TP4 : Le chemin de données de l'AMD2901

Le chemin de données est formé de la logique régulière du circuit.

Afin de profiter de cette régularité, on génère la liste de signaux sous forme d'opérateurs vectoriels (ou colonnes) via les macro-fonctions de l'outil Stratus.

Cela permet d'économiser de la place en utilisant plusieurs fois le même matériel. Par exemple, le NOT d'un mux de n bits est instancié une seule fois pour ces n bits...

1 Exemple de description avec Stratus

Considérons le circuit suivant :

Voici la structure du chemin de données correspondante : Image(datap.jpg°,nolink?

Chacune des portes occupe une colonne, une colonne permettant de traiter un ensemble de bits pour un même opérateur. La première ligne représente le bit 3, la dernière le bit 0.

Le fichier Stratus correspondant est le suivant :

```
#!/usr/bin/env python
from stratus import *
# definition de la cellule
class circuit ( Model ):
  # declaration des connecteurs
  def Interface ( self ):
    self.a = SignalIn ( "a" , 4 )
    self.b = SignalIn ( "b" , 4 )
    self.c = SignalIn ( "c" , 4 )
    self.v = SignalIn ( "v" , 1 )
    self.cout = SignalOut ( "cout", 1 )
    self.s = SignalOut ( "s" , 4 )
    self.cmd = SignalIn ( "cmd" , 1 )
    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )
  # instanciation des operateurs
  def Netlist ( self ):
    # declaration des signaux internes
    d_aux = Signal ( "d_aux", 4 )
    e_aux = Signal ( "e_aux", 4 )
    ovr = Signal ( "ovr" , 1 )
    # generation
    Generate ( "DpgenNand2", "instance_nand2_4bits"
              , param = { ?nbit? : 4 }
              )
```

```
    # instanciation
    self.instance_nand2_4bits = Inst ( "instance_nand2_4bits"
                                     , map = { ?i0? : Cat ( self.v
                                                          , self.v
                                                          , self.v
                                                          , self.v )
                                             , ?i1? : self.a
                                             , ?nq? : d_aux
                                             , ?vdd? : self.vdd
                                             , ?vss? : self.vss
                                             }
                                      )
    Generate ( "DpgenOr2", "instance_or2_4bits"
             , param = { ?nbit? : 4 }
              )
    self.instance_or2_4bits = Inst ( "instance_or2_4bits"
                                   , map = { ?i0? : d_aux
                                           , ?i1? : self.b
                                           , ?q? : e_aux
                                           , ?vdd? : self.vdd
                                           , ?vss? : self.vss
                                           }
                                    )
    Generate ( "DpgenAdsb2f", "instance_add2_4bits"
             , param = { ?nbit? : 4 }
             )
    self.instance_add2_4bits = Inst ( "instance_add2_4bits"
                                    , map = { ?i0? : e_aux
                                            , ?i1? : self.c
                                            , ?q? : self.s
                                            , ?add_sub? : self.cmd
                                            , ?c31? : self.cout
                                            , ?c30? : ovr
                                            , ?vdd? : self.vdd
                                            , ?vss? : self.vss
                                            }
                                     )
```

Ce premier fichier définit votre circuit, enregistrez-le sous le nom "circuit.py". Il faut maintenant créer un autre fichier pour instancier votre circuit :

```
#!/usr/bin/env python
from stratus import *
from circuit import circuit
# creation du circuit
mon_circuit = circuit ( "mon_circuit" )
# creation de l?interface
mon_circuit.Interface ()
# creation de la netlist
mon_circuit.Netlist ()
# sauver les fichiers mon_circuit.vst
mon_circuit.Save ()
```

Enregistrez-le sous le nom "test.py". Changez les droits du fichier afin de le rerendre executable :

```
> chmod +x test.py
```

Puis executez le fichier :

```
>./test.py
```

Si tout se passe bien, vous obtenez le fichier "mon_circuit.vst", dans le cas contraire, et mises à part des erreurs de syntaxe, il se peut que votre environnement soit mal configuré pour Stratus.

Consultez la doc au format html

"file *:/asim/coriolis/share/doc/en/html/stratus/index.html" afin de vous renseigner sur les variables d'environnement à positionner.*

Lorsque vous avez obtenu le fichier, passez à la section "Description de la partie chemin de données".

Note : Stratus étant issu du langage Python, il faut apporter une grande importance à l'indentation.

Un bon conseil, n'utilisez pas de tabulations (ou alors configurez vos éditeurs pour qu'ils transforment automatiquement les tabulations en espaces).

2 Description du chemin de données

Compléter le fichier "amd2901_dpt.py"

puis créer le fichier "test_amd2901_dpt.py"

correspondant, pour l'exécuter en utilisant le modus operandi ci-dessous.

NOTE : la ram est déjà construite.

Générer la liste de signaux .vst à partir du fichier .py en lancant le fichier :

```
> ./test_amd2901_dpt.py
```

Valider la liste de signaux de la même manière que pour la partie contrôle.

Supprimer le fichier CATAL et simuler le circuit avec asimut.

```
> asimut -zerodelay amd2901_chip pattern resultat
```

3 Rapport

Il s'agit simplement de décrire votre travail fait en TP.

Quelles sont les deux manières de concevoir une netlist ? Quels avantages y a-t-il à

faire des colonnes d'opérateurs pour le data-path ?...

Inutile de faire un roman. Soyez clairs et concis ! Les répertoires, fichiers et logins devront être mentionnés dans le rapport ainsi que vos

noms de binômes. N'oubliez pas de mettre les droits en lecture !

4 Annexe Le Makefile

Comment gérer les dépendances de tâches

La synthèse sous Alliance se décompose en plusieurs outils s'exécutant chronologiquement

sur un flux de données. Chaque outil possède ses propres options donnant des résultats plus ou moins adaptés suivant l'utilisation que l'on veut faire du circuit. Les dépendances de données dans le flux sont matérialisées dans la réalité par une pendance de fichier. Le fichier Makefile exécuté à l'aide de la commande make permet gérer ces dépendances. Différents exemples de fichiers seront fournis durant le TP. TP n'étant pas un cours sur le Makefile, nous nous limiterons à expliquer l'usage qui est fait dans les exemples fournis.

4.1 Principe de base : Les Règles

Un Makefile est un fichier contenant une ou plusieurs règles traduisant les dépendances entre les actions et les fichiers. Voici une règle type Makefile : cible1 : dépendance1 dépendance2 .... #Rq: chaque commande doit être précédée d?une tabulation commande_X commande_Y . . . Les dépendances et cibles représentent, en général, des fichiers. Seule la première régle (hormis les modèles cf. 9.0.2) du Makefile est examinée. Les règles suivantes sont ignorées si elles ne sont pas impliquées par la première. Si certaines dépendances d'une règle X sont elles-mêmes des règles dans le Makefile alors ces dernières seront examinées avant la règle X appelante. Pour chaque règle X examinée, si au moins une de ses dépendances est plus récente que sa cible alors les commandes de la règle X seront exécutées. Remarque : les commandes servent généralement à produire la cible (i.e un nouveau fichier). Une cible peut ne pas représenter un fichier. Dans ce cas, les commandes de cette règle seront toujours exécutées.

4.2 Règles de modèles

Ces règles sont plus polyvalentes car vous pouvez spécifier des règles de dépendance plus complexes. Une règle de modèle ressemble à une règle normale, sauf qu'un symbole (%) apparaît dans le nom de la cible. Les dépendances emploient également (%) pour indiquer la relation entre les noms de dépendance et le nom de la cible. La r ègle de modèle suivante spécifie comment tous les fichiers vst sont formés à partir des vbe.

#exemple de règle pour la synthèse %.vst : %.vbe boog $*

4.3 Définitions de variables

On peut définir des variables en n'importe quel endroit du fichier Makefile, mais une écriture lisible nous amène à les définir en début de fichier. #définitions de variables MY_COPY = cp -r MY_NUM = 42 MY_STRING = "hello" Elles sont utilisables à n'importe quel endroit du Makefile. Elles doivent être précé- dées du caractère $ #utilisation d?une variable dans une règle copie: ${MY_COPY} digicode.vbe tmp/

4.4 Variables prédéfinies

$@ Nom complet de la cible.

$* Nom du fichier cible sans l'extension.

$< Nom du premier fichier dépendant. $+ Noms de tous les fichiers dépendants avec des dépendances doubles répertori

ées dans leur ordre d'apparition.

> $ Noms de tous les fichiers dépendants. Les doubles sont retirés. $? Noms de tous les fichiers dépendants plus récents que la cible. $% Nom de membre pour des cibles qui sont des archives (langage C). Si, par

exemple, la cible est libDisp.a(image.o), $% est image.o et $@ est libDisp.a.