INSA de Toulouse 4IR-TDB

**Rapport projet système informatique**

JAUFFRET Pascal GRÉVIN Arthur

01/06/2015

[Introduction](#h.nsc2m8iqa652)

[**I- Compilateur**](#h.63tt7bqef5vr)

[A - Expressions Arithmétiques](#h.osvvpdlbuune)

[B - Les Structures de contrôles](#h.cqfwugkje94f)

[C- Fonction](#h.23n2ufni6l7q)

[D - Interpréteur](#h.5cw08qyjv886)

[**II- Processeur**](#h.jgkhviqeik93)

[A- Chemin sans Aléa, Affectation](#h.yhv8da78wijv)

[B - Aléas avec les instructions ADD,SUB,MUL](#h.p1gqwse21hph)

[C - Mémoire et Instruction LOAD](#h.fk44dd46wqwj)

[D - Instruction STORE](#h.rjqqkdeopbi)

# 

# Introduction

Dans ce rapport nous allons vous présenter le fonctionnement de notre projet système informatique, et en quoi il diffère des spécifications initiales du polycopié. Le lecteur pourra ainsi voir les choix de conception que nous avons effectués et les problèmes que nous avons rencontré.

# I- Compilateur

Une des grosses différences entre notre assembleur et celui de la spécification, est que nous avons fait le choix de réaliser un assembleur orienté-registre d’entrée de jeu, nous n’avons donc pas eu besoin d’utiliser de cross-assembleur.

## A - Expressions Arithmétiques

Pour implémenter les expressions arithmétiques, nous avons procédé de façon assez classique. On a utilisé une table des symboles, et de la récursivité. Dans la table des symboles, on stockera l’adresse en mémoire correspondant à nos variables ainsi que leur identifiant. On stockera aussi les variables temporaires qui sont crées lors du calcul d’expressions arithmétiques.

Ainsi notre grammaire fonctionne de la façon suivante, on va explorer l’expression arithmétique jusqu'à retomber soit sur un identifiant, soit sur un nombre décimal. On va ensuite créer un symbole temporaire, puis on va reduce l’expression en remontant, et en effaçant les symboles temporaires au fur et a mesure.

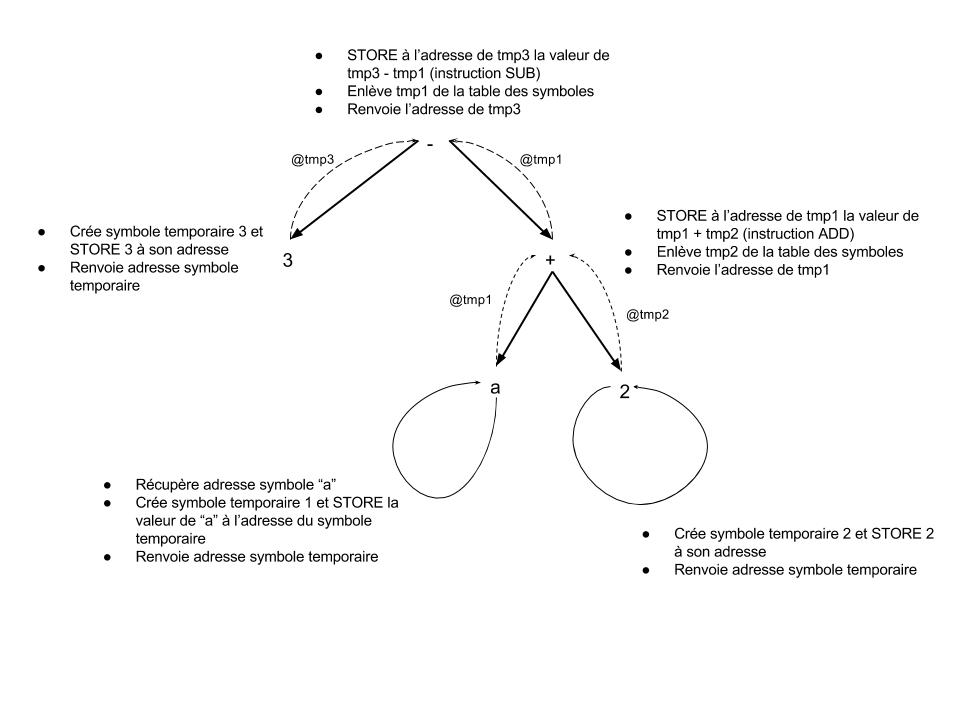
Les symboles correspondant aux identifiants sont crées lors des affectations, on peut le voir dans la grammaire suivante :

|  |
| --- |
| DECLARATION : TYPE tID{  add\_symbole($2,index\_mem++);  $$=get\_adr\_mem(get\_next()-1); }; |

Et lors d’une affectation, on store ,à l’adresse du symbole, la valeur de l’expression, et on supprime la variable temporaire crée, on permet ainsi de nettoyer la table des symboles( vu qu’à chaque fois il y a au minimum une variable temporaire crée).

|  |
| --- |
| AFFECTATION :tID tEQ EXPRESSION{  symbole \* mon\_symbole;  compteur ++;  fprintf(fichier,"%d : LOAD R1 %d\n",compteur,$3);  compteur ++;  fprintf(fichier,"%d : STORE %d R1\n",compteur,get\_symbole($1,&mon\_symbole)); enlever\_symbole(); |

On peut voir dans l’exemple suivant l’évaluation de l’expression arithmétique 3-a+2. On observe bien que lorsque l’on descend dans l'évaluation de l’expression, et que l’on retrouve soit un identifiant, soit un nombre décimal, on crée un nouveau symbole temporaire, et on store la valeur à la bonne adresse. Lorsque l’on effectue une opération, on stocke le résultat dans la première variable temporaire, et on efface ensuite la seconde( en dépilant la table des symboles, c’est à dire en enlevant le dernier élément ajouté).

La priorité des opérateurs (addition, soustraction, negation, et multiplication) est gérée par l'enchaînement successifs des lefts en début de fichiers.

|  |
| --- |
| %left tADD tSUB  %left tMUL tDIV  %left NEG |

L’opérateur %prec permet de créer la règle qui gère les nombres négatifs.

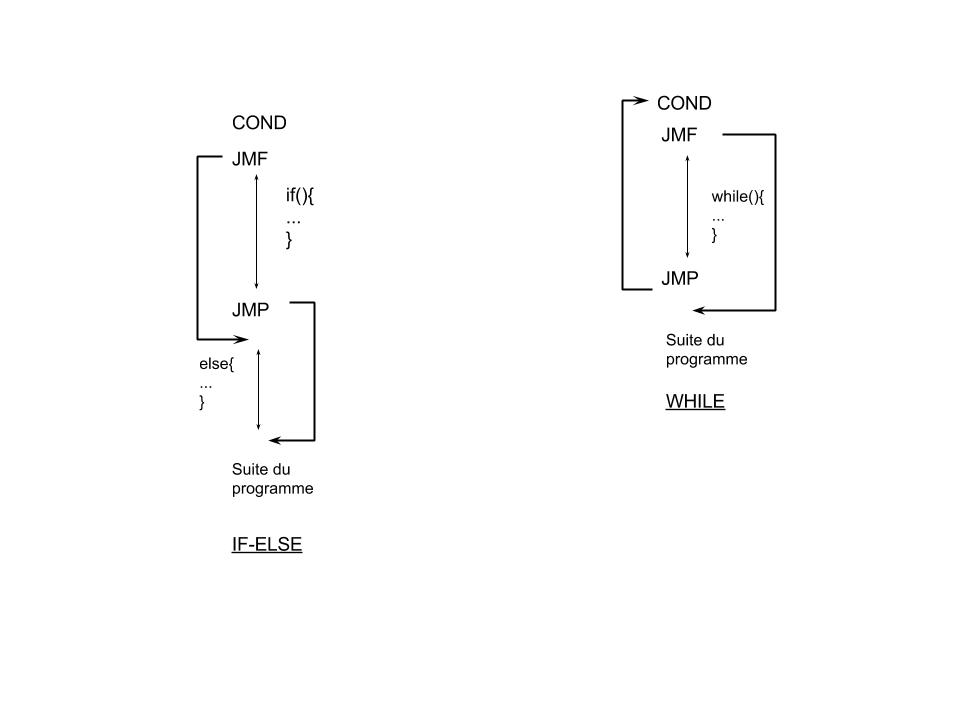
|  |
| --- |
| tSUB EXPRESSION %prec NEG |

Ainsi on peut évaluer les expressions arithmétiques en gérant correctement les priorités, l’expression 3 x 2 + -3 renverra bien 3.

## B - Les Structures de contrôles

Notre compilateur permet la gestion des ifs et des whiles, pour réaliser cette fonctionnalité, on a utilisé deux nouvelles tables : la tables des ifs et la table des whiles. Ces tables contiennent pour chaque instruction de saut, correspondant soit à des ifs ou soit a des whiles, la ligne d’origine, et la ligne de destination.

Lors de l’analyse syntaxique, on produit ainsi un premier fichier assembleur qui contient des trous. Au fur et a mesure du parsing, on remplit les tables avec les bons numéros de lignes, et a la fin du parsing on effectue une deuxième passe sur ce fichier assembleur pour compléter les trous. Le schéma suivant explique la correspondance entre IF ELSE, WHILE, et code assembleur.



La problématique n’est pas la même lors d’un if, ou d’un while. Dans les deux cas on remplit toujours le premier JMF à l’aveugle, mais le JMP par contre, dans le cas du if, on ne connaît pas la destination au moment du parsing alors que dans le cas du while, cette information a déjà été obtenu précédemment (l’adresse de la condition).

Pour résoudre cette problématique, on va utiliser deux structure similaire ressemblant à une pile dans laquelle pour chaque type de saut “JMF” ou “JMP” on va stocker au fur et à mesure de leur découverte à la fois la ligne d’origine, et la ligne de destination. Lors de la deuxième passe on pourra produire un fichier assembleur complet grâce aux informations contenues dans ces tables.

Le type d’objet contenu dans cette pile contient plusieurs informations : Le type de saut, le numéro du saut, la ligne de destination et d’origine et un paramètre adr\_jmf contenant l’adresse de la condition.(car notre assembleur est orienté registre)

On a ainsi pu coder la grammaire suivante du if :

|  |
| --- |
| IF :tIF tPAR\_OUVR EXPRESSION tPAR\_FERM{  compteur++;  fprintf(fichier,"%d : LOAD R1 %d\n",compteur,$3);  compteur++;  add\_from(compteur,JNE,$3);  fprintf(fichier,"%d : JMF R1 \n",compteur);  }BODY{  num++;  remplir(compteur+1,num);  print\_tab();}  |IF tELSE{  add\_offset();  compteur++;  add\_from(compteur,JMP,-1);  fprintf(fichier,"%d : JMP\n",compteur);  }BODY{  remplir(compteur+1,num);  num++;  }; |

On voit bien que lorsque l’on rentre dans un if, on insère un premier JMF dans la structure (type JNE, ligne d’origine compteur, et adr\_jmf vaut l’adresse de la variable temporaire contenant la condition).

Si jamais on a un IF simple, à la fin du body, on peut remplir le dernier élément non rempli et donner la ligne de destination.

Si jamais on a un else, on doit par contre rajouter le JMP dont on connait l’origine, et dans le body on peut remplir la destination. Il faut aussi pouvoir rajouter un offset au dernier jmf ajouté. Pour trouver quel est le bon jmf, on utilise les numéros, on cherche le dernier jmf ajouté donc celui avec le numéro maximum, puis on lui ajoute un offset dans sa destination de façon à ce qu’il saute une ligne plus loin pour éviter le JMP.

Lors de l’élaboration de notre grammaire, au début nous avions un problème de conflit reduce-reduce, nous l’avons résolu en factorisant notre grammaire d’origine, pour arriver à celle-ci.

Le while fonctionne de façon assez similaire comme on peut le voir dans la grammaire suivante :

|  |
| --- |
| WHILE : {  printf(" Début while : %d \n", compteur); add\_while(compteur+1,-1,JMP,-1);  }tWHILE tPAR\_OUVR EXPRESSION tPAR\_FERM{  compteur ++;  fprintf(fichier,"%d : LOAD R1 %d\n",compteur,$4);  compteur++;  fprintf(fichier,"%d : JMF R1 \n",compteur);  add\_while(-1,compteur,JNE,$4);  } BODY{ compteur++;  fprintf(fichier,"%d : JMP\n",compteur);  print\_tab\_while();  remplir\_while\_to(compteur+1);  remplir\_while\_from(compteur);  }; |

On voit que lors d’un while, on ajoute un objet JMP dans la structure, dont on ne connait pas l’origine, mais dont on connaît la destination. Puis après l'évaluation de la condition, on rajoute le JMF, un peu comme dans le if, on ne connaît pas la destination mais l’on connaît l’origine.

Puis à la fin du body, on peut remplir le JMF en lui rajoutant sa destination, puis le JMP en lui rajoutant son origine.

Lors de la deuxième passe on pourra ainsi réécrire proprement l’assembleur avec toutes les bonnes origines et destinations. Cette réécriture se fait à la fin du premier parsing, on reprend ligne par ligne chacune des instructions, et vérifie si elle appartient à l’une ou l’autre des tables, si oui, on la réécrit, en fonction du type de saut (stocké dans la structure : JNE pour les JMF, ou JMP pour les JMP).

On peut voir cette réecriture dans les lignes de codes suivantes :

|  |
| --- |
| while(fgets(ligne,size,fichier)!=NULL){ to=get\_to(compteur\_ligne,&type,&adr\_jmf\_if); to\_wh=get\_while\_to(compteur\_ligne,&type\_wh,&adr\_jmf\_while);  if((to==-1)&&(to\_wh==-1)){  fputs(ligne,fichier\_fin); }  else{  if(to != -1){  switch (type){  case JMP :  fputs("",fichier\_fin); fprintf(fichier\_fin,"%d : JMP %d\n",compteur\_ligne,to);  break;  case JNE :  fputs("",fichier\_fin);  fprintf(fichier\_fin,"%d : JMF R1 %d\n",compteur\_ligne,to);  break; } }else{  switch (type\_wh){  case JMP :  fputs("",fichier\_fin);  fprintf(fichier\_fin,"%d : JMP %d\n",compteur\_ligne,to\_wh);  break;  case JNE :  fputs("",fichier\_fin); fprintf(fichier\_fin,"%d : JMF R1 %d\n",compteur\_ligne,to\_wh);  break;  }  } }  compteur\_ligne++; } printf("FIN\n");}; |

## C- Fonction

Pour repérer les fonctions, nous avons défini une table des fonctions qui permet de faire correspondre le nom de la fonction avec son adresse, c’est à dire la ligne où se trouve la première instruction de la fonction. On ajoute une nouvelle fonction dans cette table quand on détecte une nouvelle fonction dans le yacc. De plus, on va remettre à 0 l’index de la table des symboles.

|  |
| --- |
| FONCTION :TYPE tID{  init\_table();  add\_fonction($2,compteur+1);  //printf("%d",compteur);  init\_pile\_arg();  }tPAR\_OUVR PARAMETRES tPAR\_FERM tACC\_OUVR INSTRUCTIONS tACC\_FERM; |

On détecte les paramètres demandés dans la fonctions grâce aux yacc. Chaque paramètre est une déclaration. DÉCLARATION nous renvoie une adresse mémoire; on va donc empiler chaque adresse mémoire, dans une pile, qui sera dépilé quand on rentrera dans le corp de la fonction. C’est emplacement mémoire seront les emplacements où seront copié les variable passé en argument. La pile utilisé est une pile classique défini dans un fichier .h.

|  |
| --- |
| PARAMETRES : PARAMETRE  |PARAMETRE tVIRG PARAMETRES;  PARAMETRE : DECLARATION{push\_arg($1);}; |

|  |
| --- |
| tACC\_OUVR{  while(get\_index\_pile\_arg()!=0){  //compteur++;  fprintf(fichier,"%d : POP R1\n",++compteur);  fprintf(fichier,"%d : STORE %d R1\n",++compteur,pop\_arg());  } |

On va donc dépiler chaque paramètre. Pour chaque paramètre, on affiche l’instruction POP R1 et STORE emplacement R1. Ainsi en assembleur, on récupère l’argument que l’on avait pushé dans la pile et on le met à l’emplacement mémoire que l’on avait défini dans les déclarations dans les paramètres.

Pour gérer les fonctions sur notre compilateur, nous avons fait le choix d’utiliser un système avec une pile. Les arguments sont mis dans la piles à l’appel de la fonction avec l’instruction assembleur PUSH, puis dépilés dans la fonction. Nous empilons aussi l’adresse de retour, c’est à dire le numéro de la ligne où on doit retourner après avoir effectué les instructions de la fonction, ainsi que l’adresse mémoire où sera stocké la valeur de retour.

|  |
| --- |
| RETURN : tRETURN EXPRESSION{  fprintf(fichier,"%d : LOAD R1 %d\n",++compteur,$2);  fprintf(fichier,"%d : POP R2\n",++compteur);  fprintf(fichier,"%d : STORE R2 R1\n",++compteur);  }; |

A la fin d’un fonction, s’il est nécessaire de retourner une valeur, on le fera grâce aux RETURN. On va en effet, récupérer l’adresse mémoire de la valeur de retour qui se trouvait dans la pile, grâce à l’instruction assembleur POP R2, puis on va passer la valeur de l’expression de retour, à l’adresse qui se trouve dans R2. Cette combinaison d’instruction nous permet d’avoir des valeurs de retour à la suite d’une exécution de fonction.

|  |
| --- |
| tACC\_FERM{  fprintf(fichier,"%d : POP R1\n",++compteur);  fprintf(fichier,"%d : JMPR R1\n",++compteur);  } |

Pour la fin de la définition de la fonction, on dépile une dernière fois la pile avec l’instruction POP R1 et on va faire un JUMPR à l’adresse contenu dans R1. Ceci permettra un retour à l’instruction qui se situe après l’appel de fonction.



## D - Interpréteur

Pour réaliser l’interpréteur, nous avons du mettre au point un système permettant d’effectuer les différents jumps.Les jumps rendent la lecture du fichier assembleur non linéraire, ce qui pose un problème pour une simple lecture avec yacc. A la suite d’un jump on va devoir sauter des lignes et ne pas interpréter l’instruction suivante.

Pour contourner ce problème, l’interpréteur va fonctionner en deux parties. La première est la simple reconnaissance des différentes instructions assembleurs.

Pour cela nous avons utilise lex qui reconnaît les différents tokens :

|  |
| --- |
| NUMERO [0-9]+  %%  AFC return tAFC;  STORE return tSTORE;  LOAD return tLOAD;  ADD return tADD;  MUL return tMUL;  SUB return tSUB;  DIV return tDIV;  NEG return tNEG;  EQ return tEQ;  R1 return tR1;  R2 return tR2;  SUP return tSUP;  INF return tINF;  END return tEND;  JMF return tJMF;  JMP return tJMP;  START return tSTART;  JMPR return tJMPR;  PUSH return tPUSH;  POP return tPOP;  {NUMERO} {yylval.numero=atoi(strdup(yytext));return tNUMERO;} |

Ensuite, on utilisera yacc qui permettra de reconnaître chaque instructions.

|  |
| --- |
| INSTRUCTIONS : INSTRUCTION INSTRUCTIONS  | INSTRUCTION;    INSTRUCTION : tNUMERO tAFC tR1 tNUMERO  | tNUMERO tAFC tR2 tNUMERO  | tNUMERO tADD tR1 tR2  |tNUMERO tSUB tR1 tR2  |tNUMERO tDIV tR1 tR2  | tNUMERO tMUL tR1 tR2  |tNUMERO tLOAD tR1 tNUMERO  |tNUMERO tLOAD tR2 tNUMERO  |tNUMERO tSTORE tNUMERO tR1  |tNUMERO tSTORE tNUMERO tR2  |tNUMERO tSTORE tR2 tR1  |tNUMERO tNEG tR1  |tNUMERO tEQ tR1 tR2  |tNUMERO tSUP tR1 tR2  |tNUMERO tINF tR1 tR2  |tNUMERO tJMF tR1 tNUMERO  |tNUMERO tJMP tNUMERO  |tNUMERO tEND  |tNUMERO tSTART  |tNUMERO tJMPR tR1  |tNUMERO tPUSH tR1  |tNUMERO tPOP tR1  |tNUMERO tPOP tR2; |

Pour faire le lien avec la deuxième partie de notre interpréteur, nous avons décidé d’utiliser une pile qui empilera chaque instruction à chaque fois que le yacc en reconnaîtra une.

|  |
| --- |
| #ifndef STACK\_H  #define STACK\_H  #define LOAD 1  #define STORE 2  #define AFC 3  #define ADD 4  #define MUL 5  #define SUB 6  #define DIV 7  #define NEG 8  #define EQ 9  #define JMP 10  #define JMF 11  #define R1 12  #define R2 13  #define INF 14  #define SUP 15  #define END 16  #define JMPR 17  #define START 18  #define PUSH 19  #define POP 20  typedef struct instruction{  int numero;  int type;  int var1;  int var2;  } ins;  typedef struct stack\_instruction{  int index;  ins tab\_ins [255];  }stack\_ins;  stack\_ins\* stack;  void init\_stack\_ins();  void push (int numero, int type, int var1, int var2);  ins\* pop();  void move\_index(int new\_index);  #endif |

Le fonctionnement est similaire à une pile classique, mais nous avons rajouté la fonction move\_index qui permet de bouger l’index de la pile à la position que l’on souhaite. Cela nous permettra de bouger le curseur quand on rencontre un JUMP. Cela sera aussi utile pour placer l’index au niveau de l’instruction START.

A chaque fois que l’on rencontre une instruction, on fera entrer dans la pile, son numéro de ligne, son type( LOAD, STORE, ADD, etc), et les 2 paramètres. S’il n’y a qu’un paramètre, nous avons décidé de passer en argument 0 par convention.

Ainsi on a ajouté le code suivant dans notre yacc :

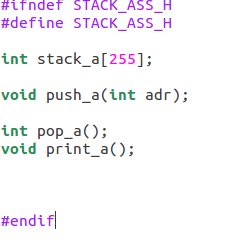
|  |
| --- |
| | tNUMERO tAFC tR2 tNUMERO{  printf("AFC R2\n");  push($1,AFC,R2,$4);  }  | tNUMERO tADD tR1 tR2{  //R1 = R1 + R2;  printf("ADD\n");  push($1,ADD,R1,R2);  } |

L’interprétation du programme empilé dans la pile se fait à la fin du yacc. Nous avons défini un tableau, tab\_mem[1024] qui représente les emplacements mémoire. Nous avons aussi défini 2 variables globales memR1 et memR2 qui représente les registres R1 et R2.

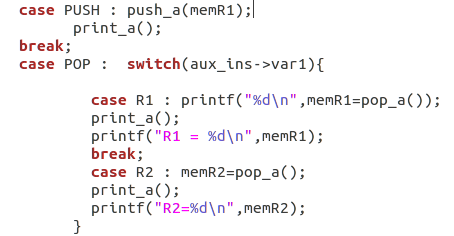


Nous dépilons donc la pile jusqu’à arriver à l’instruction END.

Pour gérer les instructions POP et PUSH, nous avons simulé une pile.



Le fonctionnement est celle d’un pile standard. On va insérer dans la pile des valeurs quand on dépilera une instruction PUSH et ressortir la dernière valeur empilé, à la suite d’une instruction de type POP.



# II- Processeur

L’implémentation du processeur s’est faite grandement en suivant les indications du sujet. Notre code assembleur étant déjà d’origine orienté-registre, nous n’avons pas eu à utiliser de cross-assembleur et avons pu nous simplifier la conception.

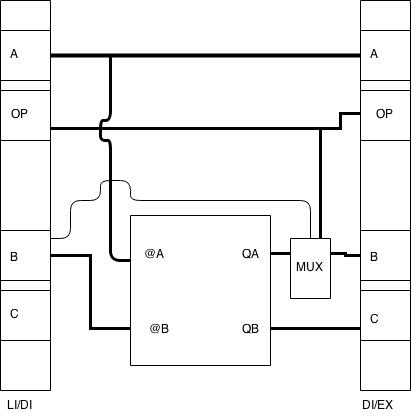
Nous avons implémenté les composants classiques du processeur, c’est-à-dire un compteur d’instructions, une mémoire d’instructions, des pipes, un banc de registre, des mutliplexeurs, une ALU et une mémoire de données.

Nous n’utilisons pas l’instruction COP dans notre compilateur donc nous ne l’avons pas intégré dans le processeur. Nous avons intégré AFC, ADD, SUB, MUL, LOAD et STORE.

## A- Chemin sans Aléa, Affectation

Nous avons légèrement dévié de la spécification initiale pour le codage des instructions arithmétiques, en effet dans notre assembleur nous utilisons le format suivant ADD R1 R2, qui met automatiquement le résultat de l’addition dans R1, on a donc dû adapter le chemin de données en conséquence.

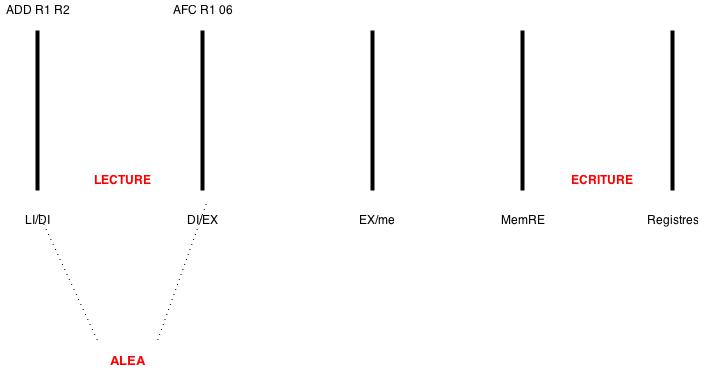
Le chemin pour effectuer l'instruction AFC est assez classique, on va simplement effectuer une écriture dans les registres.



## B - Aléas avec les instructions ADD,SUB,MUL

En voulant implémenter les instructions suivantes ADD, SUB, MUL, nous nous somme confrontés aux problèmes des aléas. En effet ADD R1 R2 demande 2 lectures dans le banc de registre et une écriture à la fin du chemin de données.

Il y a aléa car si ADD R1 R2 suit une instruction qui va écrire dans le banc de registre comme AFC R1 06, l’écriture ne sera pas encore faite quand la lecture dans le registre sera effectuée.



On remarque donc qu’il faut espacer ces instructions pour que le fonctionnement du processeur corresponde au fonctionnement attendu. Pour les espacer, il faut insérer des instructions neutres, NOP, qui n’effectue aucune tâche. Cela correspond à des 0 pour le code opération.

La solution appliquée, est de détecter les aléas. Pour détecter les aléas nous avons d’abord pensé à une logique basée sur les codes opérations. Ainsi si nous détectons un AFC suivi d’un ADD par exemple, il y a aurait un aléa.

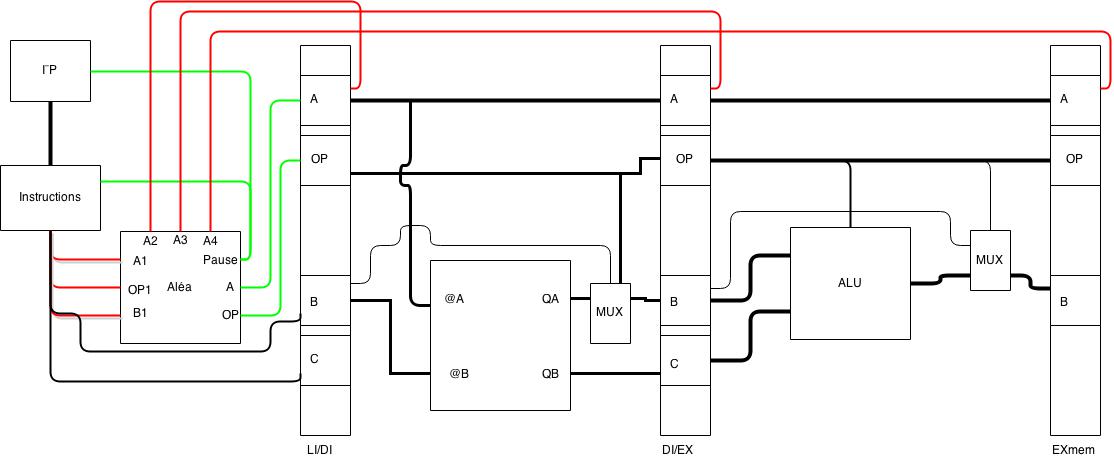
Cette solution n’est pas optimale car on peut avoir AFC R1 02 suivit d’un ADD R2 R3 et ne pas avoir d’aléa car la lecture dans les registres ne correspond au registre de l’écriture.

Nous avons donc adapté cette logique pour la rendre plus optimale. Pour chaque instruction qui demande une lecture dans les registres, nous regardons si son paramètre A(renommé A1) et B(renommé B1) correspond au paramètre A à la sortie des 3 prochains pipes, c’est à dire à la sortie de LI/DI(A2), à la sortie DI/EX(A3), à la sortie de EXmem(A4). Il n’y a pas d’aléa pour les instructions qui ne lisent pas dans les registres.

|  |
| --- |
| (A1=A2 or A1=A3 or A1=A4 or B1=A2 or B1=A3 or B1=A4) and OP1/=AFC and OP1/=00 |

Une fois l’aléa détecté, on va envoyer des 00 dans les parties OP1 et A, et mettre en pause le pointeur d’instruction et la mémoire d’instruction.

Avoir une pause bien synchronisée avec la détection des aléas fut un des problèmes de la mise en place. La bonne solution fut de placer le composant de détection d’aléas avant le registre LI/DI, à la sortie de la mémoire d’instructions. Cela permet d’avoir la pause au bon moment.



On peut voir en annexe des captures d’écrans montrant les aleas, et l’apparition des bulles (00) suite à l’apparition d’un ADD(01) après un AFC(06), qui avait déjà un aléa avant

## C - Mémoire et Instruction LOAD

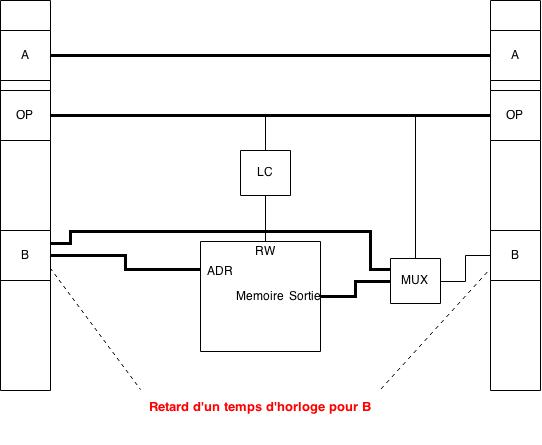
Pour implémenter l’instruction LOAD, nous devons pour la première fois utiliser le composant mémoire. Pour rappel l’instruction LOAD R1 04 va mettre le contenu de l’emplacement mémoire 04 dans le registre R1. Il y a donc une lecture dans la mémoire puis une écriture dans le registre.

Le premier problème que nous avons dû résoudre est la mise à jour de notre logique d’aléa. Pour l’instant nous détectons un aléa si le paramètre A ou B à la sortie de la mémoire d’instruction est égal à un paramètre A à la sortie d’un des 3 prochains pipes. Mais l’instruction LOAD ne va jamais déclencher d’aléa car il n’y a pas de lecture de registre.

Nous avons donc adapté notre logique pour ne pas déclencher l’insertion de NOP quand il y a un LOAD :

|  |
| --- |
| (A1=A2 or A1=A3 or A1=A4 or B1=A2 or B1=A3 or B1=A4) and OP1/=AFC and OP1/=NOP and OP1/=LOAD |

La mise en place de l’instruction LOAD a demandé la mise en place d’un multiplexeur à la sortie de la mémoire. Cela permettra de choisir la sortie mémoire ou la sortie du pipe précédent suivant l’instruction. Pour l’instant seul l’instruction LOAD va déclencher la récupération de la lecture en mémoire. Un contrôle est aussi fait pour déclencher la lecture ou l’écriture sur la mémoire.

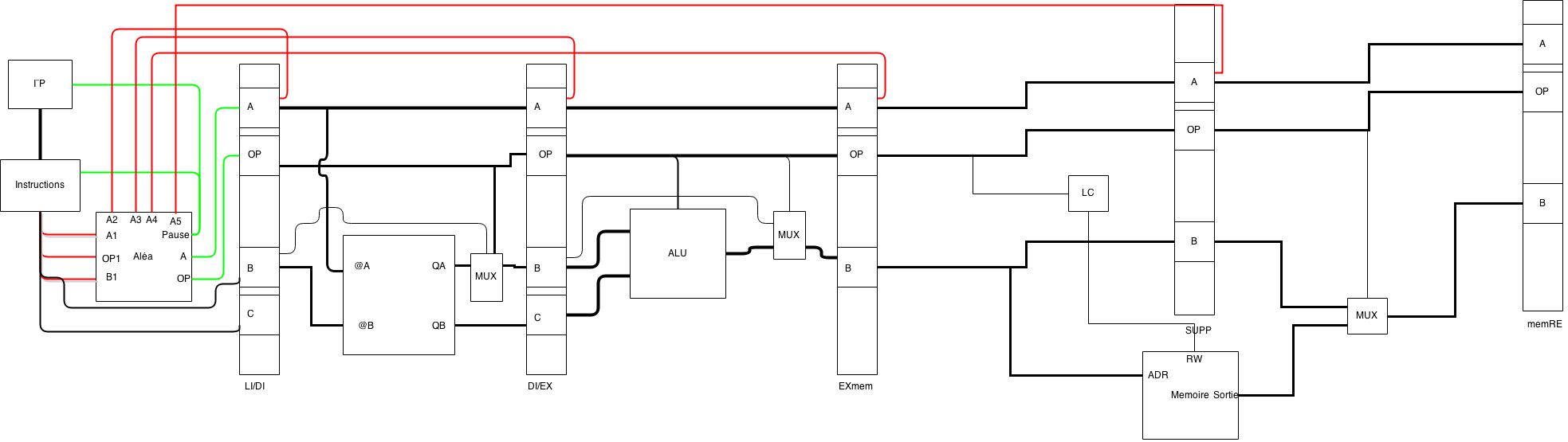


Un nouveau problème fut détecté à ce moment là. La lecture dans la mémoire était synchronisé sur l’horloge, elle retardait donc le cheminement des instructions dans les pipes. Ainsi les parties A et OP arrive un temps d’horloge avant la partie B qui était retardée dans la mémoire.

La solution que nous avons décidé d’implémenter fut l’insertion d’un nouveau pipe similaire à MemRE et EXMem qui va retarder le cheminement d’un temps d’horloge mais va permettre la synchronisation de ce cheminement.

Cette solution a eu des répercussions sur la détection d’aléa. En effet si le cheminement est plus long d’un temps d’horloge, il faut récupérer la partie A à la sortie du nouveau pipe.

Ainsi en entrée de la gestion d’aléa, on aura A1,B1,OP1 de la sortie de la mémoire d’instruction, A2 sortie de LI/DI, A3 sortie de DI/EX, A4 sortie de EXMem, A5 sortie du nouveau pipe appelé SUPP.



## D - Instruction STORE

L’instruction STORE permet de mettre le contenu d’un registre dans un emplacement mémoire. STORE 07 R1 va donc lire le registre R1 et mettre son contenu dans l’emplacement mémoire 07. Il y a donc la possibilité de déclencher un Aléa mais que sur la partie B de l’instruction. Nous avons donc ajusté notre logique d’aléa pour que la détection sur la partie A de l’instruction ne s’effectue pas pour un code opération STORE.

|  |
| --- |
| (((A1=A2 or A1=A3 or A1=A4) and OP1/=STORE)) or B1=A2 or B1=A3 or B1=A4) and OP1/=AFC and OP1/=NOP and OP1/=LOAD |

Nous avons aussi ajusté le multiplexeur en sortie de la lecture des registres. En effet, dans notre configuration la sortie A du pipe LI/DI est reliée à l’entrée A et la sortie B à l’entrée B. Actuellement nous choisissons la sortie QA du banc de registre pour la lecture. Mais dans le cas d’un STORE nous avons besoin de la sortie QB.

Nous avons donc changé notre multiplexeur de manière à prendre la sortie QB en entrée et de choisir cette voie si le code opération correspond à STORE.

Un multiplexeur va être aussi requis à l’entrée de la mémoire. Si nous avons un LOAD, l'adresse se trouvera dans la partie B du pipe EXmem mais si nous sommes en présence d’un STORE, l'adresse se trouvera dans la partie A.

Un multiplexeur choisira le bon argument à l’entrée adresse de la mémoire suivant le code opération. On trouvera sur la dernière page, le schéma représentant notre circuit final en grand.

