LM32-toolchain

Rebuilding a compiler in 2019

February 2019 (f3d6e44)

Alessandro Rubini, for GSI

Table of Contents

Overview

This package is about considering a newer compiler for the LM32 processor, that is currently used in White Rabbit (`https://www.ohwr.org/projects/white-rabbit`) and other projects inside GSI (`gsi.de`).

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.

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 newer support libraries;
- We can build this with a recent host compiler (*gcc-8.2.1*);
- Newer 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.

2 Initial Remarks

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:

`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 will be one of my starting points for LLVM support, but I didn't work on it as yet.

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, with yet-unreleased tool chain versions. 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 on OHWR or 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 *proposed_master* commit of *wrpc-sw* (commit `4e6a3ea` as I write this. Testing on other packages is welcome.

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 installed 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 now 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% ./build-generic ~/wip/lm32-toolchain/configs/gcc-4.5.3-orig
Using /home/rubini/wip/lm32-toolchain/configs/gcc-4.5.3-orig as config file
Using PREFIX=//lap-x/lm32/2019/build/install/lm32-gcc-190204-804f19-8e23-dirty
Building in "/lap-x/lm32/2019/build/build-190204-17-26"
Log file is "/lap-x/lm32/2019/build/build-190204-17-26.log"
```

From the above it can be noted that

- I work in a different directory (external hard drive, for convenience), and `build-generic` is a symlink to the repo;
- the date I run the build is `190204`, turned to the first item in the PREFIX value;
- the hash of the configuration is `804f19`: this is the variable `HASHCFG` calculated in the script;
- the hash of this repo is `8e23-dirty`: 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; such a directory can be a symbolic link. The script uses the version number, suffix and base URL as in the example above (for *gcc* there is a different directory for each version, thus the duplicate number); it downloads one *tar* file for each *package* listed in the `plist` variable (and for which is 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, 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 them all to restart a clean build, if you want.

3.2 Applying Patches

The current version of the 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. This initial commit of the whole source tree can be a very long operation, but fortunately it only applies once for each package, and not many packages are patched.

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, `190116-12-00` 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 650MB 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, but also the script and the configuration file. I did this because I tend to forget saving all information about a build, for example because I edit the configuration file for the next version I try and save with the same name so to recycle the command line.

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

```
laptopo% ./tools/recover-config build-190113-10-00.log > prev-config
```

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

To look for errors, please grep for `Error` in the log file (not 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 is the most imporant 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 `lx-pool.gsi.de`. 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.

```
lxi097$ ./tools/build-generic configs/gcc-4.5.3-newlib-3.0
Using PREFIX=//u/arubini/lm32-toolchain/install/lm32-gcc-190204-e4348a-3569
Building in "/u/arubini/lm32-toolchain/build-190204-23-55"
Log file is "/u/arubini/lm32-toolchain/build-190204-23-55.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.

4 Host System

Currently, I only run the build on *Debian-8*, using a 64-bit installation. The host compiler is *gcc-4.9.2*. This matches what *lx-pool.gsi.de* is, so it's a good first step.

5 gcc-4.5.3

As a first step, let's rebuild what we already were using. Current tools (as of Debian 8) fail the build, because some warnings are now considered errors. Thus, I provide patches for a successful build.

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 script with `configs/gcc-4.5.3-orig` we rebuild the same compiler we have been using in the last years. But if we do it in a 64-bit host we'll have a binary more suitable for current build machines. the build is 860MB and the installation is 210MB. The binary built in Debian 8 relies on `glibc-2.19`.

When tested against *wrpc-sw*, commit `54d3079`, the output binary is larger by 16 bytes than the one build with our legacy compiler binary.

One logical expression is translated differently, the code is two instructions more than it used to be, and that expression is built twice. Everything else is identical. The expression is the core of *spll_debug()*.

5.2 Upgrading Support Libraries

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. See next section.

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.

5.3 Upgrading newlib

I tried upgrading newlib to the version 2.5 (latest 2.x at time of writing) and 3.0 (latest official 3.x at time of writing).

Unfortunately, the binary size of *wrpc-sw* got bigger, because of locate-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
 102604    6888    4652  114144   1bde0 wrc-4.5.3-nl2.0.elf
 103240    8320    4652  116212   1c5f4 wrc-4.5.3-nl2.5.elf
 103240    8320    4652  116212   1c5f4 wrc-4.5.3-nl3.0.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.

Replacing all of them with *strcmp*, with the following command:

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

makes *wrc.elf* 400 bytes shorter than the original, irrespective of the newlib version we used to build the compiler (because no arcane newlib function is being used in the binary).

The change is pushed to *ohwr* and you can cherry-pick commit `bb84ad0` to build with *strcmp* without running a dirty source tree.

Configurations files `gcc-4.5.3-newlib-2.5` and `gcc-4.5.3-newlib-3.0`, part of this package, can be used for these builds.

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.

For example:

```
cd install
D=lm32-gcc-190116-1648
mv $D lm32-elf-gcc
tar cJf lm32-elf-gcc.tar.xz lm32-elf-gcc
mv lm32-elf-gcc $D
```

This works for all *gcc* builds described here.

7 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 1, 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 generated is bigger than what we get with 4.5.3.

7.1 Version 4.6

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. The configuration file is part of this package anyways.

7.2 Version 4.7

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.

7.3 Version 4.8

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

7.4 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:

```
[...]/lm32-elf/bin/ld: region ‘ram’ overflowed by 2412 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* commit `bb84ad0` as explained in Section 5.3 [Upgrading newlib], page 6,

```
lxi087$ size *elf
   text    data     bss     dec     hex filename
 102212    6884    4652  113748   1bc54 wrc-4.5.3.elf
 111500    7068    4668  123236   1e164 wrc-4.9.4.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'm currently 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.

7.5 Version 5

I built version 5.4.0, the last one in the series. It features the same size issue described in Section 7.4 [Version 4.9], page 8, but some newer optimizations reduce the binary size a little (0.7kB). Moreover, it reports some new warning messages about our code base (actually, the same as in version 4.9).

7.6 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.

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. Also, more warnings about our coding practices in *wrpc-sw* – most about *printf* formats.

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 our the ones that are not actually used. This accounts for the most of the size reduction when compared to version 5, but we are still 8kB bigger (6.3%) than what gcc-4.5 achieves.

7.7 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.

7.8 Version 8

The last one at time of writing is version 8.2. Same result as above, with the double-indirection problem. Binary is 200 bytes smaller.

8 Storage Requirements

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

- 1.0 GB for downloads
- 8.3 GB for uncompressed sources.
- 4.5 GB for installed trees (150MB to 650MB each version)
- 0.23 GB in log files (8MB to 40MB each)
- 17.5 GB for build directories (650MB to 2.8GB each)

This is 31.5GB in total.

The exact size will vary according to you filesystem layout and the length of your pathnames (which end up many times as strings in the binaries).

9 Portability of Compiler Binaries

In general, building on an *older* system is a good thing to run everywhere, because newer libraries are usually backward-compatible. So, my build on Debian-8 has only minor issues on more recent code bases.

I tried to run the most popular distributions, and build *wrpc-sw* on there, using *gcc-4.5.3* and *gcc-8.2.0* (the oldest and newest ones, as described in the next chapter.

For each test, I ran a base installation (choosing one desktop environment or another), and noted all packages I had to install to run a successful build of *wrpc-sw* (commit `bb84ad0`).

All commands below must be run as root or with *sudo*.

9.1 Ubuntu-16.04

No issues found. I only had to install two packages needed by the *wrpc-sw* build:

```
apt install libreadline-dev git
```

9.2 Ubuntu-18.04

I had a problem with *libmpfr* (Multiple Precision Floating-point Reliably). The distribution hosts version 6 while the compiler I built uses version 4, which is dynamically linked by the `cc1` executable.

Despite the difference in major version, the libraries are compatible. The easiest fix is making a symbolic link of version 4 to version 6:

```
ln -s libmpfr.so.6 /usr/lib/x86_64-linux-gnu/libmpfr.so.4
```

Additionally, I had to install some packages that are not there by default:

```
apt install gcc make libreadline-dev git
```

9.3 Debian 9

No problem besides missing tools:

```
apt install gcc make libreadline-dev git
```

9.4 Fedora 28 and 29

No problem besides missing tools:

```
yum install gcc readline-devel git
```

9.5 Scientific Linux 7.6

No problem besides missing tools:

```
yum install gcc
```

9.6 Mint 19

This Ubuntu-like distribution has to same problem as Ubuntu-18.04, but additionally it doesn't install any *libmpfr* by default. This was the fix for me:

```
apt install libreadline-dev libmpfr6
ln -s libmpfr.so.6 /usr/lib/x86_64-linux-gnu/libmpfr.so.4
```

10 Rebuild on Recent Hosts

With a few additional patches to *binutils* and *gcc*, I was able to rebuild *gcc-4.5.3* on Fedora 29, which features *gcc-8.2.1* as host compiler, which is the most recent at time of writing.

You can compile with the current patch-set part of this package, but you must first install a host library:

```
yum install mpfr-devel
```

The problem is a fine issue related to `mpfr.h`: auto-configuration of the compiler has a wrong patch for including header, and ends up picking the host one instead. If the host header is missing, compilation would fail. We can ignore the potential version difference between the host version and the cross-compiled version; the library is stable over time and not really used in our LM32 environments.

The errors were a detail in *gcc* documentation (which is built by default, so the error was fatal for the overall compilation) and a zillion warnings in *binutils*, that turn into errors if `-Werror` is used. Thus, I removed *-Werror*, as a simple solution to an otherwise impossible task.

The resulting binary can successfully run on Debian-8, even if if features an older *libc* than the one used to build it (2.19 vs. 2.28), thanks to forward- and backward-compatibility rules in system libraries.

I didn't try to build other compiler versions; if needed, there should be similar problems as the ones I patched in 4.5.3, but fewer and fewer as you build newer versions.

11 Things to Do

I think the following gcc-related things are still to be fixed within this project, althought hey are *wrpc-sw* commits.

- Remove *inline* attribute from *spll_debug*. I actually is not inline with *gcc-4.5.3* and I suspect it takes more space if built inline (by more recent compilers).
- Fix all compiler warnings with newer compilers, even if we end up using the usual 4.5.3 version.
- Try rebuilding the compilers with newer host compilers (I expect the older ones, like *gcc-4.5.3* to trigger build errors in new environments).

12 LLVM/Clang

I still did not work on this. Stay tuned.