NEW YORK UNIVERSITY

TANDON SCHOOL OF ENGINEERING

Vito Gamberini

Design and Construction of RTL Toolchains at NYU

A report on the toolchain work performed by the NYU Processor Design Team

Senior Design Project

May 20, 2024

Supervisor Dr. Ratan Dey Industry Associate Professor

Thanks to:

Prof. Jeff Epstein, for supporting the orginal ProcDesign team effort

Dr. Mark Johnson, for his patience, advice, and general expertise in all things RTL

All my homies in the JCU

Vito Gamberini, *Design and Construction of RTL Toolchains at NYU,* A report on the toolchain work performed by the NYU Processor Design Team, May 20, 2024.

Abstract

Register Transfer Level (RTL) design has classically been the domain of commercial tools orchestrated via ad hoc scripts and firm-specific tooling. Decades of work in the software world on toolchain orchestration, combined with rising interest in the open hardware movement, have made it possible to construct standardized workflows entirely free of commercial black boxes. This report details the RTL verification system used by the New York University *Processor Design Team*, which has been developed by the team since September 2023.

Contents

Abstract 2

1 Background 4

1.1 History of NYU ProcDesign 4
1.2 Traditional RTL Toolchain Design 5
1.3 Team Requirements 6
1.4 Prior Art 6

2 The NYU System 8

2.1 Overview 8
2.2 Verilator 9
2.3 CMake 10
2.4 vcpkg 12
2.5 Catch2 14
2.6 ProcDesign Utilities 15

3 Beyond the Toolchain 17

3.1 SCM and Documentation 17
3.2 Continuous Integration 17

4 Future Work 19

APPENDIX

A Documentation Example: Edge Synchronizer 21

1 *Background*

Register Transer Level (RTL) design[1] refers to the phase of the hardware implementation process where designers describe the flow of data between registers and the logical operations performed on that data. The RTL design phase bridges the gap between high-level architectural design specification, and low-level gate synthesis and layout.

[1]Bening, L. & Foster, H. *Principles of Verifiable RTL Design* ISBN: 0-7923-7788-5 (Jan. 2002).

Hand-in-hand with RTL design is verification, the process of ensuring that an RTL design meets the requirements of the specification. Verification has many phases, spanning software simulation all the way into post-fabrication of microelectronic designs. At the RTL stage, verification is focused on the logical simulation of individual components as well as integrated devices.

A strong understanding of RTL design, HDLs, or hardware generally is not essential to understanding this paper. Suffice to say that there is a style of source code that describes the operation of a piece of hardware, and that testing this source code via simulation prior to the expensive process of fabricating physical devices is a high priority for hardware designers.

The construction of these simulations, starting from the hardware description language (HDL) source code of the RTL design, integrating test drivers and a simulation engine, and finally producing outputs such as waveforms or test reports, is the job of several disparate software tools. The integration of these tools into a comprehensive, user-facing interface is the responsibility of the *toolchain.*

1.1 *History of NYU ProcDesign*

The current effort at New York University has its roots in the System-On-Chip Extension Technologies (SoCET) team at Purdue University.[*] SoCET's goals are pedagogical, the team achieves ABET outcomes for students while also providing an interface to modern hardware industry tools and practices.[2]

[*]The brainchild of Dr. Mark Johnson, SoCET has been producing high-quality microelectronic engineers straight out of undergrad for over a decade.

[2]Swabey, M. A. & Johnson, M. C. *Satisfying ABET criterion using an industrial Microelectronic Skills Incubator* in *2015 IEEE International Conference on Microelectronics Systems Education (MSE)* (2015), 28–31.

The NYU Process Design Team (ProcDesign) was envisioned as an export of the system pioneered at Purdue.[†] However, differences in resources and experience levels naturally led to a divergence in philosophy. and NYU ProcDesign has instead developed a heterodox approach to RTL design. Where the requirements of ABET and needs of industry have driven the SoCET process, ProcDesign has organized around a different set of principles:

- Using highly automated, portable, design and verification workflows
- Applying modern software integration techniques to RTL design
- Development of skills useful to both hardware and software industries

[†]The team was originally proposed to NYU by the author following a two month research program on the SoCET team.

While the output product of both teams is similar[‡], the ProcDesign team's isolation from industry norms has produced several novel methodologies especially in the realms of verification, package management, and debugging of RTL design work.

[‡]SoCET has a long history of successful tapeouts with its AFTx chip series, ProcDesign is moving towards its first tapeout in late 2025.

Today ProcDesign has dedicated RTL codebases for a RISC-V core, various bus architectures, and several peripherals[§] all designed, verified, and packaged using the principles centered by the unique circumstances and goals of the ProcDesign team.

[§]ProcDesign GitHub Organization: `https://github.com/NYU-Processor-Design`

1.2 *Traditional RTL Toolchain Design*

Traditional RTL verification toolchains, especially in academia, are ad hoc constructions around a handful of commercial simulation and verification solutions.* Electronic Design Automation (EDA) vendors supply various simulator products with differing feature sets and interfaces, and it is up to designers to build toolchains compatible with these systems.

*As a practical matter, the industry is dominated by "the big three":
- Cadence's **NCSim**
- Siemens' **Questa**
- Synopsys' **VCS**

The common form of this tooling is frequently cobbled-together scripts written in shell, Make, Perl, or Python depending on the particular affinities of their author.

The following listing demonstrates a typical Makefile in this tradition:

```
ifeq ($(shell hostname),ecegrid-lnx.ecn.purdue.edu)
  COMPILE_V               := gridtest vlog
  SIMULATE                := gridtest vsim
else
  COMPILE_V               := vlog
  SIMULATE                := vsim
endif
# ...
sim_source:
  @rm -rf source_work
  @$(LIB_CREATE) source_work
  @$(COMPILE_V) -work source_work $(AHB_FILE) > source.comp
  # ...

  # Uncomment below if you want run the simulation the normal way
  # and have it run the specified .do file
  # @$(SIMULATE) -t ps -do s_waves.do source_work.$(TB_ENTITY)

  # This way just runs it like normal and only sets up the simulation
  # but doesn't run it or add any waveforms
  @$(SIMULATE) -i -t ps source_work.$(TB_ENTITY)
```

Listing 1. Makefile code from SoCET uart_debugger, circa 2013.

This example has been pared down from the original to include only the most relevant sections of code.

This code is in no way an *exhaustive* example, it is from a small project with no dependencies, and toolchain challenges scale with the complexity of the design being implemented.

This example demonstrates several of the weaknesses of the ad hoc approach, namely:

- A reliance on install-specific minutia, here demonstrated by the detection and hardcoding of configuration information for a specific Purdue build server
- Hardcoding of source and build directory locations into the toolchain
- Toolchain configuration via manually modifying the toolchain files, such as by commenting and uncommenting specific lines

1.3 *Team Requirements*

The ProcDesign team is primarily composed of second and third year undergraduate students,[*] typically coming out of their first formal course of instruction in computer architecture, with minimal exposure to typical software or hardware development workflows. This posed an interesting challenge and oppertunity for the ProcDesign technical leadership, as the team had little relevant experience, but also no attachment to traditional methodologies.

[*]Most students at this stage will be compiling and running code with some variation of:

```
g++ *.cpp && ./a.out
```

Several requirements for the team's toolchain were identified:

- Portability, able to run on any student's available hardware
- Featureful, supporting testing, waveform generation, package management and any other development needs met by traditional workflows
- Robustness, reasonably able to adapt to a given development environment, be that changes in compiler, tool paths, install locations, etc
- Simplicity (of use), students needed to be trained on how to use and expand on the toolchain in a matter of weeks

ProcDesign also lacked anything resembling comparable funding or industry support enjoyed by other efforts. This shaped requirements equally, if not more so, than the inexperience of its team members.

Notably, simplicity of *implementation* was not considered a priority.[†] ProcDesign then (as now) considered it acceptable that development and maintenance of the toolchain workflow could be relegate to a few experts, so long as all members were able to productively make use of it.

[†]This has been a point of some contention, as the techniques leveraged by the toolchain requires a broad familiarity with C++, CMake, SystemVerilog, and vcpkg. Having even a single teammember familiar with all the involved technologies is a sustainability challenge for the team.

Additionally, the requirements inadvertently supported a "rapid-iteration" style of development. As no access to a specialized build server would be required, new ideas and components could be built and tested with relatively low overhead by the ProcDesign team.

1.4 *Prior Art*

FuseSOC[3] represents the state-of-the-art in open source toolchain development for RTL design. It met some, but not all, of the requirements of the ProcDesign team. Briefly, FuseSOC represents a natural evolution of ad hoc Python scripts, itself derived from the homegrown build system developed for the OpenRISC Reference Platform.

[3]Kindgren, O. *A Scalable Approach to IP Management with FuseSoC* in *2019 Workshop on Open Source Design Automation (OSDA)* (2019).

FuseSOC aims to solve the *vendoring*[‡] problem. The nature of the OpenRISC project was such that it rapidly accumulated a large number of slightly different RTL components, with individual codebases never propagating changes and improvements to a given component to each other. FuseSOC solves this problem by allowing for the construction and distribution of catalogs of "cores", individual RTL components.

[‡]**Vendoring** is a term originated by the Ruby community ~2006 and popularized by the Go language documentation, it refers to the wholesale copying of dependency source code directly into a project. The term derives from the destination folder for this code, the "vendor" folder.

FuseSOC intentionally does not treat RTL components as merely HDL source code collections, it has a verbose format that describes the requirements for each component[§] when used with a particular EDA suite. Notably, each individually

[§]By "component" we are speaking to the more colloquial idea of a "top level", an HDL module which exists at the "top" of the hierarchy and determines the input and output signals to the overall system.

testable or integrable component must have its own FuseSOC metadata file describing the mechanisms supported by said component.

```
CAPI=2:
name : ::i2c:1.14
filesets:
  rtl_files:
    files:
      - rtl/i2c_byte_ctrl.v
      - rtl/i2c_def.v:
          is_include_file : true
      - rtl/i2c_top.v
    file_type : verilogSource
tb:
  depend:
    - ">=vlog_tb_utils-1.0"
  files:
    - tb/tst_bench_top.v:
      file_type : verilogSource

targets:
  default:
    filesets : [rtl_files]
  sim:
    default_tool : icarus
    filesets : [rtl_files, tb]
    toplevel : tst_bench_top
```

Listing 2. Partial FuseSOC core description file from [3]

This is for a simple i2c module, consisting of only three source files.

Notably, the `sim` target only has a single toplevel module, and does not present a straightforward abstraction for integrating non-RTL source files or workflows.

This granular approach allows FuseSOC to maintain support for fifteen different EDA suites, but makes it rather verbose and demanding for component maintainers. SoCET has experimented with FuseSOC but frequently uses build system shortcuts to avoid this verbosity, and ProcDesign has avoided FuseSOC entirely.*

*Additionally, FuseSOC is a niche toolchain in a niche industry. As ProcDesign aims to arm students with skills that will find wide applicability in either software or hardware endeavors, FuseSOC was a poor fit.

When exchanging RTL components between labs using different EDA suites, FuseSOCs wide support base is a major advantage. As ProcDesign is building and testing components on student machines with no access to commercial EDA suites, this advantage could not be leveraged and the overhead of teaching the FuseSOC system was viewed as unworkable.

2 *The NYU System*

Having judged existing systems unsuitable, the ProcDesign team built their system by way of adapting existing software toolchain approaches to the needs of a hardware workflow. The major components* used by ProcDesign are:

- CMake, for overall toolchain orchestration
- Verilator, as the RTL simulation engine
- vcpkg, for package management and distribution
- Catch2, as the test interface and for generation of reports and artifacts

*The criteria for component selection were popularity (CMake), availability (Verilator, free, and the only software in its niche), and the preferences of the author (vcpkg, Catch2).

2.1 *Overview*

The following illustrates the relationship of the major processes involved in the ProcDesign toolchain in its typical configuration.

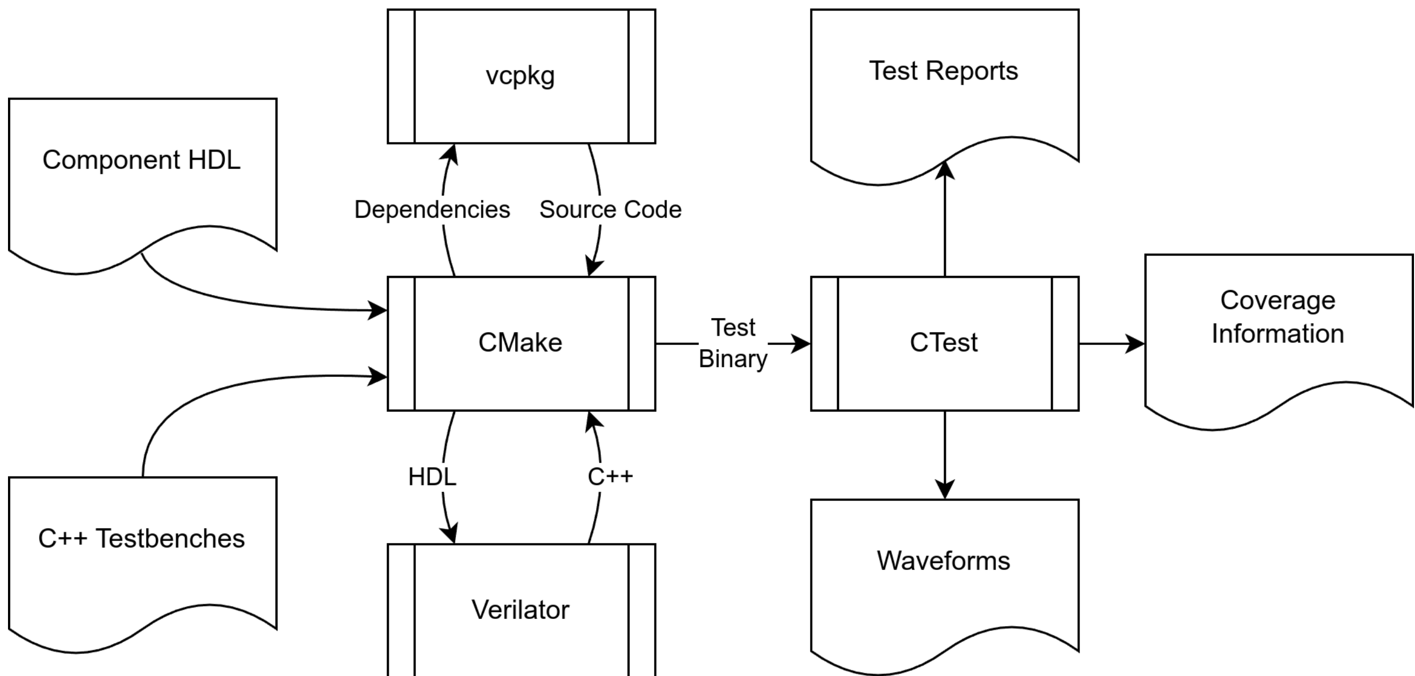

Figure 2.1.
An illustration of the major process components of the ProcDesign toolchain

A given ProcDesign repository is divided into an HDL source code folder, typically named `rtl`, and a mixed HDL and C++ test driver folder, typically named `dv` (**d**esign **v**erification). These are the primary inputs to the toolchain. Additionally, a `vcpkg.json` file describes dependencies needed from outside the current project (HDL, C++, or other code).

Those familiar with modern C++ development workflows will note that, absent the Component HDL, Verilator, and Waveforms blocks, this is a standard C++ development toolchain.

The toolchain processes these files appropriately to produce a final simulation program. It is possible to directly run this program and produce desired output

files, but this is usually delegated to CMake's dedicated test runner, known as *CTest*. Running the simulation binary will produce, at a minimum, a test report, and possibly waveforms or coverage information if requested.

2.2 *Verilator*

Verilator[4] is a package which produces C++ models of SystemVerilog* RTL designs. Depending on the nature of the model, these can either be compiled directly into a simulation program or linked into a larger C++-based test program.

[4] Snyder, W. *Verilator* version 5.024. Veripool, Apr. 5, 2024.

* At this point it is impossible not to mention that there are two major HDLs in use for RTL design. These are SystemVerilog and VHDL. ProcDesign uses SystemVerilog and some other minor HDLs, but never VHDL.

In traditional RTL verification workflows, the verification tests are themselves written in an HDL dialect. This makes some sense, as simulators understand HDL syntax and hardware engineers are competent in authoring HDL code. These test components are called "testbenches", and when using such a construct Verilator can produce a simulation program of the testbench directly.

However, the domain of HDLs is declaratively described physical hardware. Pure HDLs are closer to markup languages than programming languages, and lack many of the advanced concepts that make programming languages so convenient for describing *programmatic* behavior. For this reason, it can be preferable to write verification tests in a dedicated programming language instead of an HDL.

Supporting this, Verilator can take an RTL component design† and create a C++ model of just the component, without any surrounding testbench or other simulation code. This C++ model can then be manipulated by normal C++ code as part of a test program, as demonstrated here:

† In the context of testing this component is typically referred to as the **D**evice **U**nder **T**est, or DUT, and this convention is used throughout ProcDesign tooling and documentation.

```
module AluDevice (
  input  [1:0] op,
  input  [7:0] a,
  input  [7:0] b,
  output [7:0] out
);

assign out = op[1] ? (op[0] ? a | b : a & b) : (op[0] ? a - b : a + b);

endmodule
```

Listing 3. A simple ALU-like device

This example is from the second of the on-boarding labs performed by new ProcDesign teammembers.

```
int main() {
  VAluDevice DUT;
  for (int op = 0; op  < 4; DUT.op = ++op) {
    for (int a = 0; a < 256; DUT.a = ++a) {
      for (int b = 0; b < 256; DUT.b = ++b) {
        DUT.eval();
```

Listing 4. A C++ testbench for the `AluDevice`

The `V` prefix for the `AluDevice` typename is a default Verilator convention, but the type can be configured to be named anything.

```
        switch(DUT.op) {
          case 0:
            if((DUT.a + DUT.b) != DUT.out) return 1; break;
          case 1:
            if((DUT.a - DUT.b) != DUT.out) return 1; break;
          case 2:
            if((DUT.a & DUT.b) != DUT.out) return 1; break;
          case 3:
            if((DUT.a | DUT.b) != DUT.out) return 1; break;
        }
      }
    }
  }
}
```

The call to `DUT.eval()` advances the state of the simulation. This simple test progam returns zero if the output of the DUT is correct and one if the output is incorrect.

Verilator can optionally add waveform generation and coverage information* to the C++ model if requested. These incur minor performance penalties and so are left off when rapidly iterating on a design or testing strictly for operational success, but are essential to debugging and tracking the health of testing infrastructure.

*The use of these capabilities normally requires some intervention by the test program, but for ProcDesign it is handled transparently by the team's testing utility libraries.

Unique among the ProcDesign toolchain, Verilator poses a portability challenge. While Verilator can compile cleanly under Microsoft Windows and MSVC, it has Unix-style path assumptions built into its frontend that make *using* Verilator on Windows functionally impossible. For now, ProcDesign teammembers on Windows run the toolchain via WSL, but fixes for Verilator's pathing assumptions are being explored.

2.3 *CMake*

CMake[5] is a tool originally developed for managing source code build processes, but has evolved into a general system for managing tooling orchestration. CMake is a self-described "buildsystem generator", meaning that it determines the nature and order of jobs that must be performed,† but delegates the act of running said jobs to a dedicated buildsystem utility such as Make or Ninja.

[5] Kitware, Inc. *CMake* version 3.29.3. May 7, 2024.

†Put another way, the output of CMake is not the result desired by the user, but rather a list of instructions that will produce the result desired by the user. This list is then handed off to an associated tool that the list was designed for.

CMake serves as the primary interface for most verification operations performed by ProcDesign during the development process. CMake, as used by ProcDesign, can bootstrap most elements of the toolchain (with the exception of Verilator) without intervention from the user. It handles the invocation of Verilator and linking of the C++ model outputs into test simulation programs without burdening the user with the particulars of how such a process is performed.

CMake also provides an interface for configuration of particular simulator build settings, allowing elements like waveform generation and coverage information to be toggled without resorting to manually editing toolchain files.

The following listing demonstrates bootstrapping of vcpkg if the user has not described another dependency provider and has not disabled bootstrapping of vcpkg via the `NYU_FETCH_VCPKG` option:

```
option(NYU_FETCH_VCPKG "Fetch vcpkg if no toolchain is set" ON)

if(NOT DEFINED CMAKE_TOOLCHAIN_FILE AND NYU_FETCH_VCPKG)
  include(FetchContent)
  FetchContent_Declare(
    vcpkg
    GIT_REPOSITORY https://github.com/microsoft/vcpkg.git
    GIT_TAG master
    GIT_SHALLOW TRUE
  )
  FetchContent_MakeAvailable(vcpkg)
  set(CMAKE_TOOLCHAIN_FILE
    ${vcpkg_SOURCE_DIR}/scripts/buildsystems/vcpkg.cmake
    CACHE FILEPATH "Vcpkg toolchain file"
  )
  set(VCPKG_ROOT_DIR ${vcpkg_SOURCE_DIR} CACHE PATH "Vcpkg Root
    Directory")
endif()
```

Listing 5. Bootstrapping code from the ProcDesign component template

`FetchContent` is CMake's native package management solution, but as seen here it's rather verbose. ProcDesign uses it to bootstrap vcpkg, and then allows vcpkg to bootstrap all other dependencies from within CMake.

Options defined using the `option()` command can be controlled* via CMake command-line arguments, the following would turn off vcpkg bootstrapping:

```
cmake . -DNYU_FETCH_VCPKG=OFF
```

*Manually invoking commands, CMake or otherwise, is discouraged on the ProcDesign team. These options are typically configured via the user's development environment CMake integration. For example, the VSCode CMake Tools extension provides a "cmake.configureSettings" parameter to handle these options.

ProcDesign convention is to group HDL source code into "libraries" which describe a single component or group of related components. The following creates a library named "module" and associates a single HDL source file with it:

```
add_library(module INTERFACE)

nyu_add_sv(module
  Module.sv
)
```

Listing 6. Creating a library and adding a SystemVerilog file to it

The `nyu_*` commands are specific ProcDesign utilities for interfacing with Verilator. These are explored further in section 2.6.

This RTL component can then be linked into a C++ simulation program using code such as the following:

```
option(NYU_GEN_TRACES "Generate FST traces from tests" OFF)

if(NYU_GEN_TRACES)
  set(_traces_option TRACE_FST)
else()
  set(_traces_option)
endif()

add_executable(tests)
target_sources(tests PRIVATE
  Module.cpp
)
nyu_link_sv(tests PRIVATE module)
nyu_target_verilate(tests
  TOP_MODULES Module
  ARGS COVERAGE ${_traces_option}
)
```

Listing 7. Linking together an RTL component with a C++ testbench

Note that the generation of waveforms (here called "traces") is controlled with a CMake `option()`.

In contrast to FuseSOC, the `nyu_target_verilate()` command supports integrating multiple "top modules" into a single testing simulation.

Finally, CMake provides a framework for packaging* and distributing RTL components to be consumed by external projects. For ProcDesign, installing RTL files to a CMake export is done via a single command:

*The intricacies of CMake packaging are not within the scope of this report, but an introduction to the concepts can be found in [6].

```
nyu_install_sv(
  EXPORT projectTargets
  TARGETS module
  NAMESPACE nyu::
  EXPORT_DEST ${CMAKE_INSTALL_DATADIR}/project
  SV_DEST ${CMAKE_INSTALL_DATADIR}/project/rtl
)
```

Listing 8. Installing the `module` RTL library to an export named `projectTargets`

2.4 *vcpkg*

vcpkg[7] is a CMake-based package manager developed by Microsoft, ostensibly for C/C++ dependency management, but functionally suitable for any CMake-based toolchain regardless of the nature of the underlying dependencies.

[7] Microsoft. *vcpkg* Jan. 11, 2024.

Unlike CMake which has several custom commands implemented by the ProcDesign team to extend it for use with HDLs, ProcDesign's usage of vcpkg is entirely ordinary. The team maintains a vcpkg "registry"†, a metadata collection that describes CMake-compatible packages available for installation.

†The ProcDesign Registry: `https://github.com/NYU-Processor-Design/nyu-registry`

Each ProcDesign project has a `vcpkg.json` file that describes the dependencies and other metadata of that project, such as the following for the NYU RISC-V core:

```
{
  "name": "nyu-core",
  "version": "1.0.0",
  "description": "NYU Processor Design's Core Components Repo",
  "homepage": "https://github.com/NYU-Processor-Design/nyu-core",
  "maintainers": [],
  "license": "CC0-1.0",
  "dependencies": [
    "nyu-cmake",
    "nyu-util",
    "catch2"
  ],
  "vcpkg-configuration": {
    "default-registry": {
      "kind": "git",
      "baseline": "326d8b43e365352ba3ccadf388d989082fe0f2a6",
      "repository": "https://github.com/microsoft/vcpkg.git"
    },
    "registries": [
      {
        "kind": "git",
        "baseline": "7a6b61ca47ca041f1c6558c649cbc2ddbf11d57a",
        "repository": "https://github.com/NYU-Processor-Design/nyu-
    registry.git",
        "packages": [
          "nyu-*"
        ]
      }
    ]
  }
}
```

Listing 9. nyu-core's `vcpkg.json` file

Packages described in the `dependencies` section will automatically be installed and made available by vcpkg.

The default registry is configured to point to a Microsoft maintained registry, which contains many useful common packages for C++ development.

This entry configures vcpkg to use the NYU ProcDesign registry to provide any packages with names starting with `nyu-`.

Because vcpkg is designed around C/C++ packages, it has heuristics that alert maintainers of possible packaging errors relevant to C++. Of note for ProcDesign, vcpkg will warn about empty an "include" folder, devoid of C++ headers. The only minor divergence from standard vcpkg usage is disabling that alert with:

```
set(VCPKG_POLICY_EMPTY_INCLUDE_FOLDER enabled)
```

The specifics of registry management, portfile construction, and general maintainership of vcpkg packages is beyond the scope of this report. It is sufficient to say that ProcDesign's usage of vcpkg is consistent with the processes documented in vcpkg's reference materials with the exception of the minor issue discussed here.

vcpkg is largely transparent when following ProcDesign conventions, as it is automatically boostrapped and invoked by CMake in normal usage.

2.5 *Catch2*

Catch2[8] is a popular C++ testing framework. As Verilator outputs C++ models of RTL designs, it is appropriate to use a C++-specific test framework to organize and interface with tests exercising those models.*

Catch2 organizes operations into `TEST_CASE`s which can have arbitrary names and tags associated with them. Each test case represents a standalone exercise of the concept under test and can be run on its own, separate from the other test cases.

This example tests the previously described `AluDevice` using Catch2 following ProcDesign conventions:

[8] Hořeňovský, M. *Catch2* version 3.6.0. May 5, 2024.

* Catch2 and the ProcDesign testing utilities are comparable in purpose to the traditional RTL verification framework, "Universal Verification Methodology" (UVM), which those experienced with typical RTL design practices will be familiar with.

```
void test_op(uint8_t code, uint8_t(op)(uint8_t, uint8_t)) {
  static auto& DUT {nyu::getDUT<VAluDevice>()};
  DUT.op = code;
  DUT.a = 0;
  DUT.b = 0;

  do {
    do {
      nyu::eval(DUT);
      uint8_t result {op(DUT.a, DUT.b)};
      REQUIRE(result == DUT.out);
    } while(++DUT.b);
  } while(++DUT.a);
}

TEST_CASE("Opcode 0, Addition") {
  test_op(0, [](uint8_t a, uint8_t b) -> uint8_t { return a + b; });
}

TEST_CASE("Opcode 1, Subtraction") {
  test_op(1, [](uint8_t a, uint8_t b) -> uint8_t { return a - b; });
}

TEST_CASE("Opcode 2, And") {
  test_op(2, [](uint8_t a, uint8_t b) -> uint8_t { return a & b; });
}

TEST_CASE("Opcode 3, Or") {
  test_op(3, [](uint8_t a, uint8_t b) -> uint8_t { return a | b; });
}
```

Listing 10. Testing the `AluDevice` from Listing 3 using Catch2 and various ProcDesign utilities

`REQUIRE()` is a Catch2 macro that will cause the test to fail is the contained expression does not evaluate to true.

`TEST_CASE()` allows tests to be divided into distinct operations, so that testing and debugging of one operation does not require running the entire test suite on a given component.

In this example each operation supported by the `AluDevice` is given its own `TEST_CASE()`, but they are able to share the common `test_op()` function.

Catch2 provides its own `main()` which handles parsing command line arguments, running requested tests, and reporting results. For example, the following command[*] would run just the addition test of the above program:

[*] As discussed previously, manually running such commands is not a normal workflow for ProcDesign, and is merely discussed here for completeness.

```
./tests "Opcode 0\, Addition"
```

Which will result in:

```
Filters: "Opcode 0\, Addition"
Randomness seeded to: 773214706
=====================================================
All tests passed (65536 assertions in 1 test case)
```

Effectively all development environments have integrations for specifying and running individual tests from a dedicated test panel which tracks far more metrics (test runtime, result history, etc) than is available from manually invoking test binaries.

Invoking the test program without any arguments will run all the available tests, resulting in:

```
Randomness seeded to: 2727790692
=====================================================
All tests passed (262144 assertions in 4 test cases)
```

ProcDesign uses CMakes's CTest infrastructure to drive testing in most contexts, and Catch2 provides the ability to automatically integerate `TEST_CASE`s with CTest drivers with the following:

```
include(Catch)
catch_discover_tests(tests)
```

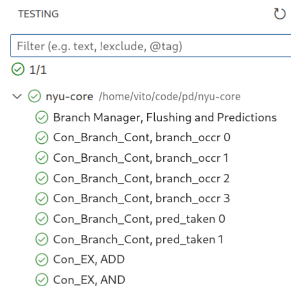

Figure 2.2.
VSCode integrated CTest panel

Catch2 provides infrastructure for setting up and tearing down test instrumentation on a per test case basis. ProcDesign uses this capacity to serialize coverage information[†] to disk after each test, and to clean up coverage state information between tests. The ProcDesign testing utilities also use Catch2 to name and serialize waveform data to disk if that capacity has been enabled for a given RTL design.

[†] **Coverage information** is a line-by-line accounting of what elements of the source code being tested has been exercised by a given test. Ideally, the entirety of an RTL component's source code should be exercised and verified by tests.

2.6 *ProcDesign Utilities*

The ProcDesign team maintains a small set of utilities for adapting CMake, Verilator, and Catch2 to its own conventions. These can be split into two main categories:

- CMake functions used to track and maintain collections of HDL source files
- C++ libraries and headers that provide convenient mechanisms for correctly authoring verification tests inside the Catch2 testing environment

There have been recent efforts to provide native CMake support for HDLs[9], but there has not been widespread consensus on how to integrate the nature of an RTL simulation with the abstractions currently provided by CMake for software development. It may be that new abstractions are needed.

[9] *CMake Issue: Verilog: Support HDL Languages.*

In any case, ProcDesign sidesteps the issue by not attempting to add HDL support to native CMake functions. Instead the team has implemented its own suite of `nyu_` prefixed functions such as `nyu_add_sv` which associates SystemVerilog source code to a specific library or simulation executable. It operates as follows:

```
function(nyu_add_sv TARGET)
  foreach(_src IN LISTS ARGN)
    file(REAL_PATH ${_src} _real)
    set_property(TARGET ${TARGET} APPEND
      PROPERTY SV_SOURCES ${_real}
    )
    set_property(TARGET ${TARGET} APPEND
      PROPERTY SV_SRC_PATHS ${_real}
    )
    set_property(TARGET ${TARGET} APPEND
      PROPERTY SV_SOURCES_NOGENEX ${_real}
    )
  endforeach()
endfunction()
```

Listing 11. `nyu_add_sv` CMake function

This function tracks SystemVerilog source files via manipulation of various properties specific to the NYU utility functions.

All of the `nyu_` CMake functions operate on this same principle, which mirrors how CMake internally implements support for various features.

The C++ testing utilities consist mostly of conveniences on top of native Verilator behavior, such as an `eval()` function that can produce multiple state evaluations in a single call, or a `reset()` function that is able to fully reset components designed following ProcDesign conventions.

One function that deserves attention is `getDUT()`,* as it solves two complicated issues for ProcDesign. The first is the issue of lifetime, Verilator models carry state information that must persist long enough to be serialized into test reports. If models were allowed to go out of scope at the end of a test, the Catch2 test reporters would not be able to access that state information. `getDUT()` ensures the lifetime of models persists appropriately throughout the simulation.

* `getDUT()` usage is demonstrated in Listing 10.

`getDUT()` has been subject to a number of bugs, as its behavior is somewhat counter to normal Verilator usage. Most recently its overload mechanism was broken due to an innocuous change in how Verilator handled individual model tracing. See [10]

The second is transparently treating tracing models (models which Verilator has generated to create waveforms) and non-tracing models the exact same, using the exact same test code, with the behavior configured by only a CMake `option()`. The `getDUT()` function uses C++20 concepts to provide overloads which return wrapped versions of tracing models. In tests these wrappers can be treated identically to non-tracing models.

3 *Beyond the Toolchain*

Other elements of the ProcDesign process, beyond just the technical details of its toolchain, are unorthodox for an academic RTL design effort. These are worth some consideration insomuch as they have impacts on the development process as great or greater than the toolchain.

3.1 *SCM and Documentation*

In keeping with the principle of developing of useful, relevant skills that members can make use of in either the software or the hardware space, best practices in source code management (SCM) and documentation are a core focus for the ProcDesign team. The SCM mechanism used by the team is git[11] and the primary documentation mechanism is Github Flavored Markdown (GFM).[12]

[11] Hamano, J. *et al. git* version 2.45.1. Apr. 29, 2024.

[12] Github. *GitHub Flavored Markdown Spec* version 0.29-gfm (Apr. 6, 2019).

ProcDesign uses an open-source inspired approach to source code management. Teammembers fork and manage their own repositories of projects they are working on, with contributions incorporated via pull requests.[*] The upstream ProcDesign repositories are controlled by experienced members of the team's leadership, with technical overseers of particular elements named as "Czars".

[*] **Pull Requests** are a process by which an individual petitions to have their code incorporated into a repository controlled by a different individual. Many git-hosting sites have formalized this process with dedicated workflows and managed tools. ProcDesign uses Github's Pull Request mechanism.

For example, the Core Czar is the immediate supervisor of the RISC-V core and controls the merging of code into the upstream repository from which the other teammembers fork. Czars also oversee elements such as the bus architectures, memory subsystem, documentation, and software components of the project.

ProcDesign uses a rebase-based git workflow, and encourages the use of Conventional Commits[13] for commit messages. Strong indoctrination of teammembers into git usages early in their introduction with the team has minimized ProcDesign's exposure to anti-patterns such as large blocks of commented code, or multiple versions of components stored in various backup locations.

[13] Petrungaro, D. *et al. Conventional Commits* version 1.0.0 (Apr. 20, 2020).

Documentation is kept entirely inside the various project git repositories.[†] As documentation is written in GFM, it is navigable and viewable directly in the repositories' Github web interfaces. Development environments used by ProcDesign typically also have integrations for rendering GFM.

[†] See Appendix A for an example of the rendered documentation.

3.2 *Continuous Integration*

Continuous integration refers to the process of continually testing a product or component as it is developed.[14] As a practical matter for most modern software development, this means running the entire testsuite on each pull request and inspecting test results and artifacts before deciding to merge a change.

[14] Booch, G. *et al. Object-Oriented Analysis and Design with Applications* ISBN: 978-0201895513 (Addison Wesley, Mar. 2001).

Whenever a ProcDesign teammember opens a pull request against a project repository, the testsuite is automatically run via Github Actions. This is possible because

of the portability of the NYU toolchain, having no commercial components that are not readily available on free-to-use testing infrastructure. An entire ProcDesign test flow is described here:

```
steps:
  - name: Install Build Dependencies
    run: |
      pacman --noconfirm -Syu
      pacman --noconfirm -S cmake ninja git curl zip unzip tar
    verilator

  - name: Checkout
    uses: actions/checkout@v4

  - name: Configure
    run: cmake . -G Ninja -DNYU_BUILD_TESTS=ON

  - name: Build
    run: cmake --build . --config Release

  - name: Test & Generate Coverage
    run: |
      ctest -C Release --output-on-failure
      sed -i -e '/\/share\//d' -e '/\/dv\//d' dv/*.dat
      verilator_coverage -write-info coverage.txt dv/*.dat

  - name: Upload Coverage
    uses: codecov/codecov-action@v4
    with:
      token: ${{ secrets.CODECOV_TOKEN }}
      files: ./coverage.txt
      fail_ci_if_error: true
```

Listing 12. Test workflow from the ProcDesign component template

The `NYU_BUILD_TESTS` CMake option controls whether or not tests are built. When used as a dependency of a parent project, tests for child components are not built.

CTest is used to run all the test cases that have been implemented for the RTL components in the repo. The following two lines coalesce the generated coverage data into a single file.

Finally, the coverage is uploaded to Codecov, a coverage tracking and alert system.

The continuous integration workflow provides project Czars with important information directly, as failing tests are an obvious indication that the changes are unready to be merged. A more in-depth metric is provided by the coverage information.

ProcDesign uses the dedicated Codecov service which will alert the project Czar if testing coverage has regressed, meaning a lower percentage of HDL source code lines are being tested after a pull request than before. As most ProcDesign repositories are at 100% coverage, this means attempting to add *any* untested HDL code to a repo causes an alert.

4 *Future Work*

There are toolchain challenges that are being actively tackled by the NYU Processor Design teammembers, such as native CMake support for HDLs, and the Verilator pathing issues on Windows. The latter especially represents a major roadblock for onboarding new engineering students into the RTL design space.

The ProcDesign team remains primarily focused on RTL design and verification, and has not yet had to fully tackled other elements of hardware fabrication such as memory compilers, synthesis for FPGA, or attempting a tapeout. These future challenges present ample oppertunities to test the NYU approach of merging modern software and hardware toolchains and workflows.

Perhaps most pressingly, as the ProcDesign team matures and founding members graduate and move on, the strength of the documentation, as well as the ease and intuitiveness of the toolchain components, will be tested. Only time will tell if these toolchain mechanisms prove resilient under the churn of new personnel, or if they prove too obtrusive and are abandoned for more traditional tooling.

APPENDIX

A Documentation Example: Edge Synchronizer

Edge Synchronizer

The edge synchronizer is a 2-bit synchronizer and an edge detector combined into a single module.

Contents

- Inputs
- Outputs
- Functionality

Inputs

Name	Width	Description
`clk`	1	Input clock
`nReset`	1	Asynchronous active-low reset
`in`	1	Asynchronous input signal

Outputs

Name	Width	Description
`out`	1	Synchronized output signal
`rise`	1	Rising edge detected
`fall`	1	Falling edge detected

Functionality

Combining a synchronizer and an edge detector creates a three-bit shift register. For convenience, each bit has a name:

2	1	0
`cmp`	`out`	`sync`

Each clock cycle, the shift register is shifted left one bit and the `in` signal is read into the `sync` bit.

The `out` bit is the synchronized output from the module.

The `cmp` bit is used to detect a rising or falling edge according to the following logic:

```
rise =  out && !cmp;
fall = !out &&  cmp;
```

Bibliography

1. Bening, L. & Foster, H. *Principles of Verifiable RTL Design* ISBN: 0-7923-7788-5 (Jan. 2002).
2. Swabey, M. A. & Johnson, M. C. *Satisfying ABET criterion using an industrial Microelectronic Skills Incubator* in *2015 IEEE International Conference on Microelectronics Systems Education (MSE)* (2015), 28–31.
3. Kindgren, O. *A Scalable Approach to IP Management with FuseSoC* in *2019 Workshop on Open Source Design Automation (OSDA)* (2019).
4. Snyder, W. *Verilator* version 5.024. Veripool, Apr. 5, 2024. `https://www.veripool.org/ftp/verilator_doc.pdf`.
5. Kitware, Inc. *CMake* version 3.29.3. May 7, 2024. `https://cmake.org/cmake/help/v3.29/`.
6. Gamberini, V. *Modern CMake Packaging: A Guide* `https://blog.vito.nyc/posts/cmake-pkg/`.
7. Microsoft. *vcpkg* Jan. 11, 2024. `https://learn.microsoft.com/en-us/vcpkg/`.
8. Hořeňovský, M. *Catch2* version 3.6.0. May 5, 2024. `https://github.com/catchorg/Catch2`.
9. *CMake Issue: Verilog: Support HDL Languages* `https://gitlab.kitware.com/cmake/cmake/-/issues/23387`.
10. Gamberini, V. & Snyder, W. *Verilator Issue: Add traceCapable indication* `https://github.com/verilator/verilator/issues/5053`.
11. Hamano, J. *et al. git* version 2.45.1. Apr. 29, 2024. `https://git-scm.com/`.
12. Github. *GitHub Flavored Markdown Spec* version 0.29-gfm (Apr. 6, 2019). `https://github.github.com/gfm/`.
13. Petrungaro, D. *et al. Conventional Commits* version 1.0.0 (Apr. 20, 2020). `https://www.conventionalcommits.org/en/v1.0.0/`.
14. Booch, G. *et al. Object-Oriented Analysis and Design with Applications* 3rd ed. ISBN: 978-0201895513 (Addison Wesley, Mar. 2001).

This document was typeset using LaTeX and a modified version of the `tufte-style-thesis` class.
The style is heavily inspired by the works of Edward R. Tufte and Robert Bringhurst.
The original class is available here: `https://github.com/sylvain-kern/tufte-style-thesis/`.
Feel free to contribute!