SimLogic manual

1 What is SimLogic

SimLogic is a bare bones logical circuit simulator for educational purposes. While a great number of sophisticated logic circuit simulators exist, this one is so simple, that anyone mastering C++ can understand it's sourcecode.

Although the filenames end on *.cpp*, writing simulations requires only *C* knowledge. However if you want to add special purpose console I/O, stripped down versions of the C++ streams *cin* and *cout* are available.

It can run on a PC or on an Arduino single board embedded computer. First *create* circuit elements and then *connect* them, as in the following code:

Listing 1: Simulation of a Nand circuit, in the project nand_example

```
#include "simlogic.h"

int main () {
    create (Input, inputA);
    create (Input, inputB);
    create (And, anAnd);
    create (Not, aNot);

    connect (inputA, anAnd.inA);
    connect (inputB, anAnd.inB);
    connect (anAnd, aNot.in);

    while (true) {
        evaluate ();
    }

    return 0;
}
```

The code above will run on a PC under Windows, Linux and OsX, using console I/O.

To make your code run on an Arduino processor board, add I/O blocks as shown below. Note that, due to the use of `#ifdef`, the code will still run on a PC as well.

Listing 2: Nand circuit with Arduino I/O added

```
#include "simlogic.h"
```

```

int main () {
    create (Input, inputA);
    create (Input, inputB);
    create (And, anAnd);
    create (Not, aNot);

    connect (inputA, anAnd.inA);
    connect (inputB, anAnd.inB);
    connect (anAnd, aNot.in);

#ifdef arduino
    pinMode (2, INPUT_PULLUP); pinMode (3, INPUT_PULLUP);
    pinMode (4, OUTPUT); pinMode (5, OUTPUT); pinMode (6, OUTPUT); pinMode (7,
#endif

    while (true) {

#ifdef arduino
        inputA.value = 1 - digitalRead (2); // Inverted because
        inputB.value = 1 - digitalRead (3); // of pull-up resistors
#endif

        evaluate ();

#ifdef arduino
        digitalWrite (4, inputA.value);
        digitalWrite (5, inputB.value);
        digitalWrite (6, anAnd.value);
        digitalWrite (7, aNot.value);
#endif

    }
    return 0;
}
```

2 How it works

Each SimLogic program is defined in a file called *simulation.cpp*. Note that this one file is the only "valuable" user made file in a project. All the rest is boilerplate code, identical for each project. If you want to make a new project *my_project*, just copy all files from folder *nand_example* to a new folder *my_project*. Rename the *.ino* file from *nand_example.ino* to *my_project.ino*. Then edit the *file simulation.cpp* to your liking and compile into an executable using any C++ compiler. The script *compile.bat* requires *gcc* being installed on Windows.

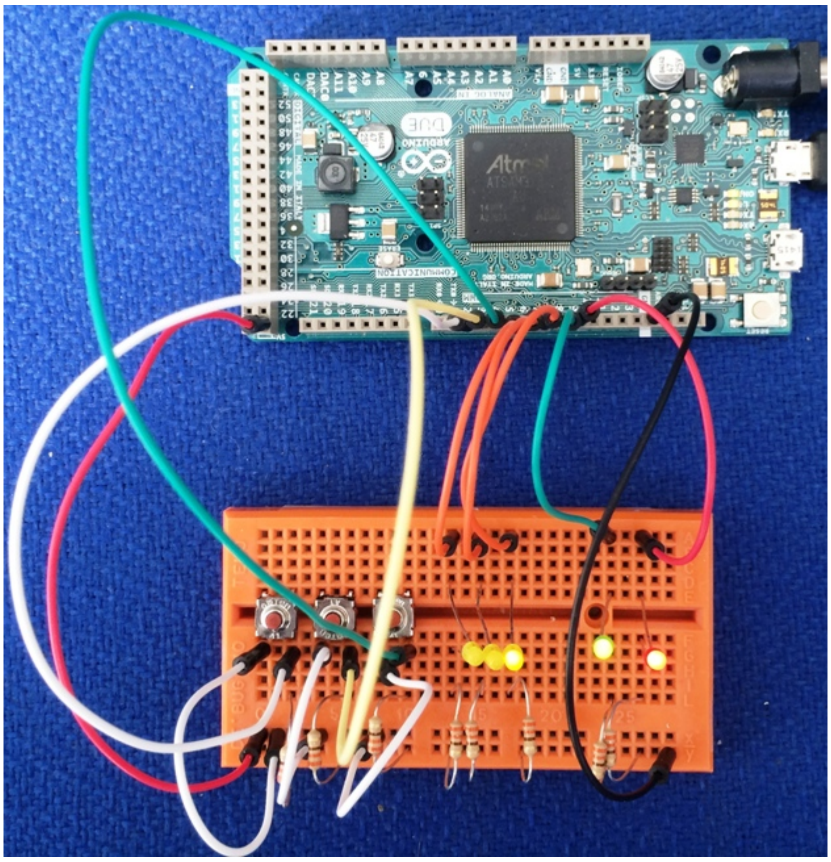
A simulation can run in three ways, depending on macro defintions in *simulation.h*, namely:

1. Console I/O on a PC using a command prompt: symbol *console* defined in *simulation.h*, compilation with *compile.bat* on PC
2. Console I/O on an Arduino using its IDE's serial monitor: symbol *console* defined, compilation in Arduino IDE. In this case hardware outputs, e.g. LEDs will also work. On the serial monitor, set *Autocroll* on, set the line terminator to *Newline* and set the speed to *9600 baud*. Input happens in one-line entry field at the top of the serial monitor window and is echoed together with output to the multinline text view field at the bottom.
3. Hardware I/O on an Arduino, using e.g. buttons and LED's: symbol *console* not defined, compilation in Arduino IDE.

N.B. A SimLogic program behaves like a PLC (Programmable Logic Controller) rather than a hardwired ciruit. This means that its circuit elements are always evaluated in creation order, exactly once for each evaluation cycle (sweep, in PLC terminology). This guarantees that the behaviour is always the same and so called *race conditions* are avoided. It also means that *the creation order of your circuit elements should run from cause to effect.* Pay attention to this, especially if your program behaves differently from what you expect.

3 Hardware

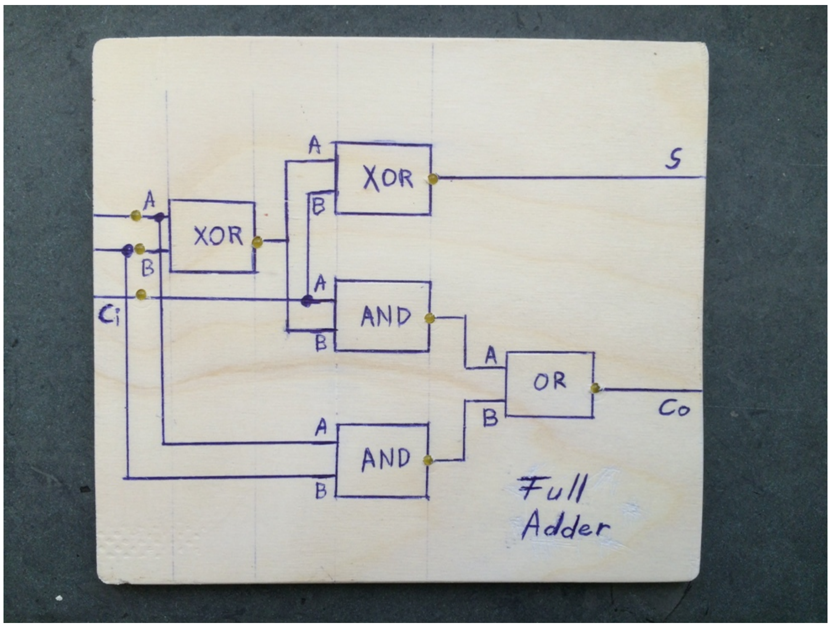
When running on an Arduino, connecting buttons and LEDs to the logic circuits provides a visual cue to what's going on. The hardware for a full adder e.g. looks as follows:



Figuur 1: Hardware for the full adder

The above circuit uses external resistors to pull the inputs down to ground level (- of the powersupply) when the switches are open. In that case the I/O mode of the inputs is specified in the code as INPUT. As can be seen in the example, INPUT_PULLUP rather than INPUT is used. This means that the inputs are pulled up to the + level of the power supply when the switches are open. In that case the external pulldown resistors can be left out. To prevent the signal from being inverted, the break contact of the switches has te be used.

To gain better insight into the behaviour of the subcircuits and in the way they cooperate to produce the desired output, the LED's can best be placed into a circuit diagram of the logic that is simulated, like shown below:



Figuur 2: Circuit diagram with LED's for the full adder

4 Elementary building blocks

The available circuit element types with their input connectors, used in the *connect* statement, are:

- False
- True
- Input: in
- And: inA, inB
- Or: inA, inB
- Xor: inA, inB
- Not: in
- Oneshot: in
- Latch: set, reset

5 Stream I/O

The simulator has its own bare bones I/O that requires no programming, it's just there when you use the console.

If you'd like to be fancy and are familiar with C++ you can use *cin* and *cout*, which, on an Arduino, are both of type *SerialStream*:

```
struct SerialStream {
    SerialStream &operator<< (const char * const data);
    SerialStream &operator<< (bool data);
    SerialStream &operator<< (int data);
    SerialStream &operator>> (char * const data);
};
```

Example of writing something to the console:

```
cout << "There are " << 2 << " boolean values, namely " << true << " and "
```

Example of reading a zero terminated string from the console:

```
cin >> answer;
```

Note that type SerialStream can be extended by inheritance.

6 Assorted facts and tips

- The project *nand_example* has been precompiled for Windows with g++ using the *compile.bat* script and will use console I/O.
- Simulations should also compile on Linux and OS X, using appropriate C++ compilers and shell scripts or make files.
- When running in a console window, terminate a simulation simply by CTRL + C.
- There's no error protection, so after an erroneous input it's best to restart the simulation.
- While simulations may use C'ish idiom, still the *.cpp* extension is mandatory.
- Feel free to use C++ rather than C if you're already familiar with the benefits of Object Oriented Programming.
- To instrument your simulation for use with Arduino hardware I/O, add I/O facilities as shown in the *nand_example*
- To run on an Arduino, go to the directory that holds your project, edit *simulation.h* according to wether or not console I/O is desired, and then use [Sketch][Upload].
- In general the "everything in one directory" approach of the Arduino IDE is followed, to keep things simple.

7 Legal notices