UNI-corn
language reference manual

Gael Zendejas (gz2255) - Manager,
Dan Sendik (drs2176) - Language Guru,
David Lalo (djl2178) - System Architect,
Adiza Awwal (asa2201) - System Architect,
Maryam Aly (mya2114) - Tester

October 15, 2018

Contents

1 Introduction

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, etc. 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 wires (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 wire. 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 wires 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.

```
// Operators module
operators {
    a = 1;
    b = 0;

    c = a and b;
    d = b or c;
    e = not d;

    out e;
}

main() {
    f = operators;
}


```

1.1 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.

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 wires.

2.1 Wire

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

2.1.1 Wire Creation

There are two main ways of initializing wires: 1) passing a constant and, 2) passing values from other modules. Once a wire is assigned, it cannot be reassigned. There is one exception: wires can be reassigned within the `main()` method. Wire names must begin with a lowercase English letter character ('a'-'z'), and must contain only alphanumeric English characters (no special symbols like $, #, !, etc.)

2.1.2 Wire value definition

Wire assignment works as follows:

```
wireA = 0110101
```

This instantiation and assignment begins with the variable name (here “wireA”) and the assignment to the wire. The wire size is automatically determined by the compiler.Note that leading 0’s are factored into the wire’s length.

2.1.3 Wire assignment

Wires can only be the output of other modules, or a constant value; wires cannot be assigned to other wires. For example, in a module with a single input *a* and single output *b*:

```
    x = 1;
    y = x and x;
    x = y and a; //error. Can’t reuse names in a module
    b = y; //error. Can’t "rename" wires
```

This is different from variable assignment in most imperative languages. At the level of the compiler, wire assignments are actually creating links between instances of modules. Conceptually, assignment attaches a wire to a module’s output. In practice, the wire is continuously receiving output values from a module.

When a wire is assigned a module’s output, its length is automatically determined by the compiler once the module resolves its own output length. Keep this in mind when using wires in modules with certain expected values.

2.1.4 Indexing

The square bracket ‘[ ]’ is used to index a wire whose value is a binary array. For a n-bit wire (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.

```
    a = 01011;
    b = 10111;
    a[2] //will output 0
    b[0] //will output 1
```

2.1.5 Indexing Modules

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

```
    /**
    Let this be a module with 2 8-bit inputs, A and B.
    Let fullAdder have two outputs: an 8-bit wire called val,
    and a 1-bit wire called carry
    **/
    x = fullAdder(A,B)[carry]//x is of length 1
    z = fullAdder(A,B)[val]//z is of length 8
    y = fullAdder(A,B)[val][0]//y is of length 1.
    //It is exactly the most significant bit z.
    a = and(A[0], B[0])//No brackets needed here
```

2.2 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 wires. However, since they have persistent memory through each cycle, they need to be specified separately.

A register is identified by the assignment symbol := which is followed by the user-defined value or variable name to which the register is assigned. Finally, they keywords `init0` or `init1` follow the assignment in order to specify the initial state (either 0 or 1 respectively) of the register. Registers must be initialized simultaneously with being declared(i.e. if you are declaring a register, you must use the init keyword). Use `:=` to initialize given a wire or constant value on the right of the operator.

```
    /**
    Declaring a variable of type register,
    initializing the register with value 0,
    and assigning the input to the value of wireB
    **/
    registerA := wireB init 0;
```

To retrieve the stored value of a register, use the keyword `val` before the register variable name.

```
    //Declaring a register and retrieving its value
    ff := wireB init 1;
    val ff; //outputs 1;
    ff1 = val ff;
```

3 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.

3.1 Keywords

In addition to all logic gate names (see 3.5), the following are reserved keywords and are not to be used as identifiers or otherwise:

from	init
out	main
for	to
print	

3.2 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 wire and a module with the same name).

3.3 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.

3.3.1 Single-line Comments

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

3.3.2 Multi-line Comments

Multi-line comments are introduced with /** and terminated with **/ and everything in between them is ignored by the compiler. There is no nesting of comments.

3.4 Whitespace

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

3.5 Operators

3.5.1 and

`and` is an operator which takes in two single-bit wires, inclusive of the index of an n-bit wire, and performs the boolean logical $\wedge$ operator on them.

```
    a = 1;
    b = 0;
    c = a and b; //operator and takes in two wires
    d = a[i] and b[i]; //operator and takes in two bits

    //note:
    x = 1;
    y = 0;

    x and y; //will output 0
    y and x; //will output 0
    y and y; //will output 0
    x and x; //will output 1
```

3.5.2 or

`or` is an operator which takes in two single-bit wires, inclusive of the index of an n-bit wire, and performs the boolean logical $\vee$ operator on them.

```
     a = 1;
     b = 0;

    a or b;  //operator or takes in two wires
    a[i] or b[i]; //operator or takes in two bits

    //note:
    x = 1;
    y = 0;

    x or y; //will output 1
    y or x; //will output 1
    y or y; //will output 0
    x or x; //will output 1
```

3.5.3 nor

`nor` is an operator which takes in two single-bit wires, inclusive of the index of an n-bit wire, and performs the boolean logical $\downarrow$ operator on them.

```
      a = 1;
      b = 0;

    a nor b;  //operator nor takes in two wires
    a[i] nor b[i]; //operator nor takes in two bits

    //note:
     x = 1;
     y = 0;

    x nor y; //will output 0
    y nor x; //will output 0
    y nor y; //will output 1
    x nor x; //will output 0
```

3.5.4 not

`not` is an operator which takes in a single bit and negates it. You can also index a wire to get a single bit then negate it. The `not` keyword comes before the variable name

```
      a = 1;

    not a;  //operator not takes in a wire data type
    not a[i]; //operator not takes in the index of a wire

    //note:
     x = 1;
     y = 0;

    not x; //will output 0
    not y; //will output 1
```

3.5.5 nand

`nand` is an operator which takes in two single-bit wires, inclusive of the index of an n-bit wire, and performs the boolean logical $\uparrow$ operator on them.

```
      a = 1;
      b = 0;

    a nand b;  //operator nand takes in two wires
    a[i] nand b[i]; //operator nand takes in two bits

    //note:
     x = 1;
     y = 0;

    x nand y; //will output 1
    y nand x; //will output 1
    y nand y; //will output 1
    x nand x; //will output 0
```

3.5.6 xor

`xor` is an operator which takes in two single-bit wires, inclusive of the index of an n-bit wire, and performs the boolean logical $\oplus$ operator on them.

```
      a = 1;
      b = 0;

    a xor b;  //operator xor takes in two wires
    a[i] xor b[i]; //operator xor takes in two bits

    //note:
     x = 1;
     y = 0;

    x xor y; //will output 1
    y xor x; //will output 1
    y xor y; //will output 0
    x xor x; //will output 0
```

3.5.7 xnor

`xnor` is an operator which takes in two single-bit wires, inclusive of the index of an n-bit wire, and performs the boolean logical $\neg\oplus$ operator on them.

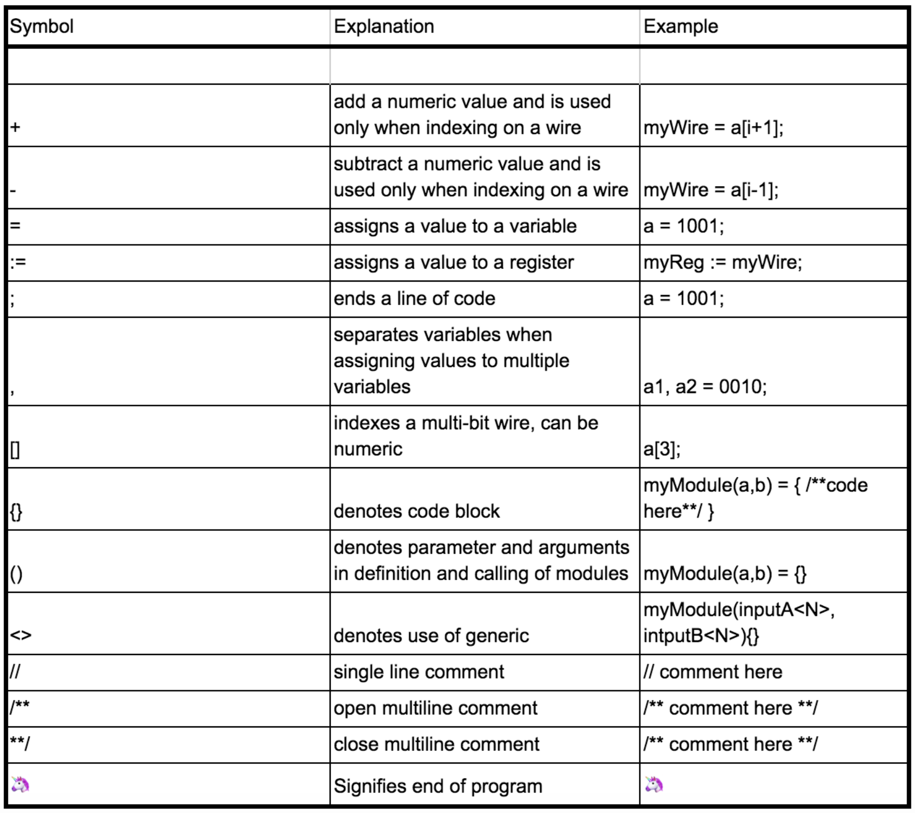
```
  a = 1;
  b = 0;

a xnor b;  //operator xnor takes in two wires
a[i] xnor b[i]; //operator xnor takes in two bits

//note:
 x = 1;
 y = 0;

x xnor y; //will output 0
y xnor x; //will output 0
y xnor y; //will output 1
x xnor x; //will output 1
```

3.6 Relational and Equality Operators



Symbol	Explanation	Example
+	add a numeric value and is used only when indexing on a wire	myWire = a[i+1];
-	subtract a numeric value and is used only when indexing on a wire	myWire = a[i-1];
=	assigns a value to a variable	a = 1001;
:=	assigns a value to a register	myReg := myWire;
;	ends a line of code	a = 1001;
,	separates variables when assigning values to multiple variables	a1, a2 = 0010;
[]	indexes a multi-bit wire, can be numeric	a[3];
{}	denotes code block	myModule(a,b) = { /**code here**/ }
()	denotes parameter and arguments in definition and calling of modules	myModule(a,b) = {}
<>	denotes use of generic	myModule(inputA<N>, intputB<N>){}
//	single line comment	// comment here
/**	open multiline comment	/** comment here **/
**/	close multiline comment	/** comment here **/
🦄	Signifies end of program	🦄

3.7 Punctuation

```
 x = 10101; // current statement finishes here
 y = 01010; // next statement begins & ends
// end of program
```

3.7.1 Semi-colon

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.

3.7.2 Comma

The comma is used to denote assignment to multiple variables.

```
    a1, a2 = MODULEX(b,c);
     /**
     both wire a1 and a2 are assigned to
     the values b and c of MODULEX
     **/
```

3.7.3 UNI-corn Emoji

The unicorn emoji is a file terminator that denotes the end of the program.

4 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.

```
// type x, y is wire
myModule(x,y){
    a = 1011;
    b = 1111;
    out c = a and b; //Designates c as the output of myModule
}
```

4.1 Input

Modules accept only wire types as parameters or Boolean values. Modules should not mutate their input.

4.2 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

5 Module Automation

5.1 Counted Loops

In order to avoid the tedium of copy-paste, modules may contain loops.

```
    for (i from 0 to 8){
         a[i] = inputA[i] and inputB[i];
    }
```

Loops take the form of the keyword `for` followed by parens `()` containing the iterator identifier (here `i`), the optional starting keyword `from` and its value (here 0), and the keyword `to` followed by its value. In general, loops execute the contained code, starting with initializing the variable `i` to the `from` value (defaults to 0 if the from is omitted) creating the wires/modules listed within the block of the loop in each step.

5.2 Generics

```
doThis(inputA<N>, intputB<N>){
    for (i from 0 to N){
         out a[i] = inputA[i] and inputB[i];
    }
}
```

6 Scope

A module only has access to its inputs and wires 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 wires, other than its output.

```
1       //example
2
3        test(x,y){
4        a = 1011;
5        b = 1111;
6        c = a and b;
7       //the scope of wire a, b, and c is only within this module
8       }
9
10      main(){
11       x = 11001;
12      //wire x, a, b, and c (on lines 4-6) are visible in main()
13      }
14
```

7 Standard Library

The standard library contains a number of modules that are included in UNI-corn as built-in modules.

7.1 Built-in Modules

Built-in modules may be called within any UNI-corn program. The current built-in modules are print, concatwires, adder and split.

7.1.1 print

The print module will make use of the reserved word, `print`, and will print the current value of a wire supplied as argument.

```
    a = 0001;
    b = 0010;
    print(a);
    //print the value of a, which is 1, to console
    print(b);
    //prints 2 to the console
    print(b[3]);
    //print the value of the 4th bit of b, 0, to the console.

```

7.1.2 concatwires

The concatwires module is a function that will combine two wires. The concatwires module takes in two wires as arguments and returns a single wire that is a concatenation of the two supplied arguments. The first wire's rightmost bit is concatenated with the second wire's leftmost bit. The resulting wire's size is the sum of the two wires' original sizes.

```
    a = 0010;
    b = 1111;
    c = concatwires(a,b);
    /**
    wire c now holds the value of wire a concatenated
    with wire b, 00101111
    **/

```

8 Sample Code

8.1 Combinatorial logic circuit

```
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;
    out sum = x xor carryIn;
    out carryOut = y or z;
}

 rippleAdder(a<N>, b<N>){
    zero = 0;
    out 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];
    }

}


```

8.2 Sequential logic circuit

```

// Toggle circuit

seqCirc {
  out c = a xor b;
  out d = not b;
  a := c init 0;
  b := d init 1;
}

main() {
  x, y = seqCirc;
}

```