Synthèse Logique

Contents

1. Introduction . . . 1
1.1 Synthèse d'Automates d'États Finis . . . 2
1.1.1 Automates de MOORE et de MEALY . . . 2
1.1.2 VHDL et **syf** . . . 3
1.1.3 Exemple . . . 3
1.2 Synthèse Logique et Optimisation Structurelle . . . 5
1.2.1 Synthèse Logique . . . 5
1.2.2 Résolution des Problèmes de *Fanout* (sortance) . . . 5
1.2.3 Visualisation de la Chaîne Longue . . . 5
2. Travail à Effectuer . . . 6
2.1 Réalisation d'un Compteur . . . 6
2.2 Réalisation d'un Digicode . . . 6
3. Compte Rendu . . . 8
Makefile . . . 8

1. Introduction

Le but de ce TME 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**
- Les outils pour la preuve formelle **flatbeh** , **proof**
- 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.

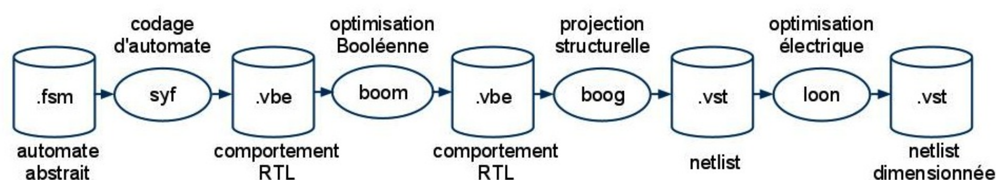

Figure 1: Figure 1 -- Flot de la synthèse logique

Ce TME 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 flux sont matérialisées dans la réalité par une dépendance de fichier. Le fichier `Makefile` exécuté à l'aide de la commande `make` permet de gérer ces dépendances.

L'usage de Makefile est obligatoire.

Pour exécuter les outils d'ALLIANCE vous devez ajouter dans le fichier **~/.bashrc**

```
source /soc/alliance/etc/profile.d/alc_env.sh
```

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.

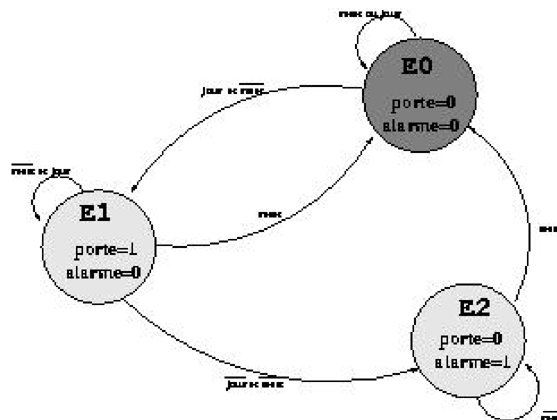

Figure 2: Figure 2 -- Automate d'état

1.1.1 Automates de MOORE et de MEALY

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.

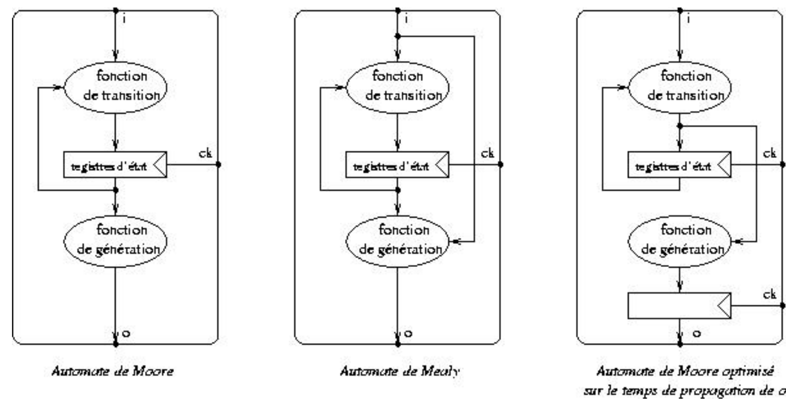

Figure 3: Figure 3 -- Automates de Moore et Mealy

1.1.2 VHDL et **syf**

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** .

1.1.3 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.

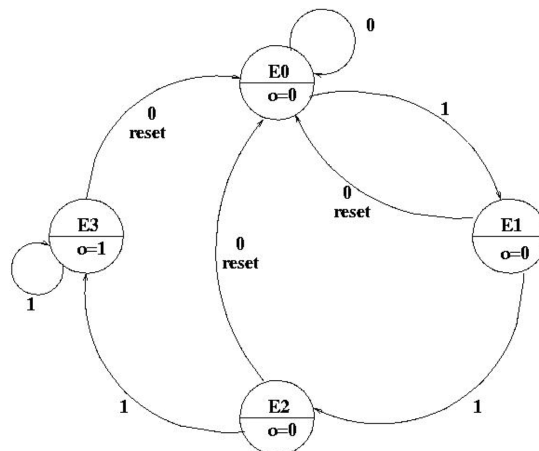

Figure 4: Figure 4 -- Exemple de compteur de "1"

Fichier **fsm** représentant le graphe précédent :

```
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;
          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** réalise la projection structurelle du comportement sur la bibliothèque de cellules précaractérisées **sxlib** afin d'obtenir la netlist. La netlist pouvant être soit au format **vst** soit au format **al** , pensez à vérifier la variable d'environnement **MBK_OUT_LO** .

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 visualisées 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** .

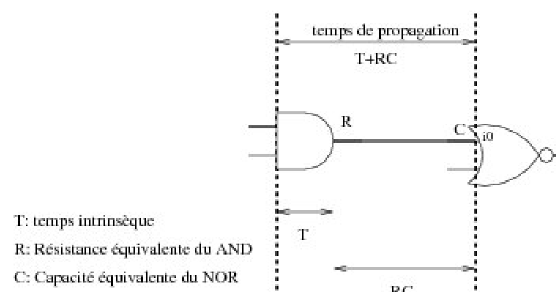

Figure 5: Figure 5 -- Calcul du temps de propagation

La résistance équivalente **R** 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.

2. Travail à Effectuer

Les différentes parties seront automatisées à l'aide d'un fichier `Makefile`.

2.1 Réalisation d'un Compteur

- En s'inspirant du compteur de trois **1** présenté, écrire au format **fsm** la description d'un compteur de cinq **1** successifs sous la forme d'un automate de Moore.
- Synthétiser l'automate avec **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.

```
etudiant@pc:TME1> syf -CEV -a <fsm_source>
```

- Visualiser les fichiers **enc** .
- Ecrire le fichier **pat** de vecteurs de test.
- Simuler avec **asimut** toutes les vues comportementales obtenues.

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

2.2 Réalisation d'un Digicode

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

1. Les chiffres de 0 à 9 sont codés en binaire naturel sur 4 bits.
2. A et B sont codés comme suit : `A = 1010`, `B = 1011`.
3. Le digicode fonctionne en deux modes :
   - Mode Jour : La porte s'ouvre en appuyant sur "O"
   - Mode Nuit : La porte ne s'ouvre que si le code est correct
4. Pour distinguer les deux modes un **timer externe** calcule le signal jour. Ce signal vaut '1' entre 8h00 et 20h00 et '0' sinon.
5. Le digicode commande une alarme dès qu'un des chiffres entrés n'est pas le bon.
6. L'automate revient dans son état d'attente automatiquement dans deux cas, grâce à un **timer externe** active le signal reset :
   - Si rien n'est entré au clavier au bout de 5 secondes,
   - Si l'alarme a sonné pendant 2mn.
7. La puce fonctionne à une fréquence de 10MHz
8. Toute pression d'une touche du clavier est accompagnée du signal **press_kbd** . Celui-ci signale à la puce que les données en sortie du clavier sont valides. Ce signal est à 1 durant un cycle d'horloge.

Le code est `53A17` .

Note

On ne vous demande pas de programmer les timers, mais vous pouvez dans votre compte rendu faire un schéma représentant le clavier, les timers et l'automate.
Le code de l'automate est câblé, toutefois si vous vous le souhaitez vous pouvez rendre votre digicode programmable. Dans ce cas il vous appartient de proposer une spécification, puis de l'implémenter.

L'interface de l'automate est le suivant :

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

Vous devez :

1. Dessiner le graphe d'états de l'automate.
2. Écrire au format **fsm** l'automate.
3. Synthétiser l'automate avec **syf** en utilisant les options de codage `-a`, `-j`, `-m`, `-o`, `-r` et en utilisant les options `-CEV`.

```
etudiant@pc:TME1> syf -CEV -a <fsm_source>
```

4. Ecrire le fichier .pat de vecteurs de test.
5. Simuler avec **asimut** toutes les vues comportementales obtenues.

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 ? Comparez aussi leurs nombres de littéraux.

5. Lancer l'optimisation du réseau booléen avec l'outil **boom** en demandant une optimisation en *surface* puis en *délai*.

```
etudiant@pc:TME1> boom -V <vbe_source> <vbe_destination>
```

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

Pour chacun des réseaux booléens obtenus précédemment, effectuer le mapping sur cellules précaractérisées :

8. Synthétiser la vue structurelle (en faisant attention à bien positionner les variables d'environnement) en lançant l'outil **boog** .

```
etudiant@pc:TME1> boog <vbe_source> <vst_destination>
```

9. Observer l'influence des options de **syf** et de **boom** avec les différences netlists obtenues.
10. 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.
11. Utiliser **xsch** pour visualiser la netlist.

```
etudiant@pc:TME1> xsch -I vst -l <vst_source>
```

Cet outil vous permet de visualiser le chemin critique, représenté en rouge.

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 !

12. Pour toutes les vues structurelles obtenues précédemment :
    - Optimiser la netlist en lancant l'outil **loon** .

    ```
    etudiant@pc:TME1> 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.
    - **Quelle est, selon vous, la meilleure des netlists ? Pourquoi ?**
13. 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.

    Deux précautions valent mieux qu'une ! Faire une vérification formelle de la netlist en la comparant au fichier comportemental d'origine issu de **syf** .

    ```
    etudiant@pc:TME1> flatbeh <vst_source> <vbe_dest>
    etudiant@pc:TME1> proof -d <vbe_origine> <vbe_dest>
    ```

    Comparer si les deux fichiers sont bien identiques.

3. Compte Rendu

Vous rédigerez un compte-rendu d'une page maximum pour ce TME dans lequel vous ferez attention à bien répondre aux questions posées ici (en gras). Vous inclurez les différents résultats obtenus surface/temps/optimisation.

Vous enverrez le compte rendu par mail (franck.wajsburt@...) avant le début du prochain TME (le sujet du mail doit être **TOOLS TP1 2017**). Vous devez indiquez vos noms (binôme) dans le compte-rendu. Vous joindrez les fichiers écrits : soit une archive contenant tous les fichiers dans le mail, soit le chemin d'accès aux fichiers, en faisant attention dans ce cas à laisser les droits. Le dessin du graphe du digicode doit être fait avec les outils de graphviz.

Vous ferez attention à joindre les différents `Makefile` créés de façon à ce que la commande `make` effectue les différentes étapes de ce [TME|. Ces fichiers doivent également fournir une règle **clean** qui permet d'effacer tous les fichiers générés.

Ces règles seront à suivre durant tous les TME de Tools.

Makefile

Le `Makefile` permet d'exprimer le processus de construction d'un circuit.

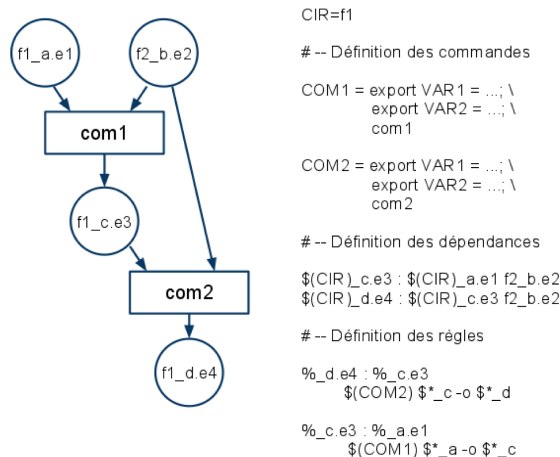

Figure 6: Figure 6 -- Exemple de Makefile