ICIC Express Letters
Volume **14**, Number **7**, July **2020**

ICIC International ©2020 ISSN 1881-803X
pp. **669–677**

DESIGN OF AN ORIGINAL ARCHITECTURE CPU SUN32

Chihiro Koyama and Naohiko Shimizu

Department of Embedded Technology
School of Information and Telecommunication Engineering
Tokai University
2-3-23, Takanawa, Minato-ku, Tokyo 108-8619, Japan
9bjnm012@mail.u-tokai.ac.jp; nshimizu@keyaki.cc.u-tokai.ac.jp

Received January 2020; accepted April 2020

Abstract. *We designed and implemented an SoC and software toolchains for the system. This system includes original RISC processor sun32, peripheral such as interrupt controller, timer, UART. The SoC is written in NSL (high-level synthesis language). We have ported an ANSI C compiler, binutils and FreeRTOS for this system. Through this project, we can acquire inclusive knowledge of hardware and software domains. We present the implementation result of this SoC, the software toolchains, Dhrystone benchmark result, difficulties in the project and effectiveness of knowledge acquisition of hardware and software through this project.*
Keywords: SoC, FreeRTOS, Original RISC processor, Binutils, ANSI C compiler

1. **Introduction.** In recent years, Internet of Things (IoT) is gaining attention from industry and academia [1]. The IoT relies on Micro Control Unit (MCU), sensors as devices. Embedded system used in IoT typically has constraints on power consumption, realtimeness, low foot print of memory [2]. Real-Time Operating System (RTOS) is one of the choices for the system which offers low power consumption, realtimeness with deterministic execution in relatively small amount of memory. Task dispatcher in RTOS depends interrupt from timer or software. To develop applications on RTOS, we need to understand behavior of RTOS itself and clock by clock behavior of underlying MCU interrupt. Components from software and hardware domains complicatedly form whole computer systems. Therefore, we need inclusive knowledge of both hardware and software including processor, interrupt, toolchains, simulation tools. However, practical processors such as x86, MIPS, and ARM are too complex; thus they are difficult to understand the architectural and implementational details to use practically. To learn about computer system, the best way is to design and build actual system [3]. In our laboratory, we developed processors for education. We have used VAX-11/780 compatible processor [4], PDP-11 compatible processor named POP11 [5], i8086 compatible processor [6] for the education in hardware such as processor and OS such as UNIX V6 for software. These systems prove that they are highly effective for engineer education. However, due to the nature of CISC, the processors themselves have complicated instructions and they are difficult to learn clock by clock behavior of processor internals. To simplify and minimize time to learn, RISC processors are suitable. We design and develop a simple and minimum computer system suitable for booting FreeRTOS to acquire inclusive knowledge of both hardware and software effectively in the course of design and development. In this paper, we report effectiveness of knowledge acquisition through designing and implementing minimum computer system. Chapter 2 presents related work, Chapter 3 presents overview of sun32 ISA and its implementation, Chapter 4 presents process to support and port software toolchains like compiler, binutils, FreeRTOS for this system, Chapter

DOI: 10.24507/icicel.14.07.669

5 presents verification of the processor with Dhrystone benchmark software and Chapter 6 discusses summary of this project.

2. **Related Work.** mist32 is also an original architecture developed in University of Tsukuba [7]. It supports GCC, binutils, newlib and xv6 as operating system. It targets for practical system with a variety of instructions. It has no FreeRTOS ports. It has xv6; however, it is not suitable for learning RTOS. LEG [8] is a processor for educational purposes. This processor uses ARMv5 ISA including MMU. It can boot Linux 3.19; however, since it bases ARMv5, processor itself is complicated for educational purposes. RISC-V is open-source ISA developed in University of Calfornia, Berkeley. RISC-V only defines ISA, and it does not define implementation of processor. Users can add instruction for their purpose. It has ports of GCC, binutils, newlib, FreeRTOS. Implementation of processor itself varies and releases. Picoprocessor [9] which intends for hardware education and software education such as OS with this processor is out of scope.

3. Design of SoC with sun32.

3.1. ISA.

We designed 32 bits RISC processor [10]. We document basic information of the architecture. We define sun32 to have 32 bits 32 integer registers r0 to r31. We make r0 is constant register hardwired to zero and r31 as return address register on function call instruction. We define special purpose control status register for condition code and some reserved bits. The supported instruction is listed in Table 1. We define type of instructions as follows. Data transfer instructions mean moving data from memory to register vice versa and moving immediate value to register. Arithmetic, Shift, Logical instructions are arithmetic and logical instruction which calculates the result on ALU (Arithmetic and Logic Unit) depending on the instructions. Compare instruction compares the operands and set corresponding flags in control status register. Branch instruction is for unconditional, function call, return and conditional branch. Supervisor instruction is for moving data between registers to control status register and return from interrupt instruction. Assembler provides pseudo-instruction and converts to corresponding instructions.

Table 1. Instruction defined in sun32

Type	Mnemonic	Type	Mnemonic
Data move	lb, lbu, lh, lhu, lw, sw, sh, sb, lui	Compare	cmp
Arithmetic	add, sub, mult, multu, div, divu, rem, remu	Shift	sll, srl, sra
Branch	beq, bne, bgt, ble, b, bult, bule, bugt, buge, call, ret	Logical	and, or, xor
Pseudo	ldh, ldl, nop, mov	Supervisor	msr, mrs, reti

3.2. Interrupt handling.

We define sun32 to allow essentially only one non-maskable interrupt and external interrupt controller mask and prioritize the interrupt requests. Figure 1 shows connection between interrupt controller and processor. Interrupt controller has 8 interrupt request signals irq0 to irq7 and irq0 has the highest priority. External devices such as timer can make interrupt request via irq0-irq7 signals. Irq signals are saved in IRR (Interrupt Request Register) for interrupt handling. IMR (Interrupt Mask Register) holds mask value for IRR. Priority encoder decides which interrupt request to handle by checking IRR and IMR. ISR (In-Service Register) holds currently handling interrupt. On interrupt, interrupt controller asserts int signal to CPU. CPU checks an interrupt in instruction fetch stage for simplicity. If CPU detects interrupt signal, it returns acknowledge signal to interrupt controller. Interrupt controller returns corresponding vectoring number in next clock cycle. CPU jumps to the location indicated by vectoring number. The interrupt vector table is in memory location of zero. Each entry is word aligned

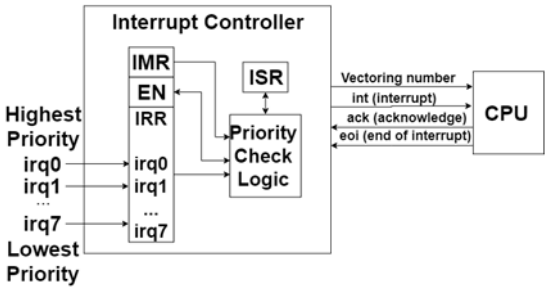

FIGURE 1. Internal structure of interrupt controller

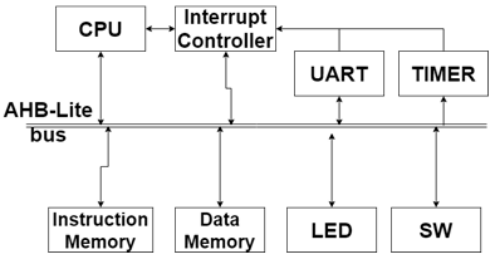

FIGURE 2. Interconnection of devices in SoC

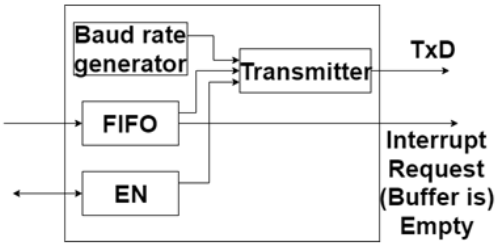

FIGURE 3. UART sender with FIFO

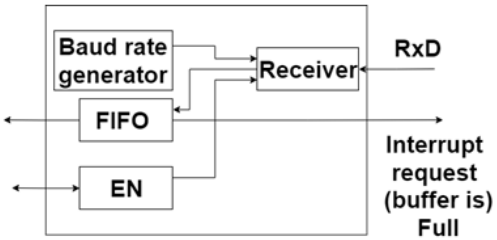

FIGURE 4. UART receiver with FIFO

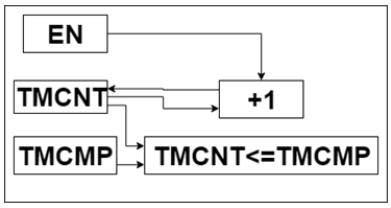

FIGURE 5. Timer module internals

and function pointer which takes no arguments and return type of void. On the exit of interrupt handler, handler must issue reti (Return from interrupt) instruction to inform end of interrupt to the interrupt handler. Programmer can issue software interrupt by writing IRR register. Interrupt controller does not allow nesting interrupt for simplicity.

3.3. **Implementation.** We implement the SoC in NSL [11]. NSL provides ability to write system in behavioral level in c-like grammar and it supports object-oriented features. Figure 2 represents interconnection of the SoC. We use AHB-Lite bus [12] to connect devices on SoC. AHB-Lite is subset of AHB and used in some ARM IP. CPU can access device's internal register via memory access instruction. We implement UART sender and receiver. It both has 256 bytes ring buffer for data. We make baud rate to 19200 bps. Figure 3 and Figure 4 show internal structure of UART. We implement internal baud rate generator for 19200 bps from 50 MHz. Figure 5 shows block diagram of timer. Each tick is in 50 MHz. We make timer to trigger interrupt request on expiry. Table 2 shows problems arisen during implementing the SoC and its resolution. We make UART module use 9600 bps originally however, in software simulation, it is to slow to simulate so we raise bps to 19200 bps. We implement processor in 3 stage multi-cycle implementation, however, with this design, critical path affects clock cycle severely, so we re-implement processor in 5 stage multi-cycle and make it faster clock frequency.

Table 2. Problem during implementing SoC

Problem	Resolution	Time spent (approx.)
UART takes time to send or receive data in software simulation.	Originally, we use 9600 bps for UART; however, we raise bps to 19200 bps. It enables faster simulation.	1 hour
Lower frequency (46.52 MHz) in previous implementation of the processor (3 stage multi-cycle implementation).	Increase stages to 5 stages. Faster frequency (66.6 MHz).	8 hours

Table 3. File lists of binutils we modified for porting

File name	Purpose	Lines
bfd/elf32-sun32.c	ELF back end for sun32	430
bfd/cpu-sun32.c	CPU information for BFD	42
cpu/sun32.cpu	CPU description for CGEN	501
cpu/sun32.opc	Supplemental routines and information for CGEN	94
gas/config/tc-sun32.c	gas ports for sun32	229
gas/config/tc-sun32.h	gas ports for sun32	52
include/elf/sun32.h	Relocation constant	35
include/elf/common.h	ELF magic number	1

Table 4. Relocation type available for sun32

Relocation type	Purpose
R_SUN32_NONE	no relocation
R_SUN32_PCREL_25	relocation of 25 bits pc relative branch
R_SUN32_HI_18	relocation of 18 bits address
R_SUN32_LO_14	relocation of 14 bits address
R_SUN32_32	section and data relocation

4. **Software Toolchain Support.** We ported binutils for binary tools like assmebler, linker, objdump. For C language support, we ported retargetable ANSI C compiler lcc [13]. These tools enable easy and practical software development on this system.

4.1. **Binutils.** Binutils are collection of binary utilities such as assembler, linker, objdump: Binutils rely on two libraries. BFD (Binary File Descriptor) library for binary file manipulation and opcodes library for assembler and disassembler. Thus, we need two libraries for porting binutils. Table 3 shows list of files for binutils ports. We use CGEN [14] for porting opcodes library. CGEN takes CPU description and generates opcodes library for the architecture. So, we did not need to write opcode library from scratch. We modified gas source file to use ported BFD and opcodes library. We use CGEN for opcode library generation so modification to gas source file is simply calling CGEN function. We use CR16's implementation of BFD as a reference since there is poor documentation for BFD backend. We support five relocations in binutils and Table 4 represents them. We use HOWTO table to describe simple relocation. Figure 6 shows HOWTO table entries. HOWTO table describes bits manipulation for relocation targets. Table 5 shows problems arisen during porting binutils and its resolution. We cannot solve disassembly mis-output yet because we use CGEN to produce opcode library, it is hard to investigate automatically generated code.

```
static reloc_howto_type sun32_elf_howto_table[] =
{
    HOWTO (R_SUN32_NONE,         /* type */
        0,                       /* rightshift */
        2,                       /* size */
        0,                       /* bitsize */
        FALSE,                   /* pc_relative */
        0,                       /* bitpos */
        complain_overflow_dont, /* complain_on_overflow
        bfd_elf_generic_reloc,   /* special_function */
        "R_SUN32_NONE",          /* name */
        FALSE,                   /* partial_inplace */
        0,                       /* src_mask */
        0,                       /* dst_mask */
        FALSE),                  /* pcrel_offset */
    HOWTO (R_SUN32_32,           /* type */
```

FIGURE 6. HOWTO table entry for SUN32_NONE

TABLE 5. Problems arisen on porting binutils

Problem	Resolution	Time spent (approx.)
Relocation does not occur on linking.	Fix HOWTO table entry.	30 hours
Wrong offset on PC relative branch.	Adjust offset to branch.	1 hour
Wrong immediate value on load upper 18 bits instruction.	Shift and AND immediate value.	1 hour
Wrong output of disassembly on some condition.	Not solved. Still in investigating.	45 hours

4.2. **ANSI C compiler.** There are a lot of compiler choices such as GCC, LLVM/Clang. Those compilers have few documentations for internal structure, so it requires time to understand internals for porting. It is not easy for understanding of compiler internals. We choose lcc a retargetable C compiler since internal structure of compiler and porting example are explained in the book [13]. It is best materials for learning practical compiler internals. Lcc is retargetable C compiler, clearly divided in machine independent front end and machine dependent backend; thus we need machine dependent backend. Machine dependent backend is in IR (Interface Record). Figure 7 shows IR for this architecture. We can define size and alignment of each type in IR. We also need to register functions to implement calling convention, creating function frames and other tasks in IR. Lcc uses

```
Interface sunIR = {
    /* size alignment outofline(dag with no c
    /* for incomplete type size is zero */
    /* size must be multiple of align */
    /* if outofline is one, constant will be
        static variable */
        1, 1, 0,  /* char */
        2, 2, 0,  /* short */
        4, 4, 0,  /* int */
        4, 4, 0,  /* long */
        4, 4, 0,  /* long long */
        4, 4, 1,  /* float */
        8, 8, 1,  /* double */
        8, 8, 1,  /* long double */
```

FIGURE 7. Definition of size and alignment in IR

```
    //stmt is operator with side effect
    //assignment
stmt: ASGNI1(addr,reg) "sb r%1,%0\n" 1
stmt: ASGNI2(addr,reg) "sh r%1,%0\n" 1
stmt: ASGNI4(addr,reg) "sw r%1,%0\n" 1
stmt: ASGNU1(addr,reg) "sb r%1,%0\n" 1
stmt: ASGNU2(addr,reg) "sh r%1,%0\n" 1
```

FIGURE 8. Tree grammar to define assignment

TABLE 6. Problems arisen on porting lcc

Problem	Resolution	Time spent (approx.)
Implementation of sun32 does not support floating point arithmetic.	Use Berkeley SoftFloat library for code generation.	10 hours
Wrong argument passing on function call.	Fix register allocation to follow calling convention.	5 hours

small specification of tree grammar to generate code generator. Figure 8 shows example of specification to define integer assignment for code generator. Table 6 shows problems arisen during porting lcc for this system and its resolution. We decide to use software float library for floating point arithmetic supports.

4.3. **FreeRTOS.** FreeRTOS is small, easy to use real time operating system and de-facto standard for MCU and small processor. It is thus suitable for small system such as sun32 SoC with no Memory Management Unit (MMU). Since FreeRTOS is small and simple system, it is easy to understand internals, so it is suited for learning practical real time operating system. We port FreeRTOS for sun32 SoC. FreeRTOS relies on some headers and functions in standard C library. We initially try to port newlib for our system; however, lcc cannot compile some code in newlib so to minimize the porting time, we determine necessary header files and functions for FreeRTOS and implement them from scratch including startup file. We have linked this library to FreeRTOS statically. Ports of FreeRTOS need modification to the files listed in Table 8. Modifications are basically configuration of FreeRTOS and register saving and restoration. We run demonstration program which just creates two tasks and make context switch at timer interrupt of 1 sec interval. Each task prints its name on UART. Result and memory usage are shown in Figure 9 and Figure 10. Sample program occupies 46560 bytes which can fit in address space of 16 bits. Table 7 shows problems arisen during porting FreeRTOS and its resolution.

```
[chihiro@archlinux sun32]$ sun32-unknown-elf-size a.out
   text    data     bss     dec     hex filename
  41872    2272    2416   46560    b5e0 a.out
[chihiro@archlinux sun32]$
```

FIGURE 9. FreeRTOS memory usage with minimal libc

```
File  Edit  View  Searc
[chihiro@archlinux
main
task 1
task 2
task 1
```

FIGURE 10. Sample program for task switch

Table 7. Problems arisen on porting FreeRTOS

Problem	Resolution	Time spent (approx.)
Newlib cannot compile with lcc.	Determine standard c facilities used in FreeRTOS and implement minimum C library.	20 hours
No facility to save and restore control status register.	Add instruction to save and restore control status register from or to general purpose register.	3 hours

Table 8. File lists of FreeRTOS port

File name	Purpose	Lines
Source/portable/lcc/sun32/port.c	Architecture dependent routines	170
Source/portable/lcc/sun32/portasm.S	Architecture dependent assembly	39
Source/portable/lcc/sun32/portmacro.h	Architecture dependent macro	151
Demo/sun32/main.c	FreeRTOS Demo program main	35
Demo/sun32/FreeRTOSConfig.h	FreeRTOS configuration	137

5. **Verification of System.** For simulation of the system, we use verilator to generate C++ codes from Verilog HDL which converted from NSL by nsl2vl then compile C++ files for executable for simulation. UART output is connected to standard output. We observe waveform on GTKWAVE. We compile software with our software toolchain for this system then load with simulator.

5.1. **Synthesis result for MAX 10.** Table 9 shows number of lines of components in SoC written in NSL. Owing to higher level of description in NSL, we can implement the SoC approximately 1000 lines of code. Table 10 shows synthesis result on Quartus 18.1.0 Lite Edition in 4 different configurations (3 and 5 stage multi-cycle and pipelined implementation). We used block RAM for instruction and data memories with 16384 words. Usage of total logic element for all configuration is nearly 10% so there is still room for other peripherals or systems. 5 stage implementations are approximately 16% faster clock frequency than 3 stage implementations.

Table 9. Modules of SoC in NSL and line size

Module name	Lines	Module name	Lines	Module name	Lines
alu32	121	interrupt_ctr	150	cla32	17
reg32	200	condcheck	42	memory_access_unit	73
core	48	instruction_fetch_unit	65	inc32	17
div32	48	ahb_lite_master	133	ahb_lite_slave	117
uart_sender	122	uart_receiver	144	timer	39
fifo	35	cache	75		

Table 10. Synthesis result with Quartus 18.1.0

Configuration	Logic elements	Registers	Fmax
5 stage multi-cycle	4573 (9%)	2193	66.6 MHz
5 stage pipe-line	4612 (9%)	2438	65.2 MHz
3 stage multi-cycle	4868 (10%)	2263	47 MHz
3 stage pipe-line	4886 (10%)	2296	47 MHz

5.2. **Dhrystone result.** We use benchmark as Dhrystone ver2.1. We build benchmark software with lcc with register attributes, binutils. This benchmark successfully ran on 4 different configurations (3 and 5 stage multi-cycle and pipelined implementation). Thus, no implementational error was found on the SoC and software toolchain. Table 11 shows the result of this benchmark on 4 different configurations. Dhrystone/sec indicates how many main loops of Dhrystone benchmark runs in one second. By dividing Dhrystone/sec with 1757 which is the result of Dhrystone benchmark on VAX11/780 gives DMIPS (Dhrystone MIPS).

Table 11. Dhrystone v2.1 result with 4 configurations

Configuration	Dhrystone/sec	DMIPS	DMIPS/MHz
5 stage multi-cycle	7129	4.05	0.06
5 stage pipe-line	24328	13.84	0.21
3 stage multi-cycle	12618	7.18	0.16
3 stage pipe-line	18726	10.65	0.22

The five stages pipelined implementation got the highest result in this section. The five stages pipelined implementation is 34% faster than the five stages multi-cycle implementation. The three stages pipelined implementation is 14% faster than the three stages multi-cycle implementation.

6. **Conclusion.** Through this project, we can effectively acquire knowledge of computer system as whole including processor, object files, RTOS, compiler, binutils and how to use software toolchain in small amount of work time. Figure 11 shows Gantt chart for work time. We spent 8 months as a total for construction of system. We spent much time in porting binutils. There are few documentations about porting or internal structure. We had to inspect other ports of binutils to understand how to port.

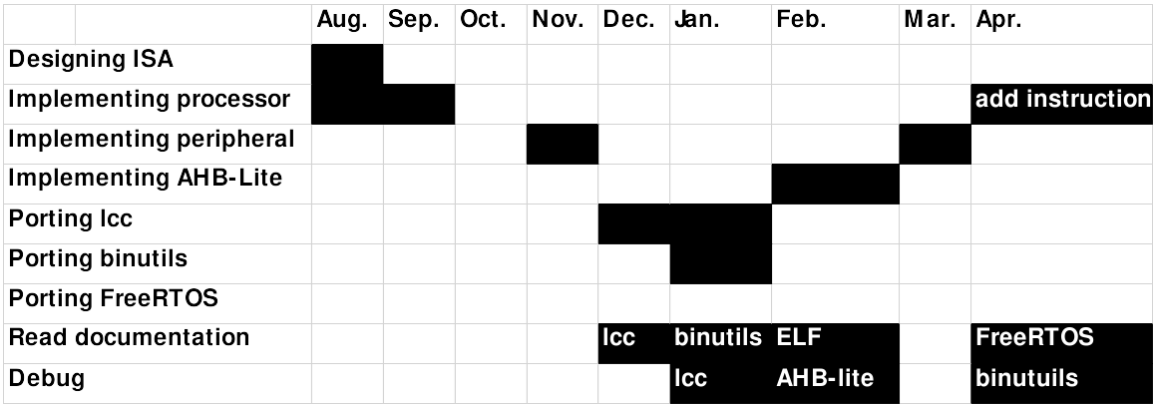

Figure 11. Gantt chart representing work time

Porting of FreeRTOS itself is relatively easy; however, implementing necessary standard C library is taking time since we cannot compile newlib with lcc.

We have gained insight of connection among components such as processor to software toolchains and deeper knowledge of processor, practical system, software toolchain. This method is effective for learning and education of both software and hardware domains ranging from processor to operating system.

REFERENCES

[1] C. Perera, C. H. Liu, S. Jayawardena and M. Chen, A survey on Internet of Things from industrial market perspective, *SIEEE Access*, vol.2, pp.1660-1679, 2014.

[2] J. Kraft, Real-time demands of the IoT, *New Electronics*, vol.49, no.15, pp.26-27, 2016.
[3] D. C. Hyde, Teaching design in a computer architecture course, *IEEE Micro*, vol.20, no.3, pp.23-28, 2000.
[4] K. Khongsomboon, N. Kondoh and N. Shimizu, Microprocessor development using SFL for educational purposes, *Proc. of the 6th International Conference on ASIC*, pp.342-345, 2005.
[5] Y. Iida and N. Shimizu, Design of POP-11 (PDP-11 on programmable chip), *Proc. of the 2004 Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair 2004*, Yokohama, Japan, pp.571-572, 2004.
[6] M. Ohyama and N. Shimizu, Development of i8086 compatible processor for VLSI design education and system on chip, *COOL Chips VII: IEEE Symposium on Low-Power and High-Speed Chips*, p.81, 2004.
[7] *https://www.ipa.go.jp/files/000028800.pdf*, Accessed on 7 May, 2020.
[8] M. Waugaman et al., LEG processor for education, *The 11th European Workshop on Microelectronics Education (EWME)*, Southampton, pp.1-5, 2016.
[9] C. J. Jiménez-Fernández, C. Baena, P. Parra, M. Valencia and A. A. Lopéz-Hinojo, Educational applications of a pico-processor design, *Technologies Applied to Electronics Teaching (TAEE)*, Seville, pp.1-5, 2016.
[10] C. Koyama, K. Sasaishi, S. Nukita and N. Shimizu, Development of original architecture CPU for SoC education, *Parthenon Society*, vol.44, pp.43-50, 2018.
[11] *http://www.overtone.co.jp/products/overture/*, Accessed on 7 May, 2020.
[12] *http://infocenter.arm.com/help/topic/com.arm.doc.ihi0033a/index.html*, Accessed on 7 May, 2020.
[13] D. R. Hanson and C. W. Fraser, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley Professional, 1995.
[14] *https://sourceware.org/cgen/docs/cgen_1.html*, Accessed on 7 May, 2020.