# MIPSfpga++
# Install Linux on Softcore MIPS on DE10-Lite FPGA.
# Or just watch me do it. Whatever.

mmskv

## Contents

# 1 Glossary

- **FPGA** – Field-Programmable Gate Array, a flexible and customizable hardware platform for implementing digital logic circuits.
- **Softcore CPU** – Processor core that can be synthesized and implemented on programmable logic devices like FPGAs.
- **MIPSfpga** – A soft-core MIPS processor provided by Imagination Technologies under a free as in beer license.
- **mipsfpga-plus** – An Open Source project that introduces UART and SDRAM pin definition to some FPGA boards.
- **BusyBox** – A set of Unix utilities that have been bundled into a single executable.
- **OpenOCD** – Open On-Chip Debugger, a software tool that provides debugging, in-system programming, and boundary-scan testing for embedded devices.

# 2 Prerequisites

- DE10-Lite board
- A modern Linux distribution
- USB to UART bridge
- FT2232 Adapter for OpenOCD with MPSSE support
- Serial terminal emulator (minicom is optional)
- Docker (Optional)

## 2.1 Setting up Intel Quartus Prime Lite

Locate the latest version of Intel Quartus Prime Lite on intel.com and download it. Upon installation, include the Quartus binaries path in our system's PATH.

## 2.2 Getting GCC toolchains

To compile MIPS executables, a specific toolchain is required. Compiling this toolchain is complex and outside the scope of this document. Therefore, a Docker image containing a pre-configured toolchain with debugger is provided for immediate use.

```
docker pull mmskv/mipsel-linux-gnu-gcc:latest
```

# 3 Launching MIPS processor on Intel DE10-Lite

To begin, clone the MIPSfpga-plus repository from GitHub.

```
git clone https://github.com/MIPSfpga/mipsfpga-plus
```

For the MIPSfpga sources, obtain the MIPSfpga Getting Started package (MFGS) that comes with Verilog sources. As the package is not freely distributable, it can't be shared publicly.

Once we have acquired the package, its contents need to be moved into the `mipsfpga-plus/core` directory.

```
cp -r MIPSfpga/MIPSfpga_GSG/rtl_up/core/* mipsfpga-plus/core
cd boards/de10_lite
make project
make all # Compiles the project in console with quartus_pgm
```

After the Verilog core is compiled, connect the DE10-Lite to the computer using a USB cable to upload the program.

Upload the compiled project via a USB Blaster onto a DE10-Lite.

```
make load
```

We can alternatively use the Quartus GUI instead of Makefiles to achieve the same results.

Optionally open the project in Quartus.

```
make open
```

Once the softcore processor is uploaded to the board, it should start displaying two counters: one in hexadecimal and one in binary. The `KEY0` can be used to reset the counters.

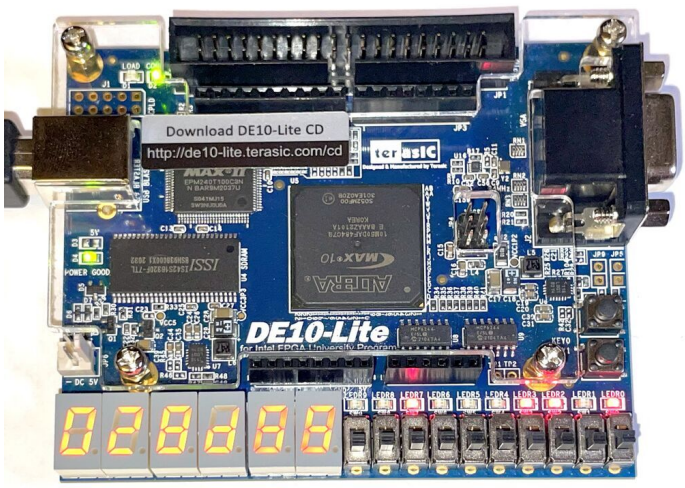


Figure 1: Counters on the board after mipsfpga load

## 3.1 Attaching Bus Blaster to Linux (udev)

Potential error may occur while running the ‘make load‘ command

```
Error (213013): Programming hardware cable not detected
make: *** [Makefile:38: load] Error 3
```

We can diagnose issue with

```
quartus_pgm --auto
```

The error ”`Unable to lock chain - Insufficient port permissions`” indicates that our user does not have permission to access the Bus Blaster device. To resolve this, either add the user to the `usb` group and log back in

```
sudo usermod -aG usb $(whoami)
```

Or set device permissions

```
sudo dmesg | grep -A5 "USB device number"
sudo chmod 666 /dev/bus/usb/001/00y # y is the device number
```

## 3.2 Testing SDRAM

To test the SDRAM functionality on the softcore processor, use the programs provided with mipsfpga-plus [1]. One such program is memtest.

```
cd mipsfpga-plus/programs/06_memtest
```

This program was written with support only for a specific archaic MIPS GCC toolchain once distributed by Imagination Technologies. We can get this toolchain in Docker at mips-mti-elf.
To compile binary and generate Motorola S-record file do

```
docker run -it -v $(pwd):/mipsfpga-plus \
    -w /mipsfpga-plus/programs/06_memtest \
    mmskv/mips-mti-elf:latest \
    make all
```

To upload the binary to the CPU, a USB to UART bridge is necessary. UART RX and TX pin definitions can be set in Quartus ’Pin Planner’ or in `mipsfpga-plus/boards/de10_lite/de10_lite.v`.
Connect USB to UART Adapter:

- GPIO 31 – RX
- GPIO 32 – TX

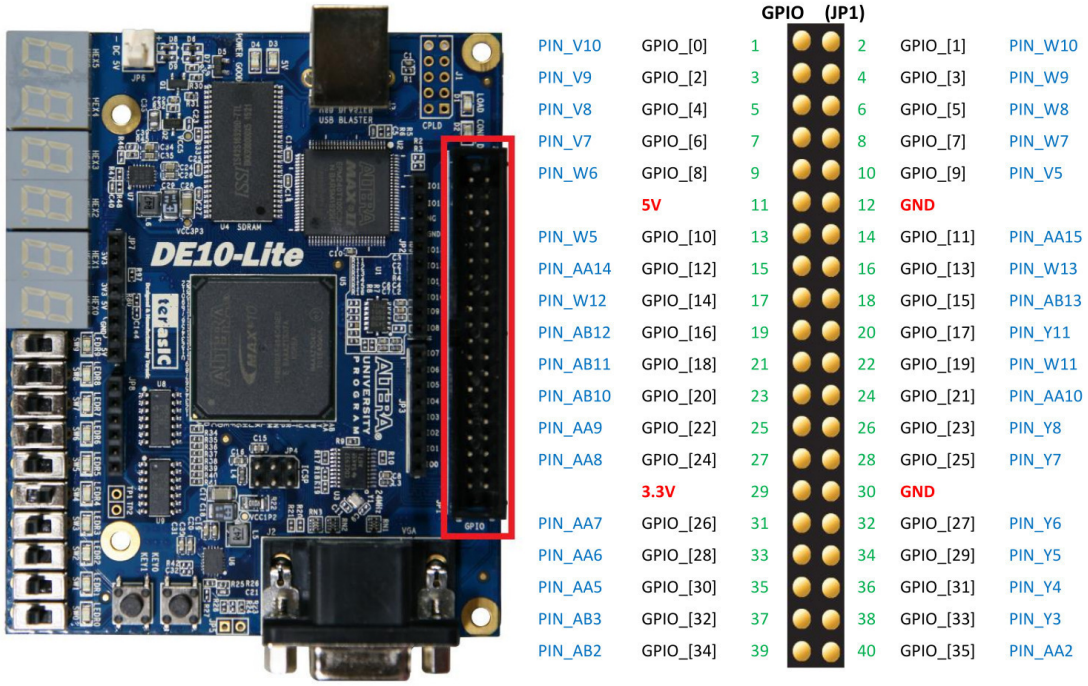


Figure 2: DE10-Lite I/O pinout.

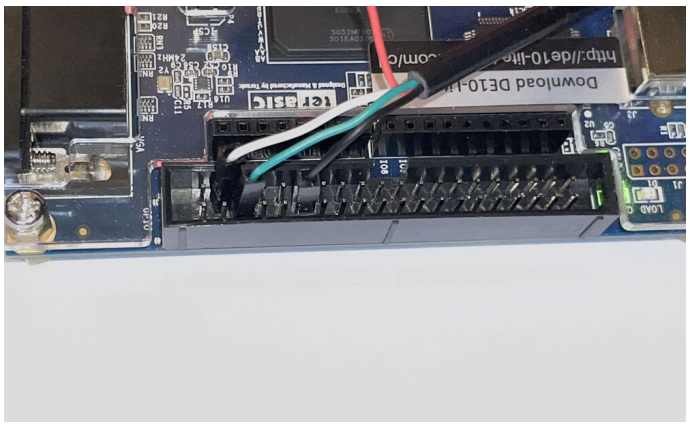

Figure 3: Correct UART connection.

It may be necessary to add our user to the `dialout` group to access the UART devices as a non-root user

```
sudo usermod -aG dialout $(whoami)
```

Or we can simply set the UART adapter's attributes to `-rw-rw-rw-`

```
sudo chmod 666 /dev/ttyUSB0
```

Load the instructions onto the CPU. A 5KB binary file transfer should take less than a second at a 115200bps baud rate. If any hangups occur, try swapping the RX and TX pin connections.

```
bash 10_upload_to_the_board_using_uart.sh
```

During operation, the LEDs will indicate the current check stage:

- LED0 – Write cycle
- LED1 – Cache reset cycle
- LED2 – Pause
- LED3 – Read cycle
- LED4 – All checks completed, no errors
- LED5 – Errors detected during the checks

The 7-segment displays will indicate: two most significant digits – test number, four least significant digits – number of detected read errors.

After approximately 17 minutes, the test should complete and display a result.

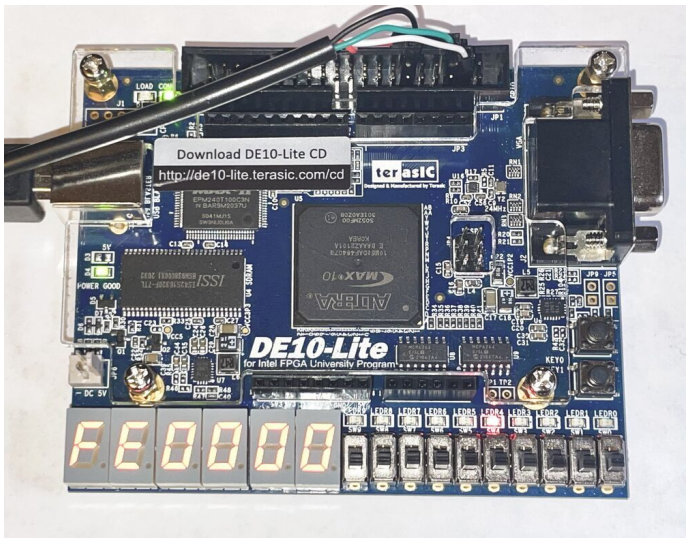


Figure 4: Successful SDRAM test. Lit LEDR4 means that all checks have passed. 0xFE is the test count and 0x0000 is the error count.

# 4 Running Linux

After validating that the SDRAM is functioning correctly, we can proceed with the objective of running Linux.

## 4.1 Building BusyBox initramfs

Since the DE10-Lite lacks built-in storage, we'll utilize filesystem in RAM images. BusyBox is an ideal option for this task due to its compact size - about 1MB of our 32MB SDRAM. Although GNU could potentially fit after some modifications, we'll proceed with BusyBox for simplicity.

```
git clone https://github.com/buildroot/buildroot --depth 1
cd buildroot
wget https://raw.githubusercontent.com/mmskv/mipsfpga-plusplus/master/patches/buildroot.patch
git apply patch
make de10lite_defconfig
make -j $(nproc)
```

Now we should have a rootfs.cpio at `./output/images/`. We will use it later.

## 4.2 Building Linux

Get the sources

```
git clone https://github.com/torvalds/linux --depth 1
cd linux
```

As noted in article [2], the modified `arch/mips/xlifpga` was used to build Linux for MIPSfpga. Unfortunately, this target since was removed in the commit 0861aa1251c749bbec4ee124ee21660c2f263da4. Reverting this commit won't work due to numerous subsequent changes. However, a patch that introduces MIPSfpga++ as a platform can be applied to the Linux tree

```
wget https://raw.githubusercontent.com/mmskv/mipsfpga-plusplus/master/patches/linux.patch
git apply
```

With the `arch/mips/mipsfpga-plusplus` target added, Linux can now be built.
Generate `.config`

```
make ARCH=mips BOARDS=mipsfpga-plusplus 32r2el_defconfig
```

Path to the initramfs image has to be specified either with a single command or with menuconfig

```
./scripts/config --set-val INITRAMFS_SOURCE \"/path/to/buildroot/output/images/rootfs.cpio\"
# or with menuconfig
make MENUCONFIG_COLOR=blackbg ARCH=mips menuconfig
```

When compiling the kernel a MIPS toolchain has to be specified since we are building for a non native target. For easier dependency management we will do it in a Docker container that has a MIPS toolchain and multiarch gdb.

```
docker run \
    --network=host -it \
    -v $(pwd)/..:/workdir \
    -w /workdir/linux \
    mmskv/mipsel-linux-gnu-gcc:latest \
    make -j $(nproc) ARCH=mips CROSS_COMPILE=/usr/sbin/mips64el-linux-gnu-
```

## 4.3 Loading linux

OpenOCD is added by mipsfpga-plus for loading and debugging programs running on the CPU.
Connect FT2232 to the board as follows:

- GPIO 17 – TCK
- GPIO 19 – TDO
- GPIO 21 – TDI
- GPIO 23 – TMS

Connect UART dongle:

- GPIO 33 – TX
- GPIO 35 – RX

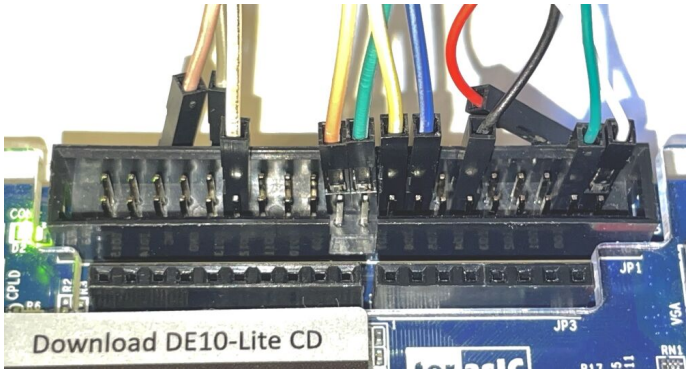


Figure 5: JTAG and UART connection.

Start OpenOCD. The OpenOCD configuration is board specific.

```
openocd -f <ft2232 config>.cfg
```

We can then use gdb to load the Linux binary onto the CPU. Note that there are two files in the current diretory; vmlinux is a binary with debug symbols, while vmlinuz is a binary with stripped debug symbols.
GDB has to be specifically compiled with MIPS target support so we will use custom gdb in a Docker container

```
docker run \
    --network=host -it \
    -v $(pwd):/linux \
    -w /linux \
    mmskv/mipsel-linux-gnu-gcc:latest \
    gdb-multiarch vmlinux
```

Upon seeing the gdb shell, connect to the openocd gdb port (default: 3333)

```
(gdb) target extended-remote :3333
```

Load the Linux binary (this will take around 5 minutes)

```
(gdb) load
```

At this point, we can execute the loaded kernel

```
(gdb) continue
```

The kernel will use the UART interface as a virtual console. If the kernel ring buffer logs don't appear after 5 seconds, consider swapping the RX and TX UART pins. After approximately 30 more seconds, the following login prompt should appear

```
Welcome to MIPSfpga++
mipsfpga-plusplus login:
```

Now, we can log in as `root` and start exploring our Linux environment on the softcore MIPS processor.

## 4.4 Running a program on Linux on MIPS on FPGA

In this section, we'll be going through the steps to execute a simple "Hello World" program on our MIPS softcore CPU.

One of the most challenging aspects of this task is transferring our compiled binary into the board's filesystem. The reason for this challenge is due to the fact that our only interface with the system is through a serial connection.

Let's begin by creating a C source file named `helloworld.c`:

```
#include <stdio.h>

int main(void){
    puts("Hello world\n");
    return 0;
}
```

Once you've created the source file, compile it using the following command:

```
docker run -it -v $(pwd):/workdir \
    -w /workdir/ \
    mmskv/mipsel-linux-gnu-gcc:latest \
    mips64el-linux-gnu-gcc -LE -march=24kc -mabi=32 helloworld.c -o helloworld
```

Upon successful compilation, you'll obtain the binary file 'helloworld'. To transfer this binary to our system, we'll need to encode it into a format that can be safely transferred over a serial connection. Here's where the command `cat helloworld | xz | base64` comes into play.

This command performs the following operations:

1. '`cat helloworld`': This reads the binary file and sends its contents to the standard output.
2. '`| xz`': The pipe symbol (`|`) redirects the output from the preceding command into the `xz` command, which compresses the binary data. This reduces the size of the data that needs to be transferred because it has many empty regions.
3. '`| base64`': This again redirects the output – this time, the compressed data – into the `base64` command. Base64 encoding is used to convert binary data into a text format, making it safe for transfer over protocols that are designed to handle text, in our case the serial tty.

The output of this command is a base64-encoded string representing the compressed binary. This can be selected and pasted into the Linux serial console.

However, a direct paste operation with heredocs might not be successful, because the CPU's processing speed might not keep up with the paste speed (noticeable from `ttyS ttyS0: 87 input overrun(s)` messages in the kernel logs). To circumvent this, a paste delay needs to be added in Minicom.

```
cat helloworld | xz | base64
```

Select resulting base64 to then paste it in Linux serial console

Simply pasting the stuff with heredocs won't work, because the cpu is so slow it can't read characters at the paste speed. (look for '`ttyS ttyS0: 87 input overrun(s)`' messages in kernel logs).

Start `minicom` terminal

```
minicom -b 115200 -D /dev/ttyUSB0
```

Add a paste delay with the following key presses

```
Ctrl-A, Z, O, Serial Port Setup, F, <Esc>, <Esc>
Ctrl-A, Z, T, F, <Basckspace>, 1, <Esc>, <Esc>
```

Next, we'll decode and decompress the binary, set the correct permissions, and finally run it

```
cat <<EOF | base64 -d | unxz > helloworld
<Ctrl-Shift-V to paste your base64 encoded binary>
EOF

chmod +x helloworld
./helloworld
```


Figure 6: Sending a helloworld binary and running it.

For those unfamiliar, the `<<EOF` syntax is known as a "here document" or "heredoc". It's a way of providing a block of text (possibly including shell script) as input to a command. This is typically used in shell scripting for multi-line input to a command. More information on heredocs can be found in the Wikipedia.

# 5 Discussion and future work

The current Linux configuration implemented in this project, while functional, is not necessarily optimal. There is room for further refinement to enhance the performance and security of the system. The .config file contains many options that could potentially be disabled to create a lighter, faster, and more secure Linux image.

During the project, it was observed that the board occasionally encountered memory errors through the JTAG interface. This issue resulted in system instability, causing the board to fail unexpectedly. The current workaround for this issue is to reflash the board using Quartus.

Another area that needs attention is the General Purpose Input/Output (GPIO) system. In this project, the GPIO functionality has not been extensively tested.

# References


[1] Stanislav Zhelnio, *MIPSfpga and SDRAM*, Habr, 2017,
`https://habr.com/en/articles/321530/`,

[2] Stanislav Zhelnio, *Running Linux on MIPSfpga and Altera FPGA*, Habr, 2017,
`https://habr.com/en/articles/333920/`,