

TP1 : Synthèse logique

1. Avant-propos
2. 1 Introduction
 1. 1.1 Synthèse d'automates d'états finis
 1. 1.1.1 SYF et VHDL
 2. 1.1.2 Exemple
 2. 1.2 Synthèse logique et optimisation structurelle
 1. 1.2.1 Synthèse logique
 2. 1.2.2 Résolution des problèmes de fanout (sortance)
 3. 1.2.3 Visualisation de la chaîne longue
 4. 1.2.4 Vérification de la netlist
3. 2. Travail à effectuer
 1. 2.1 Réalisation d'un compteur
 2. 2.2 Réalisation d'un automate pour digicode
 3. 2.3 Optimisation du réseau booléen
 4. 2.4 Mapping sur cellules précaractérisées
 5. 2.5 Visualisation de la netlist
 6. 2.6 Optimisation de la netlist
 7. 2.7 Vérification de la netlist
4. 3. Compte rendu

Avant-propos

Le but de ce TP est de présenter quelques outils de la chaîne ALLIANCE dont :

- Les outils de synthèse logique **SYF**, **BOOM**, **BOOG**, **LOON** ;
- L'éditeur graphique de netlist **XSCH** ;
- Le simulateur **ASIMUT** ;

Chaque outil possède ses propres options donnant des résultats plus ou moins adaptés suivant l'utilisation que l'on veut faire du circuit.

Ce TP portera donc sur les méthodes de génération et de validation d'une netlist de cellules précaractérisées. En effet, même s'il est acquis que les outils de génération d'ALLIANCE fonctionnent correctement, la validation de chaque vue générée est indispensable. Elle permet de limiter le coût et le temps de la conception.



Les dépendances de données dans le ?ux sont matérialisées dans la réalité par une dépendance de ?chier. Le ?chier **Make?le** exécuté à l'aide de la commande **make** permet de gérer ces dépendances. Ce TP n'étant pas un cours sur le Make?le, nous nous limiterons à expliquer l'usage qui en est fait dans les exemples fournis. (cf. Annexe)

L'usage de Make?le est obligatoire pour ce TP !!!

1 Introduction

1.1 Synthèse d'automates d'états finis

Un circuit combinatoire pur ne dispose pas de registres internes. De ce fait, ses sorties ne dépendent que de ses entrées primaires. A l'inverse, un circuit séquentiel synchrone disposant de registres internes voit ses sorties

changer en fonction de ses entrées mais aussi des valeurs mémorisées dans ses registres. En conséquence, l'état du circuit à l'instant $t+1$ dépend aussi de son état à l'instant t . Ce type de circuit peut être modélisé par un **automate d'états finis**.



L'automate de MOORE voit l'état de ses sorties changer uniquement sur front d'horloge. Les entrées peuvent donc bouger entre deux fronts sans modifier les sorties. Par contre dans le cas d'un automate de MEALY, la variation des entrées peut modifier à tout moment la valeur des sorties. Dans notre **fsm** (*finite-state machine*), on s'imposera de séparer la fonction de génération de la fonction de transition (automate de Moore). Pour cela, deux process distincts matérialiseront le calcul du prochain état et sa mise à jour.



1.1.1 SYF et VHDL

- Afin de décrire de tels automates, on utilise un style particulier de description VHDL qui définit l'architecture fsm. Le fichier correspondant possède également l'extension **.fsm**.
- A partir de ce fichier, l'outil **SYF** effectue la synthèse d'automate et transforme cet automate abstrait en un réseau booléen. **SYF** génère donc un fichier VHDL au format vbe. Comme la plupart des outils utilisés au laboratoire, il faut positionner certaines variables d'environnement avant d'utiliser **SYF**. Pour les connaître, reportez-vous au man de **SYF**.

1.1.2 Exemple

Afin de se familiariser avec la syntaxe de description d'un fichier .fsm, un exemple de compteur de trois "1" successifs est présenté. Sa vocation est de détecter par exemple sur une liaison série une séquence de trois "1" successifs. Le graphe d'états que l'on cherche à décrire est représenté sur la figure . Le format fsm est également décrit dans une page man. Pensez à la consulter.



```
entity circuit is
port (
    ck, i, reset, vdd, vss : in bit ;
    o : out bit
) ;
end circuit ;
architecture MOORE of circuit is

    type ETAT_TYPE is (E0, E1, E2, E3) ;
    signal EF, EP : ETAT_TYPE;
    -- pragma CURRENT_STATE EP
    -- pragma NEXT_STATE EF
    -- pragma CLOCK CK

begin
    process (EP, i, reset)
    begin

        if (reset=?1?) then
            EF<=E0;
        else
            case EP is
                when E0 =>
                    if (i=?1?) then
                        EF <= E1 ;
                    end if;
                -- when E1 =>
                -- when E2 =>
                -- when E3 =>
            end case;
        end if;
    end process;
end architecture;
```

```

        else
            EF <= E0 ;
        end if ;
    when E1 =>
        if (i=?1?) then
            EF <= E2 ;
        else
            EF <= E0 ;
        end if ;
    when E2 =>
        if (i=?1?) then
            EF <= E3 ;
        else
            EF <= E0 ;
        end if ;
    when E3 =>
        if (i=?1?) then
            EF <= E3 ;
        else
            EF <= E0 ;
        end if ;
    when others => assert (?1?)
        report "etat illegal";
    end case ;
end if ;
case EP is
    when E0 =>
        o <= ?0? ;
    when E1 =>
        o <= ?0? ;
    when E2 =>
        o <= ?0? ;
    when E3 =>
        o <= ?1? ;
    when others => assert (?1?)
        report "etat illegal" ;
    end case ;
end process ;

process(ck)
begin
    if (ck=?1? and not ck?stable) then
        EP <= EF ;
    end if ;
end process ;
end MOORE ;

```

1.2 Synthèse logique et optimisation structurelle

1.2.1 Synthèse logique

La synthèse logique permet d'obtenir une netlist de portes à partir d'un réseau booléen (format .vbe). Plusieurs outils sont disponibles :

- L'outil **BOOM** permet l'optimisation de réseau booléen avant synthèse.
- L'outil **BOOG** offre la possibilité de synthétiser une netlist en utilisant une bibliothèque de cellules précaractérisées telle que **SXLIB**. La netlist peut être soit au format **.vst** soit au format **.al**. Vérifier la variable d'environnement **MBK_OUT_LO=vst**.

Pour plus de renseignements sur ces outils, reportez vous au man.

1.2.2 Résolution des problèmes de fanout (sortance)

Les netlists générées contiennent parfois des signaux internes attaquant un nombre important de portes (grand fanout). Ceci se traduit par une détérioration des fronts (rise time et fall time). Il y a alors une perte en performance temporelle. Afin de résoudre ces problèmes, l'outil **LOON** remplace les cellules ayant un fanout (i.e sortance) trop grand par des cellules plus puissantes ou bien insère des buffers.

1.2.3 Visualisation de la chaîne longue

A tout moment, les netlists peuvent être éditées graphiquement. L'outil **XSCH** permet de visualiser le chemin le plus long grâce aux fichiers **.xsc** et **.vst** générés à la fois par **BOOG** et par **LOON**.



La résistance équivalente R de la figure est calculée sur la totalité des transistors du AND appartenant au chemin actif. De même, la capacité C est calculée sur les transistors passants du NOR correspondant au chemin entre i0 et la sortie de la cellule.

1.2.4 Vérification de la netlist

La netlist doit être certifiée. Pour cela, on dispose du simulateur **ASIMUT**.

2. Travail à effectuer

2.1 Réalisation d'un compteur

- En s'inspirant du compteur de trois "un" présenté, écrire la description d'un compteur de cinq "un" successifs sous la forme d'un automate de Moore.
- Lancer **SYF** avec les options de codage **-a**, **-j**, **-m**, **-o**, **-r** et en utilisant les options **-CEV**. Penser à bien positionner les variables d'environnement.

```
>syf -CEV -a <fsm_source> -
```

- Visualiser les fichiers **.enc**.
- Ecrire un fichier de vecteurs de test et simuler sous **ASIMUT**.
- Ecrire un fichier **Makefile** pour automatiser les différents appel d'outil avec les différentes options.

Que se passe-t-il si le reset n'est pas positionné en début de pattern ? Pourquoi ?

2.2 Réalisation d'un automate pour digicode

On veut réaliser une puce pour digicode. Les spécifications sont les suivantes :

```
Les chiffres de 0 à 9 sont codés en binaire naturel sur 4 bits. A et B sont codés comme suit : A
```

```
Le digicode fonctionne en deux modes :
```

- * Mode Jour : La porte s'ouvre en appuyant sur "0"
- * Mode Nuit : La porte ne s'ouvre que si le code est correct.

```
Pour distinguer les deux cas un "timer" externe calcule entre 8h00 et 20h00 et '0' sinon.
```

- * Le digicode commande une alarme dès qu'un des chiffres entrés n'est pas le bon

- * L'automate revient dans son état d'attente si rien n'est entré au clavier au bout de 5 secondes pour cela il reçoit un signal reset du timer externe.
- * La puce fonctionne à une fréquence de 10MHz.
- * Toute pression d'une touche du clavier est accompagnée du signal `press_kbd`. Celui-ci signale à la puce que les données en sortie signal est à 1 durant un cycle d'horloge.

Le code est 53A17.

L'interface de l'automate est le suivant :

- * in ck
- * in reset
- * in jour
- * in i[3 :0]
- * in 0
- * in press_kbd
- * out porte
- * out alarm

- Dessiner le graphe d'états de l'automate. (Les corrections seront distribuées)
- Le décrire au format **.fsm**.
- Le synthétiser avec **SYF** en utilisant les options de codage **-a, -j, -m, -o, -r** et en utilisant les options **-CEV**.

```
>syf -CEV -a <fsm_source> -
```

- Ecrire le fichier `.pat` de vecteurs de test.
- Simuler avec **ASIMUT** toutes les vues comportementales obtenues.
- Adapter le **Makefile** pour qu'il couvre tous les encodages possibles.

Quelles sont vos remarques concernant la complexité des expressions (i.e temps) et le nombre de registres (i.e surface) des descriptions comportementales suivant les encodages ? En déduire les deux groupes d'encodage. Comparez aussi leurs nombres de littéraux.

2.3 Optimisation du réseau booléen

- Lancer l'optimisation booléenne avec l'outil **BOOM** en demandant une optimisation en surface puis en délai.

```
>boom -V <vbe_source> <vbe_destination> -
```

- Essayer **BOOM** avec les différents algorithmes **-s, -j, -b, -g, -p...** Le mode et le niveau d'optimisation sont aussi à changer.
- Comparer le nombre de littéraux après factorisation.

2.4 Mapping sur cellules précaractérisées

Pour chacun des réseaux booléens obtenus précédemment :

- Positionner les variables d'environnement ;
- Synthétiser la vue structurelle :

```
>boog <vbe_source> -
```

- Lancer **BOOG** sur les différentes netlists pour observer l'influence des options de **SYF** et de **BOOM** ;
- Valider le travail de **BOOG** en resimulant avec **ASIMUT** les netlists obtenues avec les vecteurs de test qui ont servi à valider le réseau booléen initial.

2.5 Visualisation de la netlist

- Utiliser **XSCH** pour colorer visualiser le chemin critique.

Pour lancer l'éditeur graphique :

```
>xsch -I vst -l <vst_source> -
```

La couleur rouge désigne le chemin critique. Si vous utilisez l'option '-slide' qui permet d'afficher un ensemble de netlists, n'oubliez pas d'appuyer sur les touches '+' ou '-' pour éditer vos fichiers !

2.6 Optimisation de la netlist

Pour toutes les vues structurelles obtenues précédemment :

- Lancer **LOON**.

```
>loon <vst_source> <vst_destination> <lax_param> -
```

- Effectuer une optimisation de fanout en modifiant le facteur de fanout dans le fichier d'option **.lax**. Imposer des valeurs de capacités sur les sorties.

2.7 Vérification de la netlist

Quelle est, selon vous, la meilleure des netlists ? Pourquoi ?

À effectuer sur cette netlist :

- Valider le travail de **LOON** en resimulant sous **ASIMUT** les netlists obtenues avec les vecteurs de test qui ont servi à valider la vue comportementale initiale.
- Mettre les différents résultats (surface/temps/optimisation) dans votre rapport.

3. Compte rendu

Vous rédigerez un compte-rendu d'une page maximum pour ce TP, et vous joindrez les fichiers écrits (et uniquement les fichiers écrits, pas ceux générés par les différents outils). Vous ferez attention à bien répondre aux questions posées ici dans le compte rendu. Vous ferez également attention à joindre votre Makefile de façon à ce que la commande **make** effectue les différentes étapes de ce TP de A à Z.