Introduction

Chip design has increasingly become a more and more important field. Semiconductor chips are the building blocks of all modern electronics and are crucial for data processing, communication infrastructure, transportation, and even seemingly unrelated fields like healthcare and entertainment. Due to their far stretch, it has been extremely important to strive for increased computing power, power efficiency, and smaller chip sizes.

Chip Tape out cycle:

Due to the expensive nature of manufacturing these chips, it is also extremely important that there is great care in the chip making process. This results in an extremely long design cycle where chips are carefully designed with HDLs like Verilog or VHDL. While this is being done, the logical design is also thoroughly tested with software verification and other techniques to ensure its correctness.

After the logical design has been verified, it is built into a physical chip by placing the transistors and other components onto a chip in a step called layout design. In the next step of floorplanning, engineers are then responsible for determining the physical arrangement of the chip components in order to reach the necessary size, power, and routing constraints that are included.

Once the components have been placed, the interconnects are carefully placed between the components so that there can be efficient communication between the chip components. From here, the layout is verified to ensure that it adheres to the initial logical design. This includes very thorough checks for connectivity, spacing between units to prevent signal interference, and adherence to the design specifications.

After the layout has been signed off and approved, the masks or templates for the wafer are designed so as to ensure the manufacturer will be able to create the exact patterns and routes needed in the semiconductor. These masks are taped out and sent to the manufacturer which actually creates the wafers layer by layer. These completed silicon chips are sent back where the chips are thoroughly vetted to ensure that there were not any issues in the manufacturing process.

If the chips have passed for quality checks, they can be shipped out to customers or given to other companies for integration into other electronics. In the case that chips are incorrect after manufacturing, the company has to re-design the incorrect portions of the chip and re-tape it out.

Due to the long process, this results in a lot of wasted time, effort, and money. Once the design has been corrected, the masks and templates must be adjusted in order to create the new chip design. This results in an extremely high expense since the company may have to re-tape out multiple times – often doubling or tripling their initial costs. They may also incur additional costs as penalties due to the manufacturer having to adjust their initial chip design templates. If a mistake is not caught before it reaches the markets, there may be a general loss of trust from the consumer market or in the worst case legal consequences. [6]

However, it is extremely difficult to guarantee a design is completely bug-free. There are many instances where issues and bugs are not caught in the thorough verification process, which can cost upwards of $900 million.[4] Due to this high cost of both time, money, and effort, verification is an extremely important step in order to try to prevent as many of these issues as possible. As such, there are many verification techniques that are used in order to guarantee a high level of correctness efficiently.

Hardware Verification Techniques:

The goal of verification is to address design flaws and ensure the product meets specifications before it gets released to the public. It is a long and a multi-step process that primarily involves two parts: pre and post silicon validation

Pre-silicon validation is done prior to the tape out process and before the actual manufacturing of the chips in silicon. While it is primarily done to find logic design errors, it has also proven to find more crucial bugs and sources of sensitive data leaks.[2] It primarily uses simulation, analysis, and testing at a higher abstraction level to ensure correctness of its functionality and ensures that it adheres to the specifications. [3]

The key aspects of pre-silicon validation include:

1. Functional Verification: Engineers use simulation tools to verify the logical correctness of the chip's design. This involves running simulations to ensure that the chip performs its intended functions according to the specifications provided during the design phase. This is primarily done through writing unit tests with UVM in order to cover specific test cases.

2. Timing Analysis: Timing verification ensures that the chip's operations meet the required timing constraints. This includes checking clock frequency, setup and hold times, and propagation delays to prevent timing-related issues. This can be done with software profiling to simulate the more physical aspects of the chip and determine whether or not it is able to meet the desire timing requirements.

3. Power Analysis: Power consumption is a critical concern in modern chip design. Pre-silicon validation includes analysis of power usage at different operating conditions to ensure it aligns with power budgets and does not exceed specified limits. Likewise, this can be done with profiling to simulate the hardware and determine potential limitations before the design is made into silicon.

4. Fault Simulation: Engineers simulate potential faults to assess how the chip behaves in the presence of defects or errors. This helps in designing robust error detection and correction mechanisms.

5. Modeling and Abstraction: Different levels of abstraction, such as high-level architectural models and lower-level RTL (Register-Transfer Level) descriptions, are verified to ensure consistency across the design hierarchy.

6. Interoperability Testing: In complex systems-on-chip (SoCs) where multiple IP blocks or components interact, pre-silicon validation includes testing the interoperability of these blocks to ensure seamless communication.

7. Hardware Emulation: For large and complex designs, hardware emulation platforms may be used to validate the design at near-real-time speeds. This allows for more comprehensive testing and validation before moving to the costly silicon fabrication stage.

8. Prototyping: Physical prototypes or FPGA (Field-Programmable Gate Array) implementations of the chip may be created to validate the design in a hardware environment before committing to silicon.

These mentioned processes are relatively cheap and do not require any particularly difficult to acquire or niche skills. By writing unit tests, using software profiling tools, and writing coverage tests to determine what parts of the code base have not been tested yet, engineers can guarantee some level of correctness of their design.. However, software does not emulate many of the difficulties that can be encountered in hardware and is thus infeasible to capture the scope of all potential problems that may arise. The primary drawback is that exhaustive testing is generally difficult to achieve for large designs and there are often edge cases that validation engineers are unable to think of.

Because of these limitations, formal verification has become an increasingly popular approach to verify software designs. Formal verification aims to mathematically define a system and use mathematical proofs in order to rigorously prove correctness of that system. This involves first creating this formal description of the system and implementation and then trying to prove that the system will always produce a result that adheres to the defined specifications. Especially in hardware designs where everything is in bits and terms of logical gates, going down a more formal route and mathematically defining everything is a very intuitive approach.

While formal methods can guarantee stronger notions of correctness, there are many limitations that come with it. There are some frameworks that are capable of making this process relatively automatic and thus have a similar cost to writing and running unit tests. However, these frameworks are often limited to smaller sizes due to the correlation between size of the problem statement and time or resources to solve the problem. There are also methods of being able to address this issue such as predicate abstraction to simplify the problem or even more powerful frameworks that are able to solve these larger problems. However, being able to use these types of tools requires a large amount of experience and time in order to use them correctly. As such, for larger and more complex designs, companies will need to hire many skilled experts with these tools and will require a lot of time in order to formally verify the correctness of their designs. As such, being able to achieve the strong guarantees of correctness is difficult to achieve efficiently.

Post silicon validation is also an extremely important aspect of ensuring correctness of a design. The primary goal is to ensure that the physical hardware continues to function according to the design specifications and that the manufacturing of the chip remains correct.

This primarily consists of multiple steps:

1. Functional Verification: This involves testing the chip's functionality in real-world conditions. Engineers check whether the chip performs the intended tasks and functions correctly under various scenarios. This step aims to identify and rectify any discrepancies between the expected behavior specified during the design phase and the actual behavior exhibited by the physical chip.

2. Performance Testing: Engineers assess the chip's performance by measuring parameters such as speed, power consumption, and heat dissipation. This step ensures that the chip meets the specified performance requirements and operates within acceptable limits.

3. Power and Signal Integrity Testing: Post-silicon validation includes checking for power integrity issues, ensuring that the chip operates within the specified power supply limits. Signal integrity testing ensures that signals propagate correctly through the chip without distortion or loss.

4. Interconnect Testing: The interconnects between different components on the chip are scrutinized to ensure that they have been accurately implemented. This involves checking the pathways for electrical continuity and potential interference issues.

5. Clock Domain Crossing Verification: Chips often have multiple clock domains, and crossing between these domains can introduce synchronization issues. Post-silicon validation includes verification of proper clock domain crossing to prevent timing-related problems.

6. Environmental Testing: Chips may be subjected to different environmental conditions, such as varying temperatures and voltages, to assess their performance under diverse operating conditions.
7. Debugging and Diagnostics: In the event of discovering issues during post-silicon validation, engineers use debugging tools and diagnostic techniques to identify the root causes of problems. This may involve the use of on-chip instrumentation, built-in self-test (BIST) features, or external testing equipment.

8. Revision and Re-Tapeout: If significant issues are identified during post-silicon validation, design revisions may be necessary. This could involve modifying the chip's design to address discovered problems, followed by the creation of new masks or templates for a subsequent manufacturing run (re-tapeout).

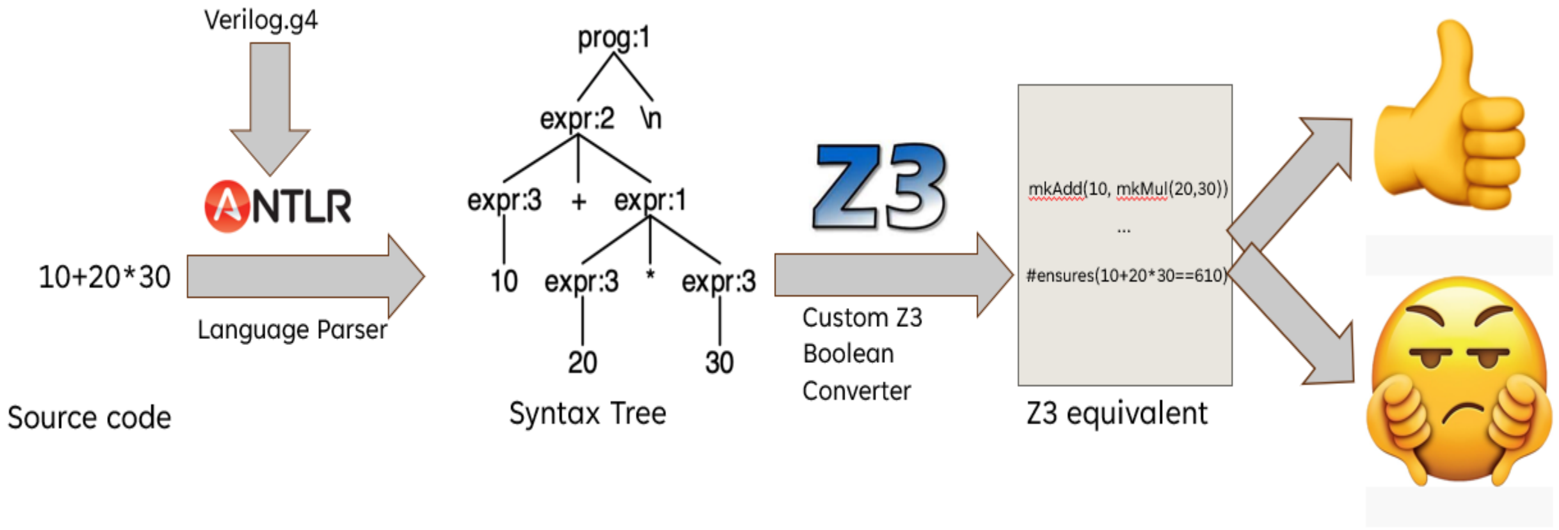
Post-silicon validation is a critical step in ensuring the reliability and functionality of semiconductor chips. It complements pre-silicon validation efforts and contributes to the overall quality and success of integrated circuit designs. The process is resource-intensive and requires collaboration among design, verification, and testing teams to address any issues that may arise in the transition from the design phase to physical silicon.

Motivation:

One of the primary motivating factors was due to my time as a design verification intern at Intel. I experience first hand using UVM and writing the unit tests in order to verify correctness of a design. Many of my co-workers were unfamiliar with the concept of formal verification. While they were fascinated by the strong guarantees that formal can provide, they were reluctant to learn a new language or framework. Additionally, they were unfamiliar with the higher level specs of the design and were primarily concerned with the verification of the specific submodule that they were working on. The main goal of this project was to create something that could provide some formal guarantees of a piece of Verilog code without straying far from the program specifications of Verilog.

Project:

This gave birth to the initial idea of “The Unusually Formal Method” (UFM), a simple program verifier for the HDL Verilog. The basic premise is inspired by the program verifier Dafny we learned about in class. In particular, I was inspired by its supposed simplicity where the programmer simply needs to write code and provide pre/post conditions and Dafny would automatically verify it for you. Likewise, my program verifier will be able to prove whether or not a module will be able to adhere to user-defined post-condition. The end goal is to be able to write a piece of Verilog code, add a special kind of post condition comment at the end, simply run my program verifier script, and the program verifier will output whether or not it was successfully able to verify or not.



The above figure demonstrates the UFM program verifier pipeline. First, the programmer must supply a section of Verilog code. This Verilog source This will then be parsed through ANTLR[1] ((ANother Tool for Language Recognition) which will convert the source code to an abstract syntax tree (AST). This AST is a simple hierarchical representation of the described module that makes it easy to parse through. From the AST, the primary goal is to use Microsoft Research's Z3 theorem solver [5] to prove correctness of the structure. Then a Z3 boolean converter program will parse through the AST to convert the Verilog source code into a Z3 equivalent script representation. The programmer can simply run the Z3 script which will output whether or not the program was successfully verified or not.

Project Tasks:

The primary contributions I made to this project are, as listed in the diagram above, the Verilog.g4 grammar and syntax while as well as the custom Z3 Boolean Converter, also called verilog2z3.py. I also implemented a simple 2 bit adder with a carry as a Verilog module to test my code with.

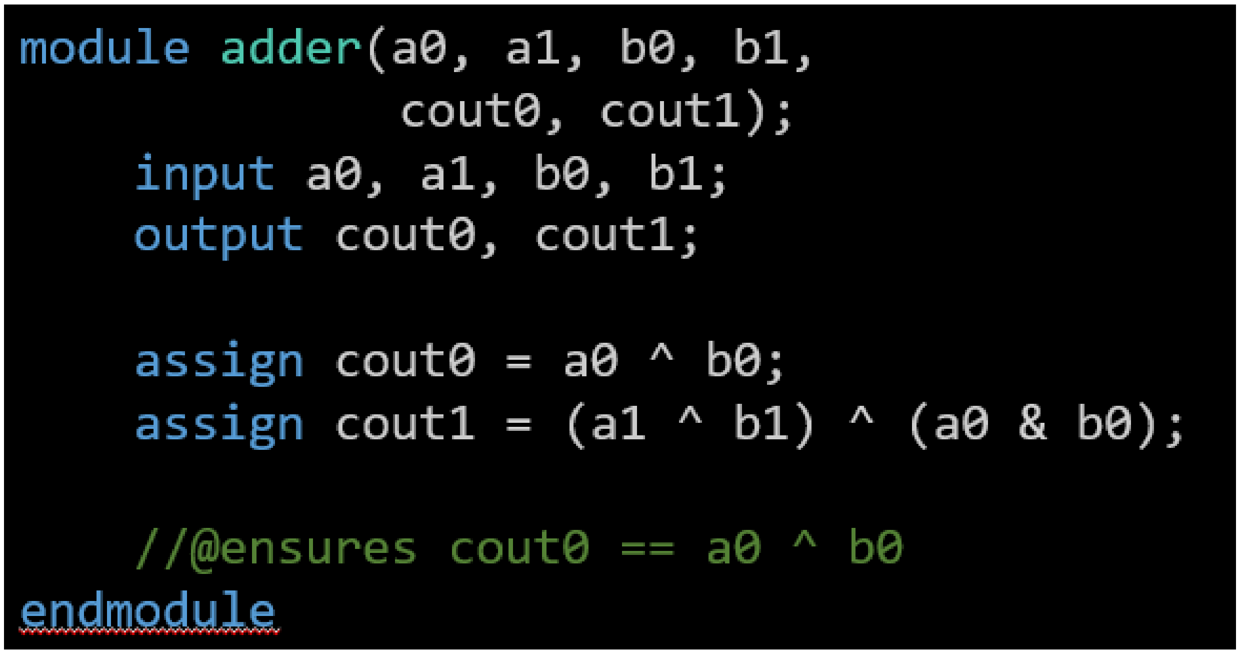
The primary difficulty that was encountered was unfamiliarity with .g4 grammar syntax and the large amount of syntax that is included in the Verilog language. Because of this, I found some sources online for assisting with the writing of syntax and grammar files. Even so, it took a long time for me to convert it into the correct format.

Another minor difficulty that was encountered was unfamiliarity with the AST structure and how to parse through it. While I was eventually able to accomplish this as a potentially inelegant method, a lot of time was used to produce very simple functionality. Due to my writing of the program in a stack like manner, it makes the program parsing overall very inflexible and if I were to extend it to the entire language of Verilog, I would likely encounter many issues.

Overall, I believe due to my inexperience and lack of background in the area of programming languages and compilers, I may have opted to approach these in a less than optimal way.

Results:

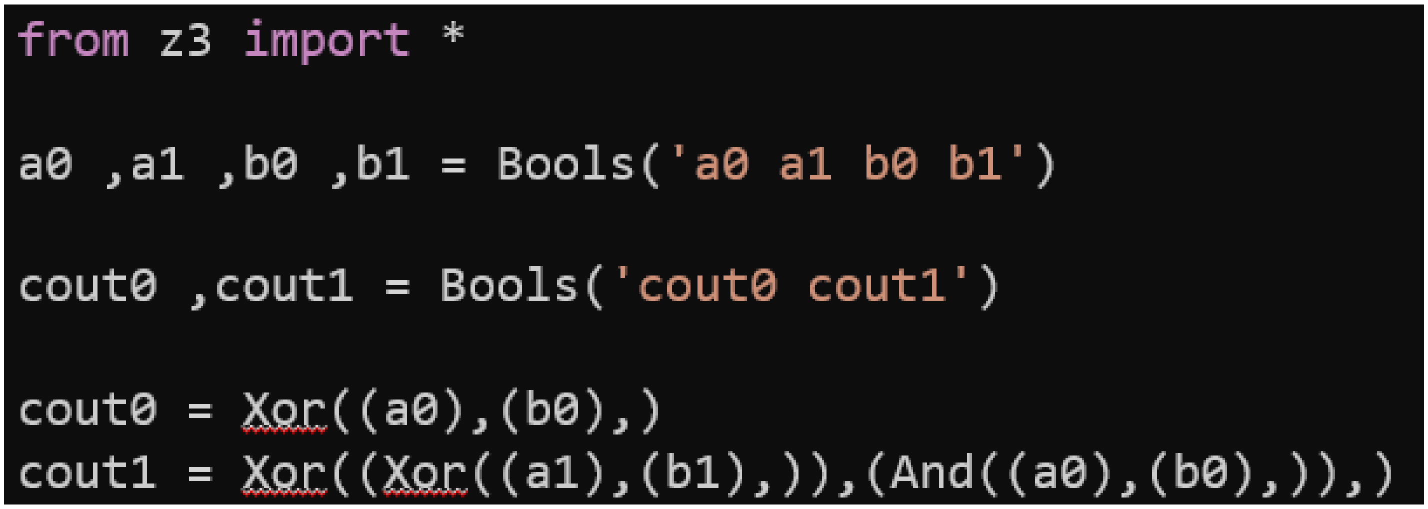
The current implementation of code is as follows. A programmer has a module of code that they want to formally verify. The programmer is then responsible to convert this into the exact bits and logic gates that would be needed in order to accomplish this. In the case for a two bit adder, for example, the code would look like something as follows:



```
module adder(a0, a1, b0, b1,
             cout0, cout1);
    input a0, a1, b0, b1;
    output cout0, cout1;

    assign cout0 = a0 ^ b0;
    assign cout1 = (a1 ^ b1) ^ (a0 & b0);

    //@ensures cout0 == a0 ^ b0
endmodule
```

In this module, we have two inputs a and b. They have been expanded into individual bits. The individual output bits are likewise also similarly assigned according to the exact logic gates that are used. This responsibility will fall into the hands of the programmer to either manually or use external tools in order to break down their design into the bitwise level components. Additionally, they will be responsible for adding an annotated post condition, which is a comment that is appended with the symbol "//@", and entails an expression that the programmer would like to show their code will be able to maintain or not.

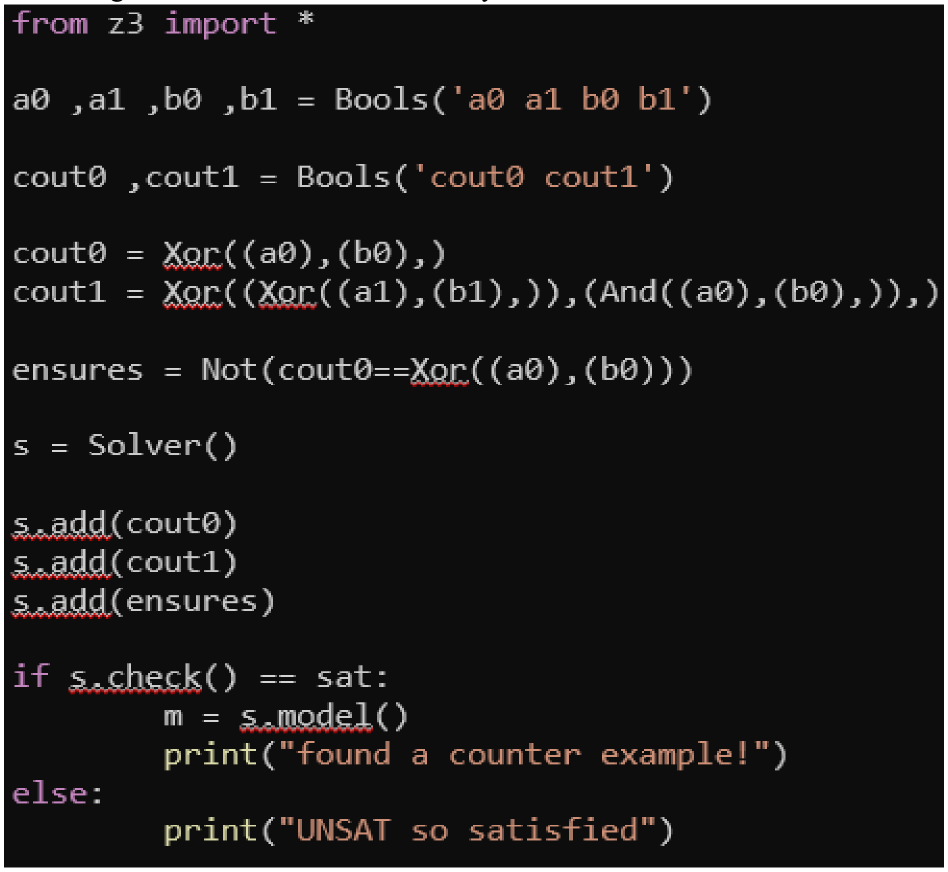
From here, the programmer can simply run the supplied verilog2z3.py script file. This will parse the original source code and produce a Z3 equivalent similar to what is shown below:



```
from z3 import *

a0 ,a1 ,b0 ,b1 = Bools('a0 a1 b0 b1')

cout0 ,cout1 = Bools('cout0 cout1')

cout0 = Xor((a0),(b0),)
cout1 = Xor((Xor((a1),(b1),)),(And((a0),(b0),)),)
```

Due to current limitations in my design, the grammar file is currently not written in a way such that ANTLR is able to correctly parse the Verilog files with annotated post conditions. Instead, it continues to treat them as comments as the annotated post condition begins with “//” similar to a comment.

Because of this, the programmer will have to manually interpret the annotated post condition themself. This can be shown as follows below, where the ensures clause is first written in a Z3 Verilog equivalent. The programmer must negate the ensures clause as if we can prove infeasibility of the assertion at the end, that guarantees that there is no input that can cause the assertion to fail. As such, we can guarantee a high level of correctness for our system.



```
from z3 import *

a0 ,a1 ,b0 ,b1 = Bools('a0 a1 b0 b1')

cout0 ,cout1 = Bools('cout0 cout1')

cout0 = Xor((a0),(b0),)
cout1 = Xor((Xor((a1),(b1),)),(And((a0),(b0),)),)

ensures = Not(cout0==Xor((a0),(b0)))

s = Solver()

s.add(cout0)
s.add(cout1)
s.add(ensures)

if s.check() == sat:
        m = s.model()
        print("found a counter example!")
else:
        print("UNSAT so satisfied")
```

Discussion:

There were many difficulties that I encountered throughout this project. In my project proposal, I mentioned that I was hoping to use Yosys and Tabby Cad Suit in order to take advantage of existing Verilog programming parsers. However,while they do provide formal guarantees within your code, it was difficult to only use certain sections. Specifically, I planned on using the existing AST parser within Yosys to produce my own AST. However, the AST parser was "atomic" in the sense that it could not be separated from Yosys's own formal program verifier. The most I figured out I could do was to do an AST dump to print out the AST in the Yosys log files. However, going through the log files and parsing them felt excessive for a program verifier that is meant to be "easy and less work" for the programmer. For these reasons, I decided to pivot from using Yosys as my AST parser and looked for something different.

I learned about the ANTLR, which was a powerful program parsing. It was seemingly able to parse through any language. However, I hesitated on using it since it seems that it requires the user to provide the syntax for the language in order for it to parse through it. Since I had no prior experience with ANTLR, it felt very intimidating to try to use it to describe a language as large as Verilog.

With some searching and assistance, I learned about the Backus-Naur form (BNF). BNF was described to be a metasyntax notation that is capable of formally describing the grammar and syntax of languages. Since BNF is able to formally and thoroughly be able to encapsulate the grammar and syntax of a language, being able to use Verilog's BNF meant that writing the grammar file for ANTLR would be a lot simpler. I was able to find a BNF for the Verilog2001 Spec, which I painstakingly parsed through and converted it into ANTLR *.g4 syntax. I removed some parts of the Verilog spec that I felt was not important for the context of my program verifier due to time limitations.

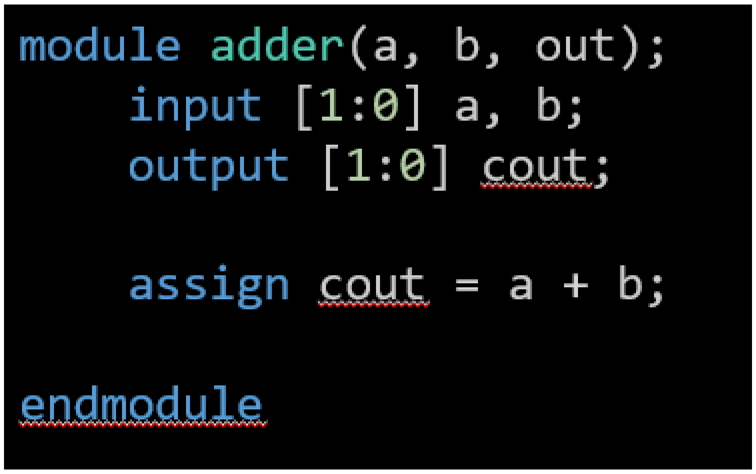
Overall, writing the grammar for the file was extremely time consuming and difficult. Had I had more experience, I could have potentially saved some time by omitting more aspects of the Verilog BNF that were not useful for the context of my program verifier. Additionally, I could have saved a lot of time in general writing the Verilog grammar file since there was a steep learning curve to what a valid grammar file was supposed to look like.

While finding a BNF form of the Verilog language ended up being extremely helpful for the context of creating my project, it also added some difficulties for this project. In particular, because I found the syntax online, it was difficult for me to define my own created syntax for the user annotated post conditions. Because of this, by the project presentation deadline, I was unable to get the grammar file and ANTLR as a result to be able to parse through my user annotated post conditions. This then required the programmer to have to add the negation of the post condition into the program themself, which goes against the requirements I hoped to meet for this project. Unfortunately I was unable to get the grammar for my post conditions working correctly.

Conclusions:

One of the primary areas is to automate the negation of the annotated postcondition within my code. As mentioned previously, currently the programmer is responsible for converting their own ensures statement into Z3 syntax. This is extremely annoying for the programmer and goes against the original notion of requiring the programmer to have as little knowledge as possible. Due to issues mentioned previously in the Discussion section of this report, trying to achieve this in the limited amount of time of a semester has been difficult.

Other areas of potential improvement could be to expand the program verification capabilities to other aspects of the Verilog language. As mentioned before, currently UFM only supports bit-level combinational logic. Naturally, one potential expansion could be to support program verification for some of the more higher level abstraction implementations. So instead of having to convert an adder to the exact bits and operations done between bits, a programmer can simply verify a module that directly adds the variable a and b together like shown below.



```
module adder(a, b, out);
    input [1:0] a, b;
    output [1:0] cout;

    assign cout = a + b;

endmodule
```

This would allow for larger and more complicated designs to be verified as writing everything in terms of bits and gates is very limiting. As such, the applicability of this project could be expanded greatly by supporting more of Verilog's abstractions.

Another potential expansion could be to support Verilog's sequential logic. At the moment, only combination logic is supported so there is no notion of time or state supported by this design. However, almost all useful and complicated designs use sequential logic in order to produce more useful work. Finite state machines (FSMs) and registers are extremely important in order to "remember" values and perform differently depending on the current state that we are in.

As Verilog is a large language, expanding to cover more of the Verilog language is out of scope for a semester-long project and would take a considerable amount of time and effort. Indeed, even once these aspects of verification are included within this project, it will likely face several limitations mentioned previously with formal methods.

The project is a simple Verilog program verifier with ANTLR and Z3-based approach. While my current implementation of this project is capable of verifying simple bit-level logic, there are many ways of expanding this project. Overall, I learned a lot through this project especially in the field of programming languages and compilers.

Bibliography:

1. ANTLR Project. (2019). ANTLR (Version 4) [Software]. Available from http://www.antlr.org/
2. Amin Rezaei and Mohammad Reza Hashemi. (2021). Formal Verification of Hardware Trojans. arXiv preprint arXiv:2106.10392. [Online] Available at: https://arxiv.org/ftp/arxiv/papers/2106/2106.10392.pdf
3. Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. 2022. A Survey on Assertion-based Hardware Verification. ACM Comput. Surv. 1, 1 (January 2022), 33 pages. https://doi.org/10.1145/nnnnnnn.
4. Intel Corporation. (1994). Intel 1994 Annual Report. [Online] Available at: https://www.intel.com/content/www/us/en/history/history-1994-annual-report.html
5. Microsoft Research. (2022). Z3: A Theorem Prover from Microsoft Research (Version 4.8.12) [Software]. Available from https://github.com/Z3Prover/z3
6. William A. Krieger and Chana Nasamran "Efficient automated tapeout system", Proc. SPIE 4186, 20th Annual BACUS Symposium on Photomask Technology, (22 January 2001); https://doi.org/10.1117/12.410687