2005

DEVELOPMENT AND VALIDATION OF A SPECIAL PURPOSE SENSOR AND PROCESSOR SYSTEM TO CALCULATE EQUILIBRIUM MOISTURE CONTENT OF WOOD

Phani Tangirala

University of Kentucky

ABSTRACT OF THESIS

DEVELOPMENT AND VALIDATION OF A SPECIAL PURPOSE SENSOR AND PROCESSOR SYSTEM TO CALCULATE EQUILIBRIUM MOISTURE CONTENT OF WOOD

Percent Moisture Content (MC %) of wood is defined to be the weight of the moisture in the wood divided by the weight of the dry wood times 100%. Equilibrium Moisture Content (EMC), moisture content at environmental equilibrium is a very important metric affecting the performance of wood in many applications. For best performance in many applications, the goal is to maintain this value between 6% and 8%. EMC value is a function of the temperature and the relative humidity of the surrounding air of wood. It is very important to maintain this value while processing, storing or finishing the wood. This thesis develops a special purpose sensor and processor system to be implemented as a small hand-held device used to sense, calculate and display the value of EMC of wood depending on surrounding environmental conditions. Wood processing industry personnel would use the hand-held EMC calculating and display device to prevent many potential problems that can show significant affect on the performance of wood.

The design of the EMC device requires the use of sensors to obtain the required inputs of temperature and relative humidity. In this thesis various market available sensors are compared and appropriate sensor is chosen for the design. The calculation of EMC requires many arithmetic operations with stringent precision requirements. Various arithmetic algorithms and systems are compared in terms of meeting required arithmetic functionality, precision requirements, and silicon implementation area and gate count, and a suitable choice is made. The resulting processor organization and design is coded in VHDL using the Xilinx ISE 6.2.03i tool set. The design is synthesized, validated via VHDL virtual prototype simulation, and implemented to a Xilinx Spartan2E FPGA for experimental hardware prototype testing and evaluation. It is tested over various ranges of temperature and relative humidity. Comparison of experimentally calculated EMC values with the theoretical values of EMC derived for corresponding temperature and relative humidity points resulted in validation of the EMC processor architecture, functional performance and arithmetic precision requirements.

KEYWORDS: Equilibrium Moisture Content, Sensors, FPGA prototype, Temperature, Relative humidity, Special Purpose Processor.

Phani Tangirala

April 15, 2005

DEVELOPMENT AND VALIDATION OF A SPECIAL PURPOSE SENSOR AND PROCESSOR SYSTEM TO CALCULATE EQUILIBRIUM MOISTURE CONTENT OF WOOD

By

Phani Tangirala

Dr. J. Robert Heath
(Director of Thesis)

Dr. Yu Ming Zhang
(Director of Graduate Studies)

April 15, 2005

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate school of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name Date

THESIS

Phani Tangirala

The Graduate School

University of Kentucky

2005

DEVELOPMENT AND VALIDATION OF A SPECIAL PURPOSE SENSOR AND PROCESSOR SYSTEM TO CALCULATE EQUILIBRIUM MOISTURE CONTENT OF WOOD

THESIS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering at the University of Kentucky

By

Phani Tangirala

Lexington, Kentucky

Director: Dr. J. Robert Heath, Associate Professor
Electrical and Computer Engineering, Lexington, Kentucky

2005

MASTER'S THESIS RELEASE

I authorize the University of Kentucky Libraries to reproduce this thesis in whole or in part for purposes of research.

Signed: _____

Date: _____

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere appreciation and thanks to my committee members. Great thanks to my advisor, Dr. Heath for his great help and guidance and also the inspiration he provided me at difficult times. Similarly, my co-advisor Dr. Radun has always given me great guidance and support. My special thanks to my committee member, Associate Professor of Department of Forestry, Dr. Conners for identifying and delineating the thesis problem and giving me a chance to work on this topic. Without their assistance, support and encouragement this work would have been impossible.

Finally, I am forever indebted to my family members for their understanding, endless patience and encouragement when it was most required.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The moisture content of wood is one of the most important variables influencing the performance of wood. Unlike many other industries, the wood industries define percent moisture content (MC %) as the weight of the moisture in the wood divided by the weight of the dry wood times 100% [1]. The amount of water contained in wood not only influences its strength and stiffness but also affects its dimensions, its susceptibility to fungal attack, its workability and its ability to accept adhesives and finishes. For better performance of wood, in many applications, the moisture content must be reduced to at least six to eight percent of its dry mass.

The moisture content of wood is responsive to the relative humidity and temperature conditions in which it is stored. Like a freshly-washed cotton sheet hanging in the sun, the moisture contained in a piece of wood will come to equilibrium with its storage conditions. The moisture content at equilibrium conditions is termed the Equilibrium Moisture Content, or EMC.

This chapter explains the importance of Equilibrium Moisture Content (EMC) to the wood industries and the motivation behind the design of a processor to calculate an EMC value from the ambient conditions in which wood is kept. The basic design requirements including accuracy and the range of values over which the device is required to operate are also given.

1.1 Equilibrium Moisture Content:

When a tree is felled, moisture exists in its wood in two modes. First, it is present in cell cavities of wood (both within the central void called a lumen and also within smaller voids within the cell wall), which is known as free water. In the second mode, the water is chemically bonded to constituents of the cell wall. This type of water is known as

bound water [2]. Figure 1-1 shows a cellular cross-section of wood with water content in different locations.

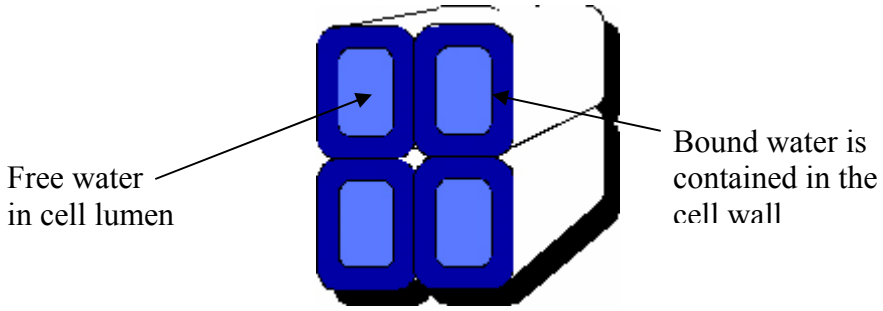


Figure 1-1: Section of wood cell showing water in liquid and chemically bound states.

When freshly felled wood is dried, the free water leaves the wood first because its removal requires much lower energy levels. This occurs without any dimensional changes to the wood because the moisture content of the cell wall itself does not change. The moisture loss continues until all the free water is released. This point, where the free water is gone but the cell wall remains saturated is known as the Fiber Saturation Point. The fiber saturation point varies a little with each type of wood, but it is generally considered to be at a moisture content of 25% to 30%. The drying of free water in cellular cross-section of wood is shown in Figure 1-2.

Figure 1-2: Section of wood cell showing the drying of free water.

To remove bound water requires higher energy levels than the removal of free water and the loss of moisture occurs more slowly compared to that of free water. Moisture loss from the cell walls also affects the dimensions of the wood causing shrinkage. The

strength of wood increases as the drying continues. Figure 1-3 shows a cellular cross-section of wood losing its bound water.

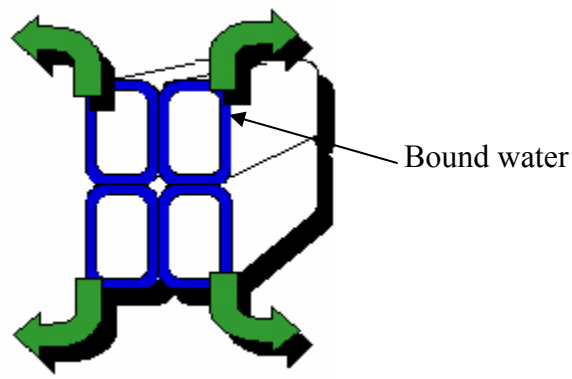


Figure 1-3: Section of wood cell showing the drying of bound water

The moisture content of wood below the fiber saturation point depends on the relative humidity and temperature of the surrounding air. If wood remains long enough in air where the temperature and relative humidity are constant, the moisture content also becomes constant at a value known as Equilibrium Moisture Constant (EMC). The number of water molecules (bound water) increase or decrease so that the vapor pressure of the wood is in equilibrium with that of the air surrounding it. The particular value of moisture content which is in equilibrium with a given relative humidity/temperature combination is EMC. The EMC at a constant temperature will increase as the relative humidity of the surrounding air increases and decreases with decreasing relative humidity. The EMC value in most parts of the United States is approximately 6%-8% [3].

The relationship between EMC, relative humidity and temperature is shown in Table 1-1. The values shown in Table 1-1 apply to most kind of species of wood for most practical purposes. These values can be approximated by the following Equation 1-1 (the equation and the information in Table 1-1 are from the USDA Wood Handbook, 1999 [4]).

$$EMC = \frac{1800}{W}\left[\frac{Kh}{1 - Kh} + \frac{K_1 Kh + 2K_1 K_2 K^2 h^2}{1 + K_1 Kh + K_1 K_2 K^2 h^2}\right] \quad (1\text{-}1)$$

EMC is moisture content in percent, and *h* is the fractional relative humidity. For temperature *T* in Celsius degrees,

$$W = 349 + 1.29T + 0.0135\,T^2$$
$$K = 0.805 + 0.000736\,T - 0.00000273\,T^2$$
$$K_1 = 6.27 - 0.00938\,T - 0.000303\,T^2$$
$$K_2 = 1.91 + 0.0407\,T - 0.000293\,T^2$$

For temperature *T* in Fahrenheit degrees,

$$W = 330 + 0.453T + 0.00415\,T^2$$
$$K = 0.791 + 0.000463\,T - 0.000000844\,T^2$$
$$K_1 = 6.34 + 0.000775\,T - 0.0000935\,T^2$$
$$K_2 = 1.09 + 0.0284\,T - 0.0000904\,T^2$$

In the above equation, T is the temperature in the surrounding atmosphere, h is the relative humidity (%/100), and EMC is the Equilibrium Moisture Content (%).

Relative humidity

° F	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
30	1.4	2.6	3.7	4.6	5.5	6.3	7.1	7.9	8.7	9.5	10.4	11.3	12.4	13.5	14.9	16.5	18.5	21	24.3
40	1.4	2.6	3.7	4.6	5.5	6.3	7.1	7.9	8.7	9.5	10.4	11.3	12.3	13.5	14.9	16.5	18.5	21	24.3
50	1.4	2.6	3.6	4.6	5.5	6.3	7.1	7.9	8.7	9.5	10.3	11.2	12.3	13.4	14.8	16.4	18.4	20.9	24.3
60	1.3	2.5	3.6	4.6	5.4	6.2	7	7.8	8.6	9.4	10.2	11.1	12.1	13.3	14.6	16.2	18.2	20.7	24.1
70	1.3	2.5	3.5	4.5	5.4	6.2	6.9	7.7	8.5	9.2	10.1	11	12	13.1	14.4	16	17.9	20.5	23.9
80	1.3	2.4	3.5	4.4	5.3	6.1	6.8	7.6	8.3	9.1	9.9	10.8	11.7	12.9	14.2	15.7	17.7	20.2	23.6
90	1.2	2.3	3.4	4.3	5.1	5.9	6.7	7.4	8.1	8.9	9.7	10.5	11.5	12.6	13.9	15.4	17.3	19.8	23.3
100	1.2	2.3	3.3	4.2	5	5.8	6.5	7.2	7.9	8.7	9.5	10.3	11.2	12.3	13.6	15.1	17	19.5	22.9
110	1.1	2.2	3.2	4	4.9	5.6	6.3	7	7.7	8.4	9.2	10	11	12	13.2	14.7	16.6	19.1	22.4
120	1.1	2.1	3	3.9	4.7	5.4	6.1	6.8	7.5	8.2	8.9	9.7	10.6	11.7	12.9	14.4	16.2	18.6	22
130	1	2	2.9	3.7	4.5	5.2	5.9	6.6	7.2	7.9	8.7	9.4	10.3	11.3	12.5	14	15.8	18.2	21.5
140	0.9	1.9	2.8	3.6	4.3	5	5.7	6.3	7	7.7	8.4	9.1	10	11	12.1	13.6	15.3	17.7	21
150	0.9	1.8	2.6	3.4	4.1	4.8	5.5	6.1	6.7	7.4	8.1	8.8	9.7	10.6	11.8	13.1	14.9	17.2	20.4
160	0.8	1.6	2.4	3.2	3.9	4.6	5.2	5.8	6.4	7.1	7.8	8.5	9.3	10.3	11.4	12.7	14.4	16.7	19.9
170	0.7	1.5	2.3	3	3.7	4.3	4.9	5.6	6.2	6.8	7.4	8.2	9	9.9	11	12.3	14	16.2	19.3
180	0.7	1.4	2.1	2.8	3.5	4.1	4.7	5.3	5.9	6.5	7.1	7.8	8.6	9.5	10.5	11.8	13.5	15.7	18.7
190	0.6	1.3	1.9	2.6	3.2	3.8	4.4	5	5.5	6.1	6.8	7.5	8.2	9.1	10.1	11.4	13	15.1	18.1
200	0.5	1.1	1.7	2.4	3	3.5	4.1	4.6	5.2	5.8	6.4	7.1	7.8	8.7	9.7	10.9	12.5	14.6	17.5
210	0.5	1	1.6	2.1	2.7	3.2	3.8	4.3	4.9	5.4	6	6.7	7.4	8.3	9.2	10.4	12	14	16.9
220	0.4	0.9	1.4	1.9	2.4	2.9	3.4	3.9	4.5	5	5.6	6.3	7	7.8	8.8	9.9			
230	0.3	0.8	1.2	1.6	2.1	2.6	3.1	3.6	4.2	4.7	5.3	6	6.7						
240	0.3	0.6	0.9	1.3	1.7	2.1	2.6	3.1	3.5	4.1	4.6								
250	0.2	0.4	0.7	1	1.3	1.7	2.1	2.5	2.9										
260	0.2	0.3	0.5	0.7	0.9	1.1	1.4												
270	0.1	0.1	0.2	0.3	0.4	0.4													

Temperature in Fahrenheit

Table 1-1: EMC values at different values of temperature and relative humidity.

1.2 Motivation:

Wood in service is generally exposed to both long-term and short-term changes in relative humidity and temperature of surrounding air. Thus wood is always undergoing at least slight changes in moisture content. It is dimensionally stable when the moisture content is greater than the fiber saturation point (i.e., when the cell walls are saturated) but it changes its dimensions below the fiber saturation point. These changes in dimensions can significantly affect the performance of wood. Wood shrinks when it loses moisture from cell walls, and swells when gaining moisture in the cell walls. This shrinking and swelling can result in warping, checking, splitting, and thereby affects the appearance and the performance of finished wood products. It is important to know the values of relative humidity and temperature (and therefore the EMC conditions) of any location where the wood products are used or stored. These values give relevant information about the conditions to be maintained while drying, processing and storing wood. By maintaining a proper environment wood will not have any significant dimensional changes and damage to finished products can be avoided.

In general while processing wood in industries, the air is regulated by dispensing steam or cool vapor mist into the atmosphere, thus achieving the required humidity conditions. It is difficult, however, to control the values of relative humidity and temperature of the surrounding air to maintain the correct levels unless relevant information on temperature, relative humidity and EMC is available. The usual manner of determining the EMC condition in industry is to first measure both the temperature and the relative humidity and then to go to a table to determine the EMC to which the ambient conditions correspond. Factory employees often are neglectful of EMC because it requires them to maintain and read thermometer/hygrometers and to consult a lookup table. In practice, therefore, EMC is a problem which is most often confronted after a problem is found to exist. A simpler device that displays EMC directly should be more useful in heading off potential environment problems before they can significantly affect the wood and cause problems with wood products. An electronic hand held device that can show the values of temperature, relative humidity of the surrounding air and also the value of EMC will

definitely be useful in these situations. If the conditions of EMC of the surrounding air are known through out the process, the finished wood product should have moisture content in the desired range, thereby making machining and finishing operations more predictable and allow avoidance of damage. Thus, maintaining proper conditions with the help of the electronic device will help reduce potential moisture-related damage, the cost of processing and time.

1.3 Problem Statement:

A hand held electronic device is needed to sense the values of temperature and relative humidity of the surrounding air and then calculate the value of EMC of wood. The devices' operating range of temperature is to be 30^{o}F to 210^{o}F. Its operating range of humidity is to be 0%RH to 100%RH. The accuracy needed for the design is 0.1° or 0.1 percent for all the values of temperature, humidity, and EMC. The final device is to be able to display the three values (temperature, relative humidity and EMC) depending on user control inputs.

The remaining chapters of this thesis first address appropriate sensor selection and design of sensor-processor interface. The primary design approach used to design the EMC processor and the details of the design are then presented. Finally, the results of post implementation VHDL simulation and experimental prototype testing are presented which resulted in validation of the sensor interface design and the EMC processor design, operation and performance.

Chapter 2

Primary Design Approach

This chapter describes the primary design approach for the design of the special purpose processor to calculate Equilibrium Moisture Content of wood. The equations on which the design is based are represented and the detailed requirements for the design are shown. This chapter also explains the various parameters considered for choosing the required components. The details of the CAD tools, HDL used for the design capture, post-synthesis simulation validation and post implementation simulation validation are also given.

2.1 Equations Used for the Design:

The Equilibrium Moisture Content (EMC) of wood is based on the values of temperature and relative humidity in the surrounding atmosphere. At a constant temperature as the value of relative humidity increases the value of EMC increases whereas at constant relative humidity value, the value of EMC decreases as the value of temperature increases. The equations (1-1) used to calculate EMC are repeated below for convenience.

$$EMC = \frac{1800}{W}\left[\frac{Kh}{1 - Kh} + \frac{K_1 Kh + 2K_1 K_2 K^2 h^2}{1 + K_1 Kh + K_1 K_2 K^2 h^2}\right] \quad (2\text{-}1)$$

For temperature T in Celsius,

$$W = 349 + 1.29T + 0.0135T^2$$
$$K = 0.805 + 0.000736T - 0.00000273T^2$$
$$K_1 = 6.27 - 0.00938T - 0.000303T^2$$
$$K_2 = 1.91 + 0.0407T - 0.000293T^2$$

And for temperature T in Fahrenheit,

$$W = 330 + 0.453T + 0.00415\,T^2$$
$$K = 0.791 + 0.000463\,T - 0.000000844\,T^2$$
$$K_1 = 6.34 + 0.000775\,T - 0.0000935\,T^2$$
$$K_2 = 1.09 + 0.0284\,T - 0.0000904\,T^2$$

In the above equation, T is the temperature of the surrounding atmosphere, h is the relative humidity (%/100), and *EMC* is the moisture content (%).

2.2 EMC Processor System Requirements and High-level Functional Organization:

From the Equation 2-1, it can be easily observed that the EMC of wood is dependent on temperature and relative humidity of the surrounding atmosphere. Therefore, the processor should have the temperature and relative humidity as its inputs. Equation set (2-1) shows that a processor is required to perform all the shown arithmetic functions required to obtain the required result of EMC. The final display should show the values of temperature, relative humidity and the corresponding EMC of wood depending on user control inputs.

From the above specifications the components needed for the design of the EMC calculator are:

1. Temperature and Relative humidity sensors: Sensors that can produce the value of temperature and relative humidity are required. The accuracy of the measured value should be to 0.1% precision to match the requirements of the design.
2. FPGA: A Field Programmable Gate Array (FPGA) device is needed for the design which can be programmed to implement the processor used to compute EMC for a given temperature and relative humidity.
3. Display device: A single 4-digit display device that can display the values of temperature, relative humidity and EMC. One digit to the right of the decimal point is required.

4. Power Source: A power source of either 5V or 3.3V that can be used for all the components of the design.

A high level functional diagram showing all the above components is shown in Figure 2-1.

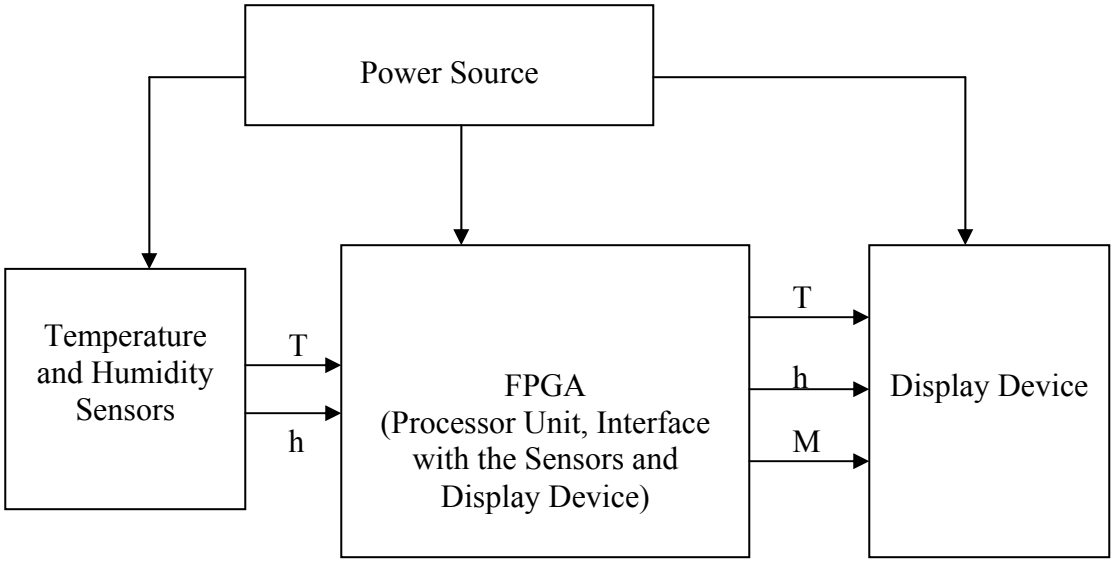


Figure 2-1: High level functional diagram of EMC processor.

2.3 Choosing Temperature and Relative Humidity Sensors:

The design of the wood moisture content (EMC) processor/calculator requires sensors which can measure both temperature and relative humidity values. The following sections describe various types of sensors available for both temperature and relative humidity and the parameters considered in choosing the sensors for the design.

2.3.1 Temperature Sensors:

Temperature sensors that are currently available can broadly be divided into two categories; analog output sensors and digital I/O sensors. The type of the output of

the sensor is considered as a parameter for choosing the sensor for the wood moisture content processor/calculator design.

Thermistors, RTDs (Resistance Temperature Detectors), thermocouples and Silicon temperature sensors are some of the analog output temperature sensors. These sensors produce their output in the form of a voltage level. These sensors are generally required to be followed by a comparator, an analog-to-digital converter, or an amplifier to make their output more useful in more common circuits. But, with the advent of high level integration of circuits, digital interfaces are available. Digital I/O sensors offer many advantages compared to that of analog output sensors. The output is in the form of 1's and 0's which make the circuits simpler. With digital output of the temperature, the comparators, analog-to-digital converters or amplifiers are not now required in circuits. The digital interface will generally be a serial interface protocol such as I^2C (Inter Integrated Circuit). Digital sensors offer the advantages of less complexity and less area compared to that of analog output sensors. Hence, a digital I/O sensor for the temperature would be an ideal choice for the design since such a sensor requires less area and is functionally simpler to integrate into the EMC processor design.

2.3.2 Relative Humidity Sensors:

Relative humidity reveals the percentage of the maximum amount of humidity present in air. The maximum amount of humidity is the upper limit of humidity air can hold at a given temperature. A very important factor in measuring relative humidity is temperature. If temperature rises or falls in a closed system, the saturation vapor pressure will increase or decrease resulting in the drop or rise of relative humidity. A slight change in temperature especially at high humidity has a significant effect on RH, since saturation pressure changes too [5].

The issues of localization and stabilization of sensors also make accurate measurement of relative humidity almost impossible. The most accurate measurements available on the market are from chilled mirror hygrometers. A mirror

is chilled down slowly until fog forms on it. The temperature of the mirror at which the fog is formed is known as the dew point. Relative humidity can be calculated from this value of temperature, but this technique is quite expensive and is not suited for hand held devices.

The remaining types of relative humidity sensors are capacitive sensors and resistive sensors. Capacitive type sensors consist of a substrate on which a thin film of polymer or metal oxide is deposited between two conductive electrodes. The sensing surface is coated with a porous metal electrode to protect it from contamination and exposure to condensation [6]. The substrate is typically glass, ceramic, or silicon. The incremental change in the dielectric constant of a capacitive humidity sensor is nearly directly proportional to the relative humidity of the surrounding environment. Resistive type humidity sensors generally measure the change in the electrical impedance of a hygroscopic medium such as a conductive polymer, salt, or treated substrate. The impedance change is in inverse exponential relation to humidity.

Capacitive sensors have many advantages compared to resistive sensors. Capacitive sensors provide more accurate relative humidity values and the values are around +/- 1.5% RH. Resistive type sensors provide lower accuracy values compared to that of capacitive type sensors, and the values are around +/- 5% RH. Resistive type humidity sensors can only be operated in non-condensing environments unlike capacitive type sensors. Resistive type sensors offer less range of relative humidity (20% to 80%) compared to that of capacitive type which offer 0% to 100% range. Furthermore, capacitive type sensors have more long term stability compared to that of resistive types.

Given all the advantages in measurement, and cost effectiveness capacitive type sensors are more suitable for the design of the wood moisture content calculator. A relative humidity sensor which produces digital output would be more advantageous compared to an analog capacitive type humidity sensor. Reasons for this are based on the complexity of the circuit and area of the final device.

2.4 SHT71 Humidity and Temperature Sensor:

From the discussion in the previous section on temperature and relative humidity sensors, it is clearly evident that digital temperature sensors and capacitive digital relative humidity sensors are best choices for the design. Different candidate sensors from different companies were considered in choosing sensors for the design. For example, Analog Devices produce digital I/O temperature sensors with a 2-wire serial interface. A list of some of the sensors considered in choosing the sensors for the design is given in Table 2-1. Table 2-1 also shows different capacitive type relative humidity sensors from different candidate companies.

Type of sensor	Name of manufacturer	Type of output and resolution	Range of output	Accuracy
Temperature	ANALOG DEVICES	10-bit digital	-40^{o}C to 125^{o}C	+/- 1^{o}C
Temperature	MAXIM	9-12 bit digital	-55^{o}C to 125^{o}C	+/- 0.5^{o}C
Temperature	SENSIRION	8-14 bit digital	-40^{o}C to 125^{o}C	+/- 0.4^{o}C
Temperature	PHILIPS	11- bit digital	-55 ^{o}C to 125^{o}C	+/-3 ^{o}C
Temperature	NATIONAL SEMICONDUCTORS	10- bit digital	-55 ^{o}C to 125^{o}C	+/- 1.5 ^{o}C
Relative humidity	HONEYWELL	Voltage output	0 % -100 % RH	+/- 2.0 % RH
Relative humidity	SENSIRION	8-12 bit digital	0 % - 100 % RH	+/- 3.0 % RH
Relative humidity	HUMIREL	Voltage output	0 % - 99 % RH	+/- 3.0 % RH

Table 2-1: Comparison of different temperature and relative humidity sensors

From Table 2-1, it can be observed that there are a wide range of devices that can be chosen for temperature sensors but there are very few relative humidity sensors that produce digital output. As given in the requirements of the device, the temperature range should be at least -1.1^{o}C to 100 ^{o}C. Many of the temperature sensors in the Table 2-1 can be operated in that range. The accuracy and the resolution of the Sensirion sensor is much better compared to that of others. From the capacitive type relative humidity sensors considered, the Sensirion sensor produces the required digital output with sufficient resolution and accuracy.

It would be really advantageous if a single sensor could provide both digital temperature and relative humidity measurements. This would avoid the use of comparators, or analog to digital converters, or amplifiers, and help reduce the complexity of the design and area of the final device. The SHT71 Sensirion humidity and temperature sensor shown in Figure 2-2 is one such device that offers a digital output of both temperature and relative humidity [7]. It uses a capacitive type sensor to measure relative humidity which allows it to have higher accuracy and required range. It produces fully calibrated digital output and it has high long term stability.

Figure 2-2: SHT71 Sensirion temperature/relative humidity sensor.

A functional block diagram representation of the SHT71 sensor is shown in Figure 2-3 [7].

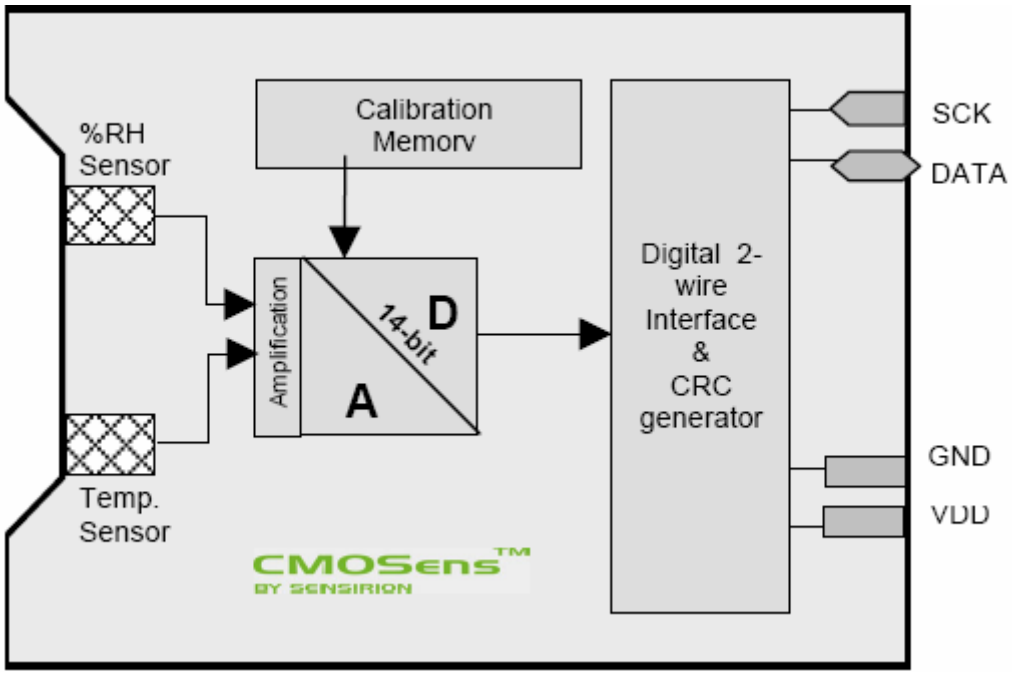


Figure 2-3: Block diagram of SHT71 humidity and temperature sensor.

The device uses a capacitive polymer sensing element for relative humidity measurement and a band gap temperature sensor for temperature measurement. Both sensors are connected to a 14-bit Analog to Digital converter and a serial interface circuit on the chip. The sensor can be calibrated in a precision humidity chamber with a chilled hygrometer as a reference. The calibration coefficients are programmed into the calibration memory. These coefficients are used internally during measurement.

From specifications given in the datasheet of the SHT71 sensor, features of the sensor can be summarized as follows.

- It has two sensors; one for relative humidity and another for temperature.
- Precise dew point calculation is possible because of the integrated temperature and humidity sensors on a single chip, which allows precise calculation of relative humidity.

- 0 % to 100 % RH ranges of relative humidity.
- -40^{o}C to 123.8 ^{o}C ranges of temperature.
- +/- 3 % RH accuracy (without non-linear compensation).
- +/- 0.4 ^{o}C temperature accuracy (without non-linear compensation).
- Low power consumption of around 30 µW.
- Low cost and low area.

The features discussed above would make the SHT71 a good choice for the design of the EMC calculator. It satisfies the requirement of 0.1 resolution required for the device and the required ranges of temperatures and relative humidity. The package diagram of SHT71 sensor is shown in Figure 2-4 [7]. From the measurements shown in the diagram the overall length of the sensor is ~ 19.5mm, and breadth is 5.08 mm. These measurements illustrate the advantage of the SHT71 sensor over other sensors in occupying minimal area on the final device. The accuracy values shown above are the values without non-linear compensation. These accuracies can be improved using some non-linear compensation that will be explained in Chapter 3.

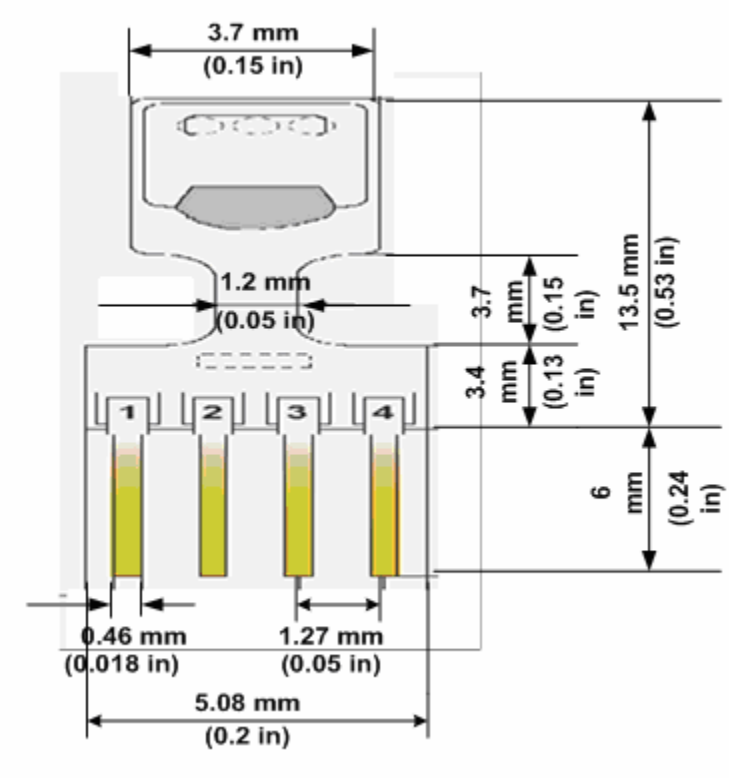


Figure 2-4: The package diagram of SHT71 sensor showing its measurements (all the values are in mm (inches))

2.5 FPGA Used for the Design:

A prototype of the design of the EMC processor would be implemented to a programmable integrated circuit (IC) for functional and performance testing. The processor would obtain required values of temperature and relative humidity from the sensor IC in proper format, and calculate the equation set (2-1) to obtain an EMC value. Field Programmable Gate Array (FPGA) devices are digital ICs that contain programmable blocks along with programmable interconnects between the programmable logic. The basic architecture of a FPGA chip is shown in Figure 2-5. The FPGA can be programmed to implement the required EMC processor and can be used in the final device. A detailed functional flow of how the FPGA can be programmed is shown in Chapter 5.

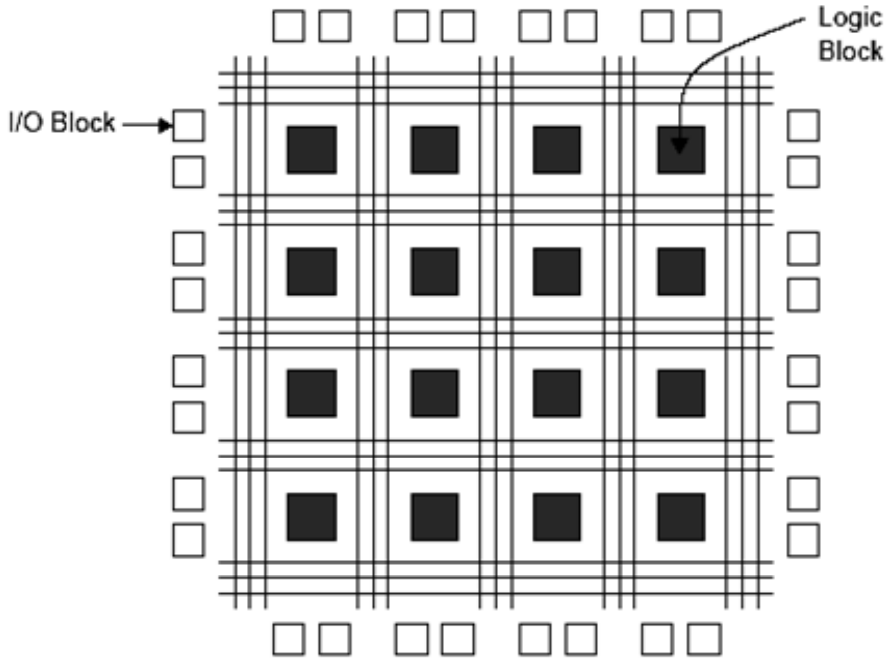


Figure 2-5: Structure of an FPGA

As shown in Figure 2-5, an FPGA chip consists of I/O Blocks and Logic Blocks. The design can be partitioned and mapped into the logic blocks by a CAD tool, and then

logic blocks needing to communicate and share signals are connected through interconnect switching matrices. The I/O blocks on the chip perimeter are used to drive signals off the chip as well as read signals coming into the chip from off the chip.

Various FPGA vendors (Xilinx, Altera, Lattice, Actel, Quicklogic etc,) are available in the market. A well-known reliable vendor is Xilinx. Xilinx produces various FPGA chip families such as Spartan2, Spartan2E, Spartan3, Virtex, Virtex2, Virtex2P, Virtex E etc. A Spartan2E FPGA was chosen as the programmable IC for this design. The Spartan2E has some advantages over other FPGAs. Though the basic architecture of Spartan2E FPGA is similar to that of the Virtex E, it is designed using 0.18µ process and is more cost optimized. It has four DLLs (Delay Locked Loops) that can be used to produce an accurate multiplied clock or divided clock. Its temperature operating range is from -40^{o}C to 125 ^{o}C. A Spartan2E XC2S200E is sufficient in size for the design. The design's equivalent gate count did not exceed the gate count of XC2S200E (200000, the maximum equivalent gate count of a system which may be mapped into the chip). The dimensions of Spartan2E XC2S200E device are shown in Figure 2-6. These dimensions are given in the package diagrams of Xilinx FPGAs.

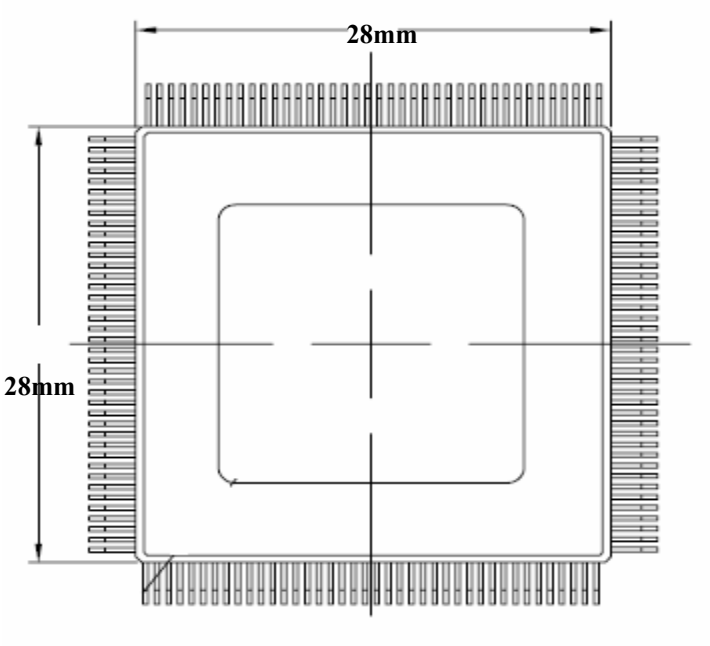


Figure 2-6: Dimensions of Xilinx Spartan2E FPGA

2.6 HDL and CAD Tools used for the Development and Validation of EMC Processor:

Very High Speed Integrated Circuit Hardware Description Language (VHDL) is used for the design capture. The design for the interface of the sensor and the processor to implement Equations (2-1) is coded in VHDL using the Xilinx 6.2.03i CAD tool set. The below design flow explains the sequence of operations in capturing the EMC processor design, validating the design via HDL simulation and then building the prototype of the design using VHDL and the Xilinx 6.2.03i tool set.

Design flow:

- Design description and capture using VHDL.
- Pre-Synthesis HDL simulation of the design using Modelsim XE-II 5.7g HDL simulator.
- Synthesizing the design using the Xilinx 6.2.03i tool set.
- Post-Synthesis HDL simulation for validation of EMC processor and sensor interface circuit design using Modelsim XE-II 5.7g HDL simulator.
- Implementing the design (map, place and route) to the selected FPGA.

- Post-place and route simulation of the implemented design for the validation of the design using the Modelsim tool. This simulation is a "virtual prototype" simulation and all the physical delays are included in the design to simulate the processor as implemented to the FPGA chip on the prototyping board.
- Generating a programming file in .bit format for FPGAs, which is used to program the processor and sensor interface into the FPGAs.
- Use of the Xilinx iMPACT tool to configure the FPGA.

Thus, VHDL is used to program the logic design for the interface of the sensor and the arithmetic unit required to perform all the arithmetic functions given in equations (2-1) into the Spartan2E FPGA. Chapter 3 will explain in detail the design of the sensor to processor interface module, along with the required protocols for communicating with the sensor and the required hardware to implement the protocols.

Chapter 3

Design of Sensor Interface

The design of the temperature and relative humidity sensor interface module is explained in this chapter. The steps required in communicating with the SHT71 Sensirion Temperature/Humidity sensor are explained. The properties of the sensor, such as temperature dependence of relative humidity and non linearity are explained and the equations used to compensate for the non-linearity are also shown. VHDL post-implementation simulation wave forms of the interface module showing all stages of serial transmission are also shown. These waveforms are used to evaluate and validate correct functional and performance operation of the sensor interface.

3.1 High Level Schematic of SHT71 Interface:

A high level schematic of the SHT71 sensor interface is shown in Figure 3-1. As shown in Figure 3-1, the SHT71 sensor has 4 pins. Pins '2' and '3' are connected to the power supply and ground respectively. The sensor needs a voltage supply between 2.4 V and 5.5 V. Pin '1' of the sensor is connected to serial clock ('SCK'), a clock signal generated by the FPGA, to synchronize the communication between the FPGA and the sensor. There is no minimum frequency of 'SCK' as the interface completely consists of static logic. Pin '4' is connected to the FPGA through the serial bi-directional data line. This serial data bus is used to transfer data in and out of the sensor.

As shown in Figure 3-1, a pull-up resistor is connected from the data bus to the power supply voltage Vdd. The data changes after the falling edge of 'SCK' and is valid on the rising edge of 'SCK'. The data must remain stable while 'SCK' is high. The data signal must not be driven high by the FPGA. The external pull-up resistor pulls the value on the data line high when required. When a value '0' is to be written to the sensor, the FPGA pulls down the value to zero. The value of the pull-up resistor is determined from the DC characteristics of the sensor.

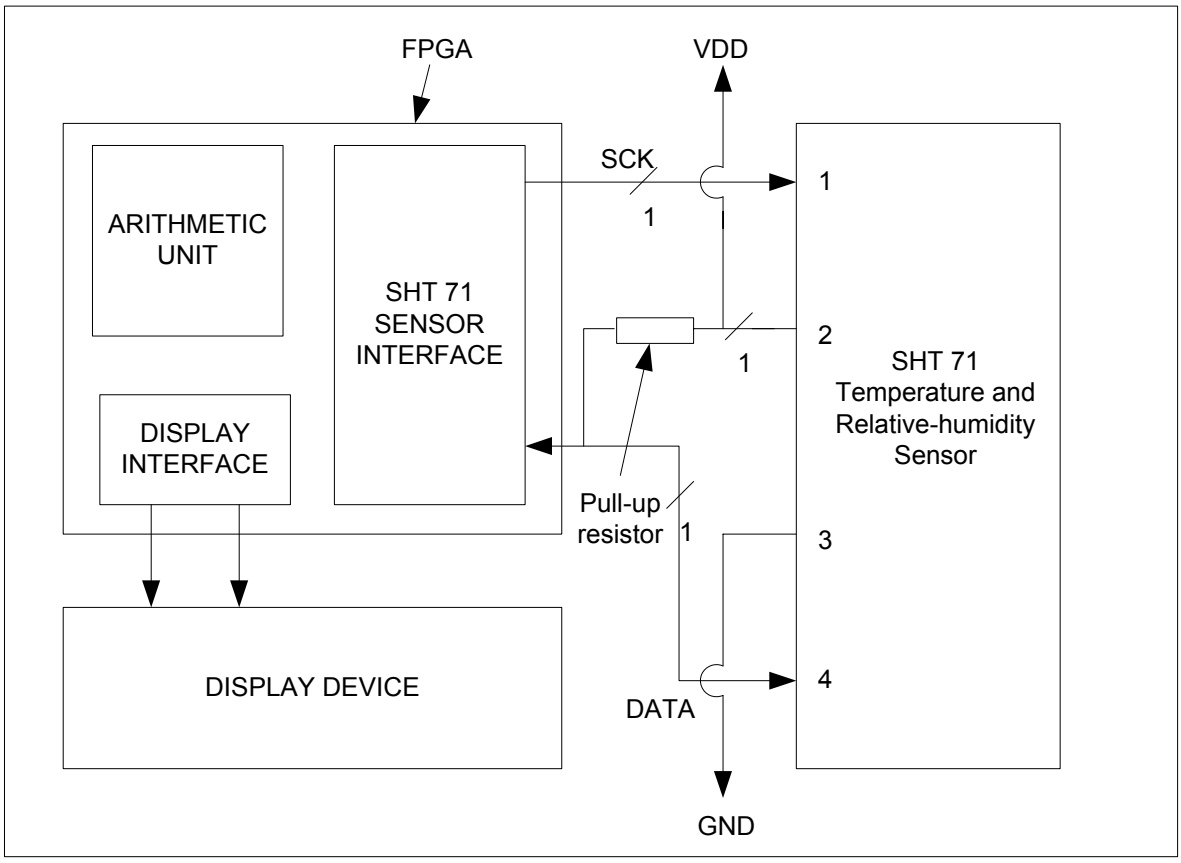


Figure 3-1: High level schematic of EMC sensor, processor, and display system showing SHT71 sensor interface.

3.2 Sensor Interface Specifications:

The two-wire ('SCK' and 'DATA' lines) serial interface of the sensor follows a set of specifications to measure the values of temperature and relative humidity [7]. The serial interface is optimized for sensor readout and power consumption. The steps followed in the serial interface protocol are as follows.

- 'Transmission start' sequence of DATA and SCK lines.
- Issue the command to measure temperature and relative humidity.
- Wait for the response from the sensor.
- Measure the value of temperature and relative humidity.

- Cyclic Redundancy Check (CRC) measurement.

First, the transmission start command is given to the sensor by the EMC processor. This command is followed by the measurement command from the EMC processor, where the processor specifies either temperature or humidity as the value to be measured. This state is followed by wait and measure states to obtain the required value. At the end of the transmission the data transmitted is checked for any CRC error. All the details of these commands are explained in sequence as follows.

1. Transmission start: The entire communication between the sensor and the FPGA is done using the two lines, DATA and SCK. To initiate a transmission, the FPGA has to lower the DATA line while SCK is high, followed by a low pulse on SCK and raising DATA again while SCK is again high. This sequence is shown in Figure 3-2.

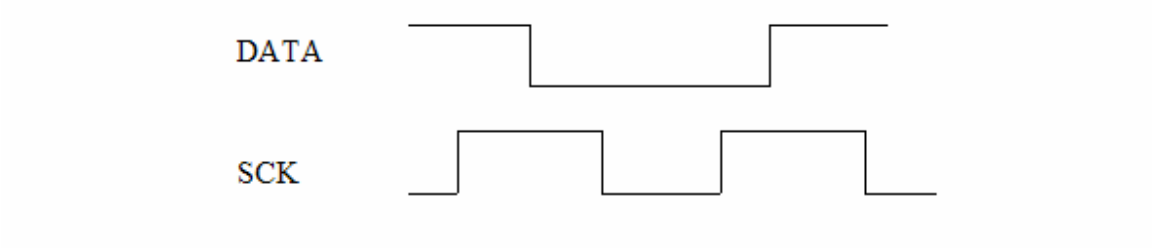


Figure 3-2: Transmission start sequence for SHT71 sensor.

2. Command for measurement: After issuing the transmission start sequence the FPGA has to send the measure command to the sensor. This is the state when the processor decides which command is to be sent. The commands used in this design are measure temperature and measure humidity. For measuring the temperature, the command that has to be sent on the data line is “00000011”. For measuring relative humidity, the command is “00000101”. As explained earlier, the data changes at the falling edge of the clock and is valid at the rising edge of the clock. After 8 SCK pulses, the FPGA should not drive the data line. At the 9^{th} SCK pulse, the sensor will bring down the data line to zero. This pulse is the ‘ack’ pulse, which is used as an acknowledge pulse. If the sensor does not bring the data line to zero, there is an error in the transmission. In such case, the transmission has to restart again by sending the command for measurement. The

sensor releases the data line after the 9th clock pulse. Figure 3-3 shows the DATA and SCK signals for the case of measuring temperature.

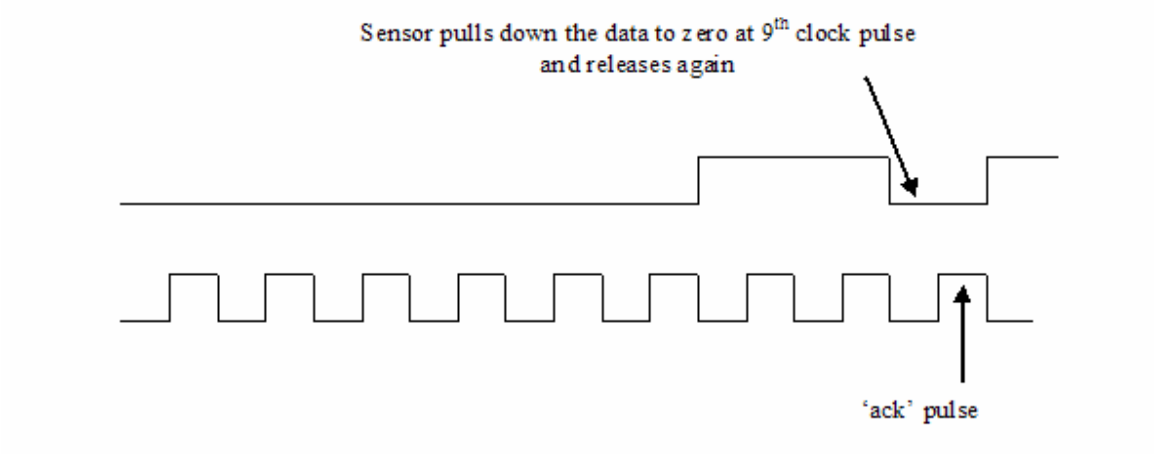


Figure 3-3: Command given to SHT71 sensor to measure temperature.

3. Wait state: After sending the command for measurement to the sensor, the interface controller of the processor has to wait until the measurement is complete. The time needed for measurement depends on the type of measurement used. The SHT71 sensor supports 8 bit/ 12 bit/14 bit measurements. Eight (8) bit measurements are used for the low accuracy applications. Twelve (12) bit and 14 bit measurements of humidity and temperature respectively are used for more accurate measurements. In this design, 12 bit measurement for humidity and 14 bit measurement for temperature are used. The time taken in wait state for a measurement is approximately 210 ms. The 'SCK' signal sent by the EMC processor has to be at logic low in wait state. And the DATA line should be high in the wait state, which means that the DATA line should not be driven by controller in wait state. The sensor signals the end of the wait state by pulling down the DATA signal to zero after 210 milli seconds. The measurement starts when SCK is toggled again after this state. The measured data is stored in sensor until the SCK is toggled. Hence the interface controller of the processor can continue with other tasks and read the data out whenever it is possible. Figure 3-4 shows the DATA and SCK signals for wait state.

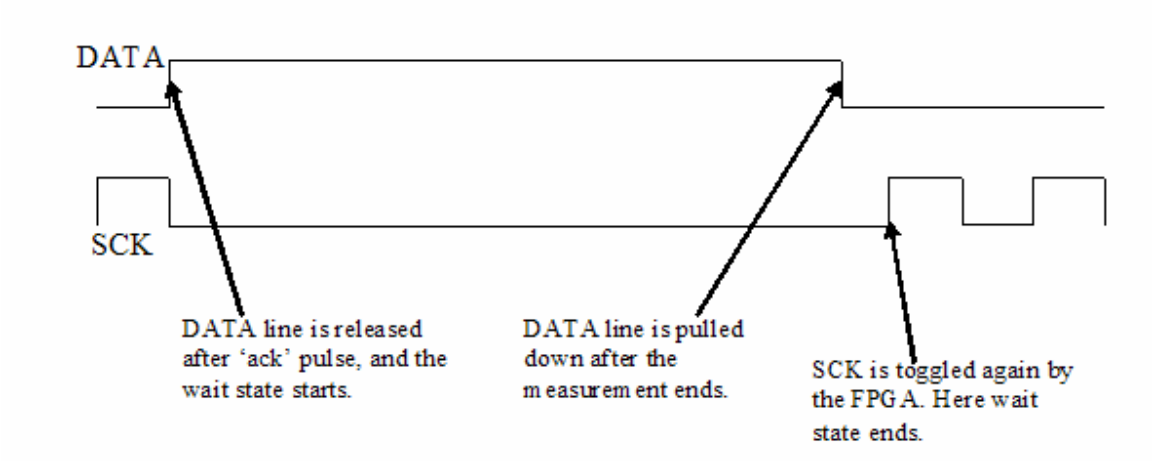


Figure 3-4: DATA and SCK lines in wait state

4. Measure the value from the sensor: In the measure state the FPGA simply has to send the serial clock SCK to the sensor for two bytes of data measurement. For every 9th SCK pulse, the DATA line is pulled low by the FPGA, indicating to the sensor that one byte of data has been received. All the values received on the serial DATA line are MSB (Most Significant Bit) first and right justified. The DATA and SCK lines in this state for a 14-bit temperature measurement are shown in Figure 3-5. If a CRC checksum is used, then the FPGA pulls down the DATA line after the second byte of measurement. Otherwise, the DATA line is released to skip the CRC state, and the sensor will wait for the next command.

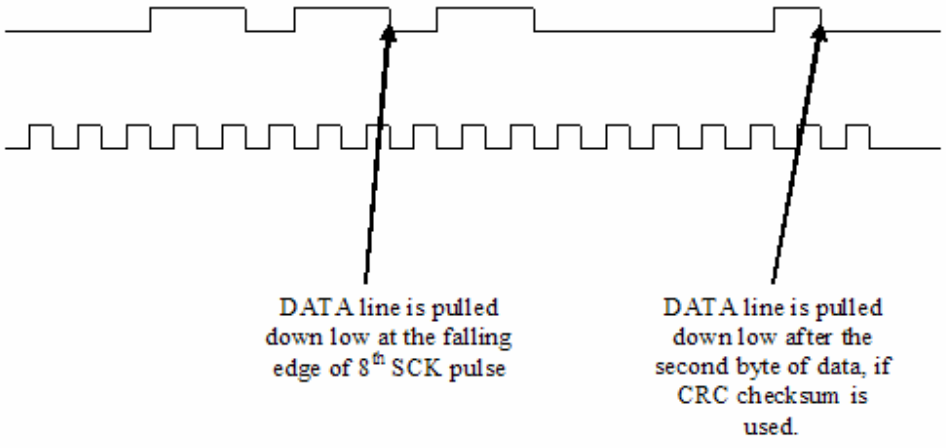


Figure 3-5: The measurement sequence of SHT71 sensor for a 14-bit value "01101111000001".

5. CRC measurement: If the DATA line is pulled down in the measure state after the second byte of data is received, then the sensor sends an 8-bit CRC value over the next 8 cycles of SCK. This 8-bit CRC value is compared with an 8-bit CRC value generated by the CRC generator of the EMC processor. If the value is mismatched then a 'soft reset' command is sent to the sensor by the processor. The 'soft reset' command for the sensor is "00011110". The command for the measurement will be issued again after the soft reset sequence. A detailed explanation of how the CRC is generated by the FPGA is explained in next section. Figure 3-6 shows the DATA and SCK sequences for CRC measurement and the soft reset sequence.

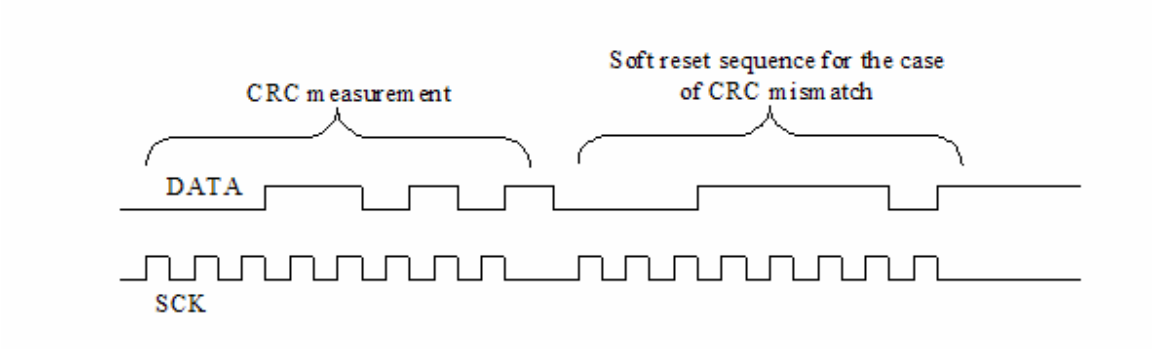


Figure 3-6: CRC measurement and soft reset sequences in case of CRC mismatch.

In addition to the above five different states of measurement, another sequence is needed in case of any communication error. It is called the connection-reset sequence. This command is useful in case the communication with the sensor is lost. This sequence will reset the serial interface, but it will preserve the contents present in the status register of the sensor. In this design the connection-reset sequence is used for every measurement before the command for measurement is issued. Thus, the serial interface is checked properly before every measurement to make sure that the correct data is received. Figure 3-7 shows the connection-reset sequence of the DATA and SCK lines. The connection

reset sequence is always to be followed by a transmission start sequence which was explained earlier in this section. The serial clock (SCK) is toggled 9 or more times while the DATA line is high.

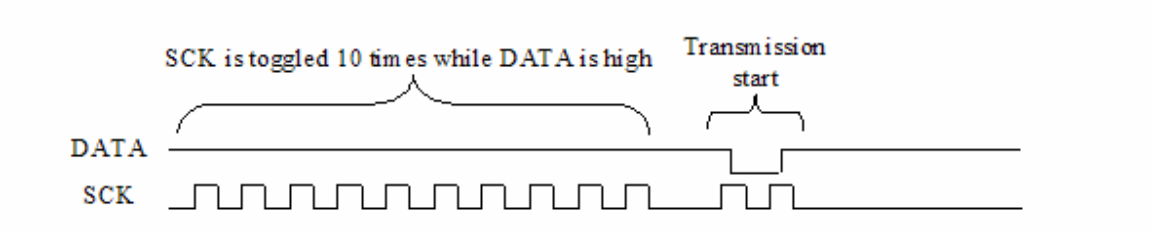


Figure 3-7: Connection reset sequence for SHT 71.

3.3 Design of CRC Generator:

The acronym CRC stands for Cyclic Redundancy Check. It is a very powerful, but easily implemented technique to obtain data reliability. This technique is used to protect the data sequence generated by the sensor from any errors during transmission. In the CRC technique an *n*- bit sequence of data is appended to the original data at the end of the transmission [8]. This appended sequence contains some information about the data transmitted, and this information has to match with the computed CRC. The CRC technique has its advantages of extreme error detection capabilities and very easy implementation. The CRC polynomial used in the SHT71 sensor is $x^8 + x^5 + x^4$. This CRC algorithm detects the following kinds of errors.

- Any odd number of errors anywhere in the transmission.
- All double-bit errors within the transmission.
- Any cluster of errors that can be contained within an 8-bit window (1-8 bits incorrect).
- Larger clusters of errors.

There are several methods to implement the CRC algorithms. The bitwise method is the most suitable method for hardware or low level implementation [9]. The hardware implementation for the CRC generator is shown in Figure 3-8. This is the same hardware used in the transmitter also.

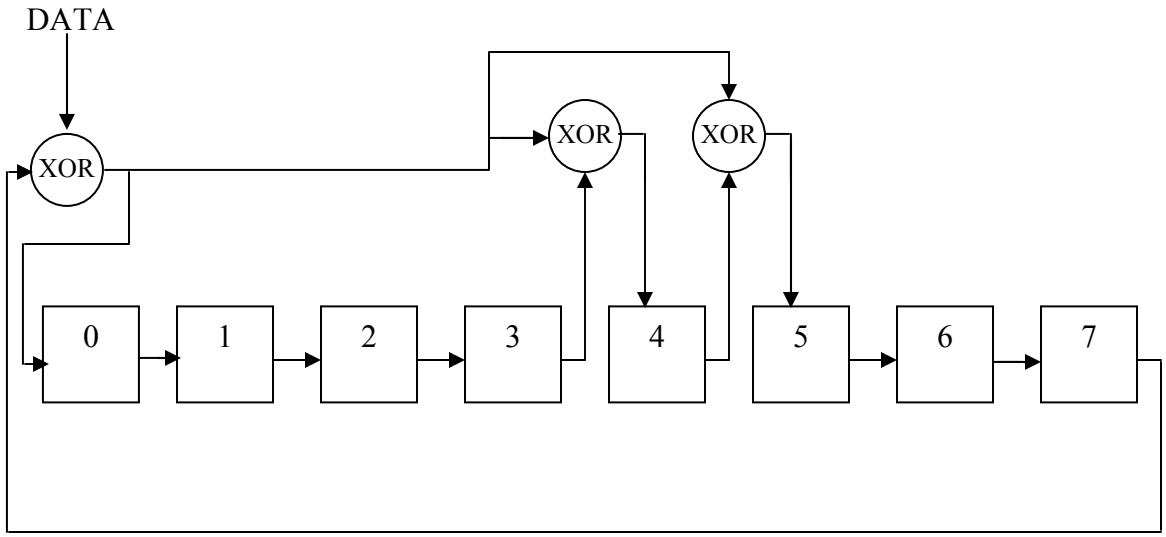


Figure 3-8: Structure of CRC generator.

The CRC register shown in Figure 3-8 is an 8-bit register. This register has the same structure as the CRC register present in the SHT71 sensor. The algorithm used to calculate the CRC value is explained in the following steps.

1. The CRC register is generally loaded with the low nibble of the status register ($s_0s_1s_2s_3$'0000) of the SHT71 sensor. It is generally initialized with a default value "00000000". In this design any of the status register commands such as read status register and write status register are not used. Hence, the CRC register is loaded with "00000000".
2. Every transmitted and received bit except the acknowledge bits from the beginning of the transmission are compared with bit 7.
3. If the transmitted or received bit is the same as bit 7, the CRC register is right shifted, and bit 0 is loaded with '0'. If the transmitted or received bit is not the same as bit 7, the CRC register is right shifted, bit 4 and bit 5 are inverted, and bit 0 is loaded with '1'.
4. The next bit is received and the steps from 2 are repeated.

The CRC value obtained through out the transmission from the CRC generator is then compared with the CRC value received in the measure sequence. If the values are the

same, then the values received from the sensor are correct. Otherwise, the measurement is repeated.

3.4 Design of Interface Module:

A high level functional diagram of the interface module is shown in Figure 3-9. This module acts as an interface between the sensor and the rest of the hardware needed to calculate Equilibrium Moisture Content. The inputs and outputs of the interface module are as shown in Figure 3-9.

- Serial bi-directional DATA line.
- SCK (clock to the sensor).
- A start signal to the arithmetic unit used to calculate EMC, and a 'done' signal from the arithmetic unit to indicate the completion of calculation.
- SO_T and SO_{RH}, two 14 bit vectors of the temperature and humidity values measured from the sensor.

The whole interface module works on the 'clock' input given as shown in Figure 3-9. A 'start' input is used to start the module. The 'clear' input asynchronously clears the contents of all the registers. The main components in the interface module are 'DATA and SCK Generator' and 'Main Controller'. The functionality of the interface module is explained through the description of the individual components in this section.

- Tri-state buffer: As given in the specifications of the sensor, the interface module never writes '1' on the bidirectional data line. An external pull-up resistor is used to pull the value on data line high. Hence the interface module uses a tri-state buffer to produce a '0' when the enable signal is low and high impedance 'Z' when the enable signal is high.
- 'Check' flip-flop: A flip flop which uses the signal 'SCK' as its clock is used to check the value on the data line when required. For example, according to the serial protocol, the sensor has to pull down the data line after the command for measurement is sent. The check flip flop is checked at that moment by the 'DATA and SCK Generator' for the error.

- ‘Error’ flip-flop: In case of any error on the data line this flip-flop is loaded with a ‘1’ by the ‘DATA and SCK Generator’. The output of this flip-flop goes to the Main Controller, which decides whether to repeat the whole measurement again, or not, depending on this error value.
- DATA and SCK Generator: The ‘DATA and SCK Generator’ generates the ‘enable’ signal for the tri-state buffer shown in Figure 3-9, and SCK for the sensor to synchronize the communication between FPGA and sensor. In addition to generating ‘enable’ and ‘SCK’ signals, the ‘DATA and SCK Generator’ generates the ‘load’ signals for the ‘check flip flop’, ‘error flip flop’ and the CRC Generator.
- CRC Generator: The detailed structure of the CRC generator was explained in the previous section. The CRC value generated through out the transmission is stored in the register ‘Crc_gen_reg’. This value is compared with the CRC value received from the sensor which is stored in ‘Crc_sr_reg’. The error signal is given as input to the Main Controller. The Main Controller generates the ‘soft reset’ sequence in case of any CRC error.

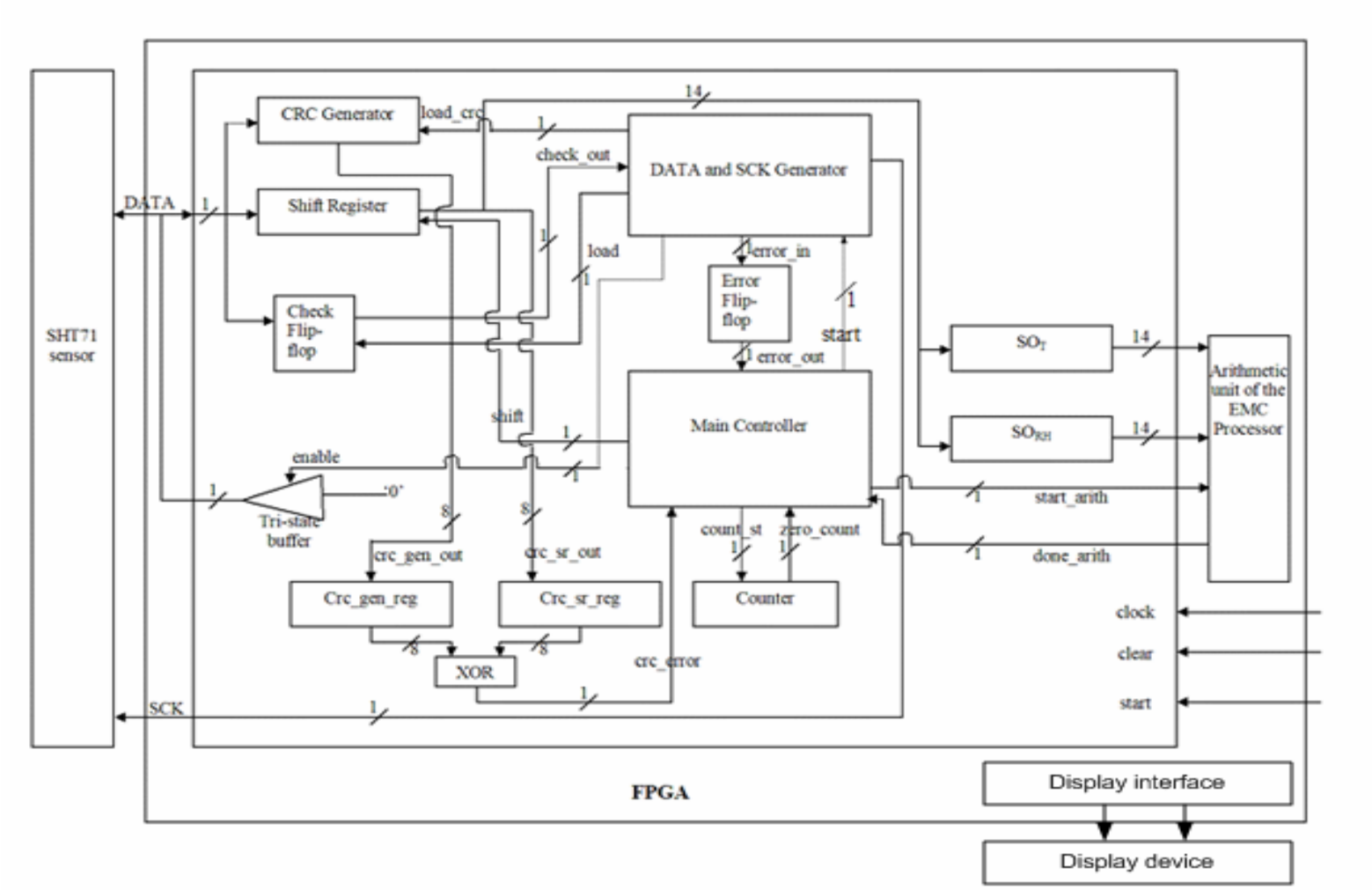


Figure 3-9: High level schematic of interface module.

- Shift Register: A serial-in parallel-out register is used to store the values of data received from the serial data line and send it to registers SO_T, and SO_{RH}. The data is serially left shifted on the positive edge of SCK. The parallel output is taken on the positive edge of system 'clock'. The width of the register is 26 bits. During a measurement, all the bits on the data line, including the acknowledge bits, are stored in this register. Depending on the control signal 'shift' from the Main Controller, the input data is taken and shifted. The shift register sends 14-bit outputs to the registers SO_T, and SO_{RH}. These values are the measured temperature and humidity values of the sensor. It also sends an 8- bit output 'crc_sr_out' to the register 'Crc_sr_reg'.
- Counter: The controller needs to be in an idle mode when the data is being measured by the sensor. For a 14-bit measurement it needs to wait approximately 210ms. Hence a counter is used to indicate to the main controller when to stop the idle mode. In the wait state the controller sends the 'count_st' signal to the counter, which upon counting the required count sends back the signal 'zero_count' to the main controller.
- Main Controller: A high level schematic of the Main Controller with the input and the output signals is shown in Figure 3-10. All the inputs and outputs are 1-bit wide. Figure 3-11 shows the clock cycle by clock cycle system flow chart of the Main Controller used in the design. The operation of the Main Controller can be explained as follows.

First, 'Start_data_sck' signal is generated by the Main Controller to initiate the 'DATA and SCK Generator'. When this signal is asserted high, the 'DATA and SCK Generator' starts its operations and wait for the command from the Main Controller. In the second step, 'Trans_reset' is given by the Main Controller to the 'DATA and SCK Generator'. When this signal is received by the 'DATA and SCK Generator', it starts generating the 'transmission start' sequence for the DATA and SCK lines. Signal 'done_trans' is a hand-shaking signal is given to the Main Controller by the DATA and SCK Generator to indicate the completion of transmission start sequence. The 'DATA and SCK Generator' waits in this state until it receives any response from the Main

Controller. After receiving the hand shake signal ‘done_trans’ the Main Controller generates ‘trans_ok’ to the ‘DATA and SCK Generator’ to end its wait state. Thus after the transmission start sequence is generated, the Main controller asserts the signal ‘Temp_m’ to the ‘DATA and SCK Generator’. In case of temperature measurement the hand-shaking signals used are ‘done_temp’ and ‘temp_ok’. After issuing the command for temperature measurement, the Main Controller enters the wait state issuing the command ‘wait_measure’ to the ‘DATA and SCK Generator’. The hand-shaking signals in wait state are ‘done_wait’ and ‘wait_ok’.

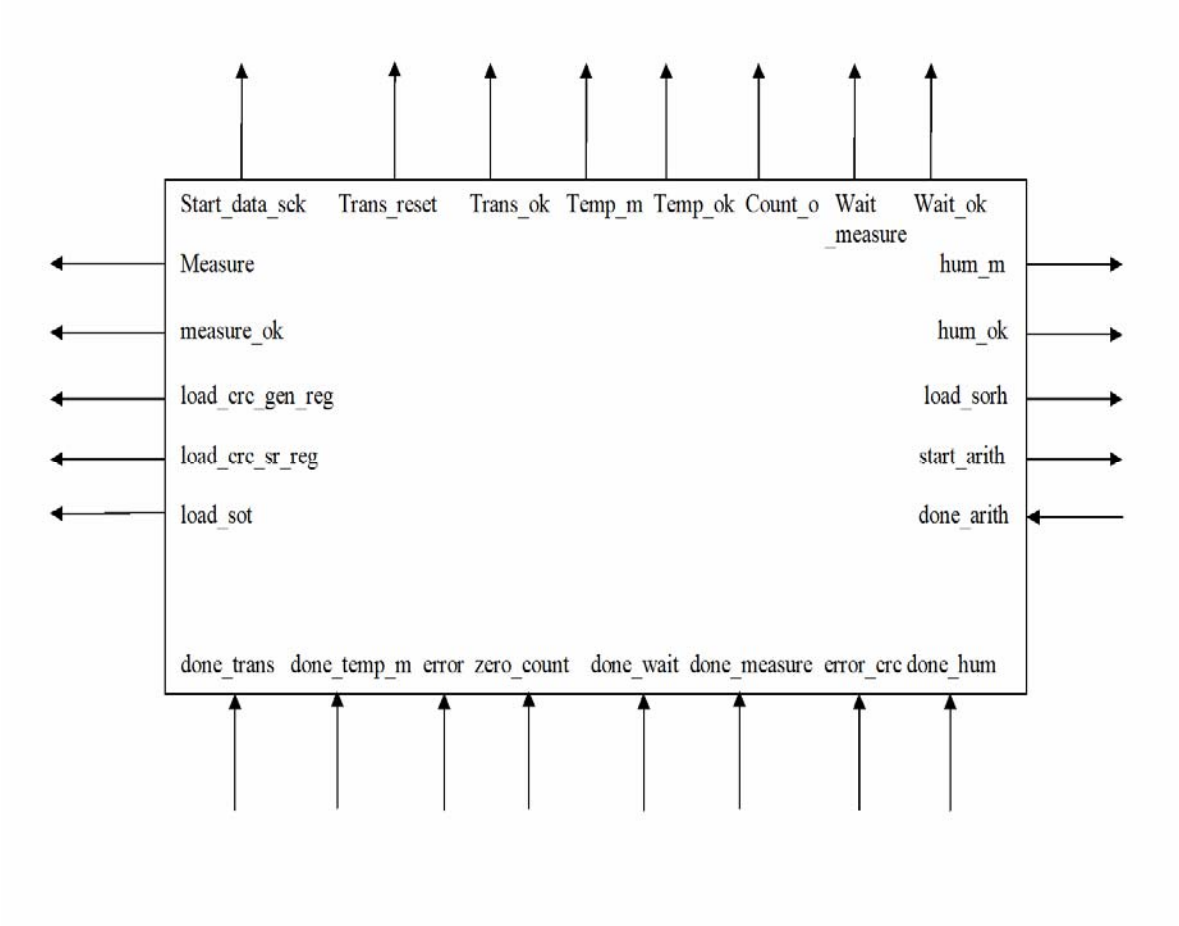


Figure 3-10: High level schematic of the Main Controller of the interface module.

At the end of the wait state the Main Controller checks for any ‘error’ in the transmission. In case of any error, the whole sequence is repeated. Otherwise, it proceeds to measure the value of the temperature by generating the signal ‘Measure’ to the ‘DATA and SCK Generator’. The hand-shaking signals used after the measurement is complete are ‘done_measure’ and ‘measure_ok’. Once the measurement is completed, the Main Controller checks if there is any CRC error by inspecting the signal ‘error_crc’. If the error is zero then the Main Controller loads the digital read out of temperature to the register SOT and proceeds to the measurement of relative humidity. In case of any CRC mismatch the ‘soft reset’ sequence is generated by the ‘DATA and SCK Generator’ and the whole transmission is repeated.

The control flow of the Main Controller for the measurement of the relative humidity is similar to that of the temperature. The Main Controller sends the signal ‘hum_m’ to the ‘DATA and SCK Generator’ for the humidity measurement. The Main Controller and the ‘DATA and SCK Generator’ uses the hand-shaking signals ‘done_hum’ and ‘hum_ok’ signals for the humidity measurement. The result of the measurement is stored in SORH register. The clock cycle by clock cycle system flowchart of the Main controller is shown in Figure 3-11. The flow chart shows all the operations of the Main Controller for every clock cycle. As shown in Figure 3-11, the Main Controller measures the temperature from the sensor at the first step. It proceeds to measuring the relative humidity if there is no error in the transmission. Once the relative humidity measurement is complete without any errors in the transmission, then the Main Controller sends a ‘start_arith’ signal to the arithmetic unit of the EMC Processor. The Main Controller waits for the response from the arithmetic unit. If it receives a ‘done_arith’ signal from the arithmetic unit, then the control proceeds to the state S2.

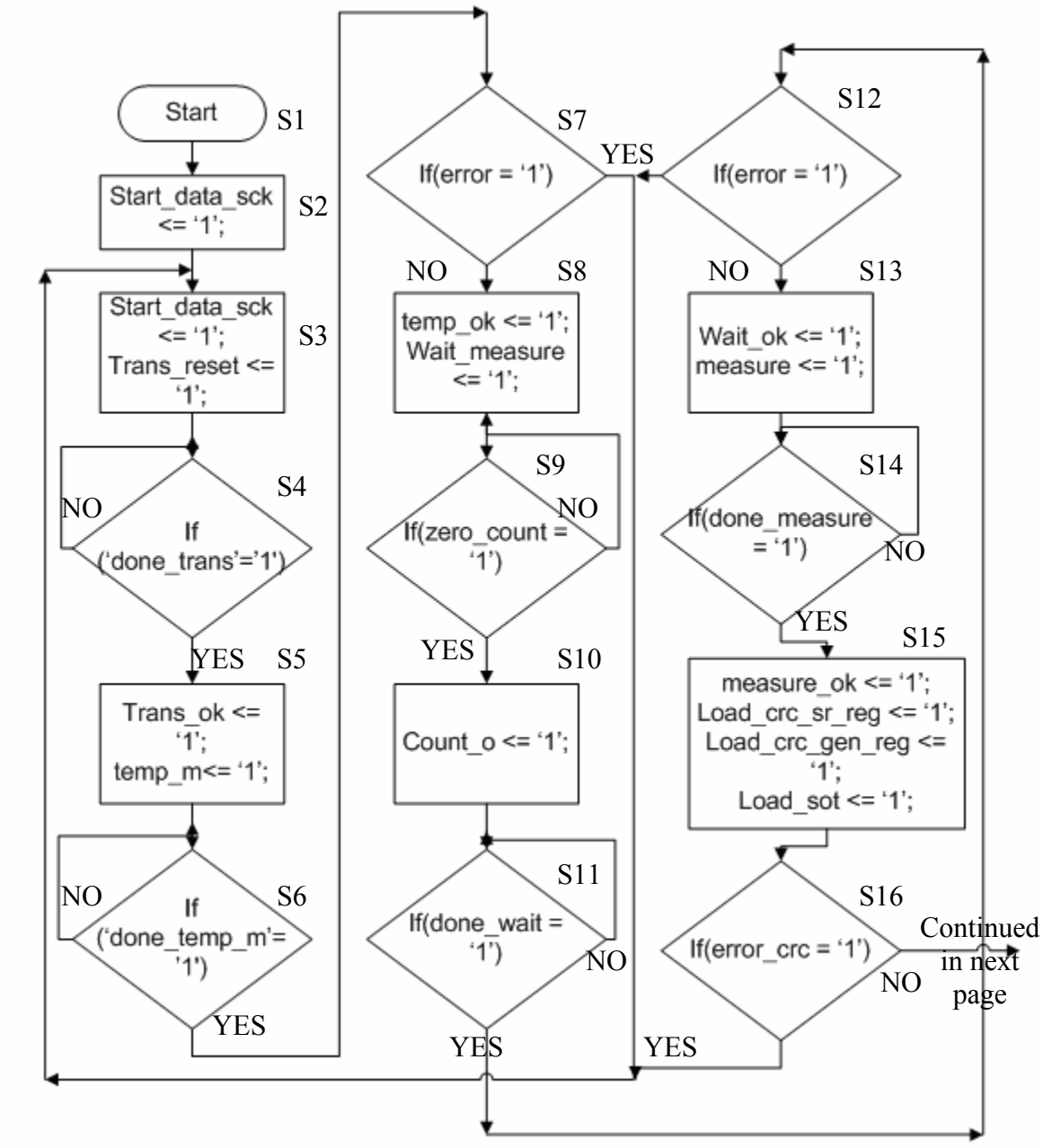


Figure 3-11: Clock cycle by clock cycle system flowchart of the interface module.

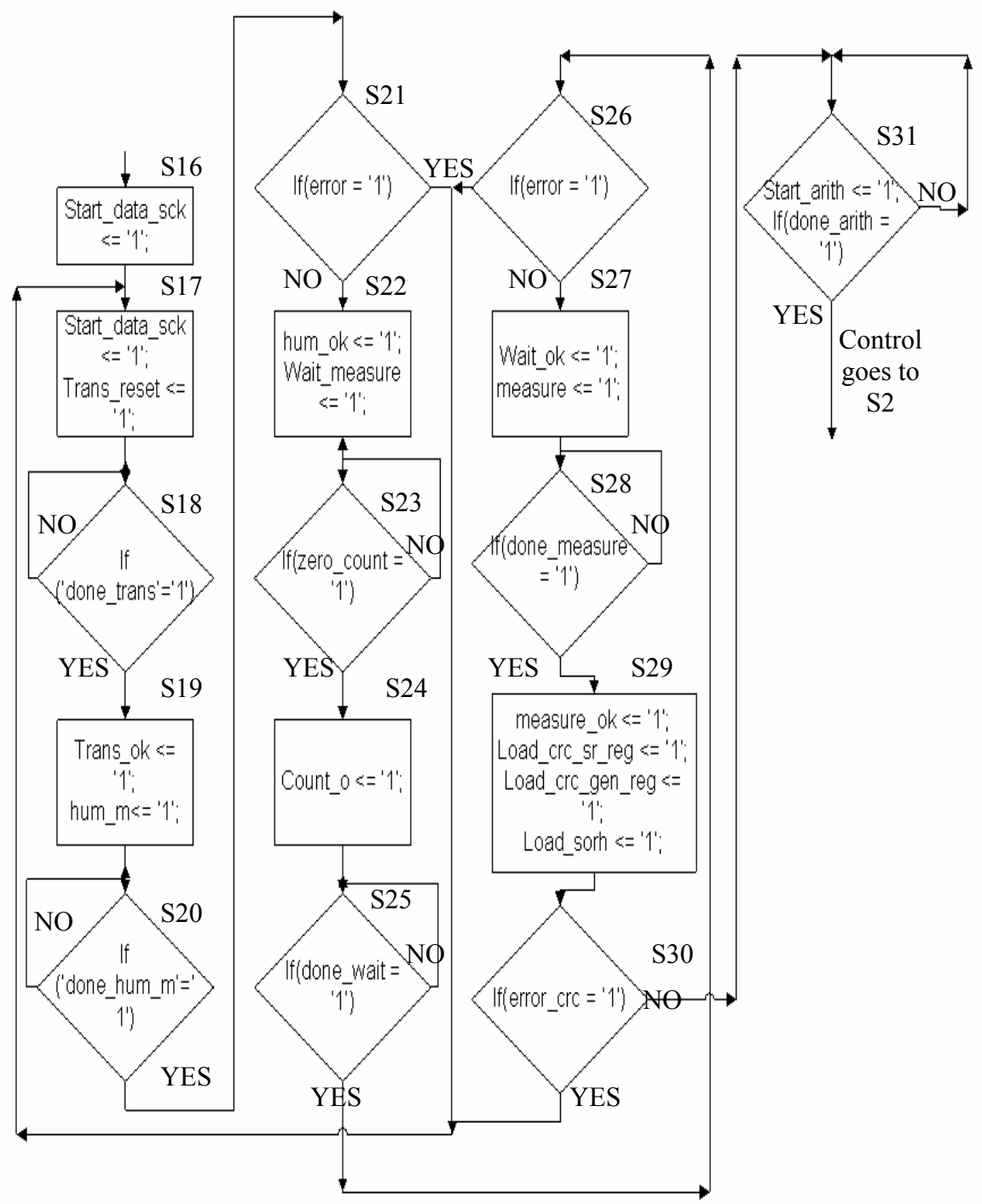


Figure 3-11: Clock cycle by clock cycle system flowchart of the interface module.

3.5 Sensor Electrical Characteristics and Design Specifications:

From the electrical characteristics of the sensor, the output peak current is 4mA [7]. This will limit the value of the pull-up resistor to be used in the design (see Figure 3-1). For an input voltage of 3.3V, the minimum value of the resistor that can be used is 3.3V/4mA = 825Ω. For an input voltage of 5V the value would be 5V/4mA = 1.25kΩ.

From the I/O signal characteristics, the maximum frequency of the SCK signal generated is 1MHz for an input voltage <4.5V, and 10MHz for an input voltage >4.5V. In the design these values are achieved by using a clock divider circuit to the main system clock. There is no minimum value of frequency for SCK. Hence any larger time periods for SCK will work for the design.

3.6 VHDL Post Implementation Results of Interface Module:

The VHDL code for the interface module shown in Figure 3-9 is synthesized and implemented to a Xilinx Spartan2E FPGA. A Tektronix logic analyzer was used to observe the waveforms for the DATA and SCK signals sent to the SHT71 sensor. These post implemented wave forms are shown in this section. The input voltage supply given to the sensor is 3.3V and the value of pull up resistor used is 1.5kΩ.

Figure 3-12 shows the waveforms of the DATA (on input line and output line), SCK, and the enable input given to the tri-state buffer. From Figure 3-12 it can be observed that a connection reset is used before transmission start to ensure a proper serial interface. The transmission start sequence is followed by a command to measure temperature. It can be seen that at the 9^{th} pulse of SCK, the DATA is pulled down to '0' and released again by the sensor. After checking for error at the 9^{th} clock pulse the controller enters into the wait mode.

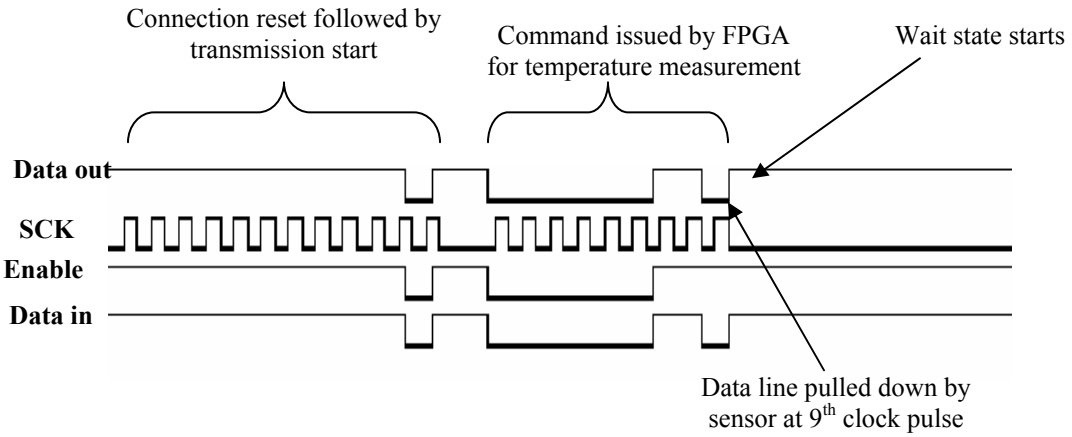


Figure 3-12: VHDL post implementation simulation results of interface module showing transmission start and command for temperature.

Figure 3-13 shows the waveforms of DATA, SCK and enable for the wait state. The DATA line is always high until the end of the wait state. The sensor pulls down the DATA (bi-directional line) to '0' after it finishes measurement. The main controller then checks the value of the DATA line and issues a command to the 'data and sck generator' to generate SCK for measure state.

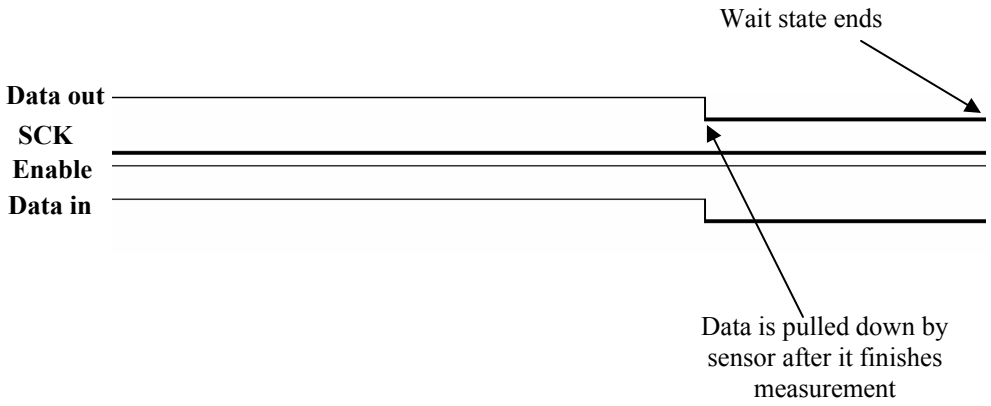


Figure 3-13: VHDL post implementation simulation results of interface module showing the wait state for temperature measurement.

Figure 3-14 shows the waveforms in the measure state of temperature. As shown, the SCK is toggled for 3 bytes of DATA including the acknowledge pulses. The first two bytes of DATA are the digital output for temperature from the sensor. The third byte of DATA is the CRC checksum sent by the sensor. This value is matched with the CRC generated by the FPGA and hence a 'soft reset' sequence is not generated. The 'enable' input to the tri-state buffer is made zero after every byte of measurement, thereby pulling the data line to zero.

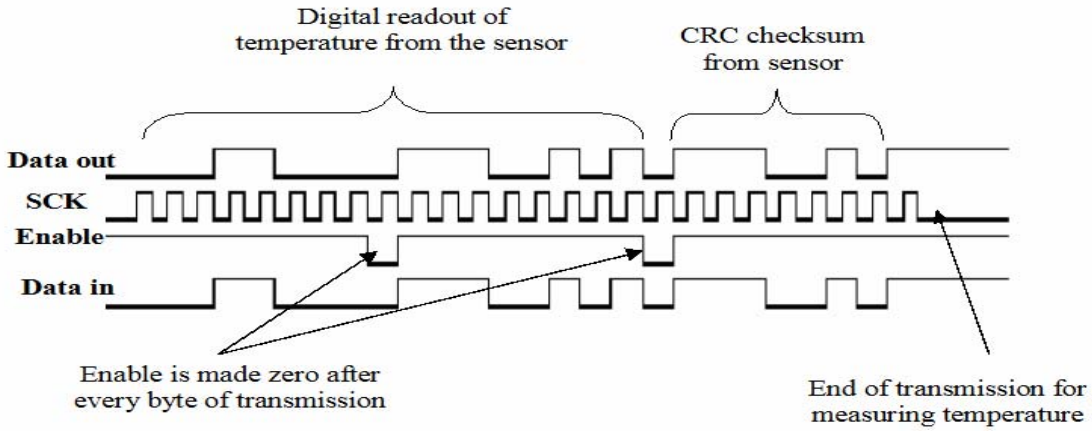


Figure 3-14: VHDL post implementation simulation results of interface module showing the digital readout of temperature and CRC checksum measurement.

Figure 3-15 shows the DATA, SCK and enable waveforms for the transmission start, Command for measurement and the wait state for humidity measurement. After the transmission start, the controller issues the command "00000101" for humidity. The sensor pulls down the data line during the acknowledgement pulse of SCK. After 90 ms the sensor pulls down the data line indicating to the controller that measurement is finished.

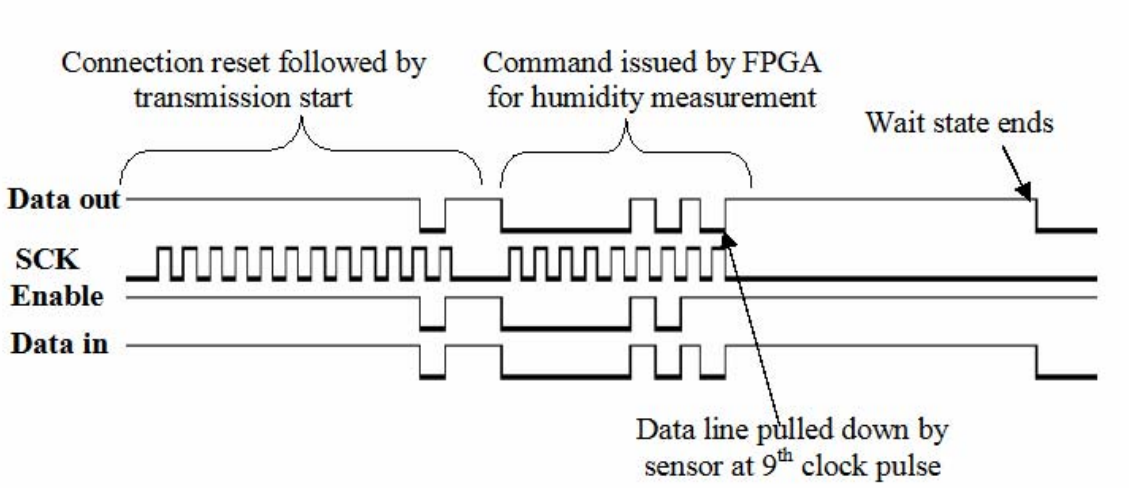


Figure 3-15: VHDL post implementation simulation results of interface module showing the transmission start, command for humidity measurement and wait state.

The digital readout of humidity and the CRC measured value transmitted over data line is shown in Figure 3-16.

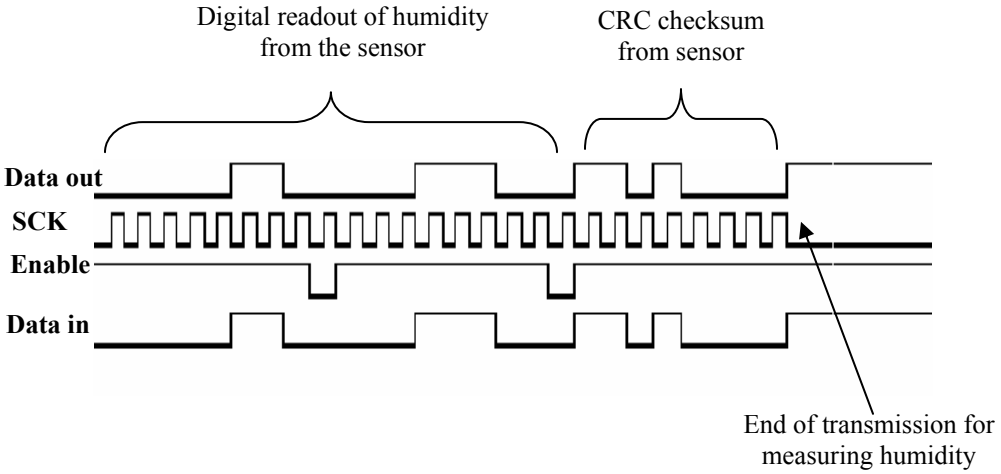


Figure 3-16: VHDL post implementation simulation results of interface module showing the digital readout of humidity and CRC checksum measurement.

Thus, from the post-implementation simulation results of the interface module it can be concluded that the interface module designed and developed in VHDL to a Spartan2E

FPGA works for an example test case of temperature and relative humidity measurement. The interface module along with the arithmetic unit of the EMC processor is tested and validated in various ranges of temperature and relative humidity. These implementation test results will be discussed in Chapter 5.

3.7 Physical Outputs and Non-Linearity Compensation of the Sensor:

The SHT71 sensor shows a small non-linearity for humidity measurements. The sensor uses a band gap PTAT (Proportional to Absolute Temperature) temperature sensor for temperature measurements. This sensor is very linear by design. Hence, in this design, certain compensation for non-linearity of the humidity values is used. The sensor provides 14- bit and 12-bit digital values for temperature and relative humidity respectively. These values have to be converted into physical values [7].

If SO_T is the digital read out of the sensor for the temperature then the physical value of the temperature is calculated using Equation 3-1[7].

$$T_{real} = d_1 + d_2 * SO_T \qquad (3\text{-}1)$$

Where d_1 = -40.00 and d_2 = 0.01 for an input voltage of 5V and temperature in Celsius.

For more accurate values of temperature equation (3-2) can be used [7]. This corrects the non-linearity of about -1^{o}C at -40^{o}C or 100^{o}C compared to the linear formula of Equation 3-1.

$$T_{real} = d_1 + d_2 * SO_T + d_3 * (SO_T - f)^2 \qquad (3\text{-}2)$$

Where d_1 = -40.00, d_2 = 0.01, $d_3 = -2*10^{-8}$ and f = 7000.

But considering the operating range of the device, Equation (3-2) is not quite necessary as it corrects a non-linearity of about -1^{o}C at -40^{o}C or 100^{o}C compared to the linear formula. Hence in this design the linear formula for temperature is used.

If SO_{RH} is the digital read out of the sensor for the humidity then the physical value of the humidity is given by the equation (3-3) [7].

$$RH_{linear} = c_1 + c_2 * SO_{RH} \qquad (3\text{-}3)$$

Equation (3-3) is the most basic formula for conversion of SO_{RH} to humidity, where c_1=0.5 and c_2=0.5.

But, from the non-linear properties of the sensor, it has around +/- 3% humidity for the values 10% RH to 90%RH. To compensate the non-linearity of the humidity, equation (3-4) is used [7].

$$RH_{linear}=c_1+c_2*SO_{RH}+c_3*SO_{RH}^2 \quad (3\text{-}4)$$

Where c_1=-4, c2=0.0405, and c_3=-2.8*10^{-6} for a 12-bit humidity measurement. The accuracy of the compensated values of humidity from 10%RH to 90%RH using equation (3-4) is +/-0.1%RH. This very much improves the linearity in the required operating ranges of humidity. Hence, equation (3-4) is used in the design to compensate the non-linearity of the humidity.

For temperatures significantly different from 25^oC, the temperature coefficients should also be considered for temperature compensation. Equation (3-5) is used to find the true value of relative humidity for a 12-bit measurement, which is temperature compensated.

$$RH_{true} = (T_{real} - 25)*(0.01 + 0.00008*SO_{RH}) + RH_{linear} \quad (3\text{-}5)$$

Thus equations (3-1), (3-4), and (3-5) are used in the design to convert the digital outputs from sensor to physical values, to correct the non-linearity and for temperature compensation. The interface module sends the 14-bit outputs SO_T and SO_{RH} to the arithmetic unit. The arithmetic unit performs the required calculations in equations (3-1), (3-4) and (3-5) along with the equations used to calculate the EMC. The design of the arithmetic processor unit is explained in detail in the following Chapter 4.

Chapter 4

Design of Arithmetic Unit

This chapter addresses the design of the arithmetic unit of the EMC processor. The bit width of the arithmetic unit required for the design is determined. Two versions of the arithmetic unit are developed for the EMC processor. They are (i) Parallel Arithmetic Unit and (ii) Serial Arithmetic Unit. The parallel arithmetic unit uses an adder, a multiplier and a divider to calculate the value of EMC. The serial arithmetic unit uses an adder and a special purpose register to compute the value of EMC. Detailed reports of the hardware required to implement individual units are presented in this chapter. A high level functional architectural diagram of the entire arithmetic unit used to calculate EMC is given. VHDL post place and route simulations and validations of the two different versions of the arithmetic unit are addressed and achieved. The two versions of the arithmetic unit are compared in terms of the utilization of logic resources and speed.

4.1 Equations and Their Design Requirements:

The equations used to calculate the Equilibrium Moisture Content (EMC) are repeated below for convenience.

$$EMC = \frac{1800}{W}\left[\frac{Kh}{1 - Kh} + \frac{K_1 Kh + 2K_1 K_2 K^2 h^2}{1 + K_1 Kh + K_1 K_2 K^2 h^2}\right] \quad (4\text{-}1)$$

For temperature T in Celsius,

$$W = 349 + 1.29T + 0.0135T^2$$
$$K = 0.805 + 0.000736T - 0.00000273T^2$$
$$K_1 = 6.27 - 0.00938T - 0.000303T^2$$
$$K_2 = 1.91 + 0.0407T - 0.000293T^2$$

And for temperature T in Fahrenheit,

$$W = 330 + 0.453T + 0.00415\,T^2$$
$$K = 0.791 + 0.000463\,T - 0.000000844\,T^2$$
$$K_1 = 6.34 + 0.000775\,T - 0.0000935\,T^2$$
$$K_2 = 1.09 + 0.0284\,T - 0.0000904\,T^2$$

In the above equations T is the temperature in the surrounding atmosphere, h is the relative humidity (%/100), and EMC is the Equilibrium Moisture Content (%).

The equations show two different sets of equations for constants W, K, K_1, and K_2. One set of equations are for T in Fahrenheit and the other for T in Celsius. The design requires displaying the value of temperature T in both Fahrenheit and Celsius depending on user control input. But no matter which equations are used, the value of EMC will be the same. In developing the arithmetic unit design, the equations for Fahrenheit are used to calculate the value of EMC. The output of the display module will need to display temperature in degrees for Celsius. To display the temperature in both Celsius and Fahrenheit, Equation (4-2) is used to convert the value from Celsius to Fahrenheit.

$$\frac{C}{5} = \frac{F - 32}{9} \tag{4-2}$$

where C is temperature in Celsius and F is temperature in Fahrenheit.

Equations (3-1), (3-4) and (3-5) used to convert the outputs of the sensor to the required physical value, and correct the non-linearity of humidity are also repeated below.

$$T_{real} = -40 + 0.01(SO_T)$$
$$RH_{linear} = -4 + 0.0405(SO_{RH}) - 0.0000028(SO_{RH})^2$$
$$RH_{true} = (T_{real} - 25)*(0.01 + 0.00008*SO_{RH}) + RH_{linear} \tag{4-3}$$

where SO_T is the 14-bit temperature output from the sensor SHT71, SO_{RH} is the 12-bit humidity output from the sensor SHT71, T_{real} is the corrected temperature value, RH_{linear} is the linear SO_{RH} value and RH_{true} is the temperature compensated value of RH_{linear}.

Equations (4-1), (4-2) and (4-3) require the design of an arithmetic unit with multiply, divide, add, and subtract capability. One of the primary design objectives is that the EMC value is to be calculated every second. A basic system of clock of time period 1.3 ms is used for the design of EMC processor. From the VHDL post implementation simulation results of the Sensor Interface Module, the time taken for obtaining the required temperature and relative humidity outputs from the sensor SHT71 is approximately 700 ms. Therefore, to achieve the condition of obtaining EMC value for every one or two seconds, a multiplier, a divider and an adder can be used in the design of the EMC processor.

Implementation of Equations (4-1), (4-2), and (4-3) require many multiplications, divisions, additions and subtractions. From the equations it can be observed that there are integers such as 1800 and there are fractions such as 0.000000844. As there are many included fractions and the result is also to be shown as a real number, simple integer arithmetic units can't be used for the design. Real number arithmetic hardware can be designed using either fixed point arithmetic or floating point arithmetic. The following sections explain in detail fixed point arithmetic and floating point arithmetic. Fixed point and floating point arithmetic units are compared and a decision is made on which system to use for the design of the arithmetic unit of the EMC processor.

4.2 Fixed Point Arithmetic:

A fixed point arithmetic system has a binary point whose position is always fixed within the arithmetic operands. The position of the binary point can be anywhere in the binary number operands. A conventional fixed point arithmetic system is usually based on a positive integer radix r and an implicit digit set $(0, 1 \ldots r-1)$. Each unsigned real number operand is represented by a digit vector of length $k + l$, with k digits for the integer part and l digits for the fraction part. The value l is also known as the scaling factor of the fixed point vector. The digit vector
($x_{k-1}x_{k-2}\ldots\ldots x_1x_0.x_{-1}x_{-2}\ldots\ldots x_{-l}$) represents the following value [10].

$$x_{k-1}x_{k-2}.....x_1x_0.x_{-1}x_{-2}.....x_{-l} = \sum_{i=-l}^{k} x_i r^i \qquad (4\text{-}4)$$

Equation (4-4) represents the value of an unsigned binary real number vector represented in fixed point format. In signed representation of binary numbers, the most significant digit is the sign bit and the value of the rest of the binary vector can be calculated the same as above. Examples of fixed point real number vectors using unsigned, sign-magnitude and two's complement representation are shown below.

The value of an 8- bit binary vector 1010.0101 with the binary point located after the first four digits can be calculated in a fixed point arithmetic system as follows.

- Unsigned : $1010.0101_2 = 1 *2^3+0*2^2+1*2^1+0*2^0 + 0*2^{-1}+1*2^{-2}+0*2^{-3}+1*2^{-4} = 10.3125_{10}$
- Sign-magnitude: $1010.0101_2 = -(0*2^2+1*2^1+0*2^0 + 0*2^{-1}+1*2^{-2}+0*2^{-3}+1*2^{-4})= = -2.3125_{10}$
- Twos complement representation : $1010.0101_2 =$ $- (1*2^2+0*2^1+1*2^0 + 1*2^{-1}+0*2^{-2}+1*2^{-3}+1*2^{-4}) = - 5.6875_{10}$

4.2.1 Precision and Range of Fixed Point Arithmetic:

The range of a fixed point binary vector that is represented in two's complement representation, with *k* integer bits and *l* fraction bits can be determined as follows. The maximum value that can be represented is given by

$2^{k-2}+2^{k-3}+.....+2^0 + 2^{-1}+2^{-2}+.....2^{-l}$

The minimum value represented is

$- (2^{k-1})$

Here the bit at position *k* -1 represents the sign of the binary vector represented in twos complementary representation.

Suppose m denotes the value of the complete width of the binary vector $k + l$, then the precision or resolution of the binary vector is given by $2^{-m/2}$. Suppose the value of m is 16 then the precision or resolution of the binary vector is $2^{-8} = 0.00390625$.

4.2.2 Arithmetic Operations in Fixed Point Arithmetic:

The hardware design of a fixed point arithmetic unit requires special attention to the binary point throughout the design. Addition and subtraction operations in fixed point arithmetic are similar to normal integer arithmetic systems. The position of the binary point of the result of either addition or subtraction is the same as that of the inputs. One should assure that the binary point position of all inputs is the same [11]. The following examples of binary vectors represented in a binary 2's complement representation show that the scaling factor of the result is the same as that of the inputs when the binary points of both inputs are at the same position and aligned.

Ex:

	1010.0101_2		1010.0101_2
+	0001.0010_2	-	0001.0010_2
	1011.0111_2		1001.0011_2

The multiplication operation in a fixed point arithmetic unit is slightly different from the above. The binary point of the multiplied result is determined by the number of fraction bits of the multiplicand and the multiplier. Suppose the scaling factor of the multiplicand is m and the scaling factor of the multiplier is n, then the scaling factor of the result is $m + n$. An example of multiplication of two 4-bit binary vectors is shown below. It can be seen that the scaling factor of the multiplicand and the multiplier is 2, but the scaling factor of the result is 2 + 2 = 4.

Ex: 10.01_2

X 01.00_2

———————

0000

0000

1001

0000

———————

010.0100_2

The scaling factor in the division operation for the fixed point arithmetic is also different from its inputs. Suppose the scaling factor of the dividend is *m* and the scaling factor of the divisor is *n*, then the value of the result will have a scaling factor of $m - n$. An example of fixed point division for two 16-bit binary vectors with scaling factors of 5 and 3 respectively is shown below. It can be seen that the result has the scaling factor of 5-3 = 2.

Ex:

$$\frac{000001011111.11010_2}{0000000001110.011_2} = 00000000000011.01_2$$

Thus the design of hardware in a fixed point arithmetic system requires special attention to the scaling factor through out the design. Though the result of addition and subtraction leaves the scaling factor of the result the same as that of the inputs, the multiplication and division operations change the scaling factor. Hence the design and HDL coding of hardware to implement fixed point arithmetic is tedious, for the designer has to take care of the binary point during every basic operation.

4.3 Floating Point Arithmetic:

Floating point arithmetic is the most common real number arithmetic used currently. In floating point arithmetic, the binary point is not fixed. The range of floating point numbers is quite large compared to that of fixed point numbers and arithmetic systems. A

floating point number has four components. They are the sign bit (sb), the significand (*s*), the exponent base (*b*), and the exponent (*e*). The exponent base *b* is usually implied and not represented in the number. Together these four components can be represented as
$M = (-1)^{sb} * s * b^{e}$
Figure 4-1 shows the floating point representation format generally used [10].

Figure 4-1: Typical representation of a floating point number.

In Figure 4-1 the sign bit field (sb) indicates whether the number is positive or negative. If 'sb' is zero the value is positive, else it is negative. The exponent field indicates a large or small number. The exponent sign is the complement of its most significant bit. The S field has the significand of the number. The significand is a normalized format of any real number.

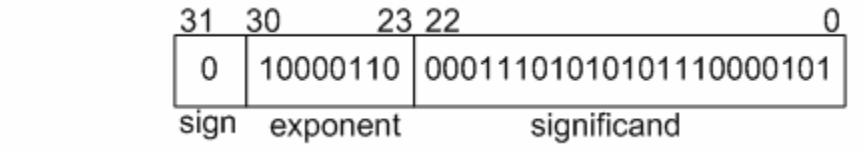
There are two IEEE standard representation formats for binary floating point numbers. The short, or single precision format, is 32 bits wide, whereas the long, or the double precision format, is 64 bits wide [12]. In the single precision format the exponent field *e* is 8 bits wide. The range of the exponent field is determined by the bias used for the format. For an 8-bit wide exponent field the range can be given by [-bias, 2^8-1-bias]. The bias used for single precision format is 127. Hence the range of the exponent field is [-127, 128]. The significand field is 23 bits wide with the range of [1, 2). The value that the significand field shows is the fraction part, and integer 1 is always to be added to the result. In the double precision format 11 bits are used for the exponent field and 52 bits for the significand field. The bias used for the exponent representation is 1023. Hence the range of the exponent field is [-1023, 1024]. The range of the significand field is the same as that of single precision, and the value is also computed in the same manner. The additional bits added to the exponent field increase the range of the number that can be represented whereas the additional bits added to the significand increase the precision of

the number represented. Though the two formats are industry standards, in many applications one can use a dedicated floating point format depending on the application.

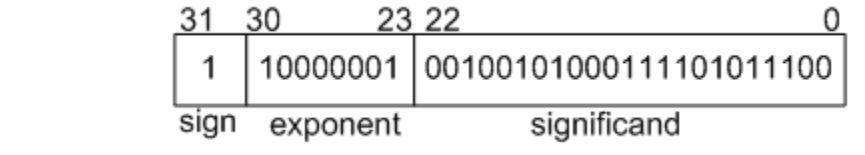
An example of binary floating point representations in IEEE standard single precision format is shown below. The bit-width is 32. The exponent field is 8-bit wide and the significand field is 23 bit wide and one most significand bit is used for sign.

142.67_{10} =$010001110.10101011100001010001111_2$ = $1.00011101010101110000101_2*2^7$
Thus, for the above example, the sign bit is zero as the number is positive. The exponent is 7. It should be added with the required bias value. The bias value for single-precision floating point format is 127. Hence the exponent field is 7 + 127 = 134 =10000110_2. The significand or mantissa is $00011101010101110000101_2$. The binary number '1' before the binary point is implied and not represented in IEEE representations.

Thus, the floating point representation for 142.67_{10} is:



Similarly, the floating point representation of -251.42_{10} is:



Floating point arithmetic unit design has two special issues to be taken care of. They are overflow and underflow. When the number to be represented is too large then the exception is known as overflow. When the number to be represented is too small to be represented, the exception is known as underflow. A larger number of bits in the exponent field will avoid the problems of overflow and underflow. The following sections will briefly describe arithmetic operations using floating point arithmetic systems.

4.3.1 Arithmetic Operations in Floating Point Arithmetic:

Addition and subtraction operations are more complicated in floating point arithmetic systems compared to multiplication and division [10].

Addition/Subtraction:

The addition of two floating point numbers is explained through an example below.

1. The two exponents of the two numbers are to be compared. The significand of the number with the lesser exponent is shifted to the right until both the exponents are equal. A one bit right shift of the significand adds the value '+1' to the exponent.

   For example, consider two floating point numbers 1.5 and 0.5 in single precision format.

   Implied bit

   1.5 = 01.10000000000000000000000*2^{0}

   0.5 = 01.00000000000000000000000*2^{-1}

   The implied bit is not represented in binary floating point numbers, but is shown here for convenience.

   To add these two numbers, the exponents of the two numbers should be the same. The number with the lesser exponent is 0.5. Therefore, the significand of the number 0.5 is right shifted once, to make both the exponents equal as shown below.

   1.5 = 01.10000000000000000000000*2^{0}

   0.5 = 00.10000000000000000000000*2^{0}

2. Once both exponents are equal, the significands are added. Most adder/subtractor circuits can be used for this operation.

   01.10000000000000000000000*2^{0}

   00.10000000000000000000000*2^{0}

   010.00000000000000000000000*2^{0}

3. Now the result of the addition/subtraction is normalized by incrementing or decrementing the exponent field of the result. The normalized form of a floating point number can be represented as 1. *xxxxxx* X 2^{yyyy}, where *x* represents the significand field and *y* represents the exponent field. Thus the significand is either

shifted right and increment the exponent, or shifted left and decrement the exponent to obtain the normalized form such that there are no leading zeros in the result.

Therefore, the result in the example can be written in normalized form as

$1.5 + 0.5 = 2 = 01.00000000000000000000000 * 2^{-1}$

The significand of the result is stored as 23 zeros, as the '1' shown above is the implied bit.

4. After normalization, the result is checked for overflow and underflow exceptions. The overflow and underflow exceptions are determined by inspecting the exponent field of the result. If the exponent is too large to be represented within the reserved exponent field, then overflow occurs. If the exponent is too small to be represented within the reserved exponent field, then underflow occurs. For example, in IEEE single precision floating point number, the exponent of a binary number has to be in the range of [-126, 127]. This range is determined by the number of bits used for the exponent field in the representation of the number. If the exponent of the result is larger than this range, then the overflow exception occurs. If the exponent of the result is smaller than this range, then the underflow exception occurs. If there is any exception, it is indicated.
5. If there is no exception, then the significand of the result is rounded to the desired number of bits.
6. After rounding, the result is checked to see if it is normalized or not. If the result is still in its normalized form then the operation is done. Otherwise, the operation is again repeated from step 3.

From the six steps given above it can be easily observed that a floating point adder/subtractor requires a lot of hardware for comparing the exponents, normalizing, rounding etc. in addition to normal addition or subtraction of significands. On the other hand, a fixed point arithmetic unit requires only the addition/subtraction unit, as the designer will take care of the scaling factor throughout the design by making sure that the radix point of all operands are properly aligned.

Multiplication:

The multiplication operation in floating point arithmetic systems is comparatively simpler than addition / subtraction. The following steps are required for floating point multiplication.

1. The two biased exponents are added. The bias value is subtracted from the result to get the required new biased exponent.

   Consider an example of binary floating point multiplication of two 4-bit binary floating point numbers $1.1100*2^4$ and $1.0011*2^{-3}$. Bit '1' before the binary point is the implied bit in this example. In the first step, the two exponents are added to get the exponent of the result, 4-3 =1.
2. The two significands are multiplied. This is generally done using any binary multiplier.

   $1.11*1.0011*2^1=10.000101*2^1$.
3. The product is normalized if needed, by shifting the exponent either right or left.

   In the above example the significand is shifted one bit right to obtain the normalized form.

   $10.000101*2^1=1.0000101*2^2$.
4. Now the result is checked for overflow and underflow exceptions. If there are any exceptions, they are indicated in the result.
5. If there are no exceptions, the result is rounded to the desired value.

   The rounded value of the result of the above example is $1.0000*2^2$.
6. The result is checked for the normalized form. If the result is not normalized, then the process is repeated from step3.
7. If the result is normalized, the sign of the result is set positive if the signs of the inputs are same; if they differ, the sign is made negative.

Division:

The division operation in floating point arithmetic systems is almost the same as the multiplication process except that the two exponents are subtracted in the first step, and

the dividend is divided by the divisor in the second step. The division operation has to go through the same sequences of normalization, rounding and checking for the sign.

From the above description of the arithmetic operations of a floating point unit, it is clearly evident that it requires a lot of hardware for normalizing and rounding.

4.4 Comparison of Fixed Point Arithmetic and Floating Point Arithmetic Systems:

Fixed point arithmetic has the binary point fixed, and the designer has to take care of the scaling factor through out the design of a fixed point arithmetic system. The designer has to make sure that the binary point is in the same place throughout the design. Floating point arithmetic has a floating binary point, and the designer has less worry about the binary point. The hardware will take care of the position of the binary point through out the design [13]. Floating point arithmetic units contain necessary hardware to match the exponents, normalize the results and round the results. These hardware units of floating point arithmetic units make sure that the binary point is in proper position throughout the operation. Thus, fixed point arithmetic systems are more difficult to design from a designer's point of view than floating point arithmetic systems.

Even though a designer's task is easier in designing floating point arithmetic units, the hardware used and thus the area occupied is much more compared to fixed point units. Floating point units require additional hardware for shifting, rounding and normalization in addition to the basic arithmetic units. Thus from a hardware point of view, fixed point units have more advantages compared to floating point units.

The range and accuracy of floating point units is better than that for fixed point units. Thus, if the design has a high priority on the range and accuracy of arithmetic operands, then floating point units are preferable.

If the final design is to be modified again and again, then floating point units have the upper hand as the design time for fixed point units is high compared to that of floating

point units. As explained earlier, the scaling factor needs to be taken care of every time the design is modified, whereas, floating point units take care of the binary point them selves.

Thus from the above comparison, it can be concluded that if a design requires less range and precision and if it has to use less area and hardware and it does not need to be modified again and again, then fixed point arithmetic would be a better choice. Fixed point arithmetic units are a good choice when the design time doesn't matter. However, if the range and precision needed are significant and design time is a critical issue, then floating point units are preferred.

According to the given design requirements, the device to be developed needs to have less hardware that can occupy less area. As the design is for a hand held device, the power consumption should be less. The range required for the device is not especially high. The maximum value in the design will be 1800 and the minimum value is 0.000000844. The range can also be significantly reduced which will be shown in next section. The resolution required is 0.1 for all quantities of temperature, relative humidity, and the EMC. Considering all the above specifications, fixed point arithmetic seems to be a better choice if the range of the device can be reduced. The following section shows some scaling operations to be performed on the given Equations (4-1), (4-2) and (4-3) to reduce the range of all the coefficients so that the bit width of the whole design can be reduced.

4.5 Design Issues of Equations:

From Equations (4-1), (4-2) and (4-3) it can be seen that the minimum value to be represented is 0.000000844. This decimal value is to be multiplied by the positive square value of temperature, hence the result of the multiplication will always be greater than 0.000000844. The maximum value to be represented is 1800. For using fixed point arithmetic in the design, it is suggestible to have the same bit-width for all the registers in the data path of the design. This will actually help in reducing hardware a great deal. If variable width registers are used in the design, the scale factor of the values will continue

to change. Hence it would become a difficult job for the designer to keep track of the binary point. In order to use fixed bit-width registers through out this design, the bit-width has to accommodate a sufficient number of bits to hold the value 1800 in the integer part, and the value 0.000000844 in the fraction part. As the value of temperature goes negative in a given range, a sign bit is also needed to hold the value of the sign of the number. Considering all these factors, the bit width needed for the data path will be 35; 1 bit for the sign bit, 11 bits for the integer part of an operand, and 23 bits for the fraction part of an operand. A data path of 35 bit-width will consist a large amount of hardware, considering the fact that all the results of equations calculated will be much less in value and will need much less bit width. Thus, Equations (4-1), (4-2) and (4-3) are scaled and modified so that the range of the arithmetic will be much less and so will the bit-width. The steps included in scaling the equations are shown below.

The original equations are

$$W = 330 + 0.453T + 0.00415\,T^2$$
$$K = 0.791 + 0.000463\,T - 0.000000844\,T^2$$
$$K_1 = 6.34 + 0.000775\,T - 0.0000935\,T^2$$
$$K_2 = 1.09 + 0.0284\,T - 0.0000904\,T^2$$

$$EMC = \frac{1800}{W}\left[\frac{Kh}{1 - Kh} + \frac{K_1Kh + 2K_1K_2K^2h^2}{1 + K_1Kh + K_1K_2K^2h^2}\right] \quad (4\text{-}5)$$

Now by introducing a new variable *T'* in the above equations, where *T'* = *T*/1024, the new equations become

$$W = 330 + 463.872\,T' + 4351.5904\,T'^2$$
$$K = 0.791 + 0.474112\,T' - 0.8849\,T'^2$$
$$K_1 = 6.34 + 0.7936\,T' - 98.041856\,T'^2$$
$$K_2 = 1.09 + 29.0816\,T' - 94.79127\,T'^2$$

$$EMC = \frac{1800}{W}\left[\frac{Kh}{1 - Kh} + \frac{K_1Kh + 2K_1K_2K^2h^2}{1 + K_1Kh + K_1K_2K^2h^2}\right] \quad (4\text{-}6)$$

Because of introducing a new variable *T'*, it can be observed that the fraction parts of all the coefficients in the above equations, and there by the fractions of the values calculated using those coefficients, are increased. So now the least value of the coefficients is 0.041856. This value of 0.041856, when compared with the much smaller value 0.000000844, will require a much less number of bits for representation. Thus, the problem of representing a very small value is solved. But, still the maximum value to be represented is 1800, which can also be reduced as follows.

In Equations (4-6), by introducing new terms $W' = W/330$, $K' = K/0.791$, $K_1' = K_1/6.34$, $K_2' = K_2/1.09$, the equations become

$$W' = 1 + 1.4336\,T' + 13.1866\,T'^2$$
$$K' = 1 + 0.5993\,T' - 1.1188\,T'^2$$
$$K_1' = 1 + 0.1252\,T' - 15.4640\,T'^2$$
$$K_2' = 1 + 26.6804\,T' - 86.9645\,T'^2$$

$$EMC' = \frac{5.4545}{W'}\left[\frac{K'h}{1.2642 - K'h} + \frac{K_1'K'h + 1.7244K_1'K_2'K'^2h^2}{0.1994 + K_1'K'h + 0.8622K_1'K_2'K'^2h^2}\right] \qquad (4\text{-}7)$$

The equations (4-7) can be rearranged as follows:

$$W' = 1 + 1.4336\,T' + 1.6483\,(8 * T'^2)$$
$$K' = 1 + 0.5993\,T' - 1.1188\,T'^2$$
$$K_1' = 1 + 0.1252\,T' - 1.933\,(8 * T'^2)$$
$$K_2' = 1 + 1.6675\,(16 * T') - 1.3859\,(64 * T'^2)$$

$$EMC' = \frac{5.4545}{W'}\left[\frac{K'h}{1.2642 - K'h} + \frac{K_1'K'h + 1.7244K_1'K_2'K'^2h^2}{0.1994 + K_1'K'h + 0.8622K_1'K_2'K'^2h^2}\right] \qquad (4\text{-}8)$$

In the same way, Equations (4-3), used for correcting non linearity and temperature compensation, are also scaled and modified using the terms, $SO_T' = SO_T/1024$ and $SO_{RH}' = SO_{RH}/1024$. The new equations become

$$T'_{real} = 64(-0.625 + 0.16(SO_T'))$$
$$RH'_{linear} = -0.125 + 1.296(SO_{RH}) - 0.09175(SO_{RH})^2$$

$$RH'_{true}=64[(T'_{real}-0.390625)*(0.01+0.08192*SO_{RH}) +0.5*RH'_{linear}] \quad (4\text{-}9)$$

Modified Equations (4-8) and (4-9) show that the maximum value in the integer part is '1' where as the least value in the fraction part is 0.08192. Though the fraction parts of all the values of all the terms cannot be guessed, by observation of the coefficients of the equations it can be guessed that all the fraction parts can be represented with a less number of bits when compared with the original equations. Thus the overall bit width of the whole system is reduced a great deal.

The modified equations were tested using the MathCAD tool with every number, 16 bits in width. Every binary real number had a sign bit, 5 bit integer part, and a10-bit fractional part. The value of the new *EMC'* is almost the same for a majority number of cases, with an error of only 0.1 for a very few values in the required range. Thus, a fixed point arithmetic system with a 16-bit data path can be designed to calculate the EMC of wood using the new modified Equations (4-8) and (4-9).

4.6 Number System Used in the EMC Processor Arithmetic Unit Design:

Now that the type of arithmetic (fixed point) and the bit-width (16-bits) of the system have been decided, the next step is to determine the type of number system to use for the arithmetic unit design of the EMC processor. The arithmetic unit of the EMC processor unit must be able to add, subtract, multiply and divide positive and negative 16-bit operands. Binary arithmetic number systems which can accommodate both positive and negative operands include:

- Sign-magnitude binary number systems.
- Two's-complement number systems.
- One's-complement number systems.

A brief description of how fixed point binary numbers are represented in these number systems and their comparison is given in this section.

- Sign-magnitude representation: A signed magnitude representation uses a sign bit appended to the rest of the number representing the magnitude. The value of '1' in the sign bit position indicates that the number is negative and the value '0' in the sign bit indicates that the number is positive. For an *m*-bit signed-magnitude binary vector (number) the range of the values is $[-(2^{m-1}-1), 2^{m-1}-1]$ [10]. Primary advantages of signed magnitude representation are its intuitive appeal, conceptual simplicity, symmetric range. The basic disadvantage is that the subtraction operation must be implemented using a separate hardware from that used to implement addition. There is no way to perform subtraction by an addition operation in sign-magnitude representation. This can be avoided in other representations. Also, the signed magnitude representation has two representations for the value '0'. They are +0 and -0. This leads to special care for number comparisons or more hardware to detect -0 and avoid it. An example of signed-magnitude representation is shown below.

  Ex: "00000" indicates +0. "10000" indicates -0.
  "00010" indicates +2. "10010" indicates -2.

- One's-complement representation: In one's complement representation, positive numbers are represented as in a sign-magnitude representation, but the negative numbers are represented by inverting each magnitude bit of the number with the sign bit being '1'. The range of an *m*-bit one's complement number is $[-(2^{m-1}-1), 2^{m-1}-1]$. One's complement representation also has the disadvantage of two values for '0'. The values "0000" and "1111" represent +0 and -0 respectively. The subtraction operation in one's complement representation does not need additional hardware as in sign magnitude representation. It can be implemented by an addition operation. The number to be subtracted (subtrahend) is complemented and added to the minuend. The carry out of the sign-bit position

resulting from the addition of the subtrahend operator to the minuend operator becomes end-around-carry and is added to the least significant bit of the sum resulting from the first addition. Thus, two addition operations may be required to implement one subtraction operation.

Ex:

5 – 6 = - 1.

0101(5)
1001(-6)
_____
1110(-1)
_____

6 – 3 = 3.

0110(6)
1100(-3)
_____
0010
1
_____
0011(3)
_____

End-around-carry → 1

- Two's complement representation: Two's complementation representation of an *m*-bit binary vector is obtained by adding '1' to the ones-complement value of the binary vector. This representation produces the range of $[-2^{m-1}, 2^{m-1}-1]$ for an *m*-bit binary vector. This representation, like the one's complement representation avoids the use of extra hardware for a subtraction operation. It avoids the two values of '0'. Instead an extra negative value is included in the range. For example for a 4 bit vector the range is -8 to 7 and there is only one value for '0' which is "0000". The carry out bits of the sign-bit positions are ignored in two's complement representation. They do not have to be added to the least significant bit of the sum producing the carry out of the sign-bit position. Never are two summations required in generating a sum of two operands.

Ex:

5 – 6 = -1.	6 – 3 = 3.
0101(5)	0110(6)
1010(-6)	1101(-3)
1111(-1)	0011(3)

A comparison of all the above number systems show that with a sign-magnitude representation, subtraction can not be performed by an addition process as it can be done in the other two representations. Both signed magnitude and ones complementation representation of operands has two values for '0' which requires extra hardware for zero detection. A comparison of the power consumption of the types of representation shows that the switching activity of the binary bits, which leads to more power consumption is more in twos complement representation when the value is switched between -1 and +1 more times [14]. But in the design of the EMC processor, there is very little probability that the values switch on between negative and positive values so frequently. Hence this would have very little effect on the design. Thus, two's complement representation of operands is a better choice for the EMC processor design compared to signed magnitude and one's complement representation of operands.

4.7 EMC Processor Arithmetic Unit Design:

The calculation of EMC requires many additions, subtractions, multiplications and divisions. Therefore, an arithmetic unit is required for the EMC processor that should be able to perform the calculations and meet the design requirements. Two different versions of the arithmetic unit are designed for the EMC processor. They are:

- EMC processor parallel arithmetic unit.
- EMC processor serial arithmetic unit.

These arithmetic units use fixed point arithmetic and two's complement number system as discussed in the previous sections. A brief description of the two types of arithmetic units is given below.

EMC processor parallel arithmetic unit: Equation sets (4-8) and (4-9) show that the calculation of EMC requires multiple additions, subtractions, multiplications and divisions. One of the design specifications indicate that the computation of Equation sets (4-8) and (4-9) is to be repeated every one or two seconds. So, with a clock cycle period on the order of micro seconds or milli seconds, it would be sufficient to use a single adder/subtractor, single multiplier, and single divider to implement all arithmetic operations in the equations. Therefore, an adder/subtractor, multiplier, and divider are designed to implement the parallel arithmetic unit.

From Equation sets (4-8) and (4-9) it can be observed that the some of the terms of the equations are independent of each other and some of the terms are dependent on each other. For example, consider the calculation of *W'* in Equation set (4-8).

$$W' = 1 + 1.4336T' + 1.6483(8 * T'^{2})$$

If one multiplier, one adder/subtractor and one divider are used for the design, then to compute the value of *W'* the value of $1.6483(8 * T'^{2})$ is computed at first. In second step, the adder/subtractor and the multiplier can be used simultaneously to calculate the terms $1 + 1.6483(8 * T'^{2})$ and $1.4336T'$ as these two terms are not dependent on each other. EMC processor parallel arithmetic unit uses this principle to calculate the value of EMC. It utilizes the adder/subtractor, the multiplier and the divider simultaneously when the terms in the equation to be computed are independent of each other as shown in the example above. This method reduces the time taken to compute the EMC compared to that of calculating every term in the equation sets serially.

EMC processor serial arithmetic unit:

The functionalities of adder/subtractor, multiplier and divider can be implemented using a single arithmetic unit that contains an adder/subtractor and a special purpose shift register. EMC processor serial arithmetic unit is designed using this principle. Thus, the hardware is reduced compared to that of the EMC processor parallel arithmetic unit. But, it is not possible to perform the parallel computation of terms that are independent of each other as in the parallel arithmetic unit because the same arithmetic unit is used for

addition, subtraction, multiplication and division operations. All the terms in the equation sets are computed one at a time. Therefore the time taken to compute the equation sets (4-8) and (4-9) using the serial arithmetic unit will be more compared to that of parallel arithmetic unit.

The following sections will give a detailed design description of the EMC processor parallel arithmetic unit and the EMC processor serial arithmetic unit. Both versions of the arithmetic unit are compared in terms of the area on chip and speed.

4.8 EMC Processor Parallel Arithmetic Unit Design:

This section explains in detail the design of the parallel arithmetic unit system used in the EMC processor. An adder/subtractor, multiplier, and divider are designed to implement the parallel arithmetic unit. The following sections detail the designs of individual arithmetic functional units, their hardware implementation, and their post place and route simulations using the Xilinx ISE 6.2.3i/Modelsim 5.7g.

4.8.1 Design of Adder/Subtractor:

The adder/subtractor functional unit can be implemented basically using (i) a combinational logic approach or a (ii) sequential logic approach. Addition and subtraction can be implemented in parallel using the combinational logic approach, and serially using a sequential approach. In general, most arithmetic adder circuits are designed to operate in parallel as they will operate considerably faster. Even though the hardware used for the parallel combinational circuits is more when compared with serial sequential circuits, it is often not prohibitively excessive. Different types of parallel combinational adder circuits are considered for this design. Some of them are

- Ripple carry adder.
- Carry-look-ahead adder.
- Carry-save adder.
- Carry select adder.
- Carry skip adder.

Of all the above adders, the ripple-carry adders and carry-look-ahead adders use much less hardware compared to others [15]. In this section the functional descriptions of ripple carry adders and carry-look-ahead adders are presented and compared.

Ripple-carry Adder/Subtractor:

A ripple-carry adder uses cascaded full adders to produce the required sum. A full adder has three inputs a_i, b_i and c_i, where a_i and b_i are the bits from operand inputs and c_i is the carry input from the previous full adder stage. The sum s_i and carry c_i of a full adder stage are given by

$$s_i = a_i \text{ xor } b_i \text{ xor } c_{i-1}$$
$$c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1} \quad (4\text{-}10)$$

The functional diagram of an N-bit ripple-carry adder is shown in Figure 4-2. The same circuit can also be used as a subtractor by complementing the input b_i and providing the value c_0 as '1'. The first full adder FA(0) stage produces the sum (s_0) and carry (c_1) based on its inputs a_0, b_0, and c_0 using Equations (4-10). The carry output c_1 becomes a carry input to the second full adder FA(1). Thus all the full adders are connected in a parallel manner to produce the desired output.

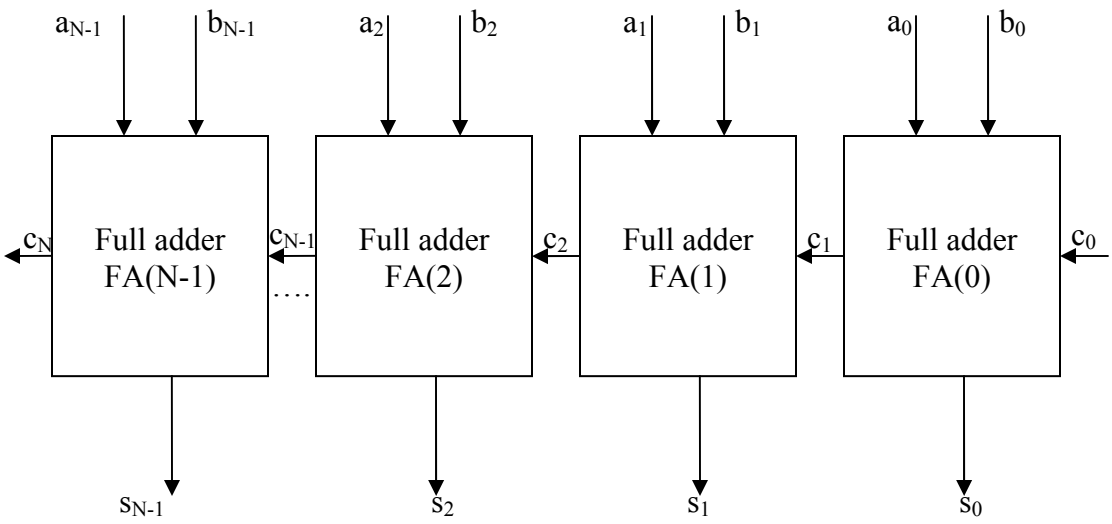


Figure 4-2: Functional block diagram of an N-bit ripple carry adder/subtractor.

As shown in Figure 4-2, the carry bit "ripples" right to left from one stage to another stage in a serial fashion and there by the name "ripple-carry adder". The delay through the entire adder depends on the number of logic stages that must be traversed. Thus for adders with more bit-width, the delay of the carry will be proportional to the bit-width of the adder and will play a serious part in design choice of the adder.

As it can be seen from Figure 4-2, the ripple carry adder uses N stages of logic, or N full adders for an N-bit adder. If t_{carry} denotes the delay from c_0 to c_1 and t_{sum} denotes the delay from c_0 to s_0, then the delay of the ripple-carry adder t_{delay} is approximated using equation (4-11) [15].

$$t_{delay} = (N-1)\, t_{carry} + t_{sum} \qquad (4\text{-}11)$$

Carry-look-ahead Adder:

The basic operation of a carry-look-ahead adder can be explained as follows. Consider equation (4-10), an equation for c_{i+1} can be written as

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i = a_i b_i + c_i(a_i + b_i)$$

Assume a carry propagate term p_i and carry generate term g_i where $p_i = a_i + b_i$ and $g_i = a_i b_i$. Then the value of c_{i+1} is

$$c_{i+1} = g_i + c_i p_i \tag{4-12}$$

Now, c_i can be represented as a function of lower order terms. For example

$$\begin{aligned} c_1 &= g_0 + p_0 c_0, \\ c_2 &= g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0, \\ c_3 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \ldots \end{aligned} \tag{4-13}$$

From Equations (4-13) it can be seen that the carry terms c_1, c_2, c_3… can be generated from the carry generate and carry propagate terms only. This would reduce the ripple-carry delay that is present in a ripple-carry adder. The second full-adder need not wait for the first full-adder to generate the carry bit. But this type of adder requires more hardware to generate the carry terms individually for every full adder stage of an N-bit adder.

The carry terms in equation (4-13) can be continued up to the fan-in limit 'r'.

The over all delay of the carry-look-ahead adder is given by $4[\log_r N]$ where 'N' is the number of bits of the adder [15]. A direct delay comparison of an N-bit ripple-carry adder to that of an N-bit carry-look-ahead adder is as follows:

$$t_{\text{delay-ripple-carry-adder}} = (N-1)\, t_{\text{carry}} + t_{\text{sum}}$$

$$t_{\text{delay-carry-look-ahead-adder}} = 4[\log_r N]$$

Thus, from delay comparisons of the two adders, it can be seen that delay is reduced a significantly in carry-look-ahead adder. But, as will be seen, the hardware required is more.

Area or Gate Count Comparison of Ripple Carry and Carry-Look-Ahead Adders:

Figure 4-3 shows an area (equivalent gate count) comparison of ripple-carry adders and carry-look-ahead adders for different bit-widths [16]. For a bit width of 16 chosen for the EMC processor arithmetic unit design, it can be seen that the carry-look-ahead adder

requires more hardware than a ripple carry adder. As the design requires less hardware, a ripple carry adder is chosen for the EMC processor arithmetic unit. The EMC processor design requires the computation of the EMC approximately every one or two seconds. Thus, even though the ripple-carry adder has more carry propagation delay compared to that of carry-look-ahead adder, its effect is negligible.

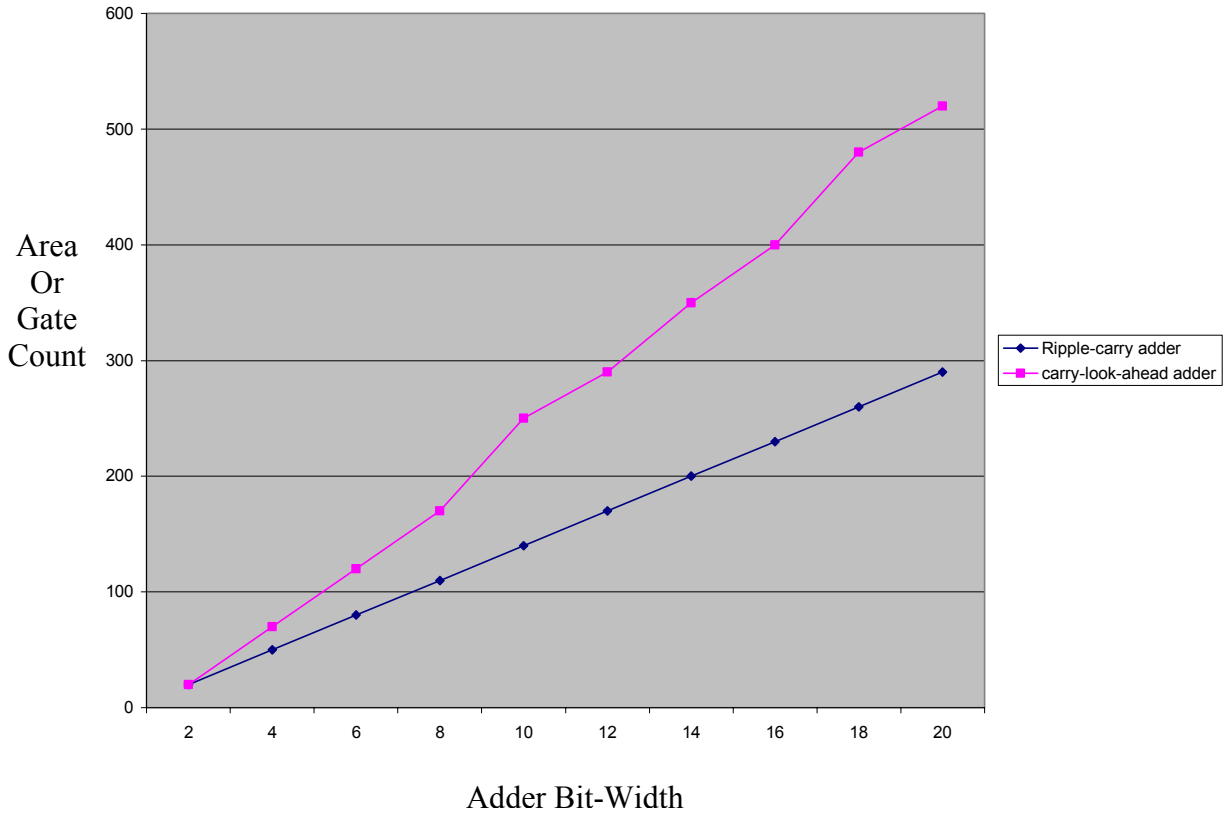


Figure 4-3: Area or gate count comparison of ripple carry and carry-look-ahead adders.

VHDL Simulation and Validation of Ripple-Carry Adder:

A ripple-carry adder of bit-width 16 was implemented for the EMC processor design. Two registers of bit-width 16 are used to hold the input operands and one 16 bit register is used to hold the value of the output operand. A one-hot state controller was designed to control the operations between the registers and ripple carry adder. The synthesis and implementation results of the final adder/subtractor are shown in this section. Example

post implementation VHDL simulation results for the ripple carry adder/subtractor are shown in Figure 4-4.

When the 16-bit adder was implemented to a Spartan2E xc2s200E FPGA, the implementation reports obtained using Xilinx ISE 6.2.3i CAD tool set show the device utilization are as follows.

Number of Slices:	64 out of 2352	2%
Number of Slice Flip Flops:	78 out of 4704	1%
Number of 4 input LUTs:	102 out of 4704	2%

From the above device utilization summary it can be observed that the number of slice flip-flops used is 78. The flip-flops used to hold the operands and the flip-flops used in one-hot state controller account for the total number of slice flip flops.

The post implementation VHDL simulation for one combination of the inputs to the adder/subtractor is shown in Figure 4-4. The ripple-carry adder/subtractor was coded in VHDL using the Xilinx 6.2.3i CAD tool set. The ripple-carry adder/subtractor was tested for various inputs to the adder using Modelsim 5.7g simulator. The main aim of the testing was to validate the functionality of the adder for various inputs. The adder was tested for different input combinations and validated.

The inputs ‘a_in’ and ‘b_in’ are the 16-bit binary operands given as inputs to the adder. The control signal ‘add_or_sub’ indicates the adder whether to generate a sum of ‘a_in’ and ‘b_in’ or to generate a difference of ‘a_in’ and ‘b_in’. Figure 4-4 shows the post implementation VHDL simulation for an instance of ‘a_in’ = “0000000010001010” and ‘b_in’ = “0000000001000100”.

The number ‘a_in’ = “000000.0010001010” represents $1*2^{-3}+1*2^{-7}+1*2^{-9}=0.1347$ in fixed point format. The number ‘b_in’ = “000000.0001000100” is $1*2^{-4}+1*2^{-8}=0.066$ in fixed point format. The addition of these two numbers should be 0.1347 + 0.066 = 0.20. The subtraction operation between these two numbers should result in 0.1347 – 0.066 = 0.06.

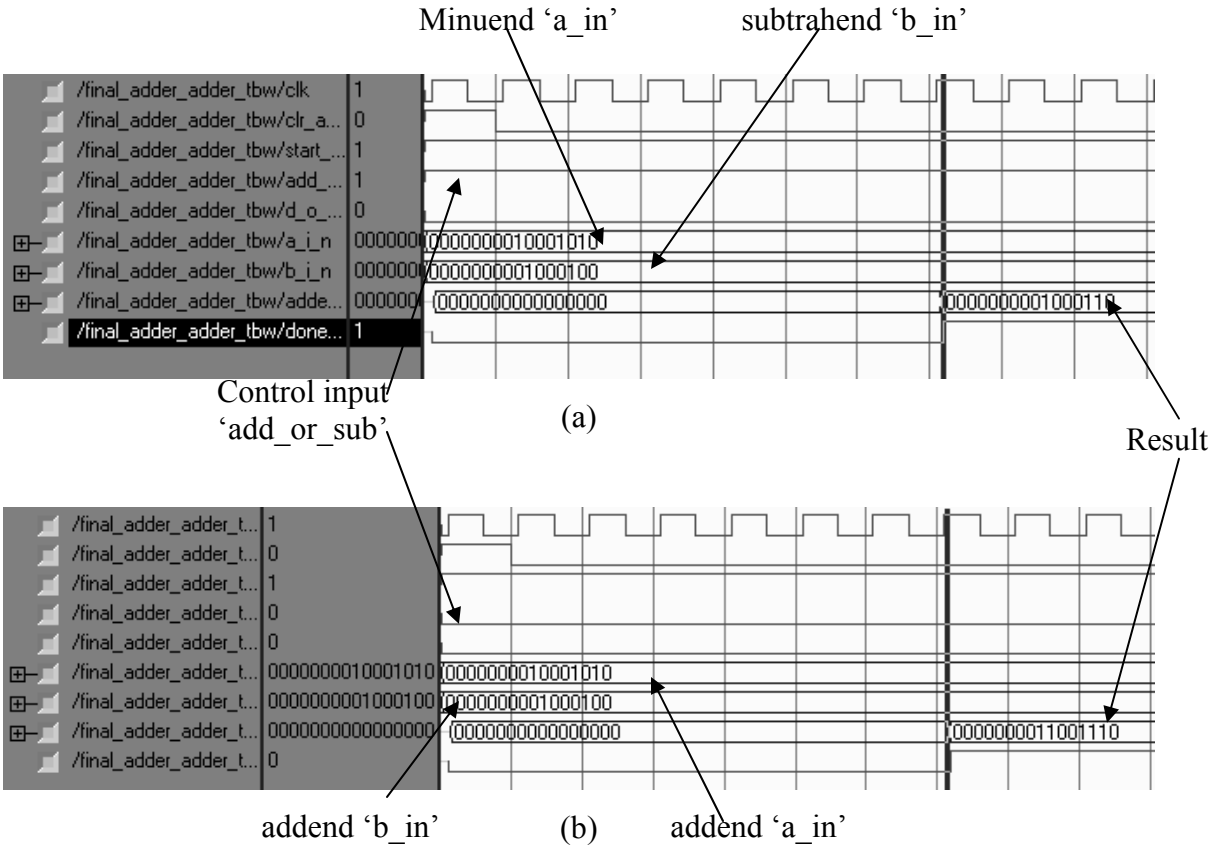


Figure 4-4: Post-place-and-route simulation for ripple-carry (a) subtractor and (b) adder.

From Figure 4-4 it can be observed that for two binary vectors ‘a_in’ =“000000.0010001010” and ‘b_in’ = “0000000001000100”, when the control input ‘add_or_sub’ is high the subtraction operation is done giving the result “000000.0001000110” =0.068; whereas when the control input is low the addition operation is done giving the result “000000.0011001110” =0.2. Thus, the ripple-carry adder/subtractor is tested and validated.

4.8.2 Design of Multiplier:

Multiplication is a most important computer arithmetic operation and also expensive. It is used in many applications including Computer Graphics, Digital Signal Processing and Process Control etc. Multiplication can be done using one of two approaches. They are

either combinational or sequential. Though the decision between the combinational approach and sequential approach was not very important the case of the adder, it is very important in case of multipliers and dividers. Combinational multipliers occupy more area on chip which often prohibits them from being used in many applications. The area occupied by combinational multipliers on a chip increases exponentially with increasing bit widths of the multiplier operands. Figure 4-5 shows a general comparison of combinational and sequential multipliers in terms of the area used and different bit-widths [16].

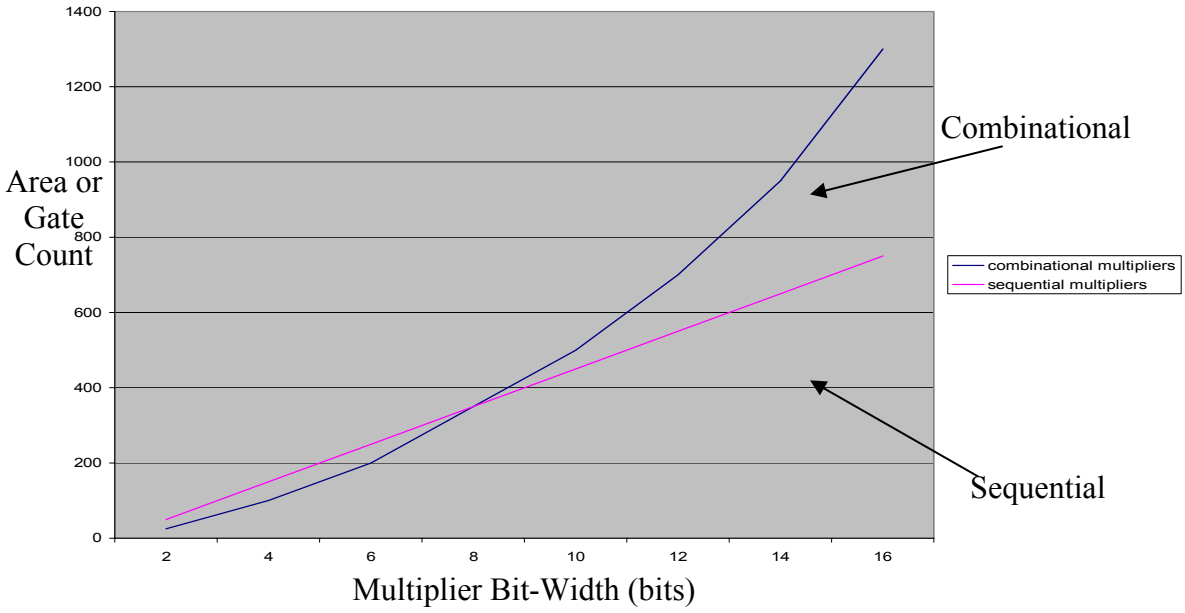


Figure 4-5: Rough comparison of combinational and sequential multipliers in terms of area for different bit-widths.

Sequential multiplication generally takes a finite number of clock cycles to perform the function, whereas combinational multiplication can be completed in less than a clock cycle. Thus, in applications where speed is a very important criterion, the combinational approach is followed. But in applications such as the EMC processor, speed is not of essence. Delay is not one of the major criterions for the current design. Hence a sequential multiplication approach is used.

The two most commonly used sequential multiplication algorithms are

1. Shift and add algorithm.
2. Booths multiplication algorithm.

Shift and add multiplication is a very basic algorithm in which only one register is used to hold the value of partial products. The partial product is added to the multiplicand if the value of the multiplicand bit considered is '1'. Partial product is shifted to right instead of left shifting the multiplicand. It uses a counter to determine the number of iterations required to compute the result.

Booths multiplication algorithm reduces the number of partial products required in any multiplication [17]. The sequential Booths multiplier does not use much more hardware than shift and add multiplier. It requires an extra flip-flop which is appended to the least significant bit of the multiplier for the double-bit inspection of the multiplier. It reduces the number of iterations required and thereby reduces the delay. Various Booth encoding algorithms are available such as Booth 2, Booth 3, and Booth 4 etc. The Booth 2 algorithm inspects two bits at a time where as Booth 3 inspects 3 and Booth 4 inspects 4 etc. Though the delay is reduced using the higher order encodings, the hardware required is also more. Thus the Booth 2 multiplication algorithm is used in the design of the EMC processor arithmetic unit.

Figure 4-6 shows a high level functional diagram of the Booth 2 multiplier used in the EMC processor design. The steps involved in the multiplication algorithm are as follows.

1. Load the register ABCO, multiplicand, and counter. The register ABCO is a 33-bit register with CO, an extra flip-flop appended to it as the least significant bit. In the first step, register ABCO is loaded with the value of the multiplier in the bit locations 16 to 1 of the register ABCO. The register 'M' is loaded with the value of 16-bit multiplicand. The counter is loaded with a value of "01111" for 16 X 16 binary multiplication. The appended flip-flop CO is loaded with value '0'.
2. Check the last two bits of register ABCO. The bits at the locations '0' and '1'of the register ABCO are checked by the controller. The value at location '0' is the value of the flip-flop CO.

3. If the last two bits of register ABCO are “10”, then the first 1 in a string of 1’s has been encountered in the multiplicand. This requires a subtraction of the multiplicand from the partial product stored in the register ABCO (Bit locations 31 to 16). In this case the multiplicand is subtracted from register A (Bit locations 31 to 16 of register ABCO). If the last two bits are “01”, the first 0 in a string of 0’s has been encountered in the multiplicand. This requires the addition of the multiplicand to the partial product in register A of register ABCO. When the last two bits of register ABCO are “00” or “11” no action is necessary and so the next shift occurs.
4. Arithmetic right shift the register ABCO. The arithmetic right shift ensures that the most significant bit of the register ABCO before the shift is duplicated into the most significant bit of the register ABCO after the shift. This step ensures that there is no sign change in the result. Decrement the counter by ‘1’.
5. If the value of the counter is not ‘0’ repeat the steps from step 2. Otherwise, store the result in the register ‘Result’.

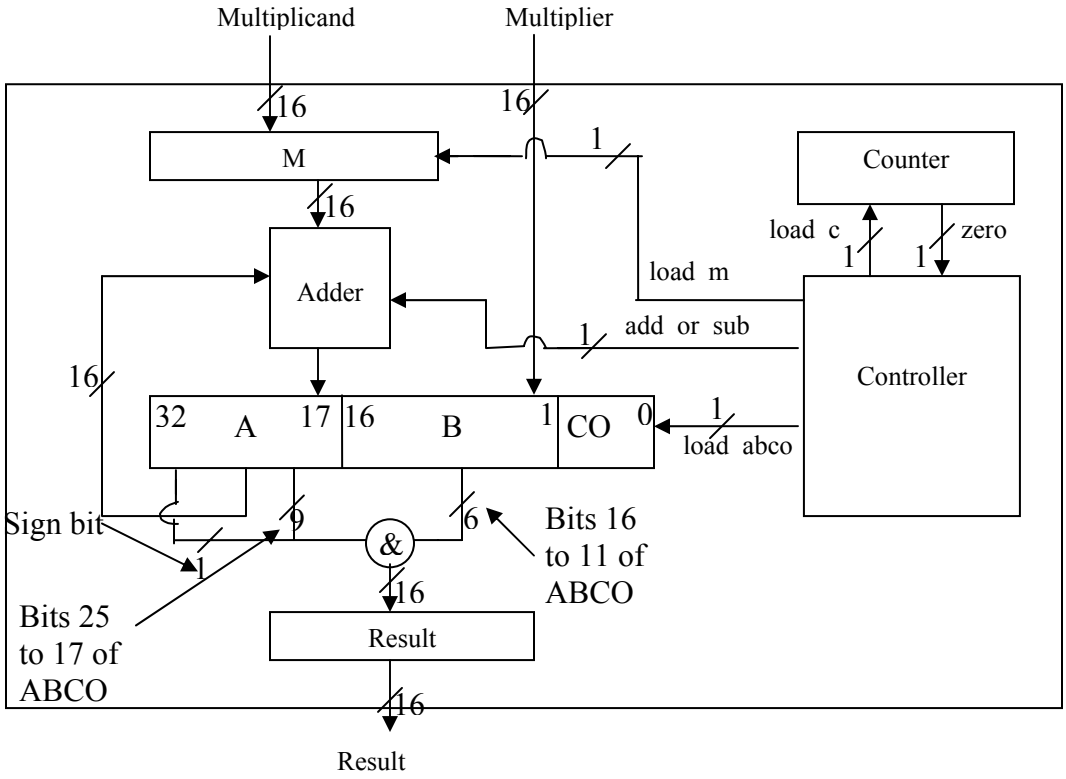


Figure 4-6: High level functional diagram of Booth 2 multiplier.

In the first step the multiplicand is loaded into the register M. The multiplier is loaded into register B of register ABCO (from bit locations 16 to 1). 'CO' is the additional flip-flop involved to perform the double-bit inspection. The counter is loaded with a value of 17. The value to be loaded in the counter is determined by the number of bits in the multiplier and the multiplicand. As the multiplier required for the EMC processor is 16 X 16, the counter should be loaded with a value of 16 + 1 = 17. The last two bits of register ABCO are checked. If the bits are "10" then the controller asserts add_or_sub signal causing the subtraction function. Then the adder performs the subtraction operation A – M. If the bits are "01" then the operation is A + M. If the bits are "00" or "11" then no action is required. In the next step, the register ABCO is arithmetic right shifted once, and the counter is also decremented by one. The same procedure repeats till the counter value is zero. The result of the

multiplication, stored in register AB is a 32 bit wide product for a 16- bit multiplicand and 16-bit multiplier. But the result taken out to the register 'Result' is 16 bits.

As explained earlier, multiplication in fixed point arithmetic of a multiplicand of scale factor s_1, with multiplier of scale factor s_2 produces a result with scale factor $s_1 + s_2$. In the current design, the scale factor of the multiplicand is 10 and that of the multiplier is also 10. Hence the scale factor of the 32 bit result is 20. Hence the 32 bit result stored in register AB has 1 sign bit, 11 bits for the integer part of the product, and 20 bits for the fraction part of the product. But, the result of the multiplication has to be used again as the input to the multiplier as per the design requirements. Hence the final result is to be reduced from 32 bit to 16 bit with 1 sign bit, 5 integer bits (bits 25 to 21 of the product in register ABCO) and 10 fraction bits (bits 20 to 11of the product in register ABCO).

Fortunately, the nature of the calculations in Equations (4-4) helps solve this problem. It can be seen that the result of every multiplication operation is less than the greater of the multiplicand or the multiplier.

For example, consider the multiplication which involves the largest number in the equations: $1.3859*64*T'^2$. For the maximum value of T' in the range, which is 210/1024 = 0.205, the value of T'^2 is 0.042. Now, T'^2 is left shifted 6 times, to get the value $64*T'^2$ and the value of the result is 2.688. Now it can be seen that the result of $1.3859*64*T'^2$ is 3.6574, which is much less than the larger of the two inputs. All the other multiplications in Equations (4-3) also follow the same trend, and produce results which fit in the selected 16-bit fixed point format. The rest of the multiplication terms which are similar to ($1.3859*64*T'^2$), such as ($1.6483*8*T'^2$), ($1.933*8*T'^2$), ($1.6675*16*T'$) are also performed in the same fashion. First, either the value of T' or T'^2 is left shifted with the specified value, and then it is multiplied with the coefficients. This method of multiplication reduces the integer bit width required to a maximum of 5.

Based on the properties of the equations as explained above, the result of the multiplication that is stored in register AB can be safely truncated, without losing any precision. Hence from register AB, the sign bit, and bits from register positions 25 to 11 are loaded into the result register. Thus, the Booth 2 multiplication algorithm produces a 16-bit result.

Logic Resource Comparison of Booth 2 Sequential Multiplier and Combinational Multiplier:

Results of implementing the Booth2 multiplier shown in Figure 4-6 to a Xilinx Spartan XC2S200E are shown below. The total number of flip-flops used is 107. The flip-flops used to hold the input operands, shift register, counter, output operand and the one-hot state controller used to control all these hardware units account to the total number of slice flip-flops used in the design of the Booth 2 sequential multiplier.

Number of Slices:	99 out of	2352	4%
Number of Slice Flip Flops:	107 out of	4704	2%
Number of 4 input LUTs:	177 out of	4704	3%
Number of bonded IOBs:	52 out of	146	35%

Total equivalent gate count for design: 1,948

A combinational multiplier can be obtained in VHDL by using the '*' operator. The Booth2 sequential multiplier and combinational multiplier are compared in terms of the resources used in Figure 4-7. The total number of slice flip-flops used in combinational multiplier is zero. But, the total equivalent gate count of the combinational multiplier is found to be 3310, whereas that of Booth2 sequential multiplier is 1948, which shows that the Booth 2 sequential multiplier designed for the EMC processor design occupies less area on chip compared to that of a combinational multiplier.

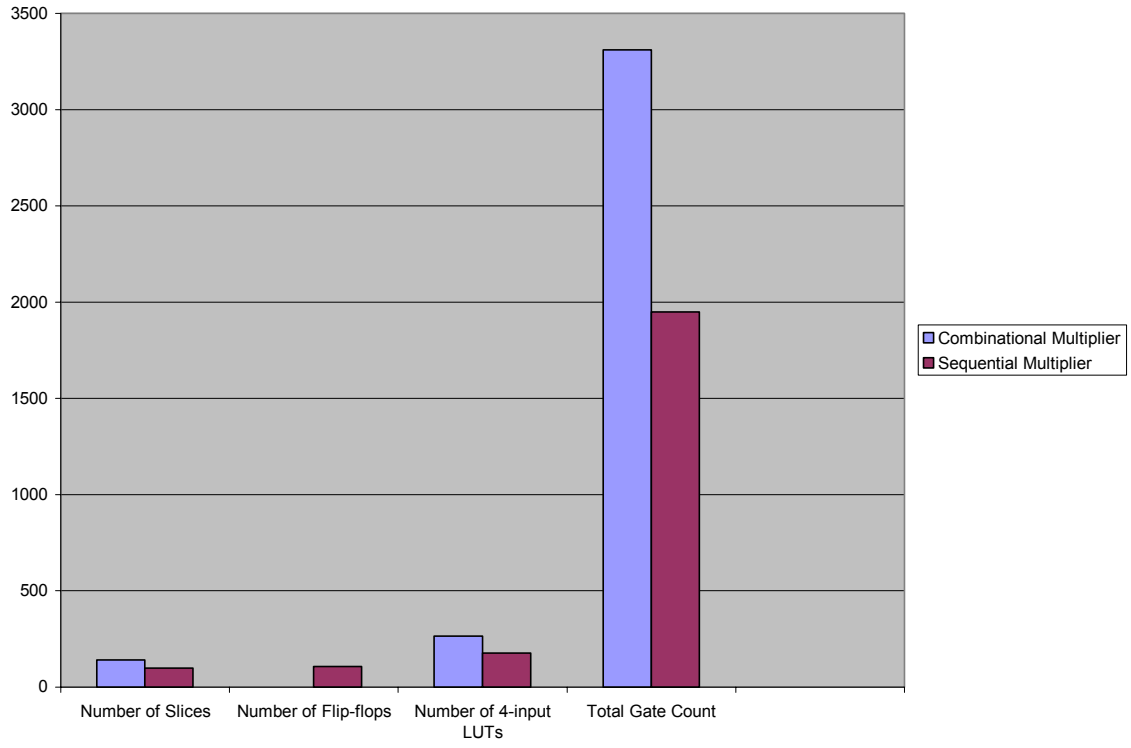


Figure 4-7: Comparison of Booth 2 and Combinational multipliers in terms of usage of device resources.

Post implementation VHDL simulation of Booth 2 multiplier:

The sequential Booth 2 multiplier shown in Figure 4-6 was developed in VHDL using Xilinx 6.2.3i CAD tool set. The VHDL design was implemented to a Spartan 2E XC2S200 FPGA. The design was tested for functionality using ModelSim 5.7g tool. The main objective of the testing was to validate the design for different combinations of input operands. The design was tested and validated for different combinations of inputs. Figure 4-8 shows an instance of the VHDL post implementation simulation of the Booth 2 sequential multiplier.

The multiplicand input that is loaded to the register M is “000001000111001”. This fixed point binary number represents the value:

$000001000111001 = 1*2^{0}+0*2^{-1}+0*2^{-2}+0*2^{-3}+1*2^{-4}+1*2^{-5}+1*2^{-6}+1*2^{-10}=1.1103$

The multiplier input that is loaded to the register ABCO of the sequential Booth 2 multiplier is “0000000001111111”. This fixed point binary number represents the value:

$0000000001111111=1*2^{-4}+1*2^{-5}+1*2^{-6}+1*2^{-7}+1*2^{-8}+1*2^{-9}+1*2^{-10}=0.1240$

Therefore, the result of the multiplication should be 1.1103*0.1240=0.1376.

From the post implementation simulation it can be seen that the result of the multiplication is “0000000010001101”. The value of the number in fixed point arithmetic is:

$0000000010001101=1*2^{-3}+1*2^{-7}+1*2^{-8}+1*2^{-10}=0.1376$

This result matches the required value and hence proves the functionality of the Booth 2 sequential multiplier.

Thus, the Booth 2 sequential multiplier is tested and validated.

Post implementation simulation of Booth 2 multiplier: The VHDL post implementation simulation of sequential Booth 2 multiplier is shown in Figure 4-8.

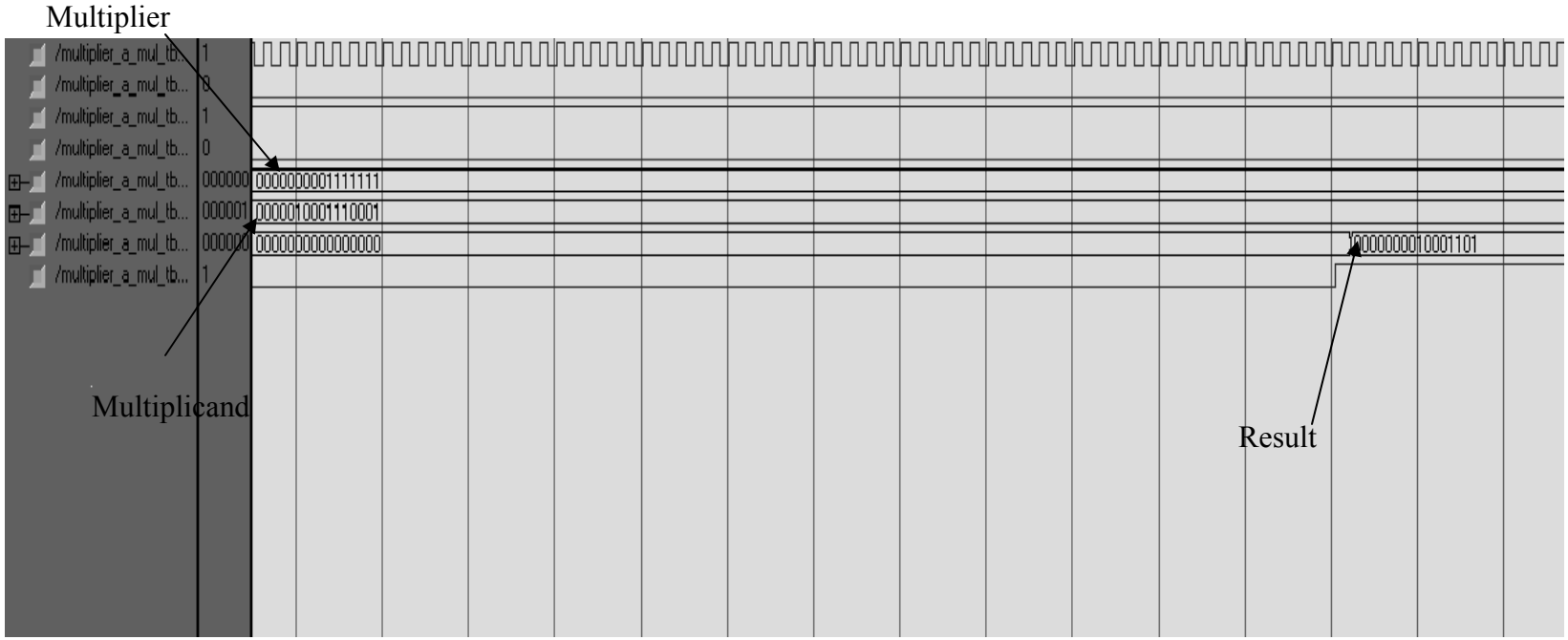


Figure 4-8: Post implementation simulation of sequential Booth 2 multiplier.

4.8.3 Design of Divider:

As with multiplication, division can be implemented using two approaches. One is a combinational approach and the other is a sequential approach. The combinational approach uses more hardware as compared to the sequential approach. To reduce hardware/area, the sequential approach will be used in the design of the EMC processor arithmetic unit.

The simplest and widely used division algorithms are digit recurrence algorithms. Generally, digit recurrence algorithms generate a fixed number of quotient bits in each iteration. The implementations of digit recurrence algorithms are of low complexity and small area [18]. But, they have more latency. The basic shift-add-subtract algorithm is one such digit recurrent algorithm. The arithmetic unit in the EMC processor uses the shift and subtract algorithm for division. A high level functional diagram of the divider used in the EMC processor design is shown in Figure 4-9.

The divider shown in Figure 4-9 is a 15 bit unsigned divider. A two's complement to unsigned converter is used at the input stage of the divider. The sign bits are taken out separately, and the result of the sign bit is found by 'xor' operation between the two operand sign bits. The magnitude of the twos complement number is converted into unsigned format depending on the sign. The unsigned dividend and divisor are given as inputs to the 15 bit divider as shown in Figure 4-9.

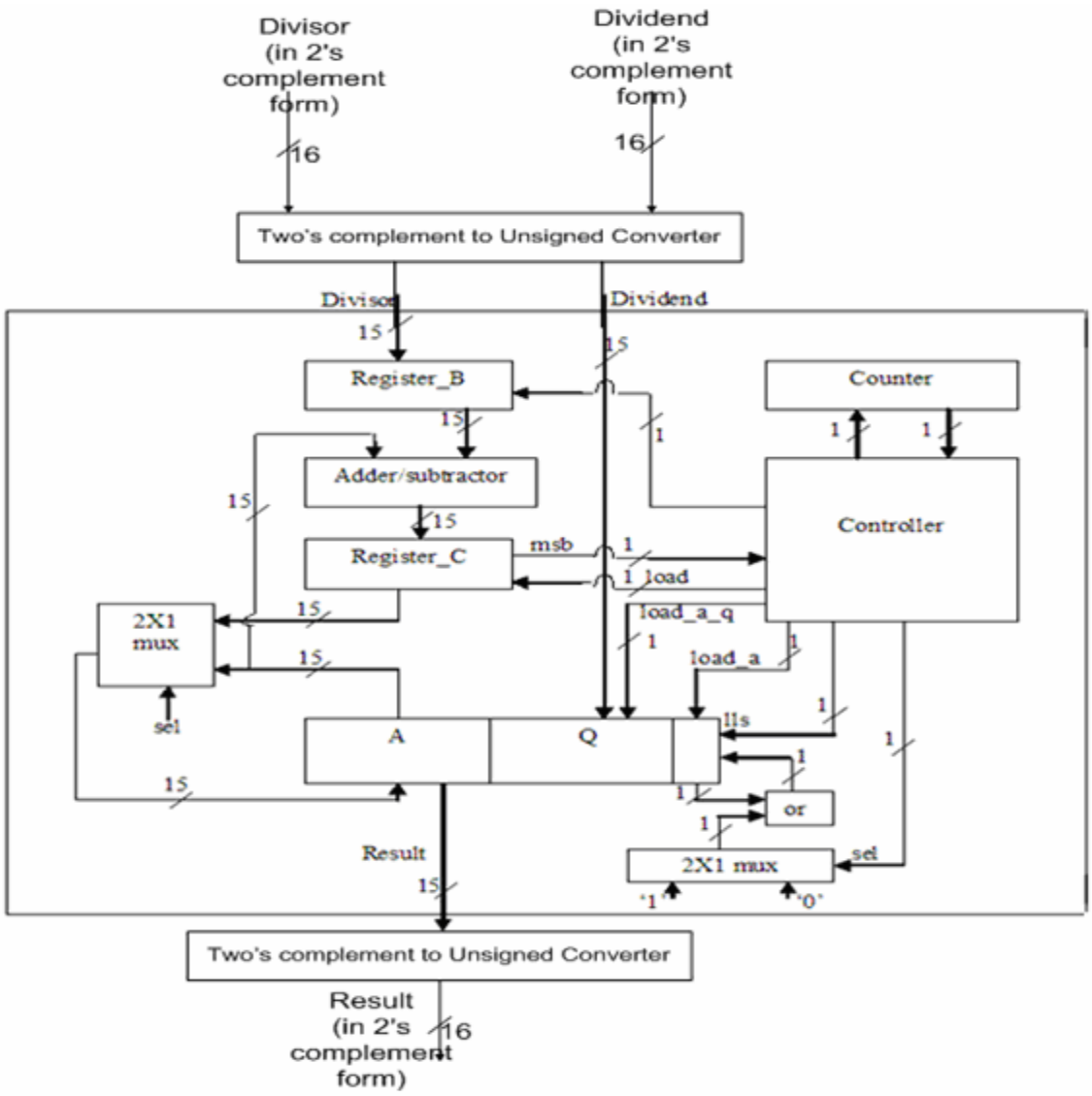


Figure 4-9: High level functional diagram of a shift-and-subtract sequential divider.

The steps in the division operation are as follows.

1. The divisor is stored in Register_B. The 15-bit dividend is loaded into the 30-bit Register AQ. The dividend is loaded to bit locations 24 to 10 of the Register AQ. The counter is loaded with a value of 14. When given a control signal, the counter starts decrementing by one on each clock cycle.

2. Contents of register A, and Register_B, are fed to subtractor. The result of the subtraction is stored in register Register_C. The most significant bit ‘msb’ of the result is checked in the one-hot type controller. If ‘msb’ is zero, then the result is overflow. Otherwise, go to step 3.
3. The register AQ is logically left shifted once, and the counter is decremented by one.
4. The value of the counter is checked by the controller. If the value is zero, then go to step 5. Otherwise, go to step 6.
5. The subtraction operation is done between Register A and Register_B, and the ‘msb’ bit is checked again. If the contents of register A are greater than that of Register_B, then the result in Register_C is stored in Register A and the least significant bit of register AQ is made ‘1’. The result is stored in register AQ. The quotient value is stored in register Q and the reminder in A.
6. A subtraction operation is done between Register A and Register_B. If the contents in register A are greater than that of Register_B, then the result in Register_C is stored in Register A. And the least significant bit of register AQ is made ‘1’. The process is repeated from step 3 again.

The result obtained from the above unsigned division is converted back to twos complement form, depending on the sign bit. But, the above unsigned division is for integers. As the numbers used are fixed point real numbers, some modifications are needed for the above divider to accommodate for fixed point real numbers. In general in the integer division of 15 X 15 divider, the dividend is concatenated with 15 most significand zeros to store the value in 30 bit register AQ. If the same approach is followed, for fixed point real numbers also, then the quotient value just shows the integer part of the result.

Thus, to accommodate fixed point real numbers, while loading the dividend into the register AQ, the 15-bit dividend is not concatenated with 15 most significant zeros. Five

most significant zeros and 10 least significant zeros are concatenated to the 15 bit dividend in register AQ. The 5 bits are concatenated to the integer part and the 10 bits are concatenated to the fraction part. This approach can be explained through the following simple example.

Consider two 5- bit fixed point real numbers, “010.01” and “001.10”. Assume that the scale factor is ‘2’ for both the numbers.
Hence, the values are $010.01 = 0*2^2 + 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25$
$001.10 = 0*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} = 1.5$

An unsigned integer binary division between these two numbers 010.01/001.10 yields the value ‘1’ indicating that the result is ‘1’. In unsigned division five most significant zeros are appended to the value “010.01”. Thus register AQ will have “00000010.01” as the dividend. The value 00000010.01/001.10 is “000.01”, which in fixed point representation is 0.25. But the value of 2.25/1.5 is not 0.25.

Now if a different approach from that discussed above is followed, three most significant, and two least significant zeros are appended to the dividend value in register AQ. Thus the value in register AQ is “000010.0100”. The value of the divisor is “001.10”. Now the result of the unsigned integer division between “000010.0100” and “001.10” is 000010.0100/001.10 = 001.10, which is 1.5. This value is correct as 2.25/1.5=1.5. Thus by appending the number of zeros that are equal to the number of integer bits on the most significant side and the number of zeros that are equal to the number of fraction bits on the least significant side, a correct result is obtained for fixed point real numbers using integer division hardware.

Logic Resource Comparison between Sequential and Xilinx IP Core dividers:
The shift and adder subtractor shown in Figure 4-9 is developed in VHDL, and synthesized to Xilinx Spartan2E FPGA using Xilinx 6.2.3i CAD tool set. The

implementation reports of the divider implemented to a Spartan2E xc2s200E are shown below. The total equivalent gate count for the designed divider is 1449.

Number of Slices:	74 out of 2352	3%
Number of Slice Flip Flops:	84 out of 4704	1%
Number of 4 input LUTs:	126 out of 4704	2%

A comparison of sequential divider used, and a pipelined divider designed using IP core generator, in terms of device resources is shown in Figure 4-10. The Xilinx IP Core Pipelined Divider is available for Spartan2E FPGA family from Xilinx design library. The pipelined library produces the result in less than a clock cycle and uses maximum depth of pipelining. Any such IP core divider can be used in the design of EMC processor arithmetic unit for the required division operation. But, the following comparison of logic resources of both dividers show that the shift and subtract sequential divider uses less area on chip.

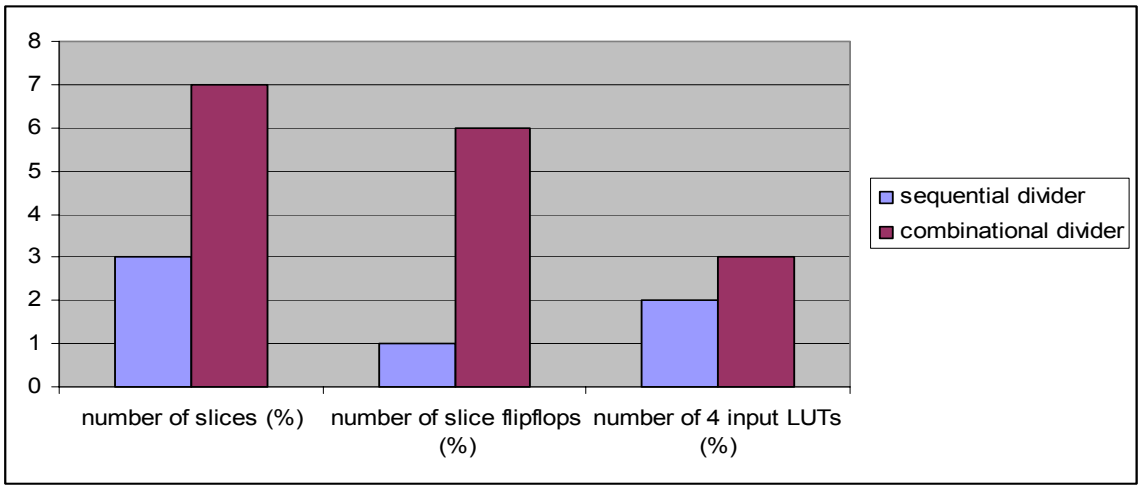


Figure 4-10: Comparison of sequential and Xilinx IP core dividers in terms of usage of device resources.

VHDL Post Implementation Simulation of Divider:

The shift and subtract sequential divider shown in Figure 4-9 was developed in VHDL using Xilinx 6.2.3i CAD tool set. The VHDL design was implemented to a Spartan 2E XC2S200 FPGA. The design was tested for functionality using Modelsim 5.7g tool. The main objective of the testing was to validate the design for different combinations of input operands. The design was tested and validated for different combinations of inputs. Figure 4-11 shows an instance of the VHDL post implementation simulation of the shift and subtract sequential divider.

The divisor input that is loaded to the Register_B is "000000.0011101110". This fixed point binary number represents the value:

$000000.0011101110 = 1*2^{-3}+1*2^{-4}+1*2^{-5}+1*2^{-7}+1*2^{-8}+1*2^{-9}=0.2324$

The dividend input that is loaded to the register AQ of the divider is "000001.0000001100". This fixed point binary number represents the value:

$000001.0000001100=1*2^{0}+1*2^{-7}+1*2^{-8}=1.0117$

Therefore, the result of the division should be 1.0117/0.2324=4.35.

From the post implementation simulation it can be seen that the result of the division is "000100.0101101001". The value of the number in fixed point arithmetic is:

$000100.0101101001=1*2^{2}+1*2^{-2}+1*2^{-4}+1*2^{-5}+1*2^{-7}+1*2^{-10}=4.35$

This result matches the required value and therefore the functionality of the shift and subtract sequential divider is validated.

Thus, the shift and subtract sequential divider is tested and validated.

Post implementation simulation of divider:

The post implementation simulation of divider is shown in Figure 4-11. The dividend value “0000010000001100” represents 1.01171, whereas the divisor value “0000000011101110” represents 0.2324. The result of the operation “0001000101101001” represents 4.35.

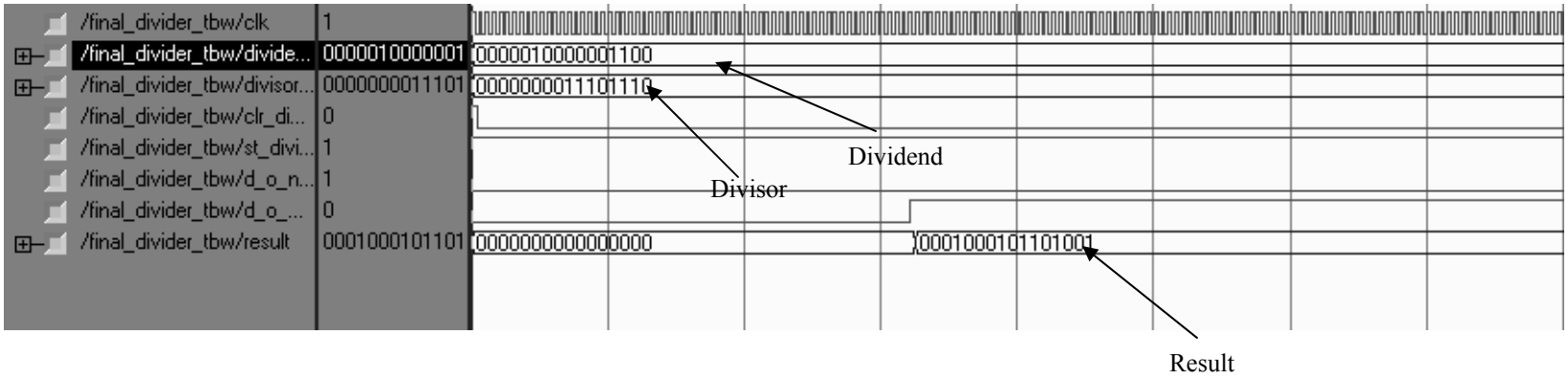


Figure 4-11: VHDL post implementation simulation of divider.

4.8.4 Design of EMC Processor Parallel Arithmetic Unit:

A high level functional schematic of the EMC processor parallel arithmetic unit is shown in Figure 4-12. The ripple-carry adder, sequential Booth 2 multiplier, and sequential shift and subtract divider are instantiated along with many registers to form the parallel arithmetic unit. The arithmetic unit gets its 14-bit inputs SO_T, and SO_{RH} from the interface module. It performs the operations shown in equations (4-8) and (4-9).

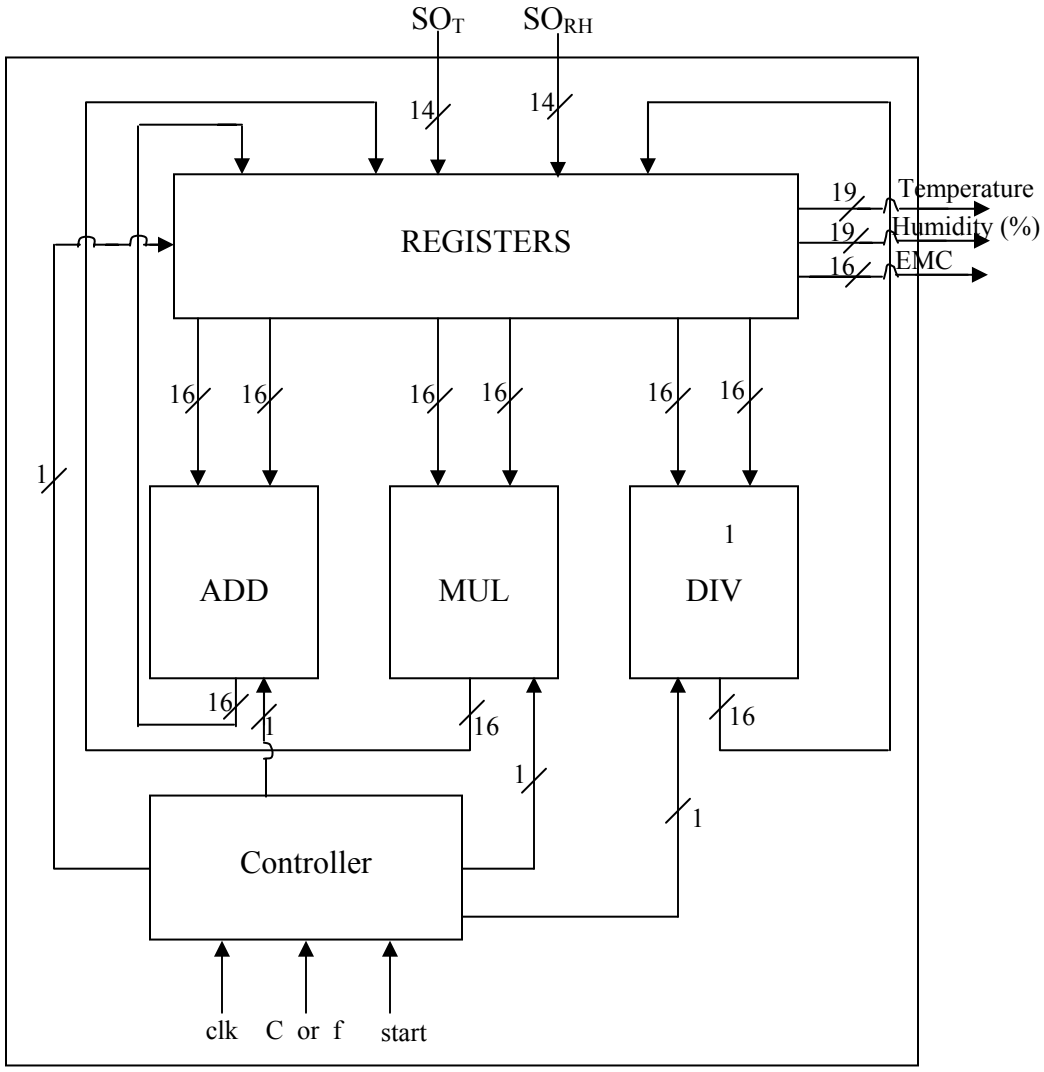


Figure 4-12: High level functional diagram of the arithmetic unit.

The outputs of the EMC processor parallel arithmetic unit are temperature, relative humidity (%), and EMC. The parallel arithmetic unit generates temperature both in Fahrenheit and Celsius. The user input 'C_or_f' given by the user selects the desired temperature output either in Celsius or Fahrenheit. These outputs are given to the display module. The controller is designed so that all the arithmetic modules are used simultaneously if possible.

The register level view of the EMC processor parallel arithmetic unit is shown in Figure 4-13. As it can be seen in Figure 4-13, the parallel computer arithmetic unit gets the 14-bit inputs SOT and SORH from the Interface Module of the EMC processor. The EMC processor parallel arithmetic unit starts its function when the Main Controller of the interface module sends a 'start' signal as an input to the Controller of the arithmetic unit. The entire parallel arithmetic unit works on the system clock provided as an input to the FPGA. When all the arithmetic operations are complete the Controller sends a 'done' signal to the Main Controller of the Interface Module. Upon receiving the 'done' signal, the Main Controller of the Interface Module sends 'arith_ok' signal to the Controller of the parallel arithmetic unit. This indicates the completion of the operation. The EMC processor parallel arithmetic unit uses various registers, multiplexors, a ripple-carry adder/subtractor, a Booth 2 sequential multiplier and a shift-subtract sequential divider. The functionality of the EMC processor parallel arithmetic unit is explained through the description of the individual components in this section.

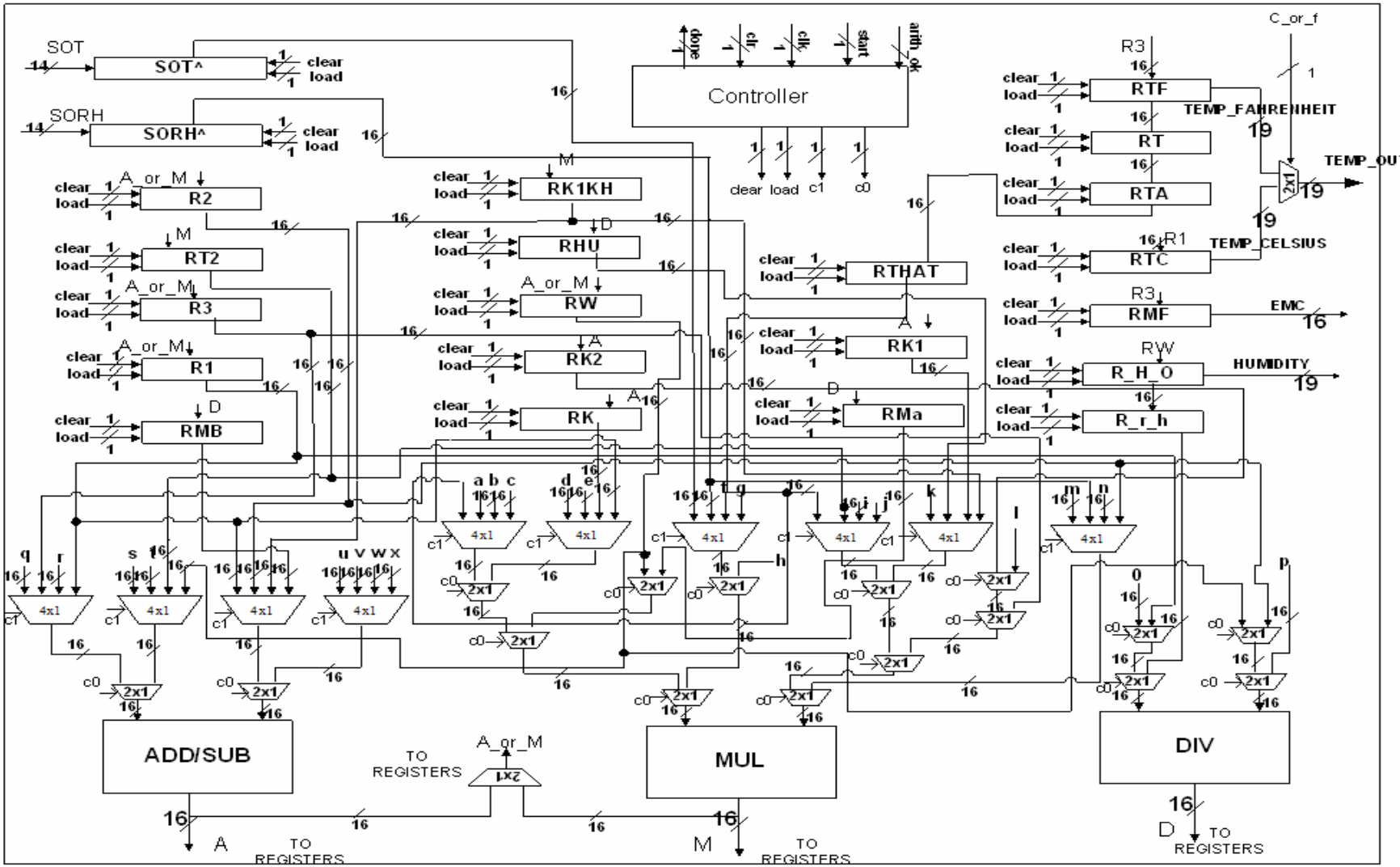


Figure 4-13: Register level view of the EMC processor parallel arithmetic unit.

- Registers: Various registers are used in the EMC processor parallel arithmetic unit to hold the values of the intermediate operands during the calculation of Equation sets (4-8) and (4-9). Register SOT^ and Register SORH^ are used to hold the values of the 14-bit inputs SOT and SORH. When a control signal is given these two registers produce the 16-bit outputs SOT^ (SOT/1024) and SORH^ (SORH/1024), those can be used as inputs to calculate the scaled Equations (4-9). Registers R1, R2 and R3 are used to hold temporary values during the operation. The inputs to these registers can be either from the ripple-carry adder or from the Booth 2 sequential multiplier. Therefore a 2X1 multiplexor is used at the outputs of the adder and the multiplier to choose the required input to be given to these registers.

The functionality of the rest of the registers used in the arithmetic unit is shown below.

RTC: This register is used to hold the value of temperature in Celsius. The 19-bit output of this register is 'TEMP_CELSIUS'.

RTF: This register is used to hold the value of temperature in Fahrenheit. The 19-bit output of this register is 'TEMP_FAHRENHEIT'.

A 19-bit 2 X 1 multiplexor is used at the output of the registers RTC and RTF. The user input 'C_or_f' is used as the control signal for this multiplexor. If 'C_or_f' is zero then temperature in Celsius is given as input to the display module. Otherwise, temperature in Fahrenheit is given as input to the display module.

RT: This register is used to hold the value of temperature in Fahrenheit. The output of this register is given as input to the register RTA.

RTA: This register gets its input from register RT. When given a control signal it logically right shifts its value and produces the value (T/1024), where T is the temperature in Fahrenheit. The output of this register is given as input to the register RT^.

RT^: This register is used to hold the value of T' (=T/1024) where T is the temperature in Fahrenheit. The output of this register is given as input to the Booth 2 sequential multiplier through the 4X1 multiplexors shown in Figure 4-13.

R_H_O: This register is used to hold the value of relative humidity in %. The output of this register is given as input to the display module and register R_r_h.

R_r_h: This register gets its input of relative humidity (%RH) from register R_H_O. Upon given a control signal this register performs logical right shift operation and generates its output. This output is given as input to the divider through the 2 X 1 multiplexor shown in Figure 4-13. The output of this register is divided by a value 6.25 to produce the required form of relative humidity that can be used to calculate the Equations (4-8).

RHU: The output of the divider is given as input to this register. It gets its input from the operation of R_r_h/6.25. This division produces the value of relative humidity (%RH/100), that can be used in further calculations. The output of this register is given as an input to the multiplier through the 4X1 multiplexors as shown in Figure 4-13.

RT2: This register is used to hold the value of the product RT^ X RT^. This value is T'^2, which is used many times in the calculation of Equation (4-8). The output of this register is given as inputs to the adder and the multiplier as shown in Figure 4-13.

RW: This register is used to hold the value of $W' = 1 + 1.4336T' + 1.6483(8 * T'^2)$. The value of *W'* has to be used later also in the calculation of other terms of Equation 4-8. Therefore, the output of the register RW is given as inputs to the adder, the multiplier and the divider through the multiplexors as shown in Figure 4-13.

RK: This register is used to hold the value of $K' = 1 + 0.5993T' - 1.1188T'^2$. This value of *K'* has to be used later also in the calculation of other terms in Equation set (4-8).Therefore, the output of the register RK is given as input to the multiplier as shown in Figure 4-13.

RK1: This register is used to hold the value of $K_1' = 1 + 0.1252T' - 1.933(8 * T'^2)$. The output of this register is given as input to the multiplier.

RK2: This register is used to hold the value of $K_2' = 1 + 1.6675(16 * T') - 1.3859(64 * T'^2)$. The output of this register also is given as one of the inputs to the multiplier through the multiplexors as shown in Figure 4-13.

RK1KH: This register is used to hold the value of $K_1'K'h$ of Equation set (4-8). The output of this register is connected to the inputs to the multiplier and the adder through the multiplexors as shown in Figure 4-13.

RMa: This register is used to hold the value of $\frac{K'h}{1.2642 - K'h}$ in the Equation set (4-8). The output of this register is provided as an input to the multiplier.

RMB: This register is used to hold the value of $\frac{K_1'K'h + 1.7244K_1'K_2'K'^2h^2}{0.1994 + K_1'K'h + 0.8622K_1'K_2'K'^2h^2}$. It gets its input from the output of the divider, and the output of register RMB is given as input to the adder.

RMF: This Register is used to hold the value of *EMC'*.

$$EMC' = \frac{5.4545}{W'}\left[\frac{K'h}{1.2642 - K'h} + \frac{K_1'K'h + 1.7244K_1'K_2'K'^2h^2}{0.1994 + K_1'K'h + 0.8622K_1'K_2'K'^2h^2}\right]$$

. The 16-bit output of this register is given as input to the display module.

The inputs to all the registers come from any of the adder/subtractor, the multiplier, the divider or other registers. The outputs of the registers are used again as inputs to the arithmetic units to perform the required calculations of Equation sets (4-8) and (4-9).

- Multiplexors: The EMC processor parallel arithmetic unit uses many 4 X 1 multiplexors and many 2 X 1 multiplexors at the inputs of the adder/subtractor, the multiplier and the divider. The control signals ('select' signals) to these multiplexors are given by the Controller of the EMC processor parallel arithmetic unit. These 'select' signals are not shown in Figure 4-13. The individual arithmetic units (the adder/subtractor, the multiplier and the divider) get their inputs from various registers. The input to an arithmetic unit at a particular instance is decided by the operation to be performed at that particular instance. Thus the Controller uses these multiplexors to choose the required operands to be given as the inputs to the arithmetic units.
- Adder/Subtractor: A ripple-carry adder/subtractor discussed in section 4.8.1 is used as the adder/subtractor for EMC processor parallel arithmetic unit. When an addition/subtraction operation is finished, it sends a 'done_add' signal (not

shown in Figure 4-13) to the Controller of the EMC processor parallel arithmetic unit to indicate that the addition/subtraction operation is completed. The output of the adder/subtractor is a 16-bit binary operand which is indicated as '**A**' in Figure 4-13.

- Multiplier: A Booth2 sequential multiplier discussed in section 4.8.2 is used as the multiplier for EMC processor parallel arithmetic unit. When a multiplication operation is finished, it sends a 'done_mul' signal (not shown in Figure 4-13) to the Controller of the EMC processor parallel arithmetic unit to indicate that the multiplication operation is completed. The output of the multiplier is a 16-bit binary operand which is indicated as '**M**' in Figure 4-13.
- Divider: A shift-subtract sequential divider discussed in section 4.8.3 is used as the divider for EMC processor parallel arithmetic unit. When a division operation is finished, it sends a 'done_div' signal (not shown in Figure 4-13) to the Controller of the EMC processor parallel arithmetic unit to indicate that the division operation is completed. The output of the divider is a 16-bit binary operand which is indicated as '**D**' in Figure 4-13.
- Constant internal signals: From Figure 4-13, it can be observed that the inputs to the multiplexors are not only the outputs of the registers but also some constant internal signals (from **'a'** to **'x'**) declared in VHDL. These constant internal signals are the values of the coefficients in Equations (4-8) and (4-9). The decimal values of these constant internal binary operands are given below.

  'a' = 1.6483; 'b' = 1.1188; 'c' = 1.933; 'd' = 0.1252; 'e' = 1.3859;
  'f' = 0.09175; 'g'=1.296; 'h' = 0.08192; 'i' = 1.4336; 'j' =0.5993;
  'k' = 1.6675; 'l' = 0.8622; 'm' = 1.8; 'n' = 0.16; 'o' = 5.4545; 'p' = 6.25;
  'q' = '1'; 'r' = 0.1994; 's' = 1.2642; 't' = 0.125; 'u' = 0.390625; 'v' = 0.01;
  'w' = 0.5; 'x' = 0.625;
- Controller: The Controller was designed using one-hot state encoding [19]. The flow chart of the one-hot state controller used in the EMC processor parallel arithmetic unit is shown in Figure 4-14. The Controller sends all control signals to the registers ('clear' and 'load'), all the control signals to the

multiplexors (c1 and c0) and the control signals to the individual arithmetic units.

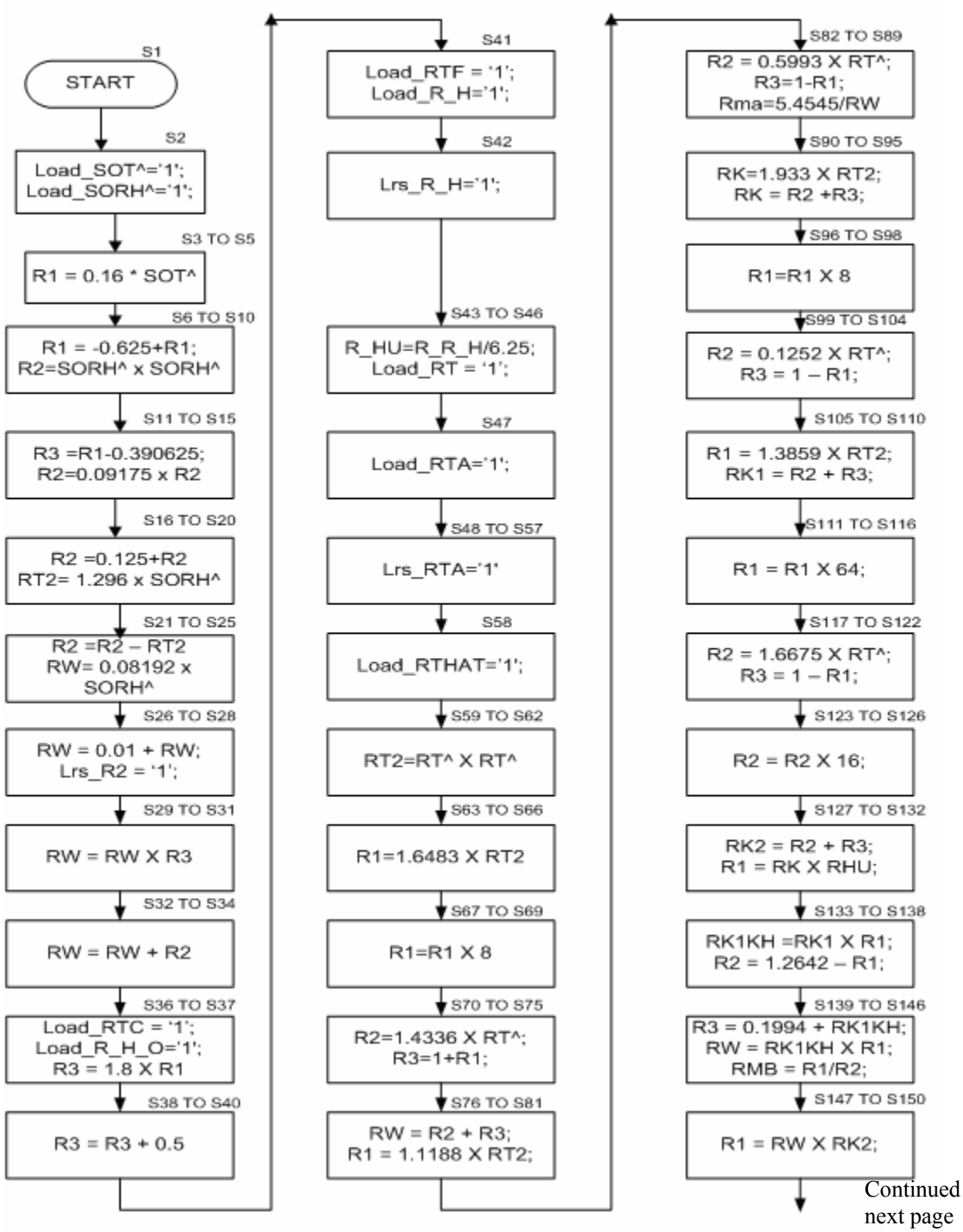


Figure 4-14: Flow chart of one-hot state controller used in the EMC processor parallel arithmetic unit.

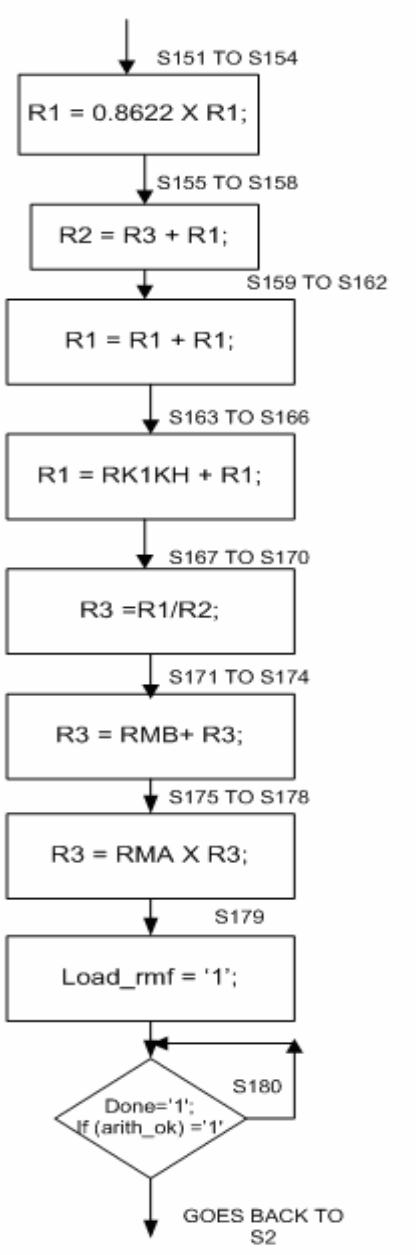


Figure 4-14: Flow chart of one-hot state controller used in the EMC processor parallel arithmetic unit.

The flowchart also shows the number of clock cycles for each operation. These clock cycle states are indicated by S1, S2… S180. When a 'start' signal is given by the Main Controller of the Interface Module, the Controller of the arithmetic unit starts its

operation at state S1 as shown in Figure 4-14. The sequence of operations is shown in Figure 4-14. As it can be observed, the individual arithmetic units (the adder/subtractor, the multiplier and the divider) are utilized in parallel to perform the calculations when possible. For example consider the states S139 to S146 of Figure 4-14. The clock cycle by clock cycle flow chart for these states is shown in Figure 4-15.

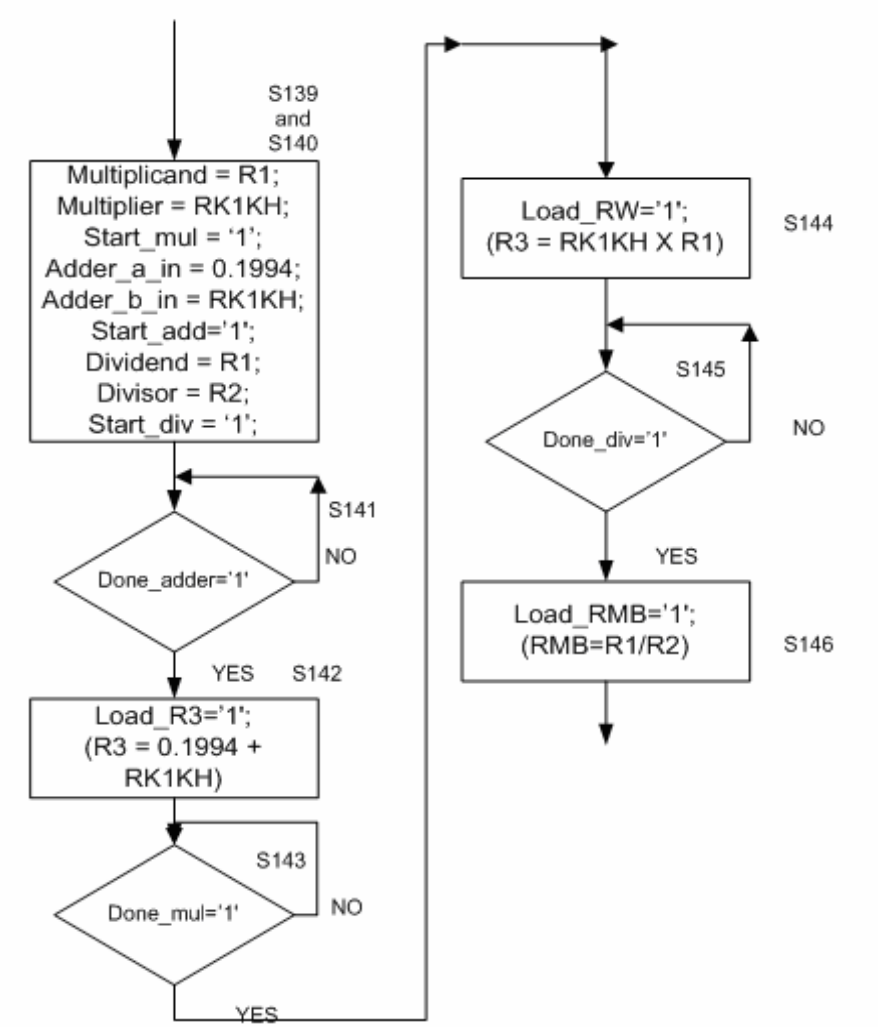


Figure 4-15: Clock cycle by clock cycle flow chart of the EMC processor parallel arithmetic unit one-hot state controller for states S139 to S146.

As shown in Figure 4-15, the inputs to the individual arithmetic unit are declared in states S139 and S140. All the arithmetic units are given the start signals ('start_add'

for the adder/subtractor, ‘start_mul’ for the multiplier and ‘start_div’ for the divider). Therefore, all the individual arithmetic units start performing the calculations simultaneously. As explained in previous sections the adder/subtractor takes least number of clock cycles to perform the operation and the division operation takes the maximum number of clock cycles. Therefore, the Controller checks for the ‘done_add’ signal from the adder/subtractor. When ‘done-add’ is high the Controller loads the register R3 and checks for the ‘done_mul’ signal as the multiplication takes less number of clock cycles than the division. When ‘done_mul’ is high the Controller loads the register RW and waits for the ‘done_div’ signal. When the division operation is also finished the Controller loads the register RMB and proceeds to state S147.

Thus, the three arithmetic units are used in parallel whenever possible. As explained above, from states S139 to S146, the addition and the multiplication operations are also performed in parallel in the time taken by the division operation. This reduces the time taken for the calculation of the EMC compared to that of the serial arithmetic unit in which the operations are performed serially.

The Controller follows the sequence of operations shown in Figure 4-14 and the value of the EMC is generated. This value is stored in register RMF. The outputs of the registers RTF, RTC, R_H_O and RMF are given as inputs to the display module of the EMC processor. Once the EMC value is computed, the Controller sends a ‘done’ signal to the Main Controller of the Interface Module and waits for ‘arith_ok’ signal. If the signal ‘arith_ok’ is high then the Controller goes back to state S2 and the operation is repeated.

The following section describes the VHDL post implementation simulation of the EMC processor parallel arithmetic unit.

4.8.5 VHDL Post Implementation Simulation of EMC Processor Parallel Arithmetic Unit:

The EMC processor parallel arithmetic unit shown in Figure 4-13 was developed in VHDL using Xilinx 6.2.3i CAD tool set. The VHDL design was implemented to a Xilinx Spartan2E XC2S200 FPGA. The design was tested for functionality using ModelSim 5.7g tool. The main objective of the testing was to validate the design for different combinations of input operands. Figures 4-16, 4-17 and 4-18 show an instance of the VHDL post implementation simulation of the EMC processor parallel arithmetic unit. When the Main Controller of the Interface Module issues a 'start' signal, the Controller starts by loading the 14-bit inputs SO_T and SO_{RH}.

Figure 4-16 shows the inputs SO_T and SO_{RH} given to the EMC processor parallel arithmetic unit.

The value of SO_T is "01110111011100" = $1*2^2+1*2^3+1*2^4+1*2^6+1*2^7+1*2^8+1*2^{10}+1*2^{11}+1*2^{12} = 7644$.

The value of SORH is "00011101100000" = $1*2^5+1*2^6+1*2^8+1*2^9+1*2^{10} = 1888$.

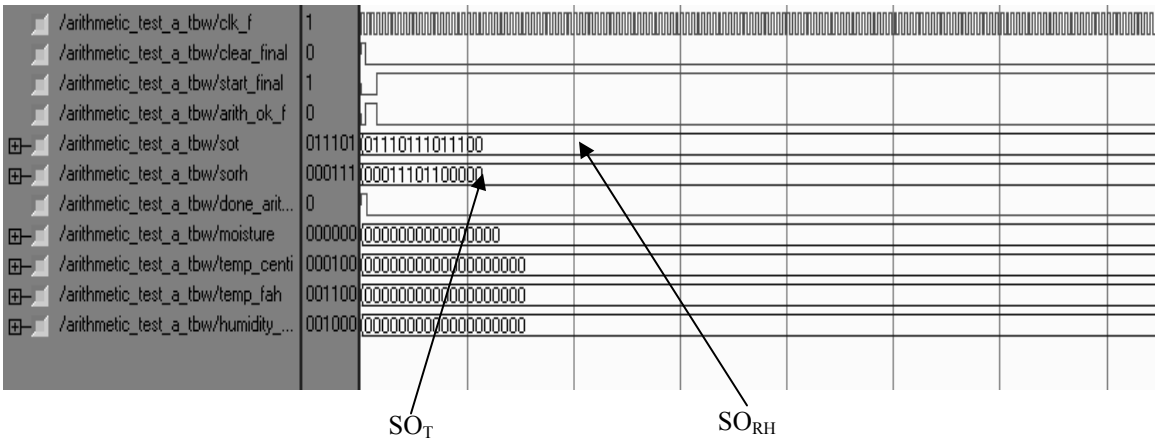


Figure 4-16: VHDL post implementation simulation of the EMC processor parallel arithmetic unit showing the inputs SOT and SORH.

For these values of SO_T and SO_{RH}, using Equation (4-9), the value of temperature in Celsius is approximately 36°C. The value of temperature in Fahrenheit is 96.7°F. The value of humidity is 64.3%RH. Figure 4-17 shows the values of temperature and relative humidity obtained in VHDL post implementation simulation for this example.

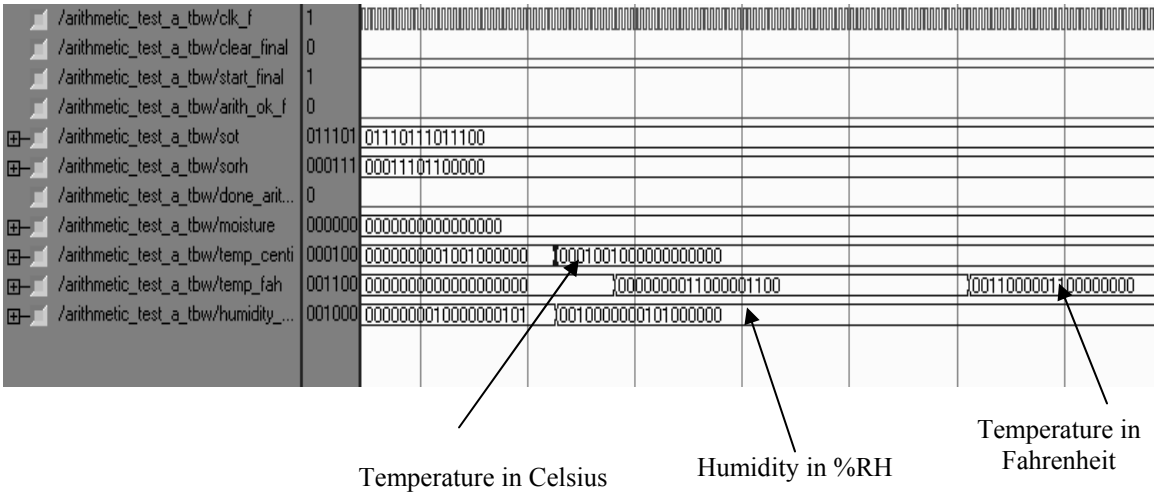


Figure 4-17: VHDL post implementation simulation of the EMC processor parallel arithmetic unit showing the values of temperature and relative humidity.

The value obtained for temperature in Celsius is "000100100.00000000000" which in fixed point arithmetic is equal to 36. The value obtained for temperature in Fahrenheit is "001100000.1100000000" which in fixed point arithmetic is equal to 96.75. The value obtained for the relative humidity is "001000000.0101000000"=64.3. Thus, the values obtained for temperature in Celsius, temperature in Fahrenheit and the relative humidity (%RH) match the theoretically computed values.

Figure 4-18 shows the value of the EMC obtained in the VHDL post implementation of the EMC processor parallel arithmetic unit. For a temperature of 96.7°F and a relative humidity of 64.3%RH the EMC value should be 11.2%.

From Figure 4-18, it can be observed that the final value of EMC is "0010110011110010", the value of which in fixed point format chosen is 11.2. Thus the EMC processor parallel arithmetic unit is tested and validated.

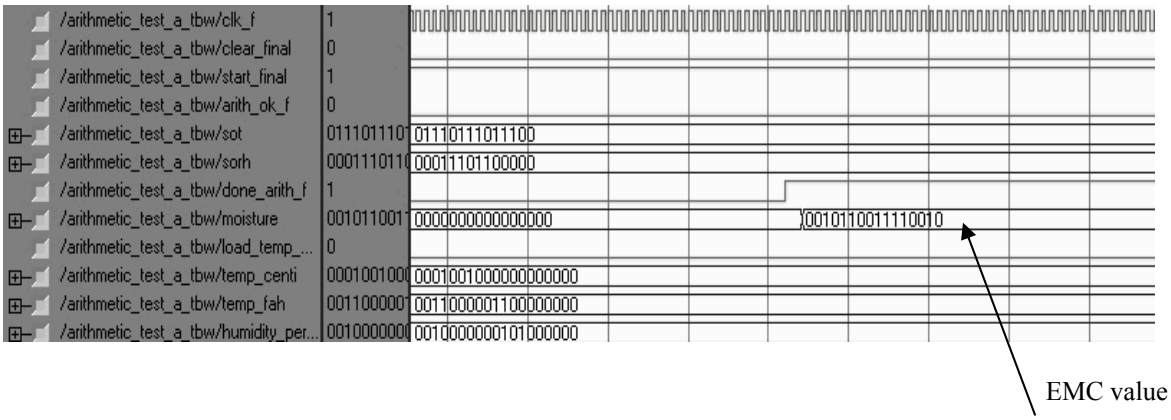


Figure 4-18: VHDL post implementation simulation of the EMC processor parallel arithmetic unit showing the value of the EMC.

EMC values for various values of temperature and relative humidity are shown in Table 4-1 as determined by post implementation VHDL simulation of the designed EMC processor parallel arithmetic unit. These values indicate that the parallel arithmetic unit as designed, computes proper values of EMC at different temperatures and relative humidity.

Values of temperature in Fahrenheit	Values of relative humidity								
	5%	15%	25%	35%	45%	55%	65%	75%	85%
30	1.4	3.7	5.5	7.1	8.7	10.4	12.4	15	18.6
50	1.4	3.6	5.5	7.1	8.7	10.3	12.3	15	18.4
70	1.3	3.5	5.4	6.9	8.5	10	12	14	17.9
90	1.2	3.4	5.2	6.7	8.1	9.7	11.5	14	17.4
110	1.1	3.2	4.9	6.3	7.7	9.2	10.9	13	16.6
130	1	2.9	4.5	5.9	7.2	8.6	10.3	13	15.8
150	0.9	2.6	4.1	5.5	6.7	8.1	9.7	12	14.9
170	0.7	2.3	3.7	4.9	6.1	7.4	9	11	14
190	0.6	1.9	3.2	4.4	5.5	6.7	8.2	10	12.9
210	0.5	1.5	2.7	3.7	4.8	6	7.4	9.2	11.9

Table 4-1: VHDL post implementation simulation values of EMC processor parallel arithmetic unit for various values of temperature and relative humidity.

4.9 EMC Processor Serial Arithmetic Unit Design:

This section explains in detail the design of the serial arithmetic unit system used in the EMC processor. The EMC processor serial arithmetic unit combines the functionalities of the adder/subtractor, the multiplier, and the divider that are designed to implement the parallel arithmetic unit, into one arithmetic unit. All the arithmetic operations are performed with an adder/subtractor and some set of registers. The following sections detail the design of the EMC processor serial arithmetic unit, its hardware implementation, and its VHDL post implementation simulations using the Xilinx ISE 6.2.3i/Modelsim 5.7g.

4.9.1 Design of the Serial Arithmetic Unit:

Figure 4-19 shows the high level functional diagram of the EMC processor serial arithmetic unit. The serial arithmetic unit uses a ripple-carry adder/subtractor discussed in section 4.8.1 to perform the addition/subtraction operations. Register EF is a general purpose shift register that can perform logical left shift and arithmetic shift functions. A one hot state controller is designed to control all the hardware units in the design. The controller performs the addition/subtraction or multiplication or division depending on the inputs ‘start_add’, ‘start_mul’ and ‘start_div’. The serial arithmetic unit performs ripple-carry addition/subtraction, Booth 2 multiplication and shift-subtract sequential division. The steps followed in addition/subtraction, multiplication and division are described below.

- Addition/Subtraction:
    1. In the first step, register A is loaded with input ‘adder_a_in’. Register B is loaded with input ‘adder_b_in’. The one hot state controller sends control signal to the multiplexor ‘mux_2x1_b’ so that the input ‘adder_b_in’ is chosen.
    2. In the second step, the outputs of register A and register B are given as inputs to the ripple carry adder/subtractor. The controller sends control signal to the multiplexor ‘mux_2x1_c’ so that the output from the register A is given as input to the adder/subtractor.

3. The one hot state controller sends control signals ‘sel_bin’ and ‘sel_cin’ to the adder/subtractor depending on the operation required. If the input ‘add_sub’ to the controller is low then the addition is performed. If ‘add_sub’ is high then the subtraction operation is performed. The result of the operation is stored in register C. The output of register C is taken out as ‘adder_out’.

- Multiplication:
    1. Load the register EF, register B, and counter. The register EF is a 32-bit register that can perform logical left shift and arithmetic right shift operations. In the first step, register EF is loaded with the value of the multiplier in the bit locations 15 to 0 of the register EF. The one hot state controller gives control signal to the multiplexor ‘mux_2x1_e’ so that the multiplicand is selected. The register ‘B’ is loaded with the value of 16-bit multiplicand. The counter is loaded with a value of “01111” for 16 X 16 binary multiplications. The flip-flop CO is loaded with value ‘0’.
    2. Check the last bit of register EF and output of the flip-flop CO. If the value of the two bits is “10”, then the first 1 in a string of 1’s has been encountered in the multiplicand. This requires a subtraction of the multiplicand from the partial product stored in the register EF (Bit locations 31 to 16). In this case the multiplicand is subtracted from register E (Bit locations 31 to 16 of register EF). If the two bits are “01”, the first 0 in a string of 0’s has been encountered in the multiplicand. This requires the addition of the multiplicand to the partial product in register E of register EF. When the two bits are “00” or “11” no action is necessary and so the next shift occurs.
    3. Arithmetic right shift the register EF. The arithmetic right shift ensures that the most significant bit of the register EF before the shift is duplicated into the most significant bit of the register EF after the shift. This step ensures that there is no sign change in the result. The least

significant bit of the register EF is given as input to the flip-flop CO. Decrement the counter by '1'.

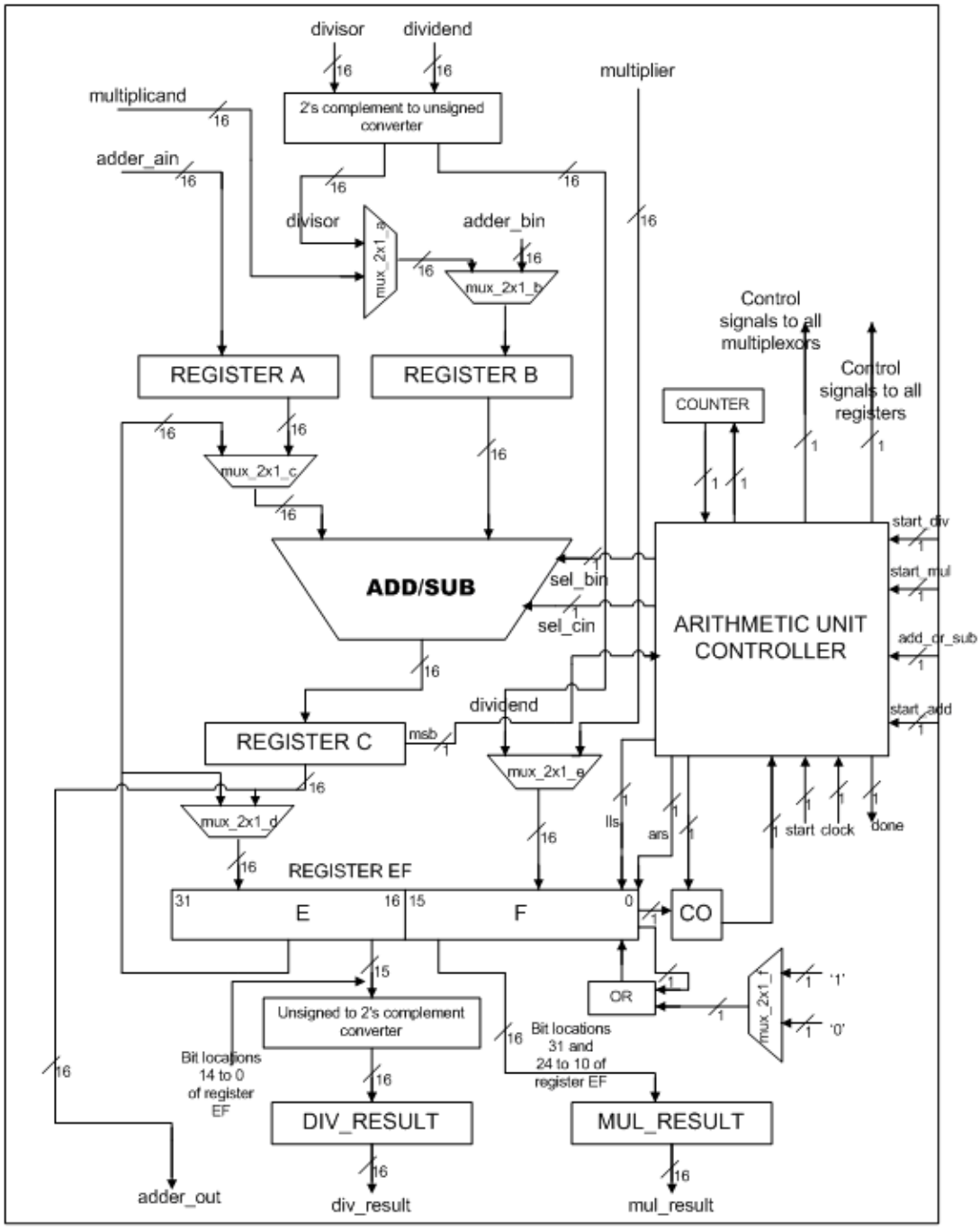


Figure 4-19: High level functional diagram of the serial arithmetic unit.

4. If the value of the counter is not ‘0’ repeat the steps from step 2. Otherwise, store the result in the register ‘MUL_RESULT’. The register ‘MUL_RESULT’ is loaded with bit locations 31 and 24 to 10 of register EF. This step is performed for the same reasons that were explained in the design of the Booth 2 sequential multiplier of the EMC processor parallel arithmetic unit.

- Division:
    1. In the first step, the dividend and the divisor input operands are converted from two’s complement representation to unsigned representation. The unsigned outputs of dividend and divisor are 16-bit wide.
    2. The divisor is stored in register B. The 16-bit dividend is loaded into the 32-bit Register EF. The dividend is loaded to bit locations 15 to 0 of the Register EF. The counter is loaded with a value of “01111” as the divisor and the dividend operands are each 16-bit wide.
    3. The register EF is logically left shifted 10 times so that the dividend operand is stored in locations 25 to 10 of the register EF. This operation is performed for the same reasons that were explained in the design of shift and subtract sequential divider of the EMC processor parallel arithmetic unit. The one-hot state controller performs this operation by issuing ‘lls’ signal high for 10 clock cycles.
    4. Contents of register E, and register B are fed to the adder/subtractor. The result of the subtraction is stored in register C. The most significant bit ‘msb’ of the result is checked in the one-hot type controller. If ‘msb’ is zero, then the result is overflow. Otherwise, go to step 5.
    5. The register EF is logically left shifted once, and the counter is decremented by one.
    6. The value of the counter is checked by the controller. If the value is zero, then go to step 7. Otherwise, go to step 8.

7. The subtraction operation is done between register E and register B, and the ‘msb’ bit is checked again. If the contents of register E are greater than that of register B, then the result in register C is stored in register E and the least significant bit of register EF is made ‘1’. The result is stored in register EF. The quotient value is stored in register F and the reminder in E.
8. A subtraction operation is done between register E and register B. If the contents in register E are greater than that of register B, then the result in register C is stored in register A. And the least significant bit of register EF is made ‘1’. The process is repeated from step 5 again.
9. The quotient of the result stored in bit locations 14 to 0 of register EF is converted to two’s complement form and the result is stored in the register ‘DIV_RESULT’.

Thus, the addition/subtraction, multiplication and division operations are performed using only one Ripple Carry adder/subtractor and a set of registers. The hardware required to perform the operations is reduced compared to that of the EMC processor parallel arithmetic unit. But the serial arithmetic unit is not capable of computing independent terms in the Equation sets (4-8) and (4-9) in parallel. All the operations are performed one at a time serially and hence the name serial arithmetic unit.

VHDL Post Implementation Simulation of the Serial Arithmetic Unit:

The serial arithmetic unit shown in Figure 4-19 was developed in VHDL using Xilinx 6.2.3i CAD tool set. The VHDL design was implemented to a Xilinx Spartan2E XC2S200 FPGA. The design was tested for functionality using ModelSim 5.7g tool. The main objective of the testing was to validate the design for different combinations of input operands and input signals. Figure 4-20 shows an instance of the VHDL post implementation simulation of the arithmetic unit used in the EMC processor serial arithmetic unit for subtraction and addition operations.

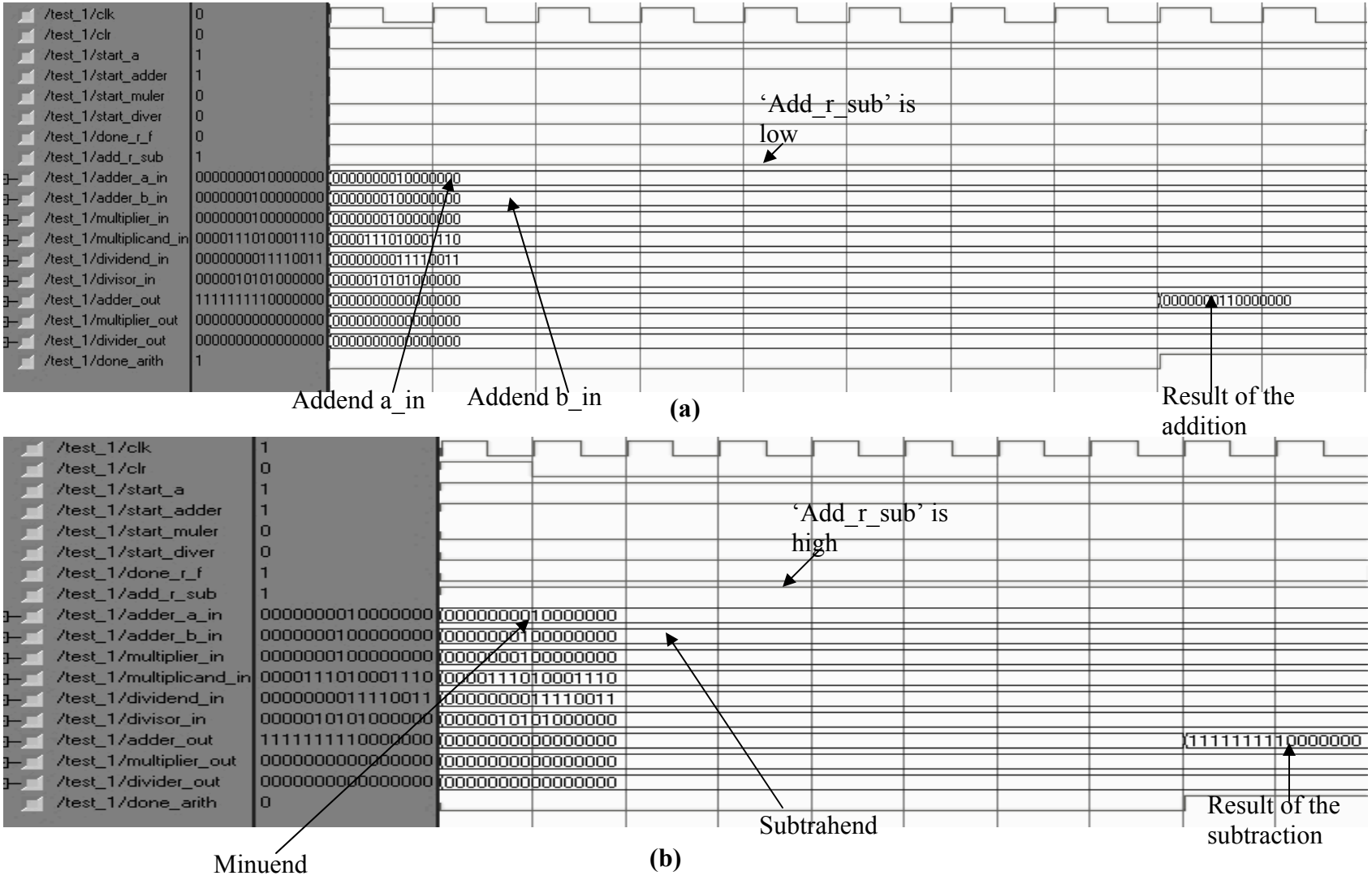


Figure 4-20: VHDL Post implementation simulation of the serial arithmetic unit for (a) addition and (b) subtraction operations.

The control signal ‘start_adder’ is given high to perform the addition/subtraction operation. When the input ‘add_r_sub’ is low the serial arithmetic unit performs the addition operation. The VHDL post implementation simulation for addition operation can be seen in Figure 4-20 (a).
The input ‘adder_a_in’ is “000000.0010000000”, which represents the value 2^{-3}=0.125 in fixed point real arithmetic. The input ‘adder_b_in’ is “000000.0100000000”, which represents the value 2^{-2}=0.25 in fixed point arithmetic. The result of the addition operation should be 0.125 + 0.25= 0.375. The post implementation waveforms in Figure 4-20 (a) show that the result of the addition is “000000.0110000000”, which represents the value $2^{-2}+2^{-3}$=0.375. Thus, the addition operation is validated.

When the input ‘add_r_sub’ is high the serial arithmetic unit performs the subtraction operation. The VHDL post implementation simulation for subtraction operation can be seen in Figure 4-20 (b).

From Figure 4-20 (b), the input ‘adder_a_in’ is “000000.0010000000”, which represents the value 2^{-3}=0.125 in fixed point real arithmetic. The input ‘adder_b_in’ is “000000.0100000000”, which represents the value 2^{-2}=0.25 in fixed point arithmetic. The result of the subtraction operation should be 0.125 - 0.25= -0.125. The post implementation waveforms in Figure 4-20 (a) show that the result of the subtraction is “111111.1110000000”, which represents the value – (000000.0010000000) = - 0.125. Thus, the subtraction operation is validated.

When the control signal ‘start_adder’ is zero and the control signal ‘start_mul’ is high the arithmetic unit performs the Booth 2 multiplication operation. The serial arithmetic unit is tested and validated for different combinations of inputs for the multiplication. Figure 4-21 shows an instance of the VHDL post implementation simulation waveform of the multiplication operation.

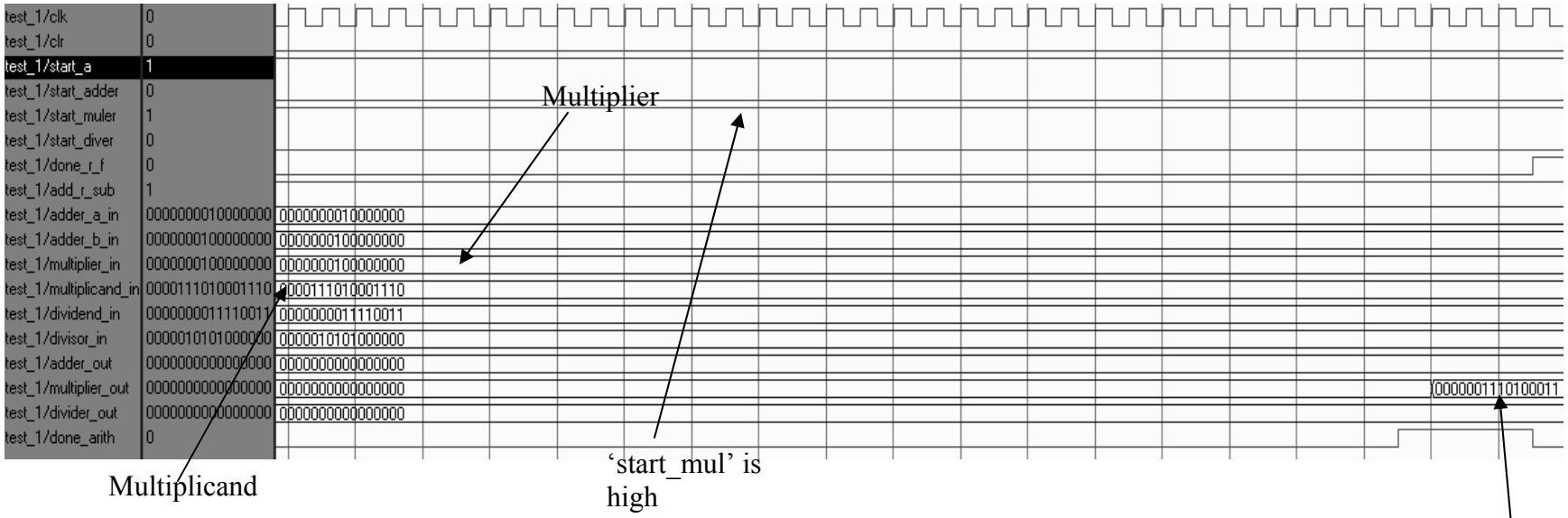


Figure 4-21: VHDL Post implementation simulation of the serial arithmetic unit for the multiplication operation.

When the control input 'start_mul' is high and 'start_add' is low, the serial arithmetic unit performs the multiplication operation. The multiplicand input given is "0000111010001110" and the multiplier input given is "0000000100000000". The result obtained is "0000001110100011".

The multiplicand input that is loaded to the register B is “000011.1010001110”. This fixed point binary number represents the value:

$000011.1010001110 = 1*2^{1}+1*2^{0}+1*2^{-1}+1*2^{-3}+1*2^{-7}+1*2^{-8}+1*2^{-9}=3.6386$

The multiplier input that is loaded to the register EF of the serial arithmetic unit is “000000.0100000000”. This fixed point binary number represents the value:

$000000.0100000000=1*2^{-2}=0.25$

Therefore, the result of the multiplication should be 3.6386*0.25=0.909

From the post implementation simulation it can be seen that the result of the multiplication is “000000.1110100011”. The value of the number in fixed point arithmetic is:

$000000.1110100011=1*2^{-1}+1*2^{-2}+1*2^{-3}+1*2^{-5}+1*2^{-9}+1*2^{-10}=0.909$

This result matches the required value and hence proves the functionality of the serial arithmetic unit multiplication.

Thus, the Booth 2 multiplication operation in the serial arithmetic unit is tested and validated.

When the control signal inputs ‘start_add’ and ‘start_mul’ are zero and the control signal input ‘start_div’ is high the serial arithmetic unit performs sequential shift and subtract operation. The serial arithmetic unit is tested and validated for different combinations of inputs for the division operation. Figure 4-22 shows an instance of the VHDL post implementation simulation waveform of the division operation.

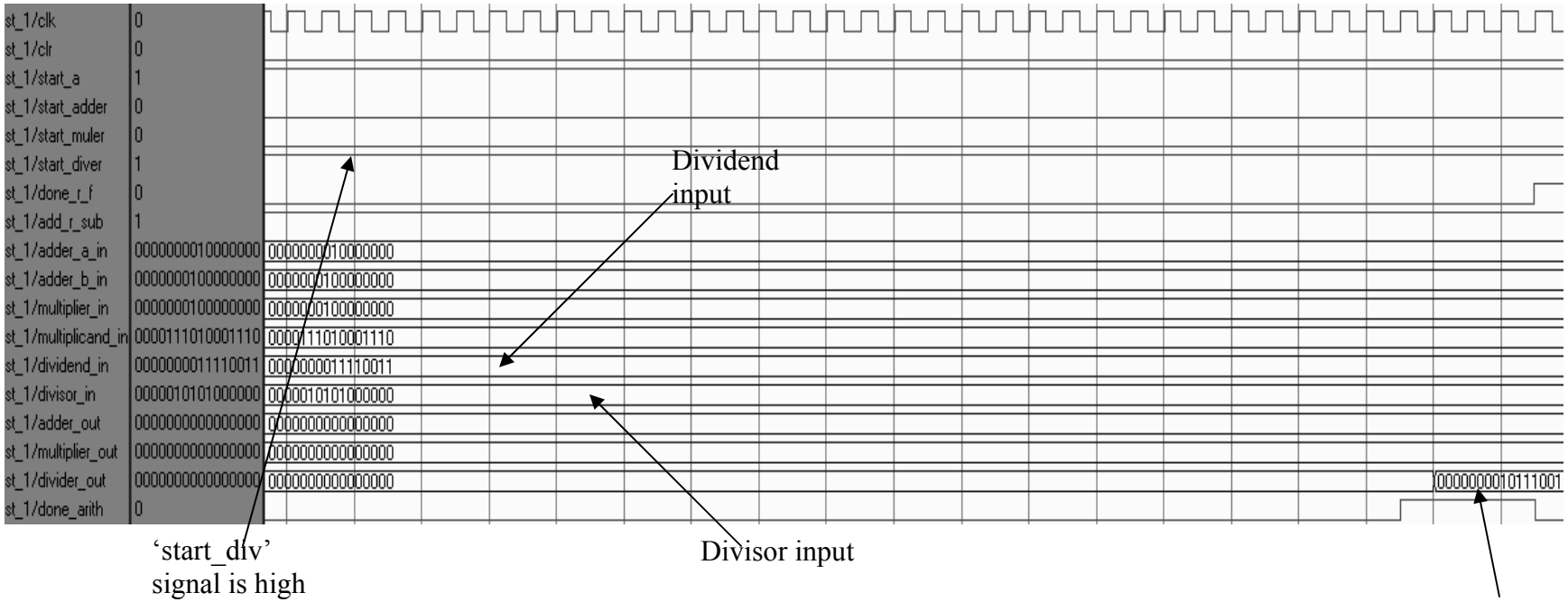


Figure 4-22: VHDL Post implementation simulation of the serial arithmetic unit for the division operation.

When the control input 'start_div' is high and 'start_add' and 'start_mul' are low, the serial arithmetic unit performs the division operation. The dividend input given is "0000000011110011" and the divisor input given is "0000010101000000". The result obtained is "0000000010111001".

The divisor input that is loaded to the register B is "000001.0101000000". This fixed point binary number represents the value:

$000001.0101000000 = 1*2^{0}+1*2^{-2}+1*2^{-4}=1.3125$

The dividend input that is loaded to the register EF of the divider is "000000.0011110011". This fixed point binary number represents the value:

$000000.0011110011=1*2^{-3}+1*2^{-4}+1*2^{-5}+1*2^{-6}+1*2^{-9}+1*2^{-10}=0.2373$

Therefore, the result of the division should be 0.2373/1.3125=0.18.

From the post implementation simulation it can be seen that the result of the division is "000000.0010111001". The value of the number in fixed point arithmetic is:

$000000.0010111001=1*2^{-3}+1*2^{-5}+1*2^{-6}+1*2^{-7}+1*2^{-10}=0.18$

This result matches the required value and hence proves the functionality of the shift-subtract sequential division of the serial arithmetic unit.

Thus, the serial arithmetic unit with one ripple-carry adder/subtractor and a set of registers is used to perform all the ripple-carry addition/subtraction operation, Booth 2 sequential multiplication and shift-subtract sequential division. The hardware of the arithmetic unit is reduced compared to that of the parallel arithmetic unit. The design of the entire EMC processor serial arithmetic unit is described in the following section.

4.9.2 Design of EMC Processor Serial Arithmetic Unit:

The register level view of the EMC processor serial arithmetic unit is shown in Figure 4-23. As it can be seen in Figure 4-23, the EMC processor serial computer arithmetic unit gets the 14-bit inputs SOT and SORH from the Interface Module of the EMC processor. It starts its function when the Main Controller of the interface module sends a 'start' signal as an input to the Controller of the arithmetic unit. The entire serial arithmetic unit works on the system clock provided as an input to the FPGA. When all the arithmetic operations are complete the Controller sends a 'done' signal to the Main Controller of the Interface Module. Upon receiving the 'done' signal, the Main Controller of the Interface Module sends 'arith_ok' signal to the Controller of the serial arithmetic unit. This indicates the completion of the operation.

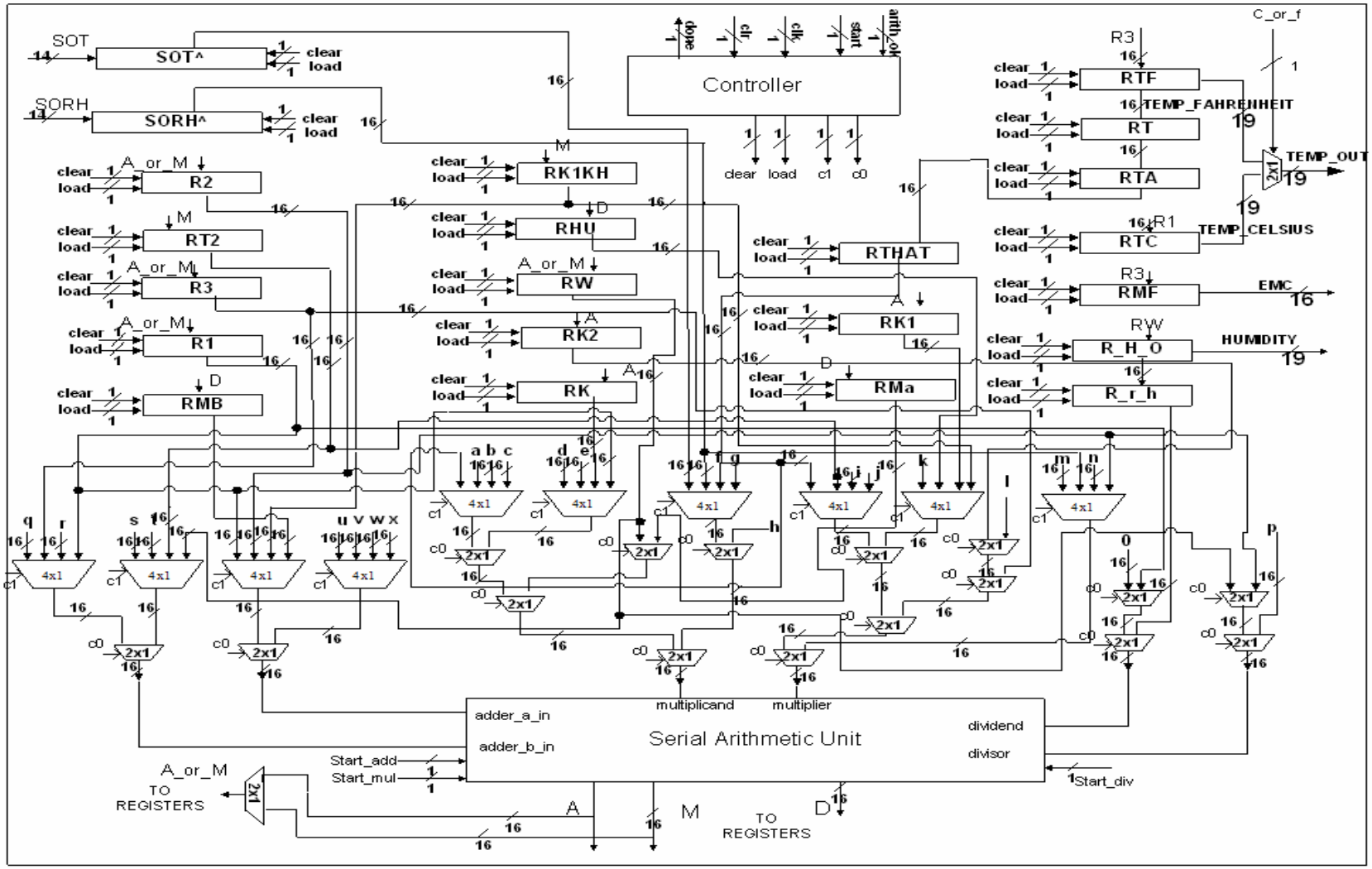
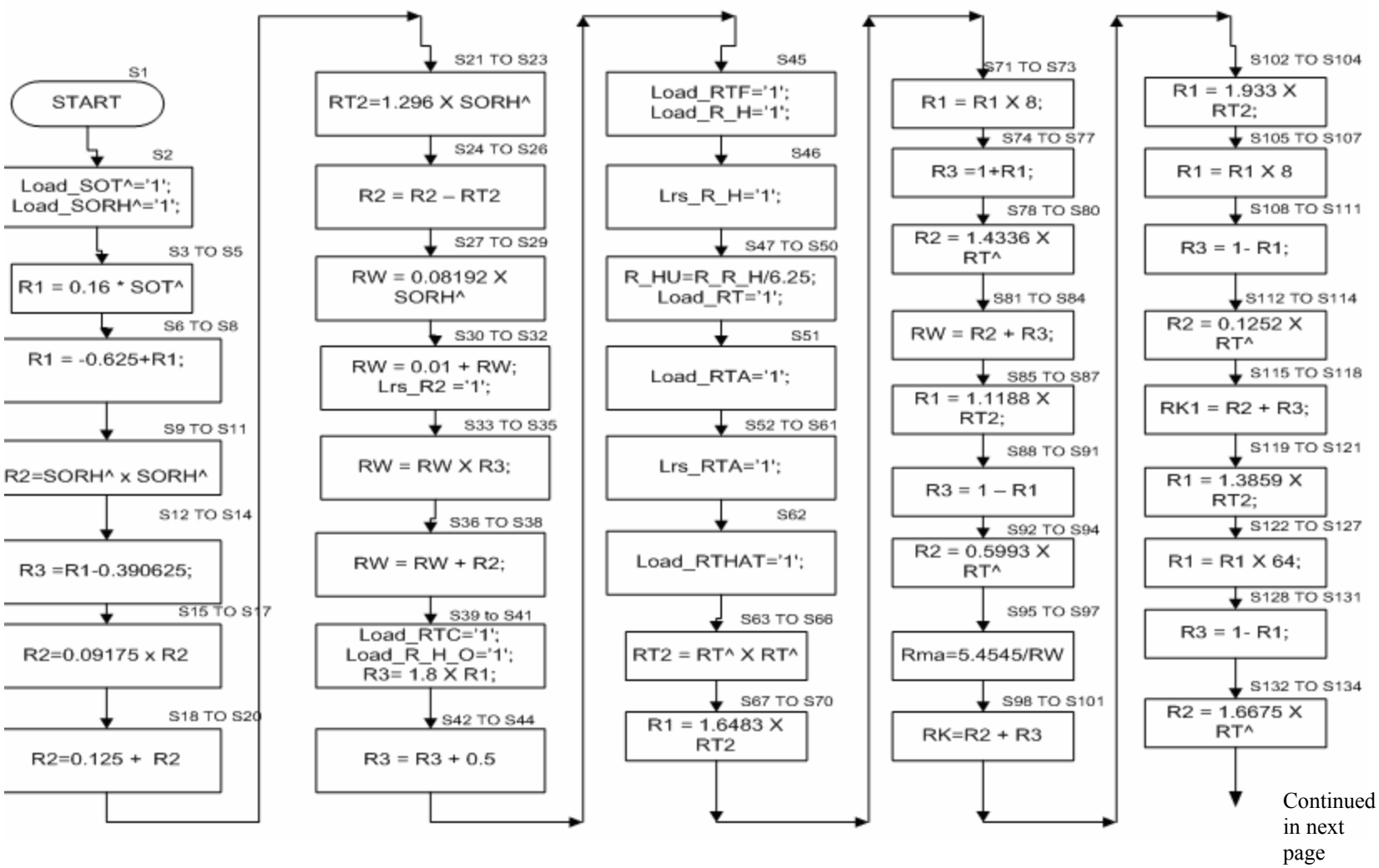


Figure 4-23: Register level view of the EMC processor serial arithmetic unit.

The EMC processor serial arithmetic uses various registers, multiplexors in addition to the serial arithmetic unit to compute the value of EMC. The functionality of all the registers and all the multiplexors and the values of all the constant internal signals (‘**a’** to ‘**x’**) are the same as those explained in section 4.8.4. Therefore, the functionality of the individual units is not repeated in this section. The main difference between the EMC processor parallel arithmetic unit and the EMC processor serial arithmetic unit is, the individual arithmetic units (the adder/subtractor, the multiplier and the divider) are not present in the EMC processor serial arithmetic unit. Instead, it uses the serial arithmetic unit to perform the calculations. The Controller of the EMC processor serial arithmetic unit sends input signals ‘start_add’, ‘start_mul’ and ‘start_div’ to perform the addition/subtraction, the multiplication and the division respectively. The serial arithmetic gets its inputs ‘adder_a_in’, ‘adder_b_in’, ‘multiplicand’, ‘multiplier’, ‘dividend’ and ‘divisor’ from the multiplexors. These outputs from the multiplexors are the same as those provided to the individual arithmetic units in the EMC processor parallel arithmetic unit. The serial arithmetic unit generates three 16-bit outputs A, M and D. A is the result of the addition/subtraction, M is the result of the multiplication and D is the result of the division. These outputs are provided as inputs to the registers in the same way that was done in the EMC processor parallel arithmetic unit.

The one-hot state Controller used in EMC processor serial arithmetic unit is slightly different from the Controller used in EMC processor parallel arithmetic unit. The flow chart of the Controller used in the serial arithmetic unit is shown in Figure 4-24. The Controller starts its operation when it is given a ‘start’ signal from the Main Controller of the Interface Module. The sequence of operation and also the clock states can be observed from Figure 4-24. As it can be observed, all the arithmetic calculations are computed serially unlike the parallel arithmetic unit. Once the value of EMC is computed, the Controller sends a ‘done_arith’ signal to the Main Controller of the Interface Module and waits for ‘arith_ok’ signal. When ‘arith_ok’ signal goes high, the control goes back to the state S2 and the sequence is repeated.



Continued in next page

Figure 4-24: Flow chart of the one-hot state controller used in the EMC processor serial arithmetic unit.

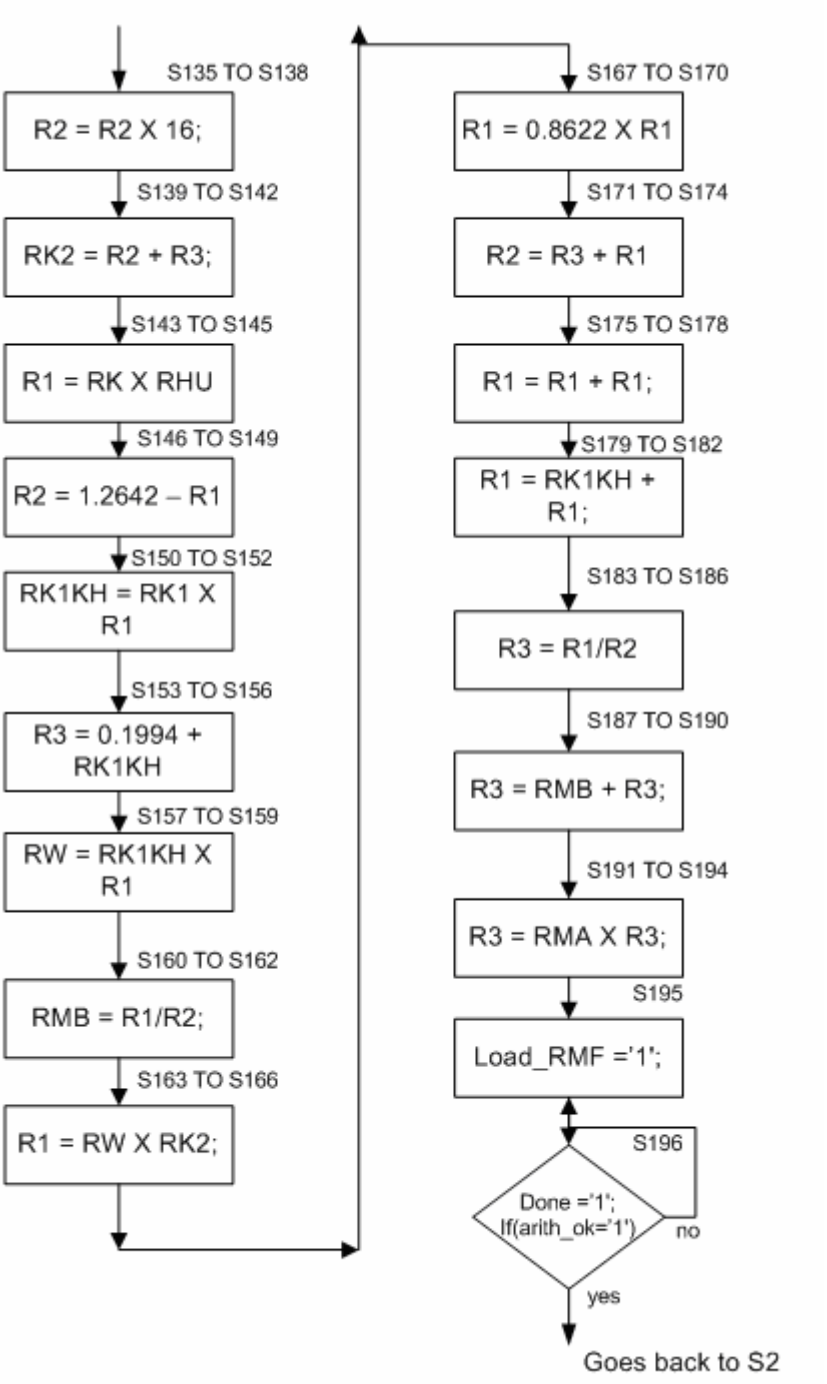


Figure 4-24: Flow chart of the one-hot state controller used in the EMC processor serial arithmetic unit.

It can be observed from Figure 4-24, that the number of states in the Controller used in the serial arithmetic unit is 196, whereas the number of states in the Controller used in the parallel arithmetic unit is 180. This increase in number of clock cycle states is due to the serial execution of the arithmetic operations. The following section describes the VHDL post implementation simulation of the EMC processor serial arithmetic unit.

4.9.3 VHDL Post Implementation Simulation of EMC Processor Serial Arithmetic Unit:

The EMC processor parallel arithmetic unit shown in Figure 4-23 was developed in VHDL using Xilinx 6.2.3i CAD tool set. The VHDL design was implemented to a Xilinx Spartan2E XC2S200 FPGA. The design was tested for functionality using Modelsim 5.7g tool. The main objective of the testing was to validate the design for different combinations of input operands. Figures 4-25, 4-26 and 4-27 show an instance of the VHDL post implementation simulation of the EMC processor serial arithmetic unit. When the Main Controller of the Interface Module issues a 'start' signal, the Controller starts by loading the 14-bit inputs SO_T and $SO_{RH.}$

Figure 4-25 shows the inputs SO_T and SO_{RH} given to the EMC processor parallel arithmetic unit.

The value of SO_T is "01110111001110" = $1*2^1+1*2^2+1*2^3+1*2^6+1*2^7+1*2^8+1*2^{10}+1*2^{11}+1*2^{12} = 7630$.

The value of SORH is "00011100001000" = $1*2^3+1*2^8+1*2^9+1*2^{10} = 1800$.

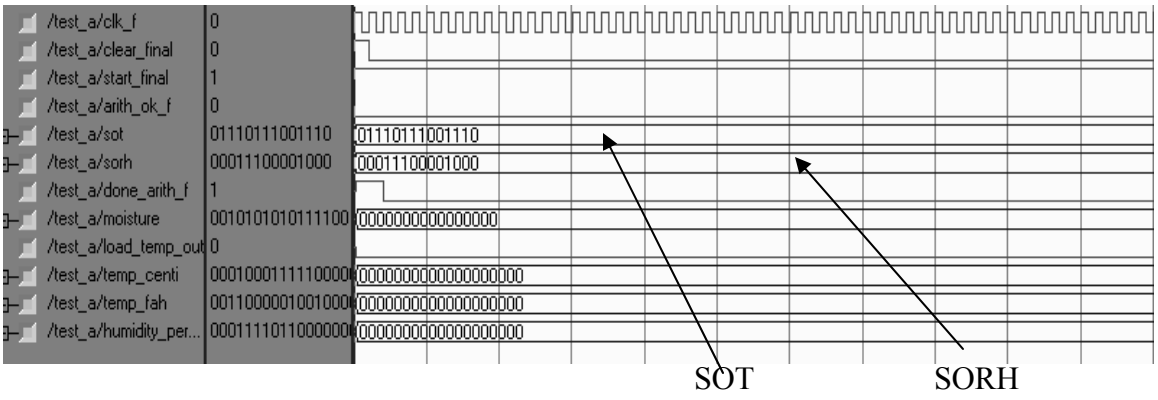


Figure 4-25: VHDL post implementation simulation of the EMC processor serial arithmetic unit showing SOT and SORH inputs.

For these values of SO_T and SO_{RH}, using Equation (4-9), the value of temperature in Celsius is approximately 35.83^{o}C. The value of temperature in Fahrenheit is 96.5^{o}F. The value of humidity is 61.5%RH. Figure 4-26 shows the values of temperature and relative humidity obtained in VHDL post implementation simulation for this example.

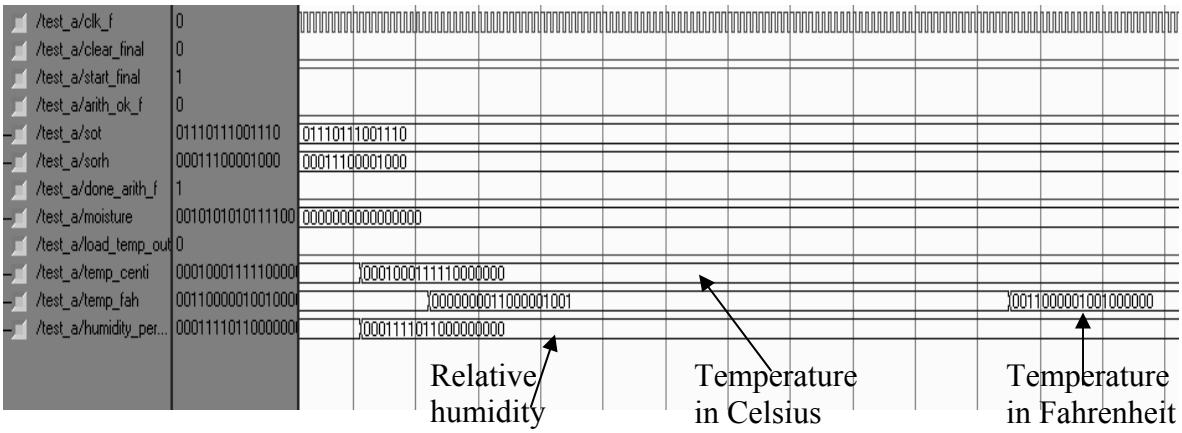


Figure 4-26: VHDL post implementation simulation of the EMC processor serial arithmetic unit showing the values of temperature and relative humidity.

The value obtained for temperature in Celsius is "000100011.1110000000" which in fixed point arithmetic equals to 35.87. The value obtained for temperature in Fahrenheit is "001100000.1001000000" which in fixed point arithmetic equals to 96.5. The value obtained for the relative humidity is "000111101.1000000000"=61.5. Thus, the values obtained for temperature in Celsius, temperature in Fahrenheit and the relative humidity (%RH) match the theoretically computed values.

Figure 4-27 shows the value of the EMC obtained in the VHDL post implementation of the EMC processor serial arithmetic unit. For a temperature of 96.5^{o}F and a relative humidity of 61.5%RH the EMC value should be 10.7%. From Figure 4-27, it can be observed that the final value of EMC is "001010.1010111100", the value of which in fixed point format chosen is 10.7. Thus the EMC processor serial arithmetic unit is tested and validated.

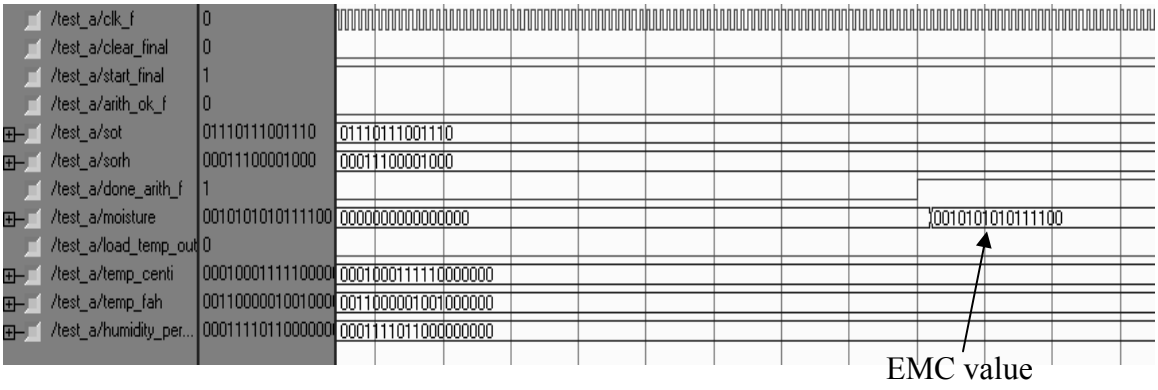


Figure 4-27: VHDL post implementation simulation of the EMC processor serial arithmetic unit showing the value of EMC.

VHDL post implementation simulation values of EMC of the EMC processor serial arithmetic unit for various values of temperature and relative humidity are shown in Table 4-2. These values indicate that the EMC processor serial arithmetic unit computes proper values of EMC at different temperatures and relative humidity.

Relative humidity

	5%	15%	25%	35%	45%	55%	65%	75%	85%
30	1.4	3.7	5.5	7.1	8.7	10.4	12.4	15	18.6
50	1.4	3.6	5.5	7.1	8.7	10.3	12.3	15	18.4
70	1.3	3.5	5.4	6.9	8.5	10	12	14	17.9
90	1.2	3.4	5.2	6.7	8.1	9.7	11.5	14	17.4
110	1.1	3.2	4.9	6.3	7.7	9.2	10.9	13	16.6
130	1	2.9	4.5	5.9	7.2	8.6	10.3	13	15.8
150	0.9	2.6	4.1	5.5	6.7	8.1	9.7	12	14.9
170	0.7	2.3	3.7	4.9	6.1	7.4	9	11	14
190	0.6	1.9	3.2	4.4	5.5	6.7	8.2	10	12.9
210	0.5	1.5	2.7	3.7	4.8	6	7.4	9.2	11.9

Temperature T in Fahrenheit

Table 4-2: VHDL post implementation simulation values of EMC processor serial arithmetic unit for various values of temperature and relative humidity.

4.10 Comparison of EMC Processor Parallel and Serial Arithmetic Systems:

A comparison of the EMC processor parallel arithmetic system and the EMC processor serial arithmetic system in terms of the resource utilization on Spartan2E xc2s200 FPGA and the number of clock cycles required to compute the EMC is presented in this section. As discussed earlier, the parallel arithmetic unit uses three individual arithmetic units (a ripple-carry adder/subtractor, a Booth2 sequential multiplier and a shift-subtract sequential divider) to compute the value of EMC. These three units are utilized in parallel whenever it is possible. Table 4-3 shows how different terms in Equations (4-8) and (4-9) are calculated in parallel by the utilization of three individual arithmetic units simultaneously in the EMC processor parallel arithmetic unit. The columns of the table indicate the hardware unit used and the rows indicate the operation performed. All the operations in a row are performed simultaneously. All these terms are calculated in serial in the EMC processor serial arithmetic unit as only one arithmetic unit is used to perform all the operations. Therefore, the hardware used in the serial arithmetic unit is reduced compared to that of the parallel arithmetic unit. The logic resource comparison of these two arithmetic units when implemented to a Xilinx Spartan2E xc2s200 FPGA is shown below. However, the difference in the total equivalent gate count obtained from the map reports using Xilinx 6.2.3i is not huge. This is due to the reason that the serial arithmetic unit uses similar hardware design used by the parallel arithmetic unit except for the additional adder/subtractors, counters and shift registers used in the design of the multiplier and the divider of the parallel arithmetic unit.

EMC processor serial arithmetic unit:
Number of Slices: 818 out of 2352 34%
Number of Slice Flip Flops: 827 out of 4704 17%
Number of 4 input LUTs: 1467 out of 4704 31%
Total equivalent gate count for design: 15,787

EMC processor parallel arithmetic unit:
Number of Slices: 827 out of 2352 35%
Number of Slice Flip Flops: 963 out of 4704 20%
Number of 4 input LUTs: 1497 out of 4704 31%
Total equivalent gate count for design: 16,533

No.	Multiplier	Adder/Subtractor	Divider
1	0.16*SOT'		
2	SORH'*SORH'	T'_{real}=-0.625+(0.16*SOT^)	
3	0.09175*SORH'^{2}	$T'_{real} - 0.390625$	
4	1.296*SO'_{RH}	0.125+(0.09175*SO'^2_{RH})	
5	0.08192*SO'_{RH}	RH'_{linear}=1.296*SO'_{RH} – (0.125+(0.09175*SO'^2_{RH}))	
6		0.01+(0.08192*SO'_{RH})	
7	(0.01+(0.08192*SO'_{RH}))* (T'_{real} – 0.390625)		
8	1.8 * T'_{real}	RH'_{true}=(0.01+(0.08192*SO'_{RH}))* (T'_{real} – 0.390625)+0.5*RH'_{linear}	
9		T in °F=1.8*T'_{real}+0.5	R_hu=R_r_h/6.25
10	$13.1866T'^2$		
11	$1.4336T'$	$1+13.1866T'^2$	
12	$1.1188T'^2$	$1+1.4336T'+13.1866T'^2$	
13	$0.5993T'$	$1-1.1188T'^2$	$M_A = \frac{5.4545}{W'}$
14	$15.4640T'^2$	$1+0.5993T'-1.1188T'^2$	
15	$0.1252T'$	$1-15.4640T'^2$	
16	$86.9645T'^2$	$1+0.1252T'-15.4640T'^2$	
17	$26.6804T'$	$1-86.9645T'^2$	
18	$K'h$	$1+26.6804T'-86.9645T'^2$	
19	$K_1'K'h$	$1.2642-K'h$	
20	$K_1'K_2'K'^2h^2$	$0.1994+K_1'K'h$	$M_B = \frac{K'h}{1.2642-K'h}$
21	$0.8622K_1'K_2'K'^2h^2$		
22		$0.1994+K_1'K'h+0.8622K_1'K_2'K'^2h^2$	
23		$0.8622K_1'K_2'K'^2h^2 + 0.8622K_1'K_2'K'^2h^2$	
24		$K_1'K'h+1.7244K_1'K_2'K'^2h^2$	
25			$M_C = \frac{K_1'K'h+1.7244K_1'K_2'K'^2h^2}{0.1994+K_1'K'h+0.8622K_1'K_2'K'^2h^2}$
26		$\frac{K_1'K'h+1.7244K_1'K_2'K'^2h^2}{0.1994+K_1'K'h+0.8622K_1'K_2'K'^2h^2} + \frac{K'h}{1.2642-K'h}$	
27	$EMC = M_A*(M_B+M_C)$		

Table 4-3: Sequence of operations in the EMC processor parallel arithmetic unit showing the parallel utilization of the arithmetic units.

The comparison of both arithmetic units in terms of logic resources used show that the serial arithmetic unit requires less hardware. But the difference in hardware between the arithmetic units is not huge. Therefore, if the EMC processor is intended to be developed on a FPGA the serial arithmetic unit doesn't have considerable advantage over the parallel arithmetic unit. However, if the EMC processor is intended to be developed on an ASIC, then the serial arithmetic unit offers more advantage as it uses less area on chip.

The total number of clock cycles taken for the computation of EMC in the serial arithmetic unit is 2808. The total number of clock cycles taken for the computation of EMC in the parallel arithmetic unit is 1942. Thus, there is a difference of around 866 clock cycles between the parallel arithmetic unit and serial arithmetic unit. The reason for this difference is not only due to the serial execution of the terms in Equations (4-8) and (4-9) but also the multiplication and division operations are slower in the arithmetic unit as discussed earlier.

One of the design constraints of the EMC processor indicates that the value of EMC should be computed approximately every one second. The interface module of the sensor takes around 500ms to generate the 14-bit outputs SOT and SORH. Therefore, in order to achieve the required time constraint, the minimum clock period required if the parallel arithmetic unit is used is approximately 0.25ms. The serial arithmetic unit requires a clock cycle of minimum time period of approximately 0.17ms to achieve the required time constraint. Thus, if the parallel arithmetic unit is used for the EMC processor, the clock frequency can be reduced more. The reduction of clock frequency reduces the dynamic power consumption of the EMC processor, which will be a big advantage for the final EMC processor device.

From the above comparisons, it can be concluded that if the EMC processor is intended to be developed on a FPGA then the parallel arithmetic unit developed for the EMC processor would be a better choice. However, if the EMC processor is intended to be developed on an ASIC, then an appropriate choice of the arithmetic unit can be made by the comparison of power consumption and area on chip between the two arithmetic units.

4.11 High Level Functional Schematic of the EMC Processor:

A high-level functional schematic of the entire EMC processor system is shown in Figure 4-28. The interface module developed for the sensor interface is instantiated with the arithmetic unit to form the entire EMC processor system. These instantiated modules are implemented to a Xilinx Spartan 2E FPGA. Thus, the FPGA gets its inputs 'clk', 'start', 'clr', 'C_or_f' inputs from outside. The input 'clk' serves as the system clock for the entire EMC processor system. The input 'start' is given to the EMC processor by user to start the computation of EMC. The input 'clr' is given by the user to clear the values of Main Controller and restart the operation. 'C_or_f' is an input given by the user to display the temperature either in Celsius or Fahrenheit.

When 'start' input is given to the Main Controller of the EMC processor, the system responds by generating the command signals to the SHT71 sensor. As discussed earlier, the Main Controller measures the temperature value first and stores the value in the register SOT shown in Figure 4-28. The same process repeats for the humidity measurement and the value is stored in register SORH. These digital readouts from the sensor are 14-bit outputs and are given as inputs to the arithmetic unit. Once the measurement of the temperature and humidity are complete, the Main Controller sends a signal 'Start_arith' to the Controller of the arithmetic unit. The Controller of the arithmetic unit responds to this input by starting its operations. At first the digital readouts from the sensor are compensated for non-linearity and are converted to the physical values. These values are used to calculate the EMC value from Equations 4-8. As discussed in previous sections, two different versions of the arithmetic unit are developed for the entire EMC processor system. Therefore either parallel arithmetic unit or serial arithmetic unit is used as the arithmetic unit for the EMC processor system. After the computation of EMC is finished, the Controller of the arithmetic units sends the outputs of temperature, humidity and EMC to the display module. The arithmetic unit generates temperature in both scales, Celsius and Fahrenheit. The input 'C_or_f' given by the user to the EMC processor declares the type of temperature output that should be sent as an input to the display module of the EMC processor as shown in Figure 4-28.

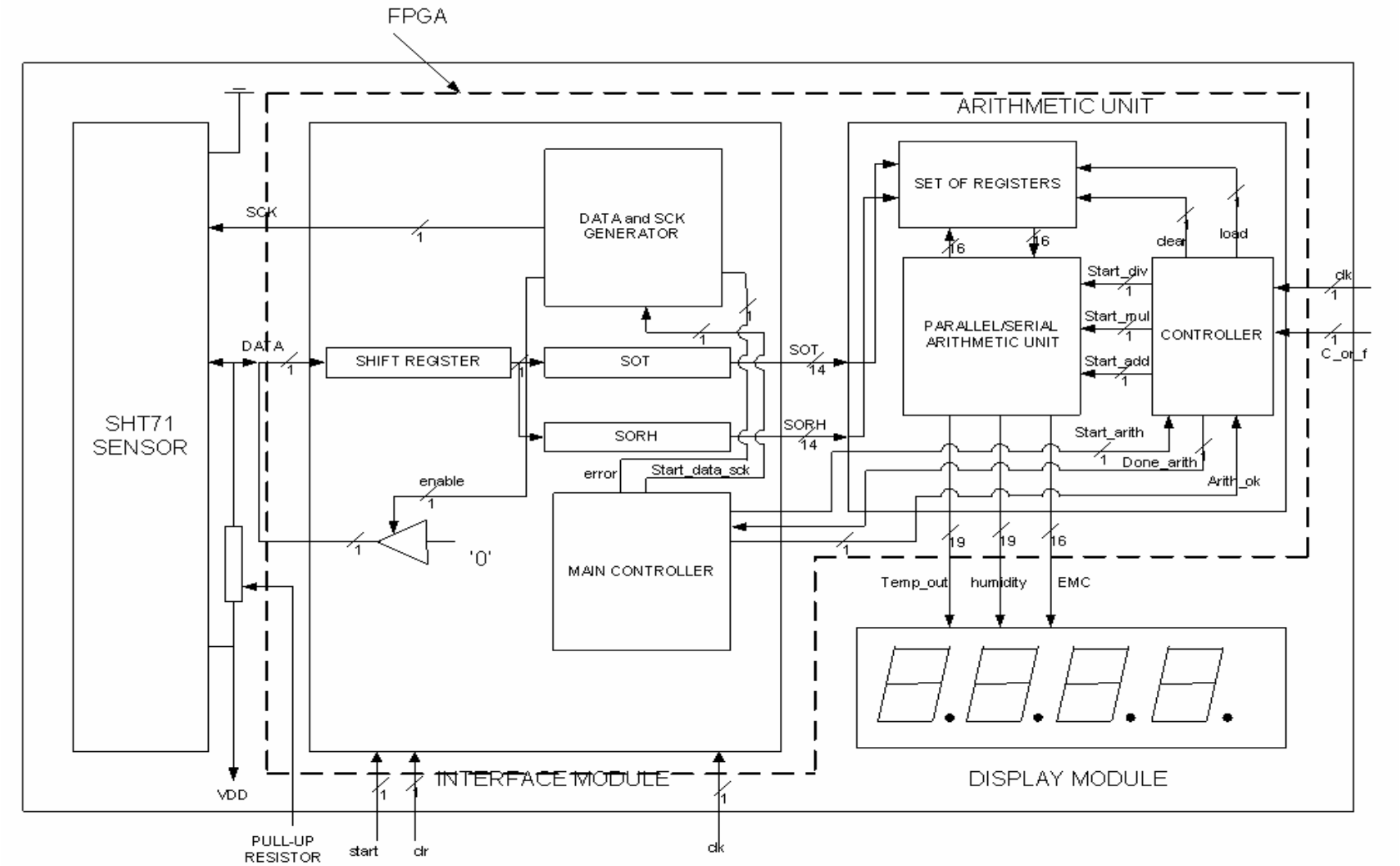


Figure 4-28: High-level functional schematic of the entire EMC processor.

The Controller of the arithmetic unit sends ‘done_arith’ signal to Main Controller to indicate that the computation is finished. The Main Controller responds to this by sending a hand-shake signal ‘arith_ok’ to the arithmetic unit and starts measuring the values of the temperature and humidity again. Thus the sequence is repeated.

Four seven segment multiplexed displays are used as the display device module in the hardware prototype developed for the EMC processor as shown in Figure 4-28. The display device module, as code in VHDL, takes the inputs of temperature, relative humidity, and EMC and produces the required anode and cathode signals for the display device.

Thus, two different EMC processors are developed, one with the parallel arithmetic unit and another with the serial arithmetic unit. These two processors are implemented to a Xilinx Spartan2E FPGA and tested for the functionality. The details of implementation and experimental hardware prototype testing are given in Chapter 5.

Chapter 5

Experimental EMC Processor Hardware Prototype Implementation and Testing

Two different versions of the EMC processor, one with the EMC processor parallel arithmetic unit, another with the EMC processor serial arithmetic unit are developed, synthesized and implemented to a Xilinx SPARTAN2E xc2s200 FPGA. Both versions of the EMC processor are tested in various conditions of atmosphere to test their functionality and performance. The basic FPGA design flow used in the EMC processor design is discussed in this chapter. The details of the implemented hardware prototype are given. Values of EMC obtained from the prototype of the EMC processor with the parallel arithmetic unit and from the prototype of the EMC processor with the serial arithmetic unit, over a range of temperature and relative humidity values are compared with given theoretical values of EMC and the functional performance of the EMC processor design is analyzed.

5.1 FPGA Design Flow:

A Xilinx Spartan2E XC2S200 Field Programmable Gate Array device (FPGA) is used to implement the EMC processor hardware prototype. The EMC processor is developed in VHDL, synthesized and implemented using Xilinx ISE 6.2.3i tool. The designed EMC processor is tested and validated using Modelsim 5.7g software. The basic design flow involved in the design of the processor using a Xilinx Spartan2E XC2S200 FPGA is explained in this section [20].

The steps involved in the design can be explained as follows:

- Design Capture: The design capture step is about entering the hardware design in high level Hardware Description Languages (HDL) like VHDL, Verilog. It also involves including any user constraint files required. The user constraint files (ucf) helps in specifying particular pin connections, and particular time delays required for the design. The EMC processor is developed in VHDL using the

Xilinx ISE 6.2.3i tool. The user constraint file developed for the EMC processor to implement to Spartan2E xc2s200 FPGA is shown in Appendix 2.

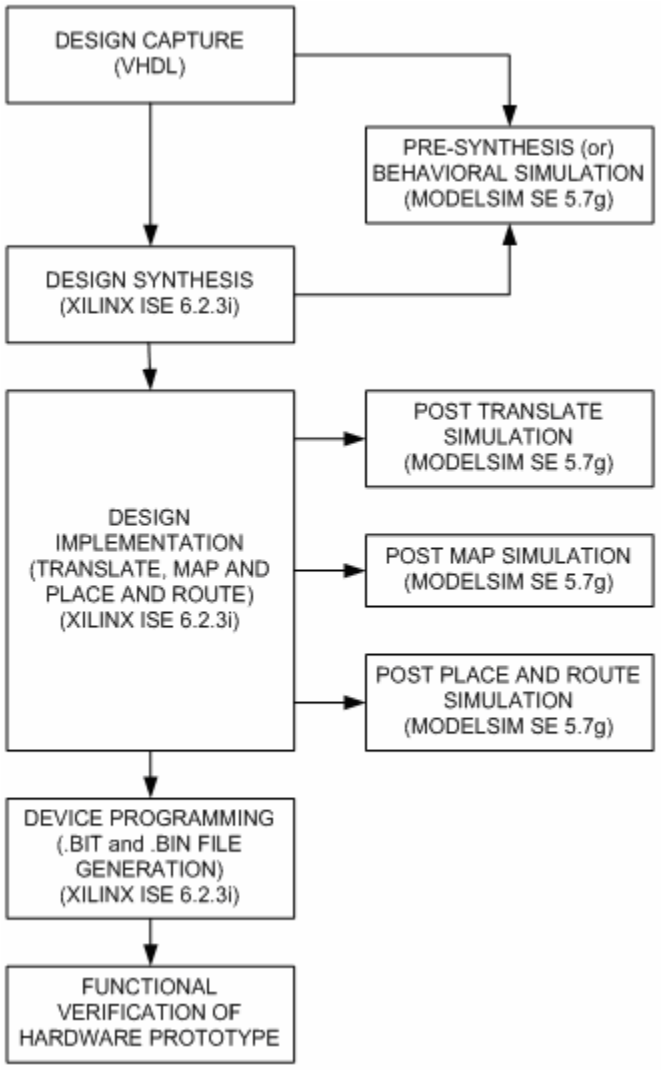


Figure 5-1: Basic FPGA design flow of Xilinx devices.

- Design Synthesis: Synthesis of a design entry in VHDL results in a digital circuit that implements the behavior implied by the VHDL description. The process of design synthesis basically operates on three types of information. The first is the VHDL entry of the design. The second is the implementation constraints to be

followed such as area and speed. The third is the set of components that are to be used to construct the design. Hierarchical designs, designs that involve several lower level components combined to form a functional circuit, are synthesized in a bottom up fashion. The lower level components are synthesized first followed by higher-level components. The VHDL descriptions of the design are transformed into EDIF (Electronic Data Interchange Format), an optimized physical realization, also called a gate level netlist.

- Pre-synthesis (or) Behavioral Simulation: The pre-synthesis (or) behavioral simulation of a design can be performed either prior to or after design synthesis. It is generally performed to verify the VHDL code syntax, and to confirm that the code is functioning as intended. Modelsim 5.7 g tool set is used to perform pre-synthesis simulation.
- Design Implementation: The steps involved in design implementation are Translate, Map, and Place and route. The translate process merges all of the input netlists (such as EDIF) and outputs a Xilinx NGD (Native Generic Database) file which can be mapped to the target device. The NGD file represents the logical design reduced to Xilinx primitives. The next step is to map the design which creates an NCD (Native Circuit Description) file. An NCD file represents the physical circuit description of the input design. The map process is followed by the place and route of the design. The place and route process uses the NCD file as its input to place and route the design. It also outputs another NCD file which can be used by the bit stream generator (to generate configuration files for FPGA) later.
- Post-translate simulation: The post-translate simulation is done before the map process and after the translate process in design implementation step. This simulation allows verifying whether the design is correctly synthesized or not. Modelsim 5.7 g tool set is used to perform post-translate simulation.
- Post-map simulation: The post map simulation shows the block delays for the design. But the routing delays are not included in this simulation. The block delays are the delays between the CLBs (Configurable Logic Blocks) and the delays between the IOBs (Input/Output Block) in the FPGA. This simulation

produces the results before the routing of the entire design is done. Modelsim 5.7 g tool set is used to perform post-map simulation.

- Post place-and-route simulation: The post place and route simulation is done after place and route process which shows all the block delays and routing delays included in the design. If there are any errors in the simulation, changes are made in design and the design implementation process is repeated.
- Xilinx Device Programming: After the design is completely routed, the programming file for FPGA is generated using bit generator in Xilinx tool. The bit generator produces a .BIT file, a binary file that can be downloaded to the FPGA to produce the required functionality. After generating the program file, the device is configured using any of the configuration techniques such as Boundary-scan technique, Master/Slave serial mode etc. Xilinx uses iMPACT tool for the configuration of FPGAs. The .BIT file generated is programmed through a JTAG cable in boundary scan technique. The iMPACT tool shows the user whether the device is properly programmed or not. If there is any problem with the interface the FPGA has to be reprogrammed. This stage also includes generating .BIN files to program PROMs that can be used in the hardware prototype. The details of programming a PROM are explained later in this chapter.
- Functional verification of the hardware prototype: Once, the FPGA is programmed, it is checked for the functionality at the final stage of the flow. If there are any errors in the functionality then changes are made in the design and the entire process is repeated from design entry stage.

A Xilinx SPARTAN2E 1.8V FPGA with 200K gates capacity is used as a hardware prototype for the processor used to calculate EMC. The post map report of the EMC processor with the parallel arithmetic unit shows that the equivalent gate count of the design is 24,459. The post map report of the EMC processor with the serial arithmetic unit shows that the equivalent gate count of the design is 23,890. Hence any FPGA with much smaller capacity than 200K gates can also be used for design. The operating temperature range of the Spartan2E FPGA in PQ208 package is -40^{o}C to 125^{o}C. This operating range is suitable to the required EMC processor's operating

range. The VHDL modules designed for the interface module and the arithmetic units are instantiated and the whole design is downloaded on to the Spartan2E FPGA using the FPGA design flow described in this section. The following sections explain the development boards used in the hardware prototype and the experimental set up used to test the hardware prototype in a wide range of temperatures and humidity.

5.2 Spartan 2E Development Board:

A Digilab 2E (D2E) development board shown in Figure 5-2 is used in the design [21]. The board has a 200,000 gate Xilinx Spartan2E FPGA in a PQ208 package that provides 143 user I/O pins [20]. All I/O pins are routed either to expansion ports shown in Figure 5-2, or to the ports and other on-board devices. The .BIT file is downloaded to the FPGA through the JTAG port using a JTAG cable in boundary-scan mode of configuration. The JTAG cable connects the board to any computers' parallel port making the interface a lot easier. The FPGA outputs are connected to another peripheral board used for user controls and output display through the expansion connectors. A Digilab bread-board is used in between the development board and the peripheral board to accommodate for the sensor and the pull-up resistors used in the design. The switch SW1 shown in Figure 5-2 varies its position between 'PORT' mode and 'JTAG' mode. For boundary-scan mode the SW1 is to be placed in 'JTAG' position. The configuration mode depends on the inputs given to the mode pins shown in Figure 5-2. In boundary-scan mode the inputs to the mode pins do not make any difference as the default values are considered. But, in other configuration modes such as programming the FPGA from an external PROM, proper inputs are to be given to the mode pins.

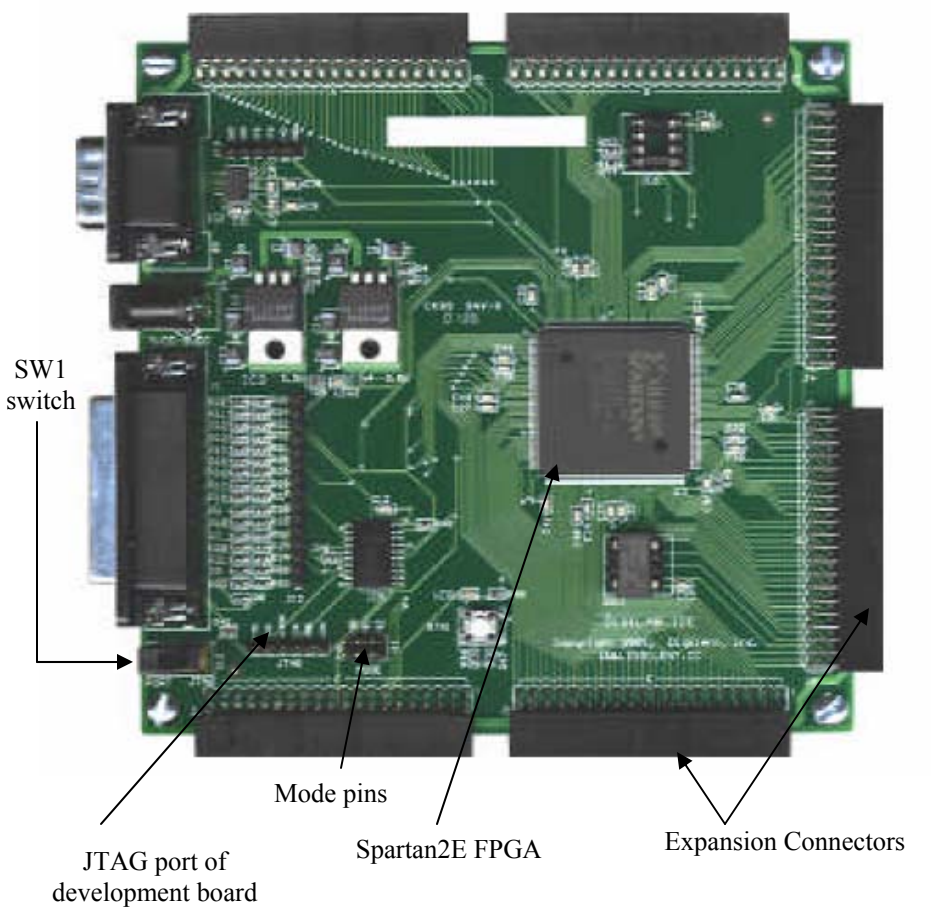


Figure 5-2: Development board of Spartan2E FPGA.

5.3 Configuration of OTP PROM:

A Xilinx OTP (One time programmable) PROM (Programmable Read Only Memory) xc17s200a is used to store the configuration bit files for the Spartan2E FPGA. The Spartan 2E FPGA loses its contents as soon as the power is turned off. Hence an xc17s200a PROM is programmed with the configuration data of the processor so that the FPGA configures itself from the PROM every time the power is turned on. The xc17s200a PROM is programmed using .BIN file generated by Xilinx iMPACT tool from the .bit file of the design. Two xc17s200a PROMs are used in the EMC processor hardware prototype. One PROM is programmed with the .BIN file generated from the EMC processor that uses the parallel arithmetic unit. Another PROM is programmed with the .BIN file generated from the EMC processor that uses the serial arithmetic unit. An external PROM programmer 'Chip Master 6000' is

used to program the PROMs. In order for FPGA to program itself from PROM the configuration mode has to be in master-serial mode. In master-serial mode the SW1 switch of the development board shown in Figure 5-2 has to be in 'PORT' position, and all the mode pins are to be connected by jumpers to ensure that the FPGA is configured properly.

5.4 Experimental Setup:

The EMC processor as implemented by the Xilinx ISE 6.2.3i was downloaded to the Spartan2E FPGA of the development board shown in Figure 5-2. The entire description of the EMC processor in VHDL (for both serial and parallel arithmetic units) is included in Appendix 1. An implementation constraints file is also written in the design entry stage assigning the proper outputs of the expansion ports to the peripheral board display of the hardware prototype that is shown in Figure 5-3. This constraints file is shown in Appendix 2. The sensor interfaces with the FPGA through the breadboard, and the output values are shown on the display device of the peripheral board. The entire experimental setup is shown in Figure 5-3. The switches on the peripheral board will control the display of the output. The display shows temperature (both in ^{o}C and ^{o}F), humidity, and the value of the EMC depending on the inputs given through the switches.

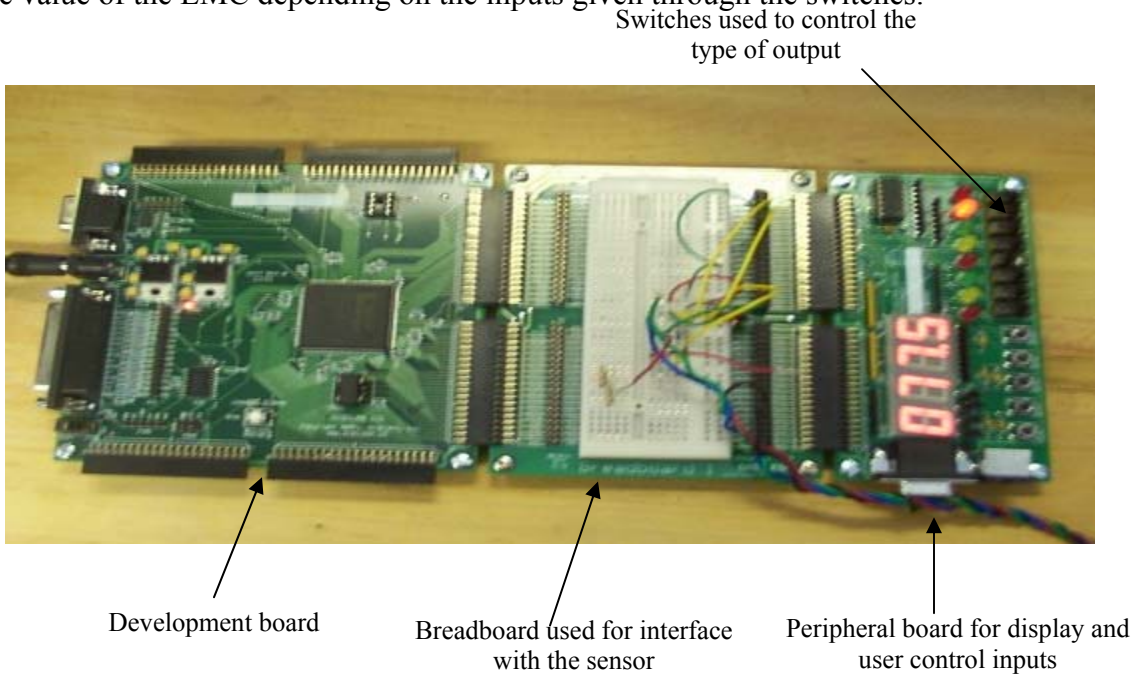


Figure 5-3: Experimental set up of hardware prototype.

5.5 Experimental EMC Processor Performance:

A temperature/humidity chamber was used to test the prototype for different ranges of temperature and humidity. This chamber is shown in Figure 5-4. The sensor placed on a breadboard was placed inside the chamber, and the conditions in the chamber were maintained with the user control inputs shown in Figure 5-4. The hardware prototype is tested for different conditions of temperatures ranging from 12.1^{o}C to 89.5^{o}C and different ranges of humidity from 9.9%RH to 92.3%RH for two PROMs. These values are shown in Table 5-1 and Table 5-2. A comparison of outputs from the prototype to the outputs calculated from the Equation set (4-8) and (4-9) for EMC using MATLAB is also shown in Table 5-1 and Table 5-2. The values of error shown indicate there are very few instances where the value of error is 0.1, which is negligible.

Figure 5-4: Temperature/humidity chamber used for testing the prototype.

value of temperature in Celsius(T)	value of relative humidity in % (h)	value of EMC observed on display device of prototype(M)	value of EMC computed from equations using MATLAB	error
12.1	23	5.2	5.1	0.1
12.9	39.5	7.9	7.9	0
13.9	51.1	9.6	9.6	0
13.1	61.5	11.6	11.5	0.1
14	91.2	21.7	21.7	0
30	12.3	2.9	2.9	0
26.5	25.5	5.4	5.4	0
27	32.6	6.5	6.5	0
30.2	53.6	9.5	9.5	0
30.4	72.4	13.4	13.3	0.1
30.1	92.3	21.5	21.5	0
49.7	9.9	2	2	0
50.3	35.7	6.1	6.1	0
50	48.3	7.9	7.9	0
51.3	59.1	9.5	9.5	0
48.6	82.3	15.3	15.3	0
48.1	90.1	18.9	18.9	0
70.9	10.8	1.7	1.7	0
70	30.3	4.7	4.7	0
71.8	44.9	6.4	6.4	0
70.1	62.6	9	9	0
70.5	81.9	13.4	13.4	0
69.3	88.3	16.1	16.1	0
84.5	12.7	1.7	1.7	0
86.7	32.9	4.2	4.2	0
80.1	52.4	6.9	6.9	0
84.6	64.3	8.3	8.3	0
87	75.3	10.3	10.3	0
89.5	88.8	14.5	14.5	0

Table 5-1: Comparison of values of EMC observed on display of EMC processor hardware prototype (parallel arithmetic unit) and value of EMC calculated using MATLAB.

value of temperature in Celsius(T)	value of relative humidity in % (h)	value of EMC observed on display device of prototype(M)	value of EMC computed from equations using MATLAB	error
13.6	23.4	5.2	5.2	0
9.7	36.9	7.4	7.4	0
10.9	45.6	8.9	8.8	0.1
11.3	69.6	13.3	13.3	0
9.5	77.4	15.6	15.5	0.1
29.2	12.5	2.5	2.5	0
26.5	27.5	5.7	5.7	0
30.2	51.6	9.2	9.2	0
30.2	70.5	12.8	12.8	0
30.1	91	20.6	20.6	0
50.3	26.5	4.9	4.9	0
50.3	45.3	7.5	7.5	0
54.5	62.5	9.9	9.9	0
49.6	80.5	14.5	14.5	0
67.1	30.5	4.8	4.8	0
68.5	47.2	6.9	6.9	0
70.2	67.6	9.9	9.9	0
70.5	80.9	13.1	13.1	0
65	87.5	16	16	0
82	15.4	2.2	2.2	0
85.2	35.3	4.6	4.6	0
84.9	48.9	6.2	6.2	0
83.2	60.3	7.8	7.8	0
87.5	72.4	9.6	9.6	0
82.1	86.7	14.2	14.2	0

Table 5-2: Comparison of values of EMC observed on display of EMC processor hardware prototype (serial arithmetic unit) and value of EMC calculated using MATLAB.

Conclusions

A special purpose sensor and processor system was developed to calculate EMC of wood. EMC of wood is related to the temperature and relative humidity of surrounding air. A SHT71, a Sensirion temperature/humidity sensor was chosen to obtain the values of temperature and relative humidity. The choice of the sensor is appropriate considering the factors that it produces digital output of both temperature and relative humidity, and it consumes low power.

The sensor requires a special controller for the interface between it and the processor. An interface module is designed which will interface the sensor to the rest of the design. The interface module performs various functions such as communicating with the sensor and obtaining the digital readouts of temperature and relative humidity. A1.5kΩ pull-up resistor is used at the data output of the interface module to pull the value high when required.

The equations used to determine EMC contain many additions, subtractions, multiplications, and divisions with coefficients in a wide range. These equations are scaled so that all the coefficients can fit in a smaller range. The scaling is used to reduce the bit-width required by the design. A two's complement binary number system and fixed point arithmetic system were used in the EMC processor design. Two different versions of the arithmetic unit were developed to calculate EMC. They are: EMC processor parallel arithmetic unit and EMC processor serial arithmetic unit. The two arithmetic units were compared in terms of logic area and speed performance.

The interface module, arithmetic unit and controller are coded in VHDL and instantiated to form the entire EMC processor system. The VHDL design was synthesized and implemented to a Xilinx Spartan2E FPGA to form the EMC processor hardware prototype. The configuration files for the EMC processor, with the parallel arithmetic unit and the EMC processor with the serial arithmetic unit, that were generated using Xilinx 6.2.3i were downloaded to two separate Xilinx xc17s200a PROMs. The hardware

prototype was tested and validated for a wide range of temperatures and humidity in a temperature/humidity chamber. The experimental results of EMC using both PROMs were tabulated and compared with the theoretical values of EMC. These values match in many cases producing a negligible error of 0.1 in a very few instances.

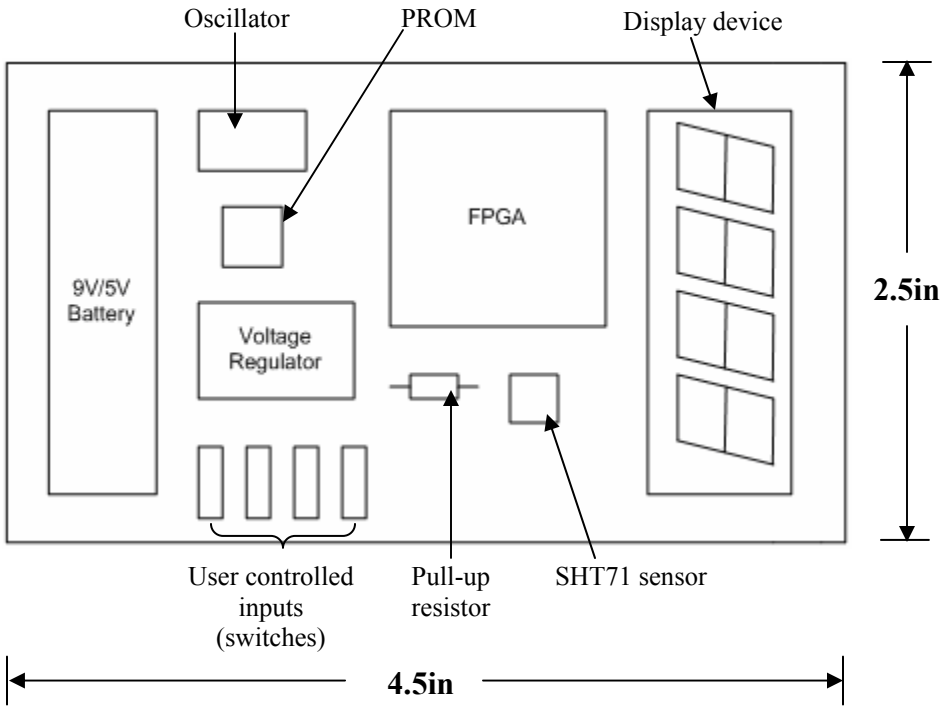


Figure 6-1: Estimated aerial top view of EMC calculating device.

Figure 6-1 shows an estimated aerial top view of the EMC calculator if the EMC processor were built on a single sided Printed Circuit Board (PCB). The designed EMC processor would be downloaded as a bit file to the FPGA shown in Figure 6-1. The dimensions of the display device, FPGA, Oscillator, and voltage regulator shown in Figure 6-1 are the same as those used in the hardware prototype. The estimated values of dimensions if a single sided Printed Circuit Board (PCB) were used are 4.5in X 2.5in. If a double sided PWB is used, then the dimensions would be much smaller. From these values of the dimensions, it can be concluded that a hand-held, battery operated device

can be built to calculate the EMC of wood using the designed special purpose EMC sensor and processor system.

Thus, a special purpose EMC processor was developed which can be implemented as a handheld device. The main concern in the design of the EMC processor was the area occupied by the design on chip. However, the total power consumed by the design is not estimated in this thesis. A power estimation of the entire EMC processor can be done as a future work and power reduction measures can be used in the design to reduce the power consumption. Two different versions of the arithmetic unit were developed in the EMC processor design. This thesis compared these two versions in terms of speed and area on chip. The parallel arithmetic unit uses more area and is faster compared to that of the serial arithmetic unit. The serial arithmetic unit uses less area and is slower compared to that of the parallel arithmetic unit. Therefore, the serial arithmetic unit is better design in terms of hardware used. However, a comparison of the two arithmetic units in terms of the power consumed can be done. This would help in deciding which arithmetic unit utilizes less power and is more suitable for a hand-held device.

Appendices:

Appendix 1: VHDL Description of the EMC Processor:

Appendix 1-1: VHDL Description of the EMC Processor using the parallel arithmetic unit:

```
--VHDL description for the EMC processor.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity moisture_processor_final is
port(clk,start_final,sw1,sw2,c_r_f,clear_final,datain:in std_logic;
        sck,dataout:out std_logic;
        an1,an2,an3,an4,ca,cb,cc,cd,ce,cf,cg,dp_final:out std_logic);
end moisture_processor_final;

architecture Behavioral of moisture_processor_final is
component moisture_cal is
port(clk_emc,clear_emc,start_emc,data_in_emc,c_or_f:in std_logic;
    sck_emc,data_out_emc:out std_logic;
  en_triout,toff_out,fromff_out,clrcheck_ff:out std_logic;
  clr_errorout,error_inff,error_outf:out std_logic;
zero_countout,hum_mout,datain_out,wait_mout,measure_mout,temp_mout,wait_okout:out std_logic;
error_crcout,letincrc_out,loadcrc_sr_out,done_arith_out,loadsot_out:out std_logic;
startarith_out,loadtemp_out:out std_logic;
moisture_emc:out std_logic_vector(15 downto 0);
temp_out,humidity_percent_emc:out std_logic_vector(18 downto 0));
end component moisture_cal;
component display_unit is
port(clk,start_display:in std_logic;
    display_in:in std_logic_vector(18 downto 0);
        anode1,anode2,anode3,anode4:out std_logic;
        dp:out std_logic;
        cat_a,cat_b,cat_c,cat_d,cat_e,cat_f,cat_g:out std_logic);
end component display_unit;
component mux_display_unit is
port(sel:in std_logic;
    a,b:in std_logic_vector(18 downto 0);
        c:out std_logic_vector(18 downto 0));
end component mux_display_unit;

signal moisture,temperature,humidity,dis_input,t_or_h:std_logic_vector(18 downto 0);
signal moisture_temp:std_logic_vector(15 downto 0);
signal clk_int:std_logic;
signal div_clk:std_logic_vector(15 downto 0);
begin
moisture <= "000"&moisture_temp;
CLKPROCESS:process (clk)
begin
if clk = '1' and clk'Event then
```

```
            div_clk <= div_clk + 1;
end if;
end process;
clk_int <= div_clk(15);

MOISCAL:moisture_cal port map(clk_emc =>clk_int, clear_emc=>clear_final, start_emc=>start_final,
                              data_in_emc=>datain, c_or_f =>c_r_f, sck_emc =>sck, data_out_emc
                              =>dataout, moisture_emc =>moisture_temp, temp_out => temperature,
                              humidity_percent_emc =>humidity);

DISPUNIT:display_unit port map   (clk =>clk_int, start_display =>start_final, display_in =>dis_input,
                              anode1 =>an1, anode2 =>an2, anode3 =>an3, anode4 =>an4, cat_a =>
                              ca, cat_b => cb, cat_c => cc, cat_d => cd, cat_e => ce, cat_f => cf,
                              cat_g => cg, dp => dp_final);
MDISP_MOIS:mux_display_unit port map(sel => sw1, a =>t_or_h,b =>moisture,  c =>dis_input);
MDISP_TORH:mux_display_unit port map(sel => sw2, a =>temperature, b =>humidity, c =>t_or_h);
end Behavioral;

--VHDL module for display unit.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity display_unit is
port(clk,start_display:in std_logic;
    display_in:in std_logic_vector(18 downto 0);
            anode1,anode2,anode3,anode4:out std_logic;
            dp:out std_logic;
            cat_a,cat_b,cat_c,cat_d,cat_e,cat_f,cat_g:out std_logic);
end display_unit;
architecture Behavioral of display_unit is
component display_comparator is
port(clk,sign_out_in,start:in std_logic;
              display_in:in std_logic_vector(17 downto 0);
              d_p:out std_logic;
              an1_f,an2_f,an3_f,an4_f,ca,cb,cc,cd,ce,cf,cg:out std_logic);
end component display_comparator;
component sign_check is
port(display_input:in std_logic_vector(18 downto 0);
    display_output:out std_logic_vector(17 downto 0);
            sign_out:out std_logic);
end component sign_check;
signal sign_out_f:std_logic;
signal display_f:std_logic_vector(17 downto 0);
begin
DC:display_comparator port map(clk =>clk, sign_out_in =>sign_out_f, start =>start_display, display_in
=>display_f, an1_f =>anode1, an2_f =>anode2, an3_f =>anode3, an4_f =>anode4, ca =>cat_a, cb =>cat_b,
cc =>cat_c, cd =>cat_d, ce =>cat_e, cf =>cat_f, cg =>cat_g, d_p => dp );
SC:sign_check port map(display_input =>display_in, display_output =>display_f, sign_out =>sign_out_f);
end Behavioral;

--This VHDL module for display comparator which gets 18-bit vector as input.  The sign --bit is considered
in another module and a signal is sent if positive or negative.
 library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity display_comparator is
port(clk,sign_out_in,start:in std_logic;
              display_in:in std_logic_vector(17 downto 0);
              d_p:out std_logic;
              an1_f,an2_f,an3_f,an4_f,ca,cb,cc,cd,ce,cf,cg:out std_logic);
end display_comparator;
architecture Behavioral of display_comparator is
component cathode_gen is
port(cathode_input:in std_logic_vector(3 downto 0);
            sel_mux:in std_logic;
            caf,cbf,ccf,cdf,cef,cff,cgf:out std_logic);
end component cathode_gen;

type state_type is(S0,S1,S2,S3,S4,S5,S6,S7,S8);
signal ps,ns:state_type;

signal f_d_f:std_logic_vector(3 downto 0);
signal display_in_f:std_logic_vector(9 downto 0);
signal display_in_int:std_logic_vector(7 downto 0);
signal i_d_f:std_logic_vector(11 downto 0);
signal cathode_input:std_logic_vector(3 downto 0);--used in generating cathode outputs.
signal sel_mux,an1,an2,an3,an4:std_logic;
begin
display_in_int <= display_in(17 downto 10);
INTEGERPROCESS:process(display_in_int,f_d_f) is
begin
if(f_d_f = "1010")then
            case display_in_int is

            when "00000000" => i_d_f <="000000000001";when "00000001" => i_d_f <="000000000010";
            when "00000010" => i_d_f <="000000000011";when "00000011" => i_d_f <="000000000100";
            when "00000100" => i_d_f <="000000000101";when "00000101" => i_d_f <="000000000110";
            when "00000110" => i_d_f <="000000000111";when "00000111" => i_d_f <="000000001000";
            when "00001000" => i_d_f <="000000001001";when "00001001" => i_d_f <="000000010000";
            when "00001010" => i_d_f <="000000010001";when "00001011" => i_d_f <="000000010010";
            when "00001100" => i_d_f <="000000010011";when "00001101" => i_d_f <="000000010100";
            when "00001110" => i_d_f <="000000010101";when "00001111" => i_d_f <="000000010110";
            when "00010000" => i_d_f <="000000010111";when "00010001" => i_d_f <="000000011000";
            when "00010010" => i_d_f <="000000011001";when "00010011" => i_d_f <="000000100000";
            when "00010100" => i_d_f <="000000100001";when "00010101" => i_d_f <="000000100010";
            when "00010110" => i_d_f <="000000100011";when "00010111" => i_d_f <="000000100100";
            when "00011000" => i_d_f <="000000100101";when "00011001" => i_d_f <="000000100110";
            when "00011010" => i_d_f <="000000100111";when "00011011" => i_d_f <="000000101000";
            when "00011100" => i_d_f <="000000101001";when "00011101" => i_d_f <="000000110000";
            when "00011110" => i_d_f <="000000110001";when "00011111" => i_d_f <="000000110010";
            when "00100000" => i_d_f <="000000110011";when "00100001" => i_d_f <="000000110100";
            when "00100010" => i_d_f <="000000110101";when "00100011" => i_d_f <="000000110110";
            when "00100100" => i_d_f <="000000110111";when "00100101" => i_d_f <="000000111000";
            when "00100110" => i_d_f <="000000111001";when "00100111" => i_d_f <="000001000000";
            when "00101000" => i_d_f <="000001000001";when "00101001" => i_d_f <="000001000010";
            when "00101010" => i_d_f <="000001000011";when "00101011" => i_d_f <="000001000100";
            when "00101100" => i_d_f <="000001000101";when "00101101" => i_d_f <="000001000110";
            when "00101110" => i_d_f <="000001000111";when "00101111" => i_d_f <="000001001000";
            when "00110000" => i_d_f <="000001001001";when "00110001" => i_d_f <="000001010000";
            when "00110010" => i_d_f <="000001010001";when "00110011" => i_d_f <="000001010010";
```

```
when "00110100" => i_d_f <="000001010011";when "00110101" => i_d_f <="000001010100";
when "00110110" => i_d_f <="000001010101";when "00110111" => i_d_f <="000001010110";
when "00111000" => i_d_f <="000001010111";when "00111001" => i_d_f <="000001011000";
when "00111010" => i_d_f <="000001011001";when "00111011" => i_d_f <="000001100000";
when "00111100" => i_d_f <="000001100001";when "00111101" => i_d_f <="000001100010";
when "00111110" => i_d_f <="000001100011";when "00111111" => i_d_f <="000001100100";
when "01000000" => i_d_f <="000001100101";when "01000001" => i_d_f <="000001100110";
when "01000010" => i_d_f <="000001100111";when "01000011" => i_d_f <="000001101000";
when "01000100" => i_d_f <="000001101001";when "01000101" => i_d_f <="000001110000";
when "01000110" => i_d_f <="000001110001";when "01000111" => i_d_f <="000001110010";
when "01001000" => i_d_f <="000001110011";when "01001001" => i_d_f <="000001110100";
when "01001010" => i_d_f <="000001110101";when "01001011" => i_d_f <="000001110110";
when "01001100" => i_d_f <="000001110111";when "01001101" => i_d_f <="000001111000";
when "01001110" => i_d_f <="000001111001";when "01001111" => i_d_f <="000010000000";
when "01010000" => i_d_f <="000010000001";when "01010001" => i_d_f <="000010000010";
when "01010010" => i_d_f <="000010000011";when "01010011" => i_d_f <="000010000100";
when "01010100" => i_d_f <="000010000101";when "01010101" => i_d_f <="000010000110";
when "01010110" => i_d_f <="000010000111";when "01010111" => i_d_f <="000010001000";
when "01011000" => i_d_f <="000010001001";when "01011001" => i_d_f <="000010010000";
when "01011010" => i_d_f <="000010010001";when "01011011" => i_d_f <="000010010010";
when "01011100" => i_d_f <="000010010011";when "01011101" => i_d_f <="000010010100";
when "01011110" => i_d_f <="000010010101";when "01011111" => i_d_f <="000010010110";
when "01100000" => i_d_f <="000010010111";when "01100001" => i_d_f <="000010011000";
when "01100010" => i_d_f <="000010011001";when "01100011" => i_d_f <="000100000000";
when "01100100" => i_d_f <="000100000001";when "01100101" => i_d_f <="000100000010";
when "01100110" => i_d_f <="000100000011";when "01100111" => i_d_f <="000100000100";
when "01101000" => i_d_f <="000100000101";when "01101001" => i_d_f <="000100000110";
when "01101010" => i_d_f <="000100000111";when "01101011" => i_d_f <="000100001000";
when "01101100" => i_d_f <="000100001001";when "01101101" => i_d_f <="000100010000";
when "01101110" => i_d_f <="000100010001";when "01101111" => i_d_f <="000100010010";
when "01110000" => i_d_f <="000100010011";when "01110001" => i_d_f <="000100010100";
when "01110010" => i_d_f <="000100010101";when "01110011" => i_d_f <="000100010110";
when "01110100" => i_d_f <="000100010111";when "01110101" => i_d_f <="000100011000";
when "01110110" => i_d_f <="000100011001";when "01110111" => i_d_f <="000100100000";
when "01111000" => i_d_f <="000100100001";when "01111001" => i_d_f <="000100100010";
when "01111010" => i_d_f <="000100100011";when "01111011" => i_d_f <="000100100100";
when "01111100" => i_d_f <="000100100101";when "01111101" => i_d_f <="000100100110";
when "01111110" => i_d_f <="000100100111";when "01111111" => i_d_f <="000100101000";
when "10000000" => i_d_f <="000100101001";when "10000001" => i_d_f <="000100110000";
when "10000010" => i_d_f <="000100110001";when "10000011" => i_d_f <="000100110010";
when "10000100" => i_d_f <="000100110011";when "10000101" => i_d_f <="000100110100";
when "10000110" => i_d_f <="000100110101";when "10000111" => i_d_f <="000100110110";
when "10001000" => i_d_f <="000100110111";when "10001001" => i_d_f <="000100111000";
when "10001010" => i_d_f <="000100111001";when "10001011" => i_d_f <="000101000000";
when "10001100" => i_d_f <="000101000001";when "10001101" => i_d_f <="000101000010";
when "10001110" => i_d_f <="000101000011";when "10001111" => i_d_f <="000101000100";
when "10010000" => i_d_f <="000101000101";when "10010001" => i_d_f <="000101000110";
when "10010010" => i_d_f <="000101000111";when "10010011" => i_d_f <="000101001000";
when "10010100" => i_d_f <="000101001001";when "10010101" => i_d_f <="000101010000";
when "10010110" => i_d_f <="000101010001";when "10010111" => i_d_f <="000101010010";
when "10011000" => i_d_f <="000101010011";when "10011001" => i_d_f <="000101010100";
when "10011010" => i_d_f <="000101010101";when "10011011" => i_d_f <="000101010110";
when "10011100" => i_d_f <="000101010111";when "10011101" => i_d_f <="000101011000";
when "10011110" => i_d_f <="000101011001";when "10011111" => i_d_f <="000101100000";
when "10100000" => i_d_f <="000101100001";when "10100001" => i_d_f <="000101100010";
when "10100010" => i_d_f <="000101100011";when "10100011" => i_d_f <="000101100100";
```

```
when "10100100" => i_d_f <="000101100101";when "10100101" => i_d_f <="000101100110";
when "10100110" => i_d_f <="000101100111";when "10100111" => i_d_f <="000101101000";
when "10101000" => i_d_f <="000101101001";when "10101001" => i_d_f <="000101110000";
when "10101010" => i_d_f <="000101110001";when "10101011" => i_d_f <="000101110010";
when "10101100" => i_d_f <="000101110011";when "10101101" => i_d_f <="000101110100";
when "10101110" => i_d_f <="000101110101";when "10101111" => i_d_f <="000101110110";
when "10110000" => i_d_f <="000101110111";when "10110001" => i_d_f <="000101111000";
when "10110010" => i_d_f <="000101111001";when "10110011" => i_d_f <="000110000000";
when "10110100" => i_d_f <="000110000001";when "10110101" => i_d_f <="000110000010";
when "10110110" => i_d_f <="000110000011";when "10110111" => i_d_f <="000110000100";
when "10111000" => i_d_f <="000110000101";when "10111001" => i_d_f <="000110000110";
when "10111010" => i_d_f <="000110000111";when "10111011" => i_d_f <="000110001000";
when "10111100" => i_d_f <="000110001001";when "10111101" => i_d_f <="000110010000";
when "10111110" => i_d_f <="000110010001";when "10111111" => i_d_f <="000110010010";
when "11000000" => i_d_f <="000110010011";when "11000001" => i_d_f <="000110010100";
when "11000010" => i_d_f <="000110010101";when "11000011" => i_d_f <="000110010110";
when "11000100" => i_d_f <="000110010111";when "11000101" => i_d_f <="000110011000";
when "11000110" => i_d_f <="000110011001";when "11000111" => i_d_f <="001000000000";
when "11001000" => i_d_f <="001000000001";when "11001001" => i_d_f <="001000000010";
when "11001010" => i_d_f <="001000000011";when "11001011" => i_d_f <="001000000100";
when "11001100" => i_d_f <="001000000101";when "11001101" => i_d_f <="001000000110";
when "11001110" => i_d_f <="001000000111";when "11001111" => i_d_f <="001000001000";
when "11010000" => i_d_f <="001000001001";when "11010001" => i_d_f <="001000010000";
when "11010010" => i_d_f <="001000010001";when "11010011" => i_d_f <="001000010010";
when "11010100" => i_d_f <="001000010011";when "11010101" => i_d_f <="001000010100";
when "11010110" => i_d_f <="001000010101";when "11010111" => i_d_f <="001000010110";
when "11011000" => i_d_f <="001000010111";when "11011001" => i_d_f <="001000011000";
when "11011010" => i_d_f <="001000011001";when "11011011" => i_d_f <="001000100000";
when "11011100" => i_d_f <="001000100001";when "11011101" => i_d_f <="001000100010";
when "11011110" => i_d_f <="001000100011";when "11011111" => i_d_f <="001000100100";
when "11100000" => i_d_f <="001000100101";when "11100001" => i_d_f <="001000100110";
when "11100010" => i_d_f <="001000100111";when "11100011" => i_d_f <="001000101000";
when "11100100" => i_d_f <="001000101001";when "11100101" => i_d_f <="001000110000";
when "11100110" => i_d_f <="001000110001";when "11100111" => i_d_f <="001000110010";
when "11101000" => i_d_f <="001000110011";when "11101001" => i_d_f <="001000110100";
when "11101010" => i_d_f <="001000110101";when "11101011" => i_d_f <="001000110110";
when "11101100" => i_d_f <="001000110111";when "11101101" => i_d_f <="001000111000";
when "11101110" => i_d_f <="001000111001";when "11101111" => i_d_f <="001001000000";
when "11110000" => i_d_f <="001001000001";when "11110001" => i_d_f <="001001000010";
when "11110010" => i_d_f <="001001000011";when "11110011" => i_d_f <="001001000100";
when "11110100" => i_d_f <="001001000101";when "11110101" => i_d_f <="001001000110";
when "11110110" => i_d_f <="001001000111";when "11110111" => i_d_f <="001001001000";
when "11111000" => i_d_f <="001001001001";when "11111001" => i_d_f <="001001010000";
when "11111010" => i_d_f <="001001010001";when "11111011" => i_d_f <="001001010010";
when "11111100" => i_d_f <="001001010011";when "11111101" => i_d_f <="001001010100";
when "11111110" => i_d_f <="001001010101";when "11111111" => i_d_f <="001001010110";
when others => i_d_f <= "000000000000";
end case;
else
case display_in_int is

when "00000000" => i_d_f <="000000000000";when "00000001" => i_d_f <="000000000001";
when "00000010" => i_d_f <="000000000010";when "00000011" => i_d_f <="000000000011";
when "00000100" => i_d_f <="000000000100";when "00000101" => i_d_f <="000000000101";
when "00000110" => i_d_f <="000000000110";when "00000111" => i_d_f <="000000000111";
when "00001000" => i_d_f <="000000001000";when "00001001" => i_d_f <="000000001001";
```

```
when "00001010" => i_d_f <="000000010000";when "00001011" => i_d_f <="000000010001";
when "00001100" => i_d_f <="000000010010";when "00001101" => i_d_f <="000000010011";
when "00001110" => i_d_f <="000000010100";when "00001111" => i_d_f <="000000010101";
when "00010000" => i_d_f <="000000010110";when "00010001" => i_d_f <="000000010111";
when "00010010" => i_d_f <="000000011000";when "00010011" => i_d_f <="000000011001";
when "00010100" => i_d_f <="000000100000";when "00010101" => i_d_f <="000000100001";
when "00010110" => i_d_f <="000000100010";when "00010111" => i_d_f <="000000100011";
when "00011000" => i_d_f <="000000100100";when "00011001" => i_d_f <="000000100101";
when "00011010" => i_d_f <="000000100110";when "00011011" => i_d_f <="000000100111";
when "00011100" => i_d_f <="000000101000";when "00011101" => i_d_f <="000000101001";
when "00011110" => i_d_f <="000000110000";when "00011111" => i_d_f <="000000110001";
when "00100000" => i_d_f <="000000110010";when "00100001" => i_d_f <="000000110011";
when "00100010" => i_d_f <="000000110100";when "00100011" =>i_d_f <="000000110101";
when "00100100" => i_d_f <="000000110110";when "00100101" => i_d_f <="000000110111";
when "00100110" => i_d_f <="000000111000";when "00100111" => i_d_f <="000000111001";
when "00101000" => i_d_f <="000001000000";when "00101001" => i_d_f <="000001000001";
when "00101010" => i_d_f <="000001000010";when "00101011" => i_d_f <="000001000011";
when "00101100" => i_d_f <="000001000100";when "00101101" => i_d_f <="000001000101";
when "00101110" => i_d_f <="000001000110";when "00101111" => i_d_f <="000001000111";
when "00110000" => i_d_f <="000001001000";when "00110001" => i_d_f <="000001001001";
when "00110010" => i_d_f <="000001010000";when "00110011" => i_d_f <="000001010001";
when "00110100" => i_d_f <="000001010010";when "00110101" => i_d_f <="000001010011";
when "00110110" => i_d_f <="000001010100";when "00110111" => i_d_f <="000001010101";
when "00111000" => i_d_f <="000001010110";when "00111001" => i_d_f <="000001010111";
when "00111010" => i_d_f <="000001011000";when "00111011" => i_d_f <="000001011001";
when "00111100" => i_d_f <="000001100000";when "00111101" => i_d_f <="000001100001";
when "00111110" => i_d_f <="000001100010";when "00111111" => i_d_f <="000001100011";
when "01000000" => i_d_f <="000001100100";when "01000001" => i_d_f <="000001100101";
when "01000010" => i_d_f <="000001100110";when "01000011" => i_d_f <="000001100111";
when "01000100" => i_d_f <="000001101000";when "01000101" => i_d_f <="000001101001";
when "01000110" => i_d_f <="000001110000";when "01000111" => i_d_f <="000001110001";
when "01001000" => i_d_f <="000001110010";when "01001001" => i_d_f <="000001110011";
when "01001010" => i_d_f <="000001110100";when "01001011" => i_d_f <="000001110101";
when "01001100" => i_d_f <="000001110110";when "01001101" => i_d_f <="000001110111";
when "01001110" => i_d_f <="000001111000";when "01001111" => i_d_f <="000001111001";
when "01010000" => i_d_f <="000010000000";when "01010001" => i_d_f <="000010000001";
when "01010010" => i_d_f <="000010000010";when "01010011" => i_d_f <="000010000011";
when "01010100" => i_d_f <="000010000100";when "01010101" => i_d_f <="000010000101";
when "01010110" => i_d_f <="000010000110";when "01010111" => i_d_f <="000010000111";
when "01011000" => i_d_f <="000010001000";when "01011001" => i_d_f <="000010001001";
when "01011010" => i_d_f <="000010010000";when "01011011" => i_d_f <="000010010001";
when "01011100" => i_d_f <="000010010010";when "01011101" => i_d_f <="000010010011";
when "01011110" => i_d_f <="000010010100";when "01011111" => i_d_f <="000010010101";
when "01100000" => i_d_f <="000010010110";when "01100001" => i_d_f <="000010010111";
when "01100010" => i_d_f <="000010011000";when "01100011" => i_d_f <="000010011001";
when "01100100" => i_d_f <="000100000000";when "01100101" => i_d_f <="000100000001";
when "01100110" => i_d_f <="000100000010";when "01100111" => i_d_f <="000100000011";
when "01101000" => i_d_f <="000100000100";when "01101001" => i_d_f <="000100000101";
when "01101010" => i_d_f <="000100000110";when "01101011" => i_d_f <="000100000111";
when "01101100" => i_d_f <="000100001000";when "01101101" => i_d_f <="000100001001";
when "01101110" => i_d_f <="000100010000";when "01101111" => i_d_f <="000100010001";
when "01110000" => i_d_f <="000100010010";when "01110001" => i_d_f <="000100010011";
when "01110010" => i_d_f <="000100010100";when "01110011" => i_d_f <="000100010101";
when "01110100" => i_d_f <="000100010110";when "01110101" => i_d_f <="000100010111";
when "01110110" => i_d_f <="000100011000";when "01110111" => i_d_f <="000100011001";
when "01111000" => i_d_f <="000100100000";when "01111001" => i_d_f <="000100100001";
```

```
when "01111010" => i_d_f <="000100100010";when "01111011" => i_d_f <="000100100011";
when "01111100" => i_d_f <="000100100100";when "01111101" => i_d_f <="000100100101";
when "01111110" => i_d_f <="000100100110";when "01111111" => i_d_f <="000100100111";
when "10000000" => i_d_f <="000100101000";when "10000001" => i_d_f <="000100101001";
when "10000010" => i_d_f <="000100110000";when "10000011" => i_d_f <="000100110001";
when "10000100" => i_d_f <="000100110010";when "10000101" => i_d_f <="000100110011";
when "10000110" => i_d_f <="000100110100";when "10000111" => i_d_f <="000100110101";
when "10001000" => i_d_f <="000100110110";when "10001001" => i_d_f <="000100110111";
when "10001010" => i_d_f <="000100111000";when "10001011" => i_d_f <="000100111001";
when "10001100" => i_d_f <="000101000000";when "10001101" => i_d_f <="000101000001";
when "10001110" => i_d_f <="000101000010";when "10001111" => i_d_f <="000101000011";
when "10010000" => i_d_f <="000101000100";when "10010001" => i_d_f <="000101000101";
when "10010010" => i_d_f <="000101000110";when "10010011" => i_d_f <="000101000111";
when "10010100" => i_d_f <="000101001000";when "10010101" => i_d_f <="000101001001";
when "10010110" => i_d_f <="000101010000";when "10010111" => i_d_f <="000101010001";
when "10011000" => i_d_f <="000101010010";when "10011001" => i_d_f <="000101010011";
when "10011010" => i_d_f <="000101010100";when "10011011" => i_d_f <="000101010101";
when "10011100" => i_d_f <="000101010110";when "10011101" => i_d_f <="000101010111";
when "10011110" => i_d_f <="000101011000";when "10011111" => i_d_f <="000101011001";
when "10100000" => i_d_f <="000101100000";when "10100001" => i_d_f <="000101100001";
when "10100010" => i_d_f <="000101100010";when "10100011" => i_d_f <="000101100011";
when "10100100" => i_d_f <="000101100100";when "10100101" => i_d_f <="000101100101";
when "10100110" => i_d_f <="000101100110";when "10100111" => i_d_f <="000101100111";
when "10101000" => i_d_f <="000101101000";when "10101001" => i_d_f <="000101101001";
when "10101010" => i_d_f <="000101110000";when "10101011" => i_d_f <="000101110001";
when "10101100" => i_d_f <="000101110010";when "10101101" => i_d_f <="000101110011";
when "10101110" => i_d_f <="000101110100";when "10101111" => i_d_f <="000101110101";
when "10110000" => i_d_f <="000101110110";when "10110001" => i_d_f <="000101110111";
when "10110010" => i_d_f <="000101111000";when "10110011" => i_d_f <="000101111001";
when "10110100" => i_d_f <="000110000000";when "10110101" => i_d_f <="000110000001";
when "10110110" => i_d_f <="000110000010";when "10110111" => i_d_f <="000110000011";
when "10111000" => i_d_f <="000110000100";when "10111001" => i_d_f <="000110000101";
when "10111010" => i_d_f <="000110000110";when "10111011" => i_d_f <="000110000111";
when "10111100" => i_d_f <="000110001000";when "10111101" => i_d_f <="000110001001";
when "10111110" => i_d_f <="000110010000";when "10111111" => i_d_f <="000110010001";
when "11000000" => i_d_f <="000110010010";when "11000001" => i_d_f <="000110010011";
when "11000010" => i_d_f <="000110010100";when "11000011" => i_d_f <="000110010101";
when "11000100" => i_d_f <="000110010110";when "11000101" => i_d_f <="000110010111";
when "11000110" => i_d_f <="000110011000";when "11000111" => i_d_f <="000110011001";
when "11001000" => i_d_f <="001000000000";when "11001001" => i_d_f <="001000000001";
when "11001010" => i_d_f <="001000000010";when "11001011" => i_d_f <="001000000011";
when "11001100" => i_d_f <="001000000100";when "11001101" => i_d_f <="001000000101";
when "11001110" => i_d_f <="001000000110";when "11001111" => i_d_f <="001000000111";
when "11010000" => i_d_f <="001000001000";when "11010001" => i_d_f <="001000001001";
when "11010010" => i_d_f <="001000010000";when "11010011" => i_d_f <="001000010001";
when "11010100" => i_d_f <="001000010010";when "11010101" => i_d_f <="001000010011";
when "11010110" => i_d_f <="001000010100";when "11010111" => i_d_f <="001000010101";
when "11011000" => i_d_f <="001000010110";when "11011001" => i_d_f <="001000010111";
when "11011010" => i_d_f <="001000011000";when "11011011" => i_d_f <="001000011001";
when "11011100" => i_d_f <="001000100000";when "11011101" => i_d_f <="001000100001";
when "11011110" => i_d_f <="001000100010";when "11011111" => i_d_f <="001000100011";
when "11100000" => i_d_f <="001000100100";when "11100001" => i_d_f <="001000100101";
when "11100010" => i_d_f <="001000100110";when "11100011" => i_d_f <="001000100111";
when "11100100" => i_d_f <="001000101000";when "11100101" => i_d_f <="001000101001";
when "11100110" => i_d_f <="001000110000";when "11100111" => i_d_f <="001000110001";
when "11101000" => i_d_f <="001000110010";when "11101001" => i_d_f <="001000110011";
```

```
            when "11101010" => i_d_f <="001000110100";when "11101011" => i_d_f <="001000110101";
            when "11101100" => i_d_f <="001000110110";when "11101101" => i_d_f <="001000110111";
            when "11101110" => i_d_f <="001000111000";when "11101111" => i_d_f <="001000111001";
            when "11110000" => i_d_f <="001001000000";when "11110001" => i_d_f <="001001000001";
            when "11110010" => i_d_f <="001001000010";when "11110011" => i_d_f <="001001000011";
            when "11110100" => i_d_f <="001001000100";when "11110101" => i_d_f <="001001000101";
            when "11110110" => i_d_f <="001001000110";when "11110111" => i_d_f <="001001000111";
            when "11111000" => i_d_f <="001001001000";when "11111001" => i_d_f <="001001001001";
            when "11111010" => i_d_f <="001001010000";when "11111011" => i_d_f <="001001010001";
            when "11111100" => i_d_f <="001001010010";when "11111101" => i_d_f <="001001010011";
            when "11111110" => i_d_f <="001001010100";when "11111111" => i_d_f <="001001010101";
            when others => i_d_f <= "000000000000";
            end case;
end if;
end process;

display_in_f <= display_in(9 downto 0);
FRACTIONPROCESS: process(display_in_f) is
begin
if(display_in_f >= "0000000000" and display_in_f <="0000101101")then
            f_d_f <= "0000";
elsif(display_in_f >= "0000101110" and display_in_f <="0010010100")then
            f_d_f <= "0001";
elsif(display_in_f >= "0010010101" and display_in_f <="0011111010")then
            f_d_f <= "0010";
elsif(display_in_f >= "0011111011" and display_in_f <="0101100000")then
            f_d_f <= "0011";
elsif(display_in_f >= "0101100001" and display_in_f <="0111000111")then
            f_d_f <= "0100";
elsif(display_in_f >= "0111001000" and display_in_f <="1000101101")then
            f_d_f <= "0101";
elsif(display_in_f >= "1000101110" and display_in_f <="1010010100")then
            f_d_f <= "0110";
elsif(display_in_f >= "1010010101" and display_in_f <="1011111010")then
            f_d_f <= "0111";
elsif(display_in_f >= "1011111011" and display_in_f <="1101100000")then
            f_d_f <= "1000";
elsif(display_in_f >= "1101100001" and display_in_f <="1111000111")then
            f_d_f <= "1001";
elsif(display_in_f >= "1111001000" and display_in_f <="1111111111")then
            f_d_f <= "1010";
end if;
end process;

PB:process(an1,an2,an3,an4,i_d_f,f_d_f)is
begin
            if(an1 = '1')then
            cathode_input <= i_d_f(11 downto 8);
            elsif(an2 = '1')then
            cathode_input <= i_d_f(7 downto 4);
            elsif(an3 = '1')then
            cathode_input <= i_d_f(3 downto 0);
            elsif(an4 = '1')then
            cathode_input <= f_d_f;
            end if;
end process;
```

```
PC:process(clk)is
begin
if(clk'event and clk = '1')then
      ps <= ns;
end if;
end process;

P1:process(clk)is
begin
if(clk'event and clk = '1')then
case ps is
          when S0 =>
                                    an1 <= '1'; an2 <= '1'; an3 <= '1'; an4 <= '1'; sel_mux <= '0'; d_p <= '1';
          when S1 =>
                                    an1 <= '0'; an2 <= '0'; an3 <= '0'; an4 <= '0'; sel_mux <= '0'; d_p <= '1';
          when S2 =>
                                    an1 <= '1'; an2 <= '0'; an3 <= '0'; an4 <= '0'; sel_mux <= '0'; d_p <= '1';
          when S3 =>
                                    an1 <= '0'; an2 <= '1'; an3 <= '0'; an4 <= '0'; sel_mux <= '0'; d_p <= '1';
          when S4 =>
                                    an1 <= '0'; an2 <= '0'; an3 <= '1'; an4 <= '0'; sel_mux <= '0'; d_p <= '0';
          when S5 =>
                                    an1 <= '0'; an2 <= '0'; an3 <= '0'; an4 <= '1'; sel_mux <= '0'; d_p <= '1';
          when S6 =>
                                    an1 <= '0'; an2 <= '1'; an3 <= '0'; an4 <= '0'; sel_mux <= '1'; d_p <= '1';
          when S7 =>
                                    an1 <= '0'; an2 <= '0'; an3 <= '1'; an4 <= '0'; sel_mux <= '0'; d_p <= '0';
          when S8 =>
                                    an1 <= '0'; an2 <= '0'; an3 <= '0'; an4 <= '1'; sel_mux <= '0'; d_p <= '1';
end case;
end if;
end process;

PD:process(ps,start,sign_out_in) is
begin
case ps is
          when S0 => if(start = '1')then
                                              ns <= S1;
                                              else
                                              ns <= S0;
                                              end if;
          when S1 => if(sign_out_in = '0')then
                                              ns <= S2;
                              else
                                              ns <= S6;
                              end if;
          when S2 => ns <= S3;
          when S3 => ns <= S4;
          when S4 => ns <= S5;
          when S5 => ns <= S1;
          when S6 => ns <= S7;
          when S7 => ns <= S8;
          when S8 => ns <= S1;
end case;
end process;
```

```
CATHODEGEN:cathode_gen port map(cathode_input => cathode_input,sel_mux => sel_mux, caf =>ca,
cbf=>cb, ccf=>cc, cdf=>cd, cef=>ce, cff=>cf, cgf =>cg);
an1_f <= an1;an2_f <= an2;an3_f <= an3;an4_f <= an4;
end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity cathode_gen is
port(cathode_input:in std_logic_vector(3 downto 0);
        sel_mux:in std_logic;
        caf,cbf,ccf,cdf,cef,cff,cgf:out std_logic);
end cathode_gen;

architecture Behavioral of cathode_gen is
component mux2x1_cathode is
port(selmux:in std_logic;
        a,b:in std_logic;
        c:out std_logic);
end component mux2x1_cathode;

signal ca_i,cb_i,cc_i,cd_i,ce_i,cf_i,cg_i:std_logic;
begin
process(cathode_input)is
begin
--if(clk'event and clk = '1')then
case cathode_input is
        when "0000" => ca_i <= '0';cb_i<='0';cc_i<='0';cd_i<='0';ce_i<='0';cf_i<='0';cg_i<='1';
        when "0001" => ca_i<='1';cb_i<='0';cc_i<='0';cd_i<='1';ce_i<='1';cf_i<='1';cg_i<='1';
        when "0010" => ca_i <= '0';cb_i<='0';cc_i<='1';cd_i<='0';ce_i<='0';cf_i<='1';cg_i<='0';
        when "0011" => ca_i <= '0';cb_i<='0';cc_i<='0';cd_i<='0';ce_i<='1';cf_i<='1';cg_i<='0';
        when "0100" => ca_i <= '1';cb_i<='0';cc_i<='0';cd_i<='1';ce_i<='1';cf_i<='0';cg_i<='0';
        when "0101" => ca_i <= '0';cb_i<='1';cc_i<='0';cd_i<='0';ce_i<='1';cf_i<='0';cg_i<='0';
        when "0110" => ca_i <= '0';cb_i<='1';cc_i<='0';cd_i<='0';ce_i<='0';cf_i<='0';cg_i<='0';
        when "0111" => ca_i <= '0';cb_i<='0';cc_i<='0';cd_i<='1';ce_i<='1';cf_i<='1';cg_i<='1';
        when "1000"=>  ca_i <= '0';cb_i<='0';cc_i<='0';cd_i<='0';ce_i<='0';cf_i<='0';cg_i<='0';
        when "1001" => ca_i <= '0';cb_i<='0';cc_i<='0';cd_i<='0';ce_i<='1';cf_i<='0';cg_i<='0';
        when others => ca_i <= '0';cb_i<='0';cc_i<='0';cd_i<='0';ce_i<='0';cf_i<='0';cg_i<='1';
end case;
--end if;
end process;
M0:mux2x1_cathode port map(selmux =>sel_mux, a => ca_i, b => '1', c => caf);
M1:mux2x1_cathode port map(selmux =>sel_mux, a => cb_i, b => '1', c => cbf);
M2:mux2x1_cathode port map(selmux =>sel_mux, a => cc_i, b => '1', c => ccf);
M3:mux2x1_cathode port map(selmux =>sel_mux, a => cd_i, b => '1', c => cdf);
M4:mux2x1_cathode port map(selmux =>sel_mux, a => ce_i, b => '1', c => cef);
M5:mux2x1_cathode port map(selmux =>sel_mux, a => cf_i, b => '1', c => cff);
M6:mux2x1_cathode port map(selmux =>sel_mux, a => cg_i, b => '0', c => cgf);
end Behavioral;

--VHDL module for 2X1 multiplexor used at the cathode outputs.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity mux2x1_cathode is
port(selmux:in std_logic;
        a,b:in std_logic;
        c:out std_logic);
end mux2x1_cathode;

architecture Behavioral of mux2x1_cathode is

begin
 process(selmux,a,b)is
 begin
 if(selmux = '0')then
        c <= a;
 else
        c <= b;
 end if;
end process;
end Behavioral;

--this module checks the sign bit of the input vector and generates the true unsigned form and a sign bit
--output also.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity sign_check is
port(display_input:in std_logic_vector(18 downto 0);
    display_output:out std_logic_vector(17 downto 0);
        sign_out:out std_logic);
end sign_check;

architecture Behavioral of sign_check is

begin
P0:process(display_input)is
begin
if(display_input(18) = '1')then
        display_output <= not(display_input(17 downto 0))+1;
        sign_out <= '1';
else
        display_output <= display_input(17 downto 0);
        sign_out <= '0';
end if;
end process;

end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity moisture_cal is
port(clk_emc,clear_emc,start_emc,data_in_emc,c_or_f:in std_logic;
    sck_emc,data_out_emc:out std_logic;
          moisture_emc:out std_logic_vector(15 downto 0);
          temp_out,humidity_percent_emc:out std_logic_vector(18 downto 0));
end moisture_cal;

architecture Behavioral of moisture_cal is
component arithmetic_unit is
   Port (clk_f,clear_final,start_final,arith_ok_f:in std_logic;
                    sot,sorh:in std_logic_vector(13 downto 0);
                              done_arith_f:out std_logic;
                              moisture:out std_logic_vector(15 downto 0);
                              temp_centi,temp_fah,humidity_percent:out std_logic_vector(18 downto 0));
end component arithmetic_unit;
component emc_cal is
port(start_all,clr_all,data_in,clk,done_arith_in:in std_logic;
    sck,data_out_tri :out std_logic;
          arith_ok_f:out std_logic;
          start_arith_out,clr_arith_out:out std_logic;
          sot_out,sorh_out:out std_logic_vector(13 downto 0));
end component emc_cal;

component mux_2x1_temp is
port(sel:in std_logic;
    a,b:in std_logic_vector(18 downto 0);
          c:out std_logic_vector(18 downto 0));
end component mux_2x1_temp;


signal clr_arith_out_f,start_arith_out_f,arithok_f,donearith_f:std_logic;
signal sot_f,sorh_f:std_logic_vector(13 downto 0);
signal temp_fah_f,temp_centi_f:std_logic_vector(18 downto 0);
begin
ARITH_MOIST:arithmetic_unit Port map (clk_f => clk_emc, clear_final =>clr_arith_out_f, start_final
=>start_arith_out_f, arith_ok_f =>arithok_f, sot =>sot_f, sorh =>sorh_f, done_arith_f =>donearith_f,
moisture =>moisture_emc, temp_centi =>temp_centi_f, temp_fah =>temp_fah_f, humidity_percent
=>humidity_percent_emc, load_temp_out => loadtemp_out);
startarith_out <= start_arith_out_f;
INTERFACE_MOIST:emc_cal port map(start_all =>start_emc, clr_all =>clear_emc, data_in
=>data_in_emc, clk =>clk_emc, done_arith_in =>donearith_f, sck =>sck_emc, data_out_tri =>
data_out_emc, en_tri_out => en_triout, to_ff_out  => toff_out, from_ff_out        =>fromff_out,
clr_error_out =>clr_errorout , error_in_ff =>error_inff, error_out_ff=>error_outf, clr_check_ff =>
clrcheck_ff, zero_count_out =>zero_countout, hum_m_out => hum_mout, data_in_out => datain_out,
wait_m_out => wait_mout, measure_m_out => measure_mout, temp_m_out => temp_mout,
error_crc_out => error_crcout, load_crc_sr_out => loadcrc_sr_out, load_sot_out => loadsot_out,
letin_crc_out => letincrc_out, wait_ok_out => wait_okout, arith_ok_f =>arithok_f,
start_arith_out =>start_arith_out_f, clr_arith_out =>clr_arith_out_f, sot_out =>sot_f,
sorh_out =>sorh_f);
done_arith_out <= donearith_f;
MUX_MOIST:mux_2x1_temp port map(sel =>c_or_f, a =>temp_fah_f, b=>temp_centi_f, c =>temp_out);
end Behavioral;

--VHDL source module for final arithmetic unit.
--
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity arithmetic_unit is
   Port (clk_f,clear_final,start_final,arith_ok_f:in std_logic;
                  sot,sorh:in std_logic_vector(13 downto 0);
                        done_arith_f:out std_logic;
                        moisture:out std_logic_vector(15 downto 0);
                        temp_centi,temp_fah,humidity_percent:out std_logic_vector(18 downto 0));
end arithmetic_unit;

architecture Behavioral of arithmetic_unit is

component final_adder_adder is
port(clk,clr_adder,start_adder,add_or_sub,d_o_n_e_r:in std_logic;
         a_i_n,b_i_n:in std_logic_vector(15 downto 0);
         adder_out:out std_logic_vector(15 downto 0);
         done_adder:out std_logic);
end component final_adder_adder;
component final_divider is
port(divisor_input,dividend_input:in std_logic_vector(15 downto 0);
         clk,clr_divider,st_divider,d_o_n_e_r:in std_logic;
         --countout:out std_logic_vector(4 downto 0);
         d_o_n_e:out std_logic;
         result:out std_logic_vector(15 downto 0));
end component final_divider;
component multiplier_a_mul is
port(clk,rst,start_mul,d_o_n_e_r:in std_logic;
         multiplicand,multiplier: in std_logic_vector(15 downto 0);
         m_result_out: out std_logic_vector(15 downto 0);
         --countout_out:out std_logic_vector(4 downto 0);
         done_mul:out std_logic);
end component multiplier_a_mul;
component final_arithmetic_controller is
   Port (clk,clr_a_c,start_a_c,done_mul,done_adder,done_div,arith_ok:in std_logic;
    rst_mul,start_mul,m_d_o_n_e_r,rst_adder,start_adder,add_or_sub,
                  a_d_o_n_e_r,rst_divider,start_divider,d_d_o_n_e_r,clr_rt,load_rt,
                  clr_rta,load_rta,lrs_rta,clr_rthat,load_rthat,clr_rhu,load_rhu,
                  clr_rt2,load_rt2,sel_rw,clr_rw,load_rw,load_rma,clr_rma,clr_rk1,
                  load_rk1,clr_rk2,load_rk2,clr_rk1kh,load_rk1kh,clr_rmb,load_rmb,
                  clr_r1,load_r1,sel_r1,leftshift_r1,clear_r2,leftshift_r2,load_r2,
                  sel_r2,clr_r3,load_r3,sel_r3_a,sel_r3_b,clr_rk,load_rk,sel_4_1,
                  sel_4_2,sel_4_3,sel_4_4,sel_4_5,sel_4_6,sel_4_7,sel_4_8,sel_2_1,
                  sel_2_2,sel_2_3,sel_2_4,sel_2_5,sel_2_6,sel_2_7,sa1,sa2,sa3,sa4,
                  sel_d_1,sel_d_2,done_arith,clr_rm,ld_rm,sel_m_a_new,clr_rsothat,
                  load_rsothat,clr_rsorhat,load_rsorhat,clr_r_h_o,load_r_h_o,lls_r_h_o,
                  clr_r_h,load_r_h,lrs_r_h,clr_rtc,load_rtc,lls_rtc,clr_rtf,load_rtf,lls_rtf,
                  sel_2_8,sel_2_9,sel_2_10,sa5,sa6,sa7,sa8,sel_m_a_new1,sel_d_3,sel_d_4,
                  sel_4_9,sel_4_10,sel_4_11,sel_4_12,lrs_r2:out std_logic);  --101
end component final_arithmetic_controller;
component multiplex_4_1 is
   Port (in1,in2,in3,in4:in std_logic_vector(15 downto 0);
       sel1,sel2:in std_logic;
```

```
        out1:out std_logic_vector(15 downto 0) );
end component multiplex_4_1;
component multiplex_2_1 is
    Port (in1,in2:in std_logic_vector(15 downto 0);
        sel:in std_logic;
        out1:out std_logic_vector(15 downto 0) );
end component multiplex_2_1;
component r_r_1_mux_21 is
    Port (r1_1,r1_2:in std_logic_vector(15 downto 0);
        clock,clear,ld_r1,selr1,leftshift:in std_logic;
                    r1_output:out std_logic_vector(15 downto 0));
end component r_r_1_mux_21;
component r_r2_mux2x1 is
    Port (clock,clear,left_shift,ld_r2,lrs_r2,sel_r2:in std_logic;
        r2_1,r2_2:in std_logic_vector(15 downto 0);
                    r2_out:out std_logic_vector(15 downto 0));
end component r_r2_mux2x1;
component r_r3_mux2x1 is
    Port (r3_1,r3_2,r3_3:in std_logic_vector(15 downto 0);
        clock,clear,ld_r3,sel_r3_a,sel_r3_b:in std_logic;
                    r3_out:out std_logic_vector(15 downto 0));
end component r_r3_mux2x1;
component r_rw_mux2x1 is
    Port (rw1,rw2:in std_logic_vector(15 downto 0);
        selw,clock,clear,ld_rw:in std_logic;
                    rwout:out std_logic_vector(15 downto 0));
end component r_rw_mux2x1;
component register_rhu is
    Port (clk,clr,load:in std_logic;
        rhu_in:in std_logic_vector(15 downto 0);
                    rhu_out:out std_logic_vector(15 downto 0));
end component register_rhu;
component register_rk is
    Port (clk,clr,load:in std_logic;
        rk_in:in std_logic_vector(15 downto 0);
                    rk_out:out std_logic_vector(15 downto 0));
end component register_rk;
component register_rk1 is
    Port (clk,clr,load:in std_logic;
        rk1_in:in std_logic_vector(15 downto 0);
                    rk1_out:out std_logic_vector(15 downto 0));
end component register_rk1;
component register_rk1kh is
    Port (clk,clr,load:in std_logic;
        rk1kh_in:in std_logic_vector(15 downto 0);
                    rk1kh_out:out std_logic_vector(15 downto 0));
end component register_rk1kh;
component register_rk2 is
    Port (clk,clr,load:in std_logic;
        rk2_in:in std_logic_vector(15 downto 0);
                    rk2_out:out std_logic_vector(15 downto 0));
end component register_rk2;
component register_rma is
    Port (clk,clr,load:in std_logic;
        rma_in:in std_logic_vector(15 downto 0);
                    rma_out:out std_logic_vector(15 downto 0));
```

```
end component register_rma;
component register_rmb is
   Port (clk,clr,load:in std_logic;
        rmb_in:in std_logic_vector(15 downto 0);
                    rmb_out:out std_logic_vector(15 downto 0));
end component register_rmb;
component register_rt is        -- 1 8 12 to 1 8 10 result -> 19 bits -> 18 downto 0
   Port (clk,clr,load:in std_logic;
        rtinput:in std_logic_vector(18 downto 0);
                    rtoutput:out std_logic_vector(18 downto 0));
end component register_rt;
component register_rt2 is
   Port (clk,clr,load:in std_logic;
        rt2_in:in std_logic_vector(15 downto 0);
                    rt2_out:out std_logic_vector(15 downto 0));
end component register_rt2;
component register_rta is
   Port (clk,clr,load,lrs:in std_logic;
        rtainput:in std_logic_vector(18 downto 0);
                    rtaoutput:out std_logic_vector(18 downto 0));
end component register_rta;
component register_rthat is  -- rthat_in may get changed to 9 downto 0;
   Port (clk,clr,load:in std_logic;
        rthat_in:in std_logic_vector(9 downto 0);
                    rthat_out:out std_logic_vector(15 downto 0));
end component register_rthat;
component register_moisture_f is
port(clk,clr,load:in std_logic;
    moisture_in:in std_logic_vector(15 downto 0);
          moisture_out:out std_logic_vector(15 downto 0));
end component register_moisture_f;
--register_r_h is used for r_h_o/16.
component register_r_h is
port(clk,clr,load,lrs:in std_logic;
             r_h_in:in std_logic_vector(18 downto 0);
             r_h_out:out std_logic_vector(15 downto 0));
end component register_r_h;
--register_r_h_o is used to hold the value of real humidity value in xx.xx% format.
component register_r_h_o is
port(      clk,clr,load,lls:in std_logic;
                    r_h_o_in:in std_logic_vector(15 downto 0);
                    r_h_o_out:out std_logic_vector(18 downto 0));
end component register_r_h_o;
--register_rsorhat is at the interface of the arithmetic unit and sensor. It gets SORH
--and gives SORH^.
component register_rsorhat is
port(clr,clk,load:in std_logic;
             sorhat_in:in std_logic_vector(13 downto 0);
             sorhat_out:out std_logic_vector(15 downto 0));
end component register_rsorhat;
--register_rsothat is at the interface of the arithmetic unit and sensor. It gets SOTH
--and gives SOTH^.
component register_rsothat is
port(clr,clk,load:in std_logic;
             sothat_in:in std_logic_vector(13 downto 0);
             sothat_out:out std_logic_vector(15 downto 0));
```

```
end component register_rsothat;
--register_rtc is used to store the value of temperature in degrees celcius. Goes input
-- display unit.
component register_rtc is
port(	clk,clr,load,lls:in std_logic;
			rtc_in:in std_logic_vector(15 downto 0);
			rtc_out:out std_logic_vector(18 downto 0));
end component register_rtc;
--register_rtf is used to store the value of temperature in degrees fahreinheit. Goes
-- to display.
component register_rtf is
port(	clk,clr,load,lls:in std_logic;
			rtf_in:in std_logic_vector(15 downto 0);
			rtf_out:out std_logic_vector(18 downto 0));
end component register_rtf;

signal
done_mul_f,done_adder_f,done_div_f,rst_mul_f,start_mul_f,m_d_o_n_e_r_f,rst_adder_f,start_adder_f,add
_or_sub_f,
		a_d_o_n_e_r_f,rst_divider_f,start_divider_f,d_d_o_n_e_r_f,clr_rt_f,load_rt_f,
		clr_rta_f,load_rta_f,lrs_rta_f,clr_rthat_f,load_rthat_f,clr_rhu_f,load_rhu_f,
		clr_rt2_f,load_rt2_f,sel_rw_f,clr_rw_f,load_rw_f,load_rma_f,clr_rma_f,clr_rk1_f,
		load_rk1_f,clr_rk2_f,load_rk2_f,clr_rk1kh_f,load_rk1kh_f,clr_rmb_f,load_rmb_f,
		clr_r1_f,load_r1_f,sel_r1_f,leftshift_r1_f,clear_r2_f,leftshift_r2_f,load_r2_f,
		sel_r2_f,clr_r3_f,load_r3_f,sel_r3_a_f,sel_r3_b_f,clr_rk_f,load_rk_f,sel_4_1_f,
		sel_4_2_f,sel_4_3_f,sel_4_4_f,sel_4_5_f,sel_4_6_f,sel_4_7_f,sel_4_8_f,sel_2_1_f,
		sel_2_2_f,sel_2_3_f,sel_2_4_f,sel_2_5_f,sel_2_6_f,sel_2_7_f,sa1_f,sa2_f,sa3_f,sa4_f,

sel_d_1_f,sel_d_2_f,clr_rm_f,ld_rm_f,sel_m_a_new_f,clr_rsothat_f,load_rsothat_f,clr_rsorhat_f,
	load_rsorhat_f,clr_r_h_o_f,load_r_h_o_f,lls_r_h_o_f,clr_r_h_f,load_r_h_f,lrs_r_h_f,clr_rtc_f,
	load_rtc_f,lls_rtc_f,clr_rtf_f,load_rtf_f,lls_rtf_f,sel_2_8_f,sel_2_9_f,sel_2_10_f,sa5_f,sa6_f,
	sa7_f,sa8_f,sel_m_a_new1_f,sel_d_3_f,sel_d_4_f,sel_4_9_f,sel_4_10_f,sel_4_11_f,sel_4_12_f,
	lrs_r2_f: std_logic;
signal	adder_b_i_n_f,adder_out_f,adder_ain_final:std_logic_vector(15 downto 0);
signal	divisor_input_f,dividend_input_f,d_result_f:std_logic_vector(15 downto 0);
signal	multiplicand_f,multiplier_f,m_result_out_f: std_logic_vector(15 downto 0);
signal	r1_1_f,r1_2_f:std_logic_vector(15 downto 0);
signal	r1_output_f:std_logic_vector(15 downto 0);
signal	r2_1_f,r2_2_f:std_logic_vector(15 downto 0);
signal	r2_out_f:std_logic_vector(15 downto 0);
signal	r3_1_f,r3_2_f,r3_3_f,r3_out_f:std_logic_vector(15 downto 0);
signal	rw1_f,rw2_f:std_logic_vector(15 downto 0);
signal	rwout_f:std_logic_vector(15 downto 0);
signal	rhu_in_f:std_logic_vector(15 downto 0);
signal	rhu_out_f:std_logic_vector(15 downto 0);
signal	rk_in_f:std_logic_vector(15 downto 0);
signal	rk_out_f:std_logic_vector(15 downto 0);
signal	rk1_in_f,rk1_out_f:std_logic_vector(15 downto 0);
signal	rk2_in_f,rk2_out_f:std_logic_vector(15 downto 0);
signal	rk1kh_in_f,rk1kh_out_f:std_logic_vector(15 downto 0);
signal	rma_in_f, rma_out_f:std_logic_vector(15 downto 0);
signal	rmb_in_f,rmb_out_f:std_logic_vector(15 downto 0);
signal	rtinput_f, rtoutput_f:std_logic_vector(18 downto 0);
signal	rt2_in_f,rt2_out_f:std_logic_vector(15 downto 0);
signal	rtainput_f,rtaoutput_f:std_logic_vector(18 downto 0);
signal	rthat_in_f:std_logic_vector(9 downto 0);
```

```
signal  rthat_out_f:std_logic_vector(15 downto 0);
signal  moisture_in_f:std_logic_vector(15 downto 0);
signal  value_1,value_2,value_3,value_4,value_5,value_6:std_logic_vector(15 downto 0);
signal  value_7,value_8,value_9,value_10,value_11,value_12,value_13:std_logic_vector(15 downto 0);
signal
value_14,value_15,value_16,value_17,value_18,value_19,value_20,value_21,value_22,value_23,value_24:
std_logic_vector(15 downto 0);
signal  m_mux4x1_1_f,m_mux4x1_2_f,m_mux4x1_3_f,m_mux4x1_4_f:std_logic_vector(15 downto 0);
signal
m_mux2x1_1_f,m_mux2x1_2_f,m_mux2x1_3_f,m_mux2x1_4_f,m_mux2x1_5_f:std_logic_vector(15
downto 0);
signal  r_h_in_f:std_logic_vector(18 downto 0);
signal  r_h_out_f:std_logic_vector(15 downto 0);
signal r_h_o_in_f:std_logic_vector(15 downto 0);
signal r_h_o_out_f:std_logic_vector(18 downto 0);
signal sorhat_in_f:std_logic_vector(13 downto 0);
signal sorhat_out_f:std_logic_vector(15 downto 0);
signal sothat_in_f:std_logic_vector(13 downto 0);
signal sothat_out_f:std_logic_vector(15 downto 0);
signal rtc_in_f:std_logic_vector(15 downto 0);
signal rtc_out_f:std_logic_vector(18 downto 0);
signal rtf_in_f:std_logic_vector(15 downto 0);
signal rtf_out_f:std_logic_vector(18 downto 0);
signal
m_mux4x1_5_f,m_mux4x1_6_f,m_mux2x1_6_f,m_mux2x1_7_f,m_mux2x1_9_f:std_logic_vector(15
downto 0);
signal a_mux4x1_1_f,a_mux4x1_2_f,a_mux4x1_3_f,a_mux4x1_4_f:std_logic_vector(15 downto 0);
signal d_mux2x1_1_f,d_mux2x1_2_f:std_logic_vector(15 downto 0);
-- value_1 = 1; value_2 = 0.1994; value_3 = 1.6483, value_4 = 1.1188;
-- value_5 = 1.933; value_6 = 0.1252; value_7 = 1.3859; value_8 = 1.4336;
-- value_9 = 0.5993; value_10 = 1.6675; value_11 = 0.8622; value_12 = 5.4545;value_13 = 1.2642;
begin
--values to be changed to 1 5 10
--value assignments to be done.
value_1  <= "0000010000000000";--'1';value_2  <= "0000000011001100";--'0.1994';
value_3  <= "0000011010010111";--'1.6483';value_4  <= "0000010001111001";--'1.1188';
value_5  <= "0000011110111011";--'1.933';value_6  <= "0000000010000000";--'0.1252';
value_7  <= "0000010110001011";--'1.3859;value_8  <= "0000010110111100";--'1.4336';
value_9  <= "0000001001100101";--'0.5993';value_10 <= "0000011010101011";--'1.6675';
value_11 <= "0000001101110010";--'0.8622';value_12 <= "0001010111010001";--'5.4545';
value_13 <= "0000010100001110";--'1.2642';value_14 <= "0000000001011101";--'0.09175';
value_15 <= "0000010100101111";--'1.296';value_16 <= "0000011100110011";--'1.8';
value_17 <= "0000000010100011";--'0.16';value_18 <= "0000000001010011";--'0.08192';
value_19 <= "0000000010000000";--'0.125';value_20 <= "0000000110010000";--'0.390625';
value_21 <= "0000000000001010";--'0.01';value_22 <= "0000001000000000";--'0.5';
value_23 <= "0000001010000000";--'0.625';value_24 <= "0001100100000000";--'6.25';


AMUX4X1_1:multiplex_4_1 port map (in1 =>value_1, in2 =>r3_out_f, in3 =>value_2in4 =>r1_output_f,
sel1 =>sa1_f , sel2 =>sa2_f, out1=>a_mux4x1_1_f);
AMUX4X1_2:multiplex_4_1 port map (in1 =>r1_output_f, in2 =>r2_out_f, in3 =>rk1kh_out_f, in4
=>rmb_out_f, sel1 =>sa3_f, sel2 =>sa4_f, out1=>a_mux4x1_2_f);

AMUX4X1_3:multiplex_4_1 port map (in1 =>value_13,--1.2642 in2 =>value_19,--0.125in3 =>rt2_out_f,
in4 =>rwout_f, sel1 =>sa5_f, sel2 =>sa6_f, out1=>a_mux4x1_3_f);
```

```
AMUX2X1_1:multiplex_2_1 port map(in1 =>a_mux4x1_1_f, in2 =>a_mux4x1_3_f, sel
=>sel_m_a_new_f, out1=>adder_ain_final);

AMUX4X1_4:multiplex_4_1 port map (in1 =>value_20,--0.390625in2 =>value_21,--0.01in3 =>value_22,-
-0.5in4 =>value_23,--0.625sel1 =>sa7_f, sel2 =>sa8_f, out1=>a_mux4x1_4_f);

AMUX2X1_2:multiplex_2_1 port map(in1 =>a_mux4x1_2_f, in2 =>a_mux4x1_4_f, sel
=>sel_m_a_new1_f, out1=>adder_b_i_n_f);

MMUX4X1_1:multiplex_4_1 port map (in1 => rthat_out_f, in2 => value_3, in3 => value_4, in4 =>
value_5, sel1 =>sel_4_1_f, sel2 => sel_4_2_f, out1=>m_mux4x1_1_f);
MMUX4X1_2:multiplex_4_1 port map (in1 =>value_6, in2 =>value_7, in3 =>rk_out_f, in4
=>r1_output_f, sel1 =>sel_4_3_f, sel2 =>sel_4_4_f, out1=>m_mux4x1_2_f);
MMUX4X1_3:multiplex_4_1 port map (in1 =>rthat_out_f, in2 =>rt2_out_f, in3 =>value_8, in4
=>value_9, sel1 =>sel_4_5_f, sel2 =>sel_4_6_f, out1=>m_mux4x1_3_f);
MMUX4X1_4:multiplex_4_1 port map (in1 =>value_10, in2 =>rhu_out_f, in3 =>rk1_out_f, in4
=>rk1kh_out_f, sel1 =>sel_4_7_f, sel2 =>sel_4_8_f, out1=>m_mux4x1_4_f);
MMUX4X1_5:multiplex_4_1 port map (in1 =>sothat_out_f, in2 =>sorhat_out_f,
in3 =>value_14,--0.09175in4 =>value_15,--1.296sel1 =>sel_4_9_f, sel2 =>sel_4_10_f,
out1=>m_mux4x1_5_f);
MMUX4X1_6:multiplex_4_1 port map (in1 => value_16, in2 =>sorhat_out_f, in3 =>value_17,--0.16,in4
=>r2_out_f, sel1 =>sel_4_11_f, sel2 =>sel_4_12_f, out1=>m_mux4x1_6_f);
MMUX2X1_1:multiplex_2_1 port map(in1 =>rk2_out_f, in2 =>value_11, sel =>sel_2_4_f,
out1=>m_mux2x1_1_f);
MMUX2X1_2:multiplex_2_1 port map(in1 =>m_mux4x1_1_f , in2 =>m_mux4x1_2_f ,
sel =>sel_2_1_f, out1=>m_mux2x1_2_f);
MMUX2X1_3:multiplex_2_1 port map(in1 =>rwout_f, in2 =>rma_out_f, sel =>sel_2_3_f,
out1=>m_mux2x1_3_f);
MMUX2X1_4:multiplex_2_1 port map(in1 =>m_mux4x1_3_f , in2 =>m_mux4x1_4_f , sel =>sel_2_5_f,
out1=>m_mux2x1_4_f);
MMUX2X1_5:multiplex_2_1 port map(in1 =>m_mux2x1_1_f, in2 =>r3_out_f, sel =>sel_2_6_f,
out1=>m_mux2x1_5_f);

MMUX2X1_6:multiplex_2_1 port map(in1 =>m_mux2x1_3_f, in2 =>m_mux2x1_2_f, sel =>sel_2_2_f,
out1=>m_mux2x1_6_f);
MMUX2X1_7:multiplex_2_1 port map(in1 => m_mux2x1_4_f, in2 => m_mux2x1_5_f, sel => sel_2_7_f,
out1=> m_mux2x1_7_f);
MMUX2X1_8:multiplex_2_1 port map(in1 => m_mux2x1_6_f,
in2 => m_mux2x1_9_f sel => sel_2_8_f, out1=> multiplicand_f);

MMUX2X1_9:multiplex_2_1 port map(in1 => m_mux4x1_5_f,
                        in2 => value_18,--0.08192
                        sel => sel_2_9_f,
                        out1=> m_mux2x1_9_f);
MMUX2X1_10:multiplex_2_1 port map(in1 => m_mux2x1_7_f,
                        in2 => m_mux4x1_6_f,
                        sel => sel_2_10_f,
                        out1=> multiplier_f);

DMUX2X1_1:multiplex_2_1 port map(in1 =>value_12,
                        in2 =>r1_output_f,
                        sel =>sel_d_1_f,
                        out1=>d_mux2x1_1_f);

DMUX2X1_2:multiplex_2_1 port map(in1 =>rwout_f,
```

```
                    in2 =>r2_out_f,
                    sel =>sel_d_2_f,
                    out1=>d_mux2x1_2_f);

DMUX2X1_3:multiplex_2_1 port map(in1 =>d_mux2x1_1_f,
                    in2 =>r_h_out_f,
                    sel =>sel_d_3_f,
                    out1=>dividend_input_f);

DMUX2X1_4:multiplex_2_1 port map(in1 =>d_mux2x1_2_f,
                    in2 =>value_24,--6.25
                    sel =>sel_d_4_f,
                    out1=>divisor_input_f);


FAC:final_arithmetic_controller port map(clk=>clk_f, clr_a_c=>clear_final,start_a_c=>start_final,
done_mul=>done_mul_f, done_adder=>done_adder_f, done_div=>done_div_f, arith_ok => arith_ok_f,
rst_mul=>rst_mul_f, start_mul=>start_mul_f, m_d_o_n_e_r=>m_d_o_n_e_r_f, rst_adder=>rst_adder_f,
start_adder=>start_adder_f, add_or_sub=>add_or_sub_f, a_d_o_n_e_r=>a_d_o_n_e_r_f,
rst_divider=>rst_divider_f, start_divider=>start_divider_f, d_d_o_n_e_r=>d_d_o_n_e_r_f,
clr_rt=>clr_rt_f, load_rt=>load_rt_f, clr_rta=>clr_rta_f, load_rta=>load_rta_f, lrs_rta=>lrs_rta_f,
clr_rthat=>clr_rthat_f, load_rthat=>load_rthat_f, clr_rhu=>clr_rhu_f, load_rhu=>load_rhu_f,
clr_rt2=>clr_rt2_f, load_rt2=>load_rt2_f, sel_rw=>sel_rw_f, clr_rw=>clr_rw_f, load_rw=>load_rw_f,
load_rma=>load_rma_f, clr_rma=>clr_rma_f, clr_rk1=>clr_rk1_f, load_rk1=>load_rk1_f,
clr_rk2=>clr_rk2_f, load_rk2=>load_rk2_f, clr_rk1kh=>clr_rk1kh_f, load_rk1kh=>load_rk1kh_f,
clr_rmb=>clr_rmb_f, load_rmb=>load_rmb_f, clr_r1=>clr_r1_f, load_r1=>load_r1_f, sel_r1=>sel_r1_f,
leftshift_r1=>leftshift_r1_f, clear_r2=>clear_r2_f, leftshift_r2=>leftshift_r2_f, load_r2=>load_r2_f,
sel_r2=>sel_r2_f, clr_r3=>clr_r3_f, load_r3=>load_r3_f, sel_r3_a=>sel_r3_a_f, sel_r3_b=>sel_r3_b_f,
clr_rk=>clr_rk_f, load_rk=>load_rk_f, sel_4_1=>sel_4_1_f, sel_4_2=>sel_4_2_f, sel_4_3=>sel_4_3_f,
sel_4_4=>sel_4_4_f, sel_4_5=>sel_4_5_f, sel_4_6=>sel_4_6_f, sel_4_7=>sel_4_7_f, sel_4_8=>sel_4_8_f,
sel_2_1=>sel_2_1_f, sel_2_2=>sel_2_2_f, sel_2_3=>sel_2_3_f, sel_2_4=>sel_2_4_f, sel_2_5=>sel_2_5_f,
sel_2_6=>sel_2_6_f, sel_2_7=>sel_2_7_f, sa1=>sa1_f, sa2=>sa2_f, sa3=>sa3_f, sa4=>sa4_f,
sel_d_1=>sel_d_1_f, sel_d_2=>sel_d_2_f, done_arith => done_arith_f, clr_rm =>clr_rm_f, ld_rm
=>ld_rm_f, sel_m_a_new => sel_m_a_new_f, clr_rsothat => clr_rsothat_f, load_rsothat => load_rsothat_f,
clr_rsorhat => clr_rsorhat_f, load_rsorhat => load_rsorhat_f, clr_r_h_o => clr_r_h_o_f, load_r_h_o =>
load_r_h_o_f, lls_r_h_o =>lls_r_h_o_f, clr_r_h => clr_r_h_f, load_r_h => load_r_h_f, lrs_r_h =>lrs_r_h_f,
clr_rtc => clr_rtc_f, load_rtc => load_rtc_f, lls_rtc => lls_rtc_f, clr_rtf => clr_rtf_f, load_rtf=>load_rtf_f,
lls_rtf => lls_rtf_f, sel_2_8 =>sel_2_8_f, sel_2_9 =>sel_2_9_f, sel_2_10 =>sel_2_10_f, sa5 => sa5_f, sa6
=> sa6_f, sa7 => sa7_f, sa8 => sa8_f, sel_m_a_new1 =>sel_m_a_new1_f, sel_d_3 => sel_d_3_f, sel_d_4
=>sel_d_4_f, sel_4_9 => sel_4_9_f, sel_4_10 => sel_4_10_f, sel_4_11 =>sel_4_11_f, sel_4_12 =>
sel_4_12_f, lrs_r2 => lrs_r2_f);


FADD:final_adder_adder port map(clk=>clk_f,
                    clr_adder=>rst_adder_f,
                                                start_adder=>start_adder_f,
                                                add_or_sub=>add_or_sub_f,
                                                d_o_n_e_r=>a_d_o_n_e_r_f,
                        a_i_n=>adder_ain_final,
                                                b_i_n=>adder_b_i_n_f,
                        adder_out=>adder_out_f,
                        done_adder=>done_adder_f);
FDIV:final_divider port map(divisor_input=>divisor_input_f,
                    dividend_input=>dividend_input_f,
                        clk=>clk_f,
                                            clr_divider=>rst_divider_f,
```

```
                                        st_divider=>start_divider_f,
                                        d_o_n_e_r=>d_d_o_n_e_r_f,
                    --countout=>dcount,
                    d_o_n_e=>done_div_f,
                                        -- overflow_out=>doverflow,
                                        --loadoverflow_out=>ldoverflow,
                    result=>d_result_f);
FMUL:multiplier_a_mul port map(clk=>clk_f,
                    rst=>rst_mul_f,
                                                start_mul=>start_mul_f,
                                                d_o_n_e_r=>m_d_o_n_e_r_f,
                    multiplicand=>multiplicand_f,
                                                multiplier=>multiplier_f,
                    m_result_out=>m_result_out_f,
--                  countout_out=>mcount,
                    done_mul=>done_mul_f);
R1F:r_r_1_mux_21 port map (r1_1=>r1_1_f,
                    r1_2=>r1_2_f,
                    clock=>clk_f,
                                        clear=>clr_r1_f,
                                        ld_r1=>load_r1_f,
                                        selr1=>sel_r1_f,
                                        leftshift=>leftshift_r1_f,
                                        r1_output=>r1_output_f);
r1_1_f <= m_result_out_f;
r1_2_f <= adder_out_f;
R2F:r_r2_mux2x1 port map (clock=>clk_f,
                    clear=>clear_r2_f,
                                        left_shift=>leftshift_r2_f,
                                        ld_r2=>load_r2_f,
                                        lrs_r2=>lrs_r2_f,
                                        sel_r2=>sel_r2_f,
                    r2_1=>r2_1_f,
                                        r2_2=>r2_2_f,
                                r2_out=>r2_out_f);
r2_1_f <= m_result_out_f;
r2_2_f <= adder_out_f;
R3F:r_r3_mux2x1 port map (r3_1=> r3_1_f,
                    r3_2=>r3_2_f,
                                        r3_3=>r3_3_f,
                    clock=>clk_f,
                                        clear=>clr_r3_f,
                                        ld_r3=>load_r3_f,
                                        sel_r3_a=>sel_r3_a_f,
                                        sel_r3_b=>sel_r3_b_f,
                                r3_out=>r3_out_f);
r3_1_f <= adder_out_f;
r3_2_f <= m_result_out_f;
r3_3_f <= d_result_f;

RWF:r_rw_mux2x1 port map(rw1=>rw1_f,
                    rw2=>rw2_f,
                    selw=>sel_rw_f,
                                        clock=>clk_f,
                                        clear=>clr_rw_f,
                                        ld_rw=>load_rw_f,
```

```
                                rwout=>rwout_f);
rw1_f <= adder_out_f;
rw2_f <= m_result_out_f;
RHUF:register_rhu port map(clk=>clk_f,
                clr=>clr_rhu_f,
                                                load=>load_rhu_f,
                rhu_in=>rhu_in_f,
                                rhu_out=>rhu_out_f);
rhu_in_f <=d_result_f;
RKF:register_rk port map(clk=>clk_f,
                clr=>clr_rk_f,
                                                load=>load_rk_f,
                rk_in=>rk_in_f,
                                rk_out=>rk_out_f);
rk_in_f <= adder_out_f;
RK1F:register_rk1 port map(clk=>clk_f,
                 clr=>clr_rk1_f,
                                                 load=>load_rk1_f,
                 rk1_in=>rk1_in_f,
                                 rk1_out=>rk1_out_f);
rk1_in_f <= adder_out_f;
RK1KHF:register_rk1kh port map(clk=>clk_f,
                    clr=>clr_rk1kh_f,
                                                              load=>load_rk1kh_f,
                    rk1kh_in=>rk1kh_in_f,
                                    rk1kh_out=>rk1kh_out_f);
rk1kh_in_f <= m_result_out_f;
RK2F:register_rk2 port map(clk=>clk_f,
                 clr=>clr_rk2_f,
                                                 load=>load_rk2_f,
                 rk2_in=>rk2_in_f,
                                 rk2_out=>rk2_out_f);
rk2_in_f <= adder_out_f;
RMAF:register_rma port map(clk=>clk_f,
                clr=>clr_rma_f,
                                                load=>load_rma_f,
                rma_in=>rma_in_f,
                                 rma_out=>rma_out_f);
rma_in_f <= d_result_f;
RMBF:register_rmb port map(clk=>clk_f,
                clr=>clr_rmb_f,
                                                load=>load_rmb_f,
                rmb_in=>rmb_in_f,
                                                rmb_out=>rmb_out_f);
rmb_in_f <= d_result_f;
RTF:register_rt port map(clk=>clk_f,
                clr=>clr_rt_f,
                                                load=>load_rt_f,
                rtinput=>rtinput_f,
                                rtoutput=>rtoutput_f);
rtinput_f <= rtf_out_f;
RT2F:register_rt2 port map(clk=>clk_f,
                 clr=>clr_rt2_f,
                                                 load=>load_rt2_f,
                 rt2_in=>rt2_in_f,
                                 rt2_out=>rt2_out_f);
```

```
rt2_in_f <= m_result_out_f;
RTAF:register_rta port map(clk=>clk_f,
                    clr=>clr_rta_f,
                                                load=>load_rta_f,
                                                lrs=>lrs_rta_f,
                    rtainput=>rtainput_f,
                                rtaoutput=>rtaoutput_f);
rtainput_f  <= rtoutput_f;
RTHATF:register_rthat port map(clk=>clk_f,
                      clr=>clr_rthat_f,
                                                          load=>load_rthat_f,
                      rthat_in=>rthat_in_f,
                                  rthat_out=>rthat_out_f);
rthat_in_f <= rtaoutput_f(9 downto 0);
RMOIS:register_moisture_f port map(clk=>clk_f,
                         clr=>clr_rm_f,
                                                                  load=>ld_rm_f,
                         moisture_in=>moisture_in_f,
                               moisture_out=>moisture);
moisture_in_f <= r3_out_f;


R_R_H:register_r_h port map(clk =>clk_f,
                     clr => clr_r_h_f,
                                                                        load =>load_r_h_f,
                                                                        lrs =>lrs_r_h_f,

                            r_h_in =>r_h_in_f ,
                                                                        r_h_out =>r_h_out_f);
r_h_in_f <= r_h_o_out_f;

R_R_H_O:register_r_h_o port map(clk => clk_f,
                          clr =>clr_r_h_o_f,

load=>load_r_h_o_f,

lls=>lls_r_h_o_f,
                                        r_h_o_in =>r_h_o_in_f ,
                                        r_h_o_out=>r_h_o_out_f);

r_h_o_in_f <= rwout_f;
humidity_percent <= r_h_o_out_f;
R_RSORHAT:register_rsorhat port map(clr =>clr_rsorhat_f,
                           clk =>clk_f,

          load =>load_rsorhat_f,
                                sorhat_in=>sorhat_in_f,
                                sorhat_out=>sorhat_out_f);

sorhat_in_f <= sorh;
R_RSOTHAT:register_rsothat port map(clr=> clr_rsothat_f,
                           clk=>clk_f,

          load=>load_rsothat_f,
                                sothat_in=>sothat_in_f,
                                sothat_out=>sothat_out_f);
```

```
sothat_in_f <= sot;

R_RTC:register_rtc port map  (clk =>clk_f,
                        clr =>clr_rtc_f,

            load=>load_rtc_f,
                                                                                lls=>lls_rtc_f,
                                        rtc_in =>rtc_in_f,
                                        rtc_out=>rtc_out_f);
rtc_in_f <= r1_output_f;
temp_centi <= rtc_out_f;
--xxxxxxxxxxxxxxxxxxxxxxxxxx
load_temp_out <= load_rtc_f;
--xxxxxxxxxxxxxxxxxxxxxxxxxx
R_RTF:register_rtf port map(clk => clk_f,
                        clr=>clr_rtf_f,
                                                                    load=>load_rtf_f,
                                                                    lls=>lls_rtf_f,
                                        rtf_in =>rtf_in_f,
                                        rtf_out =>rtf_out_f);
temp_fah            <= rtf_out_f;
rtf_in_f <=  r3_out_f;


--adder_outf <=adder_out_f ;
--divider_outf<=d_result_f;
--multiplier_outf<=m_result_out_f;
--r1outf <=r1_output_f;
--r2outf<=r2_out_f;
--r3outf<=r3_out_f;
--rwoutf<=rwout_f;
--rhuoutf<=rhu_out_f;
--rkoutf<=rk_out_f;
--rk1outf<=rk1_out_f;
--rk1khoutf<=rk1kh_out_f;
--rk2outf<=rk2_out_f;
--rmaoutf<=rma_out_f;
--rmboutf<=rmb_out_f;
--rtoutf<=rtoutput_f;
--rt2outf<=rt2_out_f;
--rtaoutf<=rtaoutput_f;
--rthatoutf<=rthat_out_f;
--multiplicand_inf<=multiplicand_f;
--multiplier_inf<=multiplier_f;
--dividend_inf<=dividend_input_f;
--divisor_inf<=divisor_input_f;
--addera_inf<=adder_a_i_n_f;
--adderb_inf<=adder_b_i_n_f;

end Behavioral;

--in process of changing to 16-bits.
--VHDL source module for twos complement ripple carry 21 bit adder/subtractor.
--twos complement overflow is checked using 'xor' of two MSBs of
--carry signal. Overflow has a chance of occurence only when either
--both numbers are positive or both are negative.
```

```
--When two positive numbers are added and an overflow occurs,
--the result is replaced with the maximum positive value that
-- can be represented by the considered representation.
--That is "011.....1". Similarly when two negative numbers are
--added and cause an overflow the result is replaced with the
--least negative value that can be represented.
--That is "100.....0". The sign bit of registerA is checked
--to determine which case of overflow has occured and result is
--replaced accordingly.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity final_adder_adder is
port(clk,clr_adder,start_adder,add_or_sub,d_o_n_e_r:in std_logic;
        a_i_n,b_i_n:in std_logic_vector(15 downto 0);
        adder_out:out std_logic_vector(15 downto 0);
        done_adder:out std_logic);

end final_adder_adder;

architecture Behavioral of final_adder_adder is

component add_sub_adder is
port(a_input,b_input: in std_logic_vector(15 downto 0);
        sel_cin,sel_bin:in std_logic;
        adder_out: out std_logic_vector(15 downto 0);
        overflow:out std_logic);
end component add_sub_adder;

component add_sub_controller is
port(clk,clear,done_r,start,add_sub,sign_a,over_flow:in std_logic;
        clr_a,load_a,clr_b,load_b,clr_c,load_c,sel_mux_1,sel_mux_2,
        done_addition,sel_bin,sel_cin,load_add_sub,clr_add_sub: out std_logic);
end component add_sub_controller;

component mux_2x1_1_adder is
port(in1,in2:in std_logic_vector(15 downto 0);
        sel:in std_logic;
        out1:out std_logic_vector(15 downto 0));
end component mux_2x1_1_adder;

component register_a_adder is
port(clr,clk,load:in std_logic;
        a_in:in std_logic_vector(15 downto 0);
        a_out:out std_logic_vector(15 downto 0);
        sign_a:out std_logic);
end component register_a_adder;

component register_b_adder is
port(clr,clk,load:in std_logic;
        b_in:in std_logic_vector(15 downto 0);
        b_out:out std_logic_vector(15 downto 0));
end component register_b_adder;
```

```
component register_c_adder is
port(clr,clk,load:in std_logic;
        c_in:in std_logic_vector(15 downto 0);
        c_out:out std_logic_vector(15 downto 0));
end component register_c_adder;

component reg_add_sub_adder is
port(clr,clk,load:in std_logic;
        add_sub_in:in std_logic_vector(15 downto 0);
        add_sub_out:out std_logic_vector(15 downto 0));
end  component reg_add_sub_adder;

signal  s_i_g_n_a,o_v_e_r_f_l_o_w,c_l_r_a,l_d_a,c_l_r_b,l_d_b,c_l_r_c:std_logic;
signal  l_d_c,selmux_1,selmux_2,selbin,selcin:std_logic;
signal
rega_out,regb_out,out_adder,over_input_1,over_input_2,mux2_in,regc_in,addsub_out:std_logic_vector(15
downto 0);
signal ld_add_sub,clear_add_sub:std_logic;
begin
over_input_1 <= "1000000000000000";
over_input_2 <= "0111111111111111";

ADDSUBCONTROL:add_sub_controller port map (clk=>clk,
                        clear=>clr_adder,
                                                done_r=>d_o_n_e_r,
                                                start=>start_adder,
                                                add_sub=>add_or_sub,
                                                sign_a=>s_i_g_n_a,
                                                over_flow=>o_v_e_r_f_l_o_w,
                        clr_a=>c_l_r_a,
                                                load_a=>l_d_a,
                                                clr_b=>c_l_r_b,
                                                load_b=>l_d_b,
                                                clr_c=>c_l_r_c,
                                                load_c=>l_d_c,
                                                sel_mux_1=>selmux_1,
                                                sel_mux_2=>selmux_2,
                        done_addition=>done_adder,
                                                sel_bin=>selbin,
                                                sel_cin=>selcin,
                                                load_add_sub=>ld_add_sub,
                                                clr_add_sub=>clear_add_sub);


ADDSUB:add_sub_adder port map(a_input=>rega_out,
                b_input=>regb_out,
                        sel_cin=>selcin,
                                        sel_bin=>selbin,
                        adder_out=>out_adder,
                        overflow=>o_v_e_r_f_l_o_w);

MUX1:mux_2x1_1_adder port map(in1=>addsub_out,
                in2=>over_input_1,
                        sel=>selmux_1,
                                        out1=>mux2_in);
```

```
MUX2:mux_2x1_1_adder port map(in1=>mux2_in,
                in2=>over_input_2,
                        sel=>selmux_2,
                                        out1=>regc_in);

REGA:register_a_adder port map(clr =>c_l_r_a,
                clk =>clk,
                                                load=>l_d_a,
                        a_in=>a_i_n,
                        a_out=>rega_out,
                                                sign_a=>s_i_g_n_a);
REGB:register_b_adder port map(clr=>c_l_r_b,
                clk=>clk,
                                                load=>l_d_b,
                        b_in=>b_i_n,
                        b_out=>regb_out);

REGC:register_c_adder port map(clr=>c_l_r_c,
                clk=>clk,
                                                load =>l_d_c,
                        c_in =>regc_in,
                        c_out=>adder_out);

REGADDSUB:reg_add_sub_adder port map(clr=>clear_add_sub,
                        clk=>clk,
                                                                load=>ld_add_sub,
                                add_sub_in=>out_adder,
                                add_sub_out=>addsub_out);

end Behavioral;

-- VHDL source module used for n-bit ripple carry adder/subtractor

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity add_sub_adder is
--generic(n:positive:=32);
port(a_input,b_input: in std_logic_vector(15 downto 0);
        sel_cin,sel_bin:in std_logic;
        adder_out: out std_logic_vector(15 downto 0);
        overflow:out std_logic);
end add_sub_adder;

architecture structural of add_sub_adder is
component bitadder_adder is          -- component declaration of onebit adder
port(a_in,b_in,sel_b,c_in:in std_logic;
        sum_bfa, cout:out std_logic);
end component bitadder_adder;

component mux_2X1a_adder is      --  component declaration of 2X1 multiplexor
```

```
port(a,b:in std_logic;
        sel : in std_logic;
        c: out std_logic);
end component mux_2X1a_adder;
signal c_in_first: std_logic;
signal carry_internal: std_logic_vector(15 downto 0);
begin

A0: mux_2X1a_adder port map(a=>'0',b=>'1',sel=>sel_cin,c=>c_in_first);
B0:bitadder_adder port map(a_in=>a_input(0),b_in=>
b_input(0),sel_b=>sel_bin,c_in=>c_in_first,sum_bfa=>adder_out(0),cout => carry_internal(0));

B1TON: for i in 1 to 15 generate

        Bi: bitadder_adder port map(a_in=>a_input(i),b_in=>
b_input(i),sel_b=>sel_bin,c_in=>carry_internal(i-1),sum_bfa=>adder_out(i),cout => carry_internal(i));

        end generate;
--2's complement overflow occurs when the carry into the MSB is not equal to
--the carry out from the MSB. This is detected using XOR gate.
overflow <= carry_internal(15) xor carry_internal(14);
end structural;

--VHDL source code of 1 bit adder/subtractor.


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity bitadder_adder is
port(a_in,b_in,sel_b,c_in:in std_logic;
        sum_bfa, cout:out std_logic);
end bitadder_adder;

architecture Behavioral of bitadder_adder is
component mux_2X1a_adder is
port(a,b:in std_logic;
        sel : in std_logic;
        c: out std_logic);
end component mux_2X1a_adder;

component bfa_adder is
port(a,b,cin: in std_logic;
        sum, carryout: out std_logic);
end component bfa_adder;
signal bout,b_in_bar:std_logic;
begin
b_in_bar <= not(b_in);

M0: mux_2X1a_adder port map(a=>b_in, b=>b_in_bar,sel=>sel_b,c=>bout);
--M1: mux_2X1_a port map(a=>'0',b=>'1',sel=>add_sub,c=>cin);            \
-- If sel_b is '0' then b_in is selected else not(b_in) is selected.
B0: bfa_adder port map(a=>a_in,b=>bout,cin=>c_in,sum=>sum_bfa,carryout=>cout);
```

```
end Behavioral;

-- VHDL source module of binary full adder used in adder/subtractor.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity bfa_adder is
port(a,b,cin: in std_logic;
            sum, carryout: out std_logic);
end bfa_adder;

architecture Behavioral of bfa_adder is
signal input:std_logic_vector(2 downto 0);
begin
input <= a&b&cin;
 process(a,b,cin,input) is
 begin
if (input = "000") then sum <= '0';carryout <= '0';
elsif (input = "001") then sum <= '1';carryout <= '0';
elsif (input = "010") then sum <= '1';carryout <= '0';
elsif (input = "011") then sum <= '0';carryout <= '1';
elsif (input = "100") then sum <= '1';carryout <= '0';
elsif (input = "101") then sum <= '0';carryout <= '1';
elsif (input = "110") then sum <= '0';carryout <= '1';
else  sum <= '1';carryout <= '1';
end if;

 end process;


end Behavioral;

-- VHDL code for n-bit input 2X1 multiplexor used in the design of Booth's multiplier.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity mux_2X1a_adder is
port(a,b:in std_logic;
            sel : in std_logic;
            c: out std_logic);
end mux_2X1a_adder;

architecture Behavioral of mux_2X1a_adder is

begin
process(sel,a,b) is
begin
if (sel = '0') then
```

```
        c <= a;
else
        c <= b;
end if;
end process;

end Behavioral;

--VHDL source module for controller used in adder/subtractor.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity add_sub_controller is
port(clk,clear,done_r,start,add_sub,sign_a,over_flow:in std_logic;
        clr_a,load_a,clr_b,load_b,clr_c,load_c,sel_mux_1,sel_mux_2,
        done_addition,sel_bin,sel_cin,load_add_sub,clr_add_sub: out std_logic);
end add_sub_controller;

architecture Behavioral of add_sub_controller is

component dff_add_sub is
port(clk,clr,din:in std_logic;
        dout:out std_logic);
end component dff_add_sub;

signal debouncer_1,debouncer_2,start_ctrl:std_logic;
signal t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,sa,sb,sc,sd,se,sf,sg,sh,si:std_logic;
begin

DEBOUNCER0: dff_add_sub port map(clk=>clk,clr=>clear,din=> start,dout=>debouncer_1);
DEBOUNCER1: dff_add_sub port map(clk=>clk,clr=>clear,din=> debouncer_1,dout=>debouncer_2);
start_ctrl <= (debouncer_1 and (not(debouncer_2))) ;

D0:dff_add_sub port map(clk=>clk,clr=>clear,din=>start_ctrl,dout=>t0);
D1:dff_add_sub port map(clk=>clk,clr=>clear,din=>t0,dout=>t1);
D2:dff_add_sub port map(clk=>clk,clr=>clear,din=>sa,dout=>t2);
D3:dff_add_sub port map(clk=>clk,clr=>clear,din=>sb,dout=>t3);
D4:dff_add_sub port map(clk=>clk,clr=>clear,din=>sc,dout=>t4);
D5:dff_add_sub port map(clk=>clk,clr=>clear,din=>sd,dout=>t5);
D6:dff_add_sub port map(clk=>clk,clr=>clear,din=>se,dout=>t6);
D7:dff_add_sub port map(clk=>clk,clr=>clear,din=>sf,dout=>t7);
D8:dff_add_sub port map(clk=>clk,clr=>clear,din=>sg,dout=>t8);
D9:dff_add_sub port map(clk=>clk,clr=>clear,din=>sh,dout=>t9);
D10:dff_add_sub port map(clk=>clk,clr=>clear,din=>si,dout=>t10);

--control equations.
sa <= t1 and add_sub;
sb <= t1 and (not(add_sub));
sc <= t2 or t3;
sd <= t4 and (not(over_flow));
se <= t4 and over_flow;
sf <= t6 and (not(sign_a));
```

```
sg <= t6 and sign_a;
sh <= t5 or t7 or t8 or ((not(done_r)) and t9);
si <= t9 and done_r;

--Output equations.
clr_a <= t10;
load_a <= t0;
clr_b <=  t10;
load_b <= t0;
clr_c <= t0;
load_c <= t5 or t7 or t8;
sel_mux_1 <= t8;
sel_mux_2 <= t7;
done_addition <= t9;
sel_bin <= t2;
sel_cin <= t2;
load_add_sub<= t2 or t3;
clr_add_sub <= t0;
end Behavioral;

--VHDL source module for D flip flop used in add_sub module..
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity dff_add_sub is
port(clk,clr,din:in std_logic;
        dout:out std_logic);
end dff_add_sub;

architecture Behavioral of dff_add_sub is
signal dout_sig:std_logic;
begin
process(clr,clk) is
begin
if(clr='1')then
        dout_sig <= '0';
elsif(clk'event and clk='1')then
        dout_sig <= din;
else
        dout_sig <= dout_sig;
end if;
end process;
dout <= dout_sig;
end Behavioral;

--VHDL source module for first multiplexor used in selection of
--registerC inputs.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity mux_2x1_1_adder is
port(in1,in2:in std_logic_vector(15 downto 0);
        sel:in std_logic;
        out1:out std_logic_vector(15 downto 0));
end mux_2x1_1_adder;

architecture Behavioral of mux_2x1_1_adder is

begin
process(in1,in2,sel)is
begin
if(sel = '0')then
        out1 <= in1;
else
        out1 <= in2;
end if;
end process;
end Behavioral;

--VHDL source module for register add_sub used in adder_subtractor.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity reg_add_sub_adder is
port(clr,clk,load:in std_logic;
        add_sub_in:in std_logic_vector(15 downto 0);
        add_sub_out:out std_logic_vector(15 downto 0));
end reg_add_sub_adder;

architecture Behavioral of reg_add_sub_adder is
signal add_sub_out_sig:std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1')then
        add_sub_out_sig <= (others=>'0');
elsif(clk'event and clk='1')then
        if(load ='1')then
                add_sub_out_sig <= add_sub_in;
    else
                add_sub_out_sig <= add_sub_out_sig;
        end if;
else
        add_sub_out_sig <= add_sub_out_sig;
end if;
end process;
add_sub_out <= add_sub_out_sig;
end Behavioral;

--VHDL source module for register A used in adder_subtractor.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_a_adder is
port(clr,clk,load:in std_logic;
        a_in:in std_logic_vector(15 downto 0);
        a_out:out std_logic_vector(15 downto 0);
        sign_a:out std_logic);
end register_a_adder;

architecture Behavioral of register_a_adder is
signal a_out_sig:std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1')then
        a_out_sig <= (others=>'0');
elsif(clk'event and clk='1')then
        if(load ='1')then
                a_out_sig <= a_in;
    else
                a_out_sig <= a_out_sig;
        end if;
else
        a_out_sig <= a_out_sig;
end if;
end process;
a_out <= a_out_sig;
sign_a<= a_out_sig(15);
end Behavioral;

--VHDL source module for register A used in adder_subtractor.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_b_adder is
port(clr,clk,load:in std_logic;
        b_in:in std_logic_vector(15 downto 0);
        b_out:out std_logic_vector(15 downto 0));
end register_b_adder;

architecture Behavioral of register_b_adder is
signal b_out_sig:std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1')then
        b_out_sig <= (others=>'0');
elsif(clk'event and clk='1')then
        if(load ='1')then
                b_out_sig <= b_in;
    else
```

```
                b_out_sig <= b_out_sig;
        end if;
else
        b_out_sig <= b_out_sig;
end if;
end process;
b_out <= b_out_sig;
end Behavioral;

--VHDL source module for register A used in adder_subtractor.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_c_adder is
port(clr,clk,load:in std_logic;
        c_in:in std_logic_vector(15 downto 0);
        c_out:out std_logic_vector(15 downto 0));
end register_c_adder;

architecture Behavioral of register_c_adder is
signal c_out_sig:std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1')then
        c_out_sig <= (others=>'0');
elsif(clk'event and clk='1')then
        if(load ='1')then
                c_out_sig <= c_in;
    else
                c_out_sig <= c_out_sig;
        end if;
else
        c_out_sig <= c_out_sig;
end if;
end process;
c_out <= c_out_sig;
end Behavioral;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity final_divider is
port(divisor_input,dividend_input:in std_logic_vector(15 downto 0);
        clk,clr_divider,st_divider,d_o_n_e_r:in std_logic;
        --countout:out std_logic_vector(4 downto 0);
        d_o_n_e:out std_logic;
        result:out std_logic_vector(15 downto 0));
```

```
end final_divider;

architecture Behavioral of final_divider is
--Component declaration starts here.
--'conver' converts a twos complement number to an unsigned number,
-- giving it sign bit also as output.
component conver is
port(a_in: in std_logic_vector(15 downto 0);
        bout: out std_logic_vector(14 downto 0));
        --sign:out std_logic);
end component conver;
--'conver2' converts an unsigned number to twos complement format
-- considering the sign.
component conver2 is
port(a_in: in std_logic_vector(14 downto 0);
        bout: out std_logic_vector(15 downto 0);
        sign:in std_logic);
end component conver2;
--'divider' divides the dividend by divisor.
component divider is
port(clk,clear, start_all: in std_logic;
        divisor_in:in std_logic_vector(14 downto 0);
        dividend_in:in std_logic_vector(14 downto 0);
        done_a, over_flow:out std_logic;
        --counterout:out std_logic_vector(4 downto 0);
        --carryout_arith:out std_logic;
        --l_l_s_out,zero_out,ctrl_count_out:out std_logic;
        division_out:out std_logic_vector(14 downto 0);
        quotient_value:out std_logic);
end component divider;
--'final_divider_controller'controls all the individual components
-- using divider,conver,conver2,final_register,quotient_register.
component final_divider_controller is
port(clk,clear_final_controller,start,over_flow,done_division,q_or_out,done_r:in std_logic;
        load_dividend,load_divisor,start_ctrl0,ld_o_flow,ld_result,ld_q_or_out,
        done_final,clr_quotient,clr_final,clear_divider,load_quotient:out std_logic);
end component final_divider_controller;

--'final_register'stores the value of final division output.
component final_register is
port(clk,clr,load_result:in std_logic;
        result_in:in std_logic_vector(15 downto 0);
        result_out:out std_logic_vector(15 downto 0));
end component final_register;

--'quotient_register'stores the value of quotient from 'divider' and
-- produces the desired result incase of overflow or zero output.
component quotient_register is
port(quotient_in:in std_logic_vector(14 downto 0);
        quotient_out:out std_logic_vector(14 downto 0);
        clk,clr,load_overflow,load_q_zero,load_quotient:in std_logic);
end component quotient_register;
--'register_divisor'stores the value of divisor input and produces output
-- msb_divisor also to calculate the sign of the result.
component register_divisor is
port(clk,load:in std_logic;
```

```
        divisin: in std_logic_vector(15 downto 0);
        divisout: out std_logic_vector(15 downto 0);
        msb_divisor: out std_logic);
end component register_divisor;
--'register_dividend'stores the value of dividend input and produces output
-- msb_dividend also to calculate the sign of the result.
component register_dividend is
port(clk,load:in std_logic;
        divin: in std_logic_vector(15 downto 0);
        divout: out std_logic_vector(15 downto 0);
        msb_dividend: out std_logic);
end component register_dividend;

--signals description.
--l_d_d_i_v_i_s_o_r -> load signal from controller to load the register_divisor.
--d_i_v_i_s_o_r -> divisor output to 'conver'.
--m_s_b_d_i_v_i_s_o_r->msb of divisor to calculate sign bit of result.
--l_d_d_i_v_i_d_e_n_d -> load signal from controller to load the register_dividend.
--d_i_v_d_e_n_d -> dividend output to 'conver'.
--m_s_b_d_i_v_i_d_e_n_d->msb of dividend to calculate sign bit of result.
--cleardivider->clear input for divider;start_divider->start input for divider.
--donedivision->done signal to controller,overflow->overflow signal to controller.
--quot_in-> input for quotient register; quot_value-> output from divider to controller.
-- loadquotient->load signal to load quotient to quotient register.
--resultin->input to final register;s_i_g_n->sign bit of the result.
--loadoverflow,loadqzero->inputs to quotient register,
--clearfinal,loadresult->inputs to final register.
signal l_d_d_i_v_i_s_o_r,
m_s_b_d_i_v_i_s_o_r,l_d_d_i_v_i_d_e_n_d,m_s_b_d_i_v_i_d_e_n_d:std_logic;
signal d_i_v_i_s_o_r, d_i_v_i_d_e_n_d,resultin:std_logic_vector(15 downto 0);
signal d_i_v_i_s_o_r_i_n, d_i_v_i_d_e_n_d_i_n:std_logic_vector(14 downto 0);
signal cleardivider,start_divider,donedivision,quot_value:std_logic;
signal quot_in,conver2_in:std_logic_vector(14 downto 0);
signal clear_quot,loadquotient,s_i_g_n:std_logic;
signal loadoverflow,loadresult,loadqzero,clearfinal,overflow:std_logic;
begin

s_i_g_n <= m_s_b_d_i_v_i_s_o_r xor m_s_b_d_i_v_i_d_e_n_d;

DIVISOR0: register_divisor port map(clk=>clk,
                                    load=>l_d_d_i_v_i_s_o_r,
                                    divisin=>divisor_input,
                                    divisout=>d_i_v_i_s_o_r,
                                    msb_divisor=>m_s_b_d_i_v_i_s_o_r);
DIVIDEND0:register_dividend port map(clk=>clk,
                                    load=>l_d_d_i_v_i_d_e_n_d,
                                    divin=>dividend_input,
                                    divout=>d_i_v_i_d_e_n_d,
                                    msb_dividend=>m_s_b_d_i_v_i_d_e_n_d);
CONVERA:conver port map (a_in=>d_i_v_i_s_o_r,
                        bout=>d_i_v_i_s_o_r_i_n);

CONVERB:conver port map (a_in=>d_i_v_i_d_e_n_d,
                        bout=>d_i_v_i_d_e_n_d_i_n);
DIVIDER0:divider port MAP(clk => clk,
                    clear =>cleardivider,
```

```
                                        start_all=>start_divider,
                                        divisor_in=>d_i_v_i_s_o_r_i_n,
                                        dividend_in=>d_i_v_i_d_e_n_d_i_n,
                                        done_a=>donedivision,
                                        over_flow=>overflow,
                                --      counterout=>countout,
                                        division_out=>quot_in,
                                        quotient_value=>quot_value);
--overflow_out <= overflow;
--loadoverflow_out <= loadoverflow;
QUOTIENTREGISTER:quotient_register port map(quotient_in=>quot_in,
                        quotient_out=>conver2_in,
                                                        clk=>clk,
                                                        clr=>clear_quot,
                                                        load_overflow=>loadoverflow,
                                                        load_q_zero=>loadqzero,
                                                        load_quotient=>loadquotient);

CONVER2A:conver2 port map(a_in => conver2_in,
                bout=>resultin,
                                        sign=>s_i_g_n);

FDC:final_divider_controller port map (clk=>clk,
                        clear_final_controller=>clr_divider,
                                                start=>st_divider,
                                                over_flow=>overflow,
                                                done_division=>donedivision,
                                                q_or_out=>quot_value,
                                                done_r => d_o_n_e_r,
                                                load_dividend=>l_d_d_i_v_i_d_e_n_d,
                                                load_divisor=>l_d_d_i_v_i_s_o_r,
                                                start_ctrl0=>start_divider,
                                                ld_o_flow=>loadoverflow,
                                                ld_result=>loadresult,
                                                ld_q_or_out=>loadqzero,
                                                done_final=>d_o_n_e,
                                                clr_quotient=>clear_quot,
                                                clr_final=>clearfinal,
                                                clear_divider=>cleardivider,
                                                load_quotient=>loadquotient);

FR:final_register port map(clk=>clk,
                clr=>clearfinal,
                                        load_result=>loadresult,
                        result_in=>resultin,
                        result_out=>result);

end Behavioral;

--done.
--in the process of 16-bit changes.
-- VHdl source module to convert a twos complement number to an unsigned binary number.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity conver is
port(a_in: in std_logic_vector(15 downto 0);
            bout: out std_logic_vector(14 downto 0));
            --sign:out std_logic);
end conver;



architecture Behavioral of conver is

component bfa_divider is
port(in_a,in_b,in_c:in std_logic;
            sum_a,cout_a:out std_logic);
end component bfa_divider;
signal b_internal,b_out,carry_int:std_logic_vector(14 downto 0);

begin
--process(a_in) is
--begin
--sign <= a_in(20);
b_out <= not(a_in(14 downto 0));

H0: bfa_divider port
map(in_a=>b_out(0),in_b=>'0',in_c=>'1',sum_a=>b_internal(0),cout_a=>carry_int(0));
H1toN: for i in 1 to 14 generate
HI: bfa_divider port map(in_a=>b_out(i),in_b=>'0',in_c=>carry_int(i-
1),sum_a=>b_internal(i),cout_a=>carry_int(i));
end generate;

-- process statement to choose output.
process(a_in,b_internal) is
begin
if(a_in(15)='1')then
 bout <= b_internal;
else
 bout <= a_in(14 downto 0);
end if;
end process;


end Behavioral;

-- VHDL source module for bfa used in arithmetic unit in divider.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity bfa_divider is
port(in_a,in_b,in_c:in std_logic;
            sum_a,cout_a:out std_logic);
end bfa_divider;
```

```
architecture Behavioral of bfa_divider is
signal input_sig:std_logic_vector(2 downto 0);
begin
input_sig <= in_a & in_b & in_c;
process(input_sig) is
begin
if(input_sig = "000")then sum_a <= '0'; cout_a <= '0';
elsif(input_sig = "001")then sum_a <= '1'; cout_a <= '0';
elsif(input_sig = "010")then sum_a <= '1'; cout_a <= '0';
elsif(input_sig = "011")then sum_a <= '0'; cout_a <= '1';
elsif(input_sig = "100")then sum_a <= '1'; cout_a <= '0';
elsif(input_sig = "101")then sum_a <= '0'; cout_a <= '1';
elsif(input_sig = "110")then sum_a <= '0'; cout_a <= '1';
elsif(input_sig = "111")then sum_a <= '1'; cout_a <= '1';
end if;
end process;
end Behavioral;

-- VHDL source module to convert a twos complement number to an unsigned binary number.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity conver2 is
port(a_in: in std_logic_vector(14 downto 0);
        bout: out std_logic_vector(15 downto 0);
        sign:in std_logic);
end conver2;


architecture Behavioral of conver2 is

component bfa_divider is
port(in_a,in_b,in_c:in std_logic;
        sum_a,cout_a:out std_logic);
end component bfa_divider;
signal b_internal,b_out,carry_int:std_logic_vector(14 downto 0);

begin
--process(a_in) is
--begin

b_out <= not(a_in(14 downto 0));

H0: bfa_divider port
map(in_a=>b_out(0),in_b=>'1',in_c=>'0',sum_a=>b_internal(0),cout_a=>carry_int(0));
H1toN: for i in 1 to 14 generate
HI: bfa_divider port map(in_a=>b_out(i),in_b=>'0',in_c=>carry_int(i-
1),sum_a=>b_internal(i),cout_a=>carry_int(i));
end generate;
--end process;
--b_out_out <= b_out;
process(a_in,sign,b_internal)is
begin
```

```
if(sign = '1') then
        bout <= sign&b_internal;
else
        bout <= sign&a_in;
end if;
end process;
end Behavioral;
--latest changes made:
--used a flip flop which is set to '1' when overflow occurs.
-- VHDL source module for divider that performs 21 x 21 bit signed division.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity divider is
port(clk,clear, start_all: in std_logic;
        divisor_in:in std_logic_vector(14 downto 0);
        dividend_in:in std_logic_vector(14 downto 0);
        done_a, over_flow:out std_logic;
        --counterout:out std_logic_vector(4 downto 0);
        --carryout_arith:out std_logic;
        --l_l_s_out,zero_out,ctrl_count_out:out std_logic;
        division_out:out std_logic_vector(14 downto 0);
        quotient_value:out std_logic);
end  divider;



architecture Behavioral of divider is

--component declaration.
component arith is
port(in1,in2:in std_logic_vector(14 downto 0);
        out1:out std_logic_vector(14 downto 0);
        carry_out:out std_logic);
end component arith;

component countera is
--generic(m:positive:=6);
port(clk,load:in std_logic;
        --counter_out:out std_logic_vector(4 downto 0);
        count_ctrl:in std_logic;
        zero:out std_logic);
end component countera;

component divider_controller is
port(clk,reset_controller,start:in std_logic;
        zero,msb:in std_logic;
        load_counter,count_ctrl,sel_mux_2x1,reset_a_q,load_a_q,load_a,
        lls,sel_mux_2x1_a,reset_rc,load_rc,load_b,done,overflow,setit,resetit: out std_logic);
end component divider_controller;

component mux_2x1 is
```

```
port(a,b,sel:in std_logic;
            c:out std_logic);
end component mux_2x1;

component mux_2x1_a is
port(a_in,b_in:in std_logic_vector(14 downto 0);
            sel:in std_logic;
            muxout:out std_logic_vector(14 downto 0));
end component mux_2x1_a;

component or_gate is
port(ina,inb:in std_logic;
            outa:out std_logic);
end component or_gate;

component regiser_b is
port(divisor:in std_logic_vector(14 downto 0);
            clk,load_b:in std_logic;
            regb_out:out std_logic_vector(14 downto 0));
end component regiser_b;

component register_a_q is
port (reset_a_q,load_a_q,clk,load_a,lls,or_out: in std_logic;
             dividend:in std_logic_vector(14 downto 0);
             a_in:in std_logic_vector(14 downto 0);
             output:out std_logic_vector(29 downto 0);
             lsb:out std_logic;
             a_out:out std_logic_vector(14 downto 0));
end component register_a_q;

component register_c is
port(a_b_bar:in std_logic_vector(14 downto 0);
            reset_rc,load_rc,clk,c_o_u_t:in std_logic;
            regc_out: out std_logic_vector(14 downto 0);
            msb: out std_logic);
end component register_c;

component setdff_div is
   Port (set_div,reset_div,clk:in std_logic;
         out_div:out std_logic);
end component setdff_div;

-- Signals declaration.
-- aout(4:0) -> output 'a' from register AQ;
-- bout(4:0) -> output 'b' from register B;
-- arith_out(4:0) -> output from arithmetic unit;
-- loadcounter -> loads the counter with value n;
-- ctrl_count -> starts decrementing the counter;
-- zerocount -> input to controller from counter;
-- m_s_b -> most significant bit of register C;
-- select_1-> select line for mux_2x1;
-- rst_a_q -> reset input for register AQ;
-- ld_aq -> load input for register AQ;
-- ld_a ->  load input for A part in register AQ;
-- l_l_s -> logical left shift for register AQ;
-- select_2 -> select line for mux_2x1_a;
```

```
-- rst_rc -> reset input for register C;
-- ld_rc -> load input for register C;
-- ld_b -> load input for register B;
-- mux_2x1_out -> output of mux2x1;
-- regcout -> output of register C;
-- ain(4:0)-> output of mux_2x1_a to A in AQ;
-- q_o -> lsb of register AQ;
-- orout -> output of OR gate;
-- c_out_arith -> carry out of arithmetic unit.



signal aout,bout,arith_out,regcout,ain:std_logic_vector(14 downto 0);
signal loadcounter,ctrl_count,zerocount,m_s_b,select_1,rst_a_q,ld_aq,ld_a,l_l_s,
        select_2,rst_rc,ld_rc,ld_b,mux_2x1_out,q_o,orout,c_out_arith,overflow_int,
        set_it,reset_it:std_logic;
signal division_out_sig:std_logic_vector(29 downto 0);
begin


A0: arith port map(in1=> aout,
                   in2 => bout,
                   out1 => arith_out,
                   carry_out=>c_out_arith);

C0:countera port map(clk=> clk,
                     load => loadcounter,
                     --counter_out=>counterout,
                     count_ctrl=>ctrl_count,
                     zero=>zerocount);

CTRL0:divider_controller port map(clk =>clk,
                                  reset_controller=>clear,
                                  start => start_all,
                                  zero=>zerocount,
                                  msb => m_s_b,
                                  load_counter=>loadcounter,
                                  count_ctrl=>ctrl_count,
                                  sel_mux_2x1=>select_1,
                                  reset_a_q=>rst_a_q,
                                  load_a_q=>ld_aq,
                                  load_a => ld_a,
                                  lls=>l_l_s,
                                  sel_mux_2x1_a=>select_2,
                                  reset_rc => rst_rc,
                                  load_rc=>ld_rc,
                                  load_b => ld_b,
                                  done => done_a,
                                  overflow=>overflow_int,
                                  setit =>set_it,
                                  resetit=>reset_it);

MUX0:mux_2x1 port map(a =>'0',
                      b =>'1',
                      sel=>select_1,
```

```
                                    c=>mux_2x1_out);

MUXA:mux_2x1_a port map(a_in=>aout,
                                     b_in=>regcout,
                                          sel=>select_2,
                          muxout=>ain);

OR0:or_gate port map(ina=>q_o,
                                        inb=>mux_2x1_out,
                                        outa=>orout);

REGB:regiser_b port map(divisor=>divisor_in,
                                         clk => clk,
                                         load_b=>ld_b,
                                         regb_out=>bout);

RAQ:register_a_q port map(reset_a_q => rst_a_q,
                                    load_a_q=>ld_aq,
                                                clk=>clk,
                                                load_a=>ld_a,
                                                lls=>l_l_s,
                                                or_out=>orout,
                              dividend=>dividend_in,
                                                a_in => ain,
                                             output=>division_out_sig,
                                       lsb => q_o,
                                                a_out =>aout);

RC:register_c port map(a_b_bar => arith_out,
                                        reset_rc=>rst_rc,
                                        load_rc=>ld_rc,
                                        clk=>clk,
                                        c_o_u_t=>c_out_arith,
                                        regc_out=>regcout,
                                        msb=>m_s_b);

SET:setdff_div Port map (set_div => set_it,
                         reset_div=>reset_it,
                                                clk=>clk,
                                                out_div=>over_flow);



--l_l_s_out <= l_l_s;
--zero_out <= zerocount;
--ctrl_count_out <= ctrl_count;
--carryout_arith <= c_out_arith;
division_out <= division_out_sig(14 downto 0);
quotient_value <=  division_out_sig(14) or division_out_sig(13) or division_out_sig(12) or
                                    division_out_sig(11) or division_out_sig(10) or
                                    division_out_sig(9) or division_out_sig(8) or
                                division_out_sig(7) or division_out_sig(6) or
                                division_out_sig(5) or division_out_sig(4) or
                                division_out_sig(3) or division_out_sig(2) or
                                    division_out_sig(1) or division_out_sig(0);
 --division_out_sig(19) or division_out_sig(18) or
```

```
                                --division_out_sig(17) or division_out_sig(16) or
                                --division_out_sig(15) or   division_out_sig(16) or division_out_sig(15) or
end Behavioral;

--in process  of 16-bit change
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity arith is
port(in1,in2:in std_logic_vector(14 downto 0);
            out1:out std_logic_vector(14 downto 0);
            carry_out:out std_logic);
end arith;

architecture Behavioral of arith is
-- component declaration.
component bfa_divider is
port(in_a,in_b,in_c:in std_logic;
            sum_a,cout_a:out std_logic);
end component bfa_divider;

signal in2_bar: std_logic_vector(14 downto 0);
signal out1_sig: std_logic_vector(14 downto 0);
signal carry_internal: std_logic_vector(14 downto 0);
begin
in2_bar <= not(in2);

D0: bfa_divider port map(in_a =>in1(0),in_b=> in2_bar(0), in_c=>'1',sum_a=>out1_sig(0),cout_a =>
carry_internal(0));

D1toN: for i in 1 to 14 generate
            DI:         bfa_divider port map(in_a =>in1(i),in_b=> in2_bar(i), in_c=>carry_internal(i-
1),sum_a=>out1_sig(i),cout_a => carry_internal(i));
              end generate;
out1 <= out1_sig;
carry_out <= carry_internal(14);
end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity countera is
--generic(m:positive:=6);
port(clk,load:in std_logic;
            --counter_out:out std_logic_vector(4 downto 0);
            count_ctrl:in std_logic;
            zero:out std_logic);
end countera;

architecture Behavioral of countera is
```

```
signal count: std_logic_vector(4 downto 0);
begin
process(clk) is
begin
--if(reset='1')then
        --count <= (others => '0');
if(clk'event and clk='1') then
        if(load ='1') then
        count <= "01110";
        elsif(count_ctrl = '1') then
        count <= count - 1;
                if(count = "00000")then
                zero <= '1';
                else zero <= '0';
                end if;
        elsif(count_ctrl = '0') then
        count <= count;
        end if;
end if;
end process;
--counter_out<= count;
end Behavioral;

--VHDL souce module for controller in divider.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity divider_controller is
port(clk,reset_controller,start:in std_logic;
        zero,msb:in std_logic;
        load_counter,count_ctrl,sel_mux_2x1,reset_a_q,load_a_q,load_a,
        lls,sel_mux_2x1_a,reset_rc,load_rc,load_b,done,overflow,setit,resetit: out std_logic);
end divider_controller;


architecture Behavioral of divider_controller is

component dff_div_controller is
port(clk,reset,d:in std_logic;
        q: out std_logic);
end component dff_div_controller;

signal  t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,sa,sb,sc,sd,se,sf,sg:std_logic;
signal debouncer_1, debouncer_2,start_ctrl:std_logic;
begin

DEBOUNCER0: dff_div_controller port map(clk=>clk,reset=>reset_controller,d => start,q =>
debouncer_1);
DEBOUNCER1: dff_div_controller port map(clk=>clk,reset=>reset_controller,d => debouncer_1,q =>
debouncer_2);
start_ctrl <= debouncer_1 and (not(debouncer_2));
```

```
D0:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => start_ctrl,q => t0);
D1:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => t0,q => t1);
D2:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => t1,q => t2);
D3:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => sa,q => t3);
D4:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => sb,q => t4);
D5:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => t4,q => t5);
D6:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => sc,q => t6);
D7:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => sd,q => t7);
D8:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => se,q => t8);
D9:dff_div_controller port map(clk=>clk,reset=>reset_controller,d => sf,q => t9);
D10:dff_div_controller port map(clk=>clk,reset=>reset_controller,d =>sg,q => t10);

-- assigning signals sa,sb,sc,sd,se, and sf.
sa <= (msb) and t2;
sb <= t10 or ((not(msb)) and t2) or ((not(msb)) and t9);
sc <= zero and t5;
sd <= (msb) and t6;
se <= ((not(msb)) and t6) or t7 or t3;
sf <= (not(zero)) and t5;
sg <= (msb) and t9;

load_counter <= t0;
count_ctrl <= t4;
sel_mux_2x1 <= t7 or t10;
reset_a_q <= '0';
load_a_q<= t0;
load_a <=t7 or t10;
lls <= t4;
sel_mux_2x1_a <= t7 or t10;
reset_rc <= t0;
load_rc <= t1 or t5 or t8;
load_b <= t0;
done <= t8;
overflow <= t3;
setit <= t3;
resetit <= t0;
end Behavioral;

-- VHDL source module for D- Flip Flop used in the controller of divider.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity dff_div_controller is
port(clk,reset,d:in std_logic;
        q: out std_logic);
end dff_div_controller;

architecture Behavioral of dff_div_controller is
signal q_sig:std_logic;
begin
process(clk,reset) is
```

```
begin
if(reset='1') then
        q_sig <= '0';
elsif(clk'event and clk='1')then
        q_sig <= d;
end if;
end process;
q <= q_sig;
end Behavioral;

-- VHDL source for 2X1 multiplexor.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity mux_2x1 is
port(a,b,sel:in std_logic;
        c:out std_logic);
end mux_2x1;

architecture Behavioral of mux_2x1 is

begin
process( sel,a,b) is
begin
if(sel = '0') then
        c <= a;
else
        c <= b;
end if;
end process;
end Behavioral;

-- VHDL source code for 5 bit 2X1 multiplexor.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity mux_2x1_a is
port(a_in,b_in:in std_logic_vector(14 downto 0);
        sel:in std_logic;
        muxout:out std_logic_vector(14 downto 0));
end mux_2x1_a;

architecture Behavioral of mux_2x1_a is

begin
process(a_in,b_in,sel) is
begin
if(sel ='0')then
```

```
        muxout <= a_in;
else
        muxout <= b_in;
end if;
end process;

end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity or_gate is
port(ina,inb:in std_logic;
        outa:out std_logic);
end or_gate;

architecture Behavioral of or_gate is

begin
outa <= ina or inb;
end Behavioral;

-- VHDL source module for register B used to hold the value of divisor.
-- Functionality:
-- loads the value of divisor into the register B when load_b input from controller
-- is '1' at the rising edge of clock, else stores the same value.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity regiser_b is
port(divisor:in std_logic_vector(14 downto 0);
        clk,load_b:in std_logic;
        regb_out:out std_logic_vector(14 downto 0));
end regiser_b;

architecture Behavioral of regiser_b is
signal regb_out_sig: std_logic_vector(14 downto 0);
begin
process(clk)is
begin
if(clk'event and clk='1')then
        if (load_b ='1') then
        regb_out_sig<= divisor;
        else
        regb_out_sig<= regb_out_sig;
        end if;
end if;
end process;
regb_out <= regb_out_sig;
```

```
end Behavioral;

-- This code is modified for dividend width of 42 bits.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity register_a_q is
port (reset_a_q,load_a_q,clk,load_a,lls,or_out: in std_logic;
            dividend:in std_logic_vector(14 downto 0);
            a_in:in std_logic_vector(14 downto 0);
            output:out std_logic_vector(29 downto 0);
            lsb:out std_logic;
            a_out:out std_logic_vector(14 downto 0));
end register_a_q;

architecture Behavioral of register_a_q is
signal output_sig: std_logic_vector(29 downto 0);
begin
process(clk,reset_a_q)is
begin
if(reset_a_q = '1') then
          output_sig <= (others=>'0');
elsif(clk'event and clk='1') then
          if(load_a_q ='1')then
          --output_sig(19 downto 0) <= dividend;
          --output_sig(39 downto 20) <= (others => '0');
          output_sig(29 downto 25) <= (others =>'0');--29 28 27 26 25
          output_sig(24 downto 10) <= dividend;--24 23 22 21 20 19 18 17 16 15 14 13 12 11 10
          output_sig(9 downto 0) <= (others => '0');--9 8 7 6 5 4 3 2 1 0
          elsif(load_a='1')then
          output_sig(29 downto 15) <= a_in;
          output_sig(0)<= or_out;
          elsif(lls='1')then
          output_sig<= output_sig(28 downto 0)& '0';
          else
          output_sig <= output_sig;
          end if;
end if;
end process;

output <= output_sig;
lsb <= output_sig(0);
a_out <= output_sig(29 downto 15);

end Behavioral;

--VHDL source code for register C that is used to hold the value of
--a + not(b) +1.
--Functionality:At rising edge of clock when reset_rc = '0' and load_rc
--is '1' then the value of a + not(b) + 1 is loaded into the register.
--MSB is the output that shows the value of carryout of the result,
```

```
--with which it can be concluded whether first 5 bits of dividend is
--greater than first 5 bits of divisor or not.
--This code is modified for a bit width of 16 bits of the division.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity register_c is
   Port (a_b_bar:in std_logic_vector(14 downto 0);
                  reset_rc,load_rc,clk,c_o_u_t:in std_logic;
                  regc_out:out std_logic_vector(14 downto 0);
                  msb:out std_logic );
end register_c;

architecture Behavioral of register_c is
signal regc_out_sig:std_logic_vector(14 downto 0);
signal msb_sig: std_logic;

begin

process(clk,reset_rc)is
begin
if(reset_rc = '1') then
        regc_out_sig <= (others=>'0');
elsif(clk'event and clk='1')then
        if(load_rc='1')then
                regc_out_sig <= a_b_bar;
                msb_sig <= c_o_u_t;
        else
                regc_out_sig <= regc_out_sig;
                msb_sig <= msb_sig;
        end if;
end if;
end process;
regc_out <= regc_out_sig;
msb <= msb_sig;
end Behavioral;

--VHDL source module for a flip flop which has two inputs, set and reset.
--When set the output is set to '1' otherwise it will always be '0';
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity setdff_div is
   Port (set_div,reset_div,clk:in std_logic;
        out_div:out std_logic);
end setdff_div;
```

```
architecture Behavioral of setdff_div is
signal out_div_sig:std_logic;
begin
process(clk) is
begin
if(clk'event and clk = '1')then
        if(reset_div = '1') then
                out_div_sig <= '0';
        elsif(set_div = '1')then
                out_div_sig <= '1';
        else
                out_div_sig <= out_div_sig;
        end if;
else
        out_div_sig <= out_div_sig;
end if;
end process;
out_div <= out_div_sig;
end Behavioral;

-- VHDL source module for controller for the final divider.
-- This controller actas as the main controller for the divider
-- which inturn has its own controller to perform the division operation.
-- This controller converts the 2's complement -> unsigned -> 2's complement
-- and stores the final result in final register.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity final_divider_controller is
port(clk,clear_final_controller,start,over_flow,done_division,q_or_out,done_r:in std_logic;
        load_dividend,load_divisor,start_ctrl0,ld_o_flow,ld_result,ld_q_or_out,
        done_final,clr_quotient,clr_final,clear_divider,load_quotient:out std_logic);
end final_divider_controller;

architecture Behavioral of final_divider_controller is

component dff_final_divider_controller is
port(clk,clr,d:in std_logic;
        q:out std_logic);
end component dff_final_divider_controller;

signal debouncer_1,debouncer_2,start_ctrl:std_logic;
signal t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,sa,sb,sc,sd,se,sf,sg,sh,si:std_logic;

begin
DEBOUNCER0: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => start,q
=> debouncer_1);
DEBOUNCER1: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d =>
debouncer_1,q => debouncer_2);
start_ctrl <= debouncer_1 and (not(debouncer_2));
```

```
D0: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => start_ctrl,q =>t0);
D1: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => t0,q =>t1);
D2: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => sa,q =>t2);
D3: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => sb,q =>t3);
D4: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => sc,q =>t4);
D5: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => sd,q =>t5);
D6: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => se,q =>t6);
D7: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => sf,q =>t7);
D8: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => si,q =>t8);
D9: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => t8,q =>t9);
D10: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => sg,q =>t10);
D11: dff_final_divider_controller port map(clk=>clk,clr=>clear_final_controller,d => sh,q =>t11);

-- control equations.
sa <= ((not(done_division)) and t2) or t1;
sb <= (done_division) and t2;
sc <= (not(over_flow))and t3;
sd <= over_flow and t3;
se <= q_or_out and t4;
sf <= ((not(q_or_out)) and t4);
sg <= t9 or (t10 and (not(done_r)));
sh <= t10 and done_r;
si <= t5 or t6 or t7;
--assigning outputs.
load_dividend <= t0;
load_divisor <= t0;
start_ctrl0 <= t1;
ld_o_flow<= t5;
ld_result<=t9;
ld_q_or_out<=t7;
done_final<=t9 or t10;
clr_quotient<=t0;
clr_final<=t0;
clear_divider<=t0;
load_quotient<=t6;

end Behavioral;

-- VHDL source module for D flip flop used in the final_divider_controller.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity dff_final_divider_controller is
port(clk,clr,d:in std_logic;
        q:out std_logic);
end dff_final_divider_controller;

architecture Behavioral of dff_final_divider_controller is
signal q_sig: std_logic;
begin
process(clr,clk) is
begin
        if(clr='1')then
        q_sig <= '0';
```

```
        elsif(clk'event and clk='1')then
                q_sig <= d;
        else
                q_sig <= q_sig;
        end if;
end process;
q<=q_sig;
end Behavioral;

-- VHDL source module for final register of  divider.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity final_register is
port(clk,clr,load_result:in std_logic;
        result_in:in std_logic_vector(15 downto 0);
        result_out:out std_logic_vector(15 downto 0));
end final_register;

architecture Behavioral of final_register is
signal result_out_sig: std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1') then
        result_out_sig <= (others=>'0');
                elsif(clk'event and clk='1') then
                if(load_result ='1')then
                        result_out_sig <= result_in;
                else
                        result_out_sig <= result_out_sig;
                end if;
else
result_out_sig <= result_out_sig;
end if;
end process;
result_out <= result_out_sig;
end Behavioral;

-- VHDL source module for quotient register that stores the value of quotient.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity quotient_register is
port(quotient_in:in std_logic_vector(14 downto 0);
        quotient_out:out std_logic_vector(14 downto 0);
        clk,clr,load_overflow,load_q_zero,load_quotient:in std_logic);
```

```
end quotient_register;

architecture Behavioral of quotient_register is
signal quotient_out_sig: std_logic_vector(14 downto 0);
begin
process(clk,clr)is
begin
if(clr='1')then
        quotient_out_sig <= (others=>'0');
        elsif(clk'event and clk='1') then
                if(load_overflow = '1') then
                        quotient_out_sig <= (others => '1');
                elsif(load_q_zero = '1') then
                        quotient_out_sig <= "000000000000001";
                elsif(load_quotient ='1')then
                        quotient_out_sig <= quotient_in;
                end if;
end if;
end process;
quotient_out <= quotient_out_sig;
end Behavioral;

-- VHDL source module for register used to store the 21 bit dividend.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_dividend is
port(clk,load:in std_logic;
        divin: in std_logic_vector(15 downto 0);
        divout: out std_logic_vector(15 downto 0);
        msb_dividend: out std_logic);
end register_dividend;

architecture Behavioral of register_dividend is
signal divout_sig:std_logic_vector(15 downto 0);
begin
process(clk) is
begin
if(clk'event and clk='1') then
        if (load ='1') then
                divout_sig <= divin;
        else
                divout_sig <= divout_sig;
        end if;
else
        divout_sig <= divout_sig;
end if;
end process;
divout <= divout_sig;
msb_dividend <= divout_sig(15);
end Behavioral;

-- VHDL source module for register used to store the 21 bit divisor.
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_divisor is
port(clk,load:in std_logic;
        divisin: in std_logic_vector(15 downto 0);
        divisout: out std_logic_vector(15 downto 0);
        msb_divisor: out std_logic);
end register_divisor;

architecture Behavioral of register_divisor is
signal divisout_sig:std_logic_vector(15 downto 0);
begin
process(clk) is
begin
if(clk'event and clk='1') then
        if (load ='1') then
                divisout_sig <= divisin;
        else
                divisout_sig <= divisout_sig;
        end if;
else
        divisout_sig <= divisout_sig;
end if;
end process;
divisout <= divisout_sig;
msb_divisor <= divisout_sig(15);
end Behavioral;

--VHDL source code for 16-bit 2X1 multiplexor used to give inputs to the
--arithmetic units.  These multiplexors will get inputs from the register
--outputs.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity multiplex_2_1 is
   Port (in1,in2:in std_logic_vector(15 downto 0);
        sel:in std_logic;
        out1:out std_logic_vector(15 downto 0) );
end multiplex_2_1;

architecture Behavioral of multiplex_2_1 is

begin

process(sel,in1,in2)is
begin
if(sel = '0')then
        out1 <= in1;
else
```

```
            out1 <= in2;
end if;
end process;


end Behavioral;

--VHDL source code for 16-bit 4X1 multiplexor used to give inputs to the
--arithmetic units.  These multiplexors will get inputs from the register
--outputs.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity multiplex_4_1 is
    Port (in1,in2,in3,in4:in std_logic_vector(15 downto 0);
          sel1,sel2:in std_logic;
          out1:out std_logic_vector(15 downto 0) );
end multiplex_4_1;

architecture Behavioral of multiplex_4_1 is
signal sel:std_logic_vector(1 downto 0);
begin
sel <= sel1 & sel2;
process(sel,in1,in2,in3,in4)is
begin
if(sel = "00")then
            out1 <= in1;
elsif(sel = "01")then
            out1 <= in2;
elsif(sel = "10")then
            out1 <= in3;
else
            out1 <= in4;
end if;
end process;


end Behavioral;

--changes being made to accomodate 5 integer bits in 16 bits registers.
--This change is required for the results like 26.9 of moisture value.
--done.
--in the process of changing to 16-bits.
-- Design of serial sequential 'nXn' multiplier using Booth's algorithm.
-- for example n is considered as 21.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity multiplier_a_mul is
port(clk,rst,start_mul,d_o_n_e_r:in std_logic;
        multiplicand,multiplier: in std_logic_vector(15 downto 0);
        m_result_out: out std_logic_vector(15 downto 0);
        --countout_out:out std_logic_vector(4 downto 0);
        done_mul:out std_logic);
end multiplier_a_mul;

-- Commenting unfinished.
--port description.
--clk -->clk,rst --> reset for the controller.
--start_mul-->start signal given to the controller.
--clr_final-->reset signal for the final register used to hold the product value.
--multiplicand -- multiplicand input;
--multiplier -- multiplier;   m_out --> final product.

architecture Behavioral of multiplier_a_mul is

component countera_mul is
--generic(m:positive:=6);
port(clk,load:in std_logic;
        --counter_out:out std_logic_vector(4 downto 0);
        count_ctrl:in std_logic;
        zero:out std_logic);
end component countera_mul;

component multiplier_controller is
port(start,zero,rb_o,co,clk,clr,or_out,done_r: in std_logic;
        clr_regabco,loadab_reset_co,loada,reset_m,load_m,ars,sel_bin,
        sel_cin,load_c,count_ctrl,load_f,clr_f,done,sel_result,
        clr_result,load_result:out std_logic);
end component multiplier_controller;


component dffa_mul is
port(clk,reset,load,d:in std_logic;
        q: out std_logic);
end component dffa_mul;


component nbitadder_mul is
--generic(n:positive:=32);
port(a_input,b_input: in std_logic_vector(15 downto 0);
        sel_cin,sel_bin:in std_logic;
        adder_out: out std_logic_vector(15 downto 0));
end component nbitadder_mul;

component reg_a_b_co_mul is
port(clk,clr,loadab_rst_co,loada,ars:in std_logic;
        a_in,b_in:in std_logic_vector(15 downto 0);
        about:out std_logic_vector(32 downto 0));

end component reg_a_b_co_mul;
```

```
component register_m_mul is
--generic(n:positive:= 8); -- generic declaration of register.
port(clk, reset: in std_logic;
        p_in: in std_logic_vector(15 downto 0);
        pout:out std_logic_vector(15 downto 0);
        loadm:in std_logic);
end component register_m_mul;

component finalregister_mul is
port(clk,clr,load:in std_logic;
        a:in std_logic_vector(15 downto 0);
        f: out std_logic_vector(15 downto 0);
        or_out:out std_logic);
end component finalregister_mul;

--'mux_result' component is used at the end to selectthe result
-- between calculated result and least value. This prevents the
--problem of having zero as result.
component mux_result_mul is
port(a:in std_logic_vector(15 downto 0);
        sel:in std_logic;
        b:in std_logic_vector(15 downto 0);
        c:out std_logic_vector(15 downto 0));
end component mux_result_mul;

component result_register_mul is
port(a:in std_logic_vector(15 downto 0);
        b:out std_logic_vector(15 downto 0);
        clk,clr_result,load_result:in std_logic);
end component result_register_mul;

signal zero_a,rb0,check,a_r_s,selbin,selcin,loadc,
        countctrl,loadF,resetM,loadin_M:std_logic;
signal reset_regabc0,load_abco,load_a,clear_f,o_r_o_u_t,s_result,c_result,ld_result:std_logic;
signal rm_out,adderout,finalreg_input,m_out: std_logic_vector(15 downto 0);
signal about_int:std_logic_vector(32 downto 0);
signal mux_result_in,result_in:std_logic_vector(15 downto 0);

--signal m_out_sig:std_logic_vector(9 downto 0);
-- zero_a --> for signal zero.
-- rb0 --> for signal RB[0].
-- check --> for flipflop output to check(c0)
-- resetA --> for reset of register A
-- loadA --> for load of register A
-- resetB --> for reset of register B
-- loadB --> for load of register B
-- resetM --> for reset of register M
-- loadin_M --> for load of register M
-- a_r_s --> for ars, l_r_s --> for lrs
-- selbin,selcin --> for sel_bin and sel_cin
-- loadC -->loadcounter, countctrl --> control for decrementing counter.
-- loadF --> load final register.
-- loadcheck --> used to load check bit flip-flop.
-- adderout --> output of adder. sout_a --> serial output of registerA.
```

```
-- check_in --> input of check bit flipflop.
begin
mux_result_in <= "0000000000000001";
rb0 <= about_int(1);
check<= about_int(0);
CONTROLLER0: multiplier_controller port map(start=>start_mul,zero=>zero_a,rb_o=>rb0,
                                co=>check,clk=>clk,clr=>rst,or_out=>o_r_o_u_t,done_r=>d_o_n_e_r,
                                clr_regabco=>reset_regabc0,loadab_reset_co=>load_abco,
                                loada=> load_a,reset_m=>resetM, load_m=>loadin_M,
                                ars=>a_r_s,sel_bin=>selbin,sel_cin=>selcin,
                                load_c=>loadc,count_ctrl=>countctrl,load_f=>loadF,clr_f=>clear_f,

          done=>done_mul,sel_result=>s_result,clr_result=>c_result,load_result=>ld_result);


REGISTERABCO: reg_a_b_co_mul port map(clk=>clk,clr=>reset_regabc0,
                                      loadab_rst_co=>load_abco,loada=>load_a,ars=>a_r_s,
                                      a_in=>adderout,b_in=>multiplier,about=>about_int);

REGISTERM: register_m_mul port map(clk=>clk,reset=>resetM,p_in=>multiplicand,
          pout=>rm_out,loadm=>loadin_M);

ADDER: nbitadder_mul port map(a_input=>about_int(32 downto 17),b_input=>rm_out,sel_cin=>selcin,
                                           sel_bin=>selbin,adder_out=>adderout);
--36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19|18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
COUNTER: countera_mul port map (clk=>clk,load=>loadc,
                                            count_ctrl=>countctrl,zero=>zero_a);

--assigning signal to finalreg_input.
--This limits the 42 bit result to 21 bits.
finalreg_input <= about_int(32) & about_int(25 downto 11);
FINALREG:finalregister_mul port map(clk=>clk,clr=>clear_f,load=>loadF,
          a=>finalreg_input,f=>m_out,or_out=>o_r_o_u_t);


MXRESULT: mux_result_mul port map(a => m_out,
                                  sel => s_result,
                                b => mux_result_in,
                                c => result_in);

RSREGISTER:result_register_mul port map(a => result_in,
                                   b => m_result_out,
                                                                clk=>clk,
                                                                clr_result=>c_result,
                                                                load_result=>ld_result);



end Behavioral;

-- Counter of width 'm' used in Booth's multiplier.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity countera_mul is
--generic(m:positive:=6);
port(clk,load:in std_logic;
        --counter_out:out std_logic_vector(4 downto 0);
        count_ctrl:in std_logic;
        zero:out std_logic);
end countera_mul;

architecture Behavioral of countera_mul is
signal count: std_logic_vector(4 downto 0);
begin
process(clk) is
begin
--if(reset='1')then
        --count <= (others => '0');
if(clk'event and clk='1') then
        if(load ='1') then
        count <= "01111";
        elsif(count_ctrl = '1') then
        count <= count - 1;
                if(count = "00000")then
                zero <= '1';
                else zero <= '0';
                end if;
        elsif(count_ctrl = '0') then
        count <= count;
        end if;
end if;
end process;
--counter_out<= count;
end Behavioral;

-- VHDL source code for final register used to store the final result.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity finalregister_mul is
port(clk,clr,load:in std_logic;
        a:in std_logic_vector(15 downto 0);
        f: out std_logic_vector(15 downto 0);
        or_out:out std_logic);
end finalregister_mul;

architecture Behavioral of finalregister_mul is
signal f_sig: std_logic_vector(15 downto 0);
begin
process(clk,clr) is
begin
if(clr='1') then
        f_sig <=(others =>'0');
elsif(clk'event and clk = '1') then
```

```
            if(load='1')then
            f_sig(15 downto 0) <= a;
            else
            f_sig <= f_sig;
            end if;
end if;
end process;
f <= f_sig;
or_out <=  f_sig(15) or f_sig(14) or f_sig(13) or
                        f_sig(12) or f_sig(11) or f_sig(10) or
                        f_sig(9) or  f_sig(8) or  f_sig(7) or
                        f_sig(6) or  f_sig(5) or  f_sig(4) or
                        f_sig(3) or  f_sig(2) or  f_sig(1) or f_sig(0);
end Behavioral;
 --f_sig(20) or f_sig(19) or f_sig(18) or
                       --f_sig(17) or f_sig(16) or f_sig(17) or f_sig(16) or

-- VHDL source code for controller used in multiplier.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity multiplier_controller is
port(start,zero,rb_o,co,clk,clr,or_out,done_r: in std_logic;
            clr_regabco,loadab_reset_co,loada,reset_m,load_m,ars,sel_bin,
            sel_cin,load_c,count_ctrl,load_f,clr_f,done,sel_result,
            clr_result,load_result:out std_logic);
end multiplier_controller;



-- i/o port description follows:
-- start --> start signal applied to the controller.
-- zero  --> 'zero' input to the controller from counter.
-- rb_o  --> 'RB[0]' bit as input.
-- co    --> 'check flip flop input'.
-- clk   --> clk as input.
-- clr   --> clear input to controller.
-- reset_a--> reset input to registerA.
-- load_a --> load input to registerA.
-- reset_b--> reset input to registerB.
-- load_b --> load input to registerB.
-- reset_m--> reset input to registerM.
-- load_m --> load input to registerM.
-- ars --> Arithmetic Right Shift signal sent to RegisterA.
-- lrs --> Logical right shift signal sent to RegisterB.
-- sel_bin-->control signal to adder that selects either b or not(b).
-- sel_cin-->control signal to adder that chooses either addition or subtraction.
-- load_c-->loads the counter with value'n'.
-- count_ctrl-->control input to counter to start decrementing count.
-- load_f--> loads the final product to final register.
-- load_co-->loads the check bit flip-flop.
-- reset_co-->resets the check bit flip-flop.
```

```
-- done --> signal to indicate that multiplication is over.

architecture Behavioral of multiplier_controller is

component dff_mul is
port(clk,reset,d:in std_logic;
            q: out std_logic);
end component dff_mul;

signal debouncer_1,debouncer_2,start_ctrl,t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12: std_logic;
signal sa,sb,sc,sd,se,sf,sg,sh,si: std_logic;

-- signal description follows.
-- debouncer_1 --> output of first flip flop in debouncer circuit.
-- debouncer_2 --> output of second flip flop in debouncer circuit.
-- start_ctrl --> start input given to controller.
-- t0...t6 --> outputs of flip flops D0..D7 used in controller.

begin
DEBOUNCER0: dff_mul port map(clk=>clk,reset=>clr,d => start,q => debouncer_1);
DEBOUNCER1: dff_mul port map(clk=>clk,reset=>clr,d => debouncer_1,q => debouncer_2);
start_ctrl <= debouncer_1 and (not(debouncer_2));
D0: dff_mul port map(clk=>clk, reset=>clr, d=>start_ctrl,q=>t0);
D1: dff_mul port map(clk=>clk, reset=>clr, d=>sa,q=>t1);
D2: dff_mul port map(clk=>clk, reset=>clr, d=>sb,q=>t2);
D3: dff_mul port map(clk=>clk, reset=>clr, d=>sc,q=>t3);
D4: dff_mul port map(clk=>clk, reset=>clr, d=>sd,q=>t4);
D5: dff_mul port map(clk=>clk, reset=>clr, d=>t4,q=>t5);
D6: dff_mul port map(clk=>clk, reset=>clr, d=>t5,q=>t6);
D7: dff_mul port map(clk=>clk, reset=>clr, d=>se,q=>t7);
D8: dff_mul port map(clk=>clk, reset=>clr, d=>t7,q=>t8);
D9: dff_mul port map(clk=>clk, reset=>clr, d=>sf,q=>t9);
D10:dff_mul port map(clk=>clk, reset=>clr, d=>sg,q=>t10);
D11:dff_mul port map(clk=>clk, reset=>clr, d=>sh,q=>t11);
D12:dff_mul port map(clk=>clk, reset=>clr, d=>si,q=>t12);

-- signal assignments for sa,sb,sc,sd,se,d0_in.


sa <= (t6 and (not(zero))) or t0;
sb <= (rb_o and (not(co)))and t1;
sc <= (not(rb_o)) and co and t1;
sd <= t2 or ((not(rb_o xor co))and t1) or t3;
se <= zero and t6;
sf <= t8 and (not(or_out));
sg <= t8 and or_out;
sh <= (t11 and not(done_r)) or t9 or t10;
si <= t11 and done_r;

-- Control equations for controller.

clr_regabco <= t10;
loadab_reset_co<=t0;
loada <= t2 or t3;
reset_m <= t10;
load_m <= t0;
```

```
ars <= t5;
sel_bin <= t2;
sel_cin <= t2;
load_c <= t0;
count_ctrl <= t5;
load_f <= t7;
clr_f <= t0;
done <= t9 or t10 or t11;
sel_result <= t10;
clr_result <= t0;
load_result <= t9 or t10;


end Behavioral;

-- VHDL source code used for flip-flops used in controller and also flip-flop
-- used for Booth's algorithm.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity dff_mul is
port(clk,reset,d:in std_logic;
        q: out std_logic);
end dff_mul;

architecture Behavioral of dff_mul is

begin
process(reset, clk) is
begin
if(reset ='1') then
        q <= '0';
elsif(clk'event and clk='1') then
        q <= d;
end if;
end process;
end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;
                                        --VHDL source module for multiplexor used at the result register
--to select either result or the least minimum result incase of
--'zero' result.
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity mux_result_mul is
port(a:in std_logic_vector(15 downto 0);
        sel:in std_logic;
        b:in std_logic_vector(15 downto 0);
        c:out std_logic_vector(15 downto 0));
end mux_result_mul;

architecture Behavioral of mux_result_mul is

begin
process(sel,a,b)is
begin
if(sel = '0')then
        c <= b;
else
        c <= a;
end if;
end process;
end Behavioral;


-- VHDL source module used for n-bit ripple carry adder/subtractor

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity nbitadder_mul is
--generic(n:positive:=32);
port(a_input,b_input: in std_logic_vector(15 downto 0);
        sel_cin,sel_bin:in std_logic;
        adder_out: out std_logic_vector(15 downto 0));
end nbitadder_mul;

architecture structural of nbitadder_mul is
component onebitadder_mul is        -- component declaration of onebit adder
port(a_in,b_in,sel_b,c_in:in std_logic;
        sum_bfa, cout:out std_logic);
end component onebitadder_mul;

component mux_2X1_a_mul is      --  component declaration of 2X1 multiplexor
port(a,b:in std_logic;
        sel : in std_logic;
        c: out std_logic);
end component mux_2X1_a_mul;
signal c_in_first: std_logic;
signal carry_internal: std_logic_vector(15 downto 0);
```

```
begin

A0: mux_2X1_a_mul port map(a=>'0',b=>'1',sel=>sel_cin,c=>c_in_first);
B0: onebitadder_mul port map(a_in=>a_input(0),b_in=>
b_input(0),sel_b=>sel_bin,c_in=>c_in_first,sum_bfa=>adder_out(0),cout => carry_internal(0));

B1TON: for i in 1 to 15 generate

        Bi: onebitadder_mul port map(a_in=>a_input(i),b_in=>
b_input(i),sel_b=>sel_bin,c_in=>carry_internal(i-1),sum_bfa=>adder_out(i),cout => carry_internal(i));

        end generate;
 --carryout <= carry_internal(4);
end structural;

-- VHDL code for n-bit input 2X1 multiplexor used in the design of Booth's multiplier.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity mux_2X1_a_mul is
port(a,b:in std_logic;
        sel : in std_logic;
        c: out std_logic);
end mux_2X1_a_mul;

architecture Behavioral of mux_2X1_a_mul is

begin
process(sel,a,b) is
begin
if (sel = '0') then
        c <= a;
else
        c <= b;
end if;
end process;

end Behavioral;

--VHDL source code of 1 bit adder/subtractor.


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity onebitadder_mul is
port(a_in,b_in,sel_b,c_in:in std_logic;
        sum_bfa, cout:out std_logic);
```

```
end onebitadder_mul;

architecture Behavioral of onebitadder_mul is
component mux_2x1_a_mul is
port(a,b:in std_logic;
            sel : in std_logic;
            c: out std_logic);
end component mux_2x1_a_mul;

component bfa_mul is
port(a,b,cin: in std_logic;
            sum, carryout: out std_logic);
end component bfa_mul;
signal bout,b_in_bar:std_logic;
begin
b_in_bar <= not(b_in);

M0: mux_2x1_a_mul port map(a=>b_in, b=>b_in_bar,sel=>sel_b,c=>bout);
--M1: mux_2X1_a_mul port map(a=>'0',b=>'1',sel=>add_sub,c=>cin);      \
-- If sel_b is '0' then b_in is selected else not(b_in) is selected.
B0: bfa_mul port map(a=>a_in,b=>bout,cin=>c_in,sum=>sum_bfa,carryout=>cout);

end Behavioral;

-- VHDL source module of binary full adder used in adder/subtractor.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity bfa_mul is
port(a,b,cin: in std_logic;
            sum, carryout: out std_logic);
end bfa_mul;

architecture Behavioral of bfa_mul is
signal input:std_logic_vector(2 downto 0);
begin
input <= a&b&cin;
 process(a,b,cin,input) is
 begin
if (input = "000") then sum <= '0';carryout <= '0';
elsif (input = "001") then sum <= '1';carryout <= '0';
elsif (input = "010") then sum <= '1';carryout <= '0';
elsif (input = "011") then sum <= '0';carryout <= '1';
elsif (input = "100") then sum <= '1';carryout <= '0';
elsif (input = "101") then sum <= '0';carryout <= '1';
elsif (input = "110") then sum <= '0';carryout <= '1';
else  sum <= '1';carryout <= '1';
end if;

 end process;
```

```
end Behavioral;

-- VHDL code for n-bit input 2X1 multiplexor used in the design of Booth's multiplier.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity mux_2X1_a_mul is
port(a,b:in std_logic;
        sel : in std_logic;
        c: out std_logic);
end mux_2X1_a_mul;

architecture Behavioral of mux_2X1_a_mul is

begin
process(sel,a,b) is
begin
if (sel = '0') then
        c <= a;
else
        c <= b;
end if;
end process;

end Behavioral;

--VHDL source code of 1 bit adder/subtractor.


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity onebitadder_mul is
port(a_in,b_in,sel_b,c_in:in std_logic;
        sum_bfa, cout:out std_logic);
end onebitadder_mul;

architecture Behavioral of onebitadder_mul is
component mux_2x1_a_mul is
port(a,b:in std_logic;
        sel : in std_logic;
        c: out std_logic);
end component mux_2x1_a_mul;

component bfa_mul is
port(a,b,cin: in std_logic;
        sum, carryout: out std_logic);
end component bfa_mul;
```

```
signal bout,b_in_bar:std_logic;
begin
b_in_bar <= not(b_in);

M0: mux_2x1_a_mul port map(a=>b_in, b=>b_in_bar,sel=>sel_b,c=>bout);
--M1: mux_2X1_a_mul port map(a=>'0',b=>'1',sel=>add_sub,c=>cin);      \
-- If sel_b is '0' then b_in is selected else not(b_in) is selected.
B0: bfa_mul port map(a=>a_in,b=>bout,cin=>c_in,sum=>sum_bfa,carryout=>cout);

end Behavioral;

-- VHDL source module of binary full adder used in adder/subtractor.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity bfa_mul is
port(a,b,cin: in std_logic;
            sum, carryout: out std_logic);
end bfa_mul;

architecture Behavioral of bfa_mul is
signal input:std_logic_vector(2 downto 0);
begin
input <= a&b&cin;
 process(a,b,cin,input) is
 begin
if (input = "000") then sum <= '0';carryout <= '0';
elsif (input = "001") then sum <= '1';carryout <= '0';
elsif (input = "010") then sum <= '1';carryout <= '0';
elsif (input = "011") then sum <= '0';carryout <= '1';
elsif (input = "100") then sum <= '1';carryout <= '0';
elsif (input = "101") then sum <= '0';carryout <= '1';
elsif (input = "110") then sum <= '0';carryout <= '1';
else  sum <= '1';carryout <= '1';
end if;

 end process;


end Behavioral;

-- VHDL code for n-bit input 2X1 multiplexor used in the design of Booth's multiplier.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity mux_2X1_a_mul is
port(a,b:in std_logic;
            sel : in std_logic;
```

```
        c: out std_logic);
end mux_2X1_a_mul;

architecture Behavioral of mux_2X1_a_mul is

begin
process(sel,a,b) is
begin
if (sel = '0') then
        c <= a;
else
        c <= b;
end if;
end process;

end Behavioral;

--VHDL source code for concatenated register for A, B, and C0.

-- Thi code is modified for 21 bit multiplication.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity reg_a_b_co_mul is
port(clk,clr,loadab_rst_co,loada,ars:in std_logic;
        a_in,b_in:in std_logic_vector(15 downto 0);
        about:out std_logic_vector(32 downto 0));

end reg_a_b_co_mul;

architecture Behavioral of reg_a_b_co_mul is
signal about_sig:std_logic_vector(32 downto 0);
begin
process(clk,clr) is
begin
if(clr='1') then
about_sig<=(others => '0');
elsif(clk'event and clk='1')then
  if(loadab_rst_co = '1') then
        about_sig(32  downto 17) <= (others => '0');
        about_sig (16 downto 1) <= b_in;
        about_sig(0)<='0';
  elsif(loada ='1') then
        about_sig(32 downto 17) <= a_in;
  elsif(ars ='1') then
        about_sig <= about_sig(32)& about_sig(32 downto 1);
  end if;
end if;
end process;
about <= about_sig;
end Behavioral;
```

```
-- This VHDL code is modified for 21 bit width multiplication.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_m_mul is
--generic(n:positive:= 8); -- generic declaration of register.
port(clk, reset: in std_logic;
            p_in: in std_logic_vector(15 downto 0);
            pout:out std_logic_vector(15 downto 0);
            loadm:in std_logic);
end register_m_mul;

architecture Behavioral of register_m_mul is
signal p_out:std_logic_vector(15 downto 0);
begin
process(clk,reset) is
begin
if(reset='1')then
            p_out<=(others=>'0');
elsif(clk'event and clk = '1') then
            if(loadm='1') then
                        p_out <= p_in;

            else
                        p_out <= p_out;
            end if;
end if;
end process;
   pout <= p_out;
  end Behavioral;

--VHDL source module for result register used to store
--the result of multiplication.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity result_register_mul is
port(a:in std_logic_vector(15 downto 0);
            b:out std_logic_vector(15 downto 0);
            clk,clr_result,load_result:in std_logic);
end result_register_mul;

architecture Behavioral of result_register_mul is
signal b_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr_result) is
begin
if(clr_result ='1') then
            b_sig <= (others => '0');
```

```
elsif(clk'event and clk='1') then
            if(load_result = '1')then
            b_sig <= a;
            else
            b_sig <= b_sig;
            end if;
else
b_sig <= b_sig;
end if;
end process;
b <= b_sig;
end Behavioral;

--When sel_r2 is '0' r2_1 else r2_2.
--VHDL source module for register R2 with multiplexor at input.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity r_r2_mux2x1 is
   Port (clock,clear,left_shift,ld_r2,lrs_r2,sel_r2:in std_logic;
        r2_1,r2_2:in std_logic_vector(15 downto 0);
                    r2_out:out std_logic_vector(15 downto 0));
end r_r2_mux2x1;

architecture Behavioral of r_r2_mux2x1 is

component register_r_2 is
   Port (clk,clr,lls,load,lrs:in std_logic;
        r2input:in std_logic_vector(15 downto 0);
                    r2output:out std_logic_vector(15 downto 0));
end component register_r_2;

component multiplex_2_1 is
   Port (in1,in2:in std_logic_vector(15 downto 0);
        sel: in std_logic;
                    out1:out std_logic_vector(15 downto 0));
end component multiplex_2_1;

signal r2_input:std_logic_vector(15 downto 0);
begin
R20: register_r_2 port map (clk => clock,
                    clr => clear,
                                                lls => left_shift,
                                                load=>ld_r2,
                                                        lrs => lrs_r2,
                                                r2input=>r2_input,
                                                r2output=>r2_out);
MUX2X1R2:multiplex_2_1 port map(in1 => r2_1,
                        in2 => r2_2,
                                                                sel => sel_r2,
                                                                out1=> r2_input);

end Behavioral;
```

```
--VHDL source module for register R2 (16 bits) that is used in the moisture calculating
--unit as a temporary registers.  It has the store and left shift functions.
--When input 'lls' is applied the content in the register is left shifted by
--one place and the new least significant bit would be '1'.

--lrs port added to accomodate for the changes needed to calculate the physical values.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_r_2 is
   Port (clk,clr,lls,load,lrs:in std_logic;
        r2input:in std_logic_vector(15 downto 0);
                    r2output:out std_logic_vector(15 downto 0));
end register_r_2;

architecture Behavioral of register_r_2 is
signal r2output_sig:std_logic_vector(15 downto 0);
begin
           process(clk,clr)is
                    begin
                    if(clr = '1')then
                                r2output_sig <= (others => '0');
        elsif(clk'event and clk='1')then
                                if(load = '1')then
                                r2output_sig <= r2input;
                                elsif(lls = '1')then
                                r2output_sig <= r2output_sig(14 downto 0)& '0';
                                elsif(lrs = '1')then
                                r2output_sig <= r2output_sig(15)&r2output_sig(15 downto 1);--arithmetic right
shift?
                                else
                                r2output_sig <= r2output_sig;
                                end if;
        else
                                r2output_sig <= r2output_sig;
        end if;
           end process;
r2output <= r2output_sig;
end Behavioral;

--
-- sel_r3_a  |        sel_r3_b | r3_out
-------------------------------------
--           0             |           0           | r3_1
--           0             |           1    | r3_3
--    1     |   0    | r3_2
--    1     |   1    | r3_3
--VHDL source module for register R3 with two multiplexors at inputs.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity r_r3_mux2x1 is
   Port (r3_1,r3_2,r3_3:in std_logic_vector(15 downto 0);
        clock,clear,ld_r3,sel_r3_a,sel_r3_b:in std_logic;
                    r3_out:out std_logic_vector(15 downto 0));
end r_r3_mux2x1;

architecture Behavioral of r_r3_mux2x1 is


component multiplex_2_1 is
   Port (in1,in2:in std_logic_vector(15 downto 0);
        sel: in std_logic;
                     out1:out std_logic_vector(15 downto 0));
end component multiplex_2_1;

component register_r_3 is
   Port (clk,clr,load:in std_logic;
        r3input:in std_logic_vector(15 downto 0);
                    r3output:out std_logic_vector(15 downto 0));
end component register_r_3;

signal mux_int, r3_input:std_logic_vector(15 downto 0);
begin
M0:multiplex_2_1 port map (in1 => r3_1,
                    in2 => r3_2,
                                                sel => sel_r3_a,
                                                out1=> mux_int);
M1:multiplex_2_1 port map (in1 => mux_int,
                           in2 => r3_3,
                                                sel => sel_r3_b,
                                                out1=> r3_input);
R3_A:register_r_3 port map(clk => clock,
                    clr => clear,
                                                load=> ld_r3,
                                                r3input=>r3_input,
                                                r3output=>r3_out);
end Behavioral;

--VHDL source module for register R3 (16 bits) that is used in the moisture calculating
--unit as a temporary registers.  It has the store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_r_3 is
   Port (clk,clr,load:in std_logic;
        r3input:in std_logic_vector(15 downto 0);
                     r3output:out std_logic_vector(15 downto 0));
```

```
end register_r_3;

architecture Behavioral of register_r_3 is
signal r3output_sig:std_logic_vector(15 downto 0);
begin
        process(clk,clr)is
                begin
                if(clr = '1')then
                        r3output_sig <= (others => '0');
      elsif(clk'event and clk='1')then
                        if(load = '1')then
                        r3output_sig <= r3input;
                        else
                        r3output_sig <= r3output_sig;
                        end if;
      else
                        r3output_sig <= r3output_sig;
      end if;
        end process;
r3output <= r3output_sig;
end Behavioral;

--When selr1 is '0' r1_1 is selected, else r1_2.
--VHDL source module for register R1 with a 2X1 multiplexor at input.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity r_r_1_mux_21 is
   Port (r1_1,r1_2:in std_logic_vector(15 downto 0);
      clock,clear,ld_r1,selr1,leftshift:in std_logic;
                r1_output:out std_logic_vector(15 downto 0));
end r_r_1_mux_21;

architecture Behavioral of r_r_1_mux_21 is

component multiplex_2_1 is
   Port (in1,in2:in std_logic_vector(15 downto 0);
      sel: in std_logic;
                out1:out std_logic_vector(15 downto 0));
end component multiplex_2_1;

component register_r_1 is
   Port (clk,clr,lls,load:in std_logic;
      r1input:in std_logic_vector(15 downto 0);
                r1output:out std_logic_vector(15 downto 0));
end component register_r_1;

signal r1_input:std_logic_vector(15 downto 0);
begin

R1: register_r_1 port map(clk => clock,
                clr => clear,
```

```
                                                lls => leftshift,
                                                load => ld_r1,
                                                r1input => r1_input,
                        r1output=> r1_output);
MUX2X1: multiplex_2_1 port map(in1 => r1_1,
                        in2 => r1_2,
                                                        sel => selr1,
                                                        out1=> r1_input);
end Behavioral;

--VHDL source module for register R1 (16 bits) that is used in the moisture calculating
--unit as a temporary registers.  It has the store and left shift functions.
--When input 'lls' is applied the content in the register is left shifted by
--one place and the new least significant bit would be '1'.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_r_1 is
   Port (clk,clr,lls,load:in std_logic;
        r1input:in std_logic_vector(15 downto 0);
                        r1output:out std_logic_vector(15 downto 0));
end register_r_1;

architecture Behavioral of register_r_1 is
signal r1output_sig:std_logic_vector(15 downto 0);
begin
           process(clk,clr)is
                        begin
                        if(clr = '1')then
                                        r1output_sig <= (others => '0');
        elsif(clk'event and clk='1')then
                                        if(load = '1')then
                                        r1output_sig <= r1input;
                                        elsif(lls = '1')then
                                        r1output_sig <= r1output_sig(14 downto 0)& '0';
                                        else
                                        r1output_sig <= r1output_sig;
                                        end if;
        else
                                        r1output_sig <= r1output_sig;
        end if;
           end process;
r1output <= r1output_sig;
end Behavioral;

--When selw is '0' rw1 is selected else rw2.
--VHDL source module for register RW with 2X1 multiplexor at the input.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity r_rw_mux2x1 is
   Port (rw1,rw2:in std_logic_vector(15 downto 0);
        selw,clock,clear,ld_rw:in std_logic;
                    rwout:out std_logic_vector(15 downto 0));
end r_rw_mux2x1;

architecture Behavioral of r_rw_mux2x1 is

component multiplex_2_1 is
   Port (in1,in2:in std_logic_vector(15 downto 0);
        sel: in std_logic;
                    out1:out std_logic_vector(15 downto 0));
end component multiplex_2_1;

component register_rw is
   Port (clk,clr,load:in std_logic;
        rw_in:in std_logic_vector(15 downto 0);
                    rw_out:out std_logic_vector(15 downto 0));
end component register_rw;
signal rwin:std_logic_vector(15 downto 0);
begin
M0: multiplex_2_1 port map (in1 => rw1,
                    in2 => rw2,
                                                    sel => selw,
                                                    out1=> rwin);
RW0: register_rw port map (clk => clock,
                   clr => clear,
                                                   load=> ld_rw,
                                                   rw_in => rwin,
                                                   rw_out=>rwout);
end Behavioral;

--VHDL source module for RW(16-bit)which is used to store the value of
--W and another intermediate value. It has store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rw is
   Port (clk,clr,load:in std_logic;
        rw_in:in std_logic_vector(15 downto 0);
                    rw_out:out std_logic_vector(15 downto 0));
end register_rw;

architecture Behavioral of register_rw is
signal rw_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr)is
begin
          if(clr='1')then
    rw_out_sig<=(others=>'0');
          elsif(clk'event and clk='1')then
                    if(load='1')then
                    rw_out_sig<=rw_in;
```

```
                    else
                    rw_out_sig<=rw_out_sig;
                    end if;
          else
          rw_out_sig<=rw_out_sig;
          end if;
end process;
rw_out <= rw_out_sig;
end Behavioral;

--VHDL source module for register M used to hold the final value of moisture.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_moisture_f is
port(clk,clr,load:in std_logic;
    moisture_in:in std_logic_vector(15 downto 0);
          moisture_out:out std_logic_vector(15 downto 0));
end register_moisture_f;

architecture Behavioral of register_moisture_f is
signal moisture_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr) is
begin
if(clr ='1') then
          moisture_out_sig <= (others => '0');
elsif(clk'event and clk='1')then
          if(load ='1')then
          moisture_out_sig <= moisture_in;
          else
          moisture_out_sig <= moisture_out_sig;
          end if;
else
          moisture_out_sig <= moisture_out_sig;
end if;
end process;
moisture_out <= moisture_out_sig;
end Behavioral;

--VHDL source module for register r_h which is used to hold the value of
-- humidity % / 100.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_r_h is
port(clk,clr,load,lrs:in std_logic;
            r_h_in:in std_logic_vector(18 downto 0);
            r_h_out:out std_logic_vector(15 downto 0));
end register_r_h;
```

```
architecture Behavioral of register_r_h is
signal r_h_out_sig: std_logic_vector(18 downto 0);
begin
process(clr,clk) is
begin
if(clr = '1')then
        r_h_out_sig <= (others => '0');
elsif(clk'event and clk = '1')then
        if(load = '1')then
        r_h_out_sig <= r_h_in;
        elsif(lrs = '1')then
        r_h_out_sig <= "0000"&r_h_out_sig(18 downto 4);
        else
        r_h_out_sig <= r_h_out_sig;
        end if;
else
        r_h_out_sig <= r_h_out_sig;
end if;
end process;
r_h_out(15) <= r_h_out_sig (18);
r_h_out(14 downto 10) <= r_h_out_sig(14 downto 10);
r_h_out(9 downto 0) <= r_h_out_sig(9 downto 0);
end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_r_h_o is
port(   clk,clr,load,lls:in std_logic;
                r_h_o_in:in std_logic_vector(15 downto 0);
                r_h_o_out:out std_logic_vector(18 downto 0));
end register_r_h_o;

architecture Behavioral of register_r_h_o is
signal r_h_o_out_sig:std_logic_vector(18 downto 0);
begin
process(clk,clr)is
begin
if(clr = '1')then
        r_h_o_out_sig <= (others => '0');
elsif(clk'event and clk='1')then
        if(load = '1') then
                if(r_h_o_in(15) = '1')then
                r_h_o_out_sig(18 downto 10)<= "1111"&r_h_o_in(14 downto 10);
                r_h_o_out_sig(9 downto 0) <=  r_h_o_in(9 downto 0);
                elsif(r_h_o_in(15)='0')then
                r_h_o_out_sig(18 downto 10)<= "0000"&r_h_o_in(14 downto 10);
                r_h_o_out_sig(9 downto 0) <=  r_h_o_in(9 downto 0);
                end if;
--              rtc_out_sig(11 downto 0) <=  rtc_in(11 downto 0);
                --rtc_out_sig <= rtc_in & "00000";
        elsif(lls = '1')then
```

```
                r_h_o_out_sig <= r_h_o_out_sig(12 downto 0)&"000000";
        else
                r_h_o_out_sig <= r_h_o_out_sig;
        end if;
else
        r_h_o_out_sig <= r_h_o_out_sig;
end if;
end process;
r_h_o_out <= r_h_o_out_sig;
end Behavioral;

--VHDL source module for RHU(16-bit)which is used to store the value of
--humidity. It has store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rhu is
   Port (clk,clr,load:in std_logic;
        rhu_in:in std_logic_vector(15 downto 0);
                rhu_out:out std_logic_vector(15 downto 0));
end register_rhu;

architecture Behavioral of register_rhu is
signal rhu_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr)is
begin
        if(clr='1')then
    rhu_out_sig<=(others=>'0');
        elsif(clk'event and clk='1')then
                if(load='1')then
                rhu_out_sig<=rhu_in;
                else
                rhu_out_sig<=rhu_out_sig;
                end if;
        else
        rhu_out_sig<=rhu_out_sig;
        end if;
end process;
rhu_out <= rhu_out_sig;
end Behavioral;

--VHDL source module for RK(16-bit)which is used to store the value of
--K. It has store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rk is
   Port (clk,clr,load:in std_logic;
        rk_in:in std_logic_vector(15 downto 0);
```

```
                    rk_out:out std_logic_vector(15 downto 0));
end register_rk;

architecture Behavioral of register_rk is
signal rk_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr)is
begin
        if(clr='1')then
    rk_out_sig<=(others=>'0');
        elsif(clk'event and clk='1')then
                if(load='1')then
                rk_out_sig<=rk_in;
                else
                rk_out_sig<=rk_out_sig;
                end if;
        else
        rk_out_sig<=rk_out_sig;
        end if;
end process;
rk_out <= rk_out_sig;
end Behavioral;

--VHDL source module for RK1(16-bit)which is used to store the value of
--K1. It has store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rk1 is
    Port (clk,clr,load:in std_logic;
        rk1_in:in std_logic_vector(15 downto 0);
                rk1_out:out std_logic_vector(15 downto 0));
end register_rk1;

architecture Behavioral of register_rk1 is
signal rk1_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr)is
begin
        if(clr='1')then
    rk1_out_sig<=(others=>'0');
        elsif(clk'event and clk='1')then
                if(load='1')then
                rk1_out_sig<=rk1_in;
                else
                rk1_out_sig<=rk1_out_sig;
                end if;
        else
        rk1_out_sig<=rk1_out_sig;
        end if;
end process;
rk1_out <= rk1_out_sig;
end Behavioral;
```

```
--VHDL source module for RK1KH(16-bit)which is used to store the value of
--K1*K*H. It has store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rk1kh is
   Port (clk,clr,load:in std_logic;
        rk1kh_in:in std_logic_vector(15 downto 0);
                rk1kh_out:out std_logic_vector(15 downto 0));
end register_rk1kh;

architecture Behavioral of register_rk1kh is
signal rk1kh_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr)is
begin
        if(clr='1')then
    rk1kh_out_sig<=(others=>'0');
        elsif(clk'event and clk='1')then
                if(load='1')then
                rk1kh_out_sig<=rk1kh_in;
                else
                rk1kh_out_sig<=rk1kh_out_sig;
                end if;
        else
        rk1kh_out_sig<=rk1kh_out_sig;
        end if;
end process;
rk1kh_out <= rk1kh_out_sig;
end Behavioral;

--VHDL source module for RK2(16-bit)which is used to store the value of
--K2. It has store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rk2 is
   Port (clk,clr,load:in std_logic;
        rk2_in:in std_logic_vector(15 downto 0);
                rk2_out:out std_logic_vector(15 downto 0));
end register_rk2;

architecture Behavioral of register_rk2 is
signal rk2_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr)is
begin
        if(clr='1')then
    rk2_out_sig<=(others=>'0');
```

```
        elsif(clk'event and clk='1')then
                if(load='1')then
                rk2_out_sig<=rk2_in;
                else
                rk2_out_sig<=rk2_out_sig;
                end if;
        else
        rk2_out_sig<=rk2_out_sig;
        end if;
end process;
rk2_out <= rk2_out_sig;
end Behavioral;

--VHDL source module for RMA(16-bit)which is used to store the value of
--Ma. It has store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rma is
   Port (clk,clr,load:in std_logic;
        rma_in:in std_logic_vector(15 downto 0);
                rma_out:out std_logic_vector(15 downto 0));
end register_rma;

architecture Behavioral of register_rma is
signal rma_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr)is
begin
        if(clr='1')then
    rma_out_sig<=(others=>'0');
        elsif(clk'event and clk='1')then
                if(load='1')then
                rma_out_sig<=rma_in;
                else
                rma_out_sig<=rma_out_sig;
                end if;
        else
        rma_out_sig<=rma_out_sig;
        end if;
end process;
rma_out <= rma_out_sig;
end Behavioral;

--VHDL source module for RMB(16-bit)which is used to store the value of
--MB. It has store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rmb is
```

```
   Port (clk,clr,load:in std_logic;
        rmb_in:in std_logic_vector(15 downto 0);
                rmb_out:out std_logic_vector(15 downto 0));
end register_rmb;

architecture Behavioral of register_rmb is
signal rmb_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr)is
begin
        if(clr='1')then
    rmb_out_sig<=(others=>'0');
        elsif(clk'event and clk='1')then
                if(load='1')then
                rmb_out_sig<=rmb_in;
                else
                rmb_out_sig<=rmb_out_sig;
                end if;
        else
        rmb_out_sig<=rmb_out_sig;
        end if;
end process;
rmb_out <= rmb_out_sig;
end Behavioral;

--VHDL source module for RSORH Hat.  18-bit register. 1 5 12
--VHDL source module for 16-bit register RSOTHAT.
--The input SOT is concatenated with 2 MSB '0's and 2 LSB '0's.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity register_rsorhat is
port(clr,clk,load:in std_logic;
         sorhat_in:in std_logic_vector(13 downto 0);
         sorhat_out:out std_logic_vector(15 downto 0));
end register_rsorhat;

architecture Behavioral of register_rsorhat is
signal sorhat_out_sig: std_logic_vector(15 downto 0);
begin
        process(clk,clr) is
        begin
        if(clr = '1') then
                sorhat_out_sig <= (others => '0');
        elsif(clk'event and clk = '1') then
                if(load = '1') then
                        sorhat_out_sig(15 downto 10) <="00"& sorhat_in(13 downto 10);--
concatenation with "00" was giving warnings.
                        sorhat_out_sig(9 downto 0) <=  sorhat_in(9 downto 0);
                        --sorhat_out_sig(1 downto 0) <="00";
                else
                        sorhat_out_sig <= sorhat_out_sig;
                end if;
        else
```

```
                sorhat_out_sig <= sorhat_out_sig;
        end if;
        end process;
sorhat_out <= sorhat_out_sig;
end Behavioral;

--VHDL source module for 16-bit register RSOTHAT.
--The input SOT is concatenated with 2 MSB '0's and 2 LSB '0's.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity register_rsothat is
port(clr,clk,load:in std_logic;
        sothat_in:in std_logic_vector(13 downto 0);
        sothat_out:out std_logic_vector(15 downto 0));
end register_rsothat;

architecture Behavioral of register_rsothat is
signal sothat_out_sig: std_logic_vector(15 downto 0);
begin
        process(clk,clr) is
        begin
        if(clr = '1') then
                sothat_out_sig <= (others => '0');
        elsif(clk'event and clk = '1') then
                if(load = '1') then
                        sothat_out_sig(15 downto 14) <="00";--concatenation with "00" was giving
warnings.
                        sothat_out_sig(13 downto 0) <=  sothat_in;
                        --sothat_out_sig(1 downto 0) <="00";
                else
                        sothat_out_sig <= sothat_out_sig;
                end if;
        else
                sothat_out_sig <= sothat_out_sig;
        end if;
        end process;
sothat_out <= sothat_out_sig;
end Behavioral;

--VHDL source module for register RT (20 bits) that is used in the moisture calculating
--unit to hold the value of t It has the store, and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rt is
   Port (clk,clr,load:in std_logic;
        rtinput:in std_logic_vector(18 downto 0);
                rtoutput:out std_logic_vector(18 downto 0));
end register_rt;
```

```
architecture Behavioral of register_rt is
signal rtoutput_sig:std_logic_vector(18 downto 0);
begin
        process(clk,clr)is
                begin
                if(clr = '1')then
                        rtoutput_sig <= (others => '0');
      elsif(clk'event and clk='1')then
                        if(load = '1')then
                        rtoutput_sig <= rtinput;
                        else
                        rtoutput_sig <= rtoutput_sig;
                        end if;
      else
                        rtoutput_sig <= rtoutput_sig;
      end if;
         end process;
rtoutput <= rtoutput_sig;
end Behavioral;

--in the process of converting it to 18 bits. 1 5 12
--VHDL source module for RT2(16-bit)which is used to store the value of
--T^*T^. It has store and output functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rt2 is
   Port (clk,clr,load:in std_logic;
      rt2_in:in std_logic_vector(15 downto 0);
                rt2_out:out std_logic_vector(15 downto 0));
end register_rt2;

architecture Behavioral of register_rt2 is
signal rt2_out_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr)is
begin
        if(clr='1')then
    rt2_out_sig<=(others=>'0');
        elsif(clk'event and clk='1')then
                if(load='1')then
                rt2_out_sig<=rt2_in;
                else
                rt2_out_sig<=rt2_out_sig;
                end if;
        else
        rt2_out_sig<=rt2_out_sig;
        end if;
end process;
rt2_out <= rt2_out_sig;
end Behavioral;
--VHDL source module for register RTA (20 bits) that is used in the moisture calculating
--unit to hold the value of t/1024.  It has the store, and output functions.
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rta is
   Port (clk,clr,load,lrs:in std_logic;
        rtainput:in std_logic_vector(18 downto 0);
                rtaoutput:out std_logic_vector(18 downto 0));
end register_rta;

architecture Behavioral of register_rta is
signal rtaoutput_sig:std_logic_vector(18 downto 0);
begin
          process(clk,clr)is
                begin
                if(clr = '1')then
                        rtaoutput_sig <= (others => '0');
        elsif(clk'event and clk='1')then
                        if(load = '1')then
                        rtaoutput_sig <= rtainput;
                        elsif(lrs='1')then
                        rtaoutput_sig <= '0' & rtaoutput_sig(18 downto 1);
                        else
                        rtaoutput_sig <= rtaoutput_sig;
                        end if;
        else
                        rtaoutput_sig <= rtaoutput_sig;
        end if;
           end process;
rtaoutput <= rtaoutput_sig;
end Behavioral;
--VHDL source module for register RTC used to hold the physical value of
--temperature in celcius.
--1 8 12
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rtc is
port(      clk,clr,load,lls:in std_logic;
                rtc_in:in std_logic_vector(15 downto 0);
                rtc_out:out std_logic_vector(18 downto 0));
end register_rtc;

architecture Behavioral of register_rtc is
signal rtc_out_sig:std_logic_vector(18 downto 0);
begin
process(clk,clr)is
begin
if(clr = '1')then
          rtc_out_sig <= (others => '0');
elsif(clk'event and clk='1')then
```

```
	if(load = '1') then
		if(rtc_in(15) = '1')then
		rtc_out_sig(18 downto 10)<= "1111"&rtc_in(14 downto 10);
		rtc_out_sig(9 downto 0) <=  rtc_in(9 downto 0);
		elsif(rtc_in(15)='0')then
		rtc_out_sig(18 downto 10)<= "0000"&rtc_in(14 downto 10);
		rtc_out_sig(9 downto 0) <=  rtc_in(9 downto 0);
		end if;
--		rtc_out_sig(11 downto 0) <=  rtc_in(11 downto 0);
		--rtc_out_sig <= rtc_in & "00000";
	elsif(lls = '1')then
		rtc_out_sig <= rtc_out_sig(12 downto 0)&"000000";
	else
		rtc_out_sig <= rtc_out_sig;
	end if;
else
	rtc_out_sig <= rtc_out_sig;
end if;
end process;
rtc_out <= rtc_out_sig;
end Behavioral;
--VHDL module for register RTF used to hold the value of temperature in
--Fahreinheit.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rtf is
port(	clk,clr,load,lls:in std_logic;
		rtf_in:in std_logic_vector(15 downto 0);
		rtf_out:out std_logic_vector(18 downto 0));
end register_rtf;

architecture Behavioral of register_rtf is
signal rtf_out_sig:std_logic_vector(18 downto 0);
begin
process(clk,clr)is
begin
if(clr = '1')then
	rtf_out_sig <= (others => '0');
elsif(clk'event and clk='1')then
	if(load = '1') then
		if(rtf_in(15) = '1')then
		rtf_out_sig(18 downto 10)<= "1111"&rtf_in(14 downto 10);
		rtf_out_sig(9 downto 0) <=  rtf_in(9 downto 0);
		elsif(rtf_in(15)='0')then
		rtf_out_sig(18 downto 10)<= "0000"&rtf_in(14 downto 10);
		rtf_out_sig(9 downto 0) <=  rtf_in(9 downto 0);
		end if;
--		rtc_out_sig(11 downto 0) <=  rtc_in(11 downto 0);
		--rtc_out_sig <= rtc_in & "00000";
	elsif(lls = '1')then
		rtf_out_sig <= rtf_out_sig(12 downto 0)&"000000";
	else
```

```
                rtf_out_sig <= rtf_out_sig;
        end if;
else
        rtf_out_sig <= rtf_out_sig;
end if;
end process;
rtf_out <= rtf_out_sig;
end Behavioral;
--in the process of converting to 18 bit registers.
--VHDL source module for RT^ which is used to store the value of t/1024
--in 16-bit register. This register has shift and output storage functions.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_rthat is
   Port (clk,clr,load:in std_logic;
        rthat_in:in std_logic_vector(9 downto 0);
                rthat_out:out std_logic_vector(15 downto 0));
end register_rthat;

architecture Behavioral of register_rthat is
signal rthat_out_sig:std_logic_vector(15 downto 0);
begin
        process(clk,clr)is
        begin
                if(clr='1')then
                rthat_out_sig <= (others=>'0');
                elsif(clk'event and clk='1')then
                        if(load='1')then
                        rthat_out_sig(15 downto 10) <= "000000";
                        rthat_out_sig(9 downto 0) <=  rthat_in;
                        else
                        rthat_out_sig <= rthat_out_sig;
                        end if;
                else
                rthat_out_sig <= rthat_out_sig;
                end if;
        end process;
rthat_out <= rthat_out_sig;
end Behavioral;
--VHDL module for Interface module in EMC Calculator.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity emc_cal is
port(start_all,clr_all,data_in,clk,done_arith_in:in std_logic;
    sck,data_out_tri :out std_logic;
--new outputs:xxxxxxxxxxxxxxxxxxxx
        en_tri_out,to_ff_out,from_ff_out,clr_check_ff: out std_logic;
        clr_error_out,error_in_ff,error_out_ff:out std_logic;
```

```
            zero_count_out,hum_m_out,data_in_out,wait_m_out,measure_m_out,temp_m_out:out std_logic;
--          crc_sr_out,crc_gen_out:out std_logic_vector(7 downto 0);
            error_crc_out,wait_ok_out:out std_logic;
            letin_crc_out,load_crc_sr_out,load_sot_out:out std_logic;
--
            in13_sorh_out,in12_sorh_out,in11_sorh_out,in10_sorh_out,in9_sorh_out,in8_sorh_out,in7_sorh_
out:out std_logic;
--
            in6_sorh_out,in5_sorh_out,in4_sorh_out,in3_sorh_out,in2_sorh_out,in1_sorh_out,in0_sorh_out:o
ut std_logic;
--new outputs:xxxxxxxxxxxxxxxxxxxx
            arith_ok_f,clr_arith_out:out std_logic;
            start_arith_out:out std_logic;
            sot_out,sorh_out:out std_logic_vector(13 downto 0));
end emc_cal;

architecture Behavioral of emc_cal is
--tri state buffer used to generate DATA signal.
component tri_state is
port(data_in,enable:in std_logic;
    data_out:out std_logic);
end component tri_state;
--SOT register used to hold the temperature value.
component sot is
port(clk,clr,load:in std_logic;
--          in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13:in std_logic;
            input:in std_logic_vector(13 downto 0);
            output:out std_logic_vector(13 downto 0));
end component sot;
--SORH register used to hold the value of SORH.
component sorh is
port(clk,clr,load:in std_logic;
            in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13:in std_logic;
            output:out std_logic_vector(13 downto 0));
end component sorh;
--SHIFT REGISTER used to hold the serial input from DATA line.
component shift_reg_interface is
port(clk,sclk,lin,clr,letin,s_r_c:in std_logic;
    pout:out std_logic_vector(25 downto 0));
end component shift_reg_interface;
--INTERFACE COUNTER used for wait state while measuring.
component interface_counter is
port(clk,clr,load_counter,start_counter:in std_logic;
            zerocount:out std_logic);
end component interface_counter;
--INTERFACE CONTROLLER, the main controller for all the oerations.
component interface_controller is
port(clk,clr,start,done_tran_reset,done_temp_m,error_ff,zerocount,
            done_wait,done_measure,done_hum_m,done_arith,done_crc_reset,error_crc:in std_logic;
            start_data_sck,clr_data_sck,active_data_sck,trans_reset,
            trans_ok,temp_m,temp_ok,hum_m,hum_ok,wait_measure,wait_ok,
            clr_sr,let_in_sr,s_r_c,clr_sot,load_sot,clr_sorh,load_sorh,
            measure,measure_ok,count_o,start_arith,clr_arith,start_counter,
            crc_reset,crc_reset_ok,clr_error_ff,clr_counter,load_counter,clr_check_ff,clr_crc_gen,
    clr_crc_gen_reg,load_crc_gen_reg,clr_crc_sr_reg,load_crc_sr_reg,arithok:out std_logic);
end component interface_controller;
```

```
--DATA and SCK generator.
component data_sclk_gen_real is
port(clk,clr,start,active_data_sclk,trans_reset,temp_m,hum_m,wait_sensor,measure,from_ff,count_wait,
    trans_ok,temp_ok,wait_ok,measure_ok,hum_ok,crc_reset,crc_reset_ok:in std_logic;
            en,sclk,error_a,to_ff,done_temp,done_trans_reset,done_wait,done_measure,done_hum,done_crc_r
eset,
            letin_crc:out std_logic);
end component data_sclk_gen_real;
--CRC_SR_REG used to store the received CRC value from Shift register.
component crc_sr_reg is
port(clk,clr,load,in7,in6,in5,in4,in3,in2,in1,in0:in std_logic;
    crc_out:out std_logic_vector(7 downto 0));
end component crc_sr_reg;
--CRC_REG used to generate CRCvalue through out the transmission.
component CRC_REG is
port(data_in, sclk, clr_reg,letin_crc,clk:in std_logic;
            crc_out:out std_logic_vector(7 downto 0));
end component CRC_REG;
--CRC_GEN_REG used to hold the value of CRC from CRC generator CRC_REG.
component crc_gen_reg is
port(clk,clr,load:in std_logic;
    gen_in:in std_logic_vector(7 downto 0);
            gen_out:out std_logic_vector(7 downto 0));
end component crc_gen_reg;
--CHECK_FF
component check_ff is
port(clk,clr,to_flipflop,data_in:in std_logic;
            from_flipflop:out std_logic);
end component check_ff;
--ERROR_FF
component error_flipflop is
port(clk,clr,error_in:in std_logic;
    error_out:out std_logic);
end component error_flipflop;

signal
en_tri,clr_sot_f,load_sot_f,clr_sorh_f,load_sorh_f,sck_f,clr_sr_f,letin_sr_f,src_f,clr_counter_f:std_logic;
signal
load_counter_f,start_counter_f,zerocount_f,clr_data_sck_f,start_data_sck_f,active_data_sclk_f:std_logic;
signal
trans_reset_f,trans_ok_f,temp_m_f,hum_m_f,wait_sensor_f,measure_f,from_ff_f,count_wait_f,temp_ok_f:
std_logic;
signal wait_ok_f,measure_ok_f,hum_ok_f
,crc_reset_f,crc_reset_ok_f,error_in_f,to_ff_f,done_temp_f:std_logic;
signal done_trans_reset_f,done_wait_f,done_measure_f,done_hum_f,done_crc_reset_f,
letin_crc_f:std_logic;
signal
clr_crc_sr_reg_f,load_crc_sr_reg_f,in7_sr_f,in6_sr_f,in5_sr_f,in4_sr_f,in3_sr_f,in2_sr_f,in1_sr_f,in0_sr_f:
std_logic;
signal shiftreg_out_f:std_logic_vector(25 downto 0);
signal crc_sr_reg_out_f,crc_gen_out_f,crc_gen_reg_out_f:std_logic_vector(7 downto 0);
signal
clr_crc_gen_f,clr_crc_gen_reg_f,load_crc_gen_reg_f,clr_check_ff_f,clr_error_ff_f,error_ff_f,error_crc_f:st
d_logic;
signal input_sot_f:std_logic_vector(13 downto 0);
```

```
--signal
in0_sot,in1_sot,in2_sot,in3_sot,in4_sot,in5_sot,in6_sot,in7_sot,in8_sot,in9_sot,in10_sot,in11_sot,in12_sot,
in13_sot:std_logic;
signal
in0_sorh,in1_sorh,in2_sorh,in3_sorh,in4_sorh,in5_sorh,in6_sorh,in7_sorh,in8_sorh,in9_sorh,in10_sorh,in1
1_sorh,in12_sorh,in13_sorh:std_logic;
begin
sck <= sck_f;
TRI:tri_state port map(data_in =>'0' ,
                       enable =>en_tri,
                       data_out =>data_out_tri);
en_tri_out <= en_tri;
load_sot_out <= load_sot_f;
SO_T:sot port map(clk =>clk,
                  clr =>clr_sot_f,
                                    load =>load_sot_f,
                                           input => input_sot_f,
                                    output=>sot_out);
input_sot_f(13) <= shiftreg_out_f(2);
input_sot_f(12) <= shiftreg_out_f(3);
input_sot_f(11) <= shiftreg_out_f(4);
input_sot_f(10) <= shiftreg_out_f(5);
input_sot_f(9) <= shiftreg_out_f(6);
input_sot_f(8) <= shiftreg_out_f(7);
input_sot_f(7) <= shiftreg_out_f(9);
input_sot_f(6) <= shiftreg_out_f(10);
input_sot_f(5) <= shiftreg_out_f(11);
input_sot_f(4) <= shiftreg_out_f(12);
input_sot_f(3) <= shiftreg_out_f(13);
input_sot_f(2)<= shiftreg_out_f(14);
input_sot_f(1) <= shiftreg_out_f(15);
input_sot_f(0) <= shiftreg_out_f(16);

--input_sot_f <= shiftreg_out_f(16 downto 2);
--in0_sot <= shiftreg_out_f(2);
--in1_sot <= shiftreg_out_f(3);
--in2_sot <= shiftreg_out_f(4);
--in3_sot <= shiftreg_out_f(5);
--in4_sot <= shiftreg_out_f(6);
--in5_sot <= shiftreg_out_f(7);
--in6_sot <= shiftreg_out_f(9);
--in7_sot <= shiftreg_out_f(10);
--in8_sot <= shiftreg_out_f(11);
--in9_sot <= shiftreg_out_f(12);
--in10_sot <= shiftreg_out_f(13);
--in11_sot <= shiftreg_out_f(14);
--in12_sot <= shiftreg_out_f(15);
--in13_sot <= shiftreg_out_f(16);
--
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
--in13_sorh_out <= in13_sorh;
--in12_sorh_out <= in12_sorh;
--in11_sorh_out <= in11_sorh;
--in10_sorh_out <= in10_sorh;
--in9_sorh_out <= in9_sorh;
--in8_sorh_out <= in8_sorh;
```

```
--in7_sorh_out <= in7_sorh;
--in6_sorh_out <= in6_sorh;
--in5_sorh_out <= in5_sorh;
--in4_sorh_out <= in4_sorh;
--in3_sorh_out <= in3_sorh;
--in2_sorh_out <= in2_sorh;
--in1_sorh_out <= in1_sorh;
--in0_sorh_out <= in0_sorh;
--
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x

SO_RH:sorh port map(clk =>clk,
                clr =>clr_sorh_f,
                                        load =>load_sorh_f,
                in0 => in0_sorh,
                        in1 => in1_sorh,
                        in2 => in2_sorh,
                        in3 => in3_sorh,
                        in4 => in4_sorh,
                        in5 => in5_sorh,
                        in6 => in6_sorh,
                        in7 => in7_sorh,
                        in8 => in8_sorh,
                        in9 => in9_sorh,
                        in10 => in10_sorh,
                        in11 => in11_sorh,
                        in12 => in12_sorh,
                        in13 => in13_sorh,
                        output=>sorh_out);
in0_sorh <= shiftreg_out_f(2);
in1_sorh <= shiftreg_out_f(3);
in2_sorh <= shiftreg_out_f(4);
in3_sorh <= shiftreg_out_f(5);
in4_sorh <= shiftreg_out_f(6);
in5_sorh <= shiftreg_out_f(7);
in6_sorh <= shiftreg_out_f(9);
in7_sorh <= shiftreg_out_f(10);
in8_sorh <= shiftreg_out_f(11);
in9_sorh <= shiftreg_out_f(12);
in10_sorh <= shiftreg_out_f(13);
in11_sorh <= shiftreg_out_f(14);
in12_sorh <= shiftreg_out_f(15);
in13_sorh <= shiftreg_out_f(16);
S_R_I:shift_reg_interface port map(clk =>clk,
                        sclk =>sck_f,
                                                                lin=>data_in,
                                                                clr=>clr_sr_f,
                                                                letin=>letin_sr_f,
                                                                s_r_c=>src_f,
                        pout =>shiftreg_out_f);
--shiftreg_out_f(25 downto 18) ->CRC
--shiftreg_out_f(17) -> ack
--shiftreg_out(16 downto 9) -> measured data
--shiftreg_out(8) -> ack
--shiftreg_out(7 downto 0) -> measured data
```

```
--Measured data in order is:
--shiftreg_out(0 to 7) and (9 to 16) as sot ot sorh (15 downto 0);
I_C:interface_counter port map(clk =>clk,
                        clr =>clr_counter_f,
                                                    load_counter =>load_counter_f,
                                                    start_counter=>start_counter_f,
                                zerocount =>zerocount_f);
I_C_LR:interface_controller port map(clk =>clk,
                                clr =>clr_all,
                                                                start=>start_all,
                                                                done_tran_reset=>done_trans_reset_f,
                                                                done_temp_m=>done_temp_f,
                                                                error_ff=>error_ff_f,
                                                                zerocount=>zerocount_f,
                                        done_wait=>done_wait_f,
                                                                done_measure=>done_measure_f,
                                                                done_hum_m=>done_hum_f,
                                                                done_arith=>done_arith_in,
                                                                done_crc_reset=>done_crc_reset_f,
                                                                error_crc=>error_crc_f,
                                        start_data_sck=>start_data_sck_f,
                                                                clr_data_sck=>clr_data_sck_f,
                                                                active_data_sck=>active_data_sclk_f,
                                                                trans_reset=>trans_reset_f,
                                        trans_ok=>trans_ok_f,
                                                                temp_m=>temp_m_f,
                                                                temp_ok=>temp_ok_f,
                                                                hum_m=>hum_m_f,
                                                                hum_ok=>hum_ok_f,
                                                                wait_measure=>wait_sensor_f,
                                                                wait_ok=>wait_ok_f,
                                        clr_sr=>clr_sr_f,
                                                                let_in_sr=>letin_sr_f,
                                                                s_r_c=>src_f,
                                                                clr_sot=>clr_sot_f,
                                                                load_sot=>load_sot_f,
                                                                clr_sorh=>clr_sorh_f,
                                                                load_sorh=>load_sorh_f,
                                        measure=>measure_f,
                                                                measure_ok=>measure_ok_f,
                                                                count_o=>count_wait_f,
                                                                start_arith=>start_arith_out,
                                                                clr_arith=>clr_arith_out,
                                                                start_counter=>start_counter_f,
                                        crc_reset=>crc_reset_f,
                                                                crc_reset_ok=>crc_reset_ok_f,
                                                                clr_error_ff=>clr_error_ff_f,
                                                                clr_counter=>clr_counter_f,
                                                                load_counter=>load_counter_f,
                                                                clr_check_ff=>clr_check_ff_f,
                                                                clr_crc_gen=>clr_crc_gen_f,
                                clr_crc_gen_reg=>clr_crc_gen_reg_f,
                                                                load_crc_gen_reg=>load_crc_gen_reg_f,
                                                                clr_crc_sr_reg=>clr_crc_sr_reg_f,
                                                                load_crc_sr_reg =>load_crc_sr_reg_f,
                                                                arithok =>arith_ok_f);
```

```
zero_count_out <= zerocount_f;
hum_m_out <= hum_m_f;
wait_m_out <= wait_sensor_f;
measure_m_out <= measure_f;
temp_m_out <= temp_m_f;
wait_ok_out <= wait_ok_f;
DATA_SCK:data_sclk_gen_real port map(clk=>clk,
                        clr=>clr_data_sck_f,
                                                start=>start_data_sck_f,
                                                active_data_sclk=>active_data_sclk_f,
                                                trans_reset=>trans_reset_f,
                                                temp_m=>temp_m_f,
                                                hum_m=>hum_m_f,
                                                wait_sensor=>wait_sensor_f,
                                                measure=>measure_f,
                                                from_ff=>from_ff_f,
                                                count_wait=>count_wait_f,
                        trans_ok=>trans_ok_f,
                                                temp_ok=>temp_ok_f,
                                                wait_ok=>wait_ok_f,
                                                measure_ok=>measure_ok_f,
                                                hum_ok=>hum_ok_f,
                                                crc_reset=>crc_reset_f,
                                                crc_reset_ok=>crc_reset_ok_f,
                            en =>en_tri,
                                                sclk=>sck_f,
                                                error_a=>error_in_f,
                                                to_ff=>to_ff_f,
                                                done_temp=>done_temp_f,
                                                done_trans_reset=>done_trans_reset_f,
                                                done_wait=>done_wait_f,
                                                done_measure=>done_measure_f,
                                                done_hum=>done_hum_f,
                                                done_crc_reset=>done_crc_reset_f,
                            letin_crc=>letin_crc_f);
to_ff_out <= to_ff_f;
from_ff_out <= from_ff_f;
CRCSRREG:crc_sr_reg port map(clk =>clk,
                    clr =>clr_crc_sr_reg_f,
                                                load =>load_crc_sr_reg_f,
                                                in7=>in7_sr_f,
                                                in6=>in6_sr_f,
                                                in5=>in5_sr_f,
                                                in4=>in4_sr_f,
                                                in3=>in3_sr_f,
                                                in2=>in2_sr_f,
                                                in1=>in1_sr_f,
                                                in0=>in0_sr_f,
                    crc_out=>crc_sr_reg_out_f);
in7_sr_f <= shiftreg_out_f(25);
in6_sr_f <= shiftreg_out_f(24);
in5_sr_f <= shiftreg_out_f(23);
in4_sr_f <= shiftreg_out_f(22);
in3_sr_f <= shiftreg_out_f(21);
in2_sr_f <= shiftreg_out_f(20);
in1_sr_f <= shiftreg_out_f(19);
```

```
in0_sr_f <= shiftreg_out_f(18);

--crc_sr_out <= crc_sr_reg_out_f;
load_crc_sr_out <= load_crc_sr_reg_f;
CRCREG:CRC_REG port map(data_in =>data_in,
                sclk =>sck_f,
                                        clr_reg =>clr_crc_gen_f,
                                        letin_crc=>letin_crc_f,
                                        clk=>clk,
                        crc_out=>crc_gen_out_f);

--crc_gen_out <= crc_gen_reg_out_f;
letin_crc_out <= letin_crc_f;

CRCGRNREG:crc_gen_reg port map(clk =>clk,
                        clr =>clr_crc_gen_reg_f,
                                                        load=>load_crc_gen_reg_f,
                        gen_in=>crc_gen_out_f,
                                gen_out=>crc_gen_reg_out_f);

error_crc_f <= (crc_sr_reg_out_f(0) xor crc_gen_reg_out_f(0)) or
           (crc_sr_reg_out_f(1) xor crc_gen_reg_out_f(1)) or
                                (crc_sr_reg_out_f(2) xor crc_gen_reg_out_f(2)) or
                                (crc_sr_reg_out_f(3) xor crc_gen_reg_out_f(3)) or
                                (crc_sr_reg_out_f(4) xor crc_gen_reg_out_f(4)) or
                                (crc_sr_reg_out_f(5) xor crc_gen_reg_out_f(5)) or
                                (crc_sr_reg_out_f(6) xor crc_gen_reg_out_f(6)) or
                                (crc_sr_reg_out_f(7) xor crc_gen_reg_out_f(7));

error_crc_out <= error_crc_f;

CHECK:check_ff port map(clk=>clk,
                clr=>clr_check_ff_f,
                                        to_flipflop=>to_ff_f,
                                        data_in=>data_in,
                        from_flipflop=>from_ff_f);
data_in_out <= data_in;
clr_check_ff <= clr_check_ff_f;
ERR:error_flipflop port map(clk=>clk,
                        clr=>clr_error_ff_f,
                                                        error_in=>error_in_f,
                        error_out=>error_ff_f);
clr_error_out <= clr_error_ff_F;
error_in_ff <= error_in_f;
error_out_ff <= error_ff_f;
end Behavioral;
--VHDL source module for CHECK flip flop which is used to store the value of DATA after ack pulse
--to check for error.  When to_ff signal is asserted from DATAnSCLK generator the value of data_in
--is loaded into flpflop. otherwise it is stored.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity check_ff is
```

```
port(clk,clr,to_flipflop,data_in:in std_logic;
        from_flipflop:out std_logic);
end check_ff;

architecture Behavioral of check_ff is
signal from_flipflop_sig:std_logic;
begin
P0:process(clk,clr)is
begin
if(clr = '1')then
from_flipflop_sig <= '0';
elsif(clk'event and clk = '1')then
        if(to_flipflop = '1')then
        from_flipflop_sig <= data_in;
        else
        from_flipflop_sig <= from_flipflop_sig;
        end if;
else
        from_flipflop_sig <= from_flipflop_sig;
end if;
end process;
from_flipflop <= from_flipflop_sig;
end Behavioral;
--VHDL source module for CRC_GEN_REG used to store the values of CRC generator output.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity crc_gen_reg is
port(clk,clr,load:in std_logic;
    gen_in:in std_logic_vector(7 downto 0);
        gen_out:out std_logic_vector(7 downto 0));
end crc_gen_reg;

architecture Behavioral of crc_gen_reg is
signal gen_out_sig:std_logic_vector(7 downto 0);
begin
P0:process(clk,clr)is
begin
if(clr = '1')then
        gen_out_sig <= (others => '0');
elsif(clk'event and clk = '1')then
        if(load = '1')then
                gen_out_sig <= gen_in ;
        else
                gen_out_sig <= gen_out_sig;
        end if;
else
        gen_out_sig <= gen_out_sig;
end if;

end process;
gen_out <= gen_out_sig;
end Behavioral;
```

```
--VHDL source module for CRC generator, which replicates the hardware in sensor.  CRC check-sum is
calculated
--depending on the data received, and is compared with the CRC check-sum on the databus, and error is
--if they are not equal; In such case, Communication reset is done sending command "00011110" on DATA
bus.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity CRC_REG is
port(data_in, sclk, clr_reg,letin_crc,clk:in std_logic;
        crc_out:out std_logic_vector(7 downto 0));
end CRC_REG;

architecture Behavioral of CRC_REG is

component dff_crc_gen is
port(sclk, clr, ip, load:in std_logic;
        op:out std_logic);
end component dff_crc_gen;

signal crc_out_sig:std_logic_vector(7 downto 0);
signal data_in_sig,sa,sb:std_logic;
begin
data_in_sig <= data_in xor crc_out_sig(7);
DFF_CRC0:dff_crc_gen port map(sclk =>sclk,
                clr =>clr_reg,
                                        ip =>data_in_sig,
                                        load =>letin_crc,
                                        op =>crc_out_sig(0));
DFF_CRC1:dff_crc_gen port map(sclk =>sclk,
                clr =>clr_reg,
                                        ip =>crc_out_sig(0),
                                        load =>letin_crc,
                                        op =>crc_out_sig(1));
DFF_CRC2:dff_crc_gen port map(sclk =>sclk,
                clr =>clr_reg,
                                        ip =>crc_out_sig(1),
                                        load =>letin_crc,
                                        op =>crc_out_sig(2));
DFF_CRC3:dff_crc_gen port map(sclk =>sclk,
                clr =>clr_reg,
                                        ip =>crc_out_sig(2),
                                        load =>letin_crc,
                                        op =>crc_out_sig(3));
sa <= crc_out_sig(3) xor  data_in_sig;
DFF_CRC4:dff_crc_gen port map(sclk =>sclk,
                clr =>clr_reg,
                                        ip =>sa,
                                        load =>letin_crc,
                                        op =>crc_out_sig(4));
sb <= crc_out_sig(4) xor data_in_sig;
DFF_CRC5:dff_crc_gen port map(sclk =>sclk,
                clr =>clr_reg,
```

```
                                                ip =>sb,
                                                load =>letin_crc,
                                                op =>crc_out_sig(5));
DFF_CRC6:dff_crc_gen port map(sclk =>sclk,
                clr =>clr_reg,
                                                ip =>crc_out_sig(5),
                                                load =>letin_crc,
                                                op =>crc_out_sig(6));
DFF_CRC7:dff_crc_gen port map(sclk =>sclk,
                clr =>clr_reg,
                                                ip =>crc_out_sig(6),
                                                load =>letin_crc,
                                                op =>crc_out_sig(7));

process(clk)is
begin
if(clk'event and clk = '1')then
        crc_out <= crc_out_sig;
end if;
end process;
end Behavioral;

--VHDL source module for the D-Flip Flop used in CRC generator. This flipflop is used to store the 7th bit
in
--CRC generator.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity dff_crc_gen is
port(sclk, clr, ip, load:in std_logic;
        op:out std_logic);
end dff_crc_gen;

architecture Behavioral of dff_crc_gen is
signal op_sig:std_logic;
begin
P0:process(sclk, clr)is
begin
if(clr = '1')then
                op_sig <= '0';

elsif(sclk'event and sclk = '1')then
                if(load = '1')then
                op_sig <= ip;
        else
                op_sig <= op_sig;

        end if;
else
        op_sig <= op_sig;
end if;
end process;
```

```
op <= op_sig;
end Behavioral;
--VHDL source module for CRC_SR register used to store the value of CRC from SR register.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity crc_sr_reg is
port(clk,clr,load,in7,in6,in5,in4,in3,in2,in1,in0:in std_logic;
    crc_out:out std_logic_vector(7 downto 0));

end crc_sr_reg;

architecture Behavioral of crc_sr_reg is
signal crc_out_sig:std_logic_vector(7 downto 0);
begin
P0:process(clk,clr)is
begin
if(clr = '1')then
            crc_out_sig <= (others => '0');
elsif(clk'event and clk = '1')then
            if(load = '1')then
                        crc_out_sig(0) <= in0;
                        crc_out_sig(1) <= in1;
                        crc_out_sig(2) <= in2;
                        crc_out_sig(3) <= in3;
                        crc_out_sig(4) <= in4;
                        crc_out_sig(5) <= in5;
                        crc_out_sig(6) <= in6;
                        crc_out_sig(7) <= in7;
            else
                        crc_out_sig <= crc_out_sig;
            end if;
else
            crc_out_sig <= crc_out_sig;
end if;
end process;
crc_out <= crc_out_sig;
end Behavioral;
--VHDL source module for a counter used in interface module, which is used to produce
--'zerocount' signal to the controller, based on which the 'wait' statement in interface
--ends.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity interface_counter is
port(clk,clr,load_counter,start_counter:in std_logic;
            zerocount:out std_logic);
end interface_counter;
```

```
architecture Behavioral of interface_counter is
signal counter_out:std_logic_vector(7 downto 0);
begin
--counter_out <= "1100000110110010"; 1010000000110111101
P0:process(clk,clr) is
begin
if(clr = '1')then
        counter_out <= (others => '0');
elsif(clk'event and clk = '1')then
        if(load_counter = '1')then
        counter_out <= "11010010";
        elsif(start_counter = '1')then
                counter_out <= counter_out - 1;
        end if;
else
        counter_out <= counter_out;
end if;
end process;
P1:process(counter_out) is
begin
if(counter_out = "00000000")then
        zerocount <= '1';
else
        zerocount <= '0';
end if;
end process;

end Behavioral;
--VHDL source module for shift register used in the interface module. This shift register gets
--the value of serial input 'lin' at the rising edge of SCK.
--Sends the value of parallel output 'pout' at the rising edge of 'CLK'.
--clk -- clock
--sck -- SCK
--lin -- serial left input
--clr -- asynchronous clear input to the register.
--letin--control input to the register to let the datain serially thru 'lin' port.
--s_r_c--control input to select whether the operation is for SCLK or CLK.


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity shift_reg_interface is
port(clk,sclk,lin,clr,letin,s_r_c:in std_logic;
    pout:out std_logic_vector(25 downto 0));
end shift_reg_interface;

architecture Behavioral of shift_reg_interface is
signal p_sig,q_sig:std_logic_vector(25 downto 0);
begin

P0:process(sclk,clr,s_r_c,clk,p_sig,q_sig)is
begin
```

```
if(clr = '1')then
		q_sig <= (others => '0');
		p_sig <= (others => '0');
elsif(s_r_c = '0')then
		if(sclk'event and sclk='1')then
			if(letin= '1')then
				q_sig <= lin&q_sig(25 downto 1);
			else
				q_sig <= q_sig;
			end if;
		end if;
		--if(letout = '1')then
	--	if(sck = '0')then
		--		lout_sig <= q_sig(15);
		--		q_sig <= q_sig(14 downto 0)&'1';--left shifted with bit '1'. Reason: to check the first bit
whether it goes to '0' or not when sensor drives it.
		--		else
		--		lout_sig <= lout_sig;
		--		q_sig <= q_sig;
		--		end if;
		--end if;
elsif(s_r_c = '1')then
		if(clk'event and clk = '1')then
			p_sig <= q_sig;
		else
			p_sig <= p_sig;
		end if;
else
		q_sig <= q_sig;
		p_sig <= p_sig;
end if;


end process;
pout <= p_sig;
end Behavioral;
--it is better to include delay states in the controller when switching between pload, letin and letout.
--VHDL
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;
--16 15 14 13 12 11 10 9 |8| 7 6 5 4 3 2 1 0
entity sorh is
port(clk,clr,load:in std_logic;
		in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13:in std_logic;
		output:out std_logic_vector(13 downto 0));

end sorh;

architecture Behavioral of sorh is
```

```
signal output_sig,outputsig:std_logic_vector(13  downto 0);
begin
                        outputsig(13) <= in0;
                        outputsig(12) <= in1;
                        outputsig(11) <= in2;
                        outputsig(10) <= in3;
                        outputsig(9) <= in4;
                        outputsig(8) <= in5;
                        outputsig(7) <= in6;
                        outputsig(6) <= in7;
                        outputsig(5) <= in8;
                        outputsig(4) <= in9;
                        outputsig(3) <= in10;
                        outputsig(2) <= in11;
                        outputsig(1) <= in12;
                        outputsig(0) <= in13;

P0:process(clk,clr)is
begin
if(clr = '1') then
            output_sig <= (others => '0');
elsif(clk'event and clk = '1')then
            if(load = '1') then
                        output_sig <= outputsig;
            else
                        output_sig  <= output_sig;
            end if;
else
            output_sig <= output_sig;
end if;
end process;
output <= output_sig;
end Behavioral;
--VHDL source module for temporary register of width 17 bits, which holds the
--value of SOT, its input is from the shift register.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;
--16 15 14 13 12 11 10 9 |8| 7 6 5 4 3 2 1 0
entity sot is
port(clk,clr,load:in std_logic;
--          in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13:in std_logic;
            input:in std_logic_vector(13 downto 0);
            output:out std_logic_vector(13 downto 0));

end sot;

architecture Behavioral of sot is
signal output_sig:std_logic_vector(13  downto 0);
begin
```

```
--outputsig <= in0 & in1 & in2 & in3 & in4 & in5 & in6 & in7 & in8 & in9 & in10 & in11 & in12 & in13;
--                      outputsig(13) <= in0;
--                      outputsig(12) <= in1;
--                      outputsig(11) <= in2;
--                      outputsig(10) <= in3;
--                      outputsig(9) <= in4;
--                      outputsig(8) <= in5;
--                      outputsig(7) <= in6;
--                      outputsig(6) <= in7;
--                      outputsig(5) <= in8;
--                      outputsig(4) <= in9;
--                      outputsig(3) <= in10;
--                      outputsig(2) <= in11;
--                      outputsig(1) <= in12;
--                      outputsig(0) <= in13;

P0:process(clk,clr)is
--variable outputsig:std_logic_vector(13 downto 0);
begin

if(clr = '1')then
output_sig <= "00000000000000";
elsif(clk'event and clk = '1')then
            if(load = '1') then
--outputsig := in0 & in1 & in2 & in3 & in4 & in5 & in6 & in7 & in8 & in9 & in10 & in11 & in12 & in13;
                        output_sig  <= input;
--          end if;
--else
--          output_sig <= output_sig;
            end if;
end if;
end process;
output <= output_sig;
end Behavioral;
--VHDL source module for tri-state buffer used in interface module.
--The port 'lout' from the shift register will be the input to a mux, which
--sets the value of enable high when lout = '0', else makes enable low.
--the input to the tristate buffer is always low.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity tri_state is
port(data_in,enable:in std_logic;
    data_out:out std_logic);

end tri_state;

architecture Behavioral of tri_state is

begin
P0:process(enable, data_in)is
begin
```

```
if(enable = '0')then
            data_out <= data_in;
else
            data_out <= 'Z';
end if;
end process;

end Behavioral;
--VHDL MODULE for multiplexor used to choose the value of temperature between
--centigrade and humidities.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity mux_2x1_temp is
port(sel:in std_logic;
    a,b:in std_logic_vector(18 downto 0);
            c:out std_logic_vector(18 downto 0));
end mux_2x1_temp;

architecture Behavioral of mux_2x1_temp is

begin
P0:process(sel,a,b)is
begin
if(sel = '0')then
c <= a;
else
c <= b;
end if;
end process;


end Behavioral;
--VHDL source module that selects inputs for the display unit.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity mux_display_unit is
port(sel:in std_logic;
    a,b:in std_logic_vector(18 downto 0);
            c:out std_logic_vector(18 downto 0));
end mux_display_unit;

architecture Behavioral of mux_display_unit is

begin
process(sel,a,b)is
begin
if(sel = '0')then
c <= a;
```

```
else
c <= b;
end if;
end process;

end Behavioral;
```

Appendix 1-2: VHDL Description of the EMC Processor serial arithmetic unit:

```
--VHDL description of the serial arithmetic unit used in EMC processor with the serial arithmetic unit.
-- VHDL source module used for n-bit ripple carry adder/subtractor

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity add_sub_adder is
--generic(n:positive:=32);
port(a_input,b_input: in std_logic_vector(15 downto 0);
        sel_cin,sel_bin:in std_logic;
        adder_out: out std_logic_vector(15 downto 0);
        carry_out:out std_logic;
        overflow:out std_logic);
end add_sub_adder;

architecture structural of add_sub_adder is
component bitadder_adder is          -- component declaration of onebit adder
port(a_in,b_in,sel_b,c_in:in std_logic;
        sum_bfa, cout:out std_logic);
end component bitadder_adder;

component mux_2X1a_adder is      --  component declaration of 2X1 multiplexor
port(a,b:in std_logic;
        sel : in std_logic;
        c: out std_logic);
end component mux_2X1a_adder;
signal c_in_first: std_logic;
signal carry_internal: std_logic_vector(15 downto 0);
begin

A0: mux_2X1a_adder port map(a=>'0',b=>'1',sel=>sel_cin,c=>c_in_first);
B0:bitadder_adder port map(a_in=>a_input(0),b_in=>
b_input(0),sel_b=>sel_bin,c_in=>c_in_first,sum_bfa=>adder_out(0),cout => carry_internal(0));

B1TON: for i in 1 to 15 generate

        Bi: bitadder_adder port map(a_in=>a_input(i),b_in=>
b_input(i),sel_b=>sel_bin,c_in=>carry_internal(i-1),sum_bfa=>adder_out(i),cout => carry_internal(i));

        end generate;
--2's complement overflow occurs when the carry into the MSB is not equal to
```

```
--the carry out from the MSB. This is detected using XOR gate.
overflow <= carry_internal(15) xor carry_internal(14);
carry_out <= carry_internal(15);
end structural;
--VHDL source module for adder result.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity adder_result is
port(clr,clk,load:in std_logic;
        adder_in:in std_logic_vector(15 downto 0);
        adder_out:out std_logic_vector(15 downto 0));
end adder_result;

architecture Behavioral of adder_result is
signal adder_out_sig:std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1')then
        adder_out_sig <= (others=>'0');
elsif(clk'event and clk='1')then
        if(load ='1')then
                adder_out_sig <= adder_in;
    else
                adder_out_sig <= adder_out_sig;
        end if;
else
        adder_out_sig <= adder_out_sig;
end if;
end process;
adder_out <= adder_out_sig;
end Behavioral;

-- Counter of width 'm' used in Booth's multiplier.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity counter_re is
port(clk,load:in std_logic;
        count_ctrl:in std_logic;
        zero:out std_logic);
end counter_re;

architecture Behavioral of counter_re is
signal count: std_logic_vector(4 downto 0);
begin
process(clk) is
begin
```

```
if(clk'event and clk='1') then
        if(load ='1') then
        count <= "01111";
        elsif(count_ctrl = '1') then
        count <= count - 1;
                if(count = "00000")then
                zero <= '1';
                else zero <= '0';
                end if;
        elsif(count_ctrl = '0') then
        count <= count;
        end if;
end if;
end process;
end Behavioral;

--VHDL source code of 1 bit adder/subtractor.


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity bitadder_adder is
port(a_in,b_in,sel_b,c_in:in std_logic;
        sum_bfa, cout:out std_logic);
end bitadder_adder;

architecture Behavioral of bitadder_adder is
component mux_2X1a_adder is
port(a,b:in std_logic;
        sel : in std_logic;
        c: out std_logic);
end component mux_2X1a_adder;

component bfa_adder is
port(a,b,cin: in std_logic;
        sum, carryout: out std_logic);
end component bfa_adder;
signal bout,b_in_bar:std_logic;
begin
b_in_bar <= not(b_in);

M0: mux_2X1a_adder port map(a=>b_in, b=>b_in_bar,sel=>sel_b,c=>bout);
--M1: mux_2X1_a port map(a=>'0',b=>'1',sel=>add_sub,c=>cin);            \
-- If sel_b is '0' then b_in is selected else not(b_in) is selected.
B0: bfa_adder port map(a=>a_in,b=>bout,cin=>c_in,sum=>sum_bfa,carryout=>cout);

end Behavioral;

-- VHDL source module of binary full adder used in adder/subtractor.

library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity bfa_adder is
port(a,b,cin: in std_logic;
            sum, carryout: out std_logic);
end bfa_adder;

architecture Behavioral of bfa_adder is
signal input:std_logic_vector(2 downto 0);
begin
input <= a&b&cin;
 process(a,b,cin,input) is
 begin
if (input = "000") then sum <= '0';carryout <= '0';
elsif (input = "001") then sum <= '1';carryout <= '0';
elsif (input = "010") then sum <= '1';carryout <= '0';
elsif (input = "011") then sum <= '0';carryout <= '1';
elsif (input = "100") then sum <= '1';carryout <= '0';
elsif (input = "101") then sum <= '0';carryout <= '1';
elsif (input = "110") then sum <= '0';carryout <= '1';
else  sum <= '1';carryout <= '1';
end if;

 end process;


end Behavioral;

-- VHDL code for n-bit input 2X1 multiplexor used in the design of Booth's multiplier.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity mux_2X1a_adder is
port(a,b:in std_logic;
            sel : in std_logic;
            c: out std_logic);
end mux_2X1a_adder;

architecture Behavioral of mux_2X1a_adder is

begin
process(sel,a,b) is
begin
if (sel = '0') then
            c <= a;
else
            c <= b;
end if;
end process;
```

```
end Behavioral;

--VHDL source module for adder result.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity adder_result is
port(clr,clk,load:in std_logic;
        adder_in:in std_logic_vector(15 downto 0);
        adder_out:out std_logic_vector(15 downto 0));
end adder_result;

architecture Behavioral of adder_result is
signal adder_out_sig:std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1')then
        adder_out_sig <= (others=>'0');
elsif(clk'event and clk='1')then
        if(load ='1')then
                adder_out_sig <= adder_in;
    else
                adder_out_sig <= adder_out_sig;
        end if;
else
        adder_out_sig <= adder_out_sig;
end if;
end process;
adder_out <= adder_out_sig;
end Behavioral;

--done.
--in the process of 16-bit changes.
-- VHdl source module to convert a twos complement number to an unsigned binary number.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity conver is
port(a_in: in std_logic_vector(15 downto 0);
        bout: out std_logic_vector(15 downto 0));
        --sign:out std_logic);
end conver;



architecture Behavioral of conver is

component bfa_divider is
```

```
port(in_a,in_b,in_c:in std_logic;
        sum_a,cout_a:out std_logic);
end component bfa_divider;
signal b_internal,b_out,carry_int:std_logic_vector(14 downto 0);

begin
--process(a_in) is
--begin
--sign <= a_in(20);
b_out <= not(a_in(14 downto 0));

H0: bfa_divider port
map(in_a=>b_out(0),in_b=>'0',in_c=>'1',sum_a=>b_internal(0),cout_a=>carry_int(0));
H1toN: for i in 1 to 14 generate
HI: bfa_divider port map(in_a=>b_out(i),in_b=>'0',in_c=>carry_int(i-
1),sum_a=>b_internal(i),cout_a=>carry_int(i));
end generate;

-- process statement to choose output.
process(a_in,b_internal) is
begin
if(a_in(15)='1')then
 bout <= '0' & b_internal;
else
 bout <= '0' & a_in(14 downto 0);
end if;
end process;


end Behavioral;
-- VHDL source module to convert a twos complement number to an unsigned binary number.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity conver2 is
port(a_in: in std_logic_vector(14 downto 0);
        bout: out std_logic_vector(15 downto 0);
        sign:in std_logic);
end conver2;


architecture Behavioral of conver2 is

component bfa_divider is
port(in_a,in_b,in_c:in std_logic;
        sum_a,cout_a:out std_logic);
end component bfa_divider;
signal b_internal,b_out,carry_int:std_logic_vector(14 downto 0);

begin
--process(a_in) is
--begin
```

```
b_out <= not(a_in(14 downto 0));

H0: bfa_divider port
map(in_a=>b_out(0),in_b=>'1',in_c=>'0',sum_a=>b_internal(0),cout_a=>carry_int(0));
H1toN: for i in 1 to 14 generate
HI: bfa_divider port map(in_a=>b_out(i),in_b=>'0',in_c=>carry_int(i-
1),sum_a=>b_internal(i),cout_a=>carry_int(i));
end generate;
--end process;
--b_out_out <= b_out;
process(a_in,sign,b_internal)is
begin
if(sign = '1') then
        bout <= sign&b_internal;
else
        bout <= sign&a_in;
end if;
end process;
end Behavioral;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity final_register is
port(clk,clr,load_result:in std_logic;
        result_in:in std_logic_vector(15 downto 0);
        result_out:out std_logic_vector(15 downto 0));
end final_register;

architecture Behavioral of final_register is
signal result_out_sig: std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1') then
        result_out_sig <= (others=>'0');
                elsif(clk'event and clk='1') then
                if(load_result ='1')then
                        result_out_sig <= result_in;
                else
                        result_out_sig <= result_out_sig;
                end if;
else
result_out_sig <= result_out_sig;
end if;
end process;
result_out <= result_out_sig;
end Behavioral;
-- VHDL source code for final register used to store the final result.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity finalregister_mul is
port(clk,clr,load:in std_logic;
        a:in std_logic_vector(15 downto 0);
        f: out std_logic_vector(15 downto 0);
        or_out:out std_logic);
end finalregister_mul;

architecture Behavioral of finalregister_mul is
signal f_sig: std_logic_vector(15 downto 0);
begin
process(clk,clr) is
begin
if(clr='1') then
        f_sig <=(others =>'0');
elsif(clk'event and clk = '1') then
        if(load='1')then
        f_sig(15 downto 0) <= a;
        else
        f_sig <= f_sig;
        end if;
end if;
end process;
f <= f_sig;
or_out <=  f_sig(15) or f_sig(14) or f_sig(13) or
                f_sig(12) or f_sig(11) or f_sig(10) or
                f_sig(9) or  f_sig(8) or  f_sig(7) or
                f_sig(6) or  f_sig(5) or  f_sig(4) or
                f_sig(3) or  f_sig(2) or  f_sig(1) or f_sig(0);
end Behavioral;
 --f_sig(20) or f_sig(19) or f_sig(18) or
                --f_sig(17) or f_sig(16) or f_sig(17) or f_sig(16) or
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity mux_2x1_1 is
port(sel:in std_logic;
        a,b:in std_logic;
        c:out std_logic);
end mux_2x1_1;

architecture Behavioral of mux_2x1_1 is

begin
process(sel,a,b)is
begin
if(sel = '0')then
        c <= a;
else
        c <= b;
end if;
end process;
end Behavioral;
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity mux_2x1_16 is
port(sel:in std_logic;
        a,b:in std_logic_vector(15 downto 0);
        c:out std_logic_vector(15 downto 0));
end mux_2x1_16;

architecture Behavioral of mux_2x1_16 is

begin
process(sel,a,b)is
begin
if(sel = '0')then
        c <= a;
else
        c <= b;
end if;
end process;
end Behavioral;
-- VHDL source module for quotient register that stores the value of quotient.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity quotient_reg_re is
port(quotient_in:in std_logic_vector(14 downto 0);
        quotient_out:out std_logic_vector(14 downto 0);
        clk,clr,load_overflow,load_q_zero,load_quotient:in std_logic);
end quotient_reg_re;

architecture Behavioral of quotient_reg_re is
signal quotient_out_sig: std_logic_vector(14 downto 0);
begin
process(clk,clr)is
begin
if(clr='1')then
        quotient_out_sig <= (others=>'0');
        elsif(clk'event and clk='1') then
                if(load_overflow = '1') then
                        quotient_out_sig <= (others => '1');
                elsif(load_q_zero = '1') then
                        quotient_out_sig <= "000000000000001";
                elsif(load_quotient ='1')then
                        quotient_out_sig <= quotient_in;
                end if;
end if;
end process;
quotient_out <= quotient_out_sig;
```

```
end Behavioral;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_a is
port(clr,clk,load:in std_logic;
            a_in:in std_logic_vector(15 downto 0);
            a_out:out std_logic_vector(15 downto 0);
            sign_a:out std_logic);
end register_a;

architecture Behavioral of register_a is
signal a_out_sig:std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1')then
            a_out_sig <= (others=>'0');
elsif(clk'event and clk='1')then
            if(load ='1')then
                        a_out_sig <= a_in;
    else
                        a_out_sig <= a_out_sig;
            end if;
else
            a_out_sig <= a_out_sig;
end if;
end process;
a_out <= a_out_sig;
sign_a<= a_out_sig(15);
end Behavioral;
--VHDL source module for register B
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_b is
port(clr,clk,load:in std_logic;
            b_in:in std_logic_vector(15 downto 0);
            b_out:out std_logic_vector(15 downto 0));
end register_b;

architecture Behavioral of register_b is
signal b_out_sig:std_logic_vector(15 downto 0);
begin
process(clr,clk) is
begin
if(clr='1')then
            b_out_sig <= (others=>'0');
elsif(clk'event and clk='1')then
            if(load ='1')then
```

```
                b_out_sig <= b_in;
    else
                b_out_sig <= b_out_sig;
        end if;
else
        b_out_sig <= b_out_sig;
end if;
end process;
b_out <= b_out_sig;
end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity register_c is
   Port (a_b_bar:in std_logic_vector(15 downto 0);
                reset_rc,load_rc,clk,c_o_u_t:in std_logic;
                regc_out:out std_logic_vector(15 downto 0);
                msb:out std_logic );
end register_c;

architecture Behavioral of register_c is
signal regc_out_sig:std_logic_vector(15 downto 0);
signal msb_sig: std_logic;

begin

process(clk,reset_rc)is
begin
if(reset_rc = '1') then
        regc_out_sig <= (others=>'0');
        msb_sig <= '0';
elsif(clk'event and clk='1')then
        if(load_rc='1')then
                regc_out_sig <= a_b_bar;
                msb_sig <= c_o_u_t;
        else
                regc_out_sig <= regc_out_sig;
                msb_sig <= msb_sig;
        end if;
end if;
end process;
regc_out <= regc_out_sig;
msb <= msb_sig;
end Behavioral;
-- VHDL source module for register used to store the 21 bit dividend.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity register_dividend is
port(clk,load:in std_logic;
        divin: in std_logic_vector(15 downto 0);
        divout: out std_logic_vector(15 downto 0);
        msb_dividend: out std_logic);
end register_dividend;

architecture Behavioral of register_dividend is
signal divout_sig:std_logic_vector(15 downto 0);
begin
process(clk) is
begin
if(clk'event and clk='1') then
        if (load ='1') then
                divout_sig <= divin;
        else
                divout_sig <= divout_sig;
        end if;
else
        divout_sig <= divout_sig;
end if;
end process;
divout <= divout_sig;
msb_dividend <= divout_sig(15);
end Behavioral;
-- VHDL source module for register used to store the 21 bit divisor.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_divisor is
port(clk,load:in std_logic;
        divisin: in std_logic_vector(15 downto 0);
        divisout: out std_logic_vector(15 downto 0);
        msb_divisor: out std_logic);
end register_divisor;

architecture Behavioral of register_divisor is
signal divisout_sig:std_logic_vector(15 downto 0);
begin
process(clk) is
begin
if(clk'event and clk='1') then
        if (load ='1') then
                divisout_sig <= divisin;
        else
                divisout_sig <= divisout_sig;
        end if;
else
        divisout_sig <= divisout_sig;
end if;
end process;
divisout <= divisout_sig;
msb_divisor <= divisout_sig(15);
```

```
end Behavioral;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity register_e_f is
port(clr_e_f,load_e_f,clk,load_e,lls,ars,or_out:in std_logic;
        e_in,f_in:in std_logic_vector(15 downto 0);
        ef_out:out std_logic_vector(31 downto 0);
        lsb:out std_logic;
        e_out:out std_logic_vector(15 downto 0);
        or_out_div: out std_logic);
end register_e_f;

architecture Behavioral of register_e_f is
signal ef_out_sig:std_logic_vector(31 downto 0);
begin
process(clk,clr_e_f)is
begin
if(clr_e_f = '1')then
ef_out_sig <= (others => '0');
elsif(clk'event and clk='1')then
        if(load_e_f = '1')then
        ef_out_sig(31 downto 16) <= (others => '0');
        ef_out_sig(15 downto 0) <= f_in;
        elsif(load_e = '1')then
        ef_out_sig(31 downto 16) <= e_in;
        ef_out_sig(0) <= or_out;
        elsif(lls = '1')then
        ef_out_sig <= ef_out_sig(30 downto 0)&'0';
        elsif(ars = '1')then
        ef_out_sig <= ef_out_sig(31)& ef_out_sig(31 downto 1);
        else
        ef_out_sig <= ef_out_sig;
        end if;
end if;
end process;
ef_out <= ef_out_sig;
lsb <= ef_out_sig(0);
e_out <= ef_out_sig(31 downto 16);
or_out_div <= ef_out_sig(14) or ef_out_sig(13) or ef_out_sig(12) or ef_out_sig(11) or
                        ef_out_sig(10) or ef_out_sig(9) or ef_out_sig(8) or ef_out_sig(7) or
                        ef_out_sig(6) or ef_out_sig(5) or ef_out_sig(4) or ef_out_sig(3) or
                        ef_out_sig(2) or ef_out_sig(1) or ef_out_sig(0);
end Behavioral;
--VHDL source module for result register used to store
--the result of multiplication.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity result_register_mul is
```

```
port(a:in std_logic_vector(15 downto 0);
        b:out std_logic_vector(15 downto 0);
        clk,clr_result,load_result:in std_logic);
end result_register_mul;

architecture Behavioral of result_register_mul is
signal b_sig:std_logic_vector(15 downto 0);
begin
process(clk,clr_result) is
begin
if(clr_result ='1') then
        b_sig <= (others => '0');
elsif(clk'event and clk='1') then
        if(load_result = '1')then
        b_sig <= a;
        else
        b_sig <= b_sig;
        end if;
else
b_sig <= b_sig;
end if;
end process;
b <= b_sig;
end Behavioral;
```

Appendix 2: Implementation Constraints File (.ucf) of the EMC Processor

```
NET "clk"LOC = "P80";
NET "start_final" LOC = "P135";
NET "sw1" LOC = "P126";
NET "sw2" LOC = "P129";
NET "c_r_f" LOC = "P133";
NET "clear_final" LOC = "P138";
NET "datain" LOC = "P173";
NET "sck" LOC = "P168";
NET "dataout" LOC = "P175";
NET "an1" LOC = "P160";
NET "an2" LOC = "P162";
NET "an3" LOC = "P164";
NET "an4" LOC = "P166";
NET "ca" LOC = "P127";
NET "cb" LOC = "P132";
NET "cc" LOC = "P134";
NET "cd" LOC = "P136";
NET "ce" LOC = "P139";
NET "cf" LOC = "P141";
NET "cg" LOC = "P146";
NET "entriout"LOC="P179";
NET "letincrcout"LOC="P115";
NET"donearith_out"LOC = "P120";
NET"clrerrorout"LOC="P121";
NET"errorinff"LOC="P122";
NET"erroroutf"LOC="P123";
NET"zerocountout"LOC="P154";
NET"hummout"LOC="P167";/*led5*/
NET"waitmout"LOC="P165";/*led4*/
NET"measuremout"LOC="P163";/*led3*/
NET"tempmout"LOC="P169"; /*led6*/
NET"loadtempout"LOC="P116";
NET"dp_final"LOC="P148";
```

References:

1. H.E. Desch and J.M. Dinwoodie, “Timber, Structure, Properties, Conversion and Use”, Food Products Press, pp-81-89, July 1996.

2. Learning resources, Moisture and shrinkage
   http://www.timber.org.au/NTEP/menu.asp?id=82

3. William T. Simpson, 1998, “Equilibrium moisture content of wood in outdoor locations in the United States and worldwide”, Res. Note FPL-RN-0268. Madison, WI: U.S. Department of Agriculture, Forest Service, Forest Products Laboratory. 11p.

4. Anonymous. 1999. Wood Handbook: Wood as an engineering material. USDA Forest Service, Forest Products Laboratory General Technical Report FPL-GTR-113.

5. Application notes, Introduction to Relative humidity
   http://www.sensirion.com/en/pdf/App_Note_Intro_to_Rel_Humidity.pdf

6. Choosing a humidity sensor
   http://www.sensorsmag.com/articles/0701/54/main.shtml

7. SHT1x/SHT7x Humidity and Temperature sensor
   http://www.sensirion.com/en/pdf/Datasheet_SHT1x_SHT7x.pdf

8. A painless guide to CRC error detection algorithms
   http://www.repairfaq.org/filipg/LINK/F_crc_v3.html

9. Application note CRC
   http://www.sensirion.com/en/pdf/CRC_Calculation_SHTxx_v1.03.pdf

10. Behrooz Parhami, “Computer Arithmetic, Algorithms and Hardware designs”, Oxford University Press, 2000.

11. Norman R. Scott, “Computer Number Systems and Arithmetic”, Prentice-Hall, Inc., 1985.

12. IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985), IEEE Press, 1985.

13. C .Inacio, D.Ombres, The DSP decision: fixed point or floating? Spectrum, IEEE, Volume: 33, Issue: 9, Sept. 1996, Pages: 72 – 74.

14. R. Henning and C. Chakrabarti, “Activity Models for use in low power, high level synthesis”, Proc. of the International Conference on Acoustics, Speech and Signal Processing, 1999.

15. Michael J. Flynn and Stuart F. Oberman, “Advanced Computer Arithmetic Design”, John Wiley & Sons, Inc. March 2001.

16. Douglas J Smith, “HDL Chip Design, A Practical Guide for Designing, Synthesizing & Simulating ASICs & FPGAs Using Vhdl or Verilog”, Doone Publications, Madison AL, March 1998.

17. A.D.Booth, “A Signed Binary Multiplication Technique”, Quarterly J. Mechanics and Applied Mathematics, Vol. 4, Pt. 2, pp. 236-240, June 1951.

18. Michael J. Flynn and Stuart F. Oberman, “Division Algorithms and Implementations”, IEEE Transactions on Computers, Vol. 46, No. 8, pp. 833-854, August 1997.

19. Steve Golson, “One-hot state machine design for FPGAs”, 3rd PLD Design Conference, March 1993. http://www.trilobyte.com/pdf/golson_pldcon93.pdf

20. FPGA design flow overview http://toolbox.xilinx.com/docsan/xilinx6/help/iseguide/html/ise_fpga_design_flow_overview.htm.

21. Digilab 2E system board reference manual www.digilentinc.com.

Vita

Author's name: Phani Tangirala

Birthplace: Nellore, India

Birth date: August 28, 1980.

Education:

Bachelor of Technology in Electronics and Communications
Jawaharlal Nehru Technological University
June 2001.