of voters is needed because of the input sharing (see Figure 5.3(b)). However, when the *Voters After Every Flip-Flop* algorithm is used, a set of voters is required for each flip-flop, as shown in Figure 5.3(c). When a circuit contains several instances of structures similar to that of Figure 5.3(a), the resulting disparity between the number of voters inserted by the *Before Every Flip-Flop* and *After Every Flip-Flop* algorithms can grow quite large. The difference between the number of voters inserted by the *Highest Fan-in Flip-Flop Input* and *Highest Fan-in Flip-Flop Output* algorithms can be explained in a similar manner.

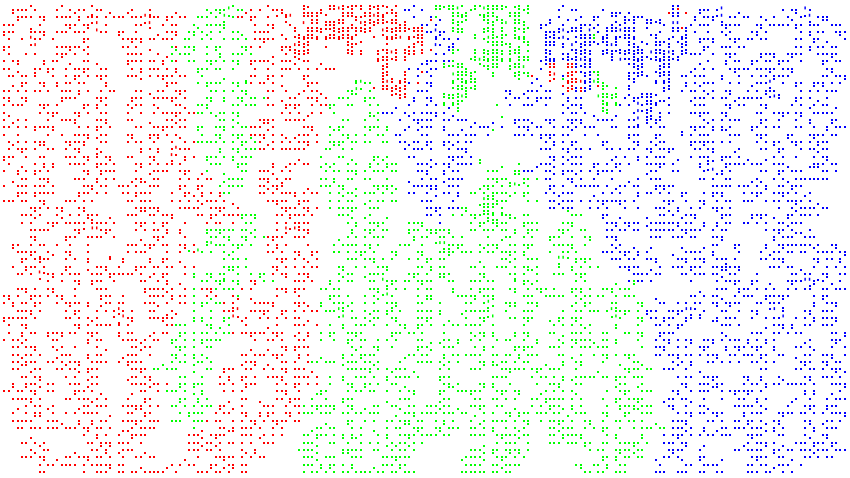
Overall, the best combination of area and timing performance results is obtained by using either the *Highest Flip-Flop Fanout* algorithm or the *Highest Fan-in Flip-Flop Output* algorithm. Their timing results are nearly as good as those of the *Voters After Every Flip-Flop* algorithm, and their area results are far better. However, when sheer timing performance is the only concern, the *Voters After Every Flip-Flop* algorithm is the best choice in the average case.

5.6 Algorithm Execution Time

Table 5.7 reports the average run times of the seven algorithms. As noted previously, the algorithmic complexities of the algorithms are very conservative upper bounds. Due to the nature of standard digital logic circuits, we expect these algorithms to scale much better than their complexities would imply. In practice, the feedback encountered in most digital circuits is simple enough for the algorithms to manage in reasonable time.

5.7 Conclusion

The experimental results in this chapter indicate that the placement of synchronization voters indeed has a significant effect on the area and timing performance of FPGA designs that use TMR. There is a wide variation in both the timing performance and area usage induced by

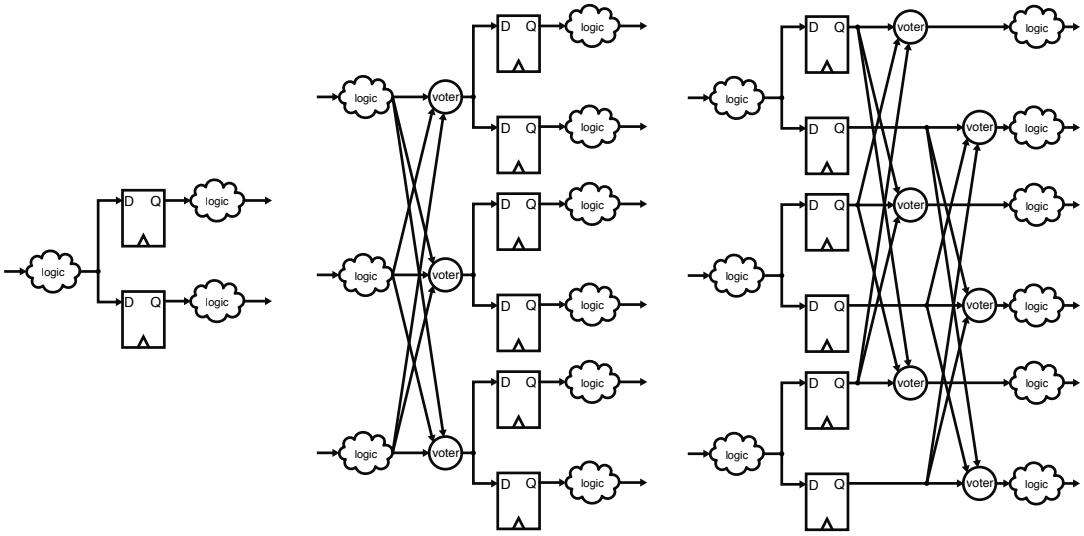


(a) *LFSRs* design without synchronization voters.

(b) *LFSRs* design using the *Before Every Flip-Flop* algorithm.

(c) *LFSRs* design using the *After Every Flip-Flop* algorithm.

Figure 5.2: FPGA slice layout of three versions of the LFSRs design, color coded by TMR replicate.



(a) Untriplicated circuit structure (b) Voters Before Every Flip-Flop (c) Voters After Every Flip-Flop

Figure 5.3: A circuit structure illustrating why putting voters before and after flip-flops changes the total voter count.

the algorithms. In the average case, there is no single voter insertion algorithm that provides the best results for both timing and area. However, the *Highest Flip-Flop Fan-out* and *Highest Fan-in Flip-Flop Outout* algorithms provide a good tradeoff between area and timing performance.

Table 5.7: Algorithm execution times.

	Voters Before Every FF	**Voters After Every FF**	**Basic SCC Decomposition**	**Highest Fan-out**	**Highest FF Fan-out**	**Highest Fan-in FF Input**	**Highest Fan-in FF Output**
blowfish	0.1 s	0.2 s	5.6 s	74.9 s	42.9 s	2322.0 s	1599.6 s
des3	0.5 s	0.5 s	0.5 s	0.9 s	0.6 s	3.1 s	3.9 s
qpsk	0.2 s	0.1 s	13.7 s	9.4 s	7.7 s	17.4 s	20.5 s
free6502	0.3 s	0.3 s	0.2 s	0.5 s	0.6 s	1.1 s	1.0 s
T80	0.9 s	0.1 s	3.9 s	4.6 s	3.8 s	49.4 s	44.2 s
macfir	0.1 s	0.1 s	2.6 s	1.4 s	0.8 s	3.4 s	3.4 s
serial_divide	0.2 s	0.3 s	0.2 s	0.1 s	0.1 s	0.3 s	0.2 s
planet	0.5 s	0.5 s	0.7 s	0.8 s	0.1 s	0.2 s	0.2 s
s1488	0.4 s	0.5 s	0.6 s	0.8 s	0.1 s	0.2 s	0.1 s
s1494	0.6 s	0.7 s	0.6 s	0.9 s	0.2 s	0.2 s	0.2 s
s298	0.3 s	0.5 s	0.4 s	0.6 s	0.9 s	1.2 s	1.5 s
tbk	0.4 s	0.3 s	0.7 s	0.2 s	0.1 s	0.4 s	0.5 s
synthetic	0.2 s	0.5 s	5.8 s	6.9 s	4.1 s	10.0 s	9.2 s
lfsrs	0.3 s	0.4 s	2.3 s	3.7 s	2.4 s	9.8 s	9.6 s
ssra_core	0.8 s	1.3 s	37.2 s	60.5 s	23.9 s	68.6 s	56.9 s
Mean execution time	0.4 s	0.4 s	5.0 s	11.1 s	5.9 s	165.8 s	116.7 s

CHAPTER 6. CONCLUSION

The experimental results obtained in this work used 15 benchmark designs to test the 7 voter insertion algorithms in terms of their impact on circuit area and timing performance. The results indicate that in order to minimize the negative timing impact of TMR, voter insertion algorithms should limit voter locations primarily to flip-flop output nets. The algorithms in this work that follow this heuristic increase the critical path length of a design by only 15.8% on average (over an untriplicated version) using the Virtex architecture and 16.0% using the Virtex-5 architecture, compared to 29.6% using the Virtex architecture and 28.4% using the Virtex-5 architecture for the other algorithms. The best overall algorithms (considering both area and timing performance impacts) are the *Highest Flip-Flop Fan-out* and the *Highest Fan-in Flip-Flop Outout* algorithms. The *Voters After Every Flip-Flop* algorithm can provide slightly better timing results at the cost of increased area overhead due to a greater number of voters. Although the experimental results have identified algorithms that perform the best on average in the timing performance and area categories, anomalies sometimes occur in specific cases due to the random nature of the place and route process. In cases where timing performance and area are critical factors in a space-based mission, the best strategy is to try several different voter insertion algorithms in order to determine the best results possible for the particular circuit and its constraints.

SRAM-based FPGAs can be very useful in space-based computing missions, but mitigation techniques are necessary. TMR used in conjuction with configuration memory scrubbing is the most common technique for FPGAs, but requires synchronization voters for resynchronizing the TMR replicates when faults are corrected. Synchronization voter insertion is tedious and error prone to perform manually, but using the algorithms presented in this work, it is possible to apply TMR and insert synchronization voters using an automated CAD tool. The BL-TMR tool (see Appendix A) is an example of such a tool. All of the algorithms presented in this work are implemented as part of the BL-TMR tool.

There are several possible directions for future work in the area of voter insertion in the context of FPGA implementations of TMR. One direction would be to conduct experiments similar to the experiments in this work using various FPGA architectures to determine how dependent the voter insertion algorithms are on the peculiarities of different architectures. Also, there is room to develop new algorithms based on the heuristics identified in this work. Perhaps existing approximation algorithms for the minimum FES problem [29] could be rectified with FPGA architectures to provide new synchronization voter insertion algorithms. Another possible direction for future work could be to develop algorithms that automatically partition TMR circuits using partitioning voters to increase tolerance of multiple independent upsets while minimizing the impact of the partitioning voters on area and timing.

REFERENCES

[1] D. Ratter, "FPGAs on Mars," Xilinx, Tech. Rep., August 2004, xCell Journal #50.

[2] M. Caffrey, M. Echave, C. Fite, T. Nelson, A. Salazar, and S. Storms, "A space-based reconfigurable radio," in *Proceedings of the 5th Annual International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, September 2002, p. A2.

[3] A. S. Dawood, S. J. Visser, and J. A. Williams, "Reconfigurable FPGAs for real time image processing in space," in *14th International Conference on Digital Signal Processing (DSP 2002)*, vol. 2, 2002, pp. 711–717.

[4] J. Villasenor and B. Hutchings, "The flexibility of configurable computing: Providing the hardware for data-intensive real-time processing," *IEEE Signal Processing Mag.*, pp. 67–84, Sept. 1998.

[5] B. Bridgford, C. Carmichael, and C. W. Tseng, "Single-event upset mitigation selection guide," *Xilinx Application Note XAPP987*, vol. 1, 2008.

[6] C. Carmichael, E. Fuller, P. Blain, and M. Caffrey, "SEU mitigation techniques for Virtex FPGAs in space applications," in *Proceedings of the Military and Aerospace Programmable Logic Devices International Conference (MAPLD)*, Laurel, MD, September 1999.

[7] N. Rollins, M. Wirthlin, M. Caffrey, and P. Graham, "Evaluating TMR techniques in the presence of single event upsets," in *Proceedings fo the 6th Annual International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*. Washington, D.C.: NASA Office of Logic Design, AIAA, September 2003, p. P63.

[8] F. Lima, C. Carmichael, J. Fabula, R. Padovani, and R. Reis, "A fault injection analysis of Virtex FPGA TMR design methodology," in *Proceedings of the 6th European Conference on Radiation and its Effects on Components and Systems (RADECS 2001)*, 2001.

[9] E. Fuller, M. Caffrey, A. Salazar, C. Carmichael, and J. Fabula, "Radiation testing update, SEU mitigation, and availability analysis of the Virtex FPGA for space reconfigurable computing," in *3rd Annual Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, 2000, p. P30.

[10] C. Carmichael, "Triple module redundancy design techniques for Virtex FPGAs," Xilinx Corporation, Tech. Rep., November 1, 2001, xAPP197 (v1.0).

[11] K. J. Gurzi, "Estimates for best placement of voters in a triplicated logic network," *Electronic Computers, IEEE Transactions on*, vol. EC-14, no. 5, pp. 711–717, Oct. 1965, 10.1109/PGEC.1965.264211.

[12] F. Kastensmidt, L. Sterpone, L. Carro, and M. Reorda, "On the optimal design of triple modular redundancy logic for SRAM-based FPGAs," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*. IEEE Computer Society Washington, DC, USA, 2005, pp. 1290–1295.

[13] B. Pratt, M. Caffrey, D. Gibelyou, P. Graham, K. Morgan, and M. Wirthlin, "TMR with more frequent voting for improved FPGA reliability," in *The International Conference on Engineering of Reconfigurable Systems and Algorithms*, July 2008.

[14] "Xilinx TMRTool," Product Brief, Xilinx Corporation, 2006.

[15] "MIL-STD-883," Method 1019.7, Ionizing Radiation (Total Dose) Test Procedure, 2006.

[16] Xilinx, "Qpro Virtex-II 1.5V radiation-hardened QML platform FPGAs," Xilinx, Inc., San Jose, CA, Datasheet DS124, December 2006.

[17] E. Fuller, M. Caffrey, P. Blain, C. Carmichael, N. Khalsa, and A. Salazar, "Radiation test results of the Virtex FPGA and ZBT SRAM for space based reconfigurable computing," in *Proceeding of the Military and Aerospace Programmable Logic Devices International Conference(MAPLD)*, Laurel, MD, September 1999.

[18] K. Morgan, D. McMurtrey, B. Pratt, and M. Wirthlin, "A comparison of TMR with alternative fault-tolerant design techniques for FPGAs," *IEEE transactions on nuclear science*, vol. 54, no. 6 Part 1, pp. 2065–2072, 2007.

[19] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies*, pp. 43–98, 1956.

[20] M. Wirthlin, N. Rollins, M. Caffrey, and P. Graham, "Hardness by Design Techniques for Field Programmable Gate Arrays," in *Proceedings of the 11th Annual NASA Symposium on VLSI Design*. Washington, D.C.: NASA Office of Logic Design, AIAA, 2003, pp. WA11.1 – WA11.6.

[21] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin, "Improving FPGA design robustness with partial TMR," in *44th Annual IEEE International Reliability Physics Symposium Proceedings*, 2006, pp. 226–232.

[22] C. Carmichael, M. Caffrey, and A. Salazar, "Correcting single-event upsets through Virtex partial configuration," *Xilinx Application Notes, XAPP216 (v1. 0)*, 2000.

[23] J. Heiner, N. Collins, and M. Wirthlin, "Fault tolerant ICAP controller for high-reliable internal scrubbing," in *2008 IEEE Aerospace Conference*, 2008, pp. 1–10.

[24] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. LaBel, M. Friendlich, H. Kim, and A. Phan, "Effectiveness of Internal Versus External SEU Scrubbing Mitigation Strategies in a Xilinx FPGA: Design, Test, and Analysis," *IEEE Transactions on Nuclear Science*, vol. 55, no. 4 Part 1, pp. 2259–2266, 2008.

[25] D. Siewiorek and R. Swarz, *Reliable computer systems: design and evaluation*. AK Peters, Ltd.

[26] D. McMurtrey, K. Morgan, B. Pratt, and M. Wirthlin, "Estimating TMR reliability on FPGAs using Markov models." 2008, [Online]. Available: http://hdl.handle.net/1877/644.

[27] Y. Li, "Synchronization issues of TMR crossing multiple clock domains: Analysis and solutions," November 2009, CHREC B5b-09 project technical report.

[28] R. Karp, "Reducibility among combinatorial problems," *Complexity of computer computations*, vol. 43, pp. 85–103, 1972.

[29] G. Even, "Approximating minimum feedback sets and multicuts in directed graphs," *Algorithmica*, vol. 20, no. 2, pp. 151–174, 1998.

[30] P. Eades, X. Lin, and W. Smyth, "A fast and effective heuristic for the feedback arc set problem," *Information Processing Letters*, vol. 47, no. 6, pp. 319–323, 1993.

[31] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*. The MIT press, 2001.

[32] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers & Mathematics with Applications*, vol. 7, no. 1, pp. 67 – 72, 1981, DOI: 10.1016/0898-1221(81)90008-0.

[33] R. Tarjan, "Depth-first search and linear graph algorithms," in *Switching and Automata Theory, 1971., 12th Annual Symposium on*, Oct. 1971, pp. 114–121.

APPENDIX A. OBTAINING AND USING THE BYU-LANL TRIPLE MODULAR REDUNDANCY (BL-TMR) TOOL

A.1 Obtaining the BL-TMR Tool

The BYU-LANL Triple Modular Redundancy (BL-TMR) Tool is the automated TMR tool that was used to perform the voter insertion experiments in this work. The tool was extended to support all of the voter insertion algorithms presented in the work. The BL-TMR Tool is an open source project that can be obtained from `http://sourceforge.net/projects/byuediftools`.

A.2 Introduction

The BYU-LANL Triple Modular Redundancy (BL-TMR) Tool is an EDIF-based tool capable of inserting redundancy in an FPGA design in order to increase reliability. Triple modular redundancy (TMR) and/or duplication with compare (DWC) are applied to an EDIF input file according to the options chosen by the user. Several different voter insertion algorithms are available for use with TMR.

A.3 Replication Toolflow

The tool is split into several subtools. This allows the user to adjust various command-line options in one phase, and then move onto the next phase. The toolflow for the tool is illustrated in Figure A.1

All of the subtools will be described briefly in this section. Full documentation of the toolflow is available at `http://sourceforge.net/projects/byuediftools`. The documentation for the subtools that were necessary to perform the experiments in this work (i.e. JEdifBuild, JEdifAnalyze, JEdifNMRSelection, JEdifVoterSelction, JEdifNMR, and JEdifReplicationQuery) will be repeated from the documentation available online in the sections that follow.

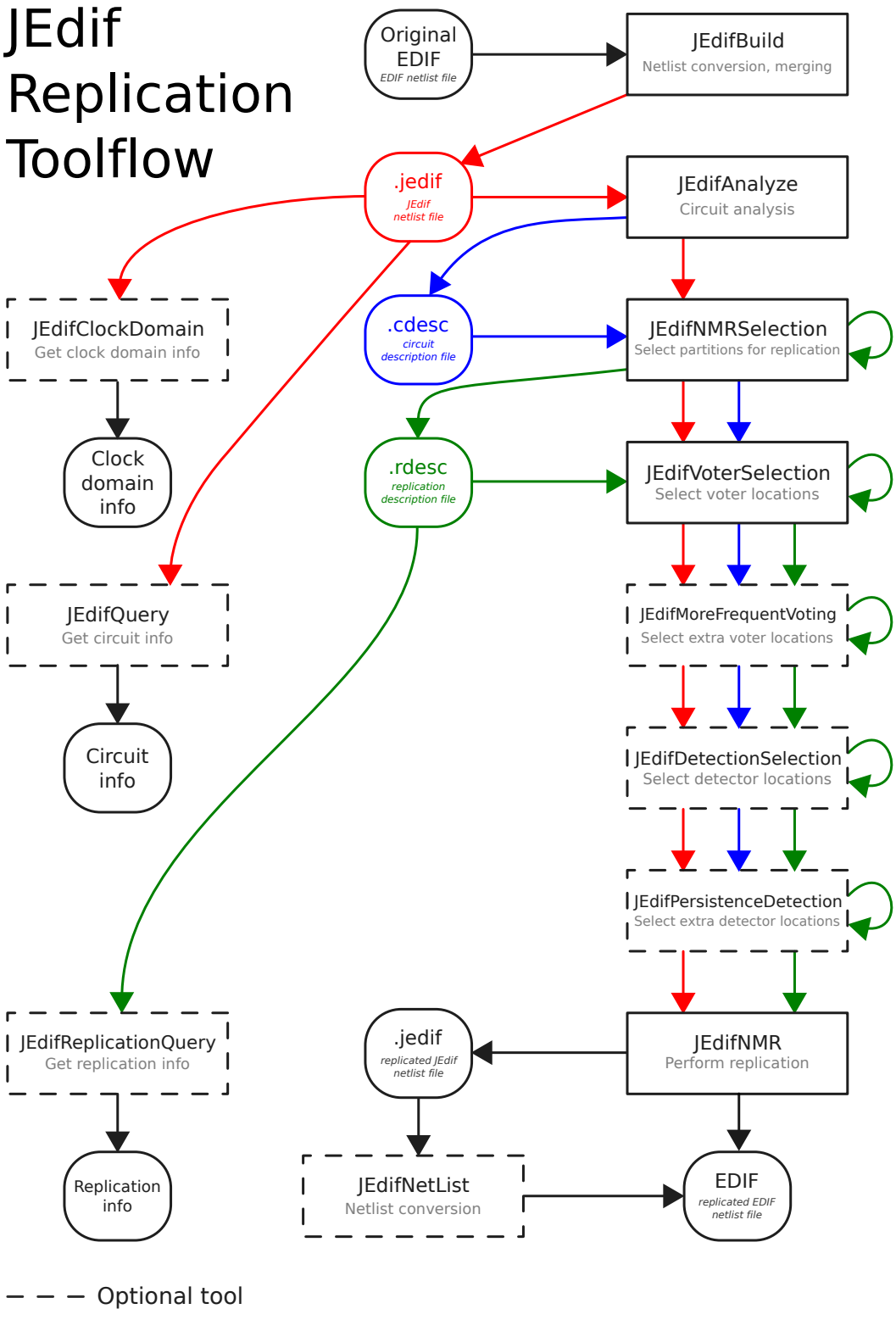


Figure A.1: The BL-TMR Tool Flow.

A.3.1 JEdifBuild

JEdifBuild creates merged netlists in a .jedif file format from one or more .edf files. By default, JEdifBuild also flattens the design and optionally performs FMAP removal, RLOC removal, SRL replacement, and half-latch removal. The .jedif file format is an intermediate file format used by the remainder of the replication tools.

A.3.2 JEdifAnalyze

JEdifAnalyze performs some basic circuit analysis necessary for subsequent executables. In particular, it performs feedback and IOB analysis. The results of JEdifAnalyze are saved in a circuit description file (.cdesc) required by later executables.

A.3.3 JEdifNMRSelection

JEdifNMRSelection determines which parts of a design will be replicated. This executable can be run in multiple passes to select different parts of a design for different kinds of replication. Each run of JEdifNMRSelection can select portions of a design for a single replication type (i.e. duplication, triplication). Design portions can be selected for replication based on available space or specific cell types, instances, ports, and clock domains specified by the user. The results of JEdifNMRSelection are saved in a replication description (.rdesc) file. This file can be modified by subsequent runs of this and other executables in the toolflow.

A.3.4 JEdifVoterSelection

JEdifVoterSelection determines the locations where voters will be inserted into a triplicated design (or triplicated portions of a design). Voter locations are determined using a feedback cutset algorithm (i.e. one of the algorithms presented in this work) and rules for voting where downscaling is necessary. The results are added into the replication description (.rdesc) file.

A.3.5 JEdifMoreFrequentVoting

Optional: JEdifMoreFrequentVoting inserts extra voters for more frequent voting within a design based on a logic levels threshold or a total number of desired partitions.

A.3.6 JEdifDetectionSelection

Optional: JEdifDetectionSelection determines detector locations for both triplicated and duplicated design portions using user-specified options. Like JEdifNMRSelection, this tool is designed to be run in multiple passes (only one replication type can be processed per pass). Results are saved in the replication description file (.rdesc).

A.3.7 JEdifPersistenceDetection

Optional: JEdifPersistenceDetection determines additional detector locations necessary for classifying persistent/non-persistent errors detected in a design. It is designed to be run in multiple passes. Results are saved in the replication description (.rdesc) file.

A.3.8 JEdifNMR

JEdifNMR performs the replication selected by previously run tools. Information about what to replicate and where to insert voters/detectors is obtained from the replication description (.rdesc) file created by the previous steps.

A.3.9 Other JEdif tools

JEdifNetList

JEdifNetlist converts a netlist in .jedif format to EDIF (.edf) format for use with other standard EDIF tools.

JEdifQuery

JEdifQuery is a tool used to query the contents of a .jedif file and to provide summary information about the EDIF design contained within.

JEdifReplicationQuery

JEdifReplicationQuery is a tool used to query the contents of a replication description file (.rdesc) and provide summary information about the kind of replication that will be performed on a design. It reports information about replication types, organs to be inserted (i.e. voters, detectors), and detection error outputs.

JEdifClockDomain

The JEdifClockDomain tool is a .jedif based tool to analyze FPGA designs to obtain information about the clock(s). The tool first identifies all clocks in a design. This information is then used to optionally display other information, such as classifying Xilinx primitives into one or more domains, showing clock crossings, etc.

A.4 JEdifBuild Options

JEdifBuild creates merged netlists in a .jedif file format from multiple .edf files. By default, JEdifBuild also flattens the design and optionally performs FMAP removal, RLOC removal, SRL replacement, and half-latch removal (functions performed by JEdifSterilize in previous versions of the toolflow). The .jedif file format is an intermediate file format used by the remainder of the replication tools.

Although flattening occurs by default, it can be disabled with the `--no_flatten` option. It is also possible to specify that specific cell types should not be flattened. This can be accomplished by adding a `'do_not_flatten'` property to the cell in the .edf file as follows:

```
(property do_not_flatten (boolean (true)))
```

If this property is used on a cell that is a black box in the main design file and is merged in from a separate .edf or .edn file, the property should be specified in the black box *definition* file, not in the main design .edf file.

It should be noted that designs that are not flattened will not be replicated properly. Any unflattened cells will be replicated as an atomic unit, preventing proper voter insertion. Use the `'do_not_flatten'` property only when this is the desired behavior.

Options can be specified on the command line or in a configuration file in any order. This section describes each of these options in detail.

```
> java edu.byu.ece.edif.jedif.JEdifBuild --help
Options:
  [-h|--help]
  [-v|--version]
  <input_file>
  [(-o|--output) <output_file>]

  [(-d|--dir) dir1,dir2,...,dirN ]
  [(-f|--file) file1,file2,...,fileN ]

  [--no_flatten]
  [--no_open_pins]
  [--blackboxes]
  [--no_delete_cells]
  [--pack_registers <{i|o|b|n}>]

  [--replace_luts]
  [--remove_fmaps]
  [--remove_rlocs]
  [--remove_hl]
  [--hl_constant <{0|1}>]
  [--hl_use_port <hl_port_name>]
  [--hl_no_tag_constant]

  [(-p|--part) <part>]

  [--write_config <config_file>]
  [--use_config <config_file>]

  [--log <logfile>]
  [--debug[:<debug_log>]]
  [(-V|--verbose) <{1|2|3|4|5}>]
  [--append_log]
```

A.4.1 File options: input, output, etc.

The following options specify the top-level input EDIF file, any auxiliary EDIF files, and the destination EDIF file.

`<input_file>`

Filename and path to the EDIF source file containing the top-level cell to be converted. This is the only required parameter.

Allowed filename extensions are:

- Parsable EDIF: edn,edf,ndf
- Binary netlist (Blackboxes): ngc,ngo
- Blackbox Utilization: bb

Parsable EDIF files will be parsed and included in the algorithms. Binary netlist files are not parsable by JEdifBuild, but the program recognizes them as blackboxes, and will not complain about not finding the entity. Blackbox utilization files allow the user to specify the resource use of the blackboxes to help in the utilization estimate and partial tmr algorithms. The file format is "Resource:Number". Below is an example:

myblackbox.bb:

```
BRAM:1
FF:400
LUT:100
```

This entity, named myblackbox, uses 1 BRAM, 400 Flipflops and 100 LUTS

`(-o|--output) <output_file>`

Filename and path to the jedif output file.

Default: `<input file>.jedif` in the current working directory.

`(-d|--dir) dir1,dir2,...,dir3`

Comma-separated list of directories containing external EDIF files referenced by the top-level EDIF file. The current working directory is included by default and need not be specified. Multiple `-d` options may be specified.

Example: `-d aux_files,/usr/share/edif/common -d moreEdifFiles/`

`(-f|--file) file1,file2,...,fileN`

Similar to the previous option, but rather than specifying directories to search, each external EDIF file is named explicitly—including the path to the file. Multiple `-f` options may be specified.

Example: `-f multBox.edn,src/adder.edf -f /usr/share/edif/blackBox.edf.`

A.4.2 Maintenance Options

The following options allow some control over what happens during the conversion process

`--no_flatten`

By default, JEdifBuild will flatten the EDIF files. Flattening is required by the TMR tools, but other applications may wish to keep the hierarchical design.

`--no_open_pins`

Do not allow the parser to infer open pins on black box definitions.

`--blackboxes`

Allow parser to continue if blackbox definitions are not found.

`--no_delete_cells`

By default JEdifBuild will remove unused cells to reduce the size of the final .jedif file. However, the user can request that these cells be retained for future use.

`--pack_registers` {**i**|**o**|**b**|**n**}

By default, the BL-TMR tool treats all ports on the input EdifCell as top-level ports (those that will be the inputs and outputs of the FPGA). The half-latch tool must therefore treat any FFs that will be packed into the IOBs differently than other FFs (at least with Virtex devices). This option allows the user to specify which IOBs the registers should be packed into: inputs (*i*), outputs (*o*), both (*b*), or none (*n*). The default is to pack both input and output registers.

A.4.3 Sterilize Options

`--replace_luts`

When this option is specified, the tool replaces LUT RAMs and SRLs with flip-flop equivalents. This option is useful because bitstream scrubbing cannot be used with designs that contain LUT RAMs or SRLs.

`--remove_fmaps`

Remove FMAPs from the input design.

`--remove_rlocs`

Remove ALL RLOCs in the design. The replication tools will not work correctly if a design contains RLOCs.

`--remove_hl`

Remove half-latches in the input design before performing replication.

Note: Not *all* half-latches can be removed at the EDIF level for all architectures. Some post-processing may be necessary.

`--hl_const` $\{0, 1\}$

Sets the polarity of the half-latch constant to be used, whether an internally-generated constant or a top-level port.

Valid options are `0` and `1`. Default: `0`.

`--hl_use_port <hl_port_name>`

Specify a top-level port to use in place of half-latches when using half-latch removal. The top-level port will have the name specified with this option and the polarity (1 or 0) specified with the `--hl_const` option.

`--hl_no_tag_constant`

When half-latch removal is used, a constant comes from either a constant generator cell (ROM16X1) or a port specified by the user. In either case, the input buffer for the port or the generator cell should be triplicated (or duplicated) to ensure reliability. By default, such instances are tagged with an EDIF property called '`half_latch_constant`' so that they can be automatically selected for replication by JEdifNMRSelection later in case partial replication is used. This option disables the default behavior of tagging safe constant instances with this property.

A.4.4 Target Part Options

`--part <partname>`

Target architecture for the design. Used to take into account part-specific properties, including the number of resources available in each part. Valid parts include all parts from the *Virtex* and *Virtex2* product lines, represented as a concatenation of the part name and package type. For example, the "Xilinx Virtex 1000 FG680" is represented as `XCV1000FG680`. This argument is *not* case-sensitive. The default is `xcv1000fg680`.

A.4.5 Configuration File Options

The BL-TMR tools can use configuration files in place of command-line parameters. If a parameter is specified in a configuration file, it will be passed to the BL-TMR tool, unless it is overridden by the same argument on the command-line.

`--use_config <config_file>`

Specify a configuration file from which to read parameters.

`--write_config <config_file>`

Write the current set of command-line parameters to a configuration file and exit. The parameters will be parsed to ensure they are valid, but the BL-TMR tool will not run. Note that only the parameters on the command-line are stored in the configuration file. When using `--write_config`, any use of `--use_config` is ignored. This is to prevent complicated cascades of configuration files combined with command-line options.

Examples:

- `--write_config JonSmith.conf` will write the command-line parameters to the file `JonSmith.conf` in the current directory.
- `--write_config /usr/lib/BL-TMR/common.conf` will write the command-line parameters to the file `/usr/share/BL-TMR/common.conf`.
- See section A.10.11, "Using Configuration Files," for more information.

A.4.6 Logging options

`--log <logfile>`

Specifies an alternate file for logging output.

`--debug[:<debug_log>]`

Specifies a file for logging the debugging output.If no file specified, debug output is printed to the log file.

`(-V|--verbose) <{1|2|3|4|5}>`

Sets the verbosity level: 1 prints only errors, 2 warnings, 3 normal, 4 log to stdout. 5 prints debug information. (default: 3)

`--append_log`

Append to the logfile instead of replacing it.

A.5 JEdifAnalyze

JEdifAnalyze performs some basic circuit analysis necessary for subsequent executables. In particular, it performs feedback and IOB analysis. The results of JEdifAnalyze are saved in a circuit description (.cdesc) file required by later executables.

The following options control the feedback and IOB analysis performed by this executable. The results of the analysis affect the execution of later steps in the toolflow.

```
>java edu.byu.ece.edif.jedif.JEdifAnalyze
Options:
  [-h|--help]
  [-v|--version]

  <input_file>
  (-o|--output) <output>

  [--pack_registers <{i|o|b|n}>]
  [--use_bad_cut_conn]
  [--no_iob_feedback]

  [(-p|--part) <part>]

  [--write_config <config_file>]
  [--use_config <config_file>]
```

```
[--log <logfile>]
[--debug[:<debug_log>]]
[(-V|--verbose) <{1|2|3|4|5}>]
[--append_log]
```

A.5.1 File Options

`<input_file>`

Filename and path to the jedif source file to be analyzed

`(-o|--output) <output_file>`

Filename and path to the ciruit description (.cdesc) file that will be output.

A.5.2 Analysis Options

`--pack_registers` {**i**|**o**|**b**|**n**}

By default, the BL-TMR tool treats all ports on the input EdifCell as top-level ports (those that will be the inputs and outputs of the FPGA). The half-latch tool must therefore treat any FFs that will be packed into the IOBs differently than other FFs (at least with Virtex devices). This option allows the user to specify which IOBs the registers should be packed into: inputs (*i*), outputs (*o*), both (*b*), or none (*n*). The default is to pack both input and output registers.

`--use_bad_cut_conn`

Use bad cut group connectivity graph

`--no_iob_feedback`

Use this option to exclude IOBs from the feedback analysis. This is useful when a top-level inout port is involved in feedback but by design will never be written to and read from at the same time. Thus there is no *real* feedback. Using this option may greatly reduce the amount of feedback found in the design and thus reduce the number of voters inserted.

A.5.3 Target Part Options

`--part <partname>`

Target architecture for the design. Used to take into account part-specific properties, including the number of resources available in each part. Valid parts include all parts from the *Virtex* and *Virtex2* product lines, represented as a concatenation of the part name and package type. For example, the "Xilinx Virtex 1000 FG680" is represented as `XCV1000FG680`. This argument is *not* case-sensitive. The default is `xcv1000fg680`.

A.5.4 Configuration File Options

The BL-TMR tools can use configuration files in place of command-line parameters. If a parameter is specified in a configuration file, it will be passed to the BL-TMR tool, unless it is overridden by the same argument on the command-line.

`--use_config <config_file>`

Specify a configuration file from which to read parameters.

`--write_config <config_file>`

Write the current set of command-line parameters to a configuration file and exit. The parameters will be parsed to ensure they are valid, but the BL-TMR tool will not run. Note that only the parameters on the command-line are stored in the configuration file. When using `--write_config`, any use of `--use_config` is ignored. This is to prevent complicated cascades of configuration files combined with command-line options.

Examples:

- `--write_config JonSmith.conf` will write the command-line parameters to the file `JonSmith.conf` in the current directory.
- `--write_config /usr/lib/BL-TMR/common.conf` will write the command-line parameters to the file `/usr/share/BL-TMR/common.conf`.
- See section A.10.11, "Using Configuration Files," for more information.

A.5.5 Logging options

`--log <logfile>`

Specifies an alternate file for logging output.

`--debug[:<debug_log>]`

Specifies a file for logging the debugging output.If no file specified, debug output is printed to the log file.

`(-V|--verbose) <{1|2|3|4|5}>`

Sets the verbosity level: 1 prints only errors, 2 warnings, 3 normal, 4 log to stdout. 5 prints debug information. (default: 3)

`--append_log`

Append to the logfile instead of replacing it.

A.6 JEdifNMRSelection

JEdifNMRSelection determines which parts of a design will be replicated. This executable can be run in multiple passes to select different parts of a design for different kinds of replication. Each run of JEdifNMRSelection can select portions of a design for a single replication type (i.e. duplication, replication). Design partitions can be selected for replication based on available space or specific cell types, instances, ports, and clock domains specified by the user. The results of JEdifNMRSelection are saved in a replication description (.rdesc) file. This file can be modified by subsequent runs of this and other executables in the toolflow.

See Section A.10 for examples of JEdifNMRSelection usage in common scenarios.

```
>java edu.byu.ece.edif.jedif.JEdifNMRSelection
Options:

  [-h|--help]
  [-v|--version]
```

```
<input_file>
(-r|--rep_desc) <rep_desc>
(-c|--c_desc) <c_desc>

--replication_type <replication_type>
[--continue]
[--override]

[--full_nmr]
[--no_partial_nmr]
[--nmr_p Port name1,Port name2,...,Port nameN ]
[--nmr_inports]
[--nmr_outports]
[--no_nmr_p port1,port2,...,portN ]
[--nmr_c cell_type1,cell_type2,...,cell_typeN ]
[--nmr_clk clock_domain1,clock_domain2,...,clock_domainN ]
[--nmr_i cell_instance1,cell_instance2,...,cell_instanceN ]
[--no_nmr_c cell_type1,cell_type2,...,cell_typeN ]
[--no_nmr_clk clock_domain1,clock_domain2,...,clock_domainN ]
[--no_nmr_i cell_instance1,cell_instance2,...,cell_instanceN ]
[--no_nmr_feedback]
[--no_nmr_input_to_feedback]
[--no_nmr_feedback_output]
[--no_nmr_feed_forward]
[--scc_sort_type <{1|2|3}>]
[--do_scc_decomposition]
[--input_addition_type <{1|2|3}>]
[--output_addition_type <{1|2|3}>]

[--merge_factor <merge_factor>]
[--optimization_factor <optimization_factor>]
[--factor_type <{DUF|UEF|ASUF|CF}>]
[--factor_value <factor_value>]
[--ignore_hard_resource_utilization_limits]
[--ignore_soft_logic_utilization_limit]

[(-p|--part) <part>]

[--write_config <config_file>]
[--use_config <config_file>]

[--log <logfile>]
[--debug[:<debug_log>]]
[(-V|--verbose) <{1|2|3|4|5}>]
[--append_log]
```

A.6.1 File Options

`<input_file>`

Filename and path to the jedif source file to be replicated.

`(-r|--rep_desc) <rep_desc>`

Filename and path to the replication description (.rdesc) file to be written. The file will be modified by subsequent runs of JEdifNMRSelection when the `--continue` option is used.

`(-c|--c_desc) <c_desc>`

Filename and path to the circuit description (.cdesc) file generated by JEdifAnalyze.

A.6.2 Replication Type Options

`--replication_type <replication_type>`

Replication type to use for this run. Must be one of `triplication` or `duplication`.

`--continue`

Select this option to build selection results on top of results from previous runs. If not selected, the replication description file (.rdesc) will be overwritten completely instead of just modified. Normally, when continuing NMR selection with this flag, only instances that have not yet been selected for a replication type will be considered. Overriding replication types for instances and ports can be accomplished by using the `--override` flag in conjunction with this flag.

`--override`

This flag may be used in conjunction with the `--continue` flag in order to override the replication type selections for instances that have already been selected.

A.6.3 Partial Replication Options

`--full_nmr`

Fully replicate the design, skipping all partial replication analysis. This method is preferred when the design is expected to fit in the target part with full replication of every resource since some time-consuming algorithms are skipped. Resource utilization estimates will still function, stopping replication and warning the user if the full replicated design is not expected to fit in the target part.

Note: `--full_nmr` will replicate all logic within the design; however, top-level ports are not replicated by default. To replicate top-level ports, use the `--nmr_inports` and `--nmr_outports` options.

`--no_partial_nmr`

This option will disable the use of partial NMR analysis to determine which parts of the circuit to replicate. Use this option in conjuction with the `--nmr_i` and `--nmr_c` options for explicit control of replicated instances. This option need not be used when the `--full_nmr` option is used.

`--nmr_p port1,port2,...,port3`

Comma-separated list of ports to be replicated.

`--nmr_inports`

Replicate all top-level input ports. The resulting EDIF file will have replicated input ports for every input port in the original design, with names such as `inputPort_TMR_0`, `inputPort_TMR_1`, and `inputPort_TMR_2`.

`--nmr_outports`

Force replication of all top-level output ports. The resulting EDIF file will have replicated output ports for every output port in the original design, with names such as `outputPort_TMR_0`, `outputPort_TMR_1`, and `outputPort_TMR_2`.

`--no_nmr_p port1,port2,...,portN`

Prevent replication of specific top-level port(s), specified as a comma-separated list. Used in conjunction with `--nmr_inports` and `--nmr_outports`. For example, the following will replicate all input ports except the clock and reset ports, assuming `Clk` and `rst` are the (case-sensitive) names of the clock and reset input ports, respectively:

```
--nmr_inports --no_nmr_p Clk,rst
```

`--nmr_c cell_type1,cell_type2,...,cell_typeN`

Force replication of specific cell type(s), specified as a comma-separated list. All instances of the types specified will be replicated. `--nmr_c` takes precedence over `--no_nmr_c`. Mulitple `--nmr_c` lists may be specified.

Examples:

- `--nmr_c bufg,ibufg,fdc`
- `--nmr_c bufg,ibufg --nmr_c fdc`

`--nmr_clk clock_domain1,clock_domain2, ...,clock_domainN`

Force replication of the specified clock domain(s), specified as a comma-separated list. Each clock domain should be specified with it's full path, not including the top level instance name, each level being separated by '/' Note: `--no_nmr_clk` takes precedence over `--nmr_clk`. Multiple `--nmr_clk` lists may be specified.

`--nmr_i cell_instance1,cell_instance2, ...,cell_instanceN`

Force replication of specific cell instance(s), specified as a comma-separated list. Note: `--no_nmr_i` takes precedence over `--nmr_i`. Multiple `--nmr_i` lists may be specified.

Example: `--nmr_i clk_bufg,multiplier16/adder16/fullAdder0`

`--no_nmr_c cell_type1,cell_type2,...,cell_typeN`

Prevent replication of specific cell type(s), specified as a comma-separated list. Multiple `--no_nmr_c` lists may be specified.

Example: `--no_nmr_c bufg,ibufg,fdc`

`--no_nmr_clk clock_domain1, clock_domain2,...,clock_domainN`

Prevent replication of specified clock domain(s), specified as a comma-separated list. Multiple `--no_nmr_c` lists may be specified.

Example: `--no_nmr_clk clk_c`

`--no_nmr_i cell_instance1, cell_instance2,...,cell_instanceN`

Prevent replication of specific cell instance(s), specified as a comma-separated list. Multiple `--no_nmr_i` lists may be specified.

Example: `--no_nmr_i clk_bufg,multiplier16/adder16/fullAdder0`

`--no_nmr_feedback`

Skip replication of the feedback section of the design. Is it *not* recommended to skip replication of the feedback section, as it is the most critical section for SEU mitigation.

`--no_nmr_input_to_feedback`

Skip replication of the portions of the design that "feed into" the feedback sections. These portions also contribute to the "persistence" of the design and should be included in replication, when possible.

`--no_nmr_feedback_output`

Skip replication of the portions of the design which are driven by the feedback sections of the design.

`--no_nmr_feed_forward`

Skip replication of the portions of the design which are not related to feedback sections (neither drive nor are driven by the feedback sections).

A.6.4 SCC Options

The following options control how BL-TMR handles strongly connected components (SCCs) and related logic. An SCC, by definition, is a maximal subgraph of circuit components that are mutually reachable. That is, following the flow of data, every component in the SCC can be reached from every other. In an SCC, each component is related to every other component. The feedback section is defined as the combination of all the strongly-connected components (SCCs). The following options determine the order in which SCCs and related logic are replicated as well as whether or not SCCs can be partitioned into smaller components.

`--ssc_sort_type` $\{1,2,3\}$

Choose the method the BL-TMR tool uses to partially replicate logic in the "feedback" section of the design. Option 1 replicates the largest SCCs first. Option 2 replicates the smallest first. Option 3 replicates the SCCs in topological order.

This option only affects the resulting circuit if only some of the feedback section is replicated. If all or none of the "feedback" section is replicated, the three options produce identical results. The difference lies in what *order* the logic in this section is added and thus what part of it is replicated if there are not enough resources available to replicate the entire section. Valid options are `1`, `2`, and `3`. Default: `3` (topological order).

`--do_scc_decomposition`

Allow portions of strongly-connected components (SCCs) to be included for replication.

By default, if a single SCC is so large that it cannot be replicated for the target part, it is skipped. This option allows large SCCs to be broken up into smaller pieces, some of which may fit in the part. This is only useful if there are not enough resources to replicate the entire set of SCCs.

`--input_addition_type` $\{1,2,3\}$

Select between three different algorithms to partially replicate logic in the "input to feedback" section of the design. Option 1 uses a depth-first search starting from the inputs to the feedback section. Option 3 uses a breadth-first search. Option 2 uses a combination of the two.

This option only affects the resulting circuit if only some of the input to feedback section is replicated. If all or none of the input to feedback section is replicated, the three options produce identical results. The difference is in what *order* the logic in this section is added and thus what part of it is replicated if there are not enough resources available to replicate the entire section.

Results may differ between the three addition types depending on the input design. It is yet not clear if one method is superior to the others in general. Valid options are `1`, `2`, and `3`. Default: `3` (breadth-first search).

`--output_addition_type` $\{1,2,3\}$

Similar to `--input_addition_type`, this option applies to the logic in the "feedback output" section, that is, logic that is driven by the feedback section.

This option only affects the resulting circuit if only some of the feedback output section is replicated. It has no effect if all or none of the feedback output section is replicated. As with `--input_addition_type`, it is yet not clear if one method is superior to the others in general. Valid options are `1`, `2`, and `3`. Default: `3` (breadth-first search).

A.6.5 Merge Factor and Optimization Factor

The following factors are used by the utilization tracker, which estimates the anticipated usage of the target chip after performing (partial) replication. All factors in this section have the precision of a Java `double`.

`--merge_factor` $\{0 \leq n \leq 1\}$

Used to fine-tune the estimation of logic resources in the target chip. Each technology has an internal, default "merge factor" which estimates the percentage of LUTs and flip-flops that will

share the same slice. As this factor is both technology and design dependent, this option allows the user to specify his/her own merge factor.

The total number of logic blocks (without taking into account optimization) is given by the following equation:

$$\text{total logic blocks} = FFs + LUTs - (mergeFactor * FFs).$$

If you need to calculate a custom mergeFactor for a specific design, use the following equation:

$$mergeFactor = \frac{(FFs + LUTs - 2 * slices)}{FFs}.$$

Must be between 0 and 1, inclusive. Default: 0.5.

`--optimization_factor` $\{0 \leq n \leq 1\}$

The "optimization factor" is used to scale down the estimate of LUTs and flip-flops used to account for logic optimization performed during mapping. For example, an optimization factor of 0.90 would assume that logic optimization techniques would reduce the required number of LUTs and FFs by 10%.

We define the optimization factor to be the number of logic blocks after optimization divided by the number of logic blocks before optimization. So the final equation for the total number of logic blocks is as follows:

$$\text{Estimate} = optimization\_factor * (FFs + LUTs - mergeFactor * FFs),$$

where Estimate must be between 0 and 1, inclusive. The default is 0.95.

`--factor_type {ASUF,UEF,DUF}`

Specify the Utilization Factor Type to be used. Valid Factor Types are:

- ASUF

  Available Space Utilization Factor: The maximum utilization of the target part, expressed as a percentage of the unused space on the part after the original (unreplicated) design has been considered.

- UEF

  Utilization Expansion Factor: The maximum increase in utilization of the target part, expressed as a percentage of the utilization of the original (unreplicated) design.

- DUF

  Desired Utilization Factor: The maximum percentage of the target chip to be utilized after performing Partial replication.

  Not case sensitive.

`--factor_value`

Specify a single Factor Value. The number has the precision of a Java `double` and is interpreted based on the Factor Type as explained above.

For example, if a design occupies 30% of the target part prior to replication, a DUF of 0.50 would use 50% of the part. An UEF of 0.50 would increase the usage by 50%, resulting in 45% usage of the part. An ASUF of 0.50 would use 50% of the available space prior to replication, resulting in 65% usage. Must be greater than or equal to 0. Default: 1.0.

`--ignore_hard_resource_utilization_limits`

This option causes all hard resource utilization limits to be ignored when determining how much of the design to replicate.

`--ignore_soft_logic_utilization_limit`

This option causes logic block utilization to be ignored when determining how much of the design to replicate. Hard resources such as BRAMs and CLKDLLs will still be tracked.

A.6.6 Target Part Options

`--part <partname>`

Target architecture for the design. Used to take into account part-specific properties, including the number of resources available in each part. Valid parts include all parts from the *Virtex* and *Virtex2* product lines, represented as a concatenation of the part name and package type. For example, the "Xilinx Virtex 1000 FG680" is represented as `XCV1000FG680`. This argument is *not* case-sensitive. The default is `xcv1000fg680`.

A.6.7 Configuration File Options

The BL-TMR tools can use configuration files in place of command-line parameters. If a parameter is specified in a configuration file, it will be passed to the BL-TMR tool, unless it is overridden by the same argument on the command-line.

`--use_config <config_file>`

Specify a configuration file from which to read parameters.

`--write_config <config_file>`

Write the current set of command-line parameters to a configuration file and exit. The parameters will be parsed to ensure they are valid, but the BL-TMR tool will not run. Note that only the parameters on the command-line are stored in the configuration file. When using `--write_config`, any use of `--use_config` is ignored. This is to prevent complicated cascades of configuration files combined with command-line options.

Examples:

- `--write_config JonSmith.conf` will write the command-line parameters to the file `JonSmith.conf` in the current directory.
- `--write_config /usr/lib/BL-TMR/common.conf` will write the command-line parameters to the file `/usr/share/BL-TMR/common.conf`.
- See section A.10.11, "Using Configuration Files," for more information.

A.6.8 Logging options

`--log <logfile>`

Specifies an alternate file for logging output.

`--debug[:<debug_log>]`

Specifies a file for logging the debugging output.If no file specified, debug output is printed to the log file.

`(-V|--verbose) <{1|2|3|4|5}>`

Sets the verbosity level: 1 prints only errors, 2 warnings, 3 normal, 4 log to stdout. 5 prints debug information. (default: 3)

`--append_log`

Append to the logfile instead of replacing it.

A.7 JEdifVoterSelection

JEdifVoterSelection determines the locations where voters will be inserted into a triplicated design (or triplicated portions of a design). Voter locations are determined using a feedback cutset algorithm and rules for voting where downscaling is necessary. The results are added into the replication description file (.rdesc).

At times, the user may wish to force voter insertion on certain nets and disable voter insertion on others. This can be accomplished by inserting ‘force_restore’ and ‘do_not_restore’ properties on selected nets in the .edf file as follows:

```
(property force_restore (boolean (true)))
(property do_not_restore (boolean (true)))
```

```
>java edu.byu.ece.edif.jedif.JEdifVoterSelection
Options:
  [-h|--help]
  [-v|--version]

  <input_file>
  (-r|--rep_desc) <rep_desc>
  (-c|--c_desc) <c_desc>

  [--after_ff_cutset]
  [--before_ff_cutset]
  [--connectivity_cutset]
  [--basic_decomposition]
  [--highest_fanout_cutset]
  [--highest_ff_fanout_cutset]
  [--highest_ff_fanin_input_cutset]
  [--highest_ff_fanin_output_cutset]

  [--write_config <config_file>]
  [--use_config <config_file>]

  [--log <logfile>]
  [--debug[:<debug_log>]]
  [(-V|--verbose) <{1|2|3|4|5}>]
  [--append_log]
```

A.7.1 File Options

`<input_file>`

Filename and path to the .jedif source file.

`(-r|--rep_desc) <rep_desc>`

Filename and path to the replication description (.rdesc) file to be modified.

`(-c|--c_desc) <c_desc>`

Filename and path to the circuit description (.cdesc) file generated by JEdifAnalyze.

A.7.2 Cutset Algorithms

Synchronization voters are essential in FPGA circuits that use TMR because they ensure that the internal state of all three TMR replicates are synchronized after configuration scrubbing. Adding synchronization voters in a design manually, however, is a difficult and error prone process. This tool uses automated cutset algorithms for selecting synchronization voter locations and inserting them in the design.

Synchronization voter insertion algorithms must determine a set of nets within a design that cuts *all* feedback in the design. Voters are placed on each of these nets to ensure that synchronization voting occurs within the feedback structures of a design. Determining a set of voter locations that satisfy this constraint is an instance of the feedback edge set (FES) problem. The algorithms used in this tool solve the FES problem for voter insertion in a way that avoids illegal cut locations. In addition, many of the algorithms employ heuristics based on FPGA architecture that attempt to minimize circuit area or timing impact.

`--before_ff_cutset`

This option selects the *Voters Before Every Flip-Flop* algorithm.

`--after_ff_cutset`

This option selects the *Voters After Every Flip-Flop* algorithm.

`--connectivity_cutset`

This option selects an algorithm that is the precursor to the *Basic SCC Decomposition Algorithm*. It is the original algorithm that removes arbitray feedback edges until all feedback is cut. This option has been shown to produce inferior results in general to the others but in some

few cases it *may* give better timing results (based on empirical data, this is not likely in real-world designs).

`--basic_decomposition`

This option selects the *Basic SCC Decomposition* algorithm.

`--highest_fanout_cutset`

This option selects the *Highest Fanout SCC Decomposition* algorithm.

`--highest_ff_fanout_cutset`

The option selects the *Highest Flip-Flop Fanout SCC Decomposition* algorithm.

`--highest_ff_fanin_input_cutset`

This option selects the *Highest Fan-in Flip-Flop Input* algorithm.

`--highest_ff_fanin_output_cutset`

This option selects the *Highest Fan-in Flip-Flop Output* algorithm.

A.7.3 Configuration File Options

The BL-TMR tools can use configuration files in place of command-line parameters. If a parameter is specified in a configuration file, it will be passed to the BL-TMR tool, unless it is overridden by the same argument on the command-line.

`--use_config <config_file>`

Specify a configuration file from which to read parameters.

`--write_config <config_file>`

Write the current set of command-line parameters to a configuration file and exit. The parameters will be parsed to ensure they are valid, but the BL-TMR tool will not run. Note that only the parameters on the command-line are stored in the configuration file. When using `--write_config`, any use of `--use_config` is ignored. This is to prevent complicated cascades of configuration files combined with command-line options.

Examples:

- `--write_config JonSmith.conf` will write the command-line parameters to the file `JonSmith.conf` in the current directory.
- `--write_config /usr/lib/BL-TMR/common.conf` will write the command-line parameters to the file `/usr/share/BL-TMR/common.conf`.
- See section A.10.11, "Using Configuration Files," for more information.

A.7.4 Logging options

`--log <logfile>`

Specifies an alternate file for logging output.

`--debug[:<debug_log>]`

Specifies a file for logging the debugging output.If no file specified, debug output is printed to the log file.

`(-V|--verbose) <{1|2|3|4|5}>`

Sets the verbosity level: 1 prints only errors, 2 warnings, 3 normal, 4 log to stdout. 5 prints debug information. (default: 3)

`--append_log`

Append to the logfile instead of replacing it.

A.8 JEdifNMR

JEdifNMR performs the replication selected by previously run tools. Information about what to replicate and where to insert voters/detectors is obtained from the replication description (.rdesc) file created by the previous steps.

```
> java edu.byu.ece.edif.jedif.JEdifNMR
Options:
  [-h|--help]
  [-v|--version]

  <input_file>
  (-r|--rep_desc) <rep_desc>
  [(-o|--output) <output_file>]
  [--edif]
  [--rename_top_cell <new_name>]

  [(-p|--part) <part>]

  [--write_config <config_file>]
  [--use_config <config_file>]

  [--log <logfile>]
  [--debug[:<debug_log>]]
  [(-V|--verbose) <{1|2|3|4|5}>]
  [--append_log]
```

A.8.1 File Options

`<input_file>`

Filename and path to the .jedif source file.

`(-r|--rep_desc) <rep_desc>`

Filename and path to the replication description (.rdesc) file containing the replication information.

`(-o|--output) <output_file>`

Filename and path to the output file. If the given filename ends in .edf or if the `--edif` option is specified, an EDIF file will be generated. Otherwise, the replicated circuit will be output in .jedif format.

`--edif`

Specifies that an EDIF (.edf) file should be generated instead of a .jedif file.

`--rename_top_cell <new_name>`

Use this option to specify a new name for the design's top cell.

A.8.2 Target Part Options

`--part <partname>`

Target architecture for the design. Used to take into account part-specific properties, including the number of resources available in each part. Valid parts include all parts from the *Virtex* and *Virtex2* product lines, represented as a concatenation of the part name and package type. For example, the "Xilinx Virtex 1000 FG680" is represented as `XCV1000FG680`. This argument is *not* case-sensitive. The default is `xcv1000fg680`.

A.8.3 Configuration File Options

The BL-TMR tools can use configuration files in place of command-line parameters. If a parameter is specified in a configuration file, it will be passed to the BL-TMR tool, unless it is overridden by the same argument on the command-line.

`--use_config <config_file>`

Specify a configuration file from which to read parameters.

`--write_config <config_file>`

Write the current set of command-line parameters to a configuration file and exit. The parameters will be parsed to ensure they are valid, but the BL-TMR tool will not run. Note that only the parameters on the command-line are stored in the configuration file. When using `--write_config`, any use of `--use_config` is ignored. This is to prevent complicated cascades of configuration files combined with command-line options.

Examples:

- `--write_config JonSmith.conf` will write the command-line parameters to the file `JonSmith.conf` in the current directory.
- `--write_config /usr/lib/BL-TMR/common.conf` will write the command-line parameters to the file `/usr/share/BL-TMR/common.conf`.
- See section A.10.11, "Using Configuration Files," for more information.

A.8.4 Logging options

`--log <logfile>`

Specifies an alternate file for logging output.

`--debug[:<debug_log>]`

Specifies a file for logging the debugging output.If no file specified, debug output is printed to the log file.

`(-V|--verbose) <{1|2|3|4|5}>`

Sets the verbosity level: 1 prints only errors, 2 warnings, 3 normal, 4 log to stdout. 5 prints debug information. (default: 3)

`--append_log`

Append to the logfile instead of replacing it.

A.9 JEdifReplicationQuery

JEdifReplicationQuery is used to query the contents of a replication description (.rdesc) file and to provide information about the type(s) of replication that will be applied to a design given the information in the file.

The tool gives information about each of the replication types (i.e. triplication, duplication) used in the design. The ports and instances selected for each type are displayed.

The tool also gives information about organs (i.e. voters, comparators) that will be inserted into the design on each net. An organ summary is provided that lists the total number of each kind of organ to be inserted.

Finally, the tool lists any detection outputs to be used as well as information about whether an output register (and which clock net) and output buffer will be used. A list of nets that will be detected on is given for each detection output.

```
>java edu.byu.ece.edif.jedif.JEdifReplicationQuery
Options:
  [-h|--help]
  [-v|--version]

  <input_file>
  (-r|--rep_desc) <rep_desc>
```

A.9.1 File Options:

`<input_file>`

Filename and path to the .jedif source file containing the EDIF environment to be queried.

`(-r|--rep_desc) <rep_desc>`

Filename and path to the replication description (.rdesc) file containing the replication information.

A.10 Common Usage of JEdifNMRSelection

This section describes a few sample scenarios and explains which combination of command line options should be used for each.

A.10.1 Full TMR

This example shows how to perform "full" TMR (triplication of all components) on a design. For larger designs, this may result in a TMR'd version of the design that does not fit in the desired chip. If this is the case, some form of "partial" TMR should be used.

In this example, the design to be triplicated is `myDesign.edf`. Both input ports and output ports are triplicated. The part used in this case is the Virtex II XC2V1000-FG456.

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myDesign.jedif \
  -c myDesign.cdesc -r myDesign.rdesc --replication_type triplication \
  --nmr_inports --nmr_outports --full_nmr --part xc2v1000fg456
```

A.10.2 Full TMR—Clock not triplicated

Some systems are not ideal for triplicated clock lines because of resource constraints. This example shows how to triplicate everything but the clock line.

This example is almost identical to the example in section A.10.1. The `--no_nmr_p` option specifies that the top-level port named `Clk` (case-sensitive) should not be triplicated. The `--no_nmr_c` option indicates that all cells of the global clock buffer type `BUFG` should also not be triplicated. This prevents the clock line after the buffer from being triplicated and the entire circuit will use the same single clock.

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myDesign.jedif \
  -c myDesign.cdesc -r myDesign.rdesc --replication_type triplication \
  --nmr_inports --nmr_outports --full_nmr --no_nmr_p Clk --no_nmr_c BUFG \
  --part xc2v1000fg456
```

A.10.3 Full TMR—No I/O triplication

Many FPGA applications are port-limited. This example shows how to prevent all inputs and outputs from being triplicated. In this example, the user must leave out the `--nmr_inports` and

`--nmr_outports` parameters so that top level ports are not included in triplication. This example also leaves out the `--part` option, using the default value (a value saved as a property in the EDIF file).

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myDesign.jedif -c myDesign.cdesc \
  -r one_counter.rdesc --replication_type triplication --full_nmr
```

A.10.4 Partial TMR—No I/O triplication

This example shows a standard usage of the BL-TMR tool for partial TMR. In this case, the design is too large to fit in the targeted device when fully triplicated. The BL-TMR tool will triplicate as much logic as possible and estimate when the target chip will be fully utilized.

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myLargeDesign.jedif \
  -c myLargeDesign.cdesc -r myLargeDesign.rdesc --replication_type triplication
```

A.10.5 Partial TMR—SCC Decomposition, custom estimation factors

In this case, the user wishes to include parts of strongly-connected components (SCCs) for triplication. This example also shows how to override the default merge and optimization factors.

```
> java java edu.byu.ece.edif.jedif.JEdifNMRSelection myLargeDesign.jedif \
  -c myLargeDesign.cdesc -r myLargeDesign.rdesc --replication_type triplication \
  --do_scc_decomposition --merge_factor 0.4 --optimization_factor 0.85
```

A.10.6 Partial TMR—Fill 50% of target device

In some cases, the user may wish to use the triplicated design on the same chip as another design. In this example the user knows that a separate design will require half of the target chip. To fill as much of the left-over 50% as possible, the user specifies a `--factor_type` of `DUF` and a `factor_value` of `0.5`. This will stop triplication of the input design when half of the target chip is utilized, according to the estimate made with the merge optimization factors.

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myLargeDesign.jedif \
  -c myLargeDesign.cdesc -r myLargeDesign.rdesc --replication_type triplication \
  --factor_type DUF --factor_value 0.5
```

A.10.7 Partial TMR—Push utilization past 100%

Due to the way mapping tools are implemented, the user may be able to fit more logic onto the target chip than estimated by the utilization tracker. The Xilinx `map` program, for example, does not map unrelated logic into the same slice until slice utilization reaches 99%. This means that much more logic can be added after this point, though the place and route step will become increasingly more difficult for the vendor tools to perform.

With this in mind, to achieve the maximum capacity on the target chip, it may be necessary to specify a desired utilization factor greater than 1.0 (more than 100% estimated utilization). The following example uses a device utilization factor of 1.5, which will stop triplication when an estimated 150% of the target part is utilized.

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myLargeDesign.jedif \
  -c myLargeDesign.cdesc -r myLargeDesign.rdesc --replication_type triplication \
  --factor_type DUF --factor_value 1.5
```

A.10.8 Partial TMR—Use 75% of available space on target device

The available space utilization factor can be used to specify the amount of space on the target device left after the unmitigated circuit is mapped. To fill the chip up to 75% of the left-over space, the user specifies a `--factor_type` of `ASUF` and a `factor_value` of `0.75`. If the original design size is estimated at using 40% of the target chip, this will stop triplication when 70% $(40 + (100 - 40) * 0.75)$ of the target chip is utilized.

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myLargeDesign.jedif \
  -c myLargeDesign.cdesc -r myLargeDesign.rdesc --replication_type triplication \
  --factor_type ASUF --factor_value 0.75
```

A.10.9 Mixed TMR/DWC—TMR Persistent Section, Duplicate the Rest

At times, it may be impossible to fit an entire triplicated design on a part. It may still be possible to triplicate feedback sections and duplicate the rest of the design. This can be accomplished by running JEdifNMRSelection twice. The first run selects feedback sections for TMR and the second run selects the rest of the design for duplication. Notice that the second run uses the `--continue` option to indicate that the replication description (.rdesc) file should be modified

rather than overwritten. The second run also uses the `--full_nmr` option to select the rest of the design. This does not override the TMR sections selected by the first run because the `--override` option is not used.

```
> java -Xmx1G edu.byu.ece.edif.jedif.JEdifNMRSelection myDesign.jedif \
  -c myDesign.cdesc -r myDesign.rdesc --part xcv1000-5-bg560 \
  --replication_type triplication --no_nmr_feedback_output --no_nmr_feed_forward
> java -Xmx1G edu.byu.ece.edif.jedif.JEdifNMRSelection myDesign.jedif \
  -c myDesign.cdesc -r myDesign.rdesc --part xcv1000-5-bg560 \
  --replication_type duplication --continue --full_nmr
```

A.10.10 Triplicate Specific Instances Only

By default, the BL-TMR tool tries to use partial replication to fill up a device when no other options are specified. In order to suppress this behavior and only replicate instances specified by the user, the `--no_partial_nmr` option can be used. (Half-latch safe constants will still be marked for replication in designs where half-latch removal is used, but this behavior can be suppressed by using the `--hl_no_tag_constant` option in JEdifBuild). Specific instances should be selected for replication using the `--nmr_i`, `--nmr_c`, `--nmr_clk`, etc. options. The following example triplicates only two instances (plus a half-latch safe constant cell if half-latch removal was used):

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myDesign.jedif \
  -c myDesign.cdesc -r myDesign.rdesc --replication_type triplication \
  --no_partial_nmr \
  --nmr_i synth_th1/virtexmultaccelerator_mini__1/a/multCol__13/reg_bottom_ysum/fdce,\
  synth_th1/virtexmultaccelerator_mini__1/b/multCol__1/reg_top_ysum/fdce__5 \
  --part xcv1000-5-bg560
```

A.10.11 Using Configuration Files

Configuration files can greatly simplify the use of any of the BL-TMR tools. The following examples show how to create and use configuration files.

Create a Configuration File

The following will write the current command-line arguments to the file `myConfig.conf`:

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myDesign.jedif -c myDesign.cdesc \
  -r myDesign.rdesc --replication_type triplication --nmr_inports --nmr_outports \
  --nmr_c DLL,fdc,clk_buf --nmr_i clk,fifo_output --no_nmr_p clk_port,data_in \
  --no_nmr_c bufg --no_nmr_i mux2,and24 --no_nmr_feed_forward --input_addition_type 1 \
  --output_addition_type 2 --merge_factor 0.85 --optimization_factor 0.90 \
  --part XCV1000-5-BG560 --log myLogFile.log --write_config myConfig.conf
```

The previous command creates the following output, stored in `myConfig.conf`:

```
#myConfig.conf, created by JEdifNMRSelection
#Tue May 12 14:44:30 MDT 2009
no_nmr_i=mux2,and24
write_config=myConfig.conf
merge_factor=0.85
no_nmr_c=bufg
nmr_i=clk,fifo_output
no_nmr_feed_forward=true
replication_type=triplication
log=myLogFile.log
nmr_inports=true
nmr_c=DLL,fdc,clk_buf
input=myDesign.jedif
c_desc=myDesign.cdesc
optimization_factor=0.9
rep_desc=myDesign.rdesc
part=XCV1000-5-BG560
input_addition_type=1
nmr_outports=true
no_nmr_p=clk_port,data_in
output_addition_type=2
```

Configuration files are defined by the `java.util.Properties` class. However, the format is simple enough that configuration files can easily be created by hand or by other programs. As seen above, the format is simply `key=value`. A hash mark (pound sign) (`#`) at the beginning of a line marks that line as a comment. BL-TMR options are given just as they would be on the command-line, with the exception that command-line options with no arguments (e.g. `--nmr_inports` and `--do_scc_decomposition`, etc.) are specified as either `true` or `false`, as seen above.

Use a Configuration File

The following example shows how to load `myConfig.conf` as a configuration file:

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection --use_config myConfig.conf
```

Combining Configuration Files and Command-line Arguments

Configuration files provide default values of BL-TMR options. Any options specified on the command-line will take precedence. (See section A.8.3, "`--use_config`" for detailed precedence information.) The following example uses the same options specified by myConfig.conf, but changes the input and output files:

```
> java edu.byu.ece.edif.jedif.JEdifNMRSelection myOtherDesign.jedif \
  -c myOtherDesign.cdesc -r myOtherDesign.rdesc --useconfig myConfig.conf
```

A.11 Sample Makefile for TMR

```
DESIGN=my_design
BUILD = edu.byu.ece.edif.jedif.JEdifBuild
ANALYZE = edu.byu.ece.edif.jedif.JEdifAnalyze
NMR_SELECTION = edu.byu.ece.edif.jedif.JEdifNMRSelection
VOTER_SELECTION = edu.byu.ece.edif.jedif.JEdifVoterSelection
MFV = edu.byu.ece.edif.jedif.JEdifMoreFrequentVoting
DETECTION_SELECTION = edu.byu.ece.edif.jedif.JEdifDetectionSelection
PERSISTENCE_DETECTION = edu.byu.ece.edif.jedif.JEdifPersistenceDetection
NMR = edu.byu.ece.edif.jedif.JEdifNMR
JVM_OPTS = -Xmx1G
PART = xcv1000-5-bg560

BUILD_OPTS = --remove_hl --replace_luts --remove_fmaps --remove_rlocs
ANALYZE_OPTS = --part $(PART)
NMR_SELECTION_OPTS = --part $(PART) --replication_type triplication --full_nmr --nmr_inports \
        --nmr_outports
VOTER_OPTS = --highest_ff_fanout_cutset
NMR_OPTS = --part $(PART)

all: $(DESIGN)_nmr.edf

$(DESIGN)_nmr.edf: $(DESIGN).jedif voter_selection.touch
        java $(JVM_OPTS) $(NMR) $(DESIGN).jedif --rep_desc $(DESIGN).rdesc $(NMR_OPTS) -o \
                $(DESIGN)_nmr.edf
```

```
voter_selection.touch: $(DESIGN).jedif nmr_selection.touch
        java $(JVM_OPTS) $(VOTER_SELECTION) $(DESIGN).jedif --rep_desc $(DESIGN).rdesc \
                --c_desc $(DESIGN).cdesc $(VOTER_OPTS)
        touch voter_selection.touch

nmr_selection.touch: $(DESIGN).cdesc
        java $(JVM_OPTS) $(NMR_SELECTION) $(DESIGN).jedif --c_desc $(DESIGN).cdesc \
                --rep_desc $(DESIGN).rdesc $(NMR_SELECTION_OPTS)
        touch nmr_selection.touch

$(DESIGN).cdesc: $(DESIGN).jedif
        java $(JVM_OPTS) $(ANALYZE) $(DESIGN).jedif -o $(DESIGN).cdesc $(ANALYZE_OPTS)

$(DESIGN).jedif: $(DESIGN).edf
        java $(JVM_OPTS) $(BUILD) $(DESIGN).edf $(BUILD_OPTS)

clean:
        rm -rf $(DESIGN).jedif $(DESIGN).cdesc $(DESIGN).rdesc $(DESIGN)_nmr.edf *.log *.touch
```

A.12 Special Notes

A.12.1 Naming Conventions

The BL-TMR tool alters the names of replicated signals, cell instances, and ports. Be aware of this when using placement (or other) constraints. An output port named `myOutport` in the original EDIF file, when triplicated, would become `myOutport_TMR_0`, `myOutport_TMR_1`, and `myOutport_TMR_2`. Similarly, a flip-flop whose instance name is `myFF` in the original file, when triplicated, would become `myFF_TMR_0`, `myFF_TMR_1`, and `myFF_TMR_2`. Net names follow the same convention.

A.12.2 Allocating More Memory for the JVM

Larger designs may require more heap memory than the Java Virtual Machine (JVM) is allocated by default. Use the `-Xmx` [1] option with the Java executable to change the maximum

[1] See `http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/java.html#Xms` for more information about this and other command-line options to the JVM.

amount of memory for the virtual machine. The following example allocates up to 256 MB of heap space for the JVM:

```
> java -Xmx256M edu.byu.ece.edif.jedif.JEdifBuild ...
```