COMMON FUNCTIONS LIBRARY

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering

in

Information Security

Submitted By

Jasdeep Singh

(Roll No. 801333008)

Under the supervision of:

Er. Sumit Sharma

Senior Verification Engineer

Dr. Jhilik Bhattacharya

Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

July 2015

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*Common Functions Library*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Information Security* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Er. Sumit Sharma and Dr. Jhilik Bhattacharya* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Jasdeep Singh

Jasdeep Singh

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Sumit

Sumit Sharma

(Sr. Verification Engineer, STM)

Dr. Jhilik Bhattacharya

(Assistant Professor, CSED)

Countersigned by

(Dr. Deepak Garg)

Head

Computer Science and Engineering Department

Thapar University

Patiala

(Dr. S. S. Bhatia)

Dean (Academic Affairs)

Thapar University

Patiala

ACKNOWLEDGEMENT

I feel pleased to have an opportunity to show my sincere gratitude towards everyone who supported me in completing the dissertation. There invaluable support helped me out and keep me motivated.

Firstly, I would like to express my genuine pleasure towards **Dr. Deepak Garg,** HoD CSED,TU Patiala to provide an opportunity to do my thesis work at STMicroelectronics India Pvt. Ltd. His priceless academic psychological support motivated me to complete the thesis along with having industrial experience.

I am extremely thankful to my guide **Dr. Jhilik Bhattacharya**, Assistant professor TU Patiala. It is a privilege to have such a supporting guide. Even her small suggestion means a lot to me. She is the one who directed me to choose the right path. Her kindness, calm-tempered nature, inspiration and timely suggestions encouraged me to complete this thesis.

I heartfully thanks to my mentor **Er. Sumit Sharma**, Senior Verification Manager, STMicroelectronics India Pvt. Ltd. for giving me the opportunity to work under his guidance for the research and development work for the project CFLib. His innovative and motivating attitude encouraged me to do the tasks more efficiently and effectively. He always corrected my mistakes politely and effortlessly.

A Special thanks to **Er. Vijay Kumar**, Senior Verification Engineer, STMicroelectronics India Pvt. Ltd. I loved working with him. I want to thank him for showing confidence in my work, his patience and his help. I am also grateful towards every member of CPU/GPU verification team under the guidance of experienced, calm and fun loving manager **Er. Pankaj Jain.**

Finally, I would like to acknowledge the love and support of my family. They kept me motivated. This thesis would not have been possible without their support.

ABSTRACT

Every programming language has an in-built function library which provides functions those can be used without being concerned about the internal operations. Similarly "*Common Functions Library"* also provides such facility to its users. A detailed research work is done in developing this static library. It is a well-structured and well-documented library with integrating similar functions along with their proper description in same files. This project is developed to implement the modular and convenient approach in writing the verification tests for newly designed embedded systems. Deploying CFLib in a centralized database can help the verification engineers to easily understand and use the functions in it. Also, CFLib helps in removing redundancy as multiple copies of the same code will not exist on the centralized database. This project is delivered as a directory containing header files and a library file as an archive file. This archive file was created using "armar" from the object files by compiling the 'C' source code using "armcc" and assembly language code using "armasm".

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The research work is carried out for the development of “Common Functions Library” to ease the process of writing verification tests for different projects. It includes functions compatible to various ARM cores (i.e., Cortex A5, A7, A9, A15 and A53) and provides a static function library to be included at the linking time. The functions in this project are grouped into files according to their functionality. Description of each and every function is provided with the declaration/prototype in the header files. The functions in CFLib are written in ‘C’ language and compiled using ARMCC i.e. ARM Compiler. The standard C, C++ code is compiled using ARMCC to give machine code for the ARM architecture-based processors. For accessing the registers in the AMR processors, ARMASM i.e. ARM Assembler is needed to be invoked to compile the low level assembly code. Also, after compilation of the code, i.e. after getting ‘*.o’ (object) files, these files are clubbed to give an achieve library file in ‘*.a’ format using ARMAR i.e. ARM librarian.

The motive for which this research work is done is to create new code, remove redundancy of the code, ease of use, and decompose the large program into easily manageable and understandable code segments. Each programmer has to go through this library while writing a new test. If there is a function already written, the programmer can use it and otherwise may contact the CFLib developer to add the required functionality in CFLib. Also, test written prior to the creation of CFLib are also revisited for integrating the CFLib into them to maintain the uniformity.

CFLib is a tool which helps in the verification of the design of a SoC or an embedded system developed using ARM cores. This is done by accessing memory, various registers, co-processor registers etc. So, before having a detailed description of CFLib, below is the brief introduction to the terms that will be used throughout the thesis.

1.1 System-on-Chip (SoC)

Semiconductor companies such as STM take the integrated circuits from semiconductor IP design and verifying companies and include these processors in the and many other components like memory controller, on chip memory and peripherals etc. to design a full-fledged system-on-chip. SoC is a device with higher degree of integration. Most of the

parts of system are included on a single chip, and may include digital, analog, mixed-signals or radio frequency circuits. The term SoC is not entirely correct as the whole system cannot be integrated on a single chip. For e.g., significant amount of memory is required for running operating systems like linux which is more than the memory that can be possible on a SoC.

1.2 Embedded systems

Embedded systems are the systems designed to do a specific task and have a dedicated hardware to for running specific software. For example TV set-top box, washing machines, routers and smartcards etc. On the other hand a general purpose system is able to run different kind of softwares and requires input and output units. For developing an embedded system, the following points are considered:

Memory: Memory size can be limited to minimize the cost

Real-time Behaviour: How critical are the deadlines to response to external events. The real-time behaviour is further categorized as:

i. **Hard real time:** Deadlines are critical, system might crash on failure to meet the deadlines. For e.g., rocket launching, if deadlines are not met, the system crashes.

ii. **Soft real time:** The deadlines are not critical. If not met, there is no direct harm to the system. For e.g., washing machine, user can reset the timer if cloths are not cleaned in the allotted time duration.

Power Consumption: The most of embedded systems have battery as power source. So, it is a necessity to minimize the overall energy consumption. The energy can be saved by switching off the core when not in used, reducing supply voltage or slowing the clock etc.

Cost: Total cost of the system depends upon many factors such as processor used, memory size, enhanced features etc. As the features increases, the total cost also increases. So, reducing the cost of the system is a significant constraint.

Time to market: The success of a system is also dependent upon the time in which it is launched. A product that is launched first with enhanced features according to the need of the market has a higher probability to be successful.

1.3 ARM Architecture

ARM (Advanced RISC Machine) Limited design microprocessors and provide licences to the other semiconductor companies for integrating them on to the System-on-chip devices. ARM is a leading manufacturer in the industry of designing RISC based microprocessors. Wide range of performance parameters are supported by the ARM architecture. The main reason for the popularity of ARM processors is the low power consumption. As these days, many wireless and handheld devices are used by us in day to day life. The main challenge is the power consumption. So, the known best solution to this problem is the usage of ARM RISC based processors.

Architectural Profile

ARM has released many versions of its microprocessors. Every new version adds some extra features to the previous version and also maintaining the backward compatibility i.e., the code which runs on the previous version can also run on the newer version properly. But the code written for the newer advanced version will not run on the previous version.

The different ARM architecture profiles are:

i. **A**(Application profile): The ARM-A profile is targeted for the design of High performance processors which has a Memory Management Unit and supports a virtual memory system. Thus, the ARM-A profile is capable of running fully featured operating system.

ii. **R** (Real-time profile): ARM-R series processor architecture is designed to meet the timing deadlines and low interrupt latency. ARM-R architectures do not support virtual memory system.

iii. **M** (Microcontroller profile): The microcontroller profile aims at low cost system. ARM-M has an exception handling model which is different from other two profiles and supports only specific instruction set.

Cortex-A series processors

This is a family of ARM multi-core processors. These processors are used by many leading smart-phone developers. These are capable of providing internet connectivity to the widest range of devices from embedded devices to low cost smart-phones.

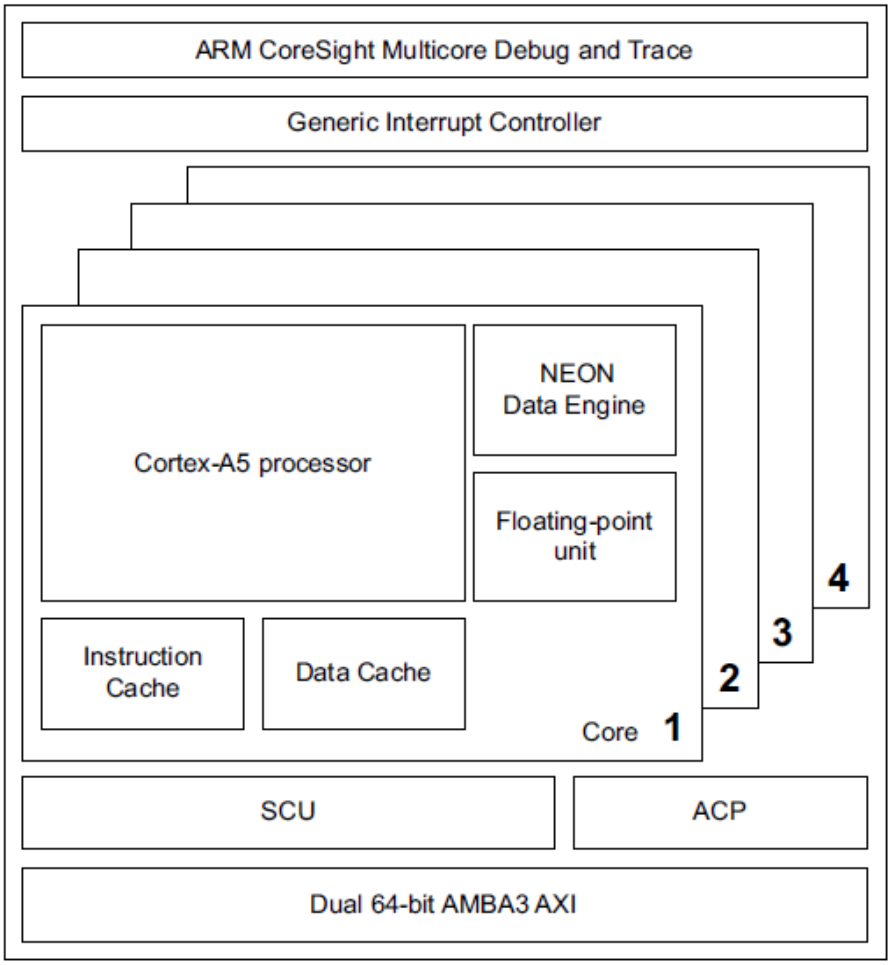

Figure 1: Cortex A5 processor

Salient features of ARM Cortex-A series processors:

i. This series of microprocessors are of 32bit.

ii. Backward compatible with previous ARM processors.

iii. Multiprocessing capability along with scalable and energy efficient performance.

iv. The Cortex-A series has powerful memory system with high performance memory management system.

v. Some of the processors of this family have:

   a. Floating point unit – Cortex-A7, A8, A9.

   b. NEON technology for SIMD and multimedia – Cortex-A7, A8, A9.

   c. Four way set associative L1 cache having size 16, 32, or 32 KB – Cortex A9.

   d. Available in to modes – Cortex A9:

      - High speed
      - Power optimized

1.4 Registers

ARM processors have 16 x 32bit general purpose registers (R0 – R15). Fifteen of the total sixteen registers R0 – R14 are for general purpose data storage and the R15 register is the program counter. The value of R15 is modified every time the instruction executes.

At any point of time coder has access to 16 general purpose registers and a CPSR register. This register has a restricted access in user mode. It is an important register, since it carries the status of various flag at the time of execution.

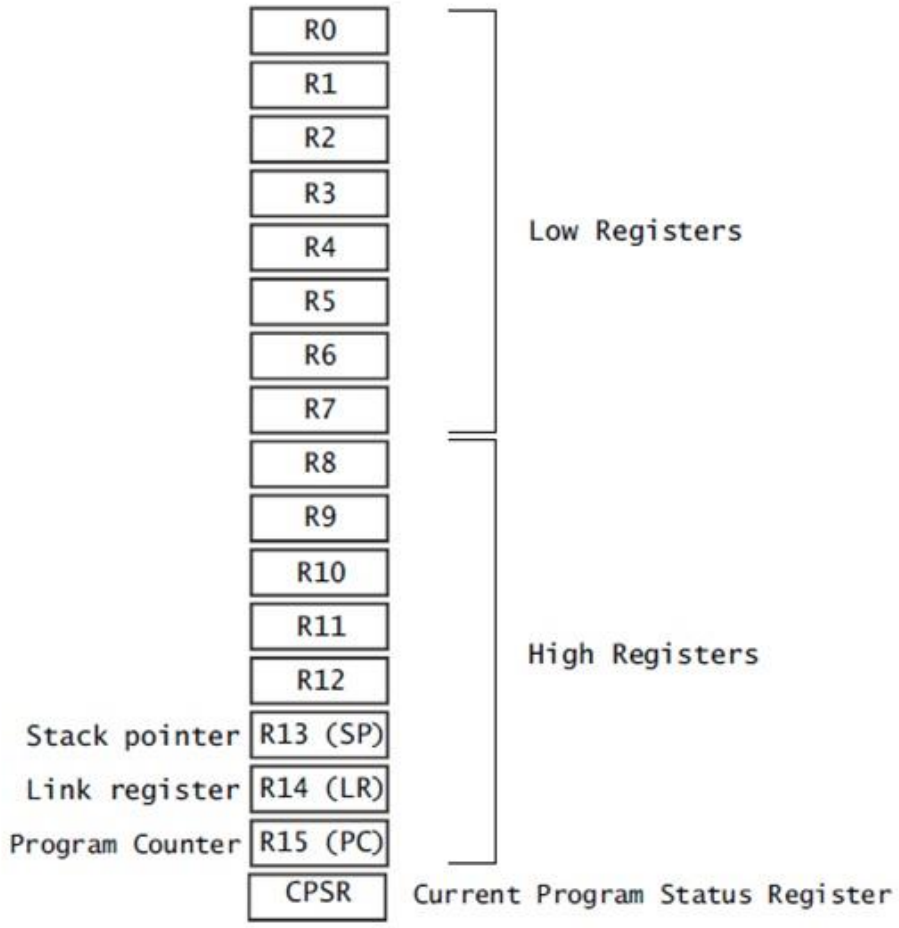

Figure 2: Cortex-A series registers visible for user code

The Current Program Status Register (CPSR) is shown for its bit assignment below:

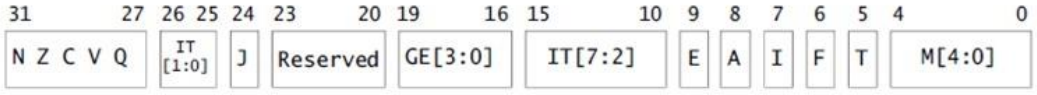

Figure 3: CPSR Register

The individual bits represent the following:

- *N* – Negative result from ALU.
- *Z* – Zero result from ALU.
- *C* – ALU operation Carry out.
- *V* – ALU operation oVerflowed.
- *Q* – cumulative saturation (also described as *sticky*)
- *J* – indicates whether the core is in Jazelle state.
- *GE*[3 :0] – used by some SIMD instructions.
- *IT* [7 :2] – If-Then conditional execution of Thumb-2 instruction groups.
- *E* bit controls load/store endianness.
- *A* bit disables asynchronous aborts.
- *I* bit disables IRQ.

- *F* bit disables FIQ.
- *T* bit – indicates whether the core is in Thumb state.
- *M*[4 :0] – specifies the processor mode

Co-processor 15

The Co-processor 15 i.e., CP15 registers are used to provide the control of many features of the core. CP15 contains 16 x 32 bit primary registers and named as c0 to c15.

1.5 Instruction set

Cortex-A series processors supports two types of assembly language instructions:

i. ARM instructions

ii. Thumb instructions

The instructions in both of these can have length 16 or 32 bits. Thus, the constant and immediate values greater than 32 bit, cannot be encoded. Apart from constant and immediate value instructions, the other type of instruction is conditional execution instructions for example less than, greater than etc. For conditional instructions the flags in the CPSR are set or reset accordingly.

2. BACKGROUND KNOWLEDGE

A detailed analysis of ARM architecture, multi-core environment, subsystems like performance monitoring unit, phase locked loop, interrupt controller etc for the development of the CFLib, has been done to gather the information that can be helpful in the process of designing and developing the common functions library.

2.1 Brief Description of Components of SoC

The various components of SoC are studied to get an idea about those components before developing functions for them. Although there are many components that can be integrated on the SoC, but only those topics are discussed below which falls in the scope of this project.

2.1.1 Subsystem

Subsystem is a smaller part which is self-contained system in a larger system. It includes a collection of inter-connected IP's and other parts that are capable of performing a certain task. The core interacts with various subsystems to perform a task. For example the I/O subsystem which is used to communicate with the many external input/output devices. A memory management subsystem, which provide features like shared virtual memory addresses, write-protected space in memory and memory mapping etc.

2.1.2 Cache

Cache is a small sized memory unit where the processor stores the data hidden from outer world. Cache is a high speed memory that resides near to the processor. It can be internal or external [1].

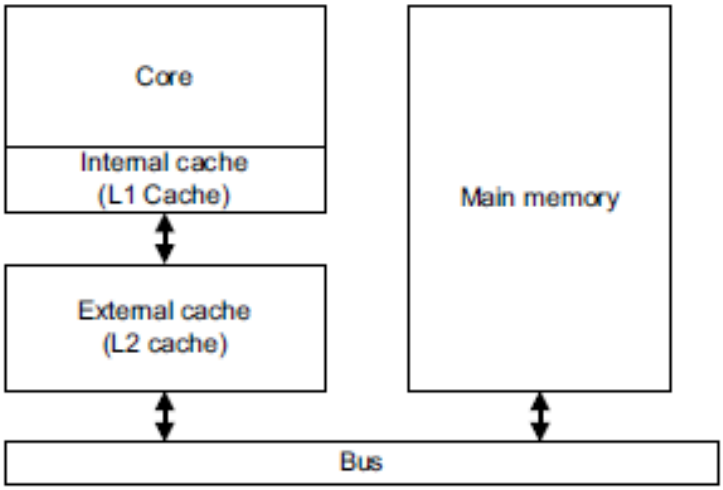

Figure 4: Basic Cache Architecture [1]

When core needs to read some data from memory, it first looks in the cache memory if it is able to find that address in the cache it uses the respective data (cache hit), else it has to look for that address in the main memory.

2.1.3 Interrupt Controller

Interrupt functions are written to handle the interrupts generated via GIC-400 interrupt controller. Generic Interrupt Controller is a set of hardware resources that are having some memory mapped registers to manage the interrupts. It is a daunting task to manage such a large number of interrupts [4].

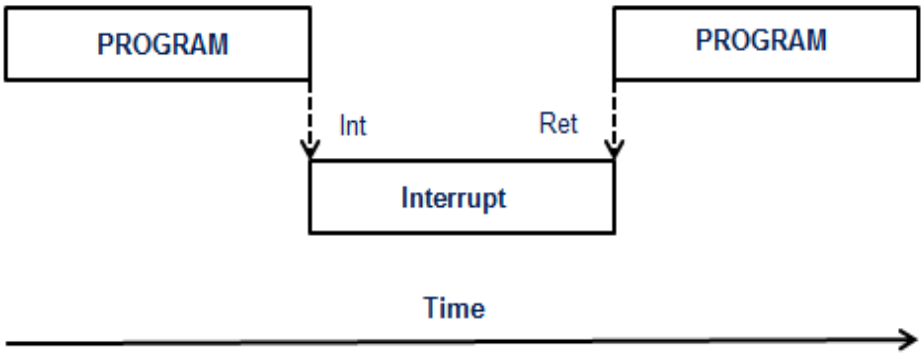

Figure 5: Flow of execution for handling Interrupt

The architecture of GIC is divided into two blocks, Distributor block and CPU interfaces.

Distributor: Distributor block is near to the external devices. It takes the interrupt from them and prioritizes it and distributes it among one of the CPU interfaces.

CPU interface: In multi-core environment each core has a CPU interface associated with it. The CPU interface handles pre-emption and priority masking for the CPU it is connected to [5].

2.1.4 Performance Monitoring Unit

In today's world with technology growing at a rapid rate and new technologies developing and taking place of previous technologies, these new techniques focuses on easing the humans life by designing newer and much more efficient products. There are some features like cost effectiveness, ease of use & higher security which are emphasized while creating a new project. Performance is also one of the main focuses in this respect. The smart mobile phones, smart TV's etc are now a days a emerging technology. Their performance is a major area where the buyers point it out. So, ARM provided an extra co-processor in there cores to measure the performance of various events that are carried out by the core to perform a specific task. When there is a need to collect information about the processor, memory system and other events, PMU serves that purpose. The processors have some logistics for this purpose. PMU monitors events and gather statics about them [2]. The events that can be monitored are:-

i. architectural and micro-architectural events

ii. implementation specific events.

PMU has three main components:

i. Event Interface: It provides events from all other interfaces to the PMU.
ii. CP15 and APB interface: The CP15 system control co-processor or external APB interface is used to program PMU registers
iii. Counters: PMU has a cycle counter and 32 counters that incremented when a they are enabled for a specific event.

Below is the block diagram of PMU:

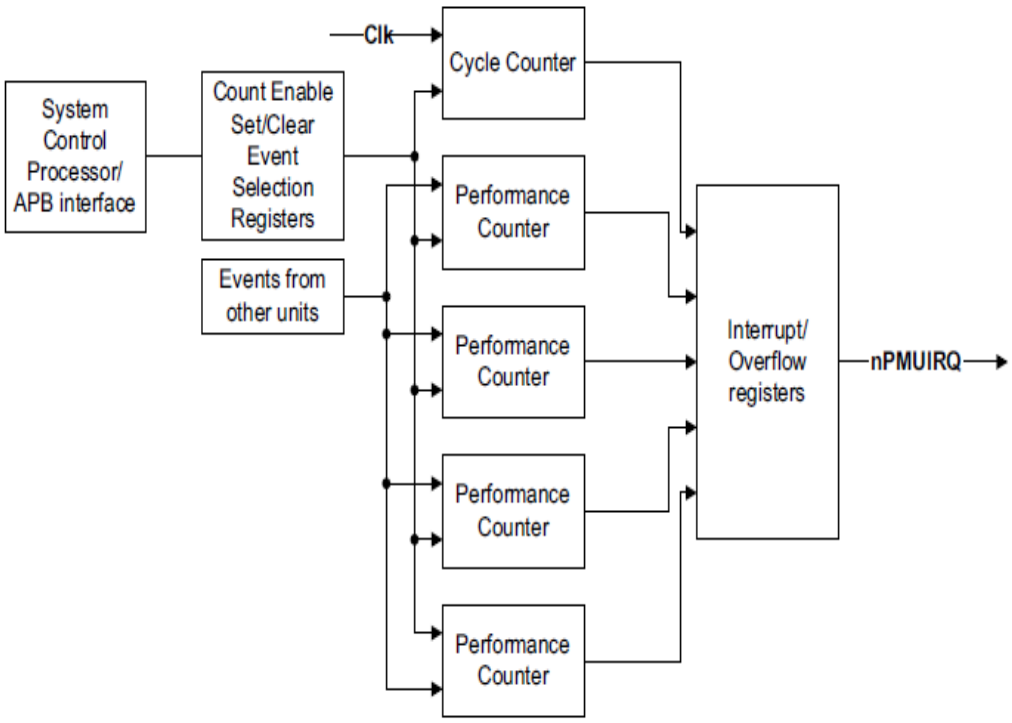

Figure 6: PMU Block Diagram [3]

2.1.5 Phase Locked Loop

PLL stands for Phase Locked Loop. It is a feedback system integrated along with a phase comparator and a voltage control oscillator (VCO) connected in a way such that the oscillator maintains the constant phase angle. The phase angle maintained by the oscillator is relative to the reference frequency signal applied to it. It is called as feedback system, since phase detector matches the phase of signal produced by the oscillator with the phase of the input signal and the output signal came back to the input signal. PLL can be used to generate constant high frequency signal from a low frequency signal.

PLL can generate the output frequency which is multiple of the input frequency. This property of PLL is used in the synchronization of the computer clock.

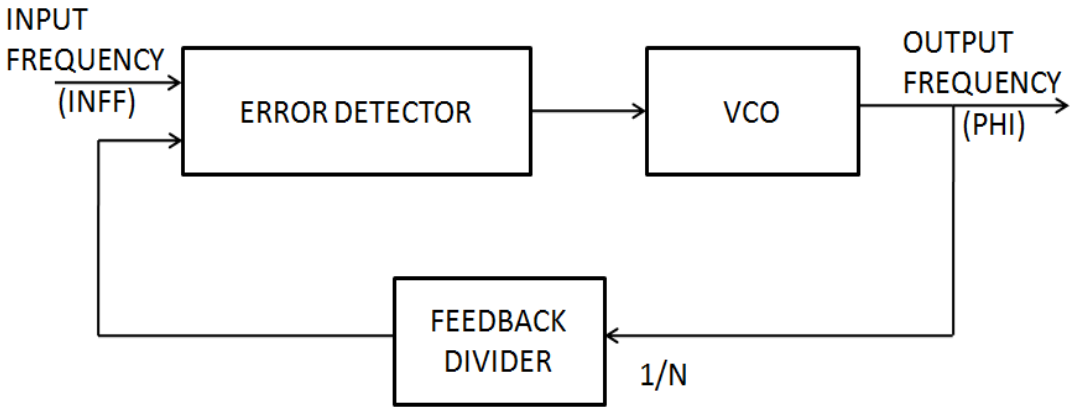

Figure 7: Basic Phase Locked Loop Model

The above block diagram shows a basic model of a PLL. It includes an error detector which takes the input frequency and compares it with the desired output frequency and gives output to the voltage control oscillator (VCO). Output given by the error detector is directly proportional to the phase difference between the two input signals given to it. This signal when provided to the VCO, it produces the output phase. The output signal is again fed back to the error detector via feedback divider, hence creating a negative feedback loop. Thus the desired output phase (PHI) is locked by removing the errors through error detector.

PLL is used to provide the desired frequencies to internal parts of the sub-system. For generating the output frequency, external clock is used as the reference input known as INFF.

Below is the formula for calculating the output frequency (phi):

$$phi = \frac{INFF * LDF}{2 * IDF * ODF}$$

Here, phi is the desired output frequency from the PLL

IDF, LDF, ODF are three constants. Where value ranges are:

IDF (Input Division Factor)-> [1, 7]

LDF (Loop Division Factor) = 2 * NDIV (input to set LDF)

NDIV -> [8, 246]

ODF (Output Division Factor) -> [1, 63]

As, until now only a few output frequencies are desired, so the values of the three constants IDF, LDF and ODF are manually calculated for frequencies 25 MHz, 100 MHz, 800 MHz and 1200 MHz . But, now as more complex and newer cores are developed by ARM that comprises of a newer 64 bit RISC processor, newer input frequencies are desired.

Also, while calculating the value of the phi, the specified range of the voltage controlled oscillator must be maintained. Below is the formula for calculating the value of VCO:

$$VCO = 2 * phi * ODF$$

For each part designer manually calculates the values of the constants that maps with the frequency desired to operate the sub-system. So, there was a need to create such an algorithm that automatically calculates the values of the constants and fed those values to give the desired phi. The challenge in this scenario is that there is only one equation to calculate phi whereas there are three constants for which the value is to be calculated. So, there is a lot of research and effort devoted to develop such an algorithm that can provide the values of IDF, ODF and NDIV for a desired phi by giving as reference INFF.

2.2 Tools and Technologies Used

2.2.1 ARM Assembler

ARM assembler “armasm” is used to interpret and convert assembly language code written in ARM assembly instructions or Thumb assembly instructions in to the machine interpretable code. ARM assembler has an integrated “armclang” tool which can assemble code written in GNU syntax that use inline C or C++ code into assembly language code. ARM assembler now supports ARM-64, ARM-32 and Thumb-32 instructions.

To invoke armasm, use following command:

armasm [options] [input_file_list]

here,

options: assembler command-line options

input_file_list: one or more input files.

2.2.2 ARM Compiler

The ARM Compiler “armcc”, is a C and C++ compiler. It is used to get the machine code for the ARM architecture based compilers by compiling the standard C and standard C++ code. There are many features in armcc that are designed to take advantage of the ARM processors and architecture[3].

The compiler can take three language source codes as input:

i. ISO C90

ii. ISO C99

iii. ISO C++

Below is the command to initiate the use of armcc:

armcc [options] [input_file_list]

here,

options: compiler command-line options

input_file_list: one or more input files.

Example:

armcc –c [options] input_file_1 . . input_file_n

2.2.3 ARM Linker

ARM linker “armlink” is used to link the pre-compiled files. These files can be object files, ELF files or libraries. It is an important tool as it takes many files in different formats and generates a single ELF file, library file or an object file. ARM linker can select suitable C or C++ libraries automatically [3]. It can minimize the ROM size usage by performing read/write compression. It is also available in 64 bit version.

The command used to invoke the armlink:

armlink [option] [input_file_list]

here,

options: linker command-line options

input_file_list: single or multiple input files that are space-separated

Example:

armlink –library=libcfl.a –libpath=./users/singhj2/cflibarm_ok.elf

2.2.4 ARM Librarian

The ARM Librarian i.e., “armar” is tool to create standard format “ar” libraries. The tool helps to create and maintain library by collecting ELF and object files. This file can be passed to the linker, instead of several ELF object files [4].

ARM librarian provides many features:

i. creating a library

ii. adding files to a library

iii. replacing a file from the library

iv. replacing a number of files with some other files

v. helps in managing the placement of files

vi. provides information about contents of the library

3. PROBLEM STATEMENT

Function Library is built by making a bunch of routines from which desired function is invoked where and when needed. It includes pre-compiled functions those would be needed by the user. The development process of library should not only optimize the time-efficiency but also it should be enough straight forward that its users could understand its usage easily.

3.1 Problem Definition

The task of a verification engineer is to find and report the bugs in the system design. He has to test the whole design by writing the verification test codes. The robustness of the test code defines the efficiency of that code which directly impacts the productivity of end product. So, the code should be reliable and it should save the engineer's time for writing the verification test code. The library includes all the functions which can run on different cores and could provide all the essential features to ease the work of verification test developers.

3.2 Gap Analysis and Justification

There is no existing centralized function library. Each developer has to manually write the code which creates lots of redundancy and impose additional effort on developer to redo the work already done. In other case, the developer may search for the code already written so as to save additional effort but it is a time consuming task to find the particular code from the massive database and also code (if found) is very difficult to understand.

The new approach is to facilitate the developer with the ease of use by providing atomic and discrete functions. This library is placed at a centralized memory location. The test developer only has to link that library file and he can use the functions in that. The code behind the function will be hidden from the test developer (Data Abstraction). Comments are written along with the respective function in the header files so that developer can understand the usage of that respective function.

The work is carried out not only to suite various available functions into a single unit, but the main tasks were to write the new functions and to automate the previously hard coded functions. The important work which was to be carried out is listed below:

1. Initiate the usage of performance monitoring unit (PMU).
2. Code to control the execution of different cores in multi-core environment.
3. Algorithm to program PLL for given output frequency (phi).
4. To remove the redundancy of the existing code and create a static library that can be deployed in a centralized location.

4. PRESENT WORK

The below sections describes the work done in development of the library of functions. Firstly, structure of CLF is explained thoroughly and then in implementation section, header files of the functions are given with description of those functions and their arguments.

4.1 Structure

The project CFLib has a well-defined structure. The base directory "cfl_release_#" (# stands for version number) contains the directory "cfl", "cfl_release_note.txt" and "libcfl.a". "libcfl.a" is the pre-compiled static library that can be linked at the run time, "cfl_release_note.txt" is the text file that provides brief description about the release version.

The directory "cflib" has a structure shown below:

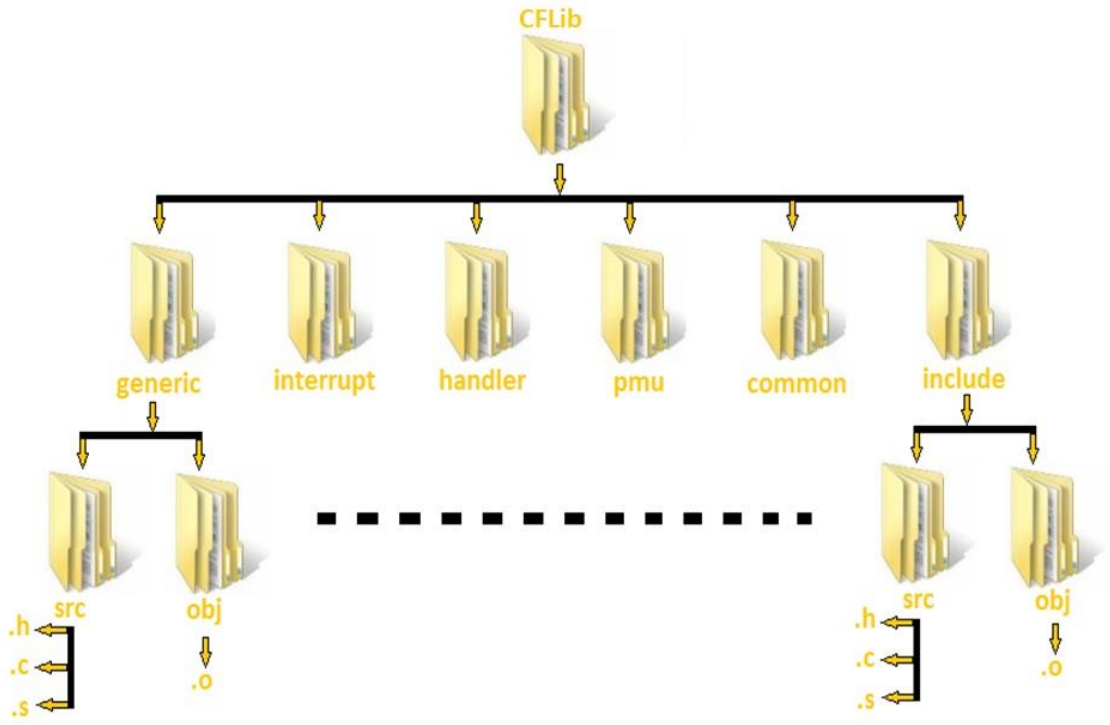

Figure 8: Structure of CFLib

CFLib is a static library which is delivered as an archive and the header file. The prototype of functions written is given below. These functions are deployed and currently used by the test writers. These are compiled using ARMCC which internally calls ARMASM. After creating the object file of each source code file. The objects files then are used to create an archive file using ARMAR.

CFLib Release: Every release of CFLib has a particular structure. The root directory is named as "cfl_release_#". It includes the following:-

a. a "libcfl.a" file and
b. Header files, and
c. Release documentation

ARM Librarian is used to create the static library "libcfl.a". It is a static library file that is to be linked while running the test. This is a precompiled library file, so it does not create any overhead while compilation.

The header files contain the prototype of the functions in those files. The test developer who needs to access the function in his test needs to include the header file of that respective function. Also, there is little bit of description about the function given in the header file for ease to understand the function and properly use that function.

The release documentation tells about the changes made, new modifications and updations in the previous release.

There is another way in to design the structure of the CFLib. According to this structure definition there are two types in which a test case can be written i.e., platform dependent and core dependent. Platform definition part is not covered here as it is to be developed by the front end design team. They will create the functions specific to the projects and those will be integrated in the CFLib. Until now, there was no function found specific to a single core. Hence, this structure is designed with keeping needs for future in mind. If in future, there are some functions developed that are specific to the core, since 64bit ARM processors have different architecture, the core specific functions will have a

different segregation as show in the below diagram. The below flow chart gives a brief idea how CFLib will be integrated in the verification tests:

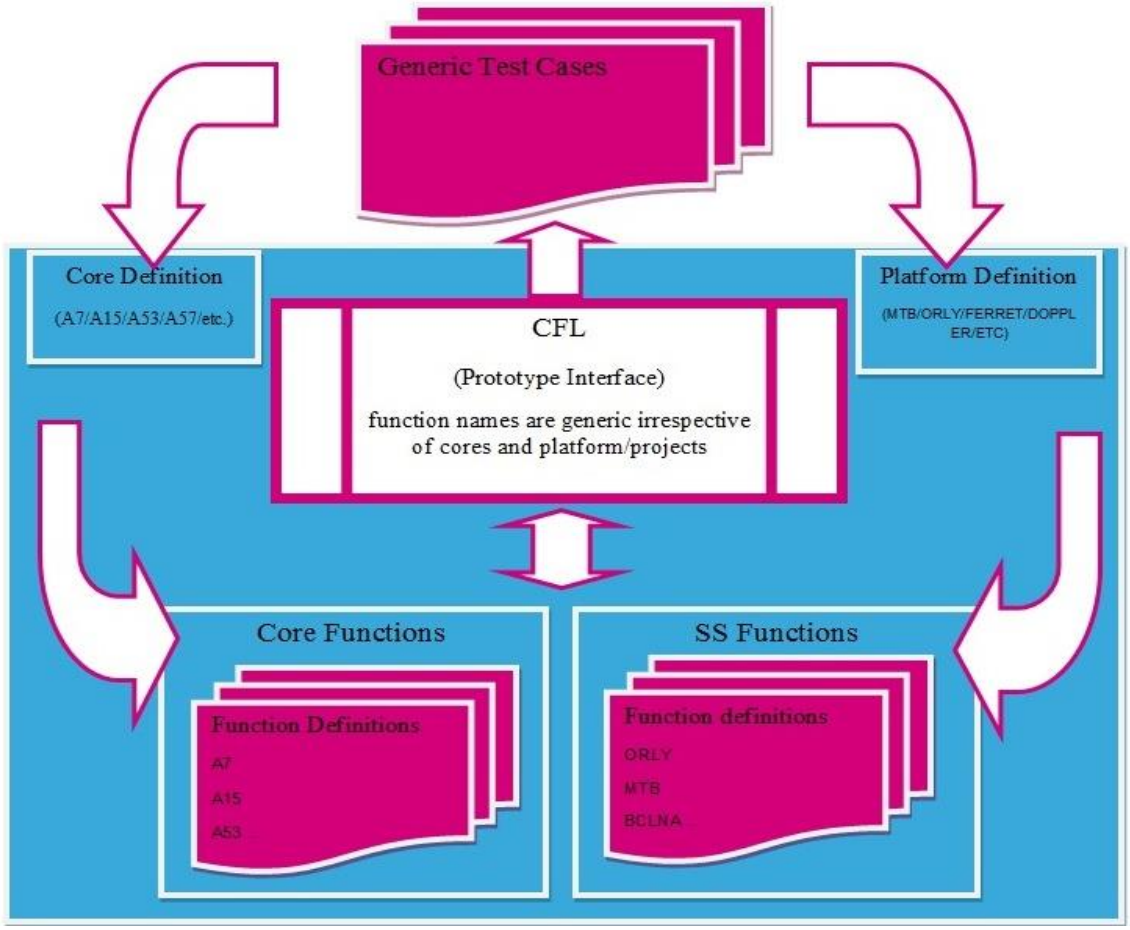

Figure 9: Integration of CFLib in Generic Test Case

4.2 Implementation

This section describes the functions developed according to the efforts. PMU, CPU execution and PLL functions toke much of the effort, so they are described and explained in different sub-sections. The other functions section includes all other functions related to Cache, sub-system, interrupt controller.

4.2.1 Performance Monitoring Unit

Performance Monitoring is a step-wise process. Firstly, the flow of execution is designed to monitor any event whether it is an architectural/micro-architectural event or it is an

implementation specific event. Below shown flow chart is designed, that is used when any event is to be monitored.

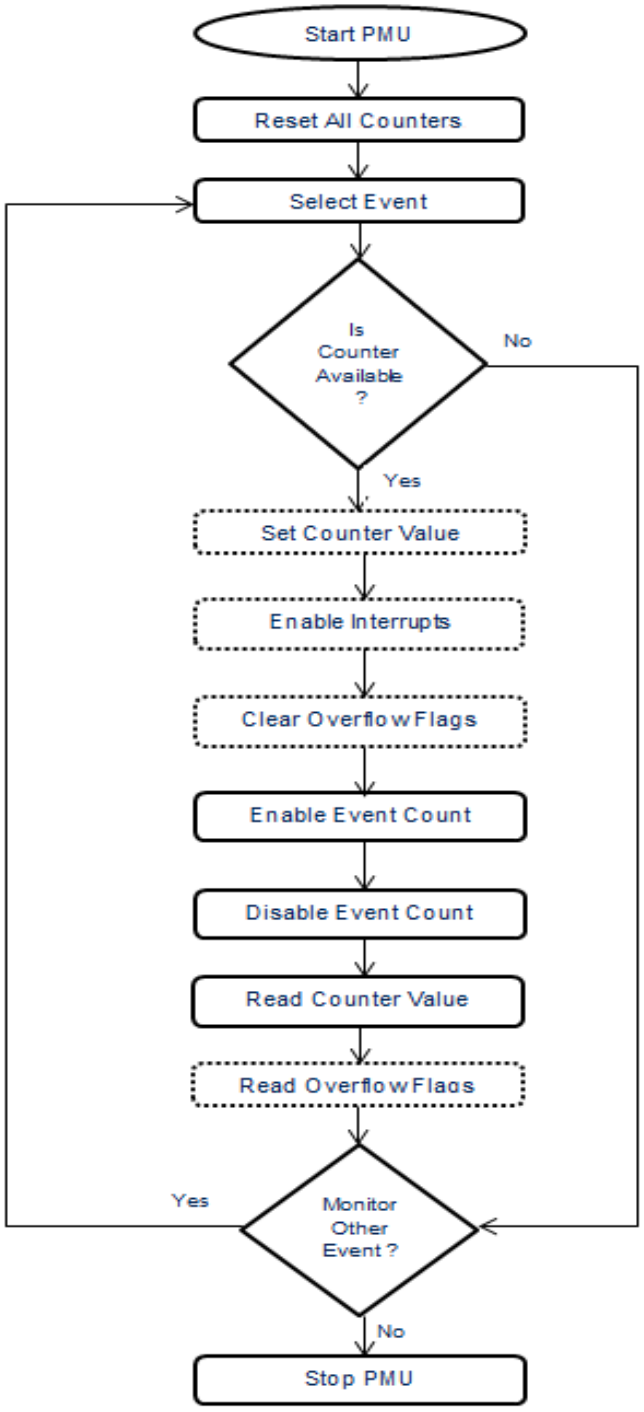

Figure 10: Flow chart for Performance monitoring

Below are various functions to program PMU for monitoring an event. These are written in sequential order in accordance with the flow chart.

i. cfl_pmu_resetAllCounters():

- **Scope:** This function resets all available counters and also the clock counter PMCCNTR. To reset the counters PMCR (Performance Monitors Control Register) is needed to be accessed.

  Setting P (Event Counter Reset) and C (Cycle Counter Reset) bits to 1 will serve our purpose.

- **Arguments:** Void.

- **Return:** Void

ii. cfl_pmu_start():

- To start PMU enable all the available counters and cycle counter.

  Set E (Enable) bit of PMCR.

- **Arguments:** Void.

- **Return:** Void

iii. cfl_pmu_stop():

- **Scope:** To stop the PMU i.e., Disable all counters.

  To disable all counters including PMCCNTR, access PMCR and Reset the Enable bit (E) to 0.

- **Arguments:** Void.

- **Return:**

  - Return 1 if successfully stopped.

  - Return 0 with error message if PMU is not started yet.

- **Dependency:** PMU must have been started before stopping.

iv. cfl_pmu_setClockDriver(arg1):

- **Scope:** This function will set the clock driver to count clock cycles according to the Boolean value passed as argument. In PMCR, D (Clock driver) bit is set to 1 to perform this function.
- **Arguments:** Boolean value
  - If value = 1, clock driver will count every 64^{th} clock cycle.
  - If value = 0, clock driver will count every clock cycle.
- **Return:**
  - Return 1 if value of argument is valid i.e., 0 or 1.
  - Return 0 with error message if value of argument is invalid.

v. cfl_pmu_readClockDriver():

- **Scope:** This function will read the D bit in PMCR and will give Boolean output.
- **Arguments:** Void.
- **Return:**
  - If output is 1 : Clock driver counts every 64^{th} clock cycle
  - Else : Clock driver counts every clock cycle

vi. cfl_pmu_selectEvent(arg1):

- **Scope:** The event to be monitored must be specified in PMXEVTYPER (Performance Monitors Event Type Select Register).This Function will automatically select a counter (if available) and sets the event passed as argument to that counter.
- **Arguments:** event mnemonic (as given in hashdefs.h) or number.
- **Return:**
  - Return 1 if event is successfully associated to one of the available counter.

- Return 0 with error message if any exception occurs.

- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - At least one of the counters must be available.

vii. cfl_pmu_deSelectEvent(arg1):

- **Scope:** This function will free the counter from the event passed as argument.
- **Arguments:** event mnemonic (as given in hashdefs.h) or number.
- **Return:**
  - Return 1 if counter is successfully freed.
  - Return 0 with error message if any exception occurs.
- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - The event in argument list must be associated to one of the available counters.

viii. cfl_pmu_setCounterValue(arg1, arg2):

- **Scope:** The initial value of the counter can be set to a desired value by accessing PMXEVCNTR (Performance Monitors Event Count Register).
- **Arguments:**
  - **arg1:** event mnemonic or number.
  - **arg2:** value to be set on to the counter.
- **Return:**
  - Return 1 if value is set on to the counter associated with the given event.

- Return 0 with error message if any exception occurs.

- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - The event in argument list must be associated to one of the available counters.

ix. cfl_pmu_enableEventCount(arg1):

- **Scope:** This function will enable the counter that is associated with the event passed as argument. PMCNTENSET (Performance monitors Count enable Set Register) register is used to enable the PMCCNTR and the desired counter.
- **Arguments:** event mnemonic (as given in hashdefs.h) or number.
- **Return:**
  - Return 1 if counter is enabled.
  - Return 0 with error message if any exception occurs.
- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - The event in argument list must be associated to one of the available counters

x. cfl_pmu_disableEventCount(arg1):

- **Scope:** Counters should be disabled when not in use. To disable counters PMCNTENCLR (Performance Monitors Count Enable Clear Register) is to be programmed. This function will disable the counter that is set to count the event passed in argument list.
- **Arguments:** event mnemonic (as given in hashdefs.h) or number.
- **Return:**

- Return 1 if counter is Disabled.
- Return 0 with error message if any exception occurs.

- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - The event in argument list must be associated to one of the available counters

xi. cfl_pmu_disableAllCounters():

- **Scope:** When accessed this function will disable all the counters.
- **Arguments:** Void.
- **Return:**
  - Return 1 if all the counters are disabled.
  - Return 0 if exception occurs.
- **Dependency:** PMU must be started.

xii. cfl_pmu_readCounter(arg1):

- **Scope:** This function reads the value of the counter set to count for the event passed in argument list.
- **Arguments:** event mnemonic (as given in hashdefs.h) or number.
- **Return:**
  - Return 1 if counter is read.
  - Return 0 with error message if any exception occurs.
- **Limitaions:**
  - PMU must be started.
  - Event number should be valid.

- The event in argument list must be associated to one of the available counters.

xiii. cfl_pmu_interruptEnable(arg1):

- **Scope:** The function cfl_pmu_interruptEnable(arg1) will find the counter set to count event passed in argument list and set the bit to 1 in PMINTENSET register for that counter. The PMINTENSET (Performance Monitors Interrupt Enable Set Register) enables the generation of interrupt requests on overflows from:
  - the Cycle Count Register, PMCCNTR
  - Each implemented event counter.
- **Arguments:** event mnemonic (as given in hashdefs.h) or number.
- **Return:**
  - Return 1 if value for counter is set to 1 in PMINTENSET register.
  - Return 0 with error message if any exception occurs.
- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - The event in argument list must be associated to one of the available counters.

xiv. cfl_pmu_interruptDisable(arg1):

- **Scope:** The PMINTENCLR register disables the generation of interrupt requests on overflows from:
  - the Cycle Count Register, PMCCNTR
  - each implemented event counter.
- **Arguments:** event mnemonic (as given in hashdefs.h) or number.
- **Return:**

- Return 1 if value for counter is set to 1 in PMINTENCLR register.
- Return 0 with error message if any exception occurs.

- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - The event in argument list must be associated to one of the available counters.

xv. cfl_pmu_clearOverflowFlag(arg1):

- **Scope:** This function clears the overflow status bit in PMOVSR for the counter set to count event passed as argument.
- **Arguments:** event mnemonic (as given in hashdefs.h) or number.
- **Return:**
  - Return 1 if value for counter is set to 1 in PMOVSR register.
  - Return 0 with error message if any exception occurs.
- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - The event in argument list must be associated to one of the available counter.

xvi. cfl_pmu_readOverflowFlag(arg1):

- **Scope:** The PMOVSR holds the state of the overflow bits for:
  - the Cycle Count Register, PMCCNTR
  - each of the implemented event counters.

  Setting a bit in PMOVSR to 1 will clear the overflow status for that counter.
- **Arguments:** event mnemonic (as given in hashdefs.h) or number.

- **Return:**
  - Return 1 if value for counter is set to 1 in PMOVSR register.
  - Return 0 with error message if any exception occurs.
- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - The event in argument list must be associated to one of the available counters.

xvii. cfl_pmu_softwareIncrement():

- **Scope:** The PMSWINC register increments a counter that is configured to count the Software increment event, event 0x00.
- **Arguments:** Void.
- **Return:**
  - Return 1 if counter is found and incremented.
  - Return 0 with error message if any exception occurs.
- **Dependency:**
  - PMU must be started.
  - Event number should be valid.
  - The event in argument list must be associated to one of the available counters.

xviii. cfl_pmu_userModeAccess();

- **Scope:** The performance monitoring register PMUSERENR enables or disables User mode access to the Performance Monitors.

  PMUSERENR.EN is user mode access bit. Possible values:
  - 0 User mode access to the Performance Monitors disabled.
  - 1 User mode access to the Performance Monitors enabled.

- **Arguments:** Void.
- **Return:**
  - Return 1 if success.
  - Return 0 otherwise.
- **Dependency:**
  - PMU must be started.

4.2.2 CPU Execution Functions

In a multi-core environment, cores can run in serial order or in parallel. By default, the cores run in parallel while testing, but there are certain verification tests which demand the core to run in serial order. The below given functions are used to run cores in serial order. Prior to development of these functions, the verification engineer has to explicitly update and then compile the boot code which was a troublesome task.

i. intcfl_gen_numberOfCores(void):

- **Scope:** this function returns number of cores present in that cluster.
- **Arguments:** void.
- **Return:** number of cores in the cluster

ii. void cfl_gen_coreSerialExecution(void *func)

- **Scope:** the below function takes void function pointer as perimeter and executes the code serially which is written inside that function.
- **Arguments:** void pointer to the function
- **Return:** void

iii. void cfl_gen_waitForAllCores(void);

- **Scope:** this function will trap the cores until each core enters it and then release all the cores together.
- **Arguments:** void.
- **Return:** void

iv. void cfl_gen_sleepSecondaryCpus(intprimary_cpuID):

- **Scope:** This function will put all the secondary CPUs in WFE

- **Arguments:** ID of primary CPU
- **Return:** void

v. void cfl_gen_wakeupSecondaryCpus(intprimary_CpuID):

- **Scope:** This function will Wake up all the secondary CPUs from WFE by sending SEV
- **Arguments:** ID of primary CPU
- **Return:** void

4.2.3 Phase Locked Loop

Phase locked Loop can be programmed in two modes:

- Integral mode
- Floating Point Mode

In integral mode, due to the discrete values of constants every desired frequency cannot be achieved. For the purpose of achieving a desired frequency and decimal value frequency the PLL is programmed in floating point mode.

Below given functions are written to program PLL to a desired frequency.

i. int cfl_pll_programming(int frequency);

- **Scope:** Programs the PLL to a desired frequency by calculating the values of NDIV, ODF and IDF and writing on to the SSCG_PLL_FREQ register and then switch CPU to the internal pll. Also maintains the valid output frequency and VCO range for PLL_4600.
- **Arguments:** desired output frequency (phi)
- **Return:** Status: Return 1 -> if pll is programmed to desired frequency

  Return 0 -> on failure

ii. int cfl_pll_switchToPll(void);

- **Scope**: If CPU is working on external clock, this function switch it to the already programmed internal pll.

- **Arguments**: None
- **Return**: Return 1 -> if switched to internal pll

  Return 0 -> if already running on internal pll.

iii. intcfl_pll_OdfProgramming(intODF_val);

- **Scope**: If CPU is working on internal pll this function changes its ODF value to a desired value. The ODF value updates the output frequency (phi) also. This functions allows the value of ODF to be changed only if the phi remains in the valid range.
- **Arguments**: value of ODF
- **Return**: Return 1 -> if switched to internal pll

  Return 0 -> if already running on internal pll.

4.2.4 Other Functions

This section includes memory read/write functions, functions to print on a specific tube address, cache functions to access and then clean and invalidate it, functions to provide interrupts to cores using generic interrupt controller and semaphore functions to access shared variables in locked environment.

Memory Functions: These functions are used to read/write data on a particular memory address. The function must be called in accordance with the need to read or write a word/half-word/byte.

i. cfl_gen_write (argument_1, argument_2);

- **Scope:** The function "cfl_gen_write(argument_1, argument_2)" is used to write a word (32 bits) given in argument_2 on a specified memory location given in argument_1.
- **Arguments: (**unsigned integer) address, (unsigned integer) data (32 bits)
- **Return:** void

ii. cfl_gen_hwrite (argument_1, argument_2);

- **Scope:** This function is used to write a half word (16 bits) given in argument_2 on a specified memory location given in argument_1.
- **Arguments: (**unsigned integer) address, (unsigned short integer) data (16 bits)
- **Return:** void

iii. cfl_gen_bwrite (argument_1, argument_2);

- **Scope:** This function if called, will write a byte (8 bits) on a specified memory address.
- **Arguments: (**unsigned integer) address, (char) data (8 bits)
- **Return:** void

iv. cfl_gen_read (argument);

- **Scope:** "cfl_gen_read(argument)" function is used to get the 32 bit word stored at the memory location passed as argument.
- **Arguments: (**unsigned integer) address.
- **Return:** void

v. cfl_gen_read (argument);

- **Scope:** "cfl_gen_read(argument)" function returns an unsigned half word from the location given in argument list.
- **Arguments: (**unsigned integer) address.
- **Return:** unsigned short integer.

vi. cfl_gen_read (argument);

- **Scope:** This function returns a byte stored at address passed as argument.
- **Arguments: (**unsigned integer) address.
- **Return:** char as integer.

Print Functions: Standard print functions print the formatted output on to the screen. They do not work for printing a message on particular tube address. So there was a need of some explicit print function which can print a formatted output on to the tube. The below given print function is developed to print the formatted output on tube.

i. cfl_gen_printMsg();

- **Scope:** This function can be used to print a string (using %s or %S), signed and unsigned integers (using %d or %D and %u or %U resp.), hexadecimal

numbers (using %x or %X) and long signed/unsigned integers and hexadecimal numbers (using %l or %L before respective type).

- **Arguments:**Variable arguments.
- **Return:** Print formatted output on the tube.

ii. void cfl_gen_exitTestCode(interror_code);

- **Scope:** this function prints the test exit status (PASS/FAIL) on the tube and then terminates the test.
- **Arguments:** error code as integer. Zero if no error.
- **Return:** void

Cluster Functions

If a system has more than four processors, then they are divided into clusters. ARM provides some registers to track the number of clusters and cores in the system. The below two functions are work upon those cluster register to get the desired information.

i. cfl_gen_getClusterID(void):

- **Scope:** This function returns the cluster identity.
- **Arguments:** void.
- **Return: r**eturn cluster ID

ii. cfl_gen_getCpuID(void):

- **Scope:** This function returns the CPU identity.
- **Arguments:** Void.
- **Return:** return CPU ID

Cache Functions

While testing the system cache, it is needed to cleaned and invalidated. The functions given in this section are written to access and then clean and invalidate the cache.

i. void cfl_cac_init(void)

- **Scope:** Enable MMU, Data and instruction caches. Also set Z(branch prediction) and SW(SWP and SWPB perform normally) bits of SCTLR
- **Arguments:** void

- **Return:** void

ii. **void cfl_cac_enableCache(void);**

- **Scope:** Enable data and instruction cache
- **Arguments:** void
- **Return:** void

iii. **void cfl_cac_disableCache(void);**

- **Scope:** Disable data and instruction cache
- **Arguments:** void
- **Return:** void

iv. **void cfl_cac_cleanInvalidateDCacheLine(int lineNum);**

- **Scope:**Clean and invalidate Data cache line by set/way
- **Arguments:** void
- **Return:** void

v. **void cfl_cac_cleanInvalidateDCacheLineMVA(int lineNum);**

- **Scope:** Clean and invalidate Data cache line by MVA
- **Arguments:** void
- **Return:** Nothing

vi. **void cfl_cac_invalidateICacheLine(int lineNum);**

- **Scope:** Invalidate Instruction cache line by set/way
- **Arguments:** line number to be invalidated
- **Return:** void

vii. **void cfl_cac_invalidateDCacheLineMVA(int lineNum);**

- **Scope:** Invalidate Instruction cache line by MVA
- **Arguments:** line number to be invalidated
- **Return:** void

viii. **void cfl_cac_cleanInvalidateCaches(void);**

- **Scope:** Clean and invalidate entire Data cache, invalidate entire Instruction cache
- **Arguments:** void
- **Return:** void

Interrupt Functions

Interrupt Controller has two components:

- Distributor
- CPU interface

Distributer gets the interrupts from various input devices. The distributor registers are programmed to eliminate the conflicts, set priorities and to assign there interrupts to one of the available CPU interfaces. Upon getting any interrupt from the distributor, CPU interface passes it to the connected CPU. If there are more than one interrupts, than CPU they are queued in the CPU interface buffer. The given functions are developed to program both the components to provide fast interrupts (FIQ) and normal interrupts (IRQ) to one of the connected CPUs.

i. void cfl_int_initDistributor(int intNum, int intType);

- **Scope:** Initialize the Distributer. Set :
  i. Group register bit,
  ii. Enable interrupt,
  iii. Set priority,
  iv. Target CPU
- **Arguments:**
  i. Interrupt number
  ii. Interrupt type : FIQ or IRQ
- **Return:** void

ii. void cfl_int_setCoreInt(int intType, int status);

- **Scope:** Enable/Disable the interrupt
- **Arguments:**
  i. Interrupt number
  ii. Interrupt type : FIQ or IRQ
- **Return:** void

iii. void cfl_int_initCPUInterface(int intType);

- **Scope:** set the CPU interface to pass the interrupts
- **Arguments:** interrupt type : FIQ or IRQ
- **Return:** void

iv. int cfl_int_checkPendingInt(int intNum);

- **Scope:** Check whether the interrupt is pending or not
- **Arguments:** interrupt number
- **Return:** pending status.
  - Return 1 -> interrupt is pending
  - Return 0 -> interrupt is not pending

Semaphore

In multi-core environment, updates to shared variables have to be protected through semaphores to ensure mutual exclusion [11]. The semaphore variable (must be global variable) placed in a shared location once initialized can be updated by any core provided it must have lock upon that semaphore. Here, binary semaphores are used to get a lock upon the shared variable.

i. void cfl_gen_initSemaphore(int *semaphore)

- **Scope:** Semaphore must be initialized only once before use.
- **Arguments:** pointer to an integer (semaphore)
- **Return:** void

ii. void cfl_gen_getSemaphore(int *semaphore)

- **Scope:** Get the lock on semaphore.
- **Arguments:** pointer to an integer (semaphore)
- **Return:** void

iii. void cfl_gen_releaseSemaphore(int *semaphore)

- **Scope:** Once acquired the semaphore must be released
- **Arguments:** pointer to an integer (semaphore)
- **Return:** void

5. RESULTS

The library is deployed on a central shared location. The authorized verification engineers can access it by linking it in their development and verification architecture. All the functions in the library were tested for their proper working on different ARM cores in scope of this project before deployment. Below given is the path of library from where it can be accessed:

Path: /data/a9cannfe2/users/singhj2/Coding/cfl_release_2/

The project CFLib has two releases. The current version in use is release_2. It contains newly added functions for interrupt controller and phased locked loop which were not there in the initial released version.

cfl_release_2 directory contains:

1. project cflib (cflib directory)
2. cfl_release_note (text file)
3. cflib static library (libcfl.a)

CFLib is an ever expanding project. If there is a need to add some functions in it, write and then compile them using “armcc” and use the following “armar” command to update the library.

Commands:

To update library: armar –ru <libaray_name.a> <.o file>

The table given below shows the comparison between the existing work and current work done.

Functions	Existing Work	Current Work
PMU	Not Programmed	Programmed and working
PLL (Integer mode)	Hard coded for frequencies 20MHz, 100MHz, 200MHz, 800MHz & 1000MHz	Programmed for every integral frequency
PLL (Fractional mode)	Not programmed	Programmed and working
CPU execution	Manual work to update the Boot code to run test code serially	Automated the process to execute the code serially
Semaphores, Memory and Print Functions	already available but not efficient for verification purpose	Modified, Integrated to library and verified for all cores
Cache	already available but scattered	Integrated to library and verified for all cores
Interrupt controller	programmed but not according to the current GIC version	made compliant with current GIC version

Table 1: Comparison between existing and current approach

Phase Locked Loop is programmed to provide the values of NDIV, IDF and ODF for any valid output frequency range. The below Table shows the sample output of PLL algorithm working in Integral mode:

`	Output Value given by algorithm		
Phi	**NDIV [8-246]**	**IDF [1-7]**	**ODF [1-63]**
20	enter valid input		
25	175	7	50
50	175	7	25
100	168	7	12
200	168	7	6
250	175	7	5
400	168	7	3
500	210	7	3
850	238	7	2
900	PLL does not support to be programmed for output frequency 900		
1000	240	6	2

Table 2: Sample of values of constants set by cfl_pll_programming function

6. CONCLUSION AND FUTURE SCOPE

The work which is carried out on the project “CFLib” shows that using functions to write a test is not only facile but also saves a lot of unnecessary effort and time. Here we conclude the main objectives of the project as convenience, accessibility, scalability and redundancy eviction.

Convenience: The test writing is more convenient with CFLib than prior approach. Now, programmers can effortlessly use the CFLib functions. The description provided with each and every function facilitates them to understand the usage of that respective function easily.

Accessibility: Project CFLib is intellectual property of CPU/GPU group. It is easily accessible to all the members of this group. If some other group in the organization needs access, it can be provided by the manager’s permissions. Also, the members in the CPU/GPU group can only read the header files and can link and use the archive library. If they need any modification, they have to contact the developers.

Scalability: ARMAR supports modification and updation in the library that has been already deployed. The developer can create the newer object files and can add them to the library. Any modification like insertion of new file, deletion and updation is possible.

Redundancy Eviction: The important feature of this project is that it eliminates the redundancy. It is placed in a centralized database from where the developers can access it. So, there will be no redundant copies of the same code on the disk.

Also, there are many other categories such as handler, debugging, boot etc. for which functions could be written and added to the CFLib library. CFLib has no boundaries; we can add any number of functions to it. ARMAR supports modifications in the deployed library file. So, although we have deployed this library but we have the features to update it and make changes in it to make it vast by adding newer function to it.

Also, Day by day new technology is evolving. According to Gordon Moore, in every one to two years, there will be twice the number of transistors on an integrated circuit (Moore's Law) [6]. The functions in library till now supports for the 32-bit processors. ARM processor Cortex A-53 is a 64 bit processor. It has a set of 64 bit registers. The next task is to make CFLib functions compatible with this 64-bit processor.

7. REFERENCES

1. *Programmers Guide: ARM® Cortex™-A Series*, Vers. 4.0, ARM, 2014.
2. *ARM Compiler Tool Chain: Assembler Reference*, Vers. 5.0, ARM, 2011.
3. *ARM Compiler Tool Chain: Using the Compiler*, Vers. 4.1, ARM, 2010.
4. *ARM Compiler Tool Chain: Linker Reference*, Vers. 4.1, ARM, 2010.
5. *ARM Compiler Tool Chain: Creating Static Software Libraries with armar*, Vers. 5.0, ARM, 2011.
6. Seal D., *Architecture Reference Manual*, 2nd ed. Boston; Addison-Wesley Professional, 2001.
7. *Architecture Reference Manual: ARMv7-A and ARMv7-R*, Vers. 1.0, ARM 2011.
8. Brylow, D.; Palsberg, J., "Deadline analysis of interrupt-driven software," *IEEE Transactions on Software Engineering*, vol.30, no.10, pp.634,655, Oct. 2004.
9. *Architecture Specification: ARM® Generic Interrupt Controller*, Vers. 2.0, ARM, 2013.
10. Gercekci, A.; Krueger, A., "Trends in Microprocessors, "*Solid-State Circuits Conference, ESSCIRC '85*, 11th European, pp.233, 233i, 16-18 Sept. 1985.
11. Zuberi, K.M.; Shin, K.G., "An efficient semaphore implementation scheme for small-memory embedded systems," *Third IEEE Proceedings : Real-Time Technology and Applications Symposium*, pp. 25, 34, 9-11 Jun 1997

Appendix A

Algorithm to calculate NDIV, IDF and ODF for programming PLL

INPUT: desired output frequency (phi)

Other Requirements: INFF (from the platform)

Valid range of: VCO, phi, NDIV, ODF, IDF

STEPS:

1. Take a default VCO value from valid range.
2. Calculate ODF with formula, ODF = round(VCO/(phi*2)).
   // use math round function for greater accuracy.
3. Now, find phi from the calculated ODF by using formula,
   phi_new = (VCO/(ODF*2));
4. Calculate ratio = phi/phi_new.
5. If ratio*VCO is out of VCO valid range then increment/decrement default VCO accordingly and goto step 2.
6. Increment IDF from 1 to 7 and calculate NDIV using formula
   NDIV = round((VCO*IDF)/(4*INFF))
   break in-between if value of NDIV becomes > 246.
7. STOP. The value of ODF, NDIV and IDF is the required values to program PLL

Appendix B

Algorithm for code serial execution

1. Get total number of cores in a cluster from L2CTLR register to a volatile global variable, say total_cores.
2. Set global variable core_to_be_exec = 0, since core 0 is considered as primary core.
3. Pass the pointer of the code (written inside a function) which is to be executed serially “cfl_gen_coreSerialExecution(void *func)”.
4. If core_to_be_exe == 0, then

   Send Set Event to wake all secondary cores from WFE mode.

   // if cores are in WFE, they wake up from WFE and if not, they ignore the signal
5. For core_to_be_exe, execute the function that is to be executed serially.
6. Increment the volatile global variable core_to_be_exe by 1.
7. Set DMB instruction to reflect the changes in global variable to all cores.
8. Send the core in WFE mode except last core.
9. The last core sends set event (SEV) instruction to all other cores to complete the serial execution and start all the cores to work parallely.

Appendix C

Video Link

https://youtu.be/ytXBL4aOvXM