

Projet Systèmes Informatiques

Du compilateur vers le microprocesseur

Daniela Dragomirescu

Eric Alata

Organisation du projet :

1. Développement d'un compilateur en utilisant LEX et YACC

- Application des Automates & Langages
- Présentation des logiciels LEX et YACC

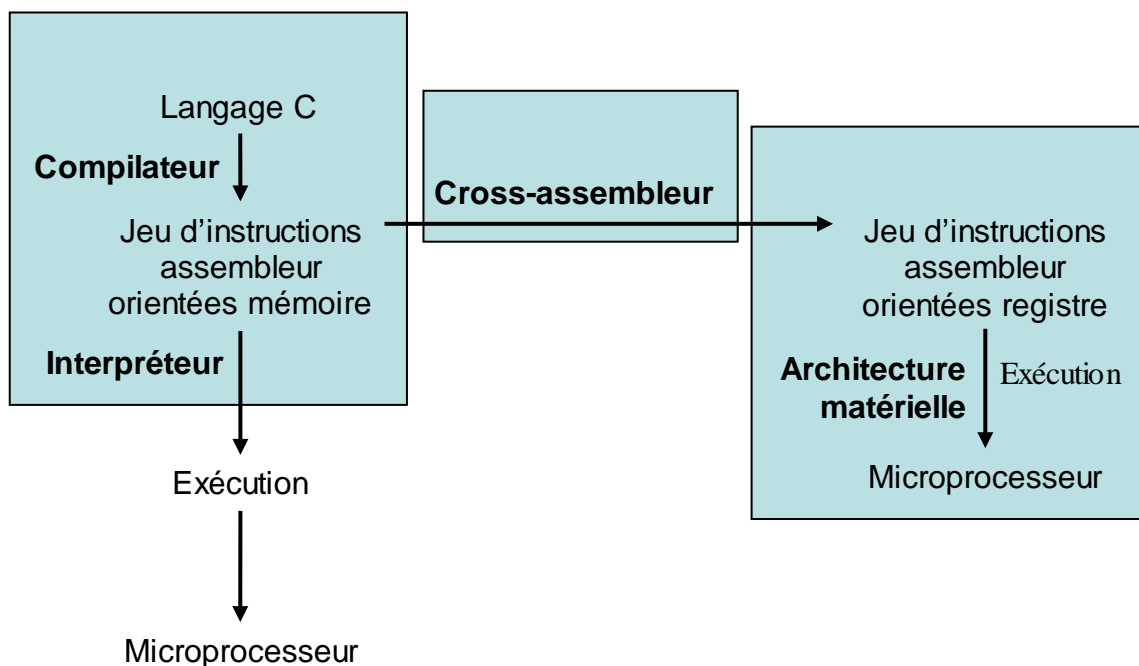
2. Conception d'un microprocesseur de type RISC avec pipeline

- Application des Architectures matérielle pour systèmes informatiques

Objectif

L'objectif de ce Projet Systèmes Informatiques est de réaliser un système informatique complet. Nous allons réaliser un compilateur qui traduit un langage source (proche du C) en un langage cible (jeu d'instructions assembleur orientées mémoire). Créer un microprocesseur correspondant à un jeu d'instructions orientées mémoire est difficile. Par contre, créer un microprocesseur correspondant à un jeu d'instructions orienté registre est plus facile. Un **cross-compileur** permettant de traduire ce jeu d'instructions orientées mémoire en un jeu d'instructions orientées registre sera donc nécessaire.

Voici le schéma complet de notre système informatique :



Réalisation d'un compilateur en utilisant LEX et YACC

1. OBJECTIF

L'objectif de ces travaux pratiques est de réaliser un compilateur d'une version simplifiée du langage C en utilisant le générateur d'analyseur lexical LEX et le générateur d'analyseur syntaxique YACC (Yet Another Compiler Compiler).

Le T.P. va être structuré en 4 parties :

1. Première partie – vous allez développer un compilateur d'un langage permettant la manipulation d'expressions arithmétiques de type C. Le compilateur va générer le code correspondant en langage assembleur. Ensuite vous allez écrire, un interpréteur du langage assembleur, afin d'obtenir l'exécution du code. Enfin vous allez réaliser un programme de test du compilateur et de l'interpréteur développés.

L'interpréteur sera fourni pour l'année 2019-2020

2. Deuxième partie – vous allez rajouter à votre langage de type C l'implémentation des expressions conditionnelles **if** et **while** .

3. Troisième partie – vous allez rajouter au compilateur le traitement de pointeurs et/ou de fonctions et/ou un traitement des erreurs (avec différents degrés de difficulté). **Cette partie est optionnelle pour l'année universitaire 2019-2020. La réalisation de cette partie conduira à des notes au-delà de 16.**

4. Cross-assembleur – **Le cross-assembleur sera fourni pour l'année universitaire 2019-2020**

- **Développement d'un compilateur pour un langage simplifié de type C et de l'interpréteur d'un langage assembleur**

Le langage d'entrée est un langage de type C qui reconnaît :

- la fonction **main** (), les accolades { et }
- les constantes de type entier(mot clé : **const**) et leur noms

- les variables de type entier (mot clé : **int**) et leur noms. Le nom d'une variable ou d'une constante doit commencer par une lettre et peut contenir de **lettres**, des **chiffres** et le underscore _
- les opérations arithmétiques définies par les signes mathématiques suivants : +, -, *, /, = et les parenthèses ()
- les séparateurs : espace, TAB et virgule
- la fin de ligne \n
- la fin d'une instruction signalée par " ; " comme en langage C classique
- la fonction printf() ayant 1 seul paramètre : la variable dont la valeur doit être affichée
- les entiers doivent être reconnus sous la forme décimale et la forme exponentielle
- notre langage C permet la déclaration de plusieurs variables dans la même ligne (ex : int toto, mimi=5 ;) ou la déclaration de plusieurs constantes dans la même ligne. Dans ce cas n'oubliez pas d'allouer de la mémoire pour chacune des variables et/ou constantes déclarées
- notre langage C fait la différence entre lettres minuscules et majuscules. Les mots clés du langage seront reconnus seulement s'ils sont écrits en minuscule.
- On autorise la déclaration des constantes et variables seulement à l'intérieur de la fonction main(). Cette partie déclarative doit se trouver avant les instructions, obligatoirement.
- aucune autre appel de fonction n'est reconnu par notre langage C simplifié, dans une première phase

Dans un premier temps on va réaliser l'analyseur lexical qui détecte les mots typés (tokens) spécifiques à notre langage (tel que décrit au-dessus) et affiche à l'écran un message qui indique le mot typé (token) lu. Une fois l'analyseur lexicographique correctement développé, on va modifier la spécification lexicale afin de passer ces mots typés à YACC.

Dans un deuxième temps on va réaliser l'analyseur syntaxique qui analyse notre langage de type C et produit comme sortie le code assembleur correspondant.

Description du langage assembleur à utiliser :

Les données sont rangées dans **une mémoire de données de type tableau**, avec **une première zone réservée aux variables et constantes déclarées** et **une deuxième zone pour les résultats temporaires** (voir cours section 5.3.4 du chapitre 3). Il faut gérer la libération de l'espace mémoire.

Les instructions connues sont :

- addition des 2 opérandes :

ADD @résultat @opérande1 @opérande2

code opération ADD 1

exemple :

si l'opérande 1 est à l'adresse 50, l'opérande 2 est à l'adresse 55 et le résultat à l'adresse 250 on génère :

1 250 50 55

Ce codage permettra une interprétation plus aisée.

- multiplication des 2 opérandes :

MUL @résultat @opérande1 @opérande2

code opération MUL 2

- soustraction

SOU @résultat @opérande1 @opérande2

code opération SOU 3

- division

DIV @résultat @opérande1 @opérande2

code opération DIV 4

- copie

COP @résultat @opérande

code opération COP 5

- affectation

AFC @résultat valeur constante

code opération AFC 6

- saut inconditionnel

JMP numéro d'instruction

code opération JMP 7

- saut si la condition est fausse

JMF @X numéro d'instruction

code opération JMF 8

On attribue un code entier **1** pour un résultat **true** et **0** pour un résultat **false**.

- comparaison

inférieur :

INF @résultat @opérand1 @opérand2
code opération INF 9

supérieur :

SUP @résultat @opérand1 @opérand2
code opération SUP A

égal :

EQU @résultat @opérand1 @opérand2
code opération EQU B

- imprimer

PRI @résultat
code opération PRI C

A la sortie de votre analyseur syntaxique vous allez écrire les instructions du code assembleur correspondant en clair dans un premier fichier, ainsi que la version codée dans un second fichier.

Dans un troisième temps vous allez développer l'interpréteur du langage assembleur **en utilisant LEX et YACC** (de la version codée générée auparavant). Cet interpréteur va afficher à l'écran le résultat de notre programme écrit en langage C et qui utilise la fonction printf().

Exemple :

```
main()
{ int i,j,k,r;
  i=3 ;
  j=4 ;
  k=8 ;
  printf (i) ;
  r=(i+j)*(i+k/j) ;
  printf ( r ) ;
}
```

Le résultat affiché par l'interpréteur à l'écran va être :

3

35

Donc, en dernière étape vous allez tester le compilateur et le interpréteur développés.

- **Implémentation des expressions conditionnelles du if et du while .**

Vous allez rajouter les expressions conditionnelles ainsi que les instructions if et while à la syntaxe identique à celle du langage C. Vous allez faire les modifications nécessaires au niveau de l'analyseur lexical et syntaxique de votre compilateur.

- **Implémentation de pointers - optionnel**

Vous allez rajouter l'implémentation de pointers en langage C. Vous allez faire les modifications nécessaires au niveau de l'analyseur lexical et syntaxique de votre compilateur. Vous allez également rajouter les instructions assembleur nécessaires pour l'implémentation de pointers.

- **Implémentation de fonctions - optionnel**

Vous allez rajouter l'implémentation de fonctions en langage C. Vous allez faire les modifications nécessaires au niveau de l'analyseur lexical et syntaxique de votre compilateur. Vous allez également rajouter les instructions assembleur nécessaires pour l'implémentation de fonctions.

- **Traitement des erreurs - optionnel**

Dans un premier temps, un traitement des erreurs simple est à implémenter au niveau du compilateur. Il permettra de détecter qu'une erreur est intervenue, d'afficher un message d'erreur et la ligne sur laquelle l'erreur se trouve dans un premier temps.

Dans un second temps il faut pouvoir analyser une erreur pour ensuite continuer l'analyse de la chaîne d'entrée.

Par la suite nous allons présenter les deux logiciels utilisés : LEX et YACC.

2. LEX : générateur d'analyseurs lexicaux

2.1 Introduction

LEX est un outil qui, à partir de la description de la lexicographie d'un langage (données d'entrée) génère un analyseur lexicographique associé. Cet analyseur généré est un programme C. Les spécifications de l'utilisateur fournies comme données de LEX sont converties dans un programme qui reconnaît une série de terminaux (tokens) et exécute, en plus, des fragments de code (actions) fournis aussi par l'utilisateur.

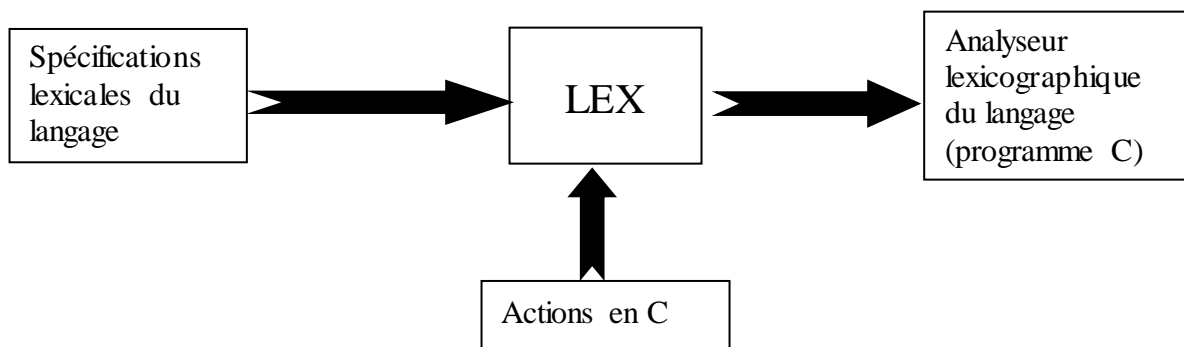


Fig.1 Générateur de l'analyseur lexical.

L'analyseur lexicographique doit ensuite être compilé pour être utilisé.

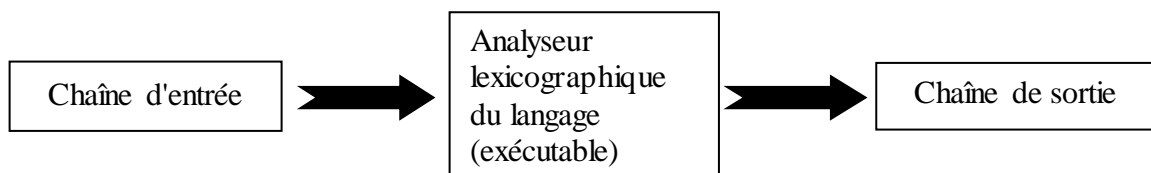


Fig.2 Utilisation de l'analyseur lexical.

Les données de ce programme exécutable doivent être une chaîne conforme aux spécifications lexicales. Au fur et à mesure que le flux d'entrée est traité les actions sont exécutées chaque fois qu'un token est reconnu. Si le flux d'entrée contient des expressions qui ne sont pas reconnues, elles sont copiées à la sortie sans aucun traitement.

Remarque:

Le programme généré par LEX est stocké dans un fichier nommé **lex.yy.c** qui contient la fonction **yylex()**. Cette fonction reconnaît les expressions définies et réalise le traitement demandé sur celles-ci.

Les spécifications lexicales du langage traité sont décrites dans un langage qui sera présenté à la section 2.2. Avant cela, on va donner deux exemples. Le but est uniquement de donner une idée de ce langage.

Exemples:

1. Programme qui élimine les blancs et les TABs à la fin de chaque ligne

Les spécifications sont les suivantes :

```
%%  
[ \t]+$ ; (attention :devant \t se trouve un blanc)
```

La spécification contient le **délimiteur %% pour indiquer le début de spécification des règles**. Dans cet exemple il y a une seule règle qui dit que chaque fois que l'on rencontre un ou plusieurs blancs ou tabs, aucune action n'est effectuée. Les crochets carrés définissent la classe formée par les caractères blancs et tab ; le signe + signifie « un ou plusieurs » caractères de ce type (répétition) ; enfin, \$ indique « fin de la ligne ». Aucune action n'est spécifiée, alors la fonction `yylex()` va ignorer cette configuration. Toute autre combinaison va être copiée à la sortie.

2. Programme qui remplace plusieurs blancs ou TABs avec un seul blanc.

```
%%  
[ \t]+ printf(" ");
```

Ici on voit apparaître l'utilisation d'une action (`printf`).

2.2 Sources LEX

Le format général d'une spécification lexicale d'un langage pour LEX est le suivant :

définitions

%%

règles

%%

sous-programmes utilisateurs

Les définitions, les sous-programmes et le deuxième %% (si il n'y a pas de sous-programmes) sont optionnels.

Les définitions seront abordées à la section 2.6.

Le premier %% doit être toujours présent pour marquer le début des règles. Le plus simple programme LEX est le suivant :

%%

Aucune définition, aucune règle. L'effet de ce programme est de copier la chaîne d'entrée à la sortie sans aucune modification.

En mode normal, les règles ont 2 parties :

- dans la première partie → l'**expression** que l'on doit trouver ;
- dans la deuxième partie → l'**action** à effectuer quand on trouve cette expression.

Exemples :

1. %%

```
int printf("trouvé mot clé INT");
```

Ce programme cherche dans les données d'entrée le mot « int » et affiche à la sortie le message « trouvé mot clé INT ».

2. %%

```
hello printf ("salut") ;
```

```
world printf ("monde") ;
```

```
sea printf ("mer") ;
```

Ce programme peut représenter le début d'un traducteur d'anglais vers le français.

2.3 Mots typés (tokens)

Dans les exemples précédents, les expressions sont réduites à un mot typé (token). On appelle un token une chaîne de caractères (lettres ou chiffres) qu'on veut détecter. Dans le cas général, une expression contient des caractères tokens et des opérateurs qui précisent des répétitions, des choix ou d'autres règles de composition de ces tokens.

Exemple:

[**^abc**]⁺ définit la répétition (+) de lettres de l'alphabet sauf a, b et c(^).

On va détailler maintenant chaque opérateur.

2.3.1 Les opérateurs

Les caractères utilisés comme opérateurs sont les suivants :

" \ [] ^ - ? . * + | () \$ / { } % < >

Si on désire les utiliser comme caractères d'un token, ils doivent être précédés du symbole \.

L'action de ces opérateurs est la suivante :

- l'opérateur " indique que tout ce que se trouve entre les signes " doit être considéré comme texte d'un token;
- l'opérateur \ indique que le signe qui le suit doit être traité comme un caractère d'un token. Ce même opérateur peut être utilisé pour la spécification portable de caractères spéciaux comme :

\n	newline
\t	TAB
\b	backspace

Exemple : Les expressions

xyz'++'

"xyz++"

xyz|+|+

ont toutes la même signification. Ces expressions cherchent la chaîne xyz++ dans le flux d'entrée. L'utilisation de l'une ou l'autre de ces variantes tient du goût de chacun.

Remarque

Si on avait écrit `xyz"+ "+"`, on aurait exprimé la répétition du token `xyz+`.

- l'opérateur `[]` délimite des classes des caractères. La signification des opérateurs n'est pas prise en compte quand ceux-ci se trouvent entre `[]`, exception faite des opérateurs `\`, `-` et `^` ;
Exemple : `[abc]` est un token constitué d'un seul caractère qui peut être soit `a`, soit `b`, soit `c`.

- l'opérateur `-` indique la plage de variation ;
Exemple `[a-z0-9<>]` signifie la classe des caractères minuscules compris entre `a` et `z`, à laquelle on ajoute les chiffres entre `0` et `9`, les symboles « plus petit » et « plus grand ».

- l'opérateur `^` indique le fait que la classe est composée du complément des caractères spécifiés. Cet opérateur apparaît toujours entre des crochets.
Exemple : `[^abc]` représente la classe de tous les caractères sauf `a`, `b` et `c` ;
`[^a-zA-Z]` représente la classe de caractères qui ne sont pas des lettres.

Remarque :

Le caractère `^` doit se trouver en première position entre les crochets.

- l'opérateur `.` signifie n'importe quel caractère ;
- l'opérateur `?` signifie que l'élément se trouvant devant lui est optionnel ;
Exemple : `ab?c` signifie soit la chaîne `abc` soit `ac`, c'est-à-dire l'apparition de `b` dans cette expression est optionnelle
- les opérateurs de répétitions
 - l'opérateur `*` indique n'importe quel nombre des apparitions (incluant `0`) du caractère ou de la classe de caractères qui le précède ;
 - l'opérateur `+` signifie une ou plusieurs apparitions du caractère ou de la classe de caractères antérieure ;

Exemple :

`[a-z]+` signifie toutes les chaînes de caractères en minuscules ;

`[A-Za-z][A-Za-z0-9]*` toutes les chaînes qui commencent avec une lettre. Cette expression est utilisée habituellement pour reconnaître les identificateurs dans les langages de

programmation.

- l'opérateur `|` est utilisé à la construction des expressions alternatives ;

Exemple : `ab|cd` signifie soit `ab` soit `cd` .

- l'opérateur `()` est utilisé pour grouper des expressions alternatives dans des expressions plus compliquées ;

Exemple : `(ab|cd+)?(ef)*` peut représenter « `abefef` », « `efefef` », « `cdef` », « `cddd` », mais non « `abcd` », « `abc` » ou « `abcdef` ».

- l'opérateur `^` est utilisé pour reconnaître les expressions qui apparaissent au début d'une ligne . Cette utilisation ne peut pas être confondue avec l'opérateur de complémentarité, car celui-ci apparaît toujours entre des crochets, comme spécifié au dessus ;

- l'opérateur `$` est utilisé pour reconnaître les expressions qui apparaissent à la fin d'une ligne ;

Exemple : `ab$` reconnaît l'expression `ab` si elle apparaît à la fin de la ligne
`ab\n` est équivalente avec `ab$`

- les opérateurs `{` et `}` spécifient soit des répétitions, si entre eux se trouvent des numéros, soit une définition si entre eux se trouve un nom ;

Exemple : `{digit}` cherche la chaîne « `digit` » et la remplace avec la définition correspondante ;

`a{1,5}` cherche les apparitions de groupes de 1 à 5 « `a` » ;

`a{2, }` cherche des apparitions de « `a` » 2 fois ou plus;

`a{3}` cherche exactement 3 apparitions de « `a` »

2.3.2 Priorité des opérateurs

Les opérateurs placés sur les premières lignes ont une priorité plus grande que ceux placés sur les dernières lignes (priorité décroissante). Sur une même ligne, les opérateurs ont la priorité décroissante suivante :

Caractères escape	\caractères spéciaux
Expressions entre crochets	[...]
Expressions entre guillemets	"..."
Opérateur utilisé pour grouper	(...)
Définitions	{nom}
Un caractère, répétition	* ? +
Concaténation	
Contexte	\$ ^
Alternative	

Remarque: LEX accepte des expressions ayant la longueur maximale de 100 caractères.

2.3.3 Les actions

Quand une expression (définie de la façon décrite précédemment) est reconnue, le programme généré par LEX va effectuer les actions spécifiées à la suite de la définition. Les actions sont, en général, des fragments de code écrits en langage C, fournis par l'utilisateur.

La plus simple des actions est l'action nulle « ; », c'est-à-dire, on reconnaît l'expression et on n'exécute rien.

Pour des actions plus complexes, il existe une série de fonctions et de variables globales comme:

- la variable globale **yytext**, qui est un pointeur vers une chaîne de caractères qui contient l'expression trouvée;

Exemple: Pour imprimer l'expression trouvée, l'action à faire est :

```
[a-z]+ printf("%s", yytext);
```

Cette action est très utilisée, elle peut être écrite plus simplement:

```
[a-z]+ ECHO;
```

- la variable globale **yylen** qui représente le nombre de caractères de l'expression trouvée;

Exemple: Pour compter les expressions trouvées et le nombre de caractères traités on écrit:

```
[a-zA-Z]+ {words++; chars +=yylen;}
```

Le dernier caractère d'une expression peut être accédé ainsi:

```
yytext[yylen-1]
```

Concaténation

Pour réaliser la concaténation de deux expressions on utilise la fonction **yymore()**. Elle indique que le token suivant reconnu va être ajouté à la valeur présente de **yytext** à la place de remplacer cette valeur. La valeur de **yyleng** est modifiée en conséquence.

```
Exemple: "xyz+"      {printf("%d → %s\n", yyleng, yytext) ;  
                        yymore();}  
                        .      printf("%s", yytext);
```

Dès qu'on détecte `xyz+` dans le flux d'entrée on va afficher `4 → xyz+` et l'expression qui suit va être affichée `xyz+"expression"`.

```
Entrée:      xyz+32  
Sortie:      4 →xyz+  
             xyz+32
```

Si on désire éliminer les derniers caractères d'une expression on utilise la fonction **yyless(n)** où **n** représente le nombre des caractères qu'on veut garder en **yytext**.

```
Exemples :  1. "integer"  { printf("%s\t ", yytext) ;  
                        yyless(2) ; }  
                        .      printf("%d → %s\t ", yyleng, yytext) ;
```

Dès qu'on détecte `"integer"` dans le flux d'entrée on va afficher `"integer"`, on garde en **yytext** les 2 premières caractères et à partir du 3^{ème} caractère on relance l'analyse suivant les règles lexicales définies. La sortie de cet exemple est, donc, la suivante:

```
Entrée : integer 32  
Sortie : integer 1→t 1→e 1→g 1→e 1→r 1→" " 1→3 1→2
```

```
2. char buffer[400];  
   %%  
   \n.* { strcpy(buffer, yytext+1);  
        yyless(1);  
   }
```

Cette spécification sauve la ligne courante dans un buffer local. L'expression `\n.*` détecte un caractère newline et toute la ligne qui le suit. Après le contenu de la ligne est sauvé dans le buffer, il est retourné à l'analyseur lexical pour analyse.

2.4 Accès aux sous-programmes d'entrée/sortie

Pour ceci on utilise les fonctions suivantes :

- **input()** → donne le caractère suivant ;
- **output(c)** → écrit le caractère c à la sortie;
- **unput(c)** → met le caractère c dans le flux de données d'entrées pour être lu ensuite par **input()**.

Une autre fonction, qui peut être redéfinie par l'utilisateur, est **yywrap()**, fonction qui est appelée chaque fois que EOF est détecté. Elle peut être utilisée pour ouvrir un nouveau fichier à la fin du fichier courant pour continuer l'analyse lexicale. Si elle n'est pas modifiée, **yywrap()** retourne 1 quand le EOF est détecté.

2.5 L'ambiguïté des règles

LEX peut travailler avec des spécifications qui contiennent des ambiguïtés. Quand l'expression de l'entrée correspond à plusieurs règles, LEX choisit selon les critères suivants :

1. L'expression plus longue est préférée.
2. Pour la même longueur, LEX préfère la règle qui apparaît en premier.

Exemple:

integer	printf("Mot clé\n");
[a-z]+	printf("Identificateur\n");

Si dans le flux de données d'entrée se trouve "integer", il va être considéré comme mot clé, car même si les 2 règles détectent des expressions de la même longueur, la règle integer est définie en premier.

En LEX un caractère est pris en compte une seule fois.

Exemple:

she	s++;
he	h++;
\n	
.	;

On va tout ignorer, à part "she" et "he". Comme "she" inclut "he" LEX va générer un programme qui ne reconnaîtra pas les apparitions de "he" à l'intérieur de "she". Si on désire

reconnaître les apparitions de "he" à l'intérieur de "she" on doit utiliser l'action REJECT qui veut dire "passe à l'alternative suivante". En général REJECT est utile quand on ne cherche pas la division de flux d'entrée en terminaux, mais on désire détecter l'apparition de certaines configurations qui sont partiellement superposées.

2.6 Définitions

L'utilisateur a besoin de facilités nouvelles pour définir les variables qui vont être utilisées par LEX. Celles-ci peuvent apparaître autant dans la partie définitions que dans la partie règles. LEX transforme les règles dans un programme. Toute action qui n'est pas effectuée par LEX est copiée dans le programme général. Il y a 2 règles pour réaliser ceci :

1. Toute ligne qui ne fait pas partie d'une règle ou d'une action et qui commence par un blanc ou un TAB est copiée dans le programme. Les variables définies avant la première apparition de l'expression%% sont des variables globales. Les lignes de commentaire sont aussi copiées dans le programme source.

2. Toute chaîne de caractères incluse entre %{et %} est copiée telle que (les délimiteurs sont éliminés). Toute ligne qui n'est pas incluse entre ces délimiteurs et qui commence à la première position va être considérée par LEX comme une substitution avec le format:

nom	expression
-----	------------

et détermine l'association de l'expression avec le nom. Les deux champs doivent être séparés au moins par un blanc ou un TAB et le nom doit commencer par une lettre.

Exemple: reconnaissance des nombres en virgule flottante

D	[0-9]	
E	[Ee][+-]?{D}+	
%%		
{D}+		
{D}+{E}		printf("integer\n");
{D}*"."{D}*({E})?		printf("real\n");

2.7 Utilisation

La compilation d'un programme LEX s'exécute en 2 étapes :

1. Génération du programme en langage C en utilisant LEX

lex source

opération qui va générer le programme en C **lex.yy.c**

2. La compilation de programme avec un compilateur C.

gcc lex.yy.c -ll

2.8 Lex et YACC

LEX fournit une fonction `yylex()` - analyseur lexical. `yylex()` est exactement le nom demandé par YACC pour l'analyseur lexical. Cette fonction est appelée par YACC pour reconnaître les tokens dans le flux d'entrée. Dans ce cas, la règle LEX doit finir par

return nom_token;

de manière que YACC soit informé du token trouvé par l'analyseur lexical.

3. YACC : générateur d'analyseurs syntaxiques

YACC - Yet Another Compiler Compiler

Le flux de données d'entrée utilisé par différents programmes présente une structure bien définie. En fait, tout programme peut être vu comme un analyseur syntaxique qui accepte un certain "langage d'entrée". Un "langage d'entrée" peut être aussi compliqué qu'un langage de programmation ou aussi simple qu'une chaîne de numéros.

YACC est un instrument qui, à partir de la description de la syntaxe du "langage d'entrée" génère l'analyseur syntaxique associé. Les utilisateurs spécifient la structure du langage d'entrée et des fragments de code en C qui doivent être exécutés chaque fois qu'une structure est reconnue. Si le « langage d'entrée » est un langage informatique, le code C effectue généralement une traduction de la structure du langage source (d'entrée) en des instructions assembleur de la machine d'exécution.

YACC transforme les spécifications du langage d'entrée dans un programme écrit en langage C qui réalise l'analyse syntaxique de données d'entrées.

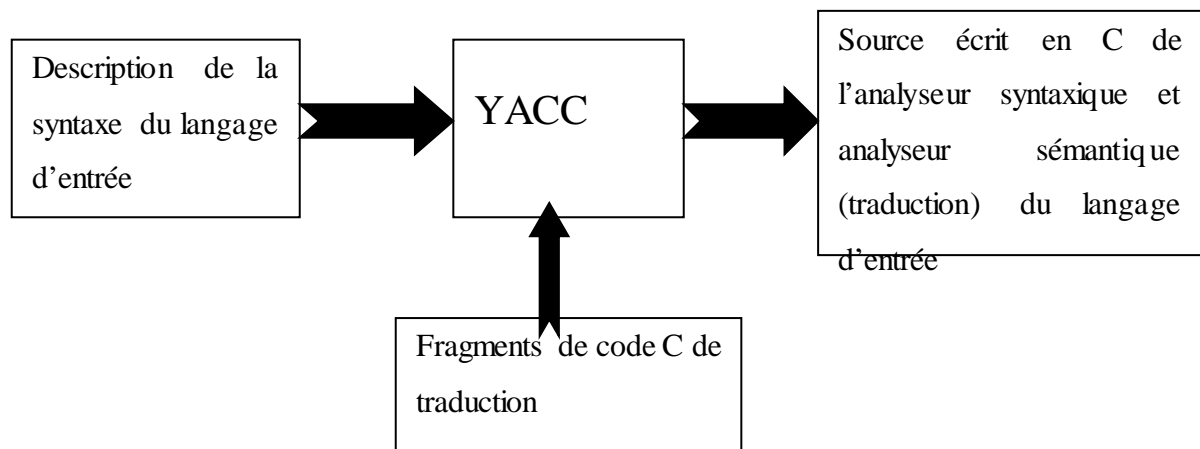


Fig.1 YACC

En utilisant YACC on peut réaliser des compilateurs de C, Pascal, Fortran ou des compilateurs pour les langages de programmations des imprimantes graphiques, etc.

3.1 Introduction

Les utilisateurs de YACC donnent les spécifications du flux d'entrée, qui inclut des règles décrivant la structure syntaxique, des fragments de programme qui représentent les actions à effectuer et des spécifications du traitement primaire des données d'entrée. YACC génère une fonction qui réalise le contrôle de données d'entrée. Cette fonction appelle la fonction de l'analyseur lexical pour reconnaître les terminaux. La conformité de la séquence des terminaux avec la syntaxe du langage d'entrée est ensuite vérifiée. Chaque fois qu'une règle est reconnue, les instructions spécifiées par l'utilisateur sont effectuées (fragments de code C de traduction).

Les règles syntaxique sont décrites en définissant pour chaque non-terminal (placé à gauche de :), la composition de terminaux et non-terminaux (placés à droite de :).

Exemple:

Data : jour '.' mois '.' an ;

Data : jour '/' mois '/' an;

où "jour", "mois", "an" sont des structures définies antérieurement, des terminaux (tokens) déjà reconnus par l'analyseur lexical qui les passe à l'analyseur syntaxique.

Les symboles ":" et ";" sont des délimiteurs pour une règle syntaxique.

Nous allons par la suite détailler la façon d'exprimer les règles et les actions.

3.2 Spécifications de base

Le format d'une spécification YACC est le suivant :

Déclarations

%%

Règles

%%

Programmes

La partie de déclarations peut manquer, ainsi que la partie de programmes. Les blancs et les TABs sont ignorés. Les commentaires sont limités par /* */ comme en C. Les règles ont le format suivant :

A : CORPS;

"A" représente un symbole non-terminal, et "CORPS" représente une chaîne de noms et lettres. Les symboles ":" et ";" sont des délimiteurs. Les noms peuvent être des symboles terminaux ou non-terminaux. YACC nécessite que les noms des symboles terminaux soient déclarés ainsi dans la partie de déclarations.

Exemple:

```
%token      nom1 nom2 ...
```


Tout nom non-défini dans la partie déclarative est considéré comme un symbole non-terminal. Tout symbole non-terminal doit apparaître dans la partie gauche d'une règle au moins une fois.

Les noms peuvent avoir une longueur arbitraire et sont formés par des lettres, "_", et des chiffres (pas au début du nom). Les majuscules et les minuscules sont distinctes.

Une lettre est spécifiée par un caractère entre ' et '. Comme en C le caractère "\" est utilisé pour la spécification portable des caractères spéciaux (newline, return, tab, etc). Pour des raisons techniques, le caractère NULL (\0 ou 0) ne doit pas être utilisé dans les règles syntaxiques.

Si plusieurs règles ont la même partie gauche, on peut utiliser le caractère "|". Il n'est pas nécessaire que toutes les règles avec la même partie gauche apparaissent ensemble. Toute fois, ceci facilite la compréhension des spécifications.

Exemple:

A	:	B C D;		A	:	B C D
A	:	E F;				E F
A	:	G;				G
						;

Si un symbole non-terminal doit reconnaître une chaîne vide, on va écrire :

```
Vide : ;
```

Parmi tous les symboles non-terminaux il en existe un très important. Celui-ci est le **symbole de début**. L'analyseur syntaxique reconnaît ce symbole. Par défaut, le symbole de début (axiome) est considéré comme étant le **premier nom** de côté gauche de la **première règle**. Si on désire, on peut déclarer ce symbole explicite en utilisant le mot clé: **%start**

Exemple: %start symbole

La fin du flux d'entrée est signalée avec un symbole spécial, nommé "end marker". Si les symboles trouvés jusqu'à end marker correspondent à une règle syntaxique, l'analyseur a fini son activité. Sinon, il signale une erreur.

3.3 Actions

A chaque règle syntaxique on associe des actions. Ces actions peuvent retourner différentes valeurs et peuvent utiliser les valeurs retournées par des autres actions. Une action est un fragment de code en C, placé entre accolades.

Exemple: règle syntaxique avec actions

```
A      : '(' B ')'  
        {printf("message %d\n", i);  
         i++;}
```

Pour faciliter la communication entre actions et l'analyseur syntaxique on utilise le **symbole \$** de la manière suivante :

- Pour retourner une valeur, l'action doit attribuer à la pseudo-variable \$\$ une valeur.

Exemple : action qui ne fait que retourner la valeur 1

```
{ $$=1; }
```

- Pour obtenir la valeur retournée d'une action précédente, l'action courante doit utiliser les pseudo-variables \$1, \$2, ... qui font référence aux valeurs retournées des composantes du côté droite d'une règle, parcouru du gauche à droite.

Exemple: 1. Soit la règle

```
A      : B C D;
```

alors \$2 représente la valeur retournée par C et \$3 la valeur retournée par D.

```
2. expr : '(' expr ')' { $$ = $2; }
```

La valeur retournée par cette action est la valeur de "expr" qui se trouve entre les parenthèses.

Par défaut, la valeur d'une règle est la valeur du premier élément de cette règle (c'est-à-dire \$1). Pour ceci, les règles de type

```
A      : B;                      A      : B  
                                     { $$ = $1; }
```

ne nécessitent aucune action.

Dans les exemples précédents, les actions étaient placées à la fin de la règle. Parfois, il est nécessaire d'effectuer des actions avant qu'une règle ne soit complètement analysée. YACC permet d'écrire des actions à l'intérieur de la règle.

Exemple :

```
A      :      B
          { $$ = 1; }
      C
          { x = $2; y = $3; }
```

L'effet est l'attribution de la valeur 1 à x et l'attribution de la valeur retournée par C à y.

Les actions qui ne se trouvent pas à la fin d'une règle sont considérées par YACC comme de nouveaux symboles non-terminaux ; la règle qui analyse ces symboles est la chaîne vide.

Dans certaines applications (ex: compilateurs de C) la sortie n'est pas le résultat direct des actions. Les actions construisent une structure d'arbre d'analyse grammaticale qui génère la sortie. Ces arbres syntaxiques sont construits en utilisant la technique suivante:

1. On écrit une fonction (en C) appelée "nœud" de façon que l'appel

```
nœud( L, n1, n2)
```

génère un nœud avec l'étiquette L, ayant les descendants n1 et n2 et qui retourne l'index de nœud créé.

2. L'arbre d'analyse peut être généré en utilisant des actions de type:

```
expr    :    expr '+' expr
          { $$ = nœud ('+', $1, $3); }
```

L'utilisateur peut définir des autres variables qui peuvent être utilisées par des actions. Ces variables doivent être définies dans la partie déclarative d'une spécification YACC entre les signes "%{ " et "%}".

Exemple:

```
%{ int toto = 0; % }
```

Les variables utilisées par YACC ont des noms qui commencent par "yy" et l'utilisateur doit éviter de définir des variables avec un tel nom.

3.4 Analyse lexicale

L'analyseur lexicale est une fonction qui doit porter le nom **yylex()**. La fonction doit retourner un entier qui représente le "**numéro du token lu**" dans le flux d'entrée. S'il existe une valeur associée à ce token elle va être attribuée à la variable externe **yyval**.

L'analyseur lexical et l'analyseur syntaxique doivent "s'entendre" sur les numéros de chaque token. Ces numéros peuvent être choisis par l'utilisateur ou par YACC. Dans les 2 cas le mécanisme "#define" du C est utilisé. Par défaut YACC choisit ces numéros. Pour un caractère YACC choisit la valeur numérique de ce caractère et pour les autres des numéros supérieur à 257. Si c'est l'utilisateur qui choisit la numérotation, il faut **toujours** choisir pour ces tokens des numéros > 257.

Pour donner un numéro à un token, sa première apparition dans la région de déclarations doit être suivie d'un nombre entier positif. Les tokens et les caractères qui ne sont pas suivis d'un entier reçoivent des numéros par défaut, comme décrit au-dessus.

Exemple :

DIGIT - est déclaré dans la partie de déclarations

```
yylex(){  
  
    extern int yyval;  
    int c;  
    c = getchar();  
    switch(c){  
  
        case '0':  
        case '1':  
        ...  
        case '9':  
  
            yyval = c - '0';  
            return (DIGIT);  
  
    }  
}
```

Cet exemple va retourner le nombre correspondant à DIGIT et sa valeur chaque fois que l'on trouve un chiffre dans le flux d'entrée.

3.5 Ambiguïté et conflits

On considère qu'un ensemble de règles syntaxiques présente une ambiguïté si la même chaîne d'entrée peut être structurée de plusieurs manières différentes.

Exemple :

$\text{expr} \rightarrow \text{expr} \text{ '-' } \text{expr} ;$

est une manière naturelle pour décrire le mode de formation d'une expression mathématique.

Mais cette règle ne nous dit pas comment on va interpréter une expression de type :

$\text{expr} - \text{expr} - \text{expr}$

Celle-ci peut être structurée ainsi :

$(\text{expr} - \text{expr}) - \text{expr}$

ou comme :

$\text{expr} - (\text{expr} - \text{expr})$

YACC détecte ce type d'ambiguïté. Un rapport plus détaillé peut être obtenu en utilisant l'option `-v` à la compilation du fichier source YACC. Il faut regarder le fichier `y.output`.

Pour avoir une idée de la manière par laquelle sont résolus les conflits de ce type, il faut jeter un coup d'œil à l'intérieur de l'analyseur syntaxique. L'analyseur syntaxique est un automate à pile qui peut réaliser quatre fonctions : **déplacement**, **réduction**, **acceptation** et **erreur**.

1. La fonction de **déplacement** ajoute un nouveau symbole à la chaîne analysée. Cette fonction remplit la pile de l'automate.
2. La fonction de **réduction** est l'action exécutée par l'automate quand il a reconnu une règle. Cette fonction vide la pile de l'automate.
3. La fonction d'**acceptation** est une fonction qui indique que le flux d'entrée a été analysé et qu'il correspond aux spécifications.
4. La fonction d'**erreur** indique que, dans le flux d'entrée, existe une sous-chaîne qui ne correspond à aucune spécification.

Revenons à l'exemple précédent. On voit que après avoir lu la chaîne « `expr - expr` » on a un conflit de type **déplacement / réduction**. Dans certain cas, un conflit de type **réduction / réduction** peut apparaître, mais jamais un conflit de type **déplacement / déplacement**. Les règles d'élimination de ces conflits sont les suivantes :

1. Dans le cas d'un conflit **déplacement / réduction** c'est le déplacement qui gagne.
2. Dans le cas d'un conflit **réduction / réduction** on choisit la règle qui apparaît la première dans les spécifications.

Donc notre chaîne d'entrée va être groupée : **expr – (expr – expr)**

Exemple : Voici un autre exemple de la puissance du YACC de résoudre les ambiguïtés.

Considérons un exemple d'un langage de programmation qui permet des structures de type « if – then – else ».

```
statement      :      IF '(' cond ')' statement
                |      IF '(' cond ')' statement ELSE statement
```

Dans ces règles IF, ELSE sont des terminaux, et « cond », « statement » sont des non-terminaux. « cond » décrit une condition et « statement » décrit une chaîne d'actions.

Si l'entrée est de type :

IF (C1) IF(C2) S1 ELSE S2

elle peut être structurée en 2 modes :

```
IF (C1) {
    IF (C2) S1
}
ELSE S2
```

ou

```
IF (C1) {
    IF (C2) S1
    ELSE S2
}
```

La deuxième interprétation est celle habituelle pour beaucoup de langages de programmation. Analysons pour cet exemple la situation où l'analyseur syntaxique est arrivé à analyser seulement :

IF (C1) IF (C2) S1

Cette chaîne peut être considérée comme une instance de la règle

IF (C1) statement

Après avoir lu le reste de l'expression

ELSE S2

on pourrait croire que cette chaîne va être considérée comme une instance de la règle

IF (C1) statement ELSE S2

Ceci été vrai si l'expression était réduite avant de faire le déplacement. Mais comme les conflits de type déplacement / réduction donnent toujours gain de cause au déplacement, ELSE va être déplacé et, après avoir lu le symbole S2, la réduction de la règle IF – ELSE de la partie droite va être faite, suivie de la réduction de la règle

IF (C1) statement.

Donc , on obtient la deuxième interprétation, celle qui est correcte.

Pour éviter les risques de problèmes concernant les ambiguïtés, il est préférable de les éviter lorsque vous donnez vos règles syntaxiques (cf. le cours).

3.6 Priorités

Il y a des situations où les règles définies précédemment ne sont pas suffisantes pour spécifier la notion de priorité. C'est, par exemple, le cas de l'analyse syntaxique des expressions arithmétiques. La majorité des constructions arithmétiques peuvent être décrites en utilisant le concept de niveaux de priorité des opérateurs et des informations sur leurs associativités.

L'associativité et la priorité des opérateurs peuvent être précisées dans la zone déclarative. Ceci est réalisé en utilisant les mots clé :

```
%left  
%right  
%noassoc
```

suivi d'une liste des symboles terminaux (tokens). Tous les symboles qui se trouvent sur une même ligne ont la même priorité. Les symboles terminaux qui se trouvent sur les premières lignes ont une priorité plus petite que celles qui se trouvent sur les lignes suivantes (priorité croissante).

Exemple :

```
%left '+' '-'  
%left '*' '/'
```

Ces lignes décrivent la priorité et l'associativité de quatre opérateurs arithmétiques. + et - ont la même priorité et l'associativité à gauche du fait du terme "left" ; * et / ont une priorité plus grande que les opérateurs précédents et l'associativité à gauche .

Le mot clé *%right* est utilisé (comme vous l'avez déjà deviné) pour l'associativité à droite et le mot clé *%noassoc* est utilisé pour décrire des opérateurs sans associativité (comme certains opérateurs du FORTRAN).

Exemple : Spécifications pour expressions arithmétiques

```
%right '='  
%left '+' '-'  
%left '*' '/'
```

```

%%
expr  :    expr '=' expr
      |    expr '+' expr
      |    expr '-' expr
      |    expr '*' expr
      |    expr '/' expr
      |    NAME
      ;

```

Conforme à cette spécification la chaîne d'entrée :

$$a = b = c * d - e - f * g$$

va être structurée en:

$$a = (b = (((c * d) - e) - (f * g)))$$

Quand on utilise un tel mécanisme, il faut spécifier aussi une priorité pour les opérateurs unaires (comme ' - ' devant un numéro). Le mot clé **%prec** modifie la priorité d'un opérateur dans le cadre d'une règle syntaxique. Il doit apparaître tout de suite après que la règle ait été définie. Dans l'exemple suivant, on donne à l'opérateur unaire ' - ' la même priorité que celle de l'opérateur ' * '.

Exemple :

```

%right '='
%left '+' '-'
%left '*' '/'

%%
expr  :    expr '=' expr
      |    expr '+' expr
      |    expr '-' expr
      |    expr '*' expr
      |    expr '/' expr
      |    '-' expr          %prec '*'
      |    NAME
      ;

```

Un symbole terminal déclaré en utilisant **%left**, **%right**, **%noassoc** n'est plus nécessaire (mais il peut) être déclaré en utilisant **%token**.

3.7 Traitement des erreurs

Le traitement des erreurs est un domaine difficile. Souvent il est inacceptable d'arrêter le traitement de données si on trouve une erreur. Donc il faut résoudre le problème de la reprise d'analyse après la rencontre d'une erreur. L'algorithme général de reprise de l'analyse implique l'élimination de certains terminaux dans la chaîne d'entrée, de manière à reprendre l'analyse devant une expression correcte.

Pour permettre le control des erreurs YACC offre une modalité simple, mais assez générale. Le symbole terminal « **error** » est réservé pour le traitement des erreurs syntaxiques. Ce token peut être utilisé dans les règles syntaxiques et pour actions de corrections de celles-ci. L'analyseur syntaxique vide sa pile jusqu'à l'état dans lequel le token « erreur » est autorisé; ensuite il effectue l'action correspondante.

Afin d'éviter de générer trop de messages d'erreur, l'analyseur syntaxique, après la détection d'une erreur, reste dans l'état d'erreur jusqu'à ce que 3 tokens successifs aient été déplacés et réduits correctement. Si, dans ce temps, une nouvelle erreur est détectée, il n'est pas généré un nouveau message d'erreur.

Exemple :

1. **statement : error ;**
signifie que dans le cas d'une erreur de syntaxe, l'analyseur syntaxique va ignorer l'expression où l'erreur est apparue. Plus précisément, l'analyseur syntaxique va continuer à lire les symboles d'entrée en cherchant 3 tokens consécutives qui sont autorisés pour l'expression courante et va commencer l'analyse avec ceux-ci.

2. **statement : error ';' ;**

indique que dans le cas d'une erreur de syntaxe, l'analyseur syntaxique va ignorer tous les terminaux ou non-terminaux jusqu'à ';'. Tous les symboles jusqu'au ';' ne vont pas être déplacés et vont être ignorés. Quand on rencontre le symbole ';' la règle sera réduite.

Une autre forme de traitement des erreurs peut être utilisée dans les programmes interactifs. On désire la réintroduction de la ligne qui a eu des erreurs, une fois qu'on a affiché le message d'erreur.

Exemple :

```
input      :      error '\n'      {printf ("Rentrez la dernière ligne") ;}
           input
                                   {$$ = $4 ; }
           ;
```

3.8 Utilisation

La compilation d'un programme YACC s'exécute en 2 étapes :

1. Génération du programme en langage C en utilisant YACC :

YACC nom_fichier_source

opération qui va générer le fichier **y.tab.c** qui contient la fonction **yyparse()** Quand cette fonction est appelée, elle va appeler à son tour la fonction **yylex()** pour obtenir les symboles d'entrée. Si une erreur est signalée, **yyparse()** retourne 1.

Si on désire générer le fichier **y.tab.h** qui contient les déclarations de type **#define** qui associe le nom des tokens définis par l'utilisateur avec le code YACC pour chaque token il faut utiliser lors de la compilation l'option **-d**.

YACC -d nom_fichier_source

L'utilisateur doit préciser dans le fichier source de YACC, dans la zone de sous-programmes, d'autres fonctions. Par exemple la fonction **main()** qui appelle la fonction **yyparse()**. Il faut aussi définir une fonction **yyerror()** (fonction définie dans la bibliothèque de YACC) qui indique qu'une erreur de syntaxe a été rencontrée.

Exemple :

```
main()
{
    return yyparse() ;
}
```

et

```
yyerror(char *s)
{
    fprintf(stderr, "%s\n", s) ;
}
```

- La compilation de ce programme:

gcc y.tab.c

Pour la compilation des fichiers de l'analyseur lexical (lex.yy.c) et de l'analyseur syntaxique (y.tab.c) il faut utiliser la commande :

gcc y.tab.c lex.yy.c -ll -o nom_executable.exe

Attention de conserver cet ordre pour les paramètres de **gcc**.

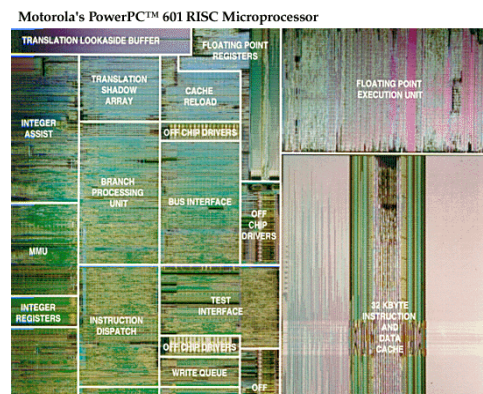
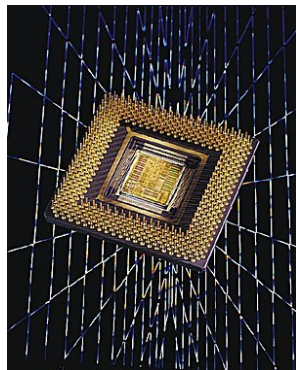
Recommandations pour l'écriture des spécifications

Il existe quelques conventions qui facilitent la compréhension, bien que n'étant pas des obligations:

1. Utilisez des noms commencent par 't' ou des majuscules pour les tokens.
2. Mettez les règles syntaxiques et les actions sur des lignes différentes.
3. Regroupez toutes les règles avec la même partie gauche ensemble.
4. Mettez le signe ';' de la fin d'une règle sur une ligne séparée.
5. Positionnez les actions d'une règle avec un décalage de 3 TABs et les règles syntaxiques avec un décalage de 2 TABs.

Toutes ces recommandations facilitent la lecture du fichier des spécifications syntaxiques.

Conception d'un microprocesseur de type RISC avec pipe-line



Note : Le cours de et les Travaux Pratiques de Architecture Matérielle de 3^{ème} année MIC sont un **pré-requis obligatoire** pour ces Travaux Pratiques.

1. Objectif

2. Première partie - vous allez concevoir et implémenter en VHDL un microprocesseur avec **pipe-line** correspondant aux **instructions assembleur : addition, soustraction, multiplication, division, copie et affectation (sans les instructions de saut et de comparaison)**. Vous allez ensuite synthétiser et optimiser ce microprocesseur du point de vue de la fréquence de fonctionnement et vous allez ensuite l'implémenter sur le FPGA Xilinx. Cette partie est obligatoire.

Pour l'année universitaire 2019-2020, l'implémentation sur FPGA est optionnelle de par le travail à distance. La synthèse et analyse de performance du processeur obtenu (fréquence de fonctionnement, consommation) restent obligatoires.

3. Deuxième partie - vous allez rajouter à votre microprocesseur les unités architecturales nécessaires à l'implémentation des instructions des saut et des comparaison (par exemple : l'unité de branchement). Vous allez ensuite synthétiser et optimiser ce microprocesseur du point de vue de la fréquence de fonctionnement et vous allez ensuite l'implémenter sur le FPGA Xilinx. **Cette deuxième partie est facultative en 2019-2020.**

Afin de vous guider dans la conception de ce microprocesseur, nous allons, dans la deuxième section de ce document, vous présenter le jeu d'instructions orientées registre que vous allez utiliser. Dans la troisième section, nous abordons l'architecture du microprocesseur que nous vous proposons d'implémenter.

2. Langage assembleur

Le jeu d'instructions orientées registre que vous allez utiliser est présenté dans le tableau 2. R_i , R_j et R_k représentent les numéros de registres sollicités par l'instruction. $@i$ et $@j$ représentent des adresses mémoire. Le format d'instruction est de taille fixe. La valeur $_$ est utilisée comme bourrage (**padding**), dans les instructions nécessitant moins de 3 opérandes. Autrement dit, toutes les instructions occupent 4 octets, comme illustré dans le tableau 3. Les jeux d'instructions utilisés par les microprocesseurs commerciaux ne possèdent pas tous cette

caractéristique. Nous vous proposons de concevoir un jeu d'instructions orientées registre avec un format de taille fixe, afin de faciliter votre conception.

Opération	Code	Format d'instruction				Description
		OP	A	B	C	
Addition	0x01	ADD	Ri	Rj	Rk	$[Ri] \leftarrow [Rj] + [Rk]$
Multiplication	0x02	MUL	Ri	Rj	Rk	$[Ri] \leftarrow [Rj] * [Rk]$
Soustraction	0x03	SOU	Ri	Rj	Rk	$[Ri] \leftarrow [Rj] - [Rk]$
Division	0x04	DIV	Ri	Rj	Rk	$[Ri] \leftarrow [Rj] / [Rk]$
Copie	0x05	COP	Ri	Rj	–	$[Ri] \leftarrow [Rj]$
Affectation	0x06	AFC	Ri	j	–	$[Ri] \leftarrow j$
Chargement	0x07	LOAD	Ri	@j	–	$[Ri] \leftarrow [@j]$
Sauvegarde	0x08	STORE	@i	Rj	–	$[@i] \leftarrow [Rj]$

Tableau 2. Jeu d'instructions

Instruction :	ADD	R1	R9	R4
Hexadécimal :	0x01	0x01	0x09	0x04
Binaire :	00000001	00000001	00001001	00000100

Tableau 3. Exemple d'instruction

Comparé au jeu d'instructions orientées mémoire des TP d'Automates et Langages, deux nouvelles instructions ont fait leur apparition : *Chargement* et *Sauvegarde*. L'instruction de chargement permet de copier dans le registre *Ri* le contenu de la mémoire à l'adresse *@j*. L'instruction de sauvegarde permet de copier dans la mémoire à l'adresse *@i* le contenu du registre *Rj*. Ce sont les seules instructions permettant un échange avec la mémoire. Un tel processeur RISC est qualifié de processeur RISC **Load/Store**. Notons que le jeu d'instruction utilisé en TP d'Automates et Langages n'est pas adapté pour ce type de microprocesseur.

3. Architecture du microprocesseur RISC

Nous vous proposons d'implémenter ici une architecture de type **RISC** avec un **pipe-line** à **5 étages** (figure 4). Cette architecture doit fonctionner sur **8 bits**. Nous reprenons en annexe le cours de Structure et Fonctionnement des Ordinateurs de 3^{ème} année MIC de Michel Cubero Castan concernant le cycle des instructions, le pipe-line et les aléas.

Notre microprocesseur va avoir :

- Une unité arithmétique et logique, *section 3.1* ;
- Un banc de registres à double port de lecture, *section 3.2* ;

- Une mémoire d'instructions, *section 3.3* ;
- Une mémoire des données, *section 3.3* ;
- Un chemin des données, *section 3.4*;
- Une unité de contrôle, *section 3.5* ;
- Une unité de détection des aléas, *section 3.6* ;
- Architecture pipe-line sur 5 étages.

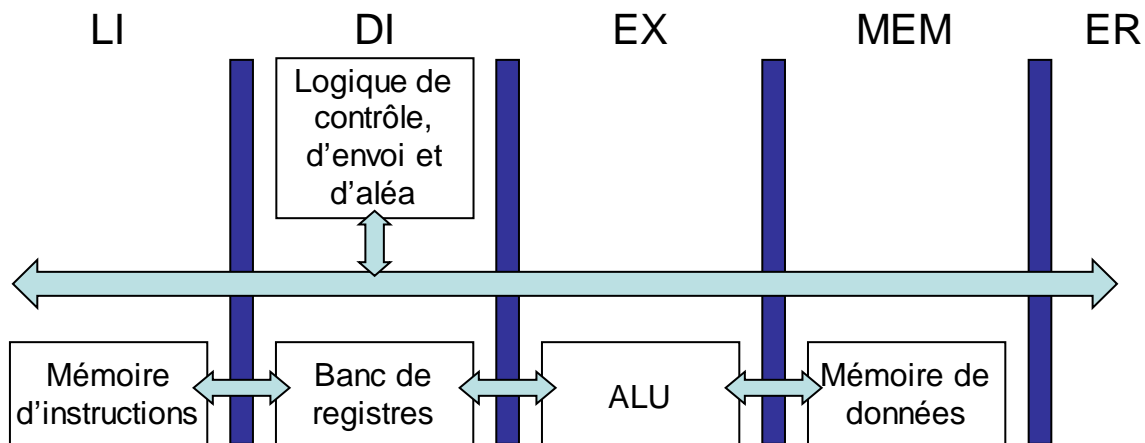


Figure 4. Architecture du microprocesseur RISC avec 5 niveaux de pipe-line

3.1 L'unité arithmétique et logique

On se propose de réaliser une UAL ayant l'interface présentée sur la figure 5. Le signal *Ctrl_Alu* informera sur l'opération à réaliser : Addition, Soustraction, Multiplication et Division. Les drapeaux (**flags**) : *N*, *O*, *Z* et *C* représentent, respectivement, une valeur négative en sortie ($S < 0$), un débordement (**overflow** : taille de $A \text{ OP } B > 8 \text{ bits}$), une sortie nulle ($S = 0$) et la retenue (**carry**) de l'addition. Les opérations arithmétiques **peuvent** être réalisées à partir des fonctions correspondantes ('+', '-', '*', '/'), **fournies dans la librairie de Xilinx**.

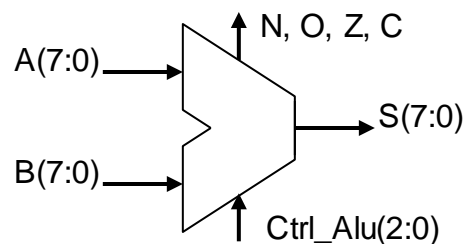


Figure 5. UAL

Les FPGA utilisés (Spartan 6) contiennent de multiplieurs câblés donc vous pouvez réaliser la fonction multiplication en utilisant simplement en VHDL le signe `*`.

Nous ne vous demandons pas d'implémenter la division dans l'année universitaire 2019-2020.

Pour mémoire, nous vous montrons un exemple de déroulement d'une multiplication sur des nombres binaires (sur 4 bits), dans la figure 6.

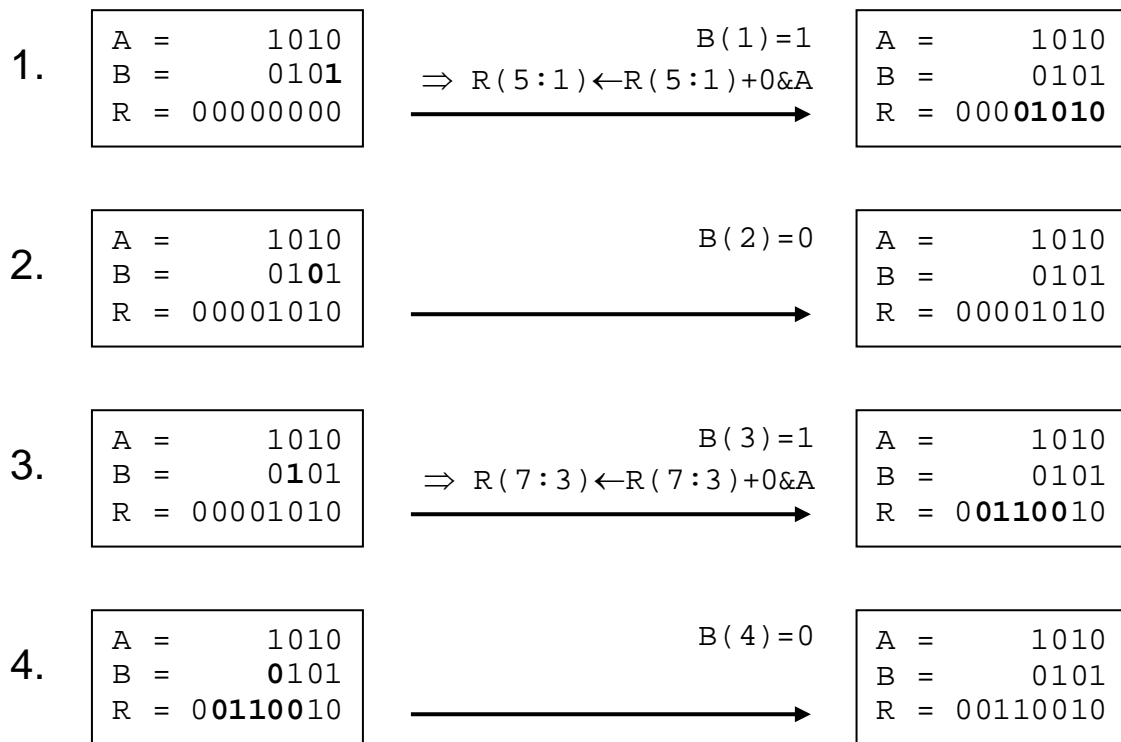


Figure 6. Déroulement d'une multiplication sur 4 bits : $10 * 5 = 50$

3.2 Banc de registres double port de lecture

On se propose de réaliser un banc de 16 registres de 8 bits avec un double accès en lecture et un accès en écriture. Le schéma de ce registre est présenté à la figure 7. Le signal reset *RST* est actif à 0 : le contenu du banc de registres est alors initialisé à *0x00*. @A et @B permettent de lire deux registres simultanément. Les valeurs

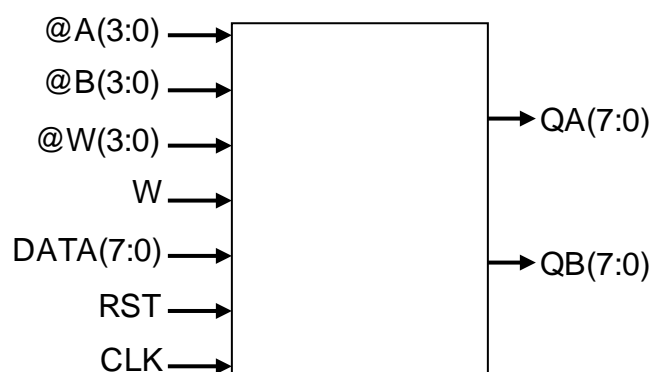


Figure 7. Banc de registres

correspondantes sont propagées vers les sorties *QA* et *QB*. L'écriture de données dans un registre se fait par le biais des entrées @W, W et DATA. W spécifie si une écriture doit être réalisée. Cette entrée est active à 1, pour une écriture. Lorsque l'écriture est activée, les données présentées sur l'entrée DATA sont copiées dans le registre d'adresse @W. On considère que le reset et l'écriture se feront synchrone avec l'horloge.

Bypass D → Q

Il arrive qu'en cours d'exécution, le processeur fasse simultanément une requête de lecture et d'écriture sur le même registre. Ceci constitue un aléa de données. Cet aléas peut être traité par l'unité d'envoi. Toutefois, afin de simplifier la conception de l'unité d'envoi, on se propose d'implémenter cette fonctionnalité directement dans le banc de registres :

Si écriture et lecture sur le même registre alors la sortie $QX \leftarrow DATA$.

3.3 Banc de mémoire

Notre architecture contient deux mémoires : une mémoire pour les données et une mémoire pour les instructions. Leur structure est présentée à la figure 8.

La mémoire des données doit permettre un accès en lecture ou en écriture. L'adresse de la zone mémoire est fournie par l'entrée @. Pour réaliser une lecture, RW doit être positionné à 1 et pour réaliser une écriture, RW doit être positionné à 0. Dans le cas d'une écriture, le contenu de l'entrée IN est copié dans la mémoire à l'adresse @. Le reset, RST , permettra d'initialiser le contenu de la mémoire à $0x00$. La lecture, l'écriture et le reset se feront synchrones avec l'horloge CLK .

Pour simplifier la conception, nous vous proposons une structure de la mémoire des instructions plus simple. Nous supposons que le programme à exécuter par le microprocesseur est déjà stocké dans cette mémoire. De plus, à l'exécution, nous empêchons toute modification du contenu de cette mémoire. Elle s'apparente à une **ROM**. Elle est alors dépourvue des entrées RST , IN et RW . Comme pour la mémoire des données, la lecture se fera synchrone avec l'horloge CLK .

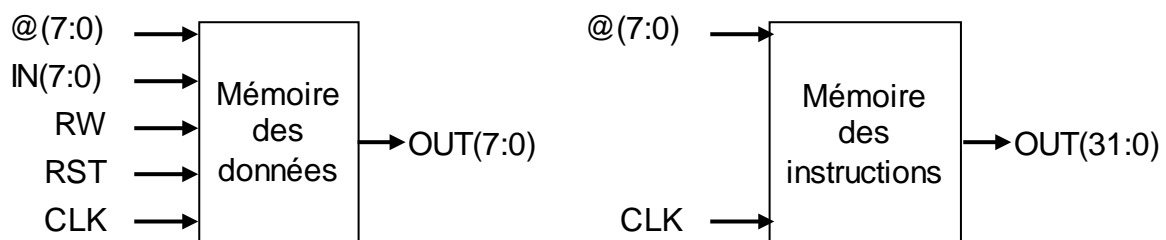


Figure 8. Mémoire des données et mémoire des instructions

3.4 Chemin des données

Nous allons à présent commencer l'intégration des différents éléments précédemment développés afin de concevoir une première version du processeur. Nous allons créer un chemin des données (présenté à l'illustration 6). Afin de vous guider dans la création de ce chemin des données, nous vous conseillons de procéder par étapes. Autrement dit, nous vous conseillons de commencer par créer un chemin de données prenant en compte uniquement une des instructions. Vous pourrez alors tester votre implémentation afin de corriger les erreurs éventuelles. Ensuite, vous pourrez intégrer une nouvelle instruction. Vous bouclerez ainsi tant qu'il reste des instructions à implémenter. La suite de cette section vous présente un exemple de démarche.

1 - Nous commençons par implémenter l'instruction *AFC*. Cette instruction permet de copier une constante dans un registre destination. Sachant que la modification d'un registre ne doit être effectuée qu'au niveau de l'étage 5, l'identifiant du registre destination et la valeur à lui affecter doivent être propagées jusqu'à cet étage. Le chemin suivi par cette instruction est donc le suivant :

- 1.1. Au niveau du premier étage, les différents champs constituant l'instruction sont placés dans le premier pipe-line (*LI/DI*) ;
- 1.2. Au niveau du second étage, aucune sélection de valeur dans le banc de registres n'est nécessaire : le code de l'opération en cours, la valeur à copier et le registre destination sont simplement propagés dans le pipe-line (*DI/EX*) ;
- 1.3. Au niveau du troisième étage, aucune exécution arithmétique n'est nécessaire pour la copie : le code de l'opération en cours, la valeur à copier et le registre destination sont simplement propagés dans le pipe-line (*EX/Mem*) ;
- 1.4. Au niveau du quatrième étage, aucune modification de la mémoire n'est nécessaire pour la copie : le code de l'opération en cours, la valeur à copier et le registre destination sont simplement propagés dans le pipe-line (*Mem/RE*) ;
- 1.5. Au niveau du cinquième étage, nous disposons de toutes les informations nécessaires pour mener à bien la copie : valeur à copier et registre destination. Cette modification se fait en utilisant l'entrée écriture du banc de registres.

Le schéma correspondant est donné en illustration 1. Dans cette illustration, *OP* représente le code d'opération et *A*, *B* et *C* les opérandes. Notons, au passage, que le chemin emprunté par cette opération nécessite, au niveau de l'étage 5, d'utiliser un composant situé schématiquement au niveau de l'étage 2. Mais, en aucune façon, le chemin en question ne passe deux fois par un même pipe-line.

2 - Sur la base de ce schéma, l'instruction *COP* peut être ajoutée. La différence par rapport au schéma précédent est l'obligation, lors du passage à l'étage 2, de propager la valeur associée au registre source et non l'identifiant de ce registre. Pour effectuer ces modifications sans pour autant altérer le bon fonctionnement de l'instruction *AFC*, nous utilisons un multiplexeur. En fonction du code de l'opération, il propage le paramètre *B* ou la valeur contenue dans le registre pointé par le paramètre *B*. Le schéma obtenu est donné en illustration 2.

3 - Remarquons que les illustrations précédentes incluent une unité arithmétique et logique sans les exploiter. Nous allons donc inclure les instructions arithmétiques pour rattacher cette UAL au chemin de données. En utilisant à nouveau des multiplexeurs pour déterminer les éléments à propager, nous obtenons l'illustration 3.

4 - En procédant de la même manière pour l'implémentation des instructions *LOAD* et *STORE*, nous obtenons les illustrations 4 et 5.

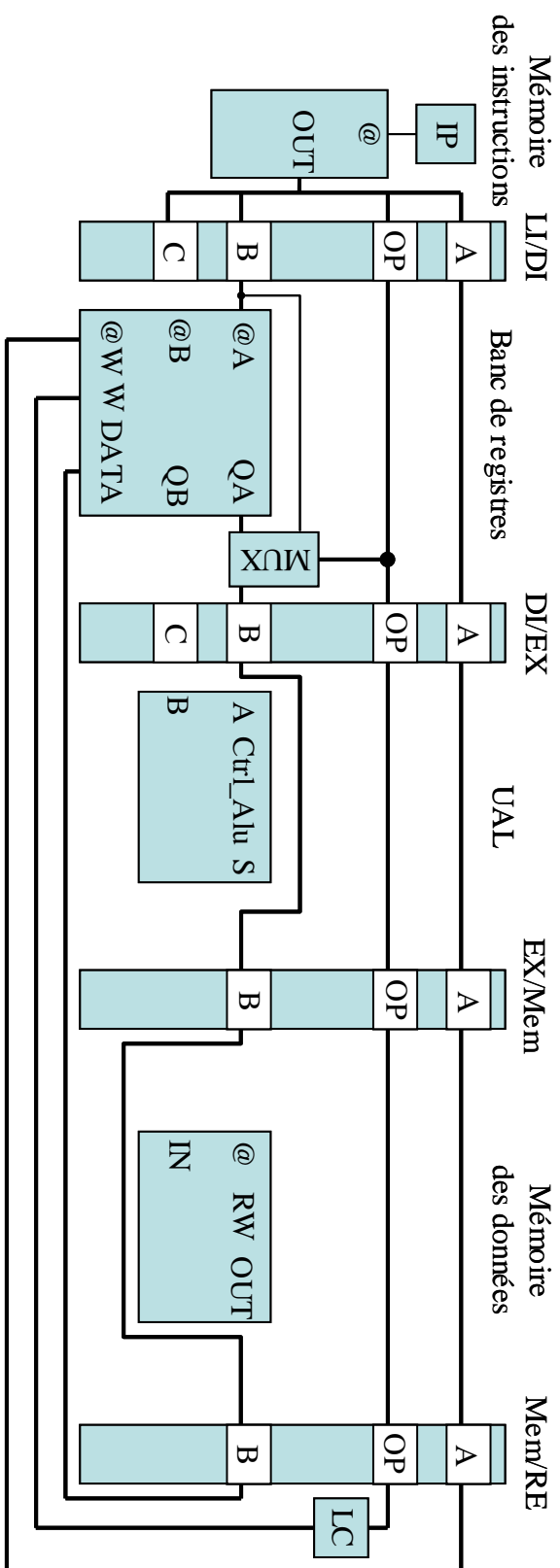


Illustration 2 : Instructions AFC, COP

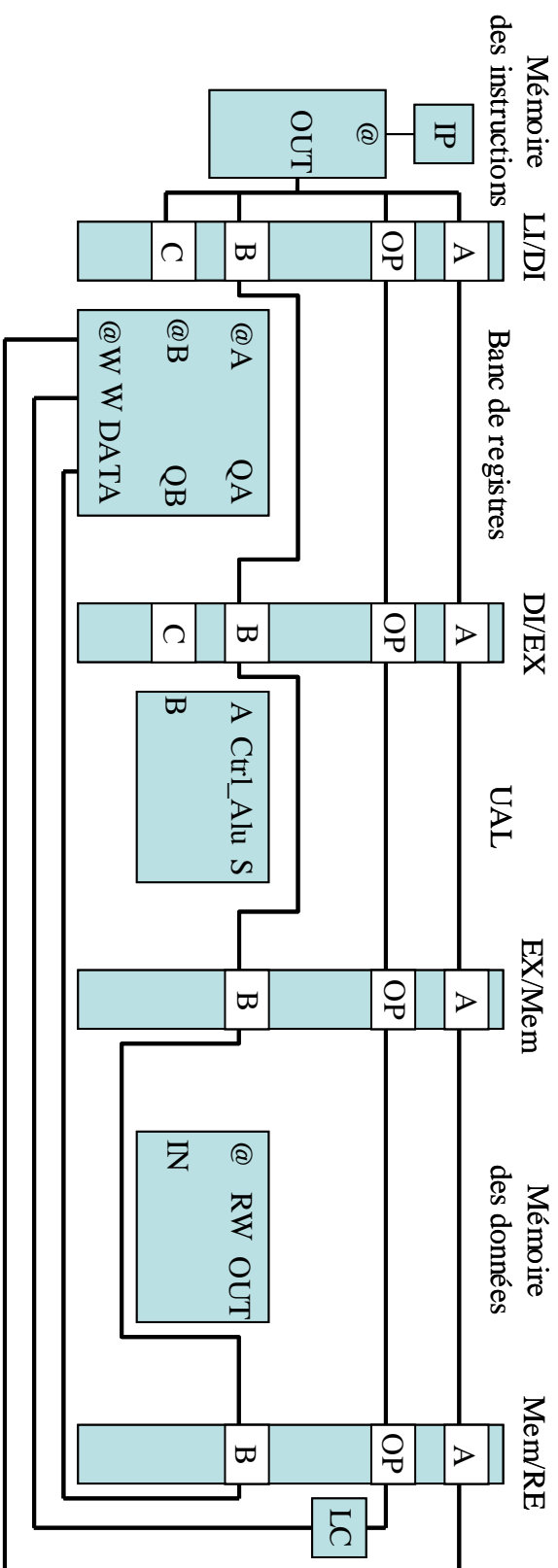


Illustration 1 : Instruction AFC

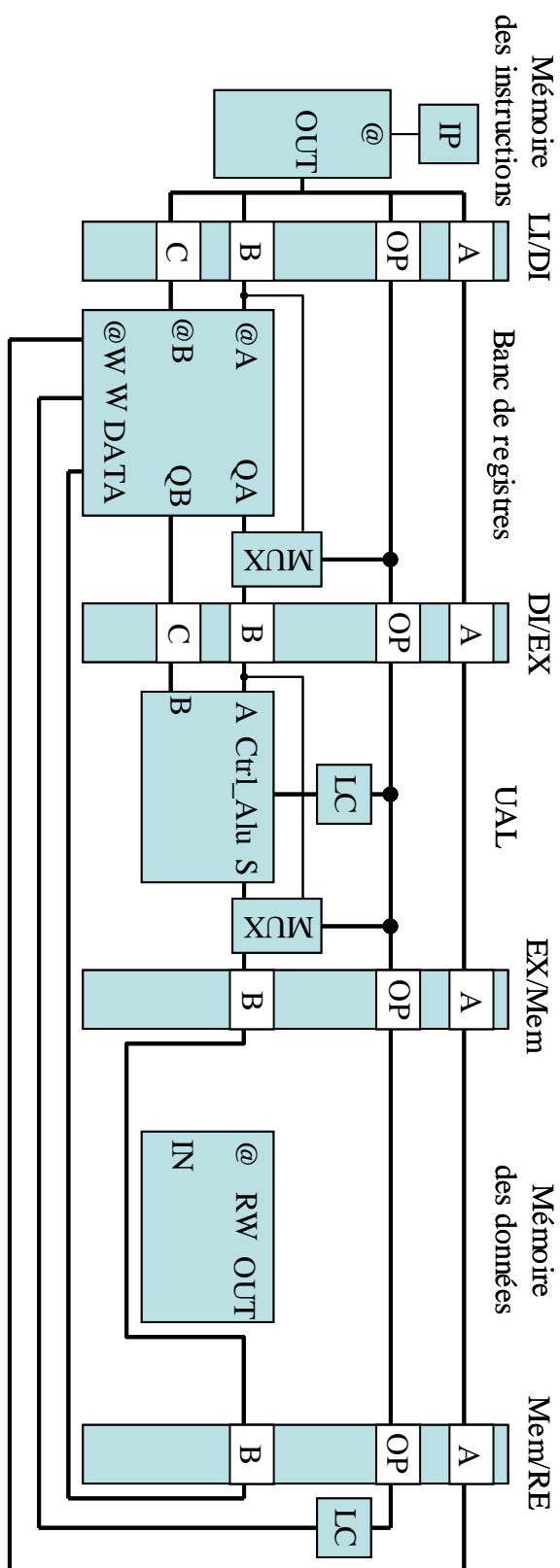


Illustration 3 : Instructions AFC, COP, ADD, MUL, DIV, SOU

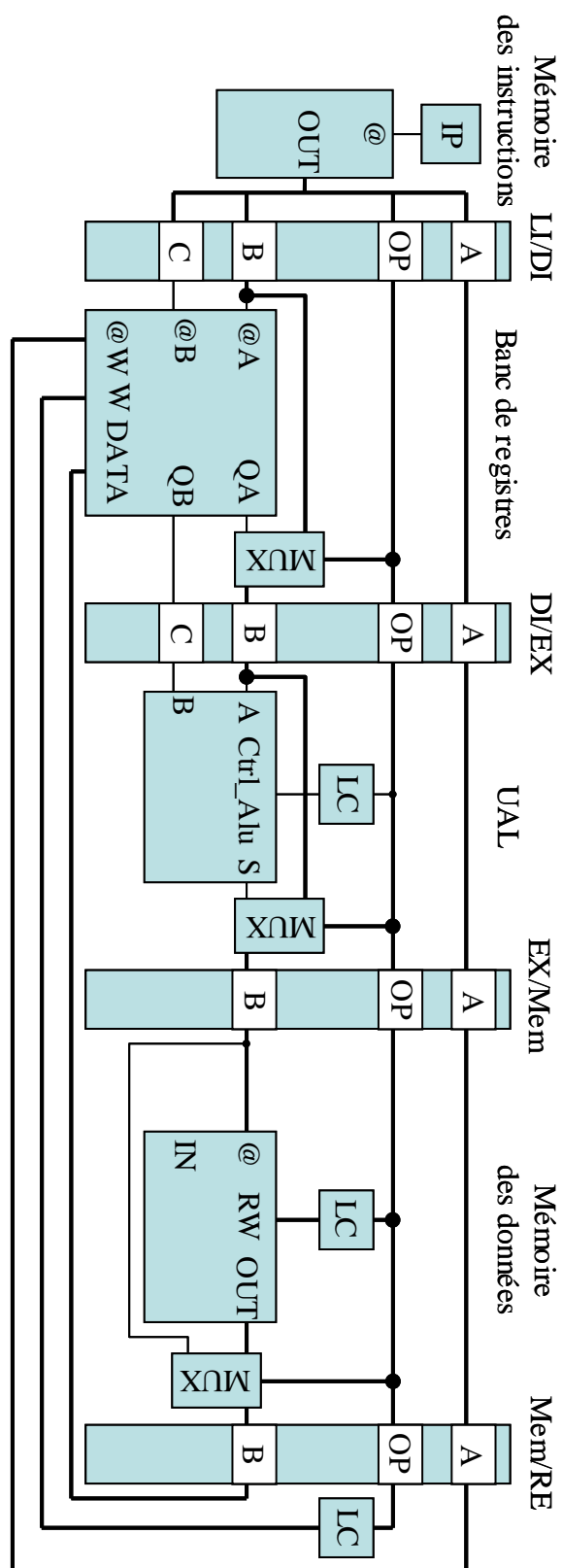


Illustration 4 : Instructions AFC, COP, ADD, MUL, DIV, SOU, LOAD

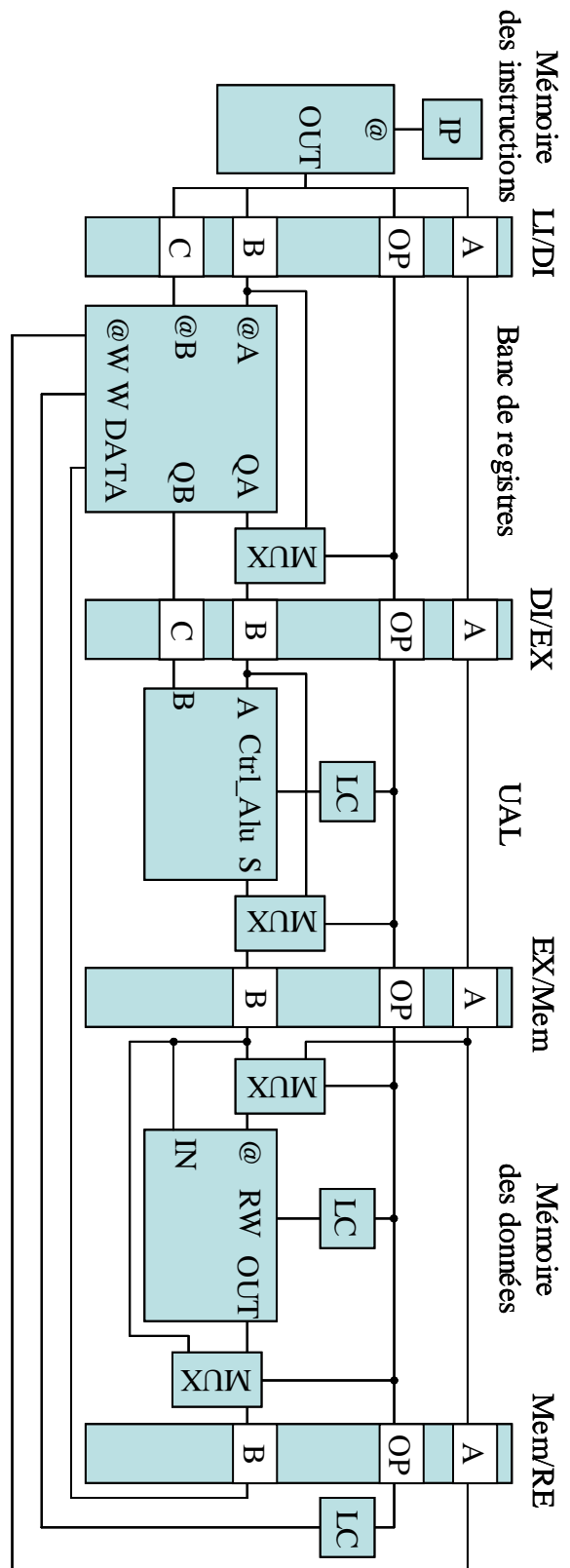


Illustration 6 : Chemin des données

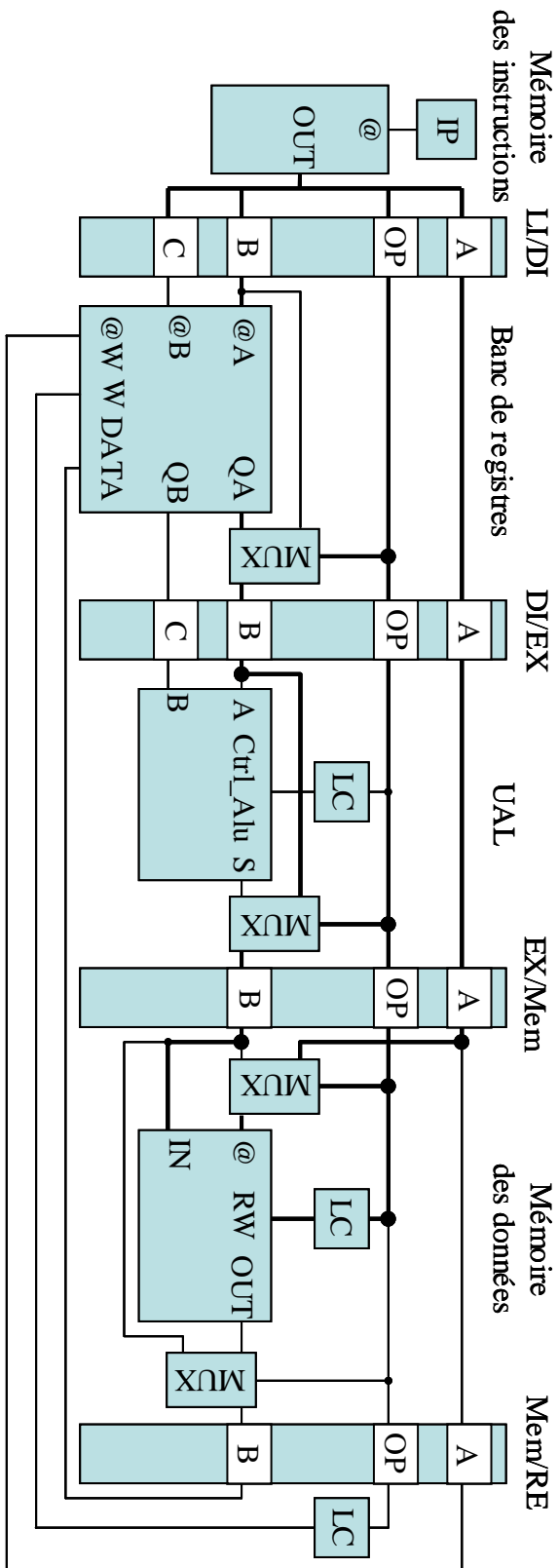


Illustration 5 : Instructions AFC, COP, ADD, MUL, DIV, SOU, LOAD, STORE

Unité de contrôle

Parallèlement au chemin de données, vous aurez à compléter la procédure de contrôle dont le rôle est de définir l'état des signaux de commandes des multiplexeurs, des bancs de mémoire et autres éléments propres à chaque étage du pipeline. Ce contrôle est fonction des instructions en cours d'exécution.

Gestion des aléas – **optionnel en 2019-2020**

Par la suite, vous serez confrontés à des situations d'aléas. Il existe deux types d'aléas : les aléas de données et les aléas de branchement. Nous nous intéresserons aux aléas de données. Afin d'illustrer ces situations, considérons le programme suivant :

```
...  
NOP  
NOP  
[R1] ← 12  
[R2] ← [R1]  
NOP  
NOP  
...
```

Supposons que le registre *R1* contient 14 et le registre *R2* contient 8. A l'issu de ce programme, la valeur affectée au registre *R2* doit être 12. Or, lorsque l'écriture associée à la première instruction sera réalisée (étage 5), la valeur associée au registre *R1* dans l'étage 4 ne sera pas encore 12 mais 14. La figure 9 présente ce cas de figure. Pour résoudre ce problème, nous vous conseillons d'utiliser une entité de gestion des aléas, qui aura pour rôle d'injecter des « bulles » en cas de détection d'aléas. Ces bulles permettent de temporiser l'exécution des étages inférieurs lors de la détection d'un conflit avec les étages supérieurs. Pour l'exemple précédent, lorsque la première instruction se situe au niveau du second pipe-line – et la seconde instruction au niveau du premier pipe-line – la comparaison entre les données du premier pipe-line et du second pipe-line permet de détecter cet aléa :

$$OP_{DI_EX} = AFC \text{ and } OP_{LI_DI} = COP \text{ and } A_{DI_EX} = B_{LI_DI}$$

Pour ce cas, l'horloge du premier pipe-line est inhibée, le temps que la première instruction finisse. Attention toutefois que le « vide » engendrée par cette temporisation n'altère pas l'état des registres. Pour éviter ceci, il faut injecter un *NOP* (*NO-OPERATION*), qui constitue la bulle. La résolution des autres cas d'aléas suit le même raisonnement.

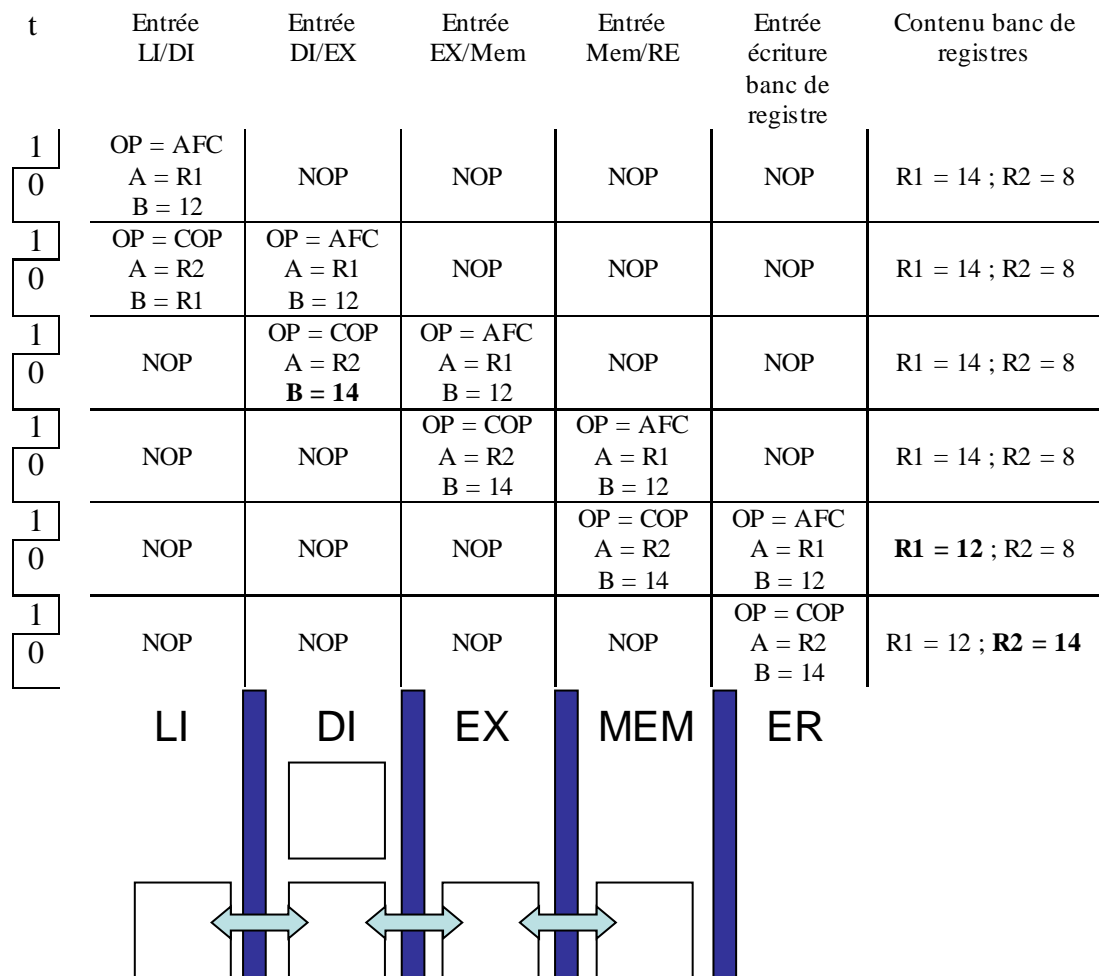


Figure 9 : Exemple d'aléa de données