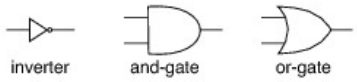
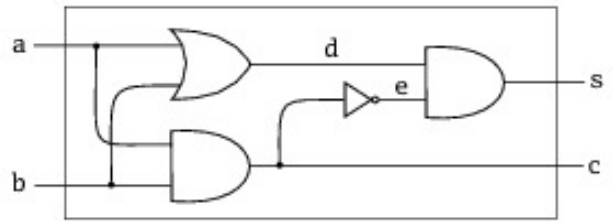
Objets et états : un simulateur de circuits

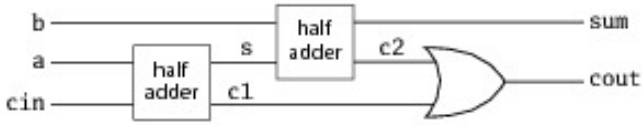
Nous allons aujourd'hui implémenter un simulateur de circuits logiques à l'aide d'objets Scala mutables et de fonctions d'ordre supérieur. Un circuit est un ensemble de portes logique de base *et* (noté `&`), *ou* (noté `|`) et *non* (noté `!`), reliés par des fils.



À tout instant, chaque fil a un état : soit 0 soit 1. Une porte logique fixe l'état de son fil de sortie en fonction du(des) état(s) de son(ses) entrée(s). Par exemple, un *demi-sommateur*, dont les entrées sont notées a et b, et dont les sorties "somme" s et "retenue" c valent respectivement $s = (a + b) \mod 2$ et $c = (a + b)/2$, est réalisé par le circuit suivant :



Un *sommateur entier* est réalisé à l'aide de deux demi-sommateurs et une porte *ou* :



Notre simulateur prendra la forme d'un mini-langage embarqué dans Scala (un *Domain Specific Language*, ou *DSL*), permettant de définir des circuits ; le changement d'état d'un fil déclenchera automatiquement le changement d'état de tous les fils qui lui sont connecté, et éventuellement l'affichage d'un message. Par exemple, on pourra simuler le comportement d'une porte NAND $t = !(x \& y)$ par le code :

```
val x, y, z, t = new Wire; // x,y entrées, z fil intermédiaire, t sortie
and(x, y, z)               // z = x & y
not(z, t)                  // t = !z
probe("nand output", t)    // afficher les changements de valeurs de t
x.sig = true               // affectation de x
y.sig = true               // affectation de y
```

Il affichera :

```
nand output = true         // résultat de l'affectation de x
nand output = false        // résultat de l'affectation de y
```

1 Simulateur instantané

1. Définir une classe `Wire` dont chaque objet représente un fil, et ayant deux champs mutables privés :
   — un boolean `signalValue` qui représente l'état du fil, de valeur par défaut `false`
   — une liste `actions: List[Action]` d'actions à effectuer quand le fil change d'état. `Action` est le type des fonctions sans arguments retournant `Unit` (donc évaluées pour leurs effets de bord). Vous définirez ce type "maison" :
   `type Action = () => Unit`
2. Définir des getter et setter personnalisés `sig` pour `signalValue` tels que si la valeur de `signalValue` change, on évalue toutes les `actions`.
3. Définir une méthode `def >>(a: Action)` qui ajoute `a` dans les `actions`, et l'évalue une première fois. Tester dans l'interprète :

```
val w = new Wire
w >> (() => println("w devient "+w.sig))
w.sig = true
w.sig = false
```

4. Définir une classe *abstraite* (vous verrez plus tard pourquoi) `CircuitSimulation` contenant les méthodes qui enregistrent dans les fils les actions correspondant aux portes logiques :
   — `def not(input: Wire, output: Wire): Unit` : si le fil `input` change d'état, alors on change l'état de `output` en conséquence ;
   — `def and(in1: Wire, in2: Wire, output: Wire): Unit` : si `i1` ou `i2` changent d'état, on change `output` en conséquence ;
   — `def or(in1: Wire, in2: Wire, output: Wire): Unit` (idem)
   — `def probe(s: String, w: Wire): Unit` : quand `w` change d'état, on affiche sa valeur précédé de la chaîne `s`.
5. Définir un objet singleton `MySimulation` qui étend `CircuitSimulation` ; tester votre code sur l'exemple de la porte NAND en première page.
6. Définir `class ComplexCircuitSimulation extends CircuitSimulation` contenant :
   — `def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire)` implémentant le demi-sommateur, et
   — `def adder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire)` implémentant le sommateur entier (voir page 1).
   Tester dans l'interprète que ces simulations répondent bien comme les circuits correspondant.

2 Simulateur avec délai

À chaque changement d'état d'un fil, notre code ci-dessus propage instantanément le changement aux fils connexes. En pratique, la propagation de l'information dans les portes logiques prend un certain temps (des nano-secondes), qui dépend de la conception de la porte en question. L'efficacité d'un circuit dépend donc de ces délais, et notre simulateur pourrait utilement afficher le temps courant (le nombre de nano-secondes écoulées depuis le début de la simulation) à chaque changement de valeur d'un fil. Modifions nos classes `Wire` et `CircuitSimulation` en conséquence.

1. Définir une classe abstraite `Simulation` qui implémente un ordonnanceur de tâche. Elle contiendra au moins les champs et méthodes :
   — `private var time = 0` : le temps courant ;

— `private var agenda: List[(Int, Action)] = List()` : la liste des paires `(t, a)` d'actions `a` qui restent à effectuer, avec leur horodatage `t` (valeur de `time` à laquelle elle devra s'exécuter). L'`agenda` doit rester trié par horodatage croissant.
— `def afterDelay(delay: Int, a: Action): Unit` : insère l'action `a` dans l'`agenda` avec l'horodatage `time + delay`.
— `def run(): Unit` : affiche `time`, puis exécute chaque action de l'`agenda` tant qu'il est non-vide.

2. Modifier `CircuitSimulation` de sorte qu'elle hérite de `Simulation`. Y ajouter trois champs entiers abstraits `notDelay`, `andDelay` et `orDelay` et modifier les méthodes `not`, `and` et `or` de façon à ce qu'elles n'effectuent leur action qu'après les délais correspondants. Modifier `probe` pour afficher le temps courant.
3. Dans l'objet `MySimulation`, définir `notDelay=1`, `andDelay=3` et `orDelay=5`. Dans la routine pricipale, rajouter des appels à `run()` après chaque affectation de fil.
4. Tester le demi-sommateur et observer l'effet des délais sur l'exécution.