Performant Rust Webserver on a Low-Cost FPGA

4. April 2025

To enhance user experience with our new AIONYX Hive product line, we've created a web-based configuration tool to configure the device. This requires a web server to run somewhere, preferably on (or beside) the FPGA. While our Hive-M, with its integrated quad-core CPU, easily accommodates this, the Hive-S presents a challenge. Built solely on an Artix™ 7 FPGA, the Hive-S lacks a dedicated CPU. To enable our web server on the Hive-S, we must implement a complete system, including a Linux-capable CPU (ideally with an MMU), a supporting System-on-Chip (SoC) infrastructure and the necessary software development toolchains for Linux and Rust, as our web server is built with Rust.

RISC-V Soft-Core CPU

To run a CPU on the Artix™ 7 FPGA, we first evaluated the MicroBlaze™ soft processor (RISC Harvard architecture). However, its limited ecosystem led us to LiteX. LiteX is a framework for building efficient RISC-V based SoCs. A functional Linux-capable SoC, including a configurable RISC-V 32-bit SMP CPU, is already provided here by the LiteX project.

Additionally, LiteX leverages Buildroot to easily create a Linux distribution for embedded environments. To accommodate our in-house IP cores and SoC modifications, we adapted the pre-existing generation process to import external RTL.

We utilize the GNU RISC-V toolchain for compiling Linux and applications that run on top (e.g., our web server).

Rust

While we now have a Linux system capable of running applications, deploying our Rust-based web server requires further steps.

Our web server is built with Rust because it is a compiled language that is inherently memory safe. Our first draft of the web server was written in Python, but since the performance on the Hive-S was poor, we have decided to switch to Rust.

To build the application for the Hive-S we need a Rust toolchain tailored to our target platform, namely *riscv32imac-unknown-linux-gnu*. Unfortunately, this target is not yet officially supported by Rust. Therefore, we must build a custom toolchain. Our objective is to execute *cargo* within our project's root directory and generate a binary executable on the target system.

For those interested in the process of adding a custom target, the following details the build procedure. Alternatively, a pre-built Rust toolchain is available here.

The following steps have been tested on an x86-64 Ubuntu 24.04 system, but may also work on other Debian based distros and architectures. To start off, you likely need to install some dependencies:

```
sudo apt-get update
sudo apt-get install build-essential pkgconf git curl cmake ninja-
build libssl-dev
```

To begin building our custom Rust toolchain, we first clone the official Rust repository and checkout the latest stable release (version *1.86.0* in our case):

```
git clone --recursive https://github.com/rust-lang/rust.git --branch
1.86.0 --depth 1
cd rust
```

Next, we define our custom target. This involves creating a new target specification within the Rust compiler's source code. Specifically, we add a file with the following content to the *targets* directory:

compiler/rustc_target/src/spec/targets/riscv32imac_unknown_linux_gnu.rs

```
use std::borrow::Cow;
use crate::spec::{CodeModel, SplitDebuginfo, Target, TargetOptions, base};

pub(crate) fn target() -> Target {
    Target {
        data_layout: "e-m:e-p:32:32-i64:64-n32-S128".into(),
        llvm_target: "riscv32-unknown-linux-gnu".into(),
        metadata: crate::spec::TargetMetadata {
            description: Some("RISC-V Linux GNU (RV32IMAC ISA)".into()),
            tier: Some(3),
            host_tools: Some(false),
            std: Some(false),
        },
        pointer_width: 32,
        arch: "riscv32".into(),

        options: TargetOptions {
            code_model: Some(CodeModel::Medium),
            cpu: "generic-rv32".into(),
            max_atomic_width: Some(32),
            features: "+m,+a,+c".into(),
            llvm_abiname: "ilp32".into(),
            supported_split_debuginfo: Cow::Borrowed(&[SplitDebuginfo::Off]),
            ..base::linux_gnu::opts()
        },
    }
}
```

Additionally, the new target specification must be registered in *compiler/rustc_target/src/spec/mod.rs*:

```
supported_targets! {
    ...
    ("riscv32imac-unknown-linux-gnu", riscv32imac_un-
known_linux_gnu),
    ...
}
```

Finally, the custom target must be added to the sanity check configuration in *src/bootstrap/src/core/sanity.rs*:

```
const STAGE0_MISSING_TARGETS: &[&str] = &[
    ...
    "riscv32imac-unknown-linux-gnu",
    ...
];
```

This step is necessary because the Rust toolchain is built incrementally, leveraging the latest stable official compiler. As our custom target is not recognized by the upstream compiler, we must explicitly include it here to prevent build errors.

With these modifications, we can now proceed to build the Rust toolchain with support for our custom target.

Important: Ensure you have a compatible GCC toolchain installed, as it is a prerequisite for the following build process. As mentioned before, we are using the GNU RISC-V toolchain. If you use x68-64 Ubuntu 24.04 as your host system, you can find a precompiled RISC-V toolchain including GCC 14.2.0 here.

First, define additional compiler flags for the toolchain:

```
export CFLAGS_riscv32imac_unknown_linux_gnu="-march=rv32imac -
mabi=ilp32"
```

This variable defines the architecture (*rv32imac*) and application-binary interface (ABI; *ilp32*) used when building Rust, which both should also be supported by the GNU toolchain that you are using.

Next, we configure the build process and specify the components to build by creating a *config.toml* file in the repository's root directory with this content:

```
profile = "dist"
change-id = 134650

[build]
target = ["riscv32imac-unknown-linux-gnu"]

[rust]
channel = "stable"
description = "Rust for NTL Hive-S SoC"

[target.riscv32imac-unknown-linux-gnu]
cc = "riscv32-unknown-linux-gnu-gcc"
ar = "riscv32-unknown-linux-gnu-ar"
```

The profile *dist* declares that we want to export and distribute the toolchain and under *[build]*, we define which additional targets we intend to build Rust for.

We can build the toolchain by simply invoking the following in the root directory of the Rust repository:

```
./x build -i --stage 1 compiler/rustc library/std
```

The build process will take some time. Eventually, when the build has succeeded, we can export the toolchain by invoking:

```
./x dist -i --stage 1
```

After this step, you should find multiple *.tar.gz files in the *build/dist/* directory relative to the root of the repository:

- *rust-1.86.0-x86_64-unknown-linux-gnu.tar.xz* is the host toolchain

- *rust-std-1.86.0-riscv32imac-unknown-linux-gnu.tar.xz* is the target toolchain with *std* support built-in
-

There are various ways to install the toolchain. For simplicity, we assume you are using *rustup* to manage your Rust toolchains. This process will then allow you to use the custom toolchain by invoking the known commands but adding *+custom* to it.

Note: We assume that *.cargo* and *.rustup* are in the *$HOME* directory.

Go to the directory you specified above. First, we need to unpack the two *.tar.gz* files:

```
tar -xvf rust-1.86.0-x86_64-unknown-linux-gnu.tar.xz
tar -xvf rust-std-1.86.0-riscv32imac-unknown-linux-gnu.tar.xz
```

Next, we can install it:

```
./rust-1.86.0-x86_64-unknown-linux-gnu/install.sh --pre-
fix=$HOME/.rustup/toolchains/custom
./rust-std-1.86.0-riscv32imac-unknown-linux-gnu/install.sh --pre-
fix=$HOME/.rustup/toolchains/custom
```

Now you should be able to invoke the custom toolchain. Let’s try it out by checking the version:

```
cargo +custom --version
# cargo 1.86.0 (adf9b6ad1 2025-02-28)
```

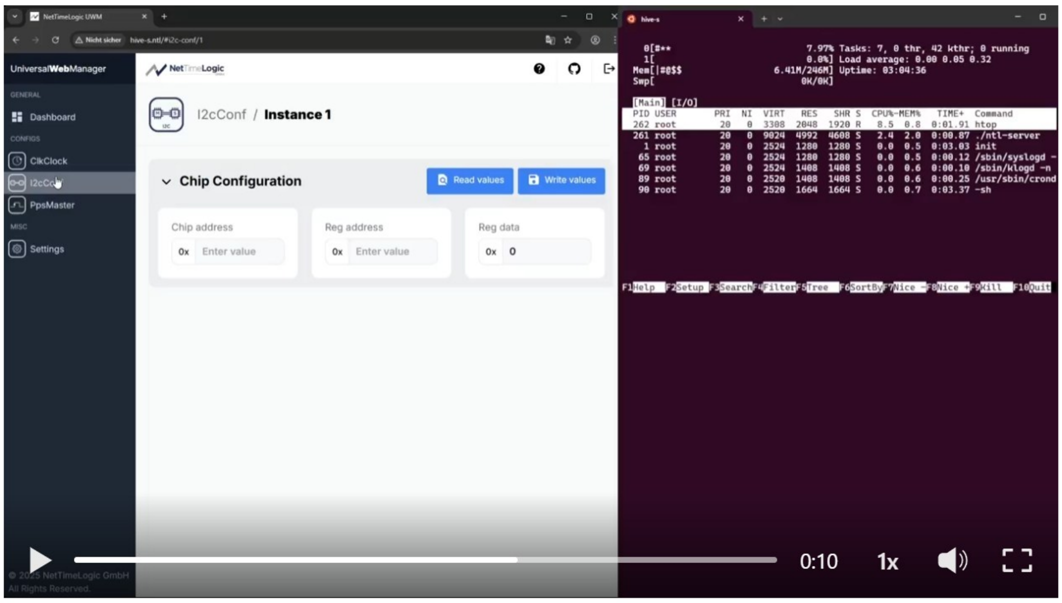
Let’s see if our custom target is supported:

```
rustc +custom --print target-list | grep "riscv32imac-unknown-linux-
gnu"
# riscv32imac-unknown-linux-gnu
```

Perfect. Now we can start building applications for the Hive-S!

Demo and Performance

The following video provides an insight of how our web server performs on the Hive-S.



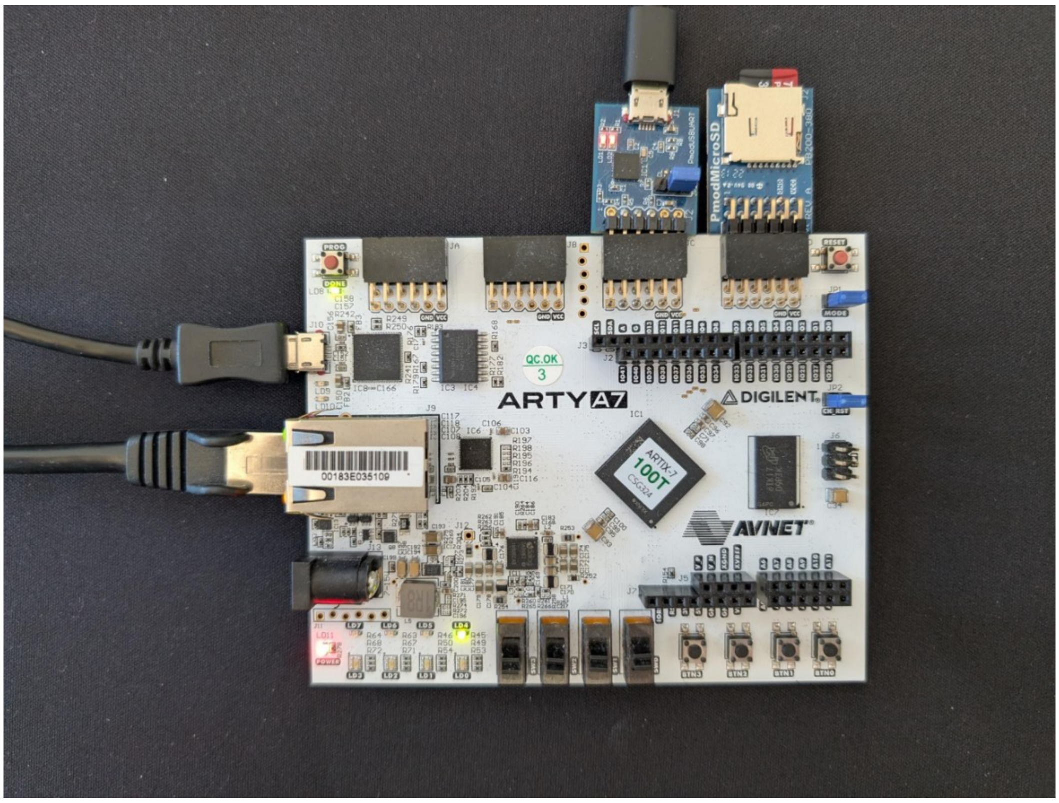
Chip address: 0x Enter value
Reg address: 0x Enter value
Reg data: 0x 0

```
    0[#**                         7.97% Tasks: 7, 0 thr, 42 kthr; 0 running
    1[                            0.0%] Load average: 0.00 0.05 0.32
  Mem[|#@$$                 6.41M/246M] Uptime: 03:04:36
  Swp[                           0K/0K]

  [Main] [I/O]
  PID USER       PRI  NI  VIRT   RES   SHR S  CPU%-MEM%   TIME+  Command
  262 root        20   0  3308  2048  1920 R   8.5  0.8  0:01.91 htop
  261 root        20   0  9024  4992  4608 S   2.4  2.0  0:00.87 ./ntl-server
    1 root        20   0  2524  1280  1280 S   0.0  0.5  0:03.03 init
   65 root        20   0  2524  1280  1280 S   0.0  0.5  0:00.12 /sbin/syslogd -
   69 root        20   0  2524  1408  1408 S   0.0  0.6  0:00.10 /sbin/klogd -n
   89 root        20   0  2520  1408  1408 S   0.0  0.6  0:00.25 /usr/sbin/crond
   90 root        20   0  2520  1664  1664 S   0.0  0.7  0:03.37 -sh

F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

The video shows the performance of the Hive-S when accessing and browsing our UniversalWebManger



The setup for testing the implementation on a Digilent ARTY A7 100T

Conclusion

Finally, we have everything together to run our web server on the Hive-S in an Artix™ 7 FPGA. Even though the CPU is running at only 125 MHz, the web server is very responsive and capable of processing some more demanding requests. In the end, we can say that this is a fitting solution for our platform and the simplest in the end-run.
In the future, we may change our SoC to use the new MicroBlaze™ V soft processor which is based on the RISC-V architecture as well, but makes it even easier to integrate with the tools we use. Stay tuned for more information around our AIONYX ecosystem.

Link Summary

- Precompiled Rust toolchain with support for RISC-V 32-bit (rv32imac): https://github.com/NetTimeLogic-OpenSource/rust/releases/tag/1.86.0-ntl
- Precompiled GNU RISC-V toolchain (x86-64 Ubuntu 24.04 only): https://github.com/NetTimeLogic-OpenSource/riscv-gnu-toolchain/releases/tag/2025.01.2
- Rust: https://www.rust-lang.org/
- Rust source code: https://github.com/rust-lang/rust
- GNU RISC-V toolchain source code: https://github.com/riscv-collab/riscv-gnu-toolchain
- LiteX: https://github.com/enjoy-digital/litex
- LiteX SoC with RISC-V CPU: https://github.com/litex-hub/linux-on-litex-vexriscv
- Buildroot: https://buildroot.org/