Etude des liens entre la synthèse architecturale et la synthèse au niveau transfert de registres

M. Aichouchi

THÈSE

présentée par

Mohamed AICHOUCHI

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté ministériel du 30 mars 1992)

Spécialité : **Microélectronique**

ETUDE DES LIENS ENTRE LA SYNTHÈSE ARCHITECTURALE ET LA SYNTHÈSE AU NIVEAU TRANSFERT DE REGISTRES

Date de Soutenance : 20 juin 1994

Composition du jury :

Messieurs :	Bernard COURTOIS	*Président*
	Habib MEHREZ	*Rapporteur*
	Georges SAGNES	*Rapporteur*
	Ahmed Amine JERRAYA	
	Jean FREHEL	*Invité*

Thèse préparée au sein du Laboratoire TIMA/INPG

A mes parents,

A ma sœur,

A mes frères,

A mes amis(es) et

A Nidal.

Remerciement

Je tiens à remercier:

Monsieur **Bernard Courtois**, Directeur de recherches au CNRS et Directeur du Laboratoire TIMA, pour m'avoir accueilli au sein du Laboratoire et pour m'avoir fait l'honneur de présider ce jury.

Monsieur **Ahmed Amine Jerraya**, Chargé de recherches au CNRS et Responsable de groupe "System-Level Synthesis", pour les nombreuses discussions et les encouragements qui m'ont beaucoup aidé dans l'orientation et l'avancement de mes travaux.

Messieurs Habib Mehrez, Maître de conférence à l'Université Paris VI, et Georges Sagnes, Professeur à l'ISIM - Université de Montpellier II, pour m'avoir fait l'honneur d'être rapporteurs de cette thèse et membres du jury.

Monsieur Jean Fréhel, Ingénieur à SGS-THOMSON pour avoir accepter mon invitation à être membre du jury.

Tous les membres de l'équipe "System-Level Synthesis" à TIMA: Inhag Park, Kevin O'Brien, Mohamed Abid, Tarek BenIsmail, Mohamed BenMohamed, Elisabeth Berrebi, Adel Changuel, Hong Ding, Goro Furuhashi, Polen Kission, Vijay Raghavan, Maher Rahmouni, Mohamed Romdhani, Carlos Valderrama et Isabelle Es salhiene.

Tous ceux qui ont contribué à la correction et à la préparation de ce manuscrit, et ils sont nombreux: Elisabeth Berrebi, Dominique Brame, Hong Ding, Mohamed Romdhani et surtout **Polen Kission**.

Le reste du Laboratoire TIMA dans son ensemble, notamment:
Alain Guyot, Meryem Marzouki, Mikhael Nicolaidis,
Isabelle Amielh, Mariama Anar Akdim, Hakim Bederr, Chantal Bénis, Mokhtar Boujit, Hicham Boutamine, Patricia Chassat, Hubert Delori, Corinne Durand-Viel, Christophe Garnier, Belgacem Hamdi, Lydie Heusch, Omar Kebichi, Vladimir Kolarik, Nadim Krim, Marcelo Lubaszewski, Salvador Mir, Luis Montalvo, Ali Skaf, Khouldoun Torki, Hedi Touati, André Vacher, Fabian Luis Vargas, ...

ETUDE DES LIENS ENTRE LA SYNTHÈSE ARCHITECTURALE ET LA SYNTHÈSE AU NIVEAU TRANSFERT DE REGISTRES

Résumé

Cette thèse présente une contribution à la compilation de silicium. Elle traite de l'intégration d'un outil de synthèse architecturale dans les environnements de CAO existants. Il s'agit de la personnalisation de l'architecture abstraite, résultat de la synthèse de haut niveau, pour la génération d'une description compatible avec les outils de simulation et de synthèse au niveau transfert de registres. Le but étant d'offrir plusieurs modèles architecturaux utilisant différents modèles de synchronisation afin de couvrir les besoins de différentes applications.

Après une introduction de l'outil de synthèse architecturale AMICAL et de plusieurs modèles architecturaux au niveau transfert de registres, cette thèse présente une méthode et un outil pour la personnalisation de l'architecture abstraite générée par AMICAL et la traduction des fichiers de sortie donnés en SOLAR en leurs équivalents VHDL. Finalement, une étude comparative des différents modèles architecturaux sur plusieurs exemples est détaillée. Cette étude montre qu'il faut plusieurs modèles architecturaux pour différentes applications. Ces modèles architecturaux se différencient entre eux par leur structure, leur bibliothèque de macro-composants et leur modèle de synchronisation utilisé.

Mots-Clefs: Compilation d'architecture, Modèles architecturaux, Macro-composants, Simulation, Synthèse logique.

Abstract

This thesis presents a contribution to the domain of silicon compilation. It deals with the integration of an architectural synthesis tool within the existing CAD environments. The main issue is the personalization of the abstract architecture resulting from the high-level synthesis, in order to generate descriptions which are compatible with the existing RTL simulation and synthesis tools. The goal is to provide several architectural models using different synchronization schemes in order to meet different application needs.

After the introduction of the architectural synthesis tool AMICAL and several RTL architectural models, this thesis presents a method and a tool for the personalization of the abstract architecture generated by AMICAL, and the translation of the resulting SOLAR files into their VHDL equivalents. Finally, a comparative study of the different architectural models using several exemples is detailed. This study indicates that we need several architectural models for different applications. These architectural models differ in the structure, the macro-component library and the synchronization model used.

Keywords: Architectural synthesis, Architectural models, Macro-components, Simulation, Logic synthesis.

Table des Matières

Liste des Figures

Liste des Tableaux

CHAPITRE 1

Introduction

Chapitre 1

Introduction

Cette thèse traite de l'intégration des outils de synthèse architecturale dans les environnements de CAO existants. La plupart des outils de synthèse ont le même objectif; pourtant l'approche "tour de Babel" fait qu'ils utilisent des langages très différents et donc, ne peuvent communiquer entre eux. Ceci implique, par exemple, la difficulté, voire l'impossibilité, de faire le pont entre un outil de synthèse comportementale et un outil de synthèse logique. Cela signifie que beaucoup d'informations sont perdues entre le passage d'un outil à un autre ou alors qu'aucune interaction ne se produit. Avec l'arrivée et l'acceptation générale de langages tels que VHDL, ce problème est surmonté.

Durant le processus de conception d'un circuit, plusieurs langages de description de matériels peuvent être utilisés. Ces langages sont classés selon les niveaux d'abstraction qu'ils manipulent et selon les domaines de description qu'ils couvrent. L'état d'avancement du processus de conception définit le niveau d'abstraction du langage à utiliser, tandis que les aspects du circuit pris en compte définissent les domaines de description auxquels appartient le langage.

Un niveau d'abstraction est caractérisé par les objets qu'il manipule. Un objet peut être un rectangle (niveau layout), un transistor, un opérateur complexe, etc. A chaque niveau d'abstraction correspond un langage de description de matériel. La plupart des travaux [10, 57, 86] distinguent plusieurs niveaux d'abstraction dont voici les plus importants:

- *niveau logique*: Ce niveau correspond au niveau d'abstraction le plus bas qui soit commun aux méthodes de conception de circuits intégrés et aux méthodes de conception de circuits imprimés à base de composants discrets. Le circuit, à ce niveau, peut être décrit comme un ensemble de portes logiques et d'éléments de mémorisation interconnectés. Plusieurs simulateurs logiques sont commercialement disponibles grâce à la longue tradition d'utilisation du niveau logique: CADAT [19], HILO [39], SYNOPSYS [82], etc.

- *niveau transfert de registres*: Dans ce niveau, il n'y a que le mode d'opération qui soit supporté. Les opérations sont interprétées comme des transferts de données entre registres; la donnée transférée peut être modifiée entre deux registres. Le domaine des valeurs est donné, à ce niveau, par un vecteur de bits (ou un vecteur de valeurs "*ininterprétées*"; *mvl7_vector*, par exemple, correspond à un vecteur de valeurs pouvant prendre sept valeurs logiques ('X', '1', '0', 'Z', 'W', 'L' et 'H')). Le niveau transfert de registres est très utile pour des circuits synchrones puisqu'il permet la modélisation et la synchronisation de circuits très complexes [76]. La simulation de descriptions à ce niveau a fait l'étude de beaucoup de recherches dans plusieurs instituts universitaires dans le monde [17, 30, 37].

- *niveau comportemental*: Ce niveau est appelé aussi niveau algorithmique. La description à ce niveau s'attache à décrire le fonctionnement (comportement) d'un modèle sans se soucier d'un éventuel découpage proche de la réalisation, donc de la structure [2]. La description prend la forme d'un algorithme du genre de ceux que l'on utilise dans les langages de programmation classiques. Le temps n'est pas nécessaire à ce niveau, mais il peut intervenir. Les objets manipulés à ce niveau sont aussi des éléments de mémorisation et des opérateurs, mais contrairement au niveau transfert de registres, les éléments de mémorisation et les opérateurs utilisés par une description à ce niveau ne

sont pas liés à des réalisations physiques. Une description comportementale définit des séquences d'opérations manipulant des structures de données [33].

- *niveau système*: Le niveau système est utilisé pour spécifier des systèmes entiers, comprenant des parties logicielles et des parties matérielles [45]. Le but d'une telle spécification est, en général, de fixer les performances et le coût du système. Les éléments manipulés à ce niveau sont des sous-systèmes (processus, mémoires, etc.).

Ces niveaux d'abstraction correspondent aux étapes de la conception d'un circuit sur silicium. Leur définition dépend de la méthodologie de conception utilisée.

Indépendamment du niveau de représentation considéré, il y a toujours un moyen de passer d'un niveau donné au niveau inférieur. Ce passage est réalisé par des "*compilateurs de silicium*".

Le terme "*compilation de silicium*" a été utilisé pour la première fois par Dave Johannsen pour désigner l'assemblage de cellules paramétrées [50]. Depuis, le terme s'est popularisé et a été utilisé dans plusieurs contextes différents. Mais les compilateurs de silicium ont existé bien avant.

Le système EXPL fût le premier système de synthèse de haut niveau. Il permettait de générer l'architecture d'un circuit en partant de sa description, au niveau transfert de registres, spécifiée dans le langage ISPL. C'est à partir de la deuxième moitié des années soixante-dix, que le domaine de la compilation d'architecture a commencé à faire l'objet d'une littérature abondante. Les outils ALERT [32] et MIMOLA [94] sont les précurseurs dans ce domaine. MacPITTS [81] a été le premier compilateur de comportement commercialement disponible. Il a été le premier à avoir généré un dessin de masques en partant d'une description comportementale. Peu de compilateurs de comportement ont pris en compte les contraintes dues aux étapes de compilation de bas niveau. Parmi les compilateurs de comportement, on peut citer, par exemple, et la liste n'est pas exhaustive, les outils suivants:

- HYPER à l'université de Berkeley [26],
- VSS à l'université d'Irvine [58],

- ELF, HAL à l'université de Carleton [34, 69],
- CMU-DA, SAW et ArchitectWorkBench à l'université de Carnegie Melon [29, 85],
- CHIPPE, SLICER, SPLICER à l'université de l'Illinois [18, 68, 72],
- CADDY/DSL à l'université de Karlsruhe [52],
- MIMOLA à l'université de Kiel [61, 62, 94],
- CAMAD à l'université de Linköping [75],
- MOVIE à l'université de Lund [9],
- SCOOP à l'université de Montpellier [79],
- ADAM à l'université de Southern California [35],
- FLAMEL, HERCULES à l'université de Stanford [64, 88],
- DAGAR à l'université de Texas [77],
- LYRA, ARYL à l'université de Tsing Hua [38],
- SPAID à l'université de Waterloo [36],
- BRIDGE à AT&T Bell Labs [89],
- YASC à Control Data Corporation [55],
- YSC et V-compiler à IBM, centre T. J. Watson [11, 14],
- CATHEDRAL I, II, III, et IV à l'IMEC [27, 78],
- PARSIFAL à General Electric [21],
- SILC à GTE [16],
- HARP à NTT LSI [84],
- PHIDEO à Philips [59],
- SYCO au TIMA, Grenoble [45, 49],
- ASYL au CSI, Grenoble [23].

Une grande majorité de ces systèmes génère des circuits composés d'une partie opérative et d'une partie contrôle. Certains de ces compilateurs sont orientés vers la génération de circuits de communication. Dans ce dernier cas, le processus de compilation avantage le parallélisme dans la partie opérative par rapport à la partie

contrôle. Les étapes principales d'un compilateur d'architecture sont l'allocation et l'ordonnancement.

La plupart des techniques d'allocation et d'ordonnancement connues ont été utilisées pour la compilation de silicium. Les techniques les plus utilisées sont les systèmes experts (DAA [53]), les méthodes gloutonnes itératives (EMUCS [40], CHARM [93], MABAL [54]), la programmation dynamique (MIMOLA [62], HAL [70]), la programmation linéaire (ADPS [71]), le découpage en "cliques" (FACET [90]), le recuit simulé ([28]), et l'analyse des chemins d'exécution [13]. Plus de détails sur ces systèmes et techniques sont donnés dans [22, 63].

Malgré les étapes accomplies par les recherches dans le domaine de la compilation de comportement [63], peu d'outils ont été utilisés de manière industrielle. Quelques expériences ont été citées dans la littérature sur des essais industriels de certains de ces compilateurs (CATHEDRAL, ArchitectWorkBench, YSC, MIMOLA, CALLAS).

La non-prolifération de ce type d'outils peut s'expliquer par la difficulté d'intégration de ces outils dans des environnements de conception existants tels que ceux fournis par CADENCE, SYNOPSYS, MENTOR, VALID, Racal-Redac, etc.
En fait, jusqu'à une date récente, peu de ces compilateurs ont utilisé des langages de spécification standards.

Avec l'arrivée de VHDL une nouvelle génération d'outils de compilation de comportement est en train de faire son apparition. L'utilisation de VHDL, en tant que langage standard, facilitera l'intégration de ces outils dans les environnements de CAO de VLSI existants.

Cette thèse présente une approche pour l'intégration du compilateur d'architecture AMICAL dans les environnements de CAO existants. Il s'agit de personnaliser l'architecture abstraite générée en y insérant des éléments additionnels (signaux globaux, bloc de synchronisation, etc.) et de traduire les fichiers de sortie donnés en SOLAR en leurs équivalents VHDL afin de générer une structure compatible avec les outils de simulation et de synthèse au niveau transfert de registres. Elle sera

organisée comme suit:

- Le chapitre suivant présentera, de manière générale, le compilateur d'architecture. Une vue globale du système AMICAL sera présentée tout en donnant sa spécification d'entrée, ses étapes de synthèse et ses modèles d'exécution générés par les différents algorithmes.

- Le Chapitre 3 présentera les différents modèles architecturaux générés par AMICAL. Les deux types de modèle de synchronisation utilisés dans le projet seront détaillés, tout en mentionnant les notions de "*pipeline*" et de test.

- Le Chapitre 4 montrera l'intégration de l'outil AMICAL dans les environnements de conception au niveau transfert de registres (RTL pour *Register Transfer Level*). PAT (*Programmable Architecture Translator*), un outil d'aide à l'ajout des signaux globaux à l'architecture générée et à la traduction de la sortie d'AMICAL, donnée en SOLAR, en son équivalent en VHDL, sera détaillé.

- Le Chapitre 5 donnera quelques résultats comparatifs des différents modèles architecturaux présentés dans le Chapitre 3.

- Finalement le Chapitre 6 conclura cette thèse en donnant quelques perspectives.

CHAPITRE 2

Compilation d'architecture

Chapitre 2

Compilation d'architecture

Ce chapitre décrit le compilateur d'architecture AMICAL. Un compilateur d'architecture est un outil de synthèse de haut niveau permettant d'automatiser le processus de compilation de l'architecture d'un circuit intégré à partir d'une spécification comportementale. AMICAL est un assistant pour la synthèse et l'exploration architecturale. Partant d'une spécification purement comportementale, donnée en VHDL, et d'une bibliothèque externe d'unités fonctionnelles, AMICAL exécute deux tâches essentielles (l'ordonnancement et l'allocation) et génère une architecture abstraite, au niveau transfert de registres, composée d'une partie opérative et d'une partie contrôle. Cette architecture peut être utilisée par les outils de conception au niveau transfert de registres.

2.1 Introduction

Un compilateur d'architecture (appelé aussi compilateur de comportement) permet d'automatiser le processus de compilation de l'architecture d'un circuit intégré à partir d'une spécification comportementale [24, 63]. Ce processus comporte une série de transformations qui permettent de passer d'un niveau comportemental à un niveau plus bas, soit le niveau structurel (ou architectural), tout en satisfaisant un

certain nombre de contraintes [7]. Cette architecture générée doit tenir compte des outils et des méthodologies qui seront utilisés pour la conception logique et physique en vue de la génération des masques.

L'utilisation des compilateurs d'architecture a deux grands avantages:

1- Le fait de travailler à un niveau architectural pour la conception d'un système, sans jamais aller au niveau des cellules de base, des transistors ou des dessins de masque, permet aux concepteurs de se concentrer sur leur tâche principale qui est la conception architecturale. Ainsi, il sera possible d'explorer plusieurs solutions architecturales avant de choisir celle qui présente le meilleur compromis entre les critères physiques (surface, vitesse, consommation, etc.) et les critères économiques (réutilisation de composants existants, temps de conception, fonctionnalité, etc.).

2- Le fait de travailler à un niveau architectural, va permettre d'ouvrir le monde de la conception de circuits aux concepteurs de systèmes. Actuellement, les domaines de conception de systèmes et de circuits sont séparés. D'un côté, on trouve les concepteurs de systèmes qui utilisent des méthodes et des outils, pour la spécification et la conception, basés sur des langages orientés logiciel tels que SDL [25], ESTELLE [42], LOTOS [60], ESTEREL [12], etc. De l'autre côté, on trouve les concepteurs de VLSI qui utilisent des méthodes et des outils, pour la spécification et la conception de circuits, basés sur des langages de description de matériel (dominés par VHDL). De ce point de vue, les circuits sont généralement spécifiés du côté système pour être réalisés par les concepteurs de VLSI.

Pour mieux profiter de tels outils de compilation d'architecture, il faut qu'ils permettent de faciliter l'exploration architecturale en fournissant des outils d'exploration de l'espace des solutions et en permettant au concepteur de modifier les résultats de la synthèse automatiquement. Il faut donc que les outils de compilation d'architecture fournissent les interfaces nécessaires permettant aux utilisateurs non-spécialistes de concevoir des circuits [74].

Ce chapitre donne une vue globale du système AMICAL. Il est organisé sous les aspects suivants:

- La section suivante détaillera l'outil de synthèse architecturale, AMICAL, dans son ensemble. Seront présentés, en détail, son entrée, ses étapes de synthèse, sa sortie ainsi que les modèles d'exécution générés par ses algorithmes de synthèse.
- Finalement ce chapitre se conclura à travers quelques commentaires et par les objectifs visés.

2.2 AMICAL: un outil de synthèse de haut niveau

Partant d'une description comportementale donnée en VHDL et d'une bibliothèque externe d'unités fonctionnelles, AMICAL, outil de synthèse de haut niveau, génère une architecture abstraite d'un circuit qui peut être facilement utilisée en entrée des outils de conception au niveau transfert de registres existants (figure 2.1).

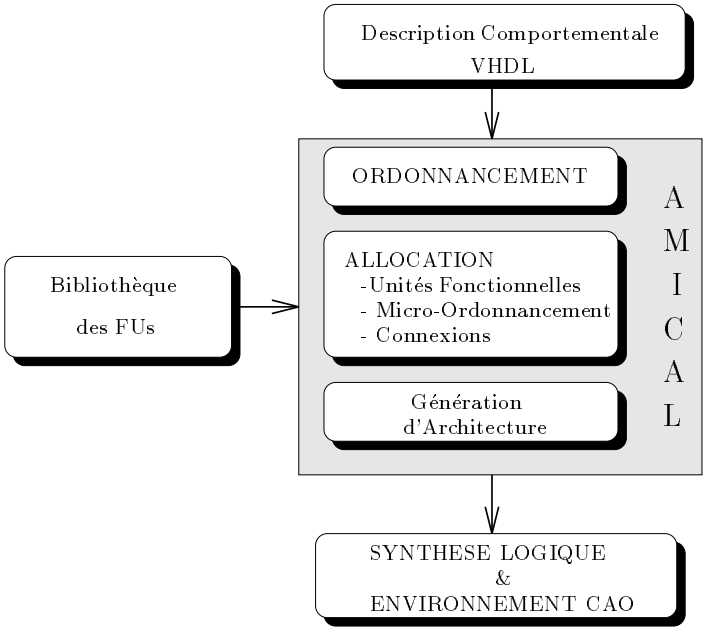


Figure 2.1: *Vue Globale d'AMICAL.*

La première étape consiste à traduire la description VHDL en une table de transitions très proche du format SOLAR [46, 66]. Le sous-ensemble VHDL accepté par AMICAL est assez large pour permettre la description d'applications industrielles. AMICAL réalise les étapes classiques d'un compilateur de comportement, soient l'ordonnancement et l'allocation.

L'architecture générée par AMICAL (appelée architecture abstraite) est composée d'une partie opérative et d'une partie contrôle. Cette architecture, décrite au niveau transfert de registres, peut facilement être interfacée avec les outils de simulation et de synthèse logique existants.

AMICAL se présente comme un environnement pour la compilation d'architecture, où la synthèse peut se faire manuellement, automatiquement, pas à pas ou partiellement manuellement et partiellement pas à pas. En mode automatique, les étapes de la synthèse sont enchaînées sans intervention du concepteur. En mode pas à pas, les tâches sont exécutées itération par itération à l'initiative du concepteur. Ce mode permet à ce dernier, de suivre l'évolution du processus de synthèse et d'intervenir à tout moment en passant en mode manuel, pour modifier les décisions des algorithmes de synthèse. En mode manuel, le concepteur a en charge tout le processus de la synthèse. Dans ce cas, le système agit en tant qu'assistant; il vérifie la cohérence des décisions prises et ne permet que les opérations correctes. En mode manuel, le concepteur est responsable de l'efficacité du résultat.

AMICAL est un assistant pour la compilation architecturale. Il permet au concepteur de guider la synthèse à travers une interface graphique. La figure 2.2 montre une vue du système AMICAL, pendant la synthèse. Cette vue est composée de quatre fenêtres dont trois sont permanentes pendant tout le processus de synthèse.

La fenêtre de droite montre un exemple de description VHDL, elle ne fait pas partie des trois fenêtres d'AMICAL. Les deux fenêtres de gauche (en haut) montrent le résultat de la compilation. La première fenêtre du haut expose le résultat de l'ordonnancement sous la forme d'une table de transitions. La fenêtre du milieu montre la partie opérative; ici le résultat d'une allocation exécutée automatiquement est représenté. La fenêtre du bas est utilisée comme une fenêtre d'information et de dialogue, cette fenêtre d'information permet de suivre la synthèse en affichant la commande en cours et toute erreur rencontrée. C'est elle qui offre à l'utilisateur AMICAL la flexibilité de passer d'un mode à un autre.

L'outil de synthèse AMICAL maintient des liens entre les différents aspects de la description. Par exemple, dans la figure 2.2, le système montre la correspondance entre la table de transitions et la partie opérative générée (les ressources nécessaires à l'exécution de la transition 7).

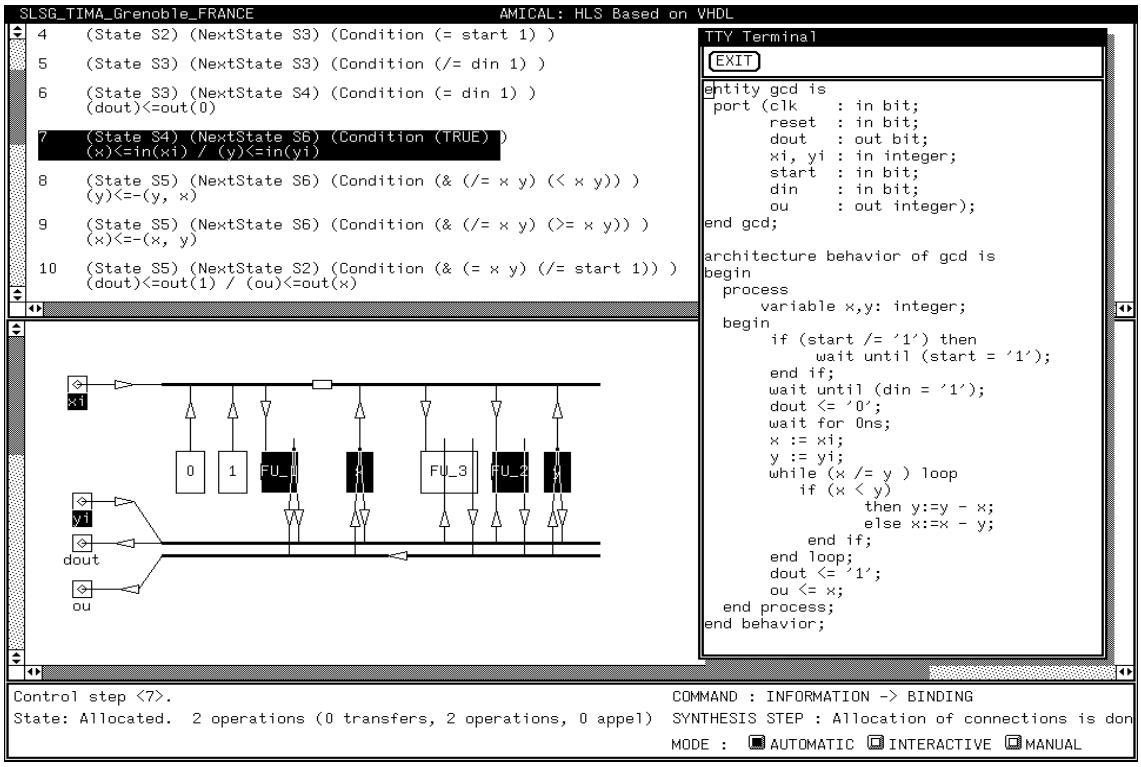


Figure 2.2: *Vue d'ensemble du système AMICAL.*

2.2.1 Architecture cible d'AMICAL

L'architecture cible d'AMICAL est un modèle général composé d'une partie contrôle, d'un ensemble d'unités fonctionnelles et d'un réseau de communication. Les deux dernières parties constituent la partie opérative. Le schéma de la figure 2.3 montre une telle architecture.

Cette architecture est synchrone, parallèle et hétérogène.
Elle est synchrone grâce au contrôleur qui ordonne le séquencement des opérations exécutées par les unités fonctionnelles et le réseau de communication. L'organisation de la synchronisation est "*virtuelle*". L'hypothèse faite est qu'à chaque cycle de base une nouvelle commande est envoyée à la partie opérative. La commande inclut l'activation de plusieurs chemins de communication (*transferts*). La synchronisation réelle du système complet sera détaillée dans le Chapitre 3.
L'architecture est parallèle; elle doit inclure plusieurs unités fonctionnelles qui peuvent s'exécuter en parallèle. La granularité du parallélisme est fixée par le processus

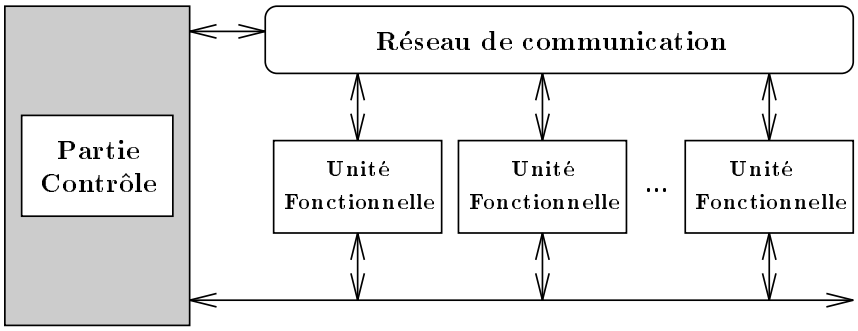


Figure 2.3: *Architecture cible d'AMICAL.*

de synthèse. L'ordonnancement effectué en début de synthèse est fait automatiquement; il définit la granularité maximale du parallélisme existant. Cependant, par des interventions manuelles, l'utilisateur peut modifier cet ordonnancement pour un parallélisme moindre.
L'architecture est hétérogène puisqu'elle permet l'utilisation de composants produits par d'autres environnements de conception et vice versa.

2.2.2 Spécification d'entrée

Comme mentionné plus haut, le processus de compilation part d'une spécification comportementale utilisant un sous-ensemble de VHDL donné. Cette description consiste en une paire *Entity/Architecture*, où l'architecture est limitée à un seul processus. Le sous-ensemble VHDL accepté est suffisamment large pour permettre la description de circuits même complexes, puisqu'il contient presque toutes les instructions séquentielles de VHDL [41]: les boucles, les branchements, les instructions de coupure de séquence (*exit*) et les appels de fonctions et de procédures. Ceci permet de décrire de grands exemples complexes et réels. De plus, l'utilisation de paquetages et de procédures VHDL permet d'importer les blocs existants dans une description comportementale.

L'entité est restreinte à la déclaration des signaux d'entrées/sorties, ceci étant considéré comme suffisant pour les besoins de la synthèse. La partie déclarative de l'architecture contient seulement les déclarations des signaux, des constantes, des

types et des variables. Les instructions structurelles, ainsi que celles qui pourraient être utilisées pour la simulation, telles que *configuration*, *alias*, *file* et *component*, ne sont pas incluses à ce niveau. La partie instruction de l'architecture contient la description d'un seul processus pouvant inclure la définition de fonctions ainsi que de procédures.

Un exemple typique d'une description comportementale est montré dans la figure 2.4.

```
1   Entity Bubble is
2       port (  reset, clk, start : IN BIT;
3               ackout, validin   : IN BIT;
4               datain            : IN INTEGER;
5               ackin, validout   : OUT BIT;
6               dataout           : OUT INTEGER);
7   end Bubble;
8   Architecture Behavior of Bubble is
9   Type memory is array(0 to 255) of INTEGER;
10  Begin
11      Bubble_sort: PROCESS
12      variable i, k, iter, size, t1, t2, t3, t4: INTEGER;
13      variable ram: memory;
14          Function get255 return integer is
15          Begin
16              return(255);
17          end get255;
18          Procedure fillram is
19          Begin
20              i := 0; size := get255;
21              while (i <= size) loop
22                  wait until (validin = '1');
23                  t1 := datain; ackin <= '1';
24                  wait until (validin = '0');
25                  ram(i) := t1; ackin <= '0'; i := i+1;
26              end loop;
27          end fillram;
28          Procedure emptyram is
29          Begin
30              i := 0; size := get255;
31              while (i <= size) loop
32                  if (ackout /= '0') then wait until (ackout = '0'); end if;
33                  t1 := ram(i); validout <= '1'; dataout <= t1;
34                  wait until (ackout = '1');
35                  validout <= '0'; i := i+1;
36              end loop;
37          end emptyram;
38      Begin
39          if (start /= '1') then wait until (start = '1'); end if;
40          fillram; i := 1;
41          while i <= size loop
42              iter := size;
43              while (iter >= i) loop
44                  t1 := iter-1; t2 := decr2(iter); t3 := ram(t1); t4 := ram(t2); iter := t1;
45                  if (t3 < t4) then ram(t1) := t4; ram(t2) := t3; end if;
46              end loop;
47              i := i+1;
48          end loop;
49          emptyram;
50      end PROCESS Bubble_sort;
51  end Behavior;
```

Figure 2.4: *Un exemple d'une description comportementale.*

La description de la figure 2.4 représente un programme d'un tri à bulles (*bubble-sort*) [15], un algorithme pris comme exemple pour montrer le sous-ensemble de VHDL accepté par AMICAL. L'algorithme commence par remplir un tableau de 255 entiers lus d'une source externe et selon un protocole bien fixe (procédure *fillram*). La méthode du tri à bulles est alors utilisée pour ordonner les éléments du tableau dans l'ordre croissant. Finalement, le résultat est récupéré en sortie selon un protocole fixe (défini par la procédure *emptyram*).

2.2.3 Bibliothèque des unités fonctionnelles

Le système AMICAL utilise une bibliothèque d'unités fonctionnelles (*Functional Units*: FUs). Si la comparaison devait être faite avec un autre système, par exemple CATHEDRAL, ces FUs sont comparables aux unités d'exécution (*Execution Units*: EXUs) [65]. Cependant, si dans CATHEDRAL, les EXUs sont extraites automatiquement de la description comportementale, les unités fonctionnelles doivent être fournies par le concepteur utilisateur d'AMICAL. Même si cette approche est moins automatique, elle permet plus de flexibilité pour l'utilisation des blocs existants.

La figure 2.5 montre un exemple de fichier de bibliothèque qui contient une liste d'unités fonctionnelles nécessaires pour la compilation de l'algorithme du tri à bulles (*Bubble-sort*).

```
1   (Bibliotheque Bub
2       (Path ./Library)
3       (Functional_Unit
4           ram (Operateur read write)
5           IO (Operateur in out)
6           SUB (Operateur - decr2 get255)
7           ADD (Operateur + get255)
8           ALU3 (Operateur + - decr2 get255)
9       )
10  )
```

Figure 2.5: *Un exemple du fichier de bibliothèque (.lib) pour l'algorithme de tri à bulles.*

Ce fichier indique:

- le chemin pour la bibliothèque d'unités fonctionnelles (mot clé *Path*); le répertoire indiqué regroupe divers fichiers (un par unité fonctionnelle) décrivant le schéma d'exécution de chacune des FUs par les signaux de contrôle et les transferts correspondant à chaque opération.
- la liste des unités fonctionnelles (mot clé *Functional_Unit*) et les opérations qu'elles peuvent exécuter.

Pour chaque unité fonctionnelle, la liste des opérations qu'elle peut exécuter correspond à la liste précédée par le mot clé *Opérateur*. Les caractéristiques de chaque unité fonctionnelle sont quant à elles décrites dans un autre fichier (fichier du répertoire *Library* ici); ce fichier est nommé du nom de l'unité fonctionnelle suivi de ".fu"; (figure 2.6(d) correspondant à Ram.fu).

Une unité fonctionnelle (FU) peut exécuter une ou plusieurs opérations; ces opérations comprennent aussi bien les opérations standards (addition, multiplication, opérations logiques, etc.), que toute nouvelle opération programmée par le concepteur. L'unité fonctionnelle peut être un bloc complexe tel qu'une mémoire cache, une unité d'entrée/sortie, etc.

Une FU peut être appelée à l'intérieur de la spécification comportementale par un appel de procédure afin de réaliser une opération donnée. L'utilisation de bibliothèques d'unités fonctionnelles offre la possibilité d'utiliser des circuits réalisés préalablement comme des blocs pour de nouveaux systèmes [73, 74].

Une unité fonctionnelle peut être spécifiée à des niveaux d'abstraction différents comme le montre la figure 2.6.

Cet exemple décrit une mémoire appelée *RAM*. Du point de vue conceptuel, la mémoire est un objet capable d'exécuter deux opérations: une écriture (*Write*) et une lecture (*Read*) qui manipulent une donnée commune (*M*, figure 2.6(a)). Au

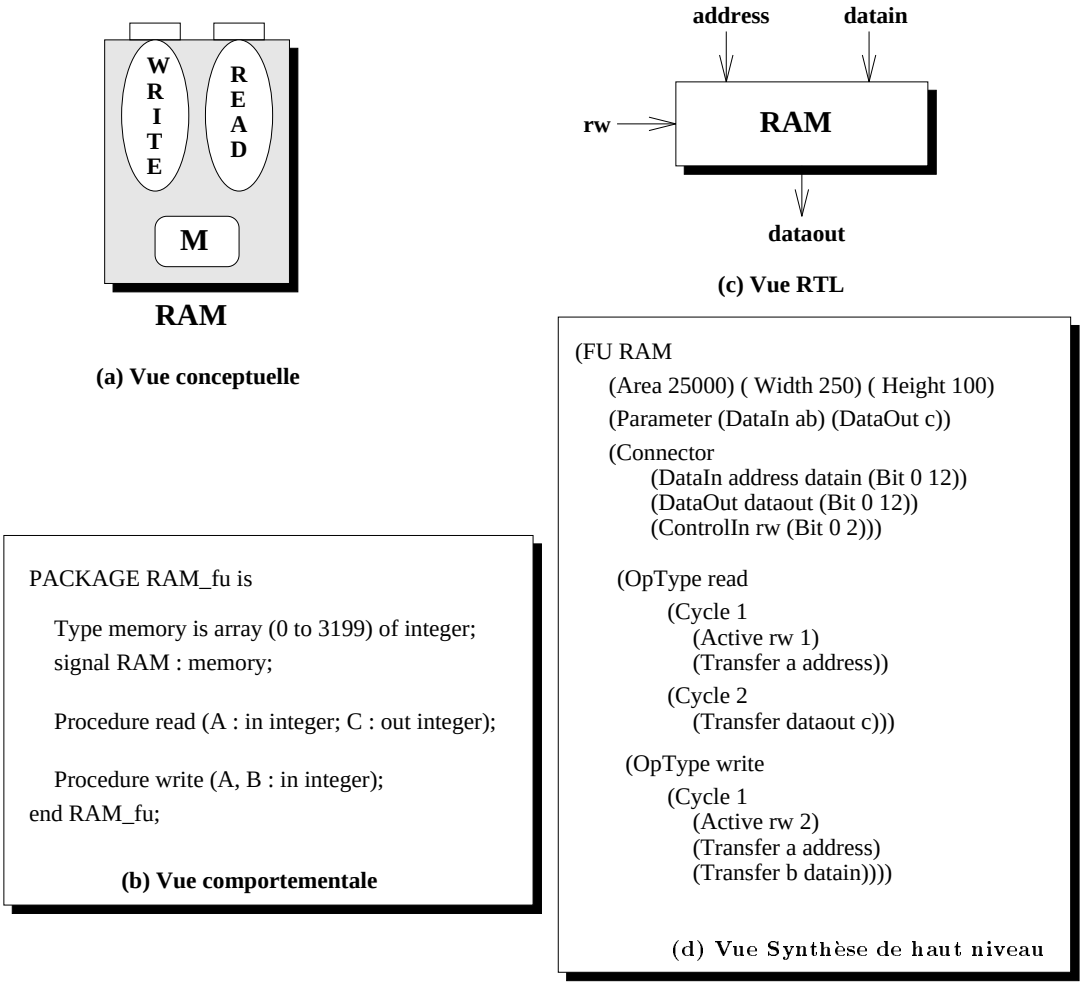


Figure 2.6: *Différents types d'abstraction d'une unité fonctionnelle.*

niveau transfert de registres (figure 2.6(b)), la FU peut être décrite en VHDL dans un paquetage qui inclut deux procédures ayant accès à un objet commun (un signal global) *array RAM*. Chaque procédure spécifie l'exécution d'une opération. La figure 2.6(c) montre une vue externe de la mémoire. Elle a deux ports d'entrée principaux (*datain* et *address*), une sortie (*dataout*) et un signal de commande (*rw*) pour la sélection de la procédure à exécuter. La vue synthèse de haut niveau (figure 2.6(d)) englobe les vues *comportementale* et *transfert de registres*. Elle comprend:

- L'interface de l'unité fonctionnelle.
- Les paramètres d'appel de l'unité fonctionnelle (appel de procédures).

- L'ensemble des opérations exécutées par l'unité fonctionnelle.
- Le micro-ordonnancement pour chaque opération.

Dans la spécification d'entrée d'AMICAL, une unité fonctionnelle est invoquée par un appel de procédure. Un des traits majeurs d'AMICAL est que la bibliothèque d'unités fonctionnelles est extensible permettant l'utilisation de blocs déjà synthétisés.

2.2.4 Etapes de synthèse

Après la compilation de la spécification d'entrée, AMICAL doit exécuter un certain nombre de tâches afin de générer l'architecture abstraite. Ces tâches sont les suivantes:

2.2.4.1 *Ordonnancement*

AMICAL utilise un algorithme d'ordonnancement à boucles dynamiques (*Dynamic Loop Scheduling*) [67]. Cet algorithme est adapté à des circuits dominés par le flux de contrôle et décrits en VHDL. C'est une approche améliorée de l'algorithme à base de chemins (*path-based approach*) proposé par Camposano [20] et Fisher [31]. L'outil d'ordonnancement lit la description donnée en VHDL et produit une machine d'états finis (*Finit State Machine; FSM*) sous forme de table de transitions. Chaque transition est définie par un état courant, une condition, un état suivant et un ensemble d'opérations à exécuter. Cette machine d'états finis est représentée selon le modèle du graphe de Mealy. Les transitions portent le nom de *macro-cycles*. Un macro-cycle peut utiliser plusieurs cycles de base (cycles d'horloge) pour l'exécution de ses opérations. La fenêtre, en haut à gauche de la figure 2.2 montre la table de transition de l'exemple du pgcd (plus grand commun diviseur); elle est composée de six états et de douze transitions. Toutes les opérations de la même transition peuvent être exécutées en parallèle.

2.2.4.2 *Allocation*

Après l'ordonnancement, les étapes suivantes de la synthèse architecturale nécessitent deux types d'information: la description ordonnancée et la bibliothèque externe des unités fonctionnelles. L'architecture résultante est à base de bus. Il n'y a pas de limitation en ce qui concerne le nombre de bus. La synthèse architecturale invoque quatre étapes [74] qui sont les suivantes:

- Allocation des unités fonctionnelles: Cette étape associe une unité fonctionnelle à chaque opération (autre que les transferts) dans la table de transitions.
- Micro-ordonnancement: Cette étape génère le schéma d'exécution de chaque opération selon l'unité fonctionnelle utilisée. Chaque opération est décomposée en un ensemble de transferts entre registres. Les transferts sont ordonnés en *micro-cycles* selon la technique ASAP (*As Soon As Possible*; le plus tôt possible). Chaque micro-cycle contient un ensemble de transferts parallèles qui s'exécutent en un seul cycle de base.
- Placement des registres: Cette étape place les registres et les unités fonctionnelles côte à côte (linéaire) de sorte à réduire le plus possible les pistes de bus (connexions). Puisque AMICAL utilise une architecture cible à base de bus, le placement des registres et des unités fonctionnelles est une opération ayant un fort impact sur l'efficacité du résultat.
- Allocation des connexions: Cette dernière étape produit la structure des bus. Pour chaque transfert, un ensemble de connexions fait de fils d'interconnexion, d'interrupteurs et de bus doit être alloué de telle sorte que les transferts parallèles ne se disputent pas les mêmes ressources. L'efficacité de cette étape dépend largement des étapes précédentes.

Toutes ces étapes, excepté le micro-ordonnancement, utilisent une approche, dite "*constructive par itération*", similaire à celle utilisée par APOLLON [43]. A chaque étape un nouvel élément est alloué ou placé. Tous ces algorithmes sont implémentés

de sorte à permettre le mélange des deux modes (manuel et automatique). Les quatre étapes de la synthèse architecturale peuvent être exécutées en séquence, soit automatiquement (l'exécution des quatre étapes est déclenchée par une seule commande), soit étape par étape. Chacune de ces étapes peut être exécutée automatiquement, pas à pas ou manuellement.

2.2.4.3 *Génération d'architecture*

AMICAL produit une architecture abstraite composée de deux sous-systèmes: une partie contrôle et une partie opérative. Cette architecture est donnée sous une forme intermédiaire nommée SOLAR [46, 66] par deux fichiers décrivant les deux parties mentionnées ci-dessus et un fichier décrivant l'interconnexion entre elles (circuit global).

2.3 Différents modèles d'exécution utilisés par le système AMICAL

L'objectif principal de cette section est de montrer les différents modèles d'exécution pouvant être générés par les différents algorithmes d'AMICAL.
Durant le passage du niveau comportemental au niveau transfert de registres, la description est raffinée. Ainsi au niveau transfert de registres, le cycle de base devient le cycle d'horloge.

Partant d'une description comportementale, AMICAL génère et gère plusieurs descriptions dont certaines dans un format propre à AMICAL, et ce, dans le but de générer une architecture du circuit sous la forme d'une description structurelle au niveau transfert de registres.
Les plus importants de ces modèles d'exécution sont:

2.3.1 Niveau comportemental

Comme mentionné plus haut (section 2.2.2), AMICAL accepte en entrée une spécification fonctionnelle (ou comportementale) décrite suivant un sous-ensemble VHDL [66]. Cette spécification est limitée à un seul processus ne contenant que des instructions séquentielles.

Dans cette section, un exemple de représentation comportementale, illustrant la description compacte d'exemples complexes, sera utilisée. Une des plus importantes justifications pour un algorithme d'ordonnancement dans la synthèse de haut niveau est de permettre les descriptions de comportement sans spécification explicite de la synchronisation.

Une description comportementale peut contenir plusieurs boucles (avec possibilité d'imbrication), des exceptions globales, des instructions *wait* et des appels de procédures (c'est-à-dire les structures qui reflètent les propriétés inhérentes de circuits de contrôle).

Avoir plusieurs instructions *wait* dans le même processus permet de décrire des circuits pouvant se synchroniser avec d'autres circuits via l'échange d'informations, le partage de données, etc. Un exemple général et classique d'une description comportementale est montré dans la figure 2.7.

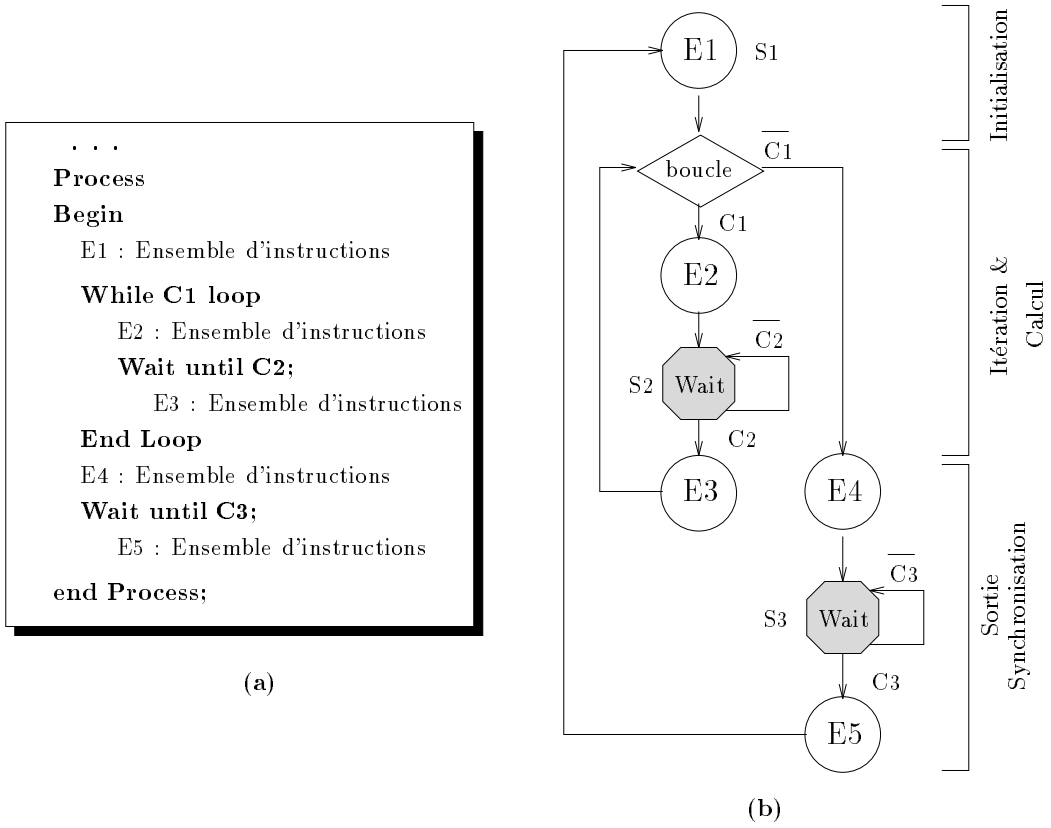


Figure 2.7: *Exemple de description comportementale.*
(a) *Processus VHDL*, (b) *Graphe de flux de contrôle correspondant.*

Ce modèle contient trois types de traitement qui consistent en une phase d'initialisation, une phase de calcul contenant une boucle d'attente sur les données d'entrée à chaque itération, et une phase de synchronisation pour la sortie des résultats par l'intermédiaire d'un acquittement. Chacun des nœuds E1-E5 dans la figure 2.7 peut contenir aussi bien aucune instruction que plusieurs instructions. Les instructions *wait* sont utilisées pour la synchronisation.

Un des traits importants du sous-ensemble VHDL accepté par l'ordonnancement à boucles dynamiques [67] est l'utilisation des exceptions globales. Pour montrer

cette utilisation et son importance, l'exemple d'un répondeur téléphonique est utilisé (figure 2.8). Cette figure montre la hiérarchie du contrôleur. L'état initial est l'état *Wait_For_A_Call*, dans lequel le système est initialisé et le flot de contrôle attend un appel et trois sonneries. Quand il les a reçues toutes les trois, une transition à l'état *OffHook* se produit. Dans cet état, un message est émis et le correspondant peut laisser un message, écouter les messages déjà enregistrés ou raccrocher. S'il raccroche, une transition immédiate vers l'état *Wait_For_A_Call* doit se faire. Une transition identique se fait quand les limites de temps imposées par le répondeur sont dépassées.

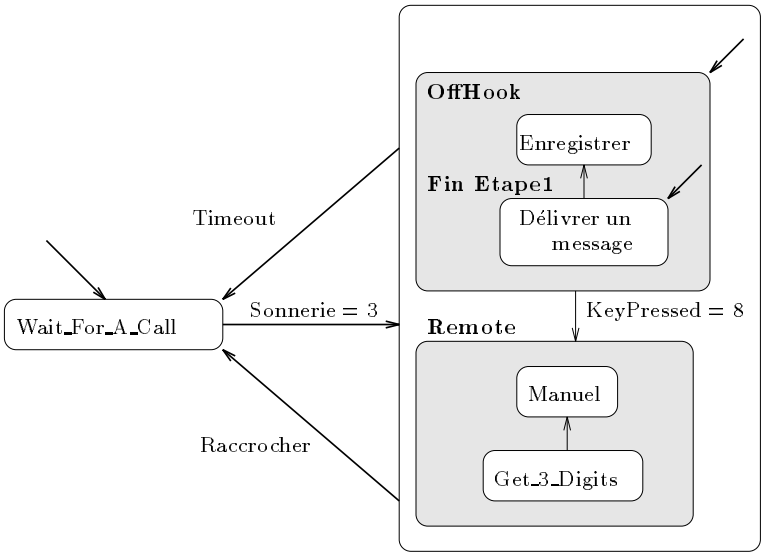


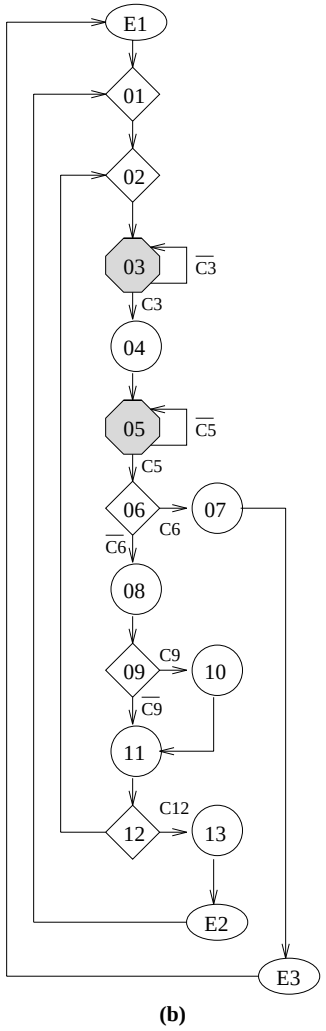
Figure 2.8: *Représentation du contrôleur du répondeur téléphonique.*

Le contrôleur est modélisé avec un seul processus VHDL qui contient sept boucles imbriquées, organisées comme dans la figure 2.9. La description complète de ce processus contient plus de 200 lignes de code VHDL (voir Annexe A). Afin d'avoir un exemple simple, une seule boucle, *Get_3_Digits*, est représentée (figure 2.9(a)). Le reste du code est représenté par les nœuds E1, E2 et E3. Pendant l'exécution d'une boucle, s'il y a une exception globale (le correspondant raccroche, par exemple), il y aura une sortie de la hiérarchie des boucles pour donner le contrôle à la boucle *Wait_For_A_Call*. Le sous-ensemble VHDL permet l'utilisation des instructions *exit* et *next* pour générer ces transitions. La synchronisation n'est pas explicite à ce

niveau d'abstraction. La durée prise en compte par une instruction *wait* n'est pas connue dans ce cas et ne peut être constante entre les différentes itérations d'un processus.

```
E1 : Ensemble d'instructions
01   OffHook : Loop
02      Get_3_Digits : Loop
03         Wait Until (KeyPressed = 0);
04         Timeout := FU_REM(ElapsedTime, 20);
05         Wait Until   ((ElapsedTime = Timeout)  OR
                  (keyPressed /= 0)  OR (HangUp = '1'));
06         If   ((HangUp = '1')   OR (ElapsedTime = Timeout))
07         Then Exit OffHook;   End if;
08         NextDigit := PasswdROM(DigitCount);
09         If   (KeyPressed /= NextDigit)  Then
10            FalsePasswd := '1';     End if;
11          DigitCount := DigitCount + 1;
12          If   (DigitCount = 3) Then
13             Exit Get_3_Digits;
            End if;
        End Loop   Get_3_Digits;
        E2 : Ensemble d'instructions
     End Loop   OffHook;
     E3 : Ensemble d'instructions
```

(a)



(b)

Figure 2.9: *Représentation VHDL du répondeur téléphonique.*
(a) *Un extrait du code VHDL*, (b) *Graphe de flux de contrôle correspondant.*

La figure 2.9(b) montre le graphe de flot de contrôle correspondant au morceau du programme VHDL donné par la figure 2.9(a).

2.3.2 Niveau étape de contrôle (macro-cycle)

L'algorithme de l'ordonnancement utilisé par AMICAL est connu sous le nom de "ordonnancement à boucles dynamiques" (DLS; *Dynamic-Loop Scheduling*) et

est adapté aux algorithmes représentés par le flux de contrôle. Son principe est basé sur "l'ordonnancement à base de chemins" (PBS; *Path Based Scheduling*). Le temps total d'exécution d'un algorithme est minimisé en prenant en compte les différents chemins de contrôle possibles rencontrés dans l'algorithme. L'algorithme identifie les instructions *wait* comme des changements d'états (d'autres changements d'états peuvent se produire à cause de contraintes imposées par l'utilisateur ou de dépendance de données). L'algorithme génère une architecture avec un maximum de parallélisme utilisant la technique AFAP (*As Fast As Possible*; le plus rapidement possible). La sortie de l'outil d'ordonnancement est une table de transitions. Chaque entrée de cette table contient le nom de l'état courant, une liste de conditions à évaluer, le nom de l'état suivant, si la condition est vraie, et une liste d'opérations à exécuter pendant la transition. La table de transitions est en effet une représentation d'une machine d'états finis de Mealy dont la représentation globale est donnée dans la figure 2.10.

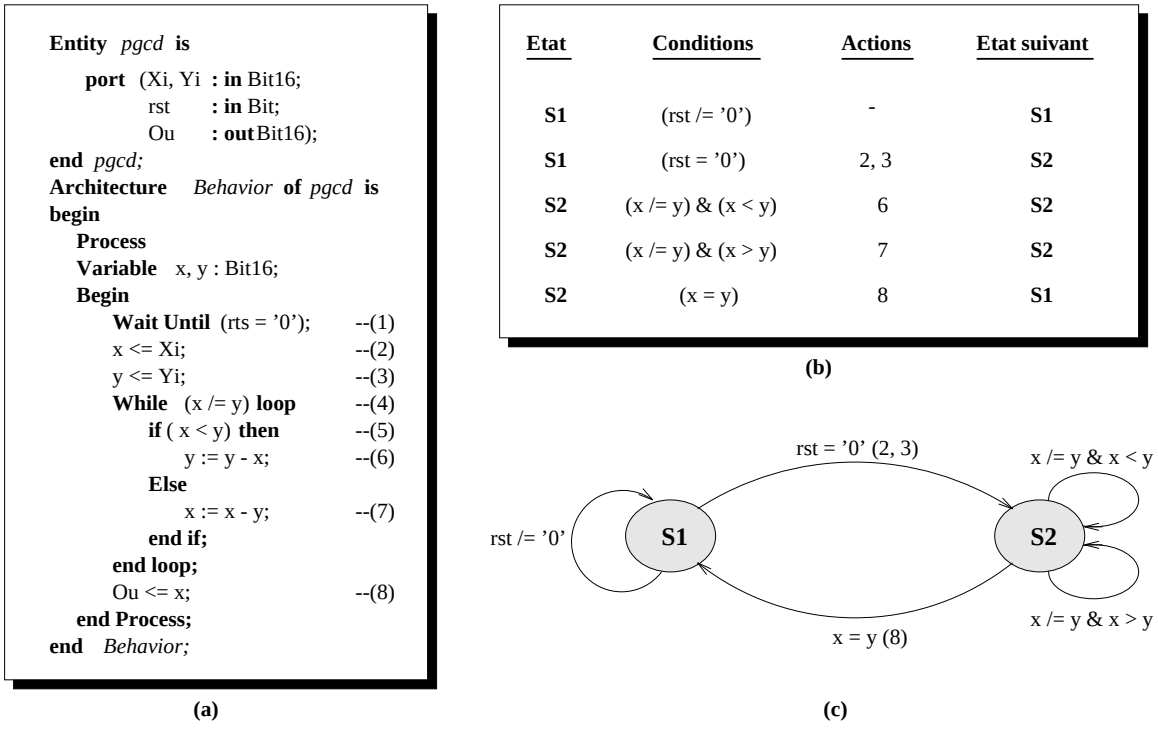


Figure 2.10: *Exemple de machine d'états finis comportementale.*
(a) *description VHDL*, (b) *table de transitions*, (c) *machine d'états finis.*

Cette table d'états représente une machine d'états finis comportementale dont

le niveau d'abstraction est toujours de haut niveau. En représentation VHDL par exemple, ce modèle peut être représenté par un seul processus contenant une seule instruction *wait* ou bien une liste de sensitivité et une instruction *case*. L'exécution de chaque option de l'instruction *case* correspond à une "étape de contrôle". Dans chaque étape de contrôle, toutes les conditions dans l'état courant sont évaluées et pour chaque condition vraie les opérations (actions) sont exécutées.

Dans ce cas, l'outil d'ordonnancement a déjà fixé l'ordre définitif d'exécution des opérations. Les opérations qui doivent s'exécuter en parallèle sont connues, ainsi que les opérations qui doivent se partager les même ressources. Ce modèle peut donner une bonne idée en ce qui concerne la synchronisation du circuit et la surface globale du circuit résultant.

2.3.3 Niveau transfert de registres (micro-cycle)

Une fois que les unités fonctionnelles ont été allouées aux opérations dans chaque étape de contrôle, le nombre de cycles d'horloge de base, nécessaire pour chaque transition, peut être identifié. Ces cycles d'horloge de base sont connus sous le nom de "micro-cycles" (μ-cycles).

Le modèle montré dans la figure 2.11 est connu sous le nom de "machine d'états finis au niveau transfert de registres" (*RTL FSM*) où chaque transition sera décomposée maintenant en un ensemble de transferts entre registres au niveau du micro-ordonnancement (voir section 2.2.4.2). Par exemple, pour exécuter une opération d'addition, plusieurs étapes séquentielles sont nécessaires. Ces étapes (ou μ-cycles) dépendent de l'unité fonctionnelle utilisée.

Les transferts entre registres correspondent aux micro-instructions (μ-instructions) qui doivent s'exécuter dans la partie opérative pendant un micro-cycle (figure 2.11(b)). Un μ-cycle correspond à un cycle de base (figure 2.12) pendant lequel:

- le contrôleur évalue les conditions présentes qui lui permettent de définir la transition à effectuer; la sélection d'une transition se traduit par l'activation

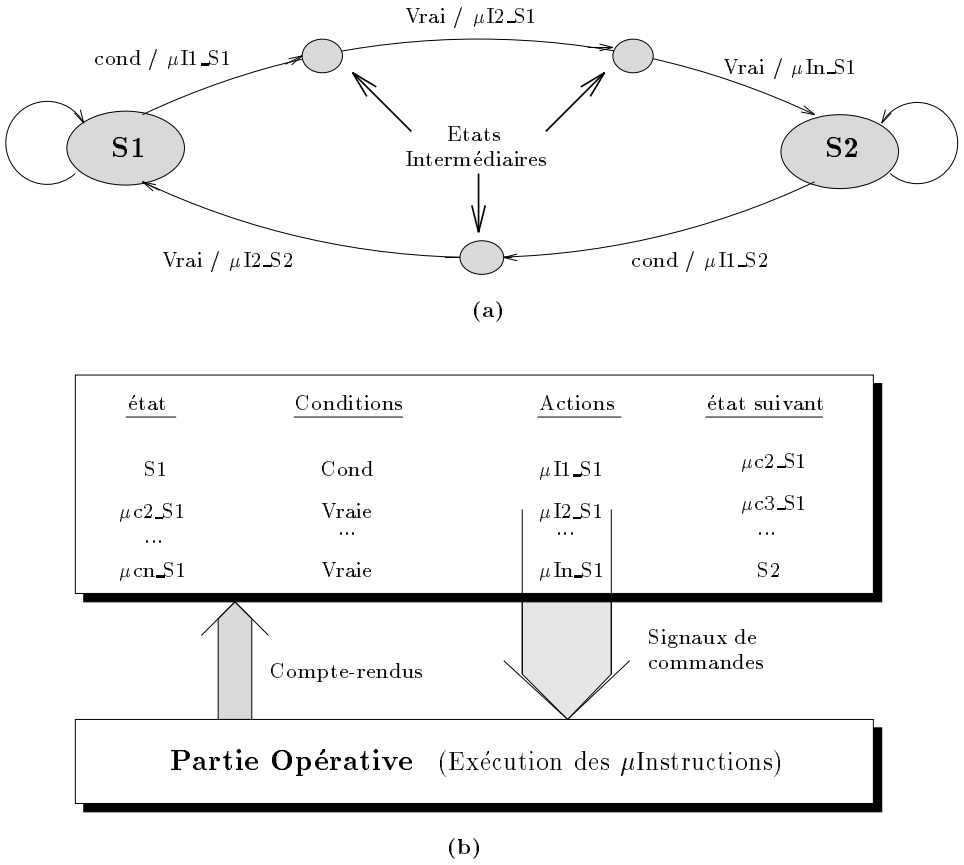


Figure 2.11: *Machine d'états finis au niveau transfert de registres.*
(a) *Machine d'états finis*, (b) *Modèle d'exécution.*

de signaux de commande vers la partie opérative.

- le chemin de données effectue un certain nombre de transferts de type R1 -> R2 (définis par le vecteur de commande venant de la partie contrôle), certaines de ses unités fonctionnelles peuvent être activées, la partie opérative envoie aussi pendant ce même délai des compte-rendus au contrôleur.

En conclusion, le passage du niveau d'abstraction comportemental au niveau transfert de registres, permet de raffiner le modèle d'exécution jusqu'au cycle d'horloge. Ceci dit, la synchronisation des circuits générés reste toujours basée sur une horloge virtuelle; il n'y a pas d'aspects matériels dans les descriptions générées tels que l'horloge et la remise à zéro. Ces signaux (appelés signaux globaux) seront

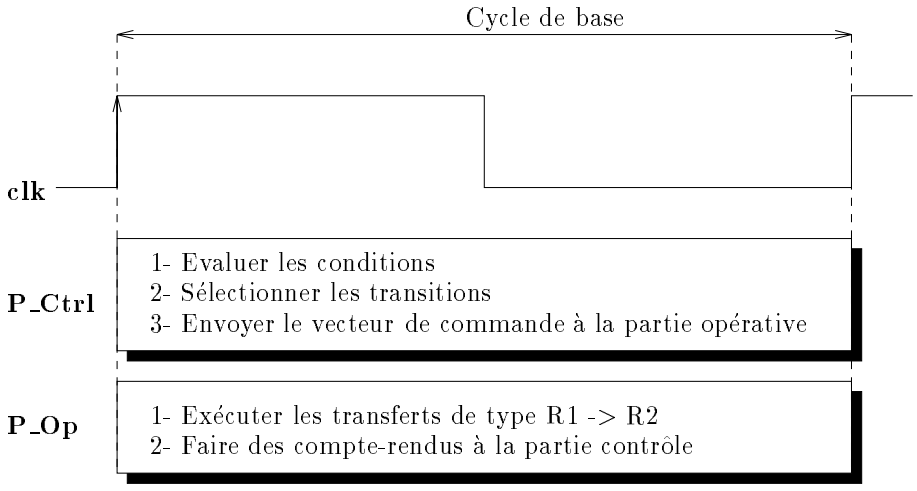


Figure 2.12: *Exécution d'un micro-cycle.*

rajoutés à l'architecture générée par un autre outil, appelé PAT (Programmable Architecture Translator), qui sera présenté dans le Chapitre 4.

2.4 Conclusion

Un outil de compilation architecturale prend une spécification comportementale et un ensemble de contraintes, et génère une structure qui implémente le comportement désiré d'un circuit et qui satisfait toutes les contraintes.

Dans ce chapitre, une vue globale du système AMICAL a été présentée tout en donnant une présentation de sa spécification d'entrée, ses étapes de synthèse et ses modèles d'exécution. AMICAL part d'une description comportementale donnée en VHDL et d'une bibliothèque externe d'unités fonctionnelles et génère une architecture abstraite au niveau transfert de registres composée d'une partie opérative et d'une partie contrôle. Afin de manipuler des circuits très complexes, AMICAL permet l'utilisation d'unités fonctionnelles complexes ainsi que la réutilisation de composants existants à l'intérieur de nouveaux circuits.

Le chapitre suivant présentera les différents modèles architecturaux générés par AMICAL. La personnalisation de l'architecture abstraite pour la génération d'architectures détaillées ainsi que deux différents types de modèles de synchronisation adaptés pour AMICAL seront détaillés tout en mentionnant les notions du *pipeline* et du test.

CHAPITRE 3

Modèles architecturaux

Chapitre 3

Modèles architecturaux

Ce chapitre présente les différents modèles architecturaux générés par AMICAL. Partant d'une description comportementale donnée en VHDL et d'une bibliothèque externe d'unités fonctionnelles, AMICAL génère une architecture abstraite de base composée d'une partie opérative et d'une partie contrôle. Selon le mode d'utilisation (architecture auto-testable ou pas) et le modèle de synchronisation utilisé (avec ou sans "pipeline"), cette architecture est divisée en deux grandes classes de modèles: architecture simple (composée d'un contrôleur et d'un chemin de données) et architecture avec interface entre les deux parties (modèle adapté pour le test ou le "pipeline"). Chaque modèle sera présenté en détail, ainsi que la bibliothèque de composants utilisée pour toute synthèse AMICAL.

3.1 Introduction

La description comportementale d'un circuit décrit son architecture externe, c'est-à-dire, la manière dont il peut être utilisé [8]. Cette architecture est décrite par l'interface du circuit (entrées/sorties accessibles à l'utilisateur) et la fonction réalisée par ce dernier.

La description architecturale (structurelle) d'un circuit est obtenue après la synthèse de haut niveau (AMICAL, par exemple) et est définie par un modèle d'organisation, une bibliothèque d'éléments utilisés (FUs) et un schéma de synchronisation.

L'architecture interne d'un circuit peut être vue comme étant l'organisation globale du circuit, celui-ci est composé d'un ensemble de sous-systèmes interconnectés. Un sous-système peut être un module standard (contrôleur, partie opérative, opérateur complexe, etc.) ou un système complexe.

Dans les descriptions de haut niveau, les informations de plus bas niveau telles que celles concernant le mode de synchronisation ne sont pas forcément spécifiées. L'insertion de ces éléments par la synthèse de haut niveau n'est cependant pas à omettre; toutes facilités doivent leur être offertes.

Partant d'une description comportementale donnée en VHDL et d'une bibliothèque externe d'unités fonctionnelles, AMICAL génère une architecture abstraite composée de deux sous-systèmes: une partie opérative et une partie contrôle. La partie opérative est spécifiée comme un ensemble de composants interconnectés, la partie contrôle quant à elle est définie comme une machine d'états finis. Le système de synchronisation à ce niveau est une horloge virtuelle gérant l'ordre d'exécution des événements entre les deux parties.

Le but de ce chapitre et des chapitres suivants est de générer une architecture détaillée à partir de l'architecture abstraite générée par AMICAL, et ce, en personnalisant cette dernière en y ajoutant des signaux "globaux" tels que les signaux d'horloge et de remise à zéro, et/ou des cellules additionnelles telles qu'un bloc de synchronisation, etc. Cette personnalisation est assurée par un outil programmable appelé PAT (Programmable Architecture Translator) qui sera présenté en détail dans le chapitre 4.

Ce chapitre est organisé sous les aspects suivants:

- La deuxième section présentera les différents modèles architecturaux générés

par AMICAL. Le modèle général de la sortie d'AMICAL sera détaillé ainsi que ses dérivées produites pour le *pipeline* et le test.

- La section 3 présentera la génération de l'architecture détaillée. L'ajout des signaux de synchronisation et des blocs de synchronisation à l'architecture abstraite sera introduit brièvement. De plus, deux différents types de modèles de synchronisation seront détaillés dans cette section, et ce, selon le type de *"pipeline"* concerné.

- La section 4 détaillera les bibliothèques de composants utilisées par AMICAL. Le passage d'une description comportementale à une description structurelle (complète) synthétisable et simulable au niveau transfert de registres par la synthèse architecturale offerte par AMICAL a recours à trois bibliothèques de composants. Ces bibliothèques sont équivalentes; ce qui les différencie c'est le niveau de description de leurs unités fonctionnelles. Ces niveaux correspondent à: l'entrée d'AMICAL, l'entrée de PAT et l'entrée des outils de conception au niveau transfert de registres (RTL) utilisant VHDL.

- La section 5 conclura ce chapitre par quelques commentaires et en donnant quelques unes des perspectives offertes par ces modèles architecturaux.

3.2 Architecture abstraite

3.2.1 Modèle général

L'architecture cible d'AMICAL est composée d'une partie opérative et d'une partie contrôle. La partie opérative correspond à un ensemble de composants interconnectés par tout un réseau de bus et d'interrupteurs. La partie contrôle est vue comme une machine d'états finis au niveau signal (c'est-à-dire, cette spécification n'inclut aucune opération sur les données).

Après la compilation de la spécification d'entrée, AMICAL réalise l'ordonnancement et l'allocation, et génère une architecture abstraite de base composée de deux sous-systèmes une partie contrôleur et une partie opérative (figure 3.1).

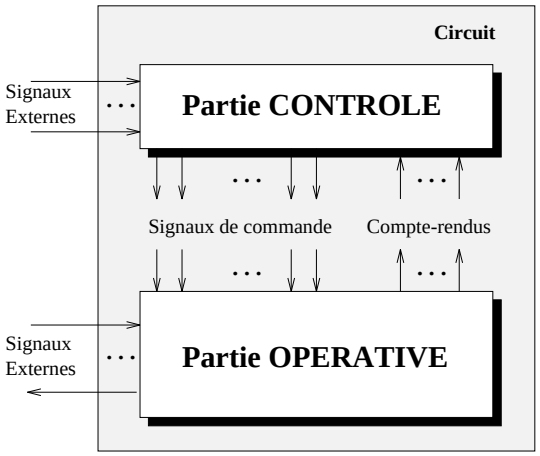


Figure 3.1: *Architecture Abstraite.*

La partie opérative correspond à un chemin de données, fait d'unités fonctionnelles allouées (pouvant exécuter les différentes opérations de la description) et connectées à des éléments de mémorisation (registres) par un réseau d'interconnexions. Le contrôleur est décrit sous la forme d'un graphe de Mealy. Les actions sont ici des activations de signaux qui commandent le chemin de données. Le vecteur de commande est généré en bloc; il n'existe donc pas d'ordre d'exécution des événements d'une même transition, que ce soit du côté partie opérative ou côté partie contrôle.

Les signaux de commande activent les unités fonctionnelles et autres composants

de la partie opérative, permettant ainsi les transferts élémentaires entre les différents éléments (FU, registres, etc.). La spécification de ces signaux (désignation) sera fournie au générateur de la partie contrôle. Parce qu'AMICAL considère la partie opérative comme un bloc qui pourra être réutilisé pour former d'autres assemblages, la désignation des signaux d'interface entre partie opérative et partie contrôle se fait par la partie opérative en fonction de ses éléments. Les interfaces externes de la partie opérative correspondent donc aux connecteurs de ce bloc. Ces derniers sont les bus permettant d'échanger les données avec l'extérieur, les compte-rendus (registres utilisés par la partie contrôle) et les signaux de commande venant de la partie contrôle.
La partie contrôle est donnée par AMICAL comme une table de transitions spécifiant l'état courant, les conditions à évaluer dans cet état, l'état suivant et les commandes à envoyer à la partie opérative pour l'exécution des opérations correspondantes pendant cette transition. Les interfaces externes correspondent aux connecteurs donnant les signaux de contrôle externes, les compte-rendus venant de la partie opérative et les signaux de commande envoyés vers cette dernière.

La figure 3.2 montre une partie opérative générée par AMICAL pour l'exemple du pgcd.

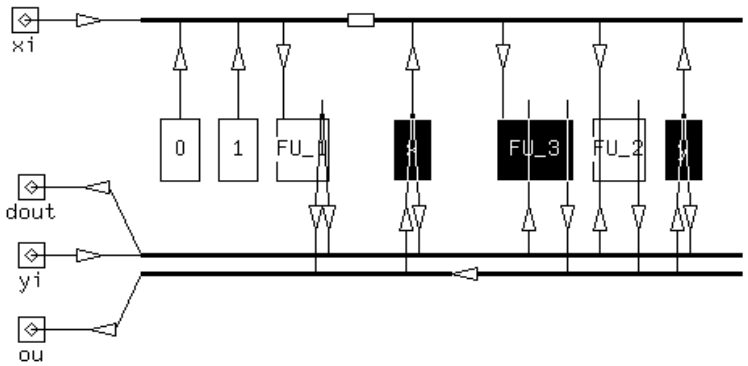


Figure 3.2: *Représentation AMICAL de la partie opérative générée pour l'exemple du pgcd.*

Elle est constituée d'éléments prédéfinis (registres et connecteurs) et d'unités fonctionnelles décrites par le concepteur (*FU_1*, *FU_2* et *FU_3*). Cette architecture est à base de bus. Elle comprend trois bus dont deux (premier bus: *Bus1* et dernier

bus: *Bus3*) sont séparés en deux segments de bus par des connecteurs horizontaux. Quant à la partie contrôle de l'exemple du pgcd, dont la figure 3.3 montre une partie de la table de transitions, elle est constituée de douze transitions et six états.

```
8    (State S5) (NextState S6) (Condition (& (/= x y) (< x y)) )
     (y)<=-(y, x)

9    (State S5) (NextState S6) (Condition (& (/= x y) (>= x y)) )
     (x)<=-(x, y)

10   (State S5) (NextState S2) (Condition (& (= x y) (/= start 1)) )
     (dout)<=out(1) / (ou)<=out(x)

11   (State S5) (NextState S3) (Condition (& (= x y) (= start 1)) )
     (dout)<=out(1) / (ou)<=out(x)

12   (State S6) (NextState S5) (Condition (TRUE) )
```

Figure 3.3: *Extrait de la table de transitions AMICAL générée pour l'exemple du pgcd.*

Si on prend par exemple la transition 9, elle spécifie la transition de l'état *S5* vers l'état *S6*, si les conditions

$$x \neq y$$

et

$$x \geq y$$

sont vérifiées. Cette transition est accompagnée de l'opération suivante:

$$x <= x - y.$$

Celle-ci est exécutée par la partie opérative; le temps d'exécution de cette instruction, d'au moins un cycle de base, est défini selon la description de l'unité fonctionnelle allouée pour cette opération.

3.2.2 Modèle avec interface entre la partie opérative et la partie contrôle

Le deuxième type d'architecture généré par AMICAL est une structure munie de deux interfaces faites de cellules de mémorisation, en plus des deux sous-systèmes de base (la partie opérative et la partie contrôle). Le schéma représentatif d'une telle architecture est donné par la figure 3.4.

La stratégie de conception adoptée consiste à intercaler des cellules d'interface entre la partie contrôle et la partie opérative assurant ainsi la mémorisation des

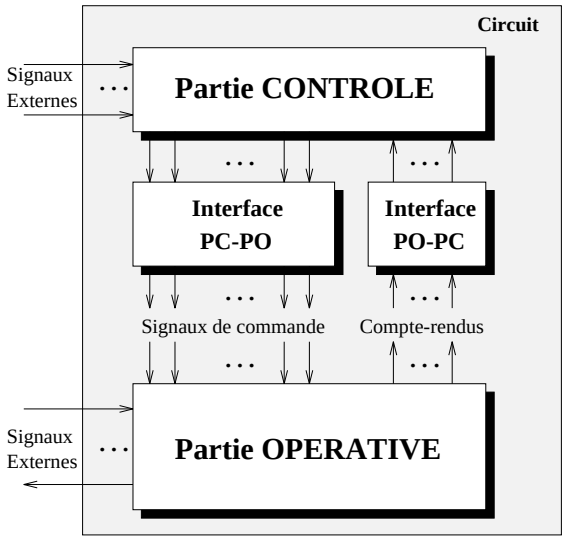


Figure 3.4: *Schéma de l'architecture avec interface générée par AMICAL.*

signaux de contrôle et des signaux de compte-rendus. Ces mêmes cellules peuvent aussi être des "*scan-cells*" qui permettraient donc de faire du test de type "*scan-path*".

Quand la partie contrôle envoie les signaux de commande à la partie opérative, ces signaux ne sont visibles par cette dernière que lorsqu'ils ont été stockés (*latchés*) par l'interface *PC-PO*. De même, les compte-rendus retournés par la partie opérative ne vont pas directement au contrôleur, ils sont mémorisés à l'interface *PO-PC*, et ne deviennent visibles par la partie contrôle qu'alors.

Ces interfaces sont à base de cellules de mémorisation. Ces cellules peuvent être définies de plusieurs façons. Le modèle adopté pour AMICAL, pour les fins de test de type "*scan-path*" et de la synchronisation, utilise des D *Flip-Flops*. Pour le cas du test, certaines cellules peuvent être remplacées automatiquement, par la suite, par leur version "*scannable*" par les outils de synthèse logique tels que Synopsys [51].

La figure 3.5 montre la représentation d'une telle interface.

Tous les signaux communs à toutes les cellules tels que les signaux d'horloge

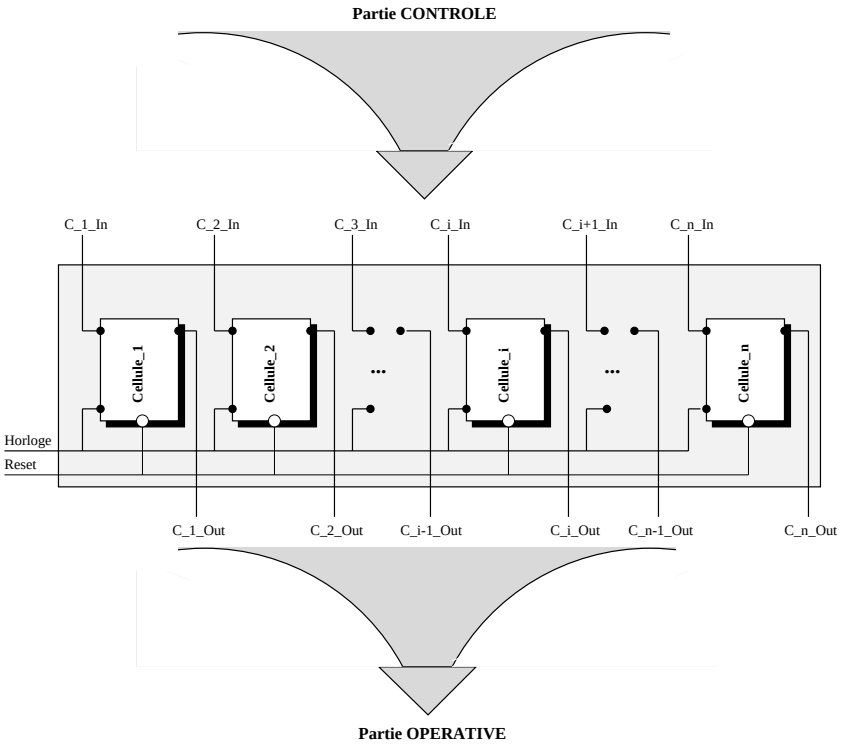


Figure 3.5: *Détails de l'interface séparant la partie contrôle et la partie opérative.*

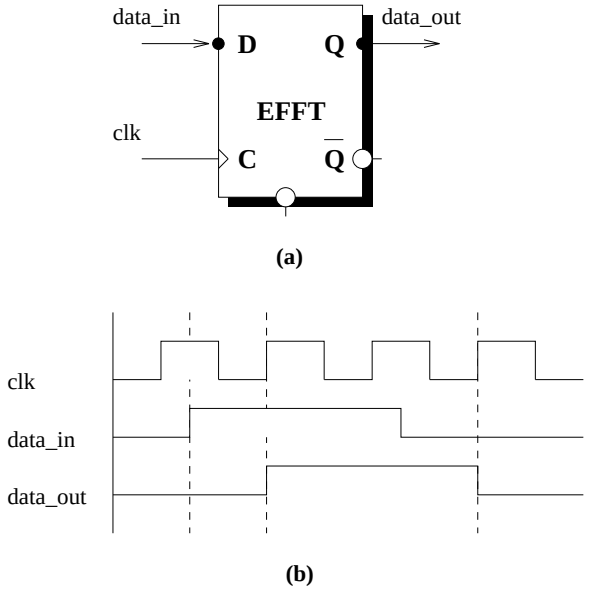
et les signaux de remise à zéro sont rajoutés par un outil programmable qui fait la traduction de la sortie RTL en VHDL, PAT, (voir Chapitre 4). Ceci vient à l'encontre d'une conception synchrone de l'interface, ainsi qu'à la testabilité du circuit complet.

Cette architecture a plusieurs avantages, dont voici les plus importants:

- Elle permet l'utilisation de nouveaux modèles de synchronisation (modèles *pipelinés*).
- Elle permet l'incorporation de modèles de test (modèles *Scan-Path*).
- Elle permet une structuration plus nette de la partie opérative.
- Elle réduit, du point de vue de la testabilité, le degré de séquentialité des composants permettant ainsi un test simplifié et donc plus rapide [51].

Comme mentionné plus haut, chaque cellule est un *flip-flop* fonctionnant sur front montant. Elle ne mémorise la donnée présente sur son entrée (*data_in*) qu'au

prochain front montant de l'horloge; cette même valeur est présente en sortie à ce même instant. Le fonctionnement d'une D *flip-flop*, ainsi que sa description VHDL sont donnés dans la figure 3.6.




```
Library mvl_7;
Use mvl_7.Types.all;
Use mvl_7.Arithmetics.all;
Entity scan_cell is
   Port( Data_In     : in  mvl7;
         Data_Out    : out mvl7;
         clk         : in mvl7);
end scan_cell;
Architecture Implementation of scan_cell is
Begin
    scan_cell_file : process
    Begin
        wait until clk = '1' and clk'event;
        Data_out <= Data_In;
    end process scan_cell_file;
end Implementation;
```

(c)

Figure 3.6: *Description de la cellule de base de l'interface.*
(a) *Représentation schématique*, (b) *Chronogramme*, (c) *Description VHDL.*

3.3 Architecture Détaillée

Le but de cette section est de montrer comment générer une architecture détaillée à partir d'un modèle abstrait généré par AMICAL. Cette génération est produite par la personnalisation de l'architecture générée en y ajoutant des signaux "globaux" (signaux d'horloge et de remise à zéro, etc.) et/ou des cellules supplémentaires telles que des blocs de synchronisation ou autres éléments (pour des fins du test, par exemple).

Chaque architecture abstraite peut être transposée sur plusieurs modèles architecturaux. Cette transposition est assurée par un outil programmable appelé PAT utilisant un fichier global dans lequel le concepteur peut spécifier tous les signaux et toutes les cellules qu'il veut ajouter [3, 6, 56].
L'architecture finale, sera appelée architecture détaillée. La figure 3.7 illustre le chemin de donnée de la transposition de l'architecture abstraite en plusieurs architectures détaillées.

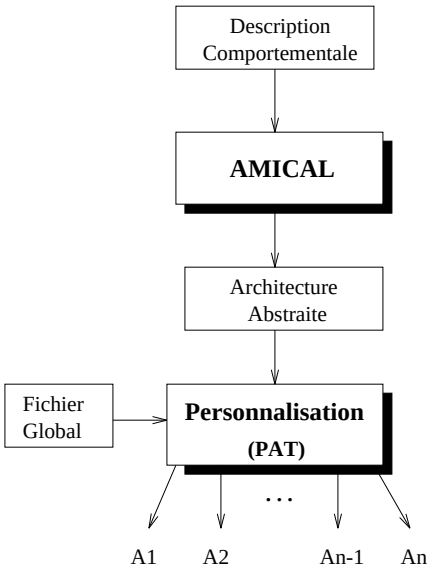


Figure 3.7: *Personnalisation de l'architecture abstraite.*

L'architecture détaillée ainsi trouvée peut dépendre:

- du modèle d'horloge généré pour cette architecture (sur front ou sur niveau),

- du modèle de communication entre la partie opérative et la partie contrôle (avec ou sans "*pipeline*"),

- de l'organisation du circuit (avec ou sans interface entre la partie opérative et la partie contrôle).

Bien entendu, pour chaque architecture finale remplissant les critères cités plus haut, une bibliothèque de composants est fournie où tous les éléments formant la partie opérative doivent être cohérents entre eux et avec le modèle d'horloge.

Le schéma de la figure 3.8 donne une vue globale d'un exemple de modèle d'architecture détaillée. Dans ce cas, le modèle de synchronisation a nécessité l'ajout d'un bloc et des signaux de synchronisation à l'architecture abstraite donnée par la figure 3.1.

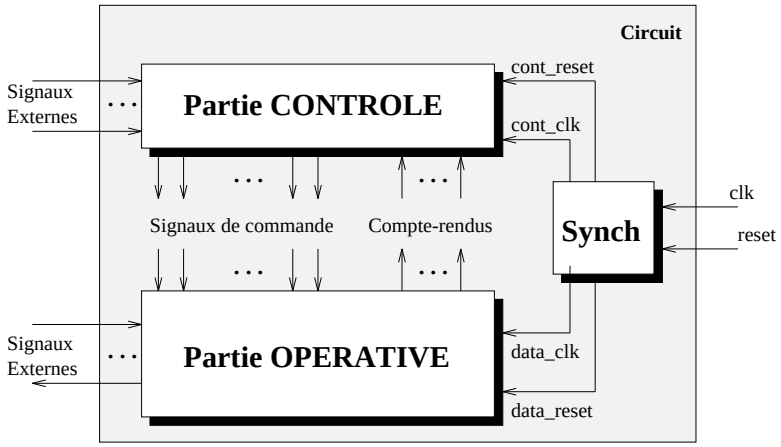


Figure 3.8: *Vue globale de l'architecture détaillée.*

Ce bloc de synchronisation est un circuit qui est séparé du reste du modèle de l'architecture abstraite. Il doit assurer la synchronisation entre la partie opérative et la partie contrôle et donner l'ordre d'exécution des événements. Ce bloc de synchronisation reçoit des signaux de synchronisation externes et les transforme en des signaux internes gérant la synchronisation de tous les éléments synchrones de la circuiterie.

Le bloc de synchronisation (*Synch*) reçoit comme entrées un signal d'horloge *clk*

et un signal d'initialisation *reset* et génère en sortie quatre signaux, dont *cont_clk* et *cont_reset* pour la partie contrôle et *data_clk* et *data_reset* pour la partie opérative.

3.3.1 Ajout des éléments de synchronisation

Avant de détailler les différents modèles de synchronisation, voici un aperçu sur le processus d'ajout des signaux globaux et les blocs de synchronisation à l'architecture de base.

PAT (Programmable Architecture Translator) est un outil aidant à interfacer la sortie abstraite d'AMICAL avec les outils de simulation et de synthèse logique existants en insérant dans l'architecture initiale des éléments de synchronisation et en traduisant les fichiers de sortie, donnée en SOLAR [46], en leurs équivalents en VHDL [41].

Cette section ne présentera pas l'outil programmable PAT, mais donnera seulement un aperçu du mode d'utilisation de PAT pour l'ajout des signaux globaux à l'architecture abstraite générée par AMICAL. Le Chapitre 4 donnera plus de détails concernant PAT et la génération des fichiers VHDL à partir des fichiers SOLAR.

Afin de générer la description VHDL utilisant les différents signaux de synchronisation, l'interface PAT permet au concepteur de spécifier explicitement ces signaux dans un fichier "*global*" dont un exemple de format correspondant à l'architecture montrée dans la figure 3.8 est donné par la figure 3.9.

Pendant la traduction des fichiers SOLAR en fichiers VHDL des description RTL, ces signaux sont ajoutés aux endroits appropriés. Le fichier "*global*", contenant tous les signaux globaux que le concepteur veut connecter au circuit, est décrit de telle sorte que pour chaque bloc, il peut exister un ensemble de signaux et un ensemble de cellules.

Plus de détails concernant ce fichier "*global*" et les principes de PAT seront donnés dans le Chapitre 4.

```
1  (Global nom_du_circuit.global
2      (Block nom_du_circuit_datapath
3          (Signal data_clk    (Dir IN)          (Attr CLOCK)    (Type BIT)      (Local clk))
4          (Signal data_reset  (Dir IN)          (Attr RESET)    (Type BIT)      (Local reset))
5          ... (declaration d'autres signaux ou cellules)
6      )
7      (Block nom_du_circuit_control
8          (Signal cont_clk    (Dir IN)          (Attr CLOCK)    (Type BIT)      (Local clk))
9          (Signal cont_reset  (Dir IN)          (Attr RESET)    (Type BIT)      (Local reset))
10         ... (declaration d'autres signaux ou cellules)
11     )
12     (Block nom_du_circuit_circuit
13         (Signal clk         (Dir IN)          (Attr CLOCK)    (Type BIT))
14         (Signal reset       (Dir IN)          (Attr RESET)    (Type BIT)
15         (Signal cont_clk    (Dir INTERNAL)    (Attr CLOCK)    (Type BIT))
16         (Signal cont_reset  (Dir INTERNAL)    (Attr RESET)    (Type BIT))
17         (Signal data_clk    (Dir INTERNAL)    (Attr CLOCK)    (Type BIT))
18         (Signal data_reset  (Dir INTERNAL)    (Attr RESET)    (Type BIT))
19         ... (declaration d'autres signaux ou cellules)
20         (Glue Clock)
21     )
22 )
```

Figure 3.9: *Fichier "global" pour l'exemple du pgcd.*

Dans la suite, on supposera que l'architecture détaillée comportera un bloc de synchronisation qui a pour fonction de distribuer les signaux d'horloge et de remise à zéro pour les différents blocs du circuit.

3.4 Modèles de Synchronisation

Un modèle de synchronisation, pour un circuit composé d'une partie opérative et d'une partie contrôle, définit le modèle de communication entre la partie opérative et la partie contrôle pendant un cycle de base [80]. Ceci correspond à la synchronisation des différentes actions à exécuter pendant chaque étape de contrôle.

3.4.1 Définition d'un cycle de base

Pour le modèle de l'architecture de la figure 3.10, où une machine de Mealy est utilisée, le cycle de base correspond à l'exécution d'une micro-instruction. Il est défini par la succession des étapes suivantes:

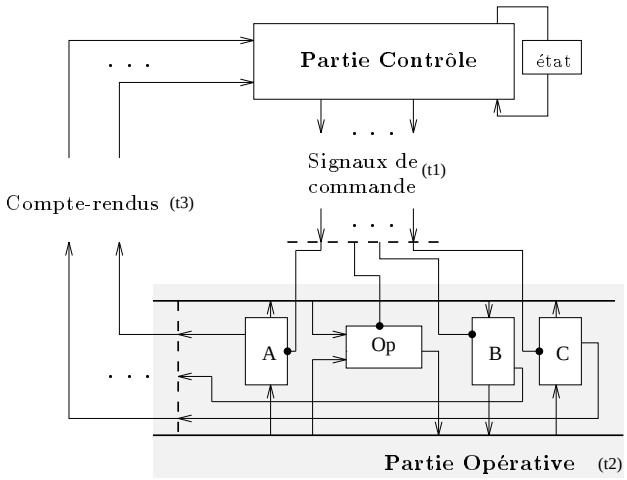


Figure 3.10: *Architecture générale.*

a) Le contrôleur a un état courant; il calcule les conditions présentes et ainsi définit son état suivant de même que la transition à effectuer.

b) Le contrôleur calcule les signaux de commande correspondant aux actions à réaliser pour la transition trouvée et les envoie à la partie opérative.

c) La partie opérative exécute les opérations correspondant aux signaux de contrôle envoyés par le contrôleur.

d) La partie opérative retourne les compte-rendus à la partie contrôle.

Le chemin critique pour l'exécution d'un cycle de base peut être noté [80] comme suit:

- t1, pour le calcul de l'état suivant et des signaux de commandes pour la partie opérative.
- t2, pour le calcul des opérations dans la partie opérative.
- t3, pour les compte-rendus qui sortent de la partie opérative à la partie contrôle.

Un cycle de base correspond, alors, à la concaténation des trois délais:

$$T = t1 + t2 + t3$$

où T représente un cycle de base qui est égal à une période d'horloge.

En pratique, t3 est généralement négligeable, donc le cycle de base sera égal à:

$$T = t1 + t2$$

3.4.2 Définition d'un transfert

Le temps d'exécution des opérations peut être de plusieurs cycles, par exemple: *deux cycles* auquel cas l'opérateur est alimenté par les valeurs d'entrée pendant le premier cycle et rend le résultat pendant le deuxième cycle. Quand une opération prend *un cycle*, les opérandes et les résultats sont transférés pendant le même cycle.

Une micro-instruction (μ-*instruction*) est composée d'un ensemble de transferts devant s'exécuter en parallèle. On distingue trois types de transferts:

$$Reg -> Bus1 -> in(Op)$$

$$out(Op) -> Bus2 -> Reg$$

$$Reg- > Bus1- > Op- > Bus2- > Reg$$

où

. *Reg* correspond à un registre qui peut être un registre source, un registre destination ou les deux en même temps.

. *Op* correspond à une unité fonctionnelle, qui pourrait exécuter n'importe quelle opération unaire, binaire ou complexe; dans la figure 3.11 Op est une unité fonctionnelle exécutant une opération binaire.

. *Bus1* et *Bus2* correspondent aux bus internes de la partie opérative et servent aux transferts des données entre registres et unités fonctionnelles.

Le chemin de données correspondant à une opération binaire est illustré sur la figure 3.11. L'opération peut être exécutée dans un seul cycle de base en trois transferts. Les deux valeurs d'entrée de *Op* sont obtenues par le transfert des valeurs des deux registres. Le résultat de l'exécution de *Op* est sauvegardé dans un registre (Reg1).

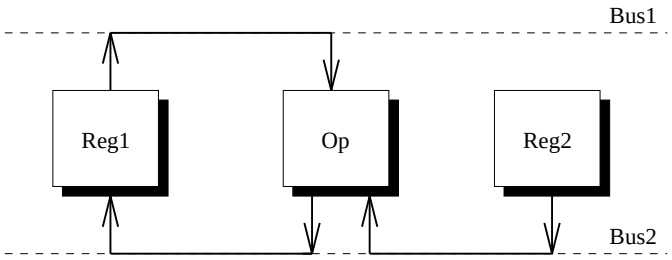


Figure 3.11: *Chemin de données d'une micro-instruction.*

Dans la suite de ce chapitre, seules les unités fonctionnelles travaillant sur un cycle, seront considérées. L'hypothèse faite est donc que chacune des opérations réalisées par les unités fonctionnelles s'exécute en un cycle de base.

3.4.3 Notion du "Pipeline" entre la partie opérative et la partie contrôle

Pour plus de flexibilité, on définit différents modèles de synchronisation qui travailleront dans deux modes différents :

* Le mode non-recouvrant : La partie opérative et la partie contrôle fonctionnent séparément; elles fonctionnent en séquentiel et à aucun moment les deux parties ne s'exécutent simultanément.

* Le mode recouvrant : La partie opérative et la partie contrôle fonctionnent en même temps (en mode *pipeline*), c'est-à-dire, pendant que le contrôleur calcule et génère le vecteur de commande pour le prochain cycle d'exécution de la partie opérative, cette dernière exécute les opérations correspondant au vecteur de commande calculé le cycle précédent. Ceci permet de réaliser du parallélisme et donc d'avoir les deux parties actives au maximum.

Pour ce qui suit, la notation *pipe_0* sera adoptée pour le mode non-recouvrant et la notation *pipe_i* pour le mode recouvrant (i prendra les valeurs 1, pour une architecture à contrôleur à un seul étage, et 2 pour une architecture à contrôleur à deux étages).

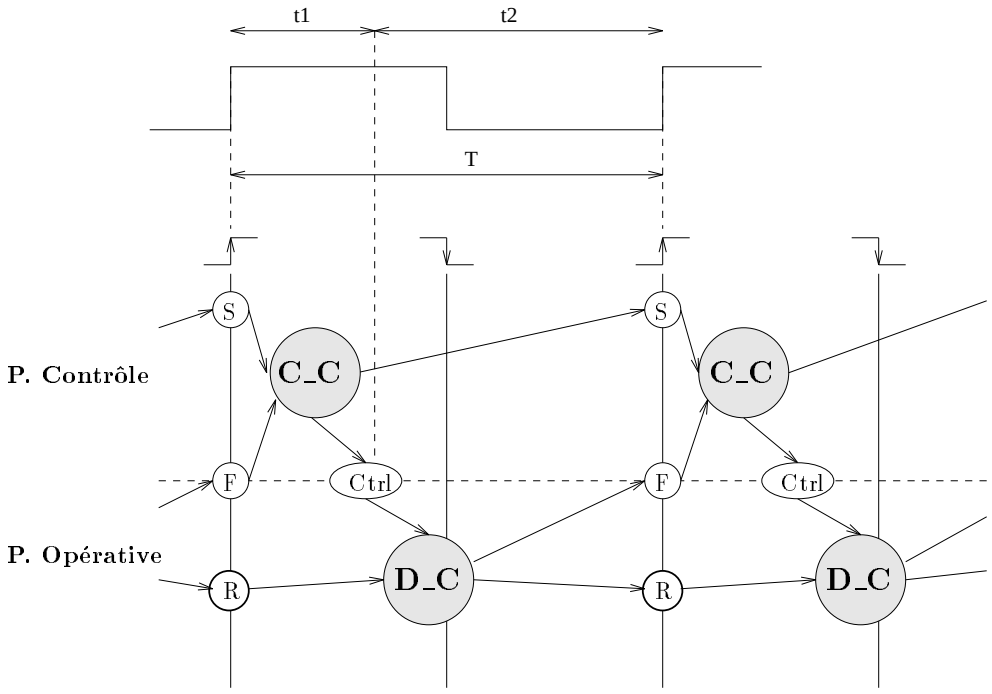
3.4.4 Modèle temporel de base, exécution sans recouvrement: (*Pipe_0*)

C'est un modèle simple pour lequel les deux parties constituant le circuit travaillent en mode non-recouvrant (c'est à dire, la partie opérative et la partie contrôle fonctionnent séquentiellement) afin de permettre à la partie contrôle de lire ses données entrantes (commandes externes et compte-rendus), de calculer les vecteurs de commande correspondant à la transition à effectuer et de les envoyer à la partie opérative qui alors exécute les opérations appropriées. Il n'existe aucune concurrence entre les deux parties.

Plusieurs modèles d'horloges peuvent être utilisés pour une telle exécution. Deux de ces modèles sont détaillés:

A- Le premier consiste à ce que tout soit exécuté en *une seule période d'horloge*. En d'autres termes, la partie opérative et la partie contrôle travaillent sans recouvrement pendant le même cycle d'horloge de telle sorte qu'à chaque front montant de l'horloge, la partie contrôle calcule la condition présente et suivant

l'état courant, trouve l'état suivant et calcule aussi les signaux de commande, correspondant à la transition, qui doivent être envoyés à la partie opérative. Ceci prend un temps variable suivant la transition. Le temps critique (*t1*) pour la partie contrôle peut être déterminé après la synthèse logique (exécutée dans notre cas par Synopsys [82]). Juste après la réception des signaux de commande par la partie opérative, celle-ci exécute les opérations appropriées. Le temps critique pour l'exécution du chemin de données (*t2*) est une fonction du délai maximum que peut prendre une unité fonctionnelle pour l'exécution d'une opération. La partie opérative doit rendre des compte-rendus à la partie contrôle, pour être pris en compte pour le lancement de l'exécution d'une autre procédure de contrôle, pendant un temps *t3* estimé nul (figure 3.12).



$T \leq t1 + t2$

Tmax = t1 +t2 = Période d'horloge de base.

C_C: Logique du contrôleur
D_C: Logique de la partie opérative
S: Etat (Courant | Suivant)
F: Compte-rendu
R: Valeur du registre
Ctrl: Vecteur de commande

Figure 3.12: *Partage d'un cycle de base.*

Pendant qu'une partie (contrôle ou opérative respectivement) est active, l'autre

partie (opérative ou contrôle respectivement) doit-être dans un état de repos en attendant qu'elle soit active à son tour, comme illustré par la figure 3.13.

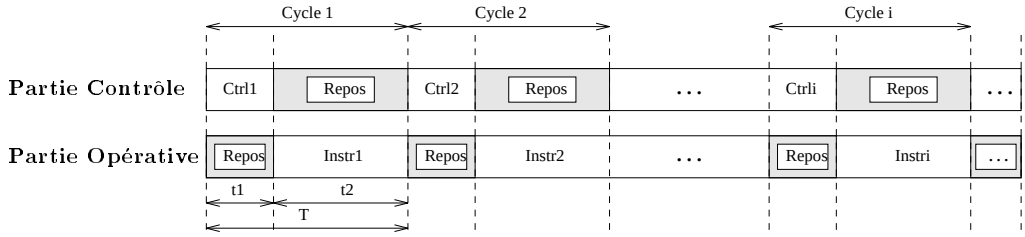


Figure 3.13: *Modèle sans "pipeline" s'exécutant sur un cycle d'horloge.*

Le grand inconvénient de ce genre de modèle est que le cycle de base (T) doit être très grand pour permettre l'exécution complète d'une étape de contrôle.

B- La seconde solution est que chaque partie doit travailler pendant *un cycle d'horloge*. Cela veut dire qu'on doit introduire un autre signal d'horloge (*phi*) pour spécifier laquelle des deux parties doit travailler en mettant l'autre au repos. La figure 3.14 explique ce schéma par des chronogrammes.

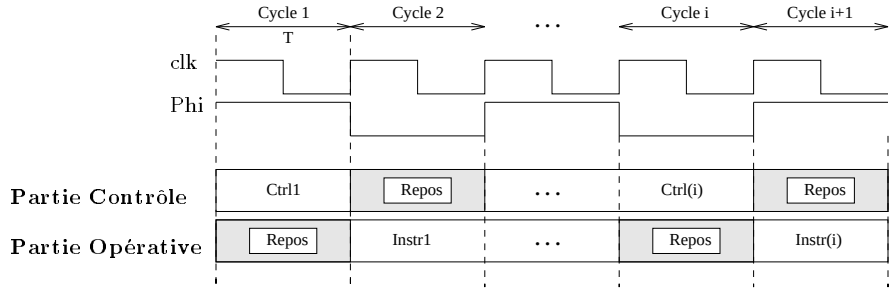


Figure 3.14: *Modèle sans "pipeline" s'exécutant sur deux cycles d'horloge.*

Pendant le *cycle 1*, la partie contrôle calcule les signaux de commande pour la partie opérative qui ne les prendra en compte qu'au prochain cycle (*cycle 2*). Quand *phi* est égal à '0', la partie contrôle est au repos et il n'y a que la partie opérative qui travaille. Quand *phi* est égal à '1', la partie contrôle travaille et la partie opérative est à son tour au repos.

Avec la première solution la période d'horloge est de:

$$T = t1 + t2,$$

alors qu'avec la deuxième solution, la période d'horloge est de:

$$T = max(t1, t2).$$

Si le nombre de cycles pour exécuter l'algorithme avec la solution A est de *Nbcycles*, au moins le double est requis pour la solution B.

Puisque

$$max(t1, t2) \geq \frac{t1 + t2}{2}$$

le temps d'exécution de l'algorithme avec la solution A ((t1 + t2) * Nbcycles) est inférieur au temps nécessaire pour la solution B.

3.4.5 Modèle temporel accéléré

Pour augmenter la vitesse d'exécution des circuits, la partie opérative et la partie contrôle sont activées en même temps. Le but est de "*pipeliner*" leur exécution de façon à ce que le temps pendant lequel elles sont activées soit maximal.

Si l'automate de contrôle n'attendait pas les résultats de l'exécution des actions opératives, on pourrait facilement maintenir la partie opérative constamment occupée. Ce n'est hélas, pas toujours le cas.

La solution pour superposer l'exécution des algorithmes des deux parties est d'utiliser le mode avec recouvrement (figure 3.15).

Pour ce modèle, on doit s'assurer que pendant que la partie opérative exécute les opérations correspondant aux signaux de contrôle générés par le contrôleur pendant le cycle i, la partie contrôle est en train de calculer les signaux de commande correspondant au cycle suivant $(i+1)$.

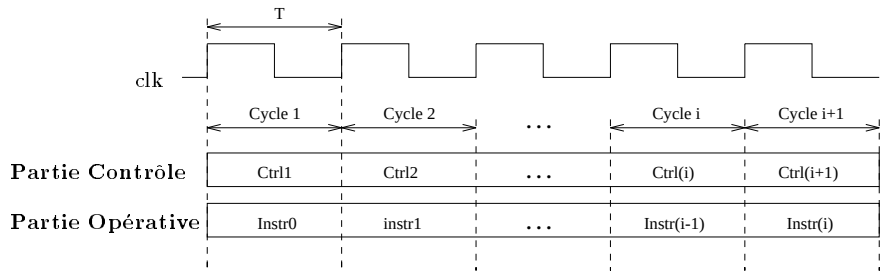


Figure 3.15: *Modèle avec recouvrement.*

Dans ce cas, la période d'un cycle est égale à:

$$T = max(max(t1), max(t2))$$

Dans ce type de modèle, une étape de contrôle s'exécute sur deux périodes d'horloge (contrairement au modèle sans "*pipeline*" qui s'exécute sur une seule période d'horloge). A chaque front montant de l'horloge, le contrôleur, en fonction de l'état courant (S) et des compte-rendus (F) retournés par la partie opérative, évalue les conditions et calcule le vecteur de commandes ($Ctrl$) qu'il ne doit envoyer à la partie opérative qu'au prochain front montant de l'horloge. En même temps la partie opérative, en fonction des valeurs présentes dans les registres (R), exécute les opérations correspondant au vecteur de commandes ($Ctrl$) calculé par la partie contrôle dans le cycle précédent (figure 3.16).

Dans ce genre de modèle, il existe deux restrictions dont on doit tenir compte:

1- Cette structure ne peut fonctionner que si les commandes envoyées de la partie contrôle à la partie opérative et les compte-rendus retournés par celle-ci sont *latchés* (mémorisés pendant un certain temps) afin de bien synchroniser les événements entre les deux parties. Deux solutions sont envisageables:

a- Soit, chaque commande est mémorisée à l'intérieur du composant approprié jusqu'au nouveau front montant de l'horloge, et ce, en créant des variables internes qui stockent les signaux de contrôle, fournis par la partie contrôle à n'importe quel instant, jusqu'au prochain front montant.

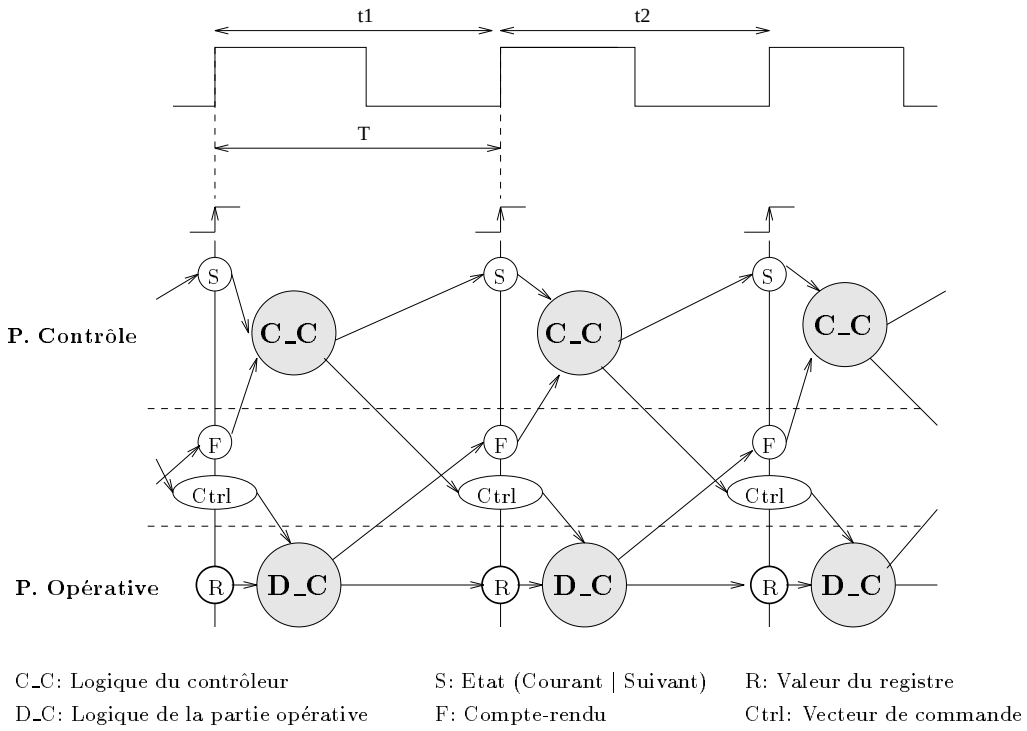


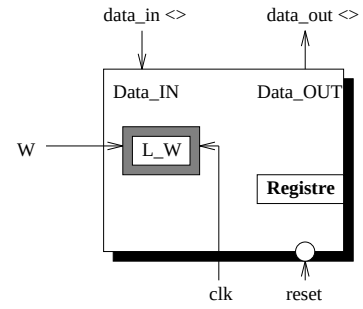
Figure 3.16: *Modèle pipeliné.*

Un exemple d'une telle représentation est donné dans la figure 3.17 pour un registre (*VariableRegister*).

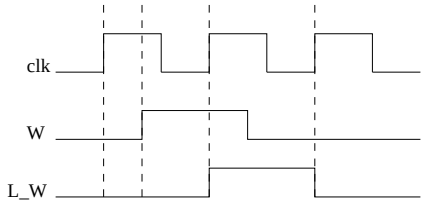
b- Soit, chaque signal de commande (resp. signal de compte-rendu) est mémorisé dans une cellule de mémorisation (*flip-flop*) située dans un bloc d'interface jusqu'au prochain front montant de l'horloge pour être envoyé à la partie opérative (resp. la partie contrôle) (voir section 3.2.2).

2- Tenir compte des cas où le calcul des signaux de commande (*Ctrli*) dans le cycle *i* dépend des résultats produits par la partie opérative (*Instr(i-1)*) dans le même cycle (cycle *i*).

Il faut que la condition de transition ne dépende pas de l'état courant mais d'un état précédent. Dans certains cas, l'insertion d'un état intermédiaire (d'un cycle en général) est nécessaire. Cet état est un état de repos (pour le contrôleur) permettant d'attendre les compte-rendus adéquats de la partie opérative pour l'évaluation des prochaines conditions à tester (figure 3.18).



(a)



(b)

```
Library mvl_7;
Use mvl_7.Types.all;
Use mvl_7.Arithmetics.all;

Entity VariableRegister  is
    Port(  Data_In     : in  BusX(0 to 7) := (others => 'Z');
           Data_Out    : out BusX(0 to 7) := (others => 'Z');
           W           : in  mvl7;
           clk         : in  mvl7;
           reset       : in  mvl7  );
end VariableRegister;

Architecture Behavior of VariableRegister is
Signal  Val_Reg : mvl7_vector(0 to 7);
Signal  L_W  : mvl7;
Begin
    Reg_file1 : process
    Begin
        wait until  clk = '1' and clk'event;
        if  reset = '0' then
            Val_Reg <= (others => '0');
        elsif L_W = '1' then
            Val_Reg <= Drive(Data_In);
        end if;
    end process  Reg_file1;
    Reg_file2 : process
    Begin
        wait until  clk = '1' and clk'event;
              L_W <= W;
    end process  Reg_file2;
    Data_Out <= Drive(Val_Reg);
end Behavior;
```

(c)

Figure 3.17: *Mémorisation des signaux de commande à l'intérieur des composants.* (a) *Représentation d'un registre*, (b) *Chronogramme*, (c) *Description VHDL.*

L'état intermédiaire est inséré par l'outil d'ordonnancement. Quand ce dernier détecte une dépendance de *pipeline*, il insère cet état de repos pour éviter tout conflit dans le processus de synchronisation. Dans la figure 3.18, on suppose que l'évaluation de la condition *Ci+1* dépend des données calculées dans l'action *Ai*; l'insertion du cycle vide (*Repos*) est donc nécessaire, dans ce cas, entre l'état *Si* et l'état *Si+1*, ainsi que le changement de la transition de *Si* qui va se dévier vers l'état *Repos*.

Pour les modèles pipelinés, deux cas sont présentés:

- modèle pipeliné avec un $pipe = 1$: cas d'une architecture à contrôleur à un

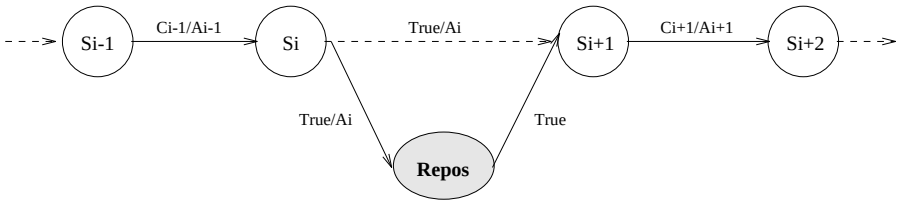


Figure 3.18: *Insertion d'un état intermédiaire.*

seul étage.

- modèle pipeliné avec un *pipe = 2* : cas d'une architecture à contrôleur à deux étages.

3.4.5.1 *Modèle pipeliné avec un pipe = 1*

Dans ce modèle, la partie contrôle est à un seul étage. Les signaux de commande calculés par le contrôleur sont directement envoyés à la partie opérative pour l'exécution des opérations appropriées. Inversement, la partie opérative rend directement les compte-rendus des opérations effectuées à la partie contrôle.

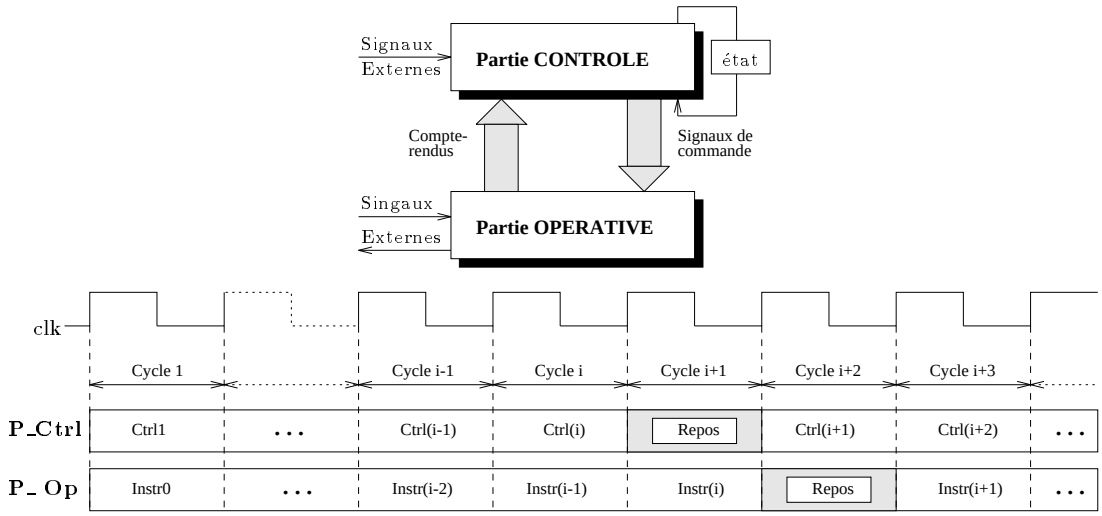


Figure 3.19: *Modèle de synchronisation avec un pipe = 1.*

La figure 3.19 illustre le partage d'une étape de contrôle en cycle d'horloge avec, bien sûr, insertion de l'état intermédiaire pour ce type d'architecture.

Les deux parties doivent être actives en même temps. Quand la partie contrôle calcule la micro-commande *i* (*Ctrl(i)*), la partie opérative doit exécuter la micro-instruction *i-1* (*Instr(i-1)*). Si les conditions à évaluer dans le contrôleur pour le cycle *i* dépendent des résultats des opérations exécutées dans la partie opérative au cycle *i-1*, alors l'insertion d'un état de repos d'*un cycle* est nécessaire pour permettre à la partie opérative de terminer l'exécution des opérations et à la partie contrôle de lire les compte-rendus retournés concernant ces opérations.

3.4.5.2 *Modèle pipeliné avec un pipe = 2*

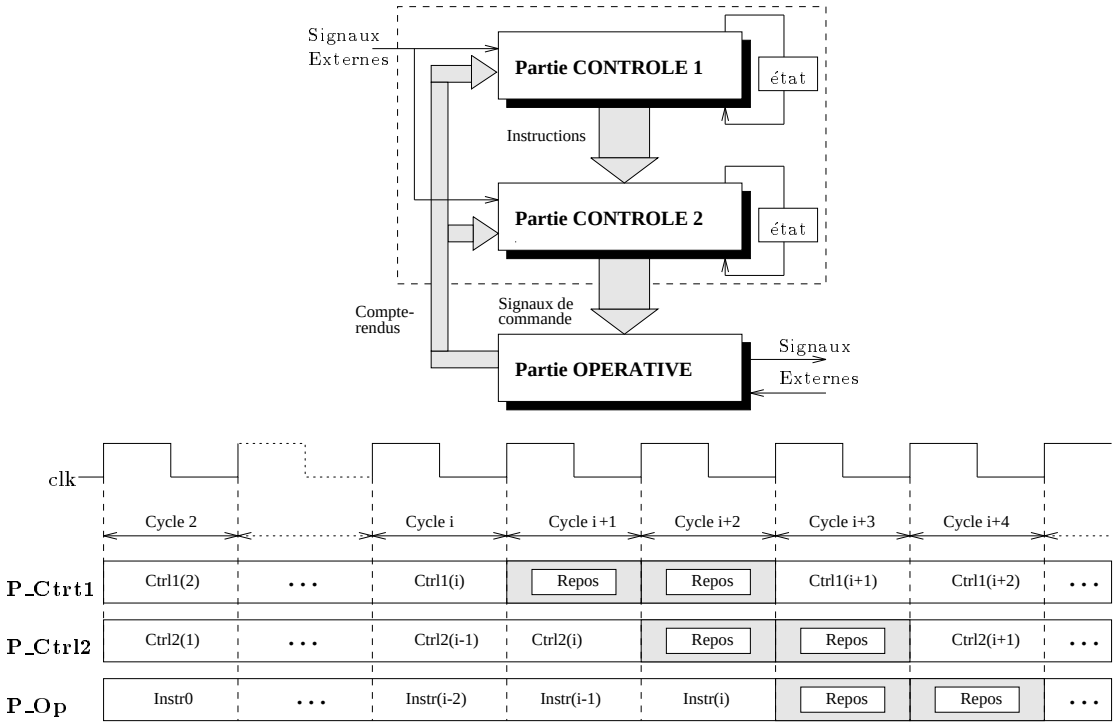


Figure 3.20: *Modèle de synchronisation avec un pipe = 2.*

Pour ce deuxième modèle, l'architecture contient un contrôleur à deux étages (tranches). La tranche inférieure exécute les instructions générées par la tranche supérieure en les découpant en sous-instructions. Ces dernières seront prises en compte par la partie opérative qui exécute les opérations appropriées et rend des compte-rendus aux deux tranches de la partie contrôle.

La figure 3.20 représente une telle architecture et donne l'insertion des cycles de repos en cas de nécessité.

De la même façon que le modèle précédent, si les conditions à évaluer dans la partie contrôle pour le cycle *i* dépendent des résultats des opérations exécutées dans la partie opérative pendant le cycle *i-2*, alors *deux cycles* de repos sont nécessaires, cette fois-ci, pour permettre à la tranche inférieure du contrôleur de recevoir les instructions de la tranche supérieure et à la partie opérative de recevoir les vecteurs de commande de la tranche inférieure et de rendre les compte-rendus aux deux étages du contrôleur.

Pour ce qui va suivre, seules les architectures à contrôleur à un seul étage seront considérées pour notre étude.

3.5 Bibliothèque des macro-blocs

Les macro-blocs sont spécifiés par deux types de composantes: les ports et la fonctionnalité (comportement). Le composant est vu comme une boîte noire avec des ports qui sont définis, soit en entrée (*IN*), soit en sortie (*OUT*), soit en entrée/sortie (*INOUT*). Par contre, la fonctionnalité définit le comportement du composant en fonction de ses entrées et sorties, et du temps.

Pour plus de flexibilité, la notion d'opération est introduite. Une ressource est définie par ses attributs physiques et sa liste d'opérations. La description physique contient la description des ports (ports d'entrée, de sortie ou d'entrée/sortie). La description d'opération contient toutes les informations nécessaires au développement de cette opération, y compris l'information temporelle. A l'intérieur de la description d'une opération, deux types de ports sont considérés: les ports de données et les ports de commandes. Les ports de commandes sont utilisés, par exemple, pour sélectionner une opération dans une UAL. Quant aux ports de données, ils sont utilisés pour permettre le transit des données.

Le but de cette section est de démontrer le modèle de composition et de synchronisation du circuit réalisé par AMICAL.

La bibliothèque d'AMICAL est décrite dans trois niveaux: le niveau d'entrée d'AMICAL, le niveau d'entrée de PAT et le niveau d'entrée des outils de conception au niveau tansfert de registres (figure 3.21).

Dans les deux premiers niveaux, la bibliothèque est décrite en SOLAR [46], tandis que dans le troisième niveau, elle est décrite en VHDL [41].

- *Premier niveau*: Dans ce niveau, le modèle abstrait des composants est présenté. Seules les entrées/sorties du composant, ainsi que les caractéristiques physiques (surface, etc.) déterminantes pour l'allocation des FUs pendant le processus de synthèse sont décrites. Ces mêmes informations sont utilisées

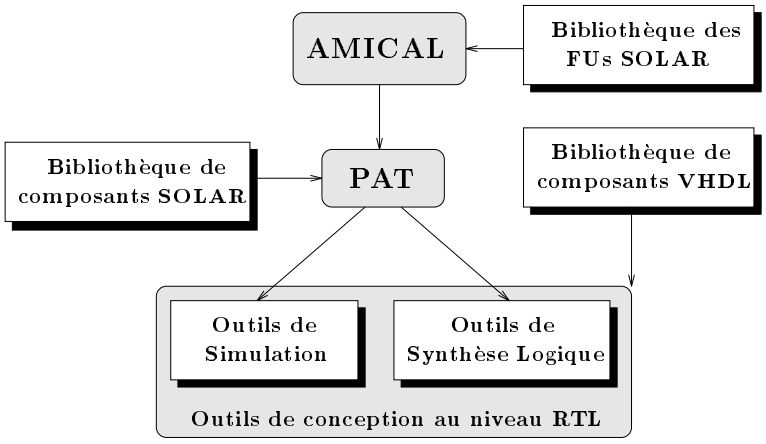


Figure 3.21: *Bibliothèque des macro-blocs.*

pour certaines évaluations pendant ou après la synthèse AMICAL. En exploitant celles-ci, le concepteur peut réaliser des interventions manuelles efficaces. Le modèle abstrait d'un exemple de registre est donné dans la figure 3.22.

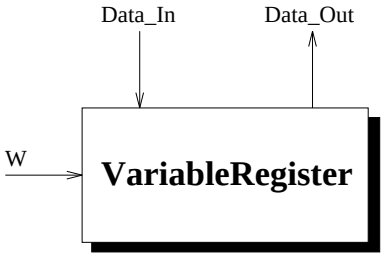


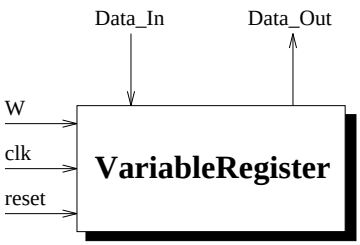
Figure 3.22: *Modèle abstrait d'un registre.*

Pour des unités fonctionnelles plus complexes, on peut se reporter à la section 2.2.3 et à la figure 2.6.

- *Deuxième niveau*: Dans ce niveau, ne sont décrites que les entrées/sorties des composants pour l'interfaçage de la partie opérative générée (*netlist*). Cette bibliothèque de composants est créée pour permettre la transition, par PAT (voir Chapitre 4), des composants abstraits en leurs équivalents VHDL. Un exemple de description d'un composant de cette bibliothèque sera détaillé dans la section 4.3.1. Les types d'objet (*mvl7*, *bit*, etc.) à utiliser suivant l'outil

de synthèse logique prévu peuvent être définis à ce niveau, permettant ainsi à l'utilisateur de pouvoir passer d'un outil à un autre pour la suite de sa compilation de silicium.

- *Troisième niveau*: Dans ce niveau, sont décrites les entrées/sorties des composants, ainsi que la fonctionnalité (comportement) des unités. Cette description est donnée en VHDL afin de permettre la simulation et la synthèse logique par les outils de conception au niveau transfert de registres utilisant VHDL. Un exemple de composant "*réel*" de cette bibliothèque, après la personnalisation et l'ajout des signaux de synchronisation, est montré dans la figure 3.23.



(a)

```
Library mvl_7;
Use mvl_7.Types.all;
Use mvl_7.Arithmetics.all;

Entity VariableRegister  is
    Port(  Data_In      : in  BusX(0 to 7) := (others => 'Z');
           Data_Out     : out BusX(0 to 7) := (others => 'Z');
           W            : in  mvl7;
           clk          : in  mvl7;
           reset        : in  mvl7  );
end VariableRegister  ;

Architecture Behavior of VariableRegister  is
Signal  Val_Reg : mvl7_vector(0 to 7);
Begin
    Reg_file1 : process
    Begin
        wait until  clk = '1' and clk'event;
        if  reset = '0' then
            Val_Reg <= (others => '0');
        elsif  W = '1' then
            Val_Reg <= Drive(Data_In);
        end if;
    end process  Reg_file1;
    Data_Out <= Drive(Val_Reg);
end Behavior;
```

(b)

Figure 3.23: *Exemple d'un modèle réel d'un registre.*
(a) *Représentation schématique*, (b) *Description VHDL.*

On peut remarquer que cette description inclut les éléments de synchronisation (clk, reset, etc.).

Dans la suite de cette section, un exemple de bibliothèque contenant quelques exemples de composants utilisés pour la plupart des circuit générés par AMICAL,

sera présenté. Cette bibliothèque est un ensemble de composants (registres, interrupteurs, etc.) adaptés aux architectures générées par AMICAL.
Ces composants sont décrits de telle sorte à être facilement assemblables et à respecter le modèle d'horloge utilisé. Seules les unités fonctionnelles effectuant des opérations varient d'un exemple à un autre selon les opérations exécutées dans la spécification comportementale de l'exemple. Une unité fonctionnelle sera présentée en exemple dans cette bibliothèque.

Comme toute bibliothèque, celle-ci suppose un modèle d'horloge bien défini afin de permettre une synchronisation globale du circuit. Dans ce cas, on suppose que:

1- Tous les éléments utilisent une même horloge appelée *clk*,

2- Tous les éléments de mémorisation sont actifs sur le front montant de l'horloge,

3- Tous les éléments de mémorisation sont connectés à un signal de remise à zéro global appelé *reset*.

On peut noter aussi, que pour tous les diagrammes de temps, on considère uniquement les modèles fonctionnels (temps de réponse égal à zéro)

3.5.1 Le registre

Il existe trois sortes de registres dans la bibliothèque d'AMICAL: le registre variable (*VariableRegister*), le registre constante (*ConstReg*) et le registre compte-rendu (*FlagReg*). Le premier est utilisé pour le stockage et l'échange de valeurs intermédiaires utilisées dans les opérations. Le second a une valeur fixe et est utilisé généralement pour l'affectation des signaux de validation en sortie. Suivant la description RTL faite, ce registre peut ne pas en être un et peut correspondre à une valeur cablée directement. Le dernier type a la même structure que le registre variable, sauf qu'en plus de ses ports, il a un signal de sortie pour le compte-rendu qui va directement à la partie contrôle (figure 3.24).

Le registre est décrit comme une entité possédant comme signaux d'entrée, les

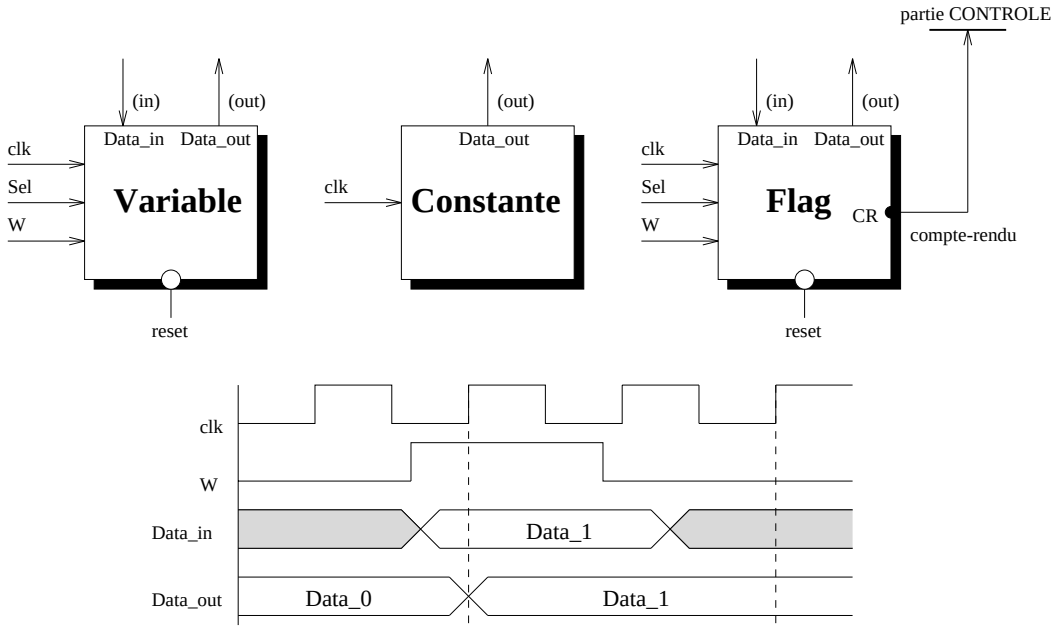


Figure 3.24: *Représentation d'un registre.*

signaux de synchronisation *clk* et *reset* et un signal d'écriture des données dans le registre *W*. Il est connecté aux bus à travers des interrupteurs permettant le transit des données du registre vers les bus et vice versa. Le registre *FlagReg* possède une sortie supplémentaire qui est branchée directement à la partie contrôle, servant de compte-rendu. La synchronisation est définie de telle sorte qu'à chaque front montant de l'horloge *clk*, et lorsque le signal d'écriture, *W*, est égal à '1', le registre mémorise la valeur présente sur le bus interne (valeur présente sur son entrée). Par contre, la valeur du registre est à tout moment accessible en lecture. En ce qui concerne le registre compte-rendu (*FlagReg*), le compte-rendu est toujours présent dans la partie contrôle.

3.5.2 L'unité fonctionnelle d'entrée/sortie: *FU_IO*

L'unité d'entrée/sortie est définie avec le signal d'entrée *in1* et le signal de sortie *out1*. Dans ce cas, l'unité d'entrée/sortie est totalement asynchrone. Elle laisse transiter les valeurs sans la mémorisation synchrone à l'intérieur. La figure 3.25 illustre ce phénomène sur un chronogramme.

L'unité d'entrée/sortie peut être utilisée à des fins d'amplification ou autres. Elle n'est pas toujours utile (et/ou utilisée) mais peut relier le chemin de données avec l'exterieur.
On peut noter que, pour des versions plus récentes, l'unité d'entrée/sortie n'est plus nécessaire.

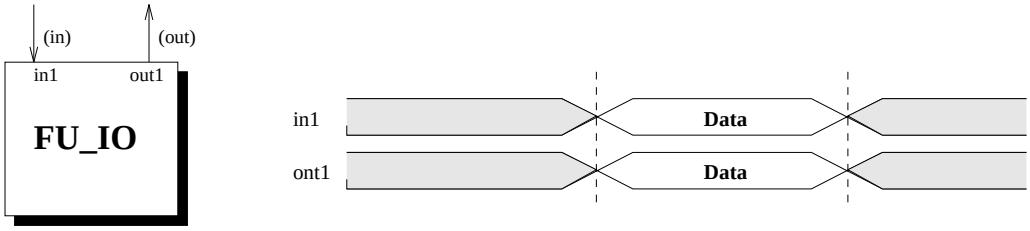


Figure 3.25: *Représentation de l'unité d'entrée/sortie* FU_IO.

3.5.3 Une unité fonctionnelle: *FU_OP*

Une unité fontionnelle *FU_OP* est utilisée pour illustrer les unités fonctionnelles en général. *FU_OP* est définie avec les signaux d'entrée *in1* et *in2*, le signal de sélection de l'opérateur *Sel* et le signal de sortie *out1*. *in1*, *in2* et *out1* sont des signaux apparent dans le chemin de données, ils seront reliés à des bus de ce dernier. Le signal *Sel* sera par contre transparent pour le chemin de données (affichage AMICAL). Il n'existera que par son activation et sa désactivation par le contrôleur. Le signal *out1* transportera le résultat de l'opération exécutée par l'unité fonctionnelle vers le bus correspondant. L'unité fonctionnelle peut être complètement combinatoire, dans ce cas, dès que les données sont prêtes sur les entrées *in1* et *in2*, *FU_OP* calcule et rend le résultat sur *out1* (figure 3.26).

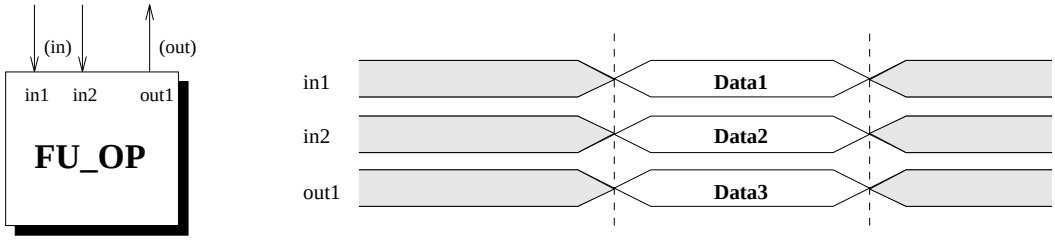


Figure 3.26: *Représentation d'une unité fonctionnelle combinatoire.*

3.5.4 Les connecteurs de bus

Les connecteurs (ou *switchs*) sont utilisés de deux façons: horizontalement et verticalement (figure 3.27).

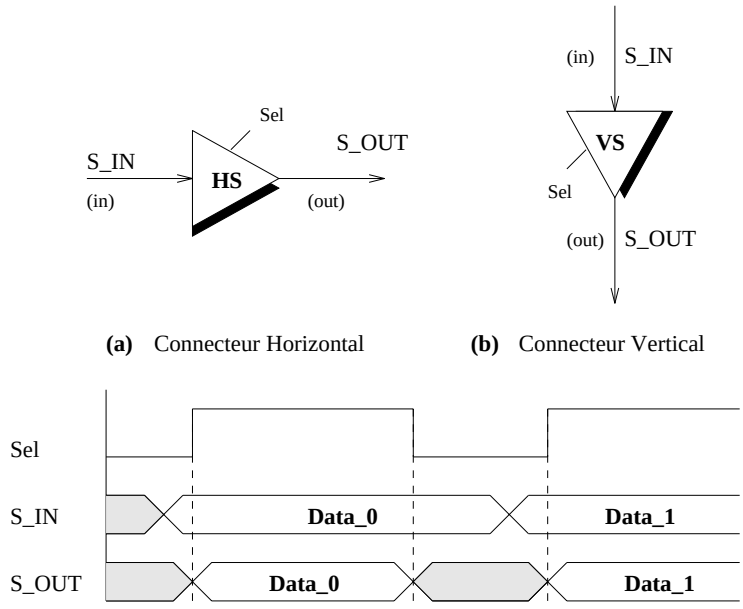


Figure 3.27: *Représentation d'un connecteur de bus.*

C'est le même connecteur qui définit le fonctionnement des deux modes. Les connecteurs horizontaux sont utilisés pour relier des segments de bus de la même piste ou des bus externes à des bus internes. Les connecteurs verticaux sont utilisés pour connecter des composants à des bus internes. Ces *switchs* sont, tout simplement, des éléments unidirectionnels possédant un signal de commande *Sel* permettant la sélection du composant et le transit des données.

Pour les deux types de connecteurs, si le signal *Sel* est égal à '1', la valeur logique du bus *S_IN* passe au bus *S_OUT*.

3.5.5 Le bus

Le bus est un ensemble de signaux pouvant acheminer des données dans les deux sens. C'est un conducteur de n bits pouvant être décomposé en plusieurs segments de bus séparés par des connecteurs de bus.

La figure 3.28 montre un exemple de décomposition d'un bus (*Bus_1*) en trois segments de bus (*BUS_11*, *BUS_12* et *BUS_13*) séparés par les connecteurs *S1*, *S2*, *S3* et *S4*. La donnée présente sur le *BUS_11* peut être transférée sur le *BUS_13* en actionnant les connecteurs *S1* et *S3*, c'est-à-dire, *Sel* = '1' (pour ces deux connecteurs). Le chemin de données a deux bus principaux pour transférer l'information entre les composants.

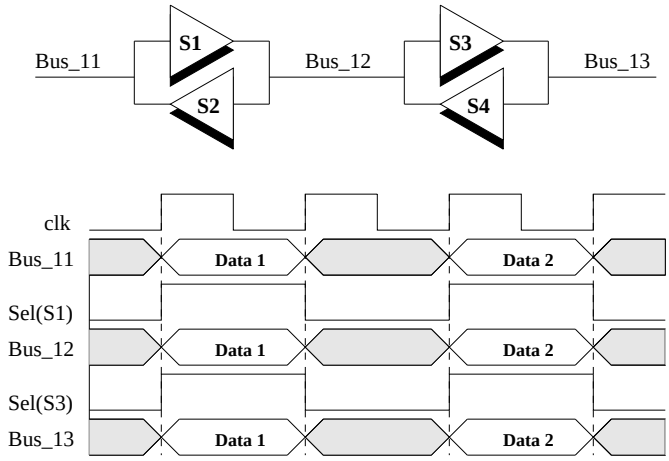
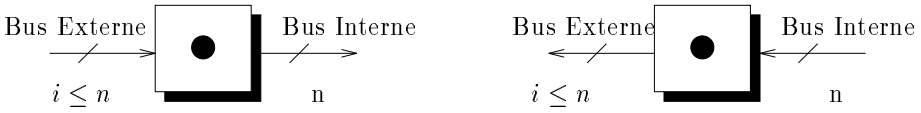


Figure 3.28: *Représentation d'une séparation de bus.*

Dans le cas d'AMICAL, le comportement de ces pistes est très important. Deux composants ne peuvent jamais écrire en même temps sur un même bus, de même, l'écriture et la lecture d'un composant ne peuvent se faire sur le même bus.

3.5.6 Le connecteur externe



(a) Connecteur d'entrée **(b)** Connecteur de sortie

Figure 3.29: *Schéma représentatif du connecteur externe.*

C'est une cellule qui est considérée comme un plot d'entrée/sortie. La taille des signaux de connection peut être, soit d'un bit, pour certains signaux de validation en entrée et/ou en sortie, soit d'un vecteur de bits, pour des signaux de données en entrée et/ou en sortie (figure 3.29).

Pour l'exemple du pgcd, pris comme exemple dans cette section (figure 3.2) pour montrer les différents types de composants utilisés, chaque cellule est appelée un certain nombre de fois et le tout forme la partie opérative du circuit. La table 3.1 résume ces différents composants par des nombres.

Composant	Nom Interne	Quantité
Connecteur externe	ExtCon	4
Soustracteur	FU_SUB	1
Entrée/Sortie	FU_IO	2
Registre de constante	ConstReg	2
Registre de compte-rendu	FlagReg	2
Connecteur horizontal	HSwitch	6
Connecteur vertical	VSwitch	17

Table 3.1: *Résumé des composants du chemin de données du pgcd.*

3.6 Conclusion

Dans ce chapitre, quelques modèles architecturaux pouvant être générés par AMICAL ont été présentés. Pour des fins de synchronisation, ou de test, l'architecture abstraite de base peut être transformée en plusieurs autres architectures. Les modèles de synchronisation fonctionnent, soit en mode normal (sans recouvrement entre la partie opérative et la partie contrôle), soit en mode *pipeliné* (avec recouvrement des deux parties). En mode *non pipeliné*, les signaux de contrôle envoyés par la partie contrôle à la partie opérative et les compte-rendus retournés à partir de la partie opérative à la partie contrôle ne sont pas obligatoirement mémorisés, dans des bistables (*flips-flops*), afin de permettre la synchronisation sur front. Par contre, en mode *pipeliné*, que ce soit pour une architecture avec ou sans interface, la mémorisation des signaux de commande et des compte-rendus est obligatoire. Dans une architecture avec interface, la mémorisation se fait au niveau de l'interface qui utilise des *D flips-flops* pour stocker les données depuis leur génération jusqu'au (prochain) front montant, tandis que dans une architecture sans interface, la mémorisation des signaux de commande se fait au niveau des composants. De plus, pour les modèles *pipelinés*, l'outil d'ordonnancement insère des états d'attente (d'un cycle pour les architectures à contrôleur à un seul étage, de deux cycles pour celles à contrôleur à deux étages et ainsi de suite ...) au cas où le contrôleur aurait besoin des résultats des opérations de la partie opérative pour évaluer les conditions des cycles suivants.

Finalement, un exemple de bibliothèque de composants a été présenté afin de permettre la génération de la description du circuit au niveau transfert de registre, à partir d'une spécification comportementale, compatible avec les outils de simulation et de synthèse logique existants.

Dans les chapitres suivants, la personnalisation de l'architecture abstraite pour la génération automatique de différentes architectures détaillées sera présentée en détail. PAT, l'outil de traduction SOLAR-VHDL ainsi que la validation, par simulation, des deux descriptions (comportementale en entrée d'AMICAL et structurelle

en sortie) seront détaillés dans le Chapitre 4.
Le Chapitre 5 présentera une étude comparative des différents modèles architecturaux générés par AMICAL.

CHAPITRE 4

Lien entre AMICAL et les outils de conception au niveau transfert de registres

Chapitre 4

Lien entre AMICAL et les outils de conception au niveau transfert de registres

Ce chapitre présente un outil, appelé PAT (Programmable Architecture Translator), permettant l'interfaçage d'AMICAL avec les outils de conception au niveau transfert de registres. PAT permet, d'une part, la traduction de la sortie d'AMICAL, donnée sous la forme SOLAR, en son équivalent VHDL, et d'autre part, la personnalisation de l'architecture abstraite en y ajoutant des signaux globaux tels que remise à zéro, horloge, etc. et des cellules additionnelles telles qu'un bloc de synchronisation, etc. En utilisant VHDL comme sortie, AMICAL peut facilement s'interfacer avec les outils de simulation et de synthèse logique au niveau transfert de registres.

4.1 Introduction

Comme mentionné dans les chapitres précédents, AMICAL, [74, 73] prend en entrée une spécification comportementale donnée en VHDL, une bibliothèque externe d'unités fonctionnelles et un fichier de contraintes. Il exécute la synthèse architecturale dont les tâches essentielles sont l'ordonnancement et l'allocation. En

fin de synthèse, AMICAL génère une architecture abstraite sous une forme intermédiaire appelée SOLAR [46]. L'architecture cible d'AMICAL étant un circuit à flux de contrôle (*control-flow dominated*) contenant une partie opérative (*netlist*) et un contrôleur, l'architecture abstraite générée est obtenue sous la forme de trois fichiers correspondant au chemin de données, au contrôleur et à l'interconnexion de ces deux blocs pour former le circuit global.

Le but de ce chapitre est de passer à un niveau encore inférieur, et ce, en cherchant à intégrer AMICAL dans les outils de CAO tels que SYNOPSYS, UNICAD, etc.

Pour passer d'un niveau à un autre à l'aide de différents outils, l'outil de synthèse comportementale doit être compatible avec les outils de synthèse de niveau inférieur aussi bien qu'avec les outils de simulation. Il sera alors possible de combiner plusieurs outils de synthèse de niveaux différents afin que les blocs générés par l'un puissent être utilisés par l'autre.

AMICAL accepte quasi tout le sous-ensemble VHDL concernant les instructions séquentielles mais pas les instructions concurrentes. Cela implique que les détails des caractéristiques matérielles telles que les modèles de synchronisation ne sont pas explicitement spécifiés dans la représentation comportementale en entrée d'AMICAL. L'outil d'ordonnancement d'AMICAL [67] génère un ordonnancement initial des événements sous forme de table de transitions. Les algorithmes d'allocation et de micro-ordonnancement traitent de la synchronisation et de l'ordonnancement des transferts individuels. En effet, la synchronisation à ce niveau est implicite et ne peut être rendue explicite que par la personnalisation de l'architecture abstraite en y ajoutant des éléments de synchronisation tels que le bloc de synchronisation, les signaux d'horloge, la remise à zéro, etc. Comme mentionné dans le chapitre précédent, suivant les signaux et les cellules ajoutés, cette méthode de personnalisation permet de générer plusieurs modèles d'architectures, et ce, en gardant le même comportement initial (voir section 3.3).

La personnalisation de l'architecture abstraite est assurée par un outil appelé PAT utilisant un fichier "global" à l'intérieur duquel le concepteur peut spécifier

tous les éléments qu'il veut ajouter pour générer l'architecture désirée. De plus, cette architecture peut être traduite du format SOLAR en son équivalent VHDL et ainsi, en utilisant VHDL comme sortie, AMICAL peut facilement s'interfacer avec les outils de simulation et de synthèse logique tels que Synopsys [82].

De plus, l'entrée et la sortie d'AMICAL peuvent être vérifiées grâce à l'utilisation d'outils de simulation décrits pour VHDL. Ceci permet la validation des résultats et la vérification de la bonne fonctionnalité des circuits générés. La validation se fait en comparant les simulations en entrée et en sortie d'AMICAL pour les mêmes vecteurs de test.

Ce chapitre sera organisé comme suit:

- La section suivante, fera un bref rappel de la sortie SOLAR du système AMICAL.
- PAT sera détaillé dans la section 3. Son fonctionnement concernant le rajout des signaux "globaux" à l'architecture abstraite pour la génération d'architectures détaillées sera illustré dans cette même section. Les fichiers VHDL générés par PAT seront présentés, ainsi que les modifications et les autres informations nécessaires pour la génération du circuit sur silicium.
- Dans la section 4, le pgcd (Plus Grand Commun Diviseur) sera utilisé comme exemple pour illustrer les résultats de la traduction et de la comparaison des simulations aux niveaux comportemental et transfert de registres.
- Finalement, une brève conclusion résumera ce chapitre.

4.2 Les fichiers de sortie d'AMICAL

Selon le modèle architectural généré (avec ou sans interface), la sortie d'AMICAL, au niveau transfert de registres, contiendra trois ou cinq fichiers décrivant le circuit entier. Ces fichiers sont donnés sous une forme intermédiaire nommée SOLAR [46].

4.2.1 Modèle sans interface

Dans ce type de modèle, la sortie contient trois fichiers décrivant une configuration globale du circuit, une partie opérative et une partie contrôle.
La partie opérative est un ensemble de composants prédéfinis (registres, connecteurs, unités fonctionnelles, etc.) interconnectés. Le contrôleur, quant à lui, est une table de transitions spécifiant les états courants, les conditions, les états suivants et les opérations à exécuter dans chaque transition (voir Chapitre 3).

4.2.1.1 *Configuration globale*

Le fichier décrivant la configuration du circuit généré par AMICAL contient une unité de conception hiérarchique représentant la structure montrée dans la figure 3.1 de la section 3.2. Cette structure contient une partie opérative, une partie contrôle, les connexions entre ces deux dernières et les interfaces externes.
La partie opérative est donnée sous forme de *netlist* composée d'une liste de composants, d'une liste de signaux et des interconnexions entre les composants et les signaux. Elle reçoit des données externes et les signaux de commande générés par le contrôleur. Les sorties de la partie opérative incluent les résultats du circuit et les compte-rendus pour la partie contrôle.
Le contrôleur quant à lui, décrit les états, les transitions entre états et les signaux de contrôle sélectionnés pour activer les actions appropriées dans la partie opérative pour chaque transition. Il exécute un algorithme de contrôle et envoie les signaux de commande à la partie opérative pour l'exécution des opérations dans cette dernière. Ses entrées incluent des signaux externes et des compte-rendus émis par la partie opérative.

Un aperçu général du fichier SOLAR de la configuration globale est donné par la figure 4.1, représentant le niveau hiérarchique le plus haut du circuit.

```
1  (Solar AMICAL_for_Circuit
2     (DesignUnit nom_du_circuit_circuit
3        (View structure (ViewType "InterconnectedSystems")
4           (Interface
5              {(Port (Array nom_du_connecteur longeur_de_bits)
6                 (Direction IN | OUT | INOUT)
7                 (Property PortType DATA | CONTROL)
8              )}
9              {(Port nom_du_connecteur
10                (Direction IN | OUT | INOUT)
11                (Bit)
12                (Property PortType DATA | CONTROL)
13             )}
14          )
15          (Contents
16             (Instance ControlPart
17                (ViewRef behaviour RT-Level ControlPart)
18                {(PortInstance nom_du_connecteur
19                   (Property PortType DATA | CONTROL)
20                )}
21             )
22             (Instance DataPath
23                (ViewRef behaviour RT-Level DataPath)
24                {(PortInstance nom_du_connecteur
25                   (Direction IN | OUT | INOUT)
26                   (Property PortType DATA | CONTROL)
27                )}
28             )
29             {(Net nom_d'interconnexion
30                (Joined
31                   (PortRef nom_du_connecteur
32                      (InstanceRef nom_du_composant))
33                   (PortRef nom_du_connecteur
34                      (InstanceRef nom_du_composant)))
35             )}
36          )
37       )
38    )
39 )
```

Figure 4.1: *Format SOLAR de la configuration globale générée par AMICAL.*

Le fichier comprend trois types d'informations: les interfaces externes, les composants utilisés et les connexions entre blocs (composants, ...). Le mot clé *Interface* spécifie les interfaces externes, correspondant aux signaux de contrôle externes et aux bus de données externes. Ces interfaces sont les mêmes que celles déclarées dans la description des macro-cycles. Le mot clé *Contents* décrit l'organisation interne du circuit qui est composée de deux blocs. Le contrôleur correspond au bloc *ControlPart*. Ses connecteurs (*PortInstance*) sont les signaux de contrôle envoyés à la partie opérative, les compte-rendus venant de cette dernière et les signaux de

contrôle externes. La partie opérative correspond au bloc *DataPath*.

Tous les signaux, y compris les interfaces externes, peuvent être de type bit ou un vecteur de bits. Les connexions entre la partie opérative et la partie contrôle sont décrites avec le mot clé *Net*. Chaque connexion reçoit un nom unique. Une connexion contient une référence aux deux signaux; un signal peut appartenir, soit à l'un des deux blocs, soit à l'interface globale.

4.2.1.2 *Partie Opérative*

Parmi les résultats de la compilation par AMICAL, on distingue le réseau d'interconnexions de la partie opérative et la spécification des signaux de contrôle. Le réseau d'interconnexions décrit la structure de la partie opérative avec les interfaces externes, les composants alloués et leurs interconnexions. Les signaux de contrôle commandent les unités fonctionnelles et réalisent les transferts élémentaires entre composants.

AMICAL considère la partie opérative comme un bloc qui pourra être réutilisé pour former d'autres assemblages. Son interface avec l'extérieur (contrôleur et extrérieur du circuit) correspond aux connecteurs du chemin de données qui sont les bus permettant d'échanger les données avec l'extérieur, les compte-rendus (registres utilisés par la partie contrôle) et les signaux de commande venant de la partie contrôle. La figure 4.2 donne un aperçu du format SOLAR de la partie opérative générée par AMICAL.

La partie opérative est constituée d'un assemblage de ressources allouées par AMICAL. La partie opérative est donc décrite par une spécification structurelle. Pour chaque composant (mot clé *Instance*), le nom du composant et ses connecteurs sont définis, ainsi que le nom du fichier qui décrit les caractéristiques de ce composant. Et pour chaque connexion (mot clé *Net*), on décrit les deux connecteurs qu'elle relie.

```
1  (Solar AMICAL_for_Datapath
2     (DesignUnit nom_du_circuit_datapath
3        (View struxture (ViewType RT-Level)
4           (Interface
5              {(Port (Array nom_du_connecteur longeur_de_bits)
6                 (Direction IN | OUT | INOUT)
7                 (Property PortType DATA | CONTROL)
8              )}
9              {(Port nom_du_connecteur
10                (Direction IN | OUT | INOUT)
11                (Bit)
12                (Property PortType DATA | CONTROL)
13             )}
14          )
15          (Contents
16             {(Instance nom_du_composant
17                (ViewRef behaviour RT-Level sorte_de_composant)
18                {(PortInstance nom_du_connecteur
19                   (Property PortType DATA | CONTROL))}
20                [(Property PlaceOrder entier)]
21             )}
22          )}
23          {(Net nom_d'interconnexion
24             (Joined
25                (PortRef nom_du_connecteur
26                   (InstanceRef nom_du_composant))
27                (PortRef nom_du_connecteur
28                   (InstanceRef nom_du_composant)))
29          )}
30       )
30    )
30 )

sorte_de_composant ::=  connecteur externe | FU_nom | FlagRegister | VariableRegister |
                        ConstantRegister | connecteur horizontal | connecteur vertical | Bus
```

Figure 4.2: *Format SOLAR de la partie opérative générée par AMICAL.*

4.2.1.3 *Partie Contrôle*

Au niveau transfert de registres, chaque opération est définie par son schéma d'exécution défini pour chaque cycle de base (micro-cycle). Pendant un micro-cycle, un ensemble de transferts élémentaires s'exécutent en parallèle. Par contre, au niveau structurel (après la séparation du modèle en deux blocs: une partie opérative et un contrôleur), chaque micro-cycle doit être transformé en un vecteur de commande. Cette séparation se fera en deux étapes. La première génère deux types d'informations:

- les commandes de sélection des composants de la partie opérative,

- les chemins de connexion qui expriment le comportement pendant un micro-cycle.

La première étape donne le format de description du contrôleur. Elle décrit la liste des connecteurs du bloc (figure 4.3).

```
1  (Solar AMICAL_for_Controller
2     (DesignUnit nom_du_circuit
3        (View behaviour (ViewType RT-Level)
4           (Interface
5              {(Port (Array nom_du_connecteur longeur_de_bits)
6                 (Direction IN | OUT | INOUT)
7                 (Property PortType DATA | CONTROL)
8              )}
9              {(Port nom_du_connecteur
10                (Direction IN | OUT | INOUT)
11                (Bit)
12                (Property PortType DATA | CONTROL)
13             )}
14          )
15          (Contents
16             (StateTable Controller
17                {(State nom_du_state
18                   (Case
19                      (Alt expression_du_condition
20                         (MicroCycle ID_du_micro_cycle)
21                         {(Path {nom_du_composant_dans_le_chemin)}
22                         {(Assign nom_du_connecteur valeur)}
23                         (NextState nom_du_state)
24                      )}
25                   )
26                )})
27             (StateList {nom_des_state})
28          )
29       )
30    )
31 )
```

Figure 4.3: *Format SOLAR du contrôleur généré par AMICAL.*

Ces connecteurs (mot clé *Interface*) sont déjà décrits comme les connecteurs du bloc (*ControlPart*) dans la figure 4.1. La description du contrôleur est faite par un diagramme d'états écrit en format SOLAR. La description d'un micro-cycle (mot clé *Alt*) comprend les informations sur le séquencement (l'état courant, la condition de transition et l'état suivant). Cette description contient les commentaires donnant l'ordre des micro-cycles et les chemins de connexions. De plus, il affecte des signaux de contrôle actifs (mot clé *Assign*). Ces chemins de connexions correspondent aux transferts exécutés, en parallèle, dans le micro-cycle. L'ensemble de ces signaux

de contrôle doit arriver à la partie opérative pendant un même micro-cycle. Cette description du contrôleur permet de générer le vecteur de commande.

La seconde étape génère le vecteur de commande complet. Cette étape est réalisée au moment de la génération du contrôleur. Pour la description générée par AMICAL, le vecteur de commande sera limité aux commandes de sélection des composants. Ces vecteurs commandent les unités fonctionnelles (sélection d'opérations et activation d'unités fonctionnelles) et les transferts à travers les chemins de connexions. Un chemin de connexion est constitué d'une série de composants par lesquels les données d'un transfert sont acheminées. Le premier et le dernier composants sont respectivement la source et la destination du transfert (figure 4.4).

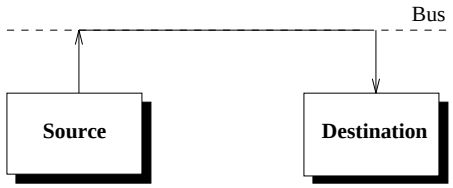


Figure 4.4: *Représentation d'un transfert simple.*

4.2.2 Modèle avec interface

Comme mentionné dans le Chapitre 3, un des modèles architecturaux généré par AMICAL est celui avec des interfaces de mémorisation entre la partie opérative et la partie contrôle (section 3.2.2). Ces interfaces sont à base de cellules *D flips-flops*. Leur rôle est de mémoriser le signal entrant (un signal de commande envoyé par la partie contrôle ou un signal de compte-rendu retourné par la partie opérative) jusqu'au prochain front montant de l'horloge pour le générer.

4.2.2.1 *Configuration globale*

Dans ce cas, le circuit global ne contient pas seulement une partie opérative et une partie contrôle comme composants essentiels, mais il aura en plus l'instanciation des deux interfaces (*PC_PO*, pour les signaux de commande sortant de la partie contrôle et *PO_PC*, pour les compte-rendus retournés par la partie opérative; figure 3.4).

La figure 4.5 montre un extrait SOLAR de la configuration globale d'une telle architecture.

```
1  (Solar AMICAL_for_Circuit_with_scan
2     (DesignUnit nom_du_circuit_scan_circuit
3        (View structure (ViewType "InterconnectedSystems")
4           (Interface
5              {(Port (Array nom_du_connecteur longeur_de_bits)
6                 (Direction IN | OUT | INOUT)
7                 (Property PortType DATA | CONTROL)
8              )}
9              {(Port nom_du_connecteur
10                (Direction IN | OUT | INOUT)
11                (Bit)
12                (Property PortType DATA | CONTROL)
13             )}
14          )
15          (Contents
16             (Instance ControlPart
17                (ViewRef AbstractArchitecture nom_du_circuit_control)
18                {(PortInstance nom_du_connecteur
19                   (Property PortType DATA | CONTROL)
20                )}
21                )
22             (Instance DataPath
23                (ViewRef AbstractArchitecture nom_du_circuit_datapath)
24                {(PortInstance nom_du_connecteur
25                   (Direction IN | OUT | INOUT)
26                   (Property PortType DATA | CONTROL)
27                )}
28             (Instance scan_pc_po
29                (ViewRef AbstractArchitecture nom_du_circuit_scan_pc_po)
30                {(PortInstance nom_du_connecteur
31                   (Property PortType CONTROL)
32                )}
33             )
34             (Instance scan_po_pc
35                (ViewRef AbstractArchitecture nom_du_circuit_scan_po_pc)
36                {(PortInstance nom_du_connecteur
37                   (Property PortType DATA)
38                )}
39             )
40             {(Net nom_d'interconnexion
41                (Joined
42                   (PortRef nom_du_connecteur
43                      (InstanceRef nom_du_composant))
44                      (PortRef nom_du_connecteur
45                      (InstanceRef nom_du_composant)))
46             )}
47          )
48 )  )  )
```

Figure 4.5: *Format SOLAR de la configuration globale avec interface.*

La structure SOLAR de la partie opérative et celle de la partie contrôle sont les mêmes que celles présentées précédemment.

4.2.2.2 *L'interface de mémorisation*

Le fichier SOLAR décrivant une des deux interfaces (celle mémorisant les signaux de commande issue de la partie contrôle vers la partie opérative) est montré dans la

figure 4.6

```
1  (Solar AMICAL_for_scan_pc_po
2     (DesignUnit nom_du_circuit_scan_pc_po
3        (View AbstractArchitecture (ViewType RT-Level)
4           (Interface
5              (Port (nom_du_signal_in
6                 (Direction IN) (BIT)
7                 (Property PortType CONTROL))
8              (Port (nom_du_signal_out
9                 (Direction OUT) (BIT)
10                (Property PortType CONTROL))
11          ... (- Déclaration de tous les signaux de commande en ENTREE
                   et en SORTIE de l'interface)
12          )
13          (Contents
14             (Instance scan_cell
15                (ViewRef Implementation scan_cell)
16                (PortInstance data_in Property PortType DATA))
17                (PortInstance data_out Property PortType DATA)))
18             ... (- Déclaration de toutes les cellules de l'interface)
19                   (Net ...
20                   ... (- Connexion de tous les signaux avec toutes les cellules)
21          )
22       )
23    )
24 )
```

Figure 4.6: *Format SOLAR de l'interface de mémorisation.*

Cette interface ne contient que l'instanciation des cellules et la déclaration des signaux d'entrée/sortie de chaque cellule. Le nombre de cellules est égal, dans ce cas, au nombre de signaux de commande augmenté du nombre total de bits des signaux de compte-rendu retournés par la partie opérative.

4.3 Génération de VHDL pour la simulation et la synthèse RTL

En sortie d'AMICAL, plusieurs fichiers (selon le modèle d'architecture généré) décrivant le circuit complet au niveau RTL, sont disponibles à partir du résultat de la synthèse architecturale. Néanmoins, il reste quelques problèmes à résoudre avant de pouvoir générer le circuit sur silicium.

Le problème le plus exigeant est de traduire la description dans un langage permettant la simulation du circuit, et son interface avec les outils de synthèse au niveau transfert de registres.

Les fichiers de sortie d'AMICAL sont donnés en format SOLAR. Pour interfacer AMICAL avec les outils de synthèse au niveau transfert de registres, ces fichiers doivent être traduits en VHDL tout en personnalisant l'architecture générée en y ajoutant des signaux globaux tels que les horloges globales, les signaux de reset, etc. qui sont jusqu'à maintenant virtuels. De plus une bibliothèque de composants est nécessaire pour la traduction et la validation de la fonctionnalité du circuit final. Cette bibliothèque est décrite à deux niveaux différents: le premier (servant à la traduction des fichiers de sortie d'AMICAL) décrit les ports des composants et leurs types dans le but de faciliter la transition des composants abstraits d'AMICAL vers les composants VHDL; et le second définit la fonctionnalité des composants donnée en VHDL dans le but de permettre la validation des circuits générés par simulation. Pour la synthèse, les outils de synthèse logique existants doivent fournir ces descriptions.

Une interface a été mise au point pour permettre la traduction de la sortie d'AMICAL en VHDL. Ce traducteur, appelé PAT [1, 3, 5, 6] est capable d'ajouter des éléments de synchronisation globaux.
Le chemin de données de PAT est résumé dans la figure 4.7.

D'après le schéma de la figure 4.7, les fichiers VHDL décrivant le circuit entier peuvent être générés à partir des fichiers SOLAR respectifs, en utilisant une

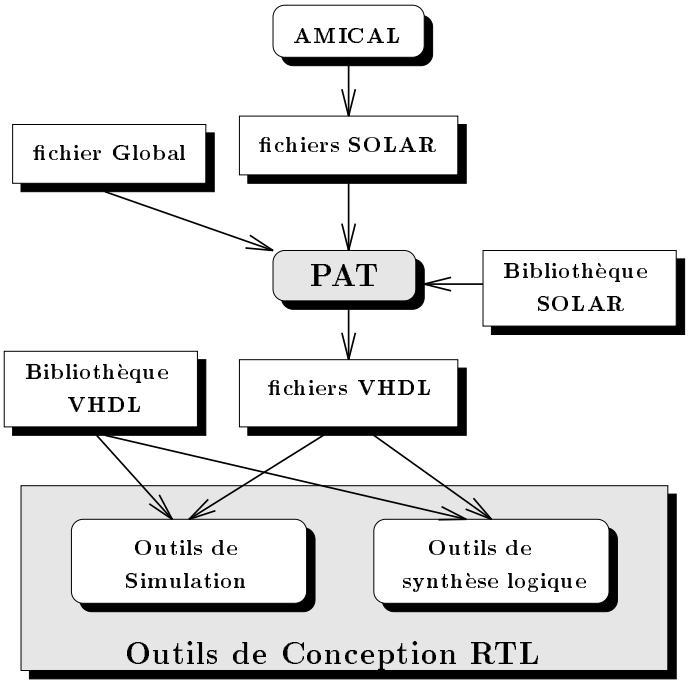


Figure 4.7: *PAT: vue globale.*

bibliothèque de composants SOLAR décrivant les ports d'entrée/sortie de chaque cellule et un fichier "global" dans lequel le concepteur peut spécifier les éléments additionnels (signaux d'horloge, signaux de contrôle, cellules supplémentaires, etc.) qu'il veut ajouter a l'architecture générée. L'architecture ainsi modifiée s'appelle architecture détaillée (voir section 3.3). Cette architecture est donnée dans un langage compatible avec les outils de simulation et de synthèse logique existants.

4.3.1 La bibliothèque de composants SOLAR

Cette bibliothèque est créée de telle sorte qu'elle ne contienne que les descriptions des ports de chaque composant utilisé par la partie opérative. Cette bibliothèque a été introduite dans le but de faciliter la transition entre les composants abstraits utilisés par AMICAL et les composants VHDL utilisés par les outils de simulation et les outils de synthèse (Synopsys). Un exemple de représentation d'un registre exprimé en SOLAR est donné dans la figure 4.8 :

```
1  (Solar register
2     (Designunit FlagReg
3        (View Implementation (Viewtype "communicating")
4           (Interface
5              (Port reset                    (Direction IN)     (BIT))
6              (Port clk                      (Direction IN)     (BIT))
7              (Port W                        (Direction IN)     (BIT))
8              (Port (Array Data_IN 8)        (Direction IN)     (BIT))
9              (Port (Array Data_OUT 8)       (Direction OUT)    (BIT))
10             (Port (Array CR 8)             (Direction OUT)    (BIT))
11          )
12       )
13    )
14 )
```

Figure 4.8: *Représentation d'un composant SOLAR.*

Ce registre est un composant qui est utilisé à la fois comme registre variable pour la mémorisation des valeurs des calculs intermédiaires et comme registre de compte-rendu. C'est grâce au port *CR* que les valeurs du registre sont transmises au contrôleur pour l'évaluation des conditions et la génération du vecteur de commande. Dans ce fichier, dont l'extension est "*.solar*", ne sont déclarés que les ports du composant, leur direction et leur type. Ceci est suffisant pour permettre la transition des composants abstraits vers les composants réels donnés en VHDL.

4.3.2 Le fichier de synchronisation global

Afin de générer le VHDL contenant les signaux pour différentes méthodes de synchronisation, l'interface PAT permet au concepteur de spécifier explicitement ces signaux dans un fichier "global". Pendant la génération du code VHDL, ces signaux sont ajoutés aux endroits appropriés.

Le principe de PAT est le suivant:

- Tous les signaux globaux externes doivent être ajoutés aux ports où ils ont été définis. Ils doivent apparaître dans les ports de l'entité spécifiée et dans la déclaration des composants.

- Tous les signaux globaux internes doivent être ajoutés aux signaux déclarés dans l'architecture.

- Toutes les cellules doivent être déclarées et instanciées dans l'architecture.
- Dans les *Port maps*, tous les ports qui ne sont pas connectés doivent être connectés aux signaux globaux.
- Dans l'architecture du contrôleur, un processus de synchronisation, sensitif à l'horloge et à la remise à zéro, doit être ajouté.

La syntaxe du fichier global est basée sur SOLAR. La grammaire d'un tel fichier est décrite dans la figure 4.9.

```
PAT        ::=  (GLOBAL nom_du_circuit {bloc})
bloc       ::=  (BLOCK nom_du_block {signal} {cellule})
signal     ::=  (SIGNAL nom_du_signal [direction] [type] [attribut] [local])
direction  ::=  (DIR mode)
mode       ::=  IN | OUT | INOUT | INTERNAL
type       ::=  (BIT)
attribut   ::=  (ATTR mode_attr)
mode_attr  ::=  CLOCK | RESET | CONTROL
local      ::=  nom_local_du_signal
```

Figure 4.9: *Grammaire de PAT.*

Un fichier "global" (défini par le mot clé *Global*) peut contenir plusieurs blocs (mot clé *Block*) à l'intérieur desquels le concepteur peut déclarer tous les signaux (mot clé *Signal*) et toutes les cellules (mot clé *Glue*) qu'il veut ajouter à l'architecture abstraite générée par AMICAL.

Les signaux contiennent obligatoirement une direction, un type et un attribut. Ils peuvent aussi contenir un nom local. Ceci veut dire que ce signal global est toujours connecté aux signaux ayant le nom (*nom_local*) déclaré dans le circuit (figure 4.10). Un attribut spécifie si le signal est du type horloge (*CLOCK*), contrôle (*CONTROL*) ou remise à zéro (*RESET*). Les types, pour le moment, sont limités au BIT. La direction peut être *IN* (un signal d'entrée), *OUT* (un signal de sortie), *INOUT* (un signal d'entrée/sortie), ou *INTERNAL* (un signal interne).

Les cellules ajoutées sont des unités fonctionnelles (simples ou complexes) qui, par leur fonctionnalité, peuvent transformer des signaux externes en des signaux internes ayant un comportement différent. Un exemple d'une cellule est le générateur

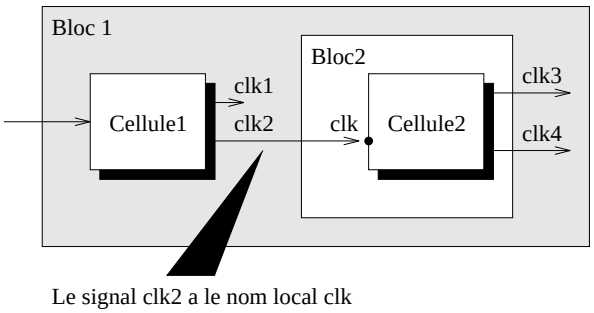


Figure 4.10: *Exemple d'un signal local.*

d'horloge qui, à partir d'une horloge externe, peut générer plusieurs horloges internes.

Toutes ces informations sont données dans un fichier dont l'extension est "*.global*". La syntaxe de ce fichier est très facile à comprendre. Chaque mot clé est placé entre parenthèse avec le nom du signal (ou cellule) et le contenu correspondant (type, attribut, etc.). Ce fichier a la structure montrée dans la figure 4.11.

```
1   (Global nom_de_l'exemple.global
2       {(Block nom_du_bloc
3           {(Signal nom_du_signal
4               (Dir IN | OUT | INOUT | INTERNAL)
5               (Attr CLOCK | RESET | CONTROL)
6               (Type BIT)
7               (Local nom_du_signal_local))
8           )}
9           {(Glue nom_de_la_cellule)
10          }
11      )}
12  )
```

Figure 4.11: *Format du fichier "global".*

Pour l'exemple du pgcd, une cellule de synchronisation est rajoutée à l'architecture initiale (figure 4.12). Cette cellule utilise deux signaux externes: *clk*, *reset* et génère quatre signaux internes: *data_clk* et *data_reset* pour la partie opérative (*gcd_datapath*), et *cont_clk* et *cont_reset* pour la partie contrôle (*gcd_control*).

Le fichier global utilisé par PAT, donnant les détails de ces signaux, pour cet exemple, est montré dans la figure 4.13. L'interprétation de ce fichier entraîne les transformations suivantes:

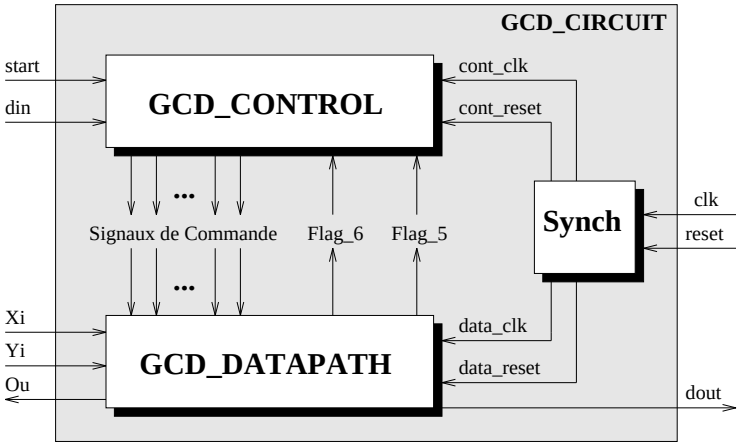


Figure 4.12: *Architecture modifiée pour le pgcd.*

(i) La déclaration de la cellule *Synch* dans le fichier de la configuration globale (*gcd_circuit*).

(ii) L'ajout des signaux globaux externes aux entités et composants appropriés (*clk* et *reset*).

(iii) L'ajout des signaux locaux aux architectures appropriées (*cont_clk*, *cont_reset*, *data_clk* et *data_reset*).

```
1  (Global pgcd.global
2      (Block pgcd_datapath
3          (Signal data_clk    (Dir IN)        (Attr CLOCK)    (Type BIT)      (Local clk))
4          (Signal data_reset  (Dir IN)        (Attr RESET)    (Type BIT)      (Local reset))
5      )
6      (Block pgcd_control
7          (Signal cont_clk    (Dir IN)        (Attr CLOCK)    (Type BIT)      (Local clk))
8          (Signal cont_reset  (Dir IN)        (Attr RESET)    (Type BIT)      (Local reset))
9      )
10     (Block pgcd_circuit
11         (Signal clk         (Dir IN)        (Attr CLOCK)    (Type BIT))
12         (Signal reset       (Dir IN)        (Attr RESET)    (Type BIT))
13         (Signal cont_clk    (Dir INTERNAL)  (Attr CLOCK)    (Type BIT))
14         (Signal cont_reset  (Dir INTERNAL)  (Attr RESET)    (Type BIT))
15         (Signal data_clk    (Dir INTERNAL)  (Attr CLOCK)    (Type BIT))
16         (Signal data_reset  (Dir INTERNAL)  (Attr RESET)    (Type BIT))
17         (Glue Synch)
18     )
19 )
```

Figure 4.13: *Fichier global pour l'exemple du pgcd.*

Le fichier global, illustré par la figure 4.13, est divisé en trois parties (*Block*) essentielles: la première contient les signaux concernant la partie opérative, la seconde,

ceux de la partie contrôle et la troisième, ceux du circuit global. On remarque qu'à ce dernier niveau, le mot clé *Glue* est utilisé pour définir le bloc de synchronisation qui constituera une cellule de la configuration en plus des deux parties (partie opérative et partie contrôle) initialement générées par AMICAL.

4.3.3 Génération des fichiers VHDL

4.3.3.1 *Traduction SOLAR-VHDL*

L'idée de traduire SOLAR en VHDL permet de réaliser deux objectifs à la fois:

- Avoir accès aux environnements de CAO de VLSI qui sont de plus en plus basés sur VHDL.
- Définir un modèle exécutable des descriptions SOLAR pour la vérification et la validation des circuits générés.

Les détails concernant les procédures de transformation SOLAR-VHDL seront donnés en Annexe B.

4.3.3.2 *Configuration Globale*

Le modèle de fichier VHDL de la description de la configuration globale d'un circuit est donné par la figure 4.14.

Dans cette représentation, en plus des deux blocs essentiels, on retrouve le bloc de synchronisation, déclaré comme composant, avec en sortie les signaux d'horloges pour chaque partie constituant le circuit.

4.3.3.3 *Partie Opérative*

La partie opérative est une structure d'interconnexion des composants de la bibliothèque prédéfinie. Elle est caractérisée par le nombre de portes de transfert et le nombre de lignes de connexion.

```
1   library mvl_7;
2   use mvl_7.Types.all;
3   use mvl_7.arithmetic.all;
4   entity nom_du_circuit_circuit is
5       port(
6           clk      : in bit;
7           reset    : in bit;
8           ...);-Tous les ports declares dans le fichier SOLAR
9   end nom_du_circuit_circuit;
10  architecture Structure of nom_du_circuit_circuit is
11      -Declaration de tous les signaux internes
12      component nom_du_circuit_control
13          port(
14                  -signaux globaux
15                  -signaux declares dans le fichier SOLAR);
16      end component;
17      component nom_du_circuit_datapath
18          port(
19                  -signaux globaux
20                  -signaux declares dans le fichier SOLAR);
21      end component;
22      Component Clock
23          port(
24                  -signaux globaux);
25      end component;
26  begin
27      ControlPart : nom_du_circuit_control
28          port map(...);
29      DataPart : nom_du_circuit_datapath
30          port map(...);
31      Clockx: Clock
32          port map(...);
33  end Structure;
34
35  configuration cfg_circuit of nom_du_circuit_circuit is
36      for Abstract_Architecture
37      end for;
38  end cfg_circuit;
```

Figure 4.14: *Aperçu de la description VHDL de la configuration générée par PAT.*

Dans le fichier VHDL de la partie opérative (figure 4.15), chaque composant est déclaré par le mot clé *Component* et par la configuration qui spécifie le couple entité/architecture correspondant à sa description.

4.3.3.4 *Partie Contrôle*

La partie contrôle est un processus décrivant des opérations qui sont elles mêmes organisées en plusieurs sous-opérations exécutées en parallèle.

La partie contrôle générée par PAT est organisée en deux processus VHDL:

```
1   library mvl_7;
2   use mvl_7.Types.all;
3   use mvl_7.arithmetic.all;
4   entity nom_du_circuit_datapath is
5       port( -signaux externes);
6   end nom_du_circuit_datapath;
7   architecture RT-Level of nom_du_circuit_datapath is
8       -declaration de signaux
9       component nom_du_composant
10              port(- Declaration des ports du composant);
11      end component;
12              ... -declaration des autres composants
13  begin
14      nom_du_composant: nom_du_composant
15              port map(- Declaration des ports du composant);
16              ... -occurrence des autres composants et leurs port map
17  end RT-Level;
```

Figure 4.15: *Aperçu du fichier VHDL de la partie opérative générée par PAT.*

- Le processus principal contient une seule instruction, *Case*, et les branchements de cette instruction correspondent aux différentes transitions (ou macrocycles). Un type énuméré est déclaré de type *State*, qui contient une liste des noms de tous les états existants. Deux variables de ce type sont aussi déclarées, *CURRENT_STATE* et *NEXT_STATE*. La première donne l'identité de l'état courant, et la deuxième donne l'état suivant. Celle-ci est initialisée au premier état (*NEXT_STATE* <= Etat initial). Le processus principal a une liste de sensibilité contenant tous les signaux déclarés en entrée, les signaux de compte-rendus et le signal *CURRENT_STATE*. L'instruction *Case* a comme condition, la valeur du *CURRENT_STATE*. Une condition de l'instruction *Case* a la forme montrée dans la figure 4.16.

```
1   case CURRENT_STATE is
2       when (S1) =>
3           if (Condition) then
4               -Generation d'un nouveau vecteur
5               NEXTSTATE <= S2;
6           end if
7           ...
8   end case;
```

Figure 4.16: *Format de représentation de l'instruction "*case*".*

Au départ de chaque boucle du processus, tous les éléments du vecteur sont remis à '0'. Ils sont supposés être actifs à '1'; si la situation inverse se présente,

elle doit être spécifiée à l'aide du fichier global. Ainsi, pour chaque nouveau vecteur, il suffit de changer seulement les signaux de commande des composants à activer.

- Le deuxième processus sert à synchroniser les transitions ou la génération des vecteurs. Ce processus est toujours le même et il est présenté dans la figure 4.17.

```
1   SYNCH: process (cont_clk, cont_reset, CURRENT_STATE)
2   begin
3       if (cont_reset = '0') then
4           –Les conditions de reset
5           CURRENT_STATE <= S1;    –Premier etat
6       elsif (clk = '1' and clk'event) then
7           CURRENT_STATE <= NEXTSTATE;
8       end if;
9   end process SYNCH;
```

Figure 4.17: *Processus de synchronisation de la partie contrôle.*

Ce processus a une liste de sensibilité contenant l'horloge de la partie contrôle et tous les signaux de remise à zéro. En mode de fonctionnement normal, l'affectation de l'état courant avec la valeur de l'état suivant est faite à chaque front montant de l'horloge. Cette affectation redéclenche le processus principal, car *CURRENT_STATE* fait partie de sa liste de sensibilité.

Le fichier VHDL de la partie contrôle de l'exemple du pgcd est donné dans la figure 4.18.

```
1   library mvl_7;
2   use mvl_7.Types.all;
3   use mvl_7.arithmetic.all;
4   entity nom_du_circuit_control is
5       port( –les signaux externes );
6   end nom_du_circuit_control;
7   architecture RT-Level of nom_du_circuit_control is
8   type State is ( –liste d'etats possible);
9   signal CURRENT_STATE, NEXTSTATE : State;
10  begin
11
12      Controleur : process (–liste des ports d'entrees, CURRENT_STATE)
13      begin
14          –initialiser les signaux du vecteur
15          NEXTSTATE <= S1; –premier etat
16          case CURRENT_STATE is
17              –corps du processus
18          end case;
19      end process Controleur;
```

```
20
21        SYNCH: process(cont_clk, cont_reset, CURRENT_STATE)
22        begin
23            –corps du processus
24        end process SYNCH;
25    end RT-Level;
```

Figure 4.18: *Aperçu du fichier VHDL de la partie contrôle générée par PAT.*

Dans la table 4.1, les tailles des programmes, en nombre de lignes de code, pour certains exemples sont présentées. Le code VHDL comportemental est utilisé en entrée d'AMICAL, le code SOLAR RTL est celui généré par AMICAL et le code VHDL RTL est celui généré par PAT.

Le code VHDL généré par AMICAL et PAT est environ vingt fois plus grand que le code VHDL comportemental correspondant. Cette différence de taille est une des raisons principales de l'utilité des outils de synthèse de haut niveau.

Exemples	VHDL comport.	SOLAR RTL	VHDL RTL
pgcd	42	949	779
Tri à bulles	92	2220	1817
Opérateur multi-fonctions	154	2629	2072
Répondeur téléphonique	151	4734	3656
Gestion de mémoire	739	23712	19846

Table 4.1: *Comparaison des tailles des programmes SOLAR et VHDL.*

4.4 Validation

Les outils de synthèse sont basés sur des algorithmes et sont des programmes qui dans un premier temps peuvent contenir des "*bugs*". L'utilisation de méthodes de vérification dans les systèmes de synthèse est donc nécessaire [57]. Il existe deux méthodes de vérification de base qui sont des approches complémentaires: la première utilise les méthodes formelles à l'intérieur du système de synthèse et la seconde utilise la validation par simulation.

4.4.1 Simulation

La simulation est utilisée, en général, pour vérifier si les étapes de conception d'un circuit ont été exécutées correctement [76]. En utilisant la synthèse, les circuits réalisés sont *corrects par construction*. Alors, qu'est ce que la simulation peut apporter de plus dans ce cas?

Indépendamment de la stratégie de synthèse (automatique ou interactive) et indépendamment des méthodes de vérification analytiques (formelles), la spécification d'entrée doit être validée par la simulation.

L'applicabilité de la simulation aux niveaux d'abstraction inférieurs peut se résumer par trois points:

- En utilisant des algorithmes de synthèse automatiques, il y a des *doutes* à ce que les circuits générés soient *corrects*. L'utilisation de la simulation à ce niveau est nécessaire puisque la synthèse ne garantit pas à tous les coups "*l'exactitude*" de la spécification initiale.

- Comme tout logiciel, les systèmes de synthèse ne sont pas dépourvus d'erreurs de programmation ("*bugs*"). Donc, une simulation au niveau inférieur (à la sortie) est aussi nécessaire pour la vérification des résultats de la synthèse.

- La transformation de la description de haut niveau en une description de bas niveau n'est pas unique. Les résultats de synthèse d'un point de vue temporel

peuvent largement varier. Soit ces caractéristiques ne sont pas spécifiées au départ, soit leur spécification est incomplète. Dans ce cas, une simulation à ce niveau est nécessaire pour les vérifications temporelles.

Pour conclure, on peut mentionner que la simulation est nécessaire dans le contexte de la synthèse de haut niveau. Elle doit se faire aux deux différents niveaux d'abstraction de cette synthèse (niveau comportemental et niveau transfert de registres).

Les outils de simulation VHDL utilisés sont ceux qui supportent quasi tout le langage VHDL et ce pour tous les niveaux d'abstraction pour lesquels VHDL peut être utilisé. Dans notre cas, pour faciliter la comparaison des résultats de simulation aux deux niveaux d'abstraction, le même vecteur de test (*stimuli*) est utilisé [76].

4.4.2 Exemple

Le système AMICAL a été testé sur plusieurs exemples. L'exemple qui va être utilisé dans cette section pour illustrer la validation est celui qui calcule le plus grand diviseur commun de deux entiers (*pgcd*). Sa description comportementale est montrée dans la figure 4.19.

La simulation d'une telle description est donnée dans la figure 4.20. Deux séries d'entrées ont été utilisées; la première est le couple d'entiers (20, 15) et la deuxième est le couple d'entiers (34, 12). Les résultats sont évidemment, 5 et 2 respectivement. Il est à noter que les deux résultats de simulation au niveau comportemental sont disponibles en sortie juste après que les valeurs d'entrées soient introduites.

Après la synthèse faite par AMICAL et la modification de l'architecture abstraite générée en architecture détaillée où les signaux de synchronisation ont été rajoutés par PAT, le résultat de la simulation au niveau RTL est donné par la figure 4.21.

Contrairement à la simulation comportementale, les résultats ne sont pas valables en sortie immédiatement après que les valeurs d'entrées soient introduites, ils sont

```
1   entity pgcd is
2       port (
3           start: in BIT
4           din  : in BIT;
5           xi, yi in INTEGER;
6           dout: out BIT;
7           ou: out INTEGER);
8   end pgcd;
9   architecture Behavior of pgcd is
10  begin
11      pgcd_file: PROCESS
12      variable x, y : INTEGER;
13      begin
14          if ( start /= '1' ) then
15              wait until ( start = '1' );
16          end if;
17          wait until ( din = '1' );
18              dout <= '0';
19              x := xi ;
20              y := yi ;
21              while ( x /= y ) loop
22                  if ( x < y ) then
23                      y := y - x ;
24                  else x := x - y;
25                  end if;
26              end loop;
27              dout <= '1';
28              ou <= x ;
29      end PROCESS pgcd_file;
30  end Behavior;
```

Figure 4.19: *Spécification d'entrée pour l'exemple du pgcd.*

décalés dans le temps. Ceci est dû au temps d'exécution pris par chaque unité fonctionnelle et au nombre de transitions effectuées. De plus, on remarque que le nombre de cycles (période d'horloge) pris pour le calcul du pgcd du premier couple d'entiers n'est pas égal à celui du second couple. Pour le premier cas, le temps de calcul est de 10 cycles alors que pour le second cas ce temps est de 18 cycles. Le nombre d'itérations exécutées pour le deuxième calcul est supérieur à celui du

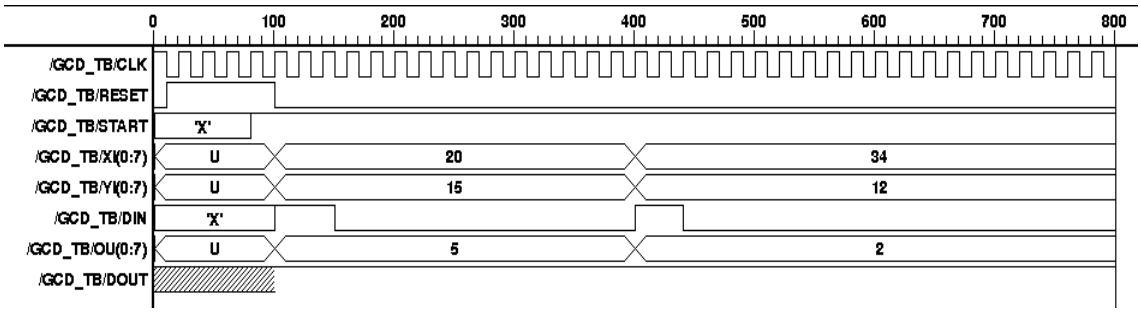


Figure 4.20: *Simulation comportementale.*

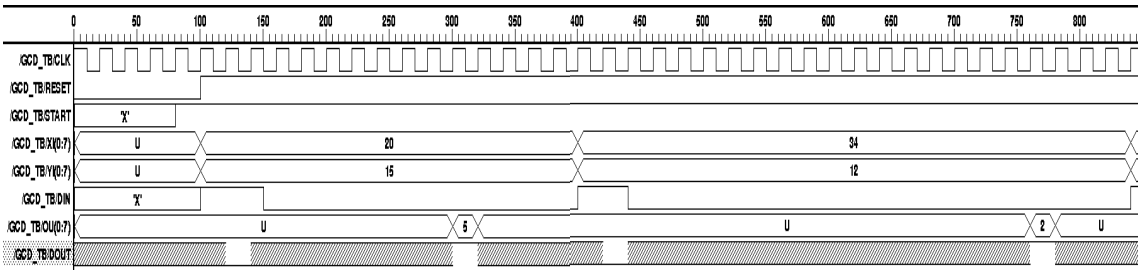


Figure 4.21: *Simulation au niveau transfert de registres.*

premier, ce qui explique la différence obtenue.

Les deux simulations ont été exécutées en utilisant le simulateur VHDL de Synopsys [83].

Cette méthode de simulation est un moyen très rapide pour détecter les erreurs de la synthèse pour un jeu de test donné. Malheureusement, on ne peut dire que la fonctionnalité du circuit est correcte. Dans le cas général, le grand problème qui reste est de vérifier que la description comportementale est équivalente à la description RTL.

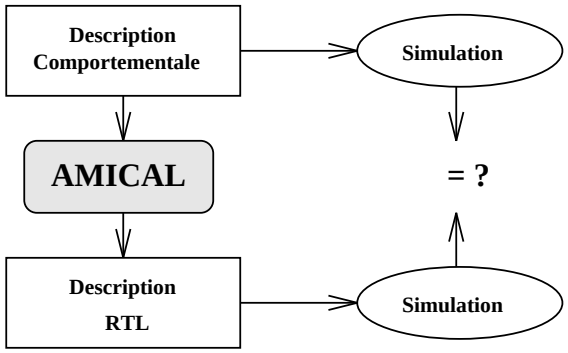


Figure 4.22: *Equivalence entre les simulations comportementale et RTL.*

4.5 Conclusion

Le but de ce chapitre était de trouver un lien entre l'outil de synthèse de haut niveau, AMICAL, et les outils de simulation et de synthèse logique existants, et ce, en transformant la structure abstraite générée en une architecture détaillée. Cette architecture est obtenue après personnalisation de l'architecture générée par l'ajout des signaux globaux tels que les signaux d'horloge, les signaux de remise à zéro, etc. et en créant une bibliothèque VHDL compatible avec les sous-ensembles utilisés par ces outils.
Les descriptions dans la synthèse de haut niveau permettent la spécification de circuits complexes. Mais parfois, la définition d'informations telles que la synchronisation et les signaux globaux n'est pas implicite. C'est pour cela que ces derniers doivent être ajoutés à la sortie de l'outil de synthèse de comportement, AMICAL.

Dans ce chapitre, PAT (Programmable Architecture Translator), qui permet la personnalisation de l'architecture abstraite et la traduction des descriptions de sortie d'AMICAL données en SOLAR en leur équivalent en VHDL, a été présenté. La validation du bon fonctionnement de l'outil, en comparant les simulations de deux spécifications, l'une comportementale en entrée d'AMICAL et l'autre, structurelle à la sortie de PAT, a été détaillée. Par comparaison, les deux simulations donnent les mêmes résultats, mais le problème majeur qui reste à résoudre est de prouver que la description comportementale est équivalente à la description structurelle donnée au niveau transfert de registres.

CHAPITRE 5

Etude comparative de plusieurs modèles architecturaux

Chapitre 5

Etude comparative de plusieurs modèles architecturaux

Ce chapitre présente une étude comparative de plusieurs modèles architecturaux générés par AMICAL. Trois exemples ont été pris pour valider cette étude: le plus grand commun diviseur (pgcd), le tri à bulles et un opérateur arithmétique multi-fonctions. Tous trois ont été utilisés pour la génération de chacun des modèles architecturaux, et ce, en utilisant les différents modèles de synchronisation (section 3.4). Dans cette étude, les résultats correspondant à chaque cas seront présentés et commentés.

5.1 Introduction

Afin de couvrir les besoins de la conception, AMICAL génère trois différents types de configuration. Parmi les configurations disponibles, la plus simple est celle composée d'une partie opérative et d'une partie contrôle. Un modèle avec une interface de mémorisation entre les deux parties principales: chemin de données et contrôleur, permet d'obtenir une architecture où les majeures fonctions synchrones sont regroupées dans le bloc d'interface. Le premier type d'architecture correspond

au cas général de tout processeur; le deuxième modèle est utilisé pour *pipeliner* le mode d'exécution des événements, il donne lieu à une architecture adaptée au test de type "*scan-path*". Le troisième modèle architectural est utilisé pour la génération de circuits auto-testables [51].

Dans ce chapitre, quelques résultats, grâce auxquels sera réalisée la comparaison des performances de chaque modèle, seront donnés. Le chapitre est organisé suivant le plan ci-dessous:

- La section suivante présente les critères qui seront utilisés pour la comparaison des performances.
- La section 3 traite des différents exemples sur lesquels l'étude a été basée.
- Les résultats et commentaires correspondants seront quant à eux détaillés dans la section 4.
- Finalement, la section 5 résumera ce chapitre par quelques conclusions.

5.2 Critères de comparaison

Pour chaque circuit généré, les performances sont évaluées à partir de sa surface, sa latence et sa vitesse exprimée par son temps de cycle d'horloge.

5.2.1 Surface

La surface des circuits générés par AMICAL est évaluée après la synthèse logique par Synopsys [82]. Synopsys offre une estimation de son résultat de synthèse. Ces estimations seront faites en ne prenant en compte que la taille des portes qui constituent le circuit. Les interconnexions ne sont pas considérées dans cette étude puisque le facteur de routage peut varier de beaucoup d'un outil de placement et routage à un autre. Cette surface est donnée en μm^2.

5.2.2 Vitesse

La vitesse d'un circuit (exprimée aussi en temps total d'exécution) est définie par le produit du nombre de cycles d'exécution en simulation par le temps que peut prendre chaque cycle.

$$T_t = T_c * Nb_c$$

où

T_t représente le temps total d'exécution,
Nb_c représente le nombre de cycles,
T_c représente le temps de cycle.

5.2.2.1 *Nombre de cycles d'exécution*

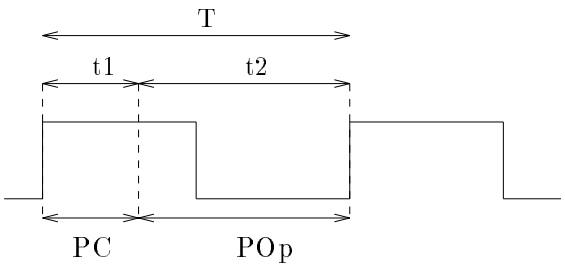
Le nombre de cycles d'exécution est calculé d'après la simulation VHDL effectuée à l'aide de l'outil de simulation de Synopsys [83] pour un vecteur de test donné. La simulation s'effectue sur les fichiers de sortie d'AMICAL, correspondant au niveau transfert de registres. Un cycle étant égal à une période d'horloge, le nombre de

cycles d'exécution équivaut au nombre de périodes écoulées entre l'instant de la prise en compte du vecteur d'entrée (début de calcul déclenché, quand les valeurs d'entrée sont présentes, par l'activation d'un signal de sélection) et l'instant où les résultats sont présents en sortie. La latence d'un circuit peut varier suivant les vecteurs d'entrée. Ceci arrive dans le cas où la description comportementale contient des boucles qui dépendent des données d'entrée.

5.2.2.2 *Temps de cycle*

Comme mentionné dans le Chapitre 3, le temps de cycle correspond à la période d'horloge minimale pour le bon fonctionnement du circuit.

Pour les architectures générées d'après le modèle sans "*pipeline*" (ce modèle sera par la suite référencé comme le modèle *Pipe_0*), le temps de cycle doit être supérieur ou égal à la somme du chemin critique de la partie contrôle et du temps d'exécution de la partie opéartive (figure 5.1, voir aussi section 3.4.4).



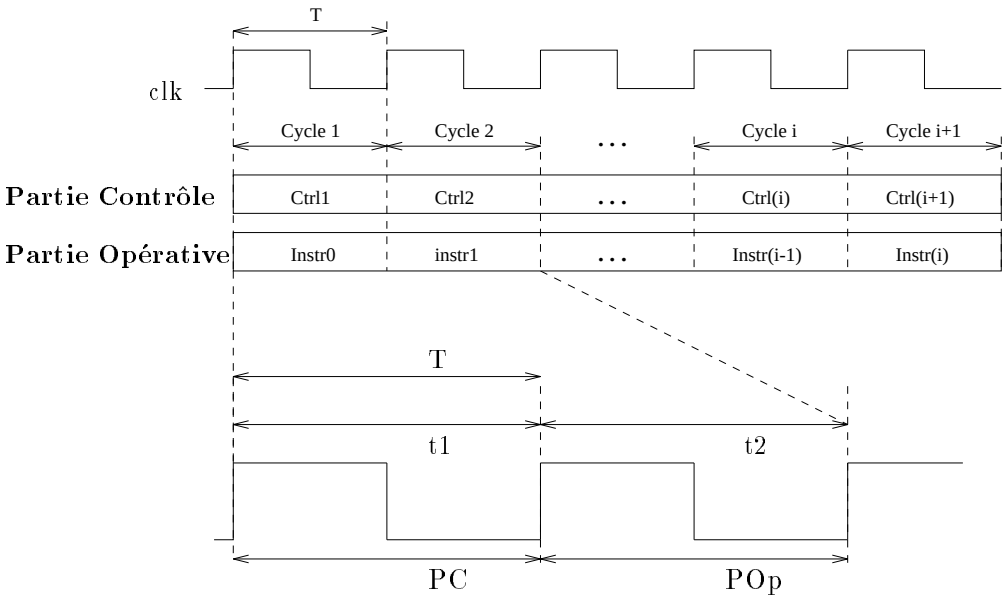
$T = \max(t1 + t2)$

t1 et t2 s'applique à la même transition.

Figure 5.1: *Temps de cycle minimal pour les modèles non pipelinés.*

Pour les architectures générées d'après le modèle avec *pipeline*, le temps de cycle est égal au maximum des deux temps: le chemin critique de la partie contrôle et le temps d'exécution de la partie opérative (figure 5.2). Dans ce cas là, quand la partie contrôle calcule le vecteur de commande pour le cycle prochain, la partie opérative est en train d'effectuer les opérations correspondant au vecteur de commande du

cycle précédent.



$$T = \max(\max(t1), \max(t2))$$

Figure 5.2: *Temps de cycle pour les modèles pipelinés.*

L'évaluation du chemin critique de la partie contrôle est obtenue après la synthèse logique effectuée par Synopsys [82].

Le temps du cycle d'exécution de la partie opérative est égal au délai maximal requis pour l'exécution complète des opérations de chaque étape de contrôle pendant un cycle. AMICAL génère un chemin de données basé sur des opérations exécutées sur des opérandes provenant de registres (ou de l'extérieur) et dont les résultats sont stockés dans des registres (ou mis en sortie vers l'extérieur). Selon ce modèle d'architecture, ce temps peut être approximé par la somme du délai d'"*exécution*" d'un registre (lecture et écriture), du délai maximal de calcul d'une unité fonctionnelle et de quatre fois le temps de propagation d'un connecteur. Le schéma de la figure 5.3 illustre ce phénoméne.

Dans ce cas, on peut approximer cette valeur par:

$$T_{exe\_Pop} = D_{Reg} + Max(D_{FU}) + 4 * T_{Sw}$$

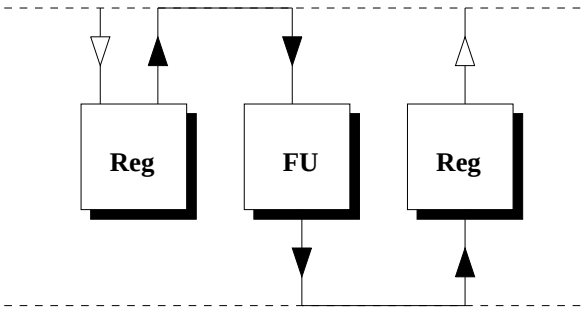


Figure 5.3: *Calcul du temps d'exécution de la partie opérative.*

où

T_{exe_Pop}: Temps d'exécution de la partie opérative,
D_{Reg}: Délai d'"*exécution*" d'un registre
(temps nécessaire pour une lecture et une écriture)
D_{FU}: Délai d'exécution d'une unité fonctionnelle,
T_{Sw}: Temps de propagation d'un connecteur.

Tous ces délais ont été estimés par l'outil de synthèse logique Synopsys [82].

5.3 Définition des exemples

5.3.1 *Le pgcd*

Le pgcd applique l'algorithme de calcul du plus grand commun diviseur de deux nombres entiers. Cet exemple est pris du *High-Level Synthesis Workshop Benchmark*. La description a été modifiée et adaptée au sous-ensemble de VHDL accepté par AMICAL tout en respectant l'algorithme. L'algorithme de calcul est montré dans la figure 5.4. La description comportementale correspondante est présentée dans la figure 4.19 de la section 4.4.2.

pgcd(X, Y)	=	pgcd(X-Y, Y)	Si	(X > Y)
pgcd(X, Y)	=	pgcd(X, Y-X)	Si	(X < Y)
pgcd(X, Y)	=	X = Y	Si	(X = Y)

Figure 5.4: *Algorithme du pgcd.*

5.3.2 *Le tri à bulles*

Cet algorithme ordonne dans l'ordre croissant un ensemble de nombres entiers. Dans cette étude, le nombre d'éléments à ordonner est de quatre, même si ce nombre peut être étendu à n'importe quelle valeur. La description VHDL du tri à bulles est donnée dans la figure 2.4 de la section 2.2.2.

5.3.3 *L' Opérateur multi-fonctions*

Cet opérateur est une unité fonctionnelle regroupant quatre fonctions arithmétiques: l'addition, la soustraction, la multiplication et la réciproque (calcul de l'inverse). Ces opérations s'exécutent sur des nombres à virgule fixe, codés sur 32 bits. Elles se partagent une UAL (additionneur-soustracteur). Les trois premières opérations sont des fonctions à deux opérandes, seule la réciproque n'a qu'une opérande. Le programme décrivant l'algorithme de l'opérateur multi-fonctions est donné dans l'Annexe A.

5.4 Résultats

Pour la validation des différentes architectures générées par AMICAL et des différents modèles de synchronisation, les trois exemples mentionnés plus haut ont été utilisés.

La figure 5.5 représente un plan de validation en donnant les différentes architectures générées par AMICAL et les différents modèles de synchronisation utilisés pour chaque type d'architecture.

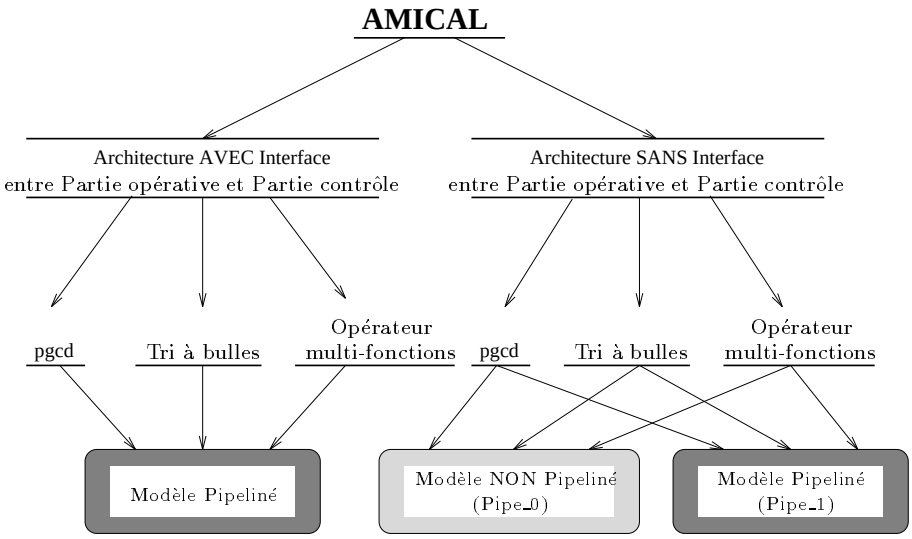


Figure 5.5: *Représentation du plan de validation.*

Dans la figure 5.5, on retrouve les deux types d'architectures: l'architecture avec interface entre la partie opérative et la partie contrôle, et l'architecture sans interface. Ce dernier modèle a été utilisé pour une architecture générale sans *pipeline* (*pipe_0*, horloge sans recouvrement) et une architecture avec le modèle de synchronisation *pipeliné* de niveau 1 (*pipe_1*).

Il est à noter que l'architecture avec interface n'est utilisée qu'avec les modèles pipelinés. Par contre, l'architecture sans interface a été utilisée pour les deux types de modèles (*pipeliné* ou *non-pipeliné*).

Chaque exemple est testé sur chacun des deux types d'architecture et chacun des deux modèles de synchronisation.

Pour chaque exemple, on donnera le type d'architecture, le modèle de synchronisation, le nombre de cycles d'exécution obtenu par simulation au niveau transfert de registres, pour des vecteurs de test donnés, et la taille du circuit après synthèse logique (seule la surface des portes est considérée sans estimation du routage) (table 5.1, table 5.2 et table 5.3).

La première ligne donne, en terme de nombre de cycles, le temps de calcul écoulépour l'exécution de chaque exemple.Ce nombre est obtenu après une simulation au niveau transfert de registres. Ce temps est dépendant des vecteurs de test utilisés. La deuxième ligne indique la taille du circuit après la synthèse logique par SYNOPSYS [82] pour deux technologies différentes: 0,5 μ et 1,2 μ. Cette taille n'inclut pas les connexions; la taille totale peut être estimée au double de la taille mentionnée par le tableau.

pgcd							
		Pipe_0		**Pipe_1**		**Interface**	
Nbre de cycles RTL		5	9	10	18	10	18
Taille (μm^2)	**Tech.** 1,2	402589,4375		478573,8750		502809,1250	
	Tech. 0,5	151965,0000		181665,0000		189475,0000	

Table 5.1: *Résultats du pgcd.*

Pour l'exemple du pgcd (table 5.1), chaque modèle possède deux cases. Elles correspondent à des vecteurs de test différents. Les deux paires de données utilisées sont (20, 15) et (34, 12) respectivement. Au niveau comportemental, le nombre de cycles est nul (il n'est pas représenté dans le tableau). Par contre, le nombre de cycles au niveau transfert de registres (RTL) varie d'un couple de données à un autre et d'un modèle architectural à un autre (*pipeliné* ou *non-pipeliné*). Dans ce premier exemple, les résultats obtenus pour les deux vecteurs d'entrée sont différents à cause du nombre supérieur d'opérations de soustraction à exécuter dans le deuxième cas

par rapport au premier. Dans le cas des architectures avec *pipeline*, le nombre de cycles au niveau transfert de registres est supérieur à celui de l'architecture *Pipe_0* car des états intermédiaires sont introduits dans le graphe de contrôle pour effectuer la comparaison entre x et y (lignes 21 à 26 de la description du *pgcd* de la section 4.5.2). Des états additionnels sont nécessaires à cause de la dépendance de données entre éléments de comparaison et éléments d'affectation (comparaison entre x et y et (affectation de x ou affectation de y)).

En ce qui concerne les surfaces, elles sont plus importantes dans les modèles *pipelinés* que les modèles *non-pipelinés*, et cela est dû au fait que les architectures des modèles *pipelinés* possèdent un surcroît de registres nécessaire à la mémorisation des signaux pour la synchronisation entre contrôleur et chemin de données. De plus dans le cas des modèles *pipelinés* du pgcd, le graphe de contrôle est plus grand: plus grand nombre d'états et de transitions.

La table 5.2 résume les résultats obtenus pour l'exemple du tri à bulles.

Tri à bulles				
		Pipe_0	**Pipe_1**	**Interface**
Nbre de cycles RTL		54	82	82
Taille (μm^2)	**Tech.** 1,2	1192799,7500	1362160,5000	1499731,7500
	Tech. 0,5	441265,0000	513865,0000	548625,0000

Table 5.2: *Résultats du tri à bulles.*

Comme mentionné précédemment, la simulation s'est effectuée pour un tri sur quatre entiers (20, 55, 46, 11). L'ordre initial de ces quatre entiers est un facteur déterminant dans le nombre de cycles pour la simulation RTL. Pour les trois architectures, les mêmes vecteurs ont été utilisés.

De même, pour l'opérateur multi-fonctions, (table 5.3), les quatre valeurs données pour les nombres de cycles correspondent aux résultats des quatre différentes opérations effectuées par l'opérateur (la multiplication, la réciproque, l'addition et la

soustraction) pour différentes données.

Opérateur multi-fonctions													
		Pipe_0				**Pipe_1**				**Interface**			
Nbre de cycles RTL		63	82	2	2	90	124	3	3	90	124	3	3
Taille (μm^2)	**Tech.** 1,2	4225338,0000				4401538,5000				4530415,5000			
	Tech. 0,5	1495835,0000				1580755,0000				1617385,0000			

Table 5.3: *Résultats de l'opérateur multi-fonctions.*

Dans ces trois tableaux, on remarque que le nombre de cycles pour exécuter un algorithme varie d'un modèle de synchronisation à un autre et non pas d'une architecture à une autre (architecture avec/sans interface de mémorisation entre la partie opérative et la partie contrôle). Ceci est dû au type de *pipe* (parallélisme) utilisé par le modèle de synchronisation et au nombre de cycles consommés par une étape de contrôle.

Dans tous les cas, le nombre de cycles dans le modèle représenté par *Pipe_0* (voir section 3.4.4) est inférieur au nombre de cycles du modèle *Pipe_1*. Dans le premier cas, la partie opérative et la partie contrôle exécutent leurs opérations pendant le même cycle, mais sans recouvrement (une étape de contrôle prend un cycle). Par contre, dans le deuxième cas, les deux parties travaillent simultanément (en parallèle) mais une étape de contrôle prend deux cycles. Il faut noter que pour *Pipe_0*, le cycle de base sera plus long pour permettre aux deux parties constituant le circuit d'exécuter leurs opérations. Ce temps de cycle est calculé par le chemin critique des deux parties (voir section 5.2.2.2).

La table 5.4 montre les différents temps de cycle pour chaque modèle. Ce temps est exprimé en *ns* (nanosecondes). Cette étude est faite sur deux technologies différentes: la première à 1,2 μ avec une alimentation classique de 5V et la deuxième à 0,5 μ avec une alimentation de 3,3V. Dans cette table, on remarque que, pour les

	pgcd			Tri à bulles			Opérateur multi-fonctions		
Technologie 1,2 μm									
	PC	**POp**	**T_C**	**PC**	**POp**	**T_C**	**PC**	**POp**	**T_C**
Pipe_0	5,08	7,41	12,49	9,34	8,57	17,91	5,8	28,08	33,88
Pipe_1	4,11	12,12	12,12	9,25	8,09	9,25	5,17	26,99	26,99
Interface	4,11	7,12	7,12	9,01	3,09	9,01	5,00	13,56	13,56
Technologie 0,5 μm									
Pipe_0	4,80	9,66	14,46	11,20	12,14	33,34	7,00	34,56	41,56
Pipe_1	12,47	14,41	14,41	11,48	9,66	11,48	6,38	34,19	34,19
Interface	4,98	10,50	10,50	11,38	4,19	11,38	6,38	14,74	14,74

Table 5.4: *Comparaison des temps de cycle.*

plus grands exemples, le temps de cycle des architectures avec le modèle de communication avec interface est toujours inférieur à celui des modèles sans interface.

La table 5.5 compare les temps d'exécution totaux pour les différents exemples et architectures. Il est intéressant de noter que le temps total d'exécution dépend de la

	pgcd			Tri à bulles			Opérateur multi-fonctions		
Technologie 1,2 μm									
	Nb_c	t_c	T_t	Nb_c	t_c	T_t	Nb_c	t_c	T_t
Pipe_0	5	12,49	62,45	54	17,91	967,14	63	33,88	2134,44
Pipe_1	10	12,12	121,20	82	9,25	758,50	90	26,99	2429,10
Interface	10	7,12	71,20	82	9,01	738,82	90	13,56	1220,40
Technologie 0,5 μm									
Pipe_0	5	14,46	72,30	54	33,34	1800,82	63	41,56	2618,28
Pipe_1	10	14,41	144,10	82	11,48	941,36	90	34,19	3077,10
Interface	10	10,50	105,00	82	11,38	933,16	90	14,74	1326,60

Table 5.5: *Comparaison des temps totaux.*

complexité de l'exemple et de l'architecture. Dans le cas du pgcd, le meilleur temps est obtenu par une architecture *non pipeliné*, tandis que dans le cas de l'opérateur

multi-fonctions, le meilleur temps est donné par une architecture avec interface.

La table 5.6 résume les résultats trouvés en ne mettant en évidence que les deux criteres objectifs: la surface et la vitesse (exprimée par le temps total d'exécution).

	pgcd		**Tri à bulles**		**Opérateur m-f**	
Technologie 1,2 μ						
	T_t	**Surface**	T_t	**Surface**	T_t	**Surface**
Pipe_0	62,45	402589,44	967,14	1192799,75	2134,44	4225338,00
Pipe_1	121,20	478573,87	758,50	1362160,50	2429,10	4401538,50
Interface	71,20	502809,12	738,82	1499731,75	1220,40	4530415,50
Technologie 0,5 μ						
	T_t	**Surface**	T_t	**Surface**	T_t	**Surface**
Pipe_0	72,30	151965,0000	1800,82	441265,0000	2618,28	1495835,0000
Pipe_1	144,10	181665,0000	941,36	513865,0000	3077,10	1580755,0000
Interface	105,00	189475,0000	933,16	548625,0000	1326,60	1617385,0000

Table 5.6: *Tableau récapitulatif.*

5.5 Conclusion

Dans ce chapitre une étude comparative des différents modèles de synchronisation et des différentes architectures générées par AMICAL a été présentée. Quelques critères de comparaison ont été appliqués sur trois exemples différents: le pgcd, le tri à bulles et l'opérateur multi-fonctions. Cette étude a montré que pour les modèles *pipelinés* (avec ou sans interface), la taille des circuits est supérieure à celle des modèles *non pipelinés*, de même que le nombre de cycles d'exécution au niveau transfert de registres est supérieur dans le cas des modèles *pipelinés*. Par contre, le temps de cycle est plus important pour les modèles *non-pipelinés*. Quant au temps total d'exécution, il dépend de l'exemple traité et de l'architecture.

CHAPITRE 6

Conclusion

Chapitre 6

Conclusion

Le but de cette thèse était de développer les différents modèles architecturaux générés par AMICAL et de faire le lien entre AMICAL et les outils de conception au niveau transfert de registres. Ce lien est développé par la personnalisation de l'architecture abstraite de base en utilisant PAT. Il fallait aussi traduire la sortie générée, en SOLAR, en son équivalent VHDL et adapter la sortie en développant différents modèles de synchronisation. Une des tâches accomplies a été de tester le système AMICAL à travers la validation de l'entrée comportementale et la sortie structurelle par les outils de simulation existants.

Dans le Chapitre 2, une vue globale du système AMICAL a été présentée. Partant d'une description comportementale donnée en VHDL et d'une bibliothèque externe d'unités fonctionnelles, AMICAL génère une architecture abstraite sous une forme intermédiaire nommée SOLAR compatible avec les outils de simulation et de synthèse au niveau transfert de registres. Dans ce chapitre, l'entrée d'AMICAL, ses étapes de synthèse, sa sortie ainsi que ses modèles d'exécution générés par ses

algorithmes de synthèse ont été présentés dans l'ensemble.

Le Chapitre 3 a présenté les différents modèles architecturaux générés par AMICAL. La génération de l'architecture détaillée et l'ajout des éléments de synchronisation à l'architecture abstraite ont été introduits dans ce chapitre ainsi que des exemples de modèles de synchronisation adaptés pour le système AMICAL. Finalement un exemple de bibliothèque de composants a été présenté pour montrer le modèle de composition et d'exécution des architectures générées.

PAT (Programmable Architecture Translator), a été présenté en détail dans le Chapitre 4. C'est un outil d'aide à la personnalisation et la traduction de la sortie d'AMICAL du langage SOLAR, en son équivalent VHDL. Il permet le lien d'AMICAL avec les outils de conception au niveau transfert de registres. Dans ce chapitre, ont été présentés aussi, le fichier global grâce auquel l'ajout des signaux globaux est autorisé ainsi que les formats généraux des fichiers SOLAR et VHDL.

Le Chapitre 5 a montré une étude comparative de plusieurs modèles architecturaux pour tester le système AMICAL. Cette étude a été faite sur chacun des modèles architecturaux et sur chacun des modèles de synchronisation en utilisant trois exemples différents. Cette étude nous a conduit à avoir besoin de plusieurs modèles architecturaux pour différentes applications.

Si le lien entre la synthèse de haut niveau et la synthèse au niveau transfert de registres semble être plus ou moins résolu du point de vue conceptuel, il reste le problème de la validation. Il s'agit de s'assurer que l'architecture générée au niveau transfert de registres correspond à la description comportementale d'entrée.
Plusieurs voies doivent être explorées. La plus pragmatique de ces voies consiste à comparer des résultats des simulations comportementales à ceux des simulations au niveau transfert de registres. Il faut noter que cette solution est facile à réaliser, mais elle ne permet qu'une vérification partielle.
La plus sûre de ces voies est la vérification formelle. Mais, dans le cas des descriptions comportementales, cette méthode rend le problème plus proche de la vérification de programmes que de la vérification de circuits.

Bibliographie

Bibliographie

[1] T. Abbassi, S. Ben Atallah, "Modélisation multi-niveaux de systèmes VLSI dans le langage VHDL", *Projet de fin d'études*, TIMA/INPG, Juillet 1992.

[2] R. Airiau, J.M. Bergé, V. Olive, J. Rouillard, "VHDL Du langage à la modélisation" *Presses Polytechniques et Universités Romandes et CNET-ENST*, 1990.

[3] M. Aichouchi, H. Ding, K. O'Brien, A.A. Jerraya, "Generation and Validation of detailed Architectures from behavioral VHDL Descriptions", *5 ème ICM (International Conference on Micrœlectronics)*, Arabie Saoudite, Décembre 1993.

[4] M. Aichouchi, K. O'Brien, A.A. Jerraya, "Generation and Validation of detailed Architectures from behavioral VHDL Descriptions", *AJSE (Arabian Journal for Sciences and Engineering)*, Version étendue, à paraître en Octobre 1994.

[5] M. Aichouchi, P. Kission, H. Ding, P. Vijay Raghavan, A.A. Jerraya, "Lien entre la Synthèse Architecturale et la Synthèse au Niveau Transfert de Registres", soumis à *TSI (Technique et Science Informatiques)*.

[6] M. Aichouchi, P. Vijay Raghavan, H. Ding, A.A. Jerraya, "Linking High-Level Synthesis And Register Transfert Level tools Using VHDL", soumis à *Revue Internationale Algérienne des Technologies Avancées*.

[7] J. Allen, "Performance-Directed Synthesis of VLSI Systems", *IEEE, Vol. 78, No. 2*, Février 1990.

[8] F. Anceau, "The Architecture of Microprocessors", *ed. Addison-Westley*, 1986.

[9] P. Anderson, L. Philpson, "Movie - an Interactive Environment for Silicon compilation tools", *IEEE Trans. on CAD, Vol. 8(6)*, Juin 1989.

[10] M.R. Barbacci, "A Comparison of Register Transfers Languages", *IEEE Trans. on computers Vol. C24 No. 2*, Février 1975.

[11] R.K. Brayton, R. Camposano, G. De Micheli, R.H.J.M. Otton, J. Van Eijndhoven, "The Yorktown Silicon Compiler System", *Silicon Compilation, ed. D.D. Gajski, pp. 204-310, Addison-Wesley Publishing Company*, 1988.

[12] G. Berry & L. Cosserat, "The Esterel Synchronous Programming Language and its Mathematical Semantics", *rapport technique*, Ecole Nat. Superieure de Mines de Paris, 1984.

[13] R.A. Bergamaschi, R. Camposano & M. payer, "Datapath Synthesis using path analysis", *28 ème DAC*, 1991.

[14] V. Berstis, "The V Compiler: Automating Hardware Design", *IEEE Design an Test, pp. 8-17*, 1987.

[15] J. Bhasker, H.C. Lee, "An optimizer for Hardware Synthesis", *IEEE Design & Test of computers, pp. 20-36*, 1990.

[16] T. Blackman, J. Fox, C. Rosebrugh, "The Silc^{TM} Silicon Compiler: Language and Features", *22 ème DAC, pp. 232-237*, Las Vegas, Juin 1985.

[17] D. Borrione, "Langage de description des systèmes logiques–Proposition pour une méthode formelle de définition", *Thèse d'état*, INPG Grenoble, 1981.

[18] F.D. Brewer, D.D. Gajski, "Knowledge Based Control in Micro-Architecture Design", *24 ème DAC, pp. 203-209*, Miami, 1987.

[19] Racal Redac Inc., *CADAT 2000 System Reference Manual*, 1991.

[20] R. Camposano, "Path-Based Scheduling for Synthesis" *IEEE T. CAD, Vol 10(1), pp. 85-93*, Janvier 1991.

[21] A.Z. Casavant et al, "A Synthesis Environment for Designing DSP Systems", *IEEE Design & Test of Computers, pp. 35-43*, 1989.

[22] R. Camposano & W. Wolf, "High-Level VLSI Synthesis", *Kluwer Academic Publishers*, 1991.

[23] M. Crates de Poulet, C. Duff, R. Leveugle, F. Poirot, G. Saucier, P. Sicard, "ASYL: A Logic And Architecture Design Automation System", *EURO-ASIC'89, pp. 183-209*, Grenoble, Janvier 1989.

[24] R.J. Cloutier, D.E. Thomas, "The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm", *27 ème DAC*, 1990.

[25] "Computer Networks and ISDN Systems", *Special issue : CCITT SDL, Vol 13, Num. 2*, 1987.

[26] C. Chu, M. Pontkonjak, M. Thaler, J. Rabaey, "HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications", *Proceeding ICCD'89, pp. 432-435*, Massachusetts, Octobre 1989.

[27] H. DeMan, F. Catthoor, G. Goossens, J. Van Meerbergen, S. Note, J. Huisken, "Architecture-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms", *Proc. IEEE, Vol. 78(2), pp. 319-355*, Février 1990.

[28] S. Devadas & A. R. Newton, "Algorithms For Hardware Allocation In Data Path Synthesis", *IEEE Trans. on CAD, Vol. 8, No. 7*, Juillet 1989.

[29] S.W. Director, A.C. Parker, D.P. Siewiorek, D.E. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems", *IEEE Trans. Circuits Syst., Vol. CAS-28(7)* Juillet 1981.

[30] J.R. Duley & D.L. Dietmeyer, "A Digital System Design Language (DDL)" *IEEE ToC, C-24(2)*, 1975.

[31] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE T. Computers, Vol C-30(7), pp. 478-490*, Juillet 1981.

[32] T.D. Friedman, S.C. Yang, "Methods Used in an Automatic Design Generator (ALERT)", *IEEE Trans. Comp. Vol. C-18, pp. 593-614*, 1969.

[33] D.D. Gajski, "Silicon Compilation", *Addison-Westley*, 1987.

[34] E.F. Girczyc, R.J.A. Buhr, J.P. Knight, "Applicability of a Subset of Ada as an Algorithmic Hardware Description Language for Graph-Based Hardware compilation", *IEEE Trans. on CAD, pp. 134-142*, Avril 1985.

[35] J. Granacki, D.Knapp, A.C. Parker, "The ADAM Advanced Design Automation System: Overview, Planner and Natural Language Interface", *22 ème DAC*, Juin 1985.

[36] B.S. Haroun, M.I. Elmasry, "Architectural Synthesis for DSP Compilers", *IEEE Trans. on CAD, Vol. 8(4), pp. 431-447*, 1989.

[37] R. Hartenstein, "Fundamentals of Structured Hardware Design", North Holland, 1977.

[38] C. Huang, Y. Chen, Y. Lin, Y. Hsu, "Data Path Allocation Based on Bipartite Weighted Matching", *27 ème DAC, pp. 499-504*, Orlando, 1990.

[39] Genrad Inc, *System HILO 4.0 System Reference Manual*, 1991.

[40] C. Y. Hitchcock & D. E. Thomas, "A Method Of Automatic Data Path Synthesis", *20 ème DAC, papier 31.3*, 1983.

[41] "IEEE Standard VHDL Langage Reference Manual", *NY USA, IEEE Std. 1076-1987*, Mars 1987.

[42] International Standard, ESTELLE (Formal description technique based on an extended state transition model), *ISO/DIS 9074*, 1987.

[43] R. Jamier, A.A. Jerraya, "APOLLON: a data-path silicon compiler", *IEEE Circuits & Devices*, Mai 1985.

[44] R. Jamier, "Génération automatique de parties opératives de circuit VLSI de type microprocesseur", *Thèse INPG*, Novembre 1986.

[45] A.A. Jerraya, "Contribution à la Compilation de Silicium et au Compilateur SYCO", *Thèse d'état*, TIMA/INPG, Grenoble, Décembre 1989.

[46] A.A. Jerraya, K. O'Brien, "SOLAR: An Intermediate Format for System-Level Design and Specification", *IFIP Intl Workshop on Hardware/software Co-Design*, Grassau, Allemagne, Mai 1992.

[47] A. Jerraya, K. O'Brien, I. Park, B. Courtois, "Towards System-Level Modeling And Synthesis", *VLSI DESIGN'92*, India, Février 1992.

[48] A.A. Jerraya, I. Park, K. O'Brien, "AMICAL: An Anteractive High Level Synthesis Environment", *Proceeding EDAC*, Février 1993

[49] A.A. Jerraya, N. Mhaya, J.P. Geronimi, B. Courtois, "SYCO - a Silicon Compiler for VLSI ASICs Specified by Algorithms", *Computer-Aided Enginneering Journal, pp. 122-130*, 1988.

[50] D. Johannsen, "Bristle blocks: A silicon compiler", *16 ème DAC, pp. 310-313*, 1979.

[51] K. Kchouk, "Intégration du Test et de la Synthèse au niveau architectural", *Projet de fin d'études*, TIMA/INPG, Août 1993.

[52] H. Krämer, M. Neher, G. Rietsche, W. Rosenstiel, "Data Path and Control Synthesis in the CADDY System", *International Workshop at INPG*, Grenoble, 1988.

[53] T. J. Kowalski & D. E. Thomas, "The VLSI Design Automation Assitant / What's in a Knowledge Base", *22 ème DAC, papier 18.1*, 1985.

[54] K. Küçükçakar & A.C. Park, "Data Path Tradeoff Using MABAL", *27 ème DAC, papier 29.3*, 1990.

[55] D.E. Krekelberg, G.E. Sobelman, C.S. John, "Yet Another Silicon Compiler", *22 ème DAC*, 1985.

[56] R. Lasocki, "PAT : Programmable Architecture Translator", *Rapport interne*, TIMA/INPG, Septembre 1992

[57] U. Lauther, "Introduction to Synthesis", *The synthesis Approach to Digital System Design, Kluwer Academic Publishers*, 1992.

[58] J.S. Lis, D.D. Gajski, "Synthesis from VHDL", *Proceeding ICCD'88, pp. 378-381*, Octobre 1988.

[59] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh et al, "PHIDEO, A silicon Compiler for High Speed Algorithms", *Proceedings EDAC'91, pp. 436-441*, Amsterdam, Mars 1991.

[60] "LOTOS a formal description technique based on the temporal ordering of observational behavior", *ISO, IS 8807*, Février 1989.

[61] P. Marwedel, "The MIMOLA Design System: Detailed Description of the Software System", *16 ème DAC, pp. 59-63*, 1979.

[62] P. Marwedel, "A New Synthesis Algorithm For The MIMOLA Software System", *23 ème DAC, papier 15.2*, 1986.

[63] M.C. McFarland, A.C. Parker, R. Camposano, "The High-Level Synthesis of Digital Systems", *IEEE, Vol. 78, No. 2, pp. 301-318*, Février 1990.

[64] G. De Micheli, D.C. Ku, "HERCULES - A System for High-Level Synthesis", *25 ème DAC*, 1988.

[65] S. Note, al., "Cathedral III: Architecture-driven HL synthesis for high throughput DSP applications", *28 ème DAC*, 1991.

[66] K. O'Brien, "Compilation de silicium: du circuit au système", *Thèse INPG*, Mars 1993.

[67] K. O'Brien, M. Rahmouni, A.A.Jerraya, "DLS: A Scheduling Algorithm for High-Level Synthesis in VHDL", *EDAC'93*, Paris, France, Février 1993.

[68] B.M. Pangrle, D.D. Gajski, "Slicer: A State Synthesizer for Intelligent Silicon Compilation", *ICCD'87, pp. 42-45*, 1987.

[69] P.G. Paulin, J.P. Knight, E.F. Girczyc, "HAL: A Multi-paradigm Approach to Automatic Data Path Synthesis", *23 ème DAC*, 1986.

[70] P.G. Paulin, J.P.Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transactions on CAD, Vol.8, No.6*, pp. 661-679, Juin 1989.

[71] C. A. Papachristou & H. Konuk, "A Linear Driven Scheduling and Allocation Method Followed By An Interconnect Optimization Algorithm", *27 ème DAC*, 1990.

[72] B.M. Pangrle, "SPLICER: A Heuristic Approach to Connectivity Binding", *25 ème DAC, pp. 536-541*, 1988.

[73] I. Park, K. O'Brien, A.A. Jerraya, "An Interactive Data-Path Allocation Algorithm", *Workshop on Control Dominated Synthesis From an RTL Rescription*, Grenoble, France, Septembre 1992.

[74] I. Park, "AMICAL: Un assistant pour la synthèse et l'exploration architecturale des circuits de commande ", *thèse INPG*, Juillet 1992.

[75] Z. Peng, "Synthesis of VLSI Systems with the CAMAD Design Aid", *23 ème DAC*, 1986.

[76] F.J. Ramming, "Synthesis Related Aspects of Simulation", *The synthesis Approach to Digital System Design, Kluwer Academic Publishers*, 1992.

[77] V.K. Rai, C.S. patwardhan, "Automated Data Path Synthesis to Avoid Global intercinnects", *Proc. IEEE, pp. 11-16*, 1991.

[78] J. Rabaey, H. Deman, J. Vanhoof, G. Goossens et al, "CATHEDRAL-II: A Synthesis System for Multiprocessor DSP Systems", *Silicon Compilation, ed. D.D. Gajski, pp. 311-360, Addison-Westley Publishing Company*, 1988.

[79] B. Rouzeyre, T. Ezzedine, G. Sagnes, "Operators Allocation in the silicon compiler SCOOP", *Integration, pp. 99-109*, 1989.

[80] C. Safinia, R. Leveugle, "Clocking scheme selection for circuits made up of a controller and a datapath", *Synthesis for Control Dominated Circuits, G. Saucier, J. Trilhe(eds), IFIP*, 1993.

[81] J.R. Southard, "MacPitts: An Approach to silicon compilation", *Computer, Vol. 16(12)*, Décembre 1983.

[82] "Synopsys Design Analyzer Reference Manual", version 3.0, Décembre 1992.

[83] "Synopsys VHDL System Simulator Tutorial", version 3.0b, Juin 1993.

[84] T. Tanaka, T. Kobayashi, O. karatsu, "HARP: Fortran to Silicon", *IEEE Trans. on CAD, Vol. 8(6)*, 1989.

[85] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, R.L. Blackburn, "The system Architect's Workbench", *25 ème DAC, pp. 337-343*, Juin 1988.

[86] D.E. Thomas et al, "Automatic data path synthesis" *Computer*, Décembre 1983.

[87] R.A.Tierney, "Modelling Complex Systems", *VLSI System Design*, Mai 1988.

[88] H. Trickey, "Flamel: A High-Level Hardware Compiler", *IEEE Trans. on CAD, pp. 259-269*, 1987.

[89] C.J. Tseng, R.S. Wei, S.G. Rothweiler, M.M. Tong, "Bridge: A Versatile Behavioral Synthesis System", *25 DAC, pp. 415-420*, 1988.

[90] C. Tseng & D. S. Siewiorek, "Facet : A Procedure for the Automated Synthesis of Digital System", *20 ème DAC, papier 31.4*, 1983.

[91] UDL/I Language Reference, Draft Version 1.0b4, 4 Octobre 1990.

[92] F.Vahid, S.Narayan et al, "SpecCharts: A Language For System-Level Synthesis", *CHDL'91, pp. 145-154*, Avril 1991.

[93] N. S. Woo & H. C. Shin, "A Technology-Adaptive Allocation Of Functional Units And Connections", *26 ème DAC, papier 37.2*, 1989.

[94] G. Zimmermann, "The MIMOLA Design System: A Computer Aided Digital Processor Design Method", *16 ème DAC*, Juin 1979.

Annexes

Annexe A

Définition des exemples

Annexe A

Définition des exemples

A.1 Opérateur multi-fonctions

Le programme suivant présente l'algorithme de l'opérateur multi-fonctions. Il exécute quatre opérations standards (addition, soustraction, multiplication et réciproque) sur des nombres à virgule fixe de 32 bits. La mantice du résultat est donnée sur 12 bits et la partie flottante est sur 20 bits.

```
1   package fu_pkg is
2       subtype int3bit is integer range 0 to 7;
3
4       function constexp20 return integer;
5       function shiftleft(i: integer) return integer;
6       function shiftright(i: integer) return integer;
7       function convleft(i : integer) return integer;
8       function selectlsb(i : integer) return integer;
9       function qsave(i, j : integer) return integer;
10      function addmul(i, j, k: integer) return integer;
11  end fu_pkg;
12
13  package body fu_pkg is
14      function constexp20 return integer is
15      constant val : integer := 2**20;
16      begin
17          return val;
18      end constexp20;
19
20      function shiftleft(i: integer) return integer is
```

```
21        variable val : integer;
22        begin
23              if (i<2**30 and i>=-2**30) then val := i * 2;
24              elsif (i>=2**30) then val := (i - 2**30 - 2**30) * 2;
25              else val := (2**30 + i + 2**30) * 2;
26              end if;
27              return (val);
28        end shiftleft;
29
30        function shiftright(i: integer) return integer is
31        variable val : integer;
32        begin
33              val := i / 2;
34              return (val);
35        end shiftright;
36
37        function convleft(i : integer) return integer is
38        variable val : integer;
39        begin
40              if (i<2**(21) and i>=-2**21) then val := i * 2**10;
41              elsif (i>=2**21) then val := (i rem 2**21) * 2**10;
42              else val := (i + 2**22) * 2**10;
43              end if;
44              return (val);
45        end convleft;
46
47        function selectlsb(i : integer) return integer is
48        begin
49              return (i rem (2**20));
50        end selectlsb;
51
52        function qsave(i, j: integer) return integer is
53        variable val : integer;
54        begin
55              if (j < 0) then val := i;
56              else val := j;
57              end if;
58              return val;
59        end qsave;
60
61        function addmul(i, j, k: integer) return integer is
62        begin
63              return (i + (j rem 2) * k);                    -return (i + j(lsb)*k)
64        end addmul;
65  end fu_pkg;
66
67  use work.fu_pkg.all;
68  entity operateur_multi-fonctions is
69        port (      input1      : in integer;
70                    input2      : in integer;
71                    sel         : in bit;
72                    com         : in int3bit;
73                    output      : out integer;
74                    done        : out bit);
75  end operateur_multi-fonctions;
76
77  architecture behaviour of operateur_multi-fonctions is
78  begin
79        process
80        variable A, B, T, X, Q: integer;
81        begin
82              wait until (sel='1');
83              case com is
84                    when 1                                   - restoring division algorithm
85                          T := constexp20;                   - T := AC & MQ
86                          X := 0;
87                          A := input1;
```

```
88                      done <= '0';
89                      wait for 0 ns;
90                      while (X /= 20) loop
91                          Q := shiftleft(T);          - Tnew := T * 2
92                          X := X + 1;
93                          T := Q - A;                 - T := Tnew - A
94                          if (T>0 and A>0) then
95                              T := T + 1;             - If T>0 and T<Tnew then T:=T + 1
96                          end if;
97                          T := qsave(Q,T);            - T = "remainder" & "quotient"
98                      end loop;
99                      Q := selectlsb(T);
100                 when 2 =>                           - mul 32bits
101                                                     - shift and add algorithm
102                     T := 0;
103                     X := 0;
104                     A := input1;
105                     B := input2;
106                     done <= '0';
107                     wait for 0 ns;
108                     while (X /= 30) loop
109                         T := addmul(T,A,B);         - T:=(T + A(i)*B) / 2
110                         T := shiftright(T);
111                         A := shiftright(A);
112                         X := X + 1;
113                     end loop;
114                     if (A<0) then
115                         T := addmul(T,A,B);         - T := (T - A(31)*B) / 2 ;
116                         Q := 0 - T;
117                     else Q := addmul(T,A,B);
118                     end if;
119                     Q := convleft(Q);
120                 when 3 =>                           - add 32bits
121                     A := input1;
122                     B := input2;
123                     Q := A + B;
124                     done <= '0';
125                     output <= Q;
126                 when 4 =>                           - sub 32bits
127                     A := input1;
128                     B := input2;
129                     Q := A - B;
130                     done <= '0';
131                     output <= Q;
132                 when 5 =>                           - result 32bits
133                     output <= Q;
134                     done <= '1';
135                 when others =>
136             end case;
137         end process;
138     end behaviour;
```

Figure A.1: *Programme VHDL de l'opérateur multi-fonctions.*

A.2 Répondeur téléphonique

```
1   Entity ctrl is
2       Port(   Ring            : in Bit;
3               Voice           : in Bit;
4               KeyPressed      : in Integer;
5               HangUp          : in Bit;
6               AnnounceSig     : out Bit;
7               MsgPanel        : out Integer;
8               Beep            : out Bit;
9               AlarmSig        : out Bit;
10              Connection      : out Bit;
11              PlayTape1       : out BIT;
12              EndTape1        : in Bit;
13              RewindTape2     : out Bit;
14              PlayTape2       : out Bit;
15              RecordTape2     : out Bit;
16              FastFwdTape2    : out Bit;
17              StandBy2        : in Bit;
18              EndTape2        : in Bit;
19              ElapsedTime     : in Integer);
20  End ctrl;
21
22  Architecture behaviour of ctrl is
23  Type Memory is Array(0 to 2) of Integer;
24  Begin
25      ans : Process
27      Function FU_REM(in1, in2: Integer) Return Integer is
28      Variable res1 : Integer;
29      Begin
30          res1 := in1 + in2;
31          res1 := res1 REM 1024;
32          Return res1;
33      End FU_REM;
35      Variable RingCount : Integer := 0;
36      Variable MsgCount : Integer := 0;
37      Variable DigitCount : Integer;
38      Variable NextDigit : Integer;
39      Variable Timeout : Integer;
40      Variable LocalTimeout : Integer;
41      Variable PasswdRAM : Memory := (1,2,3);
42      Begin
43  -**********************************************************************
44  - BLOCK: WAIT_FOR_A_CALL
45  -
46  - COUNT RINGS. (Ring = '1' => a telephone ring)
47  - If RingCount is 3 then answer (exit block).
48  - If it is less than 3 and no further rings arrive
49  - within 2sec it means that the call was abandoned.
50  - In this case, RingCount is reset to zero. A ring
51  - must last for more than 1sec before it is considered
52  - valid.
53  -
54  -**********************************************************************
55          Wait_For_A_Call : Loop
56              RingCount := 0;
57              Wait until (Ring = '1');
58              Count_3_Rings :Loop
59                  RingCount := RingCount + 1;
60                  If (RingCount = 3) Then
61                      Exit Wait_For_A_Call;
62                  End if;
63                  LocalTimeout := FU_REM(ElapsedTime, 1);- 1s for ringing
64                  Wait until ((ElapsedTime = LocalTimeout) Or (Ring = '0') Or
```

```
65                      (HangUp = '1'));
66
67    – if it dœsn't ring long enough then restart:
68    – "goto Wait_For_A_Call"
69
70                      If ((Ring = '0') Or (HangUp = '1')) Then
71                          Exit Count_3_Rings;
72                      End if;
73                      Wait until (Ring = '0');– wait for end of ringing
74                                  – 2s idle time before
75                      LocalTimeout := FU_REM(ElapsedTime,2 );
76                                  – the next ring
77                      Wait until ((ElapsedTime = LocalTimeout) Or (Ring = '1'));
78
79    – if the next ring dœsn't arrive then restart:
80    – "goto Wait_For_A_Call"
81
82                      If ((ElapsedTime = LocalTimeout) Or (HangUp = '1')) Then
83                          Exit Count_3_Rings;
84                      End if;
85                  End loop Count_3_Rings;
86              End loop Wait_For_A_Call;
87    –*********************************************************************
88    – BLOCK: OffHook
89    –
90    – The pre-recorded message is played from deck1.
91    – This message takes 30s. If the EndTape1 signal is not
92    – received after 31s or the remote control signal
93    – (KeyPressed = 8) is not received
94    – before the message is finished, there is an error
95    – and the correspondance is aborted.
96    –
97    –*********************************************************************
98              OffHook: Loop
99                  Connection <= '1'; –The receiver is picked up
100                 AnnounceSig <= '1';
101                 PlayTape1 <= '1'; –play message
102                 Timeout := FU_REM(ElapsedTime, 31) ;
103                 Wait until ((ElapsedTime = Timeout) Or (HangUp = '1') Or
104                 (EndTape1 = '1') Or (KeyPressed = 8));
105                 PlayTape1 <= '0'; –reset deck1
106                 AnnounceSig <= '0';
107
108   – IF the caller hangs up OR deck1 takes more than 31s to play
109   – the message THEN restart: "goto Wait_For_A_Call"
110
111                 If ((HangUp = '1') Or (ElapsedTime = Timeout)) Then
112                     Exit OffHook;
113                 End if;
114   –*********************************************************************
115   –
116   – If the pre-recorded message reaches its end
117   – normally (EndTape1 = '1'), the system records a new
118   – message and exits when the caller hangs up.
119   – If KeyPressed=8 then we enter the remote control mode.
120   – The user must first enter a 3 digit password.
121   – The messages so far recorded are then played
122   – in sequence. The user then has the option of
123   – rewinding the tape, replaying the messages or
124   – fast-forwarding the tape. The session ends when
125   – the user hangs up.
126   –
127   –*********************************************************************
128                 If (EndTape1 = '1') Then –record a message normally
129                     Beep <= '1';
130                     LocalTimeout := FU_REM(ElapsedTime, 1); –duration of Beep
131                     Wait until ((ElapsedTime = LocalTimeout) Or (HangUp = '1'));
```

```
132                     Beep <= '0';
133                     If (HangUp = '1') Then
134                         Exit OffHook;
135                     End if;
136                     RecordTape2 <= '1';
137                     Timeout := FU_REM(ElapsedTime, 10); -caller has 10s to start talking
138                     Wait until Voice='1' Or ElapsedTime = Timeout Or HangUp='1';
139                     If ((HangUp = '1') Or (ElapsedTime = Timeout)) Then
140                         Exit OffHook;
141                     Else      -Voice = '1' => caller is speaking
142                         MsgCount := MsgCount + 1;
143                         MsgPanel <= MsgCount;
144                     End if;
145                     Wait until (HangUp = '1');
146                     RecordTape2 <= '0';
147                     Exit OffHook; -end of session
148 -*********************************************************************
149 -
150 - REMOTE CONTROL
151 -
152 -*********************************************************************
153                 Else - (KeyPressed = 8) => remote control
154                     DigitCount := 0;
155                     Remote:Loop
156                     Get_3_Digits: Loop
157                         Wait until (KeyPressed = 0); -Button is reset
158                         Timeout := FU_REM(ElapsedTime,20) ;
159                         Wait until ((ElapsedTime = Timeout) Or (KeyPressed /= 0) Or
160                         (HangUp = '1'));
161
162 - IF the caller hangs up OR no password is entered within 20s
163 - THEN restart : "goto Wait_For_A_Call"
164
165                         If ((HangUp = '1') Or (ElapsedTime = Timeout)) Then
166                             Exit OffHook;
167                         End if;
168                         If (KeyPressed /= PasswdRAM(DigitCount)) Then
169                             AlarmSig <= '1'; -false password
170                             LocalTimeout := FU_REM(ElapsedTime,1) ;
171                             Wait until ((ElapsedTime = LocalTimeout) Or (HangUp = '1'));
172                             AlarmSig <= '0';
173                             Exit OffHook; - restart : "goto Wait_For_A_Call"
174                         End if;
175                         DigitCount := DigitCount + 1;
176                         If (DigitCount = 3) Then
177                             Exit Get_3_Digits;
178                         End if;
179                     End loop Get_3_Digits;
180 -*********************************************************************
181 -
182 - All messages are automatically played
183 -
184 -*********************************************************************
185                     RewindTape2 <= '1';
186                     Timeout := FU_REM(ElapsedTime,16); -max rewind time
187                     Wait until ((ElapsedTime = Timeout) Or (HangUp='1') Or (StandBy2 = '1'));
188                     RewindTape2 <= '0';
189                     If ((HangUp = '1') Or (ElapsedTime = Timeout)) Then
190                         Exit OffHook;
191                     End if;
192                     PlayTape2 <= '1';
193                     Timeout := FU_REM(ElapsedTime,1022); -max time to play all tape
194                     Wait until ((ElapsedTime = Timeout) Or (HangUp='1') Or (EndTape2 = '1'));
195                     PlayTape2 <= '0';
196                     MsgCount := 0;
197                     MsgPanel <= MsgCount;
198                     If ((HangUp = '1') Or (ElapsedTime = Timeout)) Then
```

```
199                     Exit OffHook;
200                 End if;
201 –*****************************************************************
202 –
203 – The caller can then manually control the answering machine
204 – enabling him to rewind the tape, replay the messages or
205 – fast forward the tape.
206 –
207 –*****************************************************************
208                 ManualControl: Loop
209                     Timeout := FU_REM(ElapsedTime, 20); –20s to enter a code
210                     Wait until ((HangUp = '1') Or (ElapsedTime = Timeout) Or
211                     (KeyPressed /= 0));
212
213 – IF the caller hangs up OR no command was entered within 20s
214 – THEN restart : "goto Wait_For_A_Call"
215
216                     If ((HangUp = '1') Or (ElapsedTime = Timeout)) Then
217                         Exit OffHook;
218                     END IF;
219                     Case (KeyPressed) is
220                         When 1 =>
221                             RewindTape2 <= '1';
222                             Timeout := FU_REM(ElapsedTime,16);
223                             Wait until ((ElapsedTime = Timeout) Or (HangUp = '1') Or
224                             (KeyPressed = 0) Or (StandBy2 = '1'));
225                             RewindTape2 <= '0';
225                         When 2 =>
227                             PlayTape2 <= '1';
228                             Timeout := FU_REM(ElapsedTime,1022);
229                             Wait until ((ElapsedTime = Timeout) Or (HangUp = '1') Or
230                             (EndTape2 = '1') Or ((KeyPressed /= 0) And
231                             (KeyPressed /= 2)));
232                             PlayTape2 <= '0';
233                         When 3 =>
234                             FastFwdTape2 <= '1';
235                             Timeout := FU_REM(ElapsedTime,16);
236                             Wait until ((ElapsedTime = Timeout) Or (HangUp = '1') Or
237                             (KeyPressed = 0) Or StandBy2 = '1'));
238                             FastFwdTape2 <= '0';
239                         When others =>
240                             Exit OffHook;
241                     End case;
242
243 – IF the operation was stopped due to a hang-up OR a timeout
244 – THEN restart : "goto Wait_For_A_Call"
245
246                     If ((HangUp = '1') Or (ElapsedTime = Timeout)) Then
247                         Exit OffHook;
248                     End if;
249                 End loop ManualControl;
250                 Exit Remote;
251             End loop Remote;
252             End if;
253             Exit OffHook;
254         End loop OffHook;
255         Connection <= '0'; –Disconnect
256     end Process;
257 end behaviour;
```

Figure A.2: *Programme VHDL du répondeur téléphonique.*

Annexe B

Règles de traduction SOLAR-VHDL

Annexe B

Règles de traduction SOLAR-VHDL

B.1 SOLAR: Une forme intermédiaire pour la synthèse de haut niveau

SOLAR est une forme intermédiaire pour la description des spécifications de contrôle au niveau système. Elle est utilisée pour la modélisation des systèmes matériels complexes à flux de contrôle [66]. SOLAR repose sur des concepts de haut niveau permettant différents types de description (structurel et comportemental) et ce à différents niveaux d'abstraction évoluant du niveau système logiciel au niveau transfert de registres.

SOLAR est un modèle de représentation des concepts de haut niveau dans la synthèse des systèmes à flux de contrôle. Il est composé d'un modèle de représentation des données et d'un langage textuel. Les systèmes sont représentés textuellement et manipulés à travers une structure de données interne et une interface procédurale.

La structure de données SOLAR est décrite en C++. Chaque objet manipulé est représenté par une classe. Une classe est composée des éléments de l'objet et d'une interface procédurale pour manipuler ces éléments.

La classe SOLAR

La classe SOLAR représente le nœud racine de la structure de données correspondant à une description SOLAR d'un système mixte (logiciel/matériel).

Les éléments de la classe sont:

- Elément privé:
  - *name*: nom du système.
- Eléments publics:
  - *designunit*: tête de la liste des unités de conception.
  - *channelunit*: tête de la liste des canaux.
  - *functionunit*: tête de la liste des unités fonctionnelles.

Le constructeur de la classe est:

- *SolarObject*(): Utilisé pour l'initialisation des éléments de la classe.

L'interface procédurale est:

- *Pre_vhdl*(): Réalise le traitement de transposition pour cette classe.
- *IsA*(): Donne le type de cet objet (dans ce cas SolarNode)
- *SetName*(ObjectType* n): Donne à l'élément privé *name* l'adresse de n.
- *GetName*(): Donne l'adresse mémoire de l'élément privé *name*.

La description C++ correspondante est:

```
1   Class SolarObject : public Object
2   {
3         NameObject *name;
4
5   public :
6         List* designunit;
7         List* channelunit;
8         List* fonctionunit;
9
10  SolarObject()
11
12  void Pre_vhdl()
13  ObjectType IsA() {return SolarNode;}
14  void SetName(NameObject* n) {name = n;}
15  NameObject* GetName() {return name;}
16  }
```

Figure B.1: *Description générale d'une Classe SOLAR.*

B.2 Concepts de base

SOLAR comporte trois différents types d'abstraction: la description comportementale au niveau SOLAR est décrite par des machines d'états hiérarchiques, la spécification structurelle se fait à partir d'unités de conception et de canaux, et la description mixte (comportementale/structurelle) se décrit en utilisant des unités fonctionnelles.

B.2.1 *Table d'états*

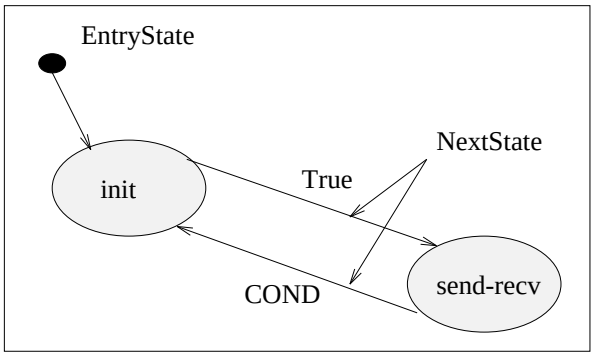
La table d'états est le constructeur de base de SOLAR. Elle contient un ensemble d'états (*states*) et quelques attributs tels que l'état d'entrée par défaut (*EntryState*), etc. (figure B.2).

La liste des états (*StateList*) donne les noms de tous les états définis dans la table d'états. Chaque état feuille contient une liste d'opérations qui sont exécutées quand l'état est activé. Ces opérations peuvent contenir des affectations aux ports ou aux variables, ainsi que des opérations de contrôle de processus.

La table d'états étant représentée par un seul processus, les états sont donc les différentes alternatives d'une instruction *case*. Cette instruction a pour argument

(condition) le signal de contrôle introduit.

Instruction NextState Cette instruction a pour sémantique de désactiver l'état courant de la machine et d'activer l'état dont le nom est passé en argument de l'instruction (figure B.2).



(a)

Figure B.2: *Représentation d'une table à deux états.*

B.2.2 *Unité de conception*

Une unité de conception (*DesignUnit*) permet la structuration d'une description système sous forme d'un ensemble de sous-systèmes communicants. Chaque unité de conception peut contenir, soit une ou plusieurs tables d'états (pour des unités de conception comportementales) qui décrivent son comportement interne, soit des instanciations de composants existants ou d'autres unités de conception (unités de conception structurelle).

Dans notre cas, les descriptions SOLAR sont donc faites à l'aide d'une unité de conception contenant une table d'états pour la description de la partie contrôle et d'une unité de conception contenant des instanciations pour les descriptions de la partie opérative et de la configuration globale. Pour le premier cas, les états de la table d'états sont séquentiels et peuvent contenir des actions nécessitant un ou plusieurs cycles d'horloge pour leurs exécutions.

B.3 Principales règles de traduction

AMICAL génère en sortie la description SOLAR de l'architecture par l'intermédiaire de trois fichiers; le premier fichier décrit la partie contrôle, le second la partie opérative et le troisième contient une instanciation de chacun des deux premiers ainsi que l'interconnexion entre elles formant ainsi le circuit global.

B.3.1 Modélisation VHDL de la partie contrôle

Comme mentionné dans la thèse, la partie contrôle est une machine d'états finis de Mealy représentée sous forme de table de transitions. Cette représentation peut se traduire en matériel par une partie logique (C) et un registre de mémorisation (R) commandé par des signaux d'horloge (figure B.3).

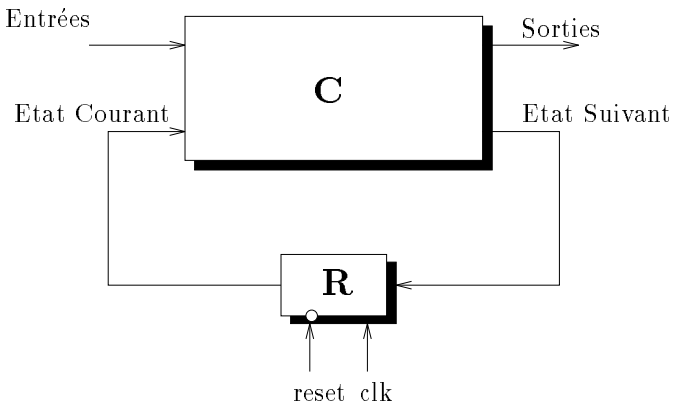


Figure B.3: *Représentation d'une machine d'états finis.*

A un tel système correspond une description SOLAR au niveau transfert de registres, dans laquelle figure une seule table d'états dont la modélisation est spécifique aux machines décrites au cycle.
La description VHDL correspondant à la partie contrôle se fera donc au moyen de deux processus:

- le premier correspond à la partie logique (C) contenant une instruction *case* permettant le branchement des différents macro-cycles (figure B.4).

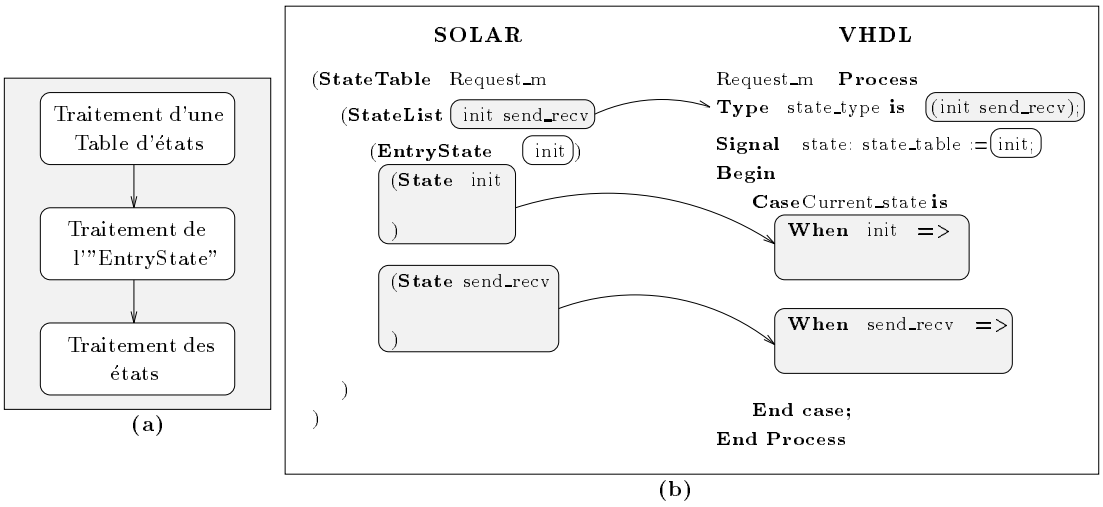


Figure B.4: *Processus de traduction de la table d'états.*
(a) *Procédé de traitement*, (b) *Exemple de traduction d'une table d'états.*

- Le deuxième correspond au registre R assurant la synchronisation de la génération des vecteurs de commande. Ce dernier processus est toujours le même et il a la structure présentée dans la figure B.5

```
1   SYNCH: Process(cont_clk, cont_reset, CURRENT_STATE)
2   Begin
3       if cont_reset = '0' then;
4           – Les conditions du reset;
5           Current_state <= EntryState;
6       elsif cont_clk = '1' And cont_clk'event then
7           Current_State <= Next_state;
8       end if;
9   end Process SYNCH;
```

Figure B.5: *Représentation VHDL du processus de synchronisation.*

B.3.2 Modélisation VHDL de la partie opérative

La partie opérative est une description structurelle composée d'instances d'unités fonctionnelles, de registres et d'un réseau d'interconnexions (bus, connecteurs, etc.). En SOLAR la description structurelle d'une unité de conception est composée:

- d'une interface (ports et accès de l'unité de conception),

- d'une liste d'instances des unités de conception,
- d'un réseau de connexions représentant l'interconnexion entre les unités de conceptions instanciées.

(i) *Interface*

L'interface d'une unité de conception regroupe des ports d'entrée/sortie et des références d'accès à des canaux. La représentation VHDL des ports est directe, alors que les accès nécessitent un traitement particulier. Toutefois un accès peut être considéré comme un ensemble de ports qui seront remplacés en VHDL par des signaux dans l'architecture.

(ii) *Instance*

L'instance en SOLAR permet l'utilisation d'une unité de conception déjà décrite en spécifiant une partie ou la totalité de son interface. On peut donc avoir plusieurs instances de la même unité de conception.
L'utilisation des instances facilite la modélisation des systèmes; en effet, ce principe permet l'utilisation de bibliothèques de travail où différentes unités de conception sont décrites (additionneurs, multiplieurs, etc.) une seule fois afin d'éviter de redécrire une unité de conception déjà utilisée. Ainsi, ces instances SOLAR seront traduites en composants (*components*) VHDL.

(iii) *Réseau de connexions*

Le réseau de connexions SOLAR représente les interconnexions entre les différentes unités de conception qui constituent le système. Les connexions rencontrées relient aussi bien des ports d'instances que des accès aux canaux. Les différentes connexions à l'intérieur d'une vue structurelle permettent de décrire une unité de conception au moyen d'autres unités de conception décrites au préalable.
La modélisation VHDL d'une description structurelle nécessite l'introduction d'éléments de conception VHDL tels que "*component*" pour la déclaration d'éléments instanciables et des instructions de connexion et d'instanciation tels que "*port map*".

La transposition en VHDL d'une telle unité de conception est représentée dans la figure B.6.

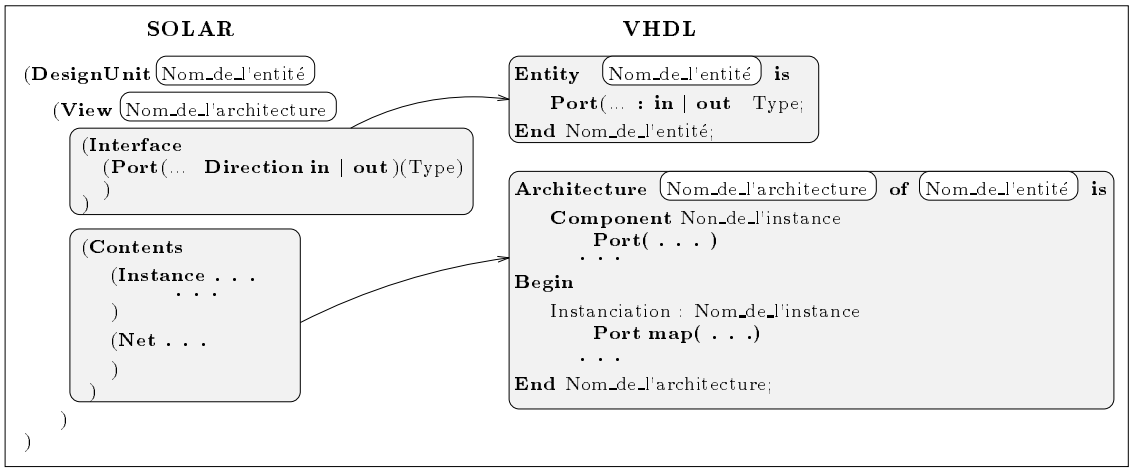


Figure B.6: *Processus de traduction de l'Unité de conception.*

Chaque fois que le mot clé *Interface* est rencontré, la traduction se fait en VHDL par la déclaration des ports (*Port*). Le mot clé SOLAR *Contents* spécifie le contenu du circuit (*DesignUnit*), donc, il est traduit en VHDL en *Architecture* contenant la fonctionnalité du circuit. Le nom de l'entité et celui de l'architecture sont donnés en SOLAR par les mots clés *DesignUnit* et *View* respectivement.

B.3.3 Modélisation VHDL du circuit global

La configuration du circuit global correspond à une description structurelle puisqu'elle instancie deux composants (partie opérative et partie contrôle) et relie ces deux derniers à l'aide de signaux et de canaux. Donc, sa traduction VHDL est la même que celle de la partie opérative.

La correspondance entre SOLAR et VHDL est donnée dans le tableau suivant:

Instructions SOLAR	Instructions VHDL
DesignUnit	Entity
View	Architecture
Interface	Port
Port	Port
StateTable	Process
State (S)	When (S)
Alt (Condition)	If (Condition) Then
StateList	Type State_Type is ()
Instance	Component
ViewRef	Architecture
PortInstance	Port
Net	Port map
Joined	Port map
InstanceRef	Instanciation (X: Instance)
PortRef	Port map

Table B.1: *Correspondance entre SOLAR et VHDL.*

Annexe C

Présentation d'un exemple complet

Annexe C

Présentation d'un exemple complet

Cette annexe présente un exemple complet de circuit généré en partant d'une description comportementale donnée en VHDL.
Parmi différentes étapes de compilation de silicium, la synthèse architecturale est réalisée par AMICAL. Toutes les étapes de synthèse de haut niveau exécutées par AMICAL ainsi que les procédures de personnalisation de l'architecture abstraite et la traduction des fichiers de sortie SOLAR en leurs équivalents VHDL seront présentées dans cette annexe.
La validation de la fonctionnalité des circuits générés par AMICAL au niveau transfert de registres se fait par simulation en utilisant le simulateur VHDL de Synopsys. Le chemin d'exécution a été testé sur plusieurs exemples dont celui qui sera présenté dans cette annexe; le pgcd.

C.1 Description comportementale

La figure C.1 montre le fichier de la description comportementale de l'algorithme qui calcule le plus grand commun diviseur (*pgcd*) de deux entiers. Ce fichier a pour

extension "*.vhd*".

```
%*****************************************************************************%
%*** Specification comportementale du pgcd ***** Nom du fichier : gcd.vhdl ***%
%*****************************************************************************%

                    entity gcd is
                         port (clk    : in bit;
                               reset  : in bit;
                               dout   : out bit;
                               xi, yi : in integer;
                               ou     : out integer;
                               start  : in bit;
                               din    : in bit);
                    end gcd;
                    architecture behavior of gcd is
                    begin
                         process
                         variable x,y: integer;
                         begin
                              if (start /= '1') then
                                   wait until (start = '1');
                              end if;
                              wait until (din = '1');
                              dout <= '0';
                              wait for 0ns;
                              x := xi;
                              y := yi;
                              while (x /= y ) loop
                                   if (x < y) then
                                        y:=y - x;
                                   else
                                        x:=x - y;
                                   end if;
                              end loop;
                              dout <= '1';
                              ou <= x;
                         end process;
                    end behavior;
```

Figure C.1: *Description comportementale du pgcd.*

Le processus commence à chaque front montant du signal *start*. Chaque nouvelle paire de données prise en entrée doit être accompagnée par la mise à '1' du signal de validation *din*. Le signal de validation *dout* est quant à lui mis à '1' quand le résultat du calcul est présent sur la sortie *ou*.

Deux signaux ont été ajoutés au ports de l'entité: *clk* (pour l'horloge) et *reset* (pour la remise à zéro). Ces deux signaux, comme le montre la figure C.1 n'interviennent pas dans le processus de calcul. Ils ont été ajoutés seulement pour avoir la même interface que celle qui sera obtenue au niveau transfert de registres.

C.2 Compilation

La première étape exécutée par AMICAL est la compilation de la description comportementale. AMICAL utilise le compilateur VHDL de CLSI. Cette étape permet de détecter des erreurs syntaxiques de programmation VHDL et de créer l'arbre syntaxique nécessaire à la suite des opérations.

C.3 Simulation de la description comportementale

La fonctionnalité de la spécification comportementale du pgcd a été vérifiée par simulation en utilisant un fichier de test où deux paires de données sont utilisées. Le programme de stimuli est donné par le fichier *gcd_tb.vhdl* (figure C.2).

```
%*****************************************************************************%
%***     Programme test    ********     Nom du fichier : gcd_tb.vhd     ***%
%*****************************************************************************%

        library mvl_7;
        use mvl_7.types.all;
        entity gcd_tb is end;
        architecture struct of gcd_tb is
            component gcd
               port (clk    : in bit;
                     reset  : in bit;
                     dout   : out bit;
                     xi, yi : in integer;
                     ou     : out integer;
                     start  : in bit;
                     din    : in bit);
            end component;

        signal Xi,Yi : integer;
        signal clk: bit := '1';
        signal reset: bit := '0';
        signal start, din, dout: bit;
        signal Ou : integer;
        begin
            main: gcd
                 port map (clk,
                           reset,
                           dout,
                           Xi,
                           Yi,
                           Ou,
                           start,
                           din);

            --
            -- Waveforms for inputs
            --
            TB : block
            begin
                waves : process
                begin
```

```
            wait for 80 ns;
            start <= '1';

            wait for 20 ns;
            Xi <= 20;
            Yi <= 15;
            din <= '1';

            wait for 50 ns;
            din <= '0';

            wait for 250 ns;
            Xi <= 34;
            Yi <= 12;
            din <= '1';

            wait for 40 ns;
            din <= '0';

            wait for 300 ns;

            end process waves;

        --
        -- The Clock
        --
         myclock:
            clk <= '0' after 10 ns when clk = '1' else
                   '1' after 10 ns;

            reset <= '0' after 10 ns , '1' after 100 ns;
    end block;
end struct;

configuration cfg_gcd of gcd_tb is
    for behavior
    end for;
end cfg_gcd;
```

Figure C.2: *Description du programme test.*

La simulation d'une telle description est montrée dans la figure C.3. Cette simulation a été exécutée avec Synopsys [83].

Les résultats du calcul sont donnés immédiatement après que les valeurs en entrées aient été validées par *din*. En effet le calcul du pgcd de chaque paire prend un délai nul (unité de temps delta) pour exécuter les opérations. La notion de cycle d'horloge n'intervient pas à ce niveau.

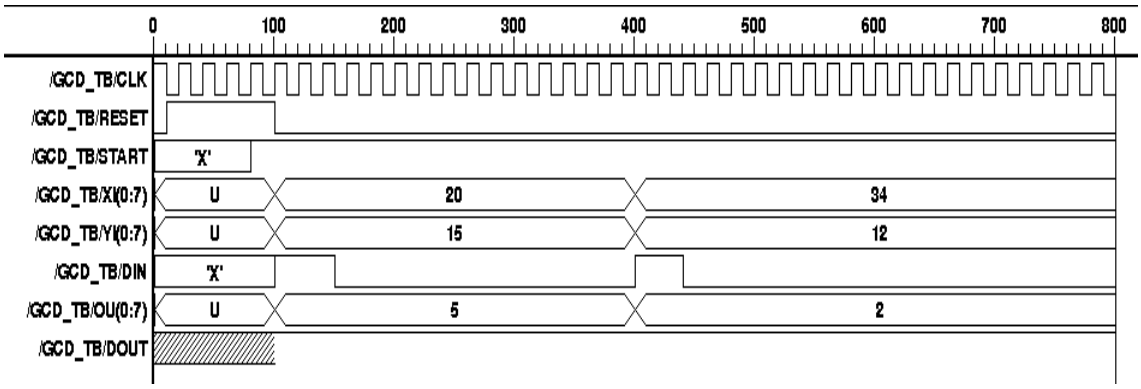


Figure C.3: *Simulation comportementale.*

C.4 Ordonnancement

L'outil d'ordonnancement d'AMICAL prend en entrée la description comportementale compilée et produit une machine d'états finis comportementale présentée sous forme de table de transitions. Cette table est sauvegardée dans un fichier dont l'extension est "*.mac*" (*gcd.mac*); des informations additionnelles telles que le nombre de bits pour la représentation de chacun des types utilisés (pour le pgcd, le seul type dont la longueur en bits est le type entier) sont sauvegardées dans un fichier dont l'extension est "*.mac.conv*" (*gcd.mac.conv*). L'outil d'ordonnancement utilise une méthode à base de chemins (*path-based method*) où il essaie de maximiser le coût du parallélisme des opérations dans chaque transition.

C.5 Synthèse

Pour la synthèse de la partie opérative, AMICAL prend en entrée trois sortes de fichier. Le premier correspond à la description comportementale ordonnancée, appelée description à étapes de contrôle (*gcd.mac*). Le deuxième type correspond aux fichiers décrivant les caractéristiques des unités fonctionnelles; chaque nouvelle unité fonctionnelle doit être introduite par l'utilisateur. Le dernier fichier correspond au fichier de technologie (*amical.technology*) contenant des informations concernant les types et les contraintes de chaque composant.

Un fichier dont l'extension "*.lib*" (*gcd.lib*) décrit la bibliothèque de composants disponibles; il contient le nom des unités fonctionnelles qui peuvent être utilisées dans la partie opérative. Les opérations qui peuvent être exécutées par chacune des unités fonctionnelles sont données dans ce fichier (figure C.4).

```
%*************************************************************%
%*** Fichier bibliotheque ******** Nom du fichier : gcd.lib ***%
%*************************************************************%

            (BIBLIOTHEQUE res.gcd
                 (PATH ./Library)
                 (FUNCTIONAL_UNIT
                      IO (OPERATEUR in out)
                      ADD (OPERATEUR +)
                      SUB (OPERATEUR -)
                      AS (OPERATEUR + -)
                 )
            )
```

Figure C.4: *Fichier décrivant la bibliothèque.*

Les caractéristiques de chaque unité fonctionnelle sont décrites dans un fichier différent ("*.fu*"); sont décrits dans ce fichier les interfaces, les appels de paramètres, les opérations exécutées par cette unité fonctionnelle ainsi que le mode d'exécution de chaque opération (nombre de cycles). La figure C.5 montre les fichiers des différentes unités fonctionnelles exécutant les opérations mentionnées dans le fichier "*.lib*" et pouvant être utilisées dans la structure de la partie opérative du pgcd.

```
%************************************************************%
%*****************  Unites Fonctionnelles  ******************%
%************************************************************%

      %************************************************%
      %*** Unite d'addition  ******  Fichier : ADD.fu ***%
      %************************************************%

      (FUNCTIONAL_UNIT ADD
         (SURFACE 1500) (LARGEUR 15) (HAUTEUR 100)
         (PARAMETRE (DONNEE_ENTREE a b) (DONNEE_SORTIE c))
         (CONNECTEUR
            (DONNEE_ENTREE in1 (BIT 0 8) in2 (BIT 0 8))
            (DONNEE_SORTIE out1 (BIT 0 8))
            (CONTROLE_ENTREE Sel (BIT 0 1)))
         (OPERATEUR + (COMMUTATIF a b)
            (CYCLE 1
               (TRANSFERT a in1)
               (TRANSFERT b in2)
               (TRANSFERT out1 c)
               (VALABLE Sel (VALEUR 1)(PENDANT 1))))
      )

%************************************************************%
%*** Unite d'addition/soustraction ********** Fichier : AS.fu ***%
%************************************************************%

       (FUNCTIONAL_UNIT AS
          (SURFACE 2000) (LARGEUR 20) (HAUTEUR 100)
          (PARAMETRE (DONNEE_ENTREE a b) (DONNEE_SORTIE c))
          (CONNECTEUR
             (DONNEE_ENTREE in1 (BIT 0 8) in2 (BIT 0 8))
             (DONNEE_SORTIE out1 (BIT 0 8))
             (CONTROLE_ENTREE Sel (BIT 0 2)))
          (OPERATEUR + (COMMUTATIF a b)
             (CYCLE 1
                (TRANSFERT a in1)
                (TRANSFERT b in2)
                (TRANSFERT out1 c)
                (VALABLE Sel (VALEUR 1) (PENDANT 1))))
          (OPERATEUR -
             (CYCLE 1
                (TRANSFERT a in1)
                (TRANSFERT b in2)
                (TRANSFERT out1 c)
                (VALABLE Sel (VALEUR 2) (PENDANT 1))))
       )

%************************************************************%
%***   Unite de soustraction   ******   Fichier : SUB.fu   ***%
%************************************************************%

     (FUNCTIONAL_UNIT SUB
        (SURFACE 1800) (LARGEUR 18) (HAUTEUR 100)
        (PARAMETRE (DONNEE_ENTREE a b) (DONNEE_SORTIE c))
        (CONNECTEUR
           (DONNEE_ENTREE in1 (BIT 0 8) in2 (BIT 0 8))
           (DONNEE_SORTIE out1 (BIT 0 8))
           (CONTROLE_ENTREE Sel (BIT 0 1)))
        (OPERATEUR -
           (CYCLE 1
              (TRANSFERT a in1)
              (TRANSFERT b in2)
              (TRANSFERT out1 c)
              (VALABLE Sel (VALEUR 1) (PENDANT 1))))
     )
```

```
%************************************************************%
%***   Unite d'entree/sortie   *******   Fichier : IO.fu   ***%
%************************************************************%

       (FUNCTIONAL_UNIT IO
          (SURFACE 2000) (LARGEUR 20) (HAUTEUR 100)
          (PARAMETRE (DONNEE_ENTREE a) (DONNEE_SORTIE c))
          (CONNECTEUR
             (DONNEE_ENTREE in1 (BIT 0 8))
             (DONNEE_SORTIE out1 (BIT 0 8))
             (CONTROLE_ENTREE Sel (BIT 0 1)))
          (OPERATEUR in
             (CYCLE 1
                (TRANSFERT a in1)
                (TRANSFERT out1 c)
                (VALABLE Sel (VALEUR 1) (PENDANT 1))))
          (OPERATEUR out
             (CYCLE 1
                (TRANSFERT a in1)
                (TRANSFERT out1 c)
                (VALABLE Sel (VALEUR 1) (PENDANT 1))))
       )
```

Figure C.5: *Bibliothèque des unités fonctionnelles utilisées pour le pgcd.*

Le format de la sortie d'AMICAL obtenue à ce stade est montré dans le figure C.6.

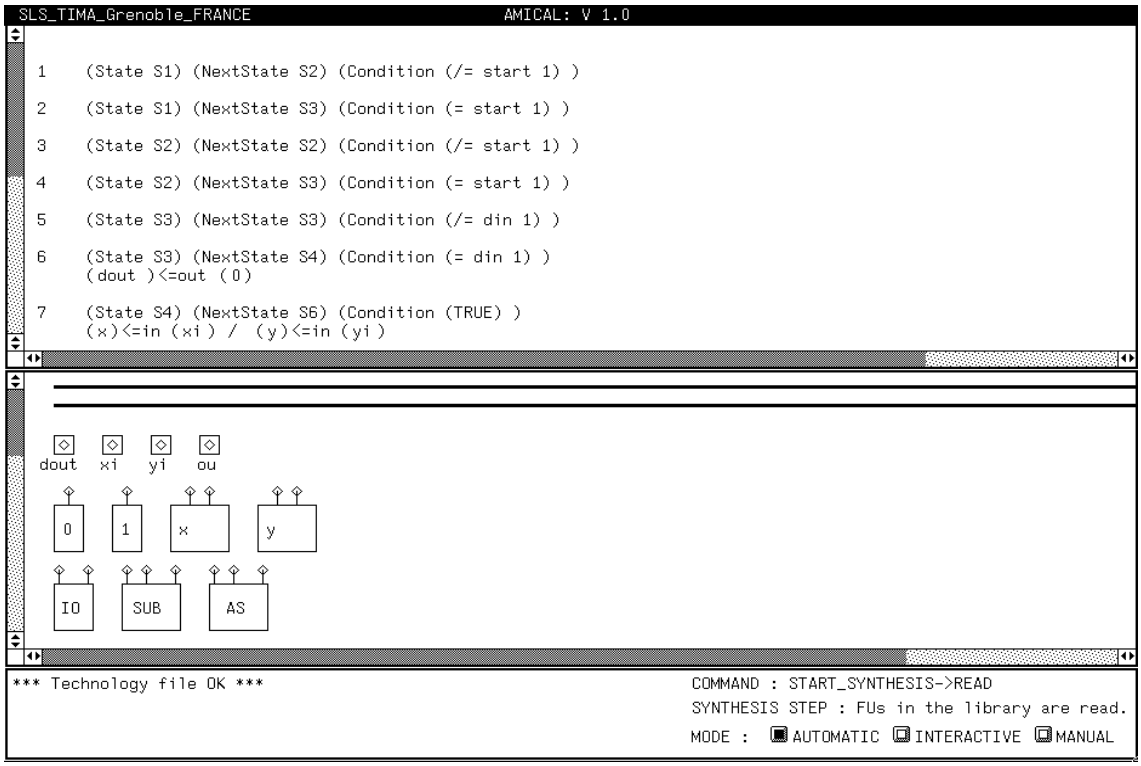


Figure C.6: *Sortie d'AMICAL avant l'allocation.*

La fenêtre du haut présente la table de transitions générée par l'ordonnancement.

Cette table contient six états et douze transitions (seulement sept transitions sont visibles ici). La fenêtre du centre montre les différentes unités fonctionnelles (*IO*, *SUB*, *AS*), les différents registres (*0*, *1*, *x*, *y*), les différents connecteurs externes (*dout*, *xi*, *yi*, *ou*) et les bus pouvant former la *netlist* de la partie opérative.

C.6 Allocation

Avec AMICAL, l'allocation peut se faire, soit automatiquement, soit pas à pas, soit manuellement. Elle invoque trois étapes essentielles: l'allocation des unités fonctionnelles, le placement des composants et l'allocation des connexions.

L'étape de l'allocation des unités fonctionnelles associe une unité fonctionnelle à chaque opération des étapes de contrôle. En mode automatique, la duplication des unités fonctionnelles est nécessaire pour permettre plus de parallélisme. La bibliothèque des unités fonctionnelles peut contenir plus d'unités que nécessaire. De plus, les unités fonctionnelles peuvent contenir des opérations qui ne sont pas utilisées. Dans l'exemple du pgcd, la seule opération nécessaire est la soustraction, cependant, la bibliothèque contient un soustracteur, un additionneur et une autre unité fonctionnelle pouvant réaliser les deux opérations (AS). Parce que la surface de l'unité AS est plus grande que celle du soustracteur, ce dernier est choisi pendant l'allocation des unités fonctionnelles quand celle-ci est lancée automatiquement.

Le placement des composants, exécuté automatiquement, place les registres et les unités fonctionnelles de sorte à optimiser les connexions. Cette tâche peut aussi être exécutée manuellement; dans ce cas l'utilisateur peut déplacer les unités fonctionnelles et les registres et les mettre dans l'ordre qu'il veut (place relative à une dimension).

L'allocation des connexions produit la structure des bus. Pour chaque transfert, un ensemble de connexions contenant des signaux, des bus et des connecteurs (*switchs*) doit être alloué de telle sorte que les transferts parallèles ne génèrent pas de conflits en se partageant les mêmes ressources. Quand cette étape est exécutée manuellement, le système s'assure que tous les transferts parallèles doivent utiliser des chemins différents.

La sortie d'AMICAL, après l'étape d'allocation, est montrée dans la figure C.7.

Suivant le mode d'exécution de chaque unité fonctionnelle (sur 1 cycle de base ou

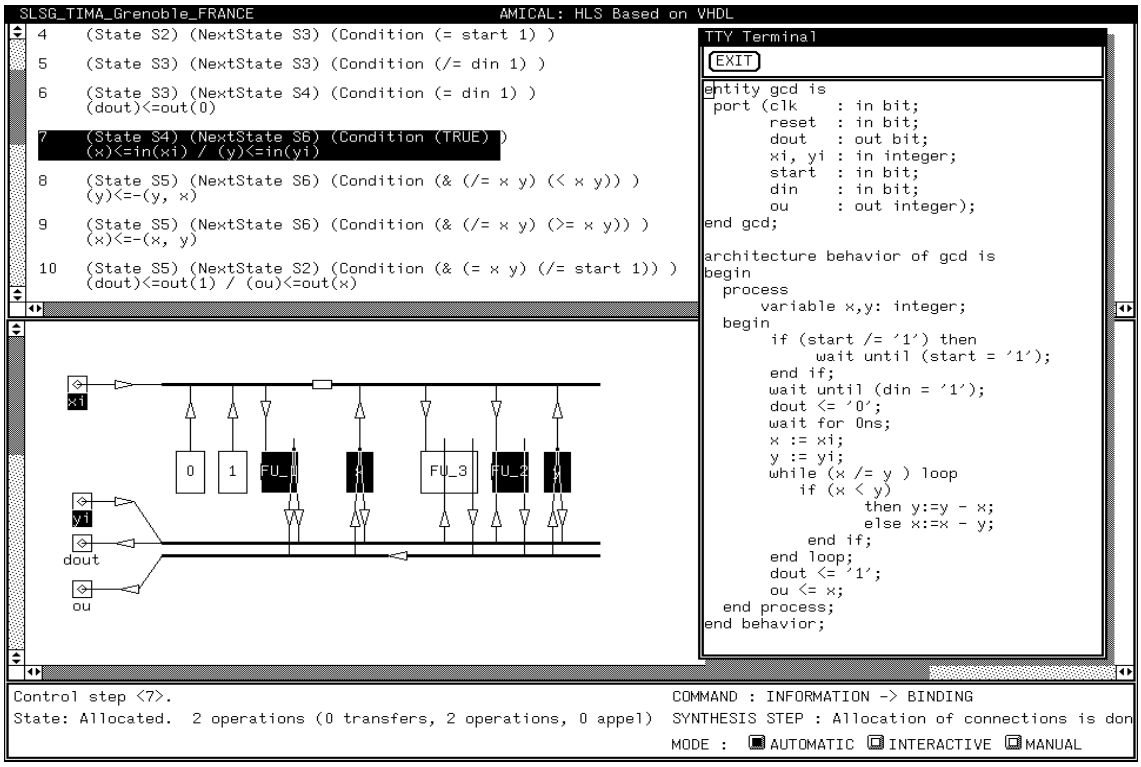


Figure C.7: *Sortie d'AMICAL après l'allocation.*

plus), la structure de la partie opérative peut changer. Le modèle montré dans cette figure est obtenu pour des unités fonctionnelles s'exécutant sur un cycle de base.

Le système AMICAL permet de vérifier le taux d'utilisation de chaque composant alloué grâce à un fichier de statistiques généré automatiquement (figure C.8).

```
%*********************************************************************%
%*** Fichier des statistiques ***** Fichier : amical_statistics ***%
%*********************************************************************%

      (Statistics of the synthesized data path
         (Filename of macro-cycle description : gcd6)
         (Filename of FUs library : gcd)
         (Total number of macro-cycles : 12)
         (Total number of micro-cycles : 12)
         (Variable Registers (Total number : 0)
         )
         (Constant Registers (Total number : 2)
            (Name 0 (Reading : 1) (Writing : 0))
            (Name 1 (Reading : 2) (Writing : 0))
         )
         (Flag Registers (Total number : 2)
            (Name x (Reading : 4) (Writing : 2))
            (Name y (Reading : 2) (Writing : 2))
         )
         (External Ports (Total number : 4)
```

```
      (Name dout (Active Cycle 3 (Rate 25.00%)))
      (Name xi   (Active Cycle 1 (Rate 8.33%)))
      (Name yi   (Active Cycle 1 (Rate 8.33%)))
      (Name ou   (Active Cycle 2 (Rate 16.67%)))
   )
   (FUs (Total Number : 3)
      (Name FU_1 (Active Cycle 8 (Rate 66.67%)))
      (Name FU_2 (Active Cycle 6 (Rate 50.00%)))
      (Name FU_3 (Active Cycle 4 (Rate 33.33%)))
   )
   (Switches (Total number : 23)
      (Name S_23 (Active Cycle 3 (Rate 25.00%)))
      (Name S_22 (Active Cycle 2 (Rate 16.67%)))
      (Name S_21 (Active Cycle 2 (Rate 16.67%)))
      (Name S_20 (Active Cycle 1 (Rate 8.33%)))
      (Name S_19 (Active Cycle 1 (Rate 8.33%)))
      (Name S_18 (Active Cycle 3 (Rate 25.00%)))
      (Name S_17 (Active Cycle 3 (Rate 25.00%)))
      (Name S_16 (Active Cycle 1 (Rate 8.33%)))
      (Name S_15 (Active Cycle 1 (Rate 8.33%)))
      (Name S_14 (Active Cycle 2 (Rate 16.67%)))
      (Name S_13 (Active Cycle 2 (Rate 16.67%)))
      (Name S_12 (Active Cycle 2 (Rate 16.67%)))
      (Name S_11 (Active Cycle 2 (Rate 16.67%)))
      (Name S_10 (Active Cycle 2 (Rate 16.67%)))
      (Name S_9  (Active Cycle 2 (Rate 16.67%)))
      (Name S_8  (Active Cycle 3 (Rate 25.00%)))
      (Name S_7  (Active Cycle 3 (Rate 25.00%)))
      (Name S_6  (Active Cycle 4 (Rate 33.33%)))
      (Name S_5  (Active Cycle 4 (Rate 33.33%)))
      (Name S_4  (Active Cycle 1 (Rate 8.33%)))
      (Name S_3  (Active Cycle 2 (Rate 16.67%)))
      (Name S_2  (Active Cycle 3 (Rate 25.00%)))
      (Name S_1  (Active Cycle 3 (Rate 25.00%)))
   )
   (Bus (Total number : 5)
      (Name BUS_1_1 (Active Cycle 12 (Rate 45.00%)))
      (Name BUS_1_2 (Active Cycle 4 (Rate 13.45%)))
      (Name BUS_2_1 (Active Cycle 8 (Rate 68.67%)))
      (Name BUS_3_1 (Active Cycle 8 (Rate 76.32%)))
      (Name BUS_3_1 (Active Cycle 8 (Rate 56.74%)))
   )
)
```

Figure C.8: *Fichier de statistiques du pgcd.*

De même, AMICAL génère un fichier d'évaluation donnant les caractéristiques de chaque unité fonctionnelle, ainsi que l'estimation de la surface totale du circuit (figure C.9).

```
%*********************************************************************%
%***  Fichier d'evaluation  *****  Fichier : amical_evaluation  ***%
%*********************************************************************%

      (Evaluation of the synthesized data path
         (Filename of macro-cycle description : gcd)
         (Filename of FUs library : gcd)
         (Allocated registers (number : 4) (area : 2500.00)
            (Variable register :)
```

```
      (Constant register : <0> <1>)
      (Flag register : <x> <y>)
      (MAX_NUMBER 20) (WEIGHT 1)
      (ESTIMATION on register allocation : SUCCESS)
   )
   (Allocated FUs (number : 3) (area 5800.00)
      (Allocated FU <FU_1> == <IO> (area : 2000.00))
      (Allocated FU <FU_2> == <IO> (area : 2000.00))
      (Allocated FU <FU_3> == <SUB> (area : 1800.00))
      (MAX_NUMBER 10) (WEIGHT 1)
      (ESTIMATION on FU allocation : SUCCESS)
   )
   (Allocated connections (area 17430.00)
      (Allocated busses (number : 3)
         (BUS_1 : <BUS_1_1> <BUS_1_2>)
         (BUS_2 : <BUS_2_1>)
         (BUS_3 : <BUS_3_1> <BUS_3_2>)
         (MAX_NUMBER 3) (WEIGHT 10)
         (ESTIMATION on bus allocation : SUCCESS)
      )
      (Allocated switches (number : 23)
         (MAX_NUMBER 50) (WEIGHT 1)
         (ESTIMATION on switch allocation : SUCCESS)
      )
   )
   (Scheduled micro-cycles (number : 6)
      (MAX_NUMBER 100) (WEIGHT 1)
      (ESTIMATION on micro_cycle scheduling : SUCCESS)
   )
   (Total area (area : 25730.00)
      (MAX_AREA 50000.00) (WEIGHT 10)
      (ESTIMATION on total area : SUCCESS)
   )
   (FINAL_ESTIMATION : SUCCESS)
)
```

Figure C.9: *Fichier d'évaluation du pgcd.*

C.7 Génération d'architecture

Après la synthèse de haut niveau, AMICAL génère, en sortie, trois fichiers donnés en SOLAR (**.solar*); ces fichiers sont nommés: *gcd_control.solar*, *gcd_datapath.solar* et *gcd_circuit.solar*. Ces fichiers incluent la spécification du contrôleur, l'interconnexion entre les éléments de la partie opérative et l'interconnexion entre la partie opérative et la partie contrôle spécifiant le circuit global.

Les trois fichiers SOLAR de l'exemple du pgcd sont donnés par les figures suivantes (figure C.10, figure C.11, figure C.12):

```
%*************************************************************************************%
%***    Partie controle    ***********************    Fichier : gcd_control.solar    ***%
%*************************************************************************************%

(SOLAR AMICAL_for_Controller
   (DesignUnit gcd_control
      (View AbstractArchitecture (ViewType "behavior")
         (Interface
            (Port start (Direction IN) (Bit) (Property PortType CONTROL))
            (Port din (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_23_dout_BUS_2_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_22_xi_BUS_1_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_21_yi_BUS_2_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_20_ou_BUS_3_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_1_BUS_1_1_0 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_2_BUS_1_1_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port CTRL_FU_1_Sel (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_3_BUS_1_1_in1_FU_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_4_out1_FU_1_BUS_3_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_5_out1_FU_1_BUS_2_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_6_BUS_1_1_BUS_1_2 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_8_x_BUS_3_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_7_BUS_1_2_x (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port CTRL_R_x (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port CTRL_W_x (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_9_x_BUS_2_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_10_BUS_3_1_BUS_3_2 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port CTRL_FU_3_Sel (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_11_BUS_1_2_in1_FU_3 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_12_in2_FU_3_BUS_2_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_13_out1_FU_3_BUS_3_2 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port CTRL_FU_2_Sel (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_15_in1_FU_2_BUS_2_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_14_BUS_1_2_in1_FU_2 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_16_out1_FU_2_BUS_3_2 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_18_y_BUS_3_2 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_17_BUS_1_2_y (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port CTRL_R_y (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port CTRL_W_y (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port S_19_y_BUS_2_1 (Direction OUT) (Bit) (Property PortType CONTROL))
            (Port (Array FLAG_x 8) (Direction IN) (Property PortType DATA))
            (Port (Array FLAG_y 8) (Direction IN) (Property PortType DATA))
         )
         (Contents
            (StateTable Controller
```

```
(State S1
   (Case
      (Alt (/= start '1')
         (NextState S2)
      )
      (Alt (= start '1')
         (NextState S3)
      )
   )
)
(State S2
   (Case
      (Alt (/= start '1')
         (NextState S2)
      )
      (Alt (= start '1')
         (NextState S3)
      )
   )
)
(State S3
   (Case
      (Alt (/= din '1')
         (NextState S3)
      )
      (Alt (= din '1')
         ; (MicroCycle 1)
         ; (Path 0 S_1 BUS_1_1 S_3 in1_FU_1)
         ; (Path out1_FU_1 S_5 BUS_2_1 S_23 dout)
         (Assign CTRL_FU_1_Sel '1')
         (Assign S_1_BUS_1_1_0 '1')
         (Assign S_3_BUS_1_1_in1_FU_1 '1')
         (Assign S_23_dout_BUS_2_1 '1')
         (Assign S_5_out1_FU_1_BUS_2_1 '1')
         (NextState S4)
      )
   )
)
(State S4
   (Case
      (Alt (TRUE)
         ; (MicroCycle 1)
         ; (Path xi S_22 BUS_1_1 S_3 in1_FU_1)
         ; (Path yi S_21 BUS_2_1 S_15 in1_FU_2)
         ; (Path out1_FU_1 S_4 BUS_3_1 S_8 x)
         ; (Path out1_FU_2 S_16 BUS_3_2 S_18 y)
         (Assign CTRL_FU_1_Sel '1')
         (Assign CTRL_FU_2_Sel '1')
         (Assign S_22_xi_BUS_1_1 '1')
         (Assign S_3_BUS_1_1_in1_FU_1 '1')
         (Assign S_21_yi_BUS_2_1 '1')
         (Assign S_15_in1_FU_2_BUS_2_1 '1')
         (Assign CTRL_R_x '0')
         (Assign CTRL_W_x '1')
         (Assign S_4_out1_FU_1_BUS_3_1 '1')
         (Assign S_8_x_BUS_3_1 '1')
         (Assign CTRL_R_y '0')
         (Assign CTRL_W_y '1')
         (Assign S_16_out1_FU_2_BUS_3_2 '1')
         (Assign S_18_y_BUS_3_2 '1')
         (NextState S6)
      )
   )
)
(State S6
   (Case
      (Alt (TRUE)
```

```
         (NextState S5)
      )
   )
)
(State S5
   (Case
      (Alt (& (/= FLAG_x FLAG_y) (< FLAG_x FLAG_y))
         ; (MicroCycle 1)
         ; (Path y S_17 BUS_1_2 S_11 in1_FU_3)
         ; (Path x S_9 BUS_2_1 S_12 in2_FU_3)
         ; (Path out1_FU_3 S_13 BUS_3_2 S_18 y)
         (Assign CTRL_FU_3_Sel '1')
         (Assign CTRL_R_y '1')
         (Assign CTRL_W_y '1')
         (Assign S_11_BUS_1_2_in1_FU_3 '1')
         (Assign S_17_BUS_1_2_y '1')
         (Assign CTRL_R_x '1')
         (Assign CTRL_W_x '0')
         (Assign S_9_x_BUS_2_1 '1')
         (Assign S_12_in2_FU_3_BUS_2_1 '1')
         (Assign S_13_out1_FU_3_BUS_3_2 '1')
         (Assign S_18_y_BUS_3_2 '1')
         (NextState S6)
      )
      (Alt (& (/= FLAG_x FLAG_y) (>= FLAG_x FLAG_y))
         ; (MicroCycle 1)
         ; (Path x S_7 BUS_1_2 S_11 in1_FU_3)
         ; (Path y S_19 BUS_2_1 S_12 in2_FU_3)
         ; (Path out1_FU_3 S_13 BUS_3_2 S_10 BUS_3_1 S_8 x)
         (Assign CTRL_FU_3_Sel '1')
         (Assign CTRL_R_x '1')
         (Assign CTRL_W_x '1')
         (Assign S_7_BUS_1_2_x '1')
         (Assign S_11_BUS_1_2_in1_FU_3 '1')
         (Assign CTRL_R_y '1')
         (Assign CTRL_W_y '0')
         (Assign S_12_in2_FU_3_BUS_2_1 '1')
         (Assign S_19_y_BUS_2_1 '1')
         (Assign S_8_x_BUS_3_1 '1')
         (Assign S_10_BUS_3_1_BUS_3_2 '1')
         (Assign S_13_out1_FU_3_BUS_3_2 '1')
         (NextState S6)
      )
      (Alt (& (= FLAG_x FLAG_y) (/= start '1'))
         ; (MicroCycle 1)
         ; (Path 1 S_2 BUS_1_1 S_3 in1_FU_1)
         ; (Path x S_7 BUS_1_2 S_14 in1_FU_2)
         ; (Path out1_FU_1 S_5 BUS_2_1 S_23 dout)
         ; (Path out1_FU_2 S_16 BUS_3_2 S_10 BUS_3_1 S_20 ou)
         (Assign CTRL_FU_1_Sel '1')
         (Assign CTRL_FU_2_Sel '1')
         (Assign S_2_BUS_1_1_1 '1')
         (Assign S_3_BUS_1_1_in1_FU_1 '1')
         (Assign CTRL_R_x '1')
         (Assign CTRL_W_x '0')
         (Assign S_7_BUS_1_2_x '1')
         (Assign S_14_BUS_1_2_in1_FU_2 '1')
         (Assign S_23_dout_BUS_2_1 '1')
         (Assign S_5_out1_FU_1_BUS_2_1 '1')
         (Assign S_20_ou_BUS_3_1 '1')
         (Assign S_10_BUS_3_1_BUS_3_2 '1')
         (Assign S_16_out1_FU_2_BUS_3_2 '1')
         (NextState S2)
      )
      (Alt (& (= FLAG_x FLAG_y) (= start '1'))
         ; (MicroCycle 1)
         ; (Path 1 S_2 BUS_1_1 S_3 in1_FU_1)
```

```
                    ; (Path x S_7 BUS_1_2 S_14 in1_FU_2)
                    ; (Path out1_FU_1 S_5 BUS_2_1 S_23 dout)
                    ; (Path out1_FU_2 S_16 BUS_3_2 S_10 BUS_3_1 S_20 ou)
                    (Assign CTRL_FU_1_Sel '1')
                    (Assign CTRL_FU_2_Sel '1')
                    (Assign S_2_BUS_1_1_1 '1')
                    (Assign S_3_BUS_1_1_in1_FU_1 '1')
                    (Assign CTRL_R_x '1')
                    (Assign CTRL_W_x '0')
                    (Assign S_7_BUS_1_2_x '1')
                    (Assign S_14_BUS_1_2_in1_FU_2 '1')
                    (Assign S_23_dout_BUS_2_1 '1')
                    (Assign S_5_out1_FU_1_BUS_2_1 '1')
                    (Assign S_20_ou_BUS_3_1 '1')
                    (Assign S_10_BUS_3_1_BUS_3_2 '1')
                    (Assign S_16_out1_FU_2_BUS_3_2 '1')
                    (NextState S3)
                )
            )
        )
        (StateList
            S1 S2 S3 S4 S6 S5)
    )
)
```

Figure C.10: *Fichier SOLAR du contrôleur du pgcd.*

```
%*************************************************************************************************%
%***    Partie operative    *********************************    Fichier : gcd_datapath.solar   ***%
%*************************************************************************************************%


(SOLAR AMICAL_for_DataPath
   (DesignUnit gcd_datapath
      (View AbstractArchitecture (ViewType "Structure")
         (Interface
            (Port (Array EBUS_1 8) (Direction OUT) (Property PortType DATA) (Property Resolved BusX))
            (Port (Array EBUS_2 8) (Direction IN) (Property PortType DATA) (Property Resolved BusX))
            (Port (Array EBUS_3 8) (Direction IN) (Property PortType DATA) (Property Resolved BusX))
            (Port EBUS_4 (Direction OUT) (Bit) (Property PortType DATA))
            (Port (Array FLAG_y 8) (Direction OUT) (Property PortType DATA))
            (Port (Array FLAG_x 8) (Direction OUT) (Property PortType DATA))
            (Port S_23_dout_BUS_2_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_22_xi_BUS_1_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_21_yi_BUS_2_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_20_ou_BUS_3_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_1_BUS_1_1_0 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_2_BUS_1_1_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port CTRL_FU_1_Sel (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_3_BUS_1_1_in1_FU_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_4_out1_FU_1_BUS_3_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_5_out1_FU_1_BUS_2_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_6_BUS_1_1_BUS_1_2 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_8_x_BUS_3_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_7_BUS_1_2_x (Direction IN) (Bit) (Property PortType CONTROL))
            (Port CTRL_R_x (Direction IN) (Bit) (Property PortType CONTROL))
            (Port CTRL_W_x (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_9_x_BUS_2_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_10_BUS_3_1_BUS_3_2 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port CTRL_FU_3_Sel (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_11_BUS_1_2_in1_FU_3 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_12_in2_FU_3_BUS_2_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_13_out1_FU_3_BUS_3_2 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port CTRL_FU_2_Sel (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_15_in1_FU_2_BUS_2_1 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_14_BUS_1_2_in1_FU_2 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_16_out1_FU_2_BUS_3_2 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_18_y_BUS_3_2 (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_17_BUS_1_2_y (Direction IN) (Bit) (Property PortType CONTROL))
            (Port CTRL_R_y (Direction IN) (Bit) (Property PortType CONTROL))
            (Port CTRL_W_y (Direction IN) (Bit) (Property PortType CONTROL))
            (Port S_19_y_BUS_2_1 (Direction IN) (Bit) (Property PortType CONTROL))
         )
         (Contents
            (Instance ou (ViewRef Implementation ExtCon_ou)
               (PortInstance E_IN (Property PortType DATA))
               (PortInstance E_OUT (Property PortType DATA)))
            (Instance yi (ViewRef Implementation ExtCon_yi)
               (PortInstance E_IN (Property PortType DATA))
               (PortInstance E_OUT (Property PortType DATA)))
            (Instance xi (ViewRef Implementation ExtCon_xi)
               (PortInstance E_IN (Property PortType DATA))
               (PortInstance E_OUT (Property PortType DATA)))
            (Instance dout (ViewRef Implementation ExtCon_dout)
               (PortInstance E_IN (Property PortType DATA))
               (PortInstance E_OUT (Property PortType DATA)))
            (Instance FU_2 (ViewRef Implementation FU_IO)
               (PortInstance in1 (Property PortType DATA))
               (PortInstance out1 (Property PortType DATA))
               (PortInstance Sel (Property PortType CONTROL))
               (Property PlaceOrder 6))
```

```
(Instance FU_3 (ViewRef Implementation FU_SUB)
   (PortInstance in1 (Property PortType DATA))
   (PortInstance in2 (Property PortType DATA))
   (PortInstance out1 (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL))
   (Property PlaceOrder 5))
(Instance FU_1 (ViewRef Implementation FU_IO)
   (PortInstance in1 (Property PortType DATA))
   (PortInstance out1 (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL))
   (Property PlaceOrder 3))
(Instance C1 (ViewRef Implementation ConstReg)
   (PortInstance Data (Property PortType DATA))
   (Property PlaceOrder 2)
   (Property ConstValue 1))
(Instance C0 (ViewRef Implementation ConstReg)
   (PortInstance Data (Property PortType DATA))
   (Property PlaceOrder 1)
   (Property ConstValue 0))
(Instance y (ViewRef Implementation FlagReg)
   (PortInstance Data_IN (Property PortType DATA))
   (PortInstance Data_OUT (Property PortType DATA))
   (PortInstance CR (Property PortType DATA))
   (PortInstance R (Property PortType CONTROL))
   (PortInstance W (Property PortType CONTROL))
   (Property PlaceOrder 7))
(Instance x (ViewRef Implementation FlagReg)
   (PortInstance Data_IN (Property PortType DATA))
   (PortInstance Data_OUT (Property PortType DATA))
   (PortInstance CR (Property PortType DATA))
   (PortInstance R (Property PortType CONTROL))
   (PortInstance W (Property PortType CONTROL))
   (Property PlaceOrder 4))
(Instance S_23 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_22 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_21 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_20 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_10 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_6 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_19 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_18 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_17 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
```

```
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_16 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_15 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_14 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_13 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_12 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_11 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_9 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_8 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_7 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_5 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_4 (ViewRef Implementation Switch)
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_3 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_2 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Instance S_1 (ViewRef Implementation Switch)
   (PortInstance S_IN (Property PortType DATA))
   (PortInstance S_OUT (Property PortType DATA))
   (PortInstance Sel (Property PortType CONTROL)))
(Net BUS_1_1 (Joined
   (PortRef S_IN (InstanceRef S_6))
   (PortRef S_OUT (InstanceRef S_22))
   (PortRef S_IN (InstanceRef S_3))
   (PortRef S_OUT (InstanceRef S_2))
   (PortRef S_OUT (InstanceRef S_1))
))
(Net BUS_1_2 (Joined
   (PortRef S_OUT (InstanceRef S_6))
```

```
   (PortRef S_OUT (InstanceRef S_17))
   (PortRef S_IN (InstanceRef S_14))
   (PortRef S_IN (InstanceRef S_11))
   (PortRef S_OUT (InstanceRef S_7))
))
(Net BUS_2_1 (Joined
   (PortRef S_IN (InstanceRef S_23))
   (PortRef S_OUT (InstanceRef S_21))
   (PortRef S_OUT (InstanceRef S_19))
   (PortRef S_IN (InstanceRef S_15))
   (PortRef S_IN (InstanceRef S_12))
   (PortRef S_OUT (InstanceRef S_9))
   (PortRef S_OUT (InstanceRef S_5))
))
(Net BUS_3_1 (Joined
   (PortRef S_OUT (InstanceRef S_10))
   (PortRef S_IN (InstanceRef S_20))
   (PortRef S_IN (InstanceRef S_8))
   (PortRef S_OUT (InstanceRef S_4))
))
(Net BUS_3_2 (Joined
   (PortRef S_IN (InstanceRef S_10))
   (PortRef S_IN (InstanceRef S_18))
   (PortRef S_OUT (InstanceRef S_16))
   (PortRef S_OUT (InstanceRef S_13))
))
(Net Net_EBUS_1 (Joined
   (PortRef EBUS_1)
   (PortRef E_OUT (InstanceRef ou))
))
(Net Net_EBUS_2 (Joined
   (PortRef EBUS_2)
   (PortRef E_IN (InstanceRef yi))
))
(Net Net_EBUS_3 (Joined
   (PortRef EBUS_3)
   (PortRef E_IN (InstanceRef xi))
))
(Net Net_EBUS_4 (Joined
   (PortRef EBUS_4)
   (PortRef E_OUT (InstanceRef dout))
))
(Net Net_FLAG_y (Joined
   (PortRef FLAG_y)
   (PortRef CR (InstanceRef y))
))
(Net Net_FLAG_x (Joined
   (PortRef FLAG_x)
   (PortRef CR (InstanceRef x))
))
(Net Net_S_23_dout_BUS_2_1 (Joined
   (PortRef S_23_dout_BUS_2_1)
   (PortRef Sel (InstanceRef S_23))
))
(Net Net_S_22_xi_BUS_1_1 (Joined
   (PortRef S_22_xi_BUS_1_1)
   (PortRef Sel (InstanceRef S_22))
))
(Net Net_S_21_yi_BUS_2_1 (Joined
   (PortRef S_21_yi_BUS_2_1)
   (PortRef Sel (InstanceRef S_21))
))
(Net Net_S_20_ou_BUS_3_1 (Joined
   (PortRef S_20_ou_BUS_3_1)
   (PortRef Sel (InstanceRef S_20))
))
(Net Net_S_1_BUS_1_1_0 (Joined
```

```
   (PortRef S_1_BUS_1_1_0)
   (PortRef Sel (InstanceRef S_1))
))
(Net Net_S_2_BUS_1_1_1 (Joined
   (PortRef S_2_BUS_1_1_1)
   (PortRef Sel (InstanceRef S_2))
))
(Net Net_CTRL_FU_1_Sel (Joined
   (PortRef CTRL_FU_1_Sel)
   (PortRef Sel (InstanceRef FU_1))
))
(Net Net_S_3_BUS_1_1_in1_FU_1 (Joined
   (PortRef S_3_BUS_1_1_in1_FU_1)
   (PortRef Sel (InstanceRef S_3))
))
(Net Net_S_4_out1_FU_1_BUS_3_1 (Joined
   (PortRef S_4_out1_FU_1_BUS_3_1)
   (PortRef Sel (InstanceRef S_4))
))
(Net Net_S_5_out1_FU_1_BUS_2_1 (Joined
   (PortRef S_5_out1_FU_1_BUS_2_1)
   (PortRef Sel (InstanceRef S_5))
))
(Net Net_S_6_BUS_1_1_BUS_1_2 (Joined
   (PortRef S_6_BUS_1_1_BUS_1_2)
   (PortRef Sel (InstanceRef S_6))
))
(Net Net_S_8_x_BUS_3_1 (Joined
   (PortRef S_8_x_BUS_3_1)
   (PortRef Sel (InstanceRef S_8))
))
(Net Net_S_7_BUS_1_2_x (Joined
   (PortRef S_7_BUS_1_2_x)
   (PortRef Sel (InstanceRef S_7))
))
(Net Net_CTRL_R_x (Joined
   (PortRef CTRL_R_x)
   (PortRef R (InstanceRef x))
))
(Net Net_CTRL_W_x (Joined
   (PortRef CTRL_W_x)
   (PortRef W (InstanceRef x))
))
(Net Net_S_9_x_BUS_2_1 (Joined
   (PortRef S_9_x_BUS_2_1)
   (PortRef Sel (InstanceRef S_9))
))
(Net Net_S_10_BUS_3_1_BUS_3_2 (Joined
   (PortRef S_10_BUS_3_1_BUS_3_2)
   (PortRef Sel (InstanceRef S_10))
))
(Net Net_CTRL_FU_3_Sel (Joined
   (PortRef CTRL_FU_3_Sel)
   (PortRef Sel (InstanceRef FU_3))
))
(Net Net_S_11_BUS_1_2_in1_FU_3 (Joined
   (PortRef S_11_BUS_1_2_in1_FU_3)
   (PortRef Sel (InstanceRef S_11))
))
(Net Net_S_12_in2_FU_3_BUS_2_1 (Joined
   (PortRef S_12_in2_FU_3_BUS_2_1)
   (PortRef Sel (InstanceRef S_12))
))
(Net Net_S_13_out1_FU_3_BUS_3_2 (Joined
   (PortRef S_13_out1_FU_3_BUS_3_2)
   (PortRef Sel (InstanceRef S_13))
))
```

```
(Net Net_CTRL_FU_2_Sel (Joined
   (PortRef CTRL_FU_2_Sel)
   (PortRef Sel (InstanceRef FU_2))
))
(Net Net_S_15_in1_FU_2_BUS_2_1 (Joined
   (PortRef S_15_in1_FU_2_BUS_2_1)
   (PortRef Sel (InstanceRef S_15))
))
(Net Net_S_14_BUS_1_2_in1_FU_2 (Joined
   (PortRef S_14_BUS_1_2_in1_FU_2)
   (PortRef Sel (InstanceRef S_14))
))
(Net Net_S_16_out1_FU_2_BUS_3_2 (Joined
   (PortRef S_16_out1_FU_2_BUS_3_2)
   (PortRef Sel (InstanceRef S_16))
))
(Net Net_S_18_y_BUS_3_2 (Joined
   (PortRef S_18_y_BUS_3_2)
   (PortRef Sel (InstanceRef S_18))
))
(Net Net_S_17_BUS_1_2_y (Joined
   (PortRef S_17_BUS_1_2_y)
   (PortRef Sel (InstanceRef S_17))
))
(Net Net_CTRL_R_y (Joined
   (PortRef CTRL_R_y)
   (PortRef R (InstanceRef y))
))
(Net Net_CTRL_W_y (Joined
   (PortRef CTRL_W_y)
   (PortRef W (InstanceRef y))
))
(Net Net_S_19_y_BUS_2_1 (Joined
   (PortRef S_19_y_BUS_2_1)
   (PortRef Sel (InstanceRef S_19))
))
(Net N_141 (Joined
   (PortRef E_IN (InstanceRef dout))
   (PortRef S_OUT (InstanceRef S_23))
))
(Net N_139 (Joined
   (PortRef E_OUT (InstanceRef xi))
   (PortRef S_IN (InstanceRef S_22))
))
(Net N_137 (Joined
   (PortRef E_OUT (InstanceRef yi))
   (PortRef S_IN (InstanceRef S_21))
))
(Net N_135 (Joined
   (PortRef E_IN (InstanceRef ou))
   (PortRef S_OUT (InstanceRef S_20))
))
(Net N_122 (Joined
   (PortRef Data_OUT (InstanceRef y))
   (PortRef S_IN (InstanceRef S_19))
))
(Net N_121 (Joined
   (PortRef Data_IN (InstanceRef y))
   (PortRef S_OUT (InstanceRef S_18))
))
(Net N_120 (Joined
   (PortRef S_IN (InstanceRef S_17))
   (PortRef Data_OUT (InstanceRef y))
))
(Net N_119 (Joined
   (PortRef out1 (InstanceRef FU_2))
   (PortRef S_IN (InstanceRef S_16))
```

```
            ))
            (Net N_118 (Joined
               (PortRef in1 (InstanceRef FU_2))
               (PortRef S_OUT (InstanceRef S_15))
            ))
            (Net N_117 (Joined
               (PortRef S_OUT (InstanceRef S_14))
               (PortRef in1 (InstanceRef FU_2))
            ))
            (Net N_116 (Joined
               (PortRef out1 (InstanceRef FU_3))
               (PortRef S_IN (InstanceRef S_13))
            ))
            (Net N_115 (Joined
               (PortRef in2 (InstanceRef FU_3))
               (PortRef S_OUT (InstanceRef S_12))
            ))
            (Net N_114 (Joined
               (PortRef S_OUT (InstanceRef S_11))
               (PortRef in1 (InstanceRef FU_3))
            ))
            (Net N_111 (Joined
               (PortRef Data_OUT (InstanceRef x))
               (PortRef S_IN (InstanceRef S_9))
            ))
            (Net N_110 (Joined
               (PortRef Data_IN (InstanceRef x))
               (PortRef S_OUT (InstanceRef S_8))
            ))
            (Net N_109 (Joined
               (PortRef S_IN (InstanceRef S_7))
               (PortRef Data_OUT (InstanceRef x))
            ))
            (Net N_105 (Joined
               (PortRef out1 (InstanceRef FU_1))
               (PortRef S_IN (InstanceRef S_5))
            ))
            (Net N_104 (Joined
               (PortRef out1 (InstanceRef FU_1))
               (PortRef S_IN (InstanceRef S_4))
            ))
            (Net N_103 (Joined
               (PortRef S_OUT (InstanceRef S_3))
               (PortRef in1 (InstanceRef FU_1))
            ))
            (Net N_102 (Joined
               (PortRef S_IN (InstanceRef S_2))
               (PortRef Data (InstanceRef C1))
            ))
            (Net N_101 (Joined
               (PortRef S_IN (InstanceRef S_1))
               (PortRef Data (InstanceRef C0))
            ))
         )
      )
   )
)
```

Figure C.11: *Fichier SOLAR de la partie opérative du pgcd.*

```
%*************************************************************************************************%
%***    Circuit global    ***********************************    Fichier : gcd_circuit.solar   ***%
%*************************************************************************************************%

(SOLAR AMICAL_for_Circuit
   (DesignUnit gcd_circuit
      (View AbstractArchitecture (ViewType "Structure")
         (Interface
            (Port dout (Direction OUT) (Bit) (Property PortType DATA))
            (Port (Array xi 8) (Direction IN) (Property PortType DATA)(Property Resolved BusX))
            (Port (Array yi 8) (Direction IN) (Property PortType DATA)(Property Resolved BusX))
            (Port (Array ou 8) (Direction OUT) (Property PortType DATA)(Property Resolved BusX))
            (Port start (Direction IN) (Bit) (Property PortType CONTROL))
            (Port din (Direction IN) (Bit) (Property PortType CONTROL))
         )
         (Contents
            (Instance ControlPart
               (ViewRef AbstractArchitecture gcd_1_control)
               (PortInstance start (Property PortType CONTROL))
               (PortInstance din (Property PortType CONTROL))
               (PortInstance S_23_dout_BUS_2_1 (Property PortType CONTROL))
               (PortInstance S_22_xi_BUS_1_1 (Property PortType CONTROL))
               (PortInstance S_21_yi_BUS_2_1 (Property PortType CONTROL))
               (PortInstance S_20_ou_BUS_3_1 (Property PortType CONTROL))
               (PortInstance S_1_BUS_1_1_0 (Property PortType CONTROL))
               (PortInstance S_2_BUS_1_1_1 (Property PortType CONTROL))
               (PortInstance CTRL_FU_1_Sel (Property PortType CONTROL))
               (PortInstance S_3_BUS_1_1_in1_FU_1 (Property PortType CONTROL))
               (PortInstance S_4_out1_FU_1_BUS_3_1 (Property PortType CONTROL))
               (PortInstance S_5_out1_FU_1_BUS_2_1 (Property PortType CONTROL))
               (PortInstance S_6_BUS_1_1_BUS_1_2 (Property PortType CONTROL))
               (PortInstance S_8_x_BUS_3_1 (Property PortType CONTROL))
               (PortInstance S_7_BUS_1_2_x (Property PortType CONTROL))
               (PortInstance CTRL_R_x (Property PortType CONTROL))
               (PortInstance CTRL_W_x (Property PortType CONTROL))
               (PortInstance S_9_x_BUS_2_1 (Property PortType CONTROL))
               (PortInstance S_10_BUS_3_1_BUS_3_2 (Property PortType CONTROL))
               (PortInstance CTRL_FU_3_Sel (Property PortType CONTROL))
               (PortInstance S_11_BUS_1_2_in1_FU_3 (Property PortType CONTROL))
               (PortInstance S_12_in2_FU_3_BUS_2_1 (Property PortType CONTROL))
               (PortInstance S_13_out1_FU_3_BUS_3_2 (Property PortType CONTROL))
               (PortInstance CTRL_FU_2_Sel (Property PortType CONTROL))
               (PortInstance S_15_in1_FU_2_BUS_2_1 (Property PortType CONTROL))
               (PortInstance S_14_BUS_1_2_in1_FU_2 (Property PortType CONTROL))
               (PortInstance S_16_out1_FU_2_BUS_3_2 (Property PortType CONTROL))
               (PortInstance S_18_y_BUS_3_2 (Property PortType CONTROL))
               (PortInstance S_17_BUS_1_2_y (Property PortType CONTROL))
               (PortInstance CTRL_R_y (Property PortType CONTROL))
               (PortInstance CTRL_W_y (Property PortType CONTROL))
               (PortInstance S_19_y_BUS_2_1 (Property PortType CONTROL))
               (PortInstance FLAG_x (Property PortType DATA))
               (PortInstance FLAG_y (Property PortType DATA))
            )
            (Instance DataPath
               (ViewRef AbstractArchitecture gcd_1_datapath)
               (PortInstance EBUS_1 (Property PortType DATA)(Property Resolved BusX))
               (PortInstance EBUS_2 (Property PortType DATA)(Property Resolved BusX))
               (PortInstance EBUS_3 (Property PortType DATA)(Property Resolved BusX))
               (PortInstance EBUS_4 (Property PortType DATA))
               (PortInstance FLAG_y (Property PortType DATA))
               (PortInstance FLAG_x (Property PortType DATA))
               (PortInstance S_23_dout_BUS_2_1 (Property PortType CONTROL))
               (PortInstance S_22_xi_BUS_1_1 (Property PortType CONTROL))
               (PortInstance S_21_yi_BUS_2_1 (Property PortType CONTROL))
```

```
   (PortInstance S_20_ou_BUS_3_1 (Property PortType CONTROL))
   (PortInstance S_1_BUS_1_1_0 (Property PortType CONTROL))
   (PortInstance S_2_BUS_1_1_1 (Property PortType CONTROL))
   (PortInstance CTRL_FU_1_Sel (Property PortType CONTROL))
   (PortInstance S_3_BUS_1_1_in1_FU_1 (Property PortType CONTROL))
   (PortInstance S_4_out1_FU_1_BUS_3_1 (Property PortType CONTROL))
   (PortInstance S_5_out1_FU_1_BUS_2_1 (Property PortType CONTROL))
   (PortInstance S_6_BUS_1_1_BUS_1_2 (Property PortType CONTROL))
   (PortInstance S_8_x_BUS_3_1 (Property PortType CONTROL))
   (PortInstance S_7_BUS_1_2_x (Property PortType CONTROL))
   (PortInstance CTRL_R_x (Property PortType CONTROL))
   (PortInstance CTRL_W_x (Property PortType CONTROL))
   (PortInstance S_9_x_BUS_2_1 (Property PortType CONTROL))
   (PortInstance S_10_BUS_3_1_BUS_3_2 (Property PortType CONTROL))
   (PortInstance CTRL_FU_3_Sel (Property PortType CONTROL))
   (PortInstance S_11_BUS_1_2_in1_FU_3 (Property PortType CONTROL))
   (PortInstance S_12_in2_FU_3_BUS_2_1 (Property PortType CONTROL))
   (PortInstance S_13_out1_FU_3_BUS_3_2 (Property PortType CONTROL))
   (PortInstance CTRL_FU_2_Sel (Property PortType CONTROL))
   (PortInstance S_15_in1_FU_2_BUS_2_1 (Property PortType CONTROL))
   (PortInstance S_14_BUS_1_2_in1_FU_2 (Property PortType CONTROL))
   (PortInstance S_16_out1_FU_2_BUS_3_2 (Property PortType CONTROL))
   (PortInstance S_18_y_BUS_3_2 (Property PortType CONTROL))
   (PortInstance S_17_BUS_1_2_y (Property PortType CONTROL))
   (PortInstance CTRL_R_y (Property PortType CONTROL))
   (PortInstance CTRL_W_y (Property PortType CONTROL))
   (PortInstance S_19_y_BUS_2_1 (Property PortType CONTROL))
)
(Net N_1
   (Joined (PortRef FLAG_y (InstanceRef ControlPart))
      (PortRef FLAG_y (InstanceRef DataPath))))
(Net N_2
   (Joined (PortRef FLAG_x (InstanceRef ControlPart))
      (PortRef FLAG_x (InstanceRef DataPath))))
(Net N_3
   (Joined (PortRef S_23_dout_BUS_2_1 (InstanceRef ControlPart))
      (PortRef S_23_dout_BUS_2_1 (InstanceRef DataPath))))
(Net N_4
   (Joined (PortRef S_22_xi_BUS_1_1 (InstanceRef ControlPart))
      (PortRef S_22_xi_BUS_1_1 (InstanceRef DataPath))))
(Net N_5
   (Joined (PortRef S_21_yi_BUS_2_1 (InstanceRef ControlPart))
      (PortRef S_21_yi_BUS_2_1 (InstanceRef DataPath))))
(Net N_6
   (Joined (PortRef S_20_ou_BUS_3_1 (InstanceRef ControlPart))
      (PortRef S_20_ou_BUS_3_1 (InstanceRef DataPath))))
(Net N_7
   (Joined (PortRef S_1_BUS_1_1_0 (InstanceRef ControlPart))
      (PortRef S_1_BUS_1_1_0 (InstanceRef DataPath))))
(Net N_8
   (Joined (PortRef S_2_BUS_1_1_1 (InstanceRef ControlPart))
      (PortRef S_2_BUS_1_1_1 (InstanceRef DataPath))))
(Net N_9
   (Joined (PortRef CTRL_FU_1_Sel (InstanceRef ControlPart))
      (PortRef CTRL_FU_1_Sel (InstanceRef DataPath))))
(Net N_10
   (Joined (PortRef S_3_BUS_1_1_in1_FU_1 (InstanceRef ControlPart))
      (PortRef S_3_BUS_1_1_in1_FU_1 (InstanceRef DataPath))))
(Net N_11
   (Joined (PortRef S_4_out1_FU_1_BUS_3_1 (InstanceRef ControlPart))
      (PortRef S_4_out1_FU_1_BUS_3_1 (InstanceRef DataPath))))
(Net N_12
   (Joined (PortRef S_5_out1_FU_1_BUS_2_1 (InstanceRef ControlPart))
      (PortRef S_5_out1_FU_1_BUS_2_1 (InstanceRef DataPath))))
(Net N_13
   (Joined (PortRef S_6_BUS_1_1_BUS_1_2 (InstanceRef ControlPart))
      (PortRef S_6_BUS_1_1_BUS_1_2 (InstanceRef DataPath))))
```

```
(Net N_14
   (Joined (PortRef S_8_x_BUS_3_1 (InstanceRef ControlPart))
      (PortRef S_8_x_BUS_3_1 (InstanceRef DataPath))))
(Net N_15
   (Joined (PortRef S_7_BUS_1_2_x (InstanceRef ControlPart))
      (PortRef S_7_BUS_1_2_x (InstanceRef DataPath))))
(Net N_16
   (Joined (PortRef CTRL_R_x (InstanceRef ControlPart))
      (PortRef CTRL_R_x (InstanceRef DataPath))))
(Net N_17
   (Joined (PortRef CTRL_W_x (InstanceRef ControlPart))
      (PortRef CTRL_W_x (InstanceRef DataPath))))
(Net N_18
   (Joined (PortRef S_9_x_BUS_2_1 (InstanceRef ControlPart))
      (PortRef S_9_x_BUS_2_1 (InstanceRef DataPath))))
(Net N_19
   (Joined (PortRef S_10_BUS_3_1_BUS_3_2 (InstanceRef ControlPart))
      (PortRef S_10_BUS_3_1_BUS_3_2 (InstanceRef DataPath))))
(Net N_20
   (Joined (PortRef CTRL_FU_3_Sel (InstanceRef ControlPart))
      (PortRef CTRL_FU_3_Sel (InstanceRef DataPath))))
(Net N_21
   (Joined (PortRef S_11_BUS_1_2_in1_FU_3 (InstanceRef ControlPart))
      (PortRef S_11_BUS_1_2_in1_FU_3 (InstanceRef DataPath))))
(Net N_22
   (Joined (PortRef S_12_in2_FU_3_BUS_2_1 (InstanceRef ControlPart))
      (PortRef S_12_in2_FU_3_BUS_2_1 (InstanceRef DataPath))))
(Net N_23
   (Joined (PortRef S_13_out1_FU_3_BUS_3_2 (InstanceRef ControlPart))
      (PortRef S_13_out1_FU_3_BUS_3_2 (InstanceRef DataPath))))
(Net N_24
   (Joined (PortRef CTRL_FU_2_Sel (InstanceRef ControlPart))
      (PortRef CTRL_FU_2_Sel (InstanceRef DataPath))))
(Net N_25
   (Joined (PortRef S_15_in1_FU_2_BUS_2_1 (InstanceRef ControlPart))
      (PortRef S_15_in1_FU_2_BUS_2_1 (InstanceRef DataPath))))
(Net N_26
   (Joined (PortRef S_14_BUS_1_2_in1_FU_2 (InstanceRef ControlPart))
      (PortRef S_14_BUS_1_2_in1_FU_2 (InstanceRef DataPath))))
(Net N_27
   (Joined (PortRef S_16_out1_FU_2_BUS_3_2 (InstanceRef ControlPart))
      (PortRef S_16_out1_FU_2_BUS_3_2 (InstanceRef DataPath))))
(Net N_28
   (Joined (PortRef S_18_y_BUS_3_2 (InstanceRef ControlPart))
      (PortRef S_18_y_BUS_3_2 (InstanceRef DataPath))))
(Net N_29
   (Joined (PortRef S_17_BUS_1_2_y (InstanceRef ControlPart))
      (PortRef S_17_BUS_1_2_y (InstanceRef DataPath))))
(Net N_30
   (Joined (PortRef CTRL_R_y (InstanceRef ControlPart))
      (PortRef CTRL_R_y (InstanceRef DataPath))))
(Net N_31
   (Joined (PortRef CTRL_W_y (InstanceRef ControlPart))
      (PortRef CTRL_W_y (InstanceRef DataPath))))
(Net N_32
   (Joined (PortRef S_19_y_BUS_2_1 (InstanceRef ControlPart))
      (PortRef S_19_y_BUS_2_1 (InstanceRef DataPath))))
(Net Net_dout
   (Joined (PortRef dout)
      (PortRef EBUS_4 (InstanceRef DataPath))))
(Net Net_xi
   (Joined (PortRef xi)
      (PortRef EBUS_3 (InstanceRef DataPath))))
(Net Net_yi
   (Joined (PortRef yi)
      (PortRef EBUS_2 (InstanceRef DataPath))))
(Net Net_ou
```

```
            (Joined (PortRef ou)
               (PortRef EBUS_1 (InstanceRef DataPath))))
         (Net Net_start
            (Joined (PortRef start)
               (PortRef start (InstanceRef ControlPart))))
         (Net Net_din
            (Joined (PortRef din)
               (PortRef din (InstanceRef ControlPart))))
      )
   )
)
```

Figure C.12: *Fichier SOLAR de la configuration globale du pgcd.*

C.8 Génération des fichiers VHDL

Les fichiers SOLAR générés par AMICAL peuvent être traduits en leurs équivalents VHDL. Cette tâche est assurée par PAT (Programmable Architecture Translator) qui, de plus, permet de personnaliser l'architecture abstraite par l'ajout de signaux globaux (signaux de sychronisation, de remise à zéro, etc.) et de blocs additionnels (blocs de synchronisation, etc.).

Avant d'exécuter PAT, l'utilisateur a besoin de spécifier les interfaces des composants, utilisés par la partie opérative, dans des fichiers SOLAR (**.solar*; figure C.13) dont le but est de permettre la transition des composants abstraits vers les composants VHDL, ainsi que tous les signaux globaux et les blocs additionnels dans un fichier "*global*" (*gcd.global*; figure C.14).

```
%*****************************************************************************%
%********************  Bibliotheque des composants SOLAR  ********************%
%*****************************************************************************%
   %*************************************************************************%
   %***  Generateur d'horloge  *********  Nom du fichier : clock.solar  ***%
   %*************************************************************************%

          (SOLAR clock
            (DESIGNUNIT Clock
              (VIEW implementation (VIEWTYPE "communicating")
                (INTERFACE
                  (PORT clk               (DIRECTION IN)    (BIT))
                  (PORT reset             (DIRECTION IN)    (BIT))
                  (PORT cont_clk          (DIRECTION OUT)   (BIT))
                  (PORT cont_reset        (DIRECTION OUT)   (BIT))
                  (PORT data_clk          (DIRECTION OUT)   (BIT))
                  (PORT data_reset        (DIRECTION OUT)   (BIT))
                )
              )
            )
          )

%*****************************************************************************%
%***   Registre Constante   *******   Nom du fichier : ConstReg.solar   ***%
%*****************************************************************************%

(SOLAR Constant
  (DESIGNUNIT ConstReg
    (VIEW implementation (VIEWTYPE "communicating")
      (INTERFACE
        (PORT (ARRAY Data 8) (DIRECTION OUT) (BIT) (Property Resolved BusX))
      )
    )
  )
)
```

```
%*************************************************************************%
%***  Connecteur externe  ********  Nom du fichier : ExtCon_dout.solar  ***%
%*************************************************************************%

(SOLAR inputoutput
  (DESIGNUNIT ExtCon_dout
    (VIEW implementation (VIEWTYPE "communicating")
      (INTERFACE
        (PORT (ARRAY E_IN 8) (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT E_OUT          (DIRECTION OUT) (BIT) (Property Resolved BusX))
      )
    )
  )
)

%*************************************************************************%
%***  Connecteur externe  **********  Nom du fichier : ExtCon_ou.solar  ***%
%*************************************************************************%

(SOLAR inputoutput
  (DESIGNUNIT ExtCon_ou
    (VIEW implementation (VIEWTYPE "communicating")
      (INTERFACE
        (PORT (ARRAY E_IN 8) (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY E_OUT 8)(DIRECTION OUT) (BIT) (Property Resolved BusX))
      )
    )
  )
)

%*************************************************************************%
%***  Connecteur externe  **********  Nom du fichier : ExtCon_xi.solar  ***%
%*************************************************************************%

(SOLAR inputoutput
  (DESIGNUNIT ExtCon_xi
    (VIEW implementation (VIEWTYPE "communicating")
      (INTERFACE
        (PORT (ARRAY E_IN 8) (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY E_OUT 8)(DIRECTION OUT) (BIT) (Property Resolved BusX))
      )
    )
  )
)

%*************************************************************************%
%***  Connecteur externe  **********  Nom du fichier : ExtCon_yi.solar  ***%
%*************************************************************************%

(SOLAR inputoutput
  (DESIGNUNIT ExtCon_yi
    (VIEW implementation (VIEWTYPE "communicating")
      (INTERFACE
        (PORT (ARRAY E_IN 8) (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY E_OUT 8)(DIRECTION OUT) (BIT) (Property Resolved BusX))
      )
    )
  )
)

%*************************************************************************%
%***  Unite d'addition  ***************  Nom du fichier : FU_ADD.solar  ***%
%*************************************************************************%

(SOLAR FU_ADD
  (DESIGNUNIT FU_ADD
    (VIEW implementation (VIEWTYPE "communicating")
```

```
      (INTERFACE
        (PORT clk            (DIRECTION IN)  (BIT))
        (PORT reset          (DIRECTION IN)  (BIT))
        (PORT Sel            (DIRECTION IN)  (BIT))
        (PORT (ARRAY in1 8)  (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY in2 8)  (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY out1 8) (DIRECTION OUT) (BIT) (Property Resolved BusX))
      )
    )
  )
)

%************************************************************************%
%*** Unite d'addition/sostraction ******* Nom du fichier : FU_AS.solar  ***%
%************************************************************************%

(SOLAR FU_AS
  (DESIGNUNIT FU_AS
    (VIEW implementation (VIEWTYPE "communicating")
      (INTERFACE
        (PORT clk            (DIRECTION IN) (BIT))
        (PORT reset          (DIRECTION IN) (BIT))
        (PORT (ARRAY Sel 2) (DIRECTION IN)  (BIT))
        (PORT (ARRAY in1 8) (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY in2 8) (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY out1 8)(DIRECTION OUT) (BIT) (Property Resolved BusX))
      )
    )
  )
)

%************************************************************************%
%***  Unite d'entree/sortie  **********  Nom du fichier : FU_IO.solar  ***%
%************************************************************************%

(SOLAR FU_IO
  (DESIGNUNIT FU_IO
    (VIEW implementation (VIEWTYPE "communicating")
      (INTERFACE
        (PORT clk            (DIRECTION IN)  (BIT))
        (PORT Sel            (DIRECTION IN)  (BIT))
        (PORT (ARRAY in1 8)  (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY out1 8) (DIRECTION OUT) (BIT) (Property Resolved BusX))
      )
    )
  )
)

%************************************************************************%
%***  Unite de soustraction  **********  Nom du fichier : FU_SUB.solar  ***%
%************************************************************************%

(SOLAR FU_SUB
  (DESIGNUNIT FU_SUB
    (VIEW implementation (VIEWTYPE "communicating")
      (INTERFACE
        (PORT clk            (DIRECTION IN)  (BIT))
        (PORT Sel            (DIRECTION IN)  (BIT))
        (PORT (ARRAY in1 8)  (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY in2 8)  (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY out1 8) (DIRECTION OUT) (BIT) (Property Resolved BusX))
      )
    )
  )
)
```

```
%*****************************************************************************%
%*** Registre de compte-rendu ************** Nom du fichier : FlagReg.solar ***%
%*****************************************************************************%

(SOLAR register
  (DESIGNUNIT FlagReg
    (VIEW implementation (VIEWTYPE "communicating")
      (INTERFACE
        (PORT res                (DIRECTION IN)  (BIT))
        (PORT clock              (DIRECTION IN)  (BIT))
        (PORT R                  (DIRECTION IN)  (BIT))
        (PORT W                  (DIRECTION IN)  (BIT))
        (PORT (ARRAY Data_IN 8)  (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY Data_OUT 8) (DIRECTION OUT) (BIT) (Property Resolved BusX))
        (PORT (ARRAY CR 8)       (DIRECTION OUT) (BIT))
      )
    )
  )
)

%*****************************************************************************%
**%***  Connecteur  *********************  Nom du fichier : Switch.solar  ***%
%*****************************************************************************%

(SOLAR switch
  (DESIGNUNIT Switch
    (VIEW port_only (VIEWTYPE "communicating")
      (INTERFACE
        (PORT clk             (DIRECTION IN)  (BIT))
        (PORT (ARRAY S_IN 8)  (DIRECTION IN)  (BIT) (Property Resolved BusX))
        (PORT (ARRAY S_OUT 8) (DIRECTION OUT) (BIT) (Property Resolved BusX))
        (PORT Sel             (DIRECTION IN)  (BIT))
      )
    )
  )
)
```

Figure C.13: *Des fichiers SOLAR des éléments de la bibliothèque du pgcd.*

```
%*****************************************************************************%
%***  Fichier global  *****************  Fichier : gcd.global  **************%
%*****************************************************************************%

 ( global GCD.global
    ( block gcd_control
         ( signal cont_clk   (dir IN) (attr clock) (type mvl7) (local clk))
         ( signal cont_reset (dir IN) (attr reset) (type mvl7) (local reset))
    )

    ( block gcd_datapath
         ( signal data_clk (dir IN) (attr clock) (type mvl7) (local clk))
         ( signal data_reset (dir IN) (attr reset) (type mvl7) (local reset))
    )
    ( block gcd_circuit
         ( signal clk ( dir IN ) ( attr clock ) ( type mvl7 ))
         ( signal reset ( dir IN ) ( attr reset ) ( type mvl7 ))
         ( signal cont_clk ( dir INTERNAL ) ( attr reset ) ( type mvl7 ))
         ( signal cont_reset ( dir INTERNAL ) ( attr reset ) ( type mvl7 ))
         ( signal data_clk ( dir INTERNAL ) ( attr clock ) ( type mvl7 ))
         ( signal data_reset ( dir INTERNAL ) ( attr reset ) ( type mvl7 ))
         ( glue Clock )
    )
 )
```

Figure C.14: *Fichier "global" du pgcd.*

Les fichiers VHDL générés par PAT sont compatibles avec les outils de simulation et de synthèse au niveau transfert de registres. Ils sont donnés par les figures suivantes (figure C.15, figure C.16, figure C.17):

```
%*************************************************************************%
%*** partie controle *************** Fichier : gcd_control.vhd **************%
%*************************************************************************%

   library mvl_7;
   use mvl_7.TYPES.all;
   use mvl_7.arithmetic.all;
   ENTITY  gcd_control  IS
       port (cont_clk : IN mvl7;
             cont_reset : In mvl7;
             start  :  IN  mvl7;
             din  :  IN  mvl7;
             S_23_dout_BUS_2_1  :  OUT  mvl7;
             S_22_xi_BUS_1_1  :  OUT  mvl7;
             S_21_yi_BUS_2_1  :  OUT  mvl7;
             S_20_ou_BUS_3_1  :  OUT  mvl7;
             S_1_BUS_1_1_0  :  OUT  mvl7;
             S_2_BUS_1_1_1  :  OUT  mvl7;
             CTRL_FU_1_Sel  :  OUT  mvl7;
             S_3_BUS_1_1_in1_FU_1  :  OUT  mvl7;
             S_4_out1_FU_1_BUS_3_1  :  OUT  mvl7;
             S_5_out1_FU_1_BUS_2_1  :  OUT  mvl7;
             S_6_BUS_1_1_BUS_1_2  :  OUT  mvl7;
             S_8_x_BUS_3_1  :  OUT  mvl7;
             S_7_BUS_1_2_x  :  OUT  mvl7;
             CTRL_R_x  :  OUT  mvl7;
             CTRL_W_x  :  OUT  mvl7;
             S_9_x_BUS_2_1  :  OUT  mvl7;
             S_10_BUS_3_1_BUS_3_2  :  OUT  mvl7;
             CTRL_FU_3_Sel  :  OUT  mvl7;
             S_11_BUS_1_2_in1_FU_3  :  OUT  mvl7;
             S_12_in2_FU_3_BUS_2_1  :  OUT  mvl7;
             S_13_out1_FU_3_BUS_3_2  :  OUT  mvl7;
             CTRL_FU_2_Sel  :  OUT  mvl7;
             S_15_in1_FU_2_BUS_2_1  :  OUT  mvl7;
             S_14_BUS_1_2_in1_FU_2  :  OUT  mvl7;
             S_16_out1_FU_2_BUS_3_2  :  OUT  mvl7;
             S_18_y_BUS_3_2  :  OUT  mvl7;
             S_17_BUS_1_2_y  :  OUT  mvl7;
             CTRL_R_y  :  OUT  mvl7;
             CTRL_W_y  :  OUT  mvl7;
             S_19_y_BUS_2_1  :  OUT  mvl7;
             FLAG_x:  IN  mvl7_vector(0 to 7);
             FLAG_y:  IN  mvl7_vector(0 to 7)      );
   END gcd_control ;

   ARCHITECTURE   AbstractArchitecture of gcd_control  is
   type STATE_TYPE is ( S1, S2, S3, S4, S6, S5 );
   signal CURRENT_STATE,NEXT_STATE:STATE_TYPE;

   begin
       Controller  : PROCESS ( start,
                               din,
                               FLAG_x,
                               FLAG_y,
                               CURRENT_STATE)
       BEGIN

       S_23_dout_BUS_2_1  <=  '0';
```

```
S_22_xi_BUS_1_1  <=  '0';
S_21_yi_BUS_2_1  <=  '0';
S_20_ou_BUS_3_1  <=  '0';
S_1_BUS_1_1_0  <=  '0';
S_2_BUS_1_1_1  <=  '0';
CTRL_FU_1_Sel  <=  '0';
S_3_BUS_1_1_in1_FU_1  <=  '0';
S_4_out1_FU_1_BUS_3_1  <=  '0';
S_5_out1_FU_1_BUS_2_1  <=  '0';
S_6_BUS_1_1_BUS_1_2  <=  '0';
S_8_x_BUS_3_1  <=  '0';
S_7_BUS_1_2_x  <=  '0';
CTRL_R_x  <=  '0';
CTRL_W_x  <=  '0';
S_9_x_BUS_2_1  <=  '0';
S_10_BUS_3_1_BUS_3_2  <=  '0';
CTRL_FU_3_Sel  <=  '0';
S_11_BUS_1_2_in1_FU_3  <=  '0';
S_12_in2_FU_3_BUS_2_1  <=  '0';
S_13_out1_FU_3_BUS_3_2  <=  '0';
CTRL_FU_2_Sel  <=  '0';
S_15_in1_FU_2_BUS_2_1  <=  '0';
S_14_BUS_1_2_in1_FU_2  <=  '0';
S_16_out1_FU_2_BUS_3_2  <=  '0';
S_18_y_BUS_3_2  <=  '0';
S_17_BUS_1_2_y  <=  '0';
CTRL_R_y  <=  '0';
CTRL_W_y  <=  '0';
S_19_y_BUS_2_1  <=  '0';
NEXT_STATE <= S1;

CASE CURRENT_STATE is
WHEN( S1 ) => IF (  start /=  '1')
              THEN
                 NEXT_STATE <= S2;
              END IF;
              IF (  start =  '1')
              THEN
                 NEXT_STATE <= S3;
              END IF;
WHEN( S2 ) => IF (  start /=  '1')
              THEN
                 NEXT_STATE <= S2;
              END IF;
              IF (  start =  '1')
              THEN
                 NEXT_STATE <= S3;
              END IF;
WHEN( S3 ) => IF (  din /=  '1')
              THEN
                 NEXT_STATE <= S3;
              END IF;
              IF (  din =  '1')
              THEN
                 CTRL_FU_1_Sel  <=  '1';
                 S_1_BUS_1_1_0  <=  '1';
                 S_3_BUS_1_1_in1_FU_1  <=  '1';
                 S_23_dout_BUS_2_1  <=  '1';
                 S_5_out1_FU_1_BUS_2_1  <=  '1';
                 NEXT_STATE <= S4;
              END IF;
WHEN( S4 ) =>
                 CTRL_FU_1_Sel  <=  '1';
                 CTRL_FU_2_Sel  <=  '1';
                 S_22_xi_BUS_1_1  <=  '1';
                 S_3_BUS_1_1_in1_FU_1  <=  '1';
                 S_21_yi_BUS_2_1  <=  '1';
```

```
                  S_15_in1_FU_2_BUS_2_1  <=  '1';
                  CTRL_R_x  <=  '0';
                  CTRL_W_x  <=  '1';
                  S_4_out1_FU_1_BUS_3_1  <=  '1';
                  S_8_x_BUS_3_1  <=  '1';
                  CTRL_R_y  <=  '0';
                  CTRL_W_y  <=  '1';
                  S_16_out1_FU_2_BUS_3_2  <=  '1';
                  S_18_y_BUS_3_2  <=  '1';
                  NEXT_STATE <= S6;
WHEN( S6 ) =>
               NEXT_STATE <= S5;
WHEN( S5 ) => IF (((unsigned( FLAG_x ))/= (unsigned( FLAG_y )))AND
                  ((unsigned( FLAG_x ))< (unsigned( FLAG_y ))))
               THEN
                  CTRL_FU_3_Sel  <=  '1';
                  CTRL_R_y  <=  '1';
                  CTRL_W_y  <=  '1';
                  S_11_BUS_1_2_in1_FU_3  <=  '1';
                  S_17_BUS_1_2_y  <=  '1';
                  CTRL_R_x  <=  '1';
                  CTRL_W_x  <=  '0';
                  S_9_x_BUS_2_1  <=  '1';
                  S_12_in2_FU_3_BUS_2_1  <=  '1';
                  S_13_out1_FU_3_BUS_3_2  <=  '1';
                  S_18_y_BUS_3_2  <=  '1';
                  NEXT_STATE <= S6;
               END IF;
               IF (((unsigned( FLAG_x))/=(unsigned( FLAG_y )))AND
                   ((unsigned( FLAG_x))>=(unsigned( FLAG_y ))))
               THEN
                  CTRL_FU_3_Sel  <=  '1';
                  CTRL_R_x  <=  '1';
                  CTRL_W_x  <=  '1';
                  S_7_BUS_1_2_x  <=  '1';
                  S_11_BUS_1_2_in1_FU_3  <=  '1';
                  CTRL_R_y  <=  '1';
                  CTRL_W_y  <=  '0';
                  S_12_in2_FU_3_BUS_2_1  <=  '1';
                  S_19_y_BUS_2_1  <=  '1';
                  S_8_x_BUS_3_1  <=  '1';
                  S_10_BUS_3_1_BUS_3_2  <=  '1';
                  S_13_out1_FU_3_BUS_3_2  <=  '1';
                  NEXT_STATE <= S6;
               END IF;
               IF (((unsigned( FLAG_x))=(unsigned( FLAG_y)))AND
                   (start /=  '1'))
               THEN
                  CTRL_FU_1_Sel  <=  '1';
                  CTRL_FU_2_Sel  <=  '1';
                  S_2_BUS_1_1_1  <=  '1';
                  S_3_BUS_1_1_in1_FU_1  <=  '1';
                  CTRL_R_x  <=  '1';
                  CTRL_W_x  <=  '0';
                  S_7_BUS_1_2_x  <=  '1';
                  S_14_BUS_1_2_in1_FU_2  <=  '1';
                  S_23_dout_BUS_2_1  <=  '1';
                  S_5_out1_FU_1_BUS_2_1  <=  '1';
                  S_20_ou_BUS_3_1  <=  '1';
                  S_10_BUS_3_1_BUS_3_2  <=  '1';
                  S_16_out1_FU_2_BUS_3_2  <=  '1';
                  NEXT_STATE <= S2;
               END IF;
               IF (((unsigned( FLAG_x))=(unsigned( FLAG_y)))AND
                  (start =  '1'))
               THEN
                  CTRL_FU_1_Sel  <=  '1';
```

```
                    CTRL_FU_2_Sel  <=  '1';
                    S_2_BUS_1_1_1  <=  '1';
                    S_3_BUS_1_1_in1_FU_1  <=  '1';
                    CTRL_R_x  <=  '1';
                    CTRL_W_x  <=  '0';
                    S_7_BUS_1_2_x  <=  '1';
                    S_14_BUS_1_2_in1_FU_2  <=  '1';
                    S_23_dout_BUS_2_1  <=  '1';
                    S_5_out1_FU_1_BUS_2_1  <=  '1';
                    S_20_ou_BUS_3_1  <=  '1';
                    S_10_BUS_3_1_BUS_3_2  <=  '1';
                    S_16_out1_FU_2_BUS_3_2  <=  '1';
                    NEXT_STATE <= S3;
                END IF;
        END CASE ;
    END PROCESS  Controller ;

    SYNCH:PROCESS
    BEGIN
        WAIT UNTIL cont_clk = '1' AND cont_clk'EVENT;
        IF ((cont_reset = '1') THEN
            CURRENT_STATE <= S1;
        ELSE
           CURRENT_STATE <= NEXT_STATE;
        END IF;
    END PROCESS;
END AbstractArchitecture ;
```

Figure C.15: *Fichier VHDL de la partie contrôle du pgcd.*

```
%*****************************************************************************%
%*** partie operative ************** Fichier : gcd_datapath.vhd *************%
%*****************************************************************************%

            library mvl_7;
            use mvl_7.TYPES.all;
            use mvl_7.arithmetic.all;
            Entity   gcd_datapath is
                port(
                     data_clk  :  IN  mvl7;
                     data_reset  :  IN  mvl7;
                     EBUS_1:  OUT BusX(0 to 7);
                     EBUS_2:  IN BusX(0 to 7);
                     EBUS_3:  IN BusX(0 to 7);
                     EBUS_4 :  OUT mvl7;
                     FLAG_y:  OUT mvl7_vector(0 to 7);
                     FLAG_x:  OUT mvl7_vector(0 to 7);
                     S_23_dout_BUS_2_1 :  IN mvl7;
                     S_22_xi_BUS_1_1 :  IN mvl7;
                     S_21_yi_BUS_2_1 :  IN mvl7;
                     S_20_ou_BUS_3_1 :  IN mvl7;
                     S_1_BUS_1_1_0 :  IN mvl7;
                     S_2_BUS_1_1_1 :  IN mvl7;
                     CTRL_FU_1_Sel :  IN mvl7;
                     S_3_BUS_1_1_in1_FU_1 :  IN mvl7;
                     S_4_out1_FU_1_BUS_3_1 :  IN mvl7;
                     S_5_out1_FU_1_BUS_2_1 :  IN mvl7;
                     S_6_BUS_1_1_BUS_1_2 :  IN mvl7;
                     S_8_x_BUS_3_1 :  IN mvl7;
                     S_7_BUS_1_2_x :  IN mvl7;
                     CTRL_R_x :  IN mvl7;
                     CTRL_W_x :  IN mvl7;
                     S_9_x_BUS_2_1 :  IN mvl7;
                     S_10_BUS_3_1_BUS_3_2 :  IN mvl7;
                     CTRL_FU_3_Sel :  IN mvl7;
                     S_11_BUS_1_2_in1_FU_3 :  IN mvl7;
                     S_12_in2_FU_3_BUS_2_1 :  IN mvl7;
                     S_13_out1_FU_3_BUS_3_2 :  IN mvl7;
                     CTRL_FU_2_Sel :  IN mvl7;
                     S_15_in1_FU_2_BUS_2_1 :  IN mvl7;
                     S_14_BUS_1_2_in1_FU_2 :  IN mvl7;
                     S_16_out1_FU_2_BUS_3_2 :  IN mvl7;
                     S_18_y_BUS_3_2 :  IN mvl7;
                     S_17_BUS_1_2_y :  IN mvl7;
                     CTRL_R_y :  IN mvl7;
                     CTRL_W_y :  IN mvl7;
                     S_19_y_BUS_2_1 :  IN mvl7      );
            end gcd_datapath;
            architecture  AbstractArchitecture of gcd_datapath is

            signal  BUS_1_1: BusX(0 to 7) ;
            signal  BUS_1_2: BusX(0 to 7) ;
            signal  BUS_2_1: BusX(0 to 7) ;
            signal  BUS_3_1: BusX(0 to 7) ;
            signal  BUS_3_2: BusX(0 to 7) ;
            signal  N_141: BusX(0 to 7) ;
            signal  N_139: BusX(0 to 7) ;
            signal  N_137: BusX(0 to 7) ;
            signal  N_135: BusX(0 to 7) ;
            signal  N_122: BusX(0 to 7) ;
            signal  N_121: BusX(0 to 7) ;
            signal  N_119: BusX(0 to 7) ;
            signal  N_118: BusX(0 to 7) ;
            signal  N_116: BusX(0 to 7) ;
```

```
signal  N_115: BusX(0 to 7) ;
signal  N_114: BusX(0 to 7) ;
signal  N_111: BusX(0 to 7) ;
signal  N_110: BusX(0 to 7) ;
signal  N_105: BusX(0 to 7) ;
signal  N_103: BusX(0 to 7) ;
signal  N_102: BusX(0 to 7) ;
signal  N_101: BusX(0 to 7) ;

Component  ExtCon_ou
   Port ( E_IN:  IN BusX(0 to 7);
          E_OUT:  OUT BusX(0 to 7)       );
end Component;
Component  ExtCon_yi
   Port ( E_IN:  IN BusX(0 to 7);
          E_OUT:  OUT BusX(0 to 7)       );
end Component;
Component  ExtCon_xi
   Port ( E_IN:  IN BusX(0 to 7);
          E_OUT:  OUT BusX(0 to 7)       );
end Component;
Component  ExtCon_dout
   Port ( E_IN:  IN BusX(0 to 7);
          E_OUT :  OUT mvl7       );
end Component;
Component  FU_IO
   Port ( clk :  IN mvl7;
          Sel :  IN mvl7;
          in1:  IN BusX(0 to 7);
          out1:  OUT BusX(0 to 7)       );
end Component;
Component  FU_SUB
   Port ( clk :  IN mvl7;
          Sel :  IN mvl7;
          in1:  IN BusX(0 to 7);
          in2:  IN BusX(0 to 7);
          out1:  OUT BusX(0 to 7)       );
end Component;
Component  ConstReg
   generic (value: integer);
   Port ( Data:  OUT BusX(0 to 7)       );
end Component;
Component  FlagReg
   Port ( reset :  IN mvl7;
          clk :  IN mvl7;
          R :  IN mvl7;
          W :  IN mvl7;
          Data_IN:  IN BusX(0 to 7);
          Data_OUT:  OUT BusX(0 to 7);
          CR:  OUT mvl7_vector(0 to 7)       );
end Component;
Component  Switch
   Port ( clk :  IN mvl7;
          S_IN:  IN BusX(0 to 7);
          S_OUT:  OUT BusX(0 to 7);
          Sel :  IN mvl7       );
end Component;

begin
   ou :  ExtCon_ou
      Port Map( E_IN=>N_135,
                E_OUT=>EBUS_1          );
   yi :  ExtCon_yi
      Port Map( E_IN=>EBUS_2,
                E_OUT=>N_137         );
   xi :  ExtCon_xi
      Port Map( E_IN=>EBUS_3,
```

```
            E_OUT=>N_139          );
dout :  ExtCon_dout
   Port Map( E_IN=>N_141,
             E_OUT=>EBUS_4          );
FU_2 :  FU_IO
   Port Map( clk=>data_clk,
             Sel=>CTRL_FU_2_Sel,
             in1=>N_118,
             out1=>N_119          );
FU_3 :  FU_SUB
   Port Map( clk=>data_clk,
             Sel=>CTRL_FU_3_Sel,
             in1=>N_114,
             in2=>N_115,
             out1=>N_116          );
FU_1 :  FU_IO
   Port Map( clk=>data_clk,
             Sel=>CTRL_FU_1_Sel,
             in1=>N_103,
             out1=>N_105          );
C1 :  ConstReg
   generic map(1)
   Port Map( Data=>N_102          );
C0 :  ConstReg
   generic map(0)
   Port Map( Data=>N_101          );
y :  FlagReg
   Port Map( reset=>data_reset,
             clk=>data_clk,
             R=>CTRL_R_y,
             W=>CTRL_W_y,
             Data_IN=>N_121,
             Data_OUT=>N_122,
             CR=>FLAG_y          );
x :  FlagReg
   Port Map( reset=>data_reset,
             clk=>data_clk,
             R=>CTRL_R_x,
             W=>CTRL_W_x,
             Data_IN=>N_110,
             Data_OUT=>N_111,
             CR=>FLAG_x          );
S_23 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_2_1,
             S_OUT=>N_141,
             Sel=>S_23_dout_BUS_2_1          );
S_22 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>N_139,
             S_OUT=>BUS_1_1,
             Sel=>S_22_xi_BUS_1_1          );
S_21 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>N_137,
             S_OUT=>BUS_2_1,
             Sel=>S_21_yi_BUS_2_1          );
S_20 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_3_1,
             S_OUT=>N_135,
             Sel=>S_20_ou_BUS_3_1          );
S_10 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_3_2,
             S_OUT=>BUS_3_1,
             Sel=>S_10_BUS_3_1_BUS_3_2          );
```

```
S_6 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_1_1,
             S_OUT=>BUS_1_2,
             Sel=>S_6_BUS_1_1_BUS_1_2          );
S_19 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>N_122,
             S_OUT=>BUS_2_1,
             Sel=>S_19_y_BUS_2_1          );
S_18 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_3_2,
             S_OUT=>N_121,
             Sel=>S_18_y_BUS_3_2          );
S_17 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>N_122,
             S_OUT=>BUS_1_2,
             Sel=>S_17_BUS_1_2_y          );
S_16 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>N_119,
             S_OUT=>BUS_3_2,
             Sel=>S_16_out1_FU_2_BUS_3_2          );
S_15 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_2_1,
             S_OUT=>N_118,
             Sel=>S_15_in1_FU_2_BUS_2_1          );
S_14 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_1_2,
             S_OUT=>N_118,
             Sel=>S_14_BUS_1_2_in1_FU_2          );
S_13 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>N_116,
             S_OUT=>BUS_3_2,
             Sel=>S_13_out1_FU_3_BUS_3_2          );
S_12 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_2_1,
             S_OUT=>N_115,
             Sel=>S_12_in2_FU_3_BUS_2_1          );
S_11 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_1_2,
             S_OUT=>N_114,
             Sel=>S_11_BUS_1_2_in1_FU_3          );
S_9 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>N_111,
             S_OUT=>BUS_2_1,
             Sel=>S_9_x_BUS_2_1          );
S_8 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>BUS_3_1,
             S_OUT=>N_110,
             Sel=>S_8_x_BUS_3_1          );
S_7 :  Switch
   Port Map( clk=>data_clk,
             S_IN=>N_111,
             S_OUT=>BUS_1_2,
             Sel=>S_7_BUS_1_2_x          );
S_5 :  Switch
   Port Map( clk=>data_clk,
```

```
                S_IN=>N_105,
                S_OUT=>BUS_2_1,
                Sel=>S_5_out1_FU_1_BUS_2_1          );
      S_4 :  Switch
         Port Map( clk=>data_clk,
                S_IN=>N_105,
                S_OUT=>BUS_3_1,
                Sel=>S_4_out1_FU_1_BUS_3_1          );
      S_3 :  Switch
         Port Map( clk=>data_clk,
                S_IN=>BUS_1_1,
                S_OUT=>N_103,
                Sel=>S_3_BUS_1_1_in1_FU_1          );
      S_2 :  Switch
         Port Map( clk=>data_clk,
                S_IN=>N_102,
                S_OUT=>BUS_1_1,
                Sel=>S_2_BUS_1_1_1          );
      S_1 :  Switch
         Port Map( clk=>data_clk,
                S_IN=>N_101,
                S_OUT=>BUS_1_1,
                Sel=>S_1_BUS_1_1_0          );
end AbstractArchitecture;
```

Figure C.16: *Fichier VHDL de la partie opérative du pgcd.*

```
%*****************************************************************************%
%*** Circuit global ***************** Fichier : gcd_circuit.vhd **************%
%*****************************************************************************%

            library mvl_7;
            use mvl_7.TYPES.all;
            use mvl_7.arithmetic.all;
            Entity   gcd_circuit is
               port ( clk : IN mvl7;
                      resrt : IN mvl7;
                      dout :  OUT mvl7;
                      xi:  IN BusX(0 to 7);
                      yi:  IN BusX(0 to 7);
                      ou:  OUT BusX(0 to 7);
                      start :  IN mvl7;
                      din :  IN mvl7      );
            end gcd_circuit;
            architecture  AbstractArchitecture of gcd_circuit is

            signal  cont_clk:mvl7 ;
            signal  cont_reset:mvl7 ;
            signal  data_clk:mvl7 ;
            signal  data_reset:mvl7 ;
            signal  N_1: mvl7_vector(0 to 7) ;
            signal  N_2: mvl7_vector(0 to 7) ;
            signal  N_3:mvl7 ;
            signal  N_4:mvl7 ;
            signal  N_5:mvl7 ;
            signal  N_6:mvl7 ;
            signal  N_7:mvl7 ;
            signal  N_8:mvl7 ;
            signal  N_9:mvl7 ;
            signal  N_10:mvl7 ;
            signal  N_11:mvl7 ;
            signal  N_12:mvl7 ;
            signal  N_13:mvl7 ;
            signal  N_14:mvl7 ;
            signal  N_15:mvl7 ;
            signal  N_16:mvl7 ;
            signal  N_17:mvl7 ;
            signal  N_18:mvl7 ;
            signal  N_19:mvl7 ;
            signal  N_20:mvl7 ;
            signal  N_21:mvl7 ;
            signal  N_22:mvl7 ;
            signal  N_23:mvl7 ;
            signal  N_24:mvl7 ;
            signal  N_25:mvl7 ;
            signal  N_26:mvl7 ;
            signal  N_27:mvl7 ;
            signal  N_28:mvl7 ;
            signal  N_29:mvl7 ;
            signal  N_30:mvl7 ;
            signal  N_31:mvl7 ;
            signal  N_32:mvl7 ;

            Component  Clock
               Port ( clk :  IN mvl7;
                      reset :  IN mvl7;
                      cont_clk :  OUT mvl7;
                      cont_reset :  OUT mvl7;
                      data_clk :  OUT mvl7;
                      data_reset :  OUT mvl7      );
            end Component;
```

```
Component  gcd_control
   Port ( cont_clk :  in mvl7;
          cont_reset :  in mvl7;
          start :  IN mvl7;
          din :  IN mvl7;
          S_23_dout_BUS_2_1 :  OUT mvl7;
          S_22_xi_BUS_1_1 :  OUT mvl7;
          S_21_yi_BUS_2_1 :  OUT mvl7;
          S_20_ou_BUS_3_1 :  OUT mvl7;
          S_1_BUS_1_1_0 :  OUT mvl7;
          S_2_BUS_1_1_1 :  OUT mvl7;
          CTRL_FU_1_Sel :  OUT mvl7;
          S_3_BUS_1_1_in1_FU_1 :  OUT mvl7;
          S_4_out1_FU_1_BUS_3_1 :  OUT mvl7;
          S_5_out1_FU_1_BUS_2_1 :  OUT mvl7;
          S_6_BUS_1_1_BUS_1_2 :  OUT mvl7;
          S_8_x_BUS_3_1 :  OUT mvl7;
          S_7_BUS_1_2_x :  OUT mvl7;
          CTRL_R_x :  OUT mvl7;
          CTRL_W_x :  OUT mvl7;
          S_9_x_BUS_2_1 :  OUT mvl7;
          S_10_BUS_3_1_BUS_3_2 :  OUT mvl7;
          CTRL_FU_3_Sel :  OUT mvl7;
          S_11_BUS_1_2_in1_FU_3 :  OUT mvl7;
          S_12_in2_FU_3_BUS_2_1 :  OUT mvl7;
          S_13_out1_FU_3_BUS_3_2 :  OUT mvl7;
          CTRL_FU_2_Sel :  OUT mvl7;
          S_15_in1_FU_2_BUS_2_1 :  OUT mvl7;
          S_14_BUS_1_2_in1_FU_2 :  OUT mvl7;
          S_16_out1_FU_2_BUS_3_2 :  OUT mvl7;
          S_18_y_BUS_3_2 :  OUT mvl7;
          S_17_BUS_1_2_y :  OUT mvl7;
          CTRL_R_y :  OUT mvl7;
          CTRL_W_y :  OUT mvl7;
          S_19_y_BUS_2_1 :  OUT mvl7;
          FLAG_x:  IN mvl7_vector(0 to 7);
          FLAG_y:  IN mvl7_vector(0 to 7)       );
end Component;
Component  gcd_datapath
   Port ( data_clk: IN mvl7:
          data_reset: IN mvl7;
          EBUS_1:  OUT BusX(0 to 7);
          EBUS_2:  IN BusX(0 to 7);
          EBUS_3:  IN BusX(0 to 7);
          EBUS_4 :  OUT mvl7;
          FLAG_y:  OUT mvl7_vector(0 to 7);
          FLAG_x:  OUT mvl7_vector(0 to 7);
          S_23_dout_BUS_2_1 :  IN mvl7;
          S_22_xi_BUS_1_1 :  IN mvl7;
          S_21_yi_BUS_2_1 :  IN mvl7;
          S_20_ou_BUS_3_1 :  IN mvl7;
          S_1_BUS_1_1_0 :  IN mvl7;
          S_2_BUS_1_1_1 :  IN mvl7;
          CTRL_FU_1_Sel :  IN mvl7;
          S_3_BUS_1_1_in1_FU_1 :  IN mvl7;
          S_4_out1_FU_1_BUS_3_1 :  IN mvl7;
          S_5_out1_FU_1_BUS_2_1 :  IN mvl7;
          S_6_BUS_1_1_BUS_1_2 :  IN mvl7;
          S_8_x_BUS_3_1 :  IN mvl7;
          S_7_BUS_1_2_x :  IN mvl7;
          CTRL_R_x :  IN mvl7;
          CTRL_W_x :  IN mvl7;
          S_9_x_BUS_2_1 :  IN mvl7;
          S_10_BUS_3_1_BUS_3_2 :  IN mvl7;
          CTRL_FU_3_Sel :  IN mvl7;
          S_11_BUS_1_2_in1_FU_3 :  IN mvl7;
          S_12_in2_FU_3_BUS_2_1 :  IN mvl7;
```

```
            S_13_out1_FU_3_BUS_3_2 :  IN mvl7;
            CTRL_FU_2_Sel :  IN mvl7;
            S_15_in1_FU_2_BUS_2_1 :  IN mvl7;
            S_14_BUS_1_2_in1_FU_2 :  IN mvl7;
            S_16_out1_FU_2_BUS_3_2 :  IN mvl7;
            S_18_y_BUS_3_2 :  IN mvl7;
            S_17_BUS_1_2_y :  IN mvl7;
            CTRL_R_y :  IN mvl7;
            CTRL_W_y :  IN mvl7;
            S_19_y_BUS_2_1 :  IN mvl7       );
end Component;

begin
   Clockx :  Clock
      Port Map( clk=>clk,
                reset=>reset,
                cont_clk=>cont_clk,
                cont_reset=>cont_reset,
                data_clk=>data_clk,
                data_reset=>data_reset          );
   ControlPart : gcd_control
      Port Map( cont_clk=>cont_clk,
                cont_reset=>cont_reset,
                start=>start,
                din=>din,
                S_23_dout_BUS_2_1=>N_3,
                S_22_xi_BUS_1_1=>N_4,
                S_21_yi_BUS_2_1=>N_5,
                S_20_ou_BUS_3_1=>N_6,
                S_1_BUS_1_1_0=>N_7,
                S_2_BUS_1_1_1=>N_8,
                CTRL_FU_1_Sel=>N_9,
                S_3_BUS_1_1_in1_FU_1=>N_10,
                S_4_out1_FU_1_BUS_3_1=>N_11,
                S_5_out1_FU_1_BUS_2_1=>N_12,
                S_6_BUS_1_1_BUS_1_2=>N_13,
                S_8_x_BUS_3_1=>N_14,
                S_7_BUS_1_2_x=>N_15,
                CTRL_R_x=>N_16,
                CTRL_W_x=>N_17,
                S_9_x_BUS_2_1=>N_18,
                S_10_BUS_3_1_BUS_3_2=>N_19,
                CTRL_FU_3_Sel=>N_20,
                S_11_BUS_1_2_in1_FU_3=>N_21,
                S_12_in2_FU_3_BUS_2_1=>N_22,
                S_13_out1_FU_3_BUS_3_2=>N_23,
                CTRL_FU_2_Sel=>N_24,
                S_15_in1_FU_2_BUS_2_1=>N_25,
                S_14_BUS_1_2_in1_FU_2=>N_26,
                S_16_out1_FU_2_BUS_3_2=>N_27,
                S_18_y_BUS_3_2=>N_28,
                S_17_BUS_1_2_y=>N_29,
                CTRL_R_y=>N_30,
                CTRL_W_y=>N_31,
                S_19_y_BUS_2_1=>N_32,
                FLAG_x=>N_2,
                FLAG_y=>N_1          );
   DataPath : gcd_datapath
      Port Map( data_clk=>data_clk,
                data_reset=>data_reset,
                EBUS_1=>ou,
                EBUS_2=>yi,
                EBUS_3=>xi,
                EBUS_4=>dout,
                FLAG_y=>N_1,
                FLAG_x=>N_2,
                S_23_dout_BUS_2_1=>N_3,
```

```
                    S_22_xi_BUS_1_1=>N_4,
                    S_21_yi_BUS_2_1=>N_5,
                    S_20_ou_BUS_3_1=>N_6,
                    S_1_BUS_1_1_0=>N_7,
                    S_2_BUS_1_1_1=>N_8,
                    CTRL_FU_1_Sel=>N_9,
                    S_3_BUS_1_1_in1_FU_1=>N_10,
                    S_4_out1_FU_1_BUS_3_1=>N_11,
                    S_5_out1_FU_1_BUS_2_1=>N_12,
                    S_6_BUS_1_1_BUS_1_2=>N_13,
                    S_8_x_BUS_3_1=>N_14,
                    S_7_BUS_1_2_x=>N_15,
                    CTRL_R_x=>N_16,
                    CTRL_W_x=>N_17,
                    S_9_x_BUS_2_1=>N_18,
                    S_10_BUS_3_1_BUS_3_2=>N_19,
                    CTRL_FU_3_Sel=>N_20,
                    S_11_BUS_1_2_in1_FU_3=>N_21,
                    S_12_in2_FU_3_BUS_2_1=>N_22,
                    S_13_out1_FU_3_BUS_3_2=>N_23,
                    CTRL_FU_2_Sel=>N_24,
                    S_15_in1_FU_2_BUS_2_1=>N_25,
                    S_14_BUS_1_2_in1_FU_2=>N_26,
                    S_16_out1_FU_2_BUS_3_2=>N_27,
                    S_18_y_BUS_3_2=>N_28,
                    S_17_BUS_1_2_y=>N_29,
                    CTRL_R_y=>N_30,
                    CTRL_W_y=>N_31,
                    S_19_y_BUS_2_1=>N_32          );
end AbstractArchitecture;
configuration cfg_gcd of gcd_circuit is
    for AbstractArchitecture
    end for;
end cfg_gcd;
```

Figure C.17: *Fichier VHDL du circuit global du pgcd.*

C.9 Simulation de la description RTL

La simulation de la description RTL se fait en utilisant les trois fichiers VHDL générés par PAT, ainsi que les descriptions fonctionnelles, données en VHDL, des différents composants constituant la partie opérative (figure C.18).

```
%*****************************************************************************%
%*****************   Bibliotheque des composants VHDL   **********************%
%*****************************************************************************%

   %*************************************************************************%
   %*** Generateur d'horloge ********************** Fichier : Clock.vhd ***%
   %*************************************************************************%

               library mvl_7;
               use mvl_7.types.all;
               entity Clock is
                  port (clk: in MVL7 := 'Z';
                        reset: in MVL7 := 'Z';
                        cont_clk: out MVL7 := 'Z';
                        cont_reset: out MVL7 := 'Z';
                        data_clk: out MVL7 := 'Z';
                        data_reset: out MVL7 := 'Z');
               end Clock;

               architecture implementation of Clock is
               begin
                  ck: process(clk, reset)
                  begin
                     cont_reset <= reset;
                     data_reset <= reset;
                     cont_clk <= clk;
                     data_clk <= clk;
                  end process ck;
               end implementation;

   %*************************************************************************%
   %*** Registre Constante ********************** Fichier : CostReg.vhd ***%
   %*************************************************************************%

           library mvl_7;
           use mvl_7.types.all;
           use mvl_7.arithmetic.all;
           entity ConstReg is
               generic(value : integer);
               port(
                    Data  : out BusX(0 to 7) := "ZZZZZZZZ");
           end ConstReg;

           architecture Implementation of ConstReg is
           begin
                  Data <= Drive(mvl7_vector(conv_signed(value,8)));
           end Implementation;

   %*************************************************************************%
   %*** Connecteur externe ********************** Fichier : ExtCon_dout ***%
   %*************************************************************************%

             library mvl_7;
             use mvl_7.types.all;
             use mvl_7.arithmetic.all;
```

```
entity ExtCon_dout is
    port(E_IN: in BusX(0 to 7) := "ZZZZZZZZ";
         E_OUT: out mvl7 := 'Z');
end ExtCon_dout;

architecture Implementation of ExtCon_dout is
begin
   ExtCon_pro: process(E_IN)
   variable value: mvl7_vector(0 to 7) := "ZZZZZZZZ";
   begin
      value := Drive(E_IN);
      E_OUT <= value(7);
   end process ExtCon_pro;
end Implementation;
```

```
%************************************************************************%
%*** Connecteur externe ********************** Fichier : ExtCon_ou ***%
%************************************************************************%

   library mvl_7;
   use mvl_7.types.all;
   use mvl_7.arithmetic.all;
   entity ExtCon_ou is
       port(E_IN: in BusX(0 to 7) := "ZZZZZZZZ";
            E_OUT: out BusX(0 to 7) := "ZZZZZZZZ");
   end ExtCon_ou;

   architecture Implementation of ExtCon_ou is
   begin
           ExtCon_pro: process(E_IN)
           begin
                   E_OUT <= E_IN;
           end process ExtCon_pro;
   end Implementation;
```

```
%************************************************************************%
%*** Connecteur externe ********************** Fichier : ExtCon_xi ***%
%************************************************************************%

   library mvl_7;
   use mvl_7.types.all;
   use mvl_7.arithmetic.all;
   entity ExtCon_xi is
       port(E_IN: in BusX(0 to 7) := "ZZZZZZZZ";
            E_OUT: out BusX(0 to 7) := "ZZZZZZZZ");
   end ExtCon_xi;

   architecture Implementation of ExtCon_xi is
   begin
           ExtCon_pro: process(E_IN)
           begin
                   E_OUT <= E_IN;
           end process ExtCon_pro;
   end Implementation;
```

```
%************************************************************************%
%*** Connecteur externe ********************** Fichier : ExtCon_yi ***%
%************************************************************************%

   library mvl_7;
   use mvl_7.types.all;
   use mvl_7.arithmetic.all;
   entity ExtCon_yi is
       port(E_IN: in BusX(0 to 7) := "ZZZZZZZZ";
            E_OUT: out BusX(0 to 7) := "ZZZZZZZZ");
   end ExtCon_yi;
```

```
            architecture Implementation of ExtCon_yi is
            begin
                    ExtCon_pro: process(E_IN)
                    begin
                            E_OUT <= E_IN;
                    end process ExtCon_pro;
            end Implementation;

%************************************************************************%
%*** Unite d'entree/sortie ************************ Fichier : FU_IO ***%
%************************************************************************%

            library mvl_7;
            use mvl_7.types.all;
            use mvl_7.arithmetic.all;
            entity FU_IO is
                port (clk   : in mvl7;
                      Sel   : in MVL7;
                      in1   : in BusX(0 to 7) := "ZZZZZZZZ";
                      out1  : out BusX(0 to 7) := "ZZZZZZZZ");
            end FU_IO;

            architecture implementation of FU_IO is
            signal L_Sel: mvl7;
            begin
               FU_IO_file: process(clk, L_Sel, in1)
               variable val_es: mvl7_vector(0 to 7) ;
               begin
                         if L_Sel = '1' then
                            out1 <= in1;
                         else
                            out1 <= "ZZZZZZZZ";
                         end if;
               end process FU_IO_file;
               FU_IO_file2: process
               begin
                         wait until clk = '1' and clk'event;
                         L_Sel <= Sel;
               end process FU_IO_file2;
            end implementation;

%************************************************************************%
%*** Unite de soustraction ************************ Fichier : FU_SUB ***%
%************************************************************************%

 library mvl_7;
 use mvl_7.types.all;
 use mvl_7.arithmetic.all;
 entity FU_SUB is
    port (clk       : in mvl7;
          Sel       : in MVL7;
          in1, in2  : in BusX(0 to 7) := "ZZZZZZZZ";
          out1      : out BusX(0 to 7) := "ZZZZZZZZ" );
 end FU_SUB;

 architecture implementation of FU_SUB is
 signal op1, op2: MVL7_vector(0 to 7) ;
 signal L_Sel: mvl7;
 begin
    alu_file1: process (clk, L_Sel, op1, op2)
    begin
        if L_Sel = '1' then
           out1 <= Drive(MVL7_vector(unsigned(op1) - unsigned(op2)));
        else
           out1 <= "ZZZZZZZZ";
        end if;
    end process alu_file1;
```

```
    alu_file2: process
    begin
        wait until clk = '1' and clk'event;
        L_Sel <= Sel;
    end process alu_file2;
        op1 <= Drive(in1);
        op2 <= Drive(in2);
 end implementation;

%************************************************************************%
%*** registre compte-rendu ****************** Fichier : FlagReg.vhd ***%
%************************************************************************%

       library mvl_7;
       use mvl_7.types.all;
       entity FlagReg is
           port(reset, clk: in MVL7;
                R         : in MVL7;
                W         : in MVL7;
                Data_IN   : in BusX (0 to 7) := (others => 'Z');
                Data_OUT  : out BusX (0 to 7)  := (others => 'Z');
                CR        : out MVL7_vector (0 to 7)  );
       end FlagReg;

       architecture implementation of FlagReg is
          signal val_reg: MVL7_vector(0 to 7) ;
          signal L_W: mvl7;
       begin
          reg_file1: process
          begin
             wait until clk = '0' and clk'event;
             if reset='0' then
                 val_reg <= (others => '0');
             elsif L_W = '1' then         -- Write --
                 val_reg <= Drive(Data_IN);
             end if;
          end process reg_file1;
          reg_file2: process
          begin
                wait until clk = '1' and clk'event;
                if reset = '1' then
                    L_W <= W;
                end if;
          end process reg_file2;
          CR <= val_reg;
          Data_OUT <= Drive(val_reg);
       end implementation;

%************************************************************************%
%*** Connecteur ***************************** Fichier : FlagReg.vhd ***%
%************************************************************************%

            library mvl_7;
            use mvl_7.types.all;
            entity Switch is
                 port (clk: mvl7;
                       S_IN : in BusX(0 to 7) := "ZZZZZZZZ";
                       S_OUT: out BusX(0 to 7) := "ZZZZZZZZ";
                       Sel  : in MVL7       );
            end Switch;

            architecture implementation of Switch is
                signal L_Sel: mvl7;
            begin
                Switch_busses1: process( clk, L_Sel, S_IN)
                begin
                      if L_Sel = '1' then
```

```
            S_OUT <= S_IN;
        else
            S_OUT <= "ZZZZZZZZ";
        end if;
    end process Switch_busses1;
    Switch_busses2: process
    begin
        wait until clk = '1' and clk'event;
        L_Sel <= Sel;
    end process Switch_busses2;
end implementation;
```

Figure C.18: *Bibliothèque des composants VHDL.*

Le programme de stimuli utilisé pour la simulation est le même que celui utilisé pour la simulation comportementale (figure C.19); la seule différence réside dans les types utilisés: les types initiaux *integer* et *bit* ont été remplacés par *BusX(0 to 7)* et *mvl7* respectivement.

```
%****************************************************************************%
%***     Programme test    ********     Nom du fichier : gcd_tb.vhd     ***%
%****************************************************************************%

        library mvl_7;
        use mvl_7.types.all;
        entity gcd_tb is end;
        architecture struct of gcd_tb is
            component gcd
                port ( clk      : in MVL7 := 'Z';
                       reset    : in MVL7 := 'Z';
                       dout     : inout mvl7 := 'Z';
                       xi       : inout BusX(0 to 7) := (others => 'Z');
                       yi       : inout BusX(0 to 7) := (others => 'Z');
                       ou       : inout BusX(0 to 7) := (others => 'Z');
                       start    : in MVL7 := 'Z';
                       din      : in MVL7 := 'Z');
            end component;

        signal Xi,Yi : BusX(0 to 7) := "ZZZZZZZZ";
        signal clk: MVL7 := '1';
        signal reset: MVL7 := '0';
        signal start, din, dout: MVL7;
        signal Ou : BusX(0 to 7) := "ZZZZZZZZ";
        begin
            main: gcd
                 port map (clk,
                           reset,
                           dout,
                           Xi,
                           Yi,
                           Ou,
                           start,
                           din);

            --
            -- Waveforms for inputs
            --
            TB : block
            begin
                waves : process
                begin
                    wait for 80 ns;
                    start <= '1';

                    wait for 20 ns;
                    Xi <= "00010100";
                    Yi <= "00001111";
                    din <= '1';

                    wait for 50 ns;
                    din <= '0';

                    wait for 250 ns;
                    Xi <= "00100010";
                    Yi <= "00001100";
                    din <= '1';
```

```
            wait for 40 ns;
            din <= '0';

            wait for 300 ns;

            end process waves;

        --
        -- The Clock
        --
         myclock:
            clk <= '0' after 10 ns when clk = '1' else
                   '1' after 10 ns;

            reset <= '0' after 10 ns , '1' after 100 ns;
    end block;
end struct;

configuration cfg_gcd of gcd_tb is
    for struct
        for main: gcd
            use configuration work.cfg_gcd_circuit;
        end for;
    end for;
end cfg_gcd;
```

Figure C.19: *Description du programme test.*

Le résultat de la simulation RTL est donné par la figure C.20.

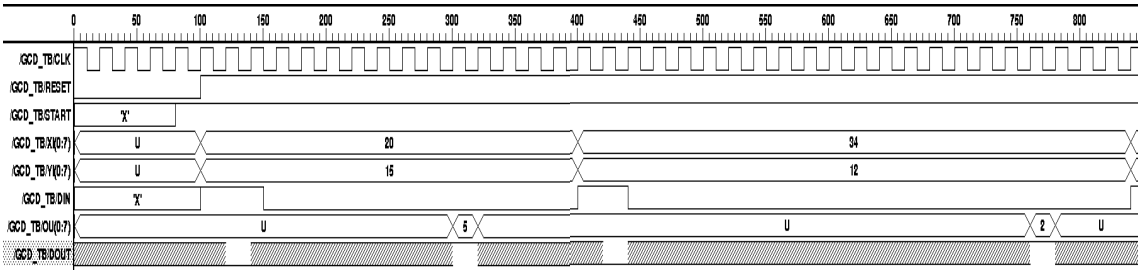


Figure C.20: *Simulation au niveau transfert de registres.*

Les résultats produits par la simulation RTL sont les mêmes que ceux trouvés par la simulation comportementale sauf qu'ils ne sont pas obtenus immédiatement après que les données d'entrée aient été introduites mais plutôt décalés dans le temps. En effet, après l'ordonnancement et le micro-ordonnancement, l'algorithme a été divisé en plusieurs étapes de calcul qui prennent chacune un cycle d'horloge. Dans ce cas, l'exécution du pgcd de la paire de données (20, 15) prend 10 cycles, tandis que la paire (34, 12) nécessite un temps de calcul de 18 cycles. Cette différence est due au nombre d'itérations supérieur dans le deuxième cas par rapport au premier.

C.10 Synthèse logique

Les unités fonctionnelles ont été synthétisées à partir de leur description VHDL utilisée pour la simulation. Dans cette section, seulement les trois fichiers VHDL générés par PAT sont présentés pour montrer la synthèse logique exécutée par Synopsys [82].

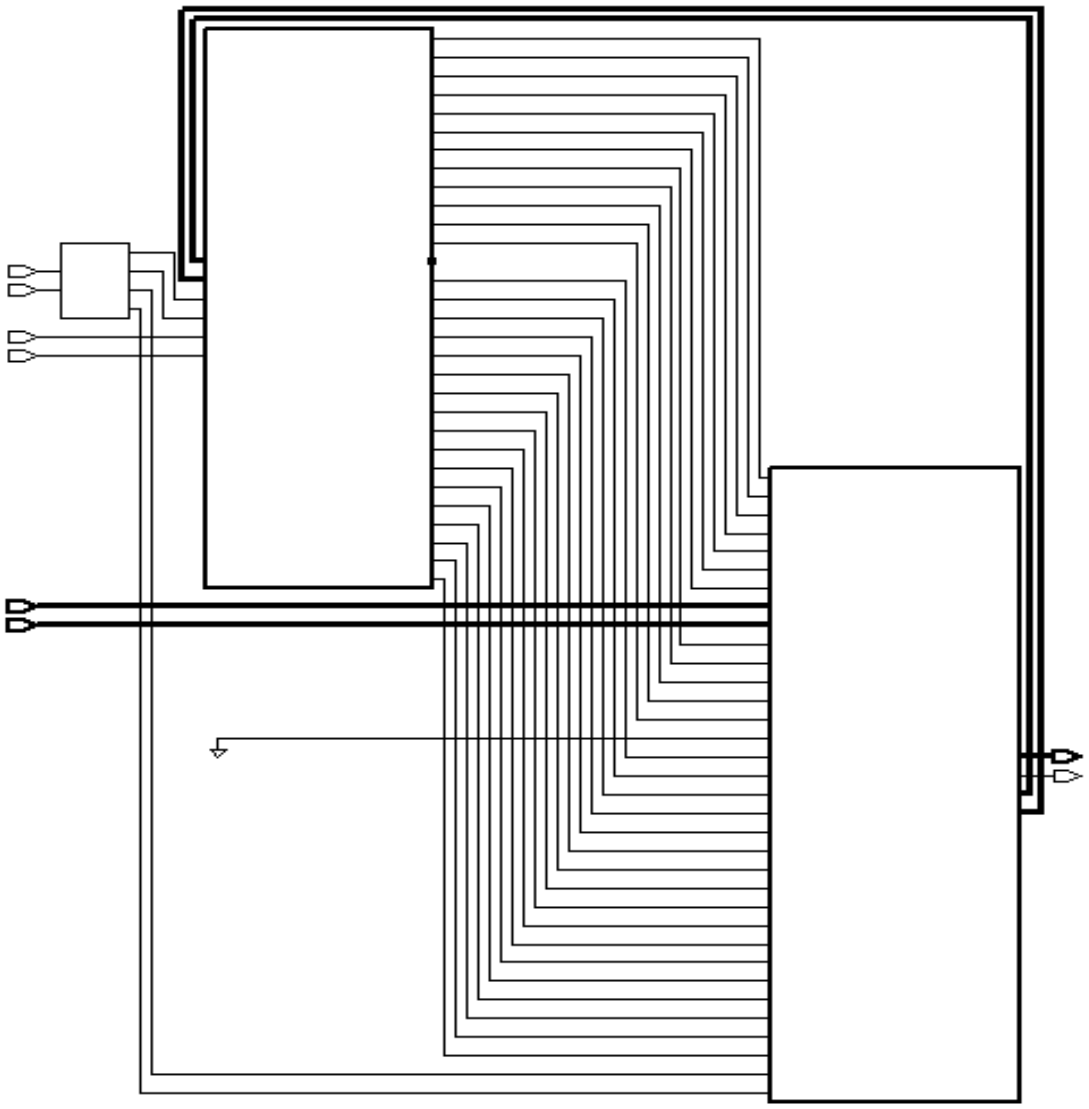


Figure C.21: *Schématique du circuit global du pgcd.*

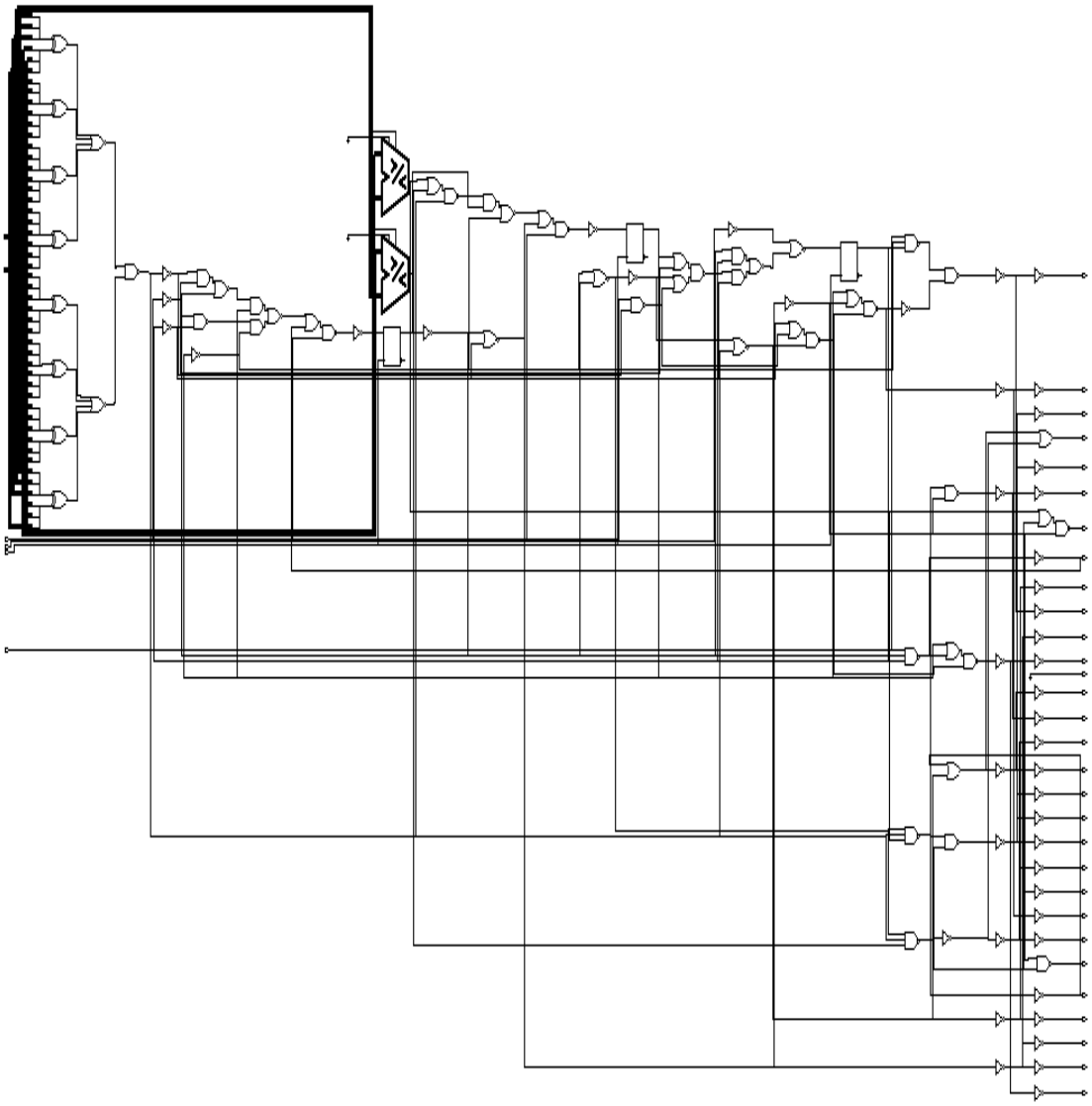


Figure C.22: *Schématique de la partie contrôle du pgcd.*

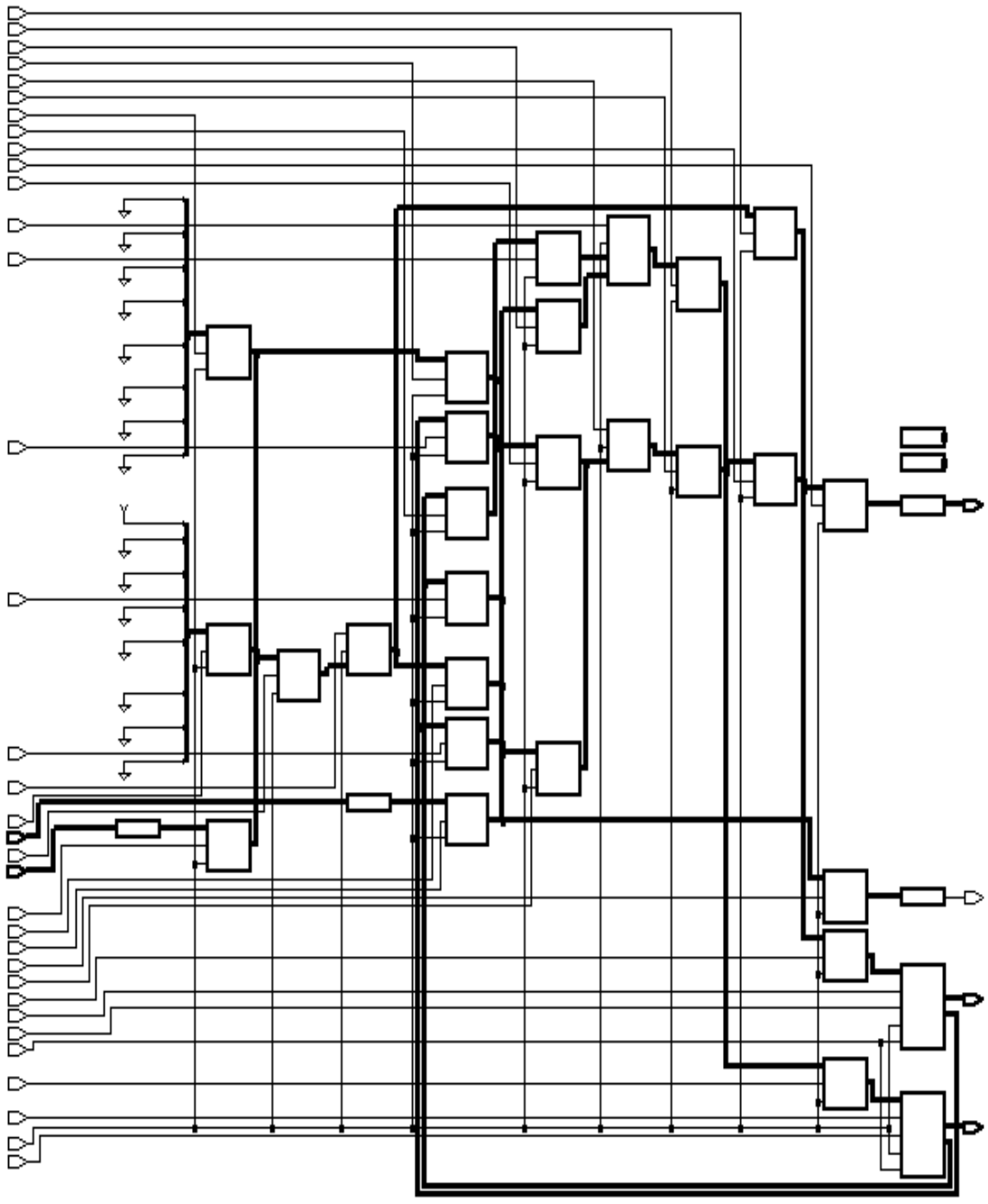


Figure C.23: *Schématique de la partie opérative du pgcd.*

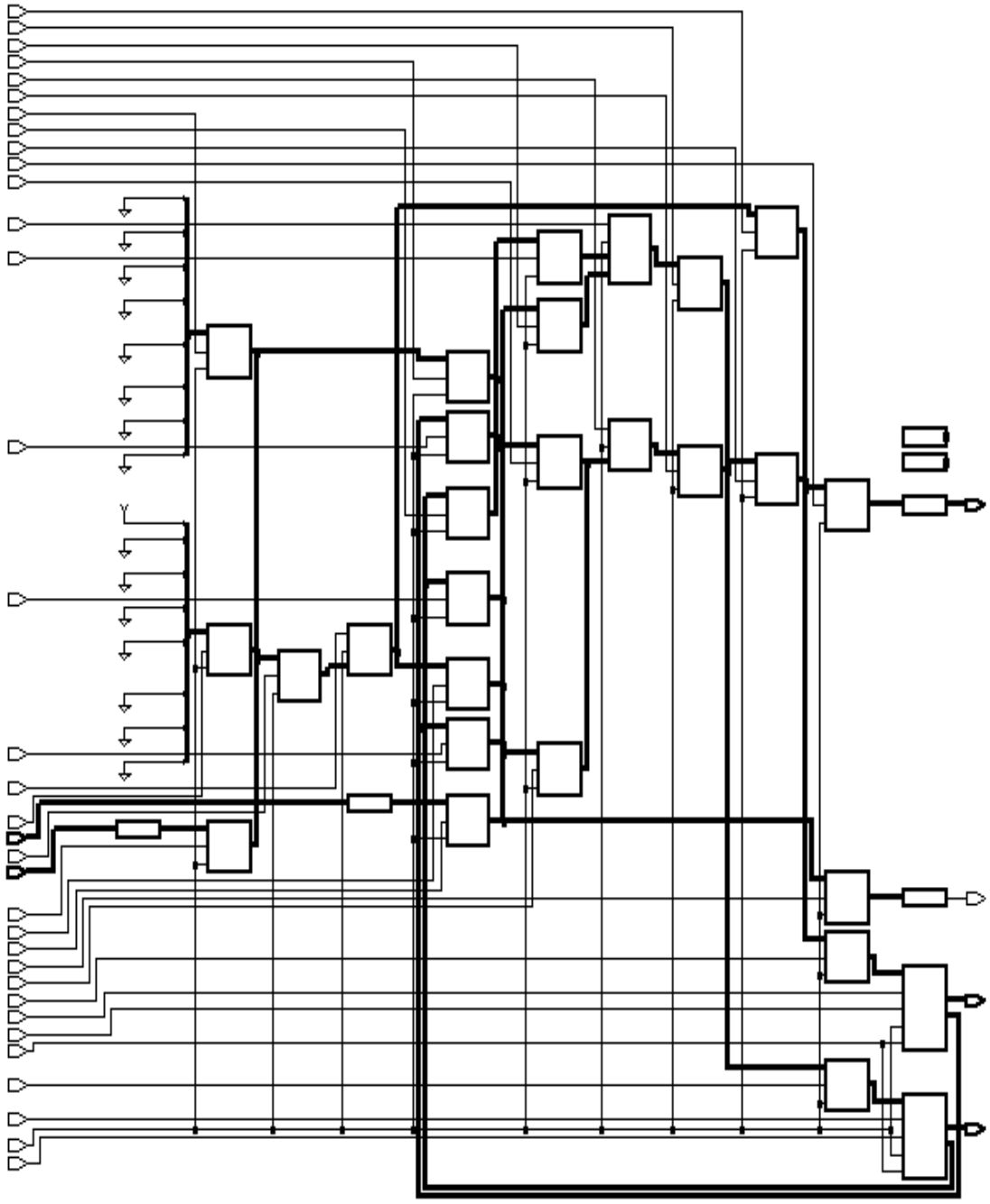


Figure C.24: *Layout du circuit du pgcd.*