TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Harsha Chaitanya Manam A156401IASM

TTU STUDENT SATELLITE OBC (ON BOARD COMPUTER) SUBSYSTEM COMPONENTS DEVELOPMENT

Master's thesis

Supervisor:	Ants Koel
	PhD
Co-Supervisor:	Kalle Tammemäe
	PhD

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Harsha Chaitanya Manam A156401IASM

TTÜ ÜLIÕPILASSATELLIIDI OBC (PARDAARVUTI) ALAMSÜSTEEMI KOMPONENTIDE VÄLJATÖÖTAMINE

magistritöö

Juhendaja:	Ants Koel
	PhD
Kaasjuhataja:	Kalle Tammemäe
	PhD

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Harsha Chaitanya Manam

[29.04.2019]

Abstract

The goal of this thesis is to develop an alternative open source development environment for TTU student satellite OBC(On Board Computer). The OBC is based on Texas Instruments TMS320C6727B floating point digital signal processor. TI provides propitiatory development environment for developing applications for the target CPU on OBC. The goal of thesis is to provide a set of open source tools against TI ecosystem for software development on OBC. This work includes building compiler tool chain, implementing peripheral drivers, interrupt support and a minimal set of Standard C library functions.

Annotatsioon

TTÜ üliõpilassatelliidi OBC (pardaarvuti) alamsüsteemi komponentide väljatöötamine

Käesoleva lõputöö eesmärk on vabavaralise arendususkeskkonna väljatöötamine TTU tudengi satelliidi pardaarvutile. Pardaarvuti kasutab oma tööks Texas Instrumendi TMS320C6727B ujukoma digitaalselt signaaliprotsessorit. Kuigi TI pakub antud platformile ka kommertsiaalseid arendusvahendeid siis selle töö peamiseks sisuks on just vabavaralise alternatiivi pakkumine. Täpsemalt sisaldab käesolev töö kompilaatori kohandamist, riistvara ajurite väljatöötamist, katkestuste tuge ja C funktsioonide teegi kompileerimist.

List of abbreviations and terms

OBC	On Board Computer
TTU	Tallinn Technical University
TI	Texas Instruments
CPU	Central Processing Unit
UHF	Ultra high frequency
JTAG	Joint Test Action Group
ABI	Application Binary Interface
GCC	GNU Compiler Collection
AST	Abstract Syntax Tree
SSA	Static Aingle Assignment
RTL	Register Transfer Language
ELF	Executable Linkable Format
COFF	Common Object File Format
PLT	Procedure Linkage Table
ROM	Read Only Memory
RAM	Random Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
UART	Universal Asynchronous Receiver Transmitter
BSS	Block Started by Symbol
PLL	Phase Locked Loop
CGT	Code Generation Tools
DSP	Digital Signal Processor
RTOS	Real Time Operating System

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

The aim of this thesis to provide an open source software development environment to the TTÜ student satellite OBC(On Board Computer) development board. Part of this work is to build a compiler toolchain and necessary software components to facilitate developers to develop applications using the environment which was created in this thesis work.

1.1.1 TTÜ Satellite Project

The TTÜ student satellite is a project of TTÜ nanosatellite programme started by Mektory Space Centre[2]. The nanosatellite program is to build a powerful nanosatellite to launch in to the orbit and also provide students a chance to learn and get experience with space and satellite technologies[2]. The following are the main components of the satellite.

Communication Subsystem:

The communication subsystem is responsible for communications between the satellite and the ground station with two different radio systems in the satellite[2][3]. The first radio provide two way communication on 435 MHz UHF(Ultra high frequency) waveband and is used for transmitting satellite communications(remote control) and satellite monitoring(telemetry)[2][3]. The second radio is operating in the 10 GHz band and is used to download satellite images[2][3]. Without a communication subsystem, the satellite would be completely "on its own", there would be no way to control the satellite and transmit data stored on it to Earth[3].

On-Board Camera Subsystem:

The On-Board camera subsystem contains two area-scan cameras with Aptina MT9P031 RGB sensor[2][4]. The main goal of the On-Board Camera subsystem is to take images with relatively high ground resolution up to (26 – 50) meters per pixel[2] [4].

Electric Power Supply Unit:

The Electric Power Supply unit is responsible for delivering uninterruptible and stable +5V supply to all components in the satellite[5]. For this the satellite has nine solar panels and Li-Ion batteries to the harvested power by solar panels[5].

On-Board Computer:

The main computer which is responsible for controlling the all the other subsystems. The On-Board Computer contains a High performance floating point Digital Signal Processor(TMS320C6727B) from TI and other peripherals for external interfacing, communication and data storage[6][1]. As this OBC is for satellite and it should be space-qualified. For main CPU TI provides a space-qualified ASIC variant with TMS320C6727B CPU.

Main focus in this thesis is the software development for the On Board Computer. This paper is going to address the problems in developing an open source development environment as a possible alternative to the existing TI Code Composer Studio based development environment and develop the software for the OBC.

1.1.2 On Board Computer Hardware details

The On Board Computer is a custom development board based on TI(Texas Instruments) TMS320C6727B floating point digital signal processor[6]. Figure 1 shows the OBC.

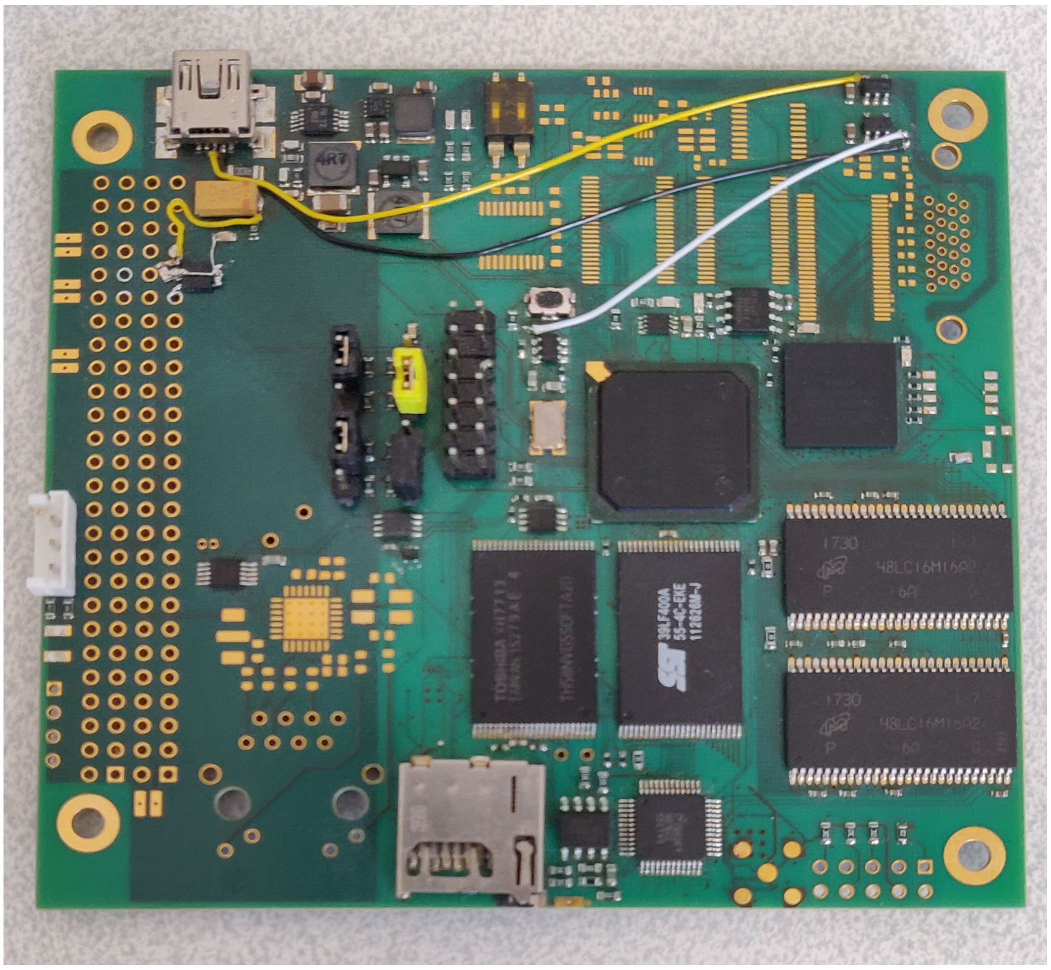


Figure 1: On Board Computer Front side[Author’s photo].

The OBC board is having the following components interfaced with the main CPU.

- 4 Gigabytes of NAND Flash for non volatile storage purpose.
- 256 Megabytes of SDRAM as fast access memory.
- 14-pin TI JTAG(Joint Test Action Group) interface for hardware enabled debugging.
- Several LEDs
- Switches
- SD-Card interface
- pinout for connecting IIC EEPROM

- UART interface to host and power supply connection through mini-USB connector.

Currently the board can be programmed through the UART interface.

1.2 Problem Statement

TI provides their own development tools to develop software for DSP CPUs which includes Code Composer Studio V3.1[7][9] and windows XP[17] as host machine(Recent versions are eclipse IDE based and not supported for our board), compiler toolchain, libraries to make development ease and proprietary JTAG(XDS560 series or XDS510 series)[8] hardware for hardware assisted debugging. This environment support only 32-bit windows XP environment. There are several development teams working on OBC software development and The TTÜ Satellite project need to buy multiple licences for each development team. The license costs are high for the funding available for the project. The other major problem is that the tools supported by the OBC CPU series and the environment supported by the tools are obsolete. The Latest TI CCS and CGT tools drop support for older CPU series. These problems leads to search for alternative environment for TI ecosystem.

2 Alternatives and Proposed Solution

2.1.1 Alternative solutions

There is no complete out of the box development environment for TI C6x series of processors particularly for TMS320C6727B. TI also provides a head-less version of compiler toolchain and necessary tools called CGT(Code Generation tools) to develop software for TI C6x series of processors[10]. These tools support different host machines and operating systems and these tool are freely available for developing software for TI C6X series of processors but with TI licence terms.

On the other side Sourcery Tools Services by Mentor Graphics[11] develop a GCC port for TI C6x with ucLinux[48] compatible ABI[12][13]. Mentor Graphics provide this toolchain for ucLinux compatible ABI not for baremetal software development and Mentor Graphics does not provide this toolchain as a binary distribution and does not provide any build specification or cross-build manual. If developers want to make use of this toolchain then they need to work on porting ucLinux for CPU(TMS320C6727B).

As part of porting GCC to TI C6X series processors, TI and Mentor Graphics submitted series of patches to GCC code base. These patches contain support for TI C67X+ CPU(TMS320C6727B)[12] variant. The support is minimal and the patches results in a bare minimum version of GCC toolchain. There is no support for any baremetal C library. This support was added by Mentor Graphics as part of their GCC port for TI C6X Series of processors.

TI also started support for Linux[14] operating system support for TI C6X series of processors[15]. Again the support is only for recent TI C6X series of processors such as TMS320C6748 series not for TMS320C6727B. TI, Mentor Graphics and the people from Linux community collaboratively started a project called linux-c6x to provide complete environment to develop Linux based applications on TI C6X series of processors[16]. This project currently support only TI C64X and C66X series of processors. The main drawback of this project is that it needs to use TI development tools. TI Code Generation Tools along with GCC toolchain to build the project. This project uses TI CGT tools to generate Linux ABI comparable objects and then use GCC linker to link these objects to generated executable binaries. Currently this project is not active anymore.

2.2 Proposed Solution

The current alternatives available for TI C6X series of CPUs does not provide a complete or ready-to-use environment for TI C67X+ CPU. The option to write a compiler toolchain require huge work and the scope ended up developing compiler than creating an ecosystem for developing software for OBC.

The other option is to use the Mentor Graphics GCC toolchain for TI C6X. As this toolchain is ucLinux compatible ABI, porting Linux to TI C67X+ CPU is needed to use this toolchain. This solution has the following problems.

- ucLinux is not real-time operating system.
- Running ucLinux need more resources when compare to other RTOS based environment.
- Porting Linux to new CPU needs lots of work and time.
- Using an ABI specific compiler limit the scope for porting other operating systems.

As the GCC toolchain is open source, Building the toolchain is possible from the source. Mentor Graphics publish their patches to GCC repository. The possible solution is to build a bare-metal toolchain, so that the toolchain is independent and more flexible to port firmware applications and RTOS schedulers.

The current thesis implement the baremetal toolchain with the necessary components to use the toolchain to develop software for the OBC.

3 System Architecture and Implementation

3.1 System Overview

This section explain the system overview. The implemented system responsible for creating development environment for OBC board. This implementation provide the developer with the tools needed for writing application code by abstracting low level system initializations. The block diagram in Figure 2 shows the functionality of the implemented system.

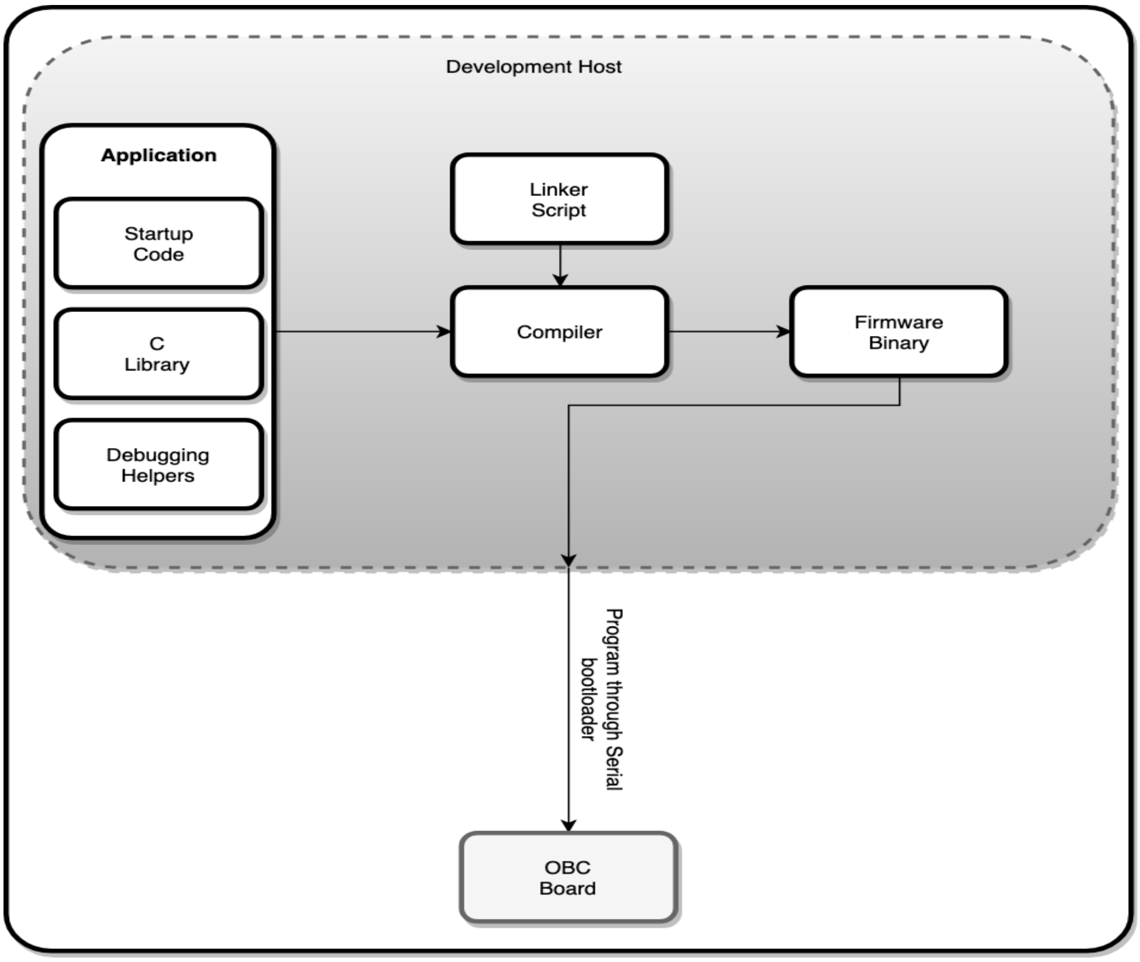


Figure 2: System overview

The developer integrate application code with the other components and use the GCC compiler to build firmware. Current implementation uses existing serial bootloader to program the binaries on to the OBC board.

The project uses GNU Makefile to integrate individual components to build the firmware. The implemented system brings bare minimum environment for application development for OBC board. Compiler optimizations, peripheral drivers for rest of the board components and any kind of application specific implementations are out of the scope for this work. This environment creates base for the developers to create applications. Tests are implemented as part of the work to verify the correctness of the each module.

3.2 TI C67X+ Architecture Overview

TI C67X+ CPU is the enhanced version of C67X CPU series from TI, a high performance floating-point digital signal processors. The maximum clock frequency of this CPU is 350 MHz[18]. At its peak frequency C67X+ CPU can achieve 2100 MFLOPS of performance. This CPU supports advanced VLIW architecture and can perform a 256 bit wide fetch. This CPU can execute up to 8 parallel instructions in single cycle and able to perform single precision(32-bit) and double precision(64-bit) floating point operations[18]. Figure 3 shows the block diagram of C67X+ CPU architecture.

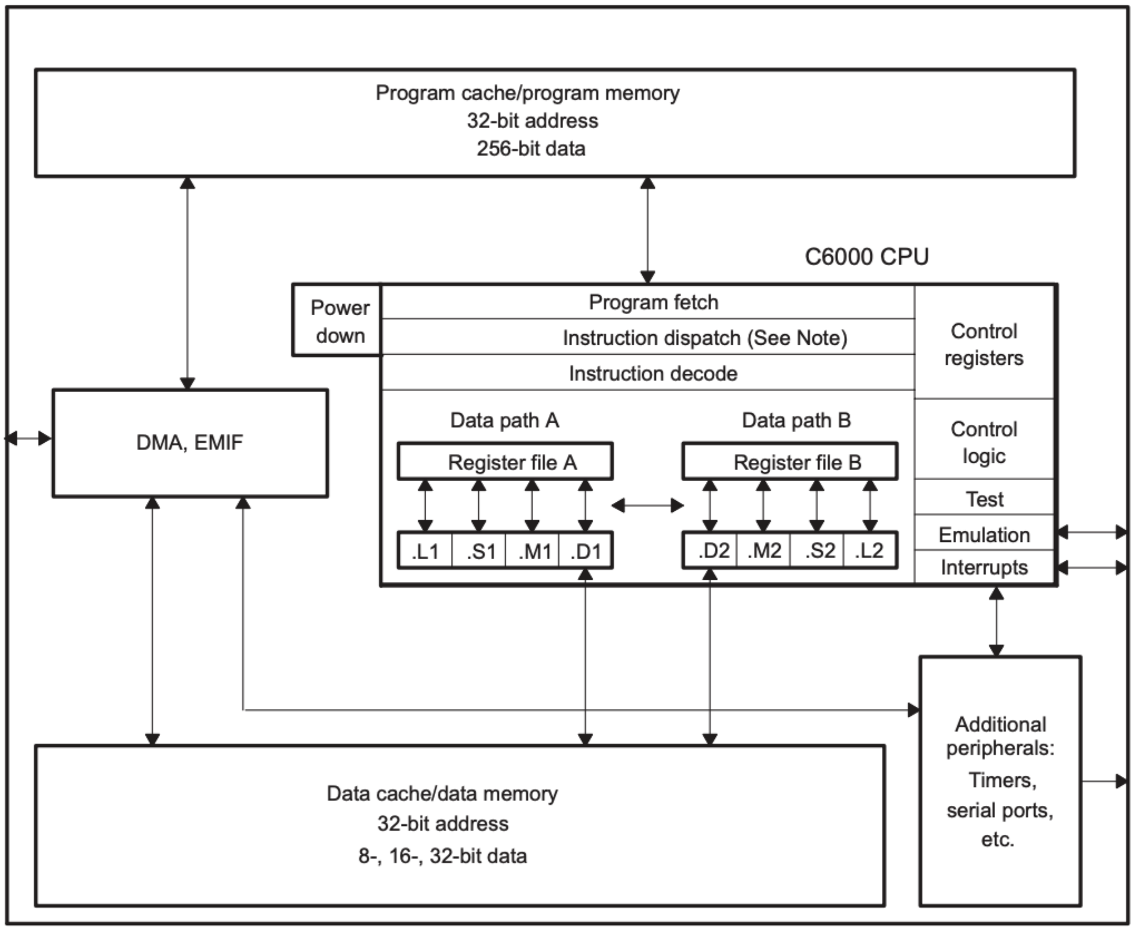


Figure 3: TMS320C67X CPU Block Diagram[19]

3.2.1 Data Path Register Descriptions and Functional Units

The C67X+ data path contains two register files each having thirty two 32-bit registers. These registers supports data lengths of 40-bit fixed point values and 64-bit floating point values. If the data lengths are higher than 32-bit values then the value will store in pairs of registers. These registers store both data and address of the data.

Each data path of register file is divided into four functional units. These functional units divided based on the operation performed by the CPU and the register file usage. All of the functional units in one data path is almost identical to the functional unit in other data path. Each functional unit is represented by Functional Unit Label(Single Alphabet letter) with data path number as subscript. The descriptions of each functional unit is described below.

.L Unit: This unit is responsible for

- 32-bit logical operations operations
- Fixed point arithmetic and compare operations on both 32-bit and 40-bit values
- floating-point arithmetic operations
- data type conversion operations(Ex: Double precision to Single precision)

.S Unit: This unit is responsible for

- shift operations on both 32-bit and 40-bit fixed point values
- arithmetic operation on only 32-bit fixed point values
- floating-point compare operations
- Branch operations
- Register to register transfer
- One special case is that control register operations can only be done in second data path(i.e in .S2 unit)

.M Unit: This functional unit is responsible for multiply operation of both fixed point and floating values. This unit also supports mixed precision multiply operations.

.D Unit: This functional unit is responsible for data load store operations and address arithmetic operations.

All the write operations in these functional units operates on the registers from the same functional units. Writing data from a register from one data path to another data path needs to use cross paths each data path has one cross path denoted by data path number followed by letter ‘X’(i.e 1X and 2X). Apart from this cross path access each data path contains paths to access data address(i.e memory addresses). These data addresses for each data were accessed by DA1 and DA2 respectively. More details of how to use these functional units in assembly language is explained in later sections.

3.2.2 CPU Control Registers

TI C67X+ CPU has 13 control registers. These control registers are responsible for CPU control path. All these registers can only access from second data path. Not all these registers are accessible for writing. Some of these registers are for representing status of CPU. Table 1 shows the list of control registers.

Register Name	Description
AMR	Addressing Mode Register
CSR	Control Status Register
FADCR	Floating-Point Address Configuration
FAUCR	Floating-point auxiliary configuration
FMCR	Floating-point multiplier configuration
ICR	Interrupt Clear Register
IER	Interrupt Enable Register
IFR	Interrupt Flag Register
IRP	Interrupt Return Pointer
ISR	Interrupt Set Register
ISTP	Interrupt Service Table Pointer
NRP	Non-Maskable Interrupt Return Pointer
PCE1	Program Count E1 Phase

Table 1: TI C67X+ Control Registers[20]

The following content in this section will explain brief description of control registers.

AMR: This register controls the addressing mode. The affected registers by this control register are from register 4 to 7(i.e A4 to A7 and B4 to B7). This register is 32-bit wide and allocate two bits for each register addressing mode configuration. There are also two block sizes specified in BK0 and BK1 to use with circular addressing mode. The addressing mode option values are:

00: Default linear addressing mode. This is the reset value.

01: Circular addressing mode with block size specified in BK0

10: Circular addressing mode with block size specified in BK1

11: Undefined.

CSR: Control Status Register responsible for controlling and providing status of several CPU functionalities. See section **2.7.4** in **Reference 20** for detailed information about all bit fields in CSR register.

ICR: Interrupt Clear Register is responsible for clearing interrupt status of all maskable interrupts. Setting(Writing 1) specific bit in this register clear the related interrupt flag.

IER: Interrupt Enable Register is responsible for enabling specific interrupts. Setting bits in this register enable Interrupt on specific interrupt lines.

IFR: Interrupt flag register holds the current enable/disable status of all interrupts. This is a Read-Only Register.

IRP: Interrupt Return Pointer register holds the return address for ISR(Interrupt Service Routine). When Interrupt occurs CPU starts executing corresponding ISR routine. After theISR routing complete its execution, the control flow returns to normal program execution by branching to the address provided by IRP.

ISR: Interrupt Set Register is responsible for triggering interrupts. Setting bits in this register trigger interrupts on specific lines.

ISTP: Interrupt Service Table Pointer holds the starting address of ISR Vector table. More detailed explanation about ISTP will found in section 2.5.5.

NRP: Nonmaskable Interrupt Return Pointer is like IRP but, it holds the return address for NMI interrupt handler.

PCE1: Program Counter Phase E1 holds the current fetch packet in E1 pipeline execution.

3.2.3 Instruction Pipeline

Instruction pipeline is a method to achieve instruction level parallelism. In instruction pipeline, the instruction execution is divided into stages and these stages run in parallel. TI C67X+ CPU implement three stage pipeline. They are

- Instruction Fetch
- Instruction Decode
- Instruction Execute

Each stage of the pipeline is divided in to several stages. The Fetch stage is divided into four stages, Decode stage is divided into two stages and Execute stage is divided into several stages based on the instruction with a maximum of 10 stages[20]. There is no pipeline interlocking mechanism in TI C67X+ CPU to check for pipeline hazards. This is because of the pipeline simplicity[20].

Fetch stage can fetch eight instructions and all the eight instructions pass through the all stages of the Fetch stage. The internal stages of Fetch phase is as follows

- **PG(Program address generate):** Program address is generated in the CPU.
- **PS(Program address send):** Program address is sent to the memory.
- **PW(Program access ready wait):** Memory read occurs.
- **PR(Program fetch packet receive):** Fetch packet is received by the CPU.

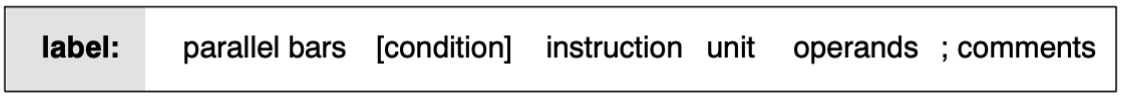
The next phase in pipeline is Decode. In this phase the Fetch Packet from Fetch phase is divided into execute packets. The Decode phase is divided into two internal stages

- **DP(Instruction Dispatch):** The instruction is assigned to appropriate functional unit.
- **DC(Instruction Decode):** The instruction is decoded and all the source and destination registers and related paths are resolved here[20].

The final phase in pipeline is Execute. This stages is divided into ten stages(from E1 to E10). But not all the instructions will require all ten stages. Based on the instruction the Execute phase span into multiple stages.

3.2.4 ISA and Assembly Program Structure

The format of assembly instructions for TI C6X architecture consists of seven blocks. Figure 4 shows how the parts of assembly is ordered to form a statement.



label: parallel bars [condition] instruction unit operands ; comments

Figure 4: Assembly instruction format[18]

Label: On high level functionality Label brings a name to a particular piece of code. In other terms Label holds address of an instruction where its defined.

Parallel bars(||): This is specific to TI C6X architecture. Parallel bars mark the instruction to execute in parallel with the previous instruction. This field can left empty for non parallel instructions.

[condition]: Condition makes the instruction execute or not based on the value of the register in the condition.

Instruction: Instruction is the actual operation to be performed in execution.

Unit: appropriate functional unit for the instruction

Operands: Operands for the instruction will be a register, memory address or immediate value. The exact requirements for the operands are based on the instruction and the functional unit(or the functional unit can be adjustable based on the operands).

; (comments): The comments for the code in asm is started with a semicolon(';').

3.3 GCC toolchain architecture

GCC(GNU Compiler collection) is part of GNU toolchain system, a collection of individual tools works in a pipeline to form a compiler toolchain. GCC initially developed only for C language. Now GNU toolchain system will support several languages, frontends on wide variety of operating systems and hardware[22]. Currently GCC is the system compiler for Unix and Linux based operating systems. GCC also used in low level firmware development. GCC bare-metal port is available and widely used by most of the hardware vendors. GCC is retargetable and re-sourcable. It means GCC supports many language frontends and generate code for multiple hardware types. This nature made GCC a widely used compiler for many new hardware architectures.

On higher level GNU toolchain compiling pipeline mainly consists of three tools. Compiler and preprocessor, Assembler and Linker. GCC compiler consists of the following components. Front-ends, middle-end and back-end. The output of the back-end is the machine specific assembly code. GNU Assembler will convert the assembly code into object code. From here the linker job will start. The linker will combine several object code files into single executable binary. Linker also do address relocations and linking external (shared or static libraries). In every stage of the compiler pipeline the tools required one or more input files and configuration options.

This section will explain architecture details and internals of GCC compiler. Later sections will explain the functionality of assembler and the linker which are the main components of the pipeline. The block diagram in Figure 5 shows the high-level functional architecture of GCC compiler.

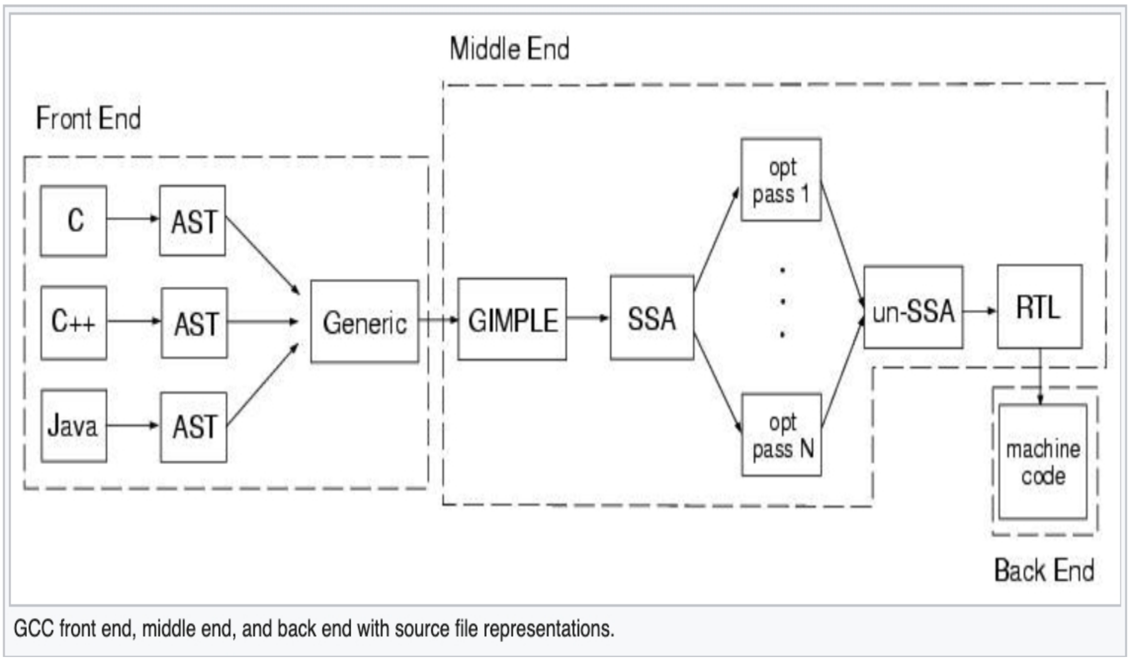


Figure 5: GCC functional architecture[23]

3.3.1 Front End

In general a language front end consists of Tokenizer(also called Lexer) and Parser. Front end reads the plain source file and converts it into AST(Abstract Syntax Tree) form by passing through Tokenizer and Parser. There are several steps in converting plain source cod into AST form. First the front end reads the code and generate tokens. Each language has different tokenizing mechanisms. Once tokens are generated then these tokens are passed to parser. The parser reads the tokens and generate intermediate code in AST data structure form. In GCC toolchain the AST is converted into another unified form called GENERIC[25].

GENERIC representation is a language independent intermediate code. As GCC have multiple language frontends, each front end produce different AST form. For portability GCC converts all language specific AST into one common form(GENERIC).

This GENERIC AST form will goes into Middle End as input.

3.3.2 Middle End

The Middle end converts the GENERIC AST into another representation called GIMPLE. GIMPLE representation is influenced by SIMPLE IL intermediate representation used in McGill University McCAT compiler program[26]. GIMPLE is the first stage code in Middle end. All architecture independent optimizations are done in GIMPLE representation.

In GIMPLE representation, all the expressions from GENERIC are represented in tuples of maximum 3 operands. GIMPLE tuples are divided into two groups. First a header describing instructions and its locations followed by a variable length body representing operands. The variable length body divided into three classes:

gimple(gbase): This is the root of gimple body. It holds the basic information needed by all the GIMPLE statements[27]. This structure takes 32 bytes.

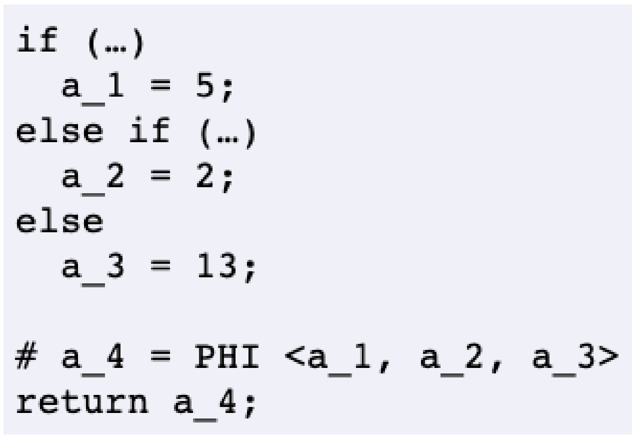
gimple_statement_with_ops: This structure holds statements with operations. This is also used in the statements with memory operations.

gimple_statement_with_memory_ops: This structure is identical to operation statements and an extra 4 fields to holds statements with operations related to memory load and store.

All the GIMPLE statements form in a hierarchy such that all the operations are followed by gbase and irrespective of the front end all the statements use same GIMPLE instruction set[28].

After the GIMPLE code is converted into SSA(Static Single Assignment). SSA representation is used by GCC for optimizations and all the optimizers to pass in SSA representation. In SSA representation all the variables are assigned only once. If any any variable is reassigned second time then the compiler create new variable assign the value to the new variable. All the new variable created for other assignments is renamed by subscripting the assignment occurrence number.

The following code in Figure 6 shows variable assignment in SSA form.



```
if (…)
  a_1 = 5;
else if (…)
  a_2 = 2;
else
  a_3 = 13;

# a_4 = PHI <a_1, a_2, a_3>
return a_4;
```

Figure 6: Sample SSA code[28]

In the above code, variable ‘**a**’ is assigned three different places and the code returned the value in ‘**a**’. As shown in the code the variable ‘**a**’ is renamed as ‘**a_1**’,’**a_2**’,’**a_3**’ for three different assignments. When it comes to return the value of the variable ‘**a**’, the code don’t know which value of the variable to return. To solve this kind of problems in code with branches, the compiler creates new variable with same naming conventions(with assignment occurrence value subscript) and use SSA statement called “PHI<>”. The PHI statement meaning that any of the variable in ‘**a_1**’,’**a_2**’,’**a_3**’. So variable ‘**a_4**’ holds the final value and the code returns the value in ‘**a_4**’. All SSA statements are placed in Reference[28].

Once GCC ran all the optimizers on SSA code then the SSA code then converted back to GIMPLE tree then passed into RTL(Register-Transfer Language) representation. RTL is also called generic machine code(abstracted machine code) form is the lowest level intermediate representation. RTL is representation is more like Lisp lists[30][31] syntax[29]. RTL represent low level features like registers, addressing modes, data structure sizes, calling conversions and bit fields. The reference to RTL statements found in Reference[29].

3.3.3 Back end

Back end is responsible for true machine code generation. It generates target hardware specific assembly code. The descriptions of target architecture configurations found in “gcc/config/<arch>” directory. The <arch>.md file describes all of the target specific information which includes:

- Register names and functional units
- Target variants
- Instruction attributes specific to different CPU types
- Relation between instructions and operands
- Cycle count for instructions
- Addressing modes
- Predicates and predicate patterns
- Control flow and branching

The <arch>.[hc] files holds the code for code generation functionality which described in <arch>.md file. Some of the instructions specified by target file is not directly converted into machine code instead of the compiler helper library generate assembly code. For example soft-float operations and divide operations are software algorithms implemented in compiler helper library.

From plain high level source code to assembly code generation is part of compiler job. Now the GNU Assembler which is part of GNU binutils[31] will convert the plain text based assembly code into target ABI(Application Binary Interface) specific object code.

3.3.4 GNU Assembler

GNU Assembler or “GNU as” is the assembler in GNU compiler toolchain[32]. The main job of the assembler is to convert plain text assembly file into object file in target ABI format. Typical object formats includes ELF[33](Executable and Linkable Format) and COFF[34](Common Object File Format). This conversion of assembly text file into object format includes:

- Transform asm statements into Target opcodes

- Assign relative addressing for the instructions
- produce PLT(Procedure Linkage Tables)
- All external functions addresses are calculated based on the PLT offsets

The two main components in GNU assembler are

CPU Backend: Responsible for describing target CPU instruction set

Object-Format Backend: Responsible for generating the pseudo instruction from the CPU Backend to target ABI format.

3.3.5 GNU Linker

GNU linker is responsible for generating executable files or shared objects by taking object file as input. An ld script is used to control the ld generating executable files. Main function of the linker includes:

- Replace relative address with (pseudo)physical address.
- Symbol relocations.
- Resolving external symbols
- Linking with external libraries

In GCC toolchain ld is triggered by another binary called “**collect2**”.

3.3.6 libgcc(GCC compiler support library):

The libgcc library is support library for GCC runtime. It supports GCC for compiling and code generations. GCC cannot perform translations for complex instructions such as multiply and divide operations or multi precision arithmetic operations for some architectures. In such scenarios GCC uses libgcc help to generate code for the statements which are complex to translate for GCC.

Libgcc also provide several helper functions for supporting GCC language front end(Ex: memory and string operations for complex C statements) and debugging support for getting calltraces and unwind operations.

For TI C67X+ architecture GCC does not support hardware baked floating-point instructions to generate code for high level statements with floating point operations. To cover this problem GCC depends on the libgcc helpers for generating code for multiply and divide operations. The architecture specific functionality of TI C6X in libgcc is located in "**<GCC Root>/libgcc/config/c6x/**" path. There are several important functions which GCC compiler is using while compiling the code.

lib1funcs.S: This assembly file implements divide function stubs in assembly. These stubs later called by GCC compiler whenever it tries to generate code for divide operations.

libunwind.S, unwind-c6x.c, unwind-c6x.h: These source code files are responsible for implementing stack unwind and calltrace dump functions.

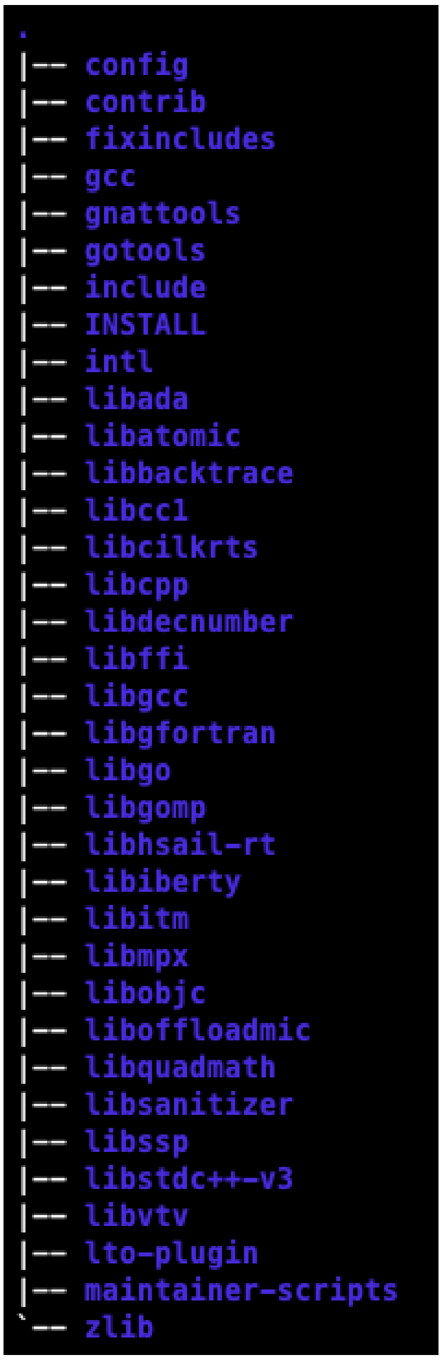
libgcc/soft-fp: This directory contains code for soft floating point operations.

Later sections will explain the procedure for building a GCC cross toolchain targeted for specific architecture and runs on the host computer.

3.4 Building GCC for TI C67X+

3.4.1 Source code organization

The GCC source span into multi-level directory structure and consists core source, runtime support libraries, test suites and build system files. The following tree diagram in Figure 7 shows the first level source structure.



```
.
|-- config
|-- contrib
|-- fixincludes
|-- gcc
|-- gnattools
|-- gotools
|-- include
|-- INSTALL
|-- intl
|-- libada
|-- libatomic
|-- libbacktrace
|-- libcc1
|-- libcilkrts
|-- libcpp
|-- libdecnumber
|-- libffi
|-- libgcc
|-- libgfortran
|-- libgo
|-- libgomp
|-- libhsail-rt
|-- libiberty
|-- libitm
|-- libmpx
|-- libobjc
|-- liboffloadmic
|-- libquadmath
|-- libsanitizer
|-- libssp
|-- libstdc++-v3
|-- libvtv
|-- lto-plugin
|-- maintainer-scripts
`-- zlib
```

Figure 7: GCC source tree structure

Detailed description about each directory is explained below:

config: Contains all top level Makefiles and autoconf configure scripts.

contrib: Several helper tools/scripts which helps in pre/post GCC building. These scripts are contributed scripts not part of the core system.

fixincludes: This directory contains code responsible for fixing the system header whenever the build has problems with system headers.

gcc: This core directory that contains all the source code of the GCC compiler. Later sections will explain about the internals of this directory.

gnattools: This directory responsible for GNAT ADA compiler toolchain support.

gotools: This directory contains support tools for Go language.

include: This directory contains internal include headers.

intl: libintl library code.

libada: ada core runtime library.

libatomic: library to add support for atomic operations.

libbacktrace: library to provide detailed symbolic backtrace of running code.

libcc1: compile-code support library for GNU Debugger GDB.

libcpp: Library for C language preprocessor.

libdecnumber: Library for decimal arithmetic.

libgcc: GCC core compiler support library.

libgfortran: GNU fortran support library.

libgomp: Library for OpenMP support.

libiberty: Core data structures and helper functions.

libquadmath: Quad precision mathematic operations runtime.

libsanitizer: Sanitizer functions.

libssp: Stack-smashing-protection library.

libstdc++V3: C++ Language runtime support library

lto-plugin: Link-Time-Optimization helper function for GNU linker.

The core GCC source which includes fronts ends, middle ends, backends and architecture specific code generation function are inside gcc subdirectory. This directory is divided into several directories. Main components in this directory will explained below.

The **C** directory contains C language front end support(Not the actual front end code). **CP** directory contains C++ language front end. The **doc** directory contains user manuals. The **ginclude** directory contains system headers. The **testsuite** directory contains all kinds of tests implemented for testing GCC toolchain. The **config** directory contains architecture specific code generation functionality. Each target architecture specific code is placed in a separate directory with the target architecture name.

The gcc root directory contains all the architecture independent and language independent core functionality such as internal data structures, optimizers for several stages(mainly for GIMPLE and RTL), build support and other features like user interface options handling. The C language front end is also resides in GCC root directory. The reason for placing C language front end in gcc core directory is the early days of GCC development. When GCC compiler was started it only supports C language and later the support for other languages are added and the structure for other languages were went into separate directories.

3.4.2 Required components to build the compiler

GCC compiler toolchain is a set of individual tools works interdependently to form a complete source to target binary pipeline. To build complete compiler toolchain with all individual tools including assembler and linker the build system need the following components[51].

gcc: GCC source is the core compiler. It contains all language runtime and support libraries.

binutils: binutils is a collection of utilities to create and modify binary files of various targetted ABI formats. The main tools includes assembler, linker, nm, objdump, objcopy and ar and many more. Some of these tools are required in compilation pipeline and some of the tools are required for post compile manipulations of generated object and binary files..

cloog: cloog(Chunky Loop Generator)[35] is a helper library for generating optimized code for loops in languages like c and fortran. If GCC is using optimization options like graphite optimizations[36] then the toolchain should build with cloog support.

gdb: GDB, The GNU Debugger[37] is the main debugger for GCC toolchain. GDB supports various languages and target architectures. Various debugging frontends internally use gdb as the debugging client either hardware assisted debugging or software on-host debugging.

gmp: GMP[38] is multi precision arithmetic library used by GCC compiler to generate optimized code for mathematic operations in the source code. GMP supports various high-level fixed and floating point operations.

isl: ISL(Integer Set Library) is a library to do operation on sets of integers[39]. GCC uses this library as internal support library for optimizations in middle end.

libelf: libelf[40] library provide functions to create, read, modify elf format files. GCC toolchain need this for several binaries such as readelf, objdump.

mpc: GNU MPC[41] is a library which provide function for arithmetic operation on high precision complex numbers.

mpfr: GNU MPFR[42] is a library which provide multi precision floating point computations. GCC needs this library to generate code for floating point arithmetic operations.

The above mentioned components are required to build GCC toolchain with C language support. Other language frontends need more components need to build with GCC[52].

3.4.3 Building Steps

The build procedure of GCC toolchain is divided into multiple stages. All the stages are depend on the previous stage outcome. The bare metal GCC compiler is targetting for TI C67X+ architecture and Linux as host and build machine. This build procedure also build binutils and GDB as part of the toolchain. See the **Appendix A** for actual build steps. All the stages of build procedure were explained below. All the sources specified in section 2.3.2 are required to download before starting the build.

Stage 1(Configuring the source): The GCC compiler use autoconf[43] generated configure script to configure the to build. This configure script support various configuration options to set GCC options to build. The main configure options are:

--host: This option needs to be specified to tell the configuration script on which host architecture the compiler toolchain is going to run. The produced compiler toolchain binaries run on the host specified in this option.

--build: This option tells the GCC configuration script on which host architecture the compiler build is going to run. This doesn't mean that the generated toolchain will run on the build machine. The --host and –build options can be different architecture variants.

--target: This option tells the GCC configuration script that the GCC toolchain built with configuration will generate code for specified target architecture. It means after the build, the GCC compiler binaries will run on the machine specified by --host option and will generate code for the architecture specified in the --target option.

--prefix: Build needs this option to install the compiled binaries as part of the build process.

--enable-languages: This options tells the build system which language frontends are going to build.

--disable-threads: This option tell the GCC to generated code for single thread system. This build is to generate a bare metal toolchain, the code will run in single thread manner. This option make sense when targetting the compiler for Operating System based ABI code generation.

--with-gnu-as: This option tells the GCC build system to build the toolchain with GNU assembler as the assembler in compilation pipeline.

--with-gnu-ld: This option tells the GCC build system to build the toolchain with GNU ld as the linker in compilation pipeline.

There are several other options needed to specify. See the build.sh mentioned in **Appendix A**.

Stage 2(Compile only compilers): GCC is Makefile[44] based build system. It make the GCC build system to split the build stages into Makefile targets. The first stage in build is to generate bootstrap compilers which are going to use to build freestanding support libraries in later stages. The Makefile target name for this is **all-gcc**. This target build the freestanding minimum compilers for internal usage. After that the generated compiler needs to be installed in the specified location.

Stage 3(Compile GCC runtime support library): This stage is responsible for compiling the runtime support library for GCC compiler. This library is called **libgcc**. The Makefile target for building libgcc is **all-target-libgcc**. As the name says the libgcc is for target architecture. So libgcc is build by the compiler built in Stage 1. After compiling the libgcc library, the compiled library needs to be installed into specified location.

Stage 4(Build final toolchain): Now the final toolchain building by using the compiler from Stage 1 and the runtime support library in Stage 2. The makefile target for this stage is **all**. After this stage the full baremetal compiler including assembler, linker and all binutils is generated.

Now the final part is to build GDB. Building of GDB is external. To build GDB a generated compiler is needed. GDB is also configurable with autoconf generated configuration script and makefile based build system. The –target option is needed to specify the target architecture for the build system. Run the make file with default target and install the binaries in prefix path specified in GCC building.

After building GDB the complete bare metal GCC toolchain for TI C67X+ target is ready to use.

3.4.4 Build problems

Building GCC for cross-platform development is a complex work. For well known architectures this is not an issue. All the vendors provide build scripts to build the GCC cross compiler for their CPU. GCC for TI C6X is not a complete product and there is not documentation about how to build the compiler for targetting TI C6X architecture.

All the build manuals provided by GCC is generic to host machine builds. Manual understanding of the build system is needed to start building the compiler from source. It took so long to build a stable and working version of the compiler. Most of the time the work done results in reinventing the cycle. Most of the errors are because of incorrect build configuration or compatibility between the tools required for building GCC. Out of all analysis the following are the major problems occurred during the compiler building.

Building compiler without libc support:

Building a baremetal compiler without libc support looks strange. But the current environment available in GCC source is to build the compiler for tests(Which is the main target of the TI C6X support patches in GCC compiler source repository) or build compiler with uclibc-ng[https://uclibc-ng.org/] library which is linux compatible ABI not for bare-metal applications. One option is to build GCC without libc support and write tiny libc and compile it with the bare minimum version of the compiler. Meanwhile rest of the software development completely relies on the libgcc runtime support library. GCC uses this library internally and developers don't link this library externally. As an alternative programs link this library explicitly to use the helper functions such as memcpy, strcpy and more in the application code.

Compilation errors in libunwind support functions in libgcc:

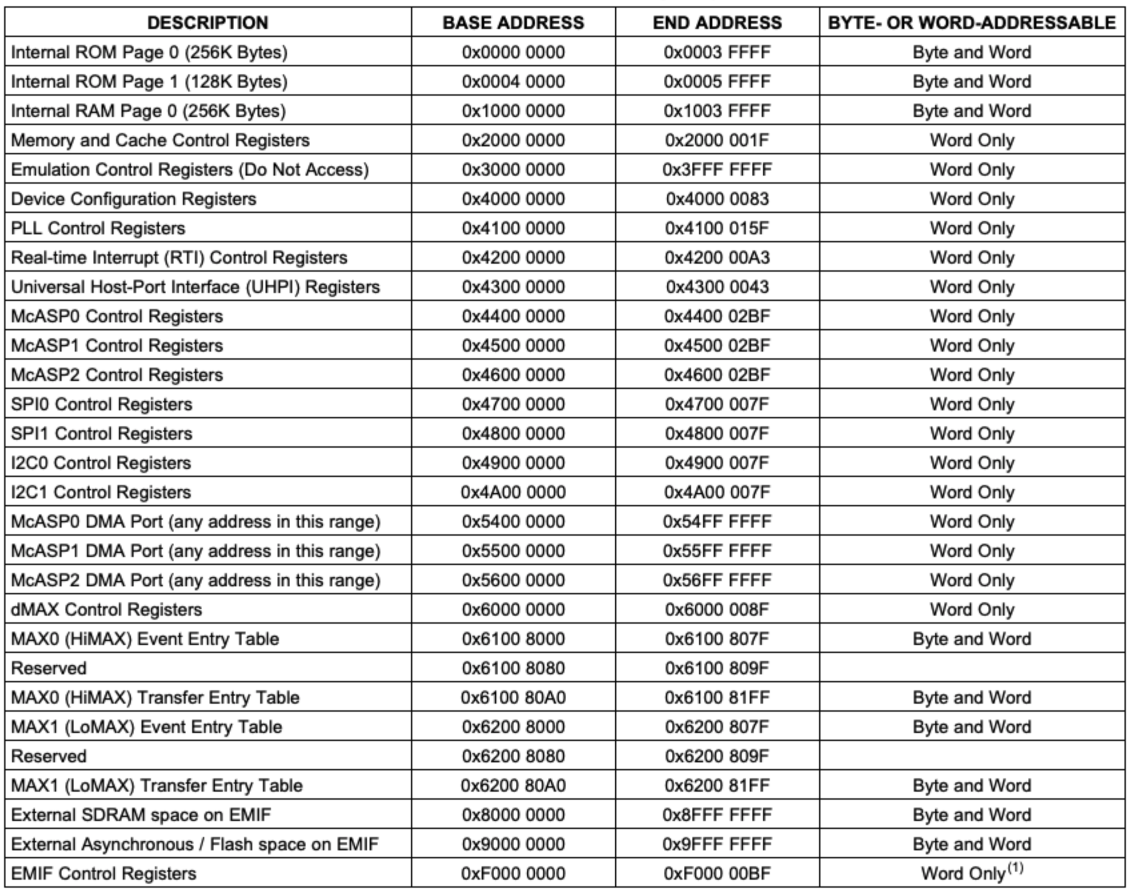
As mentioned in section 2.2.6, libgcc has stack unwind functionality. This feature is used to implement debug features. But libunwind.S assembly file the support for TI C67X+ architecture variant is partially done. The build starts throwing errors saying that the instructions are for incompatible architecture when unwind support is enabled in build configuration. This error was fixed by adjusting the assembly instructions for C67X+ architecture.

GCC assembler internal error: GCC assembler is getting this error while try to compile program using the generated compiler. This error is because of incompatible version of binutils using to build the toolchain.

3.5 TI C67X+ CPU Startup code

3.5.1 Memory Structure

TI C67X+ has flat 32-bit memory structure. All the peripherals and memory components are mapped into a linear 32-bit address range. The table in Figure 7 shows the memory map of peripherals.



DESCRIPTION	BASE ADDRESS	END ADDRESS	BYTE- OR WORD-ADDRESSABLE
Internal ROM Page 0 (256K Bytes)	0x0000 0000	0x0003 FFFF	Byte and Word
Internal ROM Page 1 (128K Bytes)	0x0004 0000	0x0005 FFFF	Byte and Word
Internal RAM Page 0 (256K Bytes)	0x1000 0000	0x1003 FFFF	Byte and Word
Memory and Cache Control Registers	0x2000 0000	0x2000 001F	Word Only
Emulation Control Registers (Do Not Access)	0x3000 0000	0x3FFF FFFF	Word Only
Device Configuration Registers	0x4000 0000	0x4000 0083	Word Only
PLL Control Registers	0x4100 0000	0x4100 015F	Word Only
Real-time Interrupt (RTI) Control Registers	0x4200 0000	0x4200 00A3	Word Only
Universal Host-Port Interface (UHPI) Registers	0x4300 0000	0x4300 0043	Word Only
McASP0 Control Registers	0x4400 0000	0x4400 02BF	Word Only
McASP1 Control Registers	0x4500 0000	0x4500 02BF	Word Only
McASP2 Control Registers	0x4600 0000	0x4600 02BF	Word Only
SPI0 Control Registers	0x4700 0000	0x4700 007F	Word Only
SPI1 Control Registers	0x4800 0000	0x4800 007F	Word Only
I2C0 Control Registers	0x4900 0000	0x4900 007F	Word Only
I2C1 Control Registers	0x4A00 0000	0x4A00 007F	Word Only
McASP0 DMA Port (any address in this range)	0x5400 0000	0x54FF FFFF	Word Only
McASP1 DMA Port (any address in this range)	0x5500 0000	0x55FF FFFF	Word Only
McASP2 DMA Port (any address in this range)	0x5600 0000	0x56FF FFFF	Word Only
dMAX Control Registers	0x6000 0000	0x6000 008F	Word Only
MAX0 (HiMAX) Event Entry Table	0x6100 8000	0x6100 807F	Byte and Word
Reserved	0x6100 8080	0x6100 809F	
MAX0 (HiMAX) Transfer Entry Table	0x6100 80A0	0x6100 81FF	Byte and Word
MAX1 (LoMAX) Event Entry Table	0x6200 8000	0x6200 807F	Byte and Word
Reserved	0x6200 8080	0x6200 809F	
MAX1 (LoMAX) Transfer Entry Table	0x6200 80A0	0x6200 81FF	Byte and Word
External SDRAM space on EMIF	0x8000 0000	0x8FFF FFFF	Byte and Word
External Asynchronous / Flash space on EMIF	0x9000 0000	0x9FFF FFFF	Byte and Word
EMIF Control Registers	0xF000 0000	0xF000 00BF	Word Only$^{(1)}$

Figure 8: TI C67X+ Memory Map[18]

TI C67X+ cpu has 384KiloBytes of internal ROM divided into two pages(Page 0 is 256KiloBytes wide which is mapped at address 0x00000000 and Page1 is 128KiloBytes wide which is mapped at address 0x00040000) and 256 KiloBytes of Internal RAM which is organised into a single 256 KiloBytes page which is mapped at address 0x10000000.

The OBC board also interfaced different types external memories to the C67X+ CPU.

- 4 GigaBytes of NAND Flash mapped at address 0x90000000
- 256 MegaBytes of SDRAM mapped at address 0x80000000

3.5.2 Boot modes

TI C67X+ CPU always boots from internal ROM at address 0x00000000. The Internal ROM bootloader will proceed to execute the code form one of the interfaced memory based on the boot mode configured by CFGPIN0 and CFGPIN1 registers[18]. The internal ROM bootloader capture the status of the CFGPIN0 and CFGPIN1 values after the reset and proceed to boot the code from appropriate location. The CFGPIN0 and CFGPIN1 configuration can be modified by the boot mode jumpers available on the OBC board. Table 2 show the boot pin configurations for the ROM bootloader.

BOOT MODE	UHPI_HCS	SPI0_SOMI	SPI0_SIMO	SPI0_CLK
UHPI	1	BYTEAD	FULL	NMUX
Parallel Flash	1	0	1	0
SPI0 Master	1	0	0	1
SPI0 Slave	1	0	1	1
I2C1 Master	1	1	0	1
I2C1 Slave	1	1	1	1

Table 2: OBC Boot Pin Configurations

Currently the OBC board supports booting from Parallel Flash. The second stage bootloader is currently running from NAND flash. It has the following functionalities.

- Starts the board and wait for the user for 3 seconds.
- If it receive any response from user it starts executing code from the address 0x91000000.

- If the user press ‘p’ then it waits for the hex input over UART and store the code at the address 0x91000000. The Developer who wrote the code need to configure the linker script with respective to this address
- If the user press ‘r’ then the bootloader continue to execute from the address 0x91000000.

3.5.3 Board initial startup code

The startup code for the OBC is responsible for running the application firmware. The startup code perform the following functions before starting the application firmware.

1. Setup stack
2. Zero BSS section.
3. Relocate application code from flash to internal RAM
4. Start the application code which will locate at the start address of the internal RAM.
5. Configure Cache
6. Configuring PLLs
7. Configuring Interrupts
8. Rest of the hardware functionality is configured by the application code based on the application

The startup code is divided into two parts. First part consists of a startup assembly code and C code for doing code relocation from Flash to internal RAM. The linker script is responsible for generating address for the application code and the startup code. When bootloader starts the code from flash, it start the startup code. The startup code runs from flash and relocate the code to internal RAM and pass the execution to relocated code.

The application code is responsible for rest of the hardware initialization.

3.5.4 Linker Script

The linker script direct the linker to properly arrange the code based on the script specifications. For the OBC board to run the applications from internal RAM, the linker script confirmation will be the following manner.

Description of the memory layout: The Linker script defined two types of memory

- FLASH: A Read-Write-Executable memory starts at address 0x091000000 and 64 KiloBytes length. Linker script is representing this region of memory as FLASH region. So the all the code goes to this location, linker generates address relative to the starting of FLASH.
- RAM: A Read-Write-Executable memory starts at address 0x10000400 and 128 KiloBytes length. This memory section is representing the internal RAM. All the code placed under this memory region, the linker will generate address relative to the RAM.

The later part of the linker is to arrange the sections in the specified memories. Apart from the standard sections generated by GCC compiler, there is a special section to handle the startup code. Linker script creating a new section called ".startup" and place this section in FLASH section. For Interrupt vectors and Interrupt service handlers stubs another section called "**.vectors**" is used.

Once the startup code is properly running, the next task is to bring some of the basic peripherals for bare minimum application. Later section 2.5 will explain about the basic peripherals working as part of the bare minimum setup.

3.5.5 PLL Controller

In general all the components in the chip will not run at same frequency. Providing clock source for all the components with right frequencies is a complex work. To address this every processor system has clock controller subsystem. The PLL(Phase-Locked Loop) controller in the C67X+ CPU system is responsible for proper distribution of clock input with configured frequencies to the all components in the chip. The block diagram in Figure 8 show the clock distribution to various peripherals in the system.

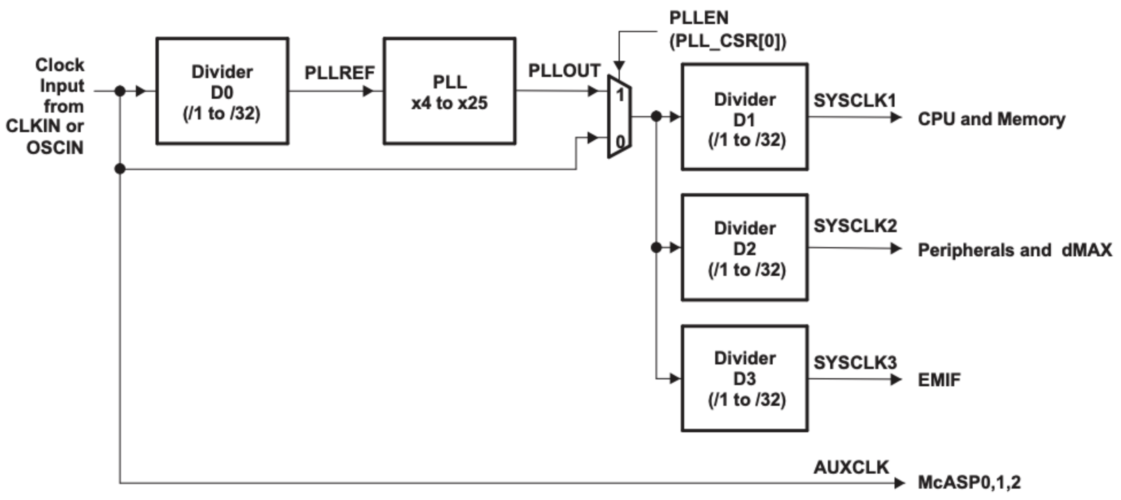


Figure 9: PLL clock distribution[18]

From the Figure 8 the PLL controller has one multiplier and four dividers. The PLL get the input clock either from internal on-chip oscillator(OSCIN) or from external clock(CLKIN)[46]. The clock is programmable using the corresponding dividers and the multiplier. The following are configuration options

- Divider 0(D0) is to divide the clock by %1 up to %32.
- The multiplier(PLL) is to multiply the clock frequency by x4 up to x25
- It is also possible to completely bypass the D0 and PLLM by configuring PLL_CSR bit in PLLEN Register.
- From PLLM the clock divided into three paths.
  - SYSCLK1 with Divider 1(D1) for CPU and memory
  - SYSCLK2 with Divider 2(D2) for peripherals and dMAX(dual Data Management Accelerator) controller.
  - SYSCLK3 with Divider 3(D3) for EMIF(External Memory Interface).
- There are components such as McASP(Multi channel Audi Serial Port) take the direct input clock bypassing all dividers and multipliers.

PLL controller provide control registers to programatically control the clock distribution for all components.

The current startup code has complete implementation for configuring clocks. All the helper functions and macros to program the PLL subsystem are available with the code. With the current configuration the OBC will run with 300 MHz CPU frequency.

3.5.6 Interrupts

Interrupts are asynchronous events generated by CPU to stop the current execution flow and to attend the respective event processing. Interrupts are key component in any computer system which need to process external events with out explicitly poll and schedule for the events to occur. In any embedded system interrupts are the best way to get notified about the asynchronous events which need CPU attention.

The TI C67X+ architecture Interrupt subsystem consists of 16 interrupt sources distributed across all the components. Table 3 show the interrupt assignment for various components.

CPU Interrupt	Source
INT0	RESET
INT1	NMI
INT2	Reserved
INT3	Reserved
INT4	RTI Interrupt 0
INT5	RTI Interrupts 1, 2, 3 and Overflow RTI Interrupts 0 and 1
INT6	UHPI CPU Interrupt
INT7	dMAX FIFO status
INT8	dMAX Transfer Complete
INT9	dMAX event 0x2
INT10	dMAX event 0x3
INT11	dMAX event 0x4
INT12	dMAX event 0x5
INT13	dMAX event 0x6
INT14	I2C0, I2C1, SPI0, SPI1
INT15	dMAX event 0x7

Table 3: C67X+ CPU Interrupt assignments[18]

In C67X+ CPU interrupts are divided into three types.

- INT0 is the RESET interrupt. This is the highest priority interrupt and can not be maskable.
- INT1 is the NMI interrupt. This is the second highest priority and cannot maskable. This interrupt is used to notify the CPU about serious hardware faults.
- INT2 and INT3 are reserved and cannot be used.
- INT4 to INT15 are maskable interrupts. These interrupt are for notifying about external events to CPU.

Interrupt Configuration: In C67X+ CPU there are several control registers responsible for configuring interrupts.

- GIE bit in CSR register is for globally enabling and disabling the interrupts. Setting GIE bit enable interrupts across the system.
- IER register is for enabling specific interrupt line for the CPU. Setting bit in this register will enable the interrupt.
- ICR register is to clear the interrupt flag in IFR register
- IFR register is for showing interrupt status information
- ISR is programatically trigger interrupt.
- ISTP for storing the ISR vector address.

Enabling and processing interrupts includes several steps and proper handling mechanisms were needed. When an interrupt occurs the CPU will switch to the ISR routine to execute. CPU will not do any context saving and restoration mechanisms to go back to old execution properly. TI CGT compiler abstract all the context saving and restoration functions and provide users with “interrupt” keyword to write ISR routines in C language. The interrupt keyword will do the following operations

1. Function return address will be IRP register value
2. Save all registers to stack,
3. Call ISR
4. Restore all registers from stack,
5. Copy IRP value to B3 register.
6. Return form ISR by branching to B3 register.

As the project is using GCC toolchain, code needs to be implemented to do all the above necessary steps done by interrupt keyword to preserve the current context when an interrupt occurs. To achieve this mechanism context saving and restoration functionality was implemented in assembly and created weak functions to serve as ISR routines for each interrupt. The Assembly helper function will do the following operations.

1. Create new stack frame to save all the registers
2. Save all registers to stack
3. save the Stack Pointer
4. create new stack frame for running ISR code.
5. Call ISR week function
6. Restore stack
7. restore Stack pointer
8. restore all registers.
9. Copy IRP value to B3
10. return to original context by branching to B3.

The week functions save the program execution from abnormal behaviours when interrupt occurs without specifying ISR handler.

Interrupt Service Fetch Packet(ISFP):

ISFP is a table of interrupt vectors. Each vector is capable of holding maximum 8 instructions. When CPU gets an interrupt then the CPU reads the address in the ISTP register and calculate the offset by multiplying the interrupt number with 32 and branch to the resulting address and start the execution from the address. ISP is the first ISR routine which will execute when CPU gets interrupt. In general the ISR routines are much bigger than 8 instructions and do not fit into this ISFP. To overcome this problem multilevel ISR mechanism is implemented.

The multilevel ISR mechanism will do the following.

1. Each ISFP packet holds the address of generic function to do common tasks such as context save and restore and address of the C ISR routine to call the actual ISR function.
2. Allocate stack for saving current context.
3. Save context of two registers(Ex: A4 and A5) into stack.
4. The generic helper function and C ISR routine addresses are stored into two registers.
5. Call Generic helper function.
6. This function will do the necessary work to save the context and call the actual ISR.
7. Restore the context after ISR.

The project contains two assembly files called **vectors.s** and **isr_helpers.s** (See **Appendix C**) to form ISFP and other functionality mentioned above to branch to the actual ISR routine.

3.6 A Tiny C library

As the compiler is without any standard C library support, It's hard to develop application without having standard library functions such as string manipulation functions, console print functions. To address this problem this project implemented a tiny C library with minimum set of standard C library helper functions. **Appendix B** shows the list of library functions implemented as part of this tiny C library.

This library is a static library. So when the program link this library only necessary components will add to the final executable.

For formatted printing over UART for debugging purpose, a wrapper function is created on top of **sprintf** function and added UART write function as hook for writing to output.

3.7 Debugging facilities

One of the hardest problem faced while developing firmware for OBC using the GCC toolchain is lack of Debugging facilities. Particularly while writing CPU startup code and bringing interrupt support. To overcome this problem, software workarounds were implemented to provide debugging facilities. The following debugging helpers are available at the time writing this thesis.

UART based print functions: All standard library style print function are implemented. All functions write the data to UART which can be seen on serial terminal.

Dump CPU Registers: Implemented helper functions to dump CPU registers over UART onto the serial terminal. This can be done by making a local dump of CPU Registers and dump them as strings over UART.

Dump call-trace: This functionality is implemented by using libgcc library's unwind functions. This feature will work when there is no interrupts. This helper function cannot trace interrupt call stack.

Dump Memory: This helper function is also similar to Dumping CPU registers. This function is useful to the state of peripheral registers by dumping the specific regions of memory.

These helper function helps to solve the problems which were needed to debug manually. But these functions are not comparable to hardware assisted debugging method(JTAG).

4 Project User Guide

This chapter describe the setup guide for the compiler environment which includes build environment specifications, building steps for the compiler and project creation guide.

4.1 System Requirements

Current implementation only supports Linux operating system. The following are the specifications used for building the GCC cross toolchain.

Hardware: X86 or X86-64 machine with 1 Gigabytes of RAM.

Operating System: Ubuntu Linux 16.04.4 (Tested on both 32-bit and 64-bit variants)

GCC Version: gcc-7.1.0

The Actual hardware used to build the compiler consists Intel Core I7[49] quad core processor with 32 Gigabytes of RAM. There is no need for this huge amount of resources. Building is also possible with minimum hardware requirements but faster and powerful hardware will results in quicker builds.

4.2 Building GCC toolchain

Install required components: GCC build system need the following components. Install them from application repository.

```
sudo apt-get install build-essential make gawk
```

Getting the sources: The sources for the GCC toolchain is located in gitlab repository. Clone the repository.

```
git clone https://gitlab.com/openc6x/gcc-c6x.git
```

After downloading the sources. Change into the sources directory.

Building the Toolchain: The current implementation provides a build script to automate all the build steps. Goto the sources directory and execute **build.sh** to build the compiler.

```
# To build the toolchain for the first time run the script without
any arguments.
./build.sh

#To clean the existing build run the following command
./build.sh clean

#run the following command to see the help
./build.sh help
```

This step will run for longer duration depends on the hardware using for the build. After finishing the build script without errors, the source directory will have a directory called **deploy.** This directory contains final compiler binaries.

The generated binaries will run on the host machine and produce code for the target CPU(TI C67X+ in this context). Update the "**PATH**" environment variable with the path of the deploy/bin directory. Execute the following command to update the PATH variable.

```
Export PATH=$PATH:<path to deploy directory>/bin
```

Replace the appropriate path in "path to deploy directory". Now the compiler binaries are in the system path.

4.3 Makefile Project

To generate firmware binary from the sources need compilation of multiple source files, linking external libraries and usage of linker scripts. Doing these steps manually is tedious task. This system uses GNU Makefile based build system to automate all build steps.

To generate firmware for OBC board, the following steps needs to be done.

- Compile startup code
  - This part consists of startup assembly and c code to do copy to RAM and BSS initialization and to start main function.
- Compile interrupt code
  - This part consists of assembly code to handle context save and restore, branch calls to interrupt service routines in C.
- Compile application code.
  - The entry point of the application code starts from main function.

After this A linker script is needed to specify and configure the binary generation from the compiled objects and also to link necessary libraries.

- Using GNU linker link all the generated object files to generate ELF binary file.
- The input format accepted by OBC bootloader is Intel Hex format[50]. Converting the ELF file into Intel Hex format is done through two steps. First convert ELF binary to RAW Binary and then convert RAW Binary into Intel Hex format. This can be done by objcopy utility from binutils package.

Appendix E shows a very basic Makefile for generating firmware in hex format.

4.4 Programming binaries on OBC board

The current OBC board has UART based bootloader to load the firmware into the board. For this the host machine needs a serial terminal program such as cutecom[50]. The following steps explains the procedure to program on to the board.

- Connect the OBC board to the Linux host through USB-to-UART cable provided with the board.
- Open cutecom application.

- Select the serial device from the "**Device**" dropdown menu.
- Select "**Settings**".
- Change the baudrate to 115200 8N1(8 bit data, No Carry and 1 stop bit).
- Select the flow control to software flow control.
- Click "**Open**" button to open the serial device.
- Press the reset button on OBC board. Now the board starts from bootloader. The board asks for the user to press any key to enter into boot menu.
- Press any key to enter into boot menu.
- The boot menu has only two commands.
    - "**P**" for program new firmware.
    - "**R**" for run the firmware from flash.
- To program the firmware press "**P**". Now the bootloader waits for the hex file through UART.
- From the cutecom click "**sendfile**" button. It will open a dialog to select the hex file. Navigate and select the hex file to flash.
- After programming the hex file, the bootloader asks to run the newly programmed firmware by pressing "**R**" through the UART.
- Press "**R**" to run the firmware or press reset button on the OBC board.

By default the bootloader copy the code in Flash memory and starts executing the code from flash. It's developer work to copy the code from flash to RAM and start execution from RAM.

5 Tests and Analysis

5.1 Test Methods

Testing the project is divided into two parts. The first part perform tests to verify the code generation and correctness of the program generated by the compiler. In the second part tests are performed to verify the each software module with compiled with GCC compiler.

There are difficulties for testing the compiler. This compiler generate code for different target which will not run on the host. So the testing of the compiler is done in two parts. First part perform tests on the host to test compiler errors. Then compile test cases as target binary to perform the tests on the target and get the test logs through UART. This is tedious part as the tester need to program the device manually several times for the test cases. The details are described below.

A test runner script is created to perform compilations on several programs with possible compiler options. The main goal of these tests are to find the compiler bugs not to find the program bugs.

Individual software modules are tested by running the test cases on the OBC board. For testing interrupts, test cases written to programatically trigger the interrupts with test functions in ISR routines. These ISR routines write on UART for logging purpose. The C library functions and debugging helper functions are tested using assert macros. Test functions were written to test all the available function in C library. These test cases run on the board and logs are gathered through UART output.

5.2 Structure of Generated Assembly Code

In TI C67x+ architecture there is no dedicated stack manipulation instructions or a dedicated Stack Pointer register. The following content explain the behaviour of the GCC generated code.

- Register B15 is used as Stack pointer,
- Register A15 is used as Frame Pointer
- Register A14 is used as Data pointer.
- Register B3 is used as Link Register to store the return address.
- Each call to function create new stack frame.
- First ten function arguments are stored in General purpose registers.
- For functions with more than ten function arguments then rest of the arguments are stored on to the stack.
- Function return value is stored in register A0.

The following code snippet in Table 4 shows the generated assembly code sample.

```
smp:     file format elf32-tic6x-elf-le


Disassembly of section .text:

00008080 <sum>:
    8080:   07bc54f4    stw .D2T1 a15,*b15--(8)
    8084:   07bd1058    add .L1X 8,b15,a15
    8088:   07bd09c2    sub .D2 b15,8,b15
    808c:   023c2074    stw .D1T1 a4,*-a15(4)
    8090:   023c4076    stw .D1T2 b4,*-a15(8)
    8094:   00bc2064    ldw .D1T1 *-a15(4),a1
    8098:   00006000    nop 4
    809c:   003c4064    ldw .D1T1 *-a15(8),a0
    80a0:   00006000    nop 4
    80a4:   00040840    add .D1 a1,a0,a0
    80a8:   020008f0    or .D1 0,a0,a4
    80ac:   07bf105a    add .L2X -8,a15,b15
    80b0:   07bc52e4    ldw .D2T1 *++b15(8),a15
    80b4:   00006000    nop 4
    80b8:   000c0362    b .S2 b3
    80bc:   00008000    nop 5

000080c0 <foo>:
    80c0:   07bc54f4    stw .D2T1 a15,*b15--(8)
    80c4:   07bd1058    add .L1X 8,b15,a15
    80c8:   07bd09c2    sub .D2 b15,8,b15
    80cc:   01bc62f6    stw .D2T2 b3,*+b15(12)
    80d0:   02001e2a    mvk .S2 60,b4
    80d4:   02000f28    mvk .S1 30,a4
    80d8:   1ffff812    callp .S2 8080 <sum>,b3
    80dc:   001008f0    or .D1 0,a4,a0
    80e0:   020008f0    or .D1 0,a0,a4
    80e4:   01bc62e6    ldw .D2T2 *+b15(12),b3
    80e8:   00006000    nop 4
    80ec:   07bf105a    add .L2X -8,a15,b15
    80f0:   07bc52e4    ldw .D2T1 *++b15(8),a15
    80f4:   00006000    nop 4
    80f8:   000c0362    b .S2 b3
    80fc:   00008000    nop 5
```

Table 4: GCC Disassembly for TI C67X+

As seen from the Table 4, there are two functions foo and sum. The "s**um**" function will take two integer arguments and return the sum of the arguments. The "**foo**" function calls the sum function and return the result given by sum function.

The highlighted lines in each functions, the code in yellow coloured lines is function prologue and the code in red coloured lines in both functions is function epilogue. Whenever a function call occurs the following steps will occur.

1. Store the current frame pointer into stack
2. Allocate new stack frame
3. Assign new stack frame address to the Frame Pointer.
4. Store B3 on stack if the function is going to use B3 register.
5. Finally the function starts executing function code.

After execution of the function code the function will restore B3 register. The stack frame created in starting of the function is restored and old frame pointer is restored from the stack. Finally branch to B3 register to return the execution to the caller.

5.3 Test Results

Table 5 shows summary of tests performed on system. GCC build system contains large test suite to test language frontend, backend and RTL generations. Tests performed in this work are for testing compiler stability and correctness of the generated program.

Test case	Result
Testing C language constructs(expressions, branching and looping, variables and scope, pointers and address operations, data structures and data handling)	Number of test cases: 164 Number of performed iterations on final implementation: 90 Number of failures: 0
Testing standard C library functions	Number of test cases: 60 Number of performed iterations on final implementation: 90 Number of failures: 0
Testing debug helper functions	Number of test cases: 30 Number of performed iterations on final implementation: 90 Number of failures: 430

Table 5: Summary of tests and results

The test cases covered C language constructs and standard C library functions. More tests needed for board support code and debug functions. Failures in debug helpers needs to be fixed. The following points provides summary of the results.

- The compiler is running without intermediate crashes.
- GCC optimizations results in improper assembly code generation. The binaries generated with GCC optimization options enabled will not run on the target.
- Interrupts are working without bugs. Interrupt nesting is not working.
- C Library and functions are passing tests without failures.
- Debug helper functions are unstable. Particularly dumping call stack will results in board reset.

The environment available with this work contains a usable compiler with working board startup code and a C standard library implementation. The compiler is usable without optimizations and the generated binaries with startup code is working well on the target board. Developers can use debug helper functions except the functions for dumping callstack. The two limitations that the developers need to take care is to avoid interrupt nesting while developing the application and also avoid GCC optimization options while compiling the source code.

Test logs are presented in **Appendix E.**

6 Conclusion and Future Work

The goal of this thesis is to create an alternative environment alternative to TI Code Composer Studio to develop software for TI C6X series of processors. The entire work is divided into two stages. In first stage build a usable baremetal GCC compiler targetting TI C67X+ CPU. In the later stage work is done on making the compiler usable on OBC board by creating minimum set of software components which includes startup code, support for basic peripherals such as PLL, Timer and GPIO, support for Interrupt subsystem and a tiny C library for standard helper functions such as memory operations string operations and more. This project uses makefile based build system to build and test the software on the target board.

6.1 Future Work and Improvements

Even though the bare minimum environment is available to develop software for the OBC board, there are many components need to be develop to use it as an alternative solution to TI CCS. The current compiler does not have any optimizations the TI CGT compiler has. DSP and Math libraries which are available for TI CGT toolchain are not available for the GCC toolchain. Implementing DSP algorithms without CPU optimised DSP libraries will results in bad performance. Improvements needs to be done for GCC compiler to overcome existing limitations to generate optimised code.

TI C6X Intrinsics: TI provide a set of helper functions called intrinsics to do single instruction operations and SIMD operation from C language without writing inline assembly[45]. This part is completely missing in GCC toolchain. GCC has wrappers to mimic these intrinsics. Internally they are normal function calls and the cost of these operations are high when compare with original TI CGT specific intrinsics.

DSP and Math libraries: There is no DSP and math library support implemented in GCC toolchain. The developers need to develop complex DSP algorithms need to implement their own DSP and Math functions.

Code optimizations: The current GCC support for TI C6X+ only using registers A0 to A15 and B0 to B15 where as this CPU has 64 registers divided into two register files. The reason for this is that GCC is using generic C67X CPU ISA. This leads to more instructions. The other problem of GCC is when using optimization options. GCC optimizations generate link time errors. Typical example for this is when compile program with -O3 optimization option the LMA sections in the objets files are overlaps with other object files even though the sections are properly described in linker script.

CPU Register access: Unlike TI CGT compiler, GCC compiler does not have any special statements to access CPU registers directly from C code. GNU inline assembly is needed to access CPU registers.

Assembly optimizer: One major feature missing from TI CGT compiler is assembly optimizer. This feature allows to write linear assembly program without concerning about functional units and parallel instructions and loop optimizations. The assembler will take care of the all optimizations. No assembler level optimizations provided by GNU assembler.

There are more software needs to be implemented for the other peripherals such as SDRAM interfacing, dMAX controller, NAND Flash support, SD-Card interfacing and more. The current debugging scheme available as part of this thesis is only limited to printing over UART. JTAG hardware based debugging is highly needed to make the development more convenient particularly while developing low level software components. The current bootloader is loading the hex file to flash the code on to the OBC board. This can be improved by replacing the existing bootloader mechanism with binary file loading mechanism. This saves lot of time to load the code on to the OBC board. There is also a strong need for RTOS support. The existing environment is enough to port the RTOS(Ex: FreeRTOS).

The implemented compiler is not a definite alternative to TI CGT with above mentioned limitations. The maturity of GCC for TI C6X is still in early stage and TI and the community members are still developing the compiler. The work done in this thesis sets a base for the future works on GCC based environment for TI C67X+ with all the features needed to develop. This thesis is conclude as the requirements set in the proposed solution are achieved.

7 References

[1] TTÜ100 Satellite Project (https://www.ttu.ee/projects/mektory-eng/satellite-programme-3/satellite-programme/) (24.05.2019)

[2] Mektory Satellite Programme (https://www.ttu.ee/projects/mektory-eng/satellite-programme-3/satellite-programme/ttu-mektory-satellite-programme/) (24.05.2019)

[3] TTÜ100 Satellite Project Communication subsystem (https://www.ttu.ee/projects/mektory-eng/satellite-programme-3/satellite-programme/wp2-c-uhf-transceiver-ku-transmitter/) (24.05.2019)

[4] TTÜ100 Satellite Project On-board camera structures (https://www.ttu.ee/projects/mektory-eng/satellite-programme-3/satellite-programme/wp1-b/) (24.05.2019)

[5] TTÜ100 Satellite Project Electric power supply (https://www.ttu.ee/projects/mektory-eng/satellite-programme-3/satellite-programme/wp2-e/) (24.05.2019)

[6] TI TMS320C6727B Floating-Point Digital Signal Processor (https://www.ti.com/product/TMS320C6727B/description) (24.05.2019)

[7] Code Composer Studio (CCS) Integrated Development Environment (IDE) (http://www.ti.com/tool/CCSTUDIO) (24.05.2019)

[8] XDS560v2 System Trace USB Debug Probe (https://www.ti.com/tool/tmdsemu560v2stm-u) (01.11.2018)

[9] Code Composer Studio Version 3 Downloads (http://processors.wiki.ti.com/index.php/Download_CCS#Code_Composer_Studio_Version_3_Downloads) (09.11.2018)

[10] C6000 code generation tools – compiler (http://www.ti.com/tool/c6000-cgt) (24.04.2019)

[11] Mentor Graphics Sourcery Tools Services (https://www.mentor.com/embedded-software/sourcery-tools-services/) (24.04.2019)

[12] TI C6X port patch work (https://gcc.gnu.org/ml/gcc-patches/2011-05/msg00746.html) (30.05.2018)

[13] Sourcery CodeBench Lite 4.5-130 for C6000 uClinux (https://sourcery.mentor.com/GNUToolchain/release2065) (24.04.2019)

[14] The Linux Kernel Archives (https://www.kernel.org/) (24.04.2019)

[15] C6000 Linux Support (http://processors.wiki.ti.com/index.php/C6000_Linux_Support) (24.04.2019)

[16] Linux-c6x project page (http://www.linux-c6x.org/wiki/index.php/Main_Page) (24.04.2019)

[17] Microsoft Windows XP wikipedia page (https://en.wikipedia.org/wiki/Windows_XP) (24.04.2019)

[18] TI TMS320C6727B Floating-Point Digital Signal Processors datasheet (https://www.ti.com/lit/ds/symlink/tms320c6727b.pdf) (24.04.2019)

[19] TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide (http://www.ti.com/lit/ug/spru733a/spru733a.pdf) (24.04.2019)

[20] TMS320C6000 Programmer's Guide (http://www.ti.com/lit/ug/spru198k/spru198k.pdf) (24.04.2019)

[21] Porting GCC to the TMS320-C6000 DSP Architecture (https://www.researchgate.net/publication/228962339_Porting_GCC_to_the_TMS320-C6000_DSP_Architecture) (24.04.2019)

[22] GNU Compiler Collection wikipedia page (https://en.wikipedia.org/wiki/GNU_Compiler_Collection#Design) (24.04.2019)

[23] GNU C Compiler Internals/GNU C Compiler Architecture (https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture) (24.04.2019)

[24] GNU Compiler Collection (GCC) Internals (https://gcc.gnu.org/onlinedocs/gccint/) (24.04.2019)

[25] GNU compiler frontend format GENERIC (https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html#GENERIC) (24.04.2019)

[26] GENERIC and GIMPLE: A New Tree Representation for Entire Functions (https://ols.fedoraproject.org/GCC/Reprints-2003/jason.pdf) (24.04.2019)

[27] GCC GIMPLE tuple representation (https://gcc.gnu.org/onlinedocs/gccint/Tuple-representation.html#Tuple-representation) (24.04.2019)

[28] Static Single Assignment details in GCC compiler (https://gcc.gnu.org/onlinedocs/gccint/SSA.html#SSA) (24.04.2019)

[29] RTL Representation in GCC (https://gcc.gnu.org/onlinedocs/gccint/RTL.html#RTL) (24.04.2019)

[30] (https://en.wikipedia.org/wiki/Lisp_(programming_language)) (24.04.2019)

[31] Lisp (programming language) wikipedia page (https://lisp-lang.org/learn/lists) (24.04.2019)

[32] GNU Binutils (https://www.gnu.org/software/binutils/) (24.04.2019)

[33] Executable and Linkable Format (https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) (24.04.2019)

[34] Common Object File Format (https://en.wikipedia.org/wiki/COFF) (24.04.2019)

[35] CLooG Code Generator (https://www.cloog.org/) (24.04.2019)

[36] GCC Optimization options (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html) (24.04.2019)

[37] GCC GNU Debugger (https://www.gnu.org/software/gdb/) (24.04.2019)

[38] Multi precision arithmetic operations library (https://gmplib.org/) (24.04.2019)

[39] Integer sets and relations operations library (http://isl.gforge.inria.fr/) (24.04.2019)

[40] ELF file format parsing library (https://directory.fsf.org/wiki/Libelf) (24.04.2019)

[41] Multi precision complex number library (http://www.multiprecision.org/mpc/) (24.04.2019)

[42] Multi precision floating point arithmetic functions (https://www.mpfr.org/) (24.04.2019)

[43] Autoconf build configurations (https://www.gnu.org/software/autoconf/) (24.04.2019)

[44] GNU Make build system (https://www.gnu.org/software/make/) (24.04.2019)

[45] C6000 Intrinsics and SIMD Operations (http://processors.wiki.ti.com/index.php/C6000_Intrinsics_and_SIMD_Operations) (01.09.2018)

[46] TMS320C672x DSP Software-Programmable Phase-Locked Loop (PLL) Controller(https://www.ti.com/lit/ug/spru879a/spru879a.pdf) (24.04.2019)

[47] Small C Library for embedded systems (https://uclibc-ng.org/) (24.04.2019)

[48] (Linux for microcontrollers) (https://sourceforge.net/projects/uclinux/) (24.04.2019)

[49] (https://www.intel.com/content/www/us/en/products/processors/core/i7-processors.html) (08.05.2019)

[50] Cutecom graphical serial terminal (http://cutecom.sourceforge.net/) (08.05.2019)

[51] Toolchains elinux.org (https://elinux.org/Toolchains) (10.05.2019)

[52] Anatomy of a Language Front End (https://gcc.gnu.org/onlinedocs/gcc-4.2.4/gccint/Front-End.html) (10.05.2019)

Appendices

A GCC Toolchain Build Procedure

The entire toolchain including all the sources are available at gitlab account.

The Curren build works only on linux(The build machine is ubuntu 16.04 LTS).

Dependencies for building:

Install the following tools on ubuntu 16.04.

```
sudo apt-get install build-essential make gawk
```

I created a build script called build.sh which I included in the repository.

Build steps:

Get the source and goto the source directory and run the build.sh. If everything compiled properly then you have GCC binaries in deploy directory in the current directory.

Build script:

```
#!/bin/bash

export CROSSHOME=$PWD
export PREFIX=$CROSSHOME/deploy
export PATH=$PATH:$CROSSHOME/deploy/bin

clean_build()
{
    cd $CROSSHOME
    rm -rf $CROSSHOME/build &> /dev/null
    rm -rf $CROSSHOME/deploy &> /dev/null
}

prepare_build()
```

```
{
    if ! [ -e $CROSSHOME/build ]
    then
        mkdir $CROSSHOME/build
        mkdir $CROSSHOME/build/c6x-src
        mkdir $CROSSHOME/build/c6x-gcc-build
        mkdir $CROSSHOME/build/c6x-gdb-build
    fi
    if ! [ -e $CROSSHOME/deploy ]
    then
        mkdir $CROSSHOME/deploy
    fi
    cd $CROSSHOME/build/c6x-src
    ln -s ../../gcc-7.1.0/* .
    ln -s ../../binutils-2.28/* .
    ln -s ../../mpfr-3.1.5 mpfr
    ln -s ../../gmp-6.1.2 gmp
    ln -s ../../mpc-1.0.3 mpc
    ln -s ../../isl-0.18 isl
    ln -s ../../cloog-0.18.1 cloog
    ln -s ../../libelf-0.8.13 libelf
    cd $CROSSHOME
}

build_gcc()
{
    cd $CROSSHOME/build/c6x-gcc-build
    echo -e "Configure gcc..."
    ../c6x-src/configure \
        --host=x86_64-pc-linux-gnu \
        --build=x86_64-pc-linux-gnu \
        --target=c6x-elf \
        --prefix=$PREFIX \
        --disable-nls \
        --disable-werror \
        --enable-__cxa_atexit \
        --enable-long-long \
        --enable-libstdcxx-time \
        --enable-languages=c \
        --enable-lto \
        --enable-gold \
        --disable-multilib \
        --enable-cheaders=c_std \
        --disable-libgomp \
        --without-system-libunwind \
        --disable-threads \
        --disable-libmudflap \
        --disable-libssp \
        --disable-libstdcxx-pch \
        --with-gnu-as \
        --with-gnu-ld \
        --disable-shared \
        --with-host-libstdcxx='-static-libgcc -Wl,-Bstatic,-lstdc++,-
Bdynamic -lm' \
        --without-headers \
        --with-newlib
    if [ $? != 0 ]
    then
        echo -e "build_gcc configure failed...!"
        return 1
    fi
    echo -e "Building gcc pass 1(all-gcc)..."
    make -j$(nproc) all-gcc
    if [ $? != 0 ]
    then
        echo -e "build_gcc pass 1 failed...!"
        return 1
    fi
    echo -e "Installing gcc pass 1(all-gcc)..."
    make -j$(nproc) install-gcc
    if [ $? != 0 ]
    then
        echo -e "Installing gcc pass 1 failed...!"
        return 1
    fi
    echo -e "Building gcc pass 2(all-target-libgcc)..."
    make -j$(nproc) all-target-libgcc
    if [ $? != 0 ]
    then
        echo -e "build_gcc pass 2 failed...!"
        return 1
    fi
    echo -e "Installing gcc pass 2(all-target-libgcc)..."
    make -j$(nproc) install-target-libgcc
    if [ $? != 0 ]
    then
        echo -e "Installing gcc pass 2 failed...!"
        return 1
    fi
    echo -e "Building gcc pass 3 final (all)..."
    make -j$(nproc) all
    if [ $? != 0 ]
    then
        echo -e "build_gcc pass 3 failed...!"
        return 1
    fi
    echo -e "Installing gcc pass 3 final (all)..."
    make -j$(nproc) install
    if [ $? != 0 ]
    then
        echo -e "Installing gcc pass 3 final failed...!"
        return 1
    fi
    echo "Building gcc success..."
    cd $CROSSHOME

}

build_gdb()
{
    export PATH=$PATH:$CROSSHOME/deploy/bin
    cd $CROSSHOME/build/c6x-gdb-build
    ../../gdb-7.10.1/configure --target=c6x-elf \
                     --prefix=$PREFIX
    if [ $? != 0 ]
    then
        echo -e "build_gdb configure failed...!"
        return 1
    fi
    make -j$(nproc)
    if [ $? != 0 ]
    then
        echo -e "build_gdb make failed...!"
        return 1
    fi
    make -j$(nproc) install
    if [ $? != 0 ]
    then
        echo -e "build_gdb install failed...!"
        return 1
    fi
    echo -e "Build gdb success..."
}

start_build()
{
    build_gcc
    if [ $? != 0 ]
    then
        echo -e "build_gcc failed...!"
        return 1
    fi
    build_gdb
    if [ $? != 0 ]
    then
        echo -e "build_gdb failed...!"
        return 1
    fi
}

fresh_build()
{
    echo -e "Cleaning build..."
    clean_build
    echo -e "Preparing build..."
    prepare_build
    if [ $? != 0 ]
    then
        echo -e "Prepare build failed...!"
    fi
    echo -e "Running Final build..."
    start_build
    if [ $? != 0 ]
    then
        echo -e "Final build failed...!"
    else
        echo -e "Build success..."
    fi
}

update_build()
{
    start_build
    if [ $? != 0 ]
    then
        echo -e "Final build failed...!"
    else
```

```
        echo -e "Build success..."
    fi
}

usage()
{
    echo -e "Usage:"
    echo -e "build.sh <[fresh]|[clean]|[prepare]|[start]|[gcc]|
[gdb]>"
}

if [ $# -lt 1 ]
then
    fresh_build
elif [ $1 = "help" ]
then
    usage
elif [ $1 = "fresh" ]
then
    fresh_build
elif [ $1 = "clean" ]
then
    clean_build
elif [ $1 = "prepare" ]
then
    prepare_build
elif [ $1 = "start" ]
then
    update_build
elif [ $1 = "gcc" ]
then
    build_gcc
elif [ $1 = "gdb" ]
then
    build_gdb
fi
```

B List of C Library Helper functions

String processing functions	strcpy, strstr, strcat, strcmp, strlen, strtok
Memory operations	memcpy, memset, memcmp
Formtted print functions	sprintf, sscanf, printf
Console print functions	puts, putc, printf
Miscellaneous functions	atoi, atol

C Linker Script

```
/*
ENTRY(_start)
SECTIONS
{
 . = 0x91000000;
 .text : { *(.text) }
 .data : { *(.data) }
 .bss : { *(.bss COMMON) }
 .bss : { *(.const) }
}
*/

ENTRY(_start)

MEMORY
{
    FLASH (rwx) : ORIGIN = 0x091000000, LENGTH = 0x10000  /*64K*/
    RAM   (rwx) : ORIGIN = 0x10000400,  LENGTH = 0x20000  /*128K*/
}

stack_size = 0x8000; /* 32K */
_stack_start = ORIGIN(RAM)+LENGTH(RAM);
_stack_end = _stack_start - stack_size;

/* No heap usage for now. Need to implement memory management.
/* heap_size = 256; */

SECTIONS
{
    .startup ORIGIN(FLASH) :
    {
        . = ALIGN(4);
        *(.startup)
    }
    . = ALIGN(4);
    _scode = .;
    .text ORIGIN(RAM) : AT(LOADADDR(.startup) + SIZEOF(.startup))
    {
    _stext = .;
        *(.vectors)
        *(.text)
        *(.text*)
    _etext = .;
    }
    .const ORIGIN(RAM) + SIZEOF(.text) : \
    AT (LOADADDR(.text) + SIZEOF(.text))
    {
    _sconst = .;
        *(.const)
    _econst = .;
    }
    .fardata ORIGIN(RAM) + SIZEOF(.text) + SIZEOF(.const) : \
    AT (LOADADDR(.const) + SIZEOF(.const))
    {
```

```
    _sdata = .;
        *(.data)
        *(.far)
        *(.fardata)
    _edata = .;
    }
    _ecode = .;
    .bss ORIGIN(RAM) + SIZEOF(.text) \
    + SIZEOF(.const) + SIZEOF(.fardata) : \
    AT (LOADADDR(.fardata) + SIZEOF(.fardata))
    {
    _sbss = .;
        *(.bss)
        *(.bss*)
        *(COMMON)
    _ebss = .;
    }
}
```

D Interrupt Service Routine in Assembly

isr_helpers.S:

```
.section   .text
.global _saveContextEnterISR
_saveContextEnterISR:
    ADDK  .S2     -256,B15
    STW   .D2T1  A15,*+B15[0]
    ADD       .S1X       0,B15,A15
      STW     .D2T2  B14,*+B15[1]
||    STW     .D1T1  A14,*+A15[2]
      STW     .D2T2  B13,*+B15[3]
||    STW     .D1T1  A13,*+A15[4]
      STW     .D2T2  B12,*+B15[5]
||    STW     .D1T1  A12,*+A15[6]
      STW     .D2T2  B11,*+B15[7]
||    STW     .D1T1  A11,*+A15[8]
      STW     .D2T2  B10,*+B15[9]
||    STW     .D1T1  A10,*+A15[10]
      STW     .D2T2  B9,*+B15[11]
||    STW     .D1T1  A9,*+A15[12]
      STW     .D2T2  B8,*+B15[13]
||    STW     .D1T1  A8,*+A15[14]
      STW     .D2T2  B7,*+B15[15]
||    STW     .D1T1  A7,*+A15[16]
      STW     .D2T2  B6,*+B15[17]
||    STW     .D1T1  A6,*+A15[18]
      STW     .D2T2  B5,*+B15[19]
||    STW     .D1T1  A5,*+A15[20]
      STW     .D2T2  B4,*+B15[21]
||    STW     .D1T1  A4,*+A15[22]
```

```
      STW     .D2T2  B3,*+B15[23]
||    STW     .D1T1  A3,*+A15[24]
      STW     .D2T2  B2,*+B15[25]
||    STW     .D1T1  A2,*+A15[26]
      STW     .D2T2  B1,*+B15[27]
||    STW     .D1T1  A1,*+A15[28]
      STW     .D2T2  B0,*+B15[29]
||    STW     .D1T1  A0,*+A15[30]
    ADDK  .S2     -48,B15
    MVKL .S2 _restoreContextFromISR,B3
    MVKH .S2 _restoreContextFromISR,B3
    B .S2X A1
    NOP 5
.global _restoreContextFromISR
_restoreContextFromISR:
    ADD      .S1X      0,B15,A0
    ADDK  .S1     48,A0
||  ADDK  .S2    48,B15
    LDW   .D1T1  *+A0[0],A15
    LDW   .D2T2  *+B15[1],B14
||  LDW   .D1T1  *+A0[2],A14
    LDW   .D2T2  *+B15[3],B13
||  LDW   .D1T1  *+A0[4],A13
    LDW   .D2T2  *+B15[5],B12
||  LDW   .D1T1  *+A0[6],A12
    LDW   .D2T2  *+B15[7],B11
||  LDW   .D1T1  *+A0[8],A11
    LDW   .D2T2  *+B15[9],B10
||  LDW   .D1T1  *+A0[10],A10
    LDW   .D2T2  *+B15[11],B9
||  LDW   .D1T1  *+A0[12],A9
    LDW   .D2T2  *+B15[13],B8
||  LDW   .D1T1  *+A0[14],A8
    LDW   .D2T2  *+B15[15],B7
||  LDW   .D1T1  *+A0[16],A7
    LDW   .D2T2  *+B15[17],B6
||  LDW   .D1T1  *+A0[18],A6
    LDW   .D2T2  *+B15[19],B5
||  LDW   .D1T1  *+A0[20],A5
    LDW   .D2T2  *+B15[21],B4
||  LDW   .D1T1  *+A0[22],A4
    LDW   .D2T2  *+B15[23],B3
||  LDW   .D1T1  *+A0[24],A3
    LDW   .D2T2  *+B15[25],B2
||  LDW   .D1T1  *+A0[26],A2
    LDW   .D2T2  *+B15[27],B1
||  LDW   .D1T1  *+A0[28],A1
    LDW   .D2T2  *+B15[29],B0
||  LDW   .D1T1  *+A0[30],A0
    ADDK  .S2     256,B15
    LDW   .D2T1  *++B15, A1
    LDW   .D2T1  *++B15, A0
    B .S2 IRP
    NOP 5
```

vectors.S:

```
.global _start
.global isr_handler_4
.global isr_handler_5
.global isr_handler_6
.global isr_handler_7
.global isr_handler_8
.global isr_handler_9
.global isr_handler_10
.global isr_handler_11
.global isr_handler_12
.global isr_handler_13
.global isr_handler_14
.global isr_handler_15
.global _saveContextEnterISR


; Macro for ISR handler
.macro VEC_ENTRY addr, handler
.align 5
.global \addr
\addr:
    STW  .D2T1 A0, *B15--
    STW  .D2T1 A1, *B15--
    mvkl .S1 _saveContextEnterISR, A0
    mvkh .S1 _saveContextEnterISR, A0
    mvkl .S1 \handler, A1
    mvkh .S1 \handler, A1
    B    .S2X A0
    NOP 5
    .endm

; Interrupt service table (IST).
.section   .vectors
.align 5
.global _reset
_reset:
    mvkl .S2 0x10028400, B15
    mvkh .S2 0x10028400, B15
    and .L2 ~0xF,B15,B15
    mvkl    .S2  _start,B1
    MVKH    .S2 _start,B1
    B       .S2 B1
    NOP        5
    NOP
.align 5
.global _nmi
_nmi:
    B   .S2 NRP
    NOP 5
.align 5
.global _vec_2_rsvd
_vec_2_rsvd:
    B   .S2 IRP
    NOP 5
```

```
.align 5
.global _vec_3_rsvd
_vec_3_rsvd:
    B   .S2 IRP
    NOP 5
VEC_ENTRY  _vec_4, isr_handler_4
VEC_ENTRY  _vec_5, isr_handler_5
VEC_ENTRY  _vec_6, isr_handler_6
VEC_ENTRY  _vec_7, isr_handler_7
VEC_ENTRY  _vec_8, isr_handler_8
VEC_ENTRY  _vec_9, isr_handler_9
VEC_ENTRY  _vec_10, isr_handler_10
VEC_ENTRY  _vec_11, isr_handler_11
VEC_ENTRY  _vec_12, isr_handler_12
VEC_ENTRY  _vec_13, isr_handler_13
VEC_ENTRY  _vec_14, isr_handler_14
VEC_ENTRY  _vec_15, isr_handler_15
```

E Makefile to generate firmware hex binaries

```
CROSS_COMPILE=c6x-elf-
ARCH=c67x+
target=OBC_bootloader
all:
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none  $(target).c -o $(target).o  -I./ -I../bsp_c6727/
inc
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none uart.c -o uart.o -I./ -I../bsp_c6727/inc
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none startup.c -o startup.o -I./ -I../bsp_c6727/inc
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none init_rti.c -o init_rti.o -I./ -I../bsp_c6727/inc
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none isr_entry.c -o isr_entry.o
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none cache.c -o cache.o
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none init_sdram.c -o init_sdram.o
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none init_emif.c -o init_emif.o
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none ../bsp_c6727/src/utils/delay.c -o delay.o -I./ -
I../bsp_c6727/inc
      $(CROSS_COMPILE)gcc -c -Wall -march=$(ARCH) -mlong-calls -mno-
dsbt -msdata=none ../bsp_c6727/src/pll/c6727b_pll.c -o c6727b_pll.o -
I./ -I../bsp_c6727/inc
      $(CROSS_COMPILE)as -march=$(ARCH) boot.s -o boot.o
      $(CROSS_COMPILE)as -march=$(ARCH) vectors.s -o vectors.o
      $(CROSS_COMPILE)as -march=$(ARCH) isr_helpers.s -o isr_helper-
s.o
      $(CROSS_COMPILE)ld -T boot.ld boot.o vectors.o isr_helpers.o
delay.o c6727b_pll.o init_emif.o init_sdram.o $(target).o startup.o
cache.o uart.o init_rti.o isr_entry.o -o $(target) -L/home/gunjack/
work/thesis/Thesis/work/compiler/gcc-c6x/deploy/lib/gcc/c6x-elf/
```

```
7.1.0/ -lgcc
      $(CROSS_COMPILE)objcopy -S $(target)  -O binary  $(target).bin
      $(CROSS_COMPILE)objcopy -I binary $(target).bin -O ihex $(tar-
get).hex
      $(CROSS_COMPILE)size $(target)

clean:
      rm -rf *.o $(target) *.bin *.hex
```

F Test Logs

Compiler Tests:

```
Test Runner: Compiler Stability
Running Tests:
PASS ->  ./testcases/00001.c
PASS ->  ./testcases/00002.c
PASS ->  ./testcases/00003.c
PASS ->  ./testcases/00004.c
PASS ->  ./testcases/00005.c
PASS ->  ./testcases/00006.c
PASS ->  ./testcases/00007.c
PASS ->  ./testcases/00008.c
PASS ->  ./testcases/00009.c
PASS ->  ./testcases/00010.c
#./testcases/00010.c: In function 'main':
#./testcases/00010.c:10:2: warning: label 'foo' defined but not used
[-Wunused-label]
#  foo:
#  ^~~
#./testcases/00010.c:4:2: warning: label 'start' defined but not used
[-Wunused-label]
#  start:
#  ^~~~~
PASS ->  ./testcases/00011.c
PASS ->  ./testcases/00012.c
PASS ->  ./testcases/00013.c
PASS ->  ./testcases/00014.c
PASS ->  ./testcases/00015.c
PASS ->  ./testcases/00016.c
PASS ->  ./testcases/00017.c
PASS ->  ./testcases/00018.c
PASS ->  ./testcases/00019.c
PASS ->  ./testcases/00020.c
PASS ->  ./testcases/00021.c
PASS ->  ./testcases/00022.c
PASS ->  ./testcases/00023.c
PASS ->  ./testcases/00024.c
PASS ->  ./testcases/00025.c
PASS ->  ./testcases/00026.c
PASS ->  ./testcases/00027.c
PASS ->  ./testcases/00028.c
PASS ->  ./testcases/00029.c
```

```
PASS ->  ./testcases/00030.c
PASS ->  ./testcases/00031.c
PASS ->  ./testcases/00032.c
PASS ->  ./testcases/00033.c
PASS ->  ./testcases/00034.c
PASS ->  ./testcases/00035.c
PASS ->  ./testcases/00036.c
PASS ->  ./testcases/00037.c
PASS ->  ./testcases/00038.c
PASS ->  ./testcases/00039.c
FAIL ->  ./testcases/00040.c
#compilation terminated.
PASS ->  ./testcases/00041.c
PASS ->  ./testcases/00042.c
PASS ->  ./testcases/00043.c
PASS ->  ./testcases/00044.c
PASS ->  ./testcases/00045.c
PASS ->  ./testcases/00046.c
PASS ->  ./testcases/00047.c
PASS ->  ./testcases/00048.c
PASS ->  ./testcases/00049.c
PASS ->  ./testcases/00050.c
PASS ->  ./testcases/00051.c
PASS ->  ./testcases/00052.c
PASS ->  ./testcases/00053.c
PASS ->  ./testcases/00054.c
PASS ->  ./testcases/00055.c
PASS ->  ./testcases/00057.c
PASS ->  ./testcases/00058.c
PASS ->  ./testcases/00059.c
PASS ->  ./testcases/00060.c
PASS ->  ./testcases/00061.c
PASS ->  ./testcases/00062.c
PASS ->  ./testcases/00063.c
PASS ->  ./testcases/00064.c
PASS ->  ./testcases/00065.c
PASS ->  ./testcases/00066.c
PASS ->  ./testcases/00067.c
PASS ->  ./testcases/00068.c
PASS ->  ./testcases/00069.c
PASS ->  ./testcases/00070.c
PASS ->  ./testcases/00071.c
PASS ->  ./testcases/00072.c
PASS ->  ./testcases/00073.c
PASS ->  ./testcases/00074.c
PASS ->  ./testcases/00075.c
PASS ->  ./testcases/00076.c
PASS ->  ./testcases/00077.c
#./testcases/00077.c: In function 'foo':
#./testcases/00077.c:28:11: warning: 'sizeof' on array function para-
meter 'x' will return size of 'int *' [-Wsizeof-array-argument]
#  if(sizeof(x) != sizeof(void*))
#           ^
#./testcases/00077.c:2:9: note: declared here
# foo(int x[100])
#         ^
#./testcases/00077.c:33:24: warning: 'sizeof' on array function para-
meter 'x' will return size of 'int *' [-Wsizeof-array-argument]
#  if(sizeof(y) <= sizeof(x))
#                        ^
#./testcases/00077.c:2:9: note: declared here
# foo(int x[100])
#         ^
PASS ->  ./testcases/00078.c
#./testcases/00078.c: In function 'main':
#./testcases/00078.c:11:6: warning: unused variable 'v' [-Wunused-
variable]
#  int v[1000];
#      ^
PASS ->  ./testcases/00079.c
PASS ->  ./testcases/00080.c
PASS ->  ./testcases/00081.c
PASS ->  ./testcases/00082.c
PASS ->  ./testcases/00083.c
PASS ->  ./testcases/00084.c
PASS ->  ./testcases/00085.c
PASS ->  ./testcases/00086.c
PASS ->  ./testcases/00087.c
PASS ->  ./testcases/00088.c
PASS ->  ./testcases/00089.c
PASS ->  ./testcases/00090.c
PASS ->  ./testcases/00091.c
PASS ->  ./testcases/00092.c
PASS ->  ./testcases/00093.c
PASS ->  ./testcases/00094.c
PASS ->  ./testcases/00095.c
PASS ->  ./testcases/00096.c
PASS ->  ./testcases/00097.c
PASS ->  ./testcases/00098.c
PASS ->  ./testcases/00099.c
#./testcases/00099.c:5:1: warning: 'vecresize' defined but not used
[-Wunused-function]
# vecresize(Vec *v, int cap)
# ^~~~~~~~~
PASS ->  ./testcases/00100.c
PASS ->  ./testcases/00101.c
PASS ->  ./testcases/00102.c
PASS ->  ./testcases/00103.c
FAIL ->  ./testcases/00104.c
#In file included from ./testcases/00104.c:1:0:
#/home/gunjack/work/thesis/Thesis/work/compiler/gcc-c6x/deploy/lib/
gcc/c6x-elf/7.1.0/include/stdint.h:9:16: fatal error: stdint.h: No
such file or directory
# # include_next <stdint.h>
#                ^~~~~~~~~~
#compilation terminated.
PASS ->  ./testcases/00105.c
PASS ->  ./testcases/00106.c
#./testcases/00106.c: In function 'main':
#./testcases/00106.c:7:12: warning: variable 's2' set but not used [-
Wunused-but-set-variable]
#  struct S2 s2;
#            ^~
PASS ->  ./testcases/00107.c
PASS ->  ./testcases/00108.c
PASS ->  ./testcases/00109.c
PASS ->  ./testcases/00110.c
PASS ->  ./testcases/00111.c
```

```
PASS ->  ./testcases/00112.c
PASS ->  ./testcases/00113.c
PASS ->  ./testcases/00114.c
PASS ->  ./testcases/00115.c
PASS ->  ./testcases/00116.c
PASS ->  ./testcases/00117.c
PASS ->  ./testcases/00118.c
PASS ->  ./testcases/00119.c
PASS ->  ./testcases/00120.c
PASS ->  ./testcases/00121.c
PASS ->  ./testcases/00122.c
PASS ->  ./testcases/00123.c
PASS ->  ./testcases/00124.c
PASS ->  ./testcases/00126.c
PASS ->  ./testcases/00127.c
PASS ->  ./testcases/00128.c
PASS ->  ./testcases/00129.c
PASS ->  ./testcases/00130.c
PASS ->  ./testcases/00133.c
#./testcases/00133.c: In function 'main':
#./testcases/00133.c:11:16: warning: suggest parentheses around '+'
in operand of '&' [-Wparentheses]
#  i = 32766 + 1 & 3;
#              ^
#./testcases/00133.c:21:16: warning: suggest parentheses around '+'
in operand of '&' [-Wparentheses]
#  u = 32766 + 1 & 3;
#              ^
#./testcases/00133.c:4:11: warning: variable 'u' set but not used [-
Wunused-but-set-variable]
#  unsigned u;
#           ^
#./testcases/00133.c:3:6: warning: variable 'i' set but not used [-
Wunused-but-set-variable]
#  int i;
#      ^
PASS ->  ./testcases/00134.c
#./testcases/00134.c: In function 'main':
#./testcases/00134.c:12:22: warning: suggest parentheses around '-'
in operand of '&' [-Wparentheses]
#  i = (1ll << 32) - 1 & 3;
#                    ^
#./testcases/00134.c:21:22: warning: suggest parentheses around '-'
in operand of '&' [-Wparentheses]
#  u = (1ll << 32) - 1 & 3;
#                    ^
#./testcases/00134.c:5:16: warning: variable 'u' set but not used [-
Wunused-but-set-variable]
#  unsigned long u;
#                ^
#./testcases/00134.c:4:7: warning: variable 'i' set but not used [-
Wunused-but-set-variable]
#  long i;
#       ^
PASS ->  ./testcases/00135.c
#./testcases/00135.c: In function 'main':
#./testcases/00135.c:5:21: warning: variable 'u' set but not used [-
Wunused-but-set-variable]
#  unsigned long long u;
#                      ^
#./testcases/00135.c:4:12: warning: variable 'i' set but not used [-
Wunused-but-set-variable]
#  long long i;
#             ^
PASS ->  ./testcases/00136.c
PASS ->  ./testcases/00137.c
PASS ->  ./testcases/00138.c
PASS ->  ./testcases/00139.c
PASS ->  ./testcases/00140.c
PASS ->  ./testcases/00141.c
#./testcases/00141.c: In function 'main':
#./testcases/00141.c:9:16: warning: variable 'foobar' set but not
used [-Wunused-but-set-variable]
#  int foo, bar, foobar;
#                ^~~~~~
#./testcases/00141.c:11:15: warning: 'foo' is used uninitialized in
this function [-Wuninitialized]
#  CAT(foo,bar) = foo + bar;
#               ^
#./testcases/00141.c:11:15: warning: 'bar' is used uninitialized in
this function [-Wuninitialized]
PASS ->  ./testcases/00142.c
PASS ->  ./testcases/00143.c
PASS ->  ./testcases/00144.c
#./testcases/00144.c: In function 'main':
#./testcases/00144.c:10:4: warning: assignment discards 'const' qual-
ifier from pointer target type [-Wdiscarded-qualifiers]
#  p = i ? 0 : (const void *) 0;
#    ^
PASS ->  ./testcases/00145.c
PASS ->  ./testcases/00146.c
PASS ->  ./testcases/00147.c
PASS ->  ./testcases/00148.c
PASS ->  ./testcases/00149.c
PASS ->  ./testcases/00150.c
PASS ->  ./testcases/00151.c
PASS ->  ./testcases/00152.c
PASS ->  ./testcases/00153.c
PASS ->  ./testcases/00155.c
#./testcases/00155.c: In function 'main':
#./testcases/00155.c:5:2: warning: statement with no effect [-Wun-
used-value]
#  sizeof((int) 1);
#  ^~~~~~
PASS ->  ./testcases/00162.c
PASS ->  ./testcases/00209.c
PASS ->  ./testcases/00210.c
#./testcases/00210.c:14:1: warning: 'stdcall' attribute directive ig-
nored [-Wattributes]
# extern void foo (void) __attribute__((stdcall));
# ^~~~~~
#./testcases/00210.c:16:1: warning: 'stdcall' attribute directive ig-
nored [-Wattributes]
# {
# ^
#./testcases/00210.c: In function 'main':
#./testcases/00210.c:31:5: warning: '__noinline__' attribute does not
apply to types [-Wattributes]
```

```
#     int a = ((ATTR int(*) (void)) function_pointer)();
#     ^~~
#./testcases/00210.c:36:5: warning: '__noinline__' attribute does not
apply to types [-Wattributes]
#     int b = ( (int(ATTR *)(void))  function_pointer)();
#     ^~~
PASS ->  ./testcases/00211.c
PASS ->  ./testcases/00213.c
#./testcases/00213.c: In function 'main':
#./testcases/00213.c:131:10: warning: statement will never be ex-
ecuted [-Wswitch-unreachable]
#       if (0) {
#          ^
#./testcases/00213.c:142:10: warning: statement will never be ex-
ecuted [-Wswitch-unreachable]
#       if (0) {
#          ^
PASS ->  ./testcases/00214.c
#./testcases/00214.c: In function 'bla':
#./testcases/00214.c:40:11: warning: unused variable 'x' [-Wunused-
variable]
#       int x = !!(__ret);
#           ^
#./testcases/00214.c: In function 'chk':
#./testcases/00214.c:49:9: warning: variable 'ret' set but not used
[-Wunused-but-set-variable]
#   _Bool ret;
#         ^~~
#./testcases/00214.c:54:1: warning: control reaches end of non-void
function [-Wreturn-type]
# }
# ^
PASS ->  ./testcases/00215.c
PASS ->  ./testcases/00217.c
PASS ->  ./testcases/00218.c
#./testcases/00218.c: In function 'convert_like_real':
#./testcases/00218.c:40:5: warning: case value '152' not in enumer-
ated type 'enum tree_code' [-Wswitch]
#     case AMBIG_CONV: /* This has bit 7 set, which must not be the
sign
#     ^~~~
#./testcases/00218.c:48:1: warning: control reaches end of non-void
function [-Wreturn-type]
# }
```

OBC Board Tests:

Test logs for one iteration

```
Test Runner: Compiler Tests
Number of Iterations: 1
Iteration 0
Test Target: Expressions
Test Case: 000 -> PASS
Test Case: 001 -> PASS
```

```
Test Case: 002 -> PASS
Test Case: 003 -> PASS
Test Case: 004 -> PASS
Test Case: 005 -> PASS
Test Case: 006 -> PASS
Test Case: 007 -> PASS
Test Case: 008 -> PASS
Test Case: 009 -> PASS
Test Case: 010 -> PASS
Test Case: 011 -> PASS
Test Case: 012 -> PASS
Test Case: 013 -> PASS
Test Case: 014 -> PASS
Test Case: 015 -> PASS
Test Case: 016 -> PASS
Test Case: 017 -> PASS
Test Case: 018 -> PASS
Test Case: 019 -> PASS
Test Case: 020 -> PASS
Test Case: 021 -> PASS
Test Case: 022 -> PASS
Test Case: 023 -> PASS
Test Case: 024 -> PASS
Test Case: 025 -> PASS
Test Case: 026 -> PASS
Test Case: 027 -> PASS
Test Case: 028 -> PASS
Test Case: 029 -> PASS
Test Case: 030 -> PASS
Test Case: 031 -> PASS
Test Case: 032 -> PASS
Test Case: 033 -> PASS
Test Case: 034 -> PASS
Test Case: 035 -> PASS
Test Case: 036 -> PASS
Test Case: 037 -> PASS
Test Case: 038 -> PASS
Test Case: 039 -> PASS
Test Case: 040 -> PASS
Test Case: 041 -> PASS
Test Case: 042 -> PASS
Test Case: 043 -> PASS
Test Case: 044 -> PASS
Test Case: 045 -> PASS
Test Case: 046 -> PASS
Test Case: 047 -> PASS
Test Target: Branching and looping
Test Case: 000 -> PASS
Test Case: 001 -> PASS
Test Case: 002 -> PASS
Test Case: 003 -> PASS
Test Case: 004 -> PASS
Test Case: 005 -> PASS
Test Case: 006 -> PASS
Test Case: 007 -> PASS
Test Case: 008 -> PASS
Test Case: 009 -> PASS
Test Case: 010 -> PASS
Test Case: 011 -> PASS
Test Case: 012 -> PASS
Test Case: 013 -> PASS
Test Case: 014 -> PASS
Test Case: 015 -> PASS
Test Case: 016 -> PASS
Test Case: 017 -> PASS
Test Case: 018 -> PASS
Test Case: 019 -> PASS
Test Case: 020 -> PASS
Test Case: 021 -> PASS
Test Case: 022 -> PASS
Test Case: 023 -> PASS
Test Case: 024 -> PASS
Test Case: 025 -> PASS
Test Case: 026 -> PASS
Test Case: 027 -> PASS
Test Case: 028 -> PASS
Test Case: 029 -> PASS
Test Case: 030 -> PASS
Test Case: 031 -> PASS
Test Case: 032 -> PASS
Test Target: Variables and Scope
Test Case: 000 -> PASS
Test Case: 001 -> PASS
Test Case: 002 -> PASS
Test Case: 003 -> PASS
Test Case: 004 -> PASS
Test Case: 005 -> PASS
Test Case: 006 -> PASS
Test Case: 007 -> PASS
Test Case: 008 -> PASS
Test Case: 009 -> PASS
Test Case: 010 -> PASS
Test Case: 011 -> PASS
Test Case: 012 -> PASS
Test Case: 013 -> PASS
Test Case: 014 -> PASS
Test Case: 015 -> PASS
Test Case: 016 -> PASS
Test Case: 017 -> PASS
Test Case: 018 -> PASS
Test Case: 019 -> PASS
Test Case: 020 -> PASS
Test Case: 021 -> PASS
Test Case: 022 -> PASS
Test Case: 023 -> PASS
Test Case: 024 -> PASS
Test Case: 025 -> PASS
Test Case: 026 -> PASS
Test Case: 027 -> PASS
Test Target: Pointers
Test Case: 000 -> PASS
Test Case: 001 -> PASS
Test Case: 002 -> PASS
Test Case: 003 -> PASS
Test Case: 004 -> PASS
Test Case: 005 -> PASS
Test Case: 006 -> PASS
Test Case: 007 -> PASS
Test Case: 008 -> PASS
Test Case: 009 -> PASS
Test Case: 010 -> PASS
Test Case: 011 -> PASS
Test Case: 012 -> PASS
Test Case: 013 -> PASS
Test Case: 014 -> PASS
Test Case: 015 -> PASS
Test Case: 016 -> PASS
Test Case: 017 -> PASS
Test Case: 018 -> PASS
Test Case: 019 -> PASS
Test Case: 020 -> PASS
Test Case: 021 -> PASS
Test Case: 022 -> PASS
Test Case: 023 -> PASS
Test Case: 024 -> PASS
Test Case: 025 -> PASS
Test Case: 026 -> PASS
Test Case: 027 -> PASS
Test Case: 028 -> PASS
Test Case: 029 -> PASS
Test Case: 030 -> PASS
Test Case: 031 -> PASS
Test Case: 032 -> PASS
Test Case: 033 -> PASS
Test Case: 034 -> PASS
Test Case: 035 -> PASS
Test Case: 036 -> PASS
Test Target: Data Handling
Test Case: 000 -> PASS
Test Case: 001 -> PASS
Test Case: 002 -> PASS
Test Case: 003 -> PASS
Test Case: 004 -> PASS
Test Case: 005 -> PASS
Test Case: 006 -> PASS
Test Case: 007 -> PASS
Test Case: 008 -> PASS
Test Case: 009 -> PASS
Test Case: 010 -> PASS
Test Case: 011 -> PASS
Test Case: 012 -> PASS
Test Case: 013 -> PASS
Test Case: 014 -> PASS
Test Case: 015 -> PASS
Test Case: 016 -> PASS
Test Case: 017 -> PASS
Test Target: C Library
Test Case: 000 -> PASS
Test Case: 001 -> PASS
Test Case: 002 -> PASS
Test Case: 003 -> PASS
Test Case: 004 -> PASS
Test Case: 005 -> PASS
Test Case: 006 -> PASS
Test Case: 007 -> PASS
Test Case: 008 -> PASS
Test Case: 009 -> PASS
Test Case: 010 -> PASS
Test Case: 011 -> PASS
Test Case: 012 -> PASS
Test Case: 013 -> PASS
Test Case: 014 -> PASS
Test Case: 015 -> PASS
Test Case: 016 -> PASS
Test Case: 017 -> PASS
Test Case: 018 -> PASS
Test Case: 019 -> PASS
Test Case: 020 -> PASS
Test Case: 021 -> PASS
Test Case: 022 -> PASS
Test Case: 023 -> PASS
Test Case: 024 -> PASS
Test Case: 025 -> PASS
Test Case: 026 -> PASS
Test Case: 027 -> PASS
Test Case: 028 -> PASS
Test Case: 029 -> PASS
Test Case: 030 -> PASS
Test Case: 031 -> PASS
Test Case: 032 -> PASS
Test Case: 033 -> PASS
Test Case: 034 -> PASS
Test Case: 035 -> PASS
Test Case: 036 -> PASS
Test Case: 037 -> PASS
Test Case: 038 -> PASS
Test Case: 039 -> PASS
Test Case: 040 -> PASS
Test Case: 041 -> PASS
Test Case: 042 -> PASS
Test Case: 043 -> PASS
Test Case: 044 -> PASS
Test Case: 045 -> PASS
Test Case: 046 -> PASS
Test Case: 047 -> PASS
Test Case: 048 -> PASS
Test Case: 049 -> PASS
Test Case: 050 -> PASS
Test Case: 051 -> PASS
Test Case: 052 -> PASS
Test Case: 053 -> PASS
Test Case: 054 -> PASS
Test Case: 055 -> PASS
Test Case: 056 -> PASS
Test Case: 057 -> PASS
Test Case: 058 -> PASS
Test Case: 059 -> PASS
Test Target: Debug functions
Test Case: 000 -> FAIL
Test Case: 001 -> FAIL
Test Case: 002 -> PASS
Test Case: 003 -> PASS
Test Case: 004 -> PASS
Test Case: 005 -> PASS
Test Case: 006 -> PASS
Test Case: 007 -> PASS
Test Case: 008 -> PASS
Test Case: 009 -> PASS
Test Case: 010 -> PASS
Test Case: 011 -> PASS
Test Case: 012 -> PASS
Test Case: 013 -> PASS
Test Case: 014 -> PASS
Test Case: 015 -> PASS
Test Case: 016 -> PASS
Test Case: 017 -> PASS
Test Case: 018 -> PASS
Test Case: 019 -> PASS
Test Case: 020 -> FAIL
Test Case: 021 -> FAIL
Test Case: 022 -> PASS
Test Case: 023 -> PASS
Test Case: 024 -> PASS
Test Case: 025 -> FAIL
Test Case: 026 -> PASS
Test Case: 027 -> PASS
Test Case: 028 -> PASS
Test Case: 029 -> FAIL
Test Summary:
Compiler Tests:
Expressions:           PASS: 48  FAIL: 0
Branching and looping: PASS: 33  FAIL: 0
Variables and Scope:   PASS: 28  FAIL: 0
Pointers:              PASS: 37  FAIL: 0
Data Handling:         PASS: 18  FAIL: 0
C Library:             PASS: 60  FAIL: 0
Debug functions:       PASS: 24  FAIL: 6
Test Runner: Done
```