Aaron J Arthurs
Digital Circuit Testing
March 18, 2004
Class Project

Mitigating SEUs With State Redundancy

Abstract:

State redundancy is presented in conjunction with using a software package to demonstrate how the concept can be used to make a bistable device more immune to SEUs.

Introduction:

Digital electronics are moving towards the sub-micron level and higher speeds. The need for reliable circuits remains, and due to the trend that modern electronics follow, more complex designs are required. With the popularity of VLSI designs, fault tolerance and avoidance takes higher priority over logic reduction.

One fault of interest is a ‘Single Event Upset’ (SEU). SEUs are unwanted bit-flips in memory (in combinational logic, unwanted logic pulses) caused by an accumulation of charged particles. In short, SEUs are transient, complementing faults. As transistors continue to reduce in size, devices become more prone to SEUs. When detecting for an SEU, an SRAM-based FPGA is ideal because "a transient effect is followed by a permanent effect" [3].

"FPGAs based on SRAM technology are most suitable for adaptive computing applications" (i.e. FPGAs are configurable to adapt to any digital circuit given the limits of size and complexity). "The computing requirements of spacecraft may be particularly suitable for adaptive computing architectures, but the space environment" contains radiation; this radiation may cause SEUs to occur in FPGAs. “Leading FPGA manufacturers are now producing radiation hardened SRAM FPGAs for space and military applications." Xilinx, for example, uses two measurements for radiation effects (e.g. for a XQR4000XL): 'Total Ionizing Dose' (TID) and 'Single Event Effects' (SEE). TID is "a measure of the maximum dose of ionising radiation an electronic device can withstand", which can be observed through the supply current; SEE involves a charged particle introduced in a digital circuit. There are several classes of SEEs (discussed in ‘Background of SEU Mitigation Techniques’) [1].

SRAM-based FPGAs are structured with 'Configurable Logic Blocks' (CLB) that are interconnected through a 'General Routing Matrix' (GRM). SEU-sensitive configuration memory cells include, but not limited to, 'Look-Up Table' (LUT) cells, multiplexor select lines, and interconnects. The LUTs in the CLBs can be configured to implement either fixed logic functions or RAM modules in user applications; "bistables such as flip-flops and latches, however, are excluded from memory modules implemented in LUTs because of area efficiency" [2]. 'Connection Boxes' (CB) and 'Switch Boxes' (SB) are used to connect CLBs together. CBs act as local bridges between CLBs and adjacent wiring channels while SBs are controlled by SRAM cells to connect horizontal and vertical wiring channels.

In later sections, the bistable devices included in SRAM-based FPGAs, particularly flip-flops, are used in conjunction with combinational logic to create a more SEU-immune bistable element. The advantage to implementing such a circuit in an FPGA is the availability of the needed logic (especially for VLSI design) and the high-level design specification that comes with an FPGA

CAD tool. Two designs are presented; one is based on static state redundancy and the other on dynamic state redundancy. Each design has their advantages and disadvantages based on reliability in the event of an SEU as well as a 'Multiple Bit Upset' (MBU). Also, in lieu of dynamic state redundancy, the idea of 'biasing' redundant states in order to handle an MBU is discussed in detail.

Background of Transient Fault Mitigation Techniques:

One mitigation technique proposed by David Brodrick, et. al., uses 'Error Detection And Correction' (EDAC) in FPGAs. Its measurement is based on SEEs, which is further broken up into the following.

A 'Single Event Latchup' (SEL) is specific to CMOS in that certain transistors are activated by radiation; the only way to de-activate these transistors is to go through a new power cycle. Because Xilinx devices are manufactured on an epitaxial layer (a growth on a crystalline substrate of a crystalline substance that mimics the orientation of the substrate), SELs do not occur. A 'Single Event Transient' (SET) is caused by a transient pulse on a data line (e.g. a flip-flop is placed in a metastable condition due to a transient fault on the clock line), which becomes "more important as feature size shrinks and operating frequency increases." An SEU is a state change in a memory device; in FPGAs, SEUs on configuration latches and user data latches need to be distinguished. A 'Single Event Functional Interrupt' (SEFI) is an interruption in normal operation (this is a special class of other SEEs).

The techniques used in the EDAC system is self-checking and readback. The self-checking technique consists of duplicating a circuit and look for consistent outputs, separating a circuit into several stages, and exploiting an internal property in a circuit. Depending on the circuit, the method used to self-check will vary. The readback technique involves feeding a bitstream into a circuit and reading the same bitstream back for comparison.

Another mitigation technique proposed by Wei-Je Huang, et. al., covers the use of LUTs as user memory devices along with the "memory coherence problem for the online transient error recovery of FPGA configuration"; this technique is known as the 'dirty bit' protocol. The protocol is based on the fact that an FPGA can be partitioned vertically or horizontally into configuration data frames, the smallest amount of data that can be accessed with a single configuration command. In short, an FPGA can be partially and concurrently reconfigured without stopping the FPGA's operations.

System transients, which are upsets on user data, can be fixed with traditional transient error recovery techniques such as roll-forward and rollback. For configuration transients, which are upsets on configuration data such as the control of SBs, readback and writeback mechanisms are used. While the readback mechanism looks for errors in the configuration data, the writeback mechanism loads the correct configuration data back in, frame-by-frame. The memory coherence problem occurs for user data (i.e. cells in LUTs used as memory modules). Before readback, a row in a LUT that stores user data may contain initial data. During readback, that initial data is saved elsewhere for the writeback operation. When writeback executes, the initial data is stored back into the row in the LUT. The memory coherence problem comes in during the time between

readback and writeback when the user data changes in the row in the LUT; the writeback operation will overwrite the new data with the initial data. Huang, et. al., suggests that stalling (disabling writing to the row in the LUT) during the time between readback and writeback is a conservative solution compared to having separate frames for configuration data and user data. The reason is that having separated frames poses a routing problem for an FPGA.

To reduce stalling even further, Huang, et. al., presents a memory coherence technique that uses a 'dirty' bit for each row in a LUT along with a 'readback-again' strategy. The idea is to stall whenever a user write operation occurs after the first readback and in presence of a user data fault; the 'dirty' bit is set to indicate that a second readback is needed if an error is detected. (The reader is referred to Figure 6 in [2], which shows a state diagram of the dirty-bit memory coherence technique.) As for implementation of this technique, Huang, et. al., uses a dual-FPGA error recovery scheme where one FGPA is the ‘device under check’ and the other FPGA is the ‘checking device;’ the checking device is essentially a finite state machine.

Upon mathematical analysis and simulation, the conclusion is that the dirty-bit memory coherence technique greatly reduces stalling compared to the 'stall-when-write' strategy.

Finally, Fernanda Lima, et. al., presents a mitigation method known as 'Double Modular Redundancy With Comparison’ (DMWC). DMWC is a combination of ‘Triple Modular Redundancy’ (TMR) and time redundancy.

The TMR structure, which is based on a three-way majority voter system handles the effect of a transient effect followed by a permanent effect. Because of its hardware redundancy, TMR is ideal for FGPAs; also, "the use of TMR in Virtex FPGAs has confirmed the efficacy of the TMR structure combined with scrubbing to recover upsets in the FPGA architecture." However, the TMR has a disadvantage in area overhead, additional input and output pins, and increase in power consumption. The advantage to the TMR structure is that it only applies to the high level design description.

TMR is typically used with bit scrubbing (continuous loading of configuration bits into an FPGA); this is assure error recovery without interruption of system operation. While bit "scrubbing allows a system to repair SEUs in the configuration memory without disrupting its operations," the TMR repairs upsets occurring in the user combinational logic by voting out the dysfunctional logic block. That is, bit scrubbing prevents an accumulation of upsets that would make TMR unreliable, and TMR handles the combinational logic as per a user application. Lima, et. al., shows a high level technique that combines TMR with time and hardware redundancy, reducing the number of I/O pads and power dissipation.

Time redundancy can be implemented using an extra flip-flop whose clock signal phase differs from the first flip-flop's clock signal phase; the output of each flip-flop is compared. Hardware redundancy can be implemented using an, e.g. TMR and 'Duplication With Comparison' (DWC), structure. With DMWC, time and hardware redundancy can be combined to avoid using full hardware redundancy and, at the same time, avoid interruption of the system in the presence of the fault. The layout is a double-redundant logic, each with two time redundant latches, resulting in an output from four XOR gates (time comparison from the first and second combinational

logic and hardware comparison from the first and second clock phase - Ordered Tc0, Hc, Tc1, Hcd). Table 1 in [3] shows the possible 'fault syndrome' combinations along with the output from the four latches.

Using the presented scheme, a double-redundant circuit is placed in a state where the first redundant block (RB0) is used and monitored for upsets while the second redundant block (RB1) is used as a spare; bit scrubbing is applied to the spare. When an upset is detected in the RB0, the circuit switches to RB1; RB1 is now the main block and RB0 is the spare until an upset is detected in RB1. Partial TMR is used as the last stage in the circuit. Two modifications to the circuit, as in general, can be made. One is to use an upset detector and voter circuit for a group of bits. Another is to implement an upset detector and voter circuit using a single state machine for a group a bits.

In the following sections, another technique called state redundancy is proposed and applied to a bit of information. This technique is, in a way, a combination of time and hardware redundancy as well as self-checking. The only exception that it takes a higher level approach to circuit design.

State Redundancy - Concept:

Normally, a sequential circuit stores a finite number of states, each of which are represented by a certain value stored in memory. With state redundancy, each state may be represented by one or more values. For example, consider a finite state machine (FSM) that has five states; the value of each state is shown in Figure 1a. Because there are five states, three bits are needed; however, three out of eight possible values are not used. State redundancy can be implemented by assigning the remaining three values to three of the states as pictured in Figure 1b.

In the last example, no extra bit is needed, but there are cases where adding one or more extra bits is preferred or required in order to integrate state redundancy. If a sequential circuit implements a power of two states, for instance, then at least one extra bit is required. Even if a sequential circuit does not use a power of two states, an extra bit is necessary to ensure state redundancy for every state. Going back to the last example, states A and E are the only two states lacking redundancy. By adding a fourth bit, both states A and E are now redundant, shown in Figure 1c.

State	Value
A	000
B	001
C	010
D	011
E	100

(a)

State	Value(s)
A	000
B	001, 101
C	010, 110
D	011, 111
E	100

(b)

State	Value(s)
A	0000, 1000
B	0001, 0101, 1001, 1101
C	0010, 0110, 1010, 1110
D	0011, 0111, 1011, 1111
E	0100, 1100

(c)

Figure 1: (a) Value of each state using no state redundancy (b) Same states utilizing the unused values, enforcing state redundancy on some states (c) Using an extra bit, redundancy is applied to all states

Besides assigning certain values to each state, certain bits can be allocated to each state. That is, if N number of states are to be implemented, then at least N bits are needed. Figure 2a pictures how the example from Figure 1a can be converted from value-wise to bit-wise state allocation. The advantage to bit-wise state allocation is the possible use of concurrently active states, if the design demands it. The disadvantage is the additional area overhead and, usually, lack of utilization. In light of area overhead, value-wise state allocation is used instead.

State	Bit
A	0
B	1
C	2
D	3
E	4

(a)

State	Bit(s)
A	0, 1
B	2, 3
C	4, 5
D	6, 7
E	8, 9

(b)

State	Bit(s)
A	0, 5
B	1, 6
C	2, 7
D	3, 8
E	4, 9

(c)

Figure 2: (a) Bit-wise state allocation with no redundancy (b) Bit-wise state allocation with double-redundancy (c) Same as [b] except that the each bit is not adjacent with its state-redundant bit

Static and Dynamic State Redundancy:

State redundancy can be applied in two ways: statically and dynamically. With static state redundancy, each state is assigned a fixed set of unique, non-overlapping values; dynamic state redundancy, on the other hand, allows for each state to have overlapping values. Figure 3 illustrates the difference between static and dynamic state redundancy.

State	Value(s)
A	000
B	001, 101
C	010, 110
D	011, 111
E	100

(a)

State	Value(s)
A	000, *101*
B	001, *101*
C	010, 110
D	011, *111*
E	100, *111*

(b)

Figure 3: (a) Static state redundancy (same as Figure 1b) (b) Dynamic state redundancy (overlapping values shown in italics)

When determining the present state using static state redundancy, a combinational function of the bits used can be derived from the state-value table. In contrast, dynamic state redundancy requires memory of the previous state whenever an overlapping value is stored. Implementing memory to store the previous state can be done in two ways. One, store the previous value (assuming non-overlapping) into a second set of bits. The problems with this approach are that more bits are required and the previous value cannot be an overlapping value; if the previous value is overlapping, then the present state cannot be determined among the sharing states. The other approach is to switch back to the previous value whenever an overlapping value is stored. For the latter approach, only the states that share the overlapping value can store that value; furthermore, any states that are allocated any overlapping values must also be allocated at least one non-overlapping value. The non-overlapping value that a state is assigned is the value that is stored back whenever an overlapping value is stored.

In Figure 3b, for example, the approach of switching back to the previous state can be realized by detecting overlapping values along with certain edges. States A and B both share the value of '101' (and states D and E share '111'). When an FSM switches to '101', the critical edge may occur in the right-most bit; that is, if the FSM previously stored state A (000), a positive-edge will occur on the right-most bit when the FSM switches to the value '101'. If the FSM previously stored state B (001), no edge will occur on the right-most bit. Whether or not a positive-edge occurs, the FSM will detect the overlapping value '101' and switch back to the appropriate state (000 or 001).

In practical designs, static state redundancy may be used to help utilize any unused values whenever less than a power of two states are defined; however, it may also be used for self-checking. Dynamic state redundancy is more suitable for error detection and correction where the overlapping values are considered to be 'error values'. When one of these 'error values' are detected, the FSM will be able to switch back to the correct value and state.

Static State Redundancy in a Bit of Information:

Before describing the designs used to create a state redundant bit, the problem of over-correction is explained in detail. First of all, a second bistable is needed to make a state redundant bit. Half of the values are assigned to state '0' and the other half assigned to state '1'; this allows for a total of six possible state-value allocations. The choice of allocation is critical due to the possibility of over-correction.

Value Y_2Y_1	State
00	"0"
01	"1"
10	"1"
11	"0"

(a)

Value Y_2Y_1	State
00	"0"
01	"1"
10	"0"
11	"1"

(b)

Figure 4: (a) Static, XOR state-value relation (b) Present state is dependent on Y_1 only

Figure 4 shows two of the six ways of allocating values to states, both of which are using static state redundancy. The XOR state-value allocation requires for the present state to be dependent on both bits at any point in time. With the XOR state-value allocation, a state will switch to a redundant value in the event of a ‘Multiple Bit Upset’ (MBU). In the event of an SEU, on the other hand, the circuit will have to correct itself by reacting to the SEU as if it was an MBU. Figure 5 clarifies how a circuit may operate using the XOR state-value allocation. As an example, if the circuit stores the value 00 and an MBU occurs, the circuit will now store 11. Because both 00 and 11 represent state ‘0’, the present state of the circuit is not affected by the MBU (the same holds true for state ‘1’). However, if the circuit stores the value 00 and an SEU occurs, the circuit will either store 01 or 10, each of which represents state ‘1’. The solution is then to complement the unaffected bit whenever an SEU occurs; however, this raises a problem of detecting the SEU from the affected bit. Due to the loss of the present state from an SEU, the circuit will have to detect for edges from each bit during a clock cycle.

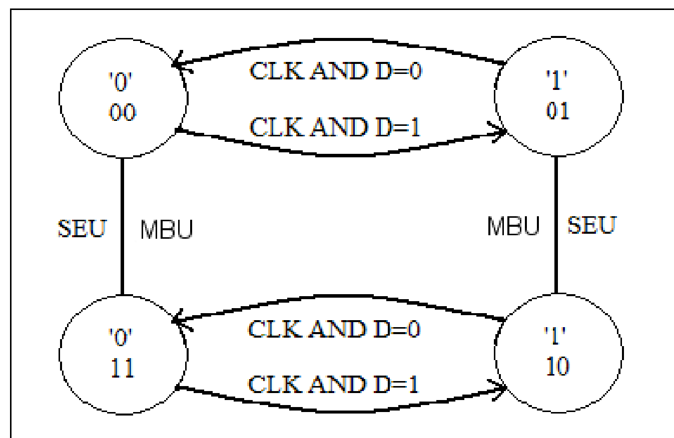


Figure 5: State diagram of the XOR state-value allocation from Figure 4a

Figure 4b, the Y_1 state-value allocation, only requires the present state to be dependent on the least significant bit; the other bit is used in the event of an SEU. Figure 6 depicts the state diagram for the Y_1 state-value allocation during a clock cycle. The states where Y_2 is 0 are considered to be normal states whereas the other states are fallback states in the event of an SEU. In a normal state, when an SEU occurs on Y_2, a positive-edge will be detected on Y_2 and cause Y_2 to flip back to 0. If an SEU occurs on Y_1 and the present value is 00, the positive-edge from Y_1 will change the value from 01 to 10; in turn, the positive-edge from Y_2 will change the value back to 00. If an SEU occurs on Y_1 and the present value is 01, the negative-edge from Y_1 will change the value from 00 to 11; in turn, the positive-edge from Y_2 will change the value back to 01. There are cases where the Y_1 state-value allocation may not correct itself, particularly if an SEU occurs while the circuit is in one the fallback states; however, the circuit will be in one of the normal states most of the time.

There are advantages and disadvantages between the XOR and Y_1 state-value allocations. The XOR state-value allocation can handle an MBU and, ideally, an SEU; therefore, a circuit using this allocation would be immune to memory errors. As it will shown later, a circuit using the XOR state-value allocation will have the tendency to over-correct itself due to the need for SEU detection from each bit. The Y_1 state-value allocation can handle an SEU during a normal state

without allowing for over-correction; however, a circuit implementing this allocation will not be immune to an SEU occurring during a fallback state nor in the event of an MBU.

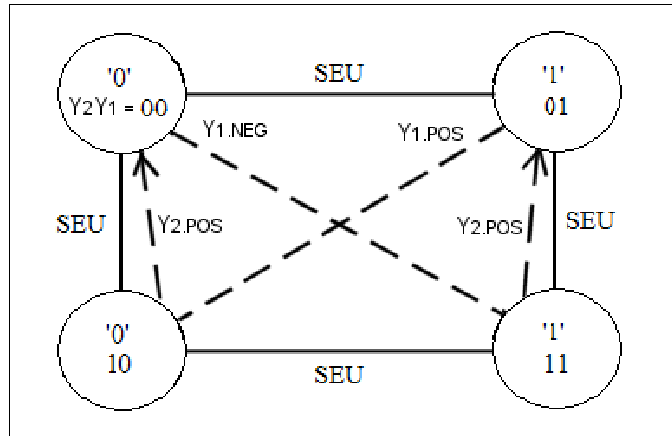
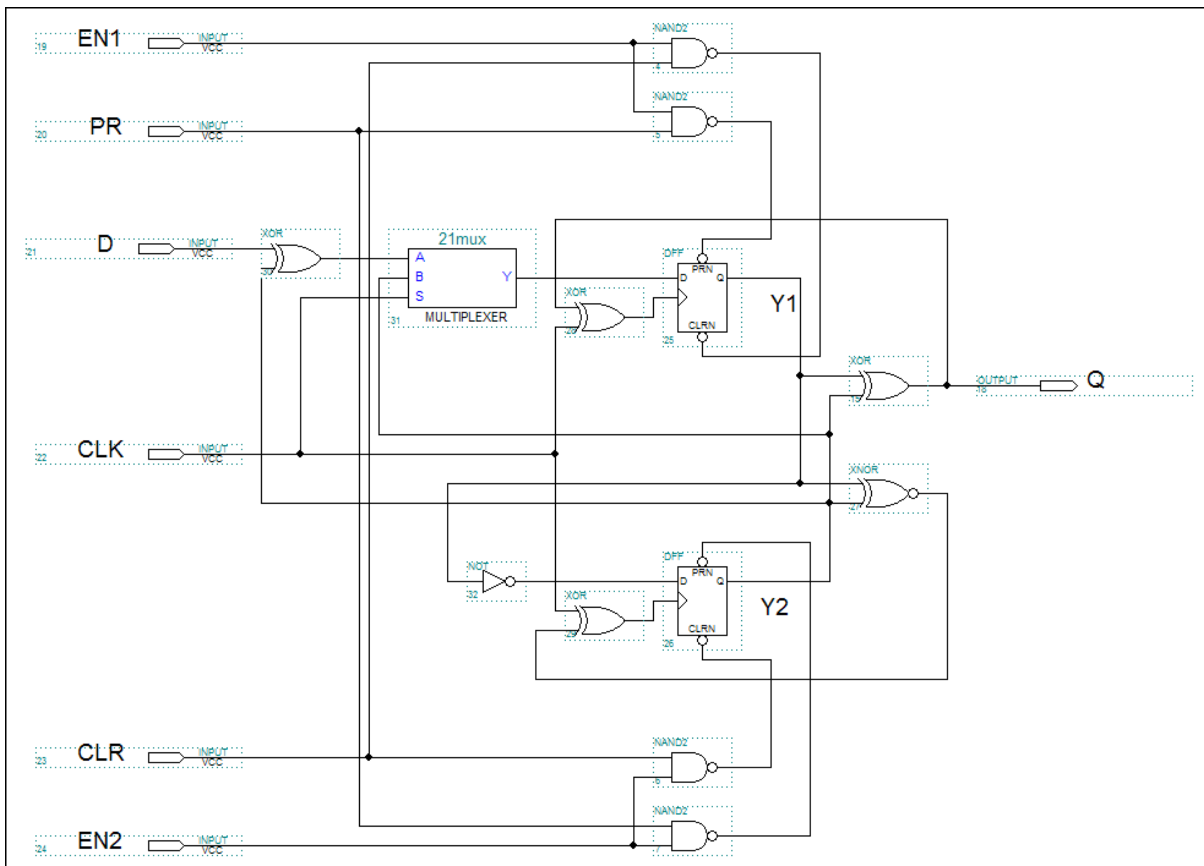


Figure 6: State diagram of the Y_1 state-value allocation from Figure 4b during a clock cycle

The two schematics that follow illustrates the use of each state-value allocation. The software package used to generate the schematics with their timing diagrams is Altera's MAX+PLUS II; the device used is the MAX7000 EPM7032LC44-6 FPGA (the same software and device is used for the schematic in 'Dynamic State Redundancy in a Bit of Information'). SEUs are injected via the EN1, EN2, PR, and CLR inputs (the EN1 and EN2 inputs are optional).



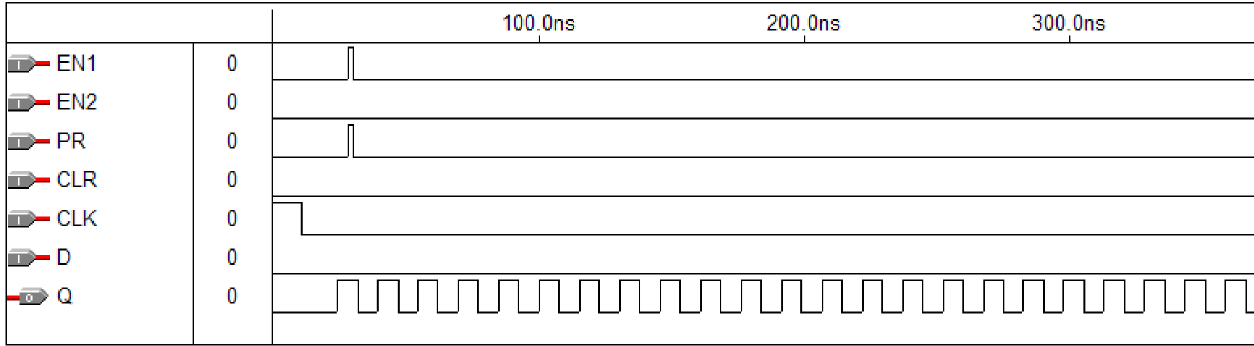
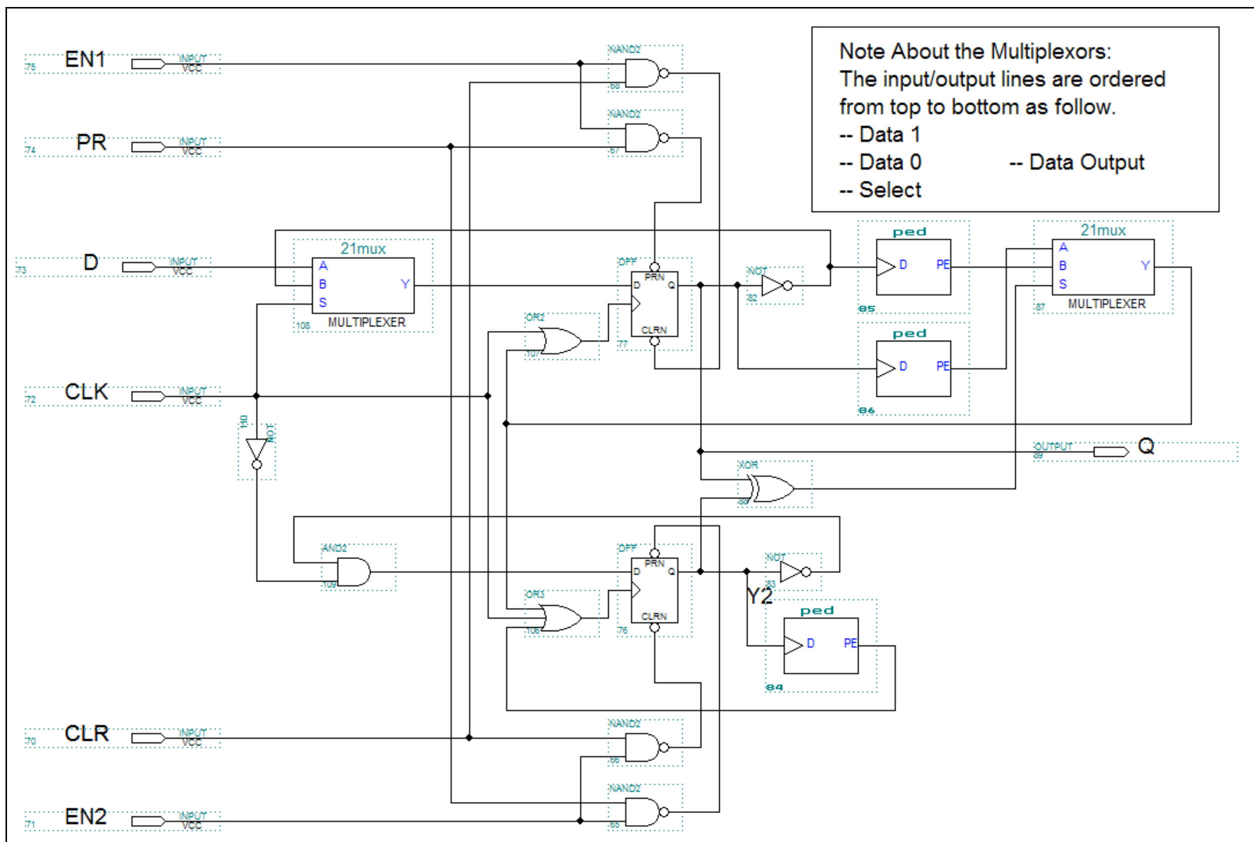


Figure 7: Schematic and timing diagram using the XOR state-value allocation

Figure 7 makes use of the XOR state-value allocation by detecting edges from the present state output. Both D flip-flops are positive-edge triggered, and the Q output is feed back to one flip-flop while the QN output (the output from the XNOR gate) is feed back to the other flip-flop. During a clock cycle and the Q output is low, when an SEU occurs in one of the flip-flops, the Q output will go high, generating a positive-edge. The positive-edge will trigger one of the flip-flops, causing that flip-flop to equate itself to the other flip-flop; in turn, the Q output will go low. If the Q output was high instead and an SEU occurs, the QN output will go high and trigger the other flip-flop; that flip-flop will differentiate itself from the other flip-flop, causing the Q output to go high. The problem with the design in Figure 7 is over-correction. That is, when an SEU occurs, the circuit will correct itself, then correct itself again due to the circuit's inability to differentiate between an edge from an SEU and an edge from self-correction.



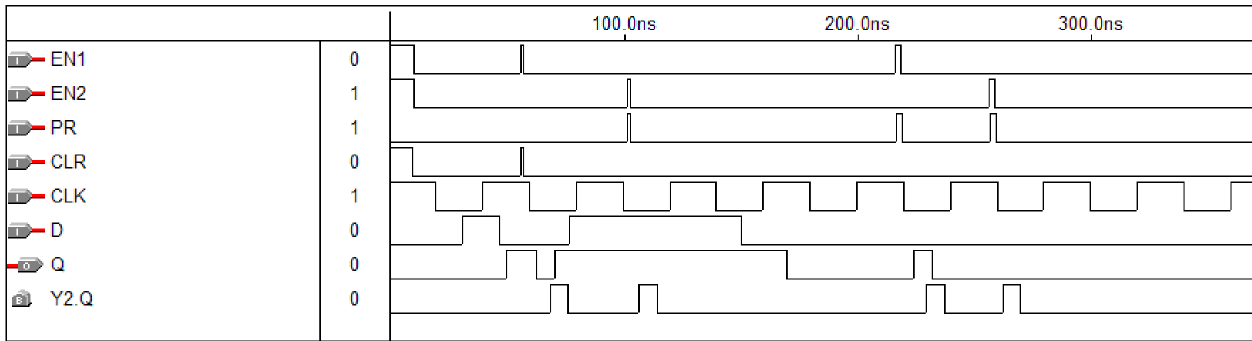


Figure 8: Schematic and timing diagram using the Y_1 state-value allocation (the PED blocks are positive-edge detectors which sends a '1' pulse)

The design using the Y_1 state-value allocation, shown in Figure 8, corrects itself appropriately in the event of an SEU during a low clock level. The major difficulty with the implementation is writing in the value from the D input during synchronization. To write in the value from the D input, the circuit's self-correction has to be disabled, so in the design in Figure 8, a high clock level disables self-correction. The drawback, however, is that when an SEU occurs during a high clock level, the circuit will remain in the incorrect state until the next clock cycle.

Dynamic State Redundancy in a Bit of Information:

Figure 9 shows how dynamic state redundancy can be implemented for bistable circuit. Two states, '0' and '1', are static redundant states represented by 00 and 11 respectively. The other two states are dynamic redundant states used to indicate an error condition. During normal operation, the circuit will be in one of the static redundant states. When an SEU hits the circuit, the circuit will fall into one of the dynamic redundant states. Depending on the previous state, the circuit will detect that it is in a dynamic redundant state and switch back to the previous state. While this circuit may handle SEUs, it will not be able to detect an MBU because an MBU will cause the circuit to change from one static redundant state to another static redundant state.

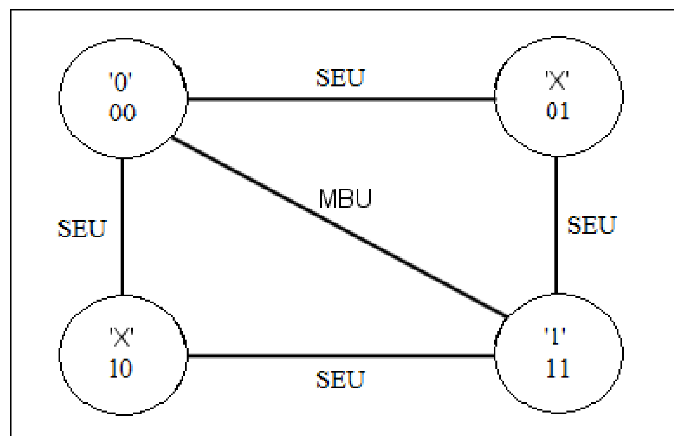


Figure 9: State diagram during a clock cycle using dynamic state redundancy

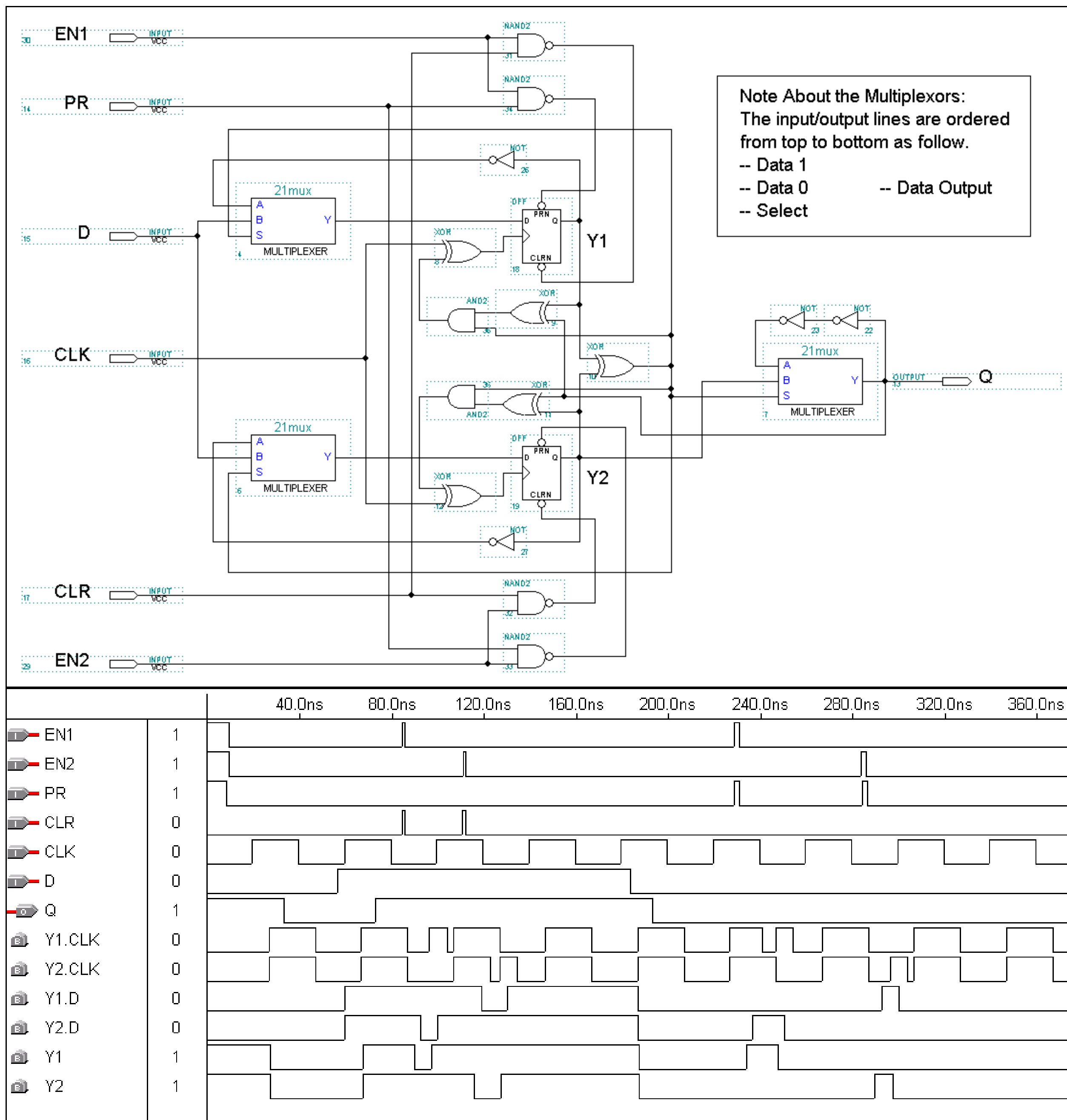


Figure 10: Schematic and timing diagram using dynamic state redundancy

The design in Figure 10 is a bistable circuit using dynamic state redundancy. Notice that the Q output is latched. During normal operation, the output Q is driven by the present state of the circuit. During self-correction, the Q output latches the previous state; this latched value is used to determine whether to detect for positive or negative edges at the flip-flops. One other possible design is to make the flip-flops dual-edge triggered instead of positive-edge triggered. In this manner, the latch at the output can be eliminated.

Conclusion:

While state redundancy helps mitigate the effect from SEUs and possibly MBUs, both error conditions cannot be eliminated. The problem with over-correction and disabling self-correction during synchronization demands further analysis. While the schematics presented use a high-level design specification, it may be more practical to implement a state redundant bit either within the package or on the transistor level.

References:

[1] Brodrick, David, Anwar Dawood, Neil Bergmann, and Melanie Wark. "Error Detection for Adaptive Computing Architectures in Spacecraft Applications." *Proceedings of the 6th Australasian Conference on Computer Systems Architecture*. ACM International Conference Proceeding Series, 2001: 19-26.

[2] Huang, Wei-Je, Edward J. McCluskey. "A Memory Coherence Technique For Online Transient Error Recovery of FPGA Configurations." *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*. International Symposium on Field Programmable Gate Arrays, 2001: 183-92.

[3] Lima, Fernanda, Luigi Carro, and Ricardo Reis. "Reducing Pin and Area Overhead in Fault-Tolerant FPGA-based Designs." International Symposium on Field Programmable Gate Arrays, 2003: 108-17.