AmaranthHDL in RadioHound: Optimizing Module Development Workflows Utilizing a Pythonic Hardware Description Language

Abstract

The configurable integrated circuits known as “field-programmable gate arrays” (FPGAs), despite offering some of the most remarkable modularity, reliability, and processing power in hardware development, also pose a high barrier to entry for implementing and integrating them in real-world projects. Hardware description languages (HDL), which, for the purposes of this paper, will be grouped and referred to collectively, present one such barrier to project development. Operating at the Register-Transfer Level (RTL) [1], HDL is inherently unintuitive and is used to describe circuit behavior within an FPGA. Indeed, any HDL programming is a strenuous endeavor, with the implementation of a 4-to-1 multiplexer requiring 10 distinct lines of code in Verilog (1 of 2 industry-standard HDLs) as well as involving a largely convoluted naming convention and function scheme [2]. In comparison, a pure C++ implementation requires just 1 line to configure the multiplexer and return a result, with data streaming and its associated libraries requiring only a few more lines to yield a satisfactory program [3].

Forgoing the complex nature of FPGA-based systems as it is, it suffices to say that HDL is in and of itself a strenuous endeavor. Many who come from prior coding experience find that the nature of these languages is hardly conducive to effective or efficient programming, with debugging, in particular, being a tall task even for a seasoned developer. However, a recent project aims to heavily reduce this barrier to entry: Amaranth HDL [4]. First released in 2021, AmaranthHDL is a modern successor to the Python library Migen [5], now operating as a standalone, Python-based HDL. Self-described as “an open-source toolchain for developing hardware based on synchronous digital logic using the Python programming language”, Amaranth reduces the strain of coding HDL and drastically simplifies the development process. While there may be minimal support for the language at this point, compared to the likes of

Verilog, the potential for an enormous reduction in development time could completely change the process of programming FPGA systems within the RadioHound project.

Moreover, the increasing ubiquity of Python has resulted in large swaths of college, high school, and even elementary-aged students having a firm command of the language [6]. Ultimately, Python's status as "the" general-purpose language means that more young developers than ever will be well-versed in its minutiae and subsequently be more equipped to tackle a variety of projects. While Python is certainly not revered for its speed [7], the toolchain structure outlined in this paper is not structurally inhibited by Python's processing speed.

Ultimately, this paper aims to outline the entire AmaranthHDL-to-Verilog development workflow, beginning with toolchain setup and continuing through to physical integration with the ADALM2000 M2K architecture [8] within AMD's Vivado Design Suite [9]. The research this paper focuses on is largely centered around the viability of the path outlined in Fig. 1, wherein a developer is able to write synthesizable Verilog via AmaranthHDL on their own machine, and then seamlessly integrate these files within a Vivado project. In short, this paper presents a proven path for HDL module development in the RadioHound ecosystem via an abstraction library in Python, and charts effective methods for optimizing said development.

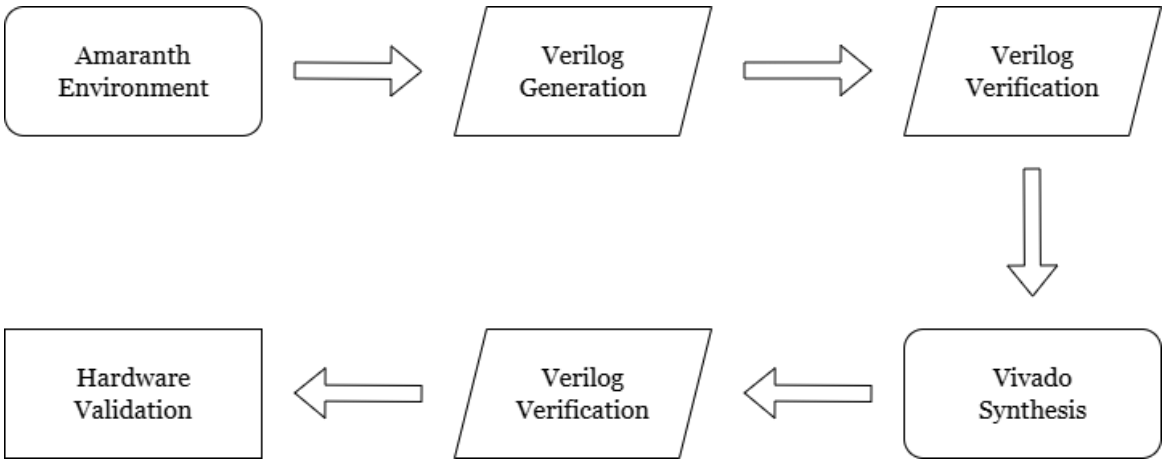


Fig. 3. AmaranthHDL to ADALM2000 M2K development pipeline

Toolchain

The inchoate stages of the development process begin with the setup of an AmaranthHDL-conducive development environment.[1] The AmaranthHDL library can be installed simply by running the `pip install amaranth` command. The additional command

[1] For the purposes of this paper, referenced commands, library installations, and any other build-specific parameters will be assumed to be tailored to a macOS (version 14+) machine.

`pip install yowasp-yosys` installs YoWASP (Yosys WebAssembly Synthesis & PnR) [10], a specialized package distributor that includes WASM-based, precompiled FPGA development tools; for this project, it will specifically include Yosys. These two installs comprise everything necessary for strictly AmaranthHDL development. However, it is generally recommended to run these commands within a pre-apportioned Python virtual environment.[2]

A useful, albeit not strictly necessary, tool is the VS Code extension Surfer, which this project uses to analyze value change dump (VCD) files. These ASCII-based waveform files provide indicators of signal change in an executed system. In the wake of the deprecation of GTKWave [11], an industry-standard waveform viewer, following the release of macOS 14 Sonoma, Surfer is a particularly lightweight and intuitive alternative to such legacy applications. Given its WASM package implementation, Surfer can ultimately be used in a variety of integrated development environments (IDEs) and code editors, enabling cross-platform viewing of these VCD files.

Finally, to verify the synthesizability of the Verilog code generated during the AmaranthHDL development process, we use Icarus Verilog (iverilog) [12], an open-source compiler for the language. After installation, iverilog enables us to confirm that a Verilog file generated from an AmaranthHDL script is, in fact, valid. Of course, whether the code works in practice is an entirely separate issue, but the iverilog status check ensures that any problems encountered are guaranteed to have arisen independently of the Python script. Ultimately, these four tools, combined, enable a developer to implement a lab-independent development workflow for initial RTL synthesis, with only nominal technical overhead and markedly increased efficiency.

Development

Following the successful setup of the toolchain outlined in the previous section, this research focused on developing the so-called “blinky” project for the ADALM2000 M2K. The hardware equivalent of the universal “Hello World” program, “blinky” aims to test the most fundamental signal variation capabilities of the GPIO pins on a development board [13], in this case, the ADALM2000 M2K, which will hereafter be referred to as the M2K.

The first 14 lines of *blinky.py*[3] constitute the header of the file, defining the basic frequency constraints that both the VCD simulation and hardware testbenching will be built around, that is, the 40MHz real-world clock of the M2K [14], the blink rate frequency of

[2] https://github.com/mackfound/amaranthHDL-blinky/blob/main/examples/README.md
[3] https://github.com/mackfound/amaranthHDL-blinky/blob/main/blinky.py

1e-6MHz, and the 1e-4MHz simulator clock frequency. Upon initializing the overarching `BlinkLED` class, the script notates the signals to be exported when the VCD file is generated later in the code. The actual functionality of the script appears in lines 32-44, where the script defines its self-referential counter, which ultimately drives the LED signal change. Ultimately, for all intents and purposes, signal changes occur on the rising edge of a clock cycle [15], meaning that once the counter reaches its upper threshold, the LED signal switches its signal from low to high or vice versa upon encountering the next rising edge of the system clock. Finally, lines 47-66 of *blinky.py* elaborate on the functions necessary to write the Verilog and VCD files, as well as to run the actual simulation depicted by the VCD file, with the `main()` function calling these final aspects of the program.

Upon execution of *blinky.py*, two new files are generated in the same working directory: *blinky.v* and *blinky.vcd*. When beginning the analysis stage of the AmaranthHDL development process, the VCD file becomes particularly useful for visualizing and understanding the fundamental timing of the circuits being elaborated in the Verilog file. Within Surfer in Fig. 1, four signals are displayed: `clk`, `counter`, `led`, and `rst`. `Clk` and `counter` are predefined signals that drive the `led` signal changes in the system. `Rst` is interpreted as "reset", and as such, remains at 0 for the entirety of a continuously operating system.

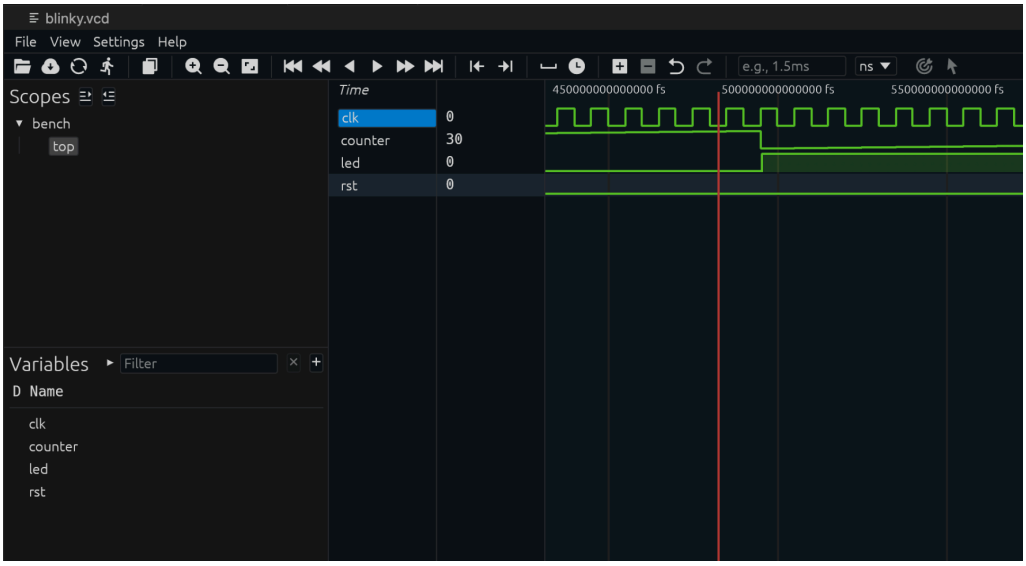


Fig. 2. Surfer visualization of apportioned waveforms from *blinky.vcd*

Being that the purposes of this paper are to illustrate the utility of a development process independent of direct Verilog analysis, the only considerations afforded to the *blinky.v* file are performed with iverilog. A single terminal command, `iverilog -t null blinky.v`, is the

sole immediate check needed to verify the synthesizability of the Verilog code in *blinky.v*, with an empty return indicating a successful execution.

All the research methods outlined in the prior sections have now enabled the development of a synthesizable Verilog file that can be instantiated within a prototypical Vivado project hierarchy.

Integration

Following a successful development of *blinky.v* via the AmaranthHDL workflow, it becomes a relatively nominal process to integrate and validate the generated module within a Vivado project's top Verilog file. The construction of the Vivado project for this paper was based on the tool command language (TCL) build script `build_pluto_win.tcl`, which, once included in the project GitHub repository, sources a variety of IP cores for simulation.

Prior to synthesis, however, several files need to be edited to ensure the blinky module was instantiated correctly within the project filesystem and execution hierarchy. Firstly, lines 12-15 of `system_project.tcl` include the following addition: `"blinky.v"\`. This line simply includes our project's built module in the filesystem reference, which `build_pluto_win.tcl` will inevitably reference. Most importantly, however, `system_top.v`[4] requires the addition of 3 more references to the blinky module, beginning with the lines `wire blinky_led` and `assign pl_gpio0 = blinky_led`, which in essence are simply the direct references to how the blinky module connects to the General Purpose Input/Output (GPIO) pins on the M2K. Finally, the module itself is fully instantiated within the file hierarchy via the following:

```
blinky i_blinky (
      .clk (rx_clk_in),
      .rst (1'b0),
      .led (blinky_led)
);
```

Through these 3 additions to preexisting files within the GitHub repository, the AmaranthHDL-generated blinky module has a defined place within the project, and upon Vivado implementation, is fully routed to physical pins on the M2K. Prior to disseminating the final results of the physical implementation, however, a caveat must be made known, that is, despite being called "blinky" due to the prototypical module defining a blinking LED on a

[4] https://github.com/mackfound/hdl/blob/main/projects/pluto/system_top.v

development board, the signal changes tied to the instantiated `blinky.v` module in this project was not intrinsically connected to an LED output on the M2K. Rather, signal changes were observed and quantified via an oscilloscope, as outlined further in this paper.

Upon connecting a `GND` pin and the `L10P` pin [16] to the `blinky.v` module was routed via an in-lab oscilloscope[5], and we are able to visualize the actual signal changes occurring on the M2K. As seen in Fig. 3, with Time per Division set to 1s, these signal changes are consistent with a square waveform and follow the cycle defined in the AmaranthHDL testbench code.

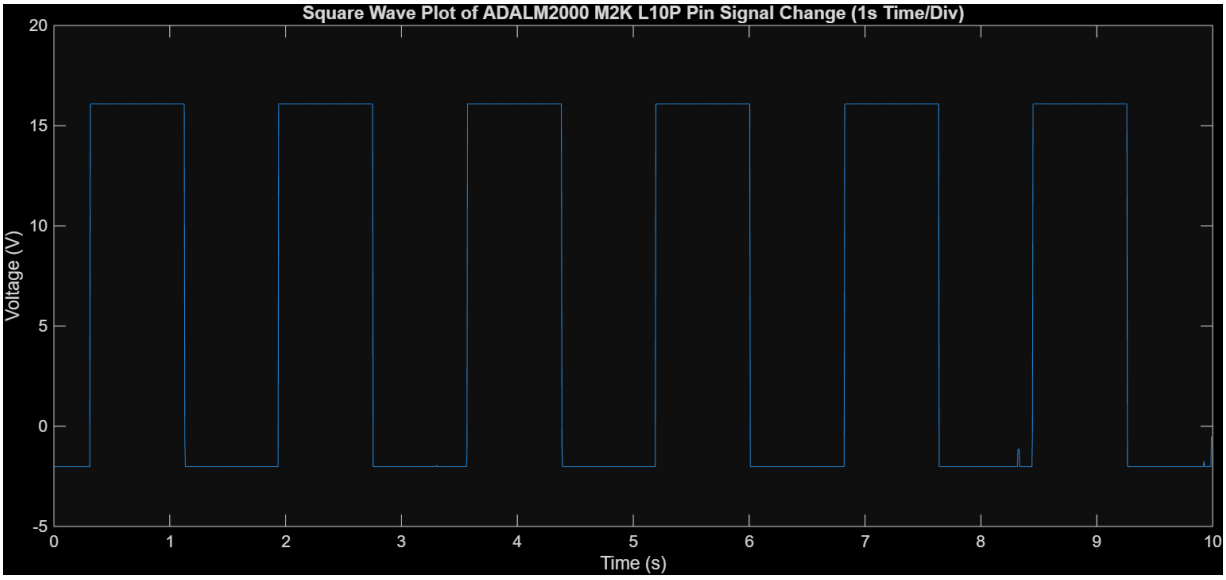


Fig. 3. MATLAB square wave plot of the M2K's L10P signal changes following successful implementation

The final step, as any research process would demand, is an attempt at graphical analysis and computational reasoning on a certain acquired dataset. The benefit of waveform data is that it is generally easy to export a CSV file and write the necessary MATLAB or Python script to reconstruct the peaks and valleys of the graph. Moreover, this analysis is necessary and beneficial when either validating or debugging an AmaranthHDL-based module, as the comparison between the VCD simulation results in Surfer in the incipient stages of development with the real-world data collected on the oscilloscope can reveal any gaps arising in the integration process, as well as verify the authenticity of the module instantiation.

[5] https://www.keysight.com/us/en/assets/7018-04570/data-sheets/5992-0140.pdf

Conclusion

In brief, this research has charted novel and effective paths for HDL module implementation, specifically within the RadioHound FPGA development ecosystem. More importantly, however, this paper outlines a proven path for researchers of any coding background to get a start on hardware programming by reducing a seemingly colossal barrier to entry into a manageable workflow. As it stands, module development using the AmaranthHDL library enables burgeoning research assistants (undergraduates and graduate students alike) to remain within the realm of high-level programming in Python while simultaneously working on low-level projects. Moreover, the nature of this AmaranthHDL-to-Verilog workflow is that the Python scripts act as the source of truth in the implementation process, meaning that a developer has not only full control over their testbenches, but also does not need to ever examine a Verilog file directly, since their aforementioned testbench is defined within the AmaranthHDL script.

Regarding a longer-term outlook, there is, in theory, no practical limit to what can be implemented within the AmaranthHDL library. Putting aside the extensive documentation and community that continues to expand each year, the intuitive nature of Python enables a potent application of the concept of top-down design, as illustrated in Fig. 4. Ultimately, in this model, which will always be more conducive to higher-level languages, tasks are maximally broken down until a developer is left with independent lines of code. In essence, this is the ultimate goal of AmaranthHDL, and by extension, any high-level abstraction of a hardware description language.

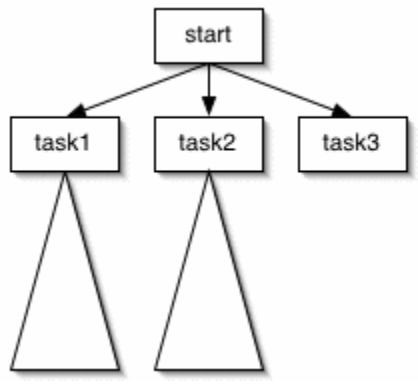


Fig. 4. Illustration of generic top-down programming design.

MaiaSDR, an open-source signal processing project built in AmaranthHDL for the ADALM-PLUTO[6] SDR (Software-Defined Radio), is an example of the power of this workflow optimization and was one of the inspirations for the research methods in this project. Given the success and replicability of MaiaSDR, it suffices to say, then, that the implementation of a high-speed data acquisition system, one aspect still in development in the RadioHound project, is not an insurmountable goal within AmaranthHDL. Ultimately, effective methods for building on this research in the future would ideally include developing asynchronous clock domains, proprietary IP cores, and full-stack SDR projects using the AmaranthHDL library.

References

[1]"Register-transfer level," Wikipedia, Mar. 05, 2021. https://en.wikipedia.org/wiki/Register-transfer_level
[2]mackfound, "amaranthHDL-blinky/examples/mux4to1.v at main · mackfound/amaranthHDL-blinky," GitHub, 2025. https://github.com/mackfound/amaranthHDL-blinky/blob/main/examples/mux4to1.v (accessed Apr. 30, 2026).
[3]"mackfound," Github.com, 2026. https://github.com/mackfound/amaranthHDL-blinky/blob/main/examples/mux4to1.cpp (accessed Apr. 30, 2026).
[4]AmaranthHDL, "Amaranth HDL documentation — Amaranth HDL toolchain 0.3 documentation," *Amaranth-lang.org*, 2020. https://amaranth-lang.org/docs/amaranth/v0.3/cover.html (accessed Apr. 30, 2026).
[5]Migen, "Migen Documentation," M-labs.hk, 2026. https://git.m-labs.hk/M-Labs/migen (accessed Apr. 30, 2026).
[6]J. Carter, "The Ubiquitous Influence of Python in the Modern Technical World," American Journal of Computer Science and Engineering Survey, vol. 12, no. 1, pp. 1–1, Mar. 2024, doi: https://doi.org/10.36846/2349-7238.24.12.10.
[7]J. VanderPlas, "Why Python is Slow: Looking Under the Hood | Pythonic Perambulations," Github.io, May 09, 2014. https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/
[8]"Model Description ADALM2000 Production." Accessed: Apr. 30, 2026. [Online]. Available: https://www.farnell.com/datasheets/2261938.pdf
[9]"AMD VivadoTM Design Suite," AMD, Aug. 23, 2024. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html
[10]"YoWASP," YoWASP, 2026. https://yowasp.org/ (accessed Apr. 30, 2026).
[11]"GTKWave," Sourceforge.net, 2025. https://gtkwave.sourceforge.net/
[12]S. Williams, "The ICARUS Verilog Compilation System," GitHub, Mar. 11, 2023. https://github.com/steveicarus/iverilog

[6] The ADALM2000 M2K and ADALM-PLUTO share very similar hardware, that is, the Zynq-7000 SoC [17]

[13]“Blinky,” Zipcpu.com, May 19, 2017. https://zipcpu.com/blog/2017/05/19/blinky.html (accessed Apr. 30, 2026).
[14]“M2K HDL Architecture [Analog Devices Wiki],” Analog.com, Oct. 27, 2025. https://wiki.analog.com/resources/fpga/docs/hdl/m2k (accessed Apr. 30, 2026).
[15]Wikipedia Contributors, “Signal edge,” Wikipedia, Mar. 19, 2025. https://en.wikipedia.org/wiki/Signal_edge
[16]Analog Wiki, “PLUTO SDR Schematic Rev. D https://wiki.analog.com/_media/university/tools/pluto/hacking/plutosdr_schematic_revd_0.1.pdf
[16]“AMD ZynqTM 7000 SoCs,” AMD, Aug. 23, 2024. https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html