Andes Programming Guide for ISA V3

Document Number PG010-16

Date Issued 2017-08-11

Copyright Notice

Contact Information

Should you have any problems with the information contained herein, you may contact Andes Technology Corporation through:

e-mail – 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
1.6	2017/08/11	1. Added N15 and D15 as supported Andes cores (Table 2, Section 2.2.1 and 12.2) 2. Added intrinsic functions for coprocessor ISA extension (Table 20 and Section 12.2.11) 3. Added "NOLOAD" to input_section_description of the SaG script format for marking an section not to be loaded at runtime (Section 15.1.2.4)
1.5	2017/03/28	1. Changed the document template to V11 2. Added descriptions for ISA V3m+ (Table 1, Section 4.2 and Section 12.2) 3. Added a compiler option "-munalign-access" and updated the possible values for "-march=" and "-mcpu=" (Section 2.2.1 and 2.2.2) 4. Added two predefined macros "NDS32_EXT_DSP" and "NDS32_EXT_ZOL" (Section 9.2) 5. Removed the limitation on ROM and flash address space from the implementation of ROM patching using indirect call functions and added a compilation flag "-mict-model=[small|large]" for the implementation. (Section 11, 11.1.2, 11.1.3) 6. Added N650, N820, E830, D10 to descriptions of Andes intrinsic functions and removed those for N12 (Section 12.2) 7. Corrected the example of __nds32__tlbop_trd (Section 12.2.8) 8. Added the nds_ldsag template for the Windows environment and updated the default file name of the linker script generated by nds_ldsag (Section 15.2)
1.4	2016/4/21	1. Added D1088 as a core supporting FPU, coprocessor, and saturation ISA extension. (Table 2) 2. Introduced two ROM patching approaches: indirect call functions and function table mechanism (Ch. 1, 11) 3. Gave example lists of AndesCores supporting V3 and V3m ISA. (Section 12.2) 4. Corrected the descriptions in "Supported CPUs" for all intrinsic functions. (Section 12.2.1, 12.2.2, 12.2.3, 12.2.4, 12.2.5, 12.2.6) 5. Extended intrinsic functions to support up to 32 interrupts. (Section 12.2.10) 6. Extended intrinsic functions to access the following system registers:

Rev.	Revision Date	Revised Content
		INT_MASK2, INT_PEND2, and INT_PRI2. (Section 12.2.10) 7. Clarified the usage and description of __nds32__set_pending_swint and __nds32__clr_pending_swint. (Section 12.2.10) 8. Added an intrinsic function "__nds32__clr_pending_hwint" to clear the pending status for edge-triggered HW interrupts. (Section 12.2.10) 9. Separated the descriptions of __nds32__get_pending_int from those of __nds32__get_all_pending_int since the latter intrinsic is deprecated. (Section 12.2.10) 10. Added an intrinsic function "__nds32__get_trig_type" to access Interrupt Trigger Type Register and updated Table 19. (Section 12.2.10) 11. Added descriptions for memory allocation functions (Ch. 18) 12. Removed "-fno-delete-null-pointer-checks" from the default applied option at -O0, -Og and -O1 (Table 27)
1.3	2016/2/19	1. Removed the note about the Virtual Hosting limitation when syscall is used in ISR and advised users not to redirect outputs when Virtual Hosting is enabled. (Chapter 18) 2. Noted that _malloc_r() and _free_r() may be called automatically when Virtual Hosting is enabled (Chapter 18) 3. Removed the note about the Virtual Hosting limitation when syscall is used in ISR and advised users not to redirect outputs when Virtual Hosting is enabled. (Chapter 18)
1.2	2015/07/28	4. Added "INCLUDE" for including other linker scripts to the SaG header syntax (Section 15.1.2.1) 5. Added two optimization options "-malign-functions" and "-malways-align" (Section 19.1.2 and 19.1.6, Table 27) 6. Added DSP extension and ZOL to Table 1 and Table 2 7. Modified the description of the input "critical" in C language ISR (Section 10.2 and 10.3) 8. Updated supported compiler options (Section 2.2.1) 9. Added "-m[no-]dsp-ext" and "-m[no-]zol-ext" to supported assembler options (Section 2.2.2) 10. Added -fno-delete-null-pointer-checks to Table 27 11. Noted the applied option differences between BSP v3.2 and BSP v4.0 (Section 19.1.7)

Rev.	Revision Date	Revised Content
		12. Added detailed descriptions about the ZOL optimization (Section 19.4) 13. Noted the usage of "-mcmodel", "-mvh", or "-mext-zol" during compilation and linking. (Section 2.2.1)
1.1	2015/04/10	1. Changed "ldsag" to "LdSaG" and "SAG" to "SaG" (Chapter 15) 2. Added syntax checking to "What's New" section (Section 1.1) 3. Added EXCLUDE_FILE to input section descriptions of SaG syntax (Section 15.1.2.4)
1.0	2015/01/26	1. Added two intrinsic functions __nds32__mtsr_isb() and __nds32__mtsr_dsb() (Section 12.1 and 12.2.2) 2. Added deprecated instructions in typographical convention index 3. In MCUlib, changed the modifier "N" to "ll" and added "F" as a conversion supportive character. Besides, changed the supportive character for the precision field as "(.precision)". (Section 17.2) 4. Added a note about the linking problem when applying -flto to a program where printf() will be redirected from libc.a by nds32_write() (Section 19.8.2) 5. Moved the description of adding -fno-omit-frame-pointer to show $fp in stack frame before the explanations about prologue and epilogue (Section 8.2.1.2) 6. Re-organized the descriptions about passing the result in memory (Section 8.2.1.3) 7. Added that -finline-functions is an enabled option at -O3 by default and may cause the increase of code size (Section 19.1.2 and 19.1.6) 8. Added a performance optimization option "-ftree-switch-shortcut" (Section 19.1.2) 9. Added nds_write() redirected from libc.a as an example to use __attribute__((used)) (Section 19.8.2) 10. Added a note to use correct signedness for arguments and return values when calling intrinsic functions (Ch. 11) 11. Added notes to explain what "nds32_nmih", ".nds32_wrh" and ".nds32_jmptbl" sections are for to C-ISR implementation. (Section 10.1, 10.2, 10.3) 12. Added explanations for optimization options "-fno-delete-null-pointer-checks" and "-fno-strict-aliasing" and "-fwrapv" (Section 0) 13. Added a summary about optimization levels (Section 19.1.6) and

Rev.	Revision Date	Revised Content
		added –Og to Table 27 14. Added Saturation Arithmetic ISA Extension to Table 2
0.5	2014/09/19	Document creation. For major features in BSP v4.0 and differences from earlier versions, please refer to Section 1.1 What's New.

Table of Contents

List of Tables

List of Figures

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**
Deprecated instructions	Lucida Console	Normal	11	Dimmed Indigo
Note or warning	Georgia	Normal	12	Red
Hyperlink	Georgia	Underlined	12	Blue

1. Overview

Andes toolchain is part of Andes Board Support Package (BSP) and AndeSight™, an integrated development environment for software development. It is mainly used for compiling, assembling, and linking users' C/C++ and assembly programs and generating executables of AndeStar™, Andes' 16/32-bit mixable instruction set architecture. For detailed information about AndeSight and AndeStar, please refer to *AndeSight User Manual* and *AndeStar Instruction Set Architecture Manual*.

Andes toolchain is built from GNU, thus the options of gcc, as, and ld are inherited. In addition to GNU-based options, Andes specific options are provided for some unique features such as performance and code size tradeoff of AndeStar.

Andes library support includes glibc, uClibc, Newlib and MCUlib. Glibc and uClibc are for OS-based applications and the other two are for non-OS applications. Newlib is an open source project and C library intended for use on embedded systems library. Based on Newlib, MCUlib is a library with Andes optimization enhancement for MCU applications and small code size.

This document focuses on the usages of compiler and assembler for toolchains of ISA V3. For toolchains based on ISA V1 or V2, please refer to *Andes Programming Guide for ISA V1 and V2*.

The following outlines the structure of this document:

- Chapter 2, 3 and 4 are simplified descriptions of AndeStar and basic usage of toolchains.
- Chapter 5, 6, and 7 describe the pseudo-ops, pseudo-instructions, and macros. Programmers can manage and write assembly with these capabilities.
- Chapter 8 describes Application Binary Interface (ABI).
- Chapter 9 describes Andes specific features.
- Chapter 10 describes Andes C language extension for interrupt service routine.
- Chapter 11 describes ROM patching approaches.
- Chapter 12 describes Andes intrinsic functions for programming respectively.
- Chapter 13 describes user and kernel space. OS or system programmers should find this chapter important when configuring Andes CPUs for interruption, MMU, ICE, local memory, and so on.

- Chapter 14 describes the static and dynamic linking and loading.
- Chapter 15 introduces a simple mechanism to generate linker scripts.
- Chapter 16 describes the object file format.
- Chapter 17 describes Andes MCUlib.
- Chapter 18 depicts Virtual Hosting.
- Chapter 19 introduces advanced programming optimization in coding level.

1.1. What's New Since BSP v4.0

The following summarizes the major enhancements in V3-family toolchains since BSP v4.0:

- **Command line options**: The compilation options, including compiler, assembler and linker options, since BSP v4.0 all follow GNU usage conventions. Post-optimization options along with some options in earlier versions are deprecated. Please refer to Section 2.2 and its subsections for the up-to-date options.
- **Operating System Reserved Registers $p0 and $p1**: For toolchains of BSP v4.0 or later versions, `$p0` and `$p1` are not recommended for use in user code. Please refer to Section 3.2.5 for usage notes about the two registers.
- **Application binary interface (ABI)**: A new ABI "ABI2FP+" is defined for floating-point toolchains (v3f and v3s) since BSP v4.0. For details, please refer to Section 8.2.2.
- **Andes pre-defined macros**: Starting from BSP v4.0, the names of Andes pre-defined macros are revised for conforming to the GCC coding conventions. See Section 9.2 for a complete list of updated Andes pre-defined macros and Section 9.2.1 for the deprecated list.
- **Virtual Hosting**: The Virtual Hosting support is implemented in standard library rather than in AICE controller program (ICEman). Please see Chapter 18 for details.
- **More syntax checking**:
  - The second operand of pseudo instruction "la" now can only accept symbol reference. Using immediate value is invalid and reported as an error.
  - In the assembly macro definition, you have to use "\" character as prefix to evaluate arguments. See Section 7.2 for details.
  - The constant suffix (e.g., "L", "UL") is used in C language. If it appears in assembly code, the assembler will help to report error.
  - Compiler now is able to report more warnings if there may be potential issues in users' programs.

2. Getting Started

2.1. Andes Instruction Set Architecture and Instructions

Andes defines three versions of baseline instruction set, denoted by the version numbers 1 to 3. Basically the later versions are the upgrade and extension of the previous versions. This document is specialized for programming with ISA V3 family, including v3, v3j, v3f, v3s and v3m toolchains. You may refer to Table 1 for instructions specific to each Andes V3 toolchain implementation version and Table 2 for extended instruction sets and their supported AndesCores.

Table 1. Instructions Specific to Andes V3 Toolchain Implementation Versions

AndeStar ISA Features		**Andes Toolchain Implementation Versions**				
Name	**Reference**	**v3**	**v3j**	**v3f**	**v3s**	**v3m**
Baseline V3	***AndeStar ISA Architecture Manual***	■	■	■	■	
Baseline V3m/*V3m+	***AndeStar ISA V3m Specification***					■
Reduced_Regs (16 registers)	***AndeStar ISA Architecture Manual***		■			■
STRING		■	■	■	■	
PE1		■	■	■	■	
PE2		■	■	■	■	
SP floating-point	***AndeStar ISA FPU Extension Manual***			■	■	
DP floating-point				■		
DSP extension	***AndeStar DSP ISA Extension Specification***	■	■	■	■	
ZOL		■	■	■	■	

* V3m+ ISA is a V3m ISA plus additional instructions for even better code size compaction when the code size optimization option "`-Os2`" or "`-Os3/-Os`" is applied. For V3m+ processors, please use the V3m toolchain and add "`–march=v3m+`" to both compiler and linker options. AndeSight IDE users can select chip profiles for V3m+ CPU cores to enable the option "`–march=v3m+`".

Table 2. ISA Extensions and Supported AndesCores

AndeStar ISA Extension	**Reference**	**Supported AndesCores**
Audio	***AndeStar Instruction Set Architecture Audio Extension Manual***	**N968, N1068**
FPU	***AndeStar Instruction Set Architecture FPU Extension Manual***	**N1068, N1337, N15, D1088, D15**
COP_ISA	***AndeStar Instruction Set Architecture Coprocessor Extension Manual***	**N1068, N1337, D1088**
Saturation	***AndeStar Saturation Arithmetic ISA Extension Specification***	**N968, N1068, N1337, N15, D1088, D15**
DSP extension and ZOL	***AndeStar DSP ISA Extension Specification***	**D1088, D15**

2.2. Command Line Options

Environment variable $PATH is suggested to include the path to Andes GNU toolchain executables. For example,

```
mypc> PATH=/home/users/bsp412/nds32le-elf-newlib-v3/bin:$PATH
mypc> echo $PATH
/home/users/bsp412/nds32le-elf-newlib-v3/bin:/bin:/usr/bin
```

2.2.1. Compiler Options

To get a list of all supported options, use command:

```
mypc> nds32le-elf-gcc --help
```

```
Usage: nds32le-elf-gcc [options] file...
Options:
 -pass-exit-codes            Exit with highest error code from a phase
 --help                      Display this information
 --target-help               Display target specific command line options
 --help={common|optimizers|params|target|warnings|[^]{joined|separate|undocumented}}[,...]
                             Display specific types of command line options
 (Use '-v --help' to display command line options of sub-processes)
 --version                   Display compiler version information
 -dumpspecs                  Display all of the built in spec strings
 -dumpversion                Display the version of the compiler
 -dumpmachine                Display the compiler's target processor
 -print-search-dirs          Display the directories in the compiler's search
                             path
 -print-libgcc-file-name     Display the name of the compiler's companion
                             library
 -print-file-name=<lib>      Display the full path to library <lib>
 -print-prog-name=<prog>     Display the full path to compiler component
                             <prog>
 -print-multiarch            Display the target's normalized GNU triplet, used
                             as a component in the library path
 -print-multi-directory      Display the root directory for versions of libgcc
 -print-multi-lib            Display the mapping between command line options
                             and multiple library search directories
 -print-multi-os-directory   Display the relative path to OS libraries
 -print-sysroot              Display the target libraries directory
 -print-sysroot-headers-suffix  Display the sysroot suffix used to find headers
```

```
-Wa,<options>                Pass comma-separated <options> on to the
                             assembler
-Wp,<options>                Pass comma-separated <options> on to the
                             preprocessor
-Wl,<options>                Pass comma-separated <options> on to the linker
-Xassembler <arg>            Pass <arg> on to the assembler
-Xpreprocessor <arg>         Pass <arg> on to the preprocessor
-Xlinker <arg>               Pass <arg> on to the linker
-save-temps                  Do not delete intermediate files
-save-temps=<arg>            Do not delete intermediate files
-no-canonical-prefixes       Do not canonicalize paths when building relative
                             prefixes to other gcc components
-pipe                        Use pipes rather than intermediate files
-time                        Time the execution of each subprocess
-specs=<file>                Override built-in specs with the contents of
                             <file>
-std=<standard>              Assume that the input sources are for <standard>
--sysroot=<directory>        Use <directory> as the root directory for headers
                             and libraries
-B <directory>               Add <directory> to the compiler's search paths
-v                           Display the programs invoked by the compiler
-###                         Like -v but options quoted and commands not
                             executed
-E                           Preprocess only; do not compile, assemble or link
-S                           Compile only; do not assemble or link
-c                           Compile and assemble, but do not link
-o <file>                    Place the output into <file>
-pie                         Create a position independent executable
-x <language>                Specify the language of the following input
                             files.
                             Permissible languages include: c c++ assembler
                             none 'none' means revert to the default behavior
                             of guessing the language based on the file's
                             extension

Options starting with -g, -f, -m, -O, -W, or --param are automatically passed on to
the various sub-processes invoked by nds32le-elf-gcc.  In order to pass other options
on to these processes the -W<letter> options must be used.
```

For target specific options, enter:

```
mypc> nds32le-elf-gcc --target-help
```

```
The following options are target specific:
 -EB                         Generate code in big-endian mode.
```

```
  -EL                       Generate code in little-endian mode.
  -G<number>                Put global and static data smaller than <number>
                            bytes into a special section (on some targets)
  -m16-bit                  Generate 16-bit instructions.
  -malign-functions         Align function entry to 4 byte.
  -malways-align            Always align function entry, jump target and return
                            address.
  -march=                   Specify the name of the target architecture.
  -mcache-block-size=       Specify the size of each cache block, which must be a power
                            of 2 between 4 and 512.
  -mcmodel=                 Specify the address generation strategy for code model.
  -mcmov                    Generate conditional move instructions.
  -mconfig-fpu=             Specify a fpu configuration value from 0 to 7; 0-3 is as
                            FPU spec says, and 4-7 is corresponding to 0-3.
  -mconfig-mul=             Specify configuration of instruction mul: fast1, fast2
                            or slow. The default is fast1.
  -mconfig-register-ports=  Specify how many read/write ports for n9/n10 cores.  The
                            value should be 3r2w or 2r1w.
  -mcpu=                    Specify the cpu for pipeline model.
  -mctor-dtor               Enable constructor/destructor feature.
  -mex9                     Use special directives to guide linker doing ex9
                            optimization.
  -mext-dsp                 Generate DSP extension instructions.
  -mext-fpu-dp              Generate double-precision floating-point instructions.
  -mext-fpu-fma             Generate floating-point multiply-accumulation
                            instructions.
  -mext-fpu-sp              Generate single-precision floating-point instructions.
  -mext-perf                Generate performance extension instructions.
  -mext-perf2               Generate performance extension version 2 instructions.
  -mext-string              Generate string extension instructions.
  -mext-zol                 Insert the hardware loop directive.
  -mfloat-abi=              Specify if floating point hardware should be used.  The
                            valid value is : soft, hard.
  -mfp-as-gp                Force performing fp-as-gp optimization.
  -mfull-regs               Use full-set registers for register allocation.
  -mhw-abs                  Generate hardware abs instructions
  -mifc                     Use special directives to guide linker doing ifc
                            optimization.
  -minnermost-loop          Insert the innermost loop directive.
  -misr-vector-size=        Specify the size of each interrupt vector, which must be
                            4 or 16.
  -mload-store-opt          Enable load store optimization.
  -mmemory-model=           Specify the memory model, fast or slow memory.
  -mno-fp-as-gp             Forbid performing fp-as-gp optimization.
  -mprint-stall-cycles      Print stall cycles due to structural or data dependencies.
                            It should be used with the option '-S'.  Note that stall
                            cycles are determined by the compiler's pipeline model
                            and it may not be precise.
```

```
       -mreduced-regs            Use reduced-set registers for register allocation.
       -mregrename               Enable target dependent register rename
                                 optimization.
       -mrelax                   Guide linker to relax instructions.
       -mrelax-hint              Insert relax hint for linker to do relaxation.
       -msoft-fp-arith-comm      Enable operand commutative for soft floating
       -munaligned-access        Enable unaligned word and halfword accesses to packed
                                 data.
       -mv3push                  Generate v3 push25/pop25 instructions.
       -mvh                      Enable Virtual Hosting support.

       Known floating-point ABIs (for use with the -mfloat-abi= option):
         hard soft

       Known floating-point number of registers (for use with the -mconfig-fpu= option):
         0 1 2 3 4 5 6 7

       Known arch types (for use with the -march= option):
         v2 v2j v3 v3f v3j v3m v3m+ v3s

       Known cmodel types (for use with the -mcmodel= option):
         large medium small

       Known cpu types (for use with the -mcpu= option):
           d10 d1088 d1088-fpu d1088-spu d15 d15f d15s e8 e801 e830 n10 n1033 n1033-fpu
           n1033-spu n1033a n1068 n1068-fpu n1068-spu n1068a n1068a-fpu n1068a-spu n12
           n1213 n1233 n1233-fpu n1233-spu n13 n1337 n1337-fpu n1337-spu n15 n15f n15s
           n6 n650 n7 n705 n8 n801 n820 n9 n903 n903a n968 n968a s8 s801 s830 sn8 sn801
```

NOTE: If you specify the options "`-mcmodel`", "`-mvh`", or "`-mext-zol`" for compilation, use GCC or G++ to link programs and apply these options for linking as well.

2.2.2. Assembler Options

To get a list of all supported options, use command:

```
mypc> nds32le-elf-as --help
```

```
      Usage: nds32le-elf-as [option...] [asmfile...]
      Options:
       -a[sub-option...]         Turn on listings
                                 Sub-options [default hls]:
                                  c       Omit false conditionals.
                                  d       Omit debugging directives.
                                  h       Include high-level source.
                                 l       Include assembly.
                                 m       Include macro expansions.
                                 n       Omit forms processing.
                                 s       Include symbols.
                                 =FILE   List to FILE (must be last sub-option).
       --alternate              Initially turn on alternate macro syntax.
       -D                       Produce assembler debugging messages.
       --defsym SYM=VAL         Define symbol SYM to given value.
       --execstack              Require executable stack for this object.
       --noexecstack            Don't require executable stack for this object.
       -f                       Skip whitespace and comment preprocessing.
       -g --gen-debug           Generate debugging information.
       --gstabs                 Generate STABS debugging information.
       --gstabs+                Generate STABS debug info with GNU extensions.
       --gdwarf-2               Generate DWARF2 debugging information.
       --help                   Show this message and exit.
       --target-help            Show target specific options.
       -I DIR                   Add DIR to search list for .include directives.
       -J                       Don't warn about signed overflow.
       -K                       Warn when differences altered for long displacements.
       -L,--keep-locals         Keep local symbols (e.g. starting with `L').
       -M,--mri                 Assemble in MRI compatibility mode.
       -maie-conf <*.aie>       Set Andes Copilot supported mata file
       --MD FILE                Write dependency information in FILE (default none).
       -nocpp                   Ignored.
       -o OBJFILE               Name the object-file output OBJFILE (default a.out).
       -R                       Fold data section into text section.
       --statistics             Print various measured statistics from execution.
       --strip-local-absolute   Strip local absolute symbols.
       --traditional-format     Use same format as native assembler when possible.
       --version                Print assembler version number and exit.
       -W  --no-warn            Suppress warnings.
       --warn                   Don't suppress warnings.
       --fatal-warnings         Treat warnings as errors.
       --itbl INSTTBL           Extend instruction set to include instructions
                                matching the specifications defined in file INSTTBL.
       -w                       Ignored.
       -X                       Ignored.
       -Z                       Generate object file even after errors.
       --listing-lhs-width      Set the width in words of the output data column of
                                the listing.
       --listing-lhs-width2     Set the width in words of the continuation lines of
                                the output data column; ignored if smaller than the
                                width of the first line.
       --listing-rhs-width      Set the max width in characters of the lines from the
                                source file.
       --listing-cont-lines     Set the maximum number of continuation lines used for
                                the output data column of the listing.
```

```
NDS32 specific command line options:
 -mel, -EL or -little     Produce little endian data.
 -meb, -EB or -big        Produce big endian data.
 -O1                      Optimize for performance.
 -Os                      Optimize for space.
 -cpu or -mcpu=<cpuname>  CPU is <cpuname>.
 -mno-fp-as-gp-relax      Suppress fp-as-gp relaxation for this file
 -mb2bb-relax             Back-to-back branch optimization
 -mno-all-relax           Suppress all relaxation for this file
 -mace=<shrlibfile>       Support user defined instruction extension
 -mcop0=<shrlibfile>      Support coprocessor 0 extension
 -mcop1=<shrlibfile>      Support coprocessor 1 extension
 -mcop2=<shrlibfile>      Support coprocessor 2 extension
 -mcop3=<shrlibfile>      Support coprocessor 3 extension
 -march=<arch name>       Assemble for architecture <arch name> which could be
                          v3, v3j, v3m, v3m+, v3f, v3s, v2, v2j, v2f, v2s
 -mbaseline=<baseline>    Assemble for baseline <baseline> which could be v2,
                          v3, v3m
 -mfpu-freg=<freg>        Specify a FPU configuration
                          <freg>
                          0:    8 SP /  4 DP registers
                          1:   16 SP /  8 DP registers
                          2:   32 SP / 16 DP registers
                          3:   32 SP / 32 DP registers
 -mabi=<abi>              Specify a abi version <abi> could be v1, v2, v2fp,
                          v2fpp
 -m[no-]mac               Enable/Disable Multiply instructions support
 -m[no-]div               Enable/Disable Divide instructions support
 -m[no-]16bit-ext         Enable/Disable 16-bit extension
 -m[no-]dx-regs           Enable/Disable d0/d1 registers
 -m[no-]perf-ext          Enable/Disable Performance extension
 -m[no-]perf2-ext         Enable/Disable Performance extension 2
 -m[no-]string-ext        Enable/Disable String extension
 -m[no-]reduced-regs      Enable/Disable Reduced Register configuration
                          (GPR16) option
 -m[no-]audio-isa-ext     Enable/Disable AUDIO ISA extension
 -m[no-]fpu-sp-ext        Enable/Disable FPU SP extension
 -m[no-]fpu-dp-ext        Enable/Disable FPU DP extension
 -m[no-]fpu-fma           Enable/Disable FPU fused-multiply-add instructions
 -m[no-]dsp-ext           Enable/Disable DSP extension
 -m[no-]zol-ext           Enable/Disable hardware loop extension
 -mall-ext                Turn on all extensions and instructions support
```

2.2.3. Linker Options

To get a list of all supported options, use command:

```
mypc> nds32le-elf-ld --help
```

```
Usage: nds32le-elf-ld [option] file…
Options:
 -a KEYWORD                    Shared library control for HP/UX
                               compatibility
 -A ARCH, --architecture ARCH  Set architecture
 -b TARGET, --format TARGET    Specify target for following input files
 -c FILE, --mri-script FILE    Read MRI format linker script
 --build-id[=STYLE]            Generate build ID note
 -d, -dc, -dp                  Force common symbols to be defined
 -e ADDRESS, --entry ADDRESS   Export all dynamic symbols
 --no-export-dynamic           Undo the effect of --export-dynamic
 -EB                           Link big-endian objects
 -EL                           Link little-endian objects
 -f SHLIB, --auxiliary SHLIB   Auxiliary filter for shared object symbol
                               table
 -F SHLIB, --filter SHLIB      Filter for shared object symbol table
 -g                            Ignored
 -G SIZE, --gpsize SIZE        Small data size (if no size, same as --shared)
 -h FILENAME, -soname FILENAME Set internal name of shared library
 -I PROGRAM, --dynamic-linker  Set PROGRAM as the dynamic linker to use
 PROGRAM
 -l LIBNAME, --library LIBNAME Search for library LIBNAME
 -L DIRECTORY, --library-path  Add DIRECTORY to library search path
 DIRECTORY
 --sysroot=<DIRECTORY>         Override the default sysroot location
 -m EMULATION                  Set emulation
 -M, --print-map               Print map file on standard output
 -n, --nmagic                  Do not page align data
 -N, --omagic                  Do not page align data, do not make text
                               readonly
 --no-omagic                   Page align data, make text readonly
 -o FILE, --output FILE        Set output file name
 -O                            Optimize output file
 -plugin PLUGIN                Load named plugin
 -plugin-opt ARG               Send arg to last-loaded plugin
 -flto                         Ignored for GCC LTO option compatibility
 -flto-partition=              Ignored for GCC LTO option compatibility
 -fuse-ld=                     Ignored for GCC linker option compatibility
 -Qy                           Ignored for SVR4 compatibility
 -q, --emit-relocs             Generate relocations in final output
 -r, -i, --relocatable         Generate relocatable output
```

```
       -R FILE, --just-symbols FILE    Just link symbols (if directory, same as
                                       --rpath)
       -s, --strip-all                 Strip all symbols
       -S, --strip-debug               Strip debugging symbols
       --strip-discarded               Strip symbols in discarded sections
       --no-strip-discarded            Do not strip symbols in discarded sections
       -t, --trace                     Trace file opens
       -T FILE, --script FILE          Read linker script
       --default-script FILE, -dT      Read default linker script
       -u SYMBOL, --undefined SYMBOL   Start with undefined reference to SYMBOL
       --unique [=SECTION]             Don't merge input [SECTION | orphan] sections
       -Ur                             Build global constructor/destructor tables
       -v, --version                   Print version information
       -V                              Print version and emulation information
       -x, --discard-all               Discard all local symbols
       -X, --discard-locals            Discard temporary local symbols (default)
       --discard-none                  Don't discard any local symbols
       -y SYMBOL, --trace-symbol       Trace mentions of SYMBOL
       SYMBOL
       -Y PATH                         Default search path for Solaris compatibility
       -(, --start-group               Start a group
       -), --end-group                 End a group
       --accept-unknown-input-arch     Accept input files whose architecture cannot
                                       be determined
       --no-accept-unknown-input-arch  Reject input files whose architecture is
                                       unknown
       --as-needed                     Only set DT_NEEDED for following dynamic libs
                                       if used
       --no-as-needed                  Always set DT_NEEDED for dynamic libraries
                                       mentioned on the command line
       -assert KEYWORD                 Ignored for SunOS compatibility
       -Bdynamic, -dy, -call_shared    Link against shared libraries
       -Bstatic, -dn, -non_shared,     Do not link against shared libraries
       -static
       -Bsymbolic                      Bind global references locally
       -Bsymbolic-functions            Bind global function references locally
       --check-sections                Check section addresses for overlaps
                                       (default)
       --no-check-sections             Do not check section addresses for overlaps
       --copy-dt-needed-entries        Copy DT_NEEDED links mentioned inside DSOs
                                       that follow
       --no-copy-dt-needed-entries     Do not copy DT_NEEDED links mentioned inside
                                       DSOs that follow
       --cref                          Output cross reference table
       --defsym SYMBOL=EXPRESSION      Define a symbol
       --demangle [=STYLE]             Demangle symbol names [using STYLE]
       --embedded-relocs               Generate embedded relocs
       --fatal-warnings                Treat warnings as errors
```

```
      --no-fatal-warnings            Do not treat warnings as errors (default)
      -fini SYMBOL                   Call SYMBOL at unload-time
      --force-exe-suffix             Force generation of file with .exe suffix
      --gc-sections                  Remove unused sections (on some targets)
      --no-gc-sections               Don't remove unused sections (default)
      --print-gc-sections            List removed unused sections on stderr
      --no-print-gc-sections         Do not list removed unused sections
      --hash-size=<NUMBER>           Set default hash table size close to <NUMBER>
      --help                         Print option help
      -init SYMBOL                   Call SYMBOL at load-time
      -Map FILE                      Write a map file
      --no-define-common             Do not define Common storage
      --no-demangle                  Do not demangle symbol names
      --no-keep-memory               Use less memory and more disk I/O
      --no-undefined                 Do not allow unresolved references in object
                                     files
      --allow-shlib-undefined        Allow unresolved references in shared
                                     libraries
      --no-allow-shlib-undefined     Do not allow unresolved references in shared
                                     libs
      --allow-multiple-definition    Allow multiple definitions
      --no-undefined-version         Disallow undefined version
      --default-symver               Create default symbol version
      --default-imported-symver      Create default symbol version for imported
                                     symbols
      --no-warn-mismatch             Don't warn about mismatched input files
      --no-warn-search-mismatch      Don't warn on finding an incompatible library
      --no-whole-archive             Turn off --whole-archive
      --noinhibit-exec               Create an output file even if errors occur
      -nostdlib                      Only use library directories specified on the
                                     command line
      --oformat TARGET               Specify target of output file
      --print-output-format          Print default output format
      -qmagic                        Ignored for Linux compatibility
      --reduce-memory-overheads      Reduce memory overheads, possibly taking much
                                     longer
      --relax                        Reduce code size by using target specific
                                     optimizations
      --no-relax                     Do not use relaxation techniques to reduce code
                                     size
      --retain-symbols-file FILE     Keep only symbols listed in FILE
      -rpath PATH                    Set runtime shared library search path
      -rpath-link PATH               Set link time shared library search path
      -shared, -Bshareable           Create a shared library
      -pie, --pic-executable         Create a position independent executable
      --sort-common                  Sort common symbols by alignment [in specified
      [=ascending|descending]        order]
      --sort-section name|alignment  Sort sections by name or maximum alignment
```

```
  --spare-dynamic-tags COUNT      How many tags to reserve in .dynamic section
  --split-by-file [=SIZE]         Split output sections every SIZE octets
  --split-by-reloc [=COUNT]       Split output sections every COUNT relocs
  --stats                         Print memory usage statistics
  --target-help                   Display target specific options
  --task-link SYMBOL              Do task level linking
  --traditional-format            Use same format as native linker
  --section-start                 Set address of named section
  SECTION=ADDRESS
  -Tbss ADDRESS                   Set address of .bss section
  -Tdata ADDRESS                  Set address of .data section
  -Ttext ADDRESS                  Set address of .text section
  -Ttext-segment ADDRESS          Set address of text segment
  -Trodata-segment ADDRESS        Set address of rodata segment
  -Tldata-segment ADDRESS         Set address of ldata segment
  --unresolved-symbols=<method>   How to handle unresolved symbols.  <method>
                                  is: ignore-all, report-all,
                                  ignore-in-object-files,
                                  ignore-in-shared-libs
  --verbose [=NUMBER]             Output lots of information during link
  --version-script FILE           Read version information script
  --version-exports-section       Take export symbols list from .exports, using
  SYMBOL                          SYMBOL as the version.
  --dynamic-list-data             Add data symbols to dynamic list
  --dynamic-list-cpp-new          Use C++ operator new/delete dynamic list
  --dynamic-list-cpp-typeinfo     Use C++ typeinfo dynamic list
  --dynamic-list FILE             Read dynamic list
  --warn-common                   Warn about duplicate common symbols
  --warn-constructors             Warn if global constructors/destructors are
                                  seen
  --warn-multiple-gp              Warn if the multiple GP values are used
  --warn-once                     Warn only once per undefined symbol
  --warn-section-align            Warn if start of section changes due to
                                  alignment
  --warn-shared-textrel           Warn if shared object has DT_TEXTREL
  --warn-alternate-em             Warn if an object has alternate ELF machine
                                  code
  --warn-unresolved-symbols       Report unresolved symbols as warnings
  --error-unresolved-symbols      Report unresolved symbols as errors
  --whole-archive                 Include all objects from following archives
  --wrap SYMBOL                   Use wrapper functions for SYMBOL
  --ignore-unresolved-symbol      Unresolved SYMBOL will not cause an error or
  SYMBOL                          warning

 NDS32 specific command line options:
  -z common-page-size=SIZE        Set common page size to SIZE
  -z defs                         Report unresolved symbols in object files.
  -z execstack                    Mark executable as requiring executable stack
```

```
      -z max-page-size=SIZE          Set maximum page size to SIZE
      -z muldefs                     Allow multiple definitions
      -z noexecstack                 Mark executable as not requiring executable
                                     stack
      --m[no-]fp-as-gp               Disable/enable fp-as-gp relaxation
      --mexport-symbols=FILE         Exporting symbols in linker script

       V3 only command line options:
      --m[no-]ex9                    Disable/enable link-time EX9 relaxation
      --mexport-ex9=FILE             Export EX9 table after linking
      --mimport-ex9=FILE             Import Ex9 table for EX9 relaxation
      --mupdate-ex9                  Update existing EX9 table
      --mex9-limit=NUM               Maximum number of entries in ex9 table
      --mex9-loop-aware              Avoid generate EX9 instruction inside loop
      --m[no-]ifc                    Disable/enable link-time IFC optimization
      --mifc-loop-aware              Avoid generate IFC instruction inside loop
```

Please pay attention to the following two NDS32-specific commands:

`--mfp-as-gp`	It's for data affinity optimization. Set `$fp` as `$gp` plus an offset to use more code density instructions such as `lwi37.fp` and `swi37.fp`.
`--mexport-symbols`	This option functions the same as the deprecated option `--mgen-symbol-ld-script`. It generates a linker script format file which saves all symbols for ROM patch to use for linking.

Linker options specialized for V3 targets are involved with either EX9 or IFC optimization. Please refer to Section 19.2 or 19.3 for detailed descriptions.

3. NDS32 Assembly Language

This chapter is intended to provide an outline and some hints for NDS assembly language. For more details about assembly programming, please consult *AndeStar Instruction Set Architecture Manual*, demo code in the package and *Using as* (GNU Assembly Manual).

3.1. General Syntax

Use "`#`" at column 1 and "`!`" 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`, `#128`, `0200`, `#0200`, `0x80`, `#0x80`, `0b10000000`, and `#0b10000000` are all identical. The leading "`#`" is optional.

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 label. For example, "`jral F1`" is different from "`jral f1`" while it is the same as "`JRAL F1`".

3.2. Registers

Please refer to *AndeStar Instruction Set Architecture Manual* in the package for detailed information.

3.2.1. General Purpose Registers (GPR)

There are 32 32-bit general purpose registers:

1. All `$r0-$r31` are 5-bit addressable.
2. 4-bit addressable ones are `$h0-$h15`, which are mapped to `$r0-$r11` and `$r16-$r19` correspondingly.
3. 3-bit addressable ones are `$o0-$o7`, which are mapped to `$r0-$r7` correspondingly.

3.2.2. Accumulators d0 and d1

There are 2 64-bit accumulators:

1. High and low portion of `$d0` and `$d1` can be accessed separately as `$d0.hi`, `$d0.lo`, `$d1.hi`, and `$d1.lo`.
2. There are instructions for moving them from and to GPRs.

NOTE: Though `$d0` or `$d1` instruction still work for assembly programming, compiler of BSP v4.0 or later versions has no longer generated them.

3.2.3. Instruction Implied Registers

Some 16-bit instructions use implied registers:

1. Register `$r5`: `BEQS38` and `BNES38.`
2. Register `$ta` (`$r15`) : `SLTI45`, `SLTSI45`, `SLT45`, `SLTS45`, `BEQZS8`, and `BNEZS8`.
3. Register `$fp` (`$r28`): `LWI37` and `SWI37`.
4. Register `$gp` (`$r29`): `LBI.GP, LHI.GP, LWI.GP, SBI.GP, SHI.GP,` and `SWI.GP`.
5. Register `$sp` (`$r31`): `LWI37.SP` and `SWI37.SP`.

3.2.4. Assembler Reserved Register $ta

Register `$ta` (`$r15`) is used

1. by assembler to translate pseudo instructions. Thus, its content may get corrupted.
2. to pass the starting address of called function at entry to the called function if PIC mode is specified. Thus, its content must be properly handled.
3. as implied register. Thus, its content must be preserved between `SLT{S}{I}45` and `B[EQ|NE]QZS8` instruction pairs.

3.2.5. Operating System Reserved Registers $p0 and $p1

Registers `$p0` and `$p1` are used by operating system as scratch registers. Since interrupt can occur at any user space instruction, its content may not be persistent from instruction to instruction.

`$p0` and `$p1` are not recommended for use in user code. Here are some reminders if you want to use the two registers in your code:

1. To avoid the corruption of `$p0` and `$p1`, lower the interrupt level to 0 if you want to do context switching in the interrupt.
2. You may use shadow `$sp,` rather than `$p0` or `$p1`, as scratch registers when switching between user-mode and superuser-mode.

3.3. Missing Operand

In most cases, assembler accepts instructions with missing operands. When this occurs, the default value of the missing operand is used.

3.3.1. Load/Store Instructions

Coded As	Meaning Accepted As
`l{b|h|w}i rt5,[ra5]`	`l{b|h|w}i rt5,[ra5+0]`
`l{b|h}si rt5,[ra5]`	`l{b|h}si rt5,[ra5+0]`
`l{b|h|w}i.p rt5,[ra5]`	`<invalid>`
`l{b|h}si.p rt5,[ra5]`	`<invalid>`
`s{b|h|w}i rt5,[ra5]`	`s{b|h|w}i rt5,[ra5+0]`
`s{b|h|w}i.p rt5,[ra5]`	`<invalid>`
`l{b|h|w} rt5,[ra5+rb5]`	`l{b|h|w} rt5,[ra5+rb5<<0]`
`l{b|h}s rt5,[ra5+rb5]`	`l{b|h}s rt5,[ra5+ rb5<<0]`
`s{b|h|w} rt5,[ra5+rb5]`	`s{b|h|w} rt5,[ra5+ rb5<<0]`
`l{b|h|w} rt5,[ra5]`	`l{b|h|w}i rt5,[ra5+0]`
`l{b|h}s rt5,[ra5]`	`l{b|h}si rt5,[ra5+0]`
`l{b|h|w}.p rt5,[ra5]`	`<invalid>`
`l{b|h}s.p rt5,[ra5]`	`<invalid>`
`s{b|h|w} rt5,[ra5]`	`s{b|h|w}i rt5,[ra5+0]`
`s{b|h|w}.p rt5,[ra5]`	`<invalid>`
`l{b|h|w}.p rt5,[ra5],rb5`	`l{b|h|w}.p rt5,[ra5],rb5<<0`
`l{b|h}s.p rt5,[ra5],rb5`	`l{b|h}s.p rt5,[ra5],rb5<<0`
`s{b|h|w}.p rt5,[ra5],rb5`	`s{b|h|w}.p rt5,[ra5],rb5<<0`
`lmw.{a|b}{d|i}{m} rt5,[ra5],rb5`	`lmw.{a|b}{d|i}{m} rt5,[ra5],rb5,0`
`smw.{a|b}{d|i}{m} rt5,[ra5],rb5`	`smw.{a|b}{d|i}{m} rt5,[ra5],rb5,0`

Coded As	Meaning Accepted As
`lwup rt5,[ra5+rb5]`	`lwup rt5,[ra5+ rb5<<0]`
`lwup rt5,[ra5]`	`<invalid>`
`swup rt5,[ra5+rb5]`	`swup rt5,[ra5+ rb5<<0]`
`swup rt5,[ra5]`	`<invalid>`
`l{w|h|b}i333 rt3,[ra3]`	`l{w|h|b}i333 rt3,[ra3+0]`
`s{w|h|b}i333 rt3,[ra3]`	`s{w|h|b}i333 rt3,[ra3+0]`
`lwi37 rt3,[$fp]`	`lwi37 rt3,[$fp+0]`
`swi37 rt3,[$fp]`	`swi37 rt3,[$fp+0]`

3.3.2. Branch Instructions

Coded As	Meaning Accepted As
`jral rb5`	`jral $lp,rb5`
`ret`	`ret $lp`
`ret5`	`ret5 $lp`

3.3.3. Special Instructions

Coded As	Meaning Accepted As
`llw rt5,[ra5+rb5]`	`llw rt5,[ra5+ rb5<<0]`
`llw rt5,[ra5]`	`<invalid>`
`scw rt5,[ra5+rb5]`	`scw rt5,[ra5+ rb5<<0]`
`scw rt5,[ra5]`	`<invalid>`
`dprefi.d dprefst,[ra5]`	`dprefi.d dprefst,[ra5+0]`
`dprefi.w dprefst,[ra5]`	`dprefi.w dprefst,[ra5+0]`
`dpref dprefst,[ra5+rb5]`	`dpref dprefst,[ra5+rb5<<0]`
`dpref dprefst,[ra5]`	`Dprefi.w dprefst,[ra5+0]`
`msync`	`msync 0`
`trap`	`trap 0`

Coded As	Meaning Accepted As
`teqz ra5`	`teqz ra5, 0`
`tnez ra5`	`tnez ra5, 0`
`break`	`break 0`

4. Machine Instructions

4.1. 32/16-bit

Full machine instructions, 32-bit and/or 16-bit, can be specified by programmers directly. They can be mixed with any restriction. 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 compiler optimization level:

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.

4.2. Unaligned Data Handling

`[l|s]mw` instructions can be used to handle unaligned data accesses. The following focuses on using `[l|s]mw` instructions for block moves like `memcpy()`.

A loop of `lmw.bim rb5,[ra5],rb5` and `smw.bim rb5,[ra5],rb5` takes care of most content except the remaining bytes which cannot be handled with a word. Compiler must handle the "packed" structure this way since the only other way is to do it byte by byte. Here "packed" means that member fields of the structure may not be aligned. In contrast, fields of a default (non-packed) structure are aligned based on their types (namely, word field is aligned on word boundary; half word field is aligned on half word boundary and so forth).

4.3. Endianness

Andes supports both big and little endian data storage although instructions only use big endian. Here are two different ways to support it:

1. static setting only – OS finds the setting when loading ELF image and properly sets the configuration in system register.
2. dynamic setting – instruction `SETEND.B` can be used to switch user space programs to big endian mode and `SETEND.L` to switch the programs to little endian mode. Once switched to different data endianness, all data access will be interpreted based on the new endianness.

5. Pseudo-ops

5.1. List of Pseudo-ops

5.1.1. GNU Default Pseudo-ops Supporting Sections

`.data subsec`	for data section. Default of subsec is `0`, which is created automatically.
`.text subsec`	for code section. Default of subsec is `0`, which is created automatically.
`.section`	for user defined sections.

5.1.2. Andes Pseudo-ops Supporting Sections

`.sdata_d`	for double-word sized (8-byte) small data items.
`.sdata_w`	for word sized (4-byte) small data items.
`.sdata_h`	for half-word sized (2-byte) small data items.
`.sdata_b`	for byte sized small data items.
`.sbss_d`	for double-word sized (8-byte) small data items.
`.sbss_w`	for word sized (4-byte) small data items.
`.sbss_h`	for half-word sized (2-byte) small data items.
`.sbss_b`	for byte sized small data items.

5.1.3. GNU Default Pseudo-ops Supporting ELF

`.align type,fill,max`	for alignment. `type` defines power-of-2 alignment. for example, `type=2` gives alignment to word (4-byte) aligned boundary. If `fill` is not specified, `0` will be filled for data sections and `nop` or `nop16` will be filled for code sections.
`.ascii`	for string constant.
`.asciz`	for zero-terminated string constant.
`.byte`	for byte data.
`.2byte`	for 2-byte data. (alignment is not forced)
`.4byte`	for 4-byte data. (alignment is not forced)
`.8byte`	for 8-byte data. (alignment is not forced)
`.double`	for double precision floating data.
`.eject`	for page break in listings.
`.else`	for conditional assembly.
`.elseif`	for conditional assembly.
`.end`	for terminating assembly.
`.endm`	for terminating macro expansion.
`.endr`	for terminating iterative assembly.

`.endfunc`	for terminating a function.
`.endif`	for conditional assembly.
`.equ symbol,expr`	for defining symbol to value expr.
`.equiv symbol,expr`	same as `.equ` except duplicate is an error.
`.err`	for signaling assembling error.
`.error string`	for signaling assembling error.
`.exitm`	for exiting macro expansion.
`.extern symbol`	ignored - only for programming discipline.
`.fail expr`	for generating error (expr<500) or warning.
`.file string`	for starting new logical file.
`.fill rept,size,value`	for filling data chunk.
`.float expr`	for single precision floating data.
`.func symbol,label`	for issuing debugging information.
`.global symbol`	for exporting symbol.
`.globl symbol`	same as .global.
`.hidden names`	for changing visibility of names.
`.hword expr`	for half-word sized data.
`.ident`	for tagging.
`.if expr`	for conditional assembly.
`.ifdef symbol`	for conditional assembly.

`.ifc str1,str2`	for conditional assembly.
`.ifeq expr`	for conditional assembly.
`.ifeqs str1,str2`	for conditional assembly.
`.ifge expr`	for conditional assembly.
`.ifgt expr`	for conditional assembly.
`.ifle expr`	for conditional assembly.
`.iflt expr`	for conditional assembly.
`.ifnc str1,str2`	for conditional assembly.
`.ifndef symbol`	for conditional assembly.
`.ifnotdef symbol`	same as `ifndef`.
`.ifne expr`	for conditional assembly.
`.ifnes str1,str2`	for conditional assembly.
`.incbin file,skip,count`	for including binary file.
`.include file`	for including source file.
`.int expr`	for integer sized data.
`.internal names`	for changing visibility of names.
`.irp symbol,values`	for starting iterative assembly.
`.list`	for generating listings.
`.long expr`	for integer sized data.
`.macro name,params`	for defining macros.

`.nolist`	for stopping generating listings.
`.octa expr`	for 16-byte sized data.
`.org expr,fill`	for moving location counter forward.
`.previous`	for swapping ELF sections.
`.popsection`	for popping ELF sections.
`.print string`	for printing string in listings.
`.protected names`	for changing visibility of names.
`.psize line,col`	for defining page size of listings.
`.purgem name`	for purging the macro definition of name.
`.pushsection name,subsec`	for pushing the current section (and subsection) onto the top of the section stack and replacing them with name and subsection.
`.quad expr`	for 8-byte sized data.
`.rept count`	for starting iterative assembly.
`.sbttl string`	for printing subtitle line in listings.
`.set symbol,expr`	for defining symbol to value `expr`.
`.short expr`	for word sized data.
`.single expr`	for single precision floating data.
`.size symbol,expr`	for specifying size of a symbol.
`.sleb128 expr`	for `SLEB128` data.
`.skip size,fill`	for size-byte data chunk.

`.space size,fill`	same as `.skip`.
`.string string`	same as `.asciz`
`.struct expr`	for switching to absolute section.
`.subsection subsec`	for swapping current subsection to subsec.
`.title string`	for printing title line in listings.
`.type name,desc`	for defining type of the symbol.
`.uleb128 expr`	for `ULEB128` data.
`.version string`	for creating `.note` section content.
`.vtable_entry table,offset`	for finding/creating a symbol table and creating a `VTABLE_ENTRY` relocation with an addend of offset.
`.vtable_inherit child,parents`	for finding the symbol child and finding/creating the symbol parent and then creating a `VTABLE_INHERIT` relocation for the parent whose addend is the value of the child symbol.
`.warning string`	for printing warning in listings.
`.word expr`	for word sized data.

Please note that `.hword`, `.half`, and `.short` are referring to 16-bit data; `.int`, `.long`, and `.word` are referring to 32-bit data; `.quad` is for 64-bit data; and `.octa` is for 128-bit data.

5.1.4. Andes Pseudo-ops Supporting ELF

`.half`	for half-word sized (2-byte) data.
`.word`	for word sized (4-byte) data.
`.dword`	for double-word sized (8-byte) data.
`.qword`	for quadruple-word sized (16-byte) data.
`.off_16bit`	to start generating only 32-bit instructions.
`.restore_16bit`	to restore a setting of starting/stopping generating only 32-bit instructions.
`.pic`	for generating PIC code. This must appear before the first assembly instruction. (first assembly line preferred)
`.debugsym`	for debugging symbols.
`.little`	for setting little endian data storage.
`.big`	for setting big endian data storage.

5.1.5. Data Declaration Pseudo-ops

`.half and .hword:`	forced 2-byte alignment.
`.int, .float, .long, .single, and .word:`	forced 4-byte alignment.
`.double and .dword:`	forced 8-byte alignment.
`.qword:`	forced 16-byte alignment.

If you do not want forced alignments, use

`.dc.h or .2byte`	for .half and .hword.
`.dc, .dc.l, and .dc.w or .4byte`	for .int, .long, and .word.
`.dc.s or .4byte`	for .float and .single.
`.dc.d or .8byte`	for .double.
`.dc.x`	for extended (12-byte) floating number.

5.1.6. Space Declaration Pseudo-ops

`.dcb`, `.dcb.d`, `.dcb.h`, `dcb.l`, `dcb.s`, `dcb.w`, and `.dcb.x`.

`.ds`, `.ds.d`, `.ds.h`, `.ds.l`, `ds.s`, and `.ds.w`.

`.space.`

`.skip.`

`.zero.`

`.fill` – will fill the data area with specified fill value.

6. Pseudo-instructions

In addition to hardware instructions, there are many software instructions defined to make assembly programming much easier. These are pseudo-instructions. This chapter makes a detailed list of pseudo-instructions along with descriptions.

NOTICE:

While some pseudo-instructions are reserved for internal processing only, some dimmed in this chapter are deprecated and not recommended. For a summary of deprecated pseudo-instructions and reasons for deprecation, please refer to Table 3.

6.1. List of Pseudo-instructions

1. load 32-bit value/address

`li rt5,imm_32`	loads 32-bit integer into register `rt5`. `sethi rt5,hi20(imm_32)` and then `ori rt5,rt5,lo12(imm_32)`
`la rt5,var`	loads 32-bit address of var into register `rt5`. `sethi rt5,hi20(var)` and then `ori rt5,rt5,lo12(var)`

2. load/store variables

`l.{bhw} rt5,var`	loads value of var into register `rt5`. `sethi $ta,hi20(var)` and then `l{bhw}i rt5,[$ta+lo12(var)]`
`l.{bh}s rt5,var`	loads value of var into register `rt5`. `sethi $ta,hi20(var)` and then `l{bh}si rt5,[$ta+lo12(var)]`
`l.{bhw}p rt5,var,inc`	loads value of var into register `rt5` and increments `$ta` by amount inc. `la $ta,var` and then `l{bhw}i.bi rt5,[$ta],inc`

`l.{bhw}pc rt5,inc`	continues loading value of var into register `rt5` and increments `$ta` by amount inc. `l{bhw}i.bi rt5,[$ta],inc.`
`l.{bh}sp rt5,var,inc`	loads value of var into register `rt5` and increments `$ta` by amount inc. `la $ta,var` and then `l{bh}si.bi rt5,[$ta],inc`
`l.{bh}spc rt5,inc`	continues loading value of var into register `rt5` and increments `$ta` by amount inc. `l{bh}si.bi rt5,[$ta],inc`.
`s.{bhw} rt5,var`	stores register `rt5` to var. `sethi $ta,hi20(var)` and then `s{bhw}i rt5,[$ta+lo12(var)]`
`s.{bhw}p rt5,var,inc`	stores register `rt5` to var and increments `$ta` by amount inc. `la $ta,var` and then `s{bhw}i.bi rt5,[$ta],inc`
`s.{bhw}pc rt5,inc`	continues storing register `rt5` to var and increments `$ta` by amount inc. `s{bhw}i.bi rt5,[$ta],inc.`

For 64-bit extension, the `{ls}.ws{p}` and `{ls}.d{p}` are defined similarly.

3. negation

`not rt5,ra5`	alias of `nor rt5,ra5,ra5`
`neg rt5,ra5`	alias of `subri rt5,ra5,0`

4. branch to label

`br rb5`	alias of `jr rb5` depending on how it is assembled. It is translated into "`jr5 rb5`" or "`jr rb5`"
`b label`	branch to label.

	depending on how it is assembled. It is translated into "`j8 label`", "`j label`", or "`la $ta,label; br $ta`"
`bge{s} rt5,ra5,label`	compares the unsigned (signed) value of `rt5` and that of `ra5`. If the value of `rt5` is greater than or equal to that of `ra5`, jump to label.
`bgt{s} rt5,ra5,label`	compares the unsigned (signed) value of `rt5` and that of `ra5`. If the value of `rt5` is greater than that of `ra5`, jump to label.
`blt{s} rt5,ra5,label`	compares the unsigned (signed) value of `rt5` and that of `ra5`. If the value of `rt5` is less than that of `ra5`, jump to label.
`ble{s} rt5,ra5,label`	compares the unsigned (signed) value of `rt5` and that of `ra5`. If the value of `rt5` is less than or equal to that of `ra5`, jump to label.
`beq rt5,ra5,label`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.
`beqz rt5,label`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.
`bne rt5,ra5,label`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.
`bnez rt5,label`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.
`bgez rt5,label`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.
`bgtz rt5,label`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.
`bltz rt5,label`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.

`blez rt5,label`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.

Note: Since there are hardware instruction `beq` and `bne` but no `bge{s}`, `bgt{s}`, `ble{s}`, and `blt{s}`, the missing ones are pseudo-code instructions. The implementation will then get wider range. That is, `beq` and `bne` have only 15-bit range but others (`beqz`, `bgez`, `bgtz`, `blez`, `bltz`, and `bnez`) have 17-bit range.

5. branch and link to function name

`bral rb5`	alias of `jral br5` depending on how it is assembled. It is translated into "`jral5 rb5`" or "`jral rb5`".
`bal fname`	depending on how it is assembled. It is translated into "`jal fname`" or "`la $ta,fname; bral $ta`".
`call fname`	call function fname same as "`bal fname`".
`bgezal rt5,fname`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.
`bltzal rt5,fname`	is a hardware instruction. Please refer to *AndeStar Instruction Set Architecture Manual*.

6. move

`move rt5,ra5`	for 16-bit, it is `mov55 rt5,ra5`
	for no 16-bit, it is `ori rt5,ra5,0`

`move rt5,var`	same as `l.w rt5,var`
`move rt5,imm_32`	same as `li rt5,imm_32`

7. push/pop

`pushm ra5,rb5`	pushes the contents of registers from `ra5` to `rb5` into stack.
`push ra5`	pushes the content of register `ra5` into stack. (same as `pushm ra5,ra5`)
`push.d var`	pushes the value of double-word variable var into stack.
`push.w var`	pushes the value of word variable var into stack.
`push.h var`	pushes the value of half-word variable var into stack.
`push.b var`	pushes the value of byte variable var into stack.
`pusha var`	pushes the 32-bit address of variable var into stack.
`pushi imm_32`	pushes the 32-bit immediate value into stack.
`popm ra5,rb5`	poppes top of stack values into registers `ra5` to `rb5`.
`pop rt5`	poppes top of stack value into register. (same as `popm rt5,rt5`)
`pop.d var,ra5`	poppes the value of double-word variable var from stack using the register `ra5` as the second scratch register. (the first scratch register is `$ta`)
`pop.w var,ra5`	poppes the value of word variable var from stack using the register `ra5`.
`pop.h var,ra5`	poppes the value of half-word variable var from stack using the register `ra5`.
`pop.b var,ra5`	poppes the value of byte variable var from stack using the register `ra5`.

6.1.1. Deprecated Pseudo-instructions

The table below lists deprecated pseudo-instructions for quick reference.

Table 3. Deprecated Pseudo-instructions

Category	Deprecated Pseudo-instructions	Reasons for Deprecation

load/store variables	`l.{bhw}p rt5,var,inc`
	`l.{bhw}pc rt5,inc`
	`l.{bh}sp rt5,var,inc`
	`l.{bh}spc rt5,inc`
	`s.{bhw}p rt5,var,inc`
	`s.{bhw}pc rt5,inc`
		These instructions must depend on $r15.

branch to label	`beq rt5,ra5,label`
	`beqz rt5,label`
	`bne rt5,ra5,label`
	`bnez rt5,label`
	`bgez rt5,label`
	`bgtz rt5,label`
	`bltz rt5,label`
	`blez rt5,label`
		These instructions can be replaced by identical hardware instructions.

branch and link to function name	`bgezal rt5,fname`
	`bltzal rt5,fname`
		These instructions can be replaced by identical hardware instructions.

push/pop	`push.d var`
	`push.w var`
	`push.h var`
	`push.b var`
	`pusha var`
	`pushi imm_32`
	`pop.d var, ra5`
	`pop.w var, ra5`
	`pop.h var, ra5`
	`pop.b var, ra5`
		The functionalities to push/pop from/to variables are not supported anymore.

6.2. Built-in Function Operators

The following function operators can be used in any assembly instructions:

1. `hi20(var)` is the high 20-bit of address of var.
2. `lo12(var)` is the low 12-bit of address of var.
3. `sda(var)` is the 15-bit signed offset of var into small data area.

7. Macros

7.1. Create 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
```

Another example provided below shows how a macro is used to simplify varied syntax for different conditions.

```
.macro load_imm rt5, imm32
.if ((\imm32 <= 0x7ffff) && (\imm32 >= -0x80000))
   movi \rt5,\imm32
.elseif (\imm32 & 0x00000fff == 0x0)
   sethi \rt5,hi20(\imm32)
.else
   sethi \rt5,hi20(\imm32)
   ori \rt5,\rt5,lo12(\imm32)
.endif
.endm
```

With such definition, no matter what range the immediate value is, you just need to write the "`load_imm`" macro and it will be expanded as appropriate instructions:

Macro	Assembly Code
`load_imm $r3,0x55`	`movi $r3,0x55`
`load_imm $r3,0x12345000`	`sethi $r3,0x12345`
`load_imm $r3,0x12345999`	`sethi $r3,0x12345 + ori $r3,$r3,0x999`

7.2. 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, please refer to 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 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, in this case "`0`", and "`\p2`" evaluating to *`b`*).

`.endm`

Mark the end of a macro definition.

8. Application Binary Interface (ABI)

The Andes architecture ABI defines the interface for compiled programs and assembled programs running on Andes architecture to work jointly. The purpose of Andes architecture ABI is to deliver high performance and binary compatibility. Section 8.1 describes the used data types in programming and how they are presented on Andes architecture. Section 8.2 gives the details of two types in Andes ABI.

8.1. Data Types

8.1.1. Byte Ordering

The byte ordering defines how the bytes that make up multi-byte data type are ordered in memory. Andes architecture ABI supports both little-endian and big-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.

8.1.2. Primitive Data Types

Table 4. Size and Byte Alignment of Primitive Data Types

Class	**Machine Type**	**Size (in Byte)**	**Alignment (in Byte)**
Integer	Unsigned byte	1	1
	Signed byte	1	1
	Unsigned half word	2	2
	Signed half word	2	2
	Unsigned word	4	4
	Signed word	4	4
	Unsigned double word	8	8
	Signed double word	8	8

Class	Machine Type	Size (in Byte)	Alignment (in Byte)
Floating Point	Single precision (IEEE 754)	4	4
	Double precision (IEEE 754)	8	8
Pointer	Instruction Pointer	4	4
	Data Pointer	4	4

8.1.3. 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.

8.1.3.1 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.

8.1.3.2 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.

8.1.3.3 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.

8.1.4. C Language Mapping of Andes Platform

Table 5. Mapping of C Primitive Data Types

C/C++ Type	**Machine Type**
[singed] char	Signed byte
unsigned char	Unsigned byte
[signed] short	Signed half word
unsigned short	Unsigned half word
[signed] int	Signed word
unsigned int	Unsigned word
[signed] long	Signed word
unsigned long	Unsigned word
[signed] long long	Signed double word
unsigned long long	Unsigned double word
size_t	Unsigned word
float	Single precision (IEEE 754)
double	Double precision (IEEE 754)
long double	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)

8.2. Calling Convention

For code generation efficiency, Andes introduces two ABI types: ABI2 and ABI2FP+. The ABI2 is the convention for integer toolchains, which uses General Purpose Registers (GPRs) for computations on all primitive types. Based on ABI2, ABI2FP+ is provided for floating-point toolchains, in which programmers have extra Floating Point Registers (FPRs) and more instructions to do floating-point computation. Please see the following sections for characteristics of the two ABI types.

8.2.1. ABI2 (for v3, v3j and v3m Toolchains)

8.2.1.1 Registers

There are 32 32-bit General Purpose Registers (GPRs) for Andes instruction set architecture. Basically they are classified into caller-saved and callee-saved registers. The following table lists the Andes GPRs commented with the ABI2 usage convention.

Table 6. Andes GPRs with ABI Usage Convention

Register	Synonym	Comments
`$r0`	`$a0`	Argument / Return / Saved by caller
`$r1`	`$a1`	Argument / Return / Saved by caller
`$r2`	`$a2`	Argument / Saved by caller
`$r3`	`$a3`	Argument / Saved by caller
`$r4`	`$a4`	Argument / Saved by caller
`$r5`	`$a5`	Argument / Saved by caller
`$r6`	`$s0`	Saved by callee
`$r7`	`$s1`	Saved by callee
`$r8`	`$s2`	Saved by callee
`$r9`	`$s3`	Saved by callee
`$r10`	`$s4`	Saved by callee
`$r11`	`$s5`	Saved by callee

Register	Synonym	Comments
`$r12`	`$s6`	Saved by callee
`$r13`	`$s7`	Saved by callee
`$r14`	`$s8`	Saved by callee
`$r15`	`$ta`	Temporary register for assembler
`$r16`	`$t0`	Trampoline register / Saved by caller
`$r17`	`$t1`	Saved by caller
`$r18`	`$t2`	Saved by caller
`$r19`	`$t3`	Saved by caller
`$r20`	`$t4`	Saved by caller
`$r21`	`$t5`	Saved by caller
`$r22`	`$t6`	Saved by caller
`$r23`	`$t7`	Saved by caller
`$r24`	`$t8`	Saved by caller
`$r25`	`$t9`	Saved by caller
`$r26`	`$p0`	Saved by caller
`$r27`	`$p1`	Saved by caller
`$r28`	`$fp`	Frame pointer / Saved by callee
`$r29`	`$gp`	Global pointer / Saved by callee
`$r30`	`$lp`	Link pointer / Saved by callee
`$r31`	`$sp`	Stack pointer

As commented in the table, some registers are also taken for special usage, such as passing argument or being stack frame pointer. They are summarized below and will be described in subsequent sections:

- Argument Passing: `$r0~$r5`.
- Return Value: `$r0~$r1`.
- Temporary Register: `$r15`. This is reserved for assembler instruction expansion.
- Trampoline Register: `$r16`. This is used as static chain register for nested function.

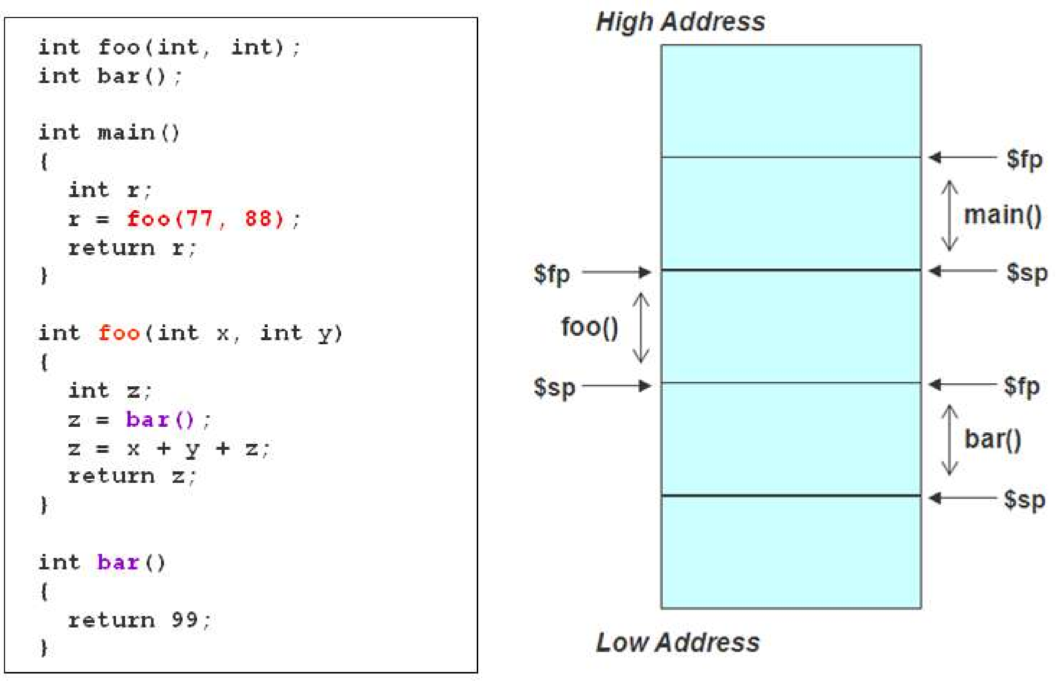
- Frame Pointer: `$r28`. This could be used for stack frame adjustment.
- Global Pointer: `$r29`. This is used to access small data area.
- Link Pointer: `$r30`. This is to save return address.
- Stack Pointer: `$r31`. This is used for stack frame adjustment.

Caller-saved and callee-saved registers are as follows:

- Caller-saved registers: `$r0~$r5`, `$r16~$r27`.
- Callee-saved registers: `$r6~$r10`, `$r11~$r14`, `$r28`, `$r29`, `$r30`.

8.2.1.2 Stack Frame

Stack frame is very important during the function invocation. Whenever caller invokes callee, the return address is automatically saved in `$lp` register, and then a corresponding stack frame is created in memory to store local variables, spill registers, and pass arguments. The stack is full-descending and each stack frame of a function is held by frame pointer (`$fp`) and stack pointer (`$sp`) with 8-byte alignment. Figure 1 below exemplifies such a scenario:



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

int main()
{
  int r;
  r = foo(77, 88);
  return r;
}

int foo(int x, int y)
{
  int z;
  z = bar();
  z = x + y + z;
  return z;
}

int bar()
{
  return 99;
}
```

Figure 1. ABI2 Stack Frame Scenario

Every stack frame is composed of 4 blocks: callee-saved area, local variables, duplicate incoming arguments, and outgoing arguments. Each block is also 8-byte alignment, so padding bytes may be needed. Note that the padding bytes in outgoing arguments block are in different direction conforming to C language standards. See Figure 2 for the memory layout of these 4 blocks within an ABI2 stack frame.

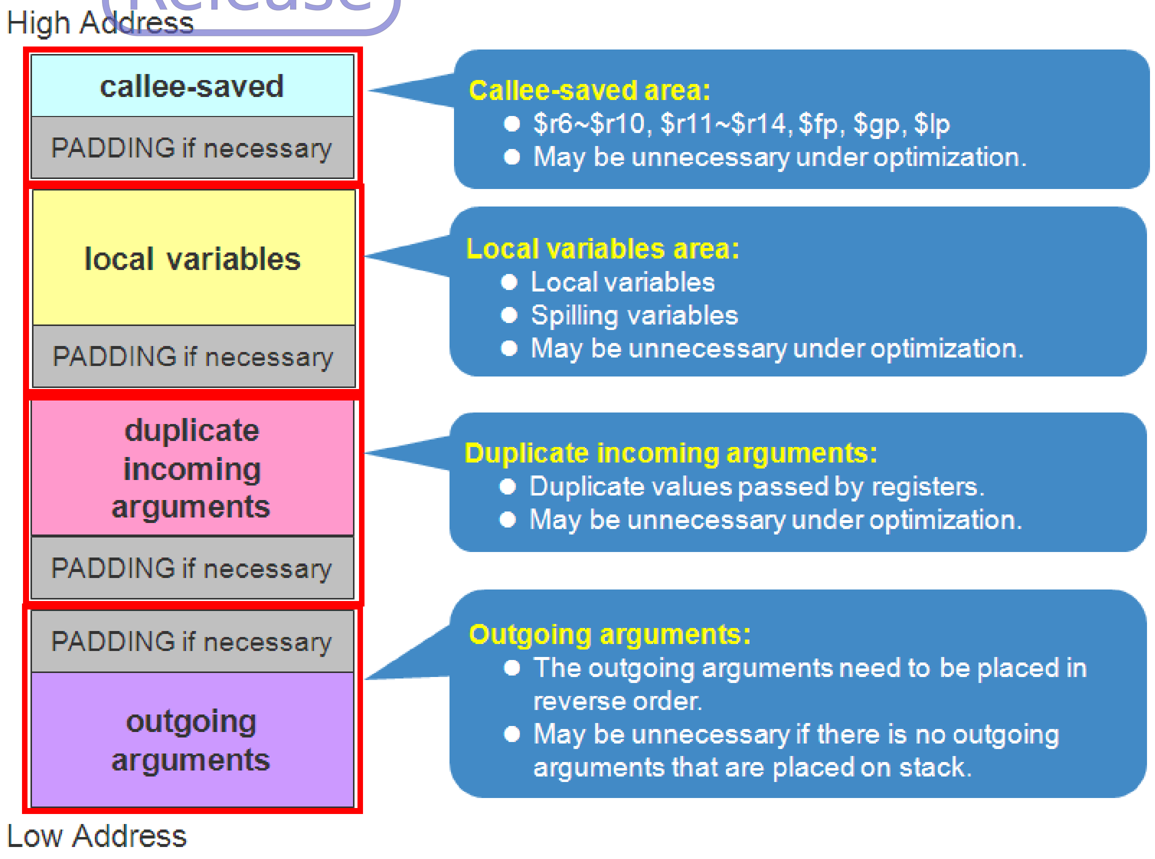

High Address
callee-saved
PADDING if necessary
local variables
PADDING if necessary
duplicate incoming arguments
PADDING if necessary
PADDING if necessary
outgoing arguments
Low Address
Callee-saved area:
● $r6~$r10, $r11~$r14, $fp, $gp, $lp
● May be unnecessary under optimization.
Local variables area:
● Local variables
● Spilling variables
● May be unnecessary under optimization.
Duplicate incoming arguments:
● Duplicate values passed by registers.
● May be unnecessary under optimization.
Outgoing arguments:
● The outgoing arguments need to be placed in reverse order.
● May be unnecessary if there is no outgoing arguments that are placed on stack.

Figure 2. ABI2 Stack Frame Layout

Conceptually, function prologue and epilogue are in charge of stack frame construction and destruction respectively. The register `$sp` will be adjusted to reserve a space for blocks and the register `$lp` will be used to return to caller after callee is finished. If the compiler option `-fno-omit-frame-pointer` is applied, the register `$fp` will also be involved in stack frame creation to record the original `$sp` position.

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

- Prologue
  1. Push callee-saved registers into stack. The caller's frame pointer (`$fp`) and return address (`$lp`) are also pushed if necessary.
  2. Set frame pointer (`$fp`) to the base of current stack frame.
  3. Calculate required size, and then adjust stack pointer (`$sp`) to the bottom of current stack frame.

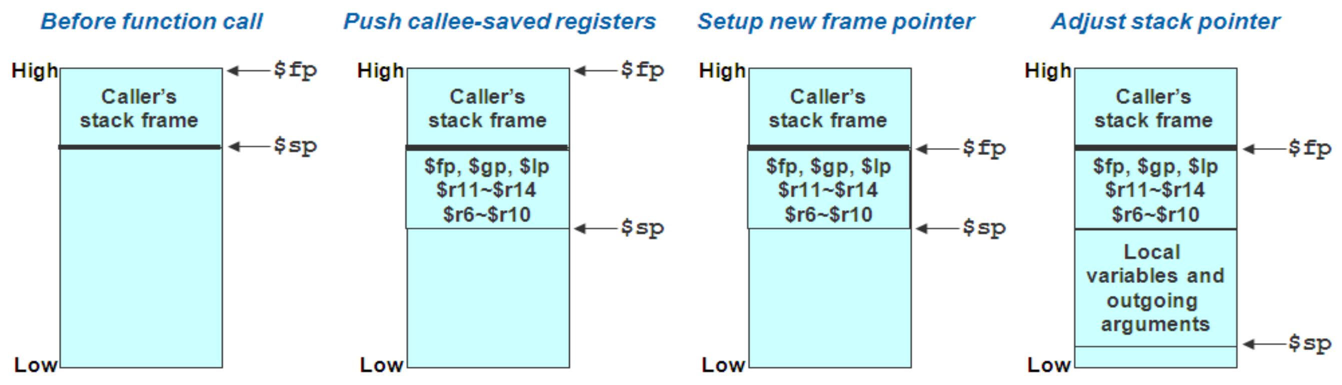


Figure 3. Function Prologue for Stack Frame Construction

- Epilogue
  1. Adjust stack pointer back to the location where callee-saved registers are going to be popped.
  2. Pop callee-saved registers from stack to restore their content.
  3. Use link pointer (`$lp`) to return to caller.

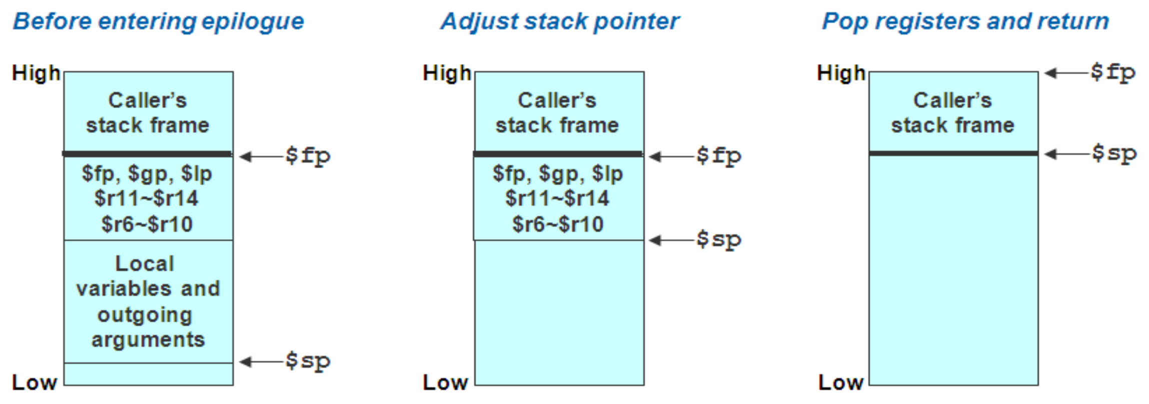


Figure 4. Function Epilogue for Stack Frame Destruction

8.2.1.3 Argument Passing and Return

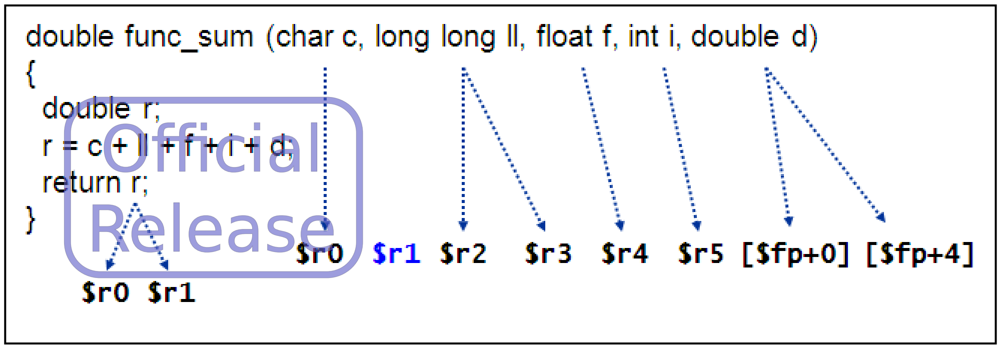
Arguments are passed in GPRs and stack. The space of stacked arguments, which is the outgoing arguments block of a stack frame, must be allocated by caller. The argument passing strategy includes the following rules:

- GPRs `$r0~$r5` are used to pass arguments.
- If the argument requires 8-byte alignment, assign the argument to the next even register number.
- If the argument is a primitive type smaller than 4 bytes, it will be zero- or sign-extended to 4 bytes.
- If GPRs `$r0~$r5` are not sufficient to hold all arguments, the remaining ones will be passed in the outgoing arguments block of caller's stack frame. Then callee is able to retrieve them by using `$fp` or `$sp` with offset calculation.
- If the argument is a composite type with a size that is not 4-byte aligned, it will be rounded up to the closest multiple of 4 bytes.
- 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.

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

- If the result is a primitive type,
  1. For primitive type smaller than 4 bytes: the return value is zero- or sign-extended to 4 bytes and returned in `$r0`.
  2. For 4-byte primitive type, the return value is returned in `$r0`.
  3. For 8-byte primitive type, the return value is returned in `$r0` and `$r1`.
- If the result is a composite type,
  1. For the size that is not larger than 8 bytes, the return value follows the same rules as when the result is a primitive type.
  2. For the size that is larger than 8 bytes or undetermined by caller and callee, the return value must be 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 `$r0` and the first argument will be passed in `$r1`.

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

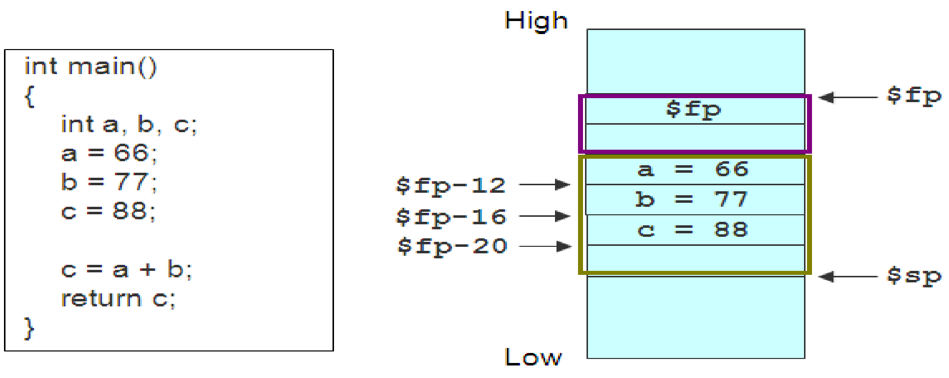

double func_sum (char c, long long ll, float f, int i, double d)
{
double r;
r = c + ll + f + i + d;
return r;
}
$r0 $r1 $r2 $r3 $r4 $r5 [$fp+0] [$fp+4]
$r0 $r1

Note that for a function with variable size (variadic function), caller is able to pass arguments like a normal function using GPRs and stack; callee is in charge of pushing argument registers into stack so that all the nameless arguments appear to have been passed consecutively in the memory for accessing. The callee must create an extra block, which is also 8-byte alignment, to store nameless arguments that are passed via GPRs. An example is given in Figure 7 in the next section.

8.2.1.4 Samples of ABI2

In this section, some C code fragments are presented as examples to show the memory layout generated by compiler. These samples are all compiled with the compiler option "`-O0 -fno-omit-frame-pointer`".

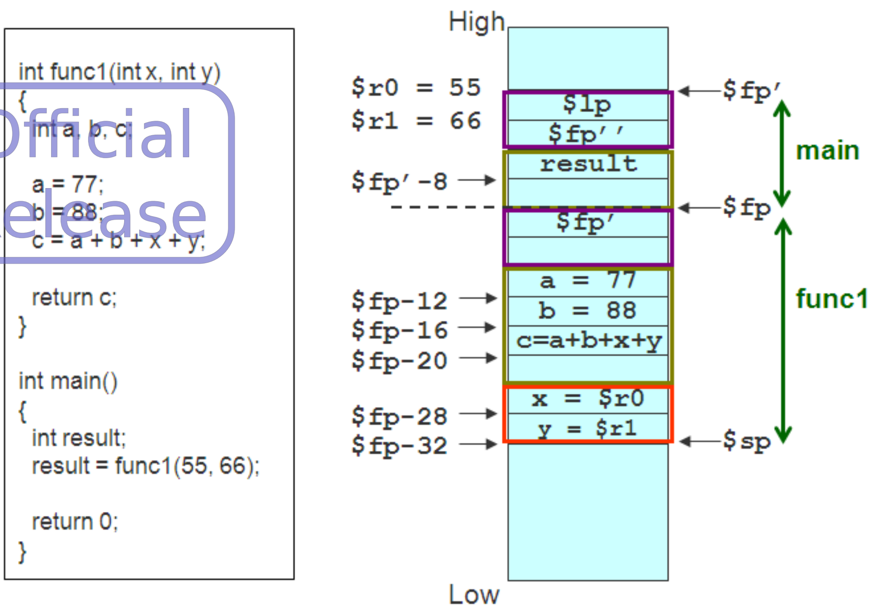
- A simple case of a function stack frame: It only contains blocks of callee-saved area and local variables. There is no need to duplicate incoming arguments or reserve a block for outgoing arguments.



```
int main()
{
    int a, b, c;
    a = 66;
    b = 77;
    c = 88;

    c = a + b;
    return c;
}
```

Figure 5. ABI2 Sample of Simple Function Stack Frame

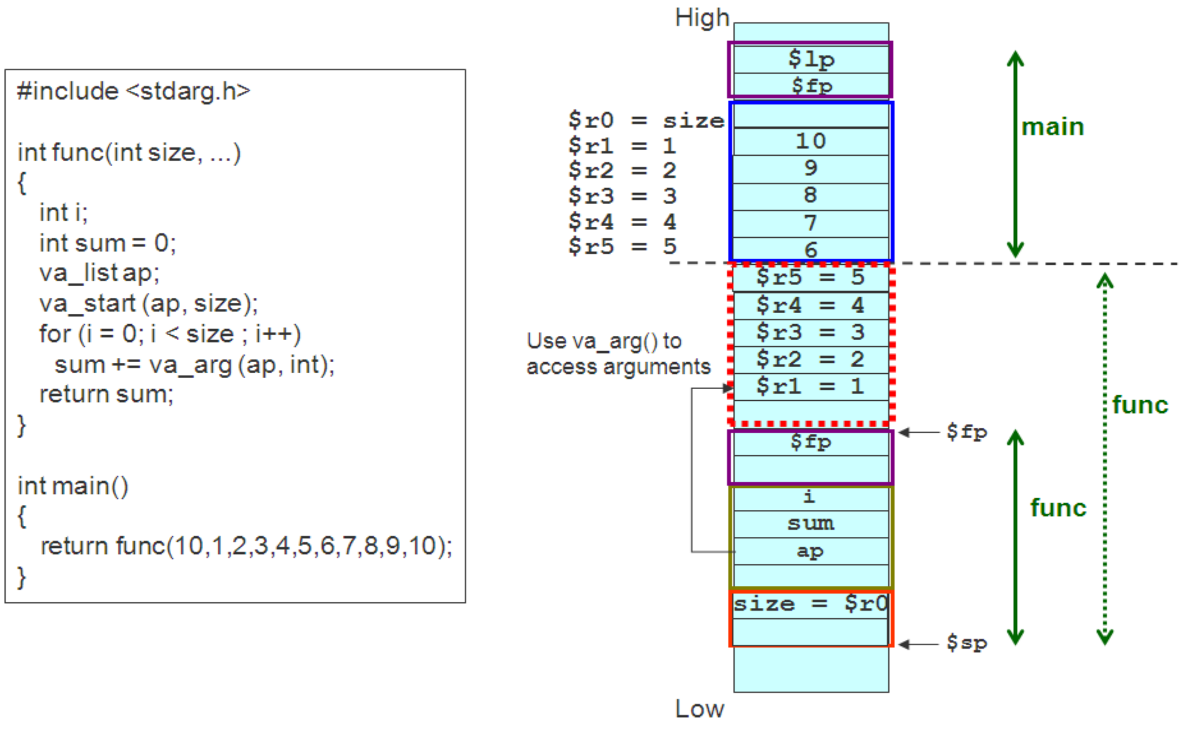
- A case of calling a function with arguments: Figure 6 shows the necessary blocks of each stack frame.



```
int func1(int x, int y)
{
  int a, b, c;

  a = 77;
  b = 88;
  c = a + b + x + y;

  return c;
}

int main()
{
  int result;
  result = func1(55, 66);

  return 0;
}
```

Figure 6. ABI2 Sample of Calling a Function with Arguments

- A case of variadic function: The nameless arguments are pushed into stack by callee.



```
#include <stdarg.h>

int func(int size, ...)
{
   int i;
   int sum = 0;
   va_list ap;
   va_start (ap, size);
   for (i = 0; i < size ; i++)
      sum += va_arg (ap, int);
   return sum;
}

int main()
{
   return func(10,1,2,3,4,5,6,7,8,9,10);
}
```

Figure 7. ABI2 Sample of Calling a Variadic Function

8.2.2. ABI2FP+ (for v3s and v3f Toolchains)

8.2.2.1 Registers

In addition to the GPRs usage in ABI2, there are extra Floating Pointer Registers (FPRs) and instructions for float/double computation in floating-point toolchain. It is helpful to generate more efficient code. The following table lists the usage of those FPRs under the ABI2FP+ convention.

Table 7. Andes FPRs with ABI Usage Convention

Register	Comments
`$fs0~$fs1` (`$fd0`)	Argument / Return / Saved by caller
`$fs2~$fs3` (`$fd1`)	Argument / Saved by caller
`$fs4~$fs5` (`$fd2`)	Argument / Saved by caller
`$fs6~$fs21` (`$fd3~$fd10`)	Saved by callee
`$fs22~$fs31` (`$fd11~$fd15`)	Saved by caller

This table is incorporated with the GPRs table usage of ABI2 (Table 6). It is clear from Table 7 that `$fs0~$fs1` are also used to return float/double value of a function.

As for caller-saved and callee-saved registers, they are listed below:

- Caller-saved registers: `$fs0~$fs5`, `$fs22~$fs31.`
- Callee-saved registers: `$fs6~$fs21.`

8.2.2.2 Stack Frame

The stack frame scenario of ABI2FP+ is almost the same as ABI2 except that there will be FPRs in callee-saved area. Therefore, some FPRs are considered to be pushed into stack in the prologue and their content will be restored in the epilogue. The difference of stack frame between ABI2 and ABI2FP+ are illustrated in the figure below:

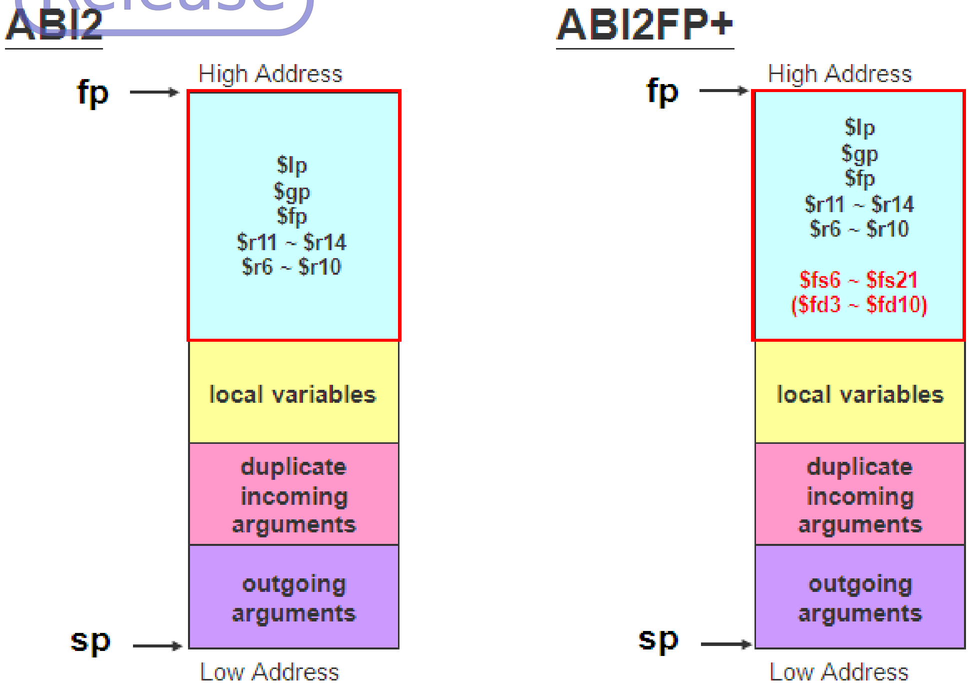


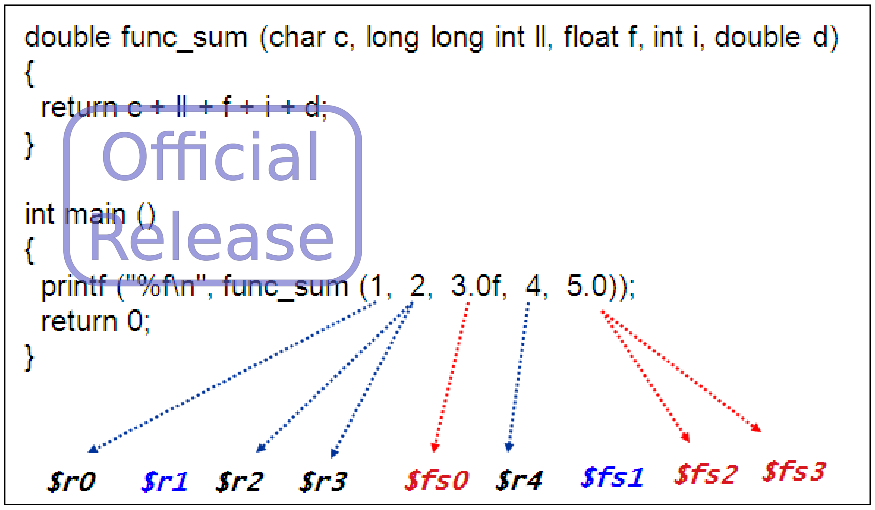
Figure 8. Stack Frame Comparison Between ABI2 and ABI2FP+

8.2.2.3 Argument Passing and Return

In ABI2FP+, arguments are passed in GPRs, FPRs, and stack. The rules of passing arguments and return value are based on ABI2 strategy with some differences:

- Function arguments with floating-point primitive types such as “float” and “double” will be passed in FPRs `$fs0~$fs5`; other primitive types are still passed in GPRs `$r0~$r5`.
- If the argument requires 8-byte alignment, assign the argument to the next even register number. Both GPR and FPR argument passing follows such a rule.
- An argument must be passed entirely in registers or entirely pushed on the stack.
- A function value of “float” will be returned in `$fs0`.
- A function value of “double” will be returned in `$fs0~$fs1`.

Here is an example of how arguments are passed under ABI2FP+:



```
double func_sum (char c, long long int ll, float f, int i, double d)
{
  return c + ll + f + i + d;
}

int main ()
{
  printf ("%f\n", func_sum (1,  2,  3.0f,  4,  5.0));
  return 0;
}
```

In addition to the rules above, there is also a major difference between ABI2 and ABI2FP+ in functions with variable size (variadic function). As FPRs are involved in passing arguments, it makes complexity, low performance and large code size of dealing with GPRs and FPRs against arguments order if callee is in charge of pushing argument registers into stack. Therefore, in the ABI2FP+, all the nameless arguments must be stored in outgoing arguments block of a stack frame by caller so that callee is able to access them via stack easily.

9. Andes Specifics

9.1. Get PC

For most V3-family toolchains, you can use the instruction "`MFUSR rt5,PC`" to move PC to the specified general purpose register `rt5`. However, for v3m toolchains, you must get PC through the general way "`JAL 4`", which stored the address of the next instruction into `$lp`. While this works fine, it does cause penalty on hardware branch prediction since it simply throws the whole prediction off balance.

9.2. Andes Predefined Macros

To see the default values of Andes predefined macros for a particular toolchain or to check if a feature is enabled as default, issue the following command:

```
$ nds32le-elf-gcc -E -dM - < /dev/null | grep NDS32
```

Predefined macros are very useful to determine which toolchain is used. The following lists the macros defined for different toolchain settings or compilation flags:

Table 8. Andes Predefined Macros

Macro Name	Description
`__NDS32__` `__nds32__`	Defined on all Andes toolchains.
`__NDS32_EB__`	Defined if using big endian toolchains.
`__NDS32_EL__`	Defined if using little endian toolchains.
`__NDS32_ABI_2__`	Defined if using ABI 2.

Macro Name	Description
__NDS32_ABI_2FP_PLUS__	Defined if using ABI2FP+.
__NDS32_ISA_V3__	Defined if using v3/v3j/v3s/v3f toolchains.
__NDS32_ISA_V3M__	Defined if using v3m toolchains.
__NDS32_REDUCED_REGS__	Defined if using GCC with the option to use reduced-set registers for register allocation (-mreduced-regs).
__NDS32_16_BIT__	Defined if using GCC with the option to generate 16-bit instructions (-m16-bit).
__NDS32_CMOV__	Defined if using GCC with the option to generate conditional move instructions (-mcmov).
__NDS32_GP_DIRECT__	Defined if using GCC with the small or medium code model option (–mcmodel=[small|medium]).
__NDS32_ISR_VECTOR_SIZE_4__	Defined if using GCC with the option to specify the size of each interrupt vector as 4 bytes (-misr-vector-size=4).
__NDS32_ISR_VECTOR_SIZE_16__	Defined if using GCC with the option to specify the size of each interrupt vector as 16 bytes (-misr-vector-size=16).
__NDS32_EXT_PERF__	Defined if using GCC with the option to generate performance extension instructions (-mext-perf).
__NDS32_EXT_PERF2__	Defined if using GCC with the option to generate performance extension version 2 instructions (-mext-perf2).
__NDS32_EXT_STRING__	Defined if using GCC with the option to generate string extension instructions (-mext-string).
__NDS32_EXT_DSP__	Defined if using GCC with the option to generate DSP extension instructions (-mext-dsp).
__NDS32_EXT_ZOL__	Defined if using GCC with the option to insert the hardware loop directive (-mext-zol).

Macro Name	Description
`__NDS32_EXT_FPU_SP__`	Defined if using GCC with the option to generate single-precision floating-point instructions (`-mext-fpu-sp`).
`__NDS32_EXT_FPU_DP__`	Defined if using GCC with the option to generate double-precision floating-point instructions (`-mext-fpu-dp`).
`__NDS32_EXT_FPU_FMA__`	Defined if using GCC with the option to generate floating-point multiply-accumulation instructions (`-mext-fpu-fma`).
`__NDS32_EXT_FPU_CONFIG_0__`	Defined if using GCC with the options to generate single-precision floating-point instructions (`-mext-fpu-sp`) and to set the FPU configuration value as 0 or 4 (`-mconfig-fpu={0|4}`). For details about FPU configuration options, please refer to *AndeStar Instruction Set Architecture FPU Extension Manual*.
`__NDS32_EXT_FPU_CONFIG_1__`	Defined if using GCC with the options to generate single-precision floating-point instructions (`-mext-fpu-sp`) and to set the FPU configuration value as 1 or 5 (`-mconfig-fpu={1|5}`). For details about FPU configuration options, please refer to *AndeStar Instruction Set Architecture FPU Extension Manual*.
`__NDS32_EXT_FPU_CONFIG_2__`	Defined if using GCC with the options to generate single-precision floating-point instructions (`-mext-fpu-sp`) and to set the FPU configuration value as 2 or 6 (`-mconfig-fpu={2|6}`). For details about FPU configuration options, please refer to *AndeStar Instruction Set Architecture FPU Extension Manual*.
`__NDS32_EXT_FPU_CONFIG_3__`	Defined if using GCC with the options to generate single-precision floating-point instructions (`-mext-fpu-sp`) and to set the FPU configuration value as 3 or 7 (`-mconfig-fpu={3|7}`). For details about FPU configuration options, please refer to *AndeStar Instruction Set Architecture FPU Extension Manual*.
`__NDS32_EXT_FPU_DOT_E__`	Defined if using GCC with the options to generate single-precision floating-point instructions (`-mext-fpu-sp`) and to set the FPU configuration value as 4, 5, 6, or 7 (`-mconfig-fpu={4|5|6|7}`). For details about FPU configuration options, please refer to *AndeStar Instruction Set Architecture FPU Extension Manual*.

The following takes __NDS32_EXT_PERF__ as an example to help you understand the usages of Andes predefined macros:

```
#if (__NDS32_EXT_PERF__)
        abs     $r0, $r0
#else
        bgez    $r0, .L1
        subri   $r0, $r0, 0
.L1:
#endif
```

9.2.1. Deprecated Predefined Macros

The following macros, though still supported for backward compatibility, are NOT recommended. They may be completely removed in the future:

Table 9. Obsolete Predefined Macros

Macro Name	Notes
NDS32_EB __NDS32_EB	Defined if using big endian toolchains.
NDS32_EL __NDS32_EL	Defined if using little endian toolchains.
NDS32_ABI_2 __NDS32_ABI_2	Defined if using ABI 2.
NDS32_BASELINE_V3 __NDS32_BASELINE_V3	Defined if using v3/v3j/v3s/v3f toolchains.
NDS32_BASELINE_V3M __NDS32_BASELINE_V3M	Defined if using v3m toolchains.
NDS32_REDUCE_REGS __NDS32_REDUCE_REGS	Defined if using GCC with the option to use reduced-set registers for register allocation (`-mreduced-regs`).
NDS32_EXT_PERF __NDS32_EXT_PERF	Defined if using GCC with the option to generate performance extension instructions (`-mext-perf`).
NDS32_EXT_PERF2 __NDS32_EXT_PERF2	Defined if using GCC with the option to generate performance extension version 2 instructions (`-mext-perf2`).
NDS32_EXT_STRING __NDS32_EXT_STRING	Defined if using GCC with the option to generate string extension instructions (`-mext-string`).
NDS32_EXT_FPU_SP __NDS32_EXT_FPU_SP	Defined if using GCC with the option to generate single-precision floating-point instructions (`-mext-fpu-sp`).
NDS32_EXT_FPU_DP __NDS32_EXT_FPU_DP	Defined if using GCC with the option to generate double-precision floating-point instructions (`-mext-fpu-dp`).

9.3. Crt0.S

The file crt0.S, in startup demo projects of Andes BSP package, contains the following AndesCore™-specific components:

- the vector table for interruptions (including exceptions and interrupts),
- the interruption dispatch examples, and
- the low-level initialization for C programs.

The vector table and interruption dispatch examples show the dispatch handling from assembly code to C functions for interrupts, useful exceptions, and error exceptions. You can modify the dispatch grouping, function names and the function definitions for your own needs. An example of changing dispatch grouping is that if a program is not intended to use the syscall exception, its handler can be changed from calling syscall_handler() to error_exception_handler().

In addition, crt0.S also invokes a predefined low-level initialization macro named nds32_init for the C compiler to support AndesCore features. The macro is enclosed between "Begin of do-not-modify" and "End of do-not-modify" after the symbol _start. **We strongly recommend that you do not touch the enclosed code sequence to ensure the proper program execution.**

Predefined in the toolchains, the nds32_init macro can be invoked in assembly code by including <nds32_init.inc> file. This macro is used to do the necessary startup initialization for the C program and AndesCore features. The following bullets explain the initialization code segment in nds32_init macro, including the special code sequence, the symbols used and their meanings:

- Symbol **_ITB_BASE_**

  The instruction sequence relating to _ITB_BASE_ is to initialize the instruction table register $ITB (User-Special Register USR #28) with the value of _ITB_BASE_.

  It is the base address of the instruction table used by ex9.it instruction. One usage of the instruction table is as follows. When linker performs code size optimization, it automatically assigns the value of _ITB_BASE_, fills the corresponding table with useful instructions, and generates ex9.it.

- Symbol **_SDA_BASE_**

  The instruction sequence relating to _SDA_BASE_ is to initialize the global data pointer register $gp (r29) with the value of _SDA_BASE_.

  It is the address in the middle of data sections. Linker places scalar data around it so that they can be accessed efficiently by $gp-based load/store instructions and their addresses can be calculated efficiently by $gp-based add instructions.

- Symbol **_stack**

  The instruction sequence relating to _stack is to initialize the stack pointer register $sp ($r31) with the value of _stack. Since _stack is a common symbol used by GNU toolchains, we follow its naming convention.

  It is the starting address of the stack used by C compiler to pass function parameters, local variables and return values. Linker obtains its value from the linker script. Since the stack usually goes from high addresses to low addresses when doing function calls, the initial stack address is normally set to the highest address of program data memory.

- FPU initialization

  The instruction sequence is to initialize the FPU and coprocessor enable control register $FUCOP_CTL and the floating-point control status register $FPCSR. It enables the floating-point support with denormalized flush-to-zero mode for FPU-based toolchains.

10. Andes C Language Extension for Interrupt Service Routine (Not Supported on S801)

Normally, programmers can't implement interrupt service routines in C language. This is because the standard C language is not designed for this job and the design of C function prologue and epilogue is not suitable for this task either. Unfortunately, implementing ISR in assembly language is a tedious and error-prone work. To relieve your burden, Andes defines three different syntaxes for system reset, interrupts and exceptions in C.

NOTE 1: Once Andes C language ISR is used, all ISR's entry points should be defined by Andes C language extension. Do not mix C language ISR with your assembly ISR unless you really know how to do it.

NOTE 2: You need to set $IPC to $IPC +4 before returning from C-ISR syscall. That is, the statement "`ptr->ipc = ptr->ipc + 4`" has to be added to your syscall function.

10.1. Syntax for System Reset Handler

Prototype:

`void NDS32ATTR_RESET("`*`<option_list>`*`") `*`reset_hdlr`*`(void);`

`<option_list>` contains zero or more of the following separated by "`;`"

1. *`vectors=XXX`*
2. *`nmi_func=YYY`*
3. *`warm_func=ZZZ`*

NOTE: The `NDS32ATTR_RESET("…")` can be replaced by `__attribute__((reset("…")))` macro. In this case, the prototype of system reset handler will be changed to –

`void __attribute__((reset("`*`<option_list>`*`"))) `*`reset_hdlr`*`(void);`

Functionality:

In Andes CPU core architecture, NMI, warm reset and cold reset share an interruption vector `0`, so special handling is necessary to distinguish one exception from another. Here Andes provides a framework which can hide the low level interfacing detail of tedious assembly coding and let you handle the real work in C language.

As soon as any of these exceptions occurs, the prologue of the reset handler generated by compiler will detect the event and dispatch the control to specific handlers that you provide with proper argument. Your handler will take over the control and do the specific job. When the job is done, the handler can decide whether or not to return the control to the reset handler. When it decides to return, an error code is used as the return value. This value can be `0` as OK or `-1` as fail. When the control goes back to the prologue of reset handler, it will either resume the operation before exception or prepare to do cold reset depending on whether the return value from the specific exception handler is `0` or `-1`.

Include File: `nds32_isr.h`

Input & Keyword:

reset_hdlr	Name of your reset handler function
reset	Keyword to signify that *reset_hdlr* is a reset handler
vectors=XXX	XXX is the total number of interruptions vectored entry point (Default: 16; 9 exception and 7 interrupt). This number is important since it is used to fill in the default handler if you don't define handlers for some vectors. For details, please refer to *AndeStar System Privilege Architecture Version 3 Manual*.
nmi_func=YYY	YYY is the name of NMI handler. (Default: NULL)
warm_func=ZZZ	ZZZ is the name of warm reset handler. (Default: NULL)

Note:

- A reset handler is mandatory in a system.
- Upon system reset, you need to put the whole system to a known state in order to use high level language like C. C language expects the .data section, .bss section and stack pointer are initialized, so global, static and auto variables can be used. This means the jobs to initialize DRAM, copy data from ROM to .data section and zero out .bss section in DRAM. The problem is how to initialize DRAM in C without using DRAM as temporarily storage. The followings are some guidelines –
  - No auto and global variable can be used before DRAM is initialized.
  - No ordinary C code can be used to initialize DRAM.
  - Only constants and registers can be used.
  - Special C macros are designed using inline assembly to do this job. Please reference the C ISR example in Andes Board Support Package for these C macros.
- `_nds32_init_mem()`: the name of memory initialization function; called by 1st level reset handler. You must implement this callback function if the memory in the target system needs to be initialized by software. One of the examples of such memory is DRAM.
  - Prototype: `void __attribute__((no_prologue)) _nds32_init_mem(void);`

Prototype:

`int nmi_func(int *reg_ptr)`

Functionality:

This is the handler that you provide to handle a NMI exception. When an NMI exception occurs, all general purpose registers are preserved to a buffer in stack and the starting address of this buffer is passed to `nmi_func` as the input.

NOTE: The address of `nmi_func` handler is stored at the "`.nds32_nmih`" section.

Input:

reg_ptr	Pointer to buffer containing values of all GPRs. The data is arranged in ascending order in the buffer based on register number. Sequence: ■ Reduced Register Set (16 registers mode): `r0-r10`, `r15`, `r28-r31` ■ Normal Register Set (32 registers mode): `r0-r31`

Return Value:

`0` means OK to resume the work before NMI occurs.

`-1` means fail and the prologue of reset handler will reset the system.

It is also OK to hold the control and never return to the reset handler.

Prototype:

`int warm_func(int *reg_ptr)`

Functionality:

This is the handler that you provide to handle a warm reset exception. When a warm reset exception occurs, all general purpose registers are preserved to a buffer in stack and the starting address of this buffer is passed to *warm_func* as the input.

NOTE: The address of *warm_func* handler is stored at the "`.nds32_wrh`" section.

Input:

reg_ptr	Pointer to buffer containing values of all GPRs. The data is arranged in ascending order in the buffer based on register number. Sequence: ■ Reduced Register Set (16 registers mode): `r0-r10`, `r15`, `r28-r31` ■ Normal Register Set (32 registers mode): `r0-r31`

Return Value:

`0` means OK to resume the work before warm reset occurs

`-1` means fail and the control should reset the system

It is also OK to hold the control and never return to the reset handler.

NOTE:

The warm reset and NMI handlers are not mandatory. Please see the examples provided below for format reference.

```
/* 8 interruptions; my_nmi as the name of NMI handler and no warm boot handler
*/
/* The following forms are equivalent */
   void NDS32ATTR_RESET("vectors=8;nmi_func=my_nmi;NULL")
        my_reset_hdlr(void);
   void NDS32ATTR_RESET("vectors=8;nmi_func=my_nmi")
        my_reset_hdlr(void);
   void NDS32ATTR_RESET("nmi_func=my_nmi;vectors=8")
        my_reset_hdlr(void);
```

```
/* 16 interruptions, no NMI and warm boot handler */
/* The following forms are equivalent */
void NDS32ATTR_RESET(“vectors=16”) my_reset_hdlr(void);
void NDS32ATTR_RESET(“”) my_reset_hdlr(void);
void NDS32ATTR_RESET() my_reset_hdlr(void);
```

10.1.1. Example

```
#include <nds32_isr.h> /* always include this file for ISR */

/*
my_reset() is a reset handler
Use my_nmi() to handle NMI
Use my_warmboot() to handle warm reset
To initiate memory, please implement the memory initiation function
“_nds32_init_mem()” mentioned earlier
*/
void NDS32ATTR_RESET(“vectors=16;nmi_func=my_nmi;warm_func=my_warmboot)
     my_reset(void);

void my_reset(void)
{

/* OK to use C statements now */
/* No global or static variables can be used yet */
/* Auto variables are OK to use now */
/* Initialize system registers here or do it later */
/* Initialize cache regs here so .data and .bss can be initialized faster
*/
__cpu_init();

/* Initialize .data and .bss sections here, so global and static can be used
later */
__c_init() ;

/* OK to use global and static variables now */
/* Initialize cpu and peripheral here */
__soc_init();
/* Ready to call main() */
  main() ;
}

int my_warm_boot(int * pReg)
{
  /* Call error recovery handler to handle warm reset */
  return try_recover(pReg, NDS32_NUM_GPR) ;
}

int my_NMI(int * pReg)
{
#ifdef BLUE_SCREEN
  /* Show register values in blue screen */
  save_crash_info(pReg, NDS32_NUM_GPR) ;

  /* Never return */
  while ( 1 ) ;
#else
  /* Save register values in storage, so we can retrieve it later */
  save_crash_info(pReg, NDS32_NUM_GPR) ;

  /* Can't recover, return fail so reset handler will do a cold boot */
  return 0 ;
#endif
}
```

10.2. Syntax for Interrupt Handlers

Prototypes:

- **For save_caller_regs**

```
void NDS32ATTR_ISR("id=xxx[;save_caller_regs;<is_nested>]")
     intr_hdlr(int vid);
```

where *save_caller_regs* and *<is_nested>* can be omitted.

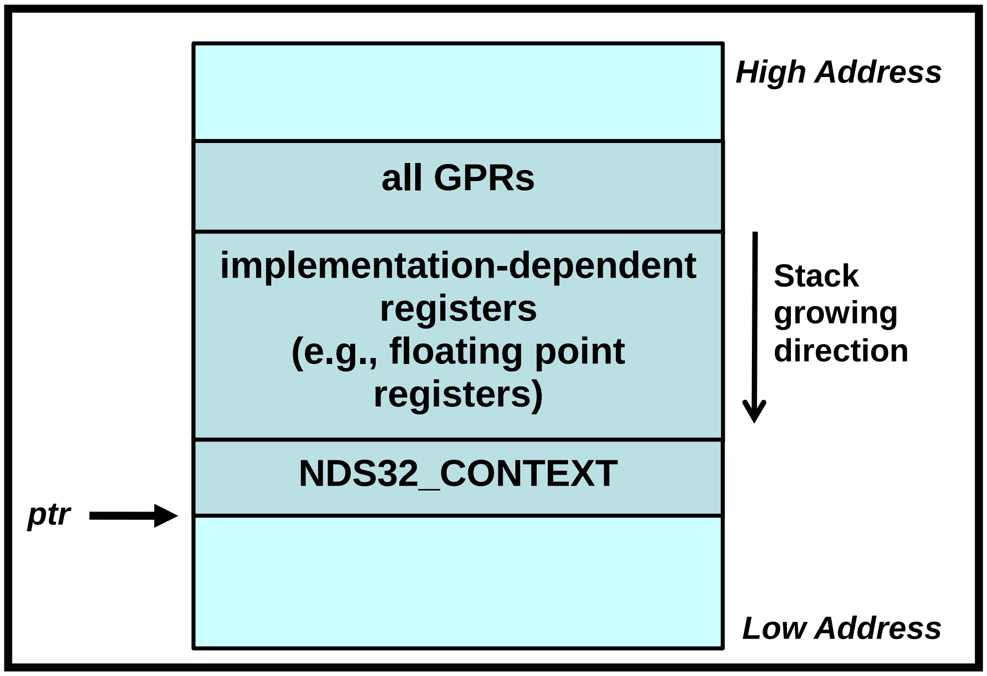
`save_caller_regs` means system will help save caller registers before entering this user-defined handler. Typical interrupt service routines should use this mode.

- **For save_all_regs**

```
void NDS32ATTR_ISR("id=xxx[;save_all_regs;<is_nested>]")
     intr_hdlr(int vid, NDS32_CONTEXT *ptr);
```

where *<is_nested>* can be omitted.

`save_all_regs` means system will help save all registers into stack before entering this user-defined handler. This mode can be used for context-switching. The stack layout looks like the following:



NOTE: In both prototypes, `NDS32ATTR_ISR("…")` can be replaced by `NDS32ATTR_INTERRUPT("…")` or `__attribute__((interrupt("…")))` macro.

Functionality:

An interrupt handler can take care of asynchronous events whether it is triggered by hardware or software. When you implement an interrupt handler, you must decide if this handler should run to completion without disturbance. If the handler allows other events to interrupt current job, it is said to be interruptible. Then, the next thing you must decide is when the handler will allow this to happen. There are three cases that need different setting in hardware and Andes has defined a parameter to control them. Please see below for the usage. An experienced programmer may decide to set the handler to `not_nested` and handle the interrupt level and global interrupt (GIE) manually.

As an aside, if advanced users want to have full control of all registers, combination of critical type interrupt and inline assembly can be used to achieve this purpose.

NOTE: The addresses of all user-defined *`intr_hdlr`* handlers are stored at the "`.nds32_jmptbl`" section.

Include File: `nds32_isr.h`

```
typedef struct
{
  int    ipc;
  int    ipsw;
} NDS32_CONTEXT;
```

Input & Keyword:

`intr_hdlr`	Name of an interrupt service routine (ISR)
`vid`	Vector ID
`ptr`	A pointer to `NDS32_CONTEXT`
`interrupt`	Keyword to signify *`intr_hdlr`* is an ISR
`id=xxx`	A series of vector ID separated by comma ("`,`"); ID should be `0` to `63`. This list allows a handler to be shared by many vectors. At least one ID number is required.
`<is_nested>`	Set to `nested`, `not_nested`, `ready_nested` or `critical`. It can be omitted.

(Default: nested)

- nested means this ISR is interruptible.
- not_nested means this ISR is not interruptible.
- ready_nested means this ISR is interruptible after PSW.GIE (global interrupt enable) is set in the function body manually by calling __nds32__setgie_en(). This is to allow ISR to finish some short critical code before enabling interrupts.
- critical means this is a critical (and usually short) handler. This ISR is not interruptible. (Note: This handler MUST be a leaf function with no child function called. In addition, the handler is advised to be put in a separate C source file and compiled with "–mno-ifc" but no "–mext-zol" to prevent IFC_LP, LB, LE, and LC registers from being corrupted in this critical handler.).

10.2.1. Example

```
#include <nds32_isr.h>

/* Timer handler; shared by vector 0, 1 and 2; save caller registers;
interruptible */
/* The following forms are equivalent */
void NDS32ATTR_ISR("id=0,1,2;save_caller_regs;nested")
     timer_hdlr(int vid);
void NDS32ATTR_ISR("id=0,1,2;nested;save_caller_regs")
     timer_hdlr(int vid);
void NDS32ATTR_ISR("id=0,1,2”)
     timer_hdlr(int vid);

/* Default handler; shared by vector 4 and 5; save all registers; not
interruptible */
/* The following forms are equivalent */
void NDS32ATTR_ISR("id=4,5;save_all_regs;not_nested")
     default_hdlr(int vid, NDS32_CONTEXT  *ptr);
void NDS32ATTR_ISR("id=4,5;not_nested;save_all_regs")
     default_hdlr(int vid, NDS32_CONTEXT  *ptr);
```

10.3. Syntax for Exception Handlers

Prototype:

- **For save_caller_regs**

```
void NDS32ATTR_EXCEPT("id=xxx[;save_caller_regs;<is_nested>]")
     excpt_hdlr(int vid);
```

where *save_caller_regs* and *<is_nested>* can be omitted.

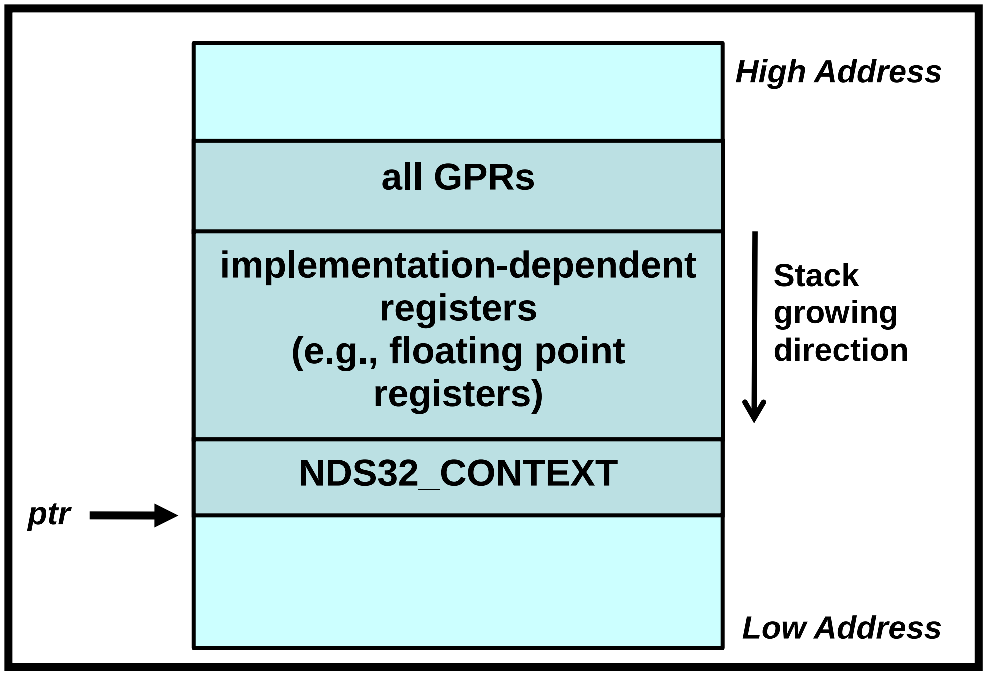
`save_caller_regs` means system will help save caller registers before entering this user-defined handler. Typical interrupt service routines should use this mode.

- **For save_all_regs**

```
void NDS32ATTR_EXCEPT("id=xxx[;save_all_regs;<is_nested>]")
     excpt_hdlr(int vid, NDS32_CONTEXT *ptr);
```

where *<is_nested>* can be omitted.

`save_all_regs` means system will help save all registers into stack before entering this user-defined handler. This mode can be used for context-switching. The stack layout looks like the following:



NOTE: In both prototypes, `NDS32ATTR_EXCEPT("…")` can be replaced by `NDS32ATTR_EXCEPTION("…")` or `__attribute__((exception("…")))` macro.

Functionality:

An exception handler can take care of synchronous events, such as division by zero or unaligned access, during the execution of software. When you implement an exception handler, you must decide whether this handler should run to completion without disturbance. If the handler allows other events to interrupt current job, it is said to be interruptible. Then, you must decide when the handler allows this to happen. Just like interrupt handler, there is a parameter to control it but exclude the usage of `nested` in the exception handler. Please see Section 10.3.1 for the usage. An experienced programmer may decide to set the handler to `not_nested` and handle the interrupt level and global interrupt (GIE) manually.

As an aside, if advanced users want to have full control of all registers, combination of critical type interrupt and inline assembly can be used to achieve this purpose.

NOTE 1: The addresses of all user-defined *`excpt_hdlr`* handlers are stored at the "`.nds32_jmptbl`" section.

NOTE 2: If a programmer needs to do the recovery, he or she should use the prototype for `save_all_regs`. Upon the occurrence of an exception, the current state of execution is saved in memory in struct `NDS32_CONTEXT` format. Then, a user-defined exception handler is invoked. After the exception has been processed, there are 2 possible actions to take by user-defined exception handler.

1. Skip the instruction that causes the exception: If it is System Call exception, the user-defined exception handler should add "`ptr->ipc = ptr->ipc + 4;`" before returning from syscall. For other exceptions, you need to know the size of the instruction in order to skip it. In struct `NDS32_CONTEXT`, there is a field called ipc which is the address that causes the exception. By looking at the contents there, you can determine the size of the instruction there. Please see Section 10.3.1.1 for an example that shows how to skip the instruction.
2. Resume the instruction that causes the exception: In this case, the user-defined exception handler should just return.

Include File: `nds32_isr.h`

Input & Keyword:

excpt_hdlr	Name of an exception handler
vid	Vector ID
ptr	A pointer to NDS32_CONTEXT
exception	Keyword to signify *excpt_hrld* is an exception handler
id=xxx	A series of vector ID separated by comma (“,”); ID should be 1 to 8. This list allows a handler to be shared by many vectors. At least one ID number is required.
<is_nested>	Set to nested, not_nested, ready_nested or critical. It can be omitted. (Default: not_nested)

- nested means this handler is interruptible.
- not_nested means this handler is not interruptible.
- ready_nested means this handler is interruptible after PSW.GIE (global interrupt enable) is set in the function body manually by calling `__nds32__setgie_en()`. This is to allow handler to finish some short critical code before enabling interrupts.
- critical means this is a critical (and usually short) handler. This handler is not interruptible. (Note: This handler MUST be a leaf function with no child function called. In addition, the handler is advised to be put in a separate C source file compiled with “-mno-ifc” but no “-mext-zol” to prevent IFC_LP, LB, LE, and LC registers from being corrupted in this critical handler..)

10.3.1. Example

```
#include <nds32_isr.h>

/* System call handler; ready_nested for you to enable GIE when you need.
*/
/* The following forms are equivalent */
void NDS32ATTR_EXCEPT("id=8;save_all_regs;nested")
     syscall_hdlr(int vid, NDS32_CONTEXT *ptr);
void NDS32ATTR_EXCEPT("id=8;nested;save_all_regs")
     syscall_hdlr(int vid, NDS32_CONTEXT *ptr);
void NDS32ATTR_EXCEPT("id=8;save_all_regs")
     syscall_hdlr(int vid, NDS32_CONTEXT *ptr);
```

10.3.1.1 Example of Skipping the Instruction that Causes the Exception

```
void NDS32ATTR_EXCEPT("id=7;save_all_regs ready_nested;") ge_hdlr(int vid,
NDS32_CONTEXT *ptr)
{
  unsigned char  inst;

/* Your exception handling code here.  */

/* About to return now, and we want to skip the instruction. */
  inst = *((unsigned char*) ptr->ipc);
  if (inst>>7)
  {
     /* Bit[7]: 1 represent 16-bit instruction.  */
     ptr->ipc += 2;
  }
  else
  {
     /* Bit[7]: 0 represent 32-bit instruction.  */
     ptr->ipc += 4;
  }
  return;
}
```

10.4. Linker Options

It is a must to link your program with a library call `libnds32_isr.a`.

Linker option `--defsym=_NDS32_VECTOR_BASE=expression` can be used to override the default base address, which is `0`.

10.4.1. Linker Script

```
EXTERN(_NDS32_VECTOR_BASE ) /* defined at the beginning of linker script */
PROVIDE (_NDS32_VECTOR_BASE = 0); /* defined inside SECTIONS */
. = _NDS32_VECTOR_BASE ;
.nds32_vector : { *(SORT_BY_NAME(.nds32_vector.*)) }
```

You can use linker option `--defsym=symbol=expression` to override the default base address

11. 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 replaced with 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.

Indirect call functions provide an easy implementation of ROM patching, yet its implementation before BSP v4.1.2 has a strict limitation on the ROM and flash address space, i.e. ±16MB. If you use a BSP version prior to v4.1.2 and have memory addressing beyond the limit, you'll have to resort to the other approach – function table mechanism.

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.

11.1. Indirect Call Functions

11.1.1. 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

11.1.1.1 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.

11.1.1.2 Add .nds32.ict Section to Linker Script

In addition to appending an attribute to function declaration, you also need to add a new section "`.nds32.ict`" to your linker script for ROM patching. To make the section overwritable, allocate it to the flash memory address as follows:

```
.nds32.ict FLASH_ADDRESS : { *(.nds32.ict) }
```

11.1.2. 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.

11.1.3. Tutorial

Given a code example like below, this section demonstrates how to perform ROM patching with indirect call:

```
#include <stdio.h>
#include <stdlib.h>
int func1(int);
int func2(int);
int func3(int);
int num1=1;
int num2=2;
int num3=3;

int main(void) {
  printf("func1(30)=%d\n", func1(30));
  printf("func2(30)=%d\n", func2(30));
  printf("func3(30)=%d\n", func3(30));
  return EXIT_SUCCESS;
}

int func1(int x) {
  return x * num1;
}
int func2(int x) {
  return x * num2;
}
int func3(int x)
{
  return x * num3;
}
```

Preparation: Modify your program by appending “indirect_call” attribute to patchable functions

Suppose that `func1`, `func2` and `func3` may need to be patched in the future, you can make them patchable by adding the “indirect_call” attribute to their declaration:

```
#include <stdio.h>
#include <stdlib.h>
int func1(int) __attribute__((indirect_call));
int func2(int) __attribute__((indirect_call));
int func3(int) __attribute__((indirect_call));
int num1=1;
int num2=2;
int num3=3;

int main(void) {
  printf("func1(30)=%d\n", func1(30));
  printf("func2(30)=%d\n", func2(30));
  printf("func3(30)=%d\n", func3(30));
  return EXIT_SUCCESS;
}

int func1(int x) {
  return x * num1;
}
int func2(int x) {
  return x * num2;
}
int func3(int x)
{
  return x * num3;
}
```

Preparation: Modify linker script by defining a .nds32.ict section

Next, add an `.nds32.ict` section to your linker script and set it to the flash address. Assuming that the base address of your flash memory is 0x510000, define the `.nds32.ict` section as follows:

```
…
.nds32.ict 0x510000 : { *(.nds32.ict) }
…
```

For a SaG-formatted file to be used with the linker script generator command `nds_ldsag` (see Chapter 15), define the `.nds32.ict` section as follows:

```
USER_SECTIONS .nds32.ict
LOAD 0x510000
{
  EXEC +0x0
      {
        * (.nds32.ict)
  }
}
```

Preparation: Compile and Link program with specific options

Depending on your address space layout, add a compilation flag from listed below to compile your program:

- `-mict-model=small` (enabled by default)

  This flag allocates 4 bytes for each call-site and is used if the address space between ROM and flash memory is within ±16MB.

- `-mict-model=large`

  This flag allocates 10 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 ±16MB, make sure you use this flag for compilation.

Then, use the options “`-Wl,--mexport-symbols=sym.ld`” to link the program and export the symbol addresses. Andes toolchain will generate `nds32_ict.s` as well as `sym.ld` after linking. Both `sym.ld` and `nds32_ict.s` are needed for patching functions. `sym.ld` contains all symbol addresses in the program and thus can prevent the linker from pulling the symbols again during the compilation of the patch code.

For example, with a linker script “`nds32.ld`” and an address space between ROM and flash memory more than 16MB, use the following commands to build the program:

```
nds32le-elf-gcc main.c -mict-model=large -Wl,-T,nds32.ld -o rom-patch-demo
-Wl,--mexport-symbols=sym.ld
```

The content of `nds32_ict.s` generated afterwards is as follows:

```
.section .nds32.ict, "ax"
.globl _INDIRECT_CALL_TABLE_BASE_
_INDIRECT_CALL_TABLE_BASE_:
   j   func2
   j   func3
   j   func1
```

DO NOT edit `nds32_ict.s`, not even to reorder the lines. It will break the program.

Create patch code

Now you can patch a function declared with the indirect_call attribute. For example, create patch code (`patch.c`) for `func2` as follows:

```
#include <stdio.h>
#include <stdlib.h>

int func2(int) __attribute__((indirect_call));

extern int num2;

int func2(int x) {
  return x * num2 * 10;
}
```

Modify linker script and sym.ld

Then, modify your linker script or SaG file so that both the patch code and the `.nds32.ict` section are set to the base address of the flash memory (0x510000 in this case). In this example, to ensure the linker know where to allocate the new `func2`, delete the line about `func2` in `sym.ld` (a file generated after linking) and modify the linker script or SaG file by adding “`INCLUDE “sym.ld”`” in the header and adding the `.nds32.ict` section.

Generate patch image

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

```
nds32le-elf-gcc patch.c nds32_ict.s -Wl,-T,patch.ld -o patch.out -nostdlib
-fno-zero-initialized-in-bss
```

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.

11.2. Function Table Mechanism

11.2.1. 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

11.2.1.1 Add 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};
```

11.2.1.2 Change 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));
```

11.2.1.3 Add 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) }
```

11.2.2. Limitations

- **Assembly code needs to be handled manually:** For example, replace “bal foo” with
```
la $ta, func_table
lwi $ta, [$ta + <offset of foo in func_table>]
jral $ta
```
- **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.

11.2.3. Tutorial

Given a code example like below, this section demonstrates how to perform ROM patching with function table mechanism:

```
#include <stdio.h>
#include <stdlib.h>
int func1(int);
int func2(int);
int func3(int);
int num1=1;
int num2=2;
int num3=3;

int main(void) {
  printf("func1(30)=%d\n", func1(30));
  printf("func2(30)=%d\n", func2(30));
  printf("func3(30)=%d\n", func3(30));
  return EXIT_SUCCESS;
}

int func1(int x) {
  return x * num1;
}
int func2(int x) {
  return x * num2;
}
int func3(int x)
{
  return x * num3;
}
```

Preparation: Modify program code

Suppose that func1, func2 and func3 may need to be patched in the future, define a func_table_t struct that contains variables for these functions and declare a variable func_table based on that structure. To prevent the compiler from optimizing out the indirection layer, DO NOT to define func_table as a const or static global variable. Then, modify each call-site for these functions so that they can be called via “func_table”.

```
#include <stdio.h>
#include <stdlib.h>
int func1(int);
int func2(int);
    int func3(int);
    int num1=1;
    int num2=2;
    int num3=3;

    typedef struct {
      int (*func1)(int);
      int (*func2)(int);
      int (*func3)(int);
    } func_table_t;

    func_table_t func_table __attribute__ ((section ("FUNC_TABLE"))); =
    {.func1 = func1,
     .func2 = func2,
     .func3 = func3};

    int main(void) {
      printf("func1(30)=%d\n", func_table.func1(30));
      printf("func2(30)=%d\n", func_table.func2(30));
      printf("func3(30)=%d\n", func_table.func3(30));
      return EXIT_SUCCESS;
    }

    int func1(int x) {
      return x * num1;
    }
    int func2(int x) {
      return x * num2;
    }
    int func3(int x)
    {
      return x * num3;
    }
```

Preparation: Modify linker script

Next, add a .FUNC_TABLE section to your linker script and set it to the flash address. Assuming that the base address of your flash memory is 0x510000, define the .FUNC_TABLE section in your linker script as follows:

```
  …
  .FUNC_TABLE 0x510000 : { *(.FUNC_TABLE) }
  …
```

Preparation: Link program with specific options

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

```
nds32le-elf-gcc main.c -Wl,-T,nds32.ld -o rom-patch -demo
-Wl,--mexport-symbols=sym.ld
```

Create patch code

Create a patch code that contains the same func_table_t struct. Note that the variables in the structure can't be reordered, added or removed. For example, create a patch code (patch.c) for func2 as follows:

```
  int func1(int);
  int func2(int);
```

```
int func3(int);

typedef struct {
  int (*func1)(int);
  int (*func2)(int);
  int (*func3)(int);
} func_table_t;

func_table_t func_table __attribute__ ((section ("FUNC_TABLE"))); =
{.func1 = func1,
 .func2 = func2,
 .func3 = func3};

extern int num2;

int func2(int x) {
  return x * num2 * 10;
}
```

Modify linker script and sym.ld

Then, modify your linker script or SaG file so that both the patch code and the `.FUNC_TABLE` section are set to the base address of the flash memory (0x510000 in this case). In this example, to ensure the linker know where to allocate the new `func2`, you can delete the line about `func2` in `sym.ld` (a file generated after linking) and modify the linker script or SaG file by adding "`INCLUDE "sym.ld"`" in the header and adding the `.FUNC_TABLE` section.

Generate patch image

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

```
nds32le-elf-gcc patch.c -Wl,-T,patch.ld -o patch.out -nostdlib
-fno-zero-initialized-in-bss
```

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.

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

NOTE: Be sure to use the correct signedness for arguments and return values when calling intrinsic functions. Starting from BSP v4.0 official, the compiler has a strict type checking. It gives warnings for incorrect signedness and reports errors if the option `-Werror` is specified.

12.1. Summary of Andes Intrinsic Functions

For each Andes intrinsic function, its syntax, mapped Andes instruction, and if compiler can schedule it or not (schedulable) are shown in the following tables.

Table 10. Intrinsics for Load/Store

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`unsigned int __nds32__llw(unsigned int *a)`	`LLW`	No	104
`char __nds32__lbup(unsigned char *a)`	`LBUP`	Yes	105
`unsigned int __nds32__lwup(unsigned int *a)`	`LWUP`	Yes	106
`unsigned int __nds32__scw(unsigned int *a, unsigned int b)`	`SCW`	No	108
`void __nds32__sbup(unsigned char *a, char b)`	`SBUP`	Yes	107
`void __nds32__swup(unsigned int *a, unsigned int b)`	`SWUP`	Yes	109

Table 11. Intrinsics for Read/Write System and USR Registers

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`unsigned int __nds32__mfsr(const enum nds32_sr srname)`	MFSR	No	111
`unsigned int __nds32__mfusr(const enum nds32_usr usrname)`	MFUSR	No	112
`void __nds32__mtsr(unsigned int val, const enum nds32_sr srname)`	MTSR	No	113
`void __nds32__mtsr_isb(unsigned int val, const enum nds32_sr srname)`	MTSR ISB	No	114
`void __nds32__mtsr_dsb(unsigned int val, const enum nds32_sr srname)`	MTSR DSB	No	115
`void __nds32__mtusr(unsigned int val, const enum nds32_usr usrname)`	MTUSR	No	116

Table 12. Miscellaneous Intrinsics

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`void __nds32__break(const unsigned int swid)`	BREAK	No	120
`void __nds32__cctlva_lck(const enum nds32_cctl_valck subtype, unsigned int *va)`	CCTL	No	121
`void __nds32__cctlidx_wbinval(const enum nds32_cctl_idxwbinv subtype, unsigned int idx)`	CCTL	No	121
`void __nds32__cctlva_wbinval_alvl(const enum nds32_cctl_vawbinv subtype, unsigned int *va)`	CCTL	No	121
`void __nds32__cctlva_wbinval_one_lvl(const enum nds32_cctl_vawbinv subtype, unsigned int *va)`	CCTL	No	121
`unsigned int __nds32__cctlidx_read(const enum`	CCTL	No	121

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
nds32_cctl_idxread subtype, unsigned int idx)			
void __nds32__cctlidx_write(const enum nds32_cctl_idxwrite subtype, unsigned int b, unsigned int idxw)	CCTL	No	121
void __nds32__cctl_l1d_invalall()	CCTL	No	121
void __nds32__cctl_l1d_wball_alvl()	CCTL	No	121
void __nds32__cctl_l1d_wball_one_lvl()	CCTL	No	121
void __nds32__dpref_qw(unsigned char *a, unsigned int b, const enum nds32_dpref subtype)	DPREF	No	124
void __nds32__dpref_hw(unsigned short int *a, unsigned int b, const enum nds32_dpref subtype)	DPREF	No	124
void __nds32__dpref_w(unsigned int *a, unsigned int b, const enum nds32_dpref subtype)	DPREF	No	124
void __nds32__dpref_dw(unsigned long long *a, unsigned int b, const enum nds32_dpref subtype)	DPREF	No	124
void __nds32__dsb()	DSB	No	126
unsigned int __nds32__get_current_sp()		No	127
unsigned long long __nds32__get_unaligned_dw(unsigned long long *a)		Yes	128
unsigned int __nds32__get_unaligned_w(unsigned int *a)		Yes	128
unsigned short __nds32__get_unaligned_hw(unsigned short *a)		Yes	128
void __nds32__isb()	ISB	No	129
void __nds32__isync(unsigned int *a)	ISYNC	No	130
void __nds32__jr_itoff(unsigned int a)	JR.ITOFF	No	131
void __nds32__jr_toff(unsigned int a)	JR.TOFF	No	132

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
void __nds32__jral_iton(unsigned int a)	JRAL.ITON	No	133
void __nds32__jral_ton(unsigned int a)	JRAL.TON	No	134
void __nds32__msync_all()	MSYNC	No	135
void __nds32__msync_store()	MSYNC	No	135
void __nds32__nop()	NOP	No	136
void __nds32__put_unaligned_dw(unsigned long long *a, unsigned long long data)		Yes	137
void __nds32__put_unaligned_w(unsigned int *a, unsigned int data)		Yes	137
void __nds32__put_unaligned_hw(unsigned short *a, unsigned short data)		Yes	137
unsigned int __nds32__return_address()		No	141
void __nds32__ret_itoff(unsigned int a)	RET.ITOFF	No	142
void __nds32__ret_toff(unsigned int a)	RET.TOFF	No	143
unsigned int __nds32__rotr(unsigned int val, unsigned int ror)	ROTR	Yes	138
void __nds32__schedule_barrier()		No	139
void __nds32__set_current_sp(unsigned int sp)		No	144
void __nds32__standby_no_wake_grant()	STANDBY	No	145
void __nds32__standby_wake_grant()	STANDBY	No	145
void __nds32__standby_wait_done()	STANDBY	No	145
void __nds32__teqz(const unsigned int a, const unsigned int swid)	TEQZ	No	149
void __nds32__tnez(const unsigned int a, const unsigned int swid)	TNEZ	No	149
void __nds32__trap(const unsigned int swid)	TRAP	No	149

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
void __nds32__setend_big()	SETEND	No	140
void __nds32__setend_little()	SETEND	No	140
unsigned int __nds32__sva(int a, int b)	SVA	Yes	146
unsigned int __nds32__svs(int a, int b)	SVS	Yes	147
void __nds32__syscall(const unsigned int swid)	SYSCALL	No	148
unsigned int __nds32__wsbh(unsigned int a)	WSBH	Yes	150

Table 13. Intrinsics for PE1 Instructions

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
int __nds32__abs(int a)	ABS	Yes	152
int __nds32__ave(int a, int b)	AVE	Yes	153
unsigned int __nds32__bclr(unsigned int a, const unsigned int pos)	BCLR	Yes	154
unsigned int __nds32__bset(unsigned int a, const unsigned int pos)	BSET	Yes	154
unsigned int __nds32__btgl(unsigned int a, const unsigned int pos)	BTGL	Yes	154
unsigned int __nds32__btst(unsigned int a, const unsigned int pos)	BTST	Yes	154
unsigned int __nds32__clip(int a, const unsigned int imm)	CLIP	Yes	156
int __nds32__clips(int a, const unsigned int imm)	CLIPS	Yes	157
unsigned int __nds32__clz(unsigned int a)	CLZ	Yes	159
unsigned int __nds32__clo(unsigned int a)	CLO	Yes	158

Table 14. Intrinsics for PE2 Instructions

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`void __nds32__bse(unsigned int *t, unsigned int a, unsigned int *b)`	BSE	Yes	161
`void __nds32__bsp(unsigned int *t, unsigned int a, unsigned int *b)`	BSP	Yes	162
`unsigned int __nds32__pbsad(unsigned int a, unsigned int b)`	PBSAD	Yes	163
`unsigned int __nds32__pbsada(unsigned int acc, unsigned int a, unsigned int b)`	PBSADA	Yes	164

Table 15. Intrinsics for String

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`int __nds32__ffb(unsigned int a, unsigned int b)`	FFB	Yes	166
`int __nds32__ffmism(unsigned int a, unsigned int b)`	FFMISM	Yes	168
`int __nds32__flmism(unsigned int a, unsigned int b)`	FLMISM	Yes	169

Table 16. Intrinsics for FPU

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`double __nds32__fcpynsd(double a, double b)`	FCPYNSD	Yes	171
`float __nds32__fcpynss(float a, float b)`	FCPYNSS	Yes	171
`double __nds32__fcpysd(double a, double b)`	FCPYSD	Yes	171
`float __nds32__fcpyss(float a, float b)`	FCPYSS	Yes	171
`unsigned int __nds32__fmfcsr()`	FMFcSR	No	174

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`void __nds32__fmtcsr(unsigned int fpcsr)`	FMTCSR	No	175
`unsigned int __nds32__fmfcfg()`	FMFCFG	Yes	173

Table 17. Intrinsics for TLBOP

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`void __nds32__tlbop_trd(unsigned int a)`	TLBOP	No	177
`void __nds32__tlbop_twr(unsigned int a)`	TLBOP	No	178
`void __nds32__tlbop_rwr(unsigned int a)`	TLBOP	No	179
`void __nds32__tlbop_rwlk(unsigned int a)`	TLBOP	No	180
`void __nds32__tlbop_unlk(unsigned int a)`	TLBOP	No	181
`void __nds32__tlbop_pb(unsigned int a)`	TLBOP	No	182
`void __nds32__tlbop_inv(unsigned int a)`	TLBOP	No	184
`void __nds32__tlbop_flua()`	TLBOP	No	185

Table 18. Intrinsics for Saturation ISA

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`int __nds32__kaddw(int a, int b)`	KADDW	Yes	187
`int __nds32__ksubw(int a, int b)`	KSUBW	Yes	188
`int __nds32__kaddh(int a, int b)`	KADDH	Yes	189
`int __nds32__ksubh(int a, int b)`	KSUBH	Yes	190
`int __nds32__kdmbb(unsigned int a, unsigned int b)`	KDMBB	Yes	191
`int __nds32__kdmbt(unsigned int a, unsigned int b)`	KDMBT	Yes	191

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`int __nds32__kdmtb(unsigned int a, unsigned int b)`	KDMTB	Yes	191
`int __nds32__kdmtt(unsigned int a, unsigned int b)`	KDMTT	Yes	191
`int __nds32__khmbb(unsigned int a, unsigned int b)`	KHMBB	Yes	192
`int __nds32__khmbt(unsigned int a, unsigned int b)`	KHMBT	Yes	192
`int __nds32__khmtb(unsigned int a, unsigned int b)`	KHMTB	Yes	192
`int __nds32__khmtt(unsigned int a, unsigned int b)`	KHMTT	Yes	192
`int __nds32__kslraw(int a, signed char b)`	KSLRAW	Yes	193
`unsigned int __nds32__rdov()`	RDOV	Yes	194
`void __nds32__clrov()`	CLROV	Yes	195

Table 19. Intrinsics for Interruption

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`void __nds32__setgie_dis()`	SETGIE	No	197
`void __nds32__setgie_en()`	SETGIE	No	197
`void __nds32__gie_dis()`		No	198
`void __nds32__gie_en()`		No	198
`void __nds32__enable_int(enum nds32_intrinsic int_id)`		No	199
`void __nds32__disable_int(enum nds32_intrinsic int_id)`		No	199
`void __nds32__set_pending_swint()`		No	201
`void __nds32__clr_pending_swint()`		No	201
`void __nds32__clr_pending_hwint(enum nds32_intrinsic int_id)`		No	202
`unsigned int __nds32__get_pending_int(enum nds32_intrinsic int_id)`		No	204

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
`unsigned int __nds32__get_all_pending_int()`		No	206
`void __nds32__set_int_priority(enum nds32_intrinsic int_id, unsigned int prio)`		No	207
`unsigned int __nds32__get_int_priority(enum nds32_intrinsic int_id)`		No	207
`unsigned int __nds32__get_trig_type(enum nds32_intrinsic int_id)`		No	209

Table 20. Intrinsics for COP Instructions

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page

`void __nds32__cpe1(const unsigned int cpn, const unsigned int cpi19)`
`void __nds32__cpe2(const unsigned int cpn, const unsigned int cpi19)`
`void __nds32__cpe3(const unsigned int cpn, const unsigned int cpi19)`
`void __nds32__cpe4(const unsigned int cpn, const unsigned int cpi19)`
	CPE1
	CPE2
	CPE3
	CPE4
		No	212

`void __nds32__cpld(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, signed int roffset, const unsigned int sv)`
`void __nds32__cpld_bi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, signed int roffset, const unsigned int sv)`
	CPLD
	CPLD.BI
		No	213

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
void __nds32__cpldi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, const signed int imm12) void __nds32__cpldi_bi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, const signed int imm12)	CPLDI CPLDI.BI	No	215
void __nds32__cplw(const unsigned int cpn, const unsigned int cprn, unsigned int *base, signed int roffset, const unsigned int sv) void __nds32__cplw_bi(const unsigned int cpn, const unsigned int cprn, unsigned int *base, signed int roffset, const unsigned int sv)	CPLW CPLW.BI	No	217
void __nds32__cplwi(const unsigned int cpn, const unsigned int cprn, unsigned int *base, const signed int imm12) void __nds32__cplwi_bi(const unsigned int cpn, const unsigned int cprn, unsigned int *base, const signed int imm12)	CPLWI CPLWI.BI	No	219
void __nds32__cpsd(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, signed int roffset, const unsigned int sv) void __nds32__cpsd_bi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, signed int roffset, const unsigned int sv)	CPSD CPSD.BI	No	221
void __nds32__cpsdi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, const signed int imm12) void __nds32__cpsdi_bi(const unsigned int cpn, const	CPSDI CPSDI.BI	No	223

Intrinsic Function Syntax	Mapped Andes Instruction	Schedulable	Page
unsigned int cprn, unsigned long long *base, const signed int imm12)			
void __nds32__cpsw(const unsigned int cpn, const unsigned int cprn, unsigned int *base, signed int roffset, const unsigned int sv) void __nds32__cpsw_bi(const unsigned int cpn, const unsigned int cprn, unsigned int *base, signed int roffset, const unsigned int sv)	CPSW CPSW.BI	No	225
void __nds32__cpswi(const unsigned int cpn, const unsigned int cprn, unsigned int *base, const signed int imm12) void __nds32__cpswi_bi(const unsigned int cpn, const unsigned int cprn, unsigned int *base, const signed int imm12)	CPSWI CPSWI.BI	No	227
unsigned long long __nds32__mfcpd(const unsigned int cpn, const unsigned int imm12)	MFCPD	No	229
unsigned int __nds32__mfcpw(const unsigned int cpn, unsigned const int imm12)	MFCPW	No	230
unsigned int __nds32__mfcppw(const unsigned int cpn, const unsigned int imm12)	MFCPPW	No	231
void __nds32__mtcpd(const unsigned int cpn, unsigned long long source, const unsigned int imm12)	MTCPD	No	232
void __nds32__mtcpw(const unsigned int cpn, unsigned int source, const unsigned int imm12)	MTCPW	No	233
void __nds32__mtcppw(const unsigned int cpn, unsigned int source, const unsigned int imm12)	MTCPPW	No	234

NOTE: Instruction scheduling is a compiler optimization used to improve instruction-level parallelism, which improves performance on machines with instruction pipelines. Namely, without changing the meaning of the code, it tries to avoid pipeline stalls by rearranging the order of instructions. The following is an instruction scheduling example:

- code example before instruction scheduling

```
...
InstructionA
InstructionB
InstructionC
...
```

- code example after instruction scheduling

```
...
InstructionA
InstructionC
InstructionB
...
```

12.2. Detailed Intrinsic Function Description

To help you quickly identify which intrinsic functions are available, each intrinsic function is specified with the Instruction Set Architecture (ISA) version and supported CPUs. The ISA version maintains backward compatibility, so a CPU with higher ISA version supports all intrinsic functions from the lower versions (but not vice versa). For example, a CPU with ISA V3 supports all intrinsic functions available in ISA V1 and V2. On the other hand, a CPU with ISA V1 does not support any intrinsic functions available in ISA V2 or V3. If a non-supported intrinsic function is executed, the CPU will generate a "Reserved Instruction Exception." Furthermore, during program execution or debugging, the ISA version can be identified by the value of the system register `MSC_CFG.BASEV`: `0` for V1, `1` for V2, and `2` for V3.

The following table shows examples of AndesCores supporting V3, V3m or V3m+ ISA and how the ISA versions are indicated by register bits of these cores.

AndeStar ISA	Examples of Supported AndesCores	Indication in Register Bits

V3	**N968, N1068, N1337, N15, D1088, D15**	`MSC_CFG.BASEV == 2`
		`& MSC_CFG.MCU == 0`

V3m	**N650, N705, N801, E801, S801**	`MSC_CFG.BASEV == 2`
		`& MSC_CFG.MCU == 1`
		`& MSC_CFG.IFC == 0`
		`& MSC_CFG.EIT == 0`

V3m+	**N820, E830**	`MSC_CFG.BASEV == 2`
		`& MSC_CFG.MCU == 1`
		`& MSC_CFG.IFC == 1`
		`& MSC_CFG.EIT == 1`

12.2.1. Intrinsics for Load/Store

The following table indicates the supported AndesCores for each intrinsic function introduced in this section.

Intrinsic Function	Supported CPUs	Page
`__nds32__llw`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	104
`__nds32__lbup`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	105
`__nds32__lwup`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	106
`__nds32__scw`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	108
`__nds32__sbup`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	107
`__nds32__swup`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	109

Name

`__nds32__llw`

Syntax

```
unsigned int __nds32__llw(unsigned int *a)
```

Where parameter "`*a`" is the memory address of variable "`a`".

Description

This intrinsic inserts a `LLW` instruction into the instruction stream. The memory address for the load locked operation is specified by `*a.`

Return Value

The `__nds32__llw` intrinsic returns the memory content of `*a`.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   …//We want to perform atomic read-modify-write operations for variable
rmw.
   unsigned int success;
   unsigned int rmw = 0x0000FFFF; //The initial value of rmw
    rmw =__nds32__llw(&rmw); //read
    … //modify
    success = __nds32__scw(&rmw, rmw); //write
                         //The variable success indicates if the SCW succeed.
   …
}
```

Name

`__nds32__lbup`

Syntax

```
char __nds32__lbup(unsigned char *a)
```

Where parameter “`*a`” is the memory address of variable “`a`”.

Description

This intrinsic inserts a `LBUP` instruction into the instruction stream. The memory address for the load operation with user mode privilege address translation is specified by `*a`.

Return Value

The `__nds32__lbup` intrinsic returns the memory content of `*a`.

Privilege Level: ALL

Name

`__nds32__lwup`

Syntax

`unsigned int __nds32__lwup(unsigned int *a)`

Where parameter "`*a`" is the memory address of variable "`a`".

Description

This intrinsic inserts a `LWUP` instruction into the instruction stream. The memory address for the load operation with user mode privilege address translation is specified by `*a`.

Return Value

The `__nds32__lwup` intrinsic returns the memory content of `*a`.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   unsigned int a;
   …
   a =__nds32__lwup(&a); //This performs memory load operation for variable.
                         //a with user mode privilege address translation
    … //processing
   __nds32__swup(&a, a); //This performs memory store operation for variable.
                         //a with user mode privilege address translation
   …
}
```

Name

`__nds32__sbup`

Syntax

```
void __nds32__sbup(unsigned char *a, char b)
```

Where:

Parameter “*a” is the memory address of variable “a”.

Parameter “b” is the byte to be stored.

Description

This intrinsic inserts a SBUP instruction into the instruction stream. The byte to be stored and the memory address for the store operation with user mode privilege address translation are specified by b and *a, respectively.

Privilege Level: ALL

Name

`__nds32__scw`

Syntax

```
unsigned int __nds32__scw(unsigned int *a, unsigned int b)
```

Where:

Parameter “`*a`” is the memory address of variable “`a`”.

Parameter “`b`” is the 32-bit word to be stored.

Description

This intrinsic inserts a `SCW` instruction into the instruction stream. The word to be stored and the memory address for the store conditional operation are specified by `b` and `*a`, respectively.

Return Value

If the store operation is successfully performed, `1` is returned. Otherwise, `0` is returned.

Privilege Level: ALL

Example

See also `__nds32__llw`

Name

`__nds32__swup`

Syntax

```
void __nds32__swup(unsigned int *a, unsigned int b)
```

Where:

Parameter "`*a`" is the memory address of variable "`a`".

Parameter "`b`" is the 32-bit word to be stored.

Description

This intrinsic inserts a `SWUP` instruction into the instruction stream. The word to be stored and the memory address for the store operation with user mode privilege address translation are specified by `b` and `*a`, respectively.

Privilege Level: ALL

Example

See also `__nds32__lwup`

12.2.2. Intrinsics for Read/Write System and USR Registers

The following table indicates the supported AndesCores for each intrinsic function introduced in this section.

Intrinsic Function	Supported CPUs	Page
`__nds32__mfsr`	All AndesCores	111
`__nds32__mfusr`	Only AndesCores with V3/V3m+ (but not with V3m) architecture	112
`__nds32__mtsr`	All AndesCores	113
`__nds32__mtsr_isb`	All AndesCores	114
`__nds32__mtsr_dsb`	All AndesCores	115
`__nds32__mtusr`	Only AndesCores with V3/V3m+ (but not with V3m) architecture	116

Name

`__nds32__mfsr`

Syntax

`unsigned int __nds32__mfsr(const enum nds32_sr srname)`

Where:

`srname` is an SR symbolic mnemonic with a prefix `NDS32_SR_`. For example, the symbolic mnemonic of processor status word register is `PSW` while its simple mnemonic is `IR0`. In this case, the legal srname is `NDS32_SR_PSW`, not `NDS32_SR_IR0`.

Description

This intrinsic returns the content of the SR specified by `srname`.

Return Value

The `__nds32__mfsr` intrinsic returns the content of the SR specified by `srname`.

Privilege Level: Superuser and above

Example

See also `__nds32__mtsr`.

Note:

If you specify a USR symbolic mnemonic as `srname`, compiler might generate a wrong instruction.

Name

`__nds32__mfusr`

Syntax

```
unsigned int __nds32__mfusr(const enum nds32_usr usrname)
```

Where:

`usrname` is a USR symbolic mnemonic with a prefix `NDS32_USR_`.

Description

This intrinsic returns the content of the USR specified by `usrname`.

Return Value

The `__nds32__mfusr` intrinsic returns the content of the USR specified by `usrname`.

Privilege Level: ALL

Example

See also `__nds32__mtusr`.

Note:

If you specify an SR symbolic mnemonic as `usrname`, compiler might generate a wrong instruction.

Name

`__nds32__mtsr`

Syntax

`void __nds32__mtsr(unsigned int val, const enum nds32_sr srname)`

Where:

`srname` is an SR symbolic mnemonic with a prefix `NDS32_SR_`. For example, the symbolic mnemonic of processor status word register is `PSW` while its simple mnemonic is `IR0`. In this case, the legal srname is `NDS32_SR_PSW`, not `NDS32_SR_IR0`.

Description

This intrinsic moves `val` to the SR specified by `srname`.

Privilege Level: Superuser and above

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   …
   unsigned int psw=__nds32__mfsr(NDS32_SR_PSW); //get the content of PSW.
    psw = psw | 0x00000080;
    __nds32__mtsr(psw, NDS32_SR_PSW); //set PSW.DT bit.
    __nds32__dsb();
   …
}
```

Note:

If you specify a USR symbolic mnemonic as `srname`, compiler might generate a wrong instruction.

Name

`__nds32__mtsr_isb`

Syntax

`void __nds32__mtsr_isb(unsigned int val, const enum nds32_sr srname)`

Where:

`srname` is an SR symbolic mnemonic with a prefix `NDS32_SR_`. For example, the symbolic mnemonic of processor status word register is `PSW` while its simple mnemonic is `IR0`. In this case, the legal srname is `NDS32_SR_PSW`, not `NDS32_SR_IR0`.

Description

This intrinsic moves `val` to the SR specified by `srname` and then executes an ISB instruction to make sure the new SR value can be observed by or affect any operation after this intrinsic function.

Privilege Level: Superuser and above

Example

```
#include “nds32_intrinsic.h”
void func(void)
{
   …
   unsigned int psw=__nds32__mfsr(NDS32_SR_PSW); //get the content of PSW.
    psw = psw | 0x00000040;
    __nds32__mtsr_isb(psw, NDS32_SR_PSW); //set PSW.IT bit.
   …
}
```

Note:

If you specify a USR symbolic mnemonic as `srname`, compiler might generate a wrong instruction.

Name

`__nds32__mtsr_dsb`

Syntax

```
void __nds32__mtsr_dsb(unsigned int val, const enum nds32_sr srname)
```

Where:

`srname` is an SR symbolic mnemonic with a prefix `NDS32_SR_`. For example, the symbolic mnemonic of processor status word register is `PSW` while its simple mnemonic is `IR0`. In this case, the legal srname is `NDS32_SR_PSW`, not `NDS32_SR_IR0`.

Description

This intrinsic moves `val` to the SR specified by `srname` and then executes a DSB instruction to make sure the new SR value can be observed by or affect any operation after this intrinsic function.

Privilege Level: Superuser and above

Example

```
#include “nds32_intrinsic.h”
void func(void)
{
   …
   unsigned int psw=__nds32__mfsr(NDS32_SR_PSW); //get the content of PSW.
    psw = psw | 0x00000080;
    __nds32__mtsr_dsb(psw, NDS32_SR_PSW); //set PSW.DT bit.
   …
}
```

Note:

If you specify a USR symbolic mnemonic as `srname`, compiler might generate a wrong instruction.

Name

`__nds32__mtusr`

Syntax

`void __nds32__mtusr(unsigned int val, const enum nds32_usr usrname)`

Where:

`usrname` is a USR symbolic mnemonic with a prefix `NDS32_USR_`.

Description

This intrinsic moves `val` to the USR specified by `usrname`.

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
       …
       unsigned int pfm_ctl=__nds32__mfusr(NDS32_SR_PFM_CTL);
        //get PFM_CTL
        pfm_ctl = pfm_ctl | 0x00000001;
        __nds32__mtusr(pfm_ctl, NDS32_SR_PFM_CTL); //enable PFMC0
       …
    }
    /* assume the access permission is enabled in user mode*/
```

Note:

If you specify an SR symbolic mnemonic as `usrname`, compiler might generate a wrong instruction.

12.2.3. Miscellaneous Intrinsics

The following table indicates the supported AndesCores for each intrinsic function introduced in this section.

Intrinsic Function	Supported CPUs	Page
`__nds32__break`	All AndesCores	120
`__nds32__cctlva_lck`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	121
`__nds32__cctlidx_wbinval`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	121
`__nds32__cctlva_wbinval_alvl`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	121
`__nds32__cctlva_wbinval_one_lvl`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	121
`__nds32__cctlidx_read`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	121
`__nds32__cctlidx_write`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	121
`__nds32__cctl_l1d_invalall`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	121
`__nds32__cctl_l1d_wball_alvl`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	121
`__nds32__cctl_l1d_wball_one_lvl`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	121
`__nds32__dpref_qw`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	124
`__nds32__dpref_hw`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	124
`__nds32__dpref_w`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	124

Intrinsic Function	Supported CPUs	Page
`__nds32__dpref_dw`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	124
`__nds32__dsb`	All AndesCores	126
`__nds32__get_current_sp`	All AndesCores	127
`__nds32__get_unaligned_dw`	All AndesCores	128
`__nds32__get_unaligned_w`	All AndesCores	128
`__nds32__get_unaligned_hw`	All AndesCores	128
`__nds32__isb`	All AndesCores	129
`__nds32__isync`	All AndesCores	130
`__nds32__jr_itoff`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	131
`__nds32__jr_toff`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	132
`__nds32__jral_iton`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	133
`__nds32__jral_ton`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	134
`__nds32__msync_all`	All AndesCores	135
`__nds32__msync_store`	All AndesCores	135
`__nds32__nop`	All AndesCores	136
`__nds32__put_unaligned_dw`	All AndesCores	137
`__nds32__put_unaligned_w`	All AndesCores	137
`__nds32__put_unaligned_hw`	All AndesCores	137
`__nds32__return_address`	All AndesCores	141
`__nds32__ret_itoff`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	142
`__nds32__ret_toff`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	143

Intrinsic Function	Supported CPUs	Page
`__nds32__rotr`	All AndesCores	138
`__nds32__schedule_barrier`	All AndesCores	139
`__nds32__set_current_sp`	All AndesCores	144
`__nds32__standby_no_wake_grant`	All AndesCores	145
`__nds32__standby_wake_grant`	All AndesCores	145
`__nds32__standby_wait_done`	All AndesCores	145
`__nds32__teqz`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	149
`__nds32__tnez`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	149
`__nds32__trap`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	149
`__nds32__setend_big`	All AndesCores	140
`__nds32__setend_little`	All AndesCores	140
`__nds32__sva`	All AndesCores	146
`__nds32__svs`	All AndesCores	147
`__nds32__syscall`	All AndesCores	148
`__nds32__wsbh`	All AndesCores	150

Name

`__nds32__break`

Syntax

```
void __nds32__break(const unsigned int swid)
```

Where:

`swid`	is a 15-bit constant value.

Description

This intrinsic unconditionally generates a breakpoint exception and transfers control to the breakpoint exception handler. The 15-bits `swid` is used as a parameter to distinguish different breakpoint features and usages.

------------Note------------

The case that `swid` > 32767 is not allowed. If it occurs, compiler will generate an error message of "`the argument swid in __nds32__break should be in the range 0-32767`".

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
       …
       __nds32__break(0x2C);
       …
    }
```

Name

__nds32__cctlva_lck
__nds32__cctlidx_wbinval
__nds32__cctlva_wbinval_alvl
__nds32__cctlva_wbinval_one_lvl
__nds32__cctlidx_read
__nds32__cctlidx_write
__nds32__cctl_l1d_invalall
__nds32__cctl_l1d_wball_alvl
__nds32__cctl_l1d_wball_one_lvl

Syntax

A. `void __nds32__cctlva_lck(const enum nds32_cctl_valck subtype, unsigned int *va)`

B. `void __nds32__cctlidx_wbinval(const enum nds32_cctl_idxwbinv subtype, unsigned int idx)`

C. `void __nds32__cctlva_wbinval_alvl(const enum nds32_cctl_vawbinv subtype, unsigned int *va,)`

D. `void __nds32__cctlva_wbinval_one_lvl(const enum nds32_cctl_vawbinv subtype, unsigned int *va,)`

E. `unsigned int __nds32__cctlidx_read(const enum nds32_cctl_idxread subtype, unsigned int idx)`

F. `void __nds32__cctlidx_write(const enum nds32_cctl_idxwrite subtype, unsigned int b, unsigned int idxw)`

G. `void __nds32__cctl_l1d_invalall()`

H. `void __nds32__cctl_l1d_wball_alvl()`

I. `void __nds32__cctl_l1d_wball_one_lvl()`

Where:

`*va`	is the virtual address for `cctl` operation.
`idx`	is a 32-bit constant which specifies the index and way for cache access.
`idxw`	is a 32-bit constant which specifies the index, way, and word offset for cache access.
`subtype`	specifies the subtype of the `cctl` operation. The detailed subtypes for various syntaxes are listed below

Syntax	CCTL subtype	Operations
A	`NDS32_CCTL_L1D_VA_FILLCK,` `NDS32_CCTL_L1D_VA_ULCK,` `NDS32_CCTL_L1I_VA_FILLCK,` `NDS32_CCTL_L1I_VA_ULCK`	Fill and lock, and unlock
B	`NDS32_CCTL_L1D_IX_WBINVAL,` `NDS32_CCTL_L1D_IX_INVAL,` `NDS32_CCTL_L1D_IX_WB,` `NDS32_CCTL_L1I_IX_INVAL`	IDX writeback and invalidate
C	`NDS32_CCTL_L1D_VA_INVAL,` `NDS32_CCTL_L1D_VA_WB,` `NDS32_CCTL_L1D_VA_WBINVAL,` `NDS32_CCTL_L1I_VA_INVAL`	All level VA writeback and invalidate
D	`NDS32_CCTL_L1D_VA_INVAL,` `NDS32_CCTL_L1D_VA_WB,` `NDS32_CCTL_L1D_VA_WBINVAL,` `NDS32_CCTL_L1I_VA_INVAL`	One level VA writeback and invalidate
E	`NDS32_CCTL_L1D_IX_RTAG,` `NDS32_CCTL_L1D_IX_RWD,` `NDS32_CCTL_L1I_IX_RTAG,` `NDS32_CCTL_L1I_IX_RWD`	Cache read
F	`NDS32_CCTL_L1D_IX_WTAG,` `NDS32_CCTL_L1D_IX_WWD,` `NDS32_CCTL_L1I_IX_WTAG,` `NDS32_CCTL_L1I_IX_WWD`	Cache write
G		Unlock all of the L1D cache lines and set the state of all of the L1D cache lines to invalid.
H		All level L1D cache writeback
I		One level L1D cache writeback

Description

This intrinsic inserts a `CCTL` instruction into the instruction stream. Please refer to the CCTL instruction in *AndeStar Instruction Set Architecture Manual* for the detailed description.

Return Value

Only `__nds32__cctlidx_read` returns the content of the cache location. All the others have no return values.

Privilege Level:

Privilege Level	Intrinsics
ALL	`__nds32__cctlva_wbinval_alvl` `__nds32__cctlva_wbinval_one_lvl`
Superuser and above	All the other types

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   …
   __nds32__cctl_l1d_invalall(); //invalid the whole data cache.
   __nds32__dsb();
   …
}
```

Name

__nds32__dpref_qw
__nds32__dpref_hw
__nds32__dpref_w
__nds32__dpref_dw

Syntax

```
void __nds32__dpref_qw(unsigned char *a, unsigned int b, const enum nds32_dpref
subtype)
void __nds32__dpref_hw(unsigned short int *a, unsigned int b, const enum
nds32_dpref subtype)
void __nds32__dpref_w(unsigned int *a, unsigned int b, const enum nds32_dpref
subtype)
void __nds32__dpref_dw(unsigned long long *a, unsigned int b, const enum
nds32_dpref subtype)
```

Where:

Parameter "*a"	is an address of an array element.
Parameter "b"	is the byte/half word/word/double word offset based on the data type in syntax.
subtype	defines subtype of the data prefetch operation.

Description

Depending on the type of variable b, this intrinsic inserts a DPREF or DPREFI instruction into the instruction stream. If b is a constant while using __nds32__dpref_w and __nds32__dpref_dw, DPREFI is inserted. Otherwise, DPREFI is inserted. It will perform a data prefetch operation for b^th array element from a. The subtype argument of this intrinsic is used as a hint to tell hardware the intended use of the prefetched data so that the hardware implementation may use different prefetch schemes to optimize the performance. Note that N9/N10/N13/N15/D10/D15 implementation prefetches the data cache block which contains a for all DPREF/DPREFI subtypes.

------------Note------------

The case that constant type b > 16383 and b < -16384 is not allowed in __nds32__dpref_w and __nds32__dpref_dw. If it occurs, compiler will generate an error message of "the constant type argument b in __nds32__dpref_w or __nds32__dpref_dw should be in the range between -16384 and 16383".

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   unsigned int a[100];
   …
   for (int i=0, i<100, i++)
    {
       __nds32__dpref(a, i+1, NDS32_DPREF_SRD); //prefetch a[i+1] for next.
                                                //iteration
       …
   }
   …
}
```

Name

`__nds32__dsb`

Syntax

`void __nds32__dsb()`

Description

This intrinsic inserts a DSB instruction into the instruction stream.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   …
   __nds32__setgie_dis(); //disable global interrupt
   __nds32__dsb(); //make sure the change in PSW.GIE is seen by any following.
                  //load/store instructions.
   …
}
```

Name

`__nds32__get_current_sp`

Syntax

`unsigned int __nds32__get_current_sp()`

Description

This intrinsic function returns the current stack point value.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
      unsigned int sp;
      sp = __nds32__get_current_sp(); //sp is the current stack point.
}
```

Name

```
__nds32__get_unaligned_dw
__nds32__get_unaligned_w
__nds32__get_unaligned_hw
```

Syntax

```
unsigned long long __nds32__get_unaligned_dw(unsigned long long *a)
unsigned int __nds32__get_unaligned_w(unsigned int *a)
unsigned short __nds32__get_unaligned_hw(unsigned short *a)
```

Where parameter "`*a`" is a memory address.

Description

These intrinsic functions perform unaligned memory read operation where `__nds32__get_unaligned_dw` gets a 64-bit data, `__nds32__get_unaligned_w` gets a 32-bit data, and `__nds32__get_unaligned_hw` gets a 16-bit data.

Privilege Level: ALL

Name

`__nds32__isb`

Syntax

`void __nds32__isb()`

Description

This intrinsic inserts an ISB instruction into the instruction stream.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   …
   unsigned int psw=__nds32__mfsr(NDS32_SR_PSW); //get the content of PSW.
    psw = psw | 0x00000040;
    __nds32__mtsr(psw, NDS32_SR_PSW); //set PSW.IT bit.
    __nds32__isb();
   …
}
```

Name

`__nds32__isync`

Syntax

```
void __nds32__isync(unsigned int *a)
```

Where parameter "`*a`" is an instruction address for serialization.

Description

This intrinsic inserts an ISYNC instruction into the instruction stream.

Privilege Level: ALL

Name

`__nds32__jr_itoff`

Syntax

```
void __nds32__jr_itoff(unsigned int a)
```

Where parameter “a” is an instruction address to be jumped to.

Description

This intrinsic branches unconditionally to an instruction address `a` and clears the IT field of the Processor Status Word (PSW) system register to turn off the instruction address translation process in the MMU. This intrinsic function guarantees that fetching of the target instruction will see PSW.IT as `0`, thus not going through the address translation process.

Privilege Level: ALL

Name

`__nds32__jr_toff`

Syntax

```
void __nds32__jr_toff(unsigned int a)
```

Where parameter "a" is an instruction address to be jumped to.

Description

This intrinsic branches unconditionally to an instruction address `a` and clears the IT and DT fields of the Processor Status Word (PSW) system register to turn off the instruction and data address translation process in the MMU. This instruction guarantees that fetching of the target instruction will see PSW.IT as `0` and PSW.DT as `0`, thus not going through the address translation process.

Privilege Level: ALL

Name

`__nds32__jral_iton`

Syntax

```
void __nds32__jral_iton(unsigned int a)
```

Where parameter "a" is an instruction address to be jumped to.

Description

This intrinsic branches unconditionally to an instruction address `a` and sets the IT field of the Processor Status Word (PSW) system register to turn on the instruction address translation process in the MMU. The program address of the next sequential instruction (PC+4) is written to Rt for the return of the function call. This intrinsic function guarantees that fetching of the target instruction will see PSW.IT as `1`, thus going through the address translation process.

Privilege Level: ALL

Name

`__nds32__jral_ton`

Syntax

`void __nds32__jral_ton(unsigned int a)`

Where parameter “a” is an instruction address to be jumped to.

Description

This intrinsic branches unconditionally to an instruction address `a` and sets the IT and DT fields of the Processor Status Word (PSW) system register to turn on the instruction and data address translation process in the MMU. The program address of the next sequential instruction (PC+4) is written to Rt for the return of the function call. This intrinsic function guarantees that fetching of the target instruction will see PSW.IT as `1` and PSW.DT as `1`, thus going through the address translation process.

Privilege Level: ALL

Name

`__nds32__msync*`

Syntax

A. `void __nds32__msync_all()`

B. `void __nds32__msync_store()`

Description

This intrinsic inserts an MSYNC instruction into the instruction stream.

`__nds32__msync_all` inserts an “MSYNC All” instruction into the instruction stream.

`__nds32__msync_store` inserts an “MSYNC Store” instruction into the instruction stream.

Privilege Level: ALL

Name

`__nds32__nop`

Syntax

```
void __nds32__nop()
```

Description

This intrinsic inserts an NOP instruction into the instruction stream.

Privilege Level: ALL

Name

```
__nds32__put_unaligned_dw
__nds32__put_unaligned_w
__nds32__put_unaligned_hw
```

Syntax

```
void __nds32__put_unaligned_dw(unsigned long long *a, unsigned long long data)
void __nds32__put_unaligned_w(unsigned int *a, unsigned int data)
void __nds32__put_unaligned_hw(unsigned short *a, unsigned short data)
```

Where:

Parameter "`*a`"	is a memory address.
Parameter "`data`"	is the data to be stored in `*a`.

Description

These intrinsic functions perform unaligned memory write operation where `__nds32__put_unaligned_dw` puts a 64-bit data, `__nds32__put_unaligned_w` puts a 32-bit data, and `__nds32__put_unaligned_hw` puts a 16-bit data.

Privilege Level: ALL

Name

`__nds32__rotr`

Syntax

`unsigned int __nds32__rotr(unsigned int val, unsigned int ror)`

Where:

`val`	is the value to be rotated
`ror`	is the rotation amount.

Description

This intrinsic right-rotates the content of `val`. The rotation amount is specified by `ror`. If `ror` is a constant, the ROTRI instruction will be inserted into the instruction stream. If `ror` is a variable, the ROTR instruction will be inserted. The result is returned.

------------Note------------

1. If `ror` is a variable, the rotation amount is specified by the low-order 5-bits of `ror`.
2. The case that constant `ror` > 31 is not allowed. If it occurs, compiler will generate an error message of "`the argument ror in __nds32__rotri should be in the range 0-31`".

Return Value

The `__nds32__rotr` intrinsic returns the value of `val` rotated by `ror`.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
    …
    unsigned int a=0x0000000F;
    a = __nds32__rotr(a, 4); //Variable a becomes 0xF0000000 after the right.
                             //rotation
    …
}
```

Name

`__nds32__schedule_barrier`

Syntax

`void __nds32__schedule_barrier()`

Description

This intrinsic creates a point so that instructions before and after the point won’t be merged by the compiler.

Name

`__nds32__setend_big`
`__nds32__setend_little`

Syntax

```
void __nds32__setend_big()
void __nds32__setend_little()
```

Description

`__nds32__setend_big` sets the data endian mode to big endian in the PSW register.

`__nds32__setend_little` sets the data endian mode to little endian in the PSW register.

Privilege Level: ALL

Example

```
     #include <nds32_intrinsic.h>
     void func(void)
     {
        …
        __nds32__setend_big(); //set the data endian mode to big endian.
        __nds32__dsb();  //make sure the change in PSW.BE is seen by any following.
                        //load/store instructions.
        …
     }
```

Name

`__nds32__return_address`

Syntax

`unsigned int __nds32__return_address()`

Description

This intrinsic function returns the return address.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
      unsigned int lp;
      lp = __nds32__return_address(); //lp is the return address of func.
}
```

Name

`__nds32__ret_itoff`

Syntax

```
void __nds32__ret_itoff(unsigned int a)
```

Where parameter "a" is an instruction address to be jumped to.

Description

This intrinsic branches unconditionally to an instruction address `a` and clears the IT field of the Processor Status Word (PSW) system register to turn off the instruction address translation process in the MMU. This intrinsic function guarantees that fetching of the target instruction will see PSW.IT as `0` and PSW.DT as `0`, thus not going through the address translation process.

Privilege Level: ALL

Name

`__nds32__ret_toff`

Syntax

```
void __nds32__ret_toff(unsigned int a)
```

Where parameter "a" is an instruction address to be jumped to.

Description

This intrinsic branches unconditionally to an instruction address `a` and also clears the IT and DT fields of the Processor Status Word (PSW) system register to turn off the instruction and data address translation process in the MMU. This intrinsic function guarantees that fetching of the target instruction will see PSW.IT as `0` and PSW.DT as `0`, thus not going through the address translation process.

Privilege Level: ALL

Name

`__nds32__set_current_sp`

Syntax

`void __nds32__set_current_sp(unsigned int sp)`

Description

This intrinsic function sets the current stack point value.

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
         //adjust sp value to sp - 4
         unsigned int sp;
         sp = __nds32__get_current_sp();
         sp = sp - 4;
         __nds32__set_current_sp(sp);
    }
```

Name

`__nds32__standby_no_wake_grant`
`__nds32__standby_wake_grant`
`__nds32__standby_wait_done`

Syntax

```
void __nds32__standby_no_wake_grant()
void __nds32__standby_wake_grant()
void __nds32__standby_wait_done()
```

Description

`__nds32__standby_no_wake_grant` inserts a "STANDBY no_wake_grant" instruction into the instruction stream.

`__nds32__standby_wake_grant` inserts a "STANDBY wake_grant" instruction into the instruction stream.

`__nds32__standby_wait_done` inserts a "STANDBY wait_done" instruction into the instruction stream.

Privilege Level: The behaviors of `__nds32__standby` under different processor operating modes are listed in the following table.

Privilege level	Intrinsic function	Andes instruction
User	`__nds32__standby_no_wake_grant`	`STANDBY no_wake_grant`
	`__nds32__standby_wake_grant`	`STANDBY no_wake_grant`
	`__nds32__standby_wait_done`	`STANDBY no_wake_grant`
Superuser	`__nds32__standby_no_wake_grant`	`STANDBY no_wake_grant`
	`__nds32__standby_wake_grant`	`STANDBY wake_grant`
	`__nds32__standby_wait_done`	`STANDBY wait_done`

Name

`__nds32__sva`

Syntax

`unsigned int __nds32__sva(int a, int b)`

Where parameter “a” and “b” are the two input integer values to be calculated.

Description & Return Value

If adding `a` and `b` results in 32-bit 2’s complement arithmetic overflow, a result of `1` is returned; otherwise, a result of `0` is returned.

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
         int a = 0x7fffffff;
         int b = 1;
         int c;
         c = __nds32__sva(a, b); //c = 1
    }
```

Name

`__nds32__svs`

Syntax

`unsigned int __nds32__svs(int a, int b)`

Where parameter "a" and "b" are the two input integer values to be calculated.

Description & Return Value

If subtracting `a` and `b` results in 32-bit 2's complement arithmetic overflow, a result of `1` is returned; otherwise, a result of `0` is returned.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     int a = 0x7fffffff;
     int b = -1;
     int c;
     c = __nds32__svs(a, b); //c = 1
}
```

Name

`__nds32__syscall`

Syntax

```
void __nds32__syscall(const unsigned int swid)
```

Where:

`swid`	is a 15-bit unsigned constant value.

------------Note------------

The case that `swid` > 32767 is not allowed. If it occurs, compiler would generate an error message of “`the argument swid in __nds32__syscall should be in the range 0-32767`”.

Description

`__nds32__syscall` inserts a SYSCALL instruction into the instruction stream.

Privilege Level: All

Name

`__nds32__teqz`
`__nds32__tnez`
`__nds32__trap`

Syntax

```
void __nds32__teqz(const unsigned int a, const unsigned int swid)
void __nds32__tnez(const unsigned int a, const unsigned int swid)
void __nds32__trap(const unsigned int swid)
```

Where:

parameter "`a`"	is a 32-bit unsigned/unsigned integer variable.
parameter "`swid`"	is a 15-bit constant value.

------------Note------------

The case that `swid` > 32767 is not allowed. If it occurs, compiler would generate an error message of "`the argument swid in __nds32__teqz/__nds32__tnez/__nds32__trap should be in the range 0-32767`".

Description

Both `__nds32__teqz` and `__nds32__tnez` generate a conditional Trap exception while `__nds32__trap` generates an unconditional Trap exception. `__nds32__teqz` generates a Trap exception and transfers control to the Trap exception handler if `a` is equal to `0`; `__nds32__tnez` generates a Trap exception and transfers control to the Trap exception handler if `a` is not equal to `0`. The parameter `swid` is used to distinguish different trap features and usages.

Privilege Level: ALL

Name

__nds32__wsbh

Syntax

`unsigned int __nds32__wsbh(unsigned int a)`

Where parameter “a” is the input variable to be swapped.

Description

The bytes within each halfword of `a` are swapped and the result is returned.

Return Value

The `__nds32__wsbh` intrinsic returns the halfword swapped value of `a`.

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
         unsigned int a = 0x03020100;
         unsigned int b;
         b = __nds32__wsbh(a);
         //b should have a value of 0x02030001
    }
```

12.2.4. Intrinsics for PE1 Instruction

The following table indicates the supported AndesCores for each intrinsic function introduced in this section.

Intrinsic Function	Supported CPUs	Page
__nds32__abs	Only AndesCores with V3 (but not with V3m/V3m+) architecture	152
__nds32__ave	Only AndesCores with V3 (but not with V3m/V3m+) architecture	153
__nds32__bclr	Only AndesCores with V3 (but not with V3m/V3m+) architecture	154
__nds32__bset	Only AndesCores with V3 (but not with V3m/V3m+) architecture	154
__nds32__btgl	Only AndesCores with V3 (but not with V3m/V3m+) architecture	154
__nds32__btst	Only AndesCores with V3 (but not with V3m/V3m+) architecture	154
__nds32__clip	Only AndesCores with V3 (but not with V3m/V3m+) architecture	156
__nds32__clips	Only AndesCores with V3 (but not with V3m/V3m+) architecture	157
__nds32__clz	Only AndesCores with V3 (but not with V3m/V3m+) architecture	159
__nds32__clo	Only AndesCores with V3 (but not with V3m/V3m+) architecture	158

PE1 ISA is configurable. For all AndesCores, the extension bit "`CPU_VER[0]`" indicates if PE1 ISA is supported. If `CPU_VER[0]` is set, PE1 intrinsic functions are supported. Otherwise, PE1 intrinsic functions are not supported. If you use PE1 intrinsic functions with an AndesCore where `CPU_VER[0]` is not set, the core will generate a "Reserved Instruction Exception".

Name

`__nds32__abs`

Syntax

`int __nds32__abs(int a)`

Where parameter “a” is the input integer value to be calculated.

Description

This intrinsic returns the absolute value of a.

Return Value

The `__nds32__abs` intrinsic returns the absolute value of a.

Privilege Level: ALL

Example

```
#include “nds32_intrinsic.h”
void func(void)
{
   int a = -4;
    int abs;
    abs = __nds32__abs(a); //compute the absolute value of a.
}
```

Name

`__nds32__ave`

Syntax

`int __nds32__ave(int a, int b)`

Where parameter “a” and “b” are the two input integer values to be calculated.

Description

This intrinsic returns the average of a and b.

Return Value

The `__nds32__ave` intrinsic returns the average of a and b.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   int a = 4;
    int b = 2;
    int ave;
    ave = __nds32__ave(a, b); //compute the average of a and b.
}
```

Name

`__nds32__bclr`
`__nds32__bset`
`__nds32__btgl`
`__nds32__btst`

Syntax

```
unsigned int __nds32__bclr(unsigned int a, const unsigned int pos)
unsigned int __nds32__bset(unsigned int a, const unsigned int pos)
unsigned int __nds32__btgl(unsigned int a, const unsigned int pos)
unsigned int __nds32__btst(unsigned int a, const unsigned int pos)
```

Where:

Parameter "a"	is the input 32-bit word.
Parameter "pos"	is a 5-bit constant, which specifies the bit position for processing.

------------Note------------

The case that pos > 31 is not allowed. If it occurs, compiler will generate an error message of "the argument pos in __nds32__bclr/__nds32__bset/__nds32__btgl/__nds32__btst should be in the range 0-31".

Description

`__nds32__bclr` clears an individual one bit from the value stored in a.

`__nds32__bset` sets an individual one bit from the value stored in a.

`__nds32__btgl` toggles one bit from the value stored in a.

`__nds32__btst` tests one bit from the value stored in a.

The bit position for these operations is specified by pos. The result is returned.

Return Value

The intrinsics return the processed result from a.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_bclr(void)
{
   …
   unsigned int a = 0xFFFFFFFF;
    a = __nds32__bclr(a, 31); //clear the MSB of a.
   …
}
void func_bset(void)
{
   …
   unsigned int a = 0;
    a = __nds32__bset(a, 31); //set the MSB of a.
   …
}
void func_btgl(void)
{
   …
   unsigned int a = 0x80000000;
    a = __nds32__btgl(a, 31); //toggles the MSB of a.
   …
}
void func_btst(void)
{
   …
   unsigned int a = 0;
    a = __nds32__btst(a, 31); //test the MSB of a. The tested result is 0.
   …
}
```

Name

`__nds32__clip`

Syntax

```
unsigned int __nds32__clip(int a, const unsigned int imm)
```

Where:

Parameter "a"	is the input value.
Parameter "imm"	is a 5-bit constant.

------------Note------------

The case that imm > 31 is not allowed. If it occurs, compiler will generate an error message of "the argument imm in __nds32__clip should be in the range 0-31".

Description

This intrinsic limits the value of a in a range between 2imm-1 and 0 and returns the limited result. For example, if imm is 0, the result should be always 0. If the value of a is negative, the result is 0 as well.

Return Value

The `__nds32__clip` intrinsic returns the clipped result from a.

Privilege Level: ALL

Name

`__nds32__clips`

Syntax

```
int __nds32__clips(int a, const unsigned int imm)
```

Where:

Parameter “`a`”	is the input value.
Parameter “`imm`”	is a 5-bit constant.

------------Note------------

The case that `imm` > 31 is not allowed. If it occurs, compiler will generate an error message of “`the argument imm in __nds32__clips should be in the range 0-31`”.

Description

This intrinsic limits the value of `a` in a range between `2imm-1` and `-2imm` and returns the limited result. For example, if `imm` is `3`, the result should be between `7` and -`8.`

Return Value

The `__nds32__clips` intrinsic returns the clipped result from `a`.

Privilege Level: ALL

Name

`__nds32__clo`

Syntax

```
unsigned int __nds32__clo(unsigned int a)
```

Where parameter “a” is the 32-bit input value.

Description

This intrinsic counts the number of successive ones leading from the most significant bit of `a` and returns the result. For example, if bit 31 of `a` is `0`, the result is `0`. If `a` has a value of `0xFFFFFFFF`, the result should be `32`.

Return Value

The `__nds32__clo` intrinsic returns the leading one counted result.

Privilege Level: ALL

Name

`__nds32__clz`

Syntax

```
unsigned int __nds32__clz(unsigned int a)
```

Where parameter “a” is the 32-bit input value.

Description

This intrinsic counts the number of successive zero leading from the most significant bit of `a` and returns the result. For example, if bit 31 of `a` is `1`, the result is `0`. If `a` has a value of `0`, the result should be `32`.

Return Value

The `__nds32__clz` intrinsic returns the leading zero counted result.

Privilege Level: ALL

12.2.5. Intrinsics for PE2 Instructions

The following table indicates the supported AndesCores for each intrinsic function introduced in this section.

Intrinsic Function	Supported CPUs	Page
`__nds32__bse`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	161
`__nds32__bsp`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	162
`__nds32__pbsad`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	163
`__nds32__pbsada`	Only AndesCores with V3 (but not with V3m/V3m+) architecture	164

PE2 ISA is configurable. For all AndesCores, the extension bit "`CPU_VER[2]`" indicates if PE2 ISA is supported. If `CPU_VER[2]` is set, PE2 intrinsic functions are supported. Otherwise, PE2 intrinsic functions are not supported. If you use PE2 intrinsic functions with an AndesCore where `CPU_VER[2]` is not set, the core will generate a "Reserved Instruction Exception".

Name

`__nds32__bse`

Syntax

```
void __nds32__bse(unsigned int *t, unsigned int a, unsigned int *b)
```

Where:

Parameter "a"	is a 32-bit word to be extracted.
Parameter "*b"	is the extraction configuration variable, which defines the number of bits extracted and the distance between a(31) and the starting MSB bit position of the extracted bits in a.
Parameter "*t"	stores the extraction result.

Description

This intrinsic behaves as a BSE instruction. Since the extraction configuration variable (*b) and the extraction result (*t) are pointers, compiler might generate some extra load/store instructions to load/store the contents of *b and *t. If you have performance concern, use inline assembly instead.

Return Value

None

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
         …
         unsigned int a = 0xF0F0F0F0; //pattern to be extracted.
         unsigned int b = 0x00000300;
         unsigned int r;
        __nds32__bse(&r, a, &b); //extract bit[31-24] of a.
                                  //The value of r becomes 0x0000000F.
                                 //The value of b becomes 0x00000324.
    }
```

Name

__nds32__bsp

Syntax

`void __nds32__bsp(unsigned int *t, unsigned int a, unsigned int *b)`

Where:

Parameter "a"	is a 32-bit word to be inserted.
Parameter "*b"	is the packing configuration variable, which defines the number of bits inserted and the distance between the 31th bit and the starting MSB bit position of the inserted bits in the packed result.
Parameter "*t"	is the packing result.

Description

This intrinsic behaves as a BSP instruction. Since the packing configuration variable (*b) and the packing result (*t) are pointers, compiler might generate some extra load/store instructions to load/store the contents of *b and *t. If you have performance concern, use inline assembly instead.

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     …
     unsigned int a = 0x0000000F; //pattern to be packed.
     unsigned int b = 0x00000300;
     unsigned int r = 0;
     __nds32__bsp(&r, a, &b); //pack bit[7-0] from a to bit[31-24] of r.
                              //The value of r becomes 0xF0000000.
                             //The value of b becomes 0x00000324.
     …
}
```

Name

`__nds32__pbsad`

Syntax

`unsigned int __nds32__pbsad(unsigned int a, unsigned int b)`

Where parameter "a" and "b" are the two 32-bit data to be calculated.

Description

This intrinsic subtracts the four un-signed 8-bit elements of a from the four unsigned 8-bit elements of b. The absolute value of each difference is added together and the result is returned.

Return Value

The `__nds32__pbsad` intrinsic returns the final absolute value.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
    unsigned int a = 0x09070605;
    unsigned int b = 0x04020301;
    unsigned int r;
    r = __nds32__pbsad(a, b); //The value of r becomes 17.
}
```

Name

`__nds32__pbsada`

Syntax

`unsigned int __nds32__pbsada(unsigned int acc, unsigned int a, unsigned int b)`

Where:

Parameter “a” and “b” are two 32-bit data to be calculated.

Parameter “acc” is the accumulation variable.

Description

This intrinsic subtracts the four un-signed 8-bit elements of a from the four unsigned 8-bit elements of b. The absolute value of each difference is added together along with acc and the accumulated result is returned.

Return Value

The `__nds32__pbsada` returns the final accumulated result.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   unsigned int a = 0x09070605;
    unsigned int b = 0x04020301;
    unsigned int r=1;
    r = __nds32__pbsada(r, a, b); //The value of r becomes 18.
}
```

12.2.6. Intrinsics for String

The following table indicates the supported AndesCores for each intrinsic function introduced in this section.

Intrinsic Function	Supported CPUs	Page
__nds32__ffb	Only AndesCores with V3 (but not with V3m/V3m+) architecture	166
__nds32__ffmism	Only AndesCores with V3 (but not with V3m/V3m+) architecture	168
__nds32__flmism	Only AndesCores with V3 (but not with V3m/V3m+) architecture	169

String ISA is configurable. For all AndesCores, the extension bit "CPU_VER[4]" indicates if String ISA is supported. If CPU_VER[4] is set, String intrinsic functions are supported. Otherwise, String intrinsic functions are not supported. If you use String intrinsic functions with an AndesCore where CPU_VER[4] is not set, the core will generate a "Reserved Instruction Exception".

Name

`__nds32__ffb`

Syntax

`int __nds32__ffb(unsigned int a, unsigned int b)`

Where	
Parameter "a"	is the input word.
Parameter "b"	is used to match each byte in parameter "a".

Description

This intrinsic will find the first byte in a that matches b. If b is a constant, the FFBI instruction will be inserted into the instruction stream. If b is a variable, the FFB instruction will be inserted.

------------Note------------

1. If b is a variable, the least significant byte in b is used to match each byte in a.
2. If b is a constant, it is prohibited to have "b > 255." If a violation occurs, compiler will generate an error message of "`the constant type argument b in __nds32__ffb should be in the range 0-255`".

Return Value

The `__nds32__ffb` intrinsic returns the location of the first byte in a that matches b. If a matching byte is found, a non-zero position indication of the first matching byte based on the current data endian (PSW.BE) mode is returned. If no matching byte is found, a zero is returned. Please refer to the FFB/FFBI instruction in *AndeStar Instruction Set Architecture Manual* for the detailed description about the return value.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
    … //assume data endian mode is little endian.
   unsigned int a = 0x1b2a3d4c;
    unsigned int b = 0x0000003d;
    int r;
    r = __nds32__ffb(a, b); //The value of r becomes -3.
}
```

Name

`__nds32__ffmism`

Syntax

`int __nds32__ffmism(unsigned int a, unsigned int b)`

Where parameter “a” and “b” are the two words to be compared.

Description

Each byte in `a` is matched with each corresponding byte in `b`. If any mis-matching byte is found, a non-zero position indication of the first mis-matching byte based on the current data endian (PSW.BE) mode is returned. If no mis-matching byte is found, a zero is returned. Please refer to the `FFMISM` instruction in *AndeStar Instruction Set Architecture Manual* for the detailed description about the return value.

Return Value

The `__nds32__ffmism` intrinsic returns the location of the first byte in `a` that mismatches the corresponding byte in `b`.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   … //assume data endian mode is little endian.
  unsigned int a = 0x1b2a3d4c;
  unsigned int b = 0x112a334c;
   unsigned int r;
   r = __nds32__ ffmism(a, b); //The value of r becomes -3.
}
```

Name

`__nds32__flmism`

Syntax

`int __nds32__flmism(unsigned int a, unsigned int b)`

Where parameter "a" and "b" are the two words to be compared.

Description

Each byte in `a` is matched with each corresponding byte in `b`. If any mis-matching byte is found, a non-zero position indication of the last mis-matching byte based on the current data endian (PSW.BE) mode is returned. If no mis-matching byte is found, a zero is returned. Please refer to the FLMISM instruction in *AndeStar Instruction Set Architecture Manual* for the detailed description about the return value.

Return Value

The `__nds32__ffmism` intrinsic returns the location of the last byte in `a` that mismatches the corresponding byte in `b`.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
    … //assume data endian mode is little endian.
   unsigned int a = 0x1b2a3d4c;
   unsigned int b = 0x112a334c;
    unsigned int r;
    r = __nds32__ flmism(a, b); //The value of r becomes -1.
}
```

12.2.7. Intrinsics for FPU

FPU ISA is configurable. Currently, only N10, N13, N15, D10 and D15 can configure with FPU. FPU intrinsic functions are supported if `FUCOP_EXIST[0]`, `FUCOP_EXIST[31]`, `CPU_VER[4]`, and `FUCOP_CTL[0]` are set. Otherwise, the AndesCore in use will generate a "Reserved Instruction Exception" or a "FPU disabled Exception."

Name

`__nds32__fcpynsd`
`__nds32__fcpynss`
`__nds32__fcpysd`
`__nds32__fcpyss`

Syntax

```
double __nds32__fcpynsd(double a, double b)
float __nds32__fcpynss(float a, float b)
double __nds32__fcpysd(double a, double b)
float __nds32__fcpyss(float a, float b)
```

Where:

Parameter "a"	is the input floating point variable whose value will be copied.
Parameter "b"	is the input floating point variable whose sign will be copied.

Description

Both `__nds32__fcpynsd` and `__nds32__fcpynss` negate and copy the sign of `b` to `a` to form a new value.

Both `__nds32__fcpysd` and `__nds32__fcpyss` copy the sign of `b` to `a` to form a new value.

Return Value

Both `__nds32__fcpynsd` and `__nds32__fcpynss` return the negating and copying result.

Both `__nds32__fcpysd` and `__nds32__fcpyss` return the copying result.

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func_fcpynsd (void)
    {
       double a = -1.5;
        double b = -1.3;
        r = __nds32__fcpynsd(a, b); //The value of r becomes 1.5.
    }
```

```
void func_fcpynss (void)
{
   float a = -1.5;
    float b = -1.3;
    float r;
    r = __nds32__fcpynss(a, b); //The value of r becomes 1.5.
}
```

```
#include <nds32_intrinsic.h>
void func_fcpysd (void)
{
   double a = -1.5;
    double b = 1.3;
    double r;
    r = __nds32__fcpysd(a, b); //The value of r becomes 1.5.
}

void func_fcpyss (void)
{
   float a = -1.5;
    float b = 1.3;
    float r;
    r = __nds32__fcpyss(a, b); //The value of r becomes 1.5.
}
```

Name

`__nds32__fmfcfg`

Syntax

`unsigned int __nds32__fmfcfg()`

Description

This intrinsic reads and returns the content of `FPCFG`.

Return Value

The `__nds32__fmfcsr` intrinsic returns the content of `FPCFG`.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     //this function checks if the SP extension exists.
     unsigned int fpcfg;
     unsigned int sp_exists;
     fpcfg = __nds32__fmfcfg(); //read fpcfg.
     sp_exists = fpcfg & 0x1;
     if (sp_exists)
          printf("SP extension exists\n");
}
```

Name

`__nds32__fmfcsr`

Syntax

`unsigned int __nds32__fmfcsr()`

Description

This intrinsic reads and returns the content of `FPCSR`.

Return Value

The `__nds32__fmfcsr` intrinsic returns the content of `FPCSR`.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
    //this function set FPU to round to zero mode.
   unsigned int fpcsr;
    fpcsr = __nds32__fmfcsr(); //read fpcsr
    fpcsr = (fpcsr & 0xfffffffc) | 3;
    __nds32__fmtcsr(fpcsr); //write fpcsr
}
```

Name

`__nds32__fmtcsr`

Syntax

```
void __nds32__fmtcsr(unsigned int fpcsr)
```

Where:

`fpcsr`	is the value to be transferred to `FPCSR`.

Description

This intrinsic stores the value of `fpcsr` into `FPCSR`.

Privilege Level: ALL

Example

See also `__nds32__fmtcsr`.

12.2.8. Intrinsics for TLBOP

For each intrinsic function in this section, the following table indicates the supported memory management types.

Intrinsic Function	Memory Management Types	Page
__nds32__tlbop_trd	MMU, MPU, SMPU	177
__nds32__tlbop_twr	MMU, MPU, SMPU	178
__nds32__tlbop_rwr	MMU	179
__nds32__tlbop_rwlk	MMU	180
__nds32__tlbop_unlk	MMU	181
__nds32__tlbop_pb	MMU, SMPU	182
__nds32__tlbop_inv	MMU	184
__nds32__tlbop_flua	MMU	185

The memory management types are configurable for all AndesCores. The configuration bits and supported CPUs for each memory management type are listed below:

Memory Management Types	Configuration Bits	Supported CPUs
No management	`MMU_CFG.MMPS = 0`	N6, N7, N8, E8, N9, N10, N13, N15, D10, D15
MMU	`MMU_CFG.MMPS = 2`	N10, N13, N15, D10, D15
MPU	`MMU_CFG.MMPS = 1` and `MMU_CFG.MMPV < 16`	N10, N13, N15, D10, D15
SMPU	`MMU_CFG.MMPS = 1` and `MMU_CFG.MMPV >= 16`	S8

The intrinsic function descriptions in this section assume the AndesCore in use has MMU as its memory management type.

Name

`__nds32__tlbop_trd` (TLB Target Read)

Syntax

`void __nds32__tlbop_trd(unsigned int a)`

where parameter "a" is the TLB entry number to be read.

Description

This intrinsic reads a specified entry in the software-visible portion of the TLB structure. The specified entry is indicated by `a`. The read result is placed in the `TLB_VPN`, `TLB_DATA`, and `TLB_MISC` registers.

The TLB entry number for a non-fully-associative N sets K ways TLB cache is as follows:

31 log2(N*K)	Log2(N*K)-1 log2(N)	Log2(N)-1 0
Ignored	Way number	Set number

Important: Since the `TLB_MISC` register contains the current process's Context ID and Access Page Size information, any use of this intrinsic function is required to save/restore the `TLB_MISC` register if you want the current process to run correctly right after this operation.

Privilege Level: Superuser and above

Exceptions: Privilege Instruction

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     unsigned int rd_num, tlb_out;
     …
  //prepare read entry number.
     …
 __nds32__tlbop_trd(rd_num);    //read TLB.
 __nds32__dsb(); //data serialization barrier.
 tlb_out = __nds32__mfsr(NDS32_SR_TLB_VPN); //move read result to tlb_out.
}
```

Name

`__nds32__tlbop_twr` (TLB Target Write)

Syntax

```
void __nds32__tlbop_twr(unsigned int a)
```

where parameter "a" is the TLB entry number to be written.

Description

This intrinsic writes a specified entry in the software-visible portion of the TLB structure. The entry is indicated by `a`. The other write operands are in the `TLB_VPN`, `TLB_DATA`, and `TLB_MISC` registers.

The TLB entry number for a non-fully-associative N sets K ways TLB cache is as follows:

31 log2(N*K)	Log2(N*K)-1 log2(N)	Log2(N)-1 0
Ignored	Way number	Set number

If the selected target entry is locked, this intrinsic will overwrite the locked entry and clear the locked flag.

Privilege Level: Superuser and above

Exceptions: Privilege Instruction

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     unsigned int w_num;
     …
//prepare write contents into TLB_VPN, TLB_DATA, TLB_MISC.
//prepare write entry number into w_num.
    …
 __nds32__tlbop_twr(rd_num);    //write TLB.
 __nds32__isb();  //inst serialization barrier.
}
```

Name

`__nds32__tlbop_rwr` (TLB Random Write)

Syntax

`void __nds32__tlbop_rwr(unsigned int a)`

where parameter "a" is the data to be written into the `TLB_DATA` portion of the TLB.

Description

This intrinsic writes a hardware-determined random TLB way in a set determined by the VA (in `TLB_VPN`) and page size (in `TLB_MISC`) in the software-visible portion of the TLB structure. The input variable "`a`" specifies the data that will be written into the `TLB_DATA` portion of the TLB structure. The other write operands are in the `TLB_VPN` and `TLB_MISC` registers.

If the ways in the specified set are all locked during the write operation of this instruction, depending on the setting in the `TBALCK` field of the MMU Control system register (`MMU_CTL`), this intrinsic may generate a precise or an imprecise "Data Machine Error" exception. Note that the default value of the `TBALCK` is to generate the exception.

Privilege Level: Superuser and above

Exceptions: Privilege Instruction, Data Machine Error

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
   unsigned int pte_addr;
   …
//TLB_VPN and TLB_MISC has been preset.
//prepare PTE address into pte_addr.
   …
__nds32__tlbop_rwr(pte_addr);   //write TLB.
__nds32__isb();  //inst serialization barrier.
}
```

Name

__nds32__tlbop_rwlk (TLB Random Write and Lock)

Syntax

```
void __nds32__tlbop_rwlk(unsigned int a)
```

where parameter "a" is the data to be written into TLB_DATA portion of the TLB.

Description

Similar to __nds32__tlbop_rwr, this intrinsic writes a hardware-determined random TLB way in a set determined by the VA (in TLB_VPN) and page size (in TLB_MISC) in the software-visible portion of the TLB structure. In addition to the write operation, this intrinsic also locks the TLB entry.

If the ways in the specified set are all locked during the write operation of this instruction, depending on the setting in the TBALCK field of the MMU Control system register (MMU_CTL), this intrinsic may generate a precise or an imprecise "Data Machine Error" exception. Note that the default value of the TBALCK is to generate the exception.

Privilege Level: Superuser and above

Exceptions: Privilege Instruction, Data Machine Error

Name

`__nds32__tlbop_unlk` (TLB Unlock)

Syntax

```
void __nds32__tlbop_unlk(unsigned int a)
```

where parameter “a” is a virtual address.

Description

This intrinsic unlocks a TLB entry if the VA in the input variable “`a`” matches the VPN of a set determined by the VA (in “`a`”) and page size (in `TLB_MISC`).

Privilege Level: Superuser and above

Exceptions: Privilege Instruction

Name

`__nds32__tlbop_pb` (TLB Probe)

Syntax

```
void __nds32__tlbop_pb(unsigned int a)
```

where parameter "`a`" is a virtual address.

Description

This intrinsic searches all TLB structures (software-visible and software-invisible) for a specified VA in the input variable "`a`" and generates an entry number where the VA matches the VPN in that entry. The search result is returned and has the following format:

31	30	29	28 n	n-1 0
NF	HW	SW	Reserved	Entry #

If the VA can be found in the software-visible part of the TLB, the "`sw`" bit will be set. If the VA can be found in the software-invisible part of the TLB, the "`hw`" bit will be set. And if the VA cannot be found in either the software-visible or software-invisible part of the TLB, the "`nf`" bit will be set.

The TLB entry number for the non-fully-associative N sets K ways TLB cache is as follows:

Log2(N*K)-1 log2(N)	Log2(N)-1 0
Way number	Set number

If this instruction encounters a multiple match condition when searching the TLB, a precise "Data Machine Error" exception will be generated.

Privilege Level: Superuser and above

Exceptions: Privilege Instruction

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
         unsigned int pb_va, tlb_ent_num;
         …
     //prepare va into pb_va.
         …
        tlb_ent_num  = __nds32__tlbop_pb(inv_va);//probe TLB entry information
     //examine tlb_ent_num.
    }
```

Name

`__nds32__tlbop_inv` (TLB Invalidate VA)

Syntax

```
void __nds32__tlbop_inv(unsigned int a)
```

where parameter "a" is a virtual address.

Description

This intrinsic function flushes the TLB entry that contains the VA in the input variable "`a`" and the page size specified in the `TLB_MISC` register (software-visible and software-invisible) except the locked TLB entries. The match condition also involves the "G" bit of a PTE entry and the CID field of the `TLB_MISC` register. Their matching logic is as follows:

- If "G" is asserted, CID *is not* part of the match condition.
- If "G" is not asserted, CID *is* part of the match condition.

If this intrinsic encounters a multiple match condition when searching the TLB, all matched entries should be invalidated and no "Data Machine Error" exception will be generated.

Privilege Level: Superuser and above

Exceptions: Privilege Instruction

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     unsigned int inv_va;
     …
 //prepare va into inv_va.
     …
 __nds32__tlbop_inv(inv_va);  //invalidate TLB entries containing unlk_va.
 __nds32__isb();  //inst serialization barrier
}
```

Name

`__nds32__tlbop_flua` (TLB Invalidate All)

Syntax

`void __nds32__tlbop_flua()`

Description

This intrinsic invalidates all TLB entries (software-visible and software-invisible) except the locked TLB entries.

Privilege Level: Superuser and above

Exceptions: Privilege Instruction

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
__nds32__tlbop_flua();   //write TLB.
__nds32__isb();   //inst serialization barrier.
}
```

12.2.9. Intrinsics for Saturation ISA

Saturation ISA is configurable. For all AndesCores, the extension bit “CPU_VER[5]” indicates if Saturation ISA is supported. If CPU_VER[5] is set, Saturation intrinsic functions are supported. Otherwise, Saturation intrinsic functions are not supported. If you use Saturation intrinsic functions with an AndesCore where CPU_VER[5] is not set, the core will generate a “Reserved Instruction Exception.”

Name

`__nds32__kaddw`

Syntax

`int __nds32__kaddw(int a, int b)`

Where parameter “a” and “b” are two input integer values to be calculated.

Description

`__nds32__kaddw` adds the signed variables of `a` and `b` with Q31 saturation.

Return Value

`__nds32__kaddw` returns the calculation results. If saturation occurs, `PSW.OV` will be set.

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
         int a = 0x7fffffff;
         int b = 2;
         int c;
         c = __nds32__kaddw(a, b); //c = 0x7fffffff and PSW.OV will be set.
    }
```

Name

`__nds32__ksubw`

Syntax

`int __nds32__ksubw(int a, int b)`

Where parameter "a" and "b" are two input integer values to be calculated.

Description

`__nds32__ksubw` subtracts signed variables `a` and `b` with Q31 saturation.

Return Value

`__nds32__ksubw` returns the calculation results. If saturation occurs, `PSW.OV` will be set.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
    int a = 0x7fffffff;
    int b = -2;
    int c;
    c = __nds32__ksubw(a, b); //c = 0x7fffffff and PSW.OV will be set.
}
```

Name

__nds32__kaddh

Syntax

int __nds32__kaddh(int a, int b)

Where parameter “a” and “b” are two input integer values to be calculated.

Description

__nds32__kaddh adds signed variables a and b with Q15 saturation.

Return Value

__nds32__kaddh returns the calculation results. If saturation occurs, PSW.OV will be set.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
    int a = 0x7fff;
    int b = 2;
    int c;
    c = __nds32__kaddh(a, b); //c = 0x7fff and PSW.OV will be set.
}
```

Name

`__nds32__ksubh`

Syntax

`int __nds32__ksubh(int a, int b)`

Where parameter "a" and "b" are two input integer values to be calculated.

Description

`__nds32__ksubh` subtracts signed variables `a` and `b` with Q15 saturation.

Return Value

`__nds32__ksubh` returns the calculation results. If saturation occurs, `PSW.OV` will be set.

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
        int a = 0x7fff;
        int b = -2;
        int c;
        c = __nds32__ksubh(a, b); //c = 0x7fff and PSW.OV will be set.
    }
```

Name

__nds32__kdmbb
__nds32__kdmbt
__nds32__kdmtb
__nds32__kdmtt

Syntax

```
int __nds32__kdmbb(unsigned int a, unsigned int b)
int __nds32__kdmbt(unsigned int a, unsigned int b)
int __nds32__kdmtb(unsigned int a, unsigned int b)
int __nds32__kdmtt(unsigned int a, unsigned int b)
```

Where parameter “a” and “b” are two 32-bit input variables to be calculated.

Description

Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the two 32-bit variables (a and b) and then double and saturate the Q31 result. When both Q15 input variables are 0x8000, saturation occurs. In this case, the result will be saturated to 0x7FFFFFFF and PSW.OV will be set.

For the inputs of the multiply operation, __nds32__kdmbb uses the bottom 16-bit Q15 contents of a and b, __nds32__kdmbt uses bottom 16-bit Q15 content of a and top 16-bit Q15 content of b, __nds32__kdmtb uses top 16-bit Q15 content of a and bottom 16-bit Q15 content of b, and __nds32__kdmtt uses the top 16-bit Q15 contents of a and b.

Return Value

These intrinsics return the Q31 result. If saturation occurs, PSW.OV will be set.

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func(void)
    {
         unsigned int a = 0x8000;
         unsigned int b = 0x8000;
         int c;
         c = __nds32__kdmbb(a, b); //c = 0x7fffffff and PSW.OV will be set.
    }
```

Name

__nds32__khmbb
__nds32__khmbt
__nds32__khmtb
__nds32__khmtt

Syntax

```
int __nds32__khmbb(unsigned int a, unsigned int b)
int __nds32__khmbt(unsigned int a, unsigned int b)
int __nds32__khmtb(unsigned int a, unsigned int b)
int __nds32__khmtt(unsigned int a, unsigned int b)
```

Where parameter "a" and "b" are two 32-bit input variables to be calculated.

Description

Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the two 32-bit variables (a and b) and then right-shift 15 bits to turn the Q30 result into a Q15 number and saturate the Q15 number as the return value. When both Q15 input variables are 0x8000, saturation occurs. In this case, the result will be saturated to 0x7FFF and PSW.OV will be set.

For the inputs of the multiply operation, __nds32__khmbb uses the bottom 16-bit Q15 contents of a and b, __nds32__khmbt uses bottom 16-bit Q15 content of a and top 16-bit Q15 content of b, __nds32__khmtb uses top 16-bit Q15 content of a and bottom 16-bit Q15 content of b, and __nds32__khmtt uses the top 16-bit Q15 contents of a and b.

Return Value

These intrinsics return the saturated Q15 result. If saturation occurs, PSW.OV will be set.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     unsigned int a = 0x8000;
     unsigned int b = 0x8000;
     int c;
     c = __nds32__khmbb(a, b); //c = 0x7fff and PSW.OV will be set.
}
```

Name

`__nds32__kslraw`

Syntax

`int __nds32__kslraw(int a, signed char b)`

Where:

Parameter “a” is the input integer to be shifted.

Parameter “b” is the shift amount.

Description

`__nds32__kslraw` performs logical left or arithmetic right shift operation with Q31 saturation. The content of a is left-shifted logically or right-shifted arithmetically based on the value of b. A positive b means logical left shift and a negative b means arithmetic right shift. The shift amount is the absolute value of b. The shifted result is saturated to a Q31 number, mainly for the left-shifted result. If saturation occurs, PSW.OV will be set.

Return Value

`__nds32__kslraw` returns the Q31 result. If saturation occurs, PSW.OV will be set.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     int a = 0x7ffffff0;
     signed char b = 1;
     int c;
     c = __nds32__kslraw(a, b); //c = 0x7fffffff and PSW.OV will be set.
}
```

Name

`__nds32__rdov`

Syntax

`unsigned int __nds32__rdov()`

Description & Return Value

This intrinsic function returns `PSW.OV` bit.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     int a = 0x7ffffff0;
     signed char b = 1;
     int c;
     unsigned int d;
     c = __nds32__kslraw(a, b); //c = 0x7fffffff and PSW.OV will be set.
     __nds32__dsb();
     d = __nds32__rdov(); //d = 1
}
```

Name

`__nds32__clrov`

Syntax

`void __nds32__clrov()`

Description

This intrinsic function clears `PSW.OV`.

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
     int a = 0x7ffffff0;
     signed char b = 1;
     int c;
     unsigned int d, e;
     c = __nds32__kslraw(a, b); //c = 0x7fffffff and PSW.OV will be set.
     __nds32__dsb();
     d = __nds32__rdov(); //d = 1
     __nds32__clrov();
     __nds32__dsb();
     e = __nds32__rdov(); //e = 1
}
```

12.2.10. Intrinsics for Interrupt

The following table indicates the supported AndesCores for each intrinsic function introduced in this section.

Intrinsic Function	Supported CPUs	Page
`__nds32__setgie_dis`	All AndesCores	197
`__nds32__setgie_en`	All AndesCores	197
`__nds32__gie_dis`	All AndesCores	198
`__nds32__gie_en`	All AndesCores	198
`__nds32__enable_int`	All AndesCores	199
`__nds32__disable_int`	All AndesCores	199
`__nds32__set_pending_swint`	All AndesCores	201
`__nds32__clr_pending_swint`	All AndesCores	201
`__nds32__clr_pending_hwint`	All AndesCores	202
`__nds32__get_pending_int`	All AndesCores	204
`__nds32__get_all_pending_int`	All AndesCores	206
`__nds32__set_int_priority`	All AndesCores	207
`__nds32__get_int_priority`	All AndesCores	207
`__nds32__get_trig_type`	All AndesCores	209

Name

__nds32__setgie_dis
__nds32__setgie_en

Syntax

```
void __nds32__setgie_dis()
void __nds32__setgie_en()
```

Description

__nds32__setgie_dis disables global interrupts (won't take effect immediately).

__nds32__setgie_en enables global interrupts (won't take effect immediately).

These two intrinsic functions generate the SETGIE instruction. You need to further use __nds32__dsb to make sure the change in PSW.GIE is seen by the subsequent instruction. Besides PSW.GIE, if you want to modify some other system registers at the same time, these two intrinsic functions will also provide better performance than __nds32__gie_dis and __nds32__gie_en.

Privilege Level: Superuser and above

Example

```
#include <nds32_intrinsic.h>
void func(void)
{
  …
  __nds32__setgie_dis(); //disable global interrupt.
  {other codes to modify system register}
  __nds32__dsb(); //make sure the new PSW.GIE value and the modified SR values
                  are seen by any following instructions.
   …
}
```

Name

`__nds32__gie_dis`
`__nds32__gie_en`

Syntax

```
void __nds32__gie_dis()
void __nds32__gie_en()
```

Description

`__nds32__gie_dis` disables global interrupts (will take effect immediately).

`__nds32__gie_en` enables global interrupts (will take effect immediately).

These two intrinsic functions generate a SETGIE instruction and a DSB instruction. The change in PSW.GIE takes effect immediately.

Privilege Level: Superuser and above

Name

`__nds32__enable_int`
`__nds32__disable_int`

Syntax

```
void __nds32__enable_int(enum nds32_intrinsic int_id)
void __nds32__disable_int(enum nds32_intrinsic int_id)
```

Description

`__nds32__enable_int` enables an interrupt or exception specified by "`int_id`".

`__nds32__disable_int` disables an interrupt or exception specified by "`int_id`".

The change in `INT_MASK` and `INT_MASK2` will be seen by the code after the intrinsic function.

The following table lists all maskable interrupts or exceptions.

Value of "int_id"	Interrupt
NDS32_INT_H0	HW0
NDS32_INT_H1	HW1
NDS32_INT_H2	HW2
NDS32_INT_H3	HW3
NDS32_INT_H4	HW4
NDS32_INT_H5	HW5
NDS32_INT_H6	HW6
NDS32_INT_H7	HW7
NDS32_INT_H8	HW8
NDS32_INT_H9	HW9
NDS32_INT_H10	HW10
NDS32_INT_H11	HW11
NDS32_INT_H12	HW12
NDS32_INT_H13	HW13

Value of "int_id"	Interrupt
NDS32_INT_H14	HW14
NDS32_INT_H15	HW15
NDS32_INT_H16	HW16
NDS32_INT_H17	HW17
NDS32_INT_H18	HW18
NDS32_INT_H19	HW19
NDS32_INT_H20	HW20
NDS32_INT_H21	HW21
NDS32_INT_H22	HW22
NDS32_INT_H23	HW23
NDS32_INT_H24	HW24
NDS32_INT_H25	HW25
NDS32_INT_H26	HW26
NDS32_INT_H27	HW27
NDS32_INT_H28	HW28
NDS32_INT_H29	HW29
NDS32_INT_H30	HW30
NDS32_INT_H31	HW31
NDS32_INT_SWI	Software interrupt
NDS32_INT_ALZ	All zero opcode reserved instruction exception
NDS32_INT_IDIVZE	Arithmetic exception (DIV by 0)
NDS32_INT_DSSIM	Default single stepping interrupt

Privilege Level: Superuser and above

Name

`__nds32__set_pending_swint`
`__nds32__clr_pending_swint`

Syntax

```
void __nds32__set_pending_swint()
void __nds32__clr_pending_swint()
```

Description

`__nds32__set_pending_swint` sets the pending status for the software interrupt (i.e., triggers the software interrupt).

`__nds32__clr_pending_swint` clears the pending status for the software interrupt (i.e., clears the software interrupt).

Note that these two functions are specifically designed for the software interrupt only and no parameter is needed. For HW interrupts, please use `__nds32__clr_pending_hwint(int_id)` instead.

This update of status in `INT_PEND` will be seen by the code after the intrinsic function.

Privilege Level: Superuser and above

Name

`__nds32__clr_pending_hwint`

Syntax

`void __nds32__clr_pending_hwint(enum nds32_intrinsic int_id)`

Description

`__nds32__clr_pending_hwint` clears the pending status of a HW interrupt specified by "`int_id`" (located in `INT_PEND` and `INT_PEND2`). Note that this intrinsic function is designed only to clear **edge-triggered** interrupts. In contrast, for level-triggered interrupts, the interrupt pending status must be cleared from the devices directly and then this new clear status will automatically propagate to the pending status registers. Consequently, there is no need to clear the pending status of level-triggered interrupts.

Also note that you should not use this intrinsic function during normal operation because HW will automatically clear the pending status for you when an edge-triggered interrupt is serviced. Therefore, this intrinsic function is only used to clear pending bits when you initialize or reprogram the interrupt controller and interrupt source devices. This clearance is needed because pending bits can be accidentally set by glitches or noise before proper initialization.

This update of pending status in `INT_PEND` and `INT_PEND2` will be seen by the code after the intrinsic function.

The possible values for "`int_id`" are listed in the following table.

Value of "int_id"	Interrupt
NDS32_INT_H0	HW0
NDS32_INT_H1	HW1
NDS32_INT_H2	HW2
NDS32_INT_H3	HW3
NDS32_INT_H4	HW4
NDS32_INT_H5	HW5
NDS32_INT_H6	HW6

Value of "int_id"	Interrupt
NDS32_INT_H7	HW7
NDS32_INT_H8	HW8
NDS32_INT_H9	HW9
NDS32_INT_H10	HW10
NDS32_INT_H11	HW11
NDS32_INT_H12	HW12
NDS32_INT_H13	HW13
NDS32_INT_H14	HW14
NDS32_INT_H15	HW15
NDS32_INT_H16	HW16
NDS32_INT_H17	HW17
NDS32_INT_H18	HW18
NDS32_INT_H19	HW19
NDS32_INT_H20	HW20
NDS32_INT_H21	HW21
NDS32_INT_H22	HW22
NDS32_INT_H23	HW23
NDS32_INT_H24	HW24
NDS32_INT_H25	HW25
NDS32_INT_H26	HW26
NDS32_INT_H27	HW27
NDS32_INT_H28	HW28
NDS32_INT_H29	HW29
NDS32_INT_H30	HW30
NDS32_INT_H31	HW31

Privilege Level: Superuser and above

Name

__nds32__get_pending_int

Syntax

```
unsigned int __nds32__get_pending_int(enum nds32_intrinsic int_id)
```

Description

__nds32__get_pending_int returns the pending status of the interrupt "int_id" (located in INT_PEND and INT_PEND2).

The possible values for "int_id" are listed in the following table.

Value of "int_id"	Interrupt
NDS32_INT_H0	HW0
NDS32_INT_H1	HW1
NDS32_INT_H2	HW2
NDS32_INT_H3	HW3
NDS32_INT_H4	HW4
NDS32_INT_H5	HW5
NDS32_INT_H6	HW6
NDS32_INT_H7	HW7
NDS32_INT_H8	HW8
NDS32_INT_H9	HW9
NDS32_INT_H10	HW10
NDS32_INT_H11	HW11
NDS32_INT_H12	HW12
NDS32_INT_H13	HW13
NDS32_INT_H14	HW14
NDS32_INT_H15	HW15
NDS32_INT_H16	HW16
NDS32_INT_H17	HW17

Value of "int_id"	Interrupt
NDS32_INT_H18	HW18
NDS32_INT_H19	HW19
NDS32_INT_H20	HW20
NDS32_INT_H21	HW21
NDS32_INT_H22	HW22
NDS32_INT_H23	HW23
NDS32_INT_H24	HW24
NDS32_INT_H25	HW25
NDS32_INT_H26	HW26
NDS32_INT_H27	HW27
NDS32_INT_H28	HW28
NDS32_INT_H29	HW29
NDS32_INT_H30	HW30
NDS32_INT_H31	HW31
NDS32_INT_SWI	Software interrupt

Privilege Level: Superuser and above

Name

`__nds32__get_all_pending_int`

Syntax

```
unsigned int __nds32__get_all_pending_int()
```

Description

`__nds32__get_all_pending_int` is deprecated due to lack of extensibility, so it should not be used. For backward compatibility, it only returns the pending status specified in Interrupt Pending Register (`INT_PEND`), which consists of only the first 16 HW interrupts (0 ~ 15) and a software interrupt.

Privilege Level: Superuser and above

Name

`__nds32__set_int_priority`
`__nds32__get_int_priority`

Syntax

```
void __nds32__set_int_priority(enum nds32_intrinsic int_id, unsigned int prio)
unsigned int __nds32__get_int_priority(enum nds32_intrinsic int_id)
```

Description

`__nds32__set_int_priority` sets the priority of an interrupt specified by “`int_id`”.

`__nds32__get_int_priority` returns the priority of an interrupt specified by “`int_id`”.

The updated priority located in `INT_PRI` and `INT_PRI2` will be seen by the code after the intrinsic function.

The following table lists all programmable interrupts.

Value of “int_id”	Interrupt
NDS32_INT_H0	HW0
NDS32_INT_H1	HW1
NDS32_INT_H2	HW2
NDS32_INT_H3	HW3
NDS32_INT_H4	HW4
NDS32_INT_H5	HW5
NDS32_INT_H6	HW6
NDS32_INT_H7	HW7
NDS32_INT_H8	HW8
NDS32_INT_H9	HW9
NDS32_INT_H10	HW10
NDS32_INT_H11	HW11
NDS32_INT_H12	HW12

Value of "int_id"	Interrupt
NDS32_INT_H13	HW13
NDS32_INT_H14	HW14
NDS32_INT_H15	HW15
NDS32_INT_H16	HW16
NDS32_INT_H17	HW17
NDS32_INT_H18	HW18
NDS32_INT_H19	HW19
NDS32_INT_H20	HW20
NDS32_INT_H21	HW21
NDS32_INT_H22	HW22
NDS32_INT_H23	HW23
NDS32_INT_H24	HW24
NDS32_INT_H25	HW25
NDS32_INT_H26	HW26
NDS32_INT_H27	HW27
NDS32_INT_H28	HW28
NDS32_INT_H29	HW29
NDS32_INT_H30	HW30
NDS32_INT_H31	HW31

Privilege Level: Superuser and above

`__nds32__get_trig_type`

Syntax

```
unsigned int __nds32__get_trig_type(enum nds32_intrinsic int_id)
```

Description

`__nds32__get_trig_type` returns the trigger type of a HW interrupt specified by "`int_id`".

The updated trigger type located in `INT_TRIGGER` will be seen by the code after the intrinsic function.

The following table lists all programmable interrupts.

Value of "int_id"	Interrupt
NDS32_INT_H0	HW0
NDS32_INT_H1	HW1
NDS32_INT_H2	HW2
NDS32_INT_H3	HW3
NDS32_INT_H4	HW4
NDS32_INT_H5	HW5
NDS32_INT_H6	HW6
NDS32_INT_H7	HW7
NDS32_INT_H8	HW8
NDS32_INT_H9	HW9
NDS32_INT_H10	HW10
NDS32_INT_H11	HW11
NDS32_INT_H12	HW12
NDS32_INT_H13	HW13
NDS32_INT_H14	HW14
NDS32_INT_H15	HW15
NDS32_INT_H16	HW16

Value of “int_id”	Interrupt
NDS32_INT_H17	HW17
NDS32_INT_H18	HW18
NDS32_INT_H19	HW19
NDS32_INT_H20	HW20
NDS32_INT_H21	HW21
NDS32_INT_H22	HW22
NDS32_INT_H23	HW23
NDS32_INT_H24	HW24
NDS32_INT_H25	HW25
NDS32_INT_H26	HW26
NDS32_INT_H27	HW27
NDS32_INT_H28	HW28
NDS32_INT_H29	HW29
NDS32_INT_H30	HW30
NDS32_INT_H31	HW31

Privilege Level: Superuser and above

12.2.11. Intrinsics for COP ISA Extension

COP ISA extension is configurable. Currently, only N10, N13 and D10 can configure with COP extension. COP intrinsic functions are supported if `CPU_VER[3]` is set.

Name

```
__nds32__cpe1
__nds32__cpe2
__nds32__cpe3
__nds32__cpe4
```

Syntax

```
void __nds32__cpe1(const unsigned int cpn, const unsigned int cpi19)
void __nds32__cpe2(const unsigned int cpn, const unsigned int cpi19)
void __nds32__cpe3(const unsigned int cpn, const unsigned int cpi19)
void __nds32__cpe4(const unsigned int cpn, const unsigned int cpi19)
```

Where:

Parameter "`cpn`" is the coprocessor number. cpn = {0,1,2,3}

Parameter "`cpi19`" is the 19-bit immediate that carries an encoded coprocessor command.

Description

These instructions send "cpi19" encoded CPE1~CPE4 coprocessor commands to the coprocessor "n" for execution.

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_cpex (void)
{
   //Send CPE 2 command 0x7bcde to coprocessor 1
   __nds32__cpe2(1, 0x7bcde);

   //Send CPE 3 command 0x7bcde to coprocessor 1
   __nds32__cpe3(1, 0x7bcde);
}
```

Name

`__nds32__cpld`
`__nds32__cpld_bi`

Syntax

```
void __nds32__cpld(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, signed int roffset, const unsigned int sv)

void __nds32__cpld_bi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, signed int roffset, const unsigned int sv)
```

Where:

Parameter "`cpn`"	is the coprocessor number. (cpn = {0,1,2,3})
Parameter "`cprn`"	is the ID number of one of the 32 coprocessor registers that receives 64-bit loaded data from the memory. (0 <= cprn <= 31).
Parameter "`base`"	is the GPR number that contains the base address of this instruction.
Parameter "`roffset`"	is the GPR number that contains the signed offset address of this instruction.
Parameter "`sv`"	is the left shift amount for offset addressing. (sv = {0,1,2,3})

Description

`__nds32__cpld` uses the calculated address of "R[base]+(R[roffset] << sv)" to load a 64-bit datum into the coprocessor register "cprn".

`__nds32__cpld_bi` uses the address of R[base] to load a 64-bit datum into the coprocessor register "cprn", and then updates R[base] with the calculated value of "R[base]+(R[roffset] << sv)".

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_cpld (void)
{
   unsigned long long *base;
   unsigned int roffset;
   //Load 64-bit data from address “base+(roffset<<2)” into
   // coprocessor 1 register 3
   __nds32__cpld(1, 3, base, roffset, 2);

   //Load 64-bit data from address “base” into coprocessor 1 register 3
   // Update “base” register with “base+(roffset<<2)”
   __nds32__cpld_bi(1, 3, base, roffset, 2);

}
```

Name

__nds32__cpldi
__nds32__cpldi_bi

Syntax

```
void __nds32__cpldi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, const signed int imm12)
```

```
void __nds32__cpldi_bi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, const signed int imm12)
```

Where:

Parameter "cpn"	is the coprocessor number. (cpn = {0,1,2,3})
Parameter "cprn"	is the ID number of one of the 32 coprocessor registers that receives 64-bit loaded data from the memory. (0 <= cprn <= 31).
Parameter "base"	is the GPR number that contains the base address of this instruction.
Parameter "imm12"	is the 12-bit immediate signed offset address of this instruction.

Description

__nds32__cpldi uses the calculated address of "R[base]+SignExtend(imm12)" to load a 64-bit datum into the coprocessor register "cprn".

__nds32__cpldi_bi uses the address of R[base] to load a 64-bit datum into the coprocessor register "cprn", and then updates R[base] with the calculated value of "R[base]+SignExtend(imm12)".

Return Value

None

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func_cpldi (void)
```

```
{
   unsigned long long *base;
   //Load 64-bit data from address “base+SignExtend(0x450)” into
   // coprocessor 1 register 3
   __nds32__cpldi(1, 3, base, 0x450);

   //Load 64-bit data from address “base” into coprocessor 1 register 3
   // Update “base” register with “base+(0x450)”
   __nds32__cpldi_bi(1, 3, base, 0x450);

}
```

Name

`__nds32__cplw`

`__nds32__cplw_bi`

Syntax

```
void __nds32__cplw(const unsigned int cpn, const unsigned int cprn, unsigned int
*base, signed int roffset, const unsigned int sv)
```

```
void __nds32__cplw_bi(const unsigned int cpn, const unsigned int cprn, unsigned
int *base, signed int roffset, const unsigned int sv)
```

Where:

Parameter "`cpn`" is the coprocessor number. (cpn = {0,1,2,3})

Parameter "`cprn`" is the ID number of one of the 32 coprocessor registers that receive 32-bit loaded data from the memory. (0 <= cprn <= 31.)

Parameter "`base`" is the GPR number that contains the base address of this instruction.

Parameter "`roffset`" is the GPR number that contains the signed offset address of this instruction.

Parameter "`sv`" is the left shift amount for offset addressing. (sv = {0,1,2,3})

Description

`__nds32__cplw` uses the calculated address of "R[base]+(R[roffset] << sv)" to load a 32-bit datum into the coprocessor register "cprn".

`__nds32__cplw_bi` uses the address of R[base] to load a 32-bit datum into the coprocessor register "cprn", and then updates R[base] with the calculated value of "R[base]+(R[roffset] << sv)".

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_cplw (void)
{
   unsigned int *base;
   unsigned int roffset;
   //Load 32-bit data from address “base+(roffset<<2)” into
   // coprocessor 1 register 3
   __nds32__cplw(1, 3, base, roffset, 2);

   //Load 32-bit data from address “base” into coprocessor 1 register 3
   // Update “base” register with “base+(roffset<<2)”
   __nds32__cplw_bi(1, 3, base, roffset, 2);

}
```

Name

__nds32__cplwi
__nds32__cplwi_bi

Syntax

```
void __nds32__cplwi(const unsigned int cpn, const unsigned int cprn, unsigned int
*base, const signed int imm12)

void __nds32__cplwi_bi(const unsigned int cpn, const unsigned int cprn, unsigned
int *base, const signed int imm12)
```

Where:

Parameter “cpn” is the coprocessor number. (cpn = {0,1,2,3})

Parameter “cprn” is the ID number of one of the 32 coprocessor registers that receives 32-bit loaded data from the memory. (0 <= cprn <= 31)

Parameter “base” is the GPR number that contains the base address of this instruction.

Parameter “imm12 is the 12-bit immediate signed offset address of this instruction.

Description

__nds32__cplwi uses the calculated address of “R[base]+SignExtend(imm12)” to load a 32-bit datum into the coprocessor register “cprn”.

__nds32__cplwi_bi uses the address of R[base] to load a 32-bit datum into the coprocessor register “cprn”, and then updates R[base] with the calculated value of “R[base]+SignExtend(imm12)”.

Return Value

None

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func_cplwi (void)
```

```
{
   unsigned int *base;
   //Load 32-bit data from address “base+SignExtend(0x450)” into
   // coprocessor 1 register 3
   __nds32__cplwi(1, 3, base, 0x450);

   //Load 32-bit data from address “base” into coprocessor 1 register 3
   // Update “base” register with “base+(0x450)”
   __nds32__cplwi_bi(1, 3, base, 0x450);

}
```

Name

`__nds32__cpsd`
`__nds32__cpsd_bi`

Syntax

```
void __nds32__cpsd(const unsigned int cpn, const unsigned int cprn, unsigned long
long *base, signed int roffset, const unsigned int sv)
```

```
void __nds32__cpsd_bi(const unsigned int cpn, const unsigned int cprn, unsigned
long long *base, signed int roffset, const unsigned int sv)
```

Where:

Parameter “`cpn`” is the coprocessor number. (cpn = {0,1,2,3})

Parameter “`cprn`” is the ID number of one of the 32 coprocessor registers that provides 64-bit stored data to the memory. (0 <= cprn <= 31)

Parameter “`base`” is the GPR number that contains the base address of this instruction.

Parameter “`roffset`” is the GPR number that contains the signed offset address of this instruction.

Parameter “`sv`” is the left shift amount for offset addressing. (sv = {0,1,2,3})

Description

`__nds32__cpsd` uses the calculated address of “R[base]+(R[roffset] << sv)” to store a 64-bit datum from the coprocessor register “cprn” into the memory.

`__nds32__cpsd_bi` uses the address of R[base] to store a 64-bit datum from the coprocessor register “cprn” into the memory, and then updates R[base] with the calculated value of “R[base]+(R[roffset] << sv)”.

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_cpsd (void)
{
   unsigned long long *base;
   unsigned int roffset;
   //Store 64-bit data to address “base+(roffset<<2)” from
   // coprocessor 1 register 3
   __nds32__cpsd(1, 3, base, roffset, 2);

   //Load 64-bit data to address “base” from coprocessor 1 register 3
   // Update “base” register with “base+(roffset<<2)”
   __nds32__cpsd_bi(1, 3, base, roffset, 2);

}
```

Name

__nds32__cpsdi

__nds32__cpsdi_bi

Syntax

void __nds32__cpsdi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, const signed int imm12)

void __nds32__cpsdi_bi(const unsigned int cpn, const unsigned int cprn, unsigned long long *base, const signed int imm12)

Where:

Parameter "cpn" is the coprocessor number. (cpn = {0,1,2,3})

Parameter "cprn" is the ID number of one of the 32 coprocessor registers that provides 64-bit stored data to the memory. (0 <= cprn <= 31).

Parameter "base" is the GPR number that contains the base address of this instruction.

Parameter "imm12" is the 12-bit immediate signed offset address of this instruction.

Description

__nds32__cpsdi uses the calculated address of "R[base]+SignExtend(imm12)" to store a 64-bit datum from the coprocessor register "cprn" into the memory.

__nds32__cpsdi_bi uses the address of R[base] to store a 64-bit datum from the coprocessor register "cprn" into the memory, and then updates R[base] with the calculated value of "R[base]+SignExtend(imm12)".

Return Value

None

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func_cpsdi (void)
```

```
{
   unsigned long long *base;
   //Store 64-bit data to address “base+SignExtend(0x450)” from
   // coprocessor 1 register 3
   __nds32__cpsdi(1, 3, base, 0x450);

   //Store 64-bit data to address “base” from coprocessor 1 register 3.
   // Update “base” register with “base+(0x450)”
   __nds32__cpsdi_bi(1, 3, base, 0x450);

}
```

Name

`__nds32__cpsw`
`__nds32__cpsw_bi`

Syntax

```
void __nds32__cpsw(const unsigned int cpn, const unsigned int cprn, unsigned int
*base, signed int roffset, const unsigned int sv)

void __nds32__cpsw_bi(const unsigned int cpn, const unsigned int cprn, unsigned
int *base, signed int roffset, const unsigned int sv)
```

Where:

Parameter “`cpn`” is the coprocessor number. (cpn = {0,1,2,3})

Parameter “`cprn`” is the ID number of one of the 32 coprocessor registers that provides 32-bit stored data to the memory. (0 <= cprn <= 31)

Parameter “`base`” is the GPR number that contains the base address of this instruction.

Parameter “`roffset`” is the GPR number that contains the signed offset address of this instruction.

Parameter “`sv`” is the left shift amount for offset addressing. (sv = {0,1,2,3})

Description

`__nds32__cpsw` uses the calculated address of “R[base]+(R[roffset] << sv)” to store a 32-bit datum from the coprocessor register “cprn” into the memory.

`__nds32__cpsw_bi` uses the address of R[base] to store a 32-bit datum from the coprocessor register “cprn” into the memory, and then updates R[base] with the calculated value of “R[base]+(R[roffset] << sv)”.

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_cpsw (void)
{
   unsigned int *base;
   unsigned int roffset;
   //Store 32-bit data to address “base+(roffset<<2)” from
   // coprocessor 1 register 3
   __nds32__cpsw(1, 3, base, roffset, 2);

   //Load 32-bit data to address “base” from coprocessor 1 register 3
   // Update “base” register with “base+(roffset<<2)”
   __nds32__cpsw_bi(1, 3, base, roffset, 2);

}
```

Name

`__nds32__cpswi`
`__nds32__cpswi_bi`

Syntax

```
void __nds32__cpswi(const unsigned int cpn, const unsigned int cprn, unsigned int
*base, const signed int imm12)
```

```
void __nds32__cpswi_bi(const unsigned int cpn, const unsigned int cprn, unsigned
int *base, const signed int imm12)
```

Where:

Parameter "`cpn`" is the coprocessor number. (cpn = {0,1,2,3})

Parameter "`cprn`" is the ID number of one of the 32 coprocessor registers that provides 32-bit stored data to the memory. (0 <= cprn <= 31)

Parameter "`base`" is the GPR number that contains the base address of this instruction.

Parameter "`imm12`" is the 12-bit immediate signed offset address of this instruction.

Description

`__nds32__cpswi` uses the calculated address of "R[base]+SignExtend(imm12)" to store a 32-bit datum from the coprocessor register "cprn" into the memory.

`__nds32__cpswi_bi` uses the address of R[base] to store a 32-bit datum from the coprocessor register "cprn" into the memory, and then updates R[base] with the calculated value of "R[base]+SignExtend(imm12)".

Return Value

None

Privilege Level: ALL

Example

```
    #include <nds32_intrinsic.h>
    void func_cpswi (void)
```

```
{
   unsigned int *base;
   //Store 32-bit data to address “base+SignExtend(0x450)” from
   // coprocessor 1 register 3
   __nds32__cpswi(1, 3, base, 0x450);

   //Store 32-bit data to address “base” from coprocessor 1 register 3.
   // Update “base” register with “base+(0x450)”
   __nds32__cpswi_bi(1, 3, base, 0x450);

}
```

Name

__nds32__mfcpd

Syntax

`unsigned long long __nds32__mfcpd(const unsigned int cpn, const unsigned int imm12)`

Where:

Parameter "cpn" is the coprocessor number. (cpn = {0,1,2,3})

Parameter "imm12" is the 12-bit immediate value that encodes the 64-bit coprocessor state space.

Description

__nds32__mfcpd moves a 64-bit datum from the 64-bit coprocessor state space "imm12" into an even/odd pair of two 32-bit GPRs.

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_mfcpd (void)
{
   unsigned long long data64;
   //Move 64-bit data from coprocessor 1 64-bit state space 10 into two GPRs
   data64 = __nds32__mfcpd(1, 10);

}
```

Name

__nds32__mfcpw

Syntax

`unsigned int __nds32__mfcpw(const unsigned int cpn, const unsigned int imm12)`

Where:

Parameter "cpn" is the coprocessor number. (cpn = {0,1,2,3})

Parameter "imm12" is the 12-bit immediate value that encodes the 32-bit coprocessor state space.

Description

__nds32__mfcpw moves a 32-bit datum from the 32-bit coprocessor state space "imm12" into a 32-bit GPR.

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_mfcpw (void)
{
   unsigned int data32;
   //Move 32-bit data from coprocessor 1 32-bit state space 10 into a GPR
   data32 = __nds32__mfcpw(1, 10);

}
```

Name

`__nds32__mfcppw`

Syntax

```
unsigned int __nds32__mfcppw(const unsigned int cpn, const unsigned int imm12)
```

Where:

Parameter “`cpn`” is the coprocessor number. (cpn = {0,1,2,3})

Parameter “`imm12`” is the 12-bit immediate value that encodes the 32-bit coprocessor state space.

Description

`__nds32__mfcppw` moves a 32-bit datum from the 32-bit coprocessor privileged state space “imm12” into a 32-bit GPR.

Return Value

None

Privilege Level: Superuser and above

Example

```
#include <nds32_intrinsic.h>
void func_mfcppw (void)
{
    unsigned int data32;
    //Move 32-bit data from coprocessor 1 32-bit privileged state space 10
    // into a GPR
    data32 = __nds32__mfcppw(1, 10);

}
```

Name

`__nds32__mtcpd`

Syntax

`void __nds32__mtcpd(const unsigned int cpn, unsigned long long source, const unsigned int imm12)`

Where:

Parameter "`cpn`"	is the coprocessor number. cpn = {0,1,2,3}
Parameter "`source`"	a 64-bit datum stored in an even/odd pair of two 32-bit GPRs.
Parameter "`imm12`"	is the 12-bit immediate value that encodes the 64-bit coprocessor state space.

Description

`__nds32__mtcpd` moves a 64-bit datum to the 64-bit coprocessor state space "imm12" from an even/odd pair of two 32-bit GPRs.

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_mtcpd (void)
{
   unsigned long long data64;
   //Move 64-bit data to coprocessor 1 64-bit state space 10 from two GPRs
   __nds32__mtcpd(1, data64, 10);

}
```

Name

`__nds32__mtcpw`

Syntax

`void __nds32__mtcpw(const unsigned int cpn, unsigned int source, const unsigned int imm12)`

Where:

Parameter "`cpn`"	is the coprocessor number. (cpn = {0,1,2,3})
Parameter "`source`"	is a 32-bit datum stored in a 32-bit GPR.
Parameter "`imm12`"	is the 12-bit immediate value that encodes the 32-bit coprocessor state space.

Description

`__nds32__mtcpw` moves a 32-bit datum to the 32-bit coprocessor state space "imm12" from a 32-bit GPR.

Return Value

None

Privilege Level: ALL

Example

```
#include <nds32_intrinsic.h>
void func_mtcpw (void)
{
    unsigned int data32;
    //Move 32-bit data to coprocessor 1 32-bit state space 10 from a GPR
    __nds32__mtcpw(1, data32, 10);

}
```

Name

`__nds32__mtcppw`

Syntax

`void __nds32__mtcppw(const unsigned int cpn, unsigned int source, const unsigned int imm12)`

Where:

Parameter “`cpn`”	is the coprocessor number. (cpn = {0,1,2,3})
Parameter “`source`”	is a 32-bit datum stored in a 32-bit GPR.
Parameter “`imm12`”	is the 12-bit immediate value that encodes the 32-bit coprocessor privileged state space.

Description

`__nds32__mtcpw` moves a 32-bit datum to the 32-bit coprocessor privileged state space “imm12” from a 32-bit GPR.

Return Value

None

Privilege Level: Superuser and above

Example

```
#include <nds32_intrinsic.h>
void func_mtcppw (void)
{
   unsigned int data32;
   //Move 32-bit data to coprocessor 1 32-bit privileged state space 10
   // from a GPR
   __nds32__mtcppw(1, data32, 10);

}
```

13. User/Kernel Space

In general, programs can be written for user-space or kernel-space applications. Any instruction available to user-space programs is always available to kernel-space programs. On the other hand, however, instructions available to kernel-space programs are only available to user-space in a restricted way, and instructions designed to allow user-space programs accessing resources are only visible to kernel-space programs.

13.1. Privilege Resources

In general, privilege resources refer to the system registers which can only be visible to kernel-space programs. Please refer to *AndeStar System Privilege Architecture Manual* in the package for detailed information.

13.1.1. Configuration System Registers

These system registers are hardwired when hardware configurations are determined before the hardware is manufactured. Thus, they are read-only registers.

13.1.2. Interruption System Registers

These system registers are properly set when an interruption occurs. Thus, they should be read-only registers and the updates on these registers must be performed carefully.

13.1.3. MMU System Registers

These system registers are all related to MMU and paging functions. Thus, they should be only used when MMU is hardware configured and under a full-blown operating system such as Linux.

13.1.4. ICE System Registers

These system registers are all related to debugging, especially when using ICE.

13.1.5. Performance Monitoring Registers

These system registers are all related to performance monitoring capability of Andes Architecture. Normally, they are accessed by the service routines of the underlying operating system.

13.1.6. Local Memory DMA Registers

These system registers are all related to instruction and data local memory of Andes Architecture when hardware is configured. Normally, they are accessed by the service routines of the underlying operating system.

13.1.7. Implementation-Dependent Registers

These system registers are reserved for use by an implementation. Their uses change from an implementation generation to the next implementation generation. Some implementations may not use all of them. Please refer to *AndeStar System Privilege Architecture Manual* in the package for details.

13.2. Privilege Resource Access Instructions

Please refer to *AndeStar System Privilege Architecture Manual* for the detailed information.

13.2.1. Read from/Write to System Registers

Table 21. Accessing System Registers

Mnemonic	Instruction	Operation
`MFSR   rt5, SRIDX`	Move from System Register	`rt5 = SR[SRIDX]`
`MTSR   rt5, SRIDX`	Move to System Register	`SR[SRIDX] = rt5`

13.2.2. Jump Register with System Register Update

Table 22. Instruction Translation On/Off

Mnemonic	Instruction	Operation
`JR.ITOFF  rb5`	Jump Register and Instruction Translation OFF	`PC = rb5;` `PSW.IT = 0;`
`JR.TOFF  rb5`	Jump Register and Translation OFF	`PC = rb5;` `PSW.IT = 0, PSW.DT = 0;`
`JRAL.ITON   rb5` `JRAL.ITON   rt5, rb5`	Jump Register and Link and Instruction Translation ON	`jaddr = rb5;` `LP = PC+4 or rt5 = PC+4;` `PC = jaddr;` `PSW.IT = 1;`
`JRAL.TON   rb5` `JRAL.TON   rt5, rb5`	Jump Register and Link and Translation ON	`jaddr = rb5;` `LP = PC+4 or rt5 = PC+4;` `PC = jaddr;` `PSW.IT = 1, PSW.DT = 1;`

13.2.3. MMU Instructions

Table 23. TLBOP Subtypes

Mnemonic	**Instruction**	**Operation**
`TLBOP Ra, TargetRead (TRD)`	Read targeted TLB entry	Read a specified entry in the software-visible portion of the TLB structure.
`TLBOP Ra, TargetWrite (TWR)`	Write targeted TLB entry	Write a specified entry in the software-visible portion of the TLB structure.
`TLBOP Ra, RWrite (RWR)`	Write PTE into a TLB entry	Write a hardware-determined random TLB way in a set determined by the VA (in TLB_VPN) and page size (in TLB_MISC) in the software-visible portion of the TLB structure.
`TLBOP Ra, RWriteLock (RWLK)`	Write PTE into a TLB entry and lock	Write a hardware-determined random TLB way in a set determined by the VA (in TLB_VPN) and page size (in TLB_MISC) in the software-visible portion of the TLB structure. Besides, it also locks the TLB entry.
`TLBOP Ra, Unlock (UNLK)`	Unlock a TLB entry	Unlock a TLB entry if the VA in the general register Ra matches the VPN of a set determined by the VA (in Ra) and page size (in TLB_MISC).

Mnemonic	Instruction	Operation
TLBOP Rt, Ra, Probe (PB)	Probe TLB entry	Search all TLB structures (software-visible and software-invisible) for a specified VA and generate an entry number where the VA matches the VPN in that entry.
TLBOP Ra, Invalidate (INV)	Invalidate TLB entries	Invalidate the TLB entry containing VA stored in Rx.
TLBOP FlushAll (FLUA)	Flush all TLB entries except locked entries	
LD_VLPT	Load VLPT page table (optional instruction)	Load VLPT page table which always goes through data TLB translation. On TLB miss, generate Double TLB miss exception.

13.3. Privileged Instructions

In general, privileged instructions refer to the instructions that can only be used by kernel-space programs. Accordingly, those listed in section 13.2 are all privileged instructions. Please refer to *AndeStar System Privilege Architecture Manual* for more information about privileged instructions.

13.3.1. IRET: Interruption Return

This instruction is used to return from interruption to the instruction and a state when the processor is being interrupted.

13.3.2. SETGIE.E/SETGIU.D: Set Global Interruption Enable

This instruction is used to control the global interrupt enable bit in the PSW register.

13.3.3. CCTL: Cache Control

This instruction is used to perform various operations on processor caches. Not all of them are available to user-space programs. Please refer to section 13.4 below for corresponding restrictions.

13.3.4. STANDBY: Wait for External Event

This instruction is used for a core to enter a standby state while waiting for the occurrence of external events. Users have to specify the SubType (`wake_grant`/`no_wake_grant`/`wait_done`) based on their needs.

13.4. Instructions for User-space Program to Access System Resources

In general, instructions for user-space program to access system resources refer to the instructions that can be used by user-space programs to perform tasks normally required kernel privilege. Please refer to *AndeStar System Privilege Architecture Manual* for more information.

13.4.1. DPREF/DPREFI: Data Prefetch

These instructions are used as hints to move data from memory to data cache in advance before the actual load or store operations reduce memory access latency.

13.4.2. SETEND.B/SETEND.L: Set Data Endian

These instructions are used to control the data endian mode in the PSW register.

13.4.3. CCTL: Cache Control

This instruction is used to perform various operations on processor caches. Only the following sub-types are available for user-space programs:

Table 24. CCTL Subtypes

Mnemonic	Instruction
L1D_VA_INVAL	Invalidate L1D cache through VA
L1D_VA_WB	Write-back L1D cache through VA
L1D_VA_WBINVAL	Write-back & invalidate L1D cache through VA
L1I_VA_INVAL	Invalidate L1I cache through VA

13.4.4. ISB/DSB: Data/Instruction Serialization Barrier

`ISB/DSB` are used to serialize pipeline hazards for certain hardware state updates affecting instruction execution. Section 13.5 discusses serializations related to CPU Control Register Accesses. There is also serialization related to Cache Control instructions (`CCTL`). For example, there is a hazard from `CCTL Instruction Cache Invalidate` to the subsequent instruction fetch. Similarly, there is a hazard from `CCTL Data Cache Invalidate` to the subsequent load/store instructions. Please consult *AndeStar Instruction Set Architecture Manual* for serialization behavior in the sections for the related instructions.

13.4.5. STANDBY: Wait for External Event

This instruction is used for a core to enter a standby state while waiting for the occurrence of external events. Then encoded `wake_grant` operand shall be ignored as if `no_wake_grant` is specified.

13.5. Serializations Related to CPU Control Register Accesses

CPU Control Registers (CCRs) include System Registers, User-Special Registers, and Coprocessor Control Registers. Certain CCRs have special control bits, which interact with some instructions. Those are called **CCR-related pipeline hazards**. In such cases, ISB or DSB may be needed to ensure that the program results are committed according to the sequential order. Here are the general occasions to **use serialization instructions when accessing CCRs**:

- After the instruction writing a CCR (such as MTSR), ISB must be inserted if the CCR is a register with side-effect to the following instructions.
- Before the instruction reading a CCR (such as MFSR), DSB must be inserted if the CCR contains the state as a result of executing the preceding instructions.

CCRs are not accessed frequently, but they need to be accessed to achieve some special control purposes in either user code or kernel code. Please consult *AndeStar System Privilege Architecture Manual* for the CCRs of interest and their related pipeline hazards. Here are some examples:

- Changing the data endian in Program Status Word register $PSW, followed by load/store instructions.
- Changing Interruption Vector Base in $IVB, followed by instructions generating exceptions.
- Changing Instruction Local Memory (ILM) Base Address in $ILMB, followed by an instruction jumping to ILM.
- Changing Data Local Memory (DLM) Base Address in $DLMB, followed by load/store instructions targeting DLM.
- Saturation instructions generating overflow followed by reading of $PSW.
- Changing Instruction Table Base $ITB, followed by ex9.it instructions.

Here are general notes for code snippet involving accesses to CCRs:

- If it is in **assembly code**, determine if the **instructions** used have any pipeline hazard related to the CCRs in question and insert ISB/DSB as appropriate.
- If it is in **C code**, determine if **C operations** have any pipeline hazard related to the CCRs in question and insert ISB/DSB as appropriate.

A special note for C code, -Os and some older V3 CPUs (including N968A with CPU Version<=9, N1068A with CPU Version<=8, and N1337 with CPU Version<=8):

When a C file is compiled with size optimization (i.e. `-Os`) using a toolchain for the above V3 family CPUs and `$PSW` is accessed through inline assembly or intrinsic functions, `ISB`/`DSB` must be inserted as appropriate. This is because some special instructions generated by `–Os` optimization may cause $PSW-related pipeline hazard for these CPUs.

Newer V3 CPUs take care of pipeline hazard directly in hardware. So, software programmers can expect sequential program behavior without using `ISB`/`DSB` even when accessing `$PSW`.

14. Linking/Loading

This chapter introduces two supported linking forms: static linking and dynamic linking.

14.1. Static Linking

The `-static` option will force the link editor to link against static version of C runtime libraries such as `libc.a` and `libm.a`. By default, the link editor will use shared version of C runtime libraries, such as `Libc.so` and `Libm.so`, unless `-static` option is used.

14.2. Dynamic Linking

This is the default linking mode performed by link editor. Dynamic linking has many advantages over static linking, such as

1. It produces smaller executables, which consume less storage and memory spaces.
2. Shared Libraries used by executables are upgradable at later time without relinking.
3. Loading and unloading shared libraries are possible at runtime.

However, it also has disadvantages over static linking, such as

1. It starts and runs slower.
2. Shared version of library is bigger than its static counterpart.

14.3. Guidelines to Decide Linking Mode

1. For systems without Linux/shared Libraries, use static linking only.
2. For complicated systems that have many executables, use dynamic linking to save storage and memory spaces.
3. To make your system upgradable after release, use dynamic linking.
4. To maximize performance or profiling, use static linking.
5. For simple systems with little executables, use static linking to save storage and memory spaces.

15. 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`).

15.1. 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.

15.1.1. 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”.

Symbol	Description
A+	Element A can have one or more occurrences. Thus, A+ can be A, AA , or AAA….
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.

15.1.2. Formal Syntax of SaG Format

15.1.2.1 Overview

The SaG-formatted script is constructed by the hierarchy of load regions, execution regions and input sections. To start with, define a script as one or more load_region_description patterns:

```
ld_script ::=
[header] load_region_description+

header ::=
((“USER_SECTIONS” section_name+)*
| (“DEFINE” variable_name expression)*)
| (“INCLUDE” “file_name”)*)
```

Note that 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 declare it in header. Otherwise, LdSaG (nds_ldsag) will show a warning message after compiling. In header syntax, USER_SECTIONS is a keyword and must be upper-cased. The following gives an example: If you define a section .my_section in the assembly file –

```
.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 is another form of header syntax. It is also a key word and must be upper-cased. You can use it to define a local variable and its value. As for expression, it is like c language expression, such as:

```
A + B
```

```
    A + 10
    10 +10
```

INCLUDE, the last form of header syntax, is a key word too and must be upper-cased. You can use it to include other linker script in the generated script. Note that file_name must be double-quoted as follows:

```
    INCLUDE "second.ld"
```

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+
    "}"
```

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)+
    "}"
```

Last, 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, or a VAR variable.

```
    input_section_description ::=
    (module_select_pattern [input_attr] "(" input_section_selector ( ","
     input_section_selector )* ")"
     | ADDR variable
     | LOADADDR variable
     | STACK "=" num
     | VAR variable "=" num
    )
```

15.1.2.2 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`	can be a decimal or hexadecimal number.
`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 or hexadecimal number. If the target object size is bigger than the value, it will report error in linking time.
`exe_region_description`	Please refer to Section 15.1.2.3.
`exe_overlay_region_description`	Please refer to Section 15.1.2.5.

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
```

15.1.2.3 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`	can be a decimal or hexadecimal number.
`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 or hexadecimal number. If the target object size is bigger than the value, it will report error in linking time.
`input_section_description`	Please refer to Section 15.1.2.4.

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
```

15.1.2.4 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
 | VAR variable “=” expression
 | variable “=” ALIGN “(“num”)”
)
```

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 at lease 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.
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.

- `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 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 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`	`ALIGN` sets a variable to the location counter aligned to the

`“(“num”)”`	next alignment boundary. If the variable name is “.” , it adjusts the location counter to the next alignment boundary.

Example

```
● program1.o KEEP (.text, +RO)     ; the output section will include the
  ; program1.0’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)
    ```
  - 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)
    ```

15.1.2.5 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`	can be a decimal or hexadecimal number.
`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`	Please refer to Section 15.1.2.6.

15.1.2.6 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 15.1.2.4.
`input_attr`	is the same as `input_attr` in Section 15.1.2.4.
`input_section_selector`	is the same as `input_section_selector` in Section 15.1.2.4.

15.1.2.7 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
   }
}
```

15.2. 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:
      -t FILE_NAME      //Read the template file, for advanced users only
                        //The default template file is nds32_template.x in
                        //Linux and nds32_template.txt in Windows
      -o FILE_NAME      //Output a file with the specified file-name
```

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.

Note that nds_ldsag may not support Cygwin path format since it is built by MinGW toolchain. Thus, DOS path format is recommended if you have to use an absolute path. For example,

```
nd_ldsag
C:/Andestech/AndeSight/ide/workspace/hello_world/test.sag –o
C:/Andestech/AndeSight/ide/workspace/hello_world/myldscript
```

Step 3 Use the newly-generated linker script to compile an object.

```
nds32le-elf-gcc -Wl,-T,myldscript hello.c -o a.out
```

16. Object Files

16.1. 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.

Please refer to *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification* for more details.

16.2. Examine ELF file

The following tools can be used to examine ELF files:

1. `nds32le-elf-readelf` displays all kind of information in an ELF file.
2. `nds32le-elf-objdump` disassembles instructions or dumps section data.

Please refer to the GNU Binutils document for more details.

Here is a partial listing generated by the command line “`nds32le-elf-readelf –a libc.a`”

```
File: 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:                           Andes Technology compact code size embedded RISC processor family
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          176 (bytes into file)
  Flags:                             0x30000042, AABI, Andes ELF V1.4, Andes Star v3.0
  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: 7

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  1
  [ 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 000006 00  AX  0   0  2
  [ 5] .rela.text._Exit  RELA            00000000 0002f8 000018 0c      8   4  4
  [ 6] .comment          PROGBITS        00000000 00003a 00002f 01  MS  0   0  1
  [ 7] .shstrtab         STRTAB          00000000 000069 000046 00      0   0  1
  [ 8] .symtab           SYMTAB          00000000 000240 0000a0 10      9   8  4
  [ 9] .strtab           STRTAB          00000000 0002e0 000018 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

There are no program headers in this file.
Relocation section '.rela.text._Exit' at offset 0x2f8 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name + Addend
00000000  000005c0 R_NDS32_RELAX_ENT 00000000   .text._Exit + 3000000c
00000002  00000919 R_NDS32_25_PCREL_ 00000000   _exit + 0

The decoding of unwind sections for machine type Andes Technology compact code size embedded
RISC processor family 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 NOTYPE  LOCAL  DEFAULT    4 $c
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000     6 FUNC    GLOBAL DEFAULT    4 _Exit
     9: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _exit

No version information found in this file.
```

17. Andes MCUlib

17.1. Features of MCUlib

While Newlib toolchains are used to build for performance and better integration compatibility with other software packages, MCUlib toolchains are recommended when aiming to build for better code size. Unlike Newlib, MCUlib doesn't support reentrancy and has its own printf implementation. The following section introduces MCUlib-specific printf implementation.

17.2. MCUlib 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 long (64-bit) int
	`h`	short (16-bit) int
	`l`	long (32-bit) int

conversion	d,i	decimal int
	u	decimal unsigned
	o	octal
	x,X	hex
	f,e,g,F,E,G	float
	c	char
	s	string
	p	pointer

Return Value

total number of characters output

Note

1. Normally compiler will use printf() to handle the parameter list of printf() except for the case that if the parameter list of printf() contains only format string, GCC compiler will translate it to puts().
2. For any target platform, the lower-level function of printf must be implemented in order to actually output printf message. In Andes evaluation board, it is done in libgloss with syscall mechanism. For users' own target boards, one of the following can be done:

   (1) Rewrite "putchar()" function to ensure the message can output to the users' boards: A step recommended for MCUlib since it is efficient and can produce the smallest code size. The prototype of putchar() in MCUlib is the same as that in standard C library. Note that for MCUlib from BSP v3.1.2 and later versions, both nds_write() and putchar() must be used to output printf message. The implementation of nds_write() is as follows:

```
void nds_write(const unsigned char *buf, int size)
{
    int   i;
    for (i = 0; i < size; i++)
        putchar(buf[i]);
}
```

NOTE: In addition to printf implementation, `nds_write()` also can be used to avoid errors when users use MCUlib and specify "`-nostartfiles`" option.

(2) Rewrite "`_fstat()`" and "`_write()`" function of libgloss: A step that works for MCUlib from BSP v3.1.0 and earlier versions and Newlib. It provides a syscall mechanism rather than function call for printf lower layer function implementation. `_fstat()` will be called before `_write()` and its implementation is as follows:

```
struct stat;
int _fstat(int fd, struct stat *buf)
{
    return 0;
}
```

The prototype of `_write()` is shown below and it's declared in `unistd.h`. Users have to handle all necessary jobs (for example, to handle outputs to files or STDERR) in their own `_write()` function.

```
int _write(int __fd, const void *__buf, int __nbyte);
```

The figure below illustrates the complete printf implementation on users' boards in comparison to that on Andes evaluation board. The parts in red fonts denote where need users' implementation.

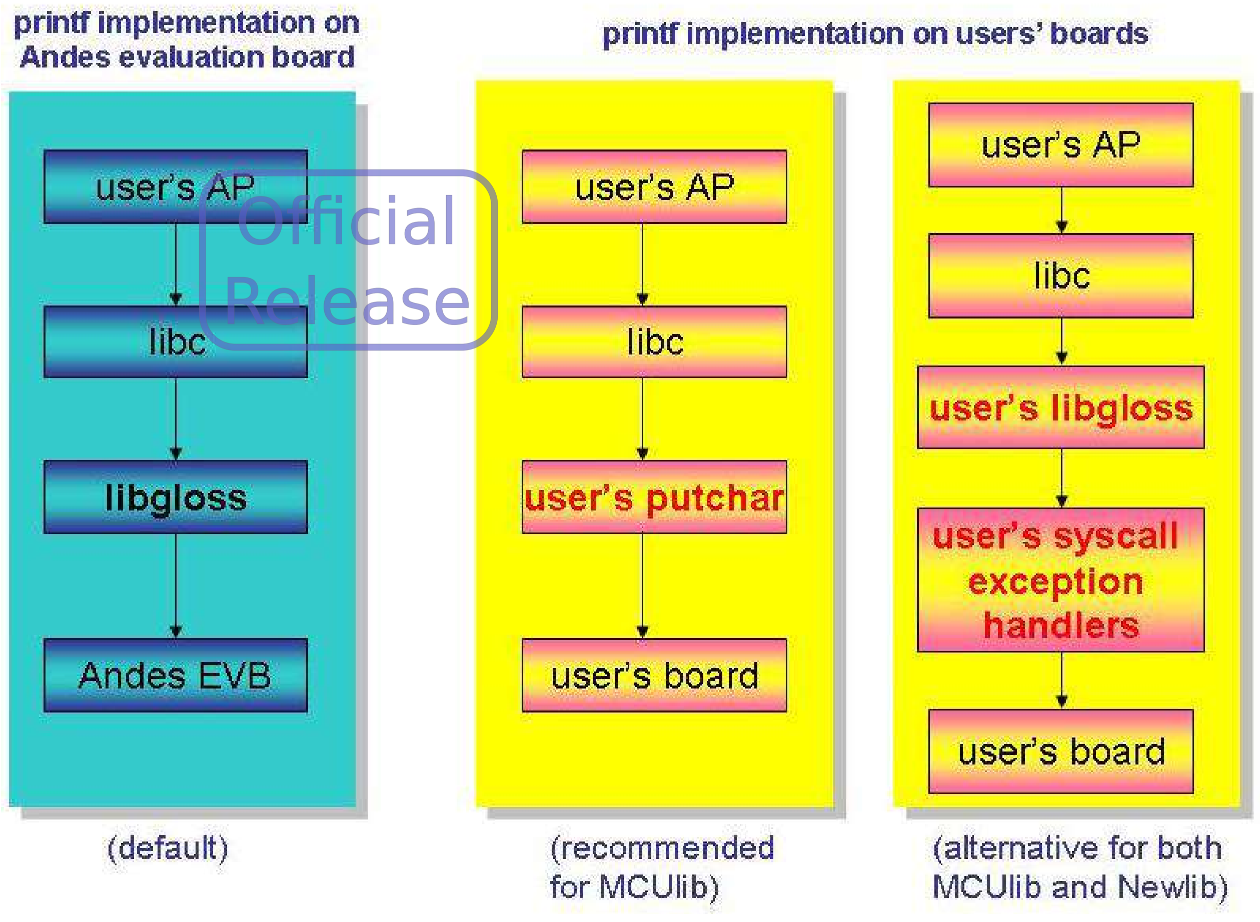

printf implementation on Andes evaluation board
printf implementation on users' boards
user's AP
libc
libgloss
Andes EVB
user's AP
libc
user's putchar
user's board
user's AP
libc
user's libgloss
user's syscall exception handlers
user's board
(default)
(recommended for MCUlib)
(alternative for both MCUlib and Newlib)

Figure 9. printf Implementation on Andes Evaluation Board and on Users' Boards

17.3. Building Libgloss

Step 1 Extract libgloss.tgz under the path **BSP_ROOT**/demo/:

```
$ tar -zxvf libgloss.tgz
```

Find libgloss-nds32-src folder generated under the same directory. It includes the following items: a Makefile, a README and a libgloss-nds32 folder containing libgloss source code files.

Step 2 Include the appropriate toolchain in environment variable PATH.

```
$ export PATH=$PATH:/BSP_ROOT/toolchains/TOOLCHAIN/bin
```

Step 3 When building libgloss for the first time, please skip this step. Otherwise, remove the existing object files and libgloss.a in the current directory.

```
$ make clean
```

Step 4 Build libgloss and generate object files.

```
$ make all
```

18. 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, testing code coverage (gcov) has to write the code coverage data to files. By Virtual Hosting, it still can be supported on target boards that don't have I/O devices.

Virtual Hosting is only supported for V3-family MCUlib and Newlib toolchains (including v3, v3j, v3f, v3s, v3m and v3m+ toolchains). In BSP v3.2, Virtual Hosting is implemented in ICEman. Starting from BSP v4.0, a more generic method is used to support Virtual Hosting on both real boards (ICEman) and the simulator.

To enable Virtual Hosting in BSP v4.0 and later versions, please add "`-mvh`" option when invoking GCC 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 following are low-level I/O functions supported by the current Virtual Hosting:

- `exit`
- `open`
- `close`
- `read`
- `write`
- `lseek`
- `unlink`
- `fstat`
- `stat`
- `gettimeofday`
- `rename`
- `isatty`
- `system`

These I/O functions may be interfered 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:

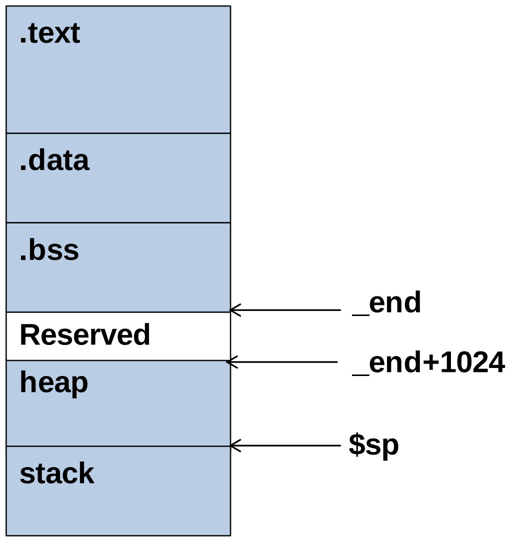
1. If Virtual Hosting is enabled, avoid redirecting the output with `putchar()` in MCUlib or `_write()` in Newlib.
2. Two functions of ANSI C library, `_malloc_r()` and `_free_r()`, may be called automatically when Virtual Hosting is enabled. In MCUlib, if the library memory allocation functions are not suitable for your application, you should implement your own `_malloc_r()` and `_free_r()`; in Newlib, you have to implement the two functions with `_realloc_r()`.

   `_malloc_r()`, `_free_r()`, and `_realloc_r()` are the reentrant variants of `malloc()`, `free()`, and `realloc()`. The prototypes of these functions are:

   ```
    void *_malloc_r(struct _reent *reent_ptr, size_t size);
    void _free_r(struct _reent *reent_ptr, void *ptr);
    void *_realloc_r(struct _reent *reent_ptr, void *ptr, size_t size);
   ```

   If your functions don't need the reentrancy, you can skip the `_reent_ptr` parameter and implement these functions just as `malloc()`, `free()`, and `realloc()`.

   These memory allocation functions dynamically allocate and free memory from the heap. In Andes library implementation, the heap extends from `(_end + 1024)` until `$sp`.



19. Advanced Programming Optimization

With Andes toolchains, you can use different coding tips to make specialized optimizations for Andes architecture. This chapter introduces some GCC compiler options to enable optimization, EX9 optimization and IFC optimization, coding preferences (such as data type "int" and auto/local variable) and coding techniques for special purposes (such as instruction "max" and "min," function with variable arguments and inline assembly language).

19.1. Optimization Options

There are lots of GCC compiler options that deal with optimizations. Here are some common options and Andes GCC compiler options to control different sorts of optimizations.

19.1.1. Options for Code Size Optimization

- Compiler Options
  `-Os`

Sometimes the code size optimizations may degrade the performance. Therefore, for V3 family toolchains, three levels of code size reduction are also supported: `-Os1`, `-Os2` and `-Os3`. Table 25 below provides detailed descriptions for the three levels.

Table 25. Three Code Size Optimization Levels of –Os

Option	**Code Size Optimization Level**
`-Os1`	Enable minimum code size optimizations. Performance is still concerned.
`-Os2`	Enable partial code size optimizations with little performance concern.
`-Os3 (-Os)`	Same as `-Os` option. Enable all code size optimizations. Performance may seriously drop.

19.1.2. Options for Code Speed Optimization

- Compiler Options

  `-O3`
  `-funroll-loops`
  `-funroll-all-loops`
  `-ftree-switch-shortcut`
  `-malign-functions`
  `-malways-align`

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 callee within the caller (See Table 27 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 the platform with cache enabled. Therefore, please refer to the descriptions in Table 26 and use these options wisely to meet your requirement.

Table 26. Two Loop Unrolling Optimization

Option	Description
`-funroll-loops`	Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. Compiler has a set of heuristics to estimate whether to unroll loop or not.
`-funroll-all-loops`	Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This option probably makes programs run more slowly if it loses locality after unrolling.

3. For `-ftree-switch-shortcut`, this is an EXPERIMENTAL option. For some particular benchmarks involving complex switch statements, this option may be useful to improve performance.

4. `-malign-functions` aligns function entries to 4-byte boundaries and `-malways-align` enforces 4-byte alignment on jump targets, return addresses and function entries. The two options are to prevent extra performance penalty due to misalignment. They are not default applied at `-Os` (including `-Os1`, `-Os2` and `-Os3`) since they may slightly increase code size. However, they are enabled by default at most of other optimization levels (see Table 27).

19.1.3. 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.

19.1.4. Options to Use EX9 Optimization

The "`ex9`" instruction can be used at link time optimization. To apply EX9 optimization, the following compiler and linker options have to be enabled at the same time:

- Compiler Option
  `-mex9`

- Linker Options
  (gcc as linker) `-Wl,--mex9`
  (ld as linker) `--mex9`

Notice that `-Os` enables these options by default. If you do not want to apply EX9 optimization at link time, use "`-Wl,--mno-ex9`" (gcc as linker) or "`--mno-ex9`" (ld as linker) to disable it.

19.1.5. Options to Use IFC Optimization

The "`ifcall`", "`ifcall9`" and "`ifret16`" instructions can be used at link time optimization. To apply IFC optimization, the following compiler and linker options have to be enabled at the same time:

- Compiler Option
  `-mifc`
- Linker Options
  (gcc as linker) `-Wl,--mifc`
  (ld as linker) `--mifc`

Notice that `-Os` enables these options by default. If you do not want to apply IFC optimization at link time, use "`-Wl,--mno-ifc`" (gcc as linker) or "`--mno-ifc`" (ld as linker) to disable it.

19.1.6. Notice on Some Optimization Options

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 compiler not to optimize out null pointer checking.

- `-fno-strict-aliasing`

  In the GCC compiler framework, it enables strict aliasing optimization at -Os, -O2, and -O3, assuming 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.

19.1.7. 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 2.2.1) to tune performance and code size. Some target options have been enabled at certain optimization levels by default. Please refer to Table 27 below for their default applied scenarios:

Table 27. Default Applied GCC 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`					√			
`-mrelax`	√	√	√	√	√	√	√	√
`-malign-functions`		√	√	√	√			
`-malways-align`		√	√	√	√			
`-minnermost-loop`							√	
`-mex9`								√
`-mifc`								√

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-relax`, `-mno-align-functions`, `-mno-always-align`, `-mno-innermost-loop`, `-mno-ex9`, and `-mno-ifc` can avoid the options in Table 27 from being enabled at their respective "default applied optimization levels."

Among the options in Table 27, `-mrelax`, `-mex9`, and `-mifc` are special cases for code generation. They do not actually change assembly code but generate directives to mark optimization candidates for linker. GCC then will pass `--relax,` `--mex9,` and `--mifc` to linker to guide it physically perform particular optimizations. If you ONLY invoke GCC to compile programs into an object file, these three options have no effect on code generation.

Note that Table 27 describes the option applied scenarios for BSP v4.0 and later versions. For toolchains from BSP v3.2, these scenarios are mostly the same except for the followings:

1. The option "`-fno-delete-null-pointer-checks`" is not supported in BSP v3.2.
2. In BSPv3.2, some additional options are enabled by default at certain optimization levels, as shown below:

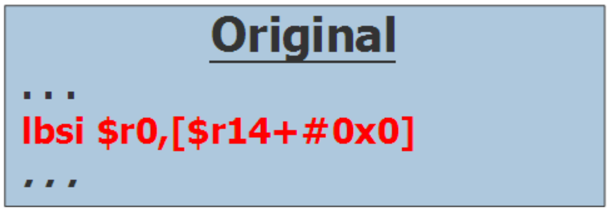
Optimization levels	**Default-applied Options (in addition to those in Table 27)**
`-O3`	`-fno-function-cse`
`-Os1`	`-fno-jump-tables`
`-Os2`	`-fno-jump-tables`

`-Os/-Os3`	`-fno-function-cse`
	`-fno-jump-tables`
	`-fno-inline-small-functions`
	`-fno-schedule-insns`

19.2. EX9 Optimization

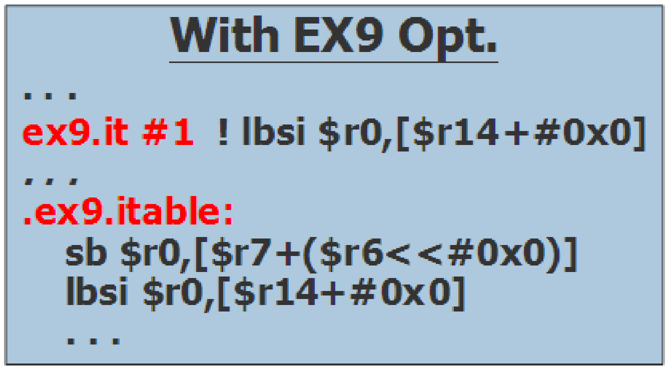
The 16-bit instruction EX9.IT (Execution on Instruction Table) fetches an indexed instruction from the 512-entry Instruction Table and executes it.

When the "`-mex9`" option is applied, the compiler will generate the EX9 table and replaces suitable 32-bit instructions with the 16-bit "`ex9.it <INDEX>`" with `<INDEX>` pointing to the corresponding 32-bit instruction. For example:



Original

```
...
lbsi $r0,[$r14+#0x0]
...
```



With EX9 Opt.

```
...
ex9.it #1  ! lbsi $r0,[$r14+#0x0]
...
.ex9.itable:
   sb $r0,[$r7+($r6<<#0x0)]
   lbsi $r0,[$r14+#0x0]
   ...
```

NOTE: For v3/3j/v3s/v3f toolchains before BSP v4.0.0, the EX9 table with only one entry is still generated even when the "`–mex9`" option isn't applied. This is for the backward compatibility issue for debuggers. This overhead has been removed since BSP v4.0.0.

There are two choices for EX9 table implementation:

1. Hardwired in the CPU RTL with no cycle penalty.
2. Residing in memory pointed to by `$ITB` register for flexibility (2-cycle penalty).

   ```
   EX9.IT:
     If (Hardwired IT) {
       Inst = Instruction_Table[imm9u];
     }else{
       Addr= IT_Base + (index * 4);
       Inst= fetch(Addr);
     }
   Execute(Inst);
   ```

   If the EX9 table resides in memory, `$ITB` must be initialized with the symbol `_ITB_BASE_` before the EX9 table is used. This action should be done in crt0.S. Please reference Section 9.3 for details.

The EX9 table should also be placed correctly in the linker script file by putting the following line after RO code:

```
KEEP(*( .ex9.itable))
```

However, if the linker script file is generated by the LdSaG utility, you will not need to do anything.

19.2.1. Export and Import

The Ex9 table can be exported by a linked module and used by another separately-linked module. This is useful when doing ROM patch. "-Wl, -mexport-ex9" and "-Wl, -import-ex9" options are used to do export and import. For example,

```
nds32le-elf-gcc main_program.c -o main_program.out -mex9
  -Wl,--mexport-ex9=ex9.table
nds32le-elf-gcc rom_patch.c -o rom_patch.out -mex9
  -Wl,--mimport-ex9=ex9.table
```

rom_patch will use the EX9 table generated when compiling main_program.

19.2.2. EX9 Table Shared by Multiple Separately-linked Program Modules

A more advance usage of EX9 optimization is sharing EX9 table by multiple separately-linked modules. "-Wl,--mupdate-ex9" option is used to update the imported EX9 table and "-Wl,--mex9-limit" option can limit the number of EX9 entries used by one module. For example, if there is a library containing common functions shared by app-1 and app-2, the following commands can share EX9 table among lib, app-1, and app-2.

```
nds32le-elf-gcc lib.c -o lib.out --mex9 -Wl,--mgen-symbol-ld-script=lib.ld
  -Wl,--mexport-ex9=ex9.itable -Wl,--mex9-limit=100
nds32le-elf-gcc lib.out app-1.c -o app-1.out --mex9 -Wl,-T,lib.ld
  -Wl,--mimport-ex9=ex9.itable --mupdate-ex9 -Wl,--mex9-limit=200
nds32le-elf-gcc lib.out app-2.c -o app-2.out --mex9 -Wl,-T,lib.ld
  -Wl,--mimport-ex9=ex9.itable -Wl,--mupdate-ex9 -Wl,--mex9-limit=200
```

If the compiler can find the instructions to translate `ex9.it` more than the limit of lib(100), app-1(200), and app-2(200), 1-100 entries is used by lib, 101-300 entries is used by app-1, and 301-500 entries is 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) entries.

19.3. IFC (Inline Function Call) Optimization

IFCall9 (16b), IFCall (32b) and IFRet16 (16b) instructions are used to share the common code sequence as inline functions.

IFC_CTL is the USR register with 2 fields:

`IFC_LP` records the PC of the instructions after IFCall9/IFCall

`IFC_ON` is set when IFCall9/IFCall is executed and cleared on IFRet16

IFCall9/IFCall:

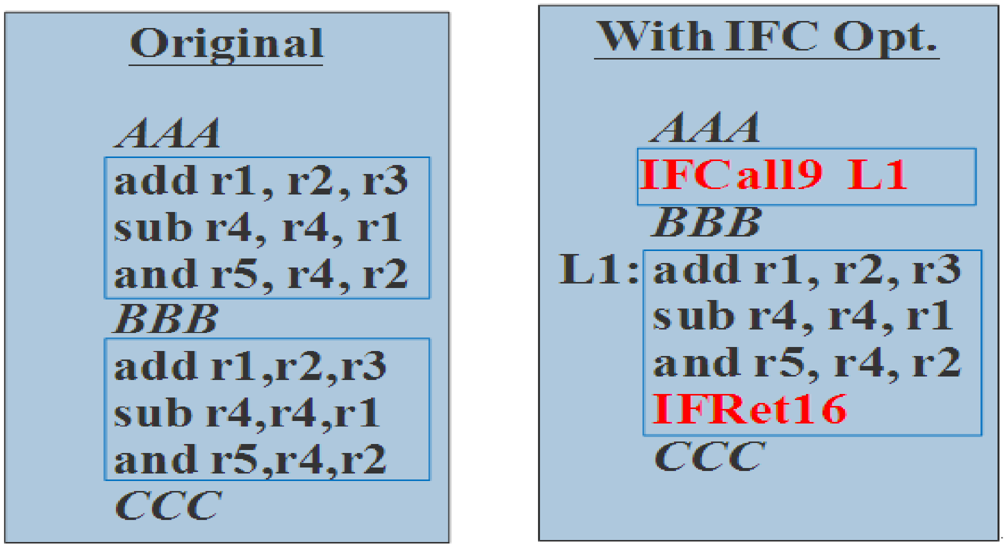
behave as a jump-and-link

```
IFC_LP= return address;
IFC_ON= 1;
```

IFret16:

```
If (IFC_ON) {
   Jump to IFC_LP;
   IFC_ON= 0;
}else{
   Do nothing
}
```

For example:



Original

```
AAA
add r1, r2, r3
sub r4, r4, r1
and r5, r4, r2
BBB
add r1,r2,r3
sub r4,r4,r1
and r5,r4,r2
CCC
```

With IFC Opt.

```
    AAA
    IFCall9  L1
    BBB
L1: add r1, r2, r3
    sub r4, r4, r1
    and r5, r4, r2
    IFRet16
    CCC
```

NOTE:

1. IFCall is a pc-relative instruction, so the distance between caller and callee must be within its branch range, +-16M. Otherwise, it may cause error.
2. `IFC_LP` should be correctly saved and restored in interrupt handlers and the context switching.

19.4. Zero Overhead Loop Optimization

ZOL (Zero Overhead Loop) is a set of mechanism in Andes DSP ISA extension to speed up performance of loops. Rather than exploiting an explicit branch instruction, it improves the loop performance by setting up the loop starting address, the loop ending address and the loop count number.

You can use the compiler option "-mext-zol" to generate code with zero overhead loops. For example, given a function "foo" like below,

```
void foo(int size, int *arr)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = i;
}
```

Its compilation results without and with -mext-zol are listed respectively as follows:

When compiled with	-O	-O -mext-zol

Compilation result	`foo:`
	`    blez   $r0, .L2`
	`    movi55 $r5, 0`
	`.L3:`
	`    swi333.bi  $r5, [$r1], 4`
	`    addi45 $r5, 1`
	`    bnes38  $r0, .L3`
	`    .align 2`
	`.L2:`
	`    ret5`
		`foo:`
		`    blez   $r0, .L2`
		`    movi55 $r2, 0`
		`    sub45  $r0, $r2`
		`    mtlbi   .L3`
		`    mtlei   .L5`
		`    mtusr  $r0, LC`
		`    isb`
		`.L3:`
		`    swi333.bi  $r2, [$r1], 4`
		`.L5:`
		`    addi45 $r2, 1`
		`    .align 2`
		`.L2:`
		`    ret5`

Shown in the above table, the compilation result with ZOL saves a conditional branch, which is a saving of 2~3 cycles per iteration (assuming `swi333.bi` and `addi45` is one cycle and `bnes38` is two cycles) and a performance gain up to 200%.

19.4.1. Zero Overhead Loop Optimization Limitations

Both the hardware and compiler have limitations on performing the zero overhead loop optimization. From the hardware side, Andes architecture doesn't allow nested zero overhead loops. For a function containing a nested loop like below, the hardware can only perform the zero overhead loop optimization on one loop, either the outer or the nested, while the compiler prefers it on the outer for minimizing initialization overhead.

```
void bar(int size1, int size2, int **arr, int val)
{
  int i, j;
  for (i = 0;i < size1;i++) // Outer Loop
    for (j = 0;j < size2;j++) // Nested Loop
      arr[i][j] = val;

}
```

On the other side, the compiler doesn't have enough information about whether the inner function uses hardware loops or not. Thus, the loops for the ZOL optimization must contain function calls that can be inline. The following code fragment, then, won't allow the ZOL optimization.

```
int bar(int n);
void foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = bar (val);

}
```

19.4.2. Disable ZOL Optimization for Specific Functions or Loops

Even though the zero overhead loop optimization significantly increases the performance of loops in most cases, it incurs initialization cost of at least 4 extra instructions and 5~10 cycles, varied by architecture. That is, not every loop can benefit from this optimization. Since the compiler doesn't have enough runtime information about the number of iteration and `-mext-zol` is a global flag to the compilation unit (i.e. single file), a function attribute "`no_ext_zol`" and a built-in function "`__nds32__no_ext_zol`" are introduced here to disable the ZOL optimization for a specific function and loop respectively.

The function attribute "`no_ext_zol`" can disable the ZOL optimization for specific functions when the compilation flag `-mext-zol` is applied. See the following example for its usage:

```
int foo(int, int *, int) __attribute__((no_ext_zol));
int foo(int size, int *arr, int val)
{
  int i;
  for (i = 0;i < size;i++)
    arr[i] = val;
}
```

The function "`__nds32__no_ext_zol`" can disable the ZOL optimization for specific loops. The following is an example that the compiler tends to perform the ZOL optimization on the outer loop but that on the inner loop is more profitable. In this case, the function "`__nds32__no_ext_zol`" can be used to disable the ZOL optimization for the outer loop.

```
#include "nds32_intrinsic.h"

void bar(int size1, int size2, int **arr, int val)
{
  int i, j;
  for (i = 0;i < size1;i++)
  {
        __nds32__no_ext_zol ();
        for (j = 0;j < size2;j++)
          arr[i][j] = val;
      }
    }
```

For a function that has two loops but only one loop can gain from the ZOL optimization, you can use "__nds32__no_ext_zol" to disable the ZOL optimization for a loop too, as exemplified below:

```
    #include "nds32_intrinsic.h"

    void foo(int size, int *arr, int val)
    {
      int i;

      for (i = 0;i < size;i++)
        arr[i] = val;

      for (i = 0;i < size/2;i++)
      {
        __nds32__no_ext_zol ();
        arr[i] = arr[i] + 3;
      }
    }
```

19.5. Instruction Max/Min of Performance Extension

AndeStar ISA performance extension offers instruction "max" and "min" to write maximum and minimum values from source registers to destination registers. Andes GCC takes advantage of the two instructions to generate optimized code for better speed and code size. To evoke "max" and "min" instructions, use ternary operators in the following formats:

```
c = (a > b) ? a : b; // generate instruction max; same for (a>=b)
c = (a < b) ? a : b; // generate instruction min; same for (a<=b)
```

Example-max-min-1 demonstrates the use of ternary operators to evoke instruction “max” and “min”:

```
/* Example-max-min-1  */

int func_max_min_1 (int i, int j, int k, int l)
{
  int max = (i > j) ? i : j;
  int min = (k <= l) ? k : l;

  return max + min;
}
```

Example-max-min-1 will be compiled with the compiler option "-O1" to the following assembly code if Andes GCC is configured to use instructions of performance extension:

```
func_max_min_1:
   ! begin of prologue
   ! end of prologue
   max     $r0, $r1, $r0
   min     $r2, $r2, $r3
   add45   $r0, $r2
   ! begin of epilogue
   ret5
   ! end of epilogue
```

19.6. Primitive Data Type "int"

Since most instructions are designed for 32-bit operands in 32-bit CPU architecture, it is usually better to declare a variable a 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 declaring a variable a type less than 32 bits.

```
/* 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:
   ! begin of prologue
   ! end of prologue
   zeh33  $r0, $r0
   zeh33  $r1, $r1
   sub333 $r2, $r0, $r1
   add5   $r0, $r1
   seh33  $r2, $r2
   seh33  $r0, $r0
   slts45 $r0, $r2
   movi55 $r0, 13
   movpi45 $r1, 17
   cmovz  $r0, $r1, $ta
```

```
    ! begin of epilogue
    ret5
    ! end of epilogue
```

Since the variables "e1, e2" are declared as type "short" in Example-type-1, the instruction "seh33" is required to extend the effective bits of a register to 32 bits so that it can serve as a 32-bit operand for instructions "slts45" and "cmovz".

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 generated assembly code below shows that no extra instruction is needed to adjust the property of variables "e1, e2" for instructions "slts45" and "cmovz".

```
func_type_2:
    ! begin of prologue
    ! end of prologue
    sub333 $r2, $r0, $r1
    add45  $r0, $r1
    slts45 $r0, $r2
    movi55 $r0, 13
    movpi45$r1, 17
    cmovz  $r0, $r1, $ta
    ! begin of epilogue
    ret5
    ! end of epilogue
```

19.7. 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 CPU with 32-bit addressing space, accessing a global variable requires several instructions to construct full 32-bit address. Similar issues also appear on function call. To call a module all over 32-bit addressing space, many instructions are also needed to calculate 32-bit address and then jump to the module via a register.

Instructions that always construct full 32-bit address could be serious issue on performance and code size. Fortunately, most programs do not require complete 32-bit addressing space because of limited resources (e.g. ROM size) in practice. You may improve the overall performance and code size simply with the concept of small data area or using different code models in compiler option.

19.7.1. Small Data Area and Relaxation

Small data area, abbreviated to SDA, is created to place global variables which can be addressed by an offset plus register `$gp`. With the help of SDA, the two to three instructions generated to access a global variable in SDA in compilation time can shrink to single instruction by relaxation optimization in link time.

Andes SDA has the section `.sdata_{b|h|w|d}` for initialized global variables and section `.sbss_{b|h|w|d}` for uninitialized ones in default linker script. Section suffix `_{b|h|w|d}` is used to denote the size of a global variable to be `{1|2|4|8}` bytes respectively. For uninitialized global variables, compiler will generate them as common symbols (`.comm` *`symbol, length`*). After linking, the symbols will be put into `.sbss_x`. If you are an assembly programmer, you can put your symbols into `.sdata_x` and `.sbss_x` for relaxation optimization. To understand how relaxation works in link time, here is an example:

```
/* Example-global-1  */

int i;
int j;
int k;
int l;

int func_global_1 ()
{
  return i + j + k + l;
}

int main ()
{
  return func_global_1 ();
}
```

To construct full 32-bit address for each global variable, it may generate assembly code like below that takes at least 8 instructions to load values into registers:

```
func_global_1:
   ! begin of prologue
   ! end of prologue
   sethi  $r2, hi20(i)
   lwi    $r1, [$r2 + lo12(i)]
   sethi  $r3, hi20(j)
   lwi    $r0, [$r3 + lo12(j)]
   add45  $r1, $r0
   sethi  $r4, hi20(k)
   lwi    $r0, [$r4 + lo12(k)]
   add45  $r1, $r0
   sethi  $r5, hi20(l)
   lwi    $r0, [$r5 + lo12(l)]
   add45  $r0, $r1
   ! begin of epilogue
   ret5
   ! end of epilogue
```

Incorporating the concept of small data area, it can generate the following assembly code where global variables "i", "j", "k", and "l" satisfy the size and type requirement of section .sbss_w of SDA and can be allocated there.

```
    .comm   i,4,4
    .comm   j,4,4
    .comm   k,4,4
    .comm   l,4,4
```

After applying relaxation optimization with linker, instructions to access these global variables are reduced to those with addressing of an offset plus $gp.

```
    005000ec <func_global_1>:
      5000ec:      3c 1c 00 87     lwi.gp $r1,[+#0x21c]
      5000f0:      3c 0c 00 84     lwi.gp $r0,[+#0x210]
      5000f4:      88 20           add45 $r1,$r0
      5000f6:      3c 0c 00 86     lwi.gp $r0,[+#0x218]
      5000fa:      88 20           add45 $r1,$r0
      5000fc:      3c 0c 00 85     lwi.gp $r0,[+#0x214]
      500100:      88 01           add45 $r0,$r1
      500102:      dd 9e           ret5 $lp
```

The offset of variables in SDA is limited to +/- 256KB for all scalar data type of V3 architecture. It is unknown if a global variable can be fitted in SDA until linking is done.

With the manipulation of relaxation optimization and SDA, the Example-global-1 can reduce the instruction counts. However, due to the size limitation of instruction immediate, advantages of relaxation optimization and SDA don't always apply to global variables in large programs. In such case, it is suggested to write programs that enclose variables in a global structure. That way, the variables can be aggregated and compiler is able to access them with "base + offset" manner.

19.7.2. Code Models

In Andes toolchains, you can tell compiler which scale your programs and data are with the option `-mcmodel=[small|medium|large]`. Specifying precise code models with this option is helpful for code generation. With clear information, compiler may directly generate smaller and better instructions without relax transformation by linker. The following are three supported code models:

- `-mcmodel=small` (code model: 16M text, 512K data+rodata)
  This option is generally suitable for most MCU programs. It tells compiler that all the function modules must be within 16M range and the global variables, including read-only data, are within 512K range. Compiler assumes that all the data is in the small data area and generates addressing with offset plus `$gp`.
- `-mcmodel=medium` (code model: 16M text, 512K data, 4G rodata)
  This is the default setting in Andes toolchains. For read-only data beyond 512K of small data area, compiler will construct full 32-bit address when accessing them (constant variables). The function modules are still within 16M range of text section; other global variables are within 512K range of small data area and accessible with `$gp` relative instruction.
- `-mcmodel=large` (code model: 4G text, 4G data + rodata)
  This is the option for large programs. All the text and data are all over complete 32-bit addressing space. Compiler uses the most conservative strategy to generate worse assembly code, leaving all the relaxation works to linker.

19.8. Link Time Optimization in GCC

Link Time Optimization (LTO) is a very aggressive optimization implemented by GCC. It gives GCC 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.

19.8.1. 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, 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:

```
$ gcc -O2 -flto -c f1.c
$ gcc -O2 -flto -c f2.c
$ gcc -O2 -flto -o f f1.o f2.o
```

or

```
$ gcc -O2 -flto -o f f1.c f2.c
```

19.8.2. 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.
- If you implement a module that may be called from the MCU standard library (e.g. the weak function `nds_write()` redirected from `libc.a`), it is suggested to use `__attribute__((used))` to prevent it from being optimized out by LTO.
- 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.

19.9. 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 (int argc, char** argv)
  {
    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".

19.10. Inline Assembly Programming

19.10.1. General

Inline assembly programming is a way GCC 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 GCC 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 GCC 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"
           "movi\t$r6, 123\n\t"
           "add\t%0, %0, $r6"
           : "=r" (ret)
           : "r" (i), "r" (j)
           : "$r6");

  return ret;
}
```

From the above example, we can see that "\n\t" is used to separate an instruction from others and "\t" 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.

19.10.2. 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 names 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;
}
```

19.10.3. Clobber List

In a clobber list, registers or memory are listed to inform GCC that these items have been modified. Registers used in an assembly code template have to be specified in the clobber list so that GCC will assume the content of the registers are invalid after the inline assembly statement and generate extra instructions to maintain correct register status. In addition to registers, "memory" can also be listed in a clobber list to make GCC update memory values.

```
/* Example-Asm-3  */

int func_asm_3 (int i, int j)
{
  int ret;

  __asm__ ("add\t%0, %1, %2\n\t"
           "movi\t$r6, 12345\n\t"
           "add\t%0, %0, $r6"
           : "=r" (ret)
           : "r" (i), "r" (j)
           : "$r6");

  return ret;
}
```

With the compiler option “-O1”, Example-Asm-3 will be compiled as:

```
func_asm_3:
   ! begin of prologue
   push.s     $r6, $r6, { }
   addi10.sp  -4
   ! end of prologue
#APP
   add        $r0, $r0, $r1
   movi       $r6, 12345
   add        $r0, $r0, $r6
#NO_APP
   ! begin of epilogue
   addi10.sp  4
```

```
        pop.s       $r6, $r6, { }
        ret5
        ! end of epilogue
```

In Example-Asm-3, GCC is informed that "`$r6`" will be clobbered by the inline assembly statement, so it generates instructions to push/pop callee-saved register "`$r6`" in prologue/epilogue in order to satisfy ABI.

19.10.4. 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 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"
           "movi\t$r6, 12345\n\t"
           "add\t%[write_2], %[read_1], $r6"
           : [write_1] "=r" (ret), [write_2] "=r" (j)
           : [read_1] "r" (i), [read_2] "1" (j)
           : "$r6");

  return ret + j;
}
```

19.10.5. Constraint Modifier "&"

GCC 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__ ("movi\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
   movi       $r1, 12345
   add        $r0, $r0, $r1
#NO_APP
   add45      $r0, $r1
   ret5
```

We can observe that the operand [read_2] uses the same register "$r1" as the operand [write_1]. Since the first instruction clobbers the operand [read_2] when writing [write_1], the second assembly instruction gets wrong content of [read_2]. To avoid this problem, apply constraint modifier "&" to an output operand to inform GCC 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)
{
  int ret1, ret2;

  __asm__ ("movi\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
   movi       $r2, 12345
   add        $r3, $r0, $r1
#NO_APP
   mov55      $r0, $r3
   add45      $r0, $r2
   ret5
```

19.10.6. Volatile

GCC 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 GCC 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__ ("setend.b");
```

19.10.7. Andes-specific Constraints

In the design of AndeStar ISA, the general registers are classified into three levels for 16/32-bit instructions code generation and some instructions implicitly use particular registers. Therefore, we provide following Andes-specific constraints in addition to the general constraint "r" for inline assembly programming.

- l: Low register class $r0 ~ $r7
- d: Middle register class $r0 ~ $r11, $r16 ~ $r19
- h: High register class $r12 ~ $r14, $r20 ~ $r31
- t: Temporary assist register $ta (i.e. $r15)
- v: Register $r5

Example-Asm-7 below demonstrates the result of these special constraints, in which we hold the value of variables i and j with high register class and assign the result to the register $r5:

```
/* Example-Asm-7  */

int func_asm_7 (int i, int j)
{
  int ret;

  __asm__ ("add\t%0, %1, %2\n\t"
           : "=v" (ret)
           : "h" (i), "h" (j));
  return ret;
}
```

The assembly code generated with the option "-O1" is shown below:

```
func_asm_7:
   movd44     $r20, $r0  ! $r20 ← $r0; $r21 ← $r1
#APP
   add        $r5, $r20, $r21
#NO_APP
   add45      $r0, $r5
   ret5
```

Appendix

Programming Tips

Move 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.

Display register information and debug on reset by GDB commands

Andes provides GDB commands to display register information and to debug on reset.

Andes-defined GDB commands to show the content of registers are –

`info registers`	lists all general purpose registers (GPR) and their contents for selected stack frame (NDS32 specific command).
`info registers cr`	lists all configuration system registers (CR) and their contents (NDS32 specific command).
`info registers dmar`	lists all local memory DMA registers (DMAR) and their contents (NDS32 specific command).
`info registers dr`	lists all EDM system registers (DR) and their contents (NDS32 specific command).
`info registers idr`	lists all implementation-dependent registers (IDR) and their contents (NDS32 specific command).
`info registers ir`	lists all interruption system registers (IR) and their contents (NDS32 specific command).
`info registers mr`	lists all MMU system registers (MR) and their contents (NDS32 specific command).
`info registers pfr`	lists all performance monitoring registers (PFR) and their contents (NDS32 specific command).
`info registers racr`	lists all resource access control registers (RACR) and their contents (NDS32 specific command).
`info registers all`	lists all registers and their contents (NDS32 specific command).

Andes also provide the following system-related GDB command to debug on reset.

`reset-and-hold`	To reset the target system and set PC to `0x0.`

This command makes the debugger hold a CPU right after the reset of the debugging target and is especially useful for boot code development.