

Codescape GNU tools for nanoMIPS

P32 Porting Guide

Revision: 1.2
01/05/2018
Public



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.

DN00184

Contents

List of Tables.....	iv
1 Introduction.....	5
1.1 Conventions.....	5
1.2 Support.....	5
2 p32 ABI Overview.....	7
2.1 What is p32 ABI.....	7
2.2 p32 porting requirements.....	7
2.3 o32/p32 ABI comparison.....	7
3 Register Naming Convention.....	9
3.1 CPU registers.....	9
4 Porting Notes.....	11
4.2 ABI/ISA Issues.....	11
4.2.1 Detecting the target API and ISA.....	11
4.2.2 ISA mode.....	11
4.2.3 Calling convention.....	11
4.3 Abstracting register usage.....	12
4.4 Abstracting instruction differences.....	12
4.4.1 Immediates.....	12
4.4.2 Compatible call sequences.....	13
4.4.3 Removal of branch delay slots.....	13
4.5 Linker transformations.....	14
4.5.1 Code safety written in assembly.....	15
4.5.2 Instruction size enforcement.....	15
4.6 Data addressing.....	16
4.7 ELF sections.....	16
4.8 Compiler intrinsics.....	16
5 Revision history.....	17

List of Tables

Table 1: ABI comparison.....	7
Table 2: CPU Register Naming Comparison.....	9
Table 3: Removing branch delay slot examples.....	13

1 Introduction

This publication provides an introduction to the p32 High Performance 32-bit Application Binary Interface (ABI).

Note: This document is a preliminary release and as such is incomplete and contains information that is subject to change although every effort has been made to ensure the contents are accurate at the time of writing. Future versions of this document will provide more comprehensive information.

Related publications

The following documents contain information that may be useful to read in conjunction with this document:

- *'nanoMIPS32 Instruction Set Technical Reference Manual'*
- *'nanoMIPS32 Programmer's Guide'*
- *'nanoMIPS32 Privileged Resource Architecture'*

Obtaining publications

You can obtain MIPS documentation in the following ways:

- <https://www.mips.com/develop/>
- You can also view release notes by typing either `grelnotes` or `relnotes` on a command line.
- You can also view man pages by typing `man title` on a command line.

1.1 Conventions

The following conventions are used throughout this document:

Convention	Meaning
<code>command</code>	Denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
<code><Variable></code>	Denotes variable entries and words or concepts being defined.
<code>[]</code>	Brackets enclose optional portions of a command or directive line.
<code>...</code>	Ellipses indicate that a preceding element can be repeated.
<code>%command</code>	Indicates that <code>command</code> is being entered at a command prompt in a terminal. The <code>%</code> is the command prompt.

1.2 Support

Support for the nanoMIPS Toolchain is available from a variety of sources. Informal, community-based support can be found on the forum. Registered licensees can get support via email and Partner Portal.

Support via forum

<https://www.mips.com/forums/>

General information about Linux on MIPS

<http://www.linux-mips.org>

Partner portal

Support for registered licensees is available via the Partner Portal; <https://partnerportal.mips.com>

2 p32 ABI Overview

This section describes the basic features of the p32 32-bit Application Binary Interface (ABI) for the nanoMIPS architecture.

2.1 What is p32 ABI

The Application Binary Interface, or ABI, is the set of rules that all binaries must follow in order to run on a nanoMIPS system. This includes, for example, object file format, instruction set, data layout, subroutine calling convention, and system call numbers. The ABI is one part of the mechanism that maintains binary compatibility across all nanoMIPS platforms.

p32 improves on o32 to provide an ABI that is efficient in both code density and performance. p32 is required for the nanoMIPS architecture.

2.2 p32 porting requirements

In order to implement p32, MIPS provides the following:

- A compiler that supports p32.
- p32 versions of each library.

To take advantage of the p32 ABI, customers must:

- Install a p32 OS, compiler, and all p32 libraries.
- Rewrite Assembly code to conform to p32 guidelines.
- Recompile all the code with the compiler that supports p32; this will require makefile modifications due to changes in command line options but default settings of a nanoMIPS compiler will automatically target p32 ABI.

2.3 o32/p32 ABI comparison

Table 1: ABI comparison

	o32	p32
Compiler used	gcc/llvm	gcc
Integer model	ILP32	ILP32
Calling convention	o32	new
Width of GP registers	32 bits	32 bits
Number of GP registers	32	32
Number of GP argument registers	4 (\$4..\$7)	8 (\$r4..\$r11)
Number of GP return registers	2 (\$2..\$3)	2 (\$r4..\$r5)
Width of FP registers	32/64 bits	64 bits
Number of FP registers	16 (FR=0) or 32 (FR=1)	32
Number of FP argument registers	4	8
Stack region alignment	8 bytes	16 bytes
Stack slot size for GPRs	32 bits	32 bits

	o32	p32
Stack slot size for FPRs	32 bits	64 bits
Debug format	dwarf	dwarf
ISAs supported	MIPS32R1 MIPS32R2 MIPS32R6	nanoMIPS32R6

Note: The FR=0 mode describes an FPU where registers are constructed of 32-bit parts and there are 16 double-precision registers. The double-precision registers exist at even indices and their upper half exists in the odd indices.

3 Register Naming Convention

This section describes the p32 ABI register naming convention and how it differs from the o32 ABI.

3.1 CPU registers

The nanoMIPS ISA specifies 32 general purpose 32-bit registers and a 32-bit program counter register. The general registers have the names `$r0..$r31`. Note that the old format `$0..$31` is no longer supported by default and an error will be emitted by the assembler. An assembler option, `-mlegacyregs`, allows to use MIPS style numeric registers format in nanoMIPS assembly. Users are encouraged to use the new style.

The table shows the names provided by `regdef.h` header and those (in brackets) supported directly by an assembler.

Table 2: CPU Register Naming Comparison

o32 Reg#	o32 Name	o32 Use (where different)	p32 Reg#	p32 Name	p32 Use	Type
\$0	zero (\$zero)		\$r0	zero (\$zero)	Hardware zero	
\$1	AT (\$at)		\$r1	AT (\$at) (also linker temp)	Assembler temporary	Caller-saved
\$2	v0 (\$v0)	Function results	\$r2	t4 (\$t4)	Temporaries	Caller-saved
\$3	v1 (\$v1)		\$r3	t5 (\$t5)		
\$4	a0 (\$a0)	Function arguments	\$r4	a0 (\$a0)	Function arguments / function results / temporary	Caller-saved
\$5	a1 (\$a1)		\$r5	a1 (\$a1)		
\$6	a2 (\$a2)		\$r6	a2 (\$a2)	Function arguments / temporary	Caller-saved
\$7	a3 (\$a3)		\$r7	a3 (\$a3)		
\$8	t0 (\$t0)	Temporary	\$r8	a4 (\$a4)		
\$9	t1 (\$t1)		\$r9	a5 (\$a5)	Temporary	Caller-saved
\$10	t2 (\$t2)		\$r10	a6 (\$a6)		
\$11	t3 (\$t3)		\$r11	a7 (\$a7)		
\$12	t4 (\$t4)		\$r12	t0 (\$t0)		
\$13	t5 (\$t5)		\$r13	t1 (\$t1)	Temporary	Caller-saved
\$14	t6 (\$t6)		\$r14	t2 (\$t2)		
\$15	t7 (\$t7)		\$r15	t3 (\$t3)		
\$16-\$23	s0-s7 (\$s0-\$s7)		\$r16-\$r23	s0-s7 (\$s0-\$s7)	Saved temporaries	Callee-saved
\$24	t8 (\$t8)		\$r24	t8 (\$t8)	Temporary	Caller-saved
\$25	t9 (\$t9)		\$r25	t9 (\$t9)		

o32 Reg#	o32 Name	o32 Use (where different)	p32 Reg#	p32 Name	p32 Use	Type
\$26-\$27	k0-k1 (\$k0-\$k1)		\$r26-\$r27	k0-k1 (\$k0-\$k1)	Kernel reserved registers	
\$28	gp (\$gp)		\$r28	gp (\$gp)	Global pointer	Callee-saved
\$29	sp (\$sp)		\$r29	sp (\$sp)	Stack pointer	Callee-saved
\$30	fp (\$fp)		\$r30	fp (\$fp)	Frame pointer / saved temporary	Callee-saved
\$31	ra (\$ra)		\$r31	ra (\$ra)	Return register	Callee-saved

4 Porting Notes

This section contains notes to assist making the transition to the nanoMIPS architecture and porting code from o32 ABI to p32 ABI so that foreseeable problems can be pre-empted.

4.2 ABI/ISA Issues

4.2.1 Detecting the target API and ISA

The pre-processor provides macros to enable users to detect what the target architecture is and the software configuration in use. An example of various macros is below:

```
#ifndef __mips__
// This is only enabled for MIPS, microMIPS or MIPS16 architectures
#endif
#ifndef __nanomips__
// This is only enabled for nanoMIPS
#endif
#ifndef __mips__
// Strictly this is not a test for architecture but instead a
// feature of an architecture. __mips is set for all MIPS architecture
// variations and will have a value of 32 for nanoMIPS to represent nanoMIPS32
#endif
#if __mips_isa_rev == 6
// This shows that the release of the architecture is version 6 and is set for
// all MIPS architectures. It is guaranteed to be a minimum of 6 for nanoMIPS
#endif
#if _MIPS_SIM == _ABIO32
// This shows that the ABI in use is o32. It is not possible to use o32 with
// nanoMIPS
#endif
#if _MIPS_SIM == _ABIP32
// This shows that the ABI in use is p32. This is the only possibility for
// nanoMIPS32
#endif
```

4.2.2 ISA mode

The nanoMIPS architecture does not include the concept of an ISA mode. Unlike microMIPS and MIPS16 there is no meaning applied to the least significant bit of a code address and it is interpreted as part of the target address and will trigger an address exception if set.

4.2.3 Calling convention

The p32 ABI offers up to eight general-purpose registers, \$a0-\$a7, that are used to pass arguments to a function and up to 2 general-purpose registers for the return values, \$a0 and \$a1. A function argument can be a scalar, a union or an aggregate with the size up to 2 register widths i.e. 64-bits for p32 ABI. Larger objects are passed by reference and in a case of a large return value (larger than 64-bits) a pointer is passed as a hidden first argument to a function. Structures and union objects can require padding to meet size and alignment constraints. Please refer to the '*Codescape GNU Tools for nanoMIPS ELF ABI Supplement Guide*' for further details.

The stack alignment for p32 must be maintained at a 16-byte alignment at all times. These rules are generally sufficient for writing assembly coded functions and the remaining rules are generally only important for compiled code. To help with porting code between different ABIs there is a header `mips/asm.h` that provides several macros that are defined depending on the ABI. For stack alignment `ALSZ`, `ALMASK` and `LOG2_STACK_ALGN` provide constants that help round a stack adjustment up to an appropriate boundary:

```
addiu    $a0, $a0, ALSZ          /* Round to stack alignment */
and      $a0, $a0, ALMASK
```

A more efficient but less intuitive way of rounding to stack alignment:

```
addiu    $a0, $a0, ALSZ          /* Round to stack alignment */
ins      $a0, $zero, 0, LOG2_STACK_ALGN
```

4.3 Abstracting register usage

The p32 ABI reassigns the purpose of some historically standard registers. In particular the return registers are no longer v0 and v1 but instead a0 and a1. This means that a function returns values in argument registers rather than other dedicated registers allowing more efficient data passing between nested function calls.

In order to avoid silent bugs when porting code from o32 to p32 the v0/v1 register names are not supported in p32. Instead, new register aliases, va0/va1 and vt0/vt1 are provided in `regdef.h` for various use-cases:

va0/va1

In o32 these names represent the two return registers and their names indicate that they are either 'v' (o32) registers or 'a' (p32) registers. Care must be taken when using them to avoid conflicting use of a0/a1 which alias with them in the p32 ABI. Given that v0/v1 are not accepted in p32 it is not possible to create conflicting uses with v0/v1.

vt0/vt1

These names represent two temporary registers and their names indicate that they are either 'v' (o32) registers or 't' (p32) registers. These should be used when porting code that uses v0/v1 as temporary registers.

When porting code that uses v0/v1 the following approach is recommended:

1. If the final values in v0/v1 are to return from the function then rename the last use of v0/v1 to va0/va1. Check that this does not create a conflict with a0/a1 between the last set of va0/va1 and the return point.
2. Rename the remaining instances of v0/v1 to vt0/vt1.
3. If the code needs to be usable with another MIPS architecture, where a function returns its first argument it is necessary to "move va0, a0" and/or "move va1, a1". This will lead to a NOP-like instruction for p32 where argument and return registers are the same but will be a useful instruction for other ABIs.

4.4 Abstracting instruction differences

4.4.1 Immediates

The nanoMIPS instruction set has changed the width of immediate values supported by many instructions. In general the width has decreased and the previous standard 16-bit immediate field is gone. The porting advice here is designed to achieve functional code on all MIPS architectures but is not necessarily most optimal; for optimal code it will be necessary to create nanoMIPS specific implementations and to reconsider certain design decisions where immediate values are concerned.

Many instructions will automatically expand themselves to multi-instruction sequences if an immediate value is out of range. This behaviour requires 'macro' support to be enabled in the assembler which is true by default and also controlled via the `.set` macro and `.set nomacro` directives.

In some cases the ISA has both a register form and an immediate form of an instruction (like `addu` and `addiu`).

`addiu` is an immediate form and requires two register operands and one immediate. When using the immediate form of an instruction the range of the immediate will be checked and an error raised if it is out of range. This is not dependent on `.set macro`.

`addu` is the register form of the instruction and requires three register operands but will accept an immediate final operand. The register form requires macro expansion support when using an immediate as the final operand. If the final operand is an in-range immediate, the result will be a single instruction that is the immediate form of the same instruction. If the final operand is an out-of-range immediate, the instruction will be expanded using the AT register as a temporary to build the immediate value.

For non-performance critical code where the immediate values are out of range switching to use the register form of an instruction is the best course of action.

4.4.2 Compatible call sequences

The o32 ABI mandates the use of `J` or `JAL` to make a call and the assembler automatically deals with a large number of special addressing modes to ensure these instructions behave according to the current settings.

nanoMIPS also supports these instructions as macros solely as a compatibility measure because they do not exist in the ISA itself. For `J` and `JAL` to be useful as an abstraction then the assembler must be operating in `.set macro` `.set reorder` mode. Where code does not need to build with o32 then `J` should be converted to `BC` and `JAL` should be converted to `BALC`.

4.4.3 Removal of branch delay slots

nanoMIPS exclusively uses compact branch and jump instructions meaning there are no delay slots to fill. This is a major difference from MIPS32 although the concept is taken from additions to the MIPS32R6 architecture.

MIPS assemblers allow users to either fill the delay slot of a branch explicitly or leave the assembler to fill it using available instructions. The nanoMIPS assembler includes support to help transition from MIPS to nanoMIPS by automatically converting delay slot branches to compact branches. This transformation is only possible when the assembler is allowed to fill the delay slots itself or a user has explicitly filled a delay slot with a `NOP`. The assembler supports the `.set reorder` directive where it automatically fills delay slots, or the `.set noreorder` directive where it assumes the user has filled delay slots.

Portable code must be written to work in `.set reorder` mode where the assembler fills delay slots. In almost all cases it is possible to do this without losing any delay slot optimisations:

Table 3: Removing branch delay slot examples

Non-portable code example	Portable code example	Comments
<pre>.set push .set noreorder bne \$t0, \$t1, 1f nop .set pop</pre>	<pre>bne \$t0, \$t1, 1f</pre>	If the instruction in the delay slot is <code>nop</code> , its usually a case of simply removing the <code>nop</code> .
<pre>.set push .set noreorder bne \$t0, \$t1, 1f move \$a0, \$t1 .set pop</pre>	<pre>move \$a0, \$t1 bne \$t0, \$t1, 1f</pre>	If the branch condition is independent of the destination register of the instruction in the delay slot, move the delay slot instruction before the branch. For MIPS/microMIPS/MIPS16 the assembler will be able to reorder that instruction back into the delay slot avoiding any loss of performance.

Non-portable code example	Portable code example	Comments
<pre>.set push .set noreorder bne \$t0, \$t1, 1f move \$t0, \$a1 do_something \$t0 .set pop ... 1: move \$t0, \$a2</pre>	<pre>#ifndef __nanomips__ /* allow the move into the delay slot to optimise the fast path */ .set push .set noreorder #endif bne \$t0, \$t1, 1f move \$t0, \$a1 #ifndef __nanomips__ .set pop #endif do_something \$t0 ... 1: move \$t0, \$a2</pre>	<p>If the destination register of the instruction in the delay slot is only useful for the branch-not-taken case, and is in the delay slot to optimise the fast path, then you can safely move the delay slot instruction after the branch, even if the branch condition is dependent on the destination register.</p> <p>Consider keeping the <code>.set noreorder</code> and making it conditional on <code>#ifndef __nanomips__</code> so it remains in the delay slot on non-nanoMIPS architectures. That obviously makes the code less understandable as the <code>move</code> is only in the delay slot on non-nanoMIPS, so consider adding comments and limiting the scope of the <code>noreorder</code> to the branch and delay slot so it's hard for a human to miss.</p> <p>In this example it is easy to prove since that register (<code>\$t0</code>) is overwritten before use after the branch target, and is used in the branch-not-taken case. Sometimes it is less clear without deeper analysis.</p>
<pre>.set push .set noreorder bne \$t0, \$t1, 1f move \$t0, \$a1 .set pop ... 1: do_something \$t0</pre>	<pre>move \$t2, \$a1 bne \$t0, \$t1, 1f ... 1: do_something \$t2</pre>	<p>If the branch condition is dependent on the destination register of the instruction in the delay slot, the delay slot instruction cannot be simply moved before the branch or it would change the branch condition.</p> <p>Either rearrange the register allocation to avoid the dependency and move the instruction in the delay slot before the branch as above...</p>
<pre>.set push .set noreorder bne \$t0, \$t1, 1f move \$t0, \$a1 .set pop ... 1: do_something \$t0</pre>	<pre>#ifdef __nanomips__ move \$t2, \$t0 move \$t0, \$a1 bne \$t2, \$t1, 1f #else .set push .set noreorder bne \$t0, \$t1, 1f move \$t0, \$a1 .set pop #endif ... 1: do_something \$t0</pre>	<p>If it is difficult to change register allocation across the larger part of the code (for example there aren't many spare temporary registers, other register constraints, or it would unacceptably increase code size in the non-nanoMIPS case) it may be preferable to conditionally rewrite the code using <code>__nanomips__</code> or <code>__mips__</code> - <code>isa_rev == 6</code>.</p>

4.5 Linker transformations

The p32 ABI is designed to support full link-time relaxation and rewriting. Some care is needed to avoid any hard-coded PC-relative offsets at assembly time and/or use appropriate annotation to ensure the code written in assembly text is guaranteed to remain the same after linking.

An expansion may happen during linking if, for instance, a symbol is out-of-range. An example of this is a `BALC` instruction being rewritten in-place to `LAPC+JALRC`. The compiler `long_call` attribute is still supported and can be used to force the behaviour at the compile time. Most users can rely on the default options to expand the code sequences.

The opposite of expansion is relaxation. The linker may transform an instruction into a more compressed one, for instance, it may replace 32-bit BC instruction with 16-bit BC if a symbol is within the range of the 16-bit instruction.

Any instruction with a relocation record pointing at it can be rewritten at the link time unless the feature is disabled.

Relaxations and expansions apply to both code and data references. A capable linker guarantees a successful link for any symbol reference. To find out what transformations the linker does for an executable, use the `--debug=target` to the linker or if GCC driver is used, `--Wl,--debug=target`; this will print all the transformations done on symbols.

This feature can be controlled as follows:

- At the module level, use compiler `-mno-relax` command line option. An object will be marked as non-relaxable. The linker will be forbidden to make any transformation in the object but can apply transformations to other modules (providing that other modules are relaxable). This option has a downside as it is fundamentally incompatible with the default code generation model (`-mmodel=auto`). The default automatic model is dependent on the linker to fix any out-of-range references. Alternatively, `-mmodel=medium` can be used with `-mno-relax` to generate immutable code. For an average size application a non-relaxable medium model will lead to successful. The medium model is comparable to the MIPS model and any out-of-range relocations would have to be fixed manually by using `long_call` attribute. As a last resort, the large model can be used to guarantee a successful link; however, this will have a negative impact on the code size.
- At the assembly code level, use assembler option `--linkrelax` or `.linkrelax` directive to enable the transformations. The relaxations and expansion for code fragments can be then controlled by wrapping the code with `.set nolinkrelax/.set linkrelax` or `.set push/.set pop`. For example:

```
.text
.linkrelax
foo:
<code>
.set nolinkrelax
<code>
.set linkrelax
<code>
```

This is equivalent to:

```
.text
.linkrelax
foo:
<code>
.set push
.set nolinkrelax
<code>
.set pop
<code>
```

Note: The various relaxation options control both the relaxations and expansions.

4.5.1 Code safety written in assembly

Avoiding hard-coded PC-relative offsets is trivially achieved by always branching, jumping and calling to symbols instead of writing absolute offsets. An example of unsafe and safe use of a pc-relative branch is below:

Unsafe code	Safe code
<pre>bc 8 lw \$a0, %gprel(data)(\$gp) andi \$a0, \$a0, 20</pre>	<pre>bc 1f lw \$a0, %gprel(data)(\$gp) andi \$a0, \$a0, 20 1:</pre>

4.5.2 Instruction size enforcement

When it is necessary to guarantee either the use of a specific instruction or force the size of an instruction then this can be achieved using a suffix on the instruction name. Add suffix 16, 32 or 48 to force the assembler to use that size instruction and ensure the linker does not change it. However, this may result in

an assembler or linker error if the instruction is either unavailable or unable to support a relocation applied at final link.

Using an instruction without a specific size leaves the assembler to choose the most efficient instruction in terms of code size; this may not always mean a 16-bit instruction is chosen even when it is available as the presence of a relocation is a strong indicator a wider instruction will be required at link time. The assembler therefore selects instructions to be as representative as possible with respect to what will happen during final link.

For example, here are `nop` instructions with and without valid suffixes:

```
nop // This will be a 16-bit NOP
nop16 // This is guaranteed to be a 16-bit NOP
nop32 // This is guaranteed to be a 32-bit NOP
```

4.6 Data addressing

The p32 ABI is not designed to place data in the text section to achieve either performance or code density. The MIPS16 concept of a literal/constant pool in the text section is not supported.

Instead of using a literal pool, nanoMIPS uses a combination of a 48-bit load immediate instruction that can materialize a 32-bit value or address in a single instruction and a significantly larger optimized data area. The optimized global data area is increased in size from 64KB to 2MB allowing simpler controls to select what is placed in that region. MIPS/microMIPS/MIPS16 would normally recommend only placing data items up to 8-bytes in size in the optimized data area, whereas nanoMIPS can generally cope with data items up to 64-bytes and beyond without exceeding the size limit for an average application

4.7 ELF sections

The p32 ABI no longer places read-only data in `.rdata` section. Instead, the read-only data is placed to the standardised `.rodata` section.

Constructors and destructors e.g. when using GCC's `__attribute__((constructor))`/`__attribute__((destructor))` are now placed in `.init_array`/`.fini_array` sections in place of `.ctors`/`.dtors`, respectively. The same applies to constructors/destructors with priorities, these are placed in `.-init_array.NNNNN` and `.fini_array.NNNNN` sections where NNNNN is the priority. Users may need to update their custom linker scripts to account for these changes.

4.8 Compiler intrinsics

There are no changes to the intrinsics in nanoMIPS. Built-ins `__builtin_mips_*` still apply depending on the availability as before.

5 Revision history

Revision	Date	Description
00.01	29 June 2017	Preliminary release.
01.00	31 January 2018	First release.