IMPLEMENTATION OF FAULT TOLERANT MEMORY BASED UART THROUGH SPI INTERFACE

TELAPROLU ANUSHA
II.M.Tech , Dept of ECE, AM Reddy Memorial College of Engineering & Technology, Petlurivaripalem.

I GANESH KUMAR
Asst. Prof, Head-Dept. of ECE, AM Reddy Memorial College of Engineering & Technology, Petlurivaripalem.

Abstract

In nanometer technologies, circuits are more and more sensitive to various kinds of perturbations. Alpha particles and atmospheric neutrons induce single-event upsets, affecting memory cells, latches, and flip-flops. They also induce single-event transients, initiated in the combinational logic and captured by the latches and flip-flops associated with the outputs of this logic. In the past, the major efforts were related on memories. However, as the whole situation is getting worse, solutions that protect the entire design are mandatory. Solutions for detecting the error in logic functions already exist, but there are only few solutions allowing the correction, leading to a lot of hardware overhead in non processor design. In this paper, we present a novel technique that includes UART architectures and an SPI for their implementations, which reduces the cost of dealys in any kinds of circuit.

Keywords: *UART,SPI*

1. Introduction

The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin. The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end: The UART full form is "Universal Asynchronous Receiver/Transmitter", and it is an inbuilt IC within a microcontroller but not like a communication protocol (I2C & SPI). The main function of UART is to serial data communication. In UART, the communication between two devices can be done in two ways namely serial data communication and parallel data communication. In serial data communication, the data can be transferred through a single cable or line in a bit- by-bit form and it requires just two cables. Serial data communication is not expensive when we compared with parallel communication. It requires very less circuitry as well as wires. Thus, this communication is very useful in compound circuits compared with parallel communication. In parallel data communication, the data can be transferred through multiple cables at once. Parallel data communication is expensive as well as very fast, as its requires additional hardware and cables. The best examples for this communication are old printers, PCI, RAM, etc.

In this communication, there are two types of UARTs available namely transmitting UART and receiving UART, and the communication between these two can be done directly by each other. For this, simply two cables are required to

communicate between two UARTs. The flow of data will be from both the transmitting (Tx) & receiving (Rx) pins of the UARTs. In UART, the data transmission from Tx UART to Rx UART can be done asynchronously (there is no CLK signal for synchronizing the o/p bits). The data transmission of a UART can be done by using a data bus in the form of parallel by other devices like a microcontroller, memory, CPU, etc. After receiving the parallel data from the bus, it forms a data packet by adding three bits like start, stop and parity. It reads the data packet bit by bit and converts the received data into the parallel form to eliminate the three bits of the data packet. In conclusion, the data packet received by the UART transfers in parallel toward the data bus at the receiving end.

2. Preliminaries

UART Block Diagram

The UART block diagram consists of two components namely the transmitter & receiver that is shown below. The transmitter section includes three blocks namely transmit hold register, shift register and also control logic. Likewise, the receiver section includes a receive hold register, shift register, and control logic. These two sections are commonly provided by a baud-rate-generator. This generator is used for generating the speed when the transmitter section & receiver section has to transmit or receive the data. The hold register in the transmitter comprises the data-byte to be transmitted. The shift registers in transmitter and receiver move the bits to the right or left till a byte of data is transmitted or received. A read (or) write control logic is used for telling when to read or write. The baud-rate-generator among the transmitter and the receiver generates the speed that ranges from 110 bps to 230400 bps. Typically, the baud rates of microcontrollers are 9600 to 115200. As the *R* and *T* in the acronym dictate, UARTs are responsible for both sending and receiving serial data. On the transmit side, a UART must create the data packet - appending sync and parity bits - and send that packet out the TX line with precise timing (according to the set baud rate). On the receive end, the UART has to sample the RX line at rates according to the expected baud rate, pick out the sync bits, and spit out the data. More advanced UARTs may throw their received data into a buffer, where it can stay until the microcontroller comes to get it. UARTs will usually release their buffered data on a first-in-first-out (FIFO) basis. Buffers can be as small as a few bits, or as large as thousands of bytes.

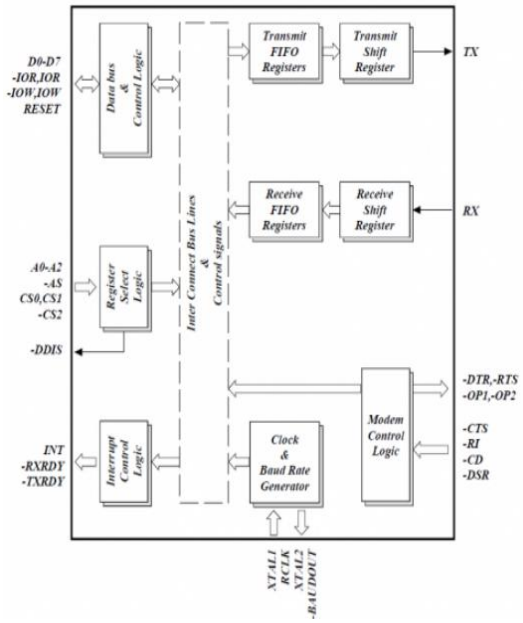

Figure 1: Internal UART block diagram

An UART (Universal Asynchronous Receiver/ Transmitter) is the microchip with programming that controls a computer's interface to its attached serial devices. UART is an integrated circuit designed for implementing the interface

for serial communications. It provides the computer with the RS-232C Data Terminal Equipment (DTE) interface so that it can "talk" to and exchange data with modems and other serial devices . As part of this interface, the UART also: Converts the bytes it receives from the system along parallel circuits into a single serial bit stream for outbound transmission On inbound transmission, converts the serial bit stream into the bytes that the system handles. Adds a parity bit (if it's been selected) on outbound transmissions and checks the parity of incoming bytes (if selected) and discards the parity bit Adds start and stop delineators on outbound and strips them from inbound transmissions May handle other kinds of interrupt and device management that require coordinating the on-chip communication of operation with high speed devices. Wait until the incoming signal becomes '0 ' (the start bit) and then start the sampling tick center. When the center reaches 7, the incoming signal reaches the middle position of the start bit. Clear the center and restart.

The UART includes both transmitter and receiver. The transmitter is a special shift register that loads data in parallel and then shifts it out bit-by-bit. The receiver shifts in data bit-by-bit and reassembles the data byte • Wait until the incoming signal becomes '0 ' (the start bit) and then start the sampling tick center. When the center reaches 7, the incoming signal reaches the middle position of the start bit. Clear the center and restart.

Serial Peripheral Interface (SPI)

The SPI is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems. The interface was developed by Motorola in the mid-1980s and has become a *de facto* standard. Typical applications include Secure Digital cards and liquid crystal displays. SPI devices communicate in full duplex mode using a master-slave architecture with a single master. The master device originates the frame for reading and writing. Multiple slave-devices are supported through selection with individual slave select (SS), sometimes called chip select (CS), lines.Sometimes SPI is called a *four-wire* serial bus, contrasting with three-, two-, and one-wire serial buses. The SPI may be accurately described as a synchronous serial interface,[1] but it is different from the Synchronous Serial Interface (SSI) protocol, which is also a four-wire synchronous serial communication protocol. The SSI protocol employs differential signaling and provides only a single simplex communication channel. SPI is one master and multi slave communication.

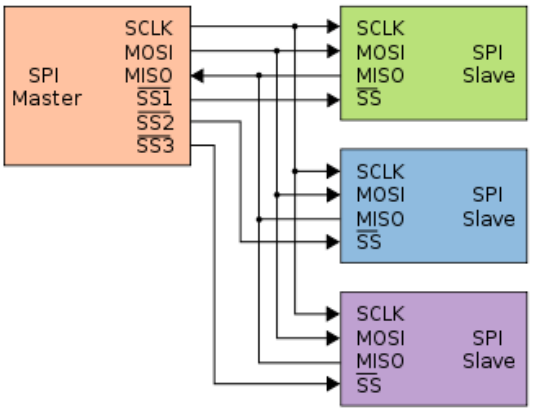

Figure 2: master and three independent slaves

In the independent slave configuration, there is an independent chip select line for each slave. This is the way SPI is normally used. The master asserts only one chip select at a time.Pull-up resistors between power source and chip select lines are

recommended for systems where the master's chip select pins may default to an undefined state.[3] When separate software routines initialize each chip select and communicate with its slave, pull-up resistors prevent other uninitialized slaves from responding.

Since the MISO pins of the slaves are connected together, they are required to be tri-state pins (high, low or high-impedance), where the high-impedance output must be applied when the slave is not selected. Slave devices not supporting tri-state may be used in independent slave configuration by adding a tri-state buffer chip controlled by the chip select signal.[3] (Since only a single signal line needs to be tristated per slave, one typical standard logic chip that contains four tristate buffers with independent gate inputs can be used to interface up to four slave devices to an SPI bus.)

3. Proposed Method

The previous descriptions detailed the UART of the memories to re-execute the K clock cycles before the error detection signal becomes active. However, before the retry phase starts, the subsequent pipeline connected to the memory output must be brought back to its Kth previous state. So, additionally to the CAM associated with the memory, an UART that preserves the related pipeline state must be inserted. This UART depends of the CAM implementation, and if a checkpoint SPI rollback implementation or a K-cycle SPI rollback implementation is considered. Fig. 7 describes the global UART implementation of a RAM (or a register file) with the first and the second implementation. The read data are saved during several clock cycles in the CAM and then it is saved in the UART for the registers related to the subsequent pipeline during a few more cycles. To perform a K-cycle SPI rollback, the CAM must save the K last read operations. However, instead of using an FIFO of K + P − 1 stages to save the data of the subsequent pipeline of P-stages, like in Section II-A, an FIFO of P − 1 stages is sufficient.

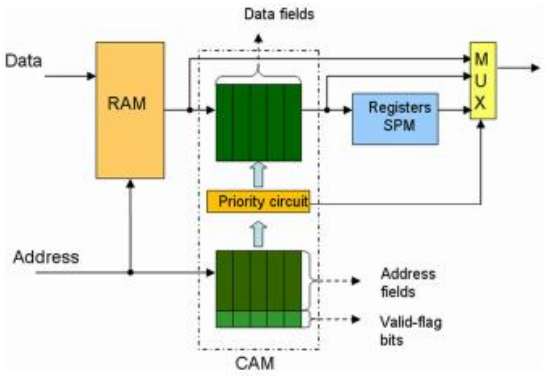

Fig. 3. Global memory UART

Therefore, the read data are saved in the CAM during the K clock cycles after they are read, and then during P − 1 additional cycles inside the registers UART. Therefore, the global implementation saves the read data during K +P−1 clock cycles as in Section II-A. The data of the FIFO are used to restore the context of the circuit K clock cycles before, and the data of the CAM to perform another time the K last operations of the circuit. Obviously, the CAM is not used during the SPI rollback phase. On the other hand, as the third implementation save the data that are written in the memory instead of those that are read, the register UART for the third implementation (Fig. 3) must be an FIFO of K + P −1 stages. In case a checkpoint SPI rollback implementation is chosen, the results are quite different. the re-execution process may involve a complete retry set. Therefore, the CAM used in each implementation must meet that requirement. To evaluate that cost, let us consider the best case for the retry sets: it

must avoid that the least recently saved context to be contaminated by an error. So, during a retry set the next save must start at least K clock cycles after the beginning of the retry set. If PM is the largest pipeline size of all regular pipeline, then each CAM must save the N + K last operations, with $N \geq K + PM - 1$. For the third implementation, the register UART, be composed of two FIFOs of P-stages. On the other hand, for the two first implementations, the requested checkpoints will be always saved either inside the CAM or in a register UART that save the least recent checkpoint (Fig. 4).

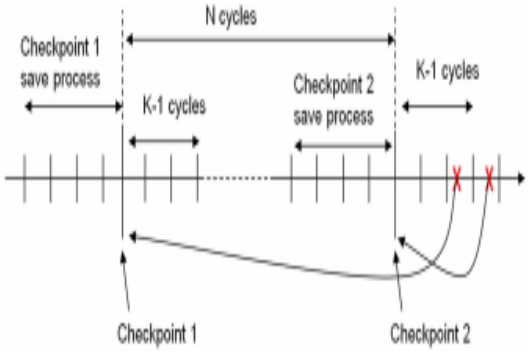

Fig. 4. Memory rollback operation

Therefore, the register UART in this implementation has only one FIFO of P-stages. So, the third CAM implementation always lead to a higher hardware overhead, due to the additional mechanism used to restore the context of the subsequent pipeline. Notice that those solutions differ from another solution described in [36], which used a read buffer and not a CAM. This solution is dedicated to instruction replay, and do not consider the global state restoration K clock cycles before of the subsequent pipeline. However, the interest of this solution is that they avoid having a priority circuit by performing a write back of the read buffer content inside the register file before the instruction replay starts [36]. Nevertheless, this idea is compatible with the first memory UART we proposed that save both read data and read addresses. The two other implementations cannot use this principle.

4. SIMULATION RESULTS

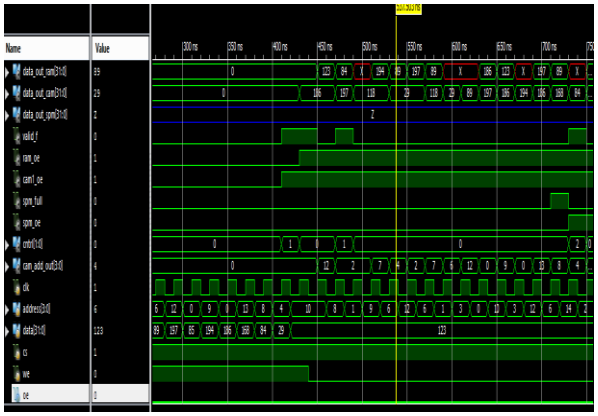

Fig. 5: Simulation output

The above result shows the simulation analysis of first and second implementation of the proposed method. Here clk, address, data, cs(chip select) ,we(write enable) ,oe(output enable) are the input signals, remaining all are output signals. Data out RAM should be monitored with respect to the address input. Initially, when we=1 then all the data input will be stored into RAM. Then memory based error checking operation performs using SPI rollbacks. Finally, ram_oe will be activated. Then Data out RAM signal will generates. DATA out CAM should be monitored with respect to the CAM address out. After error correction,cam1 oe becomes 1, then data out will be generated. The valid flag becomes active high, when ever error occurred, it will be auto corrected by using the proposed system.

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	310	35200	0%
Number of Slice LUTs	235	17600	1%
Number of fully used LUT-FF pairs	156	389	40%
Number of bonded IOBs	148	100	148%
Number of BUFG/BUFGCTRL/BUFHCEs	1	80	1%

Fig. 6: Design summary

The above table represents the design summary such as area utilized by proposed method. There is total of 35200 slice register available but only 310 of it used by the design. Similarly, out of 17600 look up tables, only 235 used.

```
                          Gate    Net
Cell:in->out      fanout  Delay   Delay  Logical Name (Net Name)
----------------------------------------  ------------
 LD:G->Q               1  0.326   0.550  s3/s2/DP_RAM/mem<8>_0 (s3/s2/DP_RAM/mem<
 LUT6:I0->O            1  0.043   0.000  s3/s2/DP_RAM/Mmux_address_1[3]_mem[15][7
 MUXF7:I0->O           1  0.169   0.000  s3/s2/DP_RAM/Mmux_address_1[3]_mem[15][7
 MUXF8:I1->O           2  0.122   0.293  s3/s2/DP_RAM/Mmux_address_1[3]_mem[15][7
 LUT5:I4->O           16  0.043   0.000  s3/s2/DP_RAM/Mmux_mem[0][7]_mem[0][7]_mu
 LD:D                    -0.035         s3/s2/DP_RAM/mem<8>_0
----------------------------------------
Total                     1.547ns (0.703ns logic, 0.844ns route)
                                  (45.4% logic, 54.6% route)
```

Fig. 7: Time summary

The above analysis shows the time summary of proposed design. It represents the total path delays and end to end delays generated in the design. The total 1.547 of ns delay is consumed.

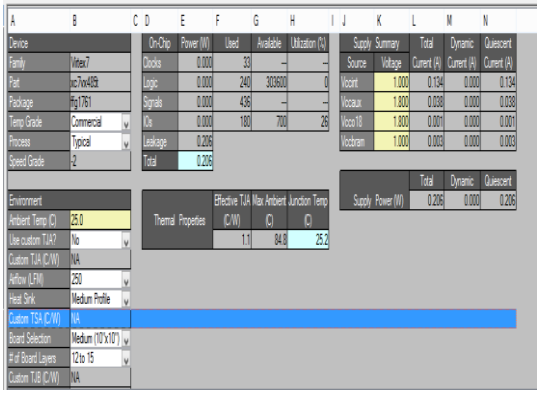

Fig. 8: Power summary

The above analysis shows the power summary of proposed design. It represents the total power consumed and power dissipation in the design. The total 0.206 watts of power is consumed.

CONCLUSION: We have presented several solutions to implement a SPI rollback technique at moderate hardware cost: a fixed K-cycle SPI rollback, or a checkpoint SPI rollback. It was also proposed three different CAM implementations for the massive storage preservation. The K-cycle SPI rollback with the two first CAM implementations provide very low hardware overhead compared to state-of-the-art technique by putting in common the data state preserves mechanism of several registers and the data state preserve mechanism of the memory. However, the downside is the number of cycles to perform the SPI rollback, as the stateof-the-art technique performs the SPI rollback in one clock cycle. In some cases, the checkpoint SPI rollback implementation provides a lower hardware overhead in case the target architecture has not a lot of memories output but has many irregularities. To have a complete soft-error tolerant implementation, we add an advance ECC implementation technique for embedded memories. It allows using ECC without any speed penalty, except in the rare case an error is detected. Due to the large part dedicated to the embedded memories in many modern designs, the global implementation of both techniques produces a very low hardware overhead. To have a complete implementation of that technique, we should add a soft-error detection technique mentioned in the introduction. Then, we could test the fault resilient property of the whole design. In addition, an improvement of the proposed technique would concern very large designs for which the proposed SPI rollback implementation is time-consuming even with an automated tool.

Moreover, the SPI rollback itself takes a lot of time in this kind of design. By splitting the whole design in several parts from which the SPI rollback could be performed independently in each of them, it may improve both of those mentioned issue at a moderate hardware cost.

REFERENCES

[1] R. C. Baumann, "Soft errors in advanced computer systems," IEEE Design Test. Comput., vol. 22, no. 3, pp. 258–266, May/Jun. 2005.

[2] N. P. Rao and M. P. Desai, "A detailed characterization of errors in logic circuits due to single-event transients," in Proc. Euromicro Conf. Digit. Syst. Design, Funchal, Portugal, 2015, pp. 714–721.

[3] B. Narasimham et al., "Characterization of digital single event transient pulse-widths in 130-nm and 90-nm CMOS technologies," IEEE Trans. Nucl. Sci., vol. 54, no. 6, pp. 2506–2511, Dec. 2007.

[4] M. P. Baze and S. P. Buchner, "Attenuation of single event induced pulses in CMOS combinational logic," IEEE Trans. Nucl. Sci., vol. 44, no. 6, pp. 2217–2223, Dec. 1997.

[5] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger, "Comparison of error rates in combinational and sequential logic," IEEE Trans. Nucl. Sci., vol. 44, no. 6, pp. 2209–2216, Dec. 1997.

[6] J. Benedetto et al., "Heavy ion-induced digital single-event transients in deep submicron processes," IEEE Trans. Nucl. Sci., vol. 51, no. 6, pp. 3480–3485, Dec. 2004.

[7] M. A. Bajura et al., "Models and algorithmic limits for an ECC-based approach to hardening sub-100-nm SRAMs," IEEE Trans. Nucl. Sci., vol. 54, no. 4, pp. 935–945, Aug. 2007.

[8] T. Bonnoit, M. Nicolaidis, and N.-E. Zergainoh, "Using error correcting codes without speed penalty in embedded memories: Algorithm, implementation and case study," J. Electron. Test., vol. 29, no. 3, pp. 383–400, Jun. 2013.

[9] P. Papavramidou and M. Nicolaidis, "Test algorithms for ECC-based memory repair in ultimate CMOS and post-CMOS," IEEE Trans. Comput., vol. 65, no. 7, pp. 2284–2298, Jul. 2015.

[10] R. Vemu, A. Jas, J. A. Abraham, S. Patil, and R. Galivanche, "A lowcost concurrent error detection technique for processor control logic," in Proc. DATE, Munich, Germany, 2008, pp. 897–902.