**République Algérienne Démocratique et Populaire**

**Ministère de l'Enseignement Supérieur et de la Recherche Scientifique**

**Université de Batna**

**Faculté des Sciences de l'Ingénieur**

**Département d'Informatique**

## THESE

**Présentée par**

**AMOKRANE Samah**

**En vue de l'obtention du diplôme de Magister en Informatique**

**Option : Informatique Industrielle**

## THEME

# Algorithme Génétique pour le Problème d'Ordonnancement dans la Synthèse de Haut Niveau pour Contrôleurs Dédiés.

**Soutenue le …… devant le jury composé de :**

| | | | |
|---|---|---|---|
| **Dr Abdelmadjid ZIDANI** | **M.C** | **Université de Batna** | **Président** |
| **Dr Mohamed BENMOHAMED** | **M.C** | **Université de Constantine** | **Rapporteur** |
| **Dr Azzeddine BILAMI** | **M.C** | **Université de Batna** | **Examinateur** |
| **Dr Mohamed Salah KHIREDDINE** | **C.C** | **Université de Batna** | **Examinateur** |

## ***Résumé***

Le progrès offert par la technologie de fabrication des circuits micro-électroniques a ouvert la voie à la conception de systèmes digitaux d'une grande complexité. D'où la nécessité progressive de concevoir des circuits à de plus haut niveaux d'abstraction comme le niveau algorithmique. La synthèse de haut niveau consiste en la traduction d'une description comportementale (algorithmique) en une description équivalente au niveau transfert de registres. Le processus de synthèse de haut niveau est composé de plusieurs phases entre autres la phase d'ordonnancement. Cette phase, qui a une grande influence sur la qualité du circuit final, définit une date d'exécution pour chaque opération de la description algorithmique tout en respectant certaines contraintes. Une grande variété d'algorithmes existe dans la littérature pour résoudre efficacement le problème d'ordonnancement, reconnu comme étant un problème NP-Complet.

Cette thèse présente un aperçu sur les différentes techniques d'ordonnancement connues actuellement. Ces algorithmes ont été classifiés, selon leur domaine d'application, en deux catégories : des algorithmes orientés flot de données et ceux orientés flot de contrôle. Pour chaque catégorie un ensemble d'algorithmes sont décrits brièvement. Finalement, on propose une technique d'ordonnancement GPBS, pour les circuits de contrôle, basée sur la combinaison de l'algorithme d'ordonnancement à base de chemins et d'un algorithme génétique.



***Mots-clés :*** CAO/VLSI, Synthèse de haut niveau, Ordonnancement, Algorithme d'ordonnancement à base de chemin, Algorithmes génétiques.



## ***Abstract***

The progress offered by the technology of micro-electronics circuits manufacture opened the way to the conception of digital systems of a big complexity. From where the progressive necessity to conceive circuits to higher levels of abstraction as the algorithmic level. The High-Level Synthesis consists in the translation of a behavioral description (algorithmic) in an equivalent description at the Register-Transfer Level. The High-Level Synthesis process is composed of several phases among others the phase of scheduling. This phase, which has a tremendous impact on the final design quality, defines a date of execution for every operation of the algorithmic description while respecting some constraints. A big variety of algorithms exists in the literature to solve efficiently the problem of scheduling, recognized as being an NP-complete problem.

This thesis presents a preview on the different scheduling techniques known currently. These algorithms have been classified, according to their domain of application, in two categories,: data flow oriented algorithms and those control flow oriented. For every category a whole of algorithms is described briefly. Finally, we propose a scheduling technique GPBS, for circuits of control, based on the combination of the Path-Based Scheduling algorithm and a genetic algorithm.



***Key-words:*** CAD/VLSI, High-Level Synthesis, Scheduling, Path based scheduling algorithm, Genetic algorithms.

## *Remerciements*

Je tient avant tout à exprimer ma profonde reconnaissance à Mr Mohamed BENMOHAMMED Maître de Conférence à l'Université de Constantine, qui a assuré la direction de ma thèse, pour son suivi et ses conseils judicieux. Qu'il trouve ici l'expression de mon profond respect.

Je remercie Mr ZIDANI, Maître de Conférence à l'Université de Batna, de m'avoir fait l'honneur de présider le jury de ma thèse.

Je remercie, également, Mr BILAMI Maître de Conférence à l'Université de Batna et Mr KHIREDDINE Chargé de Cours à l'Université de Batna, qui ont accepté d'être membres du jury de cette thèse.

Je remercie Mr DEKHINET pour toute la documentation qu'il m'a fourni, ainsi que Mr ABDESSEMED Ridha pour son aide.

Un très grand et très spécial merci à mes parents, et à toute ma famille.

Que toutes les personnes et tous mes amis qui m'ont soutenu moralement soient assurés de l'expression de ma reconnaissance.

*A mon père et ma mère,*
*A ma sœur et mes frères.*

# Liste des figures

# Liste des tableaux

# Table des matières

# Chapitre 1 Introduction

## 1.1. Introduction

La technologie VLSI (**V**ery **L**arge **S**cale **I**ntegration) a connu une grande évolution, permettant l'intégration de systèmes complexes, composés de millions de transistors, sur une seule puce. Avec de tels systèmes, le processus de conception devient très difficile à gérer sans l'utilisation efficace d'outils CAO (**C**onception **A**ssistée par **O**rdinateur).

Les outils CAO aident à automatiser une grande partie du processus de conception, permettant ainsi au concepteur de ne pas se préoccuper des considérations de bas niveau. Ceci a permis une réduction considérable du temps de conception, améliorant ainsi le temps d'arrivée sur marché - Time to market - qui est un facteur crucial de réussite pas moins important que la surface ou la vitesse. Ceci a permis aussi aux concepteurs de manipuler des circuits plus complexes et d'étudier facilement les effets de variation des paramètres de conception sur le circuit final, en d'autres termes l'exploration complète de l'espace de conception.

En plus, avec l'accroissement de la complexité des circuits, apparaît le besoin de travailler à de plus haut niveaux d'abstraction, favorisant ainsi le développement de nouvelles méthodologies de conception comme la synthèse de haut niveau appelée aussi synthèse comportementale. Cette dernière traduit une description comportementale d'un circuit en une description structurelle au niveau transfert de registres, suivant certains critères. Le comportement d'un circuit est décrit à l'aide d'un langage de description de matériel HDL (VHDL, HARDWARE C,…). Alors que la sortie au niveau RTL (**R**egister **T**ransfert **L**evel) est composée d'un contrôleur et d'un chemin de données. Le contrôleur est défini comme une machine d'états finis FSM (**F**inites **S**tates **M**achine), alors que le chemin de données est décrit à l'aide d'instances d'unités fonctionnelles, d'unités de stockage et d'unités de communication interconnectées. La machine d'états finis spécifie quelles sont les opérations que doit exécuter le chemin de données à chaque étape de contrôle.

La nature de l'application traitée, qu'elle soit orientée flot de contrôle ou orientée flot de données, fait que l'effort d'optimisation soit consacré en grande partie à la génération de la partie contrôle ou la partie opérative.

Le processus de synthèse de haut niveau est exécuté en une séquence d'étapes, dont l'ordonnancement des opérations, l'allocation des ressources aux opérations et l'assignation des ressources aux opérations sont les étapes les plus marquantes.

L'objectif de ce mémoire est double. D'une part, présenter les différentes tâches de la synthèse de haut niveau et étudier, plus spécialement, les algorithmes d'ordonnancement dans leur globalité. D'autre part, choisir parmi les algorithmes étudiés ceux qui s'adaptent le mieux aux besoins des circuits de commande. La majorité des algorithmes d'ordonnancement existants traitent les applications dominées par le flot de données et concentrent principalement sur les calculs à l'intérieur d'un bloc de base. Alors que les circuits dominés par le contrôle ont une structure de flot de contrôle complexe contenant des boucles et des instructions conditionnelles. Il est donc nécessaire de prendre en compte les caractéristiques de ces applications et de développer un ensemble de techniques qui permettent de minimiser le coût de la partie contrôle. Pour cela, notre choix a porté sur l'algorithme d'ordonnancement à base de chemin PBS (**P**ath **B**ased **S**cheduling Algorithm) qui est à la base de la majorité des algorithmes d'ordonnancement des circuits de contrôle.

L'apport de ce mémoire concerne l'extension de l'algorithme PBS par l'introduction d'un algorithme génétique qui a pour rôle le réordonnement des opérations. Ce travail est proposé dans l'optique d'apporter plusieurs améliorations et optimisations par rapport à l'algorithme PBS pré-existant.

## 1.2. Structure du mémoire

Le déroulement de ce mémoire suivra les étapes suivantes:

Le chapitre suivant présente en premier les différents niveaux d'abstraction ainsi que le flux de conception d'un circuit intégré, suivi d'une brève description des langages de description de matériel et spécialement le langage VHDL.

Le chapitre 3 est une introduction à la synthèse de haut niveau constituée de plusieurs étapes, dont l'ordonnancement et l'allocation sont les plus importantes. Outre une brève présentation de ces différentes étapes et des différentes représentations intermédiaires, ce chapitre présente trois outils de synthèse de haut niveau, à savoir AMICAL, CATHEDRAL-2/3 et SYNOPSYS Behavioral Compiler.

Le chapitre 4 décrit le problème d'ordonnancement, ainsi qu'une classification des différents types d'ordonnancement. Cette classification se base sur les représentations intermédiaires de la description comportementale.

Le chapitre 5 dresse un état de l'art sur les différentes approches et algorithmes proposés pour résoudre le problème d'ordonnancement considéré comme un problème NP-Complet.

Le chapitre 6 donne une présentation globale des algorithmes génétiques ainsi que les différents opérateurs génétiques mis en œuvre. Il présente, également, une nouvelle technique d'ordonnancement des circuits dominés par le flot de contrôle GPBS, qui consiste à utiliser un algorithme génétique en conjonction avec l'algorithme d'ordonnancement à base de chemin PBS.

Le chapitre 7 conclut et présente les perspectives de notre travail.

# Chapitre 2
# Synthèse de circuits

## 2.1. Introduction

L'évolution de la technologie des circuits intégrés a permis de réaliser des systèmes de plus en plus complexes et même leur intégration sur une seule puce. Cette évolution demande des niveaux d'intégration de plus en plus élevés, motivée par les besoins de systèmes plus performants, légers, compacts et consommant un minimum de puissance. Dans de telles circonstances, le raisonnement sur un circuit au niveau transistor ou au niveau porte devient une tâche pénible et coûteuse quand on considère les contraintes de mise en marché d'un produit. De plus, en se dégageant des niveaux de spécification ou de description les plus élémentaires, la fonctionnalité est plus facilement compréhensible et peut donc être plus aisément corrigée ou modifiée. De ce fait, et afin de réduire le coût et le temps de conception des circuits, l'automatisation du processus de conception et l'abstraction du niveau de conception sont privilégiées [1].

La synthèse automatique est l'une des approches les plus prometteuses. Elle consiste en la traduction d'une description abstraite d'un circuit en une description plus concrète. Elle permet d'évaluer plusieurs possibilités d'implémentation. Elle contribue aussi à l'amélioration de la qualité des circuits [1,2].

Pour la conception d 'un système électronique, l'une des méthodes que peut utiliser un concepteur est la méthode " top down design". Un système est construit comme une hiérarchie d'objets où les détails de réalisation se précisent au fur et à mesure que l'on descend dans cette hiérarchie ; à un niveau donné de la hiérarchie, les détails de fonctionnement interne des niveaux inférieurs sont invisibles. Plusieurs réalisations d'une même fonction pourront être envisagées, sans qu'il soit nécessaire de remettre en cause la conception des niveaux supérieurs ; plusieurs personnes pourront collaborer à un même projet, sans que chacun ait à connaître tous les détails de l'ensemble.

## 2.2. Niveaux d'abstraction

En général, la conception d'un circuit est un processus itératif. Dans les premières étapes, le système est représenté par une description comportementale. Puis, les détails structurels (i.e. comment sont interconnectés les sous-modules pour former un module) sont ajoutés au fur et à mesure, jusqu'à l'obtention d'une description structurelle (i.e. composée des éléments de base tels que portes, bascules, etc et leurs interconnexions). Enfin, les éléments de la description structurelle sont implémentés par des éléments physiques (par exemple, dessin de masque représentant une bascule ) [1].

L'abstraction, d'un composant par exemple, est une description succincte qui supprime les détails d'implantation inutiles pour en comprendre la fonctionnalité. Donc, on identifie dans chaque abstraction deux niveaux : le niveau succinct le plus élevé, ou *spécification*, et le niveau détaillé le plus bas, ou *réalisation* [3]. On distingue trois domaines de représentation d'un circuit : [2]

- **Domaine comportemental :**
  On s'intéresse à ce que fait le circuit et non pas à sa mise en œuvre. On décrit le comportement de chaque sortie en fonction des entrées et du temps. Le fonctionnement du circuit peut être spécifié par des équations booléennes, des tables de valeurs des entrées et des sorties, ou des algorithmes écrits en langages de haut niveau standards ou des langages de description de matériel spéciaux (HDLs) tels que VHDL, Verilog.

- **Domaine structurel :**
  Il sert de lien entre le domaine comportemental et le domaine physique. Le circuit y est représenté comme une interconnexion de composants de base (portes, bascules, etc.).

- **Domaine physique :**
  On spécifie comment construire une structure qui a la connectivité nécessaire pour implémenter la fonctionnalité prescrite. Généralement le circuit se présente dans ce domaine sous la forme de dessin de masque. La fonctionnalité est ignorée, dans la mesure du possible, on décrit les objets réels (par exemple les transistors d'un circuit).

Le diagramme en Y de Gajski et Kuhn [1,2,4] illustre les différents types de synthèse et les différents niveaux d'abstraction. Les trois domaines de description : comportemental, structurel et physique sont représentés par trois axes. Les cercles concentriques présentent les différents niveaux d'abstraction et leurs représentations dans chacun des trois domaines. Plus on s'éloigne du centre plus le niveau devient plus abstrait. Une tâche de synthèse peut être vue comme une transformation d'un axe à un autre et/ou une transformation d'un niveau d'abstraction à un autre [4].

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 [5]. Différents niveaux d'abstraction ont été identifiés [1,4,6] :

- **Niveau système (System level) :**
Le niveau système est utilisé pour spécifier des systèmes entiers, comprenant des parties logicielles et des parties matérielles. Les descriptions comportementales du niveau système sont généralement données en terme de processus communicants. Dans le domaine structurel, les descriptions sont en terme d'interconnexions d'éléments tels que processeurs, mémoires,

unités périphériques, etc. L'étape de base est la transaction de communication pour contrôler l'enchaînement des processus.

**- Niveau algorithmique (Algorithm level) :**
La description à ce niveau s'attache à décrire le fonctionnement (comportement) d'un modèle sans se soucier de la structure. Les objets manipulés à ce niveau sont des variables et des opérateurs. L'étape de base est le pas de calcul. Un pas de calcul se compose d'un ensemble d'opérations exécutées entre deux points successifs d'entrées/sorties et/ou de synchronisation. Il peut durer plusieurs cycles d'horloge.

**- Niveau transfert de registres (Register Transfert Level) :**
Le comportement à ce niveau est décrit par des transferts et des opérations entre registres. La structure est représentée par l'interconnexion de registres, d'opérateurs arithmétiques et logiques, etc. (on spécifie une opération et non pas une implémentation spécifique de l'opération ). Le temps est divisé en intervalles appelés états de contrôle.

**- Niveau logique (Gate level) :**
A ce niveau, la description comportementale se fait avec des équations booléennes, qui sont représentées dans le domaine structurel par des blocs logiques (bascules, portes).

**- Niveau physique (Circuit level) :**
C 'est le niveau d'abstraction le plus bas. Le comportement à ce niveau est décrit par des fonctions de transferts. La structure est représentée par l'interconnexion de composants électroniques tels que transistors, diodes, etc.

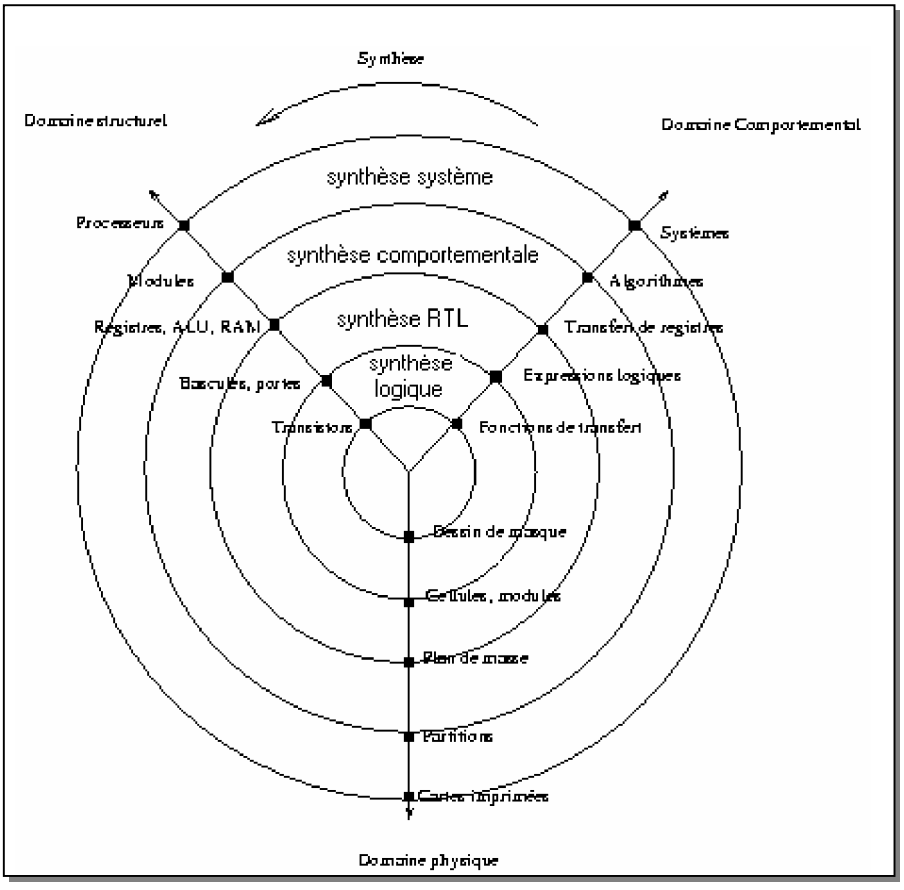



Figure 2.1: Domaines de représentation et niveaux d'abstraction( diagramme en Y de Gajski)

La décomposition en niveaux d'abstraction peut varier selon les auteurs. Certains auteurs [2,5,7] n'utilisent que quatre niveaux d'abstraction ( pour certains [2] les niveaux système et comportemental ne font qu'un seul niveau, pour d'autres [5,7] le niveau logique et physique sont réunis en un seul).

Ces niveaux d'abstraction correspondent aux étapes de la conception d'un circuit sur silicium. Le passage d'un niveau de représentation à un niveau inférieur est réalisé par ce qu'on appelle des "compilateurs de silicium".

## 2.3. Flot de conception d'un circuit

Les circuits actuels sont si complexes qu'il a bien fallu développer un ensemble de niveaux d'abstraction afin de diviser le flot de synthèse en sous-problèmes simplifiés.

Comme dit précédemment, la synthèse peut être vue comme un changement de domaine de représentation et/ou de niveau d'abstraction. Suivant le niveau d'abstraction de la description d'entrée et celui de la sortie, plusieurs types de synthèse peuvent être considérés, comme illustré par la figure 2.1 : synthèse au niveau système, synthèse comportementale, synthèse au niveau transfert de registres, synthèse logique, et synthèse physique.

### 2.3.1. Synthèse au niveau système

La première étape du flot de synthèse se situe au niveau système, c'est l'étape de partitionnement. Le partitionnement consiste, à partir des spécifications, à découper le système en sous-systèmes (processus) communicants. Un processus peut être partitionné une nouvelle fois en un ensemble de processus communicants. Le niveau système est utilisé pour spécifier des systèmes entiers, comprenant des parties logicielles et des parties matérielles. Le partitionnement peut être guidé par de nombreux paramètres comme la fonctionnalité, la synchronisation et la concurrence de tâches [7], le partitionnement matériel/logiciel [1], etc.

### 2.3.2. Synthèse de haut niveau

La synthèse de haut niveau (dite aussi synthèse architecturale ou comportementale) prend en entrée une spécification algorithmique de la fonction réalisée par le système à concevoir. Cette description sera traduite, par un outil de synthèse architecturale, en une description structurelle au niveau transfert de registres. La synthèse comportementale peut être décomposée en trois étapes majeures : l'ordonnancement, l'allocation, et l'assignation des ressources.

### 2.3.3. Synthèse au niveau transfert de registres

La synthèse RTL traduit une description de niveau transfert de registres, composée généralement d'une partie contrôle et une partie chemin de données, en une description au niveau porte. Elle transforme la spécification d'un circuit en un ensemble d'équations logiques en effectuant, elle aussi, une allocation de ressources [3]. Les opérations effectuées sur le contrôleur décrit par une machine d'états, pour une implémentation en terme d'architecture, sont l'affectation d'états (state encoding), la minimisation d'états, etc. Les opérations effectuées sur la partie chemin de données concernent le partage de ressources et le retiming [1].

### 2.3.4. Synthèse logique

Les outils de synthèse logique prennent en entrée une description sous forme de blocs logiques interconnectés. Le but est d'optimiser les blocs logiques en terme de surface du circuit généré [1,7]. Une fois les équations optimisées, celles-ci sont décomposées sur la bibliothèque de composants technologiques dont le concepteur dispose pour l'implémentation. Cette étape de recouvrement ne modifie pas la structure des équations, mais choisit un ensemble optimal de portes pour minimiser le ou les critères d'optimisation.

### 2.3.5. Synthèse physique

La synthèse physique consiste à générer les motifs géométriques des masques de fabrication des circuits. Cette étape de placement et de routage des descriptions des architectures résultantes sur le silicium termine le flot de conception.

## 2.4. Langages de description

La diversité des niveaux d'abstraction et des outils de conception (synthèse, simulation, etc.) ont conduit à la définition de langages spécialisés pour la description de matériel appelés HDLs (**H**ardware **D**escription **L**anguages). Ces langages ont des avantages importants tels que [1]: hiérarchie, possibilité de décrire la plupart des niveaux, possibilité de mélanger dans une même description des parties décrites dans des niveaux d'abstraction différents, indépendance vis-à-vis des outils CAO, etc.

Le point de départ de la synthèse d'architectures est la description comportementale du circuit à générer, exprimant sa fonctionnalité. On utilise les langages HDLs pour décrire la fonctionnalité du circuit, qui peut être comparée à un programme. La plupart des outils de synthèse comportementale sont sensibles au style d'écriture de cette description initiale, et l'efficacité de l'architecture résultante est fortement influencée par ce style d'écriture qui peut varier considérablement d'un outil à un autre [8]. L'utilisation de langages de programmation classiques tels que PASCAL et C a été envisagée, mais il leur manque la notion de temps et d'autres concepts liés au matériel tels que signaux, composants, etc. [1]

Le type du circuit à synthétiser détermine le type de langage utilisé. Pour les circuits dominés par le contrôle, on utilise des langages procéduraux tels que VHDL [9], Verilog [10], Hardware C [11]. Alors que pour les applications de type traitement du signal, on utilise des langages déclaratifs dont font partie Silage [12], Signal [13], Lustre, etc.

### 2.4.1. Langages déclaratifs

Les langages déclaratifs sont dédiés tout spécialement à des circuits de traitement du signal. Pour les applications de traitement du signal, le circuit est spécifié comme l'équation des sorties en fonction des entrées et du temps. Les langages déclaratifs représentent plus naturellement le parallélisme de la spécification, le langage Silage en est l'exemple type.

#### 2.4.1.1. Silage

Silage [12] a été développé par Hilfinger. L'objet de base est le signal qui est un vecteur ayant autant d'éléments que d'instants d'observation. L'opération fondamentale est

l'appel de fonction. Chaque appel est une expression des valeurs précédentes des variables et il exprime ce qui se passe entre chaque instant d'échantillonnage.

Une description Silage peut être rendue équivalente à un graphe de flots entre signaux. C'est pourquoi il est bien adapté aux applications de type DSP (**D**igital **S**ignal **P**rocessing), bien que ce langage comprenne également les instructions de contrôle. La figure 2.2 illustre la description d'un filtre récursif en langage Silage, l'opérateur @n signifie un retard de n unités de temps.

```
Function IIR (a1, a2, b1, b2, x : num)
    /* retourne */  y : = num =
  begin
 y = mid + a2 * mid @1 + b2 *mid@2 ;
 mid = x + a1** mid @1 + b1*mid@2 ;
  end
```

Figure 2.2 : Description Silage d'un filtre récursif

## 2.4.2. Langages procéduraux

Le comportement des circuits séquentiels peut être vu comme un programme. Ainsi il peut être décrit à l'aide de langages procéduraux. Un tel langage doit pouvoir décrire d'une part la fonctionnalité du circuit, mais également l'interface du circuit (notions portant sur la structure comme la liste des ports d'E/S, des signaux de commande, etc.). Ces langages permettent la description de tout type de circuit grâce au large éventail d'instructions offertes [8].

Les langages de description de matériel sont pour la plupart dérivés des langages de programmation classiques tels que ADA et C. Actuellement, deux langages sont largement utilisés VHDL et Verilog. Ce type de langages exige une étape de compilation qui traduit la description impérative en un graphe flot de données et de contrôle.

### 2.4.2.1. Verilog

Verilog [10] à tout d'abord été développé comme un langage propriétaire puis rendu public à la fin des années 80. Ce langage à été conçu pour la modélisation et la simulation des circuits du niveau comportemental au niveau logique. Verilog ne nécessite pas la distinction entre l'interface et le corps de la spécification. Bien que Verilog soit beaucoup plus concis que VHDL, il est aussi beaucoup plus limité.

Verilog comprend les structures de contrôle usuelles (boucles et branchement) ainsi que la notion de sous-programme. L'instruction **wait** sert à la synchronisation des processus concurrents. Deux types d'assignation existent en langage Verilog. L'assignation continue (avec le mot-clé *assign*) dénote l'assignation immédiate. L'assignation procédurale (sans mot-clé) est une assignation qui a lieu lorsque l'instruction est exécutée. La figure 2.3 donne la description comportementale Verilog d'un circuit de résolution de l'équation différentielle y''+3xy'+3y=0 sur un intervalle [0,a].

```
module DIFFEQ(x,y,u,dx,,a,clock,start) ;
        input [7 :0] a,dx ; inout [7 :0] x,y,u ;
        input [7 :0] clock, start; reg [7 :0] x1,u1,y1 ;
      always
```

```
begin
   wait(start) ;
   while (x<a)
          begin
             x1=x+dx ;
             u1= u- (3*x*u*dx)- (3*y*dx) ;
             y1 =y+ (u*dx) ;
             @(posedge clock) ;
             x := x1 ; u := u1 ; y := y& ;
          end
endmodule
```

Figure 2.3 : Exemple de description Verilog

### 2.4.2.2. VHDL

VHDL [9,44] est l'acronyme de " **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**angage ". Il a été développé dans le cadre du projet VHSIC commandité par le DoD (Département de la Défense Américain), et initié en 1980. C'est un standard IEEE (**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers) depuis 1987 (IEEE 1076-87). Sa syntaxe reprend celle d'ADA, et il a été défini, initialement, pour décrire des architectures en vue de les simuler ou de les documenter. Cependant sa large acceptation comme langage standard le rend très attractif pour tous les outils de conception de circuits et largement accepté par l'industrie [1,44].

La complexité des composants décrits en langage VHDL peut varier d'une simple porte logique à un système complexe. Un composant est décrit en VHDL par une entité de conception : *Design Entity*. L'entité est constituée de deux parties, une interface et une (ou plusieurs) architecture(s).

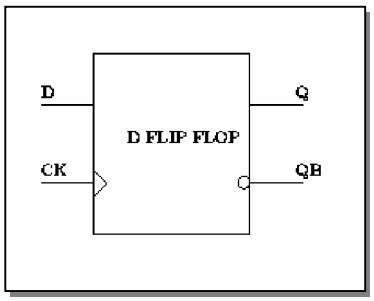



Figure 2.4.a : Bascule D

```
Architecture  STRUC of DFLIP_FLOP is
Component FLIP_FLOP
Port (A1,A2,Ck :in bit ; S, SB :out BIT) ;
end component ;

component NOT_GATE
port (A :in BIT ; B :out BIT)
end component ;
signal S1,S2,DB,CKB :BIT;
begin
     C1 :NOT_GATE port map (D, DB) ;
     C2 :NOT_GATE port map (CK, CKB) ;
     C3 : FLIP_FLOP port map (D,DB,CK,S1, S2) ;
     C4 :FLIP_FLOP port map (S1,S2,CKB,Q,QB) ;
 end STRUC ;
```

Figure 2.4.b : Description VHDL correspondante

L'interface, vision externe, est appelée *spécification d'entité* en VHDL. Elle spécifie les ports du circuit. La figure 2.4.b donne une description possible, donnée en VHDL, de l'interface d'une bascule D sensible au front montant d'une horloge (figure 2.4.a)[1].

L'architecture permet de décrire la fonction mise en œuvre, le contenu, de l'entité de façon structurelle, comportementale ou sous forme de flot de données. Le modèle structurel décrit une interconnexion de modules (composants). Le modèle comportemental contient des "process " qui contiennent des instructions séquentielles (algorithme). Le modèle flot de données est un style supplémentaire utilisé pour modéliser l'assignation parallèle de signaux. On peut associer plusieurs architectures à la même spécification d'entité (interface) [1].

Pour décrire le fonctionnement interne de la bascule (figure 2.4.a) plusieurs architectures sont possibles. Le tableau 2.1 donne trois architectures différentes [1] :

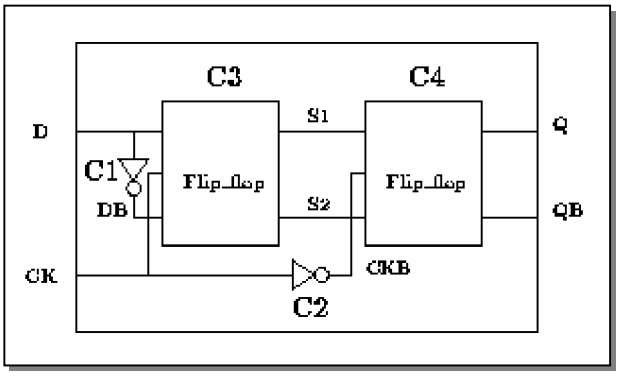



Figure 2.5 : Vision structurelle de la bascule [1]

- **Description structurelle** : La bascule est vue comme étant l'interconnexion de plusieurs sous-modules, c'est à dire des FLIP-FLOPs et des portes NOT. La première description donnée par le tableau2.1, est la description VHDL correspondant à la figure 2.5. Les deux composants FLIP-FLOP et NOT-GATE utilisés dans l'architecture sont déclarés à l'aide de l'instruction *component*. Le corps de l'architecture est un ensemble d'instantiations de composants. Au composant C1 qui correspond à une porte NOT, sont associés les signaux D et DB (D est l'entrée, DB la sortie). Les interconnexions sont faites via les paramètres effectifs (i.e. les signaux locaux et les signaux d'entrée/sortie).

**- Description flots de données** : La condition CK='1' and not CK'STABLE signifie que le signal CK est égal à '1' et qu'il vient juste de changer. L'instruction Q<=D *after* 5ns est une affectation de signaux concurrente, Q prendra la valeur courante de D après 5ns. Les deux affectations du bloc s'exécutent de manière concurrente.

- **Description comportementale** : Le fonctionnement de la bascule, contrairement à la description structurelle, est décrit ici sans se soucier de "comment elle est faite ". On s'intéresse à ce qu'elle fait ". Dans le programme VHDL, on trouve un processus «process » qui permet de faire des descriptions séquentielles. Les deux instructions d'affectation s'exécutent de manière séquentielle.

| Description structurelle | Description Flots de données | Description comportementale |
|---|---|---|
| **Architecture** STRUC **of** DFLIP-FLOP **is**<br>**component** FLIP-FLOP<br>**Port** (A1,A2,CK :**in** BIT; S, SB : **out** BIT) ;<br>**End component** ;<br>**Component** NOT-GATE<br>**Port** (A : **in** BIT; B : **out** BIT);<br>**End component**;<br>**Signal** S1, S2, DB, CKB: BIT;<br>**Begin**<br>C1 :NOT-GATE **port map** (D,DB);<br>C2 : NOT-GATE **port map** (CK,CKB);<br>C3 :FLIP-FLOP **port map** (D,DB,CK,S1,S2);<br>C4 : FLIP-FLOP **port map** (S1,S2,CKB,Q,QB);<br>**End** STRUC; | **Architecture** DATAFLOW **of** DFLIP-FLOP **is**<br>**Begin**<br>Q<=D **after** 5ns **when** CK='1' **and not** CK'stable ;<br>QB<=not D **after** 5ns **when** CK='1' **and not** CK'stable ;<br>**End** DATAFLOW ; | **Architecture** BEHAV **of** DFLIP-FLOP **is**<br>**Begin**<br>**Process** (CK)<br>**Begin**<br>If CK='1' **and not** CK'stable **then** Q<=D **after** 5ns ;<br>QB<=not D **after** 5ns ;<br>**End if** ;<br>**End process** ;<br>**End** BEHAV ; |

Tableau 2.1 : Différentes architectures associées à une même spécification d'entité

### 2.4.3. Quel langage utiliser ?

Les HDLs doivent satisfaire de multiples critères, c'est pourquoi il est illusoire de vouloir connaître le meilleur d'entre eux. La qualité d'un langage peut s'exprimer comme sa puissance d'expression, et/ou comme l'existence de simulateurs, d'outils de synthèse basés sur ce langage. Différents langages permettent d'exprimer le comportement du matériel. VHDL et Verilog permettent de décrire les circuits de type général, alors que Silage est plus recommandé pour les applications de type DSP.

La disponibilité d'outils de CAO permettant de simuler, vérifier ou synthétiser les circuits est un facteur clé pour le choix d'un langage particulier. Silage a été essentiellement utilisé dans le domaine particulier des DSP, pour lequel des outils de synthèse architecturale (e.g. CATHEDRAL) ont été développé et commercialisé avec succès. Quelques outils commerciaux de synthèse (e.g. Synopsys) et beaucoup de simulateurs ont été développés sur VHDL. Des restrictions à des sous-ensembles du langage ont été établies pour la synthèse, en particulier pour VHDL, en raison principalement de la complexité du langage complet. Les circuits ayant des fonctions comportant des structures de contrôle complexes peuvent être mieux décrits à l'aide de langages procéduraux. VHDL et Verilog sont largement utilisés.

## 2.5. VHDL

Le langage VHDL est conçu de façon à permettre de modéliser des systèmes complexes décrits à des niveaux d'abstraction très différents [6]. En d'autres termes, VHDL intègre adéquatement les méthodologies de conception de systèmes dites «Top/Down design ». De plus, VHDL est un langage modulaire et hiérarchique. Un système complexe peut être divisé en plusieurs blocs, chaque bloc peut à son tour être divisé en plusieurs sous-

blocs et ainsi de suite, et l'interface entre ces blocs se fait par des «liens de communication » ; de même pour un modèle VHDL.

VHDL a été, initialement, conçu comme un langage de simulation et modélisation des circuits complexes[8] et il a graduellement été utilisé comme un langage de synthèse. C'est ce qui fait que certaines structures ne sont pas adaptables pour la synthèse comportementale. Ainsi, chaque outil restreint-il son modèle d'entrée à un sous-ensemble du langage VHDL. Pour illustrer l'aspect de simulation du langage VHDL, on peut citer quelques exemples :

- Contrairement à C ou PASCAL, VHDL est un langage qui comprend le «parallélisme », c'est à dire que des blocs d'instructions peuvent être exécutés simultanément. Ce parallélisme est fondamental pour comprendre le fonctionnement d'un simulateur logique.
- La modélisation correcte d'un système suppose de prendre en compte, au niveau du simulateur, les imperfections du monde réel. VHDL offre donc la possibilité de spécifier des retards, de préciser ce qui se passe lors d'un conflit de bus, etc.
- …….

## 2.5.1. Description d'un module en VHDL

Un circuit décrit en VHDL est composé d'un ensemble de modules interconnectés[8]. Chaque module est appelé, en terminologie VHDL, une *entité* «design entity ». L'entité est constituée de deux composants, la description de l'*interface* et le corps de l'*architecture*. L'énoncé *entity* (entity declaration) définit l'interface du module vers l'environnement (i.e. les ports du circuit). L'énoncé *architecture* décrit quant à lui le comportement et le contenu interne du module. Le corps peut être décrit de trois façons, discutées dans la section 2.4.2.2, structurelle, comportementale ou sous forme de flots de données.

Le modèle structurel décrit une interconnexion de modules. Le modèle comportemental contient des "*process*" qui contiennent des instructions séquentielles. Le modèle flot de données est un style supplémentaire utilisé pour modéliser l'assignation parallèle de signaux.

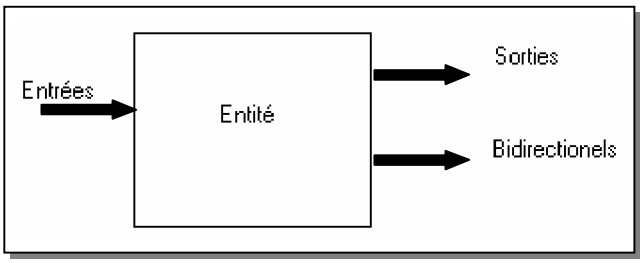
Le langage VHDL contient les notions de constantes, variables et signaux. Les variables ne sont pas directement liées à des notions de matériel ; elles sont utilisées internement au modèle comportemental, à l'instar des variables informatiques. Les signaux sont une généralisation des variables, ils modélisent les signaux électriques qui peuvent traverser les ports et auxquels est attachée la notion de retard ou de délai. Les structures de contrôle comme les boucles font partie du langage ainsi que les notions de fonctions et procédures. L'instruction *wait* sert à synchroniser l'exécution des événements, lorsque l'on modélise le comportement d'un circuit à l'aide de processus concurrents.

Les types de données, les constantes et les sous-programmes peuvent être définis à l'intérieur de l'architecture. Ils ne sont pas visibles à l'extérieur. VHDL, et afin de pallier ce problème, fournit le mécanisme de *package* pour encapsuler les déclarations et les sous-programmes qui peuvent être référencés à l'extérieur.

### 2.5.1.1. L'énoncé ENTITY

L'énoncé ENTITY (Entity declaration) décrit l'interface du circuit, il définit le nom de l'entité, les noms des ports d'entrées/sorties, la direction des ports (entrée, sortie,…) ainsi que leurs types logiques (un port de 1bit, un bus de 8 lignes, etc.). Ainsi chaque module peut-il être considéré comme une boite noire dont les ports sont spécifiés mais sans plus d'informations à propos des transformations subies par les entrées pour la génération des

sorties[8]. La figure 2.6 montre une vue générale de la syntaxe d'une déclaration d'entité. La figure 2.7 présente le code VHDL d'un énoncé entity.




**Entity** <nom de l'entité> is
**Port** (

<nomduport1>:<Direction><Typelogique>;
….;
);
**end** < nom de l'entité>;

Figure 2.6: Vue générale de la syntaxe d'une déclaration d'entité

```
Library ieee ;
use ieee.std_logic_1164.all ;

-- Description de l'entité incrementeur
entity incrementeur is
port (A : in std_logic_vector (11 downto 0) ;
D : out std_logic_vector (11 downto 0) ;
UP : in std_logic) ;
End incrementeur ;
```

Figure 2.7: Exemple d'entité avec déclaration de bibliothèque

En VHDL, les bibliothèques sont spécifiées par le mot clé *library*, et pour avoir accés à une partie de la bibliothèque on utilise l'énoncé *use*.

```
Library ieee ;
Use ieee.std_logic_1164.all ;
```

Dans l'entité incrémenteur, ci-dessus, on a accès à tous les types logiques des ports définis dans le package std_logic_1164 de la librairie **IEEE**. VHDL permet aussi d'introduire des commentaires, ils sont introduits en plaçant deux tirets consécutifs "-- " et se terminent à la fin de la ligne.

```
Entity incrementeur is
```

Les mots *entity* et *is* sont des mots clés du langage VHDL, ils spécifient le nom de l'entité créée et indiquent le début de la description d'un nouveau design en VHDL.

```
Port (
    A    : in std_logic_vector (11 downto 0) ;
    D    : out std_logic_vector (11 downto 0) ;
    UP   : in  std_logic) ;
```

On déclare ici chacun des ports de l'entité (nom donné au port, sa direction " in, out, I/O " ainsi que le type de ce port).

**End** incrementeur ;

Le mot clé *end* indique la fin de la déclaration de l'entité.

#### 2.5.1.2. L'énoncé ARCHITECTURE

L'architecture décrit le fonctionnement interne d'un circuit. L'énoncé architecture contient un identificateur *architecture* ainsi que le nom de l'énoncé *entity* qui définit ses entrées/sorties, une partie déclarative optionnelle et une liste d'actions concurrentes (corps du programme ) dont seuls les processus sont traités par la synthèse comportementale. Le rôle des autres actions concurrentes est plutôt structurel [8]. La figure 2. 8 donne une vue générale de la syntaxe VHDL d'une architecture.

**Architecture** < nom de l'architecture > **of** < nom de l'entité > **is**
Partie déclarative optionnelle : types, constantes,
signaux locaux, composants.
**Begin**
< instructions > ….
**end** < nom de l'architecture >

Figure 2.8 : Vue générale de la syntaxe d'une déclaration d'architecture

L'architecture d'un circuit peut être décrite de différentes façons : description structurelle, description comportementale et description par flot de données. Cette possibilité de donner plusieurs architectures pour la même entité, autrement dit plusieurs réalisations, permet d'attacher plusieurs modèles du même module à la même entité.

### 2.5.2. L'énoncé PROCESS

Un processus VHDL, se divise en deux parties. Une partie déclarative qui contient toutes les déclarations locales au processus qui sont nécessaires à la description du comportement (déclaration de types, constantes, sous-programmes et variables). La deuxième partie d'un processus VHDL est constituée d'un corps associé à la partie déclarative et qui regroupe des instructions séquentielles. La tâche de la synthèse comportementale est de traduire ces instructions en une architecture cible. Bien que ces instructions soient similaires à celles utilisées par les langages de programmation procéduraux, elles ont des extensions spéciales qui permettent de spécifier les caractéristiques d'un circuit, telle que la possibilité de la synchronisation de la description à certains évènements extérieurs par l'une des deux méthodes suivantes :

- Une liste de sensibilité peut être attachée à chaque processus. Cette liste définit un ensemble de signaux qui déclenchent, par le changement de l'un quelconque d'entre eux, l'activité du processus. Le système est dit " sensitif " à ces signaux. Cette liste peut être remplacée par une instruction *wait* dans le corps du processus.
- Un processus n'ayant pas de listes de sensibilité peut contenir un nombre illimité d'instructions *wait*. Cette instruction d'attente assure la synchronisation entre les processus concurrents, ainsi qu'avec l'environnement, par l'intermédiaire des signaux et des ports. Elle indique au processus que son déroulement doit être

suspendu dans l'attente d'un évènement sur un signal, et tant qu'une condition n'est pas réalisée.

**wait** [**on** liste_de_signaux][**until** condition] ;

La liste des signaux dont l'instruction attend le changement de valeur joue exactement le même rôle que la liste de sensibilité du processus, mais l'instruction *wait* ne peut pas être utilisée en même temps qu'une liste de sensibilité.

**Processus : syntaxe générale**
[Etiquette :] **process** [(liste de sensibilité)]
partie déclarative optionnelle. Variables notamment.
**begin**
corps du processus
instructions séquentielles
**end** process [étiquette] ;

Les éléments mis entre crochets sont optionnels.

## 2.5.3. Les signaux et les variables

Le langage VHDL offre deux possibilités pour le stockage et la transmission des valeurs et des données : les variables et les signaux [8].

Les signaux sont principalement destinés à établir des connexions entre des modules concurrents, et sont déclarés dans la partie déclarative de l'architecture d'une description. Lors de la déclaration d'un signal, on doit spécifier son type et il est possible de lui attribuer une valeur par défaut.

*Déclaration :*
**Signal** <nom_du_signal> : type ;

*Assignation* (se place dans le corps d'une architecture ou d'un processus) :
<nom_du_signal> <= expression ;

*exemple :*
**Signal** etat_courant, etat_suivant : bit ;
Etat_suiv <= '0' ;

En VHDL, les affectations de signaux sont effectives avec un certain retard. Chaque signal de la description ne prend sa nouvelle valeur qu'à la fin du cycle de simulation, c'est à dire lorsque tous les processus sont bloqués par des instructions d'attente et avant que le prochain cycle de simulation ne commence. Par conséquent, si un signal est affecté deux ou plusieurs fois dans le même cycle de simulation, seule la dernière affectation sera prise en compte.

Les variables sont similaires aux variables des langages de programmation, elles servent à stocker des résultats intermédiaires pour faciliter la construction d'un algorithme séquentiel. On peut déclarer des variables dans un *process*, une *procedure* ou une *function*. Il est aussi possible de leur donner une valeur initiale qui sera attribuée à chaque appel de la procédure ou de la fonction.

*Déclaration :*

**variable** <nom_de_variable> : type [: = expression] ;

*Assignation :*

<var_destination> : = <var_source> ;

Par rapport aux signaux, les variables prennent leurs nouvelles valeurs au moment de l'affectation et elles les sauvegardent jusqu'à la prochaine affectation. Il est possible d'affecter la valeur d'une variable à un signal, et inversement, pourvu que les types soient compatibles. La différence entre variables et signaux est que les premières n'ont pas d'équivalent physique dans le schéma, contrairement aux seconds.

## 2.5.4. Les types de données

VHDL, héritier d'ADA, est un langage fortement typé. Toutes les données (constante, signal, variable ou fonction) ont un type qui doit être déclaré avant l'utilisation. Comme tous les langages de programmation modernes, les types prédéfinis sont les types élémentaires (types de base) tels que les entiers, les flottants, le type booléen, etc. Ces types permettent aux concepteurs de construire :

- Des sous-types (sous-ensembles du type de base), obtenus en précisant un domaine de variation limité de l'objet considéré.
- Des types composés, obtenus par la réunion de plusieurs types de base identiques (tableaux) ou de types différents (enregistrements).

En plus des types prédéfinis et de leurs dérivés, l'utilisateur a la possibilité de créer ses propres types sous forme de types énumérés. Voici les principaux types supportés par le langage VHDL :

**Type Integer**

Parmi les types VHDL on trouve les types scalaires. Ces types ont des valeurs uniques et comprennent les entiers, les flottants et les types énumérés. VHDL manipule des valeurs entières dont la longueur dépend de l'outil utilisé. Généralement, les valeurs entières correspondent à des mots de 32 bits, soit comprises entre -2147483648 et +2147483647.

*Déclaration :*

**signal** | **variable** | **constant** nom : **integer** ;

Le symbole "|" signifie "ou".

On peut spécifier une plage de valeurs inférieure à celle obtenue par défaut, par exemple :

**signal** etat : **integer range** 0 **to** 1023 ;

Permet de créer un compteur 10bits. La même construction permet de créer un sous_type :

**subtype** etat_10 : **integer range** 0 **to** 1023 ;
**signal** etat1, etat2 : etat_10 ;

**Type énuméré**

En VHDL, l'utilisateur peut définir ses propres types de données, par simple énumération de constantes symboliques qui fixent toutes les valeurs possibles du type.

*Exemple :*

```
Type    state_type  is (start, idle, waiting, run) ;
Signal   state : state_type ;
```

**Type bit**

C'est le type de base le plus utilisé en électronique numérique. Un objet de type *bit* peut prendre deux valeurs : " 0" et "1".

*Déclaration :*

**Signal** | **variable**  nom : **bit** ;

**Type Boolean**

Le type *boolean* peut prendre deux valeurs " true " et "false". Il intervient essentiellement comme résultat d'expressions de comparaisons, IF par exemple, ou dans les valeurs renvoyées par des fonctions.

*Déclaration :*

**Signal** | **variable** | **constant**  nom : **Boolean** ;

**Type Array (tableau)**

Les tableaux sont des structures de données où tous les éléments sont du même type. Les indices des tableaux doivent être d'un type scalaire autre que flottant. A part les vecteurs de bits, les tableaux sont transformés en mémoires multi-ports ou, pour les tableaux ayant une taille relativement réduite en bloc de registres [8]. Contrairement à d'autres langages de programmation tels que C ou PASCAL, en VHDL il faut toujours passer par une définition de type. Le type *bit_vector* est l'un des plus utilisés, et est défini dans la librairie standard par :

```
SUBTYPE  Natural   IS  Integer  RANGE 0 to integer'high ;
TYPE          bit_vector IS ARRAY (Natural RANGE <>) OF BIT ;
```

Dans l'exemple ci-dessus, le nombre d'éléments n'est pas précisé dans le type, ce sera fait à l'utilisation. Par exemple :

```
Signal       etat : bit_vector (0 to 4) ;
```

Le nombre de dimensions d'un tableau n'est pas limité, les indices peuvent être définis dans le sens croissant ou décroissant.

```
type BYTE is array (7 downto 0) of BIT ; --tableau de 8 bit
type VECTOR is array (3 downto 0) of BYTE ; -- tableau à deux dimensions
variable tmp : BYTE ;
```

Une fois défini, un objet composé peut être manipulé collectivement par un nom. Comme il est aussi possible de manipuler seulement une partie des éléments d'un tableau.

```
Signal etat1 : bit_vector (0 to 4) ;
Variable etat2 : bit_vector (0 to 4) ;
……..
etat1<= etat2 ;
etat2 (0 to 2) :=  "001" ;
```

**Type Record (enregistrement)**

Les enregistrements définissent des collections d 'objets (champs) de types différents. Leur usage est répandu pour décrire des données structurées telles que les entêtes, les trames de communication, etc.

*Déclaration :*

**type** <identificateur> **is record**
Définition du record
**end record** ;

*Exemple :*

```
type clock_time   is record              -- définition d'un type
hour : integer range 0 to 12 ;
minute, seconde : integer range 0 to 59 ;
end record ;
…….
variable time_of_day : clock_time ;   --déclaration d'un objet de ce type
…….
time_of_day.hour : =3 ;                  -- utilisation de l'objet précédent
time_of_day.minute :=45 ;
time_of_day .seconde :=55 ;
```

## 2.5.5. Instructions concurrentes

Les instructions concurrentes interviennent à l'intérieur d'une architecture, dans la description du fonctionnement d'un circuit. En raison du parallélisme du langage, ces instructions peuvent être écrites dans un ordre quelconque. Les principales instructions concurrentes sont :

- Les affectations concurrentes de signaux
- Les processus
- Les instanciations de composants
- Les instructions "generate "
- Les définitions de blocs

**Affectations concurrentes de signaux**

L'instruction d'affectation a trois formes possibles : affectation simple, conditionnelle ou sélective. L'affectation simple traduit une simple interconnexion entre deux équipotentielles :

```
Nom_de_signal <= expression ;
```

L'affectation conditionnelle permet de déterminer la valeur de la cible en fonction des résultats de tests logiques :

```
Cible <= source_1 when  condition_booléenne_1 else
         source_2  when condition_booléenne_2 else
         …….
         source_n ;
```

L'affectation sélective permet de choisir la valeur à affecter à un signal en fonction des valeurs possibles d'une expression.

```
with expression select
cible <= source_1 when valeur_11| valeur_12  …..,
         source_2 when valeur_21| valeur_22  .....,
          …….
         source_n when others ;
```

**Generate**

Les instructions "generate" permettent de créer de façon compacte des structures régulières, comme les registres ou les multiplexeurs. Une instruction generate permet de dupliquer un bloc d'instructions concurrentes un certain nombre de fois, ou de créer un tel bloc si une condition est vérifiée.

*Syntaxe :*

```
--structure répétitive :
étiquette : for variable in debut to fin generate
              instructions concurrentes
end generate [étiquette] ;
```

ou :

```
--structure conditionnelle :
étiquette : if condition generate
instructions concurrentes
end generate [étiquette] ;
```

**Block**

Une architecture peut être subdivisée en blocs, de façon à constituer une hiérarchie interne dans la description d'un composant complexe.

*Syntaxe :*

```
étiquette : block [(expression_de_garde)]
--zone de déclaration de signaux, composants, etc…
begin
--instructions concurrentes
end block [étiquette];
```

Dans les applications de synthèse, l'intérêt principal des blocs est de permettre de contrôler la portée et la visibilité des noms des objets utilisés : un nom déclaré dans un bloc est local à celui-ci.

## 2.5.6. Instructions conditionnelles

Le langage VHDL supporte des instructions séquentielles conditionnelles pour la représentation du comportement conditionnel telles que les instructions *if* et *case*. Ces instructions permettent de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par une ou des expressions.

**L'instruction " if … then … else … endif "**

*Syntaxe :*

```
If expression_logique then
instructions séquentielles
[elsif expression_logique then  ]
```

instructions séquentielles
[**else**]
instructions séquentielles
**end if** ;

Son interprétation est la même que dans les langages de programmation classiques comme C ou Pascal.

**L'instruction " case … when … endcase "**

L'instruction *case* permet de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par une expression.

*Syntaxe :*

**case** expression **is**
**when** choix[ | choix …. [|choix ]] => instruction séquentielle ;
**when** choix[ | choix …. [|choix ]] => instruction séquentielle ;
……
[**when others** => instruction séquentielle ;]
**end case** ;

Cette instruction est proche de " switch " de C ou de " case of " de Pascal.

## 2.5.7. Instructions itératives

Le langage VHDL supporte les actions répétitives, telles que les boucles. Elles permettent de répéter une séquence d'instructions. Les types de boucles disponibles sont les boucles inconditionnelles (*loop*) et les boucles dont le nombre d'exécution est lié à une condition exprimée dans la tête de boucle (*while* et *for*). Les instructions *exit* et *next* permettent de sortir d'une boucle et de forcer l'exécution à la prochaine itération. A cause de leur nature répétitive, les boucles influencent d'avantage le temps d'exécution de la description comportementale que les autres instructions. De ce fait, elles nécessitent l'application d'algorithmes sophistiqués pour la synthèse comportementale.

**Boucles infinies**

[étiquette :] **loop**
séquence d'instructions
**end loop** [étiquette] ;

Pour ce type de boucle, le nombre d'itérations est infini, ne dépandant pas d'une condition.

**Boucles "For"**

[étiquette :] **for** paramètre **in** minimum **to** maximum **loop**
séquence d'instructions
**end loop** [étiquette] ;

Ou :

[étiquette :] **for** paramètre **in** maximum **downto** minimum **loop**
séquence d'instructions
**end loop** [étiquette] ;

La valeur des deux limites est connue au moment de la compilation ou de l'élaboration de la description.

**Boucle "While"**

[étiquette:] **while** condition **loop**
séquence d'istructions
**end loop** [étiquette];

La condition peut dépendre de signaux externes ou de calculs effectués dans le corps de la boucle.

## 2.5.8. Programmation modulaire

Le langage VHDL offre la possibilité de subdiviser un gros programme en petits modules qui sont mis au point indépendamment les uns des autres et rassemblés ensuite. Chaque module peut être utilisé dans plusieurs applications différentes. Les outils de base de cette construction modulaire sont les sous-programmes, les paquetages et librairies, et les paramètres génériques.

### 2.5.8.1. Procédures et fonctions

Les sous-programmes du VHDL sont utilisés pour simplifier le codage des parties de code répétitives ou communes. Les deux catégories de sous-programmes, procédures et fonctions, diffèrent par les mécanismes d'échange d'informations entre le programme appelant et le sous-programme : les fonctions ont toujours une valeur de retour est leurs appels font toujours partie d'une expression. Les sous-programmes en VHDL, peuvent être définis dans les paquetages, architectures ou processus.

**Les fonctions**

Une fonction retourne au programme appelant une valeur unique, elle a donc un type. Elle ne peut en aucun cas modifier les valeurs de ses arguments d'appel et peut exclusivement utiliser des arguments de types constantes ou signaux, dont les valeurs lui sont transmises lors de l'appel.

*Déclaration :*

**Function** <nom_fonction> [(liste_paramètres_formels) ]
**return** nom_de_type ;

*Corps de la fonction*

**Function** <nom_fonction> [(liste_paramètres_formels)]
**Return** nom_de_type **is**
**[**déclarations]
**begin**
instructions séquentielles
**return** (expression) ;
**end** [nom_fonction] ;

*utilisation*

nom_fonction (liste_paramètres_réels)

Le corps d'une fonction ne peut pas contenir d'instruction *wait*, et lors de son utilisation, le nom d'une fonction peut apparaître partout dans une expression.

**Les procédures**

Une procédure, contrairement à une fonction, peut changer les arguments qu'elle reçoit du programme appelant. Elle accepte comme arguments : des constantes, variables ou signaux, en modes "in", "out" ou "InOut" (sauf les constantes qui sont toujours en mode "in").

*Déclaration :*

**Procedure** <nom_procédure> [(liste_paramètres_formels) ];

*Corps de la procedure:*

**Procedure** <nom_procédure> [(liste_paramètres_formels)] **is**
[déclarations]
**begin**
instructions séquentielles
**end** [<nom_procedure>] ;

*Utilisation :*

nom_procédure (liste_paramètres_réels) ;

Une procédure peut être appelée par une instruction concurrente ou par une instruction séquentielle.

### 2.5.8.2. Les paquetages et les librairies

Une librairie est une collection de modules VHDL qui ont déjà été compilés. Ces modules peuvent être des paquetages, des entités ou des architectures.

Une librairie par défaut, work, est systématiquement associée à l'environnement de travail de l'utilisateur. Pour utiliser des composants et paquetages d'une certaine librairie, la librairie et les paquetages doivent d'abord être spécifiés dans le code VHDL par les commandes suivantes :

**Library** <nom_librairie> ;
**Use** <nom_librairie>. <nom_paquetage>.**ALL** ;

Le VHDL standard est défini de telle façon que les librairies *work* et *std* sont toujours visibles. Par conséquent, on n'a pas besoin de les spécifier dans le code VHDL. Le paquetage nommé *standard* contient tous les types de données et fonctions prédéfinis dans le VHDL standard.

Dans les grandes conceptions (designs), il y a souvent des déclarations et des sous-programmes fréquemment utilisés dans différents composants et par différents concepteurs. Cette situation est rendue possible par l'utilisation des paquetages. Un paquetage permet de rassembler, par exemple, des fonctions, des procédures, des types, constantes et attributs, dans un module qui peut être compilée à part, et rendu visible au moyen de la clause ***use***. Un paquetage est constitué de deux parties :

- **La déclaration :** contient les informations publiques dont une application a besoin pour utiliser correctement les objets décrits par le paquetage.

Essentiellement des déclarations de procédures ou fonctions, des définitions de types, des définitions de constantes, etc.

- **Le corps :** contient le code des fonctions ou procédures déclarées dans la partie déclaration du paquetage.

*Syntaxe :*

```
Package  <nom_package> is
[déclaration_de_sous_programmes]
[déclaration_de_constantes]
[déclaration_de_types]
……
end  [nom_package];

package body <nom_package>  is
     [corps des sous-programes déclarés]
end [<nom_package>] ;
```

*exemple :*

```
package mypack is
  function   minimum (a, b :in  std_logic_vector)
              return    std_logic_vector ;
-- déclaration de constantes et de types
  constant maxint : integer := 16#FFFF# ;
  type    arithmetic_mode_type    is (signed, unsigned) ;
end mypack ;

package body mypack is
 function minimum (a, b :  std_logic_vector)
           return std_logic_vector is
    begin
        if a<b then
           return a ;
        else return b ;
        endif ;
     end minimum ;
end mypack ;
```

L'utilisation d'un paquetage se fait au moyen de la clause *use*.

*Exemple :*

```
Library ieee;
Use ieee.std_logic_1164.ALL ;
```

## 2.5.9. Les paramètres génériques

Une spécifité intéressante de l'entité VHDL est la possibilité de configurer la description à l'aide de paramètres génériques.

Lorsque l'on crée le couple entité-architecture d'un opérateur, que l'on souhaite utiliser comme composant dans une construction plus large, il est parfois pratique de laisser certains paramètres modifiables par le programme qui utilise le composant. De tels

paramètres, dont la valeur réelle peut n'être fixée que lors de l'instanciation du composant, sont appelés paramètres génériques.
Un paramètre générique se déclare au début de l'entité, et peut avoir une valeur par défaut :

**generic** (nom : type[: = valeur_par_défaut]) ;

La même déclaration doit apparaître dans la déclaration de composant, mais au moment de l'instanciation la taille peut être modifiée par une instruction "*generic map*" :

Etiquette : nom **generic map** (valeurs)
**port map** (liste_d'association) ;

## 2.6. Conclusion

L'évolution de la technologie de la micro-électronique a permis de créer des puces avec des millions de transistors. La nécessité de méthodologies de conception rapides et efficaces a été la principale raison qui a conduit au développement des outils CAO. Ces dernières années, le temps d'arrivée sur marché (time-to-market) est d'autant important que la surface ou la vitesse du design.

Dans ce chapitre, nous avons présenté le flot complet de la synthèse de circuits intégrés suivant les principaux niveaux d'abstraction. Les exigences de productivité obligent le concepteur à choisir un niveau de description plus élevé. Nous avons aussi présenté les langages de description de matériel (HDLs), les plus couramment utilisés. Le type de langage utilisé reflète le type de circuit visé. Pour les circuits dits "dominés par le contrôle", on utilise des langages procéduraux. A l'inverse pour les applications de type traitement du signal, on utilise des langages déclaratifs. On a vu aussi en bref l'un des langages HDLs les plus utilisés, le langage procédural VHDL.

Le chapitre suivant présente les principales tâches constituant la synthèse de haut niveau, l'une des méthodes utilisées pour accélérer le processus de conception.

# Chapitre 3

# Synthèse de haut niveau

## 3.1. Introduction

La complexité des circuits actuels a augmenté de manière exponentielle [5], d'où la nécessité d'automatiser le processus de conception de circuits intégrés ainsi que l'utilisation de plus haut niveaux d'abstraction. Aujourd'hui des outils de synthèse commençant avec une description comportementale permettent l'automatisation et l'accélération du processus de conception[17,18,35,45]. La synthèse de haut niveau a de grands avantages [4,5,8,15,16] :

- Le temps de conception global va se trouver fortement réduit ;
- Estimation de performance à un niveau d'abstraction plus élevé, et exploration architecturale plus approfondie avant de choisir la solution présentant le meilleur compromis (surface, vitesse, consommation, etc.) ;
- Réutilisation plus facile de composants existants ;
- Ouverture du monde de la conception des circuits aux concepteurs de systèmes. Les outils de synthèse de haut niveau relient les domaines de conception de systèmes et de circuits ;
- Les outils de conception automatique peuvent surpasser les concepteurs humains en satisfaisant les contraintes de conception ainsi que les exigences ;
- Etc.

Depuis plus de vingt ans, la recherche en synthèse comportementale est intensive. Les outils ALERT [17] et MIMOLA[18] sont les précurseurs dans le domaine des outils de synthèse architecturale. Aujourd'hui, plusieurs outils de synthèse comportementale universitaires et commerciaux sont disponibles, mais certaines limitations ont fait que peu de ces outils soient utilisés dans le milieu industriel. En général, le modèle d'entrée est trop

restreint, la difficulté d'intégration de ces outils dans des environnements de conception existants[8,15]. Il est donc nécessaire d'introduire une nouvelle génération d'outils de synthèse comportementale pour surpasser les problèmes posés par les outils classiques.

## 3.2. Principe général de la synthèse de haut niveau

La synthèse de haut niveau, autrement dit la synthèse comportementale, correspond à un ensemble de tâches de raffinements ou de transformations qui, à partir d'une spécification comportementale, génèrent une architecture cible décrite souvent au niveau transfert de registres, tout en satisfaisant un certain nombre de contraintes [19,20,21,22].

McFarland et Al étaient les premiers a avoir donné une définition à la synthèse de haut niveau : «la tâche de la synthèse de haut niveau est de prendre une spécification du comportement voulu d'un système et un ensemble de contraintes et d 'objectifs à satisfaire, et de trouver une structure qui implémente ce comportement tout en satisfaisant les objectifs et les contraintes ».

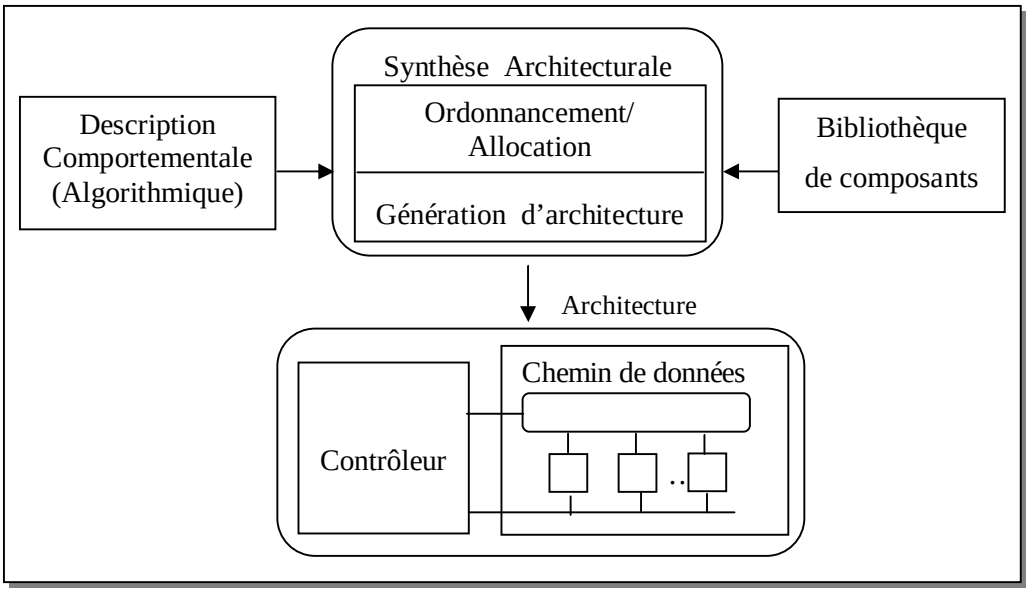



Figure 3.1 : La synthèse de haut niveau

En complément à la description comportementale d'entrée, la synthèse architecturale requiert aussi une bibliothèque de composants, comme illustré sur la figure 3.1 [23]. Ces éléments sont utilisés pour exécuter les opérations de la description [23]. Parmi les objectifs possibles on peut citer : la surface minimale, consommation minimale d'énergie, ou débit maximal. Les contraintes sont souvent définies en termes d'ordre d'exécution, durée d'accomplissement, et surface. Les contraintes et les objectifs dépendent du type de l'application à synthétiser [24].

Avec cette combinaison de la description comportementale, l'ensemble des contraintes et des objectifs, le problème de la synthèse de haut niveau est de trouver la meilleure solution architecturale.

## 3.3. Domaine d'application de la synthèse de haut niveau

La restriction d'un outil de synthèse comportementale à un domaine d'application spécifique permet d'atteindre des résultats efficaces grâce à la réduction de la complexité du

processus de synthèse [7,8]. On distingue dans la littérature deux domaines d'application, les applications orientées flot de données et les applications orientées flot de contrôle.

**Applications dominées par le flot de données**

Ces applications traitent des flots de données réguliers. Le comportement d'une telle application est généralement spécifié comme un ensemble périodique d'opérations à exécuter sur chaque nouvel ensemble de données. Les entrées et les sorties forment des signaux à débit fixe. Les applications typiques sont les filtres numériques, les codeurs/décodeurs, les circuits réalisant les algorithmes de cryptage.

**Applications dominées par le flot de contrôle**

Ces applications sont pilotées par un ensemble de commandes à interpréter. Chaque commande, séquence de contrôle, peut impliquer la lecture ou l'écriture de structures de données complexes et l'exécution d'un algorithme spécifique relatif à cette commande. Les instructions exécutées peuvent dépendre des données, de la communication avec l'environnement (protocoles de requête/acquittement), des interruptions, etc. Les applications typiques sont les contrôleurs, applications de télécommunications, convertisseurs de protocoles, etc.

La conception de systèmes complexes nécessite souvent la combinaison de composants orientés flot de données et ceux orientés flot de contrôle. Dans ce cas, le système doit être décomposé en modules orientés flot de données et modules orientés flot de contrôle. Ce découpage est effectué généralement par les concepteurs lors de la spécification du système permettant ainsi l'utilisation des outils de synthèse orientés aux domaines d'application spécifiques[8].

## 3.4. Représentation interne

Deux types de représentation internes ont été conçus pour les deux principaux domaines d'application: les graphes de flot de données **DFG** (**D**ata **F**low **G**raph) et les graphes de flot de contrôle **CFG** (**C**ontrol **F**low **G**raph). La représentation interne des systèmes complexes est présentée dans les sections suivantes (section 3.4.3).

### 3.4.1. Graphe de flot de contrôle

Un graphe de flot de contrôle est un graphe G=(V,E) où les sommets correspondent à ce que l'on appelle des blocs de base, séquence d'instructions ne contenant aucune instruction de contrôle, et les arcs décrivent le flot de contrôle.

**Définition 3.1 :Graphe de flot de contrôle (CFG)**

Un graphe de flot de contrôle est un graphe orienté G=(V,E), où:

- $V=\{v1,\ldots,vn\}$ est un ensemble fini de sommets, et
- $E \subset VxV$ est une relation de flot de contrôle dont les éléments sont des arcs orientés de séquencement.

**Définition 3.2 :Sommets du graphe de flot de contrôle**

Les sommets sont de trois classes :

- Les sommets d'instruction représentent les instructions simples telles que les affectations de variables, les opérations arithmétiques et logiques et les appels de

procédures. Cette classe de sommets est représentée par le sous-ensemble de $V, V_0$. Ces sommets possèdent un seul successeur immédiat.

- Les sommets de synchronisation correspondent à une attente d'événement extérieur tel qu'un changement de valeur d'un signal. Cette classe de sommets est représentée par le sous-ensemble $V_a$. Ces sommets ont deux successeurs dont le premier est le même sommet modélisant l'attente par un rebouclage.
- Les sommets de branchement représentent les instructions conditionnelles telles que IF, CASE, WHILE. Cette classe de sommets est représentée par le sous-ensemble $V_b$. Ces sommets peuvent avoir plusieurs successeurs immédiats.

**Définition 3.3 :Arcs du graphe de flot de contrôle**

Un arc (vi,vj) représente une relation de séquencement entre deux instructions. Un arc (vi,vj) est étiqueté par une expression booléenne Condij et signifie que l'instruction représentée par vj sera exécutée si l'instruction représentée par vi est exécutée et que l'expression Condij est vraie.

Dans ce modèle de graphe de flot de contrôle, il est supposé que les expressions des arcs issues d'un sommet représentant une instruction de branchement sont exclusives deux à deux et que pour un échantillon de données quelconque, une seule des expressions est vraie. Plus formellement, si $v_i$ est un sommet de branchement et que $v_1,\ldots,v_k$ sont les k successeurs immédiats de $v_i$, alors :

- $Condi,i_m \wedge Condi,i_h = faux$ , pour tout $m,h \in \{1,\ldots,k\}$ et $m<>h$.
- $\sum_{n=1}^{k} Cond\ i,in = vrai$ .

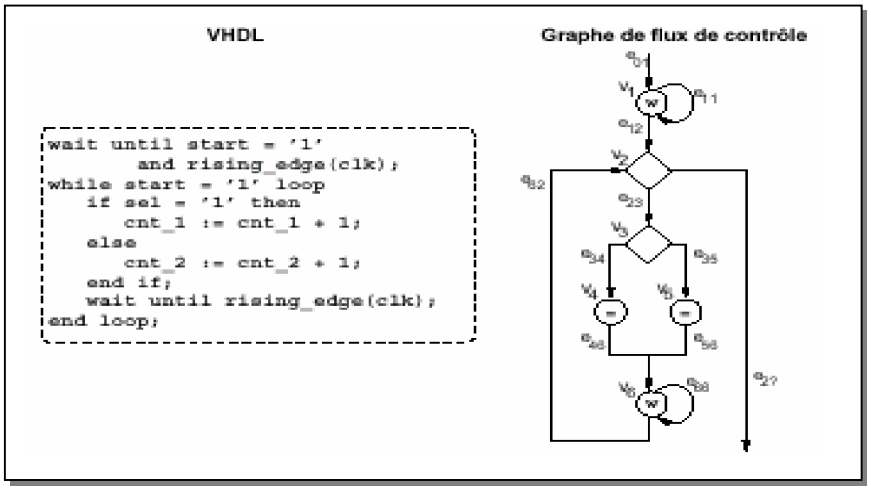



Figure 3.2 : Graphe de flot de contrôle

## 3.4.2. Graphe de flot de données

Les graphes de flot de données sont l'une des représentations les plus utilisées dans le domaine de la synthèse comportementale, vu l'évolution des ordinateurs vers des architectures massivement parallèles [8]. Les sommets du graphe de données représentent des opérations telles que les opérations arithmétiques et logiques alors que les arcs représentent les valeurs. Les arcs sont généralement associés aux variables mémorisant les valeurs intermédiaires dans un calcul complexe. Ainsi, les arcs peuvent être considérés comme une

représentation des dépendances de données entre les diverses opérations qui déterminent le parallélisme et les séquences des opérations.

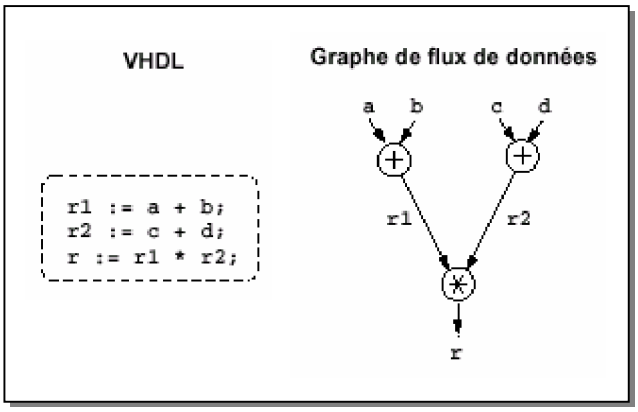



Figure 3.3 : Graphe de flot de données

**Définition 3.4 Graphe de flot de données (DFG)**
Un graphe de flot de données est un graphe orienté G=(V,E), où :

- $V=\{v_1,\ldots,v_n\}$ est un ensemble fini de sommets, et
- $E \subset VxV$ est une relation asymétrique de dépendance de données dont les éléments sont des arcs orientés.

Les sommets correspondent aux opérations et un arc orienté $e_{i,j}$ de $v_i$ à $v_j \in V$ existe, si la donnée produite par l'opération $o_i$ (représentée par $v_i$ ) est consommée par l'opération $o_j$ (représentée par l'opération $v_j$).

## 3.4.3. Représentation interne des applications mixtes

Pour les applications mixtes, les spécifications comportementales contiennent la combinaison de parties de contrôle décrivant des protocoles de communication et de parties dominées par les opérations de calcul. Les deux représentations internes déjà présentées ne permettent généralement pas une représentation efficace de ces descriptions complexes. Alors on s'est dirigé vers des solutions qui consistent à étendre et/ou à combiner les deux approches, tel que le graphe de flot de données et de contrôle (CDFG). Cette représentation étend le modèle pur du graphe de flot de données par les sommets de contrôle pour les instructions conditionnelles (IF, CASE, WHILE, etc.). Ce modèle est adapté à des applications dominées principalement par le flot de données, et est utilisé par plusieurs outils de synthèse tels que HERCULES, CALLAS, HIS, BC[25], etc.

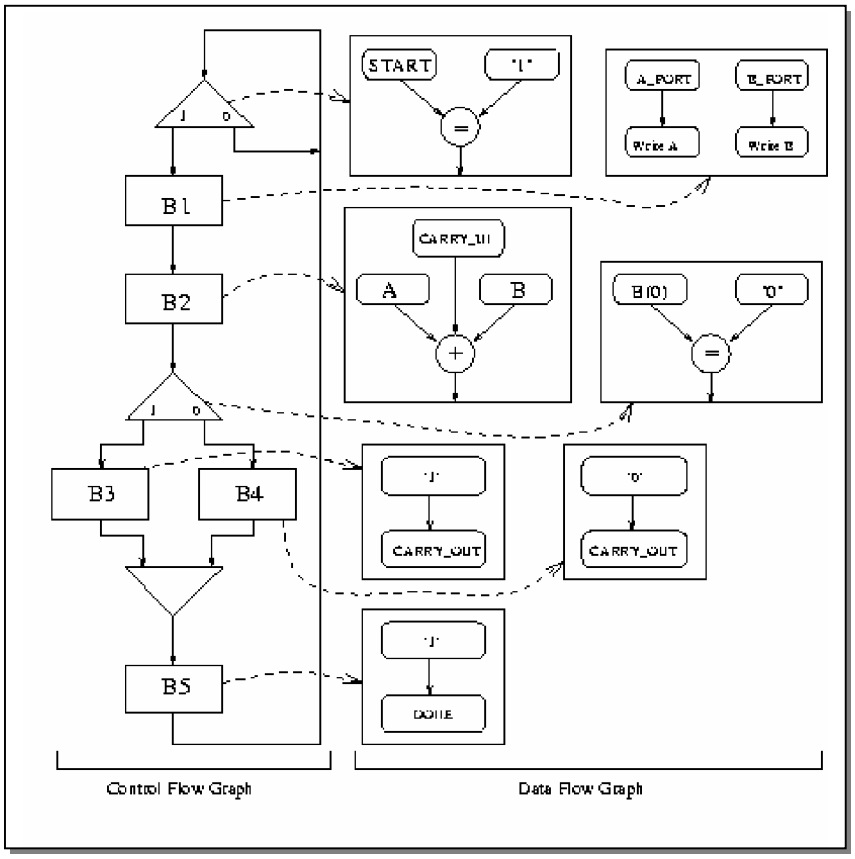



Figure 3.4 : Graphe de flot de données et de contrôle

Certains outils de synthèse utilisent des représentations internes qui leur sont spécifiques. MIMOLA utilise une représentation arborescente similaire aux arbres syntaxiques des compilateurs logiciels. CATHEDRAL utilise une représentation spécifique appelée graphe de flux de signaux (**S**ignal **F**low **G**raph) dérivée directement du langage Silage.

## 3.5. Flot de la synthèse de haut niveau

La synthèse de haut niveau transforme une description comportementale au niveau algorithmique en une description structurelle au niveau transfert de registres. La description comportementale décrit la fonctionnalité qui doit être réalisée par le système synthétisé. Dans une telle description les calculs sont décrits à l'aide d'éléments atomiques appelés *opérations*. Les opérations manipulent des éléments de stockage de données appelés *variables.* La spécification algorithmique comprend des opérations relationnelles, arithmétiques et logiques appliquées aux variables ; et des structures de contrôle telles que les boucles, les instructions de branchement et les appels de procédure. Cette description est écrite en langage de programmation ordinaire ou en langage de description de matériel spécial [22] tel que VHDL [9], Verilog[10], HardwareC[11] et Silage [12]. Une description comportementale peut être écrite par le concepteur même, ou générée par des outils de synthèse de plus hauts niveaux d'abstraction (niveau d'abstraction système).

L'architecture résultante est composée d'un contrôleur et d'un chemin de données [23,24]. Le chemin de données est décrit à l'aide d'instances d'unités fonctionnelles (additionneurs, multiplieurs, UALs et unités logiques), unités de stockage (RAMs, ROMs,

registres, etc), et d'unités de communication interconnectées (bus et multiplexeurs). Ainsi la partie opérative d'un circuit est représentée comme un ensemble de composants interconnectés «netlist ». L'ensemble de ces composants est appelé des ressources. Le contrôleur est définit comme une machine d'états finis qui détermine à chaque cycle d'horloge quelles opérations de la partie opérative doivent être exécutées et par quelles unités fonctionnelles. Le processus de synthèse de haut niveau fait correspondre les variables et les opérations de la description comportementale aux unités fonctionnelles et registres dans la description RTL respectivement.

Un ensemble d'algorithmes de synthèse s'applique sur la description d'entrée pour transformer cette spécification initiale graduellement vers une architecture cible [8]. La synthèse de haut niveau est souvent composée ,comme illustré sur la figure 3.4, de 5 étapes [7,8,21] : la compilation de la description comportementale en une représentation interne, l'ordonnancement, l'allocation, l'assignation des ressources, et la génération de l'architecture. Cette décomposition n'est pas adoptée par tous les auteurs, certains d'entre eux ajoutent une étape de pré-traitement appelée «partitionnement » [4,23], bien que d'autres la considère comme une tâche de la synthèse au niveau système; d'autres auteurs négligent l'étape d'assignation et la considère comme une sous-tâche de l'allocation [22,23]. Il est important de noter que seule l'étape d'ordonnancement est spécifique à la synthèse comportementale, les autres étapes peuvent être accomplies à l'aide d'outils de synthèse au niveau transfert de registres [7,8].

Dans cette section nous présentons les spécifités des différentes étapes de la synthèse de haut niveau. La première étape de la synthèse de haut niveau est la compilation de la description comportementale en une représentation interne. La plupart des approches utilisent les représentations basées sur les graphes (Graph-based représentations), discutés dans la section 3.3, présentées par le CFG,DFG et le CDFG.

Les trois étapes suivantes forment le cœur de la synthèse de haut niveau : l'ordonnancement des opérations, l'allocation des ressources, et l'assignation des ressources. Vu leur importance, ces tâches ont fait le sujet de plusieurs études et différents algorithmes ont été publiés[8,16,19,21,22,24,26,27,28,29,30,31,58,77,78,79,80]

- L'ordonnancement répartit les opérations en étapes de contrôle. Une étape de contrôle est l'unité de temps de base dans les systèmes synchrones, elle correspond à un cycle d'horloge.
- L'allocation est la tâche d'estimation du type et du nombre de ressources nécessaires pour aboutir à la description RTL.
- L'assignation est la tâche d'assignation des opérations aux unités fonctionnelles, des variables aux unités de stockage, des transferts de données aux unités d'interconnexion.

Le rôle de la synthèse architecturale est donc de trouver le nombre et le type de ressources matérielles nécessaires à l 'implantation des opérations (allocation), définir les instants d'exécution des opérations (ordonnancement) et de définir pour chaque opération quelle est la ressource qui l'implante (assignation) [24]. Ces tâches sont fortement interdépendantes. Car l'allocation nécessite des informations sur les opérations qui sont exécutées simultanément et ne peuvent donc pas partager une même ressource. Inversement, l'ordonnancement a besoin des informations sur l'allocation des ressources pour savoir quelles opérations peuvent être exécutées en même temps. Donc, le cas où l'ordonnancement est effectué avant l'allocation , laisse peu de marge à d'éventuel parallélisme entre opérations du fait que l'ordonnancement fixe la distribution des opérations par rapport aux cycles d'horloge [23]. Dans le cas contraire, des restrictions supplémentaires sont imposées à l'étape

d'ordonnancement, puisque à chaque élément est lié un délai d'exécution, les étapes exécutées fixent en grande partie l'exploitation temporelle par les diverses opérations[23].

Pour avoir une conception optimale, l'idéal serait l'exécution simultanée de l'ordonnancement et de l'allocation [22,23]. Cependant, à cause de la complexité de résolution de tels problèmes (problème NP-complet), plusieurs systèmes exécutent les étapes d'ordonnancement et d'allocation séparément, l'une après l'autre [32,33,34,35]; d'autres systèmes les exécutent comme des tâches entrelacées. Mais même dans le cas où l'ordonnancement et l'allocation sont considérés séparément, ils restent des problèmes NP-complets. Donc on peut dire que les trois tâches sont trés difficiles à résoudre de façon exacte à cause de leur nature combinatoire. Pour des raisons pratiques, il suffit de trouver de bonnes solutions en des temps raisonnables [4].

Une fois ces étapes sont achevées, il ne reste qu'à effectuer l'étape de génération d'architecture, dernière étape dans le flot de la synthèse de haut niveau. Cette étape permet de générer les parties opérative et contrôle du circuit.

### 3.5.1. Description comportementale

Une description comportementale est le point de départ pour la synthèse de haut niveau. C'est une description algorithmique du comportement du circuit qui ne contient aucune information ni sur sa structure ni sur la façon dont il sera réalisé [8].

Cette description est écrite en langage de description de matériel HDL (**H**ardware **D**escription **L**anguage). Certains langages permettent la description de tout type de circuit tels que VHDL [9], Verilog [10], HardwareC [11]. D'autres langages sont dédiés tout spécialement à des circuits de traitement de signal, tels que Silage [12], Signal [13]. Quelques-uns de ces langages permettent la modélisation à plus d'un niveau d 'abstraction comme VHDL et Verilog. D'autres langages sont dédiés au niveau comportemental comme Hardware C, Silage. Le choix du langage de description dépend de son admissibilité par les outils de simulation et de synthèse, et de sa capacité à abstraire le comportement souhaité.

Il existe deux styles de description, le style concurrent et le style séquentiel. Dans le style concurrent, les instructions de la description sont exécutées en parallèle et ce, perpétuellement. Par contre dans le style séquentiel, les instructions sont décrites dans un ordre particulier. VHDL et Verilog permettent des descriptions séquentielles et des descriptions de type flot de données. Hardware C permet des descriptions séquentielles, cependant les descriptions de type flot de données sont limitées [23]. Pour la description d'entrée d'une application «orientée flot de contrôle », il est plus commode d'utiliser un style séquentiel. Par contre, pour une application «orientée flot de données » il est plus commode d'utiliser un style concurrent.

VHDL est trés utilisé pour la synthèse comportementale. Or comme la sémantique du VHDL a été définie avec en vue la simulation, les outils de synthèse limitent les instructions VHDL acceptées à un sous-ensemble du langage [23]. De nombreux outils ont choisi VHDL comme langage d'entrée tels que V-Synth de Honeywell, HIS d'IBM. Verilog est moins utilisé pour la synthèse comportementale, il est utilisé par l'outil de synthèse comportementale Architect's Workbench développé à Carnegie Mellon [37]. Le langage Hardware C est utilisé comme langage d'entrée par le système Olympus de Stanford. D'autres outils utilisent le langage Silage, spécialement adapté à la modélisation de circuits à temps réel, tels que Cathedral de l'IMEC [45,46] et Hyper de Berkeley.

### 3.5.2. Compilation de la description comportementale

La première étape de la synthèse de haut niveau consiste à traduire la description comportementale en une représentation interne à l'outil de synthèse. Il existe deux classes de formes intermédiaires. Les premières se fondent sur des graphes, exprimant les dépendances de contrôle et de données contenues dans la description initiale, on les appelle formes intermédiaires orientées langage (représentations discutées dans la section 3.4). Les secondes se fondent sur des machines d'états finis, on les appelle formes intermédiaires orientées architecture. Cette étape est immédiate, lorsqu'on utilise une forme intermédiaire orientée langage, puisque les graphes sont très proches de la plupart des langages de description comportementaux. Par contre, lorsque l'on utilise une forme intermédiaire orientée architecture, cette étape nécessite des transformations complexes pour obtenir une machine d'états finis [7]. Pendant cette étape, des transformations connues des compilateurs de langages de programmation sont effectuées. De telles transformations incluent la propagation de constantes, l'élimination du code mort, la simplification des expressions, la pré-évaluation des conditions, et l'expansion des procédures, etc.

### 3.5.3. Ordonnancement

L'ordonnancement est le partitionnement de la description comportementale en sous-graphes, de manière à ce que chaque sous-graphe s'exécute en une seule étape de contrôle. Une étape de contrôle correspond à un état d'une machine d'états finis FSM (**F**inites **S**tates **M**achine) [38]. Chaque étape de contrôle peut inclure plusieurs opérations à exécuter en parallèle. Ce parallélisme est conditionné par l'algorithme d'allocation de ressources, qui doit être capable d'allouer le nombre suffisant de ressources pour l'exécution des opérations parallèles. L'ordonnancement est une tâche importante dans la HLS, parce que c'est durant cette phase que le nombre de cycles de contrôle est fixé ainsi que le parallélisme des opérations.

Les différents algorithmes d'ordonnancement peuvent être distingués selon trois critères [1,22] :

- l'interaction entre l'ordonnancement et l'allocation ;
- la fonction objectif et les contraintes ;
- le type d'algorithme d'ordonnancement.

L'ordonnancement et l'allocation sont fortement interdépendants, comme il a été décrit précédemment.

Le deuxième critère concerne la fonction objectif et les contraintes. Une fonction objectif est une mesure que le système essaie de maximiser ou de minimiser. Les contraintes quant à elles, sont des conditions qui doivent être satisfaites [1]. Les fonctions objectifs et les contraintes concernent le nombre de ressources matérielles (surface) et le temps. Il existe deux classes de problèmes d'ordonnancement : ordonnancement sous contraintes matérielles et ordonnancement sous contraintes temporelles. Dans le premier cas, la quantité des ressources matérielles étant bornée, l'ordonnancement cherche à minimiser le temps d'exécution (minimiser les étapes de contrôle) en tenant compte des unités fonctionnelles disponibles. Dans le deuxième cas, le temps d'exécution étant borné, l'ordonnancement cherche à minimiser la surface du circuit en tenant compte du nombre maximum des étapes de contrôle.

Ainsi la solution optimale est généralement définie avec des compromis entre les caractéristiques temporelles et de surface (ressources). Cependant dans certains cas, certaines

contraintes sont imposées et doivent être respectées. La solution à retenir, dans ce cas, est celle satisfaisant ces contraintes et optimisant la fonction objectif [23].

Le troisième critère concerne le type d'algorithme d'ordonnancement utilisé. Il existe deux types d'algorithmes : des algorithmes transformationnels et des algorithmes itératifs/constructifs. Les algorithmes transformationnels débutent avec un ordonnancement initial (par exemple sérialiser au maximum ou paralléliser au maximum). Puis des transformations sont appliquées pour obtenir d'autres ordonnancements, par exemple, utiliser la recherche exhaustive, branch-and-bound ou les heuristiques [22]. Alors que les algorithmes itératifs/constructifs construisent l'ordonnancement en ajoutant les opérations une à la fois, jusqu'à ce qu'il n'y ait plus d'opérations à ordonnancer. Ces algorithmes diffèrent dans le critère de sélection utilisé pour choisir la prochaine opération à ordonnancer.

Vu l'importance et la complexité de l'étape d'ordonnancement, plusieurs algorithmes ont été publiés. Certaines techniques, des plus connues, d'ordonnancement exactes sont basées sur la programmation en nombres entiers ILP (**I**nteger **L**inear **P**rogramming) [39,50]. L'ILP essaie de minimiser le nombre de ressources et le temps d'exécution à la fois (ordonnancement sous contraintes temporelles et contraintes de surface). Cependant, le temps d'exécution de l'algorithme croît exponentiellement avec le nombre des variables et le nombre d'inégalités [16]. Donc, bien qu'il a été montré qu'elles sont relativement efficaces, les techniques ILP ne sont pas facilement généralisables à l'ordonnancement orienté contrôle. En pratique, l'approche ILP est appliquée seulement aux problèmes assez simples.

Etant donné que la méthode ILP est impraticable dans le cas des circuits complexes, des méthodes heuristiques qui marchent efficacement au prix de l'optimalité ont été développés [26]. Les algorithmes d'ordonnancement heuristiques utilisent généralement des techniques itératives/constructives. Les approches constructives les plus simples sont l'ordonnancement au plus-tôt ASAP (**A**s **S**oon **A**s **P**ossible) et l'ordonnancement au plus tard ALAP (**A**s **L**ate **A**s **P**ossible).

L'approche ASAP est utilisée par les systèmes FACET de General Electric [32], early CMUDA, MIMOLA, et Flamel [22]. Avec l'approche ASAP, premièrement les opérations sont classées dans une liste suivant leur ordre topologique. Ensuite, les opérations sont placées, une par une, dans la plutôt possible étape de contrôle. Une approche similaire est d'ordonnancer les opérations au plus tard ALAP. Cet algorithme détermine la date d'exécution au plus tard d'une opération. Les chemins critiques dans le graphe de flot peuvent être déterminés en prenant l'intersection des ordonnancement ASAP et ALAP. Les opérations qui apparaissent dans les mêmes étapes de contrôle, dans les deux cas, appartiennent aux chemins critiques. Le problème avec l'ordonnancement ASAP, ainsi que l'ordonnancement ALAP, est que les opérations sur les chemins critiques n'ont aucune priorité. D'où, des opérations moins critiques peuvent être ordonnancées en premier et retardant ainsi les opérations critiques, dans le cas où des limites de ressources sont imposées à l'ordonnancement [16,22]. L'ordonnancement par liste (List scheduling) résout ce problème, en utilisant un critère global pour la sélection de la prochaine opération à ordonnancer. Pour chaque étape de contrôle, est dressée une liste des opérations qui peuvent être exécutées durant cette étape. Chaque opération est ordonnancée si les ressources nécessaires sont disponibles, sinon elle est différée à la prochaine étape de contrôle. Dans certains cas, cette approche marche presque aussi bien que le branch-and-bound [22].

La fonction de priorité peut utiliser l'*urgence* de l'opération, c'est le cas pour le système Elf, qui est définie comme le nombre minimum d'étapes de contrôle (longueur du plus court chemin) depuis l'opération jusqu'à la plus proche contrainte de temps. Dans le système Slicer la fonction de priorité est basée sur la *mobilité*, qui est la différence entre les valeurs ASAP et ALAP d'une opération [22]. Une *priorité composée* est utilisée dans le système MAHA [35], où les opérations sur les chemins critiques sont ordonnancées en

premier. Ensuite les autres opérations sont ordonnancées suivant la moindre mobilité. Le système HAL effectue l'ordonnancement par liste avec la *force* comme priorité [22]. La force entre une opération et une certaine étape de contrôle est proportionnelle au nombre d'opérations du même type qui peuvent être ordonnancées durant cette étape. Ces valeurs sont représentées par des graphes de distribution. Un exemple des algorithmes utilisant ce critère de sélection global est l'ordonnancement dirigé par les forces FDS (**F**orce **D**irected **S**cheduling). L'algorithme FDS utilise les algorithmes d'ordonnancement ASAP et ALAP pour déterminer la portée de chaque opération par rapport aux étapes de contrôle. Ensuite il dresse les graphes de distribution. Cet algorithme essai de minimiser le nombre total d'unités fonctionnelles utilisées dans l'implémentation, par la distribution uniforme des opérations de même type sur tous les états disponibles [16].

Jusqu'à maintenant, on n'en a pris en considération que les structures simples, alors qu'une description d'un circuit contient aussi des structures conditionnelles et de boucles. Plusieurs approches [40] ont été proposés pour l'ordonnancement des structures conditionnelles. L'ordonnancement AFAP (**A**s **F**ast **A**s **P**ossible) et l'un des algorithmes les plus connus [41]. AFAP considère, en premier, tous les chemins d'exécution possible et, ensuite, les ordonnances séparément. Les résultats obtenus pour les différents chemins sont ensuite combinés, par résolution des conflits entre les chemins d'exécution. Pour les structures de boucles, différentes approches ont été proposées tel que le pipelining.

### 3.5.4. Allocation/Assignation

L'allocation et l'assignation des ressources sont des étapes importantes dans la synthèse de haut niveau. Comme énoncé précédemment, l'étape d'ordonnancement répartit les opérations de la description comportementale en étapes de contrôle. La description du circuit, au niveau RTL, est composée de 2 parties : partie contrôle et partie opérative. Le contrôleur correspond à une machine d'états finis ou à une table de transition qui définit le séquencement des opérations. La structure de la partie opérative, ou chemin de données, est définit par les deux étapes d'allocation et d'assignation. L'étape d'allocation détermine le nombre et le type des ressources nécessaires pour l'exécution de la description comportementale [4,7,8,20,23]. Cette étape fixe le nombre et les types des unités fonctionnelles (additionneurs, multiplieurs, UALS, etc.), les unités de stockage (registres, bancs de registres, mémoires) et les unités d'interconnexion (multiplexeurs, bus, etc.). Puisque la bibliothèque des composants RT peut contenir plusieurs types d'unités fonctionnelles, chacune avec des caractéristiques différentes (fonctionnalité, taille, possibilité de pipeline), plusieurs solutions, qui assurent l'exécution de la description comportementale tout en satisfaisant les contraintes de conception, peuvent exister.

L'assignation (binding) des ressources permet de faire correspondre les opérations, variables, et les transferts de données dans le graphe de flot, aux unités fonctionnelles, de stockage, et d'interconnexion, respectivement. Comme pour l'allocation, l'assignation consiste en 3 tâches interdépendantes :

- L'assignation des unités fonctionnelles, fait correspondre les opérations à l'ensemble des unités fonctionnelles sélectionnées.
- L'assignation des unités de stockage, fait correspondre les valeurs (telles que constantes, variables, structures de données tels que tableaux) à des éléments de stockage (ROMs, registres et unités mémoire)
- L'assignation des unités d'interconnexion, fait correspondre les transferts de données à un ensemble d'unités d'interconnexion.

**Allocation/Assignation d'unités de stockage**

Cette tâche fait correspondre les valeurs de la description comportementale, introduites par l'intermédiaire de constantes, variables ou tableaux par exemple, aux unités de stockage telles que registres, bancs de registres, ou mémoires statiques ou dynamiques.

En mode général, les constantes sont implantées à l'aide de mémoires statiques ROMs plutôt qu'à l'aide de constantes «câblées » [23]. Ceci peut être bénéfique par la minimisation du nombre de connexion. Les possibilités de regroupement sont limitées par les temps d'accès aux constantes. Les variables sont quant à elles traduites en registres, ou en mémoires (RAM) dans le cas de tableaux. Comme la minimisation du nombre de ressources est basée sur les possibilités de partage temporel par les entités de la description comportementale, l'algorithme utilisé peut essayer de partager les divers registres entre plusieurs variables compatibles, c'est à dire dont les durées de vie ont une intersection vide. La durée de vie d'une variable et le laps de temps entre l'étape de contrôle durant laquelle une valeur est affectée à la variable et l'étape de contrôle ou elle est utilisée. Les registres qui ne sont à aucun moment accédés simultanément, peuvent être regroupés en bancs mémoire. Cependant, ce type de transformation minimise le nombre de connexions au détriment de la surface du contrôleur puisque celui-ci a à gérer les adresses des RAMs résultantes.

**Allocation/Assignation d'unités fonctionnelles**

L'allocation/assignation d'unités fonctionnelles détermine quels sont les opérateurs ou modules fonctionnels nécessaires, et à quelles opérations ils doivent être affectés pour l'exécution de l'algorithme initial [23]. Cette tâche est effectuée en prenant en compte une bibliothèque prédéfinie d'UFs. Celle-ci énumère les unités fonctionnelles disponibles ainsi que leurs caractéristiques. Les unités fonctionnelles peuvent exécuter aussi bien des opérations standards (opérations arithmétiques ou booléennes), que des opérations complexes définies par le concepteur par l'intermédiaire de procédures et de fonctions.

**Allocation/Assignation d'unités d'interconnexion**

L'allocation d'unités d'interconnexion détermine les ressources (multiplexeurs, bus, etc.) nécessaires pour la communication entre les unités (unités de stockage et unités fonctionnelles) du chemin de données. Le but ici également est de définir le réseau d'interconnexion de façon à minimiser la surface du circuit, et ceci en utilisant un nombre minimal de connexions, multiplexeurs et bus. Ce nombre influe non seulement sur la taille du chemin de données mais également sur la taille du contrôleur.

Il existe principalement deux types d'architectures : une architecture à base de bus et une architecture à base de multiplexeurs. Il existe également des architectures mixtes permettant de combiner multiplexeur et bus.

**Approches de résolution**

Il existe deux approches pour la résolution des problèmes d'allocation/assignation [1]:

- des approches globales dans lesquelles tous les éléments de la description comportementale sont considérés simultanément.
- des approches constructives/itératives dans lesquelles un sous-ensemble des éléments est considéré.

Dans une approche constructive, initialement le chemin de données est vide. Le graphe de contrôle est balayé étape par étape, et le chemin de données est construit graduellement au fur et à mesure de ce balayage [16] par l'ajout d'unités fonctionnelles, de stockage et d'interconnexion tant que c'est nécessaire. Des exemples d'algorithmes

constructifs sont ceux de type " glouton " , de type " branch and bound ". L'approche glouton est basée sur l'algorithme de résolution du " couplage maximal " dans un graphe biparti. Le problème de couplage maximal admet plusieurs solutions, et c'est de la solution choisie que dépend la complexité du réseau d'interconnexion. C'est pour cela, que dans une extension de cette approche, et afin d'obtenir la solution menant à la connectique minimale, on pondère les arcs du graphe biparti. Le poids de chaque arc représente le coût additionnel sur la structure qu'impliquerait l'assignation d'une variable à un registre, si l'approche est appliquée à l'allocation de registres par exemple. Alors qu'avec la technique branch and bound on ne choisi une solution qu'aprés évaluation de son coût et sa comparaison avec le coût de la meilleure solution déjà trouvée.

Bien que les algorithmes constructifs soient simples, et qu'ils résouent des sous-problèmes de façon optimale et en un temps acceptable (puisqu'on s'intéresse par exemple qu'à l'allocation des variables utilisées dans une seule des étapes de contrôle et non pas à toutes les variables), les solutions globales qu'ils donnent peuvent être loin de l'optimal [16]. En vu d'améliorer la qualité des résultats, on utilise les approches globales, où les tâches du problème d'allocation sont découplées, malgré qu'elles soient interdépendantes (même pris séparément, chacune de ces tâches reste un problème NP-complet). Chaque tâche peut être modélisée par l'un des problèmes de la théorie des graphes tels que : le *partitionnement en clique*, l'algorithme *left-edge*, le *coloriage des sommets d'un graphe*, etc.

Le problème du "partitionnement d'un graphe en cliques" est un problème bien connu, qui consiste à construire un graphe (appelé graphe de compatibilité) non orienté, dont les nœuds (sommets) sont les variables, ou les opérations, de la spécification. Une arête entre deux sommets indique que ces deux sommets peuvent partager le même matériel. Ainsi, le problème d'allocation, telle que l'allocation d'unités de stockage, peut être résolu par la décomposition de ce graphe en un nombre minimal de cliques[1]. Ce problème est équivalent au problème du «coloriage des sommets d'un graphe», où on utilise un graphe d'incompatibilité et on cherche à colorier les sommets de ce graphe avec un nombre minimal de couleurs et ce, de telle façon que deux sommets joints par une arête soient de couleurs différentes. Le problème du "partitionnement d'un graphe en cliques" n'a pas d'algorithme efficace (i.e. un algorithme dont le temps de calcul ne croisse pas exponentiellement avec le nombre de sommets et/ou d'arêtes), c'est pour cela qu'on utilise des approches heuristiques. L'une des heuristiques les plus connues pour résoudre ce problème est celle due à Tseng et Siewiorek [32]. En plus, la méthode de partitionnement en cliques appliquée à l'allocation de registres permet de minimiser le nombre de registres sans prendre en compte le coût des connexions. Afin de palier cet inconvénient, une amélioration de cette méthode a été proposée et consiste à pondérer les arêtes, reflétant ainsi l'effet du partage d'un registre entre deux variables sur la complexité d'interconnexion.

Kurdahi et Parker ont appliqué " l'algorithme left-edge " [43] pour l'allocation de registres. Cet algorithme a été, à l'origine, proposé pour le routage de chaînes par l'assignation de piste. Le principe est d'utiliser au mieux un registre en allouant le plus grand nombre de variables de durées de vie non recouvrantes et en créant un nouveau registre lorsqu'on ne peut plus ajouter de variables au registre courant. Bien que cet algorithme alloue le nombre minimal de registres, il ne permet pas de prendre en compte le coût associé aux interconnexions.

[1]Un clique est un sous-ensemble de sommets tel que deux sommets quelconques de ce sous-ensemble soient liés par une arête.

## 3.5.5. Génération d'architecture

La génération d'architecture est la dernière étape dans la synthèse de haut niveau. Sa tâche est de produire une description RTL du circuit synthétisé, qui est exploitée par les outils de synthèse de plus bas niveau, cette architecture composée de la partie contrôle et de la partie opérative, représente le résultat de la synthèse comportementale, elle est construite à partir des résultats des étapes précédentes.

- **Partie opérative**

Cette partie regroupe les unités fonctionnelles, les unités de stockage et les unités d'interconnexion qui permettent l'exécution des opérations de la spécification initiale.

- **Partie contrôle**

Une fois le chemin de données obtenu, le contrôleur ou la partie contrôle peut alors être synthétisé. La partie contrôle se présente au niveau transfert de registres sous forme d'une machine d'états finis. Les étapes de contrôle et leur séquencement ont été définies lors de l'ordonnancement. Le contrôleur inclut la séquence des étapes de contrôle et l'activation des composants du chemin de données pendant chaque étape de contrôle.

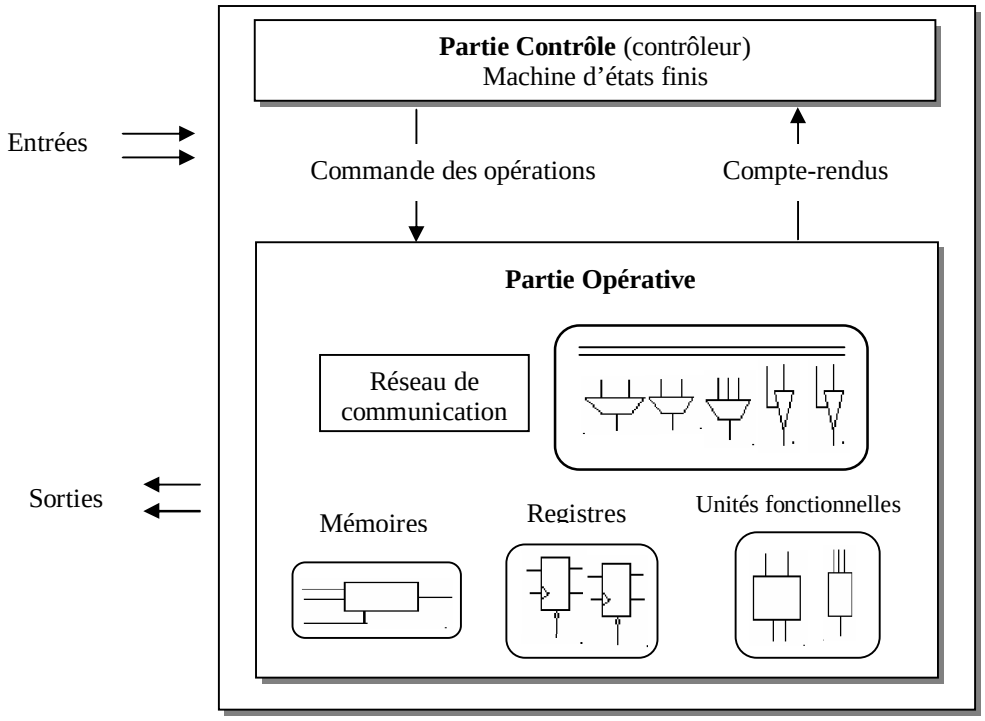



Figure 3.5 : Architecture au niveau transfert de registres

Ainsi, le contrôleur commande l'exécution des opérations à être effectuées par la partie opérative grâce à des signaux. Ces signaux, pour chaque état, reconnus comme des signaux de commande ou de contrôle, activent et désactivent aux moments opportuns les éléments du chemin de données pour effectuer les actions associées à cet état. Le chemin de données, quant à lui, engendre des comptes-rendus utilisés lors de l'évaluation de certaines conditions par la partie contrôle [23].

## 3.6. Outils de synthèse de haut niveau

Un outil de synthèse comportementale permet de transformer une description algorithmique en une architecture cible au niveau transfert de registres. Une multitude d'algorithmes de synthèse comportementale ont été publiés. Néanmoins, très peu d'entre eux ont passé le cap des recherches pour former des outils de synthèse complets, réalisant toutes les étapes nécessaires pour la génération d'une architecture en partant d'une description comportementale [23], la plupart de ces recherches ne s'intéressaient qu'à une étape de synthèse. Peu de ces outils sont parvenus à atteindre le niveau de commercialisation et s'imposer sur le marché, à l'exception de quelques outils. A cela plusieurs raisons :

- la plupart de ces outils ont été développés en milieu universitaire et les efforts ont porté plus sur les aspects fondamentaux que sur le développement d'outils commerciaux.
- la réticence des designers à migrer vers le niveau comportemental.
- la qualité des designs (tout au moins générés à l'aide des premiers outils) souvent peu compétitives.
- la difficulté d'intégration de ces outils dans des environnements de conception existants.

Les systèmes de synthèse architecturale existants peuvent être classés en deux grands groupes, même classification que celle qui existe pour les circuits : les outils orientés flot de données ou systèmes ciblant la synthèse de circuits de traitement de signal DSP (**D**IGITAL **S**IGNAL **P**ROCESSING) et les outils orientés flot de contrôle. Il existe aussi des outils à application générale. Le tableau 3.1 [7] liste quelques outils orientés flot de données. Le tableau 3.2 [7] liste quelques outils de synthèse orientés flot de contrôle.

| | |
|---|---|
| • **Cathedral-2/3** | de l'IMEC, |
| • **GAUT** | du CENT, |
| • **Lagger** | de l'université de Berkeley, |
| • **Behavioral Compiler** | de Synopsys Inc, |
| • **Mistral** | de Mentor Graphics Corp, |
| • **COSSAP** | de Synopsys Inc, |
| • **SPW** | de Cadence Alta Group. |

Tableau 3.1 : Outils de synthèse orientés flot de données.

Cette section décrit trois outils de synthèse de haut niveau représentatifs des trois principales classes, citées ci-dessus, d'outils de synthèse architecturale : AMICAL, outil universitaire orienté flot de contrôle, CATHEDRAL, outil orienté flot de données, et Behavioral Compiler, outil à orientation mixte, qui est peut-être l'outil commercial le plus connu dans le monde de la synthèse [8].

| | |
|---|---|
| • **SAW** | de l'université de Carnegie Mellon, |
| • **Olympus** | de l'université de Stanford, |
| • **Amical** | de TIMA-INP-Grenoble, |
| • **Mimola** | de l'université de Dortmund, |
| • **HIS** | d'IBM, |
| • **Behavioral Compiler** | de Synopsys Inc, |

Tableau 3.2 : Outils de synthèse orientés flot de contrôle.

## 3.6.1. Behavioral Compiler

Synopsys commercialise un des outils de synthèse comportementale sous le nom de Behavioral Compiler [25]. C'est le plus connu des rares outils commerciaux existants [8]. Il est applicable aux applications orientées flot de données et/ou flot de contrôle. Certains auteurs [8] considèrent que le domaine d'application de Behavioral Compiler est essentiellement celui des circuits orientés flot de données, dominés par les calculs arithmétiques, et qu'en matière de contrôle, il n'est pas considéré comme la solution optimale. C'est pourquoi, Synopsys a lancé un autre produit pour ces applications, le Protocol Compiler [8].

Behavioral Compiler synthétise automatiquement une structure RTL à partir d'une description comportementale et un ensemble de contraintes de haut niveau. Il offre la possibilité au concepteur d'explorer différentes architectures plus rapidement qu'une conception traditionnelle, ainsi que des optimisations au niveau de la surface et des performances en vitesse du système. Le modèle d'entrée de Behavioral Compiler est décrit soit en VHDL soit en VERILOG, tous deux de type algorithmique. L'architecture générée est du type courant, elle est composée d'un contrôleur réalisé par une machine d'états finis, d'un chemin de données et de mémoires. La liaison entre les deux parties s'effectue par des multiplexeurs uniquement. La sortie RTL de Behavioral Compiler écrite dans un format interne ne peut entrer que dans l'outil de synthèse RTL SYNOPSYS Design Compiler [7,8]. L'architecture cible est illustrée par la figure 3.6 .

Les principales caractéristiques de traitement effectués lors de la synthèse, ou plus précisément lors de l'ordonnancement, sont [7,23] :

-Le chaînage (chaining) : ordonnancement d'opérations qui se suivent, avec dépendance de données, dans une même étape de contrôle. Elles s'exécutent en séquence.
-Le multi-cycle (multi-cycling) : les opérations nécessitant un temps supérieur à la période de l'horloge (opérations multicycles) sont acceptées et ordonnancées automatiquement.
-Le pipelining de boucles (loop pipelining ou loop folding) : l'existence de nombreuses constructions de boucles dans la description comportementale, peut limiter le débit des données si elles sont ordonnancées en séquence, la parallélisation et la mise à plat de boucles augmente le parallélisme entre les tâches à ordonnancer. Ainsi, cette technique permet une utilisation optimale des ressources et augmente le débit du circuit en permettant à de nouvelles données d'être prises en compte à chaque cycle d'horloge par exemple.
-Le découpage technologique de plusieurs tableaux en une même mémoire, si leurs indices ne se chevauchent pas [23].

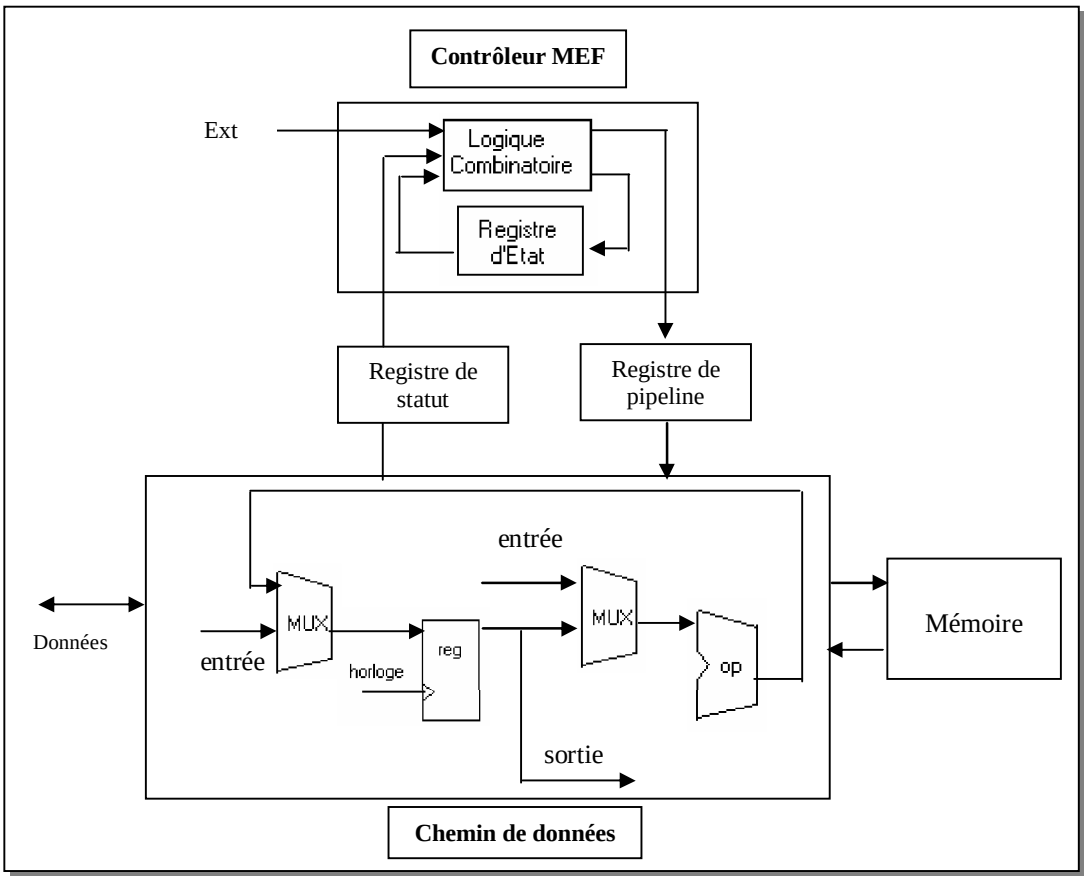



Figure 3.6 : Architecture cible de Behavioral Compiler

### 3.6.1.1. Flot de synthèse de BC

La figure 3.7 [7,8] illustre le flot de synthèse de Behavioral Compiler et synthétise le chemin de données , les mémoires et le contrôleur pour une description comportementale. Après avoir analysé la syntaxe et la fonction de la description comportementale, BC élabore le circuit pour préparer l'ordonnancement. La synthèse est effectuée sous les contraintes imposées par le concepteur dans la première phase de synthèse. Ensuite, BC utilise la bibliothèque technologique et les composants réutilisables pour la détermination des infos concernant les délais des opérations du chemin de données, de la machine d'états finis du contrôleur, des registres et du réseau de multiplexeurs [8]. Une fois l'estimation de performance est effectuée, BC exécute l'ordonnancement, l'allocation des ressources et génère la machine d'états finis pour le contrôleur, des comptes rendus détaillés sur l'ordonnancement et l'allocation sont alors disponibles. Ils servent de base à une exploration des architectures possibles en modifiant les contraintes avant l'optimisation au niveau transfert de registres. Après avoir opté pour celle qui convient le plus, le concepteur peut effectuer la simulation au niveau transfert de registres et, si le résultat est satisfaisant, BC, avec Design Compiler, construit et optimise la description au niveau RTL [7].

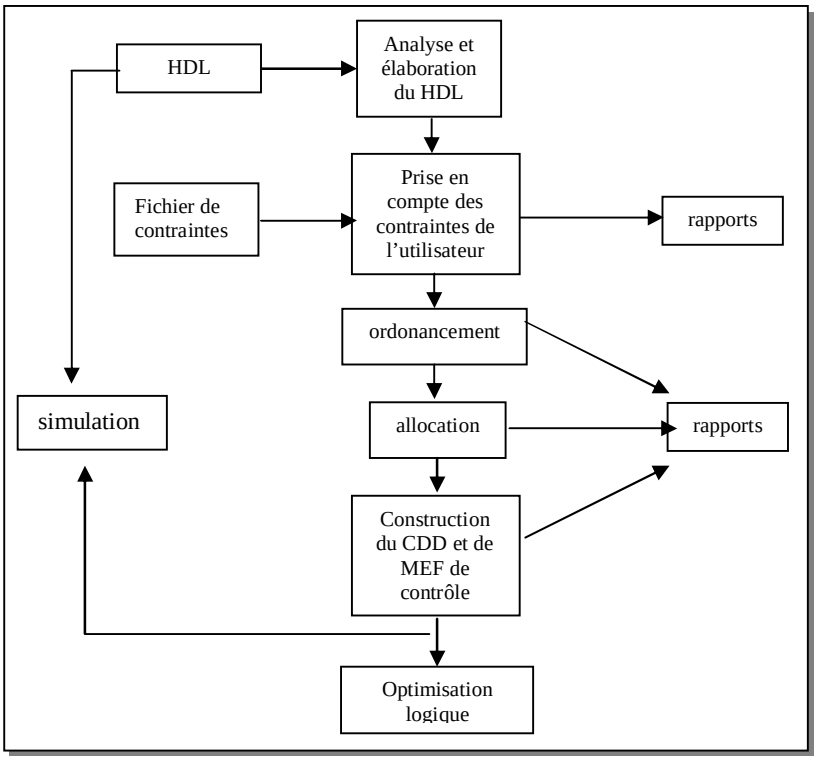



Figure 3.7 : Flot de synthèse de BC

**Mode d'ordonnancement de Behavioral Compiler**

Behavioral Compiler distingue trois principaux modes d'ordonnancement. Ces modes d'ordonnancement se différencient selon les changements autorisés des comportements des entrées/sorties et/ou le nombre de cycles de contrôle insérés entre deux instructions d'attente (instructions wait) de la description comportementale [8]. Ces trois modes d'ordonnancement sont :

- Mode en un cycle (cycle-fixed) : Le comportement externe de la description au niveau transfert de registres correspondant exactement à la description comportementale en ce qui concerne le nombre de cycles de l'exécution et des lectures et des écritures des entrées/sorties. Les entrées et sorties sont décrites au cycle d'horloge prés. Ce mode convient à la vérification des descriptions aux niveaux comportementale et RTL [7].

- Mode des macro-états (superstat-fixed) : Permet de définir les entrées et sorties par rapport à des super-états ou macro-cycles. Ces macro-états sont désignés par les instructions d'attente. Les opérations situées entre deux macro-états peuvent être décomposées par l'ordonnancement en plusieurs cycles de contrôle. L'ajout des cycles implique le changement du comportement des entrées/sorties. Les transitions de ces signaux peuvent être décalées dans le temps , mais leur ordre reste toujours inchangé. Ce mode permet une exploration architecturale plus importante avec des opérateurs à exécution variable (même fonction assurée mais avec des caractéristiques temporelles différentes) [23]. C'est le mode le plus utilisé en pratique [7].

- Mode avec des entrées/sorties flottantes (Free-Floating) : Toutes les opérations, y compris les opérations d'entrées/sorties peuvent flotter en dehors des instructions wait. Ce mode

ressemble au mode des macro-états avec la différence que l'ordre des transitions des entrées/sorties peuvent être changées mais les dépendances de données sont tout de même sauvegardées. Ce mode permet une exploration architecturale plus vaste et offre plus de possibilités de chaînage et de parallélisation des opérations [23], mais les concepteurs évitent l'utilisation de ce mode d'ordonnancement car le comportement du circuit produit par le processus de synthèse devient très éloigné par rapport à celui de la description initiale [8].

#### 3.6.1.2. Points forts et limitations

BC offre le fort avantage de traiter à la fois des circuit de type flots de données et/ou de contrôle, et permet une utilisation aisée des mémoires (migration automatique des tableaux en mémoires et fusion possible de plusieurs tableaux en une seule mémoire) [23].

Cependant, bien que BC offre trois modes d'ordonnancement différents, on peut dire qu'il regroupe les caractéristiques de trois outils différents, le traitement pour chaque mode est plus restreint [23]. En plus, BC impose un style d'écriture très restreint, ce qui est dû à la représentation interne et à l'algorithme d'ordonnancement. En particulier, le style d'écriture est extrêmement restreint au sein des sous-programmes où il ne se limite qu'à la description des opérations combinatoires. Ni instructions conditionnelles, ni boucles, ni instructions d'attente ne peuvent y être placées [8]. Mais l'inconvénient majeur de cet outil est le temps de calcul machine nécessaire pour générer une solution qui dure plusieurs heures, ce qui limite le nombre d'architectures que le concepteur peut analyser, malgré que BC permet la génération et l'exploration des architectures alternatives à partir de la même description comportementale avec des contraintes différentes.

### 3.6.2. Cathedral

Cathedral-2/3 [45,46] est un outil de synthèse d'algorithmes de traitement de signal ou DSP en ASICs à partir d'une description de haut niveau [7,23]. Cathedral-2/3 se classe parmi les outils orientés flot de données. Il est dédié aux applications numériques complexes de temps réel à moyen (CATHEDRAL-2) ou haut (CATHEDRAL-3) débit [23]. Ces applications sont caractérisées par l'exécution graduelle et continue du calcul des valeurs de sorties en fonction des valeurs présentes en entrées. Dû à ce flot constant de données, le délai d'exécution est relativement fixe pour de tels circuits. L'utilisation de Cathedral-3, par exemple, a montré que les temps de synthèse varient de quelques minutes à une heure suivant la complexité du circuit [7]. La description d'entrée de Cathedral-2/3 est écrite en langage sillage sous forme d'instructions concurrentes. L'outil est contraint par un fichier de directive, qui impose de réaliser les instructions sur deux types d'opérations : les EXUs (**EX**ecution **U**nit) et les ASUs (**A**pplication **S**pecific **U**nit). Les EXUs sont issues de la bibliothèque de l'outil, alors que les ASUs sont des opérateurs spécifiques rapides créés sur demande. La description de sortie est produite en VHDL au niveau transfert de registre ou au niveau portes.

Le domaine des applications synthétisables est étendu en intégrant Cathedral-3 à l'environnement de Cathedral-2. L'architecture générée par Cathedral-2 est constituée d'un contrôleur micro-programmé commandant des unités fonctionnelles dans un chemin de données. Ce dernier, correspond à un ensemble d'unités d'exécution et d'unités d'applications spécifiques interconnectées.

Les EXUs sont des opérateurs figés dont la fonctionnalité est rigide. Chaque unité d'exécution ne peut réaliser qu'un ensemble restreint d'instructions. Elle réalise des fonctions arithmétiques (UAL, ACCU, MUL), de mémorisation (ROM, RAM) ou d'entrée-sortie.

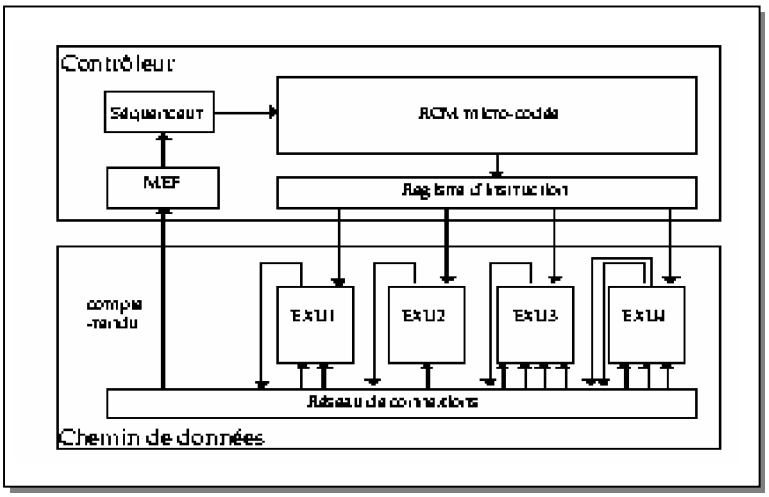



Figure 3.8 : Architecture cible de CATHEDRAL-2/3

Une autre solution offerte à l'utilisateur consiste à utiliser un deuxième type d'unités fonctionnelles, les unités d'applications spécifiques, ASUs, produites à la demande par Cathedral-3. Dans le but d'accélérer l'exécution en temps réel de certaines parties critiques d'un algorithme et/ou de réduire leur surface, le concepteur peut choisir de réaliser ces parties par des ASUs, puisqu'elles permettent d'introduire des opérateurs spécifiques plus performants que les unités d'exécution standards du système.

Les fonctions arithmétiques et le contrôle de l'ASU ne sont plus prédéfinis ni mémorisés dans une bibliothèque. Chaque ASU a sa fonctionnalité définie pour les besoins d'une application donnée. L'unité d'application spécifique est synthétisée à partir de spécifications de haut niveau, fournies par le concepteur, en langage Silage. Une ASU se compose d'un noyau arithmétique et d'un contrôleur logique local pour le décollage d'instructions et les décisions locales. Cette architecture évite la perte systématique d'un cycle, nécessaire au contrôleur global micro-programmé pour contrôler les EXUs [7]. En plus, le degré de « pipeline » des ASUs est souvent très élevé pour satisfaire les contraintes temporelles. Ainsi, il est possible de faire l'optimisation d'une unité d'application spécifique uniquement sans avoir à considérer le circuit en entier [23].

La deuxième partie de l'architecture produite par Cathedral-2/3 correspond au contrôleur. Ce dernier est constitué d'une machine d'état de Moore, d'un séquenceur, d'une ROM micro-codée et de registres d'instructions.

### 3.6.2.1. Flot de synthèse de Cathedral

Le flot de conception de Cahedral-2/3 est illustré par la figure 3.9 [7,23]. La synthèse architecturale par Cathedral fait intervenir un certain nombre de sous-tâches. En entrée, en plus de la description comportementale, le concepteur fournit les spécifications temporelles et d'autres directives pour guider les optimisations. Les optimisations peuvent même être logiques car la synthèse logique est intégrée [7].

Le flot commence dans CATHEDRAL-2 par l'organisation des mémoires. Cette tâche consiste à distribuer les structures de données (scalaires, tableaux,….) en mémoires ASICs, avec prise en considération de plusieurs paramètres tels que surface (influencée par le nombre d'unités de mémorisation), débit (influencé par la complexité des calculs d'adresse). Le flot de synthèse continue avec l'organisation des ressources matérielles et leurs affectations; les choix sont parfois avec des compromis vitesse/surface. Car plusieurs EXUs peuvent parfois

exécuter la même opération, des directives permettent de choisir la plus intéressante selon le concepteur.

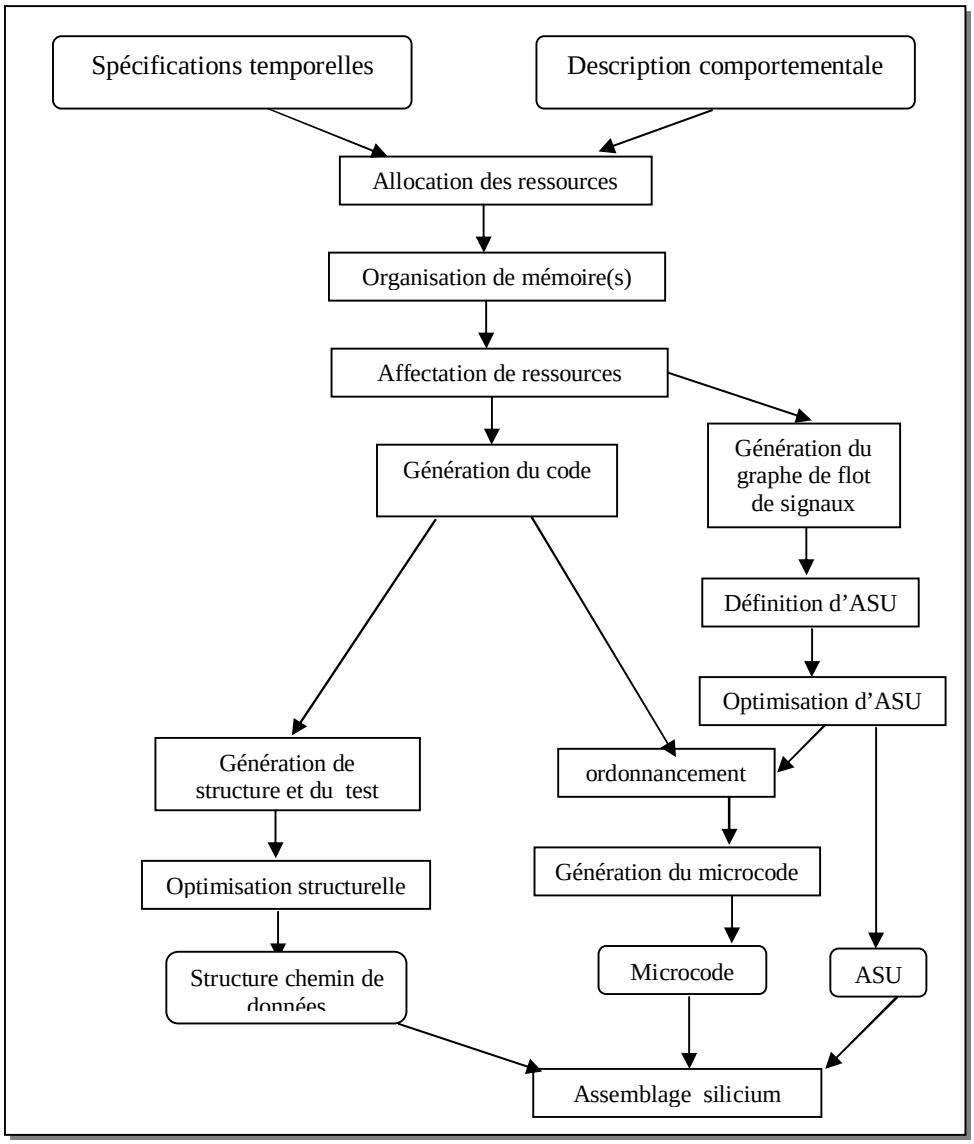



Figure 3. 9: Flot de conception de CATHEDRAL-2/3

CATHEDRAL-3 effectue la synthèse d'ASU, s'il y a une directive d'allocation d'ASU, sinon la synthèse se poursuit directement par la génération du code. La synthèse d'ASUs commence par la génération d'un graphe flot de données et de signaux. La génération de la liste de matériel nécessaire n'est possible qu'après attribution des différentes parties du graphe à des micro-instructions de l'ASU. Enfin, CATHEDRAL-3 optimise l'ASU en termes de performance et de surface.

Parmi les tâches de la synthèse, on trouve aussi, l'étape de génération de code et son optimisation. L'objectif ici est de convertire le flot de signaux en transfert entre registres.

L'étape d'ordonnancement, quant à elle, distribue les transferts de données entre registres par rapport au temps, de telle façon à minimiser le temps d'exécution de l'algorithme.

Pendant l'étape de génération et de l'optimisation de la structure, l'outil de synthèse alloue les connexions nécessaires définissant ainsi l'architecture exacte du circuit.

Enfin CATHEDRAL-2 génère le micro-code qui sera utilisé pour la génération du contrôleur.

#### 3.6.2.2. Points forts et limitations

CATEDRAL-2/3 est l'un des systèmes les plus adaptés à la synthèse de circuit DSP ou aux applications numériques complexes de temps réel. Il traite efficacement des circuit de traitement de signal grâce au pipeline automatique permis par les algorithmes d'ordonnancement et d'allocation utilisés [23]. Aussi , des estimations en performance et en surface sont disponibles après chaque étape de synthèse.

Cependant CATHEDRAL-2/3 présente aussi quelques limitations. Du fait que cet outil utilise Silage comme langage de description de circuits de traitement de signal, les possibilités pour exprimer des flots de contrôle sont très restreintes. En plus, l'utilisation de 2 langages différents en entrée et en sortie pose des problèmes supplémentaires pour la validation. Pour la simulation des deux descriptions, deux environnements différents sont requis [23].

### 3.6.3. AMICAL

Le dernier outil qu'on va présenter a été développé en milieu universitaire, au laboratoire TIMA (Technique de l'informatique et de la micro-électronique pour l'architecture d'ordinateurs) situé à Grenoble.

Le système de synthèse architecturale AMICAL [5,7,14,15,23,47], considéré aussi comme un assistant à la synthèse architecturale, s'adresse aux applications dominées par le flot de contrôle. AMICAL est un outil interactif qui produit une description VHDL au niveau RTL à partir d'une description VHDL comportementale et d'une bibliothèque d'unités fonctionnelles. Le VHDL toléré par cet outil est assez complet permettant une véritable description comportementale. Il admet plusieurs processus VHDL (« process ») dans la description initiale et tout comme SYNOPSYS Behavioral Compiler il ne traitera qu'un d'entre eux à chaque itération de synthèse [23]. AMICAL permet l'utilisation d'unités fonctionnelles complexes ainsi que la réutilisation de composants complexes déjà existants dans un nouveau design. Cet outil est compatible avec les environnements de simulation et de synthèse RTL existants. Son interactivité accroît les possibilités d'exploration de l'espace des solutions architecturale en temps réel, d'autant plus que le temps de synthèse d'AMICAL est assez court, généralement de quelques secondes à quelques minutes [7].

L'architecture cible d'AMICAL, comme illustrée par la figure 3.10 [7,8,23], est composée d'un contrôleur et d'un chemin de données. Le contrôleur est généré automatiquement durant la synthèse, il est représenté par une machine d'états finis. Ce contrôleur commande les opérations exécutées par les unités fonctionnelles et le réseau de communication. Il agit comme un contrôleur global ou central.

Le chemin de données est composé d'un ensemble d'unités fonctionnelles et d'un réseau de communication. L'architecture d'AMICAL peut inclure plusieurs unités fonctionnelles travaillant en parallèle. Ce parallélisme est déterminé automatiquement ou manuellement, durant le processus de synthèse. La complexité de ces unités fonctionnelles peut varier, elles peuvent exécuter différentes sortes d'opérations comme les opérations arithmétiques, la mémorisation, des opérations de traitement du signal, d'entrées/sorties, etc. Ce concept d'unités fonctionnelles permet d'inclure des circuits, complexes pour leur réutilisation, comme unités fonctionnelles.

Le réseau de communication établit la communication entre les unités fonctionnelles et avec le monde extérieur. Il est composé de bus, d'interrupteurs, de multiplexeurs et de registres. Permettant ainsi la génération d'une architecture à base de bus ou de multiplexeurs.

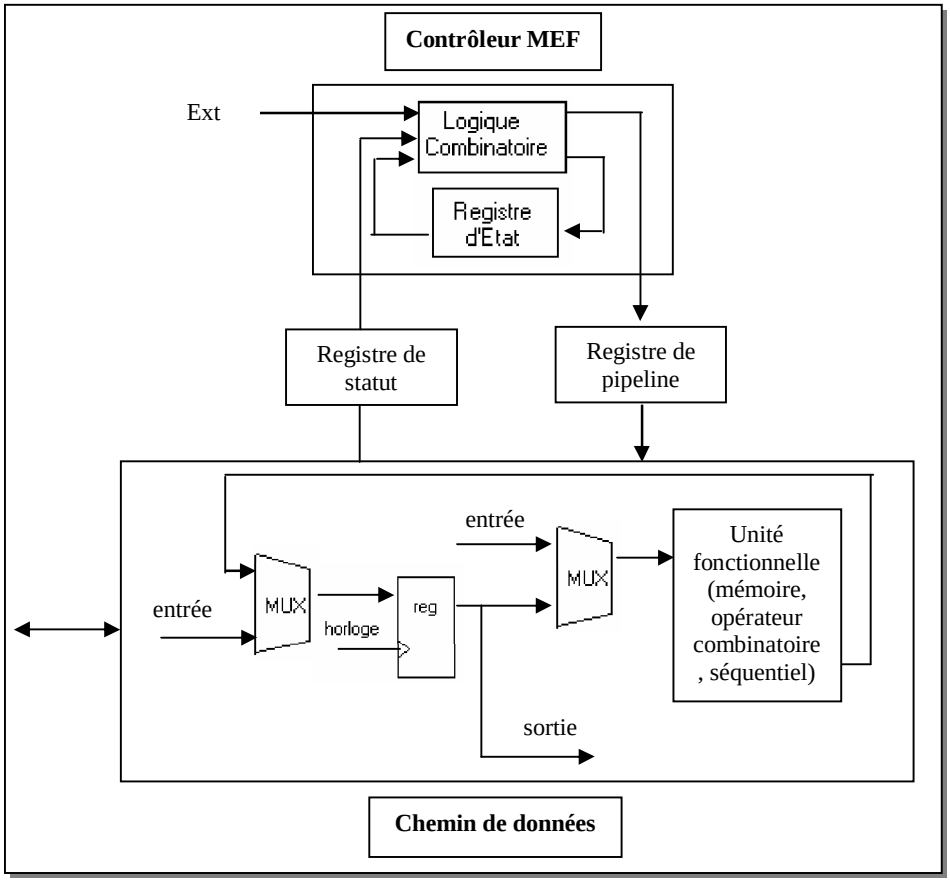



Figure 3.10: Architecture cible d'AMICAL

### 3.6.3.1. Flot de synthèse d'AMICAL

L'entrée du système AMICAL est composée d'une description comportementale en langage VHDL et d'une bibliothèque de composants, capable d'exécuter les opérations de la description comportementale. Le sous-ensemble du VHDL accepté comprend la majorité des instructions séquentielles (actions conditionnelles, affectations de signaux et de variables, instructions d'attente), permettant de décrire des algorithmes complexes [8]. Ce modèle d'entrée constitue le point de départ du flot de conception architecturale de l'outil AMICAL. La figure 3.11 [7,8,23] illustre le flot de synthèse par AMICAL.

Les différentes étapes de la synthèse sont l'ordonnancement, l'allocation (y compris l'assignation) des ressources, et la génération de l'architecture. pour avoir une conception optimale, comme il a été dit précédemment (section 3.4), l'idéal serait l'exécution simultanée de l'ordonnancement et de l'allocation. Cependant, pour dépasser la complexité de ce problème NP-complet, le système AMICAL décompose ces deux tâches en sous-tâches entrelacées. Le macro-ordonnancement, le micro-ordonnancement, l'allocation d'unités fonctionnelles et l'allocation des unités d'interconnexions respectivement.

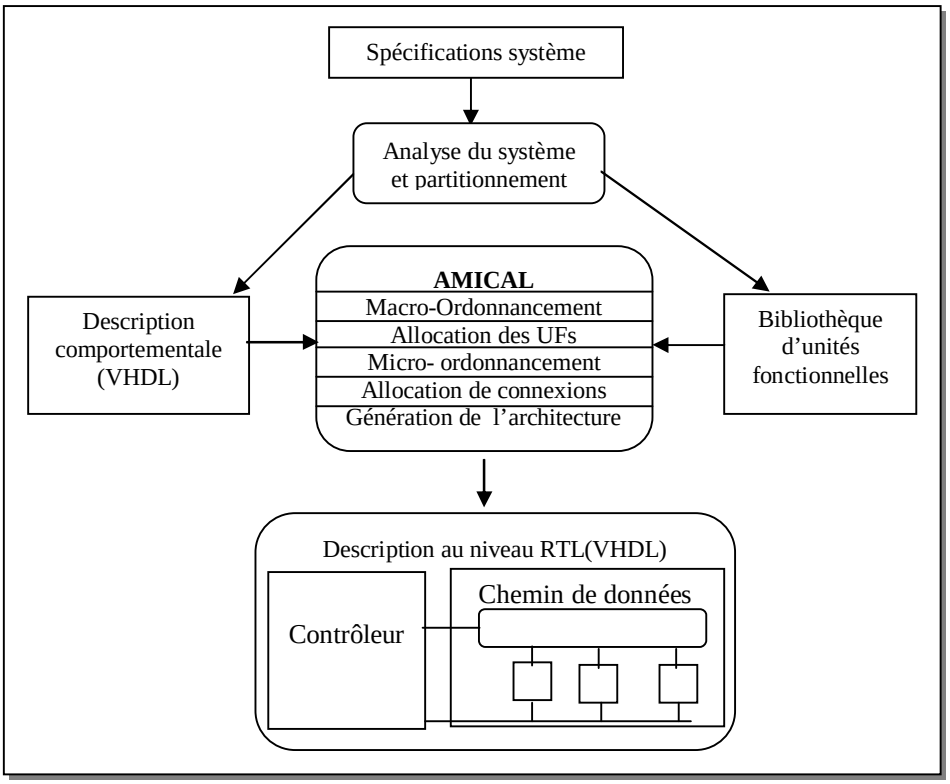



Figure 3.11: Flot de synthèse d'AMICAL

Le macro-ordonnancement réalise un ordonnancement partiel de la description comportementale et en extrait uniquement les principales étapes de contrôle (macro-cycle ou macro-étape) référencées dans la description comportementale [23]. Un macro-cycle peut utiliser plusieurs cycles de base (cycles d'horloge) pour l'exécution de ses opérations. Le macro-ordonnancement applique l'algorithme de boucle dynamique DLS (**D**ynamic **l**oop **S**cheduling), c'est un ordonnancement à base de chemins (Path-Based Algorithm). De plus, à chaque variable est associé un registre.

L'étape suivante du flot de synthèse est l'allocation des unités fonctionnelles. Elle associe une unité fonctionnelle à chaque opération dans la table de transitions. Les opérations d'une même transition ne peuvent être exécutées par une même UF. L'allocation des UFs se fait à partir d'une description ordonnancée et d'une bibliothèque externe d'unités fonctionnelles. Après cette étape, le schéma d'exécution de chaque opération est connue.

Un second ordonnancement, appelé micro-ordonnancement, est effectué. Cette étape décompose chaque opération en un ensemble de transferts entre registres. Les transferts sont ordonnancés en micro-cycles. Chaque micro-cycle contient un ensemble de transferts parallèles qui s'exécutent en un seul cycle de base.

L'étape d'allocation des connexions correspond à l'allocation des bus, interrupteurs et multiplexeurs nécessaires pour exécuter tous les transferts tout en minimisant le coût de ces connexions (nombre de bus et d'interrupteurs et/ou multiplexeurs). Cette allocation doit se faire de telle sorte que les transferts parallèles soient indépendants afin de préserver le comportement.

La dernière étape de synthèse est une génération d'architecture classique. AMICAL produit une architecture composée d'un chemin de données et d'un contrôleur.

La synthèse par AMICAL peut se faire manuellement, automatiquement, interactivement ou en combinant ces trois modes. Le mode interactif permet au concepteur d'intervenir à tout moment du processus de synthèse, en passant en mode manuel, pour modifier les décisions des algorithmes de synthèse. En mode manuel, le système agit en tant

qu'assistant ; il vérifie la cohérence des décisions prises et ne permet que les opérations correctes [15].

### 3.6.3.2. Points forts et limitations

La principale caractéristique qui différencie AMICAL des outils de synthèse d'architectures existants est la possibilité d'étendre des méthodologies permettant la réutilisation de composants au niveau comportemental [23]. Avec AMICAL, le nombre de cycles nécessaires pour l'exécution d'une unité fonctionnelle n'est pas limité, ce qui permet d'avoir une librairie plus complète avec des unités fonctionnelles très complexes. Cette approche laisse une grande flexibilité quant à l'utilisation de composants existants.

AMICAL est caractérisé par sa compatibilité et son intégration facile avec des outils et environnements de CAD existants et des environnements de simulation.

De plus, les différents modes de synthèse offerts par le système AMICAL, conception manuelle et automatisée alternées, laisse plus de liberté au concepteur pour déterminer son architecture. Contrairement aux systèmes classiques qui laissent peu de marge au concepteur dans l'orientation du résultat.

Quoique la possibilité d'introduction de modules par le concepteur, comme unités fonctionnelles pour la synthèse constitue un avantage du système AMICAL, elle peut être source de possibles erreurs de spécifications.

L'exploitation de l'approche d'interactivité nécessite une connaissance complète des algorithmes de synthèse comportementale. Bien que l'interactivité est toujours optionnelle, la multitude des représentations internes et les possibilités inutilisées peuvent constituer un embarras pour les concepteurs [8].

Une autre limitation d'AMICAL est due à son orientation à la réalisation de la synthèse d'un processus VHDL entièrement algorithmique (comportemental-séquentiel). Alors que la description d'entrée peut contenir d'autres composantes concurrentes (instructions flot de données,……).

Le tableau 3.3 récapitule les principales caractéristiques de chaque outil présenté dans ce chapitre, à savoir : le domaine d'application, le langage de description d'entrée, le langage de description de sortie, le type d'architecture produite et le schéma de synthèse.

| **Caractéristiques** | **Amical** | **Cathedral-2/3** | **Behavioral Compiler** |
|---|---|---|---|
| **Domaine d'application** | Flot de contrôle | Flot de données | Mixte flot de données/contrôle |
| **Description d'entrée** | VHDL (séquentielle) | Silage (concurrente) | VHDL/Verilog (séquentielle) |
| **Description de sortie** | VHDL (au niveau RTL) | VHDL (au niveau RTL ou logique) | VHDL/Verilog |
| **Sortie** | Partie contrôle<br>Partie opérative<br>Co-Processeur | Partie contrôle<br>Partie opérative | Partie contrôle<br>Partie opérative<br>Mémoire |
| **Schéma de synthèse** | Ordonnancement/ Allocation sur deux niveaux | Ordonnancement/ Allocation | Ordonnancement/ Allocation |

Tableau 3.3 : Récapitulatif des trois outils de synthèse

## 3.7. Conclusion

Ce chapitre est une introduction à la synthèse de haut niveau. Les étapes de la synthèse comportementale : compilation, ordonnancement, allocation/assignation et génération de l'architecture ont été introduites. Les algorithmes d'ordonnancement sont discutés de façon plus détaillée dans les chapitres suivants.

L'existence d'algorithmes variés pour les étapes de la synthèse de haut niveau, justifiée par la diversité des domaines d'application, a donnée lieu à un très grand nombre d'outils de synthèse comportementale. Cependant peu de ces outils ont connu une acceptation dans le milieu industriel. Comme exemples, trois outils de synthèse comportementale classiques, de types différents, ont été présentés. AMICAL orienté vers les applications dominées par le flot de contrôle, CATHEDRAL-2/3 dédié aux applications de traitement de signal, et Behavioral Compiler, orienté vers les deux domaines d'application.

# Chapitre 4

# Formalisation du problème d'ordonnancement

## 4.1. Introduction

L'ordonnancement (Scheduling) est l'une des étapes clés dans la synthèse de haut niveau. Gajski et Al la considèrent comme [30] " peut-être l'étape la plus importante durant la synthèse de structure ". L'ordonnancement est un problème important dans l'automatisation de la conception VLSI ; et une grande partie de la recherche HLS, a été consacrée à ce problème[4].

Dans la littérature, l'étape d'ordonnancement est peut-être l'une des plus étudiées dans le flux de la synthèse comportementale. Cela est dû au fait que c'est cette étape qui a, probablement, le plus d'influence sur la performance du circuit généré parce que le nombre de cycles de contrôle et le parallélisme des opérations y sont fixés [8,30,48].

L'ordonnancement est le partitionnement de la description comportementale, les opérations, en étapes ou cycles de contrôle [8,31]. Ces cycles de contrôle correspondent aux états de la machine d'états finis résultante qui peuvent comprendre plusieurs opérations s'exécutant en parallèle. En d'autres termes, l'ordonnancement définit une date d'exécution pour chaque opération de la description comportementale, en respectant qu'au moment d'exécution de l'opération les données nécessaires soient disponibles et qu'on soit capable d'allouer les ressources nécessaires pour exécuter toutes les opérations parallèles[30], i.e. en respectant les contraintes qui découlent des dépendances fonctionnelles et des choix architecturaux.

- **Les dépendances fonctionnelles**

  Les dépendances fonctionnelles imposent à chaque opération de ne pouvoir être exécutée que lorsque ses données d'entrées sont disponibles, c'est à dire à une date ultérieure aux dates de fin d'exécution des opérations qui produisent ces données.

- **Choix architecturaux**

  Le concepteur se heurte toujours au compromis classique, d'une vitesse maximale d'exécution pour une surface minimale de silicium. Ces deux critères évoluant en sens inverse, car il est possible d'augmenter la vitesse d'exécution en utilisant plus de ressources mais au dépens de la surface d'implémentation et vice versa.

Pour explorer l'espace de conception, une méthode généralement utilisée par la plupart des outils de synthèse consiste à borner :

- Soit la durée nécessaire à l'exécution de l'algorithme, la durée est mesurée généralement en étapes de contrôle pour des modèles architecturaux de circuits synchrones à une seule horloge;
- Soit le nombre de ressources matérielles.

Le compilateur d'architecture génère ensuite une solution architecturale qui minimise la deuxième dimension. On distingue donc deux types de problèmes d'ordonnancement :

- Ordonnancement sous contraintes de ressources, i.e. minimiser la durée d'exécution tout en fixant la quantité de ressources matérielles. Il consiste à poser le problème sous la forme suivante [23] : il existe un ensemble d'unités fonctionnelles et un ensemble d'opérations à exécuter sur ces unités fonctionnelles. Il faut donc, trouver un algorithme qui minimise le temps d'exécution en tenant compte des unités fonctionnelles disponibles.
- Ordonnancement sous contraintes de temps, i.e. minimiser la surface du circuit tout en bornant la durée d'exécution. L'équation à résoudre dans ce cas consiste à déterminer le nombre d'unités fonctionnelles nécessaire pour exécuter toutes les opérations dans le délai fixé [23].

Si on considère la description VHDL, ci-dessous, d'un additionneur avec deux signaux d'interface START et DONE [31].

```
Entity ADDER is
    Port( A_PORT, B_PORT, CARRY_IN, START : in bit ;
          SUM, CARRY_OUT, DONE : out bit ;
        ) ;

architecture BEHAVIORAL of ADDER is
begin
    process
    variable A,B :bit ;
    variable RESULT/ bit_vector (1 downto n) ;
      begin
    wait until (start =1) ;
    A := A_PORT ;
    B := B_PORT ;
    RESULT := A+B+CARRY_IN ;
    If (RESULT(1) =0) then
      CARRY_OUT <= 0 ;
    Else
       CARRY_OUT <= 1 ;
    End if ;
    DONE<= 1 ;
    End process ;
End BEHAVIORAL ;
```

On remarque que les instructions d'affectation des variables A et B peuvent être exécutées parallèlement puisqu'il n'y a aucune dépendance de données entre elles. Ceci est bien illustré sur le graphe de flots de contrôle et de données de la figure 4.1.

Le CDFG est composé de deux parties : le graphe de flot de contrôle CFG et le graphe de flot de données DFG. Chaque entrée dans le CFG a son correspondant DFG.

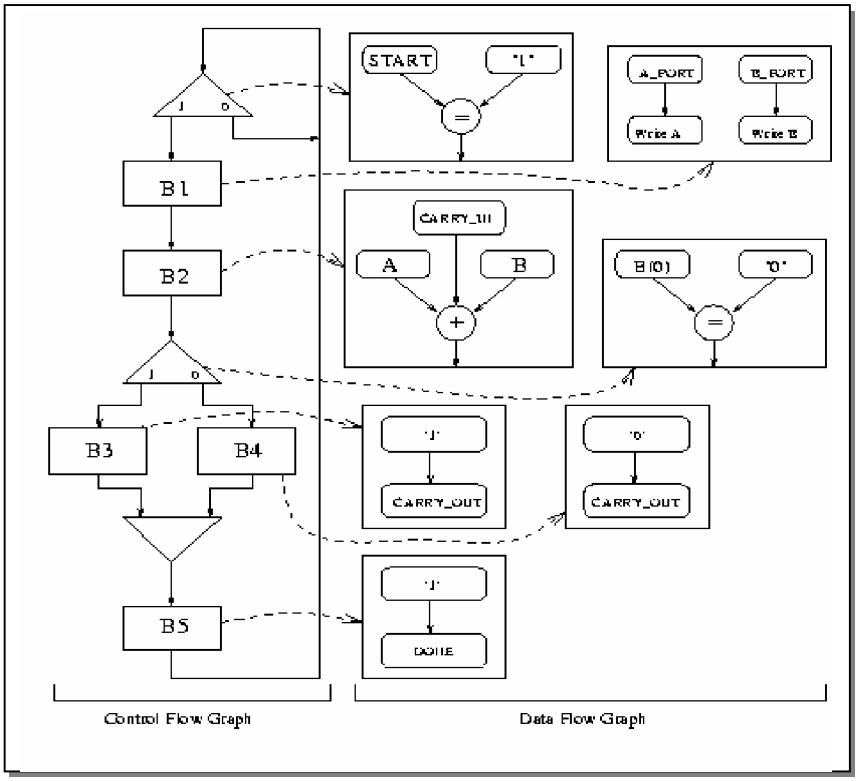



Figure 4.1 : CDFG de l'additionneur

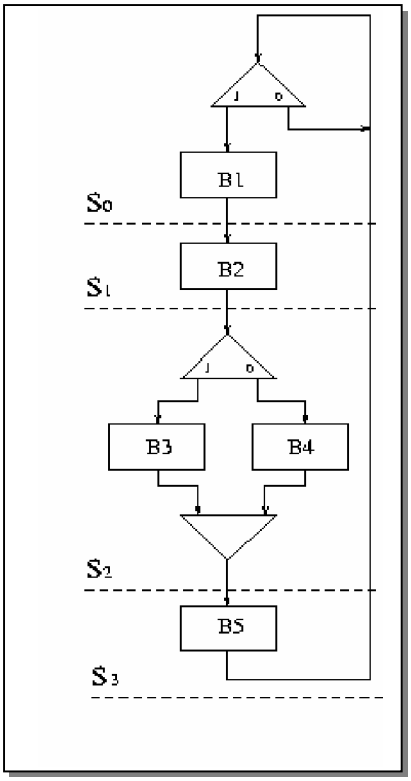



Figure 4.2 : Ordonnancement du CDFG

La figure 4.2 montre le CDFG après son ordonnancement en quatre étapes de contrôle synchrones $S_0$, $S_1$, $S_2$ et $S_3$.

## 4.2. Mode d'ordonnancement

Quel que soit le langage utilisé pour une description comportementale, une classe spéciale des instructions est définie : les instructions d'attente ou de synchronisation[8]. Pour les algorithmes d'ordonnancement, ces instructions introduisent des étapes de contrôle dans le contrôleur. En VHDL, un *wait* spécifie un état de contrôle de la spécification comportementale. Le code exécuté entre deux *wait* successifs est appelé un *pas d'exécution* (execution thread). Donc une description comportementale définit une machine d'états finis dont les états correspondent aux instructions d'attente et les transitions sont les pas d'exécution. Ainsi, la tâche principale de l'ordonnancement peut être reformulée [8]:

- Extraire des pas d'exécution de la description comportementale.
- Ordonner ces pas d'exécution sous certaines contraintes imposées par le concepteur en les décomposant dans les étapes de contrôle.

Suivant les résultats de l'ordonnancement appliqués aux pas d'exécution, notamment le nombre de cycles nécessaires à leur exécution, on distingue trois catégories d'ordonnancement [8]:

**Mode en un cycle (Cycle Mode) :**

Chaque pas d'exécution est ordonné en un seul cycle de contrôle, les opérations doivent donc être exécutées en un seul cycle d'horloge. Dans ce cas, le modèle initial est très proche de la description générée au niveau transfert de registres. Cependant, pour que la spécification comportementale puisse être synthétisable, des restrictions doivent être imposées sur le style d'écriture des pas d'exécution : interdiction des opérations multi-cycle, un pas d'exécution ne peut contenir de boucles infinies, la période de l'horloge doit être assez longue pour permettre l'exécution du pas d'exécution le plus complexe [8].

L'avantage de ce mode [8,23] est que les résultats de la simulation comportementale correspondent au cycle prés à ceux de la simulation de la description correspondante au niveau RTL, donc un même fichier de test peut être utilisé pour des résultats identiques ce qui facilite la vérification.

**Mode des macro-états (Super-state Mode) :**

Contrairement au mode précédent, dans ce mode, chaque pas d'exécution est décomposé en plusieurs cycles de contrôle dont le nombre est fixé durant le processus de synthèse [8]. La seule restriction sur le style d'écriture, dans ce mode, est l'interdiction des boucles infinies ou dépendant des données résultant de l'exécution de la même boucle.

Ce mode permet une exploration architecturale plus importante avec des opérateurs à exécution variable (avec des caractéristiques temporelles différentes). Cependant, le fait que la description générée au niveau RTL diffère de la spécification initiale rend le processus de vérification plus complexe.

**Mode des états comportementaux :**

Ce mode est le mode d'ordonnancement le plus général. Chaque pas d'exécution est réalisé par une machine d'états finis dont les transitions peuvent être exécutées en un seul cycle d'horloge et aucune restriction n'est faite sur le style d'écriture du modèle initial ce qui permet une exploration architecturale encore plus étendue. Par contre, le processus de vérification, comme pour le mode des macro-états, pose des problèmes à cause de la difficulté d'établissement de la correspondance entre les descriptions initiales et celles générées.

| | **Mode d'ordonnancement des pas d'exécution** | | |
|---|---|---|---|
| **Mode pour les entrées/sorties** | Mode en un cycle | Mode des macros-états | Mode des états comportementaux |
| E/S fixes | 1. en un cycle E/S fixes | 2. Macros E/S fixes | 3.Compotementaux E/S fixes |
| E/S flottantes | 3. en un cycle E/S flottantes | 4. Macros E/S flottantes | 6.Comportementaux E/S flottantes |

Tableau 4.1 : Modes d'ordonnancement

On distingue aussi, des stratégies pour les ordonnancements des opérations manipulant des entrées/sorties. Ceci est la conséquence de la décomposition des pas d'exécution en plusieurs cycles d'horloge qui implique généralement des changements de comportement des

entrées et des sorties du circuit. Deux modes ou stratégies pour le traitement des E/S sont connues [8] :

**Entrées/Sorties fixes :** ce mode préserve l'ordre des opérations des entrées/sorties entre elles et vis-à-vis des instructions de synchronisation [8,23]. Dans le mode des macros-états ou des états comportementaux pour un pas d'exécution donné, les lectures sont ordonnées dans le premier cycle et les écritures dans le dernier (par des restrictions sur le style d'écriture ou en utilisant des registres pour faire des mémorisations sur les entrées et les sorties).

**Entrées/Sorties flottantes :** dans ce mode, aucune contrainte n'est maintenue au niveau de l'ordre des entrées et sorties du circuit, elles sont ordonnées de façon à ce que le résultat de la synthèse satisfasse les contraintes de dépendances de données tout en minimisant le temps d'exécution, permettant ainsi une exploration architecturale plus vaste. Cependant, le comportement du circuit généré devient souvent extrêmement éloigné du comportement initial [8].

Le tableau 4.1 [8] résume les différents modes d'ordonnancement ainsi que leur combinaison avec les stratégies de traitement des entrées/sorties.

## 4.3. Différents types d'ordonnancement

L'ordonnancement a pour objet d'optimiser le nombre d'étapes de contrôle nécessaires à l'exécution d'une fonction, dans la limite des ressources matérielles disponibles et du temps de cycle[7]. L'algorithme d'ordonnancement doit tenir compte des constructions de contrôle telles que les boucles et les branchements conditionnels, les dépendances de données, et les contraintes sur les ressources matérielles. D'autres contraintes peuvent spécifier le temps de cycle, la consommation, etc.

Plusieurs algorithmes sont orientés vers les circuits de traitement de signal, dont la partie contrôle est relativement simple, donc leur but est de minimiser le coût de la partie opérative. La description d'entrée de ces algorithmes est un graphe de flot de données qui ne contient, le plus souvent, ni boucle, ni appel de procédure. Cependant dans les circuits de commande actuels, les performances de la partie contrôle dominent la performance globale du circuit, et l'organisation de la partie contrôle et la surface qu'elle occupe sont des facteurs importants. L'objectif des algorithmes d'ordonnancement pour de tels circuits est de minimiser la surface et le chemin critique du contrôleur, tout en s'assurant que le chemin de données respecte les contraintes prédéfinies.

De façon générale les algorithmes d'ordonnancement dans le domaine de synthèse comportementale peuvent être classés dans deux catégories majeures : les algorithmes d'ordonnancement orientés flot de données et les algorithmes d'ordonnancement orientés flot de contrôle. Un algorithme orienté flot de données utilise un graphe de flot de données DFG, alors qu'un algorithme orienté flot de contrôle utilise un graphe de flot de contrôle CFG comme représentation sous-jacente. Donc, les premiers consistent à sérialiser une description parallèle alors que les seconds consistent à paralléliser une description séquentielle. Dans la littérature, le nombre d'algorithmes orientés flot de contrôle est inférieur à celui des algorithmes orientés flot de données[8]. La figure 4.3 illustre les différences entre les deux approches d'ordonnancement.

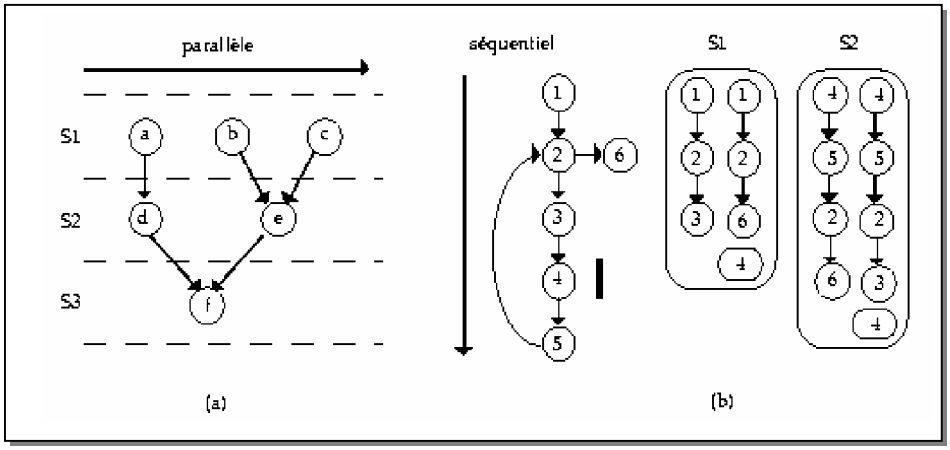



Figure 4.3 : Ordonnancement (a) flot de données (b) flot de contrôle

## 4.3.1. Algorithmes orientés flot de données

L'ordonnancement d'un graphe de flot de données consiste à affecter chaque opération à une étape de contrôle tout en respectant les contraintes. Ces contraintes fixent soit le nombre d'étapes de contrôle, soit le nombre de ressources à utiliser. Donc, le problème d'ordonnancement est un problème d'optimisation à deux dimensions, suivant les contraintes : le temps et la surface ; il définit, d'une part, la manière d'attribution des opérations aux étapes de contrôle et, d'autre part, le nombre d'étapes de contrôle nécessaires pour exécuter en sa totalité le graphe de flot de données et le nombre d'unités fonctionnelles qui sont impliquées de façon optimale.

Suivant les contraintes, les algorithmes orientés flot de données peuvent être classifiés en quatre catégories :

#### 4.3.1.1. Ordonnancement sans contraintes US (**U**nconstrained **S**cheduling**)**

Les méthodes d'ordonnancement sans contraintes reposent sur les contraintes de précédence pour attribuer une date à chaque opération. Ces algorithmes ont l'avantage d'être trés rapides, et sont généralement utilisés par les algorithmes d'ordonnancement sous contraintes pour déterminer les intervalles d'exécution ou la mobilité des opérations.

Soit $G_s(V,E)$ un graphe orienté acyclique représentant la dépendance de données, où :

$V=\{v_i ; i= 0,1,\ldots,n\}$ est l 'ensemble des opérations élémentaires (+,-,*,…)

Et $E=\{(v_i,v_j) ; i,j =0,1,\ldots,n\}$est l'ensemble des arcs de dépendance de données ; un arc $e_{ij}$ lie l'opération $v_i$ à l'opération $v_j$, si une donnée définie par $v_i$ est un opérande de $v_j$ .

Le nœud $v_0$ est une opération fictive reliée à chaque opération sans prédécesseur, chaque opération sans successeur est reliée à l'opération fictive $v_n$ avec $n= n_{ops} +1$, où $n_{ops}$ est le nombre d'opérations de la description comportementale.

Soit $D= \{d_i ; i= 0,1,\ldots,n \}$ l'ensemble des durées associés aux opérations ($d_0= d_n= 0$). La durée d 'exécution de chaque opération est connue, elle est donnée en nombre entier d'étapes de contrôle .

Soit $T= \{ti ; i= 0,1,\ldots,n\}$ l'ensemble des dates de début d'exécution des différentes opérations, i.e. l'étape où débute chaque opération ($t_0=1$). La durée totale d'exécution $T_{tot}$ est la durée nécessaire à l'exécution de l'ensemble des opérations : $T_{tot}= t_n-t_0$.

Les contraintes de précédence imposent que la date de début d'exécution d'une opération vi soit au moins supérieure ou égale au début d'exécution plus la durée d'exécution de chacun de ces prédécesseurs.

$$V_j : \quad t_i \geq t_j + d_j \qquad \forall\ i,j : (v_i,v_j) \in E \qquad (1)$$

Un ordonnancement sans contraintes définit un ensemble de début d'exécution T vérifiant la relation (1), c'est à dire satisfaisant la relation de précédence. Un ordonnancement sans contraintes de durée minimale est un ordonnancement sans contraintes avec une durée totale $T_{tot}$ minimale.

**Définition 4.1 :**
Etant donné un ensemble d'opérations V de durée associée D et un ensemble E, un ordonnancement sans contraintes de durée minimale est une application

$$\varphi : V \rightarrow N / t_i = \varphi(v_i),\ t_i \geq t_j + d_j \qquad \forall\ i, j : (v_i,v_j) \in E$$

et tel que la valeur de $t_n$ soit minimale.

L'ordonnacement au plus tot ASAP (**A**s **S**oon **A**s **P**ossible) et l'ordonnancement au plus tard ALAP (**A**s **L**ate **A**s **P**ossible) sont des algorithmes d'ordonnancement sans contraintes.

### 4.3.1.2. Ordonnancement sous contraintes de ressources RCS ( **R**essource **C**onstrained **S**cheduling**)**

En pratique, il est parfois nécessaire de limiter la surface en silicium du circuit autant pour des raisons matérielles (prix élevé) que pour des raisons d'efficacité (la probabilité de bon fonctionnement d'un circuit est d'autant plus petite que sa taille est grande) [23]. Les contraintes de ressources imposent généralement des restrictions sur le nombre d'unités fonctionnelles pour limiter la surface du circuit généré.

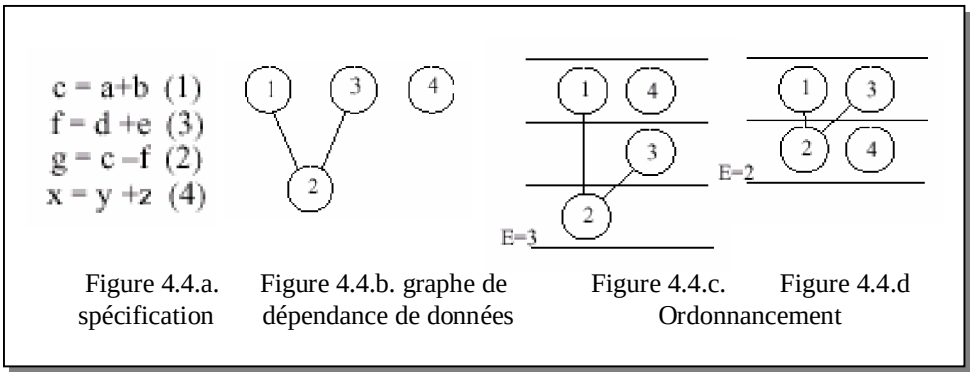



Figure 4.4.a. spécification — Figure 4.4.b. graphe de dépendance de données — Figure 4.4.c. Figure 4.4.d Ordonnancement

Figure 4.4 : Ordonnancement sous contraintes matérielles et minimisation du nombre d'étapes de contrôle

L'ordonnancement sous contraintes de ressources a pour but de minimiser le nombre d'étapes de contrôle nécessaires pour l'exécution d'une séquence d'opérations sur un nombre limité de composants [23]. Les approches les plus connues pour résoudre ce problème sont le "list-based scheduling " [49] et le "static-list scheduling".

La figure 4.4.c, ainsi que la figure 4.4.d, montrent un ordonnancement sous contraintes de ressources appliqué au graphe de dépendance de données de la figure 4.4. b.

Le nombre d'opérateurs est limité à deux opérateurs, ce qui limite le parallélisme d'action. Si l'on ordonnance l'opération 4 dans l'étape de contrôle 1, l'ordonnancement est réalisé au mieux en 3 étapes de contrôle (figure 4.4.c), alors que si l'on ordonnance l'opération 4 à l'étape de contrôle 2, l'ordonnancement peut être réalisé en 2 étapes de contrôle (figure 4.4.d).

Soit l'application $\tau : V \rightarrow \{1,2,\ldots\ldots,n_{res}\}$, elle associe à chaque opération un type unique d'opérateur. Avec $n_{res}$ le nombre de types de ressources différentes. On suppose que la fonction $\tau$ est connue. Le nombre d'opérateurs de chaque type est donné par l'ensemble $A=\{a_k ; k=1,2,\ldots\ldots n_{res}\}$.

Un ordonnancement sous contraintes de ressources définit un ensemble de début d'exécution T vérifiant la relation 1, tel que pour chaque étape de contrôle et pour chaque type d'opérateur le nombre d'opérations à exécuter reste inférieur au nombre d'opérateurs disponibles. Le problème d'ordonnancement sous contraintes de ressources de durée minimale peut être défini comme suit :

**Définition 4.2 :**
Etant donné un ensemble d'opérations V de durée associée D, un ensemble E et un ensemble A, un ordonnancement sous contraintes matérielles de durée minimale est une application

$$\varphi : V \rightarrow N / t_i = \varphi(v_i),\ t_i \geq t_j + d_j \qquad \forall\ i,j : (v_i,v_j) \in E ,$$
$$|\{v_i : \tau(v_i)=k \text{ et } t_i \leq l < t_i+d_i\}| \leq a_k, \quad k=\{1,2,\ldots\ldots.. , n_{res}\},$$
avec $l= 1,2,\ldots\ldots..,t_n$, et tel que la valeur de $t_n$ soit minimale

### 4.3.1.3. Ordonnancement sous contraintes de temps TCS (**T**ime **C**onstrained **S**cheduling)

La contrainte de temps consiste à limiter le temps d'exécution global du graphe de flux de données, autrement dit le nombre d'étapes de contrôle. L'étape d'ordonnancement consiste alors à décomposer la description comportementale en étapes de contrôle de telle manière que la durée d'exécution reste sous les limites temporelles imposées, l'objectif étant de minimiser le nombre de ressources matérielles. Un certain nombre d'algorithmes et d'heuristiques ont été publiés, le plus répandu est l'ordonnancement orienté par les forces [36].

Reprenons le graphe de flot de la figure 4.4.b. Si l'on borne la durée d'exécution à 2 étapes de contrôle, l'opération 4 peut être ordonnancée à l'étape de contrôle 1 ou 2. Dans le premier cas, on doit disposer de 3 opérateurs pour pouvoir exécuter 3 opérations en parallèle (figure 4.5.a). Dans le deuxième cas, seulement deux opérateurs sont nécessaires (figure 4.5.b).

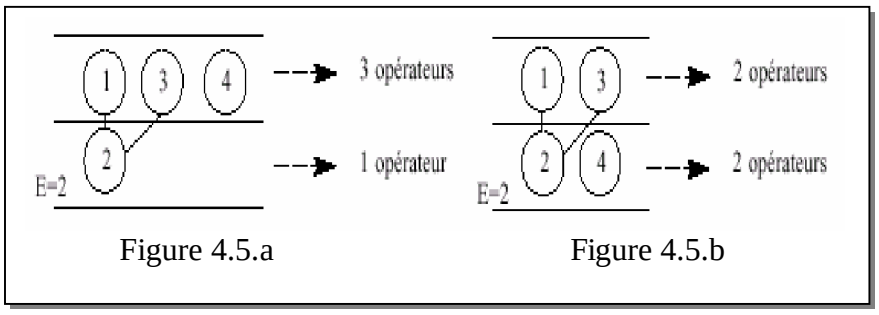



Figure 4.5.a Figure 4.5.b

Figure 4.5 : Ordonnancement sous contraintes temporelles et minimisation du nombre d'opérateurs

Le problème d'ordonnancement sous contraintes temporelles est le problème dual du problème d'ordonnancement sous contraintes matérielles : pour une date de fin d'exécution donnée (temps d'exécution), on cherche une répartition des opérations sur les étapes de contrôle tel que la surface de silicium nécessaire soit minimale (approximation de la surface par le nombre d'opérateurs de chaque type). Si $C_k$ désigne la surface d'un opérateur de type k, le problème d'ordonnancement sous contraintes de temps peut être défini de la façon suivante :

**Définition 4.3 :**
Etant donné un ensemble d'opérations V de durée associée D, un ensemble E, une durée d'exécution maximale T et des coûts $C_k$ ; un ordonnancement sous contraintes temporelles est une application :

$$\varphi : V \rightarrow \{1,2,\ldots\ldots,T\} / \; t_i = \varphi(v_i),\; t_i \geq t_j + d_j \qquad \forall\; i,j : (v_i,v_j) \in E$$
$$\text{si } a_k = \max\left(\text{Card}(\{v_i / \tau(vi) = k \text{ et } t_i \leq t \leq t_i + d_i - 1\})\right), \quad \forall\; k = \{1,\ldots\ldots,n_{type}\},$$
$$\text{et tel que } \sum_{k=1}^{ntype} a_k * c_k \quad \text{soit minimale}$$

#### 4.3.1.4. Ordonnancement sous contraintes de temps et de ressources TRCS (**T**ime-**R**essource **C**onstrained **S**cheduling**)**

Cette approche peut être considérée comme la combinaison des deux précédentes, elle consiste à trouver une solution optimale tout en satisfaisant des contraintes matérielles et temporelles à la fois. Les méthodes de programmation en nombres entiers ILP (**I**nteger **L**inear **P**rogramming)[50] permettent de résoudre ce type d'ordonnancement. L'ordonnancement sous contraintes de temps et de ressources peut être défini de la façon suivante :

**Définition 4.4 :**
Soit G = (V,E) un graphe de flot de données ; un ensemble K d'unités fonctionnelles ; une fonction type $\varphi : V \rightarrow K$; un ordre partiel sur V (dépendance de données); une contrainte sur les ressources $m_k$ pour chaque unité fonctionnelle de type k; et une durée D.

*Objectif*: trouver un ordonnancement possible sur G=(V,E) satisfaisant d'une part la relation de précédence et d'autre part les contraintes sur les ressources pour chaque unité fonctionnelle de type k et dont le nombre d'étapes de contrôle ne dépasse pas D.

### 4.3.2. Algorithmes orientés flot de contrôle

Contrairement à l'ordonnancement orienté flot de données, l'ordonnancement orienté flot de contrôle repose sur un graphe de flot de contrôle comme représentation intermédiaire. Comme un CFG est une représentation séquentielle de la description comportementale, l'algorithme d'ordonnancement orienté flot de contrôle a pour rôle d'extraire le maximum de parallélisme tout en respectant les contraintes de données. Cette approche consiste à affecter une séquence d'opérations à une étape de contrôle, et une opération peut être affecter à plusieurs étapes de contrôle.

Toutes les approches orientées flot de contrôle sont basées sur la recherche des chemins d'exécution composés de séquences d'opérations [8]. Les algorithmes considèrent tous les chemins possibles du CFG et les ordonnent séparément afin de minimiser le nombre

d'étapes de contrôle pour chaque chemin. Le résultat est un ensemble de chemins ordonnés dont les opérations peuvent être exécutées en un seul cycle de contrôle. Puis ces chemins sont regroupés afin de générer la machine d'états finis équivalente au comportement défini par le CFG.

Parmi les algorithmes d'ordonnancement orientés flot de contrôle, on trouve l'ordonnancement au plus tôt AFAP (**A**s **F**ast **A**s **P**ossible) [41] et l'ordonnancement à boucles dynamiques DLS (**D**ynamic **L**oop **S**cheduling) [51]. Par la suite seront définis formellement les concepts de base : les contraintes, les chemins ordonnés et les têtes d'un graphe de flot de contrôle.

**Définition 4.5 : contrainte**

Soit G=(V,E) un graphe de flot de contrôle, $(C_1,\ldots,C_r)$ un ensemble de contraintes. $\varphi C_k(k=1,\ldots,r)$ un ensemble de fonctions qui à un couple de sommets associe une valeur booléenne. Une contrainte $C_k$ est dite satisfaite entre deux sommets $v_i$, $v_j$ si $\varphi C_k(k=1,\ldots,r) = 1$. Les contraintes considérées sont de deux types :

- **Contraintes intrinsèques :** elles proviennent de la nature même des systèmes synchrones, où les contraintes de base portent sur l'utilisation unique d'une même ressource matérielle au cours d'une étape de contrôle. Autrement dit, au cours d'une étape de contrôle, un registre ne peut être chargé qu'une seule fois, un port d'entrée/sortie ne peut être lu ou écrit qu'une seule fois et un bus ne peut porter qu'une seule valeur.
- **Contraintes externes :** ce sont des contraintes concernant les ressources disponibles. C'est l'utilisateur qui spécifie le nombre et le type de chaque ressource.

**Définition 4.6 : chemin ordonné**

Soit G=(V,E) un graphe de flot de contrôle, soit $(c_1,\ldots,c_r)$ un ensemble de contraintes. Un chemin ordonné SP est une séquence de sommets $(v_1,\ldots,v_n)$ extraits du GFC et qui satisfait toutes les contraintes, c'est à dire pour tout couple $(v_i,v_j) \in$ SP, pour toute contrainte $C_k(k=1,\ldots,r)$, $\varphi C_k(k=1,\ldots,r) = 1$.

Chaque chemin ordonné possède trois paramètres :

- Une tête, qui représente le premier sommet du chemin noté tête(SP)
- Un successeur, qui représente le successeur immédiat du dernier sommet de SP, noté Succ(SP).
- Une expression booléenne Condp (SP).

Condp (SP) est l'expression permettant l'exécution du chemin ordonné SP, elle est dérivée par un ET logique de toutes les expressions associées aux arcs composant le chemin ordonné SP partant de Tête(SP) à Succ(SP). Cette expression est donnée par la formule suivante :

$$Condp(SP) = \wedge_{(i=1,\ldots,n-1),(j=i+1)} Cond(vi,vj) \wedge Cond(v_n,Suc(SP))$$

**Définition 4.7: Têtes d'un graphe de flot de contrôle**

A chaque graphe de flot de contrôle, on définit un ensemble de têtes qui représente l'ensemble des têtes de tous les chemins ordonnés.

$$H(G)= \{v_i \in V/ \exists\ SP \subset V, v_i = Tête(SP)\ \}$$

**Définition 4.8 : Ordonnancement à base de flot de contrôle**
Un ordonnancement d'un graphe de flot de contrôle G=(V,E) est une fonction :

$$\sigma: H(G) \rightarrow S$$
$$v_j \rightarrow s_j$$
$$F(s_j , Cond_p(SP)) = \sigma ( v_j ) \text{ tel que } v_j = Succ(SP) \text{ et } \sigma (\text{tête } (SP)) = s_j$$

S est l'ensemble des étapes de contrôle et F est la fonction de transition. L'ordonnancement consiste à fusionner tous les chemins ordonnés ayant la même tête dans le même état et une transition est faite entre un état $S_i$ représenté par $v_i$ et un état $S_j$ représenté par $v_j$ s'il existe un chemin ordonné SP ayant comme tête (SP) = $v_i$ et comme successeur Succ(SP)= $v_j$.

## 4.4. Conclusion

L'ordonnancement est une tâche très importante dans le processus de synthèse de haut niveau, il définit une date d'exécution pour chaque opération de la description comportementale. Dans ce chapitre, les dépendances de données et les choix architecturaux ont été discutés, vu le rôle qu'ils jouent dans l'étape d'ordonnancement. Un survol des différents modes d'ordonnancement a été effectué.

Ce chapitre a présenté aussi deux grandes classes d'ordonnancement dans la synthèse de haut niveau avec leurs définitions formelles à savoir, les approches d'ordonnancement orientées flot de données et celles orientées flot de contrôle. Ces deux approches diffèrent par la représentation intermédiaire de la description comportementale.

On a illustré par des exemples l'influence des résultats de l'ordonnancement sur la surface d'un circuit, ainsi que l'importance du choix des opérations à exécuter à chaque étape de contrôle.

Etant donné que le sujet principal de ce mémoire est l'ordonnancement, le chapitre suivant présentera, de façon un peu détaillée, quelques uns des plus connus algorithmes d'ordonnancement.

# Chapitre 5

# Algorithmes d'ordonnancement

## 5.1. Introduction

La plus simple méthode pour effectuer l'ordonnancement est de céder la tâche à l'utilisateur [22], comme dans le système Silc. Toutefois, il y a une tendance vers l'automatisation de l'ordonnancement. Au cours des années, les chercheurs ont essayé d'appliquer différents types de solutions au problème d'ordonnancement. Plusieurs algorithmes sont disponibles, chacun d'eux a ses avantages et ses inconvénients. Les algorithmes d'ordonnancement sont classifiés, comme illustré dans la section 4.3, en deux catégories : les algorithmes d'ordonnancement orientés flot de données et ceux orientés flot de contrôle.

## 5.2. Classification des algorithmes d'ordonnancement

Au cours du temps, on a réalisé que les techniques d'ordonnancement développées aux premiers temps de la synthèse de haut niveau n'étaient pas en position à prendre en considération les besoins des circuits dominés par le contrôle, rencontrés dans des domaines tels que les télécommunications [4]. Typiquement, les circuits dominés par le contrôle ont une structure de flot de contrôle complexe contenant des boucles et des instructions conditionnelles. Pour une conception de haute qualité, toutes les structures de contrôle doivent être prises en considération, non pas seulement les lignes de code correspondant aux blocs de base. Alors que les algorithmes d 'ordonnancement développés pour les circuits dominés par le flot de données concentrent principalement sur les calculs à l'intérieur d'un bloc de base.

Actuellement, un grand nombre d'approches et d'algorithmes sont développés afin d'essayer de satisfaire les besoins du marché. Ces approches peuvent être des approches constructives/itératives ou des approches transformationnelles.

Les approches constructives utilisent les heuristiques glouton et essayent de minimiser le coût mais ne garantissent pas nécessairement une solution optimale. Comme exemples on peut citer des heuristiques simples et rapides telles que l'ordonnancement à base de listes [52,64] ou des heuristiques plus complexes telles que l'ordonnancement orienté par les forces [36]. Le point faible ou l'inconvénient des heuristiques glouton est la possibilité d'être piégés par un minimum local de la fonction de coût, ainsi elles ne donnent pas toujours la solution optimale.

Les approches itératives probabilistes, telles que le recuit simulé, utilisent un petit ensemble de transformations qui sont appliquées à des parties sélectionnées aléatoirement dans le graphe ordonnancé. Une fonction d'acceptation (approbation) probabiliste permet d'échapper aux minimums locaux. Etant donné que ces approches nécessitent une bonne exploration de l'espace d'ordonnancement, elles manifestent typiquement des temps d'exécution larges.

Les approches exactes utilisent des techniques telles que la programmation en nombres entiers ILP (**I**nteger **L**inear **P**rogramming) [50,53] et garantissent l'optimum global, mais elles peuvent avoir des temps d'exécution larges. De telles approches représentent les décisions d'ordonnancement comme un ensemble de variables de décision ( une pour chaque assignation possible d'un opérateur à une étape de contrôle ) et un ensemble de contraintes qui doivent être satisfaites. L'ILP peut être ensuite utilisé pour trouver l'ordonnancement optimal avec respect d'une certaine fonction de coût. Cependant, le temps d'exécution dans ces approches dépend du nombre de variables et de contraintes. Cette valeur peut être extrêmement grande pour des problèmes larges.

## 5.3. Algorithmes d'ordonnancement orientés flot de données

Les algorithmes d'ordonnancement peuvent être classifiés, principalement, en des algorithmes d'ordonnancement sous contraintes de temps et des algorithmes d'ordonnancement sous contraintes de ressources. Pour le premier type d'ordonnancement, le nombre de ressources est minimisé pour un nombre fixe d'étapes de contrôle. Alors que pour le deuxième, le nombre d'étapes de contrôle est minimisé pour un coût de conception donné (nombre de ressources).

L'ordonnancement sous contraintes de temps est important pour les conceptions ciblant les applications dans les systèmes temps-réel tels que les systèmes de traitement de signal. L'ordonnancement sous contraintes de ressources est important pour les applications où l'on impose des limites à la surface des circuits. L'objectif dans ce cas est de produire un circuit avec la plus haute performance possible tout en satisfaisant les contraintes matérielles imposées[31].

On distingue aussi les algorithmes d'ordonnancement de base : ASAP (**A**s **S**oon **A**s **P**ossible) et ALAP (**A**s **L**ate **A**s **P**ossible). Ces méthodes ne considèrent que les contraintes de précédence pour attribuer une date d'exécution à chaque opération. Elles ont l'avantage d'êtres très rapides (complexité de l'ordre du nombre d'arcs).

La plupart des algorithmes d'ordonnancement nécessitent de connaître les bornes au plus tôt et au plus tard pour pouvoir ordonnancer les opérations du graphe de flot. Pour déterminer ces bornes, ces algorithmes d'ordonnancement utilisent les approches ASAP et ALAP.

## 5.3.1. Algorithme ASAP

Le principe à suivre dans cet algorithme est simple ; une opération ne peut être exécutée que si tous ses prédécesseurs, issus du graphe de flot de données, sont exécutés.

L'algorithme ASAP **(A**s **S**oon **A**s **P**ossible**)** commence avec les nœuds les plus hauts, c'est à dire les opérations sans prédécesseurs, qui sont ordonnancées dans une première étape de contrôle. Tant qu'il reste des opérations à ordonnancer, l'algorithme ASAP passe à l'étape de contrôle suivante en lui assignant toutes les opérations qui ont leurs prédécesseurs ordonnancés. L'algorithme ASAP est résumé dans la figure 5.1 [30]. Les dates de début d'exécution obtenues par l'algorithme ASAP sont présentées par le vecteur $t^s = \{ t_i^s ; i=0,1,\ldots,n\}$.

```
Procedure      ASAP(Gs(V,E)){
 Ordonnancer v0 :t0 =1 ;                    /* v0 opération fictive*/
Faire {
        Sélectionner un noeud vi tel que tous ses prédécesseurs sont ordonnancés ;
        Ordonnancer vi :  t_i^s = max_{j:(vj,vi)} t_j^s + dj ;  / (vj,vi) ∈ E
        /* la première étape aprés tous les prédécesseurs de Vi*/
        }
 jusqu'à ce que vn soit ordonnacée ;        /* vn opération fictive*/
}
```

Figure 5.1 : Algorithme ASAP.

L'algorithme ASAP est utilisé dans le système Emerald/Facet de l'université de Carnegie Mellon, et le système CATREE de l'université de Waterloo [16]. La figure 5.2.b illustre l'ordonnancement ASAP du graphe de flot de données de la figure 5.2.a.

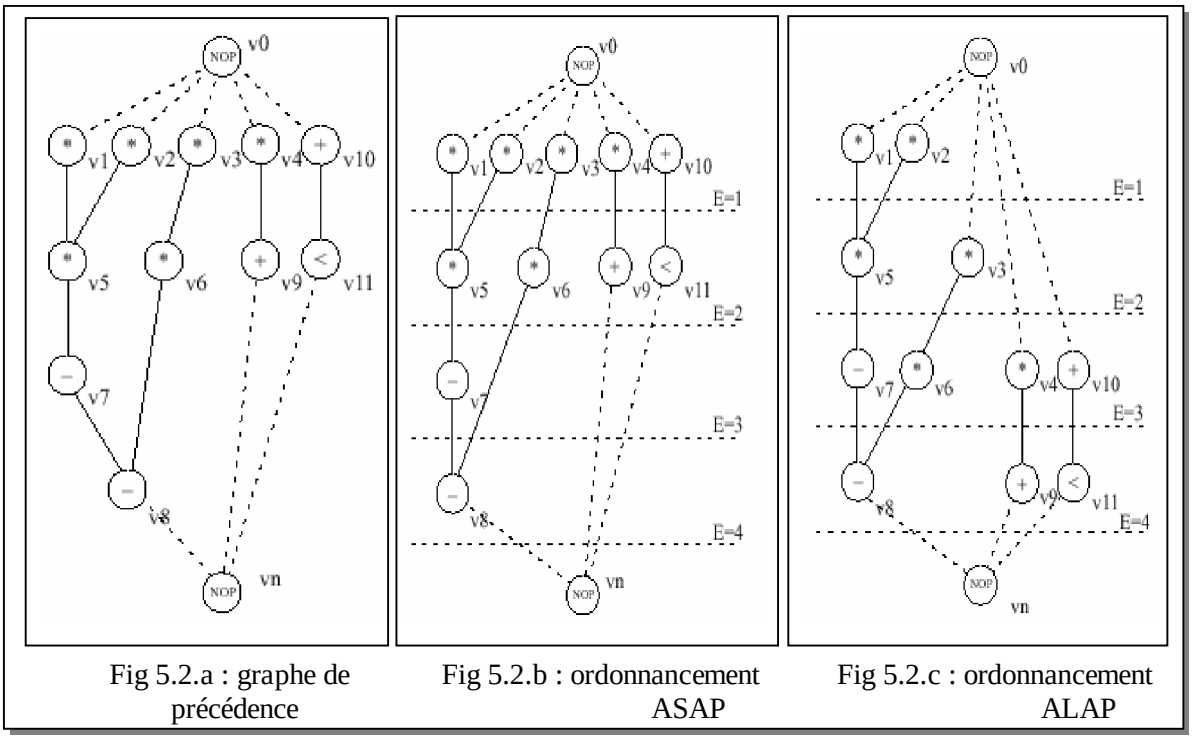



Fig 5.2.a : graphe de précédence

Fig 5.2.b : ordonnancement ASAP

Fig 5.2.c : ordonnancement ALAP

Figure 5.2 : Ordonnancements ASAP et ALAP

## 5.3.2. Algorithme ALAP

L'approche ALAP (**A**s **L**ate **A**s **P**ossible) est un raffinement du concept de l'algorithme ASAP avec remise au plus tard des opérations [16]. L'algorithme ALAP procède exactement de la même façon que l'algorithme ASAP mais les opérations, dans ce cas, sont ordonnancées en commençant par la dernière étape de contrôle (du bas du DFG vers le haut). Cet algorithme détermine l'ordonnancement le plus long possible qui demande le nombre maximum d'étapes de contrôle. Cependant, il ne réduit pas nécessairement le nombre d'UFs utilisées [16].

La figure 5.3 [30] résume l'algorithme ALAP. Les dates de début d'exécution obtenues par cet algorithme sont données par le vecteur $t^L=\{t_i^L;\ i=0,1,\ldots,n\}$. On considère une limite temporelle imposée notée $\overline{T_{tot}}$. Pour appliquer l'algorithme ALAP, il faut déterminer par avance la date limite de fin d'ordonnancement, elle est souvent prise égale à $\overline{T_{tot}}$.

Procedure ALAP ($G_s(V,E)$, $\overline{T_{tot}}$){
Ordonnancer $v_n$ : $t_n^L = T_{tot} + 1$ ;
Faire {
Sélectionner un nœud $v_i$ tel que tous ses successeurs sont ordonnancés ;
Ordonnancer vi : $t_i^L = \min_{j:(vi,vj)} t_j^L - d_i$ ; / $(v_i,v_j) \in E$
}
jusqu'à ce que $v_0$ soit ordonnancée ;
}

Figure 5.3 : Algorithme ALAP.

La figure 5.2.c illustre un ordonnancement ALAP du graphe de flot de données de la figure 5.2.a.

## 5.3.3. Programmation en nombres entiers ILP

La programmation en nombres entiers ILP (**I**nteger **L**inear **P**rogramming) [50] est l'une des méthodes utilisées pour l'ordonnancement sous contraintes temporelles, et utilisée également pour l'ordonnancement sous contraintes de ressources, elle permet de trouver une solution optimale en utilisant l'algorithme branch-and-Bound. Elle implique également quelques retour en arrière, i.e., des décisions prises auparavant peuvent être changées au fur et à mesure de l'évolution de l'ordonnancement.

Les méthodes de programmation en nombres entiers décrivent le problème de l'ordonnancement de façon exacte, sous forme d'un système d'inéquations. La résolution de ce système correspond à une recherche exhaustive. L'espace de recherche est définit selon les contraintes imposées par le concepteur.

Soit un ensemble de variables binaires avec deux indices: $X = \{x_{il},\ i = 0,1,\ldots,n\ ;\ l = 1,2,\ldots,\ \overline{T_{tot}}+1\}$. Le premier indice représente les opérations, le deuxième représente les étapes de contrôle. Avec $x_{il}=1$ si et seulement si $l=t_i$. Autrement dit $x_{il}=1$ si la date de début d'exécution de l'opération i est égale à l'étape de contrôle l.

$t_i^s$ et $t_i^L$ représentent les valeurs des dates d'exécution obtenues par les ordonnancements ASAP et ALAP. Comme la date de début d'exécution pour chaque opération est unique, alors :

$$\sum_{l=y1}^{l=y2} x_{il}=1, \qquad y1 = t_i^s \text{ , } y2 = t_i^L \quad \text{et} \quad i=0,1,\ldots,n \tag{1}$$

Les contraintes de précédence, aussi, doivent être vérifiées. Par conséquent, $t_i \geq t_j+d_j \;\; \forall\, i,j$ : $(v_j,v_i) \in E$ impose que :

$$\sum_{l=y1}^{l=y2} l.\, x_{il} \;\geq\; \sum_{l=z1}^{l=z2} l.\, x_{jl} + d_j \;\;, \;\; y1 = t_i^s, \; y2 = t_i^L, \; z1 = t_j^s, \; z2 = t_j^L, \; i,j=0,1,\ldots,n : (vj,vi) \in E \tag{2}$$

Le terme $\sum_{l=y1}^{l=y2} l.\, x_{il}$ représente le numéro d'étape durant laquelle l'opération i est exécutée.

Le problème d'ordonnancement en ILP est défini par les équations et les inégalités 1,2 et la fonction à minimiser, définie ci –dessous :

$$\text{Minimiser} \;\; \left( \sum_{k=1}^{n} (C_k * N_k) \right) \tag{3}$$

Où $1 \leq k \leq m$ types d'opérateurs sont disponibles, $N_k$ est le nombre d'UFs du type d'opérateur k, $C_k$ est le coût de chaque UF, il est généralement proportionnel à la surface de silicium de chaque type d'opérateur. La contrainte temporelle s'exprime par l'inégalité :

$$\sum_{l=y1}^{y2} l.\, x_{nl} \;\leq\; \overline{T_{tot}} \;+1, \qquad y1=t_n^s, \; y2 = t_n^L \tag{4}$$

La formulation ILP croît rapidement avec le nombre d'étapes de contrôle. Une unité en plus dans le nombre d'étapes de contrôle implique n variables x additionnelles. Par conséquent, le temps d'exécution connaît un accroissement rapide. C'est pourquoi l'approche ILP n'est utilisée, en pratique, que pour les petits problèmes (simples).

L'élimination du retour en arrière (back traking) dans la méthode ILP, peut réduire considérablement le temps d'exécution. Les méthodes heuristiques ont adoptés cette idée, par l'ordonnancement d'une opération à la fois basées sur un critère donné [31]. La section suivante décrit une telle méthode.

## 5.3.4. Algorithme d'ordonnancement par liste

Les techniques d'ordonnancement proposées dans la littérature utilisent principalement des heuristiques, elles permettent d'obtenir des solutions rapides et quasi optimales à un problème NP-complet. L'une des heuristiques les plus fréquemment employée est l'ordonnancement par liste LS (**L**ist **S**cheduling) [54,64]. L'algorithme d'ordonnancement par liste est une généralisation de l'algorithme ASAP avec inclusion de contraintes matérielles[31]. Cet algorithme effectue l'ordonnancement des opérations séquentiellement suivant une liste de priorité.

La liste de priorité des opérations est ordonnancée selon une fonction de priorité. A chaque itération, les opérations les plus prioritaires sont ordonnancées en premier et les opérations moins prioritaires sont retardées. En général, la liste de priorité est dynamique, car l'ordonnancement d'un opérateur à une étape de contrôle rend ses successeurs prêts, et y sont ainsi insérés dans la liste de priorité [31]. Les méthodes existantes diffèrent essentiellement par la fonction de priorité utilisée pour sélectionner les opérations à retarder.

Le principe d'un algorithme d'ordonnancement par liste est le suivant, à chaque étape de contrôle, les opérations candidates sont stockées dans une liste et ordonnées par une fonction de priorité. Chaque opération de la liste est prise en compte ; elle est ordonnancée si les contraintes imposées le permettent sinon elle est retardée. Ce processus est réitéré jusqu'à l'ordonnancement complet de toutes les opérations de la description comportementale.

L'ordonnancement par liste permet de résoudre aussi bien le problème de l'ordonnancement sous contraintes temporelles que sous contraintes matérielles.

**Définition 5.1 :**
A l'étape de contrôle courante l, l'ensemble des opérations candidates du type k, $U_{l,k}$ est l'ensemble des opérations du type k qui ont leur prédécesseurs déjà ordonnancés.

$$U_{l,k} = \{v_i \in V : \tau(v_i) = k / t_j + d_j \leq l \ \forall \ j : (v_j, v_i) \in E\}, k = 1,2,\ldots, n_{res}.$$

**Définition 5.2 :**
L'ensemble des opérations à finir $T_{l,k}$ est l'ensemble des opérations dont la date de début d'exécution est inférieure à l et dont l'exécution n'est pas finie à l'étape de contrôle courante. Autrement dit $T_{l,k} = \{ v_i \in V : \tau(v_i) = k / t_j + d_j > l \}$. Dans le cas où la durée d'exécution des opérateurs est égale à 1 cet ensemble est vide.

Dans le cas d'un ordonnancement sous contraintes de ressources, le nombre d'opérations candidates qui peuvent être ordonnancées à chaque étape de contrôle est limitée. Une opération candidate sera donc ordonnancée si les ressources nécessaires à son exécution sont disponibles. L'algorithme général est donné à la figure 5.4.

**LIST** (Gs(V,E), a){
    l=1 ;
    Faire {
        Pour chaque type de ressource k = 1, 2, …, $n_{res}$ {
        Déterminer les opérations candidates $U_{l,k}$ ;
        Déterminer les opérations à finir $T_{l,k}$ ;
        Sélectionner $S_k$ nœuds tels que $S_k \subset U_{l,k} / |S_k| + |T_{l,k}| \leq a_k$ ;
        }
        l=l+1 ;
    }
    jusqu'à ce que $v_n$ soit ordonnancée ;
}

Figure 5.4 : Algorithme d'ordonnancement par liste sous contraintes matérielles

Une fonction de priorité, simple et souvent utilisée, est basée sur l'intervalle de positionnement (Time Frame) des opérations. Elle est proportionnelle inversement à l'intervalle de positionnement, i.e., qui a plus de mobilité est le moins prioritaire et vice-versa

[31]. Ainsi , les opérations qui ont un large intervalle de positionnement sont différées à des étapes de contrôle ultérieures puisque le nombre des étapes de contrôle auxquelles elles peuvent être assignées est plus grand. En d'autres termes, et de façon plus formelle, à chaque étape une opération $t_i \in T$ est choisie tel que |Time Frame ($t_i$)| est le plus petit. Cette opération est assignée à l'étape de contrôle s, tel que le nombre d'opérations du type de ressources Ressource($t_i$) déjà affectées à cette étape soit minimal.

$$\text{Si } S(t_i) = s \text{ , alors } \forall\ t_j \in T_{succ(ti)} \text{ , Time Frame } (t_j) \geq \Gamma\ (t_i,t_j) + S(t_i)$$
$$\forall\ t_j \in T_{pred(ti)} \text{ , time frame } (t_j) \leq S(t_i) - \Gamma\ (t_j,t_i)$$

Le système SLICER développé à l'université de l'Illinois utilise ce type de fonction de priorité. Le système MAHA ordonnance en priorité les fonctions appartenant aux chemins critiques puis ordonnance les fonctions à plus faible mobilité. Alors que le système ELF intègre dans la fonction de priorité la mobilité des opérations et le nombre de successeurs de chaque opération [31]. Beaucoup d'autres systèmes existants utilisent l'ordonnancement par liste mais se diffèrent par la fonction de priorité qu'ils utilisent : HYPER [55], système HAL, TBS, CASH [56], NEAT et MACH. Considérons l'exemple de la figure 5.2. On suppose qu'on dispose de deux types d'opérateurs, un multiplieur et une UAL qui permet de réaliser les opérations de types addition, soustraction et comparaison. Le nombre de multiplieurs et d'UAL est : $a_1$=3 et $a_2$=1. Le temps d'exécution de la multiplication est 2, celui de l'UAL est 1. Considérons comme fonction de priorité la longueur du plus court chemin entre $v_i$ et $v_n$.

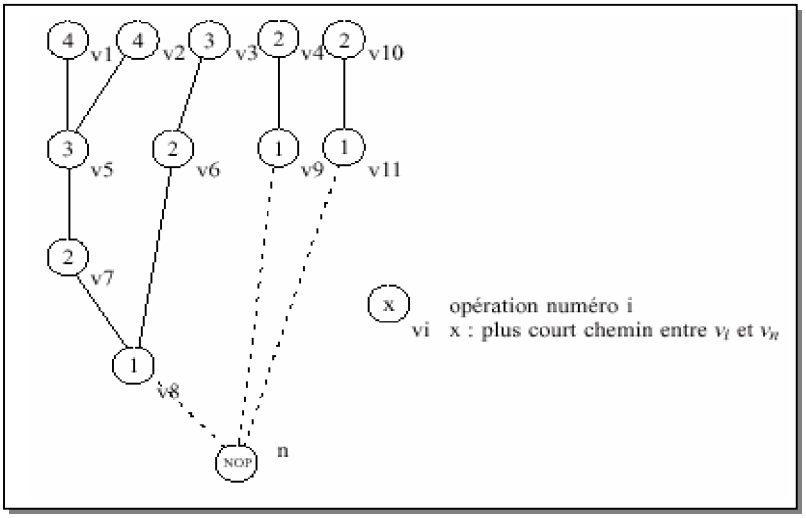



Figure 5.5 : priorités associées aux opérations

La figure 5.5 donne pour chaque opération la priorité qui lui est associée. A la première étape de contrôle, nous avons pour k=1 (multiplieur), $U_{1,1}=\{v_1,v_2,v_3,v_4\}$, $T_{1,1} = \emptyset$ . puisqu'on a seulement trois multiplieurs, on sélectionne et ordonnance les trois opérations les plus prioritaires $S_1=\{v_1,v_2,v_3\}$. Pour k=2 (UAL), $U_{1,2}=\{ v_{10}\}$, $T_{1,2} = \emptyset$ , l'opération $v_{10}$ est ordonnancée.

A la deuxième étape de contrôle, nous avons pour k=1, $U_{2,1}=\{v_4\}$, $T_{2,1} =\{v_1,v_2,v_3\}$, donc aucune opération ne peut être ordonnancée $S_1= \emptyset$. Pour k=2, $U_{2,2}=\{v_{11}\}$, $T_{2,2} = \emptyset$, $v_{11}$ est ordonnancée.

L'ordonnancement par liste assigne les opérations candidates dés les premières étapes de contrôle, sans tenir compte de manière précise l'allocation. Les premiers pas de contrôle

connaissent une forte occupation, alors que les derniers ont une faible occupation. Comme la durée de vie des variables se voit trés largement étendue pour transmettre les données aux opérations des dernières étapes de contrôle, ceci a une influence sur le nombre de registres nécessaires et qui peut être relativement important.

### 5.3.5. Algorithme d'ordonnancement par liste statique

l'algorithme d'ordonnancement par liste statique (**S**tatic **L**ist **S**cheduling) [57] diffère de l'algorithme d'ordonnancement par liste ordinaire aussi bien en assignation d'étapes de contrôle qu'en maintnance de la liste de priorité. Cette approche commence par la création d'une large liste et unique. L'algorithme utilise les approches ASAP et ALAP pour obtenir les étapes limites auxquelles est affectée chaque opération (LCS : **L**east **C**ontrol **S**tep et GCS : **G**reatest **C**ontrol **S**tep ). Ensuite, toutes les opérations sont ordonnancées en ordre décroissant suivant la valeur GCS comme première clé, et la valeur LCS comme deuxième clé.

Une fois la liste de priorité créée, les opérations sont ordonnancées séquentiellement en commençant par la dernière opération dans la liste (la plus prioritaire ). Le type de l'opération est important, si par exemple une opération d'addition est moins prioritaire que des opérations de multiplication mais qu'elle est la plus prioritaire parmi les opérations d'addition, alors elle peut être ordonnancée en premier si un additionneur est disponible. Cet algorithme a un avantage, par rapport à l'ordonnancement par liste est qu'une liste est construite statiquement une fois pour toute et elle n'évolue pas dynamiquement [31].

### 5.3.6. Ordonnancement orienté par les forces FDS

L'ordonnancement orienté par les forces FDS (**F**orce **D**irected **S**cheduling) est une méthode heuristique constructive [33,36]. C'est une technique d'ordonnancement très populaire pour l'ordonnancement sous contraintes de temps, qui a pour objectif de réduire le nombre de ressources matérielles en les répartissant au mieux sur un nombre d'étapes de contrôle fixé au préalable.

Avant de donner la description de l'ordonnancement orienté par les forces, on va passer par certaines définitions.

L'intervalle de positionnement d'une opération (Time Frame) est compris entre les dates de début d'exécution ASAP et ALAP $\{[ t_i^s , t_i^L ], i= 0,1,\ldots,n\}$. A chaque opération $v_i$ est associée une probabilité $p_i(l)$ d'être ordonnancée à une étape de contrôle l, égale à l'inverse de la longueur de l'intervalle de positionnement de l'opération :

$$p_i(l) = \frac{1}{\mu_i + 1} , \forall\ l \in [ t_i^s , t_i^L ] , \mu_i = t_i^L - t_i^s .$$

Maintenant, puisqu'on a la probabilité d'assignation d'une opération à une étape de contrôle, on peut définir, pour chaque étape de contrôle l, un graphe de distribution d'un type d'opérateur

$$\{ q_k(l) ; k=1,2,\ldots,n_{ress} \} \quad \text{par} \quad q_k(l) = \sum_{v_i:\tau(v_i)=\kappa} P_i(l)$$

Le graphe de distribution donne pour chaque étape de contrôle une estimation du nombre d'opérations qui pourront y être exécutées et donne donc une idée sur la quantité de ressources nécessaires.

L'objectif est d'équilibrer les graphes de distribution de chaque type d'opérateur par rapport aux pas de contrôle, ce qui engendre la réduction du nombre de ressources matérielles et en assure l'utilisation efficace. Une variation qui équilibre la valeur des graphes de distribution sur l'ensemble des pas de contrôle permet de minimiser le nombre d'opérateurs. Pour pouvoir mesurer les différentes variations on définit une quantité appelée force propre (self-force), cette notion relie les opérations aux étapes de contrôle. Elle est utilisée comme guide en vue de choisir la prochaine opération et son emplacement. Cette force entre une opération et une étape est proportionnelle au nombre d'opérations de même type qui pourrait prendre place pendant cette étape.

Une force propre F(i,l) mesure la variation de la distribution sur l'ensemble des pas de contrôle pondérée par la valeur du graphe de distribution $q_k(m)$, si l'opération $v_i$ / $\tau$ ($v_i$)=k est ordonnancée au pas de contrôle l, on note :

$$F(i,l)= \sum_{m=y1}^{y2} q_k(m)(\delta_{im} - p_i(m)), \qquad y1= t_i^s, y2 = t_i^L$$

Avec $\delta_{im}$ = 1 si l'opération est ordonnancée à l'étape de contrôle m, 0 sinon.

Finalement, on définit une force totale pour chaque opération, elle est égale à la somme des forces des prédécesseurs et des successeurs plus la force propre de l'opération même.

L'algorithme est résumé à la figure 5.6. L'algorithme FDS est basé sur le calcul des forces totales. Après calcul des intervalles de positionnement et des graphes de distribution, pour chaque opération la force totale est calculée pour toutes les assignations possibles de son intervalle de positionnement. L'assignation qui minimise la force totale est sélectionnée.

```
FORCE ( Gs(V,E) ; T̄tot ) {
    Faire {
        Calculer les intervalles de positionnement
        Calculer les graphes de distribution de chaque type d'opérateur
        Pour chaque opération non encore ordonnancée faire {
            Pour chaque pas de contrôle de son intervalle faire {
                Calculer la force totale
            }
        }
        ordonnancer l'opération  vi au pas de contrôle l qui minimise Fp/s(i,l)
    }
    jusqu'à ce que toutes les opérations soient ordonnancées
}
```

Figure 5.6 : Algorithme d'ordonnancement par les forces

En appliquant cette approche globale qui prend en compte l'influence de l'assignation d'une opération à une étape de contrôle sur les opérateurs, l'algorithme FDS permet une bonne utilisation des opérateurs au cours du temps.

### 5.3.7. Réordonnancement itératif

Par manque d'un schéma préventif, l'algorithme FDS est capable de produire une solution sub-optimale. On peut dépasser ce point faible par le réordonnancement de quelques opérations à chaque ordonnancement partiel [31]. La méthode de réordonnancement itératif IR(**I**teratif **R**eordonnancement) [59], procède par le réordonnancement d'une opération à la fois. Dans n'importe quel ordonnancement initial, l'assignation d'une opération à une étape de contrôle est faite en prenant en compte les dépendances des données et les contraintes. Un déplacement aléatoire est choisi, et l'opération est bloquée (immobilisée) dans la nouvelle position temporairement. De même, pour les autres déplacements jusqu'à immobilisation de toutes les opérations. Les coûts de ces déplacements sont calculés, et celui qui produit un gain maximum ( réduction maximale du coût) est choisi, et les opérations déplacées durant la même séquence que ce déplacement sont libérées (débloquées). Puis la procédure est répétée avec le nouveau schéma d'ordonnancement. La qualité du résultat produit par cet algorithme dépend de la solution initiale [31].

## 5.4. Algorithmes d'ordonnancement orientés flot de contrôle

L'algorithme d'ordonnancement à base de chemins PBS (**P**ath **B**ased **S**cheduling) [41] était le premier algorithme proposé pour répondre aux exigences du problème d'ordonnancement des applications dominées par le contrôle. Depuis, plusieurs algorithmes d'ordonnancement ont été développés pour le contrôle, mais il n'y a pas d'algorithmes qui prend en considération toutes les questions en relation [4] : manipulation des constructions de contrôle y compris les boucles, exploitation de dépendance de données, prise en considération d'une variété de contraintes (contraintes d'E/S, contraintes de temps, contraintes de ressources,…) et de critères d'optimisation (performance, surface, puissance, etc.).

L'algorithme d'ordonnancement à base d'arbres [60] est venu pour réduire, en quelque sorte, le problème d'explosion de chemins dans PBS. Cet algorithme, aussi, effectue un ordonnancement AFAP pour tous les chemins. L'arbre (rooted tree) utilisé pour représenter la collection de chemins est moins compact comparé au cas où on considérerait tous les chemins individuellement, mais peut toujours être exponentiellement large. L'algorithme d'ordonnancement à base d'arbres prend en considération les dépendances de données, ainsi, et contrairement à l'algorithme PBS, il n'est pas contraint par l'ordre initial des opérations. Il tient compte également des contraintes de ressources, mais celles impliquant les délais d'opérateurs et concernant l'ordre des événements d'E/S ne sont pas prises en considération.

Dans toutes ces approches orientées à l'ordonnancement des circuits dominés par le contrôle, un aspect important est encore ignoré : les boucles dans le flot de contrôle. Dans tous ces cas, les boucles sont "cassées" au début, par l'enlèvement des arcs de rebouclage (feedback ).

L'algorithme d'ordonnancement orienté boucles LDS (**L**oop **D**irected **S**cheduling) traverse tous les chemins du flot de contrôle, y compris ceux qui contiennent des arcs de rebouclage. Cet algorithme, cependant, ne prend pas en considération les contraintes de temps et d'E/S, et ne permet pas l'enchaînement des opérations [4].

L'algorithme d'ordonnancement à boucles dynamiques DLS (**D**ynamic **L**oop **S**cheduling) [51,61,62,63] est similaire à l'algorithme LDS, il traverse tous les chemins y compris les boucles et ne permet pas, également, l'enchaînement d'opérations.

Un autre algorithme d'ordonnancement appelé ordonnancement à base de segments SBS (**S**egment **B**ased **S**cheduling) [4], a été développé pour combler les limites des autres algorithmes. Cet algorithme a deux composants majeurs : une procédure efficace pour le parcours du flot de contrôle/flot de données et une procédure étendue pour analyser ou

localiser les états de limites (state boundaries) en prenant en compte une variété de contraintes et d'exigences. Il évite la construction de tous les chemins de contrôle avant l'ordonnancement. Durant l'ordonnancement, le graphe de contrôle est traversé et divisé en segments. Chaque segment représente une collection de chemins d'un seul état aux autres.

D'autres approches ont été proposées pour les structures de boucles telles que le : pipelining [62] et loop folding.

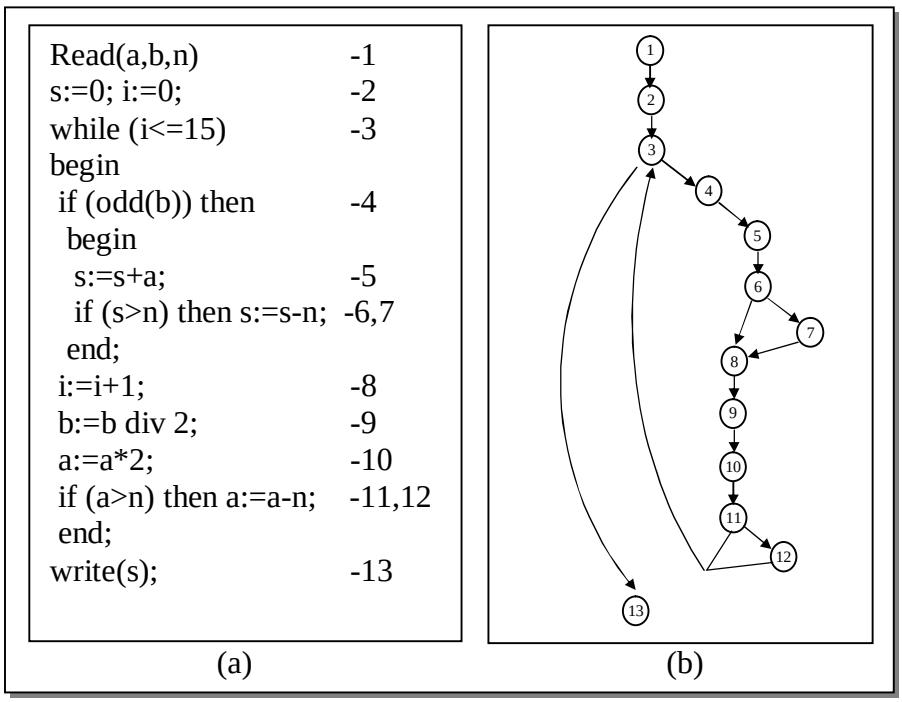



Figure 5.7 : (a) Description algorithmique de la fonction ab mod n
(b) CFG correspondant

## 5.4.1. Ordonnancement à base de chemin PBS

L'algorithme d'ordonnancement à base de chemin PBS (**P**ath **B**ased **S**cheduling) [41,63]était la première tentative pour satisfaire les besoins des circuits de contrôle [4].

L'algorithme à base de chemins minimise le nombre d'étapes de contrôle nécessaires pour exécuter chaque chemin dans le CFG [64].

En premier lieu, le graphe de flot de contrôle est rendu acyclique en éliminant tous les arcs de retour des boucles, tout en mémorisant ces arcs (le premier et le dernier nœud ) et la condition d'entrée à la boucle. Ensuite, l'algorithme extrait tous les chemins d'exécution possibles et les ordonne indépendamment. Un chemin $P=(v_1,\ldots,v_n)$ commence soit avec le premier nœud du graphe ou le premier nœud d'une boucle, et termine avec le nœud qui n'a pas de successeur. Le graphe est alors divisé en pas de contrôle de telle façon à respecter les contraintes de dépendance.

L'algorithme génère des contraintes $\{C_1,C_2,\ldots,C_r\}$ entre les nœuds qui sont affectés à des étapes de contrôle différentes. Le problème de trouver le nombre minimal d'étapes de contrôle dans chaque chemin est équivalent à trouver le nombre minimal de coupures pour chaque chemin. Pour cela, un graphe d'intervalles de contraintes (intervalles de coupure) est défini, où chaque nœud représente un intervalle de contrainte et les arcs représentent les chevauchements entre les intervalles de contrainte.

Une contrainte $C_k$ correspond à un intervalle $IC_k=\{v_i,v_{i+1},\ldots,v_j\}$, tel que $1<i<j\leq n$, signifie que la contrainte $C_k$ est violée entre les nœuds $v_{i-1}$ et $v_j$.

Le problème de minimisation du nombre d'étapes de contrôle est transformé en un problème de partitionnement en cliques du graphe (clique-partitionning graph). Une solution du partitionnement en cliques doit alors indiquer le minimum d'intervalles qui ne se chevauchent pas dans un chemin.

La technique de cliques est utilisée à nouveau pour combiner les intervalles générés des différents chemins. Le résultat de cette étape représente l'ensemble final des intervalles pour le graphe de flot de contrôle entier, on génère ainsi un ordonnancement global pour le graphe original.

**Définition 5.3 : Chemin**

Soit G=(V,E) un CFG acyclique, un chemin P est une séquence de nœuds ($v_1$, $v_2$, …, $v_n$) , où :

(i) $v_1$ est soit le premier nœud du graphe ou bien le premier nœud d'une boucle.

(ii) $v_n$ est un nœud sans successeur.

**Définition 5.4 : Chemin ordonné**

Soit P= ($v_1$, $v_2$, …, $v_n$) un chemin, soit $C_1$,…, $C_r$ un ensemble de contraintes. Un chemin ordonné SP est une séquence de nœuds ($v_i$, $v_{i+1}$, …, $v_j$) où :

(i) $1 \leq i < j \leq n$

(ii) $\exists \{C_m, \ldots, C_k\}$ tel que Succ (SP) $\in IC_m \cap IC_{m+1} \cap \ldots \cap IC_k$

Du fait que les arcs de retour des boucles ont été supprimés, les chemins ordonnés satisfaisant les conditions suivantes vont être ajoutés :

- $\forall$ SP = ($v_i$, $v_{i+1}$, …, $v_j$) tel que $\exists$ $v_i \in$ SP, $i \leq l \leq j$ et $v_l$ est le dernier nœud d'une boucle. Un chemin ordonné SP' = ($v_i$, …, $v_l$) est inséré dans l'ensemble des chemins ordonnés.
- Succ (SP') = $v_f$ tel que $v_f$ est le premier nœud de la boucle.

L'étape finale est la construction de la machine d'états finis réalisant le comportement du graphe de flot de contrôle initial. Tous les chemins ordonnés ayant la même tête auront le même état initial et tous les chemins ayant le même successeur auront le même état final.

**Avantages de l'ordonnancement à base de chemins [42] :**

- *Formulation* : puisque l'algorithme d'ordonnancement à base de chemins modélise le problème d'ordonnancement comme étant un problème de couverture sur des cliques, des solutions optimales peuvent être trouvées.
- *Ordonnancement d'une opération dans plusieurs étapes de contrôle* : la minimisation du nombre d'étapes de contrôle pour chaque chemin d'exécution d'un CFG respectant les contraintes est rendu possible par la possibilité d'ordonnancer une opération dans plusieurs étapes de contrôle.
- *Traitement efficace des structures conditionnelles* : les opérations conditionnelles ne constituent pas un handicape pour l'algorithme PBS, elles sont utilisées dans la dérivation des chemins d'exécution. Une fois les chemins générés, aucune considération n'est attribuée à ces opérations, elles sont traitées de la même façon que les autres opérations.

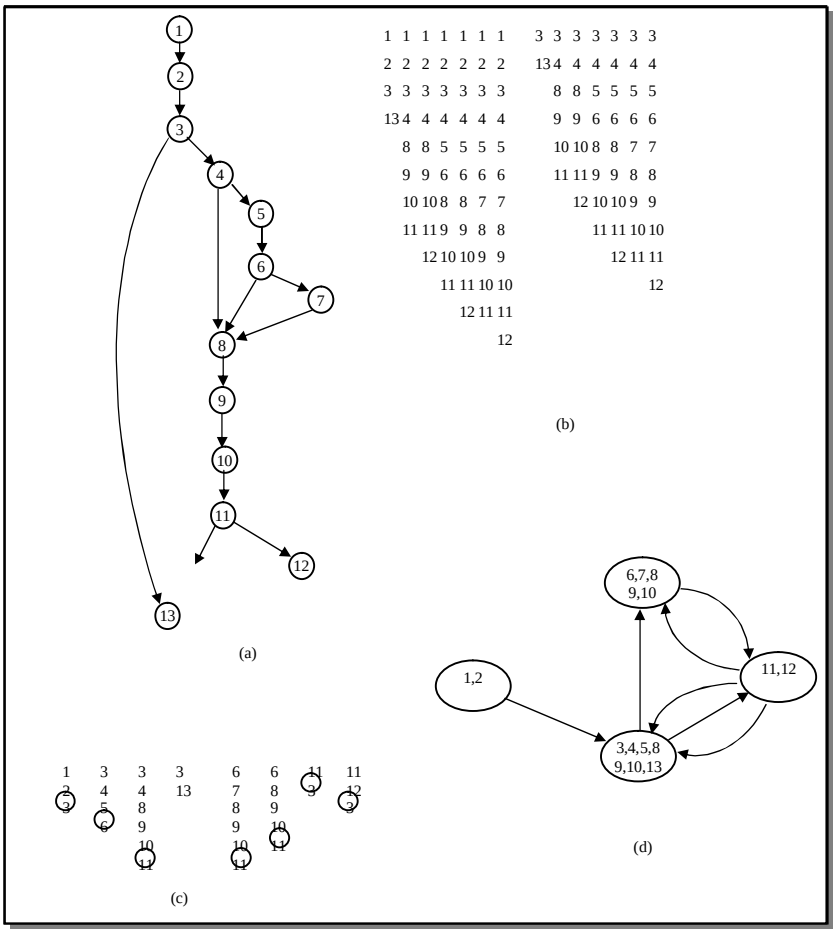



Figure 5.8 : (a) CFG acyclique, (b) Chemins, (c) Chemins ordonnées
(d) FSM pour l'ordonnancement à base de chemin

**Inconvénients de l'ordonnancement à base de chemins [42]:**

- *Explosion de chemins* : la complexité de l'ordonnancement à base de chemins est proportionnelle au nombre de chemins d'exécution qui peuvent augmenter d'une façon exponentielle avec le nombre d'opérations conditionnelles. Ainsi, cette approche demande beaucoup de temps et d'espace pour énumérer tous les chemins possibles.
- *Ordre fixe* : une autre limitation de l'algorithme est le traitement des opérations selon leur ordre d'apparition, donc les opérations ne peuvent pas être réordonnées ou parallélisées et la solution obtenue est optimale seulement pour l'ordre d'opérations donné. Dans certains cas, en réordonnant certaines opérations, il est possible d'ordonnancer un chemin en un nombre d'étapes de contrôle moindre.
- *Traitement de boucles* : la suppression des arcs de retour des boucles empêche le chevauchement d'itérations successives d'une boucle, ce qui empêche d'augmenter le parallélisme.

## 5.4.2. Ordonnancement à boucle dynamique DLS

L'algorithme d'ordonnancement à boucle dynamique DLS (**D**ynamic **L**oop **S**cheduling) [51,61,63]a été proposé pour combler les limites de l'algorithme à base de chemin, en offrant une procédure améliorée pour le traitement de boucles. Il utilise un graphe

de flot de contrôle, et préserve tous les arcs de retour des boucles permettant ainsi l'exécution parallèle des parties de deux itérations successives d'une boucle.

La première étape de l'algorithme DLS consiste à calculer la liste des successeurs, pour chaque opération dans le CFG, c'est à dire la liste des opérations ayant la possibilité de s'exécuter immédiatement après cette opération.

Partant de la table des successeurs, est calculé l'ensemble des chemins d'exécution possibles. L'idée est de réaliser un ordonnancement de chaque chemin indépendamment des autres. Le premier chemin commence avec le premier nœud dans le CFG. La génération du chemin courant est arrêtée et un nouveau chemin commence quand l'une des deux conditions suivantes est satisfaite :

- Une instruction *wait* est rencontrée.
- Le nouveau nœud rencontré est dépendant de l'un des nœuds déjà existants dans le chemin.

**Définition 5.5 : chemin ordonné**

Soit G=(V,E) un CFG et $C_1,\ldots, C_r$ un ensemble de contraintes. Un chemin ordonné SP est une séquence de nœuds $(v_i, v_{i+1}, \ldots, v_j)$ où :

(i) $\exists\ v_l$ , $C_m$ tel que $i \leq l \leq j$ et $C_m(v_l, v_{j+1})=1$.

En dernière étape, est générée la machine d'états finis correspondante. Tous les chemins ayant la même tête sont mis ensemble dans le même état.

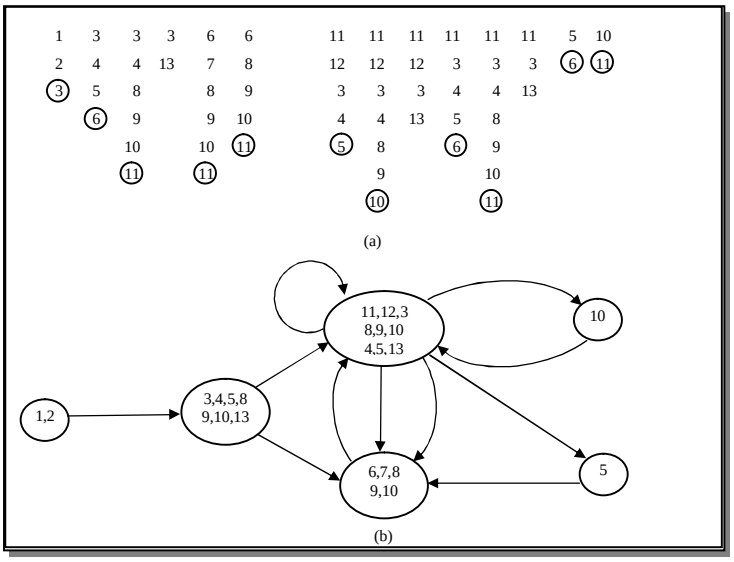



Figure 5.9 : (a) Chemins et successeurs, (b) FSM pour l'ordonnancement à boucle dynamique

**Caractéristiques de l'algorithme d'ordonnancement à boucle dynamique :**

- *Découpage à la volée :* l'ordonnancement à boucle dynamique interrompe la génération de chemins à la volée, en d'autres termes, la génération du chemin courant est arrêtée si une contrainte est violée. DLS offre ainsi la possibilité de recouvrir deux itérations successives d'une boucle. Ceci réduit largement le nombre de chemins générés.
- *Dynamicité des boucles* : avec l'algorithme DLS, les arcs de retour des boucles ne sont pas éliminés comme dans le cas de l'algorithme à base de chemin. Ainsi, il est possible de chevaucher dans certains cas les itérations successives d'une boucle, d'où la notion de dynamicité de boucles.

- *Ordre fixe :* cependant, l'algorithme DLS lui aussi, comme c'est le cas pour l'algorithme à base de chemin, est limité par l'ordre fixe des opérations. Où les opérations sont traitées dans l'ordre de leur apparition dans le CFG.

## 5.4.3. Algorithme d'ordonnancement pipeliné à base de chemin

La force principale de l'ordonnancement pipeliné à base de chemins PPS(**P**ipeline **P**ath-based **S**cheduler ) [62,63] est l'optimisation des boucles, étant donné que les boucles sont le plus souvent les parties les plus critiques temporellement dans une application. Cet algorithme diffère de l'algorithme d'ordonnancement à base de chemins ordinaire dans la méthode d'extraction des chemins. Il commence par le premier nœud dans le CFG et termine avec un nœud qui n'as pas de successeurs [63]. En cas de présence d'une boucle, un ensemble de chemins est généré où la boucle est déroulée pour une ou deux itérations. Donc en cas d'inexistence de boucles, le fonctionnement de PPS sera identique à celui de PBS.

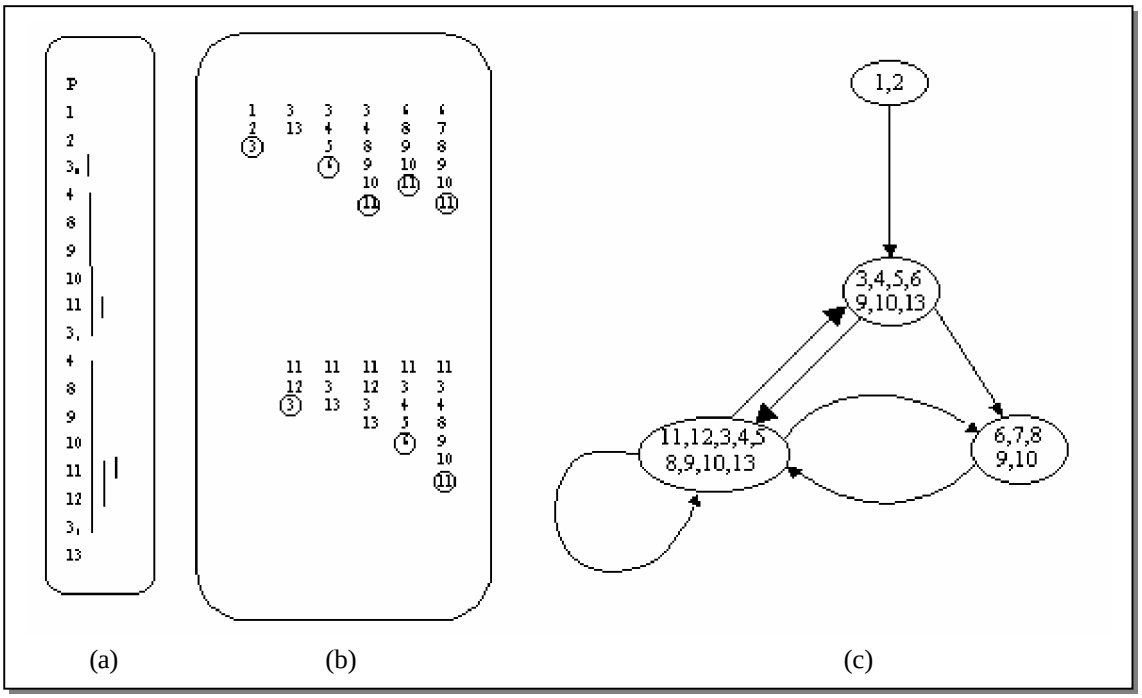



Figure 5.10 : (a) un chemin et les coupures, (b) chemins aprés coupures
(c) FSM pour l'ordonnancement pipeliné à base de chemin

PPS bénéfit des avantages des algorithmes PBS et DLS, en utilisant une technique de couverture en cliques pour découper les chemins tout en utilisant une nouvelle technique pour pipeliner les itérations d'une boucle afin d'identifier tout parallélisme qui peut y exister.

## 5.4.4. Dominator-Path Sheduling

Dominator-Path Scheduling (DPS) est une méthode d'ordonnancement global qui peut exploiter le parallélisme dans les multiples blocs de base. DPS sélectionne des groupes de blocs à ordonnancer comme un seul meta-bloc. L'arbre dominateur (dominator tree) d'une fonction est une représentation de la relation du dominateur parmi les blocs du CFG de la fonction. Un bloc de base, D, domine un autre bloc B, si tous les chemins de la racine du CFG à B doivent traverser D. Après avoir construit l'arbre du dominateur, un chemin dans cet arbre est sélectionné pour l'ordonnancement et est combiné dans un seul meta-bloc. Ce meta-bloc

résultant est ordonnancé à l'aide d'un algorithme d'ordonnancement local. Pendant l'ordonnancement, les opérations sont libres de se déplacer (tant qu'il n'y a pas de dépendance de données) à travers les limites du bloc afin d'exploiter le parallélisme sur un chemin dominateur.

DPS souffre de l'incapacité des techniques d'ordonnancement local de considérer les fréquences d'exécution différés dans les meta-blocs comme les opérations sont ordonnancées partout dans le meta-bloc combiné. Un ordonnanceur local traditionnel déplace les opérations sans prendre en considération la fréquence d'exécution potentielle de la localité dans laquelle l'opération est placée finalement. Donc, l 'algorithme d'ordonnancement peut déplacer une opération d'un bloc qui a une fréquence d'exécution relativement basse à un autre qui a une plus haute fréquence. Ce qui peut influencer le nombre de cycles d'exécution.

### 5.4.5. Software pipelining

Les boucles sont une source idéale de parallélisme du fait que des opérations de différentes itérations peuvent être ordonnancées ensemble [65]. L'une des approches consiste à faire un dépliage du graphe, c'est à dire un déroulement partiel de la boucle et puis les opérations des itérations déroulées sont ordonnancées ensemble, pour obtenir de meilleurs résultats dans certains cas. Les opérations à la fin de la boucle déroulée sont ordonnancées sans prendre en considération les opérations au début de la boucle. Donc, à la fin de chaque itération de la boucle déroulée, un délai peut survenir attendant que les opérations finissent leur exécution. Pour éviter ce délai, on peut utiliser le "software pipelining" [66]. En la présence de boucle, on ne veut pas seulement réduire le temps que prend une itération mais aussi réduire le temps entre les itérations [67].

## 5.5. Conclusion

Après la description, faite dans ce chapitre, des différents algorithmes d'ordonnancement les plus communément utilisés dans la synthèse de haut niveau, on peut constater que le problème d'ordonnancement a fait le sujet de nombreuses recherches et plus spécialement les algorithmes d'ordonnancement orientés flot de données. Cependant, ces derniers ne s'adaptent pas très bien aux besoins des circuits de commande, qui réalisent un traitement complexe sur un faible flux d'informations. Les algorithmes d'ordonnancement orientés flot de contrôle essayent de satisfaire ces besoins. La totalité de ces algorithmes sont dérivés de l'algorithme d'ordonnancement à base de chemins qui était le pionnier dans ce domaine.

Bien que ces approches retournent de bons résultats, mais il n'est pas garantit qu'elles produisent la solution optimale. C'est pourquoi on essaye de proposer une nouvelle technique basée sur les algorithmes génétiques, afin de réaliser une meilleure exploration de l'espace des solutions.

Chapitre

# 6

# Algorithme d'ordonnancement GPBS

## 6.1. Introduction

Les algorithmes de résolution du problème d'ordonnancement sont largement présentés dans la littérature [24,26,27,28,48,68,69,70,71,72,73,74,75,76], mais peu de ces travaux traitent le problème d'ordonnancement des circuits dominés par le contrôle.

L'objectif commun de ces algorithmes est de trouver la solution optimale avec respect d'un ensemble de contraintes. Ainsi, deux familles de méthodes de résolution du problème d'ordonnancement dans la synthèse de haut niveau, classifiées selon l'exactitude de la solution fournie, peuvent être distinguées [24,83]:

1. *Algorithmes exactes* : ce type d'algorithmes retourne toujours la solution optimale. L'inconvénient majeur d'appliquer ces méthodes à des problèmes d'ordonnancement du monde réel, de grande taille, est qu'ils exécutent une recherche exhaustive qui requière des temps d'exécution prohibitifs. C'est pourquoi, en pratique, ces algorithmes ne sont généralement utilisés que pour résoudre des problèmes de petite taille.

2. *Heuristiques* : ce type d'algorithmes trouve toujours une solution, mais aucune garantie n'est donnée sur la qualité de cette solution. Le plus souvent le résultat est une solution approchée (sous-optimale). L'avantage principal est qu'en général ces algorithmes retournent une solution en un temps plus réduit par rapport aux algorithmes exactes.

En général, la recherche exhaustive requière un temps d'exécution long et les heuristiques produisent des résultats de qualité médiocre. C'est pourquoi les chercheurs se sont orientés vers l'utilisation des méthodes d'optimisation pour résoudre le problème d'ordonnancement tel que les *algorithmes génétiques.* le choix des algorithmes génétiques est

basé sur le fait qu'ils ont été appliqués avec succès à plusieurs problèmes d'optimisation combinatoire et ont été jugés intéressants pour résoudre le problème d'ordonnancement [24]. Les algorithmes génétiques présentent un ensemble de qualités remarquables, ils sont simples, efficaces, robustes et généraux; et aucune connaissance de l'espace de la recherche n'est requise . Ce sont des approches probabilistes mais pas totalement aléatoires.

Dans ce chapitre, on essaye d'exploiter l'efficacité de résolution des problèmes d'optimisation complexes des algorithmes génétiques, pour résoudre le problème d'ordonnancement des circuits de contrôle.

## 6.2. Algorithmes génétiques

Les algorithmes génétiques sont des algorithmes informatiques inspirés de la théorie darwinienne. Cette théorie repose sur deux postulats simples :

- " Dans chaque environnement, seul les espèces les mieux adaptées perdurent au cours des temps, les autres étant condamnées à disparaître ".
- " Au sein de chaque espèce, le renouvellement des populations est essentiellement dû aux meilleurs individus de l'espèce ".

Ce sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de la génétique et de l'évolution naturelle, à savoir les croisements, les mutations, la sélection, etc [81]. Les algorithmes génétiques ont déjà une histoire relativement ancienne puisque les premiers travaux de John Holland sur les systèmes adaptatifs remontent à 1962. Avant cette date quelques tentatives ont été faites pour modéliser les systèmes génétiques par des systèmes informatiques (Barricelli 1957, 1962, Fraser 1960, 1962, Martin e Cockerham, 1960). Cependant l'objectif fondamental de ces études était de comprendre quelques phénomènes biologiques.

John Holland et ses étudiants dans l'Université de Michigan étaient les premiers à reconnaître l'utilité d'utiliser des Opérateurs Génétiques dans les problèmes de l'adaptation artificiels. Mais c'était Bagley en 1967 qui a mentionné, en premier lieu, l'expression " Algorithme Génétique" et le premier a avoir présenter une application pratique de cette connaissance. Aujourd'hui, leurs champs d'application sont très vastes. Les raisons du grand nombre d'applications sont la simplicité et l'efficacité de ces algorithmes. Les AG sont basés sur des mécanismes très *simples*, ils sont *robustes* car ils peuvent résoudre des problèmes fortement non-linéaires et discontinus, et *efficaces* car ils font évoluer non pas une solution mais toute une population de solutions potentielles et donc ils bénéficient d'un parallélisme puissant.

les AG sont basés sur une approche itérative et heuristique. En cela, peu d'information est nécessaire pour leur utilisation: l'espace de recherche possible et un critère d'efficacité correspondant au *fitnes*s.

Ces algorithmes utilisent un vocabulaire similaire à celui de la génétique, cependant, les processus auxquels ils font référence sont beaucoup plus complexes. Le principe des algorithmes génétiques repose sur une analogie entre un individu dans une population et la solution d'un problème parmi un ensemble de solutions potentielles : un individu (une solution) est caractérisée par une structure génétique « génotype de l'individu ou chromosome » (codage des solutions du problème). Selon les lois de survie énoncées par Darwin, seuls les individus les plus forts (les meilleures solutions) survivront et pourront donner une descendance. Les opérateurs de croisement et de mutation (recombinaison et mutation des codages des solutions) permettent de se déplacer dans l'espace des solutions du problème. A partir d'une population initiale et après un certain nombre de générations, on obtient une population d'individus forts, c'est à dire de bonnes solutions se rapprochant de la solution optimale du problème considéré.

**Méthode générale des algorithmes génétiques**

Un algorithme génétique recherche le ou les extrema d'une fonction définie sur un espace de données. Pour l'utiliser, on doit disposer des cinq éléments suivants :

- *un principe de codage de l'élément de population*. Cette étape associe à chacun des points de l'espace d'état une structure de données. Cette structure conditionne le succès des algorithmes génétiques.
- *un mécanisme de génération de la population initiale*. Ce mécanisme doit être capable de produire une population d'individus non homogène qui servira de base pour les générations futures.
- *une fonction de mérite.* Celle-ci retourne une valeur appelée fitness ou fonction d'évaluation de l'adaptation de l'individu à son environnement.
- *des opérateurs permettant de diversifier la population* au cours des générations et d'explorer en théorie la quasi-intégralité de l'espace d'état. L'opérateur de croisement recompose les gènes d'individus existant dans la population, l'opérateur de mutation a pour but de garantir l'exploration de l'espace d'états.
- *des paramètres de dimensionnement du problème*, tels que la taille de la population, le nombre total de générations ou critère d'arrêt de l'algorithme, les probabilités d'application des opérateurs de croisement et de mutation…

Le principe général du fonctionnement d'un algorithme génétique repose sur les étapes suivantes [83] :

(1) produire une population initiale d'individus aléatoires.
(2) exécuter itérativement les sous-étapes suivantes jusqu'à satisfaction du critère d'arrêt :
  (A) assignez une valeur d'aptitude à chaque individu de la population en utilisant la fonction d'évaluation.
  (B) Créez une nouvelle population de chromosomes en appliquant les opérations génétiques suivantes. Les opérations sont appliquées à des chromosomes choisis de la population avec une probabilité basée sur l'aptitude.
    (i) *Reproduction*: Reproduire un individu existant en le copiant dans la nouvelle population.
    (ii) *Croisement*: Créer deux nouveaux individus à partir de deux individus existants par recombinaison génétique de leurs chromosomes par l'opération crossover.
    (iii) *Mutation*: Créez un nouveau individu à partir d'un individu existant en subissant une mutation.
(3) l'individu qui est identifié par la méthode de désignation du résultat est retourné comme étant le meilleur chromosome produit.

On commence par générer une population d'individus de façon aléatoire. Pour passer d'une génération k à la génération k+1, les trois opérations de reproduction, croisement et mutation sont répétées pour des éléments de la population k. Des couples de parents $X_1$ et $X_2$ sont sélectionnés en fonction de leurs adaptations. L'opérateur de croisement leur est appliqué avec une probabilité $P_c$ et génère des couples d'enfants $E_1$ et $E_2$. D'autres éléments X sont sélectionnés en fonction de leur adaptation. L'opérateur de mutation leur est appliqué avec la probabilité $P_m$ et génère des individus mutés X'. Le niveau d'adaptation des enfants ($E_1$,$E_2$) et des individus mutés X' sont ensuite évalués avant insertion dans la nouvelle population. De génération en génération, la taille de la population reste constante. Lors d'une génération, la totalité d'une population peut être remplacée par ses descendants, comme c'est le cas de l'algorithme présenté ci-dessus, l'algorithme génétique est alors dit "*générationnel*" (*generational*). Ou au contraire, un petit nombre d'individus, un ou deux, sont remplacés, on

se trouve alors en présence d'un algorithme "*stationnaire*" (*steady state*). Différents critères d'arrêt de l'algorithme peuvent être choisis :

- Le nombre de générations que l'on souhaite exécuter peut être fixé à priori.
- L'algorithme peut être arrêté lorsque la population n'évolue plus.
- etc.

## 6.2.1. Codage des chromosomes

La première étape est de définir et de coder convenablement le problème. Historiquement le codage utilisé par les algorithmes génétiques était représenté sous forme de chaîne binaire contenant toute l'information nécessaire à la description d'un individu (chromosome). D'autres formes de codage sont possibles à savoir le codage réel, codage de gray, etc. Il existe deux types de difficultés dans le choix d'un codage. D'une part celui-ci doit pouvoir être adapté au problème de façon à limiter au mieux la taille de l'espace de recherche, et aussi de façon que les nouveaux chromosomes engendrés par les opérateurs de recherche soient significatifs le plus souvent possible, c'est à dire qu'ils puissent coder des solutions valides respectant les contraintes du problème.

**Définition 6.1**(*chromosome*) [24]:
soit A un alphabet (ensemble de symboles). Un chromosome X est une chaîne de symboles (gènes) qui appartiennent à l'alphabet A. le nombre de symboles dans X représente la longueur de X, notée par |X|. l'ensemble $A^l$ , $l \in N$, représente l'ensemble de tous les chromosomes X possibles avec |X|=l. X(i) dénote le $i^{ème}$ symbole, avec $0 \leq i < l$, du chromosome X.

**Définition 6.2**(*Codage Enc*, *Décodage Dec*) [24]:
soit (F, c) un problème de recherche. Soit $A^l$ un ensemble de chromosomes. La fonction surjective Dec : $A^l \rightarrow F$ est dite fonction de décodage. La fonction Enc : $F \rightarrow A^l$ est dite fonction de codage. Le codage Enc(f) d'un élément $f \in F$ est défini comme un élément de $\{X \in A^l \mid Dec(X)=f\}$. D'où, pour chaque élément $f \in F$, existe un ou plusieurs codages $X \in A^l$.

## 6.2.2. Génération de la population initiale

La rapidité de l'algorithme est fortement dépendante du choix de la population initiale d'individus. Si la position de l'optimum dans l'espace d'état est totalement inconnue, il est naturel et plus simple de générer aléatoirement des individus en faisant des tirages uniformes dans l'espace d'état en veillant à ce que les individus produits respectent les contraintes. Si par contre, des informations à priori sur le problème sont disponibles, les individus sont générés dans un sous-domaine particulier afin d'accélérer la convergence.

**Taille de la population**

Certaines approches utilisent une population à taille fixe, alors que d'autres proposent des tailles variables, qui évoluent au cours du temps. Au lieu d'utiliser une population de taille fixe où la fonction de mérite sert à la sélection, elles proposent plutôt d'indexer l'espérance de vie d'un individu à sa fonction de mérite. Les meilleurs individus reçoivent une espérance de vie supérieure à celle des moins bons. Au fur et à mesure que la population évolue, les individus vieillissent et finissent par mourir lorsqu'ils arrivent au terme de leur espérance de vie.

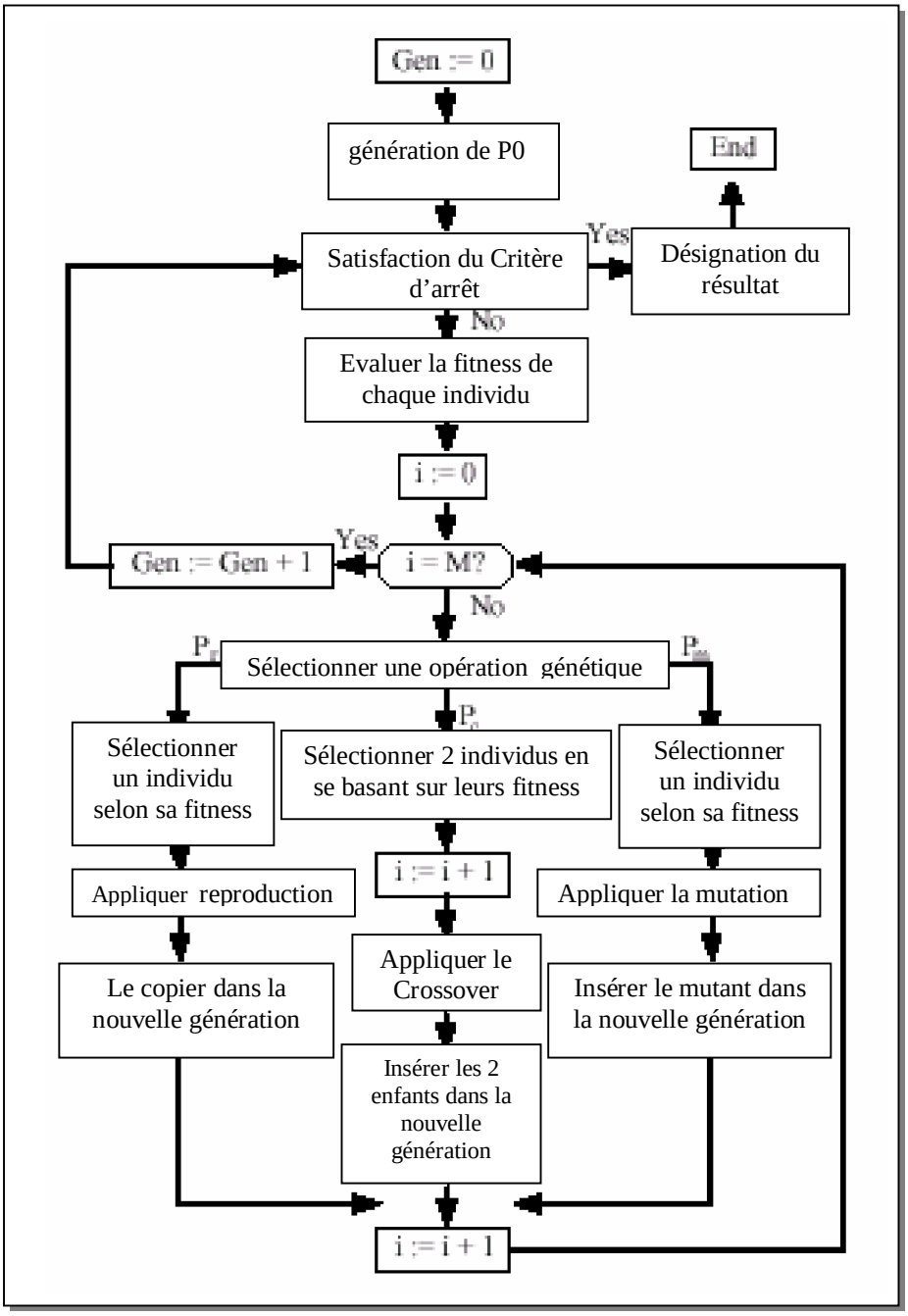



Figure 6.1 : Organigramme d'un algorithme génétique type (simple)
(Gen :numéro de la génération courante, M : taille d'une population)

## 6.2.3. Fonction de mérite (adaptation)

pour mieux concrétiser le processus d'évolution, il est nécessaire de pouvoir faire la distinction entre les chromosomes les *plus adaptés* et les *moins adaptés.* Ceci est possible par l'assignation d'une valeur d'adaptation à chaque chromosome.
La mise au point d'une bonne fonction d'adaptation (fitness function) doit respecter plusieurs critères qui se rapportent à sa complexité ainsi qu'à la satisfaction des contraintes du problème. Si le problème doit satisfaire des contraintes et que les chromosomes produits par les opérateurs de recherche codent des individus invalides, une solution parmi d'autres est d'attribuer une mauvaise *fitness* à l'élément qui a violé les contraintes afin de favoriser la reproduction des individus valides.

**Définition 6.3** (*Fitness s, fonction de poids* $\sum$ ) [24]:
soit (F, c) une instance d'un problème d'optimisation combinatoire. Soit $A^1$ un ensemble de chromosomes, et Dec : $A^1 \rightarrow F$ une fonction surjective. La fonction d'adaptation (fitness) $s : A^1 \rightarrow F$ est une fonction, avec s(X) l'adaptabilité du chromosome $X \in A^1$. La valeur d'adaptation s est attachée avec la fonction de coût c par l'utilisation d'une fonction de poids $\sum : R \rightarrow R$ tel que $s(X) = \sum (c(Dec(X)))$.

## 6.2.4. Sélection

Cet opérateur détermine la capacité de chaque individu à persister dans la population et à se diffuser. En règle général, la probabilité de survie d'un individu sera directement reliée à sa performance relative au sein de la population. Cela traduit bien l'idée de la sélection naturelle : les gènes les plus performants ont tendance à se diffuser dans la population tandis que ceux qui ont une performance relative plus faible ont tendance à disparaître. Donc, après avoir réaliser l'évaluation d'une génération, la population à un instant t, on opère une sélection à partir de la fonction d'adaptation. La sélection permet d'identifier statistiquement les meilleurs individus d'une population et d'éliminer les mauvais. Il existe plusieurs méthodes de sélection, parmi lesquelles on trouve :

- **Sélection par roulette biaisée**

La méthode RWS (Roulette Wheel Selection*)* consiste à associer à chaque individu de la population un segment dans la roulette. La largeur de ce segment est proportionnelle à sa fitness ou, plus précisément, à une probabilité d'être sélectionné proportionnelle à sa fitness. Donc à chaque individu X est associé une probabilité *sel(X)* tel que *sel(X)* = $s(X) / \sum s(x)$ (où :$x \in P$). La sélection d'un individu revient à choisir aléatoirement un point du segment avec une distribution de probabilité uniforme. Avec ce système, les grands segments, c'est-à-dire les bons individus, seront plus souvent adressés que les petits. Une certaine diversité est cependant maintenue, car même les individus les moins performants conservent une chance d'être choisis.

- **Sélection par tournoi**

la sélection par tournoi consiste à choisir aléatoirement un certain nombre d'individus, et à sélectionner pour la reproduction celui qui a la plus grande adaptation. Cette étape est répétée est autant de fois qu'il y a d'individus à remplacer dans la génération. Les individus qui participent à un tournoi restent dans la population et sont de nouveau disponibles pour les tournois ultérieurs.

- **Sélection par élitisme**

Du fait que le caractère aléatoire de la sélection ne garantit pas que le meilleur individu soit conservé, la méthode de sélection par élitisme surpasse cet inconvénient en recopiant automatiquement le meilleur individu de chaque génération directement dans la génération suivante. Le modèle élitiste, en fait, n'est pas une méthode de sélection en soi, mais plutôt une variante qu'on ajoute à d'autres méthodes.

- **Sélection par troncature**

Dans cette méthode, la sélection n'est pas laissée au hasard. Les individus sont triés selon leur fonction d'adaptation, ensuite sont choisit les T premiers individus de la génération pour générer la suivante. Cela revient en quelque sorte à généraliser le modèle élitiste à tout le processus de sélection. Le problème avec cette méthode est qu'on ne peut pas maintenir une diversité génétique suffisante dans la population.

## 6.2.5. Croisement

Le croisement (crossover) permet d'enrichir la diversité de la population, favorisant ainsi l'exploration de l'espace de recherche, en manipulant la structure des chromosomes. Chaque fois qu'un individu est retenu par le processus de sélection, on détermine aléatoirement s'il doit participer à un croisement (avec une probabilité Pc) ou être transmis directement à l'opérateur de mutation. Soit i un individu choisi par l'opérateur de sélection. Soit Ri un nombre aléatoire, $0 \leq Ri \leq 1$. On a:

- Si $Ri \leq Pc$ alors appliquer l'opérateur de croisement sur i
- Si Ri>Pc alors on peut pas appliquer l'opérateur de croisement sur i

Classiquement, les croisements sont envisagés avec deux parents ( X1 et X2) et génèrent deux enfants (E1 et E2). Cette combinaison des parents utilise la notion de *points de coupures* qui correspond aux points au niveau desquels s'effectuera l'échange de matériel génétique.

- **Croisement à un point**

Le croisement à un point est le croisement le plus simple. Pour effectuer ce type de croisement, on sélectionne aléatoirement un point de coupure k qui soit compris entre 1 et L-1 ( L est la longueur du chromosome) puis on subdivise le génotype de chacun des parents en deux parties de part et d'autre part de ce point. On échange ensuite les deux sous-chaînes terminales de chacun des deux chromosomes, ce qui produit deux enfants.

$$E1(i) = \begin{cases} X1(i) \text{ si i appartient à } [1,k] \\ X2(i) \text{ si i appartient à } [k+1,l] \end{cases} \qquad E2(i) = \begin{cases} X2(i) \text{ si i appartient à } [1,k] \\ X1(i) \text{ si i appartient à } [k+1,l] \end{cases}$$

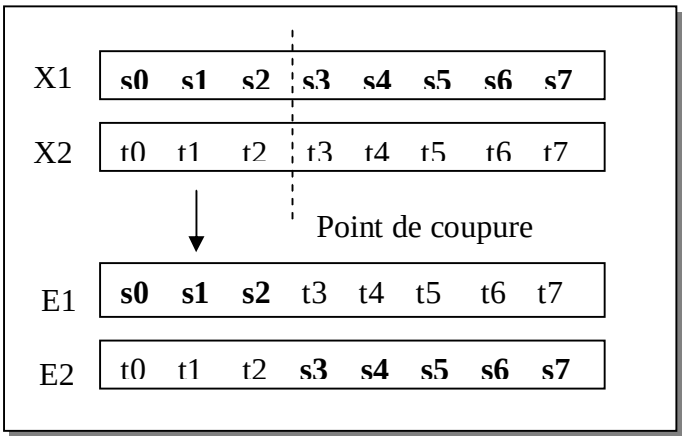



Figure 6.2 : Opérateur de croisement à un seul point de coupure

- **Croisement multipoints**

Ce type de croisement peut être vu comme une généralisation du croisement à un point, en découpant le chromosome non pas en 2 sous-chaînes mais en k sous-chaînes.

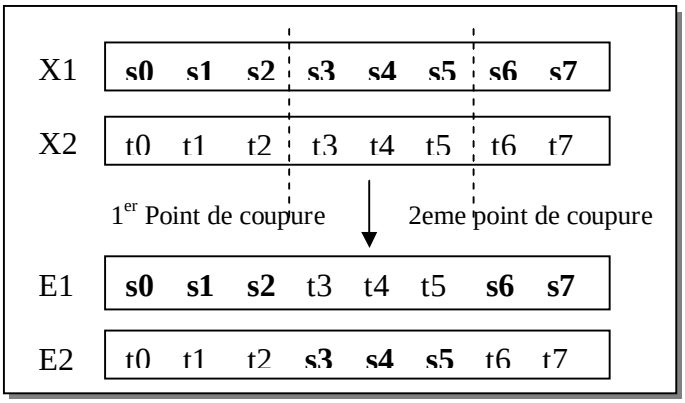



Figure 6.3 : Opérateur de croisement à deux points de coupure

- **Croisement uniforme**

Le croisement uniforme peut être vu comme un croisement multi-points dont le nombre de coupures est indéterminé a priori. Pratiquement on utilise un "masque de croisement", engendré aléatoirement pour chaque couple d'individus, qui est un mot binaire de même longueur que les chromosomes. Un "0" à la n-ième position du masque laisse inchangé les symboles à la n-ième position des deux génotypes, un "1" déclenche un échange des symboles correspondants.

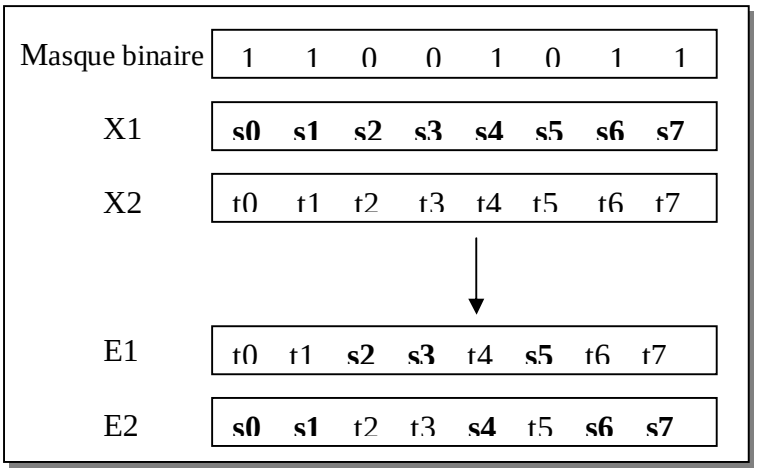



Figure 6.4 : Opérateur de croisement uniforme

## 6.2.6. Mutation

Classiquement, l'opérateur de mutation modifie aléatoirement les symboles d'un chromosome. La mutation est traditionnellement considérée comme un opérateur intervenant à la marge dans la mesure où sa probabilité est en général fixée assez faible (de l'ordre de 1%). Mais elle confère aux algorithmes génétiques une propriété très importante où tous les points de l'espace de recherche peuvent être atteints. Cet opérateur est donc d'une grande importance et il est loin d'être marginal et permet d'assurer une recherche aussi bien globale que locale, selon le poids et le nombre des bits mutés. Ainsi, les mutations garantissent mathématiquement que l'optimum global peut être atteint. Si Pm est la probabilité de mutation, $0 \leq r \leq 1$ un nombre tiré au hasard et que l'on ait $r < Pm$ alors le chromosome va subir une mutation. Il existe aussi une variante où plusieurs bits peuvent muter au sein d'un même chromosome.

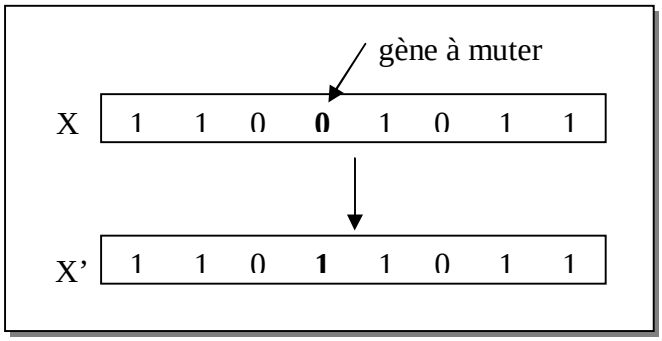



Figure 6.5 : Opérateur de mutation

## 6.3. Les Algorithmes Génétiques au service d'autres algorithmes

Les Algorithmes Génétiques sont donc des outils simples et efficaces pour explorer un espace de performances donné. A ce titre, ils sont souvent utilisés pour faire fonctionner ou pour affiner d'autres algorithmes. Nous devons à leur inventeur la première de ce type d'utilisation : John Holland. En effet, cet auteur a mis les AG au cœur des *Systèmes Classificateurs* **(SC** – aussi appelés *learning classifier system* - LCS) qu'il propose comme modèle d'intelligence artificielle [88]. Les SC complètent la variable qui fait l'objet de recherche par des conditions quant à l'état de l'environnement. L'algorithme cherche alors à développer des solutions adaptées à chaque configuration différente de cet environnement. L'objet de l'évolution est alors les règles formées de deux composantes : [*CONDITIONS* | ACTION]. Ces règles ressemblent par conséquent à des éléments de raisonnement de type [*S'il y a des nuages dans le ciel* et *si je dois sortir* | prendre parapluie].

Par ailleurs, la puissance calculatoire des algorithmes génétiques a été mis au service d'autres algorithmes pour les accélérer ou pour les affiner. Ainsi les AG ont été utilisés pour faire évoluer la structure des automates cellulaires ou les poids des connections dans un réseau neuronal artificiel.

A l'inverse des méthodes traditionnelles de résolutions numériques, les algorithmes génétiques ne sont pas fondés sur une approche analytique mais sur une approche itérative et heuristique. En cela, peu d'information est nécessaire pour leur utilisation : l'espace de recherche possible et un critère d'efficacité. Les ingénieurs et les spécialistes en recherche opérationnelle ont très vite vu le potentiel de leur utilisation comme outil de résolution numérique. Il s'en est suivi, depuis le début des années 90, une liste importante d'articles sur le sujet comparant les algorithmes génétiques à d'autres méthodes de résolution numérique, qu'elles soient analytiques ou aléatoires. Toutes ces comparaisons concluent sur l'efficacité des algorithmes génétiques [88].

## 6.4. Algorithme GPBS (**G**enetic **P**ath **B**ased **S**cheduling**)**

Les algorithmes génétiques sont basés sur une idée simple où l'évolution semble être un moyen efficace pour favoriser l'émergence de solutions adaptées à un environnement complexe. Motivé par la simplicité et la robustesse des algorithmes génétiques [87], est apparus une grande orientation vers la résolution des problèmes d'optimisation par ces algorithmes. Plusieurs travaux basés sur les algorithmes génétiques ont été menés dans le domaine de l'ordonnancement. Les études recensées dans la littérature traitent majoritairement du cas de l'ordonnancement des circuits dominés par le flot de données [24,71,72,84,85,86], tandis que les algorithmes d'ordonnancement des circuits de contrôle à base d'algorithmes génétiques sont minimes (presque inexistants).

Dans ce chapitre on propose une nouvelle technique d'ordonnancement des circuits dominés par le flot de contrôle (**algorithme GPBS**), qui consiste à utiliser un algorithme génétique en conjonction avec l'algorithme d'ordonnancement à base de chemin PBS pour construire un nouvel algorithme d'ordonnancement qui améliore la qualité de la solution retournée.

L'usage d'une population de solutions, plutôt que d'une seule solution, est un trait essentiel et intéressant des algorithmes génétiques comme il va accélérer la recherche et assurer la convergence vers de bonnes solutions, qui peuvent même être des solutions optimales. En outre, il aide à prévenir une convergence à des solutions sub-optimales dans les problèmes à espace complexe. Aussi, les algorithmes génétiques n'ont pas besoin de pré-connaissance spéciale au sujet de l'espace du problème à traiter. Ce point considéré comme un avantage et caractéristique marquante des algorithmes génétiques, peut aussi en quelque sorte relativement retarder la convergence. Donc coupler les algorithmes génétiques avec d'autres algorithmes propres au domaine à traiter va accélérer la convergence et améliorer considérablement la qualité de la solution obtenue.

Comme on veut développer une technique d'ordonnancement orienté flot de contrôle, on a opté pour l'algorithme d'ordonnancement à base de chemins pour le coupler avec un algorithme génétique. L'algorithme à base de chemins est à la source de la majorité des algorithmes d'ordonnancement des circuits de contrôle, et il est bien adapté aux circuits orientés flot de contrôle. A lui seul il peut fournir dans certains cas des solutions optimales, alors comment si on le couple avec un algorithme génétique.

Donc, cette nouvelle technique va tirer profit de l'adaptation de l'algorithme PBS à l'ordonnancement des circuits orientés flot de contrôle, ainsi que de la puissance d'exploration de l'espace offerte par les algorithmes génétiques.

Comme l'algorithme d'ordonnancement à base de chemins (PBS) représente la base de l'approche qu'on propose dans ce chapitre, c'est important de rappeler les principaux concepts de cet algorithme.

## 6.4.1. Algorithme d'ordonnancement à base de chemins PBS

L'ordonnancement à base de chemins est une technique qui analyse tous les chemins du CFG (**C**ontrol **F**low **G**raph) et effectue l'ordonnancement de chaque chemin indépendamment, donc minimise le nombre d'étapes de contrôle nécessaires pour exécuter chaque chemin dans le CFG. Les chemins dans le CFG surviennent d'opérations conditionnelles et de boucles, et manier tel type d'opérations est une partie intégrante de l'approche.

L'ordonnancement est basé sur la résolution de contraintes sur les chemins. Ces contraintes sont des restrictions générales sur l'ordonnancement, et peuvent être dû aux ressources, délai, ou toute autre mesure de coût (contraintes extrinsèques et intrinsèques). Les seuls éléments dans la formulation de l'algorithme sont les chemins du flot de contrôle et les contraintes. Les contraintes sont représentées par des intervalles dans les chemins du flot de contrôle, comme montré dans la Figure 6.6. Le CFG dans la Figure 6.6(a) contient une opération conditionnelle qui résulte en deux chemins (Figure 6.6(b)). Supposons qu'il y a trois contraintes imposées sur l'ordonnancement:

1) le design devrait contenir seulement un additionneur,
2) seulement un soustracteur,
3) et le temps du cycle ne devrait pas dépasser 15 ns.

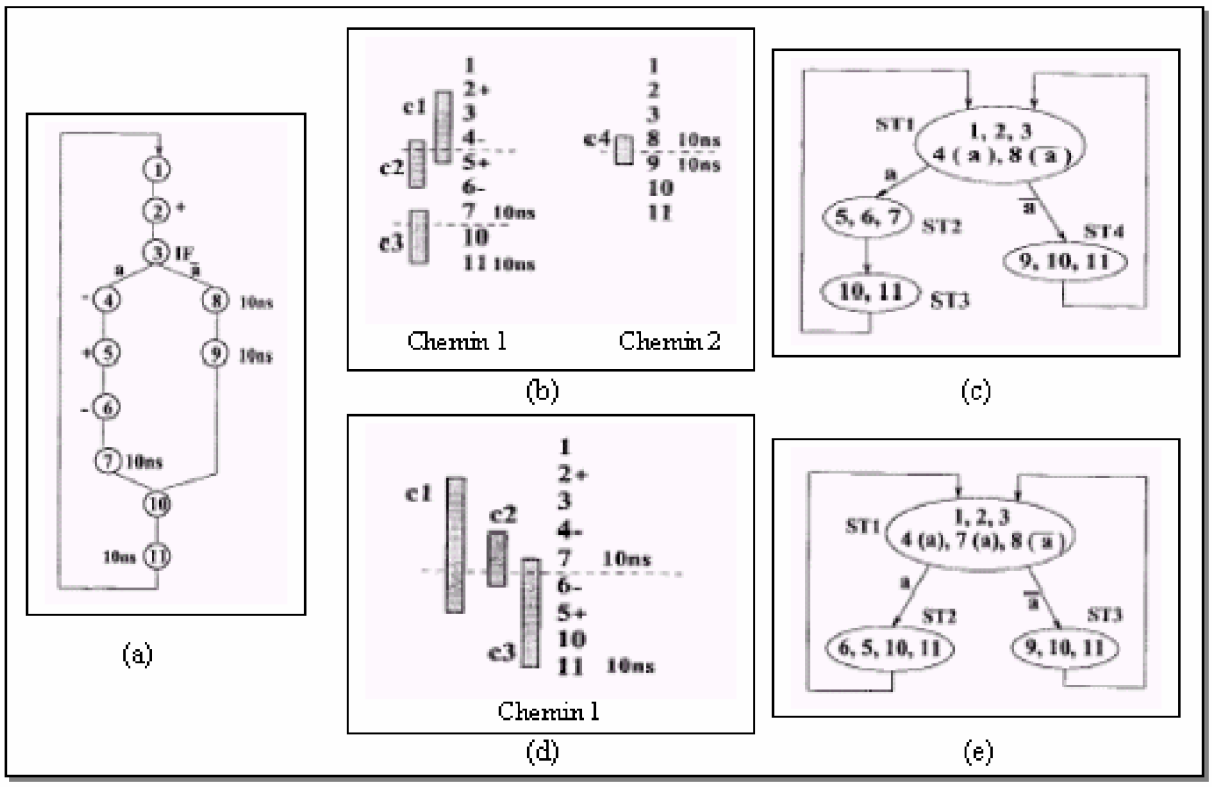



Figure 6.6 : Exemple d'un ordonnancement à base de chemins, (a) : CFG, (b) Les chemins et les contraintes, (c) Machine d'états finis, (d) Chemin1 et contraintes après réordonnement des opérations 5,6 et 7, (e) FSM après réordonnement.

Les opérations dans chaque chemin sont vérifiées et un intervalle de la contrainte est créé la où une violation est possible. Par exemple, le chemin 1contient deux additions et deux soustractions ce qui dépasse les contraintes 1) et 2). Donc, les intervalles de contraintes sont créés entre les opérations qui causent les violations de contraintes. Pour satisfaire la contrainte d'un seul additionneur, les deux opérations d'addition (2 et 5) ne peuvent pas être ordonnancées dans la même étape de contrôle, d'où un intervalle de contrainte (IC1) est créé de l'opération 2 à 5 (comme on peut utiliser un intervalle de 3 à 5 comme on l'a décrit dans la section 5.4.1) . Cet intervalle indique à l'algorithme d'ordonnancement que la séquence d'opérations entre 2 et 5 doit être coupée en au moins deux étapes de contrôle, tel que les opérations 2 et 5 finissent dans des étapes de contrôle différentes (ce qui permet à l'additionneur d'être partagé). La contrainte d'un seul soustracteur résulte en l'intervalle de contrainte IC2 entre les opérations 4 et 6; et la contrainte du temps du cycle 15 ns résulte en les intervalles IC3 et IC4 (on suppose qu'il y a dépendance de données entre les opérations 7-11 et 8-9 et le délai du chaînage des opérations 7–11 et 8–9 dépasse 15 ns).
Les coupes sont représentées comme lignes horizontales pointillées dans la Figure 6.6(b). La figure 6.6(c) montre la machine d'états finis finale (FSM) et l'ordonnancement des opérations du CFG suivant l'ensemble de contraintes donnés dans la figure 6.6(a) et (b). La séquence d'opérations dans le chemin 1 nécessite 3 étapes de contrôle pour son exécution, cependant, le chemin 2 peut être ordonnancé dans deux étapes de contrôle.

L'algorithme d'ordonnancement à base de chemins est très adéquat pour l'ordonnancement des circuits de contrôle et il présente de grands avantages, dont on peut citer :
- *Formulation Exacte avec des contraintes générales:* l'algorithme PBS modèle le problème d'ordonnancement comme un problème de couverture en cliques. Dans presque tous les cas,

les solutions optimales et exactes peuvent être trouvées. Les contraintes dans la formulation à base de chemins sont complètement générales.
- *Optimisation de chaque chemin de contrôle:* le nombre minimum d'étapes de contrôle est obtenu pour chaque chemin dans un CFG donné suivant certaines contraintes. Cela optimise la performance du circuit entier en terme du nombre de cycles nécessaires pour compléter toute séquence d'opérations. Cela peut exiger que quelques opérations soient ordonnancées dans de multiples états (e.g., opérations 10 et 11 dans la Figure 6.6(c) dans les états 3 et 4).
- *Opérations Conditionnelles:* les opérations conditionnelles et les boucles sont utilisées pour la dérivation des chemins de contrôle. Une fois les chemins sont formés, l'algorithme PBS traite les opérations conditionnelles comme toute autre opération.

Cependant, et malgré ses avantages, l'algorithme PBS a aussi des limitations qui peuvent influencer la qualité de l'ordonnancement finale, entre autres, l'ordre fixe des opérations qui limite le parallélisme. PBS ne change pas l'ordre des opérations dans un chemin. Les intervalles de contraintes sont créés pour un ordre fixe des opérations. Dans certains cas, en réordonnant les opérations il peut être possible d'ordonnancer un chemin en moins d'étapes de contrôle. Par exemple, considérons le CFG dans la Figure 6.6(a), et supposons qu'il n'y a pas de dépendances de données entre les opérations 5, 6, et 7. En réordonnant ces opérations comme montré sur la figure 6.6(d), les intervalles de contraintes s'étendent et se chevauchent l'un avec l'autre, ce qui permet de satisfaire toutes les contraintes dans le chemin 1 avec une seule coupe. En conséquence, on aura un état en moins dans la machine d'états finis et le chemin 1 peut être exécuté en deux étapes de contrôle seulement, comme le montre la figure 6.6(e).

## 6.4.2. Combinaison de l'algorithme PBS avec un Algorithme Génétique

Vu les points forts et les handicapes de l'algorithme PBS ainsi que les algorithmes génétiques, et afin de permettre au système d'ordonnancement d'aboutir à des temps d'exécution minimes, il est nécessaire de combiner les meilleurs caractéristiques de ces deux approches. La nouvelle technique d'ordonnancement décrite dans ce mémoire accomplit ces objectifs en intégrant un algorithme génétique dans un algorithme d'ordonnancement à base de chemins, ce qui donne comme résultat un système avec les caractéristiques suivantes :

1) il extrait les chemins d'exécution possibles du CFG et traite les opérations conditionnelles comme toute autre opération.
2) il utilise les contraintes et les approches de couverture en cliques (comme dans PBS) pour obtenir une solution exacte pour un CFG et des contraintes données.
3) il réordonne les opérations du CFG, par le biais d'un algorithme génétique, afin d'augmenter le parallélisme et maximiser les chevauchements de contraintes.

Les deux premiers points sont similaires à l'algorithme PBS ordinaire. Alors que le troisième point est propre à notre approche et sera décrit en détails dans les sections suivantes.

### 6.4.2.1. Le réordonnement et son effet dans une approche à base de chemins

PBS considère une séquence fixe d'opérations suivant laquelle sont construits les intervalles de contraintes et d'où les coupures et les étapes de contrôle sont dérivés. Pour un ordre fixe d'opérations donné, l'algorithme trouve une solution optimale, cependant, comme on la vu dans l'exemple de la figure 6.6, il peut y avoir un classement différent qui produit des intervalles de contraintes différents et moins d'étapes de contrôle.

Un chemin peut contenir plusieurs opérations conditionnelles et plusieurs blocs de base[1]. Chaque bloc de base, peut appartenir à plusieurs chemins. Le problème du réordonnement est qu'il affecte tous les chemins qui contiennent les opérations réordonnées et un ordre qui est bénéfique dans un chemin peut être inefficace dans un autre chemin. Par exemple, considérons le CFG, les chemins et les contraintes de la figure Figure 6.7(a) et (b). Le chemin 1 contient deux contraintes qui ne se chevauchent pas et a besoin de 2 coupures, 3 étapes de contrôle; alors que le chemin 2 a 2 contraintes qui se chevauchent et a besoin de 1 coupure, 2 étapes de contrôle. Si le bloc de base 6-7-8-9 est réordonné en 8-7-6-9, le chemin 1 aura 2 contraintes en chevauchement , 1 coupure, 2 étapes de contrôle, mais le chemin 2 devient non optimal, nécessitant 2 coupures et 3 étapes de contrôle (Figure 6.7(c)).

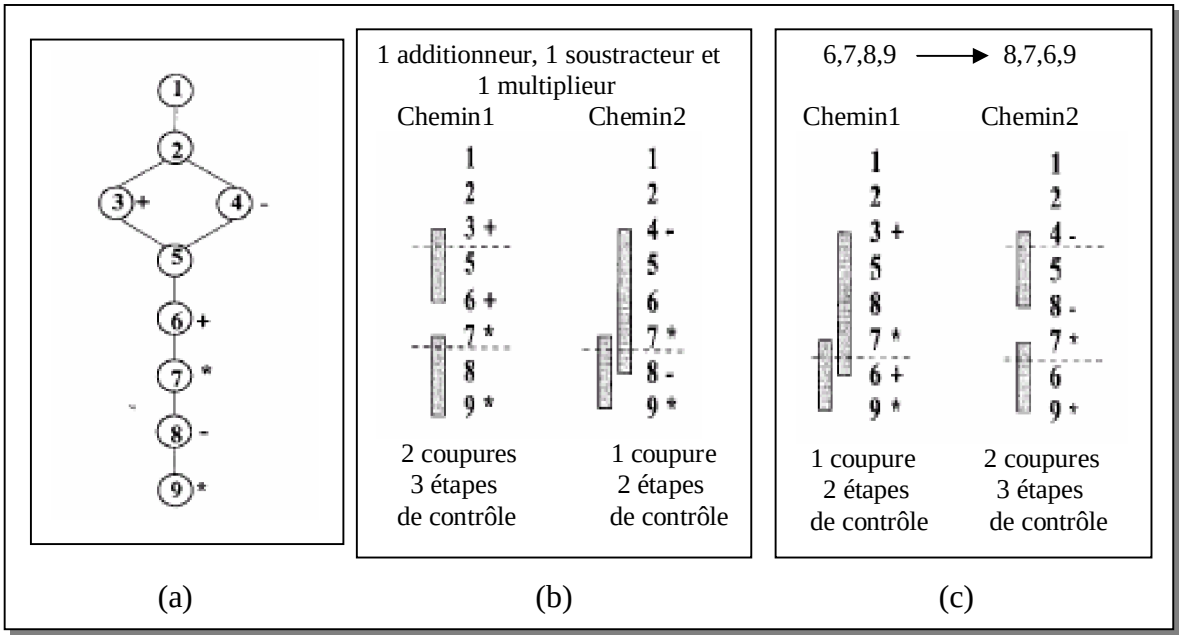



Figure 6.7: Effet du réordonnement sur l'ensemble des chemins.
(a) CFG, (b) Chemins et contraintes avec un additionneur, un soustracteur et un multiplieur, (c) Chemins et contraintes aprés réordonnement.

En général, il est peut être impossible de trouver un réordonnement qui produit des résultats optimaux dans tous les chemins (bien que dans ce cas, l'ordre 7-6-8-9 fût optimal). Cependant, intuitivement, plus les intervalles de contraintes sont larges, plus ils ont tendance à être en chevauchement, ce qui donne en général moins de coupures et moins d'étapes de contrôle.

L'approche de réordonnement présentée dans cette section suit les critères ci-dessous :

1) Réordonner les opérations de telle façon à élargir les intervalles de contraintes et maximiser leurs chevauchement.
2) Réordonner les opérations de telle façon à permettre plus de parallélisme ce qui peut réduire et minimiser le délai.
3) Réordonner les opérations à l'intérieur de blocs de base seulement (pas en travers des opérations conditionnelles).

Pour résoudre le problème de réordonnement, un algorithme génétique est appliqué, utilisant comme fonction de coût une combinaison des critères 1) et 2) expliqués ci-dessus.

Les intervalles de contraintes peuvent être le résultat d'opérations appartenant à des blocs de base différents, aussi bien qu'opérations à l'intérieur d'un bloc de base. La

[1] Un bloc de base est une séquence d'instructions qui ne contient aucune instruction de branchement.

figure6.8(a) montre des exemples d'intervalles de contraintes qui se terminent dans un bloc de base (IC1), contenus dans un bloc de base (IC2) et commençant dans un bloc de base (IC3). Comme l'algorithme de réordonnement travaille seulement à l'intérieur de bloc de base, il est nécessaire de prendre en considération les opérations au-dessus et en-dessous du bloc de base qui peuvent causer le début ou la terminaison d'intervalles de contraintes à l'intérieur du bloc principale. Il faut noter que les contraintes qui commencent au-dessus, ou se terminent en dessous, du bloc de base ne sont en rapport avec aucune opération à l'intérieur du bloc de base et donc elles ne sont pas affectées par le réordonnement du bloc de base.

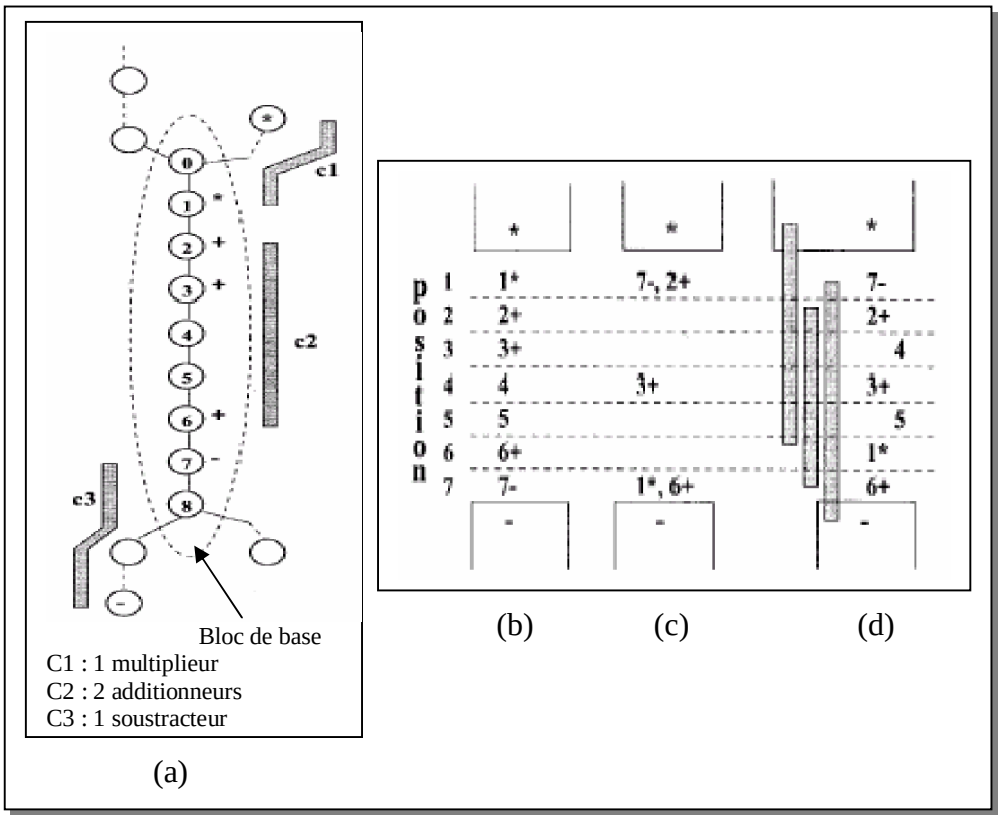



Figure 6.8 : Algorithme de réordonnement. (a) CFG initial, (b) Bloc de base et opérations génératrices de contraintes, (c) Positions cibles pour le réordonnement, (d) Bloc de base final réordonné.

Nous allons en premier considérer un exemple du fonctionnement de l'algorithme de réordonnement. La figure 6.8(a) montre une partie d'un CFG avec les contraintes, nous allons voir comment le bloc de base doit être réordonné pour maximiser les chevauchements de contraintes. Sans réordonnement, le chemin avec les contraintes C1, C2 et C3 nécessite 3 coupures et 4 étapes de contrôle.

Afin d'élargire un intervalle de contrainte, les opérations qui causent la contrainte devraient être espacées autant que possible. Si une contrainte commence ou se termine à l'extérieur du bloc de base (tel que les contraintes C1 et C3 dans la Figure 6.8(a)) alors l'opération commençant, ou terminant, la contrainte peut être considérée fixe au sommet, respectivement au fond, du bloc de base. Fondamentalement, toutes les opérations génératrices de contrainte au-dessus du bloc de base sont considérées comme un nœud fixe au sommet du bloc de base. De la même façon, toutes les opérations génératrices de contrainte en-dessous du bloc de base sont considérées comme un nœud fixe au fond du bloc de base. La figure 6.8(b) montre le bloc de base avec les nœuds compressées au-dessus et en-dessous.

Afin d'espacer en dehors les opérations génératrices de contraintes il serait meilleur de 1) pour la contrainte C1, l'opération 1 devrait être placée en dessous aussi loin que possible, 2) pour la contrainte C3, l'opération 7 devrait être placée au-dessus aussi loin que possible, et 3) pour la contrainte C2, les opérations 2, 3, et 6 devraient être également espacées dans le bloc de base.

Les positions cibles des opérations dans le bloc de base sont montrées dans la figure 6.8(c). Les positions définitives dans cet exemple sont données dans la figure 6.8(d), qui exigent seulement 1 coupure et 2 étapes de contrôle. Dans cet exemple, il a été supposé que les opérations n'avaient pas de dépendances de données et qu'elles pouvaient être déplacées librement. Dans la réalité, les dépendances de données doivent être prises en considération, ce qui peut limiter les possibilités de réordonnement.

Ces concepts sont la base de notre algorithme de réordonnement , appelé **GPBS**, dont les principales étapes sont données ci-dessous.

## 6.4.3. Différentes étapes de l'algorithme GPBS

### 1 - Traitement des boucles

comme on l'a déjà préciser dans le chapitre précédent, l'algorithme à base de chemin commence par une étape de prétraitement qui consiste en l'élimination des arcs de retour des boucles. La première opération de la boucle, ainsi que la dernière et la condition de rebouclage sont stockées. Ainsi le CFG devient un graphe orienté acyclique.

### 2 - Définition des chemins

Extraire tous les chemins d'exécution possibles du CFG. Un chemin commence soit avec le premier nœud du graphe ou le premier nœud d'une boucle.

### 3 – Définition des intervalles de coupure

Traverser le CFG et pour chaque bloc de base identifier toutes les opérations génératrices de contraintes à travers, au-dessus et en dessous du bloc de base, de la même manière que dans l'algorithme PBS ordinaire. Une contrainte est définie par paires d'instructions ne pouvant pas s'ordonnancer dans la même étape de contrôle. Dans la figure6.8, cette étape identifie la multiplication et la soustraction comme types d'opération génératrices de contraintes au-dessus et en dessous du bloc de base, respectivement.

Donc, traverser le CFG et pour chaque bloc de base (nommé BB) appliquer les étapes de traitement suivantes :

a) Associer un niveau à chaque opération dans le bloc de base en se basant sur les dépendances de données entre les opérations. Par exemple, une opération dont les entrées sont des entrées ou des variables créées à l'extérieur du bloc de base aura le niveau 0. Une opération dont les entrées sont créées par les nœuds de niveaux aura un niveau égal à Max(x,y)+1.

b) Trier les opérations dans un ordre croissant suivant leurs niveaux. Cela regroupe ensemble les opérations de même niveau, qui peuvent être exécutées en parallèle. Le nouveau bloc de base trié selon les niveaux est appelé BB_Trié.

c) Parcourir les opérations dans le nouveau bloc de base trié BB_Trié et créer deux listes: une liste d'opérations génératrices de contraintes (Op_GC) et une liste d'opérations non génératrices de contraintes (Op_NGC). Par construction, ces listes sont aussi triées par niveau.

d) Suivant les contraintes d'entrée, calculer les positions cibles (PC(op)) pour chaque opération génératrice de contrainte qui appartient à la liste Op_GC. Ces positions cibles sont calculées par espacement des opérations associées à une contrainte donnée tout au long du bloc de base (inclure toutes les opérations externes arrangées au sommet ou au fond du BB ). Dans la figure 6.8(c), les positions cibles pour toutes les opérations génératrices de contraintes sont :
PC(2) = PC(7) = 1, PC(3) = 4 , PC(1) = PC(6) = 7.

**4 – Réordonnement des blocs de base par l'algorithme génétique**

Réordonner les opérations de chaque bloc de base en utilisant un algorithme génétique dont la fonction de coût repose sur la distance entre la position courante de l'opération et sa position cible (pour les opérations génératrices de contraintes), ainsi que sur le respect des dépendances de données. La fonction de coût utilisée dépend de deux facteurs. Pour un nœud donné $op_i$, son coût dépend du suivant :

1) La distance entre la position courante et la position cible de l'opération ($PC(op_i)$ – position_courante($op_i$)). Plus on minimise la distance, plus c'est meilleur pour l'élargissement des intervalles de contraintes donc plus on minimise le coût. Cette distance est appelée dist_to_PC ($op_i$, position).
2) Le respect des niveaux (niveau ($op_i$)). Plus on respecte les dépendances de données plus on minimise le coût ainsi on augmente le parallélisme.

L'algorithme de réordonnement d'opérations dans un bloc de base par l'algorithme génétique est montré dans la figure 6.12.

**5 – Utilisation de l'algorithme de partitionnement en cliques**

Le problème de minimisation du nombre d'étapes de contrôle est transformé en un problème de partitionnement en cliques du graphe(clique-partitionning graph), qu'on applique à l'ensemble des chemins avec les nouveaux blocs de base.

**6 – Construction de la machine d'états finis**

L'étape finale est la construction de la machine d'états finis réalisant le comportement du graphe de flot de contrôle initial. Tous les chemins ordonnés ayant la même tête auront le même état initial et tous les chemins ayant le même successeur auront le même état final.

**Algorithme GPBS ;**

$N_P$ = Nombre de chromosomes de la population
$N_G$ = Nombre de générations
$N_E$ = Nombre de chromosomes enfants
$P_M$ = Probabilité de mutation

```
Début
Trait_Boucles(CFG);              // Procédure pour le traitement des boucles
Definition_Chemins();            // Définition des chemins d'exécution possibles
Définition_Intevalles();         // Définition des intervalles de coupure, et pour chaque
                                 // bloc de base extraire les listes Op_GC et Op_NGC et
                                 // associer à chaque opération un niveau de dépendance
Pour k=1 jusqu'à Nbre_BB faire // Appliquer l'algorithme génétique à chaque Bloc de base
  Début
      P = Init_Population(BB[k]) ;          // cette procédure génère la population initiale
      Calcul_Coût(P) ;          // Calcul_Coût() est utilisée pour calculer le coût de
                                // chaque chromosome de la population, selon l'équation 6.1
      Evaluation_Fitness(P);    // Cette procedure est utilisée pour calculer la valeur
                                // d'adaptation de chaque chromosome dans la population,
                                // elle est basée sur l'équation 6.2
      Pour  i = 1 to N_G  Faire
      Début
            Pour j = 1 to N_E Faire              // On génère N_E enfants
            Début
             X = Selection_Parents(P) ;          // Dans notre méthode on utilise un seul
                                // chromosome pour la génération d'un  chromosome enfant
             E_j = Crossover (X);                // un chromosome enfant E_j est généré
            Fin Pour ;

            Pour j = 1 to N_E Faire              // Application de l'opérateur de mutation
            Début
             x= random();                        // Générer un nombre aléatoire
             if (x < P_M) then Mutation (E_j);   // Si le nombre aléatoire est inférieure à la
                                // probabilité de Mutation, on applique l'opérateur de
                                // Mutation au chromosome enfant E_j
            Fin Pour ;

            Calcul_Cost(E);                      // Estimation des coûts des chromosomes enfants
            P = Selection_NouvelleGénération(P,E) ;   // Sélection de la population de la
                        // nouvelle génération à partir des chromosomes de la population
                        // courante et des chromosomes enfants
            Evaluation_Fitness(P);
      Fin Pour;
      Retourner le meilleur chromosome dans P
    Fin Pour ;
Clique_Partitionning(solution) ;  // Appliquer l'algorithme de partitionnement en cliques aux
                        //nouveaux chemins générés après réordonnement des blocs de base
Construct_FSM() ;                                // Construction de la machine d'états finis
Fin Algorithme ;
```

Figure 6.12 : Algorithme GPBS

## 6.4.4. Définition de l'algorithme génétique et de ses différents opérateurs

Notre algorithme génétique est caractérisé par un ensemble de procédures (codage des chromosomes, fonction de coût, croisement,…) qu'on détaille dans ce qui suit :

### 1- Le codage

Il consiste à représenter chacune des solutions possibles par un ensemble de une ou plusieurs chaînes de longueur finie appelée chromosome; chaque gène contient un code représentant une variable du système. Dans notre cas, chaque chromosome est le codage d'une séquence d'opérations d'un bloc de base. Deux gènes successifs d'un chromosome représentent deux numéros d'opérations adjacentes dans l'ordre proposé. La taille d'un chromosome est égale à N tel que N est le nombre d'opérations dans la séquence d'opérations du bloc de base qu'on veut réordonner.

Par exemple, si on veut coder le bloc de base de la figure 6.8(b): BB[1]=1-2-3-4-5-6-7. Le chromosome représentant ce bloc de base aura la forme suivante :

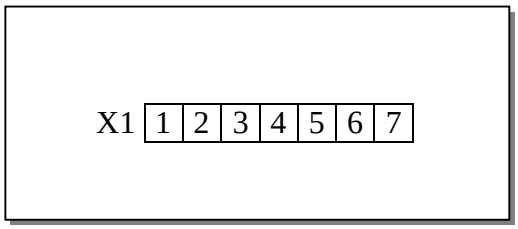



Figure 6.9 : Codage du chromosome

### 2- La population initiale

la première génération d'individus peut être construite de trois façons : soit de manière aléatoire, soit par l'utilisateur, soit à la fois en partie donné et en partie créée.
Notre population initiale se compose de deux types de solutions. Celles issues de l'application de l'algorithme d'ordonnancement à base de chemins (un des blocs de base générés à l'étape 2 de GPBS) et celles générées aléatoirement. Pour la génération aléatoire des solutions restantes, on doit respecter que:

- Chaque solution doit comprendre les mêmes opérations du bloc de base qu'on a choisi de réordonner dans un ordre différent.

### 3- Fonction d'évaluation et estimation du coût

Une fonction d'adaptation (fitness function) est utilisée pour calculer les valeurs d'adaptation ou la force (fitness value) des chromosomes de la population. Une valeur fitness d'un chromosome exprime la qualité de la solution que le chromosome représente. Les chromosomes avec une plus grande valeur d'adaptation représentent des solutions de haute qualité.

Cette fonction caractérise le problème à résoudre et établit la base pour la sélection. Dans la plupart des cas, la fonction d'adaptation peut être assimilée à la fonction objective d'un problème d'optimisation classique. Elle inclura aussi des amendes pour les contraintes violées. Si l'objectif du problème d'optimisation est de minimiser le coût, les chromosomes à coût faible auront une meilleure valeur d'adaptation.

On réordonne les opérations de chaque bloc de base en utilisant une fonction de coût qui repose, comme on l'a dit précédemment, sur la distance entre la position courante de l'opération et sa position cible (pour les opérations génératrices de contraintes), ainsi que sur

le respect des dépendances de données. La fonction de coût utilisée dépend de deux facteurs. Pour un nœud donné $op_i$, son coût dépend du suivant :

1) Plus on minimise la distance, plus c'est meilleur pour l'élargissement des intervalles de contraintes donc plus on minimise le coût. Cette distance est appelée dist_to_PC ($op_i$, position).
2) Le respect des niveaux (niveau ($op_i$)). Plus on respecte les dépendances de données plus on minimise le coût ainsi on augmente le parallélisme.

$$\textbf{Coût (X)} = \sum_{opi} \textbf{(}\alpha \textbf{ * | dist\_to\_PC(op}_i\textbf{ ,position}_i\textbf{) |) + (}\ \beta \textbf{ * K )  / op}_i \in \textbf{(Op\_GC)} \qquad \textbf{(6.1)}$$

Où $\alpha < 0$,

$\beta$ est un paramètre de pénalisation des opérations qui ne respectent pas les dépendances de données. K est le nombre d'opérations qui n'ont pas respecté ce critère. A chaque fois qu'on a une paire d'opérations adjacentes ($op_i$ , $op_j$) tel que niveau($op_i$) > niveau($op_j$), on incrémente K de 1.

Après estimation du coût, une fonction d'adaptation simple (fitness function) peut être exprimée par la formule suivante :

$$\textbf{F(X)} = \frac{1}{Coût(X)} \qquad \textbf{(6.2)}$$

**4 - La sélection**

Elle consiste à choisir les individus à partir desquels va être obtenue la génération suivante ou les chromosomes enfants. La sélection des parents est basée sur la valeur *fitness* des chromosomes. Nous utilisons pour la sélection la roulette biaisée de Golberg. Pour chaque individu, la probabilité de sélection est calculée (rapport de la fitness de l'individu sur la somme des fitness de l'ensemble de la population).

$$P_s(X) = F(X) / \sum F(x) \text{ (où :} x \in P)$$

La méthode de la roulette biaisée permet de choisir les paires de chromosomes qui seront utilisées pour appliquer l'opérateur Crossover. Le choix est un processus probabilistique. Les chromosomes avec les grandes valeurs fitness sont les plus probables à être choisis que ceux qui ont des valeurs fitness faibles. Le code de l'algorithme de la roulette biaisée est le suivant :

```
Algorithme Roulette-Biaisée ;
  Début
        i=0 ;                    // Le i^eme chromosome de la population
        accumulation =0 ;
        r = random (0,1) ;   // r: nombre aléatoire entre 0 et 1 généré
                             // par la fonction random
        Tant que (accumulation < r) faire
        début
              i=i+1;
              accumulation = accumulation + Ps(i) ; // Ps(i) : probabilité de sélection
                                                    // du chromosome i
        fin TQ ;
        retourner i ;
  Fin Roulette-Biaisée ;
```

### 5- Le croisement

Par cette opération, deux individus sélectionnés échangent une partie de leurs gènes pour donner naissance à deux nouveaux individus. L'opérateur de croisement utilisé dans notre algorithme génétique est le croisement à deux points de coupure. La particularité du croisement qu'on va utiliser est qu'il est, contrairement au croisement traditionnel, réalisé à l'aide d'un seul individu (bloc de base), on peut parler d'un clonage. Un individu est choisi et le croisement est effectué entre deux sous-chaînes ou sous-chromosomes. Donc les sous-chromosomes sont considérés comme des chromosomes entiers dans le cas traditionnel. Les deux points de coupure sont choisis aléatoirement et les deux sous chaînes sont permutées.
Soit un individu X représenté par le chromosome de la figure 6.9, l'opérateur de croisement peut alors choisir les nœuds 3 et 6 comme points de coupure, et donner un nouvel individu X':

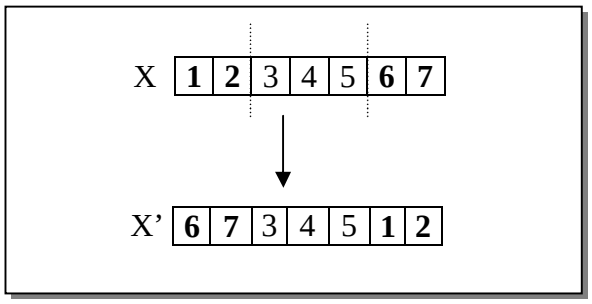



Figure 6.10 : Exemple de croisement

### 6- La mutation

la mutation constitue une exploration aléatoire de l'espace des solutions et permet d'échapper aux minimums locaux. Pour assurer, l'évolution de la population, la probabilité de mutation doit rester faible. L'opérateur de mutation choisi aléatoirement quelques uns des chromosomes enfants pour subir la mutation. Ce choix se fait comme suit :

1- Générer une valeur aléatoire pour un chromosome enfant,
2- Si cette valeur est inférieure à la probabilité de mutation $P_M$, le chromosome est choisi pour la mutation,
3- Une fois un chromosome est choisi pour la mutation, deux gênes sont choisis aléatoirement dans le bloc de base et leurs valeurs sont permutées.

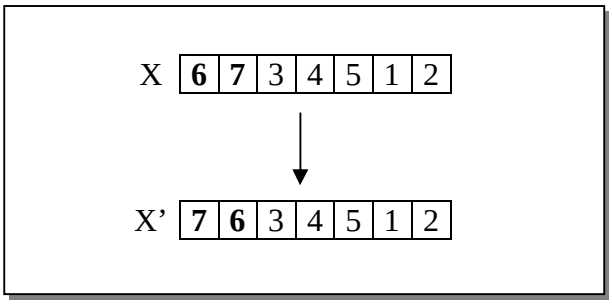



Figure 6.11 : Exemple de mutation

### 7- Sélection de la nouvelle génération

Un nombre fixe de chromosomes est choisi, de la population de la génération courante et des chromosomes enfants, pour former la population de la nouvelle génération. Cette

sélection est basée sur leurs coûts. Par exemple, si la taille de la population est égale à $N_P$ et le nombre de chromosomes enfants générés est $N_E$. Pour garder la taille de la population fixe, on fait trier en ordre décroissant les chromosomes de la génération courante et les chromosomes enfants selon leurs coûts. Les meilleurs $N_P$ chromosomes sont choisis pour former la nouvelle génération.

**8- Critère d'arrêt**

On a choisi d'utiliser le nombre de générations comme critère d'arrêt.

## 6.5. Conclusion

Le problème d'ordonnancement dans la synthèse de haut niveau est un problème NP-Complet. Dans la littérature, Plusieurs approches sont utilisées pour résoudre ce problème. En général, la recherche exhaustive prend beaucoup de temps de calcul et les heuristiques donnent des résultats de qualité médiocre. Dans ce chapitre, on propose une technique d'ordonnancement des circuits de contrôle, basée sur le couplage d'un algorithme génétique et l'algorithme d'ordonnancement à base de chemins, appelé GPBS (**G**enetic **P**ath **B**ased **S**cheduling).

L'algorithme de base qu'on a choisi, a été l'algorithme d'ordonnancement à base de chemin pour son habilité, le plus souvent, à retourner des solutions exactes et sa capacité de manipuler les applications à flot de contrôle et les opérations conditionnelles. Pour surpasser le problème d'ordre fixe des opérations dans l'algorithme PBS, un algorithme de réordonnement basé sur les algorithmes génétiques a été proposé. ce dernier permet l'extraction du parallélisme et l'élargissement des intervalles de contraintes ce qui améliore considérablement les résultats de l'algorithme PBS.

Le choix des algorithmes génétiques est basé sur le fait qu'ils ont été appliqués, avec succès, à plusieurs problèmes d'optimisation combinatoires, et ont été jugés très adéquats pour la résolution du problème d'ordonnancement. En utilisant l'algorithme d'ordonnancement à base de chemins, une connaissance spécifique est incorporée à l'algorithme génétique afin d'améliorer son efficacité. Ainsi, on tire profit de la puissance d'exploration des algorithmes génétiques et de l'adaptation de l'algorithme PBS à l'ordonnancement des circuits de contrôle pour avoir des solutions de haute qualité en un temps d'exécution acceptable.

# Chapitre 7
# Conclusion et Perspectives

## 7.1. Synthèse

L'objectif de cette thèse a été de présenter la synthèse de haut niveau, et plus spécialement présenter la tâche clé dans le flot de la synthèse de haut niveau : l'ordonnancement. Nous avons essayé de définir les différentes variations sur le problème d'ordonnancement et de décrire divers algorithmes d'ordonnancement communément utilisés aujourd'hui. En s'appuyant sur cette étude, nous proposons une nouvelle approche d'ordonnancement des circuits de commande basée sur les algorithmes génétiques.

Le chapitre 1 situ le cadre de cette thèse. L'évolution très rapide de la technologie VLSI (**V**ery **L**arge **S**cale **I**ntegration) permet déjà de réaliser des systèmes numériques complets sur une même puce. Ces systèmes hautement complexes doivent être conçus de manière efficace car le temps d'arrivée du produit sur le marché est un facteur crucial de réussite. D'où la nécessité d'utiliser de plus haut niveaux d'abstraction comme le niveau algorithmique.

Le chapitre 2 présente le flux de conception d'un circuit intégré et les différents niveaux d'abstraction. Il donne également une brève description des langages HDLs.

Le chapitre 3 donne un aperçu sur la synthèse de haut niveau. Cette méthodologie de conception est définie comme un ensemble de raffinements permettant la traduction d'une description algorithmique en une description structurelle au niveau transfert de registres.

Vu que les recherches de cette thèse se focalisent sur l'étape d'ordonnancement, un accent particulier est mis sur le problème d'ordonnancement. Ce dernier est introduit formellement dans le chapitre 4. L'ordonnancement est un problème clé dans la synthèse de haut niveau des circuits de commande, il appartient à la classe des problèmes NP-Complets. Par conséquent, il est nécessaire de développer des stratégies d'ordonnancement efficaces et

capables de produire des solutions de haute qualité, tout en satisfaisant un ensemble d'objectifs et de contraintes, en un temps d'exécution raisonnable.

Le chapitre 5 présente les différents algorithmes d'ordonnancement les plus communément utilisés dans le monde de la synthèse de haut niveau. Le choix de l'algorithme d'ordonnancement dépend du type de circuit à synthétiser.

Dans le cadre de cette thèse, on traite les circuits de commande, d'où la proposition d'une nouvelle technique adaptée à l'ordonnancement des circuits dominés par le flot de contrôle, l'algorithme GPBS. Le chapitre 6 décrit comment les algorithmes génétiques peuvent être utilisés pour la recherche d'une solution optimale dans le problème d'ordonnancement. L'approche d'ordonnancement qu'on a proposé repose sur le couplage d'un algorithme génétique avec l'algorithme d'ordonnancement à base de chemins. L'algorithme GPBS permet de mettre fin au problème d'ordre fixe des opérations dans l'algorithme PBS. Il permet l'extraction du parallélisme et l'élargissement des intervalles de contraintes ce qui améliore considérablement les résultats de l'algorithme PBS.

## 7.2. Perspectives

Les perspectives de notre travail portent sur l'aspect pratique aussi bien que sur l'aspect théorique. Sur le plan pratique, la mise en œuvre d'un outil d'ordonnancement qui implante l'efficacité de notre approche, l'algorithme GPBS technique proposée dans ce mémoire, constitue une perspective immédiate. Ainsi, ça nous permet de l'appliquer à des applications plus complexes ce qui permet d'explorer les limites de cette approche.

Sur le plan théorique, plusieurs axes de travail sont envisageables. Tout d'abord, une extension possible de l'algorithme de réordonnement présenté dans ce mémoire serait d'autoriser le déplacement des opérations à travers les nœuds conditionnels, c'est à dire entre les différents blocs de base. Chose qui n'as pas été prise en considération dans notre approche à cause de son impact sur le contrôleur qui aura à manipuler l'exécution spéculative des opérations.

L'approche de réordonnement présentée dans ce mémoire est une technique qui permet à l'algorithme d'ordonnancement à base de chemin d'exploiter le parallélisme, ce qui peut résulter en moins d'étapes de contrôle par chemin. Néanmoins, la complexité de l'approche à base de chemin est encore proportionnelle au nombre de chemins qui peut être exponentiel. Finalement, un axe qui offre, à notre avis, le plus de perspective de recherche, concerne l'amélioration de l'algorithme GPBS par la réduction de cette complexité. Ainsi et afin de réduire la complexité de l'algorithme à base de chemin, il est vital de réduire le nombre total de chemins d'exécution qui doivent être maniés.

# BIBLIOGRAPHIE


- **[1]** M. Belhadj, " Conception d'architectures en utilisant SIGNAL et VHDL", Thèse de Doctorat, Université de Rennes 1- Institut de Formation Supérieure en Informatique et Communication, 1994.
- **[2]** V. Tchoumatchenko, " Modélisation, Architecture et Outils de Synthèse pour Additionneurs Rapides", Institut National Polytechnique de Grenoble, Décembre 1998.
- **[3]** B. Laurent, "Conception des Blocs Réutilisables Réflexion sur la Méthodologie" , Thèse de Doctorat, Institut National Polytechnique de Grenoble, 1999.
- **[4]** P. Ellervee , " High Level Synthesis of Control and Memory Intensive Applications", Thèse de Doctorat, Royal Institute of Technology, Stockholm, 2000.
- **[5]** M. Aichouchi, " Etude des liens entre la synthèse architecturale et la synthèse au niveau transfert de registres", Thèse de Doctorat, Institut National Polytechnique de Grenoble, 1994.
- **[6]** G. Jervan, " Decision Diagram Synthesis from VHDL", Master Thesis, Tallinn Technical University, 1998.
- **[7]** E. Berrebi, " Méthodologie pour l'application industrielle de la synthèse comportementale", Thèse de Doctorat, INPG, 1997.
- **[8]** Z. Sugar, " Synthèse comportementale basée sur l'ordonnancement ", Thèse de Doctorat, INPG, Mai 2000.
- **[9]** IEEE Standard VHDL Language Reference Manual, Juin 1994.
- **[10]** D.E. Thomas, P. Moorby, " The VERILOG Hardware Description Language", Kluwer Academic Publishers, 1991.
- **[11]** D. Ku, G. De Micheli, " Hardware C- A Language for Hardware Design", Technical report, Computer Systems Laboratory, Université de Stanford, 1990.
- **[12]** P.N. Hilfinger, " A High Level Language and Silicon Compilation for Digital Signal Processing", Proc. of IEEE Custom Integrated Circuits Conference, pp.213-216, 1985.
- **[13]** T. Gautier, P. Le Guernic, " Signal, A Declarative Language for Synchronous Programming of Real-Time Systems", Computer Science, Functional Languages and Computer Architectures, 274, 1987.
- **[14]** A.A. Jerraya, H. Ding, P. Kission et M. Rahmouni, " Behavioral Synthesis and Component Reuse with VHDL", Kluwer Academic Publishers, 1997.
- **[15]** H. Ding, " Synthèse Architecturale Interactive et Flexible", Thèse de Doctorat, INPG, Avril 1996.
- **[16]** Y.L. Lin, " Recent Development in High Level Synthesis ", National Science Council of R.O.C, 1999.
- **[17]** T.D. Friedman, S.C. Yang, " Methods used in automatic design generator (ALERT) ", IEEE trans on computers, C-18:593-614, 1969.
- **[18]** G. Zimmermann, " The MIMOLA design system : A Computer aided digital processor design method". In Proceedings of the Design Automation Conference, 1979.
- **[19]** K. Kuchcinski, " An approach to high level synthesis using constraint logic programming", Euromicro 98, Août 1998.
- **[20]** D. Gajski, N. Dutt, A. Wu et S. Lin, " High-Level Synthesis, Introduction to chip and system design", Kluwer Academic Publisher, 1992.

- **[21]** S. Govindarajan, " Algorithms for Design Space Exploration and High Level Synthesis for Multi-FPGA Reconfigurable Computers", PhD Thesis, University of Cincinnati, Mars 2000.
- **[22]** J. Silc, " Scheduling strategies in high level synthesis", Technical Report CSD-94-3, Jozef Stefan Institute "Slovenia", Mars 1994.
- **[23]** P. Kission, " Exploitation de la hiérarchie et de la réutilisation de blocs existants par la synthèse de haut niveau", Thèse de Doctorat, INPG, Janvier 1996.
- **[24]** M. J. Maria Heijligers, " The application of Genetic Algorithms to High-Level Synthesis", PhD Thesis, University of Eindhoven, Octobre 1996.
- **[25]** D.W. Knapp, " Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler", Prentice Hall, 1996.
- **[26]** S. Haynal, " Automata-based Symbolic Scheduling", PHD Thesis, Université de Californie, Décembre 2000.
- **[27]** A.H. Timmer, J.A.G. Jess, " Exact Scheduling Strategies based on Bipartite Graph Matching" , Proceedings of the European Design &Test Conference (ED&TC-EuroASIC), pp.42-47, Paris, Mars 6-9, 1995.
- **[28]** M. Narasimhan, J. Ramanujam, " Tighter lower bounds for scheduling problems in high-level synthesis", NSF Research Initiation Award CCR-9210422, 1997.
- **[29]** B. Landwehr, P. Marwedel et R. Dömer, " OSCAR: Optimum Simultaneous Scheduling, Allocation and Ressource Binding based on Integer Programming", Rapport N° 484, Université de Dortmund, Avril 1994.
- **[30]** M. Narasimhan, " Exact Scheduling Techniques for High-Level Synthesis", Master of Science in Electrical Engineering, Louisiana State University, 1998.
- **[31]** S. Govindarajan, " Scheduling Algorithms for High-Level Synthesis", Term Paper ECE 834, Cours : Digital Design Environments, University of Cincinnati, Mars 1995.
- **[32]** C.J. Tseng, D.P. Siewiorek, " Automated Synthesis of Data Paths in Digital Systems", IEEE Trans. CAD, 5, pp.379-395, 1986.
- **[33]** P.G.Paulin, J.P.Knight, " Algorithms for High Level Synthesis ", IEEE Design and Test of Computers, pp.18-31, Décembre 1989.
- **[34]** L. Stok, " Architectural Synthesis and Optimisation of Digital Systems", PhD Thesis, Technological University of Eindhoven, 10 juillet 1991.
- **[35]** A.C. Parker, J.T. Pizarro et M. Mlinar, " MAHA : a program for data path synthesis", Proceedings of the 23rd DAC, pp.462-466, Juin 1986.
- **[36]** P.G. Paulin, J.P. Knight, " Force Directed Scheduling in Automatic Data Path Synthesis", Proc. 24th ACM/IEEE Design Automation Conference, pp.195-202, 1987.
- **[37]** D. E. Thomas, E. D. Lagnese, R.A. Walker, J. A. Nestor, J. V. Rajan et B. L. Blackburn, " Algorithmic and Register Transfer Level Synthesis: The System Architect's workbench ", Kluwer Academic Publishers, 1990.
- **[38]** B. Lee, " Specification and Design of Reactive Systems", PhD Thesis, University of California, Berkeley, 2000.
- **[39]** C-T. Hwang, Al, " A Formal Approach to The Scheduling Problem in High Level Synthesis", IEEE Trans. CAD/ICAS,pp.464-475, Avril 1991.
- **[40]** T. Kim, N. Yonezawa, J.W.S. Liu et D.L. Liu, " A Scheduling Algorithm for Conditional Resource Sharing – A Hierarchical Reduction Approach", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, N° 4, pp.425-437, Avril 1994.
- **[41]** R. Camponsano, " Path-Based Scheduling for Synthesis", IEEE transactions on CAD, 1991.

- **[42]** M. Belarbi, " Compilateur des Circuits de Contrôle dans un Environnement de Haut Niveau", Université Djillali Liabes, ????????
- **[43]** F. Kurdahi, A. Parker, " REAL: A Program for Register Allocation", 24th Design Automation Conference, pp.210-215, 1987.
- **[44]** S. Sjoholm, Lennart Lindh, " VHDL for Designers", Prentice Hall Europe, 1997.
- **[45]** S. Note, W. Guerts, F. Catthoor et H. De Man, " Cathedral-III : Architecture-Driven High-Level Synthesis for High Throughput DSP Applications", 28th ACM/IEEE Design Automation Conference, 1991.
- **[46]** H. De Man, J. Rabaey, P. Six et L. Claesen, " Cathedral II : A Silicon Compiler for Digital Signal Processing ", IEEE Design and Test, pp.13-25, 1986.
- **[47]** J. Dusina, " Vérification formelle des résultats de la synthèse de haut niveau ", Thèse de Doctorat, Université Joseph Fourier- Grenoble1, Octobre 1999.
- **[48]** J.A. Nestor, G. Krishnamoorthy, " SALSA : A New Approach to Scheduling with Timing Constraints", IEEE Transactions on CAD, Mai 1993.
- **[49]** M. C. Mc Farland, " Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions", 23rd ACM/IEEE Design Automation Conference, 1986.
- **[50]** J. Lee, Y. Hsu et Y. Lin, " A New Integer Linear Programming Formulation for the Scheduling Problem in Data-Path Synthesis", International Conference on Computer-Aided Design, 1989.
- **[51]** K. O'Brien, M. Rahmouni et A.A. Jerraya, " DLS : A Scheduling Algorithm for High Level Synthesis in VHDL", Proc. of the European Conference on Design Automation, Paris, France, Février 1993.
- **[52]** G. Goossens, J. Vandewalle et H. De Man, " Loop Optimization in Register Transfer Scheduling for DSP Systems", Proceedings 26th DAC, pp.826-831, Juin 1989.
- **[53]** C. Gebotys, M. Elmasry, " A Global Optimization Approach for Architectural Synthesis", Proceedings 28th DAC, pp.2-7, Juin 1991.
- **[54]** R.A. Walker, S. Chaudhuri, " High Level Synthesis: Introduction to the Scheduling Problem", IEEE Design & Test, 1995.
- **[55]** M. potkonjak, J. Rabaey, " A Scheduling and Ressource Allocation Algorithm for Hierarchical Signal Flow Graphs", 26th Design Automation Conference (DAC) , Las Vegas, Juin 1989.
- **[56]** P. Guiberlet, H. Kramer et W. Rosentiel, " Casch- A Scheduling Algorithm for High-Level Synthesis", Proc.EDAC, pp.311-315, 1991.
- **[57]** R. Jain, A. Mujumdar, A. Sharma et H. Wang, " Empirical evaluation of some high level synthesis scheduling heuristics", Proc. of 28th DAC, pp.210-215, 1991.
- **[58]** A.F. Dias, M. Akil, C. Lavarenne et Y. Sorel, " Vers la Synthèse Automatique de Circuits à partir de Graphes Algorithmiques Factorisés ", 5ème Workshop AAA sur l'Adéquation Algorithme Architecture – Rocquencourt, Janvier 2000.
- **[59]** I.C. Park, C.M. Kyung, " Fast and Near Optimal Scheduling in Automatic Data Path Synthesis", Proc. of the 28th DAC, pp.680-685, 1991.
- **[60]** S. H. Huang, Y. L. Jeang, C. T. Hwang, Y. C. Hsu et J. F. Wang, " A Tree-Based Scheduling Algorithm for Control-Dominated Circuits", Proc. of 30th ACM/IEEE Design Automation Conference, pp. 578-582, Juin 1993.
- **[61]** M. Rahmouni, K. O'Brien et A. Jerraya, " A Loop-Based Scheduling Algorithm for Hardware Description Languages", Journal of Parallel Processing Letters, vol. 4, No. 3, pp.351-364, 1994.

- **[62]** M. Rahmouni, A. Jerraya, "PPS: A Pipeline Path-based Scheduler", Proc. of 6th ACM/IEEE European Design and Test Conference, Paris, France, Février 1995.
- **[63]** M. Rahmouni, A. Jerraya, " Formulation and Evaluation of Scheduling Techniques for Control Flow Graphs", Proc. of EURO-DAC'95, 1995.
- **[64]** A.M. Sllame , V. Drabek, " An Efficient List-Based Scheduling Algorithm for High Level Synthesis", Proceedings of the Euromicro Symposium on Digital System Design (DSD'02), IEEE 2002.
- **[65]** N.J. Warter, " Modulo Scheduling with Isomorphic Control Transformations", PhD Thesis, University of Illinois, 1994.
- **[66]** p ge 47, chap 5
- **[67]** F.R. Boyer, " Optimisation lors de la synthèse de circuits à partir de langages de haut niveau" , Thèse PhD, Université de Montréal, Avril 2001.
- **[68]** S. Haynal, F. Brewer, " A Model for Scheduling Protocol-Constrained Components and Environments ", Proc. 36th ACM/IEEE Design Automation Conference , pp.292-295, 1999.
- **[69]** T. Zeitlhofer, B. Wess," Operation Scheduling for Parallel Functional Units", Proc. ICASSP'99, Volume 4, USA, Mars 15-19, 1999.
- **[70]** M. Nourani, C. Papachristou et Y. Takefuji, " A Neural Network Based Algorithm for the Scheduling Problem in High-Level Synthesis", IEEE 1992.
- **[71]** M.J.M. Heijligers, L.J.M. Cluitmans et J.A.G. Jess, " High-Level Synthesis Scheduling and Allocation using Genetic Algorithms", Proceedings of the Asia and South Pacific Design Automation Conference, pp.61-66, Chiba, Japan, 1995.
- **[72]** M.J.M. Heijligers, J.A.G. Jess, " High-Level Synthesis Scheduling and Allocation using Genetic Algorithms based on Constructive Topological Scheduling Techniques", Internation Conference on Evolutionary Computing, Perth, Western Australia, 1995.
- **[73]** G. Lakshminarayana, K.S. Khouri et N.K. Jha, " Wavesched: A Novel Scheduling Technique for Control-Flow Intensive Behavioral Descriptions", IEEE 1997.
- **[74]** J. Zhu, D.D. Gajski, " Soft Scheduling in High Level Synthesis ", DAC'99, New Orleans, Louisiana, 1999.
- **[75]** A.H. Timmer, M.J.M. Heijligers, L. Stok et J.A.G. Jess, " Module Selection and Scheduling using Unrestricted Libraries", Proceedings of the EDAC/EuroASIC Conference, pp.547-551, Paris, France, Février 22-25, 1993.
- **[76]** B. Mesman, M.T.J. Strik,, A.H. Timmer, J.L. Van Meerbergen et J.A.G. Jess, " An Integrated Approach to Register Binding and Scheduling " , Proceedings of the ProRISC Workshop on Circuits, Systems and Signal Processing, pp.365-373, 1997.
- **[77]** F. Gruian, K. Kuchcinski, " A Constraint Logic Programming Based Approach to High-Level Synthesis for Low Power", Proceedings of the 24th EUROMICRO Conference, Sweden, Août 25-27, 1998.
- **[78]** E. Casseau, C. Jego et E. Martin, " Synthèse architecturale d'applications temps réel pour technologies submicroniques ", RSTI-TSI, Revue des Sciences et Technologies de l'Information, Série Technique et Science Informatique, Hermes-Lavoisier, volume 23, pp.35-66, 2004.
- **[79]** S.S. Bhattachryya, P.K. Murthy et E.A. Lee, " Synthesis of Embedded Software from Synchronous Data Flow Specifications", Journal of VLSI Signal Processing 21, Kluwer Academic Publishers, pp.151-166, 1999.
- **[80]** R. Schreiber, S. Aditya, B. Ramakrishna Rau, V. Kathail, S. Mahlke, S. Abraham et G. Snider, " High-Level Synthesis of Non programmable Hardware Accelerators", Hewlett-Packard Company, Mai 2000.

- **[81]** V. Miranda, L.M. Proença, " Genetic/Evolutionary Algorithms and Application to Power Systems".
- **[82]** E. Ramat, G. Venturini, C. Lente et M. Slimane, " Planification de projets à contraintes de ressources multiples : un algorithme génétique basé sur la fréquence des gènes", Deuxième Congrès International Franco-Québécois de Génie Industriel- ALBI 1997.
- **[83]** J.R. Koza, " Genetic Programming", Encyclopedia of Computer Science and Technology, Août 1997.
- **[84]** H. Harmanani, R. Saliba et M. Khouri, " A Genetic Algorithm for Testable Data Path Synthesis", IEEE 2001.
- **[85]** T. Yamada, R. Nakano, " Scheduling by Genetic Local Search with Multi-Step Crossover", The Fourth International Conference on Parallel Problem Solving from Nature (PPSNV IV), Allemagne, pp. 960-969, Septembre 22-23, 1996.
- **[86]** X-J. Zhang, K-W. Ng et G.H. Young, " High Level Synthesis using Genetic Algorithms for Dynamically Reconfigurable FPGAs", IEEE 1997.
- **[87]** F. Bérard, C.A. Pantel, L. Pibouleau et S. Domenech, " Résolution de problèmes d'ordonnancement en génie des procédés", Deuxième Congrès International Fronco-Québécois de Génie Industriel, ALBI 1997.
- **[88]** T. Vallée, M. Yildizoglu, " Présentation des algorithmes génétiques et de leurs applications en économie", v. 4.2, Décembre 2003.