http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 6 (1): 66-73, 2007
ISSN 1812-5638

Dynamically Reconfigurable (Self-modifiable) Architecture for Embedded System-on-Chip Applications

N. Ramadass, S. Natarajan and J. Raja Paul Perinbam
College of Engineering, Guindy Anna University, Chennai, India

Abstract: Present trends in embedded system design indicates that the future world of computing will be ruled by System-on-Chip (SoC) platforms consisting of configurable components. Commercially available SoC platforms provide only limited amount of static customization of hardware. Now that SoC technology is emerging, single-chip multi processors are becoming feasible. Designing using such platforms has two drawbacks. First is the complexity of their on-chip interconnects. Second is the ambiguity regarding at what level software should be integrated. However, with increasing applications, diversity, time-varying requirements and convergence of multiple functionalities in a single embedded system, this is a growing demand for SoC that can be dynamically configured to adapt to changing requirements. In this fact, the design and implementation of a self-modifiable system is described. The reconfiguration logic and embedded application execute concurrently. Also, the data required for each reconfiguration is generated on demand. The viability of design is validated with an implementation of reconfigurable display controller. The methodology and design flow are also discussed. The developed system is currently being used to implement a reconfigurable speech codec.

Key words: Embedded architecture, System-on-Chip, dynamic self-reconfiguration, FPGA

INTRODUCTION

Embedded systems have significant heterogeneity in their computation. In the low level, embedded systems have regular, repetitive computations operating on large states of data with predictable data dependencies. At the higher level, the computations have irregular dependencies. Some system design problems are mainly driven by speed of operation, chip area and cost. If the highest operating speed is required and the algorithm is known, the only possible solution is to design an expensive ASIC. If a slight change in the algorithm occurs, the same ASIC cannot be used, forcing an and complicated redesign process. On the contrary, when the speed is not critical as in case of soft embedded systems, a General Purpose Processor (GPP) can be used. They have an advantage that a change in algorithm can be easily implemented with a change in software. Modern day GPP's are superscalar and pipelined to get more parallelism in software. They cover a large chip area. As reducing chip area and cost, increasing speed of operation and flexibility in algorithmic changes becomes increasingly important, reconfigurable systems have come into existence.

Developments in technology have given FPGAs the recognition as a reconfigurable logic with an emerging commercial future (Lala and Walker, 2000; Bondalapati and Prasanna, 2002). Reconfigurable computing involves manipulation of logic within the reconfigurable device at run-time. In the other words, the design of the hardware may change in response to the demands placed upon the system. These systems are characterized by their ability to continue to operate without interruption while subsections of the array logic are reconfigured. This presents the embedded engineers with intriguing new possibilities in the development of applications that maintain their own internal configuration state (Natarajan and Ramadass, 2003).

Classification of dynamically reconfigurable architecture: Dynamically reconfigurable architecture can be classified into two (Natarajan and Ramadass, 2003; Hartenstein, 2001). They are intra-circuit dynamic reconfiguration and inter-circuit dynamic reconfiguration. Former architecture employs circuits which are partial in functionality and intrinsically require huge dynamic reconfiguration to complete the entire functionality. In case of latter architecture, inactive circuits can be dynamically reconfigured to allow more functions to be performed with smaller devices.

In inter-circuit dynamic reconfiguration, modification of system operation is done at the system level. Inter-circuit reconfiguration requires a controller to schedule and implement dynamic reconfiguration. All FPGA's have

Corresponding Author: N. Ramadass, College of Engineering, Guindy Anna University, Chennai, India

a simple, configuration controller integrated into the array, which loads the initial configuration from an external source (Brown and Rose, 1996; Smith, 2000). Currently no FPGA's integrate controllers complex enough to support and schedule reconfigurations. Hence, external devices have always been required to manage and load bit streams, after the initial configurations has been made.

The modification controller can be either a software controller executing on a standard microprocessor/microcontroller, or a dedicated hardware controller. Inter-circuit reconfigurations require large functional changes. Each circuit configuration executes for a significant portion of time, before the next reconfiguration is required. In other words, inter-circuit reconfigurations require less frequent and many functional reconfigurations. Analyzing these characteristics, software controller is better suited for inter-circuit reconfigurable systems (Natarajan and Ramadass, 2004). But interrupt latencies in processor/controller while handling reconfiguration requests contribute significantly to reconfiguration time. A novel way to reduce large reconfiguration latencies is to implement the modification controller on an FPGA. The scheduling and sequencing operation of the application is captured early in the design process and this information is transformed automatically into controller logic. Presently, there is no tool existing for this purpose and this has been done manually based on simulations of the reconfigurable operation of the application.

The modification controller can be implemented on the same FPGA, eliminating the need for an external FPGA for this purpose. The advantage of this approach is that the overall system complexity is reduced and since the controller is closer to the array, latencies associated with accessing the configuration port is reduced. This controller configuration is static on the array throughout the execution of the application.

Self-modifiable system architecture and algorithm: The main component of self-modifiable system is shown in Fig. 1.

The event is an external signal based on which the request for reconfiguration is generated. The modification controller is responsible for processing these requests and convert them into appropriate control signals to select the current configuration to load. The bitstream generator produces the required configuration bits on demand, in response to these requests.

The algorithm for self-modification is as follows:

- At power on, the standard JTAG configuration controller transfers bitstreams from PROM to FPGA.
- This serial bit stream configures the modification buffer, modification controller and bitstream generator.
- Control is transferred to modification controller and system begins to execute.

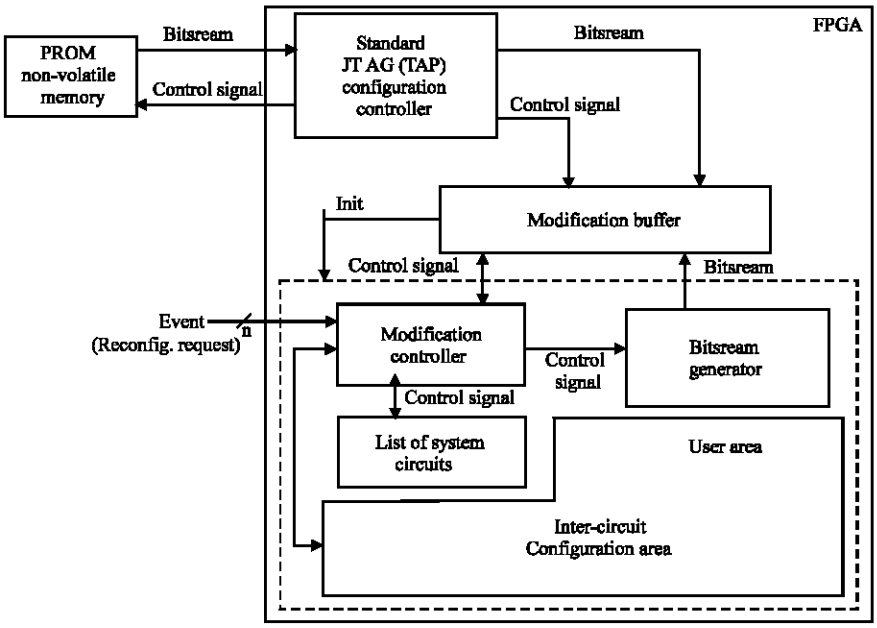

Fig. 1: Self-modifiable architecture

- Whenever a reconfig request is received, the next configuration is selected from the list of system circuits and corresponding control signals are sent to bitstream generator.
- The modification controller manages the dynamic loading of internally generated bitstream onto the array.
- Execution continues.

Once each reconfiguration is complete, the execution of the application continues immediately.

Implementation and analysis: The implementation stage consists of three stages. They are modeling, simulation and prototyping. The entire design was done using VHDL with same files used for simulation using Modelsim and prototyping using XILINX webpack tool. The prototyping platform consists of XILINX VIRTEX PRO FPGA.

The application implemented is a reconfigurable display controller as shown in Fig. 2 and operation is briefed in Table 1.

The switch inputs are used as event to trigger reconfiguration request. When S0, S1 = 01 is chosen, the system implements circuit 1 to drive 16 LEDs connected at output port and tristates the ports connected to 7 segment display. When S0, S1 = 10 is chosen, the system is reconfigured to circuit 2 to drive lsb LED and all other ports are tristated. When S0, S1 = 11 is chosen, the system is reconfigured to circuit 3 to drive 7 segment display and all the 16 LED ports are tristated.

Modeling: To reduce the difficulty in managing such a dynamically schedulable application and to provide a reliable implementation, the tools used must address the following issues:

- Automatic or manual partitioning of a conventional design
- Specification of the dynamic constraints
- Verification of the dynamic implementation through dynamic simulations at major steps of the design flow
- Automatic generation of the context controller for HDL or C implementation
- Dynamic floor planning management and guidelines for modular back-end implementation

The resulting adapted design flow is shown in Fig. 3. It is based on standard FPGA design tools. The input of the design flow is a conventional HDL static description of the application. Also multiple descriptions of a HDL file

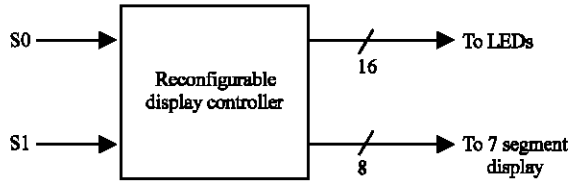

Fig. 2: Reconfigurable display controller

Table 1: Briefed operation

Switches			
S0	S1		
0	0	-	No action
0	1	-	16 LEDs (Circuit 1)
1	0	-	1 LED (Circuit 2)
1	1	-	7 segment display (Circuit 3)

can be provided to enable dynamic scheduling between the two hardware contexts sharing the same interfaces and area on the FPGA.

Based on the knowledge of the design architecture and the use of each sub-module in time, we can indicate which part of the feature to dynamically load and under which conditions. It is also possible to specify data management constraints to retain some internal states of the application after unloading and reloading the corresponding hardware context. By identifying portions of the design in the code at instance level (VHDL process or VHDL assignment), the modification specification can be made flexible and independent of the application description style. The outputs of this partitioning task are:

- A VHDL entity and architecture set corresponding to an identified modifiable hardware context and containing relevant HDL code
- A modification constraint file that contains the definition of each hardware context (in terms of content) and the associated constraints for loading and unloading them. The dynamic relations between two hardware contexts can also be specified, making them share the same area of the FPGA or by declaring them mutually exclusive in time.
- A VHDL entity and architecture set corresponding to the static part of the final implementation. This part includes all primary design instances on which no dynamic constraints have been applied. These instances will remain permanently inside the FPGA.

Simulation: Implementing such dynamic modification mechanisms must be checked-and with standard simulation tools. To be able to do so, we had to adapt the

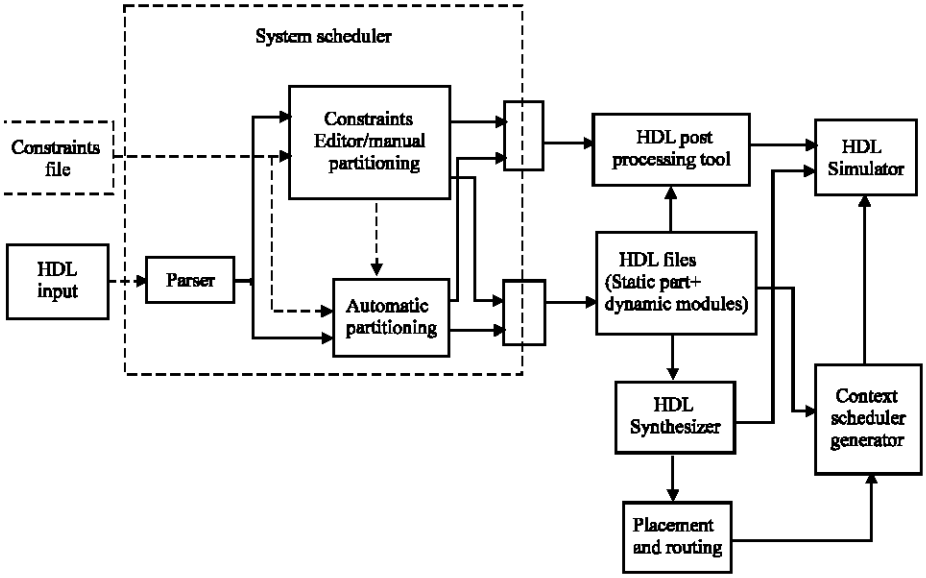

Fig. 3: Self-modification design flow

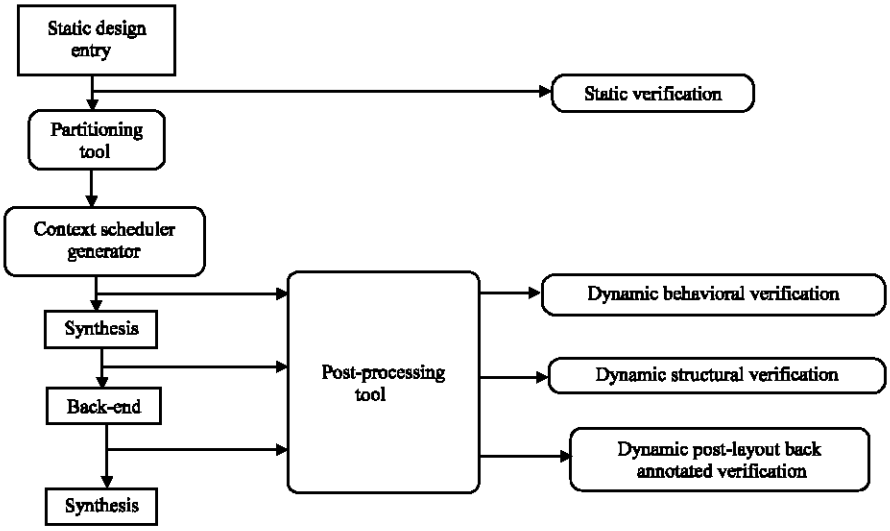

Fig. 4: Self-modification verification flow

new verification flow to verify the dynamic behavior of the design and the coherence of modification constraints applied to and the use of the design during simulation. The adapted verification flow is shown in Fig. 4. As a result, this dynamic verification can be performed with behavioral, post- synthesis, or post-layout HDL netlists.

Enter a partitioned database to the post-processing tool, which generates an equivalent VHDL description of the schedulable design that can be simulated under static HDL simulators. The unloading of each hardware context is modeled by a wrapper that isolates the inputs and outputs of each hardware context from the rest of the design according to the relevant scheduling constraints. This wrapper model is shown in Fig. 5. When a hardware context is not present inside the device, its output generates X or Z states to the rest of the design.

The post-processing tool generates two VHDL configurations:

- Functional context scheduling controller.
- Physical context scheduling controller.

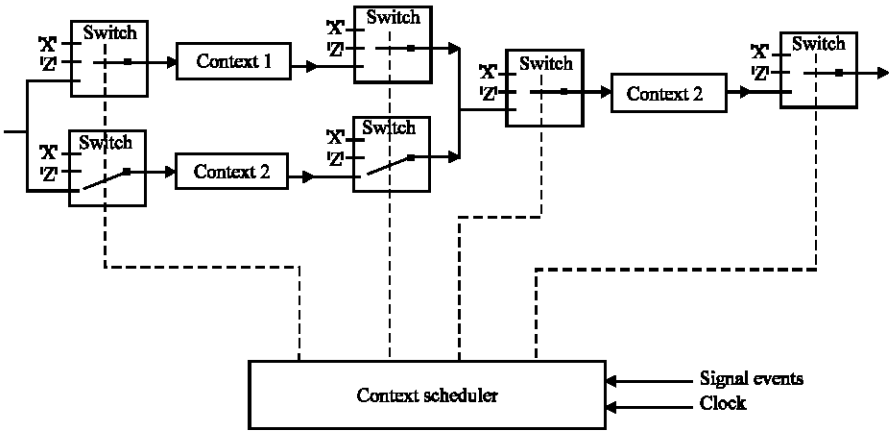

Fig. 5: Switch insertion for simulation

```
- - Static Configuration section
modif_controller: m_controller port map(. . . . );

- - Dynamic  Configuration section
        - - s0, s1 = 01
config 1: ckt 1 port map (Status ⇐ status . . . . );

        - - s0, s1 = 10
config 2: ckt 2 port map (Status ⇐ status . . . . );

        - - s0, s1 = 11
config 3: ckt 3 port map (Status ⇐ status . . . . );
```

Fig. 6: Structural VHDL code

The functional context scheduling controller is used during dynamic simulation. This module controls isolation switches by detecting events inside the application, according to the constraints. To assist with the verification process, the functional context scheduling controller can also issue different warnings each time a dynamic context is requested in violation of exclusivity rules defined in the constraints file.

The physical context scheduling controller is a synthesizable version of the physical context scheduling controller and is mapped as a static part of the FPGA. It detects the loading and unloading conditions according to the constraints and manages the hardware scheduling of the FPGA by reading bitstreams in storage memories and rewriting the FPGA's configuration. This module also provides an interface to monitor the modification process for hardware debugging purposes.

For dynamic behavioral verification, we can enter an estimation of the bitstream lengths into the post-processing tool to take into account the scheduling delays. After layout, we can place them with accurate ones, while a back-annotated VHDL file netlist can replace the VHDL partitioned code to obtain accurate vital verifications. Then, the static part of the design and the VHDL code of each hardware context is synthesized separately to obtain separate netlists. We can then use the back-end flow to place and route each context and to generate the associated bitstreams. Figure 6 shows a segment of structural level VHDL code.

The simulation results are shown in Fig. 7.

Prototyping result: Figure 8 shows a complete hardware scheduling architecture consisting of 4 real-time hardware contexts. The contexts are:

- Ledcounter
- Toggleled
- Seven-segment display
- Context switch input

When the system is switched on, it waits for a context switch event which is the context switch (cntxtsw) input. Once the event occurred, then the first hardware context (ledcounter) is scheduled to run. This context configures the FPGA to function as a 16-bit binary counter and display its count value on 16 LEDs. It simultaneously checks for context switch event. Again if the event occurs, then the first hardware context is deactivated and the second hardware context (toglled) is scheduled to run. This context configures the FPGA to toggle an LED at the rate of 1Hz. Once an event occurs at cntxtsw input the current context us deactivated and the next hardware context (sevsegdisp) is scheduled to run. The synthesis report obtained using Leonardo Spectrum Synthesis tool is as follows:

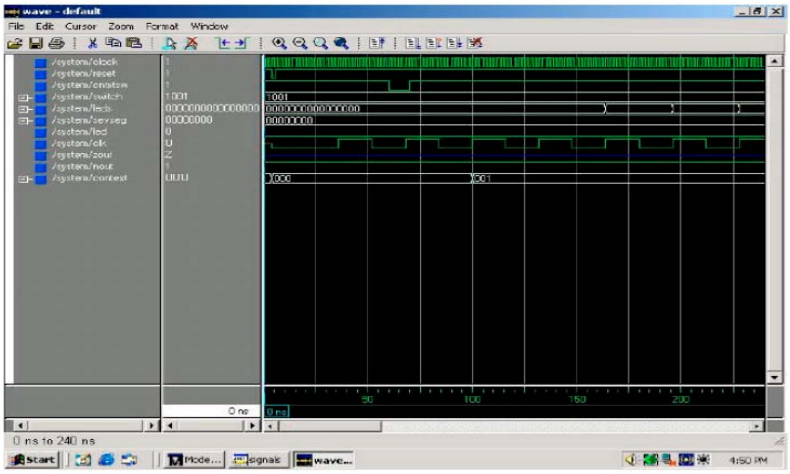

Fig. 7: Simulation waveform

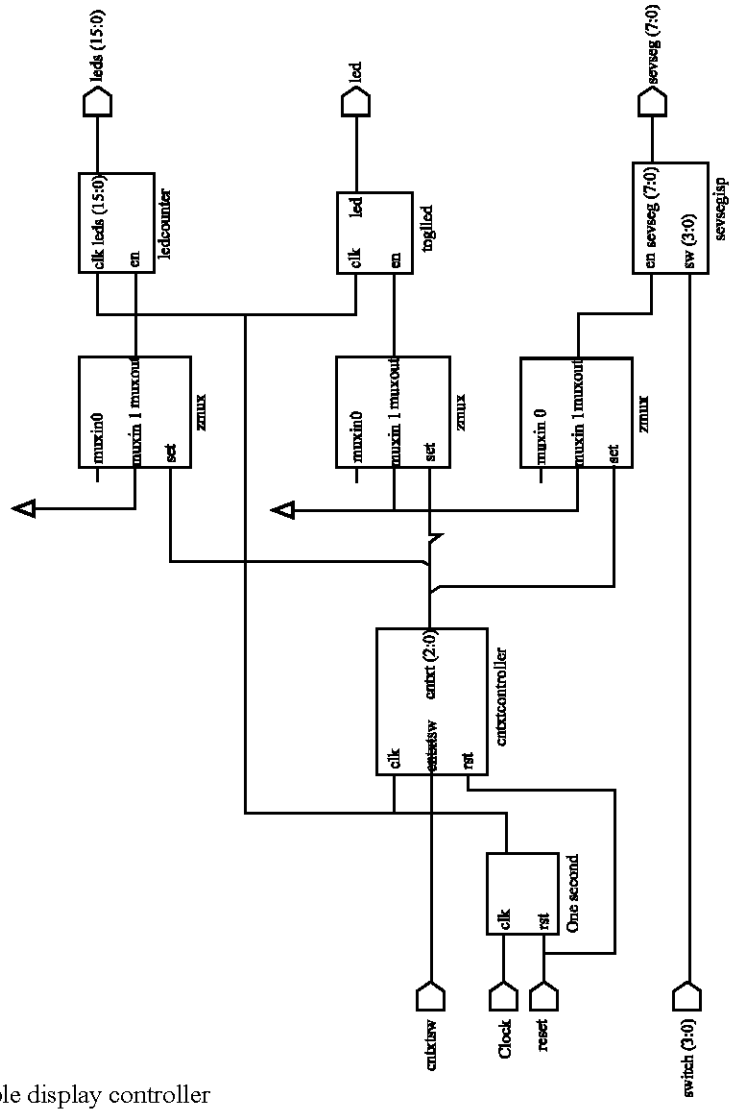

Fig. 8: Reconfigurable display controller

HDL Synthesis Resport

Macro Statistics

#	ROMs	: 1
	16x8-bit ROM	: 1
#	Registers	: 7
	1-bit register	: 4
	16-bit register	: 1
	3-bit register	: 2
#	Counters	: 2
	16-bit up counter	: 1
	26-bit up counter	: 1
#	Multiplexer	: 3
	2-to-1 multiplexer	: 3
#	Tristates	: 4
	1-bit tristate buffer	: 4

* Final Report *

Device utilization summary:

Selected Device : V100epq240

Number of Slices:	45	out	of	3072	1%
Number of Slice Flip Flops:	68	out	of	6144	1%
Number of 4 input LUTs:	83	out	of	6144	1%
Number of bonded IOBs:	31	out	of	146	21%
Number of TBUFs :	1	out	of	3072	0%
Number of GCLKs :	1	out	of	4	25%

TIMING REPORT

Clock Information:

Clock Signal	Clock buffer (FF name)	Load
ul_onesec:q	NONE	40
clock	BUFGP	28

Timing Summary:

Speed Grade : - 6

Minimum period: 11.935ns (Maximum Frequency: 83.787MHz)
Minimum input arrival time before clock: 6.442ns
Maximum output required time after clock: 8.542ns
Maximum combinational path delay: 9.735ns

Compledted process "Synthesize"

Fig. 9: Synthesis report

The synthesis results are shown in Fig. 9.

CONCLUSIONS

The presented prototype implementation architecture of a SoC architecture is capable to perform well for any embedded SoC applications. The actual design is scalable even for high-end embedded applications. Our future research work is focused towards developing an embedded speech codec based on this architecture. Furthermore, a hardware real-time operating system module is desired.

REFERENCES

Bondalapati, R. and V.K. Prasanna, 2002. Reconfigurable computing systems. In: Proc of IEEE, 90: 1201-1217.

Brown, S. and J. Rose, 1996. FPGA and CPLD architectures: A tutorial. IEEE Design and Test Computers, pp: 42-57.

Hartenstein, R., 2001. A decade of reconfigurable computing: A visionary retrospective. In: Proc. DATE, pp: 642-649.

Lala, P.K. and A.Walker, 2000. A on-line reconfigurable FPGA. In: Proc. of Defect and Fault Tolerance in VLSI systems: IEEE, pp: 275-280.

Natarajan, S. and N. Ramadass, 2003. An Architecture for RTCR based mixed-signal reconfigurable embedded device. Proc ICET, pp: 41-44.

Natarajan, S. and N. Ramadass, 2004. Design of an embedded computing system with RTOS for prototyping of research and laboratory process. Proc. NSNI, pp: 607-610.

Smith, M.J.S., 2000. Application Specific Integrated Circuits. Pearson Education Inc, pp: 722.