PROGRAMMABLE WALSH PATTERN GENERATOR USING CPLD

A

Project Report

Submitted to

GIANT METREWAVE RADIO TELESCOPE,

(National Centre for Radio Astrophysics),
(Tata Institute of Fundamental Research)

Under the Guidance of

Mr. Ajith Kumar B.
Engineer-E
(GMRT-NCRA-TIFR)

&

Mr. Navnath D. Shinde
Technical Assistant-C
(GMRT-NCRA-TIFR)

Submitted by:

Meha Kainth
B.TECH (ECE)
(Jaypee University of Information Technology)

Date:

CERTIFICATE

This is to certify that ***Ms. Meha Kainth***, a graduate student of B.Tech, Jaypee University of Information Technology, Solan (HP) has undertaken a project entitled ***"Programmable Walsh Pattern Generator using CPLD"*** at this institute from 15.06.2009 to 26.08.2009. She has done very good work and completed the project successfully.

Ajith Kumar B.
Engineer – E
GMRT –NCRA-TIFR

Navnath D. Shinde
Technical Assistant-C
GMRT –NCRA-TIFR

ACKNOWLEDGEMENTS

Working at Giant Meterwave Radio Telescope for student training program (STP) has been enriching experience for me. This project is result of many people's dedication, inspiration, guidance, knowledge and moral support. I would like to take this opportunity to thank them.

My foremost thanks to my guides, Mr. Ajith Kumar B. and Mr. Navnath D. Shinde, both of whom have helped me in every possible way in my project. It was really kind of them to devote their precious time and solve my difficulties. They were always eager to help me whenever I approached them, be it day or night. They not only cleared my concepts during this project, but also deepened my knowledge of the related topics.

I am highly grateful to the honorable Center Director of NCRA-GMRT, Mr. Rajaram Nityananda; Chief Scientist Mr. Yashwant Gupta; for giving me a chance to work in such a fabulous environment with all intellectuals around. I would like to express deep gratitude towards Mrs. N.S. Deshmukh, STP Coordinator, GMRT, for solving all my problems and offering me help to make my stay here more comfortable.

I feel delighted to express my sincere thanks to all the members of the Analog Lab, Ms. Sweta Gupta, Mr. Satpal Gole, Mr. Sudhir Phakatkar, Mr. R. Gosavi, Mr. Ramdas and Mr. Mangesh Bhor for providing me guidance and help through various stages of the project. I would also like to thank Srinivas for helping me with my project work and Aarti Sandikar for her cooperation.

Last but not the least; I am thankful to Mr. S. Sabhapathy for his support and sharing his beliefs and ideas with me and other STP students and for taking us all for the wonderful Shirdi trip and am also thankful to my friends Rolly Seth, Pritama Kundu, Abhijeet for their support.

Finally, I want to say thanks to all those who helped me directly or indirectly during this project and during my stay at GMRT Observatory.

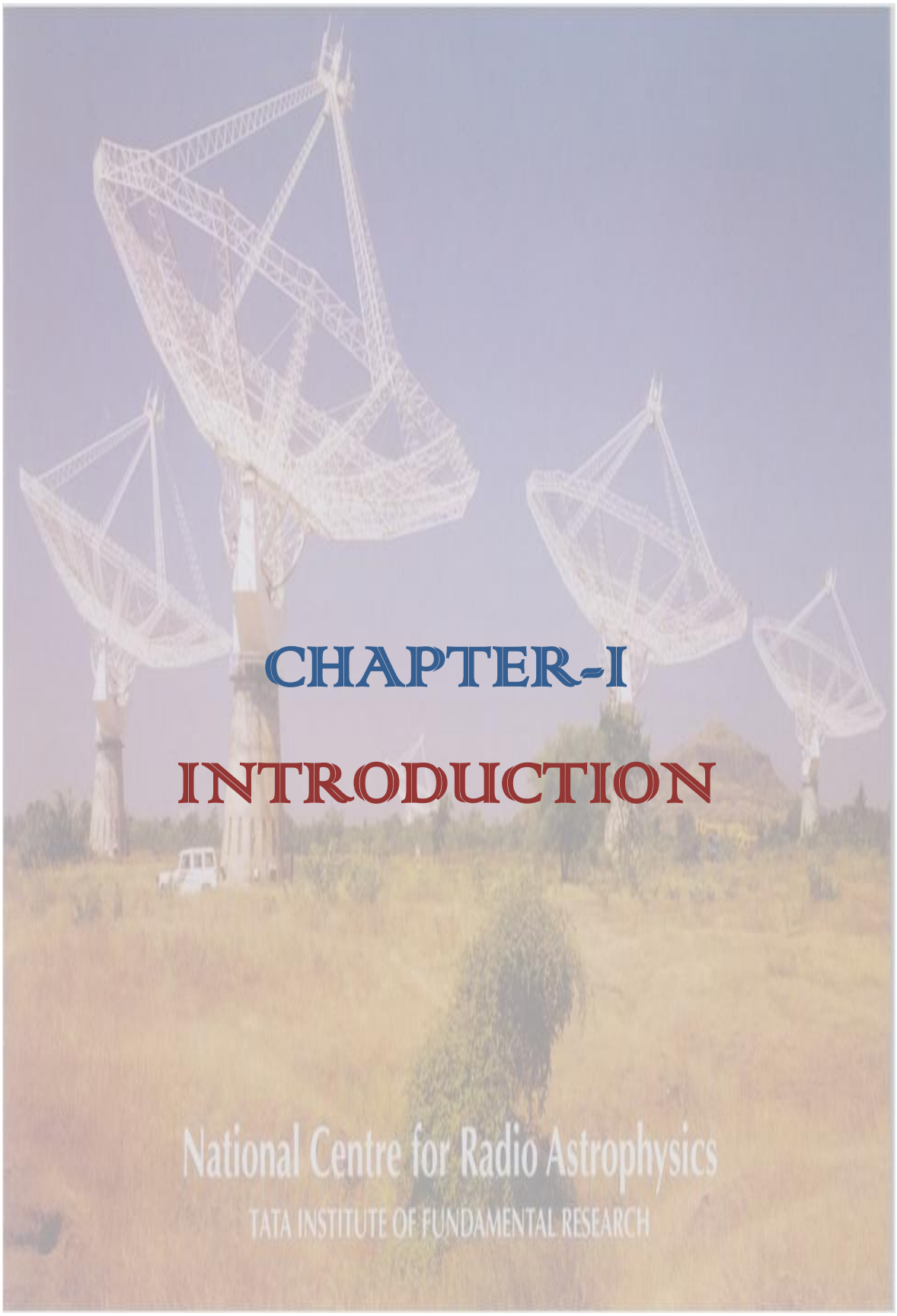
Meha Kainth

CONTENTS

ABSTRACT

The VLSI technology has made revolutionary changes in status of electronic industry and capturing the digital design by schematic entry, hardware description or combination of both. With the recent advancements in Field Programmable Gate Arrays (FPGA) and Complex Programmable Devices (CPLD), it has been possible to design a system on a programmable chip (SOC).

A circuit to control certain Front-end System parameters, through MCM-2 (in the LO Synthesiser) in the GMRT receiver chain is developed. This report describes in detail the circuit design and implementation. An introduction to Walsh functions is given and an efficient method of Walsh pattern generation has been described. The logic for Walsh pattern generation forms the main part of the front end control circuit along with various other logics for noise pattern generation, sequency pattern generation and control of monitoring signals. The overall architecture for various patterns generation has been programmed in a CPLD using Xilinx Design Tools 8.2i, so as to develop a SOC. Additional functional features have been added to this circuit to provide more flexibility to a user while observation. Finally, the overall circuit has been implemented on a PCB using Altium Design Tools 6.1and the final PCB has been tested online. The various results obtained after testing have been compared with the calculated and the simulated results.



CHAPTER-I

INTRODUCTION

1.1. INTRODUCING GMRT

Giant Metrewave Radio Telescope (GMRT), located near Pune in India, is the world's largest radio telescope working at metre wavelengths. It is operated by the National Centre for Radio Astrophysics, a part of the Tata Institute of Fundamental Research, Mumbai. In the form of GMRT, NCRA (National Center for Radio Astrophysics, TIFR) has set up a unique facility for radio astronomical research using the metre wavelengths range of the radio spectrum.

Giant Metrewave Radio Telescope (GMRT) is located at a site about 80 KM north of Pune. It is an aperture synthesis array consisting of 30 fully steerable giant parabolic dishes of 45m diameter. GMRT is one of the most challenging experimental programmes in basic sciences undertaken by Indian scientists and engineers.

GMRT, about 10 km east of Narayangaon town on the Pune-Nasik highway, was selected after an extensive search in many parts of India, considering several important criteria such as low man-made radio noise, availability of good communication, vicinity of industrial, educational and other infrastructure and, a geographical latitude sufficiently north of the geomagnetic equator in order to have a reasonably quiet ionosphere and yet be able to observe a good part of the southern sky as well.

1.1.1. Why Metre Wavelengths?

The study of universe at high frequencies can easily be done, but the most challenging work was to develop a telescope which can work at low frequencies or more specifically Radio Frequencies because this is the range of frequencies in which maximum noise lies. Since in other countries like USA, the RF Noise level is too high, no other country had taken an initiative to develop such a system working at this frequency range. The RF noise level being comparatively low in India,

gave an opportunity to pick up this challenging project of GMRT working at Radio frequencies. Hence, the metre wavelength part of the radio spectrum has been particularly chosen for study with GMRT because man-made radio interference is considerably lower in this part of the spectrum in India. Although there are many outstanding astrophysics problems which are best studied at metre wavelengths, there has, so far, been no large facility anywhere in the world to exploit this part of the spectrum for astrophysical research.

GMRT currently operates at five observing bands centered at 150 MHz, 235 MHz, 327 MHz, 610 MHz and an L-band extending from 1000 to 1450 MHz. The Astronomical bodies that can be best studied at metre wavelengths through GMRT are – Pulsars, Sun, Jupiter Radio Bursts, Hydrogen Lines, Milky way and other nearby Galaxies.

1.1.2. Array Configuration

The number and configuration of the dishes was optimized to meet the principal astrophysical objectives which require sensitivity at high angular resolution as well as ability to image radio emission from diffuse extended regions. The GMRT has a hybrid configuration, with 14 of its antennas randomly distributed in a central region (~ 1 km across), called the central square. The distribution of antennas in the central square was deliberately "randomized" to avoid grating lobes.

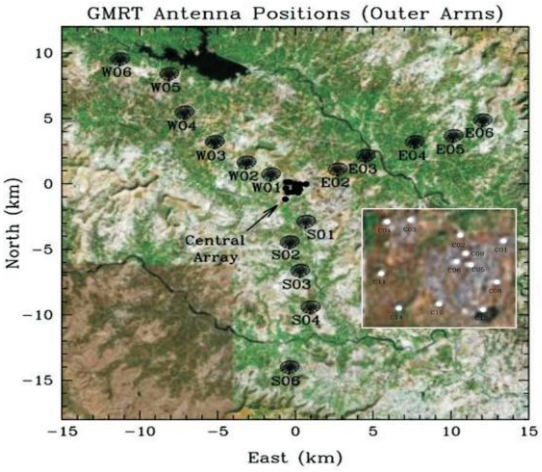


Figure 1.1-The GMRT array as viewed from space.

Figure1.2-One of the 30 GMRT antennae

The remaining antennas are distributed in a roughly Y shaped configuration, with the length of each arm 1 of the Y being ~14 km. The maximum baseline length between the extreme arm antennas is ~25 km. The arms are called the "East", "West" and "South" arms. The central square antennas provide a large number of relatively short baselines. This is very useful for imaging large extended sources, whose visibilities are concentrated near the origin of the UV

plane. The arm antennas on the other hand are useful in imaging small sources, where high angular resolution is essential. A single GMRT observation hence yields information on a variety of angular scales

.

GMRT is an indigenous project. The construction of 30 large dishes at a relatively small cost has been possible due to an important technological breakthrough achieved by Indian Scientists and Engineers in the design of light-weight, low-cost dishes. The design is based on what is being called the `**SMART**' concept - for **S**tretch **M**esh **A**ttached to **R**ope **T**russes.

1.1.3. GMRT front-ends and back-ends

Radio waves from the distant cosmic source impinge on the antenna and create a fluctuating voltage at the antenna terminals. This voltage varies at the same frequency as the cosmic electro-magnetic wave, referred to as the *Radio Frequency (RF)*. The voltage is first amplified by the front-end (or Radio Frequency) amplifier. The signal is weakest here, and hence it is very important that the amplifier introduce as little noise as possible.

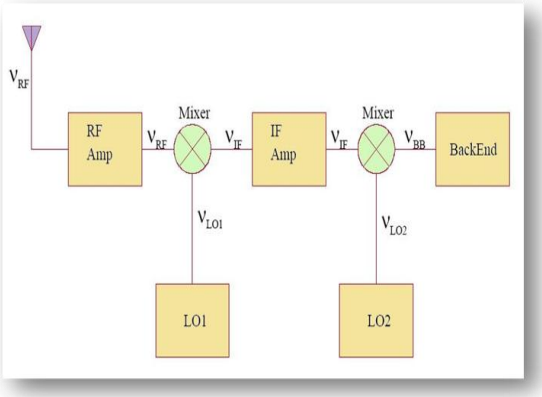


Figure 1.3-Block diagram of a single dish Radio Telescope

Front end amplifiers hence usually use low noise solid state devices. After amplification, the signal is passed into a *mixer*. A mixer is a device that changes the frequency of the input signal. Mixers have two inputs, one for the signal whose frequency is to be changed (the RF signal in this case), and the other input is usually a pure sine wave generated by a tunable signal generator, the *Local Oscillator (LO)*. The output of the mixer is at the beat frequency of the radio frequency and the local oscillator frequency. So after mixing, the signal is now at a different (and usually lower) frequency than the RF, this frequency is called the *Intermediate Frequency (IF)*. The main reason for mixing (also called heterodyning) is that though most radio telescopes operate at a wide range of radio frequencies, the processing required at each of these frequencies is identical. The economical solution is to convert each of these incoming radio frequencies to a standard IF and then to use the same back-end equipment for all possible RF frequencies that the telescope accepts. In telescopes that use co-axial cables to transport the signal across long distances, the IF frequency is also usually chosen so as to minimize transmission loss in the cable. Sometimes there

is more than one mixer along the signal path, creating a series of IF frequencies, one of which is optimum for signal transport, another which is optimum for amplification etc. This is called a *'super-heterodyne'* system. After conversion to IF, the signal is once again amplified (by the IF amplifier), and then mixed to a frequency range near 0 Hz *(the Base Band (BB))* and then fed into the *backend* for further specialized processing. What backend is used depends on the nature of the observations.

The GMRT accepts radio waves in six bands from 50 MHz to 1.4 GHz and has IFs at 130 MHz, 175 MHz and 70 MHz. The GMRT receiver chain is shown schematically in Figure 1.4.

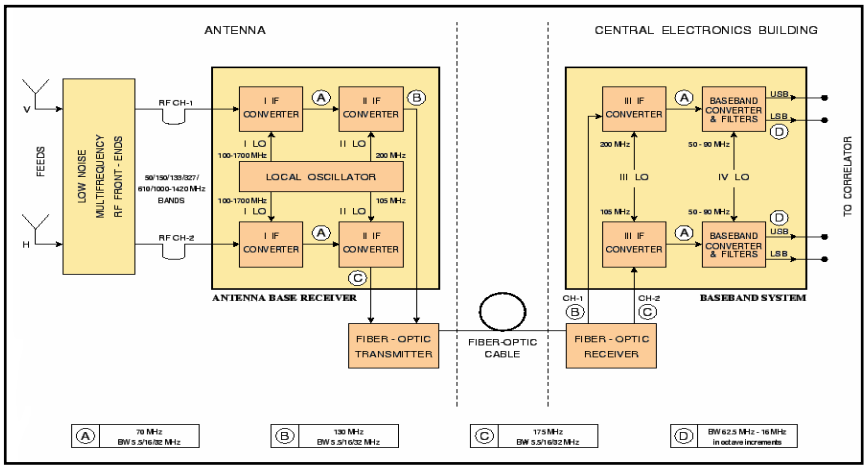
- The first block is the multi-frequency front-end. This is located in a rotating turret at the prime focus. All the feeds and low noise RF front-ends have been configured to receive dual polarization signals. Lower frequency bands (from 50 to 610 MHz) have dual circular polarization channels, i.e. left circular and right circular polarizations which have been labeled as CH1 and CH2 respectively. The L-band (1000-1450 MHz) system has dual linear polarization channels, i.e. vertical and horizontal polarizations (also labeled CH1 and CH2 respectively).

- The first local oscillator (I LO, situated at the base of the antenna, inside a shielded room) converts the RF band to an IF band centered at 70 MHz. After passing the signal through a bandpass filter of selectable bandwidth, the IF at 70 MHz is then translated (using II LO) to a second IF at 130 MHz and 175 MHz for CH1 and CH2, respectively. The maximum bandwidth available at this stage is 32 MHz for each channel.

- This frequency translation is done so that signals for both polarizations can be frequency division multiplexed onto the same fiber for transmission to the Central Electronics Building (CEB). The IF signal at 130 MHz and 175 MHz along with telemetry and LO round trip phase carriers directly modulate a laser diode operating at 1300nm antennas and the CEB.

- At the CEB, these signals are received at 'nm' wavelengths which are coupled to a single mode fiber-optic link by the Fiber-Optic Receiver and the 130 and 175 MHz signals are then separated out and sent for base band conversion. The baseband converter section converts the 130 and 175 MHz IF signals first to 70 MHz IF (using III LO), these are

- then converted to upper and lower sidebands (each at most 16 MHz wide) at 0 MHz using a tunable IV LO. There are also two Automatic Level Controllers (ALCs) in the receiver chain.



1.4- Schematic Diagram of GMRT Receiver chain

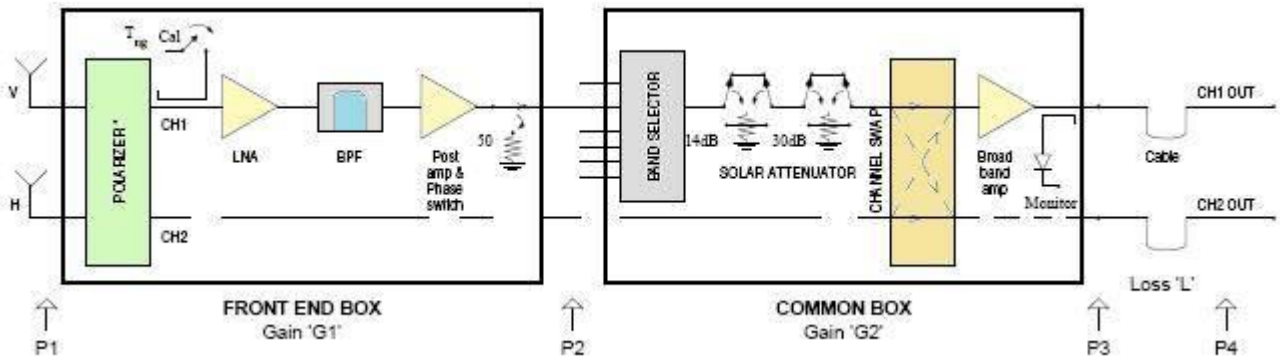
- There are a variety of digital backends available at the GMRT. The principal backend used for interferometric observations is a 32 MHz wide FX correlator. The FX correlator produces a maximum of 256 spectral channels for each of two polarizations for each baseline.

- All control and monitoring commands to the system are sent from the central computer through a telemetry system. The telemetry system provides a Measurement, Control and Monitor (MCM) circuit for sending digital control data to the circuits and for acquiring analog monitor data.

1.2. EXISTING FRONT- END ELECTRONICS

- At the focus of each antenna, each feed has two low noise amplifiers (one for each polarization), with a noise injection facility where the user can select any of the 4 values of injected noise power. The two signals go to a common box (also on the feed turret) where the user can select which frequency signals appear at the output of the common box since only 2 cables go down to the antenna base. The common box has facilities for the user to select solar attenuators (0, 14, 30 or 44 dB) and swap the two polarization channels.

- There is a bandpass filter after the LNA in the front end box, which is followed by a post amplifier, to have the required output power at the end of front end box. Thereafter the signal is modulated with Walsh function using phase switching to minimize the effect of crosstalk between different signals. Mini-circuits double balanced mixer SBL-1MH is used for phase-switching.



2.5- Schematic Diagram of GMRT Front End System

- At the correlator the exact reverse phase switching is done for each antenna so that the original phase is recovered just before the cross correlation is done. Such a scheme can greatly reduce the cross-talk at all points between the RF amplifier and the baseband.

1.3. FRONT-END CONTROL FROM LO SYNTHESISER

All the front end parameters are controlled through a MCM-5 in the front end. Few of the parameters of the front end system are varied at the time of observation of a celestial source. All these front end parameters along with switching ON/OFF of the front end MCM are controlled at the Antenna Base Rack in the Local Oscillator Synthesiser. Before analyzing the requirements for the front end control of the GMRT, it is necessary to understand what is noise calibration and Walsh switching and why are these required for the GMRT.

1.3.1. Noise Calibration

An additional noise signal (of known strength) is injected (as described above) into the RF signal to measure the system temperature. The noise generator at the front-end of each antenna can be switched in both power and Time Domains for different durations as per the requirement. The advantages of noise injection facility are as follows:

- This facility is used in order to calibrate the gain of the receiver. By adding noise of known strength, SNR is decreased be a known amount and the variation in gain of the system can be measured. For example, the front end noise generator is to be kept on continuously for duration in order to troubleshooting the receiver system. This is however, not used for online observation purposes.

- It may be used a secondary calibration system, the primary being the standard celestial sources. By synchronously measuring the total power, it is possible to calibrate the system temperature.

The synchronous total power measurement however has not yet been implemented.

1.3.2. Walsh Switching

Signals from one antenna could leak into another antenna at various points along the signal flow chain. This is normally referred to as *cross-talk*. This would cause a spurious correlation between the baseband signals from these two antennas. This leakage can be minimized by switching the phase of the RF signal of each antenna by a pattern that is ortho-normal to the pattern used for all other antennas. Typically the ortho-normal functions used are Walsh functions, and this scheme is called *Walsh Switching*.

1.3.3. Requirements for the Front End System from the LO Synthesiser:

- Generation and selection of noise patterns for noise switching.
- Generation and selection of Walsh patterns for Walsh switching at the front end.
- Switching ON and OFF the front end MCM.

Switching of Front-end Noise Generator in Time Domain:

The Noise switching in power domain is achieved by switching appropriate attenuators and the switching in time domain by a switching pattern in time, which is produced at base of the antenna in the Antenna Base Rack (ABR) and will be described later. It is possible to inject noise at any one of four levels. Four patterns are available, NGN1, NGN2, NGN3, NGN4 corresponding to 0%, 25%, 50%, and 100% ON period for the noise generator. The noise at any antenna can be switched on and off (on sub second time scales) according to a pre-determined pattern, which is encoded in PROMs in the Antenna Based Receiver (ABR).

Walsh switching at the front end:

A phase switching facility using separate Walsh functions for each signal path is available at the RF section of the receiver. As discussed previously, a set of Walsh patterns are produced at the base of each antenna, for the following two purposes:

- To phase-modulate the RF output of the LNA in the front-end using a double balanced mixer as a broad band phase switch. This is to take care of coupling between the two signal channels of an antenna as well as coupling of signal between antennas. The demodulation will be done at an appropriate location in the correlator.

- To phase-modulate the return LO referance signal. As phase-coherent LO reference signals at 105 Mhz are received at CEB through the return link from all thirty antennas, it is felt that there might be coupling between signals at CEB to result in inaccuracies of RTP measurement. Hence phase switching of the LO return signal might be essential. However, this scheme has not been implemented as yet.

128 Walsh patterns with 128 transitions over a period of the function, defined as sequency, have been generated using a program. The sequency pattern uniquely defines the start of Walsh pattern. So, there are 64 pairs of Walsh patterns for use in 30 antennas (for each of the two channels of every antenna) divided into 'Low Group' and 'High Group'. The first and the last patterns are not used. The required Walsh patterns for each antenna are also encoded in PROMs situated at the ABR.

Switching ON and OFF the Front end MCM

MCM-5 is placed in the front end and is to be switched off at the time of observation. This is done because an MCM consist of microprocessor, digital circuitry which generates a lot of RFI (Radio Frequency Interference) and since this MCM is in the front end, near the feeds, the feeds can pick up the RFI and thus resulting in undesirable results. Again the required signal for controlling the MCM ON/OFF is controlled at the ABR.

1.3.4. Control at the ABR

The Monitor and Control Module-2 (MCM-2) in the control digital Plug-In-Unit, D-49 PIU which is placed at the ABR in the LO Synthesiser, controls the selection of various patterns at the ABR which will be combined with the RF signals at the front-end and the control of front end MCM, as per the online commands issued by the user in the control room. A Walsh card

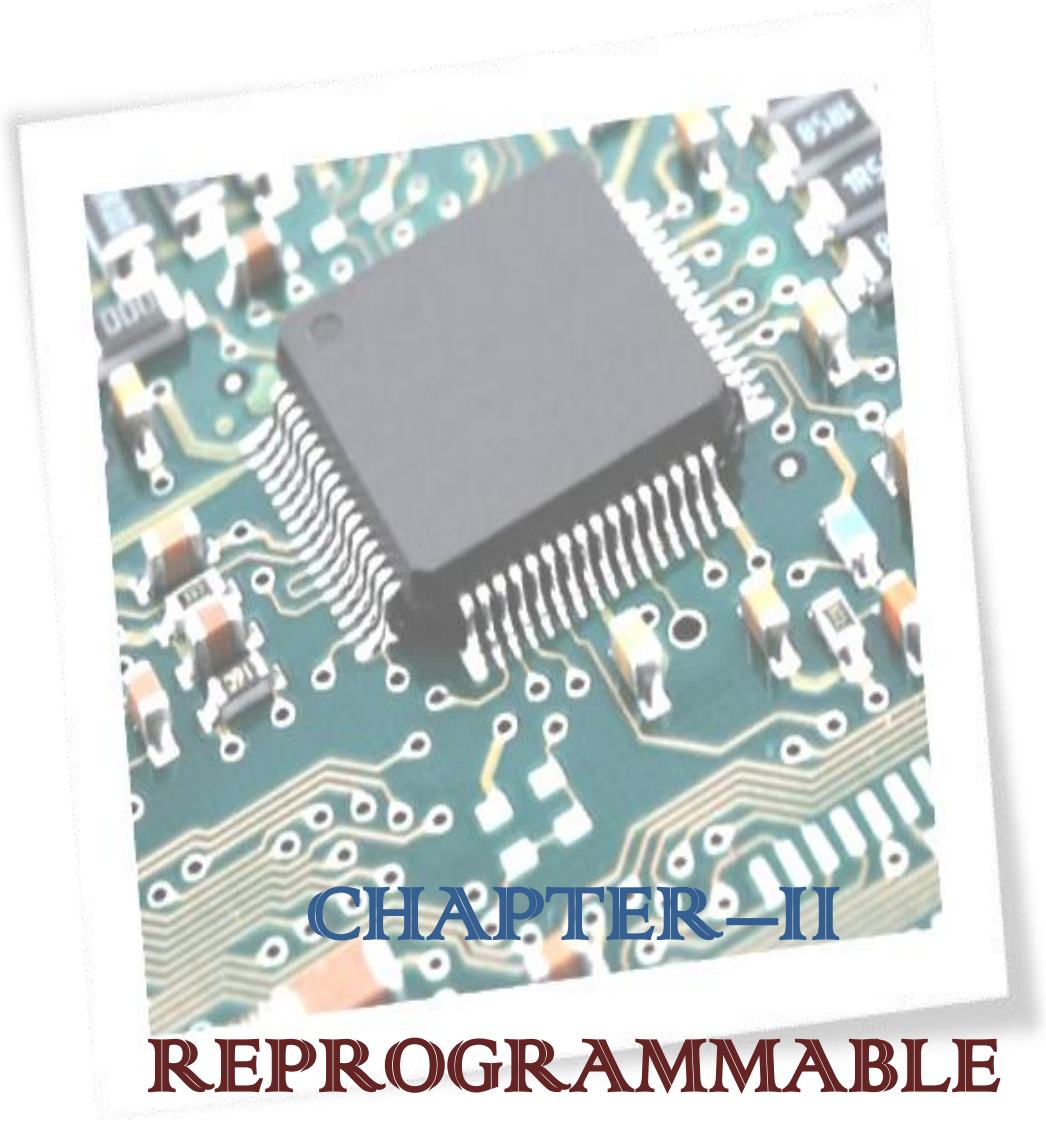
PCB WNG_RO in the PIU is used for this purpose and consists of EPROM which is encoded with the various patterns this PCB also generates the various monitoring signals for enabling the Monitoring card PCB in the D-49 PIU. The complete process of pattern selection has been discussed later. An interface panel with a 10-way terminal strip has been provided at the rear of the ABR rack to bring the patterns from rear of D-49 PIU to a convenient location for connecting to the 10-core front-end cable.

1.4. PROBLEM DEFINITION

The main aim of the project is to make Front End control to control Front End system parameters through MCM-2 in the ABR. This involves improving the Walsh card PCB in the D-49 PIU. This PCB consists of digital circuitry having a number of IC's with differential line driver circuitry for long distance transmission of signals from the antenna base to the front end. All this Digital circuitry part can be simulated in the CPLD and thus saving the cost and providing better performance in comparison to the EPROM based circuitry of the existing Walsh card. Hence, the new Walsh card consists of the CPLD and differential line driver circuitry with minimum Components and thus size. Besides reducing the size, additional functionality can be added to the card by programming the CPLD in order to improve the overall performance and provide more flexibility to the user for better analysis while observation.

Overview of the work that has to be done:

- Understanding the working and function of the Walsh Card PCB.
- Understanding the Walsh functions and finding the most efficient method for generation of Walsh Patterns for the GMRT.
- Developing the logic for Walsh pattern generation, Sequency pattern generation, noise pattern generation, front end MCM control and Monitoring signals control.
- Selection of a suitable CPLD.
- Physical realization of the various logics in the CPLD.
- Preparing a PCB with this CPLD and other circuitry for implementing the complete functionality of the Walsh card PCB.
- Testing the PCB online.



CHAPTER–II

REPROGRAMMABLE SYSTEMS

2.1. REPROGRAMMABLE SYSTEMS

Reconfigurable Computing has been described as hardware going soft. A computing platform based on this principle has an architecture that can be modified by the software to suit the application at hand. An algorithm will always run fastest when directly hardwired (DSP, ASIC). Dramatic performance gains can therefore be obtained by implementing an algorithm in the reconfigurable portion of such a system. Prompted by the development of new types of sophisticated field-programmable devices (FPDs), the process of designing digital hardware has changed dramatically over the past few years. Unlike previous generations of technology, in which board-level designs included large numbers of SSI chips containing basic gates, virtually every digital design produced today consists mostly of high-density devices. This applies not only to custom devices like processors and memory, but also for logic circuits such as state machine controllers, counters, registers, and decoders. When such circuits are destined for high-volume systems they have been integrated into high-density gate arrays. The most compelling advantages of FPDs are instant manufacturing turnaround, low start-up costs, low financial risk and (since programming is done by the end user) ease of design changes.

The market for FPDs has grown dramatically over the past decade to the point where there is now a wide assortment of devices to choose from. A designer today faces a daunting task to research the different types of chips, understand what they can best be used for, choose a particular manufacturer's product, learn the intricacies of vendor-specific software and then design the hardware. Confusion for designers is exacerbated by not only the sheer number of FPDs available, but also by the complexity of the more sophisticated devices.

Field-Programmable Device (FPD) is a general term that refers to any type of integrated circuit used for implementing digital hardware, where the chip can be configured by the end user to realize different designs. Programming of such a device often involves placing the chip into a special programming unit, but some chips can also be configured "in-system". Another name for FPDs is *programmable logic devices* (PLDs).

2.1.1. Types of Programmable Logic Devices/FPDs

Many types of programmable logic are available. The current range of offerings includes everything from small devices capable of implementing only a handful of logic equations to huge CPLDs that can hold an entire processor core (plus peripherals). In addition to this incredible difference in size there is also much variation in architecture. Some of the most common types of programmable logic devices have been mentioned as under:

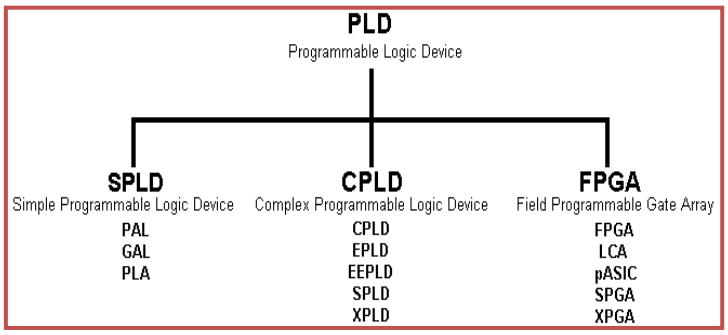


Figure 2.1- Hierarchy of various PLDs

PLDs

At the low end of the spectrum are the original Programmable Logic Devices (PLDs). These were the first chips that could be used to implement a flexible digital logic design in hardware. In other words, you could remove a couple of the 7400-series TTL parts (ANDs, ORs, and NOTs) from your board and replace them with a single PLD. Other names for this class of device are Programmable Logic Array (PLA), Programmable Array Logic (PAL), and Generic Array Logic (GAL).

PLDs are often used for address decoding, where they have several clear advantages over the 7400-series TTL parts that they replaced. First, of course, is that one chip requires less board area, power, and wiring than several do. Another advantage is that the design inside the chip is flexible, so a change in the logic doesn't require any rewiring of the board. Rather, the decoding logic can be altered by simply replacing that one PLD with another part that has been programmed with the new design.

Inside each PLD is a set of fully connected macrocells. These macrocells are typically comprised of some amount of combinatorial logic (AND and OR gates, for example) and a flip-flop. In other words, a small Boolean logic equation can be built within each macrocell. This equation will combine the state of some number of binary inputs into a binary output and, if necessary, store that output in the flip-flop until the next clock edge. Of course, the particulars of the available logic gates and flip-flops are specific to each manufacturer and product family. But the general idea is always the same.

CPLDs

As chip densities increased, it was natural for the PLD manufacturers to evolve their products into larger (logically, but not necessarily physically) parts called *Complex Programmable Logic*

Devices (CPLDs). For most practical purposes, CPLDs can be thought of as multiple PLDs (plus some programmable interconnect) in a single chip. The larger size of a CPLD allows the implementation of either more logic equations or a more complicated design. In fact, these chips are large enough to replace dozens of 7400-series parts. When a CPLD is configured, the internal circuitry is connected in a way that creates a hardware implementation of the software application. Unlike processors, CPLDs use dedicated hardware for processing logic and do not have an operating system. CPLDs are truly parallel in nature so different processing operations do not have to compete for the same resources. As a result, the performance of one part of the application is not affected when additional processing is added. Also, multiple control loops can run on a single CPLD device at different rates. CPLD-based control systems can enforce critical interlock logic and can be designed to prevent I/O forcing by an operator. However, unlike hard-wired printed circuit board (PCB) designs which have fixed hardware resources, CPLD-based systems can literally rewire their internal circuitry to allow reconfiguration after the control system is deployed to the field.

CPLDs store their logic design in an EPROM, EEPROM, flash or SRAM memory that associates each programmable connection point with a memory cell (is the connection open or closed). The building block of a CPLD is the macro cell, which contains logic implementing disjunctive normal form expressions and more specialized logic operations. CPLD– Produced by Altera, AMD, Lattice, Xilinx and *etc*

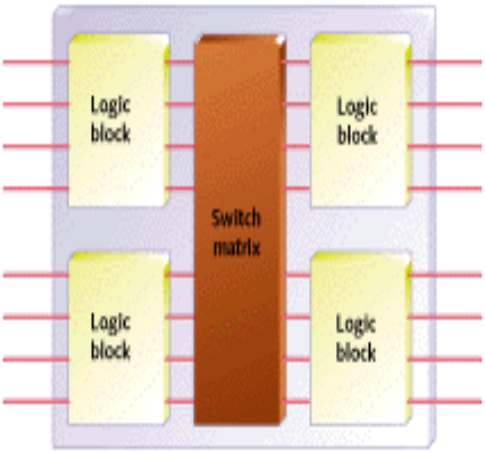


Figure 2.2- Internal structure of a CPLD

2.1.2. Architecture of CPLD

As shown in above figure, there are logic blocks which are themselves comprised of macrocells and interconnect wiring, just like an ordinary PLD. The macrocells within a function block are usually fully connected. If a device contains multiple function blocks, then the function blocks are further interconnected. In concept, CPLDs consist of multiple PAL-like logic blocks interconnected together via a programmable switch matrix. Typically, each logic block contains 4 to 16 macrocells, depending on the architecture.

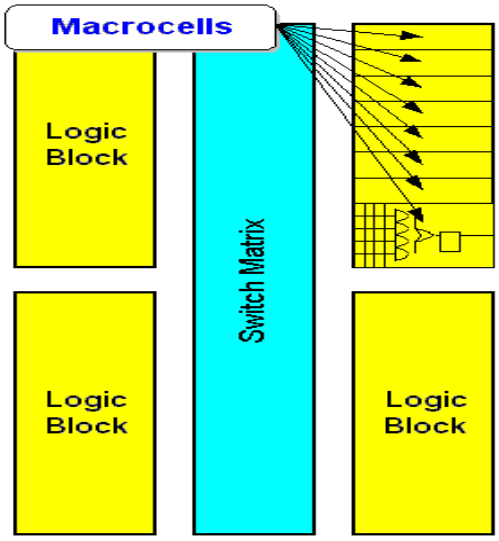


Figure 2.3- Arrangement of Macro Cells within a logic block

Some of the major variations between CPLD architectures include the number of product terms per macrocell, whether product terms from one macrocell can be borrowed or allocated to another macrocell, and whether the interconnect switch matrix is fully- or partially-populated. In some architectures, when the number of product terms required exceeds the number available in the macrocell, additional product terms are borrowed from an adjoining macrocell. This makes the CPLD device useful for a wider variety of applications.

When borrowing product terms from an adjoining macrocell, that macrocell may no longer be useful. In some architectures, the macrocell still has some basic functionality. Borrowed product terms usually mean increased propagation delay.

Another difference in architectures is the number of connections within the switch matrix. A switch matrix supporting all possible connections is fully populated. A partially- populated switch supports most, but not all, connections. The number of connections within the switch matrix determines how easy a design will fit in a given device. With a fully-populated switch matrix, a design will route even with a majority of the device resources used and with fixed I/O pin assignment. Generally, the delays within a fully populated switch matrix are fixed and predictable. At the high-end (in terms of numbers of gates), there is also a lot of overlap in potential applications with CPLDs. Because of its less flexible internal architecture, the delay through a CPLD (measured in nanoseconds) is more predictable and usually shorter. In short, we can conclude the following points about CPLDs:

- **Significant characteristics for the CPLD-architecture:**
  - » product terms generated in programmable macrocells.
  - » typically one dedicated flip-flop per macrocell.
  - » many macrocells per logic-block.
  - » typically all logic-blocks identical.
  - » minimum two logic-blocks per device.
  - » routing between logic-blocks via global switch matrix.

- **Main-advantages :**
  - » predictable timing
  - » fast pin-to-pin delay
  - » efficient resource utilization by switch-matrix
  - » medium design complexities possible

Programming a CPLD

In order to program a CPLD, solder the device to its printed circuit board, and then feed it with a serial data stream from a personal computer. The CPLD is in-circuit programmable and contains a circuit that decodes the data stream and configures the CPLD to perform its specified logic function via a JTAG interface or from an on-board embedded processor. This makes it possible to erase and reprogram the device internals.

FPGAs

FPGAs use a grid of logic gates, similar to that of an ordinary gate array, but the programming is done by the customer, not by the manufacturer. The term "field-programmable" means the array is done outside the factory, or "in the field." FPGAs are usually programmed after being soldered down to the circuit board, in a manner similar to that of larger CPLDs. In most larger FPGAs the configuration is volatile, and must be re-loaded into the device whenever power is applied or different functionality is required.

The main difference between FPGAs and CPLDs is that FPGAs have a volatile memory, thus it requires to be programmed after power up. CPLDs do not. Also, FPGAs usually consume more power than CPLDs due to their SRAM nature. Finally, CPLDs do not have as many registers or memory storage as FPGAs. In general CPLDs are a good choice for wide combinatorial logic applications while FPGAs are more suitable for large state machines (i.e. microprocessors). FPGAs are internally based on Look-up tables (LUTs) while CPLDs form the logic functions with sea-of-gates (e.g. sum of products).

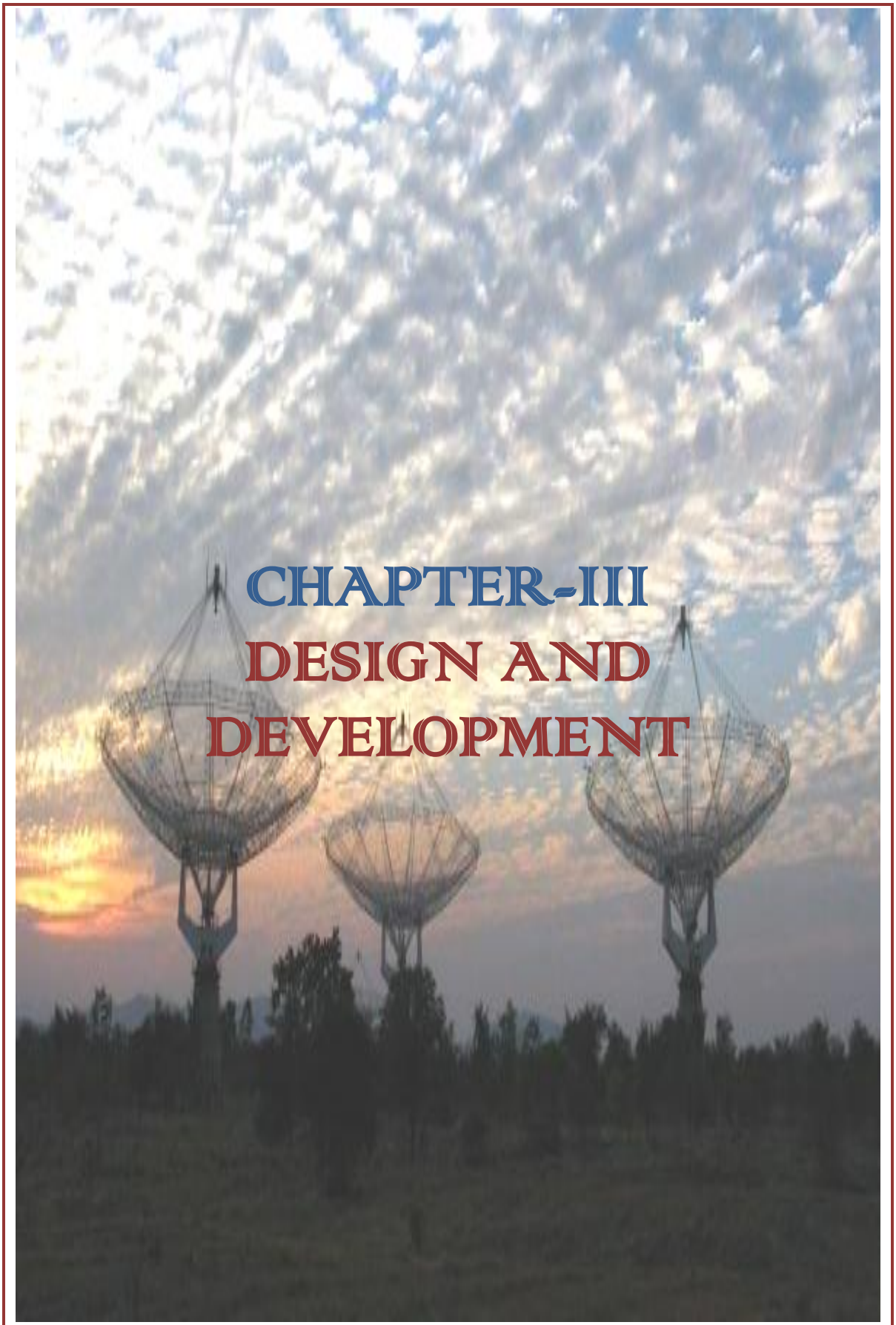
2.1.3. CPLD vs. EPROM

Before PLDs were invented, read-only memory (ROM) chips were used to create arbitrary combinational logic functions of a number of inputs. Consider a ROM with m inputs (the address lines) and n outputs (the data lines). When used as a memory, the ROM contains 2^m words of n bits each. Now imagine that the inputs are driven not by an m-bit address, but by m independent

logic signals. Theoretically, there are 2^m possible Boolean functions of these m signals, but the structure of the ROM allows just 2^n of these functions to be produced at the output pins. The ROM therefore becomes equivalent to n separate logic circuits, each of which generates a chosen function of the m inputs.

The advantage of using a ROM in this way is that any conceivable function of the m inputs can be made to appear at any of the n outputs, making this the most general-purpose combinatorial logic device available. Also, PROMs (programmable ROMs), EPROMs (ultraviolet-erasable PROMs) and EEPROMs (electrically erasable PROMs) are available that can be programmed using a standard PROM programmer without requiring specialized hardware or software. However, there are several disadvantages:

- They are usually much slower than dedicated logic circuits,
- They cannot necessarily provide safe "covers" for asynchronous logic transitions so the PROM's outputs may glitch as the inputs switch,
- They consume more power, and
- Because only a small fraction of their capacity is used in any one application, they often make an inefficient use of space.
- PROM and EPROM can only be encoded with the help of a separate piece of lab equipment called a device programmer. On the other hand, CPLDs based on EEPROM or Flash technology are in-circuit programmable. In other words, the additional circuitry that's required to perform device (re)programming is provided within the FPGA or CPLD silicon as well. This makes it possible to erase and reprogram the device internals via a JTAG interface or from an on-board embedded processor.



CHAPTER-III
DESIGN AND DEVELOPMENT

3.1. WALSH FUNCTIONS

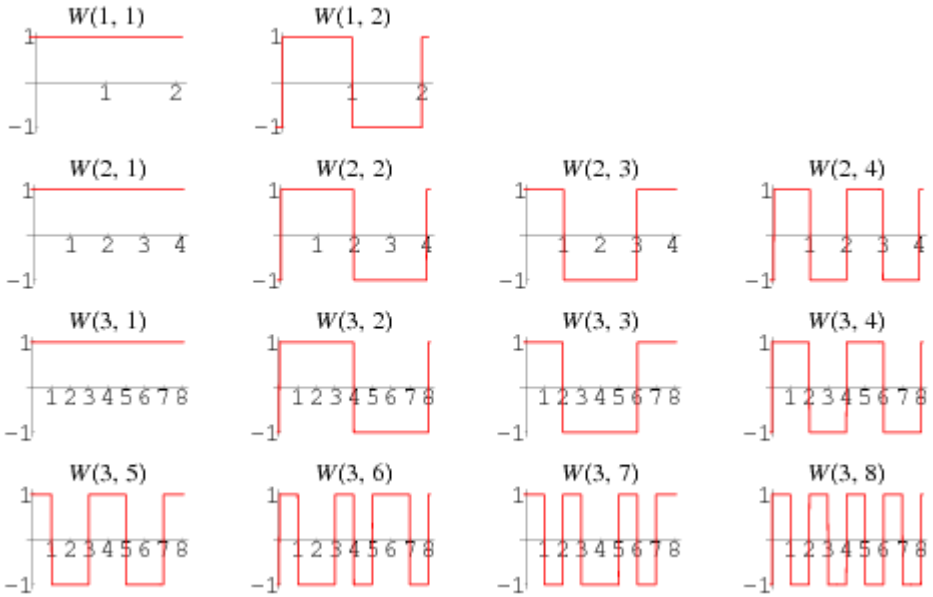


Figure 3.1-walsh patterns

Walsh functions are a series of square waves that can be combined to create almost any waveform. These functions consist of trains of square pulses (with the allowed states being -1 and 1) such that transitions may only occur at fixed intervals of a unit time step; the initial state is always +1. The function *WAL(i,k)* represents a waveform, known as *Walsh Pattren*, as a function of time *k* with *i* transitions over the period of the function defined as *Sequency function*. Thus, the parameter n can be interpreted as "one half the number of zero crossings per unit time. *i* is known as the *normalized sequency* or *sequency*. For an index *n*, $i=log_2 n$. There are 2^n Walsh functions of length 2^n. Furthermore, within the set of 2^n functions there is one function of zero sequency, one of (normalized) sequency 2^{n-1}, and one pair (odd and even) of each (normalized) sequency from *1* to $2^{n-1} - 1$. The Walsh functions are commonly subdivided into the even functions *Cal (i,k)*, and the odd functions *Sal (i,k)* which are defined by:

$$Cal\ (i,k) = wal\ (2i,k)$$

$$Sal\ (i,k) = wal\ (2i-1,k)$$

3.1.1. Applications of Walsh Functions

Walsh functions and transforms are important analytical tools for signal processing and have wide applications in digital communication, digital image processing, statistical analysis, solving differential equations as well as in digital logic design. The Walsh function series can be applied

to many areas where sinusoidal techniques have previously dominated. This is so in the design of digital equipment for communication and computer applications, where the two levels that form the function match binary logic. In the applications of the Walsh transform to CDMA, VLSI testing, cryptography, and digital signal and image processing, the fastest approach is the hardware implementation.

Applications of Walsh functions in Radio Astronomy Instruments

An effective means of getting a further reduction of several tens of decibels in the unwanted responses is known as phase switching. In case of phase switching, the RF signal received at one of the antennas is periodically reversed in phase by a switch that is driven by a square wave a few tens of Hertz. The component of the multiplier outputs resulting from the signals entering the antennas reverses in sign following the action of the switch, and is rectified by synchronous detectors. The unwanted signals, which are not phase switched, produce components which are essentially constant at the multiplier outputs and thus are eliminated by the synchronous detection.

For phase switching a multielement array in which the products of the signals from all possible pairs of antennas are to be formed, phase switching can be represented by multiplication of the received signals by periodic functions that alternate in time between values of +1and -1. For the m_{th} and n_{th} antennas let these functions be $f_m(t)$ and $f_n(t)$. Synchronous detection of the multiplier output for these two antennas requires a reference waveform $f_m(t)f_n(t)$, and any nonvarying components from the multiplier are reduced by a factor

$$\frac{1}{T}\int_0^T f_m(t)f_n(t)$$

after averaging for a time T. This factor will be zero if $f_m(t)$ and $f_n(t)$ are orthogonal over the interval T or a submultiple of it.

3.1.2. Definition and Properties

Many different definitions for Walsh functions are known and used for various applications. These functions are commonly defined in terms of a subset, the Rademacher functions.

- A Rademacher function of the nth order is defined as:

$$R_m(k) = Sgn(\sin 2\pi 2^m k), \quad m=0,1,2\ldots \qquad (2.1)$$

Where, the signum function Sgn(y) is defined by:

$$Sgn(y) = \begin{cases} +1, y \geq 0 \\ -1, y < 0 \end{cases} \quad (2.2)$$

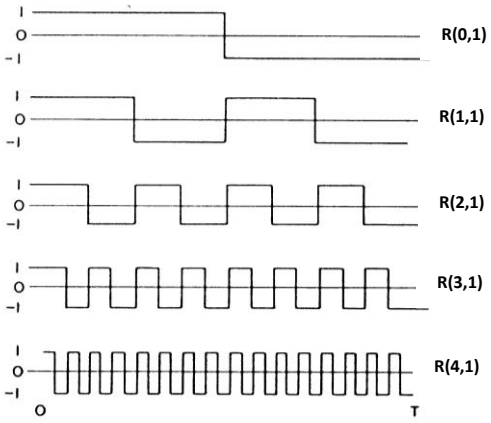


Figure 3.2- Rademacher-functions R_0 to R_4

Property 3.1:

Rademacher functions form an orthogonal, but incomplete set. Actually, they form a subset of the Walsh set of functions. The Rademacher functions $R_m(k)$ are the walsh functions of index 2^m-1

$$R_0(k) = wal(2^0\text{-}1, k) = sgn(\sin 2\pi k)$$

$$R_1(k) = wal(2^2\text{-}1, k) = sgn(\sin 4\pi k)$$

And in general,

$$R_m(k) = wal(2^{m+1}\text{-}1, k) = sgn(\sin 2^{m+1}\pi k), \; k \geq 0. \quad (2.3)$$

First Walsh pattern i.e. *w(0,k)* is a series of all 1's and 0 sequency.

Property 3.2:

The product of any two Walsh functions is a third Walsh function

$$wal(i, k) \cdot wal(j, k) = wal(p, k) \quad (2.4)$$

such that

$$\langle i \oplus j \rangle = \langle p \rangle \quad (2.5)$$

Where (.) means that if *i, j, p* are expressed as binary numbers, then *p* is formed from the bit-by-bit modulo-2 sum of *i* and *j*. Equation (2.3) may be written alternatively as

$$\langle i \oplus j \oplus p \rangle = \langle 0 \rangle$$

For convenience and simplicity of notation the Walsh functions can be considered to be formed from (0, 1) values rather than from (+ 1, - 1) values; multiplication of the functions is then replaced by modulo-2 addition, the (0, 1) values x_i being related to the (+1, -1) values y_i by the equation

$$y_i = 1 - 2x_i$$

wal (i, k) will for briefness be written *w(i)* and $R_m(k)$ will be written R_m.

Property 3.3:

An arbitrary Walsh function *w(i)* may be written as a linear combination of Walsh functions all with index of the form 2^m . For if the binary equivalent of *i* is $b_z b_{z-1} \ldots b_3 b_2 b_1 b_0$, (with b_0 as the least significant digit) then

$$i = (b_0 2^0 \oplus b_1 2^1 \oplus b_2 2^2 \oplus \ldots\ldots)$$

$$= \sum_{m=0}^{z} b_m 2^m \qquad (2.6)$$

where the summation symbol denotes the modulo-2 sum. Therefore,

$$w(i) = \sum_{m=0}^{z} b_m w(2^m), \qquad b_m \in \{0,1\} \qquad (2.7)$$

This is evident from (2.4) and (2.5) and shows that an arbitrary Walsh function *w(i)* may be written as a linear combination of Walsh functions with indices of the form 2^m, and that the combination required is obtained directly from the binary equivalent of *i*.

For example, if *i* = 50 (= 110010 in binary)

$$w(50) = w(2^5) \oplus w(2^4) \oplus w(2^1)$$

$$= w(32) \oplus w(16) \oplus w(2)$$

$$= w(32) \oplus w(13)$$

Property 3.4:

Any Walsh function may also be expressed as a linear combination of the Rademacher functions:

$$w(i) = \sum_{m=0}^{z} g_m w(2^{m+1}\text{-}1) = \sum_{m=0}^{z} g_m R_m , \qquad g_m \in \{0,1\} \qquad (2.8)$$

The g_m forms the Gray code equivalent of *i*.

3.1.3. Generation of Walsh Patterns

Many different definitions for Walsh functions are known and used for various applications. They include: Walsh functions in strict sequency ordering (known also as Walsh - Kaczmarz) related to Sal and Cal symmetries; Walsh functions in dyadic Paley ordering (known also as Dyadic) known also as Gray code ordered functions; Walsh functions in natural Hadamard ordering; Walsh functions in X-ordering; and Walsh functions in reverse Gray ordering. Since they are square waves, they are very easy to create. Various Walsh function generators exist that use different methods. The sequence generators having the widest applicability are those generating a set of Walsh functions, although in some cases these functions are obtained by first generating Rademacher functions. Ideally, the generated functions should be orthogonal to each other and some designs are better in achieving this than others. For the second case, the Walsh orderings form a group under modulo-2 addition and such generators can be easily implemented using multiplicative EXOR gates.

For our project, we have adopted the method to generate the Walsh patterns in strict sequency ordering directly from the primary set of Rademacher functions. This method has been adopted as it is simpler and requires lesser hardware resources as compared to other methods reviewed. This method generates all the i Walsh functions at the same time, where i is the index of Walsh functions. As already stated, the sequency of a Walsh function is defined as the number of zero crossings in one cycle. In strict sequency order, each row has one more crossings between 1's and -1's than the row above. Thus, alternate even and odd patterns i.e. Cal-Sal patterns are obtained in this ordering. The various steps and algorithm for generation have been discussed as under.

Algorithm for Walsh pattern generation

In the algorithm for Walsh functions generation, the original 1's are kept, but -1's are replaced by 0, and all the Walsh functions are generated in the complete interval between 0 and 1 rather than between -½ and ½. After the changing of original symbols the basic properties like a modulo-2 addition of two Walsh functions yields another Walsh function hold for all the Walsh functions. To generate, we use the properties described above.

Step-I: Let $\vec{B} = (b_{n-1}, b_{n-2}, \ldots\ldots, b_j, \ldots\ldots, b_0)$ be a binary vector representing an index of the Walsh function (i.e. the sequency), where $0 \leq j \leq n-1$.

Step-II: obtain the Gray code $\vec{G} = (g_{n-1}, g_{n-2}, \ldots\ldots, g_j, \ldots\ldots, g_0)$ from the natural binary code

$\vec{B} = (b_{n-1}, b_{n-2}, \ldots\ldots, b_j, \ldots\ldots, b_0)$, where $g_j = b_{j+1} \oplus b_j$ for $0 \le j \le n\text{-}2$, and $g_{n-1} = b_{n-1}$.

Step-III: Generate the Walsh functions in strict sequency ordering as follows (i is the index of walsh patterns):

- Some of the Walsh patterns (say WR patterns) can be obtained directly from the Rademacher functions as

$$WR(2^{i+1}\text{-}1, k) = R_i(k) \quad \textit{(using property 2.1)}$$
$$W(0,k) = WR(0,k) = +1, \qquad \textit{for all } k$$

- Remaining Walsh functions are obtained from these WR patterns as follows:

$$w(i) = \sum_{j=0}^{n-1} g_j \, WR(2^{j+1}\text{-}1) = \sum_{j=0}^{n-1} g_j R_j, \quad \textit{(using property 2.4)}$$

- While obtaining the patterns using the above equation, reduce any combinations of 3 or more Walsh functions (wherever possible) to combinations of two Walsh functions which have already been obtained in the sequence of patterns. *(Using property 2.2 and 2.3)*
- The Cal patterns are even functions having even sequency and Sal patterns are odd functions having odd sequency. Every even indexed ($\vec{B}$ is even) walsh pattern i.e. the Cal pattern is to be complemented.

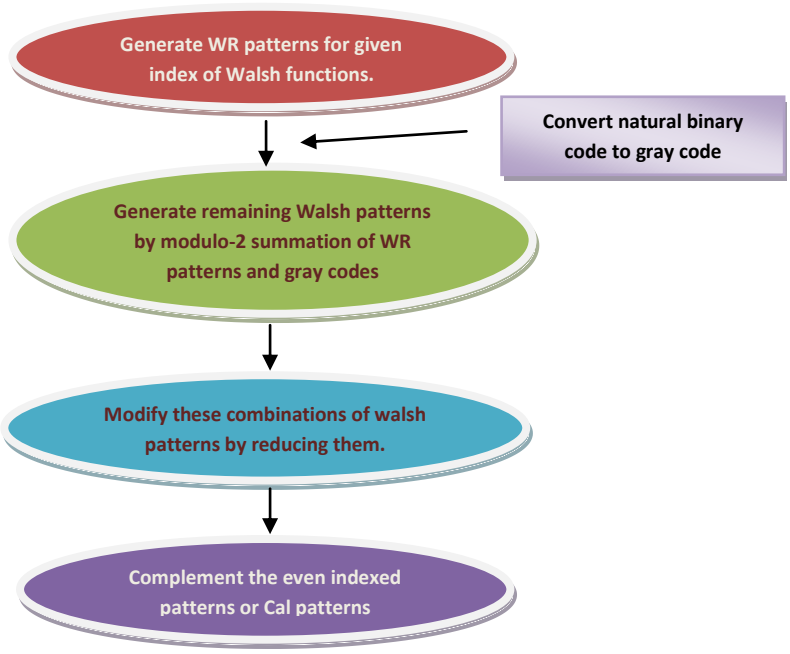


Figure 3.3-flow graph for Walsh patterns generation

3.2. SELECTION OF WALSH PATTERNS AND NOISE PATTERNS IN D-49 PIU:

In the following sections, the existing design for pattern selection in D-49 PIU has been highlighted and then the new design has been discussed.

3.2.1. Existing Design

The 128 Walsh patterns have been divided into two groups as *High Group* and *Low Group*. The control signals required for enabling/disabling Walsh Patterns, selection of *High Group* or *Low Group* and Noise ON/OFF are generated in the control PIU D-49 in the ABR at the base of each antenna . The Digital Plug-In-Unit (PIU), D-49 of the Local Oscillator (LO) Synthesiser in the ABR is the control PIU and contains the following PCBs:

- The MCM PCB, which is the interface to the LO systems.
- The Reset PCB RESET_R1.
- The Control PCB CON_R2.
- The Walsh and Noise Generator control PCB WNG_RO with ID Code D 85, used in the Monitoring scheme of the synthesizer as well as controlling a few parameters of the front-end at the antenna.
- The Power supply PCB.
- The Monitor multiplexer PCB, MON32_R0.
- The front panel LED indicator PCB, LED49_R0.

A Block Diagram of the control PIU-D49 is shown in the following figure.

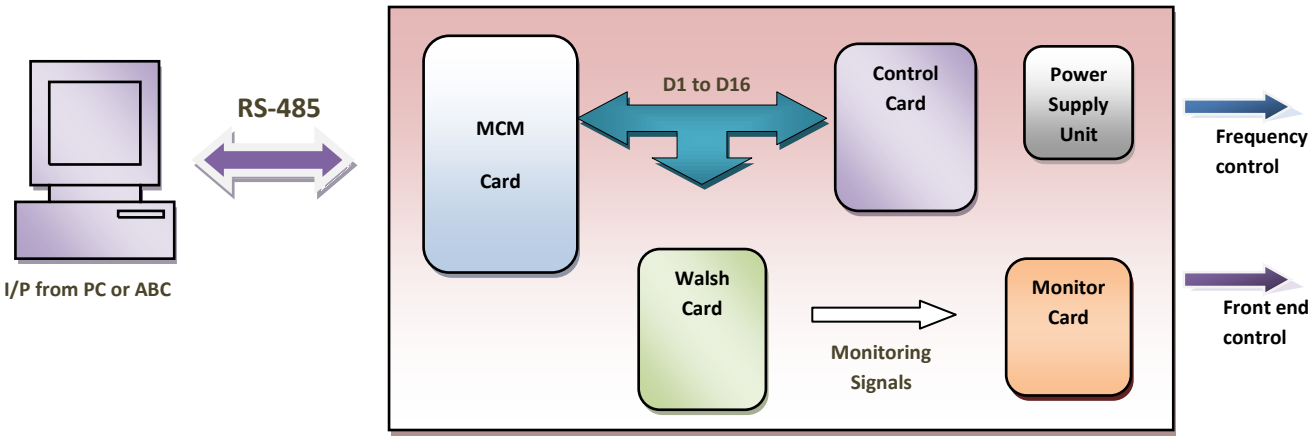
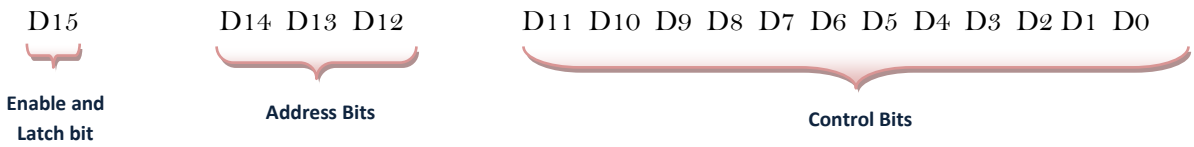


Figure 3.4- Block Diagram of D-49 PIU

MCMs are general purpose microcontroller based cards which provide 16 TTL digital control outputs and can monitor up to 64 input analog signals. These MCMs are the interface to all the settable GMRT subsystems, like the front-ends, the LOs, the attenuators, etc. In detail, at each antenna, MCM 5 is the interface to the front end system, while MCMs 2, 3, and 9 are the interface to the LO and IF systems. Input to MCM card is provided from PC or ABC through a RS-485 line. These inputs are the various commands issued by a user for setting the parameters of the electronics and for monitoring a range of parameters. The 16 bit digital data from MCM card is divided into three parts as under:



The address bits for address are used to obtain, in principle a total of 8×12=96 independent control bits. Also, the Enable and Latch bit helps in latching the data so that any Address Group can be set without worrying about the control bits already set through another address group.

- Address groups 0, 1, 2 and 4 are used for the control of GMRT synthesizer to set the required frequency. The controls under these groups are controlled using the Control card PCB.
- Address group 6 contains control bits used for monitoring. These controls are operated by the Walsh card PCB.
- Address group 7 contains control bits used for setting a few of the front-end parameters related to Walsh switching. The controls under this group are controlled by the Walsh card PCB.
- Address groups 3 and 5 are not used.

The digital data from MCM is available on 20 pin FRC Connector and then sent to control card PCB CON_R2 and Walsh and noise generation PCB WNG_RO. Depending on the address group selected (as stated above), Control card PCB or the Walsh card PCB comes into operation and gives the required outputs as per the control bits selected. Thus WNG_RO PCB comes into operation by group 6 and group 7 selections and is used for generating monitoring outputs and selection of noise ON/OFF or Walsh enable/disable respectively. The block diagram of the existing Walsh Card WNG_RO is as follows:

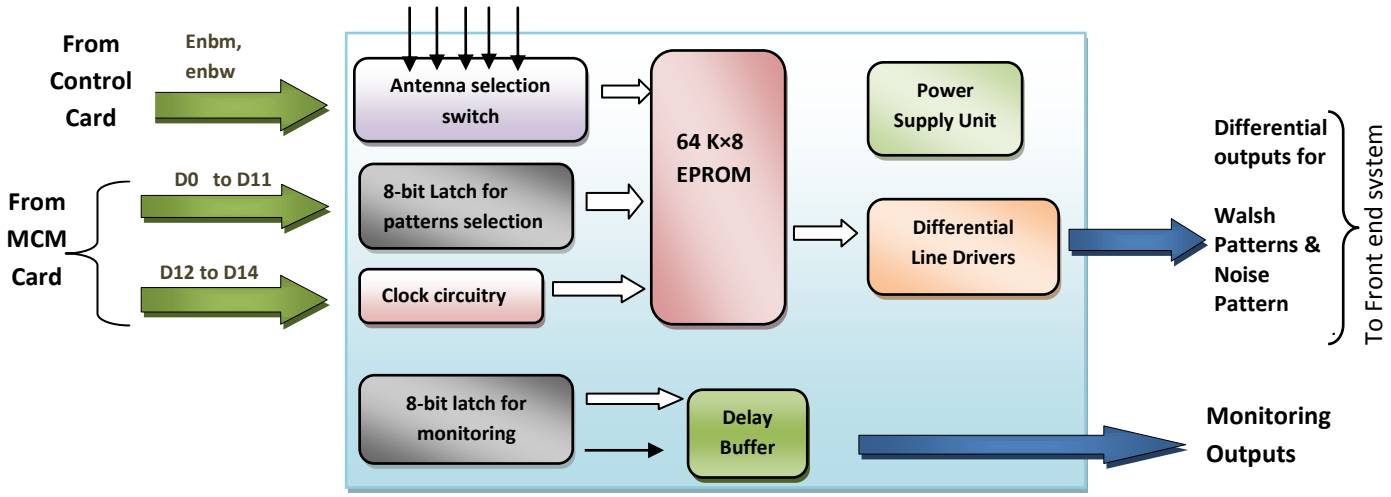


Figure 3.5- Block Diagram of Walsh Card

3.2.2. New Design and Development

As per our project, we aim to use a CPLD based circuit for the functions under the group-7 and control of functions under group-6. Thus the CPLD has to be programmed with logic for these operations and to be placed in the Walsh card circuit along with few other components and digital IC's. In this section, we discuss the architecture and the logic synthesis and simulations for generating and selecting the various patterns on the new Walsh card. The main differences between the previous design and the new design are as follows:

- The heart of the new Walsh card circuit is a CPLD which contains the main logic for various pattern generation and selection instead of EPROM. The various advantages of using CPLD over EPROM have been discussed in the beginning.

- In case of the new Walsh card, we are generating a desired pattern based on the selection inputs at that particular instant, instead of storing them in advance.

- The new Walsh card can be used for generating only the Cal patterns or the Sal patterns, in comparison to the existing circuit which uses both the Cal and the Sal patterns.

- Additional functionality can easily be added by reprogramming the CPLD at any time in future to provide more flexibility to the users. Some of these additional modifications will be discussed in later sections.

The advantages of using a CPLD based Walsh card circuit are as follows:

- *Consist of a programmable Walsh pattern generator:*

  Instead of storing all the 128 Walsh patterns, Sequency pattern and noise pattern in advance, we now generate the desired pattern dynamically as per the requirements of the user. This will reduce the requirement of the resources for the operation.

- *Independent channel selection facility for taking the Walsh patterns output on any of the channels of an antenna:*

  The user can now have the provision to select independent channels of a particular antenna and add Walsh patterns on these channels as follows:

  - To add Walsh pattern on one of the channels, i.e., Channel (CH1) 1 or Channel 2 (CH2) while disabling Walsh pattern on the other.
  - To add different Walsh patterns on both the channels from Group-1 or Group-2. For CH-1 or CH-2, respectively, at the same time.
  - To add the CH-1 Walsh pattern on CH-2.
  - To add the CH-2 Walsh pattern on CH-2.

- *Variable time period selection for all the patterns:*

  The complete 128 bits pattern for Walsh function, Sequency and the pattern for noise may be needed for certain time duration depending on the need of the user and the system. The new circuit thus gives the facility to select variable time period for each of the bits of these patterns. The user can select either a 4ms, 8ms, 16ms or 32ms duration time period each bit of every pattern.

- *Independent time period of noise patterns and Walsh patterns:*

  The length and time period of noise patterns and 128-bits Walsh patterns, sequency patterns is independent of each other, i.e., the time period of noise pattern can be varied while keeping the time period for Walsh pattern and sequency pattern constant and vice-versa.

- *Only CAL patterns selection:*

  In order to remove the phase ambiguity between Cal and Sal patterns, only Cal patterns have been used for the antennae.

The new circuit consists of various blocks replacing the components in the previous Walsh card circuits which have been described above. The blocks are as follows:

- CPLD with core logic for various pattern generation and monitoring outputs.
- Differential line driver circuitry

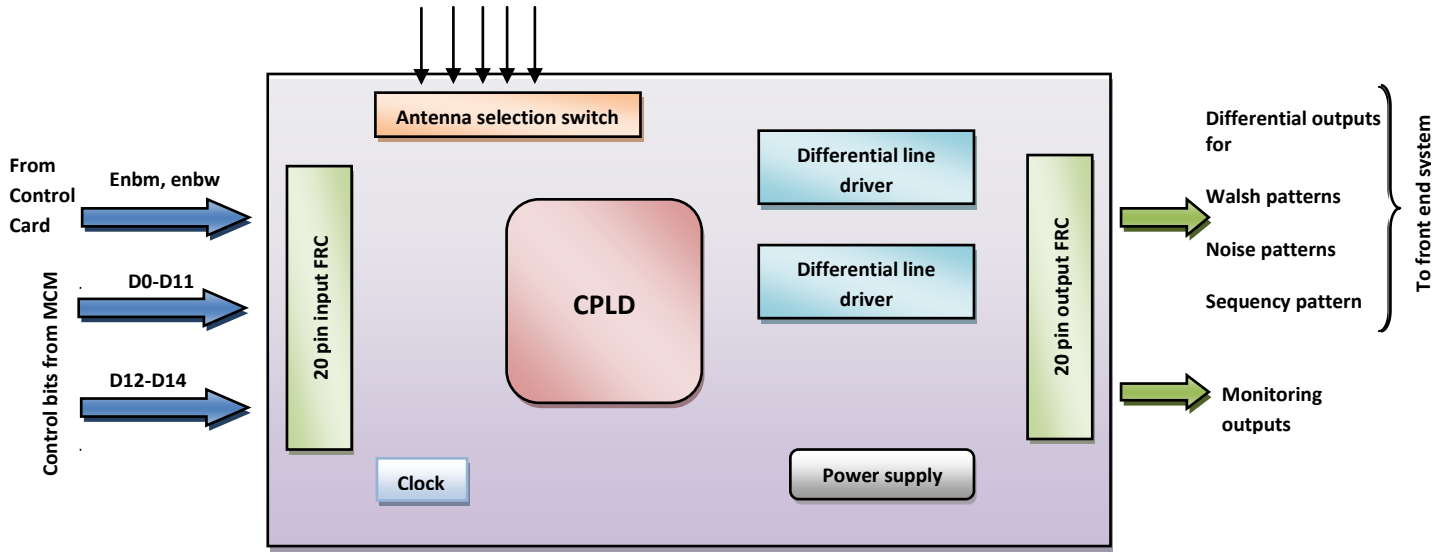


Figure 3.6- Block diagram of new Walsh card

Working of the circuit

❖ The 16 bit control bits D0-D14 are input to the Walsh card from the MCM card and the two latch enable bits enbw and enbm from the Control card, through the 20 pin input FRC. 'enbw' is the enable bit for the group 7 and 'enbm' is the enable bit for group 6. Both these bits are generated by a 3:8 decoder in the control card from D15-D12 bits coming from the MCM card.

❖ *Antenna selection switch:* A DIP-5 switch is used for selecting the antenna to be operated.

❖ *CPLD* is dedicated for pattern outputs and monitoring outputs. It consists of:

- *Walsh latch:* 10 bit latch used for latching the control bits D0 to D9. The remaining 2 bits D11-D12 are not used. It holds the control bits for clock frequency selection, Walsh pattern selection, channel selection, noise pattern selection, switching ON and OFF the front end MCM, and at the output; we obtain control bits C0-C9. It is enabled when group 7 has been selected after a low to high transition on 'enbw'. The output bit C4 is used for MCM ON/OFF.

- *Monitoring latch*: 6 bit latch for latching the control bits D0 to D5 for selecting monitoring signals and is enabled when group 6 has been selected after a low to high transition on 'enbm'. The latched out bits are the monitoring outputs.

- *Delay Buffer:* A Delay buffer is used for delaying one of the monitoring outputs from the 6-bit monitoring latch.

- *Clock Circuitry:* This is required for synchronizing all the components for pattern generation. Clock circuitry comprises of 1.024 MHz crystal oscillator and two 3 stage binary counters. Frequency division is required for various patterns to be ON/OFF for a fixed duration of time in order to synchronize with the system. Thus, an internal clock frequency division by 2^{12} is needed which is further divided variably from 2^1 to $2^{3.}$ The variable frequency division is selected using C2-C3 for noise generator and C8-C9 for Walsh generator. For F_{in} = 1.024 MHz

  $F_1 = F_{21} = (1.024 \times 10^6) / 4096$
  $= 250$ Hz $\qquad T_1 = T_{21} = 4$ ms

  $F_{22} = 250/2$
  $= 125$ Hz $\qquad T_{21} = 8$ ms

  $F_{23} = 250/4$
  $= 62.5$ Hz $\qquad T_{22} = 16$ ms

  $F_{24} = 250/2$
  $= 31.25$ Hz $\qquad T_{23} = 32$ ms

- *Walsh pattern generator:* 128 patterns can be generated of length 128 bits each for both the channels of the 30 antennae and these are selected by combination of antenna selection bits. Clock input is from the binary counter for Walsh generator. The control bits C5-C7 are used for pattern selection on each of the channels. Each pattern is a 128 bit pattern.

- *Sequency pattern generator:* 128 bit pattern is generated for indicating start and stop of the Walsh pattern. It is to be synchronized with the Walsh patterns and hence same clock input is used as the Walsh pattern.

- *Noise generator:* NGN pattern is obtained at its output by combination of C0-C1. The clock input is from the binary counter for noise generator. It is independent of Walsh and Sequency patterns and can be switched ON/OFF at any time by changing the control bits combination.

❖ *Differential line drivers:* The patterns thus produced are processed in differential line drivers MC 3487, so that long lines to the front-end can be driven. Additional outputs of the pattern and Sequency are provided for retransmission to CEB through the LOR return link electronics. Differential line driver are required for transmitting the signals over long distance, such as from the antenna base to the front end. For each input, there are two differential outputs, where one is positive and the other one is its reverse. The Walsh patterns, Sequency pattern, noise pattern, MCM on/ off signal from the CPLD are sent through two differential line drivers giving a total of 8 signals at the output. At the frontend there is a differential line receiver IC MC3486 which recombines the differential input and then further processing is performed.

The following diagram shows the structure of the CPLD. Each block and its logic for implementation have been discussed in the following section.

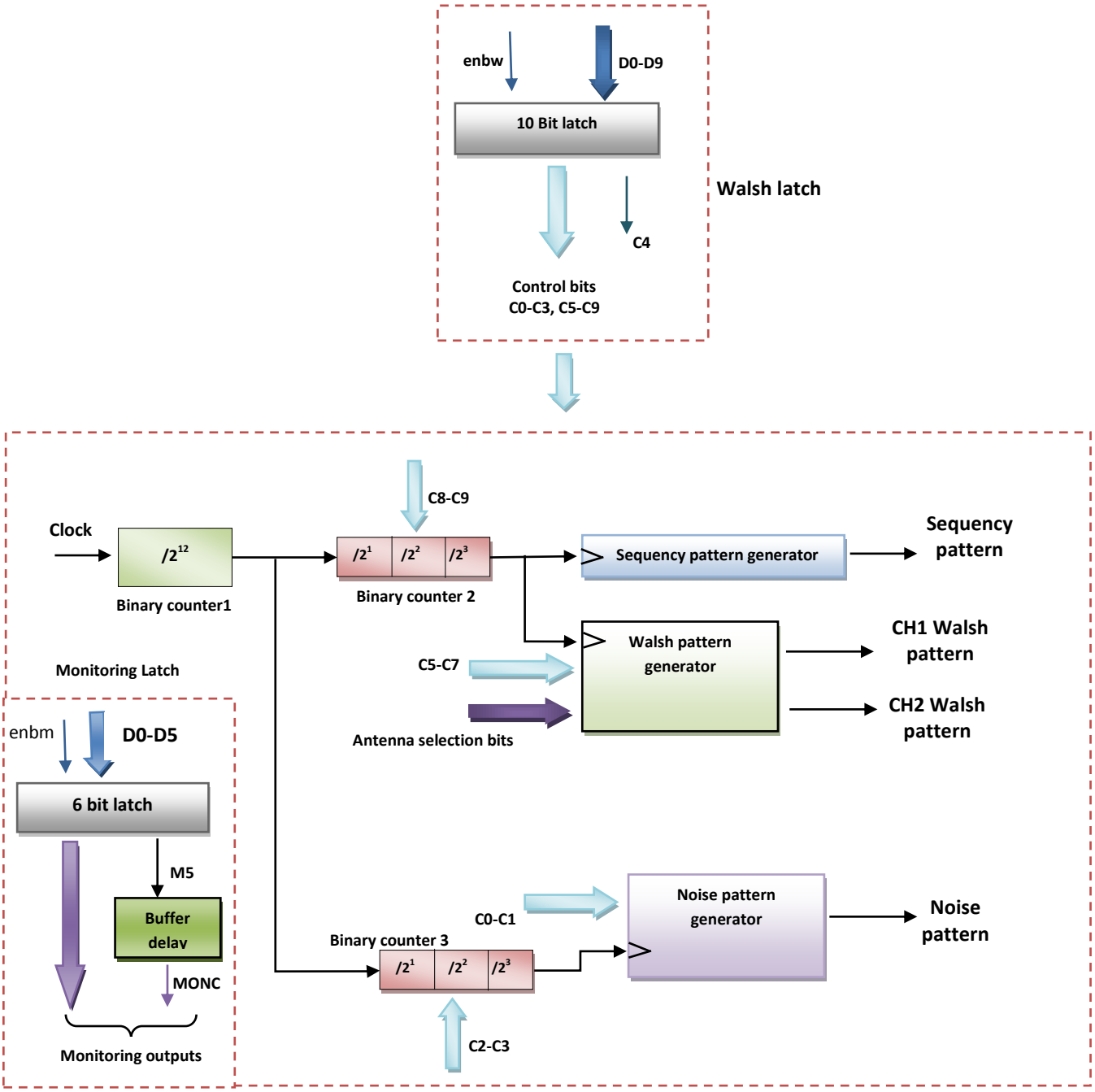


Figure 3.7-structure for pattern generation circuitry (programmed in CPLD)

Architecture for Clock circuitry

The clock circuitry consists of 3 binary counters as follows:

- One of the counters divides the incoming clock signal frequency from the clock oscillator by 2^{12} (i.e. 4096). The divided clock signal now becomes the driving clock signal for rest of the circuit operation. The remaining two counters are 3 stage counters (counts from 0 to 127) used for dividing the above clock signal further by 2^1 or 2^2 or 2^3 or none.
- Two 4:1 multiplexers are to be used for selecting any one of the clock signals for walsh generator and noise generator, respectively.
- The select lines are the control bits C2-C3 for varying the clock frequency of the noise generator and C8-C9 are the select lines for varying the clock frequency of the Walsh and Sequency generator.

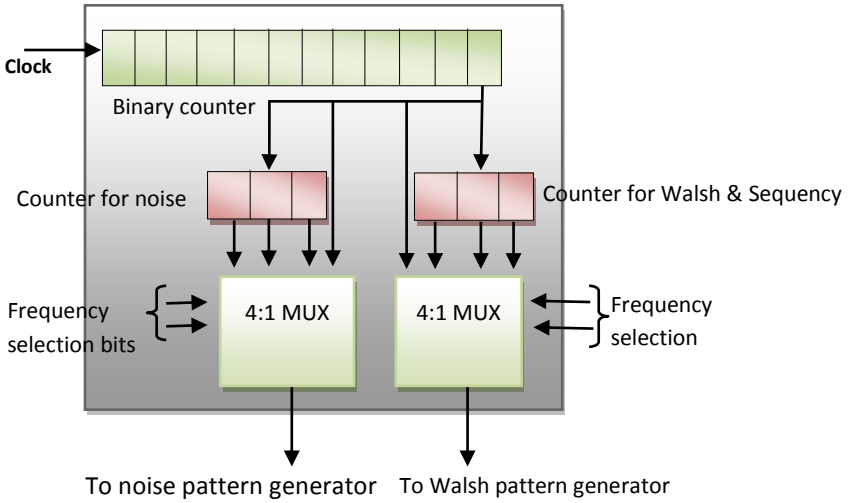


Figure 3.8-block diagram for clock circuitry

Table 3.1- Truth Table for Clock Circuitry

Frequency selection bits for Noise pattern generator		Frequency selection bits for Walsh pattern generator		Selected Clock frequency ($F_{in}/2^n$)	Time period
D3	**D2**	**D9**	**D8**	**n**	**($1/F_{in}$)**
0	0	0	0	0	4 ms
0	1	0	1	1	8 ms
1	0	1	0	2	16 ms
1	1	1	1	3	32 ms

*F_{in}=1.024 Mhz

Architecture and Logic for Walsh Pattern Generator

We have discussed earlier that 128 Walsh patterns must be generated for each of the channels of 30 GMRT Antennae, where first and the last Walsh pattern in the series will not be used i.e. patterns *w(0)* and *w(127)* are not used. Thus, the index for Walsh patterns is 128. Thus, there are 63 pairs of Cal-Sal patterns for Walsh switching. But, we have decided to use only the Cal patterns for Walsh switching and hence the new circuit generates 62 Cal patterns at the output. The first 31 Cal patterns are grouped into Group-1 and the next 31 Cal patterns are grouped into Group-2. Out of these only 60 patterns are required for each of the 2 channels of the 30 antennae and so we actually require 60 Cal patterns from cal(1) to cal(60) which is w(2) to w(120), respectively (cal(0) is w(0)). The logic for Walsh function generator is based on the algorithm discussed in previous section. By implementing this algorithm, we can generate all the 128 patterns of 128 bit length, simultaneously and keep only 62 of these patterns. But at the output, we need only one pattern for each channel of the antenna which is obtained using a multiplexer. We obtain single bit-by-bit output at every clock cycle. The various blocks of the generator are as follows:

- *7 T Flip-Flops with positive edge triggering which form a binary counter to count from 0 to 127*. At every low-high transition of the clock, each of the flip flops output bit is obtained forming Walsh-Rademacher patterns. After 127^{th} counts, the counter is again reset to **0.**

- *XOR Gates* for performing modulo-2 summation between Rademacher patterns and other Walsh patterns. This forms the combinational part of the circuitry.

- *Two 31:1 multiplexers* for selecting Walsh patterns on individual channels. The select lines for these multiplexers are the antenna selection bits through Antenna selection switch.

- Combination of antenna selection bits is used as the input for selecting a corresponding Walsh pattern for that particular antenna.

- Control bits are used for channel selection for the selected antenna. C5-C7 used for channel selection.

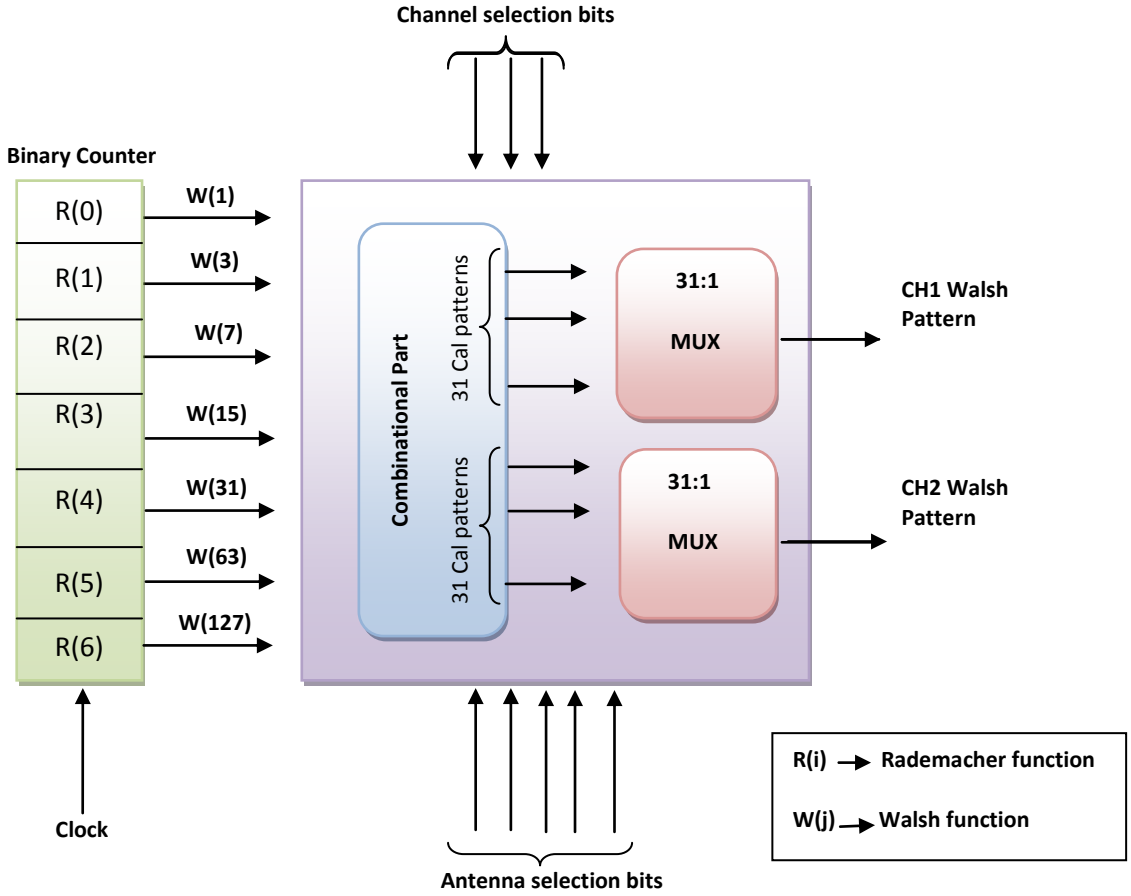


Figure 3.9-block diagram representing the architecture of Walsh pattern generator

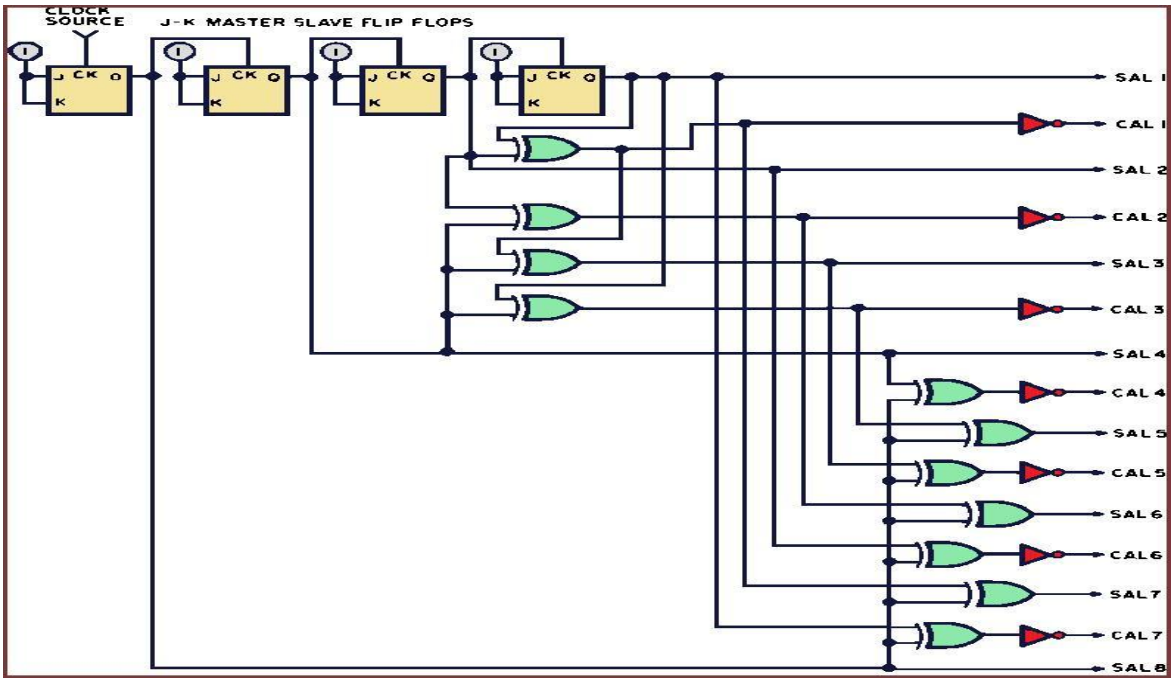


Figure 3.10-circuit diagram for generating 16 Walsh patterns

Figure 3.10 shows the circuit diagram for generation of set of 16 Walsh patterns. The same circuit has been used to generate 128 Walsh patterns by adding XOR gates and T- Flip Flops.

Table 3.2-Truth Table for Walsh Generator

Antenna selection Bits					Channel selection bits			Channel	
DS0	DS1	DS2	DS3	DS4	D7	D6	D5	CH1	CH2
X	X	X	X	x	0	0	0	WALSH OFF	WALSH OFF
0	0	0	0	0	0	0	1	CAL 1	WALSH OFF
					0	1	0	WALSH OFF	CAL 32
					0	1	1	CAL 1	CAL 32
					1	0	0	CAL1	CAL1
					1	0	1	CAL32	CAL32
0	0	0	0	1	0	0	1	CAL 2	WALSH OFF
					0	1	0	WALSH OFF	CAL 33
					0	1	1	CAL 2	CAL 33
					1	0	0	CAL2	CAL2
					1	0	1	CAL33	CAL33
0	0	0	1	0	0	0	1	CAL 3	WALSH OFF
					0	1	0	WALSH OFF	CAL 34
					0	1	1	CAL 3	CAL 3
					1	0	0	CAL3	CAL3
					1	0	1	CAL34	CAL34
0	0	0	1	1	0	0	1	CAL 4	WALSH OFF
					0	1	0	WALSH OFF	CAL 35
					0	1	1	CAL 4	CAL 35
					1	0	0	CAL4	CAL4
					1	0	1	CAL35	CAL35
0	0	1	0	0	0	0	1	CAL 5	WALSH OFF
					0	1	0	WALSH OFF	CAL 36
					0	1	1	CAL 5	CAL 36
					1	0	0	CAL5	CAL5
					1	0	1	CAL36	CAL36
0	0	1	0	1	0	0	1	CAL 6	WALSH OFF
					0	1	0	WALSH OFF	CAL 37
					0	1	1	CAL 6	CAL 37
					1	0	0	CAL6	CAL6
					1	0	1	CAL37	CAL37
0	0	1	1	0	0	0	1	CAL 7	WALSH OFF
					0	1	0	WALSH OFF	CAL 38
					0	1	1	CAL 7	CAL 38
					1	0	0	CAL7	CAL7
					1	0	1	CAL38	CAL38
0	0	1	1	1	0	0	1	CAL 8	WALSH OFF
					0	1	0	WALSH OFF	CAL 39
					0	1	1	CAL 8	CAL 39
					1	0	0	CAL8	CAL8
					1	0	1	CAL39	CAL39
0	1	0	0	0	0	0	1	CAL 9	WALSH OFF

					0	1	0	WALSH OFF	CAL 40
					0	1	1	CAL 9	CAL 40
					1	0	0	CAL9	CAL9
					1	0	1	CAL40	CAL40
0	1	0	0	1	0	0	1	CAL 10	WALSH OFF
					0	1	0	WALSH OFF	CAL 41
					0	1	1	CAL 10	CAL 41
					1	0	0	CAL10	CAL10
					1	0	1	CAL41	CAL41
0	1	0	1	0	0	0	1	CAL 11	WALSH OFF
					0	1	0	WALSH OFF	CAL 42
					0	1	1	CAL 11	CAL 42
					1	0	0	CAL11	CAL11
					1	0	1	CAL42	CAL42
0	1	0	1	1	0	0	1	CAL 12	WALSH OFF
					0	1	0	WALSH OFF	CAL 43
					0	1	1	CAL 12	CAL 43
					1	0	0	CAL12	CAL12
					1	0	1	CAL43	CAL43
0	1	1	0	0	0	0	1	CAL 13	WALSH OFF
					0	1	0	WALSH OFF	CAL 44
					0	1	1	CAL 13	CAL 44
					1	0	0	CAL13	CAL13
					1	0	1	CAL44	CAL44
0	1	1	0	1	0	0	1	CAL 14	WALSH OFF
					0	1	0	WALSH OFF	CAL 45
					0	1	1	CAL 14	CAL 45
					1	0	0	CAL14	CAL14
					1	0	1	CAL45	CAL45
0	1	1	1	0	0	0	1	CAL 15	WALSH OFF
					0	1	0	WALSH OFF	CAL 46
					0	1	1	CAL 15	CAL 46
					1	0	0	CAL15	CAL15
					1	0	1	CAL46	CAL46
0	1	1	1	1	0	0	1	CAL 16	WALSH OFF
					0	1	0	WALSH OFF	CAL 47
					0	1	1	CAL 16	CAL 47
					1	0	0	CAL16	CAL16
					1	0	1	CAL47	CAL47
1	0	0	0	0	0	0	1	CAL 17	WALSH OFF
					0	1	0	WALSH OFF	CAL 48
					0	1	1	CAL 17	CAL 48
					1	0	0	CAL17	CAL17
					1	0	1	CAL48	CAL48
1	0	0	0	1	0	0	1	CAL 18	WALSH OFF
					0	1	0	WALSH OFF	CAL 49
					0	1	1	CAL 18	CAL 49
					1	0	0	CAL18	CAL18
					1	0	1	CAL49	CAL49
1	0	0	1	0	0	0	1	CAL 19	WALSH OFF
					0	1	0	WALSH OFF	CAL 50
					0	1	1	CAL 19	CAL 50
					1	0	0	CAL19	CAL19

					1	0	1	CAL50	CAL50
1	0	0	1	1	0	0	1	CAL 20	WALSH OFF
					0	1	0	WALSH OFF	CAL 51
					0	1	1	CAL 20	CAL 51
					1	0	0	CAL20	CAL20
					1	0	1	CAL51	CAL51
1	0	1	0	0	0	0	1	CAL 21	WALSH OFF
					0	1	0	WALSH OFF	CAL 52
					0	1	1	CAL 21	CAL 21
					1	0	0	CAL21	CAL52
					1	0	1	CAL52	CAL52
1	0	1	0	1	0	0	1	CAL 22	WALSH OFF
					0	1	0	WALSH OFF	CAL 53
					0	1	1	CAL 22	CAL 53
					1	0	0	CAL22	CAL22
					1	0	1	CAL53	CAL53
1	0	1	1	0	0	0	1	CAL 23	WALSH OFF
					0	1	0	WALSH OFF	CAL 54
					0	1	1	CAL 23	CAL 23
					1	0	0	CAL23	CAL54
					1	0	1	CAL54	CAL54
1	0	1	1	1	0	0	1	CAL 24	WALSH OFF
					0	1	0	WALSH OFF	CAL 55
					0	1	1	CAL 24	CAL 55
					1	0	0	CAL24	CAL24
					1	0	1	CAL55	CAL55
1	1	0	0	0	0	0	1	CAL 25	WALSH OFF
					0	1	0	WALSH OFF	CAL 56
					0	1	1	CAL 25	CAL 56
					1	0	0	CAL25	CAL25
					1	0	1	CAL56	CAL56
1	1	0	0	1	0	0	1	CAL 26	WALSH OFF
					0	1	0	WALSH OFF	CAL 57
					0	1	1	CAL 26	CAL 57
					1	0	0	CAL26	CAL26
					1	0	1	CAL57	CAL57
1	1	0	1	0	0	0	1	CAL 27	WALSH OFF
					0	1	0	WALSH OFF	CAL 58
					0	1	1	CAL 27	CAL 58
					1	0	0	CAL27	CAL26
					1	0	1	CAL58	CAL58
1	1	0	1	1	0	0	1	CAL 28	WALSH OFF
					0	1	0	WALSH OFF	CAL 59
					0	1	1	CAL 28	CAL 59
					1	0	0	CAL28	CAL28
					1	0	1	CAL59	CAL59
1	1	1	0	0	0	0	1	CAL 29	WALSH OFF
					0	1	0	WALSH OFF	CAL 60
					0	1	1	CAL 29	CAL 60
					1	0	0	CAL29	CAL29
					1	0	1	CAL60	CAL60
1	1	1	0	1	0	0	1	CAL 30	WALSH OFF
					0	1	0	WALSH OFF	CAL 61

					0	1	1	CAL 30	CAL 61
					1	0	0	CAL30	CAL30
					1	0	1	CAL61	CAL61
1	1	1	1	0	0	0	1	CAL 31	WALSH OFF
					0	1	0	WALSH OFF	CAL 62
					0	1	1	CAL 31	CAL 62
					1	0	0	CAL31	CAL31
					1	0	1	CAL62	CAL62
1	1	1	1	1	x	x	x	-	-

Logic for Sequency pattern generation

This pattern is generated by a binary counter which increments from 0 to 127 for every low-high clock transition for obtaining a 128 bit long pattern. The counter resets after 128th count. The output gets a high for first count and low for the remaining counts. Bit-by-bit pattern is obtained at every clock cycle. Sequency pattern is free running i.e. is depends only on the clock transition and not the control bits or any of the latches. Thus, this pattern is always obtained for group-6 and group-7 selection.

Architecture and Logic for noise patterns generator

4 patterns are generated corresponding to noise patterns of duty cycle 25%, 50%, 0% and 100%.

- There is a *4:1 Multiplexer* which gives one of the noise patterns at the output. The select lines are the control bits C0-C1. It is to be noted that unlike the Walsh generator, only single pattern is generated here when its control bit combination has been selected and not all the patterns are generated simultaneously. We obtain a bit-by bit output for every clock input from the binary counter for noise patterns generator.
- 25% and 50% patterns have been are generated using *2 binary counters* which count from 0 to 7 and are incremented with every low-high transition of the clock, respectively.
- For 25% duty cycle, NGN2 pattern is obtained which is high for 2 counts of one of the counters.
- For 50% duty cycle NGN3 pattern is obtained which is high for 4 counts of the other counter.
- NGN1 pattern is always low for every clock cycle.
- NGN4 is always high for every clock cycle.

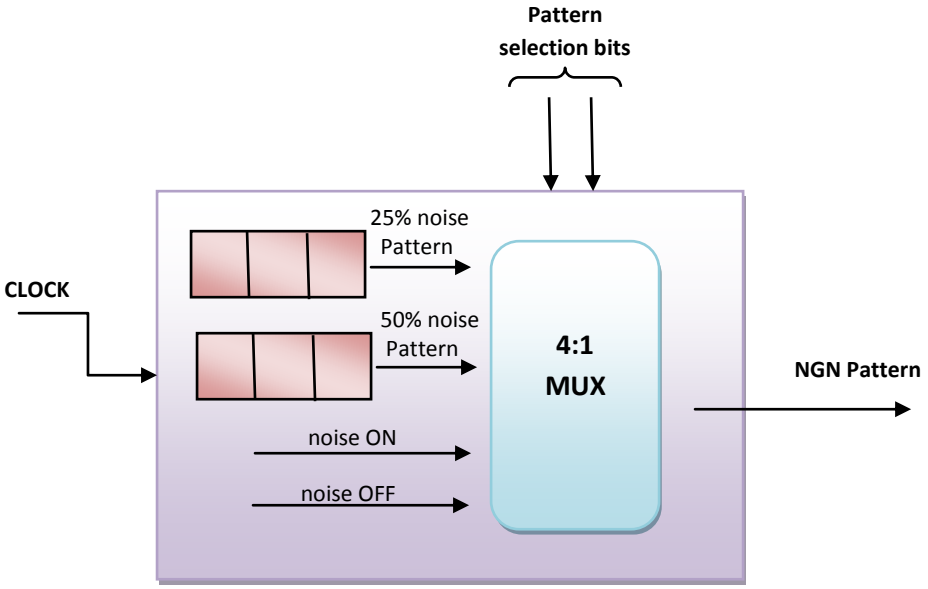


Figure 3.11- Block Diagram for Noise Pattern Generator

Table 3.3- Truth Table for Noise Pattern selection

Pattern selection bits		Noise pattern (NGN)	Duty Cycle
D1	D0		
0	0	NGN1	0% (Noise OFF)
0	1	NGN2	25%
1	0	NGN3	50%
1	1	NGN4	100%

CHAPTER-IV

PHYSICAL REALIZATION ON CPLD

4.1 PHYSICAL IMPLEMENTATION ON CPLD CHIP

After developing the design, our next step is to implement it in a CPLD. We have chosen the Xilinx XC95108-15 PC84C from the Xilinx XC9500 family and Xilinx Design Tools 8.2i for synthesis and simulation of the design.

4.1.1. Architecture of XC95108

The XC95108 is a high-performance CPLD providing advanced in-system programming and test capabilities for general purpose logic integration. It is comprised of 6 function blocks of 18 macrocells (total of 108 macrocells) providing 2,400 usable gates with propagation delays of 7.5 ns and is available in 84 pin PLCC. See Figure for the architecture overview.

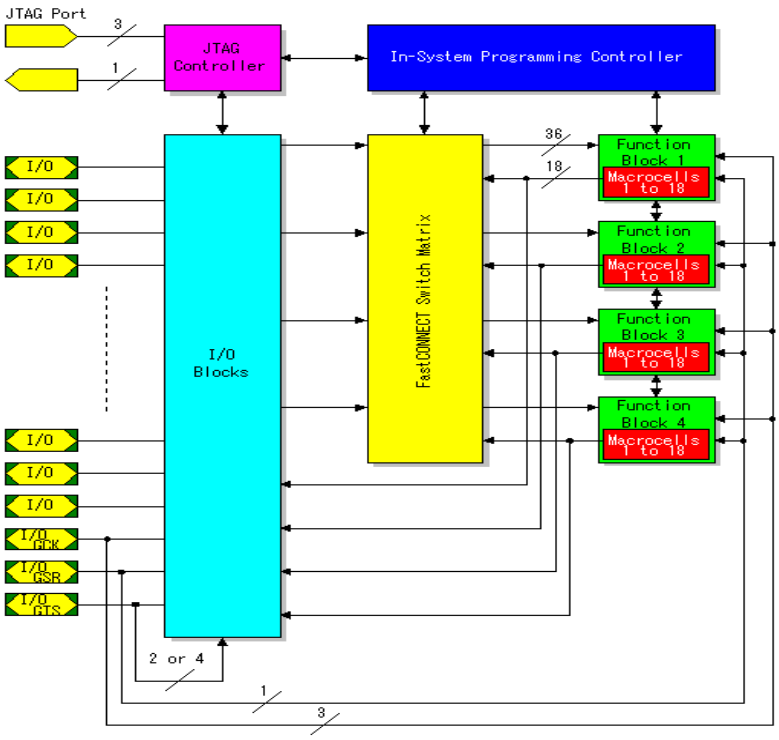


Figure 4.1- Architecture overview of XC95108

Features of XC95108

- 7.5 ns pin-to-pin logic delays on all pins
- fCNT to 125 MHz
- 108 macrocells with 2400 usable gates
- Up to 108 user I/O pins
- 5 V in-system programmable (ISP)

- Endurance of 10,000 program/erase cycles

- Program/erase over full commercial voltage and temperature range

- Enhanced pin-locking architecture

❖ Flexible 36V18 Function Block

- 90 product terms drive any or all of 18 macrocells within Function Block

- Global and product term clocks, output enables, set and reset signals

- Extensive IEEE Std 1149.1 boundary-scan (JTAG) support.
- Programmable power reduction mode in each macrocell
- Slew rate control on individual outputs
- User programmable ground pin capability
- Extended pattern security features for design protection
- High-drive 24 mA outputs
- 3.3 V or 5 V I/O capability
- Advanced CMOS 5V Fast FLASH technology
- Supports parallel programming of more than one XC9500 concurrently
- Available in 84-pin PLCC, 100-pin PQFP, 100-pin TQFP and 160-pin PQFP packages

4.1.2. Device Programming

There are number of steps to be followed in order to program the CPLD and implement the design in it. The following flow chart shows the steps involved in the design flow. We will be describing the step-by-step procedure for programming XC95108 for the design of Walsh card with reference to this design flow.

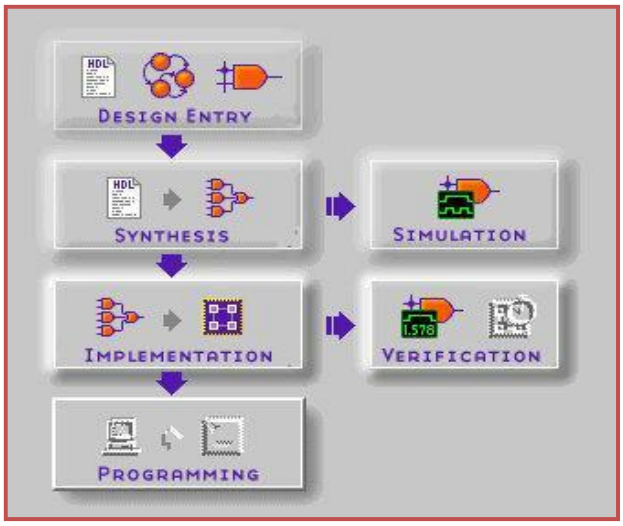


Figure 4.2-Flow Graph for programming CPLD

Design Entry

A description of the hardware's structure and behavior is written in a high-level hardware description language (usually VHDL or Verilog) and that source code is then compiled and downloaded prior to execution. Care should be taken while writing the source codes as these codes will be executed and transformed to programming logic for distributing resources (macrocells) in the CPLD. The code should be effective in order to allocate the resources optimally in the device.

Hardware description languages(HDL's) such as VHDL have made it possible for circuit and board design to be done without resorting to paper allowing computers to manage the design database and automate the translation between various representations of the systems . The VHSIC Hardware Descriptive Language (VHDL) is an industry standard language used to describe hardware from the abstract to the concrete level. VHDL is a powerful language with numerous language constructs that are capable of describing very complex behavior. VHDL is very adaptable, owing to its architecture, allowing designers, electronic design automation companies and the semiconductor industry to experiment with new language concepts to ensure good design tools and data interoperability. VHDL descriptions specify exactly what functions a new device would have to perform and the timing information associated with it. Through simulation of these descriptions, the design of new device can be accurately modeled before being physically verified. Also, it allows the detail structure of a design to be synthesized from more abstract specification, allowing designers to concentrate on more strategic design decisions and reducing time to market.

Xilinx ISE Tools 8.2i have been used to define the VHDL source codes. We have made the source code for programming the CPLD for the Walsh card using VHDL. There is a 'main_gen' code for the overall structure described in previous chapter in order to interface the different modules for Walsh pattern generator, noise pattern generator, sequency pattern generator, clock circuitry, latches using the logic explained in previous chapter. All the source codes have been annexed in Appendix A for reference. The various truth tables and the logics described in previous chapter were used for implementing every block in the structure.

Table 4.1-Nomenclature for structure and VHDL codes

Nomenclature used for VHDL codes	Nomenclature in the structure
COUNTER_MASTER	Binary counter 1
COUNT_WGN	Binary counter 2
COUNT_NGN	Binary counter 3
WNG_MON	Monitoring latch
WNG_LATCH	Walsh latch
WALSH_GEN	Walsh pattern generator
WNG_MUX	
NOISE_GEN	Noise pattern generator
SEQUENCY_GEN	Sequency pattern generator
DS(0:4)	Antenna selection bits
Dat1,dat2,dat3,dat4,dat5,dat6, dat7, dat8, dat9, dat10	D0-D9
enbm	(enable bit for monitoring Latch i.e.(group6))
Enbw	(enable bit for Walsh Latch i.e.(group7))
MON(0-5)	Monitoring Outputs
MONC	MONC
NGN	Noise pattern
SEQ	Sequency pattern
WNG5	D4
WP1	CH1 Walsh pattern
WP2	CH2 Walsh pattern

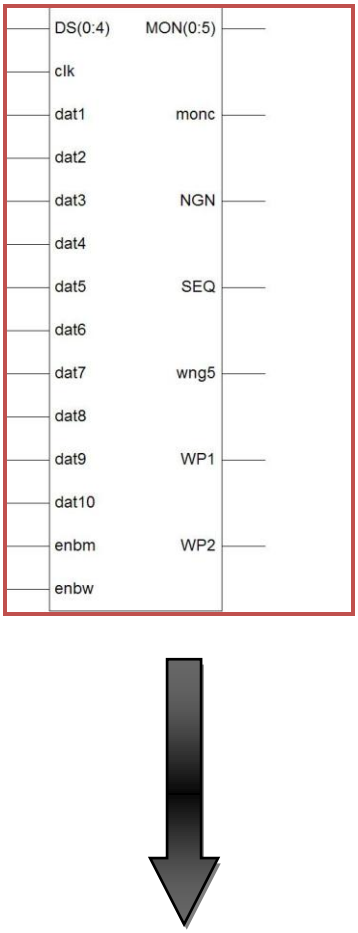
Logic Synthesis

Compilation only begins after a functionally correct representation of the hardware exists. This hardware compilation consists of two distinct steps. First, an intermediate representation of the hardware design is produced. This step is called synthesis and the result is a representation called a netlist. Logic synthesis is a process by which an abstract form of desired circuit behavior (typically register transfer level (RTL)) is turned into a design implementation in terms of logic gates and flip flops. The netlist is device independent, so its contents do not depend on the particulars of the FPGA or CPLD; it is usually stored in a standard format called the Electronic Design Interchange Format (EDIF).

Following steps are to be followed in order to synthesize the VHDL source codes for our project.

- Check the syntax by double clicking "check syntax" option under "synthesize-XST" under "implement design" in "processes window".
- For Synthesis, select implement Design-XST in the Xilinx ISE window.
- Then go to process →properties.
- Under the synthesis tab, select "Optimization Goal = Area" and "Optimization Effort = Normal". Then click OK.
- Double click "synthesize-XST".

We have checked the RTL schematic and the synthesis report for our design as shown below:



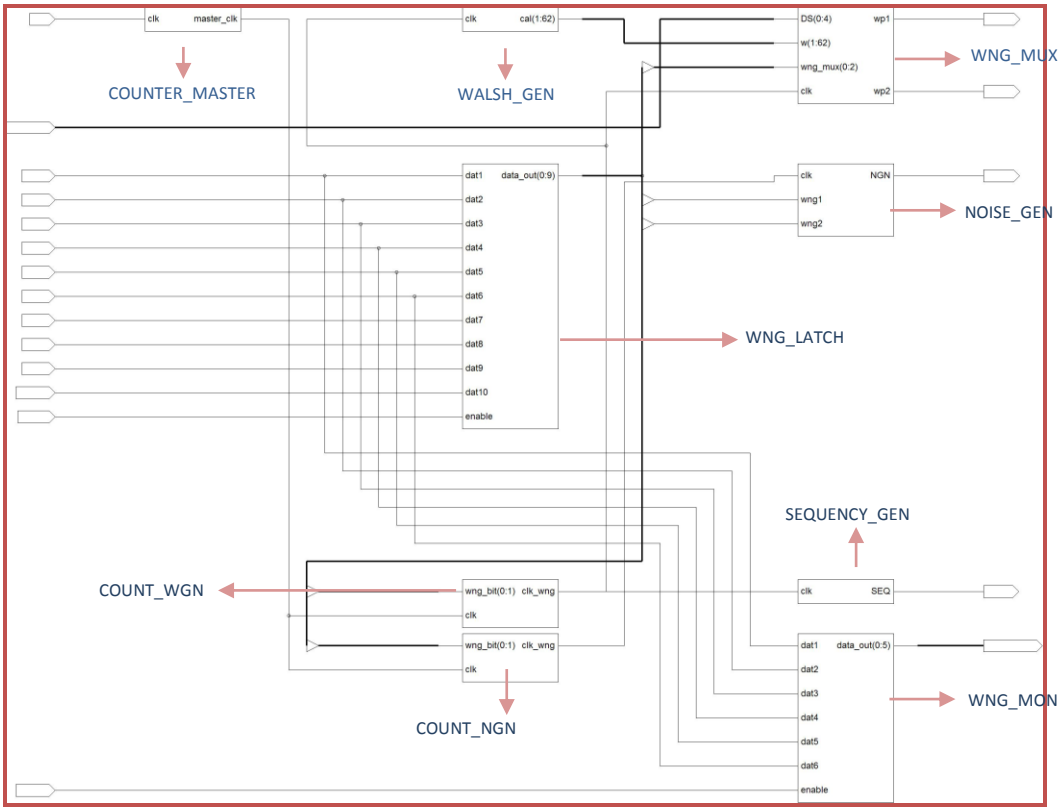


Figure 4.3-RTL schematic for Black Box View and structure view

```
=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : main_gen.ngr
Top Level Output File Name         : main_gen
Output Format                      : NGC
Optimization Goal                  : Area
Keep Hierarchy                     : YES
Target Technology                  : XC9500 CPLDs
Macro Preserve                     : YES
XOR Preserve                       : YES
wysiwyg                            : NO

Design Statistics
# IOs                              : 30

Cell Usage :
# BELS                             : 649
#      AND2                        : 189
#      AND3                        : 15
#      INV                         : 225
#      OR2                         : 134
#      XOR2                        : 86
# FlipFlops/Latches                : 57
#      FD                          : 57
# IO Buffers                       : 30
#      IBUF                        : 18
#      OBUF                        : 12
=========================================================================
```

Figure 4.4- View of Synthesis Report

Implementation

The second step in the translation process is called "Implement Design". It consists of "Fit" process. This step involves mapping the logical structures described in the netlist onto actual macrocells, interconnections, and input and output pins. This process is similar to the equivalent step in the development of a printed circuit board, and it may likewise allow for either automatic or manual layout optimizations. The result of the Fit process is a bitstream. Each CPLD (or family) has its own, usually proprietary, bitstream format. The bitstream is the binary data that must be loaded into the CPLD to cause that chip to execute a particular hardware design.

Following steps must be followed in order to implement our design in the CPLD:

- Select implement Design-XST in the Xilinx ISE window.
- Then go to process → properties.
- Add user constrain by generating UCF file using "Assign Pin Package Option" for input-output pin configuration. The UCF file has been annexed in Appendix B.
- Under the fitting tab, select "Implementation Template =Optimize Density" and "Logic Optimization = Density". Then click OK.

The "Fitter Report" giving the summary of CPLD resources, input-output pin list, logic, function blocks used and allocation of macrocells in each function block, equations (containing product terms) was generated, the summary of implementation is shown below:

Summary

Design Name	main_gen
Fitting Status	Successful
Software Version	I.31
Device Used	XC95108-15-PC84
Date	3-15-2009, 6:30PM

RESOURCES SUMMARY

Macrocells Used	Pterms Used	Registers Used	Pins Used	Function Block Inputs Used
101/108 (94%)	252/540 (47%)	58/108 (54%)	30/69 (44%)	157/216 (73%)

PIN RESOURCES

Signal Type	Required	Mapped	Pin Type	Used	Total
Input	17	17	I/O	26	63
Output	12	12	GCK/IC	2	3
Bidirectional	0	0	GTS/IO	2	2
GCK	1	1	GSR/IO	0	1
GTS	0	0			
GSR	0	0			

GLOBAL RESOURCES

Signal mapped onto global clock net (GCK2)	clk

POWER DATA

Macrocells in high performance mode (MCHP)	101
Macrocells in low power mode (MCLP)	0
Total macrocells used (MC)	101

Figure 4.5-Summarized Fitter Report

Simulation

Typically, the synthesis step is followed or interspersed with periods of functional simulation. That's where a simulator is used to execute the design and confirm that the correct outputs are produced for a given set of test inputs so that the designer can at least be sure that his logic is functionally correct before going on to the next stage of development. We have used the Xilinx Simulation Tool 8.2i for simulating the design. The simulated results are as follows:

The following results have been obtained for **D0=1, D2=0, D2=0, D3=0, D4=1, D5=1, D6=1, D7=0, D8=0, D9=0**

NGN <=NGN2 (25%); WP1<=CAL 2; WP2<=CAL 33; WNG5<=1;

MON 0<=0; MON 1<=1; MON2<=0, MON 3<=0, MON 4<=0; MON5<=0

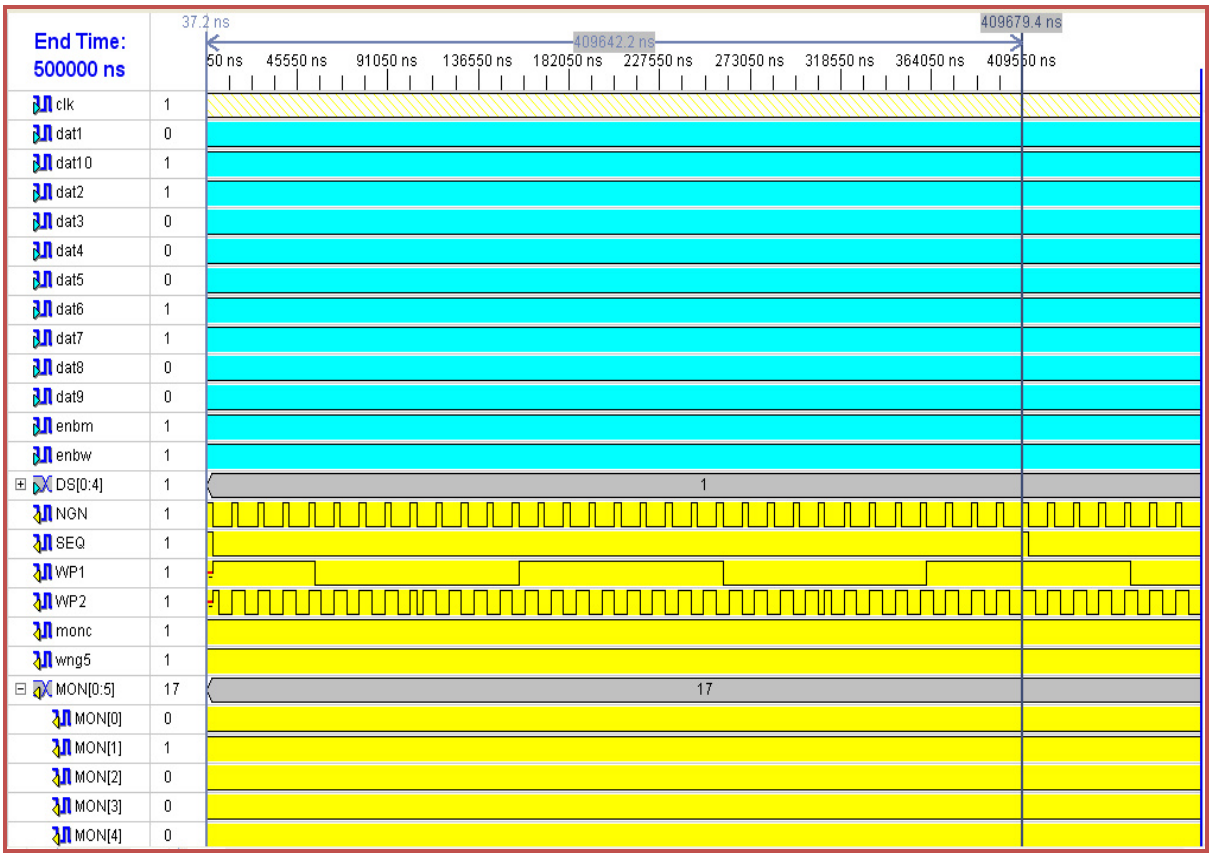


Figure 4.6-Simulation results for the design

In the next figure, the simulation results for the 62 cal patterns have been shown for two time periods. Each time period consists of a 128 bit pattern. Each pattern bit is generated with the transition in clock. Every pattern repeats itself after one time period.

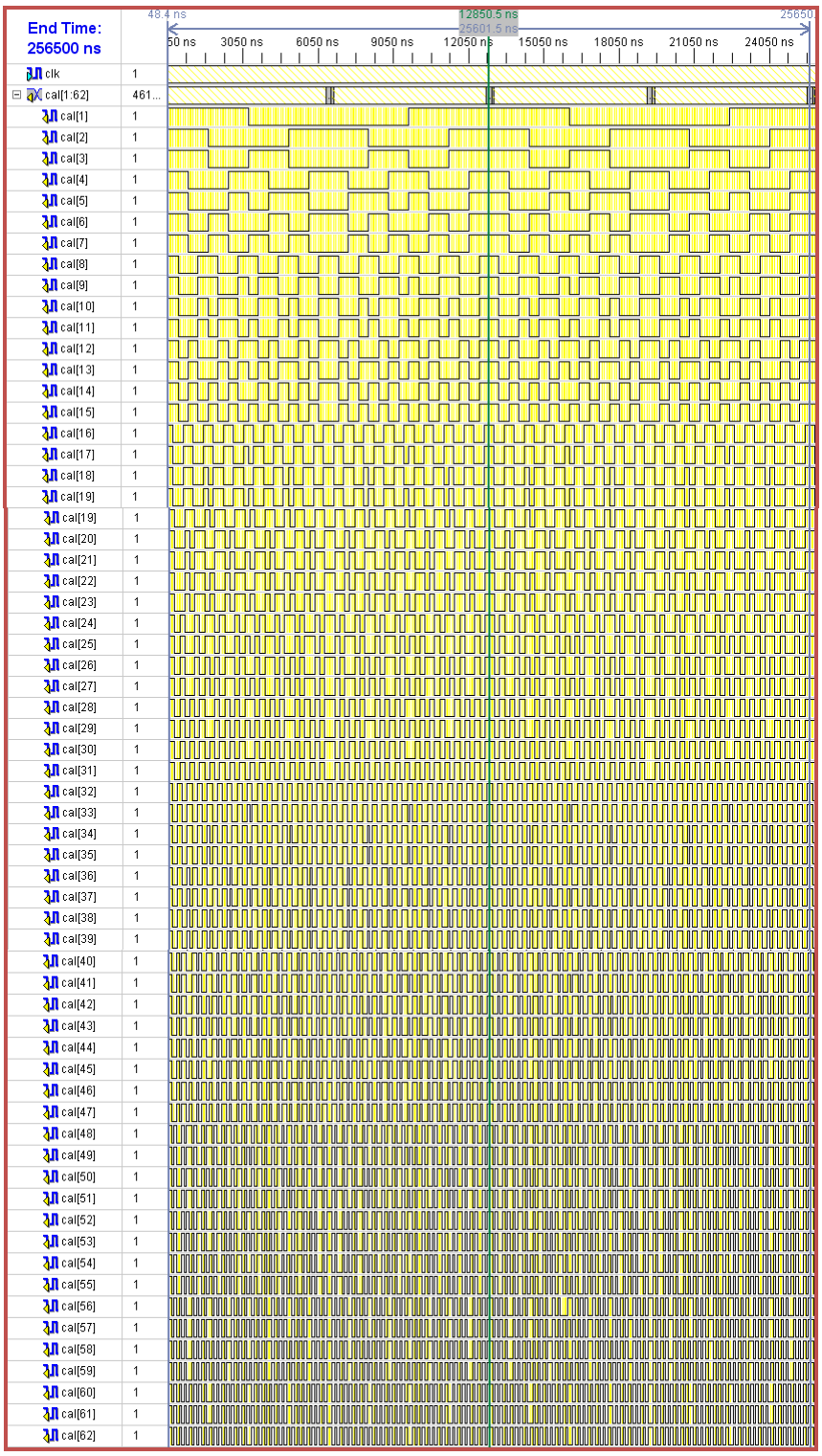


Figure 4.7- Simulated results for 62 cal patterns

In the following simulation results for Walsh generator unit, we have shown the output 128 bit Walsh patterns on both the channels for different selections for antenna 1. It is to be noted that the end of each pattern is marked by a new 128 bit sequency pattern.

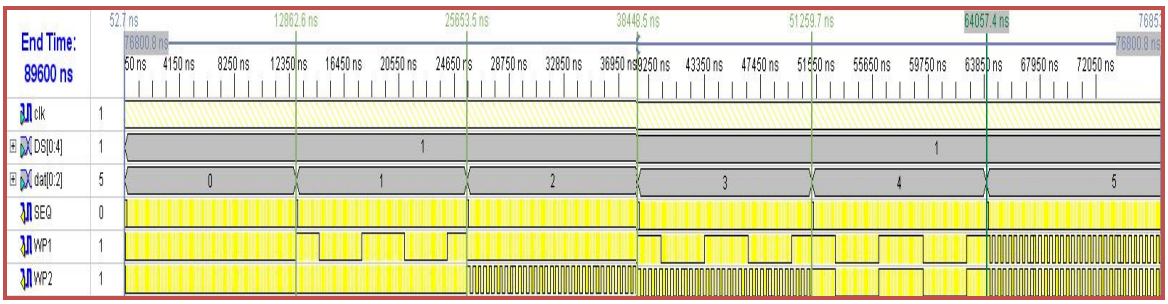


Figure 4.8-Simulation results for Walsh generator unit.

In the following simulation results for noise generator unit, we have shown the output noise patterns for different control bit selections.

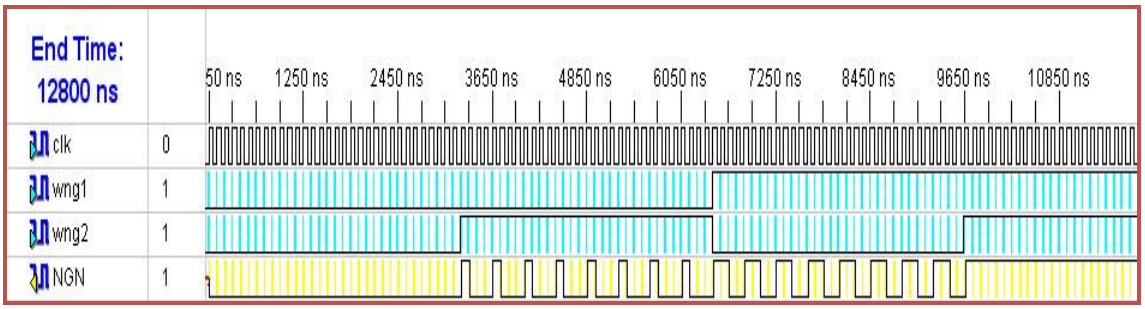


Figure 4.9-Simulation results for Noise pattern generator

Programming

Once a bitstream is created for a particular CPLD, we need to somehow download it to the device. The details of this process are dependent upon the chip's underlying process technology.

XC95108 is programmed using the in-system programmable circuit with a JTAG interface. To physically implement the design in a CPLD chip, a development kit is necessary. The development kit must be connected to a PC running ISE Tool through a downloading cable in order for the chip to be programmed.

Figure 4.10-Devenlopment Kit

Development kit used by us is "Mechatronics Universal Trainer Kit" (Test Equipment Model-MXUK-SMD-001). The view of the kit is shown in the figure next. After the implementation of the design,

- Run the process "impact" under "generate programming file" process.
- Right click the icon and select "program". The device is programmed and a "program succeeded" message is displayed.

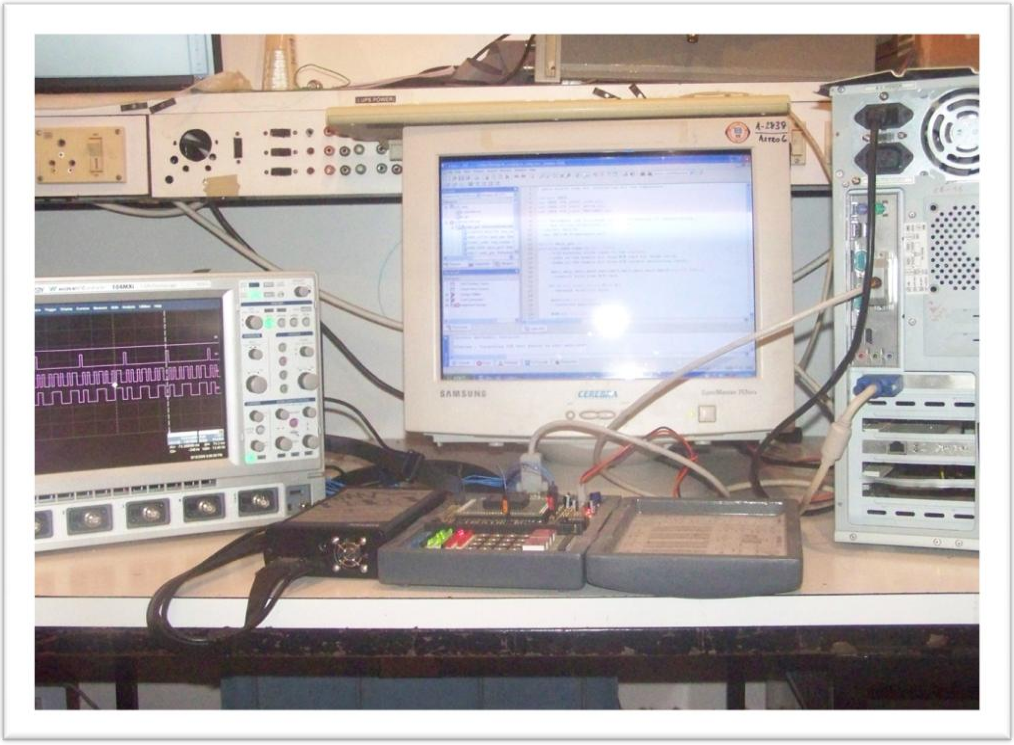


Figure 4.11-Setup for Device Programming

4.2. RESULTS ON MSO

Inputs for the various control bits were given through the input switches on the development board used for programming the CPLD. The outputs at the respective pins of the CPLD chip were taken using Mixed Signal Oscilloscope (LeCroy MS-500 Mixes Signal Oscilloscope). The results obtained on the MSO are shown below. These have been taken for internal clock frequency division by 1 for all the patterns and antenna selected is 2 through input switches on the development board.

- D_0=>50% noise pattern (NGN 3)
- D_1=> Sequency Pattern
- Both channels enabled with different Cal patterns i.e.

D_2=>CAL 33; D_3=>CAL 2

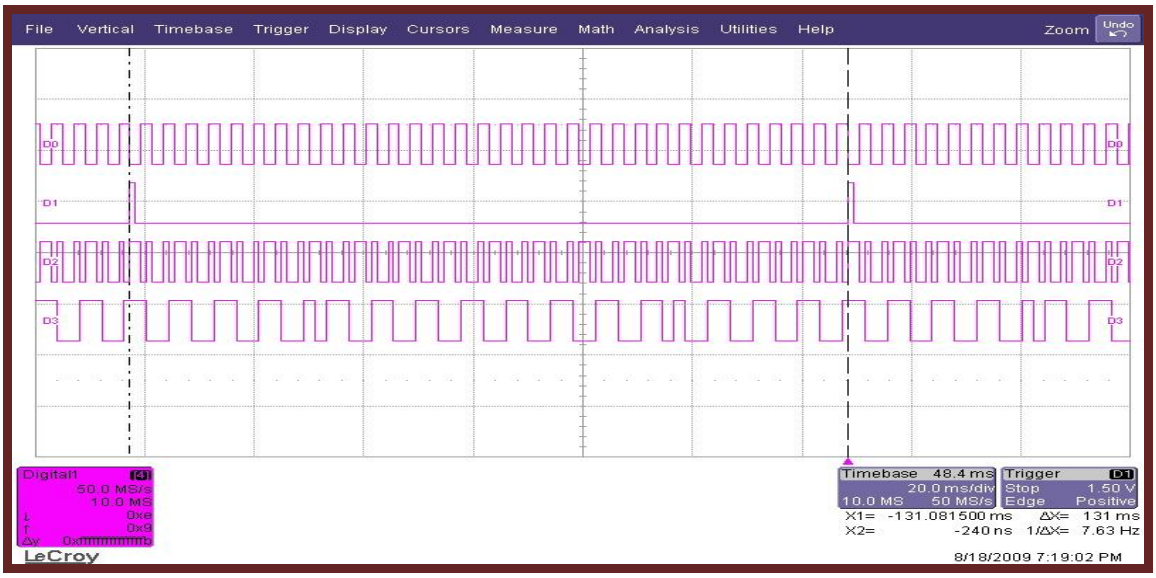


Figure 4.12- Output results of programmed CPLD chip on MSO

Observations and Calculations

❖ *Calculation of number of bits in each of the Walsh and Sequency patterns:*

Sequency pattern marks the start and stop of Walsh patterns. In order to calculate the number of bits in pattern obtained in MSO, we need to find the time difference between two consecutive Sequency patterns.

The incoming clock frequency to the CPLD chip is 4 MHz from the development board. It is internally divided by 4096. For our selection, we have chosen further frequency division by 1.

$$Fin = 4MHz$$
$$F_1 = 4000000/4096 = 976.5Hz \cong 977Hz$$
$$\therefore F_{11} = F_1/1 = 977Hz$$

- Time period for single bit in each sequency pattern is (observed on MSO):

$$\Delta X_1 = 1ms$$
$$\therefore \Delta F_1 = 1/\Delta X_1 = 977Hz$$

- Time period for a complete sequency pattern (as shown in figure 3.12) is:

$$\Delta X_2 = 131ms$$
$$\therefore \Delta F_2 = 1/\Delta X_2 = 7.63Hz$$

- The number of bits in each sequency pattern is:

$$no.ofbits = \Delta F_1 / \Delta F_2 = 977 / 7.63 = 128.047 bits$$
$$\cong 128 bits$$

Thus the length of each Walsh pattern=128 bits

❖ *Independent time period for Walsh patterns and Noise Patterns:*

In the following figures, we show the various results obtained by varying the frequency selection bits for the noise patterns while keeping the frequency selection for the Walsh patterns constant. A 50% duty cycle noise pattern has been chosen for this frequency variation and cal1 and cal32 patterns have been selected on channel 1 and channel 2 respectively.

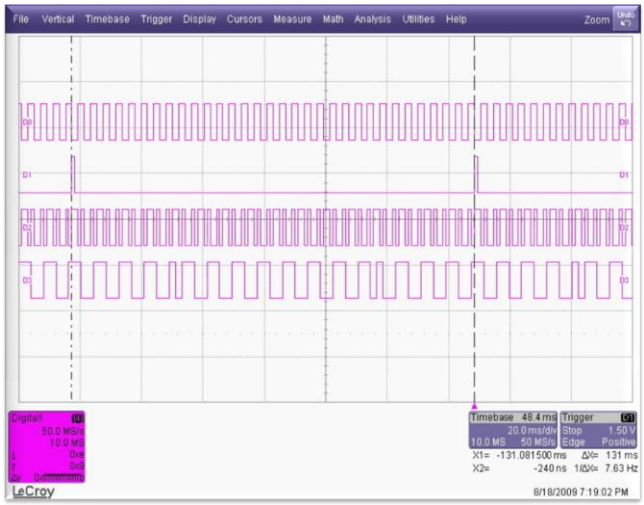
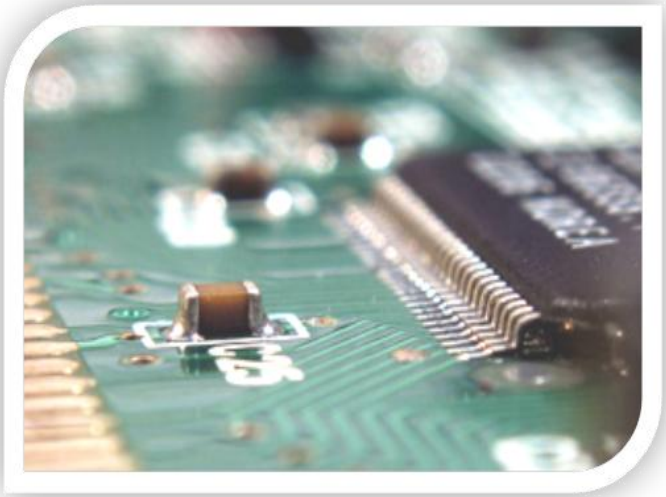


Figure 4.13 a- results for 4 ms time period selection for 50 % Noise pattern

Figure 4.13 b- results for 8 ms time period selection for 50 % Noise pattern

Figure 4.13 c- results for 16 ms time period selection for 50 % Noise pattern

Figure 4.13 d- results for 32 ms time period selection for 50 % Noise pattern



CHAPTER-V

HARDWARE IMPLEMENTATION & TESTING

5.1. WALSH CARD PCB DESIGN

After programming the CPLD chip, next step is to fix the CPLD chip onto a PCB to complete the hardware implementation of the new Walsh Card circuit based on CPLD. The block diagram for implementation has been shown in the chapter 3. For implementing this on printed circuit board, a PCB design was made using Altium Designer 6.1. Following steps are involved in PCB design:

- *Making the schematic for the circuit*: this involves placing the various components used for the design onto a schematic sheet and indicating net labels (connections) between various components. The schematic for our circuit is as follows:

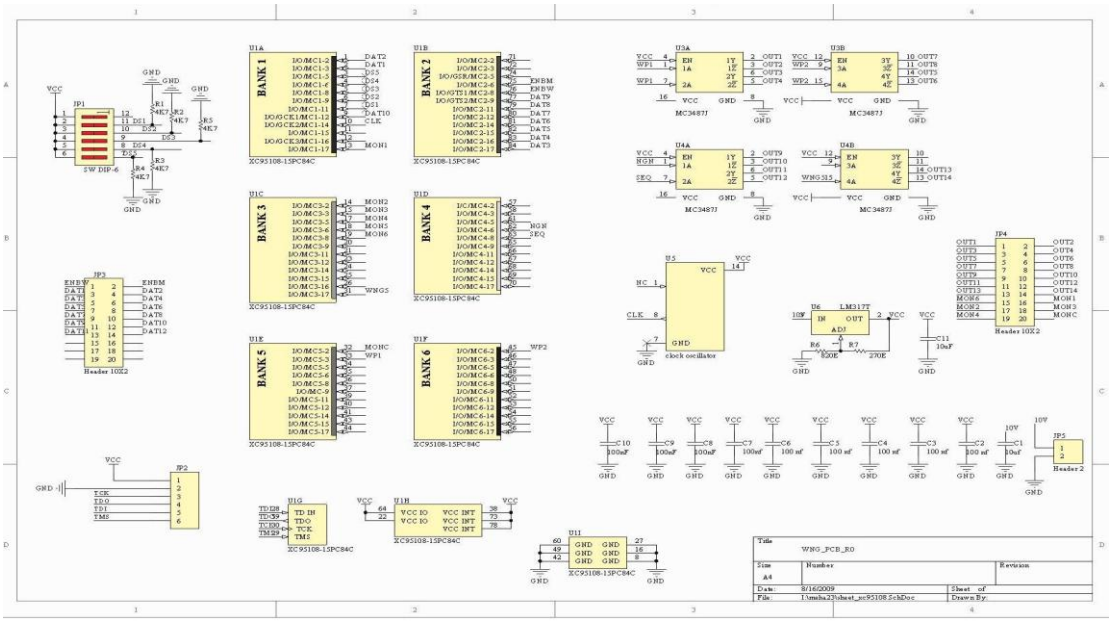


Figure 5.1- Schematic for Walsh Card PCB

- *PCB Layout:* After making the schematic, it is updated onto a PCB. The PCB design sheet in the PCB design window consist of the details footprints of the components, drill details , PC layers, solder masks, tracks and their widths between various components, etc. these tracks were laid by routing using manual routing and auto-routing options. The base for a PC84 package is PLCC84. Its footprint had to be made by modifying footprints of similar components.
- *Generation of output files:* Gerber files are required for manufacturing the PCB. These files were generated as output. There are other files as well and these have been annexed in Appendix C.

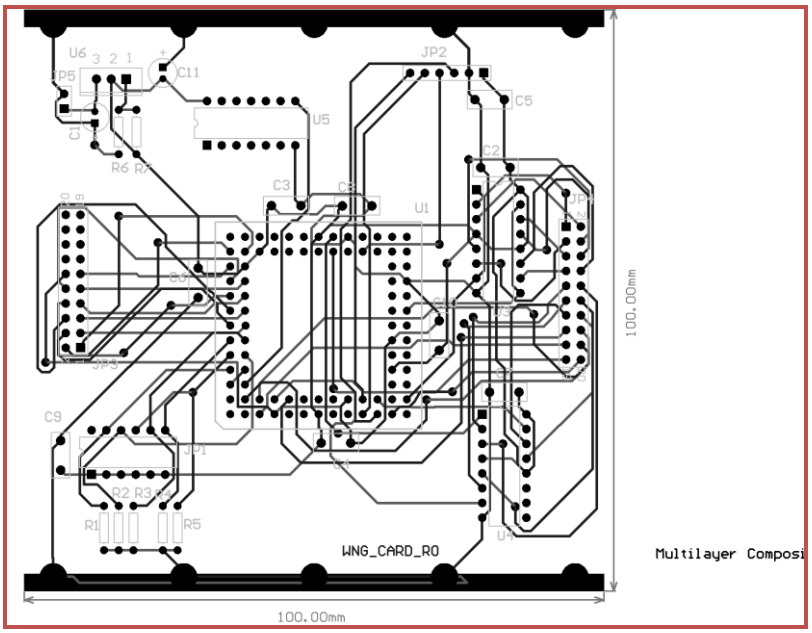


Figure 5.2-Multilayer composite output for Walsh card PCB

5.2. TESTING THE WALSH CARD PCB

The final step of the project work involves testing the Walsh card PCB. The final Walsh card PCB after manufacturing and soldering the components onto it is as shown below:

Figure 5.3- Walsh card PCB

5.2.1. Test Setup

- For testing the PCB, firstly the Walsh card PCB is placed in D-49 PIU. Output of the microcontroller (SBC086) in the MCM-2 i.e. the 15 control bits and the two enable bits for group-6 and group-7, i.e., enbm and enbw, respectively, are given as input to the Walsh card PCB, at the respective pins of the 20-pin input FRC, through the control card. +10 volt and -10 volt power supply was also provided which is brought to +5 volt by the voltage regulator in the circuit. The view of the D-49 PIU after assembly of MCM-2, Control card, Walsh card and power supply card is as follows:

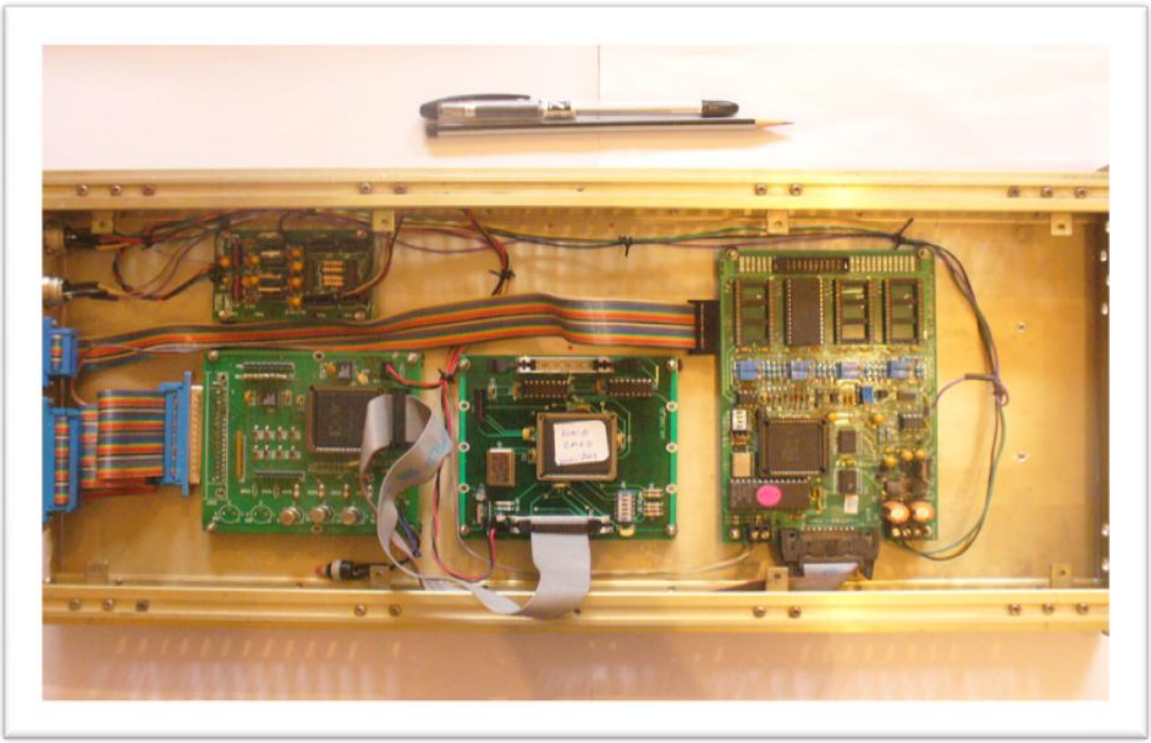


Figure 5.4- D-49 PIU with control card, CPLD based Walsh Card PCB, MCM-2 (left to right in lower row) and power supply PCB (in upper row)

- In the next step, the D-49 PIU is connected to DOS based computer through which 16 bit hexadecimal words are given to the MCM for performing various operations through the Walsh card PCB.
- The outputs from the CPLD and the 20 pin FRC are observed on the MSO connected to it.

The complete Test setup is as shown below:

Figure 5.5- Test Setup

5.2.2. Procedure for Testing

- First provide power supply to all the units.
- In the DOS PC make the command files containing the HEX codes for different test operations.
- Give the following commands on the DOS PC:
  - ✓ C:\ABR\MEHA\MCMNEW1
  - ✓ Then select port 1.
  - ✓ Give command 18 with MCM address 2 to start communication with the MCM-2.
  - ✓ Give NULL command to check execution.
  - ✓ Give command 20 to enter the file name containing the command.
  - ✓ Enter the name of the file.
- Check the corresponding output on the output pins under test in the MSO.
- Repeat the above steps (except first step) for different command files containing the HEX codes for different operations.

Making the command file

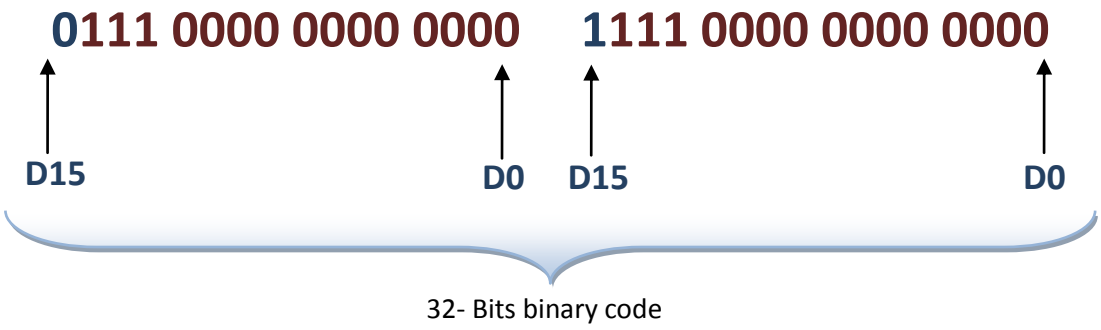
Command files contain the 32 bit HEX words given to MCM through the computer. Care should be taken while making a command file as it contains the sequence of 16 bit input bits given to the Walsh card PCB through the MCM-2. Each 16 bit word in the 32 bits word indicates what function is to be performed by the circuit. The configuration of all the bits for different operations has been described in chapter 3. In a command file, 32 bit data is to be entered, where, the first 16 bits are for low on D15 enable bit and the next 16 bits are for a high on the enable bit D15. This gives a low to high transition on the enable bit D15 so that the decoder in the control card will generate a corresponding low or high on the group-6 or group-7 enable bit leading to a transition on this bit. This in turn will enable the latches which are positive edge triggered.

For example, the following 32 bit HEX command format is to be followed as shown below. This command simply selects the group-7 by enabling enbw. The 32 bit binary code is given next.

HEX CODE:

7000 F000

BINARY CODE:



A list of command files that were used for testing has been given below with their names, the corresponding HEX codes with the ON bits in the first 12 control bits of the 16 bits sequence, and the desired function:

Table 5.1- List of command files used for testing the circuit

Command file name	Function	Hex codes	ON bits
FEON	Switch on the front end MCM	7010 F010	D4 is high
FEOF	Switch OFF front end MCM	7000 F000	All b12 bits are low
NGON	Enable 100% duty cycle noise pattern	7003 F003	D0, D1 are high
NG50	Enable 50% duty cycle noise pattern	7002 F002	D1 is high
NG25	Enable 25% duty cycle noise pattern	7001 F000	D0 is high
NGOF	Noise off	7000 F000	All 15 bits are off
WL1	Enable Walsh pattern only on channel 1 of the selected antenna	7020 F020	D5 is high
WL2	Enable Walsh pattern only on channel 2 of the selected antenna	7040 F040	D6 is high
WLB	Enable Walsh patterns on both the channels of the selected antenna	7060 F060	D5, D6 are high
WL1-CH12	Enable channel 1 Walsh pattern on both channels of selected antenna	7080 F080	D7 is high
WL2-CH12	Enable channel 2 Walsh pattern on both the channels of the selected antenna	70A0 F0A0	D5, D7 are high
NG50-F1	Enable divide by 2 frequency selection for 50% duty cycle noise pattern	7006 F006	D1, D2 are high
NG50-F2	Enable divide by 4 frequency selection for 50% duty cycle noise pattern	700A F00A	D1, D3 are high
NG50-F3	Enable divide by 8 frequency selection for 50% duty cycle noise pattern	700E F00E	D1, D2, D3 are high
WLB-F1	Enable divide by 2 frequency selection for different Walsh patterns on both the channels of the selected antenna	7160 F160	D5, D6, D8 are high
WLB-F2	Enable divide by 4 frequency selection for different Walsh patterns on both the channels of the selected antenna	7260 F260	D5, D6, D8 are high
WLB-F3	Enable divide by 8 frequency selection for different Walsh patterns on both the channels of the selected antenna	7360 F360	D5, D6, D8, D9 are high

MALL	Enable 25% duty cycle noise pattern, different Walsh patterns on both channels of the selected antenna	7071 F071	D0, D4, D5, D6 are high
MON	Enable the monitoring controls	603F E03F	D0, D1, D2, D3, D4 are high

5.3. TEST RESULTS

Following are the results obtained on the various output pins of the CPLD in the Walsh card PCB on MSO for the command files given in the table 5.1and the antenna no.1 has been selected by the setting the switches position to 0. It is to noted that in following figures,

- ✓ D0 is the noise pattern
- ✓ D1 is the sequency pattern
- ✓ D2 is the Walsh pattern for channel 1.
- ✓ D3 is the Walsh pattern for channel 2.
- ✓ D10 is the front end control signal.

I. Results for FEON & FEOF:

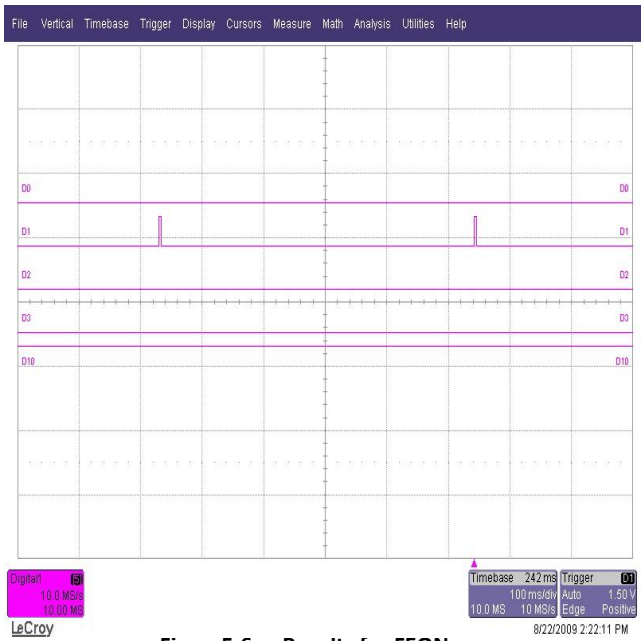


Figure5.6 a- Results for FEON

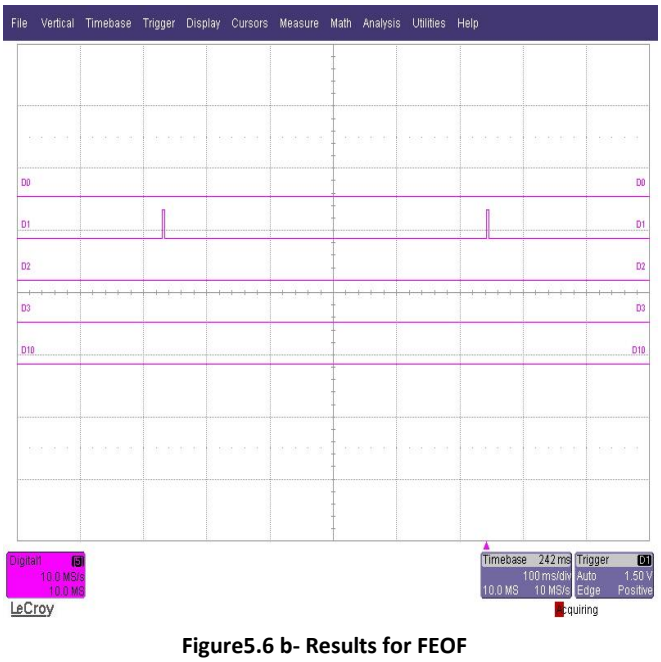


Figure5.6 b- Results for FEOF

II. Results for NGON, NGOF, NG50 & NG25

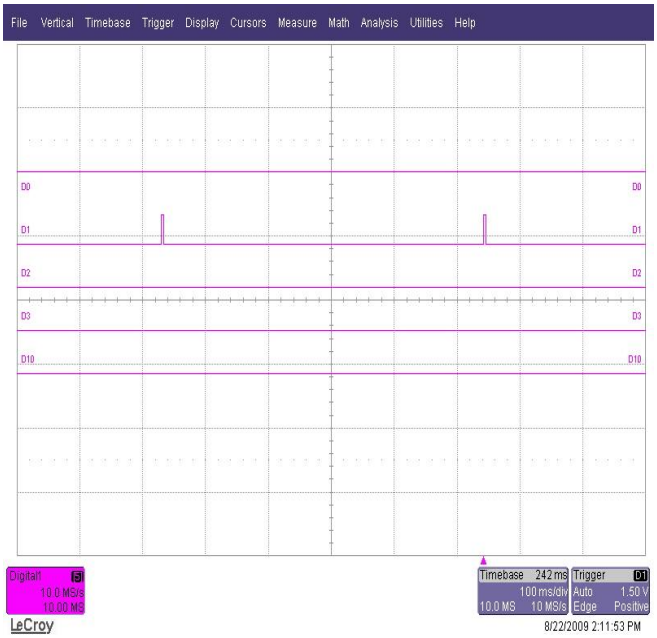


Figure 5.7a-Results for 100% duty cycle noise pattern

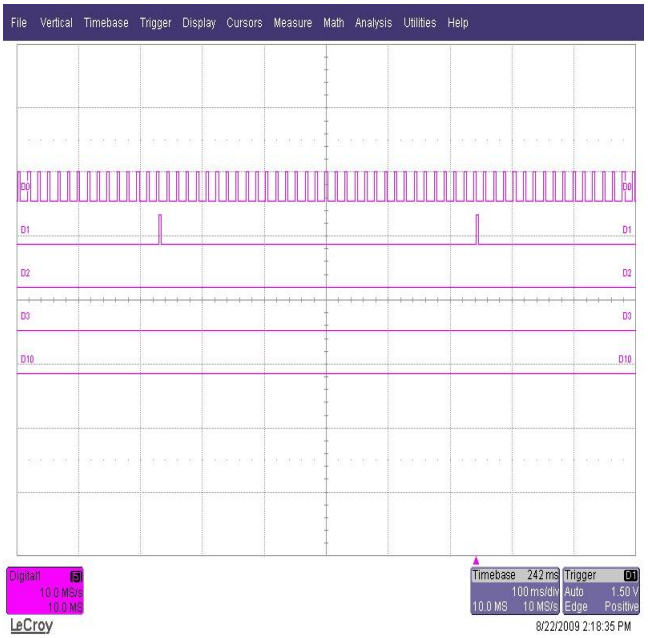


Figure 5.7b-Results for 25% duty cycle noise pattern

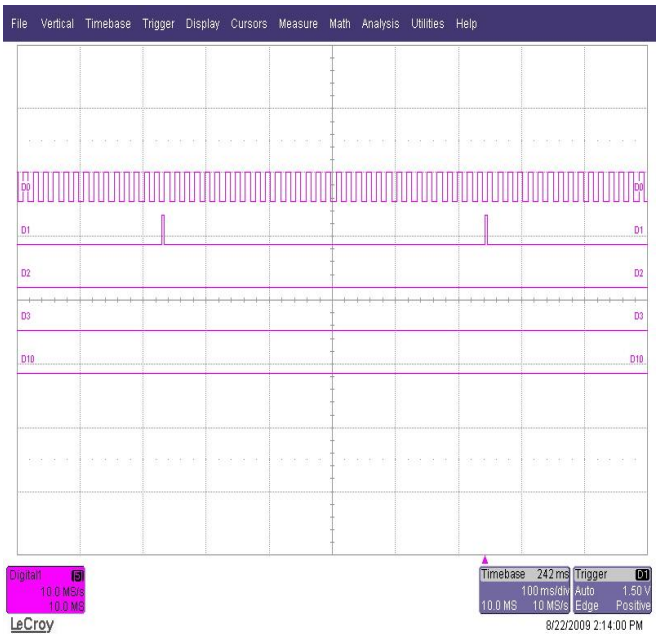


Figure 5.7c-Results for 50% duty cycle noise pattern

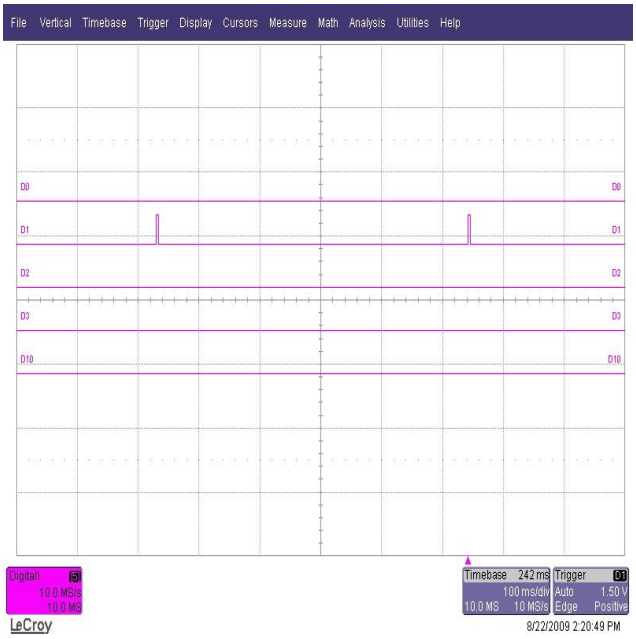


Figure 5.7d-Results for noise pattern disable

III. Results for WL1, WL2, WLB, WL1-CH12, WL2-CH12

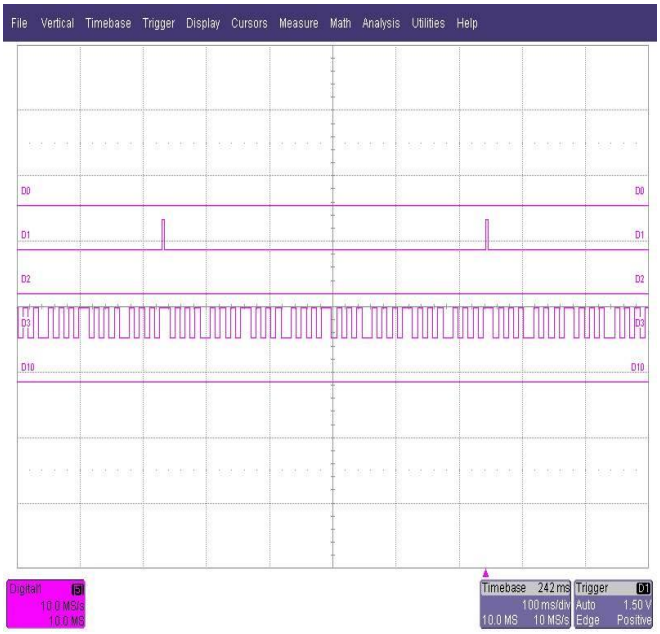


Figure 5.8a-Results for Walsh pattern on channel-1

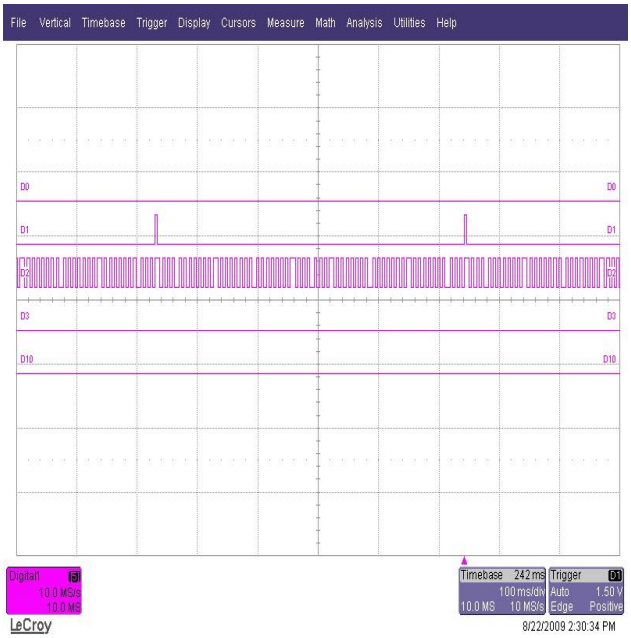


Figure 5.8b-Results for Walsh pattern on channel-2

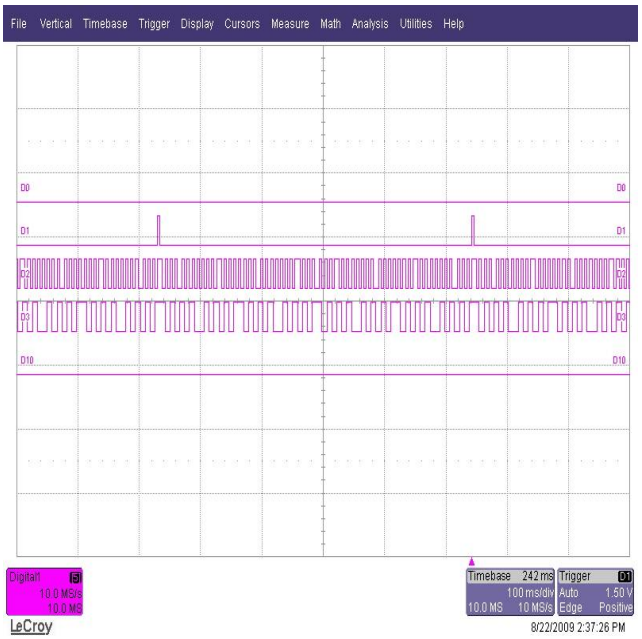


Figure 5.8c-Results for Walsh patterns on both channels

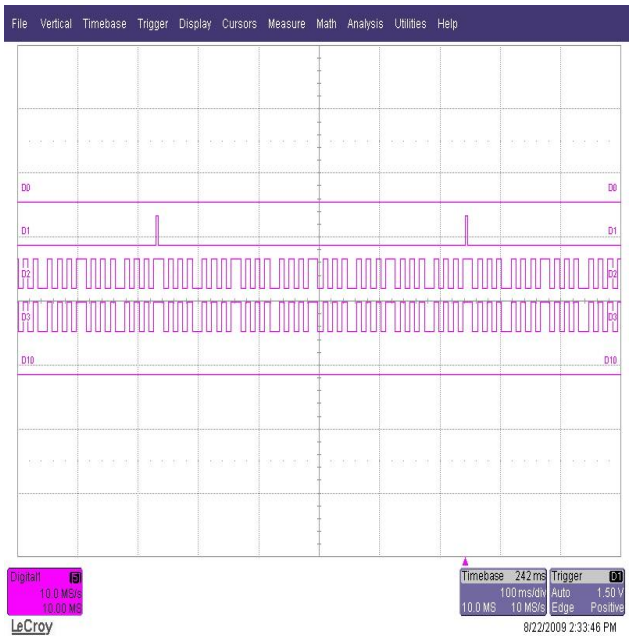


Figure 5.8d-Results for channel-1 Walsh pattern on channel-2

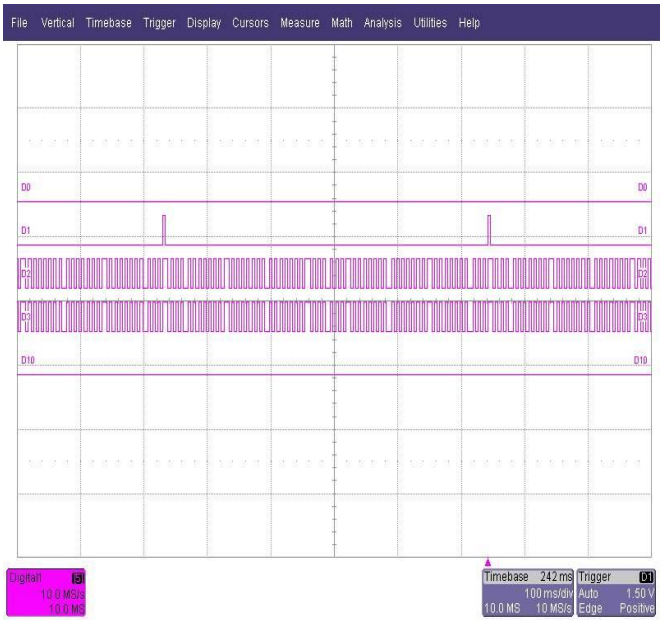


Figure 5.8e-Results for channel-2 Walsh pattern on channel-1

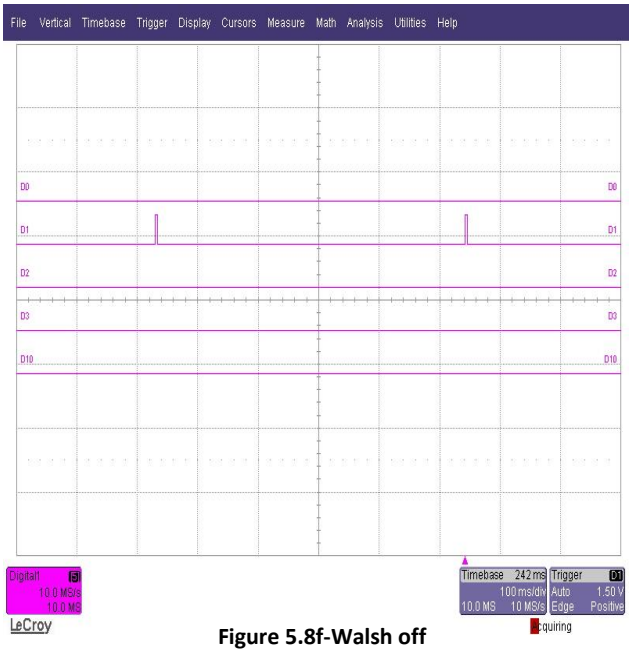


Figure 5.8f-Walsh off

IV. Results for MALL

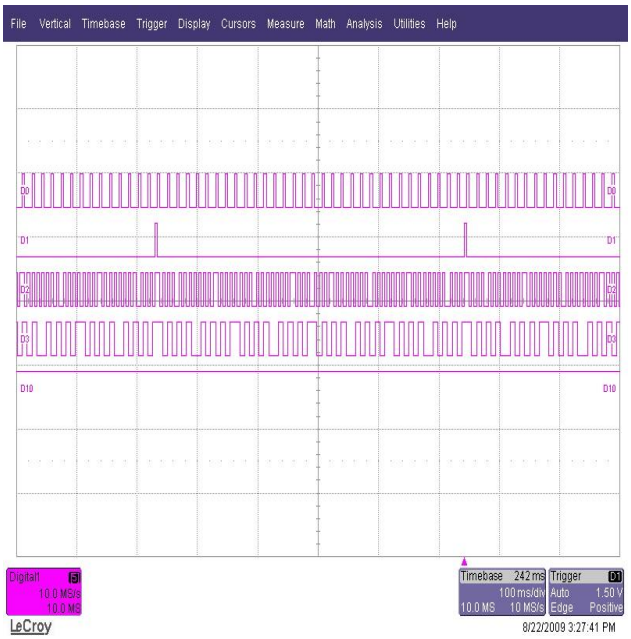


Figure 5.9-Results for the signals enabled at once

V. Results for MON

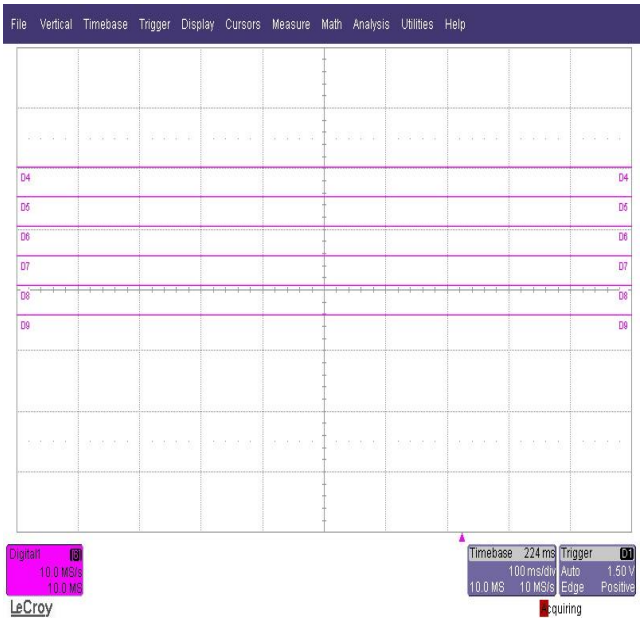


Figure 5.10-Results for monitoring signals enabled

VI. Results for MWB-F1, MWB-F2, MWB-F3 & NG50F1, NG50F2, NG50 F3.

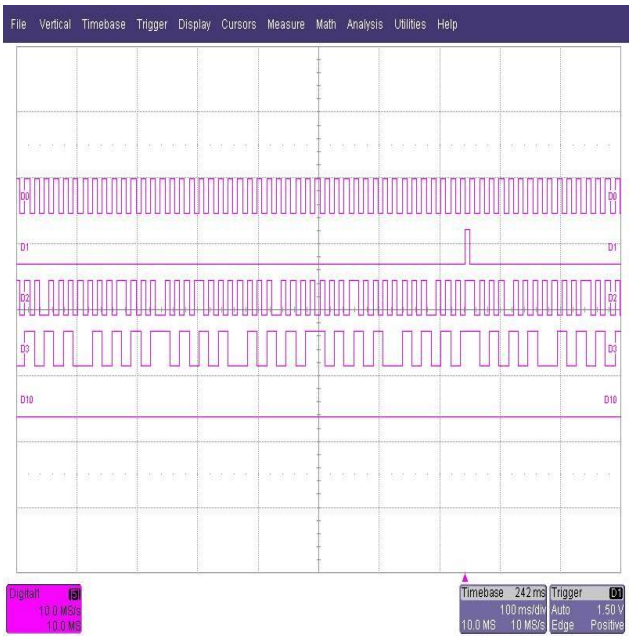


Figure 5.10a-Results for the freq. division by 2 for Walsh patterns while keeping it constant for the noise pattern

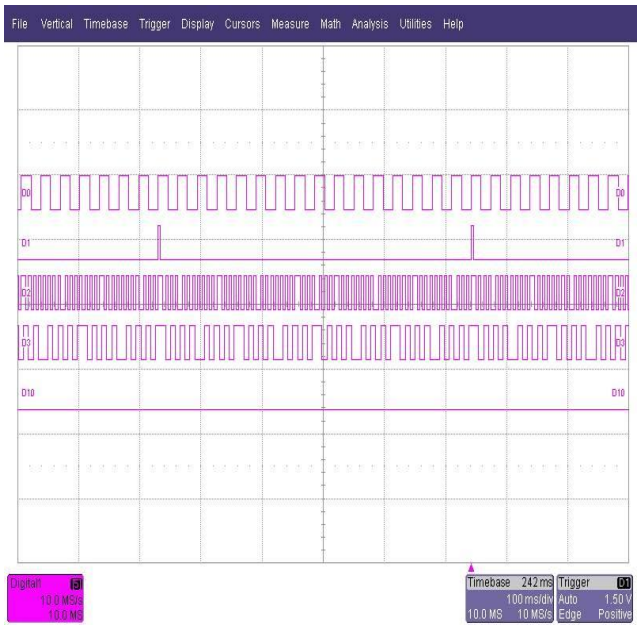


Figure 5.11a-Results for the freq. division by 2 for noise pattern while keeping it constant for the Walsh patterns

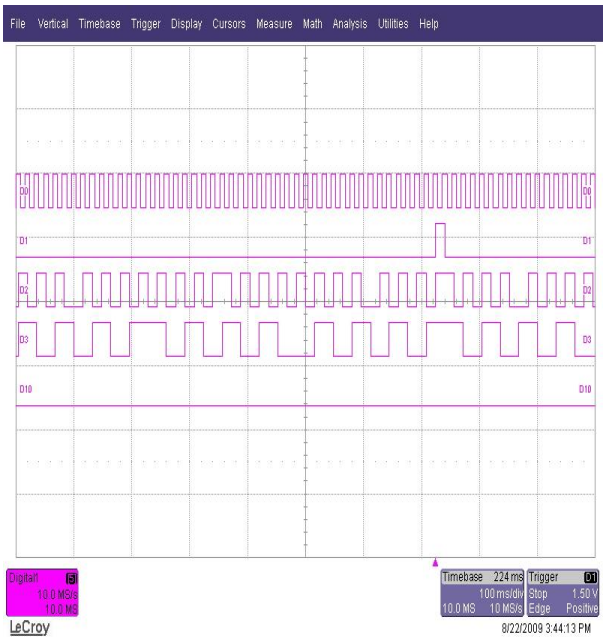


Figure 5.10b-Results for the freq. division by 4 for Walsh patterns while keeping it constant for the noise pattern

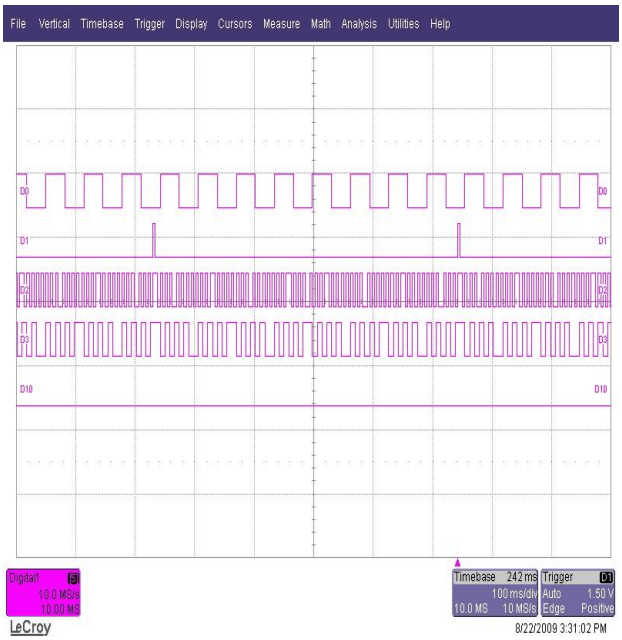


Figure 5.11b-Results for the freq. division by 4 for noise pattern while keeping it constant for the Walsh patterns

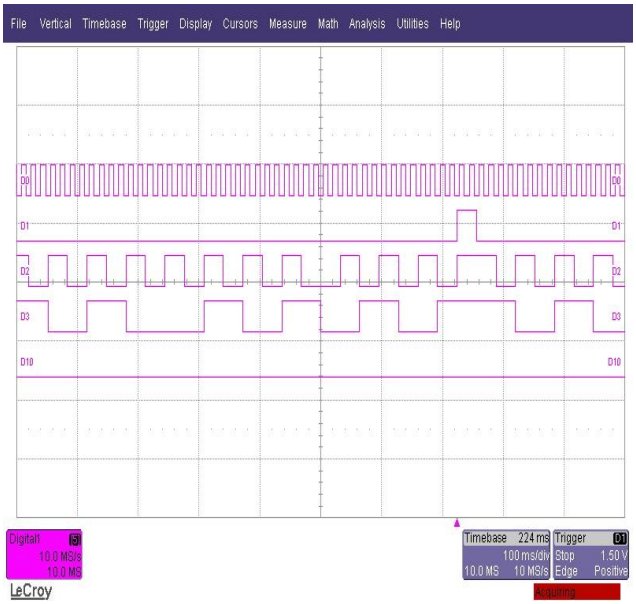


Figure 5.10c-Results for the freq. division by 8 for Walsh patterns while keeping it constant for the noise pattern

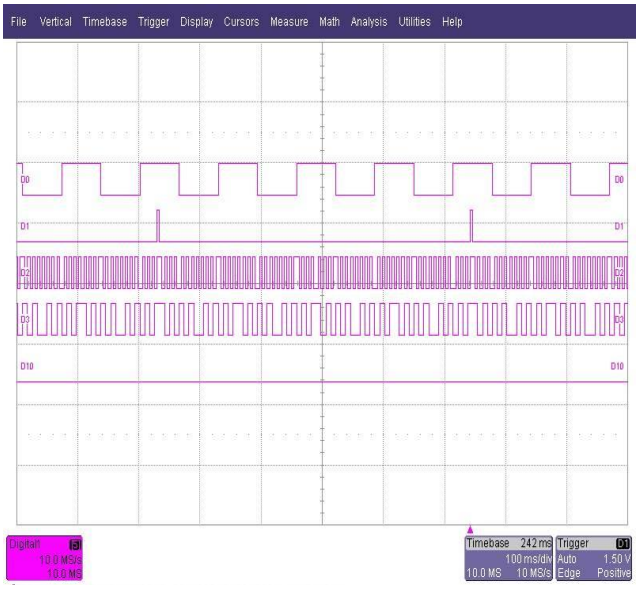
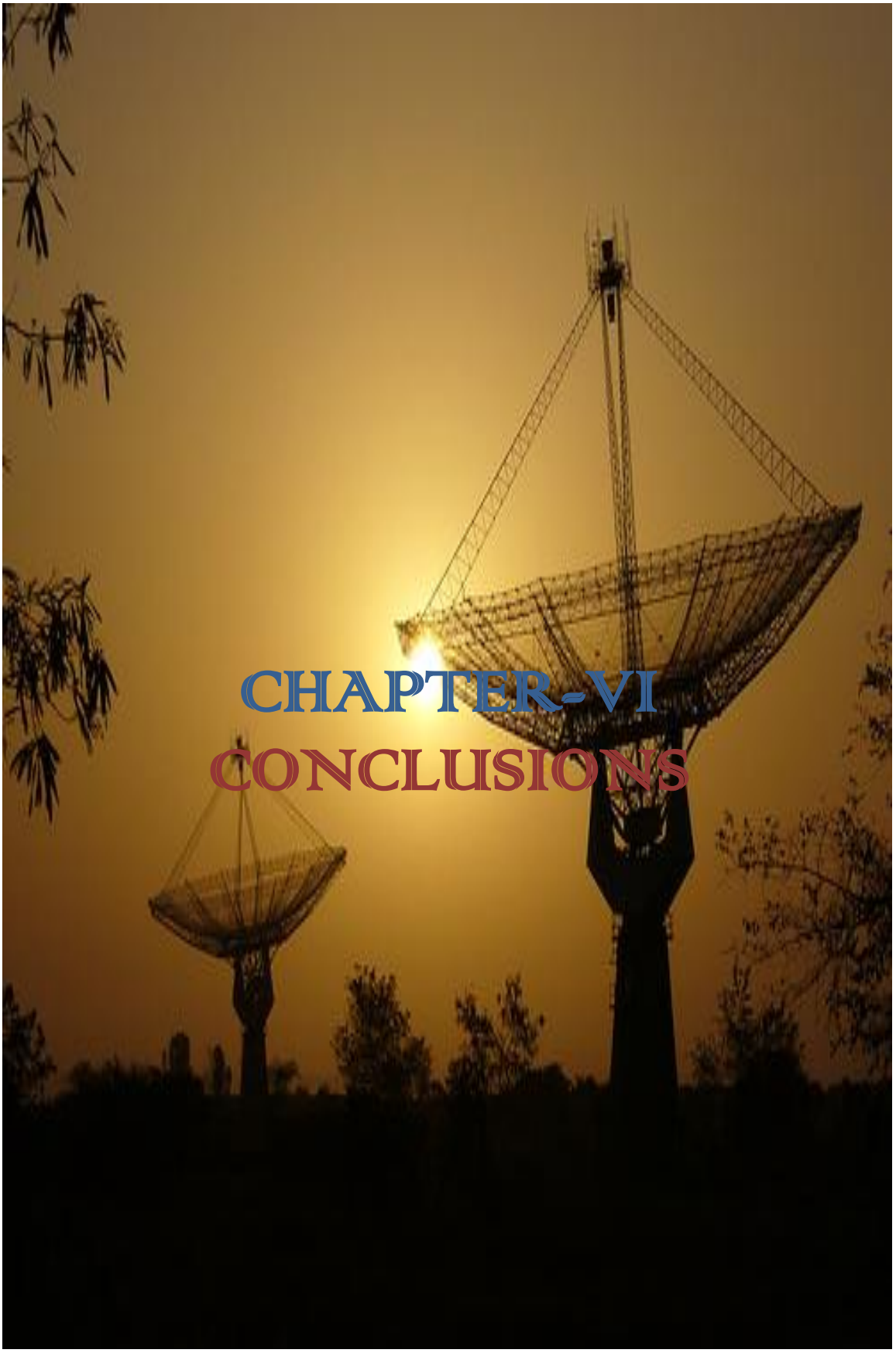


Figure 5.11c-Results for the freq. division by 8 for noise pattern while keeping it constant for the Walsh patterns

It is to be noted in figures 5.10a, 5.10b, 5.10c that even though the frequency for both the Walsh patterns on both the channels is being varied, the 50% duty cycle pattern for noise remains constant. Similarly in the figures 5.11a, 5.11b, 5.11c, the frequency for 50% noise pattern is varying but that of Walsh patterns remains constant.

After analyzing all these results we find that these results are as expected and these also verify the simulation results and the results obtained on the Development Board.



CHAPTER-VI
CONCLUSIONS

6.1. CONCLUSIONS

The Walsh Card PCB has been designed to control certain front end parameters required at the time of observation of a source. This circuit mainly consists of a programmable chip for generating various Walsh patterns, noise patterns, sequency patterns and monitoring signals and the differential line drivers for long distance transmission of these signals from the ABR to the front end. The chip can be reprogrammed as per the changes required in future. Extra facilities required during observation have been added to the circuit. The circuit has been tested thoroughly and found to meet the specifications and hence integrated with the MCM-2 in D-49 PIU at the ABR.

The CPLD based Walsh card adds many facilities for front end control. The new circuit allows a user to select variable time periods for all the patterns. The user can also select independent time periods for Noise patterns and the Walsh Patterns, sequency patterns. Another facility is that the user can select independent channels for selecting Walsh functions for a particular antenna. The user can disable Walsh pattern on any of the channels or have different patterns on both the channels simultaneously or have one channel's pattern on the other.

The performance of the Walsh card PCB after integration into the D-49 PIU has been tested and found to be as expected. The settings for the patterns selection are controlled through the DOS based computer and the results obtained are as per the design.

FUTURE SCOPES

In order to further modify the D-49 PIU in the LO Synthesiser, the Control card PCB for frequency control of the LO Synthesiser and the Walsh card PCB for Front-end control, both can be combined by using a single CPLD. The logic for functioning of the control card has already been implemented on a separate CPLD based PCB. To further reduce the space occupied by this separate Control card PCB and Walsh card PCB, both of these can be merged to a single CPLD based PCB which will control all the operations of group0, group1, group2, group3, group4, group5, group6 and group7 of the LO Synthesiser.

The hardware resources requirement of the CPLD will increase but this can be overcome by using a CPLD of higher macrocells such as XC95144, or higher from XC9500 family of Xilinx CPLDs, etc.

REFERENCES

[1] Benjamin Jacoby, *"Walsh Functions: A Digital Fourier series."*

[2] B.J. Falkowski and T. Sasao, *"Unified algorithm to generate Walsh functions in four different orderings and its programmable hardware implementations."*

[3] B.J. Falkowski., *"Recursive relationships, fast transforms, generalisations and VLSI iterative architecture for Gray code ordered Walsh functions."*

[4] B. J. Falkowski, T. Sasao, T. Łuba, Nanyang technological university, Singapore Kyushu institute of technology, Japan ,Warsaw university of technology, Poland, *"Programmable hardware implementation based on Four Walsh sequences."*

[5] Bogdan J. Falkowski and Tsutomu Sasao, *"Implementation of walsh function generator of order 64 Using lut cascades."*

[6] Granlund, fellow, IEEE, a. R. Thompson, member, IEEE, and b. G. Clark, *" An Application of Walsh Functions in Radio Astronomy Instrumentation."*

[7] Jayaram N Chengalur, Yashwant Gupta, K. S. Dwarakanath, *"Low Frequency Radio Astronomy"*, Edition 4.

[8] Volnei A. Pedroni, *Circuit Design with VHDL*, MIT Press, Cambridge, Massachusetts

[9] J. Bhasker, *A VHDL Primer*, Pearson Education.

[10] www.xilinx.com for tutorials about CPLD fitting and XC9500 CPLDS

- CPLD Fitting, Tips and Tricks- http://www.xilinx.com/support/documentation/application_notes/xapp444.pdf
- CPLD user I/O Guide- http://www.xilinx.com/support/documentation/user_guides/ug445.pdf

APPENDIX A

VHDL Source Codes

```
--main source code for interfacing all the components.

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity main_gen is

port(clk,enbw,enbm:in std_logic;
    --clk:external clock input to the circuit.
       --enbw is the enable bit from MCM card for walsh latch.
       --enbm is the enable bit from MCM cardfor monitoring latch.
     dat1,dat2,dat3,dat4,dat5,dat6,dat7,dat8,dat9,dat10:in std_logic;
        --control bits from MCM card.
       DS:in std_logic_vector(0 to 4);
        --antenna selection bits.
        monc:out std_logic;
        --buffered monitoring output.
        MON:out std_logic_vector(0 to 5);
        --monitoring outputs
        wng5:out std_logic;
        --data bit used in line line driver circuitry.
        NGN,WP1,WP2,SEQ:out std_logic);
        --patterns
end main_gen;

architecture Behavioral of main_gen is
--frequency counter for dividing incoming clock frequency by 4096.
component freq_counter_master is
port(clk:in std_logic;
```

```
 master_clk:out std_logic);
end component freq_counter_master;

--latch for group 7 selection for walsh, noise and sequency pattern generation.
component latch_gen is
port(enable:in std_logic;
      dat1,dat2,dat3,dat4,dat5,dat6,dat7,dat8,dat9,dat10: in std_logic;
            data_out: out std_logic_vector(0 to 9));
end component latch_gen;

--frequency counter for dividing the incoming clock frequency by 2 or 4 or 8.
--DAT9,DAT10 ARE THE CONTROL BITS FOR SELECTING CLOCK FREQUENCY FOR WALSH PATTERNS AND SEQUENCY GENERATION.
--DAT3,DAT4 ARE THE CONTROL BITS FOR SELECTING CLOCK FREQUENCY FOR NOISE PATTERN GENARATION.
component freq_counter is
port(clk:in std_logic;
     wng_bit:in std_logic_vector(0 to 1);
     clk_wng:out std_logic);
end component freq_counter;

--latch for group 6 selection for monitoring outputs generation.
component latch_gen2 is
port (enable:in std_logic;
      dat1,dat2,dat3,dat4,dat5,dat6: in std_logic;
            data_out: out std_logic_vector(0 to 5));
end component latch_gen2;

--walsh patterns generator.
component walsh_gen is
port(clk:in std_logic;
     cal:out std_logic_vector(1 to 62));
end component walsh_gen;

--sequency pattern generator.
component sequency_gen is
port(clk:in std_logic;
     SEQ:out std_logic);
```

```
end component sequency_gen;

--noise patterns generator.
--DAT1,DAT2 ARE FOR SELECTING THE NOISE PATTERS:ON,OFF,25%,50%
component noise_gen is
port(clk,wng1,wng2:in std_logic;
        NGN:out std_logic);
end component noise_gen;

--multiplexer for selecting walsh patterns on independent channels.
--DS0,DS1,DS2,DS3,DS4 ARE THE ANTENNA SELECTION BITS.
--DAT6,DAT7,DAT8 ARE THE MUX. SELECTION BITS FOR OBTAINING THE WALSH
PATTERNS ON INDEPENDENT CHANNEL FOR A PARTICULAR ANTENNA.
component  mux1 is
port (clk:in std_logic;
            wng_mux:in std_logic_vector(0 to 2);
            DS: in std_logic_vector(0 to 4);
      wp1,wp2:out std_logic;
            w:in std_logic_vector(1 to 62));
end component mux1;

signal clk_wng,clk_ngn,master_clk:std_logic;
signal wng:std_logic_vector(9 downto 0);
signal cal_patterns:std_logic_vector(1 to 62 );
signal mon_test:std_logic_vector(0 to 5);

begin
wng5<=wng(4);
mon<=mon_test;
monc<=mon_test(5) after 30 ns;--buffering mon5 to generate monc after a
30 ns delay.

--frequency counter for frequency division by 2^12.
COUNTER_MASTER:freq_counter_master port map(clk,master_clk);
--walsh latch unit.
WNG_LATCH:latch_gen port map(enbw,
dat1,dat2,dat3,dat4,dat5,dat6,dat7,dat8,dat9,dat10,wng);

--frequency counter for division of incoming clock freq. by 2 or 4 or 8
for walsh and sequency patterns generation.
```

```
COUNT_WNG:freq_counter port map(master_clk,wng(8 downto 0),clk_wng);

--monitoring latch unit.
WNG_MON:latch_gen2                                                    port
map(enbm,dat1,dat2,dat3,dat4,dat5,dat6,mon_test);

--walsh patterns generator unit.
WAL1: walsh_gen port map (clk_wng,cal_patterns);

--sequency pattern generation unit.
SEQ1:sequency_gen port map (clk_wng,seq);

--frequency counter  division of incoming clock freq. by 2 or 4 or 8 for
noise patterns.
COUNT_NGN: freq_counter port map(master_clk,wng(3 downto 2),clk_ngn);

--noise patterns generator unit.
NGN1:noise_gen port map (clk_ngn,wng(1),wng(0),NGN);

--32:1 multiplexers unit for obtaining walsh patterns on independent
channels.
MUX:mux1 port map(clk_wng,wng(7 downto 0),ds,wp1,wp2,cal_patterns);

end behavioral;
```

****

```
--FREQUENCY COUNTER UNIT: DIVIDES THE INCOMING CLOCK FREQUENCY BY 2^12.
--IT CONTAINS A 12-BIT COUNTER FOR COUNTING FROM 0 TO 4095.
--THE OUTPUT FROM THIS COUNTER BECOMES THE MASTER CLOCK FOR REST OF THE
CIRCUIT.

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity freq_counter_master is
port(clk:in std_logic;
        master_clk:out std_logic);
end freq_counter_master;

architecture Behavioral of freq_counter_master is
begin
process(clk)
variable count_master:integer range 0 to 31;
begin
if (clk='1' and clk'event) then
if (count_master<16) then
master_clk<='1'; else master_clk<='0';
end if;
count_master:=count_master+1;

if (count_master=32) then
count_master:=0;
end if;
end if;
end process;
end Behavioral;
```

****

```
--FREQUENCY COUNTER UNIT: DIVIDES THE CLOCK (FROM FIRST FREQUENCY
COUNTER) BY 2^2 OR 2^4 OR 2^3 OR NONE DEPENDING ON THE CONTROL BITS.
--IT CONTAINS A COUNTER FOR COUNTING FROM 0 TO 7 AND A 4:1 MULTIPLEXER
FOR SELECTING DESIRED FREQUENCY DIVISION.

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity freq_counter is
port(clk:in std_logic;
```

```
wng_bit:in std_logic_vector(0 to 1);
     clk_wng:out std_logic);
end freq_counter;

architecture Behavioral of freq_counter is
signal count:std_logic_vector(2 downto 0);
begin
process(clk)
variable clkv:std_logic;
begin

if (clk'event and clk='1') then
if count<7 then count<=count+1;
else count<="000";
end if;
end if;
case wng_bit is
when "00"=> clkv:=clk;clk_wng<=clkv;--THE INCOMING CLOCK FREQUENCY
                                     REMAINS UNCHANGED.
when "01"=> clk_wng<=count(0);-INCOMING CLOCK FREQUENCY IS DIVIDED BY 2.
when "10"=> clk_wng<=count(1);-INCOMING CLOCK FREQUENCY IS DIVIDED BY 4.
when "11"=> clk_wng<=count(2);-INCOMING CLOCK FREQUENCY IS DIVIDED BY 8.
when others=> clk_wng<='0';-THE CLOCK IS DISABLED FOR ANY OTHER
                            COMBINATION OF THE CONTROL BITS.
end case;
end process;
end Behavioral;
                                ***************
--MONITOR LATCH UNIT. FOR LOW TO HIGH TRANSITION OF THE ENABLE INPUT,
THE INCOMING DATA BITS ARE LATCHED OUT.
--THE OUTPUT BITS ARE USED FOR MONITORING.

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
```

```
--use UNISIM.VComponents.all;

entity latch_gen2 is
port (enable:in std_logic;
      dat1,dat2,dat3,dat4,dat5,dat6: in std_logic;
            data_out: out std_logic_vector(0 to 5));
end latch_gen2;

architecture Behavioral of latch_gen2 is
begin
process(enable)
begin
if (enable='1' and enable'event) then
data_out(0)<=dat1;
data_out(1)<=dat2;
data_out(2)<=dat3;
data_out(3)<=dat4;
data_out(4)<=dat5;
data_out(5)<=dat6;
else null;end if;
end process;
end Behavioral;
```

****

```
--WALSH LATCH UNIT:
-- FOR LOW TO HIGH TRANSITION OF THE ENABLE INPUT, THE INCOMING DATA BITS ARE LATCHED OUT.
--THE OUTPUT BITS ARE USED AS CONTROLS FOR VARIOUS PATTERNS GENARTION AND FRQUENCY SELECTION.

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity latch_gen is
```

```
port (enable:in std_logic;
       dat1,dat2,dat3,dat4,dat5,dat6,dat7,dat8,dat9,dat10: in std_logic;
            data_out: out std_logic_vector(9 downto 0));
end latch_gen;

architecture Behavioral of latch_gen is
begin
process(enable)
begin
if rising_edge(enable) then
data_out(0)<=dat1;
data_out(1)<=dat2;
data_out(2)<=dat3;
data_out(3)<=dat4;
data_out(4)<=dat5;
data_out(5)<=dat6;
data_out(6)<=dat7;
data_out(7)<=dat8;
data_out(8)<=dat9;
data_out(9)<=dat10;
else null;
end if;
end process;
end Behavioral;
```

****

```
--WALSH PATTERN GENERATOR UNIT.
--THE PROGRAM GENERATES ALL THE 128 WALSH PATTERNS BUT ONLY THE CAL
OUTPUTS ARE TAKEN AT THE OUTPUT.
--CAL PATTERNS 0 AND 63 ARE NOT USED.

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity walsh_gen is
port(clk:in std_logic;
     cal:out std_logic_vector(1 to 62));
 end walsh_gen;

architecture Behavioral of walsh_gen is
signal qn1,qn3,qn7,qn15,qn31,qn63,qn127:std_logic;
begin
process(clk,qn1,qn3,qn7,qn15,qn31,qn63,qn127)
variable q1,q3,q7,q15,q31,q63,q127:std_logic:='0';
variable w:std_logic_vector(124 downto 2);

begin
--TOGGLE     FLIP-FLOPS     WHICH     GENERATE     THE     CONTROL     WALSH
PATTERNS:WAL(1),WAL(3),WAL(7),WAL(15),WAL(31),WAL(63),WAL(127)

if (clk='1' and clk'event) then
q127:=not q127;
qn127<=q127;
end if;

if (qn127='1' and qn127'event) then
q63:=not q63;
qn63<=q63;
end if;

if (qn63='1' and qn63'event) then
q31:=not q31;
qn31<=q31;
end if;

if (qn31='1' and qn31'event) then
q15:=not q15;
qn15<=q15;
end if;

if (qn15='1' and qn15'event) then
q7:=not q7;
qn7<=q7;
end if;
```

```
if (qn7='1' and qn7'event) then
q3:=not q3;
qn3<=q3;
end if;

if (qn3='1' and qn3'event) then
q1:=not q1;
qn1<=q1;
end if;
--w(1):=qn1;                          --WAL(1)
--w(3):=qn3;                                    --WAL(3)
--w(7):=qn7;                                       --WAL(7)
--w(15):=qn15;                                     --WAL(15)
--w(31):=qn31;                                     --WAL(31)
--w(63):=qn63;                                     --WAL(63)
--w(127):=qn127;                                 --WAL(127)
w(2):=not(qn1 xor qn3);                        --CAL(1)
w(4):=not(qn3 xor qn7);                        --CAL(2)
w(5):=(not w(2)) xor qn7;
w(6):=not(qn1 xor qn7);                        --CAL(3)
w(8):=not(qn7 xor qn15);                       --CAL(4)
w(9):=(not w(6) )xor qn15;
w(10):=not(w(5) xor qn15);                     --CAL(5)
w(11):=(not w(4)) xor qn15;
w(12):=not(qn3 xor qn15);                      --CAL(6)
w(13):=(not w(2)) xor qn15;
w(14):=not (qn1 xor qn15);                     --CAL(7)
w(16):=not(qn15 xor qn31);                     --CAL(8)
w(17):=(not w(14) )xor qn31;
w(18):=not(w(13) xor qn31);              --CAL(9)
w(19):=(not w(12)) xor qn31;
w(20):=not(w(11) xor qn31);              --CAL(10)
w(21):=(not w(10)) xor qn31;
w(22):=not (w(9) xor qn31);              --CAL(11)
w(23):=(not w(8) )xor qn31;
w(24):=not(qn7 xor qn31);                      --CAL(12)
w(25):=(not w(6)) xor qn31;
w(26):=not(w(5) xor qn31);                     --CAL(13)
w(27):=(not w(4)) xor qn31;
w(28):=not (qn3 xor qn31);                     --CAL(14)
```

```
w(29):=(not w(2) )xor qn31;
w(30):=not(qn1 xor qn31);                    --CAL(15)
w(32):=not(qn31 xor qn63);                   --CAL(16)
w(33):=(not w(30) )xor qn63;
w(34):=not(w(29) xor qn63);                  --CAL(17)
w(35):=(not w(28)) xor qn63;
w(36):=not(w(27) xor qn63);                  --CAL(18)
w(37):=(not w(26)) xor qn63;
w(38):=not (w(25) xor qn63);                 --CAL(19)
w(39):=(not w(24) )xor qn63;
w(40):=not(w(23) xor qn63);                  --CAL(20)
w(41):=(not w(22)) xor qn63;
w(42):=not(w(21) xor qn63);                  --CAL(21)
w(43):=(not w(20)) xor qn63;
w(44):=not (w(19) xor qn63);                 --CAL(22)
w(45):=(not w(18)) xor qn63;
w(46):=not(w(17) xor qn63);                  --CAL(23)
w(47):=(not w(16) )xor qn63;
w(48):=not(qn15 xor qn63);                   --CAL(24)
w(49):=(not w(14)) xor qn63;
w(50):=not(w(13) xor qn63);                  --CAL(25)
w(51):=(not w(12)) xor qn63;
w(52):=not (w(11) xor qn63);                 --CAL(26)
w(53):=(not w(10) )xor qn63;
w(54):=not(w(9) xor qn63);                   --CAL(27)
w(55):=(not w(8)) xor qn63;
w(56):=not(qn7 xor qn63);                    --CAL(28)
w(57):=(not w(6)) xor qn63;
w(58):=not (w(5) xor qn63);                  --CAL(29)
w(59):=(not w(4)) xor qn63;
w(60):=not(qn3 xor qn63);                    --CAL(30)
w(61):=(not w(2) )xor qn63;
w(62):=not(qn1 xor qn63);                    --CAL(31)
w(64):=not(qn63 xor qn127);                  --CAL(32)
--w(65):=(not w(62) )xor qn127;
w(66):=not(w(61) xor qn127);                 --CAL(33)
--w(67):=(not w(60)) xor qn127;
w(68):=not(w(59) xor qn127);                 --CAL(34)
--w(69):=(not w(58)) xor qn127;
w(70):=not (w(57) xor qn127);                --CAL(35)
```

```
--w(71):=(not w(56) )xor qn127;
w(72):=not(w(55) xor qn127);              --CAL(36)
--w(73):=(not w(54)) xor qn127;
w(74):=not(w(53) xor qn127);              --CAL(37)
--w(75):=(not w(52)) xor qn127;
w(76):=not (w(51) xor qn127);             --CAL(38)
--w(77):=(not w(50) )xor qn127;
w(78):=not(w(49) xor qn127);              --CAL(39)
--w(79):=(not w(48) )xor qn127;
w(80):=not(w(47) xor qn127);              --CAL(40)
--w(81):=(not w(46)) xor qn127;
w(82):=not(w(45) xor qn127);              --CAL(41)
--w(83):=(not w(44)) xor qn127;
w(84):=not (w(43) xor qn127);             --CAL(42)
--w(85):=(not w(42) )xor qn127;
w(86):=not(w(41) xor qn127);              --CAL(43)
--w(87):=(not w(40)) xor qn127;
w(88):=not(w(39) xor qn127);              --CAL(44)
--w(89):=(not w(38)) xor qn127;
w(90):=not (w(37) xor qn127);             --CAL(45)
--w(91):=(not w(36)) xor qn127;
w(92):=not(w(35) xor qn127);              --CAL(46)
--w(93):=(not w(34) )xor qn127;
w(94):=not(w(33) xor qn127);              --CAL(47)
--w(95):=(not w(32) )xor qn127;
w(96):=not(qn31 xor qn127);               --CAL(48)
--w(97):=(not w(30) )xor qn127;
w(98):=not(w(29) xor qn127);              --CAL(49)
--w(99):=(not w(28)) xor qn127;
w(100):=not(w(27) xor qn127);             --CAL(50)
--w(101):=(not w(26)) xor qn127;
w(102):=not (w(25) xor qn127);            --CAL(51)
--w(103):=(not w(24) )xor qn127;
w(104):=not(w(23) xor qn127);             --CAL(52)
--w(105):=(not w(22)) xor qn127;
w(106):=not(w(21) xor qn127);             --CAL(53)
--w(107):=(not w(20)) xor qn127;
w(108):=not (w(19) xor qn127);            --CAL(54)
--w(109):=(not w(18)) xor qn127;
w(110):=not(w(17) xor qn127);             --CAL(55)
```

```
--w(111):=(not w(16) )xor qn127;
w(112):=not(qn15 xor qn127);              --CAL(56)
--w(113):=(not w(14)) xor qn127;
w(114):=not(w(13) xor qn127);             --CAL(57)
--w(115):=(not w(12)) xor qn127;
w(116):=not (w(11) xor qn127);            --CAL(58)
--w(117):=(not w(10) )xor qn127;
w(118):=not(w(9) xor qn127);              --CAL(59)
--w(119):=(not w(8)) xor qn127;
w(120):=not(qn7 xor qn127);               --CAL(60)
--w(121):=(not w(6)) xor qn127;
w(122):=not (w(5) xor qn127);             --CAL(61)
--w(123):=(not w(4)) xor qn127;
w(124):=not(qn3 xor qn127);               --CAL(62)
--w(125):=(not w(2)) xor qn127;
--w(126):=not(qn1 xor qn127);             --CAL(63)
--w(0):='1';                                          --CAL(0)
for i in 1 to 62 loop
cal(i)<=w(2*i);
end loop;
end process;
end Behavioral;
```

****

```
--THE PROGRAM CONSISTS OF FOUR 32:1 MULTIPLEXERS FOR OBTAINING CAL
PATTERNS ON INDIVIDUAL CHANNELS .
--THERE ARE 5 ANTENNA SELECTION BITS FOR EACH OF THE MULTIPLEXER.
--THERE ARE 3 CONTROL BITS FOR SELECTING THE MULTIPLEXER FOR CHANNELS
SELECTION.

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

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

entity mux1 is
```

```
port (clk:in std_logic;
            wng_mux:in std_logic_vector(0 to 2);
            DS: in std_logic_vector(0 to 4);
      wp1,wp2:out std_logic;
            w:in std_logic_vector(1 to 62));
end mux1;

architecture Behavioral of mux1 is
begin
process(clk,w)
begin
if (clk='1' and clk'event)     then
case wng_mux is
when "000"=>
wp1<='0';wp2<='0';

when "001"=>
wp2<='0';
case DS is
when "00000"=> wp1<=w(1);
when "00001"=> wp1<=w(2);
when "00010"=> wp1<=w(3);
when "00011"=> wp1<=w(4);
when "00100"=> wp1<=w(5);
when "00101"=> wp1<=w(6);
when "00110"=> wp1<=w(7);
when "00111"=> wp1<=w(8);
when "01000"=> wp1<=w(9);
when "01001"=> wp1<=w(10);
when "01010"=> wp1<=w(11);
when "01011"=> wp1<=w(12);
when "01100"=> wp1<=w(13);
when "01101"=> wp1<=w(14);
when "01110"=> wp1<=w(15);
when "01111"=> wp1<=w(16);
when "10000"=> wp1<=w(17);
when "10001"=> wp1<=w(18);
when "10010"=> wp1<=w(19);
when "10011"=> wp1<=w(20);
when "10100"=> wp1<=w(21);
```

```
when "10101"=> wp1<=w(22);
when "10110"=> wp1<=w(23);
when "10111"=> wp1<=w(24);
when "11000"=> wp1<=w(25);
when "11001"=> wp1<=w(26);
when "11010"=> wp1<=w(27);
when "11011"=> wp1<=w(28);
when "11100"=> wp1<=w(29);
when "11101"=> wp1<=w(30);
when "11110"=> wp1<=w(31);
when others=> wp1<='0';
end case;

when "010"=>
wp1<='0';
case DS is
when "00000"=> wp2<=w(32);
when "00001"=> wp2<=w(33);
when "00010"=> wp2<=w(34);
when "00011"=> wp2<=w(35);
when "00100"=> wp2<=w(36);
when "00101"=> wp2<=w(37);
when "00110"=> wp2<=w(38);
when "00111"=> wp2<=w(39);
when "01000"=> wp2<=w(40);
when "01001"=> wp2<=w(41);
when "01010"=> wp2<=w(42);
when "01011"=> wp2<=w(43);
when "01100"=> wp2<=w(44);
when "01101"=> wp2<=w(45);
when "01110"=> wp2<=w(46);
when "01111"=> wp2<=w(47);
when "10000"=> wp2<=w(48);
when "10001"=> wp2<=w(49);
when "10010"=> wp2<=w(50);
when "10011"=> wp2<=w(51);
when "10100"=> wp2<=w(52);
when "10101"=> wp2<=w(53);
when "10110"=> wp2<=w(54);
```

```
when "10111"=> wp2<=w(55);
when "11000"=> wp2<=w(56);
when "11001"=> wp2<=w(57);
when "11010"=> wp2<=w(58);
when "11011"=> wp2<=w(59);
when "11100"=> wp2<=w(60);
when "11101"=> wp2<=w(61);
when "11110"=> wp2<=w(62);
when others=> wp2<='0';
end case;

when "011"=>
case DS is
when "00000"=> wp1<=w(1);wp2<=w(32);
when "00001"=> wp1<=w(2);wp2<=w(33);
when "00010"=> wp1<=w(3);wp2<=w(34);
when "00011"=> wp1<=w(4);wp2<=w(35);
when "00100"=> wp1<=w(5);wp2<=w(36);
when "00101"=> wp1<=w(6);wp2<=w(37);
when "00110"=> wp1<=w(7);wp2<=w(38);
when "00111"=> wp1<=w(8);wp2<=w(39);
when "01000"=> wp1<=w(9);wp2<=w(40);
when "01001"=> wp1<=w(10);wp2<=w(41);
when "01010"=> wp1<=w(11);wp2<=w(42);
when "01011"=> wp1<=w(12);wp2<=w(43);
when "01100"=> wp1<=w(13);wp2<=w(44);
when "01101"=> wp1<=w(14);wp2<=w(45);
when "01110"=> wp1<=w(15);wp2<=w(46);
when "01111"=> wp1<=w(16);wp2<=w(47);
when "10000"=> wp1<=w(17);wp2<=w(48);
when "10001"=> wp1<=w(18);wp2<=w(49);
when "10010"=> wp1<=w(19);wp2<=w(50);
when "10011"=> wp1<=w(20);wp2<=w(51);
when "10100"=> wp1<=w(21);wp2<=w(52);
when "10101"=> wp1<=w(22);wp2<=w(53);
```

```
when "10110"=> wp1<=w(23);wp2<=w(54);
when "10111"=> wp1<=w(24);wp2<=w(55);
when "11000"=> wp1<=w(25);wp2<=w(56);
when "11001"=> wp1<=w(26);wp2<=w(57);
when "11010"=> wp1<=w(27);wp2<=w(58);
when "11011"=> wp1<=w(28);wp2<=w(59);
when "11100"=> wp1<=w(29);wp2<=w(60);
when "11101"=> wp1<=w(30);wp2<=w(61);
when "11110"=> wp1<=w(31);wp2<=w(62);
when others=> wp2<='0';wp1<='0';
end case;

when "100"=>
case DS is
when "00000"=> wp1<=w(1);wp2<=w(1);
when "00001"=> wp1<=w(2);wp2<=w(2);
when "00010"=> wp1<=w(3);wp2<=w(3);
when "00011"=> wp1<=w(4);wp2<=w(4);
when "00100"=> wp1<=w(5);wp2<=w(5);
when "00101"=> wp1<=w(6);wp2<=w(6);
when "00110"=> wp1<=w(7);wp2<=w(7);
when "00111"=> wp1<=w(8);wp2<=w(8);
when "01000"=> wp1<=w(9);wp2<=w(9);
when "01001"=> wp1<=w(10);wp2<=w(10);
when "01010"=> wp1<=w(11);wp2<=w(11);
when "01011"=> wp1<=w(12);wp2<=w(12);
when "01100"=> wp1<=w(13);wp2<=w(13);
when "01101"=> wp1<=w(14);wp2<=w(14);
when "01110"=> wp1<=w(15);wp2<=w(15);
when "01111"=> wp1<=w(16);wp2<=w(16);
when "10000"=> wp1<=w(17);wp2<=w(17);
when "10001"=> wp1<=w(18);wp2<=w(18);
when "10010"=> wp1<=w(19);wp2<=w(19);
when "10011"=> wp1<=w(20);wp2<=w(20);
when "10100"=> wp1<=w(21);wp2<=w(21);
when "10101"=> wp1<=w(22);wp2<=w(22);
when "10110"=> wp1<=w(23);wp2<=w(23);
when "10111"=> wp1<=w(24);wp2<=w(24);
when "11000"=> wp1<=w(25);wp2<=w(25);
when "11001"=> wp1<=w(26);wp2<=w(26);
```

```
when "11010"=> wp1<=w(27);wp2<=w(27);
when "11011"=> wp1<=w(28);wp2<=w(28);
when "11100"=> wp1<=w(29);wp2<=w(29);
when "11101"=> wp1<=w(30);wp2<=w(30);
when "11110"=> wp1<=w(31);wp2<=w(31);
when others=> wp2<='0';wp1<='0';
end case;

when "101"=>
case DS is
when "00000"=> wp1<=w(32);wp2<=w(32);
when "00001"=> wp1<=w(33);wp2<=w(33);
when "00010"=> wp1<=w(34);wp2<=w(34);
when "00011"=> wp1<=w(35);wp2<=w(35);
when "00100"=> wp1<=w(36);wp2<=w(36);
when "00101"=> wp1<=w(37);wp2<=w(37);
when "00110"=> wp1<=w(38);wp2<=w(38);
when "00111"=> wp1<=w(39);wp2<=w(39);
when "01000"=> wp1<=w(40);wp2<=w(40);
when "01001"=> wp1<=w(41);wp2<=w(41);
when "01010"=> wp1<=w(42);wp2<=w(42);
when "01011"=> wp1<=w(43);wp2<=w(43);
when "01100"=> wp1<=w(44);wp2<=w(44);
when "01101"=> wp1<=w(45);wp2<=w(45);
when "01110"=> wp1<=w(46);wp2<=w(46);
when "01111"=> wp1<=w(47);wp2<=w(47);
when "10000"=> wp1<=w(48);wp2<=w(48);
when "10001"=> wp1<=w(49);wp2<=w(49);
when "10010"=> wp1<=w(50);wp2<=w(50);
when "10011"=> wp1<=w(51);wp2<=w(51);
when "10100"=> wp1<=w(52);wp2<=w(52);
when "10101"=> wp1<=w(53);wp2<=w(53);
when "10110"=> wp1<=w(54);wp2<=w(54);
when "10111"=> wp1<=w(55);wp2<=w(55);
when "11000"=> wp1<=w(56);wp2<=w(56);
when "11001"=> wp1<=w(57);wp2<=w(57);
when "11010"=> wp1<=w(58);wp2<=w(58);
when "11011"=> wp1<=w(59);wp2<=w(59);
when "11100"=> wp1<=w(60);wp2<=w(60);
when "11101"=> wp1<=w(61);wp2<=w(61);
```

```
when "11110"=> wp1<=w(62);wp2<=w(62);
when others=> wp2<='0';wp1<='0';
end case;

when others=>
wp1<='0';wp2<='0';
end case;
end if;
end process;
end Behavioral;
```

****

```
--SEQUENCY PATTERN GENERATION.
--THE PROGRAM CONSIST OF A COUNTER WHICH COUNTS FROM 0 TO 127.
--FOR FIRST COUNT THE OUTPUT IS HIGH AND FOR REMAING 127 COUNTS, THE
OUTPUT IS LOW.

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity sequency_gen is
port(clk:in std_logic;
        SEQ:out std_logic);
end sequency_gen;

architecture Behavioral of sequency_gen is
signal countseq:integer range 0 to 127;
begin
process(clk)
begin
if (clk='1' and clk'event) then
if (countseq<1) then
seq<='1'; else seq<='0';
end if;
```

```
countseq<=countseq+1;

if (countseq=127) then
countseq<=0;
end if;end if;
end process;
end Behavioral;
```

****

```
--NOISE PATTERNS GENERATION UNIT.
--THE PROGRAM CONSISTS OF A COUNTER WHICH DIVIDES THE INCOMING CLOCK FREQUENCY BY 4.
--FOR NOISE ON, THE OUTPUT NGN PATTERNS IS ALWAYS HIGH.
--FOR NOISE OFF, THE OUTPUT NOISE PATTERN IS ALWAYS LOW.
--FOR NOISE 25%/75%, THE OUTPUT PATTERN IS HIGH FOR FIRST COUNT AND LOW FOR REMAINING 3 COUNTS;
--FOR NOISE 50%, THE OUTPUT PATTERN IS HIGH FOR FIRST TWO COUNTS AND LOW FOR REMAINING 2 COUNTS;
--THERE IS A 4:1 MUX FOR SELECTING EACH OF THE PATTERNS.

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity noise_gen is
port(clk,wng1,wng2:in std_logic;
        NGN:out std_logic);
end noise_gen;

architecture Behavioral of noise_gen is
begin
process(clk,wng1,wng2)
variable countngn2,countngn3:integer range 0 to 3;
begin
if (clk='1' and clk'event) then
```

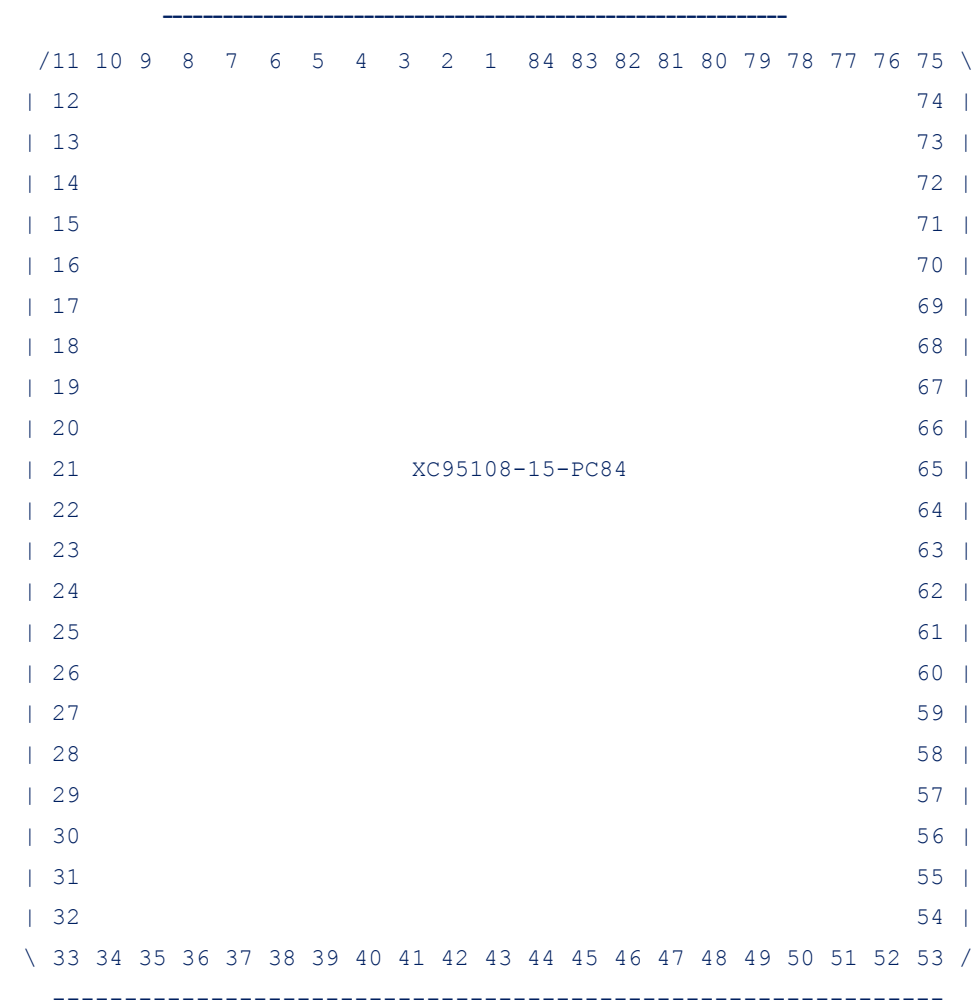
```
if (wng1='0' and wng2='1') then
countngn3:=0;
if (countngn2<1) then
ngn<='1'; else ngn<='0';
end if;countngn2:=countngn2+1;
if( countngn2=4) then
countngn2:= 0 ;
end if;end if;

if (wng1='1' and wng2='0') then
countngn2:=0;
if (countngn3<2) then
ngn<='1'; else ngn<='0';
end if;
countngn3:=countngn3+1;
if( countngn3=4) then
countngn3:= 0 ;
end if; end if;

if (wng1='0' and wng2='0') then
ngn<='0';
end if;
if (wng1='1' and wng2='1') then
ngn<='1';
end if;end if;
end process;
end Behavioral;
```

****

APPENDIX B

Device Pin Out

Device : XC95108-15-PC84



Pin No.	Signal Name	Pin No.	Signal Name
1	dat2	43	TIE
2	dat1	44	TIE
3	DS<4>	45	WP2
4	DS<3>	46	TIE
5	DS<2>	47	TIE
6	DS<1>	48	TIE
7	DS<0>	49	GND
8	GND	50	TIE
9	dat10	51	TIE
10	clk	52	TIE
11	TIE	53	TIE

12	TIE	54	TIE
13	MON<0>	55	TIE
14	MON<1>	56	TIE
15	MON<2>	57	TIE
16	GND	58	TIE
17	MON<3>	59	TDO
18	MON<4>	60	GND
19	MON<5>	61	TIE
20	TIE	62	NGN
21	TIE	63	SEQ
22	VCC	64	VCC
23	TIE	65	TIE
24	TIE	66	TIE
25	TIE	67	TIE
26	TIE	68	TIE
27	GND	69	TIE
28	TDI	70	TIE
29	TMS	71	TIE
30	TCK	72	TIE
31	wng5	73	VCC
32	monc	74	TIE
33	WP1	75	enbm
34	TIE	76	enbw
35	TIE	77	dat9
36	TIE	78	VCC
37	TIE	79	dat8
38	VCC	80	dat7
39	TIE	81	dat6
40	TIE	82	dat5
41	TIE	83	dat4
42	GND	84	dat3

```
Legend :  NC  = Not Connected, unbonded pin
         PGND = Unused I/O configured as additional Ground pin
         TIE  = Unused I/O floating -- must tie to VCC, GND or other signal
         VCC  = Dedicated Power Pin
         GND  = Dedicated Ground Pin
         TDI  = Test Data In, JTAG pin
         TDO  = Test Data Out, JTAG pin
         TCK  = Test Clock, JTAG pin
         TMS  = Test Mode Select, JTAG pin
  PROHIBITED  = User reserved pin
```

APPENDIX C

Data sheets links

- Data sheet for XC95108 PC84C-
  http://www.xilinx.com/support/documentation/data_sheets/DS066.pdf

- Data sheet for MC3487-
  http://www.datasheetcatalog.org/datasheet/texasinstruments/mc3487.pdf

- Data sheet for crystal Oscillator (Andhra Electronics)-
  http://www.andhraelec.com/httpdocs/catalogue/CXO%20300.pdf

- Data Sheet for voltage Regulator LM317-
  http://www.national.com/ds/LM/LM117.pdf