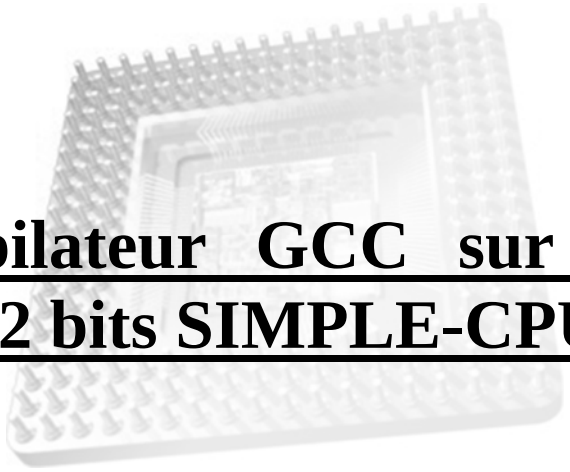
Institut Supérieur d'Informatique, de Modélisation et de leurs Applications.
Campus Universitaire des Cézeaux
63000 Clermont-Ferrand
France



Portage du compilateur GCC sur le processeur RISC 32 bits SIMPLE-CPU.

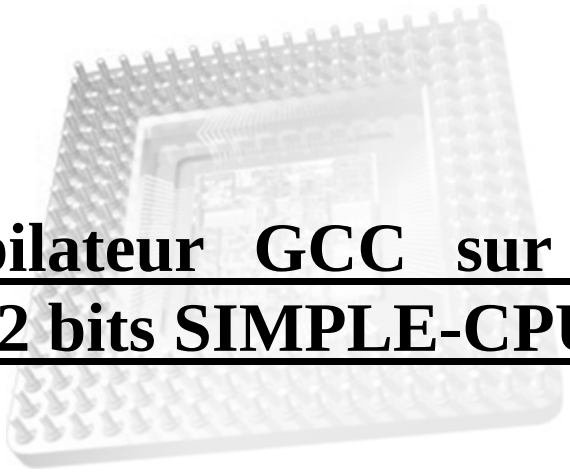
Rapport de projet ISIMA 3ème année.

Etudiants ISIMA : **Michel LAVERNE**
Wilfrid ROUX

Responsable : **Paul PINAULT**

Mars 2006

Institut Supérieur d'Informatique, de Modélisation et de leurs Applications.
Campus Universitaire des Cézeaux
63000 Clermont-Ferrand
France



Portage du compilateur GCC sur le processeur RISC 32 bits SIMPLE-CPU.

Rapport de projet ISIMA 3ème année.

Etudiants ISIMA : **Michel LAVERNE**
Wilfrid ROUX

Responsable : **Paul PINAULT**

Mars 2006

Résumé

L'évolution des langages informatiques a amené les développeurs à ne plus se soucier des contraintes matérielles et de la programmation de bas niveau, pour développer dans des langages de plus haut niveau. Ces différents niveaux d'abstractions peuvent être classés comme suit : au plus bas se trouvnt des langages dépendant du matériel comme l'assembleur, juste au dessus se trouvent des langage comme le C, puis le C++. On pourrait caractériser le haut de cette hiérarchie avec des langages extrêmement haut niveau comme l'UML. Cependant un schéma UML n'a jamais permis de fournir un programme fonctionnel. Au jour d'aujourd'hui aucun logiciel ne peut accepter ce degré d'abstraction. Mais les langages de haut degré d'abstraction sont utilisés pour faciliter le portage des programmes entre différentes plateformes, mais également dans un souci de gain de temps de développement. Les outils utilisés pour produire du code machine à partir de ces langages sont les **compilateurs**.

Le projet ***SIMPLE-CPU*** doit fournir à la communauté du logiciel libre un ensemble de composants à la fois matériel et logiciel pour comprendre les bases fondamentales du fonctionnement des ordinateurs. Un fois le cœur du processeur conçu, et le jeu d'instructions défini, un **compilateur** C doit être ajouté au projet. Ceci permettant aux programmeurs de réutiliser des portions de codes déjà éprouvées et ne pas réinventer la roue. Le **compilateur** C est une pièce majeure de la réussite du projet ***SIMPLE-CPU***.

Le projet présenté ici montre les réflexions et choix fait pour le **portage** d'un **compilateur** issu du monde open source (***GCC***) sur le processeur ***SIMPLE-CPU***. Bien que la charge de travail ait été telle que le projet n'a pu livrer une version fonctionnelle du **compilateur**, l'ensemble des concepts indispensables à cette réalisation ont été abordés et discutés, permettant ainsi de fournir une base de travail pour la suite du projet.

Mots clés : ***GCC***, **Compilateur, portage, *SIMPLE-CPU***.

Abstract

Evolutions of programming language lead developers to no longer deal with hardware constraints and low-level programming, to develop in higher level codes. These different abstraction levels may be ranked as follow: bottom is the hardware dependant languages such as assembly just above is the C language, and the C++ and on top are the abstract language like UML. However drawing an UML diagram as never permit to produce functional programs. At the moment no software can handle this level of abstraction. But abstract languages, as high as possible, are used to program easier portage from a platform to another, and also by worry for the time of development. The tools used to build machine code from these high-level languages are called **compiler**.

The ***SIMPLE-CPU*** project is to provide to the open source community a set of component both software and hardware in order to understand the fundamental functions of a computer. Once the core was designed, and the instructions set define, a C **compiler** has to be provided to the project. In fact C code is the most used language for embedded component as the ***SIMPLE-CPU***. This allow programmer to reuse already coded software components and not reinvent the wheel every time. C **compiler** is a masterpiece for the success of the ***SIMPLE-CPU*** project.

The project introduces here shows the reflections and choices made for the **portage** of an open source **compiler** (***GCC***) on the ***SIMPLE-CPU*** processor. Even if the workload was so heavy that the project did not produce a functional version of the **compiler**, the whole concept have been discussed and studied, allowing the project to have a starting point for the following.

Key words : ***GCC***, **Compiler**, ***SIMPLE-CPU***, **portage**.

Table des Matières

Résumé 3
Abstract 4
Table des Matières 5
Table des Illustrations 6
Table des Codes sources 6
Glossaire 7
1. Introduction 9
1.1. Les compilateurs (théorie) 9
2. Contexte de travail 10
2.1. Simple CPU 10
1.1.1. Présentation du projet 10
2.1.2. Architecture 11
2.1.3. Jeu d'instructions 11
2.2. GCC 12
2.2.1. Présentation 12
2.2.2. Portage de GCC 14
2.2.3. Les fichiers de description 14
2.2.4. Le métalangage RTL 15
2.3. Binutils 17
2.3.1. Présentation 17
2.3.2. Les fichiers de description et Cgen 18
3. Réalisation 21
3.1. Etude préliminaire 21
3.1.1. Orientations possibles 21
3.1.2. Orientation choisies 21
3.1.3. Le FR30 22
3.2. Compilation de GCC pour FR30 23
3.3. Modification des fichiers de descriptions 24
4. Conclusion 29
Références 30

Table des Illustrations

Figure 1: Structure de GCC 13
Figure 2 : Arborescence des processeurs 18
Figure 3 : Structures RTL 19
Figure 4 : Définition de l'instruction ADDd 20
Figure 5 : Tableau comparatif des architectures Simple-CPU / FR30 22
Figure 6 : Arborescence créée 23
Figure 7 : Commandes de compilation de GCC pour FR30 23
Figure 8 : Commande de compilation des binutils. 25
Figure 9 : Configuuration du make de GCC 26
Figure 10 : Ajout dans le PATH 26
Figure 11 : Compilation et installation de GCC 26
Figure 12 : Compilation d'un fichier d'exemple avec le nouveau compilateur. 26
Figure 13 : Emploi du temps du projet 29

Table des Codes sources

Code 1 : Exemple de code RTL 16
Code 2 : Structure des opcodes 24
Code 3 : Opcode de l'instruction 'mul' FR30 24
Code 4 : Opcode de l'instruction 'ISIMA' modifiée. 25
Code 5 : Pattern RTL de l'instruction 'mul' FR30. 25
Code 6 : Pattern RTL modifié pour l'instructiuon 'ISIMA' 25
Code 7 : Code source de test 1 26
Code 8 : Code source de test 2 26

Glossaire

B

Back-end : Dans le domaine de l'informatique, se réfère à l'étape finale d'un flux de traitement. C'est le terme employé dans le contexte de *GCC* pour désigner les composants permettant la génération de code pour une machine donnée.

Binutils : (ou GNU Binutils) ensemble d'outils de programmation, pour la manipulation de code objet.

Bit : quantité élémentaire d'information représentée par un chiffre binaire. Un bit peut prendre les valeurs «vrai », 1, ou « faux », 0.

Byte : (octet) Unité de mesure communément utilisée en informatique, elle désigne une suite de 8 bits.

C

Compilateur : Programme informatique qui convertit un langage source en un langage cible.

Cross-compilation : (compilation croisée) Méthode de compilation permettant de générer le code pour une architecture spécifique sur une machine d'architecture différente. Ex : Compilation de programme PowerPC sur machine x86.

D

Double-word : (double mot) Unité de longueur utilisée en informatique, elle désigne une suite de 32 bits, soit 4 octets.

F

Front-end : Dans le domaine de l'informatique, se réfère à l'étape initiale d'un flux de traitement. C'est le terme employé dans le contexte de *GCC* pour désigner les composants permettant le *parsing* et la génération de l'arbre à partir de code source.

G

GCC : Gnu **C**ompiler **C**ollection, ensemble de compilateurs développés par le projet GNU.

GNU : Gnu's **N**ot **U**nix, Système d'exploitation libre.

Guile : (ou GNU Guile) est un interpréteur de *Scheme*.

L

LISP : Langage de programmation informatique fonctionnel, reconnaisable à sa syntaxe parenthèsée.

Logiciel Libre : Logiciel qui peut être utilisé, copié, étudié, modifié et redistribué sans restriction.

M

Middle-end : Etape intermédiaire d'un flux de traitement.

O

Octet : Unité de mesure communément utilisée en informatique, elle désigne une suite de 8 bits.

Opcode : Operationnal **Code,** nom donné au code (binaire ou mnémonique) décrivant une instruction.

P

PATH : (chemin) variable d'environnement des systèmes d'exploitation.

Parseur : Analyse une séquence d'entrée afin d'en déterminer sa structure grammaticale.

R

RTL : Register **T**ransfer **L**anguage, représentation intermédiaire utilisée par *GCC*, exprimé en LISP.

RISC : Restricted **I**nstruction **S**et **C**omputer (Ordinateur à jeu d'instruction réduit) architecture matérielle de microprocesseur, basée sur un jeu d'instruction de taille fixe et de nombre réduit.

S

Scheme : Langage de programmation fonctionnel dérivé du *LISP*.

W

Word : (mot) Unité de mesure communément utilisée en informatique, elle désigne une suite de 16 bits, soit 2 octets

1. Introduction

Le compilateur est une pièce maîtresse pour tout composant électronique micro programmé. En effet il permet au développeur de s'abstraire de la contrainte du matériel pour se consacrer à la production de code. Munir un nouveau composant d'un compilateur permet à celui-ci de pouvoir être utilisé avec des codes sources issues d'orientations diverses. Dans le cadre du projet *SIMPLE-CPU*, la partie logicielle était encore à réaliser. C'est la raison pour laquelle le sous projet de portage du compilateur *GCC* fut mis en place.

Le présent rapport est divisé en deux grandes parties, avec tout d'abord une présentation des outils utilisés, et ensuite la présentation du travail effectué et les résultats obtenus. Mais dans un premier temps, nous présenterons une rapide introduction sur les compilateurs.

1.1. Les compilateurs (théorie)

Un compilateur est un programme informatique qui traduit un langage, le langage source, en un autre, appelé le langage cible, en préservant la signification du texte source. Ce schéma général décrit un grand nombre de programmes différents ; et ce que l'on entend par « signification du texte source » dépend du rôle du compilateur. Lorsqu'on parle de compilateur, on suppose aussi en général que le langage source est, pour l'application envisagée, de plus haut niveau que le langage cible, c'est-à-dire qu'il présente un niveau d'abstraction supérieur.

En pratique, un compilateur sert le plus souvent à traduire un code source écrit dans un langage de programmation en un autre langage, habituellement un langage d'assemblage ou un langage machine. Le programme en langage machine produit par un compilateur est appelé code objet.

Le premier compilateur a été écrit par Grace Hopper dans les années 50. Cependant on crédite souvent l'équipe de John Backus de chez IBM du premier compilateur complet, qui fut réalisé pour le langage Fortran en 1957.

2. Contexte de travail

Le projet de 3ème année de l'ISIMA se compose d'un volume horaire estimé à 120h par monôme. La plupart des projets étant mené en binôme. Le présent projet a la particularité d'impliquer bien plus que les seuls étudiants de l'ISIMA. Tout d'abord par sa philosophie qui se veut ouverte sur la communauté du logiciel/matériel libre. Dans ce cas l'ensemble des documents rédigés devra être publié et mis à disposition grâce au site Internet du projet [WEB1]. D'autre part, en plus de Mr PINAULT, instigateur et responsable du projet, une équipe d'étudiants de l'IUT d'informatique de Clermont-Ferrand travaille sur une autre partie logicielle. Ils mettent au point un compilateur assembleur, et un simulateur pour le projet.

2.1. Simple CPU

Ce projet est le fruit du travail de Mr Paul PINAULT. Il le décrit ainsi :

« Ce projet est né de plusieurs idées, tout d'abord le souhait personnel de travailler sur un projet matériel alors que mon activité professionnelle m'en a détournée ces dernières années. Ensuite, la volonté de réaliser ce à quoi je réfléchi depuis déjà plusieurs années la réalisation d'un système informatique complet en partant de rien ou presque, comme ont pu le faire les pionniers de l'informatique des années 60 (la technologie d'aujourd'hui en plus). Une sorte de retour aux sources des temps modernes pour comprendre mais aussi expliquer comment un ordinateur fonctionne.

Je souhaite par ce projet démystifier l'ordinateur, son fonctionnement interne ; mettre à disposition de la communauté un support pour des cours et TP et offrir, à ceux qui trouveront à ce projet un fort intérêt, un outil pour la création de produits commerciaux.
Ce projet est une brique parmi d'autre dans le monde de l'informatique libre, celle que l'on ne réalise pas que pour les bénéfices mais aussi pour l'offrir aux autres. J'espère ainsi que ce travail amènera d'autres individus à transformer créer de cela des applications, plus complexes, utiles à la société.

Les contributeurs de tous poils, privés, professionnels, particuliers, entreprises, ados, adultes, retraités, hommes, femmes, Français ou Chinois sont les bienvenues à bord de ce projet que je ne pourrai concevoir totalement seul. C'est ainsi que pour les raisons ci-dessus je prends à ma charge et pour mon plaisir la création d'une v.1 fonctionnant du coeur mais que je vous encourage à proposer votre aide pour faire vivre ce projet et l'enrichir. »

La philosophie de ce projet est donc clairement orientée vers le développement collaboratif. Ainsi, un effort particulier sera apporté à l'utilisation d'outils issus du monde du logiciel/matériel libre.

1.1.1. Présentation du projet

Le portage informatique consiste à implémenter un logiciel, une fonctionnalité, voire un système d'exploitation dans un autre environnement que celui d'origine. Cet environnement est donc soit logiciel, soit matériel.

Le portage informatique revient souvent à reprendre le code source du composant existant dans son environnement initial, puis de lui apporter les modifications nécessaires pour qu'il puisse fonctionner sur la plate-forme de destination. Dans ce type de cas, le développeur sera reconnaissant à ceux ayant conçu le dit composant d'avoir utilisé des pratiques visant à la portabilité, par exemple en évitant toute violation de la norme du langage d'implémentation.

2.1.2. Architecture

Simple CPU est un processeur RISC 32 bits avec accès linéaire à la mémoire, utilisant la méthode *load-store*. Il est constitué du minimum d'instructions possibles avec nombres de paramètres.

Le cœur *SIMPLE-CPU* contient 32 registres *double word* (32 bits) nommés R00 à R31 dont 29 à usage général. En effet :

R00 est réservé au compteur de programme.
R01 est réservé pour le pointeur de pile.
R02 est utilisé pour les *flags*, quelques bits de configurations spécifiques.

R00 et R02 peuvent être utilisés dans n'importe quelle instruction comme des registres standard.

Les caractéristiques principales de l'architecture Simple CPU sont les suivantes :

- Toutes les instructions sont de 32 bits de longueur, même si elles en nécessitent moins.
- Chacune s'exécute en un seul cycle.
- L'espace mémoire n'est pas divisé entre espace d'instructions et espace de données.
- Les instructions commencent toujours sur une adresse multiple de 4, mais les données peuvent être au format *octet* (*byte*), *word* ou *double-word*.

2.1.3. Jeu d'instructions

Le jeu d'instructions de *SIMPLE-CPU* est composé de 21 opérations de bases auxquelles se rajoutent les différents modes que chacune peut avoir.

On trouve ainsi les instructions d'accès aux données :

STORE : Stockage en mémoire.
LOADd : Chargement direct de registre
LOADm : Chargement de registre à partir de la mémoire.
LOADr : Chargement de registre à partir d'un autre registre.

Les instructions logiques :

AND : ET logique 32 bits.
OR : OU logique 32 bits.
XOR : OU EXCLUSIF logique 32 bits.
ROT : Permutation de bits.

Les instructions de saut :

CPYB : Recopie de bit d'un registre source vers un registre destination.
JB : Saut conditionnel.
JMPd : Saut direct.
JMPr : Saut à une sous-fonction identifiée par un registre.

Les opérations arithmétiques :

ADDd : Addition de valeur directe.
ADDr : Addition de registres.
MUL : Multiplication, résultat stocké dans deux registres.
DIV : Division entière.

Les instructions de sous programme :

CALL : Appel d'une sous fonction.
CALLr : Appel d'une sous fonction identifiée par un pointeur en registre.
PUSH : Sauvegarde de registre.
POP : Restauration de registre.

L'instruction sans opération :

NOP

La plus part de ces instructions acceptent plusieurs modes de fonctionnement (ou *alias*). C'est ainsi que même si le jeu d'instructions semble être réduit à une vingtaine d'instructions élémentaires, l'ensemble des modes de fonctionnement permet d'atteindre un total d'une centaine d'instructions.

2.2. GCC

2.2.1. Présentation

Acronyme de "GNU Compiler Collection". *GCC* est le compilateur du projet GNU lancé en 1984 par Richard Stallman, alors chercheur du laboratoire sur l'Intelligence Artificielle au MIT (États-Unis). Il s'agit d'une collection de logiciels libres intégrés capables de compiler divers langages de programmation, dont C, C++, Objective-C, Java, Ada et Fortran.

Avant d'énumérer les différentes possibilités de réaliser un portage de *GCC* vers une cible, nous avons étudié la structure et le fonctionnement internes de *GCC*. Il faut savoir que

GCC a été développé de telles sortes que le coeur du compilateur reste indépendant du langage à compiler et de la cible. Donc nous pouvons décomposer *GCC* en 3 modules (Figure 1: Structure de GCC)

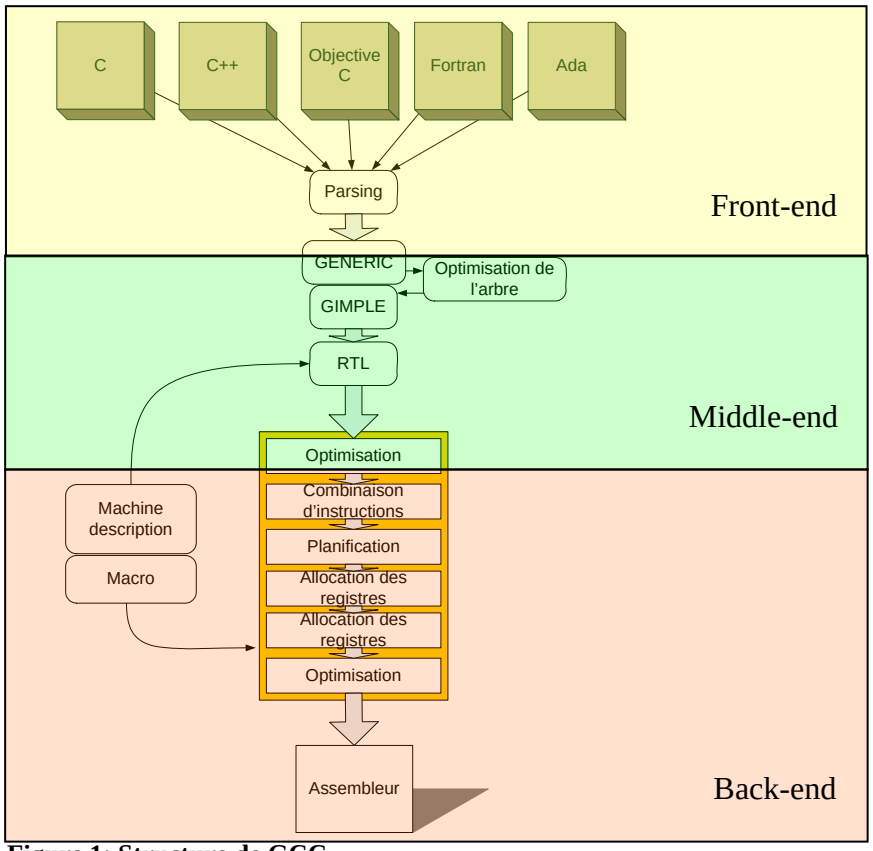


Figure 1: Structure de GCC

Module 1, le *front-end*:

Ce module assure la fonction de parseur et génère l'arbre qui va être utilisé par la suite. Il comporte donc les étapes nécessaires au *parsing.*

- Le découpage du programme en lexèmes (analyse lexicale).
- La vérification de la correction de la syntaxe du programme (analyse syntaxique).
- L'analyse des structures de données (analyse sémantique).

Module 2, le *middle-end* :

Cette partie, interne à *GC*, *C* a plusieurs rôles :

- L'optimisation de l'arbre créé par la *front-end.*
- La transformation de l'arbre en code intermédiaire.
- L'application de techniques d'optimisation sur le code intermédiaire.

Module 3, le *back-end* :

Enfin ce troisième module permet la génération du code de destination spécifique à la machine de cible, grâce à :

- L'allocation de registres et la traduction du code intermédiaire en code objet, avec éventuellement l'insertion de données de débogage et d'analyse de l'exécution ;
- Enfin vient la phase d'édition des liens.

2.2.2. Portage de GCC

Pour modifier *GCC,* il y a quatre possibilités :

La première consiste à modifier le code source de *GCC* pour l'adapter à la cible voulue, à savoir le processeur *SIMPLE-CPU*. Mais avec cette méthode la philosophie même de *GCC* aurait été perdue. En effet *GCC* a été conçu et développé afin de pouvoir réaliser la compilation d'un grand nombre de langages sources, vers un grand nombre de processeurs cible, tout en restant indépendant que ce soit de l'un ou de l'autre. Cette possibilité a rapidement été écartée du fait de la complexité du code de *GCC,* mais également du manque de documentation de celui-ci (qui est un problème latent de *GCC*).

La deuxième approche fut de convertir le code assembleur généré par *GCC* vers le code assembleur pour le processeur *SIMPLE-CPU*. Même si cette approche est totalement réalisable, elle présente deux principaux problèmes. Tout d'abord il revenait à déplacer le problème en créant un traducteur de code assembleur et non plus un compilateur. Deuxièmement, le bon fonctionnement d'un tel outil n'aurait pas put être certifié du fait de la non exhaustivité des tests pouvant être réalisés. En effet même si un tel programme pouvait fournir des résultats satisfaisant sur une certaine quantité de code, son fonctionnement n'aurait pu être garanti sur des codes sources de complexité élevé et faisait intervenir des méthode peut utilisées. On pensera particulièrement à l'utilisation des registres/mémoires, points sensibles lors de la compilation.

Finalement l'utilisation de la versatilité de *GCC* semble être la meilleure possibilité. En effet la philosophie de *GCC* est d'être le plus indépendant possible tant des langages sources que des machines de destinations. Des fonctionnalités pour étendre le champ d'action de *GCC* y ont été incorporées et il est donc « facilement » possible d'ajouter le support de nouveaux langages source, avec l'ajout de *front-end,* ou de machines cibles, avec l'ajout de *back-end.* C'est bien l'ajout d'un *back-end SIMPLE-CPU* qui nous intéresse dans le cadre de ce projet. En effet la création de *back-end* consiste en l'écriture de fichiers de description utilisés pour la compilation de *GCC.* Ainsi un compilateur *GCC* spécifique à la cible est compilé fonctionnant sur une machine hôte (*cross-compilation*). C'est la méthode selon laquelle le projet a été mené. Là encore deux approches sont ouvertes. Afin de ne pas créer ces fichiers de description à partir de rien, méthode longues et inutiles car consommatrice de temps et de ressources, il est conseillé de s'inspirer des fichiers de description fournis avec *GCC,* et si possible d'une cible dont l'architecture est proche de celle visée.

2.2.3. Les fichiers de description

Comme on a pu le voir, la compilation de *GCC* pour un cible particulière nécessite une description précise de l'architecture de cette cible. Cette description s'effectue grâce à plusieurs fichiers placés dans le répertoire ./gcc/config/<arch>/ . Ces fichiers qui sont au nombre minimal de 4 sont les suivants :

- <arch>.md

Le fichier principal de la description de machine. Il contient les patterns RTL issus du code intermédiaire généré par *GCC* et le code ASM résultant.

- <arch>.h

Le fichier <arch>.h définit le nombres de registres, les instructions, le nombre de cycles utiles pour chacune, les modes de fonctionnement et tout les paramètres propres à la machine cible.

- <arch>_proto.h

Le fichier <arch>_proto.h définit les prototypes de fonctions que l'on trouve dans le fichier <arch>.c

- <arch>.c

Ces fichiers contiennent l'ensemble des fonctions et leurs prototypes. Ces fonctions servent dans la génération de code assembleur et dans la reconnaissance des patterns RTL.

Et éventuellement :

- t-<arch>

Compléments au makefile de *GCC* lors de l'utilisation de cette machine.

- <arch>-mods.def
- <arch>.opt

Gère les options possibles lors de la compilation.

Une fois ces fichiers créés, il ne reste qu'à modifier le fichier de configuration général de *GCC* qui est nommé :

- config.gcc

2.2.4. Le métalangage RTL

Le langage de transfert de registre (RTL) a deux significations en informatique. La première est une représentation intermédiaire employée par le compilateur de *GCC*. La seconde se rapporte également à un langage qui définit avec précision ce que chaque instruction dans un processeur fait, à un niveau du détail qui permet la synthèse du matériel.
L'acronyme RTL est également employé pour le niveau de transfert de registre, c'est un attribut d'un langage de description matériel. RTL est employé pour représenter le code compilé sous une forme plus près de langage assembleur par rapport au langage de plus haut niveau que *GCC* compile.

RTL est issu de la représentation d'arbre de syntaxe abstraite, modifié par divers passages dans le '*middle-end*' de *GCC*. Puis Il est converti en langage assembleur pour la cible préalablement choisie. *GCC* emploie actuellement la forme RTL pour effectuer une partie de son travail d'optimisation. RTL est habituellement écrit sous une forme qui ressemble à une expression de LISP:

Une liste RTL est composée de 5 types d'objets :

- expressions,
- entiers,
- entiers larges,
- chaînes de caractères,
- les vecteurs

Les entiers sont à comprendre au sens des types du C (int), c'est à dire des nombres entiers.

Les entiers larges sont également à comprendre au sens des types de C, mais ils ont une taille supérieure aux entiers précédents.

Les chaînes de caractères sont identiques aux char * du C. Mais en RTL, une chaîne de caractères ne peut être nulle (une chaîne vide est considérée comme un pointeur null : char temp[10]= ""; équivaut à char * temp=null;)

Les vecteurs contiennent un nombre arbitraire d'expressions. Et comme pour les chaînes de caractères, un vecteur avec un nombre nul d'éléments n'est pas possible. Il correspond dans ce cas là à un pointeur nul.

Les expressions RTL notée RTX sont des structures C, mais sont généralement des pointeurs sur des typedef nommés rtx.

Les expressions sont classées par leurs codes (notés RTX pour "RTL eXpression"). Les codes que l'on a à disposition sont énumérés dans le fichier rtl.def. Ils sont équivalents à des énumérations du C. De plus, les codes sont indépendants de la cible. Des macros de manipulation de codes sont disponibles :

GET_CODE (x) pour l'extraction.

PUT_Code (x, newcode) pour l'insertion de nouveau code.

Le code RTX détermine le nombre et le type des opérandes qui sont rattachés à l'expression. Un de ces inconvénients est que, contrairement au Lisp, nous ne pouvons pas déterminer la nature des opérandes d'un simple coup d'oeil. Il faut d'abord déterminer le code RTX, puis en déduire leur nature.

Voici un exemple d'un code RTX :

```
(set:SI (reg:SI 0) (plus:SI (reg:SI 1) (reg:SI 2)))
```

Code 1 : Exemple de code RTL

Cette expression représente l'addition du registre 1 au registre 2, et le stockage du résultat dans le registre 0.

Les expressions des cibles sont disponibles dans les fichiers ".md"

Le RTL produit par *GCC* est unique pour chaque processeur cible. Cependant, la signification du RTL est plus ou moins indépendante de la cible : il serait habituellement possible de lire et comprendre un morceau de RTL sans savoir pour quel processeur il a été produit. De même, la signification du RTL ne dépend pas habituellement du langage de haut niveau original du programme.

2.3. Binutils

Les *binutils* sont une collection d'outils de programmation développés sous licence libre (Projet GNU) pour la manipulation du code d'objet dans divers formats de fichier. Ils sont typiquement employés avec *GCC* et GDB.

A l'origine, la collection *binutils* était composée seulement de petits utilitaires, mais plus tard l'assembleur de GNU (GAS) et l'éditeur de liens de GNU (GLD) ont été inclus dans le package, puisque leurs fonctionnalités sont étroitement liées. La plupart des *binutils* sont des programmes plutôt simples. La majeure partie de la complexité est encapsulée dans les bibliothèques de BFD et de libopcodes qu'elles partagent.

2.3.1. Présentation

Le contenu du package est le suivant :
Programmes installés: addr2line, ar, as, c++filt, gprof, ld, nm, objcopy, objdump, ranlib, readelf, size, strings et strip
Bibliothèques installées : libiberty.a, libbfd.[a,so] et libopcodes.[a,so]

Voici une courte description des programmes :

addr2line traduit les adresses de programme en noms de fichier et numéros de ligne. Suivant une adresse et le nom d'un exécutable, il utilise les informations de déboguage disponibles dans l'exécutable pour trouver le fichier source et le numéro de ligne associés à cette adresse.

ar crée, modifie et extrait des archives. Une archive est un simple fichier contenant une collection d'autres fichiers dans une structure qui rend possible la récupération des fichiers originaux individuels (aussi appelés membres de l'archive).

as est un assembleur. Il assemble la sortie de *GCC* en fichiers objet.

c++filt est utilisé par l'éditeur de liens pour récupérer les symboles C++ et Java, pour empêcher les fonctions surchargées d'arrêter brutalement le programme.

gprof affiche les données de profilage d'appels dans un graphe.

ld est un éditeur de liens. Il combine un certain nombre d'objets et de fichiers archives dans un seul fichier, en déplaçant leurs données et en regroupant les références de symboles.

nm liste les symboles disponibles dans un fichier objet.

objcopy est utilisé pour traduire un type de fichier objet en un autre type.

objdump affiche des informations sur le fichier objet donné, les options contrôlant les informations à afficher. L'information affichée est surtout utile aux programmeurs qui travaillent sur les outils de compilation.

ranlib génère un index du contenu d'une archive et le stocke dans l'archive. L'index liste tous les symboles définis par les membres de l'archive qui sont des fichiers objet déplaçables.

readelf affiche des informations sur les binaires de type elf.

size liste les tailles de section et le total pour les fichiers objets donnés.

strings affiche, pour chaque fichier donné, la séquence de caractères affichables qui sont d'au moins la taille spécifiée (par défaut, 4). Pour les fichiers objets, il affiche, par défaut, seulement les chaînes des sections d'initialisation et de chargement. Pour les autres types de fichiers, il parcourt le fichier entier.

strip supprime les symboles des fichiers objets.

Afin de fournir les *binutils* de *SIMPLE-CPU*, il faut donc les modifier par l'intermédiaire des fichiers *opcode*. Ces fichiers seront décrits ultérieurement.
Après avoir brièvement analyser les fichiers de la cible de comparaison (à savoir le Fujitsu FR30), nous constatons qu'ils portent tous l'entête d'un programme *CGEN*. Ce dernier semble les avoir généré automatiquement. Nous orientons donc notre recherche vers *CGEN* et ces différents fichiers.

2.3.2. Les fichiers de description et Cgen

CGEN est un *framework* ou « cadriciel » développant des générateurs d'outils tels que des assembleurs, des désassembleurs et des simulateurs pour différents processeurs. Il implémente un langage de description pour décrire l'architecture et l'organisation d'une CPU sans référencer n'importe quelle particularité. *CGEN* est écrit en *Scheme* et peut être exécuté sous l'interpréteur GNU *Guile*. Il est placé sous licence de logiciel libre.

Donc *CGEN* nous génère les fichiers *opcodes* vus précédemment. Pour cela, nous devons lui fournir des fichiers contenant toutes les caractéristiques de *SIMPLE-CPU*. Ces fichiers sont nommés '<arch>.cpu' et '<arch>.opc'

Le fichier '<arch>.cpu' contient toutes les descriptions du processeur cible. Il est écrit en *Scheme* (comme les fichiers de configuration de *GCC*).

Comme le représente le schéma suivant, un fichier '<arch>.cpu' implémente une architecture bien précise. Puis pour cette architecture, on peut définir plusieurs familles de CPU, et de modèle de CPU. Voici la représentation arborescente que l'on peut en faire.

Arborescence générale	Arborescence du FR30	Arborescence du FR30
architecture / \\ cpu-family1 cpu-family2 .. / \\ machine1 machine2 ... / \\ model1 model2 ...	Fujitsu FR30 / \\ fr30bf ... / \\ fr30bf ... / \\ fr30-1 ...	SimpleCPU / \\ SC91_32 SC91_64 | SC91 / \\ SC91_32_A SC91_32_B

Figure 2 : Arborescence des processeurs

Voici les différents patrons des structures et leurs correspondances pour le FR30.

Patrons des structures	Structures du FR30	Structures de *SIMPLE-CPU*

`(define-arch`
`  (name architecture-name)`
`  (comment "description")`
`  (attrs attribute-list)`
`  (default-alignment aligned|unaligned|forced)`
`  (insn-lsb0? #f|#t)`
`  (machs mach-name-list)`
`  (isas isa-name-list)`
`)`
	`(define-arch`
	`  (name fr30)`
	`  (comment "Fujitsu FR30")`
	`  (default-alignment forced)`
	`  (insn-lsb0? #f)`
	`  (machs fr30)`
	`  (isas fr30)`
	`)`
		`(define-arch`
		`  (name SimpleCPU)`
		`  (comment "Simple-CPU")`
		`  (default-alignment forced)`
		`  (insn-lsb0? #f)`
		`  (machs SC91)`
		`)`

`(define-cpu`
`  (name cpu-name)`
`  (comment "description")`
`  (attrs attribute-list)`
`  (endian big|little|either)`
`  (insn-endian big|little|either)`
`  (data-endian big|little|either)`
`  (float-endian big|little|either)`
`  (word-bitsize n)`
`  (parallel-insns n)`
`  (file-transform transformation)`
`)`
	`(define-cpu`
	
	`  (name fr30bf)`
	`  (comment "Fujitsu FR30 base family")`
	`  (endian big)`
	`  (word-bitsize 32)`
	`)`
		`(define-cpu`
		
		`  (name SC91_32)`
		`  (comment "Simple-CPU 32b")`
		`  (endian big)`
		`  (word-bitsize 32)`
		`)`

`(define-mach`
`  (name mach-name)`
`  (comment "description")`
`  (attrs attribute-list)`
`  (cpu cpu-family-name)`
`  (bfd-name "bfd-name")`
`  (isas isa-name-list)`
`)`
	`(define-mach`
	`  (name fr30)`
	`  (comment "Generic FR30 cpu")`
	`  (cpu fr30bf)`
	`)`
		`(define-mach`
		`  (name SC91)`
		`  (comment "Simple-CPU Machine")`
		`  (cpu SC91_32)`
		`)`

`(define-model`
`  (name model-name)`
`  (comment "description")`
`  (attrs attribute-list)`
`  (mach machine-name)`
`  (state`
`  (variable-name-1 variable-mode-1)`
`  ( ... )`
`  )`
`  (unit name "comment" (attributes))`
`)`
	`(define-model`
	`  (name fr30-1)`
	`  (comment "fr30-1")`
	`  (attrs)`
	`  (mach fr30)`
	`  (pipeline all ""`
	`  ()`
	`  ((fetch)`
	`  (decode)`
	`  (execute)`
	`  (writeback)`
	`  )`
	`  )`
	`)`
		`(define-model`
		`  (name SC91_32_A)`
		`  (comment "Simple-CPU 32b evolution A")`
		`  (attrs)`
		`  (mach SC91)`
		`)`

Figure 3 : Structures RTL

Dans ce fichier, nous trouvons d'autres informations tels que les décompositions de la ligne d'instructions. Grâce à ceci, les programmes assembleur et désassembleur peuvent convertir une instruction de mnémoniques en code machine et vice versa.

Voici un exemple explicatif sur les instructions d'additions de *SIMPLE-CPU*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction
Code instruction								Destination					Opérande 1					Opérande 2															
1	0	1				Value		Rdd					Rnn					Value														ADD r
1	0	0	0	0	0	0	0	Rdd					Rnn					Rmm			Mode											ADD d

```
(dnf Code        "Code Instruction"      () 31  24)
(dnf Rd          "Registre Rdd"          () 19  23)
(dnf Rn          "Registre Rnn"          () 18  14)
(dnf Rm          "Registre Rmm"          () 13   9)
(dnf Mode        "Mode de l'addition"    ()  8   6)
(dnf C_Value     "Code Value"            () 25  24)
(dnf Value       "Value"                 () 13   0)
```

Figure 4 : Définition de l'instruction ADDd

Nous devons effectuer ce travail pour toutes les instructions du coeur. Ainsi l'instruction ADDd peut être construite comme suit : Code+Rd+Rn+Rm+Mode (le signe '+' est à comprendre au sens concaténation et non pas au sens arithmétique ou logique). S'il y a des interruptions dans la séquence, le programme complétera avec des 0.

Un autre fichier peut être présent. Il s'agit de '<arch>.opc'. Il contient du code C spécifique à la cible. Il sert à construire de façon plus aisée les instructions complexes que supporte les coeurs.

Une fois ces fichiers créés, nous pouvons lancer *CGEN*. L'exécution se fait à partir de l'interpréteur Guile. Pour le fonctionnement de l'interpréteur, une documentation est disponible sur le site officiel du projet GNU.

Les fichiers générés par *CGEN* sont appelés fichiers *opcode.* Voici une brève présentation :

- '<arch>-desc.h'

Définitions des macros, des énumérations, des types qui servent à la description de la cible.

- '<arch>-desc.c'

Tableau de description de la cible. Il ne contient pas de syntaxe assembleur, ni d'information sémantique.

- '<arch>-ibld.c'

Routines pour assembler ou désassembler les instructions.

- '<arch>-opc.h'

Déclarations necessaries pour l'assembleur/désassembleur qui ne sont pas utilise ailleurs et donc laissé hors de `<arch>-desc.h'.

- '<arch>-opc.c'

Tableaux des syntaxes assembleur.

- '<arch>-asm.c'

Routines assembleur.

- '<arch>-dis.c'

Routines de désassemblage.

- '<arch>-opinst.c'

Tableaux des instances des opérandes. Cette description précise pour chaque instruction quels éléments sont lus et lesquels sont écrits. Il implémente également les systèmes de contrôle pour le parallélisme.

Étant donné que ces fichiers sont générés automatiquement par *CGEN*, nous ne les avons pas étudié dans le détail. Cependant, pour confirmer l'orientation choisie, nous verrons ultérieurement que nous modifierons l'un de ces fichiers pour analyser les répercutions.

3. Réalisation

3.1. Etude préliminaire

La première étape de l'élaboration de solution fut de fixer les outils utilisés le long du projet. En effet la réactivité de la communauté du logiciel libre fait que les versions des programmes se succèdent à rythmes soutenus, amenant améliorations et modifications. Mais certaines versions récentes peuvent présenter des défaillances. C'est la raison pour laquelle la version de *GCC* utilisée est la version 3.4.5 alors que la dernière *release* est la version 4.1.0 (du 28 Février 2006). De la même manière, la version des *binutils* utilisée est la version 2.16.1. Et celle de *cgen* est la 1.0.

3.1.1. Orientations possibles

Comme décrit précédemment plusieurs orientations sont possibles pour effectuer le portage d'un compilateur sur une machine particulière.

Tout d'abord on pourra considérer le codage d'un compilateur de toute pièce en utilisant les outils d'analyse. Cette approche bien qu'extrêmement formatrice, car reposant sur les bases de l'enseignement ISIMA, à pour inconvénient un temps de développement extrêmement long puisque le travail doit être mené à partir de rien. Or il existe déjà nombre de compilateurs C.

La deuxième grande orientation revient donc à utiliser un compilateur déjà présent et à l'adapter pour le processeur *SIMPLE-CPU*. Or la philosophie du projet est basée sur le développement collaboratif. Le choix de *GCC* s'est donc imposé de lui-même.

Pour adapter *GCC* il existe plusieurs possibilités, déjà détaillées précédemment.

La première consiste à modifier le code source de *GCC* pour l'adapter à la cible voulue, à savoir le processeur *SIMPLE-CPU*. Mais avec cette méthode la philosophie même de *GCC* est perdue. Cette possibilité a rapidement été écartée du fait de la complexité du code de *GCC*, mais également du manque de documentation de celui-ci.

L'utilisation de la versatilité de *GCC* semble être la meilleure possibilité. En effet la philosophie de *GCC* est d'être le plus indépendant possible tant des langages sources que des machines de destinations. Des fonctionnalités pour étendre le champ d'action de *GCC* y ont été incorporées et il est donc « facilement » possible d'ajouter le support de nouveaux langages source, avec l'ajout de *front-end*, ou de machine cibles, avec l'ajout de *back-end*. C'est bien l'ajout d'un *back-end SIMPLE-CPU* qui nous intéresse dans le cadre de ce projet.

3.1.2. Orientation choisies

L'orientation donnée au projet est d'utiliser l'ensemble des composants mis à disposition que ce soit *GCC* ou les *binutils*. En effet une telle approche permet de s'appuyer sur des méthodes éprouvées et des outils performants. De plus le gain en temps est

appréciable du fait des différents exemples donnés dans les versions de *GCC* et/ou *binutils*. En effet nombre de cibles sont supportées par *GCC*. Comme les codes sources sont distribués en même temps que les fichiers binaires, il est aisé de pouvoir étudier le travail réalisé pour telle ou telle architecture. C'est ainsi que le choix s'est porté sur l'étude plus particulière du processeur FR30 de Fujitsu.

3.1.3. Le FR30

Afin de partir d'une base solide pour la création des fichiers de description, c'est le coeur FR30 de Fujitsu qui a été préféré. Celui-ci a en effet été intégré aux distributions de *GCC*. De plus son architecture interne est très proche de celle de simple CPU.

	FR30	Simple CPU
Architecture	RISC 32 bits	RISC 32 bits
Registres	16 x 32 bits	32 x 32 bits
Instructions	165	21

Figure 5 : Tableau comparatif des architectures Simple-CPU / FR30

On remarquera que le nombre d'instructions diffère entre les deux coeurs. Cela provient du fait de la méthode de comptage du nombre d'instructions. En effet, sur Simple CPU on considère comme une instruction un unique mnémonique. Or pour chacun de ces mnémoniques le mode de fonctionnement est contenu dans les opérandes. Ainsi de 21 opérations "affichées" on passe facilement à plus d'une centaine d'instructions "réelles".
Dans le FR30, il compte aussi les alias ce que l'on ne fait pas pour le simple CPU.

Le choix du processeur cible étant effectué, on considère les fichiers fournis dans la distribution de *GCC* pour les adapter à Simple CPU. A savoir :

- crti.asm
- crtn.asm
- fr30.c
- fr30.h
- fr30.md
- fr30-protos.h
- lib1funcs.asm
- t-fr30

3.2. Compilation de GCC pour FR30

Comme précisé précédemment, la version de *GCC* a été fixée afin de ne pas avoir de problème de compatibilité dus aux différentes versions successives. La partie suivante présente la compilation de *GCC* pour le processeur FR30.

- Dans un premier temps, trois dossiers de travail sont créés (Figure 6 : Arborescence créée)

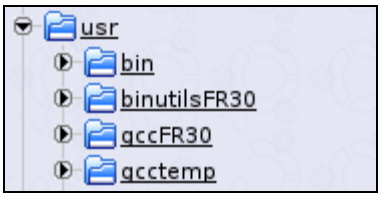


Figure 6 : Arborescence créée

➢ /usr/gccFR30/
Contient gcc-3-4-5.

➢ /usr/binutilsFR30/
Contient les *binutils* en version 2-6-10.

➢ /usr/gcctemp/
Contiendra les binaires compilés.

- Les commandes utilisées pour la compilation sont les suivantes :

```
root@xdoofy:/usr/binutilsFR30# ./binutils-2.16.1/configure
                    --target=fr30-fujitsu-elf
                    --prefix=/usr/gcctemp

root@xdoofy:/usr/binutilsFR30# make
root@xdoofy:/usr/binutilsFR30# make install

root@xdoofy:/usr/gccFR30# ./gcc-3.4.5/configure
                    --prefix=/usr/gcctemp
                    --enable-languages="c"
                    --target=fr30-fujitsu-elf

root@xdoofy:/usr/gccFR30# export PATH=$PATH:/usr/gcctemp/bin/

root@xdoofy:/usr/gccFR30# make
root@xdoofy:/usr/gccFR30# make install
```

Figure 7 : Commandes de compilation de GCC pour FR30

3.3. Modification des fichiers de descriptions

Afin de confirmer la possibilité de réalisation du projet, nous avons voulu compiler *GCC* en modifiant une seule instruction pour notre cible de référence. Donc dans cette partie, <arch> correspond au FR30 de Fujitsu. Pour cela, nous modifions directement les fichiers *opcodes*. Cette procédure aurait pu se faire en modifiant le fichiers '<arch>.cpu'. Mais pour des raisons de simplicité, nous modifierons directement les fichiers générés.

Nous allons présenter les 4 grands points nécessaires à la réalisation.

1. Modification des fichiers *opcode*.
2. Compilation des *binutils*.
3. Modification du fichier de description '<arch>.md' de *GCC*.
4. Compilation finale de *GCC*.

La modification des fichiers *opcode* se fait par l'intermédiaire de '<arch>-desc.c'. Nous déciderons arbitrairement de modifier la multiplication.

Les fonctions sont représentées par le type `CGEN_IBASE`. Ces représentations sont structurées sous formes de tableaux.

```
typedef struct
{
  int num;
  const char *name;
  const char *mnemonic;
  int bitsize;
  CGEN_INSN_ATTR_TYPE attr;
} CGEN_IBASE;
```

Code 2 : Structure des opcodes

num : numéro énumérée des instructions
name: Nom de l'instruction
mnemonic: mnemonique apparaissant dans le fichier '.s'.
bitsize: longueur totale de l'instruction en bit.
attr: type des attributs de la fonction.

Voici la fonction originale (celle du FR30 ligne 766)

```
/* mul $Rj,$Ri originale */
  {
    FR30_INSN_MUL, "mul", "mul", 16,
    { 0|A(NOT_IN_DELAY_SLOT), { (1<<MACH_BASE) } }
  },
```

Code 3 : Opcode de l'instruction 'mul' FR30

Nous pouvons constater que la fonction est encapsulée par des accolades et des virgules car *CGEN* a créé un tableau de fonctions.

Voici maintenant la fonction modifiée.

```
 /* mul $Rj,$Ri modifiée*/
  {
    FR30_INSN_MUL, "ISIMA", "ISIMA", 16,
    { 0|A(NOT_IN_DELAY_SLOT), { (1<<MACH_BASE) } }
  },
```

Code 4 : Opcode de l'instruction 'ISIMA' modifiée.

Ainsi, nous avons uniquement changé le mnémonique de la multiplication. Nous avons donné le même nom pour faciliter le travail et éviter les confusions entre nom et mnémonique. Le numéro de la fonction reste identique à savoir la valeur énumérée FR30_INSN_MUL.

La deuxième étape consiste à compiler les sources des *binutils*. Ci après nous indiquons les commandes à exécuter. Attention, il faut respecter la configuration des dossiers. Le temps de compilation peut être de 10 min à 1 heure en fonction de la puissance de l'ordinateur.

```
root@xdoofy:/usr/binutilsFR30# ./binutils-2.16.1/configure
                               --target=fr30-fujitsu-elf
                               --prefix=/usr/gcctemp

root@xdoofy:/usr/binutilsFR30# make
root@xdoofy:/usr/binutilsFR30# make install
```

Figure 8 : Commande de compilation des binutils.

Pendant que l'ordinateur compile les *binutils*, nous pouvons modifier les fichiers de description de *GCC* à savoir 'gcc/config/<arch>/<arch>.md'.

Nous rappelons que les informations contenues dans ces fichiers sont au format Lisp. Après avoir identifier le vecteur d'expressions correspondant à la multiplication.

Voici la fonction originale (celle du FR30 ligne 788)

```
;; Signed multiplication producing 32 bit result from 32 bit inputs
(define_insn "mulsi3"
  [(set (match_operand:SI 0 "register_operand"             "=r")
          (mult:SI (match_operand:SI 1 "register_operand" "%r")
                   (match_operand:SI 2 "register_operand"  "r")))
   (clobber (reg:CC 16))]
  ""
  "mul  %2, %1\\n\\tmov\\tmdl, %0"
  [(set_attr "length" "4")]
)
```

Code 5 : Pattern RTL de l'instruction 'mul' FR30.

Maintenant, modifions le mnémonique *mul* par *ISIMA*.

```
;; Signed multiplication producing 32 bit result from 32 bit inputs
(define_insn "mulsi3"
  [(set (match_operand:SI 0 "register_operand"             "=r")
          (mult:SI (match_operand:SI 1 "register_operand" "%r")
                   (match_operand:SI 2 "register_operand"  "r")))
   (clobber (reg:CC 16))]
  ""
  "ISIMA  %2, %1\\n\\tmov\\tmdl, %0"
  [(set_attr "length" "4")]
)
```

Code 6 : Pattern RTL modifié pour l'instructiuon 'ISIMA'

Après cette modification, et une fois la compilation précédente achevée, nous pouvons compiler *GCC* à proprement dit. Pour cela, nous exécutons les commandes suivantes :

```
root@xdoofy:/usr/gccFR30# ./gcc-3.4.5/configure
                          --prefix=/usr/gcctemp
                          --enable-languages="c"
                          --target=fr30-fujitsu-elf
```

Figure 9 : Configuuration du make de GCC

Une fois la configuration faite, il faut penser à modifier le *PATH* en incluant le chemin d'accès du dossier qui contient les exécutables créés lors de la compilation des *binutils*.

```
root@xdoofy:/usr/gccFR30# export PATH=$PATH:/usr/gcctemp/bin/
```

Figure 10 : Ajout dans le PATH

Ensuite, il ne reste plus qu'à créer l'exécutable *GCC*.

```
root@xdoofy:/usr/gccFR30# make
root@xdoofy:/usr/gccFR30# make install
```

Figure 11 : Compilation et installation de GCC

Maintenant, nous allons tester notre nouveau compilateur. Pour cela, nous allons créer un code C utilisant la multiplication.

```
root@xdoofy:/usr/exemple# Fujitsu-FR30-gcc-3.4.5 -s test.c
```

Figure 12 : Compilation d'un fichier d'exemple avec le nouveau compilateur.

D'un premier jet, nous avons testé le code suivant :

```
void main (void)
{
        return 32*10;
}
```

Code 7 : Code source de test 1

A notre surprise, le code ne contenait aucune multiplication. Puis, nous avons déduit que le *middle-end* de *GCC* devait optimiser et transformer le code.

```
void main (void)
{
        return 320;
}
```

Code 8 : Code source de test après passe d'optimisation

Donc, nous avons ensuite codé un programme un peu plus complexe afin de que *GCC* ne puisse pas ou peu optimisé le code. Nous pourrions ainsi visualiser le résultat de nos modifications.

```
int main(void){
int a=323;
int b=467653;
return a*b;}
```

Code 9 : Code source de test v.2

Voici le résultat de la manipulation :

```
       .file  "test.c"
       .text
       .p2align 2
       .globl main
       .type  main, @function
main:
       enter  #12
       ldi:8  #252, r1
       extsb  r1
       mov    fp, r2
       addn   r1, r2
       ldi:20 #323, r1
       st     r1, @r2
       ldi:8  #248, r1
       extsb  r1
       mov    fp, r2
       addn   r1, r2
       ldi:20 #467653, r1
       st     r1, @r2
       ldi:8  #252, r1
       extsb  r1
       mov    fp, r2
       addn   r1, r2
       ldi:8  #248, r1
       extsb  r1
       addn   fp, r1
       ld     @r2, r2
       ld     @r1, r1
       mul    r1, r2
       mov    mdl, r1
       mov    r1, r4
       leave
       ret
       .size  main, .-main
       .ident "GCC: (GNU) 3.4.5"
```

Code 10 : Code assembleur avant modification

```
       .file  "test.c"
       .text
       .p2align 2
       .globl main
       .type  main, @function
main:
       enter  #12
       ldi:8  #252, r1
       extsb  r1
       mov    fp, r2
       addn   r1, r2
       ldi:20 #323, r1
       st     r1, @r2
       ldi:8  #248, r1
       extsb  r1
       mov    fp, r2
       addn   r1, r2
       ldi:20 #467653, r1
       st     r1, @r2
       ldi:8  #252, r1
       extsb  r1
       mov    fp, r2
       addn   r1, r2
       ldi:8  #248, r1
       extsb  r1
       addn   fp, r1
       ld     @r2, r2
       ld     @r1, r1
       ISIMA         r1, r2
       mov    mdl, r1
       mov    r1, r4
       leave
       ret
       .size  main, .-main
       .ident "GCC: (GNU) 3.4.5"
```

Code 11 : Code assembleur après modification

Maintenant que nous avons modifié une instruction, nous pouvons estimer la charge horaire afin de réaliser le portage de *GCC* et des *binutils* pour *SIMPLE-CPU*. Nous avons fixé un maximum de 10 min par ligne de code. Dans ces 10 minutes, nous comptons, bien évidemment, la réflexion, la réalisation et la vérification. Sachant également que les lignes de codes sont répétitives, surtout lors des descriptions des fonctions, donc dans ce cas, le temps de réflexion diminue considérablement. La vérification peut se faire de façon groupée.

Pour les fichiers '<arch>.cpu' et '<arch>.opc', nous estimons le temps maximum à 350H, mais après avoir diminué le temps dû aux tâches répétitive, nous fixons ce délai à 250H.

Pour les fichiers *back-end* de *GCC*, nous avons calculé un temps d'environ 650H. Ce temps peut paraître excessif, mais il faut savoir que les fichiers de descriptions *GCC* sont plus nombreux, et également qu'il n'existe à l'heure actuelle aucun outil qui permette la génération automatique. Cependant, sur le site officiel de l'outil *CGEN*, nous avons lu que les développeurs ont l'ambition de créer une nouvelle version qui générera automatiquement les fichiers de descriptions *binutils* et de *GCC*. Ainsi, le portage de *GCC* vers une cible sera plus aisé. Pour le moment, nous devons coder ces fichiers "à la main", mais il faut savoir qu'ils

sont extrêmement répétitifs. Donc une fois que l'on a compris comment les coder, le temps prévu pour les suivantes diminue, c'est pourquoi nous avons revu ce temps à la baisse. Nous évaluons donc ce temps à 400H.

Donc nous pensons que le portage de *GCC* vers *SIMPLE-CPU* peut s'effectuer en un temps d'environ 650H. Ce temps tient également compte de la connaissance que nous avons acquise tout au long de ce rapport.

4. Conclusion

Bien que le but initial du projet ne soit pas atteint, le travail effectué a permis de fournir, une documentation la plus compète possible sur les étapes du portage de *GCC* sur le processeur *SIMPLE-CPU*, des exemples, et une estimation du temps nécessaire pour mener à bien le projet. En effet les objectifs ont été changés durant le déroulement du projet puisque la première étape devait être l'étude de faisabilité. Mais la charge de travail pour effectuer cette étude dans de bonne condition fut telle que l'objectif fut dès lors de fournir une étude de faisabilité la plus complète possible, comme c'est le cas avec ce rapport et les documents annexes.

L'emploi du temps du projet est présenté (Figure 13 : Emploi du temps du projet). Il se divise en trois grandes parties non distinctes. Une première partie concernant l'étude de *GCC*, de son fonctionnement interne et des *back-end.* Lorsque les problèmes de compilation se sont présentés avec *GCC*, et que le lien a été fait avec les *binutils*, le projet s'est orienté vers cette piste afin de cerner l'ensemble du problème et ne pas se cantonner à l'étude simple de *GCC*. Durant cette partie concernant les *binutils*, l'outil *cgen* a été pointé et étudié afin de voir comment générer des fichiers de description de machine pour les *binutils*. Dans le même temps, le travail sur *GCC* continuait en apportant de nouvelles avancées, en particulier la modification d'instructions réussie. La dernière partie du projet fut plus consacrée à la préparation des documents à savoir le rapport, la soutenance de projet et le site web. Ces matériaux servant évidemment à l'étude menée.

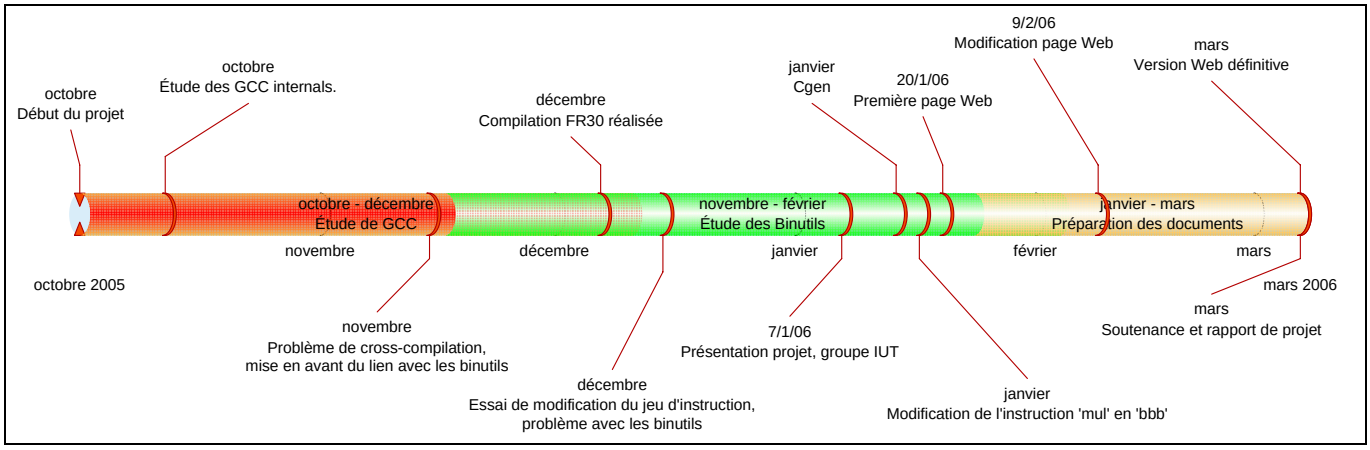


Figure 13 : Emploi du temps du projet

D'un point de vue pédagogique, ce projet nous a permis de nous confronter à différentes difficultés réelles du travail d'ingénieur. En effet, que ce soit à cause du manque de documentation, ou de la confrontation à de concepts nouveaux, ce travail a été jalonné de difficultés. De plus, le travail de recherche et de compréhension plus que de « codage », et par là le côté plus théorique que pratique, est également une difficulté à surmonter, mais qui fait bien partie du travail de l'ingénieur. Il semble en effet que concevoir un processeur RISC 32bits soit plus facile pour les étudiant de l'ISIMA que de concevoir le compilateur C qui vient dessus. Ce projet s'inscrit pourtant bien dans la démarche qu'un ingénieur doit mener pour ses projets, à savoir l'étude de possibilités et le choix motivé de certaines.

Références

Bibliographie :

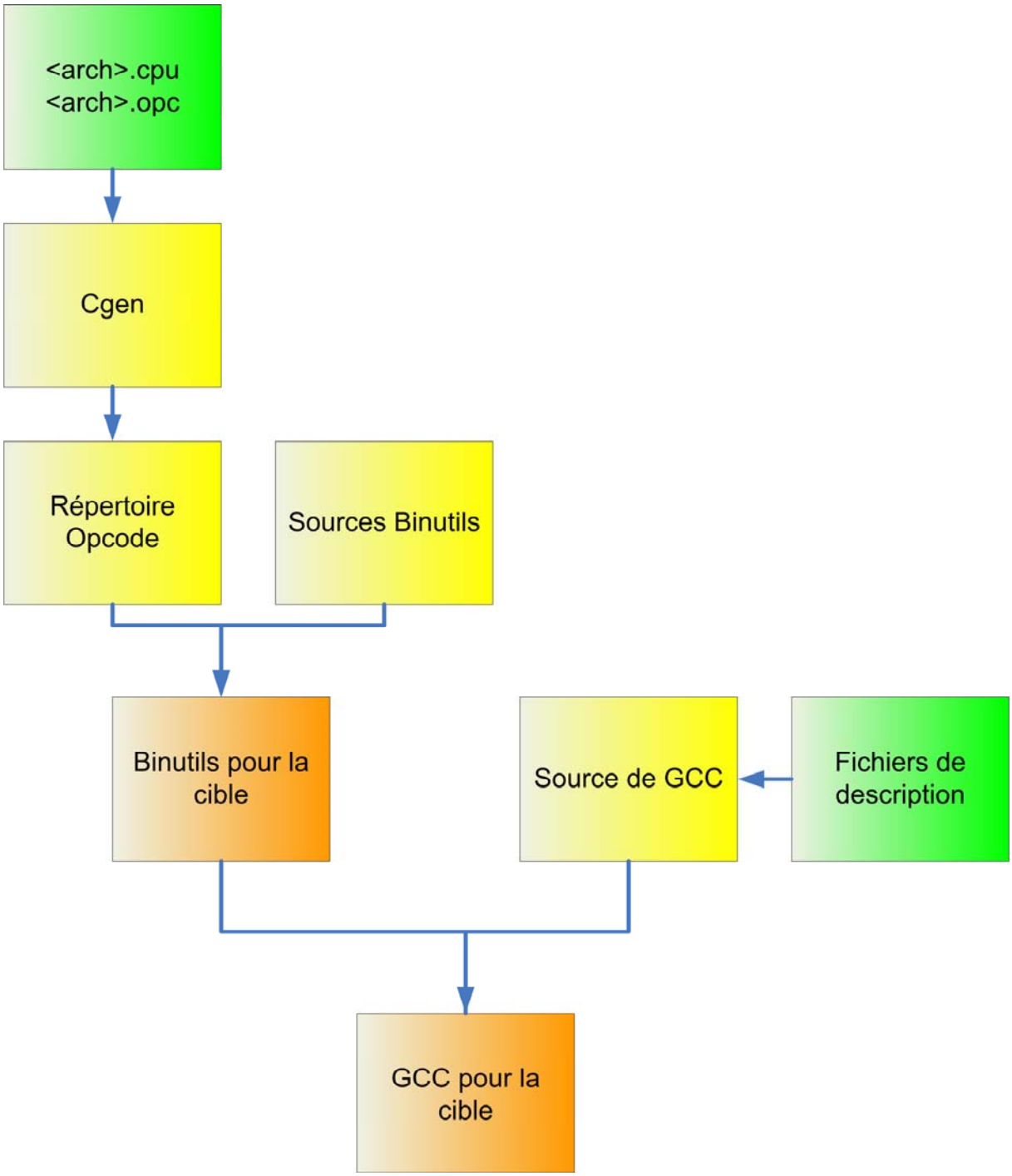
[1] “GNU Compiler Collection Internals (GCC)” *Richard M. Stallman and the GCC Developer Community*
[2] “FR30 series datasheet” *Fujitsu semiconductors*
[3] “SIMPLE-CPU INSTRUCTION SET SC91-A” *Disk91*

Liens utiles :

[WEB1] *www.simple-cpu.com*
[WEB2] *http://gcc.gnu.org*
[WEB3] *http://sources.redhat.com/cgen*
[WEB4] *http://fr.wikipedia.org/wiki/Wiki*
[WEB5] *http://www.gnu.org/software/guile/guile.html*

Annexes

Le schéma présente une vue de l'ensemble des programmes et fichiers rentrant en jeu pour l'élaboration du compilateur GCC pour une cible particulière. On remarque en vert les fichiers à modifier, en jaune les fichiers et/ou programmes intermédiaires et finalement les exécutables en orange.



Annexe 1 : Vue des fichiers et programmes