République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de 8 Mai 1945 – Guelma -

Faculté des Mathématiques, d'Informatique et des Sciences de la matière

Département d'Informatique


# Mémoire de Fin d'études Master



**Filière :** Informatique

**Option : Informatique Académique**

**Thème :**

# Prototypage Rapide des Microprocesseurs

**Encadré Par :**

Berrehouma Nabil

**Présenté par :**

**Maghmouli Hadjer**

**Juin 2013**

# Table des matières

# Table des figures

# Liste des tableaux

# TABLE DES ABREVIATIONS

| TABLE DES ABREVIATIONS | |
|---|---|
| ADL | Langages de description d'architecture |
| PI | instructions partielles |
| ILP | Instruction level parallelism |
| ArchC | Langage de description d'architecture |
| TLM | Transaction-Level Modeling |
| MIPS | Microprocessor without Interlocked Pipeline Stages |
| ISA | Instruction Set Architecture |
| ISS | Instruction Set Simulator |
| SOC | System On Chip |
| RTL | Resistor Transfer Level |
| HDL | Hardware Description Language |
| TLM | Transaction Level Module |
| CPU | Central Processing Unit |
| VHDL | VHSIC Hardware Description Language(VHSIC Very hight speed integreted circuit) |
| GMF | Graphical Modeling Framework |

# Introduction Générale

Avec l'augmentation des taux d'intégration des systèmes sur puces et l'accélération de leurs délais de mise en marché. La nécessité de développer des méthodes et des outils de conception des microprocesseurs adaptés à ces types de systèmes est devenue de plus en plus cruciale. La conception d'un microprocesseur est divisée souvent en deux phases : la conception de l'architecture où il est décrit les bus et leurs tailles, les registres, les pipelines, etc. ainsi que le rôle de chaque composant et ensuite le jeu d'instruction supporté par le microprocesseur conçu. La contrainte qui s'impose avec cette approche de conception est que les tests et vérifications sont faits sur le microprocesseur après son mise sur le silicium. Ce qui entraîne des retards considérables dû aux itérations et en retours en arrière si des disfonctionnements sont détectées.

Toutefois, si des simulations sont faites avant la mise sur le silicium, les erreurs sont corrigées rapidement. L'exploration des architectures est une nécessité absolue dans les environnements de conception des solutions pour les architectures des processeurs. Où un ensemble des processeurs candidats sont à considérer lors la conception d'une plateforme et le choix se fait sur des critères fixés à l'avance. Cependant, l'exploration reste une tache excessivement délicate sans le recours à des outils de simulation/vérification efficaces.

Pour répondre à cette limitation les langages des descriptions des architectures de processeurs (ADL) permettent actuellement de générer d'une manière quasi-automatique des prototypes des architectures de processeur sous forme de simulateurs de jeu d'instruction avec des outils binaires comme les assembleurs, les éditeurs de liens et les débogueurs. Bien que, ces langages facilitent la tâche d'exploration, leur inconvénient est les retours en arrière lors des paramétrages. Dans ce projet nous présentons une nouvelle méthodologie qui marie entre l'approche MDE très utilisée en génie logiciel et les langages de description d'architecture pour le prototypage et l'exploration des architectures des processeurs. Vu la contrainte du temps, nous nous limitons dans ce projet au prototype d'un processeur

type MIPS.

Le reste de ce mémoire est organisé comme suite :

Dans le premier chapitre, nous essayerons de donner un aperçu sur le domaine des langages de description et d'exploration des architectures qui est de grande importance dans la conception des processeurs .Un survol de l'état de l'art est présenté , un langage ADL sera choisi pour être utiliser dans la suite du projet.

Dans le deuxième chapitre, nous abordons en détail l'ADL ArchC et le langage de simulation des systèmes embarqués SystemC qui sont les deux langages avec lesquels nous auront développer nos prototypes.

Dans le chapitre trois, nous mettons en évidence l'apport majeur des approches de la conception basée sur les récentes méthodologies dirigées par les modèles (MDE). Nous nous intéresserons essentiellement à l'utilisation de MDE dans le contexte d'exploration d'architectures.

Le quatrième chapitre est consacré à la présentation de notre contribution dans laquelle nous cernons précisément notre cas d'étude qui est le processeur MIPS, et nous présentons un cycle générique pour le prototypage et d'exploration. Notre contribution est validée par l'exécution d'un benchmark standard où on a recueilli des résultats sur le nombre des instructions exécutées et le temps d'exécution par instruction. Notre objectif et de mettre cet outil entre les mains des concepteurs cet outil, alors nous n'avons pas intéressé par l'analyse des résultats.

Notre mémoire est clôturé par une conclusion où nous avons présenté les étapes futures pour accomplir ce travail.

# Chapitre 1

# Introduction aux ADL

## 1.1 Introduction

Les méthodologies de conception des microprocesseurs sont relativement complexes. Afin de remédier à ce problème, les développeurs de logiciels ont introduit la notion d'outils de prototypage rapide. Ceci consiste à mettre en place un ensemble langages permettant la conception des microprocesseurs dans un haut niveau d'abstraction et donne la possibilité à des analyses systématiques et la génération automatique des outils et prototype de simulation. La complexité reste au niveau des outils nécessaires pour l'exploitation optimum des nouvelles architectures existantes sur le marché. En effet, plusieurs méthodologies de conception ont été proposées pour le développement d'un flot de conception complet partant d'une spécification au niveau système de l'application, allant jusqu'au prototypage sur plate-forme matérielle. Dans ce chapitre, nous allons s'intéresser par le concept des langages de description des architectures des processeurs qui est considère comme le pierre angulaire pour toutes les techniques de prototypages des processeurs. A ce propos, nous faisons un survol rapides sur les principaux ADL qui existent en essayons de faire une classification dont la finalité est d'arriver à dégager le meilleur en fonction des orientations de notre projet.

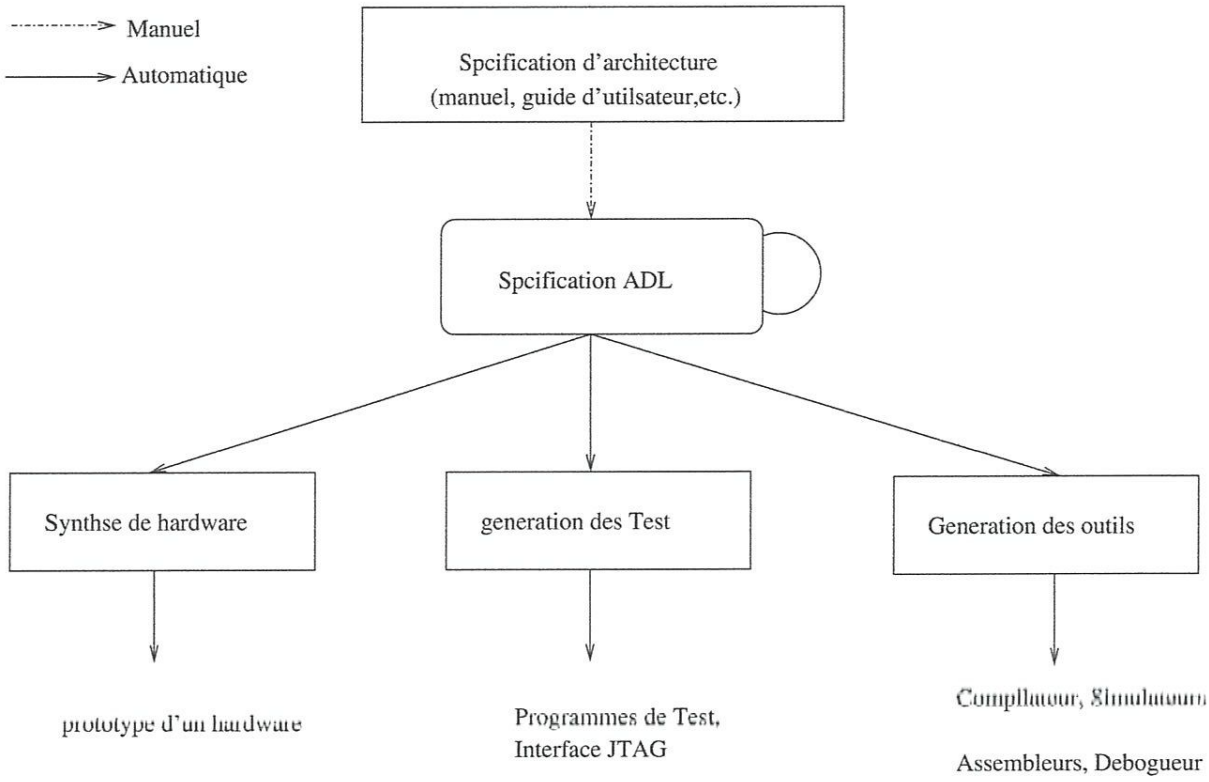



FIGURE *1.1 – Conception Orienté ADL des processeurs [15]*

## 1.2 État de l'art sur les langages de description d'architecture

Les ADL sont classés suivant leurs natures d'information en trois catégories : structurel, comportemental et mixte. Les ADL structurelles capturent la structure en termes de composantes architecturales et leurs connectivités. Les ADL comportementales capturent le comportement de jeu d'instructions de l'architecture du processeur. Les ADL mixtes capturent à la fois la structure et le comportement de l'architecture [20]. Dans ce qui suit de ce chapitre, nous présentons un état de l'art sur les ADL existants et essayons de faire une synthèse afin de dégager le meilleur ADL pour l'adopter dans les chapitres suivants.

## 1.3 Classification orientée contenu

### 1.3.1 ADL structurels

Il y a deux aspects importants à considérer pour concevoir un ADL :

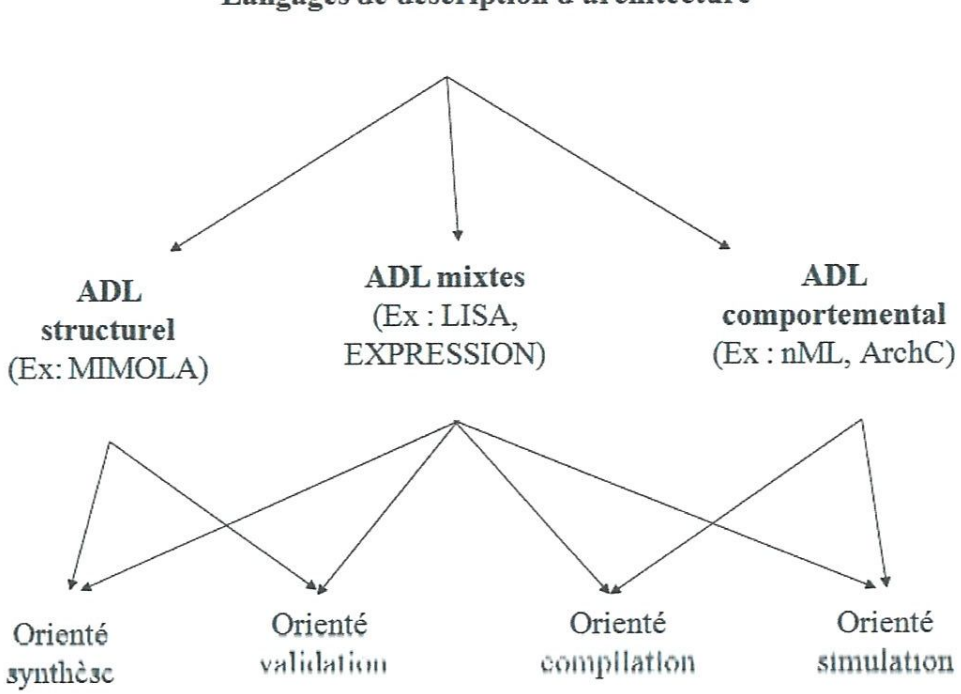



FIGURE *1.2 – Classification des ADL*

1. **Avec un niveau d'abstraction élevé** Il est très difficile de saisir les caractéristiques des différents types de processeurs avec un niveau de détail de petite granularité. Une façon courante d'obtenir une spécification d'un processeur est de décrire ses composantes comme des boites noires avec des input/output munies d'un comportement.
2. **Niveau de transfert de registre (RT-niveau)** est un niveau d'abstraction très utilisé par les électroniciens. Il s'agit de spécifier un processeur dans un niveau très bas (niveau registre, niveau portes logique). Cette manière de description est assez proches de la réalité mais très coûteuse en termes du temps et d'effort.

Nous présentons brièvement dans ce qui suit un ADL structurel : MIMOLA

### MIMOLA

MIMOLA [14] est un ADL structurel développé à l'université de Dortmund, en Allemagne. Il a été proposé à l'origine pour la conception des micro-architectures. L'un des avantages majeurs de MIMOLA est que la même description peut être utilisée pour la synthèse, la simulation, la génération de tests et la compilation. Une chaîne d'outils,

```
LOCATION_FOR_PROGRAMCOUNTER PCReg ;
LOCATION_FOR_INSTRUCTIONS IM[0..1023] ;
```

FIGURE *1.3 – Code pour identifier le compteur de programme et les emplacements des instructions dans la mémoire*

```
Res := MAC(x,  y,  z) ;
```

FIGURE *1.4 – Opération de multiplication effectuée par MAC*

y compris le synthétiseur matériel MSSH, le générateur de code MSSQ, le compilateur MSST, le simulateur fonctionnelle MSSB et le simulateur MSSU du niveau RTL ont été développés sur la base du langage MIMOLA. MIMOLA a également été utilisé par le compilateur RECORD. La description du MIMOLA contient trois parties :

- l'algorithme à compiler
- le modèle du processeur cible
- liens supplémentaires et des règles de transformation

La partie logicielle (description d'algorithme) décrit les programmes d'applications dans une syntaxe de type PASCAL-like. Le modèle de processeur décrit la micro-architecture sous la forme d'un réseau (netlist) des composants. Les informations de liaison sont utilisées par le compilateur pour placer les modules importants tel que les compteurs de programme et les mémoires d'instructions. Le segment de code 1.3 spécifie le compteur de programme et l'emplacement de mémoire d'instruction

La partie algorithmique du MIMOLA est une extension de PASCAL. Contrairement à d'autres langages de haut niveau, il donne des références aux registres physiques et aux mémoires.Il permet également l'utilisation de composants matériels à l'aide d'appels de procédure. Par exemple, si la description du processeur contient un composant appelé MAC, les programmeurs peuvent écrire le segment de code 1.4 pour utiliser l'opération de multiplication-accumulation réalisée par MAC. Le processeur est modélisé comme une netlist de modules qui le composent. MIMOLA permet la modélisation des structures matérielles arbitraire (programmables ou non programmables).Un certain nombre d'opérateurs primitifs prédéfinis existe. Les entités de base de modèles de matériel MIMOLA sont les modules et les connexions. Chaque module est défini par son interface de port et de son comportement. L'exemple de la figure1.5 montre la description d'un module multifonction ALU. La clause CONBEGIN / CONEND comprend un ensemble

```
MODULE ALU
         (IN inp1, inp2: (31:0);
         OUT outp: (31:0);
         IN ctrl;
         )
CONBEGIN
        outp <- CASE ctrl OF
        0: inp1 + inp2 ;
        1: inp1 - inp2 ;
        END;
CONEND;
```

FIGURE *1.5 – la description d'un module de calcul ALU multi-fonctionnel*

```
CONNECTIONS ALU.outp -> ACC.inp
            ACC.outp -> ALU.inp
```

FIGURE *1.6 – Description de MIMOLA pour relier deux module : ALU et l'accumulateur ACC*

de tâches concurrentes. Dans un exemple une affectation conditionnelle au port de sortie *outp* est précisé, ce qui dépend du bit d'entrée *CTRL*. Le netlist est formé en connectant les ports aux instances des modules. Par exemple, la description 1.6 relie deux modules : ALU et l'accumulateur ACC.

### 1.3.2 ADL comportementales

La difficulté d'extraction de jeu d'instructions peut être évitée en faisant abstraction des informations comportementales des détails structurels. Les ADL comportementales spécifient explicitement la sémantique d'instruction et ignore les détails des structures du matériel. Typiquement, il y a une correspondance un-à-un entre les ADL comportementales et le manuel de référence de jeu d'instruction. Cette section décrit brièvement deux ADL comportementales : nML et ArchC.

#### nML

nML [9]est un ADL qui permet la définition des jeux d'instructions. c'est le fruit d'un travail proposé à l'Université technique de Berlin, en Allemagne. nML a été utilisé par les générateurs de code SRC et CHESS, et simulateurs de jeu d'instructions Sigh/Sim et d'autres vérificateurs de systèmes (System Checkers). nML exploite la ressemblance des instructions pour réduire la description des jeu d'instructions. Les instructions sont orga-

```
op numeric_instruction(a:num_action, src:SRC, dst:DST)
   action {
   temp_src = src;
   temp_dst = dst;
   a.action;
   dst = temp_dst;
    }
   op num_action = add | sub
   op add()
   action = {
   temp_dst = temp_dst + temp_src
}
```

FIGURE *1.7 – Spécification d'instruction définie par nML*

nisée sous forme arborescente. Dans le sommet une instruction principale, et les éléments subordonnés sont des instructions intermédiaires(PI). La relation entre les éléments peut être établie en utilisant deux règles de composition : *AND-rule* ou *OR-rule*. La *AND-rule* regroupe plusieurs PI dans une plus grande PI et la *OU-rule* énumère un ensemble d'alternatives pour un PI. Donc la définition d'instruction en nML peut se présenter sous la forme d'un arbre *and/or*. Chaque dérivation possible de l'arbre correspond à une instruction du jeu d'instructions. Pour atteindre l'objectif de partager les descriptions d'instructions, le jeu d'instructions est décorée par une grammaire attribuée. Chaque élément dans la hiérarchie a quelques attributs. Les valeurs des attributs d'un élément non-feuille peuvent être calculées à partir de valeurs d'attributs de ses enfants. La description dans la segment 1.7 nML montre un exemple de spécification de l'instruction *numeric_instruction*. La définition de l'instruction numérique *numeric-instruction* combine trois instructions partielles (PI) avec le *AND-rule* : l'action *num-action*, *SRC* et *DST*. La première PI, *num_action*, utilise *OR-rule* pour décrire les options valables des actions qui sont *add* ou *sub*. Le comportement commun de toutes ces options est défini dans l'attribut *num-action* de *numeric_instruction*. Chaque option de *num_action* doit avoir un attribut action propre à lui qui défini son comportement spécifique, qui est appelé par la ligne *a.action*. nML capte aussi l'information de structure utilisée par l'architecture de jeu d'instructions (ISA). Par exemple, les unités de stockage doivent être déclarées pour être visibles dans le jeu d'instruction. nML prend en charge trois types de stockages : RAM, registre, et le stockage transitoire. Le stockage transitoire se réfère aux états de la machine qui sont conservés que pour un nombre limité de cycles, par exemple, les valeurs sur les bus. Les

calculs n'ont pas un délais dans le modèle de synchronisation nML, seules les unités de stockage ont du retard.

#### ArchC

ArchC [21] est une ADL open-source qui a été conçu au Laboratoire de systèmes informatiques (LSC) de l'Institut d'informatique de l'Université de Campinas (University of Campinas IC-UNICAMP). ArchC est un langage simple, capable de décrire une architecture de processeur, ainsi qu'une hiérarchie de mémoire, qui suit le style de syntaxe SystemC. Son principal objectif est de fournir suffisamment d'information, au bon niveau d'abstraction, afin de permettre aux utilisateurs d'explorer et de vérifier une nouvelle architecture en générant automatiquement des outils logiciels tels que l'assembleur, les simulateurs, les linkers et les débogueurs. Une description de l'architecture dans ArchC est divisée en deux parties : la description des instructions (*ISA_AC*) et la description des ressources de l'architecture (*AC_ARCH*). Dans la description *AC_ISA*, le concepteur fournit des détails sur les formats d'instruction, la taille et les noms associés à toutes les informations nécessaires au décodage et le comportement de chaque instruction. La description *AC_ARCH* informe ArchC sur les périphériques de stockage, la structure pipeline, etc. Sur la base de ces deux descriptions, ArchC peut générer des simulateurs (en utilisant SystemC), et un ensemble d'outils binaires associé.

### 1.3.3 ADL mixtes

Les ADL mixtes couvrent à la fois les détails structurels et comportementaux de l'architecture. Cette section décrit brièvement un ADL mixte : LISA.

#### LISA

LISA [25](Language for Instruction Set Architecture) a été développé dans l'université des Technologies à Aachen, en Allemagne il s'agit d'un langage orienté vers la production des simulateur avec une qualité supérieur. Un aspect important du langage LISA est sa capacité à capturer le chemin de contrôle (Control Path ) explicitement. Ce qui est important conjointement avec la modélisation des Data Path pour une simulation des architectures avec des pipelines. LISA a également été utilisé pour générer des back-end des compilateurs C. Les descriptions de LISA sont composées de deux types de déclarations :

```
PIPELINE int = {Fetch; Decode; IALU; MEM; WriteBack}
PIPELINE flt = {Fetch; Decode; FADD1; FADD2;
FADD3; FADD4; MEM; WriteBack}
PIPELINE mul = {Fetch; Decode; MUL1; MUL2; MUL3; MUL4;
MUL5; MUL6; MUL7; MEM; WriteBack}
PIPELINE div = {Fetch; Decode; DIV; MEM; WriteBack}
```

FIGURE *1.8 – extrait d'une description d'une pipeline en LISA*

```
OPERATION i_type IN pipe_int.Decode {
DECLARE {
GROUP opcode={ADDI || SUBI}
GROUP rs1, rd = {fix_register};
}
CODING {opcode rs1 rd immediate}
SYNTAX {opcode rd ``," rs1 ``," immediate}
BEHAVIOR { reg_a = rs1; imm = immediate; cond = 0;
}
ACTIVATION {opcode, writeback}
}
```

FIGURE *1.9 – le comportement lors du décodage de deux opérations de type immédiat (i_type) ADDI et SUBI dans le stage DLX du pipeline*

ressources et de l'exploitation. Les déclarations de ressources portent sur les ressources matérielles telles que les registres, les pipelines, et des mémoires. Le modèle de pipeline définit tous les chemins possibles par lesquels les différentes opérations peuvent en passer. Un exemple de la description pipeline est donnée dans la figure 1.8.

Les opérations sont les objets de base de LISA. Ils représentent le comportement, la structure et le jeu d'instructions de l'architecture modélisée. La définition d'une opération capture la description des différentes propriétés comme le comportement de l'opération, les informations du jeu d'instructions, et la synchronisation. Ces propriétés sont définis dans plusieurs sections. LISA exploite les points communs des opérations similaires en les regroupant en un seul. Le segment de code 1.9 décrit le comportement lors du décodage de deux opérations de type immédiat (*i_type*) *ADDI* et *SUBI* dans le stage *DLX* du pipeline. Le comportement global d'une opération peut être obtenu en combinant ses définitions de comportement dans toutes les étapes du pipeline.

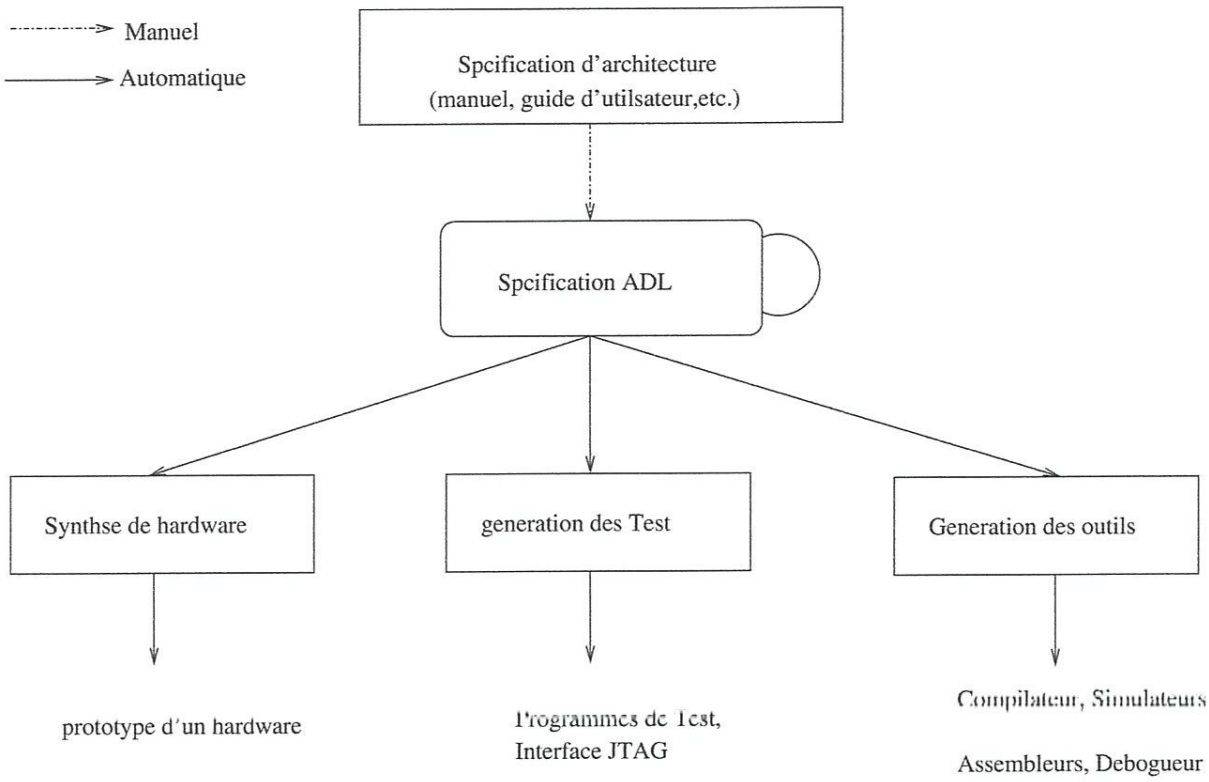



FIGURE *1.10 - Exploration d'espace d'architecture basée ADL*

## 1.4 Classification orientée objectif

Les ADL ont été utilisés avec succès comme des langages de spécification pour le développement des processeurs. Avec le temps, l'évaluation rapide des architectures est devenue nécessaire pour l'exploration des espaces vastes des alternatives possibles des conceptions et de trouver la meilleure conception possible sous les déférentes contraintes comme la puissance, la performance, la chaleur dissipée, etc. La figure 1.10 montre un flux d'exploration d'architecture basée ADL .Les programmes sont compilés et simulés, et les évaluations sont utilisées pour modifier la spécification ADL. Le simulateur généré produit des données qui peuvent etre utilisés pour évaluer des jeu d'instructions, la performance d'un algorithme, et la taille requise en mémoire et nombre des registres. Le modèle de matériel généré peut fournir une idée plus précise concernant l'espace silicium occupé, la fréquence d'horloge ou la consommation d'énergie.

Les ADL peuvent être classés en quatre catégories en fonction de l'objectif :

- Orientée compilation
- Orientée simulation

- Orientée synthèse
- Orientée validation

Dans cette section, nous décrivons brièvement les ADL sur la base de la classification basée sur l'objectif.

### 1.4.1 ADL orienté compilation

L'objectif d'un tel ADL est de permettre la génération automatique des compilateurs croisés. Un compilateur est classé comme croisé s'il peut être adapté pour générer du code pour des processeurs cibles différents. Cet objectif est atteint en fournissant les informations données en entrée aux ADL au compilateur avec le code source pour qu'il puisse adapter le code binaires généré en fonction de l'architecture cible.

### 1.4.2 ADL orienté simulation

La simulation peut être réalisée à différents niveaux d'abstraction. Au plus haut niveau d'abstraction, la simulation fonctionnelle (simulation du jeu d'instructions ) du processeur peut être réalisée uniquement par la modélisation de l'ensemble d'instructions. Les ADL comportementaux permettent la génération de simulateurs fonctionnels. Les modèles de simulation du cycle-précis des pipeline fournissent des informations temporelle plus détaillé car ils sont à un niveau plus bas d'abstraction. les ADL structurels sont de bons candidats pour la génération de simulateur cycle précis.

### 1.4.3 ADL orienté synthèse

Des ADL structurels comme MMIMOLA sont adaptés pour la génération de matériel. Certaines langages comportementales telles nML sont également utilisés pour la production de matériel. Par exemple, le générateur de matériel GO se base sur nML.

### 1.4.4 ADL orienté validation

Les ADL ont été utilisés avec succès dans le milieu universitaire ainsi que l'industrie pour permettre la génération des tests pour la validation fonctionnelle des processeurs embarqués.

## 1.5 Conclusion

Après une étude succincte des ADL, nous avons arrivé a choisir l'ADL ArchC pour l'utiliser dans la suite de notre travail. Notre choix est motivé par plusieurs arguments dont les principaux sont :

- La séparation explicite du jeu d'instruction du la description des ressources.
- La disponibilité de l'outil et de la documentation.
- La génération des simulateurs en SystemC qui est un langage très populaire dans le domaine des conception des microprocesseurs.

# Chapitre 2

# Des Langages Open-Source

## 2.1 Introduction

Après une discussion d'état de l'art concernant les ADL. Nous avons décidé d'opter pour l'ADL ArchC vu ses caractéristiques qui sont adaptés à notre travail . Au cours de ce chapitre, nous allons présenter en détail ce langage ainsi que le langage dont lequel il fourni ses résultats en l'occurrence SystemC.

## 2.2 Une Introduction à ArchC

ArchC [21]est ADL open-source qui a été conçu au Laboratoire de systèmes informatiques (LSC) de l'Institut d'informatique de l'Université de Campinas (IC-Unicamp) Au Brésil. ArchC est basé sur SystemC[12],qui reçoit un fort soutien de nombreuses entreprises dans le secteur de l'informatique. L'objectif d'ArchC est de fournir aux concepteurs d'architecture un outil qui peut leur permettre d'évaluer rapidement de nouvelles idées dans des domaines tels que : le processeur et la conception des jeux d'instruction, la hiérarchie de la mémoire et d'autres aspects de la recherche en architecture d'ordinateur. Une Spécification en Archc est divisé en deux parties essentielles :

1. Les éléments de l'architecture *AC_ARCH* : déclare périphériques de stockage, la structure du pipeline, etc
2. Le jeu d'instruction *AC_ISA* : e concepteur fournit des détails sur les instructions.

ArchC prend en entrée une description de processeur et permet la génération automatique de :

1. Un simulateur de jeu d'instruction en SystemC (ISS : Instruction Set Simulator)
2. Un ensemble d'outils binaire (assembleur, éditeur de lien, des-assembleur, débogueur)
3. Un back-end d'un compilateur C++

L'ISS offres des capacités d'interfaçage TLM [1] et la modélisation d'interruption. Plusieurs instances du simulateur peuvent coexister dans la même plate-forme, permettant ainsi la construction facile de multiprocesseurs et des systèmes complexes et hétérogènes.

### 2.2.1 Description des ressources

ArchC utilise les informations structurelles concernant les ressources disponibles au sein de d'une architecture en vue de générer automatiquement un simulateur. Le concepteur doit fournir de telles informations dans la description (AC_ARCH), qui est essentiellement composé d'éléments de stockage et les déclarations de pipelines. Le niveau de détail utilisé dans cette description dépendra du niveau d'abstraction que le concepteur veut pour son modèle. Par exemple, on peut simuler le jeu d'instructions de l'architecture MIPS[19], sans tenir compte de son pipeline. Cela rend la description du comportement d'utilisation très simple et exige également quelques informations structurelles. En revanche, pour les modèles avec pipeline, le concepteur doit fournir à ArchC plus de détails sur la structure du processeur, comme indiqué sur la figure2.1. Nous donnons dans la table2.1un résumé sur les mots clés utilisés pour la description de l'architecture.

### 2.2.2 Description du jeu d'instructions

La description *AC_ISA* donne ArchC toutes les informations dont il a besoin pour synthétiser automatiquement un décodeur, ainsi que le comportement de chaque instruction dans l'architecture. Cette description est divisée en deux fichiers, l'un contenant l'instruction et les déclarations de format et un autre contenant le comportement d'instruction. La figure 2.2 montre un exemple de description *AC_ISA* extrait de notre modèle MIPS. Il contient des instructions, des formats, des informations et des déclarations de décodage de syntaxe de montage, illustrant les principales caractéristiques d'une description *ISA* dans ArchC. Chaque instruction doit avoir un format préalablement déclarée qui lui est associé. Le concepteur déclare une instruction par le mot-clé *ac_instr* et il peut

1. Transaction Level Modeling : un niveau de conception en SystemC qui permet la définition des modèles sous forme des composantes qui communiquent à travers des interfaces

| Mot-clé | Description |
|---|---|
| AC_ARCH | Une description des ressource d'une l'architecture commence toujours par ce mot-clé. Le concepteur doit fournir le nom du modèle (par exemple MIPS). |
| ac_wordsize | Déclare la taille mot supportée par processeur en nombre de bits. |
| ac_regbank | Déclare le banc des registres et de sa capacité (par exemple, le banc de registre RB dispose de 34 registres). |
| ac_reg | Déclare un seul registre (par exemple IF_ID). |
| ac_mem | Déclare une mémoire d'une taille donnée (par exemple, MEM a 256K). La taille peut être exprimée en octets (sans abréviation de l'unité nécessaire), en kilo-octets (K ou k), en mégaoctets (M ou m) ou en gigaoctets (G ou g). |
| ac_tlm_port | Déclare un port de communication externe TLM. Il est suivi par le nom du port et sa taille. Cette taille, tout comme dans ac_mem, peut être exprimée en octets ou ses multiples. |
| ac_tlm_intr_port | Déclare un port de communication TLM interruption, suivie par le nom du port. |
| ARCH_CTOR | Initialise la déclaration du constructeur *AC_ARCH*. |
| ac_isa | Une indication du nom du fichier contenant la description du jeu d'instruction. |
| set_endian | Définit la méthode d'utilisation de l'architecture mémoire en tant que «*big*» ou «*little*». |

TABLE *2.1 – Résumé sur les mots clés utilisés pour la description de l'architecture*

```
AC_ARCH(mips){
  ac_wordsize 32;
  ac_mem MEM:256K;
  ac_regbank RB:34;
  ac_pipe pipe = {IF, ID, EX, MEM, WB};
  ac_format Fmt_IF_ID = "%npc:32";
  ac_format Fmt_ID_EX ="%npc:32 %data1:32 %data2:32 %imm:32:s rs:5 %rt:5
  %rd:5%regwrite:1%memread:1 %memwrite:1" ;
  ac_format Fmt_EX_MEM ="%alures:32 %wdata:32 %rdest:5 %regwrite:1
  %memread:1 %memwrite:1";
  ac_format Fmt_MEM_WB = "%wbdata:32 %rdest:5 %regwrite:1";
  ac_reg<Fmt_IF_ID> IF_ID;
  ac_reg<Fmt_ID_EX> ID_EX;
  ac_reg<Fmt_EX_MEM> EX_MEM;
  ac_reg<Fmt_MEM_WB> MEM_WB;
  ARCH_CTOR(mips) {
    ac_isa("mips_isa.ac");
    set_endian("big");
  };
};
```

FIGURE *2.1 – Description des ressources*

l'attribuer un format sous forme d'un code C++. Dans l'exemple, le format *Type_R* est associée à instruction add. Ceci permet au concepteur d'accéder à chaque champ d'instruction individuellement pour décrire les comportements d'instruction. Les symboles du langage d'assemblage, comme les noms des registres, peuvent être regroupées par la clause *ac_asm_map*. Les syntaxes d'instructions et le codage d'opérandes sont spécifiés par la clause *set_asm*. En outre, les pseudo-instructions peuvent être décrites en fonction des instructions précédemment créés par la construction *pseudo_inst*. Dans l'exemple, nous avons déclaré les noms des registres MIPS (en utilisant l'identifiant *reg*) et les ont utilisés comme un type d'opérande dans la déclaration de la syntaxe de l'instruction add. Nous avons également créé le MIPS pseudo instruction *li* (charge immédiate).

### Spécification ISA

Nous donnons dans la table2.2 un résumé sur les mots clés utilisés pour la description du jeu d'instruction.

```
AC_ISA(mips){

  ac_format Type_R = "%op:6 %rs:5 %rt:5 %rd:5 0x00:5 %func:6";
  ac_format Type_I = "%op:6 %rs:5 %rt:5 %imm:16:s";
  ac_format Type_J = "%op:6 %addr:26";

  ac_instr<Type_R> add, sub, instr_and, instr_or, mult, div;
  ac_instr<Type_R> mfhi, mflo, slt, jr;
  ac_instr<Type_R> addu, subu, multu, divu, sltu;
  ac_instr<Type_R> sll, srl;
  ac_instr<Type_I> lw, sw, beq, bne;
  ac_instr<Type_I> addi, andi, ori, lui, slti;
  ac_instr<Type_I> addiu, sltiu;
  ac_instr<Type_J> j, jal;

  ac_asm_map reg {
    "$"[0..31] = [0..31];
    "$zero" = 0;
    "$at" = 1;
    "$kt"[0..1] = [26..27];
    "$gp" = 28;
    "$sp" = 29;
    "$fp" = 30;
    "$ra" = 31;
  }

  ISA_CTOR(mips){

    lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);
    lw.set_decoder(op=0x23);

    sw.set_asm("sw %reg, %imm(%reg)", rt, imm, rs);
    sw.set_decoder(op=0x2B);

    add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
    add.set_decoder(op=0x00, func=0x20);

    addu.set_asm("addu %reg, %reg, %reg", rd, rs, rt);
    addu.set_decoder(op=0x00, func=0x21);

    ...

    pseudo_instr("li %reg, %imm") {
      "lui %0, \%hi(%1)";
      "ori %0, %0, %1";
    }
    ...
  };
};
```

FIGURE 2.2 – *Description du jeu d'instructions*

| Mot-clé | Description |
|---|---|
| AC_ISA | Une description ISA commence toujours avec ce mot-clé. Le concepteur doit fournir le nom du modèle (par exemple MIPS dans la figure2.2). |
| ISA_CTOR | Il initialise simplement la déclaration du constructeur *AC_ISA*. |
| ac_format | On déclare un format d'instruction et de ses champs (par exemple, sur la figure2.2, la ligne 2, un format *Type_R* est défini comme étant la concaténation de cinq champs : la première, appelée op, se compose de six bits.). |
| $ac\_instr \prec fmt \succ$ | Il déclare une instruction et l'attache à un format prédéfini.Dans la figure2.2, par exemple, l'instruction $ac\_instr \prec Type\_R \succ add$,est liée au format Type_R. |
| set_decoder | On initialise la séquence de décodage d'instruction, qui est un élément essentiel pour la génération automatique de l'assembleur pour le modèle de processeur exécutable. La séquence est composée de paires $\prec field\_name = value \succ$. Par exemple dans la figure2.2 un flux de bits provenant de la mémoire est une instruction *lw* si est seulement si le champ *op* contient la valeur 0x23 |
| ac_asm_map | Indique une correspondance entre les symboles d'assemblage et les valeurs (par exemple définir l'ensemble du registre des noms et leurs numéros correspondants dans le banc des registres de l'architecture du processeur ). |
| pseudo_instr | Décrit des aléas en termes d'instructions décrites précédemment. |
| set_asm | Associe une syntaxe pour l'assemblage et le codage des opérandes à une instruction. Pour chaque opérande, il doit y avoir un champ d'instruction associé, précisant l'encodage opérande (par exemple dans la Fig 2.2, l'instruction add utilise trois opérandes de type de reg qui sont associés, respectivement, avec les champs *rd*, *rs* et *rt* . |

TABLE 2.2 *Résumé sur les mots clés utilisés pour la description du jeu d'instruction*

### Description des comportements des instructions

La figure2.3 montre un exemple de description du comportement ArchC. Il présente une partie du comportement de l'instruction add. Notons que ce code est divisé, au moyen d'une structure de choix multiple swithcase C++, de sorte que ArchC peut déterminer quelles opérations sont exécutées au niveau de chaque étage de pipeline. Une instruction

```
void ac_behavior( add, stage ){

  switch(stage) {

  case IF:
    IF_ID.npc = ac_pc + 4;
    break;

  case ID:
    break;

  case EX:
     EX_MEM.alu_result = ID_EX.rs + ID_EX.rt;
     ...
    break;

  case MEM:
     MEM_WB.alu_result = EX_MEM.alu_result;
     MEM_WB.rd = EX_MEM.rd;
     ...
     break;

  case WB:
      RB.write(MEM_WB.rd, MEM_WB.alu_result);
      break;

  default:
      break;
  }
};
```

FIGURE *2.3 – description des comportements des instructions*

*switch* similaire peut également être utilisé pour décrire les instructions multi-cycles en cas de besoin. ArchC permet que la description de comportement soit exprimés en plusieurs niveaux d'abstraction. Par exemple, dans les premiers stades de la conception, la précision des stages peut ne pas être important. Normalement, le premier modèle d'une nouvelle architecture n'a pas d'informations de synchronisation. Pour ce modèle préliminaire, le comportement d'une instruction donnée serait juste une séquence C++ qui représente les opérations que cette instruction va exécuter dans le matériel. Dans ce dernier cas, le simulateur généré exécuter une instruction par cycle.

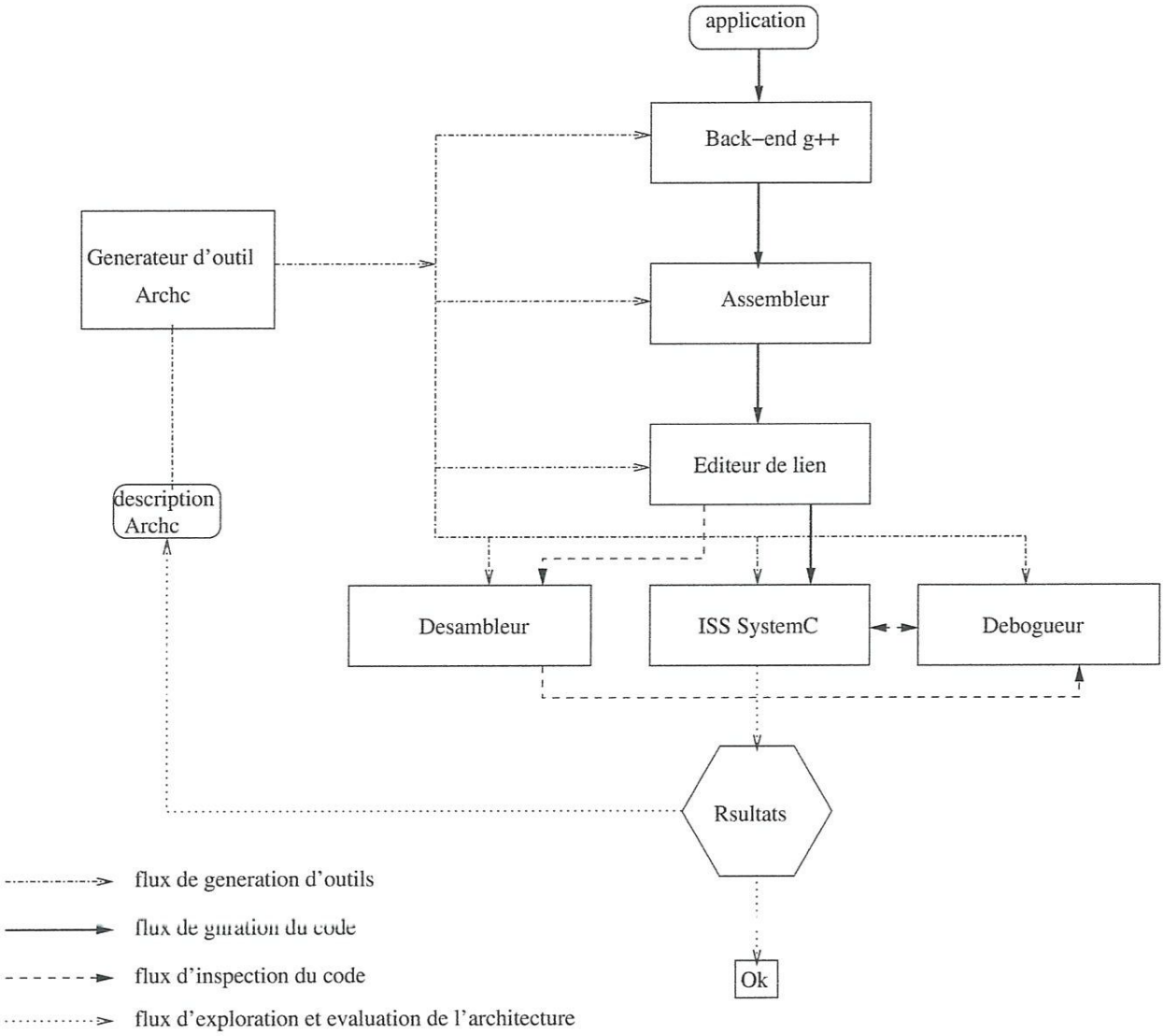



FIGURE 2.4 – *Méthodologie de conception en ArchC[3]*

### 2.2.3 Méthodologie de conception en ArchC

Archc propose une démarche méthodologique pour l'exploitation maximum des ses opportunités. cette démarche se décompose en quatre flux de conceptions[3] qui se manifestent simultanément pour réaliser et explorer des prototypes en vu de l'obtention d'une version optimisée en fonction de types d'applications exécutées. La figure 2.4 présente les flux suivants :

1. Flux de génération d'outils : D'après la description ADL d'un processeur cible (et quelques fichiers auxiliaires) le générateur de l'outil synthétise back-end d'un compilateur, un ISS(Instruction set simulator), et un ensemble d'utilitaires binaires.
2. Flux de génération du code : Le code source de l'application est compilé, assemblée

et les liens édités, résultant en un code exécutable qui fonctionne sur un modèle de processeur (ISS).

3. Flux d'inspection du code : Souvent des étapes d'analyse et d'inspection du code lors de l'exécution sont nécessaires, ArchC prévoit un flux à ce propos en utilisant un dés-assembleur et un débogueur.

4. Flux d'évaluation et d'exploration des architectures : Après inspection de code, ce qui peut nécessiter le désassemblage et le débogage, le résultat de la simulation est évalué face aux exigences et critères de conception. Si une exigence ou un critère n'est pas rempli, un autre architecture ou une modification de l'architecture de départ peut être choisi et le processus est répété jusqu'à ce que toutes les contraintes et les critères soient satisfaits.

## 2.3 SystemC

Pour faire face à la complexité croissante des circuits et des systèmes embarqués. Le niveau d'abstraction de la conception a été élevée au cours des dernières décennies : a partir des transistors au portes logiques, des portes au niveau Registres (RTL), et depuis quelques années à partir de RTL aux modèles basé sur C/C++[11]. Cette évolution a entraîné le développement de langage de description de système SystemC qui a été normalisé par l'IEEE. SystemC [4] a été introduit par l'Initiative Open SystemC (OSCI), une association indépendante, à but non lucratif composé de partenaires industriels et académiques. Dans la partie suivante SystemC est brièvement passé en revue ensuite un exemple simple de modélisation SystemC est fourni.

### 2.3.1 Pourquoi utiliser SystemC ?

1. Orienté objet : réutilisation plus facile des blocs, ajout des structures nécessaires plutôt que définir un nouveau langage

2. Fournit un langage commun simple et une base de modèles pour faire les systèmes et les conceptions IP [2] .

3. SystemC est complètement open-source

2. Intlectuel property : des boites noir qui représentent des composantes électroniques standards

```
include "systemc.h"
SC_MODULE (and3)
 {
   sc_in<bool> e1;
   sc_in<bool> e2;
   sc_in<bool> e3;
   sc_out<bool> s;
   void compute_end()
     {
      S= e1 & e2 & e3 ;
     }
   SC_CTOR (and3)
    {
     SC_METHOD (compute_and) ;
     Sensitive << e1;
     Sensitive << e2;
     Sensitive << e3;
    } ;
} ;
```

FIGURE *2.6 – un port combinatoire AND à trois entrées[24]*

### 2.4.1 Explication de l'exemple

#### Inclusion des définitions SystemC : #include "systemc.h"

Tous les modules utilisant des classes de SystemC doivent inclure cette déclaration. Il est nécessaire de donner au compilateur GCC[10], lors de la compilation, le chemin vers les librairies SystemC. Typiquement, ce genre d'option se positionne dans un fichier Makefile[3] .

#### Déclaration du module : SC_MODULE(and3)

C'est la déclaration du module, effectuée à l'aide d'une macro SystemC. On lui donne en argument le nom du module. Dans la suite du code, on déclare ses variables internes, ainsi que ses méthodes. Même s'il vaut mieux séparer déclaration et implémentation, on a ici tout regroupé dans la déclaration de la classe pour produire un code plus lisible.

3. Est un fichier de configuration pour l'utilitaire Make qui autoatise la compilation du code c/c++ [10]

### Déclaration des ports

Les macros *sc_in* , *sc_out* et *sc_inout* servent à définir des ports qui seront reliés à des signaux(un type particulier et très simple de canal). Ces macros prennent en argument le type du signal (booléen, vecteur, logique multivaluée, etc...). On définit donc ici quatre ports, 3 en entrée, 1 en sortie, tous de type booléen (bit).

### Déclaration des processus

Un processus est une fonction de type *void f()* Ici on déclare la fonction *compute_and* qui réalise le ET logique des 3 ports en entrée. Il faut maintenant déclarer au moteur de simulation :

- que cette méthode est en fait un processus
- et quelle est sa liste de sensibilité.

C'est l'objectif de la prochaine instruction.

### Déclaration du constructeur et des listes de sensibilité des processus

Chaque classe C++ doit avoir un constructeur, dont le but est de faire toutes les initialisation de l'objet en question. On le déclare ici à l'aide de la macro *SC_CTOR* qui prend en argument le nom du module en question. Ici, la seule initialisation consiste à déclarer que la fonction *compute_and* n'est pas seulement une fonction pratique, mais un processus. C'est fait par la macro *SC_METHOD* qui est l'un des trois types de processus possible.

### L'instruction sensitive

Sert à préciser la liste de sensibilité du dernier processus déclaré. Une fonction AND étant combinatoire, on déclare donc le processus compute_and comme étant sensible aux trois signaux d'entrée, sur chacun de leur front.

## 2.5 conclusion

Durant ce chapitre, une étude approfondie sur deux langages essentiels dans notre projet est menée. Ceci nous permettra avec ce qu'on va voir dans le chapitre suivant de présenter notre contribution qui s'agit d'une enrichissement de la cycle d'exploration des architectures par une approche MDE.

# Chapitre 3

# Une approche MDE

## 3.1 Introduction

Pour le développement des systèmes MPSoC d'une manière générale et particulièrement les processeurs, l'ingénierie dirigée par les modèles(MDE) offre une approche prometteuse pour le traitement de leurs complexités. Dans ce chapitre on va parler d'une manière détaillée sur cette approche (MDE) et son apport significatif sur la facilitation de conception et exploration des architectures des processeurs.

## 3.2 MDE : Model Driven Engineering

MDE[18] est un domaine de l'ingénierie du système dans lequel les processus reposent en grande partie sur l'utilisation de modèles. C'est un paradigme de génie logiciel qui suggère d'utiliser des modèles comme les objets principaux du développement de logiciels. Il repose sur deux principes de base : abstraction et d'automatisation3.1.

1. **L'Abstraction** suggère l'utilisation d'un langage de méta-modélisation pour développer des formalismes spécifiques pour le domaine étudié.
2. **L'Automatisation** Il suggère la transformation des modèles entre plusieurs formalismes y inclut les formalismes textuels.

### 3.2.1 Modèles et Meta-modèles

L'idée de MDE, et plus particulièrement, née de la recherche récente du développement de logiciels. MDE évolué comme un changement de paradigme de la technologie orientée

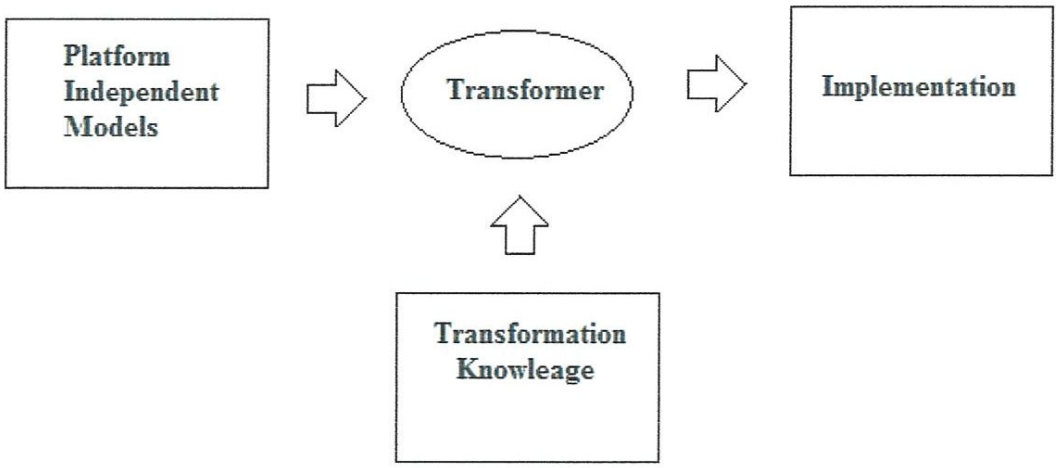



FIGURE *3.1 – MDE : abstraction et automatisation*

objet, dans lequel le principe de base « tout est un modèle ». MDE se base sur les modèles, mais il est également basé sur les relations entre un modèle et le système à l'étude.

### Modèle

Les modèles jouent un rôle majeur dans la MDE. La définition la plus générale dit qu'un modèle est une représentation simplifiée de la réalité, ou, plus formellement, un modèle est un ensemble de déclarations concernant un système données[5]. En fait, on peut dire que le modèle est un ensemble clair d'éléments formels qui décrit quelque chose en cours de développement dans un but précis. En plus de ce qui est spécifié par la définition d'un modèle, un modèle d'ingénierie doit posséder, à un degré suffisant, les cinq caractéristiques clés suivantes :

1. **Abstraction** : Un modèle est toujours un rendu réduit du système qu'il représente.
2. **Compréhensibilité** : Il ne suffit pas de faire abstraction des détails, il faut aussi présenter ce qui reste sous une forme plus directe (par exemple, une notation)pour facilité la compréhension du modèle.
3. **Précision** : Un modèle doit fournir une représentation fidèle aux fonctionnalités du système modélisé.
4. **Prévisibilité** : Nous devrions être en mesure d'utiliser un modèle de prévision correcte des propriétés intéressantes mais non évidente du système modélisé, soit

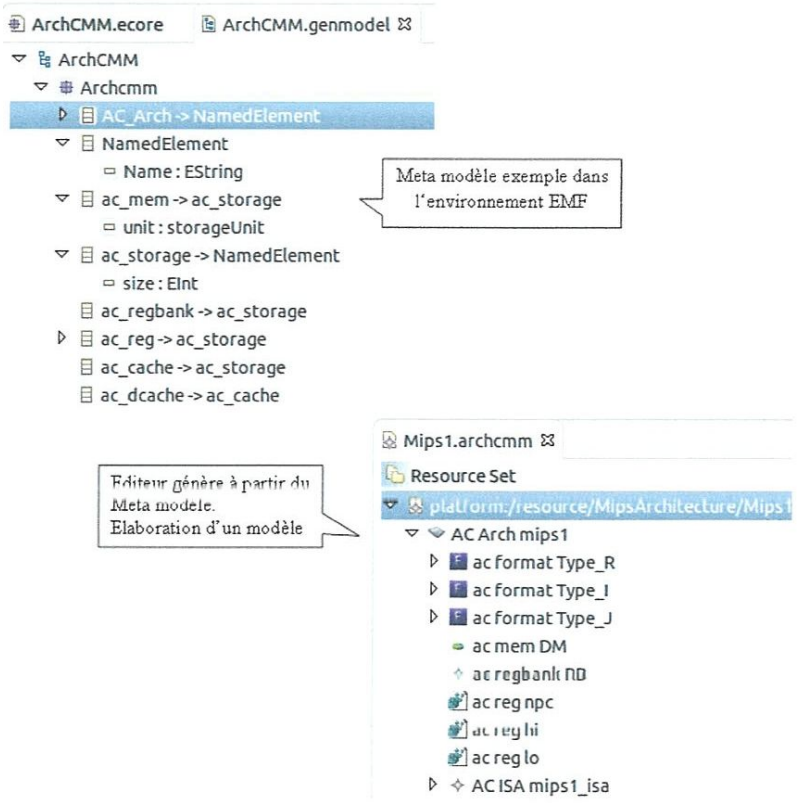



FIGURE *3.4 – Exemple d'utilisation d'un éditeur graphique généré par EMF*

Le but de ce framework est de faciliter la manipulation des modèles afin de permettre leur intégration dans la plate-forme Eclipse.

C'est dans cet objectif que plusieurs fonctionnalités ont été développées pour permettre le développement et la manipulation facile de nouveaux métas modèles dans la plate-forme d'Eclipse. Parmi ces fonctionnalités, celle qu'est certainement la plus agréable est la génération automatique (à partir du méta modèle) d'un simple éditeur graphique permettant l'édition des modèles sous forme arborescente. Chacun des nœuds de l'éditeur représentera une instance d'une méta classe.

Cette fonctionnalité s'utilise très facilement dans le framework EMF. Il suffit de demander la génération des classes Java composant l'éditeur graphique correspondant à un méta-modèle données puis d'exécuter ces classes dans la plate-forme Eclipse afin de visualiser l'éditeur graphique. Nous avons utilisé cette fonctionnalité sur un méta-modèle d'exemple et nous avons pu élaborer notre modèle grâce à cet éditeur graphique, comme l'illustre la figure **??**

appelée génération. JET dispose de trois types différents d'expressions : des directives, des expressions et scriptlets.

- **Les Scriptlets** : peuvent contenir du code java.
- **Les Expressions** : permettent d'insérer des valeurs de chaîne à l'intérieur de la sortie du JET .
- **Les directives** : définissent les paramètres du modèle de JET.

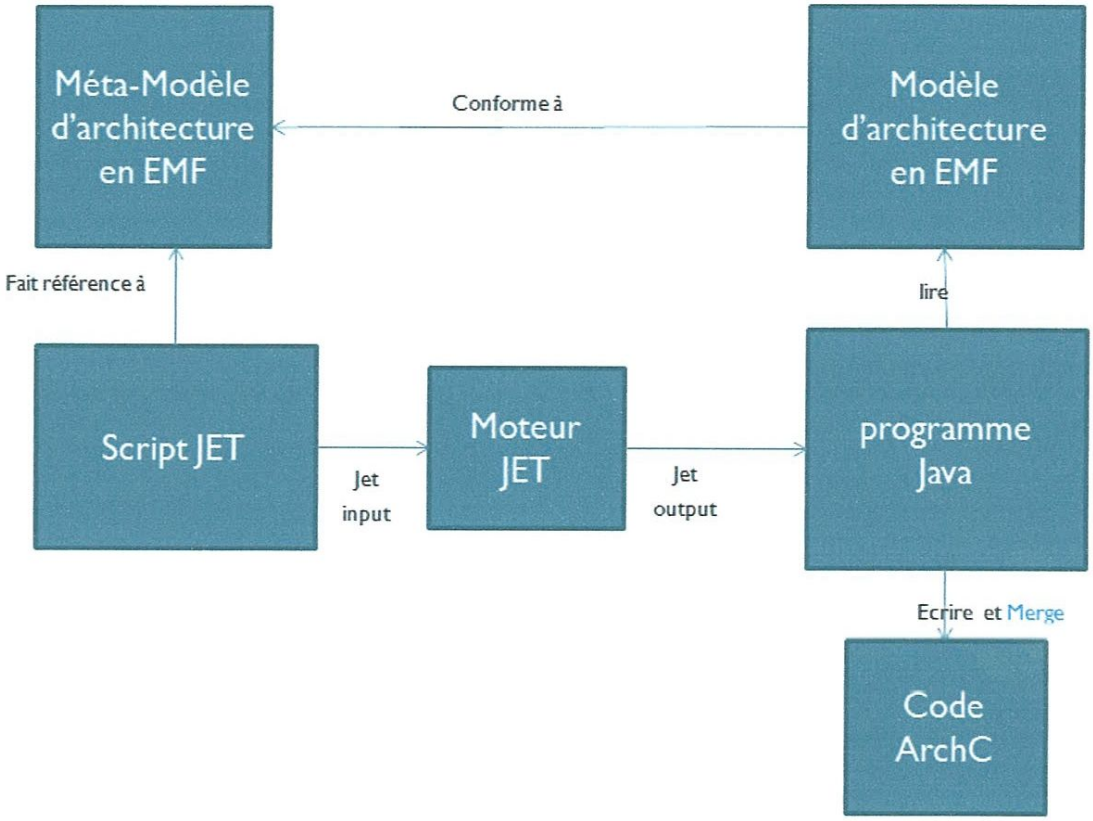
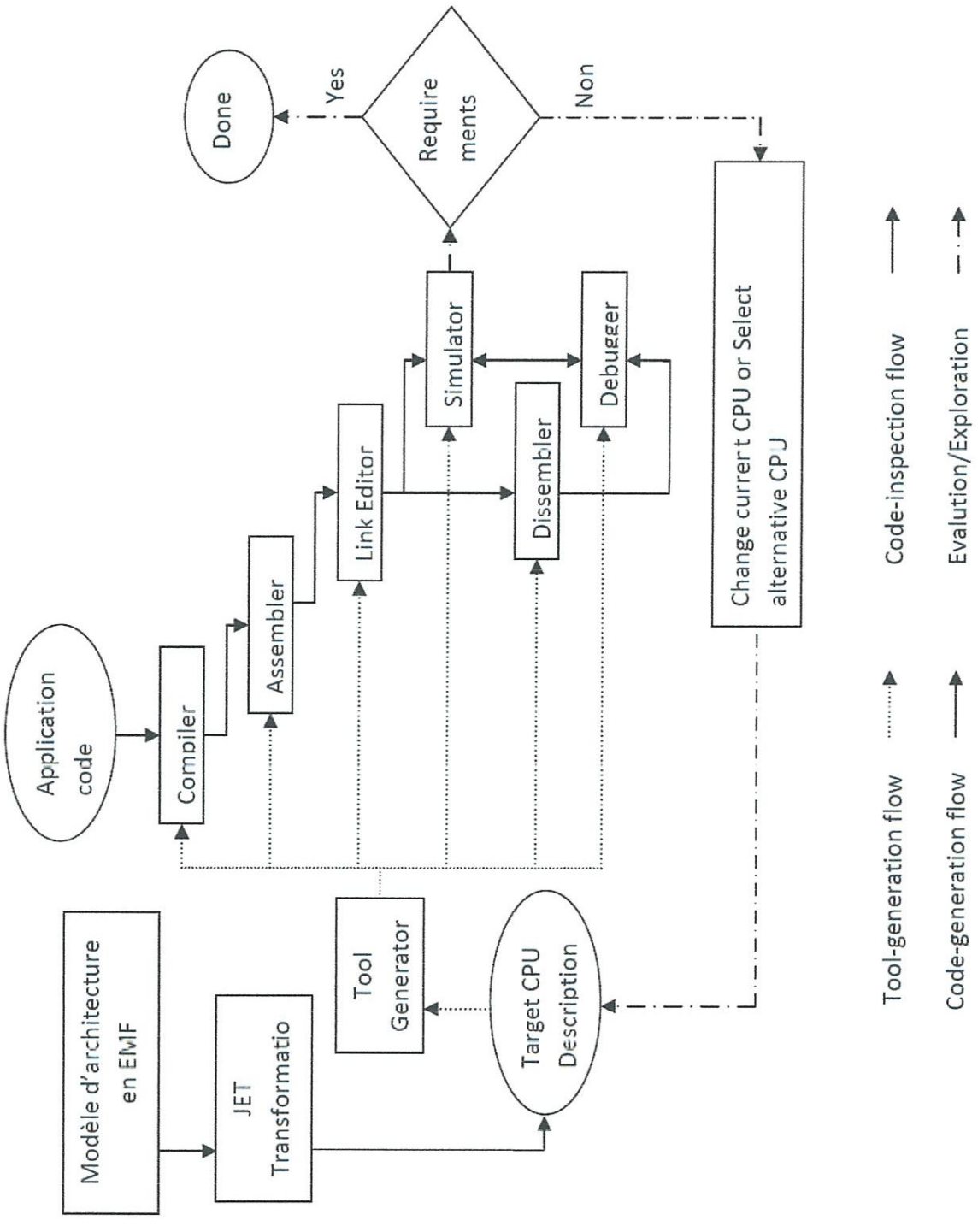



FIGURE *3.5 – Architecture de JET (java Emitter Template )*

## 3.6 Mise enœuvre

Pour la réalisation de notre projet nous avons utilisés les deux outils vu dans les paragraphes précédents (EMF, JET). Qui nous ont aidées pour la génération des fichiers nécessaires pour le prototypage du microprocesseur MIPS. Premièrement, nous avons crée un méta modèle avec EMF pour générer un éditeur graphique permettant l'édition des instances d'architectures de processeurs sous forme arborescente. Notre outil permet en plus de l'?édition des modèles plusieurs autres options comme par exemple le sauvegarde

sous forme XML du modelés, la vérification des contraintes d ?Intégrité sur les instances des architectures, la traçabilité[17]. Ces modèles serons ensuite transformé a avec JET vers des fichiers de code en ArchC (ex. mips.ac)qui vont être l ?entrée vers le flux de génération d'outils de langage ArchC. Ce cycle est représenté dans la figure 3.6


Modèle d'architecture en EMF
JET Transformatio
Tool Generator
Target CPU Description
Application code
Compiler
Assembler
Link Editor
Simulator
Dissembler
Debugger
Require ments
Yes
Done
Non
Change currert CPU or Select alternative CPU
Tool-generation flow
Code-generation flow
Code-inspection flow
Evalution/Exploration

## 3.7 Conclusion

Grâce à MDE et ses outils sous Eclipse (EMF, JET) la procédure de conception d'une nouvelle architecture de processeur est significativement devenue plus facile et efficace.

Notre objectif a travers l'utilisation de cette approches (MDE) est d'arriver à automatiser le flux de conception d'architecture et permettre l'exploration rapide en ce basons sur des résultats fournit par des flux de conception classique définie par des ADL tel que ArchC.

Dans le chapitre suivant, nous allons validé notre approche par le prototypage d'un cas d ?étude qu'est le processeur MIPS.

# Chapitre 4

# Prototypage du microprocessuer MIPS

## 4.1 INTRODUCTION

Dans ce chapitre, nous présentons une version légèrement simplifiée de l'architecture de du processeur MIPS. Notre objectif et de présenter l'architecture et le jeu d'instruction du ce processeur afin de l'utiliser comme cas d'étude pour le prototypage avec l'approche et l'outil que nous avons développés. Notre description représente ce que doit connaître un programmeur souhaitant programmer en assembleur, ou la personne souhaitant écrire un compilateur pour ce processeur :

- Les registres visibles.
- L'adressage de la mémoire.
- Le jeu d'instructions.
- Les mécanismes de traitement des interruptions et exceptions.

## 4.2 Architecture externe

Le processeur possède deux modes, le mode utilisateur (ou user) pour exécuter les applications, et le mode noyau (ou kernel) pour exécuter le système. Ces 2 modes sont nécessaires à l'exécution sur de plusieurs processus sur un même processeur[6].

### 4.2.1 Registres visibles

Les registres visibles du MIPS, c-à-d. qui sont manipulés par les instructions implicitement ou explicitement, ont tous une taille de 32 bits. Le MIPS visant par construction l'exécution de multiples processus, des mécanismes de protections sont mis en uvre pour l'accès aux registres relatifs au système. Ces derniers registres appartiennent à un coprocesseur système dit coprocesseur 0 ou *cop0*. Les accès à ce coprocesseur ne peuvent avoir lieu qu'en mode noyau.

#### Registres non protégés

MIPS possède 35 registres manipulés par les instructions standards :

- $R_i(0 \leq i \leq 31)$ 32 registres généraux : Ces registres sont directement adressés par les instructions, et permettent de stocker des résultats de calcule intermédiaires. Le registre $R_0$ est un registre particulier :
  - la lecture fournit la valeur constante "0x00000000"
  - l'écriture ne modifie pas son contenu.

  Le registre $R_{31}$ est utilisé par les instructions d'appel de procédures (instructions BGEZAL, BLTZAL, JAL et JALR) pour sauvegarder l'adresse de retour.
- Le Registre $PC$ : C'est le compteur de programme (Program Counter),il contient l'adresse de l'instruction en cours d'exécution. Sa valeur est modifiée suite à l'exécution des instructions.
- Les Registres $HI$ et $LO$ sont utilisés pour la multiplication ou la division. Ces deux registres 32 bits sont utilisés pour stocker le résultat d'une multiplication ou d'une division, qui est un mot de 64 bits.

#### Registres protégés (registres du coprocesseur 0)

L'architecture MIPS définit 32 registres (numérotés de 0 à 31), qui ne sont accessibles, en lecture comme en écriture, que par les instructions privilégiées. On dit qu'ils appartiennent au coprocesseur système. En pratique, cette version du processeur MIPS en utilise 4 pour la gestion des interruptions et des exceptions

- **SR** Registre d'état (Status Register).
- **CR** Registre de cause (Cause Register).
- **EPC** Registre d'exception (Exception Program Counter).

- **BAR** Registre d'adresse illégale (Bad Address Register).

Dans notre prototype, nous s ?intéresseront uniquement par les registre visibles.

## 4.3 Adressage mémoire

1. **Adresses octet** : Toutes les adresses émises par le processeur sont des adresses octets, ce qui signifie que la mémoire est vue comme un tableau d'octets, qui contient aussi bien les données que les instructions. Les adresses sont codées sur 32 bits. Les instructions sont codées sur 32 bits. Les échanges de données avec la mémoire se font par mot (4 octets consécutifs), demi mot(2 octets consecutifs), ou par octet. Pour les transferts de mots et de demi-mots, le processeur respecte la convention "big endian". L'adresse d'un mot de donnée ou d'une instruction doit être multiple de 4. L'adresse d'un demi-mot doit être multiple de 2. (on dit que les adresses doivent être "alignées"). Le processeur part en exception si une instruction calcule une adresse qui ne respecte pas cette contrainte.
2. **Calcul d'adresse** : Il existe un seul mode d'adressage, consistant à effectuer la somme entre le contenu d'un registre général $R_i$, défini dans l'instruction, et d'un déplacement qui est une valeur immédiate signée, sur 16 bits, contenue également dans l'instruction :

$$adresse = R_i + Déplacement$$

3. **Mémoire virtuelle** : Notre prototype utilise une version du processeur MIPS qui ne possède pas de mémoire virtuelle, c'est à dire que le processeur ne contient aucun mécanisme matériel de traduction des adresses logiques en adresses physiques. Les adresses calculées par le logiciel sont donc transmises au système mémoire sans modifications. On suppose que la mémoire répond en un cycle. Si une anomalie est détectée au cours du transfert entre le processeur et la mémoire, le système mémoire peut le signaler au moyen d'un signal d'erreur, qui déclenche un départ en exception.
4. **Segmentation** : L'espace mémoire est découpé en 2 segments identifiés par le bit de poids fort de l'adresse :
   - $adr31 = 0 \Rightarrow$ segment utilisateur
   - $adr31 = 1 \Rightarrow$ segment système

Quand le processeur est en mode superviseur, les 2 segments sont accessibles. Quand le processeur est en mode utilisateur, seul le segment utilisateur est accessible. Le processeur part en exception si une instruction essaie d'accéder à la mémoire avec une adresse correspondant au segment système alors que le processeur est en mode utilisateur.

## 4.4 Jeu d'instruction

### 4.4.1 Généralités

Le processeur possède 57 instructions qui se répartissent en 4 classesmips1 :

- 33 instructions arithmétiques/logiques entre registres
- 12 instructions de branchement
- 7 instructions de lecture/écriture mémoire
- 5 instructions systèmes

Toutes les instructions ont une longueur de 32 bits et possèdent un des trois formats suivants :

- **Le format J** n'est utilisé que pour les branchements à longue distance (inconditionnels).

  **Le format I** est utilisé par les instructions de lecture/écriture mémoire, par les instructions utilisant un opérande immédiat, ainsi que par les branchements courte distance (conditionnels).

- **Le format R** est utilisé par les instructions nécessitant 2 registres sources (désignés par *RS* et *RT*) et un registre résultat désigné par *RD*.

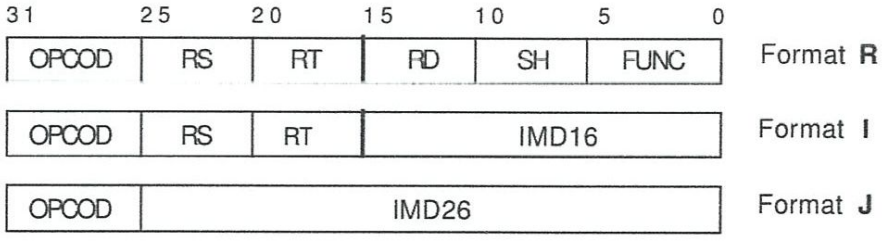



FIGURE *4.1 – formats d'instruction sous MIPS*

| Assembleur | Opération | Format |
|---|---|---|
| Beq Rs, Rt,Label | Branch if Equal $PC \leftarrow PC + 4 + (l * 4)$ if $Rs = Rt$ $PC \leftarrow PC + 4$ if $Rs \neq Rt$ | I |
| Bne Rs, Rt,Label | Branch if not Equal $PC \leftarrow PC + 4 + (l * 4)$ if $Rs \neq Rt$ $PC \leftarrow PC + 4$ if $Rs = Rt$ | I |
| Bgez Rs, Label | Branch if Greater$PC \leftarrow PC + 4 + (l * 4)$ if $Rs \geq 0$ $PC \leftarrow PC + 4$ if $Rs \prec 0$ | I |
| Bgtz Rs, Label | Branch if Greater than 0$PC \leftarrow PC$+4+$(l$*4) if $Rs \succ 0$ $PC \leftarrow PC + 4$ if $Rs \leq 0$ | I |
| Blez Rs, Label | Branch if Less$PC \leftarrow PC$+4+$(l$*4) if $Rs \leq 0$ $PC \leftarrow PC + 4$ if $Rs \succ 0$ | I |
| Bltz Rs, Label | Branch if less than 0$PC \leftarrow PC + 4 + (l * 4)$ if $Rs \prec 0$ $PC \leftarrow PC + 4$ if $Rs > 0$ | I |
| Bgezal Rs, Label | Branch if Greater$PC \leftarrow PC + 4 + (l * 4)$ if $Rs \geq 0$ $PC \leftarrow PC + 4$ if $Rs \prec 0$ $R_{31} \leftarrow PC + 4$ dans les deux cas | I |
| Bltzal Rs, Label | Branch if less than 0$PC \leftarrow PC + 4 + (l * 4)$ if $Rs \prec 0$ $PC \leftarrow PC + 4$ if $Rs \geq 0$ $R_{31} \leftarrow PC + 4$ dans les deux cas | I |
| J Label | Jump $PC \leftarrow PC(31:28)\|\|I * 4$ | J |
| Jal Label | Jump and link $R31 \leftarrow PC + 4$ $PC \leftarrow PC(31:28)\|\|I * 4$ | J |
| Jr Rs | Jump Register $PC \leftarrow Rs$ | R |
| Jalr Rs | Jump and Link Register $R_{31} \leftarrow PC + 4$ $PC \leftarrow Rs$ | R |
| Jalr Rd, Rs | Jump and Link Register $Rd \leftarrow PC$+4 $PC \leftarrow Rs$ | R |

TABLE 4.5 – *Extrait du jeu d'instruction MIPS*

## 4.5 Prototypage du processeur MIPS

Pour faire le prototypage on aurait du passé par plusieurs étapes, qu'on discute dans les sections suivantes.

### 4.5.1 La définition d'un Meta modèle pour le prototypage moyennant l'ADL ArchC

Grace a l'outil EMF d'eclipse nous avons établi un meta-modèle, qui représente tous les composantes d'ArchC nécessaire pour la simulation d'un processeur donné .la figure 4.2 donne une idée sur notre méta-modèle

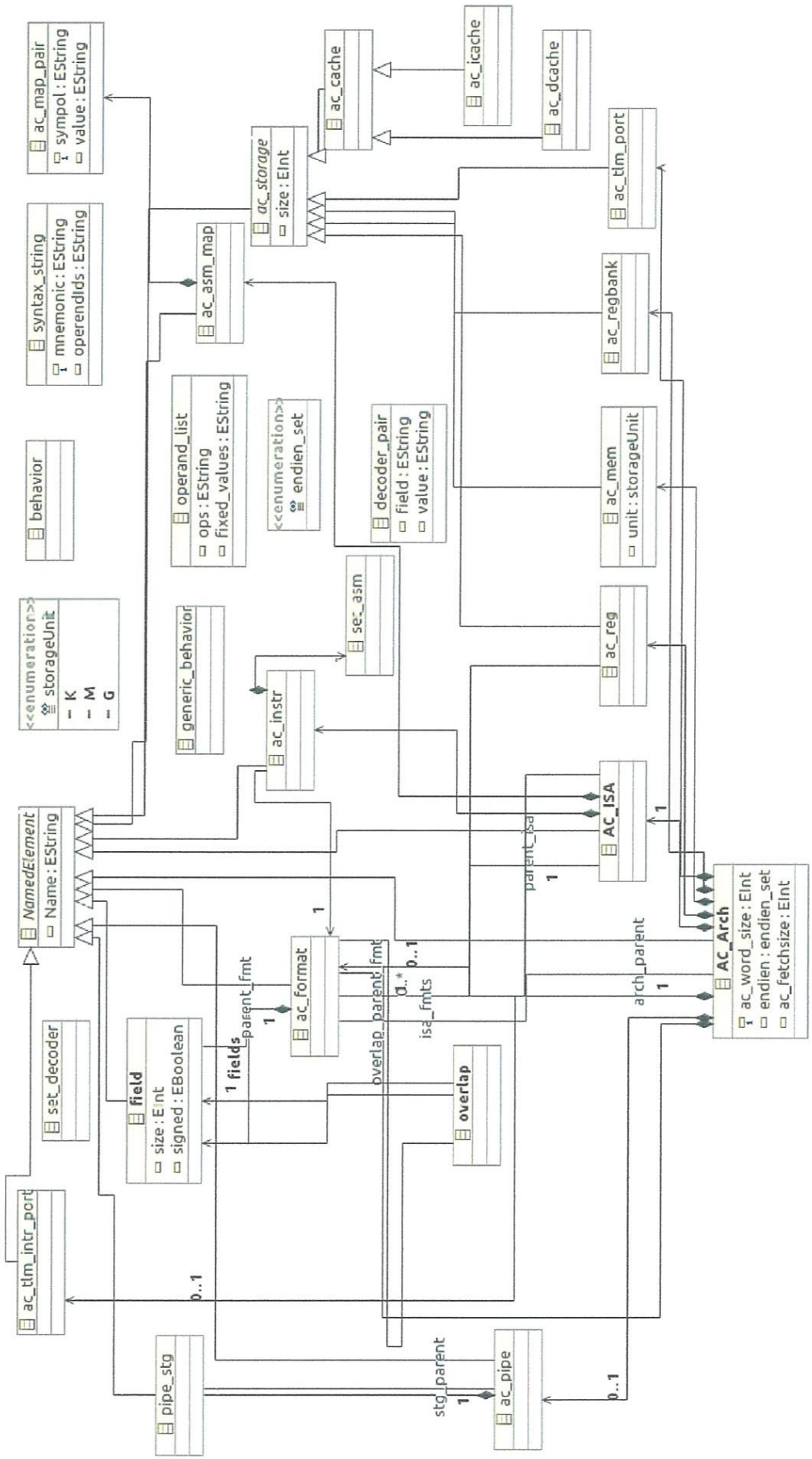



FIGURE 4.2 – *Méta-modèle pour une architecture ARchC*

### 4.5.2 Génération du code pour chaque entité du méta-modèle

avant de manipuler les instances de notre méta-modèle nous devons avoir pour chaque entité du ce dernier une interface et une classe java qui permettent de définir ses caractéristiques et son comportement. Les fragments des codes 4.4,4.3 montrent successivement l'interface et la classe d'implémentation associées à l'entité AC_Arch .

### 4.5.3 Génération de l'éditeur de modèle d'architectures des processeurs

Après avoir défini les codes pour les différentes entités du méta-modèle, nous pouvons utiliser l'une des avantages de l'outil EMF qui nous permet de générer automatiquement un éditeur arborescent sur la base du code java fourni pour chaque entité du méta-modèle. La figure 4.5 montre une capture d'écran lors de l'édition de notre modèle du processeur MIPS avec l'éditeur généré.

Parmi les composantes importantes on cite :

- **AC_Arch** : contient l'architecture de ressource du processeur.
- **AC_ISA** : Comporte les composantes de la classe d'architecture de jeu d'instruction.
- **AC_reg** : c'est les registres utilisés par le processeur.
- **Ac_meme** : pour déclare les objets de stockage.
- **Ac_pipe** : entité pour définir les pipelines du microprocesseur.
- **Set_endian** : Définit *endianness*. L'endianness pour l'architecture Peut être :little ou big.
- **Ac_regbank** : Déclare le banc de registre

## 4.6 Cycle d'exploration de l'architecture

Grâce à notre éditeur (voir figures 4.5 , 4.7), nous avons créé une architecture pour un processeur modèle MIPS selon les descriptions qui ont été abordé au début de ce chapitre

Apres cette phase, nous somme basculé a la partie de vérification et de la simulation pour cela on a crée un générateur de fichier avec l'outil JET pour générer les fichiers essentiel pour ArchC :

```
public interface AC_Arch extends NamedElement {
/**
 * Returns the value of the '<em><b>Ac word size</b></em>' attribute.
 * The default value is <code>"32"</code>.
 * <!-- begin-user-doc -->
 * <p>
 * If the meaning of the '<em>Ac word size</em>' attribute isn't clear,
 * there really should be more of a description here...
 * </p>
 * <!-- end-user-doc -->
 * @return the value of the '<em>Ac word size</em>' attribute.
 * @see #setAc_word_size(int)
 * @see archcmm.ArchcmmPackage#getAC_Arch_Ac_word_size()
 * @model default="32" required="true"
 * @generated
 */
int getAc_word_size();
/**
 * Sets the value of the '{@link archcmm.AC_Arch#getAc_word_size
 *<em>Ac word size</em>}' attribute.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @param value the new value of the '<em>Ac word size</em>' attribute.
 * @see #getAc_word_size()
 * @generated
 */
void setAc_word_size(int value);
....
EList<ac_format> getArch_formats();

/**
 * Returns the value of the '<em><b>Arch mems</b></em>' containment
 *reference list.
 * The list contents are of type {@link archcmm.ac_mem}.
 * <!-- begin-user-doc -->
 * <p>
 * If the meaning of the '<em>Arch mems</em>' containment reference
 * list isn't clear,
 * there really should be more of a description here...
 * </p>
 * <!-- end-user-doc -->
 * @return the value of the '<em>Arch mems</em>' containment
 * reference list.
 * @see archcmm.ArchcmmPackage#getAC_Arch_Arch_mems()
 * @model containment="true"
 * @generated
 */
EList<ac_mem> getArch_mems();
...
} // AC_Arch
```

FIGURE 4.3 – *Interface d'implémentation associées à l'entité AC_Arch*

```
public class AC_ArchImpl extends NamedElementImpl implements AC_Arch {
/**
 * The default value of the '{@link #getAc_word_size()
 *<em>Ac word size</em>}' attribute.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @see #getAc_word_size()
 * @generated
 * @ordered
 */
protected static final int AC_WORD_SIZE_EDEFAULT = 32;
/**
 * The cached value of the '{@link #getAc_word_size()
 *<em>Ac word size</em>}' attribute.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @see #getAc_word_size()
 * @generated
 * @ordered
 */
protected int ac_word_size = AC_WORD_SIZE_EDEFAULT;
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
protected AC_ArchImpl() {
super();
}
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public int getAc_word_size() {
return ac_word_size;
}

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public void setAc_word_size(int newAc_word_size) {
int oldAc_word_size = ac_word_size;
ac_word_size = newAc_word_size;
if (eNotificationRequired())
eNotify(new ENotificationImpl
(this, Notification.SET, ArchcmmPackage.AC_ARCH__AC_WORD_SIZE
, oldAc_word_size, ac_word_size));
}
```

```
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public EList<ac_mem> getArch_mems() {
if (arch_mems == null) {
arch_mems = new EObjectContainmentEList<ac_mem>(ac_mem.class, this ,
 ArchcmmPackage.AC_ARCH__ARCH_MEMS);
}
return arch_mems;1
}
...
} //AC_ArchImpl
```

FIGURE *4.4 - Classe d'implémentation associées à l'entité AC_Arch*

- Un fichier de Description des ressources *Ac_Arch.ac.*
- Un fichier de Description du jeu d'instructions *Ac_Isa.ac.*
- Un fichier pour décrire le comportent des instructions *Ac_Isa.cpp.*

Ces fichier serons l'entré de compilateur *acsim*(voir figure 4.8) d'archC qui générera un code source en SystemC. Ce code est modifié [1] avant d ?être compiler par le compilateur GCC.

L ?exécutable obtenue représente notre ISS qui seras alimenté en entré par un Benchmark et produit des résultats qui vont être récupérer par un analyseur de (cette opération est faite manuellement) afin de mettre a jours l'architecture de départ exprimé en EMF (voir figure 4.8) en fonction des critères de performances prédéfinies. La figure 4.9 représente les résultats obtenue suite à l'exécution du Benchmark par notre ISS avec des données de déférentes tailles.

1. Pour des raisons de compatibilité avec les version de compilateur

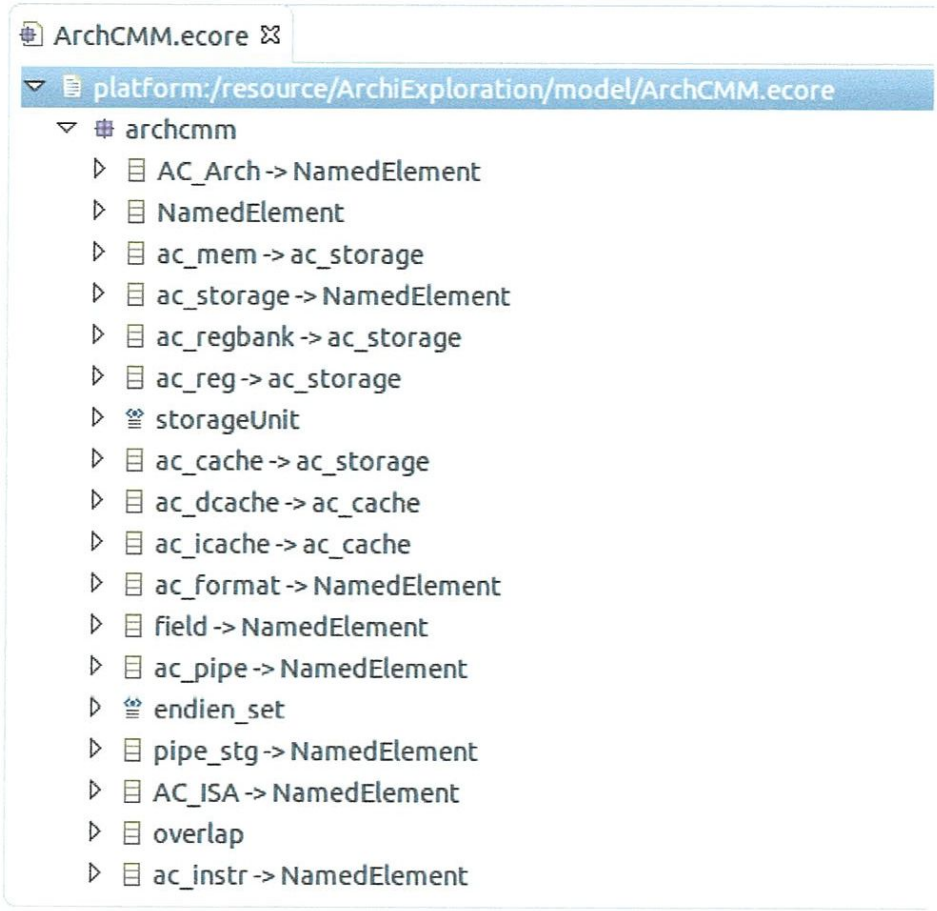



FIGURE 4.5 – *Edition de l'architecture MIPS avec notre éditeur*

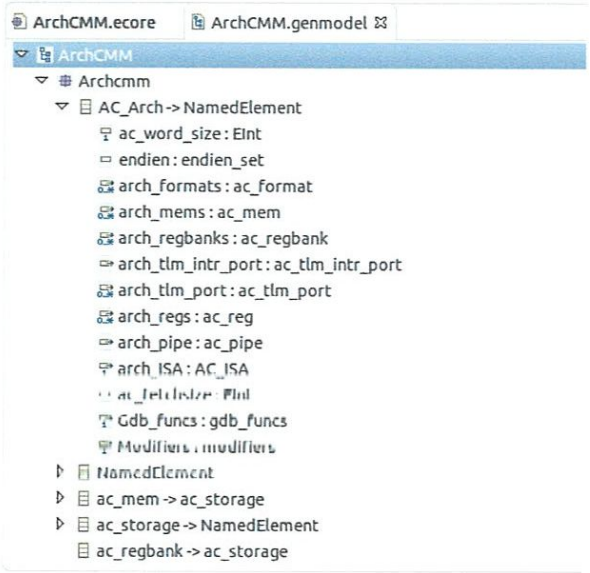



FIGURE 4.6 – *Extrait de l'édition du modèle d'architecture de MIPS*

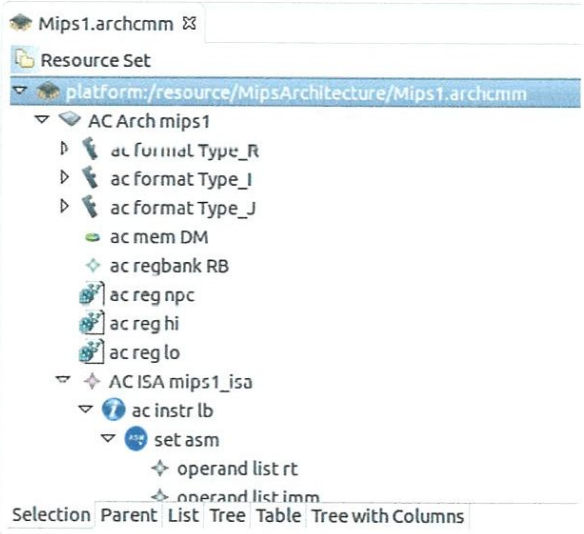



FIGURE 4.7 – *Extrait de la description du jeu d'instruction avec notre editeur*

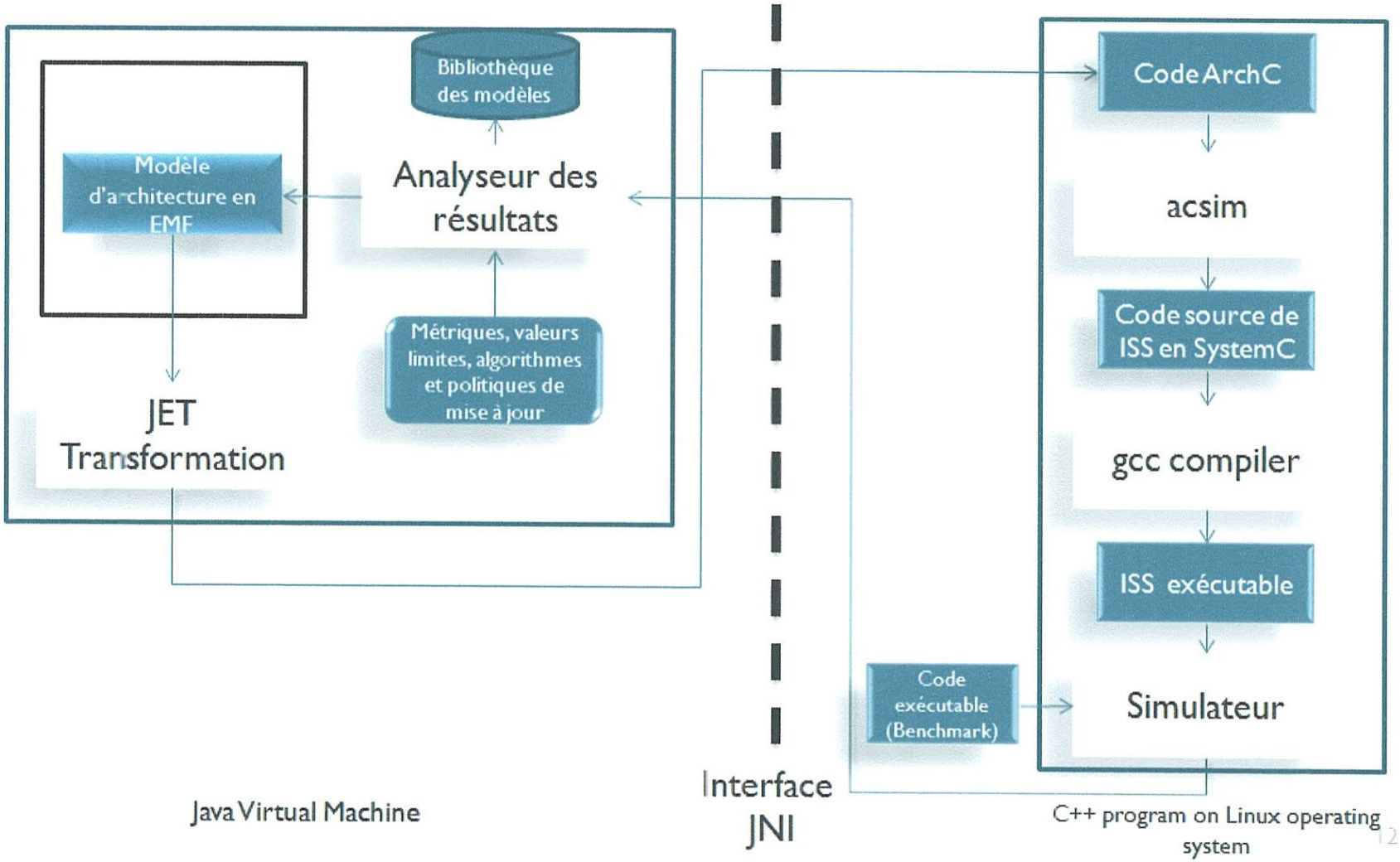



FIGURE *4.8 – Cycle d'exploration d'architecture proposé*

Table Printing

| Name Of CPU | Nombre de Ligne | Times user | Times system | real Times | Number of instru... | Simulation spee... |
|---|---|---|---|---|---|---|
| mipsl | 500000 | 0.26 | 0.02 | 0.30 | 14412623 | 55433.17 |
| mipsl | 400000 | 23.29 | 0.10 | 23.42 | 1183382279 | 50810.75 |
| mipsl | 300000 | 23.65 | 0.11 | 23.79 | 1183382279 | 50037.31 |
| mipsl | 200000 | 21.76 | 0.12 | 21.91 | 989374482 | 45467.58 |
| mipsl | 100000 | 27.73 | 0.39 | 28.18 | 1183382279 | 42675.16 |
| mipsl | 1008 | 0.03 | 0.00 | 0.03 | 1212365 | 40412.17 |

Print

FIGURE *4.9 – Statistiques après la simulation*

On a aussi pu illustrer les résultats de simulation dans un histogramme,ou l'axe des X représente la taille des données en entrée et l'axe des Y représente le nombre des instructions exécutées par l'ISS (Figure4.10).

Nous rappelons que notre objectif est n'est pas d ?analyser les résultats obtenus mais plutôt les mettre dans la disponibilité du concepteur pour servir au bon choix des architectures.

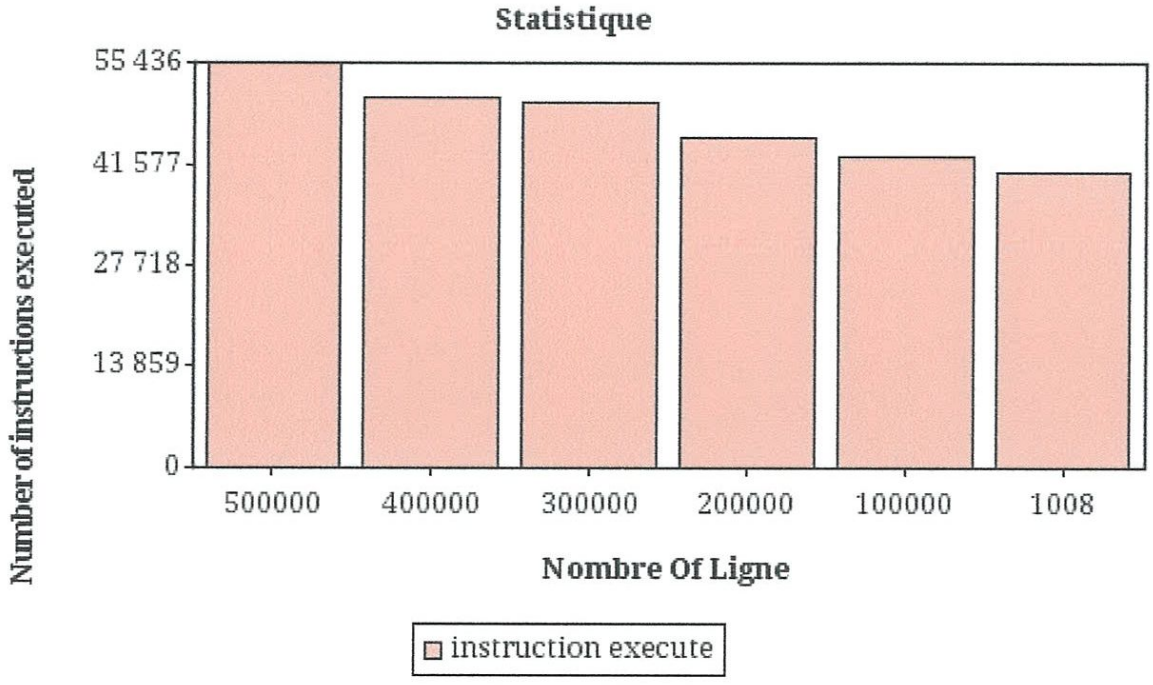



FIGURE *4.10 – Statistiques après la simulation*

## 4.7 Conclusion

A la fin de ce chapitre, nous pouvons dire qu'on a réussi à mettre en œuvre notre approche MDE pour le prototypage et l'exploration des architectures des processeurs. En

effet à partir d'un modèle d'un processeur édité avec un éditeur arborescent très fiable, nous somme arrivé à produire une description textuel en ArchC, cette description va être l'entrée d'un générateur d'outils que nous l'avons exploité uniquement pour générer un simulateur de jeu d'instruction en SystemC. Notre simulateur accepte plusieurs benchmark en entrée et produit des résultats qu'on laisse le soin aux concepteurs des processeurs de les analyser.

# Conclusion Générale

Tous au long de ce mémoire, nous avons suivi une démarche progressive pour aboutir à une vision claire de notre contribution. En effet, nous avons discuté dans un premier temps un ensemble des langages de description d'architecture pour arriver à en dégager l'ADL ArchC qui présente plusieurs caractéristiques qui l'avantagent par rapport aux autres et parmi lesquels on précise qu'il génère un code en SystemC qui est considéré à son tour comme l'un des outil les plus puissants pour la simulation des systèmes embarqué et en particulier les processeurs. Ensuite, nous avons réussi à intégrer les flux de conception proposés par ces langages dans une méthodologie plus générale qui s'appuie sur les modèles et les transformation de ces modèles d'une manière complètement automatique, nous avons obtenu alors un cycle générique de prototypage et d'exploration qui minimise considérablement l'effort du concepteur des architectures processeur et lui fournissant des statistiques très pratiques sur des architectures qu'il peut les adapter très facilement grâce à notre travail.

Nous avons utilisé pour la réalisation de notre travail un ensemble diversifié d'outils qui vient à leur tête l'environnement Eclipse qu'on aurait dû maîtriser ses techniques (notamment le développement des plugins) d'une manière approfondie. Nous étions aussi dans la capacité de surmonter le problème de communication des parties complètement hétérogènes (par exemple un code java avec un code c++) . Tout ce travail est élaboré sous l'environnement linux ce qui nous a demandé un investissement majeur en terme du temps pour cerner les différents problèmes de configuration, sachant que notre travail touche des aspects très liés au systèmes.

La puissance de notre approche réside dans la capacité d'évolution des modèles guidées par les résultats de la simulation fournis suite à l'exécution des modèles SystemC. Nous envisageons dans la suite de ce travail d'élargir notre espace d'exploration et de prototypage pour couvrir aussi bien les systèmes multiprocesseurs.

Par ailleurs, Nous devons définir des spécifications référentielles qui reposent sur des méta-modèles solides qui permettent d'exprimer les concepts relatifs aux systèmes temps réel et systèmes embarqués tels que la synchronisation, la concurrence, et les annotations des contraintes qualitatives et quantitatives.

# Bibliographie


[1] Hardware Description Language Based on the Verilog Hardware description Language. New york , 1996. pp. 1364-1995. Technical report.

[2] IEEE standard VHDL language manual. new york , 1994. pp. 1076-1993. Technical report.

[3] Guido Araújo al. *The ArchC Architecture Description Language v2.0 Reference Manual.* Laboratoire de systèmes informatiques (LSC) de l'Institut d'informatique de l'Université de Campinas (IC-Unicamp), Av. Albert Einstein, 1251 13084-971 PO Box 6176 - Campinas/SP, August 2007.

[4] David C. Black and Jack Donovan. *SystemC : From the Ground Up.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[5] Xavier Blanc, Olivier Collaborateur. Salvatori, and Philippe Desfray. *MDA en action : ingénierie logicielle guidée par les modèles.* Architecte logiciel. Eyrolles, Paris, 2005.

[6] Robert Britton. *MIPS Assembly Language Programming.* Computer Science Department California State University, Chico Chico, California, 2004.

[7] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins.* Addison-Wesley Professional, 3 edition, 2008.

[8] Benchaabene Hadda et Haddade Sara. Environnement visuel pour la modélisation des systèmes embarquès. Memoire master, Université 08 mai Guelma, 2011.

[9] M. Froricke. *The NML Machine Description Formalism.* Bericht (Technische Universität Berlin. Fachbereich 20, Informatik). Technische Universität Berlin, Fachbereich 20, Informatik, 1991.

[10] Brian J. Gough and Richard M. Stallman. *An Introduction to GCC.* Network Theory Ltd., 2004.

[11] D. Große and R. Drechsler. *Quality-Driven SystemC Design.* Springer, 2009.

[12] T. Grötker. *System Design with SystemC.* Springer, 2002.

[13] Java Emitter Template (JET) Lars Vogel. http ://www.vogella.com/, 2009.

[14] Peter Marwedel. The mimola design system : Tools for the design of digital processors. In *Proceedings of the 21st Design Automation Conference*, DAC '84, pages 587–593, Piscataway, NJ, USA, 1984. IEEE Press.

[15] Prabhat Mishra and Nikil Dutt. Architecture description languages for programmable embedded systems. In *In IEE Proceedings on Computers and Digital Techniques*, pages 285–297, 2005.

[16] A.Chaoui O.Labbani N.Berrehouma, E.Bourennane. Visual tool for systemc functionnal mpsoc design. 16-17 November 2011.

[17] E.Bourennane N.Berrehouma, A.chaoui. Une approche mda pour l'exploration des architecture mpsoc.

[18] Object Management Group. http ://www.omg.org, October 2010.

[19] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition : The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[20] Nikil Dutt Prabhat Mishra. *Processor Description Languages.* 9 juillet 2008.

[21] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. Archc : a systemc-based architecture description language. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 66–73, 2004.

[22] IEEE Std. Systemc language reference manual. ieee std 1666-2005. 31 mars 2006. Technical report.

[23] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

[24] COURS EN LIGNE SYSTEMC. http ://comelec.enst.fr/hdl/sc-intro.html, October 2010.

[25] V. Zivojnovic, S. Pees, and H. Meyr. Lisa-machine description language and generic machine model for hw/sw co-design. In *VLSI Signal Processing, IX, 1996., [Workshop on]*, pages 127–136, 1996.