REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE SAAD DAHLEB DE BLIDA

FACULTE DES SCIENCES

DEPARTEMENT D'INFORMATIQUE

MEMOIRE

En vue de l'obtention du diplôme d'ingénieur d'état
en Informatique

Option :systèmes d'informations et bases de données

STRUCTURE D'ACCUEIL :
Centre de Développement des Technologies Avancées (CDTA)
Laboratoire de microélectronique

THEME

Contribution à l'ordonnancement des instructions d'un graphe de flots de données contrôlées

DIPLOMANTS :

Mr BOUGHERARA MAAMAR
Mr BEHA LAMINE

Membres du Jury :

Mme. A. MELLAK (PRESIDENTE DU JURY, Université de Blida)
Mme. A. SELLALI (EXAMINATRICE, Université de Blida)
Mr. A. OULD AISSA (EXAMINATEUR, Université de Blida)
Mr. A. MAHDOUM (ENCADREUR, CDTA)

PROMOTION 2002/2003

DEDICACE

Il y a toujours une raison finale pour toute chose. La raison finale pour mes cinq ans c'est achever ce travail, et je dois remercier le bon Dieu qui nous a donné le courage, la patience et la santé. Dieu merci !
Alors ce modeste travail mérite d'être dédié aux êtres les plus chers dans ma vie.

A ma mère, la lumière de mes yeux.

A la sagesse et la douceur de mon père qui m'a aidé pendant tout mon cursus de formation.

A mon deuxième père et très cher frère DJAMEL pour son encouragement

A ma deuxième mère et très chère sœur FATMA qui était toujours à coté de moi et qui m'a aidé dans les moments difficiles.

A mon frère BRAHIM et ma sœur ILHEM et SIHEM

A toute la famille BOUGHERARA.

A mon très cher oncle FODIL ABDELKADER qui m'a aidé dans tous les moments.

A toute la famille FODIL.

A mon très cher promoteur MAHDOUM ALI pour son aide.

A mon très cher Ami HAMLATI ANIS qui m'a aidé dans tous les moments.

A tous mes amis qui me connaissent, sans exception.

A tous mes collègues universitaires et à tous les étudiants de la session 2002/2003.

MAAMAR

Dédicace

Je dédie ce modeste travail :

- *A mon tendre père qui était mon meilleur promoteur.*
- *A ma tendre mère qui par ses conseils et ses prières m'a toujours soutenu et encouragé.*
- *A mes très chers frères et sœurs .*
- *A ma belle sœur et beaux frères.*
- *A mes nièces et neveux.*
- *A tous mes amis.*
- *A toute ma promotion.*

Beha lamine

REMERCIEMENTS

Tout d'abord merci à Dieu de nous avoir donné la santé et la volonté pour terminer ce projet.

Au terme de ce travail, nous tenons à remercier vivement notre promoteur :

Mr MAHDOUM ALI

qui nous a proposé le sujet .Il nous a beaucoup aidé et encouragé tout au long de ce travail.

Nous tenons aussi à remercier l'enseignante MOKHTARI pour son aide et ses précieux conseils.

Nous voudrions également associer à nos remerciements toute l'équipe d'enseignants de l'université de BLIDA qui nous a guidé le long de notre parcours d'études.

Nous tenons à exprimer nos remerciements au personnel du C.D.T.A de BABA HASSEN.

Que tous nos collègues et ami (e) s qui nous ont soutenu et encouragé pour mener à terme ce travail, trouvent ici l'expression de notre profonde reconnaissance.

A LA MEMOIRE

DES VICTIMES DE LA

CATASTROPHE NATIONALE DU

21 MAI 2003

RESUME

L'évolution de l'informatique et de l'électronique dans les cinquante dernières années a été remarquable.

Restreinte à ses débuts aux applications scientifiques ou militaires, elle fait aujourd'hui partie de la vie quotidienne. Une évolution si rapide que plusieurs ordinateurs qui appartenaient au domaine de la science fiction il y a vingt ans sont aujourd'hui disponibles pour le grand public.

Ceci est dû au développement considérable de la technologie des semi-conducteurs et à des outils de CAO des circuits intégrés (CIs) de plus en plus performants. En effet, différents outils de synthèse, d'analyse et de vérification interviennent dans la CAO des CIs et ce, à différents niveaux d'abstraction. Parmi ces outils, on distingue celui de la synthèse de parties opératives (ensemble de registres et d'unités fonctionnelles interconnectés) qui consiste à ordonnancer les instructions d'un algorithme à implémenter sur un ASIC (circuit intégré à application spécifique) dans un souci d'obtenir un circuit autant performant que possible.

Un flot de données contrôlé engendrant un nombre exponentiel de chemins de données, la méthode exacte pour un tel ordonnancement est à écarter (la complexité algorithmique ne serait pas polynomiale), et il s'agit alors d'opter pour une heuristique.

Notre travail consiste, à partir de résultats partiels, à développer une technique d'ordonnancement efficace des instructions (pouvant être assujetties à des contraintes imposées par l'utilisateur) de flots de données contrôlés, visant un bon compromis entre la surface de la partie opérative et sa vitesse d 'exécution en termes de nombre de cycles.

SOMMAIRE

TABLES DES FIGURES

Tables des algorithmes

CHAPITRE 1

INTRODUCTION GENERALE

1.1 Introduction générale:

L'ordonnancement est requis dans tout processus où les tâches doivent être exécutées selon une procédure bien établie (exemple processus industriels) où le but recherché est l'amélioration de la production et la productivité.

Il est aussi rencontré dans la planification dans le temps des différentes tâches d'un projet, et dans bien beaucoup d'autres domaines.

La complexité de l'ordonnancement diffère d'un domaine à l'autre, mais dans tous les cas, il s'agit de conduire au mieux (d'optimiser) les tâches tout en considérant les différentes contraintes imposées par le problème concerné.

Pour ce qui est de notre travail, l'ordonnancement consiste à optimiser le temps d'exécution d'un ensemble d'instructions affectées à une partie opérative tout en considérant différentes contraintes.

Celle-ci est un composant d'un circuit intégré et est constituée généralement de registres (pour mémoriser les données), d'unités fonctionnelles (opérateurs arithmétiques, logiques et relationnels) pour exécuter les différentes instructions (sous le contrôle d'une partie de contrôle), et d'interconnexions servant pour le transfert des données entre les registres et les unités fonctionnelles.

Les différentes instructions décrites dans un algorithme sont alors soumises à des techniques de synthèse (dont l'ordonnancement) pour être implémentées par la suite sur un circuit intégré à application spécifique ou **ASIC** (**A**pplication-**S**pecific **I**ntegrated **C**ircuit).

Ce type d'implémentation est adopté lorsque des critères de performance sont posés, ou lorsque le type de l'application le nécessite.

Pour une montre électronique par exemple, il n'est pas possible de porter avec soi un **PC** pour avoir l'heure ! Ainsi, lorsque l'application le nécessite ou lorsqu'un algorithme bien défini est fréquemment exécuté, il

est plus intéressant de l'implémenter par un **ASIC** plutôt que par un processeur d'un **PC** qui lui, est à usage général (son avantage lorsque l'application le permet) mais moins performant qu'un **ASIC**.

L'implémentation d'un algorithme sur un **ASIC** nécessite, outre d'autres tâches, l'ordonnancement de ses instructions.

Du fait que la programmation linéaire entière n'est pas pratique pour des tailles de chemins de données importantes, des heuristiques sont alors à envisager telles que leur complexité algorithmique soit polynomiale, tout en visant une solution au problème de l'ordonnancement, proche de l'optimum.

L'ordonnancement nécessite lui-même un traitement préliminaire, lequel travail a été effectué par un binôme encadré par notre encadreur dans le cadre de leur thèse d'Ingénieur [2].

Ce travail sera brièvement présenté au deuxième chapitre suivi au troisième chapitre par la présentation des techniques d'ordonnancement de base. Nous verrons que l'ordonnancement dont il est question dans notre contexte se fait en deux étapes.

Celles-ci seront discutées dans les chapitres quatre et cinq .

Enfin, nous terminons par une conclusion générale, suivie par des références bibliographiques.

CHAPITRE 2

PRE-ORDONNANCEMENT

CHAPITRE 2 : PRE-ORDONNANCEMENT

2.1 Introduction :

Notre travail porte sur l'ordonnancement de flots de données simples extraits d'un flot de données contrôlées. Les différents ordonnancements seront par la suite synthétisés en vue d'un ordonnancement global de tous les chemins.

Toutefois, ce travail ne peut se faire d'emblée, mais nécessite un traitement préliminaire portant sur l'analyse de l'algorithme à implémenter pour identifier les structures de contrôle, ainsi que les différents chemins de données. Nous commercerons par définir les notions de flots de données simples et contrôlées, et verrons par la suite que la complexité algorithmique du problème posé nécessite certaines transformations préparant l'ordonnancement proprement dit. Ces différents aspects, abordés dans une autre thèse d'ingénieur, seront présentés dans ce chapitre.

2.2. Flots de données simples et contrôlés

2.2.1. Flot de données simples :

C'est un flot de données dont les instructions s'exécutent en absence de toute structure de contrôle.

Dans ce type de flot de données, toutes les instructions du flot s'exécutent une fois que c'est permis, et il n'existe qu'un seul chemin de données. Les flots de données indiqués en gras dans les figures **FIG 2.3** page 13, en sont un exemple.

2.2.2. Flot de données contrôlées :

Dans ce type de flot de données par contre, seules les instructions se trouvant dans les branches conditionnelles correspondantes aux conditions dont la valeur logique est vraie s'exécutent.

Ainsi, dans le flot de la Figure **FIG 2.1**, les instructions **INST4** et **INST5** ne s'exécutent que si la valeur logique de A est fausse.

```
INST1 ;
INST2 ;
if(A)
then INST3 ;
else {INST4 ;
      INST5 ;
     }
endif
INST6 ;
```

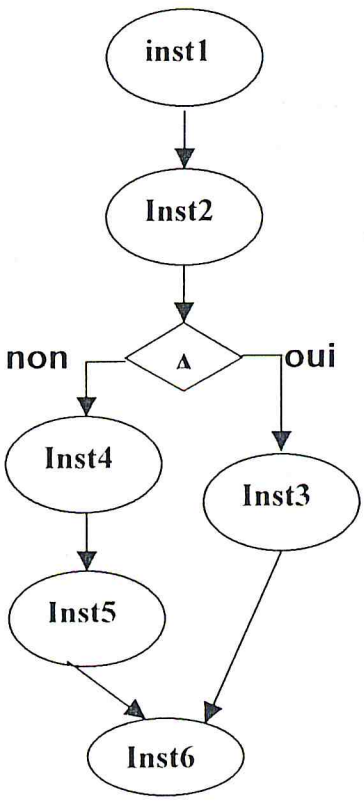


Fig. 2.1. Un exemple de flot de données contrôlées

Notons que dans ce type de flots, le nombre de chemins de données est égal à M_1, où : $M_1 = \prod_{i=1}^{N} N_i$, N_i étant le nombre de chemins possibles pour la $i^{ème}$ structure de contrôle, et N étant le nombre de structures de contrôle.

2.3. Transformation algorithmique

Comme il a été dit précédemment, l'ordonnancement des instructions d'un algorithme à implémenter sur un **ASIC** nécessite un travail préliminaire.

Il s'agit tout d'abord de transformer un algorithme en un ensemble de flots de données contrôlés ou **CDFG (*Controlled Data Flow Graph*).**

Cette transformation consiste essentiellement à détecter toutes les structures de contrôle (***if-then-else***) et les instructions de chaque branche conditionnelle.

En remarquant que selon la valeur d'une condition relative à une structure de contrôle donnée, l'une ou l'autre branche seulement sera exécutée.

En généralisant pour tout l'algorithme, selon les valeurs des conditions des différentes structures de contrôle, un ensemble d'instructions bien défini sera exécuté par la partie opérative. Cet ensemble est appelé, dans notre contexte, *chemin de données*. Les valeurs des différentes conditions pouvant être quelconques, il est évident que la combinaison de toutes ces valeurs engendrera plusieurs chemins de données.

Considérant N structures de contrôle en séquence, le nombre de tels chemins est alors : $M_1 = \prod_{i=1}^{N} N_i$, N_i étant le nombre de chemins inclus dans la $i^{ème}$ structure de contrôle (noter que les ifs imbriqués sont admis). Dans le cas particulier où chaque chemin contient 2 branches, on aurait : $M_1=2^N$.

L'algorithme général (**alg1**) de l'ordonnancement serait :

pour *i=1* ***jusqu'à*** M_1
faire ***ordonnancer le chemin i ;***
fait
Synthèse des résultats partiels et report du résultat global ;

2.4 TECHNIQUE DE FUSION DE BRANCHES CONDITIONNELLES (*COLLAPSING*) :

Outre la transformation de l'algorithme en un **CDFG**, il n'est pas encore possible de procéder à l'ordonnancement. En effet, la complexité algorithmique de alg_1 est égale à $O(c*M_1)$, $O(c)$ étant la complexité algorithmique de l'ordonnancement d'un seul chemin de données.
Il est clair que même si cette dernière est polynômiale, celle de l'algorithme général ne le serait pas. L'implémentation directe de l'algorithme alg_1 pour une valeur importante de M_1 aurait pour conséquence son exécution "infinie" et les résultats de l'ordonnancement ne pourront pas alors être générés au bout d'une durée raisonnable.
Il est clair que pour un ordonnancement possible, le nombre de chemins doit être réduit (sans pour autant fausser la logique de l'algorithme à implémenter sur un **ASIC** !).

L'une des techniques de réduction consiste à assimiler les branches de certaines structures de contrôle à une seule (technique de *collapsing*). Les structures de contrôle concernées sont celles dont les instructions incluses dans leurs branches ne sont pas assujetties à des contraintes.
Dans ce cas, les instructions concernées peuvent être placées dans le *même* état de la machine à états finis décrivant l'algorithme. Précisons qu'il existe une différence entre *état* et *cycle d'exécution* et que pour plus de précisions concernant cette technique, le lecteur peut se référer à [2].

Cette technique de réduction aura donc pour conséquence d'aboutir à M_2 chemins de données ($1 \leq M_2 \leq M_1$), et l'algorithme alg_1 se transformera au deuxième algorithme (***alg₂***) suivant :

pour i=1 jusqu'à M_2
faire ordonnancer le chemin i ;
fait
Synthèse des résultats partiels et report du résultat global ;

Dans le cas où plusieurs structures de contrôle contiennent des instructions assujetties à des contraintes, la valeur de M_2 peut rester importante, et le problème de l'ordonnancement en un temps raisonnable reste posé. Une autre réduction du nombre de chemins (tout en respectant encore la logique de l'algorithme à implémenter) s'impose alors, moyennant une technique reposant sur l'insertion de points de coupure.

2.5. INSERTION DE POINTS DE COUPURE (*CUTS*) :

La stratégie consiste à réduire le nombre de combinaisons entre les valeurs logiques des conditions correspondant aux différentes structures de contrôle.

En effet, supposons que 3 structures de contrôle se trouvent en séquence dans un algorithme et que chacune d'elles comporte 2 branches. En considérant toutes les combinaisons sur les valeurs logiques des 3 conditions, on obtiendrait 8 ($= \prod_{i=1}^{3} N_i = 2^3$) chemins. Supposons maintenant que l'on insère un point de coupure entre la deuxième et la troisième structure de contrôle, isolant ainsi cette dernière des deux autres. Le nombre de chemins de données serait alors 4($= \prod_{i=1}^{2} N_i = 2^2$) chemins, auxquels il faut ajouter les 2 chemins relatifs à la dernière structure de contrôle, soit

6 chemins au total. Cet exemple simple montre que l'insertion d'un point de coupure a permis de passer de 8 à 6 chemins.

Cependant, cette insertion n'est pas sans incidence sur le nombre de cycles d'exécution de l'**ASIC** implémentant l'algorithme.
En effet, l'insertion de ce point de coupure interdit d'exécuter en parallèle une instruction de l'une des branches de la troisième structure de contrôle avec une autre contenue dans une branche de la première ou la deuxième structure de contrôle *même* si ces deux instructions sont indépendantes. Concernant le cas général, tout l'enjeu consiste à déterminer le nombre et l'emplacement des points de coupure permettant un ordonnancement faisable, tout en anticipant sur une partie opérative performante.

2.6. GENERATION DE CHEMINS DE DONNEES :

Les techniques de fusion de branches conditionnelles et d'insertion de points de coupure permettent d'aboutir à un nombre M_3 ($1 \leq M_3 \leq M_2$), transformant l'algorithme ***alg*$_2$** à l'algorithme ***alg*$_3$** *suivant :*

***pour** i=1 **jusqu'à** M$_3$*
***faire** ordonnancer le chemin i ;*
fait
Synthèse des résultats partiels et report du résultat global ;

La transformation de l'algorithme de l'utilisateur en un **CDFG**, puis l'application éventuelle des techniques de réduction du nombre de chemins de données sont enfin suivies par l'identification des chemins de données issus de ces traitements et qui seront ordonnancés séparément.
Notons que la dernière instruction d'un chemin est celle qui précède directement le point de coupure inséré juste après, et sa première instruction est celle qui suit immédiatement un point de coupure donné.

Il est évident que les premières et dernières instructions du **CDFG** initial restent respectivement les premières et les dernières dans les chemins qui les contiennent. Les résultats de ces ordonnancements sont alors synthétisés pour générer par la suite l'ordonnancement global.

Ce chapitre a constitué une description succincte du travail préliminaire à l'ordonnancement, effectué par un binôme dans le cadre de leur thèse d'ingénieur [2]. Ce travail constitue le point de départ du nôtre, c'est-à-dire étant donné un ensemble de chemins de données dans lesquels sont indiquées des informations, il s'agit d'ordonnancer chacun de ces chemins, puis de synthétiser les résultats pour en constituer un ordonnancement global.

Les autres informations données portent, pour chaque instruction, sur la condition d'exécution, ainsi que la (ou les) contrainte(s) éventuelle(s) interdisant d'exécuter l'instruction concernée en parallèle avec d'autres, même si elles sont indépendantes.

Dans ce qui suit, nous discuterons brièvement de l'identification des chemins de données inclus dans un algorithme **(FIG 2.2)**

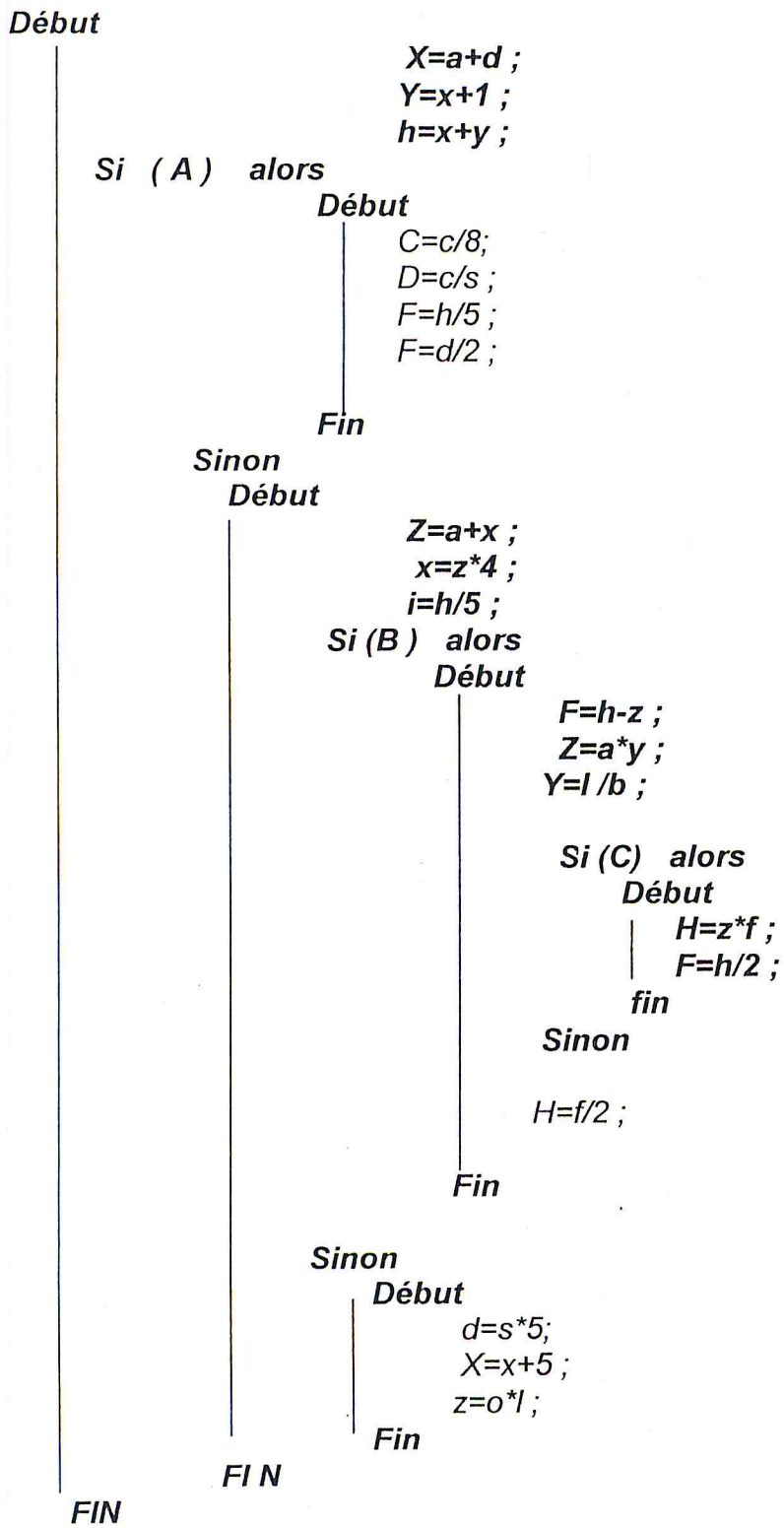


FIG 2.2 . Exemple d'un programme algorithmique

L'organigramme *:Soit l'organigramme de l'exemple de l'algorithme précédent :*

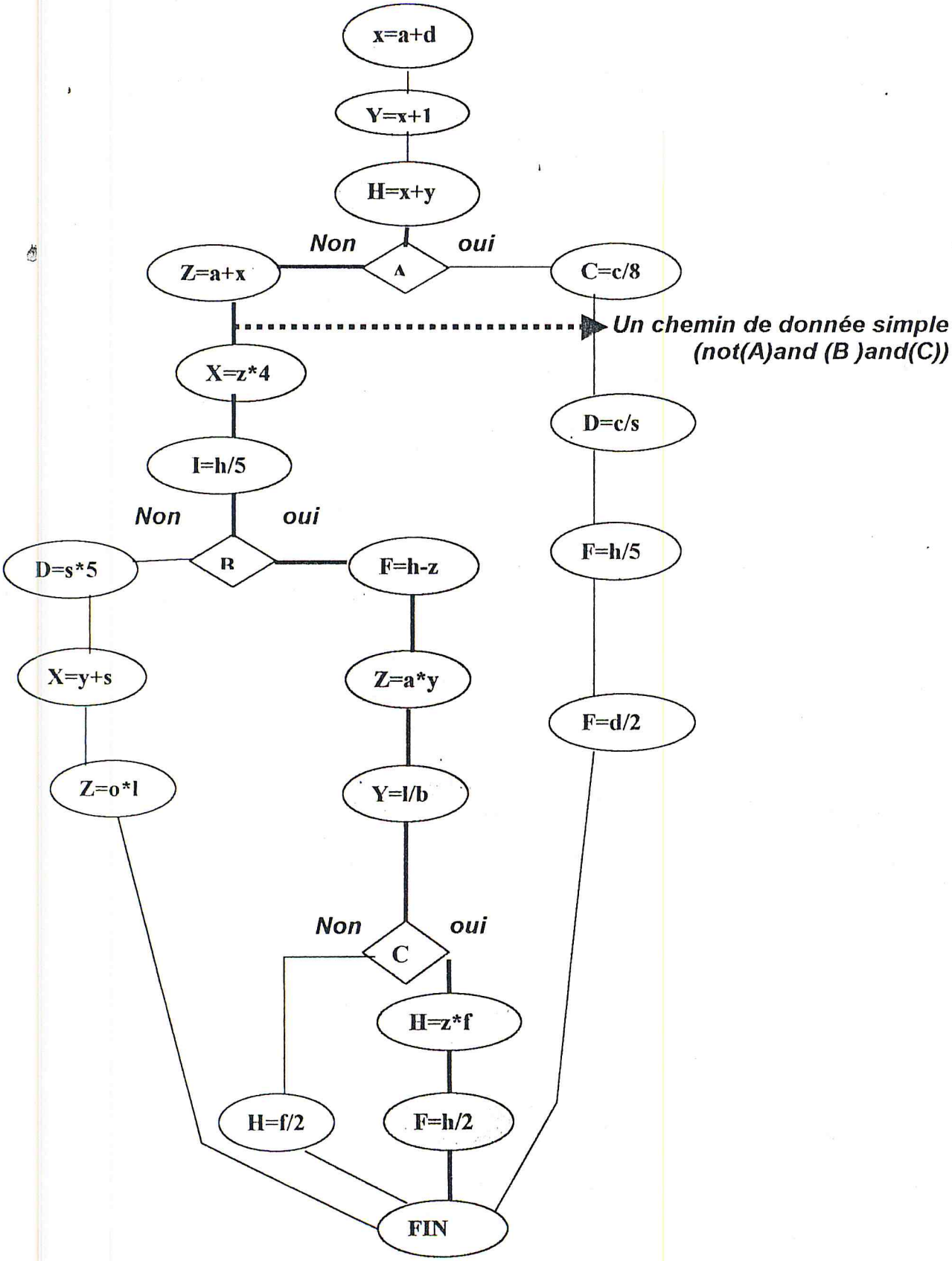


FIG.2.3 organigramme du programme algorithmique

On voit dans la **Fig (2.3)** un exemple de flot de données contrôlé qui se compose de 4 flots de données simples.

Il existe 4 chemins de données correspondant aux conditions :

A , not(A) and B and C , not(A)and B and not(C), not(A)and not(B).

Par exemple, le flot de données simple représenté en gras dans la **FIG (2.3)** s'exécute lorsque la condition **not(A) and (B) and(C)** devient vraie.

La figure suivante représente le pré-ordonnancemnt :

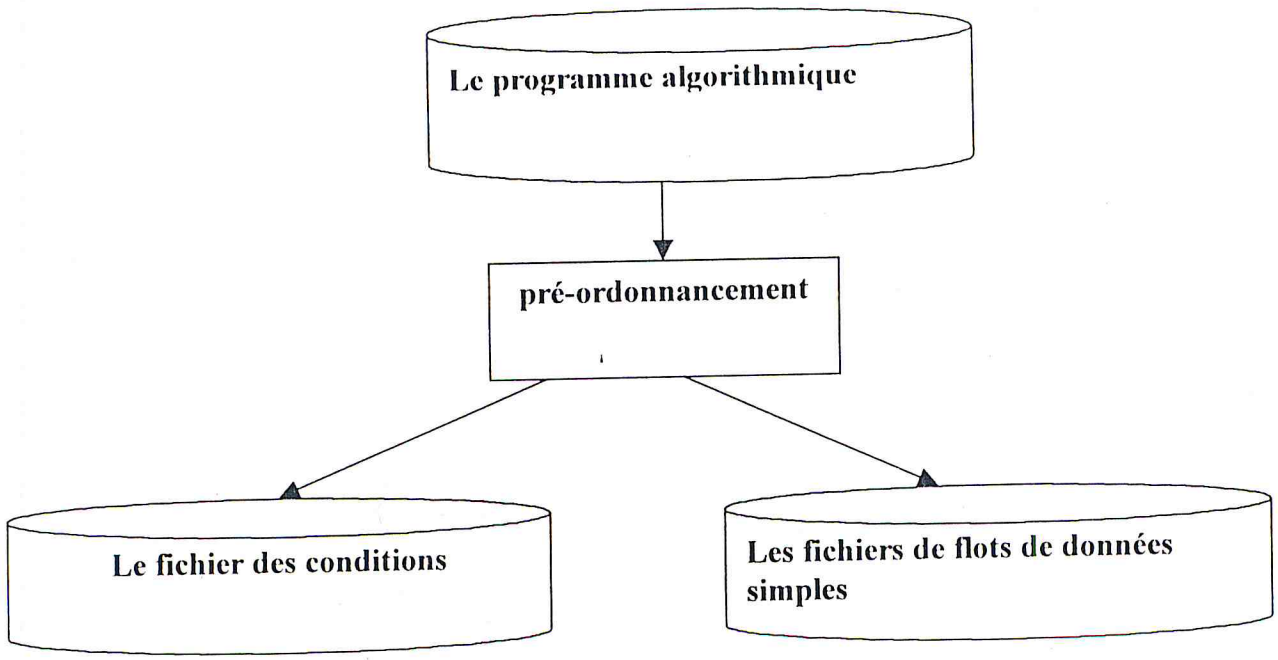


Fig 2.4 le Pré-ordonnancement

Les formats de ces fichiers sont décrits ci-après

N° instruction	N° condition	N° cycle	L 'instruction
N°contrainte	N° contrainte	.....	N°contrainte	fin

N° instruction indique le numéro de l'instruction dans le fichier des Instructions.

N° condition indique le numéro de l'enregistrement dans le fichier des conditions (condition permettant d'exécuter l'instruction)

N° cycle indique le numéro du cycle dans lequel est ordonnancée l'instruction indiquée dans le champ suivant
(ce numéro est initialisé à 0)

L 'instruction indique l'instruction donnée

N°contrainte 0 : l'instruction considérée n'est assujettie à aucune contrainte vis a vis des autres instructions

Autre : un ou plusieurs N° d'enregistrements dans le fichier des instructions (instructions avec les quelles l'instruction considérée est en conflit)

Remarque :le N° contrainte sera expliqué (traité) dans les chapitres suivants

Fin indique la fin d'un enregistrement du fichier instructions

Fig. 2.5. Format d'un enregistrement du fichier des instructions

not (A) and not (B) and not (C)

not (A) and not (B) and C

not (A) and B and not (C)

not(A) and B and C ◄..... la condition pour exécuter le flot simple représenté en page 13

A and not (B) and not(C)

A and not(B) and C

A and B and not (C)

A and B and C

Une instruction ayant comme numéro de condition 3 (dans le fichier des instructions) ne pourra alors s'exécuter que si la condition not (A) and B and not (C) est vraie.

Fig. 2.6 Exemple d'un fichier de conditions

Nous donnons ci-après (**FIG 2.7**) le flot de données simples, correspondant à la condition **(NOT (A) AND (B) AND (C))** et qui est extrait du flot de données contrôlées de la **(FIG 2.3).**

```
1   4   0   x=a+d
3   4   fin
2   4   0   y=x+1
5   fin
3   4   0   h=x+y
1   6   fin
4   4   0   z=a+x
1   fin
5   4   0   x=z*4
2   fin
6   4   0   l=h/5
3   fin
7   4   0   f=h-z
0   fin
8   4   0   z=a*y
9   10   11   fin
9   4   0   y=l/b
8   10   fin
10   4   0   h=z*f
8   fin
11   4   0   f=h/2
8   fin
```

Fig. 2.7 Exemple du fichier de flot de données simples

Considérons dans l'exemple précédent (**FIG 2.7**) les deux lignes suivantes :

```
3   4   0   h=x+y
1   6   fin
```

La troisième instruction (h=x+y) est initialement "ordonnancée" dans le cycle 0.

Elle ne peut s'exécuter que si la quatrième condition (quatrième enregistrement dans le fichier des conditions) est vraie.

L'instruction considérée (h=x+y) est en contrainte avec la première et la sixième instruction. Aussi, elle ne pourra être ordonnancée ultérieurement dans le même cycle que ces deux instructions.

2.7 CONCLUSION :

Ce chapitre constitue la présentation d'un travail préliminaire portant sur la détermination à partir d'un algorithme décrit en pseudo – C de tous les chemins de données.

le problème de complexité algorithmique et sa résolution y ont été aussi abordés, préparant ainsi l'ordonnancement proprement dit, et qui va être décrit dans les chapitres suivants .

CHAPITRE 3

ORDONNANCEMENT

CHAPITRE 3 L'ORDONNANCEMENT

3.1 Introduction :

Nous présentons dans ce qui suit quelques techniques d'ordonnancement.

Nous discuterons principalement des techniques d'ordonnancement de base, montrer leurs limites, et introduire la méthodologie d'ordonnancement retenue.

3.2 Techniques d'ordonnancement de base :

Il existe essentiellement trois techniques d'ordonnancement de base d'un chemin de données :

- **la technique ASAP (As Soon As Possible)**
- **la technique ALAP (As Late As Possible)**
- **la technique Unit-Constraint Schedule**

Nous allons les présenter en prenant l'exemple du flot de données suivant :

$$result = max (0.875 * x , 0.5 * y)$$

avec $x = max (| a | , | b |)$ et $y = min (| a | , | b |)$

3.2.1. La technique ASAP :

Cette technique consiste à allouer *le plus tôt possible* les unités fonctionnelles aux différentes opérations du flot de données. Ceci donnerait pour l'exemple cité précédemment le diagramme d'ordonnancement suivant :

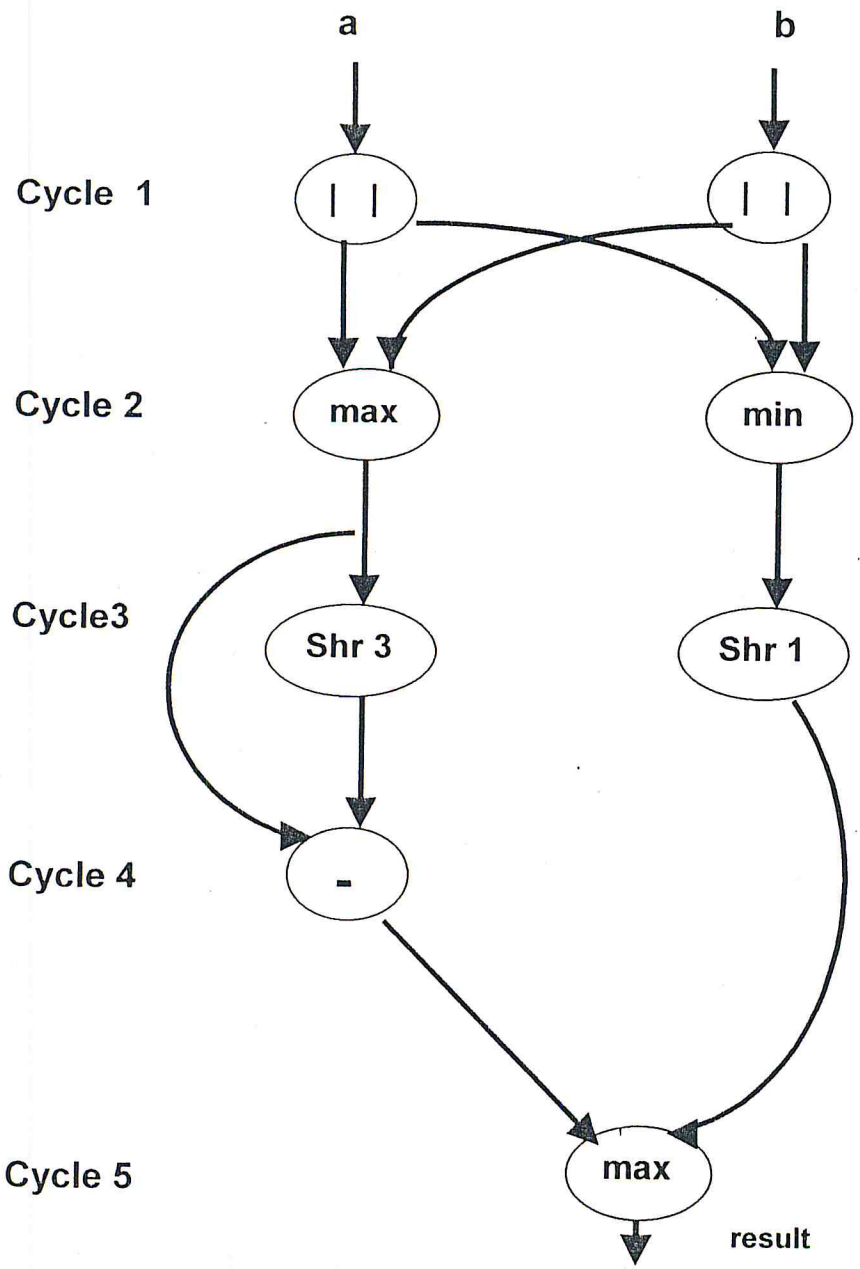


Fig.3.1.Technique ASAP

Notons que :

Shr1 :c'est le décalage à droite d'un bit.

Shr3 : décalage à droite de trois bits.

Max : c'est la fonction qui calcule le maximum

Min : c'est la fonction qui calcule le minimum.

| | : le signe pour le calcul de la valeur absolue.

- : le signe pour la soustraction.

Ce graphe décrit la technique d'ordonnancement de base ASAP pour affecter a chaque opération son cycle d'exécution :

Pour le cycle n°1 : calcul de la valeur absolue pour les variables « a et b » en même temps.

Pour le cycle n°2 : détermination du minimum et du maximum entre « a et b ».

Pour le cycle n°3 : le décalage à droite d'un bit pour le minimum, le décalage à droite de trois bits pour le maximum.

Pour le cycle n°4 : soustraction entre (x) et (0.125x).

Pour le cycle n°5 : déterminer le maximum entre le résultat de la soustraction (0.875x) et (y/2).

3.2.2. La technique ALAP :

Cette technique consiste à allouer *le plus tard possible* les unités fonctionnelles aux différentes opérations du flot de données. Ceci donnerait pour l'exemple cité précédemment le diagramme d'ordonnancement suivant :

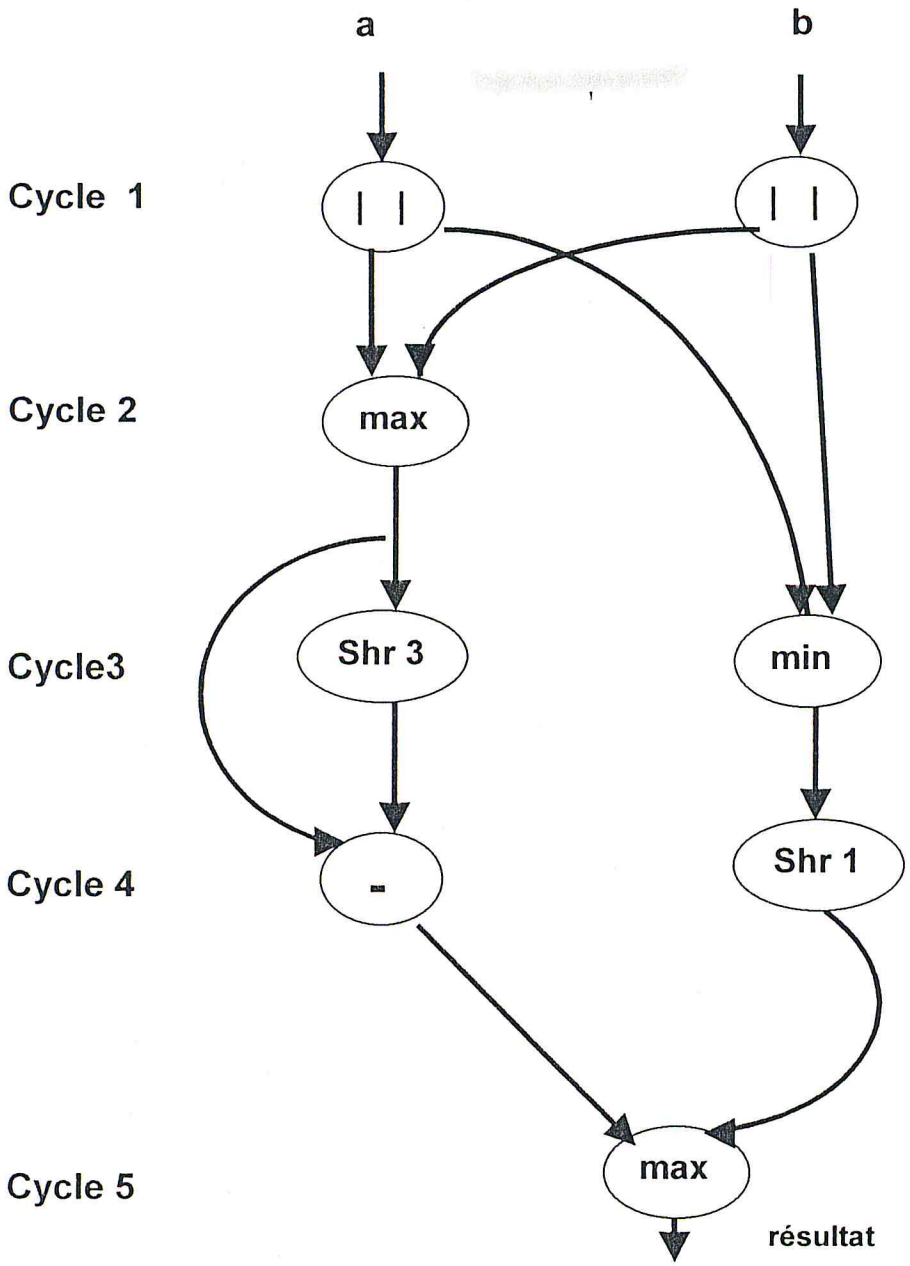


Fig.3.2.Technique ALAP

Dans cette technique les cycles d'exécution sont :

Pour le cycle n°1 : calcul de la valeur absolue pour les variables « a et b » en même temps.

Pour le cycle n°2 : détermination du maximum entre « a et b ».

Pour le cycle n°3 : décalage le décalage à droite de trois bits pour le maximum en même temps que l'opération du minimum entre « a et b ».

Pour le cycle n°4 : calcul de (0.125x) en même temps que (y/2).

Pour le cycle n°5 : déterminer le maximum entre le résultat de la soustraction et le décalage à droite d'un bit.

3.2.3. La technique Unit-Constraint Schedule :

Cette technique tient compte de contraintes éventuelles imposées sur certaines opérations, les empêchant d'être ordonnancées durant le même cycle. Un exemple en est donné ci-dessous, où il existe une contrainte sur l'utilisation d'une seule unité fonctionnelle calculant la valeur absolue:

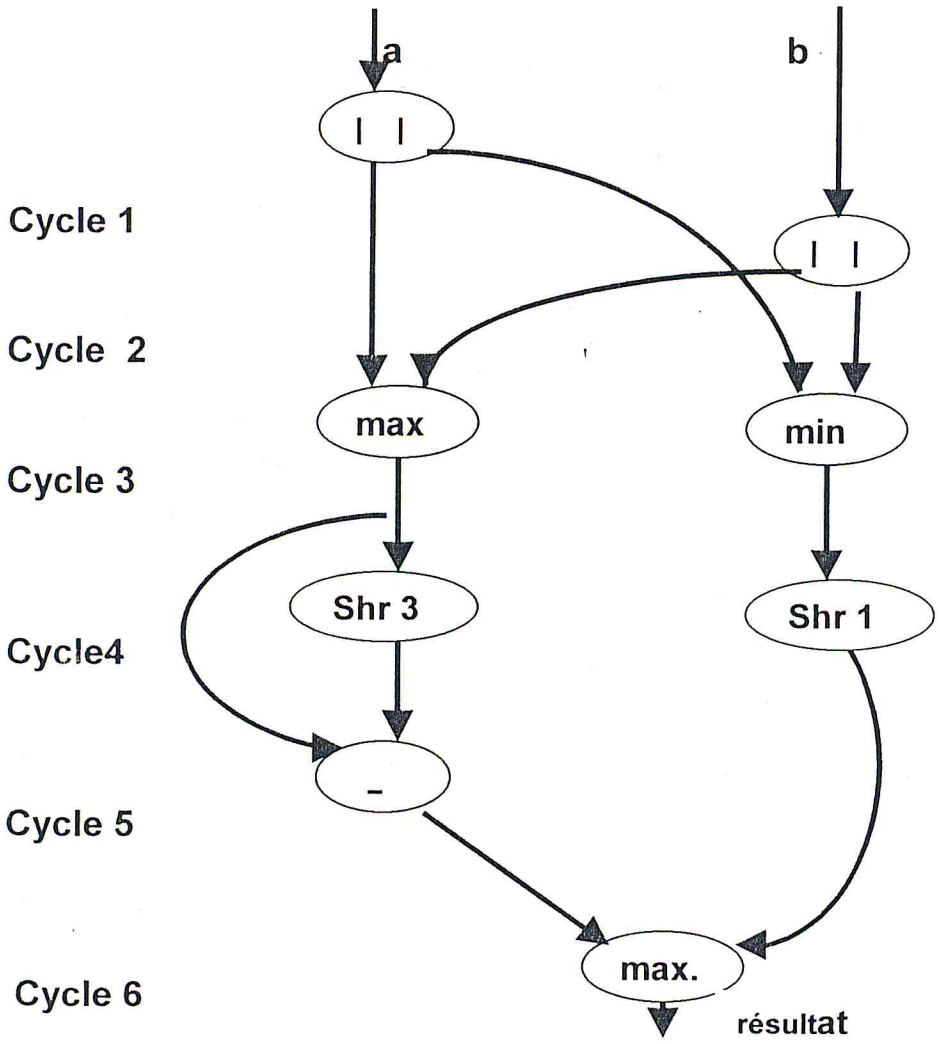


Fig.3.3.Ordonnancement ASAP avec la contrainte d'utiliser un seule unité fonctionnelle réalisant la valeur absolue

Dans cette technique les cycles d'exécution sont :

Pour le cycle n°1 : calcul de la valeur absolue pour la variable « a ».

Pour le cycle n°2 calcul de la valeur absolue pour la variable « b ».

Pour le cycle n°3 : la détermination du minimum et du maximum entre « a et b ».

Pour le cycle n°4 : le décalage à droite d'un bit pour le minimum, le décalage à droite de trois bits pour le maximum.

Pour le cycle n°5 : soustraction entre le max (a et b) et le shr3

Pour le cycle n°6 : déterminer le maximum entre le résultat de la soustraction et le décalage à droite d'un bit.

3.3. Limites des trois techniques :

A partir des diagrammes donnés précédemment, nous pouvons émettre les remarques suivantes :

- la technique **ASAP** permet un gain de temps dans la mesure où une instruction donnée est exécutée le plus tôt possible, mais au détriment de la surface puisque plusieurs unités fonctionnelles de même type (exemple addition) peuvent être sollicitées en même temps

- la technique **ALAP** permet aussi un gain de temps, mais çà peut toujours être au détriment de la surface

- le choix de l'une des techniques **ASAP** ou **ALAP** se fera en fonction de celle qui donnerait une surface plus optimale. En effet, pour l'une d'elles, certaines opérations identiques peuvent se trouver ordonnancées dans des cycles différents, ce qui permettrait d'utiliser un nombre minimal de ressources physiques pour exécuter de telles opérations

- la technique "**Unit-Constraint Schedule**" offre l'avantage d'être plus générale relativement aux deux premières qui ne tiennent pas compte de l'existence de contraintes

Afin de faire face à un flot de données quelconque (pouvant comporter des contraintes), nous avions écarté les techniques **ASAP** et **ALAP.**

La troisième technique plus générale, peut être combinée avec la première ou la deuxième technique de base, selon le paramètre surface offert. Notons au passage que le paramètre ***surface*** porte aussi sur d'autres ressources physiques et peut être optimisé par un outil d'allocation efficace de ressources physiques, étude ne rentrant pas dans le contexte de notre travail.

En général, cette troisième technique est combinée avec la première, la deuxième, ou la combinaison des deux, et ce, pour faire face à des applications exigeantes en matière de vitesse (**Exemple** : applications temps réel).

Notons que le paramètre ***surface*** peut être plus optimisé par un outil d'allocation, et ne pose plus de nos jours beaucoup de problèmes du fait du développement technologique considérable que l'on connaît actuellement (technologie **MOS 0.13µ** déjà utilisée à l'échelle industrielle qui passera très prochainement à la technologie **MOS 0.09µ**).

Si la troisième technique paraît inévitable, il convient toutefois d'opter pour une méthodologie d'ordonnancement appropriée afin de ne pas se trouver face au problème de complexité algorithmique. L'utilisation de la programmation linéaire entière illustre parfaitement ce problème comme nous allons le constater à travers ce qui suit [1]:

Soit à résoudre par la méthode de la programmation linéaire entière l'ordonnancement du chemin de données indiqué dans la (**Fig 3.4**), où O_i ($1 \leq i \leq 11$) représente une opération, $1 \leq E \leq 4$ est un cycle d'exécution dans la méthode ASAP et $1 \leq L \leq 4$ celui dans la méthode **ALAP**. A partir de ces deux techniques d'ordonnancement de base, l'intervalle d'exécution de chaque opération N_i est déduit facilement (**Cf. Fig 3.5**) où S_i $1 \leq i \leq 4$ représente un cycle d'exécution).

Soit l'ensemble suivant des opérations arithmétiques à exécuter:

R=A*b ;m=C*d ;n=E*f ;

l=A*c ;p=D+e ;q=R*m ;

j=c*n ; R2=1+l ; R3=a <p ;

K=q-f ; R1=k-j ;

Nous allons illustrer cet exemple en utilisant les techniques d'ordonnancement étudiées précédemment, et affecter à chaque opération son cycle d'exécution exact.

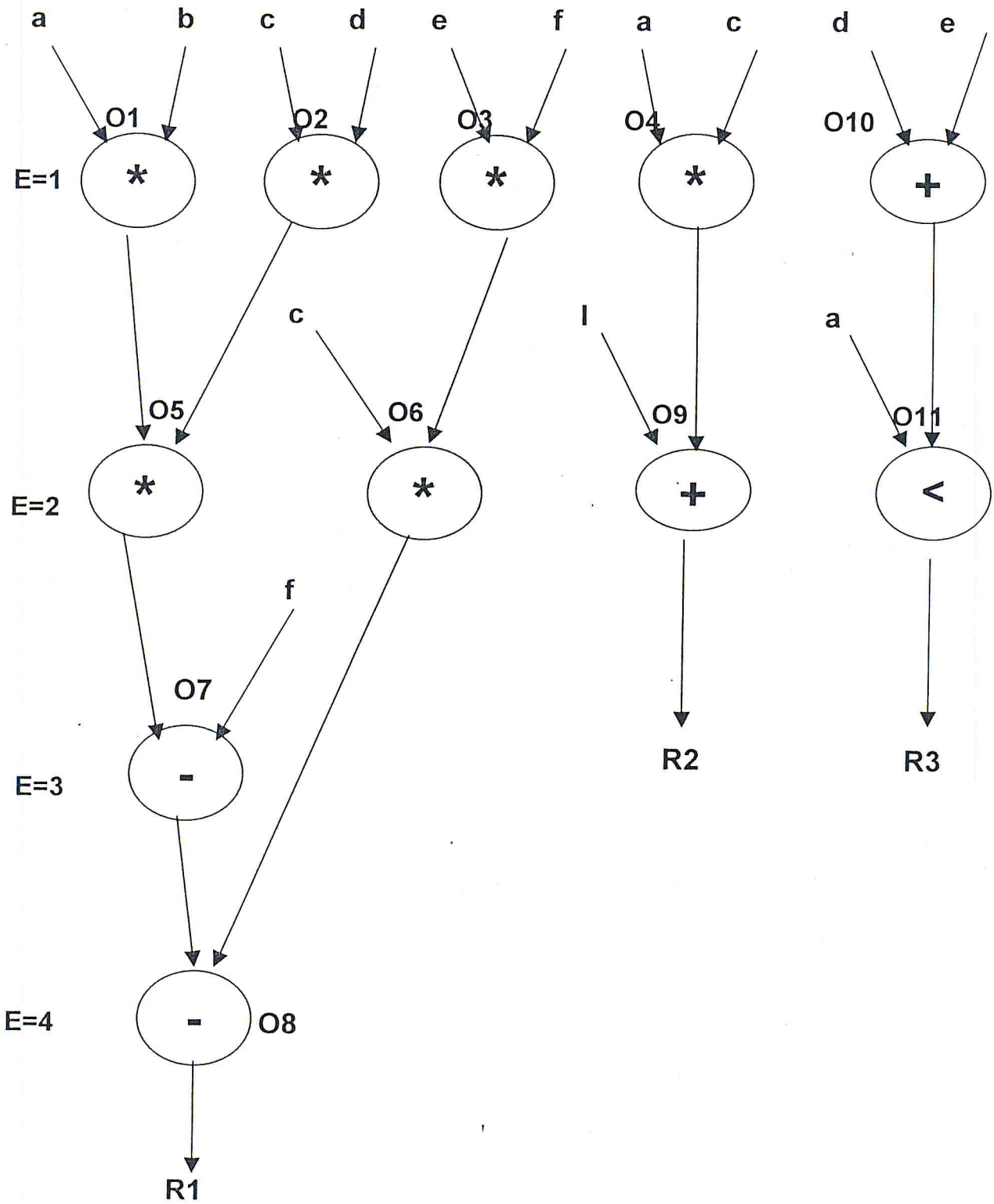


Fig.3.4.a Un ordonnancement par la technique ASAP

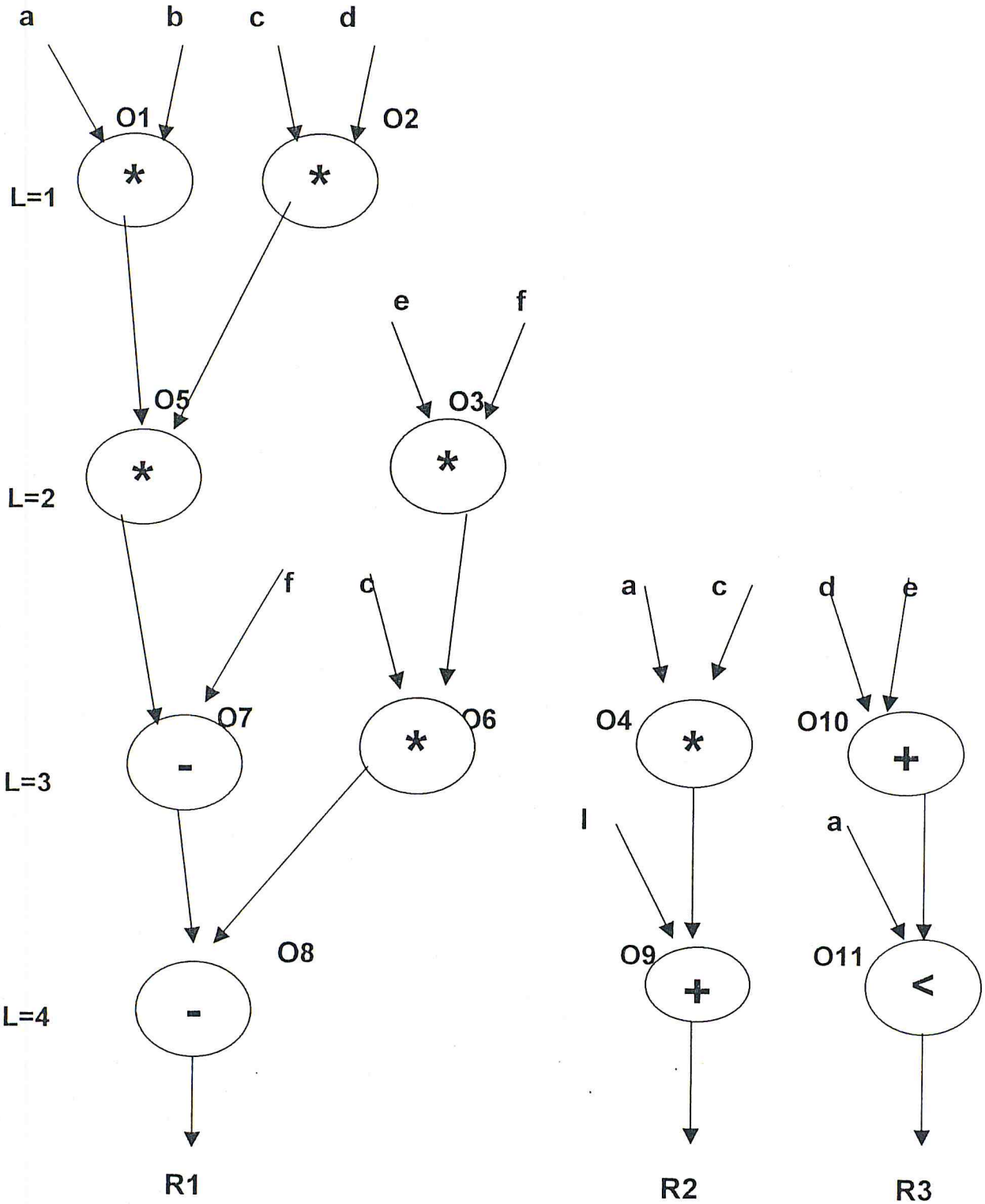


Fig.3.4.b Un ordonnancement par la technique ALAP

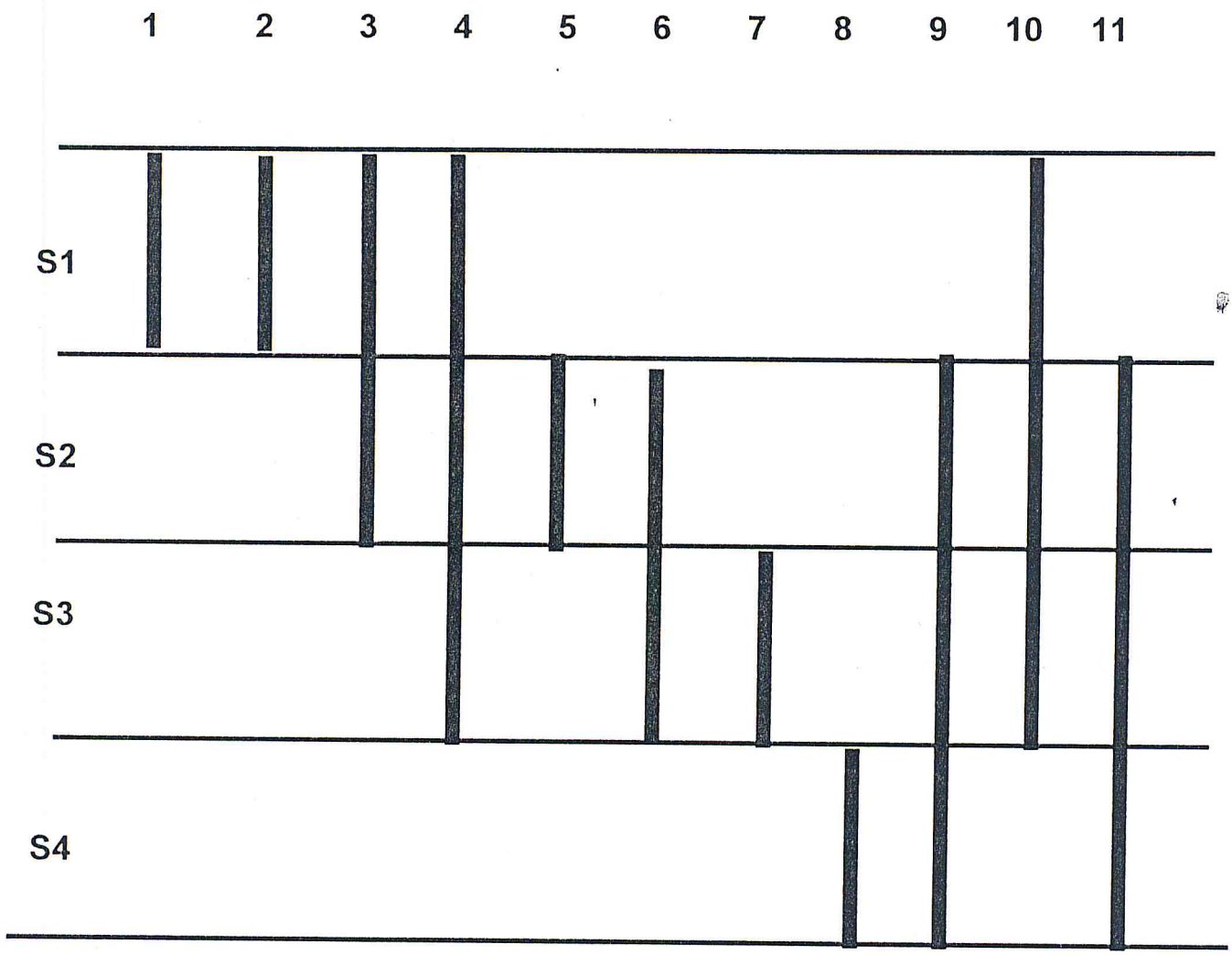


Fig.3.5. Durées de vie des opérations

Dans notre contexte, la programmation linéaire entière détermine l'ordonnancement optimal des différentes opérations en utilisant la méthode de séparation et évaluation, connue aussi dans la terminologie anglo-saxonne par ***"the branch and bound method"***, et en recourant éventuellement à des retours en arrière - ***backtracking*** – dans le sens où certaines décisions prises initialement sont réétudiées au fur et à mesure que le processus progresse.

Soient $S_{E\ k}$ et S_{Lk} les cycles d'exécution respectifs dans lesquels l'opération O_k est ordonnancée par les méthodes ASAP et ALAP. Il est clair que $E_k \leq L_k$.

En effet, l'exécution de l'opération O_k doit commencer pas plus avant que S_{Ek} et pas plus tard que L_{Ek} . Le nombre de cycles d'exécution entre S_{Ek} et L_{Ek} est appelé intervalle de mobilité de l'opération O_k , c'est à dire,

$$m\_intrv\ (O_k) = \{\ s_j\ /\ E_k \leq j \leq L_k\ \}$$

L'intervalle de mobilité de l'opération O_4 par exemple, est $\{s_1, s_2, s_3\}$ (Cf. Figure 1.5) puisque les méthodes **ASAP** et **ALAP** permettent de l'exécuter respectivement aux cycles 1 et 3 (E_4=1 et L_4=3).

Dans ce qui suit, nous allons nous baser sur les ordonnancements **ASAP** et **ALAP**, et les intervalles de mobilité de chaque opération pour formuler le problème d'ordonnancement à résoudre par la programmation linéaire entière :

Soit OP= $\{O_i\ ;\ 1 \leq i \leq n\}$ l'ensemble des opérations dans le chemin de données, et t_i = type(O_i) le type de chaque opération O_i. Soit T=$\{t_k\ ;\ 1 \leq k \leq m\}$ l'ensemble de tous les types d'opérations possibles. L'ensemble OP_{tk} est alors l'ensemble des opérations appartenant à OP et qui sont de type t_k , c'est à dire,

$$OP_{tk} = \{O_i\ ;\ O_i \in OP \wedge type(O_i) = t_k\ \}$$

Soit $INDEX_{tk}$ l'ensemble des indices des opérations incluses dans OP_{tk} , c'est à dire,

$$INDEX_{tk} = \{i\ ;\ O_i \in OP_{tk}\ \}.$$

Soit N_{tk} le nombre d'unités fonctionnelles exécutant les opérations de type t_k , et soit C_{tk} le coût d'une telle unité.

Soit $S = \{ s_j ; 1 \leq j \leq r \}$ l'ensemble des cycles d'exécution disponibles pour ordonnancer les opérations.

Enfin, soient x_{ij} des variables appartenant à $B=\{0,1\}$ telles que $x_{ij} = 1$ si l'opération O_i est exécutée dans le cycle j, et $x_{ij} = 0$ sinon.

Le problème d'ordonnancement peut être alors formulé comme suit :

$$\textbf{Minimiser} \sum_{k=1}^{m} (C_{tk} * N_{tk}) \qquad (3.1)$$

sous les contraintes :

$$\forall i \ 1 \leq i \leq n, \quad \sum_{E_i \leq j \leq L_i} x_{ij} = 1 \qquad (3.2)$$

$$\forall j \ 1 \leq j \leq r, \quad \forall k \ 1 \leq k \leq m, \quad \sum_{i \in INDEX_{tk}} x_{ij} \leq N_{tk} \qquad (3.3)$$

et

$$\forall \ i, j, \ O_i \in Preced_{Oj} [\sum_{E_i \leq k \leq L_i} (k * x_{ik}) - \sum_{E_j \leq l \leq L_j} (l * x_{jl})] \leq -1 \qquad (3.4)$$

La fonction objective (3.1) minimise le coût total des unités fonctionnelles requises.

La condition (3.2) impose à ce que chaque opération O_i soit exécutée durant un et un seul cycle d'exécution, pas plus avant que E_i (**ASAP**) et pas plus tard que L_i (**ALAP**).

La condition (3.3) assure à ce que chaque cycle d'exécution contienne au maximum N_{tk} opérations de type t_k .

Enfin, la condition (3.4) garantit à ce que pour chaque opération O_j , tous ses prédécesseurs ($Preced_{Oj}$) soient exécutés dans les cycles antérieurs à celui de O_j. En d'autres termes, si $x_{ik} = x_{jl} = 1$, alors $k < l$.

Appliquons maintenant la formulation de la programmation linéaire entière pour le problème de l'ordonnancement à l'exemple indiqué dans la **(Fig3.4).**

Du fait que le chemin de données contient 4 types différents d'opérations (multiplication, addition, soustraction et comparaison), nous avons besoin de quatre types d'unités fonctionnelles.

Soient C_m , C_a , C_s et C_c les coûts respectifs d'utilisation du multiplieur, de l'additionneur, du soustracteur et du comparateur.

Soient N_m , N_a , N_s et N_c les nombres de multiplieurs, d'additionneurs, de soustracteurs et de comparateurs, respectivement sollicités dans l'ordonnancement.

La formulation du problème d'ordonnancement serait :

$$\textit{Minimiser}\ (C_m{*}N_m + C_a{*}N_a + C_s * N_s + C_c * N_c) \qquad (3.5)$$

sous les contraintes :

$x_{11}=1$; $x_{21}=1$; $x_{31}+x_{32}=1$; $x_{41}+x_{42}+x_{43}=1$; $x_{52}=1$; $x_{62}+x_{63}=1$; $x_{73}=1$; $x_{84}=1$;

$x_{92}+x_{93}+x_{94}=1$; $x_{10,1}+x_{10,2}+x_{10,3}=1$; $x_{11,2}+x_{11,3}+x_{11,4}=1$; ***(Cf 3.2)***

$x_{11}+x_{21}+x_{31}+x_{41} \leq N_m$; $x_{32}+x_{42}+x_{52}+x_{62} \leq N_m$; $x_{43}+x_{63} \leq N_m$;

$x_{10,1} \leq N_a$; $x_{92}+x_{10,2} \leq N_a$; $x_{93}+x_{10,3} \leq N_a$; $x_{94} \leq N_a$;

$x_{73} \leq N_s$; $x_{84} \leq N_s$;

$x_{11,2} \leq N_c$; $x_{11,3} \leq N_c$; $x_{11,4} \leq N_c$; *(Cf 3.3)*

$x_{11}-2x_{52} \leq -1$; $x_{21}-2x_{52} \leq -1$;

$x_{31}+2x_{32}-2x_{62}-3x_{63} \leq -1$;

$2x_{52}-3x_{73} \leq -1$;

$2x_{62}+3x_{63}-4x_{84} \leq -1$; $3x_{73}-4x_{84} \leq -1$;

$x_{41}+2x_{42}+3x_{43}-2x_{92}-3x_{93}-4x_{94} \leq -1$;

$x_{10,1}+2x_{10,2}+3x_{10,3}-2x_{11,2}-3x_{11,3}-4x_{11,4} \leq -1$; *(Cf 3.4)*

Supposons que $C_m=2$ et $C_a=C_s=C_c=1$. La fonction objective (3.5) est alors minimisée tout en satisfaisant toutes les contraintes pour les valeurs suivantes :

$N_m=2$, $N_a=N_s=N_c=1$,

$x_{11}=x_{21}=x_{32}=x_{43}=x_{52}=x_{63}=x_{73}=x_{84}=x_{94}=x_{10,2}=x_{11,4}=1$, et tous les autres x_{ij} égaux à 0.

Ces résultats donnent alors l'ordonnancement indiqué dans la Fig 3.6 dans laquelle l'opération O_i est ordonnancée dans le cycle j si et seulement si $x_{ij} = 1$.

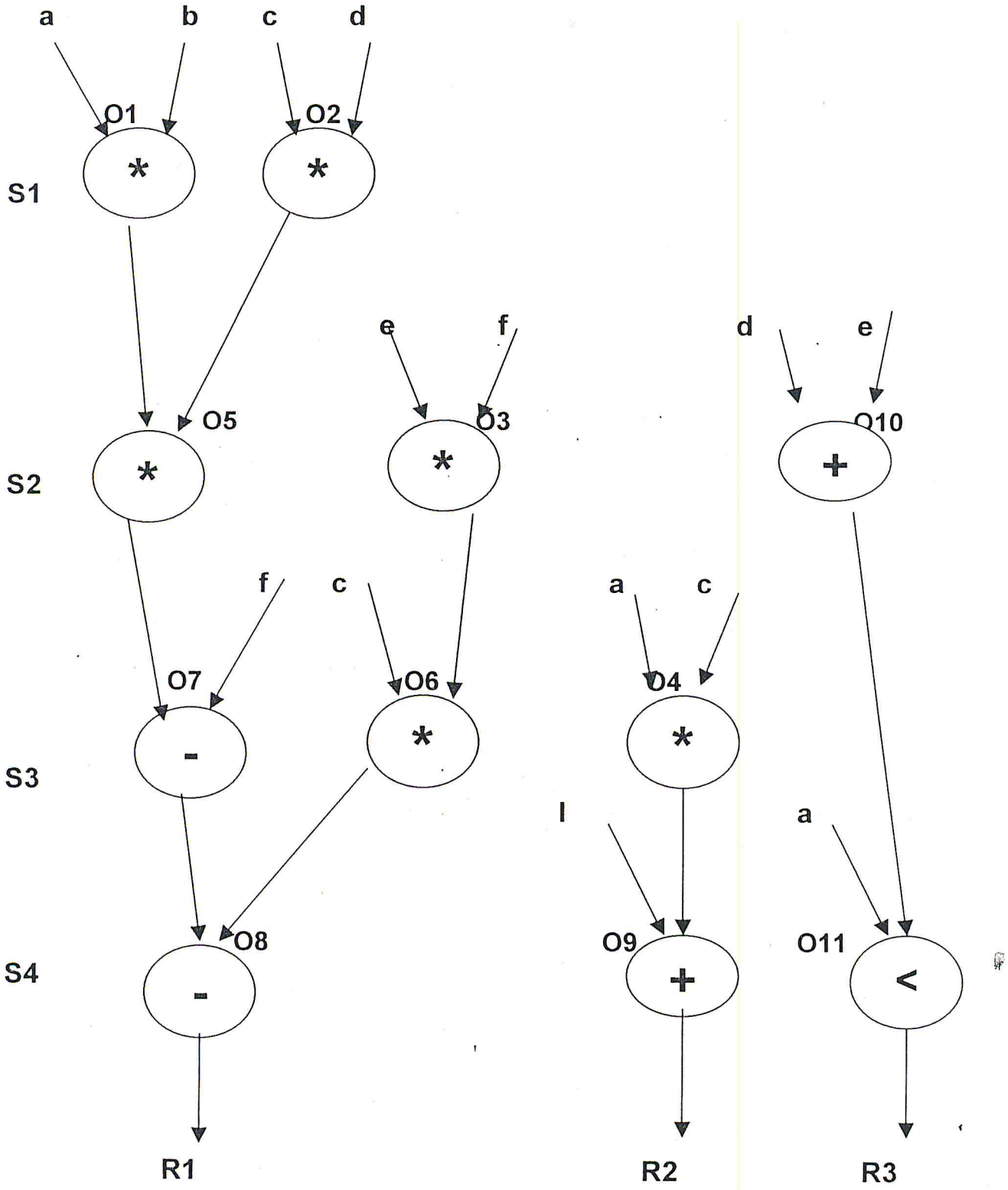


Fig.3.6. Ordonnancement final

Nous terminerons cette phase par les remarques importantes suivantes :

- la taille de la formulation de la programmation linéaire entière croît rapidement avec le nombre de cycles. Si nous augmentons par exemple ce nombre par 1 seulement, nous aurons n variables x_{ij} supplémentaires dans les inégalités puisque nous devons considérer le cycle ajouté pour chaque opération
- le nombre d'inégalités augmente aussi rapidement dès lors que la taille du chemin de données devient conséquente.

CHAPITRE 4

DEPENDANCE DES INSTRUCTIONS

4.1 INTRODUCTION :

Nous avons présenté au troisième chapitre les techniques d'ordonnancement de base, tout comme nous avions indiqué qu'une méthode adéquate doit être adoptée. Nous avons en particulier montré pourquoi la programmation linéaire entière ne peut être envisagée. L'ordonnancement ,qui consiste dans notre cas à optimiser le degré du parallélisme des instructions, est tributaire de la dépendance des instructions.

La résolution de ce problème donnerait un ordonnancement partiel et non final du fait que l'implémentation de l'algorithme par un ASIC pourrait être impossible, pour des raisons de surface par exemple.

De ce fait, une deuxième étape , partant sur la satisfaction des contraintes ,est nécessaire et achèvera l'ordonnancement.

Ces deux étapes seront abordées dans les deux chapitres suivants.

4.2 CONDITIONS DE BERNSTEIN :

Rappelons que l'objet de notre étude est d'ordonnancer les chemins du flot de données contrôlé en un nombre minimal possible de cycles, et ce, en tenant compte des différentes contraintes. Minimiser le nombre de cycles revient donc à maximiser le parallélisme.

Toutefois, ce parallélisme n'est pas toujours permis, en particulier, et pour des raisons évidentes, lorsque des instructions sont dépendantes, c'est-à-dire lorsque l'exécution d'une instruction donnée dépend d'une (ou de plusieurs) variable(s) qui est (sont) en fait un(des) résultat(s) d'une ou d'autres instructions.

Il apparaît donc logique que le premier pas de l'ordonnancement consiste à déterminer les dépendances des différentes instructions contenues dans un chemin de données donné. De manière informelle, une instruction I_j dépend d'une instruction I_i s'il existe au moins une variable dans l'instruction I_j qui représente une donnée pour cette instruction et une sortie (ou résultat)

pour l'instruction I_i. Dans ce cas, il faudrait alors exécuter I_i, puis, une fois le résultat obtenu, exécuter I_j. **Bernstein** a défini cette dépendance d'instructions de manière plus formelle, comme indiqué ci-après.

Soient **inst1** et **inst2** deux instructions. On définit deux fonctions **R** et **W** telles que :

R(inst) est l'ensemble des variables de l'instruction inst non mises à jour par inst.

W(inst) est l'ensemble des variables de l'instruction inst qui sont mises à jour par inst.

Deux instructions **inst1** et **inst2** peuvent s'exécuter en parallèle si et seulement si les conditions suivantes sont vérifiées :

- **R(inst1) ∩ W(inst2) = ∅**
- **R(inst2) ∩ W(inst1) = ∅**
- **W(inst1) ∩ W(inst2) = ∅**

Exemple :

Début

x=a+b ; (inst1)

y=c+d ; (inst2)

z=x+y ; (inst3)

Fin

Les conditions de Bernstein donnent alors :

R(inst1)={a,b} ; W(inst1)={x}

R(inst2)={c,d} ; W(inst2)={y}

R(inst3)={x,y} ; W(inst3)={z}

Du fait que :

R(inst1) ∩ W(inst2) =∅, R(inst2) ∩ W(inst1) =∅ et W(inst1) ∩ W(inst2) =∅, les instructions **inst1** et **inst2** peuvent donc s'exécuter en parallèle.

Par contre, du fait que **R(inst3) ∩ W(inst1) = {x},** les instructions **inst1** et **inst3** ne peuvent s'exécuter en parallèle (inst3 doit s'exécuter une fois **inst1** exécutée).

Le même raisonnement peut être fait pour les instructions **inst2** et **inst3.** Du fait que **R(inst3) ∩ W(inst2) = {y},** l'instruction **inst3** ne peut s'exécuter qu'une fois **inst2** exécutée.

4.3 IMPACT DE LA DEPENDANCE DES INSTRUCTIONS SUR L'ORDONNANCEMENT :

Le nombre de cycles d'exécution d'un chemin de données dépend beaucoup plus de la nature des instructions qui y sont incluses ainsi que d'autres contraintes, plutôt que de sa taille.

Ainsi, des millions d'instructions indépendantes et non assujetties à aucune contrainte peuvent s'exécuter en un seul cycle, alors que des instructions d'un chemin de données de faible taille mais fortement dépendantes peuvent nécessiter plusieurs cycles d'exécution.

Aussi, il est clair de déterminer de manière correcte les dépendances des instructions afin de ne pas augmenter inutilement le nombre de cycles d'exécution.

4.4 TRAITEMENT DU PREMIER ORDONNANCEMENT

L'estimation de la consommation de la puissance dynamique d'un circuit **CMOS** dépend, en plus de plusieurs autres paramètres, de la manière dont sont agencées les portes logiques du circuit, c'est-à-dire de leurs dépendances les unes des autres.

Aussi, un traitement a été fait dans [3], [4] et [5] pour déterminer le niveau de chaque porte logique.

Les portes logiques du même niveau i peuvent (sous certaines conditions) opérer en parallèle car indépendantes.

La sortie d'une porte logique de niveau k n'est stable et entièrement déterminée qu'après que toutes les portes (de niveau inférieur à k) contribuant dans la détermination des entrées de la porte considérée aient généré des sorties stables. Les niveaux des portes sont déterminés comme suit :

- niveau (porte G)=1 si toutes les entrées de G sont primaires, c'est-à-dire ne sont pas des sorties d'autres portes
- niveau (porte G)=1+k ; k étant le maximum des niveaux des portes "alimentant " G, c'est-à-dire celles dont la sortie constitue une entrée pour G

En faisant l'analogie des portes logiques avec les instructions et les niveaux des portes avec les cycles d'exécution des instructions, nous avions adapté ce traitement pour déterminer les dépendances des instructions selon le schéma de la **(Fig. 4.1).**

Ainsi, les niveaux des portes obtenus par le traitement ne sont autres que les cycles des instructions correspondantes.

Notons toutefois que cette adaptation n'est pas directe et nécessite des traitements supplémentaires.

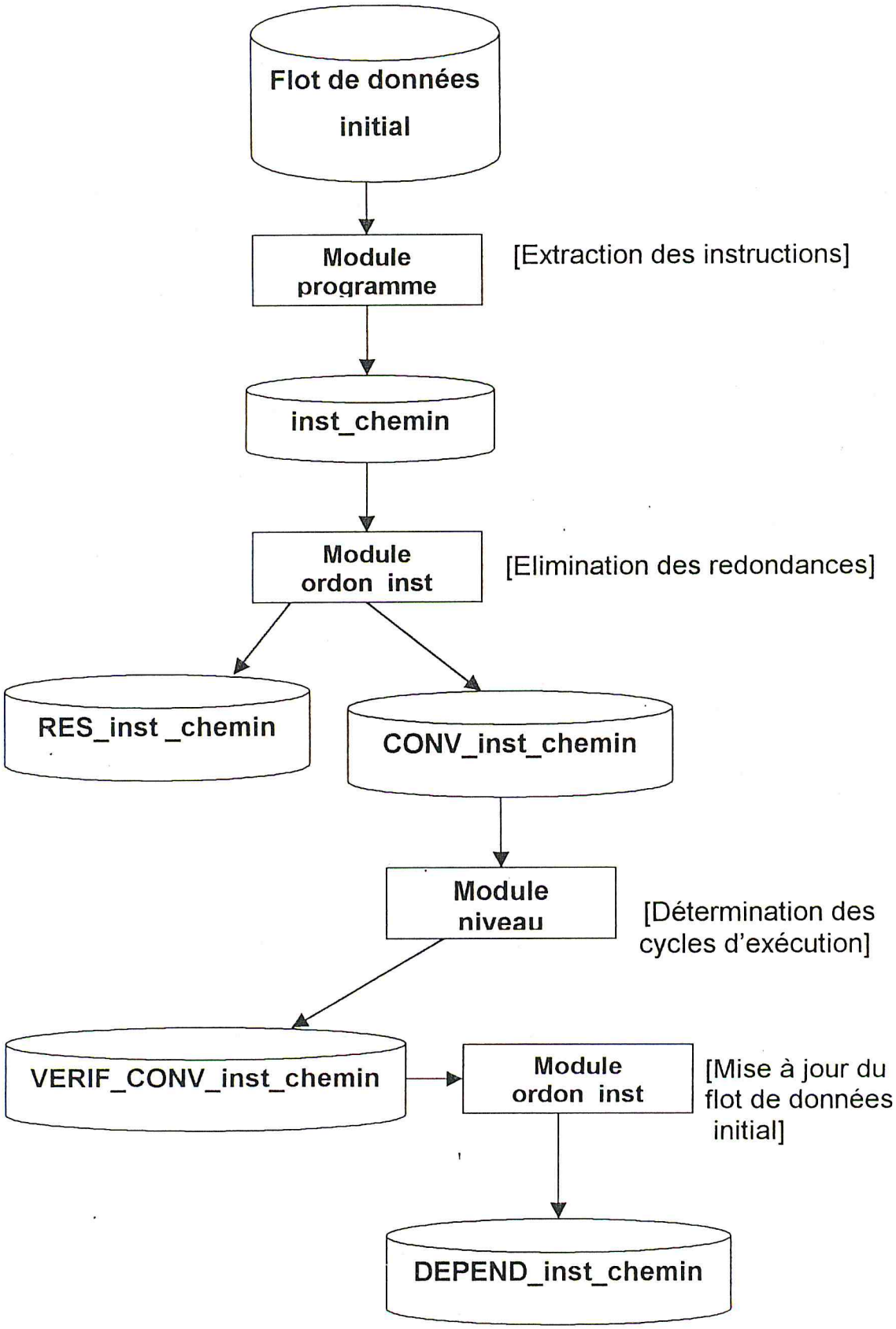


Fig.4.1 Détermination des dépendances des instructions

Notons que :

Flot de données initial "chemin" : fichier indiquant les flots de données simples à ordonnancer.

Inst_chemin : fichier regroupant uniquement les instructions de l'un des flots de données à ordonnancer.

Res_inst_chemin : fichier regroupant les instructions qui sont dans le fichier inst_chemin après avoir transformé les noms de certaines variables pour le besoin.

Conv_inst_chemin : fichier décrivant les instructions dans le format *SPOT* pour déterminer les dépendances des instructions.

Verif_conv_inst_chemin: fichier indiquant les dépendances des instructions.

Depend_inst_chemin : fichier contenant les résultats de la 1ère étape de l'ordonnancement.

Le schéma de la (**Fig.4.1**) décrit les différentes étapes ou modules nécessaires à la détermination des dépendances des instructions.

Ces différents modules représentés dans ce schéma seront décrits dans ce qui suit.

En effet, nous présentons brièvement les étapes nécessaires pour notre travail dans la première partie de l'ordonnancement :

1. **Détecter toutes les instructions ayant des noms de variables de sortie identiques**
2. **Modifier ces noms de variables de manière à ce que tous les noms des variables de sortie soient différents**
3. **Mettre à jour les noms de variables des données qui constituent aussi des variables de sortie pour d'autres instructions et dont les noms ont été modifiés**

4. **Adapter les données à celles de la détermination des niveaux**
5. **Appliquer la procédure de détermination des niveaux des portes**
6. **Transformer les résultats en instructions et cycles**
7. **Mise à jour des noms des variables modifiés**

Nous allons maintenant décrire nos différents algorithmes que nous avons conçus pour la détermination des dépendances entre les instructions.

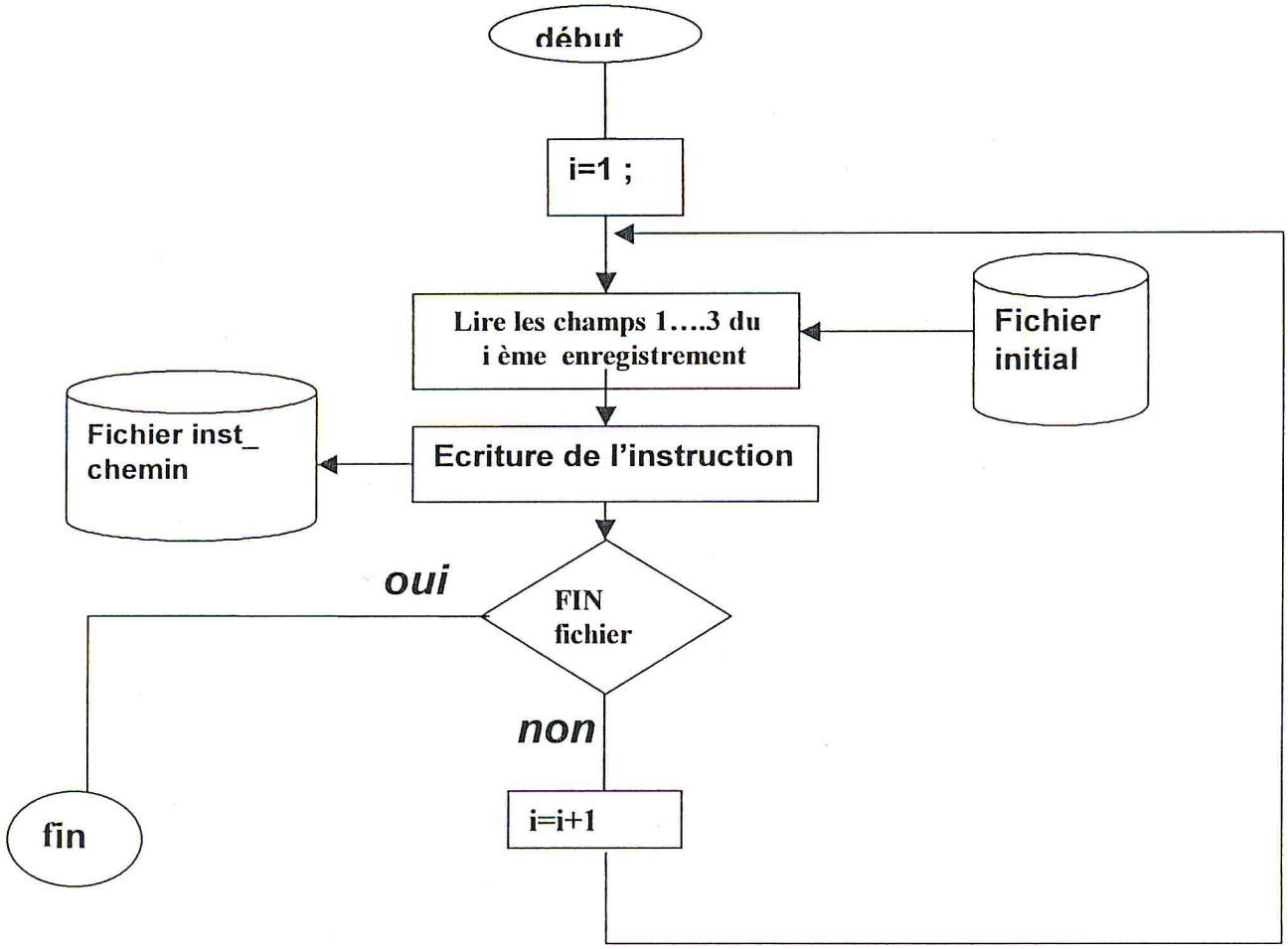
4.5 LES ALGORITHMES DE LA PREMIERE PARTIE :

On a conçu dans cette partie de détection des dépendances des instructions, trois modules « **programme** », « **ord_inst** » et « **niveau** » .

Nous allons décrire leurs algorithmes dans ce qui suit :

ALGO 4.1: du module « programme » : [extraction des instructions]

Organigramme del'ALGO 4.1 :



Algorithme de l'ALGO 4.1 :

début

1 : ouvrir le fichier initial.

2 : atteindre le premier enregistrement.

3 : lecture du premier champ.

4 : lecture du deuxième champ.

5 : lecture du troisième champ.

6 : lecture du champ qui contient l'instruction.

7 : écriture de l'instruction dans un autre fichier d'instruction.

8 : lecture du champ suivant.

9 : si le contenu de ce champs est « fin » allez a 11

10 :sinon, allez à 8 :.

11 : si non fin de fichier allez à 3 (pour l'enregistrement suivant)

12 : si fin de fichier alors fin du traitement

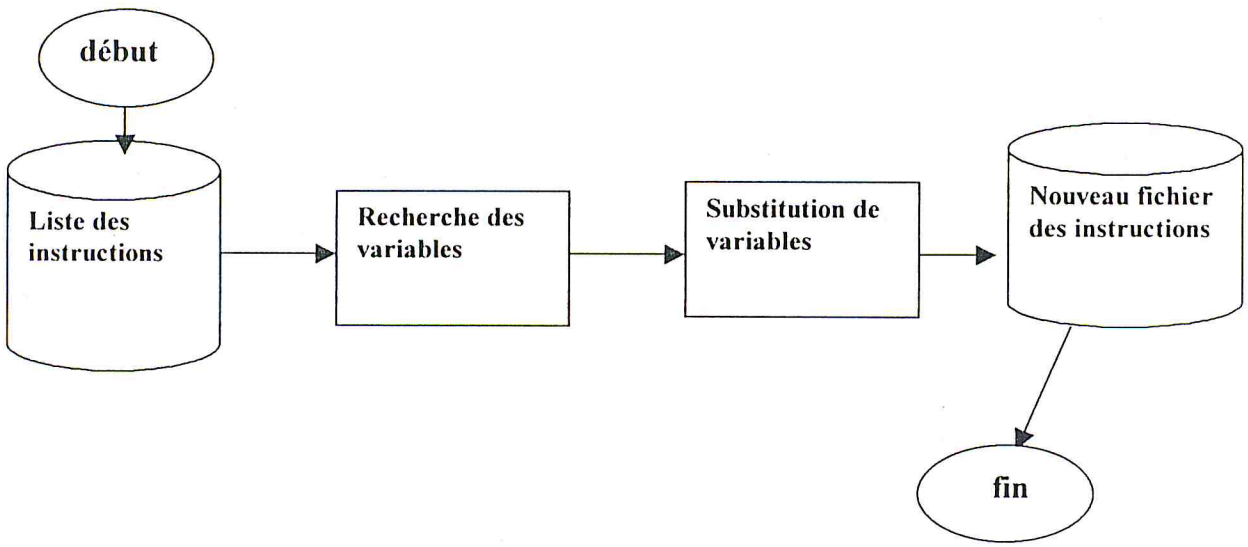
fin

ALGO 4.2 du module « ordont inst » : [Elimination de redondance]

Ce programme se décompose en 2 parties

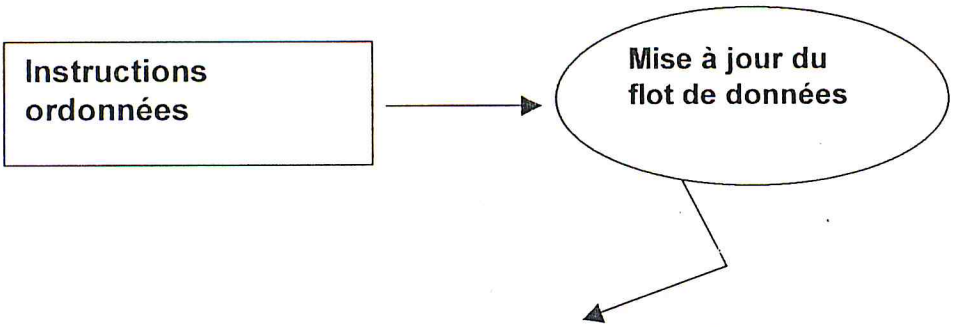
La première consiste à éliminer les redondances des variables d'entrées des instructions. On fait alors la substitution des variables communes :

- 1. considérer les variables qui sont avant le signe égal (=) :
- 2. substitution éventuelle des variables qui sont après le signe égal (=).



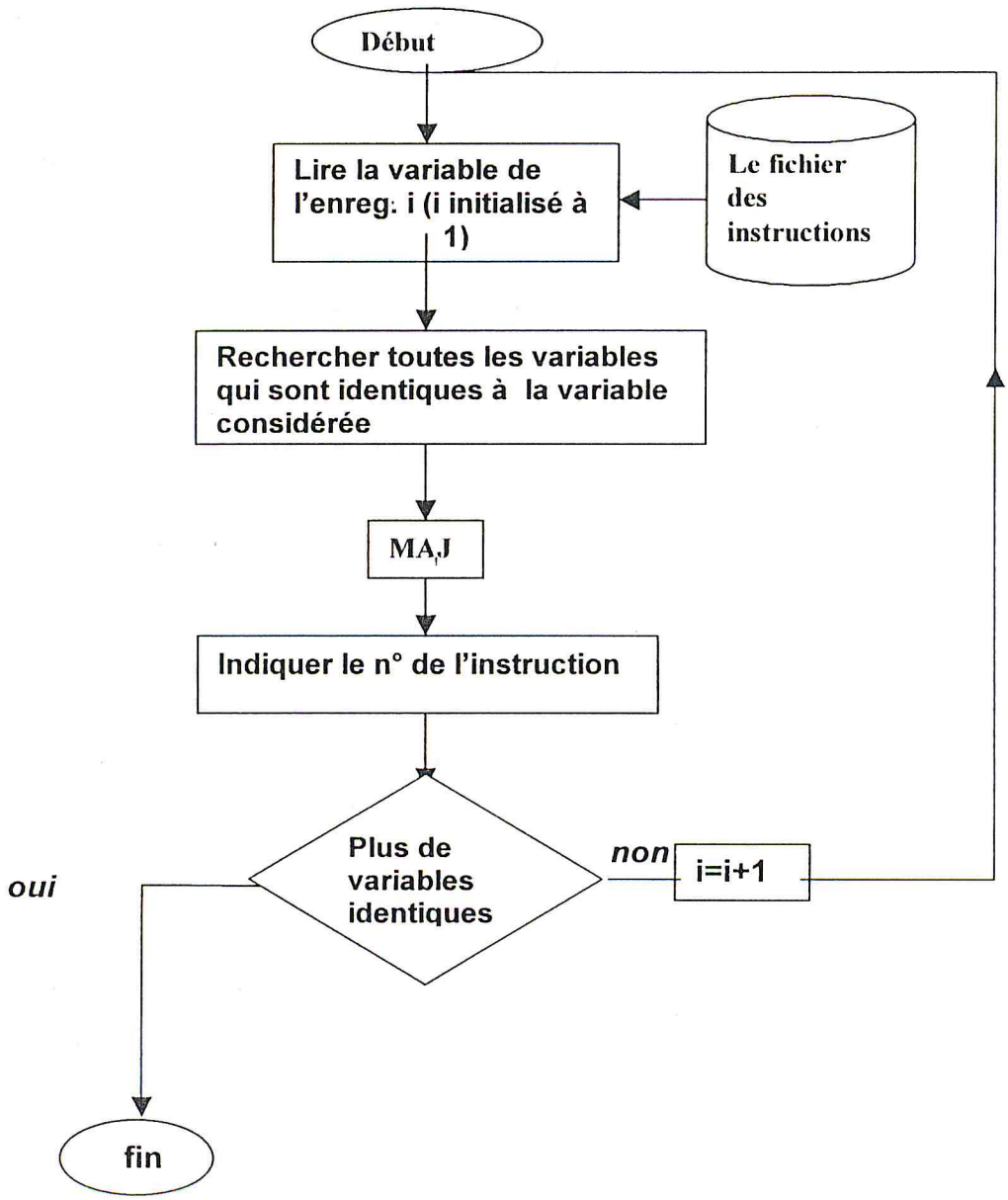
La deuxième consiste à mettre à jour tout cycle d'exécution des instructions dans le flot de données initial

L'algorithme de ces deux parties est décrit ci-après.



Champs des cycles d'exécution pour le flot simple

Organigramme de l'ALGO 4.2 :



Algorithme de l'ALGO 4.2 :

Début

1 : ouvrir le fichier d'instruction

2 : lire tous les enregistrement du fichier

3 : ranger chaque instruction dans une liste avec un numéro

4 : une fois terminé,prendre la première variable avant le signe égale de la première instruction de la liste

5 : rechercher toutes les autres variables avant le signe égale qui sont identiques avec la première

6 : mettre à jour la variable trouvée et mettre son numéro dans l'indice de la première variable et indiquer que l'instruction est traitée

7 : si fin de liste, alors prendre une autre variable non traitée et allez a 5

8 : si toutes les variables sont traités et fin de liste, alors fin du premier traitement.

9 : après avoir exécuté le module niveau, ouvrir le fichier des niveaux des instructions.

10 : mettre à jour ces niveaux dans le flot de données simple de départ.

11 : fin du traitement.

Fin

ALGO 4.3 du module « niveau» :[détermination des cycles d'exécution] :

Dans ce module on détermine les niveaux d'exécution pour chaque porte logique.

Ce module a besoin du fichier des instructions qui est représenté en gras dans le module **ordon_inst Algo4.2** et qui nous donne un format spécial pour la détermination des niveaux.

Les cycles d'exécution

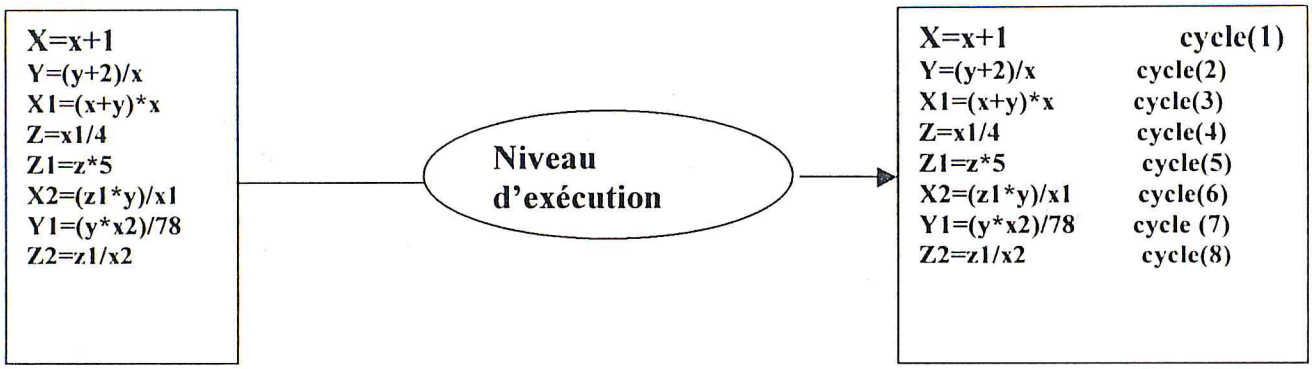
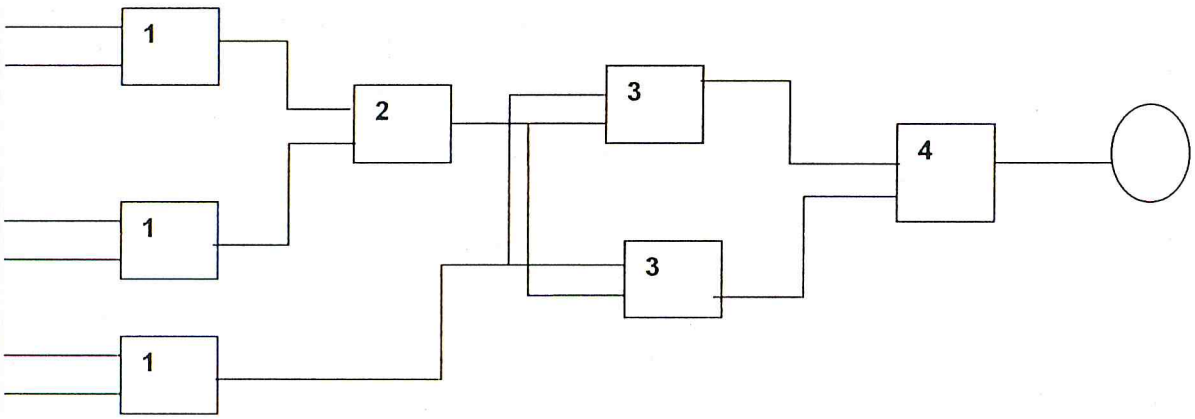


Figure – 4.2- exemple de détermination des cycles

Exemple illustrant la méthode « NIVEAU » :



Les niveaux des portes logiques sont déterminés selon l'algorithme donné au §4.4 en page40. Par la suite, une transformation des portes et des niveaux en instructions et cycles d'exécution, respectivement, est effectuée.

4.6 DISCUSSION DES RESULTATS A TRAVERS UN EXEMPLE

Avant de conclure ce chapitre, nous allons illustrer la détermination de la dépendance des instructions à travers l'exemple du flot de données décrit ci - après qui est extrait a travers un programme algorithmique après avoir exécuté le pré-ordonnancement qui nous donne les flots de données simples.

```
1   1   0   x=a+d
3   4   fin
2   1   0   y=x+1
5   fin
3   1   0   h=x+y
1   6   8   fin
4   1   0   z=a+x
1   fin
5   1   0   x=z*4
2   8   fin
6   1   0   l=h/5
13   fin
7   1   0   f=h-z
24   15   fin
8   1   0   z=a*y
9   3   5   fin
9   1   0   y=l/b
8   14   16   20   fin
10   1   0   h=z*f
16   fin
```

```
11   1   0   f=h/2
19    fin
12   1   0   r=g+c
3   4   fin
13   1   0   y=x+r
6   25   fin
14   1   0   w=x+l
20   9   fin
15   1   0   delta=x*z
18   7   fin
16   1   0   delta=delta+3
9   10   fin
17   1   0   h=delta/5
18   19    fin
18   1   0   u=delta-z
17   fin
19   1   0   delta=y+delta
17   11    fin
20   1   0   y=l/b
9   14   fin
21   1   0   h=b*f
25   28   fin
22   1   0   f=h+6
0    fin
23   1   0   delta=d-f
32   41    fin
24   1   0   y=x+5
7   fin
25   1   0   h=delta+y
31   26    fin
```

```
26   1   0   z=h/5
25   fin
27   1   0   t=z*4
29   fin
28   1   0   t=t+5
31   fin
29   1   0   s=h-t
12   19   34   fin
30   1   0   e=f*s
9   10   11   fin
31   1   0   y=s*4
38   40   fin
32   1   0   h=e*f
18   fin
33   1   0   e=h/2
40   fin
34   1   0   x=a+f
31   44   fin
35   1   0   y=e+delta
42   43   fin
36   1   0   delta=x+delta
11   26   fin
37   1   0   z=e/s
16   fin
38   1   0   x=s+h
40   39   fin
39   1   0   l=h/5
38   fin
40   1   0   f=h-a
33   38   fin
```

```
41   1   0   z=a*f
0   fin
42   1   0   e=d/l
35   fin
43   1   0   s=delta*f
35   fin
44   1   0   h=s/5
37   fin
45   1   0   h=h*z
0   fin
```

Fig.4.3 Flot de données initial

Pour les besoins du traitement, seules les instructions (parmi toutes les informations contenues dans le fichier **(FIG4.3**) sont extraites et sauvegardées dans un autre fichier **FIG 4.4**. Cette tâche est effectuée grâce au module **programme « ALGO 4.1 »**.

Le fichier résultant est alors celui indiqué ci-après :

```
x=a+d
y=x+1
h=x+y
z=a+x
x=z*4
l=h/5
f=h-z
z=a*y
y=l/b
h=z*f
f=h/2
r=g+c
y=x+r
```

```
w=x+l
delta=x*z
delta=delta+3
h=delta/5
u=delta-z
delta=y+delta
y=l/b
h=b*f
f=h+6
delta=d-f
y=x+5
h=delta+y
z=h/5
t=z*4
t=t+5
s=h-t
e=f*s
y=s*4
h=e*f
e=h/2
x=a+f
y=e+delta
delta=x+delta
z=e/s
x=s+h
l=h/5
f =h-a
z=a*f
e=d/l
s=delta*f
```

```
h=s/5
h=h*z
```

Fig.4.4 Le fichier inst_chemin

Afin d'exploiter une procédure de détermination des niveaux des portes logiques d'un circuit pour la détermination des dépendances des instructions d'un flot de données, nous avions opté pour certaines transformations, dont celle de donner des noms différents pour toutes les instructions. Ainsi, l'exemple donné dans la **FIG 4.4** est transformé au grâce au module **ordon_inst** « **ALGO 4.2** »indiqué dans la (**FIG 4.5**).

```
x=a+d
y=x+1
h=x+y
z=a+x
x1=z*4
l=h/5
f=h-z
z1=a*y
y1=l/b
h1=z1*f
f=h1/2
r=g+c
y2=x1+r
w=x1+l
delta=x1*z1
delta1=delta+3
h3=delta1/5
u=delta1-z1
delta2=y2+delta1
```

```
y3=l/b
h4=b*f2
f3=h4+6
delta3=d-f3
y4=x1+5
h5=delta3+y4
z2=h5/5
t=z2*4
t1=t+5
s=h5-t1
e=f3*s
y5=s*4
h6=e*f3
e1=h6/2
x2=a+f3
y6=e1+delta3
delta4=x2+delta3
z3=e1/s
x3=s+h6
l1=h6/5
f4 =h6-a
z4=a*f4
e2=d/l1
s1=delta4*f4
h7=s1/5
h8=h7*z4
```

Fig.4.5 Fichier RES_inst

Le flot donné dans la **FIG4.5** est par la suite transformé en un autre format (format "**SPOT**") tel qu'indiqué dans la **Figure 4.6,** pour la détermination des niveaux des portes logiques d'un circuit. Notons qu'il y' a alors analogie entre *instructions* et *portes*, *cycles* et *niveaux*, *flot de données* et *circuit*.

```
a   d % x % a+d
x   1 % y % x+1
x   y % h % x+y
a   x % z % a+x
z   4 % x1 % z*4
h   5 % l % h/5
h   z % f % h-z
a   y % z1 % a*y
l   b % y1 % l/b
z1   f % h1 % z1*f
h1   2 % f % h1/2
g    c % r % g+c
x1   r % y2 %x1+r
x1   l % w % x1+l
x1   z1 % delta % x1*z1
delta   3 % delta1 % delta+3
delta1   5 % h3 % delta1/5
delta1   z1 % u % delta1-z1
y2   delta1 % delta2 % y2+delta1
l   b % y3 % l/b
b   f2 % h4 % b*f2
h4    6 % f3 % h4+6
d    f3 % delta3 % d-f3
x1   5 % y4 % x1+5
```

```
delta3   y4 % h5 % delta3+y4
h5   5 % z2 % h5/5
z2   4 % t % z2*4
t   5 % t1 % t+5
h5   t1 % s % h5-t1
f3   s % e % f3*s
s   4 % y5 % s*4
e   f3 % h6 % e*f3
h6   2 % e1 % h6/2
a   f3 % x2 % a+f3
e1   delta3 % y6 % e1+delta3
x2   delta3 % delta4 % x2+delta3
e1   s % z3 % e1/s
s   h6 % x3 % s+h6
h6   5 % l1 % h6/5
h6   a % f4 % h6-a
a   f4 % z4 % a*f4
d   l1 % e2 % d/l1
delta4   f4 % s1 % delta4*f4
s1   5 % h7 % s1/5
h7   z4 % h8 % h7*z4
```

Fig.4.6 Format "SPOT" du fichier CONV_inst_chemin

Notons que :

- le 1er champ indique les entrées d'une porte logique
- le 2ème champ indique la sortie de la porte logique
- le 3ème champ indique le comportement de la porte logique

Cette transformation est effectuée grâce au module **ordon_inst** « **ALGO 4.2** »indiqué dans la **FIG 4.1**.

Les résultats de la procédure incluse dans le module **Niveau « ALGO 4.3»** sont alors ceux donnés dans la **FIG 4.7**

```
Level of Gate 12
1
Level of Gate 1
1
Level of Gate 4
2
Level of Gate 2
2
Level of Gate 8
3
Level of Gate 5
3
Level of Gate 3
3
Level of Gate 24
4
Level of Gate 15
4
Level of Gate 13
4
Level of Gate 7
4
Level of Gate 6
4
```

```
Level of Gate 20
5
Level of Gate 16
5
Level of Gate 14
5
Level of Gate 10
5
Level of Gate 9
5
Level of Gate 19
6
Level of Gate 18
6
Level of Gate 17
6
Level of Gate 11
6
Level of Gate 21
7
Level of Gate 22
8
Level of Gate 34
9
Level of Gate 23
9
Level of Gate 36
10
Level of Gate 25
10
```

```
Level of Gate 26
11
Level of Gate 27
12
Level of Gate 28
13
Level of Gate 29
14
Level of Gate 31
15
Level of Gate 30
15
Level of Gate 32
16
Level of Gate 40
17
Level of Gate 39
17
Level of Gate 38
17
Level of Gate 33
17
Level of Gate 43
18
Level of Gate 42
18
Level of Gate 41
18
Level of Gate 37
18
```

```
Level of Gate 35
18
Level of Gate 44
19
Level of Gate 45
20
```

Fig.4.7 Fichier VERIF_CONV_inst_chemin

Enfin, ce format est transformé en termes de cycles et d'instructions (voir FIG 4.8) grâce au module **ordon_inst** « **ALGO4.2** »indiqué dans la FIG 4.1.

```
1  1  1  x=a+d
3  4  fin
2  1  2  y=x+1
5  fin
3  1  3  h=x+y
1  6  8  fin
4  1  2  z=a+x
1  fin
5  1  3  x=z*4
2  8  fin
6  1  4  l=h/5
13  fin
7  1  4  f=h-z
24  15  fin
8  1  3  z=a*y
9  3  5  fin
```

```
9   1   5   y=l/b
8   14   16   20   fin
10   1   5   h=z*f
16   fin
11   1   6   f=h/2
19   fin
12   1   1   r=g+c
3   4   fin
13   1   4   y=x+r
6   25   fin
14   1   5   w=x+l
20   9   fin
15   1   4   delta=x*z
18   7   fin
16   1   5   delta=delta+3
9   10   fin
17   1   6   h=delta/5
18   19   fin
18   1   6   u=delta-z
17   fin
19   1   6   delta=y+delta
17   11   fin
20   1   5   y=l/b
9   14   fin
21   1   7   h=b*f
25   28   fin
22   1   8   f=h+6
0   fin
23   1   9   delta=d-f
32   41   fin
```

```
24   1   4   y=x+5
7   fin
25   1   10   h=delta+y
31   26   fin
26   1   11   z=h/5
25   fin
27   1   12   t=z*4
29   fin
28   1   13   t=t+5
31   fin
29   1   14   s=h-t
12   19   34   fin
30   1   15   e=f*s
9   10   11   fin
31   1   15   y=s*4
38   40   fin
32   1   16   h=e*f
18   fin
33   1   17   e=h/2
40   fin
34   1   9   x=a+f
31   44   fin
35   1   18   y=e+delta
42   43   fin
36   1   10   delta=x+delta
11   26   fin
37   1   18   z=e/s
16   fin
38   1   17   x=s+h
40   39   fin
```

```
39   1   17   l=h/5
38   fin
40   1   17   f=h-a
33   38   fin
41   1   18   z=a*f
0   fin
42   1   18   e=d/l
35   fin
43   1   18   s=delta*f
35   fin
44   1   19   h=s/5
37   fin
45   1   20   h=h*z
0   fin
```

Fig.4.8. Fichier DEPAND_inst_chemin

4.7 CONCLUSION :

Ce chapitre a porté sur la résolution du problème de dépendances des instructions, et constitue une étape seulement de l'ordonnancement.

En effet, nous verrons au chapitre suivant que la nature des problèmes et l'existence de contraintes nous imposent une autre étape de l'ordonnancement.

CHAPITRE 5

SATISFACTION DES CONTRAINTES

5.1INTRODUCTION :

Le premier ordonnancement ne suffit pas car l'existence des contraintes imposées par l'utilisateur nous oblige de faire un deuxième ordonnancement portant sur la satisfaction des contraintes, et que nous allons décrire dans ce qui suit.

5.2 NECESSITE DES CONTRAINTES-UTILISATEUR :

Dans le cas où les différentes instructions ne sont assujetties à aucune contrainte, l'ordonnancement décrit dans le chapitre précédent constitue un ordonnancement final.

Cependant, en réalité, des contraintes sont imposées par l'utilisateur afin de pouvoir produire une partie opérative de taille raisonnable.

En effet, supposons qu'un millier d'additions sont indépendantes. Leur exécution simultanée nécessiterait alors 1000 additionneurs, ce qui nécessiterait une surface énorme pour l'implémentation du circuit intégré.

Afin de pallier ce problème, des contraintes sont introduites afin de forcer l'exécution de certaines instructions indépendantes dans des cycles différents, pouvant de ce fait utiliser la même ressource physique (additionneur, ...).

A titre d'illustration, nous donnons à partir de la **Fig.5.1**, un ordonnancement d'instructions (**Fig.5.2),** et dans la **Fig 5.3,** l'ordonnancement des mêmes instructions avec l'existence d'une contrainte entre les instructions 2 et 4.

(1) w=a+b ;

(2) x=c+d ;

(3) y=a+c ;

(4) z=b+d ;

Fig.5.1 Exemple d'un flot d'instructions indépendantes

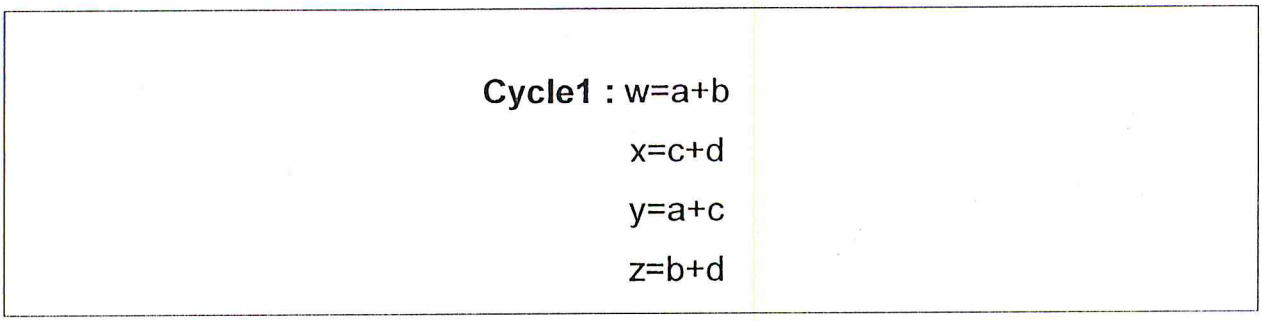


Fig.5.2. Ordonnancement des instructions de la Fig.5.1 (avec aucune contrainte)

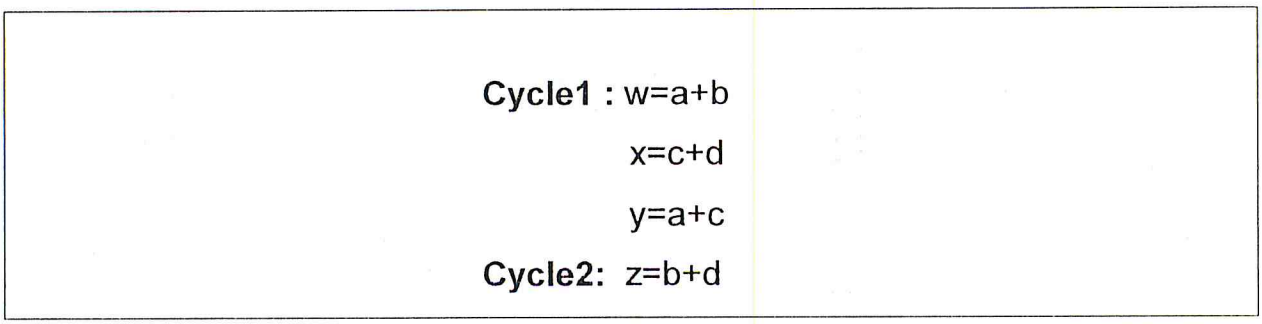


Fig.5.3 Ordonnancement des instructions de la Fig.5.1 (avec existence d'une contrainte)

Ainsi, l'implémentation du flot décrit dans la **Fig.5.1** aboutirait à une partie opérative exécutant le flot en un seul cycle, mais nécessitant 4 additionneurs. En revanche, l'implémentation selon l'ordonnancement indiqué dans la **Fig.5.3** engendrerait 2 cycles d'exécution, mais 3 additionneurs

seulement. La génération des contraintes de l'utilisateur doit donc tenir compte de ce compromis temps – surface.

5.3 IMPACT DE LA SATISFACTION DES CONTRAINTES SUR L'ORDONNANCEMENT :

Comme nous venons juste de le mentionner, il est nécessaire d'ordonnancer des instructions affectées par la même contrainte dans des cycles différents. Cependant, la satisfaction de ces contraintes d'une manière quelconque peut avoir un impact négatif sur les performances de la partie opérative.

Considérons ce problème en nous servant de l'exemple indiqué dans la **Figure 5.4**

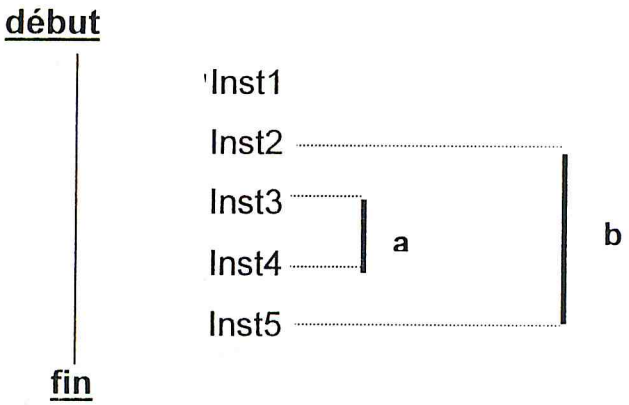


Fig.5.4 Exemple d'un flot de données avec 2 contraintes

La satisfaction des contraintes *a* et *b* nécessite d'ordonnancer les instructions 3 et 4 dans des cycles différents.

Ceci va de même pour les instructions 2 et 5. Ceci peut se faire par l'insertion d'un point de coupure (***cut***) au niveau de l'instruction 3, et un autre au niveau de l'instruction 2. En supposant que les 5 instructions sont indépendantes, on aboutirait à l'ordonnancement indiqué dans la **Figure 5.4**

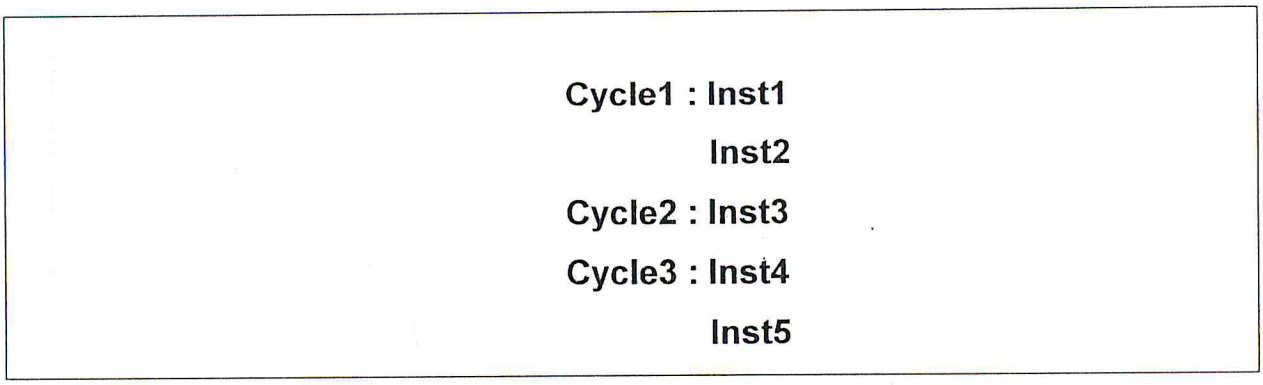


Fig.5.5 Un ordonnancement possible du flot de la Fig.5.4

Un autre ordonnancement possible (indiqué dans la **Fig.5.6)** est obtenu en insérant un *seul* point de coupure au niveau de l'instruction 3.

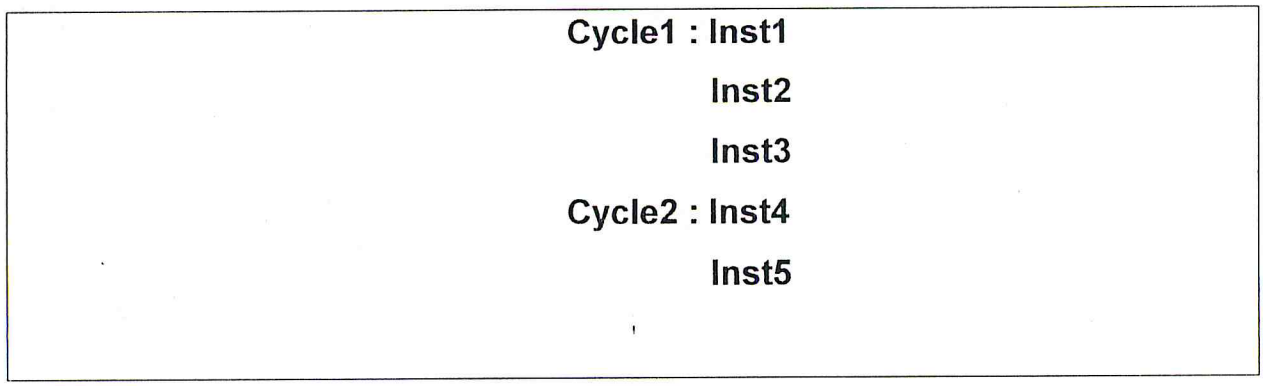


Fig.5.6. Un autre ordonnancement possible du flot de la Fig.5.4

Ainsi, les deux ordonnancements indiqués dans les **Figures 5.5** et **5.6** permettent de lever les contraintes *a* et *b*, à la différence que le deuxième est plus intéressant, puisque aboutissant à 2 cycles seulement.

Dans le cas général, un bon choix d'insertion de points de coupure permettrait de satisfaire toutes les contraintes, tout en obtenant un nombre intéressant de cycles (avec un nombre réduit de points de coupure). En fait, ce problème est combinatoire et nécessite un traitement adéquat pour le résoudre. Ceci est l'objet de ce qui suit.

5 .4 TRAITEMENT DU SECOND ORDONNANCEMENT :

Le problème dont il est question peut être formulé comme suit :

Trouver le nombre minimal de cycles tout en satisfaisant toutes les contraintes.

Nous remarquons que ce problème peut se ramener au problème suivant :

Trouver le nombre minimal de couleurs pour colorer tous les nœuds d'un graphe.

Ce dernier problème n'est autre que celui des graphes colorés et peut se formuler formellement comme suit :

Soit un graphe G=(V,E) ; V={noeuds}; E={arêtes}
Trouver la fonction ***f*** *minimisant* ***K*** $(1 \leq K \leq |V|)$ *telle que:*
$f : V \rightarrow \{1,2, \ldots, K\}$
$f(v_i) \neq f(v_j) \ \forall \ (v_i , v_j) \in E$

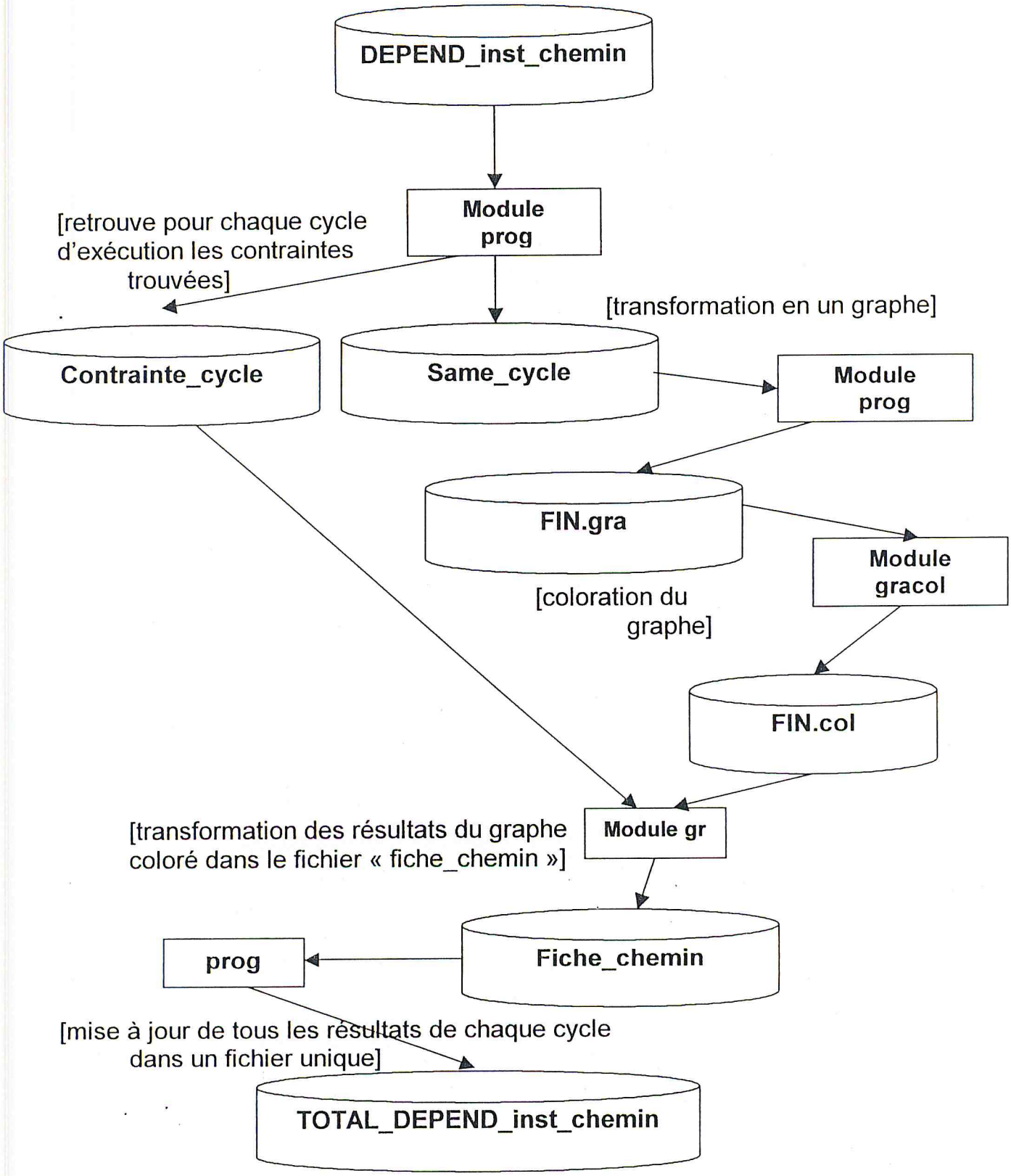
Le problème des graphes colorés est connu NP-complet [6], et jusqu'ici on conjecture qu'il n'existe pas d'algorithme pour le résoudre en un temps polynômial ([6], [7]). Il existe plusieurs techniques de coloration de graphes, dont [8] et [9].

Pour notre part, nous avions adapté à notre problème la technique de coloration de graphes adoptée dans [10], [11] et [12].

Ayant été constaté que le nombre chromatique dépend de l'ordre de coloration des nœuds du graphe, et du fait qu'il n'est pas possible de considérer tous les ordres possibles, la technique utilisée consiste à répartir les nœuds du graphe à travers k ($k \in N$) classes en se basant sur le critère de degrés de nœuds (les nœuds ayant sensiblement le même degré

appartiennent à la même classe), puis à colorer le même graphe en considérant tous les ordres sur les classes (et non les ordres des nœuds), ce qui d'une part assure une coloration possible (le nombre d'arrangements sur les classes est très inférieur à celui sur les nœuds), et d'autre part donne de meilleurs résultats (la solution la plus optimale par rapport à celles obtenues est retenue).

Le traitement de la deuxième partie de l'ordonnancement, sera suivi selon le schéma indiqué dans la **figure 5.7**.


DEPEND_inst_chemin
Module prog
[retrouve pour chaque cycle d'exécution les contraintes trouvées]
[transformation en un graphe]
Contrainte_cycle
Same_cycle
Module prog
FIN.gra
Module gracol
[coloration du graphe]
FIN.col
[transformation des résultats du graphe coloré dans le fichier « fiche_chemin »]
Module gr
prog
Fiche_chemin
[mise à jour de tous les résultats de chaque cycle dans un fichier unique]
TOTAL_DEPEND_inst_chemin

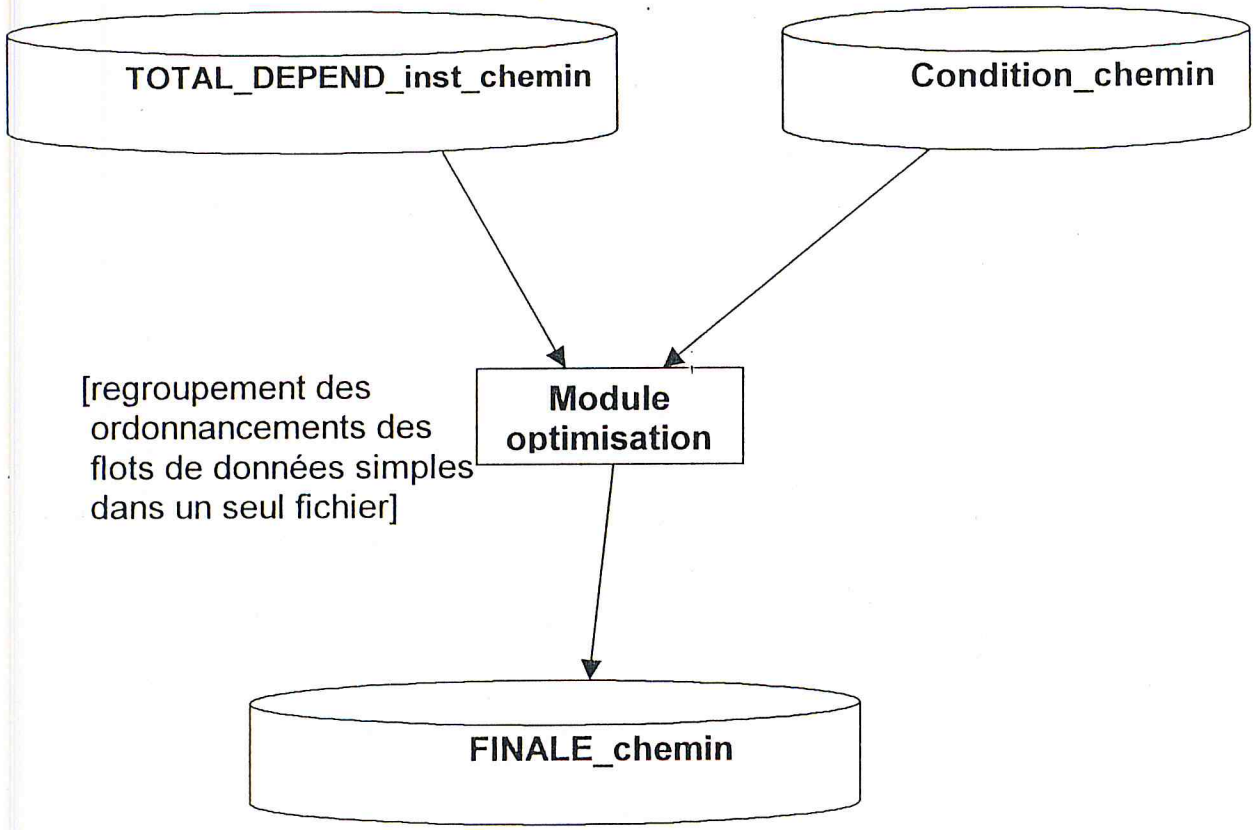


Fig.5.7 Satisfaction des contraintes

Notons que les différents fichiers utilisés sont les suivants:

Contrainte cycle : fichier indiquant les contraintes entre les différentes instructions.

Same cycle : fichier contenant les instructions ayant le même cycle d'exécution après la 1ère étape de l'ordonnancement.

Fin.gra : graphe résultant de la transformation des contraintes et des instructions.

Fin.col : fichier contenant les résultats engendrés par la coloration des nœuds du graphe.

Fiche chemin : fichier contenant les résultats obtenus après la 2ème étape de l'ordonnancement pour un cycle donné.

Totale depend inst chemin : fichier contenant les résultats obtenus après la 2ème étape de l'ordonnancement pour tous les cycles.

Condition chemin : c'est le fichier qui indique les conditions du flot de données contrôlées à ordonnancer.

Final chemin : c'est le fichier qui regroupe l'ordonnancement de tous les flots de données simples avec les conditions d 'exécution (résultat global).

Le schéma de la **Fig. 5.7** décrit les différentes étapes ou modules nécessaires à la deuxième partie de l'ordonnancement, c'est à dire la satisfaction des contraintes.

Ces différents modules représentés dans ce schéma seront décrits dans ce qui suit.

Toutefois, l'adaptation de cette technique à notre problème n'est pas directe, et nécessite des traitements :

1. **Associer pour chaque instruction I_i un nœud n_i**
2. **Générer une arête e_{ij} entre les nœuds n_i et n_j si les instructions I_i et I_j sont assujetties à la même contrainte**
3. **Appliquer l'outil de coloration de graphes**
4. **Transformer les résultats obtenus : les instructions dont les nœuds associés ont été colorés par la même couleur seront ordonnancées au même cycle**
5. **Mise à jour des cycles d'exécution des instructions**

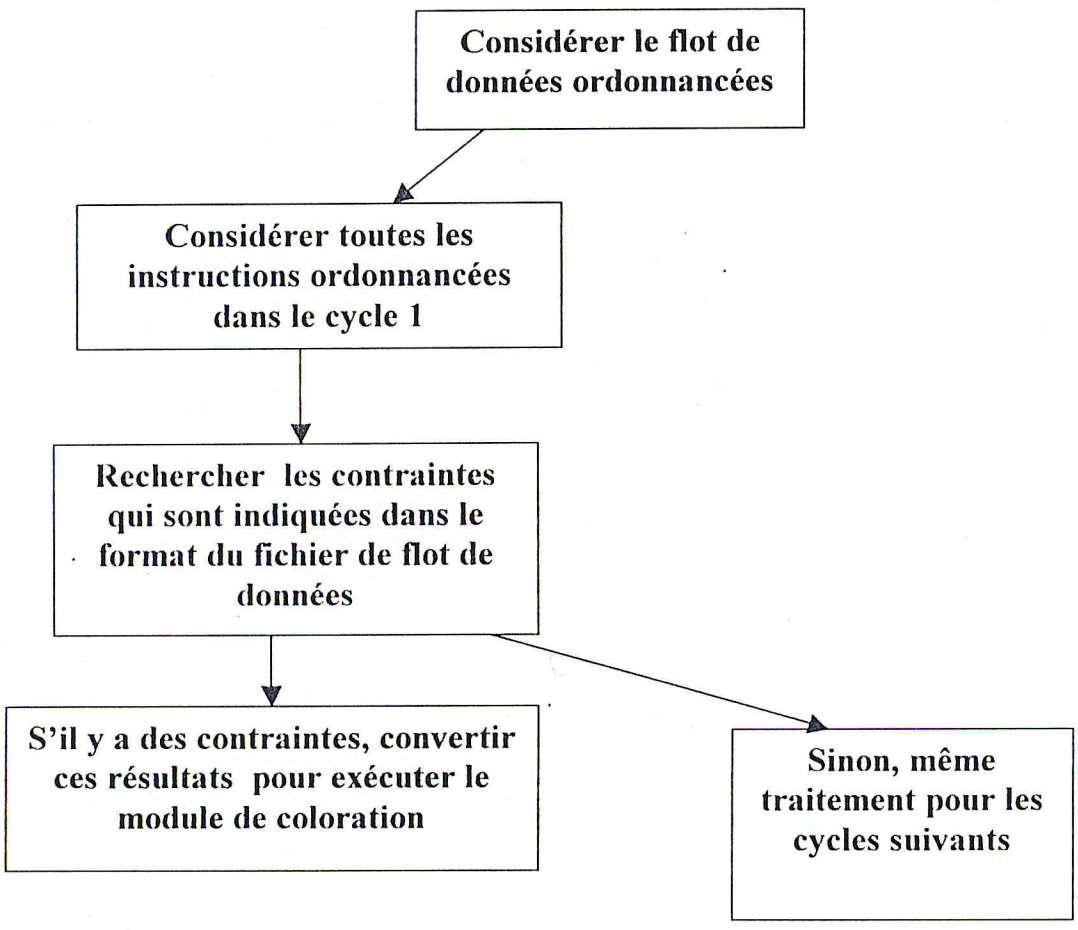
Nous allons décrire, à travers un exemple, ces transformations dans la section qui suit.

5 .5 LES ALGORITHMES DE LA DEUXIEME PARTIE

ALGO 5.1 :du module « prog »[détection des contraintes pour chaque cycle]:
Le rôle de cet algorithme est de vérifier s'il existe des contraintes dans le flot de données ordonnées **(DEPAND_inst_chemin)** obtenu par la partie dépendance des instructions des contraintes de l'utilisateur.

On décrit maintenant l'organigramme qui permet la recherche des contraintes

ORGANIGRAMME : Soit l'organigramme suivant qui concerne le module « prog »



ALGORITHME de l'ALGO 5.1 :

début

1 : ouvrir le flot de données à traiter

2 : rechercher toutes les contraintes trouvées pour un cycle

3 : vérifier ces contraintes qui sont trouvées avec des instructions de ce cycle

4 : s'il n'y a pas des contraintes dans ce cycle, allez à 1 : pour le cycle suivant

5 : s'il y'a des contraintes dans ce cycle, allez à 6

6 : créer un fichier pour placer ces contraintes pour le traitement de graphe coloré « transformation en un graphe»

7 : après avoir terminé le traitement de coloration de graphe et la mise à jour du flot de données, passez au cycle suivant et allez à 2 :

8 : si fin du flot de données, TERMINE.

fin

ALGO5.2 :de module « gr » [transformation du résultat de la coloration]

Dans cet algorithme, on fait le traitement pour transformer les résultats du graphe coloré vers le fichier d'instructions pour un seul cycle traité : associer les instructions et cycles aux nœuds et couleurs respectivement.

Algorithme de l'ALGO 5.2 :

1 : prendre les résultats obtenus dans le fichier du graphe coloré « FIN.col »

2 : à chaque couleur trouvée, rechercher toutes les instructions dont les nœuds associés ont la même couleur.

3 : ces instructions sont ordonnancées dans le même cycle (la mise à jour de leur cycle d'exécution).

4 : allez à 2 pour la couleur suivante.

5 : si fin de couleur, alors FIN du traitement.

ALGO 5.3 du module « gracol »[la coloration des nœuds] :

Le rôle du module « gracol » est la coloration des nœuds suivant une nouvelle technique **[10]** qui consiste à trouver la meilleure coloration du graphe(nombre minimum de couleurs).

ALGO 5.4 du module « optimisation » [regroupement dans un seul fichier de tous les résultats des flots de données traités] :

Ce module, comme nous l'avons conçu, regroupe tous les ordonnancements des chemins simples en un seul ordonnancement. Ce résultat est rangé dans le fichier « **FINALE_chemin** ».

L'algorithme est alors le suivant :

```
pour j=1 jusqu'à N      ( N= le maximum des cycles d'exécution dans tous les
                                                              flots simples)
faire pour i=1 jusqu'à M      ( M = le nombre de tous les flots simples)
    faire  prendre toutes les instructions ordonnées dans le cycle j ;
           pour chaque instruction traitée
           faire ajouter sa condition d'exécution
           fait
    fait
fait
```

5.6 DISCUSSION DES RESULTATS A TRAVERS UN EXEMPLE

Considérons l'exemple de la **Figure 5.8**

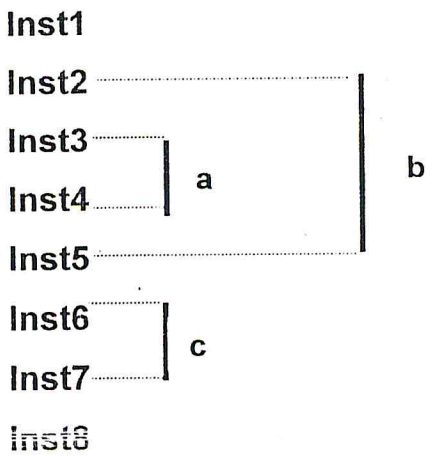


Fig.5.8. Exemple d'un flot de données avec 3 contraintes

Les deux premières transformations donnent le graphe décrit dans la **Figure 5.9.**

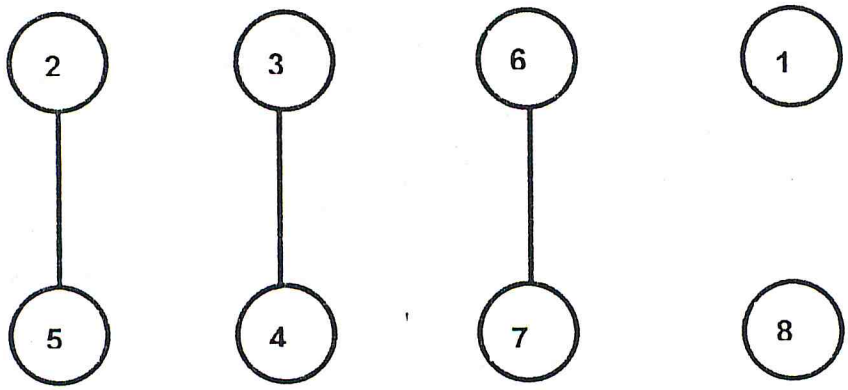


Fig.5.9. Graphe associé aux contraintes indiquées dans la Fig.4.8

Notons que les nœuds correspondent aux instructions indiquées dans la **Figure 5.8.**

L'application de l'outil de coloration de graphes donne le résultat suivant :

Couleur 1 : affectée aux instructions 2, 3, 6, 1, 8

Couleur 2 : affectée aux instructions 4, 5 et 7

La transformation de ces résultats à ceux du problème concerné donne l'ordonnancement possible suivant :

Cycle1 : instructions 1, 2, 3, 6, 8

Cycle2 : instructions 4, 5, 7

Nous allons encore illustrer cette étape par un autre exemple.

Soit le fichier de données indiqué dans la **Fig 5.9.** Ce fichier est le résultat de la première étape de l'ordonnancement (dépendance des instructions). Le nom de ce fichier est **«DEPEND_'nom de fichier '»** .

```
1   1   1   x=a+d
3   4   fin
2   1   2   y=x+1
5  fin
3   1   3   h=x+y
1   6   8   fin
4   1   2   z=a+x
1   fin
5   1   3   x=z*4
2   8   fin
6   1   4   I=h/5
13   fin
7   1   4   f=h-z
24   15   fin
```

```
8   1   3   z=a*y
9   3   5   fin
9   1   5   y=l/b
8   14  16  20   fin
10  1   5   h=z*f
16  fin
11  1   6   f=h/2
19   fin
12  1  1   r=g+c
18 3   4   fin
13  1   4   y=x+r
6   25  fin
14  1   5   w=x+l
20  9   fin
15  1   4   delta=x*z
18  7   fin
16  1    5   delta=delta+3
9   10  fin
17   1    6    h=delta/5
18  19   fin
18  1   6   u=delta-z
17  fin
19  1   6   delta=y+delta
17  11   fin
20  1   5   y=l/b
9   14  fin
21  1   7   h=b*f
25  28   fin
```

```
22   1   8   f=h+6
0   fin
23   1   9   delta=d-f
32   41   fin
24   1   4   y=x+5
7   fin
25   1   10   h=delta+y
8  31   26   fin
26   1   11   z=h/5
25   fin
27   1   12   t=z*4
29   fin
28   1   13   t=t+5
30   fin
29   1   14   s=h-t
12   19   34   fin
30   1   15   e=f*s
9   10   11   fin
31   1   15   y=s*4
38   40   fin
32   1   16   h=e*f
18  fin
33   1   17   e=h/2
40  fin
34   1   9   x=a+f
31   44   fin
35   1   18   y=e+delta
42   43   fin
36   1   10   delta=x+delta
11   26   fin
```

```
37   1   18   z=e/s
16   fin
38   1   17   x=s+h
40   39  fin
39   1   17   l=h/5
38   fin
40   1   17   f=h-a
33   38   fin
41   1   18   z=a*f
0   fin
42   1   18   e=d/l
35   fin
43   1   18   s=delta*f
35   fin
44   1   19   h=s/5
37   fin
45   1   20   h=h*z
0   fin
```

Fig.5.10 Fichier résultant de la 1ère étape de l'ordonnancement.

Nous rappelons que la satisfaction des contraintes se fait pour chaque cycle (le problème de contraintes ne se pose pas pour des instructions ordonnancées dans des cycles différents). Par conséquent, nous utilisons le fichier « **same_cycle** » pour y ranger les instructions ayant le même cycle après la 1ère étape de l'ordonnancement.

CYCLE N° 1 :

```
1   1   1   x=a+d
3   4   fin
12   1   1   r=g+c
18   3   4   fin
```

Le traitement de la satisfaction des contraintes est fait par le module **PROG « ALGO 5.1 »**.

Nous constatons qu'il existe une contrainte entre l'instruction n° 1 et l'instruction n°3, et une autre entre les instructions n°1 et n°4 . Toutefois, les instructions 3 et 4 sont ordonnancées dans d'autres cycles. De ce fait, les contraintes 1-3 et 1-4 sont déjà satisfaites dès la 1ère phase de l'ordonnancement.

Il en va de même pour les contraintes 12-18, 12-3 et 12-4 du fait que l'instruction 12 est ordonnancée dans le 1er cycle, alors que les instructions 18, 3 et 4 sont respectivement ordonnancées dans les cycles 6, 3 et 2.

Pour le cycle N°2 on a :

```
2   1   2   y=x+1
5   fin
4   1   2   z=a+x
1   fin
```

Les instructions 2 et 5 étant ordonnancées dans des cycles différents (respectivement 2 et 3), la contrainte 2-5 est satisfaite dès la 1ère phase de l'ordonnancement. Il en est de même pour les instructions 4 et 1 (ordonnancées respectivement dans les cycles 2 et 1).

Pour le **cycle N°3**, on trouve dans le fichier « **same_cycle** », ce qui suit :

```
3  1  3   h=x+y
1  6  8  fin
5  1  3  x=z*4
2  8  fin
8  1  3   z=a*y
9  3  5  fin
```

Fig.5.11 Instructions ordonnancées dans le 3ème cycle

Un fichier est alors généré (grâce au module **prog** « **ALGO 5.1** »), et contient les contraintes détectées au 3ème cycle.

Le contenu de ce fichier pour le cycle n°3 est le suivant:

CONTRAINTE DE CYCLE N°3 :

```
1  3  8  fin
2  5  8  fin
```

Fig.5.12 Contraintes détectées dans le 3ème cycle

Notons que :

- Le premier champ indique le numéro de la contrainte
- Les deuxième et troisième champ indiquent les numéros des instructions ordonnancées au 3ème cycle et assujetties à la même contrainte.

Il est à remarquer que seules ces contraintes sont extraites à partir du fichier indiqué dans la Figure 4. du fait que les autres contraintes concernent des instructions ordonnancées dans des cycles différents.

Afin de pouvoir exploiter l'outil de coloration de graphes ([10], [11], [12]), il s'agit de transformer les données indiquées dans **la Fig 5.12** en un fichier (**FIN.gra**) tel qu'indiqué dans la **Fig 5.13**.

```
3   8   FIN
5   8   FIN
END
```

Fig.5.13 Format de fichier (FIN.gra) correspondant aux données de la Fig.5.12

Cette transformation de données est aussi faite par le module **PROG**.

Le format indiqué dans la **Fig5.12** correspond au graphe G=(V,E) tel que : V={3, 5, 8} et E={e1, e2} ; e1 est l'arête 3-8; e2 est l'arête 5-8 (**Cf. Fig.5.13**).

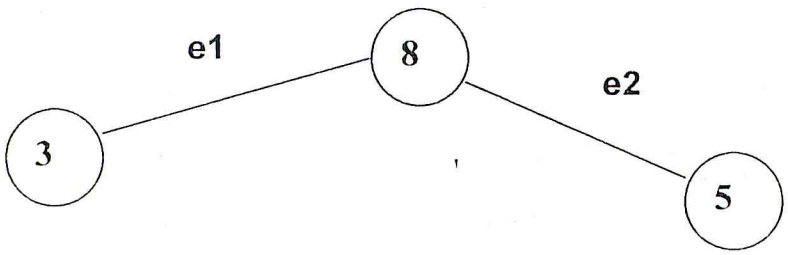


Fig.5.14 Graphe associé aux contraintes et instructions du 3ème cycle.

L'exécution du module **GRACOL** « **ALGO5.3** »génère le fichier **FIN.col** qui contient la coloration des nœuds associés aux instructions (Cf. **Fig54.15**).

```
Couleur 1 : 3  5   FIN
Couleur 2 : 8   FIN
END
```

Fig.5.15. Coloration d'un graphe

La coloration obtenue indique que les nœuds 3 et 5 ont la même couleur, et que le nœud 8 en prend une autre. En d'autres termes, les instructions 3 et 5 peuvent être ordonnancées durant le même cycle, alors que pour satisfaire les contraintes 3-8 et 5-8, l'instruction 8 est ordonnancée dans un autre cycle.

Une mise à jour du fichier **same_cycle (Cf. Fig.5.11)** est alors effectuée, et le résultat est celui indiqué dans la **Fig 5.16.**

```
3  1  3   h=x+y
1  6  8  fin
5  1  3  x=z*4
2  8  fin
8  1  4   z=a*y
9  3  5  fin
```

Fig.5.16. Mise à jour du fichier same-cycle

Notons que cette mise à jour est réalisée par le module **gr** et est sauvegardée dans un fichier intermédiaire. Cette mise à jour entraîne aussi une mise à jour des autres cycles supérieurs à celui qui vient d'être traité. Les instructions de ces cycles seront à leur tour traitées selon le traitement qui vient juste d'être décrit.

Après avoir traité puis arrangé toutes les instructions (grâce au module **prog**), le résultat de cette étape est sauvegardé dans le fichier **TOTALE_DEPEND (Cf. Fig.5.17).**

```
1  1  1  x=a+d
3  4  fin
2  1  2  y=x+1
5  fin
3  1  3  h=x+y
```

```
1   6   8   fin
4   1   2   z=a+x
1   fin
5   1   3   x=z*4
2   8   fin
6   1   5   l=h/5
13   fin
7   1   5   f=h-z
24   15   fin
8   1   4   z=a*y
9   3   5   fin
9   1   9   y=l/b
8   14   16   20   fin
10   1   7   h=z*f
16   fin
11   1   11   f=h/2
19   fin
12   1   1   r=g+c
18  3   4   fin
13   1   6   y=x+r
6   25   fin
14   1   7   h=x+y
20   9   fin
15   1   6   delta=x*z
18   7   fin
16   1   8   delta=delta+3
9   10   fin
17   1   11   h=delta/5
18   19   fin
18   1   10   u=delta-z
17   fin
```

```
19   1   10   delta=y+delta
17   11   fin
20   1   8   y=l/b
9   14   fin
21   1   12   h=b*f
25   28   fin
22   1   13   f=h+6
0   fin
23   1   14   delta=d-f
32   41   fin
24   1   6   y=x+5
7   fin
25   1   15   h=delta+y
8   31   26   fin
26   1   16   z=h/5
25   fin
27   1   17   t=z*4
29   fin
28   1   18   t=t+5
30   fin
29   1   19   s=h-t
12   19   34   fin
30   1   20   e=f*s
9   10   11   fin
31   1   20   y=s*4
38   40   fin
32   1   21   h=e*f
18   fin
33   1   23   e=h/2
40   fin
```

```
34   1  14   x=a+f
31   44   fin
35   1   25   y=e+delta
42   43   fin
36   1   15   delta=x+delta
11   26   fin
37   1   24   z=e/s
16  fin
38   1   23   x=s+h
40   39   fin
39  1  22   l=h/5
38  fin
40   1.  22   f=h-a
33   38   fin
41   1   24   z=a*f
0   fin
42   1   24   e=d/l
35  fin
43   1   24   s=delta*f
35  fin
44   1   26   h=s/5
37  fin
45   1   27   h=h*z
0   fin
```

Fig.5.17. Résultats de l'ordonnancement d'un flot de données.
Fichier TOTALE_DEPAND_inst_chemin

Enfin, l'ordonnancement de *tous* les fichiers de données sont regroupés en un *seul* fichier global (le fichier **FINALE_ « nom_fichier »**, exemple :**FINALE_chemins**) par le module « **optimisation(ALGO 5.4)** ». Pour chaque cycle, les instructions ordonnancées dans ce cycle, ainsi que leur condition d'exécution (mise entre crochets dans la **Fig.5.18,** et correspondant à la condition incluse dans le fichier **condition_ « nomdefichier »** -**Cf. Fig. 5.10-**), de tous les chemins de données, sont regroupées dans le fichier global **(Cf. Fig.5.18).**

Notons enfin que les instructions assujetties à des conditions d'exécution *différentes* appartiennent à des chemins de données *différents*.

Cycle1 :

y=a+b [A and B]
z=a+c [A and B]
u=a+d [not(A) and not(B)]
v=b-d [not(A) and not(B)]
t=a-c [not(A) and B]
r=c+d [A and not(B)]

Cycle2 :

h=z+a [A and B]
x=u-b [not(A) and not(B)]
v=v-a [not(A) and not(B)]
i=t+b [not(A) and B]
f=a+b [A and not(B)]

Cycle3 :

```
m=h+c [A and B]
n=x-d [not(A) and not(B)]
p=v+a [not(A) and B]
s=f-a [A and not(B)]
```

Fig.5.18. Exemple de résultats de l'ordonnancement du CDFG

(regroupement des ordonnancements de tous les fichiers de données).

Fichier FINALE_chemin.

```
not(A) and not(B)
not(A) and B
A and not(B)
A and B
```

Fig.5.19. Fichier des conditions d'exécution des instructions du CDFG

Fichier « condition_chemin»

5.7 CONCLUSION :

Nous avons montré dans ce chapitre que la satisfaction des contraintes a un impact sur le nombre de cycles, donc sur la vitesse du circuit intégré implémentant l'algorithme de l'utilisateur.

Nous avons mentionné que ce problème n'est pas polynomial, et qu'il est plus précisément NP_complet.

La résolution de ce problème par la technique des graphes colorés a été également présentée.

CONCLUSION GENERALE

CONCLUSION GENERALE :

L'ordonnancement a une grande importance dans différents domaines, notamment les processus industriels.
Il nécessite un traitement préliminaire « pré-ordonnancement » qui consiste à transformer l'algorithme a implémenter par un ASIC « application spécifique integrated circuit » en un flot de données contrôlé (CDFG).

La complexité algorithmique de l'ordonnancement d'un CDFG n'étant pas polynômiale, nous avions discuté brièvement des transformations nécessaires (fusion éventuelle de branches conditionnelles et partitionnement) afin de pouvoir ordonnancer un flot de données contrôlé en un temps polynômial, tout en visant une partie opérative efficace du point de vue vitesse (réduction du nombre de cycles). Nous avions noté que ces différentes tâches font partie d'un travail préalable effectué dans le cadre d'une thèse d'Ingénieur [2].

Nous avions par la suite présenté quelques techniques d'ordonnancement de base, et indiqué leurs limites. Notre travail consistant à ordonnancer des flots de données déjà générés, nous avions présenté les formats de tous les fichiers d'entrée, ainsi que tous les modules qui constituent notre exécutable.

La suite de ce mémoire a consisté principalement à la présentation des deux grandes étapes de l'ordonnancement, à savoir le traitement de la dépendance des instructions et la satisfaction des contraintes de l'utilisateur. Nous avions particulièrement mis en relief le caractère *NP-complétude* de la deuxième tâche, et montré comment des techniques efficaces permettent d'aboutir à des résultats intéressants.

Enfin, les différents modules qui constituent notre exécutable ont été expliqués à travers des exemples simples, puis à travers des exemples plus concrets, espérant ainsi avoir facilité au lecteur la compréhension des différents modules. Notons que ces modules ont été implémentés en C sous le système d'exploitation LINUX.

REFERENCES

[1] D. GAJSKI, N. DUTT, A. WU, S. LIN "High Level Synthesis: Introduction to Chip and System Design" Kluwer Academic Publishers

[2] A. ADMANE, M. DJEBAILI "Vers un ordonnancement polynomial d'un nombre exponentiel de chemins de données" mémoire d'ingénieur en Recherche Opérationnelle USTHB 2000

[3] A. MAHDOUM "SPOT: An Estimation of the Switching Power Dissipation in CMOS Circuits and Data Paths Tool" SASIMI'97 1-2 Dec.97 Osaka Japan

[4] A. MAHDOUM "SPOT: Un Outil à Base d'un Algorithme Génétique pour Estimer la Consommation Maximale de la Puissance Dynamique des Circuits CMOS" CSCA'99 8-9 Nov.99 Hôtel Sheraton Alger

[5] A. MAHDOUM "SPOT: An Estimation of the Maximal and the Average Switching Power Dissipation in CMOS Circuits" DATE'02 Designer's Forum 4-8 March 2002 Palais des Congrès Paris France

[6] M.R. GAREY, D.S. JOHNSON "Computers and intractability: a guide to the theory of NP-completeness" Freeman San Francisco.

[7] "The P versus NP Problem, Millennium prize problems" http://www.claymath.org/Millennium_Prize_Problems/P_vs_NP/

[8] JOHNATHAN, TERNER "Technique de Coloration de Graphes"

[9] BRELAZ "Technique de Coloration de Graphes"

[10] H. BELKOUCHE, F. LOUIZ, A. MAHDOUM "FEWERCOLORS : A New Technique for Solving the Graph Coloring Problem" DCIS'2000 21-24 Nov.99 Montpellier France

[11] H. BELKOUCHE, F. LOUIZ, A. MAHDOUM "FEWERCOLORS: "A New Technique for Solving the Graph Coloring Problem" DATE'02 Designer's Forum 4-8 March 2002 Palais des Congrès Paris France

[12] A. MAHDOUM, H. BELKOUCHE, F. LOUIZ "FEWERCOLORS: A New Technique for Solving the Graph Coloring Problem" 7th IEEE/ICECS 17-20 Dec. 2000 Beirut Lebanon

[13] A. V. AHO, J. E. HOPCROFT, J. D. ULLMAN "Data Structures and Algorithms" Addison-Wesley Publishing Company

[14] J.M.RIGAUD, A.SAYAH "Programmation en langage C" BERTI édition 1992 .

[15] Jesse Liberty Le Langage C++