

Codescape GNU tools for nanoMIPS

ELF ABI Supplement

Revision: 1.3

27/03/2019

Public

The logo consists of the letters 'MIPS' in a bold, blue, stylized sans-serif font. The 'M' is formed by two thick diagonal strokes meeting at the top and a vertical stem. The 'I' is a single thick vertical stroke. The 'P' is formed by a thick vertical stem and a horizontal bar that curves around to the right. The 'S' is formed by two thick horizontal strokes connected by a curved middle section.

This publication contains proprietary information which is subject to change without notice and is supplied 'as is' without warranty of any kind. MIPS, the MIPS logo, Meta, and Codescape are trademarks or registered trademarks of MIPS Tech, LLC. All other logos, products, trademarks and registered trademarks are the property of their respective owners.

Contents

List of Figures.....	v
1 Introduction.....	6
2 ABI Supplement Reference.....	7
3 Function Calling Sequence.....	8
3.1 Register conventions.....	8
3.2 Function parameter passing.....	9
3.3 Passing a variable number of arguments.....	10
The va_list type.....	10
The va_start() macro.....	10
The va_arg() macro.....	10
Argument passing example.....	11
4 nanoMIPS Stack Frame.....	13
4.1 Stack frame organization.....	14
4.2 Frame chaining.....	15
5 nanoMIPS Object Format.....	16
5.1 ELF header.....	16
ELF machine, e_machine.....	16
Processor specific flags, e_flags.....	16
Program header types.....	16
5.2 nanoMIPS ABI flags.....	16
ABIFlags structure.....	17
5.3 Sections.....	18
Section types.....	18
Special sections.....	18
The small data area.....	19
6 Code and Data Models.....	20
6.1 Linker behaviour.....	20
6.2 Compiler command-line options.....	21
6.3 List of code sequences.....	21
Address calculation operation.....	22
Data access operation.....	23
Function call operation.....	25
6.4 Relocations.....	27
6.5 Referring to the GP in code sequences.....	28
6.6 Assembly code compatibility with previous MIPS ABIs.....	28
7 TLS Models.....	29

7.1 TLS Runtime Handling.....	29
7.2 TLS Descriptors.....	29
Code sequences.....	29
Resolver stub.....	30
7.3 List of traditional TLS code sequences.....	30
Global Dynamic.....	30
Local Dynamic.....	30
Initial Exec.....	31
Local Exec.....	32
8 nanoMIPS Data Representation.....	34
8.1 Byte ordering.....	34
8.2 Fundamental Types.....	36
8.3 Aggregates, union and arrays.....	37
Examples.....	37
8.4 Bit fields in structures.....	39
9 Relocation Types.....	40
Relocation Operands.....	40
Relocation Operators.....	40
Data Relocations.....	43
Place-holder Relocations for Linker Relaxations.....	44
10 nanoMIPS Appendix.....	45
Virtual Tables.....	45
Representation of pointers to member functions.....	45
Reserved TRAP, BREAK and SYSCALL codes.....	45

List of Figures

Figure 1: Stack Frame.....14

1 Introduction

This document describes the C/C++ Application Binary Interface for the nanoMIPS 32-bit architecture.

2 ABI Supplement Reference

This section provides the following information for the nanoMIPS ABI:

- Function Calling Sequence
- Stack Frame
- Object Format
- Code and Data Models
- TLS Models
- Data Representation
- Relocation Types

3 Function Calling Sequence

This topic describes the register conventions used for the nanoMIPS hardware.

The nanoMIPS architecture offers 32 general purpose registers, each 32-bit wide. The following table lists all registers and their purpose.

Register	Register	Use	Type
\$r0	\$zero	Hardware zero	
\$r1	\$AT	Assembler temporary	Caller-saved
\$r2..\$r3	\$t4..\$t5	Temporaries	Caller-saved
\$r4..\$r5	\$a0..\$a1	Function arguments / function results / temporary	Caller-saved
\$r6..\$r11	\$a2..\$a7	Function arguments / temporary	Caller-saved
\$r12..\$r15	\$t0..\$t3	Temporary	Caller-saved
\$r16..\$r23	\$s0..\$s7	Saved temporaries	Callee-saved
\$r24..\$r25	\$t8..t9	Temporary	Caller-saved
\$r26..\$r27	\$k0..\$k1	Kernel registers	
\$r28	\$gp	Global pointer	Callee-saved
\$r29	\$sp	Stack pointer	Callee-saved
\$r30	\$fp	Frame pointer	Callee-saved
\$r31	\$ra	Return register	Callee-saved

3.1 Register conventions

- \$zero (\$r0) - hardware zero i.e. always read as zero. It can also be used as a destination register, however, the result is lost.
- \$AT (\$r1) - the register is reserved for the assembler or linker and is used as a temporary register e.g. for expansions of pseudo instructions.
- \$t0-\$t3, \$t4-\$t5, \$t8-\$t9 (\$r12-\$r15, \$r2-\$r3, \$r24-\$r25) - by convention, these registers can be used by subroutines without preserving their contents. However, the registers must be saved before invoking subroutines as there is no guarantee that any of the registers are not destroyed by a subroutine call. The missing \$t6-\$t7 in the naming convention is intentional to keep \$t8-\$t9 tied to registers \$r24-\$r25 for minimal porting effort of MIPS code to nanoMIPS as these registers are likely to be used in many places.
- \$a0-\$a7 (\$r4-\$r11) - used to pass up to 8 arguments to a subroutine. The first two argument registers are dual-purpose: they are used to pass argument and used as return registers. Unused argument registers can be used as temporaries. If the return value is too large to fit into two registers, the compiler will pass a pointer as an invisible first argument.
- \$s0-\$s7 (\$r16-\$r23) - by convention, these registers must be saved and restored by subroutines that use them i.e. the contents at the exit must be the same as on the entry to a subroutine.
- \$k0-\$k1 (\$r26-\$r27) - reserved by the OS/exception handler. These are used as temporaries with original values not restored.
- \$gp (\$r28) - used as global pointer. Typically can be used in embedded applications to provide efficient access to C static/extern data. The addressable range for this area is 2MiB (note: the classic MIPS could only access 64KiB of memory).

- `$sp ($r29)` - used as stack pointer. It takes explicit instructions to raise and lower the stack pointer, so MIPS code usually adjusts the stack only on subroutine entry and exit; and it is the responsibility of the subroutine being called to do this. `$sp` is normally adjusted, on entry, to the lowest point that the stack will need to reach at any point in the subroutine. It must maintain quadword alignment (16-bytes) and grow toward the lower addresses.
- `$fp ($r30)` - uses a frame pointer to keep track of the stack if it's impossible to determine the stack adjustment at the compile time e.g. when the amount is determined at run-time; this may happen for things like variable sized arrays placed on the stack (allowed in C99 standard). The frame pointer, if needed, is initialized to a constant position relative to the function's stack frame with a bias of 4096 bytes. The bias causes `$fp` to point 4096 bytes below the function's top of the current stack frame (toward the lower addresses). If the `$fp` is not used then it can be used as a saved temporary.
- `$ra ($r31)` - return address. The register holds the address to which control should be returned at the end of a subroutine. The register must be saved if a subroutine calls another subroutine and restored before the exit.

Note that the use of named registers is now mandatory which is a deviation from the original MIPS. The format `$N` where `N` is the register number has been deprecated. Alternatively, `$rN` can be used as a substitute for the old format and the number directly maps to the register number.

3.2 Function parameter passing

Arguments are passed to a subroutine in one of two locations: integer (general purpose) registers or on the stack. Use of registers and/or memory depends on the number of arguments, their type and positions in the argument list. The ABI allows to pass up to 8 arguments to a subroutine. The ABI defines only soft-float ABI, thus, floating-point arguments are being treated as integers of the same size. Floating-point return values shall also be passed in the integer registers.

The calling sequence has the following characteristics:

- All stack regions are quadword (16 byte) aligned.
- Up to eight integer registers (`$a0` . . `$a7`) may be used to pass integer arguments.
- All integer arguments are passed as 32-bit words, with signed or unsigned bytes and halfwords expanded (promoted) as necessary. The caller is responsible for the promotion.
- All pointers and addresses are 32-bit objects.
- Floating point arguments are treated as integers of the same size.
- Structs, unions, or other composite types are treated as a sequence of words, and are passed in integer registers as though they were simple scalar parameters to the extent that they fit, with any excess on the stack packed according to the normal memory layout of the object. More specifically:
 - Regardless of the struct field structure, it is treated as a sequence of 32-bit chunks.
 - A structure cannot be split between registers and the stack i.e. it can either be passed in registers or on the stack.
 - A union, either as the parameter itself or as a struct parameter field, is treated as a sequence of integer words for purposes of assignment to integer parameter (argument) registers.
 - Array fields of structs are passed like unions. Array parameters are passed by reference (unless the relevant language standard requires otherwise).
 - Right-justifying small scalar parameters in their save area slots notwithstanding, struct parameters are always left-justified. This applies both to the case of a struct smaller than 32 bits, and to the final chunk of a struct which is not an integral multiple of 32 bits in size. The implication of this rule is that the address of the first chunk's save area slot is the address of the struct, and the struct is laid out in the save area memory exactly as if it were allocated normally.
- An argument that cannot be allocated to an argument register is passed in memory on the stack and pointed to by the stack pointer at the time of call.
- An argument can be up to two register widths in size i.e. 64-bits object can be passed in a pair of 32-bit registers; alignment will be applied (depending on the type) as necessary.

- All vector types are passed by reference.
- No requirement is placed on the callee either to allocate space and save the argument registers, or to save them in any particular place.
- Variable argument functions follow the same rules with the exception that the callee is responsible for reserving space for anonymous arguments in the general purpose argument register save area on function entry.

3.3 Passing a variable number of arguments

The `va_list` type

The `va_list` type is used to track arguments passed to a variadic function. An argument is passed in one of the three locations: general purpose registers, floating-point registers or in the overflow area (the stack). The prologue of a function shall save the incoming argument registers (both GP and FP registers) within its own stack frame. Then the `va_list` can refer to any argument depending on its type and position in the argument list.

In this soft-float ABI document, floating-point types are passed in integer registers, hence, both the pointer to FP area and the FP offset are not used but reserved for future use.

```
typedef struct __va_list {
    void *__overflow_argptr; // next stack arg
    void *__gpr_top;         // top of the GP arg reg save area
    void *__fpr_top;         // top of the FP arg reg save area
    signed char __gpr_offset; // offset from __gpr_top to next GP reg arg
    signed char __fpr_offset; // offset from __fpr_top to next FP reg arg
} va_list;
```

The `va_start()` macro

The macro is required to initialize all the fields of a `va_list` argument and it should be invoked before accessing any unnamed arguments. The `MAX_ARGS_IN_REGISTERS` is 8, `num_gprs` is the number of general registers that hold named incoming arguments and the `num_fprs` is the number of floating point registers that hold named incoming floating-point arguments.

- `__overflow_argptr` - this pointer is used to get arguments passed on the stack. It is initialized to the first (lowest address) named incoming argument on the stack and it is updated to point to the next argument on the stack.
- `__gpr_top` - this pointer points to the top of the general purpose argument register save area.
- `__fpr_top` - this pointer points to the top of the floating-point argument register save area.
- `__gpr_offset = (MAX_ARGS_IN_REGISTERS - num_gprs) * UNITS_PER_WORD`; this holds the offset in bytes from the `__gpr_top` pointer to the location where the next general purpose argument register is saved.
- `__fpr_offset = (MAX_ARGS_IN_REGISTERS - num_fprs) * UNITS_PER_FPREG`; this holds the offset in bytes from the `__fpr_top` pointer to the location where the next floating-point argument register is saved.

Note that `__fpr_top` and `__fpr_offset` are reserved for future use and currently unused in the soft-float mode.

The `va_arg()` macro

The macro is used to modify parameter `ap` of `va_list` type to get to the value of successive arguments. The following is simplified C code that calculates the address of an argument for a type and updates the `va_list` after processing an argument.

```
/*
TOP be the top of the GPR or FPR save area;
OFF be the offset from TOP of the next register;
CUR_OFF copy of OFF to calculate the ADDR for
the current TYPE
SIZE be the number of bytes in the argument type;
RSIZE be the number of bytes used to store the argument
```

```

    when it's in the register save area; and
    OSIZE be the number of bytes used to store it when it's
    in the stack overflow area.
    ADDR the address of argument either in the register or
    stack overflow area.
    ALIGN required alignment for TYPE.
    REG_WIDTH is the register width, either the general
    purpose or floating-point register.
*/

type va_arg (va_list ap, type)
{
    int offs, size, rsize, osize, regsize, cur_off;
    intptr_t top, off, ovfl, addr;
    unsigned align;

    size = int_size_in_bytes (type);
    align = function_arg_alignment (mode, type);

    if (passed_in_gprs) {
        top = ap.__gpr_top;
        off = ap.__gpr_offset;
        rsize = round_up (size, UNITS_PER_WORD);
        osize = rsize;
        reg_width = UNITS_PER_WORD;
    } else {
        top = ap.__fpr_top;
        off = ap.__fpr_offset;
        rsize = UNITS_PER_HWFPVALUE;
        osize = MAX (GET_MODE_SIZE (TYPE_MODE (type)), UNITS_PER_WORD);
        reg_width = UNITS_PER_HWFPVALUE;
    }

    if (off > 0) {
        if (align > reg_width)
            off &= -align;
        cur_off = off;
        off -= rsize;
        if (off >= 0) {
            addr = *(type *) (top - cur_off + (BYTES_BIG_ENDIAN ? rsize - size : 0));
            goto done;
        }
    }

    ovfl = ap.__overflow_argptr;
    if (align > UNITS_PER_WORD)
        ovfl = (ovfl + 2 * UNITS_PER_WORD - 1) & -(2 * UNITS_PER_WORD);
    addr = (void *) (ovfl + (BYTES_BIG_ENDIAN ? osize - size : 0));
    ovfl += osize;

done:
    return *(type *) addr;
}

```

Argument passing example

The following code is an example of how various types will be passed in registers or on the stack using the calling conventions.

```

typedef struct {
    int a;
    double b;
} sarg_t;

typedef struct {
    char a[2];
    int b;
} sarg2_t;

int ia, ib;
long long lla;
float fa;
double da;
sarg_t sa;
sarg2_t s2a;

r = foo(ia, fa, da, sa, ib, lla, s2a);

```

Argument	Register	Stack offset
ia	\$a0	X
fa*	\$a1	X
da	\$a2-\$a3	X
sa^	\$a4	X
ib	\$a5	X
lla	\$a6-\$a7	X
s2a	X	0-7

Notes:

* - it might be subject to the default argument promotion causing different argument register/stack allocation. It is assumed that the prototype for `foo` is provided here.

^ - Passed by reference.

X - Unavailable/not used.

4 nanoMIPS Stack Frame

Each subroutine allocates a stack frame on the runtime stack, as required. A frame is allocated for each non-leaf function and for each leaf function that requires stack storage.

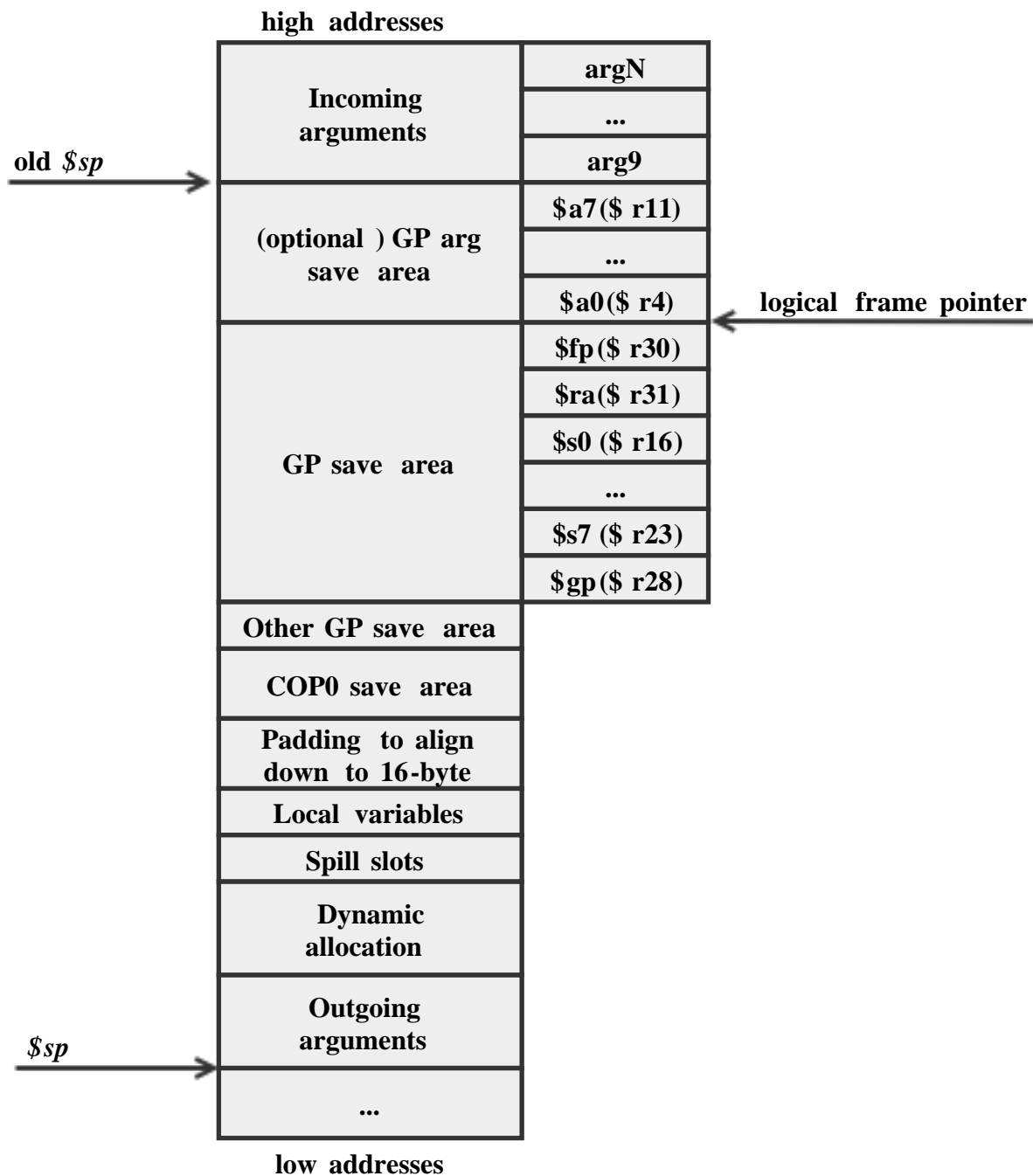
A non-leaf function is one that calls other function(s); a leaf function is one that does not itself make any function calls.

The stack grows downward from high addresses towards lower addresses. The stack frame may have a fixed or dynamically extendable size.

4.1 Stack frame organization

The following diagram shows how the compiler lays out the stack.

Figure 1: Stack Frame



The following requirements apply to the stack frame:

- The stack pointer shall maintain quadword (16-byte) alignment.
- The stack pointer shall be decremented in the function prologue and, if required, restored prior to return.
- Frame chain consisting of frame pointer and return address, if required, to be saved below the logical frame pointer.
- The logical frame pointer, if required, must be biased by 4096 bytes downward from the top of the frame.
- Allocate space for (in the order specified):
 - GP argument save area. Used only for variadic functions. Space for argument registers for anonymous arguments shall be allocated by the caller. Space must be padded to the alignment requirements.
 - Saved temporary registers area. Space for `$fp`, `$ra`, `$s0-$s7`, `$gp` registers, respectively. These must be saved before changing their contents and restored at exit.
 - Caller saved registers in an interrupt handler. Used only in interrupt handler routines.
 - COP0 save area. Space to preserve `STATUS` and `EPC` registers.
 - Padding to align down to quadword (0, 4, 8 or 12 bytes).
 - Local variables and temporaries.
 - Outgoing arguments if passed arguments to a function do not fit into the argument registers.

Most of the time, the order of registers being saved and restored will not matter for the user. However, the order is mandatory for assembly programmers, especially, when accessing a required slot or when mixing `SAVE` or `RESTORE` instructions with manually saved/restored registers. Note that the order of saved/restored registers is also used by the `SAVE/RESTORE` instructions. Refer to the nanoMIPS ISA for further details.

4.2 Frame chaining

The p32 ABI allows more efficient stack backtracing compared to older MIPS ABIs as the frame pointers form a chain. To achieve this there is a fixed offset from any frame-pointer to the parent frame-pointer save location. In other words, there is no need to scan instructions to find the location of the parent frame pointer. The frame pointer, however, is now biased by 4096 bytes to enable full use of the unsigned 12-bit offsets in memory instructions when using the frame pointer as a base.

Stack backtracing requires the frame pointer/return address registers saved just below the logical frame pointer in the aforementioned order.

We can calculate the addresses needed for correct stack unwinding as follows:

```
logical frame pointer = (FP + 4096)
caller's frame pointer = *(int32_t *) (FP + 4096 - 4)
return address = *(int32_t *) (FP + 4096 - 8)
```

Note: The frame pointer is not mandatory. However, if the supporting tools enable the frame pointer then the location of the frame pointer is guaranteed and the return address is guaranteed for non-leaf functions at the specified location.

5 nanoMIPS Object Format

This section describes the object format; including information about the ELF header and object files.

5.1 ELF header

This section covers nanoMIPS specific variations in the header fields. The remaining fields take the usual values and meanings from the gABI specification.

ELF machine, e_machine

Name	Value
EM_NANOMIPS	249

Processor specific flags, e_flags

Name	Mask	Description
EF_NANOMIPS_LINKRELAX	0x00000001	Control link-time relaxation
EF_NANOMIPS_PIC	0x00000002	Position independent code
EF_NANOMIPS_32BITMODE	0x00000004	Indicates 32-bit object for 64-bit architecture
EF_NANOMIPS_PID	0x00000008	Position independent data
EF_NANOMIPS_PCREL	0x00000010	PC-relative mode, trivially position independent
EF_NANOMIPS_ABI	0x0000F000	nanoMIPS ABI
EF_NANOMIPS_MACH	0x00FF0000	Machine variant
EF_NANOMIPS_ARCH	0xF0000000	nanoMIPS architecture

nanoMIPS ABI bits, EF_NANOMIPS_ABI

Name	Value	Description
E_NANOMIPS_ABI_P32	1	32-bit nanoMIPS ABI
E_NANOMIPS_ABI_P64	2	64-bit nanoMIPS ABI

nanoMIPS architecture bits, EF_NANOMIPS_ARCH

Name	Value	Description
E_NANOMIPS_ARCH_32R6	0	32-bit nanoMIPS Release 6 ISA
E_NANOMIPS_ARCH_64R6	1	64-bit nanoMIPS Release 6 ISA

Program header types

Name	Value
PT_NANOMIPS_ABIFLAGS	0x70000000

5.2 nanoMIPS ABI flags

ABI flags is a MIPS specific structure encapsulated in the `.MIPS.abiflags` section. The structure for nanoMIPS is identical to MIPS, except that it is created within a section named `.nanoMIPS.abiflags`. Most of the fields for

nanoMIPS take identical values and meanings as MIPS. The exceptions are the processor specific ISA extensions and the Application Specific Extension (ASE) flags.

ABIFlags structure

Field Name	Type	Comments
version	unsigned short	Version of flags structure
isa_level	unsigned char	Level of the ISA: 32, 64
isa_rev	unsigned char	Revision of the ISA: 6-n
gpr_size	unsigned char	Size of general purpose registers
cpr1_size	unsigned char	Size of co-processor 1 registers
cpr2_size	unsigned char	Size of co-processor 2 registers
fp_abi	unsigned char	Floating-point ABI
isa_ext	unsigned long	Processor specific extension
ases	unsigned long	Mask of ASEs used
flags1	unsigned long	Mask of general flags
flags2	unsigned long	Mask of general flags

Values for the xxx_size bytes of an ABI flags structure

Name	Value	Description
AFL_REG_NONE	0x00	No registers
AFL_REG_32	0x01	32-bit registers
AFL_REG_64	0x02	64-bit registers
AFL_REG_128	0x03	128-bit registers

Values for the ASEs field of an ABI flags structure

Name	Value	Description
NANOMIPS_ASE_TLB	0x00000001	TLB control
NANOMIPS_ASE_EVA	0x00000004	Enhanced VA Scheme
NANOMIPS_ASE_MCU	0x00000008	MCU (MicroController) extension
NANOMIPS_ASE_MT	0x00000040	Multi-threading extension
NANOMIPS_ASE_VIRT	0x00000100	Virtualization extension
NANOMIPS_ASE_MSA	0x00000200	MSA extension
NANOMIPS_ASE_RESERVED1	0x00000400	was MIPS16 ASE
NANOMIPS_ASE_RESERVED2	0x00000800	was MICROMIPS ASE
NANOMIPS_ASE_DSPR3	0x00002000	DSP R3 extension
NANOMIPS_ASE_CRC	0x00008000	Cyclic Redundancy Check extension
NANOMIPS_ASE_GINV	0x00020000	Global INvalidate extension
NANOMIPS_ASE_xNMS	0x00040000	Full base ISA, not the nanoMIPS subset

Values for the floating-point ABI field of the ABI flags structure

Name (Value)	Description
Val_GNU_NANOMIPS_ABI_FP_ANY (0)	Not specified
Val_GNU_NANOMIPS_ABI_FP_DOUBLE (1)	Double-precision hard float

Name (Value)	Description
Val_GNU_NANOMIPS_ABI_FP_SINGLE (2)	Single-precision hard float
Val_GNU_NANOMIPS_ABI_FP_SOFT (3)	Soft float

5.3 Sections

Section types

Name	Value
SHT_NANOMIPS_ABIFLAGS	0x70000000

SHT_NANOMIPS_ABIFLAGS

The type of the section for the nanoMIPS specific ABI flags structure.

Special sections

These section names are specific to nanoMIPS:

Name	Type	Attributes
.ssdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.ssbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.nanoMIPS.abiflags	SHT_NANOMIPS_ABIFLAGS	SHF_ALLOC

.nanoMIPS.abiflags

This section contains the nanoMIPS specific ABI flags structure.

.ssdata

This section holds initialized data of type byte, short, float, and double that contribute to the program's memory image. It must be placed within 256KiB positive range of the Global Pointer (GP).

.ssbss

This section holds uninitialized data of type byte, short, float, and double. It must be placed within 256KiB positive range of GP. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, SHT_NOBITS.

The meaning of these section names from gABI is specialized for nanoMIPS:

Name	Type	Attributes
.sdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.sbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.got	SHT_PROGBITS	SHF_ALLOC

.sdata

This section holds initialized data of type word and, only for 64-bit, double-word that contribute to the program's memory image. It must be placed within 2MiB positive range of GP.

.sbss

This section holds uninitialized data of type word and, only for 64-bit, double-word. It must be placed within 2MiB positive range of GP. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, SHT_NOBITS.

.got

This section holds the global offset table consisting of relocated address entries. It must be placed in positive range of GP.

The small data area

The small data area is a region of 2MiB starting at the Global Pointer (GP) which contains the small data sections and the GOT. These sections may be placed in any order, as long as the placement constraints of each individual section are satisfied. The recommended ordering to best exploit GP-relative addressing capabilities, is as follows:

GP =>

```
.got
.ssdata
.ssbss
.sdata
.sbss
```

6 Code and Data Models

This topic describes the code and data models supported by the nanoMIPS p32 ABI. The models offer a way to control the size of intermediate object files and fully linked executables. They provide flexibility to produce the most compressed code possible or to produce code which accommodates the largest of applications.

There are 3 models available:

- **automatic:** generates the most compressed code possible by relying on the linker to expand or relax code appropriately, depending on a given symbol's visibility at link-time.
- **medium:** generates code which is appropriate for average-sized applications without relying on linker relaxations or expansions. The GP area can not exceed 2MiB and direct calls have a maximum range of 32 MiB.
- **large:** generates code which is appropriate for large applications without relying on linker relaxations or expansions. There are no inherent size limits when using this model.

Even though the automatic model will produce the smallest code size, the code generated at compile-time in this model is expected to be a safe approximation of the size of the fully linked code.

The automatic model requires a linker which can perform relaxations and expansions, while the medium and large models do not. Even so, linker relaxations and expansions can be used with the medium and large models, but they will be less effective than in the automatic model.

In addition to the models, there are 4 addressing modes:

- **absolute:** addresses are fixed at link-time. This mode is rarely necessary but has some potential for energy efficiency.
- **PC-relative:** addresses appear as offsets from the PC and are used in PC-relative instructions. This mode produces position-independent code.
- **GP-relative:** addresses appear as offsets from the GP and are used in GP-relative instructions. Symbols are placed in the small data section, also known as `.sdata`. This mode produces position-independent data for some or all symbols of an application.
- **GOT-dependent:** addresses are kept in the GOT and are loaded by using offsets between the GP and a given symbol's entry in the GOT. This mode produces dynamically linkable code.

Absolute addressing can use PC-relative instructions, but only if they can decrease code size without decreasing performance.

Position-independent data requires data sections which can be relocated independently of code. This can be achieved by enforcing GP-relative addressing for all non-pre-emptible data symbols.

Pre-emptible symbols can only be accessed using GOT-dependent addressing. A pre-emptible symbol is a symbol whose local definition may be overridden by an external definition at dynamic-link-time. When doing a static link, the linker will relax all GOT-dependent sequences into PC-relative or absolute sequences.

There are no restrictions on intermixing different addressing modes.

A symbol can use only one code model and one addressing mode at a time. A compiler must decide which combination is appropriate for each symbol, however, a linker can change the addressing mode as it sees fit, when performing relaxations or expansions.

6.1 Linker behaviour

A linker can expand code sequences when a symbol is out of range of the instructions used and it can relax code sequences if it is possible to use a more compact sequence for a symbol. A linker must check the `LINKRELAX` ELF header flag to find out if it can do any such transformations. If the flag is not set, the linker must only fix-up relocations and not change any instructions in text sections. It can still introduce extra code such as trampolines/stubs between sections and control how instructions are relocated, without changing the instructions themselves.

A linker can delete the `$gp` setup instruction if it manages to eliminate all other references to `$gp` from a function. This instruction can be found by looking for a `R_NANOMIPS_PC21_S1` relocation applied to the `_gp` symbol. If all references to `$gp` have been eliminated from a function, then the `$gp` register can be removed from the function's `SAVE` and `RESTORE` instructions, which may enable the use of their 16-bit variants. `SAVE` and `RESTORE` instructions which have `$gp` in their list can be found by looking for the `R_NANOMIPS_SAVERESTORE` relocation.

A linker can find the information it needs for making decisions about relaxations and expansions by looking at the input objects' ELF header flags and through its command-line options.

A linker must check the `PCREL` ELF header flag to find out if it must emit PC-relative sequences or is allowed to emit absolute sequences when doing transformations. It must also check the `PID` ELF header flag to find out whether it must enforce position-independent addressing for non-pre-emptible data symbols, and the nanoMIPS ABI flags to find out whether it is restricted to using nanoMIPS subset instructions.

It is advisable to give end users a command-line option which turns off linker expansions and relaxations, so that they can have more control over the link.

A linker does not need to know which model was explicitly chosen by the user at compile-time.

A linker must give an error if it finds any absolute sequences in position-independent code objects.

A simple linker, which does not support any relaxations and expansions, must give an error if it comes across automatic model sequences that it can not relax or expand.

6.2 Compiler command-line options

Option	Description
<code>-mcmmodel={auto,medium,large}</code>	Used to select the code model for non-pre-emptible symbols.
<code>-mpcrel</code>	Forces PC-relative addressing for non-pre-emptible symbols. This also effectively disables absolute addressing.
<code>-mno-pcrel</code>	Avoids PC-relative addressing for non-pre-emptible symbols where alternative sequences are not larger or slower.
<code>-mgpopt</code>	Enables GP-relative addressing for non-pre-emptible symbols which are within the size limit specified by the <code>-G</code> option.
<code>-mno-gpopt</code>	Disables GP-relative addressing completely.
<code>-mpid</code>	Enforces position-independent data by requiring GP-relative addressing for all non-pre-emptible data symbols.
<code>-fpic</code>	Turns on GOT-dependent addressing for pre-emptible symbols. If the model is not automatic, this will generate medium model GOT-dependent sequences.
<code>-fPIC</code>	Turns on GOT-dependent addressing for pre-emptible symbols. If the model is not automatic, this will generate large model GOT-dependent sequences.
<code>-mrelax</code>	Enables linker relaxation for the compilation unit. The compiler emits a <code>.linkrelax</code> directive in the assembly file which causes an ELF header flag to be set in the object.

Note: `-mlong-calls` forces the generation of large model sequences only for non-pre-emptible functions.

Note: The automatic model is set as the default model and linker relaxations are enabled by default.

Note: If no model is specified, both `-fpic` and `-fPIC` will enable automatic model GOT-dependent sequences.

6.3 List of code sequences

The following tables list all of the code sequences used for all possible combinations of code model and addressing mode. There are a few sequences which can be generated only by either a compiler or a linker. A compiler is

responsible for the bulk of the generation, while a linker adjusts existing sequences by relaxing or expanding them. Each sequence has a set of conditions which need to be fulfilled in order for it to be chosen by a compiler or linker. The sequences are in ascending order according to code size.

Table terminology

non-pre: Short for non-pre-emptible symbol.

pre: Short for pre-emptible symbol.

A, M, L: Abbreviations for the automatic, medium, and large model, respectively. These apply only to compiler-generated sequences.

NMS: This is the abbreviation used for the nanoMIPS subset, which excludes some instructions from the ISA for power efficiency reasons.

NMF: This is the abbreviation used for the full nanoMIPS ISA. This is used to contrast against NMS.

insn32 mode: When enabled, the generation of 16-bit and 48-bit instructions is forbidden. This is not equivalent to NMS or NMF, since both of them contain instructions which are not 32-bit.

[ls]/[suff]: The "[ls]" means that this instruction can be either a load or a store. The "[suff]" stands for all the supported suffixes (H, B, W, D, WC1, DC1 for both loads and stores, and HU, BU, WU for loads-only).

Address calculation operation

Category	Model	Conditions	Code sequence	Size	Remarks
Absolute addressing					
non-pre	AML	4KiB-aligned	lui reg, %hi(symbol)	4	Must be used only for symbols which are explicitly 4KiB-aligned.
non-pre	A	NMS-only, >=2B-aligned, within PC +1MiB	lapc.h reg, symbol	4	Must be used only instead of LUI %hi + ORI %lo.
non-pre	AML	NMF-only	li reg, symbol	6	Uses LI[48].
non-pre	AML		lui reg, %hi(symbol); ori reg, reg, %lo(symbol)	8	
PC-relative addressing					
non-pre	AML	4KiB-aligned	aluipc reg, %pcrel_ hi(symbol)	4	Must be used only for symbols which are explicitly 4KiB-aligned.
non-pre	A	>=2B-aligned, within PC+1MiB	lapc.h reg, symbol	4	
non-pre	AML	NMF-only	lapc.b reg, symbol	6	
non-pre	AML		aluipc reg, %pcrel_ hi(symbol); ori reg, reg, %lo(symbol)	8	
GP-relative addressing					

Category	Model	Conditions	Code sequence	Size	Remarks
non-pre	AM	$\leq 2\text{B}$ -aligned, within GP+256KiB	<code>addiu.- b reg, \$gp, %gprel(symbol)</code>	4	
non-pre	AM	$\geq 4\text{B}$ -aligned, within GP+2MiB	<code>addiu.- w reg, \$gp, %gprel(symbol)</code>	4	
non-pre	L	NMF-only	<code>addiu.b32 reg, \$gp, %gprel(symbol)</code>	6	
non-pre	L		<code>lui reg, %gprel_- hi(symbol); ori reg, reg, %gprel_- lo(symbol); addu reg, reg, \$gp</code>	12	
GOT-dependent addressing					
non-pre		within GP+2MiB, refcount ≥ 2 , -Os	<code>lw reg, %got_- disp(symbol) (\$gp)</code>	4+4 data	Generated only by a linker.
pre	AM		<code>lw reg, %got_- disp(symbol) (\$gp)</code>	4+4 data	
pre	L	NMF-only	<code>lwpc reg, %got_pcrel_- 32(symbol)</code>	6+4	
pre	L		<code>aluipc reg, %got_- pcrel_- hi(symbol); lw reg, %got_- lo(symbol) (reg)</code>	8+4	

Data access operation

Category	Model	Conditions	Code sequence	Size	Remarks
Absolute addressing					
non-pre	AML	4KiB-aligned	<code>lui reg1, %hi(symbol); [ls][suff] reg2, 0(reg1)</code>	6/8	Must be used only for symbols which are explicitly 4KiB-aligned. Uses LW[16]/ SW[16] if the registers happen to be GPR3.

Category	Model	Conditions	Code sequence	Size	Remarks
non-pre	AML		lui reg1, %hi(symbol); [ls][suff] reg2, %lo(symbol) (reg1)	8	
PC-relative addressing					
non-pre	AML	NMF-only, word-sized symbol	[ls]wpc reg2, symbol	6	
non-pre		≥ 2 B-aligned, within PC+1MiB, reg1 and reg2 are GPR3	laptopc. h reg1, symbol; [ls][suff] reg2, 0(reg1)	6	Generated only by the linker.
non-pre	AML	4KiB-aligned	aluipc reg1, %pcrel_ hi(symbol); [ls][suff] reg2, 0(reg1)	6/8	Must be used only for symbols which are explicitly 4KiB-aligned. Uses LW[16]/SW[16] if the registers happen to be GPR3.
non-pre	AML		aluipc reg1, %pcrel_ hi(symbol); [ls][suff] reg2, %lo(symbol) (reg1)	8	
GP-relative addressing					
non-pre	AM	byte/half-sized symbol, within GP+256KiB	[ls][bh] reg2, %gprel(symbol) (\$gp)	4	
non-pre	AM	word-sized symbol, within GP+2MiB	[ls]w reg2, %gprel(symbol) (\$gp)	2/4	Uses LW[GP16]/SW[GP16] if the register happens to be GPR3.
non-pre	L	NMF-only	addiu.b32 reg1, \$gp, %gprel(symbol) [ls][suff] reg2, 0(reg1)	8/10	Uses LW[16]/SW[16] if the registers happen to be GPR3.

Category	Model	Conditions	Code sequence	Size	Remarks
non-pre	L		lui reg1, %gp _{rel} _ hi(symbol); addu reg1, reg1, \$gp; [ls][suff] reg2, %gp _{rel} _ lo(symbol) (reg)	12	
GOT-dependent addressing					
pre	A		lw reg1, %got_ page(symbol) (\$gp); [ls] [suff] reg2, %got_ ofst(symbol) (reg1)	8+4 data	Generated only by the compiler. The linker has to transform this into one of the other pre-emptible sequences or relax it into a non-pre-emptible sequence.
pre	M		lw reg1, %got_ disp(symbol) (\$gp); [ls] [suff] reg2, 0(reg1)	6/8+4 data	Uses LW[16]/SW[16] if the registers happen to be GPR3.
pre	L	NMF-only	lwpc reg1, %got_pcrel- 32(symbol); [ls][suff] reg2, 0(reg1)	8/10+4 data	Uses LW[16]/SW[16] if the registers happen to be GPR3.
pre	L		aluipc reg1, %got_ pcrel_ hi(symbol); lw reg1, %got_ lo(symbol) (reg1); [ls][suff] reg2, 0(reg1)	10/12+4 data	Uses LW[16]/SW[16] if the registers happen to be GPR3.

Function call operation

Category	Model	Conditions	Code sequence	Size	Remarks
Absolute addressing					
non-pre	AM	within PC+-32MiB	balc symbol	2/4	

Category	Model	Conditions	Code sequence	Size	Remarks
non-pre	L	NMF-only	li reg, symbol; jalrc \$31, reg	8	Will always use JALRC[16] because it will be skipped altogether in insn32 mode, as it is using LI[48].
non-pre	L		lui reg, %hi(symbol); ori reg, reg, %lo(symbol); jalrc \$31, reg	10/12	Uses JALRC[32] in insn32 mode and JALRC[16] otherwise.
PC-relative addressing					
non-pre	AM	within PC+32MiB	balc symbol	2/4	
non-pre	L	NMF-only	lapc.b reg, symbol; jalrc \$31, reg	8	Will always use JALRC[16] because it will be skipped altogether in insn32 mode, as it is using LAPC.B, which is an alias for ADDIUPC[48].
non-pre	L		aluipc reg, %pcrel_ hi(symbol); ori reg, reg, %lo(symbol); jalrc \$31, reg	10/12	Uses JALRC[32] in insn32 mode and JALRC[16] otherwise.
GOT-dependent addressing					
non-pre		within GP+2MiB, refcount>=2, -Os	lw reg, %got_ disp(symbol) (\$gp); jalrc \$31, reg	6/8+4 data	Generated only by the linker. Uses JALRC[32] in insn32 mode and JALRC[16] otherwise.
pre	AM		lw reg, %got_ call(symbol) (\$gp); jalrc \$31, reg : R_ NANOMIPS_ JALR symbol	6/8+4 data	Generated only by the compiler. The linker has to transform this into one of the other pre-emptible sequences or relax it into a non-pre-emptible sequence. Uses JALRC[32] in insn32 mode and JALRC[16] otherwise.

Category	Model	Conditions	Code sequence	Size	Remarks
pre	L	NMF-only	lwpc reg, %got_pcrel- 32(symbol); jalrc \$31, reg	8+4 data	Will always use JALRC[16] because it will be skipped altogether for insn32 mode, as it is using LWPC, which is 48 bits long.
pre	L		aluipc reg, %got_- pcrel_- hi(symbol); lw reg, %got_- lo(symbol) (reg); jalrc \$31, reg	10/12+4 data	Uses JALRC[32] in insn32 mode and JALRC[16] otherwise.

6.4 Relocations

The following relocations are only used in the automatic model:

- %got_page, %got_ofst: together they tell the linker that this is a data access (either a load or a store); these are fake relocations and their only purpose is to enable the linker to perform relaxations

The following relocations are used in both the automatic and the medium code model:

- %got_disp: 21-bit (19-bit scaled) GOT displacement pointing to the symbol's GOT entry
- %got_call: tells the linker that the address being loaded is a text address which will be used to call a function; also used to indicate to the static linker that this function is eligible for lazy binding
- R_NANOMIPS_JALR16, R_NANOMIPS_JALR32: tells the linker that it can relax this JALRC16 or JALRC32 to a BALC, if the symbol is non-pre-emptible.

The following relocations are only used in the large code model:

- %gprel_hi: 20-bit high part of a 32-bit GP displacement
- %gprel_lo: 12-bit low part of a 32-bit GP displacement
- %got_pcrel32: 32-bit PC displacement pointing to the symbol's GOT entry
- %got_pcrel_hi: 20-bit high part of a 32-bit PC displacement pointing to the symbol's GOT entry
- %got_lo: 12-bit low part of the address of the symbol's GOT entry.

The following relocations are used in all models:

- %gprel: 18 or 21-bit GP displacement for auto and medium, and 32-bit GP displacement for large
- %pcrel_hi: 20-bit high part of a 32-bit PC displacement
- %lo: 12-bit low part of a 32-bit address

%got_pcrel32 can only be used when targeting the full nanoMIPS ISA.

%lo is used in both absolute and PC-relative addressing because the upper part of the address will always end up at a 4KiB boundary.

6.5 Referring to the GP in code sequences

Both GP-relative and GOT-dependent addressing refer to the GP (Global Pointer) in their code sequences and need to have the address of the GP loaded into the `$gp` register.

For GOT-dependent addressing, the `$gp` needs to be explicitly set up in every function which uses it and can be called from a module which has a different GOT, and thus a different value for GP. It does not need to be set up if the function is local, all of its callers are local, and all of its callers are guaranteed to have a correct `$gp`.

For GP-relative addressing, the `$gp` only needs to be set up before entering `main()`. This is an implicit setup of `$gp`. Note that combining these two addressing modes will make it necessary to explicitly set up `$gp` in a function which can be called externally, even if that function contains only GP-relative sequences.

The implicit setup is part of library code, while the explicit setup is generated by the compiler. The code sequence for the explicit setup is just a regular PC-relative address calculation, while the implicit setup sequence is restricted to `LAPC.B` (i.e. `ADDIUPC`), as this prevents us from relying on linker transformations for critical library code (e.g. the `crt`, etc.) and frees us from being boxed in by the limitations of absolute addressing.

The GP does not have any special alignment requirements. This gives the linker more flexibility in determining the data layout of the application.

Note that the `$gp` register is callee-saved, so it needs to be saved and restored before returning in every function which uses it.

6.6 Assembly code compatibility with previous MIPS ABIs

The `.cpadd`, `.cpreturn` and `.cprestore` assembler directives are ignored for nanoMIPS. The `.cpsetup` and `.cpload` directives have been kept, but they are now used only for generating the `$gp` setup instruction. `.-cpsetup` is preferred over `.cpload`, which has been kept only for the sake of compatibility.

The following relocation operands from previous MIPS ABIs have been repurposed for nanoMIPS:

- `%got_page`, `%got_ofst`: are now fake relocations which enable the linker to perform relaxations
- `%got_disp`: is now 21 bits (19-bits scaled) instead of 16 bits
- `%got`: is now treated in the same way as `%got_disp`; this means that adding `%lo` to a `%got` will corrupt the address being calculated
- `%hi`, `%lo`: are now 20 and 12 bits, respectively

7 TLS Models

The nanoMIPS ABI has support for all of the traditional TLS models. In addition to this, it also offers the possibility of using TLS descriptors to get better performance compared to the traditional dynamic models.

In order to gain more control over code size, all of the TLS code sequences have been adapted to follow the automatic/medium/large approach taken in the code and data models.

Note that TLS is not supported for the nanoMIPS subset.

7.1 TLS Runtime Handling

The runtime TLS layout has been redesigned to take advantage of the unsigned offset `LW[U12]` nanoMIPS instruction, thus extending the possible range of symbols inside the TLS block.

This means that the implicit offsets of 0x7000 for the TP and 0x8000 for the DTV pointers have been removed. The TP now points to the end of the TCB and the DTV pointers to the start of their TLS blocks.

Note that the nanoMIPS TLS ABI is based on Variant 1 from Ulrich Drepper's "ELF Handling for Thread-Local Storage" paper.

7.2 TLS Descriptors

The purpose of TLS descriptors is to improve performance compared to the traditional dynamic TLS models. This is achieved through minimizing calls to `__tls_get_addr` by having the dynamic linker provide TP offsets for TLS symbols in initially loaded modules, and using these offsets to calculate the full symbol address locally instead of going through the dynamic resolver for every symbol.

The TLS descriptors design for nanoMIPS is very similar to existing designs for other architectures, with one important difference: there is no static resolver. It has been effectively inlined into the local resolver stub.

Code sequences

Model	Conditions	Code sequence	Size	Static relocations	Dynamic relocations	Remarks
AM	within GP +2MiB	<code>addiu.w \$a0, \$gp, %tlsdesc_ got(symbol) balc %tlsdesc_ call(symbol)</code>	6/8	<code>R_NANOMIPS_ symbol; R_NANOMIPS_ symbol</code>	<code>GOT: R_NANOMIPS_ symbol; GOT: 0</code>	The two GOT entries will get filled by the dynamic linker.
L		<code>addiu.b32 \$a0, \$gp, %tlsdesc_ got(symbol) balc %tlsdesc_ call(symbol)</code>	8/10	<code>R_NANOMIPS_ symbol; R_NANOMIPS_ symbol</code>	<code>GOT: R_NANOMIPS_ symbol; GOT: 0</code>	The two GOT entries will get filled by the dynamic linker.

`R_NANOMIPS_TLS_DESC_GOT` and `R_NANOMIPS_TLS_DESC_GOT_I32` are filled by the static linker with the address of the first GOT entry used for the specified TLS symbol.

`R_NANOMIPS_TLS_DESC_CALL` is filled by the static linker with the address of the linker-generated local resolver stub.

If the symbol is in an initially loaded module, `R_NANOMIPS_TLS_DESC` will cause the dynamic linker to fill the first GOT entry with zeros and the second entry with the symbol's TP offset. If it is not, `R_NANOMIPS_TLS_DESC` will cause the dynamic linker to fill the first entry with the address of the dynamic TLS descriptors resolver and the second entry with a pointer to an argument for the resolver which will help it resolve the symbol's address.

Resolver stub

The static linker will generate a local stub which will either call the dynamic resolver and return the resulting address, or it will calculate the symbol's address directly, using the TP offset patched in by the dynamic linker.

The stub must be generated in every module which uses TLS descriptors.

```
__tlsdesc_stub:
    lw $t9, 0($a0)
    beqzc $t9, 1f
    jrc $t9
1:
    lw $a0, 4($a0)
    rdhwr $t9, $29
    addu $a0, $t9, $a0
    jrc $ra
```

In the above implementation, the stub first checks the contents of the first GOT entry. If it is filled with zeros, it loads the contents of the second GOT entry (i.e. the symbol's TP offset), adds it to the TP, and returns it. If the first entry is non-zero, the stub calls the function at the address contained in the entry (i.e. the dynamic TLS descriptors resolver), passes it the same argument it received, and returns the address returned by the dynamic resolver.

The total size of this stub is 20 bytes.

7.3 List of traditional TLS code sequences

Global Dynamic

Model	Conditions	Code sequence	Size	Static relocations	Dynamic relocations	Remarks
AM	within GP +2MiB	<code>addiu.w \$a0, \$gp, %t1sgd(symbol)</code> <call to <code>__tls_get_addr</code> >	4	<code>R_NANOMIPS_</code> symbol	GOT[n]: <code>R_NANOMIPS_</code> symbol; GOT[n+1]: <code>R_NANOMIPS_</code> symbol	The address of the symbol will be returned in \$a0 by <code>__tls_get_addr</code> .
L		<code>addiu.b32 \$a0, \$gp, %t1sgd(symbol)</code> <call to <code>__tls_get_addr</code> >	6	<code>R_NANOMIPS_</code> symbol	GOT[n]: <code>R_NANOMIPS_</code> symbol; GOT[n+1]: <code>R_NANOMIPS_</code> symbol	

Local Dynamic

Model	Conditions	Code sequence	Size	Static relocations	Dynamic relocations	Remarks
TLS Block Address						
AM	within GP +2MiB	<code>addiu.w \$a0, \$gp, %t1sld(symbol)</code> <call to <code>__tls_get_addr</code> >	4	<code>R_NANOMIPS_</code> symbol	GOT[n]: <code>R_NANOMIPS_</code> symbol; GOT[n +1]: 0	The address of the symbol will be returned in \$a0 by <code>__tls_get_addr</code> .

Model	Conditions	Code sequence	Size	Static relocations	Dynamic relocations	Remarks
L		addiu.b32 \$a0, \$gp, %tlsld(symbol); <call to __tls_get_addr>	6	R_NANOMIPS_symbol	GOT[n]: R_NANOMIPS_TLS_DTPMOD symbol; GOT[n+1]: 0	
Variable Address Calculation						
AM	within DTP +64KiB	addiu reg, reg_dtp, %dtprel(symbol)	4	R_NANOMIPS_TLS_DTPREL16 symbol		reg_dtp is a GPR which contains the module address returned by __tls_get_addr.
L		li48 reg, %dtprel(symbol); addu reg, reg, reg_dtp	8/10	R_NANOMIPS_TLS_DTPREL_I32 symbol		
Variable Access						
A	within DTP +4KiB	[ls] reg, %dtprel(symbol)(reg_dtp)	4	R_NANOMIPS_TLS_DTPREL12 symbol		
M	within DTP +64KiB	addiu reg1, reg_dtp, %dtprel(symbol); [ls] reg2, 0(reg1)	6/8	R_NANOMIPS_TLS_DTPREL16 symbol		
L		li48 reg, %dtprel(symbol); [ls]x reg2, reg1(reg_dtp)	10	R_NANOMIPS_TLS_DTPREL_I32 symbol		

Initial Exec

Model	Conditions	Code sequence	Size	Static relocations	Dynamic relocations	Remarks
TP Setup						
AML		rdhwr reg_tp, \$29	4			The compiler will choose a GPR for reg_tp as it sees fit. As long as no kernel emulation is involved, it is preferable for the compiler to re-do the setup than to save/restore reg_tp from stack.

Model	Conditions	Code sequence	Size	Static relocations	Dynamic relocations	Remarks
Variable Address Calculation						
AM	within GP +2MiB	lw reg, %gottprel(s (\$gp)); addu reg, reg, reg_ - tp	6/8	R_NANOMIPS_ symbol	GOT[n]: R_NANOMIPS_TLS_TPREL symbol	
L		lwpc reg, %gottprel_ pc- 32(symbol); addu reg, reg, reg_ - tp	8/10	R_NANOMIPS_ PC_I32 symbol	GOT[n]: R_NANOMIPS_TLS_TPREL symbol	
Variable Access						
AM	within GP +2MiB	lw reg1, %gottprel(s (\$gp); [ls]x reg2, reg1(reg_ - tp)	8	R_NANOMIPS_ symbol	GOT[n]: R_NANOMIPS_ symbol	reg_tp must be used as the base register to enable the linker to relax to a Local Exec sequence, if possible.
L		lwpc reg1, %gottprel_ pc- 32(symbol); [ls]x reg2, reg1(reg_ - tp)	10	R_NANOMIPS_ PC_I32 symbol	GOT[n]: R_NANOMIPS_ symbol	reg_tp must be used as the base register to enable the linker to relax to a Local Exec sequence, if possible.

Local Exec

Model	Conditions	Code sequence	Size	Static relocations	Dynamic relocations	Remarks
TP Setup						
AML		rdhwr reg_tp, \$29	4			The compiler will choose a GPR for reg_tp as it sees fit. As long as no kernel emulation is involved, it is preferable for the compiler to re-do the setup than to save/restore reg_tp from stack.
Variable Address Calculation						

Model	Conditions	Code sequence	Size	Static relocations	Dynamic relocations	Remarks
AM	within TP +64KiB	addiu reg, reg_tp, %tprel(symk	4	R_NANOMIPS_TLS_TPREL16 symbol		
		addiu[48] reg, reg_tp, %tprel(symk	6	R_NANOMIPS_TLS_TPREL_I32 symbol		Generated only by the linker. When relaxing Global Dynamic to Local Exec, reg_tp will have to be set up for each sequence. Because we don't have to worry about preserving reg_tp in this case, the linker can relax to the source- destroying ADDIU[48] instead of the LI[48]+ADDU sequence for the large model. This is worth doing only for address calculations.
L		li48 reg, %tprel(symk addu reg, reg, reg_ tp	8/10	R_NANOMIPS_TLS_TPREL_I32 symbol		
Variable Access						
A	within TP+4KiB	[ls] reg, %tprel(symk (reg_tp)	4	R_NANOMIPS_TLS_TPREL12 symbol		
M	within TP +64KiB	addiu reg1, reg_tp, %tprel(symk [ls] reg2, 0(reg1)	6/8	R_NANOMIPS_TLS_TPREL16 symbol		
L		li48 reg, %tprel(symk [ls]x reg2, reg1(reg_ tp)	10	R_NANOMIPS_TLS_TPREL_I32 symbol		

8 nanoMIPS Data Representation

8.1 Byte ordering

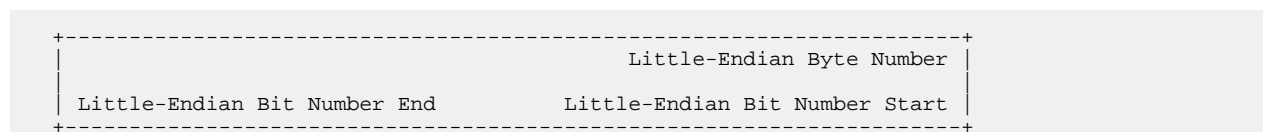
The standard recognizes the following data formats:

- 8-bit byte
- 16-bit halfword
- 32-bit word
- 64-bit doubleword
- 128-bit quadword

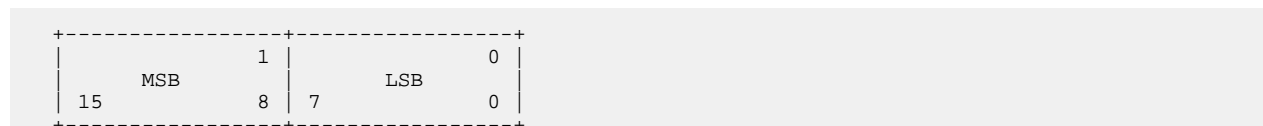
Byte ordering defines how the bytes that make up halfwords, words, doublewords, and quad-words are ordered in memory. Most significant byte (MSB) byte ordering, or big endian as it is sometimes called, means that the most significant byte is located in the lowest addressed byte position in a storage unit (byte 0). The nanoMIPS processor supports either big endian or little endian byte ordering.

The figures below illustrate the conventions for bit and byte numbering within various width storage units. These conventions hold for both integer data and floating-point data, where the most significant byte of a floating-point value holds the sign and at least the start of the exponent.

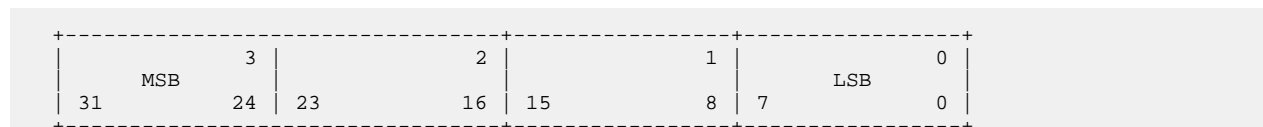
Little-endian bit and byte numbering



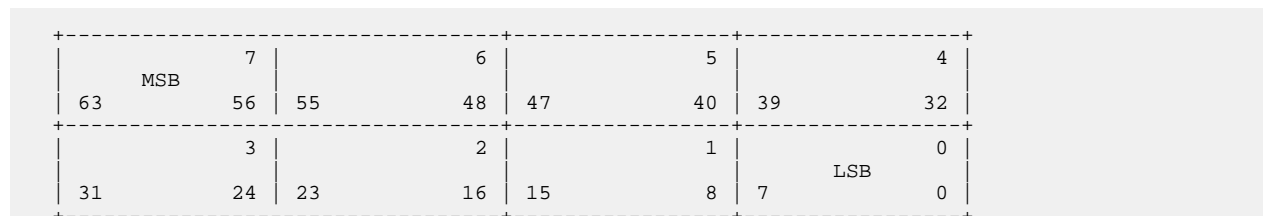
Little-endian bit and byte numbering in half-words



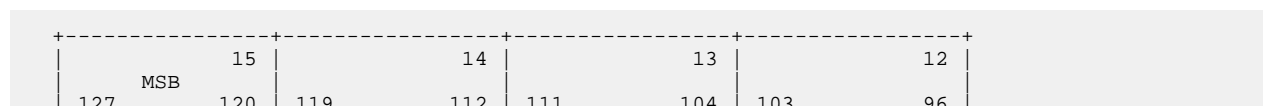
Little-endian bit and byte numbering in words



Little-endian bit and byte numbering in doublewords



Little-endian bit and byte numbering in quadwords



11		10		9		8	
95	88	87	80	79	72	71	64
7		6		5		4	
63	56	55	48	47	40	39	32
3		2		1		0	
31	24	23	16	15	8	7	0
						LSB	0

Big-endian bit and byte numbering

Big-Endian Byte Number	
Big-Endian Bit Number Start	Big-Endian Bit Number End

Big-endian bit and byte numbering in half-words

0	1
MSB	LSB
0	15
7	8

Big-endian bit and byte numbering in words

0	1	2	3
MSB			LSB
0	15	16	31
7	8	23	24

Big-endian bit and byte numbering in doublewords

0	1	2	3
MSB			
0	15	16	31
7	8	23	24
4	5	6	7
			LSB
32	47	48	63
39	40	55	56

Big-endian bit and byte numbering in quadwords

0	1	2	3
MSB			
0	15	16	31
7	8	23	24
4	5	6	7
32	47	48	63
39	40	55	56
8	9	10	11
64	79	80	95
71	72	87	88
12	13	14	15
			LSB
96	111	112	127
103	104	119	120

8.2 Fundamental Types

The following tables shows how the fundamental type in C/C++ types map to the nanoMIPS processor. It can be seen that natural alignment is required for all types.

A null pointer (for all types) has the value zero.

Type	C/C++ type	nanoMIPS architecture	p32	Notes	
			Size (bits)	Alignment (bits)	
Integral	char	unsigned byte	8	8	
	signed char	signed byte	8	8	
	unsigned char	unsigned byte	8	8	
	[signed] short	signed halfword	16	16	
	unsigned short	unsigned halfword	16	16	
	[signed] int	signed word	32	32	
	unsigned int	unsigned word	32	32	
	[signed] long	signed word	32	32	
	unsigned long	unsigned word	32	32	
	[signed] long long	signed doubleword	64	64	Since C99.
	unsigned long long	unsigned doubleword	64	64	Since C99.
	_Bool/bool	unsigned byte	8	8	Since C99 or in C++ +. False as 0, True as 1.
	enum	signed or unsigned word	32	32	
Pointer	type *	data pointer	32	32	
	type (*) ()	code pointer	32	32	
	type&	data pointer	32	32	C++
Floating Point	float	single precision (IEEE 754)	32	32	
	float _Complex	2 single precision (IEEE 754)	64	32	Since C99. Layout as struct { float re, im; };
	double	double precision (IEEE 754)	64	64	
	double _Complex	2 double precision (IEEE 754)	128	64	Since C99. Layout as struct { double re, im; };
	long double	quad precision (IEEE 754-2008)	64	64	

Type	C/C++ type	nanoMIPS architecture	p32		Notes
	long double _Complex	2 quad precision (IEEE 754-2008)	128	64	Since C99. Layout as struct { long double re, im; };

8.3 Aggregates, union and arrays

- The size of any object, including aggregates and unions, is always a multiple of the alignment of the object.
- Structure and union objects can require padding to meet size and alignment constraints.
- An array uses the same alignment as its elements.
- The content of any padding is undefined.
- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require internal padding, depending on the previous member.

Examples

In the following examples, members' byte offsets for little-endian implementations appear in the upper right corners; offsets for big-endian implementations in the upper left corners.

Structure Smaller Than a Word

```
struct {
    char c;
};
byte aligned, sizeof is 1
```

```
+-----+
| 0      0 |
|  c      |
+-----+
```

No Padding

```
struct {
    char c;
    char d;
    short s;
    int n;
};
word aligned, sizeof is 8
```

little endian:

```
+-----+
|          2 |          1 |          0 |
|      s      d      c      |
+-----+
|          4 |
|      n          |
+-----+
```

big endian:

```
+-----+
| 0      1 | 2          |
|  c      d      s      |
+-----+
| 4          |
|      n          |
+-----+
```

Internal Padding

```
struct {
    char c;
    short s;
```

```
};
halfword aligned, sizeof is 4

little endian:
+-----+-----+-----+
|           2 |           1 |           0 |
|           s |         pad |           c |
+-----+-----+-----+

big endian:
+-----+-----+-----+
| 0 | 1 | 2 |
| c | pad | s |
+-----+-----+-----+
```

Internal and Tail Padding

```
struct {
    char c;
    double d;
    short s;
};
doubleword aligned, sizeof is 24

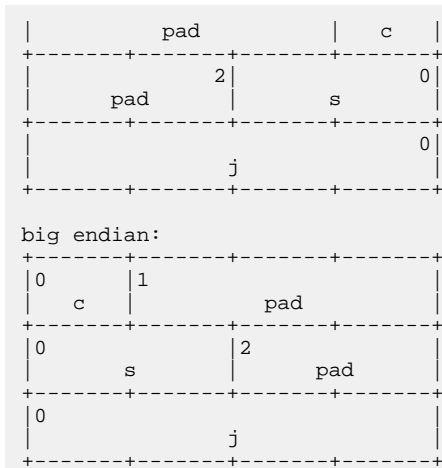
little endian:
+-----+-----+-----+
|           1 |           0 |
|         pad |           c |
+-----+-----+-----+
|           4 |
|         pad |
+-----+-----+-----+
|           8 |
|           d |
+-----+-----+-----+
|          12 |
|           d |
+-----+-----+-----+
|          16 |          18 |
|         pad |           s |
+-----+-----+-----+
|          20 |
|         pad |
+-----+-----+-----+

big endian:
+-----+-----+-----+
| 0 | 1 |
| c | pad |
+-----+-----+-----+
| 4 |
|         pad |
+-----+-----+-----+
| 8 |
|           d |
+-----+-----+-----+
| 12 |
|           d |
+-----+-----+-----+
| 16 |          18 |
|           s |         pad |
+-----+-----+-----+
| 20 |
|         pad |
+-----+-----+-----+
```

Union Allocation

```
union {
    char c;
    short s;
    int j;
};
word aligned, sizeof is 4

little endian:
+-----+-----+
|           1 |           0 |
```



8.4 Bit fields in structures

C allows you to define structures that can have 'bit-fields', that is, a standard integer type with a specified number of bits.

Bit-field type	Width (w)	Range
signed char	1 to 8	$2^{(w-1)}$ to $2^{(w-1)} - 1$
char		0 to $2^{(w-1)}$
unsigned char		0 to $2^{(w-1)}$
signed short	1 to 16	$2^{(w-1)}$ to $2^{(w-1)} - 1$
short		$2^{(w-1)}$ to $2^{(w-1)} - 1$
unsigned short		0 to $2^{(w-1)}$
signed int	1 to 32	$2^{(w-1)}$ to $2^{(w-1)} - 1$
int		$2^{(w-1)}$ to $2^{(w-1)} - 1$
unsigned int		0 to $2^{(w-1)}$
signed long	1 to 32	$2^{(w-1)}$ to $2^{(w-1)} - 1$
long		$2^{(w-1)}$ to $2^{(w-1)} - 1$
unsigned long		0 to $2^{(w-1)}$

Plain bit-fields always have signed or unsigned values depending on whether the basic type is signed or unsigned. In particular, char bit-fields are unsigned while short, int, and long bit-fields are signed. A signed or unsigned modifier overrides the default type.

In a signed bit-field, the most significant bit is the sign bit; sign bit extension occurs when the bit-field is used in an expression. Unsigned bit-fields are treated as simple unsigned values.

Bit-fields follow the same size and alignment rules as other structure and union members, with the following additions:

- Bit-fields are allocated from left to right (most to least significant).
- A bit-field must reside entirely in a storage unit that is appropriate for its declared type. Thus a bit-field never crosses its unit boundary. However, an unnamed bit-field of non-zero width is allocated in the smallest storage unit sufficient to hold the field, regardless of the defined type.
- Bit-fields can share a storage unit with other struct/union members, including members that are not bit-fields. Of course, struct members occupy different parts of the storage unit.
- Unnamed types of bit-fields do not affect the alignment of a structure or union, although member offsets of individual bit-fields follow the alignment constraints.

9 Relocation Types

Relocation Operands

- S - The absolute address of the symbol from the RELA record
- A - The 32-bit addend from the RELA record
- P - The PC that the reloc is being applied to
- G - The value of `_gp`
- TP - The value of the thread pointer for current thread
- DTP - The address of the TLS block for the module that provides S, for current thread.

Relocation Operators

Each of the relocation operators supported by the nanoMIPS ABI is shown below along with the ELF reloc number, the operation to calculate the value and the method of encoding the value in an instruction.

Relocation (Value)	Operation	Bit encoding
R_NANOMIPS_PC21_S1 (14)	offset = (S + A - (P + 4)) is_signed_value (offset, nbits = 22) is_aligned(offset, 2)	op[0] = offset[21], op[20:1] = offset[20:1]
R_NANOMIPS_GOTPC_I32 (34)	offset = (GOT(S + A) - (P + 4)) is_signed_value (offset, nbits = 32)	op[31:0] = offset[31:0]
R_NANOMIPS_GOT_DISP (33)	offset = (GOT(S + A) - G) is_unsigned_value (offset, nbits = 21) is_aligned(offset, 4)	op[20:2] = offset[20:2]
R_NANOMIPS_GOT_CALL (37)	offset = (GOT(S + A) - G) is_unsigned_value (offset, nbits = 21) is_aligned(offset, 4)	op[20:2] = offset[20:2]
R_NANOMIPS_GOT_PAGE (38)	expand to got slot for S + A	none
R_NANOMIPS_GPREL19_S2 (20)	offset = (S + A - G) is_unsigned_value (offset, nbits = 21) is_aligned(offset, 4)	op[20:2] = offset[20:2]
R_NANOMIPS_GPREL7_S2 (25)	offset = (S + A - G) is_unsigned_value (offset, nbits = 9) is_aligned(offset, 4)	op[6:0] = offset[8:2]
R_NANOMIPS_GPREL_HI20 (26)	offset = (S + A - G)	op[0] = offset[31], op[11:2] = offset[30:21], op[20:12] = offset[20:12]
R_NANOMIPS_HI20 (28)	offset = S + A	op[0] = offset[31], op[11:2] = offset[30:21], op[20:12] = offset[20:12]
R_NANOMIPS_I32 (32)	address = S + A, is_signed_value (address, nbits = 32)	op[31:0] = address[31:0]
R_NANOMIPS_GPREL17_S1 (23)	offset = (S + A - G) is_unsigned_value (offset, nbits = 18) is_aligned(offset, 2)	op[17:1] = offset[17:1]
R_NANOMIPS_GPREL16_S2 (24)	offset = (S + A - G) is_unsigned_value (offset, nbits = 18) is_aligned(offset, 4)	op[17:2] = offset[17:2]
R_NANOMIPS_GPREL18_S3 (21)	offset = (S + A - G) is_unsigned_value (offset, nbits = 22) is_aligned(offset, 8)	op[20:3] = offset[20:3]
R_NANOMIPS_GPREL18 (22)	offset = (S + A - G) is_unsigned_value (offset, nbits = 18)	op[17:0] = offset[17:0]

Relocation (Value)	Operation	Bit encoding
R_NANOMIPS_GOT_OFST (39)	expand to got slot for S + A	none
R_NANOMIPS_LO12 (29)	offset = S + A	op[11:0] = offset[11:0]
R_NANOMIPS_GOT_LO12 (36)	offset = GOT(S + A)	op[11:0] = offset[11:0]
R_NANOMIPS_PC7_S1 (18)	offset = (S + A - (P + 2)) is_signed_value (offset, nbits = 8) is_aligned(offset, 2)	op[0] = offset[7], op[6:1] = target[6:1]
R_NANOMIPS_PC11_S1 (16)	offset = (S + A - (P + 4)) is_signed_value (offset, nbits = 12) is_aligned(offset, 2)	op[0] = offset[11], op[10:1] = offset[10:1]
R_NANOMIPS_PC4_S1 (19)	offset = (S + A - (P + 2)) is_unsigned_value (offset, nbits = 5) is_aligned(offset, 2)	op[3:0] = offset[3:0]
R_NANOMIPS_PC14_S1 (15)	offset = (S + A - (P + 4)) is_signed_value (offset, nbits = 15) is_aligned(offset, 2)	op[0] = offset[14], op[13:1] = offset[13:1]
R_NANOMIPS_PC25_S1 (13)	offset = (S + A - (P + 4)) is_signed_value (offset, nbits = 26) is_aligned(offset, 2)	op[0] = offset[25], op[24:1] = offset[24:1]
R_NANOMIPS_PC10_S1 (17)	offset = (S + A - (P + 2)) is_signed_value (offset, nbits = 11) is_aligned(offset, 2)	op[0] = offset[10], op[9:1] = offset[9:1]
R_NANOMIPS_PC_HI20 (27)	offset = (S + A) - ((P + 4) & ~0xfff)	op[0] = offset[31], op[11:2] = offset[30:21], op[20:12] = offset[20:12]
R_NANOMIPS_GOTPC_HI20 (35)	offset = GOT(S + A) - ((P + 4) & ~0xfff)	op[0] = offset[31], op[11:2] = offset[30:21], op[20:12] = offset[20:12]
R_NANOMIPS_PC_I32 (31)	offset = (S + A) - (P + 4) is_signed_value (offset, nbits = 32)	op[31:0] = offset[31:0]
R_NANOMIPS_GPREL_I32 (30)	offset = (S + A - G)	op[31:0] = offset[31:0]
R_NANOMIPS_GPREL_LO12 (42)	offset = (S + A - G)	op[11:0] = offset[11:0]
R_NANOMIPS_LO4_S2 (40)	offset = (S + A)	op[3:0] = offset[5:2]
R_NANOMIPS_HI32 (41)	offset = (S + A)	op[31:0] = offset[63:32]
R_NANOMIPS_TLS_GD (83)	offset = (GOT_TLS(S + A) - G) is_unsigned_value (offset, nbits = 21) is_aligned(offset, 4)	op[20:2] = offset[20:2]
R_NANOMIPS_TLS_GD_I32 (84)	offset = (GOT_TLS(S + A) - G) is_signed_value (offset, nbits = 32)	op[31:0] = offset[31:0]
R_NANOMIPS_TLS_LD (85)	offset = (GOT_TLS_MODULE(S) - G) is_unsigned_value (offset, nbits = 21) is_aligned(offset, 4)	op[20:2] = offset[20:2]
R_NANOMIPS_TLS_LD_I32 (86)	offset = (GOT_TLS_MODULE(S) - G) is_signed_value (offset, nbits = 32)	op[31:0] = offset[31:0]
R_NANOMIPS_TLS_DTPREL12 (87)	offset = (S - DTP) is_unsigned_value(offset, nbits = 12)	op[11:0] = offset[11:0]
R_NANOMIPS_TLS_DTPREL16 (88)	offset = (S - DTP) is_unsigned_value(offset, nbits = 16)	op[15:0] = offset[15:0]
R_NANOMIPS_TLS_DTPREL_I32 (89)	offset = (S - DTP) is_signed_value(offset, nbits = 32)	op[31:0] = offset[31:0]
R_NANOMIPS_TLS_GOTTPREL (90)	offset = (GOT_TLS(S + A) - G) is_unsigned_value (offset, nbits = 21) is_aligned(offset, 4)	op[20:2] = offset[20:2]
R_NANOMIPS_TLS_GOTTPREL_PC32 (91)	offset = (GOT_TLS(S + A) - (P + 4)) is_signed_value(offset, nbits = 32)	op[31:0] = offset[31:0]

Relocation (Value)	Operation	Bit encoding
R_NANOMIPS_TLS_TPREL12 (92)	offset = (S - TP) is_unsigned_value(offset, nbits = 12)	op[11:0] = offset[11:0]
R_NANOMIPS_TLS_TPREL16 (93)	offset = (S - TP) is_unsigned_value(offset, nbits = 16)	op[15:0] = offset[15:0]
R_NANOMIPS_TLS_TPREL_I32 (94)	offset = (S - TP) is_signed_value(offset, nbits = 32)	op[31:0] = offset[31:0]

The set of instructions compatible with each relocation operator is shown below along with example usage.

Relocation	Instructions	Usage
R_NANOMIPS_PC21_S1	MOVE.BALC LAPC[32]	MOVE.BALC reg, reg, symbol Note: This is for both branches and data reference.
R_NANOMIPS_GOTPC_I32	LWPC[48], LDPC[48]	LWPC reg, %got_pcrel32(symbol)
R_NANOMIPS_GOT_DISP	LWGP, LDGP	LW reg, %got_disp(symbol)(gp)
R_NANOMIPS_GOT_CALL	LWGP, LDGP	LW reg, %got_call(symbol)(gp)
R_NANOMIPS_GOT_PAGE	LWGP, LDGP	LW reg, %got_page(symbol)(gp)
R_NANOMIPS_GPREL19_S2	LW[GP], SW[GP], ADDIU.W	LW reg, %gpel(symbol)(gp)
R_NANOMIPS_GPREL7_S2	LW[GP16], SW[GP16]	LW16 reg, %gpel(symbol)(gp)
R_NANOMIPS_GPREL_HI20	LUI	LUI reg, %gpel_hi(symbol)
R_NANOMIPS_HI20	LUI	LUI reg, %hi(symbol)
R_NANOMIPS_I32	LI[48]	LI reg, symbol
R_NANOMIPS_GPREL17_S1	LH[GP], LHU[GP], SH[GP]	LH reg, %gpel(symbol)(gp)
R_NANOMIPS_GPREL16_S2	LDC1[GP], LWC1[GP], SDC1[GP], SWC1[GP], LWU[GP]	LHU reg, %gpel(symbol)(gp)
R_NANOMIPS_GPREL18_S3	LD[GP], SD[GP]	LD reg, %gpel(symbol)(gp)
R_NANOMIPS_GPREL18	LB[GP], LBU[GP], SB[GP], ADDIU.B	LB reg, %gpel(symbol)(gp)
R_NANOMIPS_GOT_OFST	LB, LBU, LD, LH, LHU, LW, LWU, SB, SD, SH, SW, LDC1, LWC1, SDC1, SWC1	LW reg, %got_ofst(symbol)(reg)
R_NANOMIPS_LO12	LB, LBU, LD, LH, LHU, LW, LWU, SB, SD, SH, SW, LDC1, LWC1, SDC1, SWC1, ORI, ADDIU	ORI reg, reg, %lo(symbol) Notes: Do not use with ADDIU from compiled code, use ORI for power efficiency. ADDIU may be necessary for compatible ASM code with pre-nanoMIPS.
R_NANOMIPS_GOT_LO12	LB, LBU, LD, LH, LHU, LW, LWU, SB, SD, SH, SW, LDC1, LWC1, SDC1, SWC1, ORI	ORI reg, reg, %got_lo(symbol)
R_NANOMIPS_PC7_S1	BEQZC16, BNEZC16	BEQZC16 reg, symbol
R_NANOMIPS_PC11_S1	BEQIC, BGEIC, BGEUIC, BNEIC, BLTIC, BLTUIC	BEQIC reg, imm, symbol
R_NANOMIPS_PC4_S1	BEQC16, BNEC16 (forward branch)	BEQC16 reg, reg, symbol
R_NANOMIPS_PC14_S1	BC2EQZC, BC2NEZC, BC1EQZC, BC1NEZC, BEQC, BGEC, BGEUC, BNEC, BLTC, BLTUC	BEQC reg, reg, symbol
R_NANOMIPS_PC25_S1	BALC, BC	BC symbol
R_NANOMIPS_PC10_S1	BALC16, BC16	BC16 symbol

Relocation	Instructions	Usage
R_NANOMIPS_PC_HI20	ALUIPC	ALUIPC reg, %pcrel_hi(symbol)
R_NANOMIPS_GOTPC_HI20	ALUIPC	ALUIPC reg, %got_pcrel_hi(symbol)
R_NANOMIPS_PC_I32	LAPC[48]	ADDIUPC reg, %pcrel32(symbol)
R_NANOMIPS_GPREL_I32	ADDIU[GP48]	ADDIU.B32 reg, %gpel(symbol)
R_NANOMIPS_GPREL_LO12	LB, LBU, LD, LH, LHU, LW, LWU, SB, SD, SH, SW, LDC1, LWC1, SDC1, SWC1, ORI, ADDIU	ORI reg, reg, %gpel_lo(symbol) Notes: Do not use with ADDIU from compiled code, use ORI for power efficiency. ADDIU may be necessary for compatible ASM code with pre-nanoMIPS.
R_NANOMIPS_TLS_GD	LW[GP], SW[GP], ADDIU.W	ADDIU.W reg, gp, %tlsd(symbol)
R_NANOMIPS_TLS_GD_I32	ADDIU[GP48]	ADDIU.B32 reg, gp, %tlsd(symbol)
R_NANOMIPS_TLS_LD	LW[GP], SW[GP], ADDIU.W	ADDIU.W reg, gp, %tlsd(symbol)
R_NANOMIPS_TLS_LD_I32	ADDIU[GP48]	ADDIU.B32 reg, gp, %tlsd(symbol)
R_NANOMIPS_TLS_DTPREL12	LB, LBU, LD, LH, LHU, LW, LWU, SB, SD, SH, SW, LDC1, LWC1, SDC1, SWC1	LW reg, %dtprel(symbol)(reg_dtp)
R_NANOMIPS_TLS_DTPREL16	ADDIU	ADDIU reg, reg_dtp, %dtprel(symbol)
R_NANOMIPS_TLS_DTPREL_I32	LI	LI reg, %dtprel(symbol)
R_NANOMIPS_TLS_GOTTPREL	LWGP, LDGP	LW reg, %gottprel(symbol)(gp)
R_NANOMIPS_TLS_GOTTPREL_PC32	LWPC[48], LDPC[48]	LWPC reg, %got_pcrel32(symbol)
R_NANOMIPS_TLS_TPREL12	LB, LBU, LD, LH, LHU, LW, LWU, SB, SD, SH, SW, LDC1, LWC1, SDC1, SWC1	LW reg, %tprel(symbol)(reg_tp)
R_NANOMIPS_TLS_TPREL16	ADDIU	ADDIU reg, reg_tp, %tprel(symbol)
R_NANOMIPS_TLS_TPREL_I32	LI	LI reg, %tprel(symbol)

Data Relocations

Relocation (Value)	Operation	Bit encoding
R_NANOMIPS_32 / R_NANOMIPS_WORD (1)	S + A	=> target[31:0]
R_NANOMIPS_64 / R_NANOMIPS_DWORD (2)	S + A	=> target[63:0]
R_NANOMIPS_NEG (3)	-S + A	=> target[ABI_WIDTH]
R_NANOMIPS_ASHIFTR_1 (4)	sign_extend ((S + A) >> 1)	=> target[ABI_WIDTH]
R_NANOMIPS_UNSIGNED_8 (5)	S + A	=> target[7:0]
R_NANOMIPS_SIGNED_8 (6)	S + A	=> target[7:0]
R_NANOMIPS_UNSIGNED_16 (7)	S + A	=> target[15:0]
R_NANOMIPS_SIGNED_16 (8)	S + A	=> target[15:0]
R_NANOMIPS_RELATIVE (9)	S + A	=> target[ABI_WIDTH]
R_NANOMIPS_GLOBAL (10)	S + A	=> target[ABI_WIDTH]
R_NANOMIPS_JUMP_SLOT (11)	lazy_stub (S)	=> target[ABI_WIDTH]
R_NANOMIPS_IRELATIVE (12)	ifunc_resolver (S)	=> target[ABI_WIDTH]
R_NANOMIPS_COPY (44)	copy_data (S)	=> variable, sizeof (S)

Relocation (Value)	Operation	Bit encoding
R_NANOMIPS_TLS_DTPMOD (80)	tls_module_number (S)	=> target[ABI_WIDTH]
R_NANOMIPS_TLS_DTPREL (81)	tls_module_offset (S)	=> target[ABI_WIDTH]
R_NANOMIPS_TLS_TPREL (82)	tls_static_offset (S)	=> target[ABI_WIDTH]

Place-holder Relocations for Linker Relaxations

Relocation (Value)	Symbol encoding (value)	Description
R_NANOMIPS_ALIGN (64)	alignment	Requested alignment
R_NANOMIPS_FILL (65)	fill-value	Fill value, 8/16/32 bits is encoded in symbol size.
R_NANOMIPS_MAX (66)	max-fill	
R_NANOMIPS_INSN32 (67)	none	Inhibit relaxation to 16-bits, out-of-range expansion is possible.
R_NANOMIPS_FIXED (68)	none	No relaxation or expansion at this instruction.
R_NANOMIPS_NORELAX (69)	none	Inhibit relaxation/expansion until subsequent RELAX relocation
R_NANOMIPS_RELAX (70)	none	Activate relaxation/expansion until subsequent NORELAX relocation
R_NANOMIPS_SAVERESTORE (71)	function symbol	Placeholder relocation for 32-bit save/restore instructions with GP-bit in PIC mode.
R_NANOMIPS_INSN16 (72)	none	Inhibit relaxation to 32-bits.
R_NANOMIPS_JALR32 (73)	none	Hint for a 32-bit jump-and-link instruction to allow it to be relaxed for locally resolved symbols to an unconditional branch.
R_NANOMIPS_JALR16 (74)	none	Hint for a 16-bit jump-and-link instruction to allow it to be relaxed for locally resolved symbols to an unconditional branch.
R_NANOMIPS_JUMPTABLE_LOAD (75)	jump-table symbol	Hint for a scaled load instruction from a jump-table to be modified to match type-relaxation in the jump-table.

10 nanoMIPS Appendix

Virtual Tables

Representation of pointers to member functions

A pointer-to-function member type looks like:

```
struct {  
    __P __pfn;  
    ptrdiff_t __delta;  
}
```

The MIPS/microMIPS/MIPS16 ISAs use the lowest bit in addresses to indicate the ISA mode, hence, the lowest bit of `__pfn` could not be used to indicate whether the function that will be called through a pointer-to-member-function is virtual. In case of MIPS/microMIPS/MIPS16, this bit was encoded in `__delta` and the `__delta` shifted by to the left to make space for it. As the nanoMIPS ISA does not use the lowest bit in addresses to indicate the ISA mode, the lowest bit in `__pfn` is now used to mark a pointer-to-member-function as virtual or not.

Reserved TRAP, BREAK and SYSCALL codes

The nanoMIPS architecture does not raise an exception from the `DIV` or `UDIV` instructions when dividing by zero and instead requires additional code to detect and raise an exception. There are two mechanisms to report a divide by zero exception; either `trap` with code 7 or `break` with code 7. These codes should be avoided by other software.

The Unified Hosting Interface (UHI) is supported for the nanoMIPS architecture and uses a `SYSCALL 1` instruction as the trigger to request servicing of a semi-hosting operation from runtime support code. For compatibility with UHI other software should avoid using `SYSCALL` with code 1.