AFRL HDL RF

November 27, 2024

Jay Convertino

Contents

1 Usage

1.1 Introduction

AFRL HDL RF contains FPGA RF projects. The goal of this project is to have all RF projects in one place. On top of that this uses a build system to simplify all of the steps for generating the RF system into one step. These projects provide a base system that targets various RF frontends. Targets have the RF frontend built into the board, the fpga, or are a seperate development board added to the FPGA development board. This project uses a python based build system to tie together image generation. Meaning that if you choose a target that FPGA image, the software (linux at the moment) are all built for the target. Then the results are put into a SDCARD image (future allow for other targets such as flash). This image can be written to an SDCARD using various utilities.

1.2 Quick Start

1. Clone this repo
2. Install Requirements listed above.
3. Make sure all requirements are accessable from the command line.
4. execute: python system_builder.py to build all targets.

Each part of a target is stored in a directory the represents the part of it that needs to be created. FPGA source files are stored in hdl, sw has the software needed for the various baremetal or operating systems. System builder is given targets that choose the needed project files and sets up the software parameters needed for the build. This allows the same parts to be reused for different targets. FPGA images uses for both baremetal and Linux for example. All targets generate a log in the log folder. Without debug enabled this will only contain the commands executed during the build, a good place to find out how to do parts manually. The output folder will have the project build outputs, artifacts, binaries, and any images. These are seperated into folders that contain the name of the target that contains its information.

1.3 Directory Guide

Below highlights important folders from the root of the directory. Output and log are created during the the system_builder execution.

1. **docs** Contains all documentation related to this project.
    - **manual** Contains user manual and github page that are generated from the latex sources.
2. **hdl** Contains source files fusesoc FPGA projects.
    - **ip** General IP cores used in projects.
    - **projects** Main projects used for FPGA builds.
    - **sim** IP cores and scripts for simulations.
    - **util** Utilities for FPGA IP cores.
3. **img_cfg** Contains genimage config files
4. **py** Contains source files builder.
5. **sw** Contains source files for linux buildroot
6. **output** Is a folder generated that contains all build outputs.
    - **hdl** Contains all FPGA csode
    - **linux** Contains all linux code
    - **genimage** Contains all its builds artifacts.
    - **sdcard** Contains sdcard image files and source files for sd-card image.
7. **log** Is a folder generated that contains all information logged from execution.

1.4 Dependencies

- system_build.py
    - gitpython
    - progressbar2
- OS
    - Tested on Ubuntu 22.04
- HDL
    - Vivado (Tested with 2022.2.2)
    - Quartus (Tested with 22.4)
    - fusesoc (2.4 or greater)
    - gtkwave
    - Icarus

  - arm-none-eabi-gcc version 11 (needed for bootgen, done by python script at end of HDL build)
  - aarch64-linux-gnu-gcc version 11 (needed for bootgen, done by python script at end of HDL build)

- Software
  - build-essentials
  - genimage
  - make
  - mkimage
  - bootgen
  - gcc
  - which
  - sed
  - gcc
  - g++
  - bash
  - patch
  - gzip
  - bzip2
  - perl
  - tar
  - cpio
  - unzip
  - rsync
  - file
  - bc
  - wget
  - find
  - xargs
  - diff
  - cmp
  - diff3
  - sdiff
  - ld
  - as
  - gold
  - mcopy

1.5 System Builder

The main python program in charge of building the targets is system_builder.py. This calls a library called builder and other libraries to carry out target generation. Targets are specified by receipes in the build.yml file. It will also pull all sup-repositories automatically. Dependcy checking is included, but this is very simple at this moment and uses the deps.txt file to parse command names and checks if they exist. It does not check versions. Each target will be built with its current status show in its own progress bar. This shows the time elapsed, percent complete, status, and name of current target being build.

What system_builder.py does exactly is to call other build systems. It essentially isn't made to replace things such as cmake, vivado, make, buildroot, etc. It is made to tie those together for a target in a receipe. This receipe tells system_builder how to create each piece, and what order to do it in. Each piece it calls is responsable for generating its output products using its original build system. New receipes for build methods can be added using a yaml file in the python directory. These are used to fill in how to use a build system and what to expect. This allows system_builder.py to be quickly updated with new tools for future targets for more interesting receipes.

build.yml is the default yml target receipe file system builder looks for. This file specifies targets with parts (receipe) that contain commands for builds. These parts can be concurrent for multithreading, or sequenctial for one at a time. The order these are executed is from the top down. Meaning the top command will be executed before the one below it.

Sample build.yml

```
zed_fmcomms2-3_linux_busybox_sdcard:
  concurrent:
    fusesoc: &fusesoc_fmcomms2-3
      path: hdl
      project: AFRL:project:fmcomms2-3:1.0.0
      target: zed_bootgen
    buildroot: &buildroot_fmcomms2-3
      path: sw/linux/buildroot-afrl
      config: zynq_zed_ad_fmcomms2-3_defconfig
  sequential:
    script: &output_files_fmcomms2-3
      exec: python
      file: py/output_gen.py
      args: "--rootfs output/linux --bootfs output/hdl --
        ↪ dest output/sdcard"
```

```
    genimage:
      path: img_cfg
zc702_fmcomms2-3_linux_busybox_sdcard:
  concurrent:
    fusesoc:
      <<: *fusesoc_fmcomms2-3
      target: zc702_bootgen
    buildroot:
      <<: *buildroot_fmcomms2-3
      config: zynq_zc702_ad_fmcomms2-3_defconfig
  sequential:
    script:
      <<: *output_files_fmcomms2-3
    genimage:
      path: img_cfg
```

System builder can be easily run by simply executing the following from the root of the AFRL HDL RF directory.

```
$ python3 system_builder.py
```

Which will build all targets listing in the build.yml file. The following are all the options available (–help will print this to the console).

```
--list_targets        List all targets.
--list_commands       List all available yaml build
    ↪ commands.
--list_deps           List all available dependencies.
--clean               remove all generated outputs,
    ↪ including logs.
--deps DEPS_FILE      Path to dependencies txt file, used
    ↪ to check if command
                        line applications exist.
--build CONFIG_FILE   Path to build configuration yaml
    ↪ file. build.yml is
                        default.
--target TARGET       Target name from list. None will
    ↪ build all targets by
                        default.
--debug               Turn on debug logging messages
--dryrun              Run build without executing commands
    ↪ .
--noupdate            Run build without updating
    ↪ submodules.
--nodepcheck          Run build without checking
    ↪ dependencies.
```

For instance, if you would like to build a single target you can use the following.

```
$ python system_builder.py --target zed_fmcomms2-3
    ↪ _linux_busybox_sdcard
```

After executing the build command you will see the following output to your console. This will inform you of how the build is going, what the build has done, and what targets have been built.

Successful build:

```
Checking for dependencies...
Checking for dependencies complete.
Checking for submodules...
Checking for submodules complete.
Starting build system targets...

[0:13:23] 100% [################] Status: SUCCESS    |
    ↪ Target: zed_fmcomms2-3_linux_busybox_sdcard
[0:13:37] 100% [################] Status: SUCCESS    |
    ↪ Target: zc702_fmcomms2-3_linux_busybox_sdcard
[0:13:38] 100% [################] Status: SUCCESS    |
    ↪ Target: zc706_fmcomms2-3_linux_busybox_sdcard

Completed build system targets.
```

If the build fails, you will have the following.

Failed build:

```
Starting build system targets...

 [0:26:36]  85% [#############...] Status:  ERROR     |
    ↪ Target: zcu102_fmcomms5_linux_busybox_sdcard

ERROR: build system failure, see log file log/240513
    ↪ _1715617815.log.
```

Always check the log file listing to debug any error messages. Also using –dryrun with –debug can also speed up troubleshooting of bugs that are things such as bad paths or missing dependencies.

deps.txt is the default value system builder looks for. This file specifies any executable dependency of the project. These are list line by line in a simple text file. No version checks at the moment. If any are missing the application will print out the missing executable and quit.

Sample deps.txt

```
genimage
fusesoc
make
quartus_cpf
```

quartus
mkimage
vivado
xsct
bootgen
gcc

1.6 Understanding Output Products

The output folder contains four folders.

1. genimage
2. hdl
3. linux
4. sdcard

Genimage contains temperary files for the genimage utility. Each target is listed by its name in the tmp folder contained within.

hdl is the destination for the fusesoc build output. Each folder contained within is named after the target from system builder. Within that is the project files for the tool used to build the FPGA image. If you need to alter the FPGA base image this is the place to start.

linux has the buildroot results for each project. The name will be the target name. The results are items such as the executables, kernel, and the root file system images.

sdcard contains images for sdcards. It will contain folders with the target name, and within it are the parts used for the image and a final image, sdcard.img. The sdcard.img file is the one used with your sdcard imaging software to put it on its destination sdcard. BOOTFS contains all of the boot files from the build processes and rootfs contains the image for the base file system in ext4.

1.7 Using SDCARD images

Pick the way you prefer to transfer your image. If you're going to use dd I recommend blanking the card first so old data is removed.

```
$ dd if=sdcard.img of=/dev/sde bs=512 status=progress
```

My prefered method is gnome-disk-utility.

1. Open gnome-disk-utility
2. Insert the sdcard into your reader.
3. Select the destination device under Disks.
4. In the right side hamburger menu click 'Restore Disk Image'.
5. In the new windown, find your image. Once you've found it click start restoring.
6. Confirm you want to restore the image.
7. Input your password for sudo access.
8. Wait for the image to be restored.
9. Once it is complete, click the top 'Eject' button.
10. Exit the application.
11. Insert the sdcard in your target platform.
12. Poweron and enjoy.

2 RF Systems

All RF systems in this project will be some sort of base FPGA project with a software package to control it. These systems are based on various open source projects.

2.1 FMCOMMS2-3

The FMCOMMS2-3 is a Analog Devices FMC development board for VHF and UHF ranges. This project uses the Analog Devices HDL base project for the FPGA. This base has customizations by AFRL for the Arria10 based targets. These targets do not exist in the original. It also fixes a few bugs and makes the data path for RX/TX a mirror image of each other.

2.2 FMCOMMS5

The FMCOMMS5 is a Analog Devices FMC development board for VHF and UHF ranges. This project uses the Analog Devices HDL base project for the FPGA. It also fixes a few bugs and makes the data path for RX/TX a mirror image of each other.