MIPS nanoMIPS

GNU Toolchain Getting Started Guide

Revision 1.2
30 Apr 2018
Public

Filename: MIPS_nanoMIPS_GNU_Toolchain_Getting_Started_Guide

Issue status: External

Document status: Approved

Source control revision: 12

Document number: DN00183

Contents

1. Introduction

This is a Getting Started guide for the ‘Codescape GNU tools for nanoMIPS’ (henceforth called ‘nanoMIPS Toolchain’ in this manual).

This document is divided up into the following sections:

Installation	Describes how to manually install a toolchain package.
Building a Simple Program	Describes how to build a simple program using the nanoMIPS toolchain with various options, giving examples.
Running a Program in QEMU	Demonstrates running a simple program on the QEMU simulator.
Debugging using GDB and QEMU	Explains step-by-step how to debug a program running on QEMU with the GDB debugging tool, with an example.

1.1. 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 nanoMIPS	http://www.linux-mips.org
Partner portal	Support for registered licensees is available via the Partner Portal; https://partnerportal.mips.com

2. Installation

The nanoMIPS toolchain is provided as gzip compressed tarballs for manual installation. There are four packages to cover the four supported host platforms:

Table 1: Package names

Host Platform	Filename
CentOS-5 32-bit or later	Codescape.GNU.Tools.Package.2018.04-02.for.nanoMIPS.Bare.Metal.CentOS-5.x86.tar.gz
CentOS-5 64-bit or later	Codescape.GNU.Tools.Package.2018.04-02.for.nanoMIPS.Bare.Metal.CentOS-5.x86_64.tar.gz
Windows 32-bit	Codescape.GNU.Tools.Package.2018.04-02.for.nanoMIPS.Bare.Metal.Windows.x86.tar.gz
Windows 64-bit	Codescape.GNU.Tools.Package.2018.04-02.for.nanoMIPS.Bare.Metal.Windows.x86_64.tar.gz

To install the toolchain decompress and unpack the appropriate package into a folder of your choice. To avoid complications with build systems it is recommended that you do not install to a path where any folder has a space in the name. An example of unpacking the package on CentOS-5 32-bit is:

```
> tar xzf Codescape.GNU.Tools.Package.2018.04-02.for.nanoMIPS.Bare.Metal.CentOS-5.x86.tar.gz
```

This will produce a folder called `nanomips-elf/2018.04-02`; the executables can then be found within the `bin` folder.

3. Building a Simple Program

The industry standard for building and executing a program in languages like C involves several stages. For a program that may span several files:

1. The **compiler** (cc1) translates each file of source code into a file of *assembly code* (.s). This also includes running the **pre-processor**; a part of the compiler that evaluates macros in the source code.
2. The **assembler** (nanomips-elf-as) then translates the assembly file into a file of *object code* (.o).
3. The **linker** (nanomips-elf-ld) finally takes the object files generated from all the source program files and combines them into a single *executable* file, or *ELF* file (.elf or no extension). This can now be run on the hardware, or in a tool like QEMU (see later).

The nanoMIPS toolchain follows this process. The individual tools involved are normally invoked by a special high-level program called the **compiler driver** (nanomips-elf-gcc) which can be used to do all of these tasks in sequence. The driver will generally only leave behind the final executable, removing intermediate assembly and object files unless told otherwise.

3.1. Compiling a C file

Below are several examples of compiling the program found in Appendix A:

- `nanomips-elf-gcc -c hello.c -o hello.o`

  Compile and assemble hello.c to an object file 'hello.o' without invoking the linker afterwards (-c).

- `nanomips-elf-gcc -c hello.c -o hello.o -save-temps`

  Compile hello.c to an object 'hello.o' without removing any intermediate pre-processed or assembly files (-save-temps). They will be called 'hello.i' (pre-processed code) and 'hello.s'.

- `nanomips-elf-gcc -c hello.c -o hello.o -g -O2 -march=i7200`

  Compile hello.c to an object 'hello.o' with debugging information in the executable (-g) and moderate code size/performance optimisations enabled (-O2) such that it can run on an i7200 architecture (-march=i7200).

The compiler driver, by default, produces little-endian soft-float nanoMIPS32R6 code. It also does not optimise the generated code, potentially translating source code into an unnecessarily large and slow executable version. Commands such as the above can tweak this behaviour.

The following command-line options can alter the behaviour of compilation:

Option	Effect
-c	Compile the source file to an object, but not link.
-E	Preprocess the source file only.
-S	Compile the source file to assembly.
-o *outFile*	Specify the name of the output executable, or other output file if using the –c, -E or -S options above.
-march=<arch>	Specify an architecture to compile for.
-save-temps	Leave behind all intermediate files (pre-processed, assembly, object code).
-O0	Disable optimisations.

Option	Effect
-O1	Enable basic optimisations to eliminate dead or redundant code.
-O2	Optimise to balance performance and code size.
-Os	Optimise for code size at the cost of performance.
-O3	Optimise for performance at the cost of code size.
-g	When applied in combination with any of the optimisation levels above, produce debug information to help interrogate the program state at runtime. When applied to higher optimisation levels the accuracy of debug information will degrade.
-Og -g	Enable enough optimisations to produce readable assembly code with acceptable performance without degrading the debug experience.

3.2. Linking objects

The linker also requires information about the target IP to know what to link in and in what format. Most of this is handled automatically, but the structure of the final executable can be given by a **link script**. Here, we produce an executable from an object file with the Unified Hosting Interface (UHI) semi-hosting library targeting the I7200 IP core:

```
nanomips-elf-gcc -march=i7200 -Tuhi32.ld hello.o -o hello.elf
```

(The 'uhi32.ld' file is part of the toolchain. However, if you were to define a link script in your working directory at link time, it could be used in the same way.)

More details about linker scripts is available in the GNU Tools Programmer's Guide and the linker documentation found in share/docs/ld.pdf in the toolchain installation.

Command-line options that may influence the linker via the compiler driver include:

Option	Effect
-T<linkScript>	Specify which link script to use.

4. Running a Program in QEMU

After having built a program, the next step is to execute it.

The nanoMIPS toolchain includes a version of QEMU usable for simulating execution of bare metal code and has support for the UHI form of semi-hosting. This allows a target program to interact directly with the host for purposes of printing messages to the terminal, exiting when execution completes and even access to files from the host file-system.

The basic Hello World example, after having been built with the UHI link script, can be run as follows:

```
qemu-system-nanomips -semihosting -nographic -kernel hello.elf
```

This will produce output on the terminal from the target program as:

```
Hello world!
300
```

The following command-line options for QEMU are used above:

Option	Effect
-semihosting	Use UHI semi-hosting.
-kernel *bzImage*	Use *bzImage* as the kernel image, ie. the program to run.
-nographic	Don't create a graphical window the program can draw to.

5. Debugging using GDB and QEMU

Now that the test program can be run, debugging facilities are the next essential. The GDB tool in the GNU nanoMIPS Tools provides this facility, and is able to work in conjunction with QEMU. Below is a demonstration of how GDB and QEMU may be used to debug the 'hello.c' test program.

1. Build the test program as follows:

   ```
   nanomips-elf-gcc –g hello.c -o hello.elf -Tuhi32.ld
   ```

   (The UHI link script is required to use semihosting with QEMU.)

2. Run the executable with QEMU:

   ```
   qemu-system-nanomips -semihosting -nographic -gdb tcp::1111 -S –
   kernel hello.elf
   ```

   This runs the 'hello.elf' executable with semihosting (-semihosting) and no graphics (-nographic), waiting for a GDB service to connect to it on TCP port 1111 (-gdb tcp:1111). The system will be frozen at startup until ‘c’ is pressed or a connected GDB service starts a debugging session (-S).

3. In a new window, run GDB and connect it to the QEMU process still running:

   ```
   nanomips-elf-gdb --eval-command=”target remote:1111” hello.elf
   ```

   This starts up GDB, where it will first evaluate a command that connects it to a remote QEMU service on port 1111 (--eval-command=”target remote:1111”). The executed file must also be specified so GDB can see the debug information inside it (hello.elf).

The GDB and QEMU services are now connected and the GDB process can be used to debug the program running on the QEMU service. (Note that these services need not be on the same computer, as the connection between them is TCP.)

Normal GDB debugging features are now usable. A sample session of using GDB as above is show below:

```
(gdb) break main
Breakpoint 1 at 0x80200528: file hello.c, line 13.
(gdb) continue
Continuing.
Breakpoint 1, main () at hello.c:13
13          printf ("Hello world!\n");
(gdb) next
14          printf ("%d\n", f(25));
(gdb) step
f (a=25) at hello.c:5
5           int i, j = 0;
(gdb) next
6           for (i = 1 ; i < a ; i++)
(gdb) step
7               j += i;
(gdb) next
6           for (i = 1 ; i < a ; i++)
(gdb) finish
Run till exit from #0  f (a=25) at hello.c:6
0x80200534 in main () at hello.c:14
14          printf ("%d\n", f (25));
Value returned is $1 = 300
(gdb) next
15              return 0;
(gdb) continue
Continuing.
Remote connection closed
Qemu finished and emulated program.
```

Appendix A. hello.c

```
#include <stdio.h>
int f (int a)
{
  int i, j = 0;
  for (i = 1 ; i < a ; i++)
    j += i;
  return j;
}
int main()
{
  printf ("Hello world!\n");
  printf ("%d\n", f (25));
  return 0;
}
```