Discrete Event Simulation

Extended Example: Discrete Event Simulation

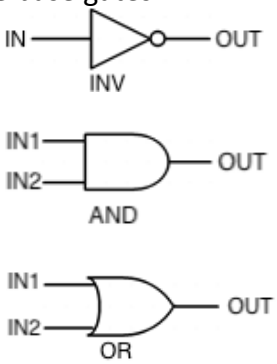
- Here's an example that shows how assignments and higher-order functions can be combined in interesting ways: We will construct a digital circuit simulator
- The simulator is based on a general framework for discrete event simulation

Digital Circuits

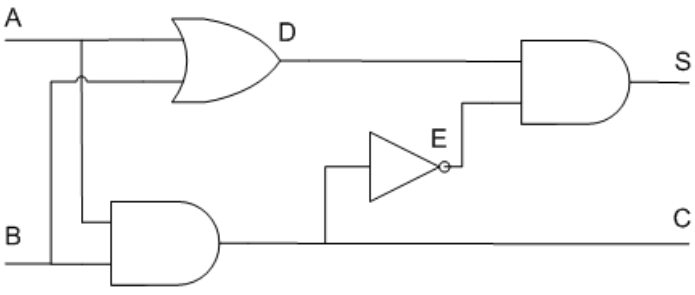
- Let's start with a small description language for digital circuits
- A digital circuit is composed of **wires** and of functional components
- Wires transport signals that are transformed by components
- We represent signals using Booleans true and false
- The base components(gates) are:
    - The **Inverter**, whose output is the inverse of its input
    - The **AND Gate**, whose output is the conjunction of its inputs
    - The **OR Gate**, whose output is the disjunction of its inputs
- Other components can be constructed by combining these base components
- The components have a reaction time(delay): their outputs don't change immediately after a change to their inputs

Digital Circuit Diagrams

- The base gates:



- The Half-Adder diagram:



sum: **S = A | B & ¬ (A & B)**
carry: **C = A & B**

A Language for Digital Circuits

- We describe the elements of a digital circuit using the following Scala classes and functions. To start with, the class Wire models wires
- Wires can be constructed as follows:

  val a = new Wire; val b = new Wire; val c = new Wire

  or equivalently:

  val a, b, c = new Wire
- Then, there are the following functions. Each has a side effect that creates a gate:

  def inverter(input: Wire, output: Wire): Unit
  def andGate(a1: Wire, a2: Wire, output: Wire): Unit
  def orGate(o1: Wire, o2: Wire, output: Wire): Unit

Constructing Components

- More complex components can be constructed from these
- For example, a half-adder can be defined as follows:

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire): Unit = {
  val d = new Wire
  val e = new Wire
  orGate(a, b, d)
  andGate(a, b, c)
  inverter(c, e)
  andGate(d, e, s)
}
```

- This half-adder can in turn be used to define a full adder(gets an input carry - cin as an input parameter):

```
def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire): Unit = {
  val s = new Wire
  val c1 = new Wire
  val c2 = new Wire
  halfAdder(b, cin, s, c1)
  halfAdder(a, s, sum, c2)
  orGate(c1, c2, cout)
}
```

- **Exercise**: What logical function does this program describe?

```
def f(a: Wire, b: Wire, c: Wire): Unit = {
  val d, e, f, g = new Wire
  inverter(a, d)
  inverter(b, e)
  andGate(a, e, f)
  andGate(b, d, g)
  orGate(f, g, c)
}
```

  The function f(a, b, c) describes: a & ¬b | b & ¬a == a ^ b, so the answer is: **a != b**

Discrete Event Simulation

- A discrete event simulation performs actions, specified by the user at a given moment. An action is a function that doesn't take any parameters and which returns Unit: ***type Action = () => Unit***
- The time is simulated; it has nothing to do with the actual time

Simulation Trait

- A concrete simulation happens inside an object that inherits from the trait Simulation, which has the following signature:

```
trait Simulation {
  /** Returns the current simulated time **/
  def currentTime: Int = ???
  /** Registers an action 'block' to perform after a given delay
  *  relative to the current time **/
  def afterDelay(delay: Int)(block: => Unit): Unit = ???
  /** Performs the simulation until there are no actions waiting **/
  def run(): Unit = ???
}
```

The Wire Class

- A wire must support three basic operations:
  - ***getSignal: Boolean*** – returns the current value of the signal transported by the wire
  - ***setSignal(sig: Boolean): Unit*** – modifies the value of the signal transported by the wire
  - ***addAction(a: Action): Unit*** – attaches the specified procedure to the actions of the wire. All of the attached actions are executed at each change of the transported signal
- Here is an implementation of the class Wire:

```
class Wire {
  private var sigVal = false
  private var actions: List[Action] = Nil
  def getSignal(s: Boolean) = sigVal
  def setSignal(s: Boolean): Unit =
      if (s != sigVal) {
          sigVal = s
          actions foreach( _() )
      }
  def addAction(a: Action): Unit = {
      actions = a :: actions
      a()
  }
}
```

- The state of a wire is modeled by two private variables:
  - ***sigVal*** – represents the current value of the signal
  - ***actions*** – represents the actions currently attached to the wire

The Inverter

- We implement the inverter by installing an action on its input wire. This action produces the inverse of the input signal on the output wire
- The change must be effective after a delay of InverterDelay units of simulated time
- We thus obtain the following implementation:

```
def inverter(input: Wire, output: Wire): Unit = {
    def invertAction(): Unit = {
        val inputSig = input.getSignal
        afterDelay(InverterDelay) { output setSignal !inputSignal }
    }
    input addAction invertAction
}
```

- The action should be performed every time the input wire changes its signal. We achieve that by adding the inverter action to the input wire, so that the wire itself would perform this action every time its signal changes

The AND Gate

- The AND gate is implemented in a similar way. The action of an AND gate produces the conjunction of input signals on the output wire. This happens after a delay of AndGateDelay units of simulated time
- We thus obtain the following implementation:

```
def andGate(in1: Wire, in2: Wire, output: Wire): Unit = {
    def andAction(): Unit = {
        val in1Sig = in1.getSignal
        val in2Sig = in2.getSignal
        afterDelay(AndGateDelay) { output setSignal (in1Sig & in2Sig) }
    }
    in1 addAction andAction
    in2 addAction andAction
}
```

- We add the AND action to both inputs. That way we make sure that whenever one of the two input changes, the output signal would be recomputed

The OR Gate

- The OR gate is implemented analogously to the AND gate:

```
def orGate(in1: Wire, in2: Wire, output: Wire): Unit = {
    def orAction(): Unit = {
        val in1Sig = in1.getSignal
        val in2Sig = in2.getSignal
        afterDelay(OrGateDelay) { output setSignal (in1Sig | in2Sig) }
    }
    in1 addAction orAction
    in2 addAction orAction
}
```

Exercise: *What happens if we compute in1Sig and in2Sig inline inside afterDelay instead of computing them as a value?*

```
def orGate2(in1: Wire, in2: Wire, output: Wire): Unit = {
    def orAction(): Unit = {
        afterDelay(OrGateDelay) { output setSignal (in1.getSignal | in2.getSignal) }
    }
    in1 addAction orAction
    in2 addAction orAction
}
```

Answer: *The sampling would take place at the same time after OR gate delay and the output change in the sampling would appear at exactly the same time. In the original orGate we would sample first, then wait orGateDelay time units and then set the output afterwards. So orGate2 does not model OR gates faithfully.*