LOW-POWER WIRELESS DISTRIBUTED SIMD ARCHITECTURE CONCEPT: AN 8051 BASED REMOTE EXECUTION UNIT

by

Vyasa Sai

B.Tech, Jawaharlal Nehru Technological University, 2005

MS, North Dakota State University, 2008

Submitted to the Graduate Faculty of

Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in Computer Engineering

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Vyasa Sai

It was defended on

February 27, 2013

and approved by

James T. Cain, PhD, Professor Emeritus, Electrical and Computer Engineering Department

Yiran Chen, PhD, Assistant Professor, Electrical and Computer Engineering Department

Steven P. Levitan, PhD, John A. Jurenko Professor, Electrical and Computer Engineering Department

Bryan A. Norman, PhD, Associate Professor, Industrial Engineering Department

Dissertation Director: Marlin H. Mickle, PhD, Bell of Pennsylvania/Bell Atlantic Professor, Electrical and Computer Engineering Department

LOW-POWER WIRELESS DISTRIBUTED SIMD ARCHITECTURE CONCEPT: AN 8051 BASED REMOTE EXECUTION UNIT

Vyasa Sai, PhD

University of Pittsburgh, 2013

Power has become a critical aspect in the design of modern wireless systems, especially in passive device nodes such as Radio Frequency Identification (RFID) tags, sensor nodes etc. Passive RFID tags in particular use simple logic that is used to respond with a unique code or data to identify objects when queried by an interrogator, whereas wireless passive sensor devices use microcontrollers for sensor data processing. There is a need for a Minimal Instruction Set Architecture (MISA) for such passive nodes with regard to low power. In this context, passive node capabilities need to be explored, possibly to suit target applications, in order to enable more than just identification and perhaps less than those of a conventional microcontroller Instruction Set Architecture (ISA).

This dissertation research demonstrates a low-power wireless distributed processor architecture concept. The data and program instructions are stored on a powered interrogator providing wireless supervisory control for the remote passive node that has a basic processing core called the remote execution unit (REU). The interrogator and the passive node (REU) combination can be viewed as a complete processor or as multiple processing units forming the basis for a wireless distributed Single Instruction Multiple Data (SIMD) processor.

This research introduces and investigates the REU architecture using an 8051-MISA with the goal of reducing power consumption of the system. A novel low power data-driven symbol decoder-CRC along with the 8051-MISA based execution core design form the frontend and core part of the REU architecture. Clocked and asynchronous digital logic implementations of the REU core design are presented and correspondingly the power, area and speed comparisons are also provided.

Lack of strong support by commercial CAD tools is a major hurdle for synthesis of asynchronous designs. This research also presents a high-level design flow used to implement the asynchronous logic for the REU using traditional clocked CAD flows. This research work demonstrates immense potential to realize low power wireless passive sensor nodes for biomedical, automation, environmental, etc., applications especially while providing the basis for a programmable passive remote unit for distributed processing.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to sincerely acknowledge the valuable advice, guidance and constant support of my Advisor, Dr. M. H. Mickle throughout this research. I am deeply thankful to Dr. J. T. Cain, Dr. Y. Chen, Dr. S. P. Levitan and Dr. B. A. Norman, for their time, thoughtful inputs and valuable feedback.

This dissertation is dedicated to my mother, Dr. S. S. Kalpana. I would like to express my appreciation and thanks to my mother for her encouragement and inspiration throughout my PhD research. I wish to convey my special thanks to all my colleagues and friends for their support.

1.0 INTRODUCTION

Wireless sensor networks (WSN) are generally made up of a set of autonomous multifunctional sensor nodes distributed throughout a specific environment for monitoring real world data. These sensor nodes are used to collect environmental data and transfer this data to the user through the network. Besides collecting raw data, a node may also need to perform computations on the recorded data, eliminating the need to transfer raw data to a central server for each measurement [1], [2].

Consider a scenario with many raw sensor data readings that must be sampled simultaneously so as not to skew the measurements in time and correspondingly reducing the possible control bandwidth. The number of sensors required may be very large for some applications, e.g. environmental monitoring. By first principles, this situation is illustrated in Figure 1.1 for a set of n sensor nodes. In Figure 1.1, ε and Δ represent the data transmit time from each sensor to the central server and the preprocessing or conditioning time for the data at the individual sensors done in parallel respectively. In many cases, the raw sensor data must be preprocessed or conditioned before being used in system calculations in order to reduce the transmitted data. The raw data readings are compared to a threshold value in order to determine whether this data needs to be transmitted or not. If the raw sensor data reading is above the threshold value, it is transmitted to the central server instantaneously; else an aggregate value is

transmitted that includes the current reading along with the other data readings below the threshold[1].The transmission time (nε) especially in such a scenario is significantly reduced as opposed to preprocessing done at the central control where each and every sensor reading needs to be transmitted.

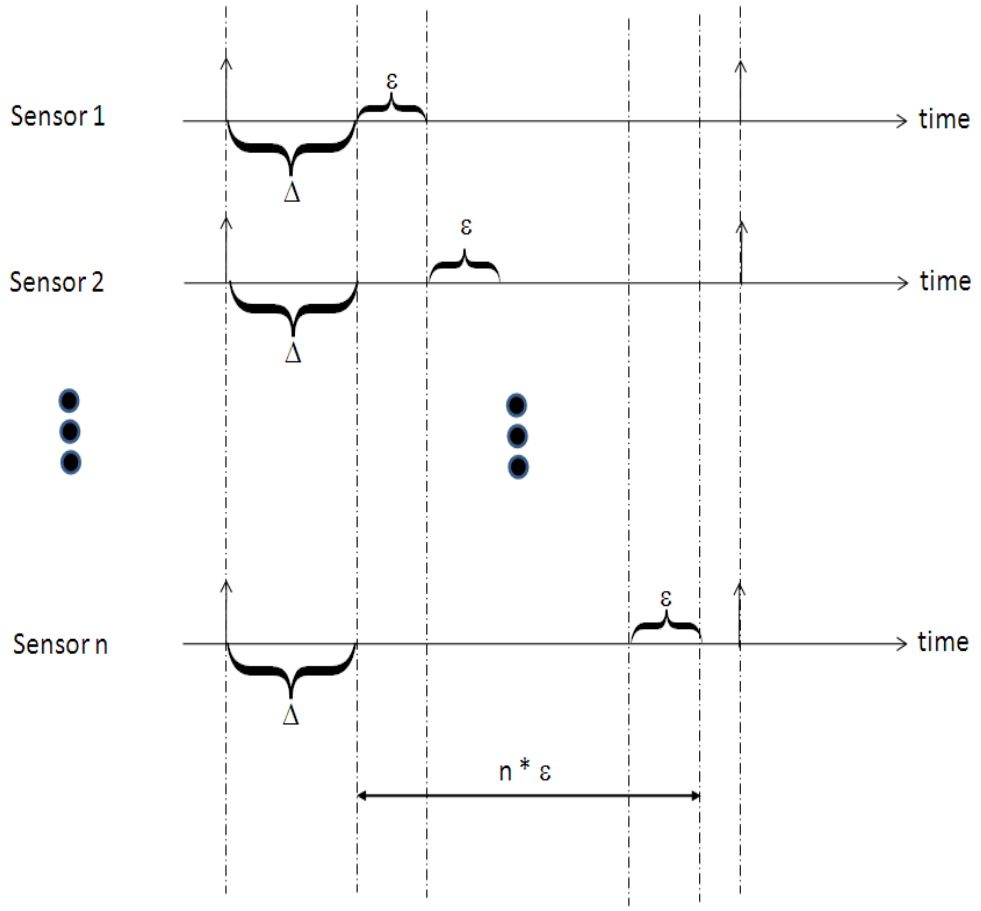

Figure 1.1: Timing Chart for a Sensor Network

This decrease in the amount of transmitted data in turn reducing the frequent radio transmissions is critical in increasing the power efficiency of the node [1]. There are many scenarios, in which the sensor data at each node is preprocessed or conditioned before the central server can further use it e.g. biomedical, physiological monitoring, environmental

monitoring, etc, [1], [2]. The main design constraint in such applications is the finite power budget for each wireless sensor node, as they require continuous and detailed monitoring over a long period of time.

1.1 OVERVIEW OF POWER TERMINOLOGY AND LOW POWER TECHNIQUES

Power has become a critical aspect in the design of modern processors and especially in wireless sensor devices such as embedded controllers, etc. The evolution of wireless devices with respect to size, weight, and battery life has enhanced their use in wider and more critical application spaces. For most portable devices, the integrated circuit (IC) components that form the digital processor are known to consume significant portions of the total system power [3]. High performance processors used in devices lead to the use of high clock frequency based designs that in turn lead to high power consumption. The IC typically dissipates power in the form of heat causing circuit degradation and operating failures. With the emergence of applications for battery-operated and battery-free portable wireless devices, thermal considerations and reliability issues increase and thus, there is a corresponding increased need for low power designs.

Power consumed by digital CMOS circuits can be broadly classified into two types: Static Power Consumption and Dynamic Power Consumption as shown in equation (1).

The power consumption equation is given as follows [3]:

$$P_{total} = P_{static} + P_{dynamic} \quad (1)$$

Static power (P_{static}) also known as leakage power is consumed when the circuit is said to be inactive or static or in a non-switching state. The main source of static power is due to the leakage of current from supply rail to the ground via various paths in the circuit. The leakage current can arise from substrate injection, sub-threshold effects, tunneling effects, etc. Leakage power is also influenced by nanometer CMOS technologies.

Switching power and internal power together add up for the dynamic power consumption ($P_{dynamic}$) of the circuit. Typically the dynamic power is dissipated when the circuit is active or in a switching state. Switching power dissipated is due to the charging and discharging of the load capacitance of the circuit. Internal power dissipated is due to the charging and discharging of the internal nodes of a cell. Also when both the PMOS and NMOS transistors are ON, short circuit current dissipates (short circuit) power that also contributes to the internal power consumption. Short circuit power is influenced by input transition times and the size of the transistors. For high performance systems, dynamic power is known to be the major portion of the total power consumption. The dynamic power can be described as follows [4]:

$$P_{dynamic} = C_L V_{dd}^2 f_{clk} \alpha + t_{sc} V_{dd} I_{peak} f_{clk} \quad (2)$$

Where switching activity is represented by α, C_L is the load capacitance; V_{dd}, f_{clk}, I_{peak}, and t_{sc} shown in equation (2) represent the supply voltage, frequency of the system clock, total internal switching current, and time duration of the short circuit current, respectively.

A short overview of low power design techniques is presented in the following paragraphs. The following circuit techniques are most commonly used to minimize the power consumption in wireless sensor nodes [5], [6].

A significant amount of a high performance processor's total power is being consumed due to the global clock that contributes to the dynamic power consumption. Asynchronous designs are increasingly becoming an integral part of numerous wireless sensor networks [7], [8], [9], [10] due to their low power advantages. These designs are characterized by the absence of any global periodic signal that acts as a clock. In other words, these designs do not use any explicit clock circuit, and, therefore, wait for specific signals that indicate completion of an operation before they go on to execute the next operation. Low power consumption, no clock distribution, fewer global timing issues, the absence of clock skew problems are the primary advantages of asynchronous designs over synchronous designs.

Power Supply Gating is also a low power circuit technique widely used to reduce the subthreshold leakage current of the system [11]. This process allows unused blocks in the system to be powered down in order to reduce the leakage current. This technique has been implemented in the Harvard sensor network system [12].

A subthreshold operation technique allows supply voltages (V_{dd}) lower than threshold voltages (V_{th}) to be used for lowering the active power consumption. This technique was first used in the complete processor design for wireless sensor networks at the University of Michigan [13], [14], [15].

1.2 OVERVIEW OF RFID BASED SYSTEMS

1.2.1 RFID Tag based Systems

RFID has become a key technology for automatic identification systems as it ensures automatic, accurate and real-time information tracking and management. RFID systems consist of tag(s) and interrogator(s) equipped with antennas as shown in Figure 1.2 [16]. The basic function of an RFID system is to automatically identify a person or an object that is "tagged." In general, an RFID tag is mainly composed of a microchip, and an antenna, which is used for wireless data transmission. These tags, upon being queried by an RFID interrogator transmit data over the air to reply. The data exchange between an interrogator and a tag is through RF signals. RFID tags can be broadly classified based on their power source as Passive or Active tags. Active tags are battery powered for carrying out all their on-board processing and data transmissions. The read range of active tags is about 100m or greater, and these tags are priced at about $20 or more. Passive tags are low cost and battery-free. Passive tags are powered by the impinging RF wave, which is also used for communication from an interrogator. The size of a passive RFID microchip is very small, about $0.4mm^2$ [17], [18]. Low cost passive RFID tags are used in tracking, supply chain automation, contactless credit cards, human implants, mobile robotics, unmanned medical nursing, container safety, etc

One of the well-known small and inexpensive passive RFID tag varieties is the Electronic Product Code (EPC). EPC tags are low cost and are designed to identify objects using a unique code [19]. These tags have a small amount of on-board memory. They store an

index to point to a database that stores information related to the tagged object, such as what is on a barcode.

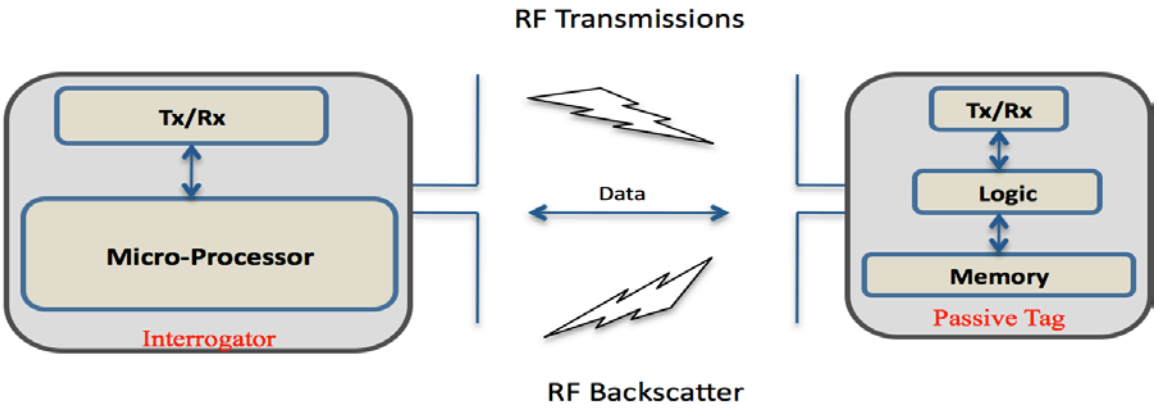

Figure 1.2: General Passive RFID System Architecture

The power consumption of a passive RFID microchip is one of the major limiting factors for read ranges of the tags. The basic design blocks of a microchip of an RFID tag consist of frontend, digital logic unit, and memory. The digital logic unit accounts for more than 35% of the power consumed by the entire tag [20], [21]. The input data decoding procedure, generally part of the tag frontend, is also a significant source of tag power consumption [22]. These tags use a high frequency clocked symbol decoder block to implement the decoding process.

1.2.2 RFID Sensor based Networks

In general, deployment of conventional sensor networks for environmental monitoring is limited due to the active life span of the on-board non-rechargeable power source. The number of sensors required may be very large for such an application. The sensors are battery powered,

and there is overhead involved for the periodic maintenance of the battery-assisted sensors. There has been much research into prolonging the limited lifetime of WSNs through efficient circuit, architecture and communication techniques [5], [6]. In summary, the use of a WSN system is strictly limited by the battery life of the sensor nodes. RFID-based sensory systems, however are extremely useful for maintenance and deployment of many sensor units, but have the advantage of being battery-free. In addition to this advantage, the wireless characteristics and unique ID aspects of the RFID system are proving to be a great asset to WSN [23], [24], [25] for the development of WPSNs.

1.2.2.1 Wireless Passive Sensor Networks A WPSN is a non-disposable and cost efficient system that operates based on the incoming received power [23], [24], [26], [27], [28]. The concept to remotely feed a sensor node on the power from an external RF source has led to the emergence of WPSNs. This concept was first introduced to power a passive RFID tag. It is well known that passive RFID design blocks form the basis for passive sensor node architectures [29]. Passive sensor node operating frequencies fall under the same industrial, scientific and medical (ISM) frequency bands as most RFID applications. The latest trend in environmental monitoring applications is to have sensor nodes operating at power levels low enough to enable the use of energy harvesting techniques [30], [31]. This facilitates a deployed system, in theory, for continuous sensing of a considerable extended period of time thereby reducing recurring costs.

Building blocks of a typical RF based wireless sensor node architecture consist of a sensing unit, a communication unit, a processing unit and a power source as shown in Figure 1.3 [23], [27]. The components of a sensing unit, in most cases, include a sensor(s) and an

analog-to-digital converter (ADC). A sensor is a device generally used to measure some physical quantity such as temperature, light, etc. The ADC is used to convert the typical received analog data signal into a digital signal so as to be processed by the microcontroller. The processing unit consists of a low power microcontroller and a storage block. The choice of the processing unit depends on the type of the power source available to the node. The microcontroller processes data, controls, and coordinates other component functionalities. The communication unit consists of an RF transceiver module that transmits and receives data to/from other devices connected to the wireless network. In the case of a WPSN, the power unit mainly delivers the RF-DC converted power to the rest of node units and also stores additional power based on availability.

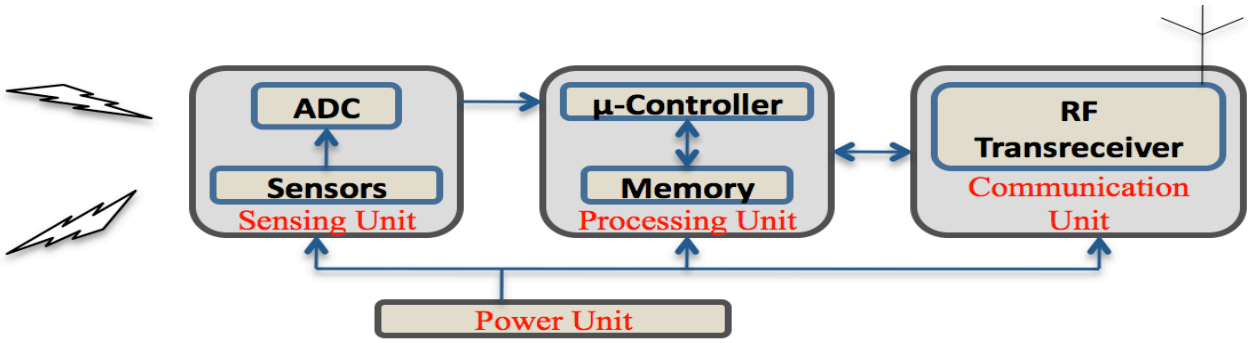

Figure 1.3: General WPSN Node Architecture

The major differences in the architectures of a conventional WSN node and a WPSN node are in the hardware of the power unit and the transceiver [23]. The power unit of the conventional WSN generally consists of a battery along with a support block called the power generator. The power unit for a WPSN node is basically an RF-to-DC converter-capacitor network. The converted DC power is used to wake up and operate the node or is kept in a

charge capacitor for future usage. A short range RF transceiver, typically a major power consuming unit on the node, is used in a conventional WSN as compared to a much simpler transceiver for modulated backscattering in the WPSN node [23], [32].

1.2.3 Power Comparisons of passive RFID nodes

Table 1.1 presents a current overview of the power consumption of various types of RFID based passive nodes. The RFID tag based digital processor design reported in [33], [34] is a conventional fixed function IC that is implemented as a non-programmable state machine that responds with a hard-coded ID when queried by the interrogator. In [35], [36] the sensor integrated passive RFID tag has a fixed ID assigned to each sensor in order to support maintenance and field deployment of many sensors. The associated digital processor does not support any arbitrary computation and typically reports sensed data in addition to the RFID tag functionality.

WISP (Wireless Identification and Sensing Platform) is a battery-free sensing and computation platform that uses a low power full programmable microcontroller for enhanced functionality of the RFID tag based sensing [37], [38]. In [37], [38] the wireless passive RFID sensing design is compliant with the UHF RFID interrogator. Table 1.1 clearly illustrates the significant increase in power consumption from a typical RFID passive tag to the computationally enhanced passive RFID nodes. In [24], a general-purpose low power 16-bit programmable microcontroller (MSP430F2132) is used for managing the entire passive node. But the use of full microcontrollers is known to consume significant amounts of power especially in the context of passive sensing.

Table 1.1: Power Comparisons of passive node based on their functionality

Reference	Full-Design Type	Power (μW)	Comments
Man (2007) [33]	Passive RFID Tag	3.436*	Process: 0.18μm Voltage: 1.8 V (*Baseband processor)
Yang (2010) [34]	Passive RFID Tag	0.963*	Process: 0.18μm Voltage: 1.1V (*Baseband processor)
Cho (2005) [35]	Passive RFID Tag-Temperature & Photo Sensor	5.1	Process: 0.25μm Voltage: 1.5 V
Jun (2010) [36]	Passive RFID Tag-Temperature Sensor	6*	Process: 0.18μm Voltage:0.8 V (*Baseband processor)
Joshua (2006) [37]	Passive RFID based Sensing platform-μC	5400**	6MHz, 3V (**Microcontroller)
Alanson (2007) [38]	Passive RFID based Sensing platform-μC	846**	3MHz, 1.8V (**Microcontroller)

1.3 STATEMENT OF THE PROBLEM

Passive RFID technology is becoming increasingly common in different environments such as home, office, industry, hospitals, library, etc enabling quick and anytime access to real-time data on uniquely identifiable passive nodes throughout their entire lifetime. Passive RFID based sensor nodes such as passive RFID tags and WPSNs mainly deal with the collection or storage of data, and transmission of that data back to the interrogator. The interrogator primarily collects and processes the data sent by the nodes. Such nodes are typically not programmable, as they have conventional fixed function IC's as their digital processors. This restricts the computational flexibility available to the node.

WPSN being an emerging research area; there is little documentation on all the power efficient scenarios applicable to passive sensor devices. In [23], [24], [26], [27] efficient antenna designs, low-power transceivers were introduced for WPSNs. But it is not only important to have energy efficient front-end and power unit designs, there is also a need to have low-power novel processor designs that allow greater ranges for WPSN nodes. In the context of having preprocessing done at the sensor side, existing computationally enhanced nodes are known to consume considerable power as mentioned in Table 1.1. This limits the performance of an RFID system especially with respect to operating ranges. Thus, to create low power passive RFID based nodes either to fit the need of any particular class of applications or as a standalone, it is desirable to remove or reduce as many of the power consuming characteristics as possible.

Low power IC design optimizations can be achieved at various levels, such as the algorithm-level, the architecture-level and the circuit-level. The research vision of this dissertation is to build a new generation of architectures for low power applications most of which are derived from new notions of distributed computing. Generally, distributed computing is mostly interpreted as multiple cores and processors. Serial, single thread processing is not viewed as distributed other than in data flow chains with proximate hardware elements.

Single Instruction Multiple Data (SIMD) is a well know class of parallel computers in Flynn's taxonomy [39]. SIMDs have the ability to perform the same operation on multiple data simultaneously for processors with multiple processing units. Synchronisation between processors is not required. SIMD processing is also a form of vector processing. An add operation in a traditional scalar processor would produce a single result by adding up one pair of

operands. In a SIMD processor, a single add operation produces multiple sums of independent operand pairs on different processing units. Figure 1.4 represents the inherent parallelism in a SIMD processing flow.

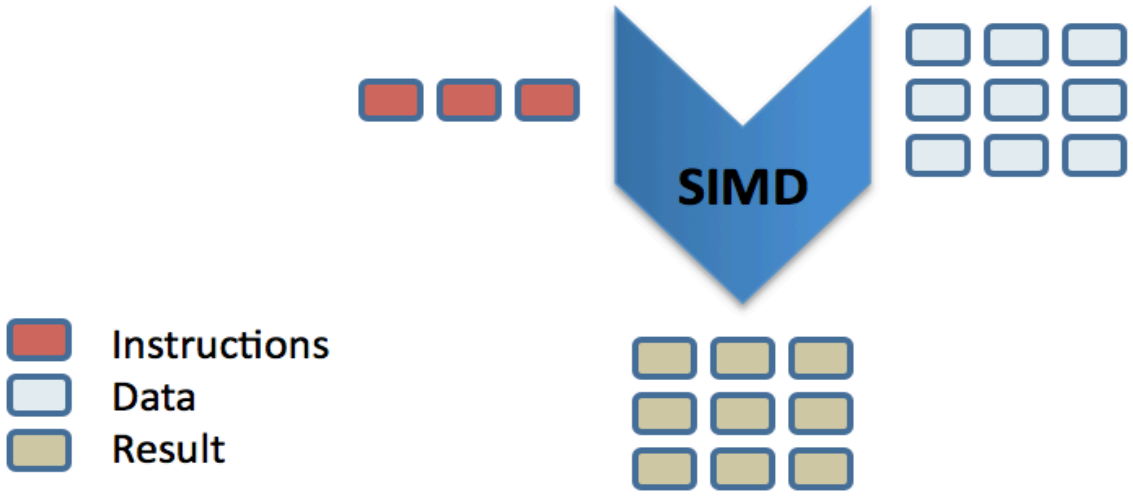

Figure 1.4: A SIMD Processing Flow

Applications where a single sample produces multiple values, which are operated on at a large number of data points can take advantage of the SIMD architecture. Due to the higher level of parallelism available in SIMD architectures, instructions can be simultaneously applied to all of the data in the processing units within a single operation. Such a conventional SIMD class of processor architecture typically has wired implementation of multiple processing units that execute the same instruction sequence on different data items. The reconfigurability and scalability of processing units in such wired SIMD implementations are not convenient.

This research, however, will distribute single thread processing to a wirelessly connected digital processing core of a passive node executing sequential instructions as a single

thread paradigm. A conceptual distributed architecture with a reduced instruction set, combined with low power circuit techniques, will be introduced and investigated with the goal of reducing the power consumption of the system. These low-power considerations for the processing core will also be based on factors such as target application and trade-offs that can be made as long as the functionality required of an application is met within a given time constraint. There is a need for novel architectures that take into account such factors, especially for passive device applications.

The low-power distributed architecture will consist of splitting the processor architecture of a node into two basic design blocks to support multiple remote passive processors with wireless reconfigurability. Each of the multiple passive processors is represented as a wireless node (WN), in Figure 1.5. The digital processing core of the WN will be the remote execution unit (REU), which will consist of instructions that provide basic flexibility in manipulating data. The other block forms the interrogator (active block), which will act as the control unit for the REU that supports the types of instructions such as decision, branching, etc. The WN will wirelessly execute instructions issued by the interrogator. In this scenario, the program to be executed by the WN will be stored in the interrogator and the commands will be transmitted to the REU one at a time. The interrogator remotely sends instructions to the WN, which are accordingly executed and corresponding results are communicated back to the interrogator. Thus, the interrogator and the REU based node combination can be viewed as a complete processor or as multiple processing units (nodes) forming a wireless SIMD distributed architecture class of processor systems.

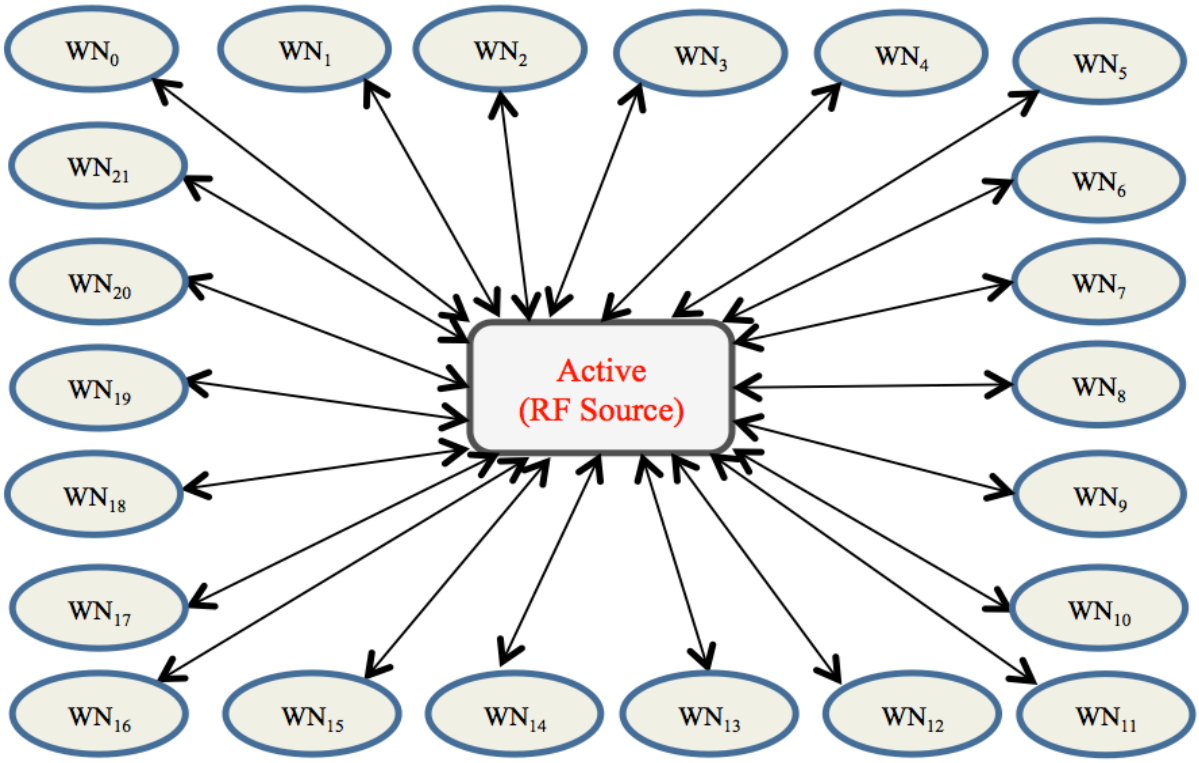

Figure 1.5: Wireless SIMD Network Architecture

The wireless SIMD distributed processor architecture allows for a flexible wireless reconfigurability and scalability of the number of processing units as opposed to a conventional wired SIMD system. In other words, the REU based node lends itself a distributed architecture based remote node processor(s), which can be replicated to produce a wireless SIMD distributed processor system.

As part of the dissertation research, the REU will be implemented in two ways to allow comparison especially with respect to power consumption, speed and area. Both the implementations will consist of a frontend block for RF communication and a core block as an execution unit. The frontend block will be used to decode the input command and check for

validity of the received command. Both the REU versions will have the same frontend implementation; the major implementation with the difference being the core design. The first REU core implementation will be a clocked design that executes instructions based on clock pulses provided wirelessly by the interrogator at essentially any frequency from DC to the limit of the wireless medium or the technology implementation of the REU. The other REU core implementation will be an asynchronous design that uses no explicit clocking mechanism. Both the asynchronous and the clocked REU designs will be implemented using clocked CAD tools.

In summary, the main objectives of this dissertation are (1) to develop a low power programmable REU core design of the node processor as a wireless distributed architecture that operates remotely from the interrogator, and (2) to implement both the asynchronous and the clocked REU core designs and correspondingly investigate for comparison of their respective power consumptions.

1.4 OUTLINE OF THE DISSERTATION

- Chapter 2 introduces the distributed processor architecture concept. A sample instruction sequence flow between the interrogator and wireless node (REU) is presented for illustrative purposes using 8051 instructions.

- Chapter 3 describes the proposed high-level architectures for the clocked and the asynchronous REU designs that include the individual REU front-end and the REU core.

This chapter also presents the 8051 subset of instructions chosen for REU design and associated concepts needed for the implementation of the REU architecture.

- Chapter 4 presents the high-level design flow using clocked CAD flows that were used to design, synthesize, and implement the clocked and the asynchronous REU logic. It elucidates the modifications to the traditional clocked CAD flows in order to implement the clock-less modules of the REU design. This chapter also presents the post-layout simulation and verification results of the frontend, core and the entire REU. It also provides a comparison of power, area and speed for both the clocked and asynchronous REU core implementations.

- Chapter 5 presents the conclusions and future directions of the dissertation research.

2.0 WIRELESS DISTRIBUTED PROCESSOR ARCHITECTURE CONCEPT

2.1 THE ARCHITECTURAL EMBODIMENT

Conventional processors have basic blocks such as Control, Memory and ALU connected and hard wired as a single processing unit. A conventional processor is basically distributed into blocks to support multiple remote passive processors with wireless reconfigurability. A wireless SIMD distributed architecture concept for low power applications is introduced in this section. The distributed architecture of a conventional processor consists of two blocks namely an active block and a WN block as shown in Figure 2.1. Active implies a battery or wired power source. The active block (also known as the C&M (control and memory) unit) is always connected to the power supply. This block contains major components such as the control and storage units that are larger in size and/or consume a considerable amount of power (for example: Controller, RAM, ROM, etc). The design of the C&M block can be very flexible as it can be designed as a synchronous design block due to constant power being supplied to it. The core of the passive block consists of a digital processing core is the REU. This block is not connected to any power supply instead uses power extracted from the incoming RF signal from the active block. Both the blocks communicate thru RF signals as they can be efficiently transmitted through free space. The RF modulated signal from the active block is demodulated for incoming commands, and the output is the simple modulation of the backscatter from the antenna.

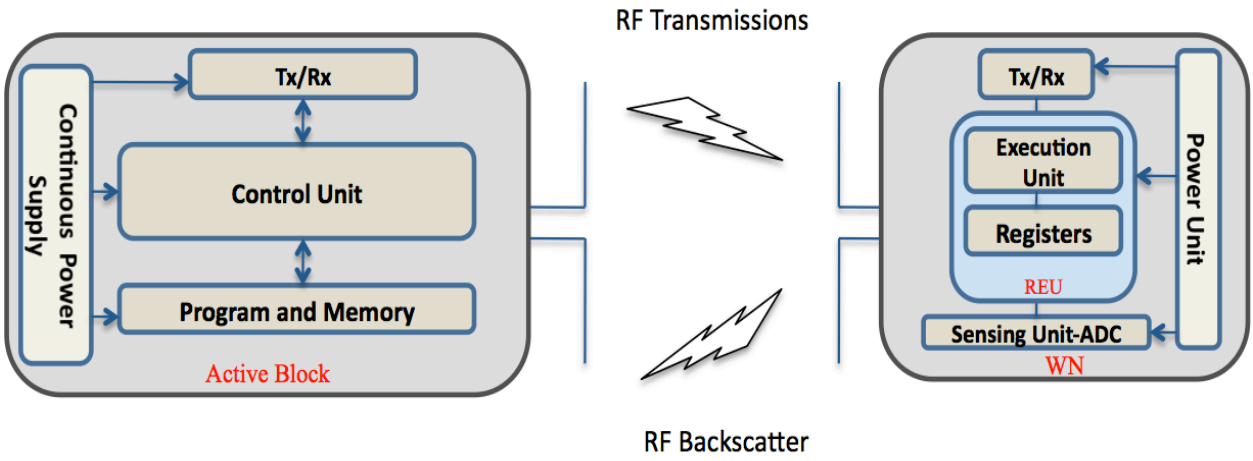

Figure 2.1: Proposed Distributed Architecture

2.2 AN APPLICATION SCENARIO

In an RFID based system, the active block acts as the interrogator and the WN acts as the passive tag. A typical RFID interrogator wirelessly transmits commands to the remote passive RFID tag, which then executes these commands and responds back to the interrogator [40]. Passive RFID tags typically use Application Specific Integrated Circuit (ASIC) design to provide logic to respond to commands from an interrogator. The commands from the interrogator can be viewed as instructions issued to a digital computer [41]. Thus, the interrogator and the tag combination can be viewed as a complete processor or as multiple processing units. This will form the basis of the proposed distributed concept.

The Control and Memory (C&M) is an RF equipped control and storage block and the WN block is an REU with minimal storage capacity. The first block is allowed the flexibility to

overall be a classical von Neumann or Harvard type architecture. Commands stored on this block are transmitted wirelessly to the REU. The intent is to keep the REU block as simple as possible so as to maintain low power requirements and/or an effective read range from the C&M. Any unnecessary complexity on the passive REU will be moved onto the C&M powered block. One of the main power reduction techniques employed in the proposed REU design is to eliminate the need for a clock. The execution rate of the remote processor is to be controlled by the C&M block (interrogator).

The focus of the current research is the elements and concepts of the design of a passive execution unit that operates remotely and wirelessly from the C&M. The low-power distributed architecture will provide the basis for a passive reconfigurable processor with multiple execution REU's [42], [43]. This is a classical single instruction multiple data processor architecture as with the ILLIAC [44]. This distributed architecture has the potential for developing low power applications in sensor networks, radar, digital signal processing, instrumentation, measurement, medical electronics, embedded systems, etc.

A basic state diagram for an RFID tag-Sensor node is shown in Figure 2.2 [35]. Such nodes are generally used in environment monitoring, traffic control, battlefield surveillance, etc. This RFID tag-Sensor node basically uses an RFID type unique code assigned to each sensor to enhance the wireless environment monitoring support and deployment procedures. The basic states of this type of transponder are On, Interrogating and Active. Upon receiving the energizing RF field, the transponder enters the On state. On request from the base station, the transponder enters the Interrogating state, and the demodulator and the decoder are activated to

enable the respective blocks. The basic blocks of the transponder include the ROM and the sensor block. The sensor and the ROM blocks are enabled exclusively by the command from the central server (interrogator) for power management [35]. There can be multiple sensor blocks based on the different types of sensors needed for an application. Finally, in the active state, the selected functional block is enabled and the requested information is sent back to the interrogator. The number of RFID tag-Sensor nodes may be very large in applications such as environmental monitoring and, hence, it is important for the nodes to be able to successfully send all the measured data to the interrogator avoiding collisions. The RFID (tag/interrogator) interface acts as a serial bus that travels through the air. In a wired serial bus application, bus contention is prevented by arbitration. The RFID interface also needs arbitration so that only one node transmits data over the "bus" at one time. Current RFID protocols use many existing collision prevention methods that make sure that only one tag communicates at one time [54].

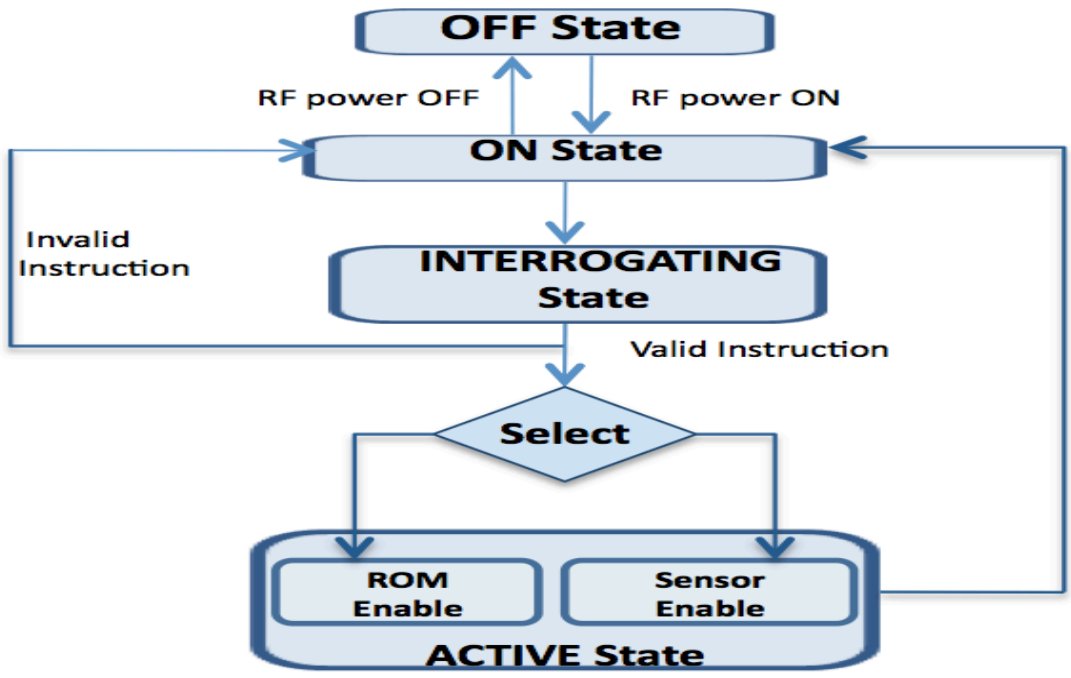

Figure 2.2: State Diagram of a (RFID tag-Sensor) Transponder

Wireless nodes basically sense specific aspects of a region in which they are deployed and occasionally send sensed data to the requested, interrogator. The sensed data generally contain errors due to many factors such as resource constraints and environmental factors. Therefore, the interrogators cannot rely on single-sensor data sensed at a point of time. Data redundancy is another issue with the data sensed by various sensors. Hence in many applications the aggregated form of sensed data from single or multiple sensors over time is preferred [2]. Finding average temperature, velocity, location, pressure, etc., are well-known examples in many applications. When interrogators require an aggregated form of sensed data, performing computation of the sensed data and sending its aggregate reduces the communication overhead [45]. Existing RFID Tag-sensor nodes typically do not have any programmable arithmetic processing capabilities [35]. Adding lightweight computational elements based on the target application can enhance the wireless node.

Sensor networks employ preprocessing at the node so that every sensor sample need not be transmitted on the radio therefore not consuming all the wireless bandwidth available to the network. Transmission of only necessary sensor data readings over the radio saves the available stored energy on the node. The focus of this dissertation is on low power solutions to wireless passive sensor node processor architectures based on 8051 instructions. The choice of the 8051 is justified by the fact that it is still one of the most popular embedded processors [46], [47], [48], [49]. Furthermore, due to its small size and low cost, it has numerous applications where power efficiency is necessary. The 8051 microcontroller most commonly used in wireless nodes is considered as an example for exploring its ISA and its application to the proposed distributed

processor design concept. Using the 8051-ISA based customization of the REU architecture as part of the distributed architecture is illustrated in the following paragraphs.

Consider a simple communication scenario between the interrogator and the node for the aggregation of sensor values at the node using an 8051 instruction [50]. The amount of temporary storage and the ALU capabilities of the WPSN node processor will be chosen to maintain low power requirements. Assume a 8051 default register bank (R0-R7) as the temporary memory space available for the execution unit. The major function is an ADD operation and, hence, the choices of the arithmetic instructions that would be part of the REU on the passive node are ADD A, Rn and/or ADDC A, Rn. The minimal data transfer instruction necessary would be the MOV A, Rn; MOV Rn, A and MOV Rn, #DATA (8-bit). The passive node processor will support only those features required to interface and communicate with the interrogator. Therefore, the branch, comparison, load and store instructions compatible with the i8051 ISA will be implemented on the interrogator side rather than on the passive side.

Consider an ADD operation: ADD A, R1 (A = A+R1), where R1 denotes one of the eight (R0-R7) 8-bit 8051 working registers for a selected register bank and A denotes the 8-bit accumulator register. Figure 2.3 represents a high-level sequence diagram for an ADD operation. Let the sensed data be stored in any of the sensor registers that could be any one of the (R0-R7) directly mapped to the sensor. The interrogator sends out instructions to transfer the sensed data stored in the sensor register to the R1 (assuming not a sensor register) and A registers respectively. These 8051 working registers along with the accumulator form the temporary storage on the REU processor. On receiving the ADD instruction, the passive node

processor (REU) performs the addition operation on the already existing value in the accumulator and the new R1 value. The computed result is stored in the accumulator register. Alternatively, the tag could then be instructed to send the result back to the interrogator in a register to memory transfer where the register is mapped as a transmitting buffer. The interrogator serves the role of main memory for storage of large data items. Loads (reads) and stores (writes) are performed by the interrogator and tag communicating data values using the over-the-air protocol. The C&M node will contain main memory that acts as the major storage area for the majority of data items.

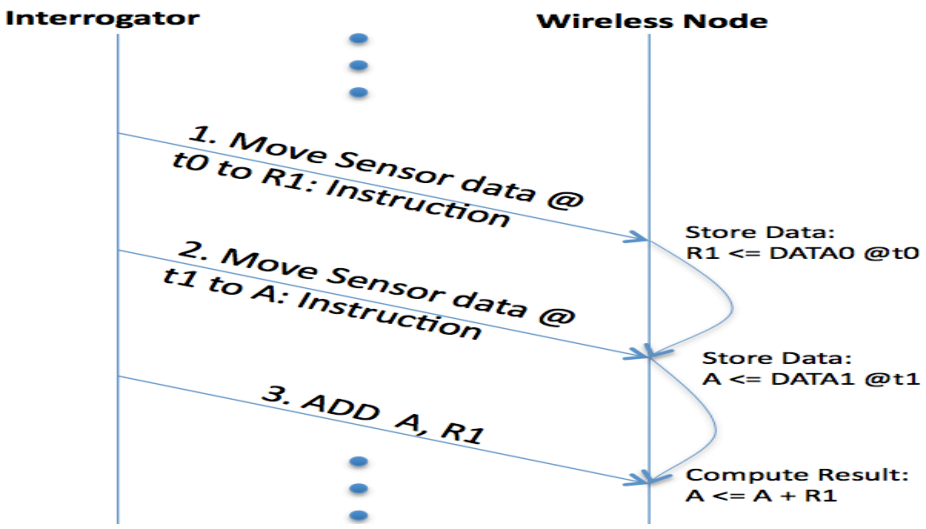

Figure 2.3: Sequence diagram for an ADD operation

Sensor applications require special purpose hardware suitable to cater to a different set of requirements. It is necessary to explore the instruction architecture space in order to enhance the capability of the execution unit depending on the application. Characteristics of the target applications and the utility of the sensors make it important to choose applicable hardware for sensor networks on a case-by-case basis. Some of the well-known basic core algorithms form a

class of simple applications such as the sum-array (aggregation of all values in a list), Top10 (finds top 10 values in a list), majority consensus (finds the majority values in a list), min-max finder (finds minimum and maximum values in a list), Binary search (typical search algorithm for a sorted list), Matrix Multiplication (matrix multiplication for small size matrices), etc. [50]. To arrive at an energy efficient processing solution on a sensor node, there are always communication/computation tradeoffs. Hence, the choice of a design for sensor node architecture not only depends on the various power management techniques, but also on the application space.

The procedures to choose the components and the associated instructions based on the 8051 architecture illustrated in this chapter can be used to generalize and extend these concepts to any microprocessor ISA such as the Motorola 6800, Intel 8085, etc. In summary, the distributed architecture components, especially the ones in the passive cell, need to be based on the target application requirements. Once the application requirements are determined, the ISA of the target architecture can be selected for the implementation of the distributed architecture. A minimal set of instructions based on the corresponding minimal register (ROM/RAM) storage can be chosen to form the target ISA. Because these instructions are dependent on registers of the target architecture, it is necessary to choose only the required registers, which can be part of the passive cell, providing maximum flexibility. Therefore, determination of the right combination of different factors blending into a distributed design will decide the power requirement of the passive cell based on the application. The next chapter introduces the proposed low power REU designs based on the 8051 ISA that form the passive cell architecture.

3.0 PROPOSED LOW POWER REU ARCHITECTURES

This chapter presents the major contributions of this research: a novel data-driven symbol decoder and minimal 8051-instruction set based computation unit. The proposed decoder and computation unit form the basis for the REU frontend and the REU core architectures respectively. The input instruction is an encoded bit sequence that will have command bits (Op-code) as well as data bits. The command bits instruct the processor to perform a specific operation on the input data bits. A data-driven CRC block is used as part of the frontend to check the validity of the received instruction. The frontend consists of registers and delay elements that are used to decode and store the received encoded instructions. The basic necessary set of instructions supported by the REU based on the target application is defined as the minimal ISA (MISA). The core unit is based on a MISA compatible with the 8051 ISA. The REU core consists of an 8051-compatible ALU unit and controller and requires a minimum number of temporary storage registers to support the selected 8051 instructions.

Both the data-driven symbol decoder design and the computation unit designs can be independently used for various wireless applications such as RFID, WPSN, etc. A potential architecture for combining the data-driven decoder design and the execution unit to form the low power REU will be established in this chapter. The motivation for low power design choices and details of individual REU components will be also described in the following

sections. The major focus of this chapter will be to introduce the design philosophies, and elements of the REU architectures that have been implemented as custom asynchronous and clocked digital designs to reduce power requirements.

3.1 REU FRONTEND

3.1.1 Motivation

Wireless serial data transfer environments are always prone to noise and transmitter-receiver synchronization issues especially over long distances [51]. The synchronization issues in general concern the need for the receiver to use the same clock frequency as the transmitter in order to accurately detect the transmitted data. In modern data transmissions, all traditional receivers extract the original data from the encoded bit stream using an explicit clock. This clock is extracted from the transmitted data, or is separately generated using additional circuitry [35], [52].

Manchester encoding and variable pulse width encoding techniques are very widely used in wireless digital transmissions [53], [25]. For example, a variable pulse width encoding technique is currently used in passive RFID tags [54]. Oversampling with a clock is a classical decoding process for pulse width modulated signals [35], [52]. The demodulated input to the decoder is the pulse interval encoded data that are converted to the regular binary symbols at the decoder output. The conventional decoding scheme is shown in Figure 3.1.

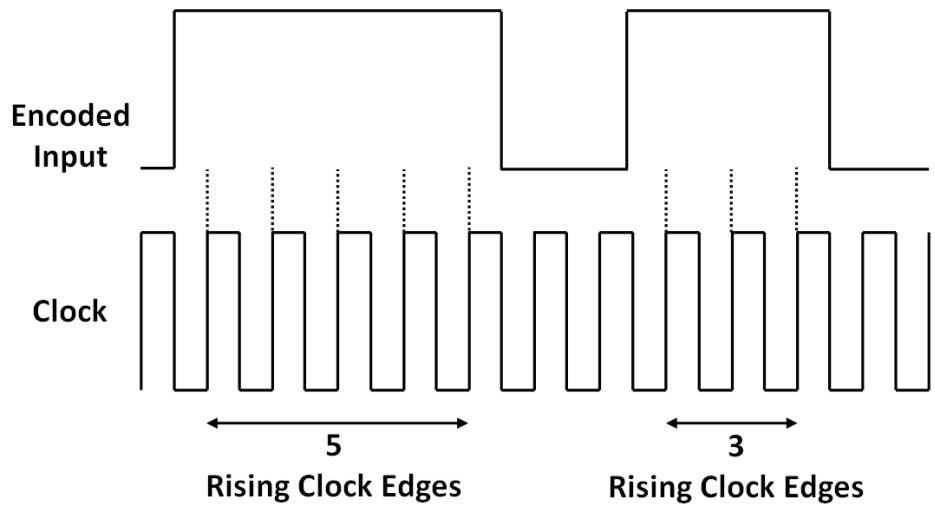

Figure 3.1: Conventional Decoding Scheme

The conventional decoder block represented in Figure 3.2 is a part of the conventional RFID tag frontend architecture [35], [52]. The decoder mainly consists of counters, registers, comparators, and a high frequency oscillator. This decoder is explicitly driven by a high frequency clock of several MHz. The demodulated input is converted to a series of binary symbols '1' 's and '0' 's by comparing the number of clock pulses occurring in each symbol period. In Figure 3.1, when the symbol period is high, '1' is identified based on the occurrence of five (5) clock pulses whereas '0' is identified based on the occurrence of only three (3) clock pulses. A typical data rate of 40kHz, corresponding to a symbol period length of 6.25µs, is considered as an example [54]. The decoded data are fed back to a digital block which actually runs at a much lower frequency range of about 40kHz - 640kHz [52]. Hence, a low frequency clock generator is required to convert the high frequency clock of several MHz to a low frequency clock to drive the digital backend of the tag. Such high frequency operations consume considerable amounts of power [33], [55].

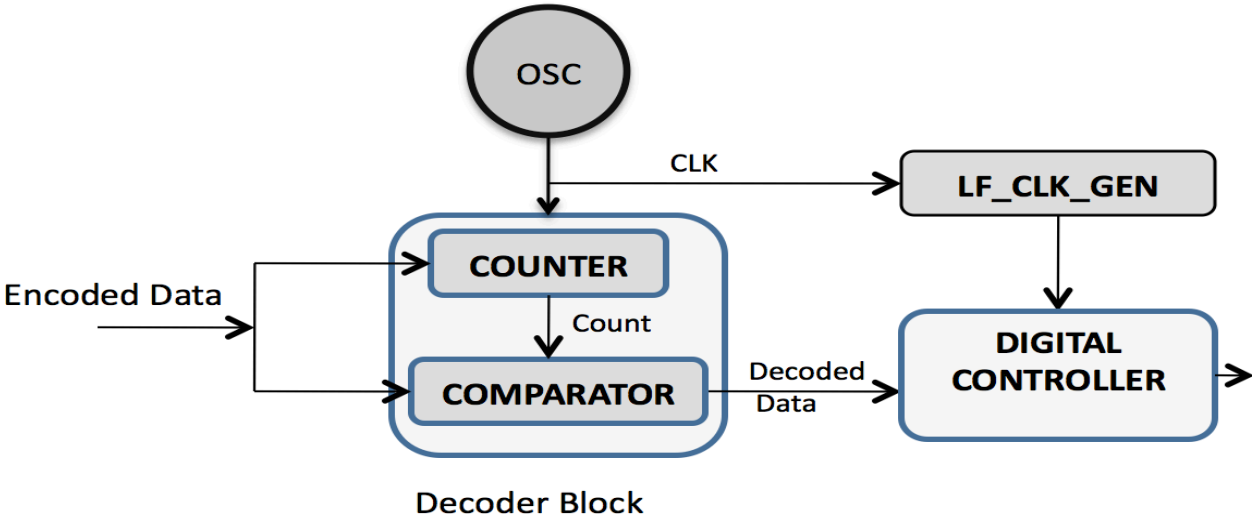

Figure 3.2: Conventional Decoder block of a passive RFID Tag

The next section introduces a novel pulse width-coding (PWC) scheme for low power applications. The symbol width properties of the PWC scheme allow the traditional decoder at the receiver side to be replaced with a low-power data-driven decoding circuit. The major advantages of the proposed PWC scheme include minimal decoder hardware, synchronization among the transmitter, receiver, and low-power decoder. The low-power PWC scheme can be applied to various wireless passive device designs such as passive tags, passive sensor nodes, etc.

3.1.2 Pulse Width Coding Scheme

The PWC scheme represents symbol-0 with a square wave (with time period P_0), which has a pulse width D_0, and symbol-1 with a square wave (with time period P_1), which has a pulse width D_1, where $D_0 << D_1$ and $P_0 = P_1$. Figure 3.3 represents an example 8-bit input symbol data stream "01011010" in the proposed PWC encoded format. From Figure 3.3, P_0 and P_1

represent time periods of symbol-0 and symbol-1, respectively. Each of symbol-1 and symbol-0 occupies an active time span (i.e., pulse width interval) equal to that of D_0 and D_1, respectively. The PWC decoding mechanism is described in the next section.

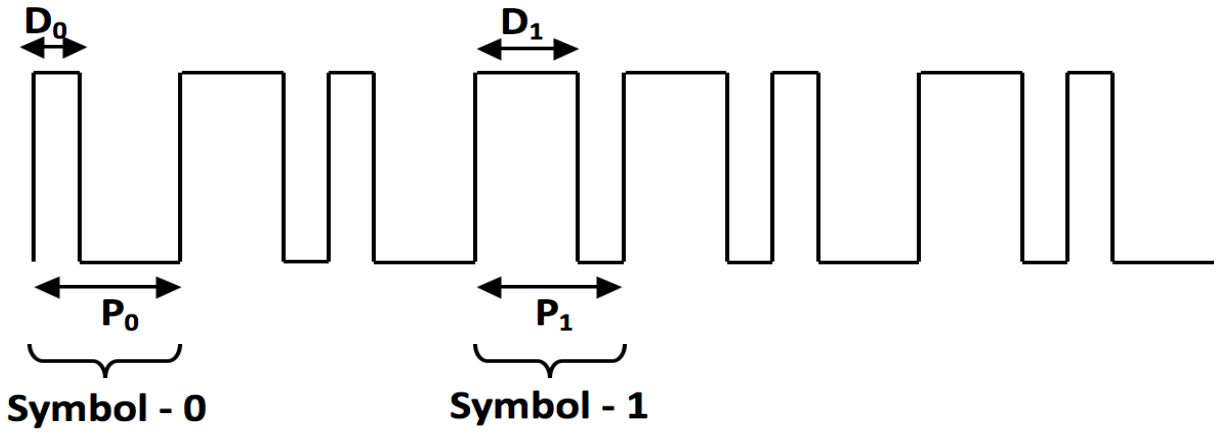

Figure 3.3: Pulse Width Encoded Data

3.1.3 PWC Decoding Mechanism

The received waveform after demodulation results in an encoded digital data stream as shown in Figure 3.3. The proposed PWC decoding scheme is shown in Figure 3.4 using the encoded symbol data "01011010" as an example. The encoded data can be decoded by sampling (Signal A in Figure 3.4), with the same encoded data stream delayed by time Δ (Signal B in Figure 3.4). In other words, Signal A (PWC encoded data) represents the input demodulated serial data representing the symbol data stream "01011010". The Signal B is the delayed version of the incoming PWC Signal A. The rising edge of Signal B is used to sample the incoming Signal A. It can be clearly seen from Figure 3.3 that the first decoded output binary bit is "0", second decoded binary bit is "1" and so on. At the rising edge of Signal B, the decoded bit is "1",

whenever both the signals occur (i.e., especially symbol-1). Hence for the above-mentioned Signal A example, the output binary decoded bit sequence is "01011010". For successful PWC decoding, it is necessary to have $D_0 < \Delta < D_1$. One possible way of reduced power consumption is by minimizing the value of D_0, which minimizes the delay circuitry in particular.

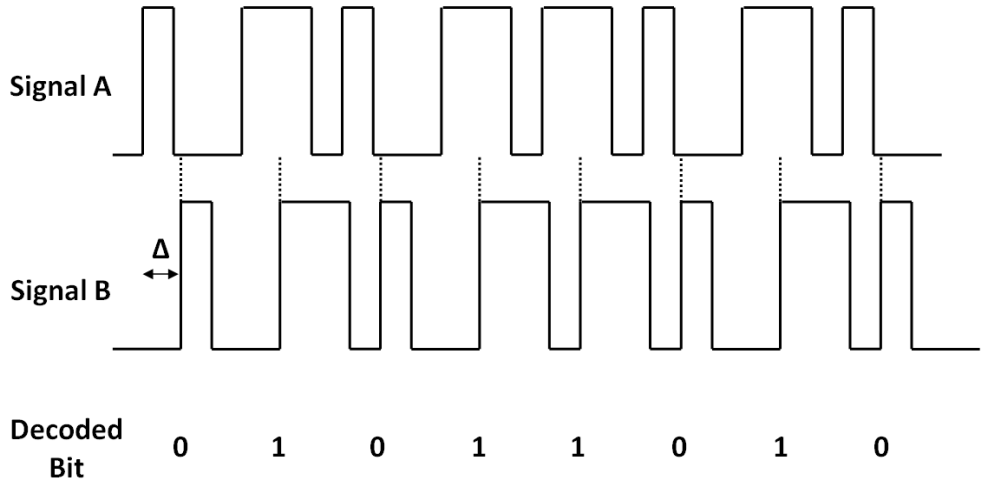

Figure 3.4: PWC Decoding Scheme

3.1.4 Data-Driven Decoder Design

This section introduces a data-driven decoder design to perform the PWC decoding. The term 'data-driven' is used to indicate that the decoder is solely driven by the input data and, hence, eliminates the need for any typical external clock driven mechanisms.

To implement the data-driven decoder as a digital circuit, the required components are a D flip-flop and a delay buffer. The demodulated encoded data stream (Signal A in Figure 3.4) is connected to the input terminal of a traditional D Flip-Flop, and the same encoded data

stream delayed by time Δ (Signal B in Figure 3.4) is connected to the clock input. This is shown in Figure 3.5. It is clear that the input encoded data is sampled at every rising edge of the delayed version of the same encoded data in order to distinguish '1' and '0'. The incoming encoded data is decoded by data driven element shown in Figure 3.5 and corresponding decoded data is stored in a shift register following the decoding. This is a data driven mechanism that eliminates use of high frequency clock signals to decode the incoming data stream largely reduces the dynamic power consumption [56], [57].

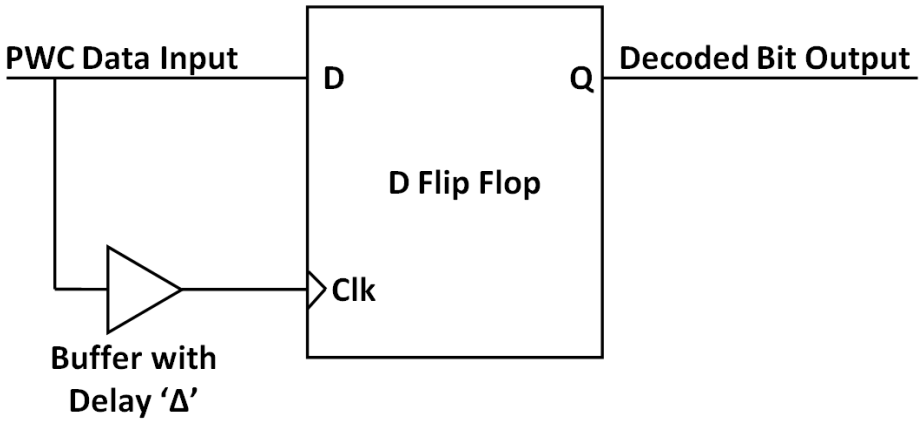

Figure 3.5: Data-Driven Decoding Element

A potential application for the data driven decoding scheme is in Gen-2 RFID systems [54]. These RFID systems currently use a variable pulse interval encoding to represent symbol-0 and symbol-1 [54]. The proposed decoding circuit will reduce the power consumed at the receiver while the encoder design remains unchanged at the transmitter.

The proposed decoding scheme illustrated in Figure 3.4 uses a self-clocking mechanism eliminating the use of an external clock [56]. This scheme basically uses delayed input data to

drive the decoder components such as a shift register, etc. In other words, the delayed input data acts as a clock to trigger components of the decoder.

The block diagram for the proposed data-driven symbol decoder is shown in Figure 3.6. The decoder architecture consists of a shift register, comparator and a delay block. The delay in hardware generally translates to a buffer element. A buffer element is typically built using an even number of inverters. The proposed data-driven architecture eliminates the use of high frequency driven counters and a high frequency oscillator, which are typically used in the conventional decoder as shown in Figure 3.2.

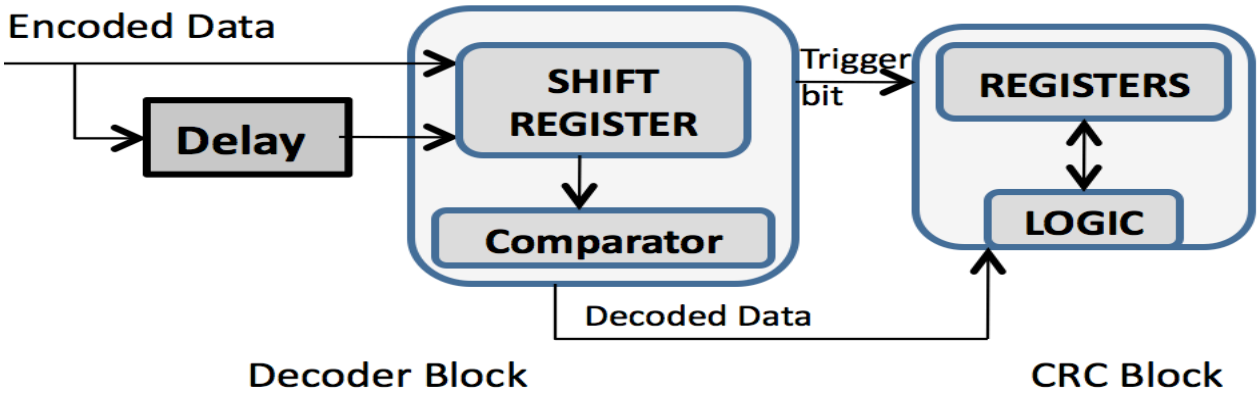

Figure 3.6: Data-Driven Decoder-CRC Unit

The simulation results have shown that the data-driven decoder consumes considerably less power when compared to the conventional decoder design [56]. These dynamic power consumption simulation results of the data-driven decoder and the conventional decoder were 58nW and 9195nW respectively. The data-driven decoder forms the major part of the REU

frontend. The detailed implementation and the design methodology of the decoder as part of the REU frontend are presented in Chapter 4.

3.1.5 REU Frontend Architecture

Main Features of the Frontend Design are as follows:

1) Data-Driven Decoder-CRC-16 block forms the REU frontend design. This block uses the low power self-clocking mechanism to decode the incoming instruction and check for validity of the received instruction.
2) The REU design supports two types of instructions based on 8051 MISA:
    a. 25-bit Input Frame: LENGTH (1-bit)-OPCODE (8-bit)-CRC (16-bit) and
    b. 33-bit Input Frame: LENGTH (1-bit)-OPCODE (8-bit)-DATA (8-bit)-CRC (16-bit)
3) Supports a counter-less Variable-Length Instruction Identification algorithm implementation.

The data-driven decoder-CRC implementation as shown in Figure 3.7 consists of a shift register of bit width ‘n’ that is used for decoding and storing the decoded input data stream. Here, ‘n’ is a function of the data width and the CRC length used for the input data stream. The encoded data are sampled at the rising edge of the delayed data in order to generate decoded bits, which are stored in the shift register and correspondingly used in the CRC check.

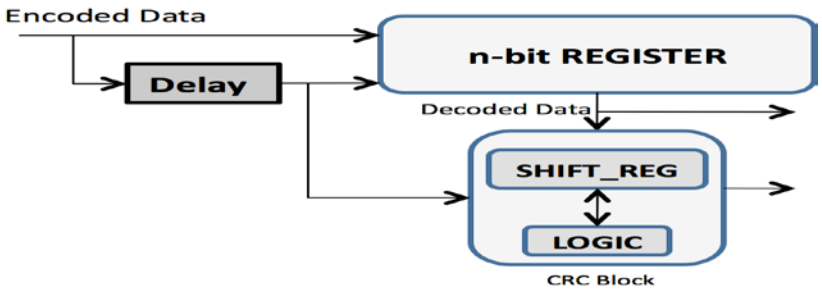

Figure 3.7: REU Frontend Block Diagram

Counters typically are used to keep track of the number of shifts especially in designs using shift registers. Counters in RFID symbol decoders are also used to count the number of clock pulses, which provides the basis for accurately decoding the incoming encoded data. Elimination of the counter hardware for low power needs alternate mechanisms to identify whenever the shift register is full. In the proposed architecture, an additional single-bit is added to the register and is the only additional hardware required to indicate that all the decode data have been successfully loaded into the register and is ready to be used for the next operation. Traditionally, all the flip-flops (registers) in a shift register are reset to '0'. In the proposed n-bit shift register design; the 0^{th} register bit is reset to bit '1', and all the remaining bits are set to '0'(assuming left shift). When the content of the extra register is bit '1', then the n-bit shift register is considered full. The only modifications needed to keep track of decoded data are: an extra 1-bit flip-flop (register) and the reset sequence to initialize the n-bit shift register. This forms the n-bit register in the decoder design.

A pseudo-code is used as an example to differentiate the (25-bit and 33-bit) widths of the incoming instructions at the Frontend an algorithm along with an illustration of the above-mentioned decoding process. Once the shift register indicates it is full, the decoded data are

checked for correct message transmission to the intended receiver. For variable instruction length inputs, the following algorithm is introduced.

Counter-less Variable-Length Instruction Identification Pseudo-code

Reset values:

reg(33 downto 0) <= "0000000000000000000000000000000001"; // reset register

len_detect16 and len_detect8 <= ‘0’; //reset flags

On the rising edge of delayed data (data_clk), each decoded bit is stored in register “reg”. The ‘reg’ width is 34 (reg(33 downto 0)) at it included the storage space for the 33-bit instruction and extra bit. After decoding of every bit, the following pseudo-code is executed:

```
if (rising_edge of data_clk) then                          //check trigger edge
        if (len_detect8 = '0' and len_detect16 = '0') then  // check flag status
                    if (reg (1 downto 0) = “11”) then       // check register start bits
                            len_detect16 <= '1';            // set flag to identify 16-bit
                    elsif (reg(1 downto 0) = “10”) then     // check register start bits
                            len_detect8<= '1';              // set flag to identify 8-bit
                    end if;
        end if;
end if;
```

The “len_detect8” and “len_detect16” are flags which are set on the first occurrence of “10” and “11” in the decoder register that stores the decoded input bit stream respectively. These flags are used to differentiate the two instruction lengths. The first bit in “10” represents the reset value of decoder register, reg(0), and the second bit ‘0’ indicates the opcode instruction length (25-bit). Similarly, the first bit in “11” represents the reset value of the decoder register reg(0) and the second bit ‘1’ indicates the opcode-data instruction length (33-bit). Once the length of the instruction is known, the different fields in the input instruction can be easily identified and processed accordingly. Only one of the two flags (either len_detect8 or len_detect16) is always high during every instruction execution process.

In a traditional CRC hardware implementation, a simple shift register in combination with XOR logic performs serial CRC computations on an input serial data stream. The typical CRC design computation flow is shown in Figure 3.8 (a). This clocked decoder-CRC block is considered part of the frontend of an RFID passive tag and is known to consume a significant amount of power [55], [56].

Typically, a combinational design logic implementation of a CRC consists of simple registers (Flip-Flops) and XOR gates. The final CRC result is computed based on the initial value (seed), decoded data and the intermediate CRC values. The decoded data and the appended CRC are stored in the n-bit register; the initial CRC value is stored in a seed register that is also used to store intermediate CRC values. In the context of a CRC design block implemented as a combinational logic in the proposed architecture, the CRC block is triggered

after the completion of the decoding process. The combinational CRC design computation flow for the data-driven decoder is shown in Figure 3.8 (a).

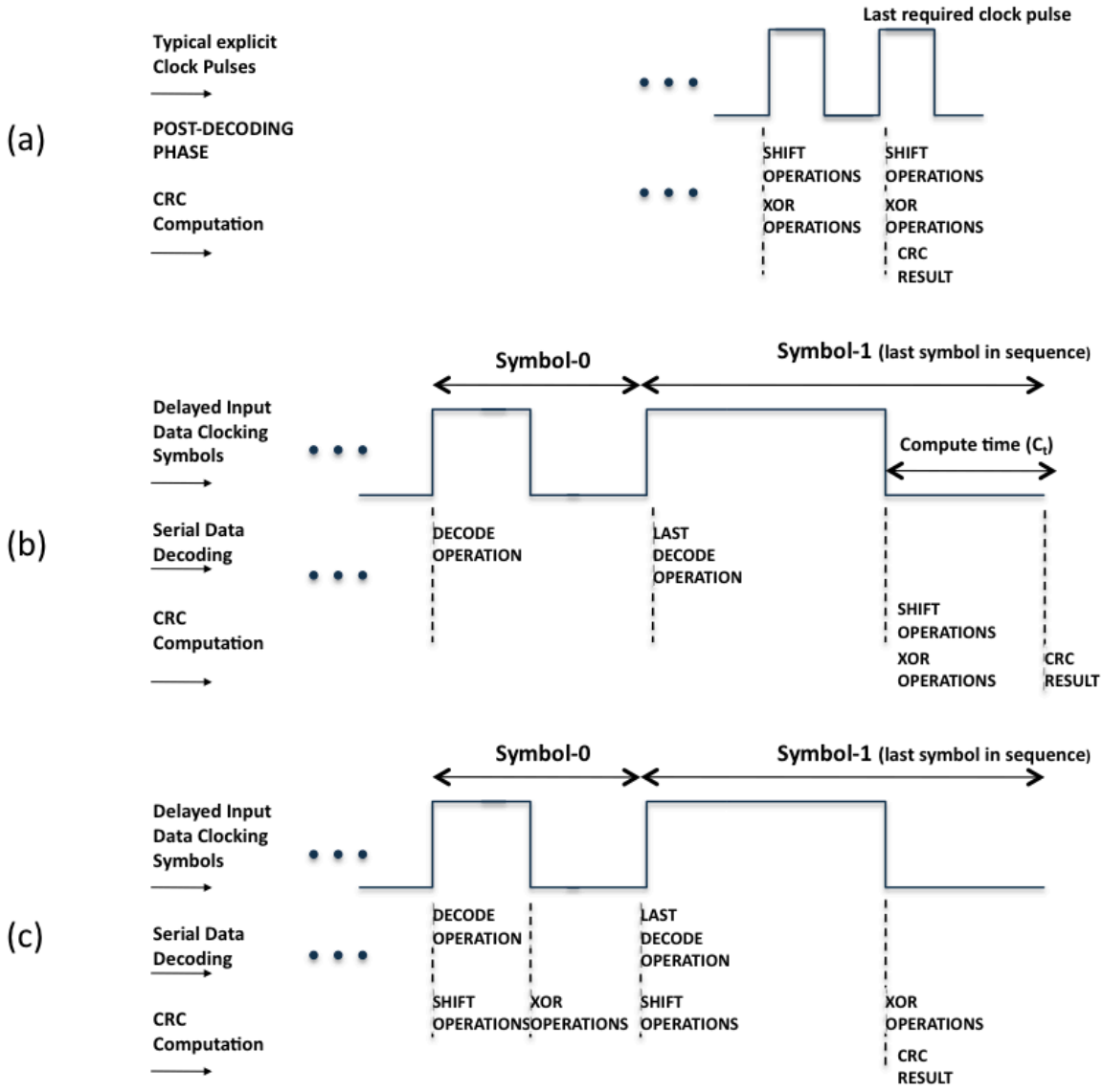

Figure 3.8: Design Computation Flow (a) Traditional clock-driven CRC (b) Data-Driven Decoder-Combinational CRC [56] (c) Data-Driven Decoder-CRC [57]

For the RFID clocked tag decoder and the data-driven decoder, the CRC computation for error detection check is performed only after the demodulated data are decoded and

available for use. The clock-driven process requires extra clock cycles to drive the CRC circuit in addition to the cycles used for decoding each bit. Implementing the CRC design as a combinational block increases the complexity of the design as 'n' increases. Computing CRC after data decoding further slows down the processing speed of the frontend block of the target wireless device such as an RFID passive tag.

The novelty of the proposed data-driven CRC implementation is the ability to take advantage of specific encoding properties of the input sequence in order to realize a self-clocking CRC at the passive receiver side for low power applications. The proposed frontend design consists of an n-bit register for decoding and storing the incoming data and a typical 16-bit register for performing CRC computation on the decoded bits. A data-driven implementation uses the delayed input encoded data as a clocking signal in order to sample the input data [56].

At every rising edge (signal going high) of the delayed encoded signal (also termed as delayed input data clocking symbols as in Figure 3.8 (b) and (c)), the encoded symbol is sampled, decoded and stored in a 1-bit flip-flop (register) as shown in Figure 3.7. At the same rising edge, typical CRC-CCITT fixed shift operations are performed on the 16-bit seed CRC register. On the falling edge (signal going low) of the same delayed encoded signal, the basic set of typical XOR operations are performed on the stored decoded bit obtained at the rising edge of the same symbol and/or intermediate CRC operations. This final computed CRC value is updated to the CRC output register. The proposed data-driven CRC computation flow has been illustrated in Figure 3.8 (c) [57].

A high-level input-output pin diagram of the REU frontend is shown in Figure 3.9. The decoded input instruction frame that is generated by the frontend acts as an input block to the REU core unit. A potential acceptable decoded "*data_in*" input frame to the frontend consists of two sizes: *len*(1-bit)-*opcode*(8-bit)-*CRC*(16-bit) and *len*(1-bit)-*opcode*(8-bit)-*src_data*(8-bit)-*CRC*(16-bit) supported by the REU design. The *len* (bit) field is used to differentiate the two 25-bit and the 33-bit frames. The *opcode* field represents the 8-bit 8051-instruction opcode format and the *CRC* represents the 16-bit code used to detect transmission errors in the received 8-bit *opcode* and/or *src_data*. The *src_data* field represents the 8-bit 8051-source data format. A typical CRC-16 polynomial of the form is $g(x)= x^{16}+x^{12}+x^5+1$ is part of the frontend architecture. The 'ctr' is set by the frontend block on receiving a valid input frame and is used as trigger for the sequence of events that must take place in order to execute a specific instruction on the REU. The detailed description of the input-output signals used in Figure 3.9 is presented in Table 3.1. The opcode is an 8-bit 8051-instruction opcode that includes a source register and the destination register [71]. The "src_data" is an 8-bit data that is generally part of the 16-bit 8051-instructions. The 8051 MISA based REU core supports a set of the 8-bit and 16-bit 8051-instructions that will be explained in detail in the following section.

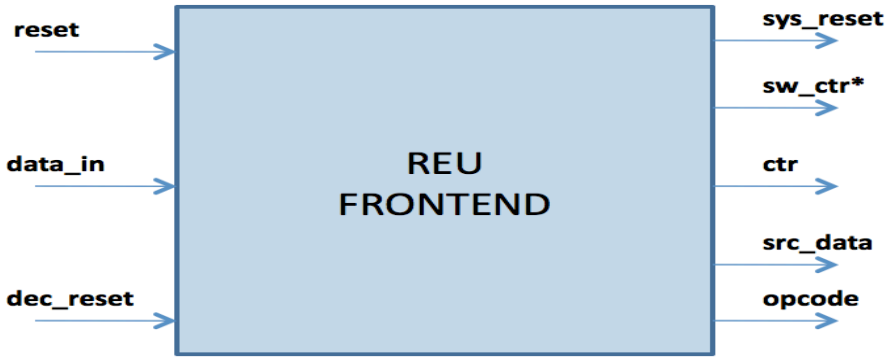

Figure 3.9: REU Frontend Pin Diagram

Table 3.1: REU Frontend Input-Output Signal Descriptions

Signal	Bit-Width	Descriptions	Signal	Bit-Width	Descriptions
Reset	1	Used to reset the system for the start of a new program	*Dec_reset*	1	Used to reset necessary signals of the Front end block.
Opcode	8	Instruction opcode (part of the received frame sequence)	*ctr*	1	Write signal that enables writing data to the 8-register bank (temporary storage (set/reset by controller)
data_in	8	Encoded data that is either a Part of the 16-bit Instruction data (part of the received frame sequence)	*Sw_ctr**	1	Used as the set/reset the frontend switch (*if switch used)
sys_reset	1	Used to reset the system for the start of a new program.	*Src_data*	8	Input data (data_in) to ALU

The simulation, verification and implementation details of the REU frontend will be presented in Chapter 4.

3.2 REU CORE DESIGN

The passive REU is based on a reduced 8051 Instruction Set Architecture (ISA). The choice was based on providing a maximum 8051 flexibility using the default 8051 register bank. This reduced 8051 ISA forms the 8051 MISA that will be described in the following sections.

3.2.1 8051-MISA for REU

The active block or in other words the interrogator will transmit program instructions to the REU that executes these instructions and returns the results back to the interrogator. The REU, for example, will have the capability to perform simple functions like OR, XOR, AND, ADD, etc. that will be compatible with the 8051. This interrogator and the REU together form a complete processor.

8051 is an 8-bit microcontroller that includes an instruction set of 255 operation codes as shown in Table 3.2 [71]. The branch, comparison, load, and store instructions will be implemented on the interrogator side rather than on the REU side. The amount of temporary storage and the ALU capabilities of the REU will be chosen to maintain low power requirements. The REU will be programmed to execute a basic set of 8051-instructions that largely includes arithmetic (ADD, ADDC, SUBB, INC, DEC), Logical (ANL, ORL, XRL, CLR, CPL, RL, RLC, RR, RRC, SWAP), data transfer (MOV, XCH) and Boolean manipulation (CLR, SETB, CPL) instructions. The most demanding instructions like the divide (DIV), multiply (MUL) and decimal adjust (DA) will not been included in the MISA keeping the REU as minimal as possible, but form a part of the interrogator's instructions set. The MISA can be further enhanced based on a case-by-case requirement for any chosen target application during the REU design process.

Table 3.2 represents the 8051 instruction set by opcode. The instructions highlighted in **bold** represent all 116 instructions that form the MISA instruction set of the REU core design.

Only the instructions highlighted in ***bold-italic*** represent the instructions that are part of the 8051 instruction ISA.

Table 3.2: REU-8051 Instruction Subset (MISA)

	0x00	**0x01**	**0x02**	**0x03**	**0x04**	**0x05**	**0x06**	**0x07**	**0x08**	**0x09**	**0x0a**	**0x0b**	**0x0c**	**0x0d**	**0x0e**	**0x0f**
0x00	**NOP**	AJMP	LJMP	***RR***	***INC***	INC	INC	INC	***INC***	***INC***	***INC***	***INC***	***INC***	***INC***	***INC***	***INC***
0x10	JBC	ACALL	LCALL	***RRC***	***DEC***	DEC	DEC	DEC	***DEC***	***DEC***	***DEC***	***DEC***	***DEC***	***DEC***	***DEC***	***DEC***
0x20	JB	AJMP	RET	***RL***	***ADD***	ADD	ADD	ADD	***ADD***	***ADD***	***ADD***	***ADD***	***ADD***	***ADD***	***ADD***	***ADD***
0x30	JNB	ACALL	RETI	***RLC***	***ADDC***	ADDC	ADDC	ADDC	***ADDC***	***ADDC***	***ADDC***	***ADDC***	***ADDC***	***ADDC***	***ADDC***	***ADDC***
0x40	JC	AJMP	ORL	ORL	***ORL***	ORL	ORL	ORL	***ORL***	***ORL***	***ORL***	***ORL***	***ORL***	***ORL***	***ORL***	***ORL***
0x50	JNC	ACALL	ANL	ANL	***ANL***	ANL	ANL	ANL	***ANL***	***ANL***	***ANL***	***ANL***	***ANL***	***ANL***	***ANL***	***ANL***
0x60	JZ	AJMP	XRL	XRL	***XRL***	XRL	XRL	XRL	***XRL***	***XRL***	***XRL***	***XRL***	***XRL***	***XRL***	***XRL***	***XRL***
0x70	JNZ	ACALL	ORL	JMP	***MOV***	MOV	MOV	MOV	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***
0x80	SJMP	AJMP	ANL	MOVC	DIV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0x90	MOV	ACALL	MOV	MOVC	***SUBB***	SUBB	SUBB	SUBB	***SUBB***	***SUBB***	***SUBB***	***SUBB***	***SUBB***	***SUBB***	***SUBB***	***SUBB***
0xa0	ORL	AJMP	MOV	INC	MUL	UNDEF	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0xb0	ANL	ACALL	CPL	***CPL***	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE
0xc0	PUSH	AJMP	CLR	***CLR***	***SWAP***	XCH	XCH	XCH	***XCH***	***XCH***	***XCH***	***XCH***	***XCH***	***XCH***	***XCH***	***XCH***
0xd0	POP	ACALL	SETB	***SETB***	DA	DJNZ	XCHD	XCHD	DJNZ	DJNZ	DJNZ	DJNZ	DJNZ	DJNZ	DJNZ	DJNZ
0xe0	MOVX	AJMP	MOVX	MOVX	***CLR***	MOV	MOV	MOV	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***
0xf0	MOVX	ACALL	**MOVX**	MOVX	***CPL***	MOV	MOV	MOV	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***	***MOV***

The remaining two instructions highlighted only in **bold** form the set of instructions that have been modified to suit the REU requirements. A modified instruction with respect to its common 8051 functionality is the MOVX instruction (MOVX @R_i, A). The 8-bit instruction opcode used for this MOVX instruction is “11110010”. The functionality of the instruction was modified to suit the target REU core design. Upon the execution of this instruction, data

available in the accumulator register will be transferred to a destination register. The destination register is used to hold data that is transmitted out of the REU based on the request from the interrogator. The other modified instruction is the NOP that can be used as an external reset sent by the interrogator to the REU in order to clear up all the data from the previous operations from the registers and signals.

For instance, an ADD operation R3 =#R1+R2, where Ri denotes a register, i#=#0,1,2,.. The interrogator sends out the R1 and R2 values to load and store it in the temporary storage on the REU. Then, the ADD operation is performed by the REU and the computed result is sent back to the interrogator on request. The interrogator will contain main memory that acts as the major storage area for large volumes of data items. This is the same technique used to power passive RFID tags. Thus such tags do not need periodic battery replacements [35]. This makes such tags suitable for embedded tag applications, especially where long-term monitoring is not economical.

Table 3.3 contains the notes for data addressing mnemonics for the instruction sets described in 3.4.

Table 3.3: REU-8051 Data Mnemonics

R_n	Working registers (R_0-R_7)
#data	8-bit constant embedded in instruction
A	Accumulator

The choice of 116 instructions is based on providing a maximum 8051 flexibility using the 8051 register bank (R_0-R_7 /A) register set. 3.4 represents the selected MISA of 116

instructions for the REU along with the corresponding description for each instruction. Thus reducing the selected instructions to less than half of the 256 instructions usually supported by an 8051. Using the MISA significantly reduces the power consumption of the system when compared to systems using the entire 8051-ISA. The 8051-MISA is used for both the asynchronous and the clocked versions of the REU core design implementations.

Table 3.4: MISA 8051 Instructions

Instruction	Type	Description	Bytes
ADD A, Rn	Arithmetic	Add register to accumulator (A)	1 Byte
ADD A, #data	Arithmetic	Add immediate data to A	2 Bytes
ADDC A, Rn	Arithmetic	Add register to A with carry flag	1 Byte
ADDC A, #data	Arithmetic	Add immediate data to A with carry flag	2 Bytes
SUBB A, Rn	Arithmetic	Subtract register from A with borrow	1 Byte
SUBB A, #data	Arithmetic	Subtract immediate data from A with borrow	2 Bytes
INC A	Arithmetic	Increment A	1 Byte
INC Rn	Arithmetic	Increment register	1 Byte
DEC A	Arithmetic	Decrement A	1 Byte
DEC Rn	Arithmetic	Decrement register	1 Byte
ANL A, Rn	Logic	AND register to A	1 Byte
ANL A, #data	Logic	AND immediate data to A	2 Bytes
ORL A, Rn	Logic	OR register to A	1 Byte
ORL A, #data	Logic	OR immediate data to A	2 Bytes
XRL A, Rn	Logic	Exclusive OR register to A	1 Byte
XRL A, #data	Logic	Exclusive OR immediate data to A	2 Bytes

Table 3.4 (continued)

CLR A	Logic	Clear A	1 Byte
CPL A	Logic	Complement A	1 Byte
RL A	Logic	Rotate A left	1 Byte
RLC A	Logic	Rotate A left through carry	1 Byte
RR A	Logic	Rotate A right	1 Byte
RRC A	Logic	Rotate A right through carry	1 Byte
SWAP A	Logic	Swap nibbles within A	1 Byte
MOV A, Rn	Data Transfer	Move register to accumulator (A)	1 Byte
MOV A, #data	Data Transfer	Move immediate data to A	2 Bytes
MOV Rn, A	Data Transfer	Move A to register	1 Byte
MOV Rn, #data	Data Transfer	Move immediate data to Rn	2 Bytes
XCH A, Rn	Data Transfer	Exchange register with A	1 Byte
NOP	Program	No operation	1 Byte
CLR C	Boolean Manipulation	Clear carry flag	1 Byte
SETB C	Boolean Manipulation	Set carry flag	1 Byte
MOV @R0, A	Data Transfer	Move A to destination transfer register	1 Byte

3.2.2 CLOCKED REU CORE

The clocked core design of the proposed REU is presented in this section.

3.2.2.1 Architecture A high-level view of the architecture of the clocked REU core unit is shown in Figure 3.10. The clocked REU core architecture mainly consists of three blocks, namely, a controller, ALU and register file. The decoded input instruction frame that is generated from a frontend block acts as an input to REU core unit as shown in Figure 3.10. The

opcode is an 8-bit 8051-instruction opcode that includes a source register and the destination register. The 116-instruction built-in REU core supports dual length 8051-instructions. The "data_in" input port accepts 8-bit data that are part of the 16-bit 8051-instructions.

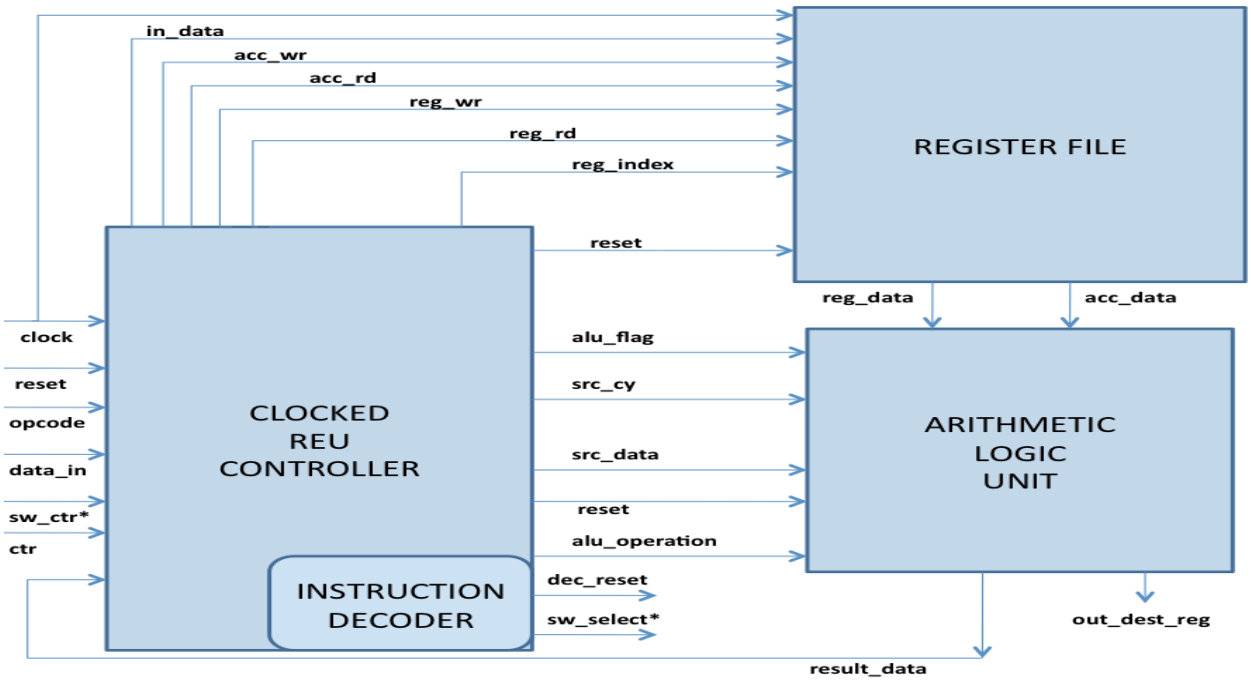

Figure 3.10: High-level Clocked REU Core Architecture

The REU core block is the major block that is clocked in order to compute the corresponding execution of an instruction. The ALU unit is basically responsible for arithmetic and logic operations on 8-bit operands and each of which will be implemented as a combinational block. The register file will be implemented as a sequential block that acts as a temporary data memory, which is triggered by the clock signal. The register file consists of nine 8-bit registers that represent the eight working registers (R0-R7) and an accumulator (A). The controller will be modeled behaviorally as a sequential logic block based on a set of states for every decoded instruction.

Each state will be triggered by the rising edge of the received clock signal implying the number of states needed for an instruction equals the number of clock pulses as illustrated in Table 3.5.

Table 3.5: Clocked REU Cycles

Instruction	Clocked REU Cycles	Instruction	Clocked REU Controller Cycles
ADD A, Rn	5	CLR A	4
ADD A, #data	5	CPL A	5
ADDC A, Rn	5	RL A	5
ADDC A, #data	5	RLC A	5
SUBB A, Rn	5	RR A	5
SUBB A, #data	5	RRC A	5
INC A	5	SWAP A	5
INC Rn	5	MOV A, Rn	5
DEC A	5	MOV A, #data	4
DEC Rn	5	MOV Rn, A	5
ANL A, Rn	5	MOV Rn, #data	4
ANL A, #data	5	XCH A, Rn	7
ORL A, Rn	5	NOP	2
ORL A, #data	5	CLR C	2
XRL A, Rn	5	SETB C	2
XRL A, #data	5	MOV @R0, A	4

Under each state, a group of signals is either set or reset corresponding to the received instruction. It should be noted that the ALU computation will be implemented as a combinational logic and executed in one cycle, but the initial and the final set of cycles are essential to set/reset signals of the core and frontend which are necessary at the start/end of every instruction. Table 3.6 represents the description of each set of signals connected internally or externally to/from controller, ALU and the register file. The REU core architecture is shown

in Figure 3.10. It also includes signals that involve the frontend and a switch design, which will be explained in the following sections.

Table 3.6: Clocked REU Intermediate Signal Descriptions

Signal (bit_width)	Description	Signal (bit_width)	Description
Reset (1)	Resets the initial state before the start of execution of a set of operations	*reg_rd (1)*	Read signal that enables reading data from the 8- register bank (register file (set/reset by controller)
Opcode (8)	Instruction opcode (part of the received frame sequence)	*reg_wr (1)*	Write signal that enables writing data to the 8- register bank (register file (set/reset by controller)
data_in (8)	Part of the 16-bit Instruction data (part of the received frame sequence)	*reg_index (3)*	Signal that indicates the address of one of the 8 registers to read/write from the target register.
alu_flag (1)	Triggers any requested ALU operations (set/reset by controller)	*acc_data (8)*	Accumulator data to be read is stored on to this register for an ALU operation when acc_rd is set.
src_cy (1)	Carry bit necessary for ALU computation	*reg_data (1)*	One of the 8-register data to be read is stored on to this register for an ALU operation when reg_rd is set.
src_data (8)	Input data (data_in) to ALU	*Result_data (1)*	Computed ALU result is placed on to this register which later is to be stored into the accumulator or the registers of the register bank based on whether acc_wr or reg_wr is set.
alu_operation (5)	Indicates type of ALU operation	*acc_data (8)*	Input data (data_in) to accumulator
acc_rd (1)	Read signal that enables writing to the accumulator register (set/reset by controller)	*out_dest_reg (8)*	Computed ALU result is placed on to this register.
ctr_bit (1)	Used as an enable signal for clock gating	*dest_ac (1)*	Output auxillary bit register
acc_wr (1)	Write signal that enables writing to the accumulator register (set/reset by controller)	*dest_cy (1)*	Output carry bit register
in_data (8)	Input data to register bank	*dest_ov (1)*	Output overflow bit register
dec_reset (1)	Used to reset necessary signals of the clockless-Front end block.	*sw_ctr* (1)*	Used as the set/reset the frontend switch (*if switch used)
Sw_select (1)*	Used as the select pin of a frontend switch (*if switch used)		

3.2.2.2 Low Power Techniques The following are the main low power techniques used to reduce power for the clocked REU design:

(a) **MISA for REU**

The main power reduction technique is the reduced ISA for the REU implementation. As the program to be executed by the REU is stored in the interrogator (C&M) side, the need for

program memory at the REU is eliminated. There still may be a need for local scratch pad memory at the REU, although the number of bytes is drastically reduced in order to possibly satisfy the power requirements. The REU executes the commands wirelessly issued by the interrogator. The MISA chosen for the REU consists of 116 instructions compatible with the 8051 ISA. The choice of MISA relies on a set of instructions dependent on the nine 8-bit register (R0-R7 and/or accumulator) based operations.

(b) **Programmable Clock Frequency based Wireless Gating**

A clock signal is known to toggle for every cycle of a processor even if the inputs and outputs remain constant. This significantly contributes to the dynamic power consumption of the processor. Hence it is necessary to avoid clock transitions inside any digital block when it becomes idle. Clock gating is a well-known technique used in clocked designs to reduce the dynamic power consumption. This technique allows only necessary portions of the circuitry to switch thus reducing the switching power of the design [21], [46].

Consider a scenario of an Interrogator and a tag setup. Assume that the tag has its core digital processor in the form of the proposed clocked REU. The execution rate of the entire REU circuitry is controlled by the clocking of the interrogator that acts as a control unit via wireless commands. In other words, the interrogator transmits clock signals wirelessly to the REU that steps thru each set of finite states associated with individual received instructions. The concept of wireless clock gating comes from the fact that the interrogator, not only transmits a series of global periodic clock signals, but also has the knowledge of the exact number of clock signals and the corresponding clock frequency required. This allows the REU circuit to switch at a desired low frequency for only a required number of states keeping the dynamic power

consumption in check. The major clock gating logic is on the interrogator side, tracking the number of exact clock signals required for each of the 8051-REU instructions. But once the received instruction is decoded asynchronously by the frontend at the REU side, a control signal is set high after which the clock signals are received by the REU and subsequently REU states are stepped thru accordingly.

The low power advantage is based on three main factors: programmable multi-clock frequency (lowering clock frequency), clock gating, and no dedicated clock generator for the REU. Wireless clocking also provides the flexibility of a programmable clock frequency for the REU that can be tuned, based on the necessity.

3.2.3 ASYNCHRONOUS REU CORE

3.2.3.1 Motivation Asynchronous designs are increasingly becoming an integral part of numerous wireless applications [10], [62], [63], [64], [65] due to their low power advantages. Low power consumption, no clock distribution, fewer global timing issues, no clock skew problems, higher operating speed, etc., are advantages of asynchronous circuits over synchronous circuits. Low power design especially plays an important role in high-performance microprocessors, digital circuits, etc. which use high frequency clocked designs [3]. Power consumption increases as the clock frequency increases. Asynchronous design is largely autonomous and is not governed by any explicit clock. In other words, asynchronous designs do not use any clock circuit and, hence, wait for a specific amount of time or specific signals that indicate completion of an operation before they go on to execute the next operation. These potential advantages provide the necessary motivation for considering an asynchronous design

for the REU core. The distributed architecture concept is based on classic microprocessor ISA's and architecture for low power applications.

3.2.3.2 Architecture A high-level view of the operation of the asynchronous REU core unit is shown in Figure 3.11. The decoded input instruction frame that is output from a frontend decoder block acts as an input to REU core unit. The input frame to the core unit consists of three fields: 'reset' (1-bit), "opcode" (8-bit), and "data_in" (8-bit). The reset bit is used as part of every instruction as this actually initiates the start to every new set of related operations corresponding to a specific task. The 'ctr_bit' is set/reset by the front-end block that is actually used as trigger for the sequence of events that need to take place in order to execute a specific instruction on the REU. The opcode is an 8-bit 8051-instruction opcode that includes a source register and the destination register. The data_in is an 8-bit data that is generally part of the 16-bit 8051-instructions. The 116-instruction built-in REU core currently supports dual length type of 8051-instructions.

The REU core architecture mainly consists of three blocks namely controller, ALU and register file. The decoded input frame is received by the controller and based on the instruction opcode; it performs the expected operation along with setting up the signals necessary to initiate an ALU operation, write or reads to/from register file, etc. The ALU unit is basically responsible for arithmetic and logic operations.

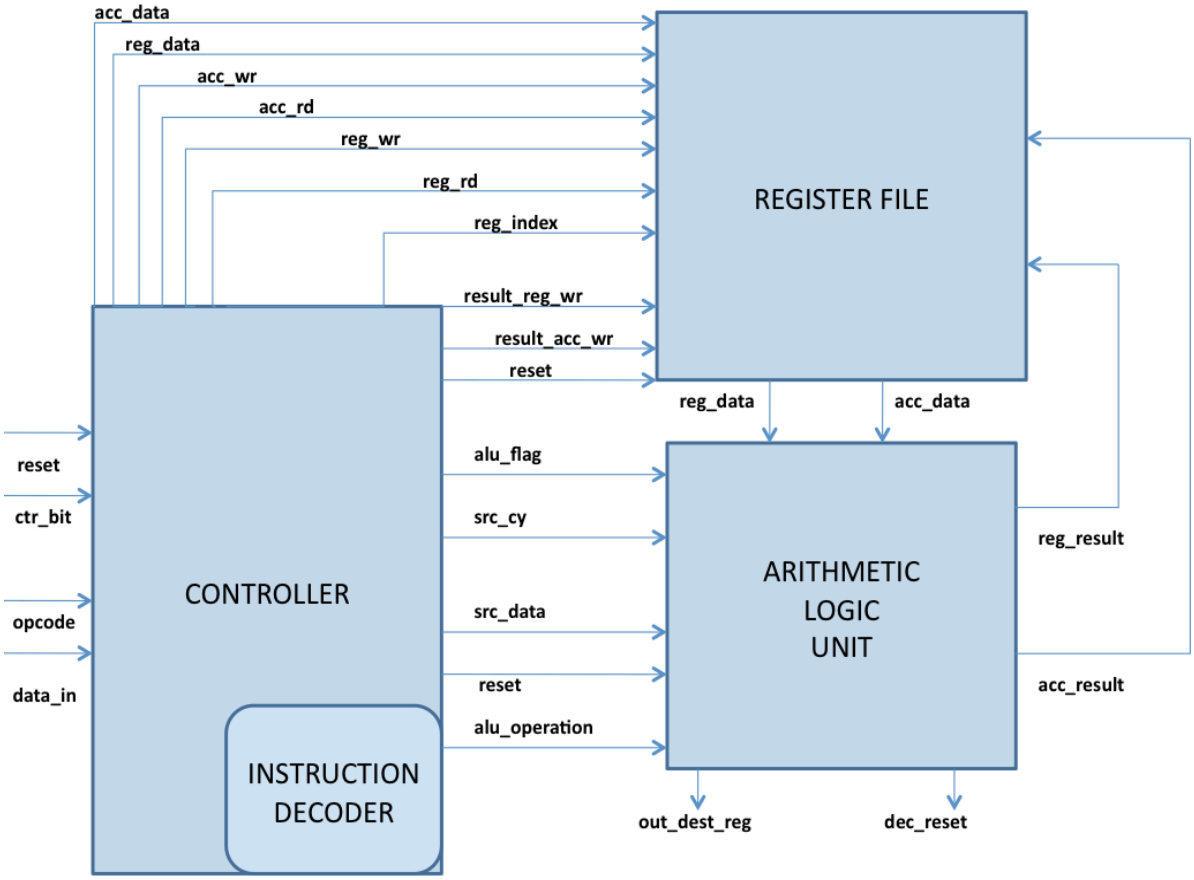

Figure 3.11: High-level Async-REU Core Architecture

The asynchronous REU core design is not governed by a global clock and, hence, uses specific delays in order to activate after the completion of the previous operation. In other words, each block of REU core has an enable signal in order to activate the operation as opposed to clocking each block to finish its respective operation. The 'ctr_bit', register file write signals and 'alu_flag' single bit signals are used as enable signals for the controller, register file, and ALU respectively. Table 3.7 represents the description of each set of signals connected internally or externally to/from controller, ALU and the register file. A top-level

VHDL code of the REU core architecture shown in Figure 3.11 is presented in the Section 1.01(a)(i)Appendix E.

Table 3.7: Asynchronous REU Intermediate Signal Descriptions

Signal (bit_width)	Description	Signal (bit_width)	Description
Reset (1)	Resets the initial state before the start of execution of a set of operations	*reg_rd (1)*	Read signal that enables reading data from the 8- register bank (register file (set/reset by controller)
Opcode (8)	Instruction opcode (part of the received frame sequence)	*reg_wr (1)*	Write signal that enables writing data to the 8- register bank (register file (set/reset by controller)
data_in (8)	Part of the 16-bit Instruction data (part of the received frame sequence)	*reg_index (3)*	Signal that indicates the address of one of the 8 registers to read/write from the target register.
alu_flag (1)	Triggers any requested ALU operations (set/reset by controller)	*acc_data (8)*	Accumulator data to be read is stored on to this register for an ALU operation when acc_rd is set.
src_cy (1)	Carry bit necessary for ALU computation	*reg_data (1)*	One of the 8-register data to be read is stored on to this register for an ALU operation when reg_rd is set.
src_data (8)	Input data (data_in) to ALU	*reg_result (8)*	Computed ALU result is placed on to this register to be stored into the 8-register bank when result_reg_wr is set.
alu_operation (5)	Indicates type of ALU operation	*acc_data (8)*	Input data (data_in) to accumulator
acc_rd (1)	Read signal that enables writing to the accumulator register (set/reset by controller)	*out_dest_reg (8)*	Computed ALU result is placed on to this register.
ctr_bit (1)	Triggers the sequence of operations for execution of an instruction	*dest_ac (1)*	Output auxillary bit register
acc_wr (1)	Write signal that enables writing to the accumulator register (set/reset by controller)	*dest_cy (1)*	Output carry bit register
rg_data (8)	Input data to register bank	*dest_ov (1)*	Output overflow bit register
dec_reset (1)	Used to reset necessary signals of the clockless-Front end block.	*result_reg_wr (1)*	Write signal that enables writing the computed ALU result back to the 8-register bank (set/reset by controller)
result_acc_wr (1)	Write signal that enables writing the computed ALU result back to the accumulator register (set/reset by controller)	*acc_result (8)*	Computed ALU result is placed on to this register to be stored into the accumulator when result_acc_wr is set.

3.2.3.3 Low Power Techniques The following are the main low power techniques used to reduce power for the asynchronous REU design:

(a) **MISA for REU**

The main power reduction technique is the reduced ISA for the REU implementation. As the program to be executed by the REU is stored in the interrogator side, the need for program memory at the REU is eliminated. There still may be a need for local scratch pad memory at the REU, although the number of bytes is drastically reduced in order to possibly satisfy the power requirements. The REU executes the commands wirelessly issued by the interrogator. The MISA chosen for the REU consists of 116 instructions compatible with the 8051 ISA. The choice of MISA relies on a set of instructions dependent on the nine 8-bit register (R0-R7 and/or accumulator) based operations.

(b) **Asynchronous design**

The REU core block will be implemented as delay based logic. REU core architecture is not based on any global clock, but is event driven. In other words, certain critical signals are allowed to occur after a particular time interval. This time interval (delay) is generally based on the time required to complete a previous operation with a valid result. A sample scenario for an ADD operation is considered below:

Figure 3.12 represents a high-level sequence diagram for an ADD operation. If we consider an ADD operation: ADD A, R1 (A = A+R1), where R1 denotes one of the eight (R0-R7) 8-bit 8051 working registers for a selected register bank and A denotes the 8-bit accumulator register. Once the frontend receives the encoded instruction, it is decoded into the

corresponding 8-bit 8051 binary opcode and checked for its validity. On decoding the valid instruction, the opcode is passed onto the REU core for processing. The controller of the core decodes the opcode accordingly and activates all signals corresponding to the add operation. On successful opcode, 'reg_rd' (register read signal), 'acc_rd' (accumulator read signal), 'alu_flag' (begin computation signal) and 'result_acc_wr' (accumulator result write signal) are all set at different times based on processing delay. These delay times are based on the order of occurrence and time taken to finish an operation. The order of events for an ADD operation: R1 and accumulator values are read from the register file; the ALU unit performs addition on the data and finally the computed result is written back into the accumulator. The 'reg_rd' and 'acc_rd' are set as soon as the instruction is decoded, 'alu_flag' is set only after the register and accumulator data are read and available for addition, 'result_acc_wr' is set as soon as add operation is done and ready with the result. All the signals are reset to zero after the result is successfully stored in the accumulator. A detailed timing diagram for these signals is presented in Figure 3.12 where δ_1 represents the time required to read the necessary data from the register file for the operation and δ_2 represents the time required to compute the ALU operation and produce the result, which is then ready to be written back to the accumulator.

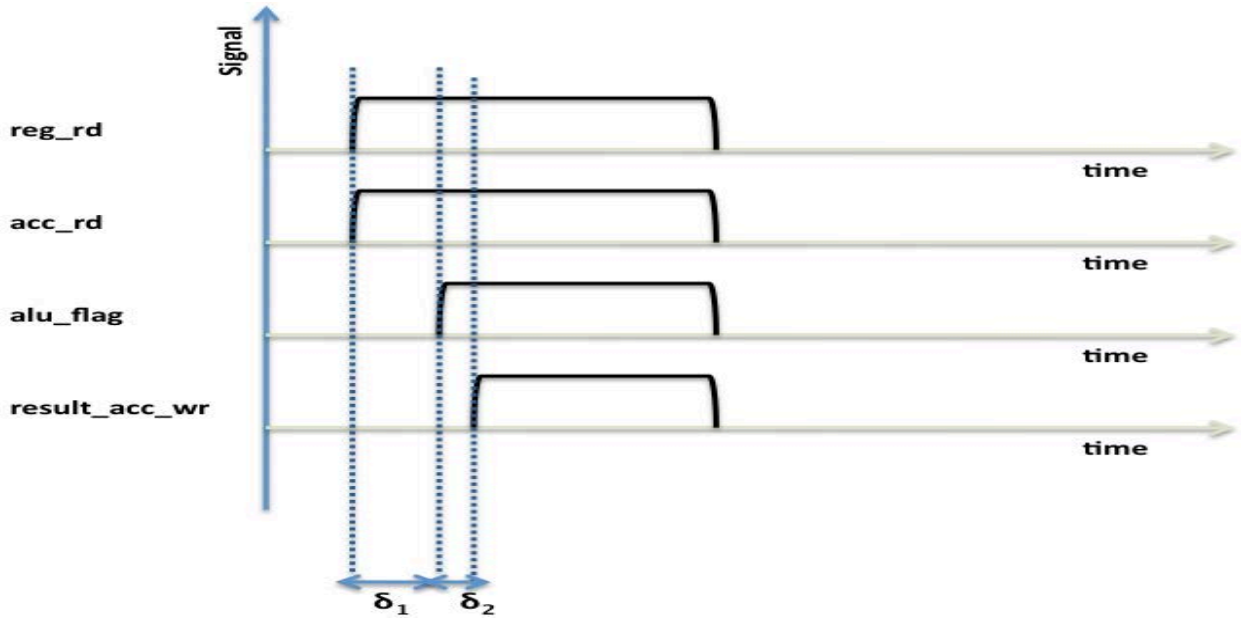

Figure 3.12: Timing scenario for an ADD operation

The main set of control signals delayed by a specific time interval for the REU-Core include ‘reg_rd’, ‘acc_rd’, ‘reg_wr’, ‘acc_wr’, ‘result_reg_wr’, ‘result_acc_wr’, ‘alu_flag’ and ‘dec_reset’. The next section presents the entire REU architecture that integrates the REU core with the frontend for low power applications.

3.3 PROPOSED REU ARCHITECTURES

The data-driven decoder in combination with a data-driven CRC form the frontend block of the REU as described in Section 3.1. The REU core consists of the 8051 compatible ALU unit and a set of temporary storage registers as illustrated in the previous sections. A possible combination of the frontend and the core blocks to form the REU architecture is explained in

this section. Two types of high-level architectures of the proposed REU are shown in Figure 3.13 and Figure 3.14 respectively.

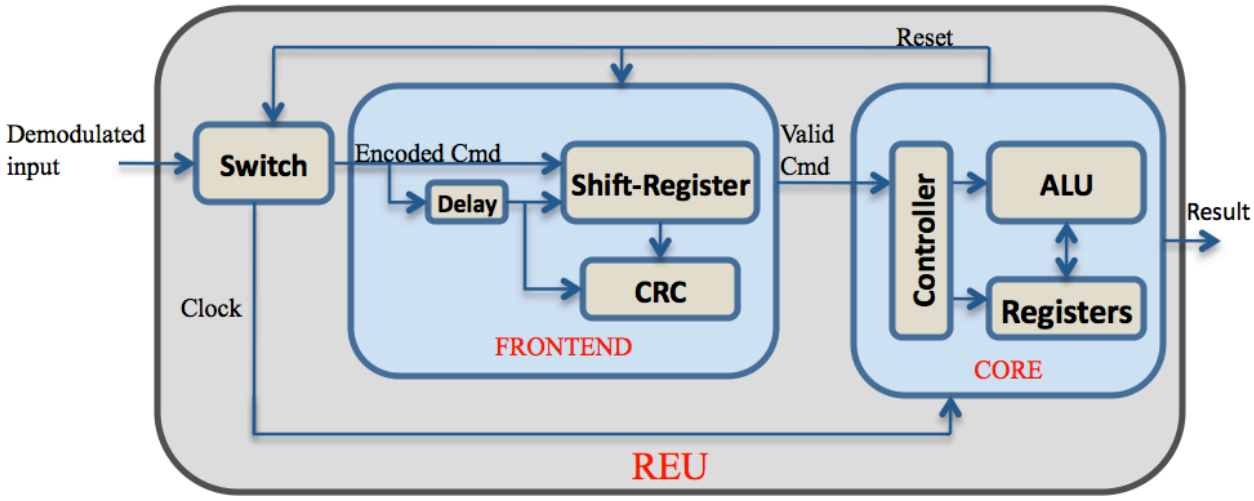

Figure 3.13: Proposed Clocked REU High-level Block Diagram

The clocked REU architecture shown in Figure 3.13 has its execution rate controlled by the wireless clocking of the interrogator, which acts as the control unit through the wireless commands. The demodulated input command is shifted into the frontend registers through a switch. The main functionality of the switch is to accurately channelize the demodulated input (encoded cmd first and clock signals later) according to the frontend and the REU core respectively. Before transmission, the encoding scheme at the interrogator ensures that a valid command and valid CRC are both true only when the full command instruction has been shifted into the frontend registers. This is accomplished by a few additional bits in the command that signals when both command/instruction and CRC are true, to switch the input from the frontend to the clocking core circuit for the logical operation of the REU. On successful execution of the command, the core resets the frontend and switch designs, making it ready to receive its next

command. The asynchronous REU architecture shown in Figure 3.14 has its execution rate controlled by the incoming rate of the wireless command. It has the common frontend block implementation of the clocked REU. The main difference between both the REU designs with respect to implementation is only the core block. The core represented in Figure 3.14 is implemented as an asynchronous design.

Asynchronous designs may be known for low power, but are much harder to design. Such implementations are not based on the global periodic pulses typically known as the clock signal. The implementation and verification of an asynchronous design is a much longer process when compared to the synchronous designs because great care must be taken to ensure timing and data integrity. There are no standard commercial complete design solution tools for asynchronous designs. Hence, the lack of strong support of commercial CAD tools is a major hurdle for synthesis of asynchronous designs.

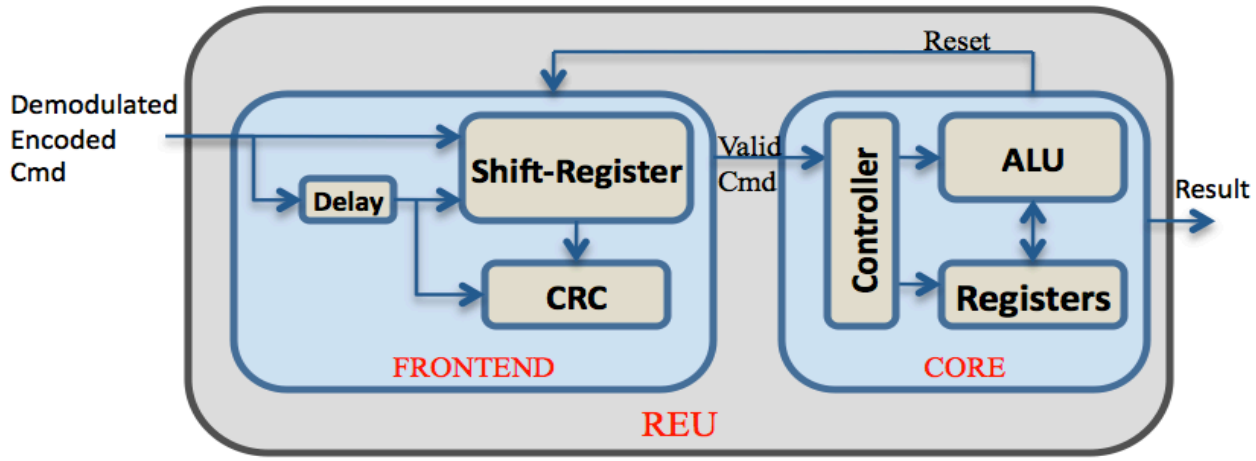

Figure 3.14: Proposed Asynchronous REU High-level Block Diagram

The REU has been designed; synthesized and implemented using standard CAD tool flows. It should also be noted that these tools were solely designed with an intent to test and implement clocked designs. Using these tools to reasonably respond to asynchronous designs is a task in itself. Both the clocked and asynchronous REU core architectures were implemented as digital designs. The implementation process of both the architectures enables a valid comparison between the clocked and asynchronous design performance in aspects of power, speed and area. Considerable time has also been spent to ensure successful implementations allowing a valid comparison. The post-layout level details of the verification and implementation of both the versions of the REU core design and their corresponding power, area and speed results are presented and compared in the next chapter.

4.0 REU DESIGN IMPLEMENTATIONS AND RESULTS

This chapter presents a high-level design flow for clocked and asynchronous REU designs, along with the corresponding post-layout simulation results. Another significant contribution of this research is the high-level design flow of asynchronous REU architectures using synchronous or clocked CAD (computer-aided design) tools. The minimum modifications to the traditional clocked CAD flows necessary for the successful implementation of such asynchronous templates are introduced in the next section.

A standard ASIC flow begins at the specifications for the target design. Once the design specifications are clear, the design is partitioned into logical modules. Each logical module is modeled using an electronic design automation based descriptor language such as VHDL (very-high-speed integrated circuits hardware descriptor language). The design is tested for functional correctness using functional simulation based testbenches.

Lack of strong support of commercial CAD tools is a major hurdle for the synthesis of asynchronous designs. Asynchronous VHDL designs are known to use necessary delay constructs such as wait, delay, etc.; as part of the VHDL implementation due to the absence of a global clock. These VHDL delay constructs are not synthesizable. Standard VHDL compilers

(for example Xilinx ISE, Altera Quartus II, etc.) are not known to synthesize VHDL code that implements an asynchronous design. Based on the Conventional Hardware Descriptor Languages, most asynchronous design methodologies [66], [67], [68] that have been proposed are not very accessible to standard high-level design tools. Current implementation of asynchronous designs involve either use of an entirely new cell library and/or specialized tools. These act as impediments in the adoption of existing clocked CAD flows in the implementation of asynchronous designs. A high-level CAD flow is introduced in this section for the REU designs, especially the asynchronous ones, which requires minimum changes to traditional clocked design flow.

4.1 DESIGN FLOW IMPLEMENTATION USING CLOCKED CAD TOOL FLOWS

The remaining section discusses the proposed design flow of implementing an REU architecture using state of the art clocked CAD tools. This section illustrates different design and verification phases starting from the VHDL design all the way to the design layout, including the CAD flow modifications necessary for the asynchronous templates as well.

Synchronous designs are governed by a global clock for the accurate and timely execution of the functions. Asynchronous design is largely autonomous and is not governed by any explicit clock. In other words, asynchronous designs do not use any clock circuit and, hence, wait for specific signals that indicates completion of an operation before they go on to execute the next instruction.

The REU architectures introduced in Chapter 3 consist of three major components : frontend, clocked core and asynchronous core. Once the design specifications are clear, the design is partitioned into logical modules. Each logical module is modeled using VHDL. Figure 4.1 presents the different modules that depict the major blocks of both the REU architectures shown in Figure 3.13 and Figure 3.14. Each module is first simulated and verified independently. On successful individual verifications, the modules are combined to form the final design and were finally again verified for the overall expected functionality.

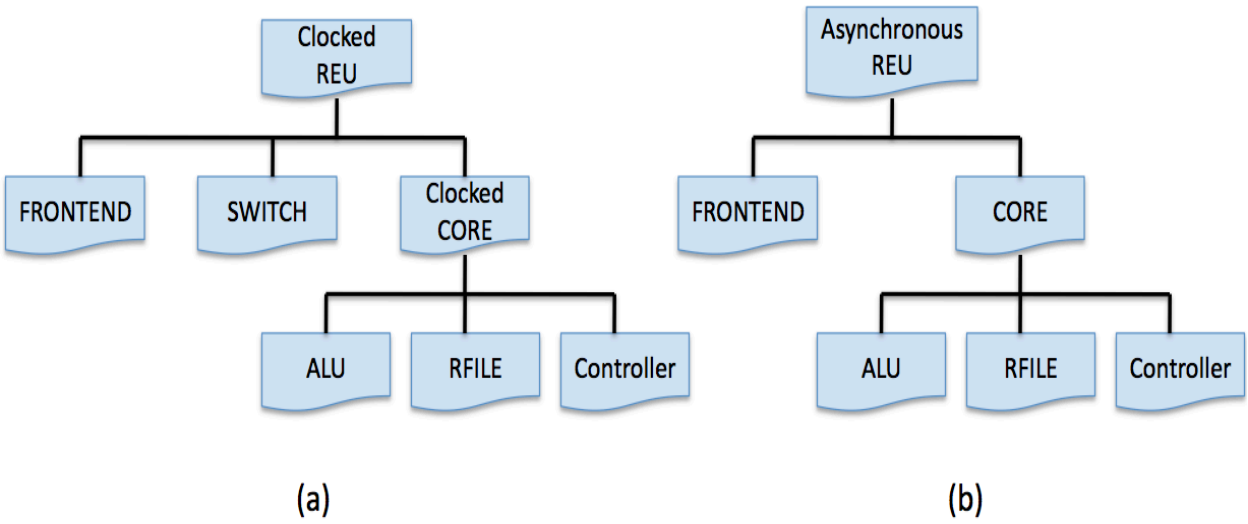

Figure 4.1: Modules (a) Clocked REU (b) Asynchronous REU

The testbenches especially used for the verification of REU core designs typically cover all types of instructions that are part of the 8051-MISA covering operations performed on all the different registers available. The testbenches used for the frontend cover dual length based 8051-MISA encoded instructions. The overall testbench used for the REU, includes performing operations on just received data and data currently stored in these registers, is setup, for

instance, on the lines of an sum-array application. The order of instructions starts with storing different data in the available registers and then performing all various operations such as addition with carry, logic operations, etc. on the existing and the new data. The repetition of instructions is based on various combinations of using the same operation-different register, different operations-same register with different data. The details of the testbenches are described in the later sections.

The REU tool based design flow process along with the verification is illustrated in Figure 4.2. All the design flow modifications as shown in Figure 4.2 are specifically applicable only for asynchronous designs.

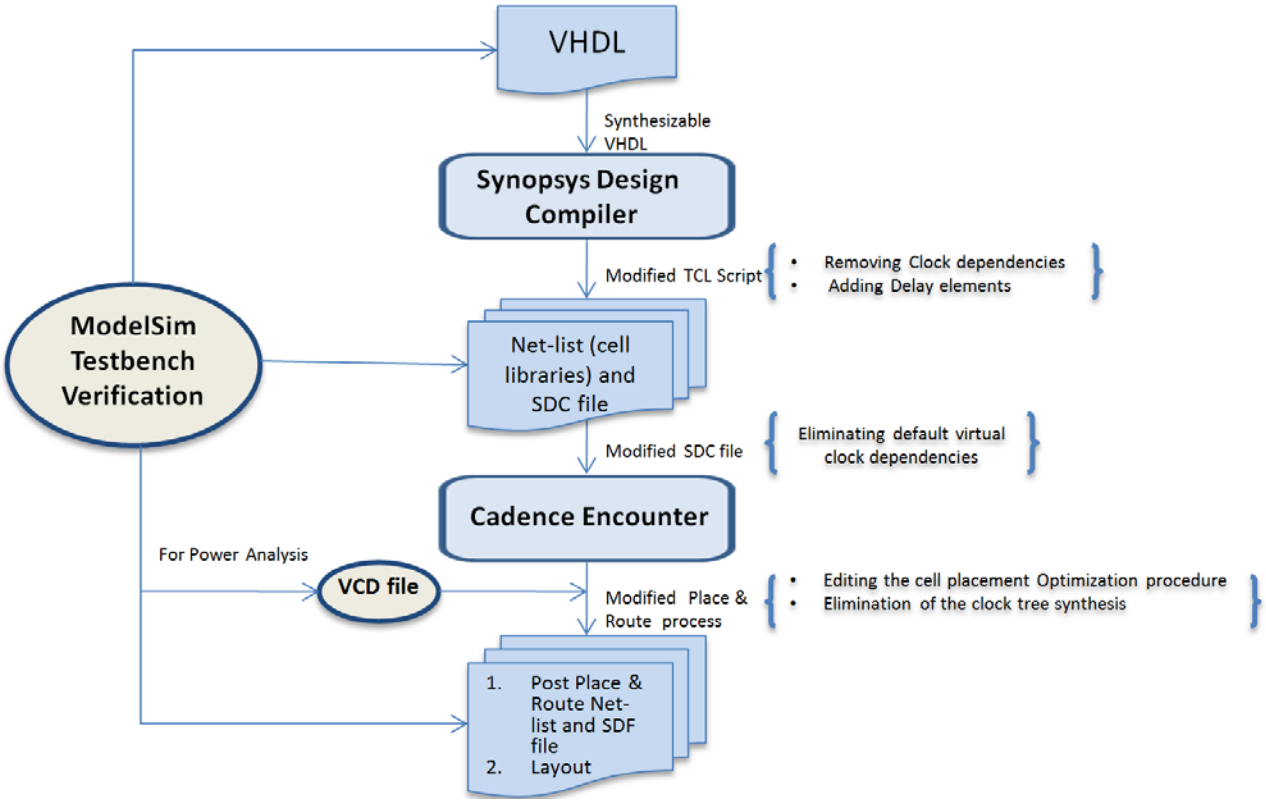

Figure 4.2: High Level Design Flow

The major steps (Figure 4.2) used to implement the REU design are given in the following sections [69]:

4.1.1 Simulate and Verify the VHDL design using *ModelSim*

Mentor Graphic's *ModelSim* tool is typically used to simulate and debug the design for the logic verification. The REU design blocks, which include both the clock driven and asynchronous templates, are modeled using VHDL.

The VHDL based asynchronous blocks of the REU design use delay constructs for exact timing requirements in order to implement functional correctness, in contrast to the clocked design versions. The frontend design uses input data as a clock instead of an external global clock as shown in Figure 4.3. The "data" signal shown in Figure 4.3 acts as the input data port, "data_clk" signal is defined as an internal signal that stores the input data, and "tmp" is realized as a shift register to store the decoded data. Delay constructs embedded in the design are necessary and are not only used in the frontend to successfully decode and store input instructions into elements such as a shift register, but also to delay asynchronous core related specific signals so as to activate them after the completion of a previous operation. The design using VHDL delay constructs are not synthesizable.

```
data_clk <= data;
if (data_clk'event and data_clk='1') then
    tmp <= tmp(2 downto 0)& data;
end if;
```

Figure 4.3: A portion of the sample VHDL code

The purpose of this exercise is to verify the logical operation of the design in simulation. A customized testbench is used to verify the correct functionality of the clocked and the asynchronous based designs in *ModelSim SE 6.*4 on a Linux based platform.

4.1.2 Generate a synthesizable design using *Synopsys Design Compiler*

Given the successful verification of the VHDL file using ModelSim, the next step is to generate a synthesized net-list for the design using the *Synopsys Design Compiler*.

The "dc_shell" command interface provides a script execution environment based on TCL (Tool Command Language). The basic directives of a TCL script include setup environment variables, constraints, basic compilation directives, etc.

In case of clock driven modules, the main parameter for synthesis is the specification of the clock frequency in the TCL script. In case of the asynchronous modules, all the statements that involve the non-synthesizable VHDL delay constructs are identified and removed. It is necessary to identify specific cell names and their corresponding inputs and outputs in the schematic of the design in order to insert the necessary delays.

The major modification in the script for asynchronous modules is to avoid specifying any target clock frequency for synthesis. The other major modification to the TCL script is the inclusion of necessary delays using the TCL based delay commands. The synthesizable delay

commands normally available are ‘set_max_delay’ and ‘set_min delay’, which can be separately inserted into the TCL script during the synthesis process. These commands have options to specify start/end sets of cells and their inputs/outputs pins in the schematic along with a fixed target delay value. Identification of the necessary cells and their corresponding begin/end points in the design schematic is the key to maintaining the timing of the entire system which in turn aids the correct working of the design. Figure 4.4 shows a sample usage of ‘set_min_delay’ command along with the required parameters. These parameters in Figure 4.4 represent a delay of 0.3 ns, origin cell name U11 and its output port Y, destination cell name tmp_reg [] and its corresponding input port CLK.

```
compile -ungroup_all -map_effort medium
check_design

set_min_delay 0.3 -from U11/Y -to {tmp_reg[0]/CLK tmp_reg[1]/CLK tmp_reg[2]/CLK tmp_reg[3]/CLK tmp_reg[4]/CLK}

compile -incremental_mapping -map_effort medium

check_design
redirect ./DC_reports/constraint_violators.rep { report_constraint -all_violators -verbose }
```

Figure 4.4: A portion of a sample TCL script with the delay command

In the case of clock based designs, the above mentioned modification to the TCL script is not necessary. Typically, during the synthesis process for any type of design, *Design Compiler,* after executing the updated TCL script, reads in the synthesizable VHDL file and generates a synthesized cell-level net-list in Verilog according to a standard cell library. The generated Verilog file shares the same I/O ports as the initial VHDL, along with the description of cells and their interconnections. These cells typically consist of basic components such as AND gate, OR gate, D-F/F, etc. Another necessary timing constraint file produced after synthesis is called the SDC (Synopsys Design Constraints) file, which is input during the place

and route process. SDC is a TCL based format constraining file. Generally, clocked CAD flows assume a virtual clock (or use specific clock for clocked designs) for all its purposes, if a clock is not explicitly specified as in the case of asynchronous designs. The SDC file does contain timing constraints related to an assumed virtual clock. Hence the SDC file is modified to remove these virtual clock dependencies before using this file in the process to produce the layout.

The next step is to compile and simulate the generated net-list Verilog file along with the target library using *ModelSim*. This net-list is checked for any delay issues, clock speeds (only in case of synchronous design), or any errors caused due to the misinterpretation of the input design by the synthesis tool. On the successful verification of the net-list using the previous testbench for correct functionality, the layout of the chip is generated.

4.1.3 Generate the layout using *Cadence Encounter*

After successful synthesis of a design, the *Cadence Encounter* tool is used to perform a physical place and route of the previously obtained net-list of standard cells.

The two files obtained after post-synthesis, namely the Verilog net-list file and the modified SDC file, are provided as input to the *Encounter* tool. Next, a sequence of events need to be performed : to import the design, specify the floor plan, power planning, placement, timing check and finally to route the design. In the above sequence of events, it is necessary to skip an important procedure called the “clock tree synthesis” (an integral part of generating

layout for any clocked circuit) for asynchronous based templates. The use of automatic pre-place optimization done during the cell placement process is also restricted in order to have all the delay elements intact in the design. This modification is not included as part of the layout generation of any clock based design.

After verification of the entire design for connectivity and geometry, the final place and route layout is obtained, along with the derived post-place and route net-list file and a generated standard SDF (standard delay format) file. The generated SDF file contains library cell models and related delay information.

The resulting post-place and route net-list are simulated and verified for the expected functional operation using *ModelSim* along with the SDF file. Design of a working asynchronous/clocked circuit chip is complete on the successful post-layout functional verification.

4.1.4 Power estimation with *Cadence Encounter*

There are two options of generating power reports for a design. One option is to generate a basic power report that does not require any specific testbench and the other is a testbench based power report. The basic power report provides a general overall power consumption of the design and the testbench based power report provides power consumption of the design based on the switching activity mentioned in the testbench. During the final leg of the place and route process, under *report* tab there is a power option available with Encounter to produce the basic

power report. The testbench based power report generation involves a switching activity file called the VCD (Value Change Dump) file that is derived from the initial testbench using *ModelSim*. During the final leg of the place and route process, the generated VCD file is input to the power rail analysis option available with Encounter to produce the power report.

4.2 REU POST-LAYOUT SIMULATION RESULTS

4.2.1 Clocked REU Core

The REU core acts as a controller for the entire REU architecture that decodes the command, performs the corresponding ALU operation and generates control signals for the register file. The REU core is currently designed as a finite state machine and the clock is used to trigger each one of the sets of states based on the decoded command/instruction. The individual blocks of the current clocked REU system consist of a register file and a controller. Each individual block is tested for accurate operation by a sample testbench. The testbench that is used includes instructions for moving necessary data to all the eight registers. The post-layout simulation and verification of the clocked REU core design has been successfully completed as shown in Figure 4.5 for a clock frequency 10 MHz for the 8051 instructions. For the detailed testbench used in this simulation is shown in the Section 1.01(a)(i)Appendix B. For all the input, output and signal definitions used in the Figure 4.5 refer to the Chapter 3. The register value in reg5 (01010101) is added to the accumulator value (10101010) to obtain a result of 11111111, which is stored as the first non-zero value in the *result_data_temp* register (highlighted by an arrow marker in Figure 4.5). This is the first ALU instruction executed in the testbench. After the execution of the final

instruction (MOVX) of the testbench, each of the corresponding intermediate multi-bit register values are highlighted in rectangular orange box shown in Figure 4.5. The accumulator data (01010101) from the previous instruction is transferred to the destination register (*des_out*) and is highlighted by an orange-circled marker in Figure 4.5. All the REU instructions have been successfully verified for the expected operation.

The area dimension including the pads of the 8051 core and REU core layout as estimated by Cadence Encounter is 67596 μm^2 and 15748 μm^2 respectively. Figure 4.6 shows the layout of the REU core chip. The core layout area occupied by the clocked REU core is about 7,917 μm^2 (91 x 87).

Figure 4.5: Clocked REU Core Post-Layout Simulation

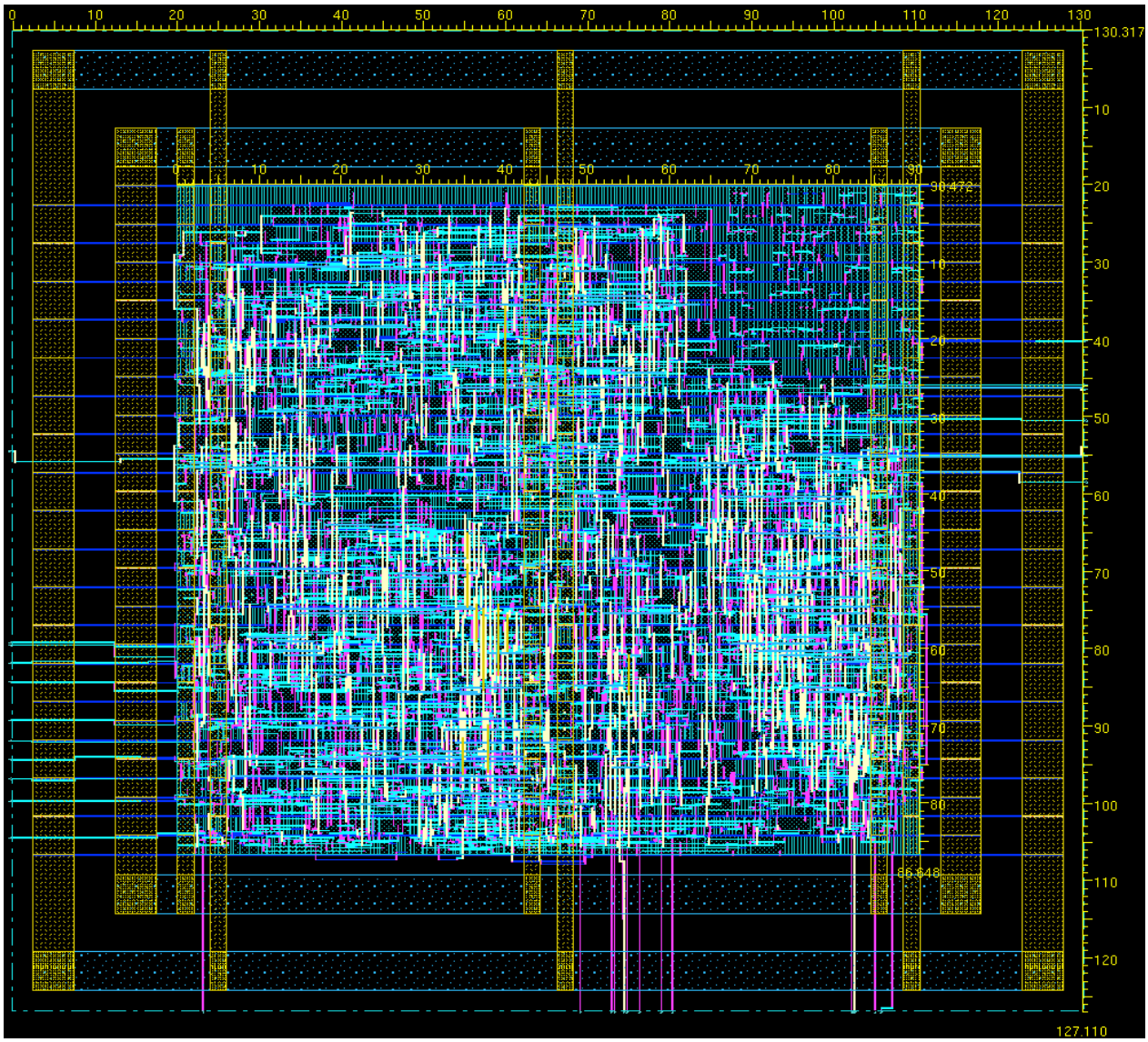

Figure 4.6: Clocked REU Core Layout

4.2.2 Asynchronous REU Core

The asynchronous REU consists of the REU core and frontend as main blocks. The successful post-layout verification of the REU core design has been simulated as shown in Figure 4.7. The clocked core version's testbench is used for the asynchronous core simulation with the same order and frequency except for the clock signal. For all the input, output and signal definitions

used in the Figure 4.7 refer to the Chapter 3. The add operation result (11111111) same value as in the case of clocked version is highlighted by an arrow marker in Figure 4.7. The accumulator data (01010101) from the previous instruction is transferred to the destination register (*dest*) that is the same value as in the case of the clocked version and is highlighted by an orange-circled marker in Figure 4.7. The corresponding intermediate multi-bit register values are highlighted in rectangular orange box shown in Figure 4.7. All the REU core instructions have been successfully verified for the expected operation.

Figure 4.7 represents the simulation results for a testbench consisting of the sample set of instructions of the ones used for testing the clocked REU core design. Figure 4.8 shows the layout of the asynchronous REU core design. The dimension of the core area layout as estimated by Cadence Encounter is 9,024 μm2 (96 x 94).

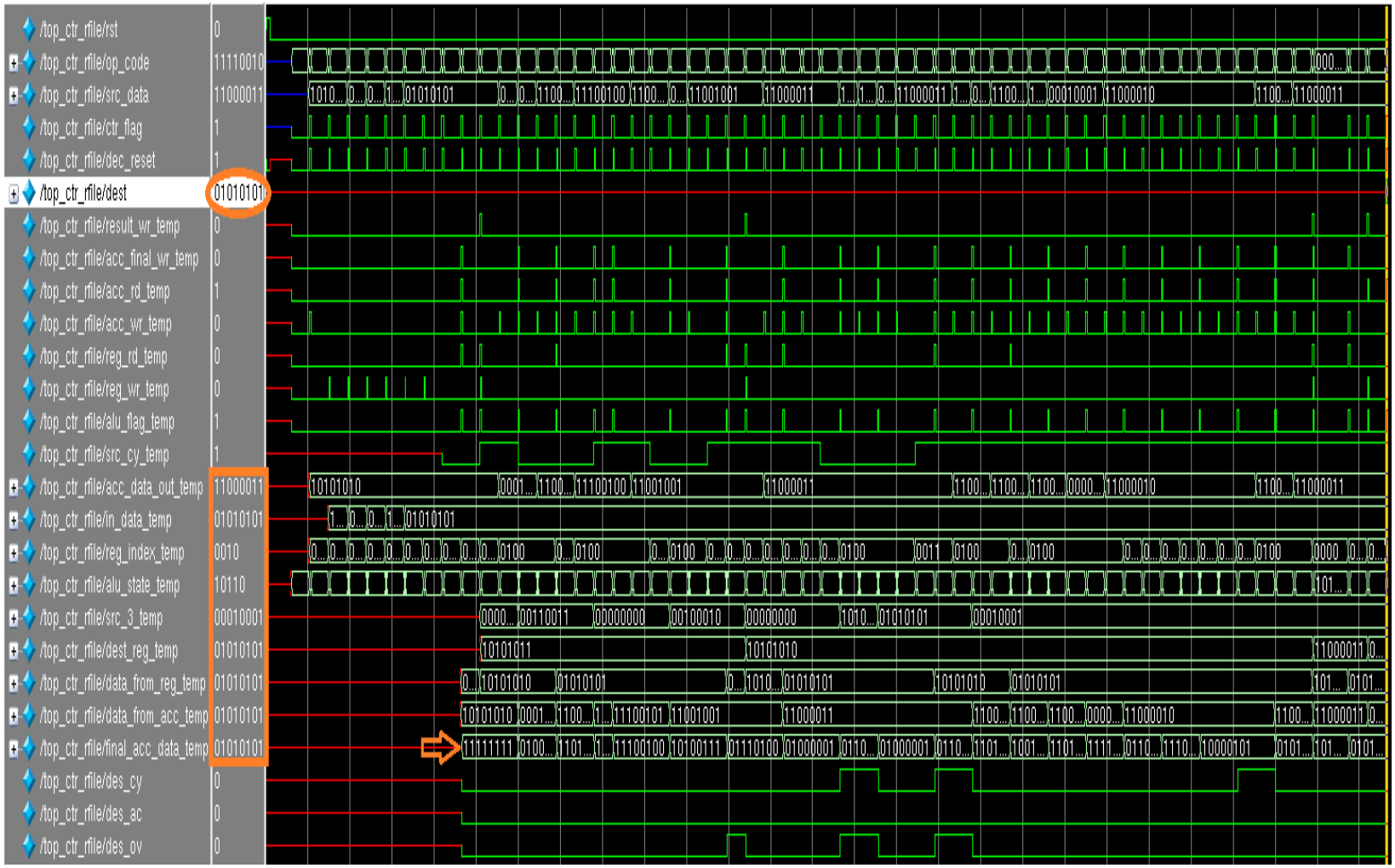

Figure 4.7: Asynchronous REU Core Post-Layout Simulation

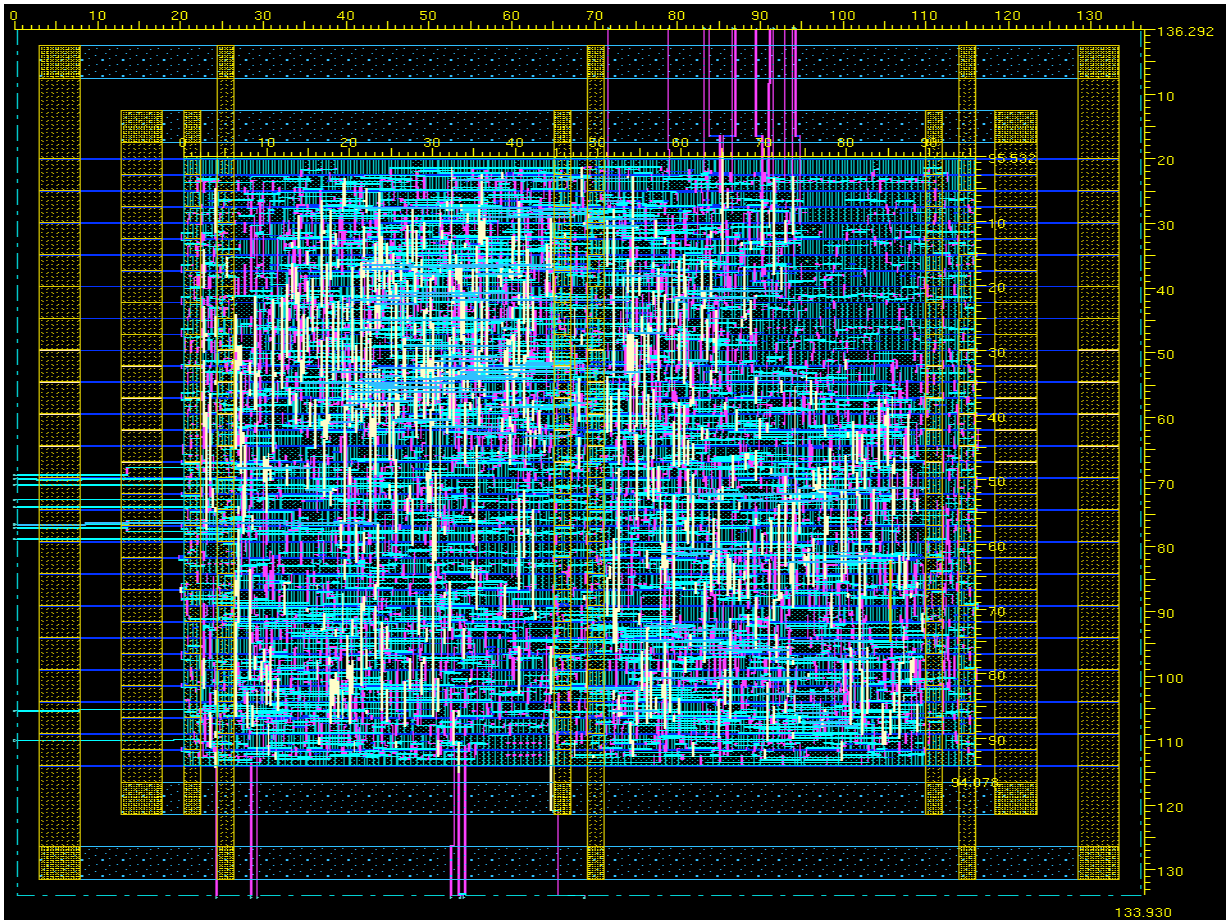

Figure 4.8: Asynchronous REU core Layout

4.2.3 REU Frontend

The decoder and the CRC block architectures introduced in Chapter 3 were independently simulated and verified. Both the decoder and the CRC blocks were successfully integrated to form the frontend REU design. Figure 4.9 represents a sample simulation for the Front-end design of the REU design. According to the data-driven encoding scheme, the testbench consists of encoded ‘0’s and ‘1’’s with ‘0’ having a pulse width much less than the pulse width

of a ‘1’. This testbench uses a pulse width of ‘1’, about 100 ns and the pulse width of ‘0’ = 2.5 ns, encoded bit-period is about 250 ns and a data clock is delayed by 3 ns.

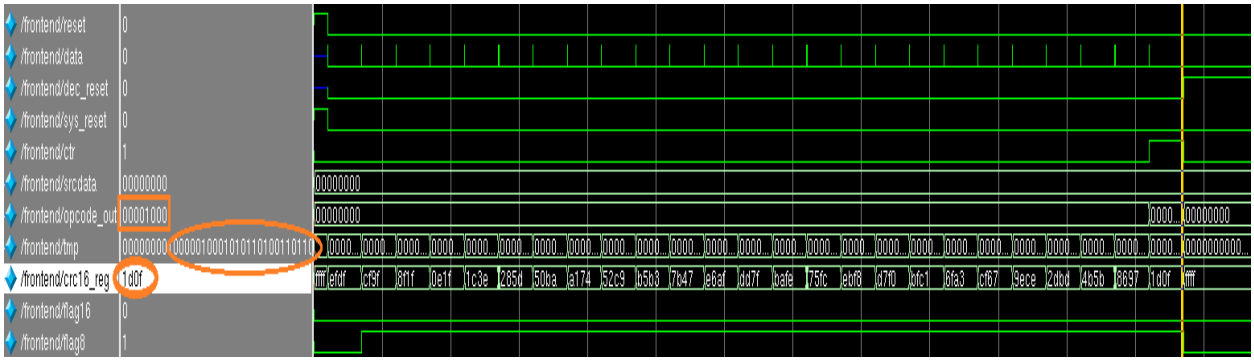

Figure 4.9: Frontend Post-Layout Simulation

The received encoded data is successfully decoded, and its format is of the form: LENGTH (1-bit)-OPCODE (8-bit)-CRC (16-bit), in other words a 25-bit input “0-00001000-1010110100110110”. The binary data highlighted as a circle represents the decoded data that is stored in the 34-bit shift register denoted by the ‘tmp’ signal output shown in Figure 4.9. The input data that is used as a clock signal for sampling the corresponding data signal is delayed by 3ns to produce the expected decoded output. The LENGTH (bit) field is used to differentiate the two variable length input frames (LEN-OPCODE-CRC [25-bit] and LEN(1-bit)-OPCODE(8-bit)-DATA(8-bit)-CRC(16-bit) [33-bit]) supported by the REU design. The OPCODE here represents the 8-bit 8051-instruction opcode format and the CRC -16 represents the 16-bit code used to detect transmission errors in the received 8-bit OPCODE. The CRC-16 polynomial was used to generate the above CRC value is $g(x)= x^{16}+x^{12}+x^{5}+1$.

Figure 4.9 represents a successful 25-bit input data format simulation result for the Front-end design. The 25-bit input frame input was used as the testbench: "0-00001000-1010110100110110"(in decoded format). The CRC-16 register was initially loaded with $FFFF_h$ and the "tmp" register with all 0's except for tmp(0). Now all of the 16 bits of the received CRC, and then the input bit stream is clocked into the "tmp" register beginning with the MSB. On identifying the input frame length, the decoder samples the encoded input frame on every rising edge of the delayed data to produce the correct decoded output as shown in register "tmp" as in Figure 4.9. As shown in Figure 4.9, the CRC-16 register simultaneously computes the CRC-value along with the decoding process and the transmitted data are valid as the of the CRC register value equals $1D0F_h$ at the end of the decoding process. This CRC register is denoted by "CRC16_reg" and its corresponding value is highlighted in the orange circle as shown in Figure 4.9. On identifying the validity of the 8-bit opcode, two main outputs are successfully updated with accurate values: opcode (8-bit) ("opcode_out") and control signal ("ctr"). It can be clearly seen from Figure 4.9 that the "opcode_out" and the "ctr" output registers are successfully updated with "00001000" (outlined in an orange rectangular box in Figure 4.9) and '1' respectively at the end of the decoding process. This opcode represents an INC Rn instruction. A decoder reset ("dec_reset") was also included in the testbench that was used to reset certain flags used for the internal operation of the REU core design at the end of the every instruction execution. This pin is triggered by the REU core design output and is not controlled externally. A sample VHDL code and a corresponding TCL script are presented in the Section 1.01(a)(i)Appendix C and Section 1.01(a)(i)Appendix D respectively.

Figure 4.10 shows the layout of the frontend design. The area dimension including pads for the layout as estimated by Cadence Encounter is 91 x 90 μm^2. The total number of cells used in the design as estimated by the Synopsys Design Vision is 607.

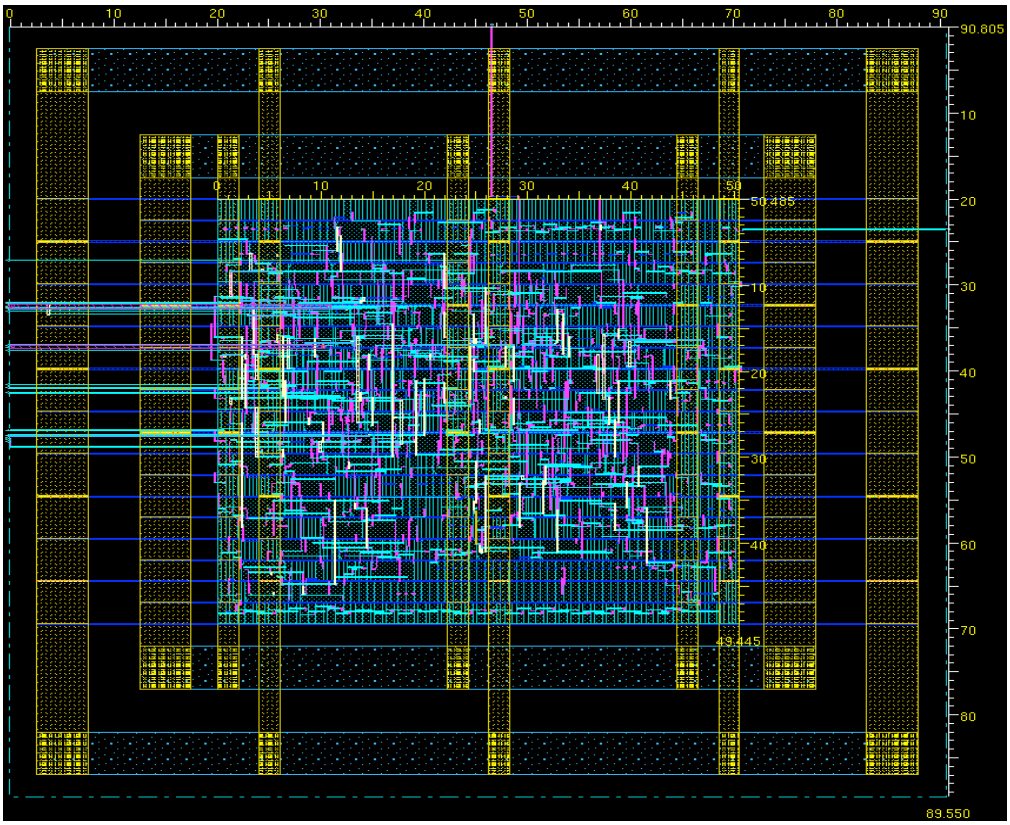

Figure 4.10: Layout of the REU Frontend

The clocked REU frontend design differs with the asynchronous REU frontend design mainly with the switch output signal that is the input to the REU core as illustrated in Chapter 3. The area occupied by the clocked REU frontend is the same as that of the asynchronous version.

4.2.4 Clocked REU

On successful post-layout verfication of the REU core and frontend, a switch component was implemented as demultiplexer and was tested for its functionality. On successful verfication of the switch design, it was intergrated with the fronend and REU core components as shown in Chapter 3 to form the clocked REU design. The successful post-layout of the entire clocked REU design has been successfully simulated and verified for various frequencies ranging from 1MHZ to 80 MHz. Figure 4.11 and Figure 4.12 represent successful simulation results of processing multiple variable (25 bit and 33-bit) instruction data formats for clocked REU with the Switch design at a clock frequency of 10 MHz.

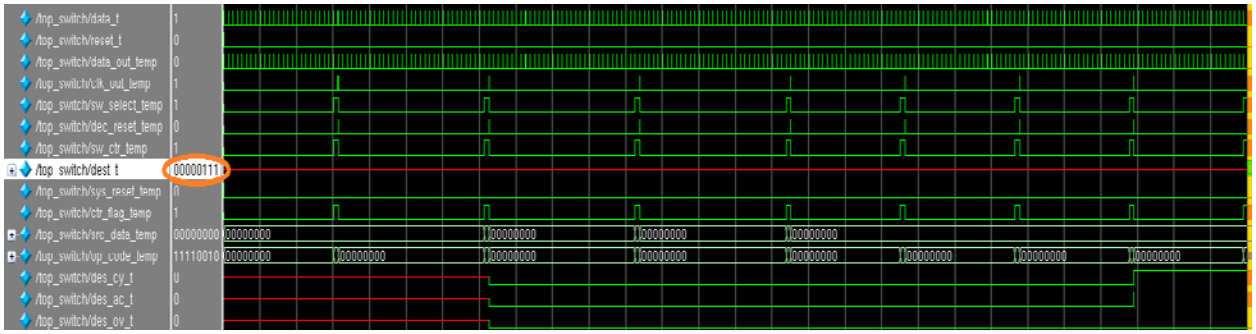

Figure 4.11: Clocked REU Post-Layout Simulation

Instruction Sequence-Simulation Test Bench for Figure 4.11*:*

CLR A and Content after execution in A =00000000

MOV R_0, #DATA1 and Content after execution in R_0 =01010101

MOV R_1, #DATA2 and Content after execution in R_1 = 10101010

MOV R_2, #DATA3 and Content after execution in R_2 = 00001000

MOV A, R_0 and Content after execution in A = 01010101

ADD A, R_1 and Content after execution in A = 11111111

ADD A, R_2 and Content after execution in A = 00000111

MOVX @ R_0 and A Content after execution in the output register dest_t = 00000111

In Figure 4.11, the result (stored in dest_t register) of the adding three 8-bit binary numbers stored in R_0 R_1 *and* R_2 respectively. The result highlighted by an orange circled marker represents the successful update of the final result of the add operation A and R_2. Finally the successful transfer of the computed data (A → dest_t) to the output register "dest_t". In Figure 4.12, the zoomed in Figure 4.11 version of the successful final add operation result update in the accumulator, and its transfer (A → dest_t) to the output register "dest_t" has been verified and updated accordingly.

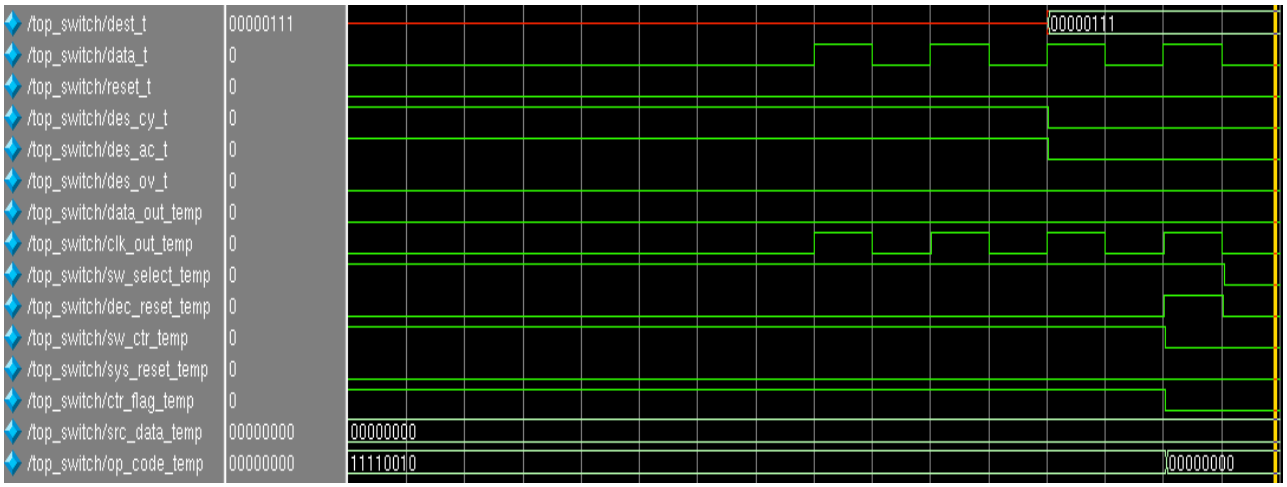

Figure 4.12: Final Result Simulation (zoomed_in version of Figure 4.11)

Figure 4.13 shows the layout of the clocked REU chip. The area dimension of the layout as estimated by Cadence Encounter is 142 x 139 μm^2.

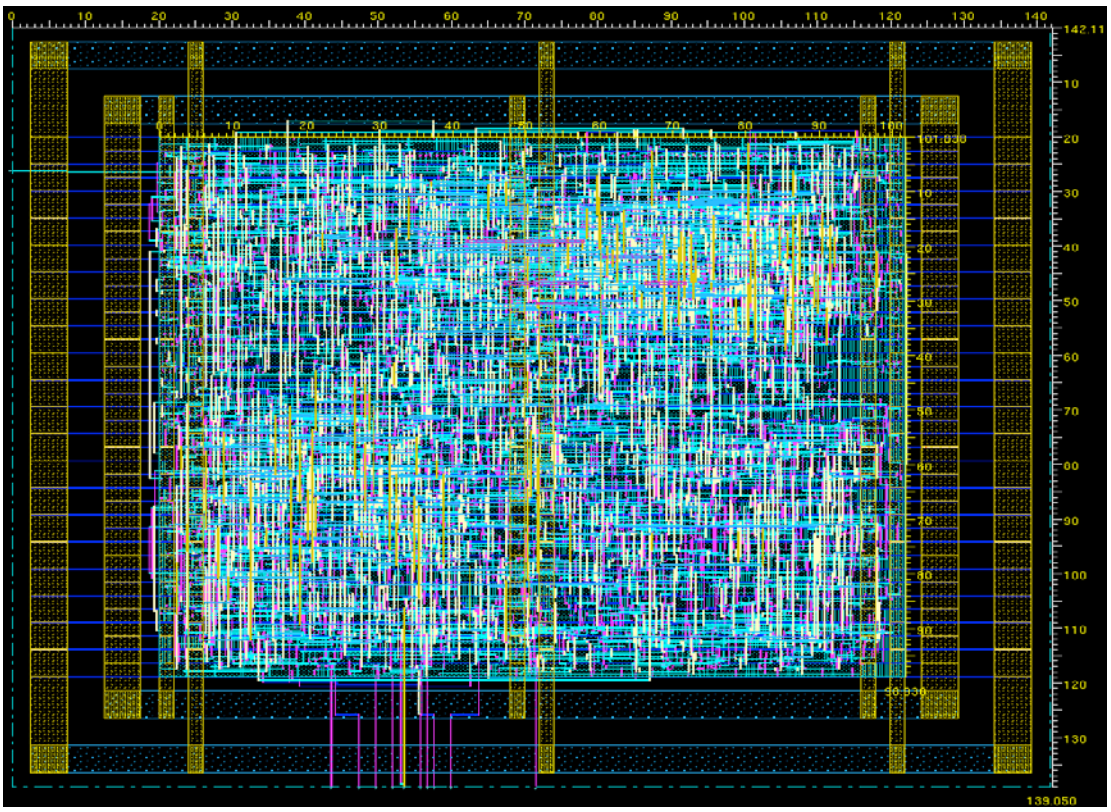

Figure 4.13: Clocked REU Layout

4.2.5 Asynchronous REU

On successful independent post-layout verfications of the asynchronous REU core and frontend, both of these modules were intergrated to form the asynchronous REU design as shown in Chapter 3. This entire REU design was successfully simulated and verified for various variable (25 bit and 33-bit) instruction data formats as shown in Figure 4.14.

The simulation of the asynchronous REU design shown in Figure 4.14 uses the same testbench as described in the previous section for the clocked design.

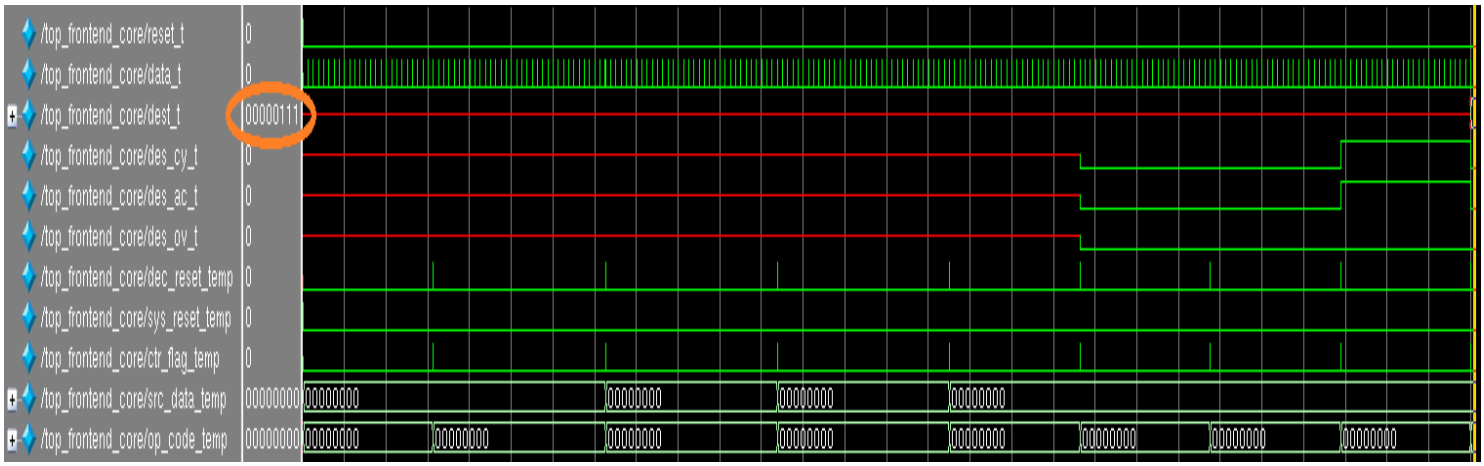

Figure 4.14: Asynchronous REU Post-Layout Simulation

In Figure 4.14, the result (stored in "dest_t" register) of the adding three 8-bit binary numbers stored in R_0 R_1 *and* R_2 respectively. The result highlighted by an orange circled marker represents the successful update of the final result of the add operation A and R_2. Finally the successful transfer of the computed data (A → dest_t) to the output register "dest_t" is completed.

Figure 4.15 shows the layout of the asynchronous REU chip. The core area dimension of the asynchronous REU layout as estimated by Cadence Encounter is 108 x 107 μm^2. The core area dimension of the clocked REU layout for comparison is about 102 x 99 μm^2.

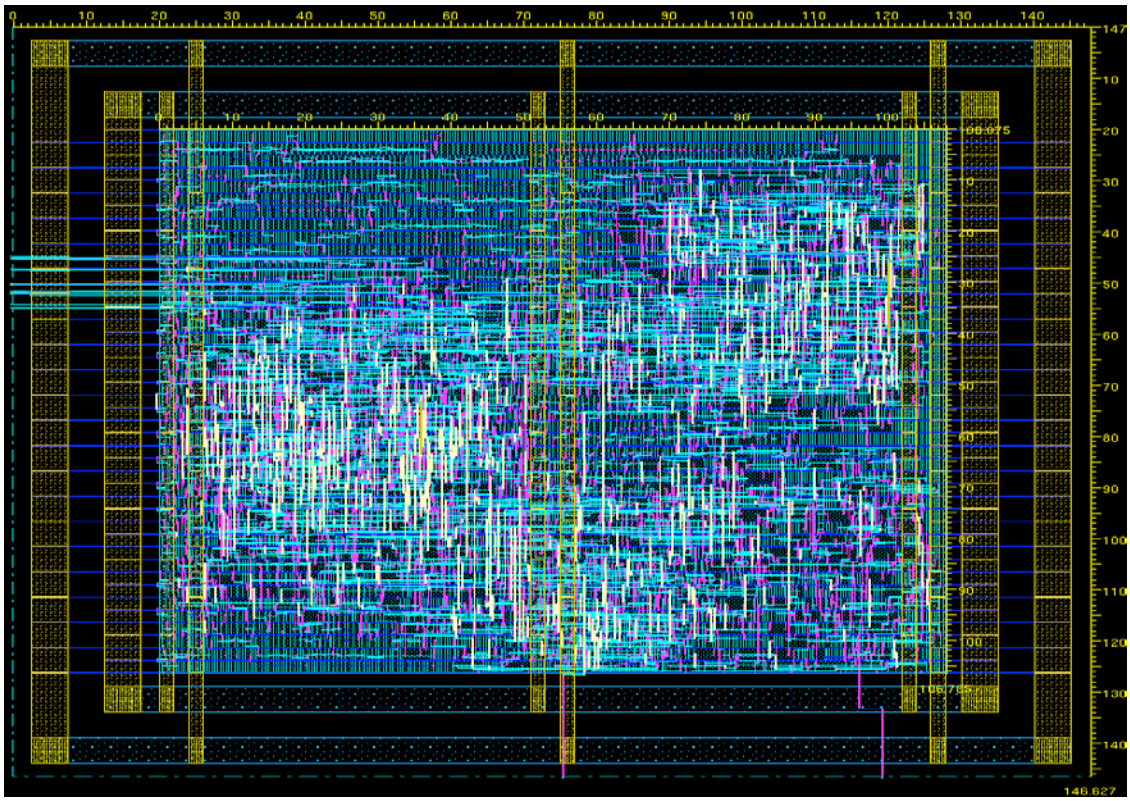

Figure 4.15: Asynchronous REU Layout

4.3 REU COMPARISONS

Both REU designs mainly consist of a core block and a frontend block. The main implementation difference between both the REU design versions is the core design, as both have the same the frontend design implementation. The 116 8051-instructions based REU core has been implemented both as a clocked and an asynchronous design. This section presents the comparisons of both the REU core implementations in terms of power, speed and area.

4.3.1 Power

The current 8051 models used in wireless biomedical sensor applications; embedded systems, etc., typically run at clock frequencies of 50MHz or greater [46], [47], [48], [49]. An existing 8051 Microcontroller core model [71] was identified and is used as a reference model for comparison with the clocked REU and the asynchronous REU. This complete ISA based 8051 core (256 instructions) is a fully synchronous design compatible with the Intel 8051 μC. This architecture has a higher performance average compared to the traditional one as it executes most of the instructions in one clock cycle. This model and its derivatives are used in wireless sensor applications [46].

The 8051 core model consists of four major blocks: ALU, control unit, serial interface unit and timer-counter. This 8051 core model has been debugged for a successful compilation of all its design blocks. This core design has been successfully synthesized using a target 45nm PTM technology for a supply voltage of 1.1 V. Hence, use of a common target technology, libraries and supply voltage for all the three models (clocked REU core, asynchronous REU core and the 8051 μC core) makes a justification for an accurate power comparison.

Figure 4.16 shows the layout of the 8051 core chip. The area dimension including the pads of the layout as estimated by Cadence Encounter is 262 x 258 μm^2. The total number of cells in the layout estimated using Synopsys Design Vision is 11,022.

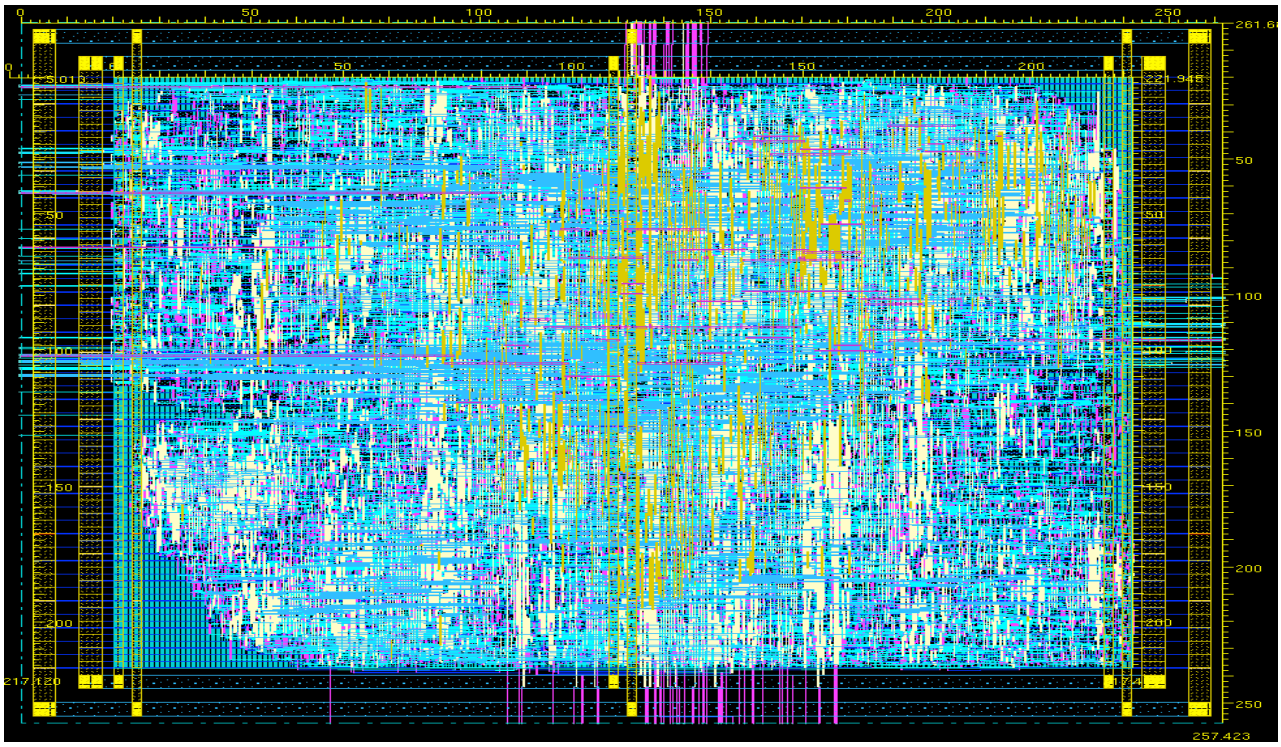

Figure 4.16: 8051 µC Core Layout

The clock speed for execution with the clocked design is a major determining factor for power consumption. Figure 4.17 summarizes the total power, dynamic power and leakage power values for the complete ISA based 8051 core (256 instructions) at 1 MHZ, 10 MHZ, 30 MHZ, 60 MHz and 80 MHZ clock frequencies respectively. These power values were generated by Cadence Encounter power option based on the input target clock frequency for the design. It can be clearly seen from Figure 4.17, as the frequency increases from 1 MHZ to 80 MHZ, the total power consumption also increases.

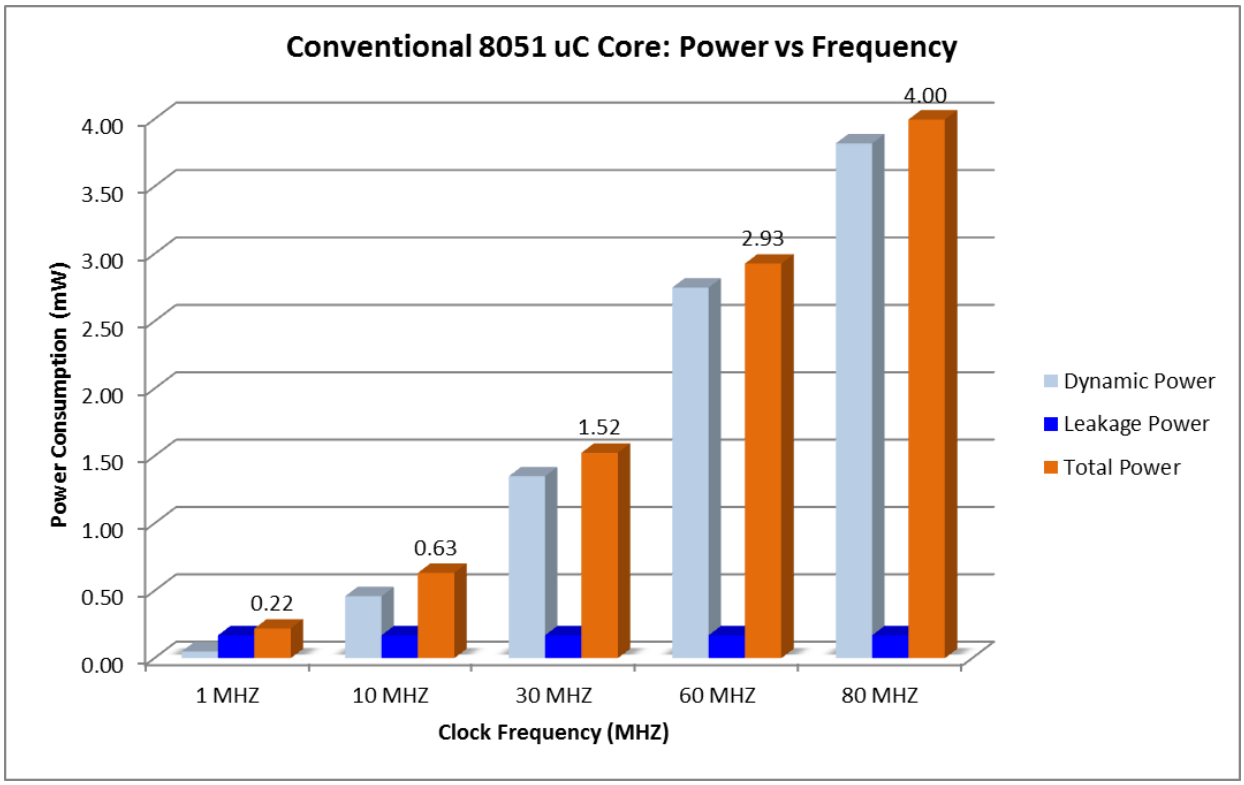

Figure 4.17: 8051uC: Power Consumption Vs Clock Frequency

Figure 4.18 reports a power comparison graph for direct data visualization of the clocked REU core (116 instructions) for a frequency range of 1 MHZ - 80 MHZ. It can be clearly seen from Figure 4.17 and Figure 4.18 that as the frequency increases from 1 MHZ to 80 MHZ, the dynamic power consumption increases which in turn contributes to increase in the total power consumption for both the complete ISA based 8051 core (256 instructions) and the clocked REU core (116 instructions) models. There is about 79 % decrease in the total power consumption of the clocked REU core when compared to the 8051 core design. The major reason for low power consumption of the clocked REU core when compared to the 8051 core is that the REU supports less than half the regular 8051 ISA. This savings in power is thus credited to the interrogator/REU distributed architecture concept.

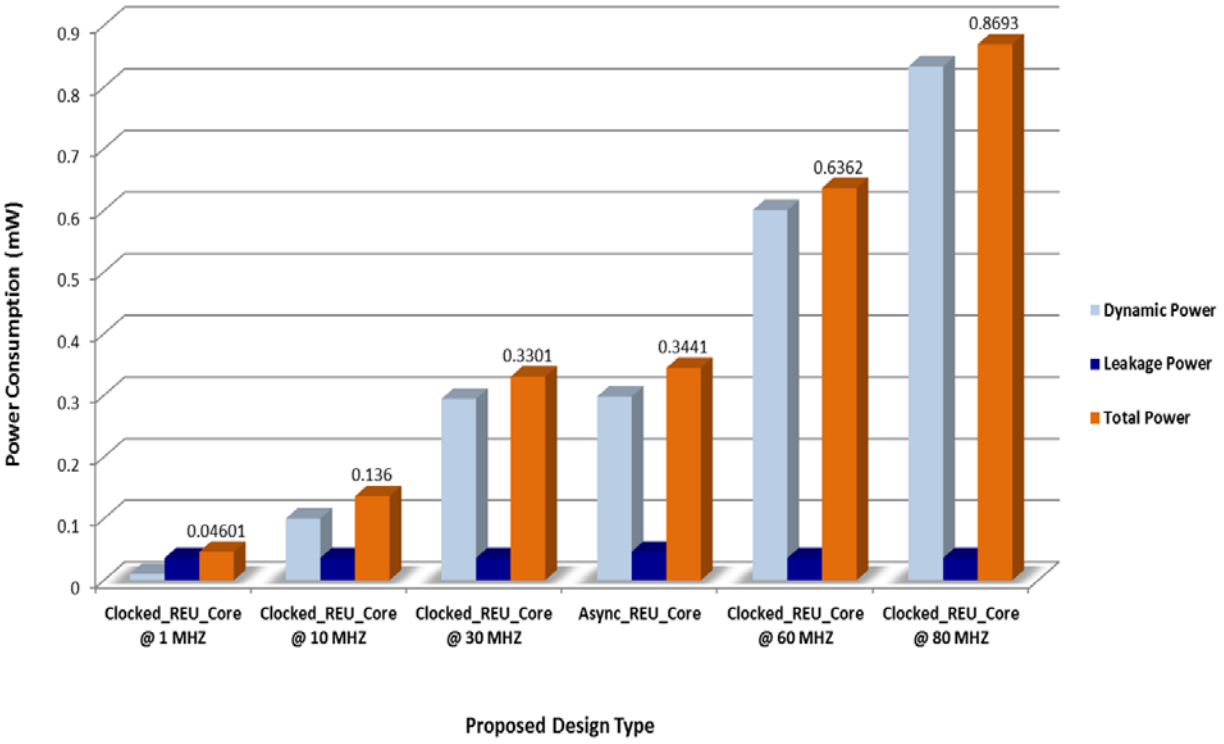

Figure 4.18: Power Consumption Vs Proposed REU Core Design Types

Figure 4.18 also illustrates the individual and the total power consumption values for the asynchronous REU core (116 instructions). The total power consumption of the asynchronous core is a fixed value for any frequency sweep. This fixed value of the asynchronous core design lies within the range of power consumption values of the clocked core design at 30MHZ and 60MHZ respectively, as shown in Figure 4.18. With the increase in clock frequency, the dynamic power consumption of the clocked core is the main contributor to the total power consumption. It can be clearly seen that the power consumption of the clocked core has significantly higher power consumption at higher frequencies when compared to the asynchronous core. The next section presents the execution speed comparisons for the asynchronous and the clocked REU cores.

4.3.2 Speed

Given that the frontend designs are the same in both cases, the execution time for comparison is based solely on the time to complete an instruction execution once the instruction has been determined to be valid. Obviously, execution (operation) time for the asynchronous version is fixed by the minimum delays possible in the REU design for the given technology, which will have some minimal difference from instruction to instruction but is not affected by the clock frequency. The execution time for the clocked version is solely the clocking time required for the execution of an instruction. The speed comparison for the clocked REU core is consequently relative to the clock frequency chosen. The following Figure 4.19 is an illustration of the execution time versus the clock frequency in the range of 10 MHZ-80 MHZ for the clocked REU core for set of sample instructions. The execution time for the clocked REU instructions decreases with the clock frequency.

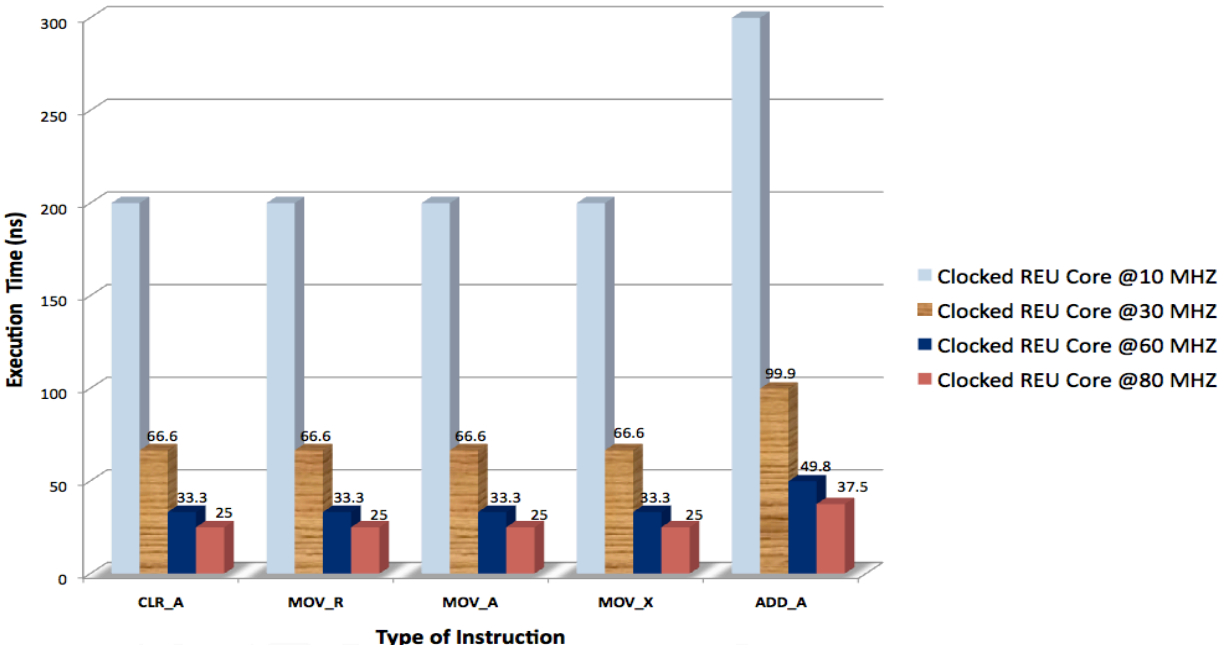

Figure 4.19: Execution time vs Instruction Type for REU Clocked Core

From Figure 4.18, it can be seen that the total power consumption value of the clocked core running at 60MHZ is the closest upper bound to the total power consumption value of the asynchronous core. Hence, the execution speed of the asynchronous core is compared to the clocked core running at 60 MHZ.

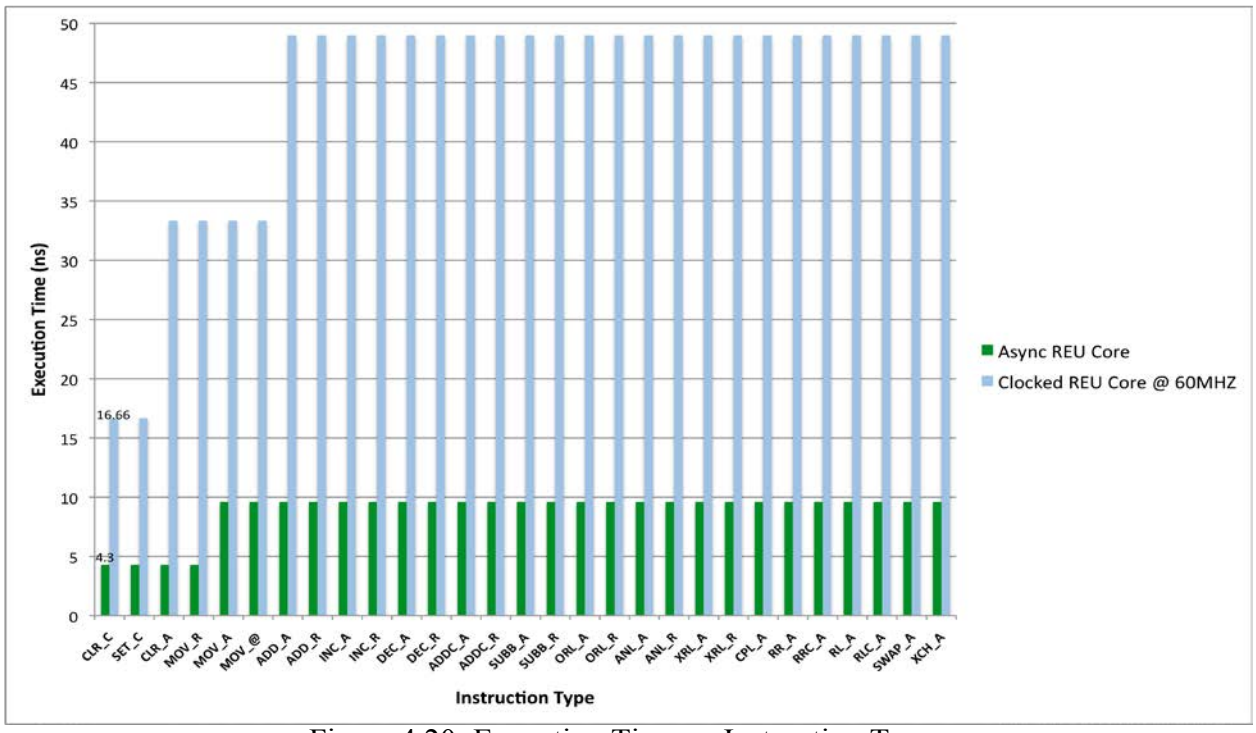

Figure 4.20: Execution Time vs Instruction Type

Figure 4.20 illustrates the execution time comparisons of the 8051 instructions that are part of the asynchronous core and the clocked core running at 60 MHZ. It can be seen from this figure that the clocked core executes its instructions accordingly in one, two and three clock cycles, whereas the asynchronous core executes the same set of instructions at a much faster rate. As the clock frequency is lowered (<<60MHz), the clocked core will have lower execution speeds whereas the asynchronous core will have a constant and much faster execution speed.

4.3.3 Area

The third basis of comparison is layout area required to fabricate the REU chip. Area is also a function of the actual subset of the 8051 instructions required. However, for the purpose of example in this research, the set of 116 instructions have been chosen for implementation.

The layout area of the 8051 core (256 instructions) compared to the clocked (116 instructions) REU and the asynchronous (116 instructions) REU core are presented in Figure 4.21. As can be seen from this figure, the clocked core (116 instructions) occupies slightly less area than that of the asynchronous core (116 instructions). The clocked and asynchronous difference is mainly due to delay elements required in the asynchronous core. It can be clearly seen a significant reduction in the core area for both versions of the REU core when compared to the 8051 core. The total occupied core area by the clocked and asynchronous REU core designs are about 84% and 81% respectively lower when compared to the 8051µC core.

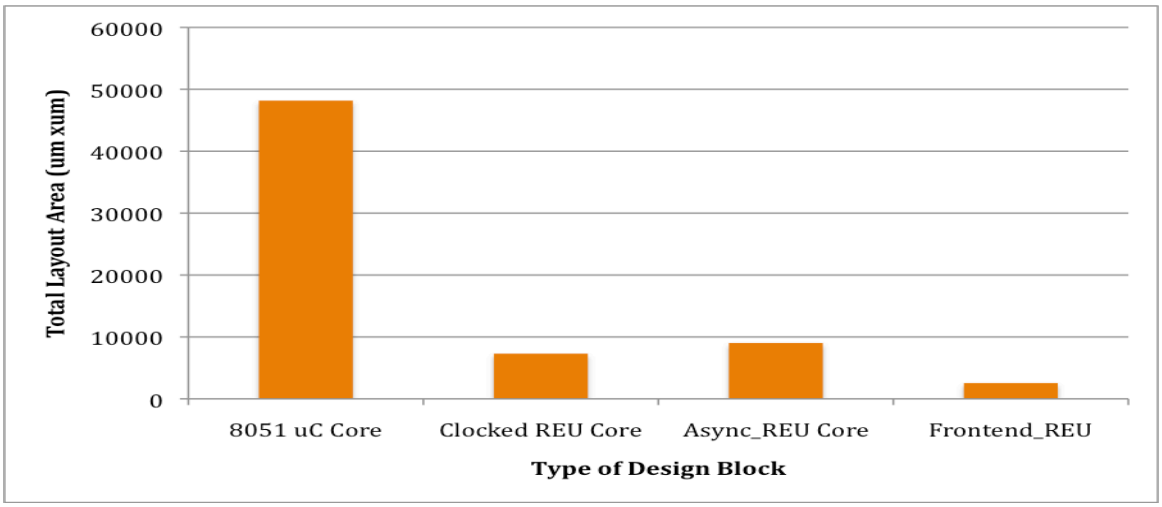

Figure 4.21: Layout Area Comparisons

4.3.4 Summary

This research has explored speed, area and power consumption for both the clocked and asynchronous REU core implementations based on 116 8051-instructions to understand their corresponding tradeoffs. The asynchronous core design has definite advantage of low power consumption with those of a clocked design at high frequencies (>>40MHZ). For low frequencies (<<40MHZ), the asynchronous design has a definite speed advantage at the cost of additional power consumption.

Based on the available power requirements, it is either necessary to use a clocked core running at very low frequencies compromising on the speed or use the asynchronous core running at much higher execution speeds. It can be concluded in such a context that the use of asynchronous or clocked design will depend on the power and execution speed requirements of the applications.

5.0 CONCLUSIONS

A WPSN system of passive nodes is remotely powered by an RF source and hence power consumption is a critical concern in such systems. This dissertation has presented novel low power programmable solutions applicable to such wireless nodes. A low-power wireless distributed node processor design concept has been demonstrated. A data-driven symbol decoder as part of the REU frontend has also been introduced as opposed to the conventional over-clocking symbol decoding process used in RF based devices. The clocked and asynchronous versions of the REU core design, synthesis and layout generation were successfully performed. Simulation results for area, speed and power of the post-layout REU design were presented. It is evident from the post-layout results that the proposed asynchronous REU core has lower power consumption when compared to the clocked version at higher frequencies. The clocked REU core design running at low frequencies (<<30 MHz) have an edge over using the asynchronous core with respect to power consumption, but run at much slower speeds. The target application requirements of power, speed, functionalities play an important role in choosing the ISA subset for the REU. A high-level asynchronous design flow has also been presented which is necessary to implement the asynchronous core using clocked CAD tool flows. This research provides the user with the flexibility of the 8051 software and development tools with the opportunity to further optimize the power, area and speed of the REU for any specific application if necessary.

This research has the potential to realize WPSN node applications for environmental, structural and medical fields especially while providing the basis for a programmable, reconfigurable and a low power passive processing unit for distributed computing.

5.1 CONTRIBUTIONS

This dissertation provides multi-domain low power solutions to increase the range of wireless passive devices such as RFID based nodes, generally used in biomedical sensors, environmental monitoring, supply chain logistics, etc. This research presented innovative architectures and design methodologies in the context of such power-constrained wireless nodes.

The interrogator and a set of passive nodes in combination are viewed as a wireless SIMD distributed system with the REU as the digital processing core of the passive node. This dissertation research has successfully designed and implemented both the clocked and asynchronous programmable REU core architectures based on the 116 instructions subset of the 8051 ISA. A high-level design flow for the asynchronous REU core implementation using synchronous CAD tools was also developed. The REU frontend was also implemented as a data-driven symbol decoder architecture for low power applications. The simulated power results show that the asynchronous REU core consumes considerably less power when compared to the clocked REU core running at high frequencies. The interrogator/REU architecture concepts and design elements related to instruction choices and separation of hardware between the interrogator and REU blocks introduced in this dissertation can be

extended to implement other microprocessor ISA's, e.g. Motorola 6800, Intel 8085, etc. The primary contributions of this work are listed as follows:

- Introduced the Wireless SIMD Distributed Architecture Concept
  - To remotely execute a subset of instructions on the passive nodes (REU) while requiring low power.
- Design and Implementation of a Low Power Programmable REU architecture
  - REU Frontend
    - Implemented as a data-driven symbol decoder-CRC design that eliminates the need for high frequency oscillators for low power applications.
  - REU Core
    - A subset of the 8051 ISA determined appropriate was chosen to implement the logic of the REU core for low power.
    - Implemented asynchronous and clocked REU core designs based on the 116 instructions subset of the 8051 ISA.
      - The resultant asynchronous design was found to consume lower power and have higher instruction execution speeds when compared to the clocked REU core design at high frequencies.
  - Developed a high-level CAD design flow for an asynchronous REU core
    - Modified the traditional clocked design flow in order to accomplish an asynchronous design implementation.

5.2 FUTURE DIRECTIONS

While the development of a low power REU is the focus of this dissertation research, the methodology of distributing program execution can be applied to other application fields such as compiler theory, distributed computing and environmental monitoring and wireless body sensor networks. Additionally, design parameter based security features can also be enhanced while providing a basis for a programmable passive processing unit.

The asynchronous nature of the REU provides for many opportunities for timing variations based on the instruction being executed and the amount of energy harvested by the REU. Exploration of such opportunities available to such REU designs can open up research avenues especially in enhancing the lightweight privacy and security features of the device.

The passive REU design is a type of flexible ASIC that can be used with remote sensing devices such as implantable medical sensors within the human body or long term environmental monitoring. The power, area and speed of such a design can be satisfactorily evaluated on a development tool before actually producing a custom programmable ASIC. Such a design process needs new tools to be able to target any ISA. This will require further development of new compilation techniques that focus on optimizing the execution of the distributed program.

The incorporation of sensors with the REU lowers the deployment cost enabling sensor networks to be deployed in applications such as Internet of Things (IOT) where they were previously too costly to develop a viable solution. Such a REU based passive device will need to be adapted for such applications enabling continued development of the existing protocols.

APPENDIX A

8051 INSTRUCTION DESCRIPTIONS

Table A1 represents 8051 instructions along with the corresponding description for each instruction.

Table A1: 8051 Instruction Descriptions

Instruction	Description	Instruction	Description
ACALL	Absolute Call	MOV	Move Memory
ADD, ADDC	Add Accumulator (With Carry)	MOVC	Move Code Memory
AJMP	Absolute Jump	MOVX	Move Extended Memory
ANL	Bitwise AND	MUL	Multiply Accumulator by B
CJNE	Compare and Jump if Not Equal	NOP	No Operation
CLR	Clear Register	ORL	Bitwise OR
CPL	Complement Register	POP	Pop Value From Stack
DA	Decimal Adjust	PUSH	Push Value Onto Stack
DEC	Decrement Register	RET	Return From Subroutine
DIV	Divide Accumulator by B	RETI	Return From Interrupt
DJNZ	Decrement Register and Jump if Not Zero	RL	Rotate Accumulator Left
INC	Increment Register	RLC	Rotate Accumulator Left Through Carry
JB	Jump if Bit Set	RR	Rotate Accumulator Right
JBC	Jump if Bit Set and Clear Bit	RRC	Rotate Accumulator Right Through Carry
JC	Jump if Carry Set	SETB	Set Bit
JMP	Jump to Address	SJMP	Short Jump
JNB	Jump if Bit Not Set	SUBB	Subtract From Accumulator With Borrow
JNC	Jump if Carry Not Set	SWAP	Swap Accumulator Nibbles
JNZ	Jump if Accumulator Not Zero	XCH	Exchange Bytes
JZ	Jump if Accumulator Zero	XCHD	Exchange Digits
LCALL	Long Call	XRL	Bitwise Exclusive OR
LJMP	Long Jump	UNDEF	Undefined Instruction

APPENDIX B

TESTBENCH FOR CLOCKED REU CORE @ 10MHz

The following script is a detailed testbench that was used in the post-layout simulation and verification of the 116 8051 instructions of the clocked REU core design for a 10 MHZ clock frequency. This testbench simulation is presented in Figure 4.5 as part of Chapter 4.

```
-----------------------------------------------
---TESTBENCH FOR CLOCKED_REU_CORE---10Mhz
-----------------------------------------------
        force -freeze rst 1
        force -freeze ctr 0
        force -freeze sw_ctr 0
        force -freeze clk 0
        run 50000 ps
        force -freeze rst 0
        run 250000 ps
        ------------------------------------------------------
        ----------MOVE DATA TO ACCUMULATOR----------
        force -freeze ctr 1
        force -freeze sw_ctr 1
        ##### MOVE DATA TO ACCUMULATOR ######
        force -freeze  op_code 01110100
        force -freeze src_data  10101010
        run 25000 ps

        force -freeze clk 1
        run 50000 ps
        force -freeze clk 0
        run 50000 ps
        force -freeze clk 1
        run 50000 ps
        force -freeze clk 0
        run 50000 ps
        force -freeze clk 1
        run 50000 ps
```

```
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------MOVE DATA TO all REGISTERS----------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO REGISTER R0 ######
force -freeze  op_code 01111000
force -freeze src_data  10101010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps

force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO REGISTER R1 ######
force -freeze op_code 01111001
force -freeze src_data 00011000
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
```

```
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO REGISTER R2 ######
force -freeze op_code 01111010
force -freeze src_data 01010100
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO REGISTER R3 ######
force -freeze  op_code 01111011
force -freeze src_data  10101010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
```

```
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO REGISTER R4 ######
force -freeze  op_code 01111100
force -freeze src_data  01010101
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps

force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO REGISTER R5 ######
force -freeze  op_code 01111101
force -freeze src_data  01010101
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------USING ADD A,Rn--------------------
force -freeze  op_code 00000000
force -freeze ctr 0
```

```
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### SET CARRY TO ZERO ######
force -freeze  op_code 11000011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ADD contents of REGISTER R5 TO ACCUMULATOR ######
force -freeze  op_code 00101101
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------------USING INC Rn----------------
--Current value in R4  = 01010101
----INC R4          -- R4  = 01010110
---------solution---
-------R4 = 01010110
-----------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
```

```
##### INC Rn ######
force -freeze  op_code 00001000
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------USING ADD with #data-------------
----MOV A,#16H   --A  = 00010110
----ADD A,#33    --A  = 01001001,#33=00110011
---------solution---
-------C = 0
-------ACC = 49H
----------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  00010110
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
```

```
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ADD #DATA TO ACCUMULATOR ######
force -freeze src_data  00110011
force -freeze  op_code 00100100
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------USING ORL----------------------
--MOV A,#C3H --A  = 11000011
--MOV R5,#55H --R5 = 01010101
--ORL A,R5    --A  = 11010111 or D7H
---solution---
--ACC = D7H
-------------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
```

```
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ORL contents of REGISTER R5 TO ACCUMULATOR ######
force -freeze  op_code 01001101
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------------USING INC A----------------
----MOV A,#E4H     --A  = 11100100
----INC A          --A  =11100101 or (E5)
---------solution---
-------ACC = E5H
-----------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11100100
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
```

```
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### INC A ######
force -freeze  op_code 00000100
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------------USING DEC A----------------
----ACCUMULATOR ALREADY HAS 11100101
----DEC A          -- A  = 11100100
---------solution---
-------ACC = E4H
-----------------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### DEC A ######
force -freeze  op_code 00010100
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
```

```
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------------USING SUBB with #data----------------
---Initially assume Carry is '0'
----MOV A,#C9H     --A  = 11001001
----SUBB A,#22    --A  =10100111 or (A7)  ,#22 = 00100010
---------solution---
-------C = 0
-------OV = 0
-------AC = 0
-------ACC = A7H
------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11001001
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
```

```
force -freeze ctr 1
force -freeze sw_ctr 1
##### SET CARRY TO ZERO ######
force -freeze  op_code 11000011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### SUBB #DATA TO ACCUMULATOR ######
force -freeze  op_code 10010100
force -freeze src_data  00100010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------------USING SUBB (A-R2) WITH CARRY------------------
---Initially assume Carry is '1'
----MOV A,#0C9H  --A  = 11001001
----MOV R2,#54H  --R2 = 01010100
----SUBB A,R2    --A  = 01110100
---------solution---
-------C = 0
-------AC = 0
-------OV = 1
-------ACC = 74H
-----------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
```

```
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data 11001001
run 25000 ps

force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### SET CARRY TO ONE ######
force -freeze  op_code 11010011
run 25000 ps

force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### SUBB contents of REGISTER R2 from ACCUMULATOR ######
force -freeze  op_code 10011010
run 25000 ps

force -freeze clk 1
run 50000 ps
force -freeze clk 0
```

```
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------------USING DEC Rn----------------
--Current value in R0  = 10101010
----DEC R0          -- R0  = 10101011
---------solution---
-------R0 = 10101011
------------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### DEC Rn ######
force -freeze  op_code 00011000
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
-----------------USING ANL--------------------
--MOV A,#C3H --A  = 11000011
--MOV R5,#55H --R5 = 01010101
--ANL A,R5    --A  = 01000001 or 41H
```

```
---solution---
--ACC = 41H
------------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ANL contents of REGISTER R5 TO ACCUMULATOR ######
force -freeze  op_code 01011101
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
```

```
force -freeze clk 0
run 50000 ps
----------------USING ADDC with #data----------------
---Initially assume Carry is '0'--------------
----MOV A,#C3H     --A  = 11000011
----ADDC A,#A9H    --A  = 01101100, #A9=10101001
---------solution---
-------C = 1
-------OV = 1
-------AC = 0
-------ACC = 6CH
-----------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### CLR CARRY TO ZERO ######
force -freeze  op_code 11000011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
```

```
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ADDC #DATA TO ACCUMULATOR ######
force -freeze  op_code 00110100
force -freeze src_data  10101001
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------USING ANL with #data-------------
----MOV A,#C3H   --A  = 11000011
----ANL A,#55    --A  = 01000001 or 41,#55=01010101
---------solution---
-------ACC = 41H
------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
```

```
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ADD #DATA TO ACCUMULATOR ######
force -freeze src_data  01010101
force -freeze  op_code 01010100
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------------USING ADDC ------------------------
---Initially assume Carry is '1'
----MOV R0,#0AAH --R0  = 10101010
----MOV A,#0C3H  --A   = 11000011
----ADDC A,R0    --A   = 01101110 OR 6EH
---------solution---
-------C = 1
-------OV = 1
-------AC = 0
-------ACC = 6E
------------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000011
run 25000 ps
```

```
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### SET CARRY TO ONE ######
force -freeze  op_code 11010011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ADDC contents of REGISTER R3 TO ACCUMULATOR ######
force -freeze  op_code 00111011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
```

```
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------USING ORL with #data-------------
----MOV A,#C2H   --A  = 11000010
----ORL A,#11    --A  = 11010011 or C3, #11 = 00010001
---------solution---
-------ACC = C3H
-----------------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ORL #DATA TO ACCUMULATOR ######
force -freeze src_data  00010001
force -freeze  op_code 01000100
run 25000 ps

force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
```

```
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------USING XRL----------------------
--MOV A,#C3H --A  = 11000011
--MOV R5,#55H --R5 = 01010101
--XRL A,R5    --A  = 10010110 or 96H
---solution---
--ACC = 96H
-------------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### XRL contents of REGISTER R5 TO ACCUMULATOR ######
force -freeze  op_code 01101101
run 25000 ps
force -freeze clk 1
```

```
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------USING XRL with #data-------------
----MOV A,#C2H   --A  = 11000010
----XRL A,#11    --A  = 11010011 or D3, #11 = 00010001
---------solution---
-------ACC = D3H
------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
```

```
force -freeze sw_ctr 1
##### XRL #DATA TO ACCUMULATOR ######
force -freeze src_data  00010001
force -freeze  op_code 01100100
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------CLR ACCUMULTOR----------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
force -freeze  op_code 11100100
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------CPL ACCUMULTOR----------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
```

```
force -freeze sw_ctr 1
force -freeze  op_code 11110100
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------RR ACCUMULTOR----------
--Before = A C2h =11000010
--After = A = 61h =01100001
-----------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
```

```
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ROTATE RIGHT ACCUMULATOR BY BIT ######
force -freeze  op_code 00000011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------RRC ACCUMULTOR----------
--Before = A C2h =11000010 and C=0
--After = A = 01100001 and C = 0
-----------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
```

```
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ROTATE RIGHT ACCUMULATOR BY BIT WITH CARRY ######
force -freeze  op_code 00010011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------RL ACCUMULTOR----------
--Before = A = C2h =11000010
--After = A = 85h = 10000101
--------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
```

```
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ROTATE LEFT ACCUMULATOR BY BIT ######
force -freeze  op_code 00100011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------RLC ACCUMULTOR----------
--Before = A = C2h =11000010 and C = 0
--After = A = 85h = 10000100 and C = 1
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
```

```
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### ROTATE RIGHT ACCUMULATOR BY BIT WITH CARRY ######
force -freeze  op_code 00110011
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------SWAP ACCUMULTOR----------
--Before = A = C5h =11000101
--After = A = 5Ch = 01011100
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000101
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
```

```
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### SWAP (NIBBLE) ACCUMULATOR ######
force -freeze  op_code 11000100
run 25000 ps

force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------USING XCH A,Rn----------------------
--MOV A,#C3H  --A  = 11000011
--MOV R0,#AAH --R0 = 10101010
--XRL A,R0    --A  = 10101010 AND R0 = 11000011
---solution---
--A  = 10101010 and R0 = 11000011
-----------------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE DATA TO ACCUMULATOR ######
force -freeze  op_code 01110100
force -freeze src_data  11000011
run 25000 ps
force -freeze clk 1
run 50000 ps
```

```
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### XCH EXCHANGE contents of REGISTER R0 AND ACCUMULATOR ######
force -freeze  op_code 11001000
run 25000 ps

force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
----------------MOV A,R5---------------------
--MOV A,R5
--Before = A = 10101010
--After = A = 01010101
```

```
-----------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE contents of REGISTER R5 TO ACCUMULATOR ######
force -freeze  op_code 11101101
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------MOV R0,A----------------------
--MOV R0,A
--Before = R0 = 11000011
--After = R0 = 01010101
-----------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE contents of REGISTER R5 TO ACCUMULATOR ######
force -freeze  op_code 11111000
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
```

```
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
------------------MOV @R0,A----------------------
--MOV @R0,A
--TRANSMITTING REGISTER (T) <- A (8-bit)
--Transfer ACC data to an external register (data_out) used to transmit data
--Before = data_out = 00000000
--After = data_out = 01010101
------------------------------------------------------------
force -freeze  op_code 00000000
force -freeze ctr 0
force -freeze sw_ctr 0
run 200000 ps
force -freeze ctr 1
force -freeze sw_ctr 1
##### MOVE contents of REGISTER R5 TO ACCUMULATOR ######
force -freeze  op_code 11110010
run 25000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
force -freeze clk 1
run 50000 ps
force -freeze clk 0
run 50000 ps
```

APPENDIX C

VHDL CODE FOR FRONTEND

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use ieee.std_logic_unsigned.all ;

entity frontend is
  port(data : in  std_logic;
       reset : in  std_logic;
       dec_reset : in  std_logic;

       sys_reset : out  std_logic;
       ctr : out std_logic ;
       srcdata : out std_logic_vector(7 downto 0);
       opcode_out  : out std_logic_vector(7 downto 0));
end frontend;

architecture frontend_beh of frontend is

  signal tmp: std_logic_vector(33 downto 0);
  signal data_clk: std_logic;
  signal crc16_reg, crc_int : std_logic_vector (15 downto 0);
  signal xor12   : std_logic;
  signal xor0    : std_logic;
  signal xor5    : std_logic;
  signal xor16   : std_logic;
  signal reg4,reg11,reg15 : std_logic;
  signal sregout  : std_logic_vector(7 downto 0);
  signal flag8, flag16   : std_logic;

  begin

    process (data_clk,data,tmp,reset,dec_reset)

            variable flag8, flag16   : std_logic;

      begin
```

```
        if ((reset = '1' and dec_reset = '1') or (reset = '1' and dec_reset = '0') or (reset = '0' and dec_reset = '1')
then

                if(reset = '1') then
                        sys_reset <= '1';
        else
                        sys_reset <= '0';
                end if;

                tmp <= "00000000000000000000000000000001";
                data_clk <= '0';
        crc16_reg <= "1111111111111111";
                crc_int <= "0000000000000000";
                sregout <= "00000000";
                xor12 <= '0';
        xor0 <= '0';
        xor5 <= '0';
        xor16 <= '0';
                reg15 <= '0';
                reg11 <= '0';
             reg4 <= '0';

                flag16 := '0';
                flag8 := '0';
                srcdata <= "00000000";
                opcode_out <= "00000000";

                ctr <= '0';

      else

        data_clk <= data;

        if(reset = '0') then
                sys_reset <= '0';
    end if;

    if (data_clk'event and data_clk='1') then

        tmp <= tmp(32 downto 0) & data;


                reg15 <= crc16_reg(15);
                reg11 <= crc16_reg(11);
             reg4 <= crc16_reg(4);

                crc16_reg(15 downto 13) <= crc16_reg(14 downto 12);
         crc16_reg(11 downto 6)  <= crc16_reg(10 downto 5);
                crc16_reg(4 downto 1)   <= crc16_reg(3 downto 0);

                xor16 <= reg15 ;
                crc16_reg(12) <= '1';
                crc16_reg(5) <= '1';
        crc16_reg(0) <= '1';
```

```
                if (flag16 = '0' and flag8 = '0') then

                        if (tmp(1 downto 0) = "11") then

                                flag16 := '1';

                        elsif (tmp(1 downto 0) = "10") then
                                flag8 := '1';

                        end if;

                end if;

                ctr <= '0';

end if;

        if(data_clk = '0')then

         xor16 <= reg15 xor tmp(0);
                crc16_reg(12) <= xor16 xor reg11;
                crc16_reg(5) <= xor16 xor reg4;
         crc16_reg(0) <= xor16 xor '0';

                crc_int <= crc16_reg;



                if (tmp(33) = '1' and flag16 = '1') then

                        if(crc_int = "0001110100001111") then

                        --Received data CRC-16 CHECK FOR 1D0F

                                opcode_out <= tmp (31 downto 24); -- 8-bit opcode
                srcdata <= tmp (23 downto 16); -- 8-bit data
                                ctr <= '1';

                else

                        ctr <= '0';

                        --Ignores the command if the received data is invalid

                end if;
        end if;

                if (tmp(25 downto 24) = "10" and flag8 = '1') then

                        if(crc_int = "0001110100001111") then

                        --Received data CRC-16 CHECK FOR 1D0F

                                opcode_out <= tmp (23 downto 16); -- 8-bit opcode
                                ctr <= '1';
```

```
                    else
                    ctr <= '0';
                    --Ignores the command if the received data is invalid
                end if;
                end if;
            end if;
        end if;

    end process;
end frontend_beh;
```

APPENDIX D

TCL SCRIPT FOR FRONTEND

```
#########################################################
# Change following configurations for your design  #
#########################################################
# HDL file names (.v or .vhd)                    #
set my_HDL_files frontend.vhd


# Top-level Module / Entity name                 #
set my_toplevel frontend


# The name of the clock pin                     #
# If no clock-pin exists, pick anything          #
set my_clock_pin no_clk

# Target frequency in MHz for optimization        #
set my_clk_freq_MHz 200

# Delay of input signals (Clock-to-Q, Package etc.)#
set my_input_delay_ns 0.1

# Reserved time for output signals (Holdtime etc.) #
set my_output_delay_ns 0.1

#########################################################
define_design_lib WORK -path ./DC_WORK
file mkdir DC_reports
set verilogout_show_unconnected_pins "true"
set_ultra_optimization true
set_ultra_optimization -force

set ext [file extension $my_HDL_files]
if { $ext == ".v" } {
   analyze -f verilog $my_HDL_files
} elseif { $ext == ".vhd" } {
   analyze -f vhdl $my_HDL_files
```

```
} else {
   puts "File Format Error!"
   quit
}

elaborate $my_toplevel

current_design $my_toplevel

#set_dont_touch [get_nets tmp]

#set_min_delay 1.5 -from data -through U17 -to tmp_reg[0]
#set_max_delay 0.5 -from {in} -to {out}

#set_wire_load_model -name "" -library "gscl45nm"

check_design
link
uniquify

set my_period [expr 1000 / $my_clk_freq_MHz]

set find_clock [ find port [list $my_clock_pin] ]
if { $find_clock != [list] } {
  set clk_name $my_clock_pin
  create_clock -period $my_period $clk_name
} else {
  set clk_name vclk
  create_clock -period $my_period -name $clk_name
}

# If input need to be buffered, enable this       #
#set_driving_cell  -lib_cell INVX1  [all_inputs]
set_input_delay $my_input_delay_ns -clock $clk_name [remove_from_collection [all_inputs]
$my_clock_pin]
set_output_delay $my_output_delay_ns -clock $clk_name [all_outputs]

compile -ungroup_all -map_effort medium
check_design

set_min_delay  3 -from U302/Y -to {tmp_reg[0]/CLK tmp_reg[1]/CLK tmp_reg[2]/CLK
tmp_reg[3]/CLK tmp_reg[4]/CLK tmp_reg[5]/CLK tmp_reg[6]/CLK tmp_reg[7]/CLK tmp_reg[8]/CLK
tmp_reg[9]/CLK tmp_reg[10]/CLK tmp_reg[11]/CLK tmp_reg[12]/CLK tmp_reg[13]/CLK tmp_reg[14]/CLK
tmp_reg[15]/CLK tmp_reg[16]/CLK tmp_reg[17]/CLK tmp_reg[18]/CLK tmp_reg[19]/CLK tmp_reg[20]/CLK
tmp_reg[21]/CLK tmp_reg[22]/CLK tmp_reg[23]/CLK tmp_reg[24]/CLK tmp_reg[25]/CLK tmp_reg[26]/CLK
tmp_reg[27]/CLK tmp_reg[28]/CLK tmp_reg[29]/CLK tmp_reg[30]/CLK tmp_reg[31]/CLK tmp_reg[32]/CLK
tmp_reg[33]/CLK xor16_reg/CLK reg15_reg/CLK reg11_reg/CLK reg4_reg/CLK flag16_reg/CLK
flag8_reg/CLK crc16_reg_reg[0]/CLK crc16_reg_reg[1]/CLK crc16_reg_reg[2]/CLK crc16_reg_reg[3]/CLK
crc16_reg_reg[4]/CLK crc16_reg_reg[5]/CLK crc16_reg_reg[6]/CLK crc16_reg_reg[7]/CLK
crc16_reg_reg[8]/CLK crc16_reg_reg[9]/CLK crc16_reg_reg[10]/CLK crc16_reg_reg[11]/CLK
crc16_reg_reg[12]/CLK crc16_reg_reg[13]/CLK crc16_reg_reg[14]/CLK crc16_reg_reg[15]/CLK}

set_min_delay 3.5 -from U149/Y -to {ctr_reg/S}

set_min_delay 4 -from dec_reset -to {U320/B}
```

```
compile -incremental_mapping -map_effort medium

check_design
redirect ./DC_reports/constraint_violators.rep { report_constraint -all_violators -verbose }

set filename [format "%s%s"  $my_toplevel "_SYN.v"]
write -f verilog -output $filename

set filename [format "%s%s"  $my_toplevel "_SYN.sdf"]
write_sdf $filename

set filename [format "%s%s"  $my_toplevel ".sdc"]
write_sdc $filename

redirect ./DC_reports/timing.rep { report_timing }
redirect ./DC_reports/power.rep { report_power }
redirect ./DC_reports/area.rep { report_area }
redirect ./DC_reports/clock.rep { report_clock }
redirect ./DC_reports/resource.rep { report_resource }
redirect ./DC_reports/cell.rep { report_cell }

quit
```

APPENDIX E

VHDL CODE FOR ASYNCHRONOUS REU CORE

The following vhdl code represents a top-level module of the asynchronous REU core design described in Figure 3.11 as part of Chapter 3.

```
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use WORK.REU_opcode_lib.all;
--------------------------------------------------------------------------------
entity top_ctr_rfile is

   port(   rst             : in  STD_LOGIC ;
           op_code : in  STD_LOGIC_VECTOR (7 downto 0) ;
           src_data : in  UNSIGNED (7 downto 0) ;
           ctr_flag : in STD_LOGIC;

           dec_reset :out STD_LOGIC;
           dest   : out UNSIGNED (7 downto 0);
           des_cy  : out STD_LOGIC;
           des_ac  : out STD_LOGIC;
           des_ov  : out STD_LOGIC
 );

end top_ctr_rfile;

--------------------------------------------------------------------------------

architecture BHV of top_ctr_rfile is

component reu_base is
       port(   rst             : in  STD_LOGIC;
               op_code : in  STD_LOGIC_VECTOR (7 downto 0);
               src_data   : in  UNSIGNED (7 downto 0);
               ctr_flag : in STD_LOGIC;
               result_wr : out  STD_LOGIC;
```

```
            acc_final_wr : out  STD_LOGIC;
            acc_rd : out  STD_LOGIC;
            acc_wr : out  STD_LOGIC;
            acc_data_out: out  UNSIGNED (7 downto 0);
            data_reg : out  UNSIGNED (7 downto 0);
            rd : out  STD_LOGIC;
            wr : out  STD_LOGIC;
            reg_index :out  STD_LOGIC_VECTOR (3 downto 0);
            exe_state :out UNSIGNED (4 downto 0);
            src_3   : out  UNSIGNED (7 downto 0);
            alu_flag_out : out  STD_LOGIC;
            src_cy  : out  STD_LOGIC );
end component reu_base ;

component rfile is
   port(rst         : in  STD_LOGIC;
            reg_index    : in  STD_LOGIC_VECTOR (3 downto 0);
            in_data      : in  UNSIGNED (7 downto 0);
            result_in_data  : in  UNSIGNED (7 downto 0);
            result_wr    : in  STD_LOGIC;
            rd           : in  STD_LOGIC;
            wr           : in  STD_LOGIC;
            out_data     : out UNSIGNED (7 downto 0));
end component rfile ;

component alu is
   port(   rst : in STD_LOGIC;
            alu_state :in  UNSIGNED (4 downto 0);
            src_1   : in  UNSIGNED (7 downto 0);
            src_2   : in  UNSIGNED (7 downto 0);
            src_3   : in  UNSIGNED (7 downto 0);
            src_cy  : in  STD_LOGIC;
            alu_flag : in STD_LOGIC;
            dec_reset : out STD_LOGIC;
            des_reg   : out UNSIGNED (7 downto 0);
            des_acc : out UNSIGNED (7 downto 0);
            des_out   : out UNSIGNED (7 downto 0);
            des_cy  : out STD_LOGIC;
            des_ac  : out STD_LOGIC;
            des_ov  : out STD_LOGIC );
end component alu ;

component acc_reg is
   port(rst         : in  STD_LOGIC;
            acc_data      : in  UNSIGNED (7 downto 0);
            final_acc_data  : in  UNSIGNED (7 downto 0);
            acc_final_wr    : in  STD_LOGIC;
            acc_rd          : in  STD_LOGIC;
            acc_wr          : in  STD_LOGIC;
            acc_data_out     : out UNSIGNED (7 downto 0) );
end component acc_reg;


   signal data_from_reg_temp  : UNSIGNED (7 downto 0);
   signal data_from_acc_temp  : UNSIGNED (7 downto 0);
   signal in_data_temp  : UNSIGNED (7 downto 0);
```

```
    signal reg_index_temp: STD_LOGIC_VECTOR (3 downto 0);
    signal reg_rd_temp : STD_LOGIC;
    signal reg_wr_temp : STD_LOGIC;
    signal result_wr_temp : STD_LOGIC;
    signal src_3_temp  : UNSIGNED (7 downto 0);
    signal src_cy_temp : STD_LOGIC;
    signal alu_state_temp  : UNSIGNED (4 downto 0);
    signal dest_reg_temp  : UNSIGNED (7 downto 0);
    signal reg_flag_temp :STD_LOGIC;
    signal acc_flag_temp :STD_LOGIC;
    signal alu_flag_temp :STD_LOGIC;
    signal acc_data_out_temp : UNSIGNED (7 downto 0);
    signal final_acc_data_temp : UNSIGNED (7 downto 0);
    signal acc_final_wr_temp : STD_LOGIC;
    signal acc_rd_temp : STD_LOGIC;
    signal acc_wr_temp : STD_LOGIC;

begin

    controller:  reu_base port map (rst => rst,
                                    op_code => op_code,
                                    src_data   => src_data,
                                    ctr_flag => ctr_flag,

                                    reg_index => reg_index_temp,
                                    rd => reg_rd_temp,
                                    wr => reg_wr_temp,
                                    result_wr  => result_wr_temp,
                                    acc_final_wr    => acc_final_wr_temp,
                                    acc_rd          =>acc_rd_temp,
                                    acc_wr    => acc_wr_temp,
                                    acc_data_out => acc_data_out_temp,
                                    alu_flag_out => alu_flag_temp,

                                    data_reg => in_data_temp,
                                    exe_state => alu_state_temp,
                                    src_3   => src_3_temp,
                                    src_cy  => src_cy_temp);

   reg_file: rfile port map (rst => rst,
                                    reg_index  => reg_index_temp,
                                    in_data    => in_data_temp,
                                    out_data   => data_from_reg_temp,
                                    result_in_data =>dest_reg_temp,
                                    result_wr => result_wr_temp,
                                    rd         => reg_rd_temp,
                                    wr         => reg_wr_temp);

     alu_comp: alu port map (   rst => rst,
                                    alu_state  => alu_state_temp,
                                    src_1   => data_from_acc_temp,
                                    src_2   => data_from_reg_temp,
                                    src_3   => src_3_temp,
                                    src_cy  => src_cy_temp,
                                    alu_flag => alu_flag_temp,
                                    dec_reset => dec_reset,
```

```
                                des_acc   => final_acc_data_temp,
                                des_reg => dest_reg_temp,
                                des_out => dest,
                                des_cy  => des_cy,
                                des_ac  => des_ac ,
                                des_ov  => des_ov );

  reg_acc: acc_reg port map(rst       => rst,
                                acc_data  => acc_data_out_temp,
                                final_acc_data  => final_acc_data_temp,
                                acc_final_wr    => acc_final_wr_temp,
                                acc_rd          =>acc_rd_temp,
                                acc_wr    => acc_wr_temp,
                                acc_data_out    => data_from_acc_temp  );

end BHV;

-- end of file --
```

REFERENCES

[1] F. Hu, S. Lakdawala, Q. Hao, M. Qiu, "Low-Power, Intelligent Sensor Hardware Interface for Medical Data Pre-Processing," IEEE Transactions on Information Technology in Biomedicine, vol.13, no.4, pp. 656-663, July 2009.

[2] S. Hariharan, N. B. Shroff; "Maximizing Aggregated Information in Sensor Networks Under Deadline Constraints," IEEE Transactions on Automatic Control, vol.56, no.10, pp.2369-2380, Oct. 2011.

[3] J. Rabaey and M. Pedram, Low Power Design Methodologies. Norwell, MA: Kluwer Academic Publishers, pp. 21-24,1996.

[4] M. Keating, D. Flynn; R. Aitken; A. Gibbons; K. Shi; "Low Power Methodology Manual: For System-on-Chip Design", *Springer*, 2007.

[5] I. F. Akyildiz et al., "A Survey on Sensor Networks," IEEE Communications Mag., vol. 40, no. 8, pp. 102–14, Aug. 2002.

[6] M. Hempstead, M. J. Lyons, D. Brooks and G,-Y Wei, "Survey of Hardware Systems for Wireless Sensor Networks," Journal of Low Power Electronics (JOLPE), Vol. 4., No. 1, April 2008.

[7] V. Ekanayake, C. Kelly, R. Manohar, "An ultra low-power processor for sensor networks". Proceedings of the 11th International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 27–36, Oct. 2004.

[8] V. Ekanayake, C. Kelly, and R. Manohar, "BitSNAP: Dynamic significance compression for a low-energy sensor network asynchronous processor". Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pp.144-154, March 2005.

[9] C. Kelly,V. Ekanayake, and R. Manohar, "SNAP: A sensor network asynchronous processor", Proceedings of the 9th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 24- 33, May 2003.

[10] L. Necchi, L. Lavagno , D. Pandini , L. Vanzago, "An ultra-low energy asynchronous

processor for Wireless Sensor Networks", Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems, pp.8 pp.-85, March 2006.

[11] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, T. N. Vijaykumar, "Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories". IEEE International Symposium on Low Power Electronics and Design (ISLPED), pp. 90- 95, June 2000.

[12] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, D. Brooks, "An ultra low power system architecture for sensor network applications". The 32nd Annual International Symposium on Computer Architecture (ISCA), pp. 208- 219, June 2005.

[13] S. Hanson, B. Zhai, M. Seok, B. Cline, K. Zhou, M. Singhal, M. Minuth, J. Olson, L. Nazhandali, T. Austin, D. Sylvester and D. Blaauw, "Performance and variability optimization strategies in a sub-200 mV, 3.5 pJ/inst, 11 nW subthreshold processor" IEEE Symposium on VLSI Circuits, pp.152-153, June 2007.

[14] L. Nazhandali, B. Zhai, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, T. Austin, D. Blaauw, "Energy optimization of subthreshold-voltage sensor network processors," The 32nd Annual International Symposium on Computer Architecture (ISCA), pp. 197-207, June 2005.

[15] B. Zhai, L. Nazhandali, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, D. Blaauw, T. Austin, "A 2.60pJ/Inst subthreshold processor for optimal energy efficiency," IEEE Symposium on VLSI Circuits, pp. 154-155, June 2006.

[16] S. Preradovic, N. Karmakar, "Modern RFID Readers", Microwave Journal, pp. 85-97, 2007.

[17] R. Imura, "The World's Smallest RFID u-Chip, brings about new business and lifestyles", Symposium on VLSI Circuits. Digest of Technical Papers., pp. 120-123, June 2004.

[18] A. Juels, "RFID security and privacy: A research survey" IEEE Journal on Selected Areas in Communications, vol.24, no.2, pp. 381-394, 2006.

[19] [Online] EPCglobal Website: http://www.epcglobalinc.org/ last accessed, Dec 2011.

[20] S. Sarma, M.H. Mickle, D. McFarlane, P. Cole, D.W. Engels, Guest Editorial, "Special Section on RFID." IEEE Transactions on Automation Science and Engineering, vol.6, no.1, pp. 1-3, 2009.

[21] S. Wanggen, Z. Yiqi, *et al*, "Design of an ultra-low-power digital processor for passive UHF RFID tags," Journal of Semiconductors, vol.30, no. 4, April 2009.

[22] J. W. Lee, H. Kwon, B. Lee, "Design consideration of UHF RFID tag for increased reading Range," IEEE MTT-S International Microwave Symposium Digest, pp.1588-1591, June 2006.

[23] O. B. Akan, M. T. Isik, B. Baykal, "Wireless Passive Sensor Networks," IEEE Communications Magazine, vol. 47, no. 8, pp. 92-99, August 2009.

[24] R. D. Fernandes, N. B. Carvalho, J. N. Matos, "Design of a battery-free wireless sensor node," IEEE International Conference on Computer as a Tool (EUROCON), pp.1-4, April 2011.

[25] N. Cho *et al.* "A 8-uw, 0.3-mm RF-powered transponder with temperature sensor for wireless environment monitoring", *IEEE* International Symposium of Circuits and Systems, vol.5, pp. 4763–4766, May 2005.

[26] M. T. Isik and O. B. Akan, "PADRE: Modulated Backscattering- based PAssive Data REtrieval in Wireless Sensor Networks," Proc. IEEE Wireless Communications and Networking Conference (WCNC), pp.1-6, April 2009.

[27] A. Bereketli, O. B. Akan, "Communication coverage in wireless passive sensor networks," IEEE Communications Letters, vol.13, no.2, pp.133-135, Feb. 2009.

[28] R. D. Fernandes, A. S. Boaventura, N. B. Carvalho, J. N. Matos, "Increasing the range of wireless passive sensor nodes using multisines," IEEE International Conference on RFID-Technologies and Applications (RFID-TA), pp.549-553, Sept. 2011.

[29] M. Philipose et al., "Battery-free wireless identification and sensing," IEEE Pervasive Computing, pp. 37–45, Jan.-March 2005.

[30] R. Amirtharajah and A. P. Chandrakasan, "Self-powered signal processing using vibration -based power generation," IEEE Journal of Solid-State Circuits, vol.33, pp.687-695, 1998.

[31] Y. Ammar, A. Buhrig, M. Marzencki, B. Charlot, S. Basrour, K. Matou, M. Renaudin, "Wireless sensor network node with asynchronous architecture and vibration harvesting micro power generator," Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies. ACM, Oct. 2005.

[32] H. Stockman, "Communication by Means of Reflected Power," Proc. I.R.E., vol. 36, pp.1196–1204, Oct. 1948.

[33] A. S. W. Man, E. S. Zhang, H. T. Chan, V. K. N. Lau, C. Y. Tsui, "Design and Implementation of a Low-power Baseband-system for RFID Tag", IEEE International Symposium on Circuits and Systems (ISCAS), pp.1585-1588, May 2007.

[34] X. Yang, J. Huang, X. Feng, J. Shen, Y. Qi, X. Wang , "Novel baseband processor for ultra-low-power passive UHF RFID transponder," IEEE International Conference on RFID-Technology and Applications (RFID-TA) , pp.141-147, June 2010.

[35] N. Cho, S.-J. Song, S. Kim, S. Kim, H.-J. Yoo, “A 5.1-μW UHF RFID tag chip integrated with sensors for wireless environmental monitoring,” Proceedings of the 31st European Solid-State Circuits Conference (ESSCIRC 2005), pp. 279- 282, Sept. 2005.

[36] J. Yin, J. Yi, et al., “A system-on-chip EPC Gen-2 passive UHF RFID tag with embedded temperature sensor,” IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), pp.308-309, Feb. 2010.

[37] R. S. Joshua, S. Alanson, P. Pauline, M. Alexander, S. Roy. “A wirelessly powered platform for sensing and computation”, Proceedings of the 8th International Conference on Ubiquitous Computing, pp. 495-506, Sept. 2006,

[38] A. P. Sample, D. J. Yeager, P. S. Powledge, J. R. Smith, “Design of a Passively-Powered, Programmable Sensing Platform for UHF RFID Systems”, IEEE International Conference on RFID, pp.149-156, March 2007.

[39] M. Flynn, "Some Computer Organizations and Their Effectiveness". IEEE Trans. Comput. **C-21**: 948, 1972.

[40] S. Dontharaju, S. Tung, A. K. Jones, L. Mats, J. Panuski, J. T. Cain, M. H. Mickle, The unwinding of a protocol, IEEE Communications Magazine, vol.45, no.4, pp.4-10, April 2007.

[41] V. Sai, A. Ogirala, and M. H. Mickle, "Low Power Radio Frequency Identification Design Using Custom Asynchronous Passive Computer" Journal of Low Power Electronics (JOLPE), Vol. 6, N° 4, December 2010.

[42] V. Sai, A. Ogirala, and M. H. Mickle, “Low Power Solutions for Wireless Passive Sensor Network (WPSN) Node Processor Architecture”, Chapter 23 in “Intelligent Sensor Networks: Across Sensing, Signal Processing, and Machine Learning,” Fei Hu and Qi Hao, editors, Taylor & Francis LLC, CRC Press, 2012.

[43] V. Sai, A. Ogirala, and M. H. Mickle, “A Low-Power Wireless Distributed Dynamic Network Design Concept: FFT Processor Architecture”, Communications, ACTA Press, Vol. 1, Issue 1, 2012.

[44] G. H. Barnes, R. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, “The Illiac IV Computer”, IEEE Transactions on Computers, C(17):746-757, Aug. 1968.

[45] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, “Energy- efficient Communication Protocols for Wireless Microsensor Networks,” Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, vol.2, pp.4-7, Jan. 2000.

[46] Y. Li; Y. Lian; V. Perez, "Design optimization for an 8-bit microcontroller in wireless biomédical sensors," IEEE Biomedical Circuits and Systems Conference (BioCAS), pp. 33-36, Nov. 2009.

[47] S. Saponara, L. Fanucci, A. J. Morello, “Power Optimization of digital IP Macrocells for Embedded Controls Systems”, IEEE International Conference on Industrial Technology, vol. 3, pp. 1617 – 1620, Dec. 2004.

[48] F. Iozzi, S. Saponara, A. J. Morello, L. Fanucci, “8051 CPU Core Optimization for Low Power at Register Transfer Level”, PhD Research in Microelectronics and Electronics, vol. 2, pp. 178 – 181, July 2005.

[49] S. Saponara, L. Fanucci, and P. Terreni, “Architectural-Level Power Optimization of Microcontroller Cores in Embedded Systems”, IEEE Transactions On Industrial Electronics, vol. 54, No. 1, pp. 680 – 683, February 2007.

[50] S. Mysore, B. Agrawal , F. T. Chong , Timothy Sherwood, “Exploring the Processor and ISA Design for Wireless Sensor Network Applications,” Proceedings of the 21st International Conference on VLSI Design, pp.59-64, Jan. 2008.

[51] D. Cotroneo, A. Migliaccio, S. Russo, “Reliable Monitoring of Network-related Performance Parameters in Wireless Environments,” Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), pp. 271-278, Feb. 2005.

[52] M. Mi, “RFID radio circuit design in CMOS,” Technical Report, Ansoft Corporation, Pittsburgh, Pennsylvania, USA, 2006.

[53] V. Sai, A. Ogirala, and M. H. Mickle, “A Low-Power Pulse Width Coding Scheme for Communication Receiver Systems”, Communications, ACTA Press, Vol. 1, Issue 1, 2012.

[54] EPC Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocols for Communications at 860MHz–960MHz, Ver 1.1.0, EPCGlobal, 2005, available: http://www. gs1.org/gsmp/kc/epcglobal/uhfc1g2

[55] R. Barnett, G. Balachandran, S. Lazar, B. Krarner, G. Konnail, S. Rajasekhar, and V. Drobny, “A passive UHF RFID transponder for EPC Gen 2 with -14dbm sensitivity in 0.13am CMOS,” IEEE International Conf. Solid-State Circuits, pp. 582–623, Feb 2007.

[56] V. Sai, A. Ogirala, and M. H. Mickle, "Low-Power Data Driven Symbol Decoder for UHF Passive RFID Tag," Journal of Low Power Electronics (JOLPE) - Vol. 8 N° 1, February 2012.

[57] V. Sai, A. Ogirala, and M.H. Mickle, “Serial Data Driven Cyclic Redundancy Check Generator for low power RFID Applications” Journal of Low Power Electronics (JOLPE), Vol. 8, N° 5, December 2012.

[58] N. Chabini, W. Wolf, “An approach for reducing dynamic power consumption in synchronous sequential digital designs,” Proceedings of the ASP-DAC, pp. 198-204, Jan. 2004.

[59] Arizona State University, “Predictive Technology Model (PTM)”, Online available at: http://ptm.asu.edu/

[60] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, Digital Integrated Circuits, 2nd edn., Prentice Hall, 2002.

[61] A. Shebaita, Y. Ismail “Lower power, lower delay Design scheme for CMOS Tapered Buffers”, Design & Test Workshop (IDT), pp.1-5, 2009.

[62] D. Sharma and R. Mehra "Low Power, Delay Optimized Buffer Design using 70nm CMOS Technology", International Journal of Computer Applications, vol. 22, issue 3, pp. 13-18, 2011.

[63] D. Caucheteux, E. Beigné, E. Crochon, M. Renaudin, “AsyncRFID: Fully Asynchronous Contactless Systems, Providing High Data Rates, Low Power and Dynamic Adaptation,” Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pp.86-97, March 2006.

[64] J. Kessels, T. Kramer, G. d. Besten, A. Peeters, V. Timm, “Applying Asynchronous Circuits in Contactless Smart Cards”, Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 36-44, April 2000.

[65] A. Abrial, J. Bouvier, P. Senn, M. Renaudin, P. Vivet, “A New Contactless Smart Card IC using an On-Chip Antenna and an Asynchronous Microcontroller”, IEEE Journal of Solid-State Circuits, vol. 36, n° 7, pp. 1101-1107, July 2001.

[66] P.-L. Siu, C.-S. Choy, C. F. Chan, K.-P. Pun, “A Contactless Smartcard Designed with Asynchronous Circuit Technique”, Proceedings of the 29th European Solid-State-Circuits Conference (ESSCIRC), pp. 213-216, Sept. 2003.

[67] C. H. van Berkel, M. B. Josephs, and S. M. Nowick, “Special issue on asynchronous circuits and systems,” eds., Proceedings of the IEEE, vol.87, Issue 2, Feb. 1999.

[68] S. Hauck, “Asynchronous design methodologies: An overview,” Proceedings of the IEEE, vol.83, Issue 1, pp.69-93, Jan. 1995.

[69] T. Nanya, “Challenges to dependable asynchronous processor design,” In T. Sasao, editor, Logic Synthesis and Optimization, chapter 9, pages 191–213. Kluwer Academic Publishers, 1993.

[70] V. Sai, A. Ogirala, and M. H. Mickle, "Implementation of an Asynchronous Low-Power Small-Area Passive Radio Frequency Identification Design Using Synchronous Tools

For Automation Applications," Journal of Low Power Electronics (JOLPE), Vol. 8, N° 4, August 2012.

[71] Oregano Systems, “MC8051 IP core User Guide” (Version 1.3) [Online]. Available: http://www.oreganosystems.at/?page_id=96