LOW POWER DESIGN IMPLEMENTATION OF A SIGNAL ACQUISITION MODULE

by

RAVI BHUSHAN THAKUR

B.Tech, Jawaharlal Nehru Technological University, INDIA 2007

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2010

Approved by:

Major Professor
Dr. Don Gruenbacher

Abstract

As semiconductor technologies advance, the smallest feature sizes that can be fabricated get smaller. This has led to the development of high density FPGAs capable of supporting high clock speeds, which allows for the implementation of larger more complex designs on a single chip. Over the past decade the technology market has shifted toward mobile devices with low power consumption at or near the top of design considerations. By reducing power consumption in FPGAs we can achieve greater reliability, lower cooling cost, simpler power supply and delivery, and longer battery life.

In this thesis, FPGA technology is discussed for the design and commercial implementation of low power systems as compared to ASICs or microprocessors, and a few techniques are suggested for lowering power consumption in FPGA designs. The objective of this research is to implement some of these approaches and attempt to design a low power signal acquisition module.

Designing for low power consumption without compromising performance requires a power-efficient FPGA architecture and good design practices to leverage the architectural features. With various power conservation techniques suggested for every stage of the FPGA design flow, the following approach was used in the design process implementation: the switching activity is addressed in the design entry, and synthesis level and software tools are utilized to get an initial estimate of and optimize the design's power consumption. Finally, the device choice is made based on its features that will enhance the optimization achieved in the previous stages; it is configured and real time board level power measurements are made to verify the implementation's efficacy

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to take this opportunity to express my gratitude to Dr. Don Gruenbacher, Dr William Kuhn and Dr Andrew Rys for their inspiration and constant support throughout my Masters. I would like to thank Dr Gruenbacher for his patience and guidance that made the completion of this work possible. Finally, I would like to thank my family and friends for their unrelenting support and encouragement.

Chapter 1

Introduction

A Field Programmable Gate Array, FPGA, is a digital device that is hardware programmable by the user so that a specific design can be configured for a task. Depending on the device used, the design can be either burned temporarily, semi-permanently or loaded from an external memory every time during device power up. FPGAs have found immense application potential in fields of communication, computing and consumer electronics due to the design flexibility that it provides. The FPGA platform allows for the design to be modified even after the system has been manufactured. This is a highly desirable trait for applications that need to have a fast time to market and it has enabled system engineers to cope with changing market requirements.

Application Specific Integrated Circuits (ASICs), fuse programmed Custom Gate Arrays (CGAs), Programmable Array Logic (PAL), Programmable Logic Array (PLA) and Programmable Logic Devices (PLDs) [**1**] are the other Integrated Circuit technologies available to designers for implementing digital logic. ASICs have always been preferred by the industry for design and implementation of digital systems, but over the last decade we have seen FPGA technology come a long way from a few thousand gates to more than a million gates in recent times. It is this increase in the gate density and the capability to operate at high clock frequencies that has enabled the use of FPGAs in many applications for which ASICs were preferred earlier.

With the industry requirement of making the systems as small and portable as possible, the factors on top of every design team are the size and power considerations. These portable systems need to have high functionality and are expected to have battery lives lasting weeks or even months at a time. While ASICs have been successfully used in the past to provide such a requirement, an increased need for a fast time to market has proven the use of FPGAs to be more

beneficial. Table 1-1 below [**2**] summarizes the design choices available for electronic module design. The factors used to make the comparison are the cost, time to market and the flexibility offered by the platform. The platform referred to as custom processor is defined as a product designed for a particular application while the generic microprocessor refers to a general off-the-shelf microprocessor. The speed of the technology is decreasing while its design flexibility increases as you go down the table

Technology	Performance/ cost	Time until running	Time to high performance	Time to alter functionality
ASIC	Very High	Very Long	Very Long	Impossible
Custom processor/ DSP	Medium	Long	Long	Long
FPGA	Low-Medium	Short	Short	Short
Generic microprocessor	Low-Medium	Short	Not attainable	Short

Table 1-1: Available technology choices [2]

With all of the technologies available, it is up to the designers to choose the platform most suited for the application they are designing. This choice has to be made considering all the features of a particular technology that will allow the designers to meet design specifications and market requirements. Bio-medical devices are one field that has utilized these technologies. Systems have been implemented in ASICs and microprocessors, and in this thesis we explore the viability of an FPGA based system design of a module for application in this field. In this chapter, I have provided an overview of the FPGA device architecture, applications, design flow and described the objectives of this thesis work.

1.1 FPGA architecture

The FPGA is made up of a large number of identical logic cells also known as logic elements (LE) [l]. The term logic cell (LC) is used by Xilinx while Altera refers to them as logic elements;

both describing the same elemental component of the FPGA architecture. The LEs or LCs are grouped to from a slice; there are two LEs in a slice. These logic elements or logic cells are the core building blocks of an FPGA. The number of logic cells/elements has increased from 64 in the Xilinx XC2064 to 200,000 in the Altera Cyclone III device family and this value is increasing with every new device family introduced.

These individual cells are interconnected by a matrix of wires and programmable switches. The user defined operation is implemented by specifying each block with a simple logic function and closing the programmable switches and combining these basic blocks to get the desired functional logic block. A more complex design is implemented by interconnecting these blocks to create the desired digital logic circuit. The three configurable components of an FPGA are shown in the figure below:

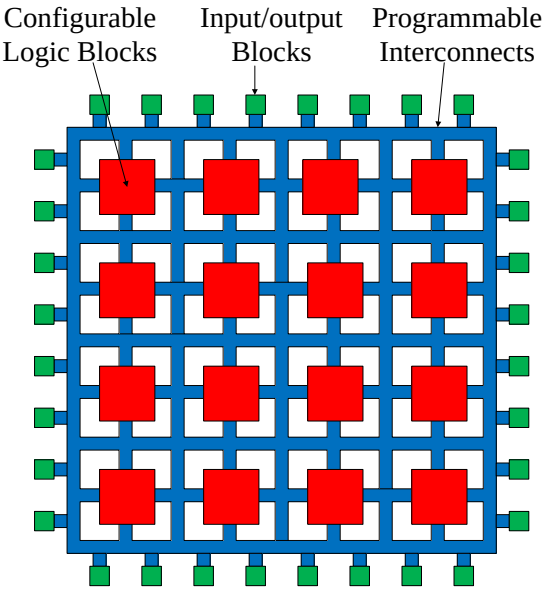

Figure 1-1: Configurable elements of a FPGA [3]

- Configurable Logic Blocks (CLBs) or Logical Array Blocks (LABs)
- Programmable Interconnects

- Input/output Blocks (I/O)

1.1.1 Configurable Logic Blocks

CLBs (as referred to by Xilinx) are the basic logic units of the FPGA [**1**]. Logic array blocks or LABs as referred to by Altera, have the same concept as the CLBs. Every CLB consists of a number of slices, each slice consisting of functional blocks called Logic Cells. These Logic Cells have in them Lookup Tables (LUTs) which are configurable switch matrices with 4 or 6 inputs, multiplexers and flip-flops.

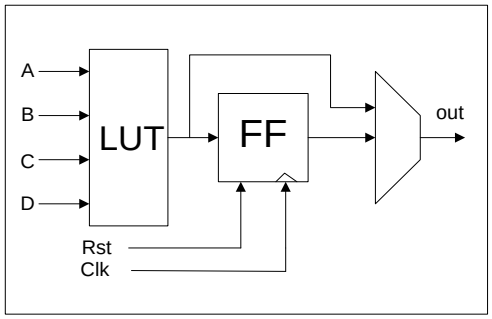

Figure 1-2: FPGA logic cell

Two of these logic cells make up a slice and a CLB consists of 1, 2 or 4 slices depending on the FPGA vendor and a particular FPGA family. The basic difference between CLBs and LABs is the definition of what components constitute a logic cell.

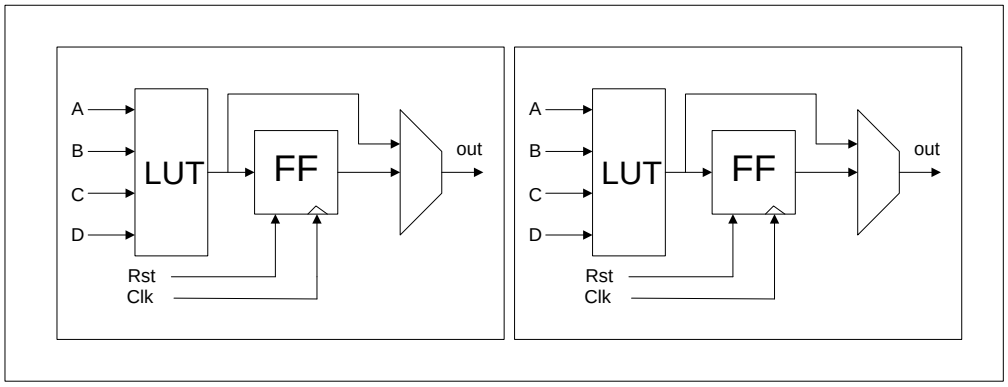

Figure 1-3: FPGA slice

As the name suggests the LUT stores the truth table of the combinational logic function to be performed [**1**]. Every combination of the input signals points to a particular cell in the look up table, which contains the desired value. In certain earlier architectures, it was suggested that the use of multiplexers (MUX) instead of LUTs was more advantageous. This approach however was not adopted because the MUX-based architecture did not provide high speed carry logic chains that help speed up arithmetic processing. The LUT can be formed using SRAM cells, EEPROM cells, anti-fuses or FLASH cells. They are SRAM based in many devices which allows for the use of a 4-bit LUT as a 16x1 RAM block or a 16 bit shift register.

1.1.2 Programmable Interconnects

While the CLBs perform the required logic functions, the programmable interconnects are responsible for routing the signals between CLBs or from the CLBs to and from the I/O blocks. The responsibility of performing this routing function is handled by the design software, which greatly reduces the complexity of the design procedure. These interconnects are either local interconnects available to individual CLBs or global interconnects that route signals between the CLBs. Some FPGAs are also provided with dedicated interconnects for improving device performance like a global clock network to route critical clock paths and carry chain logic between the LEs to assist in speeding up arithmetic operations.

1.1.3 Input/output Blocks

I/O blocks contain circuitry that facilitates transfer of signals to and from package pins to the internal signal lines. These blocks are distributed on the periphery of the FPGA architecture and are provided with registers to achieve glitch free signal switching. Devices that are provided with high speed I/O blocks and embedded transceiver modules are a great advantage in

communication applications. Current FPGA devices support many I/O standards such as 3.3V LVTTL, 3.3V LVCMOS, 1.5V etc in the Cyclone II device family [**4**].

1.1.4 Additional Features

In addition to the blocks described above FPGA architectures are provided with additional features like [**1**]:

- Embedded RAM: To provide for the memory requirements of the user applications, FPGA architectures are provided with embedded RAM block arrays. Depending on the architecture these blocks can be located on the periphery of the device, arrange into columns or scattered within the device. These blocks can be used independently or as a single block depending on the application.

- Embedded DSP blocks: Arithmetic operations like multipliers and adders are required to perform DSP applications. These operations, especially multiplication, when implemented by connecting large number of CLBs can be logic consuming and are prone to glitch errors. The embedded DSP blocks provide error free operation and also consume less power while taking up less logic area on the device.

- Embedded Microprocessors: Some FPGA architectures have microprocessor cores embedded in them, referred to as microprocessor cores. These cores can be either hard microprocessor cores where the core is a dedicated part of the IC or soft microprocessor cores where the core is implemented in the general purpose logic cells.

1.2 FPGA Applications

ASIC designers have always used FPGAs as a resource to test and verify their system design before the final ASIC implementation. Nowadays with the development of advanced FPGAs,

they are being used for just about any application. These high-performance FPGAs contain features like embedded processors and high-speed input/output (I/O) interfaces to name a few, which enable their use in the design of many Digital Signal Processing (DSP) applications and System on Chip (SOC) designs. FPGAs are finding their way into the following Digital Logic fields [**1**].

- Custom Silicon: As discussed earlier, FPGAs are being used in applications that previously could only be done in either custom silicon or ASICs.
- Digital Signal Processing: Traditionally DSP has been implemented on DSP processors which are specialized microprocessors whose architecture has been optimized to quickly and efficiently perform the operations required to perform a DSP algorithm. With current FPGA technology enabling the embedding of dedicated arithmetic blocks and availability of large amounts of RAM on chip, we are seeing FPGAs being considered for many DSP applications.
- Embedded microcontrollers: Microcontrollers have been used in applications that are required to provide simple control functions at a low cost. Generally, embedded microcontrollers are application specific with memories, I/O etc packaged with the processor. FPGAs are more than capable of providing the functionality of the microcontroller and with the costs falling, the end result is a more customizable and flexible system
- Physical layer communications: In network systems FPGAs have been used to interface communication chips with the high level networking protocols layers.
- Reconfigurable computing: This concept was proposed in the 1960s in a paper by Gerald Estrin [**1**]. It talked about a computer built with a standard processor and an array of

reconfigurable hardware. The core processor controls the functionality of the reconfigurable hardware *i.e.* FPGA which is tailor made to perform the specific task at hands such as DSP or image processing and after the task was done the same reconfigurable hardware could be used to perform any other task.

1.3 Design Flow

The figure here shows the standard design flow of an FPGA application design process [**5**]

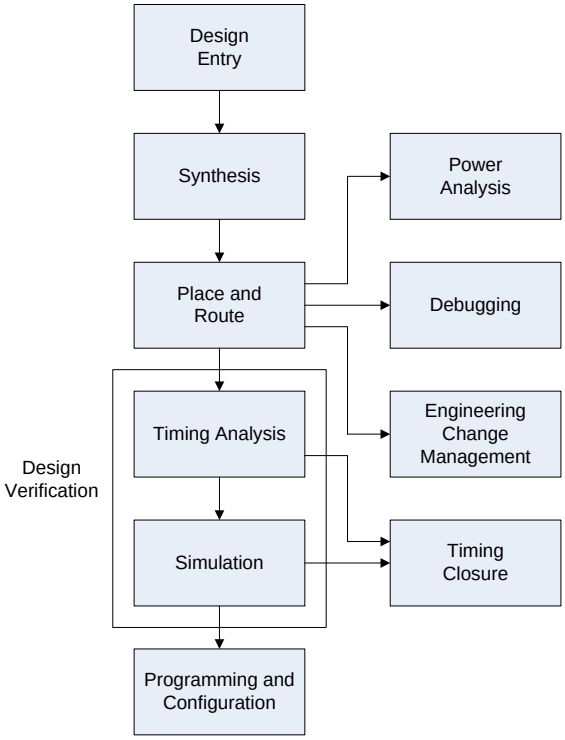

Figure 1-4: FPGA design flow [4]

1.3.1 Design entry

For the implementation of any design in an FPGA the digital design can be created with either a schematic digital design editor, Hardware Description Language (HDL), or a combination of the two. Selection of the design entry method is up to the designer and the design that needs to be implemented. For a complex design, the schematic design approach is recommended, but if the designer is comfortable with an algorithmic approach, then an HDL is

the better choice. HDL entry is faster and isolates the designer from the details of the hardware implementation, but it lags in performance and if you are hardware oriented, the schematic entry method is a better choice.

1.3.2 Synthesis

You can synthesize your design once the design files have been created. The synthesis process converts HDL code into a device netlist format. The device netlist is the gate level circuit as described by the HDL code. The synthesis process will check code syntax and analyze the hierarchy of your design which ensures that your design is optimized for the design architecture you have selected. The device netlist file generated is used in the next step.

1.3.3 Place and Route

This step of the design flow matches the logic and timing requirements of the project with the available resources of the target device. It assigns logic functions to the best logic cell location to meet routing and timing requirements, and selects appropriate interconnection paths and pin assignments. After the place and route is done, the design is tested and debugging is performed if needed. The power analysis step, after place and route, helps the designer compare block level power consumption values to the design specifications and alter the design if he wishes to. This assists in power consumption control of the final design.

1.3.4 Design Verification

Simulation and Timing analysis of the design constitute design verification. These are performed using EDA simulation tools to test functional and timing validity of the design and compare the results to the original design requirements. Verifications can be either RTL simulations or post -translation simulations. In RTL or behavioral simulations the code is tested to check if it performs as intended. This simulation is very fast and any errors or concerns can be

easily traced back to the HDL code. The post translation or functional simulations are performed to check the logical operation of the circuit generated after the synthesis step. If the circuit operation is not as intended, the changes in the design are made and the design steps are performed again.

1.3.5 Programming and Configuration

Configuration is a process in which the designed circuit (a bitstream file) is downloaded into the FPGA. Through this method the FPGA is configured from within design software.

1.4 Motivation and objectives

FPGAs have been used in the development of many portable systems and the fact that the above mentioned features can be utilized to optimize and enhance the systems functioning as per the user's requirements, have made them a very popular platform for digital logic implementation. With power consumption becoming a major concern in portable wireless devices, many researchers have studied and quantified FPGA power consumption and proposed strategies to reduce it.

This thesis work was done to reduce power consumption in such an application. The following objectives were achieved in this research

1. Proposed low power design methodologies were studied and their effects were quantified
2. A wireless signal acquisition module was designed and tested to confirm proper functioning
3. Design features were added to reduce the power consumption in the system

4. Simulation and power estimation was used to observe the efficacy of the design methods used to reduce power consumption

5. A custom test board was designed to perform real time on-board power measurements

6. Power estimation results were compared to real time FPGA current measurements to gauge the accuracy of the estimator tools and the testing approach taken.

1.5 Thesis outline

This thesis is organized as follows:

In chapter 2, I provide details of related work done in the field of low power FPGA designs. In this chapter, the research work done on the techniques for low power design implementation is categorized, and the work done in each of the categories is discussed.

In chapter 3, the various FPGA power components are described and the modes of operation of an FPGA device are discussed. This is followed by discussion on the standard design methodologies that are practiced to achieve low power FPGA design.

In chapter 4, the hardware and software choice i.e. the device choice and the CAD software choices made are discussed, outlining their advantages and other features that make them suitable for the purposes of this thesis work

In chapter 5, the design work done for this thesis is discussed. The HDL code written is described followed by the details about the process involved in designing a custom board for making power measurements.

In chapter 6, the details of the low power techniques used in the HDL coding process are discussed along with the strategies implemented for testing purposes. The results of power analysis done on the individual modules using the power analyzer tool are discussed.

In chapter 7, the device operation is demonstrated and the strategy implemented for the power analysis done and the work done to allow for making real time board measurements are described. The real time board measurements made are provided in this chapter.

In chapter 8, conclusions are drawn and future work is discussed.

Chapter 2

Related work in Low power FPGA design

With the development of high density FPGAs with embedded features as mentioned above, it is possible now to choose FPGAs for applications that in the past were primarily implemented using ASCIs and microprocessors. As mentioned earlier, high NRE costs associated with ASICs along with the difficulty in prototyping designs [**1**] has caused designers to choose FPGAs for those applications. For microprocessors, one thing that affects applications implemented on them is the fact that in a few years the processor will become obsolete [**2**]. The most radical and expensive solution to this problem of processor obsolescence is to design the system all over again around a new processor. This results in loss of hundreds of man hours which could be spent refining the application. Depending on the life of the application, this solution is temporary at best.

In this chapter, the research work done in the field of low power FPGA design is presented. As mentioned in a previous chapter, designing for low power consumption without compromising performance requires power-efficient FPGA architecture and good design practices to leverage the architectural features. Power conservation techniques are provided for every level of the FPGA design flow. The researchers in [**6**] have taken a comprehensive look at these techniques at every stage of the application design process. These techniques can be broadly classified into: device level techniques, where the manufacturing process, elemental circuitry and device architecture are the points of interest; system level techniques where the digital logic implementation and device operating conditions are targets for these techniques; and CAD tools at design software level that can assist designers in the optimization process.

2.1 Device level techniques

Power reduction techniques have been suggested for as low as the transistor level. The affect of transistor gate length reduction coupled with reduction in supply voltage is shown to have a substantial improvement in dynamic power consumption reduction of an FPGA [**7**] [**8**]. Methods are suggest in [**9**] for controlling CMOS leakage at transistor level, and reducing switching power by using low voltage swing flip flops. Models generated in [**10**] for leakage and timing variations in an FPGA are used to improve the device design and architecture. This results in substantial improvement in leakage reduction. Improvements in routing architecture also helps reduce power by reducing the number of routing elements needed. Other architectural changes like increasing LUT size from 4 inputs to either 6 or 7 inputs reduces power, as less routing is needed between LUTs improving dynamic power consumption. LUTs are implemented using a low leakage smaller transistor which improves device static power consumption [**6**].

2.2 System level techniques

The techniques suggested are applied on the system design to improve power efficiency while some alter the systems operating conditions to achieve optimization. Some of the techniques studied are provided in this section.

2.2.1 Voltage scaling

In this technique, supply voltage of the circuit is varied to reduce power consumption. This can lead to reduction in system performance due to an increase in circuit delay. The delay occurs as the gate switching speed reduces with reduction in supply voltage [**11**] [**12**]. In [**12**], the researchers have developed and tested a voltage controller that dynamically controls supply voltage and is responsible for ensuring that voltage is not reduced beyond a certain level which affects system operation and delay requirements.

There are two errors associated with this technique: I/O errors and delay errors [**12**]. In I/O errors, the supply voltage is at such a low value where a logic level high on an I/O line is lower than the threshold voltage of I/O blocks and is interpreted as logic level low. In delay errors, the voltage reduction brings about a reduction in the switching speed, which in turn reduces critical path delay. If the voltage is lowered further, this delay can become longer than the clock period, and the FPGA no longer meets timing requirements. I/O voltages are not varied to maintain the FPGAs compatibility with other components on the board level. Care should be taken in determining the optimal voltage so that it helps reduce power consumption and at the same time maintains system reliability. In [**13**] and [**14**], researchers have designed systems implementing these design techniques; measured and quantified the improvement in power consumption of the design.

2.2.2 Clock gating

FPGA interconnects are a major portion of the static and dynamic power consumption [**11**] [**8**] [**15**] as shown in Figure 2-1 [**8**].

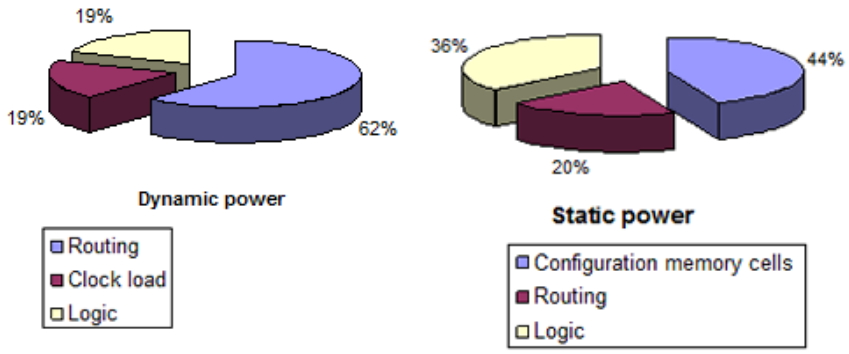

Figure 2-1: Spartan -3 core power consumption [8]

In order to reduce the effect of interconnect power dissipation, switching activity on them needs to be reduced. This reduction can be achieved by methods like clock gating and clock frequency

scaling [**16**] [**14**] [**17**].Clock gating is described in this section and clock scaling is discussed in the next.

In the clock gating technique, specified synchronous components of the system are disabled by removing the clock to it during either idle or standby mode of operation. Experimental results show that clock gating technique applied to FPGAs has a higher dynamic power reduction than ASICS [**16**]. The simplest method of clock gating is by using a single AND gate with both the clock and enable signal as inputs. This technique, however, is not without drawbacks. This implementation will invariably lead to setup and hold time violations in the design due to improper alignment of the clock edges. Another technique is to use a flip flop to sync the enable signal with the clock and reduce clock misalignment [**18**]. Both approaches are shown in Figure 2-2.

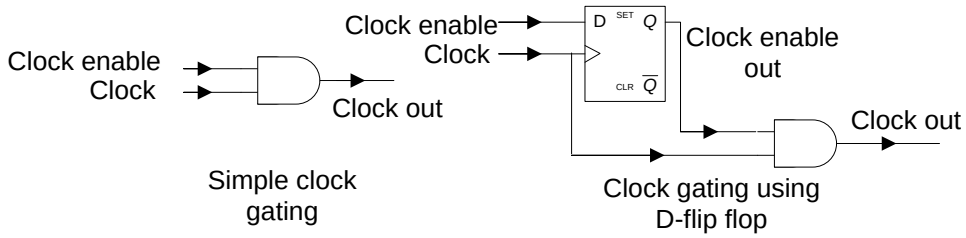

Figure 2-2: Clock gating techniques

2.2.3 Clock scaling

In clock scaling, clock rates are adjusted based on the amount of computation required [**17**]. A reduction in frequency results in reduced signal activity and lower power consumption. Dynamic clock frequency scaling can be achieved by designing clock managers, and special care needs to be taken in the design. Improper design can affect both functioning and power consumption of the device. Two such design concerns are clock skew and clock hazards or clock glitching. Clock skew is when a single clock reaches different components in a design at

different times. This may be due to interconnect length and capacitive effects during transmission. This skew can lead to alignment difficulties and loss of functionality in applications where alignment is very important, like data communication. Glitching on signal lines can cause the system to go into a metastable state and compromise circuit operation

An approach suggested to reduce skew is by introducing a PLL and using feedback from the logic to dynamically adjust the input frequency. Glitching during dynamic clock frequency adjustment is addressed by making sure that proper alignment is achieved between the current clock and the clock the device is switching to. While reduction in clock frequency does lower power consumption, it also affects the design performance. One way to account for this is by introducing parallelism in the design. While this can lead to increase in the logic area consumed, experimental results have shown that power consumption increases disproportionately to area occupied when operating at a lower frequency [**14**].

2.2.4 Glitch reduction

A large fraction of FPGA power consumption is caused by glitches [**19**] [**6**]. One mitigation technique studied [**20**] is pipelining, where registers are placed in between combinational circuit blocks to reduce occurrence of spurious signal activity that can affect the operation of the circuit. Another approach to reduce glitching, is adding configurable delay elements [**21**] to the input of each logic element to align their arrival times. The experimental results showed a significant improvement in power reduction in both approaches. Embedded DSP and RAM blocks are an added advantage in the device structure. Mapping components into these blocks can help improve power efficiency of the design in both FPGAs and ASICs [**22**].

2.3 FPGA CAD tools

FPGA CAD tools are provided with techniques to further optimize the design. Power consumption in clock networks can be reduced by utilizing clock aware placement algorithms implemented in CAD tools which also optimizes the design's speed and routability. Also, dividing the design into clock domains can lead to a reduction in complexity and area of a clock network [**6**]. As mentioned above, both Altera and Xilinx provide designers with power estimation and analysis tools. They also provide early power estimators [**5**] which are spreadsheets and can be used to estimate power consumption before the design is completed. These estimates as based on estimated resource utilization details generated by the design software and signal activity and operating condition requirements of the design.

The power analysis, however, needs more detailed information to make power estimations after the design is completed. The analysis tools take into account capacitance effects, leakage and signal activity at each node to perform the estimations and hence are more accurate than early power estimates. If the simulation results are unknown, vectorless estimations can be used to supplement information needed in the power analysis. Vectorless estimation determines signal activity at a node using the estimated inputs of that node and the logic operation of the node. It is typically fast and does not require any input signal information. However, the fact that it does not take into account interactions between nodes, makes it less accurate.

The CAD software is also responsible for placement and routing of the design on the device. Low power mapping algorithms [**6**] are implemented during this process to minimize power. These algorithms achieve this by reducing interconnect length between logic blocks by proper mapping into the device, and in general they minimize power by absorbing as many high activity nodes when gates are packed into LUTs and minimizing node duplication.

Chapter 3

Design methodologies for Low power design

In this section, we take a look at how the power consumption in an FPGA application is characterized. The FPGA operation modes are discussed and then a few proposed design methodologies and approaches for low power consumption are described.

3.1 FPGA power components

Power consumption of an FPGA is classified into four basic components. They are Power Up or In-Rush power, Configuration power, Static power, and Dynamic power. These factors need to be considered in designing the power supply for the low power application

3.1.1 In-Rush power

During power-up of the device, supply voltages need to be ramped up to their operating values and the device voltages need to be ramped up to a stable state. To achieve this, a substantial amount of logic array current is needed for a specific duration. This high current is called the in-rush current. The amount of time it takes for the supply voltages to reach their steady state values depends on how high this current is, which could be in the order of a few amperes.

3.1.2 Configuration power

In SRAM based FPGA systems where the device configuration data is stored on a non-volatile external memory like an EEPROM or Flash memory, this power is drawn while the system is configured during power up. The device draws this current to load configuration data from memory and program the logic and I/O blocks.

3.1.3 Static power

Static power depends on the amount of static current that the device draws when it is not performing any operation, after the device is powered up and configured, due to inactivity in clock and data paths. It is dissipated either as transistor leakage current or bias current. As dimensions of transistors shrink, the amount of leakage current has increased. This has resulted in static power becoming a larger fraction of total power consumed.

3.1.4 Dynamic power

This power is a factor of switching capacitances and routing capacitances and is dissipated when signals charge the capacitive nodes present within logic blocks, routing wires in the interconnect logic, I/O pins and other board level traces driven by the FPGA outputs. This power depends on the operating frequency and signal switching activities in the devices capacitive nodes. Smaller transistor dimensions result in smaller capacitances and hence lower dynamic power dissipation. The dynamic power can be modeled as follows [**17**];

$$P_d = kV_{DD}{}^2 C_L f \qquad (1)$$

Where f is clock frequency, V_{DD} is supply voltage and C is the output capacitance of a gate.

The graph in Figure 3-1 shows the inrush and configuration power consumption components described above for SRAM FPGAs [**23**]. These two components are important because designers must account for them when designing the power supplies. These components are an issue especially when we have multiple FPGAs drawing power from a single source. If the device undergoes frequent on and off cycles, these components have to be accounted for in calculating battery life.

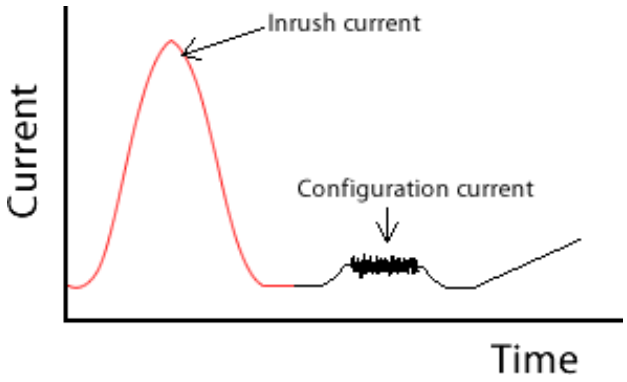

Figure 3-1: SRAM FPGA current components

3.2 FPGA power modes

After classifying the power consumption components we need to take a closer look at the device operation for accurate total power consumption estimates. FPGA device operation can be divided into five modes: power-up, configuration, stand-by, active, and sleep. The amount of time the device spends in each of these modes can affect total power consumption.

3.2.1 Power-up mode

In this mode the device is powered up and brought to a state where configuration can be done. In SRAM-based FPGA systems this includes the power supplies ramping up to a steady state and the device being reset for configuration. The in-rush current is seen in this mode and as mentioned above, it can be on the order of a few amps. To moderate this surge, devices are powered up in complex sequences which add to the cost and complexity of the system.

3.2.2 Configuration mode

Once the device has been powered up and reset, the device is configured. The configuration is done with either data stored on an external memory or with bitstream data downloaded from an external device. This sequence is performed every time after power-up is completed as well as when prototyping boards are programmed in a development environment. The amount of current

consumed is in the range of a few hundred milliamps which is acceptable for devices that get power from the electrical grid, but for portable devices this can be a power critical mode.

3.2.3 Standby mode

In this mode, the device is configured but not active. In this mode, the device is either waiting for an input from the user or an interrupt form a coprocessor working in parallel. The power consumed in this mode is static power. The amount of static power consumed depends on the FPGA technology and the operating temperature. SRAM FPGA devices consume more static power than FLASH FPGA devices and this difference is higher at higher temperatures.

3.2.4 Active mode

In this mode the device is performing operations described by the application bitstream. The power consumed in this mode is not just dynamic power due to switching activities but also includes static power. Just as in standby mode, SRAM FPGAs consume more power than FLASH FPGAs and this difference is more at higher temperatures.

3.2.5 Sleep mode

To conserve power in portable battery powered devices, power is turned off when the system is in idle state. This mode is different from standby mode. In standby mode the device is still consuming static power even when it is idle, while in sleep mode the device maintains only minimal power required to bring it back to operational mode. This has a drawback in SRAM FPGAs because the configuration data has to be reloaded every time the device is coming out of sleep mode, thus consuming the configuration power. The graph in Figure 3-2 shows the power profile. We know that the device operates majorly in the sleep mode. So, even though the power consumption is minimal in this mode, the duration spent by the design in it results in the power consumed in this mode becoming a major part of the total power consumed

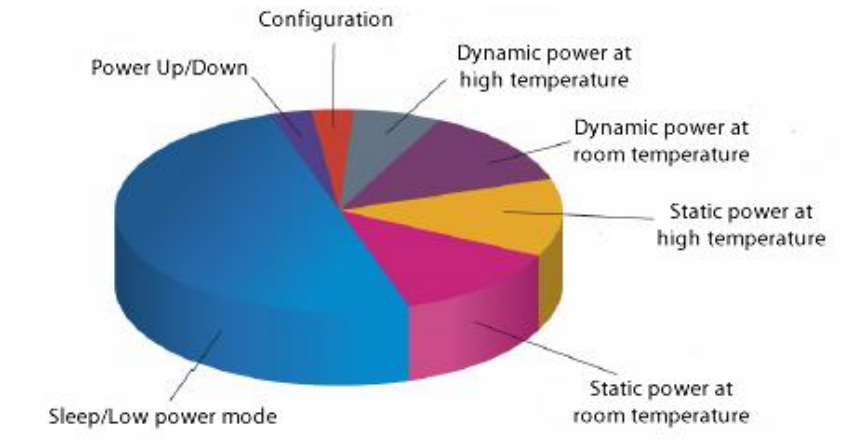

Figure 3-2 FPGA power profile [22]

3.3 Low power design approaches

Different low power techniques provide different percentage improvements, so used alone they might not provide the desired optimization; moreover power reduction achieved is based on the test design and test device. But these approaches used together substantially reduce overall design power consumption. Table 3-1 [**24**] shows the contribution of each of these approaches in a microprocessor device design's overall power reduction.

Technique	Saving
Low power synthesis	15%
Clock gating	8%
Logic/architectural changes	45%
Voltage reduction	32%

Table 3-1: Power optimization achieved [24**]**

Now that we have an idea about the power consumption components and how this total power is distributed over the different modes of the device's operation, we can make design decisions from the very first step that allow us to identify problem areas and mitigate them with appropriate technology choices and design strategies. By using low power techniques to optimize the design and operation, significant reduction in power consumption can be achieved. In [**13**]

the designers of an audio decoder could achieve 55% power saving compared to other available decoders.

3.3.1 Proper device selection

First step in the design process is choosing a right FPGA to run the application on. This is an important step in the design process. By choosing the right device, the system design can be tailored to achieve optimization for that particular device. The architecture and device structure are very important factors in device selection. Static power, as mentioned earlier, is due to transistor leakage and bias currents. By choosing devices that employ transistors with low leakage current, static power consumption can be controlled. Transistors with longer channel lengths and higher thresholds are also used in the device structure to address this issue. Researchers in [**10**] have shown that by tuning the device a 39% improvement in the leakage yield was achieved and this coupled with architectural optimization, the leakage yield improved by 73%. In [**8**] the researchers designed a 90 nm FPGA core architecture that consumes 46% less active power and 99% less standby power when compared to a Spartan-3 FPGA core.

In [**7**], experimental results have shown that reducing gate length and operating voltage has a significant effect on the gate capacitance and dynamic power consumption. The findings are summarized in Table 3-2 below. The lower gate size has achieved a 15% reduction in gate capacitance while a reduction of supply voltage from 1.2V to 1V has reduced consumption by 17%. These two techniques resulted in a 40% overall reduction of the dynamic power

	Virtex-4 FPGA 90 nm	Virtex-5 FPGA 65nm	% of change	Power Ratio
V_{CCINT}	1.2	1	-16.60%	0.69
C_{TOTAL}	1	0.85	-15%	0.85
Power	1.44	0.85	-40%	0.59

Table 3-2: Results summary [6]

From the technology point of view, there are three main types of FPGA technologies: Anti-fuse, SRAM and FLASH [**1**], In comparison, Anti-fuse technologies consumes less static power than SRAM and FLASH FPGAs because of its finer grained architecture while they also occupy less real estate. Dynamic power consumption is lower because of small switching and routing capacitances, smaller interconnect delays and hence they are faster than their counterparts. So one would think that this is the most appropriate technology for commercial use, but it has some drawbacks. Anti-fuse technology is more expensive to manufacture, which neutralizes its speed and power advantages. Moreover, anti-fuse devices are OTP (one-time programmable) devices (once the device is programmed, it cannot be programmed again). If any design changes need to be made later in the systems life cycle the entire FPGA device has to be discarded and replaced. During the testing process it is not economical to cycle test patterns through the system, which also increases the test time in a development or prototyping environment

Even though both FLASH and SRAM FPGAs are reprogrammable devices, FLASH technology is not preferred over SRAM because these devices require five additional manufacturing process steps on top of the standard CMOS technology. FLASH devices also tend to consume more static power due to their vast number of pull up resistors. So depending on the application, proper choice of device can be made. Table 3-3 summarizes the features of these technologies.

Feature	SRAM	Anti-fuse	E2PROM/ FLASH
Technology node	State of the art	One or more generations behind	One or more generations behind
Reprogrammable	Yes (in system)	No	Yes (in system or offline)
Reprogramming speed (incl. erasing)	Fast	--------	3x slower than SRAM
Volatile (must be programmed on power-up)	Yes	No	No (but can be if required)
Requires external configuration file	Yes	No	No
Good for prototyping	Yes (very good)	No	Yes (reasonable)
Instant-on	No	Yes	Yes
IP security	Acceptable (especially when using bitstream encryption)	Very good	Very good
Size of configuration cell	Large (six transistors)	Very small	Medium-small (two transistors)
Power consumption	Medium	Low	Medium

Table 3-3: FPGA technology summary [1]

3.3.2 Voltage and temperature control

As mentioned above, transistor leakage current increases with an increase in temperature and lowering the supply voltage has significant effect on leakage current. In Virtex-5 devices [**7**] it has been shown than a ±5% variation in core voltages causes an approximate ±15% change in static power consumption and ±10% change in dynamic power consumption. FPGAs are designed to meet performance requirements within ±5% variation in the supply voltages from the nominal. In the case of junction temperature for the same family of devices, a decrease in the temperature form 100°C to 85°C causes an approximate 20% reduction in static power consumption while also increasing device reliability.

3.3.3 Supply voltage scaling and Power switching

As mentioned earlier, dynamic power is proportional to the square of the supply voltage. So by scaling it down the dynamic power can be reduced. But a reduction in supply voltage affects the device performance as CMOS gates run slower at a lower voltage. The design can be modified to overcome this performance degradation. Having more logic running in parallel to boost performance at a lower supply voltage does not have a big impact on the dynamic power consumption and the advantage of the scaled down power can be observed. In [**12**] experimental results have shown that dynamic voltage gating of the device voltage performed by a voltage controller resulted in a power consumption reduction of 4% to 54% on average. Typical reduction values were 20% to 30%.

In power switching, any unused on-chip resources are independently turned off to save power. The device can be divided into blocks based on the power consumption and each block is separately switched to conserve power. The block size can be as small as an individual CLB, controlled by the user directly or by configuring the device using an appropriate bitstream.

3.3.4 Reducing clock activity and signal activity

As mentioned earlier, dynamic power consumption depends on the clock frequency and amount of signal activity in the design. Signal gating can be used to reduce the activity on design interconnects, but existence of large clock distribution networks can diminish any gain achieved with signal gating alone [**17**]. Due to high capacitances of these global interconnects, 25% of the overall dissipation is due to clock signals and this value can go as high as 50% in highly pipelined circuits. This can be controlled by turning the clock off when the circuit is idle and turning it back on when required, which is called clock gating. The designer needs to be careful because this can lead to set-up and hold time violations in the design.

Spurious activity in interconnects, known as glitching, can also cause an increase in dynamic power consumption. In combinational circuits like adders and multipliers with multiple stages, testing has shown that glitching accounts for 80% of the signal activity [**19**]. Pipelining is shown to reduce the energy per operation by 40% to 90% [**20**] by reducing glitching. Adding configurable delay elements in the device architecture can also reduce glitching by 87% [**21**] and reduce the overall FPGA power consumption by 17%. Glitching in logic block interconnects can also be reduced by either gating or pipelining.

Another type of switching activity that can be controlled is the change in states of finite state machines (FSM). By encoding the FSM states, number of toggles of flip flops during state transitions can be reduced. FSMs are usually encoded in standard binary format but by using gray code the number of toggles during state transitions is reduced. In larger designs that extensively use state machines, this technique does have a valuable impact. The table below shows the differences in number of toggles between binary and gray code.

Binary code	# of Toggles	Gray code	# of Toggles
000	3	000	1
001	1	001	1
010	2	011	1
011	1	010	1
100	3	110	1
101	1	111	1
110	2	101	1
111	1	100	1

Table 3-4: Comparing toggle rates of binary and gray code

Another form of FSM encoding is called one-hot state machine encoding. In this type of encoding, each state uses one flip-flop. For example, a four state FSM will need a 4 bit register to be declared as shown in Table 3-5.

State[0]	4'b0001
State[1]	4'b0010
State[2]	4'b0100
State[3]	4'b1000

Table 3-5: One hot FSM encoding method

The advantage of one hot encoding is that they are typically faster. The speed does not depend on the number of states. A design with many encoded states can slow down. This type of encoding is easy to design and synthesize. The logic is larger when compared to either gray coding or binary, but in larger designs the logic occupied is comparable and this encoding makes it easier to debug.

3.3.5 Clock frequency scaling

The clock frequency has a direct effect on power, as mentioned above in the model for dynamic power consumption. A significant reduction in power consumption due to a reduction of only the signal activities is seen in asynchronous circuits. To be more effective in synchronous circuits, the clock also needs to be controlled. FPGAs, when compared to ASICS, have shown to achieve a dynamic power saving of 50% to 80% [**16**]. When compared to their ASIC counterparts, the total power saving ranged only from 6% to 30% due to the effect of high static power consumption in FPGAs. Though clock gating is one way to reduce the clock signal activity, clock frequency scaling is another method to control dynamic power consumption. In devices with higher capacitances in clock distribution circuits, this technique helps reduce the power lost at output capacitances of these lines. Lowering clock frequency affects the system performance, but in [**14**] the researchers have demonstrated the tradeoff between logic implemented and clock frequency in respect to power consumption. They have shown experimentally that at low frequencies a 5 fold increase in occupied area resulted in a 1.5 times

increase in dynamic power while the same factor of increase in area at a higher frequency resulted in a 3.8 times increase in dynamic power.

In cases where the design sub-systems are in idle state or the computational needs have reduced, the clock to these modules can be reduces without affecting the application throughput. This method is called dynamic clock scaling and has been successfully implemented in [**17**]. This method in conjunction with dynamic voltage scaling can be very effective in reducing the dynamic and overall power consumption. The device at lower clock speeds maintains switching speeds due to the lowered supply voltage levels [**17**].

3.3.6 CAD tools

Both Altera and Xilinx provide users with power estimation tools which can be used either during the design process or after the final design is done to get an idea about the device power consumption [**5**]. During the design process, early power estimators, provided as a spreadsheet, can be used to estimate power consumption of that part of the design. After the design process is done, power analyzer tools utilize detailed switching activities provided by the user. The device routing and placement information along with detailed device temperature and cooling information is also used to estimate the power consumed. Although the most accurate way to measure power consumption of the device is by hardware measurements, power estimators are a convenient way to determine the power budget. This along with power efficient routing algorithms implemented during the place and route step of the design process, help enhance the devices architectural features.

Chapter 4

Hardware and Software

This section describes the need for proper device selection for a design and the steps taken in making the choice for this thesis are discussed. The device features are presented along with some details about the Quartus II design software and PowerPlay power optimization tool.

4.1 Device Selection

A wide variety of FPGA devices are available in the market today, each with one or more of the features that were discussed in Chapter 1. Using a desired design entry method with appropriate power conservation techniques is only half of the job. It is equally important to choose the right device to run the application on, and it should provide the designer with the required resources while being flexible enough to accommodate any future design and resource requirement changes.

For this thesis design, Altera's FPGA devices were chosen for implementation due to the availability and the author's familiarity with the design software. The Altera FPGA families chosen to perform the analysis on are given below:

- Stratix series: High bandwidth, high density FPGAs
- Arria series: Midrange FPGAs with transceivers, optimized for mainstream protocols
- Cyclone series: Built from ground up for low cost, low power consumption

Preliminary power analysis measurements were made using the PowerPlay power analyzer. Steps taken in performing the device power estimations and their results are provided and discussed in a later chapter. The FPGA EP2C35F672 which belongs to the Cyclone II device family of Altera was chosen based on these estimation results.

4.2 Cyclone II FPGA Architecture

For this thesis, HDL code implementation was done on the Cyclone II device family. The device architecture is described below [**4**].

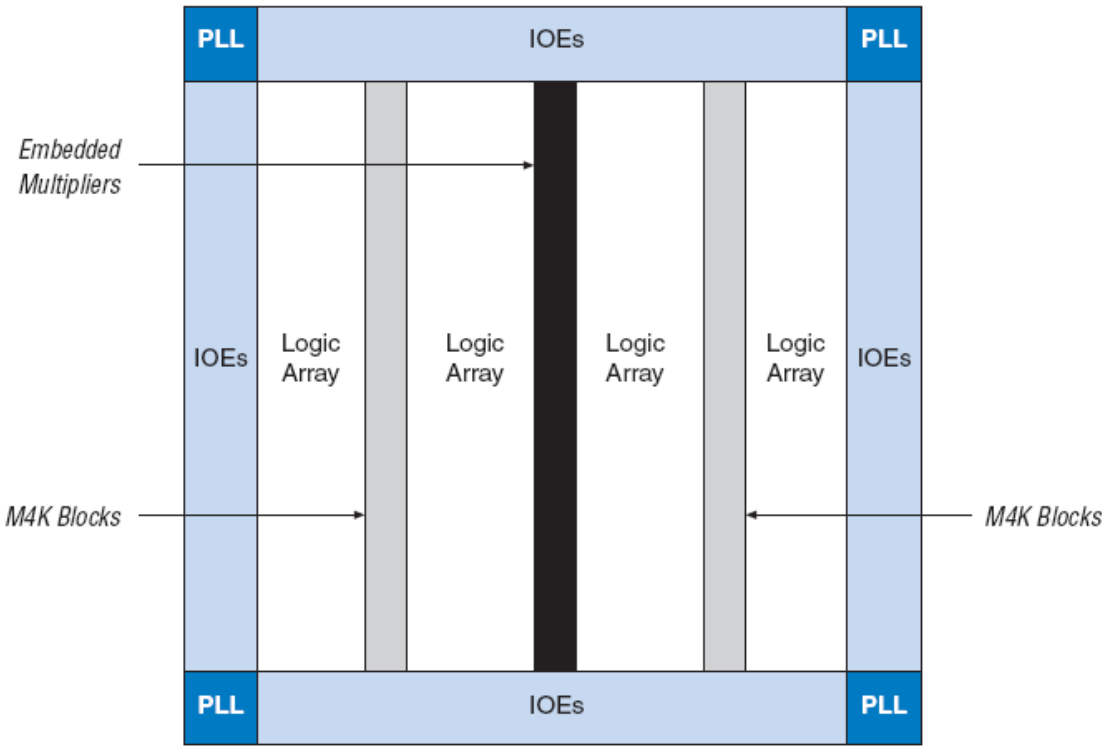

Figure 4-1: Cyclone II FPGA architecture [3]

In Figure 4-1, the area denoted as Logic Array is the array of CLBs (Altera LABs), where each LAB consists of 16 logic elements. The Interconnect structures are not shown here, but the special function blocks are shown. The Cyclone II FPGA has embedded in it multiplier blocks, RAM blocks (shown as M4K blocks), and Phase Locked Loops denoted as PLL in the figure. As labeled, the IOEs are the input/output elements or input/output blocks.

4.2.1 Interconnect structures

The architecture of cyclone II is a two-dimensional row and column based architecture. The row and column interconnects provide the required connectivity between different CLBs, multipliers and RAM blocks. These interconnects are of varying speeds. Along with 4 PLLs, the

cyclone II device is also provided with a global clock network. The clock network consists of up to 16 global clocks running through the entire device, and it is used to provide clock to all device resources like multipliers, memory blocks along with the CLBs and IOEs.

The device architecture has special performance enhanced routing lines called multi-track interconnects. They consist of routing lines of different speeds, used to achieve connectivity within a design block or between design blocks. Depending on how critical a certain interconnect path is, the compiler decides if it needs to be on the faster lines or not, in order to boost the design performance.

4.2.2 Embedded multipliers

Each of the embedded multipliers can be used to implement either two independent 9x9 bit multipliers or one 18x18 bit multiplier. These multipliers have been optimized for multiplication intensive DSP functions like Finite Impulse Response (FIR) filters, Fast Fourier Transform (FFT) etc. The number of embedded multiplier columns depends on the device; between 1 and 3; and these columns are the same length as the LAB column. For example, in the EP2C35 device, we have 1 embedded multiplier column with 35 multipliers which can be used to implement either 70 9x9 multipliers or 35 18x18 multipliers. These multipliers operate at a maximum frequency of 250 MHz at the highest speed grade.

4.2.3 Embedded memory blocks

Each of the embedded memory blocks has a size of 4068 RAM bits. These blocks can be used to implement different types of memory such as true Dual port memory (dual port RAM has ability to simultaneously read and write different memory cells at different addresses), simple dual port, single port RAM, ROM and first-in-first-out (FIFO) buffers with or without parity. The M4K blocks also support the following features:

- Byte enable: The byte enable allows input data to be masked so the device can write to specific bytes.
- Parity bits: One parity bit for each byte can be used to implement parity checking for error detection
- Shift register: Using RAM based first in first out (FIFO) buffers instead of LE registers saves general logic resources, increases overall speed and reduces power consumption.
- Various clock modes: The input and output registers can be clocked in different modes either independent of each other, one clock controlling the two ports or one clock and clock enable to provide register control.
- Address clock enable: This allows for the address busses to hold the previous address value for as long as the enable signal is given.
- Global clock networks and Phase locked loops

The global clock network has a clock control block that can be used to select between the PLL clock outputs, dedicated clock inputs and dual purpose clock inputs. Every device has a different number of the above resources available in them to select from. If the dedicated clock pins are not used as clock inputs, they can be used as general purpose I/O pins. As mentioned before, the global clock network provides clocks for all resources within the device, like the CLBs, IOEs, memory blocks, and embedded multipliers.

The global clock lines can also be used for control signals, such as clock enables and synchronous or asynchronous clears fed from the external pin. Internal logic can also drive the global clock network for internally generated global clocks and asynchronous clears, clock enables, or other control signals. Cyclone II devices have up to 4 PLLs that can be used for clock

multiplication and division, phase shifting. Some of the other features supported by the PLLS are programmable phase shifting, clock outputs for differential I/O support etc.

4.2.4 Cyclone II Logic Element structure

Figure shows the structure of the logic element of the FPGA device [**4**].

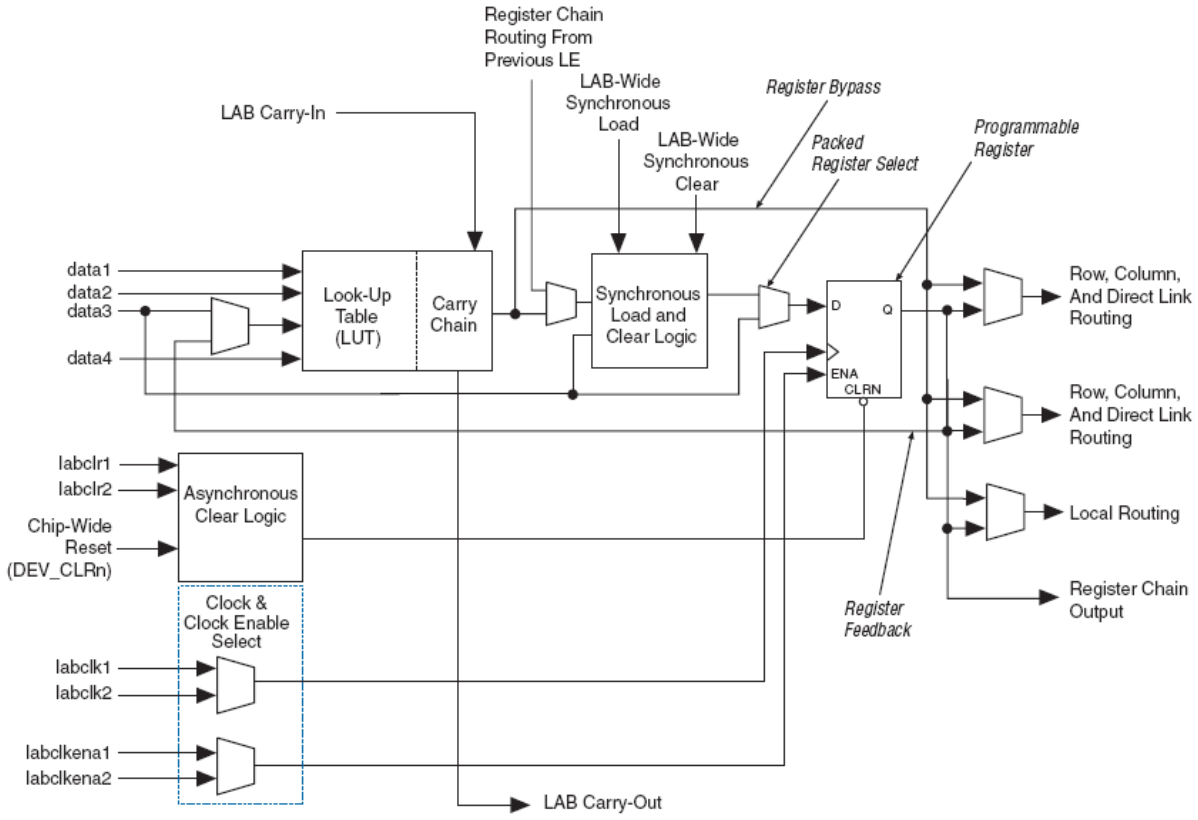

Figure 4-2: Cyclone II FPGA Logic Element [3]

Each LE in the cyclone II device consists of:

- A four-input look-up table (LUT), which is a function generator that can implement any function of four variables
- A programmable register which can be configured for D, T, JK or SR operation
- Carry chain connections for implementing fast carry logic. The Quartus II compiler creates carry chain logic during the design processing.

- A register chain connection. This feature allows the registers in a CLB to be cascaded together for shift register implementation.
- The ability to drive all types of interconnects: local, row, column, register chain, and direct link interconnect. Each LE has 3 of outputs that allow for driving these routing resources. The LUT and the register outputs can drive these interconnects independently
- Support for register packing. The high routability feature of the FPGA architecture and ability of the synthesis tool and place and route tool to utilize this feature allows the efficient use of the LEs. This allows the LUT in a CLB to be utilized independently of the registers i.e. the LUT can be used to perform one operation while the register available in that CLB can be used, either individually or in conjunction with other registers, to perform a completely different task. This reduces the size of a design and usually fits a design into a smaller device
- Support for register feedback. This is when the output of the programmable register is fed back as an input to the LUT.

4.3 Quartus II and Power Play power analyzer

Quartus II design software is one of the design and synthesis tools available to designers [**5**] [**25**]. These tools are designed to take advantage of device architecture and special features to map the design into logic and help achieve the designer's optimization requirements. The designer can choose to optimize for area, timing or power of the design. Area driven synthesis will reduce the amount of logic used. This also helps reduce dynamic power consumption due to optimized and reduced logic levels. Timing driven synthesis uses timing constraint information provided by the user to determine the design's routing. This is done at the cost of consuming extra logic on the FPGA device. In power driven synthesis the main objective is to reduce the

power consumption. Reduction in area and power driven synthesis go hand in hand. For example, using embedded features of the FPGA like multipliers, embedded RAM and special interconnect structures are very effective in reducing power consumption [**22**].

The software allows the designer to select the factor that needs to be optimized at both the analysis and synthesis step and fitter step. In the analysis and synthesis step, the design is compiled to check for syntax, logical completeness and inconsistencies. The design is synthesized and resource utilization decisions such as using embedded multipliers or memory blocks are taken. The last step of the compilation process is generating the files necessary to program the device and this process is called "fitting". Quartus II fitter places and routes the design. Using the database file created during analysis and synthesis step of the design compilation, the fitter assigns resources on the selected device to match the logic and project requirements of area, timing or power. The logic functions are assigned to optimal logic cell locations and appropriate interconnection paths are chosen. The final result of the fitting process is a bitstream file that is used to program the FPGA.

Signal activity files (.saf) or value change dump files (.vcd) help the fitter fully optimize design routing for power conservation. These files contain the signal activities of all signals in the design and are determined after a full design simulation performed on the post-fit netlist. If the input vectors used to perform simulation represent the typical behavior expected at the system inputs, the .saf file or .vcd file will reflect the actual behavior at every design node. If the user does not provide either of these files, Quartus II software uses vectorless estimation [**6**] [**5**] [**25**]to estimate the signal activities. This involves estimating the switching activity at each node based on the activity of its inputs and the logical function performed at that node [**6**]. This

information is used to optimize the post-fit design for power consumption. The figure below is the recommended design flow for power driven compilation

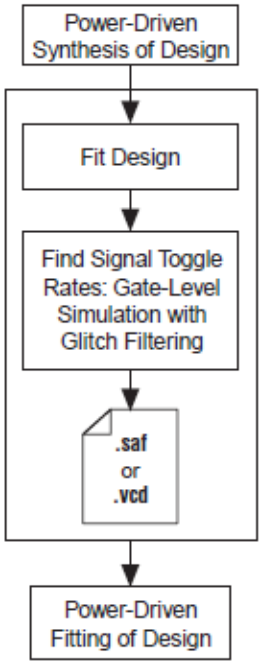

Figure 4-3: Quartus II power driven synthesis flow [24]

4.3.1 Power estimation and analysis

Altera allows for estimating power consumption at different stages of the design process. The early power estimator is a Microsoft excel spreadsheet that allows for estimating power consumption and heat dissipation on a design that is not ready for full compilation. The environmental conditions and estimated logic consumption are used to make these early estimations. For a accurate estimation of the completed design's power consumption, the Power Play power analyzer tool is utilized. This tool uses information given by the user to make power consumption and heat dissipation estimates. The analyzer takes into account the following factors:

- Device selected: FPGA families differ in power consumption mainly due to the difference in architecture. A larger device tends to consume more static power because of its higher transistor count.
- Environmental Conditions:
    1. Air flow: It can be specified as either “still air” where no fan is used to remove heated air away from the device or a foot per minute rating of a fan used is specified.
    2. Heat sink: The heat sink’s cooling capacity is entered as the case to ambient thermal resistance. The thermal compound that interfaces the heat sink to the device also affects heat dissipation.
    3. Board thermal model: Junction to board thermal resistance determines the thermal resistance of the paths through the board.
- Device resources used:
    1. The type of output pin determines the output capacitances. Output pins that drive off-chip component have high load capacitances leading to higher dynamic power consumption. I/O pins have pull up resistors which draw static power
    2. If the number of logic elements and embedded features in the device are more, it tends to draw more power than a device with a smaller number of such elements.
    3. Global signal networks consume more dynamic power due to high capacitance associated with the routing interconnect.
- The final factor that the analyzer considers is signal activity at each node of the design. The two statistics that are used are the toggle rate and static probability of signals in the individual nodes. Toggle rate is the average number of signal changes per unit of time

and static probability is the fraction of time the signal is at logic level high. Static power consumption majorly depends on routing and logic, and it can be affected by high static probabilities due to state dependent leakage.

In summary, device and device resource data reports are generated by the Quartus II software during the compilation process. The board's thermal model and ambient cooling conditions are specified by the user. The signal activity is generated by performing a full design simulation on the design post-fit netlist generated by Quartus II. The simulator generates a file indicating the signal activity in the design. The PowerPlay power analyzer tool uses all of this information to perform the power analysis. In the next section I describe the sections of the analyzer compilation report.

4.3.2 Compilation report

The Power Play power analyzer tools compilation report is divided into the following sections

- Summary: This section of the report has details of the results obtained from the analysis. It has the total thermal power dissipation, core dynamic and static power dissipation and I/O thermal power dissipation values. It also provides the user with a confidence metric. If the metric is low, it means that the toggle rate data that was provided was insufficient and either the vectorless estimated values or default values for toggle rates were used for the nodes missing this information.

- Settings: This sections has the details of the settings for the operating conditions, toggle rates etc.

- Simulation files read: This section shows the various simulation files used for the simulation purposes. If using a .vcd file, you can, in this section, specify a block of time

from the simulation results you want to use to perform the analysis for, by specifying the VCD start and end time.

- Operating conditions used: This shows the settings used for the ambient conditions and also the estimated values for conditions not set by the user.
- Thermal power dissipation by block type: The static and dynamic power dissipated by the different blocks in the design is shown here along with the average toggle rate by block type.
- Thermal power dissipation by hierarchy: The static and dynamic power dissipated by design hierarchy is shown here along with the average toggle rate of the blocks routed in that hierarchy.
- Core dynamic thermal power dissipation by clock domain: The power dissipated for each clock domain is shown here. For domains with no clock specified, a value of zero is assigned
- Current drawn from voltage supplies: The current drawn from each of the voltage supplies is shown here. It is further categorized by the I/O bank and voltage.
- Confidence metric: If the analyzer considers the toggle rate data to be from a source which is poor predictor of real time toggle rate data, for example using vectorless estimation for the toggle rates or using default values, the confidence metric is low.

Chapter 5

Design Work

The design work done for this thesis is described in this section. The desired functionality is described followed by a discussion of the code written. Later in the chapter, details of the custom board designed for testing purposes is discussed and the issues faced during the design process and the approach taken to resolve them is also described. The techniques used to reduce power consumption in the design are presented in a later chapter along with the power estimation results and real time measurements.

5.1 HDL design code

The primary objective of this thesis design is to study the power consumption of an FPGA device for an application in which input signal from different sources is taken, signal processing is performed, and the resulting information is stored in an external memory. For this thesis, the input sources selected were a single channel of digital data and a single channel of analog data. In the testing process, the digital data would be provided by the means of a RS232 connection to a computer. For the analog data an arbitrary waveform generator would be used. An SRAM memory was chosen as the external memory. It was not critical for the memory to be non-volatile, so SRAM memory was chosen over FLASH memory. SDRAM was not selected because some additional power would be needed to constantly refresh the data in the memory.

A low pass FIR filter was implemented to serve as the signal processing requirement in the analog data path. The block diagram of the preliminary design is shown below. The design is subdivided into the serial module, an analog module consisting of the analog-to-digital and digital-to-analog converters, a signal processing module consisting of the FIR filter, a memory

module consisting of a memory controller writing to an external memory, and the user interface for the user to give input commands to the application and receive output.

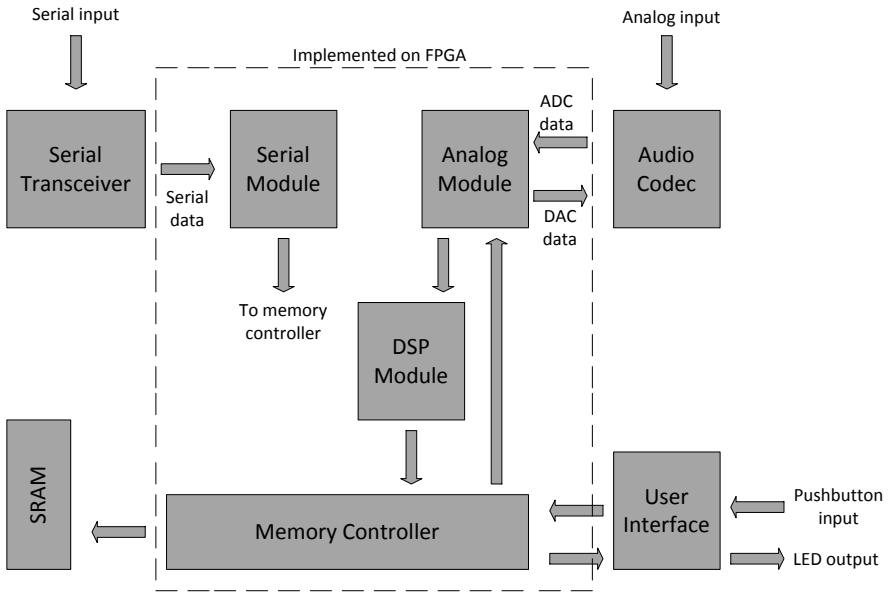

Figure 5-1: Design block diagram

The serial, DSP and memory modules are implemented in verilog HDL. The serial module gets its input from a serial transceiver like MAX-232, while the memory module sets up the data, address and control signals for the external SRAM memory. An audio codec is the central part of the design of the analog module. In the analog module, HDL code is written for components needed to program the audio codec for desired operation and data formatting modules like serial-to-parallel converters and parallel-to-serial converters.

5.1.1 Serial Module

The serial module is designed to read data sent from a computer via a serial RS232 connection. It reads the data sent serially and sends it to the memory controller to be stored in the external memory. The RS232 frame structure shown in Figure 5-2.

Figure 5-2: RS232 frame structure

The serial input line is always high, except when there is a frame put on the line in which case it first drives the line low to indicate that data is available. This is called the start bit, after which the data bits are sent. To indicate the end of transfer, the serial line is forced high. For the purposes of this design, a 16 bit word was chosen as the width of the data written to and read from the memory. Shown in Figure 5-3 is the finite state machine (FSM) implemented to read the serial data in. The three states are idle, receive and hold. In the idle state the state machine waits for the occurrence of the start bit. When the start bit is seen the state machine goes in to the receive state where the 8 data bits are stored into a register and after all the 8 bits are received, the FSM goes back to the idle state. After two frames are received, the FSM sends the 16 bit data to the memory controller to be written to the memory.

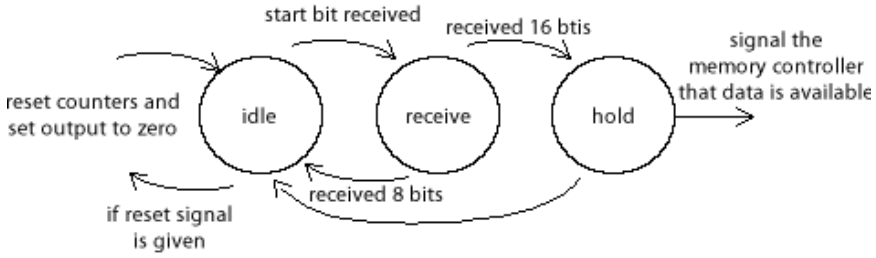

Figure 5-3: Serial module FSM flow

When the 16 bits are received in the receive state, the FSM asserts a ready signal to the memory controller indicating that the 16 bit data is available to be written and it goes into a temporary hold state before going to the idle state. The HDL code for the serial module is provided in the appendices.

5.1.2 Analog module

The central part of the analog module is the Wolfson WM8731 audio codec [**26**]. This codec has many features that make it a good choice for this design implementation. It is a low power IC with variable sampling frequencies from 8 kHz to 96 kHz which allow for flexibility in the design implementation. To program the audio codec for the desired functionality, appropriate control words need to be written to control registers. The control words can be written using either the 2-wire or 3-wire serial interface mode and the codec can be configured to format the audio data in either I^2S, right justified, left justified or DSP modes with the multi-bit sigma delta ADCs and DACs of the device capable of supporting 16/20/24/32 bit word lengths.

The audio codec allows for three analog inputs and a single analog output. The inputs are configured as two line inputs for stereo and a mono microphone input. The IC can be put into either a standby mode or power down mode to save power by means of software control. The device has a register called the power down control register that needs to be written to in order to power down either the entire device or individual sections. By changing the control words written to the control registers, the volume levels of the inputs can be changed and also muted, sections of the device be powered off and built in filters can be enabled to provide noise reduction and reduction in the dc component from the audio signal. The codec can be operated in either slave mode or master mode. In slave mode, the clocks required for the codec operations are provided by external inputs while in master mode, the codec's built in crystal oscillator is invoked for all of the devices clock requirements. The datasheet of the codec describes the following clocks as the ones that are generated or given as inputs depending on the mode:

- ADCLRC clock that controls whether right channel or left channel data is present on the output line of the ADC, the frequency of the clock in this design is 48.8 kHz

- DACLRC clock that controls whether the right channel or left channel data is present on the input line of the DAC, the frequency of the clock in this design is 48.8 kHz
- BCLK is the clock on whose negative edge the data bits are put on the output line of the ADC or need to be put on the input line of the DAC, the frequency of this clock in the design is 3.125 MHz
- XCK is the clock that needs to be given as input to the codec and its default value is 12.5 MHz

The output of the ADC and the input to the DAC is serial data of length as configured by the user is in 2's complement form.

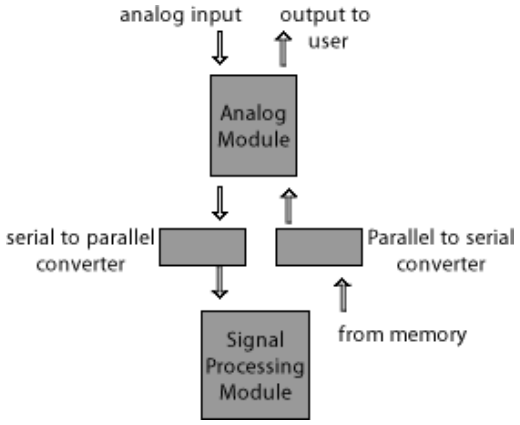

Figure 5-4: Data formatting modules

For this design, intermediate signal formatting modules were implemented to convert the data output of the ADC from serial data to parallel format, and vice versa for the parallel data output read from the memory into serial data input to the codec DAC. The serial to parallel uses the ADCLRC clock and the BCLK clock for the conversion process, while the parallel to serial converter uses the DACLRC clock and BCLK clock to properly clock the data into the DAC input line of the codec. The HDL code for the data formatting modules is provided in the appendices. The next section describes the signal processing module to which the input and

output data is 16 bits long. In this application, the user is provided with the data stored in the external memory via the DAC on the output line of the codec.

The codec configuration is provided below.

- The mono microphone input is selected to the ADC and the device DAC is enabled
- The right and left channels of the stereo input are muted.
- The codec is programmed to operate in master mode with the audio data set to 16 bit length available in the I^2S mode
- For the sampling control of the device, normal mode of operation is chosen over the USB mode, which gives flexibility in setting the frequency of the master clock
- The sampling frequency is for the ADC and DAC is set to 48 kHz

The analog module consists of two entities, one is responsible to write the command words to the codec and the other entity sets up the command words needed based on internal signals. The entity writing to the codec is called codec_prgm using two-wire serial interface, and the entity setting up the data is data_setup. When the device is done powering up, data_setup generated the first control word to be written by appending the address of the control register with the control word appropriate for the desired operation. This data word is sent to codec_prgm that programs the codec. After each control word is written to the appropriate register, and codec_prgm indicates to data_setup that the operation is done. A new data word is sent to codec_prgm to program the codec. Once the control words are written, they are not to be refreshed again unless the analog module gets the input indicating that the application wants to turn off the codec. At that time the codec is written to again with the appropriate control words. The HDL code for the analog module is provided in the appendices.

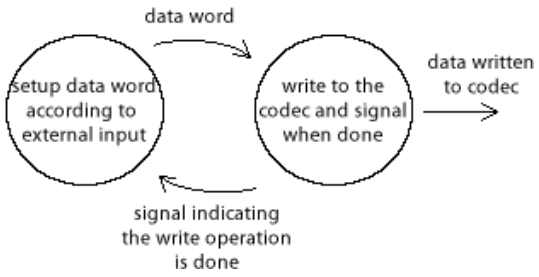

Figure 5-5: Analog module operation flow

5.1.3 Filter Module

An N-tap low pass FIR filter was designed for the application. The filter block diagram is shown in Figure 5-6 below. Value N denotes the number of filter coefficients or the length of the filter with N+1 taps. A low pass filter with a filter of length 32 was designed for the purpose of this thesis design. The filter coefficients denoted by h_0 to h_N are stored in an internal memory block and read into the entity performing the filtering process. The input data and the filter coefficients are passed through a multiply accumulate operation and the final result is the filtered output. As seen in the figure, each sample is multiplied by a filter coefficient and after a delay is sent to the next stage, all the products obtained from each of the stages is added up and the result is the sample of the filter output. This process is called multiply – accumulate or MAC.

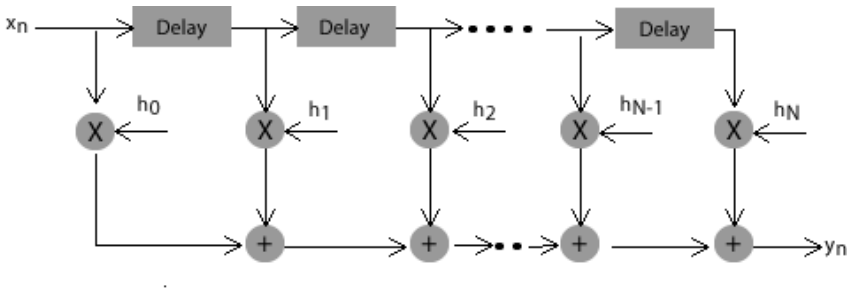

Figure 5-6: FIR filter block diagram

The filter module block diagram is shown below

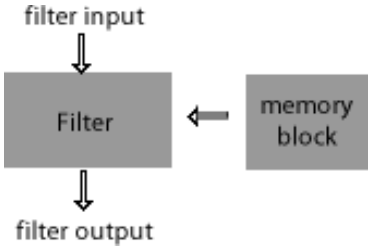

Figure 5-7: Filter module block diagram

The input data is available to the filter at edge of the ADCLRC clock. If the data is stereo i.e. information is available on both the channels, the filter would need to run at twice the ADCLRC clock speed. Since we are using the mono microphone input the data is available only on one of the channels while the other channel is mute, hence the filter is run at 48.8 kHz which is the rate of the ADCLRC clock. The HDL code for the filter module is provided in the appendices. The multiply accumulate (MAC) process is a widely used operation in digital signal processing. While there are a number of microprocessors and microcontrollers that use special dedicated units that are designed to perform this task fast and efficiently they still have a draw back when compared to FPGAs.

In a processor environment this task is performed in a loop, where it takes at least one clock cycle to perform each of the MAC operations. For example [**2**] a 256 tap filter means that 256 MAC operations need to be performed on each sample and this takes 256 clock cycles. In an FPGA, due to the fundamentally parallel processing structure all of the 256 MAC operations are performed in one clock cycle making the FPGA platform a favorable choice for such applications.

The filter coefficients were generated in MATLAB using the Parks-McClellan method for designing an FIR filter. The filter pass band is from 0 to 5 kHz, where 5 kHz to 7 kHz is the

transition band and the stop band is all frequencies higher than 7 kHz. The memory instantiated in the design is initialized with the coefficients generated in MATLAB for the filter. The coefficients need to be in a format that the synthesis software can use to load the values in. The file is called a memory initialization file and is of the extension .mif. A MATLAB function is used to convert the generated floating point filter coefficients into 2's complement form. The code for the filter and the code used for the format conversion are provided in the appendices. The MATLAB plot of the filter designed is shown in Figure 5-8.

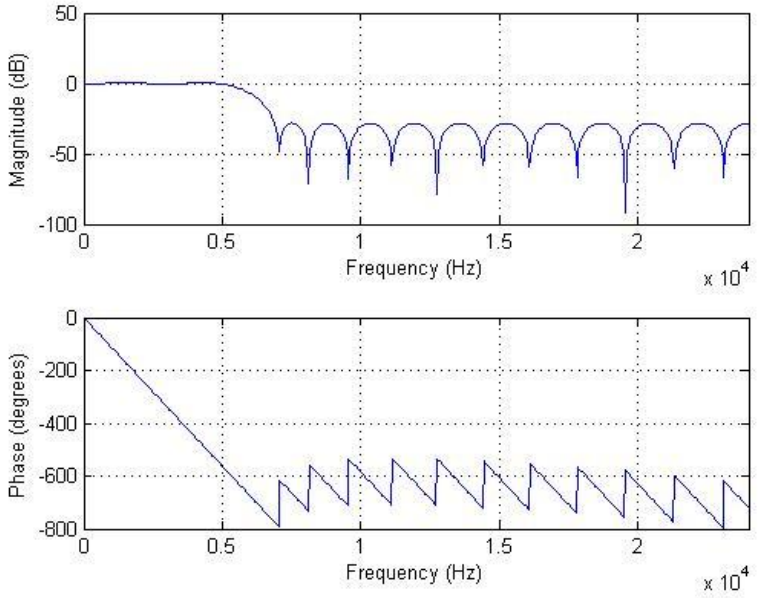

Figure 5-8: FIR filter magnitude and phase plots

5.1.4 Memory module

The memory module is used to facilitate the writing of data to the external SRAM memory. The memory module is comprised of a resolver module that is responsible to provide the memory controller module with appropriate data and control signals while the memory controller is responsible for generating the right output signals that drive the external memory. The

resolver's duties are generating the read or write addresses, and it signals the memory controller as to whether a read or write action needs to be performed.

In this design we have 16 bit data coming from the serial module as well as the analog module to be stored in the memory. In order to differentiate the two data sets, the memory was split into two banks, one for each set. So depending on the source of the input data, the resolver generates the appropriate bank address and signals the memory controller. To assist the resolver in deciding the data source, a bank select multiplexing module is used. The received serial and analog data are given as inputs to this module which then multiplexes the data onto a single output line and indicates the data source to the resolver. If data from both the sources are available, higher priority is given to the analog data because that data is generated at a faster clock. The block diagram of the multiplexer is shown in Figure 5-9.

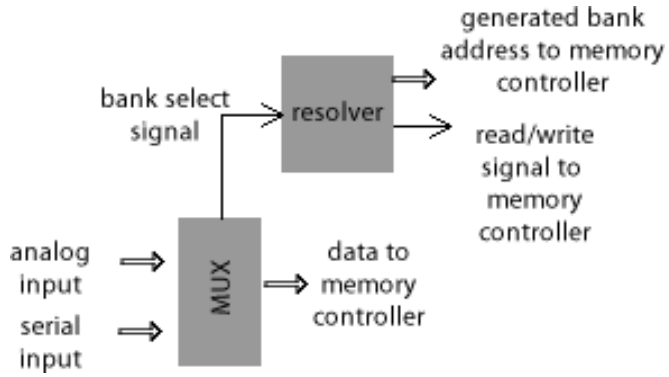

Figure 5-9: Bank select module

The resolver in its idle state waits for the input signal indicating a read or write operation. If a write operation is present, it also waits to see what bank it needs to write to based on the input from the bank select multiplexer module. When it knows what memory bank is selected, the address is generated along with a signal indicating the memory controller of the operation to be performed and the corresponding address. If read input is seen, the FSM goes into a loop flushing out all of the external memory contents. The resolver in this loop increments the read

addresses beginning from address 0 after each successful read operation, until all the memory locations have been read. After all the data from the memory is read out, the resolver goes back into the idle state and normal operation is resumed. The state flow diagram is shown in Figure 5-10.

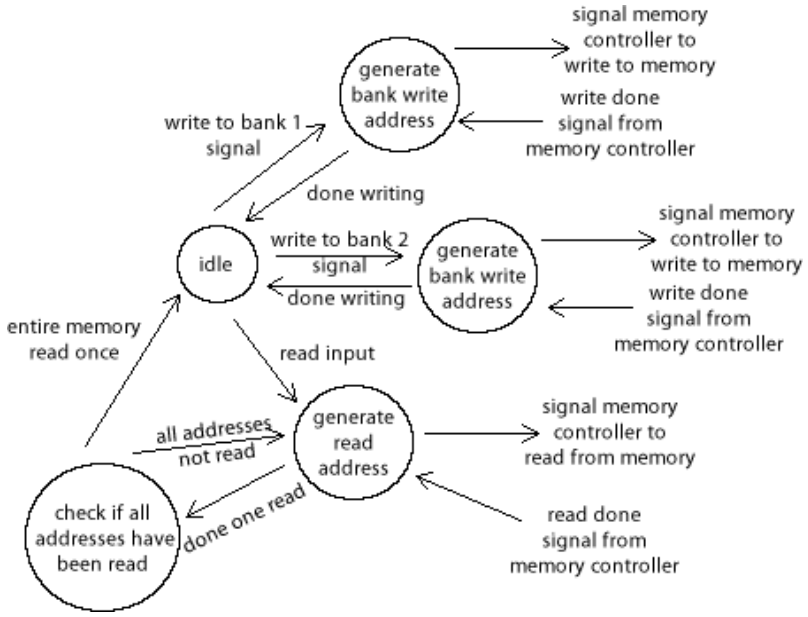

Figure 5-10: Memory module state flow diagram

The memory controller takes as inputs the data and address for a write operation or just the address for a read operation. It sets up the required interface signals for the external memory, meeting the timing requirements of the read and write operations. This module indicates to components higher in the hierarchy when a read or write operation are done. The HDL code for the resolver and memory controller module are provided in the appendices.

5.1.5 User interface

The interface in this design includes the user input indicating to the application that it needs to perform a burst transfer of all of the data stored in the external memory. In this design application, this input is given via a pushbutton. The user is also provided with the option of resetting the device asynchronously using an external reset input also given via a pushbutton.

The design approaches taken to optimize the application's power consumption will be discussed in the next chapter. As discussed in the previous chapters, clock and signal gating are very effective techniques. To indicate the user of the device's mode of operation, a LED is lit up or turned off indicating the mode to be a power save mode or normal operating mode respectively.

5.2 Test board design

Reduction of power consumed being the critical point in this thesis, it is important to have a means of taking detailed measurements to describe the total and individual block level power consumption profile of the application and quantify the affects of the different low power design approaches taken. For this purpose a test PCB was designed that would house the desired FPGA device and all of the IC components needed to execute the designed application. While the process is running, there are various interest points in the design which need to be probed for making the power measurements. For example, isolating the power consumption of the device by each voltage source can help in choosing better and power efficient components in the design of the supply circuitry.

While the FPGA power consumption is optimized in the design process, power consumption on the system level also needs to be addressed. By using techniques to properly regulate the power given to the other system components, we can supplement the FPGA power conservation and achieve a highly desired system level power consumption reduction. To allow for the power testing on the PCB, test points are provided that make the process of taking measurements using test equipment more convenient. Current sense resistors are used to measure the power drawn from the various supply voltages and these voltages have been isolated from each other on the PCB to allow for testing the affect of the voltage being scaled to the different system components on the total power consumption.

5.2.1 Design schematic

Every design process begins with first building the schematic of the circuit. For this thesis work Cadence Allegro's schematic capture tool, Design Entry CIS was used to do the schematic design. The CIS stands for component information system that allows the user to reuse known components and parts data in their design. This tool has a design rules check (DRC) feature that allows the user to check the design for any of the defined physical or electrical design rules and avoid any potential costly engineering errors.

Before beginning the design, the FPGA device selection is an important decision. The power analysis comparison of the different FPGA devices as discussed earlier showed that Altera's cyclone II FPGA device family was an appropriate choice. The results of the power analysis simulations are provided in a later chapter. The Altera DE2 board development kit provided by Terasic was used as the base design for the system schematic. The DE2 board has a number of components on it that were not needed for this design, components like LCD display, seven segment displays, Ethernet controller, DRAM memory, FLASH memory, SD card reader, video decoder and video DAC. It also provided the ability to connect to a USB device and a PS/2 serial interface to connect a PS/2 keyboard or mouse.

The components needed were the push buttons, toggle switches, LEDs, on board oscillators and an external clock input via a SMA connector, RS232 serial interface, audio codec, SRAM memory and USB blaster to program the FPGA device. The DE2 board has two 40 pin expansion headers, to provide for debugging one of the 40 pin headers was provided in this board design.

The schematics provided by the manufacturer were followed and minor modifications and additions were done to build the schematic for the test board with the required components. The

power supply design for the DE2 board was modified to generate only the required voltages needed for the system. Along with the regulated power supply generated from the power grid input to the device, an alternate way of powering the device was provided. The user had the choice to either power the device either through the grid or by giving individual external voltage inputs. This choice would be made by connecting the appropriate header pins using a jumper. The external voltage supply was provided to allow for powering down idle system components and also quantify power drawn from each source independently. Current sense resistors were used in the voltage path to allow for making voltage measurements across them in order to calculate the power. These resistors have low resistance values and high current tolerances.

The DE2 board was provided with a USB blaster to program the FPGA. Cyclone II FPGA supports the following configurations modes: Active serial (AS) mode, Passive serial (PS) mode and JTAG mode [**4**]. In AS mode, the device is programmed by a simple low cost serial configuration device where the device reads in the configuration data and generates the control signals on its own. In PS mode, the device is programmed by an intelligent host i.e. a microprocessor that provides the configuration data as well as the control signals. An intelligent host provides the data and control signals in JATG mode as well but the data is downloaded into the device via a JTAG cable. The DE2 board gave the user the ability to program via the computers USB port and also in conjunction with a CPLD provides the ability to store a design that gets loaded into the FPGA during device power up. This feature was not required for this thesis design, so it was decided that the simple JTAG programming mode would be implemented. Section VI of the cyclone II handbook [**4**] was referred to for the circuit diagram to implement this mode. Apart from being simple, all that you needed for implementing this mode was a header to connect the programmer cable to the board while the use of the USB blaster

meant that additional components that needed to be powered by the on board supply were required.

For the schematic building process, the part symbol of the FPGA was downloaded from the Altera website and all the other part symbols not available in the software's symbol library were created using the create part option of the software's part manager tool. With a lot of the components in the original schematic not being utilized, there were a large number of FPGA I/O pins that were now unused. The cyclone II datasheet suggested that the unused pins not be left floating. The unused pins need to be either tied to ground or to the supply voltage. It was decided to ground all of the unused FPGA pins. After the schematic for the entire design was built, it was exported to begin the layout process.

5.2.2 Preparing for layout

After the schematic is built and checked for design errors, it is time to move on to the next design step and layout the circuit and design the PCB. The tool used to perform the circuit layout and design the PCB was Cadence – Orcad's layout tool; Layout Plus. The schematic capture program generates a netlist file that is read into layout plus to perform the layout process. Analogous to the part symbol in a schematic, every component in a layout is represented by its "footprint". The foot print is the copper area on the circuit board onto which a component will be soldered to when the PCB is assembled; these are sometimes referred to as land patterns. It is very critical that the footprint be correct, any discrepancy can lead to improper connection of the component to the rest of the circuit and result in improper system functioning.

So the first step after finishing the schematic design and before beginning to layout the circuit is to make sure that you have footprints defined for all of the components that you are going to use in your design and have them assigned to their respective components before

generating the netlist. The layout tools have libraries that contain many of the standard footprints for many package types. If the footprint library does not have the footprint of any of your components you can do one of three things. First, you can go to the device manufacturer's website and download the footprint if they have it available for your layout tool. Second, you can modify an existing footprint in the program's library if it matches the package type but it does not have the same dimensions. Or third, you can create the footprint from scratch.

The fist method is not suggested and even if the manufacturer does have the footprints available for download, the chances of a version compatible to your program depends on the program you are using i.e. if it is a program widely used in the industry or a program meant for a hobby designer. So if the components package type is available in the library, the second method is the easy solution. Fig 5.11 shows the footprint for a standard 676 pin BGA which needs to be modified to a 674 pin footprint.

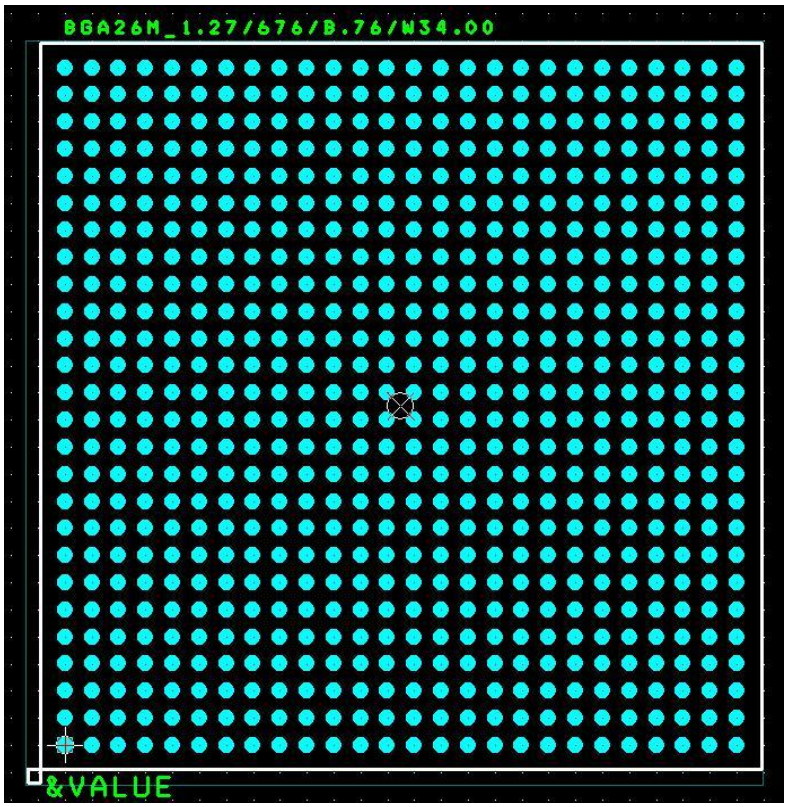

Figure 5-11: 676 pin BGA foot print in the standard library

If the component package type is not in the library, most of the time an existing footprint of a package similar to it can be modified, to avoid errors it is advised you start the footprint from scratch. The component's datasheet have the components mechanical information that you can use to make the correct footprint. Some datasheets also have a recommended land pattern defined in them to help the designer. Shown in fig 5.12 is the package outline for the cyclone II FPGA [**4**] used in this thesis work. It is a 672 pin fine line Ball Grid Array (BGA) package and the package outline has information like distance between the centers of the pins, width of the pin, length of the device package, width of the device, its clearance from the flat PCB surface and other such mechanical information to help the designer design the appropriate footprint to house the component on. These dimensions are given in inches or millimeters and all the dimensions are in the same units. If manufacturer provides the dimensions in both metrics, one

of them is in parenthesis, for example, if in the datasheet, the two metrics indicated are inches and millimeter and millimeter is given in parenthesis, it is denoted in the legend as inches (mm).

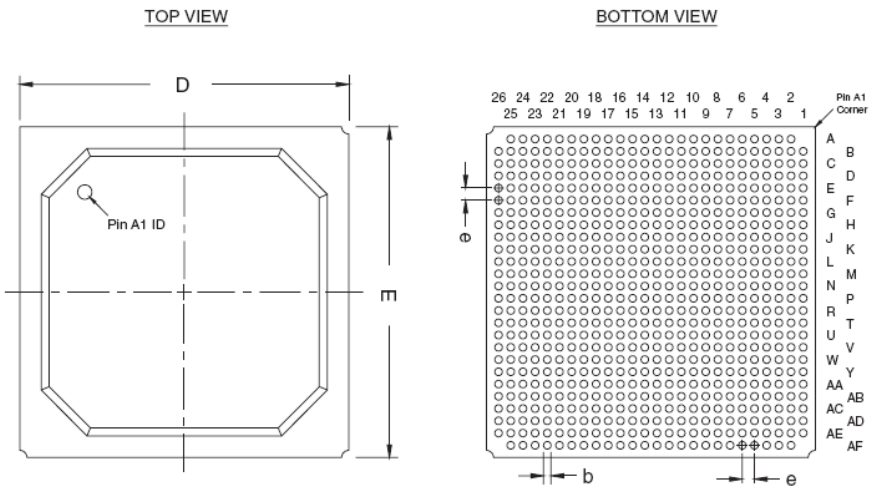

Figure 5-12: 674 pin BGA package outline [25**]**

In the datasheet of the FPGA, the manufacturer provides the designer with the numeric values for the package outline dimensions denoted by e, b, D and E in a table format similar to the one shown below. The units of the dimensions are also provided in the table.

Symbol	**Symbol**		
	Min.	Nom.	Max.
D	23.00 BSC		
E	23.00 BSC		
b	0.5	0.6	0.7
e	1.00 BSC		

Table 5-1: Package dimensions [25**]**

This information is used to modify the existing footprint. The corner four pads in the standard BGA are removed to get the arrangement of the pads as shown in the outline and the distances between the pads are modified along with the size of the pads to the value given in the table to create the desired 674 pin BGA footprint required for the cyclone II FPGA.

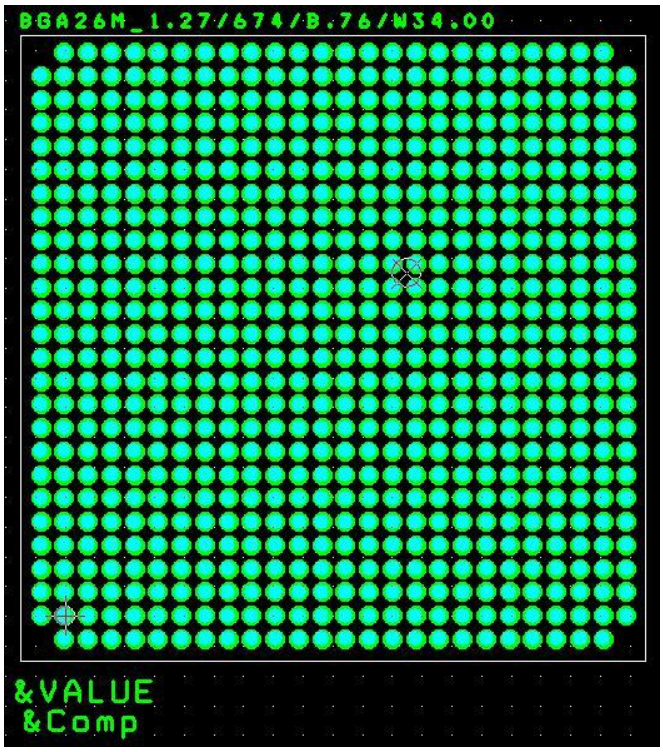

Figure 5-13: Modified 674 pin BGA footprint

Another thing a designer needs to keep in mind is having a proper padstack. A padstack is the dimensions of a footprint pad defined on each of the layers of a PCB. When making footprints you have to make sure the padstack is compliant with the standards for PCB tolerances of the PCB manufacturing company. The company's website will have a list of minimum specification for spacing and other considerations that they can manufacture. For example, PCB manufacturers require a minimum of 10 mils or 0.010 inches of inner layer clearance i.e. if you have a through hole on a circular pad to mount a component and the hole goes from the top layer of the PCB down to the bottom layer through an inner layer, the radius of the pad in the inner layer must be 10 mils larger than the one on the top and bottom layers.

In summary, before the netlist file can be generated for the design process to move on to the next step i.e. the board layout, the designer, keeping the fabrication specifications and the device

mechanical dimensions in mind, needs to create the footprints of all the components and assign them to their respective parts in the schematic.

5.2.3 Design layout

In this section, I will describe the steps that I took in performing the layout process. Individual layer and layer stack details and the details of component placement and interconnect routing done will be discussed in this section.

5.2.3.1 Layer stack

After all of the footprint assignments have been made and the schematic has been checked for any design errors, the netlist is generated and imported into Layout Plus to begin the layout process. Before you start placing components and routing any interconnects, the layer structure of the PCB needs to be defined. The designer has to decide the number of layers the PCB is going to have and what purpose each of those layers is going to serve. Each layer can be used for one of four purposes.

- The layer can be used to route the signals between the board components
- The layer can be used as a power layer used to route power to the entire board (usually the inner layers)
- The layer can be used as a ground layer (usually the inner layers but sometimes on two layer boards the bottom layer is the ground layer)
- The layer can have the components placed on them with some minor routing (usually the top and bottom layers)

The layer stack up is also important for designs with high frequency digital or analog signals. For example, in a multi layer board where an inner layer is used for routing, a power layer or

ground layer is stacked close to it to reduce the signal loop. Also, interference between high frequency signals routed on two layers is prevented by placing a power or ground layer between them isolating them from each other. Stacking a power plane over a ground plane utilizes the distributed capacitance between the two planes, functioning as a decoupling capacitor. The distributed capacitance is used for decoupling in cases where the frequency is greater than 50 MHz. At these high frequencies, discrete capacitors become ineffective at providing decoupling.

The layer stack up for the test board is shown below. The PCB has 6 layers with the top layer housing all of the components and the some routing between the components is done on this layer. The inner layers serve as power and ground layers; the power layers are primarily used as voltage supply for the BGA, so any area on the layers that is not serving the BGA is used for routing. The bottom layer is mainly a routing layer but also has some the decoupling capacitors housed on it.

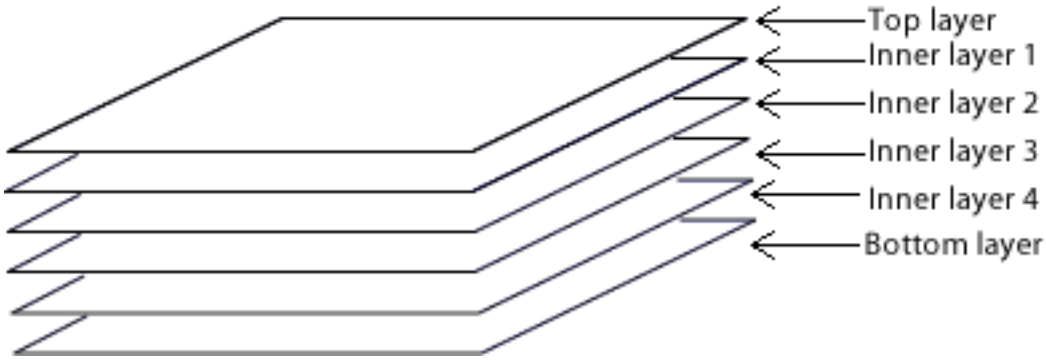

Figure 5-14: Test board layer stack up

The BGA needs to be supplied with two voltages; VCCIO which is 3.3 V and VCCINT which is 1.2 V. Inner layers 1 and 3 were used to supply the BGA with these voltages. The area of the board where the power needs to be delivered is small, so the rest of the layer was chosen to perform some of the signal routing for the other components on the board. Inner layer 2 was split and designated as the digital and analog ground layer because as mentioned earlier, it

reduces signal loop for the digital signal and analog signal traces that are routed on the layers above it. And finally, inner layer 4 and the bottom layer are used primarily for routing the BGA pins. The table below summarizes the layer stack of the PCB designed.

Layer	Order	Function
Top	1	Component placement and routing
Inner 1	2	Supply plane for the BGA and routing
Inner 2	3	Split ground plane
Inner 3	4	Supply plane for the BGA and routing
Inner 4	5	Routing layer
Bottom	6	Routing layer

Table 5-2: Test board layer functions

5.2.3.2 Component placement

After the layer stack has been defined; we can start the layout process by placing the components onto the top layer of the board area. The top layer of the PCB is shown below

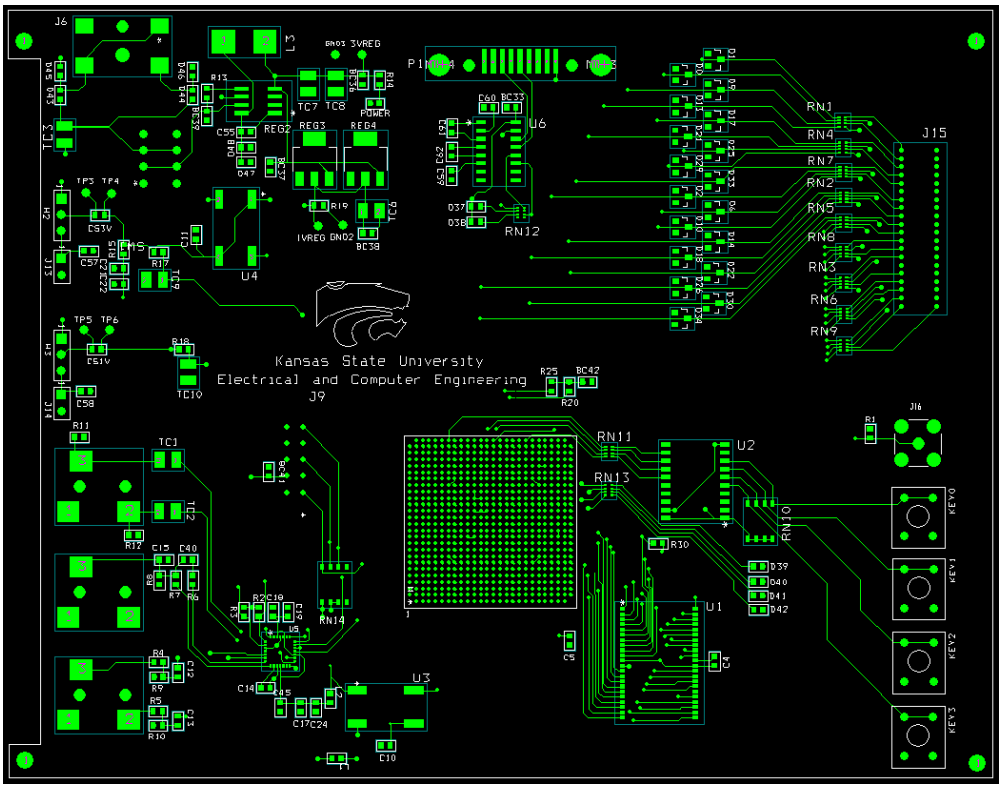

Figure 5-15: PCB Top layer

The components are placed in a manner that allows for efficient routing of the interconnect signals. The component placement is up to the designer and there are not many rules that need to be followed except for the pad to pad spacing specification of the fabrication company and the fact that it should make the routing easy. The PCB top layer can have text and line art printed on it by screen printing, this is called the silk screen. The silk screen is used to indicate the component reference designators, test points and other information which can be helpful in assembling and testing the circuit board.

On the top layer you can see the FPGA near the center of the board and the SRAM memory with reference designator U1 near the FPGAs bottom right corner. The analog inputs are on the left edge of the board, the push buttons the SMA connector are on the right edge along with the expansion header on the top right corner of the board. The top left area of the board has the power supply circuitry and top center of the board we have the serial port. The bottom layer is also used for component placement. Here you can see the bypass capacitors near the board center and schottky diodes on the top right corner of the board.

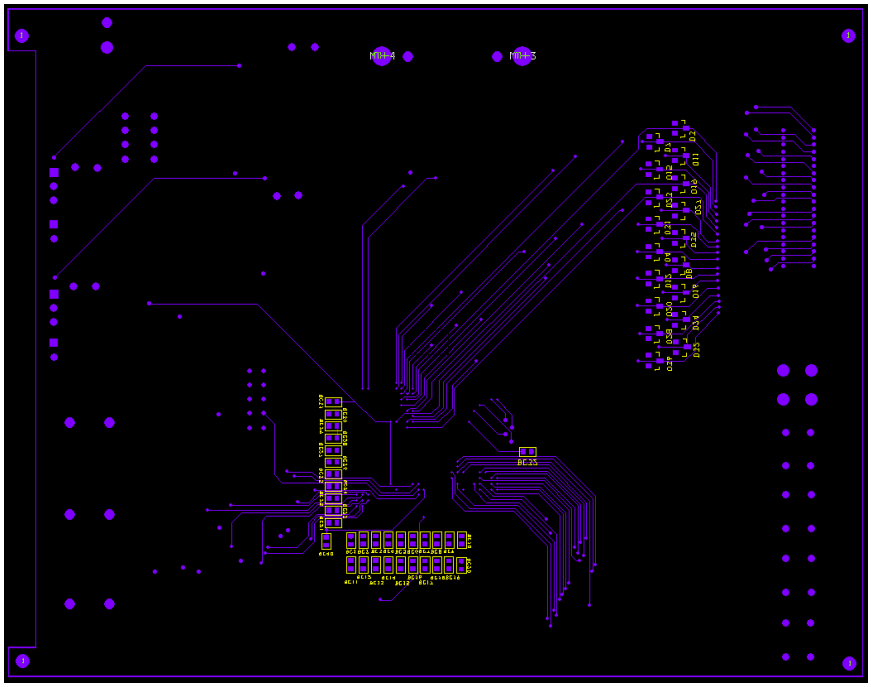

Figure 5-16: PCB Bottom layer component placement

5.2.3.3 Defining power planes

Now that the components have been placed, it is time to begin routing. It is suggested to start routing the power and ground signals before the other interconnects, or at least have the power and ground planes defined (If you have any) before the other signals are routed. The BGA needs two voltages; 3.3 V and 1.2 V; to be supplied to many pins distributed throughout the BGA package. The BGA is a fine line BGA and it is not advisable to route the power to each pin via a

separate power line. So, the power is supplied to a copper plane underneath the BGA and the pins that need the 3.3 V or 1.2 V are connected to the corresponding plane. Inner layer 1 is used to provide the VCCIO voltage to the BGA. The figure here has the plane layer defined as the rectangle near the board center; you can also see the routing done on the layer's unused region.

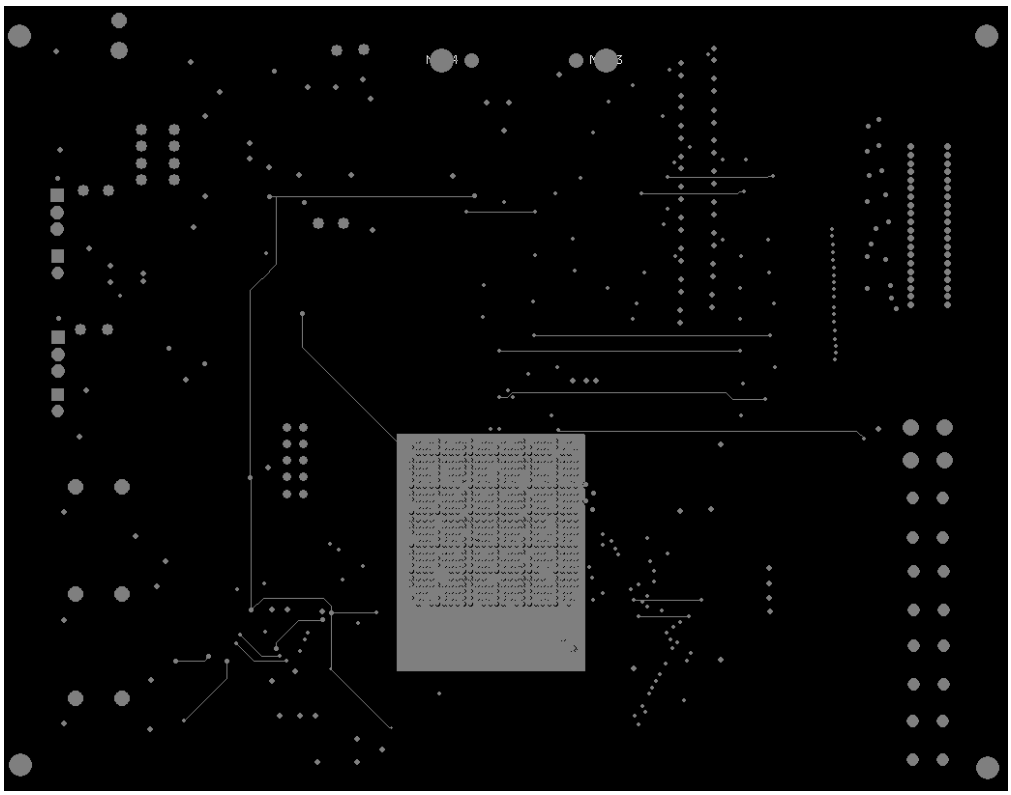

Figure 5-17: PCB Inner layer 1

The plane is defined in layout plus by drawing an obstacle. The obstacle type is a copper pour which indicates that when the board is being fabricated this region needs to be filled with copper, forming the plane layer. The voltage is supplied to the layer by burying a trace carrying the desired voltage into the plane. This floods the entire copper plane with the desired voltage and any pins connected to the plane can draw that voltage.

When the copper pour area is created, there are a few specifications to keep in mind. They are the copper pour's width and clearance. Let's look the clearance first. When a via is used to

route a trace to an inner plane layer or ground layer, it connects to the layer via a thermal relief, these are the protrusions connecting the circular inner pad to the plane layer in the figure below, and if it has to pass through other inner layers, there has to be a separation between the intermediate inner layers and the via pad on that layer. This is seen as the gap between the three circular pads and the plane layer. This separation is called inner layer clearance or just clearance. This clearance is important and it prevents the vias from connecting to the wrong inner layers. Every fabrication company has a minimum clearance value that they can fabricate.

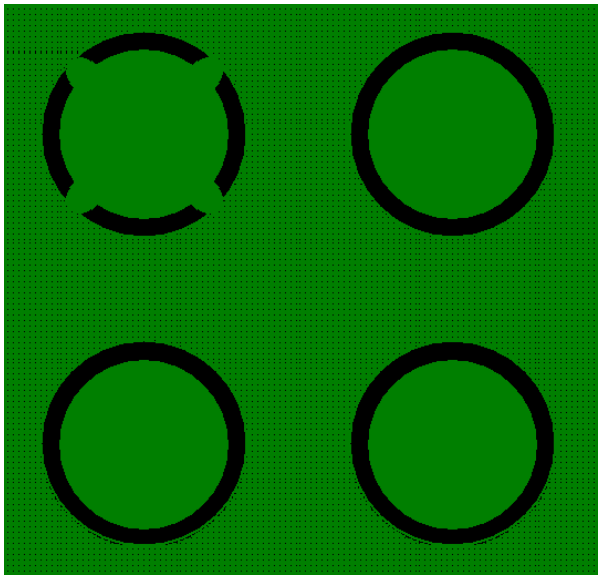

Figure 5-18: Via thermal relief and inner clearance

In width value is more of a program directive. In the case of two adjacent vias, the width is the distance between clearance area outer circles of the vias. These two specifications are given in mils or inches. The minimum clearance value is 5 mils or 0.005 inches and the width can be as low as 1 mil. Having low clearance values can lead to improper fabrication result in vias connecting to the wrong layers while larger clearance values can lead to the formation of islands in cases where there are a large number of via holes in a small are in the board.

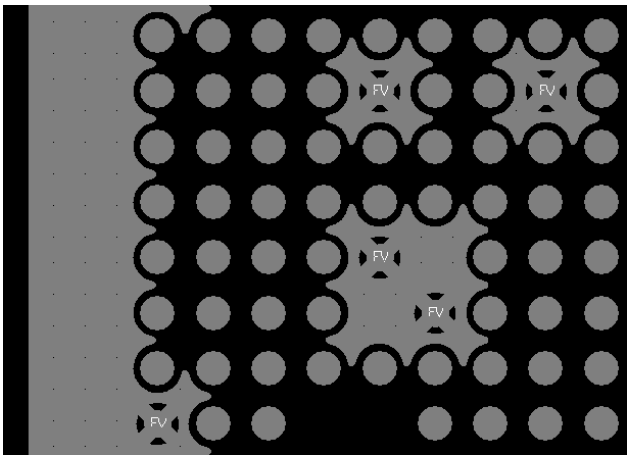

Figure 5-19: Copper pour islands

The figure below shows a section of the VCCIO plane layer created in the design layout with a clearance value of 5 mils and a width value of 1 mil.

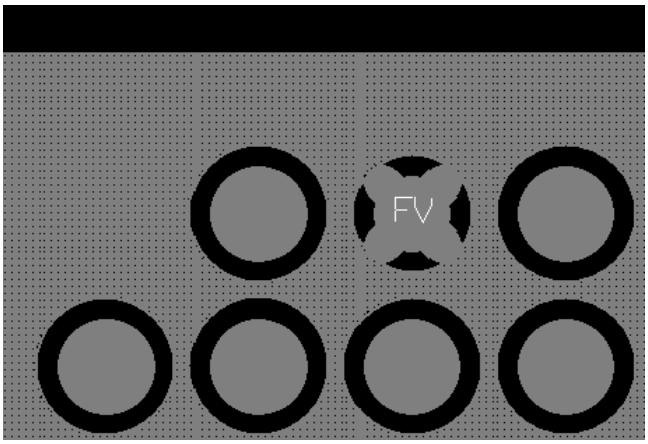

Figure 5-20: Copper pour with proper clearance and width

The connection to a plane layer can be made either through the thermal reliefs or the plane can be flooded. When the plane is flooded, the clearance area is filled with copper; as shown below. Flooding is not suggested for vias connecting to a large plane layer because heat applied for soldering is dissipated into the large conducting layer leading to improper soldering of components. If more heat is applied to perform the soldering, it might damage the component.

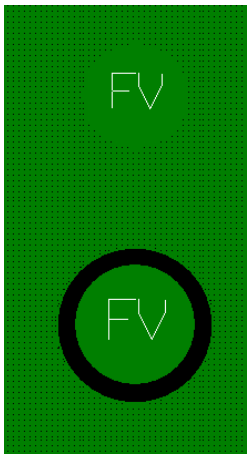

Figure 5-21: Via connection flooded onto the plane

The VCCINT supply is given to the BGA in the same manner on inner layer 3 near the center of the board and a 3.3 V supply is provided to the dual schottky diode packs in the top right area.

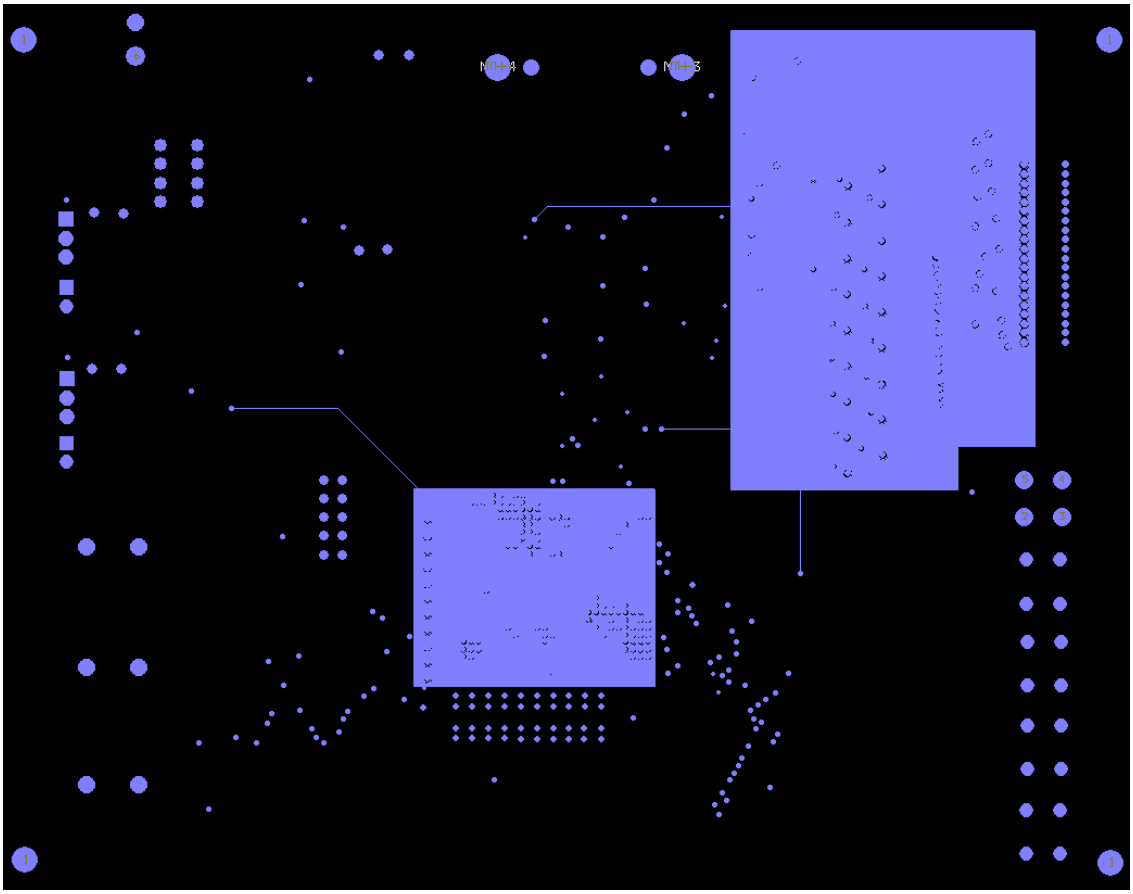

Figure 5-22: PCB inner layer 3

5.2.3.4 Ground planes

The design uses the audio codec for the application's analog module [**26**], so there is a need to have a separate analog and digital ground. This is recommended because; digital signals are characterized by rail to rail voltage switching, while the analog signals vary between the rails. So if they share the same ground, the digital switching injects noise into the analog part of the circuit. A single layer can be split to form the analog and digital ground planes or the two planes can be on separate layers. In the case of a split ground plane, care must be taken during routing because, if a digital signal is routed over a split ground plane, the return loop path changes from the digital ground to the analog ground adding switching noise into the analog ground plane. This can be prevented by proper routing of the signals by making sure all the digital signals are routed over the digital ground plane and the analog signals are router over the analog ground plane. The figure shows the split ground plane design on the PCB inner layer 2.

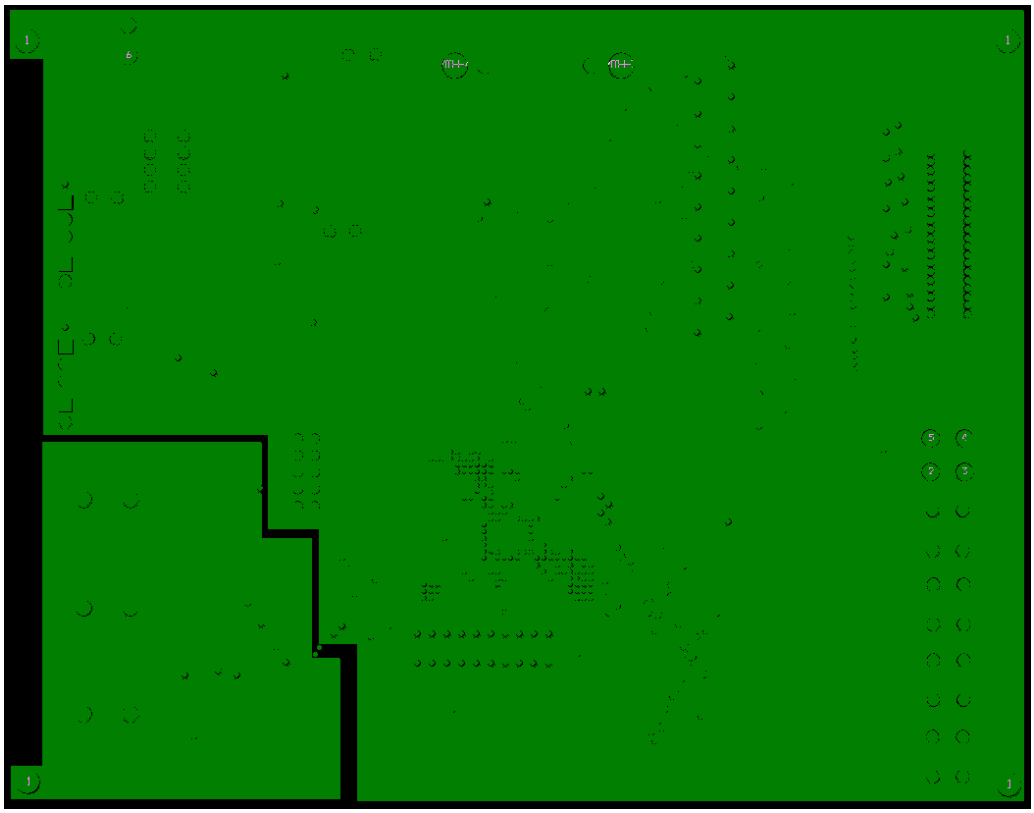

Figure 5-23: PCB Inner layer 2

It is important from the circuit's point of view that the analog and digital ground be connected at a point to prevent the analog and digital signals from floating relative to each other causing undesired effects. So the two grounds are connected together by a ferrite bead which acts as choke to the switching noise. Separating the analog and digital voltage supplies with the ferrite bead is also a god idea. Any variation in the digital supply voltages can adversely affect the signal quality in the analog circuit.

5.2.3.5 Interconnect routing

After all the components are placed and the plane and ground layers are defined, all that needs to be done is the interconnect routing. As seen in fig 5.15, the routing for most of the components was achieved on the top layer itself. The routing process for most of the components was pretty straightforward and also the analog signals were routed over the analog ground and the digital signals were routed over the digital ground successfully; achieving the desired signal loop isolation between the two interconnect groups.

The only challenging component to route was the FPGA. The FPGA is a 674 pin Ball Grid Array with all of the pins located underneath the package. This particular package was a very fine line BGA package with 1 mm distance between the pins whose diameter was 0.6 mm nominally [**4**]. This made it impossible to route the pins out laterally on a single layer. To address this, it was decided to first bring out the pins to different lower layers and then try to route them out to the other components laterally. With the plane and ground layers being inner layers, all the pins of the BGA ended up being routed to inner layers.

For each of the layers that a pin needed to be routed to, a unique via was created. Every layer required a unique via, because a general purpose via meant for all the layers would occupy pad space even on the layers it was not connected to. So the via needed to start at the top and end at

the inner layer it needs to route the trace to. A via hole that does not go through all of the board layers, but terminates in an inner layer is a blind via. A via that starts in an inner layer and ends in another inner layer without reaching either the top or bottom layer of the board is called a buried via.

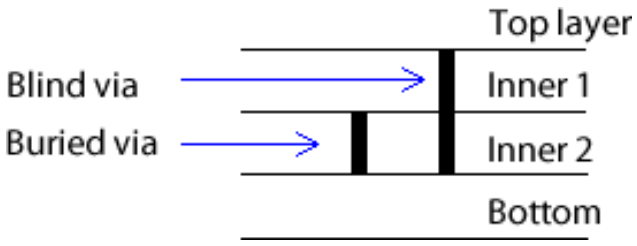

Figure 5-24: Blind and Buried vias

Five sets of blind vias were created in order to route the BGA pins from the top layer to all of the inner layers and through via for the pins routed all the way to the bottom layer. To save component space on the top layer, some of the decoupling capacitors for the FPGA's voltage supply were placed on the bottom layer. To connect them to the appropriate power plane and also to the ground plane, two more sets of blind vias were created.

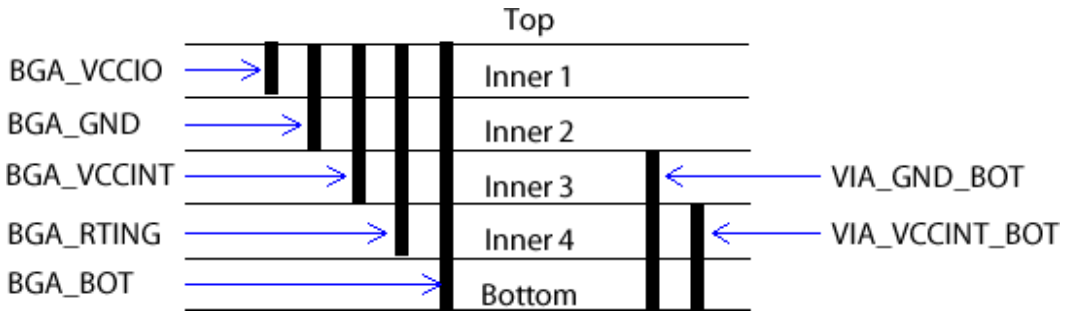

Figure 5-25: Vias created for BGA routing

This via shown in fig 5.26 is BGA_VCCINT and it is used to route a BGA pin from the top layer down to inner layer 3 where it connects the pin to the plane layer used to supply the 1.2V VCCINT voltage to the BGA. You can see in the figure that on the layers the net does not

connect to, there is a clearance area between the pad and the intermediate layers and on inner layer 3 the pad connects to the plane using thermal reliefs as described earlier.

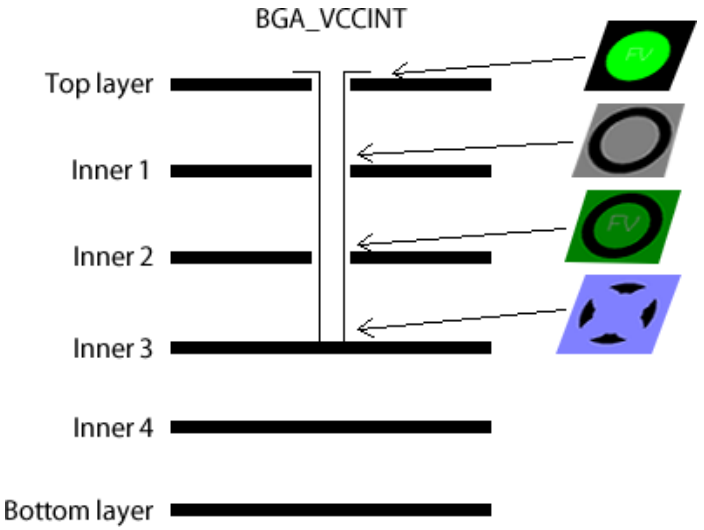

Figure 5-26: Via BGA_VCCINT

After the power and ground pins were attached to their respective plane layers, the I/O pins were routed. The figure below shows the location of the I/O pins that need to be routed on the package. The traces in pink were routed on inner layer 4 while the traces in red were routed on the bottom layer. While there are a few pins that needed to be routed out of the middle of the package, the fact that a majority of the pins were in the edges reduced the routing difficulty a little. Ideally, it would take 5 layers (3 for routing and 2 ground planes isolating the routing layers) to route these pins. The reason for all those layers is to avoid routing two traces; on adjacent layers; on top of each other. The switching activity of the signals on these traces will cause capacitive coupling between the two, interfering with the signals on the traces. This capacitive coupling depends on the frequencies of the signals travelling on the traces. But as the number of layers increase, the cost of fabrication increases as well.

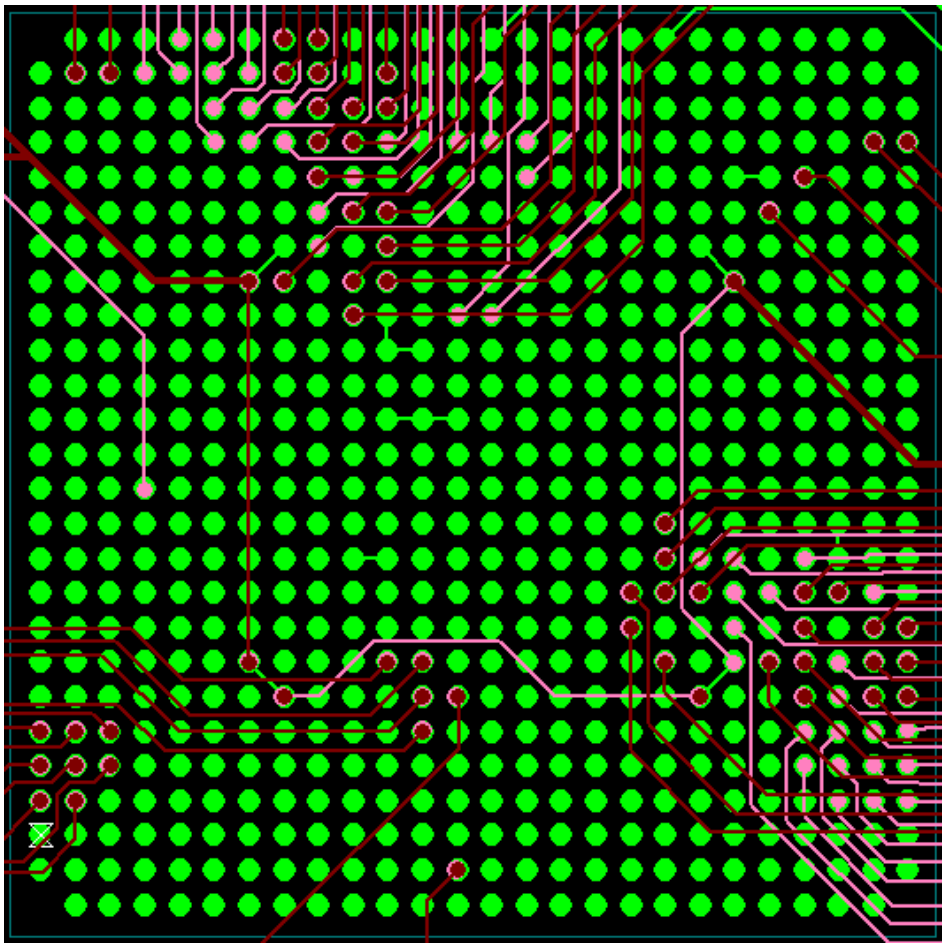

Figure 5-27: FPGA routing on Inner layer 4 and Bottom layer

In this design, the I/O pins that need to be routed on these two layers are used to connect the FPGA to the SRAM and the expansion header. It can be safely assumed that these signals are not very high frequency; in fact the 50 MHz signal is the highest frequency in the entire design. At this low frequency, the capacitance effect can be ignored and hence these traces are routed very close to each other adjacent layers and in some cases they run parallel on top of each other as shown in figure 5-28.

The interconnect routing to the other components that could not be done on the top layer were performed in the inner layers. A few other sets of buried vias were designed to assist in their routing. In the routing purposes, the trace width and the trace spacing are two important

specifications. The trace width needs to be small enough to allow the BGA pins; located a few rows in from the edges; to be able to rout them out successfully. The fabrication companies specify the min trace width that they manufacture as 5 mils wide. This was chosen as the signal trace width for the routing purposes in this layout. The traces that route the power are usually wider than the other signal traces; a width of 10 mils was used. For proper fabrication of the traces and pads, the fabrication companies provide specifications for the minimum spacing between traces as 5 mils and 6 mils is the standard value. The spacing between a trace and an adjacent pad should be a minimum of 5 mils and 8 mils is the standard value. For vias we have to consider two spacing values. The clearance value of 5 mils as mentioned above is one of them, and the other is spacing. The size of the pad on a layer should be made larger than the size of the via drill hole; the standard value being 5 mils. These two dimensions together are called inner layer clearance, with 10 mils being the minimum value.

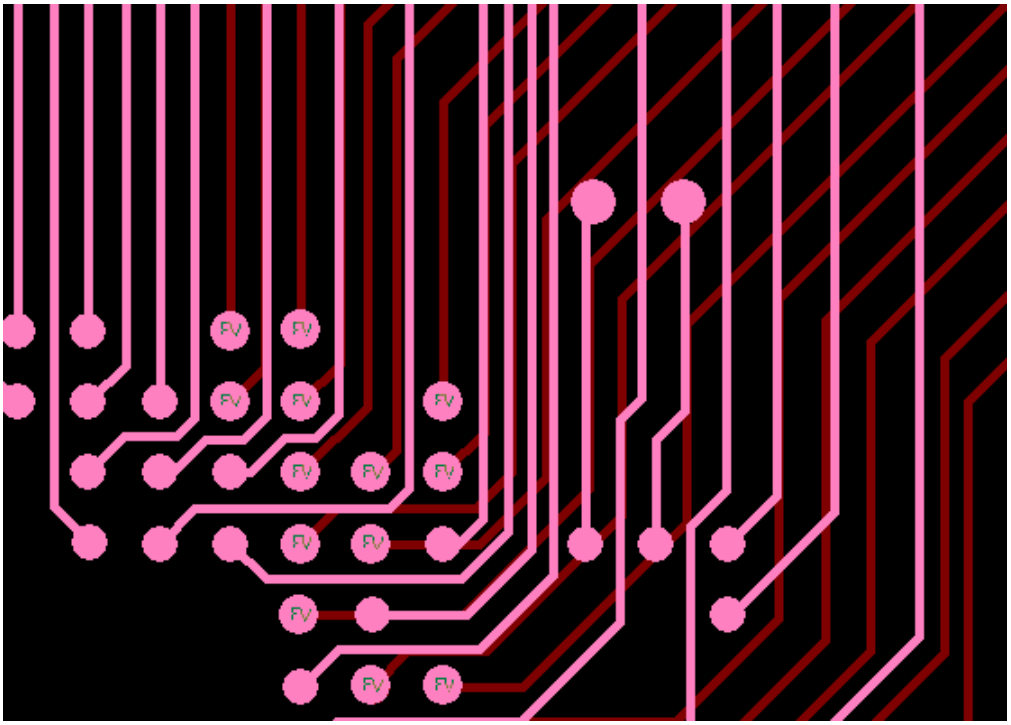

Figure 5-28: Parallel trace routes on adjacent layers

Chapter 6

Design power optimization and preliminary analysis

In this chapter I will discuss the low power design methodologies implemented in the design process. The implementation on each individual module is discussed and the optimization achieved is shown. First I will describe the simulation strategy and how it was used to test each of the modules, followed by a comparison of the power analysis results.

6.1 Design power analysis strategy

As discussed in an earlier chapter, the PowerPlay power analyzer in the Quartus II software is an efficient tool for device power consumption analysis. The designer needs to provide the tool with the design's complete signal activity details and the environmental and operating conditions. As mentioned earlier, if the complete signal activity is not available to the tool, vectorless estimation is used to supplement the missing activity information.

The signal activity information is gathered by performing simulation on the post fit netlist generated by the design software and saving this information in either a .saf or a .vcd file. It is suggested to use a .saf file because .vcd files of moderately complex designs can end up being very large in size. Third party simulation software compatible with Quartus II can be used to generate the .vcd file needed for this purpose. Quartus II design software is provided with a simulator that can be used to generate the .saf file. A good gauge of the accuracy of the power analyzer results is the confidence metric calculated during the power analysis. This is provided in the analysis reports; a high confidence metric is ideal.

For this thesis work it was decided to generate .saf files to be used for the power analysis. The Quartus II simulator was used to perform the simulation and generate the desired signal activity file. In the Quartus II simulator the simulation inputs are user generated waveforms that

represent the input signals. The waveforms are called test vectors and Quartus II software generates a vector waveform file (.vwf) that is given as input to the simulator. This simulator does not support the use of testbenches to generate the inputs but it supports the use of a .vcd file describing the input test vectors and the simulation output can be exported as either a ,saf or .vcd file.

Creating the input test vectors in Quartus II is a tedious task and for this thesis work it would not be practical to draw out the high frequency input waveforms lasting enough time to perform a proper simulation. For this reason, Modelsim-Altera simulation tool was used to generate the input vectors. Testbenches were written in verilog describing the input vectors and a .vcd file was generated that described the input signal activities. This .vcd file was read into the simulator and the simulation was performed to generate the .saf simulation output file, containing the signal activity information of all of the nodes in the design. This file was then read into the power analyzer to perform the analysis.

The testbenches used to generate the input vectors that were used to make the preliminary device tests and final design power analyses are provided in the appendices. All of the aforementioned operating modes of a FPGA device were simulated and their resulting signal activity files used to perform the power analysis. The different operating modes that were simulated and their power analysis results are discussed in the next chapter. The individual modules were also simulated and the effectiveness of the power conservation technique implemented in them is checked. The results of these tests on the individual modules are discussed in a later section and the testbenches used to generate the input vectors are provided in the appendices.

Figure 6-1 shows the flow of the test strategy described above.

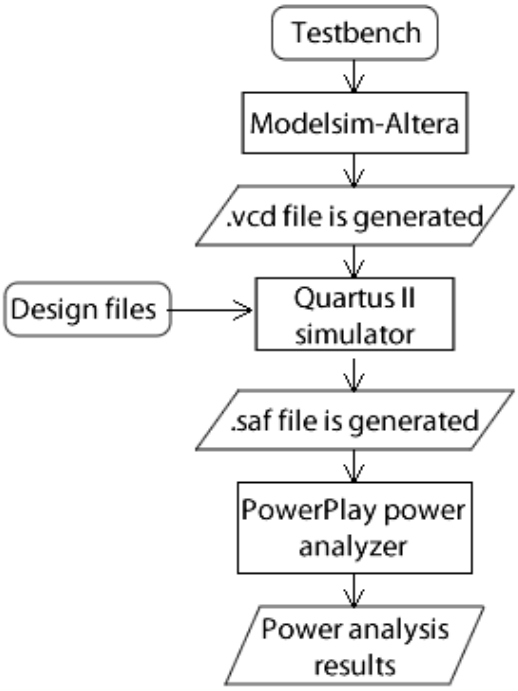

Figure 6-1: Design power analysis flow implemented

The device's operating temperature conditions set for the power analysis are given below. The power analyzer uses this information along with the signal activity data to perform a complete power analysis.

- Junction temperature range: 0 °C to 85 °C
- Ambient temperature: 25 °C
- Cooling solution: 23mm heat sink with 200 LFpM airflow, with
    1. Junction to case thermal resistance: 3.1 °C/W
    2. Case to heat sink thermal resistance: 0.10 °C/w
    3. Heat sink to ambient thermal resistance: 2/80 °C/W
- Board thermal model: Typical model used, with
    1. Junction to board thermal resistance: 5.50 °C/W
    2. Board temperature: 25 °C

6.2 Preliminary device analysis

This strategy was used to test different FPGA devices and choose the FPGA best suited for the purpose of this thesis based on the results. The table below summarizes the results obtained.

Device family	Device	Core dynamic thermal power dissipation(mW)	Core static thermal power dissipation(mW)	I/O thermal power dissipation(mW)	Total thermal power dissipation(mW)
Stratix II	EP2S180F1020C3	36.99	1360.47	42.74	1440.19
Stratix III	EP3SL340F1517C2	29.3	1177.73	76.94	1283.97
Arria GX	EP1AGX90EF1152C6	28.93	791.31	30.87	851.11
Arria II GX	EP2AGX260FF3515	12.69	807.78	54.48	874.95
Cyclone	EP1C12F256C6	9.69	82.64	4.94	97.24
Cyclone II	EP2C35F672C6	7.97	76.76	41.48	129.21

Table 6-1: Preliminary FPGA device power test results

The results show that the Cyclone II FPGA family's device EP2C35F672C6 consumes the lowest dynamic power when compared to the other devices tested using the same design. While the results show that the Cyclone EP1C FPGA tested consumes the least total power of all of the devices. Compilation summary showed that 99% of the available logic resources of this device were consumed for this design and the device chosen was the biggest in the family. This means that there is no room left in the device for any future design changes or addition of any new features. Also, an interesting point to notice is the very low I/O power dissipation. The reason for such loe power dissipation is that the 99% resource utilization So, The Cyclone II family's EP2c35F672 FPGA was chosen for the purposes of this thesis work. The confidence metric was high for all of these simulations.

6.3 Individual module analysis

In this section I will describe the low power design technique used in each of the modules and discuss the results of the power analysis done on those modules. These modules are implemented on the Cyclone-II EP2C35F672 FPGA.

6.3.1 Serial module

This module is responsible for receiving data transferred on the serial input line to the design and set up the signals so it can be written into the memory. The technique suited for this module is clock gating where the clock to the module is turned off when the module is inactive.

If no input data is being sent, the serial data line is set high. This condition can be used to see if the design is inactive or not. In this design, a clock control module is implemented that waits for a specific amount of time for data to be sent on the input line. If no new data is seen on the line in that time interval, the clock to the module is turned off. The block diagram of this implementation is shown below. During normal operation, if there is no data on the serial line, the clock to the module is turned off. If new data is seen on the input line, the module is enabled and normal operation is resumed.

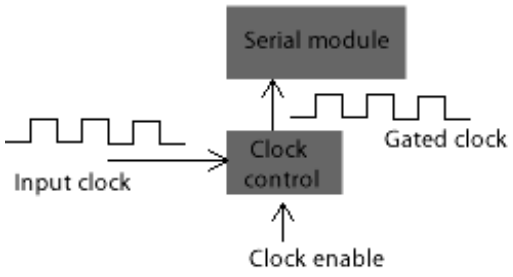

Figure 6-2: Serial module clock control block diagram

This technique is designed to reduce the dynamic power consumption of the module. This module is tested by performing two sets of simulations. The first is the serial module implemented without any clock gating logic and in the second simulation, the module was

implemented with clock gating logic and the clock to the module turned off to see the saving in dynamic power. The table below summarizes the simulation results:

	Logic Utilization	Core Dynamic power (mW)	Core Static power (mW)	I/O thermal power (mW)	Total Thermal power (mW)
Serial module without clock gating	74 / 33,216 <1%	1.75	79.69	2.38	83.82
Serial module with clock gating	121 / 33,216 <1%	0.31	79.69	2.56	82.56

Table 6-2 : Serial module power analysis results

In table 6-2 we see a small increase in the logic occupied on the device by adding the clock gating logic, while the dynamic power has reduced by 82.3 % when the clock to the module is gated.

An example application could be a data logger that reads serial data from sensors and sends it to a memory module to be stored. Let us assume that serial module is active only for five minutes every hour to take the sensor readings and is idle for the remaining time. If no clock gating is performed for when the module is idle, over the course of a thirty day month, the module would have consumed 1.26 watt hours of dynamic energy. If clock gating is implemented, the module consumes 0.3096 watt hours of dynamic energy which is a saving of 75.4% in the dynamic power consumption over the thirty day period.

6.3.2 DSP module

This module consists of the FIR filter which requires the MAC operation to be implemented. The use of the DSP block available in today's FPGA devices is said to help in the reduction of

the logic area consumed by the design and also reduce the power consumption of the design. The effect of clock gating on the dynamic power consumption is also studied for this module.

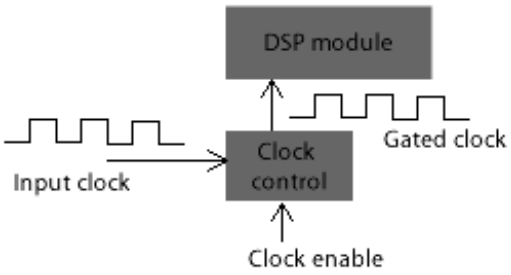

Figure 6-3: DSP module clock control block diagram

For this module the first set of simulations were done to see the effect of using the embedded DSP blocks on the logic area occupied by the design. The results are shown in the table below.

	Total logic elements utilized	Total registers utilized	Embedded multipliers utilized
Filter of length 32 implemented in logic	11,533 / 33,216 (35 %)	999	0 / 70 (0 %)
Filter of length 32 implemented in DSP blocks	1,563 / 33,216 (5 %)	503	62 / 70 (89 %)
Filter of length 64 implemented in DSP blocks	1,624 / 33,216 (5 %)	524	64 / 70 (91 %)

Table 6-3: Filter module device utilization test results

It is evident from the table that using DSP blocks for the MAC function implementation of the FIR filter has significant effect on the logic area occupied by the design and the number of registers used. A 30% reduction of logic element utilization is seen and a 49.6% reduction in the register utilization is seen. The filter module with double the length was also implemented and it can be seen from the table that the increase in resource utilization is minimal.

Power analysis of the three cases mentioned above was also done and the results are shown in table 6-4. It is seen that there is not a significant reduction in the total power consumption with the design mapped in DSP blocks as compared to the design mapped in logic. The reduction in

dynamic power is 17%. Also, there isn't any change in the power consumption values for the 32 length FIR filter and the 64 length FIR filter implemented in the DSP blocks of the device. These results show that while mapping logic in DSP blocks might not improve the power consumption significantly, they do reduce the logic element utilization which can be very helpful in large and complex designs

	Core dynamic power(mW)	Core static power(mW)	I/O thermal power(mW)	Total thermal power(mW)
Filter of length 32 implemented in logic	2.05	79.74	34.61	116.41
Filter of length 32 implemented in DSP blocks	1.69	79.74	34.61	116.04
Filter of length 64 implemented in DSP blocks	1.7	79.74	34.61	116.05

Table 6-4: Filter module power consumption

The next set of simulations and power tests were done to see the effect of clock gating on the module's dynamic power consumption. The clock gating logic was implemented on the 32 length FIR filter implemented in the DSP blocks. The results of the power analysis are shown in Table 6-5.

	Core dynamic power(mW)	Core static power(mW)	I/O thermal power(mW)	Total thermal power(mW)
Filter of length 32 implemented in DSP blocks without clock gating	1.69	79.74	34.61	116.04
Filter of length 32 implemented in DSP blocks with clock gating	0.19	79.74	34.50	114.43

Table 6-5: Effect of clock gating on filter module dynamic power consumption

The effect of clock gating is significant as seen in the table above. There is an 88.75% reduction in the dynamic power consumption with the implementation of clock gating i.e. 88.75% of the power was saved by turning off the idle module.

An example application would be a module that is looking for a wireless control signal in a certain frequency band that utilizes this module to filter out the unwanted frequencies. The module operates for only five minutes in every hour looking for the control signal and is idle the rest of the hour. The length of the filter would not drastically affect the power consumption as seen in table 6-4. The module would consume 1.22 watt hours of dynamic energy over a period of thirty days. If clock gating is implemented to shut down the module when it is idle, the dynamic energy consumed reduces to 0.23 watt hours of dynamic energy for the same duration which is a saving of 81.2%.

6.3.3 Memory module

This module is responsible to set up all the signals required to successfully read and write to the external SRAM. Clock gating was implemented in this module and its effects on the dynamic power consumption are studied.

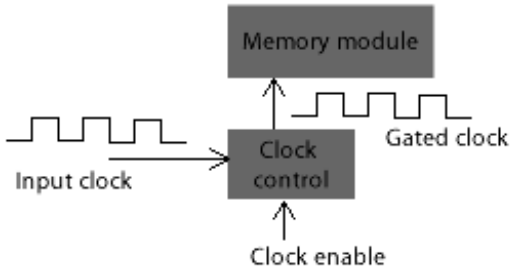

Figure 6-4: Memory module clock control block diagram

The results of the power analysis done are shown below:

	Core dynamic power(mW)	Core static power(mW)	I/O thermal power(mW)	Total thermal power(mW)
Memory module without clock gating	1.6	79.75	41.59	122.94
Memory module with clock gating	0.01	79.75	41.52	121.29

Table 6-6: Effect of clock gating on memory module

The effect of clock gating is very evident from the values in the table above, the reduction is 99.4%. The other thing to notice is the high I/O thermal power compared to the other modules. This is because of the constant activity of the address, data and control signal lines going to the SRAM chip from the FPGA via the device's I/O pins.

An example application could be the data logger where the data that needs to be stored is acquired for only five minutes every hour. The memory module is operational in this time, but it is in idle state for the rest of the hour. Without clock gating, the module will consume 1.15 watt hours of dynamic energy over a period of thirty days. If clock gating is implemented and the clock to the module is turned off when in idle state, the dynamic power consumption reduces to 0.1026 watt hours of energy for the same 30 day period, which is a 91% reduction in the dynamic power consumption.

6.3.4 Analog module

The main components of this module are the ones responsible for programming the audio codec, while the others are the data formatting components i.e. the serial to parallel and the parallel to serial converters. The power save in this module is achieved by powering off the audio codec when the appropriate disable signal is received. The table below shows the power analysis done for this module. The first simulation done is for the normal operation of the

module. In the second simulation, the disable signal is given, which turns off the clocks to the data formatting components. The table shows that disabling the clock has resulted in very little power saving; only 2.5%. The reason this is so low is the fact that in the analog module the power reduction is achieved on the board level by powering off the audio codec which in turn powers down the data formatting components.

	Core dynamic power(mW)	**Core static power(mW)**	**I/O thermal power(mW)**	**Total thermal power(mW)**
Analog module normal operation	2.71	79.75	40.56	123.02
Analog module device disabled	2.64	79.75	40.1	122.49

Table 6-7: Analog module power analysis result

When the signal activity files were studied, they showed no activity in the data formatting components as expected but the signal activity seen is mainly due to the clocks required to reprogram the codec and bring it out of sleep mode. These clocks are responsible for the dynamic power consumption observed in the power analysis results in the disabled mode of the module. The high I/O power consumption is due to the activity on the I/O lines that are used to communicate between the audio codec and the FPGA.

Chapter 7

Final design operation and analysis

In this chapter, I will describe the working of the thesis design and provide details of the strategy implemented to carry out the power analysis and the difficulties faced in making on-board measurements. The results of the power analysis and on-board measurements made are also discussed.

7.1 Design operation

As described earlier, this design was done to receive and store analog and digital data. The digital data is sent to the module via a RS232 serial connection and the analog data is received and converted to into digital data using an audio codec. These data sets are stored in an external SRAM memory and if the user wishes to retrieve this data (which is indicated via a pushbutton for this design), it is sent out through the digital to analog converter of the audio codec. The DSP module is used to provide filtering of the incoming analog data.

The design's complete operation is summarized below:

- The serial module receives the data via the serial input and converts it into a 16 bit data word to be stored into the external memory. When this module receives the 16 bits of data, it indicates to the memory module that valid serial input data, ready to be written, is available on the data line.
- The analog module programs the audio codec on power up and the codec's A2D converter converts the analog signal to digital which is outputted in serial format. The module's data formatting component converts the serial data into 16 bit data word. This data is sent to the DSP module to be filtered. The module has another data formatting

component that converts the 16 bit parallel data sent from the memory into serial form so that it can be sent out to the user in analog form via the D2A converter of the codec.

- The DSP module takes the data sent from the analog module and filters it. When the filter has finished filtering a sample, it indicates to the memory module that a valid filter output data is available to be written and the output is paced on the data line.
- The memory module receives the data and control signals from the serial and DSP modules and writes the data into their respective banks in the memory. This module generates the appropriate addresses and control signals needed by the external memory to successfully perform a write operation. When the user gives the external read input, the memory module starts at address 0 and serially reads all of the SRAM addresses. The data read is sent to the analog module and from there it is provided as output to the user. After the analog module outputs a 16 bit data word, it indicates to the memory module that it is ready to send another 16 bits. The memory module then sends the data available on the next address location to the analog module until all of the memory locations have been read.

To test the signal acquisition module design, a 1 kHz analog signal was given as input to the design. The serial data given was a sine wave generated in MATLAB. The sine wave of 5 kHz frequency was converted to 2's complement form and sent to the device serially using a PC. The read input was then given to study the memory contents on an oscilloscope.

The images shown here are the output waveforms of the design seen on an oscilloscope:

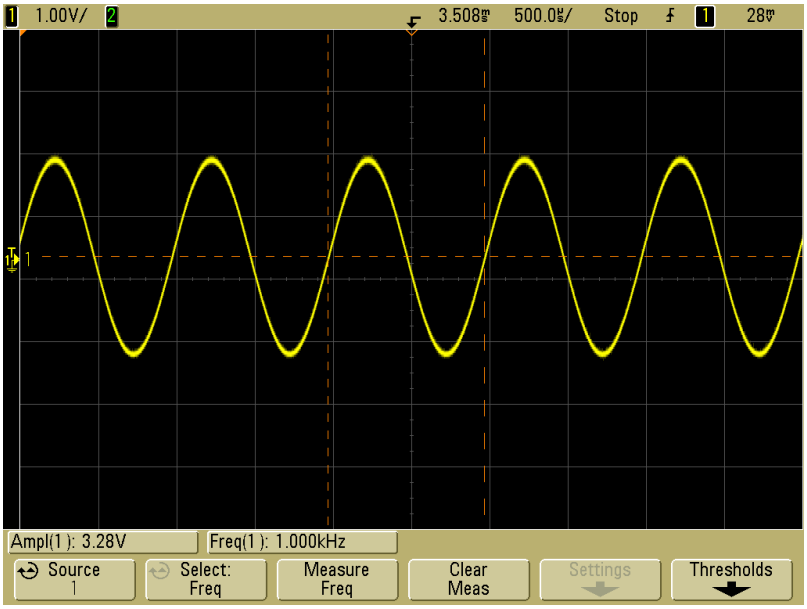

Figure 7-1: Analog output for 1 kHz input frequency

The image in figure 7-1 is the analog output of the device. A sine wave of 1 kHz frequency and 300 mV_{P-P} amplitude was given as input to the design. The audio codec provides amplifies the signal to 3.28 V_{P-P}. The image below is the 5 kHz signal sent from MATLAB.

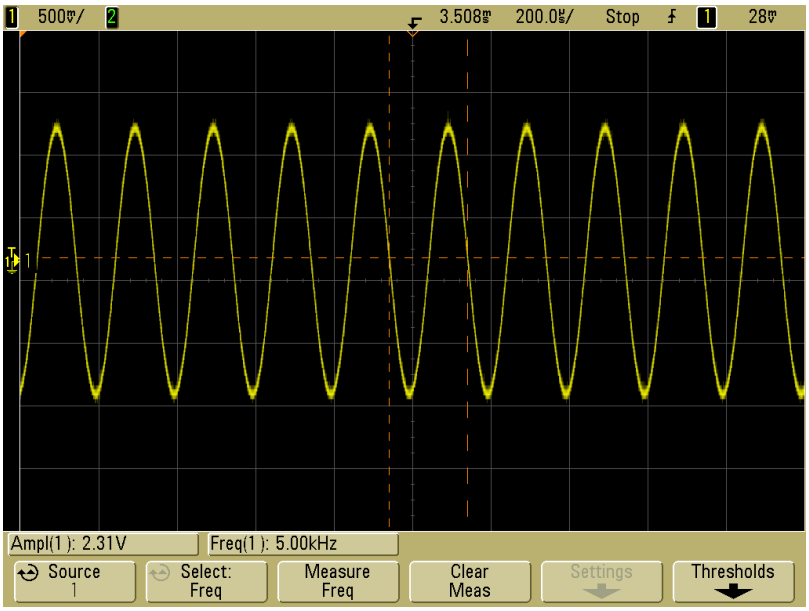

Figure 7-2: 5 kHz sine wave data sent over serial interface

The stop band of the FIR filter implemented in the design begins at 7 kHz. To verify its operation a 13 kHz analog signal was given as analog input and the following output was seen on the oscilloscope. This confirms that the filter is operating as expected.

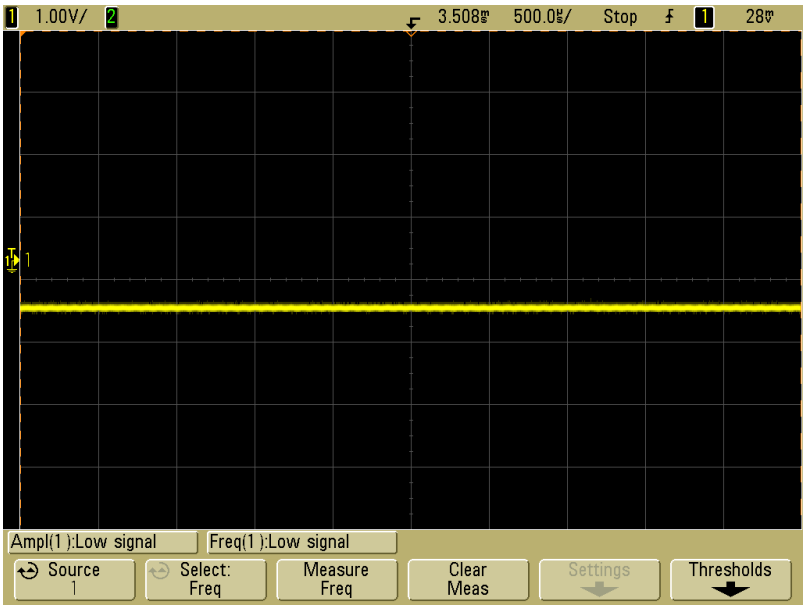

Figure 7-3: Analog output for 13 kHz input frequency

The power conservation logic as described in the previous chapter works as follows:

- IF no new data is seen on the serial input line for 3 seconds, the clock to the serial module is turned off and a LED lights up to indicate this. The module comes back to normal operation if any serial data is seen.
- In the analog module the audio codec is disabled via software if a disable signal is received. The disable signal also turns off the clock to the module components that program the audio codec but the clocks needed for programming when it needs be enabled remain active. In this design, the disable signal is provided via a toggle switch and a LED lights up to indicate that it has been asserted. The module resumes normal operation when the disable signal is removed or if the user asserts the external

signal to output the memory contents via the codec's D2A converter. For wireless applications an example for this disable signal could be a signal that indicates insufficient input signal strength. When the analog module is power down, the DSP module operating on the clock generated by the codec is powered down too.

- If both the serial and analog modules are disabled, the memory module is powered down resulting in the power down of the entire device. If either of the two modules were to resume normal operation, this module does so too. The read input also causes this module to come out of the disable mode.

7.2 Design simulation and power analysis

After the design operation was verified, power analysis was performed. For a more detailed analysis, the design's operation was divided into modes: input mode, output mode, serial shutdown mode, and analog shutdown mode and device power down mode. The design operation in each of these modes is discussed along with the power analysis results.

7.2.1 Input mode

Like the name suggests, the input mode is when both the serial and analog data are being received and written to the external memory. The strategy for the power analysis of this mode was the same as described in earlier chapters; Modelsim-Altera was used to generate the input vectors and the Quartus II simulator was used to simulate the design and generate the signal activity file which is used by the PowerPlay power analyzer to perform the power analysis. The testbench generated all of the required clocks including the ones provided by the audio codec and the input data signals were provided at the correct data rate. A sine wave was generated in verilog and converted into 2's complement form which represented the A2D converter output of the audio codec and an 8 bit counter was implemented representing the serial data which

incremented after each byte was sent on the serial input line. Though this simulation does not represent every possible input combination, it is a sufficient representation of the expected inputs to the design. The testbench used to generate this vector input file is provided in the appendix. The power analysis results of this mode are as follows:

Total Thermal Power Dissipation	131.76 mW
Core Dynamic Thermal Power Dissipation	10.07 mW
Core Static Thermal Power Dissipation	79.77 mW
I/O Thermal Power Dissipation	41.93 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Table 7-1: Design input mode power analysis results

The high dynamic power consumption is as expected and the reason for I/O power being high is all of the activity on the I/O pins carrying data and control signals to the audio codec and the external memory.

7.2.2 Output mode

In this mode, the read input was asserted in the vector input file which would cause the design to read all of the external memory contents and send them out on the digital output line going to the audio codec's D2A converter. The testbench provided all of the required clocks; also all of the data on the serial and analog inputs were still made available. In the output mode, the read operation results in the memory outputting a valid 16 bit data corresponding to the read address generated. So it is required that the data bus reading from the memory be provided with data representing the ones outputted by the external memory. This needs to be done at the proper rate in the vector file so that the output operation may be properly simulated. The testbench used to generate this vector input file is provided in the appendix. The power analysis results of this mode are shown in table 7-2.

Total Thermal Power Dissipation	131.47 mW
Core Dynamic Thermal Power Dissipation	9.88 mW
Core Static Thermal Power Dissipation	79.77 mW
I/O Thermal Power Dissipation	41.83 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Table 7-2: Design output mode power analysis results

Just like in the input mode the high dynamic power consumption is as expected and the reason the I/O power is so high is because of all of the activity on the I/O pins carrying data and control signals etc to the audio codec and the external memory.

The .saf files from the above two modes were used together to generate a power analysis report that represents the input and output as one single benchmark to compare the other modes against. The power analysis results are as follows:

Total Thermal Power Dissipation	131.39 mW
Core Dynamic Thermal Power Dissipation	9.94 mW
Core Static Thermal Power Dissipation	79.77 mW
I/O Thermal Power Dissipation	41.68 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Table 7-3: Design normal operation power analysis results

7.2.3 Serial shutdown mode

In this mode the serial module is shutdown by tying the serial input line high in the input vectors. The data going to the analog module is unaltered and all the clocks required are also active in the vector input file. The clock controller in the serial module will see that the input line is high and turn off the clock to the module. In the design, this process occurs if the serial line is tied to high for three seconds, but for simulation purposes this happens after 50 ns in the vector input file. The testbench used to generate this vector input file is provided in the appendix. The power analysis results of this mode are as follows:

Total Thermal Power Dissipation	128.96 mW
Core Dynamic Thermal Power Dissipation	7.22 mW
Core Static Thermal Power Dissipation	79.76 mW
I/O Thermal Power Dissipation	41.97 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Table 7-4: Device serial module shut down power analysis results

As shown in the above table, the disabling of the serial module has resulted in reduction of signal activity in the design and hence the dynamic power consumption is lower than the normal operation mode. Compared to the design's normal operation, we see a 27.36% reduction in the dynamic power consumption.

7.2.4 Analog shutdown mode

In this mode the input signal to the analog module that instructs the module to power down the audio codec is asserted in the input vectors. In real time, this causes the clock outputs of the audio codec to be turned off. To simulate this in the testbench, the clocks that represent the audio codec clock outputs are tied low. As seen in the previous chapter, this does not affect the analog module very significantly. The data input given to the serial input is not altered and the data line representing the audio codec's A2D output is tied low. The testbench used to generate this vector input file is provided in the appendix. The power analysis results of this mode are as follows:

Total Thermal Power Dissipation	131.33 mW
Core Dynamic Thermal Power Dissipation	9.83mW
Core Static Thermal Power Dissipation	79.77 mW
I/O Thermal Power Dissipation	41.73 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Table 7-5: Design codec shutdown mode power analysis results

As seen in the table of values, this mode did not affect the total power consumption at all. This is rather interesting because in the previous section we see that the filter module shows significant dynamic power reduction when clock gating is implemented. The signal activity files

were studied and the signal activity of the signals in these modules was zero as expected and still had no effect on the dynamic power consumption. This is because the power reduction is achieved in the previous section by reducing signal activity. But in this design I have clock signals active in the module to enable the module when needed. So the power consumption reduction achieved on the logic interconnects is nullified by the higher consumption on the clock lines as they have higher switching capacitances [**17**]. Also, in the previous section it was shown that the clock gating in the serial module has a higher power saving than the analog module which can is seen here as well.

7.2.5 Device power down mode

This mode of operation is when both the analog and serial modules are powered down which results in the memory module to be powered down as well. To simulate this mode, the serial input line is tied high to turn off the module as in the serial shutdown mode and like in the codec shutdown mode the clocks are tied to low along with the codec A2D converter output. With these two modules turned off, the clock distribution modules turns off the clock to the DSP and memory modules which results in the entire device being power down. The dynamic power consumption of this mode can be expected to be the lowest one of all of the modes. The testbench used to generate this vector input file is provided in the appendix. The power analysis results of this mode are as follows:

Total Thermal Power Dissipation	124.58 mW
Core Dynamic Thermal Power Dissipation	3.10 mW
Core Static Thermal Power Dissipation	79.76 mW
I/O Thermal Power Dissipation	41.73 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Table 7-6: Design power down mode power analysis results

From the table it is clear that the device powering down results in significant dynamic power consumption reduction. The percentage dynamic power saving is 68.8% compared to the normal operation mode; compared to the serial shutdown mode the dynamic power saving observed is 41.4% and compared to the analog shutdown mode the dynamic power saving is 67.7%.

Now that we have characterized the power saving achieved in each of these modes, let us now see how they affect the total power consumption of an example application. Let us consider a data logging application where the modules operate for the following percentage of time during the day

1. Serial module operates independently for 35%
2. Analog module operates independently for 20%
3. Both the modules operate together for 15%
4. Device is powered off for the remaining 30%

Based on the estimated power consumption values, over a period of 30 days

1. With no power down modes the devices consumes 7.25 watt hours of dynamic energy
2. With the power down modes implemented, this value reduces to 4.99 watt hours of dynamic energy. This is a 31% reduction in consumption

7.3 Real time board measurement challenges

The test board design was done to facilitate detailed power consumption measurements on the design. After the design process was completed, the board design files were sent to manufacturing companies to be fabricated and assembled with the required components. The motivation in designing the board was to be able to study each of the device supply voltages independently. The board design provided the user with two options for supplying the required

voltages to the board components. One of the options was the conventional power supply option where the voltage from the power grid is regulated and provided to all of the components on the board. The other option was that the user could provide the required voltages using laboratory voltage generators. This would allow to test the effects of voltage scaling and voltage control on the device's total power consumption. Current sensing resistors in the voltage supply paths were added to allow for the independent measurement of the current drawn from each of the voltage sources. These design features would provide us with an opportunity to study and characterize the power consumption of this design and possibly many others in detail.

This however could be put into practice due to high manufacturing cost of the printed circuit board alone. The PCB fabrication was quoted to cost in the range of $8,000 to $15,000 by the manufacturing companies. The reason for this PCB price was the routing method used for the BGA. As explained earlier, the 674 pin BGA is a very fine line BGA and routing its pins out to the other components on the on only the top and bottom layers of the board could not be achieved. The layer stack as described in chapter 5 was implemented to perform the routing. The manufacturing companies informed that the fabrication of blind vias designed for this purpose were the factor that resulted in such a high manufacturing cost. To create a blind via during manufacturing, the etched layers were laminated and then the holes were drilled in the proper locations. And in this design the buried vias go from the top layer to all of the inner layers, which meant that multiple lamination and drilling steps would be performed. This and the fact that two sets of those buried vias started from the bottom layer into an inner layer increased the manufacturing complexity causing the high board manufacturing cost.

Other options had to be researched in order to perform real time measurements. The Altera DE2 development board for the Cyclone II FPGA by Terasic was used in the design process to

test and debug the design operation. The board's schematic showed that the regulated power supply was distributed to different power nets on the board through zero ohm resistors. It was decided that these resistors would be a good location for current measurements in the current path of the device. The figure below shows a section of the power supply schematic page of the DE2 board [**27**], where we can see the power distribution nodes of the board. These nodes are used to provide the different nets with the required 5V, 3.3V, 1.8V and 1.2V regulated supply voltages. The inductor BEAD as described in earlier chapters is used as a choke preventing any noise from the analog circuitry to enter the digital part.

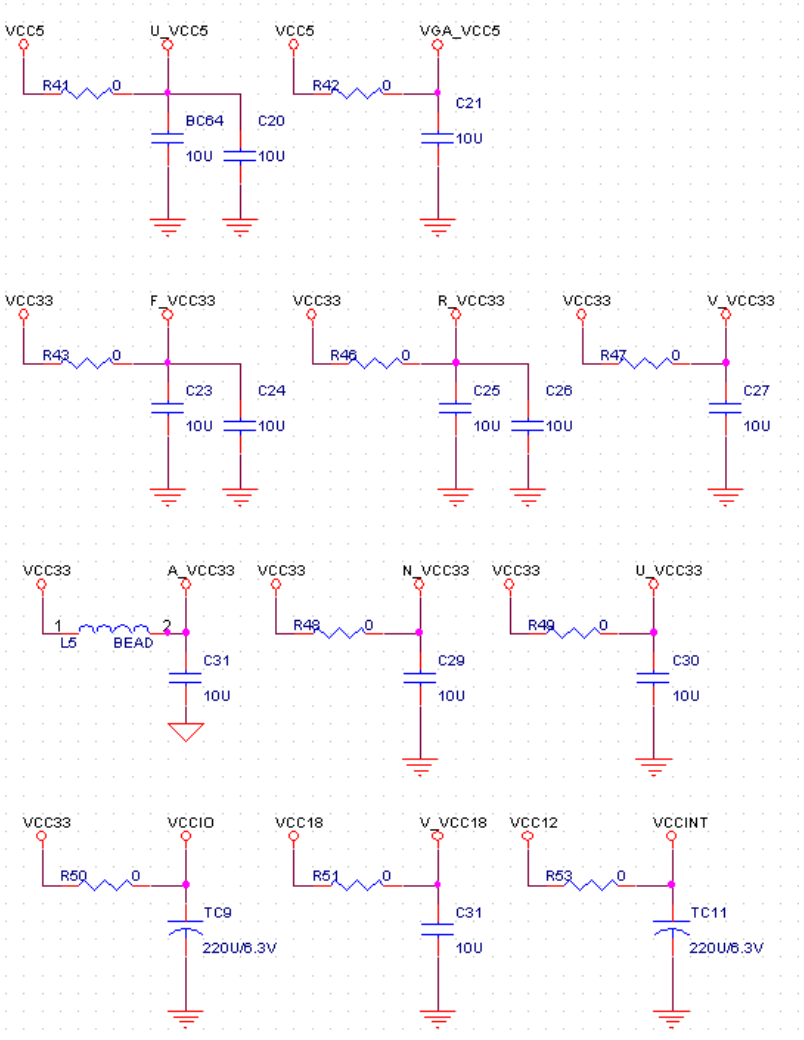

Figure 7-4: Power supply section of the DE2 board schematic [27**]**

For making real time FPGA device power measurements it was decided to measure the currents at R50 and R52 which are the zero ohm resistors carrying the 3.3 V VCCIO voltage and 1.2 V VCCINT voltage respectively to the FPGA device. To make the current measurements convenient, these resistors were desoldered and brought out to header pins using wires soldered to the resistor pads. The header pins were mounted to the side of the board to allow for making current measurements during the device operation as shown in the image below.

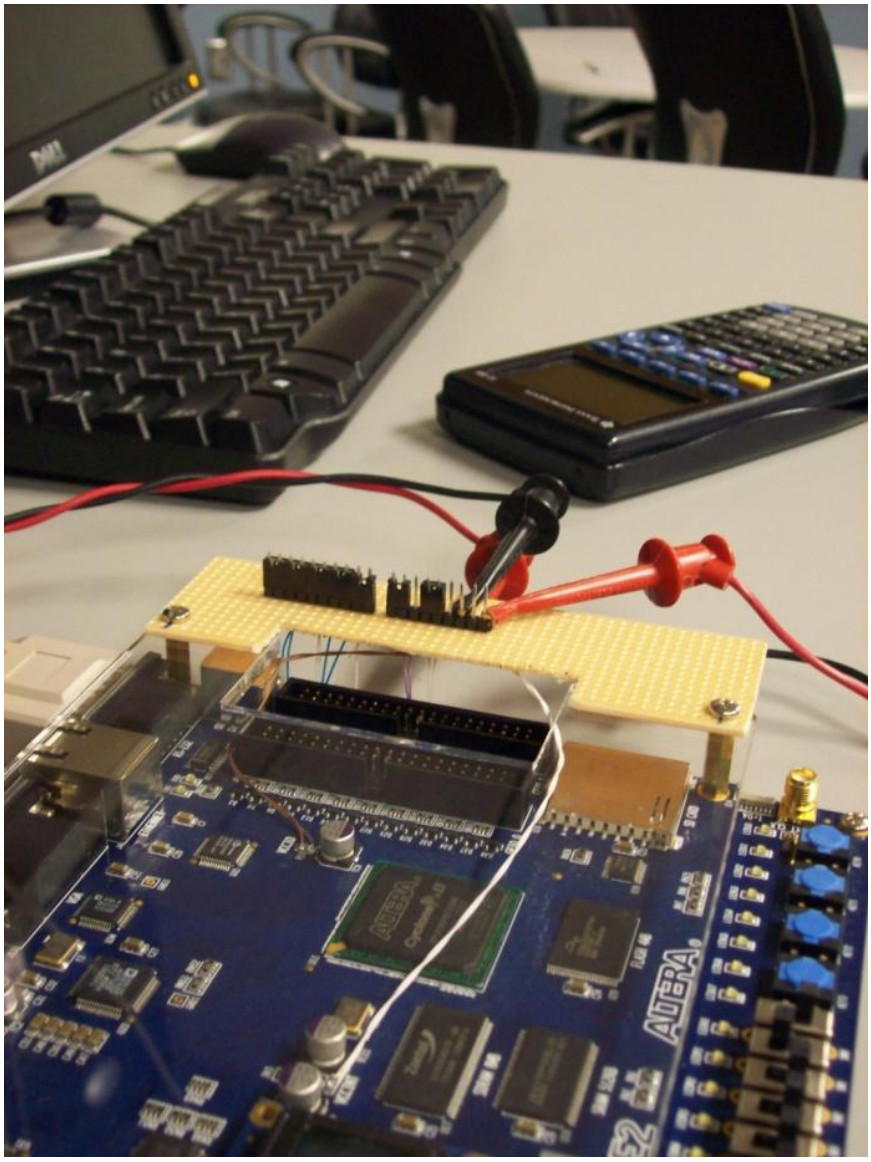

Figure 7-5: Modified DE2 board to allow FPGA current measurements

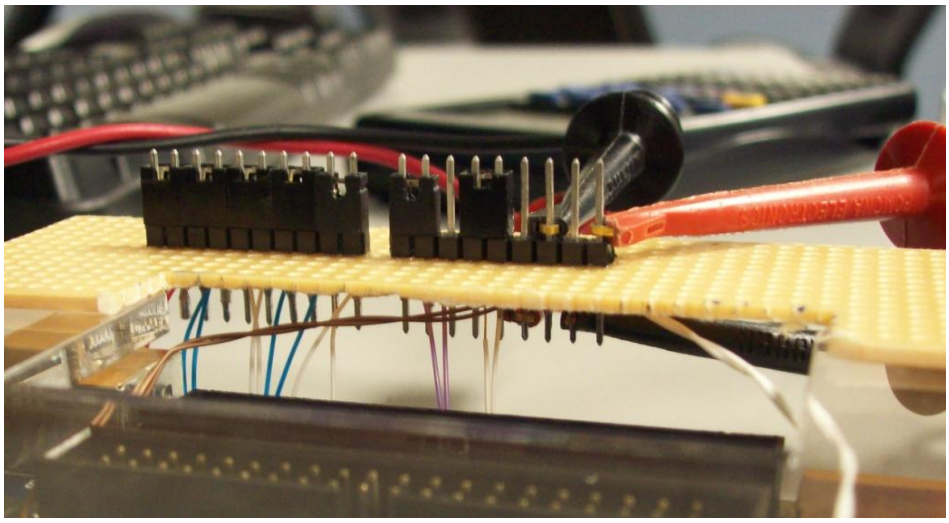

Figure 7-6: Ammeter probes connected to the header pins for current measurements

The current drawn by the FPGA on the VCCINT and VCCIO power supplies can now be measured with ammeters connected to the header pins. The measurement and test setup for this design are discussed in the next section.

7.4 Measurement and Test setup

7.4.1 Establishing a baseline

Before performing the current measurements, a simple design was implemented on the FPGA and the design power analysis was done. The on-board measurements of the currents drawn by the FPGA from the VCCIO and VCCINT will be made and observe if these values corroborate with values obtained by simulation.

The baseline design was a simple 30 bit counter which increments on every positive edge of the 50 MHz master clock. The counter output of the module and is connected to I/O pins that bring the signals out onto header pins on the test board. The power analysis results for this design are provided here.

Total Thermal Power Dissipation	131.89 mW
Core Dynamic Thermal Power Dissipation	1.75 mW
Core Static Thermal Power Dissipation	79.77 mW
I/O Thermal Power Dissipation	50.37 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Table 7-7: Base line test power analysis result

The power analysis report file contains more information than just the summary of the power analysis results. It provides information like:

- Thermal Power Dissipation by Block Type
- Thermal Power Dissipation by Hierarchy
- Current Drawn from Voltage Supplies Summary
- VCCIO Supply Current Drawn by Voltage

Among the available values, the one metric that can be compared to the on-board measurements is the "VCCIO Supply Current Drawn by Voltage". The estimated values for this metric are shown in table 7-8.

Current Drawn from Voltage Supplies Summary				
Voltage Supply	Total Current Drawn	Dynamic Current Drawn	Static Current Drawn	Minimum Power Supply Current
VCCINT	70.04 mA	1.59 mA	68.45 mA	70.04 mA
VCCIO	14.48 mA	5.06 mA	9.42 mA	14.48 mA

Table 7-8: Power analysis report on the current drawn from voltage supplies

The baseline counter design is programmed onto the FPGA and the measurements of the currents drawn from VCCIO and VCCINT are taken. The measurements were taken as follows;

- The asynchronous reset button was pressed to disable the clock. The current drawn values can be approximated as the base static current drawn values.

- After the reset is released the maximum value of the current drawn is the maximum mode current consumption value.

The measurements obtained by performing these steps are shown in Table 7-9.

	I_{VCCINT}	I_{VCCIO}
Base static current	22.87	9.82
Maximum mode current	23.05	15.58

Table 7-9: Baseline design real time current measurements

When the FPGA finished programming the current drawn from the VCCINT supply was measured as 23 mA, which is different than the estimated 70 mA by a pretty large margin. While the estimates of static current and dynamic current drawn from VCCIO and VCCINT appear to be good metrics to compare, their very high values appear to be worst case current drawn estimates that the device power supply circuitry can to be designed to.

The table shows that the increase in dynamic power in the two cases is 5.76 mA, this is close to the 5.06 mA estimated dynamic current drawn from the FPGA. So from the study of this design we can say that while an exact comparison cannot be made for the current consumption values, the values for I_{VCCIO} agree closely in the two cases. For the I_{VCCINT} measurements one observation that can be made is if an increase or decrease in the reported value corresponded to an increase or decrease of the measured values.

7.4.2 Design test setup and measurements

As discussed in a previous chapter, the digital data input to the device was given via a RS232 serial interface link and the analog input was generated using an arbitrary waveform generator. The memory contents outputted through the D2A converter of the audio codec is seen on an oscilloscope and for the current measurements, the header pins are connected to digital multimeters. Figure 7-7 shows the board setup for testing the operation of the design and making

real time current measurements. The USB blaster is used to configure the FPGA device with the bitstream generated by the design software. Analog input coming from the waveform generator and analog output going to the oscilloscope is shown. The serial interface provides the digital data and the header pins allow the user to make current measurements. The cyclone II FPGA is shown along with two pushbutton inputs to the design. Key1 is the asynchronous reset while Key2 is the external input which when asserted outputs the memory contents through the audio codec's analog D2A converter output. SW1 is the toggle switch used as the disable signal for the audio codec. Two LEDs (not shown in the figure) are used to indicate when the design goes into serial shutdown and analog shutdown modes.

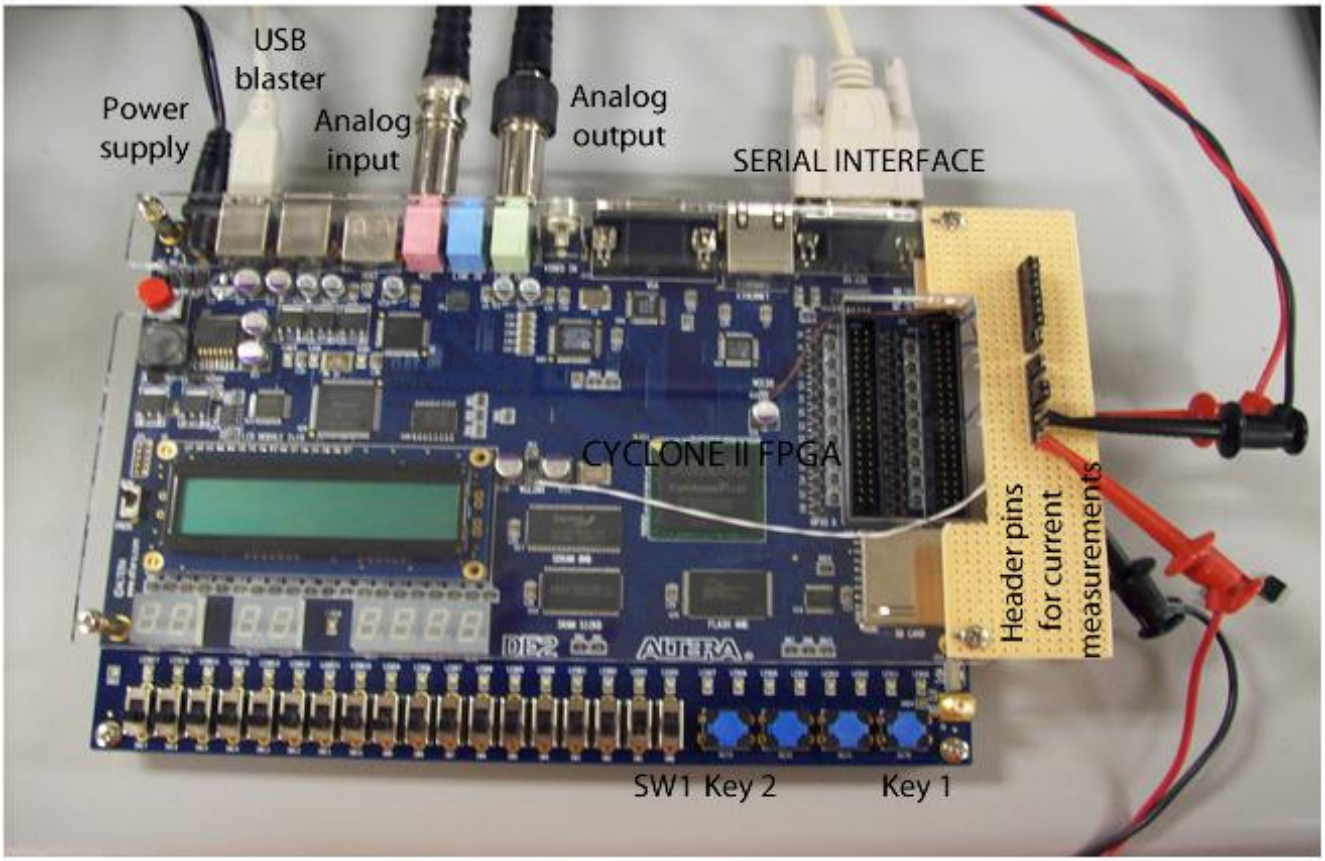

Figure 7-7: Altera DE2 board setup

The procedure followed in making the measurements after programming the FPGA is provided below:

- The asynchronous reset button is pressed to take the base static current measurements without any inputs given.
- The reset is released and both the analog and digital inputs are given to the board. The mode current values are measured. This mode is the input mode.
- The read input is given with the inputs active and the mode current values for this mode are measured. This mode is the output mode.
- The audio codec is disabled by asserting the disable toggle input of the design. The mode current values in this case are measured. This mode is the analog shutdown mode.
- The serial is also disabled and the current values are measured. This is the device power down mode.
- Keeping the serial disabled, the codec is enabled by de-asserting the disable toggle input and the current measurements are made. This is the serial shutdown mode.

The table below summarizes the reported values of dynamic current for the different FPGA device operating modes. The static current drawn is the first row of the table while the other rows show the values for dynamic current estimations for the particular mode.

Operating mode	Reported values	
	I_{VCCINT} (mA)	I_{VCCIO} (mA)
Estimated static current	70.26	9.71
Input mode	1.59	5.06
Output mode	8.3	1.54
Serial shutdown	6.19	1.52
Analog shutdown	8.36	1.44
Device power down	2.74	1.44

Table 7-10: Reported voltage supply current drawn values

The table below shows the measured values of the supply current for the different modes.

Operating mode	Measured values	
	I_{VCCINT} (mA)	I_{VCCIO} (mA)
Base static current	25.15	9.544
Input mode	26.37	11.4
Output mode	26.28	11.56
Serial shutdown	26.05	14.75
Analog shutdown	26.47	14.77
Device power down	25.07	17.84

Table 7-11: Measured voltage supply current drawn values

The table below shows the increase in the I_{VCCIO} supply current drawn from the base static current.

Operating mode	I_{VCCIO} dynamic current drawn (mA)
Input mode	1.856
Output mode	2.016
Serial shutdown	5.206
Analog shutdown	5.226
Device power down	8.296

Table 7-12: Change in current value from base static current for each mode

Operating mode	Reported I_{VCCIO} total current drawn	Measured I_{VCCIO} total current drawn
Input mode	14.77	11.4
Output mode	11.25	11.56
Serial shutdown	11.23	14.75
Codec shutdown	11.15	14.77
Device power down	11.15	17.84

Table 7-13: Comparing the measure and reported VCCIO supply total current drawn for each mode

The first thing you will notice is while in Table 7-10 the dynamic power reported in the device power down mode is 1.44 mA but in table 7-12 the measured current has increased by 8.296 mA when compared to the base static current. This increase is because, in this mode the FPGA device has to illuminate two LEDs, hence drawing a higher current. This is also the reason for serial shutdown mode and analog shutdown mode having higher current increase. From the measured values we observe that this causes an addition of roughly 3 mA increase in current consumption per LED illuminated. When this value is factored out in table 7-13 we see that the I_{VCCIO} values in the two sets agree closely for most cases.

Apart from this, while the reported values show a decrease in the dynamic consumption of I_{VCCIO} form input mode to output mode, the measured values indicate a higher dynamic consumption in the output mode when compared to the input mode. The two set of values agree when comparing the fact that in both the cases serial shutdown consumes less dynamic current compared to analog shutdown.

7.4.3 Analyzing results

Comparison of the power consumption estimates and the values measured on the board was done as a means to validate the simulation results for the low power design approach implemented. Due to the board limitations, it was not possible to indentify each of the device's power consumption components in the measured value. By comparing the available data, it is

observed that the FPGA draws a third of the current specified in the simulation and still operates correctly. Hence, it can be safely assumed that the I_{VCCIO} value reported is a worst case scenario calculation for the device current.

The I_{VCCINT} current drawn data measured on the board and reported by the analyzer are found to be comparable to each other for a majority of the modes. The fact that for the measured values for I_{VCCINT}, serial shutdown mode draws less current when compared to analog shutdown mode. This is also the case in the simulation results. So, we can safely state that the simulation results can be used to obtain a first order estimation of the designs power consumption and study the efficacy of the design approach used. The measured results have shown that the power saving observed in the analysis results do appear in the on-board device operation.

Chapter 8

Conclusions

FPGAs are fast becoming a popular technology platform for many system designers because of the flexibility and robustness they provide. For low power implementations, ASICs and microprocessors have always been preferred in the past over FPGAs. This is due to their architectural nature of drawing substantial amounts of static power. Continued research in improving the device architecture, device process techniques of the FPGA resulting in low leakage devices and device architecture allowing for power efficient routing have reduced the technology gap that previously existed. This in conjunction with design techniques aimed at power reduction and the availability of tools to help optimize the designs have allowed the use of FPGAs in applications for which ASICs were preferred.

For this thesis, results of preliminary design power analysis done on various FPGA devices are used to select the appropriate one. The EP2C35F672 device of Altera's Cyclone II family is chosen because its power consumption was low as desired, which was achieved without having to compromise on device resource availability for future improvements.

The system design is successfully completed and its operation verified. The design is divided into functional modules and methods to improve individual module power efficiency are implemented in the designing process. Techniques like clock gating and mapping logic into embedded structures on the FPGA are utilized to optimize the design's power and logic area consumption. Mapping the MAC operations of the DSP module into DSP blocks resulted in a 30% reduction of the device logic area consumed and 49.6% reduction in the register consumption. Clock gating was effective in reducing 82.3%, 88.75% and 99.4% of dynamic power consumption for the serial, DSP and memory modules respectively. The power reduction

achieved on the system level with these optimized modules is analyzed. The device operation is divided into modes and power consumption in each mode is estimated. The serial shutdown mode showed a 27.36% decrease in the dynamic power consumption compared to normal operation. The analog shutdown mode did not show any improvement because in this mode the signal activities on the logic interconnects are reduced using clock gating, but the clocks, on the higher capacitance global interconnects, used to enable these modules when required are the predominant cause of the dynamic power consumption and nullify the gain achieved by clock gating.

The layout for the test board is done but it could not be fabricated due to high manufacturing cost. The DE2 development board was modified to allow for real time measurements, but the limited knowledge of the board's design and limitations on how detailed these measurements are did not allow for a complete analysis of these measured values. The measurements on the baseline design showed that the I_{VCCIO} measurements made were comparable to the estimated values, while for the I_{VCCINT} measurements the observation made was weather a variation in the measured value corresponded to a similar variation of the measured value. The comparison of the values showed that they agree with each other for a majority of the operating modes and with the available data it can be stated that the simulation results can be used to justify the design methodology implemented and the low power implementation is achieved for this design.

8.1 Future work

The lack of a detailed analysis of the current measurements needs to be addressed. Improvements on the test board layout to allow for more fine grained measurements than designed for initially can be done and if its fabrication is feasible, this limitation can be overcome. If the board fabrication is not feasible, the current test board needs to be studied more

closely to gain a better understanding of the measured values. Different designs can be implemented to help gain this understanding. Details of the algorithm implemented to perform the power analysis can be used to help correlate the measure values with the estimated ones. With this detailed data the low power design approaches used can be further optimized, and the effect of other approaches like dynamic clock scaling and voltage scaling need to be studied for this design.

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*. Newnes, 2004.

[2] K. Parnell and R. Bryner, "Comparing and Contrasting FPGA and Microprocessor System Design and Development," Xilinx, Inc White Paper, 2004.

[3] B. Zeidman. (2006, Mar.) EETimes.com. [Online]. http://www.eetimes.com/design/programmable-logic/4014815/All-about-FPGAs

[4] Altrea Corporation, "Cyclone II Device Handbook," Altrea Corporation Device handbook, 2008.

[5] Altera Corporation, "Introduction to the Quartus II Software," Altera Corporation User Manual, 2009.

[6] J. Lamoureux and W. Luk, "An Overview of Low-Power Techniques for Field-Programmable Gate Arrays," in *Adaptive Hardware and Systems, 2008. AHS '08. NASA/ESA Conference on*, 2010, pp. 338-345.

[7] P. Abusaidi, M. Klein, and B. Philofsky, "Virtex-5 FPGA System Power Design Considerations," Xilinx, Inc White Paper, 2008.

[8] T. Tuan, A. Rahman, S. Das, S. Trimberger, and S. Kao, "A 90-nm Low-Power FPGA for Battery-Powered Applications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 296-300, Feb. 2007.

[9] S. M. S. kang, "Elements of low power design for integrated systems," in *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, 2003, pp. 205-210.

[10] H.-Y. Wong, L. Cheng, Y. Lin, and L. He, "FPGA device and architecture evaluation considering process variations," in *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, 2005, pp. 19-24.

[11] E. Kusse and J. Rabaey, "Low-energy embedded FPGA structures," in *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, 1998, pp. 155-160.

[12] C. T. Chow, L. S. M. Tsui, P. H. W. Leong, W. Luk, and S. J. E. Wilton, "Dynamic voltage scaling for commercial FPGAs," in *Field-Programmable Technology, 2005. Proceedings.*

2005 IEEE International Conference on, 2005, pp. 173-180.

[13] S. J. Nam, et al., "A low power MPEG I/II layer 3 audio decoder," in *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, 2001, pp. 353-356vol2.

[14] P. P. Czapski, "Power Optimization Techniques in FPGA Devices: A Combination of System-And Low-Levels," *International Journal of Electrical, Computer, and Systems Engineering*, vol. 1, no. 3, p. 148, 2007.

[15] V. George, H. Zhang, and J. Rabaey, "The design of a low energy FPGA," in *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, 1999, pp. 188-193.

[16] Y. Zhang, J. Roivainen, and A. Mammela, "Clock-Gating in FPGAs: A Novel and Comparative Evaluation," in *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, 2006, pp. `584-590.

[17] I. Brynjolfson and Z. Zilic, "Dynamic clock management for low power applications in FPGAs," in *Custom Integrated Circuits Conference, 2000. CICC. Proceedings of the IEEE 2000*, 2000, pp. 139-142.

[18] Altera, Inc, "Clock Control Block (ALTCLKCTRL)," Altera, Inc Megafunction User Guide, 2008.

[19] G. Sutter, E. Todorovich, and E. Boemo, "Design of Power Aware FPGA-based Systems".

[20] S. J. E. Wilton, S.-s. Ang, and W. Luk, "The impact of pipelining on energy per operation in field-programmable gate arrays," *IN FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS. PROCEEDINGS OF THE 13TH INTERNATIONAL WORKSHOP, FPL 2004, LECTURE NOTES IN COMPUTER SCIENCE, LNCS 3203*, pp. 719--728, 2004.

[21] J. Lamoureux, G. Lemieux, and S. Wilton, "GlitchLess: Dynamic Power Minimization in FPGAs Through Edge Alignment and Glitch Filtering," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 11, pp. 1521-1534, 2008.

[22] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 203-215, 2007.

[23] H. Belhadj, V. Aggrawal, A. Pradhan, and A. Zerrouki, "Power-Aware FPGA Design," Actel Corporation White Paper, 2009.

[24] J. P. Brennan, A. Dean, S. Kenyon, and S. Ventrone, "Low power methodology and design techniques for processor design," in *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, 1998, pp. 268-273.

[25] Altera Corporation, "Quartus II Handbook," Altera Corporation User manual, 2009.

[26] Wolfson Microelectronics, "Portable Internet Audio CODEC with headphone Driver and Programmable Sample Rates," Wolfson Microelectronics Datasheet, 2004.

[27] Terasic Technologies, "TERASIC CYCLONE II EP2C35 Development & Education BOARD," Terasic Technologies Schematic, 2006.

Appendix A - Design top level HDL code

```
//Signal acquisition module Top level HDL design
//Author: Ravi Thakur

module sig_acqmod(
input clk_50,
input ar,
//codec signals
inout SDAT,
output SCLK,
output AUD_DACDAT,
input AUD_ADCDAT,
output AUD_XCK,
input AUD_BCLK,
input AUD_DACLRCK,
input AUD_ADCLRCK,
//codec disable input
input codec_pwrdwn,
//read input from the user
input read,
//serial port
input sl_in,
output sl_sleep,
//codec_ctrl
output codec_disable,
//sram
inout [15:0] sram_dq,
output [17:0] sram_a,
output sram_oe,
output sram_ce,
output sram_we,
output sram_ub,
output sram_lb
);

//analog module
wire clk_amod;//50MHz clk goin to data_setup
wire clk_datarate;//48.8 kHz clock going to serial clk ctrl
wire [15:0] filter_in;//going to the dsp module
wire rd_begin;//read signals to the memory module
wire read_capture;

//DSP module
wire clk_DSP;//50 MHZ clock to set write signals to the memory module
wire [15:0] filter_out;//going to the memory module
wire filter_wr;

//memory module
wire clk_mmod;
wire sram_empty;//indicating the analog module to stop sending read signals
wire [15:0] sram_out;//data to be sent to the analog module

//serial module
wire [15:0] serial_data;//data sent to the memory module
```

```
wire serial_wr;//write signal sent to the memory module

//device control
wire device_disable;//when both codec and serial are off the this signal turns off the device
wire codec_reset;//brings the device out of sleep when read input is given

//instantiations

analog_module   mod1 (ar,clk_50,clk_amod,clk_datarate,read,sram_empty,codec_pwrdwn,SDAT,SCLK,
                      AUD_ADCDAT,AUD_DACDAT,AUD_XCK,AUD_BCLK,AUD_ADCLRCK,
                      AUD_DACLRCK,filter_in,ram_out,rd_begin,codec_reset,codec_disable,read_capture);

DSP_module      mod2 (AUD_ADCLRCK,clk_DSP,ar,filter_in,read_capture,filter_out,filter_wr);

serial_module   mod3 (ar,clk_50,clk_datarate,sl_in,serial_data,serial_wr,sl_sleep);

memory_module   mod4 (clk_mmod,ar,filter_wr,filter_out,serial_wr,serial_data,rd_begin,sram_empty, sram_out,
                       sram_dq,sram_a,sram_oe,sram_ce,sram_we,sram_ub,sram_lb);

device_pwrdwn   inst0 (clk_50,ar,codec_disable,sl_sleep,device_disable);

clk_distributor inst1 (clk_50,ar,codec_reset,device_disable,clk_amod,clk_DSP,clk_mmod);

endmodule
```

Appendix B - Analog module HDL code

Analog module top level:

```
//Analog module Top level design
//Author: Ravi Thakur

module analog_module(
input ar,
input clk_50,
input clk_codec,
output clk_datarate,
input read,
input sram_empty,
input codec_pwrdwn,
inout SDAT,
output SCLK,
input AUD_ADCDAT,
output AUD_DACDAT,
output AUD_XCK,
input AUD_BCLK,
input AUD_ADCLRCK,
input AUD_DACLRCK,
output [15:0] filter_in,
input [15:0] sram_out,
output rd_begin,
output codec_reset,
output codec_disable,
output read_capture
);

wire [23:0] data_codec;
wire done,activate;
wire reset;
wire read_edge;

assign codec_reset = ~(~ar|reset);

data_setup inst2 (clk_codec,data_codec,done,codec_reset,activate,codec_disable);

codec_prgm inst3 (clk_datarate,codec_reset,SCLK,SDAT,data_codec,activate,done);

serial2parallel inst4 (ar,AUD_ADCDAT,AUD_BCLK,AUD_ADCLRCK,filter_in);

parallel2serial inst6 (ar,sram_out,AUD_BCLK,AUD_DACLRCK,clk_50,read,sram_empty,AUD_DACDAT,
                       read_edge,rd_begin,read_capture);

codec_ctrl inst10 (clk_50,ar,codec_pwrdwn,read_edge,reset,codec_disable,clk_datarate,AUD_XCK);

endmodule
```

data_setup HDL code:

```
//Design component to set up the control words for the audio codec
//Author: Ravi Thakur

module data_setup (
clk_50,
data_codec,
done,
ar,
activate,
codec_disable
);
input  clk_50;
input  done;
input  ar;
output [23:0]data_codec;
output activate;
input codec_disable;

reg  [15:0]ctrl_word[6:0];//data to write to the codec registers
reg  [5:0]address;//address of the registers

wire [23:0]data_codec={8'h34,ctrl_word[address]};

wire  activate =((address < 6'd7) && (done==1'b1))? 1'b0:1'b1;

always @(negedge ar or posedge done) begin
        if (~ar)
                address=0;
        else
                if (address < 6'd7)
                        address=address+6'd1;
end

always @ (posedge clk_50)
begin
        ctrl_word[0]= 16'h0e42;        //master mode, bit clk not inverted, I2S, 16 bit
        ctrl_word[1]= 16'h0814;        //sound select: mic input given to adc.
        ctrl_word[2]= 16'h1000;                //mclk
        ctrl_word[3]= 16'h0080;                //LLine in muted
        ctrl_word[4]= 16'h0280;                //RLine in muted
        ctrl_word[5]= 16'h1201;                //active
   ctrl_word[6]= codec_disable? 16'h0ce0:16'h0c01;//power down if true
end

endmodule
```

codec_prgm HDL code:

```
//Design component to program the audio codec
//Author: Ravi Thakur

module codec_prgm (
        clk_br,
        ar,
        SCLK,//I2C CLOCK
        SDAT,//I2C DATA
        data_codec,
        activate,
        done
);
        input  clk_br,ar;
        input  [23:0]data_codec;
        input  activate;
        inout  SDAT;
        output SCLK;
        output done;

reg ack_enable;
reg bit_sent;
reg sclk_enable;
reg done;
reg [23:0]data_sent;
reg [5:0]counter;

wire SCLK=sclk_enable | ( ((counter >= 4) & (counter <=30))? ~clk_br :1'b0 );
wire SDAT=ack_enable?1'bz:bit_sent ;

always @(posedge clk_br or negedge ar )
 begin
        if(~ar)
                counter=0;
        else
                if (activate==0)
                        counter=0;
                else
                        if (counter < 6'd33)
                         counter=counter+6'd1;
end

always @(posedge clk_br )
 begin
case (counter)
        6'd0  : begin  done=0;bit_sent=1; sclk_enable=1;ack_enable=1'b0;end
        //start
        6'd1  : begin data_sent=data_codec;bit_sent=0;ack_enable=1'b0;end
        6'd2  : sclk_enable=0;
        //SLAVE ADDR
        6'd3  : bit_sent=data_sent[23];
```

```
            6'd4  : bit_sent=data_sent[22];
            6'd5  : bit_sent=data_sent[21];
            6'd6  : bit_sent=data_sent[20];
            6'd7  : bit_sent=data_sent[19];
            6'd8  : bit_sent=data_sent[18];
            6'd9  : bit_sent=data_sent[17];
            6'd10 : bit_sent=data_sent[16];
            6'd11 : ack_enable=1'b1;//ACK

            //SUB ADDR
            6'd12  : begin bit_sent=data_sent[15];ack_enable=1'b0; end
            6'd13  : bit_sent=data_sent[14];
            6'd14  : bit_sent=data_sent[13];
            6'd15  : bit_sent=data_sent[12];
            6'd16  : bit_sent=data_sent[11];
            6'd17  : bit_sent=data_sent[10];
            6'd18  : bit_sent=data_sent[9];
            6'd19  : bit_sent=data_sent[8];
            6'd20  : ack_enable=1'b1;//ACK

            //DATA
            6'd21  : begin bit_sent=data_sent[7];ack_enable=1'b0;end
            6'd22  : bit_sent=data_sent[6];
            6'd23  : bit_sent=data_sent[5];
            6'd24  : bit_sent=data_sent[4];
            6'd25  : bit_sent=data_sent[3];
            6'd26  : bit_sent=data_sent[2];
            6'd27  : bit_sent=data_sent[1];
            6'd28  : bit_sent=data_sent[0];
            6'd29  : ack_enable=1'b1;//ACK

            //stop
    6'd30 : begin bit_sent=1'b0;sclk_enable=1'b0; ack_enable=1'b0; end
    6'd31 : sclk_enable=1'b1;
    6'd32 : begin bit_sent=1'b1; done=1; end
endcase
end
endmodule
```

serial2parallel converter HDL code:

```
//serial-to-parallel converter design
//Author: Ravi Thakur

module serial2parallel(
input ar,
input adc_out,
input bclk,
input adclrck,
output reg [15:0] parallel_out
);

reg lrck_prev;//stores the prev value
reg lrck_capture;
reg done;//to reset the capture signal to default 0 value
```

```
parameter N=5'd15;

//edge capture of adclrck
always @ (negedge bclk or negedge ar)
begin
	if(~ar)
	lrck_prev = 1'b0;//ADCLRCK is default 0 signal
	else
	lrck_prev = adclrck;
end

wire lrck_edge = adclrck & ~lrck_prev;//captures both the neg and pos edge using xor operation

always @ (negedge bclk or negedge ar)
begin
	if(~ar)
		lrck_capture = 1'b0;
	else
		if(lrck_edge)
			lrck_capture = 1'b1;//active high signal
		else
			if(done)
				lrck_capture = 1'b0;
end

//serial to parallel conversion

reg [4:0]counter;

always @ (negedge bclk or negedge ar)
begin
	if(~ar)
		begin
			counter = 5'd0;
			done = 1'd0;
		end
	else
		begin
			if(lrck_capture==1'b1 && done==1'b0)
				begin
					if(counter<=5'd15)
						begin
							parallel_out[N-counter]=adc_out;
							counter = counter+5'd1;

						  if(counter==5'd16)
						    begin
							done=1'd1;

						    end
						end
				end
			else
				begin
					counter=5'd0;
					done=1'd0;
```

```
                                        end
                        end//end else
end//end always

endmodule
```

parallel2serial converter HDL code

```
//parallel-to-serial converter design
//Author: Ravi Thakur

module parallel2serial(
input ar,
input [15:0] parallel_in,
input bclk,
input daclrck,
input clk_50,
input read,
input sram_empty,
output dac_in,
output read_edge,
output reg rd_begin,
output reg read_capture
);

reg lrck_prev;//stores the prev value
reg lrck_capture;
reg done;//to reset the capture signal to default 0 value

//edge capture of adclrck
always @ (negedge bclk or negedge ar)
begin
        if(~ar)
        lrck_prev = 1'b0;//ADCLRCK is default 0 signal
        else
        lrck_prev = daclrck;
end

wire lrck_edge = daclrck ^ lrck_prev;//captures both the neg and pos edge using xor operation

always @ (negedge bclk or negedge ar)
begin
        if(~ar)
                lrck_capture = 1'b0;
        else
                if(lrck_edge)
                        lrck_capture = 1'b1;//active high signal
                else
                        if(done)
                                lrck_capture = 1'b0;
end

//parallel to serial conversion

reg [4:0]counter;
```

```
assign dac_in = lrck_capture? parallel_in[5'd15-counter]: 1'b0;

always @ (negedge bclk or negedge ar)
begin
	if(~ar)
		begin
			counter = 5'd0;
			done = 1'd0;
		end
	else
		begin
			if(lrck_capture==1'b1 && done==1'b0)
				begin
					if(counter<=5'd15)
						begin
							counter = counter+5'd1;
							if(counter==5'd16)
							begin
							  done=1'd1;
							end
						end
				end
			else
				begin
					counter=5'd0;
					done=1'd0;
				end
		end//end else
end//end always

reg pass;
//the rd_begin signal is given to the sram controller only when the external read input
// is given by the user
reg read_prev;

//FSM to capture the read input edge
always @ (posedge clk_50 or negedge ar)
	if(~ar)
	read_prev=1'b1;
	else
	read_prev=read;

assign read_edge = ~read & read_prev;

always @ (posedge clk_50 or negedge ar)
	begin
	if(~ar)
				read_capture=1'b0;
	else
				if(read_edge)
					read_capture=1'b1;
				else
					if(sram_empty)
						read_capture=1'b0;//change to 1'b0
	end
```

```
always @ (posedge clk_50 or negedge ar)
	begin
		if(~ar)
			rd_begin = 1'b0;
		else
			begin
				if(daclrck && pass==1'b0 && read_capture==1'b1)
					begin
						rd_begin=1'b1;
						pass=1'b1;
					end
				else
					begin
						rd_begin=1'b0;
						if(~daclrck)
							pass=1'b0;
					end
			end
	end
endmodule
```

codec_ctrl HDL code:

```
//Design component to power down the audio codec
//Author: Ravi Thakur

module codec_ctrl(
	input clk,
	input ar,
	input codec_pwrdwn,
	input read_edge,
	output reg reset,//active high
	output reg codec_disable,//active high
	output clk_br,
	output clk_xck
	);

reg  [9:0]counter;

assign  clk_br=counter[9];//48.8KHz
assign  clk_xck=counter[1];//12.5 MHz

reg codec_pwrdwn_prev;

//posedge capture of codec_pwrdwn
always @(negedge ar or posedge clk)
	if(~ar)
		codec_pwrdwn_prev = 1'b0;
   else
      codec_pwrdwn_prev = codec_pwrdwn;

wire codec_pwrdwn_posedge;
wire codec_pwrdwn_negedge;
```

```
assign codec_pwrdwn_posedge = codec_pwrdwn & ~codec_pwrdwn_prev;//posedge capture
assign codec_pwrdwn_negedge = ~codec_pwrdwn & codec_pwrdwn_prev;//negedge capture

always @(negedge ar or posedge clk)
        if(~ar)
                begin
                        codec_disable = 1'b0;
                end
   else
      if(codec_pwrdwn_posedge)
                        begin
                                codec_disable = 1'b1;
                        end
      else
                        if(codec_pwrdwn_negedge||read_edge)
                                begin
                                        codec_disable = 1'b0;
                                end


always @ (posedge clk or negedge ar)
        if(~ar)
                        reset = 1'b0;
                else
                        if(codec_pwrdwn_posedge||codec_pwrdwn_negedge||read_edge)
                                reset = 1'b1;
                        else
                                reset = 1'b0;

always @(posedge clk or negedge ar)
begin
        if(~ar)
                counter = 10'd0;
        else
                counter=counter+10'd1;
end
endmodule
```

Appendix C - DSP module HDL code

DSP module top level:

```
//DSP module Top level design
//Author: Ravi Thakur

module DSP_module(
input AUD_ADCLRCK,
input clk_filter,
input ar,
input [15:0] filter_in,
input read_capture,
output [15:0] filter_out,
output wr_begin
);

wire ADCLRCK_G;

fir_filter inst1 (ADCLRCK_G,clk_filter,ar,filter_in,filter_out,wr_begin);
filter_clkctrl inst2 (AUD_ADCLRCK,read_capture,ADCLRCK_G);

endmodule
```

fir_filter HDL code:

```
//FIR-filter design
//Author: Ravi Thakur

module fir_filter(clk,clk_50,ar,adc_in,dac_out,wr_begin);

input clk,clk_50,ar;
input signed [15:0]adc_in;
output signed [15:0]dac_out;
output reg wr_begin;

//memory interface
reg [15:0]data_in=16'h0;
wire [4:0]rd_addr;
reg wr_en=1'b0;
wire [15:0]q_out;

reg signed [15:0] filt_coeff[31:0];
reg [4:0] index;
integer counter;
integer count;
reg signed [15:0] adc_shift[31:0];
reg signed [35:0]sum;
reg signed [32:0]multiply;

parameter N=6'd30;

//instantiations
```

```
memory  inst1 (rd_addr,clk,data_in,wr_en,q_out);

assign rd_addr=index;

//read from the memory
always @ (posedge clk )
  begin
  if(index<=31)
          begin
            filt_coeff[index-6'h1]=q_out;
            index=index+5'h1;
          end
  else
           begin
            index=0;
           end
  end

//Filtering

assign dac_out = {sum[35],sum [31:17]} ;

always @ (posedge clk or negedge ar)
          begin
                  if(~ar)
                          begin
                                  sum=36'd0;
                          end
                  else
                          begin
                                  sum=36'b0;
                                  for(count=6'h0;count<6'd31;count=count+6'h1)//shifting
                                          begin
                                                  adc_shift[N-count+1]=adc_shift[N-count];

                                          end
                                                  adc_shift[0]=adc_in;

                                  for(counter=0;counter<=6'd31;counter=counter+6'h1)
                                          begin
                                                  multiply=adc_shift[counter]*filt_coeff[counter];
                                                  sum=sum+multiply;
                                          end

                  end//end else
          end//end always

          reg pass;

always @ (posedge clk_50 or negedge ar)
          begin
                  if(~ar)
                          wr_begin = 1'b0;
                  else
                          begin
                                  if(clk && pass==1'b0)
```

```
                                        begin
                                                wr_begin=1'b1;
                                                pass=1'b1;
                                        end
                                else
                                        begin
                                                wr_begin=1'b0;
                                                if(~clk)
                                                        pass=1'b0;
                                        end
                                end
                end
endmodule
```

filter_clkctrl HDL code:

```
//Filter clock controller design
//Author: Ravi Thakur

module filter_clkctrl(
input AUD_ADCLRCK,
input read_capture,
output ADCLRCK_G
);

assign ADCLRCK_G = ~read_capture && AUD_ADCLRCK;

endmodule
```

Appendix D - Serial module HDL code

serial_module top level:

```
//Serial module Top level design
//Author: Ravi Thakur

module serial_module(
input ar,
input clk_50,
input clk_datarate,
input sl_in,
output [15:0] serial_data,
output wr_ready,
output sl_sleep
);

wire gated_clk_sl;
wire clk_br;

clk_div inst7 (gated_clk_sl,clk_br,ar);
serial_rx inst8 (ar,gated_clk_sl,clk_br,sl_in,serial_data,wr_ready);
serial_clk_ctrl inst9 (clk_50,ar,sl_in,clk_datarate,gated_clk_sl,sl_sleep);

endmodule
```

clk_div HDL code:

```
//Baud rate clock generator design
//Author: Ravi Thakur

module clk_div(clk_in,clk_out,ar);
input clk_in,ar;//clk in is 50 MHz
output reg clk_out;

reg clks=1'b0;

parameter n=12;
parameter reset=1'b0,cont=1'b1;
parameter [n-1:0]limit=12'd2605;//count value to generate the desired clock
reg [n-1:0]counter;

always @ (posedge clk_in or negedge ar)
   begin
      if(~ar)
        begin
          clks=reset;
          clk_out=0;
          counter=0;
        end
      else
        begin
```

```
        case(clks)
          reset: begin
                   counter=0;
                   clks=cont;
                   clk_out=0;
                 end

          cont: begin
                  if(counter==limit-1)
                          begin
                            clk_out=~clk_out;
                                  counter=0;
                           end
                  else
                          begin
                            counter=counter+12'd1;
                                      end
                end
        endcase
      end
    end
endmodule
```

serial_rx HDL code:

```
//Design component that receives the serial data
//Author: Ravi Thakur

module serial_rx(ar,clk,clk_br,sl_in,data_16,wr_ready);

// module external interface
input ar,clk,clk_br,sl_in;
output reg [15:0]data_16;   //the 16 bit word going to the sram block and the filter/encoder block
output reg wr_ready; //wr_ready to the sram block

reg [4:0]count=5'd0;//counter for the 16 bit data going to the sram.
          //It is reset to zero only when the ar is pressed or if the 36th bit is received.
reg [3:0]counter=4'd0;//counter to keep track of no of bits received in each packet
reg [1:0]cs=2'd0;//current state of the FSM

reg flag=1'b0;//sets or resets wr_ready at 50MHz
wire flagstate;

assign flagstate=flag;//wire between sl_rx running at clk_br and the FSM that sets and resets wr_ready at 80MHz

parameter idle=2'b00,receive=2'b01,hold=2'b10;

always @ (posedge clk_br or negedge ar)
        begin
                if(~ar)
                        begin
                                data_16=16'b0;
                                count=5'b0;
                                counter=4'b0;
```

```
                        cs=idle;
                        flag=1'b0;//assuming sram wr_ready active high
                end
        else
                begin
                        case(cs)
                                idle: begin
                                                flag=1'b0;
                                                if(sl_in)
                                                        begin
                                                                flag=1'b0;
                                                                cs=idle;
                                                        end
                                                else
                                                        begin
                                                                cs=receive;
                                                                counter=4'b0;
                                                        end
                                        end

                                receive: begin
                                                data_16[count]=sl_in;
                                                count=count+5'b1;
                                                counter=counter+4'b1;
                                                if(count>15)//16 bits of data has been read
                                                        begin
                                                                flag=1'b1;
                                                                cs=hold;
                                                        end
                                                else
                                                 begin
                                                        if(counter<8)
                                                                begin
                                                                        cs=receive;
                                                                end
                                                        else
                                                                begin
                                                                        cs=idle;
                                                                end
                                                 end
                                         end

                                hold: begin
                                            cs=idle;
                                            count=5'b0;
                                      end
                        endcase
end//end else
end//end always


reg pass;//counter to reset wr_ready after 1 clk cycle.
        always @ (posedge clk)
        begin
                if(flagstate==0)//flag not set in main FSM
                        begin
```

```
                        pass=1'b0;
                        wr_ready=1'b0;
                end
        else
                begin
                        if(~(pass&&flagstate))//if flag is set and this is the first
                                //time the control is passing through this state
                                begin
                                        wr_ready=1'b1;//write start is active high
                                        pass=1;//indicating that one pass has been made
                                end
                        else
                                begin
                                        wr_ready=1'b0;//all consequent passes send the control here
                                                // until flag is reset in the main FSM
                                end
                end
    end//end always
endmodule
```

serial_clk_ctrl HDL code:

```
//serial module clock controller design
//Author: Ravi Thakur

module serial_clk_ctrl(
    input clk_50,
    input ar,
    input sl_in,
    input count_clk,//48.8KHz clk for counting the wait counter upto 3 sec
    output gated_clk_50,
    output reg sl_sleep//led is active high
    );
reg [18:0] count;
reg clk_disable;
assign gated_clk_50 = ~clk_disable && clk_50;
always @ (posedge count_clk or negedge ar)
    begin
        if(~ar)
            begin
                clk_disable=1'b0;
                count = 19'h0;
                sl_sleep = 1'b0;
            end
        else
            begin
                if(sl_in)
                    begin
                        count = count + 19'h1;
                        if(count == 19'h477c0)// the fsm waits for 3 sec
                            begin
                                clk_disable = 1'b1;
                                sl_sleep = 1'b1;
                            end
```

```
                    end
                else
                    begin
                        clk_disable = 1'b0;
                        count = 19'h0;
                        sl_sleep = 1'b0;
                    end
            end//end else
        end//end always
endmodule
```

Appendix E - Memory module HDL code

memory_module HDL code:

```
//Memory module Top level design
//Author: Ravi Thakur

module memory_module(
input clk_mmod,
input ar,
input wr_begin,
input [15:0] filter_out,
input wr_ready,
input [15:0] serial_data,
input rd_begin,
output sram_empty,
output [15:0] sram_out,
inout [15:0] sram_dq,
output [17:0] sram_a,
output sram_oe,
output sram_ce,
output sram_we,
output sram_ub,
output sram_lb
);
wire wr_enable;
wire bank;
wire [15:0] sram_in;
wire rd_fin,wr_fin;
wire rd_start, wr_start;
wire [17:0] addr_rd;
wire [17:0] addr_wr;

bank_select inst11 (clk_mmod,ar,wr_begin,filter_out,wr_ready,serial_data,wr_enable,bank,sram_in,wr_fin);

resolver inst12 (clk_mmod,ar,bank,rd_begin,wr_enable,rd_fin,wr_fin,addr_rd,rd_start,addr_wr,wr_start,
                 sram_empty);

sram_cntrl inst13 (ar,clk_mmod,rd_start,rd_fin,addr_rd,wr_start,wr_fin,addr_wr,sram_in,sram_out,
                   /*Internal (SRAM) interface*/ sram_dq,sram_a,sram_oe,sram_ce,sram_we,sram_ub,sram_lb);
endmodule
```

bank_select HDL code:

```
//Design component to select the memory bank the data needs to be written to
//Author: Ravi Thakur

module bank_select(
input clk,
input ar,
input filter_wr,
input [15:0] filter_data,
```

```
input serial_wr,
input [15:0] serial_data,
output reg wr_begin,//active high
output reg bank,//0-audio data bank,1-serial data bank
output [15:0] sram_in,
input wr_done,
output state0,
output state1,
output state2,
output bank_debug);

//capture the filter write edge
reg filter_wr_prev;
reg filter_wr_captured;
reg filter_wr_clear;
reg [15:0] filter_data_captured;

always @(negedge ar or posedge clk)
        if(~ar)
                filter_wr_prev = 1'b0;
        else
                filter_wr_prev = filter_wr;

wire filter_wr_posedge;

assign filter_wr_posedge = filter_wr & ~filter_wr_prev;//posedge capture

always @(negedge ar or posedge clk)
    if(~ar)
      begin
       filter_wr_captured = 1'b0;
       filter_data_captured = 16'd0;
      end
    else
      if(filter_wr_posedge)
        begin
         filter_wr_captured = 1'b1;
         filter_data_captured = filter_data;
        end
      else
        if(filter_wr_clear)
          filter_wr_captured = 1'b0;

//capture the serial write edge
reg serial_wr_prev;
reg serial_wr_captured;
reg serial_wr_clear;
reg [15:0] serial_data_captured;

always @(negedge ar or posedge clk)
        if(~ar)
                serial_wr_prev = 1'b0;
        else
                serial_wr_prev = serial_wr;

wire serial_wr_posedge;
```

```
assign serial_wr_posedge = serial_wr & ~serial_wr_prev;//posedge capture

always @(negedge ar or posedge clk)
       if(~ar)
         begin
          serial_wr_captured = 1'b0;
          serial_data_captured = 16'd0;
         end
       else
         if(serial_wr_posedge)
           begin
             serial_wr_captured = 1'b1;
             serial_data_captured = serial_data;
           end
         else
           if(serial_wr_clear)
             serial_wr_captured = 1'b0;


assign bank_debug = bank;
reg select;
assign sram_in = select ? serial_data_captured : filter_data_captured;

reg [2:0]state;
assign state0 = state[0];
assign state1 = state[1];
assign state2 = state[2];
parameter idle=3'b000, filterinput=3'b001, filterwait = 3'b010, serialinput=3'b011, serialwait =3'b100;

always @ (posedge clk or negedge ar)
begin
  if(~ar)
    begin
      state=idle;
      wr_begin=1'b0;
      select=1'b0;
      bank=1'b0;
      filter_wr_clear=1'b0;
      serial_wr_clear=1'b0;
    end
   else
     begin
       case(state)
         idle: begin
             if(filter_wr_captured)
               state=filterinput;
             else
               if(serial_wr_captured)
                 state=serialinput;
               else
                 begin
                   state=idle;
                   wr_begin=1'b0;
                   select=1'b0;
                   bank=1'b0;
```

```
            filter_wr_clear=1'b0;
            serial_wr_clear=1'b0;
           end
        end//end idle


    filterinput: begin
                  wr_begin=1'b1;
                  filter_wr_clear=1'b1;
                  bank=1'b0;
                  select=1'b0;
                  state = filterwait;
                 end

    filterwait: begin
                  wr_begin=1'b0;
                  if(~wr_done)
                     begin
                      state=idle;
                     end
                   else
                    begin
                      state=filterwait;
                    end
          end

    serialinput: begin
                   wr_begin=1'b1;
                   serial_wr_clear=1'b1;
                   bank=1'b1;
                   select=1'b1;
                   state=serialwait;
                 end

     serialwait: begin
                   wr_begin=1'b0;
                    if(~wr_done)
                      begin
                           state=idle;
                      end
                     else
                       begin
                           state=filterwait;
                        end
                 end

     default:begin
                state=idle;
                wr_begin=1'b0;
                select=1'b0;
                bank=1'b0;
                filter_wr_clear=1'b0;
                serial_wr_clear=1'b0;
              end
  endcase
end //end else
```

```
end//end always

endmodule
```

resolver HDL code:

```
//Design component that is responsible to generate the addresses for the memory controller
//Author: Ravi Thakur

module resolver(
input clk,
input ar,
input bank_sel,//to select which bank of the sram to write to
input rd_begin,//active high. this indicates when the p2s is ready to send a new 16 bit word
input wr_begin,//signal from bank select indicating a new 16 bit work is ready to be written
input rd_fin,//active low input from the sram controller
input wr_fin,//active low input from the sram controller
output reg [17:0]addr_rd,
output reg rd_start,//active low
output reg [17:0] addr_wr,
output reg wr_start,//active low
output reg sram_empty
);

reg [16:0] bank1_addr_count;
reg [16:0] bank2_addr_count;
reg [17:0] rd_addr_count;
reg [2:0]state;
reg fin;//indicates if all memory addresses have been read once

parameter  idle=3'b000,writeidle1=3'b001,writeidle2=3'b010,emptymem=3'b011,emptyidle=3'b100,
           readwait=3'b101;

parameter bank1 = 18'h00000;//starting address of the memory
parameter bank2 = 18'h20000;//this is binary 100000000000000000 which points to first memory
//location after half of the size of the sram. the first half is for the serial data

always @ (posedge clk or negedge ar)
begin
        if(~ar)
                begin
                        rd_addr_count=18'b0;
                        bank1_addr_count=17'b0;
                        bank2_addr_count=17'b0;
                        wr_start=1'b1;
                        rd_start=1'b1;
                        state=idle;
                        sram_empty=1'b0;
                        fin=1'b0;
                end
        else
                begin
                case(state)
```

```
idle: begin
            sram_empty=1'b0;
              if(rd_begin==1'b1)
                    begin
                          state=emptymem;
                          rd_addr_count=18'h0;
                    end
               else
                    begin
                          if( wr_begin==1'b1 && rd_begin==1'b0)
                                begin
                                if(~bank_sel)
                                begin
                                      wr_start = 1'b0;
                                      addr_wr = bank2 + bank2_addr_count;
                                      state=writeidle2;
                                end
                                else
                                begin
                                      wr_start = 1'b0;
                                      addr_wr = bank1 + bank1_addr_count;
                                      state=writeidle1;
                                end
                                end
                          else
                                begin
                                      state=idle;
                                end
                          end
            end

writeidle1:begin
                    wr_start=1'b1;
                    if(~wr_fin)
                    begin
                          state=idle;
                          bank1_addr_count = bank1_addr_count + 17'h1;
                    end
                    else
                    begin
                          state=writeidle1;
                    end
          end

writeidle2:begin
                    wr_start=1'b1;
                    if(~wr_fin)
                    begin
                          state=idle;
                          bank2_addr_count = bank2_addr_count + 17'h1;
                    end
                    else
                    begin
                          state=writeidle2;
                    end
            end
```

```
                emptymem: begin
                                addr_rd = rd_addr_count;
                                rd_start=1'b0;
                                if(rd_addr_count==18'h3FFFF)
                                        fin=1'b0;
                                else
                                        fin=1'b0;
                                        state=emptyidle;
                                  end

                 emptyidle: begin
                                rd_start=1'b1;
                                if(~rd_fin)
                                begin
                                        rd_addr_count=rd_addr_count+18'h1;
                                        state=readwait;
                                end
                                else
                                begin
                                        state=emptyidle;
                                end
                           end

                 readwait: begin
                             if(rd_begin)
                               begin
                                if(fin==1'b1)
                                begin
                                        sram_empty=1'b1;
                                        state=idle;
                                fin=1'b0;
                                end
                              else
                                  state=emptymem;
                              end
                        end

                 default: begin
                                state=idle;
                                rd_addr_count=18'b0;
                                bank1_addr_count=17'b0;
                                bank2_addr_count=17'b0;
                                wr_start=1'b1;
                                rd_start=1'b1;
                                state=idle;
                                sram_empty=1'b0;
                                fin=1'b0;
                         end
        endcase
end//end else
end//end always
endmodule
```

sram_cntrl HDL code:

```
//Design component to set up the data, address and control signals to the memory
//Author: Ravi Thakur

module sram_cntrl(
  input ar,          // External interface
  input clk,
  input rd_start,
  output reg rd_done,
  input [17:0] rd_a,
  input wr_start,
  output reg wr_done,
  input [17:0] wr_a,
  input [15:0] d_in,
  output reg [15:0] q_out,

  // Internal (SRAM) interface
  inout [15:0] sram_dq,
  output [17:0] sram_a,
  output sram_oe,
  output sram_ce,
  output sram_we,
  output sram_ub,
  output sram_lb

  );

/*  Section to capture or clear start edges */
  reg      wr_start_prev;
  reg      wr_start_captured;
  reg      wr_start_clear;
  reg [17:0] wr_a_captured;
  reg [15:0] d_in_captured;
  reg      rd_start_prev;
  reg      rd_start_captured;
  reg      rd_start_clear;
  reg [17:0] rd_a_captured;

// FF to allow edge detection on wr_start
  always @(negedge ar or posedge clk)
     if(~ar)
       wr_start_prev = 1'b1;
     else
       wr_start_prev = wr_start;

 wire wr_start_edge;
 assign wr_start_edge = ~wr_start & wr_start_prev;

 // Simple FSM to hold or clear captured wr start edges
 always @(negedge ar or posedge clk)
      if(~ar)
        begin
        wr_start_captured = 1'b1;
        wr_a_captured = 18'b0;
```

```
        d_in_captured = 16'b0;
        end
      else
       if(wr_start_edge)
         begin
           wr_start_captured = 1'b0;
           wr_a_captured = wr_a;
           d_in_captured = d_in;
         end
        else
          if(wr_start_clear)
            wr_start_captured = 1'b1;


  wire rd_start_edge;
  assign rd_start_edge = ~rd_start & rd_start_prev;

// FF to allow edge detection on rd_start
   always @(negedge ar or posedge clk)
      if(~ar)
        rd_start_prev = 1'b1;
      else
        rd_start_prev = rd_start;

  // Simple FSM to hold or clear captured rd start edges
  always @(negedge ar or posedge clk)
       if(~ar)
         begin
         rd_start_captured = 1'b1;
         rd_a_captured = 18'b0;
         end
       else
         if(rd_start_edge)
           begin
           rd_start_captured = 1'b0;
           rd_a_captured = rd_a;
           end
         else
           if(rd_start_clear)
             rd_start_captured = 1'b1;



   /*  Section to define and control main FSM */

            parameter [2:0] Idle=3'o0, Begin_Write=3'o1, /*Write_En=3'o2,*/ Write_Fin=3'o3,
                      Begin_Read=3'o4, Read_En=3'o5, Read_Fin=3'o6;

   reg [3:0]  cs;

   // Declare simple control lines to be replicated
   reg      oe;
   reg      ce;
   reg      we;
   reg      be;//for the upper and lower byte enable signal
```

```
  reg      addr_sel;   // = wr_addr if 0, = rd_addr = 1
  assign   sram_a = addr_sel ? rd_a_captured : wr_a_captured;

  assign sram_dq = oe ? d_in_captured : 16'bz;


  assign sram_oe = oe;
  assign sram_ce = ce;
  assign sram_we = we;
  assign sram_ub = be;
  assign sram_lb = be;

always @(negedge ar or posedge clk)
          if(~ar)
            begin
                    cs = Idle;
                    oe = 1'b1;
                    ce = 1'b1;
                    we = 1'b1;
                    be = 1'b1;
                    addr_sel = 1'b0;
                    wr_done = 1'b1;
                    rd_done = 1'b1;
                    wr_start_clear = 1'b0;
                    rd_start_clear = 1'b0;
                    q_out = 16'b0;
             end
          else
          case(cs)
                    Idle :
                      if(~wr_start_captured)
                         begin
                          cs = Begin_Write;
                          oe = 1'b1;
                          ce = 1'b0;
                          we = 1'b0;
                          be = 1'b0;
                          addr_sel = 1'b0;
                          wr_done = 1'b1;
                          rd_done = 1'b1;
                          wr_start_clear = 1'b0;
                          rd_start_clear = 1'b0;
                        end
                           else if(~rd_start_captured)
                                    begin
                                     cs = Begin_Read;
                                     oe = 1'b0;
                                     ce = 1'b0;
                                     we = 1'b1;
                                     be = 1'b0;
                                     addr_sel = 1'b1;
                                      wr_done = 1'b1;
                                      rd_done = 1'b1;
                                      wr_start_clear = 1'b0;
                                      rd_start_clear = 1'b0;
                                 end
```

```
        else
                begin
                    cs = Idle;
                    oe = 1'b1;
                    ce = 1'b1;
                    we = 1'b1;
                    be = 1'b1;
                    addr_sel = 1'b0;
                    wr_done = 1'b1;
                    rd_done = 1'b1;
                    wr_start_clear = 1'b0;
                    rd_start_clear = 1'b0;
                 end

Begin_Read :
              begin
                cs = Read_Fin;
                oe = 1'b0;
                ce = 1'b0;
                we = 1'b1;
                be = 1'b0;
                addr_sel = 1'b1;
                wr_done = 1'b1;
                rd_done = 1'b1;
                wr_start_clear = 1'b0;
                rd_start_clear = 1'b1;
              end

 Read_Fin :
              begin
                cs = Idle;
                oe = 1'b0;
                ce = 1'b0;
                we = 1'b1;
                be = 1'b0;
                addr_sel = 1'b1;
                wr_done = 1'b1;
                rd_done = 1'b0;
                wr_start_clear = 1'b0;
                rd_start_clear = 1'b0;
                q_out = sram_dq; // Grab SRAM Q output
              end

Begin_Write :
              begin
                cs = Write_Fin;
                oe = 1'b1;
                ce = 1'b0;
                we = 1'b0;
                be = 1'b0;
                addr_sel = 1'b0;
                wr_done = 1'b1;
                rd_done = 1'b1;
                wr_start_clear = 1'b1;
                rd_start_clear = 1'b0;
             end
```

```
        Write_Fin :
                    begin
                     cs = Idle;
                     oe = 1'b1;
                     ce = 1'b0;
                     we = 1'b0;
                     be = 1'b0;
                     addr_sel = 1'b0;
                     wr_done = 1'b0;
                     rd_done = 1'b1;
                     wr_start_clear = 1'b0;
                     rd_start_clear = 1'b0;
                    end

        default :
                    begin
                     cs = Idle;
                     oe = 1'b1;
                     ce = 1'b1;
                     we = 1'b1;
                     addr_sel = 1'b0;
                     wr_done = 1'b1;
                     rd_done = 1'b1;
                     wr_start_clear = 1'b0;
                     rd_start_clear = 1'b0;
                    end

endcase
endmodule
```

Appendix F - Clock distributor and device power down components HDL code

Clk_distributor HDL code:

```
//Design clock distributor
//Author: Ravi Thakur

module clk_distributor(
input clk_50,
input ar,
input codec_reset,
input disable_clk,
output clk_codec,
output clk_filter,
output clk_mmod
);

assign clk_codec = (~codec_reset|disable_clk) ? 1'b0 : clk_50;
assign clk_filter = (~ar|disable_clk) ? 1'b0 : clk_50;
assign clk_mmod = (~ar|disable_clk) ? 1'b0 : clk_50;

endmodule
```

device_pwrdwn HDL code:

```
//Design clock controller
//Author: Ravi Thakur

module device_pwrdwn(
input clk,
input ar,
input codec_disable,
input sl_sleep,
output reg device_disable
);

always @ (posedge clk or negedge ar)
	begin
		if(~ar)
			device_disable=1'b0;
		else
			if(sl_sleep==1'b1&codec_disable==1'b1)
				device_disable=1'b1;
			else
				device_disable=1'b0;
	end//end always
endmodule
```

Appendix E - MATLAB code for generating the filter coefficients

```
%Generating Filter Coefficients for FIR Filter
%Ravi Thakur
fs=16e3;%Half of the 48kHz sampling frequency
[n,f0,a0,w]=firpmord([5.2e3,5.4e3],[1,0],[0.0001,0.1],fs);
%The stopband edge frequencies are taken to be beyond 5.4kHz
%0.01 ripple in the passband and stopband
b=firpm(31,f0,a0,w);%filter length of 31
freqz(b,1,1024,fs);%the response is plotted
coeff_to_mif(b,16,'memory.mif');%the coefficients are generated in signed format
%this code provided by the instructor converts them into two's complement
%from to be %used in the verilog code.
%%
b=firpm(30,[0 0.205 0.287 1],[1 1 0 0]);
figure(1);
freqz(b,1,512,48e3);
[h,w]=freqz(b,1,512,48e3);
figure(2);
plot(w,abs(h));
coeff_to_mif(b,16,'memory.mif');
```

coeff_to_mif function:

```
function coeff_to_mif(b, k, filename)
%
%  Use:  coeff_to_mif(b, k, filename)
%          b = floating point coefficients
%          k = # of bits (assuming 2's complement format)
%          filename = name of file (in single quotes)
%
%  Author:  D. Gruenbacher
%
%   Created:  Feb. 28, 2006
%

n = length(b);

b_int = b./abs(max(b))*(2^(k-1)-1);
b_int = round(b_int);

for i=1:n          % Go through negative numbers and convert to 2's comp
   if (b_int(i) < 0)
      b_int(i) = 2^k + b_int(i);
   end
end

fid = fopen(filename, 'wt');    % Open the output file

fprintf(fid, 'DEPTH = %d;\n', n);
fprintf(fid, 'WIDTH = %d;\n', k);
fprintf(fid, 'ADDRESS_RADIX = DEC;\n');
fprintf(fid, 'DATA_RADIX = DEC;\n \n');

fprintf(fid, 'CONTENT\n');
```

```
fprintf(fid, '\t BEGIN \n');

for i=1:n
   fprintf(fid, '%d : %d;\n', i-1, b_int(i));
end

fprintf(fid, 'END;\n');
fclose(fid);

% End of function
```

Appendix F - Analog module testbenches

Analog module input operation:

```
`timescale 1ns/1ns

//Analog module input operation test bench
//Author: Ravi Thakur

module analog_module_tb;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter clk_bclk_pulse=160;//3.125 MHz clock
  parameter bclk_negedge_pulse=320;
  parameter clk_lrclk_pulse=10246;//48.8 KHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter padding = 5126;

  //Input signals
  reg clk_50;
  reg ar;
  reg AUD_ADCDAT;
  reg AUD_BCLK;
  reg AUD_ADCLRCK;
  reg AUD_DACLRCK;
  reg codec_pwrdwn;
  reg read;
  reg [15:0] parallel_in;

  //intermediate signals
  wire [15:0] a2d_bits;

  //sine wave generator

 function real sin;
   input x;
   real x;
   real x1,y,y2,y3,y5,y7,sum,sign;
   begin
    sign = 1.0;
    x1 = x;
    if (x1<0)
      begin
        x1 = -x1;
        sign = -1.0;
      end
    while (x1 > 3.14159265/2.0)
      begin
        x1 = x1 - 3.14159265;
        sign = -1.0*sign;
```

```
      end
    y = x1*2/3.14159265;
    y2 = y*y;
    y3 = y*y2;
    y5 = y3*y2;
    y7 = y5*y2;
    sum = 1.570794*y - 0.645962*y3 +
          0.079692*y5 - 0.004681712*y7;
    sin = sign*sum;
   end
 endfunction

 assign a2d_bits = $rtoi(xin * 32768);

//initial block

  initial
    begin
      $dumpfile("codec_innorop.vcd");
      $dumpvars(0,analog_module_tb);
      #50000000;
      $finish;
    end

  initial
    begin
      t=0;
      ar=1'b1;
      clk_50=1'b1;
      AUD_BCLK=1'b0;
      AUD_ADCLRCK=1'b0;
      AUD_DACLRCK=1'b0;
      codec_pwrdwn=1'b0;
      read=1'b1;
      parallel_in = 16'd0;

      #10 ar =1'b0;
      #30 ar = 1'b1;

    end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end

//50 MHz clock
always
  #clk_pulse clk_50 = ~clk_50;

//3.125 MHz clock
always
  #clk_bclk_pulse AUD_BCLK = ~AUD_BCLK;
```

```
//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_ADCLRCK = ~AUD_ADCLRCK;

//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_DACLRCK = ~AUD_DACLRCK;


always
begin
    #clk_lrclk_pulse     AUD_ADCDAT = a2d_bits[15];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[14];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[13];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[12];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[11];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[10];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[9];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[8];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[7];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[6];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[5];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[4];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[3];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[2];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[1];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[0];
    #clk_bclk_pulse AUD_ADCDAT = 1'b0;
    #clk_bclk_pulse AUD_ADCDAT = 1'b0;
    #padding AUD_ADCDAT = 1'b0;
end
endmodule
```

Analog module output operation:

```
`timescale 1ns/1ns

//Analog module output operation test bench
//Author: Ravi Thakur

module analog_module_tb;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter clk_bclk_pulse=160;//3.125 MHz clock
  parameter bclk_negedge_pulse=320;
  parameter clk_lrclk_pulse=10246;//48.8 KHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter padding = 5126;

  //Input signals
```

```
 reg clk_50;
 reg ar;
 reg AUD_ADCDAT;
 reg AUD_BCLK;
 reg AUD_ADCLRCK;
 reg AUD_DACLRCK;
 reg codec_pwrdwn;
 reg read;
 reg [15:0] parallel_in;

 //intermediate signals
 wire [15:0] a2d_bits;

 //sine wave generator

function real sin;
  input x;
  real x;
  real x1,y,y2,y3,y5,y7,sum,sign;
  begin
   sign = 1.0;
   x1 = x;
   if (x1<0)
     begin
       x1 = -x1;
       sign = -1.0;
     end
   while (x1 > 3.14159265/2.0)
     begin
       x1 = x1 - 3.14159265;
       sign = -1.0*sign;
     end
   y = x1*2/3.14159265;
   y2 = y*y;
   y3 = y*y2;
   y5 = y3*y2;
   y7 = y5*y2;
   sum = 1.570794*y - 0.645962*y3 +
         0.079692*y5 - 0.004681712*y7;
   sin = sign*sum;
  end
endfunction

 assign a2d_bits = $rtoi(xin * 32768);

//initial block

  initial
    begin
      $dumpfile("codec_opnorop.vcd");
      $dumpvars(0,analog_module_tb);
      #50000000;
      $finish;
    end

  initial
```

```
    begin
      t=0;
      ar=1'b1;
      clk_50=1'b1;
      AUD_BCLK=1'b0;
      AUD_ADCLRCK=1'b0;
      AUD_DACLRCK=1'b0;
      codec_pwrdwn=1'b0;
      read=1'b1;

      #10 ar =1'b0;
      #30 ar = 1'b1;
      #20452 read = 1'b0;
      #30 read = 1'b1;

   end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end

//50 MHz clock
always
  #clk_pulse clk_50 = ~clk_50;

//3.125 MHz clock
always
  #clk_bclk_pulse AUD_BCLK = ~AUD_BCLK;

//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_ADCLRCK = ~AUD_ADCLRCK;

//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_DACLRCK = ~AUD_DACLRCK;

always
begin
    #clk_lrclk_pulse     AUD_ADCDAT = a2d_bits[15];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[14];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[13];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[12];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[11];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[10];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[9];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[8];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[7];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[6];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[5];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[4];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[3];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[2];
```

```
   #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[1];
   #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[0];
   #clk_bclk_pulse AUD_ADCDAT = 1'b0;
   #clk_bclk_pulse AUD_ADCDAT = 1'b0;
   #padding AUD_ADCDAT = 1'b0;
end

always
begin
  #clk_lrclk_pulse  parallel_in = a2d_bits;
end

endmodule
```

Analog module shutdown:

```
`timescale 1ns/1ns

//Analog module shutdown test bench
//Author: Ravi Thakur

module analog_module;
 parameter clk_pulse=10;//50 MHz clock
  //Input signals
  reg clk_50;
  reg ar;
  reg AUD_ADCDAT;
  reg AUD_BCLK;
  reg AUD_ADCLRCK;
  reg AUD_DACLRCK;
  reg codec_pwrdwn;
  reg read;
  wire [15:0] parallel_in;
//initial block
  initial
    begin
      $dumpfile("codec_pwrdwnop.vcd");
      $dumpvars(0,analog_module_tb);
      #50000000;
      $finish;
    end

  initial
    begin
      t=0;
      ar=1'b1;
      clk_50=1'b0;
      AUD_ADCDAT=1'b0;
      AUD_BCLK=1'b1;
      AUD_ADCLRCK=1'b1;
      AUD_DACLRCK=1'b1;
      codec_pwrdwn=1'b0;
      read=1'b1;
      #10 ar =1'b0;
      #30 ar = 1'b1;
```

```
      #20 codec_pwrdwn=1'b0;
   end//end initial block

//50 MHz clock
always
  #clk_pulse clk_50 = ~clk_50;
Endmodule
```

Appendix G - Serial module testbenches

Serial module normal operation:

```
`timescale 1ns/1ns

//Serial module normal operation test bench
//Author: Ravi Thakur

module serial_module_tb;
  reg ar;
  reg clk_in;
  reg sl_in;
  reg [7:0] sl_count;
  wire [9:0] serial_in;

parameter clk_pulse=10;//50 MHz clock
parameter serial_rate_posedge = 104167;//9600 serial datarate clock
parameter serial_word_rate = 1145837;
assign serial_in = {1'b1,sl_count,1'b0};//stop bit, 8 bit word, stop bit

  initial
    begin
      $dumpfile("smtb.vcd");
      $dumpvars(0,serial_module_tb);
      #100000000;
      $finish;
    end

//initial block
  initial
    begin
      ar=1'b1;
      sl_in=1'b1;
      clk_in=1'b1;
      sl_count = 8'd64;


      #10 ar =1'b0;
      #30 ar = 1'b1;

    end//end initial block

//50 MHz clock
always
  #clk_pulse clk_in = ~clk_in;

//serial data in
always
  begin
  #serial_rate_posedge sl_in = serial_in[0];
  #serial_rate_posedge sl_in = serial_in[1];
  #serial_rate_posedge sl_in = serial_in[2];
  #serial_rate_posedge sl_in = serial_in[3];
```

```
  #serial_rate_posedge sl_in = serial_in[4];
  #serial_rate_posedge sl_in = serial_in[5];
  #serial_rate_posedge sl_in = serial_in[6];
  #serial_rate_posedge sl_in = serial_in[7];
  #serial_rate_posedge sl_in = serial_in[8];
  #serial_rate_posedge sl_in = serial_in[9];
  end

//serial word increment
always
  #serial_word_rate sl_count = sl_count + 8;

endmodule
```

Serial module shutdown:

```
`timescale 1ns/1ns

//Serial module shutdown test bench
//Author: Ravi Thakur

module serial_module_tb;
  reg ar;
  reg slclk;
  reg count_clk;
  reg sl_in;
parameter clk_pulse=10;//50 MHz clock
parameter serial_rate_posedge = 104167;//9600 serial datarate clock
parameter count_clk_pulse = 10246;//48.8 kHz clock for counting
parameter serial_word_rate = 1145837;

//initial block
 initial
    begin
      $dumpfile("smtb_wclkg.vcd");
      $dumpvars(0,serial_module_tb);
      #100000000;
      $finish;
    end

  initial
    begin
      ar=1'b1;
      sl_in=1'b1;
      slclk=1'b1;
      count_clk=1'b1;
      #10 ar =1'b0;
      #30 ar = 1'b1;
    end//end initial block

//50 MHz clock
always
  #clk_pulse slclk = ~slclk;

//48.8 kHz clock
```

```
always
  #count_clk_pulse count_clk = ~count_clk;
endmodule
```

Appendix H - DSP module testbenches

DSP module normal operation:

```
`timescale 1ns/1ns

//DSP module normal operation test bench
//Author: Ravi Thakur

module filter_module;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter clk_lrclk_pulse=10246;//48.8 KHz clock

  //Input signals
  reg clk_50;
  reg ar;
  reg clk;
  wire [15:0] adc_in;
  wire [15:0] dac_out;
  wire wr_begin;

  //sine wave generator

 function real sin;
   input x;
   real x;
   real x1,y,y2,y3,y5,y7,sum,sign;
   begin
    sign = 1.0;
    x1 = x;
    if (x1<0)
      begin
        x1 = -x1;
        sign = -1.0;
      end
    while (x1 > 3.14159265/2.0)
      begin
        x1 = x1 - 3.14159265;
        sign = -1.0*sign;
      end
    y = x1*2/3.14159265;
    y2 = y*y;
    y3 = y*y2;
    y5 = y3*y2;
    y7 = y5*y2;
    sum = 1.570794*y - 0.645962*y3 +
          0.079692*y5 - 0.004681712*y7;
    sin = sign*sum;
   end
```

```
 endfunction

 assign adc_in = $rtoi(xin * 32768);

//initial block

  initial
    begin
      $dumpfile("fmdsp.vcd");
      $dumpvars(0,filter_module);
      #50000000;
      $finish;
    end

  initial
    begin
      t=0;
      ar=1'b1;
      clk_50=1'b1;
      clk=1'b1;

      #10 ar =1'b0;
      #30 ar = 1'b1;
   end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end
//50 MHz clock
always
  #clk_pulse clk_50 = ~clk_50;

//48.8 KHz clock
always
  #clk_lrclk_pulse clk = ~clk;

endmodule
```

DSP module shutdown:

```
`timescale 1ns/1ns

//DSP module shutdown test bench
//Author: Ravi Thakur

module fm_dsp;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
```

```
 parameter clk_lrclk_pulse=10246;//48.8 KHz clock

 //Input signals
 reg clk_50;
 reg ar;
 reg clk;
 reg disable1;
 wire [15:0] adc_in;

 //sine wave generator

function real sin;
  input x;
  real x;
  real x1,y,y2,y3,y5,y7,sum,sign;
  begin
   sign = 1.0;
   x1 = x;
   if (x1<0)
     begin
       x1 = -x1;
       sign = -1.0;
     end
   while (x1 > 3.14159265/2.0)
     begin
       x1 = x1 - 3.14159265;
       sign = -1.0*sign;
     end
   y = x1*2/3.14159265;
   y2 = y*y;
   y3 = y*y2;
   y5 = y3*y2;
   y7 = y5*y2;
   sum = 1.570794*y - 0.645962*y3 +
         0.079692*y5 - 0.004681712*y7;
   sin = sign*sum;
  end
endfunction

 assign adc_in = $rtoi(xin * 32768);

//initial block

 initial
   begin
     $dumpfile("fmdsp32_dis.vcd");
     $dumpvars(0,fm_dsp);
     #50000000;
     $finish;
   end


 initial
   begin
     t=0;
     ar=1'b1;
```

```
      clk_50=1'b1;
      clk=1'b1;
      disable1=1'b0;

      #10 ar =1'b0;
      #30 ar = 1'b1;
      #60 disable1 = 1'b1;

   end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end
//50 MHz clock
always
  #clk_pulse clk_50 = ~clk_50;

//48.8 KHz clock
always
  #clk_lrclk_pulse clk = ~clk;

endmodule
```

Appendix I - Memory module testbenches

Memory module write operation:

```
`timescale 1ns/1ns

//Memory module write operation test bench
//Author: Ravi Thakur

module memory_module_tb;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter serial_word_rate = 1145837;
  parameter serial_2word_rate = 21875074;

  //Input signals
  reg clk;
  reg ar;
  reg filter_wr;
  wire [15:0] filter_data;
  reg serial_wr;
  wire [15:0] serial_data;
  reg rd_begin;
  reg device_pwrdwn;

  //inout signals
  wire [15:0] sram_dq;
  wire [15:0]q;

  //intermediate signals
  wire [15:0] a2d_bits;
  reg [15:0] serial_bits = 16'd64;
  reg oe;

  assign sram_dq = oe ? 16'bz : a2d_bits;
  assign q=sram_dq;

  //sine wave generator

 function real sin;
   input x;
   real x;
   real x1,y,y2,y3,y5,y7,sum,sign;
   begin
    sign = 1.0;
    x1 = x;
    if (x1<0)
      begin
        x1 = -x1;
```

```
      sign = -1.0;
    end
  while (x1 > 3.14159265/2.0)
    begin
      x1 = x1 - 3.14159265;
      sign = -1.0*sign;
    end
  y = x1*2/3.14159265;
  y2 = y*y;
  y3 = y*y2;
  y5 = y3*y2;
  y7 = y5*y2;
  sum = 1.570794*y - 0.645962*y3 +
        0.079692*y5 - 0.004681712*y7;
  sin = sign*sum;
 end
endfunction

assign a2d_bits = $rtoi(xin * 32768);
assign filter_data = a2d_bits;
assign serial_data = serial_bits;

//initial block
 initial
  begin
    $dumpfile("mem_nwrop.vcd");
    $dumpvars(0,memory_module_tb);
    #50000000;
    $finish;
  end

 initial
  begin
    t=0;
    ar=1'b1;
    clk=1'b1;
    oe=1'b0;//sram_dq is output
    filter_wr=1'b0;
    serial_wr=1'b0;
    rd_begin=1'b0;
    device_pwrdwn=1'b0;

    #10 ar =1'b0;
    #30 ar = 1'b1;

  end//end initial block

//sine wave always block
always
 begin
  #tstep t = t + tstep;
  xin = sin(2*pi*f*t*1e-9);
 end

//50 MHz clock
always
```

```
  #clk_pulse clk = ~clk;

//filter wr_ready pulse
always
begin
  #lrclk_posedge_pulse filter_wr=1'b1;
  #30 filter_wr=1'b0;
end

//serial wr_ready pulse
always
begin
  #serial_2word_rate serial_wr=1'b1;
  #30 serial_wr=1'b0;
end

always
  begin
  #serial_word_rate serial_bits = serial_bits+6;
  end

endmodule
```

Memory module read operation:

```
`timescale 1ns/1ns

//Memory module read operation test bench
//Author: Ravi Thakur

module memory_module_tb;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter serial_word_rate = 1145837;
  parameter serial_2word_rate = 21875074;

  //Input signals
  reg clk;
  reg ar;
  reg filter_wr;
  wire [15:0] filter_data;
  reg serial_wr;
  wire [15:0] serial_data;
  reg rd_begin;
  reg device_pwrdwn;

  //inout signals
  wire [15:0] sram_dq;
  wire [15:0]q;
```

```
 //intermediate signals
 wire [15:0] a2d_bits;
 reg [15:0] serial_bits = 16'd64;
 reg oe;

 assign sram_dq = oe ? 16'bz : a2d_bits;
 assign q=sram_dq;

 //sine wave generator

function real sin;
  input x;
  real x;
  real x1,y,y2,y3,y5,y7,sum,sign;
  begin
   sign = 1.0;
   x1 = x;
   if (x1<0)
     begin
       x1 = -x1;
       sign = -1.0;
     end
   while (x1 > 3.14159265/2.0)
     begin
       x1 = x1 - 3.14159265;
       sign = -1.0*sign;
     end
   y = x1*2/3.14159265;
   y2 = y*y;
   y3 = y*y2;
   y5 = y3*y2;
   y7 = y5*y2;
   sum = 1.570794*y - 0.645962*y3 +
         0.079692*y5 - 0.004681712*y7;
   sin = sign*sum;
  end
endfunction

 assign a2d_bits = $rtoi(xin * 32768);

 assign filter_data = a2d_bits;
 assign serial_data = serial_bits;

//initial block

  initial
    begin
      $dumpfile("mem_nrdrop.vcd");
      $dumpvars(0,memory_module_tb);
      #50000000;
      $finish;
    end

  initial
    begin
      t=0;
```

```
      ar=1'b1;
      clk=1'b1;
      oe=1'b0;//sram_dq is output
      filter_wr=1'b0;
      serial_wr=1'b0;
      rd_begin=1'b0;
      device_pwrdwn=1'b0;

      #10 ar =1'b0;
      #30 ar = 1'b1;

   end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end

//50 MHz clock
always
  #clk_pulse clk = ~clk;

//filter wr_ready pulse
always
begin
  #lrclk_posedge_pulse filter_wr=1'b1;
  #30 filter_wr=1'b0;
end

//serial wr_ready pulse
always
begin
  #serial_2word_rate serial_wr=1'b1;
  #30 serial_wr=1'b0;
end

//incrementing serial data
always
  begin
  #serial_word_rate serial_bits = serial_bits+6;
  end

//generating rd_start signals
always
  begin
    #lrclk_posedge_pulse rd_begin = 1'b1;
    #30 rd_begin = 1'b0;
  end

endmodule
```

Memory module shutdown:

```
`timescale 1ns/1ns

//Memory module shutdown operation test bench
//Author: Ravi Thakur

module memory_module_tb;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter serial_word_rate = 1145837;
  parameter serial_2word_rate = 21875074;

  //Input signals
  reg clk;
  reg ar;
  reg filter_wr;
  wire [15:0] filter_data;
  reg serial_wr;
  wire [15:0] serial_data;
  reg rd_begin;
  reg device_pwrdwn;

  //inout signals
  wire [15:0] sram_dq;
  wire [15:0]q;

  //intermediate signals
  wire [15:0] a2d_bits;
  reg [15:0] serial_bits = 16'd64;
  reg oe;

  assign sram_dq = oe ? 16'bz : a2d_bits;
  assign q=sram_dq;

  //sine wave generator

 function real sin;
   input x;
   real x;
   real x1,y,y2,y3,y5,y7,sum,sign;
   begin
    sign = 1.0;
    x1 = x;
    if (x1<0)
      begin
        x1 = -x1;
        sign = -1.0;
      end
    while (x1 > 3.14159265/2.0)
```

```
      begin
        x1 = x1 - 3.14159265;
        sign = -1.0*sign;
      end
    y = x1*2/3.14159265;
    y2 = y*y;
    y3 = y*y2;
    y5 = y3*y2;
    y7 = y5*y2;
    sum = 1.570794*y - 0.645962*y3 +
          0.079692*y5 - 0.004681712*y7;
    sin = sign*sum;
   end
 endfunction

 assign a2d_bits = $rtoi(xin * 32768);
 assign filter_data = a2d_bits;
 assign serial_data = serial_bits;

//initial block

  initial
    begin
      $dumpfile("mem_dpwrdwn.vcd");
      $dumpvars(0,memory_module_tb);
      #50000000;
      $finish;
    end

  initial
    begin
      t=0;
      ar=1'b1;
      clk=1'b1;
      oe=1'b0;//sram_dq is output
      filter_wr=1'b0;
      serial_wr=1'b0;
      rd_begin=1'b0;
      device_pwrdwn=1'b0;

      #10 ar =1'b0;
      #30 ar = 1'b1;
      #20 device_pwrdwn = 1'b1;

    end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end

//50 MHz clock
always
  #clk_pulse clk = ~clk;
```

```
//filter wr_ready pulse
always
begin
  #lrclk_posedge_pulse filter_wr=1'b1;
  #30 filter_wr=1'b0;
end

//serial wr_ready pulse
always
begin
  #serial_2word_rate serial_wr=1'b1;
  #30 serial_wr=1'b0;
end

//incrementing serial data
always
  begin
  #serial_word_rate serial_bits = serial_bits+6;
  end

//generating rd_start signals
always
  begin
    #lrclk_posedge_pulse rd_begin = 1'b1;
    #30 rd_begin = 1'b0;
  end

endmodule
```

Appendix I - Signal acquisition module operating modes testbenches

Normal operation input:

```
`timescale 1ns/1ns

//Final design input operation test bench
//Author: Ravi Thakur

module normal_op_in;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter clk_bclk_pulse=160;//3.125 MHz clock
  parameter bclk_negedge_pulse=320;
  parameter clk_lrclk_pulse=10246;//48.8 KHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter padding = 5126;
  parameter serial_rate_posedge = 104167;
  parameter serial_word_rate = 1145837;

  //Input signals
  reg clk_50;
  reg ar;
  reg AUD_ADCDAT;
  reg AUD_BCLK;
  reg AUD_ADCLRCK;
  reg AUD_DACLRCK;
  reg codec_pwrdwn;
  reg read;
  reg sl_in;
  reg [7:0] sl_count = 8'd64;
  wire [9:0] serial_in;
  reg enable;

  //intermediate signals
  wire [15:0] a2d_bits;

  //sine wave generator

 function real sin;
   input x;
   real x;
   real x1,y,y2,y3,y5,y7,sum,sign;
   begin
    sign = 1.0;
    x1 = x;
    if (x1<0)
      begin
        x1 = -x1;
        sign = -1.0;
```

```
      end
    while (x1 > 3.14159265/2.0)
      begin
        x1 = x1 - 3.14159265;
        sign = -1.0*sign;
      end
    y = x1*2/3.14159265;
    y2 = y*y;
    y3 = y*y2;
    y5 = y3*y2;
    y7 = y5*y2;
    sum = 1.570794*y - 0.645962*y3 +
          0.079692*y5 - 0.004681712*y7;
    sin = sign*sum;
   end
 endfunction

 assign a2d_bits = $rtoi(xin * 32768);
 assign serial_in = ~enable ?{1'b1,sl_count,1'b0} : 10'b1111111111;

//initial block

  initial
    begin
      $dumpfile("nopin.vcd");
      $dumpvars(0,normal_op_in);
      #50000000;
      $finish;
    end

  initial
    begin
      t=0;
      ar=1'b1;
      sl_in=1'b1;
      clk_50=1'b1;
      AUD_BCLK=1'b0;
      AUD_ADCLRCK=1'b0;
      AUD_DACLRCK=1'b0;
      codec_pwrdwn=1'b0;
      read=1'b1;
      enable = 1'b0;

      #10 ar =1'b0;
      #30 ar = 1'b1;
   end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end
//50 MHz clock
always
  #clk_pulse clk_50 = ~clk_50;
```

```
//3.125 MHz clock
always
  #clk_bclk_pulse AUD_BCLK = ~AUD_BCLK;

//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_ADCLRCK = ~AUD_ADCLRCK;

//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_DACLRCK = ~AUD_DACLRCK;

always
begin
    #clk_lrclk_pulse     AUD_ADCDAT = a2d_bits[15];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[14];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[13];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[12];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[11];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[10];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[9];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[8];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[7];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[6];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[5];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[4];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[3];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[2];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[1];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[0];
    #clk_bclk_pulse AUD_ADCDAT = 1'b0;
    #clk_bclk_pulse AUD_ADCDAT = 1'b0;
    #padding AUD_ADCDAT = 1'b0;
end

always
  begin
  #serial_word_rate sl_count = sl_count + 1;
  end

always
  begin
  #52084 sl_in = 1'b1;
  #serial_rate_posedge sl_in = serial_in[0];
  #serial_rate_posedge sl_in = serial_in[1];
  #serial_rate_posedge sl_in = serial_in[2];
  #serial_rate_posedge sl_in = serial_in[3];
  #serial_rate_posedge sl_in = serial_in[4];
  #serial_rate_posedge sl_in = serial_in[5];
  #serial_rate_posedge sl_in = serial_in[6];
  #serial_rate_posedge sl_in = serial_in[7];
  #serial_rate_posedge sl_in = serial_in[8];
  #serial_rate_posedge sl_in = serial_in[9];
  #52084 sl_in = 1'b1;
  end
```

```
endmodule
```

Normal operation output:

```
`timescale 1ns/1ns

//Final design output operation test bench
//Author: Ravi Thakur

module normal_op_out;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter clk_bclk_pulse=160;//3.125 MHz clock
  parameter bclk_negedge_pulse=320;
  parameter clk_lrclk_pulse=10246;//48.8 KHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter padding = 5126;
  parameter serial_rate_posedge = 104167;
  parameter serial_word_rate = 1145837;

  //Input signals
  reg clk_50;
  reg ar;
  reg AUD_ADCDAT;
  reg AUD_BCLK;
  reg AUD_ADCLRCK;
  reg AUD_DACLRCK;
  reg codec_pwrdwn;
  reg read;
  reg sl_in;
  reg [7:0] sl_count = 8'd64;
  wire [9:0] serial_in;
  reg enable;
  reg oe;

  wire [15:0] sram_dq;
  wire [15:0]q;

  //intermediate signals
  wire [15:0] a2d_bits;

  assign sram_dq = oe ? 16'bz : a2d_bits;
  assign q=sram_dq;



 //sine wave generator

 function real sin;
   input x;
```

```
  real x;
  real x1,y,y2,y3,y5,y7,sum,sign;
  begin
   sign = 1.0;
   x1 = x;
   if (x1<0)
     begin
       x1 = -x1;
       sign = -1.0;
     end
    while (x1 > 3.14159265/2.0)
     begin
       x1 = x1 - 3.14159265;
       sign = -1.0*sign;
     end
    y = x1*2/3.14159265;
    y2 = y*y;
    y3 = y*y2;
    y5 = y3*y2;
    y7 = y5*y2;
    sum = 1.570794*y - 0.645962*y3 +
          0.079692*y5 - 0.004681712*y7;
    sin = sign*sum;
   end
 endfunction

 assign a2d_bits = $rtoi(xin * 32768);
 assign serial_in = ~enable ?{1'b1,sl_count,1'b0} : 10'b1111111111;

//initial block

  initial
   begin
     $dumpfile("nopout.vcd");
     $dumpvars(0,normal_op_out);
     #50000000;
     $finish;
   end

  initial
    begin
      t=0;
      ar=1'b1;
      sl_in=1'b1;
      clk_50=1'b1;
      AUD_BCLK=1'b0;
      AUD_ADCLRCK=1'b0;
      AUD_DACLRCK=1'b0;
      codec_pwrdwn=1'b0;
      read=1'b1;
      enable = 1'b0;
      oe=1'b0;

      #10 ar =1'b0;
      #30 ar = 1'b1;
      #50 read=1'b0;
```

```
        oe=1'b0;
      #40 read=1'b1;

   end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end

//50 MHz clock
always
  #clk_pulse clk_50 = ~clk_50;

//3.125 MHz clock
always
  #clk_bclk_pulse AUD_BCLK = ~AUD_BCLK;

//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_ADCLRCK = ~AUD_ADCLRCK;

//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_DACLRCK = ~AUD_DACLRCK;

always
begin
    #clk_lrclk_pulse     AUD_ADCDAT = a2d_bits[15];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[14];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[13];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[12];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[11];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[10];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[9];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[8];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[7];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[6];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[5];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[4];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[3];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[2];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[1];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[0];
    #clk_bclk_pulse AUD_ADCDAT = 1'b0;
    #clk_bclk_pulse AUD_ADCDAT = 1'b0;
    #padding AUD_ADCDAT = 1'b0;
end


always
  begin
  #serial_word_rate sl_count = sl_count + 1;
  end
```

```
always
  begin
  #52084 sl_in = 1'b1;
  #serial_rate_posedge sl_in = serial_in[0];
  #serial_rate_posedge sl_in = serial_in[1];
  #serial_rate_posedge sl_in = serial_in[2];
  #serial_rate_posedge sl_in = serial_in[3];
  #serial_rate_posedge sl_in = serial_in[4];
  #serial_rate_posedge sl_in = serial_in[5];
  #serial_rate_posedge sl_in = serial_in[6];
  #serial_rate_posedge sl_in = serial_in[7];
  #serial_rate_posedge sl_in = serial_in[8];
  #serial_rate_posedge sl_in = serial_in[9];
  #52084 sl_in = 1'b1;
  end
endmodule
```

Serial powerdown:

```
`timescale 1ns/1ns

//Final design serial powerdown test bench
//Author: Ravi Thakur

module sl_pwrdwn;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter clk_bclk_pulse=160;//3.125 MHz clock
  parameter bclk_negedge_pulse=320;
  parameter clk_lrclk_pulse=10246;//48.8 KHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter padding = 5126;
  parameter serial_rate_posedge = 104167;
  parameter serial_word_rate = 1145837;

  //Input signals
  reg clk_50;
  reg ar;
  reg AUD_ADCDAT;
  reg AUD_BCLK;
  reg AUD_ADCLRCK;
  reg AUD_DACLRCK;
  reg codec_pwrdwn;
  reg read;
  reg sl_in;
  reg [7:0] sl_count = 8'd64;
  wire [9:0] serial_in;
  reg enable;

  //intermediate signals
```

```
 wire [15:0] a2d_bits;

 //sine wave generator
function real sin;
  input x;
  real x;
  real x1,y,y2,y3,y5,y7,sum,sign;
  begin
   sign = 1.0;
   x1 = x;
   if (x1<0)
     begin
       x1 = -x1;
       sign = -1.0;
     end
   while (x1 > 3.14159265/2.0)
     begin
       x1 = x1 - 3.14159265;
       sign = -1.0*sign;
     end
   y = x1*2/3.14159265;
   y2 = y*y;
   y3 = y*y2;
   y5 = y3*y2;
   y7 = y5*y2;
   sum = 1.570794*y - 0.645962*y3 +
         0.079692*y5 - 0.004681712*y7;
   sin = sign*sum;
  end
endfunction

 assign a2d_bits = $rtoi(xin * 32768);
 assign serial_in = ~enable ?{1'b1,sl_count,1'b0} : 10'b1111111111;

//initial block

 initial
  begin
    $dumpfile("slpwrdwn.vcd");
    $dumpvars(0,sl_pwrdwn);
    #50000000;
    $finish;
  end

 initial
   begin
     t=0;
     ar=1'b1;
     sl_in=1'b1;
     clk_50=1'b1;
     AUD_BCLK=1'b0;
     AUD_ADCLRCK=1'b0;
     AUD_DACLRCK=1'b0;
     codec_pwrdwn=1'b0;
     read=1'b1;
     enable = 1'b0;
```

```
      #10 ar =1'b0;
      #30 ar = 1'b1;
      #50 enable = 1'b1;

   end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end

//50 MHz clock
always
  #clk_pulse clk_50 = ~clk_50;

//3.125 MHz clock
always
  #clk_bclk_pulse AUD_BCLK = ~AUD_BCLK;

//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_ADCLRCK = ~AUD_ADCLRCK;

//48.8 KHz clock
always
  #clk_lrclk_pulse AUD_DACLRCK = ~AUD_DACLRCK;

always
begin
    #clk_lrclk_pulse     AUD_ADCDAT = a2d_bits[15];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[14];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[13];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[12];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[11];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[10];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[9];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[8];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[7];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[6];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[5];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[4];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[3];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[2];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[1];
    #bclk_negedge_pulse  AUD_ADCDAT = a2d_bits[0];
    #clk_bclk_pulse AUD_ADCDAT = 1'b0;
    #clk_bclk_pulse AUD_ADCDAT = 1'b0;
    #padding AUD_ADCDAT = 1'b0;
end

always
  begin
  #serial_word_rate sl_count = sl_count + 1;
```

```
  end

always
  begin
  #52084 sl_in = 1'b1;
  #serial_rate_posedge sl_in = serial_in[0];
  #serial_rate_posedge sl_in = serial_in[1];
  #serial_rate_posedge sl_in = serial_in[2];
  #serial_rate_posedge sl_in = serial_in[3];
  #serial_rate_posedge sl_in = serial_in[4];
  #serial_rate_posedge sl_in = serial_in[5];
  #serial_rate_posedge sl_in = serial_in[6];
  #serial_rate_posedge sl_in = serial_in[7];
  #serial_rate_posedge sl_in = serial_in[8];
  #serial_rate_posedge sl_in = serial_in[9];
  #52084 sl_in = 1'b1;
  end
endmodule
```

Analog powerdown:

```
`timescale 1ns/1ns

//Final design analog powerdown test bench
//Author: Ravi Thakur

module analog_powerdown;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
  parameter clk_pulse=10;//50 MHz clock
  parameter clk_bclk_pulse=160;//3.125 MHz clock
  parameter bclk_negedge_pulse=320;
  parameter clk_lrclk_pulse=10246;//48.8 KHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter padding = 5126;
  parameter serial_rate_posedge = 104167;
  parameter serial_word_rate = 1145837;

  //Input signals
  reg clk_50;
  reg ar;
  reg AUD_ADCDAT;
  reg AUD_BCLK;
  reg AUD_ADCLRCK;
  reg AUD_DACLRCK;
  reg codec_pwrdwn;
  reg read;
  reg sl_in;
  reg [7:0] sl_count = 8'd64;
  wire [9:0] serial_in;
  reg enable;
```

```
 //intermediate signals
 wire [15:0] a2d_bits;

 //sine wave generator

function real sin;
  input x;
  real x;
  real x1,y,y2,y3,y5,y7,sum,sign;
  begin
   sign = 1.0;
   x1 = x;
   if (x1<0)
     begin
       x1 = -x1;
       sign = -1.0;
     end
   while (x1 > 3.14159265/2.0)
     begin
       x1 = x1 - 3.14159265;
       sign = -1.0*sign;
     end
   y = x1*2/3.14159265;
   y2 = y*y;
   y3 = y*y2;
   y5 = y3*y2;
   y7 = y5*y2;
   sum = 1.570794*y - 0.645962*y3 +
         0.079692*y5 - 0.004681712*y7;
   sin = sign*sum;
  end
endfunction

assign a2d_bits = $rtoi(xin * 32768);
assign serial_in = ~enable ?{1'b1,sl_count,1'b0} : 10'b1111111111;

//initial block
 initial
   begin
     $dumpfile("apwrdwn.vcd");
     $dumpvars(0,analog_powerdown);
     #50000000;
     $finish;
   end

 initial
   begin
     t=0;
     ar=1'b1;
     sl_in=1'b1;
     clk_50=1'b1;
     AUD_BCLK=1'b0;
     AUD_ADCLRCK=1'b0;
     AUD_DACLRCK=1'b0;
     codec_pwrdwn=1'b0;
     read=1'b1;
```

```
    enable = 1'b0;
    AUD_ADCDAT = 1'b0;

    #10 ar =1'b0;
    #30 ar = 1'b1;
    #50 codec_pwrdwn = 1'b1;

  end//end initial block

//sine wave always block
always
  begin
    #tstep t = t + tstep;
    xin = sin(2*pi*f*t*1e-9);
  end

//50 MHz clock
always
  #clk_pulse clk_50 = ~clk_50;

always
  begin
  #serial_word_rate sl_count = sl_count + 1;
  end

always
  begin
  #52084 sl_in = 1'b1;
  #serial_rate_posedge sl_in = serial_in[0];
  #serial_rate_posedge sl_in = serial_in[1];
  #serial_rate_posedge sl_in = serial_in[2];
  #serial_rate_posedge sl_in = serial_in[3];
  #serial_rate_posedge sl_in = serial_in[4];
  #serial_rate_posedge sl_in = serial_in[5];
  #serial_rate_posedge sl_in = serial_in[6];
  #serial_rate_posedge sl_in = serial_in[7];
  #serial_rate_posedge sl_in = serial_in[8];
  #serial_rate_posedge sl_in = serial_in[9];
  #52084 sl_in = 1'b1;
  end
endmodule
```

Device powerdown:

```
`timescale 1ns/1ns

//Final design device powerdown test bench
//Author: Ravi Thakur

module device_powerdown;

  real t,xin;
  parameter tstep = 20492;//for 48.8KHz sampling freq
  parameter f=4e3;//sin wave freq
  parameter pi=3.14159;
```

```
  parameter clk_pulse=10;//50 MHz clock
  parameter clk_bclk_pulse=160;//3.125 MHz clock
  parameter bclk_negedge_pulse=320;
  parameter clk_lrclk_pulse=10246;//48.8 KHz clock
  parameter lrclk_posedge_pulse = 20492;
  parameter padding = 5126;
  parameter serial_rate_posedge = 104167;
  parameter serial_word_rate = 1145837;

  //Input signals
  reg clk_50;
  reg ar;
  reg AUD_ADCDAT;
  reg AUD_BCLK;
  reg AUD_ADCLRCK;
  reg AUD_DACLRCK;
  reg codec_pwrdwn;
  reg read;
  reg sl_in;

//initial block
  initial
   begin
     $dumpfile("dpwrdwn.vcd");
     $dumpvars(0,device_powerdown);
     #50000000;
     $finish;
   end

  initial
    begin
      t=0;
      ar=1'b1;
      sl_in=1'b1;
      clk_50=1'b1;
      AUD_BCLK=1'b0;
      AUD_ADCLRCK=1'b0;
      AUD_DACLRCK=1'b0;
      codec_pwrdwn=1'b0;
      read=1'b1;
      AUD_ADCDAT = 1'b0;
            sl_in = 1'b1;

      #10 ar =1'b0;
      #30 ar = 1'b1;
      #50 codec_pwrdwn = 1'b1;

    end//end initial block

//50 MHz clock
 always
   #clk_pulse clk_50 = ~clk_50;
 endmodule
```