Building your own Microblaze toolchain from source

Mar 30, 2017

I recently needed to work with a synthesized Microblaze CPU set up as a microcontroller. After using Vivado to generate the initial SDK I decided to work through building a toolchain from source.

There are two Microblaze worlds, so to speak. The big ones configured with an MMU are able to run Linux, and for those we have pre-packaged toolchains such as `gcc-microblaze-linux-gnu` in Fedora. For the small ones (no MMU, meant to run bare-metal or an RTOS) one typically uses the Vivado-supplied GCC toolchain however that may be outdated (GCC 5.2 in Vivado 2016.04 for example) and may not have features that you require. Furthermore the Vivado-supplied toolchain was for some reason built for a 32-bit x86 host.

Sources

Microblaze support seems to be open-sourced and generally available, however Xilinx do not make much effort to get their changes into mainline so we need to use Xilinx's forks of various projects in order to build something that can compile and link for Microblaze. Luckily Xilinx maintains a Registered Guest Resources site where we can obtain source snapshots for anything open-source that they utilize. There we find source archives for the entire toolchain, either the GCC 5.2 that is shipped in Vivado 2016.04 or at this time a GCC 6.2 as well. These contain:

- GCC
- GDB
- binutils (Xilinx's changes don't appear to be in mainline)
- newlib (the standard C library implementation for small systems)

Extract the archive and then untar the contents (each source snapshot is a tarball inside that archive).

cosstool-ng

The toolchain itself is built by utilizing the crosstool-ng toolchain builder. Xilinx maintain their own fork of crosstool-ng with some Microblaze-specific changes. I chose to use that, though the top-level Makefile needed changes from mainline crosstool-ng in order to build (I patched the Xilinx fork with changes from mainline).

As a starting point, I used the samples/microblaze-xilinx-elf configuration as the crosstool-ng `.config` and then ran `make menuconfig`. Note that experimental support (`CT_EXPERIMENTAL`) is enabled: this is required for Microblaze to be an option in crosstool-ng.

Setup

We need to build crosstool-ng but we don't have to install it (it can be run from its source directory). To do that:

```
./configure --enable-local
make
```

The `ct-ng` binary will be run to use crosstool-ng.

Configuration

Copy the sample configuration file to `.config` and then run `ct-ng menuconfig` to further configure the toolchain. The options you should configure include:

- no MMU

- set endianness to match your synthesized target (the default is big-endian)
- build a multilib toolchain (that way you can adjust compilation based on selected Microblaze CPU options, for example the barrel shifter is optional)
- build a sysrooted toolchain using cross-compilation
- the target operating system is `bare-metal`
- the target binary format is ELF
- for `binutils`, configure with `--disable-sim`, the simulator will not build for microblaze and you're unlikely to need it, so turn it off.
- for `gcc`, enable LTO support but disable graphite: unfortunately the library graphite needs to do its work will not build for microblaze at this time.
- for `newlib`, enable space savings

Also for each of the tools (gcc, binutils, gdb, etc) configure the source location to point to the absolute path to your extracted Xilinx source snapshots.

Building

Save your configuration and then run `./ct-ng make`, this will hopefully build the toolchain but it will take a very long time of course. In fact, crosstool-ng will build a host toolchain first and then use that to build the cross-toolchain.

If successful, you will now have a directory named `tool-build` one level up from your crosstool-ng directory. This contains a `bin` directory with the resulting toolchain. Assuming typical options, the toolchain will have a prefix of `microblaze-xilinx-elf` for a big-endian toolchain or `microblazeel-xilinx-elf` for little-endian.

Trying things out

The Xilinx SDK builds an archive named `libxil.a` (the `xil` library) which in turn contains an entry point and some peripheral initialization code. The source code for this is provided and you can rebuild that library as needed, though you can use the pre-built one as-is if you wish. You will find that the linker will require `libxil` either way. To test out the new toolchain, supply a path to the `libs` directory containing your copy of `libxil.a` via the `-L` option.

We can ask the toolchain to dump the options which which it was built in order to sanity-check our configuration:

```
microblaze-xilinx-elf-gcc -v
```

Finally, we can build a sample program:

```
microblaze-xilinx-elf-gcc -L/path/to/libs foo.c -o foo.elf
```

And we can generate a binary suitable for loading on hardware:

```
microblaze-xilinx-elf-objcopy -S -I elf32-big foo.elf -O binary foo.bin
```

(adjust the above for your toolchain prefix/endianness as needed). The ELF should be usible in the Vivado debugger (a wrapper over GDB), or GDB itself if you have set up the Xilinx debug bridge, and the binary can be loaded onto the target if you have some means of doing that beside the debugger.

/var/log/andrey

/var/log/andrey yurovsky My blog about Linux systems programming, firmware, and cars.