The Classic: ALU with Full Datapath

Team Chicken Dinner

Blake Beitter, Derek Bellanca, William (Ben) Speicher, Philip Foster, Daniel Wright

CS 4341.501

November 20, 2017

**Part 1: Updated Writeup and Member Tasks**

**Description**

Our team has decided to create an ALU with a full datapath. Our team is going to create a multi-state, digitally encoded processing unit using Verilog. The project is going to implement the functionality of a basic calculator by creating an ALU with a full datapath where the system can add, subtract, and multiply~~, and divide~~ two numbers. It will also be able to do logical AND, OR, XOR, and NOT. The system will also save the result of that operation so that another number can be added. The ALU will be able to be ~~turned on, off,~~ reset or cleared. We have decided to use ~~8~~ 16-bit integer values, as going too much larger would end up becoming unmanageable for the length of time we have.

The ALU will be placed in a basic 4-function calculator which can add, subtract, multiply, ~~and~~ ~~divide~~ (we were not able to find a good way to do it without adding too many clock cycles) two values entered by a user. Additionally, the calculator will have a “set” and “clear” functionality which can store a value into a register and clear that register for later calculations. It will also be able to do the logical operations AND, OR, and XOR on two values, and do logical NOT on one value. ~~If we have enough time, we would also like to add additional functionality such as computing logarithms, exponents, square roots, and other mathematical functions, as well as the ability to compute more complex equations with multiple operations with correct order of operations.~~

An ALU by itself is not particularly useful, as it needs additional circuitry to provide instruction and data inputs, as well as take the output and do something useful with it. We feel that a calculator is an excellent choice to show off the functionality of the ALU. This provides a familiar user interface to most people, as well as

We feel that implementing an ALU will help solidify the ideas and concepts that we learn in class, and provide valuable experience which will help us potentially find employment in a related field. The project will also help us get a better understanding of how modern computer processors operate.

**Member Tasks**

|  |  |
| --- | --- |
| **Member** | **Task** |
| Blake Beitter | Multiply Function, **ALU, State Machines** |
| William (Ben) Speicher | **Full Circuit, ALU, State Machines, Verilog** |
| Daniel Wright ~~& Najee Grey~~ | Gate Logic, **ALU** |
| Derek Bellanca | Subtract Function, **Report, State Machines** |
| Philip Foster | Clear Function, **Keypad, ALU, State Machines** |
| ~~Najee Grey~~  All Members | Add function |
| All Members | ~~Divide Function~~ |
| All Members | Research into possible implementation |
| All Members | Circuit diagrams and Simulation |

**Part 2: Updated Software Research**

**Software Discovery**

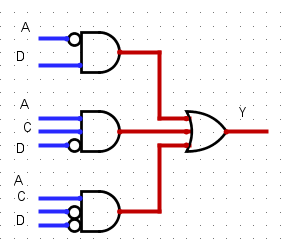
Logisim is a circuit development tool used to design and simulate digital logic circuits. The software is open-source, and runs on any machine supporting Java 5 or later. It can also run on Windows, MacOS X, and Linux making it an easy-to-use and accessible tool for all of our cohort members. All members should be able to use Logisim without any difficulty, as it has been used for each of our homework assignments.  Some cohort members have also used Logisim to write lab reports for the lab section.

The Logisim interface is a user-friendly and has a intuitive user-interface. Wires are color-coded for ease-of-use, and are easily drawn horizontally or vertically. The wires easily attach themselves to components and other wires. Depending on the complexity of the circuit design, we can design “subcircuits”, allowing us to build a hierarchy of circuits and “subcircuits” in an easily readable and organized format.

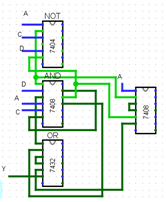
Logisim also provides all of the components we will need, including gates, multiplexers, decoders, encoders, **buttons**, flip-flops, ~~RAM~~, **logic functions**, and arithmetic circuits. One of the most important aspects of Logisim is that it includes combinational analysis, allowing us to convert between circuitry, truth tables, and Boolean expressions. This will help us ensure that our design is correct throughout the entire design process.  Additionally, we will be using Verilog in the project to create simulations. We are all familiar with this software, as all members of the cohort have used it to complete homework assignments.

**Logisim screenshots**

Normal logic gates we have used in class:



The same circuit above, implemented using 7400 series ICs.  The library containing these chips can be found by downloading the “7400 series Logisim library from Ben Oztalay”. A link can be found here: http://www.cburch.com/logisim/links.html



**Part 3: System Design: Diagrams, Tables, Write-Up**

**Descriptions**

**Parts List**

* **Arithmetic Logic Unit (ALU)**
  + 2 16-Bit Multiplexers
  + 1 AND Gate
  + 1 OR Gate
  + 1 XOR Gate
  + 1 NOT Gate
  + 1 Comparator
  + 1 16-Bit Adder
  + 1 16-bit Subtractor
  + 1 16-bit Multiplier
* **Finite State Machine**
  + 1 2-Bit Register
  + 10 1-bit AND Gates
  + 1 1-Bit NAND Gate
  + 2 NOT Gates
  + 2 OR Gates
* **Calculator**
  + 2 16-Bit Multiplexers
  + 1 3-Bit Multiplexer
  + 2 16-Bit Register
  + 1 3-Bit Register
  + 1 ALU Component
  + 1 FSM Component
  + 3 1-Bit AND Gates
  + 1 1-Bit OR Gate
  + 1 16-bit Multiplier
  + 1 16-Bit Adder

**Inputs**

* **Control FSM**
  + **Press** - Activates when anything is pressed. Signals that data was entered
  + **Code** - Data input from the system
* **2. ALU**
  + **a** - Operand 1 that is used in the operation
  + **b** - Operand 2 that is used in the operation
  + **op** - Operator that is used in operation
* **Calculator**
  + **Pressed -** Activates when anything is pressed. Signals that data was entered
  + **Code -** Data input from the system

|  |  |
| --- | --- |
| **Code** | **Meaning** |
| 00000-01001 | Number |
| 01010 | Addition |
| 01011 | Subtraction |
| 01100 | Multiplication |
| 01110 | Assignment |
| 01111 | Reset |
| 11010 | AND |
| 11011 | OR |
| 11100 | XOR |
| 11101 | NOT |

|  |  |
| --- | --- |
| **Opcode** | **Expression** |
| 000 | A ADD B |
| 001 | A SUB B |
| 010 | A MULT B |
| 100 | A AND B |
| 101 | A OR B |
| 110 | A XOR B |
| 111 | NOT A |

**Outputs**

* **Control FSM**
  + **swap** - If it is an operator, set to 1 so a value can be moved to the next register
  + **calc - State = 10**, activated when the enter key is pressed
  + **op** - The operator entered by the keypad. Ignored if not in the range [13:10] or [28:26].
  + **rst** - Reset, is only true when code is equal to 01111 (see output for the keypad)
* **ALU**
  + **err** - Is high when overflow or underflow occurs
  + **out** - The result of the operation
* **Calculator**
  + **input -** Displays the input value
  + **output -** Displays the output value

**Features and Operations**

* **Control FSM**
  + Stores the current state of the calculator (i.e. which input to accept)
  + Transitions between states when appropriate, when signal 13 is received
* **ALU**
  + Performs all arithmetic and logical operations (+, -, \*, AND, OR, XOR, NOT)
  + Can output an error for an overflow or an underflow
* **Calculator**
  + Displays the output and input values
  + Connects the FSM and the ALU
  + Inputs the values
  + Translates the keypresses into 16-bit values
  + Displays an error if an underflow or overflow occurs

**Modes and States**

* **States**
  + **enter1** - Calculator is started in enter1 state. User enters first operand. State machine stays in enter1 until overflow occurs (sending it to the error state) or an operator is entered. If logical NOT is entered, state proceeds directly to result
  + **enter2** - Entering an operator changes enter1 to enter2. enter2 is similar to enter1, except operators overwrite the value from enter1, and pressing **ENTER** will then output the result of the ALU. This subsequently moves the current state to result. Alternatively, if there is underflow or overflow, it proceeds to the error state
  + **result** - Pressing **ENTER** while in enter2 moves the current state to result. While in the result state, the user may press an operator to act on the accumulated result as if they had entered in in state enter1, or they may press a number to start anew. Pressing a number acts as the **CLEAR** command. Entering a logical NOT while in result will perform the logical NOT operation on result and stay in result
  + **error** - Current state moves to error whenever overflow or underflow occurs. Moves back to enter1 whenever anything is entered on the keypad

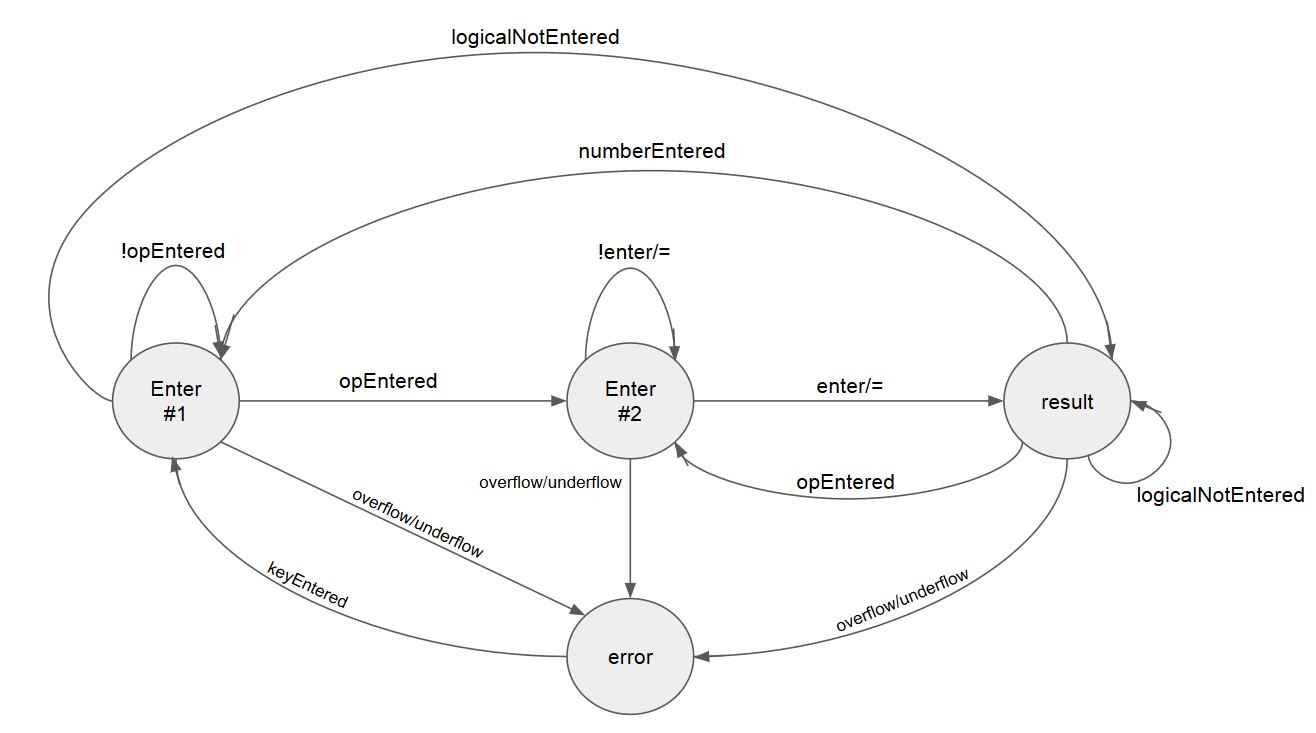
**Current State**

**Registers**

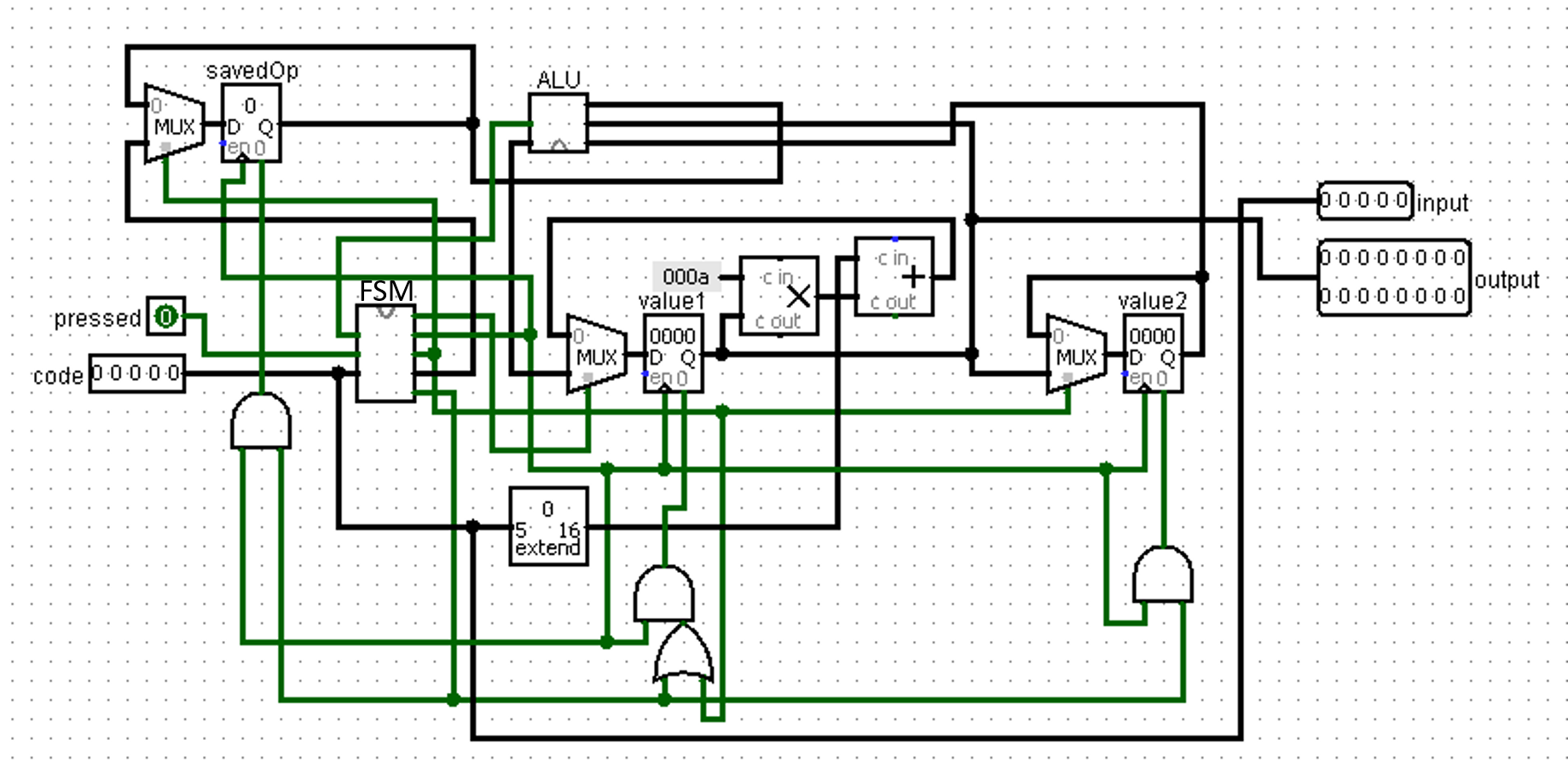
* **Saved op**
  + Stores the current operation (+, -, \*, AND, OR, XOR, NOT)
* **Value 1**
  + Stores the value being edited
* **Value 2**
  + Stores the first operand while the second is being edited
* **State:Store**
  + Stores the current state of the FSM

**Diagrams**

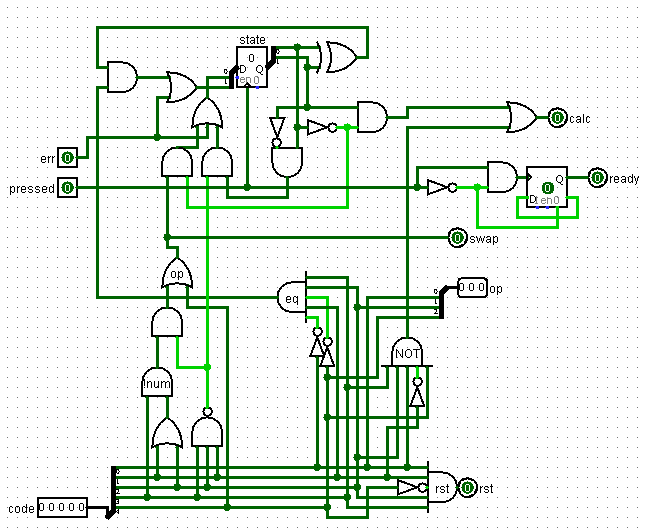
**State Machine**

****

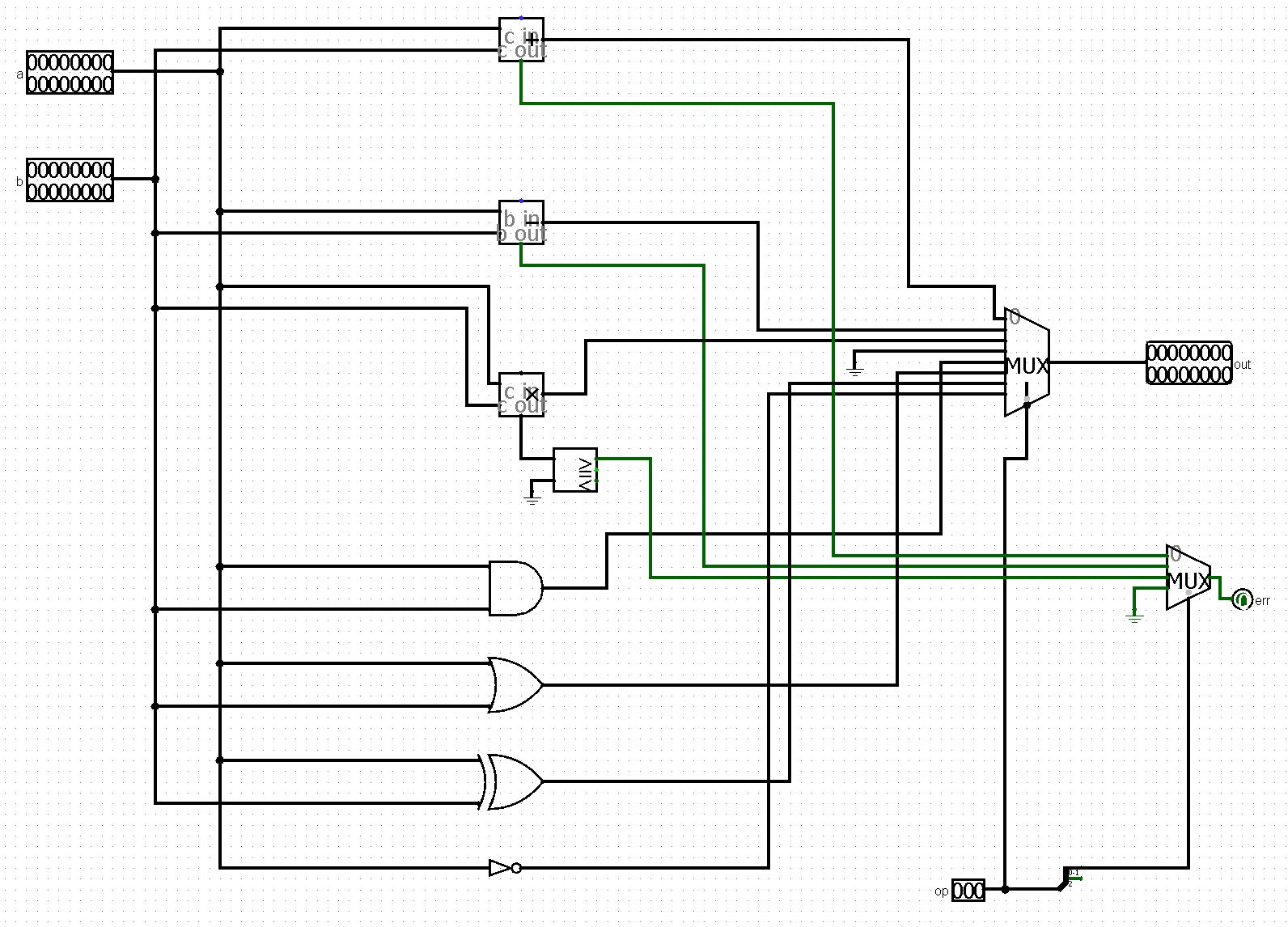
**Main Circuit Diagram**



**FSM Circuit Diagram**



**ALU Circuit Diagram**



**Part 4: Verilog Simulator with Output**

NOTE: Using seed 59 achieves these results. Further output not shown for brevity of one page.

input: 15, output: 0 //reset the calculator, sets to 0

input: 0, output: 0 //enter in 0. it's still 0.

input: 2, output: 2 //enter in 2, now it's 2.

input: 8, output: 28 //enter in 8, make it 2\*10+8=28.

input: 27, output: 0 //enter in the OR operation. Move value1 to value2.

input: 6, output: 6 //enter in 6.

input: 14, output: 30 //calculate 11100|00110. the result is 30, aka 11110.

input: 15, output: 0 //reset the calculator.

input: 0, output: 0

input: 3, output: 3

input: 2, output: 32

input: 11, output: 0 //enter in the subtract operation

input: 9, output: 9

input: 8, output: 98

Error //underflow! 32 is less than 98.

input: 14, output: 64890 //bad result of underflow. ignored.

input: 15, output: 0

input: 0, output: 0

input: 4, output: 4

input: 6, output: 46

input: 10, output: 0 //enter in the add operation

input: 1, output: 1

input: 14, output: 47 //calculate 46+1. What a doozy.

input: 15, output: 0

input: 6, output: 6

input: 6, output: 66

input: 9, output: 669

input: 12, output: 0 //enter in the multiply operation

input: 1, output: 1

input: 8, output: 18

input: 14, output: 12042 //669\*18=12042

input: 15, output: 0

input: 7, output: 7

input: 1, output: 71

input: 8, output: 718

input: 12, output: 0 //multiply again

input: 6, output: 6

input: 8, output: 68

input: 14, output: 48824 //718\*68=48824

input: 15, output: 0

input: 8, output: 8

input: 8, output: 88

input: 5, output: 885

input: 27, output: 0 //OR

input: 8, output: 8

input: 4, output: 84

input: 14, output: 885 //885|84=885

**References**

[1] Burch, Carl. “Logism.” Logism, 2.7.0, SourceForge.net, 21 Mar. 2011, www.cburch.com/logisim/index.html.