UNI-corn: The language for abstract circuit logic

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

September 19, 2018

Abstract

Construct a language that allows users to abstractly create circuit logic without visualization.

1 Introduction

You are probably sitting in your Fundamentals class right now thinking to yourself, "What did I get myself into?". While you sit impatiently, your professor pulls up the the circuit representation of a bit adder and you immediately find yourself cluelessly staring at it. You try to understand what all these wires are doing and what these weird gates represent. Eventually you get the hang of it, but you still get frustrated when you hear the word "Logisim".

UNI-corn is circuit logic from a programmer's perspective. Inspired by great hardware descriptive languages like Lava and Virgil, but with syntax more familiar to software developers. Boolean logic and binary manipulation are at the core of UNI-corn. UNI-corn's relatively simple syntax and ease of translation from circuit diagrams into executable virtual circuits make it an excellent tool for hardware engineers and students.

2 Use Cases and Purpose

UNI-corn is ideal for creating programs that simulate circuit logic. Programmers or students will be able to create algorithms for hardware-level arithmetic operations such as adders (full, carry-lookahead, ripple), multiplexers, and registers. These algorithms could then be used to build larger components like a multiplier using arithmetic or a ROM using registers. Additionally, programs in the language would lend themselves well to bit-level manipulation of numbers using unary operations.

3 Data Types and Primitives

bool:

- Boolean value, zero or one

int:

- Numerical value between 0 and 64

wire:

- Contains an array of only zeros and ones
- Number of bits in a wire is determined at the time of creation.
- Number of bits is denoted between a set of square brackets immediately following the variable name(e.g. wire foo[8] = 00000101)

4 Operators

4.1 Built-In Functions and Gates

AND:

- Output is 1 only if all inputs are 1; accepts two arguments
  e.g. AND(0,1) = 0

OR:

- Output is 1 if either input is 1; output is 0 only if both inputs are 0; accepts two arguments
  e.g. OR(0,1) = 1

NOT:

- Output is 1 if input is 0; output is 0 if input is 1; accepts one argument
  e.g. NOT(1) = 0

NOR (negates OR):

- Output is 1 only if both inputs are 0; accepts two arguments

NAND (negates AND):

- Output is 1 only if all inputs are NOT 1; output is only 0 when all inputs are 1; output is opposite of AND gate; accepts two arguments

XOR:

- Output is 1 if only of the inputs is 1; output is 0 when both inputs are 1 or when both inputs are 0; accepts two arguments

XNOR:

- Output is only 1 if both inputs are 0 or both inputs are 1; output is 0 when either of the inputs is 1; accepts two arguments

$\langle\langle$ (Left bitshift operator):

- Takes a binary array that will be shifted as argument 1 and an integer to denote the number of bits to shift as argument 2; returns a binary array of the same length as the original; leftmost bit value is shifted out and discarded. The new bit that is shifted into the rightmost position is always a zero. e.g. $\langle\langle$ (00000101, 1) this is shifting the leftmost bit of the binary array by 1 bit and placing it in the rightmost position; the result is 00001010

$\rangle\rangle$ (Right bitshift operator):

- Takes a binary array that will be shifted as argument 1 and an integer to denote the number of bits to shift as argument 2; returns a binary array of the same length as the original; the array on the right side of the operator is the number of bitshifts to perform; rightmost bit value is removed and placed as the leftmost bit e.g. $\rangle\rangle$ (00001110,1) this is shifting the rightmost bit of the binary array by 1 bit and placing it in the leftmost position; the result is the result is 00000111

5 Nice To Have: Additional features and functionalities

This ***Nice to Have*** section highlights the features that the team would like to implement after approval from Professor Edwards and with the permission of time. However, they are not essential.

Syntactic Sugar (aka UNIcorn dust): Transform:

- $A * B$ into $AND(A, B)$
- $A + B$ into $OR(A, B)$
- ˜A into $NOT(A)$

Probes:

- In gate: Returns value of some wire—a way of specifying which to print when.
- Register: Returns value stored in register.

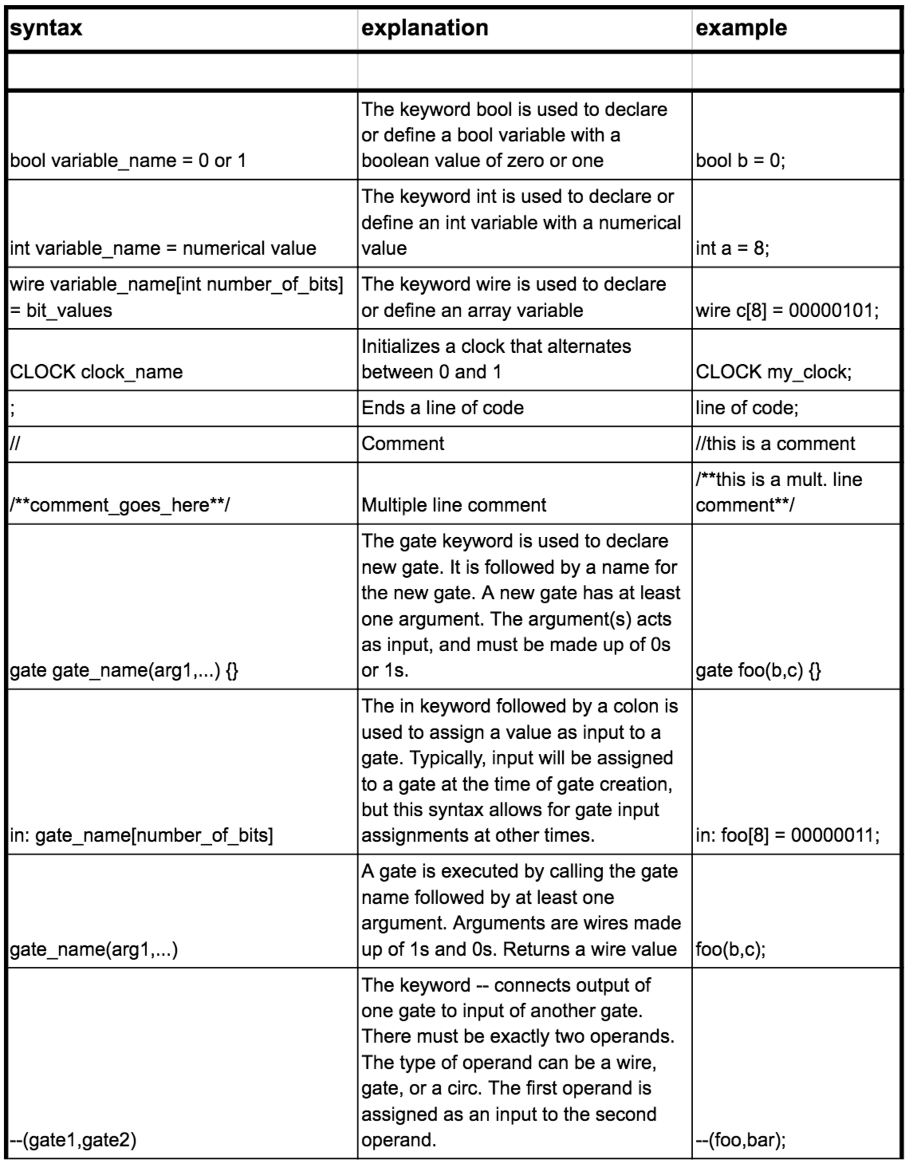
Generalizing:

- Generic sized arguments (i.e. gates with n-bit inputs), depending on generalized loops.
- Gate overloading.
- Implementing built-in gates for generic input size.

Consolidation:

- Concatenating wires into larger wires.

6 Syntax Table



syntax	explanation	example
bool variable_name = 0 or 1	The keyword bool is used to declare or define a bool variable with a boolean value of zero or one	bool b = 0;
int variable_name = numerical value	The keyword int is used to declare or define an int variable with a numerical value	int a = 8;
wire variable_name[int number_of_bits] = bit_values	The keyword wire is used to declare or define an array variable	wire c[8] = 00000101;
CLOCK clock_name	Initializes a clock that alternates between 0 and 1	CLOCK my_clock;
;	Ends a line of code	line of code;
//	Comment	//this is a comment
/**comment_goes_here**/	Multiple line comment	/**this is a mult. line comment**/
gate gate_name(arg1,...) {}	The gate keyword is used to declare new gate. It is followed by a name for the new gate. A new gate has at least one argument. The argument(s) acts as input, and must be made up of 0s or 1s.	gate foo(b,c) {}
in: gate_name[number_of_bits]	The in keyword followed by a colon is used to assign a value as input to a gate. Typically, input will be assigned to a gate at the time of gate creation, but this syntax allows for gate input assignments at other times.	in: foo[8] = 00000011;
gate_name(arg1,...)	A gate is executed by calling the gate name followed by at least one argument. Arguments are wires made up of 1s and 0s. Returns a wire value	foo(b,c);
--(gate1,gate2)	The keyword -- connects output of one gate to input of another gate. There must be exactly two operands. The type of operand can be a wire, gate, or a circ. The first operand is assigned as an input to the second operand.	--(foo,bar);

7 Code Sample

This program creates an algorithm within the hardware. While the adders are hardcoded, the multiplication happens iteratively. In this case, we multiply 2 times 6, by sequentially repeating addition of 2, 6 times. But notice that there are not physically 6 adders (only 1). Instead, this program uses 2 and 6 as variable inputs. It is essentially a really long-winded, hardware way of writing: for (i=0; i<6; i++) sum+= 2; print(sum);

```
Multiplier (
       wire a = 0010
       wire b = 0110
)
//n-bit multiplier. Actually running this would require a generic Adder as well,
//which is not defined in this program. So in reality 4-bit Multiplier is called by overloading.
gate Multiplier (wire a[n], wire b[n])

       wire time = Counter()[out];

       wire sum[n] = Adder(a, sum)[out];
       BeqC = EQ(b, time)[out];
       out = Register(sum, BeqC);

end (out[n])


//4-bit multiplier. Takes in a and b and returns their product.
gate Multiplier (a[4], b[4])

       wire time = Counter()[out];

       wire sum[4] = Adder(a, sum)[out];
       BeqC = EQ(b, time)[out];
       out = Register(sum, BeqC);

end (out[4])


//8-bit multiplier. Takes in a and b and returns their product.
gate Multiplier (wire a[8], wire b[8])

       wire time = Counter()[out];

       wire sum[8] = Adder(a, sum)[out];
       BeqC = EQ(b, time)[out];
       out = Register(sum, BeqC)[out];

end (out[8])


//8-bit Adder. Takes in a and b and returns their product. (To use 4-bit or n-bit adder, we'd
have to generalize, which are called by their respective multipliers, we'd have to write those
too).
gate Adder (a[8], b[8])

       out[0] = HalfAdder(a[0], b[0])[out];
              c1 = HalfAdder(a[0], b[0])[carry];

       s1 = HalfAdder(a[1], b[1])[out];
       out[1] = XOR(c1, s1);
              or1 = OR(a[1], b[1]);
              and1 = AND(c1, or1);
              pc1 = HalfAdder(a[1], b[1])[carry];
              c2 = OR(pc1, and1);

       s2 = HalfAdder(a[2], b[2])[out];
       out[2] = XOR(c2, s2);
              or2 = OR(a[2], b[2]);
              and2 = AND(c2, or2);
              pc2 = HalfAdder(a[2], b[2])[carry];
              c3 = OR(pc2, and2);
```

```
                //etc, etc

        s6 = HalfAdder(a[6], b[6])[out];
        out[6] = XOR(c6, s6);
                or6 = OR(a[6], b[6]);
                and6 = AND(c6, or6);
                pc6 = HalfAdder(a[6], b[6])[carry];
                c7 = OR(pc6, and6);

        s7 = XOR(a[7], b[7])[out];
        out[7] = XOR(c7, s7);

end (out[8])


//Half-adder.
gate HalfAdder (a, b)

        out = XOR(a, b);
        carry = AND(a, b);

end (out, carry)

//Counts the current cycle.
gate Counter ()

        const wire one[8] = 00000001;
        wire reg[8] = Register(regUp, CLOCK);
        wire regUp[8] = Adder(one, reg);
        carry = reg;

end (out)


//Returns 1 if a == b. 0 otherwise. 8-bit
gate EQ (a[8], b[8])

        e0 = XNOR(a[0], b[0]);
        e1 = XNOR(a[1], b[1]);
        e2 = XNOR(a[2], b[2]);
        e3 = XNOR(a[3], b[3]);
        e4 = XNOR(a[4], b[4]);
        e5 = XNOR(a[5], b[5]);
        e6 = XNOR(a[6], b[6]);
        e7 = XNOR(a[7], b[7]);

        out = AND(e0, e1, e2, e3, e4, e5, e6, e7);

end (out)
```

🦄