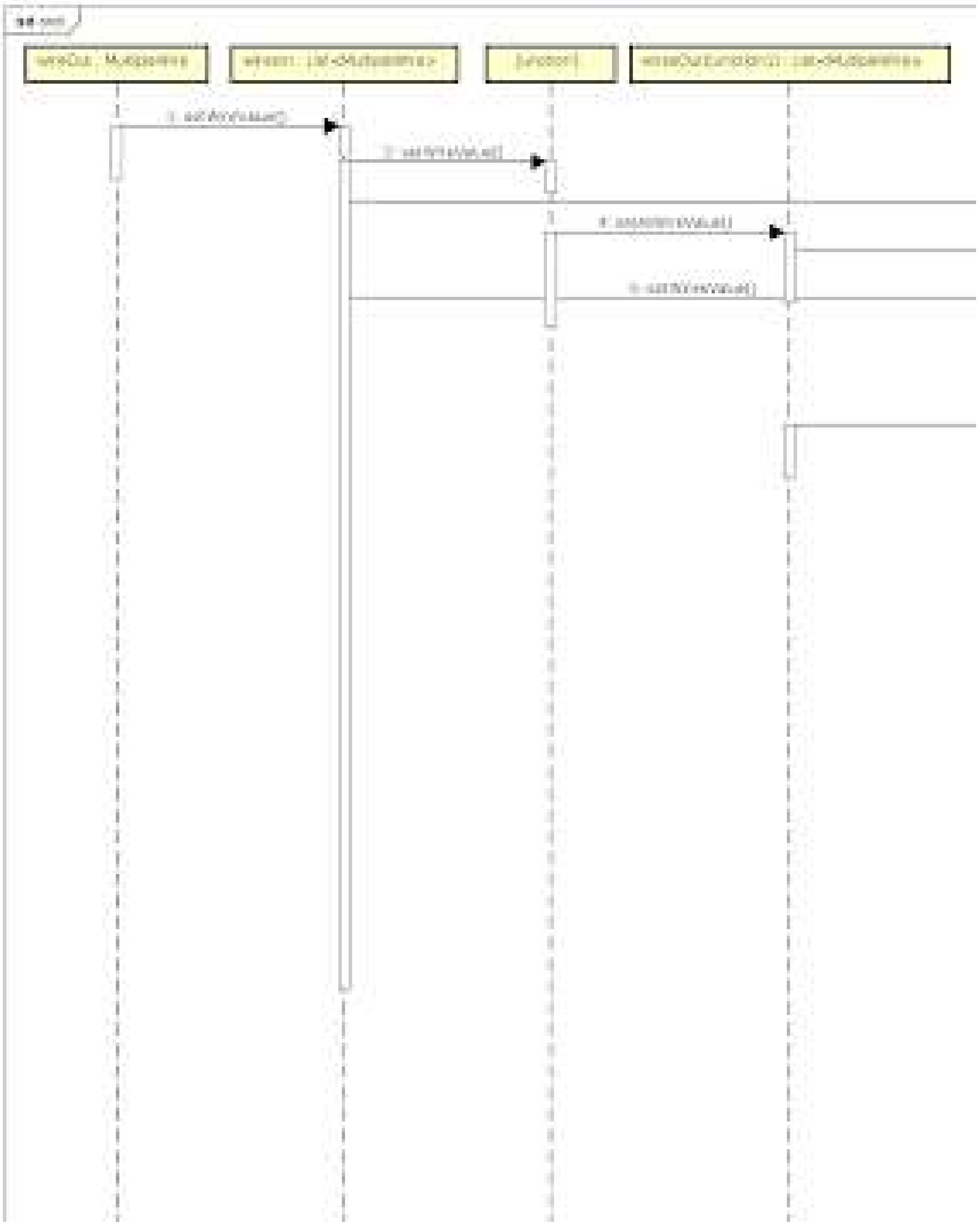

sd
wireOut : Multipipeline
wiresIn : List<Multipipeline>
junction
wiresOutJunction : List<Multipipeline>
1: setAll(value)
2: setAll(value)
4: setAll(value)
5: setAll(newValue)

This is the beginning, where the we initialize the hashmap if wireout. We initialize the wiresIn and wiresOut outside of the class.
i.e.

```
HashMap<String, MultipleWire> inputs = new HashMap<String, MultipleWire>();
MultipleWire input1 = new MultipleWire(0); // A
MultipleWire input2 = new MultipleWire(0); // B
MultipleWire input3 = new MultipleWire(1); // C_IN
inputs.put("input1", input1);
inputs.put("input2", input2);
inputs.put("input3", input3);
HashMap<String, MultipleWire> outputs = new HashMap<String, MultipleWire>();
HashMap<String, Component> noInnerComponents = new HashMap<String, Component>();
Adder adder = new Adder(inputs, outputs, noInnerComponents, "adder", "Adds two bits");
```

This is the main driver code in the 1bit adder example. We put these in, we simulate creating the adder and make our own wires that go into the adder. In the application we will have some other device going into it, but as of now we are letting these wires be set as turned on/off.

Then in the adder class. We have the four different junctions as represented in the

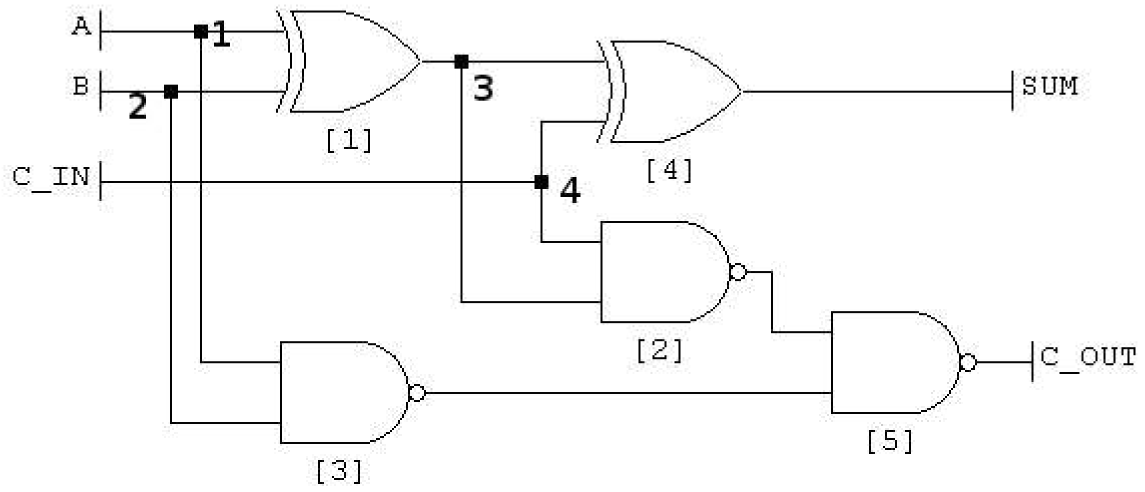
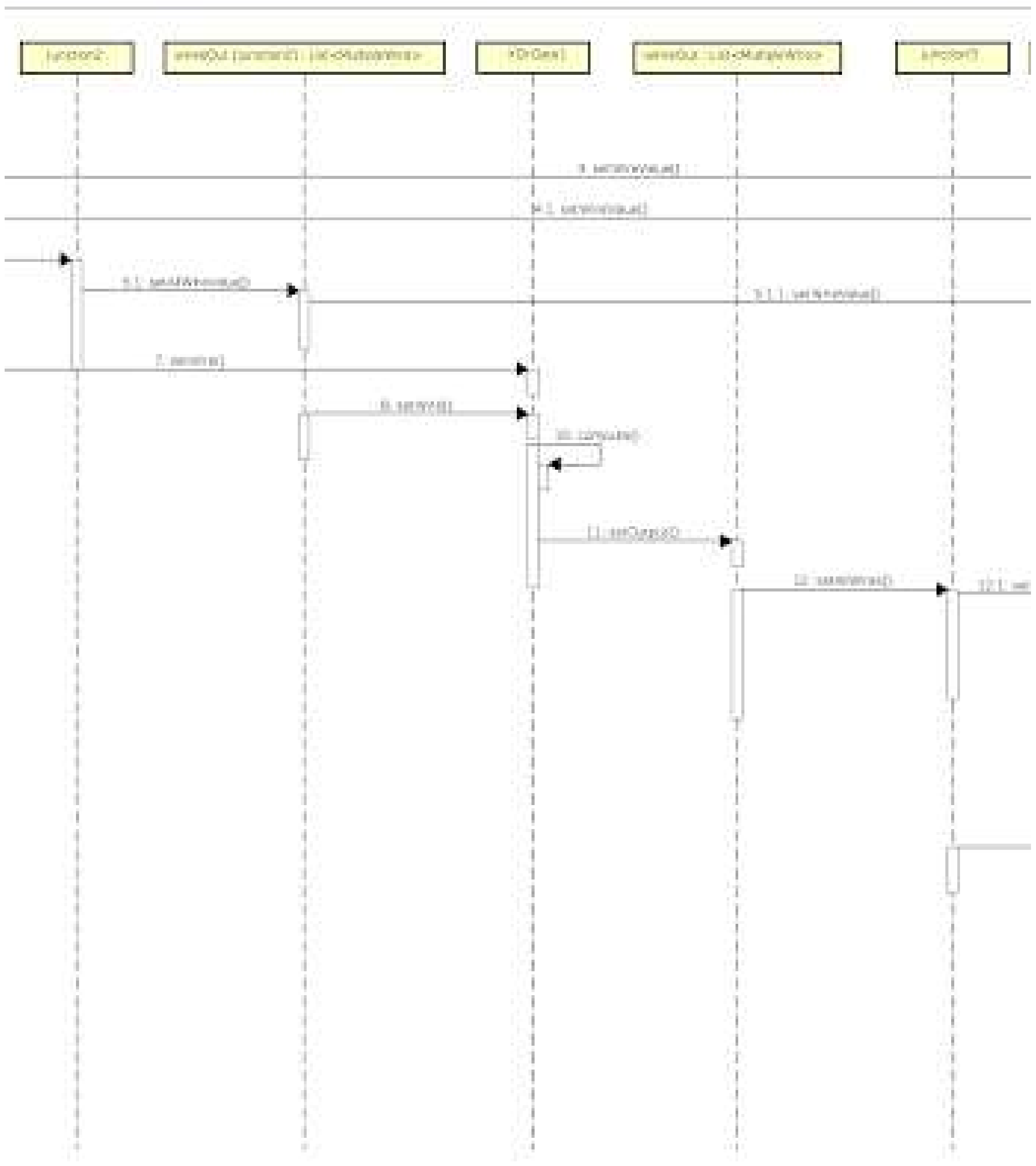


diagram.

First we get junction1, and set it’s input to A, and then we the code in the junction will then set it’s outputs to Nand3, and Xor1 accordingly.

```
public class Junction extends Component {

	public Junction(HashMap<String, MultipleWire> in, HashMap<String, MultipleWire> out,
			HashMap<String, Component> innerComponents, String name, String
description) {
		super(in, out, innerComponents, name, description);
		this.wiresOut.get("output1").getValue().bind(this.wiresIn.get("input1").getValue());
		this.wiresOut.get("output2").getValue().bind(this.wiresIn.get("input1").getValue());

	}
}
```

This is done by the code above.

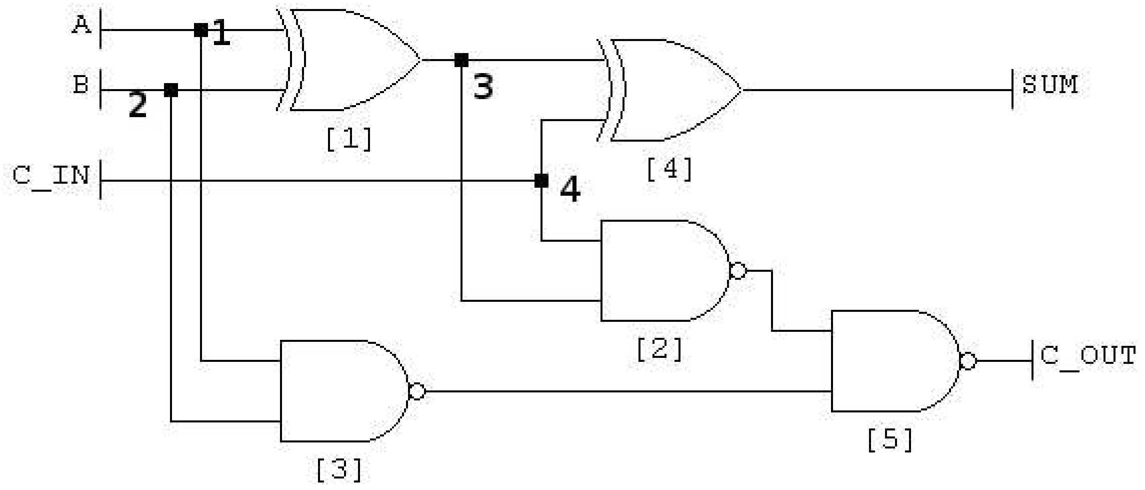


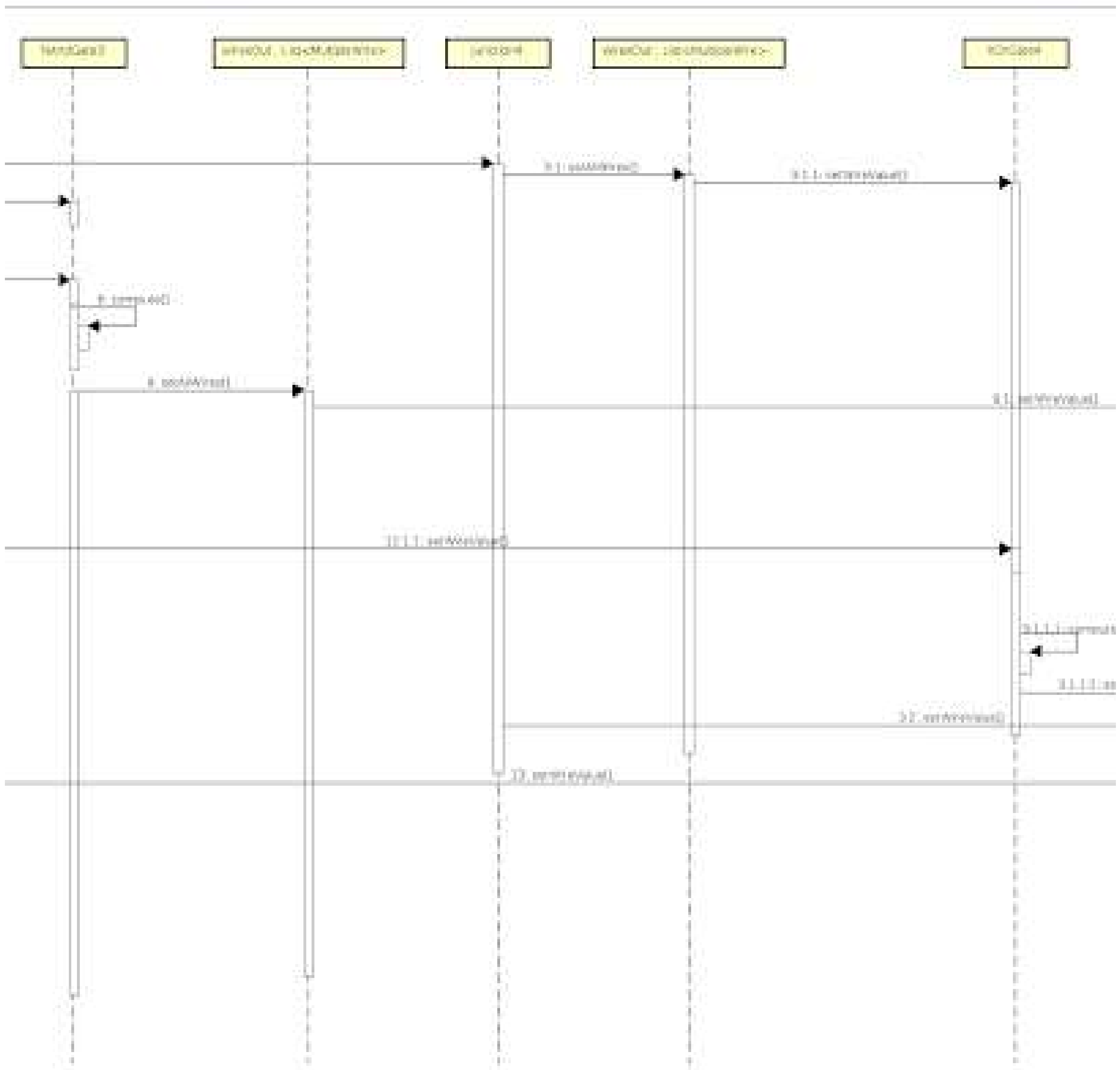
Above is a continuation of the same exact thing happening in junction2, however now we have the XORgate1. The compute function is computed internally by the binding function which is :

```
public class XorGate extends Component {

	public XorGate(HashMap<String, MultipleWire> in, HashMap<String, MultipleWire> out,
				HashMap<String, Component> innerComponents, String name, String
description) {
		super(in, out, innerComponents, name, description);
		this.wiresOut.get("output1").getValue().bind(Bindings.createIntegerBinding(new
Callable<Integer>() {
				@Override
				public Integer call() throws Exception {
					return wiresIn.get("input1").getValue().get() ^
wiresIn.get("input2").getValue().get();
				}

		}, this.wiresIn.get("input1").getValue(), this.wiresIn.get("input2").getValue()));

	}

}
```

The binding will then turn the output into the xor of the inputs received from the xor item. So in actuality the compute function is never called, but for clarity it is included. (The binding function internally performs the compute() and the setOutput().

This then leads to junction3. performs the same thing as junctions 1 and 2, however the values go out to XOR 4 and Nand2.

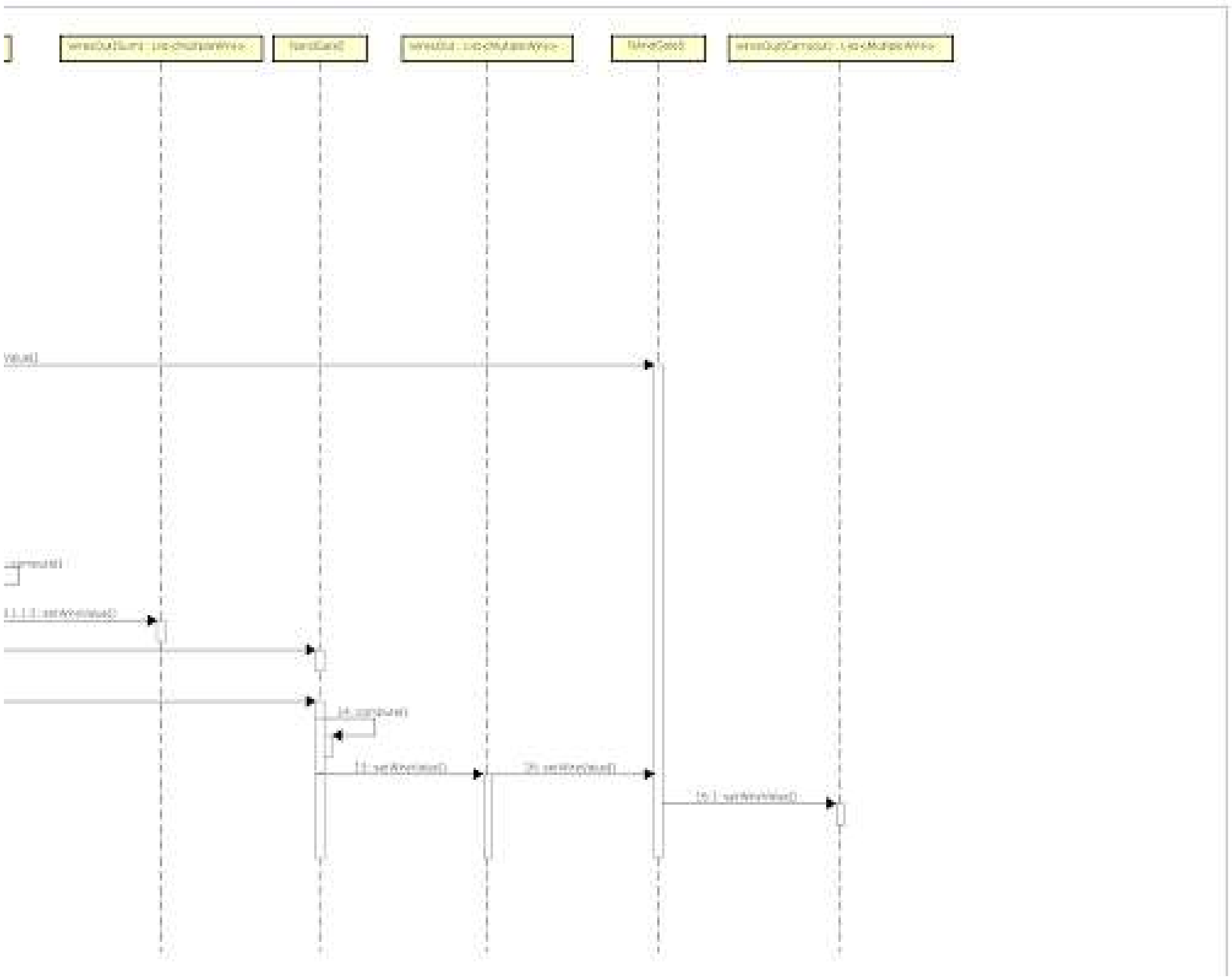




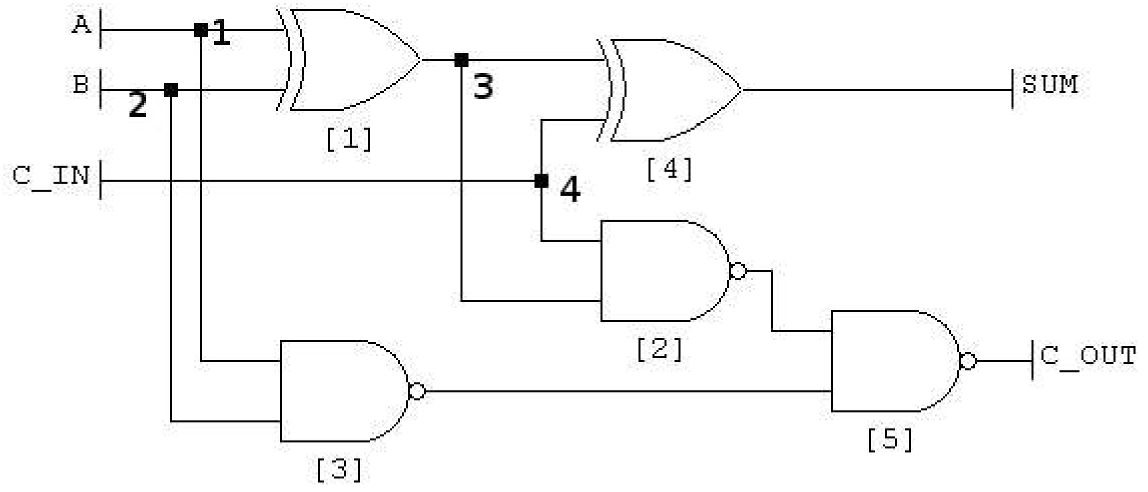
In this part it is similar to the xor gate above. The binding function, for NandGate3 is compute() and setOutput(). The code is as shown :

```
/**
 * Class NandGate, this class is a little bit different since max/min doesn't work here. I had utilize
the create integer binding function to create a new binding
 * to turn the integer input values into boolean values. From there, it utilizes a true nand operation,
and then returns 1/0 accordingly.
 * @author Frank Hu
 *
 */
public class NandGate extends Component {
	NumberBinding output = null;
	public NandGate(HashMap<String, MultipleWire> in, HashMap<String, MultipleWire> out,
			HashMap<String, Component> innerComponents, String name, String
description) {

		super(in, out, innerComponents, name, description);
		Bindings.min(this.wiresIn.get("input1").getValue(),
this.wiresIn.get("input2").getValue());

		this.wiresOut.get("output1").getValue().bind(Bindings.createIntegerBinding(new
Callable<Integer>() {
			@Override
			public Integer call() throws Exception {
				boolean wiresInOne = false;
				boolean wiresInTwo = false;
				wiresInOne = wiresIn.get("input1").getValue().get() == 0 ? false :
true;
				wiresInTwo = wiresIn.get("input2").getValue().get() == 0 ? false :
true;
				if (!(wiresInOne && wiresInTwo)) {
					return 1;
				} else {
					return 0;
				}
			}

		}, this.wiresIn.get("input1").getValue(), this.wiresIn.get("input2").getValue()));

	}


}
```

Xorgate4, is similar to xorgate1. It does the same exact thing. We do not wire this output up to anything, however the hashmap set value of “output1” of the adder is the output of xorgate1.



As we continue through the sequence diagram. We get to Nandgate5 and nandgate2, which in the diagram below creates the output of C_out. By utilizing the code above, and by simply setting the hashmaps of each of the components of “input1” and “input2” for each of the gates correctly it should output the correct sum and C_out accordingly.


A
B
C_IN
1
2
3
4
[1]
[4]
[2]
[3]
[5]
SUM
C_OUT