

UNI-corn

A hardware description language for the software programmer

[Introduction](#)

[Language Tutorial](#)

[Compiler Setup](#)

[Compiler File Structure](#)

[Using the Compiler](#)

[Running with flags](#)

[Your first UNI-corn program: half_adder.uni](#)

[UNI-corn program building blocks](#)

[Language Reference Manual](#)

[Program Structure](#)

[Data Types](#)

[Bus](#)

[Bus Creation](#)

[Bus Value Definition](#)

[Bus Assignment](#)

[Binary Expressions](#)

[Integer Expressions](#)

[Indexing](#)

[Indexing Modules](#)

[Register](#)

[Lexical Conventions](#)

[Keywords](#)

[Identifiers](#)

[Comments](#)

[Whitespace](#)

[Logical Operators](#)

[Syntactic Sugar for Logical Operators](#)

[Punctuation](#)

[Modules](#)

[Input](#)

[Output](#)

[Module Automation](#)

[Counted Loops](#)

[Generics](#)

[Scope](#)

[Standard Library](#)

[Print](#)

[Grammar](#)

[Project Plan](#)

[The Plan](#)

[Git Info](#)

[Language Evolution](#)

[The 'bus' Data Type](#)

[Registers](#)

[Input and Output](#)

[Logical Operators](#)

[Iterations of sequential circuit design](#)

[Translator Architecture](#)

[Test Plan](#)

[Unit Testing](#)

[Integration Testing](#)

[Test Automation](#)

[Team Responsibilities for Testing](#)

[Full Code Listing](#)

[Lessons Learned](#)

1. Introduction

Hardware description languages (HDLs) are powerful tools used by computer engineers and architects to abstract increasingly complex digital logic systems. Since the 1980's HDLs have allowed engineers to produce digital circuit designs in more efficient ways both to test outputs of large, interdependent designs and to visualize them using computer aided design (CAD). **UNI-corn** hopes to add to that body of work, focusing on the computer science student.

Many academic institutions offering bachelor's degrees in computer science often initiate students with an introduction to the field along with some focus on software programming. For many computer science students, especially those with little to no prior engineering experience, the study of computer architecture can be one of the more challenging aspects of a curriculum. Great tools such as the graphical digital design application—Logisim—and HDLs like System Verilog and VHDL exist to aid students in constructing the digital logic and microarchitectures learned in a typical introductory computer architecture class. However, one of the key challenges that a student might experience with learning something like Verilog or VHDL is the simple problem of unfamiliarity. With many introductory programming courses focusing on languages like Java, C++, and Python, learning the (admittedly necessarily) unique syntax of Verilog or VHDL can add to the mounting concepts for a student to master. UNI-corn aims to solve for that by using Java-like syntax. Emphasis on *aims*.

A key caveat to this experiment is that the topic of “how to make intrinsically challenging science, technology, engineering, and mathematics (STEM) concepts easier” is fraught with conflicting data and viewpoints. UNI-corn does not presume to solve for this issue. It's simply in a syntax with which a student who's taken a previous programming course may already be familiar.

That said, UNI-corn functions like other HDLs, allows programmers to build digital circuits to test their outputs on a given clock cycle. Using the basic building blocks of **logic gates**, UNI-corn can be used to build combinational and sequential logic designs such as different types of counters, adders, finite state machines, etc. These can be organized into modules that contain input(s) and output(s) to produce more complex designs.

2. Language Tutorial

2.1. Compiler Setup

The language compiler is developed in OCaml, which needs to be installed on the machine used to run the UNI-corn program. You can install OCaml through OPAM (OCaml Package Manager). The easiest way to install OCaml on most systems is through a package management system such as Homebrew, `apt-get`, or `dpkg`. On Homebrew, for example, you can run the following commands:

```
$ brew install ocaml
$ brew install opam
$ opam init
```

You'll also need to install the OCaml LLVM library, which can also be installed via OPAM.

```
$ opam install llvm.3.6
```

Here you must verify that the version of LLVM that OPAM installs is the same as the version that's installed via brew. You'll also need to take note of where brew places its executables. The PATH should match something like the one below, depending on your machine's setup:

```
$ /usr/local/Cellar/llvm/7.0.0/bin/llc
```

2.2. Compiler File Structure

The UNI-corn compiler comes in the `unicorn.tar.gz` tarball. Decompressing it gives the compiler directory structure below. The compiler files are each explained in Section 6 : Architectural Design. For now, it suffices to know two things:

1. Programs should be run from the top-level `/unicorn/` directory
2. There are test files under `/unicorn/tests` that can be used for testing

```

/unicorn/
|-- compiler/
|   |-- scanner.ml
|   |-- parser.mly
|   |-- ast.ml
|   |-- modfill.ml
|   |-- harden.ml
|   |-- semant.ml
|   |-- elaborate.ml
|   |-- indexing.ml
|   |-- noloop.ml
|   |-- simplelines.ml
|   |-- topsort.ml
|   |-- unic.ml
|   |-- codegen.ml
|-- tests/
|   |-- hello_world.uni
|   |-- comments/
|       |-- test1.uni
|       |
|   |-- buses/
|       |-- test1.uni
|       |
< other test cases continue >
|-- make.sh
|-- run.sh
|-- README.txt

```

2.3. Using the Compiler

UNI-corn files have a `.uni` extension as their file names. In this example, we will be compiling a file called `half_adder.uni`, assuming you're running this from the `/unicorn/` directory. The code snippet is below, just for reference. Do not worry about understanding the code right now. There is a run-through of the program in [Subsection 2.4](#) and there is an in-depth description of the program in Section 3. It suffices to know that the expected value to print out is the $101 \oplus 000 = 101$.

```

1  // Code for a half adder
2  halfAdder(a<3>, b<3>) {
3      sum = a xor b;
4      carry = a and b;
5
6      out: sum<3>, carry<3>;
7  }
8
9  main() {
10     print ha: halfAdder(100b, 001b)[0];
11     out:;
12 }
13

```

```

1  // example c file
2  void tick();
3
4  int main (){
5      tick();
6  }

```

1. First run a shell script to compile all the OCaml files that will then be used to compile UNI-corn:

```
$ ./make.sh
```

2. To compile the .uni file, run the run.sh shell script like so:

```
$ ./run.sh half_adder cfile.c
>> 101
```

Note that you should omit the file extensions when using this script

2.3.1. Running with flags

You can access information about each stage of the compilation process by running the following command, replacing `-flag` with one of the

options for the table below. Remember to take note of where the `unic.native` file is placed during compilation.

```
$ ./unic.native -flag filename.uni
```

- `-a` **ast:** prints the abstract syntax tree (AST) for the program
- `-m` **modfilled ast:** prints a AST with modules expanded
- `-h` **hardened ast:** AST with variables replaced by values
- `-s` **semantic ast:** semantically-checked AST (same as hast)
- `-n` **netlist:** prints netlist (i.e. unordered list of gates)
- `-sl` **netlist (simplified):** prints netlist with simplified code
- `-i` **indices:** prints netlist with expanded indices (i.e. values)
- `-n2` **netlist (more simplified):** netlist with simplified
- `-t` **netlist (topsorted):** topologically sorted netlist
- `-io` **netlist (IO):** topologically sorted netlist with IO values
- `-l` **llvm ir:** prints the LLVM IR code
- `-c` **default:** checks and prints the generated LLVM IR

2.4. Your first UNI-corn program: `half_adder.uni`

Let's revisit the small simple half adder program from above. A "Hello World" program is typically used to introduce the very basic components of a programming language. Since the purpose of an HDL is to create a programming abstraction of digital circuit logic, the small, simple half adder seems like an appropriate equivalent. Hopefully, as is one of the primary intents of UNI-corn, the syntax will look familiar to anyone who has coded in a program like Java or C++.

2.4.1. UNI-corn program building blocks

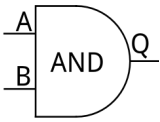
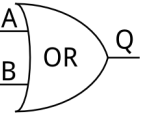
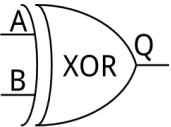
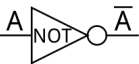
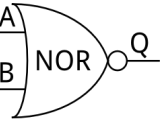
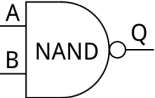
At the highest level, a UNI-corn program consists of the following key components. For the Java or C++ programmer, respective analogies have been drawn.

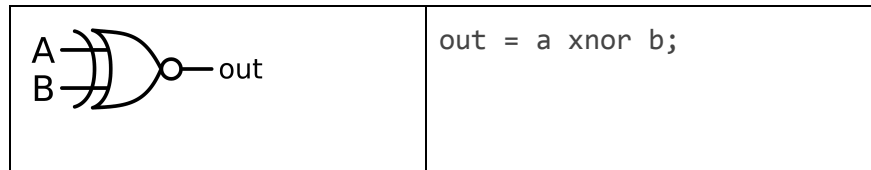
- **Comments, Expressions, and Blocks:** Comments are set the same way as in Java or C using two backslashes (`//`). Like these two languages, expressions end in semicolons.

- **Data Types:** As with the underpinnings of computer hardware—the very thing this language is used to describe—UNI-corn has but a single data type (get the name now?)—binary numbers. These are typed as any sequence of ones and zeros followed immediately by a b. The language of this type can be described by the following regular expression $(0 | 1)^*b$ and the following binary strings are in this language: **1b**, **0b**, **10b**, **11b**, **100b**, **101b**, **111b**, etc.

Integers do exist but their use is restricted as magic numbers for use in defining the size of a bus, dereferencing a bus, or to specify the iterations of a loop.

- **Operators:** The most intrinsic building blocks of digital circuit logic and sole operators behind UNI-corn are the digital logic gates below:

Gate	Syntax
	$q = a \text{ and } b;$
	$q = a \text{ or } b;$
	$q = a \text{ xor } b$
	$a\emptyset = \text{not } a;$
	$q = a \text{ nor } b;$
	$q = a \text{ nand } b;$



Any other gates and logic can be built using a combination of the gates above.

- **Bus:** A bus is the only kind of variable that can be created. Since there is only one data type, all bus variables are of type binary string. These can be set using the equals sign (=) to a binary string literal, another bus, or an expression that evaluates to a binary string. The names are alphanumeric (further naming rules can be found section X of the language reference manual below). In the half adder code snippet above, `a` and `b` are buses that are the inputs to the `halfAdder()` module and `sum` and `carry` are buses that are its outputs.

Because all buses are of type binary string, there's no need to specify the type like in other languages. However, you do need to specify the size of the bus when assigning a bus as input or output. This is done using angle brackets (`<int>`). More on this in section X of the language reference manual.

- **Modules:** These are akin to functions or methods. In digital circuit terms, this is a self-contained subcircuit, potentially with inputs and outputs, that can be reused throughout a larger circuit. In the half adder program above, a module named `halfAdder()` is defined in lines 1 through 7 and then invoked in line 10. This subcircuit has two inputs—`a` and `b`—that it then manipulates with logic gates to produce two outputs—`sum` and `carry`. The code of a module is enclosed in curly brackets.
 - **Inputs:** Inputs to a module are optional. Any inputs needed can be specified in the same way that you specify parameters to a function in Java or C. Like in these just-mentioned languages, the values of inputs are set during the module invocation.
 - **Outputs:** Outputs are **not** optional. Each and every module must have an output, even if that output is `null`. Outputs can be either 1 (which can `null`) or more. Outputs are

specified using the **out** keyword followed by a colon and then either nothing, a binary string, or a bus. In the half adder example, the `halfAdder()` module has two outputs and the `main()` module has one output—**null**—which is set by leaving the space between the colon and the end of the expression (the semicolon) empty.

- **main()**: A special kind of module that is required in the program. All other modules must be invoked here for them to be evaluated. The `main()` module also received ticks from the clock (more on this later).

The final structure of a module ends up looking something like:

```
name(in0<K>,...inL<K>) { expr0;...;exprN; out:M; }
```

where $K, M \in \mathbb{Z}^+$ and $N, L \in \mathbb{Z}$.

- **Print**: A special function used to print to stdout. This is one aspect of UNI-corn that has no Java or C equivalent and whose syntax is slightly more like a functional programming language. The function `print` is followed by a parameter (separated by a space) to which the value of the expression on the right hand side of the colon is assigned. `print` then outputs the value of the parameter to stdout. In the half adder program above, the parameter is printed is named `ha` and its value is the first output of the `halfAdder()` module (as denoted by the dereferencing operation `[0]`...more on this later).
- **neigh!**: Because the developers of UNI-corn are a silly bunch, all UNI-corn programs require a completely arbitrary end of file terminator (EOFT). This can either be a unicorn emoji (🦄) that comes with the latest Unicode standards or simple the keyword **neigh!**.

That's it! With perhaps the exception of dereferencing one or more of the outputs of a module, this quick run through should provide you with everything you need to understand and run the `halfAdder()` example or create a small program of your own. Again, a more in-depth rule of the aspects of a language can be found in the Language Reference Manual below.

3. Language Reference Manual

UNI-corn is a simple hardware description language. Using the basic building blocks of combinational logic, UNI-corn can be used to build more complex digital circuits such as different types of counters, adders, mealy state machines, and many other common circuits. As is customary with digital circuits, modules—a collection of one or more gates with input and output—lie at the core of UNI-corn. Furthermore, like with a digital circuit board, sequence of operations are not determined by the code's sequence in a text file but rather by the direction and flow of the circuit's buses (either within a module or from module to module). UNI-corn's sole data types are the rise (1) and the fall (0) of electricity flowing through each bus. As discussed further in the the next chapter, these binary values can be represented and introduced into your program via a couple of different forms. As an introduction, below is a simple program that will perform all the basic logic operations on a pair of buses with constant values. A file need only have the extension .uni and, because UNI-corn has no classes, every file must have a `main()` method (more on this later) wherein the modules can run based on the system's clock.

3.1. Program Structure

Program Structure Programs in UNI-corn consist of a `main()` block and modules. Modules define circuits that can be arbitrarily recreated within other circuits, given some input. The main block will simply contain calls upon a module (or set of modules) that the programmer wants to simulate. The compiler converts the main into a function called `tick()` that can be linked with .c files. Its inputs and outputs can be accessed through global variables.

3.2. Data Types

UNI-corn is peculiar in that it only has one data type, but it also has arbitrarily many data types, from a different perspective. These are buses. Because there is only one data type, identifiers do not need to be preceded by a type. UNI-corn is a statically inferred language.

3.3. Bus

A bus is conceptually similar to an array of booleans represented in binary, but is represented in code as a string. It is the UNI-corn representation of physical buses. A bus's length acts essentially as a data type; type-checking happens entirely on a bus-length basis. In this sense there are arbitrarily many data types (if we consider each bus-length a distinct type).

3.3.1. **Bus Creation**

There are two main ways of initializing buses: 1) passing a constant and, 2) passing values from other modules. Buses can also be initialized with a binary expression. Once a bus is assigned, it can't be reassigned. Bus names must contain only alphanumeric English characters or the single quote ' (no other special symbols like \$, #, !, _ etc.). They must begin with an English letter.

bus:

literal
identifier
expression

3.3.2. **Bus Value Definition**

Buses must have a value assigned at creation. The instantiation and assignment begins with the variable name and the assignment to the bus, followed by a lowercase letter 'b'. The bus size is automatically determined by the compiler. Note that leading 0's are factored into the bus's length.

3.3.3. **Bus Assignment**

Buses can have only one series of boolean values, but it has to be followed by the letter b. (i.e. 10001b would be a series of boolean values followed by the letter b)

3.3.4. **Binary Expressions**

All logical UNI-corn expressions are considered binary expressions and eventually evaluated to a binary literal, but can be made up of numerous logic gate operators and identifiers. All expressions are left-right associative and must exist only on the right side of a bus or register definition. NOTE: Assignments, module calls, print function calls, and loops are also considered expressions.

BinExpr:

literal
BinExpr BinaryOp BinExpr
UnaryOp BinExpr
(BinExpr)
identifier
BinExpr[IntExpr]
Assignments
ModuleCalls
Loops

UnaryOp:

not BinExpr
Not literal

3.3.5. Integer Expressions

Integer Expressions and integers exist in UNI-corn but are not allowed to be assigned. They are used solely for indexing and generics purposes. An integer expression can be a numerical value, an identifier, or an infix-style expression with either '+' or '-' as an operator between two operands.

Int Expr:

IntExpr + IntExpr
IntExpr - IntExpr
literal
identifier

3.3.6. Indexing

The square bracket '[']' is used to index a bus whose value is a series of binary values. For a n-bit bus (i.e. of length n) the 0th index will be the most significant bit and the n-1th index will be the least significant bit. The user can have infinitely many indices. The user can specify a range while indexing.

Index:

identifier[integer]
identifier[IntExpr]
identifier[IntExpr : IntExpr]

3.3.7. Indexing Modules

For modules that return multiple outputs, the value assigned to a bus must be one and only one of those outputs. This output is specified by square brackets [] enclosing the variable name of the output bus as defined in the module. If a particular bit in an output bus is desired, double brackets [][] may be used. In the case where there is only 1 output, brackets may be omitted.

3.4. Register

Registers are built-in types comprised of combinational logic. As such their internal behavior is like modules in that they have input and output buses. However, since they have persistent memory through each cycle, they need to be specified separately. Registers can be assigned a binary string, a bus variable, or an expression of arbitrary size. The binary string value of the initialized value can be of arbitrary size. A register is identified by the assignment symbol := which is followed by the bus name to which the register is assigned. Finally, the user must specify the initial state of the bus. In order to initialize the value of the register, the user must enclose the initial value (either 0 or 1) in between asterisks, *. Use := to initialize given a bus or constant value on the right of the operator.

register:

$$\text{BinExpr} := \text{BinExpr} * \text{BoolVal} *$$

3.5. Lexical Conventions

A UNI-corn program is broken down and parsed into tokens through lexical transformations. Tokens can be keywords, identifiers, operators, whitespace, assignment symbols, indexing symbols, punctuation, and various brackets. There are no constants in UNI-corn with the exception of boolean 1 or 0.

3.5.1. Keywords

In addition to all logic gate names (see 3.5.5), the following are reserved keywords and are not to be used as identifiers or otherwise:

from	print
out	main
for	to
make	neigh!

3.5.2. Identifiers

Identifiers are used to name variables and user-built modules. An identifier is a sequence of letters and digits that must start with a lowercase letter. Uppercase and lowercase letters are considered different characters. Identifiers can only be established during variable or module declarations. Identifiers of different types must have unique names(i.e.there cannot be a bus and a module with the same name).

assign statement:

```
identifier = BinExpr;  
identifier = BooleanLiteral;
```

3.5.3. Comments

Comments are characters delineated by a specific combination of symbols and contain characters that are not executed by the program. There are two types of comments, Single-line and Multi-line comments.

- **Single-line Comments** Single-line comments are introduced with // and everything following the double slashes is ignored by the compiler until a newline
- **Multi-line Comments** Multi-line comments are introduced with /** and terminated with */ and everything in

3.5.4. Whitespace

Blanks, tabs, and newlines are ignored, except to separate tokens

3.5.5. Logical Operators

Logical operators must evaluate to boolean expressions and return true or false in the form of '0' or '1'

- **and** is an operator which takes in two single-bit buses, inclusive of the index of an n-bit bus, and performs the boolean logical \wedge operator on them.
- **or** is an operator which takes in two single-bit buses, inclusive of the index of an n-bit bus, and performs the boolean logical \vee operator on them.

- **nor** is an operator which takes in two single-bit buses, inclusive of the index of an n-bit bus, and performs the boolean logical \downarrow operator on them.
- **not** is an operator which takes in a single bit and negates it. You can also index a bus to get a single bit then negate it. The **not** operator comes before the variable name.
- **nand** is an operator which takes in two single-bit buses, inclusive of the index of an n-bit bus, and performs the boolean logical \uparrow operator on them.
- **xor** is an operator which takes in two single-bit buses, inclusive of the index of an n-bit bus, and performs the boolean logical \oplus operator on them.
- **xnor** is an operator which takes in two single-bit buses, inclusive of the index of an n-bit bus, and performs the boolean logical $\neg\oplus$ operator on them.

3.6. Syntactic Sugar for Logical Operators

<code>*=</code>	<code>and</code>	<code>a *= b;</code>
<code>+=</code>	<code>or</code>	<code>a += b;</code>
<code>#=</code>	<code>xor</code>	<code>a #= b;</code>
<code>!+</code>	<code>nor</code>	<code>a !+ b;</code>
<code>!*</code>	<code>nand</code>	<code>a !* b;</code>
<code>!#</code>	<code>xnor</code>	<code>a *# b;</code>
<code>!</code>	<code>not</code>	<code>! a;</code>

3.7. Punctuation

UNI-corn code contains three forms of punctuation, the semicolon, the comma, and and EOF terminator.

- **Semicolon:** In UNI-Corn the semicolon is used to denote statement separation. As a statement separator, the semicolon is used to demarcate boundaries between two separate statements. NOTE: putting a semicolon

on a separate line does nothing. The compiler will discard a semicolon that is placed on a line by itself without any statement before it.

- **Comma:** The comma is used to denote assignment to multiple variables.
- **EOF:** The unicorn emoji is a file terminator that denotes the end of the program. The user can also type in the word "neigh!" instead of placing the unicorn emoji at the end of the file in order to denote the end of the program.

3.8. Modules

Modules are named functions that are defined by the user. They may accept a number of parameters, perform a logical operation, and designate a number of outputs using the **out** keyword. Modules can be "called" as long as the call contains the same number of arguments as parameters in its definition.

module declarations:

identifier (params) { statement list }

params:

/ nothing */*

identifier

identifier<IntExpr>

module calls:

identifier (actuals list)

actuals list:

/ nothing */*

Bus

BoolVal

3.8.1. Input

Modules accept only bus types as parameters or Boolean values. Modules should not mutate their input. Parameters must use angled brackets containing the size of the bus.

3.8.2. **Output**

The keyword **out** is utilized in order to retrieve the most recent call to an instantiated module. There can be multiple outputs from a module. Every module, even main, need to have an out statement at the end even if it's not outputting anything. Parameters specified as output also must specify size using angled brackets.

3.9. **Module Automation**

3.9.1. **Counted Loops**

In order to avoid the tedium of copy-paste, modules may contain loops. NOTE: The user must place a semicolon after the for loop's body. Loops take the form of the keyword **for** followed by parens () containing the iterator identifier, the optional starting keyword **from** followed by its value, and the keyword **to** followed by its value. In general, loops execute the contained code, starting with initializing the variable (typically, i) to the from value (defaults to 0 if the from is omitted) creating the buses/modules listed within the block of the loop in each step.

3.9.2. **Generics**

Generics are used in the parameter declarations when defining a module. They are effectively used as placeholders to allow a buses size to be defined later.

3.10. **Scope**

A module only has access to its inputs and buses declared within its brackets. The only exception is that the main() method has access to everything within that file. Any module can call any other module (as long as there is no recursive definition), but a module can't access another module's buses, other than its output.

3.11. **Standard Library**

The standard library contains modules that are included in UNicorn as built-in modules. Currently the only built-in module is print.

3.11.1. **Print**

The print module will make use of the reserved word, print, and will print the current value of a bus supplied as argument. Following the **print** keyword, the user defines a variable name to which the printed item is assigned. One print

statement is used per value. To print multiple values, you must use multiple print statements.

3.12. Grammar

Program:

module calls

module calls:

identifier (actuals list)

actuals list:

/ nothing */*

Bus

BoolVal

module declarations:

identifier (params) { statement list }

params:

/ nothing */*

identifier

identifier<IntExpr>

variable declarations:

identifier = BinExpr;

assign statement:

identifier = BinExpr;

identifier = BooleanLiteral;

Register:

*BinExpr := BinExpr * boolVal **

Index:

identifier[integer]

identifier[IntExpr]

identifier[IntExpr : IntExpr]

Int Expr:

IntExpr + IntExpr
IntExpr - IntExpr
literal
identifier

BinExpr:

literal
BinExpr BinaryOp BinExpr
UnaryOp BinExpr
(BinExpr)
identifier
BinExpr[IntExpr]
Assignments
module calls
loops

UnaryOp:

not BinExpr
not literal

bus:

literal
identifier
BinExpr

literal:

stringBoolLiteral
integer

4. Project Plan

4.1. The Plan

Below is the what, who, and when of how the UNI-corn compiler go developed. Descriptions of what each column means as follows:

Deliverable: Mainly the actual code file that would enable a feature under the Features column.

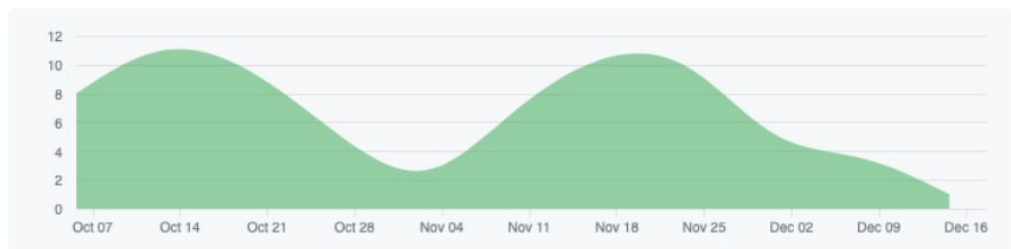
Lead: Architect and director for how to code and structure that file. Contributed significantly to that file and thus that feature set.

Contributor: Contributed substantially to the codebase for that file, under direction of the Lead. Assisted Lead in testing, researching solutions to a coding problem, etc.

Collaborator: Was physically (or live virtually) present during the coding of the file. Contributed to the thought process or administrative tasks required to deliver.

4.2. Git Info

Commit History Highlights



Lalo: 84 | Gael: 47 | Maryam: 42 | Dan: 12 | Adiza: 12

4. Language Evolution

UNI-corn was originally conceived as an alternative for the programmatically-inclined to better grasp the concepts of electrical engineering with a focus on creating simulations of circuits using a language designed to emulate Java syntax. Since Java is a relatively beginner friendly programming language and is well-known, this would allow UNI-corn to be an ideal entry point for electrical engineering for those who already have some programming background and learn best using a programmatic approach. Visualization tools for digital circuit design can be cumbersome and require users to learn a new application. With UNI-corn, was designed to resemble popular language syntax while remaining clear and powerful enough to simulate a large number of combinational and sequential logic circuits.

As we set out to materialize our vision of UNI-corn, we began by conceptualizing what our syntax would look like. The first iteration of UNI-corn syntax was roughly based on the Logisim visualization tool (a familiar tool to our team). Because this is a hardware descriptive language, we began by creating simple sequential circuits and translating them into C code. Once they were in C, we adapted and stripped the syntax to best fit with the vocabulary of circuit design. UNI-corn was originally overly verbose with lots of keywords in an attempt to help programmers new to circuit design better understand the details. Currently, UNI-corn features a streamlined syntax, a small set of keywords, and loosely resembles the familiar structure of an Object-Oriented language. At its core, UNI-corn marries the simplicity of a ‘building block’ system (consisting of module definitions, module calls, scoping rules, and a main function) with the non-sequential, input/output control flow of circuit design visualization software.

4.1. The ‘bus’ Data Type

In addition to integer and boolean data types, we included a special data type called ‘wire’ which was an array of boolean values. This ‘wire’ data type went through many iterations, but the boolean array(now known as ‘bus’) remains as one of the core details of UNI-corn. As our team progressed and better understood the project, we decided that the user didn’t need to define integer data types and single boolean values could be rolled into our ‘bus’ data type. In its current state, UNI-corn contains only the bus data type. While integers exist in UNI-corn code, they are used sparingly for indexing and generics and are not valid variable values.

4.2. Registers

Registers existed in the earliest iterations of UNI-corn however, the syntax was drastically different. Faced with challenges in the Scanner and Parser regarding

left-right associativity, register syntax was altered from a right-associate to a more familiar and simple left-associative design.

4.3. Input and Output

Inputs and output lines were designed to closely resemble to vocabulary and conceptuality of wires acting as inputs and outputs to logic gates and circuitry in general. Inputs were originally expressed as an explicit line of code within a module, but have since been moved to match Java's syntax of parameter variables defined in the function definition. Therefore, module inputs are defined when the module is defined. Outputs remained functionally the same throughout the design process. Additionally, an output line is required in the main module in order to compile. This is because the main module is treated like any other module except that it is required for a valid UNI-corn file.

4.4. Logical Operators

Logic gates were a simple implementation with the only significant evolution being a transition from prefix to infix notation.

4.5. Iterations of sequential circuit design

Version 1: Overly verbose syntax, prefix expressions, and module blocks

```
module seqCirc() {  
  
    //Variable A of type register with two inputs (input, state)  
    wire a = val X;  
    wire b = val Y;  
    wire c = XOR(a,b);  
    wire d = NOT(b);  
  
    register init0 X := c;  
    register init1 Y := d;  
  
    out:c;  
    out:d;  
  
}  
  
Main() {  
  
    seqCirc();  
  
}
```

```
wire z = seqCirc()[c];
```

Version 2: removal of data type and module initialization keywords, use of module parameters as circuit inputs, generics, indexing, integer expressions in indexing, loops, infix logical expressions

```
main(a[5], b[5]) {
    rippleAdder(a, b);
}

fullAdder(a[0], b[0], carryIn[0]){
    x = a xor b;
    y = a and b;
    z = carryIn and x;
    sum = x xor carryIn;
    carryOut = y or z;
}

rippleAdder(a<N>, b<N>){
    zero = 0;
    sum[0] = fullAdder(a[0], b[0], zero)[sum];

    for (i from 1 to N){
        lastCarry[i] = fullAdder
            (a[i-1], b[i-1], lastCarry[i-1])[carryOut];
        sum[i] = fullAdder(a[i], b[i], lastCarry[i])[sum];
    }
}
```


MVP: use of carryIn parameter tick function, standard library print function from module output, bus variables now ending with 'b', nested binary expressions, integer expressions in for loop.

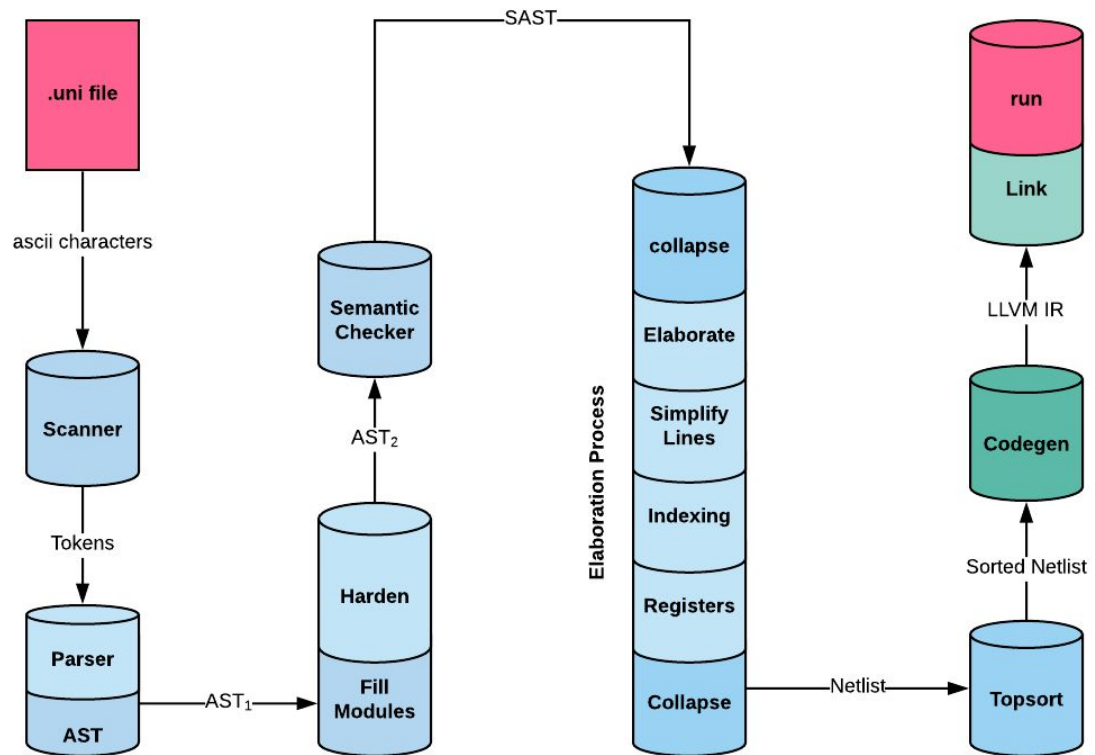
```
main() {
    a = 1011010b;
    b = 0011101b;
    m = modA(a,b)[sum];
    print m: m;
    out;;
}

fA(a, b, carryIn){
    axb = a xor b;
    sum = axb xor carryIn;
    carry = (axb and carryIn) or (a and b);
    out: sum, carry;
}

modA(a<n>,b<n>){
    c[0] = 0b;

    for (i from 0 to n-1){
        sum[i] = fA(a[i],b[i],c[i])[sum];
        c[i+1] = fA(a[i],b[i],c[i])[carry];
    };
    out:sum<n>;
}
```

5. Translator Architecture



6. Test Plan

There were a series of unit tests suites and integration test suites written for the UNI-corn compiler. Following the implementation of each compiler component (e.g. harden.ml, codgen.ml etc) all previous and latest test scripts were ran to ensure the most recent compilation did not interfere with a previous functionality of the language.

1) The test suite is all under one directory called testCases

a) Every subdirectory in ./testsCases/

- i) checks that every aspect of the Uni-Corn language reference manual produces the expected output. Each subdirectory is named after the feature it's testing. There are test cases in there that contain invalid programs which should report errors.

b) There is a programs folder in ./testCases called programs:

- i) This has a bunch of programs like alu.uni and genAdder2.uni which are programs that are fully written in Uni-Corn.

6.1. Unit Testing

As the compiler was developed, tests were written for every new feature or component to verify its functionality. For each feature, depending on the number of equivalence classes, one to two unit tests were written and added to the appropriate subdirectory within the test directory.

6.2. Integration Testing

The integration tests were slowly added over the development timeline. Within these tests, most bugs within the compiler were uncovered. Some, bug that showed up during integration testing displayed a fundamental problem within the compiler implementation and therefore evoked restructuring.

6.3. Test Automation

The testall.sh script in the unicorn directory compiles, runs and links all the files within the test directory. The results of the testall script are then pretty printed with results showing if the test either failed or passed. Those intended to pass are denoted with the word "pass" and those intended to fail are marked with a "fail."