Imperative Programming 2:
Circuit simulation

Hongseok Yang
University of Oxford

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages ...

From http://www.scala-lang.org

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages ...

object, class, inheritance, mutable state (var)

From http://www.scala-lang.org

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages ...

From http://www.scala-lang.org

pattern matching, high-order function, lazy evaluation

Today we will study programming idioms that use features of OO and FP together.

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages ...

From http://www.scala-lang.org

Today's lecture

- We will develop a circuit simulator that uses both OO and FP features.
- Expected outcome: Students can use the prog. idioms in the simulator for solving new problems.
- Based on Chapter 18.

Has anyone written a even-driven program such as a GUI program or an Android app before?

What about a scheduler?

Two programming idioms

- In their basic form, both idioms use an object that stores a list of functions in its var field:

```
var l : List[() => Unit]
```

- First idiom : To schedule and process dynamically generated tasks.
- Second idiom : To manage tasks that should be triggered when a certain event happens.

First idiom: Work-list algorithm

```
class Scheduler {
  var workList : List[() => Unit] = Nil

  def add(work: () => Unit) { workList ::= work }

  def run() {
    while (!workList.isEmpty) {
      val item::rest = workList
      workList = rest
      item()
    }
  }
}
```

First idiom: Work-list algorithm

```
class Scheduler {
  var workList : List[() => Unit] = Nil

  def add(work: () => Unit) { workList ::= work }

  def run() {
    while (!workList.isEmpty) {
      val item::rest = workList
      workList = rest
      item()
    }
  }
}
```

1. `item()` can call `add` and modify `workList`.

First idiom: Work-list algorithm

```
class Scheduler {
  var workList : List[() => Unit] = Nil

  def add(work: () => Unit) { workList ::= work }

  def run() {
    while (!workList.isEmpty) {
      val item::rest = workList
      workList = rest
      item()
    }
  }
}
```

1. `item()` can call `add` and modify `workList`.
2. A more sophisticated scheduling policy.

Second idiom: Observer pattern (Scala version)

```
class Counter {

  var x : Int = 0






  def inc() {
    x += 1

  }
  def get : Int = x
}
```

Second idiom: Observer pattern (Scala version)

```
class Counter {
  var observers : List[() => Unit] = Nil
  var x : Int = 0








  def inc() {
    x += 1

  }
  def get : Int = x
}
```

Second idiom: Observer pattern (Scala version)

```
class Counter {
  var observers : List[() => Unit] = Nil
  var x : Int = 0







  def inc() {
    x += 1
    observers foreach (_ ())
  }
  def get : Int = x
}
```

Second idiom: Observer pattern (Scala version)

```
class Counter {
  var observers : List[() => Unit] = Nil
  var x : Int = 0

  def add(o: () => Unit) {
    observers ::= o
    o()
  }
  def inc() {
    x += 1
    observers foreach (_ ())
  }
  def get : Int = x
}
```

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

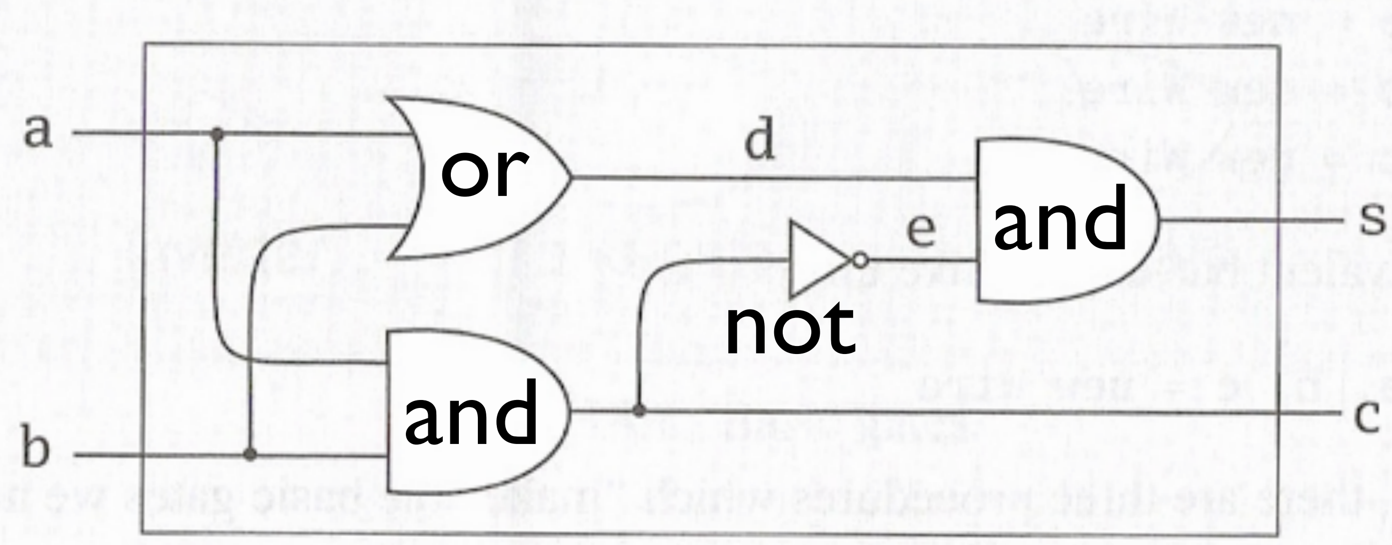


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

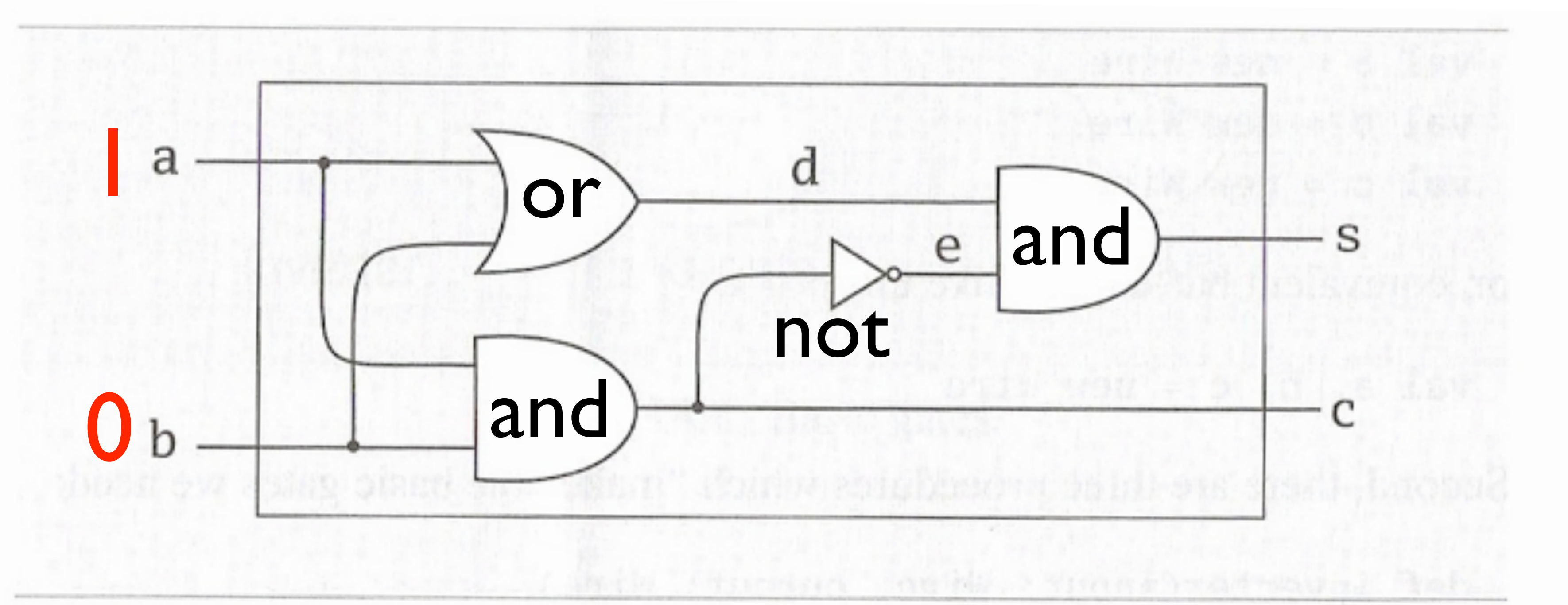


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

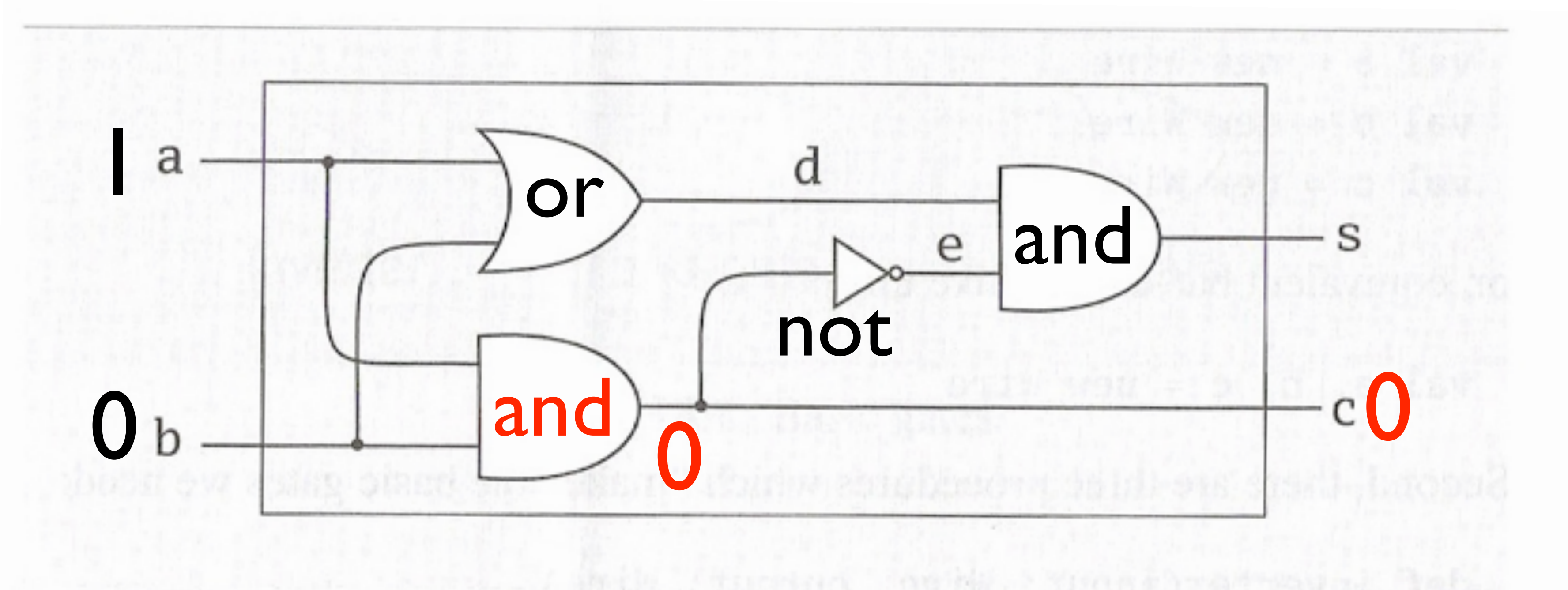


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

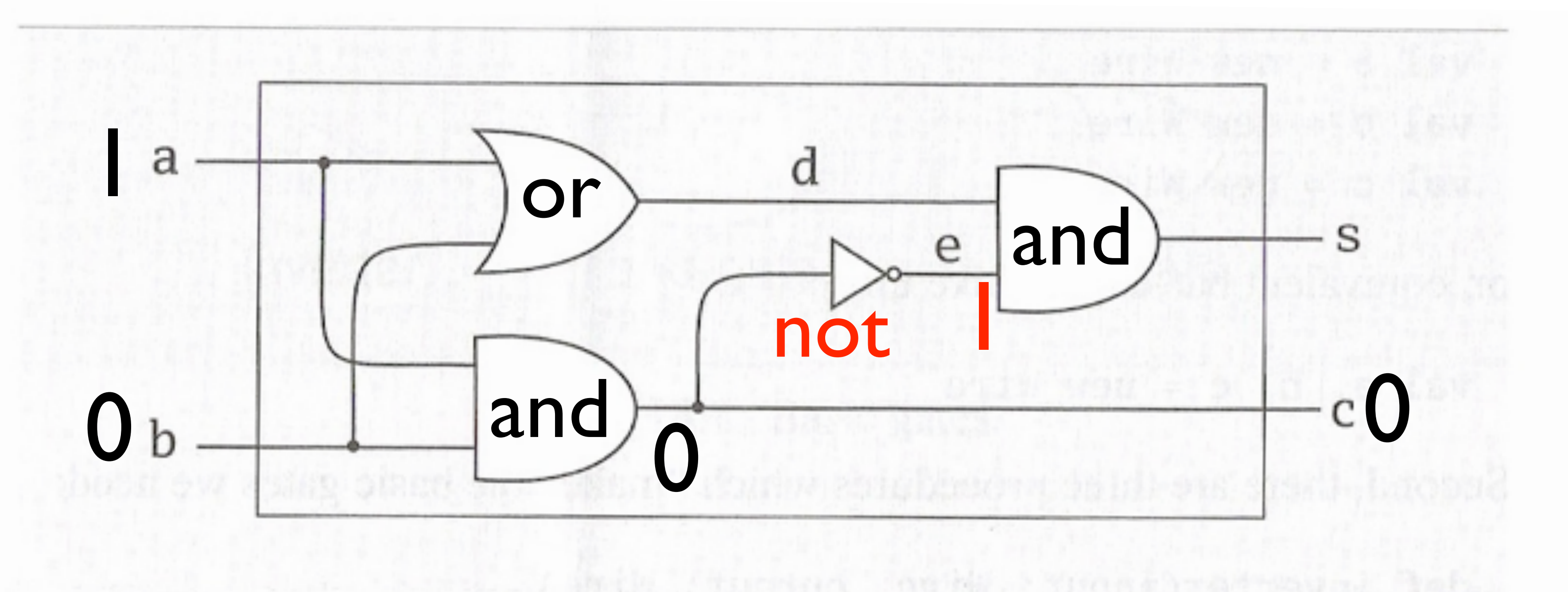


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

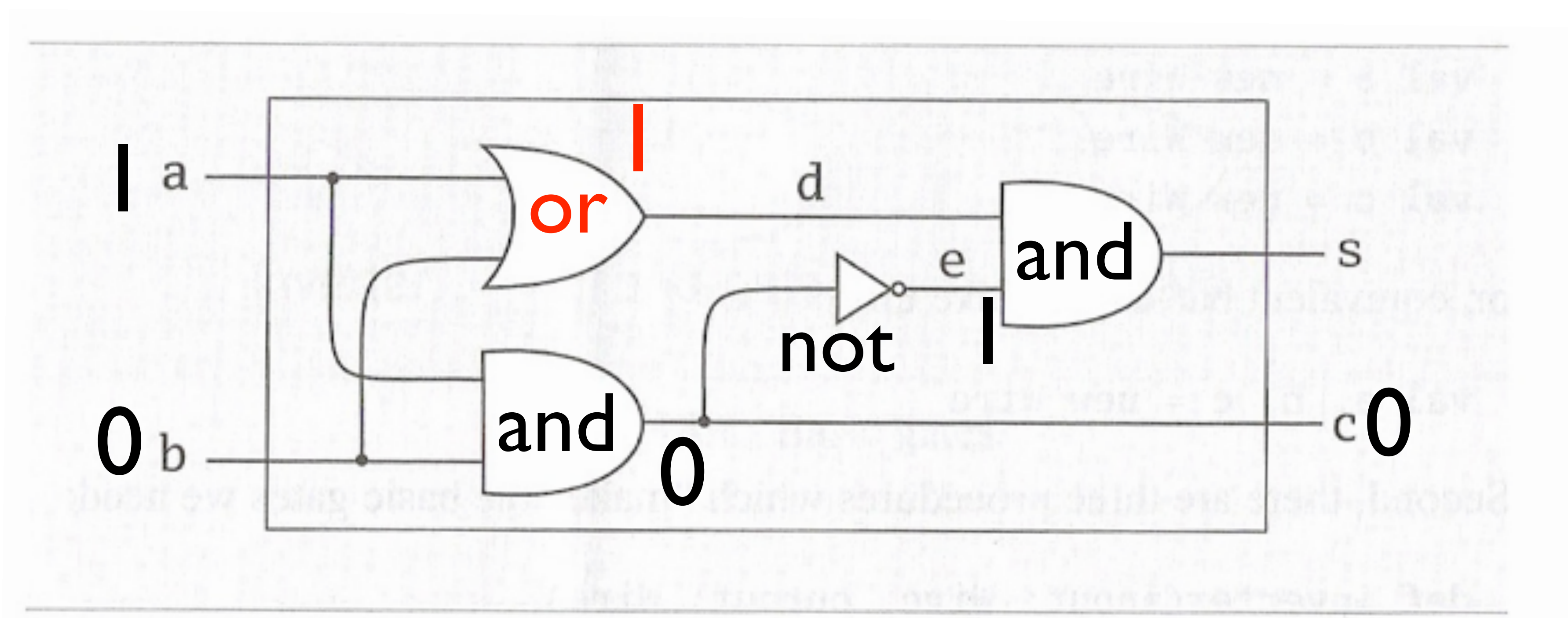


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

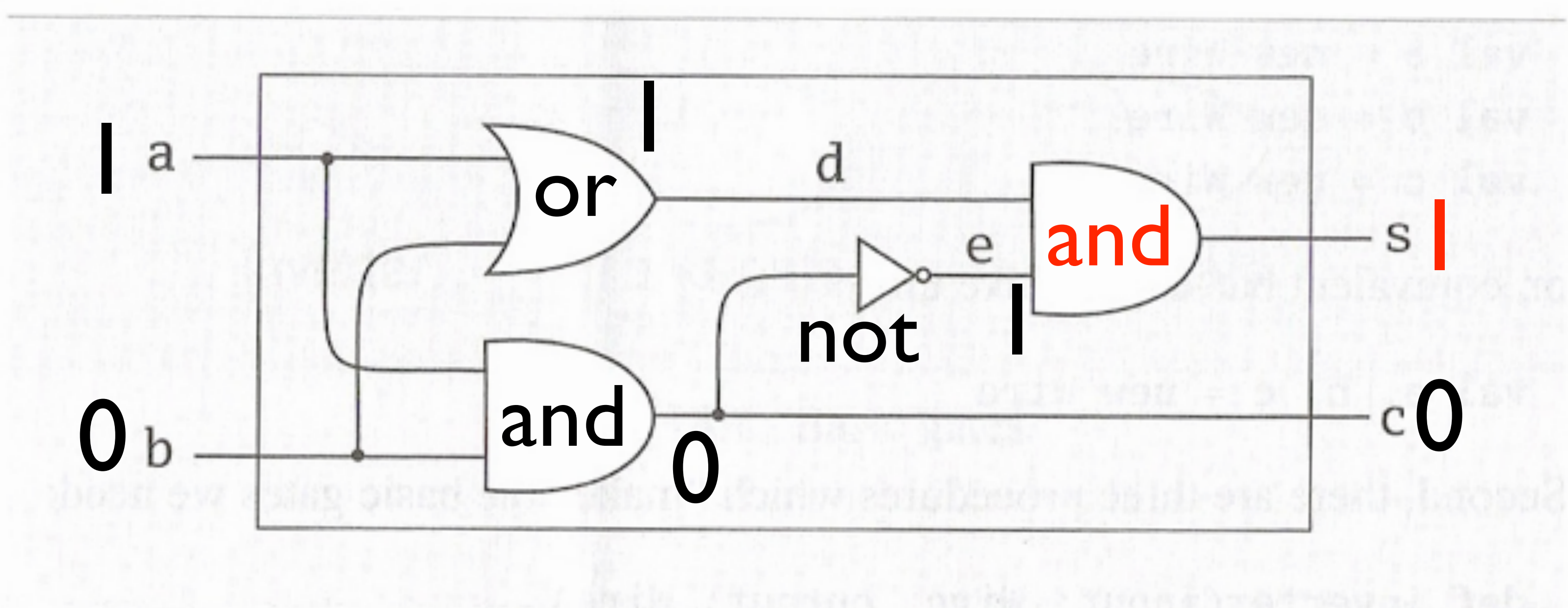


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

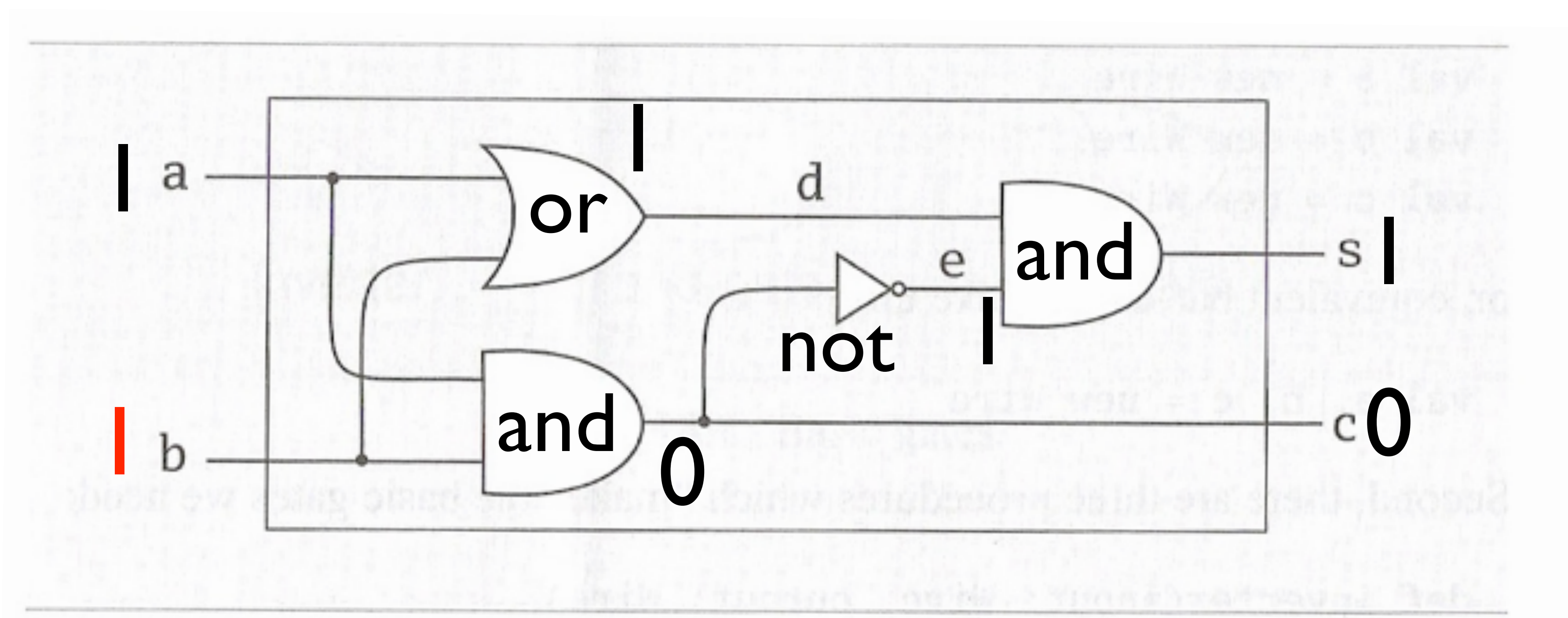


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

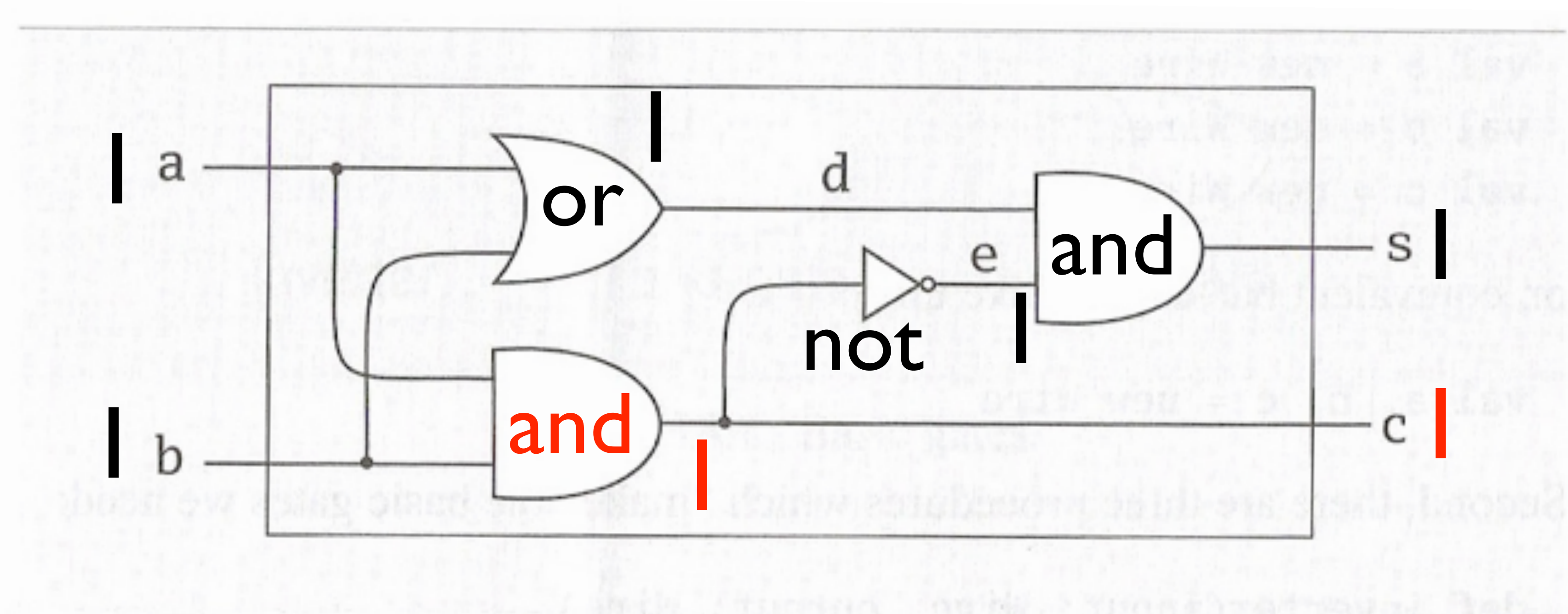


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

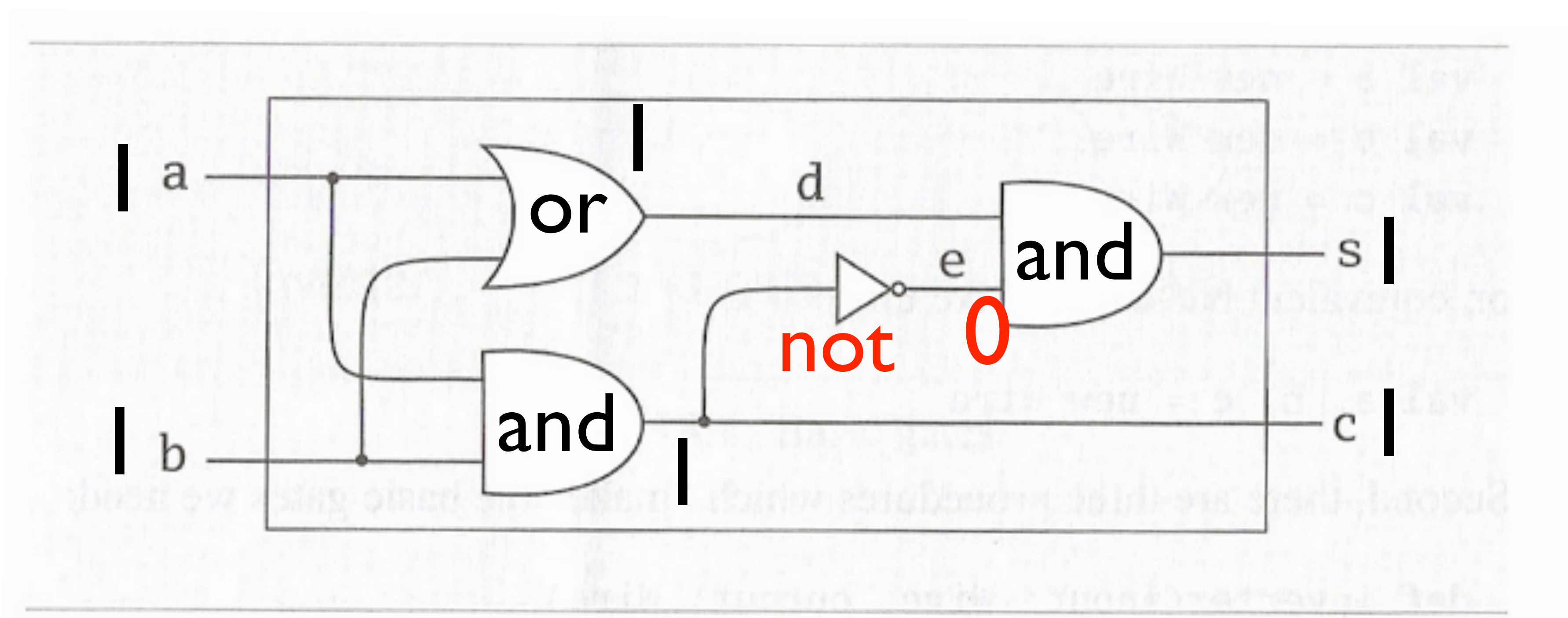


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

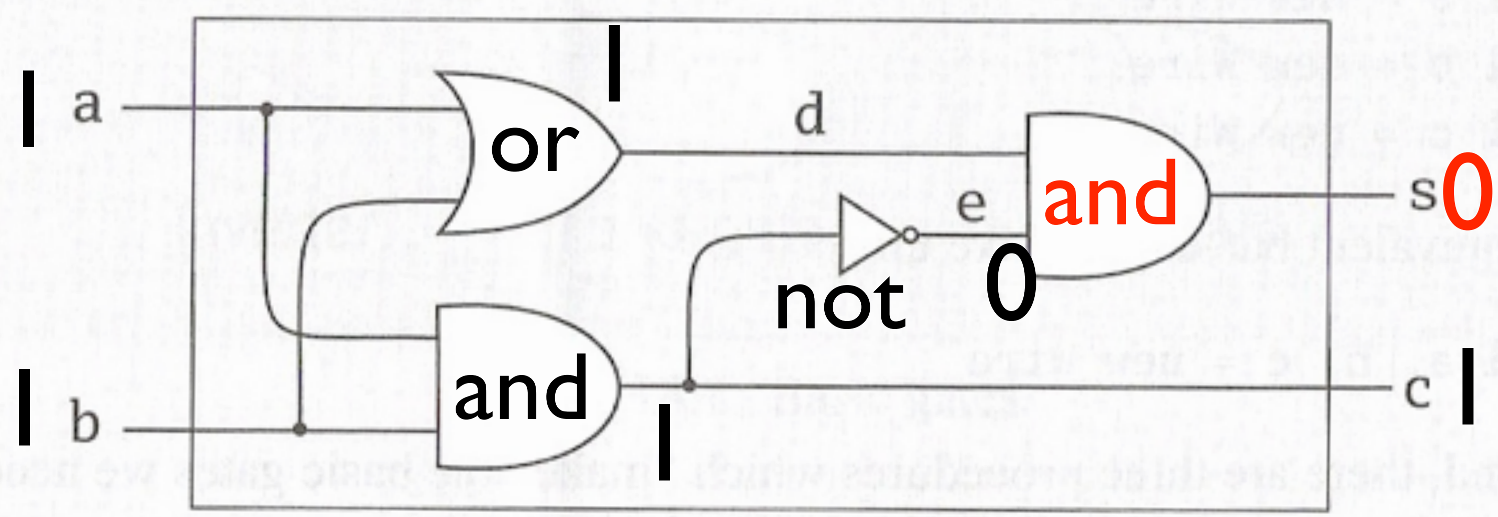


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

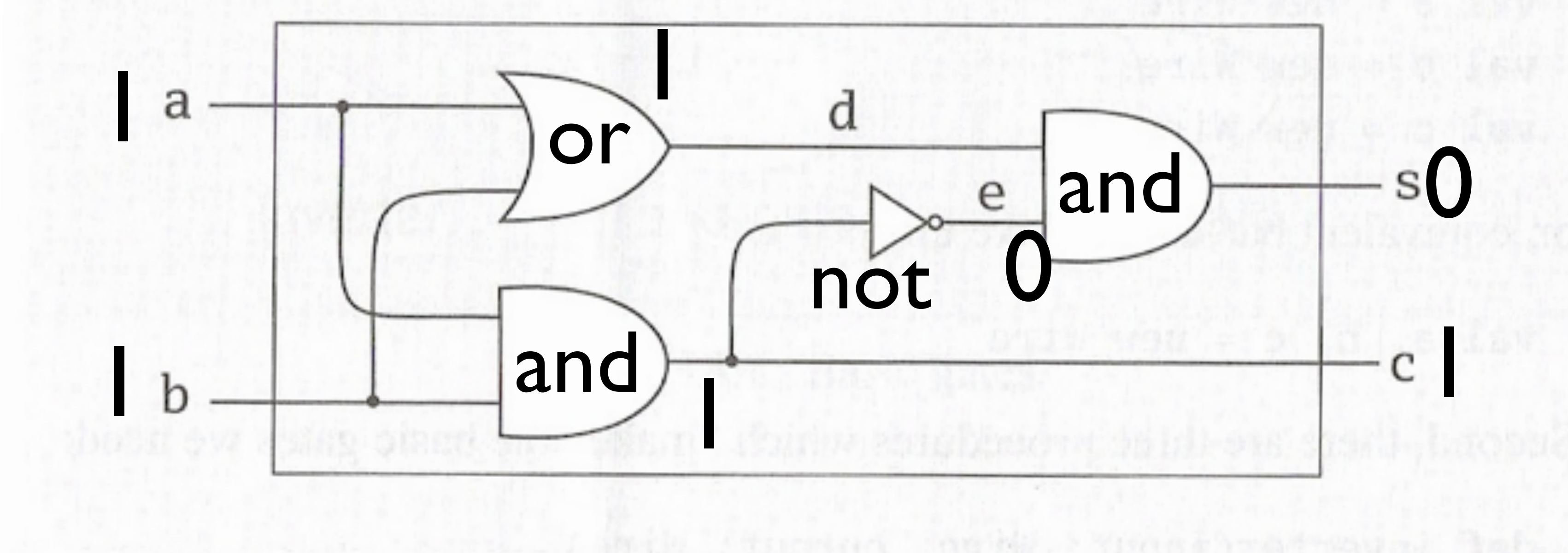


Figure 18.2 · A half-adder circuit.

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

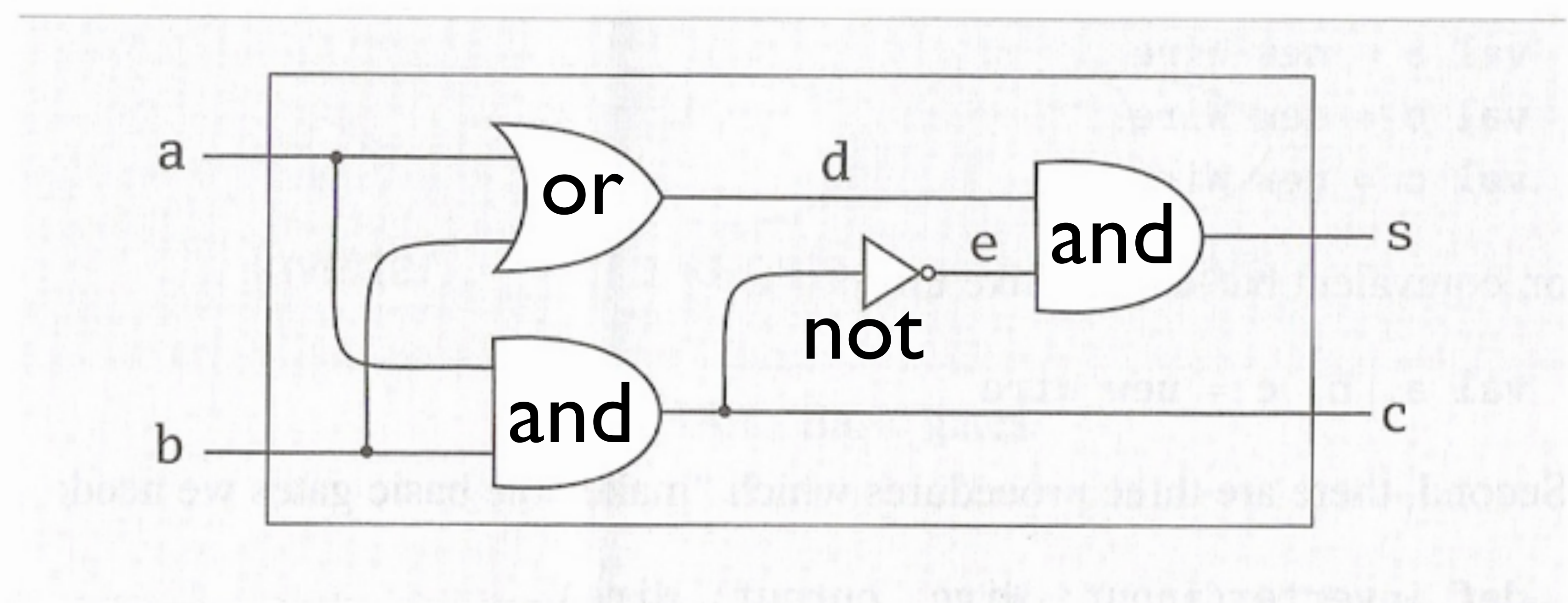


Figure 18.2 · A half-adder circuit.

```
val a,b = new Wire
val d,e = new Wire
val s,c = new Wire
orGate(a,b,d)
andGate(a,b,c)
inverter(c,e)
andGate(d,e,s)
```

Programming challenge: Circuit simulator

- The simulator should allow us to describe a circuit and to run it.

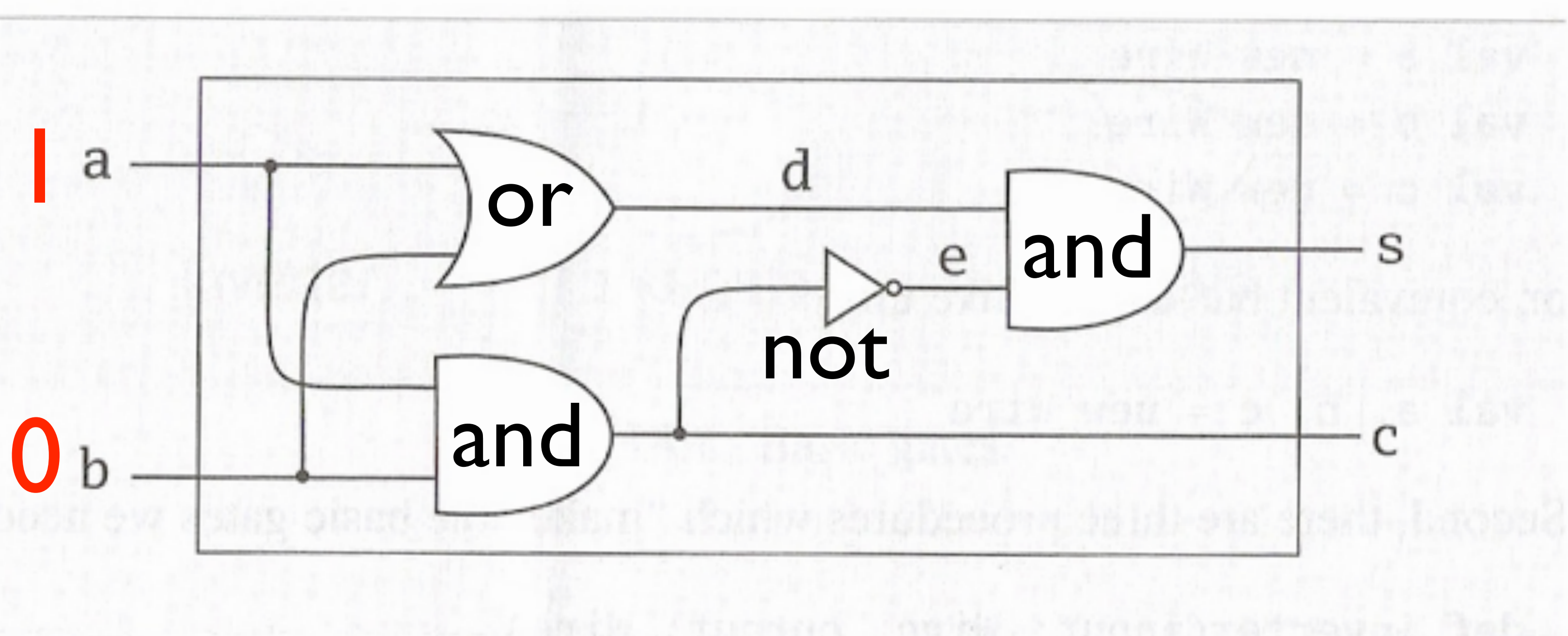


Figure 18.2 · A half-adder circuit.

```
val a,b = new Wire
val d,e = new Wire
val s,c = new Wire
orGate(a,b,d)
andGate(a,b,c)
inverter(c,e)
andGate(d,e,s)

a setSignal true
b setSignal false
run()
```

[Q] Would you model wires as observables (corresponding to counters) or observers? What about gates?

- The simulator should allow us to describe a circuit and to run it.

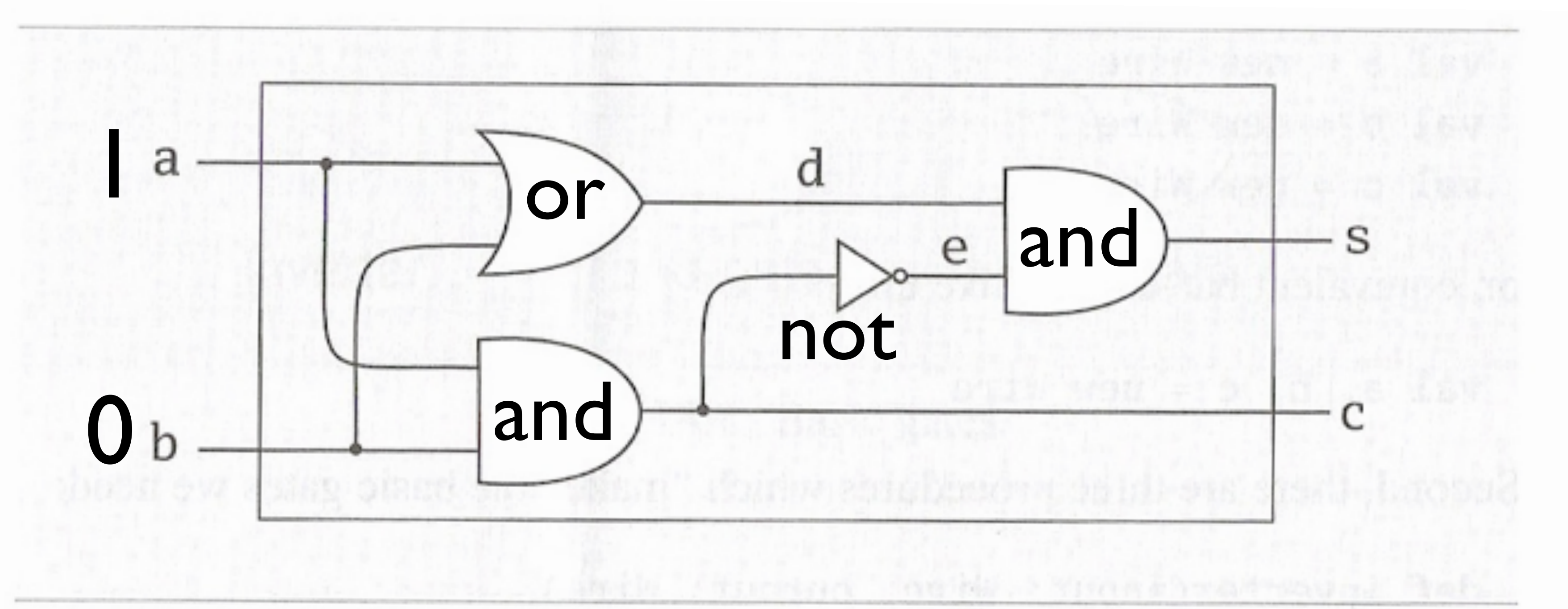


Figure 18.2 · A half-adder circuit.

```
val a,b = new Wire
val d,e = new Wire
val s,c = new Wire
orGate(a,b,d)
andGate(a,b,c)
inverter(c,e)
andGate(d,e,s)

a setSignal true
b setSignal false
run()
```

MySim

curtime = 2
workList = [(3,t_1), (5,t_2), (9,t_3)]

- MySim schedules tasks.
- Work-list algorithm.
- (3,t_1) represents a task that should run function t_1 at time 3.

MySim

curtime = 2
workList = [(3,t_1), (5,t_2), (9,t_3)]

- MySim schedules tasks.
- Work-list algorithm.
- (3,t_1) represents a task that should run function t_1 at time 3.

wire$_2$

sigVal = false
obs = [f_1]

wire$_3$

sigVal = false
obs = [g_1]

wire$_1$

sigVal = true
obs = [h_1, h_2]

- wire$_n$ uses the observer pattern.
- Logic gates are functions, and observers of wires.

MySim

curtime = 2
workList = [(3,t_1), (5,t_2), (9,t_3)]

- MySim schedules tasks.
- Work-list algorithm.
- (3,t_1) represents a task that should run function t_1 at time 3.

wire$_2$

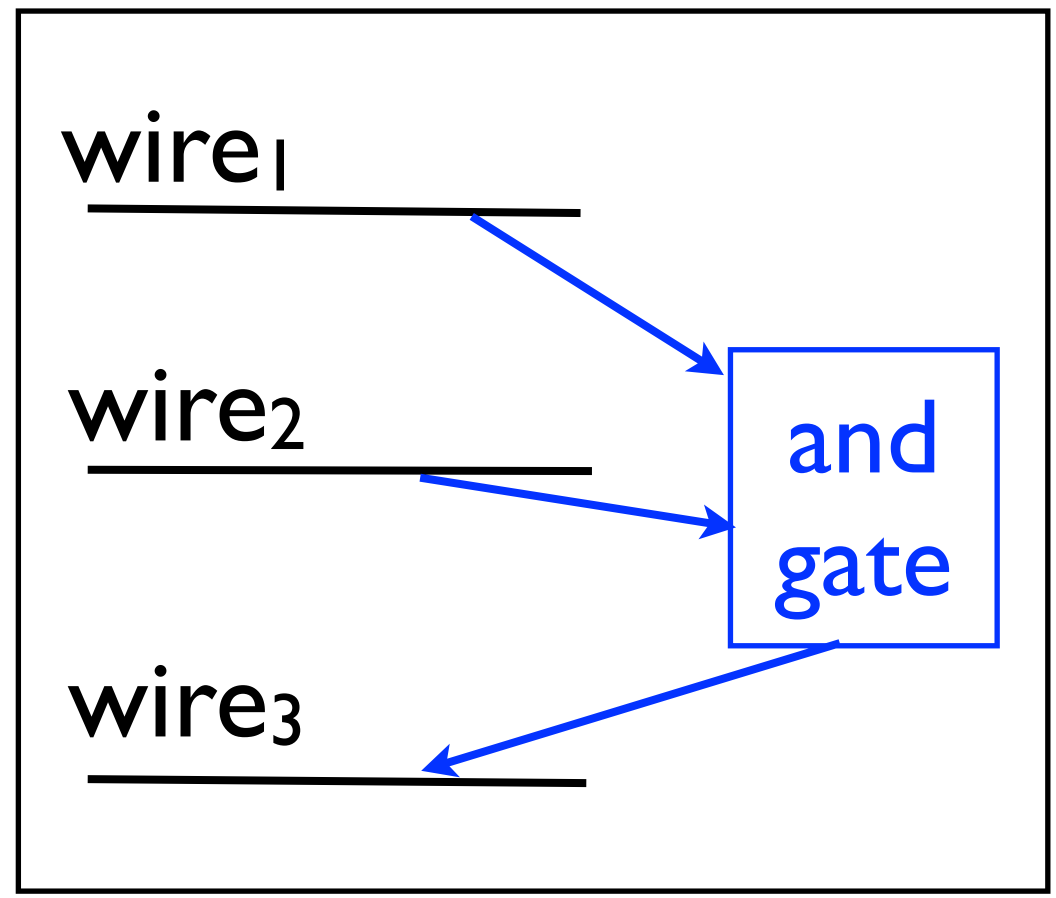
sigVal = false
obs = [f_1]

wire$_1$

sigVal = true
obs = [h_1, h_2]

wire$_3$

sigVal = false
obs = [g_1]



- wire$_n$ uses the observer pattern.
- Logic gates are functions, and observers of wires.

MySim

curtime = 3
workList = [(5,t_2), (9,t_3)]

- MySim schedules tasks.
- Work-list algorithm.
- (3,t_1) represents a task that should run function t_1 at time 3.

$wire_2$

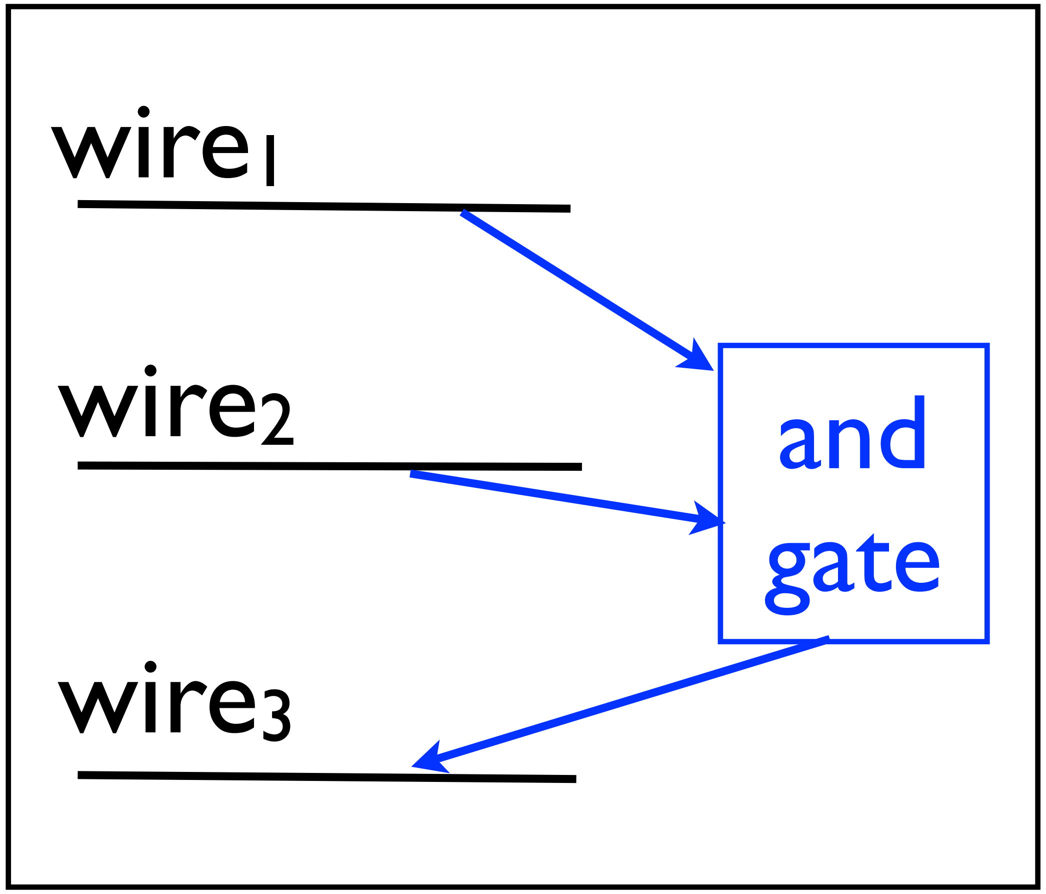
sigVal = true
obs = [f_1]

$wire_1$

sigVal = true
obs = [h_1, h_2]

$wire_3$

sigVal = false
obs = [g_1]



- $wire_n$ uses the observer pattern.
- Logic gates are functions, and observers of wires.

MySim

curtime = 3

workList = [(5,t_2), (9,t_3)]

- MySim schedules tasks.
- Work-list algorithm.
- (3,t_1) represents a task that should run function t_1 at time 3.

$wire_2$

sigVal = true

obs = [f_1]

$wire_3$

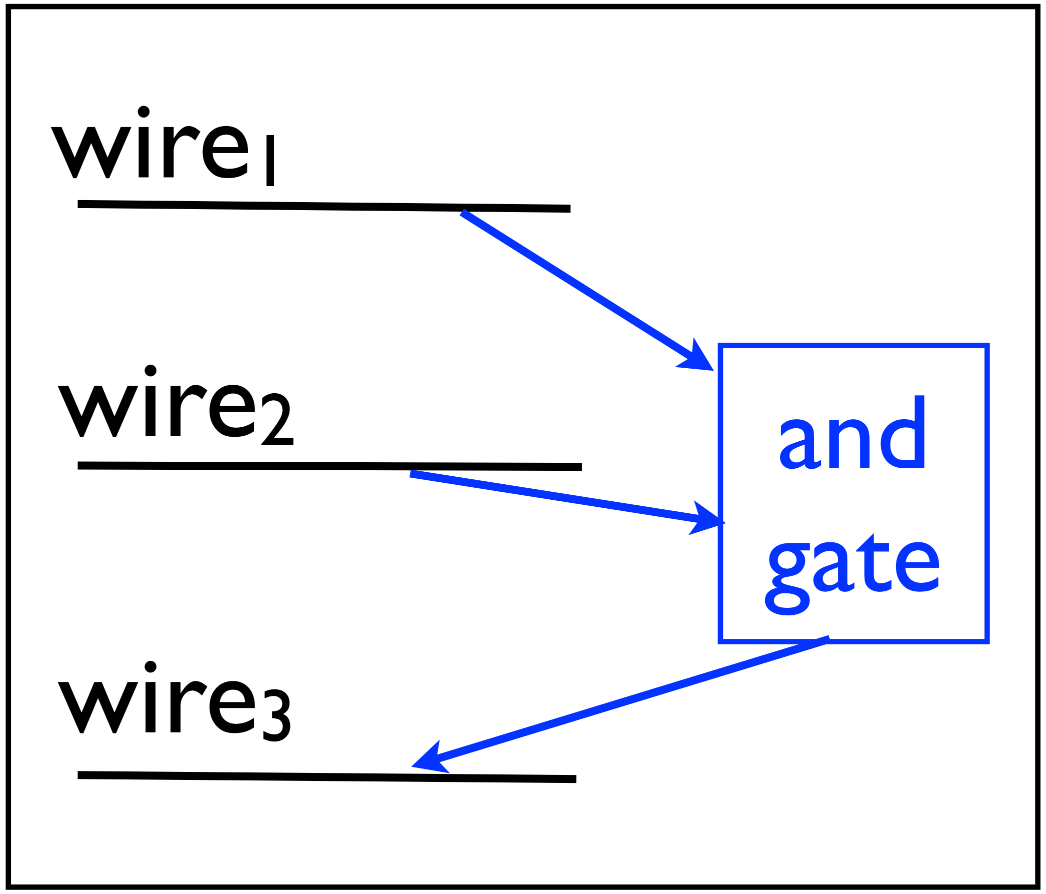
sigVal = false

obs = [g_1]

$wire_1$

sigVal = true

obs = [h_1, h_2]



- $wire_n$ uses the observer pattern.
- Logic gates are functions, and observers of wires.

MySim

curtime = 3

workList = [$(4,t_4)$, $(5,t_2)$, $(9,t_3)$]

- MySim schedules tasks.
- Work-list algorithm.
- $(3,t_1)$ represents a task that should run function t_1 at time 3.

wire$_2$

sigVal = true

obs = [f_1]

wire$_1$

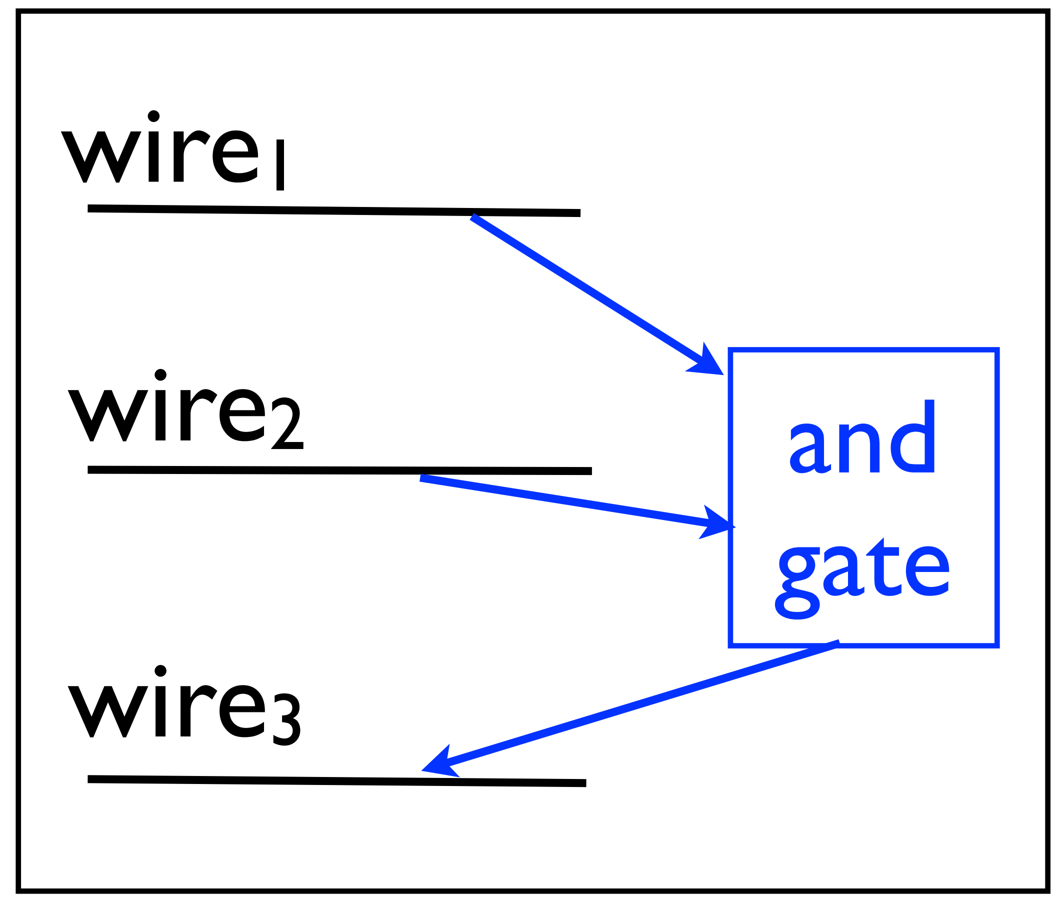
sigVal = true

obs = [h_1, h_2]

wire$_3$

sigVal = false

obs = [g_1]



- wire$_n$ uses the observer pattern.
- Logic gates are functions, and observers of wires.

MySim

curtime = 4
workList = [(5,t_2), (9,t_3)]

- MySim schedules tasks.
- Work-list algorithm.
- (3,t_1) represents a task that should run function t_1 at time 3.

wire$_2$

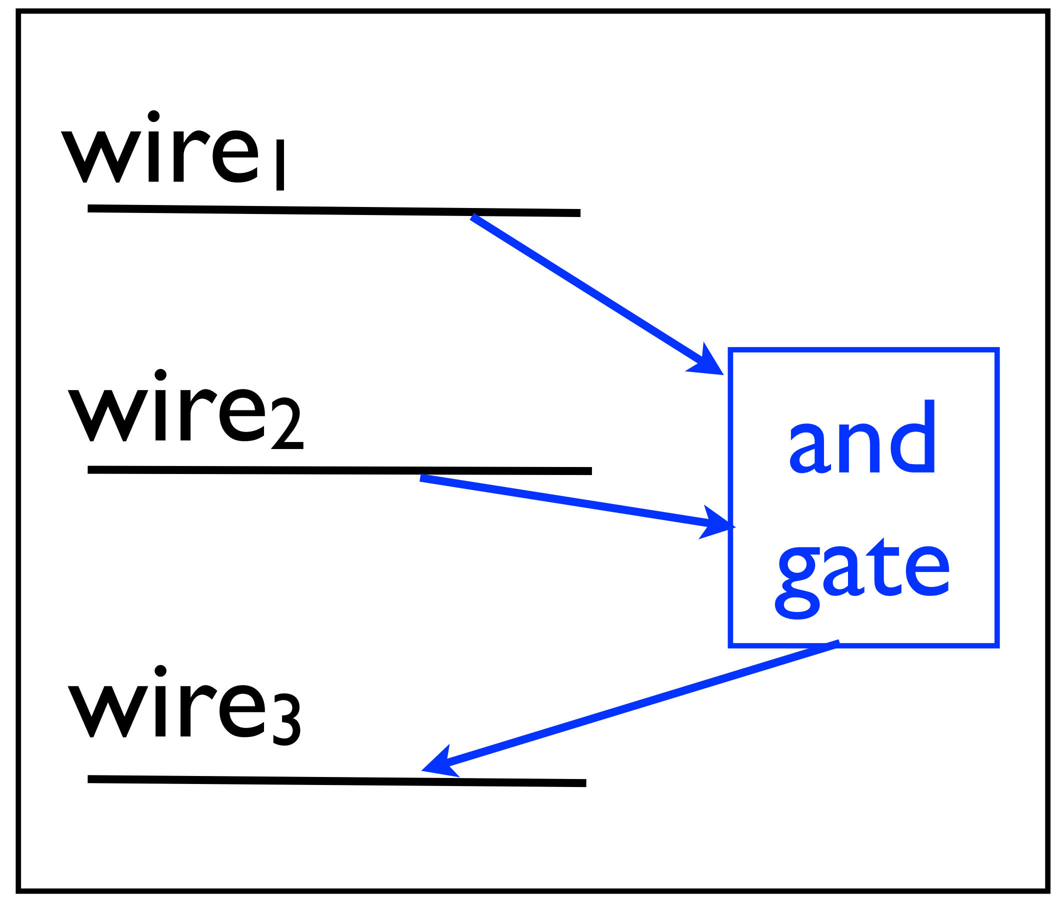
sigVal = true
obs = [f_1]

wire$_1$

sigVal = true
obs = [h_1, h_2]

wire$_3$

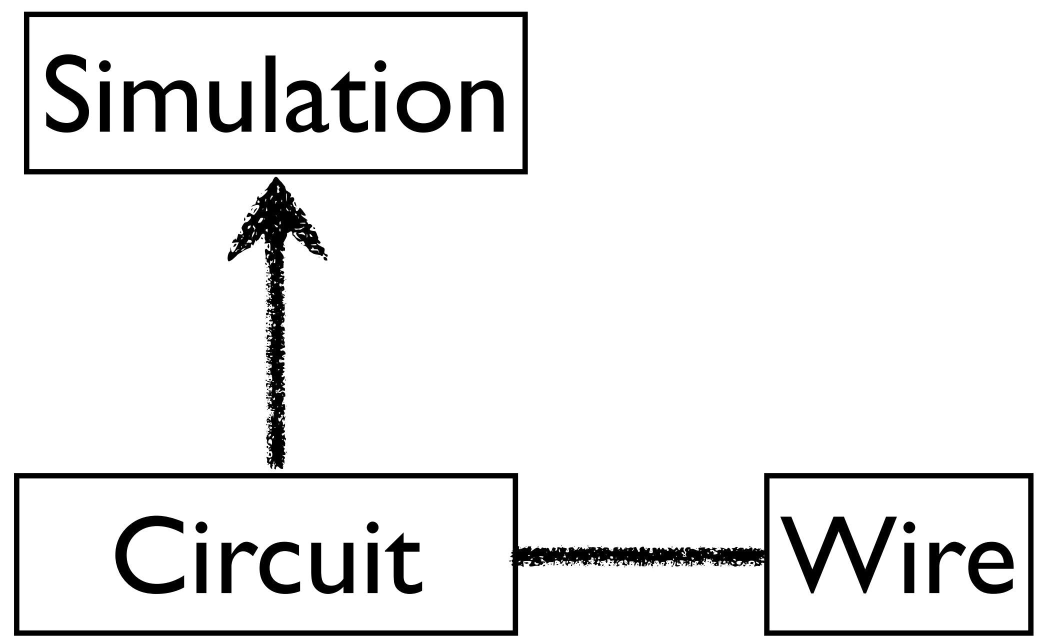
sigVal = true
obs = [g_1]



- wire$_n$ uses the observer pattern.
- Logic gates are functions, and observers of wires.

Class hierarchy

Work-list algo.

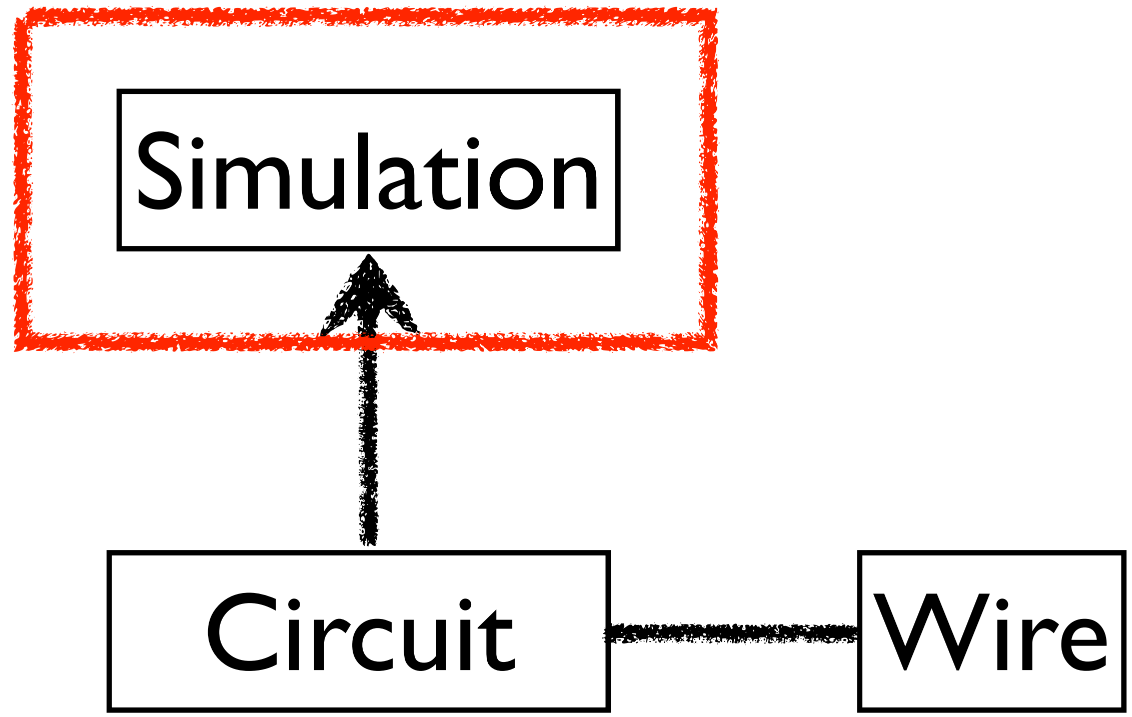


Class & methods
for wires & gates.
MySim.

Nested class.
wire$_1$, wire$_2$, ...
Observer pattern.

Class hierarchy

Work-list algo.



Class & methods
for wires & gates.
MySim.

Nested class.
$wire_1$, $wire_2$, ...
Observer pattern.

Work-list algo.

```
abstract class Simulation {
  type Work = (Int, ()=>Unit)
  var curtime = 0
  var workList: List[Work] = Nil
















}
```

Work-list algo.

Tasks sorted by time.

```
abstract class Simulation {
  type Work = (Int, ()=>Unit)
  var curtime = 0
  var workList: List[Work] = Nil

  private def insert(l:List[Work], w:Work): List[Work] =
    if (l.isEmpty || w._1 < l.head._1) w::l
    else l.head :: insert(l.tail, w)

}
```

```
abstract class Simulation {
  type Work = (Int, ()=>Unit)
  var curtime = 0
  var workList: List[Work] = Nil

  private def insert(l:List[Work], w:Work): List[Work] =
    if (l.isEmpty || w._1 < l.head._1) w::l
    else l.head :: insert(l.tail, w)

  def afterDelay(delay: Int)(block: => Unit) {
    val work = (curtime + delay, () => block)
    workList = insert(workList, work)
  }

}
```

Work-list algo.
Tasks sorted by time.

afterDelay adds a new work item.
block is a by-name parameter.

```
abstract class Simulation {
  type Work = (Int, ()=>Unit)
  var curtime = 0
  var workList: List[Work] = Nil

  private def insert(l:List[Work], w:Work): List[Work] =
    if (l.isEmpty || w._1 < l.head._1) w::l
    else l.head :: insert(l.tail, w)

  def afterDelay(delay: Int)(block: => Unit) {
    val work = (curtime + delay, () => block)
    workList = insert(workList, work)
  }

  def run() {
    afterDelay(0){ println("*** simulation start ***") }
    while (!workList.isEmpty) {



    }
  }
}
```

Work-list algo.

Tasks sorted by time.

afterDelay adds a new work item.

block is a by-name parameter.

run executes items in workList.

Work-list algo.
Tasks sorted by time.

```
abstract class Simulation {
  type Work = (Int, ()=>Unit)
  var curtime = 0
  var workList: List[Work] = Nil

  private def insert(l:List[Work], w:Work): List[Work] =
    if (l.isEmpty || w._1 < l.head._1) w::l
    else l.head :: insert(l.tail, w)

  def afterDelay(delay: Int)(block: => Unit) {
    val work = (curtime + delay, () => block)
    workList = insert(workList, work)
  }

  def run() {
    afterDelay(0){ println("*** simulation start ***") }
    while (!workList.isEmpty) {



    }
  }
}
```

[Q1] Complete the def'n of run.

afterDelay adds a new work item.
block is a by-name parameter.
run executes items in workList.

Work-list algo.
Tasks sorted by time.

```
abstract class Simulation {
  type Work = (Int, ()=>Unit)
  var curtime = 0
  var workList: List[Work] = Nil

  private def insert(l:List[Work], w:Work): List[Work] =
    if (l.isEmpty || w._1 < l.head._1) w::l
    else l.head :: insert(l.tail, w)

  def afterDelay(delay: Int)(block: => Unit) {
    val work = (curtime + delay, () => block)
    workList = insert(workList, work)
  }

  def run() {
    afterDelay(0){ println("*** simulation start ***") }
    while (!workList.isEmpty) {
      val (time, task) :: rest = workList
      workList = rest; curtime = time
      task()
    }
  }
}
```

[Q1] Complete the def'n of run.

afterDelay adds a new work item.
block is a by-name parameter.
run executes items in workList.

Work-list algo.

Tasks sorted by time.

```
abstract class Simulation {
  type Work = (Int, ()=>Unit)
  var curtime = 0
  var workList: List[Work] = Nil

  private def insert(l:List[Work], w:Work): List[Work] =
    if (l.isEmpty || w._1 < l.head._1) w::l
    else l.head :: insert(l.tail, w)

  def afterDelay(delay: Int)(block: => Unit) {
    val work = (curtime + delay, () => block)
    workList = insert(workList, work)
  }

  def run() {
    afterDelay(0){ println("*** simulation start ***") }
    while (!workList.isEmpty) {
      val (time, task) :: rest = workList
      workList = rest; curtime = time
      task()
    }
  }
}
```

[Q2] Find a resource invariant.

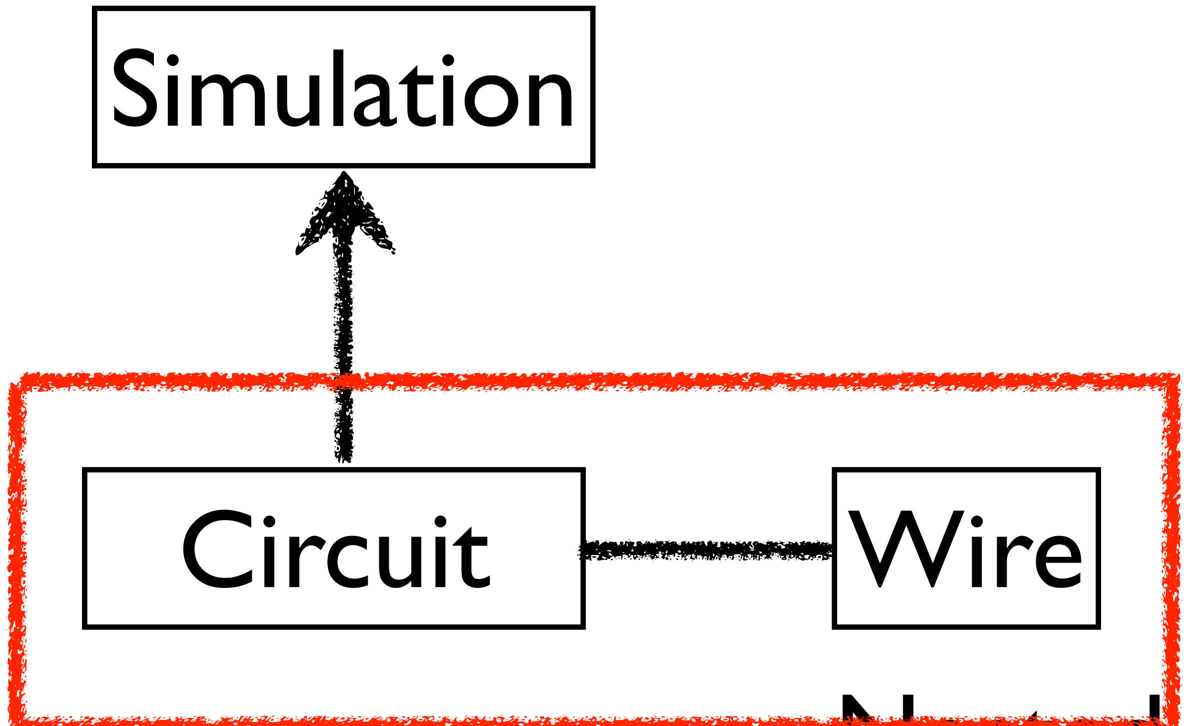
afterDelay adds a new work item.

block is a by-name parameter.

run executes items in workList.

Class hierarchy

Work-list algo.



Class & methods
for wires & gates.
MySim.

Nested class.
wire$_1$, wire$_2$, ...
Observer pattern.

```
abstract class Circuit extends Simulation {

}
```

Inheritance enables code reuse.

```
abstract class Circuit extends Simulation {
  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int
















}
```

Inheritance enables code reuse.

Delays to be specified later.

```
abstract class Circuit extends Simulation {
  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {
    private var sigVal = false

    def getSignal = sigVal
    def setSignal(s: Boolean) =
      if (s != sigVal) { sigVal = s                          }

  }










}
```

Inheritance enables code reuse.
Delays to be specified later.

Nested class.

```
abstract class Circuit extends Simulation {
  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {
    private var sigVal = false
    var observers: List[()=>Unit] = Nil
    def getSignal = sigVal
    def setSignal(s: Boolean) =
      if (s != sigVal) { sigVal = s; observers foreach (_()) }
    def addObs(o: ()=>Unit) { observers ::= o; o() }
  }

}
```

Inheritance enables code reuse.
Delays to be specified later.

Nested class.
Observer pattern.

```
abstract class Circuit extends Simulation {
  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {
    private var sigVal = false
    var observers: List[()=>Unit] = Nil
    def getSignal = sigVal
    def setSignal(s: Boolean) =
      if (s != sigVal) { sigVal = s; observers foreach (_()) }
    def addObs(o: ()=>Unit) { observers ::= o; o() }
  }

  def inverter(input: Wire, output: Wire) ...

  def andGate(a1: Wire, a2: Wire, output: Wire) ...
  def orGate(a1: Wire, a2: Wire, output: Wire) ...
}
```

Inheritance enables code reuse.
Delays to be specified later.

Nested class.
Observer pattern.

```
abstract class Circuit extends Simulation {
  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {
    private var sigVal = false
    var observers: List[()=>Unit] = Nil
    def getSignal = sigVal
    def setSignal(s: Boolean) =
      if (s != sigVal) { sigVal = s; observers foreach (_()) }
    def addObs(o: ()=>Unit) { observers ::= o; o() }
  }

  def inverter(input: Wire, output: Wire) {
    def invertAction() ...



    input addObs invertAction
  }
  def andGate(a1: Wire, a2: Wire, output: Wire) ...
  def orGate(a1: Wire, a2: Wire, output: Wire) ...
}
```

Inheritance enables code reuse.
Delays to be specified later.

Nested class.
Observer pattern.

Register an obs.,
which adds a task.

```scala
abstract class Circuit extends Simulation {
  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {
    private var sigVal = false
    var observers: List[()=>Unit] = Nil
    def getSignal = sigVal
    def setSignal(s: Boolean) =
      if (s != sigVal) { sigVal = s; observers foreach (_()) }
    def addObs(o: ()=>Unit) { observers ::= o; o() }
  }

  def inverter(input: Wire, output: Wire) {
    def invertAction() {
      val inputSig = input.getSignal
      afterDelay(InverterDelay) { output setSignal !inputSig }
    }
    input addObs invertAction
  }
  def andGate(a1: Wire, a2: Wire, output: Wire) ...
  def orGate(a1: Wire, a2: Wire, output: Wire) ...
}
```

Inheritance enables code reuse.
Delays to be specified later.

Nested class.
Observer pattern.

Register an obs.,
which adds a task.

```
abstract class Circuit extends Simulation {
  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {
    private var sigVal = false
    var observers: List[()=>Unit] = Nil
    def getSignal = sigVal
    def setSignal(s: Boolean) =
      if (s != sigVal) { sigVal = s; observers foreach (_()) }
    def addObs(o: ()=>Unit) { observers ::= o; o() }
  }

  def inverter(input: Wire, output: Wire) {
    def invertAction() {
      val inputSig = input.getSignal
      afterDelay(InverterDelay) { output setSignal !inputSig }
    }
    input addObs invertAction
  }
  def andGate(a1: Wire, a2: Wire, output: Wire) ...
  def orGate(a1: Wire, a2: Wire, output: Wire) ...
}
```

Inheritance enables code reuse.
Delays to be specified later.

Nested class.
Observer pattern.

Register an obs., which adds a task.

[Q] Complete the def'ns of andGate and orGate.

```
abstract class Circuit extends Simulation {
  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {
    private var sigVal = false
    var observers: List[Work] = Nil
    def getSignal = sigVal
    def setSignal(s: Boolean) =
                                   h (_()) }
  }

                                   nputSig }
  }
  def andGate(a1: Wire, a2: Wire, output: Wire) ...
  def orGate(a1: Wire, a2: Wire, output: Wire) ...
}
```

Inheritance enables code reuse.
Delays to be specified later.

Nested class.
Observer pattern.

er an obs.,
dds a task.

```
def andGate(a1: Wire, a2: Wire, output: Wire) {
  def andAction() = {
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (a1Sig && a2Sig)
    }
  }
  a1 addObs andAction
  a2 addObs andAction
}
```

[Q] Complete the def'ns of andGate and orGate.

Creating MySim

```
object MySim extends Circuit {
  def InverterDelay = 1
  def AndGateDelay = 3
  def OrGateDelay = 5
}
```

- Creates a singleton object MySim that inherits and specialises the abstract class Circuit.
- Simpler than the usual alternative.

Creating MySim

```
object MySim extends Circuit {
  def InverterDelay = 1
  def AndGateDelay = 3
  def OrGateDelay = 5
}
```

- Creates a singleton object MySim that inherits and specialises the abstract class Circuit.
- Simpler than the usual alternative.

```
class MySimClass extends Circuit {
  def InverterDelay = 1
  def AndGateDelay = 3
  def OrGateDelay = 5
}
val MySim = new MySimClass
```

Summary

- Two idioms for using high-order function and mutable state together.

  1. Work-list algorithm.
  2. Observer pattern.

- I presented a simpler version of circuit simulation. Look at the textbook for a full version.
- Read Chapter 18.