μMPS2 Cross Toolchain Guide

Tomislav Jonjic

September 27, 2011

Abstract

Development for μMPS2 is typically done in the C programming language, using a toolchain running on a separate system (most often on the system that hosts the μMPS2 virtual machine itself). Such a toolchain is often referred to as a cross toolchain. The present guide describes best practices of building and using a cross GNU toolchain[1] to prepare programs for μMPS2.

Contents

1 Introduction

The GNU toolchain is the de facto standard and by far the most supported toolchain for embedded MIPS systems today, and thus a natural choice for μMPS2. The GNU toolchain comprises, most importantly, the GNU Compiler Collection (GCC)

[1]Although this guide covers only the GNU toolchain, it should be mentioned that viable alternatives to it might exist in the near future. One promising alternative is the LLVM project which, as of release 3.0, contains an experimental MIPS backend. See http://developer.mips.com/tools/compilers/ for a comprehensive list of MIPS compilers.

and the GNU Binutils suite of “binary tools” (assembler, linker, and several other utilities).

As of this writing, there are essentially two variations of the GNU toolchain for MIPS targets in widespread use:

- toolchains for bare-metal MIPS targets, which are suited for building standalone programs that do not depend on an existing operating system or other run-time support;
- toolchains configured for Linux-based targets, suitable for building user space programs for MIPS systems running GNU/Linux.

Despite the various differences between these two types of toolchains, both can be used to build programs for μMPS2 with some precaution. This guide will only describe how to build a bare-metal toolchain, since those are inherently better suited for our purposes then the other type. Usage description covers both types, however.

Document Structure

Section 2 describes how to build a cross GNU toolchain suitable for μMPS2. The remaining sections cover the most important MIPS-specific aspects of the toolchain and parts of μMPS2 that relate in some way to the toolchain. Sections that cover advanced material are marked with an asterisk ($*$) and can be safely skipped on first reading.

The reader who is eager to get quickly up and running can, after having built or otherwise obtained a functional cross toolchain, proceed to the example given in Appendix A which shows how to build a μMPS .core executable.

2 Building a Cross Toolchain

This section presents instructions for building a bare-metal MIPS cross toolchain. These instructions have been tested on GNU/Linux systems; it is hoped, however, that they will also be useful for other Unix-like environments. Similarly, release 2.21 of Binutils[2] and 4.6.0 of GCC[3] have been tested most extensively by the author with the procedure described below, but it is expected that any other recent releases will work as well.

The instructions are presented as a series of shell commands which the user can opt to collect into a shell script for convenience. First, we need to define some environment variables which will be referred to later on:

```
$ export XT_PREFIX=$HOME/umpsxt
$ export XT_TARGET=mipsel-elf
$ export PATH=$XT_PREFIX/bin:$PATH
```

The `XT_TARGET` variable holds the name of target we are configuring for. If your target machine is big endian, replace `mipsel-elf` with `mips-elf`.

The `XT_PREFIX` variable specifies the top-level installation directory the toolchain components are going to be installed under. While this would typically be `/usr/local` for most GNU packages installed from source, we chose nevertheless

[2]http://www.gnu.org/software/binutils/
[3]http://gcc.gnu.org/

to install everything under a separate subtree (in this case, of the user's home directory). This structure allows easy management of multiple versions of the toolchain: simply install each toolchain under a separate prefix and remove that prefix when no longer needed. This is especially true of GCC, which does not support an "uninstall" Make target at this time.

We follow the practice of using a separate build tree from the source tree.

2.1 Building Binutils

Configure, build and install Binutils as follows:

```
$ mkdir binutils-bld
$ cd binutils-bld
$ ../binutils-2.21/configure --prefix=$XT_PREFIX \
                             --target=$XT_TARGET \
                             --disable-nls
$ make
$ make install
```

The `--prefix` option specifies the top-level installation directory; this defaults to `/usr/local`. Since we are building a cross toolchain, the `--target` configuration option must be specified. The `--disable-nls` option disables native language support.

2.2 Building GCC

We now proceed to build the cross compiler.[4] We will only build the compiler for the C language, out of the many supported by GCC. Therefore, you only need to download the `gcc-core-`VERSION`.tar.*` distribution.

2.2.1 Prerequisites

Besides the usual prerequisites (e.g. an ISO C90 compiler, a POSIX-compliant shell, GNU Make) GCC requires the following libraries as of release 4.6.0:

- GNU Multiple-Precision Library (GMP)
  `http://gmplib.org/`
- GNU MPFR Library
  `http://www.mpfr.org/`
- MPC Library
  `http://www.multiprecision.org/`

Use your package management system to install the development packages for the above libraries, if possible.

[4] To be precise, we are building what is often called in jargon a "stage 1" cross compiler; that is, a compiler for the target platform which can be used to build a C library for the target system and subsequently a full toolchain with run-time components. In terms defined by the C standard, our stage 1 compiler is suitable for freestanding environments (such as operating system kernels), as opposed to hosted environments. See `http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf` for more details.

2.2.2 Configuration and Building

Configure, build, and install the cross compiler as follows:

```
$ mkdir gcc-bld
$ cd gcc-bld
$ ../gcc-4.6.1/configure --prefix=$XT_PREFIX \
                         --target=mipsel-elf \
                         --enable-languages=c \
                         --without-headers \
                         --disable-nls
$ make all-gcc
$ make install-gcc

$ # Build libgcc (optional)
$ make all-target-libgcc
$ make install-target-libgcc
```

As for Binutils, we specify the target when configuring. `--enable-languages=c` selects the subset of GCC frontends we want to build.

The `--without-headers` option prevents the build of relying on headers from a target library, which we lack. This option currently only affects GCC supporting libraries (e.g. `libgcc`).

The `all-gcc` and `install-gcc` Make targets cause only the cross compiler to be built and installed, but not any support library (e.g. libgcc and any other support library that has not been disabled).

The last two commands (optional) cause the libgcc run-time library to be built. This library mostly consists of implementations of integer and floating point arithmetic operations that are not directly supported by the underlying architecture. A typical example is support for the ISO C99 `long long` type; since the MIPS I ISA does not provide 64-bit integer arithemetic instructions, GCC will generate calls to software implementations of those.

3 GNU Toolchain MIPS Specifics

This section describes MIPS-specific aspects of GCC and certain characteristics of the MIPS ABI(s) the programmer needs to be aware of.

3.1 MIPS-Specific Compiler Options

We describe here some common code generation options that are specific to MIPS targets. Although the GCC manual[5] should be considered the authoritative reference for these, the plethora of options can be daunting at first for programmers new to the MIPS platform. For this reason, we present a summary of the most important code generation options from the perspective of μMPS2 users:

-EL

Generate little-endian code (default for little-endian configurations, i.e. `mipsel-*-*`).

[5] http://gcc.gnu.org/onlinedocs/

`-EB`
: Generate big-endian code (default for big-endian configurations, i.e. `mips-*-*`).

`-mips1`
: Generate code for the MIPS I ISA. Depending on the configuration, this may or not be the default. μMPS2 comprises an extended R2000/R3000 MIPS CPU, which implements the MIPS I ISA.

`-mabi=abi`
: Specify the ABI dialect. For MIPS I targets, abi can be either `32` or `eabi`. `-mabi=32`, the default for MIPS I targets, is recommended for μMPS2 programs.

`-mabicalls`, `-mno-abicalls`
: Emit (do not emit) position independent code (PIC), as required by shared libraries or position independent executables (PIEs). See section 3.3 for more details.

  `-mabicalls` is the default on SVR4-based target configurations (including GNU/Linux). In such cases, `-mno-abicalls` along with `-fno-pic` must be specified to prevent generation of PIC code.

`-mgpopt`, `-mno-gpopt`
: Enable (do not enable) the GP-relative addressing scheme optimization. See Section 3.2 for details.

`-G` size
: Set the maximum size (in bytes) of data objects that can be placed in the "small data" sections (e.g. `.sdata` and `.sbss`), provided that GP-relative addressing is enabled. A zero value will effectively disable GP-relative addressing altogether, independently of any other option. The default value for `-G` depends on the configuration, but it is in practice either eight or zero bytes.

3.2 GP-Relative Addressing $*$

One of the consequences of the MIPS instruction set architecture's fixed instruction length (32 bits) for code is that data object accesses are relatively inconvenient, since a 32-bit address cannot be encoded in a single instruction. Consider, for example, how a word-sized global variable (`foo`) is read:

```
        lui     $t0, %hi(foo)
        lw      $t1, %lo(foo)($t0)
```

The first instruction loads the most significant 16 bits of the address into bits 16-31 of register `t0`, while the remaining bits are specified in the "immediate" part of the `lw` instruction. It is obvious that this can cause a substantial increase of code size and performance deterioration in general if the program exhibits a lot of access patterns similar to the one above.

A solution adopted by MIPS compilers consists of placing "small" data objects (e.g. word-sized global variables) into special data sections that are allocated a contiguos region of the resulting data segment. Furthermore, a register (by convention `gp`) is reserved to point to (the middle of) this region. Thus, as long as the size of the resulting memory region reserved for small data objects is less than 64 KB, these can be accessed with a single instruction, as illustrated with the following fragment:

```
        lw      $t1, %gp_rel(foo)($gp)
```

The above scheme requires coordination between the compiler and linker, in addition to some effort from the programmer:

- First, the compiler must be instructed to enable GP-relative addressing via the `-mgpopt` nad `-G` options, which respectively enable the GP-relative addressing optimization and set the maximum size of data objects for which GP-relative access is going to be used. The specified size must also be supplied to the GNU linker (ld) through the `-G` option.
- The linker must be instructed (using a linker script) to define a symbol that points to the middle of the above-described small data region; by convention, that symbol's name is `_gp`.

  The linker scripts distributed with μMPS2 already provide for this. Additionally, the above pointer (that is, the initial value of the `gp` register) is also contained in the .aout file header (see [1]).
- The `gp` register must be initialized to point to the middle of the small data region by the startup code. In standard startup code provided by μMPS2 (see `crt*.S`), this is done in the `__start` routine, before control is transferred to `main()`.

3.3 MIPS Position Independent Code $*$

A typical application on systems such as GNU/Linux is composed of many independently linked units. These independently linked units (i.e. shared libraries or, in ELF terminology, dynamic shared objects) are combined together at program load time by the dynamic linker to create the process image and can even be loaded at run time to augment the process image.

Since different programs generally result in different address spaces, shared libraries cannot have fixed load (virtual) addresses; instead, the dynamic linker loads each object in such a way that it avoids conflicts with existing segments that comprise the process image. Furthermore, physical pages of memory that host the code and read-only data segments belonging to the shared library are shared between processes, making it is impossible to perform any address space dependent relocations on the code itself. Together, this imposes a very strict requirement on shared library code: it must be position independent — that is, it must be independent of the virtual address the library is mapped at.

The essential characteristic of position independent code[6] (PIC) is the absence of hardcoded absolute addresses in instructions. Instead of relying on absolute addresses, the compiler generates instruction sequences that are used to compute the required address at run time.

A crucial aid in address calculation is a data structure called the global offset table (GOT). Each unit comprising the process image has as separate GOT; each GOT

[6]The reader should not confuse this generally adopted meaning of the term "position independent code" with relocatable code — code suitable for linking with other object code in order to create an executable or shared library.

entry contains a pointer to a global data object or procedure entry point referenced by that unit. Since the GOT is part of the data segment, it can be safely modified by the dynamic linker to reflect the correct function or data symbol addresses (i.e. relative to the program's address space).

The calling convention mandates that, when using position independent code, the callee is given its address in the `t9` register. This makes it straightforward to compute the location of the GOT, since the offset of the callee's entry point from the GOT is known at link time. This computation is typically done in the function's prologue, and the GOT address is cached in register `gp`.

As a way of example, contrast the function-call code sequences for non-PIC and PIC code (details of the calling convention that do not differ between the two modes — notably argument structure setup — are not shown). The non-PIC version is a lone instruction:

```
jal     <address>
```

The position independent equivalent of the above is:

```
lw      $t9, <symbol entry>($gp)
nop
jalr    $t9
```

In the above sequence, the address of the called function is first retrieved from the corresponding GOT entry into register `t9`, which is then used as the target in the jump to register instruction. (The instruction in-between is the load delay slot.) An analogous scheme is used for references to data objects.

It is clear that, due to indirect addressing, position independent code incurs a non-negligible overhead. On systems that support shared libraries, this overhead is justified by their benefits, of course. For programs that cannot (or do not) gain advantage from position independent code — the most obvious example being operating system kernels — this overhead would be gratuitous. Perhaps even more importantly from the point of view of μMPS2 users, position independent code is both less readable (to the extent machine code can be) and more difficult to interpret by code inspection tools (such as the built-in debugger in the μMPS2 emulator).

As can be inferred from the above, shared libraries — as found on modern ELF-based systems — require considerable support from the operating system; for this reason alone, they will almost certainly be out of reach for typical μMPS projects. Furthermore, we note that the μMPS2 .aout format, like the original variants a.out format, does not adequately support shared libraries.[7] The a.out format, while very simple to understand, is inherently inflexible and non-extensible, as it does not support arbitrary code, data, or metadata sections. It is thus outright difficult, if not impossible, to include the necessary shared library-related metadata in a structured manner.

For further information on the topics discussed above, interested reasers can consult a variety of resources. The MIPS specifics of the System V ABI, on which the Linux MIPS ABI is based, are defined in [2]. A much more readable introduction is given in [3]. [4] is a comprehensive reference on the topic of object formats, linking, and loading.

[7]The inability of a.out and its early successors (such as COFF) to efficiently support dynamic linking was one of the primary motivations behind the transition to the ELF format on GNU/Linux and BSD-based systems. These early object formats also proved to be inadequate for environments of more complex languages, such as C++.

4 Linker Scripts

A linker script describes how sections from one or more input object files should be merged to create an output object file. Among other things, a linker script defines the memory layout of the resulting object file, which in our case will always be an ELF executable file. The linker script command language accepted by GNU ld is described in its manual.[8]

The GNU toolchain does not directly support the μMPS2 .aout and .core object formats, of course. Instead, ELF executables must be converted to one of the μMPS2 object formats using the `umps2-elf2umps` utility. Multiple linker scripts are supplied with μMPS2, each one being suitable for a specific μMPS2 target object format. When an .aout/.core linker script is used, the desired memory layout of the .aout/.core file is reflected in the linked ELF file, and thus the latter can be converted to the former in a relatively straightforward manner by `umps2-elf2umps`.

The following linker scripts are supplied with μMPS2:

- `umpsaout.ldscript`
  As indicated by its name, this script should be used for ELF executables aimed to be converted into the .aout format.

- `elf32btsmip.h.umpsaout.x`
  `elf32ltsmip.h.umpsaout.x`
  These scripts, for big-endian and little-endian targets respectively, are provided for backward compatibility with previous releases of μMPS. They are suitable for .aout object files. Note, however, that they are considered deprecated and may be removed in a future release.

- `umpscore.ldscript`
  This script should be used for ELF executables aimed to be converted into the .core variant of the .aout format.

- `elf32btsmip.h.umpscore.x`
  `elf32ltsmip.h.umpscore.x`
  These scripts, for big-endian and little-endian targets respectively, are provided for backward compatibility with previous releases of μMPS. They are suitable for .core object files. Note, however, that they are considered deprecated and may be removed in a future release.

All of the above scripts are installed in <prefix>`/share/umps2`, where <prefix> denotes the top-level installation directory under which μMPS2 was installed. Users whose requirements are not met by the standard linker scripts are advised to modify one of `umpsaout.ldscript` or `umpscore.ldscript`, according to suitability. The `umpsaout.ldscript` and `umpscore.ldscript` scripts are suitable for both little-endian and big-endian targets.

A linker script is specified to the linker using the `-T` option.

All scripts supplied with μMPS2 specify the address of the `__start` symbol as the program entry point. See Section 5 for information about program startup.

[8] `http://sourceware.org/binutils/docs-2.21/ld/index.html`

5 Program Startup

Before control can be passed to compiled C code (e.g. the `main()` entry point), it is necessary to properly initialize its environment; this is the task of the "runtime support" modules provided with μMPS2. On MIPS systems, this usually includes initialization of the `gp` register if GP-relative addressing is used or if the code is position independent.

The following startup modules, installed in <prefix>`/share/umps2`, are provided with μMPS2:

- `crtso.S`: This module contains startup code suitable for .core object files (i.e. the OS kernel).
- `crti.S`: This module contains startup code suitable for .aout object files (i.e. user mode programs).

For an example on how to integrate this modules in your programs, see Appendix A.

For backward compatibility with previous releases of μMPS, the startup modules are also provided in pre-assembled form, as position independent relocatable ELF files (crtso.o and crti.o) that use the GNU/Linux ABI[9]. These are installed in <prefix>`/lib/umps2`. Note, however, that they are deprecated and are not recommended for use by new programs.

6 The libumps Library

The libumps library provides ROM service stubs, convenient access to processor control registers, and wrapper function for some μMPS2-specific processor instructions (i.e. extensions of the MIPS I instruction set). See [1] for a complete description of this facility.

Like the startup modules, the library is provided in both source (`libumps.S`) and binary (`libumps.o`) forms, but please note that the former is the preferred way to include `libumps` in your projects.

A An Example Makefile

In this section we present a simple but complete Makefile for a μMPS2 kernel project, which the reader can use as a starting point for more elaborate Makefiles. Note that part of the syntax is specific to GNU Make.

For the sake of this example, assume the project consists of two C modules, `main.c` and `util.c`, and that we want to link the libumps library and the standard startup module with our kernel.

```
# Cross toolchain variables
XT_PRG_PREFIX = mipsel-elf-
CC = $(XT_PRG_PREFIX)gcc
LD = $(XT_PRG_PREFIX)ld

# uMPS2-related paths
```

[9]The precise names of the target format used by the GNU/Linux MIPS ABI are elf32-tradbigmips and elf32-tradlittlemips for big- and little-endian targets, respectively.

```
 7 UMPS2_DIR_PREFIX = /usr/local
 8 UMPS2_DATA_DIR = $(UMPS2_DIR_PREFIX)/share/umps2
 9 UMPS2_INCLUDE_DIR = $(UMPS2_DIR_PREFIX)/include/umps2
10
11 # Compiler options
12 CFLAGS_LANG = -ffreestanding -ansi
13 CFLAGS_MIPS = -mips1 -mabi=32 -mno-gpopt -G 0 \
14               -mno-abicalls -fno-pic
15 CFLAGS = $(CFLAGS_LANG) $(CFLAGS_MIPS) \
16          -I$(UMPS2_INCLUDE_DIR) -Wall -O0
17
18 # Linker options
19 LDFLAGS = -G 0 -nostdlib \
20           -T $(UMPS2_DATA_DIR)/umpscore.ldscript
21
22 # Add the location of crt*.S to the search path
23 VPATH = $(UMPS2_DATA_DIR)
24
25 .PHONY : all clean
26
27 all : kernel.core.umps
28
29 kernel.core.umps : kernel
30         umps2-elf2umps -k $<
31
32 kernel : main.o util.o crtso.o libumps.o
33         $(LD) -o $@ $^ $(LDFLAGS)
34
35 clean :
36         -rm -f *.o kernel kernel.*.umps
37
38 # Pattern rule for assembly modules
39 %.o : %.S
40         $(CC) $(CFLAGS) -c -o $@ $<
```

On lines 1-9 we set toolchain and μMPS2-releated parameters. The CC and LD predefined Make variables, used in implicit rules, are modified to match the corresponding program names of our MIPS toolchain.

On lines 11-16 we define compiler options. With the MIPS-specific options we specify, among other things, that position include code should not be generated and that GP-relative addressing should not be used. Apart from those, the only other option that may not be immediately obvious is `-ffreestanding`, which specifies that the program we are building is targeted at a freestanding execution environment. (A freestanding execution environment, as opposed to a hosted environment, is one in which only a small subset of the standard library headers need to be provided[10], and in which program startup and termination is implementation-defined.)

Linker option are specified in the LDFLAGS variable. Since the end goal is a kernel image in the .core format, the `umpscore.ldscript` linker script is used, as specified via the `-T` option. The `-G` option is the linker counterpart to the compiler's `-G` option (see Section 3.1). The `-nostdlib` linker option prevents the linker

[10]The `<stdarg.h>` header is, notably, one of the standard headers guaranteed to be present even in a freestanding compilation environment by any conforming implementation of the standard.

from using any default library search path.

The assignment to the VPATH variable on line 23 has the effect of adding the location of the standard startup module source files to the prerequisite search path list.

A custom pattern rule is included (lines 39-40) for assembly modules (in this example, `crtso.S` and `libumps.S`). The assembler is invoked via the `gcc` driver program for convenience, since assembly source files need to be preprocessed.

The remainder of the Makefile consists of straightforward rule definitions.

References

[1] Michael Goldweber and Renzo Davoli. μMPS2 Principles of Operation, 2011.

[2] System V Application Binary Interface: MIPS Processor Supplement, 3rd Edition.

[3] Dominic Sweetman. See MIPS Run, 2nd edition. Morgan Kaufmann Publishers Inc., 2006.

[4] John R. Levine. Linkers and Loaders. Morgan Kaufmann Publishers Inc., 1999.