Andes Programming Guide for ISA V5

Document Number PG012-35

Date Issued 2025-08-28

Copyright © 2017–2025 Andes Technology Corporation.
All rights reserved.

Copyright Notice

Copyright © 2017–2025 Andes Technology Corporation. All rights reserved.

AndesCore, AndeSight, AndeShape, AndeSoft, AndeStar, AndesClarity, Andes AutoOpTune, AndeSentry Secure Boot, AndeSentry MCU-TEE, AndeSim and AndeSysC are trademarks owned by Andes Technology Corporation. Please see https://www.andestech.com/en/trademark for a complete list of Andes trademarks and logos. All other logos and product names are the property of their respective owners.

This document contains confidential information pertaining to Andes Technology Corporation. Use of this copyright notice is precautionary and does not imply publication or disclosure. Neither the whole nor part of the information contained herein may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means without the written permission of Andes Technology Corporation.

The product described herein is subject to continuous development and improvement. Thus, all information herein is provided by Andes in good faith but without warranties. This document is intended only to assist the reader in the use of the product. Andes Technology Corporation shall not be liable for any loss or damage arising from the use of any information in this document, or any incorrect use of the product.

Contact Information

Should you have any problems with the information contained herein, you may contact Andes Technology Corporation through:
Email – support@andestech.com
Website – https://es.andestech.com/eservice/

Please include the following information in your inquiries:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem

General suggestions for improvements are welcome.

Revision History

Rev.	Revision Date	Revised Content
3.5	2025/08/28	1. Updated the document template to V19. 2. Restored descriptions about V5E ISA. (Chapter 1 and Table 1) 3. Added linker options to export and import .riscv.jvt. (Section 2.3) 4. Restored descriptions about RV32E ABI. (Section 7.2 and 7.2.2) 5. Added how to use the Zcmt extension in indirect call functions for ROM patching. (Section 9.1.4) 6. Restored the note about RV32E stack alignment for the input section of linker scripts. (Section 13.1.2.4) 7. Added a reference link for differences between LD and LLD. (Section 18.10.2)
3.4	2025/04/10	1. Updated the document template to V17. 2. Updated Andes-specific compiler options. Added how to customize VLEN for the specified vector extension. (Section 2.1) 3. Updated the ISA specifications using -march since zicsr and zifencei must be specified explicitly starting from AndeSight v5.4 (Chapter 3) 4. Removed “arch”, “rvc”, “norvc” from supported arguments of the directive “.option” and removed related descriptions. (Section 5.3.1 and 5.3.2) 5. Provided detailed descriptions for memory blocks within a stack frame and modified ABI example descriptions/figures accordingly. Reorganized ABI subsections to improve coherency. (Section 7.2.1 subsections). 6. Added the calling convention descriptions for the vector ABI. (Section 7.2 and 7.2.4) 7. Added the reference link for vector cryptography intrinsic functions. (Chapter 12) 8. Added "SORT_INPUT_SECTION" as an input attribute keyword for input sections in SaG syntax. (Section 13.1.2.4) 9. Added pragmas to enable auto-vectorization for a specific loop (Section 17.5) 10. Added that LLVM from AndeSight v5.4.0 doesn't support standard vector calling convention variant whereas GCC does. (Section 18.10 and 7.2.4) 11. Added that with GCC from AndeSight v5.4 or later, the illegal instruction exception for a whole register move when vill=1 can be worked around by executing vset{i}vl{i} first. (Section 18.10) 12. Added a section about how to check -march arguments implied

Rev.	Revision Date	Revised Content
		by -mcpu=PROCESSOR. (Section 18.11)
3.3	2024/11/20	1. Removed the section "Compiler options supported in V3 but not in V5". 2. Removed descriptions about RV32E toolchain. (Chapter 1) 3. Added three compiler options: -bf16ms, -mext-ntlh, and -mext-zilsd. (Section 2.1) 4. Synced the supported standard pseudo-ops with upstream (Section 5.3.1) 5. Added four Andes pseudo-ops (.option no16bit, .option notablejump, .option noexecit, and .option innermostloop) and noted the usage change for .option norvc from AndeSight v5.4. (Section 5.3.2) 6. Updated the supported pseudo instructions. (Section 5.4) 7. Provided the RISC-V psABI documentation link as a reference for Andes ABI, noted that the code generation may vary depending on compiler implementation and optimizations, and removed descriptions about RV32E ABI. (Chapter 7) 8. Removed the comprehensive list of all __riscv_* predefined macros for the AndeStar architecture and directed users to refer to the RISC-V C API for details. (Chapter 8) 9. Fixed prototypes of the following v5 CSR intrinsics: __builtin_riscv_csrr(),__nds__mfsr(),__builtin_riscv_csrw() and __nds__mtsr. (Section 10.2.1) 10. Updated the description of the parameter LOCALITY for the intrinsics of CMO. (Section 10.2.2) 11. Gave a deprecation notice for Andes RVB and RVK intrinsics and provided the corresponding intrinsic replacements. (Chapter 12) 12. Added "EXECIT" and "JVT" to selectors of input_section_attr in SaG syntax (Section 13.1.2.4) 13. Removed the LdSaG support for v3 toolchains. Removed the --version option in the help message and the example. Removed the template file nds32_template.txt and set nds32_template_v5.txt as default. (Section 13.2) 14. Added Andes GCC compiler support on auto-vectorization for RVV and vector computation on BF16. (Section 17.5 and 18.8.2.2) 15. Supported __attribute__((cold)) and __attribute__((hot)); removed the description about the optimization level limitation of __attribute__((optimize())); noted about the instability of __attribute__((optimize())) and #pragma GCC optimize (). (Section 18.1.1) 16. Added Andes scalar/vector BFloat16 arithmetic extensions for

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Rev.	Revision Date	Revised Content
		BF16 support on Andes 45- and 65-series processors and reorganized BF16-related descriptions. (Section 18.8) 17. Noted the binary discrepancies between LLVM on Linux and on Windows and their cause. (Section 18.10)
3.2	2024/06/07	1. Added that -mext-bf16min is only supported by v5f and v5d toolchains. (Section 2.1) 2. Fixed the typo “-mstrict-align” to “-mno-strict-align”. 3. Added two linker options “--m[no-]execit-jump” and “--m[no-]truncation-check”. (Section 2.3) 4. Replaced "xv5" with "xandes" to represent the Andes V5 extension in the -march option. (Chapter 3) 5. Fixed the typos in the registers to which the value is returned when a floating-point primitive type is larger than FLEN bits. (Section 7.2.3.3) 6. Modified the description of the macro "__nds_execit" and fixed the typo on the default cmodel macro for 64-bit toolchains. (Table 13) 7. Fixed incorrect data types for the__nds__brev8 syntax in Table 22. 8. Fixed incorrect data types in example code of some RV64-only RVK intrinsics. (Section 12.4.3) 9. Supported the PROVIDE syntax in input sections of SaG files. (Sectio 13.1.2.4) 10. Modified the description of -malways-align in Section 17.1.2. 11. Removed the note that -finline-functions is only default applied at -O2 in LLVM. (Table 25) 12. Added descriptions of -mext-bf16min in Section 18.8.1. 13. Modified CMO descriptions to clarify that Andes compiler doesn't generate CMO instructions automatically. (Section 17.6 and 17.6.1) 14. Fixed a typo “-fma” to “-mfma” in Section 17.1.5 and a typo “startup.S” to “start.S” in Appendix I. 15. Added troubleshooting descriptions for the error message "Relocation truncated to fit". (Appendix III)
3.1	2023/12/04	1. Added a compiler option “-mext-vector=” and described how to enable specific vector extensions with “-mext-vector” or “-mext-vector=”. (Section 2.1) 2. Updated the compiler in Chapter 3 examples. 3. Added an intrinsic __nds__prefetch for cache management

Rev.	Revision Date	Revised Content
		operations (Table 15 and Section 10.2.2) 4. Added a note about the suggested sequence of predefined sections for input_section_selector; added descriptions about stack alignment (Section 13.1.2.4 and Appendix I) 5. Updated the I/O functions supported by the Virtual Hosting; added a note about supported open() flags and their fopen() modes. (Chapter 16) 6. Added a description about how to avoid C code that causes misaligned vector access. (Section 17.5) 7. Updated compiler options and the example code for loop data prefetch optimization and updated the intrinsic to insert CMO prefetch instructions at other positions. (Section 17.6.1 and 17.6.2)
3.0	2023/07/26	1. Updated the document template to V15. 2. Added a compiler option "-mext-bf16min" (Section 2.1) 3. Added a section to introduce Andes RVB and RVK intrinsic functions. (Chapter 12) 4. Added a section about RVV functions in MCUlib. (Section 15.4) 5. Modified -mcpu and -mtune descriptions to support 60-series cores. (Section 17.1.2) 6. Modified auto-vectorization descriptions for supporting scalable length and noted the compiler option(s) to enable and disable the optimization. (Section 17.5) 7. Added a section to introduce the uses of CMO prefetch instructions (Section 17.6)
2.9	2022/12/14	1. Removed the compiler option "-mext-zk" and added "-mext-cmo", "-mext-svinval", "-mext-zc", "-mext-zkns" and "-mext-zvlsseg". (Section 2.1) 2. Added commands to list target-specific linker options and added some EXEC.IT-related options to linker help messages; modified the description of "--m[no-]relax-cross-section-call". (Section 2.3) 3. Noted that the latest toolchains may have problems linking libraries built with an older LTO version.
2.8	2022/09/13	1. Listed only Andes-specific compiler, assembler and linker options (Section 2.1~2.3) 2. Added a macro "__ANDES" and replaced the macro "__nds_v5" with "__riscv_xandes"; removed "__nds_exec" from the default macros for v5 toolchains. (Table 13) 3. Added two linker options "--mela-cross-section-call" and "--

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Rev.	Revision Date	Revised Content
		mexecit-jal-over-2mib" for code size reduction. (Section 17.1.1) 4. Removed the description of CPU-hardwired implementation of exec.itable (Section 18.2). 5. Notified users to ignore the error messages shown when executing the command to generate exec.itable and lib.ld. Added a command to generate lib.out. (Section 18.2.2) 6. Updated the mapping of the "medium" code model for RV32.
2.7	2022/05/11	1. Added the compiler option "-mext-zbabcs" for supporting RVB hardware instructions zba, zbb, zbc and zbs. (Section 2.1) 2. Added macros __riscv_zba, __riscv_zbb, __riscv_zbc and __riscv_zbs for Zba, Zbb, Zbc and Zbs extensions. 3. Added LdSaG options "--keep-debug-macro", "--no-sort-loads" and "–executable-start". (Section 13.2)
2.6	2022/02/10	1. Revised code model descriptions for clarity.
2.5	2022/01/21	1. Added -mbf16 and -mzfh options to the compiler help messages. Noted that the two options are only supported by v5f or v5d toolchains. (Section 2.1) 2. Removed two assembler options –gstabs and –gstabs+. (Section 2.2) 3. Removed __nds__lrw, __nds__scw, __nds__lrd and __nds__scd as load and store instructions may still be generated between these intrinsic functions. (Table 17, Table 18, Section 10.2.4 and 10.2.5) 4. Modified descriptions of INCLUDE and #include for the SaG header; added a usage note for ZI section; supported gp registers in input sections. (Section 13.1.2.1 and 13.1.2.4) 5. Added that MCUlib and Newlib both are not thread-safe. (Section 15.1) 6. Explained -mcpu and -mtune options for code speed optimization (Section 17.1.2) 7. Noted that nm may not function properly for programs built with -flto. 8. Described the calling convention issue with v5f toolchains for programs having conversion between _Float and double data. 9. Described the use of intrinsic functions __nds_fcvt_bf16_s and __nds_fcvt_s_bf16 for programs compiled with -mbf16. (Section 18.7) 10. Added a section about LLVM auto-vectorization (Section 17.5) 11. Described the linker error caused by GCC 10 changing to use -

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Rev.	Revision Date	Revised Content
		fno-common by default. (Section 18.9) 12. Removed the known issue about sizeof for C++ programs in clang RISC-V ABI implementation. (Section 18.10)
2.4	2021/06/22	1. Supported the new style of architecture extension test macros and deprecated the old one; added a macro __riscv_arch_test to check if the compiler supports the new test macros. (Chapter 8) 2. Added a note that printf() may be replaced with iprintf() since AndeSight v5.0.0 official release and instructed users how to prevent the replacement of their own printf(). (Section 15.2)
2.3	2021/03/02	1. Supported the compiler option "-mcmov" for 45-series AndesCore processors.
2.2	2020/12/22	1. Removed the compiler option "-merror-on-no-atomic" and added two options "-mb20282" and "-mext-vector" (Section 2.1) 2. Added the macro "__riscv_vector". (Section 8.1/Table 13) 3. Added instrinsic functions "__nds_fcvt_s_bf16" and "__nds_fcvt_bf16_s". (Section 10.1/Table 16, Section 10.2.3) 4. Added a usage notice for the optimization option "-mfma". (Section 17.1.5) 5. Added a section about Andes compiler support for half-precision floats. (Section 18.7) 6. Added a known issue that the Clang ABI implementation will result in wrong padding in C++. (Section 18.10) 7. Modified the description of the GDB command "reset-and-hold". (Appendix II)
2.1	2020/09/14	1. Changed the document template to V14. 2. Modified the examples of intrinsic functions __nds__lrw, __nds__scw, __nds__lrd and __nds__scd (Section 10.2.4 and 10.2.5)
2.0	2020/07/14	1. Added how to reduce the linking time for large programs when LLD is in use. (Chapter 4) 2. Added that the compiler will perform zero extensions if an unsigned integer variable less than 64 bits long is declared in 64-bit AndeStar processor architecture. (Section 18.4) 3. Modified Example-va-1 in Section 18.5. 4. Listed the differences between GCC and LLVM. (Section 18.10)
1.9	2020/03/10	1. Added the support of 32-bit Linux toolchains (Chapter 1) 2. Added LLVM clang and lld support to Andes toolchains (Chapter

Rev.	Revision Date	Revised Content
		1, Section 9.1.3, Section 9.2.3, Section 13.2, Chapter 16, Chapter 17) 3. Listed LLVM compiler options. 4. Described Andes multilib Linux toolchains and how to build program with them (Chapter 3) 5. Described that the LLD or BFD linker used by Andes toolchains. (Chapter 4) 6. Added clang support to AndeStar V5 predefined macros. 7. Added a LLVM-specific code size optimization option "-oz" (Section 17.1.1) 8. Described how "-funroll-loops" and "-funroll-all-loops" work with LLVM compiler. (Section 17.1.2) 9. Marked that "-finline-functions" is default applied at -o2 level for LLVM only. (Table 25) 10. Described the function attribute and pragma to disable optimization in LLVM (Section 18.1.2) 11. Listed the identifiers that should not be redefined when using LTO and provided an example of marking a function with "__attribute__((used))". 12. Updated start.S in V5 startup demos so that the vector table can be used for PLIC or CLIC interrupts. (Appendix I)
1.8	2019/12/06	1. Updated the tutorials on how to use indirect calls or function table mechanism to patch functions in ROM (Section 9.1.3 and 9.2.3)
1.7	2019/11/19	1. Changed the document template to V13. 2. Added descriptions of ROM patching (Chapter 9) 3. Modified the "address" description in the SaG syntax so that its value can be a macro defined by the DEFINE keyword (Section 13.1.2.2, 13.1.2.3 and 13.1.2.5) 4. Changed the return values of _nds_scw and of _nds_scd (Section 10.2.4 and 10.2.5)
1.6	2019/04/11	1. Fixed typos in Section 5.4.
1.5	2019/03/28	1. Added V5E to Andes ISA (Chapter 1) 2. Listed Andes-specific command line options. 3. Added SCATTERASSERT to the syntax of SaG-formatted scripts. (Section 13.1.2) 4. Added that the __nds_execit macro is enabled by default only on ELF toolchains and the exec.it extension is only supported by ELF toolchains. (Section 17.1.4 and Section 18.2)

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Rev.	Revision Date	Revised Content
		5. Added descriptions of the RV32E ABI. 6. Added the predefined macro __riscv_32e.
1.4	2018/11/16	1. Added descriptions about Andes PLIC intrinsic functions (Chapter 11)
1.3	2018/7/26	1. Renamed v5m to v5 2. Replaced nds32le-elf-[gcc|as|ld|gcov] with riscv[32|64]-elf-[gcc|as|ld|gcov]. 3. Renamed EX9 to EXEC.IT (Section 2.1, Section 5.3.2, Section 17.1.4, Section 17.1.6, Section 18.2 and subsections, Appendix I) 4. Described how Andes V5 toolchains correspond to RISC-V extensions and included glibc and newlib for the library supports. (Chapter 1) 5. Updated target-specific compiler options; replaced the table "supported compiler options in AndeStar V3 and V5" with a list of V3 options no longer available in V5; noted that -mex9 and -munaligned-access are renamed in V5 (Section 2.1) 6. Added -mext-dsp to assembler options (Section 2.2) 7. Updated linker options (Section 2.3) 8. Removed "!" as it can't be used as a comment symbol for RISC-V toolchains and deleted "#" from integer formats. (Section 5.1) 9. Added a pseudo-op ".attribute Tags" (Section 5.3.2) 10. Removed the description that AndeStar V5 only supports aligned data access. (Section 6.2) 11. Added descriptions of AndeStar V5 floating point ABI (Section 7.2, Section 7.2.3.1~7.2.3.3) 12. Added __riscv_cmodel_large to V5 predefined macros (Table 13); listed the default presence of V5 macros in 32-bit/64-bit V5 toolchains. 13. Added ENTRY, EXTERN, #include to the header syntax of SaG-formatted scripts. (Section 13.1.2) 14. Modified the description of max_size so that its value can be a local variable defined by the DEFINE keyword (Section 13.1.2.2 and 13.1.2.3) 15. Added LdSaG options -h, -v, --version, and -load-zi and updated descriptions of the LdSaG example (Section 13.2) 16. Added that users also need to overwrite _fstat() in libgloss for printf to output strings. (Section 15.2) 17. Added thajt newlib toolchains support Virtual Hosting. (Chapter 16)

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Rev.	Revision Date	Revised Content
		18. Changed the instruction table for EXEC.IT to 1024 entries and changed imm9u to imm10u. (Section 18.2) 19. Added how to keep the imported look-up table after using options "--mimport-execit". (Section 18.2.1) 20. Described the operation of "-Wl,--mupdate-execit" and the usage of "-Wl,--mexecit-limit". (Section 18.2.2) 21. Described the usage of -msave-restore. (Section 17.2) 22. Added description of -mcmodel=large and noted that it's alias to -mcmodel=medany on RV32. 23. Updated the description of start.S (Appendix I)
1.2	2018/06/06	1. Changed the document template to V12 2. Fixed typos in Section 6.1 and Section 18.2 ($ITB to $uitb) 3. Added that functions converting decimal strings to floating points in MCUlib have a precision limitation. (Section 15.3) 4. Updated Table 25 by applying -fno-delete-null-pointer-checks at -O0, -Og and -O1 and applying -mex9 at -Os2. 5. Added descriptions about attribute__((optimize())) and #pragma GCC optimize. (Section 17.2) 6. Added a function attribute "no_ex9" to disable the ex9 optimization for specific functions. (Section 18.2.3)
1.1	2017/11/16	1. Updated target-specific compiler options (Section 2.1) and added "-mex9" to AndeStar V5 toolchains. 2. Added a RISC-V option and some Andes-specific options to the assembler. (Section 2.2) 3. Added Andes-specific options to the linker (Section 2.3) 4. Added pseudo-ops for ex9 optimization, linker relaxation, innermost loop, generation of non-PIC code, and push and pop. (Section 5.3.2) 5. Changed the default C/C++ type for RISC-V from signed char to unsigned char. (Section 7.1.4) 6. Added predefined macros for V5m toolchains (Table 13) and added reference tables to list default presence of macros to V5 toolchains. 7. Added intrinsic functions "get_current_sp" and "set_current_sp"; changed "unsigned long" to "long" for ecall functions. (Table 14 and Section 10.2.1) 8. Added alias intrinsic functions for CSR read/write instructions and other alias intrinsic functions for compatibility reason. (Table 14, Table 16, Section 10.2.1 and 10.2.3)

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Rev.	Revision Date	Revised Content
		9. Added descriptions of EX9 optimization. (Section 17.1.4 and 18.2) 10. Added "-msave-restore", "-minnermost-loop", "-mex9" and "-mgp-insn-relax" to Table 25. 11. Noted that compiler options starting with -O, -g, -f and -m must be applied along with the -flto option. 12. Added descriptions for start.S in Andes startup demo programs. (Appendix I) 13. Noted that the suffix "L" to constant values is required if users want to use GDB to set registers in a 64-bit AndesCore processor. (Appendix II)
1.0	2017/09/27	Initial Release

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Table of Contents

COPYRIGHT NOTICE I

CONTACT INFORMATION I

REVISION HISTORY II

LIST OF TABLES XVI

LIST OF FIGURES XVII

1. OVERVIEW 1

2. COMMAND LINE OPTIONS 3

2.1. COMPILER OPTIONS 4
2.2. ASSEMBLER OPTIONS 9
2.3. LINKER OPTIONS 10

3. MULTILIB LINUX TOOLCHAINS 12

4. LLD AND BFD LINKERS 13

5. V5 ASSEMBLY LANGUAGE 15

5.1. GENERAL SYNTAX 15
5.2. REGISTERS 15
5.3. PSEUDO-OPS 16
5.3.1. Standard pseudo-ops *16*
5.3.2. Andes pseudo-ops *20*
5.4. PSEUDO INSTRUCTIONS 21
5.5. MACROS 26
5.5.1. Creating macros in assembly code *26*
5.5.2. Assembler directives for macros *26*

6. MACHINE INSTRUCTIONS 28

6.1. 32/16-BIT 28
6.2. ENDIANNESS 28

7. APPLICATION BINARY INTERFACE (ABI) 29

7.1. DATA TYPES 30
7.1.1. Byte ordering *30*
7.1.2. Primitive data types *30*
7.1.3. Composite data types *31*
7.1.4. C language mapping of Andes platform *32*
7.2. CALLING CONVENTION 33
7.2.1. Integer ABI *33*

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7.2.2. *RV32E ABI* ... 44
7.2.3. *Floating point ABI* ... 45
7.2.4. *Vector ABI* ... 47

8. ANDESTAR V5 PREDEFINED MACROS ... 49

9. ROM PATCHING ... 51

9.1. INDIRECT CALL FUNCTIONS ... 52
9.1.1. *Implementation of indirect call functions* ... 52
9.1.2. *Limitations* ... 53
9.1.3. *Tutorial* ... 54
9.1.4. *Using Zcmt extension in patch code* ... 57
9.2. FUNCTION TABLE MECHANISM ... 58
9.2.1. *Implementation of function table mechanism* ... 58
9.2.2. *Limitations* ... 59
9.2.3. *Tutorial* ... 59

10. ANDESTAR V5 INTRINSIC FUNCTION PROGRAMMING ... 63

10.1. SUMMARY OF ANDESTAR V5 INTRINSIC FUNCTIONS ... 63
10.2. DETAILED INTRINSIC FUNCTION DESCRIPTIONS ... 72
10.2.1. *Intrinsics for RV32I and RV64I* ... 72
10.2.2. *Intrinsic for RV32 and RV64 Cache Management Operations "CMO"* ... 86
10.2.3. *Intrinsics for RV32 and RV64 floating-point extensions "F" and "D"* ... 88
10.2.4. *Intrinsics for RV32 and RV64 atomic extension "A"* ... 99
10.2.5. *Intrinsics for RV64-only atomic extension "A"* ... 108

11. ANDES PLIC INTRINSIC FUNCTIONS ... 117

11.1. SUMMARY OF ANDES PLIC INTRINSIC FUNCTIONS ... 117
11.2. DETAILED INTRINSIC FUNCTION DESCRIPTIONS ... 119
11.2.1. *Intrinsics for PLIC* ... 119
11.2.2. *Intrinsics for PLIC_SW* ... 127

12. ANDES RVB AND RVK INTRINSIC FUNCTIONS ... 134

12.1. SUMMARY OF ANDES RVB INTRINSIC FUNCTIONS ... 135
12.2. DETAILED DESCRIPTIONS OF ANDES RVB INTRINSIC FUNCTIONS ... 136
12.2.1. *Intrinsics for RV32 and RV64 RVB extension* ... 136
12.2.2. *Intrinsics for RV64-only RVB extension* ... 142
12.3. SUMMARY OF ANDES RVK INTRINSIC FUNCTIONS ... 145
12.4. DETAILED DESCRIPTIONS OF ANDES RVK INTRINSIC FUNCTIONS ... 147
12.4.1. *Intrinsics for RV32 and RV64 RVK extension* ... 147
12.4.2. *Intrinsics for RV32-only RVK extension* ... 161
12.4.3. *Intrinsics for RV64-only RVK extension* ... 173

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

13. LINKER SCRIPT GENERATION 184
13.1. Script format SaG and its syntax 184
13.1.1. *BNF notation for SaG syntax* 184
13.1.2. *Formal syntax of SaG format* 186
13.2. Linker script generator (LdSaG) 202
14. OBJECT FILES 204
14.1. ELF file 204
14.2. Examining ELF file 205
15. ANDES MCULIB 207
15.1. Features of MCUlib 207
15.2. MCUlib printf implementation 207
15.3. Precision limitation of floating point conversion functions 209
15.4. MCUlib's RISC-V Vector Extension (RVV) Functions 210
16. VIRTUAL HOSTING 211
17. ADVANCED OPTIMIZATION 212
17.1. Optimization options 212
17.1.1. *Options for code size optimization* 212
17.1.2. *Options for code speed optimization* 213
17.1.3. *Options to remove unused sections* 215
17.1.4. *Options to use EXEC.IT optimization* 215
17.1.5. *Notice on some optimization options* 216
17.1.6. *Optimization levels and default applied options* 217
17.2. Saving code size for function prologue and epilogue 218
17.3. Addressing space for programs 220
17.3.1. *Code models* 221
17.4. Link time optimization 222
17.4.1. *Using LTO* 222
17.4.2. *Notice when applying LTO* 222
17.5. Auto-vectorization for RISC-V V extension 224
17.6. Cache management operations 226
17.6.1. *Loop data prefetch optimization* 226
17.6.2. *CMO prefetch instructions for other instructions* 227
18. ADVANCED PROGRAMMING 228
18.1. Function attribute and pragma for optimization 228
18.1.1. *Function attribute and pragma for optimization in GCC* 228
18.1.2. *Function attribute and pragma to disable optimization in LLVM* 231
18.2. Compression optimization through look-up table (EXEC.IT) 232

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.2.1. Export and import *232*
18.2.2. Look-up table shared by multiple separately-linked program modules *233*
18.2.3. Disable EXEC.IT optimization for specific functions *234*
18.3. PRIMITIVE DATA TYPE "INT" 235
18.4. PRIMITIVE DATA TYPE "UNSIGNED INT" 237
18.5. FUNCTION WITH VARIABLE NUMBER OF ARGUMENTS 239
18.6. INLINE ASSEMBLY PROGRAMMING 241
18.6.1. General *241*
18.6.2. Symbolic operand name *242*
18.6.3. Clobber list *242*
18.6.4. Read-write operand *243*
18.6.5. Constraint modifier "&" *245*
18.6.6. Volatile *246*
18.7. HALF-PRECISION FLOATING POINT 247
18.8. BRAIN FLOATING POINT (BFLOAT16 OR BF16) 248
18.8.1. RISC-V minimal BFloat16 extension *248*
18.8.2. Andes Scalar/Vector BFloat16 arithmetic extension *249*
18.8.3. Andes Scalar BFLOAT16 conversion extension (XAndesBFHCvt) *255*
18.9. PORTING TO GCC 256
18.10. DIFFERENCES BETWEEN GNU AND LLVM COMPILERS AND LINKERS 257
18.10.1. GCC vs. Clang *257*
18.10.2. LD vs. LLD *258*
18.11. -MARCH ARGUMENTS IMPLIED BY -MCPU 259

APPENDIX 260

APPENDIX I. START.S 260
APPENDIX II. PROGRAMMING TIPS 262
Moving libc.a to the beginning of text section *262*
Displaying register information and debugging on reset by GDB commands *263*
APPENDIX III. TROUBLESHOOTING 264
Error message "Relocation truncated to fit" *264*

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

List of Tables

Table 1. RISC-V extensions corresponding to Andes toolchains and AndeStar ISA 1
Table 2. Vector extensions enabled when applying -mext-vector or -mext-vector= 6
Table 3. Extensions enabled when applying -mext-vector or -mext-vector= with -mzfh 7
Table 4. Extensions enabled when applying -mext-vector or -mext-vector= with -mext-bf16min 7
Table 5. Supported standard pseudo ops 16
Table 6. Supported standard pseudo instructions 21
Table 7. Supported pseudo instructions for accessing control and status registers 24
Table 8. Size and byte alignment of primitive data types 30
Table 9. Mapping of C primitive data types 32
Table 10. Andes GPRs with integer ABI usage convention 33
Table 11. Andes GPRs with RV32E ABI usage convention 44
Table 12. Andes FPRs with their ABI usage convention 45
Table 13. AndeStar custom predefined macros 49
Table 14. Intrinsics for RV32I and RV64I 63
Table 15. Intrinsics for RV32 and RV64 Cache Management Operations “CMO” 67
Table 16. Intrinsics for RV32 and RV64 floating-point extensions “F” and “D” 68
Table 17. Intrinsics for RV32 & RV64 atomic extension “A” 70
Table 18. Intrinsics for RV64-only atomic extension “A” 71
Table 19. PLIC Intrinsic Function Syntax 117
Table 20. PLIC SW Intrinsic Function Syntax 118
Table 21. RVB Intrinsic Function Syntax 135
Table 22. RVK Intrinsic Function Syntax 145
Table 23. Code size optimization levels 212
Table 24. Two loop unrolling optimization 214
Table 25. Default applied compiler options at each optimization level 217
Table 26. Causes and workarounds of “Relocation truncated to fit” 264

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

List of Figures

Figure 1. AndeStar V5 integer ABI stack frame scenario 35
Figure 2. Function prologue for stack frame construction 36
Figure 3. Function epilogue for stack frame destruction 36
Figure 4. V5 ABI stack frame layout 37
Figure 5. Blocks in a stack frame 38
Figure 6. ABI example of simple function stack frame 39
Figure 7. ABI example of calling a function with arguments 40
Figure 8. ABI example of outgoing arguments in stack frame 41

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Typographical Convention Index

Document Element	Font	Font Style	Size	Color
Normal text	Georgia	Normal	12	Black
Command line, source code or file paths	Lucida Console	Normal	11	Indigo
VARIABLES OR PARAMETERS IN COMMAND LINE, SOURCE CODE OR FILE PATHS	**LUCIDA CONSOLE**	**BOLD + ALL-CAPS**	11	**INDIGO**
Hyperlink	Georgia	Underlined	12	Blue

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

1. Overview

Andes toolchains are a part of Andes Board Support Package (BSP) and AndeSight, an integrated development environment for software development. They are mainly used for compiling, assembling, and linking users' C/C++ and assembly programs and generating executables of AndeStar V5, a superset of RISC-V instruction set architecture. For detailed information about AndeSight and AndeStar, see *AndeSight User Manual*, *AndeStar V5 Instruction Extension Specification* and *The RISC-V Instruction Set Manual - Volume I: User-Level ISA, Version 2.2*.

Andes toolchains cover four AndeStar ISA V5 implementations: V5, V5F, V5D and V5E. They correspond to RISC-V extensions as follows:

Table 1. RISC-V extensions corresponding to Andes toolchains and AndeStar ISA

Prefix \ ISA	V5	V5F	V5D	V5E
nds32-elf	rv32imac_zicsr_zifencei	rv32imafc_zicsr_zifencei	rv32imafdc_zicsr_zifencei	rv32emac_zicsr_zifencei
nds64-elf	rv64imac_zicsr_zifencei	rv64imafc_zicsr_zifencei	rv64imafdc_zicsr_zifencei	
nds32-linux	rv32imac_zicsr_zifencei		rv32imafdc_zicsr_zifencei	
nds64-linux	rv64imac_zicsr_zifencei		rv64imafdc_zicsr_zifencei	

NOTE

1. Andes ELF toolchains support A extensions only when the `-matomic` option is specified.
2. Starting from AndeSight v5.4, the I or E extension no longer implies the zicsr and zifencei extensions. As a result, zicsr and zifencei must be specified explicitly if the ISA is configured using `-march.` For example, when specifying `-march=rv64imacxandes`, ensure that the options `_zicsr` and `_zifencei` are also included for proper functionality.

Andes toolchains are built from GNU and LLVM, thus they inherit gcc, as, and ld options of GNU and clang and lld options of LLVM. In addition to GNU- and LLVM-based options, Andes specific options are provided for some unique features such as performance and code size tradeoff of AndeStar.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Andes library support includes glibc, Newlib and MCUlib. Glibc is for OS-based applications and the other two are for non-OS applications. Newlib is a C library intended for use on embedded systems. MCUlib is a C library with Andes optimization enhancement for MCU applications and smaller code size than Newlib.

This document focuses on the usages of the compiler and assembler for toolchains of AndeStar ISA V5. The following outlines the structure of this document:

- Chapter 2 describes basic usage of toolchains.
- Chapter 3 describes how to build programs with Andes multilib Linux toolchains.
- Chapter 4 describes the LLD or BFD linker used by Andes toolchains.
- Chapter 5 describes basic syntax, pseudo-ops, pseudo-instructions and macros of V5 assembly language. Programmers can write assembly with these capabilities.
- Chapter 6 describes machine instructions.
- Chapter 7 describes Application Binary Interface (ABI).
- Chapter 8 describes Andes C predefined macros.
- Chapter 9 describes ROM patching approaches.
- Chapter 10 describes Andes intrinsic functions for programming.
- Chapter 11 introduces intrinsic functions to access Andes Platform-Level Interrupt Controller (PLIC).
- Chapter 12 introduces intrinsic functions for operations in RISC-V Bit-Manipulation (RVB) and Scalar Cryptography (RVK) extensions.
- Chapter 13 introduces a simple mechanism to generate linker scripts.
- Chapter 14 describes the object file format.
- Chapter 15 describes Andes MCUlib.
- Chapter 16 depicts Virtual Hosting.
- Chapter 17 introduces various optimization techniques.
- Chapter 18 describes advanced or specialized programming techniques for specific requirements.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

2. Command line options

Use the environment variable `$PATH` to include the path of Andes GNU toolchain executable files. For example,

- To include a toolchain path for 32-bit cores

```
mypc> export PATH=/home/users/andesight/nds32le-elf-mculib-v5/bin:$PATH
mypc> echo $PATH /home/users/andesight/nds32le-elf-mculib-
v5/bin:/bin:/usr/bin
```

- To include a toolchain path for 64-bit cores

```
mypc> export PATH=/home/users/andesight/nds64le-elf-mculib-v5/bin:$PATH
mypc> echo $PATH
/home/users/andesight/nds64le-elf-mculib-v5/bin:/bin:/usr/bin
```

The prefix of toolchain executable files for 32-bit cores is `riscv32-elf-` and that for 64-bit cores is `riscv64-elf-`.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

2.1. Compiler options

To get a list of supported compiler options for 32-bit or 64-bit cores, use the following commands:

```
mypc> riscv[32|64]-elf-[gcc|clang] --help
```

To obtain target specific options for GCC, enter

```
mypc> riscv[32|64]-elf-gcc --target-help
```

To obtain target specific options for LLVM, enter

```
mypc> riscv[32|64]-elf-clang --help-hidden
```

Among the supported GNU or LLVM compiler options, the following are applicable only to Andes targets:

-Os1	Optimize for size level 1. This option will disable execit to prevent performance drop.
-Os2	Optimize for size level 2. This option will disable execit for innermost loop to prevent performance drop.
-Os3	Optimize for size level 3 which mean don't care performance.
-mace-s2s=	Argument for pass to Andes's ACE source-to-source translator.
-malways-align	Always align function entry, jump target and return address.
-matomic	Use atomic extension instructions.
-mbf16	Support Andes BFLOAT16 conversion extension
-mbf16ms	Support Andes Mode Switch BFLOAT16 extension
-mconfig-mul=	Specify configuration of instruction mul.
-mcpu=**PROCESSOR**	Optimize the output for **PROCESSOR**. Same as -mtune=.
-mctor-dtor	Enable constructor/destructor feature.
-mdiv	Use hardware instructions for integer division.
-mexecit	Use special directives to guide linker to perform exec.it optimization.
-mext-bf16min	Support RISC-V minimal BFloat16 extension (Zfbfmin, Zvfbfmin and Zvfbfwma)
-mext-cmo	Use CMO hardware instructions including Zicbom, Zicboz, Zicbop

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

-mext-dsp	Use hardware DSP instructions.
-mext-ntlh	Enable non-temporal locality hints instructions.
-mext-svinval	Enable fine-grained address-translation cache invalidation
-mext-vector	Enable RISC-V vector extension
-mext-vector=	Enable RISC-V vector extensions for embedded processors. Permissible values for this option are: 'zve32x', 'zve32f', 'zve64x', 'zve64f', and 'zve64d'.
-mext-zbabcs	Use RVB hardware instructions including zba, zbb, zbc, zbs.
-mext-zc	Enable Code Size Reduction extensions.
-mext-zilsd	Enable RV32 load store pair extensions.
-mext-zkn	Enable NIST Algorithm Suite.
-mext-zkns	Enable NIST Algorithm Suite and ShangMi Algorithm Suite.
-mext-zks	Enable ShangMi Algorithm Suite.
-mext-zvkns	Enable Vector Bit-manipulation and NIST Algorithm Suite and ShangMi Algorithm
-mext-zvlsidx	Enable RISC-V vector index load-store extension.
-mext-zvlss	Enable RISC-V vector strided load-store extension.
-mext-zvlsseg	Enable RISC-V vector zvlsseg extension.
-mfma	Generating fma instructions.
-mgp-insn-relax	Use special directives to guide linker doing gp insn relax optimization.
-minnermost-loop	Insert the innermost loop directive.
-mmove-bytes-per-loop=N	Set N bytes of data movement can be handled per loop iteration.
-mno-16-bit	Do not genearte C extension instructions.
-mno-zcmp	Disable Zcmp extension.
-mno-zcmt	Disable Zcmt extension.
-mno-zvfh	Disable zvfh support.
-mrestrict-even-reg-for-regpair	Restrict register pair must start with even register number.
-mvh	Enable Virtual Hosting support.
-mzfh	Support Zfh half-precision floating-point extension

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

NOTE

1. If you specify the options `-mcmodel` or `-mvh` for compilation, use the compiler to link programs and apply these options for linking as well.
2. The options `-mbf16, -mbf16ms, -mext-bf16min,` and `-mzfh` are only supported by v5f and v5d toolchains.
3. The option `-mext-vector` or `-mext-vector=EXTENSION` is used to enable specific vector extension(s). Table 2 below summarizes corresponding extensions that can be enabled when applying the option to processors of different ISA. If `-mext-vector` and `-mext-vector=EXTENSION` are specified at the same time, Andes compiler will prioritize the value provided by `-mext-vector=EXTENSION`.

Table 2. Vector extensions enabled when applying -mext-vector or -mext-vector=

Compiler option \ ISA	rv32v5	rv64v5	rv[32|64]v5f	rv[32|64]v5d
`-mext-vector`	zve32x	zve64x	zve32f	zve64d
`-mext-vector=zve32x`	zve32x	zve32x	zve32x	zve32x
`-mext-vector=zve32f`	INVALID	INVALID	zve32f	zve32f
`-mext-vector=zve64x`	zve64x	zve64x	zve64x	zve64x
`-mext-vector=zve64f`	INVALID	INVALID	zve64f	zve64f
`-mext-vector=zve64d`	INVALID	INVALID	INVALID	zve64d

Note that both `-mext-vector` and `-mext-vector=EXTENSION` enable zvl128b by default, setting the minimum vector length to 128 bits (i.e., VLEN ≥128). To customize the minimum vector length, just adopt the option `-mext-vector=EXTENSION`, where `EXTENSION` can be one of the following: `zvl[32|64|128|256|512|1024|2048]b`. You can also specify both zve* and zvl* extensions together with `vector=EXTENSION`, such as `-mext-vector=zve64d_zvl2048b`.

When used together with `-mzfh` or –`mext-bf16min`, the option `-mext-vector` or `-mext-vector="EXTENSION"` not only enables standard vector extensions listed in Table 2 but

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

also scalar or vector extensions for half-precision floating-point or brain floating-point, as shown in Table 3 and Table 4.

Table 3. Extensions enabled when applying -mext-vector or -mext-vector= with -mzfh

Compiler options \ ISA	rv[32|64]v5f	rv[32|64]v5d
-mzfh -mext-vector	zve32f +zfh +zvfh	zve64d +zfh +zvfh
-mzfh -mext-vector=zve32x	zve32x +zfh	zve32x +zfh
-mzfh -mext-vector=zve32f	zve32f +zfh +zvfh	zve32f +zfh +zvfh
-mzfh -mext-vector=zve64x	zve64x +zfh	zve64x +zfh
-mzfh -mext-vector=zve64f	zve64f +zfh +zvfh	zve64f +zfh +zvfh
-mzfh -mext-vector=zve64d	INVALID	zve64d +zfh +zvfh

Table 4. Extensions enabled when applying -mext-vector or -mext-vector= with -mext-bf16min

Compiler options \ ISA	rv[32|64]v5f	rv[32|64]v5d
-mext-bf16min -mext-vector	zve32f +zfbfmin +zvfbfmin	zve64d +zfbfmin +zvfbfmin
-mext-bf16min -mext-vector=zve32x	zve32x +zfbfmin	zve32x +zfbfmin
-mext-bf16min -mext-vector=zve32f	zve32f +zfbfmin +zvfbfmin	zve32f +zfbfmin +zvfbfmin
-mext-bf16min -mext-vector=zve64x	zve64x +zfbfmin	zve64x +zfbfmin

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

ISA / Compiler options	rv[32|64]v5f	rv[32|64]v5d
`-mext-bf16min` `-mext-vector=zve64f`	zve64f +zfbfmin +zvfbfmin	zve64f +zfbfmin +zvfbfmin
`-mext-bf16min` `-mext-vector=zve64d`	INVALID	zve64d +zfbfmin +zvfbfmin

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

2.2. Assembler options

To get a list of supported assembler options for Andes 32- or 64-bit cores, use the following commands:

```
mypc> riscv[32|64]-elf-as --help
```

Among the supported assembler options, the following are applicable only to Andes targets:

```
-mno-16-bit         don't generate rvc instructions
-matomic            enable atomic extension
-mace               Support user defined instruction extension
-O1                 optimize for performance
-Os                 optimize for space
-mext-dsp           enable dsp extension
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

2.3. Linker options

To get a list of supported GNU linker options for Andes 32- or 64-bit cores, use the following commands:

```
mypc> riscv[32|64]-elf-ld --help
```

To get a list of support LLVM linker options for Andes 32- or 64-bit cores, issue as follows:

```
mypc> riscv[32|64]-elf-ld.lld --help
```

To obtain target specific GNU linker options, enter

```
mypc> riscv[32|64]-elf-ld --target-help
```

To obtain target specific LLVM linker options, enter

```
mypc> riscv[32|64]-elf-ld.lld --help-hidden
```

Among the supported GNU or LLVM linker options, the following are applicable only to Andes targets:

```
--mexport-symbols=FILE    Export global symbols into linker script
--m[no-]relax-cross-section-call   Enable/Disable linker relaxation of
                                   cross- section call pseudo-instructions
--mno-workaround          Disable all workarounds
--m[no-]truncation-check  Disable/Enable the check for relocation truncated
                          to fit
--m[no-]execit            Disable/Enable link-time EXEC.IT relaxation
--mexport-execit=FILE     Export .exec.itable after linking
--mimport-execit=FILE     Import .exec.itable for EXEC.IT relaxation
--mkeep-import-execit     Keep imported .exec.itable
--mupdate-execit          Update existing .exec.itable
--mexecit-limit=NUM       Set maximum number of entries in .exec.itable for
                          this times
--mexecit-loop-aware      Avoid generating EXEC.IT instruction inside loop
--m[no-]execit-fls        Disable/Enable EXEC.IT for floating load/store
                          instructions
--m[no-]execit-rvv        Disable/Enable EXEC.IT for RVV instructions
--m[no-]execit-xdsp       Disable/Enable EXEC.IT for XDSP instructions
--m[no-]execit-auipc      Disable/Enable EXEC.IT conversion for auipc
                          instruction
--m[no-]execit-jal        Disable/Enable EXEC.IT conversion for jal
                          instruction
--m[no-]execit-jal-over-2mib    Disable/Enable EXEC.IT conversion for jal
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
                          instruction over the first 2MiB page of text
                          section
--m[no-]execit-jump       Disable/Enable EXEC.IT conversion for JUMP
                          instructions
--mexport-table-jump=FILE Export .riscv.jvt after linking
--mimport-table-jump=FILE Import .riscv.jvt for table jump relaxation
```

Note that `--mno-workaround` and `--m[no-]truncation-check` are supported only by the GNU linker, while the other Andes-specific options are available for both the GNU and LLVM linkers.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

3. Multilib Linux toolchains

Andes 32-bit and 64-bit Linux toolchains provides GCC compiler and base libraries for AndeStar ISA V5 and V5D. The multilib toolchains build programs with ISA V5D by default. Yet you can use options `-march` and `-mabi` to specify another ISA and ABI that are compatible with your build target. For example, given a program `foo.c` being built as follows, it will be complied with "`-march=rv[32|64]imafdc_zicsr_zifencei_xandes -mabi=[ilp32d|lp64d]`" by default:

```
$ riscv[32|64]-linux-gcc foo.c
```

To build the program with ISA V5 and associated ABI, just specify the ISA and ABI with `-march` and `-mabi` as follows:

```
$ riscv[32|64]-linux-gcc -march=rv[32|64]imac_zicsr_zifencei_xandes
-mabi=[ilp32|lp64d] foo.c
```

The toolchains also provide alias options to directly map to different ISA and ABI combinations, just like what `-march` and `-mabi` options do. These alias options are summarized in the following table.

Alias Name	Automatic Expansion
-march=rv32v5	-march=rv32imac_zicsr_zifencei_xandes -mabi=ilp32
-march=rv32v5d	-march=rv32imafdc_zicsr_zifencei_xandes -mabi=ilp32d
-march=rv64v5	-march=rv64imac_zicsr_zifencei_xandes -mabi=lp64
-march=rv64v5d	-march=rv64imafdc_zicsr_zifencei_xandes -mabi=lp64d

NOTE

Using `xv5` to represent the Andes V5 extension in the `-march` option (e.g. `-march=rv32imacxv5)` has been deprecated since AndeSight v5.3.0.

4. LLD and BFD Linkers

While LLD is mostly a drop-in replacement for BFD, it has some limitations on interpretation of linker scripts and may behave distinctly from BFD sometimes. The details are as follows:

- LLD may leave a large gap between sections with large offsets. For example, given a linker script with sections like below,

  ```
  …
  SECTIONS {
      . = 0x00000000;
      .text1 : { *(.text1) }
      . = 0x10000000;
      .text2 : { *(.text2) }
  }
  …
  ```

  LLD will produce an ELF file with a size of at least 0x10000000 bytes as it only generates a single segment for the whole program. This may cause the linking to fail if the offset is too large. To fix the problem, just specify the LMA of each section manually, such as the following:

  ```
  …
  SECTIONS {
      . = 0x00000000;
      .text1 : AT(ADDR(.text1)) { *(.text1) }
      . = 0x10000000;
      .text2 : AT(ADDR(.text2)) { *(.text2) } }
  …
  ```

  The presence of `AT` expressions in the above example will lead LLD to generate a separate segment for each section.

- LLD has a different semantic interpretation for the command "`. = <NUMBER>`" inside an input section description from that of BFD. In particular, given a linker script like below,

  ```
  …
  .foo 0x10000 : {
      . = 0x100;
  }
  …
  ```

LLD will have the location counter (the dot "`.`") set back to the absolute address `0x100` rather than an offset to the VMA of the output section (i.e., `.foo + 0x100`). If you encounter this error with LLD, just use the command "`. = ADDR(SECTION) + OFFSET`" (e.g., "`. = ADDR(.foo) + 0x100`") instead to set the location counter correctly.

- Linking large programs can take a long time. With LLD, you can specify the option "`-Wl,--threads`" in `LDFLAGS` to enable the multi-threaded linking and reduce the linking time.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

5. V5 assembly language

This chapter is intended to provide an outline and some hints for V5 assembly language. For more details about assembly programming, see *The RISC-V Instruction Set Manual - Volume I: User-Level ISA, Version 2.2*, demo code in the package, as well as *Using as* (GNU Assembly Manual) and *RISC-V Assembly Programmer's Manual*.

5.1. General syntax

Use "`#`" anywhere in the line except inside quotes. Start a comment at the end of line.

Multiple instructions in a line are allowed though not recommended and should be separated by "`;`".

An integer can be specified in decimal, octal (prefixed with `0`), hexadecimal (prefixed with `0x`), or binary (prefixed with `0b`) format. For example, `128`, `0200`, `0x80`, and `0b10000000` are all identical.

A floating number uses "`e`" and "`E`" to for exponential portion, "`f`" and "`F`" for single precision floating point constant, and "`d`" and "`D`" for double precision floating point constant; for example, `0f12.345` or `0d1.2345e12`.

Assembler is not case-sensitive in general except user defined labels. For example, "`jalr F1`" is different from "`jalr f1`" but the same as "`JALR F1`".

5.2. Registers

See *The RISC-V Instruction Set Manual - Volume I: User-Level ISA, Version 2.2* for detailed information.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

5.3. Pseudo-ops

Pseudo-ops (Pseudo-operations), also called assembler instructions or assembler directives are instructions to the assembler that does not generate any machine code. This section lists standard pseudo-ops and Andes-specific pseudo-ops supported in AndeStar V5.

5.3.1. Standard pseudo-ops

Andes supports standard assembler directives, including GNU .-prefixed and RISC-V-specific options, as shown in Table 5 below.

Table 5. Supported standard pseudo ops

Directive	Arguments	Description
`.align`	`INTEGER`	align to power of 2 (alias for `.p2align`, which is preferred.)
`.p2align`	`P2,[PAD_VAL=0],MAX`	align to power of 2
`.balign`	`B,[PAD_VAL=0]`	byte align
`.file`	`"FILENAME"`	emit `FILENAME` to symbol table
`.globl`	`SYMBOL_NAME`	emit `SYMBOL_NAME` to symbol table (scope GLOBAL)
`.local`	`SYMBOL_NAME`	emit `SYMBOL_NAME` to symbol table (scope LOCAL)
`.comm`	`SYMBOL_NAME,SIZE,ALIGN`	emit common object to .bss section
`.common`	`SYMBOL_NAME,SIZE,ALIGN`	emit common object to .bss section
`.ident`	`"STRING"`	accepted for source compatibility
`.section`	`[{.text,.data,.rodata,.bss}]`	emit section (if not present, default .text) and make it current
`.size`	`SYMBOL, SYMBOL`	accepted for source compatibility
`.text`		emit .text section (if not present) and make it current
`.data`		emit .data section (if not present) and make it current
`.rodata`		emit .rodata section (if not present) and make it current

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Directive	Arguments	Description
.bss		emit .bss section (if not present) and make current
.string	"STRING"	emit STRING
.asciz	"STRING"	emit STRING (alias for .string)
.equ	NAME, VALUE	constant definition
.macro	NAME ARG1 [, ARGN]	begin macro definition
.endm		end macro definition
.type	SYMBOL, @FUNCTION	accepted for source compatibility
.option	{pic,nopic,relax,norelax,push,pop}	RISC-V options. See the bullet item .option below for a more detailed description.
.byte	EXPRESSION [, EXPRESSION]*	8-bit comma-separated words
.2byte	EXPRESSION [, EXPRESSION]*	16-bit comma-separated words
.half	EXPRESSION [, EXPRESSION]*	16-bit comma-separated words
.short	EXPRESSION [, EXPRESSION]*	16-bit comma-separated words
.4byte	EXPRESSION [, EXPRESSION]*	32-bit comma-separated words
.word	EXPRESSION [, EXPRESSION]*	32-bit comma-separated words
.long	EXPRESSION [, EXPRESSION]*	32-bit comma-separated words
.8byte	EXPRESSION [, EXPRESSION]*	64-bit comma-separated words
.dword	EXPRESSION [, EXPRESSION]*	64-bit comma-separated words
.quad	EXPRESSION [, EXPRESSION]*	64-bit comma-separated words
.float	EXPRESSION [, EXPRESSION]*	32-bit floating point values
.double	EXPRESSION [, EXPRESSION]*	64-bit floating point values
.quad	EXPRESSION [, EXPRESSION]*	128-bit floating point values
.dtprelword	EXPRESSION [, EXPRESSION]*	32-bit thread local word
.dtpreldword	EXPRESSION [, EXPRESSION]*	64-bit thread local word
.sleb128	EXPRESSION	signed little endian base 128, DWARF
.uleb128	EXPRESSION	unsigned little endian base 128, DWARF

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Directive	Arguments	Description
.zero	INTEGER	zero bytes
.variant_cc	SYMBOL_NAME	annotate the symbol with variant calling convention
.attribute	NAME, VALUE	RISC-V object attributes. For more detailed descriptions, see the bullet item .attribute below.

■ .attribute

The .attribute directive is used to record information about an object file/binary that a linker or runtime loader needs to check for compatibility.

.attribute take two arguments. The first argument of .attribute is the symbolic name of the attribute or the attribute number, where the prefix Tag_RISCV_ can be omitted, and the second argument can be a string or number.

The syntax for .attribute is as follows.

```
.attribute <NAME_OR_NUMBER>, <ATTRIBUTE_VALUE>

NAME_OR_NUMBER  := <ATTRIBUTE-NAME>
                 | [1-9][0-9]*

ATTRIBUTE_VALUE := <STRING>
                 | <NUMBER>
```

■ .option

- pic/nopic

  This option sets the code model to PIC (position independent code) or non-PIC.

- relax/norelax

  This option enables/disables linker relaxation for the following code region.

NOTE

1. A code region followed by .option relax will emit R_RISCV_RELAX/ R_RISCV_ALIGN even if the linker does not support relaxation.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

2. The recommended way to disable linker relaxation of specific code region is using `.option push`, `.option norelax` and `.option pop`. This can prevent linker relaxation from being enabled accidentally if you have disabled linker relaxation.

- `push/pop`
  This option pushes/pops current options to/from the options stack.

5.3.2. Andes pseudo-ops

`.no_execit_begin`	marks the start of the region where the EXEC.IT optimization must be disabled in the linker.
`.no_execit_end`	marks the end of the region where the EXEC.IT optimization must be disabled in the linker.
`.innermost_loop_begin`	marks the start of the innermost loop.
`.innermost_loop_end`	marks the end of the innermost loop.
`.option execit`	to enable the EXEC.IT optimization.

The following pseudo-ops mark the beginning of specific code regions. These code regions need to be enclosed by `.option push` and `.option pop`.

`.option no16bit`	marks the start of the region where 16-bit instructions are not included.
`.option notablejump`	marks the start of the region where tablejump (Zcmt) instructions are not included.
`.option noexecit`	marks the start of the region where the EXEC.IT optimization must be disabled in the linker.
`.option innermostloop`	marks the start of the innermost loop.

Among the above directives, `.option no16bit` serves the same purpose as the previous `.option norvc` in earlier AndeSight releases, which also stopped generating 16-bit instructions.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

5.4. Pseudo instructions

In addition to hardware instructions, there are many software instructions defined to make assembly programming much easier. These are pseudo instructions. Andes supports standard RISC-V pseudo instructions and those for accessing control and status registers, as listed in Table 6 and Table 7 below.

Table 6. Supported standard pseudo instructions

Pseudo Instruction	Base Instruction(s)	Meaning	Comment
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address	With `.option nopic` (Default)
`la rd, symbol`	`auipc rd, symbol@GOT[31:12]` `l{w|d} rd, symbol@GOT[11:0](rd)`	Load address	With `.option pic`
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address	
`lga rd, symbol`	`auipc rd, symbol@GOT[31:12]` `l{w|d} rd, symbol@GOT[11:0](rd)`	Load global address	
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global	
`l{bu|hu|wu} rd, symbol`	`auipc rd, symbol[31:12]` `l{bu|hu|wu} rd, symbol[11:0](rd)`	Load global, unsigned	
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global	

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Pseudo Instruction	Base Instruction(s)	Meaning	Comment
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global	
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global	
`nop`	`addi x0, x0, 0`	No operation	
`li rd, immediate`	Myriad sequences	Load immediate	The assembler can generate different instruction sequences to load a specific numeric value into a register.
`mv rd, rs`	`addi rd, rs, 0`	Copy register	
`not rd, rs`	`xori rd, rs, -1`	Ones' complement	
`neg rd, rs`	`sub rd, x0, rs`	Two's complement	
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word	
`sext.b rd, rs`	`slli rd, rs, XLEN - 8` `srai rd, rd, XLEN - 8`	Sign extend byte	This is a single instruction when the `Zbb` extension is available.
`sext.h rd, rs`	`slli rd, rs, XLEN - 16` `srai rd, rd, XLEN - 16`	Sign extend halfword	This is a single instruction when the `Zbb` extension is available.
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word	
`zext.b rd, rs`	`andi rd, rs, 255`	Zero extend byte	

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Pseudo Instruction	Base Instruction(s)	Meaning	Comment
`zext.h rd, rs`	`slli rd, rs, XLEN - 16` `srli rd, rd, XLEN - 16`	Zero extend halfword	This is a single instruction when the `Zbb` extension is available.
`zext.w rd, rs`	`slli rd, rs, XLEN - 32` `srli rd, rd, XLEN - 32`	Zero extend word	This is a single instruction when the `Zba` extension is available.
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero	
`snez rd, rs`	`sltu rd, x0, rs`	Set if != zero	
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero	
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero	
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register	
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value	
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate	
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register	
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value	
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate	
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero	
`bnez rs, offset`	`bne rs, x0, offset`	Branch if != zero	
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero	
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero	
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero	

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Pseudo Instruction	Base Instruction(s)	Meaning	Comment
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero	
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >	
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤	
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned	
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned	
`j offset`	`jal x0, offset`	Jump	
`jal offset`	`jal x1, offset`	Jump and link	
`jr rs`	`jalr x0, rs, 0`	Jump register	
`jalr rs`	`jalr x1, rs, 0`	Jump and link register	
`ret`	`jalr x0, x1, 0`	Return from subroutine	
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine	
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine Tail call far-away subroutine	It will use `x7` as scratch register when the `Zicfilp` extension is available.
`fence`	`fence iorw, iorw`	Fence on all memory and I/O	
`pause`	`fence w, 0`	PAUSE hint	

Table 7. Supported pseudo instructions for accessing control and status registers

Pseudo Instruction	**Base Instruction(s)**	**Meaning**
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Pseudo Instruction	Base Instruction(s)	Meaning
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`fsrmi rd, imm`	`csrrwi rd, frm, imm`	Swap FP rounding mode, immediate
`fsrmi imm`	`csrrwi x0, frm, imm`	Write FP rounding mode, immediate
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags
`fsflagsi rd, imm`	`csrrwi rd, fflags, imm`	Swap FP exception flags, immediate
`fsflagsi imm`	`csrrwi x0, fflags, imm`	Write FP exception flags, immediate

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

5.5. Macros

5.5.1. Creating macros in assembly code

When writing assembly code, you can define macros to generate assembly outputs. This is an efficient way to repeat similar statements or simplify varying syntax for complicated conditions. For example, the below definition specifies a macro "`sum`" to put a sequence of numbers into memory:

```
.macro sum from,to
  .long \from
.if \to-\from
  sum "(\from+1)",\to
.endif
.endm
```

With that definition, "`sum 0,5`" is equivalent to this assembly code fragment:

```
.long 0
.long 1
.long 2
.long 3
.long 4
.long 5
```

5.5.2. Assembler directives for macros

The directives `.macro` and `.endm` allow you to define macros. The following descriptions give the basic usages. For more details and other directives, see GNU Assembly Manual *Using as*.

`.macro` *`macname`*
`.macro` *`macname macargs ...`*

Begin the definition of a macro called *`macname`*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with "*`=deflt`*". For example, these are valid `.macro` statements:

- `.macro comm`
  Begin the definition of a macro called `comm`, which takes no arguments.
- `.macro plus1 p, p1`
- `.macro plus1 p p1`
  Either statement begins the definition of a macro called `plus1`, which takes two

arguments; if you want to use arguments within the macro definition, you have to use "\" character as its prefix. In this case, use "\p" or "\p1" to evaluate the arguments.

- `.macro reserve_str p1=0 p2`

  Begin the definition of a macro called `reserve_str`, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as "`reserve_str` *a*,*b*" (with "\p1" evaluating to *a* and "\p2" evaluating to *b*), or as "`reserve_str ,`*b*" (with "\p1" evaluating as the default, which is "0" in this case, and "\p2" evaluating to *b*).

`.endm`

Mark the end of a macro definition.

6. Machine instructions

6.1. 32/16-bit

Full machine instructions, 32-bit and/or 16-bit, can be specified by programmers directly. They can be mixed without any restrictions. By default compiler generates 32/16-bit mixed instructions, but you can apply a compiler option `-mno-16-bit` to generate pure 32-bit instructions.

In general, instructions may get converted into corresponding 16/32-bit version depending on the optimization level of the compiler:

1. When `-O0` or `-Os` is specified, a 32-bit instruction will get converted into its 16-bit version whenever possible.
2. When `-On` (`n=1-3`), `-Og` or `-Ofast` is specified, a 16-bit instruction may get converted back to its 32-bit version to fulfill alignment requirement.

6.2. Endianness

AndeStar V5 only supports little endian data storage and instructions.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7. Application Binary Interface (ABI)

The AndeStar V5 architecture ABI defines the interface for compiled programs and assembled programs to work jointly on AndeStar V5 architecture while ensuring high performance and binary compatibility. This ABI is closely aligned with the official RISC-V processor-specific ABI (psABI). For detailed and up-to-date information about the ABI, see the official RISC-V psABI documentation.

This chapter provides simplified descriptions of the AndesStar V5 ABI, along with a few sample code and stack layout illustrations, to help you grasp the fundamentals without needing to digest the full official psABI documentation. Specifically, Section 7.1 introduces the supported data types in programming and their representation on AndeStar V5 architecture, and Section 7.2 gives the details of the AndeStar V5 ABI types.

Note that the code generation examples in this chapter may vary slightly depending on compiler implementations and optimizations. However, the overall layout will remain consistent across implementations.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7.1. Data types

7.1.1. Byte ordering

The byte ordering defines how the bytes that make up multi-byte data type are ordered in memory. AndeStar V5 architecture ABI now supports only little-endian byte ordering.

- Little-endian: The least significant byte of a data is stored at the lowest memory address.
- Big-endian: The least significant byte of a data is stored at the highest memory address.

7.1.2. Primitive data types

Table 8. Size and byte alignment of primitive data types

Class	Machine Type	RV32		RV64	
		Size (in Byte)	Alignment (in Byte)	Size (in Byte)	Alignment (in Byte)
Integer	Unsigned byte	1	1	1	1
	Signed byte	1	1	1	1
	Unsigned halfword	2	2	2	2
	Signed halfword	2	2	2	2
	Unsigned word	4	4	4	4
	Signed word	4	4	4	4
	Unsigned doubleword	8	8	8	8
	Signed doubleword	8	8	8	8
Floating Point	Single precision (IEEE 754)	4	4	4	4
	Double precision (IEEE 754)	8	8	8	8
Pointer	Instruction pointer	4	4	8	8
	Data pointer	4	4	8	8

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7.1.3. Composite data types

Composite Data Types is a collection of primitive data types and other composite data types that can be used to construct a program.

7.1.3.1 Array type

Array Type is a sequence of homogenous data elements (i.e. of the same primitive data type). The alignment of an array is determined by the alignment of its elements' data type. The size of an array is the multiplication of the size of its data type and the number of its elements.

7.1.3.2 Aggregate and union type

An aggregate is a data type that data elements are laid out sequentially in memory. A union is a data type that stores each of its elements at the same memory address at different times.

The alignment of an aggregate or a union is equal to the alignment of its most-aligned component. The size of an aggregate is the smallest multiple of its alignment that is sufficient to hold all of its elements when they are laid out. The size of a union is the smallest multiple of its alignment that is sufficient to hold the union's largest element.

7.1.3.3 Bit-field type

A bit-field is a member of an aggregate or union which defines an integral object with specified of bits. The layout of bit-fields within an aggregate is defined by the appropriate language binding. When there are unused portions of such a member that are sufficient for the following member to start at its natural alignment, the following member can use the unallocated portions.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7.1.4. C language mapping of Andes platform

Table 9. Mapping of C primitive data types

C/C++ Type	Machine Type	
	RV32	RV64
singed char	Signed byte	
[unsigned] char	Unsigned byte	
[signed] short	Signed halfword	
unsigned short	Unsigned halfword	
[signed] int	Signed word	
unsigned int	Unsigned word	
[signed] long	Signed word	Signed doubleword
unsigned long	Unsigned word	Unsigned doubleword
[signed] long long	Signed doubleword	
unsigned long long	Unsigned doubleword	
size_t	Unsigned word	Unsigned doubleword
float	Single precision (IEEE 754)	
double	Double precision (IEEE 754)	
long double	Double precision (IEEE 754)	Two double precision (IEEE 754)
float _Complex	Two single precision (IEEE 754)	
double _Complex	Two double precision (IEEE 754)	
long double _Complex	Two double precision (IEEE 754)	

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7.2. Calling convention

AndeStar V5 follows the RISC-V calling conventions defined in the RISC-V psABI documentation for integer ABI, RV32E ABI, floating point ABI, and vector ABI.

The integer ABI is for v5 toolchains and the RV32E ABI is for v5e toolchains. The two ABIs use General Purpose Registers (GPRs) for computations on all primitive types. In contrast, the floating point ABI is for v5f/v5d toolchains, which use GPRs for computations on integer primitive types and Floating Point Registers (FPRs) for computations on floating point primitive types. In addition, the vector ABI is for computations on vector types when vector extension is enabled using the option `-mext-vector` or `-mext-vector=EXTENSION`.

XLEN and FLEN refer to the widths of GPR and FPR, respectively. XLEN is 32 for RV32 toolchains and 64 for RV64 toolchains. FLEN is undefined for v5/v5e toolchains, 32 for v5f toolchains, and 64 for v5d toolchains.

7.2.1. Integer ABI

7.2.1.1 Registers

There are 32 XLEN-bit General Purpose Registers (GPRs) for Andes V5 instruction set architecture. The GPRs can be roughly classified into caller-saved and callee-saved registers. The following table lists the GPRs with their ABI usage convention.

Table 10. Andes GPRs with integer ABI usage convention

Register	ABI name	Description	Saver
x0	zero	Hard-wired zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary register	Caller
x6	t1	Temporary register	Caller
x7	t2	Temporary register	Caller

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Register	ABI name	Description	Saver
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10	a0	Function arguments/return values	Caller
x11	a1	Function arguments/return values	Caller
x12	a2	Function arguments	Caller
x13	a3	Function arguments	Caller
x14	a4	Function arguments	Caller
x15	a5	Function arguments	Caller
x16	a6	Function arguments	Caller
x17	a7	Function arguments	Caller
x18	s2	Saved register	Callee
x19	s3	Saved register	Callee
x20	s4	Saved register	Callee
x21	s5	Saved register	Callee
x22	s6	Saved register	Callee
x23	s7	Saved register	Callee
x24	s8	Saved register	Callee
x25	s9	Saved register	Callee
x26	s10	Saved register	Callee
x27	s11	Saved register	Callee
x28	t3	Temporary register	Caller
x29	t4	Temporary register	Caller
x30	t5	Temporary register	Caller
x31	t6	Temporary register	Caller

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7.2.1.2 Stack frame

Stack frames are very important for calling a function. Whenever a caller invokes a callee, the return address is automatically saved in the `ra` register, and then a corresponding stack frame is created in memory to store local variables, spill registers, and pass arguments. The stack is full-descending and each stack frame of a function is managed by the frame pointer (`fp`) and stack pointer (`sp`) with 16-byte alignment. Figure 1 below illustrates stack frames for function calls in a V5 integer ABI scenario:

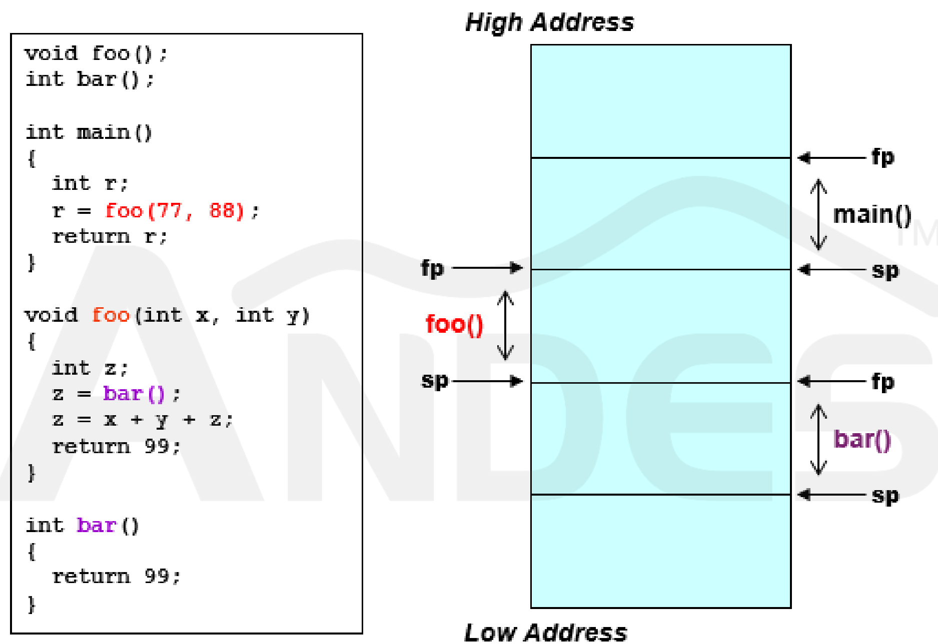


Figure 1. AndeStar V5 integer ABI stack frame scenario

Conceptually, the function prologue is responsible for constructing the stack frame, while the epilogue handles its destruction. The register `sp` is adjusted to create space for the stack frame, and the register `ra` is used to return to the caller after the callee finishes execution. If the compiler option `-fno-omit-frame-pointer` is applied, the register `fp` is also involved in stack frame creation, pointing to the base address of the stack frame.

The following illustrates the works in the prologue and epilogue, with the option `-fno-omit-frame-pointer` applied to show detailed stack frame information:

- Prologue
  1. Adjust the stack pointer (`sp`) to create a space for the stack frame.
  2. Push callee-saved registers into the stack, including the frame pointer (`fp`) and

return address (`ra`) if necessary.

3. Set the frame pointer (`fp`) to the base address of the current stack frame.

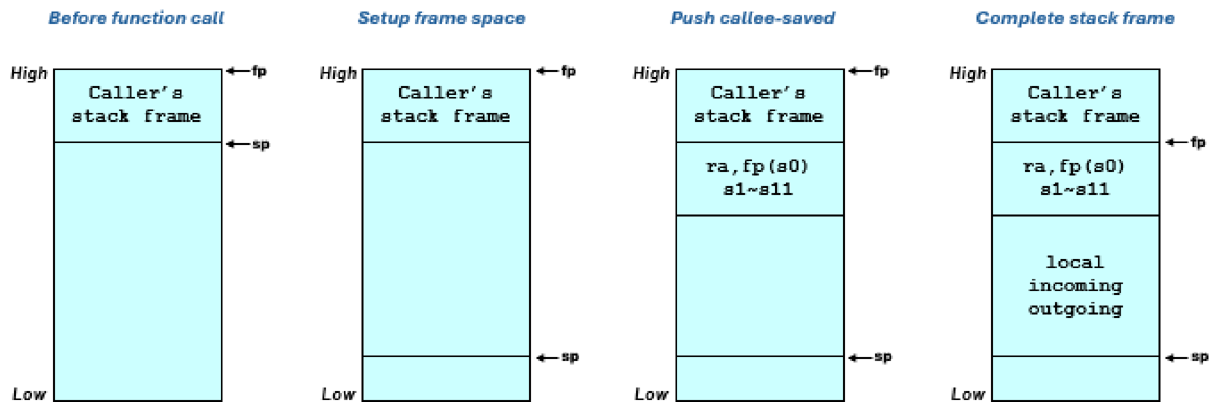


Figure 2. Function prologue for stack frame construction

- Epilogue
  1. Restore callee-saved, frame pointer (`fp`), and return address (`ra`) registers.
  2. Destroy the current stack frame by resetting the stack pointer (`sp`) to the location of the caller's stack frame.
  3. Use the return address (`ra`) to return to the caller.

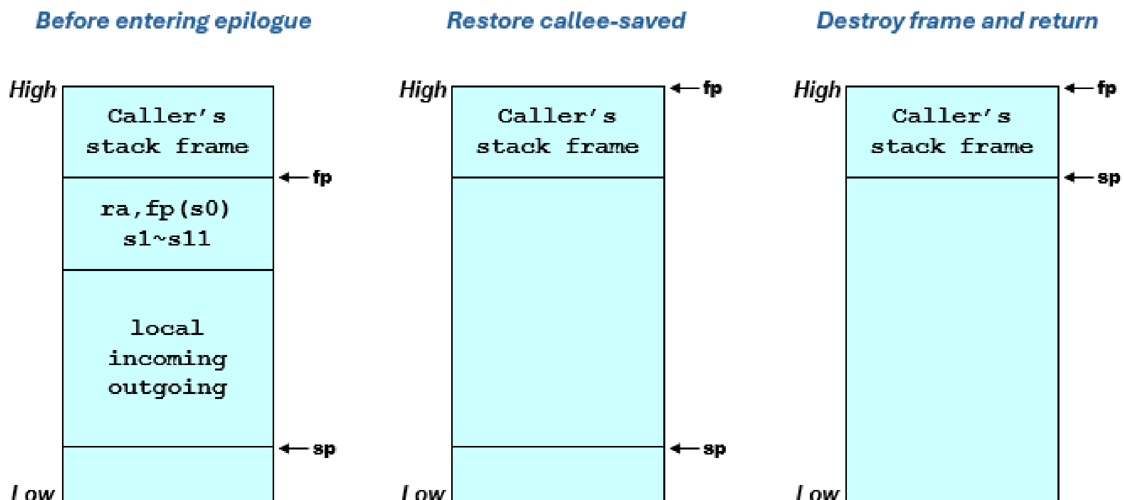


Figure 3. Function epilogue for stack frame destruction

In principle, every stack frame is composed of four blocks: callee-saved, local variables, incoming arguments, and outgoing arguments. Each block must be aligned to 16 bytes and thus may require padding bytes within the block. Note that to conform to C language

standards, the padding bytes in the outgoing arguments block are aligned in a direction opposite to those of other blocks. See Figure 4 for the memory layout of these four blocks within a V5 ABI stack frame.

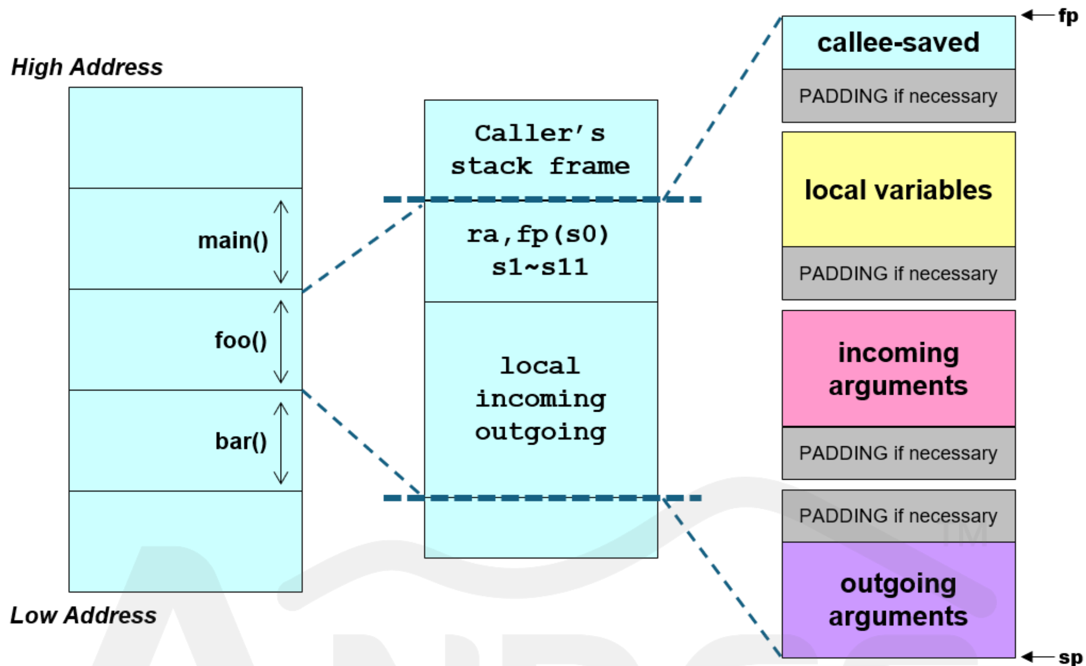


Figure 4. V5 ABI stack frame layout

The following explains the purpose of each memory block in the stack frame and the specific data each block contains:

- **Callee-saved:** When some callee-saved register are used in the callee function, this block is created to contain their original values, ensuring that they can be restored when the function returns to the caller.
- **Local variables**: This block saves local values for the callee function, including local variables declared in the function, spilled register values, and temporary calculated values, when necessary.
- **Incoming arguments**: According to Table 10, incoming function arguments are passed by registers a0 - a7. Typically, the arguments are accessed via registers directly and there is no need to allocate memory space for them. However, when the compilation flag -O0 is used, incoming argument values may be duplicated, leading to the creation of this memory block to store the values.
- **Outgoing arguments**: This block is necessary when the callee function invokes other functions with arguments and the registers a0 - a7 are insufficient to pass all the outgoing arguments.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7.2.1.3 Examples of V5 Integer ABI

The four blocks within the stack frame of a function are not always clearly present in the assembly code after compiler optimizations are applied to programs. In fact, each block only appears if specific criteria are met. Figure 5 below illustrates the four blocks within a stack frame, represented by four different colored rectangles in the middle, with their presence requirement on the left and memory layout on the right. To satisfy the 16-byte alignment requirement, each block consists of four 4-byte slots (16 bytes total) in the figure.

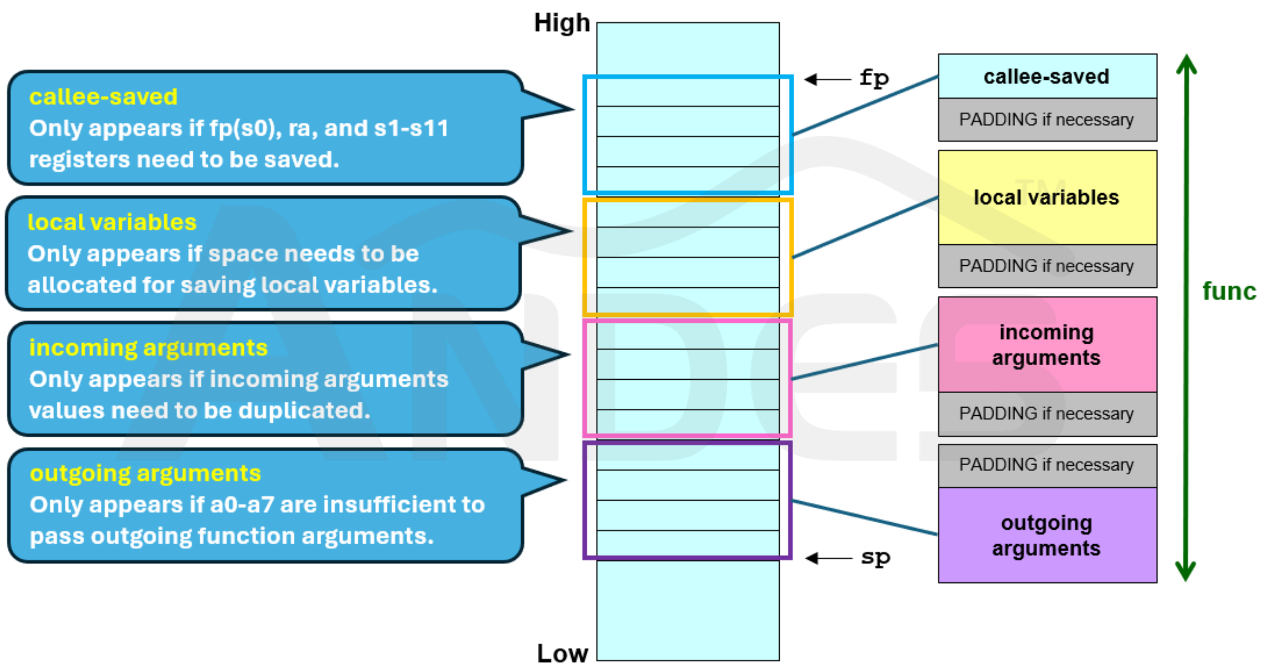


Figure 5. Blocks in a stack frame

This section provides three C code fragments to demonstrate stack frames with distinct memory layouts generated by the compiler. These examples, all compiled with the compiler options "`-O0 -fno-omit-frame-pointer`", provide detailed information about the stack frame layouts and allow investigation into associated blocks and the placement of variables within them, as illustrated and explained below.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Example 1: A simple function stack frame

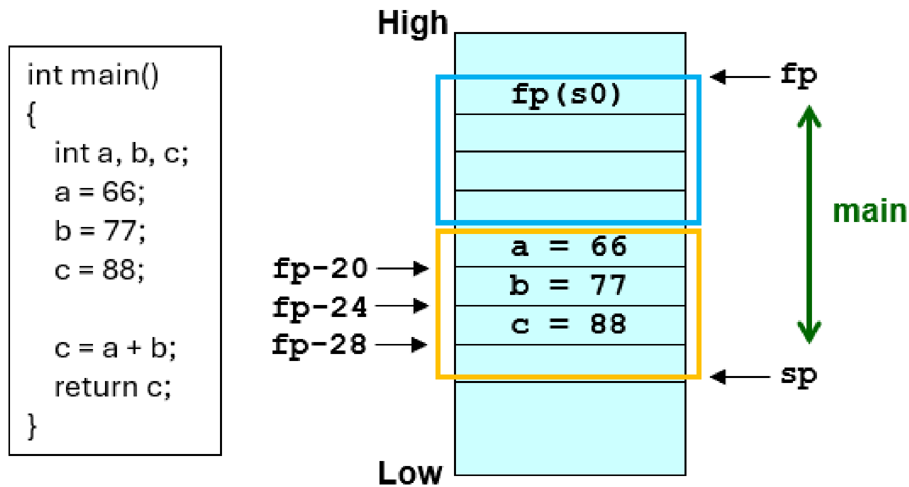


Figure 6. ABI example of simple function stack frame

- Callee-saved: This block only saves the frame pointer (`fp`) because this is a leaf function. In addition, it requires 12 padding bytes to meet the 16-byte alignment requirement.
- Local variables: This block stores the values of local variables and requires 4 padding bytes to meet the alignment requirement.
- Incoming arguments: This block is unnecessary because there is no incoming arguments.
- Outgoing arguments: This block is unnecessary because there is no outgoing arguments.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Example 2: Calling a function that duplicates incoming arguments

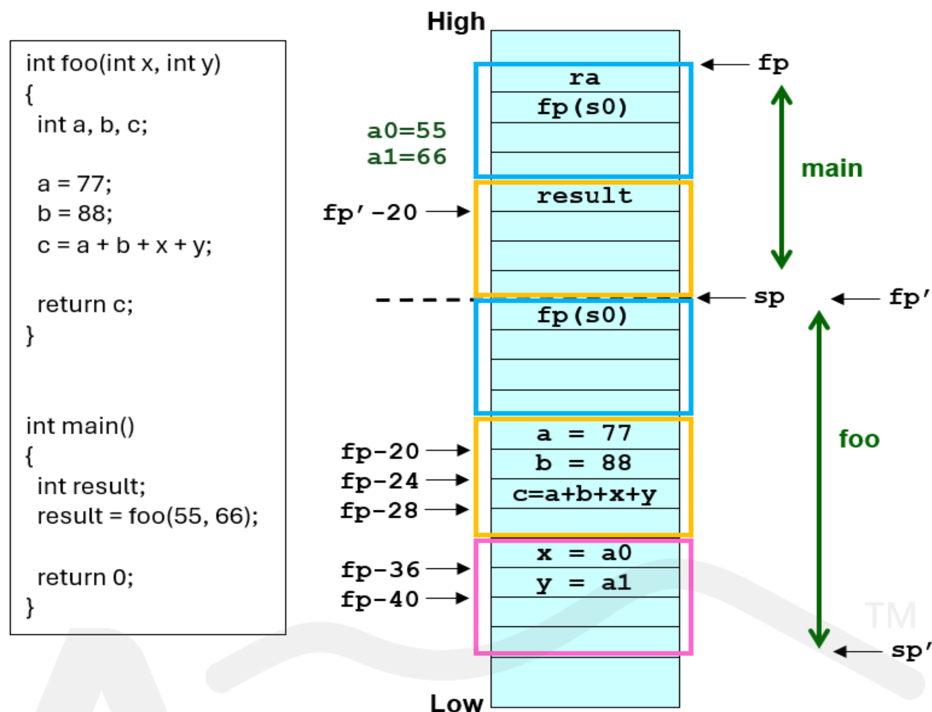


Figure 7. ABI example of calling a function with arguments

- Stack frame of `main()`
  - Callee-saved: For calling `foo(55, 66)`, the `main()` function saves the original return address (`ra`) and frame pointer (`fp`) in this block.
  - Local variables: The local variable `result` is stored in this block.
  - Incoming arguments: This block is unnecessary because there is no incoming arguments.
  - Outgoing arguments: This block is unnecessary because the registers `a0` and `a1` are sufficient to pass the arguments `55` and `66` for calling `foo()`.

- Stack frame of `foo()`
  - Callee-saved: This block only saves the frame pointer (`fp`) because this is a leaf function. In addition, it requires 12 padding bytes to meet the 16-byte alignment requirement.
  - Local variables: This block stores the values of local variables and requires 4 padding bytes to meet the alignment requirement.
  - Incoming arguments: This block is created to save incoming arguments. It stores the duplicated values of registers `a0` and `a1` and requires 8 padding

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

bytes to meet the alignment requirement.

- Outgoing arguments: This block is unnecessary because there is no outgoing arguments.

Example 3: Calling a function with outgoing arguments

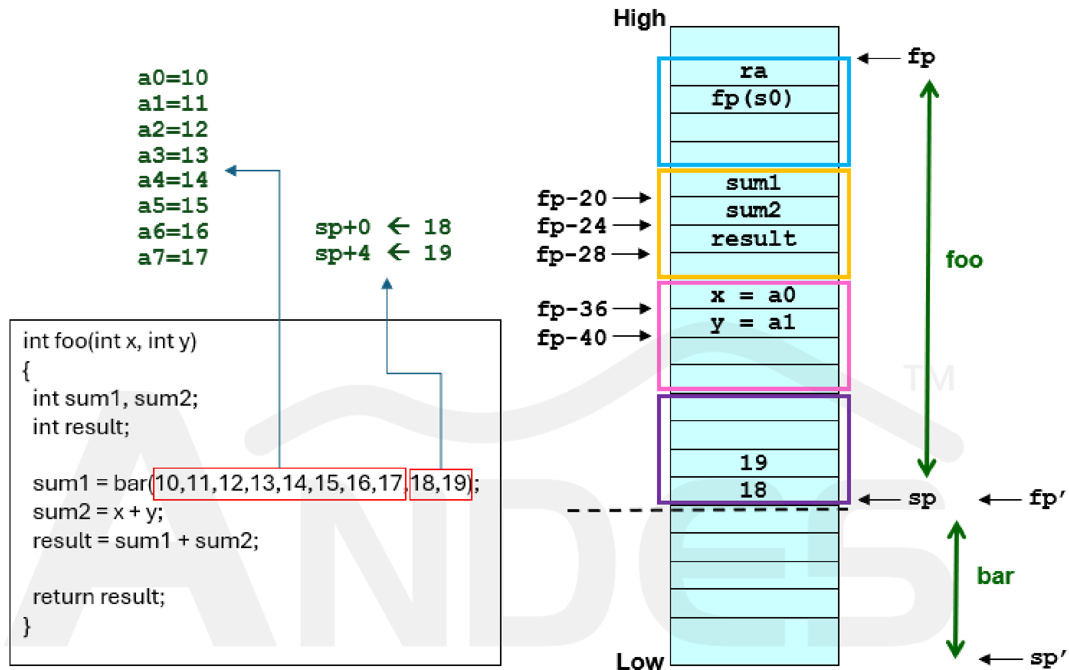


Figure 8. ABI example of outgoing arguments in stack frame

- Callee-saved: This block saves the return address (`ra`) and the frame pointer (`fp`). In addition, it requires 8 padding bytes to meet the alignment requirement.
- Local variables: This block stores the local variables `sum1`, `sum2`, and `result` and requires 4 padding bytes to meet the alignment requirement.
- Incoming arguments: This block stores the duplicated values of register `a0` and `a1` and requires 8 padding bytes for the alignment requirement.
- Outgoing arguments: This function calls `bar()`by passing ten arguments. The registers `a0-a7` are only able to hold the first eight argument values. Hence, this outgoing arguments block is created to store the rest two argument values `18` and `19` so that they can be accessed by the frame pointer register (`fp'`) after entering the `bar()` function. Besides, this block requires 8 padding bytes in a reverse direction to meet the alignment requirement.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7.2.1.4 Argument passing and return

Arguments are passed in GPRs and stack. The space of stacked arguments (i.e., the outgoing arguments block in a stack frame) must be allocated by the caller. The argument passing obeys the following rules:

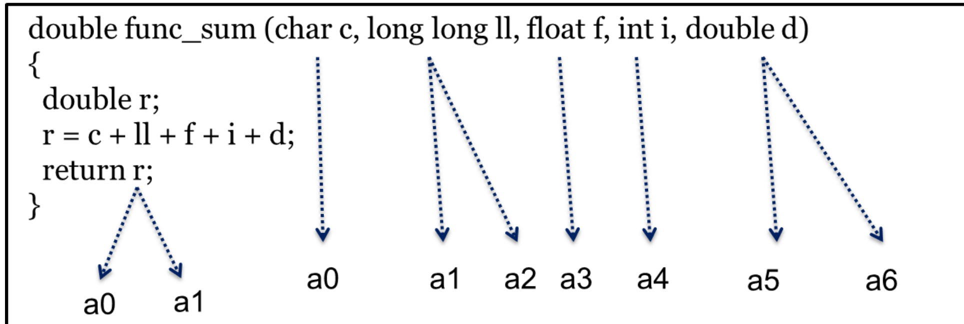
- XLEN is 32 for RV32 and 64 for RV64.
- GPRs `a0~a7` are used to pass arguments.
- If the argument is a primitive type smaller than XLEN bits, it will be zero- or sign-extended to XLEN bits.
- If GPRs `a0~a7` are not sufficient to hold all arguments, the remaining ones will be passed in the outgoing arguments block of caller's stack frame. Then the callee is able to retrieve them using `fp` or `sp` with offset calculation.
- If the argument is a composite type with a size that is not XLEN bits aligned, it will be rounded up to the closest multiple of XLEN bits.
- An argument that is not a primitive type can be assigned to both registers and the stack. In this case, the first part of the argument is copied to the GPRs and the rest part of it to the stack.
- For a variadic function, the caller passes arguments like a normal function using GPRs and stack; the callee is responsible for pushing argument registers into stack so that all the nameless arguments can be consecutive in the memory for accessing. The callee must create an extra block of 16-byte alignment to store the nameless arguments that are passed via GPRs.

The function return value is determined by the type of the result:

- If the result is a primitive type,
  1. For a primitive type smaller than XLEN bits, the return value is zero- or sign-extended to XLEN bits and returned in `a0`.
  2. For a XLEN-bit primitive type, the return value is returned in `a0`.
  3. For a 2*XLEN-bit primitive type, the return value is returned in `a0` and `a1`.
- If the result is a composite type,
  1. For a size not larger than 2*XLEN bits, the return value follows the same rules as when the result is a primitive type.
  2. For a size larger than 2* XLEN bits or undetermined by the caller and callee, the return value is returned at a memory reference that is passed as an extra argument when the function is called. In that case, the address for the result will

be placed in `a0` and the first argument will be passed in `a1`.

Here is an example of how arguments are passed and how value is returned.



7.2.2. RV32E ABI

There are 16 XLEN-bit General Purpose Registers (GPRs) for Andes V5E instruction set architecture. This calling convention is the same as that of the integer ABI, except for the following:

- The stack pointer must be aligned to a 32-bit boundary.
- Registers `x16-x31` do not participate in the RV32E ABI. Among the 16 GPRs in use (i.e., `x0-x15`), there are only six argument registers, `a0-a5`, only two callee-saved registers, `s0-s1`, and only three temporaries, `t0-t2`.

Table 11. Andes GPRs with RV32E ABI usage convention

Register	ABI name	Description	Saver
x0	zero	Hard-wired zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary register	Caller
x6	t1	Temporary register	Caller
x7	t2	Temporary register	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10	a0	Function arguments/return values	Caller
x11	a1	Function arguments/return values	Caller
x12	a2	Function arguments	Caller
x13	a3	Function arguments	Caller
x14	a4	Function arguments	Caller
x15	a5	Function arguments	Caller

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

7.2.3. Floating point ABI

7.2.3.1 Registers

There are 32 XLEN-bit General Purpose Registers (GPRs) and 32 FLEN-bit Floating Point Registers (FPRs) for Andes V5F/V5D instruction set architecture. Both the GPRs and FPRs can be roughly classified into caller-saved and callee-saved registers. The GPRs and their ABI usage convention are summarized in Table 10. For the FPRs and their ABI usages, see the following table.

Table 12. Andes FPRs with their ABI usage convention

Register	ABI name	Description	Saver
f0-f7	ft0-ft7	Temporary registers	Caller
f8-f9	fs0-fs1	Callee-saved registers	Callee
f10-f17	fa0-fa7	Argument registers	Caller
f18-f27	fs2-fs11	Callee-saved registers	Callee
f28-f31	ft8-ft11	Temporary registers	Caller

7.2.3.2 Stack frame

The stack frames in the floating point ABI are almost the same as those in the integer ABI, with additional considerations for floating point registers following similar policy. See Section 7.2.1.2 for detailed information.

7.2.3.3 Argument passing and return

Arguments are passed in GPRs, FPRs and stack. The space of stacked arguments (i.e., the outgoing arguments block in a stack frame) must be allocated by a caller. The argument passing obeys the following rules:

- XLEN is 32 for RV32 toolchains and 64 for RV64 toolchains whereas FLEN is 32 for v5f toolchains and 64 for v5d toolchains.
- GPRs a0~a7 and FPRs fa0~fa7 are used to pass arguments.
- If the argument is a primitive type smaller than XLEN bits, it will be zero- or sign-extended to XLEN bits.
- If GPRs a0~a7 and FPRs fa0~fa7 are not sufficient to hold all arguments, the remaining ones will be passed in the outgoing arguments block of the caller's stack

frame. Then the callee is able to retrieve them using `fp` or `sp` with offset calculation.

- If the argument is a composite type with a size that is not XLEN bits aligned, it will be rounded up to the closest multiple of XLEN bits.
- An argument that is not a primitive type can be assigned to both registers and the stack. In this case, the first part of the argument is copied to the GPRs and the rest part of it to the stack.
- A floating-point argument that is no more than FLEN bits wide is passed in a floating-point argument register if there is one available.
- A struct containing just one floating-point value is passed as though it were a standalone floating-point real.
- A struct containing two floating-point values that are no more than FLEN bits wide is passed in two floating-point registers if there are available ones. Otherwise, it is passed according to the integer calling convention. Note that the two registers need not to be an aligned pair.
- A complex floating-point number or a struct containing just one complex floating-point number is passed as though it were a struct containing two floating-point reals.
- A struct containing one floating-point no more than FLEN bits wide and one integer no more than XLEN bits wide, in either order, is passed in a floating-point register and an integer register if there are available ones, with the integer zero- or sign-extended as though it were a scalar. Otherwise, it is passed according to the integer calling convention.
- For a variadic function, the caller passes arguments like a normal function using GPRs, FPRs, and stack; the callee is responsible for pushing argument registers into stack so that all the nameless arguments can be consecutive in the memory for accessing. The callee must create an extra block of 16-byte alignment to store the nameless arguments that are passed via GPRs and FPRs.

The function return value is determined by the type of the result:

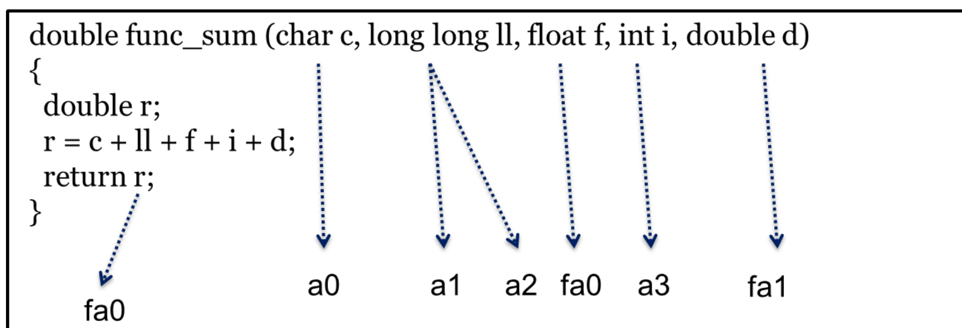
- If the result is a primitive type,
  1. For a non-floating-point primitive type smaller than XLEN bits, the return value is zero- or sign-extended to XLEN bits and returned in `a0`.
  2. For a XLEN-bit non-floating-point primitive type, the return value is returned in `a0`.
  3. For a 2*XLEN-bit non-floating-point primitive type, the return value is returned in `a0` and `a1`.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

4. For a floating-point primitive type less than or equal to FLEN bits, the return value is returned in `fa0`.
5. For a floating-point primitive type larger than FLEN bits, the return value is returned in `fa0` and `fa1` if possible. Otherwise, the return value will be returned in memory and the address for the result will be placed in `fa0`.

- If the result is a composite type,
  1. For a struct containing one or two floating-point values smaller than FLEN bits, the return value is returned in `fa0` and `fa1`.
  2. For a complex floating-point number, it is treated as a struct containing two elements.
  3. For a size no larger than 2*XLEN bits, the return value follows the same rules as when the result is a primitive type.
  4. For a size larger than 2* XLEN bits or undetermined by the caller and callee, the return value is returned at a memory reference that is passed as an extra argument when the function is called. In that case, the address for the result will be placed in `a0` and the first argument will be passed in `a1`.

Here is an example of how arguments are passed and how value is returned in RV32 v5d toolchains.



7.2.4. Vector ABI

AndeStar V5 has standard and variant calling conventions for the vector ABI. The standard calling convention treats all vector registers as caller-saved registers, whereas the variant calling convention classifies vector registers into caller-saved and callee-saved and uses the caller-saved ones to pass function arguments and return values. For more information about

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

the two calling conventions, see the official RISC-V psABI documentation.

Note that the variant calling convention is not supported until AndeSight v5.4.0 release and the support is limited to GCC, not LLVM. Starting from AndeSight v5.4.0, functions that pass arguments or return values via vector registers must adhere to the variant calling convention. Due to the differences in RVV calling conventions, ensure that objects compiled using AndeSight v5.4.0 toolchains and GCC are not linked with those compiled using earlier toolchains or LLVM.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8. AndeStar V5 predefined macros

AndeStar V5 architecture follows RISC-V C API specification with additional custom macros to support Andes-specific extensions. These custom predefined macros are listed in Table 13 below. For a detailed list of RISC-V preprocessor definitions, see the official documentation "RISC-V C API".

Table 13. AndeStar custom predefined macros

Macro Name	Value	Description
`__ANDES`	1	This macro is defined when toolchains are provided by Andes Technology.
`__nds_bf16`	1	The macro is defined if the Andes BLOAT16 conversion extension is available.
`__nds_bf16ms`	1	The macro is defined if the Andes Mode Switch BFLOAT16 extension is available.
`__nds_execit`	1	The macro is defined if the architecture supports AndeStar EXEC.IT extension. To enable this macro, see Section 17.1.4 to apply the options required for EXEC.IT optimization.
`__riscv_xandes`	Computed from Arch version	The macro is defined if the architecture supports AndeStar V5 extension profile. The option `-march=rv*xandes` controls the macro. All V5 toolchains have this option enabled by default.

To view the default value of respective AndeStar V5 predefined macros for a particular toolchain or to check if a feature is enabled by default, issue the following command:

```
$ riscv[32|64]-elf-[gcc|clang] -E -dM - < /dev/null | grep __nds_
```

In addition, AndeStar V5 macros include architecture extension test macros that can be used to check the availability and version of specific architecture extensions. However, these test macros are not supported by all the compilers, so you will need to use `__riscv_arch_test` to check if the compiler in use provides the support. The defined value of each architecture

extension test macro is computed from the architecture version using the following formula:

```
<VALUE> =  <ARCH_MAJOR_VERSION> * 1,000,000 + <ARCH_MINOR_VERSION> * 1,000 +
           <ARCH_REVISION_VERSION>
```

For example, if the version of an architecture extension is 0.92, its corresponding test macro will have the value as 92000. If the architecture version is 2.1, the value of the test macro will be 2001000.

NOTE

The predefined macro `__nds_v5` has been deprecated. This macro was used to indicate support for the AndeStar V5 extension profile, with a value of `1` when enabled. It was controlled by the option `-march=rv*xandes`, which is enabled by default in all AndeStar V5 toolchains.

---

9. ROM patching

Generally speaking, programs in ROM can't be modified after the embedded system IC is taped out. If one would like to upgrade features or fix some problems for programs in ROM, he normally has to put the patch code into the flash memory so that the old implementation can be substituted by the new one. This is known as ROM patching.

ROM patch can be applied through indirect call functions or function table mechanism. Indirect call is an Andes C language extension specially for ROM patching. With the indirect call attribute added to patchable functions and some modifications on the linker script, the code burnt to the ROM has an indirection layer on the flash. When a function is being called, it will look up the function table on the layer for its target address. ROM patching therefore can be achieved through configurations on the function table.

The function table mechanism also applies ROM patches via an indirect layer. It has no addressing limitation and is more portable using the standard C language and few GNU extension. Yet its implementation for ROM patching is comparatively complicated because it requires modifications on many parts of the program for adding the user-defined function table and calling functions through the table.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

9.1. Indirect call functions

9.1.1. Implementation of indirect call functions

The implementation requires modifications on the following parts:

1. Program code or header file - Add an indirect call attribute to declaration of patchable functions
2. Linker script - Add a function table section and allocate it to the flash memory address

9.1.1.1 Apply indirect call attribute to function declaration in your program or header file

To make a function patchable in C programs, you need to add an attribute "`__attribute__((indirect_call))`" to its declaration. It is strongly recommended to put the function declaration containing the indirect call attribute in the header file. This can save the trouble of repeating the attribute in source files and avoid the problem of "mixed calls".

"Mixed calls" of a function refer to a function that is declared inconsistently in different source files and should be avoided when you implement indirect call functions for ROM patching. The following is an example: the function "`foo`" is declared with the indirect call attribute in `main.c` and without the attribute in `bar.c`.

```
<main.c>
int foo(int) __attribute__((indirect_call));
int bar(int);
int foo(int v)
{
  return v;
}
int main()
{
  bar (1) + foo(1);
  return 0;
}
```

```
<bar.c>
int foo(int);
int bar(int v)
{
  return foo(v) +1;
```

```
}
```

Though Andes toolchain can detect mixed calls of a function and try to fix them, the linker gives warnings for the problem:

```
Warning: there are mixed indirect call function 'foo'
```

To get around this error, just put the function declaration with the indirect call attribute in the header file.

9.1.1.2 Add .nds.ict section to linker script

In addition to appending an attribute to function declaration, you also need to add a new section "`.nds.ict`" to your linker script for ROM patching. To make the section overwritable, allocate it to the flash memory address as follows:

```
.nds.ict FLASH_ADDRESS : { *(.nds.ict) }
```

9.1.2. Limitations

Here are some limitations of indirect call implementation:

- **Indirect call functions can't be inline:** To ensure the program is patchable, Andes compiler forbids indirect call functions to be inline.
- **The indirect call attribute applies to extern functions only:** Namely, you cannot declare "`static void foo();`" as an indirect call function by appending "`__attribute__((indirect_call))`" to it.
- **Standard C Library is not recommended for indirect call functions:** The standard C library as compiled binaries has complex call sequence hierarchy and may result in unexpected consequences when used with indirect call functions.
- **Assembly code needs to be handled manually:** For example, use "`bal foo@ICT`" for "`bal foo`".
- **The patch code can't access static variables in the original code:** This is a C language convention that a static variable can't be accessed by any translation units outside its scope.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

9.1.3. Tutorial

Given an example C code "main.c" below, this section demonstrates how to perform ROM patching with indirect call:

```
#include <stdio.h>

void hello(void) {
  printf("hello ");
}

void foo(void) {
  hello();
  printf("world\n");
}

int main(void) {
  foo();
  return 0;
}
```

Preparation: Modify your program by appending "indirect_call" attribute to patchable function(s)

Suppose that you want to make the function "foo" of the program patchable, add the attribute "__attribute__((indirect_call))" to its declaration:

```
__attribute__((indirect_call))
void foo(void) {
    // ...
}
```

Preparation: Modify linker script by defining a .nds.ict section

Next, add an .nds.ict section to your linker script and set it to the flash address. For a layout that ROM starts at 0x0 and the flash memory starts at 0x10000, define the .nds.ict section in the linker script as follows:

```
…
.nds.ict 0x10000 : { *(.nds.ict) }
…
```

Or, write a SaG-formatted file that define the .nds.ict section as follows and use the command nds_ldsag to generate a linker script (see Chapter 13):

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
USER_SECTIONS .nds.ict
LOAD_ROM 0x0 {
  EXEC_ROM 0x0 {
    *(+RO,+RW,+ZI)
  }
}

LOAD_FLASH 0x10000 {
  EXEC_FLASH 0x10000 {
    *(.nds.ict)
  }
}
```

Preparation: Compile and link program with specific options

Determine an ICT (Indirect Call Table) model appropriate for your address space layout and specify the corresponding flag to compile the program. The compilation flags for respective ICT models are listed below:

- `-mict-model=tiny`
  This flag allocates 8 bytes for each call-site and is used if the address space between ROM and flash memory is within ±2MB. This model results in generation of smaller table size. Each entry in the function table has a size of 4 bytes.

- `-mict-model=small` (enabled by default)
  This flag allocates 8 bytes for each call-site and is used if the address space between ROM and flash memory is within ±2GB. This model generates a function table of 8-byte entries.

- `-mict-model=large`
  This flag allocates 20 bytes for each call-site. It results in larger code size, yet has no limitation on address space layout. If the address space between ROM and flash memory is beyond ±2GB, make sure you use this flag for compilation. Note that this model is only available for 64-bit toolchains and must work with the large code model (`-mcmodel=large`).

Also, append the options "`-Wl,--mexport-symbols=sym.ld`" to link the program and export the symbol information to the file "`sym.ld`", which is required later for patching functions. `sym.ld` contains all symbol addresses in the program and thus can allow the linker to reference symbols in the original program when generating the patch code

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

For example, with a linker script "main.ld" and an address space of 16MB between ROM and flash memory, use the following commands to build the program:

```
$ riscv32-elf-[gcc|clang] -mict-model=small -Wl,-T, main.ld -Wl,--
mexport-symbols=sym.ld main.c -o main
```

Create patch code

Now you can patch a function declared with the indirect_call attribute. For example, create patch code (patch.c) that contains a new definition of the function "foo" as follows:

```
#include <stdio.h>

void hello(void);

__attribute__((indirect_call))
void foo(void) {
  hello();
  printf("patch\n");
}
```

Care must be taken to ensure that the new function is ABI-compatible with the original one. Nevertheless, it is okay to call global functions in the original program, such as the "hello" function in this example.

Write linker script to place ICT table and patch code into flash

As the ROM is not writable, the .nds.ict section (i.e., the ICT table) and the following patch code must be placed into the flash memory, which starts at 0x10000 in this case. This can be specified by defining the .nds.ict section in a linker script as follows:

```
…
.nds.ict 0x10000 : { *(.nds.ict) }
…
```

Or, write a SaG-formatted file as follows to create a new linker script utilizing the linker script generator command nds_ldsag (see Chapter 13):

```
USER_SECTIONS .nds.ict

LOAD_FLASH 0x10000 {
  EXEC_FLASH 0x10000 {
    *(.nds.ict)
    *(+RO,+RW,+ZI)
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
    }
}
```

Generate patch image

Rename the new linker script as "`patch.ld`" and generate the patch image using the commands below:

```
$ riscv32-elf-[gcc|clang] -mict-model=small -fno-zero-initialized-in-
bss -nostdlib -Wl,-T,sym.ld -Wl,-T,patch.ld patch.c -o patch
```

The option "`-nostdlib`" prevents the linker from grabbing C library into the patch image while "`-fno-zero-initialized-in-bss`" prevents the compiler from putting variables into the `.bss` section. The latter is used because the original code that clears the `.bss` section doesn't know the new `.bss` section in the patch code.

9.1.4. Using Zcmt extension in patch code

To utilize the Zcmt extension in patch code, use the linker options "`-Wl,--mexport-table-jump`" and "`-Wl,--mimport-table-jump`" to export and import the `.riscv.jvt` section, which contains the jump table required for the Zcmt extension. For example,

```
$ riscv32-elf-[gcc|clang] -mict-model=small -Wl,-T, main.ld -Wl,--mexport-
symbols=sym.ld -Wl,--mexport-table-jump=riscv.jvt main.c -o main

$ riscv32-elf-[gcc|clang] -mict-model=small -fno-zero-initialized-in-bss
-nostdlib -Wl,-T,sym.ld -Wl,-T,patch.ld -Wl,--mimport-table-jump=riscv.jvt
patch.c -o patch
```

In addition, take note of the following restrictions when using the Zcmt extension in the patch.

- The patch code must not update the jump table for the Zcmt extension.
- Only indirect call functions are allowed in the flash. If the flash contains other types of functions, the table jump relaxation in the patch may operate incorrectly.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

9.2. Function table mechanism

9.2.1. Implementation of function table mechanism

This mechanism requires modifications on the following parts:

1. Program code - Add a function table for patchable functions
2. Program code - Change each call-site for patchable functions
3. Linker script - Add a function table section and allocate it to the flash memory address

9.2.1.1 Adding function table for patchable functions to your program

In your program, define a structure that includes variables for patchable functions. For example,

```
int bar(int);
int foo(int);

typedef struct {
  int (*foo)(int);
  int (*bar)(int);
} func_table_t;
```

Declare a variable "func_table" and initialize the data for patchable functions. In case "func_table" is optimized out by the compiler, DO NOT declare it as a static or const variable.

```
struct func_table_t func_table __attribute__ ((section ("FUNC_TABLE")))
=
 {.foo = foo,
  .bar = bar};
```

9.2.1.2 Changing every call-site for patch-able functions in your program

For example, given the call-site for the function "bar" like below:

```
printf ("bar 10 = %d\n", bar (10));
```

Modify it as follows so that it can be called via func_table:

```
printf ("bar 10 = %d\n", func_table.bar (10));
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

9.2.1.3 Adding function table section to linker script

Add a new section ".FUNC_TABLE" to your linker script. To make the section overwritable, allocate it to the flash memory address as follows:

```
.FUNC_TABLE FLASH_ADDRESS : { *(.FUNC_TABLE) }
```

9.2.2. Limitations

- **Assembly code needs to be handled manually:** For example, replace "bal foo" with
```
la t0, func_table
lw t0, offset of foo in func_table(t0)
jalr t0
```
- **The patch code can't access static variables in the original code:** This is a C language convention that a static variable can't be accessed by any translation units outside its scope.

9.2.3. Tutorial

Given an example C code "main.c" like below, this section demonstrates how to perform ROM patching with function table mechanism:

```
#include <stdio.h>

void hello(void) {
  printf("hello ");
}

void foo(void) {
  hello();
  printf("world\n");
}

int main(void) {
  foo();
  return 0;
}
```

Preparation: Modify program code

Suppose that you want to make the function "foo" of the program patchable, declare a func_table_t struct which holds an array of function pointers to the patchable function.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Next, initialize the table with the original "foo" function and place the table in a user-defined section ".func_table". Then, modify all call-sites for the function "foo" so that they can be called through the function pointers in the table.

```
__attribute__((section(".func_table")))
struct func_table_t {
  void (*foo)(void);
} func_table = { .foo = foo };

int main(void) {
  func_table.foo();
  return 0;
}
```

Preparation: Modify linker script

Next, add a .func_table section to your linker script and set it to the flash address. For a layout that ROM starts at 0x0 and the flash memory starts at 0x10000, define the .func_table section in your linker script as follows:

```
…
.func_table 0x10000 : { *(.func_table) }
…
```

Or, write a SaG-formatted file that define the .func_table section as follows and use the command nds_ldsag to generate a linker script (see Chapter 13):

```
USER_SECTIONS .func_table

LOAD_ROM 0x0 {
  EXEC_ROM 0x0 {
    *(+RO,+RW,+ZI)
  }
}
LOAD_FLASH 0x10000 {
  EXEC_FLASH 0x10000 {
    *(.func_table)
  }
}
```

Preparation: Link program with specific options

Then, use the options "-Wl,--mexport-symbols=sym.ld" to link the program and export the

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

symbol information to the file "sym.ld", which is required later for patching functions. For example, with a linker script "main.ld", use the following commands to build the program:

```
$ riscv32-elf-[gcc|clang] -Wl,-T,main.ld -Wl,--mexport-symbols=sym.ld
main.c -o main
```

Create patch code

To patch the function "foo" in "main.c", create patch code (patch.c) that contains a new definition of "foo" as follows:

```
#include <stdio.h>

void hello(void);

void foo(void) {
  hello();
  printf("patch\n");
}
```

Care must be taken to ensure that the new function is ABI-compatible with the original one. Nevertheless, it is okay to call global functions in the original program, such as the "hello" function in the above example.

Then, declare a function table as you did for the original program and initialize the function table with the new "foo" function. The layout and order of the function table in the patch code must be exactly the same as that for the original program.

```
__attribute__((section(".func_table")))
struct func_table_t {
  void (*foo)(void);
} func_table = { .foo = foo };
```

Edit sym.ld to clear reference to the original "foo"

The sym.ld generated after linking the original program contains the address of the original "foo" function. Before you link the patch code and generate the patch image, make sure to remove the reference from sym.ld by deleting the following line:

```
foo = <address>;
```

Modify linker script and sym.ld

As the ROM is not writable, the .func_table section and the following patch code must be

placed into the flash memory, which starts at 0x10000 in this case. This can be specified by defining the `.func_table` section in a linker script as follows:

```
…
.func_table 0x10000 : { *(.func_table) }
…
```

Or, write a SaG-formatted file as follows to create a new linker script utilizing the linker script generator command `nds_ldsag` (see Chapter 13):

```
USER_SECTIONS .func_table

LOAD_FLASH 0x10000 {
  EXEC_FLASH 0x10000 {
    *(.func_table)
    *(+RO,+RW,+ZI)
  }
}
```

Generate patch image

Rename the modified linker script as "`patch.ld`" and generate the patch image using the commands below:

```
$ riscv32-elf-[gcc|clang] -fno-zero-initialized-in-bss -nostdlib -Wl,-T,sym.ld -Wl,-T,patch.ld patch.c -o patch
```

The option "`-nostdlib`" prevents the linker from grabbing C library into the patch image while "`-fno-zero-initialized-in-bss`" prevents the compiler from putting variables into the `.bss` section. The latter is used because the original code that clears the `.bss` section doesn't know the new `.bss` section in the patch code.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

10. AndeStar V5 intrinsic function programming

In compiler theory, an intrinsic function is a function available in a given language whose implementation is handled specially by the compiler. If a function is intrinsic, the code for that function is usually inserted inline, avoiding the overhead of a function call and allowing highly efficient machine instructions to be emitted for that function.

The current AndeStar intrinsic functions are for users (including OS engineers) who don't want to program in assembly. They cover all the operations which the compiler cannot generate.

NOTE

Be sure to use the correct signedness for arguments and return values when calling intrinsic functions. The compiler has a strict type checking. It gives warnings for incorrect signedness and reports errors if the option `-Werror` is specified.

10.1. Summary of AndeStar V5 intrinsic functions

For each AndeStar V5 intrinsic function, its syntax, mapped AndeStar V5 instruction, and schedulability (i.e. if the compiler can schedule it or not) are shown in the following tables.

Table 14. Intrinsics for RV32I and RV64I

Intrinsic Function Syntax	Alias to	Mapped AndeStar Instruction	Schedulable	Page
`void __nds__fence(const unsigned int PIORW, const unsigned int SIORW)`		`fence`	No	72
`viod __nds__fencei(void)`		`fence.i`	No	74
`long __nds__ecall(long ID)`		`ecall`	No	75
`long __nds__ecall1(long ID, long ARG1)`		`ecall`		75
`long __nds__ecall2(long ID, long ARG1, long ARG2)`		`ecall`		75

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Intrinsic Function Syntax	Alias to	Mapped AndeStar Instruction	Schedulable	Page
long __nds__ecall3(long ID, long ARG1, long ARG2, long ARG3)		ecall		75
long __nds__ecall4(long ID, long ARG1, long ARG2, long ARG3, long ARG4)		ecall		75
long __nds__ecall5(long ID, long ARG1, long ARG2, long ARG3, long ARG4, long ARG5)		ecall		75
long __nds__ecall6(long ID, long ARG1, long ARG2, long ARG3, long ARG4, long ARG5, long ARG6)		ecall		75
void __nds__ebreak(unsigned long ARG)		ebreak	No	76
unsigned long __nds__csrrw(unsigned long SRC, const unsigned int CSRN)		csrrw csrrwi	No	77
unsigned long __nds__swap_csr(unsigned long VAL, const unsigned int CSRN)	__nds__csrrw (VAL, CSRN)	csrrw csrrwi	No	77
unsigned long __nds__csrrs(unsigned long SRC, const unsigned int CSRN)		csrrs csrrsi	No	78
unsigned long __nds__read_and_set_csr(unsigned long SETMASK, const unsigned int CSRN)	__nds__csrrs (SETMASK, CSRN)	csrrs csrrsi	No	78

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Intrinsic Function Syntax	Alias to	Mapped AndeStar Instruction	Schedulable	Page
unsigned long __nds__csrrc(unsigned long **SRC**, const unsigned int **CSRN**)		csrrc csrrci	No	79
unsigned long __nds__read_and_clear_csr(unsigned long **CLRMASK**, const unsigned int **CSRN**)	__nds__csrrc (**CLRMASK**, **CSRN**)	csrrc csrrci	No	79
unsigned long __nds__csrr(const unsigned int **CSRN**)		csrr	No	80
unsigned long __builtin_riscv_csrr(unsigned int **CSRN**)	__nds__csrr(**CSRN**)	csrr	No	80
unsigned long __nds__mfsr(unsigned int **CSRN**)	__nds__csrr(**CSRN**)	csrr	No	80
unsigned long __nds__read_csr(const unsigned int **CSRN**)	__nds__csrr(**CSRN**)	csrr	No	80
void __nds__csrw(unsigned long **SRC**, const unsigned int **CSRN**)		csrw csrwi	No	81
__builtin_riscv_csrw(unsigned long **SRC**, unsigned int **CSRN**)	__nds__csrw(**SRC**, **CSRN**)	csrw csrwi	No	81
void __nds__mtsr(unsigned long **SRC**, unsigned int **CSRN**)	__nds__csrw(**SRC**, **CSRN**)	csrw csrwi	No	81
void __nds__write_csr(unsigned long **VAL**, const unsigned int **CSRN**)	__nds__csrw(**CSRN**)	csrw csrwi	No	81

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Intrinsic Function Syntax	Alias to	Mapped AndeStar Instruction	Schedulable	Page
void __nds__csrs(unsigned long **SRC**, const unsigned int **CSRN**)		csrs csrsi	No	82
void __nds__set_csr_bits(unsigned long **SETMASK**, const unsigned int **CSRN**)	__nds__csrs(**SETMASK**, **CSRN**)	csrs csrsi	No	82
void __nds__csrc(unsigned long **SRC**, const unsigned int **CSRN**)		csrc csrci	No	83
void __nds__clear_csr_bits(unsigned long **CLRMASK**, const unsigned int **CSRN**)	__nds__csrc(**CLRMASK**, **CSRN**)	csrc csrci	No	83
unsigned long __nds__get_current_sp()			No	84
unsigned long __builtin_riscv_get_current_sp()	__nds__get_current_sp()		No	84
void __nds__set_current_sp(unsigned long **VALUE**)			No	85
unsigned long __builtin_riscv_set_current_sp(unsigned long **VALUE**)	__nds__set_current_sp(**VALUE**)		No	85

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Table 15. Intrinsics for RV32 and RV64 Cache Management Operations "CMO"

Intrinsic Function Syntax	Alias to	Mapped Andes Instruction	Schedulable	Page

`void __nds__prefetch(void *ADDR, unsigned IS_WRITE, unsigned LOCALITY, unsigned IS_DATA)`		`prefetch.i`
		`prefetch.r`
		`prefetch.w`
			Yes	86

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Table 16. Intrinsics for RV32 and RV64 floating-point extensions "F" and "D"

Intrinsic Function Syntax	Alias to	Mapped Andes Instruction	Schedulable	Page
unsigned long __nds__frcsr()		frcsr	No	88
unsigned long __nds__fscsr(unsigned long SRC)		fscsr	No	89
void __nds__fwcsr(unsigned long SRC)		fscsr	No	90
unsigned long __nds__frrm()		frrm	No	91
unsigned long __nds__fsrm(unsigned long SRC)		fsrm fsrmi	No	92
void __nds__fwrm(unsigned long SRC)		fsrm fsrmi	No	93
unsigned long __nds__frflags()		frflags	No	94
unsigned long __builtin_riscv_frflags()	__nds__frflags()	frflags	No	94
unsigned long __nds__fsflags(unsigned long SRC)		fsflags fsflagsi	No	95
unsigned long __builtin_riscv_fsflags(unsigned long SRC)	__nds__fsflags(SRC)	fsflags fsflagsi	No	95
void __nds__fwflags(unsigned long SRC)		fsflags fsflagsi	No	96
float __nds_fcvt_s_bf16(__bf16 BF16_SRC)		fcvt.s.bf16	Yes	97

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Intrinsic Function Syntax	Alias to	Mapped Andes Instruction	Schedulable	Page
`__bf16 __nds_fcvt_bf16_s(float FP32_SRC)`		`fcvt.bf16.s`	Yes	98

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Table 17. Intrinsics for RV32 & RV64 atomic extension "A"

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
signed int __nds__amoswapw(signed int SRC, signed int* BASE, const unsigned int ORDERING)	amoswap.w	No	99
signed int __nds__amoaddw(signed int SRC, signed int* BASE, const unsigned int ORDERING)	amoadd.w	No	100
unsigned int __nds__amoxorw(unsigned int SRC, unsigned int* BASE, const unsigned int ORDERING)	amoxor.w	No	101
unsigned int __nds__amoandw(unsigned int SRC, unsigned int* BASE, const unsigned int ORDERING)	amoand.w	No	102
unsigned int __nds__amoorw(unsigned int SRC, unsigned int* BASE, const unsigned int ORDERING)	amoor.w	No	103
signed int __nds__amominw(signed int SRC, signed int* BASE, const unsigned int ORDERING)	amomin.w	No	104
signed int __nds__amomaxw(signed int SRC, signed int* BASE, const unsigned int ORDERING)	amomax.w	No	105
unsigned int __nds__amominuw(unsigned int SRC, unsigned int* BASE, const unsigned int ORDERING)	amominu.w	No	106
unsigned int __nds__amomaxuw(unsigned int SRC, unsigned int* BASE, const unsigned int ORDERING)	amomaxu.w	No	107

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Table 18. Intrinsics for RV64-only atomic extension "A"

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`signed long __nds__amoswapd(signed long long SRC, signed long long* BASE, const unsigned int ORDERING)`	amoswap.d	No	108
`signed long __nds__amoaddd(signed long long SRC, signed long long* BASE, const unsigned int ORDERING)`	amoadd.d	No	109
`unsigned long __nds__amoxord(unsigned long long SRC, unsigned long long* BASE, const unsigned int ORDERING)`	amoxor.d	No	110
`unsigned long __nds__amoandd(unsigned long long SRC, unsigned long long* BASE, const unsigned int ORDERING)`	amoand.d	No	111
`unsigned long __nds__amoord(unsigned long long SRC, unsigned long long* BASE, const unsigned int ORDERING)`	amoor.d	No	112
`signed long __nds__amomind(signed long long SRC, signed long long* BASE, const unsigned int ORDERING)`	amomin.d	No	113
`signed long __nds__amomaxd(signed long long SRC, signed long long* BASE, const unsigned int ORDERING)`	amomax.d	No	114
`unsigned long __nds__amominud(unsigned long long SRC, unsigned long long* BASE, const unsigned int ORDERING)`	amominu.d	No	115
`unsigned long __nds__amomaxud(unsigned long long SRC, unsigned long long* BASE, const unsigned int ORDERING)`	amomaxu.d	No	116

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

10.2. Detailed intrinsic function descriptions

10.2.1. Intrinsics for RV32I and RV64I

Name

`__nds__fence`

Syntax

```
void __nds__fence(const unsigned int PIORW, const unsigned int SIORW)
```

Description

This intrinsic inserts a `fence` instruction into the instruction stream. The constant operands of **`PIORW`** and **`SIORW`** are defined in the following table:

unsigned int IORW	value
FENCE_W	1
FENCE_R	2
FENCE_RW	3
FENCE_O	4
FENCE_OW	5
FENCE_OR	6
FENCE_ORW	7
FENCE_I	8
FENCE_IW	9
FENCE_IR	10
FENCE_IRW	11
FENCE_IO	12
FENCE_IOW	13
FENCE_IOR	14
FENCE_IORW	15

Return Value: None

Privilege Level: ALL

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Make sure “valid” is set after “data” is written.
   unsigned int valid = 0;
   unsigned int data;
    data = 1000;
    __nds__fence(FENCE_W, FENCE_W);
    valid = 1;
}
```

Name

`__nds__fencei`

Syntax

`viod __nds__fencei(void)`

Description

This intrinsic inserts a `fence.i` instruction into the instruction stream.

Return Value: None

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Self-modify program code.
   unsigned int *code;
   code = 0x20000;
    *code = 0x12345678;
    __nds__fencei();
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__ecall, __nds__ecall[1..6]`

Syntax

```
long __nds__ecall(long ID)
long __nds__ecall1(long ID, long ARG1)
long __nds__ecall2(long ID, long ARG1, long ARG2)
long __nds__ecall3(long ID, long ARG1, long ARG2, long ARG3)
long __nds__ecall4(long ID, long ARG1, long ARG2, long ARG3, long ARG4)
long __nds__ecall5(long ID, long ARG1, long ARG2, long ARG3, long ARG4,
      long ARG5)
long __nds__ecall6(long ID, long ARG1, long ARG2, long ARG3, long ARG4,
       long ARG5, long ARG6)
```

Description

This intrinsic inserts an `ecall` instruction into the instruction stream.

Return Value: Return value for ecall ID.

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Insert an "ecall" instruction.
   long status;
    status = __nds__ecall(0x12341234);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__ebreak`

Syntax

`void __nds__ebreak(unsigned long ARG)`

Description

This intrinsic inserts an `ebreak` instruction into the instruction stream.

Return Value: None

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Insert an "ebreak" instruction.
    __nds__ebreak(0x98761234);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__csrrw`

Syntax

```
unsigned long __nds__csrrw(unsigned long SRC, const unsigned int CSRN)
```

Alias

```
unsigned long __nds__swap_csr(unsigned long VAL, const unsigned int CSRN)
```

Description

This intrinsic inserts a `csrrw` or `csrrwi` instruction into the instruction stream. The content of the source operand `SRC` will be written into the CSR `csrn`. The original content of the CSR `csrn` will be returned.

Return Value: The original content of the CSR `csrn`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Read and write "mie" CSR.
   unsigned int src, mie_data;
   src = 0x888;
   mie_data = __nds__csrrw(src, 0x304);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__csrrs`

Syntax

`unsigned long __nds__csrrs(unsigned long SRC, const unsigned int CSRN)`

Alias

`unsigned long __nds__read_and_set_csr(unsigned long SETMASK, const unsigned int CSRN)`

Description

This intrinsic inserts a `csrrs` or `csrrsi` instruction into the instruction stream. The content of the source operand `SRC` will be used to set bits in the CSR `csrn`. The original content of the CSR `csrn` will be returned.

Return Value: The original content of the CSR `csrn`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Read and set bits in "mie" CSR.
   unsigned int src, mie_data;
   src = 0x888;
   mie_data = __nds__csrrs(src, 0x304);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__csrrc`

Syntax

```
unsigned long __nds__csrrc(unsigned long SRC, const unsigned int CSRN)
```

Alias

```
unsigned long __nds__read_and_clear_csr(unsigned long CLRMASK, const
unsigned int CSRN)
```

Description

This intrinsic inserts a `csrrc` or `csrrci` instruction into the instruction stream. The content of the source operand `SRC` will be used to clear bits in the CSR `csrn`. The original content of the CSR `csrn` will be returned.

Return Value: The original content of the CSR

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Read and clear bits in "mie" CSR.
   unsigned int src, mie_data;
   src = 0x888;
   mie_data = __nds__csrrc(src, 0x304);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__csrr`

Syntax

`unsigned long __nds__csrr(const unsigned int CSRN)`

Alias

```
unsigned long __builtin_riscv_csrr(unsigned int CSRN)
unsigned long __nds__mfsr(unsigned int CSRN)
unsigned long __nds__read_csr(const unsigned int CSRN)
```

Description

This intrinsic inserts a `csrr` or `csrri` pseudo-instruction into the instruction stream. The original content of the CSR `csrn` will be returned.

Return Value: The original content of the CSR

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Read "mie" CSR.
   unsigned int mie_data;
   mie_data = __nds__csrr(0x304);
}
```

Name

`__nds__csrw`

Syntax

`void __nds__csrw(unsigned long SRC, const unsigned int CSRN)`

Alias

`void __builtin_riscv_csrw(unsigned long SRC, unsigned int CSRN)`

`void __nds__mtsr(unsigned long SRC, unsigned int CSRN)`void

`__nds__write_csr(unsigned long VAL, const unsigned int CSRN)`

Description

This intrinsic inserts a `csrw` or `csrwi` pseudo-instruction into the instruction stream. The content of the source operand **SRC** will be written into the CSR `csrn`.

Return Value: None

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Write “mie” CSR.
  unsigned int src;
  src = 0x888;
  __nds__csrw(src, 0x304);
 }
```

Name

`__nds__csrs`

Syntax

`void __nds__csrs(unsigned long SRC, const unsigned int CSRN)`

Alias

`void __nds__set_csr_bits(unsigned long SETMASK, const unsigned int CSRN)`

Description

This intrinsic inserts a `csrs` or `csrsi` pseudo-instruction into the instruction stream. The content of the source operand **SRC** will be used to set bits in the CSR `csrn`.

Return Value: None

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Set bits in “mie” CSR.
  unsigned int src;
  src = 0x888;
  __nds__csrs(src, 0x304);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__csrc`

Syntax

```
void __nds__csrc(unsigned long SRC, const unsigned int CSRN)
```

Alias

```
void __nds__clear_csr_bits(unsigned long CLRMASK, const unsigned int CSRN)
```

Description

This intrinsic inserts a `csrc` or `csrci` pseudo-instruction into the instruction stream. The content of the source operand **`SRC`** will be used to clear bits in the CSR `csrn`.

Return Value: None

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Clear bits in "mie" CSR.
  unsigned int src;
  src = 0x888;
  __nds__csrc(src, 0x304);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__get_current_sp`

Syntax

`unsigned long __nds__get_current_sp()`

Alias

`unsigned long __builtin_riscv_get_current_sp()`

Description

This intrinsic returns the content of the stack pointer register.

Return Value: The current stack pointer register value.

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
  unsigned long spval;
  spval = __nds__get_current_sp();
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__set_current_sp`

Syntax

`void __nds__set_current_sp(unsigned long VALUE)`

Alias

`unsigned long __builtin_riscv_set_current_sp(unsigned long VALUE)`

Description

This intrinsic sets the content of the stack pointer register.

Return Value: None.

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
  unsigned long spval;
  spval = 0x80000
  __nds__set_current_sp(spval);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

10.2.2. Intrinsic for RV32 and RV64 Cache Management Operations "CMO"

Name

`__nds__prefetch`

Syntax

```
void __nds__prefetch(void * ADDR, unsigned IS_WRITE, unsigned LOCALITY,
unsigned IS_DATA)
```

where the parameters

`ADDR` is the memory address to prefetch.

`IS_WRITE` must be a compile-time constant `1` or `0`. It is set to `1` if the prefetch is for a write to the memory address.

`LOCALITY` must be a compile-time constant integer from `0` to `3`, with `0` indicating no temporal locality. This parameter primarily determines the additional inserted instruction before a prefetch instruction when the RISC-V Zihintntl extension is enabled.

`IS_DATA` must be a compile-time constant `1` or `0`. It is set to `0` if the address is going to be accessed by an instruction fetch or `1` if to be accessed by a data read or write.

Description

This intrinsic inserts a `prefetch.i`, `prefetch.r` or `prefetch.w` instruction into the instruction stream. The mapping between combinations of `IS_WRITE` and `IS_DATA` and their corresponding inserted prefetch instruction is summarized below.

IS_WRITE	IS_DATA	LOCALITY	Inserted prefetch instruction
0	0	0~3	prefetch.i
0	1	0~3	prefetch.r
1	0	0~3	prefetch.i
1	1	0~3	prefetch.w

However, if the RISC-V Zihintntl extension is also enabled, another instruction may also be inserted according to `LOCALITY` before the prefetch instruction listed above. The following table lists the additional instruction that may be inserted.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

LOCALITY	Additional inserted instruction before the prefetch instruction with Zihintntl enabled
0	ntl.all
1	ntl.pall
2	ntl.p1
3	--

For example, with `IS_WRITE=0`, `IS_DATA=0`, and `LOCALITY=0`, the inserted instruction is `prefetch.i` alone when Zihintntl is not enabled and `ntl.all` followed by `prefetch.i` when Zihintntl is enabled.

Return Value: None

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(int *input, int *output, int n)
{
   for(int i = 0; i < n; i++)
   {
      // prefetch data reads in advance.
      __nds__prefetch(input + (i + 40), 0, 0, 1);
      // prefetch data writes in advance.
      __nds__prefetch(output + (i + 40), 1, 0, 1);
      output[i] = input[i];
   }
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

10.2.3. Intrinsics for RV32 and RV64 floating-point extensions “F” and “D”

Name

`__nds__frcsr`

Syntax

`unsigned long __nds__frcsr()`

Description

This intrinsic inserts a `frcsr` pseudo-instruction into the instruction stream. The content of the CSR `fcsr` will be returned.

Return Value: The content of the CSR `fcsr`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
  //Read “fcsr” CSR.
  unsigned int fcsr_data;
  fcsr_data = __nds__frcsr();
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__fscsr`

Syntax

`unsigned long __nds__fscsr(unsigned long SRC)`

Description

This intrinsic inserts a `fscsr` pseudo-instruction into the instruction stream. The content of the source operand `SRC` will be written into the CSR `fcsr`. The original content of the CSR `fcsr` will be returned.

Return Value: The original content of the CSR `fcsr`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
  //Read and write “fcsr” CSR.
  unsigned int src, fcsr_data;
  src = 0x60;
  fcsr_data = __nds__fscsr(src);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__fwcsr`

Syntax

`void __nds__fwcsr(unsigned long SRC)`

Description

This intrinsic inserts a "`fscsr rs`" pseudo-instruction into the instruction stream. The content of the source operand **`SRC`** will be written into the CSR `fcsr`.

Return Value: None

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Write "fcsr" CSR.
   unsigned int src;
   src = 0x60;
   __nds__fwcsr(src);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__frrm`

Syntax

`unsigned long __nds__frrm()`

Description

This intrinsic inserts a `frrm` pseudo-instruction into the instruction stream. The content of the CSR field `fcsr.FRM` will be returned.

Return Value: The content of the CSR field `fcsr.FRM`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Read "fcsr.FRM" CSR field.
  unsigned int frm_data;
  frm_data = __nds__frrm();
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__fsrm`

Syntax

`unsigned long __nds__fsrm(unsigned long SRC)`

Description

This intrinsic inserts a `fsrm` or `fsrmi` pseudo-instruction into the instruction stream. The content of the source operand **`SRC`** will be written into the CSR field `fcsr.FRM`. The original content of the CSR field `fcsr.FRM` will be returned.

Return Value: The original content of the CSR field `fcsr.FRM`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
  //Read and write "fcsr.FRM" CSR field.
  unsigned int src, frm_data;
  src = 0x3;
  frm_data = __nds__fsrm(src);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__fwrm`

Syntax

```
void __nds__fwrm(unsigned long SRC)
```

Description

This intrinsic inserts a "`fsrm rs`" or "`fsrmi imm`" pseudo-instruction into the instruction stream. The content of the source operand **`SRC`** will be written into the CSR field `fcsr.FRM`.

Return Value: None

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Write "fcsr.FRM" CSR field.
   unsigned int src;
   src = 0x3;
   __nds__fwrm(src);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__frflags`

Syntax

`unsigned long __nds__frflags()`

Alias

`unsigned long __builtin_riscv_frflags()`

Description

This intrinsic inserts a `frflags` pseudo-instruction into the instruction stream. The content of the CSR field `fcsr.FFLAGS` will be returned.

Return Value: The content of the CSR field `fcsr.FFLAGS`.

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //Read “fcsr.FFLAGS” CSR field.
   unsigned int flags_data;
   flags_data = __nds__frflags();
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__fsflags`

Syntax

`unsigned long __nds__fsflags(unsigned long SRC)`

Alias

`unsigned long __builtin_riscv_fsflags(unsigned long SRC)`

Description

This intrinsic inserts a `fsflags` or `fsflagsi` pseudo-instruction into the instruction stream. The content of the source operand `SRC` will be written into the CSR field `fcsr.FFLAGS`. The original content of the CSR field `fcsr.FFLAGS` will be returned.

Return Value: The original content of the CSR field `fcsr.FFLAGS`.

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
  //Read and write "fcsr.FFLAGS" CSR field.
  unsigned int src, flags_data;
  src = 0x0;
  flags_data = __nds__fsflags(src);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__fwflags`

Syntax

`void __nds__fwflags(unsigned long SRC)`

Description

This intrinsic inserts a “`fsflags rs`” or “`fsflagsi imm`” pseudo-instruction into the instruction stream. The content of the source operand **`SRC`** will be written into the CSR field `fcsr.FFLAGS`.

Return Value: None

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
  //Write “fcsr.FFLAGS” CSR field.
  unsigned int src;
  src = 0x0;
  __nds__fwflags(src);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds_fcvt_s_bf16`

Syntax

`float __nds_fcvt_s_bf16(__bf16 BF16_SRC)`

Description

This intrinsic converts BFLOAT16 data to single-precision floating-point (SP) data.

Return Value: float

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
  __bf16 bf_input;
  float sf_result;

  bf_input = __nds_fcvt_bf16_s(sf_result);
  sf_result = __nds_fcvt_s_bf16(bf_input);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds_fcvt_bf16_s`

Syntax

`__bf16 __nds_fcvt_bf16_s(float FP32_SRC)`

Description

This intrinsic converts single-precision floating-point (SP) data to BFLOAT16 data.

Return Value: BFLOAT16

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
  __bf16 bf_input;
  float sf_result;

  bf_input = __nds_fcvt_bf16_s(sf_result);
  sf_result = __nds_fcvt_s_bf16(bf_input);
 }
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

10.2.4. Intrinsics for RV32 and RV64 atomic extension "A"

Name

__nds__amoswapw

Syntax

signed int __nds__amoswapw(signed int **SRC**, signed int* **BASE**, const unsigned int **ORDERING**)

Description

This intrinsic inserts an AMOSWAP.W instruction into the instruction stream. The memory address for the operation is specified by **BASE**. The constant operands of **ORDERING** are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address **BASE**

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic swap operation.
  signed int data, newv, oldv;
  newv = 10;
  // new value: 10
  oldv = __nds__amoswapw(newv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amoaddw`

Syntax

```
signed int __nds__amoaddw(signed int SRC, signed int* BASE, const unsigned
int ORDERING)
```

Description

This intrinsic inserts an `AMOADD.W` instruction into the instruction stream. The memory address for the operation is specified by **BASE**. The constant operands of **ORDERING** are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address **BASE**

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic add operation.
   signed int data, addv, oldv;
   addv = 10;
   // new value: data + 10
   oldv = __nds__amoaddw(addv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amoxorw`

Syntax

```
unsigned int __nds__amoxorw(unsigned int SRC, unsigned int* BASE, const
unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOXOR.W` instruction into the instruction stream. The memory address for the operation is specified by **BASE**. The constant operands of **ORDERING** are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address **BASE**

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic xor operation.
  unsigned int data, xorv, oldv;
  xorv = 0x22334455;
  // new value: data xor 0x22334455
  oldv = __nds__amoxorw(xorv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amoandw`

Syntax

```
unsigned int __nds__amoandw(unsigned int SRC, unsigned int* BASE, const
unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOAND.W` instruction into the instruction stream. The memory address for the operation is specified by `BASE`. The constant operands of `ORDERING` are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address `BASE`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic AND operation.
  unsigned int data, andv, oldv;
  andv = 0x22334455;
  // new value: data AND 0x22334455
  oldv = __nds__amoandw(andv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amoorw`

Syntax

```
unsigned int __nds__amoorw(unsigned int SRC, unsigned int* BASE, const
unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOOR.W` instruction into the instruction stream. The memory address for the operation is specified by `BASE`. The constant operands of `ORDERING` are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address `BASE`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic or operation.
  unsigned int data, orv, oldv;
  orv = 0x22334455;
  // new value: data OR 0x22334455
  oldv = __nds__amoorw(orv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amominw`

Syntax

```
signed int __nds__amominw(signed int SRC, signed int* BASE, const unsigned
int ORDERING)
```

Description

This intrinsic inserts an `AMOMIN.W` instruction into the instruction stream. The memory address for the operation is specified by **BASE**. The constant operands of **ORDERING** are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address **BASE**

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic min operation.
   signed int data, cmpv, oldv;
   cmpv = 10;
   // new value: minimum(data, cmpv)
   oldv = __nds__amominw(cmpv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amomaxw`

Syntax

```
signed int __nds__amomaxw(signed int SRC, signed int* BASE, const unsigned
int ORDERING)
```

Description

This intrinsic inserts an `AMOMAX.W` instruction into the instruction stream. The memory address for the operation is specified by **BASE**. The constant operands of **ORDERING** are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address **BASE**

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic max operation.
   signed int data, cmpv, oldv;
   cmpv = 10;
   // new value: maximum(data, cmpv)
   oldv = __nds__amomaxw(cmpv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amominuw`

Syntax

```
unsigned int __nds__amominuw(unsigned int SRC, unsigned int* BASE, const
unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOMINU.W` instruction into the instruction stream. The memory address for the operation is specified by `BASE`. The constant operands of `ORDERING` are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address `BASE`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic unsigned min operation.
  unsigned int data, cmpv, oldv;
  cmpv = 0x22334455;
  // new value: unsigned minimum(data, cmpv)
  oldv = __nds__amominuw(cmpv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amomaxuw`

Syntax

```
unsigned int __nds__amomaxuw(unsigned int SRC, unsigned int* BASE, const
unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOMAXU.W` instruction into the instruction stream. The memory address for the operation is specified by `BASE`. The constant operands of `ORDERING` are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address `BASE`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic unsigned min operation.
  unsigned int data, cmpv, oldv;
  cmpv = 0x22334455;
  // new value: unsigned maximum(data, cmpv)
  oldv = __nds__amomaxuw(cmpv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

10.2.5. Intrinsics for RV64-only atomic extension "A"

Name

`__nds__amoswapd`

Syntax

`signed long __nds__amoswapd(signed long long SRC, signed long long* BASE, const unsigned int ORDERING)`

Description

This intrinsic inserts an `AMOSWAP.D` instruction into the instruction stream. The memory address for the operation is specified by **BASE**. The constant operands of **ORDERING** are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address **BASE**

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic swap operation.
   signed long long data, newv, oldv;
   newv = 10;
   // new value: 10
   oldv = __nds__amoswapd(newv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amoaddd`

Syntax

```
signed long __nds__amoaddd(signed long long SRC, signed long long* BASE,
const unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOADD.D` instruction into the instruction stream. The memory address for the operation is specified by `BASE`. The constant operands of `ORDERING` are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address `BASE`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
    //We want to perform an atomic add operation.
   signed long long data, addv, oldv;
   addv = 10;
   // new value: data + 10
   oldv = __nds__amoaddd(addv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amoxord`

Syntax

```
unsigned long __nds__amoxord(unsigned long long SRC, unsigned long long*
BASE, const unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOXOR.D` instruction into the instruction stream. The memory address for the operation is specified by **`BASE`**. The constant operands of **`ORDERING`** are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address **`BASE`**

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
    //We want to perform an atomic xor operation.
   unsigned long long data, xorv, oldv;
   xorv = 0x22334455;
   // new value: data xor 0x22334455
   oldv = __nds__amoxord(xorv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amoandd`

Syntax

```
unsigned long __nds__amoandd(unsigned long long SRC, unsigned long long*
BASE, const unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOAND.D` instruction into the instruction stream. The memory address for the operation is specified by `BASE`. The constant operands of `ORDERING` are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address `BASE`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic AND operation.
  unsigned long long data, andv, oldv;
  andv = 0x22334455;
  // new value: data AND 0x22334455
  oldv = __nds__amoandd(andv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amoord`

Syntax

```
unsigned long __nds__amoord(unsigned long long SRC, unsigned long long*
BASE, const unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOOR.D` instruction into the instruction stream. The memory address for the operation is specified by `BASE`. The constant operands of `ORDERING` are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address `BASE`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic or operation.
  unsigned long long data, orv, oldv;
  orv = 0x22334455;
  // new value: data OR 0x22334455
  oldv = __nds__amoord(orv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amomind`

Syntax

`signed long __nds__amomind(signed long long SRC, signed long long* BASE, const unsigned int ORDERING)`

Description

This intrinsic inserts an `AMOMIN.D` instruction into the instruction stream. The memory address for the operation is specified by **`BASE`**. The constant operands of **`ORDERING`** are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address **`BASE`**

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
    //We want to perform an atomic min operation.
   signed long long data, cmpv, oldv;
   cmpv = 10;
   // new value: minimum(data, cmpv)
   oldv = __nds__amomind(cmpv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

__nds__amomaxd

Syntax

```
signed long __nds__amomaxd(signed long long SRC, signed long long* BASE,
const unsigned int ORDERING)
```

Description

This intrinsic inserts an AMOMAX.D instruction into the instruction stream. The memory address for the operation is specified by BASE. The constant operands of ORDERING are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address BASE

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
    //We want to perform an atomic max operation.
   signed long long data, cmpv, oldv;
   cmpv = 10;
   // new value: maximum(data, cmpv)
   oldv = __nds__amomaxd(cmpv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amominud`

Syntax

```
unsigned long __nds__amominud(unsigned long long SRC, unsigned long long*
BASE, const unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOMINU.D` instruction into the instruction stream. The memory address for the operation is specified by `BASE`. The constant operands of `ORDERING` are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address `BASE`

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic unsigned min operation.
  unsigned long long data, cmpv, oldv;
  cmpv = 0x22334455;
  // new value: unsigned minimum(data, cmpv)
  oldv = __nds__amominud(cmpv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__amomaxud`

Syntax

```
unsigned long __nds__amomaxud(unsigned long long SRC, unsigned long long*
BASE, const unsigned int ORDERING)
```

Description

This intrinsic inserts an `AMOMAXU.D` instruction into the instruction stream. The memory address for the operation is specified by **BASE**. The constant operands of **ORDERING** are defined in the following table:

unsigned int ORDERING	value
UNORDER	0
RELEASE	1
ACQUIRE	2
SEQUENTIAL	3

Return Value: The content of the memory address **BASE**

Privilege Level: ALL

Example

```
#include <nds_intrinsic.h>
void func(void)
{
   //We want to perform an atomic unsigned max operation.
  unsigned long long data, cmpv, oldv;
  cmpv = 0x22334455;
  // new value: unsigned maximum(data, cmpv)
  oldv = __nds__amomaxud(cmpv, &data, UNORDER);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

11. Andes PLIC intrinsic functions

Andes V5 toolchains provide intrinsic functions to access Andes Platform-Level Interrupt Controller (PLIC). The PLIC intrinsic functions are enabled with the symbols NDS_PLIC_BASE and NDS_PLIC_SW_BASE. To activate these intrinsic functions, you need to define and assign them to the PLIC/PLIC_SW base addresses before including the platform header file nds_v5_platform.h. For example,

```
#define NDS_PLIC_BASE     0xE4000000
#define NDS_PLIC_SW_BASE  0xE6400000
#include <nds_v5_platform.h>
```

11.1. Summary of Andes PLIC intrinsic functions

The tables below list Andes PLIC and PLIC SW intrinsic functions.

Table 19. PLIC Intrinsic Function Syntax

Intrinsic Function Syntax	Page
void __nds__plic_set_feature (unsigned int **FEATURE**)	119
void __nds__plic_set_threshold (unsigned int **THRESHOLD**)	120
void __nds__plic_set_priority (unsigned int **SOURCE**, unsigned int **PRIORITY**)	121
void __nds__plic_set_pending (unsigned int **SOURCE**)	122
void __nds__plic_enable_interrupt (unsigned int **SOURCE**)	123
Void __nds__plic_disable_interrupt (unsigned int **SOURCE**)	124
unsigned int __nds__plic_claim_interrupt(void)	125
void __nds__plic_complete_interrupt(unsigned int **SOURCE**)	126

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Table 20. PLIC SW Intrinsic Function Syntax

Intrinsic Function Syntax	Page
void __nds__plic_sw_set_threshold (unsigned int **THRESHOLD**)	127
void __nds__plic_sw_set_priority (unsigned int **SOURCE**, unsigned int **PRIORITY**)	128
void __nds__plic_sw_set_pending (unsigned int **SOURCE**)	129
void __nds__plic_sw_enable_interrupt (unsigned int **SOURCE**)	130
void __nds__plic_sw_disable_interrupt (unsigned int **SOURCE**)	131
unsigned int __nds__plic_sw_claim_interrupt(void)	132
void __nds__plic_sw_complete_interrupt(unsigned int **SOURCE**)	133

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

11.2. Detailed intrinsic function descriptions

11.2.1. Intrinsics for PLIC

Name

`__nds__plic_set_feature`

Syntax

```
void __nds__plic_set_feature (unsigned int FEATURE)
```

where the parameter

`FEATURE` is the feature that will be enabled for PLIC. It includes the following options:

- `NDS_PLIC_FEATURE_PREEMPT` to enable priority-based preemption
- `NDS_PLIC_FEATURE_VECTORED` to enable vectored mode for PLIC

Description

This function sets the feature for PLIC.

Return Value

None

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__plic_set_threshold`

Syntax

```
void __nds__plic_set_threshold (unsigned int THRESHOLD)
```

where the parameter

THRESHOLD is the interrupt priority threshold for PLIC. The greater the value is, the higher the threshold is.

Description

This function sets the interrupt priority threshold for PLIC.

Return Value

None

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__plic_set_priority`

Syntax

```
void __nds__plic_set_priority (unsigned int SOURCE, unsigned int PRIORITY)
```

where parameters

`SOURCE`	is the ID number of the specified interrupt.
`PRIORITY`	is the priority level.

Description

This function sets the priority level for the specified PLIC interrupt.

Return Value

None

Name

`__nds__plic_set_pending`

Syntax

```
void __nds__plic_set_pending (unsigned int SOURCE)
```

where the parameter

`SOURCE` is the ID number of the specified interrupt.

Description

This function sets the pending status for the specified PLIC interrupt.

Return Value

None

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__plic_enable_interrupt`

Syntax

```
void __nds__plic_enable_interrupt (unsigned int SOURCE)
```

where the parameter

SOURCE is the ID number of the specified interrupt.

Description

This function enables the specified PLIC interrupt.

Return Value

None

Name

`__nds__plic_disable_interrupt`

Syntax

```
void __nds__plic_disable_interrupt (unsigned int SOURCE)
```

where the parameter

`SOURCE` is the ID number of the specified interrupt.

Description

This function disables the specified PLIC interrupt.

Return Value

None

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__plic_claim_interrupt`

Syntax

```
unsigned int __nds__plic_claim_interrupt(void)
```

Description

This function gets the ID number of the claimed interrupt.

Return Value

The ID number of the claimed interrupt, which is read from PLIC

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__plic_complete_interrupt`

Syntax

```
void __nds__plic_complete_interrupt(unsigned int SOURCE)
```

where the parameter

SOURCE is the ID number of the specified interrupt.

Description

This function sets the specified interrupt as completed so that the next such interrupt can be forwarded to the PLIC and processed.

Return Value

None

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

11.2.2. Intrinsics for PLIC_SW

Name

`__nds__plic_sw_set_threshold`

Syntax

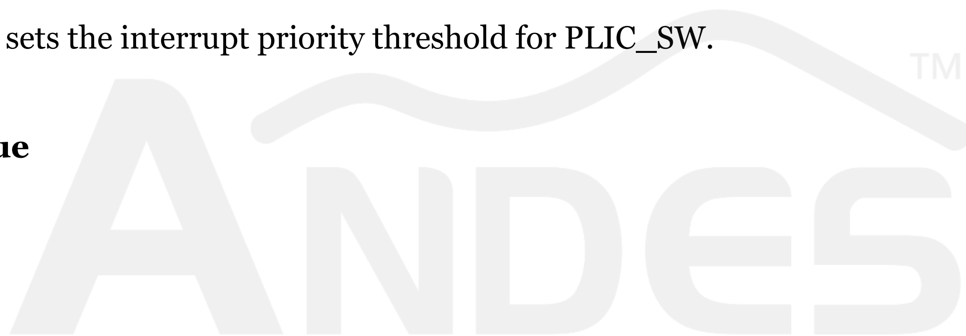
```
void __nds__plic_sw_set_threshold (unsigned int THRESHOLD)
```

where the parameter

THRESHOLD is the interrupt priority threshold for PLIC_SW. The larger the value is, the higher the threshold is.

Description

This function sets the interrupt priority threshold for PLIC_SW.



Return Value

None

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__plic_sw_set_priority`

Syntax

```
void __nds__plic_sw_set_priority (unsigned int SOURCE, unsigned int
PRIORITY)
```

where the parameters

`SOURCE`	is the ID number of the specified interrupt.
`PRIORITY`	is the priority level.

Description

This function sets the priority level for the specified PLIC_SW interrupt.

Return Value

None

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__plic_sw_set_pending`

Syntax

```
void __nds__plic_sw_set_pending (unsigned int SOURCE)
```

where the parameter

`SOURCE` is the ID number of the specified interrupt.

Description

This function sets the pending status for the specified PLIC_SW interrupt.

Return Value

None

Name

`__nds__plic_sw_enable_interrupt`

Syntax

```
void __nds__plic_sw_enable_interrupt (unsigned int SOURCE)
```

where the parameter

`SOURCE` is the ID number of the specified interrupt.

Description

This function enables the specified PLIC_SW interrupt.

Return Value

None

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__plic_sw_disable_interrupt`

Syntax

```
void __nds__plic_sw_disable_interrupt (unsigned int SOURCE)
```

where the parameter

SOURCE	is the ID number of the specified interrupt.

Description

This function disables the specified PLIC_SW interrupt.

Return Value

None

Name

`__nds__plic_sw_claim_interrupt`

Syntax

```
unsigned int __nds__plic_sw_claim_interrupt(void)
```

Description

This function gets the ID number of the claimed interrupt.

Return Value

The ID number of the claimed interrupt, which is read from PLIC_SW.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__plic_sw_complete_interrupt`

Syntax

```
void __nds__plic_sw_complete_interrupt(unsigned int SOURCE)
```

where the parameter

`SOURCE` is the ID number of the specified interrupt.

Description

This function sets the specified interrupt as completed so that the next such interrupt can be forwarded to the PLIC_SW and processed.

Return Value

None

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

12. Andes RVB and RVK intrinsic functions

Andes toolchains provide intrinsic functions for operations which the compiler cannot generate code to perform in RISC-V Bit-Manipulation (RVB) and Scalar Cryptography (RVK) extensions. To activate these intrinsic functions, you need to include the header file `nds_intrinsic.h` and compile your program with the option `-mext-zbabcs` for RVB and `-mext-zkns` for RVK.

Note that these RVB and RVK intrinsic functions have been deprecated since AndeSight v5.4.0 release and will no longer be available after a grace period of two major AndeSight releases. It is recommended to replace them with the official RISC-V intrinsic functions outlined in Section 12.2 and Section 12.4. When using the official RISC-V intrinsic replacements, ensure that you include an associated header file (i.e., `riscv_bitmanip.h` or `riscv_crypto.h`). For more information about RISC-V intrinsic functions for RVB and Scalar Cryptography extensions, see the official documentation RISC-V C API; for details about Vector Cryptography extensions, find them in Andes Vector Intrinsics Document on the Andes GitHub repository.

Both Andes GCC and LLVM compilers emit warnings when these deprecated intrinsics are used. To suppress the warnings, compile your program with the option "`-D_NDS_SUPPRESS_DEPRECATED`".

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

12.1. Summary of Andes RVB intrinsic functions

The table below lists Andes intrinsic functions for RVB extensions and indicates whether they are supported for RV32 and RV64.

Table 21. RVB Intrinsic Function Syntax

Intrinsic Function Syntax	RV32	RV64	Page
`int __nds__clz_32(int SRC)`	√	√	136
`int __nds__ctz_32(int SRC)`	√	√	137
`int __nds__cpop_32(unsigned int SRC)`	√	√	138
`long __nds__clmul(long SRC1, long SRC2)`	√	√	139
`long __nds__clmulh(long SRC1, long SRC2)`	√	√	140
`long __nds__clmulr(long SRC1, long SRC2)`	√	√	141
`long __nds__clz_64(long SRC)`		√	142
`long __nds__ctz_64(long SRC)`		√	143
`int __nds__cpop_64(unsigned long SRC)`		√	144

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

12.2. Detailed descriptions of Andes RVB intrinsic functions

12.2.1. Intrinsics for RV32 and RV64 RVB extension

Name

`__nds__clz_32`

Syntax

`int __nds__clz_32(int SRC)`

where the parameter

`SRC` is a source operand to determine the return value.

Official RISC-V intrinsic replacement

`unsigned __riscv_clz_32(uint32_t SRC)`

This requires including the header file `riscv_bitmanip.h`.

Description

This function counts the number of 0 before the first 1, starting from bit 31 and progressing to bit 0. Accordingly, if the input is 0, the output is 32; if the most-significant bit of the input is 1, the output is 0.

Return Value

The return value is between 0 and 32.

Example

```
#include <nds_intrinsic.h>
int clz_32(int rs1)
{
    return __nds__clz_32(rs1);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__ctz_32`

Syntax

```
int __nds__ctz_32(int SRC)
```

where the parameter

`SRC` is a source operand to determine the return value.

Official RISC-V intrinsic replacement

```
unsigned __riscv_ctz_32(uint32_t SRC)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function counts the number of 0 before the first 1, starting from bit 0 and progressing to bit 31. Accordingly, if the input is 0, the output is 32; if the least-significant bit of the input is 1, the output is 0.

Return Value

The return value is between 0 and 32.

Example

```
#include <nds_intrinsic.h>
int ctz_32(int rs1)
{
    return __nds__ctz_32(rs1);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__cpop_32`

Syntax

```
int __nds__cpop_32(unsigned int SRC)
```

where the parameter

`SRC` is a source operand to determine the return value.

Official RISC-V intrinsic replacement

```
unsigned __riscv_cpop_32(uint32_t SRC)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function counts the number of 1 (i.e., set bits) in `SRC`.

Return Value

The return value is the number of 1 (i.e., set bits) in `SRC`.

Example

```
#include <nds_intrinsic.h>
int cpop_32(int rs1)
{
    return __nds__cpop_32(rs1);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__clmul`

Syntax

```
long __nds__clmul(long SRC1, long SRC2)
```

where the parameters

SRC1 and SRC2 are two source operands to determine the return value.

Official RISC-V intrinsic replacement

RV32:

```
uint32_t __riscv_clmul_32(uint32_t SRC1, uint32_t SRC2)
```

RV64:

```
uint64_t __riscv_clmul_64(uint64_t SRC1, uint64_t SRC2)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function is a carry-less multiplication function that produces the lower half of the 2·XLEN carry-less product.

Return Value

The return value is the lower half of the 2·XLEN carry-less product.

Example

```
#include <nds_intrinsic.h>
long clmul(long a, long b)
{
    return __nds__clmul(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__clmulh`

Syntax

`long __nds__clmulh(long SRC1, long SRC2)`

where the parameters

`SRC1` and `SRC2` are two source operands to determine the return value.

Official RISC-V intrinsic replacement

RV32:

```
uint32_t __riscv_clmulh_32(uint32_t SRC1, uint32_t SRC2)
```

RV64:

```
uint64_t __riscv_clmulh_64(uint64_t SRC1, uint64_t SRC2)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function is a carry-less multiplication function that produces the upper half of the 2·XLEN carry-less product.

Return Value

The return value is the upper half of the 2·XLEN carry-less product.

Example

```
#include <nds_intrinsic.h>
long clmulh(long a, long b)
{
    return __nds__clmulh(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__clmulr`

Syntax

```
long __nds__clmulr(long SRC1, long SRC2)
```

where the parameters

`SRC1` and `SRC2` are two source operands to determine the return value.

Official RISC-V intrinsic replacement

RV32:

```
uint32_t __riscv_clmulr_32(uint32_t SRC1, uint32_t SRC2)
```

RV64:

```
uint64_t __riscv_clmulr_64(uint64_t SRC1, uint64_t SRC2)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This is a carry-less multiplication function that produces bits 2·XLEN−2:XLEN-1 of the 2·XLEN carry-less product. It is used to accelerate CRC calculations. The `r` in the function name stands for "reversed", as the function is equivalent to bit-reversing the inputs, performing a clmul operation, and then bit-reversing the output.

Return Value

The return value is the bits 2·XLEN−2:XLEN-1 of the 2·XLEN carry-less product.

Example

```
#include <nds_intrinsic.h>
long clmulr(long a, long b)
{
    return __nds__clmulr(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

12.2.2. Intrinsics for RV64-only RVB extension

Name

`__nds__clz_64`

Syntax

`long __nds__clz_64(long SRC)`

where the parameter

`SRC` is a source operand to determine the return value.

Official RISC-V intrinsic replacement

`unsigned __riscv_clz_64(uint64_t SRC)`

This requires including the header file `riscv_bitmanip.h`.

Description

This function counts the number of 0 before the first 1, starting from the most-significant bit (i.e., XLEN-1) and progressing to bit 0. Accordingly, if the input is 0, the output is XLEN; if the most-significant bit of the input is 1, the output is 0.

Return Value

The return value is between 0 and XLEN.

Example

```
#include <nds_intrinsic.h>
long clz_64(long rs1)
{
    return __nds__clz_64(rs1);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__ctz_64`

Syntax

`long __nds__ctz_64(long SRC)`

where the parameter

`SRC` is a source operand to determine the return value.

Official RISC-V intrinsic replacement

`unsigned __riscv_ctz_64(uint64_t SRC)`

This requires including the header file `riscv_bitmanip.h`.

Description

This function counts the number of 0 before the first 1, starting from the least-significant bit (i.e., 0) and progressing to the most-significant bit (i.e., XLEN-1). Accordingly, if the input is 0, the output is XLEN; if the least-significant bit of the input is 1, the output is 0.

Return Value

The return value is between 0 and XLEN.

Example

```
#include <nds_intrinsic.h>
long ctz_64(long rs1)
{
    return __nds__ctz_64(rs1);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__cpop_64`

Syntax

```
int __nds__cpop_64(unsigned long SRC)
```

where the parameter

`SRC` is a source operand to determine the return value.

Official RISC-V intrinsic replacement

```
unsigned __riscv_cpop_64(uint64_t SRC)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function counts the number of 1 (i.e., set bits) in the least-significant word of `SRC`.

Return Value

The return value is between 0 and XLEN.

Example

```
#include <nds_intrinsic.h>
int cpop_64(unsigned long rs1)
{
    return __nds__cpop_64(rs1);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

12.3. Summary of Andes RVK intrinsic functions

The table below lists Andes intrinsic functions for RVK extensions and indicates whether they are supported for RV32 and RV64.

Table 22. RVK Intrinsic Function Syntax

Intrinsic Function Syntax	RV32	RV64	Page
long __nds__clmul(long SRC1, long SRC2)	√	√	147
long __nds__clmulh(long SRC1, long SRC2)	√	√	148
long __nds__clmulr(long SRC1, long SRC2)	√	√	149
long __nds__xperm4(long SRC1, long SRC2)	√	√	150
long __nds__xperm8(long SRC1, long SRC2)	√	√	151
long __nds__brev8(long SRC)	√	√	152
int __nds__zip(int SRC)	√		161
int __nds__unzip(int SRC)	√		162
int __nds__aes32dsi(int SRC1, int SRC2, unsigned char BS)	√		163
int __nds__aes32dsmi(int SRC1, int SRC2, unsigned char BS)	√		164
int __nds__aes32esi(int SRC1, int SRC2, unsigned char BS)	√		165
int __nds__aes32esmi(int SRC1, int SRC2, unsigned char BS)	√		166
long __nds__aes64ds(long SRC1, long SRC2)		√	173
long __nds__aes64dsm(long SRC1, long SRC2)		√	174
long __nds__aes64es(long SRC1, long SRC2)		√	175
long __nds__aes64esm(long SRC1, long SRC2)		√	176
long __nds__aes64im(long SRC)		√	177
long __nds__aes64ks1i(long SRC, unsigned int RNUM)		√	178
long __nds__aes64ks2(long SRC1, long SRC2)		√	179
long __nds__sha256sig0(long SRC)	√	√	153
long __nds__sha256sig1(long SRC)	√	√	154
long __nds__sha256sum0(long SRC)	√	√	155
long __nds__sha256sum1(long SRC)	√	√	156
int __nds__sha512sig0h(int SRC1, int SRC2)	√		167

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

int __nds__sha512sig0l(int SRC1, int SRC2)	√		168
int __nds__sha512sig1h(int SRC1, int SRC2)	√		169
int __nds__sha512sig1l(int SRC1, int SRC2)	√		170
int __nds__sha512sum0r(int SRC1, int SRC2)	√		171
int __nds__sha512sum1r(int SRC1, int SRC2)	√		172
long __nds__sha512sig0(long SRC)		√	180
long __nds__sha512sig1(long SRC)		√	181
long __nds__sha512sum0(long SRC)		√	182
long __nds__sha512sum1(long SRC)		√	183
long __nds__sm4ed(long SRC1, long SRC2, unsigned char BS)	√	√	157
long __nds__sm4ks(long SRC1, long SRC2, unsigned char BS)	√	√	158
long __nds__sm3p0(long SRC)	√	√	159
long __nds__sm3p1(long SRC)	√	√	160

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

12.4. Detailed descriptions of Andes RVK intrinsic functions

12.4.1. Intrinsics for RV32 and RV64 RVK extension

Name

`__nds__clmul`

Syntax

```
long __nds__clmul(long SRC1, long SRC2)
```

where the parameters

`SRC1` and `SRC2` are two source operands to determine the return value.

Official RISC-V intrinsic replacement

RV32:

```
uint32_t __riscv_clmul_32(uint32_t SRC1, uint32_t SRC2)
```

RV64:

```
uint64_t __riscv_clmul_64(uint64_t SRC1, uint64_t SRC2)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function is a carry-less multiplication function that produces the lower half of the 2·XLEN carry-less product.

Return Value

The return value is the lower half of the 2·XLEN carry-less product.

Example

```
#include <nds_intrinsic.h>
long clmul(long a, long b)
{
    return __nds__clmul(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__clmulh`

Syntax

```
long __nds__clmulh(long SRC1, long SRC2)
```

where the parameters

`SRC1` and `SRC2` are two source operands to determine the return value.

Official RISC-V intrinsic replacement

RV32:

```
uint32_t __riscv_clmulh_32(uint32_t SRC1, uint32_t SRC2)
```

RV64:

```
uint64_t __riscv_clmulh_64(uint64_t SRC1, uint64_t SRC2)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function is a carry-less multiplication function that produces the upper half of the 2·XLEN carry-less product.

Return Value

The return value is the upper half of the 2·XLEN carry-less product.

Example

```
#include <nds_intrinsic.h>
long clmulh(long a, long b)
{
    return __nds__clmulh(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__clmulr`

Syntax

```
long __nds__clmulr(long SRC1, long SRC2)
```

where the parameters

`SRC1` and `SRC2` are two source operands to determine the return value.

Official RISC-V intrinsic replacement

RV32:

```
uint32_t __riscv_clmulr_32(uint32_t SRC1, uint32_t SRC2)
```

RV64:

```
uint64_t __riscv_clmulr_64(uint64_t SRC1, uint64_t SRC2)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This is a carry-less multiplication function that produces bits 2·XLEN−2:XLEN-1 of the 2·XLEN carry-less product. It is used to accelerate CRC calculations. The `r` in the function name stands for “reversed”, as the function is equivalent to bit-reversing the inputs, performing a clmul operation, and then bit-reversing the output.

Return Value

The return value is the bits 2·XLEN−2:XLEN-1 of the 2·XLEN carry-less product.

Example

```
#include <nds_intrinsic.h>
long clmulr(long a, long b)
{
    return __nds__clmulr(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__xperm4`

Syntax

```
long __nds__xperm4(long SRC1, long SRC2)
```

where the parameters

`SRC1` and `SRC2` are two source registers for the operation.

RISC-V intrinsic replacement

RV32:

```
uint32_t __riscv_xperm4_32(uint32_t SRC1, uint32_t SRC2)
```

RV64:

```
uint64_t __riscv_xperm4_64(uint64_t SRC1, uint64_t SRC2)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function operates on nibbles. The `SRC1` register contains a vector of XLEN/4 4-bit elements. The `SRC2` register contains a vector of XLEN/4 4-bit indexes. The result is each element in `SRC2` replaced by the indexed element in `SRC1`, or zero if the index into `SRC2` is out of bounds.

Example

```
#include <nds_intrinsic.h>
long xperm4(long a, long b)
{
    return __nds__xperm4(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__xperm8`

Syntax

```
long __nds__xperm8(long SRC1, long SRC2)
```

where the parameters

SRC1 and SRC2 are two source registers for the operation.

Official RISC-V intrinsic replacement

RV32:

```
uint32_t __riscv_xperm8_32(uint32_t SRC1, uint32_t SRC2)
```

RV64:

```
uint64_t __riscv_xperm8_64(uint64_t SRC1, uint64_t SRC2)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function operates on bytes. The SRC1 register contains a vector of XLEN/8 8-bit elements. The SRC2 register contains a vector of XLEN/8 8-bit indexes. The result is each element in SRC2 replaced by the indexed element in SRC1, or zero if the index into SRC2 is out of bounds.

Example

```
#include <nds_intrinsic.h>
long xperm8(long a, long b)
{
    return __nds__xperm8(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__brev8`

Syntax

```
long __nds__brev8(long SRC)
```

where the parameter

`SRC` is a source register for the operation.

Official RISC-V intrinsic replacement

RV32:

```
uint32_t __riscv_brev8_32(uint32_t SRC)
```

RV64:

```
uint64_t __riscv_brev8_64(uint64_t SRC)
```

This requires including the header file `riscv_bitmanip.h`.

Description

This function reverses the bits in each byte of the `SRC` register.

Example

```
#include <nds_intrinsic.h>
long brev8(long a)
{
    return __nds__brev8(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha256sig0`

Syntax

```
long __nds__sha256sig0(long SRC)
```

where the parameter

`SRC` is a source register for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sha256sig0(uint32_t SRC)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the Sigma0 transformation function used in the SHA2-256 hash function for the SRC register. For RV32, it operates on the entire XLEN of the SRC register. For RV64, it operates on the low 32 bits of SRC and the result is sign extended to XLEN bits.

Example

```
#include <nds_intrinsic.h>
long __nds__sha256sig0(long a)
{
    return __nds__sha256sig0(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha256sig1`

Syntax

```
long __nds__sha256sig1(long SRC)
```

where the parameter

`SRC` is a source register for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sha256sig1(uint32_t SRC)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the Sigma1 transformation function used in the SHA2-256 hash function for the `SRC` register. For RV32, it operates on the entire XLEN of the `SRC` register. For RV64, it operates on the low 32 bits of `SRC` and the result is sign extended to XLEN bits.

Example

```
#include <nds_intrinsic.h>
long __nds__sha256sig1(long a)
{
    return __nds__sha256sig1(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha256sum0`

Syntax

`long __nds__sha256sum0(long SRC)`

where the parameter

SRC is a source register for the operation.

Official RISC-V intrinsic replacement

`uint32_t __riscv_sha256sum0(uint32_t SRC)`

This requires including the header file `riscv_crypto.h`.

Description

This function implements the Sum0 transformation function used in the SHA2-256 hash function for the **SRC** register. For RV32, it operates on the entire XLEN of the **SRC** register. For RV64, it operates on the low 32 bits of **SRC** and the result is sign extended to XLEN bits.

Example

```
#include <nds_intrinsic.h>
long __nds__sha256sum0(long a)
{
    return __nds__sha256sum0(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha256sum1`

Syntax

```
long __nds__sha256sum1(long SRC)
```

where the parameter

`SRC` is a source register for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sha256sum1(uint32_t SRC)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the Sum1 transformation function used in the SHA2-256 hash function for the `SRC` register. For RV32, it operates on the entire XLEN of the `SRC` register. For RV64, it operates on the low 32 bits of `SRC` and the result is sign extended to XLEN bits.

Example

```
#include <nds_intrinsic.h>
long __nds__sha256sum1(long a)
{
    return __nds__sha256sum1(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sm4ed`

Syntax

`long __nds__sm4ed(long SRC1, long SRC2, unsigned char BS)`

where the parameters

`SRC1`, `SRC2` and `BS` are three source operands for the operation.

Official RISC-V intrinsic replacement

`uint32_t __riscv_sm4ed(uint32_t SRC1, uint32_t SRC2, const int BS)`

This requires including the header file `riscv_crypto.h`.

Description

This function implements T-tables in hardware style approach to accelerate the SM4 round function. It performs a block encrypt/decrypt operation on SM4 block ciphers. A byte is extracted from `SRC2` based on `BS`, to which the SBox and linear layer transforms are applied, before the result is XORed with `SRC1` and written back to the destination register. On RV64, the 32-bit result is sign extended to XLEN bits.

Example

```
#include <nds_intrinsic.h>
long sm4ed(long a, long b)
{
    return __nds__sm4ed(a, b, 0);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sm4ks`

Syntax

`long __nds__sm4ks(long SRC1, long SRC2, unsigned char BS)`

where the parameters

`SRC1` , `SRC2` and `BS` are three source operands for the operation.

Official RISC-V intrinsic replacement

`uint32_t __riscv_sm4ks(uint32_t SRC1, uint32_t SRC2, const int BS)`

This requires including the header file `riscv_crypto.h`.

Description

This function implements T-tables in hardware style approach to accelerate the SM4 Key Schedule. It performs the Key Schedule operation on SM4 block ciphers. A byte is extracted from `SRC2` based on `BS`, to which the SBox and linear layer transforms are applied, before the result is XORed with `SRC1` and written back to the destination register.

Example

```
#include <nds_intrinsic.h>
long sm4ks(long a, long b)
{
    return __nds__sm4ks(a, b, 0);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sm3p0`

Syntax

```
long __nds__sm3p0(long SRC)
```

where the parameter

`SRC` is a source operand for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sm3p0(uint32_t SRC)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the P0 transformation function used in the SM3 hash function for `SRC`.

Example

```
#include <nds_intrinsic.h>
long sm3p0(long a)
{
    return __nds__sm3p0(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sm3p1`

Syntax

```
long __nds__sm3p1(long SRC)
```

where the parameter

`SRC` is a source operand for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sm3p1(uint32_t SRC)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the P1 transformation function used in the SM3 hash function for `SRC`.

Example

```
#include <nds_intrinsic.h>
long sm3p1(long a)
{
    return __nds__sm3p1(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

12.4.2. Intrinsics for RV32-only RVK extension

Name

`__nds__zip`

Syntax

`int __nds__zip(int SRC)`

where the parameter

`SRC` is a source register for the operation.

Official RISC-V intrinsic replacement

`uint32_t __riscv_zip_32(uint32_t SRC)`

This requires including the header file `riscv_bitmanip.h`.

Description

This function places the upper half of the source register **SRC** into odd bits of the destination and the lower half of the source register **SRC** into even bits of the destination.

Example

```
#include <nds_intrinsic.h>
int zip(int a)
{
    return __nds__zip(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__unzip`

Syntax

`int __nds__unzip(int SRC)`

where the parameter

`SRC` is a source register for the operation.

Official RISC-V intrinsic replacement

`uint32_t __riscv_unzip_32(uint32_t SRC)`

This requires including the header file `riscv_bitmanip.h`.

Description

This function places the even bits of `SRC` into the low half of the destination and the odd bits of `SRC` into the high bits of the destination.

Example

```
#include <nds_intrinsic.h>
int unzip(int a)
{
    return __nds__unzip(a);
}
```

Name

__nds__aes32dsi

Syntax

```
int __nds__aes32dsi(int SRC1, int SRC2, unsigned char BS)
```

where the parameters

SRC1, **SRC2** and **BS** are three source operands for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_aes32dsi(uint32_t SRC1, uint32_t SRC2, const int BS)
```

This requires including the header file riscv_crypto.h.

Description

This function sources a single byte from **SRC2** according to **BS**, applies the inverse AES SBox operation to it, and XORs the result with **SRC1**.

Example

```
#include <nds_intrinsic.h>
int aes32dsi(int a, int b)
{
    return __nds__aes32dsi(a, b, 3);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__aes32dsmi`

Syntax

```
int __nds__aes32dsmi(int SRC1, int SRC2, unsigned char BS)
```

where the parameters

`SRC1`, `SRC2` and `BS` are three source operands for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_aes32dsmi(uint32_t SRC1, uint32_t SRC2, const int BS)
```

This requires including the header file `riscv_crypto.h`.

Description

This function sources a single byte from `SRC2` according to `BS`, applies the inverse AES SBox operation and a partial inverse MixColumn to it, and then XORs the result with `SRC1`.

Example

```
#include <nds_intrinsic.h>
int aes32dsmi(int a, int b)
{
    return __nds__aes32dsmi(a, b, 3);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__aes32esi`

Syntax

```
int __nds__aes32esi(int SRC1, int SRC2, unsigned char BS)
```

where the parameters

SRC1, **SRC2** and **BS** are three source operands for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_aes32esi(uint32_t SRC1, uint32_t SRC2, const int BS)
```

This requires including the header file `riscv_crypto.h`.

Description

This function sources a single byte from **SRC2** according to **BS**, applies the forward AES SBox operation to it, and then XORs the result with **SRC1**.

Example

```
#include <nds_intrinsic.h>
int aes32esi(int a, int b)
{
    return __nds__aes32esi(a, b, 3);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__aes32esmi`

Syntax

```
int __nds__aes32esmi(int SRC1, int SRC2, unsigned char BS)
```

where the parameters

`SRC1`, `SRC2` and `BS` are three source operands for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_aes32esim(uint32_t SRC1, uint32_t SRC2, const int BS)
```

This requires including the header file `riscv_crypto.h`.

Description

This function sources a single byte from `SRC2` according to `BS`, applies the forward AES SBox operation and a partial forward MixColumn to it, and then XORs the result with `SRC1`.

Example

```
#include <nds_intrinsic.h>
int aes32esmi(int a, int b)
{
    return __nds__aes32esmi(a, b, 3);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha512sig0h`

Syntax

```
int __nds__sha512sig0h(int SRC1, int SRC2)
```

where the parameters

`SRC1` and `SRC2` are two source operands for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sha512sig0h(uint32_t SRC1, uint32_t SRC2)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the high half of the Sigma0 transformation used in the SHA2-512 hash function for `SRC1` and `SRC2`. It computes the Sigma0 transformation of the SHA2-512 hash function in conjunction with the `sha512sig0l` function.

Example

```
#include <nds_intrinsic.h>
int sha512sig0h(int a, int b)
{
    return __nds__sha512sig0h(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha512sig0l`

Syntax

`int __nds__sha512sig0l(int SRC1, int SRC2)`

where the parameters

`SRC1` and `SRC2` are two source operands for the operation.

Official RISC-V intrinsic replacement

`uint32_t __riscv_sha512sig0l(uint32_t SRC1, uint32_t SRC2)`

This requires including the header file `riscv_crypto.h`.

Description

This function implements the low half of the Sigma0 transformation used in the SHA2-512 hash function for `SRC1` and `SRC2`. It computes the Sigma0 transformation of the SHA2-512 hash function in conjunction with the `sha512sig0h` instruction.

Example

```
#include <nds_intrinsic.h>
int sha512sig0l(int a, int b)
{
    return __nds__sha512sig0l(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha512sig1h`

Syntax

```
int __nds__sha512sig1h(int SRC1, int SRC2)
```

where the parameters

`SRC1` and `SRC2` are two source operands for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sha512sig1h(uint32_t SRC1, uint32_t SRC2)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the high half of the Sigma1 transformation used in the SHA2-512 hash function for `SRC1` and `SRC2`. It computes the Sigma1 transformation of the SHA2-512 hash function in conjunction with the `sha512sig1l` instruction.

Example

```
#include <nds_intrinsic.h>
int sha512sig1h(int a, int b)
{
    return __nds__sha512sig1h(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

__nds__sha512sig1l

Syntax

```
int __nds__sha512sig1l(int SRC1, int SRC2)
```

where the parameters

SRC1 and SRC2 are two source operands for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sha512sig1l(uint32_t SRC1, uint32_t SRC2)
```

This requires including the header file riscv_crypto.h.

Description

This function implements the low half of the Sigma1 transformation used in the SHA2-512 hash function for SRC1 and SRC2. It computes the Sigma1 transformation of the SHA2-512 hash function in conjunction with the sha512sig1h instruction.

Example

```
#include <nds_intrinsic.h>
int sha512sig1l(int a, int b)
{
    return __nds__sha512sig1l(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

__nds__sha512sum0r

Syntax

```
int __nds__sha512sum0r(int SRC1, int SRC2)
```

where the parameters

SRC1 and SRC2 are two source operands for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sha512sum0r(uint32_t SRC1, uint32_t SRC2)
```

This requires including the header file riscv_crypto.h.

Description

This function implements the Sum0 transformation used in the SHA2-512 hash function for SRC1 and SRC2.

Example

```
#include <nds_intrinsic.h>
int sha512sum0r(int a, int b)
{
    return __nds__sha512sum0r(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha512sum1r`

Syntax

```
int __nds__sha512sum1r(int SRC1, int SRC2)
```

where the parameter

`SRC1` and `SRC2` are two source operands for the operation.

Official RISC-V intrinsic replacement

```
uint32_t __riscv_sha512sum1r(uint32_t SRC1, uint32_t SRC2)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the Sum1 transformation used in the SHA2-512 hash function for `SRC1` and `SRC2`.

Example

```
#include <nds_intrinsic.h>
int sha512sum1r(int a, int b)
{
    return __nds__sha512sum1r(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

12.4.3. Intrinsics for RV64-only RVK extension

Name

`__nds__aes64ds`

Syntax

`long __nds__aes64ds(long SRC1, long SRC2)`

where the parameters

`SRC1` and `SRC2` are two source registers for the operation.

Official RISC-V intrinsic replacement

`uint64_t __riscv_aes64ds(uint64_t SRC1, uint64_t SRC2)`

This requires including the header file `riscv_crypto.h`.

Description

This function uses the two 64-bit source registers `SRC1` and `SRC2` to represent the entire AES state, and produces half of the next round output by applying the Inverse ShiftRows and SubBytes steps.

Example

```
#include <nds_intrinsic.h>
long aes64ds(long a, long b)
{
    return __nds__aes64ds(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__aes64dsm`

Syntax

`long __nds__aes64dsm(long SRC1, long SRC2)`

where the parameters

`SRC1` and `SRC2` are two source registers for the operation.

Official RISC-V intrinsic replacement

`uint64_t __riscv_aes64dsm(uint64_t SRC1, uint64_t SRC2)`

This requires including the header file `riscv_crypto.h`.

Description

This function uses the two 64-bit source registers `SRC1` and `SRC2` to represent the entire AES state, and produces half of the next round output by applying the Inverse ShiftRows, SubBytes and MixColumns steps.

Example

```
#include <nds_intrinsic.h>
long aes64dsm(long a, long b)
{
    return __nds__aes64dsm(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__aes64es`

Syntax

`long __nds__aes64es(long SRC1, long SRC2)`

where the parameters

`SRC1` and `SRC2` are two source registers for the operation.

Official RISC-V intrinsic replacement

`uint64_t __riscv_aes64es(uint64_t SRC1, uint64_t SRC2)`

This requires including the header file `riscv_crypto.h`.

Description

This function uses the two 64-bit source registers `SRC1` and `SRC2` to represent the entire AES state, and produces half of the next round output by applying the ShiftRows and SubBytes steps.

Example

```
#include <nds_intrinsic.h>
long aes64es(long a, long b)
{
    return __nds__aes64es(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__aes64esm`

Syntax

`long __nds__aes64esm(long SRC1, long SRC2)`

where the parameters

`SRC1` and `SRC2` are two source registers for the operation.

Official RISC-V intrinsic replacement

`uint64_t __riscv_aes64esm(uint64_t SRC1, uint64_t SRC2)`

This requires including the header file `riscv_crypto.h`.

Description

This function uses the two 64-bit source registers `SRC1` and `SRC2` to represent the entire AES state, and produces half of the next round output by applying the ShiftRows, SubBytes and MixColumns steps.

Example

```
#include <nds_intrinsic.h>
long aes64esm(long a, long b)
{
    return __nds__aes64esm(a, b);
}
```

Name

`__nds__aes64im`

Syntax

`long __nds__aes64im(long SRC)`

where the parameter

`SRC` is a source register for the operation.

Official RISC-V intrinsic replacement

`uint64_t __riscv_aes64im(uint64_t SRC)`

This requires including the header file `riscv_crypto.h`.

Description

This function applies the inverse MixColumns transformation to two columns of the state array packed into the 64-bit register `SRC`. It accelerates the inverse MixColumns step of the AES block cipher, and aids creation of the decryption Key Schedule.

Example

```
#include <nds_intrinsic.h>
long aes64im(long a)
{
    return __nds__aes64im(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__aes64ks1i`

Syntax

```
long __nds__aes64ks1i(long SRC, unsigned int RNUM)
```

where the parameters

SRC and **RNUM** are operands for the operation.

Official RISC-V intrinsic replacement

```
uint64_t __riscv_aes64ks1i(uint64_t SRC, const int RNUM)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the rotation, SubBytes and Round Constant addition steps of the AES block cipher Key Schedule.

Example

```
#include <nds_intrinsic.h>
long aes64ks1i(long a)
{
    return __nds__aes64ks1i(a, 0);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__aes64ks2`

Syntax

```
long __nds__aes64ks2(long SRC1, long SRC2)
```

where the parameters

`SRC1` and `SRC2` are two operands for the operation.

Official RISC-V intrinsic replacement

```
uint64_t __riscv_aes64ks2(uint64_t SRC1, uint64_t SRC2)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the additional XORing of key words as part of the AES block cipher Key Schedule.

Example

```
#include <nds_intrinsic.h>
long aes64ks2(long a, long b)
{
    return __nds__aes64ks2(a, b);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha512sig0`

Syntax

```
long __nds__sha512sig0(long SRC)
```

where the parameter

`SRC` is a source register for the operation.

Official RISC-V intrinsic replacement

```
uint64_t __riscv_sha512sig0(uint64_t SRC)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the Sigma0 transformation function used in the SHA2-512 hash function for the `SRC` register.

Example

```
    #include <nds_intrinsic.h>
    long sha512sig0(long a)
    {
        return __nds__sha512sig0(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha512sig1`

Syntax

```
long __nds__sha512sig1(long SRC)
```

where the parameter

SRC is a source register for the operation.

Official RISC-V intrinsic replacement

```
uint64_t __riscv_sha512sig1(uint64_t SRC)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the Sigma1 transformation function used in the SHA2-512 hash function for the **SRC** register.

Example

```
#include <nds_intrinsic.h>
long sha512sig1(long a)
{
    return __nds__sha512sig1(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha512sum0`

Syntax

```
long __nds__sha512sum0(long SRC)
```

where the parameter

`SRC` is a source register for the operation.

Official RISC-V intrinsic replacement

```
uint64_t __riscv_sha512sum0(uint64_t SRC)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the Sum0 transformation function used in the SHA2-512 hash function for the `SRC` register.

Example

```
#include <nds_intrinsic.h>
long sha512sum0(long a)
{
    return __nds__sha512sum0(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Name

`__nds__sha512sum1`

Syntax

```
long __nds__sha512sum1(long SRC)
```

where the parameter

SRC is a source register for the operation.

Official RISC-V intrinsic replacement

```
uint64_t __riscv_sha512sum1(uint64_t SRC)
```

This requires including the header file `riscv_crypto.h`.

Description

This function implements the Sum1 transformation function used in the SHA2-512 hash function for the **SRC** register.

Example

```
#include <nds_intrinsic.h>
long sha512sum1(long a)
{
    return __nds__sha512sum1(a);
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

13. Linker script generation

While GNU linker has a complicated language to specify the image format, Andes offers a rather simple mechanism for you to specify the memory map and generate the linker script. By following Andes-defined SaG (Scattering-and-Gathering) format, you can easily create a description file about image component arrangement required to generate a linker script using the command option `nds_ldsag`. The following sections give detailed introduction to SaG script format and Andes linker script generator LdSaG (`nds_ldsag`).

13.1. Script format SaG and its syntax

SaG (Scattering-and-Gathering) is an Andes-defined script format for describing the memory map of an application to the linker. With the file extension .sag, a SaG-formatted description file can specify:

- the load memory address (LMA).
- the attributes and maximum size of each load region.
- the virtual memory address (VMA), which is also the execution address.
- the attributes and maximum size of each execution region.
- the input sections for each execution region.

13.1.1. BNF notation for SaG syntax

The table below summarizes the BNF symbols that are used to describe the SaG syntax.

Symbol	Description
"	It is used to indicate a character is used as its literal character. For example, the definition A“+”B can only be replaced by the pattern A+B while the definition A+B can be replaced by patterns AB, AAB, or AAAB.
A ::= B	Defines A as B. The ::= notation means “is defined as”. Thus, A::= B"+", for example, means that A is equivalent to B+.
[A]	Optional element A. For example, [A] can be A or “NULL”.
A+	Element A can have one or more occurrences. Thus, A+ can be A, AA , or AAA....

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Symbol	Description
A*	Element A can have zero or more occurrences. Thus, A* can be “NULL”, A, AA , or AAA….
A | B	Either element A or B can occur, but not both. The | notation means “or”.
(A B)	The () notation stands for “grouping”. Therefore, (AB) means element A and B are grouped together. That is, both A and B have to occur and can be regarded as one unit.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

13.1.2. Formal syntax of SaG format

13.1.2.1 Overview

The SaG-formatted script is constructed by the hierarchy of load regions, execution regions and input sections. To start with, follow below to define a script as one or more `load_region_description` patterns:

```
ld_script ::= [header] load_region_description+
[assert_information]

header ::= ( ("USER_SECTIONS" section_name ("," section_name)*)
           | ("DEFINE" variable_name expression)
           | ("INCLUDE" '"' file_name '"')
           | ("ENTRY" "(" symbol_name ")" )
           | ("EXTERN" "(" symbol_name+ ")" )
           | ("#include" '"' file_name '"')
         )*

assert_information ::= ("SCATTERASSERT"  assert_description)*
```

Header

`USER_SECTIONS`, `DEFINE`, `INCLUDE`, `ENTRY` and `EXTERN` are keywords in `header` and must be upper-cased.

- `USER_SECTIONS`

If there is any user-defined section used in your source files and the section is not defined in generic linker script, you have to use the keyword `USER_SECTIONS` to declare the section in `header`. Otherwise, LdSaG (`nds_ldsag`) will show a warning message after compilation. For example, if you've defined a section `.my_section` in the assembly file as follows,

```
.section .my_section, "ax"
```

you have to declare the section in the SaG-formatted script like below:

```
USER_SECTIONS .my_section
LOAD 0x00100000
{
    EXEC +0x00000000
    {
        * (+RO, .my_section)
        * (+RW,+ZI)
```

```
            STACK = 0x00700000
        }
    }
```

- DEFINE

DEFINE is another form of header. It is used to define a macro and its value. As for expression, it must be presented as c language expression such as the following:

```
DEFINE C A + B
DEFINE D A + 10
DEFINE E 10 + 10
```

- INCLUDE

INCLUDE is used to include another linker script in the generated one. Its argument file_name must be double-quoted as follows:

```
INCLUDE "another.ld"
```

- ENTRY

ENTRY is used to specify the entry point where the first instruction executes in a program. It can be declared anywhere (i.e., not necessarily in the first line) in a SaG-formatted script. If an entry point is not defined in header, the following default entry setting will be used:

```
ENTRY (_start)
```

- EXTERN

EXTERN is used to force symbols to be entered in the output file as undefined. It has the same effect as the command-line linker option -u. The following gives an example of its usage.

```
EXTERN (sym1 sym2 sym3)
```

- #include

In header of the SaG-formatted script, you can also use #include to import SaG DEFINEs from external files, which may only contain comments in addition to the DEFINEs. When declared, the filenames must be enclosed with a pair of double quotes, as shown below.

```
#include “define.sag”
```

Note that nested include files are not supported in the imported files at the moment.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Load region

Next, define a `load_region_description` as a load region name, optionally followed by attributes or size specifiers, and one or more execution region descriptions:

```
load_region_description ::=
load_region_name (address|("+"offset)) [load_attr][max_size]
"{"
   exe_region_description+
"}"
```

Execution region

An `exe_region_description`, in turn, is defined as an execution region name, a base address specification, optionally followed by attributes or size specifiers, and one or more input section descriptions:

```
exe_region_description ::=
exe_region_name (address| ("+" offset)) [exe_attr][max_size]
"{"
   (input_section_description)+
"}"
```

Input section

Define an `input_section_description` as a source module selector pattern optionally followed by input attributes, an address variable, a load address variable, a stack, a gp register, or a VAR variable.

```
input_section_description ::=
(module_select_pattern exclude_description [input_attr] "("
input_section_selector ( "," input_section_selector )* ")"
  | ADDR [NEXT] variable
  | LOADADDR [NEXT] variable
  | STACK "=" num
  | GP "=" num
  | VAR variable "=" expression
  | variable "=" ALIGN "(" num ")"
  )
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Assert information

Define an assert_information to perform complex size checks. The assert_information consists of the uppercased SCATTERASSERT as the keyword and an assert_description which allows the following uppercase keywords for different address patterns:

- LOADBASE: the base address of the load region
- LOADLENGTH: the size of the load region
- LOADLIMIT: the end address of the load region
- IMAGEBASE: the base address of the execution region
- IMAGELENGTH: the size of the execution region
- IMAGELIMIT: the end address of the execution region

In an assert_description, you can use "+" to accumulate region sizes and compare the result with an integer number. The following is an example.

```
ROM 0x10000 {
  RAM1 0x10000 { *(+RO) }
  RAM2 0x20000{ *(+RW,+ZI) }
}
SCATTERASSERT (IMAGELENGTH (RAM1) + IMAGELENGTH (RAM2) < 0x10000)
SCATTERASSERT (LOADLENGTH (RAM1) + LOADLENGTH (RAM2) < 0x10000)
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

13.1.2.2 Load region description

Syntax

```
load_region_description ::=
load_region_name (address|(”+”offset)) [load_attr][max_size]
“{“
   (exe_region_description | exe_overlay_region_description)+
 “}”
```

where

load_region_name	consists of letters, underscore and numbers. Note that the first character must not be a number.
address	Its value can be a decimal/hexadecimal number or a macro defined by the DEFINE keyword.
offset	can be a decimal or hexadecimal number. If it is used in the first load region, then +offset means that the base address begins offset bytes after zero. Otherwise, it means offset bytes beyond the end of the preceding load region.
load_attr	is defined as “ALIGN alignment” where ■ ALIGN is a keyword and must be upper-cased ■ alignment can be a two-to-the-power decimal or hexadecimal number.
max_size	specifies the maximum size of the load region. Its value can be a decimal/hexadecimal number or a macro defined by the DEFINE keyword. If the target object size is bigger than the value, it will report errors during linking.
exe_region_description	See Section 13.1.2.3.
exe_overlay_region_description	See Section 13.1.2.5.

Example

```
LOAD_ROM_1 0x0000 ALIGN 0x4 0x10000 ; the LOAD_ROM_1 will be aligned
                                    ; to 4-byte aligned address and the max size is 64k
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

13.1.2.3 Execution region description

Syntax

```
exe_region_description ::=
exe_region_name (address| ("+" offset)) [exe_attr][max_size]
"{"
   (input_section_description)+
"}"
```

where

exe_region_name	consists of letters, underscore and numbers. Note that the first character must not be a number.
address	Its value can be a decimal or hexadecimal number or a macro defined by the DEFINE keyword.
offset	can be a decimal or hexadecimal number. If it is used in the first execution region in the load region, then +offset means that the base address begins offset bytes after the base of the containing load region. Otherwise, it means offset bytes beyond the end of the preceding execution region.
exe_attr	is defined as "ALIGN alignment" where ■ ALIGN is a keyword and must be upper-cased. ■ alignment can be a two-to-the-power decimal or hexadecimal number.
max_size	specifies the maximum size of the load region. Its value can be a decimal/hexadecimal number or a macro defined by the DEFINE keyword. If the target object size is bigger than the value, it will report errors during linking.
input_section_description	See Section 13.1.2.4.

Example

```
EXEC_ROM_1 0x0000 ALIGN 0x4 0x8000 ; the EXEC_ROM_1 will be aligned
                                   ; to 4-byte aligned address and the max size is 32k
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

13.1.2.4 Input section description

Syntax

```
input_section_description ::=
(module_select_pattern exclude_description [input_attr] "("
 input_section_selector ( "," input_section_selector )* ")"
 | ADDR [NEXT] variable
 | LOADADDR [NEXT] variable
 | STACK "=" num
 | GP "=" num
 | VAR variable "=" expression
 | variable "=" ALIGN "("num")"
 | PROVIDE "(" variable "=" expression ")"
)
```

where

`module_select_pattern`	is defined as "`(filename)+`" where ■ `filename` can be any object file name or path of the object file. The wildcard character * matches zero or more characters while ? matches any single character.
`exclude_description`	is defined as `EXCLUDE_FILE "(" (filename )+ ")"` where ■ `EXCLUDE_FILE` is a keyword and must be upper-cased For example, `* EXCLUDE_FILE(hello.o) (+RO, +RW, +ZI)` is to put all objects except for `hello.o` into this region.
`input_attr`	is defined as one of the following: ■ `KEEP` is a keyword and must be upper-cased. It marks the sections that should not be eliminated when link-time garbage collection is in use. ■ `SORT` is a keyword and must be upper-cased. It sorts the module file by name. ■ `SORT_INPUT_SECTION` is a keyword and must be upper-cased. It sorts the input section by name.

`input_section_selector` is defined as

```
("+" input_section_attr
  [NOLOAD][LMA_FORCE_ALIGN]
| input_section_pattern
  [NOLOAD][input_section_setting]
  [input_section_lma_setting]
| group_input_section_pattern )
```

Where:

- `input_section_attr` is an attribute selector matched against the input section attributes. Recognized selectors include -
  - `RO`: Select both read-only code and read-only data.
  - `RW`: Select both read-write code and read-write data.
  - `ZI`: Select zero initialized data.
  - `RO-CODE`: Select read-only code.
  - `RO-DATA`: Select read-only data.
  - `RW-CODE`: Select read-write code.
  - `RW-DATA`: Select read-write data.
  - `ISR`: Select interrupt service routine.
  - `EXECIT`: Select the look-up table `.exec.itable` from the read-only code as an independent section.
  - `JVT`: Select `.riscv.jvt`, the section that stores table jump target addresses, from the read-only code as an independent section.

NOTE

1. The suggested sequence of predefined sections is RO, RW, and then ZI. In Andes library, `CRT0` sets memories after the symbol `_edata` to `0` by default. As the symbol is at the end of RW section, sections after RW should be all zero-initialized. If you want to change the sequence of RW and RO sections, be sure to rewrite `CRT0` for

allowing sections that are not zero-initialized after RW.

2. ZI section must be arranged after other sections since the LMA and the VMA of ZI-related sections (.bss, .sbss_f, .sbss_b, .sbss_h, .sbss_w and .sbss_d) are the same. If ZI section precedes others, the LMA of the succeeding section will fail to find ZI section's VMA to follow. The following shows a misplacement of ZI section:

```
LOAD 0x1000 {
      ROM 0x1000 { *(+RO) }
      RAM 0x2000 { *(+ZI,+RW) }
}
```

In the example, the LMA of RW section is supposed to follow the LMA of ZI section. However, since the LMA and the VMA of ZI section are the same, the LMA of RW section will be excluded.

- `NOLOAD` marks a section not to be loaded at runtime, used as the NOLOAD directive in the GNU linker script.

- `LMA_FORCE_ALIGN` forces the LMA alignment of sections to be same as the VMA alignment.

- `input_section_pattern ::=(.text | .data|…)` where
  - `.text` refers to the following set - `(.text .stub .text.* .gnu.linkonce.t.*)` `(*(.text.*personality*))` `(.gnu.warning)`
  - … refers to any section name (including user-defined name) that is matched against the input

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

section name. It allows wildcard character *, which matches zero or more characters.

- `input_section_setting ::= "("num")"`

  This setting fills `input_section_pattern` to align the number that `num` denotes. `num` can be a decimal or hexadecimal number.

- `input_section_lma_setting ::= LMALIGN "("num")" | LMA_FORCE_ALIGN`
  - `LMALIGN` aligns this section to the number that `num` denotes.
  - `LMA_FORCE_ALIGN` forces the LMA alignment of this section to be the same as the VMA alignment.

- `group_input_section_pattern ::= "[" input_section_pattern ("," input_section_pattern)* "]"`

  Compared with `input_section_pattern` which generates respective sections, `group_input_section_pattern` generates only one output section named as the first `input_section_pattern` for the latter `input_section_patterns` to join, avoiding the gap of each section. For example,

  - Example 1 (`input_section_pattern`) :

    `*(.text, .text1)`

    → Output: `.text { *(.text) }`
    `.text1 { *(.text1) }`

  - Example 2 (`group_input_section_pattern`):

    `*([.text, .text1])`

    → Output: `.text { *(.text, .text1) }`

	In Example 2, `*([.text, .text1])` as `group_input_section_pattern` generates only one section while `*(.text, .text1)` as an `input_section_pattern` in Example 1 generates two sections.
`ADDR [NEXT] variable`	assigns the VMA to a variable. The variable consists of letters, underscore and numbers. Note that its first character must not be a number. `NEXT` is a keyword and must be upper-cased. If it is set, the variable will be the VMA for the start of the next section rather than that for the end of the previous section.
`LOADADDR [NEXT] variable`	assigns the LMA to a variable. The variable consists of letters, underscore and numbers. Note that its first character must not be a number. ■ `NEXT` is a keyword and must be upper-cased. If it is set, the variable will be the LMA for the start of the next section rather than that for the end of the previous section.
`STACK "=" num`	assigns the stack address. `STACK` will generate "`PROVIDE (_stack = num);`" into the output script; `num` can be a decimal or hexadecimal number. **NOTE** According to the standard RISC-V ABIs, the stack alignment must be 16 bytes for RV32I or RV64I, and 4 bytes for RV32E.
`GP "=" num`	assigns the address to which the gp register would point. `GP` will generate "`PROVIDE (__global_pointer$ = num);`" into the output script; `num` can be a decimal or hexadecimal number.
`VAR variable "=" expression`	defines a variable and its value. The variable consists of

letters, underscore and numbers. Note that its first character must not be a number.

- The `expression` here is identical to C expressions, but it only allows "+, -, *, /".

`variable "=" ALIGN "("num")"`

`ALIGN` sets a variable to the location counter aligned to the next alignment boundary. If the variable name is ".", it adjusts the location counter to the next alignment boundary.

`PROVIDE "(" variable "=" expression ")"`

is similar to `VAR`, but with a lower priority. If the variable has been defined by `VAR`, or if it is a global variable defined in a C program, this syntax will be ignored by the linker.

Example

```
● program1.o KEEP (.text, +RO)     ; the output section will include
    ; the program1.o's .text and read-only sections as its input
    ; section and it will not be eliminated by gc-section
● ADDR _data_start             ; assigns the VMA to _data_start
● LOADADDR _data_start         ; assigns the LMA to _data_start
● STACK = 0x200000             ; assigns the stack address to 0x200000
● VAR my_var = 0x1000          ; defines a custom variable my_var and sets
                               ; its value as 0x1000
```

Notes

- To avoid ambiguity errors, take note not to import `input_section_descriptions` using the same `module_select_patterns` along with duplicate `input_section_selectors` in a description file. The following examples present illegal usages from Example 1 to 3 and legal usages from Example 4 to 6.
  - Example 1 (illegal):
    ```
    *(.text)
    *(.data, .text)
    ```
  - Example 2 (illegal):
    ```
    *(+RO)
    *(+RO-CODE)
    ```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

- Example 3 (illegal):

```
hello.o (+RW-DATA)
hello.o (+RW)
```

- Example 4 (legal):

```
hello.o (.text)
*(.data, .text)
```

- Example 5 (legal):

```
hello.o (+RO)
*(+RO)
```

- Example 6 (legal):

```
*(.text)
*(+RO)
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

13.1.2.5 Execution overlay region description

Syntax

```
exe_overlay_region_description ::=
exe_region_name (address| ( "+"  offset)) [exe_attr]  "OVERLAY"
pagesize
"{"
    (overlay_input_section_description)+
"}"
```

where

exe_region_name	consists of letters, underscore and numbers. Note that the first character must not be a number.
address	Its value can be a decimal/hexadecimal number or a macro defined by the DEFINE keyword.
offset	can be a decimal or hexadecimal number. If it is used in the first execution region in the load region, then +offset means that the base address begins offset bytes after the base of the containing load region. Otherwise, it means offset bytes beyond the end of the preceding execution region.
exe_attr	is defined as "ALIGN alignment" where ■ ALIGN is a keyword and must be upper-cased. ■ alignment can be a two-to-the-power decimal or hexadecimal number.
OVERLAY	is the keyword and it must be the upper case.
pagesize	is the size of each overlay page. When it is set to 0, software overlay is used.
overlay_input_section_description	See Section 13.1.2.6.

13.1.2.6 Overlay input section description

Syntax

```
overlay_input_section_description ::=
(output_section_name "{"(module_select_pattern [input_attr]+
"("input_section_selector ( "," input_section_selector )* ")" "}") +
```

where

`output_section_name`	consists of letters, underscore and numbers. Note that the first character must not be a number.
`module_select_pattern`	is the same as `module_select_pattern` in Section 13.1.2.4.
`input_attr`	is the same as `input_attr` in Section 13.1.2.4.
`input_section_selector`	is the same as `input_section_selector` in Section 13.1.2.4.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

13.1.2.7 Examples

- Example 1:

```
LOAD_ROM 0x10000    ; ROM starts from 0x10000
{
    EXEC_RAM 0x10000 ; RAM starts form 0x10000
    {
        *(+RO)       ; read-only section's VMA = LMA
    }
    EXEC_ROM 0x20000
    {
        *(+RW,+ZI)   ; read-write and zero-init's VMA starts from
0x20000
                     ; LMA follows RO section
    }
}
```

- Example 2 (overlay):

```
USER_SECTIONS .overlay0, .overlay1, .overlay2
ROM 0x0 ;LMA start address 0x0
{
    RAM 0x0 ;VMA start address 0x0
    {
      *(+RO, +RW, +ZI)      ;put all generic section here
      STACK = 0xA00000      ;assign stack address
    }
}
ROM_OVLY 0x14000 ;LMA start address 80K
{
    RAM2 0x4000 OVERLAY 0x2000    ;VMA start address 0x4000. using
overlay, each overlay pagesize is 0x2000
    {
      .overlay0 {* (.overlay0)};LMA 0x14000, VMA 0x4000
      .overlay1 {* (.overlay1)};LMA 0x16000, VMA 0x6000
      .overlay2 {* (.overlay2)};LMA 0x18000, VMA 0x8000
    }
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

13.2. Linker script generator (LdSaG)

With a SaG-formatted script file in hand, you can use the command option `nds_ldsag` to generate a corresponding linker script. Its usage is as follows:

```
$ ./nds_ldsag
./nds_ldsag: [option] file

Options:
    -h                   //Output the details of all options.
    -v                   //Output the version of LdSaG.
    -t FILE_NAME         //Read the template file, for advanced users only.
                         //The default template file is nds32_template_v5.txt
    -o FILE_NAME         //Output a file with the specified file-name.
    --load-zi            //ZI region are set to NOLOAD by default.
                         //Use this option to set ZI region as loadable.
    --keep-debug-macro   //Generate .debug_macro sections, which are not
                         //generated by default, for debugging into the macros.
    --no-sort-loads      //Disable sorting load regions. (LLD requires load
                         //regions sorted.)
    --executable-start   //Specify the starting execution region for the
                         //executable.
```

If the output filename is not specified, Andes linker will generate a linker script using the default output name `nds32.ld`.

The following example demonstrates how to use `nds_ldsag` to generate a linker script with a .sag file:

Step 1 Write a SaG-formatted description file like `test.sag` below.

```
LOAD_ROM 0x10000 ; ROM starts from 0x10000
{
    EXEC_RAM 0x10000 ; RAM starts form 0x10000
    {
        *(+RO,+RW,+ZI) ; put read-only, read-write, zero-
                        init
                         ; into ROM and RAM
    }
}
```

Step 2 Use nds_ldsag to read the description file and output a linker script in the given filename.

```
./nds_ldsag test.sag -o myldscript
```

A linker script is generated; in this case, it's myldscript.

Step 3 Use the newly-generated linker script to compile an object.

```
riscv[32|64]-elf-[gcc|clang] -Wl,-T,myldscript hello.c -o a.out
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

14. Object files

14.1. ELF file

ELF stands for Executable and Linking Format. Currently, this is the only format supported by Andes toolchains.

There are three types of ELF object files:

1. Relocatable file is for linking with other object files to create an executable or a shared object file.
2. Executable file is a program suitable for execution.
3. Shared object file is either for link editor to link with other relocatable and shared object files to create another object file or for dynamic linker to link with an executable and other shared objects to create a process image.

See *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification* for more details.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

14.2. Examining ELF file

The following tools can be used to examine ELF files:

1. `riscv[32|64]-elf-readelf` displays all kind of information in an ELF file.
2. `riscv[32|64]-elf-objdump` disassembles instructions or dumps section data.

See the GNU Binutils document for more details.

Here is a partial listing generated by the command line "`riscv32-elf-readelf -a libc.a`"

```
File: riscv32-elf/lib/libc.a(lib_a-_Exit.o)
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          448 (bytes into file)
  Flags:                             0x1, RVC, soft-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         10
  Section header string table index: 9

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000000 00  AX  0   0  2
  [ 2] .data             PROGBITS        00000000 000034 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          00000000 000034 000000 00  WA  0   0  1
  [ 4] .text._Exit       PROGBITS        00000000 000034 000010 00  AX  0   0  2
  [ 5] .rela.text._Exit  RELA            00000000 000148 000030 0c   I  7   4  4
  [ 6] .comment          PROGBITS        00000000 000044 00003d 01  MS  0   0  1
  [ 7] .symtab           SYMTAB          00000000 000084 0000a0 10      8   7  4
  [ 8] .strtab           STRTAB          00000000 000124 000024 00      0   0  1
  [ 9] .shstrtab         STRTAB          00000000 000178 000046 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
There are no section groups in this file.

There are no program headers in this file.

Relocation section '.rela.text._Exit' at offset 0x148 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name + Addend
00000000  00000812 R_RISCV_CALL      00000000   __riscv_save_0 + 0
00000000  00000033 R_RISCV_RELAX                0
00000008  00000912 R_RISCV_CALL      00000000   _exit + 0
00000008  00000033 R_RISCV_RELAX                0

The decoding of unwind sections for machine type RISC-V is not currently supported.

Symbol table '.symtab' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS _Exit.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    2
     4: 00000000     0 SECTION LOCAL  DEFAULT    3
     5: 00000000     0 SECTION LOCAL  DEFAULT    4
     6: 00000000     0 SECTION LOCAL  DEFAULT    6
     7: 00000000    16 FUNC    GLOBAL DEFAULT    4 _Exit
     8: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND __riscv_save_0
     9: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _exit

No version information found in this file.
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

15. Andes MCUlib

15.1. Features of MCUlib

MCUlib toolchains are used to build for smaller code size. Unlike Newlib, it doesn't support reentrancy and has its own printf implementation. It is not thread-safe either, which is in common with Newlib.

15.2. MCUlib printf implementation

This section introduces the MCUlib-specific printf implementation.

Name

`printf`

Syntax

`int printf (const char *format, ……)`

Where the format has the following form:

`%[flag][field width][.precision][modifier][conversion]`

And, the following are the characters supported in MCUlib printf's format specification fields:

Field	Supportive Character	Description
Flag	-	left justify, pad right with blanks
	0	pad left with 0 for numerics
	+	always print sign, + or -
	#	alternate form
	' '	(blank)
field width	(field width)	
precision	(.precision)	
modifier	ll	long (64-bit) int
	h	short (16-bit) int
	l	long (32-bit on RV32, 64-bit on RV64) int
conversion	d,i	decimal int
	u	decimal unsigned

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

`o`	octal
`x,X`	hexadecimal
`f,e,g,F,E,G`	float
`c`	character
`s`	string
`p`	pointer

Return Value

total number of characters printed

Note

1. If the format string doesn’t contain any conversions, the compiler may optimize the call to `puts()`.
2. On RV32, the `ll` modifier for specifying long (64-bit) data types is not supported.
3. Some low-level functions must be implemented for `printf` to actually output the string. On Andes evaluation boards, they are implemented in libgloss via syscall mechanism. For other target boards, you must override the prototype of functions “`_write()`” and “`_fstat()`” in libgloss:

```
ssize_t _write(int __fd, const void *__buf, size_t __nbytes) {
  // Write out characters from “__buf” to “__buf + __nbytes - 1”
  // to a device
}

int _fstat (int file, struct stat * st) {
        return 0;
}
```

4. Starting from AndeSight v5.0.0 official release, `printf()` may be replaced with `iprintf()` for the reduction of stack size if there is no floating-point parameter. This, however, may lead to the replacement of `printf()` provided by users with the builtin version. To prevent such replacement, you can specify `-fno-builtin` to avoid using all builtin functions or just disable the builtin `printf()` family by specifying `-fno-builtin-printf`, `-fno-builtin-sprintf` and `-fno-builtin-snprintf` separately.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

15.3. Precision limitation of floating point conversion functions

In MCUlib, functions that covert decimal strings to binary floating-point, such as `atof()`, `strtod()`, and `scanf()`, have a precision limitation of having the normalized result as $A*10^{\pm B}$ where A is an integer of 15 digits at maximum and B is a power of ten exponent that has a value no more than 22. For example, 9.87654321098765e43 is expressed as 987654321098765*10000000000000000000000 and 1.23456 as 123456/100000. If the normalization of these functions goes beyond such limitation, the conversion fails and produces a result of 0.0.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

15.4. MCUlib's RISC-V Vector Extension (RVV) Functions

MCUlib offers an alternative implementation of string and memory-related functions using RVV to enhance their performance. These functions are denoted by the suffix "`_rvv`." For instance, `strcpy_rvv()` serves as the RVV version of `strcpy()`. Please note that (1) the RVV versions of functions may not yield performance improvements for data of small size, and (2) they can only be utilized on Andes target processors supporting the RVV extension. The following list all the RVV functions in MCUlib.

- memccpy_rvv
- memchr_rvv
- memcmp_rvv
- memcpy_rvv
- memmove_rvv
- mempcpy_rvv
- memrchr_rvv
- memset_rvv
- rawmemchr_rvv
- stpcpy_rvv
- stpncpy_rvv
- strcat_rvv
- strchr_rvv
- strcmp_rvv
- strcpy_rvv
- strcspn_rvv
- strlcat_rvv
- strlcpy_rvv
- strlen_rvv
- strncat_rvv
- strncmp_rvv
- strncpy_rvv
- strnlen_rvv
- strpbrk_rvv
- strspn_rvv
- strtok_r_rvv

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

16. Virtual Hosting

Via Virtual Hosting, I/O requests of target boards without I/O devices can be directed to GDB on the host side, thereby accelerating development processes and shortening development cycles. For example, code coverage (gcov) requires writing code coverage data to files. By Virtual Hosting, the feature still can be supported on target boards that don't have I/O devices.

Used with MCUlib or newlib toolchains, Virtual Hosting is supported on both real boards (through ICEman) and the simulator.

To enable Virtual Hosting, add the `-mvh` option when invoking the compiler to compile and link programs. This option will link the programs with a Virtual Hosting library where functions redirect I/O requests to ICEman or the simulator. These requests will then be passed to GDB, invoking I/O services on the host side and sending results back to ICEman or the simulator.

The low-level I/O functions supported by the up-to-date Virtual Hosting include `open`, `close`, `read`, `write`, `seek`, `flen`, `remove`, `rename`, `istty`, `clock`, `time`, `get_cmdline` and `exit`. These functions may be interrupted by Ctrl+C, leading Virtual Hosting to fail in the middle of program execution. Thus, you should have your programs check the return code to see if Virtual Hosting has been done successfully. You may retry the operation if necessary.

NOTE

1. When Virtual Hosting is enabled, avoid redirecting the output with `_write()`.
2. For `open()` flags supported by the current Virtual Hosting and their corresponding `fopen()` modes, see the table below.

`open()` **flags**	`fopen()` **modes**
O_RDONLY	"r"
O_RDWR	"r+"
O_WRONLY | O_CREAT | O_TRUNC	"w"
O_RDWR | O_CREAT | O_TRUNC	"w+"
O_WRONLY | O_CREAT | O_APPEND	"a"
O_RDWR | O_CREAT | O_APPEND	"a+"

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

17. Advanced optimization

This chapter introduces diverse optimization techniques to improve performance, reduce code side, and enhance execution efficiency for programming with Andes architecture.

17.1. Optimization options

The following list common options and Andes GCC and LLVM compiler and linker options that deal with optimizations.

17.1.1. Options for code size optimization

- Compiler Options
  `-Os`

  Sometimes the code size optimizations may degrade the performance. Therefore, for V5 family toolchains, several levels of code size reduction are supported: `-Os1`, `-Os2`, `-Os3` and `-Oz`. Table 23 below provides detailed descriptions for these levels.

Table 23. Code size optimization levels

Option	Code Size Optimization Level
`-Os1`	This option enables minimum code size optimizations. Performance is still concerned.
`-Os2`	This option enables partial code size optimizations with little performance concern.
`-Os3 (-Os)`	This option works the same as the `-Os` option to enable all code size optimizations. Performance may seriously drop with this option.
`-Oz`	Supported by LLVM only, this option works the same as `-Os` option to enable all code size optimizations. Performance may seriously drop with this option.

- Linker options
  `--mrelax-cross-section-call`
  `--mexecit-jal-over-2mib`

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

The option `--mrelax-cross-section-call` enables relaxation of `call` and `tail` pseudo instructions into a `jal` instruction to which the caller and callee are in different output sections. This option is by default disabled. To use the option, you must ensure text sections in the linker script are arranged in a way that only the range of the jump will be reduced by the optimization. For example, text sections must not be specified at fixed locations for the use of this option, or it may result in out-of-range errors.

The option `--mexecit-jal-over-2mib` enables `exec.it` optimization of `jal` instructions when the caller and callee are outside the first 2MiB page of the program text section. If a program has multiple disjointed text sections (e.g., one starting at 0x400000 and another at 0x10000000), by default only `jal` in the first 2MiB page (0x400000-0x5ffffe) can be converted into an `exec.it` instruction. By enabling `--mexecit-jal-over-2mib`, the `jal` instructions in subsequent 2MiB pages can also be converted. This option is by default disabled. To use this option, you must make sure that no text sections can cross 2MiB boundary. Otherwise, linking errors may occur.

17.1.2. Options for code speed optimization

- Compiler options
  `-O3`
  `-funroll-loops`
  `-funroll-all-loops`
  `-malways-align`
  `-mcpu=PROCESSOR_NAME` or `-mtune=PROCESSOR_STRING`

The followings are some notes you should pay attention when using these options:

1. For `-O3`, sometimes the code size may increase dramatically after this option is applied. This is because `-O3` also implies `-finline-functions` that can expand the content of the callee within the caller (See Table 25 for enabled options at `-O3`). To avoid such function inlining optimization, just use the option `-fno-inline-functions`.

2. For `-funroll-loops` and `-funroll-all-loops`, take note that unrolling loop is not always good for performance on platforms with cache enabled. Please see Table 24 and use these options wisely to meet your requirement.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Table 24. Two loop unrolling optimization

Option	Use with GCC	Use with LLVM
-funroll-loops	Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. The compiler has a set of heuristics to estimate whether to unroll loop or not.	The compiler has a set of heuristics to estimate whether to unroll loop or not, even if the number of iterations is uncertain when the loop is entered. This option may make programs run more slowly if it loses locality after unrolling.
-funroll-all-loops	Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This option may make programs run more slowly if it loses locality after unrolling.	This option is accepted but ignored by clang.

3. -malways-align enforces 4-byte alignment on jump targets, return addresses and function entries to prevent extra performance penalty due to misalignment. The option is not default applied at -Os (including -Os1, -Os2, -Os3, and -Oz) since it may slightly increase code size. However, it is enabled by default at most other optimization levels (see Table 25).

4. -mcpu=PROCESSOR_NAME or -mtune=PROCESSOR_STRING optimizes the program for the given processor by specification of the processor name or series. Both options will optimize the program according to processor features such as pipeline latency.

   For -mcpu, the permissible values are names of individual AndesCore processors such as n25, n27, n45 or ax65. As to -mtune, it accepts all valid values for -mcpu and processor alias such as andes-25-series, andes-27-series, andes-45-series, and andes-60-series. Processor alias for -mtune will be resolved according to the architecture (i.e., rv32 or rv64) of the target processor in use. For example, the alias andes-45-series will be resolved to n45 or nx45. If neither -mtune nor -mcpu is specified, the default value set for the optimization is n25.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Moreover, the compiler supports multilib for `andes-25-series`, `andes-45-series` and `andes-60-series`. That is, using the option `-mtune=andes-[25|45|60]-series` also enables the compiler to change to the libraries built with 25-, 45- or 60- series pipeline models.

17.1.3. Options to remove unused sections

To remove unused sections, the following compiler and linker options have to be enabled at the same time:

- Compiler options
  `-ffunction-sections`
  `-fdata-sections`

- Linker options
  (gcc as linker) `-Wl,--gc-sections`
  (ld as linker) `--gc-sections`

These options are suggested to be used along with the option `-Wl,--print-gc-sections` (gcc as linker) or `--print-gc-sections` (ld as linker). By doing so, you can easily see what sections are discarded by linker.

17.1.4. Options to use EXEC.IT optimization

The `exec.it` instruction can be used at link time optimization for ELF toolchains. To apply the EXEC.IT optimization, the following compiler and linker options have to be enabled at the same time:

- Compiler option
  `-mexecit`

- Linker options
  (gcc as linker) `-Wl,--mexecit`
  (ld as linker) `--mexecit`

Notice that `-Os` enables these options by default. If you do not want to apply the EXEC.IT optimization at link time, use "`-Wl,--mno-execit`" (gcc as linker) or "`--mno-execit`" (ld as linker).

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

17.1.5. Notice on some optimization options

The compiler assumes that a valid program must be well-defined by the C language standard. If there is any undefined behavior in your program, the result is unpredictable and unexpected consequence could occur anytime. This section describes some optimization options that may help you to detect undefined behavior of your programs in the early stages. These options may also be workarounds if you have no choice but to write invalid programs for some reason. Please be aware of each option's behavior and effects before leveraging them in various cases.

- `-fno-delete-null-pointer-checks`
  In the C language standard, programs cannot safely dereference null pointers, and no code or data element resides there. However, this assumption is not true in some cases, especially for embedded platform. Thus, if you have to dereference the memory address 0x00000000, please use `-fno-delete-null-pointer-checks` to tell the compiler not to optimize out null pointer checking.

- `-fno-strict-aliasing`
  In the compiler framework, it may enable strict aliasing optimization on the assumption that the strictest aliasing rules applicable to the language being compiled. If a program contains pointer casting, it may break the strict aliasing rule. Therefore, it would be better not to use pointer casting in your programs. If you must use it, having the option `-fno-strict-aliasing` is recommended. Otherwise, the execution result may be unexpected.

- `-fwrapv`
  The C language standard considers the overflow of a signed value is undefined behavior. That means a valid program must never generate signed overflow when computing an expression and the compiler is able to perform some optimization under such condition. If you must have invalid code containing signed overflow, please compile it with `-fwrapv`, which tells the compiler to treat signed overflow as wrapping.

- `-mfma`
  The option `-mfma` is used to generate floating-point fused multiply-add (FMA) instructions. It is enabled by default in Andes Compiler to increase the performance, albeit an unsafe floating-point math optimization which may decrease precision. If you need a program result compatible with IEEE 754, use `-mno-fma` to disable this optimization.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

17.1.6. Optimization levels and default applied options

The following summarizes the optimization levels that Andes compiler supports:

`-O0`	Do not optimize.
`-Og`	Optimize for speed with better debuggability than `-O1`
`-O1`	Optimize for speed
`-O2`	Optimize more for speed
`-O3`	Optimize most for speed
`-Os1`	Optimize for size
`-Os2`	Optimize more for size
`-Os3`	Optimize most for size

You can also use Andes target specific options (see Section 2.1) to tune performance and code size. Some target options have been enabled at certain optimization levels by default. Please see Table 25 below for their default applied scenarios:

Table 25. Default applied compiler options at each optimization level

Mnemonic	-O0	-Og	-O1	-O2	-O3	-Os1	-Os2	-Os/-Os3
-fomit-frame-pointer	√	√	√	√	√	√	√	√
-fno-delete-null-pointer-checks	√	√	√	√	√	√	√	√
-finline-functions				√	√			
-malways-align		√	√	√	√			
-msave-restore							√	√
-minnermost-loop							√	
-mexecit							√	√
-mgp-insn-relax	√	√	√	√	√	√	√	√

Note that options that are not default applied at some optimization level can still be turned on when you issue them. Similarly, using `-fno-omit-frame-pointer`, `-fdelete-null-pointer-checks`, `-fno-inline-functions`, `-mno-always-align`, `-mno-save-restore`, `-mno-innermost-loop`, `-mno-execit` and `-mno-gp-insn-relax` can avoid the options in Table 25 from being enabled at their respective "default applied optimization levels."

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

17.2. Saving code size for function prologue and epilogue

Before using a callee-saved register, the compiler must save the content of the register, adjust the stack pointer at the start of the function and reverse before the function returns to the caller. This is called function prologue and epilogue.

Since AndeStar V5 doesn't provide instructions to load/store multiple registers, performing the function prologue and epilogue may result in enormous code size when a program contains lots of functions. In this instance, you can use the option `-msave-restore` to reduce the code size, but mind that the additional function call it introduces may cause a little performance degradation.

Also, be aware that the option `-msave-restore` has some limitations. It does not function for the following programs or functions even though it is applied:

- Programs that are also compiled with the option `-fno-omit-frame-pointer`
- Functions that call the `alloca` function
- Functions that take variable number of arguments

Here is an example to show the functionality of the option `-msave-restore`. Given a program like below,

```
/* Example-save-restore  */
#include <stdio.h>
int bar(int );
int foo(int a){
  asm volatile ("# Force compiler using callee-saved register"
                 :::"a0", "s0", "s1", "s2", "s3");
  return bar(a);
}
```

The following table lists the difference of code generation when the program is compiled without and with `-msave-restore`. The compilation results show that `-msave-restore` enables significant code reduction by saving 12 instructions on function prologue and epilogue.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Compiled with	-O1	-O1 -msave-restore

Compilation result	foo:
	add sp,sp,-32
	sw ra,28(sp)
	sw s0,24(sp)
	sw s1,20(sp)
	sw s2,16(sp)
	sw s3,12(sp)
	mv a5,a0
	#APP
	# 4 "example-save-restore.c" 1
	# Force compiler using callee-saved register
	# 0 "" 2
	#NO_APP
	mv a0,a5
	call bar
	lw ra,28(sp)
	lw s0,24(sp)
	lw s1,20(sp)
	lw s2,16(sp)
	lw s3,12(sp)
	add sp,sp,32
	jr ra
		foo:
		call t0,__riscv_save_4
		mv a5,a0
		#APP
		# 4 "example-save-restore.c" 1
		# Force compiler using callee-saved register
		# 0 "" 2
		#NO_APP
		mv a0,a5
		call bar
		tail __riscv_restore_4

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

17.3. Addressing space for programs

It is easy to locate local variables because they are only accessed via frame pointer or stack pointer within a stack frame and will be destroyed at the end of the function. However, it is not the case for global variables, which are used to store information shared among functions and tasks. In AndesCore processors, accessing a global variable requires several instructions to construct a full addressing space (32-bit or 64-bit). Similar issues also appear on function calls. To call a function all over the addressing space, many instructions are also needed to calculate possible addresses within the entire addressing space and then jump to the function via a register.

Instructions that always construct a full addressing space can seriously affect performance and code size. Fortunately, most programs do not require complete addressing space because of limited resources (e.g. ROM size) in practice. You may improve the overall performance and code size simply using compiler options of different code models.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

17.3.1. Code models

With Andes toolchains, you can use the option -mcmodel=[small|medium|large|medlow|medany] to tell the compiler which scale your programs and data are. Specifying precise code models with this option is helpful for code generation as the information facilitates the compiler to generate smaller and faster programs directly. The following list supported code models for this option.

- -mcmodel=small (alias to -mcmodel=medlow for V3 backward compatibility)
- -mcmodel=medium

  For RV32, this option is alias to -mcmodel=medlow; for RV64, this is alias to -mcmodel=medany.

- -mcmodel=medlow

  This is to generate code for the medium-low (medlow) code model, which allows the code to address the whole RV32 address space or the lowest 2 GiB and highest 2GiB of the RV64 address space (i.e., 0x0 ~ 0x000000007FFFF7FF and 0xFFFFFFFF7FFFF800 ~ 0xFFFFFFFFFFFFFFFF). Programs can be statically or dynamically linked. Medlow is the default code model for RV32 because it usually gets better code size and performance after linker relaxation.

- -mcmodel=medany

  This is to generate code for the medium-any (medany) code model, which allows the code to address the range between -2 GiB and +2 GiB from its position. Programs can be statically or dynamically linked. Medany is the default code model for RV64.

- -mcmodel=large

  For RV32, this is alias to -mcmodel=medlow for V3 backward compatibility.
  For RV64, this is to generate code for the large code model, which allows the code to address the whole RV64 address space. Programs can be statically or dynamically linked. Note that unless the access distance to a symbol exceeds -2 GiB or +2 GiB from the execution code, it is suggested to use the medany code model, rather than large, for RV64.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

17.4. Link time optimization

Link Time Optimization (LTO) is a very aggressive optimization implemented by GCC and LLVM. It gives a compiler the capability of emitting its internal representation into object files, so that all the different compilation units that make up a single executable can be optimized as a single module.

17.4.1. Using LTO

If you would like to apply LTO on your program, make sure you use GCC to complete all the works of building a program, including compilation and linking. Then, the compiler is able to interact with linker plugin to perform optimization.

The option `-flto` triggers the main LTO features. Given several source files like below, you can create an executable with this option:

```
$ riscv32-elf-[gcc|clang] -O2 -flto -c f1.c
$ riscv32-elf-[gcc|clang] -O2 -flto -c f2.c
$ riscv32-elf-[gcc|clang] -O2 -flto -o f f1.o f2.o
```

or

```
$ riscv32-elf-[gcc|clang] -O2 -flto -o f f1.c f2.c
```

When the `-flto` option is applied to the linker, compiler options starting with `-O`, `-g`, `-f` and `-m` all have to be applied too for the best optimization result.

17.4.2. Notice when applying LTO

Because LTO takes all objects as a single module to perform optimizations, there are some limitations that you need to be aware of:

- Avoid defining the same module name as it's presented in the library. This may confuse LTO when linking objects.
- The symbol listing tool `nm` may not work as expected for programs built with the `-flto` option.
- It is advised that you avoid re-defining reserved identifiers that may be used by the C implementation. These identifiers include all global functions and variables prefixed with an underscore "__" . If you define a function that may be called from library, such as redefining the `_write()` function from MCUlib to customize the output of the printf

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

function, make sure to mark it with "`__attribute__((used))`" so that the function won't be optimized away by LTO. For example,

```
__attribute__((used))
ssize_t _write(int fd, const void *buf, size_t nbytes) {
   /* ... */
}
```

- Please make sure all the modules of the project are included in the build process. If your project has something to do with patch code, which is invisible during LTO process, the patch code module is not supposed to be compiled with the `-flto` option.
- The latest toolchains may have problems linking libraries built with an earlier LTO version. If you get a compilation error indicating the library was generated with an older, not-as-expected LTO version, try resolving it by building the source with an up-to-date toolchain.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

17.5. Auto-vectorization for RISC-V V extension

Both Andes GCC and LLVM compilers support auto-vectorization for RISC-V Vector extension (RVV). To enable auto-vectorization for RVV, you must compile your programs with the option `-mext-vector` or `-mext-vector="EXTENSION"` while having an optimization level of at least `-O2` for LLVM or `-O3` for GCC. For more about `-mext-vector` or `-mext-vector="EXTENSION"`, see Section 2.1.

For example, given a short program "`vadd.c`" as follows,

```
void vadd(int *a, int *b, int *c, int n) {
  for (int i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
  }
  return;
}
```

you can issue as follows to enable auto-vectorization for it:

- GCC: `riscv64-elf-gcc -S -O3 -mext-vector vadd.c`
- LLVM: `riscv64-elf-clang -S -O2 -mext-vector vadd.c`

The assembly, excerpted below, shows vector instructions are generated after the vectorization.

```
        vsetvli a5, zero, e32, m2, ta, mu
.LBB0_7:
        vl2re32.v      v8, (a4)
        vl2re32.v      v10, (t4)
        vadd.vv v8, v10, v8
        vs2r.v  v8, (t2)
        sub     t3, t3, t0
        lea.b.ze       t2, t2, t1
        lea.b.ze       t4, t4, t1
        lea.b.ze       a4, a4, t1
        bnez    t3, .LBB0_7
```

To disable auto-vectorization for the entire program, just compile it with the following options:

- GCC: `-fno-tree-slp-vectorize -fno-tree-vectorize`
- LLVM: `-fno-tree-slp-vectorize -fno-tree-vectorize -mno-implicit-float`

On the other hand, to disable auto-vectorization for a specific loop, use the pragmas "`#pragma clang loop vectorize(disable)`" for LLVM and "`#pragma GCC novector`" for GCC. For example,

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
void foo(int *a, int *b, int *c, int n) {
#pragma clang loop vectorize(disable)
#pragma GCC novector
  // The loop will NOT be vectorized.
  for (int i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
  }
  // The loop will be vectorized.
  for (int i = 0; i < n; i++) {
    b[i] = a[i] + c[i];
  }

  return;
}
```

Note that Andes 27- and 45-series vector processors do not support unaligned vector access. As a result, it is important to ensure that your program won't trigger the compiler to generate unaligned vector load and store instructions. This includes avoiding actions like casting a pointer directing to short to a pointer directing to an integer and then performing a sequence of word copies, as shown in the following code segment. According to the C standards, when the resulting pointer is not correctly aligned for the referenced type, the behavior is undefined. This leads the compiler to assume that the pointer in such a case is aligned for the referenced type. It will therefore vectorize the for loop and generate vector word load and store instructions, resulting in misaligned access exceptions during runtime.

```
void Copy(short x[], short y[], short L) {
  int *tmpx = (int *) x;
  int *tmpy = (int *) y;
  for (short L2 = L >> 1; L2 != 0; L2--)
    *tmpy++ = *tmpx++;
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

17.6. Cache management operations

Andes compiler supports the RISC-V CMO extension (i.e., Cache Management Operations for RISC-V) for programs compiled with the option `-mext-cmo`. However, it does not generate CMO prefetch instructions automatically by default. To utilize these CMO instructions, you have to enable the loop data prefetch optimization or use the CMO intrinsic function. The subsequent sections introduce the usages in detail.

17.6.1. Loop data prefetch optimization

Loop data prefetch optimization allows a compiler to generate instructions to prefetch memory and improve the performance of loops that access large arrays. Both Andes GCC and LLVM compilers support the loop data prefetch optimization. To enable the optimization, you must compile programs with the following options and have an optimization level of `-O1` or greater:

- `-mext-cmo`

  This option allows the compiler to support RISC-V CMO extension.

- `-fprefetch-loop-arrays`

  This option allows the compiler to insert prefetch instructions into loops.

Note that a compiler may not be able to insert prefetch instructions successfully if the loop body is too short. In such a case, you can apply `-funroll-loops` to configure the compiler to unroll loops and create more opportunities for the instruction insertion. For more details about the option, see Section 17.1.2.

The compiler determines the address to prefetch according to the loop size and prefetch distance parameter `X`. GCC and LLVM support separate options to set the prefetch distance parameter `X`, as shown below:

- For GCC:`--param prefetch-latency=X`
- For LLVM: `-mllvm -prefetch-distance=X`

Either of the above options specifies the prefetch distance `X` for the dedicated compiler (GCC/LLVM) to estimate the number of iterations to prefetch ahead in a loop. When `X` is not specified explicitly, it will be set to `480` by default.

The following uses an example to demonstrate how to enable the loop data prefetch

optimization. Given a short program "loop.c" like below,

```
void test(int *a, int *b, int n) {
   for (int i = 0; i < n; i += 2) {
      a[i] = b[i];
   }
   return;
}
```

You can issue as follows to enable the optimization with GCC.

```
$ riscv32-elf-gcc loop.c -S -O1 -mext-cmo -fprefetch-loop-arrays --param
prefetch-latency=480 loop.c
```

Its assembly, excerpted below, shows inserted prefetch instructions from the optimization.

```
prefetch.r 0(a5)
lw  t1,-480(a5)
sw  t1,0(a4)
lw  t2,-472(a5)
sw  t2,8(a4)
```

With the same program example, you can also issue as follows to enable the loop data prefetch optimization with LLVM.

```
$ riscv32-elf-clang loop.c -S -O1 -funroll-loops -mext-cmo -fprefetch-loop-
arrays  -mllvm -prefetch-distance=480 loop.c
```

Its assembly, excerpted below, shows prefetch instructions as well.

```
prefetch.r 704(a2)
lw  a3, 0(a2)
sw  a3, 0(a5)
lw  a3, 8(a2)
sw  a3, 8(a5)
```

17.6.2. CMO prefetch instructions for other instructions

Existing compiler options only support inserting CMO prefetch instructions for loops. To insert prefetch instructions at other positions, you need to utilize the intrinsic function `__nds__prefetch` and apply the compiler option "`-mext-cmo`". For more details about the syntax and usage of the intrinsic function, see section 10.2.2.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18. Advanced programming

This chapter explores some advanced or specialized programming techniques for specific requirements.

18.1. Function attribute and pragma for optimization

Compiler optimization options do not always guarantee performance improvement as they only benefit some functions. In this case, you can use dedicated function attributes or pragma to ensure better performance by enabling or disabling optimization for certain functions.

18.1.1. Function attribute and pragma for optimization in GCC

The attribute `__attribute__((cold))` informs the compiler that the function is unlikely to be executed and should be optimized for code size rather than speed. For example,

```
int handle_err(int, int *) __attribute__((cold));
int handle_err(int id, int *record)
{
  *record = id;
}
```

The attribute `__attribute__((hot))` informs the compiler that the function is a hot spot and should be optimized aggressively for speed. For example,

```
int foo(int, int *, int) __attribute__((hot));
int foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
```

The attribute `__attribute__((optimize()))` specifies optimization flags for a specific function. For example,

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
int foo(int, int *, int) __attribute__((optimize("O0")));
int foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
```

The pragma `#pragma GCC optimize ()` is similar to `__attribute__((optimize()))` but it applies optimization flags to all functions following the pragma. For example,

```
#pragma GCC optimize ("O0")
// All functions following this pragma will be compiled with O0
int foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
```

The arguments of the attribute and pragma must be GCC options that control optimization. The options always start with `-f` or `-O`. For a full list of GCC optimization options, see the GCC online document *Options That Control Optimization*.

In addition, `#pragma GCC optimize ()` can also be used to affect a range of functions, rather than all following functions. Just enclose the range by `#pragma GCC push_options` and `#pragma GCC pop_options`. For example,

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
// Functions within the scope between pragma GCC push_options and
// progama GCC pop_options will be compiled with O0
int foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
#pragma GCC pop_options
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
// bar will be compiled with the option from the command line and will
// not be affected the O0 flag that pragma GCC optimize sets.
int bar(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
```

You can also use `#pragma GCC optimize ()` and `#pragma GCC reset_options` to enclose the affected range. For example,

```
#pragma GCC optimize ("O0")
// Functions following this pragma will be compiled with O0.
int foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
#pragma GCC reset_options
// The flag that pragma GCC optimize sets has been cleared. bar will be
// compiled with the option from the command line and not affected by
// the O0 flag that pragma GCC optimize sets.
int bar(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
```

Note that the attribute `__attribute__((optimize()))` and the pragma `#pragma GCC optimize ()` are not recommended to be used in production code as they do not guarantee consistent effects on all compiler optimizations and may behave differently across different GCC versions.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.1.2. Function attribute and pragma to disable optimization in LLVM

Clang provides the attribute `__attribute__((optnone))` to disable optimizations for a specific function. For example,

```
__attribute__((optnone)) int foo(int, int *, int);
int foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
```

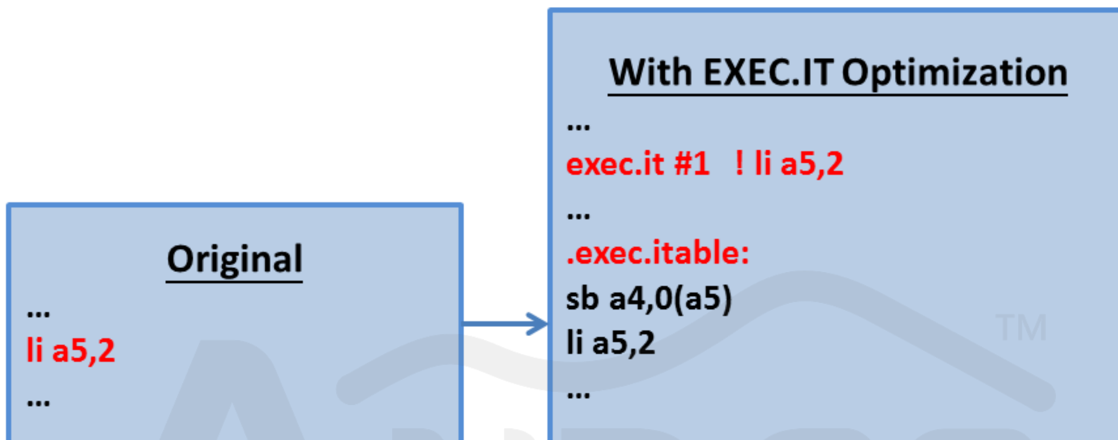
Clang also provides a range-based pragma, `#pragma clang optimize [on|off]`, to disable optimization for a range of function definitions. For example,

```
#pragma clang optimize off
// All functions following this pragma will be decorated with optnone
int foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
#pragma clang optimize on
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.2. Compression optimization through look-up table (EXEC.IT)

Supported by ELF toolchains only, the 16-bit instruction EXEC.IT (Execution on Instruction Table) fetches an indexed instruction from an instruction table of 1024 entries and executes it. When the `-mexecit` option is applied, the compiler will generate a look-up table `exec.itable` and replaces suitable 32-bit instructions with the 16-bit "`exec.it <INDEX>`" in which `<INDEX>` points to a corresponding 32-bit instruction. For example:



The look-up table `exec.itable` is pointed by the `$uitb` register, which needs to be initialized with the symbol `_ITB_BASE_` before the table is used. The initialization should be done in `start.S`. See Appendix I for details.

The look-up table `exec.itable` should also be placed correctly in the linker script file by putting the following line after RO code:

```
KEEP(*( .exec.itable))
```

However, if the linker script file is generated by the LdSaG utility, you can save the trouble.

18.2.1. Export and import

The look-up table `exec.itable` can be exported by a linked module and used by another separately-linked module. This is useful when doing ROM patch. Options "`-Wl, --mexport-execit`" and "`-Wl, --mimport-execit`" are used for the export and import. For example,

```
riscv[32|64]-elf-gcc main_program.c -o main_program.out -mexecit
-Wl,--mexport-execit=exec.itable
riscv[32|64]-elf-gcc rom_patch.c -o rom_patch.out -mexecit
-Wl,--mimport-execit=exec.itable
```

In this example, rom_patch will use the look-up table generated when main_program is compiled.

The options "--mimport-execit" option enable the linker to perform the compression optimization according to the imported look-up table and delete the table from the final executable file. To save the imported table, you can use "--mkeep-import-execit" to keep it in the executable file or use "-Wl,--mexecit-export" to export it to another file.

18.2.2. Look-up table shared by multiple separately-linked program modules

An advanced usage is to have the look-up table shared among multiple separately-linked modules. In this instance, you can use "-Wl,--mupdate-execit" to update the imported table. The options enable the linker to first execute the compression optimization according to the imported table and then perform the regular EXEC.IT optimization for the remaining code. The updated table will be kept in the executable file automatically. You can also use "-Wl,--mexecit-limit=N" to limit the maximum number of look-up table entries for a module. If you want to set the maximum table entries to zero, use "-Wl,--mexecit-limit".

For example, if a library contains functions shared by app-1 and app-2, you may use the following commands to make the look-up table shared among lib, app-1, and app-2. If error messages appear when executing the first command that generates `exec.itable` and `lib.ld`, simply ignore them.

```
//Command to generate exec.itable and lib.ld
riscv[32|64]-elf-gcc lib.c -o lib.out -mexecit -Wl,--mexport-
symbols=lib.ld -Wl,--mexport-execit=exec.itable -Wl,--mexecit-limit=100

//Command to generate lib.out
riscv[32|64]-elf-gcc lib.c -c -o lib.out -mexecit

riscv[32|64]-elf-gcc lib.out app-1.c -o app-1.out -mexecit -Wl,-T,lib.ld
-Wl,--mimport-execit=exec.itable -Wl,--mupdate-execit
-Wl,--mexecit-limit=200 -Wl,--mexport-execit=exec.itable

riscv[32|64]--elf-gcc lib.out app-2.c -o app-2.out -mexecit
-Wl,-T,lib.ld  -Wl,--mimport-execit=exec.itable -Wl,--mupdate-execit
-Wl,--mexecit-limit=200 -Wl,--mexport-execit=exec.itable
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

If the number of translated entries exceed the sum of lib, app-1 and app-2 limits (i.e., 100 + 200 + 200), 1-100 entries are used by lib, 101-300 entries are used by app-1, and 301-500 entries are used by app-2. If lib only use A entries (<100), app-1 only use B entries (<200), and app-2 only use C entries (<200), lib will use entries from 1 to A, app-1 will use entries from (A+1) to (A+B), and app-2 will use entries from (A+B+1) to (A+B+C).

18.2.3. Disable EXEC.IT optimization for specific functions

In some instances, you may want to turn off the EXEC.IT optimization for certain functions when it is default applied to the optimization level you specify. This is because the EXEC.IT optimization, though significantly decreasing the code size, may also result in serious performance degradation at the same time.

The function attribute `no_execit` can be used to disable the EXEC.IT optimization for a specific function when the compilation flag `-mexecit`, `-Os`, `-Os2` or `-Os3` is applied. The following gives an example for its usage:

```
int foo(int, int *, int) __attribute__((no_execit));
int foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.3. Primitive data type "int"

Since most instructions are designed for 32-bit operands in 32/64-bit AndeStar V5-based processor architecture, it is usually better to declare a variable type at least 32 bits long. That is, when the size of variable storage is not a concern, the primitive data type "int" is preferred to those less than 32 bits.

The example below shows the outcome when a variable type less than 32 bits is declared.

```
/* Example-type-1  */
int
func_type_1 (int a, int b, int c)
{
  short e1, e2;

  e1 = a - b;
  e2 = a + b;

  if (e1 > e2)
    return 13;

  return 17;
}
```

The following assembly code is generated when Example-type-1 is compiled with the compiler option "-O1":

```
func_type_1:
    sll       a0,a0,16
    srl       a0,a0,16
    sll       a1,a1,16
    srl       a1,a1,16
    sub       a5,a0,a1
    add       a0,a0,a1
    sll       a5,a5,16
    sra       a5,a5,16
    sll       a0,a0,16
    sra       a0,a0,16
    bgt       a5,a0,.L3
    li        a0,17
    ret
    .align  2
.L3:
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
        li          a0,13
        ret
```

Since the variables "e1, e2" are declared as the type "short" in Example-type-1, the instructions "sll" and “sra” are generated to extend the effective bits of a register to 32 bits so that it can serve as a 32-bit operand for instruction "bgt".

In contrast, in Example-type-2, "e1, e2" are declared as type "int".

```
/* Example-type-2  */
int
func_type_2 (int a, int b, int c)
{
  int e1, e2;
  e1 = a - b;
  e2 = a + b;

  if (e1 > e2)
    return 13;

  return 17;
}
```

The assembly code of Example-type-2 below shows that no extra instruction is needed to adjust the property of variables "e1, e2" for instruction "bgt".

```
func_type_2:
        sub     a5,a0,a1
        add     a0,a0,a1
        bgt     a5,a0,.L3
        li      a0,17
        ret
        .align  2
.L3:
        li      a0,13
        ret
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.4. Primitive data type "unsigned int"

Unsigned integers should obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer. In 64-bit AndeStar V5-based processor architecture, if you declare an unsigned integer variable less than 64 bits long, the compiler will generate extra instructions to perform zero extension to ensure that the result of an unsigned arithmetic operation conforms to the C/C++ language specification. As a result, when the size of variable storage or the range of variable value is not a concern, the primitive data type “unsigned long” or “int” is preferred to unsigned integer less than 64 bits.

The example below shows the outcome when an unsigned integer less than 64 bits is declared.

```
/* Example-type-3  */

 void func_type_3( uint32_t N,  int32_t *C, int16_t *A, int16_t val)
 {
   uint32_t i, j;
   for (i = 0; i < N; i++)
     for (j = 0; j < N; j++)
       C[i*N+j] = (int32_t)A[i*N+j] * (int32_t)val;
 }
```

The following shows the assembly code of Example-type-3 at line 6 when it is compiled with the compiler option "-O3":

```
.L4:
    sll     a5,a4,32
    srl     a5,a5,32
    sll     a6,a5,1
    add     a6,a2,a6
    lh      a6,0(a6)
    sll     a5,a5,2
    mulw    a6,a6,a3
    add     a5,a1,a5
    addw    a4,a4,1
    sw      a6,0(a5)
    bne     a7,a4,.L4
```

Since the parameter “N” and variables “i, j” are declared as the type “unsigned int” in Example-type-3, the instructions “sll” and “srl” are generated to perform zero extension.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

In contrast, in Example-type-4, the parameter "N" and variables "i, j" are declared as the type "int".

```
/* Example-type-4  */
void func_type_4( int32_t N,  int32_t *C, int16_t *A, int16_t val)
{
   int32_t i, j;
   for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        C[i*N+j] = (int32_t)A[i*N+j] * (int32_t)val;
}
```

The assembly code of Example-type-4 below shows that no extra instruction is needed to perform zero extension.

```
L4:
      lh      a4,0(a5)
      add     a6,a6,4
      mulw    a4,a4,a3
      add     a5,a5,2
      sw      a4,-4(a6)
      bne     a7,a5,.L4
```

18.5. Function with variable number of arguments

When there is a need to write a function with variable number of arguments like "prinf()", an ellipsis ("…") can be used to replace the optional arguments. The declaration of such a function requires at least one named argument before the ellipsis to denote the prototype of the list of anonymous arguments, such as "int func(int x, …)".

To load the values of the anonymous arguments, header file "stdarg.h" has to be included first to introduce a special data type va_list and three macros va_start(), va_arg(), and va_end() that manipulate the variable number of arguments.

Data type "va_list" is used to record the current information of the list of anonymous arguments. It has to be initialized by va_start() with the named argument right before the ellipsis. After va_start() is called, the value of each anonymous argument can be loaded sequentially based on the information of "va_list". For each va_start(), va_end() must be invoked in the same function to clean up the argument list allocated in the memory. Between a pair of va_start() and va_end(), va_arg() is called successively to traverse the argument list one by one. Thereby the value of a pointed argument from the list can be loaded by the current variable with a specified type. The below gives an example of how va_list, va_start(), va_arg(), and va_end() work in a function that accepts variable number of arguments.

```
/* Example-va-1  */

#include <stdarg.h>

void my_printf (char* format, ...)
{
  va_list ap;
  int i;
  int c;
  long long int ll;
  double f;

  va_start (ap, format);
  i = va_arg (ap, int);

  /* 'char' is promoted to 'int' when passed through '...'
```

```
     so you should pass 'int' not 'char' to 'va_arg'  */
  c = va_arg (ap, int);

  ll = va_arg (ap, long long int);

  /* 'float' is promoted to 'double' when passed through '...'
    so you should pass 'double' not 'float' to 'va_arg'  */
  f = va_arg (ap, double);

  printf (format, i, c, ll, f);

  va_end (ap);
}

int main ()
{
  my_printf ("Hello: %d %c %lld %f\n", 23, (char) 'X', (long long int)
12399, 3.4f);
  return 0;
}
```

In Example-va-1, one variable "ap" is declared as type "va_list", and it is initialized by va_start() with the last named argument "format". Statement va_arg(ap, int) returns a value of type "int" and updates the content of variable "ap" to point to the next argument from the list. Values of consecutive anonymous arguments can be loaded by successive calls of va_arg() with a corresponding type in turn.

Note that an anonymous argument with type "char" and "short" will be promoted to one with type "int" when it is passed from a caller function to callee function. So is an anonymous argument with type "float" promoted to one with type "double". Thus, when loading values of anonymous arguments, use type "int" or "double" for va_arg() rather than type "char", "short", or "float".

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.6. Inline assembly programming

18.6.1. General

Inline assembly programming is a way a compiler provides to write assembly code embedded in C program. The following displays the basic form of inline assembly programming:

```
__asm__ ("an assembly code template"
         : a list of output operands
         : a list of input operands
         : a list of clobber registers);
```

As shown above, an inline assembly statement starts with "`__asm__ (...)`" or "`asm (...)`" and includes four parts separated by colons: a string of an assembly code template, a list of output operands, a list of input operands, and a list of clobber registers. The first part, an assembly code template, contains the set of assembly instructions and is essential to inline assembly statement. The rest three parts are used to fulfill the instructions and can be optional. The following gives an example of an inline assembly statement that only has a string of assembly code starting with a comment symbol as its output string.

```
__asm__ ("! A test of inline assembly code");
```

Since the compiler can't recognized the output string of an inline assembly statement, it simply outputs that string enclosed in "`#APP`" and "`#NO_APP`" in generated assembly code. Then, the whole assembly code can be validated and assembled by assembler.

An assembly instruction normally has an output operand and two input operands. An operand in an assembly instruction is presented by a symbol "`%`" followed by a number starting from `0`. In Example-Asm-1, "`%0`", "`%1`", and "`%2`" represent three operands and the compiler will replace them from the output operand list to the input operand list when the output string of the assembly code template is generated.

```
/* Example-Asm-1  */

int func_asm_1 (int i, int j)
{
  int ret;

  __asm__ ("add\t%0, %1, %2\n\t"
           "li\tt0, 123\n\t"
           "add\t%0, %0, t0"
```

```
              : "=r" (ret)
              : "r" (i), "r" (j)
              : "t0");

  return ret;
}
```

In the above example, "\n\t" is used to separate an instruction from others and "\t" is to separate an instruction from its first operand in an assembly code template.

Each operand in the input/output operand list is specified by a constraint in double quotes and a C expression in parentheses. In Example-Asm-1, "=r" (ret), "r" (i) and "r" (j) are the cases. A constraint of an operand is used to indicate the addressing mode. Constraint "r" means operands should be placed in general registers and constraint modifier "=" is used for output operands, indicating the operands are write-only.

18.6.2. Symbolic operand name

Another way to specify an operand is to use a symbolic operand name in the form of "[name]" as shown in Example-Asm-2. It's quite flexible to give a symbolic operand name in that it has no relation to any symbol table. Any name is valid no matter it is in C symbol or not, but be sure that no two operands shares the same symbolic name in an asm statement.

```
/* Example-Asm-2  */

int func_asm_2 (int i, int j)
{
  int ret;

  __asm__ ("add\t%[output], %[input_1], %[input_2]"
              : [output] "=r" (ret)
              : [input_1] "r" (i), [input_2] "r" (j));

  return ret;
}
```

18.6.3. Clobber list

In a clobber list, registers or memory are listed to inform a compiler that these items have been modified. Registers used in an assembly code template have to be specified in the clobber list so that the compiler will assume the content of the registers are invalid after the inline assembly statement and generate extra instructions to maintain correct register status. In

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

addition to registers, "memory" can also be listed in a clobber list to make the compiler update memory values.

```
/* Example-Asm-3  */

int func_asm_3 (int i, int j)
{
  int ret;

  __asm__ ("add\t%0, %1, %2\n\t"
           "li\ts0, 12345\n\t"
           "add\t%0, %0, s0"
           : "=r" (ret)
           : "r" (i), "r" (j)
           : "s0");

  return ret;
}
```

With the compiler option "-O1", Example-Asm-3 will be compiled as:

```
func_asm_3:
   addi        sp, sp, -16
   sw          s0, 12(sp)
#APP
   add         a0, $a0, a1
   li          s0, 12345
   add         a0, a0, s0
#NO_APP
   lw          s0, 12(sp)
   addi        sp, sp, 16
   ret
```

In Example-Asm-3, the compiler is informed that "s0" will be clobbered by the inline assembly statement, so it generates instructions to push/pop callee-saved register "s0" in prologue/epilogue in order to satisfy ABI.

18.6.4. Read-write operand

Each operand in the input and output operand list can be referenced by numbers from "0" to "n-1" in increasing order, where n stands for the total number of operands. Thus, a constraint

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

with a number can be used to denote certain operand and furthermore manipulate read-write operands. An operand that has the constraint "0" will be placed in the same location as operand 0, thus specifying a read-write operand. The rest read-write operands can be manipulated likewise. In Example-Asm-4, "1" is used to allow the input operand [read_2] to have the same register as the second output operand [write_2].

```
/* Example-Asm-4  */

int func_asm_4 (int i, int j)
{
  int ret;

  __asm__ ("add\t%[write_1], %[read_1], %[read_2]\n\t"
           "li\tt0, 12345\n\t"
           "add\t%[write_2], %[read_1], t0"
           : [write_1] "=r" (ret), [write_2] "=r" (j)
           : [read_1] "r" (i), [read_2] "1" (j)
           : "t0");

  return ret + j;
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.6.5. Constraint modifier "&"

A compiler may assume that input operands are read before output operands are written and then allocate output operands in the same registers as unrelated input operands. However, such an assumption doesn't apply when there is more than one instruction in the assembler code template. Example-Asm-5 demonstrates this problem.

```
/* Example-Asm-5  */

int func_asm_5 (int i, int j)
{
  int ret1, ret2;

  __asm__ ("li\t%[write_1], 12345\n\t"
           "add\t%[write_2], %[read_1], %[read_2]"
           : [write_1] "=r" (ret1), [write_2] "=r" (ret2)
           : [read_1] "r" (i), [read_2] "r" (j));

  return ret1 + ret2;
}
```

Compile Example-Asm-5 using the option "-O1":

```
func_asm_5:
#APP
   li         a0, 12345
   add        a1, a0, a0
#NO_APP
   add        a0, a0, a1
   ret
```

In the above example, the operand [read_1] uses the same register "a0" as the operand [write_1]. Since the first instruction clobbers the operand [read_1] when writing [write_1], the second assembly instruction gets wrong content of [read_1]. To avoid this problem, apply constraint modifier "&" to an output operand so that the compiler is informed not to allocate the input and output operands in the same registers. As shown in Example-Asm-6, constraint modifier "&" is used to ensure all output operands reside in different registers from input operands.

```
/* Example-Asm-6  */

int func_asm_6 (int i, int j)
{
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
    int ret1, ret2;

    __asm__ ("li\t%[write_1], 12345\n\t"
             "add\t%[write_2], %[read_1], %[read_2]"
             : [write_1] "=&r" (ret1), [write_2] "=&r" (ret2)
             : [read_1] "r" (i), [read_2] "r" (j));

    return ret1 + ret2;
  }
```

The assembly code of Example-Asm-6 shows no problem of overlapping registers:

```
func_asm_6:
#APP
   li        a5, 12345
   add       a4, a0, a1
#NO_APP
   add       a0, a5, a4
   ret
```

18.6.6. Volatile

A compiler may move or delete assembly statements in view of optimization strategy. For example, an inline assembly statement to access hardware status without dependency on any instruction will likely be removed by compiler optimization. To avoid these unwanted optimization effects, use keyword "`__volatile__`" or "`volatile`" after `asm` statement to switch off optimization and preserve the inline assembly code.

```
__asm__ __volatile__ ("ebreak");
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.7. Half-Precision floating point

Andes compiler supports the data type _Float16 and generates instructions in Zfh extension (i.e., half-precision floating-point instructions compliant with the IEEE 754-2008) for programs compiled with the option -mzfh.

Since default argument promotions are only applied to standard floating-point types, _Float16 values are not promoted to double when passed as variadic or untyped arguments. As a consequence, some caution must be taken when using certain library facilities with _Float16. For example, there is no printf format specifier for _Float16, and a _Float16 argument, unlike float ones, will not be implicitly promoted to double when passed to printf. In such a case, you will have to explicitly cast it to double before using it with an %f or similar specifier, as shown in the example below.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   volatile _Float16 a[16];

   for (int i = 0; i < 16; i++)
     a[i] = ((short)(rand() >> 16)) / 16384.0f;

   for (int i = 0; i < 16; i++)
     printf("%2.6f ", (double)a[i]);

   return 0;
}
```

Notice that there is a calling convention issue on v5f toolchains. The compiler will generate soft float library calls to convert between double and _Float16 data. As the soft float library in default v5f toolchains is compiled without the zfh support, the _Float16 data is allocated to GPR. However, when the zfh support is enabled for v5f toolchains, the _Float16 data will be allocated to FPR instead. This calling convention conflict in v5f toolchains will lead to wrong results for programs that have conversion between double and _Float16 data. As a result, when using a v5f toolchain enabled with the zfh support, please avoid the data type double and apply the compiler option "-fsingle-precision-constant" to ensure that floating point constants are treated as single precision.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.8. Brain floating point (BFloat16 or BF16)

BFloat16, a floating point format widely used in deep learning, are supported on Andes processors through one or more of the following schemes:

- RISC-V minimal BFloat16 extension
  Andes compiler generates instructions with RISC-V zfbfmin, zvfbfmin, or zvfbfwma extension for non-arithmetic BF16 operations.

- Andes scalar/vector BFloat16 arithmetic extension
  This extension allows BF16 arithmetic computation in either BFloat16 or Float16 format, depending on the configuration of a specific CSR value.

- Scalar BFLOAT16 conversion extension in AndeStar V5 ISA
  This extension supports instructions to convert between BF16 and FP32 values and provide intrinsics to do the conversions.

The following sections explain how to use and enable each of these schemes with Andes compiler.

18.8.1. RISC-V minimal BFloat16 extension

Andes compiler supports the BF16 floating-point data type `__bf16` and generates instructions with the zfbfmin extension for programs compiled with the option `-mext-bf16min`. Although `__bf16` is viewed as a storage type in this scheme, it does not support arithmetic operations. Instead, the compiler automatically promotes it to the `float32` type for these operations. This allows you to mix `__bf16` and float computations in programs directly, without needing to convert `__bf16` to `float32` using intrinsic functions first. Nonetheless, mixed-type arithmetic operations involving both `__bf16` and `_Float16` are still not allowed.

Andes compiler also supports RISC-V vector minimal BF16 extension for zvfbfmin and zvfbfwma. You can program with associated RVV intrinsics and compile with "`-mext-vector -mext-bf16min`". For details about the BF16 vector intrinsic functions, see the document *RISC-V Vector (V) Extension Intrinsics*.

18.8.2. Andes Scalar/Vector BFloat16 arithmetic extension

In terms of BFloat16 arithmetic computation, Andes scalar/vector BFloat16 arithmetic extension, a mechanism different from RISC-V zfbfmin extension, is deployed. The extension supports all operations specified by half-precision floating point (Float16) scalar and vector instructions defined in the RISC-V specification. It provides the same instructions as Float16, differing only in the data type being handled, and employs the same instruction encoding as Float16 for both scalar and vector instructions. As a result, you can manipulate a CSR value to determine whether to use BFloat16 or Float16 as the floating point (FP) mode. For details about the CSR that controls the FP mode, see the User Miscellaneous Control Register section in *AndeStar V5 System Privileged Architecture and CSR Specification*.

18.8.2.1 Scalar computation on BFloat16

To support BFloat16 arithmetic, the scalar variable data type `__bf16` is upgraded from a storage-only type to an arithmetic type. You can configure the FP mode by invoking the intrinsic functions `__nds__csrs()` (set CSR) and `__nds__csrc()` (clear CSR) and adjusting the first bit within the CSR `NDS_UMISC_CTL`.

To enable the code generation for Andes BF16 arithmetic extension, apply -mbf16ms when compiling. For example,

```
$ riscv64-elf-clang -mbf16ms ./add.c
```

The following gives two examples of switching the FP mode between Float16 and BFloat16 and enabling the scalar BFloat16 extension for compilation.

Example 1: Setting the FP mode to BFloat16

```
$ riscv64-elf-clang -mbf16ms ./add.c

#include <nds_intrinsic.h>
__bf16 add(__bf16 a, __bf16 b){
  __nds__csrs(1, NDS_UMISC_CTL);  //Set FP Mode to BF16
  return a + b;
}

$ riscv64-elf-clang -mbf16m ./matmul_bf16.c
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Example 2: Mixing BFloat16 and Float16 computations

```
$ riscv64-elf-clang -mbf16ms ./mix_add.c
#include <nds_intrinsic.h>

float mix_computation_add(_Float16 a0, _Float16 a1, __bf16 b0, __bf16
b1){
  __nds__csrc(1, NDS_UMISC_CTL);  // set to FP16 mode
  float promote_sum_fp16 = a0 + a1;

  __nds__csrs(1, NDS_UMISC_CTL);  // set to BF16 mode
  float promote_sum_bf16 = b0 + b1;

  __nds__csrc(1, NDS_UMISC_CTL);  // reset to FP16 mode
  return promote_sum_bf16 + promote_sum_fp16;
}
```

18.8.2.2 Vector computation on BFloat16

Vector computation on the BFloat16 data type can be performed in two ways. The first approach is to enable auto-vectorization with the Andes compiler and compile at an optimization level of at least `-O2` for LLVM and `-o3` for GCC. The programs being compiled must also incorporate operations that can be vectorized and operated on the BFloat16 data type. For more about auto-vectorization, see Section 17.5. To enable the BFloat16 auto-vectorization for the compilation, apply the compiler option `-mext-vector` along with `-mbf16ms`. For example,

- GCC: `riscv64-elf-gcc -mbf16ms -mext-vector -O3 ./add.c`
- LLVM: `riscv64-elf-clang -mbf16ms -mext-vector -O2 ./add.c`

The second approach is to use Andes RVV intrinsics for BFloat16 vector computations, reductions, and miscellaneous RVV instructions. These intrinsics thoroughly correspond to the Float16 RVV intrinsics defined in the official RVV Intrinsic Specification and are supported by both Andes GCC and LLVM compilers.

The intrinsic functions for BFloat16 RVV instructions follow the naming scheme of explicit (i.e., non-overloaded) RVV intrinsic functions and include the BFloat16 type (i.e., `bf16`) for

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

the RETURN_TYPE field. For example, while the intrinsic function __riscv_vfmv_v_f_f16m4() is designated for the Float16 vector move instruction (vfmv.v.f), __riscv_vfmv_v_f_bf16m4() is for the BFloat16 vector move instruction.

In addition to non-overloaded RVV intrinsic functions, Andes compilers also support overloaded variants. However, they only check the argument type in the overloaded mechanism, making it impossible to differentiate intrinsics that share identical argument type and intrinsic name, such as vbfloat16m1_t vfcvt_f(vint16m1_t); and vfloat16m1_t vfcvt_f(vint16m1_t);. To address this issue, the following four BFloat16 intrinsic series are renamed using the suffix _bf, rather than _f, to distinguish them from their Float16 counterparts.

Float16	**BFloat16**
vfcvt_f	vfcvt_bf
vfwcvt_f	vfwcvt_bf
vfncvt_f	vfncvt_bf
vfncvt_rod_f	vfncvt_rod_bf

The table below lists all scalable vector data types for BFloat16.

Types	__bf
EMUL=1/8	N/A
EMUL=1/4	vbfloat16mf4_t
EMUL=1/2	vbfloat16mf2_t
EMUL=1	vbfloat16m1_t
EMUL=2	vbfloat16m2_t
EMUL=4	vbfloat16m4_t
EMUL=8	vbfloat16m8_t

The following gives an example of using BFloat16 vector intrinsics to perform matrix multiplication.

```
$ riscv64-elf-clang -mbf16ms -mext-vector ./matmul.c
```

```
#include <stdint.h>
#include <riscv_vector.h>
#include <nds_intrinsic.h>
void ndsv_mat_mul_bf16_v(__bf16 * src1, __bf16 * src2, __bf16 * dst,
uint32_t row, uint32_t col, uint32_t col2)
{
```

```
  __nds__csrs(1, NDS_UMISC_CTL);  // Set FP Mode in BFloat16
  const uint32_t tiling_size = 128;
  int row_4 = row >> 2;
  int max_row = row_4 << 2;

  const __bf16* A = src1;
  const __bf16* B = src2;
  __bf16* C = dst;
  int r, k, kk, cc, vl;
  int kk_tiling_size, cc_tiling_size, max_kk_tiling_size;
  const __bf16* A11, * B11, * A21, * B21, * A31, * B31, * A41, * B41;
  __bf16 * C1, * C2, * C3, * C4;

  for (kk = 0; kk < col; kk += tiling_size) {
   kk_tiling_size = ((kk + tiling_size) > col ? col : (kk +
tiling_size));
   if ((kk + tiling_size) <= col) {
     max_kk_tiling_size = kk_tiling_size - (tiling_size & 3);
   } else {
     max_kk_tiling_size = col - ((col - kk) & 3);
   }

   for (cc = 0; cc < col2; cc += tiling_size) {
     cc_tiling_size = ((cc + tiling_size) > col2 ? col2 : (cc +
tiling_size));
     for (r = 0; r < max_row; r += 4) {
       int mulLen = cc_tiling_size - cc;
       while (mulLen > 0) {
         vl = __riscv_vsetvl_e16m4(mulLen);
         vbfloat16m4_t v_reg0 = __riscv_vfmv_v_f_bf16m4(0.0f, vl);
         vbfloat16m4_t v_reg4 = __riscv_vfmv_v_f_bf16m4(0.0f, vl);
         vbfloat16m4_t v_reg8 = __riscv_vfmv_v_f_bf16m4(0.0f, vl);
         vbfloat16m4_t v_reg12 = __riscv_vfmv_v_f_bf16m4(0.0f, vl);

         A11 = A + r * col + kk;
         A21 = A11 + col;
         A31 = A21 + col;
         A41 = A31 + col;
         B11 = B + kk * col2 + cc_tiling_size - mulLen;
         B21 = B11 + col2;
         B31 = B21 + col2;
         B41 = B31 + col2;
         C1 = C + r * col2 + cc_tiling_size - mulLen;
         C2 = C1 + col2;
         C3 = C2 + col2;
         C4 = C3 + col2;
         for (k = kk; k < max_kk_tiling_size; k += 4) {
           vbfloat16m4_t v_reg16 = __riscv_vle16_v_bf16m4(B11, vl);
           vbfloat16m4_t v_reg20 = __riscv_vle16_v_bf16m4(B21, vl);
           vbfloat16m4_t v_reg24 = __riscv_vle16_v_bf16m4(B31, vl);
           vbfloat16m4_t v_reg28 = __riscv_vle16_v_bf16m4(B41, vl);

           __bf16 tmpa11 = *A11++;
           __bf16 tmpa21 = *A21++;
           __bf16 tmpa31 = *A31++;
           __bf16 tmpa41 = *A41++;
           v_reg0 = __riscv_vfmacc(v_reg0, tmpa11, v_reg16, vl);
           v_reg4 = __riscv_vfmacc(v_reg4, tmpa21, v_reg16, vl);
           v_reg8 = __riscv_vfmacc(v_reg8, tmpa31, v_reg16, vl);
           v_reg12 = __riscv_vfmacc(v_reg12, tmpa41, v_reg16, vl);

           __bf16 tmpa12 = *A11++;
           __bf16 tmpa22 = *A21++;
           __bf16 tmpa32 = *A31++;
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
          __bf16 tmpa42 = *A41++;
          v_reg0 = __riscv_vfmacc(v_reg0, tmpa12, v_reg20, vl);
          v_reg4 = __riscv_vfmacc(v_reg4, tmpa22, v_reg20, vl);
          v_reg8 = __riscv_vfmacc(v_reg8, tmpa32, v_reg20, vl);
          v_reg12 = __riscv_vfmacc(v_reg12, tmpa42, v_reg20, vl);

          __bf16 tmpa13 = *A11++;
          __bf16 tmpa23 = *A21++;
          __bf16 tmpa33 = *A31++;
          __bf16 tmpa43 = *A41++;
          v_reg0 = __riscv_vfmacc(v_reg0, tmpa13, v_reg24, vl);
          v_reg4 = __riscv_vfmacc(v_reg4, tmpa23, v_reg24, vl);
          v_reg8 = __riscv_vfmacc(v_reg8, tmpa33, v_reg24, vl);
          v_reg12 = __riscv_vfmacc(v_reg12, tmpa43, v_reg24, vl);

          __bf16 tmpa14 = *A11++;
          __bf16 tmpa24 = *A21++;
          __bf16 tmpa34 = *A31++;
          __bf16 tmpa44 = *A41++;
          v_reg0 = __riscv_vfmacc(v_reg0, tmpa14, v_reg28, vl);
          v_reg4 = __riscv_vfmacc(v_reg4, tmpa24, v_reg28, vl);
          v_reg8 = __riscv_vfmacc(v_reg8, tmpa34, v_reg28, vl);
          v_reg12 = __riscv_vfmacc(v_reg12, tmpa44, v_reg28, vl);

          B11 += 4 * col2;
          B21 += 4 * col2;
          B31 += 4 * col2;
          B41 += 4 * col2;
        }

        for (; k < kk_tiling_size; k++) {
          vbfloat16m4_t v_reg16 = __riscv_vle16_v_bf16m4(B11, vl);

          __bf16 tmpa11 = *A11++;
          __bf16 tmpa21 = *A21++;
          __bf16 tmpa31 = *A31++;
          __bf16 tmpa41 = *A41++;
          v_reg0 = __riscv_vfmacc(v_reg0, tmpa11, v_reg16, vl);
          v_reg4 = __riscv_vfmacc(v_reg4, tmpa21, v_reg16, vl);
          v_reg8 = __riscv_vfmacc(v_reg8, tmpa31, v_reg16, vl);
          v_reg12 = __riscv_vfmacc(v_reg12, tmpa41, v_reg16, vl);
          B11 += col2;
        }

        vbfloat16m4_t v_reg16 = __riscv_vle16_v_bf16m4(C1, vl);
        vbfloat16m4_t v_reg20 = __riscv_vle16_v_bf16m4(C2, vl);
        vbfloat16m4_t v_reg24 = __riscv_vle16_v_bf16m4(C3, vl);
        vbfloat16m4_t v_reg28 = __riscv_vle16_v_bf16m4(C4, vl);
        v_reg0 = __riscv_vfadd(v_reg0, v_reg16, vl);
        v_reg4 = __riscv_vfadd(v_reg4, v_reg20, vl);
        v_reg8 = __riscv_vfadd(v_reg8, v_reg24, vl);
        v_reg12 = __riscv_vfadd(v_reg12, v_reg28, vl);

        __riscv_vse16(C1, v_reg0, vl);
        __riscv_vse16(C2, v_reg4, vl);
        __riscv_vse16(C3, v_reg8, vl);
        __riscv_vse16(C4, v_reg12, vl);
        mulLen -= vl;
      }
    }

    for (; r < row; r++) {
      int mulLen = cc_tiling_size - cc;
      while (mulLen > 0) {
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```
          vl = __riscv_vsetvl_e16m4(mulLen);
          vbfloat16m4_t v_reg0 = __riscv_vfmv_v_f_bf16m4(0.0f, vl);

          A11 = A + r * col + kk;
          B11 = B + kk * col2 + cc_tiling_size - mulLen;
          B21 = B11 + col2;
          B31 = B21 + col2;
          B41 = B31 + col2;
          C1 = C + r * col2 + cc_tiling_size - mulLen;
          for (k = kk; k < max_kk_tiling_size; k += 4) {
            vbfloat16m4_t v_reg16 = __riscv_vle16_v_bf16m4(B11, vl);
            vbfloat16m4_t v_reg20 = __riscv_vle16_v_bf16m4(B21, vl);
            vbfloat16m4_t v_reg24 = __riscv_vle16_v_bf16m4(B31, vl);
            vbfloat16m4_t v_reg28 = __riscv_vle16_v_bf16m4(B41, vl);

            __bf16 tmpa11 = *A11++;
            __bf16 tmpa12 = *A11++;
            __bf16 tmpa13 = *A11++;
            __bf16 tmpa14 = *A11++;
            v_reg0 = __riscv_vfmacc(v_reg0, tmpa11, v_reg16, vl);
            v_reg0 = __riscv_vfmacc(v_reg0, tmpa12, v_reg20, vl);
            v_reg0 = __riscv_vfmacc(v_reg0, tmpa13, v_reg24, vl);
            v_reg0 = __riscv_vfmacc(v_reg0, tmpa14, v_reg28, vl);
            B11 += 4 * col2;
            B21 += 4 * col2;
            B31 += 4 * col2;
            B41 += 4 * col2;
          }
          for (; k < kk_tiling_size; k++) {
            vbfloat16m4_t v_reg20 = __riscv_vle16_v_bf16m4(B11, vl);
            __bf16 tmpa11 = *A11++;
            v_reg0 = __riscv_vfmacc(v_reg0, tmpa11, v_reg20, vl);
            B11 += col2;
          }
          vbfloat16m4_t v_reg16 = __riscv_vle16_v_bf16m4(C1, vl);
          v_reg0 = __riscv_vfadd(v_reg0, v_reg16, vl);
          __riscv_vse16(C1, v_reg0, vl);
          mulLen -= vl;
        }
      }
    }
  }
}
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.8.3. Andes Scalar BFLOAT16 conversion extension (XAndesBFHCvt)

This extension supports instructions to convert between BF16 values and FP32 values. For programs compiled with the option `-mbf16`, you will need to use intrinsic functions `__nds_fcvt_bf16_s` and `__nds_fcvt_s_bf16` to perform the conversion manually. For more details about the two intrinsic functions, see `__nds_fcvt_s_bf16` and `__nds_fcvt_bf16_s` in Section 10.2.3. The following is an usage example of the intrinsic functions.

```
#include <nds_intrinsic.h>
#include <stdlib.h>
int main ()
{
    float fa = 2.5;
    __bf16 bfa;

    bfa = __nds_fcvt_bf16_s(fa);
    fa =  __nds_fcvt_s_bf16(bfa);
    if(fa != 2.5)    abort ();
    else             exit (0);
}
```

NOTE

For Andes processors that support both the RISC-V minimal BFloat16 extension and this conversion extension for non-arithmetic operations, it is strongly recommended to prioritize the RISC-V minimal BF16 extension. This is because the RISC-V minimal BF16 extension not only covers the conversion between BF16 and F32 but also provides enhanced compiler support.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.9. Porting to GCC

This section describes common issues when porting to a different version of GCC and provides solutions for them.

- Linker error caused by the change of the default option `-fcommon` to `-fno-common` in GCC 10:
  A common mistake made when using C is omitting `extern` when declaring a global variable in a header file. If the header is included by several files, it will result in multiple definitions of the same variable. This problem, though ignored by previous GCC versions, is reported as a linker error by GCC 10, which is default applied with `-fno-common` rather than `-fcommon`. To avoid the error in GCC 10 or later versions, do not have tentative definitions like below when declaring global variables in header files

  ```
  int x;
  ```

  but use `extern` like below instead

  ```
  extern int y;
  ```

  and ensure each global is defined in exactly one C file.

  For a code compiled without `-fcommon` (e.g., using GCC 10 or above), you can use `__attribute__((__common__))` to place tentative definitions of particular variables in a common block. For legacy C code where all tentative definitions should be placed into a common block, you may compile it with `-fcommon` as a workaround.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.10. Differences between GNU and LLVM compilers and linkers

18.10.1. GCC vs. Clang

Clang tries to be compatible with GCC as much as possible, but some GCC features are still unsupported or incompatible in Clang. The differences between GCC and Clang are listed below:

- Some GCC extensions are not implemented in Clang yet. For more, but not a complete list, of unimplemented GCC extensions in Clang, see the Clang document.

- Clang does not support nested functions, which are a complex feature used infrequently and unlikely to be implemented anytime soon. In C++11, nested functions can be emulated by assigning lambda functions to local variables.

- Clang does not support GCC built-in functions `__builtin_va_arg_pack` and `__builtin_va_arg_pack_len`.

- Clang replaces the GCC built-in function `__builtin_shuffle` with `__builtin_shufflevector`.

- Clang does not support the GCC extension that allows variable-length arrays in structures.

- Andes LLVM has not supported gcov-based profiling yet.

- It is an undefined behavior in C++ programs when a function must return a value but doesn't. Clang will handle it by adding `abort()` to the end of the function.

- Minor precision issues and binary discrepancies may be found between LLVM on Linux and Windows (MinGW). It is because the compiler performs constant folding for floating-point expressions, which calculates the determinable expressions before code execution. The calculations are done by Glibc on Linux to calculate and UCRT on Window, and the algorithm differences in the two libraries may thus result in the output discrepancies. GCC, in contrast, shows consistent results on Linux and Windows because it uses MPFR for the calculations.

- In AndeSight v5.4.0, LLVM does not support standard vector calling convention variant, whereas GCC does. Therefore, ensure that object files compiled with GCC are not linked with those compiled with LLVM when using AndeSight v5.4.0.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

- According to the RISC-V Vector Extension Specifications, if the `vill` bit is set, any attempt to execute a vector instruction that depends upon `vtype` will raise an illegal instruction exception. However, some instructions, such as `vset{i}vl{i}` and whole register loads and stores, do not depend on `vtype`, although whole register move instructions remain `vtype`-dependent.

  GCC from AndeSight v5.4.0 or later versions can work around the illegal instruction exception for whole register move instructions by ensuring that a `vset{i}vl{i}` instruction is executed beforehand. This guarantees the `vsetvl` insertion and clears the `vill` bit to 0 for the whole register move. For LLVM in AndeSight v5.4.0, in contrast, such a workaround is not supported.

18.10.2. LD vs. LLD

LLD is designed to be a drop-in replacement of the GNU linker LD, and is therefore highly compatible, though not entirely identical. Minor differences exist between the two linkers. For details about the differences, see the article LLD and GNU Linker Incompatibilities.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

18.11. -march arguments implied by -mcpu

Each -mcpu=PROCESSOR is implied with certain -march arguments by default. If you do not explicitly specify -march arguments for your program, Andes compiler will compile it with the default -march arguments corresponding to the specified -mcpu=PROCESSOR. To check the default -march arguments for a specific Andes processor, follow the steps below:

Step 1 Compile the program (e.g., hello.c) using the option -mcpu to generate an object file.

```
$ riscv[32|64]-elf-[gcc|clang] -c -mcpu=PROCESSOR hello.c -o hello.o
```

Step 2 Use readelf to dump the architecture attribute of the object file.

```
$ riscv[32|64]-elf-readelf -A hello.o
```

For example, with a program hello.c and the toolchain nd32le-elf-mculib-v5, you can use the following commands to check the default -march arguments for AndesCore N25:

```
$ riscv32-elf-gcc -c -mcpu=n25 hello.c -o hello.o
$ riscv32-elf-readelf -A hello.o
```

Then, you will get the following message, where parts highlighted in red fonts display the architecture attribute of the generated object (hello.o) and the default -march arguments for AndesCore N25 used to compile.

```
ttribute Section: riscv
File Attributes
  Tag_RISCV_stack_align: 16-bytes
  Tag_RISCV_arch:
 "rv32i2p1_m2p0_c2p0_zicsr2p0_zifencei2p0_zmmul1p0_zca1p0_xandes5p0_xand
 escodense1p0_xandesperf1p0"
  Tag_RISCV_unaligned_access: Unaligned access
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Appendix

Appendix I. start.S

The file `start.S` in V5 startup demo projects includes the following components:

- trap entry examples,
- vector table for interruptions (vectored external PLIC or CLIC interrupts), and
- low-level initialization for C programs.

The trap entry examples illustrate dispatch handling for trap events, from assembly code to C functions. You may override the `trap_handler()` weak function, depending on your requirements.

The vector table `__vectors` and interrupt dispatch examples show the vector service routine for vectored external PLIC or CLIC interrupts, from assembly code to C functions. You may override any weak function of an interrupt vector service routine depending on your needs. For example, you may use your own handler to replace the vector service routine to the PLIC or CLIC interrupt source 1 by overriding the `entry_irq1()` weak function in `<PROJECT>/src/<PLATFORM>/interrupt.c`. Please note that if the PLIC/CLIC device supports a number ***N*** of interrupt sources, a minimum alignment of "*2^ceiling(log2(**N**)) x 4*" bytes is required for the vector table `__vectors`. In `start.S`, the number of interrupt sources for PLIC is set to 32 and that for CLIC to 83 by default, so the vector table `__vectors` is aligned to 128 and 512 bytes for PLIC and CLIC respectively. If your PLIC/CLIC supports a different number of interrupt sources, make sure to modify the alignment for the vector table `__vectors` in `start.S`.

In addition, `start.S` also invokes a code segment of low-level initialization for C programs. The following bullets explain the initialization code segment in `start.S`, including the special code sequence, the used symbols, and their meanings:

- Symbol `__global_pointer$`

  The instruction sequence relating to `__global_pointer$` is to initialize the global data pointer register `gp (x3)` with the value of `__global_pointer$`. The code is as follows:

```
.option push
.option norelax
```

```
la gp, __global_pointer$
.option pop
```

The symbol `__global_pointer$` is the address in the middle of the data sections. The linker places scalar data around to enable easy access using gp-based load/store instructions and the calculation of addresses using gp-based add instructions.

- Symbol `_ITB_BASE_`

The instruction sequence related to `_ITB_BASE_` is used to initialize the instruction table register `uitb` (user mode CSR) with the value of `_ITB_BASE_`. The symbol `_ITB_BASE_` is the base address of the instruction table required for the instruction `exec.it`. When the linker performs code size optimization, it automatically assigns the value of `_ITB_BASE_`, fills the corresponding table with useful instructions, and generates `exec.it`.

- Symbol `_stack`

The instruction sequence related to `_stack` is used to initialize the stack pointer register `sp (x2)` with the value of `_stack`. The symbol `_stack` is the start address of the stack used by the C compiler to pass function parameters, local variables, and return values. The following code is used to avoid loading the address into `sp` directly:

```
la t0, _stack
mv sp, t0
```

The linker obtains its value from the linker script. The stack proceeds from high addresses to low addresses when performing function calls; therefore, the initial stack address is normally set to the highest address in the program data memory.

If you assign the stack value using “`PROVIDE(_stack = . );`” in the linker script, make sure to include “`. = ALIGN(16) ;`” before this statement to align `sp` to 16 bytes.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Appendix II. Programming tips

Moving libc.a to the beginning of text section

The static libraries are normally at the end of text section. During the process of symbol resolution using static libraries, linker scans the object files and archives from left to right as input on the command line. If the input is an archive, linker scans through the list of member modules that constitute the archive to match any unresolved symbols. That explains why static libraries are placed at the end of the linker commands.

There are several methods to move `libc.a` to the beginning of text section. The following is an example achieved via modification of the linker script:

```
.text  :
{     /* output section rule */
       /* exclude file input section rule */
      *(EXCLUDE_FILE(<YOUR_APPLICATION_OBJECT_FOLDER>/*).text
       /* default input section rule */
        *(.text)
}
```

The above modified linker script forces the object files under your application object folder to be excluded in the beginning of text section, thereby enabling linker to place `libc.a` in the beginning of text section.

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Displaying register information and debugging on reset by GDB commands

Andes provides the following GDB commands to display register information:

`info registers`	lists all general purpose registers (GPR) and their contents for selected stack frame.
`info registers all`	lists all registers and their contents.

Andes also provides the following system-related GDB command to debug on reset:

`reset-and-hold`	resets the target system and holds PC at the base address of the reset vector.

This command makes the debugger hold a processor right after the reset of the debugging target and is especially useful for boot code development.

NOTE

To use GDB to set registers in a 64-bit AndesCore processor, make sure you add an "L" suffix to constant values as follows:

```
set $dexc2dbg = 0x1000L
```

The information contained herein is the exclusive property of Andes Technology Corporation. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Appendix III. Troubleshooting

Error message “Relocation truncated to fit”

The error message appears when values being stored for relocation exceed the range the target can represent. Its possible causes and workarounds are summarized in the following table. If you can’t resolve the error message using the following workarounds and want to disable the check for relocation truncated to fit, just compile with the options “`-Wl, --mno-truncation-check`”.

Table 26. Causes and workarounds of “Relocation truncated to fit”

Cause	**Workaround**
Overflow/underflow caused by excessive expression values, such as “`.word .L10-.L4`”	• Enlarge the data type, such as changing “`.word .L10-.L4`” to “`.dword .L10-.L4`”. • Shorten the expression values using measures like rearranging the output sections.
The address offset exceeds the limit allowed by the instruction, such as ±1MB for the instruction “`j farfaraway`”.	• Shorten the offset between the caller and the callee • Add trampoline stubs • Invoke the jump/call macro instead of using the `j/jal` instruction
In non-Sv32 virtual memory modes like Sv39, address offsets exceed the limits allowed by some instructions, such as “`jal far`”.	Set the code model to large using the option “`-mcmodel=large`” in non-Sv32 modes.
Relaxations of fixed sections, such as “`.data 0x100000`” in a linker script	Disable the relaxations by applying the options “`-mno-relax -mno-gp-insn-relax`”.
The jump table generated for switch statements by the compiler includes long offsets between CODE and RODATA sections. This results in overflow/underflow issues similar to those caused by excessive expression values.	Disable the jump table by applying the option “`-fno-jump-tables`”.