# Basic logic circuits

Carl-Johannes Johnsen

Department of Computer Science
University of Copenhagen

February 27, 2017

In this lecture, we will be looking at some basic combinatorial circuits, and how to implement them using SME.

We will try to construct the following circuits:

- Basic logical gates
- Decoder
- Half adder
- Full adder
- *n*-bit adder

A logic gate is a circuit abstraction with one or more inputs, and an output. It computes its output value based on the logic function it is mimicing. We look at the four basic logic gates:

`AND` - outputs 1 iff. all of its inputs are 1, otherwise 0

`OR` - outputs 1 if one or more of its inputs are 1, otherwise 0

`NOT` - outputs the inverse of its input, e.g. 1 becomes 0

`XOR` - outputs 1 iff. exactly one of its inputs are 1, otherwise 0

| Bit1 | Bit2 | AND | OR | NOT | XOR |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Implementing each of the gates in SME is very simple. As with CSP, we create a process for each gate. Each gate has two input busses, except for `NOT`, which only has one, and one output bus. Each bus contains one `bool` value. We choose `bool`, to ensure that the later VHDL generation will construct wires, and not full-blown 32-bit busses.

Each of the processes will take its inputs, and put the result of its logic function on its output bus.

## Buses

```
using SME;

namespace LogicGates
{
    [TopLevelInputBus, InitializedBus]
    public interface Input : IBus
    {
        bool bit1 { get; set; }
        bool bit2 { get; set; }
    }

    [TopLevelOutputBus, InitializedBus]
    public interface Output : IBus
    {
        bool And { get; set; }
        bool Or  { get; set; }
        bool Not { get; set; }
        bool Xor { get; set; }
    }
}
```

Note: for compactability, the inputs and outputs have been gathered on a single bus.

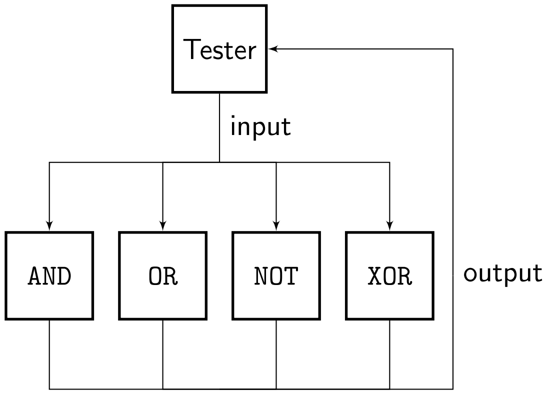
## AND gate

```
public class AND : SimpleProcess
{
    [InputBus]
    Input input;

    [OutputBus]
    Output output;

    protected override void OnTick()
    {
        output.And = input.bit1 && input.bit2;
    }
}
```

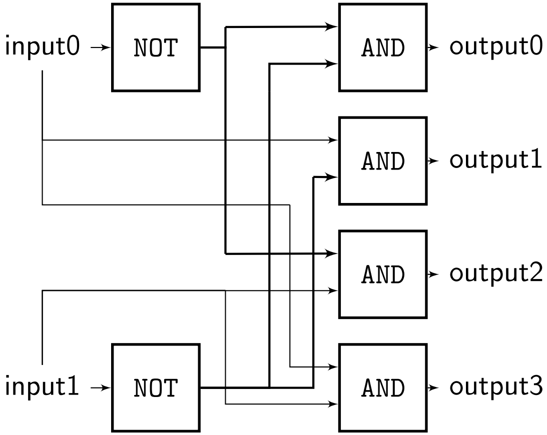
To test our implementation, we are going to need a process, which is going to send data on the input bus for each of the components, and is going to verify that each component outputs the expected value. Since we are only working with two bits, we can try all the possible combination of inputs.


Tester
input
AND
OR
NOT
XOR
output

Now that we have made some basic gates, we are going to construct our first combinatorial circuit: The decoder.

A decoder takes an $n$-bit input, and produces an $2^n$-bit output, where the $n$th output bit is set to 1, if the input is the binary representation of $n$. All the other output bits are set to 0.

We will start by looking at a 2-bit decoder for simplicity. To implement the decoder, we will combine the basic logic gates as follows:



## Buses.cs

```
using SME;

namespace Decoder
{
        public interface Input : IBus
        {
                bool Bit0 { get; set; }
                bool Bit1 { get; set; }
        }

        public interface Output : IBus
        {
                bool Bit0 { get; set; }
                bool Bit1 { get; set; }
                bool Bit2 { get; set; }
                bool Bit3 { get; set; }
        }

        public interface BitBus : IBus
        {
                bool Bit { get; set; }
        }
}
```

## Decoder.cs

```
public class Not0 : SimpleProcess
{
        [InputBus]
        Input input;

        [OutputBus]
        Internal0 output;

        protected override void OnTick()
        {
                output.Bit = !input.Bit0;
        }
}
```

Now, we also want to construct an *n*-bit decoder, however, this is not trivial, as SME requires everything to be known at compile time, and we cannot make a generic process, as these depend on the names of the buses.

To solve this problem, we can use C# templates:

- For each input bit, create a bus
- For each bus, create a `NOT` gate
- Create $2^n$ output busses
- For each output bus, attach an `AND` gate
- Connect the input busses and the output from the `NOT` gates, so that the binary representation of *n* will produce a `1` at the *n*th `AND` gate

## Buses.tt

```
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
<# int BitWidth = 6; #>
<# int pow = 64; #>
using System;
using SME;

namespace ScalDecoder {

<# for (int i = 0; i < BitWidth; i++)
{
#>
        [InitializedBus] public interface Input<#Write(""+i);#> : IBus { int Bit { get;
<#
}
#>

<# for (int i = 0; i < pow; i++)
{
#>
        [InitializedBus] public interface Output<#Write(""+i);#> : IBus { int Bit { get;
<#
}
#>

}
```

## Decoder.tt

```
<# for (int i = 0; i < BitWidth; i++)
{
#>
                public class NOT<#Write(""+i);#> : SimpleProcess
                {
                        [InputBus] Input<#Write(""+i);#> input;
                        [OutputBus] InputN<#Write(""+i);#> output;

                        protected override void OnTick()
                        {
                                output.Bit = ~input.Bit & 1;
                        }
                }
<#
}
#>

<# for (int i = 0; i < pow; i++)
{
#>
                public class AND<#Write(""+i);#> : SimpleProcess
                {
<# for (int j = 0; j < BitWidth; j++)
{
#>
                        [InputBus] Input<# if (((i >> j) & 1) == 0) { Write("N"); } Writ
<#
}
#>
                        [OutputBus] Output<#Write(""+i);#> output;
```

As with the basic logic gates, we need to construct a test process, which sends input to the circuit, and verifies that the output is as expected. In the case of our 2-bit decoder, we can also try all possible ($2^2 = 4$) values.

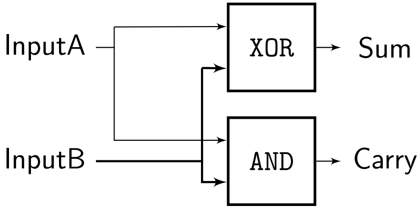
To test our *n*-bit decoder we need to construct another template, which will try all possible combinations.

The final circuit we will be looking at is the adder. To construct a full *n*-bit adder, we are going to need two subcomponents: An half-adder and an full-adder, both of which can be implemented using the basic logic gates.

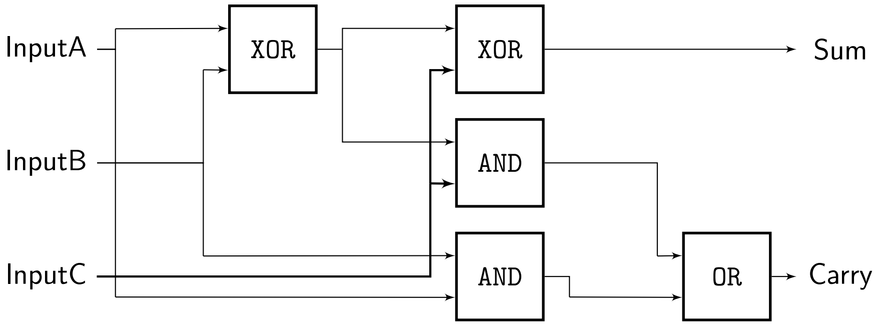
The half-adder takes two input bits, and produces two output bits. The first output bit is the `sum` bit, and the second is the `carry` bit.

The full-adder is similar to the half-adder, with the only difference being an additional input.

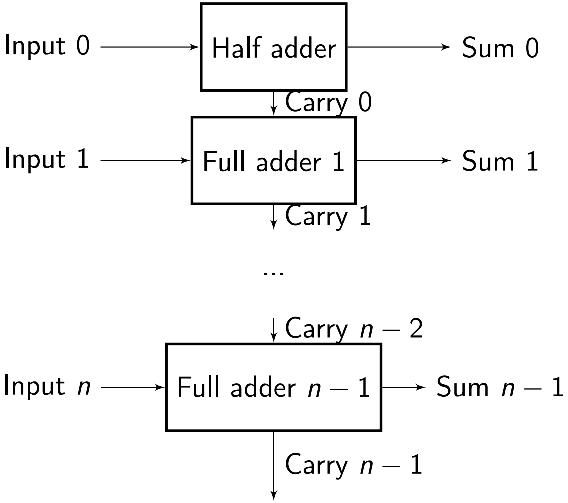
The combinatorial circuit for the half-adder.



The combinatorial circuit for the full-adder.



## How to wire up the sub circuits into a full *n*-bit adder



To implement this, we start by combining each of the sub circuits by using our basic logic gates, and by combining them into an *n*-bit adder.

Then we test our circuit in the same manner as before, with a tester process.