

Travail propre au sujet 1 (LINUX et JEDEC)

Votre travail va consister à réaliser un certain nombre de sous-programmes autour de ce qui a été appris en a2i11 (logique).

Une équation logique disjonctive comporte plusieurs termes séparés par des OU et ne contenant que des ET. Une telle équation sera appelée array et sera modélisée de la façon suivante :

$$y = a \cdot b + a \cdot \bar{b} \Leftrightarrow ON = \begin{pmatrix} 01 \\ 10 \end{pmatrix} \quad \text{et} \quad y = a \cdot b \cdot c + a \cdot \bar{b} \cdot \bar{c} \Leftrightarrow ON = \begin{pmatrix} 101 \\ 11\bar{X} \end{pmatrix}$$

Lorsque l'on a plusieurs sorties :

a3	a2	a1	a0	s1	s0
0	1	0	1	1	1
0	1	1	0	0	1
1	1	0	1	1	0

(ce qui n'est pas mentionné correspond à 00 en sortie)

La table de vérité ci-contre est incomplète. Elle sera modélisée par une structure C :

```
struct arraytyp {
    int nlin,ncole,ncols;
    cubetyp
    datae[MAX_LINE],datas[MAX_LINE];
}psujet={3,4,2,{5,6,13},{3,1,2}};
```

Sous Linux vous ne pouvez travailler qu'avec le compte « ge1 » avec mot de passe « stu3511 » pour avoir accès à un répertoire qui s'appelle projetut1. Ce répertoire contient un fichier source appelé logic2.c ainsi que deux répertoires et leur programmes correspondants : alliance un ensemble de logiciels français destiné à créer des ASICs et plpl destiné à programmer des PALs en partant d'un certain langage pour aboutir à un fichier JEDEC.

Tous les sujets utilisent logic2.c que l'on vous demande de lire. Il effectue un certain nombre de choses, comme la simplification d'équations [void absorbArray(struct arraytyp *p);void combineArray(struct arraytyp *p);void simplifieconsensusArray(struct arraytyp *p);] ou au contraire la mise sous forme canonique [void mintermArray(struct arraytyp *p);] Votre travail consistera à partir de ce fichier et en général à lui ajouter des sous-programmes (profitez-en pour le copier sous un autre nom : cp logic2.c monprog.c). Choisissez un nom qui vous est propre car d'autres étudiants peuvent se logger en ge1 et effacer votre travail. Il convient donc de le sauver régulièrement sur disquette : « copy monprog.c a: »

alliance est une suite logicielle française d'outils pour réaliser des ASICs (Très gros circuits programmables).

plpl est une suite logicielle permettant de programmer des PALs à partir d'un langage comme VHDLmais moins puissant).

Travail à réaliser

1°) Écrire void readarray(struct arraytyp *p); qui demande à l'utilisateur d'entrer un array sous la forme qui vous intéresse. La difficulté de ce sous programme est d'entrer les X, c'est à dire que quand l'utilisateur entre un X il faut le coder sur 2 bits non conjoints.

Remarques :

- Vous avez tous ce sous-programme à réaliser et rien ne vous empêche de vous mettre à plusieurs pour le réaliser.

- Vous pouvez entreprendre la suite sans faire ce sous-programme puisque comme on vous le montre dans logic2.c il est possible de déclarer une variable avec initialisation.

Vous avez plusieurs exemples dans logic2.c :

```
//p1={11,4,1,{0,7,1,12,3,2,5,13,8,15,10},{1,1,1,1,1,1,1,1,1,1,1}};  
p1={4,4,2,{0,12,259,8},{3,257,1,0}},  
p2={4,4,2,{0,12,3,8},{3,1,1,2}},  
p3={5,4,2,{0,12,259,3,1036},{3,257,1,1,1}},  
pcomb={6,3,2,{0,1,3,5,7,6},{3,3,3,2,3,3}};
```

2°) Il vous faut réaliser un ensemble de procédures permettant de réaliser un fichier JEDEC à partir de vos arraytyp. Pour cela déplacer-vous dans le répertoire plpl/examples et visualisez l' exemple prtut1.pld.

Lancer ensuite : `../bin/plc -i prtut1.pld -o prtut1.eq1` qui produit un fichier prtut1.eq1.

Lancez ensuite : `../bin/optimize -i prtut1.eq1 -o prtut1.eq2` qui produit un fichier prtut1.eq2 d' équations simplifiées.

Lancez ensuite : `../bin/jm -i prtut1.eq2 -o prtut1.jed` qui produira le fichier JEDEC.

On vous demande de comprendre la syntaxe des fichiers *.pld pour les produire directement à partir de vos arraytyp. Vous prendrez les conventions sur les noms des broches que vous voulez.

3°) Si le coeur vous en dit, lisez le fichier *.eq1 parce que c' est ce genre de fichier que vous allez produire dans cette deuxième partie (optionnelle). Les numéros qui apparaissent sont les numéros de pattes du composant sauf s' ils sont précédés d' un # auquel cas il s' agit d' une valeur numérique, en général 0 ou 1.

Une fois compris, on vous demande de réaliser un sous-programme qui réalise directement ce genre de fichier à partir d' un arraytyp.

Travail propre au sujet 2 (Équations VHDL)

Votre travail va consister à réaliser un certain nombre de sous-programmes autour de ce qui a été appris en a2i11 (logique).

Une équation logique disjonctive comporte plusieurs termes séparés par des OU et ne contenant que des ET. Une telle équation sera appelée array et sera modélisée de la façon suivante :

$$y = a \cdot b + a \cdot \bar{b} \Leftrightarrow ON = \begin{pmatrix} 01 \\ 10 \end{pmatrix} \quad \text{et} \quad y = a \cdot b \cdot c + a \cdot \bar{b} \cdot \bar{c} \Leftrightarrow ON = \begin{pmatrix} 101 \\ 11\bar{X} \end{pmatrix}$$

Lorsque l'on a plusieurs sorties :

a3	a2	a1	a0	s1	s0
0	1	0	1	1	1
0	1	1	0	0	1
1	1	0	1	1	0

(ce qui n'est pas mentionné correspond à 00 en sortie)

La table de vérité ci-contre est incomplète. Elle sera modélisée par une structure C :

```
struct arraytyp {
    int nlin,ncole,ncols;
    cubetyp
    datae[MAX_LINE],datas[MAX_LINE];
}psujet={3,4,2,{5,6,13},{3,1,2}};
```

Sous Linux vous ne pouvez travailler qu'avec le compte « ge1 » avec mot de passe « stu3511 » pour avoir accès à un répertoire qui s'appelle projetut1. Ce répertoire contient un fichier source appelé logic2.c ainsi que deux répertoires et leur programmes correspondants : alliance un ensemble de logiciels français destiné à créer des ASICs et plpl destiné à programmer des PALs en partant d'un certain langage pour aboutir à un fichier JEDEC.

Tous les sujets utilisent logic2.c que l'on vous demande de lire. Il effectue un certain nombre de choses, comme la simplification d'équations [void absorbArray(struct arraytyp *p);void combineArray(struct arraytyp *p);void simplifieconsensusArray(struct arraytyp *p);] ou au contraire la mise sous forme canonique [void mintermArray(struct arraytyp *p);] Votre travail consistera à partir de ce fichier et en général à lui ajouter des sous-programmes (profitez-en pour le copier sous un autre nom : cp logic2.c monprog.c). Choisissez un nom qui vous est propre car d'autres étudiants peuvent se logger en ge1 et effacer votre travail. Il convient donc de le sauver régulièrement sur disquette : « copy monprog.c a: »

alliance est une suite logicielle française d'outils pour réaliser des ASICs (Très gros circuits programmables).

plpl est une suite logicielle permettant de programmer des PALs à partir d'un langage comme VHDLmais moins puissant).

Travail à réaliser

1°) Ecrire void readarray(struct arraytyp *p); qui demande à l'utilisateur d'entrer un array sous la forme qui vous intéresse. La difficulté de ce sous programme est d'entrer les X, c'est à dire que quand l'utilisateur entre un X il faut le coder sur 2 bits non conjoints.

Remarques :

- Vous avez tous ce sous-programme à réaliser et rien ne vous empêche de vous mettre à plusieurs pour le réaliser.

- Vous pouvez entreprendre la suite sans faire ce sous-programme puisque comme on vous le montre dans logic2.c il est possible de déclarer une variable avec initialisation.

Vous avez plusieurs exemples dans logic2.c :

```
//p1={11,4,1,{0,7,1,12,3,2,5,13,8,15,10},{1,1,1,1,1,1,1,1,1,1,1,1}};
p1={4,4,2,{0,12,259,8},{3,257,1,0}},
p2={4,4,2,{0,12,3,8},{3,1,1,2}},
p3={5,4,2,{0,12,259,3,1036},{3,257,1,1,1}},
pcomb={6,3,2,{0,1,3,5,7,6},{3,3,3,2,3,3}};
```

2°) Le but est de générer à partir d'un arraytyp deux fichiers : une description comportementale en VHDL et un fichier de pattern. Nous allons vous apprendre dans un instant à générer du VHDL quand au fichier de pattern, cela ressemble aux vecteurs tests. Le binaire asimut permet de vérifier que le fichier VHDL correspond bien aux spécifications du fichier de patterns.

Le fichier VHDL produit aura forcément l'extension *.vbe.

Le prototype du sous-programme : void array2VHDL(struct arraytyp p);

Le fichier VHDL "demo.vbe" correspondant s'écrit (tout ce qui commence par -- est un commentaire non obligatoire) :

```
ENTITY demo IS
PORT (
    a          : in BIT_VECTOR(0 TO 3) ;-- 4 entrees
    s          : out BIT_VECTOR(0 TO 1) ;-- 2 sorties
    vdd, vss   : in BIT -- les alimentations
);
END demo;
ARCHITECTURE mydemo OF demo IS
BEGIN
    s(1) <= (NOT(a(3)) AND a(2) AND NOT(a(1)) AND a(0)) OR --0101
            (a(3) AND a(2) AND NOT(a(1)) AND a(0));      --1101
    s(0) <= (NOT(a(3)) AND a(2) AND NOT(a(1)) AND a(0)) OR --0101
            (NOT(a(3)) AND a(2) AND a(1) AND NOT(a(0)));  --0110
END mydemo;
```

Les parenthèses sont obligatoires en VHDL. L'idée est d'en ouvrir une pour chaque cube de la fermer d'ajouter un OU puis d'en ouvrir une pour le cube suivant ...

Un essai de compilation du fichier généré pourra se faire avec : alliance/bin/asimut -b -c demo (si le fichier est demo.vbe)

3°) Le fichier de patterns "demo.pat" produit aura forcément l'extension .pat (toute ligne commençant par un # est un commentaire) :

```
in vdd;
in vss;
in a(3 to 0);
out s(1 to 0);
begin
# a=0101, we expect value 3 on s
pat_1 : 1 0 0101 ?11 ;
# a=0110, we expect value 1 on s
pat_2 : 1 0 0110 ?01 ;
# a=13, we expect value 2 on s
pat_3 : 1 0 1101 ?10 ;
# a=7, we still expect bin 0 on s
pat_4 : 1 0 0111 ?00 ;
end;
```

Le prototype du sous-programme : void array2Pat(struct arraytyp p);

4°) La vérification de l'ensemble se fait avec : (vous êtes supposés être à la racine de alliance :

```
alliance/bin/asimut -b demo demo specif
```

le premier `demo` est en fait `demo.vbe`, le deuxième est `demo.pat` et `specif` contiendra le résultat du test (`specif.pat`)

En cas de bon fonctionnement vous verrez à l'écran :

```
.....  
linking ...  
executing ...  
###----- processing pattern 0 -----###  
###----- processing pattern 1 -----###  
###----- processing pattern 2 -----###  
###----- processing pattern 3 -----###
```

Si aucun message d'erreur n'est donné, cela signifie que les patterns attendus sont au rendez-vous donc que tout fonctionne comme prévu.

Travail propre au sujet 3 (VHDL et with select when)

Votre travail va consister à réaliser un certain nombre de sous-programmes autour de ce qui a été appris en a2i11 (logique).

Une équation logique disjonctive comporte plusieurs termes séparés par des OU et ne contenant que des ET. Une telle équation sera appelée array et sera modélisée de la façon suivante :

$$y = a \cdot b + a \cdot \bar{b} \Leftrightarrow ON = \begin{pmatrix} 01 \\ 10 \end{pmatrix} \quad \text{et} \quad y = a \cdot b \cdot c + a \cdot \bar{b} \cdot c \Leftrightarrow ON = \begin{pmatrix} 101 \\ 11\bar{X} \end{pmatrix}$$

Lorsque l'on a plusieurs sorties :

a3	a2	a1	a0	s1	s0
0	1	0	1	1	1
0	1	1	0	0	1
1	1	0	1	1	0

(ce qui n'est pas mentionné correspond à 00 en sortie)

La table de vérité ci-contre est incomplète. Elle sera modélisée par une structure C :

```
struct arraytyp {
    int nlin,ncole,ncols;
    cubetyp
    datae[MAX_LINE],datas[MAX_LINE];
}psujet={3,4,2,{5,6,13},{3,1,2}};
```

Sous Linux vous ne pouvez travailler qu'avec le compte « ge1 » avec mot de passe « stu3511 » pour avoir accès à un répertoire qui s'appelle projetut1. Ce répertoire contient un fichier source appelé logic2.c ainsi que deux répertoires et leur programmes correspondants : alliance un ensemble de logiciels français destiné à créer des ASICs et plpl destiné à programmer des PALs en partant d'un certain langage pour aboutir à un fichier JEDEC.

Tous les sujets utilisent logic2.c que l'on vous demande de lire. Il effectue un certain nombre de choses, comme la simplification d'équations [void absorbArray(struct arraytyp *p);void combineArray(struct arraytyp *p);void simplifieconsensusArray(struct arraytyp *p);] ou au contraire la mise sous forme canonique [void mintermArray(struct arraytyp *p);] Votre travail consistera à partir de ce fichier et en général à lui ajouter des sous-programmes (profitez-en pour le copier sous un autre nom : cp logic2.c monprog.c). Choisissez un nom qui vous est propre car d'autres étudiants peuvent se logger en ge1 et effacer votre travail. Il convient donc de le sauver régulièrement sur disquette : « copy monprog.c a : »

alliance est une suite logicielle française d'outils pour réaliser des ASICs (Très gros circuits programmables).

plpl est une suite logicielle permettant de programmer des PALs à partir d'un langage comme VHDL mais moins puissant).

Travail à réaliser

1°) Écrire void readarray(struct arraytyp *p); qui demande à l'utilisateur d'entrer un array sous la forme qui vous intéresse. La difficulté de ce sous programme est d'entrer les X, c'est à dire que quand l'utilisateur entre un X il faut le coder sur 2 bits non conjoints.

Remarques :

- Vous avez tous ce sous-programme à réaliser et rien ne vous empêche de vous mettre à plusieurs pour le réaliser.
- Vous pouvez entreprendre la suite sans faire ce sous-programme puisque comme on

vous le montre dans logic2.c il est possible de déclarer une variable avec initialisation.

Vous avez plusieurs exemples dans logic2.c :

```
//p1={11,4,1,{0,7,1,12,3,2,5,13,8,15,10},{1,1,1,1,1,1,1,1,1,1,1,1}};  
p1={4,4,2,{0,12,259,8},{3,257,1,0}},  
p2={4,4,2,{0,12,3,8},{3,1,1,2}},  
p3={5,4,2,{0,12,259,3,1036},{3,257,1,1,1}},  
pcomb={6,3,2,{0,1,3,5,7,6},{3,3,3,2,3,3}};
```

2°) Le but est de générer à partir d'un arraytyp deux fichiers : une description comportementale en VHDL et un fichier de pattern. Nous allons vous apprendre dans un instant à générer du VHDL quand au fichier de pattern, cela ressemble aux vecteurs tests. Le binaire asimut permet de vérifier que le fichier VHDL correspond bien aux spécifications du fichier de patterns.

Le fichier VHDL produit aura forcément l'extension *.vbe.

Le prototype du sous-programme :

```
void array2VHDLWithSelect(struct arraytyp p);
```

Le fichier VHDL "demo.vbe" correspondant s'écrit (tout ce qui commence par -- est un commentaire non obligatoire) :

```
ENTITY demo IS  
PORT(  
    a          : in BIT_VECTOR(3 DOWNT0 0) ;-- 4 entrees  
    s          : out BIT_VECTOR(1 DOWNT0 0) ;-- 2 sorties  
    vdd, vss   : in BIT -- les alimentations  
);  
END demo;  
ARCHITECTURE mydemo OF demo IS  
BEGIN  
    with a select -style with select when  
        s <= "11" when "0101", -- premiere ligne  
            "01" when "0110", -- deuxieme ligne  
            "10" when "1101", -- troisieme ligne  
            "00" when others;  
END mydemo;
```

Un essai de compilation du fichier généré pourra se faire avec :

```
alliance/bin/asimut -b -c demo
```

(si le fichier est demo.vbe)

3°) Le fichier de patterns "demo.pat" produit aura forcément l'extension .pat (toute ligne commençant par un # est un commentaire) :

```
in vdd;  
in vss;  
in a(3 to 0);  
out s(1 to 0);  
begin  
# a=0101, we expect value 3 on s  
pat_1 : 1 0 0101 ?11 ;  
# a=0110, we expect value 1 on s  
pat_2 : 1 0 0110 ?01 ;  
# a=13, we expect value 2 on s  
pat_3 : 1 0 1101 ?10 ;  
# a=7,we still expect bin 0 on s  
pat_4 : 1 0 0111 ?00 ;  
end;
```

Le prototype du sous-programme : `void array2Pat(struct arraytyp p);`

4°) La vérification de l'ensemble se fait avec (vous êtes supposés être à la racine de alliance) :

```
alliance/bin/asimut -b demo demo specif
```

le premier demo est en fait demo.vbe, le deuxième est demo.pat et specif contiendra le résultat du test (specif.pat)

En cas de bon fonctionnement vous verrez à l'écran :

```
.....  
linking ...  
executing ...  
###----- processing pattern 0 -----###  
###----- processing pattern 1 -----###  
###----- processing pattern 2 -----###  
###----- processing pattern 3 -----###
```

Si aucun message d'erreur n'est donné, cela signifie que les patterns attendus sont au rendez-vous donc que tout fonctionne comme prévu.

Travail propre au sujet 4 (VHDL et when else)

Votre travail va consister à réaliser un certain nombre de sous-programmes autour de ce qui a été appris en a2i11 (logique).

Une équation logique disjonctive comporte plusieurs termes séparés par des OU et ne contenant que des ET. Une telle équation sera appelée array et sera modélisée de la façon suivante :

$$y = a \cdot b + a \cdot \bar{b} \Leftrightarrow ON = \begin{pmatrix} 01 \\ 10 \end{pmatrix} \quad \text{et} \quad y = a \cdot b \cdot c + a \cdot \bar{b} \cdot \bar{c} \Leftrightarrow ON = \begin{pmatrix} 101 \\ 11\bar{X} \end{pmatrix}$$

Lorsque l'on a plusieurs sorties :

a3	a2	a1	a0	s1	s0
0	1	0	1	1	1
0	1	1	0	0	1
1	1	0	1	1	0

(ce qui n'est pas mentionné correspond à 00 en sortie)

La table de vérité ci-contre est incomplète. Elle sera modélisée par une structure C :

```
struct arraytyp {
    int nlin,ncole,ncols;
    cubetyp
    datae[MAX_LINE],datas[MAX_LINE];
}psujet={3,4,2,{5,6,13},{3,1,2}};
```

Sous Linux vous ne pouvez travailler qu'avec le compte « ge1 » avec mot de passe « stu3511 » pour avoir accès à un répertoire qui s'appelle projetut1. Ce répertoire contient un fichier source appelé logic2.c ainsi que deux répertoires et leur programmes correspondants : alliance un ensemble de logiciels français destiné à créer des ASICs et plpl destiné à programmer des PALs en partant d'un certain langage pour aboutir à un fichier JEDEC.

Tous les sujets utilisent logic2.c que l'on vous demande de lire. Il effectue un certain nombre de choses, comme la simplification d'équations [void absorbArray(struct arraytyp *p);void combineArray(struct arraytyp *p);void simplifieconsensusArray(struct arraytyp *p);] ou au contraire la mise sous forme canonique [void mintermArray(struct arraytyp *p);] Votre travail consistera à partir de ce fichier et en général à lui ajouter des sous-programmes (profitez-en pour le copier sous un autre nom : cp logic2.c monprog.c). Choisissez un nom qui vous est propre car d'autres étudiants peuvent se logger en ge1 et effacer votre travail. Il convient donc de le sauver régulièrement sur disquette : « copy monprog.c a: »

alliance est une suite logicielle française d'outils pour réaliser des ASICs (Très gros circuits programmables).

plpl est une suite logicielle permettant de programmer des PALs à partir d'un langage comme VHDLmais moins puissant).

Travail à réaliser

1°) Écrire void readarray(struct arraytyp *p); qui demande à l'utilisateur d'entrer un array sous la forme qui vous intéresse. La difficulté de ce sous programme est d'entrer les X, c'est à dire que quand l'utilisateur entre un X il faut le coder sur 2 bits non conjoints.

Remarques :

- Vous avez tous ce sous-programme à réaliser et rien ne vous empêche de vous mettre à plusieurs pour le réaliser.
- Vous pouvez entreprendre la suite sans faire ce sous-programme puisque comme on

vous le montre dans logic2.c il est possible de déclarer une variable avec initialisation.

Vous avez plusieurs exemples dans logic2.c :

```
//p1={11,4,1,{0,7,1,12,3,2,5,13,8,15,10},{1,1,1,1,1,1,1,1,1,1,1,1}};
p1={4,4,2,{0,12,259,8},{3,257,1,0}},
p2={4,4,2,{0,12,3,8},{3,1,1,2}},
p3={5,4,2,{0,12,259,3,1036},{3,257,1,1,1}},
pcomb={6,3,2,{0,1,3,5,7,6},{3,3,3,2,3,3}};
```

2°) Le but est de générer à partir d'un arraytyp deux fichiers : une description comportementale en VHDL et un fichier de pattern. Nous allons vous apprendre dans un instant à générer du VHDL quand au fichier de pattern, cela ressemble aux vecteurs tests. Le binaire asimut permet de vérifier que le fichier VHDL correspond bien aux spécifications du fichier de patterns.

Le fichier VHDL produit aura forcément l'extension *.vbe.

Le prototype du sous-programme :

```
void array2VHDLWhenElse(struct arraytyp p);
```

Le fichier VHDL "demo.vbe" correspondant s'écrit (tout ce qui commence par -- est un commentaire non obligatoire) :

```
ENTITY demo IS
PORT(
  a          : in BIT_VECTOR(3 DOWNTO 0) ;-- 4 entrees
  s          : out BIT_VECTOR(1 DOWNTO 0) ;-- 2 sorties
  vdd, vss   : in BIT -- les alimentations
);
END demo;
ARCHITECTURE mydemo OF demo IS
BEGIN
  -- style when else
  s <= "11" when a="0101" else -- premiere ligne
      "01" when a="0110" else -- deuxieme ligne
      "10" when a="1101" else -- troisieme ligne
      "00";
END mydemo;
```

Un essai de compilation du fichier généré pourra se faire avec :

```
alliance/bin/asimut -b -c demo
```

(si le fichier est demo.vbe)

3°) Le fichier de patterns "demo.pat" produit aura forcément l'extension .pat (toute ligne commençant par un # est un commentaire) :

```
in vdd;
in vss;
in a(3 to 0);
out s(1 to 0);
begin
# a=0101, we expect value 3 on s
pat_1 : 1 0 0101 ?11 ;
# a=0110, we expect value 1 on s
pat_2 : 1 0 0110 ?01 ;
# a=13, we expect value 2 on s
pat_3 : 1 0 1101 ?10 ;
# a=7,we still expect bin 0 on s
pat_4 : 1 0 0111 ?00 ;
end;
```

Le prototype du sous-programme : void array2Pat(struct arraytyp p);

4°) La vérification de l'ensemble se fait avec : (vous êtes supposés être à la racine de alliance :

```
alliance/bin/asimut -b demo demo specif
```

le premier demo est en fait demo.vbe, le deuxième est demo.pat et specif contiendra le résultat du test (specif.pat)

En cas de bon fonctionnement vous verrez à l'écran :

```
.....  
linking ...  
executing ...  
###----- processing pattern 0 -----###  
###----- processing pattern 1 -----###  
###----- processing pattern 2 -----###  
###----- processing pattern 3 -----###
```

Si aucun message d'erreur n'est donné, cela signifie que les patterns attendus sont au rendez-vous donc que tout fonctionne comme prévu.

Travail propre au sujet 5 (VHDL NETLIST en ET/OU CMOS)

Votre travail va consister à réaliser un certain nombre de sous-programmes autour de ce qui a été appris en a2i11 (logique).

Une équation logique disjonctive comporte plusieurs termes séparés par des OU et ne contenant que des ET. Une telle équation sera appelée array et sera modélisée de la façon suivante :

$$y = a \cdot b + a \cdot \bar{b} \Leftrightarrow ON = \begin{pmatrix} 01 \\ 10 \end{pmatrix} \quad \text{et} \quad y = a \cdot b \cdot c + a \cdot \bar{b} \cdot c \Leftrightarrow ON = \begin{pmatrix} 101 \\ 11\bar{X} \end{pmatrix}$$

Lorsque l'on a plusieurs sorties :

a3	a2	a1	a0	s1	s0
0	1	0	1	1	1
0	1	1	0	0	1
1	1	0	1	1	0

(ce qui n'est pas mentionné correspond à 00 en sortie)

La table de vérité ci-contre est incomplète. Elle sera modélisée par une structure C :

```
struct arraytyp {
    int nlin,ncole,ncols;
    cubetyp
    datae[MAX_LINE],datas[MAX_LINE];
}psujet={3,4,2,{5,6,13},{3,1,2}};
```

Sous Linux vous ne pouvez travailler qu'avec le compte « ge1 » avec mot de passe « stu3511 » pour avoir accès à un répertoire qui s'appelle progetut1. Ce répertoire contient un fichier source appelé logic2.c ainsi que deux répertoires et leur programmes correspondants : alliance un ensemble de logiciels français destiné à créer des ASICs et plpl destiné à programmer des PALs en partant d'un certain langage pour aboutir à un fichier JEDEC.

Tous les sujets utilisent logic2.c que l'on vous demande de lire. Il effectue un certain nombre de choses, comme la simplification d'équations [void absorbArray(struct arraytyp *p);void combineArray(struct arraytyp *p);void simplifieconsensusArray(struct arraytyp *p);] ou au contraire la mise sous forme canonique [void mintermArray(struct arraytyp *p);] Votre travail consistera à partir de ce fichier et en général à lui ajouter des sous-programmes (profitez-en pour le copier sous un autre nom : cp logic2.c monprog.c). Choisissez un nom qui vous est propre car d'autres étudiants peuvent se logger en ge1 et effacer votre travail. Il convient donc de le sauver régulièrement sur disquette : « copy monprog.c a : »

alliance est une suite logicielle française d'outils pour réaliser des ASICs (Très gros circuits programmables).

plpl est une suite logicielle permettant de programmer des PALs à partir d'un langage comme VHDLmais moins puissant).

Travail à réaliser

1°) Écrire void readarray(struct arraytyp *p); qui demande à l'utilisateur d'entrer un array sous la forme qui vous intéresse. La difficulté de ce sous programme est d'entrer les X, c'est à dire que quand l'utilisateur entre un X il faut le coder sur 2 bits non conjoints.

Remarques :

- Vous avez tous ce sous-programme à réaliser et rien ne vous empêche de vous mettre à plusieurs pour le réaliser.
- Vous pouvez entreprendre la suite sans faire ce sous-programme puisque comme on vous le montre dans logic2.c il est possible de déclarer une variable avec initialisation.

Vous avez plusieurs exemples dans logic2.c :

```
//p1={11,4,1,{0,7,1,12,3,2,5,13,8,15,10},{1,1,1,1,1,1,1,1,1,1,1,1}};  
p1={4,4,2,{0,12,259,8},{3,257,1,0}},  
p2={4,4,2,{0,12,3,8},{3,1,1,2}},  
p3={5,4,2,{0,12,259,3,1036},{3,257,1,1,1}},  
pcomb={6,3,2,{0,1,3,5,7,6},{3,3,3,2,3,3}};
```

2°) Le but est de générer à partir d'un arraytyp deux fichiers : une description structurelle en VHDL et un fichier de pattern. Nous allons vous apprendre dans un instant à générer du VHDL quand au fichier de pattern, cela ressemble aux vecteurs tests. Le binaire asimut permet de vérifier que le fichier VHDL correspond bien aux spécifications du fichier de patterns.

Le fichier VHDL produit aura forcément l'extension *.vst.

Le prototype du sous-programme :

```
void array2VHDLNetListEtOu(struct arraytyp p);
```

Le fichier VHDL "demo.vst" correspondant (tout ce qui commence par -- est un commentaire non obligatoire) se présente comme à la page suivante où l'on tente d'expliquer la relation entre VHDL et un schéma.

Il existe dans la librairie sclib ("alliance/cells/sclib") un certain nombre de fichiers intéressants comme :

o3_y.vbe (OU à trois entrées) a3_y.vbe (ET à 3 entrées) qui peuvent être utilisés. Il suffit de prendre l'interface PORT(...) et de la transformer en COMPONENT xxx_y port(...) END COMPONENT. Pour vous faciliter le travail vous disposez dans logic2.c de deux sous-programmes appelés nand et schemaEnNands capables de générer un programme VHDL netlist en NAND). Créer deux sous programmes void et(meme param) et ou(meme param) pour réaliser ensuite un schéma en ET/OU (il n'y a pas grand chose à changer).

Un essai de compilation du fichier demo.vst généré pourra se faire avec :

```
export MBK_IN_LO=vst  
export MBK_CATA_LIB=alliance/cells/sclib  
alliance/bin/asimut -c demo
```

Les deux export permettent de positionner des variables d'environnement. A noter que ces exports ne sont à réaliser qu'une seule fois : toute utilisation ultérieure de asimut se fera directement sans redéfinir les variables d'environnement. Le premier export demande à asimut d'utiliser des fichiers de type vst. La deuxième variable demande à asimut d'utiliser une librairie CMOS standard appelée sclib.

3°) Le fichier de patterns "demo.pat" produit aura forcément l'extension .pat (toute ligne commençant par un # est un commentaire) :

```
in vdd;  
in vss;  
in a(3 to 0);  
out s(1 to 0);  
begin  
# a=0101, we expect value 3 on s  
pat_1 : 1 0 0101 ?11 ;  
# a=0110, we expect value 1 on s  
pat_2 : 1 0 0110 ?01 ;  
# a=13, we expect value 2 on s  
pat_3 : 1 0 1101 ?10 ;  
# a=7,we still expect bin 0 on s  
pat_4 : 1 0 0111 ?00 ;
```

```
end;
```

Le prototype du sous-programme : `void array2Pat(struct arraytyp p);`

4°) La vérification de l'ensemble se fait avec : (vous êtes supposés être à la racine de alliance :

```
alliance/bin/asimut demo demo specif
```

le premier demo est en fait `demo.vst`, le deuxième est `demo.pat` et `specif` contiendra le résultat du test (`specif.pat`)

En cas de bon fonctionnement vous verrez à l'écran :

```
.....  
linking ...  
executing ...  
###----- processing pattern 0 -----###  
###----- processing pattern 1 -----###  
###----- processing pattern 2 -----###  
###----- processing pattern 3 -----###
```

Si aucun message d'erreur n'est donné, cela signifie que les patterns attendus sont au rendez-vous donc que tout fonctionne comme prévu.

```
ENTITY demo IS
```

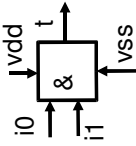
```
PORT(
  a
  s
  vdd, vss
);
END demo;
```

```
ARCHITECTURE mydemo OF demo IS
```

```
-- on commence par declarer les composants que l'on utilise :
```

```
COMPONENT a2_y port( -- ET à deux entrées
```

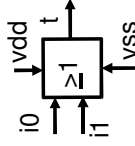
```
  i0 : in BIT;
  i1 : in BIT;
  t : out BIT;
  vdd : in BIT;
  vss : in BIT
);
```



```
END COMPONENT;
```

```
COMPONENT o2_y port( -- OU à deux entrées
```

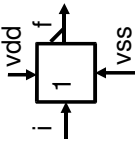
```
  i0 : in BIT;
  i1 : in BIT;
  t : out BIT;
  vdd : in BIT;
  vss : in BIT
);
```



```
END COMPONENT;
```

```
COMPONENT n1_y port( --inverseur
```

```
  i : in BIT;
  f : out BIT;
  vdd : in BIT;
  vss : in BIT
);
```



```
END COMPONENT;
```

```
-- on declare ensuite les variables qui nous serviront de connexions
```

```
SIGNAL na : BIT_VECTOR(3 DOWNTO 0); -- NON des entrées
```

```
SIGNAL is0,is1,is2,is3,is4,is5,is6 : BIT;
```

```
-- on décrit ensuite le schéma ET/OU appelé NETLIST
```

```
BEGIN -- on complètement toutes les entrées a(i) dans na(i)
```

```
c0:n1_y PORT MAP(i=>a(0),f=>na(0),vdd=>vdd,vss=>vss);
```

```
c1:n1_y PORT MAP(i=>a(1),f=>na(1),vdd=>vdd,vss=>vss);
```

```
c2:n1_y PORT MAP(i=>a(2),f=>na(2),vdd=>vdd,vss=>vss);
```

```
c3:n1_y PORT MAP(i=>a(3),f=>na(3),vdd=>vdd,vss=>vss);
```

```
-- on décrit s(1) Eh OUI, il se simplifie !!!
```

```
c4:a2_y PORT MAP(i0=>na(1),i1=>a(0),t=>is0,vdd=>vdd,vss=>vss);
```

```
c5:a2_y PORT MAP(i0=>a(2),i1=>is0,t=>s(1),vdd=>vdd,vss=>vss);
```

```
--on décrit s(0) Lui ne se simplifie pas
```

```
c6:a2_y PORT MAP(i0=>a(0),i1=>na(1),t=>is1,vdd=>vdd,vss=>vss);
```

```
c7:a2_y PORT MAP(i0=>is1,i1=>a(2),t=>is2,vdd=>vdd,vss=>vss);
```

```
c8:a2_y PORT MAP(i0=>is2,i1=>na(3),t=>is3,vdd=>vdd,vss=>vss); --1er minterm
```

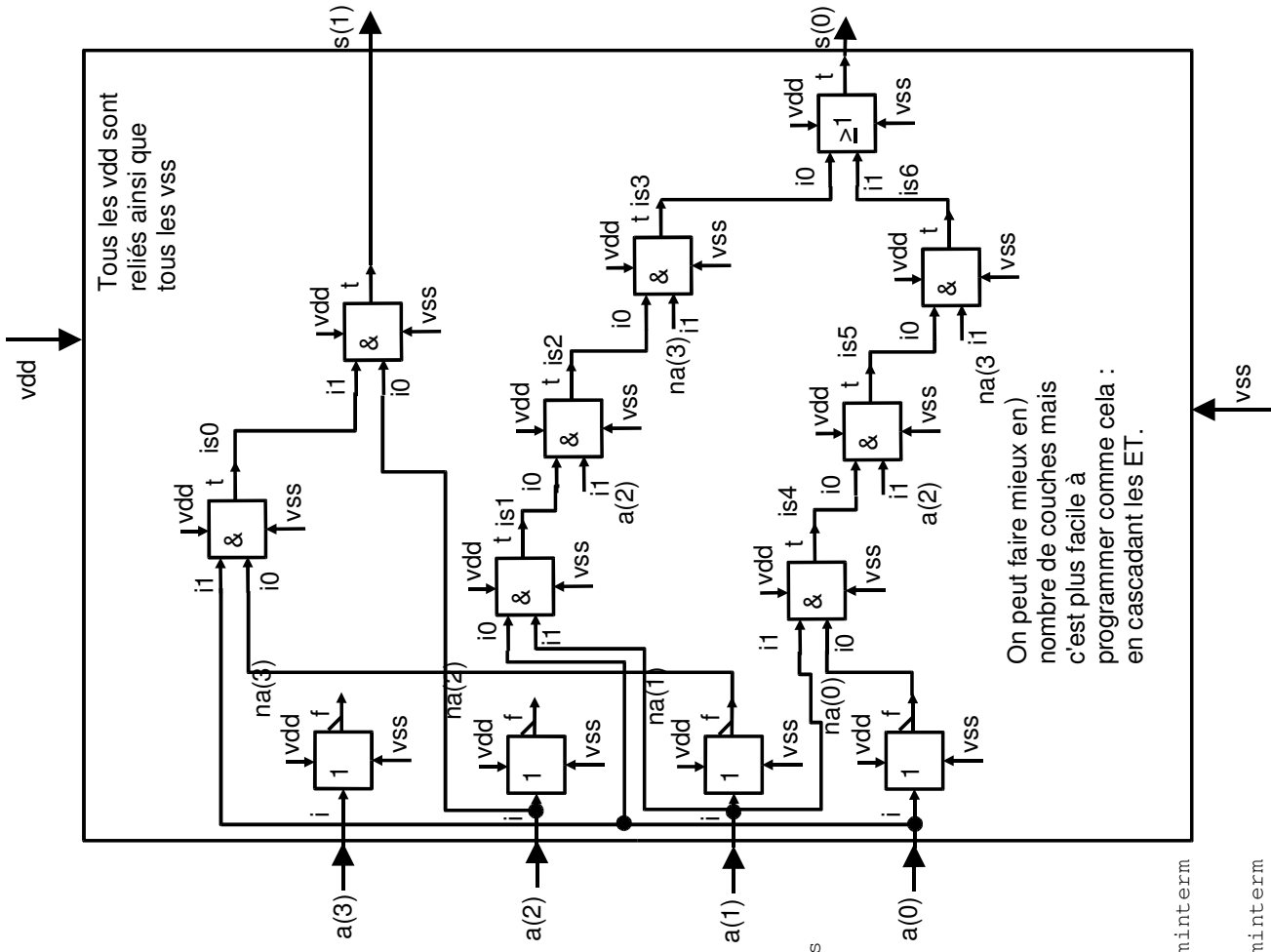
```
c9:a2_y PORT MAP(i0=>na(0),i1=>a(1),t=>is4,vdd=>vdd,vss=>vss);
```

```
c10:a2_y PORT MAP(i0=>is4,i1=>a(2),t=>is5,vdd=>vdd,vss=>vss);
```

```
c11:a2_y PORT MAP(i0=>is5,i1=>na(3),t=>is6,vdd=>vdd,vss=>vss); --2° minterm
```

```
c12:o2_y PORT MAP(i0=>is3,i1=>is6,t=>s(0),vdd=>vdd,vss=>vss); --OU des minterms
```

```
END mydemo;
```



Tous les vdd sont
reliés ainsi que
tous les vss

On peut faire mieux en
nombre de couches mais
c'est plus facile à
programmer comme cela :
en cascadeant les ET.

Travail propre au sujet 6 (VHDL NETLIST en NAND CMOS)

Votre travail va consister à réaliser un certain nombre de sous-programmes autour de ce qui a été appris en a2i11 (logique).

Une équation logique disjonctive comporte plusieurs termes séparés par des OU et ne contenant que des ET. Une telle équation sera appelée array et sera modélisée de la façon suivante :

$$y = \overline{a} \cdot \overline{b} + a \cdot \overline{b} \Leftrightarrow ON = \begin{pmatrix} 01 \\ 10 \end{pmatrix} \text{ et } y = \overline{a} \cdot \overline{b} \cdot c + a \cdot \overline{b} \Leftrightarrow ON = \begin{pmatrix} 101 \\ 11X \end{pmatrix}$$

Lorsque l'on a plusieurs sorties :

a3	a2	a1	a0	s1	s0
0	1	0	1	1	1
0	1	1	0	0	1
1	1	0	1	1	0

(ce qui n'est pas mentionné correspond à 00 en sortie)

La table de vérité ci-contre est incomplète. Elle sera modélisée par une structure C :

```
struct arraytyp {
    int nlin,ncole,ncols;
    cubetyp
    datae[MAX_LINE],datas[MAX_LINE];
}psujet={3,4,2,{5,6,13},{3,1,2}};
```

Sous Linux vous ne pouvez travailler qu'avec le compte « ge1 » avec mot de passe « stu3511 » pour avoir accès à un répertoire qui s'appelle projetut1. Ce répertoire contient un fichier source appelé logic2.c ainsi que deux répertoires et leur programmes correspondants : alliance un ensemble de logiciels français destiné à créer des ASICs et plpl destiné à programmer des PALs en partant d'un certain langage pour aboutir à un fichier JEDEC.

Tous les sujets utilisent logic2.c que l'on vous demande de lire. Il effectue un certain nombre de choses, comme la simplification d'équations [void absorbArray(struct arraytyp *p);void combineArray(struct arraytyp *p);void simplifieconsensusArray(struct arraytyp *p);] ou au contraire la mise sous forme canonique [void mintermArray(struct arraytyp *p);] Votre travail consistera à partir de ce fichier et en général à lui ajouter des sous-programmes (profitez-en pour le copier sous un autre nom : cp logic2.c monprog.c). Choisissez un nom qui vous est propre car d'autres étudiants peuvent se logger en ge1 et effacer votre travail. Il convient donc de le sauver régulièrement sur disquette : « copy monprog.c a : »

alliance est une suite logicielle française d'outils pour réaliser des ASICs (Très gros circuits programmables).

plpl est une suite logicielle permettant de programmer des PALs à partir d'un langage comme VHDLmais moins puissant).

Travail à réaliser

1°) Écrire void readarray(struct arraytyp *p); qui demande à l'utilisateur d'entrer un array sous la forme qui vous intéresse. La difficulté de ce sous programme est d'entrer les X, c'est à dire que quand l'utilisateur entre un X il faut le coder sur 2 bits non conjoints.

Remarques :

- Vous avez tous ce sous-programme à réaliser et rien ne vous empêche de vous mettre à

plusieurs pour le réaliser.

- Vous pouvez entreprendre la suite sans faire ce sous-programme puisque comme on vous le montre dans logic2.c il est possible de déclarer une variable avec initialisation.

Vous avez plusieurs exemples dans logic2.c :

```
//p1={11,4,1,{0,7,1,12,3,2,5,13,8,15,10},{1,1,1,1,1,1,1,1,1,1,1,1}};  
p1={4,4,2,{0,12,259,8},{3,257,1,0}},  
p2={4,4,2,{0,12,3,8},{3,1,1,2}},  
p3={5,4,2,{0,12,259,3,1036},{3,257,1,1,1}},  
pcomb={6,3,2,{0,1,3,5,7,6},{3,3,3,2,3,3}};
```

2°) Le but est de générer à partir d'un arraytyp deux fichiers : une description structurelle en VHDL et un fichier de pattern. Nous allons vous apprendre dans un instant à générer du VHDL quand au fichier de pattern, cela ressemble aux vecteurs tests. Le binaire asimut permet de vérifier que le fichier VHDL correspond bien aux spécifications du fichier de patterns.

Le fichier VHDL produit aura forcément l'extension *.vst.

Le prototype du sous-programme : void array2VHDLNetListEtNON(struct arraytyp p);

Le fichier VHDL "demo.vst" correspondant (tout ce qui commence par -- est un commentaire non obligatoire) se présente comme à la page suivante où l'on tente d'expliquer la relation entre VHDL et un schéma.

Il existe dans la librairie sclib ("alliance/cells/sclib") un certain nombre de fichiers intéressants comme :

na3_y.vbe (ET-NON à 3 entrées) et na4_y.vbe (ET-NON à 4 entrées) qui peuvent être utilisés. Il suffit de prendre l'interface PORT(...) et de la transformer en COMPONENT xxx_y port(...) END COMPONENT.

Remarque : l'utilisation de NANDs à 4 entrées avec comme limitation 4 entrées pour les fonctions peuvent être envisagés pour vous simplifier la vie. Pour vous faciliter le travail vous disposez dans logic2.c de deux sous-programmes appelés nand et schemaEnNands capables de générer un programme VHDL à l'écran en NAND. Modifier le sous-programme nand pour générer votre schéma en NANDs de manière un peu plus optimisée (moins de couches). Modifier ensuite le sous-programme schemaEnNands pour qu'il génère un fichier correctement, c'est à dire avec la bonne déclaration SIGNAL

```
export MBK_IN_LO=vst  
export MBK_CATA_LIB=alliance/cells/sclib  
alliance/bin/asimut -c demo
```

Les deux export permettent de positionner des variables d'environnement. A noter que ces exports ne sont à réaliser qu'une seule fois : toute utilisation ultérieure de asimut se fera directement sans redéfinir les variables d'environnement.. Le premier export demande à asimut d'utiliser des fichiers de type vst. La deuxième variable demande à asimut d'utiliser une librairie CMOS standard appelée sclib.

3°) Le fichier de patterns "demo.pat" produit aura forcément l'extension .pat (toute ligne commençant par un # est un commentaire) :

```
in vdd;  
in vss;  
in a(3 to 0);  
out s(1 to 0);  
begin  
# a=0101, we expect value 3 on s  
pat_1 : 1 0 0101 ?11 ;  
# a=0110, we expect value 1 on s
```

```
pat_2 : 1 0 0110 ?01 ;
# a=13, we expect value 2 on s
pat_3 : 1 0 1101 ?10 ;
# a=7, we still expect bin 0 on s
pat_4 : 1 0 0111 ?00 ;
end;
```

Le prototype du sous-programme : `void array2Pat(struct arraytyp p);`

4°) La vérification de l'ensemble se fait avec : (vous êtes supposés être à la racine de alliance :

```
alliance/bin/asimut demo demo specif
```

le premier demo est en fait demo.vst, le deuxième est demo.pat et specif contiendra le résultat du test (specif.pat)

En cas de bon fonctionnement vous verrez à l'écran :

```
.....
linking ...
executing ...
###----- processing pattern 0 -----###
###----- processing pattern 1 -----###
###----- processing pattern 2 -----###
###----- processing pattern 3 -----###
```

Si aucun message d'erreur n'est donné, cela signifie que les patterns attendus sont au rendez-vous donc que tout fonctionne comme prévu.

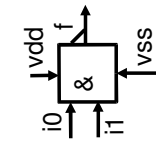
```
ENTITY demo IS
PORT(
```

```
  a : in BIT_VECTOR(3 DOWNTO 0) ; -- 4 entrees
  s : out BIT_VECTOR(1 DOWNTO 0) ; -- 2 sorties
  vdd, vss : in BIT -- les alimentations
);
```

```
END demo;
```

```
ARCHITECTURE mydemo OF demo IS
```

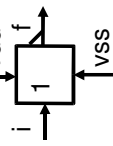
```
-- on commence par declarer les composants que l'on utilise :
COMPONENT na2_y port( -- ET-NON à deux entrées
```



```
);
```

```
END COMPONENT;
```

```
COMPONENT n1_y port( --inverseur
```



```
);
```

```
END COMPONENT;
```

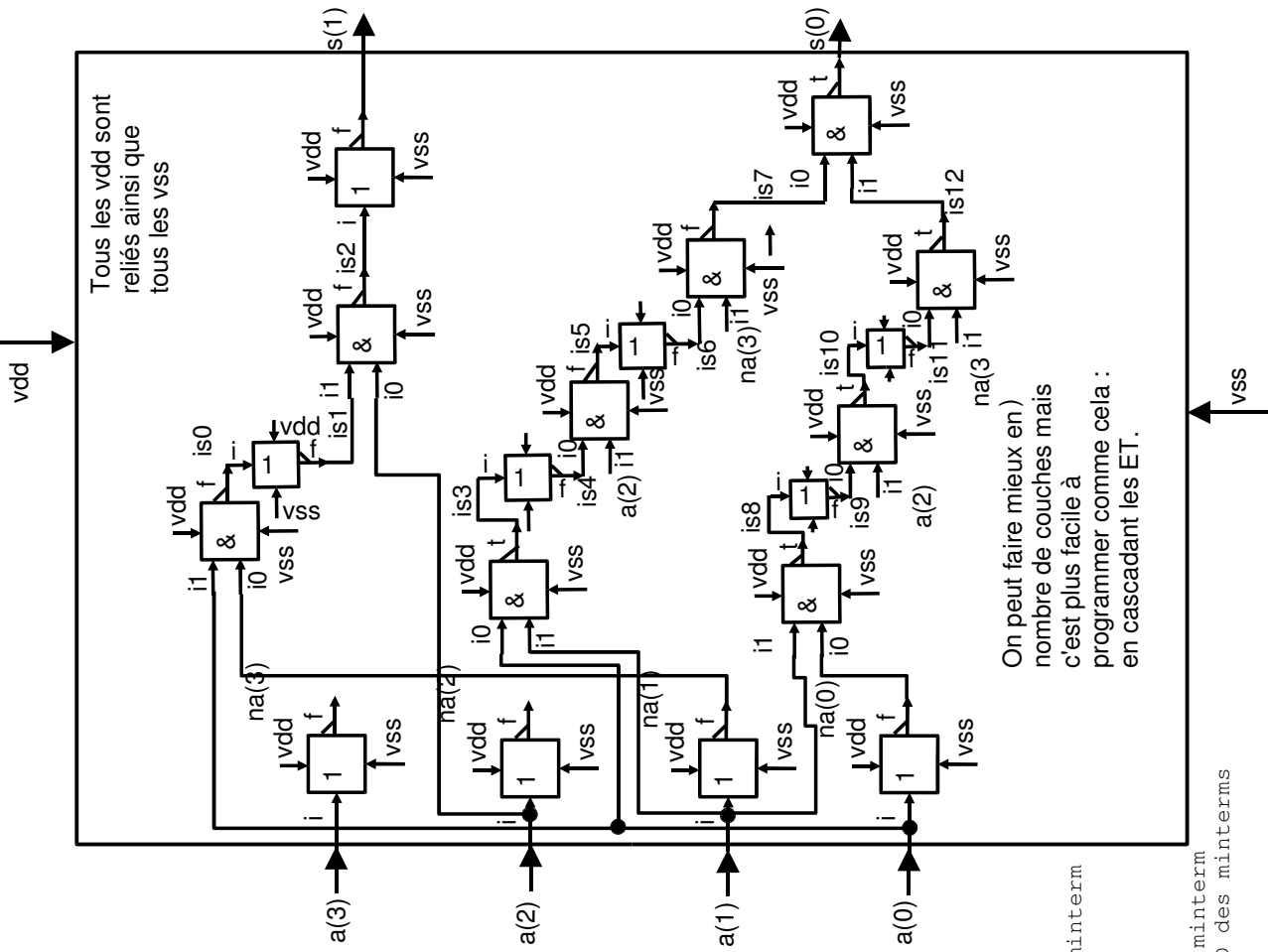
```
-- on declare ensuite les variables qui nous serviront de connexions
SIGNAL na : BIT_VECTOR(3 DOWNTO 0); -- NON des entrees
SIGNAL is0, is1, is2, is3, is4, is5, is6, is7, is8, is9, is10, is11, is12 : BIT;
```

```
-- on décrit ensuite le schéma ET-NON appelé NEILLIST
```

```
BEGIN -- on complètement toutes les entrées a(i) dans na (i)
```

```

c0:n1_y PORT MAP(i=>a(0),f=>na(0),vdd=>vdd,vss=>vss);
c1:n1_y PORT MAP(i=>a(1),f=>na(1),vdd=>vdd,vss=>vss);
c2:n1_y PORT MAP(i=>a(2),f=>na(2),vdd=>vdd,vss=>vss);
c3:n1_y PORT MAP(i=>a(3),f=>na(3),vdd=>vdd,vss=>vss); -- line 33
-- on décrit s(1) Eh OUI, il se simplifie !!!
c4:na2_y PORT MAP(i0=>na(1),i1=>a(0),f=>is0,vdd=>vdd,vss=>vss);
c5:n1_y PORT MAP(i=>is0,f=>is1,vdd=>vdd,vss=>vss);
c6:na2_y PORT MAP(i0=>a(2),i1=>is1,f=>is2,vdd=>vdd,vss=>vss);
c7:n1_y PORT MAP(i=>is2,f=>s(1),vdd=>vdd,vss=>vss);
--on décrit s(0) Lui ne se simplifie pas
c8:na2_y PORT MAP(i0=>a(0),i1=>na(1),f=>is3,vdd=>vdd,vss=>vss);
c9:n1_y PORT MAP(i=>is3,f=>is4,vdd=>vdd,vss=>vss);
c10:na2_y PORT MAP(i0=>is4,i1=>a(2),f=>is5,vdd=>vdd,vss=>vss);
c11:n1_y PORT MAP(i=>is5,f=>is6,vdd=>vdd,vss=>vss);
c12:na2_y PORT MAP(i0=>is6,i1=>na(3),f=>is7,vdd=>vdd,vss=>vss); --1er minterm
c13:na2_y PORT MAP(i0=>na(0),i1=>a(1),f=>is8,vdd=>vdd,vss=>vss);
c14:n1_y PORT MAP(i=>is8,f=>is9,vdd=>vdd,vss=>vss);
c15:na2_y PORT MAP(i0=>is9,i1=>a(2),f=>is10,vdd=>vdd,vss=>vss);
c16:n1_y PORT MAP(i=>is10,f=>is11,vdd=>vdd,vss=>vss);
c17:na2_y PORT MAP(i0=>is11,i1=>na(3),f=>is12,vdd=>vdd,vss=>vss); --2° minterm
c18:na2_y PORT MAP(i0=>is7,i1=>is12,f=>s(0),vdd=>vdd,vss=>vss); --NAND des minterms
END mydemo;
```



Travail propre au sujet 7 (DES simplifié)

DES simplifié

Pour étudier un peu sa façon de fonctionner, on vous propose un algorithme DES simplifié nommé S-DES développé par Professor Edward Schaefer of Santa Clara University. Il fonctionne sur des blocs de données 8 bits (contre 64 bits pour DES) et avec une clé de 10 bits contre 56 bits pour DES.

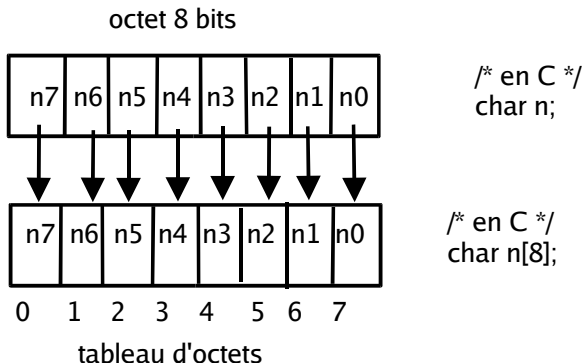
Nous allons commencer par expliquer son fonctionnement. Cet algorithme comprend deux parties :

- génération de clés
- chiffrement par rondes et autre

Commençons donc par expliquer la génération des clés.

1°) Génération de clés

Avant de commencer nous allons expliciter correctement nos conventions. Un octet peut être vu comme un ensemble de 8 bits dont on représente ci-dessous une vue avec le poids faible à droite. Si un tel octet doit être transformé en tableau de caractères avec chacun des bits de l'octet dans une case du tableau alors notre transformation se fera comme ci-dessous, c'est à dire que la case 0 contiendra le bit de poids fort ... et la case 7 le bit de poids faible. En clair dans nos raisonnements la case 0 du tableau sera toujours à gauche. Cette convention restera vraie quelle que soit la taille en bits que l'on transforme en tableau.



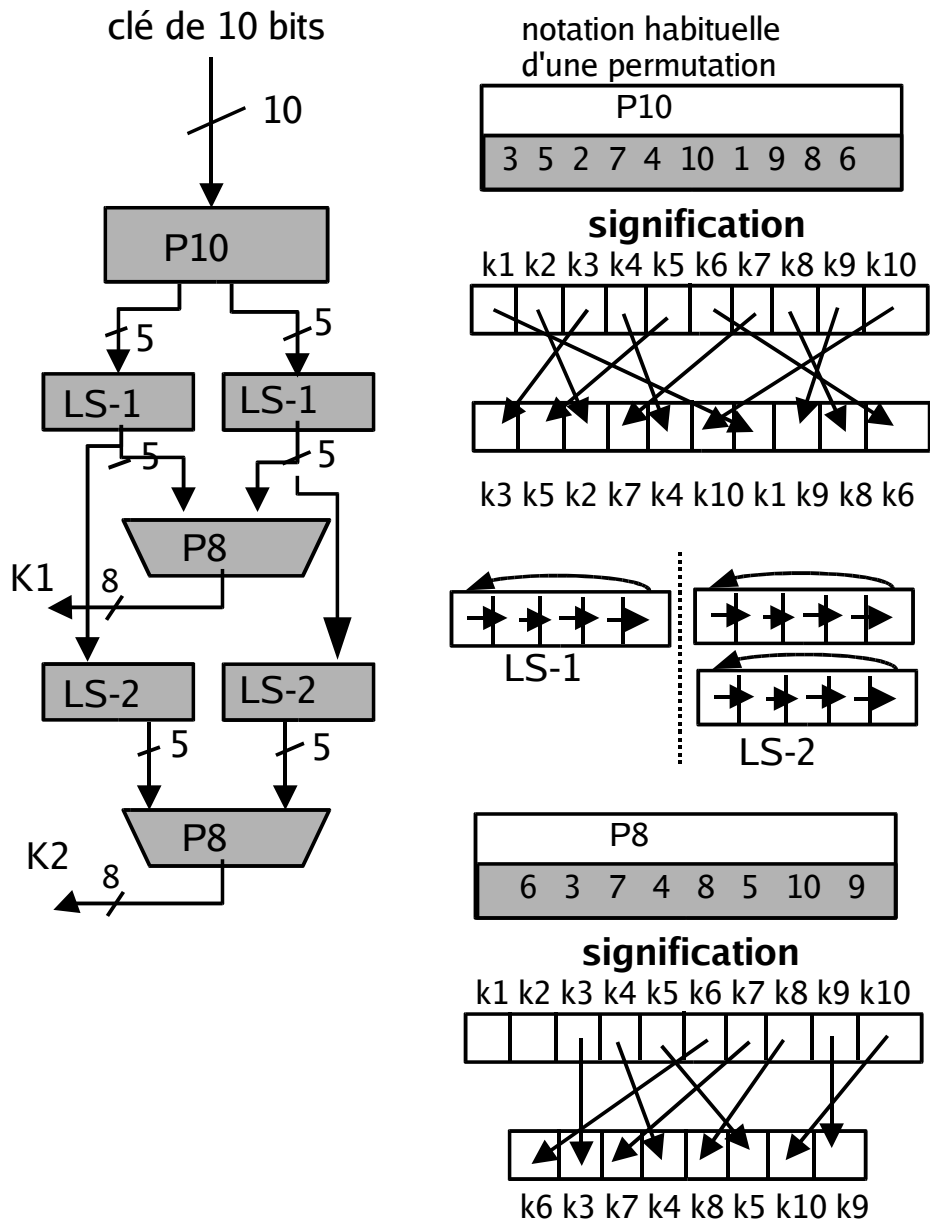
On vous donne un programme en C permettant de générer deux clés de 8 bits à partir d'une clé de 10 bits suivi des explications à l'aide d'un dessin.

```
void generecles(short cle10, char cle8[2]){
char clef10[10], i, gauche[5], droite[5], clef8[8], res10[10], gaucheLS_1[5],
  droiteLS_1[5], gaucheLS_2[5], droiteLS_2[5];
char p10[10]={2, 4, 1, 6, 3, 9, 0, 8, 7, 6};
char p8[8]={5, 2, 6, 3, 7, 4, 9, 8};
  for (i=0; i<10; i++) /* conversion 10 bits -> tableau 10 cases */
    clef10[9-i]=(cle10 & 1<<i)>>i;
  /* permutation p10 */
  for(i=0; i<10; i++)
    res10[i]=clef10[p10[i]]; // pourrait etre res10[9-i]=..... en fonct du poids
  /* separation gauche droite */
  for (i=0; i<5; i++){
    gauche[i]=res10[i];
    droite[i]=res10[i+5];
  }
  /* decalage circulaire d'un cran LS-1 */
  for (i=0; i<5; i++){
    gaucheLS_1[i]=gauche[(i+1)%5];
```

```

droiteLS_1[i]=droite[(i+1)%5];
}
/* reconstitution de la première clé */
/* d'abord sur 10 bits */
for (i=0;i<5;i++){
clef10[i]=gaucheLS_1[i];
clef10[i+5]=droiteLS_1[i];
}
/* puis sur 8 bits avec p8 */
for (i=0;i<8;i++){
clef8[i]=clef10[p8[i]];
/* conversion du tableau 8 cases en char (8 bits) */
cle8[0]=128*clef8[0]+64*clef8[1]+32*clef8[2]+16*clef8[3]+
8*clef8[4]+4*clef8[5]+2*clef8[6]+clef8[7];
/* decalage circulaire de deux crans LS-2 */
for (i=0;i<5;i++){
gauche[i]=gaucheLS_1[(i+1)%5];
droite[i]=droiteLS_1[(i+1)%5];
}
}
for (i=0;i<5;i++){
gaucheLS_2[i]=gauche[(i+1)%5];
droiteLS_2[i]=droite[(i+1)%5];
}
/****** suite après la figure *****/

```



```

/***** suite *****/
/* reconstitution de la deuxième clé */
/* d'abord sur 10 bits */
for (i=0;i<5;i++){
    clef10[i]=gaucheLS_2[i];
    clef10[i+5]=droiteLS_2[i];
}
/* puis sur 8 bits avec p8 */
for (i=0;i<8;i++)
    clef8[i]=clef10[p8[i]];
/* conversion du tableau 8 cases en char (8 bits) */
cle8[1]=128*clef8[0]+64*clef8[1]+32*clef8[2]+16*clef8[3]+
    8*clef8[4]+4*clef8[5]+2*clef8[6]+clef8[7];
}

```

3°) Algorithme de codage et décodage

Les rondes peuvent être vues comme des fonctions f_k :

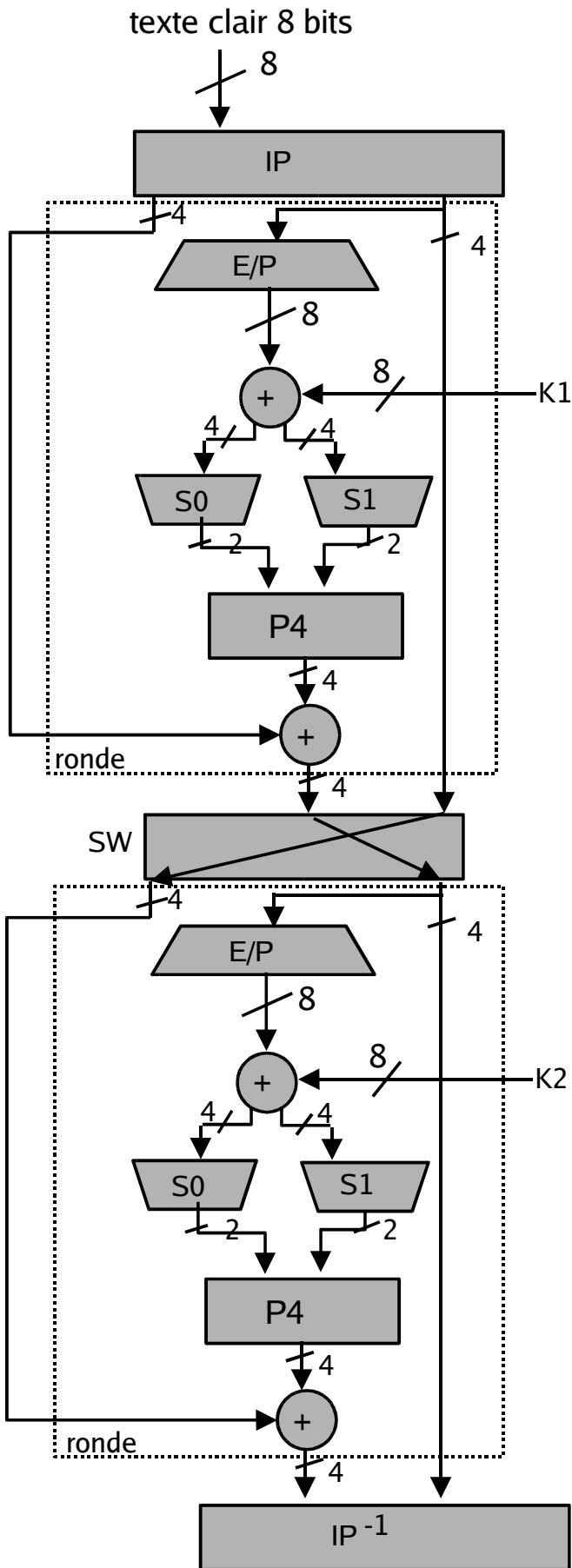
$$f_k(L,R) = (L \oplus F(R,SK), R)$$

où \oplus dénote le ou exclusif, SK une sous-clé, L 4 bits côté gauche (poids forts) et R le côté droit (poids faible). Les deux points qui méritent d'être expliqués sont le ou exclusif et la fonction F.

La présence du ou exclusif est due à une de ses célèbres priorités : si l'on a $a \oplus b = c$ alors $a = c \oplus b$: on peut retrouver a à partir de c et de b. Cela aura une conséquence importante sur l'algorithme final, il y aura peu de différence entre le codage et le décodage.

Les fonctions F feront quant à elles la force de l'algorithme. Elles introduisent des non linéarités fondamentales pour éviter une cryptanalyse facile basée sur la résolution de système d'équations.

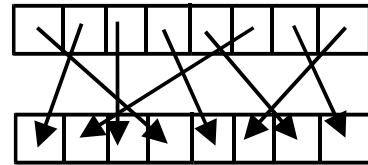
L'explication tient dans le dessin suivant :



IP
2 6 3 1 4 8 5 7

signification

k1 k2 k3 k4 k5 k6 k7 k8

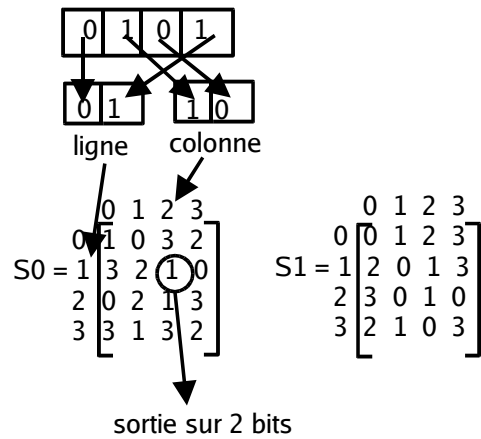


k2 k6 k3 k1 k4 k8 k5 k7

IP^{-1}
4 1 3 5 7 2 8 6

E/P
4 1 2 3 2 3 4 1

P4
2 4 3 1



$$S_0 = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 3 & 2 & 1 & 0 \\ 2 & 0 & 2 & 1 & 3 \\ 3 & 3 & 1 & 3 & 2 \end{bmatrix}$$

$$S_1 = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 2 & 0 & 1 & 3 \\ 3 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 & 3 \end{bmatrix}$$

Travail à réaliser

Écriture d'un algorithme de chiffrement et déchiffrement simplifié : S-DES en reprenant les morceaux de programme donnés dans le texte.

Les autres sujets sont plutôt des suggestions et donc pas aussi détaillés que les sept précédents. Ils nécessitent donc de chercher de la documentation sur Internet.

Travail propre au sujet 8 (Vigenere)

Introduction

On peut trouver sur Internet de la documentation sur le cryptographie Vigenere (et même probablement en français). Vous trouverez aussi sur mon site perso un cours de cryptographie classique qui l'explique un peu aussi.

Travail à réaliser

Écrire un programme qui permet de réaliser un chiffrement et un déchiffrement Vigenere

Sujet 9 (probablement difficile)

Introduction

DES (Data Encryption Standard) est le standard de chiffrement américain depuis 1977. Beaucoup critiqué au départ à cause de la participation de la NSA à sa réalisation, il doit être remplacé par AES qui à été choisi en 2001. Il fonctionne par chiffrement de blocs de 64 bits à l'aide d'une clé de 56 bits.

Travail à réaliser

On vous demande de télécharger sur Internet le code de DES, par exemple à l'adresse

<http://www.funet.fi/pub/crypt/cryptography/symmetric/des/>

On y trouve un fichier [des-linux-1.0.tar.gz](#)

Vous allez utiliser ce code pour réaliser un programme qui propose à l'utilisateur les différents modes de fonctionnement d'un tel algorithme :

- ECB : Electronic Codebook Mode
- CBC : Cipher Block Chain Mode
- CFB : Cipher Feedback Mode
- OFB : Output Feedback Mode

Sujet 10 (probablement difficile)

Introduction

IDEA (International Data Encryption Algorithm). Il fonctionne par chiffrement de blocs de 64 bits à l'aide d'une clé de 128 bits.

Travail à réaliser

On vous demande de télécharger sur Internet le code de IDEA, par exemple à l'adresse

`http://www.funet.fi/pub/crypt/cryptography/symmetric/idea/`

Vous allez utiliser ce code pour réaliser un programme qui propose à l'utilisateur les différents modes de fonctionnement d'un tel algorithme :

- ECB : Electronic Codebook Mode
- CBC : Cipher Block Chain Mode
- CFB : Cipher Feedback Mode
- OFB : Output Feedback Mode

D'autres sujets peuvent être suggérés par les étudiants.
--