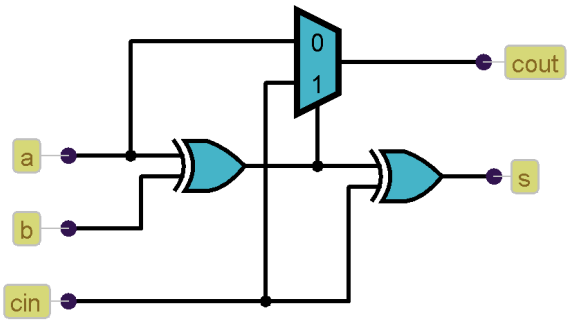
Architecture des ordinateurs

1APPSN 2022-2023

Circuits combinatoires

1. Traduction d'un schéma en langage de description SHDL

Le schéma suivant réalise une 'tranche' d'addition de 1 bit, avec retenue entrante et retenue sortante :



Écrire le module SHDL qui implémente ce circuit, avec l'interface suivant :

```
module fulladder(a, b, cin : s, cout)
```

Solution :

```
module fulladder(a,b,cin:s,cout)
   x = a*/b + /a*b
   s = x*/cin + /x*cin
   cout = /x*a + x*cin
end module
```

2. Algèbre de Boole

a. Simplifier l'expression suivante :

$$A.\bar{B}.C + A.\bar{C}.D + A.C + \bar{D}$$

Solution :

$$A.\bar{B}.C + A.\bar{C}.D + A.C + \bar{D}$$

$$= A.\bar{C}.D + A.C + \bar{D}$$

$$= A.\bar{C} + A.C + \bar{D}$$

$$= A + \bar{D}$$

b. Simplifier l'expression suivante :

$$x\bar{y}\bar{t}z + zx\bar{t} + \bar{z} + z\bar{x}$$

Solution :

$$= zx\bar{t} + \bar{z} + z\bar{x}$$

$$= zx\bar{t} + \bar{z} + \bar{x}$$

$$= z\bar{t} + \bar{z} + \bar{x}$$

$$= \bar{t} + \bar{z} + \bar{x}$$

c. Quine McCluskey

Simplifier l'expression suivante :

$$f(a,b,c,d) = \sum(4,6,8,9,10,12,14,15)$$

Solution :

étape 1			étape 2			étape 3	
4	0100	✓	4-6	01-0	✓	4-6-12-14	-1-0
8	1000	✓	4-12	-100	✓	8-10-12-14	1--0
6	0110	✓	8-9	100-			
9	1001	✓	8-10	10-0	✓		
10	1010	✓	8-12	1-00	✓		
12	1100	✓	6-14	-110	✓		
14	1110	✓	10-14	1-10	✓		
15	1111	✓	12-14	11-0	✓		
			14-15	111-			

Table des implicants premiers :

			4	6	8	9	10	12	14	15
8-9	100-	*								
14-15	111-	*			✓	✓			✓	✓
4-6-12-14	-1-0	*	✓	✓				✓		
8-10-12-14	1--0	*			✓		✓	✓		

Tous les implicants premiers sont essentiels :

$$f(a,b,c,d) = a\bar{b}\,\bar{c} + abc + b\bar{d} + a\bar{d}$$

3. Décodeurs avec enable

Construire un décodeur 2:4 avec enable, puis un décodeur 3:8 avec enable à l'aide de deux instances de décodeurs 2:4.

Solution :

```
module dec2to4(e[1..0], en : s[3..0])
   s[3] = en*e[1]*e[0]
   s[2] = en*e[1]*/e[0]
   s[1] = en*/e[1]*e[0]
   s[0] = en*/e[1]*/e[0]
end module
```

```
module dec3to8(e[2..0], en : s[7..0])
  dec2to4(e[1..0], en1 : s[7..4])
  dec2to4(e[1..0], en0 : s[3..0])
  en1 = en*e[2]
  en0 = en*/e[2]
end module
```

4. Décodeurs 7-segments hexadécimal

Construire un décodeur 4 bits pour des afficheurs hexadécimaux.

Solution :

```
module dec7segH(x[3..0] : seg[6..0])
   decoder4to16(x[3..0] : d[15..0])
   a = d[0] + d[2] + d[3] + d[5] + d[6] + d[7] + d[8] + d[9] +
d[10] + d[12] + d[14] + d[15]
   b = d[0] + d[1] + d[2] + d[3] + d[4] + d[7] + d[8] + d[9] +
d[10] + d[13]
   c = d[0] + d[1] + d[3] + d[4] + d[5] + d[6] + d[7] + d[8] +
d[9] + d[10] + d[11] + d[13]
```

```
    d = d[0] + d[2] + d[3] + d[5] + d[6] + d[8] + d[9] + d[11] +
d[12] + d[13] + d[14]
    e = d[0] + d[2] + d[6] + d[8] + d[8] + d[10] + d[11] + d[12]
+ d[13] + d[14] + d[15]
    f = d[0] + d[4] + d[5] + d[6] + d[8] + d[9] + d[10] + d[11] +
d[12] + d[14] + d[15]
    g = d[2] + d[3] + d[4] + d[5] + d[6] + d[8] + d[9] + d[10] +
d[11] + d[13] + d[14] + d[15]
    seg[6..0] = g & f & e & d & c & b & a
 end module
```

5. Encodeurs de priorité

Construire un encodeur de priorité 3:8 à l'aide d'une seule instance d'encodeur de priorité 2:4.

Solution :

```
module priority8to3(e[7..0] : s[2..0], act)
  priority4to2(e[7..4] : h[1..0], actH)
  priority4to2(e[3..0] : l[1..0], actL)
  act = actH + actL
  s[2] = actH
  s[1..0] = actH*h[1..0] + /actH*l[1..0]
end module
```

```
module priority4to2(e[3..0] : s[1..0], act)
  priority2to1(e[3..2] : h, actH)
  priority2to1(e[1..0] : l, actL)
  act = actH + actL
  s[1] = actH
  s[0] = actH*h + /actH*l
end module
```

```
module priority2to1(e[1..0] : s, act)
  s = e[1]
  act = e[1] + e[0]
end module
```

6. Incrémenteur

Construire un circuit combinatoire qui incrémente un nombre de 4 bits.

Solution :

```
module incr4(e[3..0] : s[3..0])

  s[0] = /e[0]
  s[1] = e[1]*/e[0] + /e[1]*e[0]
```

```
  s[2] = e[2]*/inv2 + /e[2]*inv2
  inv2 = e[1]*e[0]
  s[3] = e[3]*/inv3 + /e[3]*inv3
  inv3 = e[2]*e[1]*e[0]

end module
```

Circuits séquentiels

1. Registre 4 bits

Construire un registre de 4 bits. On ajoutera une entrée de validation (enable)

Solution :

```
module reg4(rst, h, en, e[3..0] : s[3..0])
   s[3..0] := e[3..0] on h, reset when rst, enabled when oe
end module
```

2. Compteur 4 bits

```
module cnt4(rst, h : a, b, c, d)
   d := /d on h, reset when rst
   c := /d*c + d*/c on h, reset when rst
   cd = c*d
   b := /cd*b + cd*/b on h reset when rst
   bcd = b*c*d
   a := /bcd*a + bcd*/a on h reset when rst
end module
```

3. Compteur 4 bits avec incrémenteur et enable

Créer un compteur 4 bits avec des bascules D et le module `incr4`. On ajoutera une sortie `max` qui indique que la valeur maximale est atteinte

Solution :

```
module cnt4(rst, h : s[3..0], max)
   incr4(s[3..0], next[3..0])
   s[3..0] := next[3..0] on h, reset when rst, enabled when en
   max = s[3]*s[2]*s[1]*s[0]
end module
```

4. Composition de compteurs

Créer un compteur 8 bits avec deux compteurs 4 bits

Solution :

```
module cnt8(rst, h, en : s[7..0], max)
    enH = maxL*en
    max = maxL*maxH
    cnt4(rst, h, enH : s[7..4], maxH)
    cnt4(rst, h, en : s[3..0], maxL)
end module
```

5. Compteur 4 bits avec remise à zéro synchrone

Solution :

```
module cnt4Z(rst, h, sclr : s[3..0])
    s[3..0] := /t[3..0]*s[3..0] + t[3..0]*/s[3..0] on h,reset
when rst
    t[0] = sclr*s[0] + /sclr*1
    t[1] = sclr*s[1] + /sclr*s[0]
    t[2] = sclr*s[2] + /sclr*s[0]*s[1]
    t[3] = sclr*s[3] + /sclr*s[0] *s[1]*s[2]
end module
```

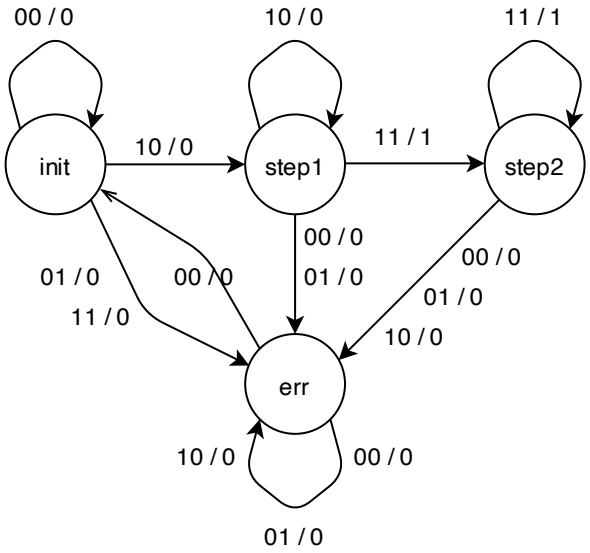
6. Enregistrement du min et du max d'une mesure

Créer un circuit qui met à jour et mémorise à chaque front d'horloge, les valeurs min et max d'une valeur sur 8 bits

```
module minmax2(rst, h, e[7..0] : max[7..0], min[7..0])
    max[7..0] := e[7..0] on h, reset when rst, enabled when
supMax
    ucmp8(e[7..0], max[7..0] : supMax, eq1)
    min[7..0] := e[7..0] on h, set when rst, enabled when infMin
    ucmp8(min[7..0], e[7..0] : infMin, eq2)
end module
```

7. Système de mise en marche de sécurité

Le schéma suivant représente le graphe d'états de type Mealy d'un système de mise en marche sécurisé. Les entrées sont a et b ; la sortie m. L'appui sur a, puis sur b déclenche la mise en marche ; toute autre séquence amène à l'état d'erreur.



Implémenter cette machine d'états en utilisant la technique « one-hot encoding » (un point de mémorisation par état).

Solution :

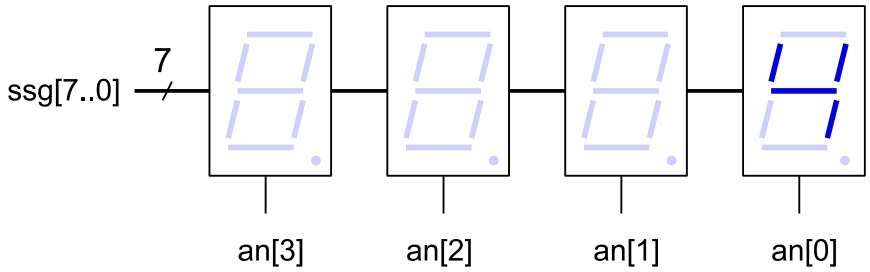
```
module security_onehot (rst, h, a, b : m)
   init := init*/a*/b + err*/a*/b on h, set when rst
   step1 := init*a*/b + step1*a*/b on h, reset when rst
   step2 := step1*a*b + step2*a*b on h, reset when rst
   err := init*(/a*b + a*b) + step1*(/a*/b + /a*b) + step2*(/a +
/b) + err*(a + b) on h, reset when rst

   m = step1*a*b + step2*a*b
end module
```

8. Utilisation d'un compteur de phases

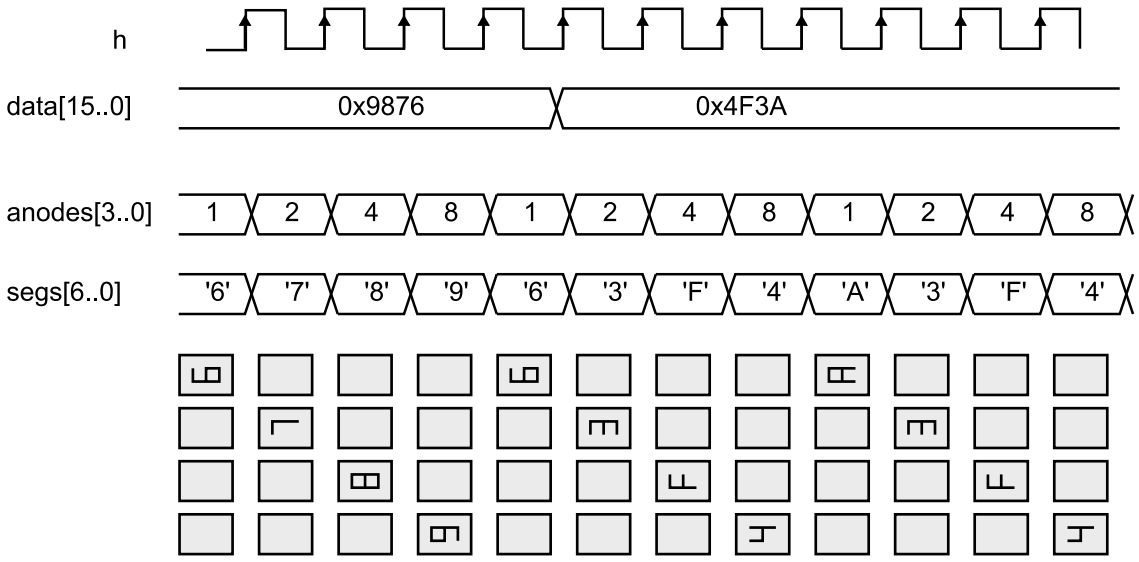
Certains circuits séquentiels suivent une séquence de phases. L'utilisation d'un compteur suivi d'un décodeur de phase est alors la méthode de conception la plus directe.

On va réaliser par cette méthode le multiplexage dans le temps des 4 afficheurs 7-segments. On rappelle le câblage des afficheurs, avec les cathodes `ssg[7..0]` toutes reliées entre elles pour chacun des 4 afficheurs afin de minimiser le nombre de signaux :



L'inconvénient, c'est qu'on ne peut alors afficher que le même chiffre sur tous les afficheurs à un moment donné ! Un multiplexage temporel est donc nécessaire pour donner l'illusion optique d'un chiffre différent sur chaque afficheur.

À chaque pas de temps, on active un seul afficheur (en agissant sur les anodes), et on place sur les cathodes `ssg[7..0]` le décodage des 4 bits de donnée qu'on souhaite afficher sur cet afficheur. En faisant circuler assez vite l'afficheur actif, on donne l'illusion d'un affichage simultané sur tous les afficheurs. Le chronogramme suivant illustre le processus :



En utilisant un compteur, un décodeur et le décodeur 7-segments `dec7segsH`, concevoir le module qui réalise ce multiplexage.

```
 module affmux(rst, h, data[15..0] : an[3..0], ssgs[6..0])
    count2(rst, h : phase[1..0])
    decoder2to4(phase [1..0] : an[3..0])
    digit[3..0] = an[0]*data[3..0] + an[1]*data[7..4] +
an[2]*data[11..8] + an[3]*data[15..12]
    dec7segH(digit[3..0] : ssgs[6..0])
 end module
```

9. Algorithme câblé : suite de Fibonacci

Créer un circuit séquentiel qui produit sur 8 bits à chaque front d'horloge un nouveau terme de la suite de Fibonacci (1, 1, 2, 3, 5, 8, 13, etc.).

Solution :

```
module fibo(rst, h : prec[7..0])
   curr[7..1] := sum[7..1] on h, reset when rst
   curr[0] := sum[0] on h, set when rst

   prec[7..1] := curr[7..1] on h, reset when rst
   prec[0] := curr[0] on h, set when rst

   adder8(curr[7..0], prec[7..0], 0 : sum[7..0], _)
end module
```

10. Algorithme cablé : calcul du PGCD

Créer un circuit séquentiel qui calcule de PGCD de deux nombres codés sur 8 bits. L'interface du circuit sera :

```
module pgcd(rst, h, a[7..0], b[7..0] : r[7..0], end)
```

L'algorithme à implémenter est le suivant :

```
x <- copie de a
y <- copie de b
tant que x <> y :
   Si x > y : x <- x – y
   Si x < y : y <- y – x
r <- x
```

Le signal end indique que la conversion est terminée.

Solution :

```
Module pgcd(rst, h, a[7..0], b[7..0] : x[7..0], end)

   init := 0 on h, set when rst
   iter := init + iter*/xEqY on h, reset when rst
   end := iter*xEqY + end on h, reset when rst

   x[7..0] := a[7..0]*init + iter*xSupY*sub[7..0] +
iter*/xSupY*x[7..0] + /iter*x[7..0] on h, reset when rst
```

```
    y[7..0] := b[7..0]*init + iter*ySupX*sub[7..0] +
iter*/ySupX*y[7..0] + /iter*y[7..0] on h, reset when rst


    sub8(arg1[7..0], arg2[7..0] : sub[7..0])
    arg1[7..0] = x[7..0]*xSupY + y[7..0]*ySupX
    arg2[7..0] = y[7..0]*xSupY + x[7..0]*ySupX


    ucmp8(x[7..0], y[7..0] : xSupY, xEqY)
    ySupX = /xSupY*/xEqY


 end module
```

11. Mémoire ROM

Utiliser une instance de mémoire ROM de 16 mots de 7 bits pour réaliser un décodeur 7 segments hexadécimal. Voir la documentation en ligne pour initialiser le contenu de la ROM.

Solution :

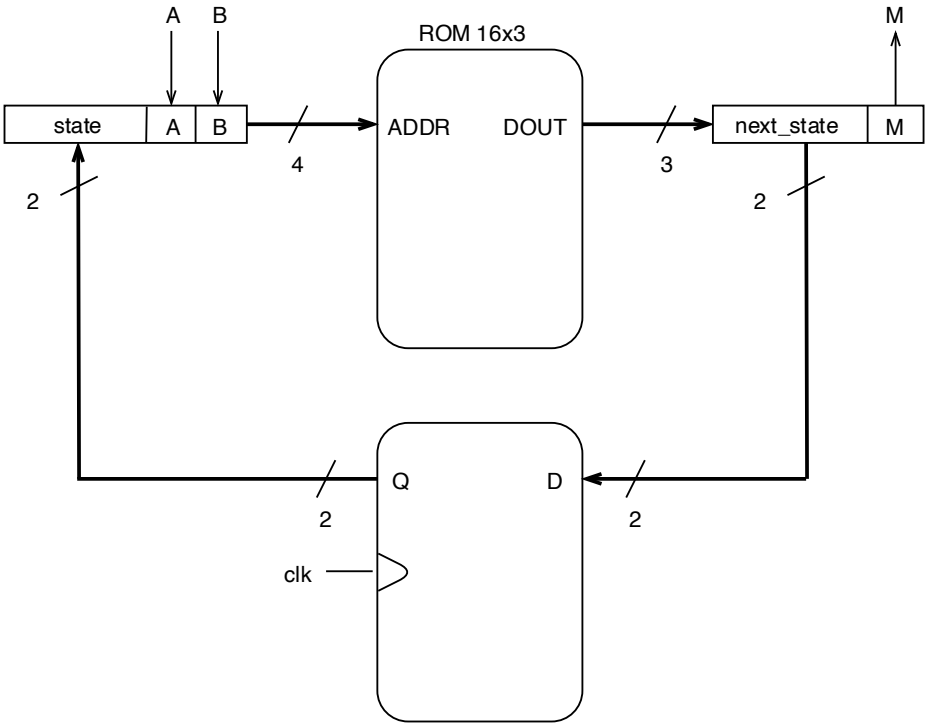
```
 module square(a[3..0] : sqrt[6..0])
    $rom(a[3..0] : sqrt[6..0])
 end module
```

Avec le fichier d'initialisation :

```
 [
    {
       "0000": "0000000",
       "0001": "0000001",
       "0010": "0000100",
       "0011": "0001001",
       …
    }
 ]
```

12. Machine de Mealy avec ROM

Le schéma suivant représente une machine de Mealy à 2 entrées A et B et une sortie M.



Convertir ce schéma en un module SHDL et trouver le contenu mémoire de la ROM pour réaliser le système de mise en marche de sécurité.

Solution :

```
Module mealy(rst, h, a, b : m)
   state[1..0] := next_state[1..0] on h, reset when rst
   $rom(state[1..0] & a & b : next_state[1..0] & m)
end module
```

Avec le fichier d'initialisation :

```
[
   {
      "0000": "000",
      "0001": "110",
      "0010": "010",
      "0011": "110",
      "0100": "110",
      "0101": "110",
      "0110": "010",
      "0111": "101",
      "1000": "110",
      "1001": "110",
      "1010": "110",
      "1011": "101",
      "1100": "000",
      "1101": "110",
      "1110": "110",
      "1111": "110"
```

```
    }
]
```