# Project Synopsis

## Project Introduction

- Project: UVM Code Generation on Per Release Basis for NVMe IP
- Company: Intel Corporation
- Degree of Technical Contribution: 100%
- Completion Time: 2 months at ~20% capacity

At Intel, I developed a UVM Code Generation tool that takes an XML file supplied by the architecture team and auto generates thousands of class definitions for NVMe registers and register files within minutes. The project was vital to the design verification process and greatly improved the team's ability to develop tests and increase code coverage.

## Project Requirements and Goals

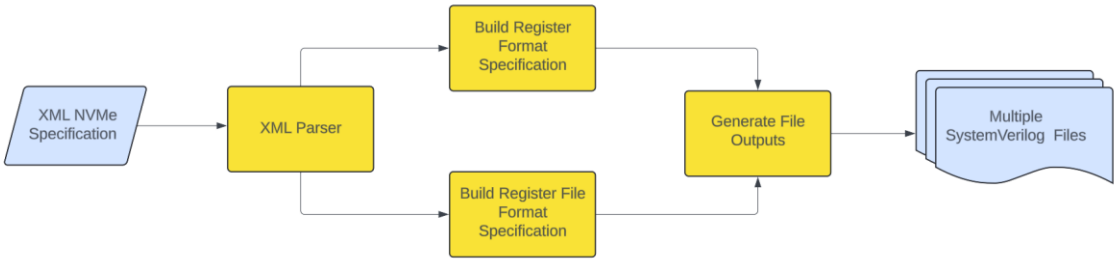
The table below shows the application requirements and the capabilities of the final product. The requirements were extracted from several design specification meetings. I completed all requirements and 66% of my stretch goals before completing my internship.

| Capability | Requirement | Stretch Goal | Accomplishment |
|---|---|---|---|
| **Output Files** | 1 SystemVerilog UVM output file per register block, 1 SystemVerilog UVM output file per register | N/A | Requirement Satisfied |
| **Application Run Time** | < 3 Hours | As Low as Possible | < 3 minutes; Stretch goal and requirement satisfied |
| **Compilation** | Code should compile (slight manual intervention acceptable) | Code should compile – No manual intervention necessary | Requirement satisfied |
| **Scalability** | N/A | Script usable across other IPs at Intel | Stretch Goal Satisfied – Script is usable and tested on neighboring IP |
| **Adaptation** | Script should adapt to architecture changes | N/A | Requirement Satisfied |
| **Automaticity** | Script should be integrated within the CI to be run per release. | N/A | Requirement Satisfied |

These specifications are designed to maximize reliability and impact. In addition to the requirements above, I thought of making the application usable at other IPs (scalability goal) so that it becomes useful across the organization. This scalability goal heavily influenced my design decisions.

## Application Architecture

The flow chart below shows a high-level overview of the application design. The yellow blocks represent processes. The blue blocks represent input / output files:




XML NVMe Specification: Contains the entire NVMe Architecture specification including the names of register blocks, registers, fields, and descriptions.

XML Parser: Python module which parses the XML data to be processed and sent.

Build Register (File) Format Specification: Python module that uses Jinja templates and YML files to specify the format of the Register (File) class definition. Since these class definitions are different, they are specified in two different formats.

Generate File Outputs: The files with the formatted code are generated and placed in the appropriate directories within the codebase.

Multiple SystemVerilog Files: Contain the class definitions for all NVMe register blocks and registers in separate SystemVerilog Files.

## Design Decisions

To meet the design requirements, I implemented the application with the following features.

1. The application is run with 2 arguments. (1) XML file location, and (2) output directory. All other static files needed for processing such as YML files and Jinja templates are contained within an isolated environment. This ensures the application is simple to use and guarantees that other IPs can benefit from its use without having to worry about excess collateral inputs and outputs.
2. The XML file format is the most widely used file type for architecture specification across Intel. Since the XML file is kept up to date by the architecture team, the application can easily respond to evolving architectural changes.
3. Multiple threads are used to speed up the input / output operations for all files.
4. Using Jinja templates and YML files to specify format makes format modifications simple. This further accomplishes the scalability requirement.

Github: https://github.com/ChinmayBhide154