LM32-toolchain

Rebuilding a compiler in 2019
April 2023 (1ce6142)

Alessandro Rubini, for GSI

Table of Contents

1 Abstract (TL;DR)

Currently, the White Rabbit PTP Core (*wrpc*) includes an LM32 processor, and we are building stuff with *gcc-4.5.3*, mostly using a build of mine that dates back to November 2011, o a rebuild of mine dated 2019 (i.e., the previous version of this package).

This work is about refreshing the tool chain, for both White Rabbit and other LM32-based project.

We must be able to:

- Rebuild that compiler with current or almost-current hosts;
- Run newer compilers to see if new features can benefit our projects;
- Try alternative tool chains, such as LLVM.

The current status of this work is:

- We have a working 64-bit build of gcc-4.5.3, with original libraries
- We have gcc-4.5.3 with newer support libraries;
- We can build gcc-4.5.3 with up to gcc-11 as host compiler
- Newer lm32 compilers work, but create bigger binaries and are not suitable for our projects, at least the big ones
- No work on LLVM is yet completed.

After refreshing this work in 2023, I confirm that the best choice for our lm32 projects is using *gcc-4.5.3* with the original set of support libraries. In any case, all versions I built are uploaded to `lx-pool`.

2 Initial Remarks

This work was initially performed in 2019, on request (and sponsoring) by GSI, and was later refreshed in 2023.

2.1 Other Works

Before starting this, I checked what other people published. Unfortunately I found very little, and I welcome different pointers. This is the state of the art, to my knowledge, as of 2019-01. I didn't check further sources in 2023, as the previous scripts were still working, and the customer claimed to not be interested in post-gcc-8 releases.

`https://github.com/shenki/lm32-build-scripts`
: A script by Joel Stanley. The repo is just two commits: his build script for gcc-6.2 and a later update to gcc-6.3. This confirmed to me that LM32 is not abandonware within *gcc*, but the specific versions are not the best choice (e.g., a daily snapshot of *newlib*, that I can't find any more.

`https://github.com/optixx`
: The package includes the whole LM32 Verilog sources and support tools, but the toolchain suggested is a snapshot of gcc-4.5, which is older than our current setup.

`https://github.com/m-labs/llvm-lm32`
: This may be my starting points for LLVM support, but *gcc* support was considered ok for this work, so I didn't work on LLVM.

2.2 gcc Version Numbering

Up to gcc-4.x, the second number in the version number was like a major release, so gcc-4.6 was a different thing than gcc-4.5, while the third number was just a maintainance/bugfix release. Starting with version 5, the first number is the major release. So gcc-5.3 is a bugfix release of the gcc-5 cycle.

For this reason, I tested the last one of most gcc-4.x series (4.5.3 revealed good over the years so I skipped 4.5.4), but only the last one of versions 5, 6, 7, 8.

2.3 This Work

Note: this section is meta-information about this package; please goto Chapter 3 [The Build Script], page 2, if you want to start compiling.

This document is quite verbose, because I hope the information will be useful in the future, when somebody will pick it up to go further – as already happened from 2019 to 2023. As a side effect, I use it as my own reference while working on this.

The package includes a shell script, configuration files, patches and a document. Binaries are not included in this repository, but I placed important tarballs in the GSI network. In a package about rebuilding, binaries are out of scope.

To build the output formats of the document, just *make* in the `doc/` subdirectory of the *git* clone. You may need to install the *texinfo* package.

The document is written in TeXinfo, the GNU project documentation standard. It may be old-fashioned, but it revealed a future-proof choice, when I made it. However, I love being able to place white space at the beginning of the lines, to make sense of the file in my editor, and avoid markup in @example section so I can copy them from the shell terminal into the editor (or from it to the shell). For this reason, I preprocess the real input file. The source file is *lm32-toolchain.in* and not the `.texinfo` one; if you edit the latter, it will be overwritten at the next built (that's why I make it read-only, as a warning).

If you are the new maintainer of this package and you want to move to a different source format, please consider using the source file *lm32-toolchain.in* as your starting point, as an alternative the text output file might be better, or the *html* output if you can import the italics and tty-face markup.

2.4 Testing

The rebuilt toolchains were tested using the current *master* commit of *wrpc-sw* (commit `566f213f`, tagged as `wrpc-v4.2-0`, with *gsi_defconfig* as configuration. I compared with the output of *gcc-4.5.3* that was built in the previous iteration of this package (Febuary 20219).

Unfortunately, `wrpc-v4.2-0` had a few build isses for me (and, I suspect, for other users). So fixed them, and put the patches in this package, in `patches/wrpc`.

Finally, please note that in order to build *wrpc-sw* on a freshly-installed Linux distribuition, you need to install *git* and *libreadline-dev* (which is called *readline-devel* in *rpm*-based distributions).

3 The Build Script

Obviously, everybody and their friend wrote a build script. Some are simple sequences of commands, like the ones in Section 2.1 [Other Works], page 1, some are very complex tools, like buildroot or (got forbid!) yocto. I prefer something in the middle, with some factorization but not much, to keep things simple.

Fact is, my script of 2010 still works fine, so I recycle it here. It is a shell script (`tools/build-generic` that relies on a configuration file, which is a dozen lines long. Mainly, you state which package versions to download and what build options to apply. For example:

```
PREFIX="$(/bin/pwd)/install/lm32-gcc-$(date +%y%m%d-%H%M)"
TARGET="lm32-elf"
GCC_CONFIG="--disable-libssp"

# table of programs,versions and so on
prog gcc      8.2.0    xz   https://ftp.gnu.org/gnu/gcc/gcc-8.2.0
prog binutils 2.31.1   xz   http://ftp.gnu.org/gnu/binutils
prog newlib   3.0.0    gz    ftp://sources.redhat.com/pub/newlib
[...]
# package list: what to get
plist="gcc binutils gdb newlib mpc mpfr gmp"
```

The script is then called with the configuration file as an argument, and it creates a complete build log which is timestamped by the minute. The configuration above also installs in a place that is timestamped by the minute, but you may prefer a difference choice (as I did when creating the binaries that I eventually distributed).

To prevent any error and to be able to recover what your did, the script saves itself and the configuration file to the log file, before the build starts/.

Following a request from GSI, the installation directory includes the day of the build and two hashes: the git commit of this repository and the configuration file. That's because the basename of the directory (as appearing in the output of *gcc -v*) can be saved during FPGA builds, so to keep track of which toolchain was used to build each binary image.

Please note that the configurations in this package define `PREFIX` by themselves. The build script offers a default (within `/opt` but configurations override it.

The build directory (and log file) currently use a timestamp-based name, because I prefer to keep all build logs, with errors; I don't want any to be overwritten while I work on several builds at the same time.

For example, this is what I'm getting now, while editing this:

```
laptopo% ./tools/build-generic ./configs/gcc-4.5.3-orig
Using ./configs/gcc-4.5.3-orig as config file
Using PREFIX=//lap-x/wip/lm32-toolchain/install/lm32-gcc-230329-92a789-7be6
Config file is ./configs/gcc-4.5.3-orig
Building in "/lap-x/wip/lm32-toolchain/build-230329-17-24"
Installing in "//lap-x/wip/lm32-toolchain/install/lm32-gcc-230329-92a789-7be6"
Log file is "/lap-x/wip/lm32-toolchain/build-230329-17-24.log"
```

From the above it can be noted that

- I work in a different directory (external hard drive, for convenience);
- the date I run the build is `230329`, turned to the first item in the PREFIX value;
- the hash of the configuration is `92a789`: this is the variable `HASHCFG` calculated in the script;
- the hash of this repo is `7be6`: variable `HASHREPO`;
- the build directory and log file are named using a timestamp instead of the hashes.

3.1 Downloads

As a first step, the script is downloading source files. The download directory it `./downloads` where you invoke the script; you may pre-set this as a symbolic link. The script uses the version number, suffix and base URL as in the example above to generate the download URL. It downloads one *tar* file for each *package* listed in the `plist` variable (and for which it uses the corresponding `prog` line above it. If the file is already there, it is not downloaded, without a check of integrity. Files that are already in place can be symbolic links (for example, in one iteration I already had most of them due to *buildroot* runs over the last years, so I symlinked them all and avoided downloading).

Then, all source *tar* files of interest are expanded into `./src/` where you invoked the script. All relevant packages create a directory with the same base name, and this name is preserved. For example:

```
laptopo% tar tf downloads/mpc-1.0.3.tar.gz | head -1
mpc-1.0.3/
laptopo% ls -d src/mpc*
src/mpc-0.9/  src/mpc-1.0.3/
```

If the target directory within `./src` exists, it is preserved. This allowed me to test my patches easily. You can remove the source directories to restart a clean build, if you want.

3.2 Applying Patches

The build script applies patches, if any exist in the `patches` subdirectory of this package. It does so by creating a local *git* repository, committing the untarred files and then using `git-am` to patch. Creating the initial commit of the whole source tree may take quite some time, but fortunately it only happens once for each package, and only the ones that require patches.

This is an example from my first build of *gcc-4.5.3* within this package:

```
Uncompressing ../downloads/gcc-4.5.3.tar.bz2...
Patching gcc-4.5.3
Initialized empty Git repository in /u/arubini/lm32-toolchain/src/gcc-4.5.3/.git/
Applying: gcc/doc: fix use of @itemx
[...]
```

please note that from “Initialized” to “Applying” above, you may wait more than for the uncompression step (which of the two is longer depends on your disk speed, RAM size and current load).

When patches exist, the script creates a marker file to note that they are already applied. You can remove the marker file (e.g. `gcc-4.5.3-patched`) and the `.git` directory within the package to start clean again.

For each package that I had to patch, I provide the git-generated patch-set in the subdirectory `patches/`*`pkg-x.y`*. This set is what is automatically used by the script.

3.3 The Build Directory

Each run of the script creates a new build directory, called `build-$(date +%y%m%d-%H-%M)` (for example, `230331-12-00` if I built at noon today). The log file has the same name, with a trailing `.log`.

If an error happens and you re-run the build, everything will start again in a different directory. This allows me to ensure I didn’t forget something and what works for me will for you as well. If you need to debug a failed build, you can `cd` to the build directory and reproduce the error or try your fixes.

The size of each build directory is from 750MB up to 2.3GB, in the range of versions I document here as working. Don’t be shy about removing those when you are done with each of them.

3.4 Using the Log File

The log file includes the complete compilation log, so your can look for errors. At the beginning of the log you find the script and the configuration file. I did this because I tend to forget the details about each build, for example because I edit the configuration file for the next build trial, without changing the name.c

To recover the configuration that was used in a build, you can use `tools/recover-config`: cccccc

```
laptopo% ./tools/recover-config build-230331-10-00.log > prev-config
```

Similarly, `tools/recover-script` is there, but I never had to change the script for any build I describe in this document.

To look for errors, please grep for `Error` in the log file (note the upper-case 'E'). If none is there, the build was successful. I think I've been a little lazy with error checking in the script itself.

To find what your build time was (so to plan your coffee break when you run it again), check for lines starting with '`###`':

```
laptopo% grep '^###' ../../build-190115-19-07.log
### Tue Jan 15 19:07:56 CET 2019: config binutils: "../../src/binutils[...]"
[...]
### Tue Jan 15 19:27:48 CET 2019: install it all: "make install"
### Tue Jan 15 19:28:15 CET 2019: done: "true"
```

In each line above, the final string is the command being executed, that's why it is just `true` in the final 'done' message.

Finally, if you don't remember what build a specific log refers to, `tools/describe-log` extracts a subset of the configuration file, so you know the gcc, binutils and newlib versions, which are the most important information item:

```
laptopo% tools/describe-log build-190205-08-39.log
build-190205-08-39.log: gcc 4.7.4
build-190205-08-39.log: binutils 2.21.1
build-190205-08-39.log: newlib 3.0.0
```

The size of the log file goes from 7.5MB to 30MB, in the range of versions described here.

3.5 Example Run

This is an example run of the script, as executed on a debian-9 virtual machine. Pleae note that I suggest to run in the root directory of this package (possibly after filling `./downloads` with the files you already have), because the `.gitignore` file already supports it.

```
 debian-9$ ./tools/build-generic configs/gcc-4.5.3-orig
Using PREFIX=/home/rubini/lm32-toolchain/install/lm32-gcc-230320-92a789-1dd6
Config file is configs/gcc-4.5.3-orig
Building in "/home/rubini/lm32-toolchain/build-230320-11-38"
Installing in "/home/rubini/lm32-toolchain/install/lm32-gcc-230320-92a789-1dd6"
Log file is "/home/rubini/lm32-toolchain/build-230320-11-38.log"
 Dowloading https://ftp.gnu.org/gnu/gcc/gcc-4.5.3//gcc-4.5.3.tar.bz2
 [...]
 Uncompressing ../downloads/gcc-4.5.3.tar.bz2...
```

Then, patches are applied and the build proceeds, finally installing the compiler binary and support files.

3.6 Storage Requirements

This is a summary of the storage required for the builds described here (i.e. all versions):

- 600 MB for downloads
- 5.9 GB for uncompressed sources.
- 2.8 GB for installed trees (150MB to 650MB each version)
- 0.19 GB in log files (8MB to 40MB each)

- 12 GB for build directories (750MB to 2.3GB each)

The exact size will vary according to your filesystem layout and other details, so take the numbers above as a very rough estimation.

4 Host Systems

I built the compilers on a number of host systems, all of them x86-64 architecture.

```
Debian-9 (gcc-6.3.0, glibc-2.24)
Debian-10 (gcc-8.3.0, glibc-2.28)
Debian-11 (gcc-10.2.1, glibc-2.31)
Rocky-6 (gcc-11.3.1, glibc-2.34)
Ubuntu-22.04 (gcc-11.3.0, glibc-2.35)
```

Some builds failed, as detailed below, but we are still able to build all versions from 4.5.3 to 8 (I didn't try newer ones, as this set is enough for our needs).

In general, *glibc* ensures backward compatibility, so anything built against previous versions will run with later versions. This doesn't work backwards: binaries built agains newer versions will have some undefined sysmbols if dinamically linked against older versions.

Thus, I suggest to deploy the binaries built with debian-9, because they will run on every host in the list above.

4.1 Debian-9

On a fresh install of Debian-9, I had to add some packages in order to build the cross-compiler:

```
apt install git make gawk gcc g++ texinfo
```

The C++ compiler is only needed for versions newer than 4.5.3.

Configuration gcc-4.5.3-orig worked out of the box, but in order to build the other 4.5.3 configurations (See Section 5.2 [gcc-4.5.3 configs], page 7) I had to install package `libmpfr-dev`. The problem with *libmpfr* is described in Chapter 7 [libmpfr], page 9, but can be safely ignored if you use my suggested output binaries.

Newer versions of *gcc* built successfully, with my chosen configuration.

The binaries I uploaded to `lx-pool` are the ones built with Debian-9.

4.2 Debian-10

The behaviour is the same as in Section 4.1 [Debian-9], page 6: same packages to add, and same success/failure on configurations.

I admit I did not install `libmpfr-dev` nor build the related configurations.

4.3 Debian-11

The behaviour is the same as in Section 4.1 [Debian-9], page 6: same packages to add, and same success/failure on configurations. Again, I did not test builds with a newer *libmpfr*.

4.4 Rocky-6

I run the build on a GSI host, so I don't know what packages are missing in a fresh install, but I expect them to be the same as in Debian, or fewer if some are installed by default.

For version 4.5.3, the same result as in Debian applies: the “original” choice of support libraries builds, but newer ones fail, because I miss package *mpfr-devel*. Those build are not interestign anyways.

gcc-4.9.4 and *gcc-5.4.0* fail to build because the host compiler, *gcc-11* is exceedngly picky about some idioms used there. I spent no timein fixing it, because we can build elsewhere, and 4.5.3 which is our old-and-wise best choice builds properly.

Versions 6, 7, 8, build properly.

4.5 Ubuntu-22.04

The behaviour is the same as in Section 4.4 [Rocky-6], page 6, (as expected, because the host compiler is version 11 like above).

Starting from a fresh install, I had to add the usual package set:

```
apt install git make gawk gcc g++ texinfo
```

Build of "updated" gcc-4.5.3 configurations fail because I did not install *libmpfr-devel*, but we are not interested in those anyways.

5 gcc-4.5.3

As a first step, let's rebuild what we were using at the beginning of our LM32 projects. Newer host compilers complain about some code in *gcc-4.5.3* sources.

Most of the errors are related to documentation, where incorrect keywords were used, both in *gcc* and *binutils*. Some errors are because of "unused expression result" in some macro expansions, and one "value may be used uninitialized". The last error was a "mismatching prototype" because of a missing `const`. All of these are fixed in my patch-set.

5.1 Original Configuration

By running the build script with `configs/gcc-4.5.3-orig` we rebuild the same compiler we have been using originally.

The build is 750MB and the installation is 160MB. The binary built in Debian 9 relies on `glibc-2.24` and runs on all distributions I tested in this work.

When tested against *wrpc-sw*, commit `wrpc-v4.2` plus my patches, the output binary is smaller than what we got with the 2019 build:

```
   text    data     bss     dec     hex filename
 102088    8320    4652  115060   1c174 wrpc-sw-old-compiler/wrc.elf
 101444    6888    4652  112984   1b958 wrpc-sw-4.5.3/wrc.elf
```

Now, it looks like the deployed version in 2019 was the one using *newlib-3.0*, which increases the binary because of some extra localication (wide-char and such stuff) in *strcasecmp*.

Other differences depend on a better allocation of registers: a number of functions are a few instructions shorter because they save fewer registers to the stack. For example, *cmd_init* does not save R13 and makes a few indirect references using R11, where old code moves register. The resulting code is one instruction shorter. This may depend from better host libraries used in building the compiler, but I really have no sharp idea.

I tried a build on Rocky-6, with *gcc-4.5.3* built there. Despite the still-newer host libraries I found no differences, and *wrc.elf* is identical to what is built with the debian-9-built compiler, which I thus bless for deployment.

5.2 Extra configurations for gcc-4.5.3

Trying to "update" the old-and-wise compiler version, I tried three different choices of support packages. Here I document the outcome, but I don't suggest to used them in production (i.e. you can skip reading this).

gcc-4.5.3-updated

Configuration `configs/gcc-4.5.3-updated` creates the same base compiler but uses more recent support libraries: *mpc*, *mpfr*, *gmp*. Using a newer version of *binutils*, such as 2.28.1, is not feasible because the two *libiberty* in *gcc* and *binutils* differ in some symbol and cause a miscompilation (we could find a workaround, but it is not worth the effort).

For *newlib* I chose 2.0 because more recent versions would increase the binary size of our code, mainly because of support for local languages, as described later.

No new patches are needed with this tool-set. This compiler builds exactly the same code as the previous one – as expected, because support libraries are about multi-precision mathematical expressions, which we don't use.

Please note, however, that in order to build this you need to install *mpfr-devel* (Debian and derivatives) or *libmpfr-dev* (RH derivatives). Also, the problem the problem described in Chapter 7 [libmpfr], page 9, applies.

gcc-4.5.3 with updated newlib

In these configurations I upgraded the versions of newlib. The choices made in 2019 were not updated, because I suggest sticking with the *-orig* configuration. So, I rebuilt with version 2.5 (latest 2.x back then) and 3.0 (latest official 3.x at the time). Now we have 3.3 and 4.1, but I won't test them, because for this project we prefer the oldest choice.

Like in 2019, but the output size of *wrpc-sw* got bigger because of localization-related changes in case-insensitive character matching.

This is the size of the compiled `wrpc-sw` file in the various situations:

```
   text    data     bss     dec     hex filename
 101444    6888    4652  112984   1b958 ../wrpc-sw-4.5.3/wrc.elf
 101444    6888    4652  112984   1b958 ../wrpc-sw-4.5.3-updated/wrc.elf
 102080    8320    4652  115052   1c16c ../wrpc-sw-4.5.3-newlib-2.5/wrc.elf
 102088    8320    4652  115060   1c174 ../wrpc-sw-4.5.3-newlib-3.0/wrc.elf
```

By checking with "`nm --size-sort`" we can verify that there is no change for us between version 2.5 and 3.0, while the size difference from 2.0 to 2.5 comese from

- `impure_data`: 1064 bytes, not present in 2.0
- `__global_locale`: 364 bytes, not present in 2.0
- `_setlocale_r`: 144 bytes, not present in 2.0
- `strcasecmp`: 60 bytes longer
- other smaller functions related to wide characters

All of these come from *strcasecmp*, only used in command matching.

We can replacing all of them with *strcmp*, with the following command:

```
sed -i s/strcasecmp/strcmp/ shell/*.c
```

The resulting sizes are as follows:

```
   text    data     bss     dec     hex filename
 101052    6884    4652  112588   1b7cc wrc.elf-newlib-2.5
 101052    6884    4652  112588   1b7cc wrc.elf-newlib-3.0
 101052    6884    4652  112588   1b7cc wrc.elf-orig
 101052    6884    4652  112588   1b7cc wrc.elf-updated
```

This means that the output gets shorter than the original one, and by avoiding *strcasecmp* we can safely use newer support libraries.

Please note that the *wrpc-sw* documentation uses lowercase, all constant strings in command parsing are lower case and nobody, to my knowledge, ever used uppercase. So I consider the change a safe one.

I didn't try other versions of *newlib* in 2023, while upgrading this package, because we don't expect any benefit from it – we expect issues like above, if any.

6 Installing the Compiler

After building, possibly using my convoluted installation names it is possible to make a *tar* file of the directory and uncompress it in a different pathname. When `lm32-elf-gcc` is called, it will find all its support files using relative pathnames.

I tarred all compilers I describe here, built on Debian-9, and placed them on `lx-pool.gsi.de/arubini/gcc-lm32-2023/`:

```
lxi098$ ls -lh gcc-lm32-2023
total 4.2G
-rw-rw-r-- 1 arubini bel 221M Apr  1 16:45 lm32-gcc-4.5.3-newlib-2.5.xz
-rw-rw-r-- 1 arubini bel 225M Apr  1 16:47 lm32-gcc-4.5.3-newlib-3.0.xz
-rw-rw-r-- 1 arubini bel 214M Apr  1 16:49 lm32-gcc-4.5.3-updated.xz
-rw-rw-r-- 1 arubini bel 214M Apr  1 16:50 lm32-gcc-4.5.3.xz
-rw-rw-r-- 1 arubini bel 497M Apr  1 16:54 lm32-gcc-4.9.4.xz
-rw-rw-r-- 1 arubini bel 572M Apr  1 16:58 lm32-gcc-5.4.0.xz
-rw-rw-r-- 1 arubini bel 676M Apr  1 17:04 lm32-gcc-6.5.0.xz
-rw-rw-r-- 1 arubini bel 704M Apr  1 17:09 lm32-gcc-7.4.0.xz
-rw-rw-r-- 1 arubini bel 776M Apr  1 17:15 lm32-gcc-8.2.0.xz
```

The “updated” versions of gcc-4.5.3 will not run on Debian-9 because I changed to *mpfr* link, as described in the following section.

Again, I suggest deploying `lm32-gcc-4.5.3.xz`.

7 The MPFR library

While choosing the versions of support libraries for building the compiler, I found some issues with *libmpfr*. They are solved in the binaries I suggest to deploy, but I think it’s useful to describe the issue, that may surface again in the next iteration of this work.

This library ((multi precision floating-point with rounding) is required to build *gcc*, but we do not use any of its features in *wrpc-sw* and other LM32 projects, so we really don’t care about versions. For this reason, we do not need the latest one, but the one that is easier for the build.

The “original” configuration used version 3.0.1, which was linked statically by the compiler, so no mismatch between the build directory and the host operating system could happen.

In the “updated” configurations, I used the latest MPFR library (multi precision floating-point with rounding) that was released as a tarball on *gnu.org* in the previous iteration of this document. This is version 4.0.1, which has some difference in the build system that forces dynamic linking. I tried one build with version 4.2.0, the lates in 2023, but the result is the same.

As a result, the executable *cc1* (i.e. the core of the compiling process) looks for *libmpfr.so* in the host libraries, and if the current host system has a different version than the build system, we have a mismatch.

In particular, Debian-9 has libmpfr.so.4 while all other distributions in my test-set have libmpfr.so.6. So compiler binaries built on Debian-9 will fail in newer systems with this error:

```
lm32-elf/4.5.3/cc1: error while loading shared libraries:
        libmpfr.so.4: cannot open shared object file: No such file or directory
```

But, in practice, the libraries are compatible, and I don’t know the reason of the incompatible chang in the major number.

Because of compatibility, the problem can be fixed in several ways:

- Making a symbolic link from `libmpfr.so.4` to `libmpfr.so.6`;

- Replacing the string in the binary file *cc1* (“`sed -i s/libmpfr.so.4/libmpfr.so.6/ ..../cc1`”);
- Building *gcc* with the older version.

For simplicity, I chose to build with *mpfr-3.0.1*, which is linked statically to *cc1*.

Therefore, in all configurations I offer in this package, besides the “updated” 4.5.3 ones, ‘I stick to *mpfr-3.0.1*, and binaries build on Debian-9 will run unchanged on any distribution in the set we are interested in.

8 Newer gcc Versions

This chapter documents what I achieved with newer gcc versions. In general I’m not very happy and I suggest sticking with 4.5.3.

As explained in Section 2.2 [gcc Version Numbering], page 2, the major number is either *4.y* for version numbers *4.y.z* or *x* for versions *x.y.0* with x >= 5.

The main problems I found with newer compiler versions is that the binary size they emit is bigger than what we get with 4.5.3.

My choice for support libraries is as follows:

- binutils: the version suggested by each compiler version;
- newlib: always 3.0, to avoid problems with obsolete code, despite the extra length of *strcasecmp*;
- mpc: always 1.0.3, as in the “updated” 4.5.3 build;
- mpfr: always 3.1.6, to avoid the problem described in Chapter 7 [libmpfr], page 9;
- gmp: always 6.1.2, as in the “updated” 4.5.3 build.

8.1 Version 4.6

In 2019 I tried building *gcc-4.6.4* with *newlib-3.0*, but it fails with this bad error while compiling *_ffsdi2.o* amd other mathematical functions:

```
xgcc: internal compiler error: Segmentation fault (program cc1)
Please submit a full bug report,
with preprocessed source if appropriate.
```

Considering newer compilers work, I gave up. I dind’t try again in 2023.

8.2 Version 4.7

In 2019 I tried building 4.7.4 but it failed in the configuration phase for *libgcc*, in this check:

```
checking whether to use setjmp/longjmp exceptions... unknown
configure: error: unable to detect exception model
```

All later versions where this suceeds reply “yes” to this check, so we may imagine to just force it on. Unfortunately, this is really a feature of the compiler that was missing for *lm32* at the time, and the *autoconf* test is identical here and in later versions where it works.

This version, thus, has no working lm32 support and must be ignored. No further test were made in 2023

8.3 Version 4.8

Version 4.8 (I tried the last, 4.8.3) as the same problem as version 4.7.

8.4 Version 4.9

This version works, if we select an older version of *mpfr*. It builds successfully and can compile *wrpc-sw*, but there are issues.

The generated *wrpc-sw* binary is much bigger. Using the default configuration it won't even fit in RAM, both with original code and after removing *strcasecmp*

```
[...]/lm32-elf/bin/ld: region ‘ram’ overflowed by 3804 bytes

[...]/lm32-elf/bin/ld: region ‘ram’ overflowed by 1268 bytes
```

By changing `.config` we can increase ram size and check the size and where the problem lies. The example below uses *gsi_defconfig* with a cahnge in memory size, and the *strcmp* modification:

```
  text    data     bss     dec     hex filename
101444    6888    4652  112984   1b958 wrpc-sw-4.5.3/wrc.elf
110360    7068    4668  122096   1dcf0 wrpc-sw-4.9.4-nocase/wrc.elf
111460    8504    4668  124632   1e6d8 wrpc-sw-4.9.4/wrc.elf
```

This 10kB size increase, mostly in actual code, is the effect of a change in compiler behaviour: all register accesses are now performed with a double indirection.

For example, let's see a single access: `timer_get_tics()`, where `syscon` points to a register block whose address is determined at run time:

```
volatile struct SYSCON_WB *syscon;
uint32_t timer_get_tics(void) {return syscon->TVR;}
```

This is the output of the build: ("`lm32-objdump -dr dev/syscon.o`"):

```
00000000 <timer_get_tics>:
   0:   78 02 00 00     mvhi r2,0x0
                        0: R_LM32_HI16  .rodata.cst4+0x10
   4:   38 42 00 00     ori r2,r2,0x0
                        4: R_LM32_LO16  .rodata.cst4+0x10
   8:   28 41 00 00     lw r1,(r2+0)
   c:   28 21 00 00     lw r1,(r1+0)
  10:   28 21 00 1c     lw r1,(r1+28)
  14:   c3 a0 00 00     ret
```

Instructions at address 0 and 4 load r2 with a pointer from `.rodata.cst4`. This is the address of *syscon*. Then the real value of *syscon* is retrieved (offset 8), the variable is read (offset c), and offset 28 for `TVR` is applied (offset 10).

This is one instruction longer than needed and requires to store the address of `syscon`. To confirm, we can check that `.rodata.cst4` is mainly an array of *syscon* pointers ("`lm32-objdump --full-contents -r dev/syscon.o`").

With *gcc-4.5.3* we had this instead:

```
00000000 <timer_get_tics>:
   0:   78 01 00 00     mvhi r1,0x0
                        0: R_LM32_HI16  syscon
   4:   38 21 00 00     ori r1,r1,0x0
                        4: R_LM32_LO16  syscon
   8:   28 21 00 00     lw r1,(r1+0)
   c:   28 21 00 1c     lw r1,(r1+28)
  10:   c3 a0 00 00     ret
```

This is not a problem I am able to solve, because it's a core compiler issue and I verified it is not related to our use of "volatile". Besides, have no "near" working version to compare against, it's not feasible a review of patches in *gcc* history to find the source of the error.

Worse, this behaviour persists in all later *gcc* versions.

8.5 Version 5

I built version 5.4.0, the last one in the series. It features the same size issue described in Section 8.4 [Version 4.9], page 11, but some newer optimizations reduce the binary size a little (0.6kB). Moreover, it reports some new warning messages about our code base (actually, the same as in version 4.9), about variables set but not used.

8.6 Version 6

Version 6.3 is known to work, because it's the one used by Joel Stanley (see Section 2.1 [Other Works], page 1). So I tried building it and then version 6.5.0, the latest release of the gcc-6 series. Here I only report about 6.5.0.

Here I had a build error, because of *int32_t* and *uint32_t* redefinition. So I applied a patch to fix it, similar to the one already applied for `intptr_t`. Also, in complained for a mismatch in a function definition. These are fixed by my patch-set.

Additionally, this version spits a huge number of warning, mostly about formats of *printf* and a few type mismatch in pointers. I did not fix these.

As expected, it works, but it still has the double-indirection problem of *gcc-4.9*. The binary is 1.5kB smaller than what we get with *gcc-5* because of new optimizations.

The double-indirection problem has a different form, as the compiler now spits several data sections with the address of *syscon*, so the linker can garbage-collect out the ones that are not actually used. This accounts for most of the size reduction when compared to version 5, but we are still 10kB bigger than what gcc-4.5 achieves.

8.7 Version 7

I tested version 7.4, the latest one. As usual: more warnings (just a few: different pointer type passed to *softpll*, smaller size (1.5kB less), but the major problem of double-indirection persists.

8.8 Version 8

The last one when I made this work in 2019 was version 8.2. Same result as above, with the double-indirection problem. Binary is 200 bytes smaller but still bigger than what emitted by our original 4.5.3.

8.9 Version 9 and later

I didn't not try building any later version.

9 Deploying a compiler

Considering all things described above, I suggest deploying a binary tarball of *gcc-4.5.3* with the original set of support binaries, as built on Debian-9. Such archive can be uncompressed in any pathname and used in place. It can run in all the distributions I was asked to test.

This is, as a recap, the size of `wrc.elf` built with the various compilers (without applying the *strcasecmp* patch).

```
   text    data     bss     dec     hex filename
 102088    8320    4652  115060   1c174 wrpc-sw-old-compiler/wrc.elf
 101444    6888    4652  112984   1b958 wrpc-sw-4.5.3/wrc.elf
 101444    6888    4652  112984   1b958 wrpc-sw-4.5.3-updated/wrc.elf
 102080    8320    4652  115052   1c16c wrpc-sw-4.5.3-newlib-2.5/wrc.elf
 102088    8320    4652  115060   1c174 wrpc-sw-4.5.3-newlib-3.0/wrc.elf
 111460    8504    4668  124632   1e6d8 wrpc-sw-4.9.4/wrc.elf
```

```
110844    8504    4668  124016   1e470 wrpc-sw-5.4.0/wrc.elf
109544    8320    4636  122500   1de84 wrpc-sw-6.3.0/wrc.elf
108120    8320    4636  121076   1d8f4 wrpc-sw-7.4.0/wrc.elf
108048    8320    4636  121004   1d8ac wrpc-sw-8.2.0/wrc.elf
```

10 LLVM/Clang

I still did not work on this. I suspect we are moving to RiscV, so we should better avoid more lm32-related work.