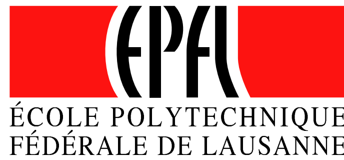

Projet de diplôme : Développement PXA250

Julien Pilet & Stéphane Magnenat

21 Février 2003
$Revision : 1.125 $

Laboratoire d'Architecture des Processeurs
Faculté d'Informatique et Communication
Professeur : Paolo Ienne
Responsable : René Beuchat

Table des matières

1 Introduction **7**
1.1 L'informatique embarquée 7
1.2 But du projet 7
1.3 A propos de ce document 8
1.3.1 Organisation 8
1.3.2 Conventions typographiques 8

2 Présentation du matériel **9**
2.1 Qu'est-ce qu'Armonie ? 9
2.2 Rappels techniques sur l'architecture ARM 11
2.3 Particularités du processeur PXA250 11
2.3.1 Les caches de mémoire 12
2.3.2 Fréquences et tensions d'alimentations 13
2.4 La carte EzUSB 13

3 Directions et choix **14**
3.1 Directions 14
3.2 Plateforme pédagogique 14
3.3 Carte processeur pour le robot cyclope 14
3.4 Outils de développement 15
3.5 Système multimédia embarqué 15
3.6 Collaboration extérieure 16

4 Utiliser Armonie **17**
4.1 Outils de développement 17
4.1.1 Jelie 17
4.1.2 Compilateur croisé 17
4.1.3 Debogueur 19
4.2 Plan mémoire du PXA 250 19
4.3 Initialisation 19
4.3.1 Initialisation du contrôleur de SDRAM 19
4.4 Utilisation de la mémoire flash 21
4.5 Charger un programme sans système d'exploitation 21
4.6 Changer les fréquences du processeur et du bus 23
4.7 Guide d'utilisation de Linux 25
4.7.1 Prérequis 25
4.7.2 Démarrage 25
4.7.3 Travailler depuis l'ordinateur de développement 25
4.7.4 Une distribution GNU/Linux complète 26
4.7.5 Contrôler l'écran LCD 26

5 Chaîne d'outils GNU **27**
5.1 Automatisation de la création des outils 27
5.2 GDB 28

6 Jelie **30**
6.1 Conception objet 30
6.2 Fonctions JTAG 31
6.3 Contrôleur port parallèle 32
6.4 Contrôleur USB 32
6.4.1 Détail du protocole 33
6.5 *Debug handler* 34
6.5.1 La table de vecteurs d'exceptions 35
6.5.2 *Special Debug State* 35
6.5.3 Communication entre l'hôte et le gestionnaire de débogage 36
6.6 GNU automake et autoconf 36
6.7 Convivialité et ligne de commande 38
6.8 Changement de fréquences 38
6.9 Améliorations possibles 38
6.9.1 Programmation de la mémoire flash 39
6.9.2 Contrôleurs JTAG 39

7 Linux sur Armonie **40**
7.1 La communauté Linux ARM 40
7.2 Le noyau Linux 40
7.2.1 Démarrage du noyau, plan mémoire et MMU 40
7.2.2 Sources et compilation 41
7.2.3 Les modifications apportées au noyau 42
7.2.4 Configuration et compilation 42
7.2.5 Le Mubus 43
7.2.6 Le bus I^2C 44
7.2.7 Le CompactFlash 44
7.2.8 L'écran LCD 44
7.2.9 Réseau via USB 45
7.3 Le système d'exploitation 45
7.3.1 Busybox 45
7.3.2 Démarrage 46

8 Système multimédia embarqué **47**
8.1 Présentation 47
8.2 Conception eMediaKit 48
8.2.1 Disposition physique 48
8.2.2 Contrôleur Ethernet 48
8.2.3 Puce AC97 50
8.2.4 Son 50
8.2.5 Ecran et surface tactile 50
8.2.6 Périphériques d'interaction homme-machine 50
8.2.7 Contrôleur maître USB 51

9 Consommation et performances **52**
9.1 Les tests de performances 52
9.1.1 Unité arithmétique et logique 53
9.1.2 Multiplications entières 53
9.1.3 Lectures en mémoire 54
9.1.4 Ecritures en mémoire 54
9.1.5 Lectures en cache 55

9.1.6 Ecritures en cache . . . 56
9.2 Consommation . . . 57
9.2.1 Consommation du coeur . . . 57
9.2.2 Consommation de l'alimentation 3.3 V . . . 57
9.2.3 Consommation totale . . . 58
9.3 Performances . . . 58
9.3.1 Performances . . . 58
9.3.2 Rapport performances / consommation . . . 58
9.3.3 Performances par rapport à un PC . . . 62
9.4 Conclusion . . . 62

10 A propos de ce projet 66
10.1 Difficultés . . . 66
10.1.1 Processeurs partiellement défectueux . . . 66
10.1.2 Débogage outils, logiciel et matériel . . . 66
10.1.3 Documentation floue ou erronée . . . 66
10.2 Outils utilisés et télé-travail . . . 67
10.2.1 Présentation de l'espace de travail . . . 67
10.2.2 Debian . . . 67
10.2.3 CVS, copies de sauvegarde et serveur de fichier . . . 67
10.2.4 Caméra . . . 68
10.2.5 Doxygen . . . 68
10.2.6 Outils utilisés pour le développement. . . . 68
10.2.7 Outils utilisés pour le rapport et la présentation. . . . 68
10.3 Continuations possibles . . . 68
10.3.1 Plateforme pédagogique . . . 69
10.3.2 Robot cyclope . . . 69
10.3.3 Système multimédia embarqué . . . 69
10.3.4 La gestion d'énergie sous Linux . . . 69
10.4 Conclusions . . . 69
10.5 Remerciements . . . 69

Table des figures

2.1 Face supérieure de notre carte, Armonie 10
2.2 Schéma bloc Armonie 10
2.3 Système Armonie complet à trois cartes 11
2.4 Exemple d'instruction ARM 11
2.5 Organisation du cache d'instructions du PXA250 (tiré de la page 4-1 du manuel de l'utilisateur [2]). Le cache de donnée est organisé de la même manière. 12

3.1 Robot cyclope 15

4.1 Chaîne d'outils de développement 18
4.2 Carte de l'espace mémoire d'Armonie. 20
4.3 Programme d'exemple qui utilise le Mubus sans système d'exploitation. 22
4.4 Programme d'initialisation de pile et d'appel de la fonction `main`. 23
4.5 Fenêtre de configuration de GDB pour la communication avec Jelie. 24
4.6 Environnement graphique de débogage d'un programme tournant sur Armonie sans système d'exploitation. 24

5.1 Comment GDB a accès au PXA250 grâce à Jelie. 28
5.2 Le fichier `tm.h` utilisé pour compiler GDB 5.2.1. 29

6.1 Structure interne de Jelie. 31
6.2 Deux manières d'envoyer des données par port JTAG : en relisant le contenu du registre ou en rafale. 32
6.3 Chaîne de transmission JTAG par USB 33
6.4 Diagramme de fonctionnement du gestionnaire de débogage. 35
6.5 Organisation des fichiers utilisés par automake et autoconf. 37

7.1 Exemple de script Jelie qui démarre Linux sur Armonie. 41
7.2 Plan mémoire au chargement de Linux 42
7.3 Linux sur Armonie 43
7.4 Liste des commandes supportées par busybox. 46

8.1 Dessin du système multimédia embarqué construit 47
8.2 Schéma bloc des circuits imprimés composant le système multimédia embarqué 48
8.3 Schéma bloc eMediaKit 48
8.4 Carte de l'espace mémoire d'Armonie avec eMediaKit. 49
8.5 Connection de l'écran 18 bits au Milli-BUS LCD 16 bits 51

9.1 Consommation du coeur 59
9.2 Consommation de l'alimentation 3.3V 60
9.3 Consommation totale d'Armonie 61
9.4 Comparaison des performances relatives 63
9.5 Performances relatives / consommation totale 64

9.6 Comparaison des vitesses entre le PXA250 et des Pentium III et IV 65

10.1 Environnement de travail 67

Liste des tableaux

2.1 Résumé des mémoires caches du PXA250. 12

4.1 Organisation des secteurs dans la mémoire flash 21

6.1 Table de vecteur d'exception ARM 34
6.2 Protocole de communication du gestionnaire de débogage vers Jelie. 36
6.3 Configurations de fréquences du PXA250 (tiré du tableau 3-1, page 3-5 du manuel du développeur [3]). 39

Liste des annexes

1. Manuel utilisateur de Jelie.
2. Manuel de référence du programmeur de Jelie.
3. Manuel du pilote Mubus pour Linux.
4. Spécification Milli-Bus.
5. Spécification A-Bus.
6. Rapport du projet de semestre précédent.
7. Schémas Armonie (version construite, pxa250main).
8. Schémas eMediaKit (version construite).
9. Errata des schémas.
10. Mesures de performances et de consommation.
11. Spécification Mubus
12. Schéma du connecteur JTAG

Chapitre 1

Introduction

1.1 L'informatique embarquée

L'informatique s'est grandement démocratisée au cours des vingt dernières années. L'invention de l'interface graphique a permis à une partie importante de la population, ayant peu ou pas de connaissances techniques, d'utiliser un ordinateur à la maison ou au travail, avec plus ou moins de succès. L'informatique de bureau une fois établie, c'est en direction de l'embarqué que les développements les plus importants ont été effectués.

Si les années 90 ont été celles de l'ordinateur personnel, les années 2000 seront celles des assistants numériques et téléphones portables multimédias. Ils intègrent beaucoup de fonctions, et sont à la fois agendas, téléphones, appareils de photos et ordinateurs complets. Pour répondre à cette nouvelle demande, les fabricants de processeurs conçoivent des modèles de plus en plus puissants, qui fonctionnent à basse consommation et intègrent un nombre important de périphériques. Notre projet et le processeur que nous utilisons, le PXA250 de la famille XScale d'Intel, s'inscrivent dans ce mouvement.

1.2 But du projet

Ce projet consiste à mettre en valeur et à rendre utilisable une carte développée lors de notre précédent projet de semestre. Cette carte, nommée Armonie, est un ordinateur à usages divers (robot mobile Cyclope, système multimédia embarqué et support pédagogique) avec un processeur de type ARM.

Au début du projet, la carte existait et fonctionnait, mais il n'y avait ni logiciel, ni système d'exploitation, ni d'outils de développement. De plus, bien qu'Armonie soit extensible, il n'existait pas d'extensions. Le but était d'effectuer les travaux suivants :

1. créer les outils de développement à partir des outils GNU, garantissant ainsi une solution de développement libre, fonctionnelle, gratuite et adaptée à nos besoins ;
2. porter Linux et écrire différents pilotes de périphériques, suivant le temps à disposition et nos besoins ;
3. développer une carte d'extension, apportant de nouvelles fonctionnalités. Ecrire le logiciel nécessaire suivant le temps disponible.

1.3 A propos de ce document

Ce document est le rapport de notre projet de diplôme. A ce titre, c'est un éclairage, une analyse et une synthèse du travail qui a été effectué. Mais nous aimerions aussi qu'il puisse servir de point de départ et de référence à de futurs projets, afin que le travail réalisé profite au maximum de personnes possible. C'est pourquoi une partie significative de ce rapport parle de nos outils et de notre méthode de développement.

Des annexes expliquent comment utiliser nos outils, sans rentrer dans les détails de leur fonctionnement interne. Ce document fait souvent référence aux annexes ou à de la documentation externe. Néanmoins, nous l'avons écrit de telle façon qu'il puisse être lu et compris sans recourir à ces références. Ces dernières seront par contre utiles pour tout développement se basant sur notre travail.

1.3.1 Organisation

Ce document est organisé en chapitres, suivant une logique bien définie :
- Le chapitre 2 présente le matériel utilisé, tant au niveau du processeur que de la carte ;
- Le chapitre 3 présente nos choix par rapport au projet ;
- Le chapitre 4 explique comment utiliser Armonie et les outils de développement ;
- Les chapitres 5, 6 et 7 détaillent le fonctionnement interne et la compilation des outils et de Linux ;
- Le chapitre 8 présente l'application de type système multimédia embarqué utilisant la carte d'extension eMediaKit ;
- Le chapitre 9 offre une analyse de la consommation et des performances d'Armonie ;
- Le chapitre 10 expose notre méthode de travail et conclut le document.

1.3.2 Conventions typographiques

Afin que ce document soit bien lisible, et que sa consultation soit agréable, nous avons utilisé les conventions typographiques suivantes :
- Les mots en anglais, sont en *italique*.
- Les noms de fichiers, de classes, de fonctions et les extraits de code sont écrits avec une police à `espacement fixe`.
- Les abréviations sont en PETITES MAJUSCULES.

Chapitre 2

Présentation du matériel

Ce chapitre présente le matériel utilisé pour ce projet : introduction de notre carte Armonie, rappel des particularités de l'architecture ARM, description du processeur utilisé et de la carte d'interface JTAG sur USB.

2.1 Qu'est-ce qu'Armonie ?

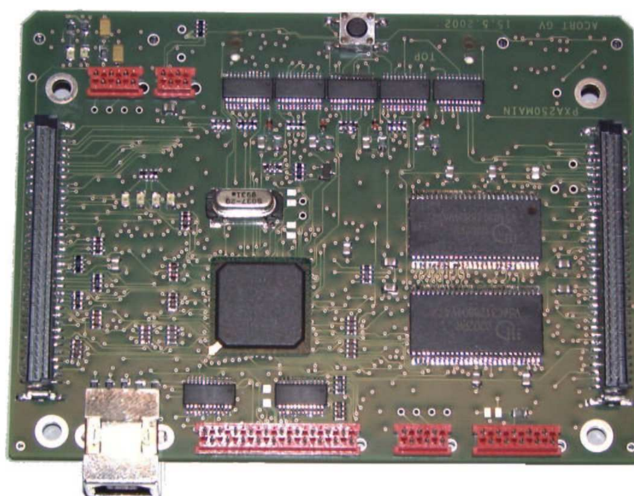

FIG. 2.1 – Face supérieure de notre carte, Armonie

Cette section décrit brièvement Armonie[1].

Armonie est une carte processeur que nous avons réalisée lors de notre projet de semestre. Elle a les particularités suivantes :

- Processeur Intel PXA250, à 200 ou 400 MHz (architecture ARM) ;
- 64 méga-octets de mémoire dynamique SDRAM compatible PC100 ;
- 1 méga-octets de mémoire flash ;
- Emplacement d'extension CompactFlash (CF) ;
- Bus d'alimentation A-BUS et d'extension Milli-BUS[2] ;
- Connecteur USB esclave ;
- Connecteurs série, I^2C, Mubus[3] et JTAG.

[1] Pour plus d'informations sur la carte Armonie, veuillez consulter le rapport de projet de semestre, disponible en annexe 6.

[2] Les spécifications A-BUS et Milli-BUS sont disponibles en annexe 5 et 4.

[3] Le Mubus est un petit bus de 8 bits de large qui a été developpé au LAMI et qui est encore utilisé au LAP.

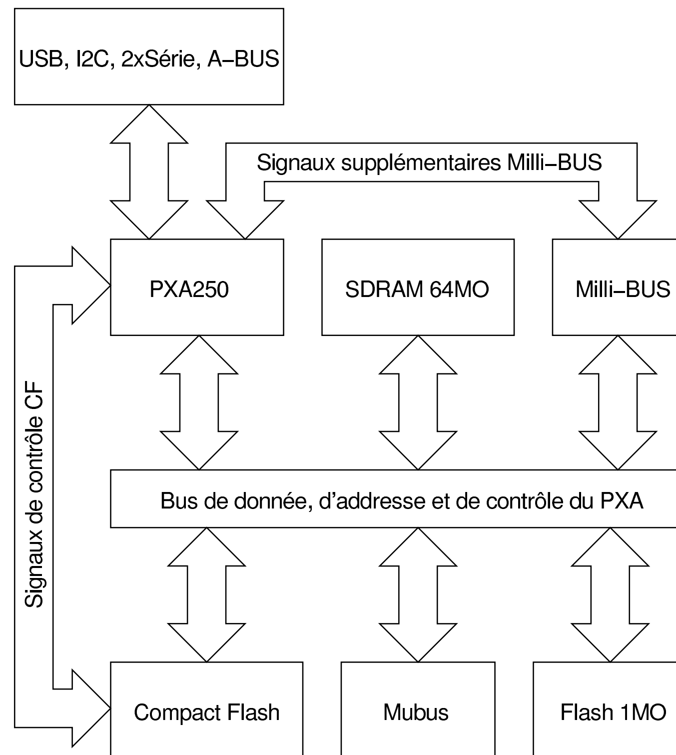

FIG. 2.2 – Schéma bloc Armonie

Armonie est visible sur la figure 2.1. Un schéma bloc est présenté sur la figure 2.2. Le processeur possède un bus de donnée de 32 bits de large, utilisé sur 8, 16 ou 32 bits suivant les périphériques :

Périphérique	Taille utilisée du bus de donnée
Milli-BUS	32
SDRAM	32
mémoire flash	16
CompactFlash	16
Mubus	8

Les autres connecteurs ou phériphériques (série, I^2C) sont directement reliés au PXA250.

Le bus d'addresse est de 26 bits et donne accès à 64 méga-octets pour chacun des 6 signaux de sélection de circuit (*chip select*) de mémoire statique ou flash, pour les quatres signaux de selection de mémoire SDRAM et pour les deux cartes CompactFlash.

Armonie ne contient pas sa propre alimentation, et intègre peu de périphériques. Toutes les fonctionnalités absentes sur Armonie sont reportées sur les connecteurs Milli-BUS (extension) et A-BUS (alimentation). Ainsi, un système complet nécessite 3 cartes. La figure 2.3 page 11 décrit un tel système.

La conception d'Armonie porte une attention particulière à la consommation. En effet, l'adaptation à l'embarqué a été l'un des critères de choix du PXA250. L'énergie est restée une préoccupation importante tout au long du projet.

Armonie est compatible avec d'autres cartes développées pendant la même période au LAP, notamment les cartes de Cédric Gaudin (RokAPEX et RokEPXA) et celle de Damien Baumann (EPXA4-USB2), grâce aux bus A-BUS et Milli-BUS.

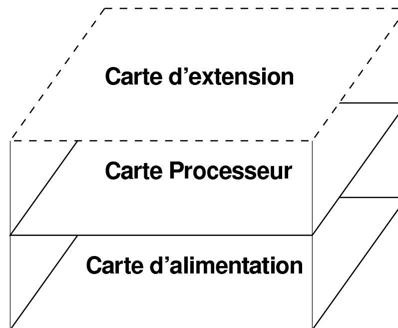

FIG. 2.3 – Système Armonie complet à trois cartes

2.2 Rappels techniques sur l'architecture ARM

Armonie utilise un processeur de la famille ARM. C'est une famille de processeurs à instructions de taille fixe de 32 bits, avec un jeu d'instruction à exécution conditionnelle. La figure 2.4 montre un exemple d'instruction ARM.

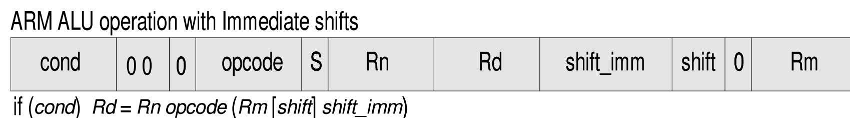

FIG. 2.4 – Exemple d'instruction ARM

L'architecture ARM compte plusieurs versions, allant de 1 à 5[4]. La famille XScale, dont fait partie notre processeur, implémente la version V5TE de l'architecture (MMU, thumb, instructions DSP, mais pas de calcul en virgule flotante).

Ce jeu d'instructions est optimisé pour que le processeur qui l'implémente remplisse le plus possible ses unités fonctionnelles, et ceci à chaque coup d'horloge ou presque. En effet, la richesse de ce jeu permet d'obtenir du code compact et efficace. Ainsi, le silicium est mieux utilisé et le rapport

$$R = \frac{E_{calcul}}{E_{fuite}}$$

est maximisé, où E_{calcul} est l'énergie dépensée pour le calcul et E_{fuite} est l'énergie perdue dans les courants de fuite. Ce rapport représente donc le rendement.

2.3 Particularités du processeur PXA250

Le processeur Intel XScale PXA250 possède de nombreux périphériques intégrés, ne consomme pas beaucoup (70 mW pour le coeur en utilisation typique à 100 MHz) et permet de changer sa tension d'alimentation (0.81 V - 1.43 V) et sa fréquence (100 MHz - 400 MHz) pour adapter la consommation et les performances à la charge de calcul. C'est le successeur du StrongARM.

Il possède les principaux périphériques intégrés suivants :

- contrôleur de mémoires dynamiques SDRAM,
- contrôleur USB esclave,
- contrôleur CompactFlash,
- ports I^2C, séries, Bluetooth et infrarouge, AC97, SPI, SSP,
- contrôleur LCD.

[4]Les processeurs ARM ont parfois des numéros plus élevés (par exemple ARM 9). Ce nom ne fait pas référence à l'architecture, mais à la version du processeur.

2.3.1 Les caches de mémoire

Le PXA250 possède 32 kilo-octets de cache de données et 32 kilo-octets de cache d'instructions. En plus, il possède deux mini-caches de 2 kilo-octets (instructions et données) prévus pour des opérations de débogages.

Les deux caches principaux sont organisés en 32 ensembles de 32 voies comme illustré sur la figure 2.5. Les mini-caches possèdent également 32 ensembles, mais seulement 2 voies. La table 2.1 résume les caractéristiques des différents caches présents sur le PXA250.

Les mini-caches de données et d'instructions ne peuvent être utilisés que par l'intermédiaire du port JTAG, pour pouvoir réaliser un méchanisme de débogage croisé. Le chapitre 6 explique comment nous avons exploité ce système.

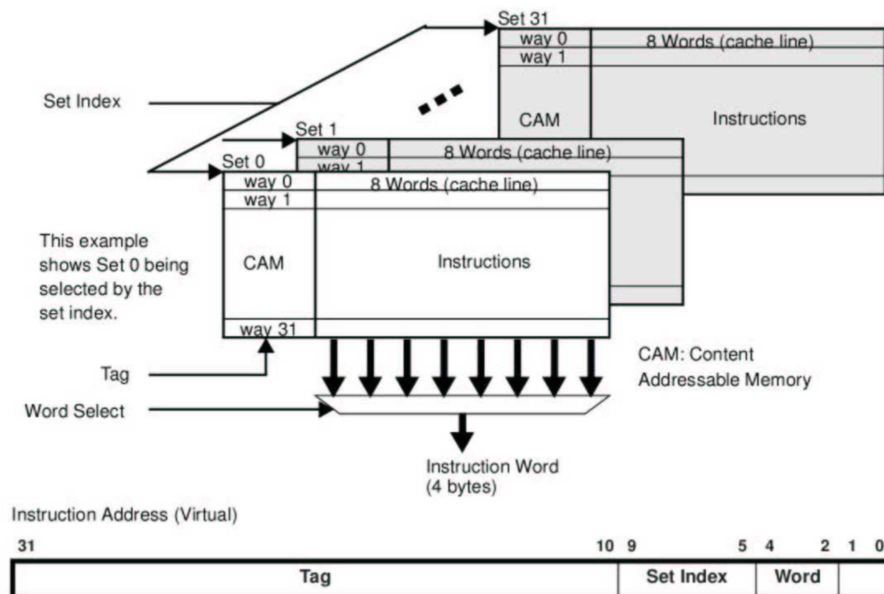

FIG. 2.5 – Organisation du cache d'instructions du PXA250.
Le cache de donnée est organisé de la même manière. Ce document est tiré de la page 4-1 du manuel de l'utilisateur [2].

Cache	Ensembles	Voies	Taille	Mots par ligne
d'instructions	32	32	32 ko	8
mini-cache d'instructions	32	2	2 ko	8
de données	32	32	32 ko	8
mini-cache de données	32	2	2 ko	8

TAB. 2.1 – Résumé des mémoires caches du PXA250.

Le cache d'instructions n'est pas modifié lors d'écritures en mémoire. Ceci rend impossible la réalisation de programmes auto-modifiants. Il faut aussi prendre garde à invalider le cache d'instructions entre le chargement d'un programme (depuis une mémoire de masse) et son exécution : on risque sinon d'exécuter un contenu caché invalide.

Pour améliorer les performances ou pour rendre le comportement d'un programme plus prévisible, le PXA250 permet de vérouiller des lignes de cache d'instructions.

Il s'agit, une fois qu'une ligne cache l'adresse voulue, d'empêcher le processeur de la réutiliser pour une autre adresse. Jusqu'à 28 lignes de caches d'instructions par ensemble peuvent être vérouillées. De la même manière, il est possible de convertir en mémoire RAM jusqu'à 28 lignes de cache de données par ensemble, ce qui équivaut à 28 kilo-octets. Nous n'avons pas testé ces fonctionnalités.

Le tourniquet (*round robin*) est l'algorithme utilisé pour remplacer les lignes de caches de données et d'instructions. Chaque ensemble possède un compteur pointant sur la voie à utiliser lors d'une prochaine allocation de ligne. Une fois la voie 31 utilisée, le compteur cherche la première voie non verrouillée à partir de la voie 0.

2.3.2 Fréquences et tensions d'alimentations

L'alimentation du PX250 est intéressante : il est possible de l'adapter suivant la fréquence de fonctionnement désirée.

Deux tensions principales sont à distinguer :
- l'alimentation du coeur, variable entre 0.85 V et 1.3 V ;
- l'alimentation du bus et des signaux externes, à 3.3 V.

Le PXA250 fait preuve d'une certaine souplesse en ce qui concerne ses fréquences de fonctionnement. Deux fréquences sont à distinguer : celle du coeur, cadencé entre 100 MHz et 400 MHz, et celle du bus interne (*PXbus*) que nous avons testé à 50 MHz et à 100 MHz. La section 6.8 aborde plus en détails les mécanismes de changements de fréquences. La section 4.6 explique comment utiliser nos outils pour opérer ces changements.

2.4 La carte EzUSB

Lors d'un précédent projet de semestre au LAP (à l'époque le LAMI), une petite carte d'expérimentation contenant un microcontrôleur EzUSB[28] de Cypress a été développée par Sebastian Gerlach, en vue de créer un analyseur logique USB.

Cette carte contient :
- le microcontrôleur, un dérivé du 8051 intégrant un coeur USB,
- le connecteur esclave USB,
- deux connecteurs 10 pattes connectés aux ports B et C du microcontrôleur,
- quelques autres connecteurs, ainsi qu'un quartz.

Cette carte est intéressante pour notre projet car elle permet d'avoir des pattes 3.3 V gérables individuellement à travers l'USB avec en prime une petite capacité de calcul, le tout simplement et à faible coût.

Nous avons utilisé cette carte comme adaptateur JTAG.

Chapitre 3

Directions et choix

Ce chapitre présente les différents objectifs d'Armonie que nous avons voulu développer et les choix qui en ont découlés.

3.1 Directions

Afin de mettre en valeur notre carte, nous avons choisi trois directions différentes et tenté d'obtenir un résultat qui soit satisfaisant du point de vue de chacune. Ces directions sont :

- Une plateforme pédagogique,
- Une carte processeur pour le robot cyclope,
- Un système multimédia embarqué.

Les sections qui suivent les détaillent. Ces trois axes couvrent un horizon aussi large que possible tout en gardant une complexité maîtrisable dans un projet de diplôme à deux. De plus, une partie importante du travail (les outils, le portage de Linux) est commune aux trois directions.

3.1.1 Plateforme pédagogique

Dans les années à venir, René Beuchat donnera pour l'institut Informatique et Communication deux nouveaux cours : systèmes embarqués et systèmes embarqués temps-réel.

Ces cours s'accompagneront de travaux pratiques. Notre carte et surtout les outils de développement qui vont avec pourront fournir une des plateformes pour ce cours. Dans un contexte pédagogique, il est intéressant d'avoir une carte développée en interne. Le résultat est non seulement tout à fait adapté aux besoins du laboratoire, mais permet en plus de bien comprendre le fonctionnement de la carte, de faire rapidement des extensions et de plus facilement réparer les cartes en cas de problème. De plus, avec des processeurs de la complexité du XScale, avoir une bonne maîtrise interne et une documentation complète du logiciel de bas niveau est indispensable si l'on veut un travail de qualité. C'est la raison pour laquelle, malgré un prix plus élevé qu'une solution commerciale classique (iPaq de Compaq par exemple), notre carte présente un intérêt pédagogique.

3.1.2 Carte processeur pour le robot cyclope

Il existe au LAP un robot mobile, le cyclope. Ce robot se présente sous la forme d'un cylindre de 15 cm de diamètre et de 5 cm de haut. Il possède deux roues, des capteurs de choc, des capteurs de distance et une caméra linéaire. La figure 3.1 présente une photo du robot cyclope. Au centre de l'image, on peut voir la carte processeur avec un HC12 (une version 68336 existe aussi). C'est cette carte là qu'Armonie vient remplacer. Les dimensions d'Armonie ont été choisies en fonction de celles du cyclope, afin de s'y intégrer harmonieusement. Au cours du projet, nous avons décidé de moins nous concentrer sur le cyclope, et plus sur les deux autres voies. Plusieurs raisons motivent notre choix :

– Pour s'interfacer correctement avec le robot Cyclope, Armonie a besoin d'une carte intermédiaire qui s'occupe de l'alimentation sur batterie et de l'accès aux capteurs. Adrian Spycher a réalisé une telle carte.
– Les cartes de Cédric Gaudin, basées sur des FPGA, semblent plus appropriées pour cette direction.

Ainsi, comme notre temps est limité, nous avons préféré nous concentrer sur des outils de qualités, intéressants pour l'enseignement et sur une carte d'extension, permettant de découvrir plus de périphériques extérieurs au PXA250. Nous terminons là de parler du cyclope.

FIG. 3.1 – Robot cyclope

3.1.3 Système multimédia embarqué

La puissance du XScale permet de décompresser du son, et, dans une mesure raisonnable, de la vidéo. Alliée à sa faible consommation, cette puissance rend l'idée d'un système multimédia embarqué attrayante et réaliste. Nous avons donc décidé de développer une carte d'extension, nommée eMediaKit, qui, se connectant sur le port Milli-BUS, apporte des fonctionnalités multimédias. Elle contient divers connecteurs d'entrée-sortie sons, un écran LCD tactile, un adaptateur Ethernet et un contrôleur USB maître. Cette carte a été développée en collaboration avec Cédric Gaudin. En effet, l'utilisation du Milli-BUS, dont la spécification est claire, permet d'utiliser plusieurs cartes processeurs avec la même carte d'extension.

3.2 Choix

Tout au long du projet, nous avons dû faire certains choix que nous résumons ici.

3.2.1 Outils de développement

Les outils de développement commerciaux pour PXA250 sont onéreux. Nous avons donc décidé d'écrire nos propre outils, en nous basant sur :
- du matériel existant au LAP (carte EzUSB),
- les outils de développement croisés GNU,
- la documentation JTAG du PXA250.

3.2.2 Le choix du système d'exploitation

Avant de nous décider pour Linux, nous avons analysé les choix à notre disposition. Les systèmes d'exploitation suivants fonctionnaient sur le PXA250 au moment de notre choix :
- Windows CE,
- eCos,
- NetBSD,
- Linux.

Utiliser du logiciel libre chaque fois que possible a toujours été une ligne de conduite lors de ce projet. Windows CE n'étant pas libre, nous l'avons exclu d'office. eCos est intéressant car il est petit et assez léger. Par contre, il ne supporte pas la mémoire protégée ni le multi-utilisateurs et son support réseau est limité. Il a bien moins de fonctionnalités que Linux ou NetBSD. De plus, il ne supporte que peu de périphériques intégrés au PXA250. Nous l'avons donc aussi éliminé.

Restaient NetBSD et Linux. Entre les deux le choix a été plus difficile. Nous avons choisi Linux parce qu'il supporte mieux les périphériques intégrés du PXA250 et parce que, sur XScale, la communauté des développeurs Linux est plus grande que la communauté des développeurs NetBSD. En particulier, Linux supporte le contrôleur LCD du PXA250, que NetBSD ne sait pas piloter.

3.3 Collaboration extérieure

Le XScale étant un processeur récent et attrayant, plusieurs groupes s'y intéressent. Conscients que nous avons tous quelque chose à gagner à collaborer, nous avons travaillé en rapport avec K-Team, une entreprise de robotique située à Préverenge. K-Team désirait aussi faire une carte, mais n'avait pas plus d'expérience que nous dans le XScale au début du projet. Nous avons décidé de faire une carte simple, sans alimentation et avec les fonctionnalités minimales (processeur, mémoire, ports), utilisant des connecteurs d'extensions pour le reste. K-Team, qui voulait une carte plus complexe, a ainsi pu disposer d'une base matérielle fonctionnelle. De plus, afin de pouvoir utiliser notre carte, nous avons écrit du logiciel – Jelie – que K-Team utilise aussi.

Une fois que la carte de K-Team a été construite, nous avons aidé à la faire fonctionner en fournissant outils et conseils, en particulier pour le démarrage de Linux.

Chapitre 4

Utiliser Armonie

Ce chapitre donne une vue générale des outils de développement sur Armonie puis explique comment les utiliser. Les détails d'installation, de compilation ou de fonctionnement de ces outils sont abordés dans les chapitres 5 à 7.

Deux possibilités sont proposées pour exécuter un logiciel sur la carte. La première solution permet au système d'être autonome : les logiciels sont mémorisés dans la mémoire *flash* ou sur un support CompactFlash. Pour les essais, on choisira de préférence la deuxième solution, et on enverra les logiciels depuis un ordinateur de développement, par l'intermédiaire d'outils de développement croisé. Dans ce cas, les programmes pourront s'exécuter depuis la mémoire SDRAM ou depuis le mini-cache d'instruction.

Ce chapitre explique aussi l'utilisation de Linux sur Armonie, en partant des fichiers de la distribution. Le chapitre 7 expliquera en détail comment développer Linux lui-même.

4.1 Outils de développement

Pour développer de nouveaux logiciels pour Armonie, plusieurs outils sont nécessaires. Un compilateur croisé compilera du code source en programme exécutable qui devra ensuite être chargé sur Armonie avant l'exécution. Plusieurs moyens de chargement sont possibles : par port JTAG, par port série ou par connection USB.

Nous verrons au chapitre 6 que notre programme Jelie, via le port JTAG, permet le chargement de programmes – Linux par exemple. Linux tourne correctement sur Armonie, et permet l'utilisation facile du réseau sur port USB. L'exécution et le test de nouveaux programmes depuis un poste de développement deviennent alors très aisés. Le débogage reste néanmoins possible sans système d'exploitation en utilisant GDB[1] et Jelie.

La figure **??** page ?? montre les différentes possibilités de branchement d'Armonie, et la figure 4.1 décrit la collaboration des outils de développement Armonie.

4.1.1 Jelie

Jelie est notre programme de pilotage d'interface JTAG pour les processeurs XScale d'Intel. Il permet de charger, d'exécuter et de déboguer des programmes. Il sert aussi à connecter GDB sur un PXA250 pour déboguer depuis un PC. Sur une carte Armonie, Jelie permet également d'effacer et de modifier la mémoire flash.

Le manuel de l'utilisateur de Jelie (annexe 1) explique comment utiliser ce programme, tandis que le chapitre 6 détaille son fonctionnement interne.

Nous avons commencé le développement de Jelie en juin 2002. Actuellement, Jelie est composé d'environ 10000 lignes de C, de C++ et d'assembleur ARM. Il est maintenant fonctionnel et documenté.

[1] Le débogueur utilisé est Insight [8], une version de GDB munie d'une interface graphique.

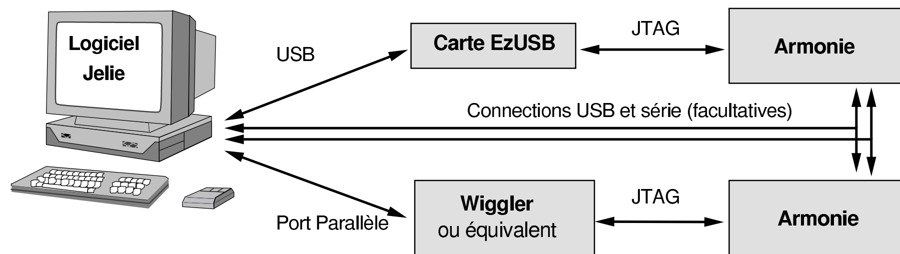

FIG. 4.1 – Connectiques possibles entre Armonie et un ordinateur de développement.

Il y a deux manières possibles de connecter le port JTAG d'Armonie à un ordinateur de développement. Les ports USB esclave et série d'Armonie peuvent aider le développement, mais ne sont pas nécessaires.

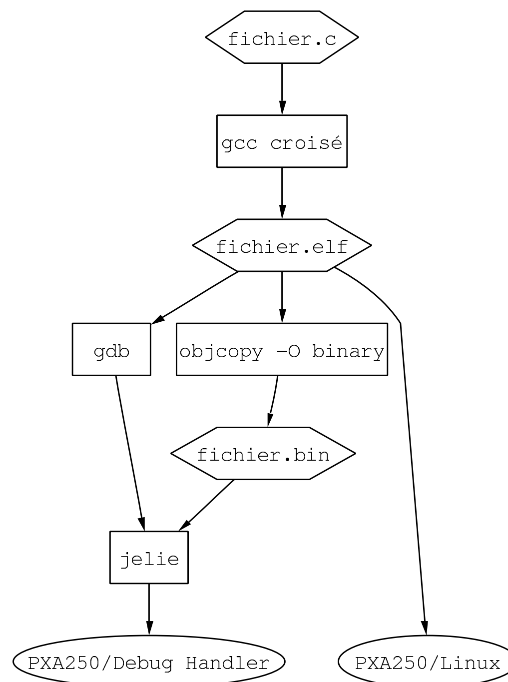

FIG. 4.2 – Chaîne d'outils de développement

Ce graphe montre les différents chemins possibles entre un fichier source et le PXA250. Objcopy, GDB et Linux peuvent les trois charger un programme ELF en mémoire.

4.1.2 Compilateur croisé

Le compilateur croisé que nous utilisons est GCC 3.2.2 pour Linux avec la bibliothèque C μClibc. Le nom de l'exécutable à appeler est : `arm-uclibc-gcc`.

Voici un bref résumé des options de GCC :

-g inclut les informations de débogage.

-o fichier spécifie le nom du fichier résultant.

-c produit un fichier object et n'effectue pas l'édition de liens.

-e label lors de l'édition de liens, spécifie le point d'entrée, sous forme d'adresse ou de label.

-nostartfiles -nostdlib -lgcc ne lie pas de bibliothèque C ni le programme par défaut qui appelle la fonction main.

-Ttext 0xA0000000 lors de l'édition de liens, place le segment *text* à l'adresse 0xA0000000.

-O2 et -O3 active les optimisations.

4.1.3 Débogueur

Le débogueur à utiliser est GDB 5.3, légèrement modifié pour s'interfacer avec Jelie et le PXA250. Trois logiciels sont utilisés simultanément : GDB et Jelie sur le PC hôte et l'application en cours de débogage sur le processeur cible.

4.2 Plan mémoire du PXA250

La figure 4.2 page 20 cartographie l'espace mémoire de la carte Armonie.

4.3 Initialisation

Pour qu'Armonie fonctionne correctement, un programme doit commencer par initialiser le contrôleur de mémoire et quelques périphériques. Ce programme d'initialisation peut soit se trouver en mémoire flash, soit être chargé par le port JTAG dans le mini-cache d'instructions.

Normalement, la séquence est la suivante :

1. Le processeur saute à l'adresse 0x00000000, le vecteur de redémarrage. A cette adresse se trouve une table d'instructions *branch* menant vers différents gestionnaires d'exceptions.
2. Si le démarrage se fait par le JTAG en vue de débogage, les fonctionnalités de débogage sont activées.Cette étape n'est pas utile lors d'un démarrage depuis la mémoire flash (voir page **??** pour plus de détails).
3. Le gestionnaire d'exception de démarrage configure les GPIO (*General Purpose Input Output*, patte d'entrée/sortie), un port série et le contrôleur de mémoire vive.
4. Le processeur est maintenant prêt à exécuter un programme.

Jelie effectue automatiquement cette séquence pour initialiser Armonie.

4.3.1 Initialisation du contrôleur de SDRAM

Le site web d'Intel[2] fournit une application qui génère le code assembleur qui configure le contrôleur de mémoires à partir d'une description précise des composants connectés (*timings*, tailles, etc).

Le code généré ne convient pas à l'exécution en mini-cache d'instructions, car il est impossible dans ce cas d'accéder en mode données au programme. Nous l'avons donc modifié en remplaçant toutes les instructions

[2] http ://appzone.intel.com/pcg/pxa250/memory/

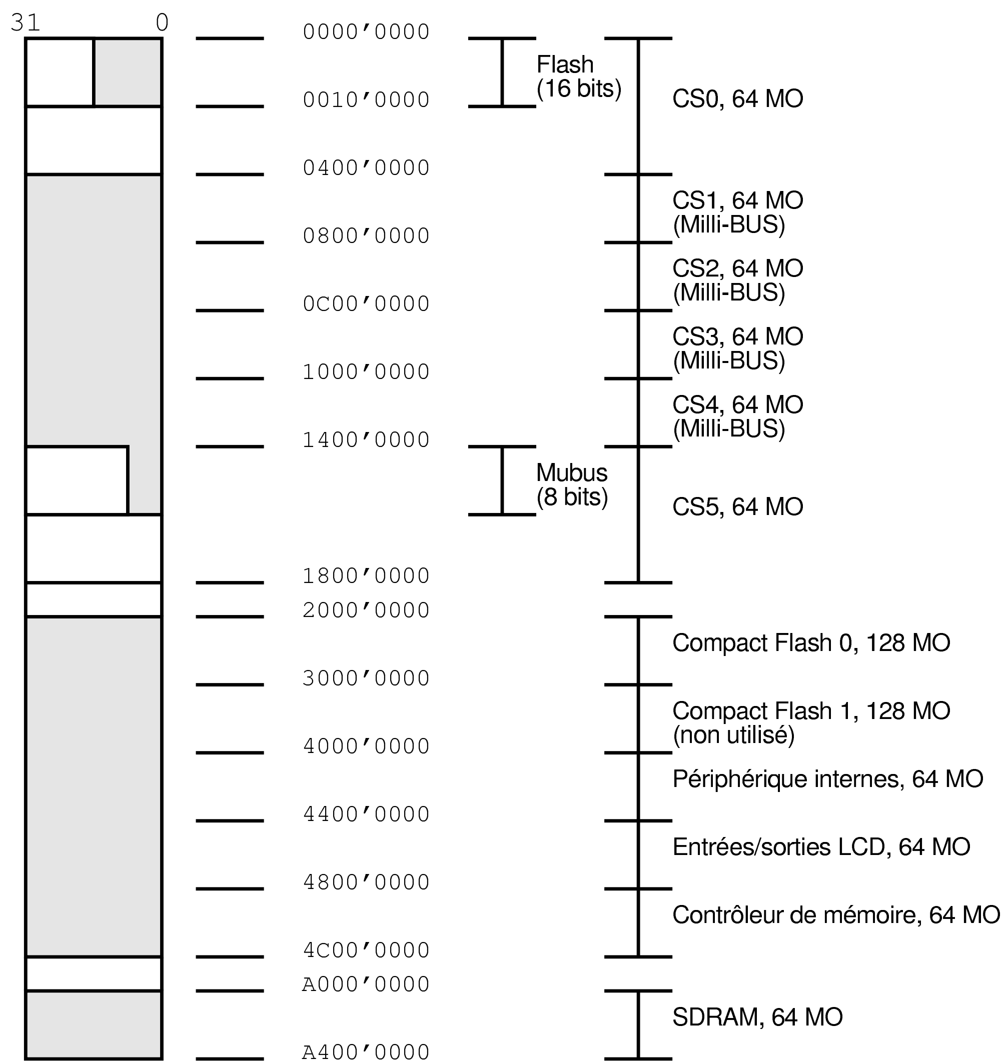

FIG. 4.3 – Carte de l’espace mémoire d’Armonie.

```
ldr rx, =constante
```

par une macro chargeant le registre en plusieurs étapes de 8 bits, avec un `mov` et autant de `add` que nécessaire.

4.4 Utilisation de la mémoire flash

La mémoire flash est une mémoire non-volatile et réinscriptible. Elle permet des accès aléatoires en lecture, mais pas en écriture. Pour l'écriture, la mémoire est partitionnée en secteurs de tailles variables. En effet, de par sa technologie, l'écriture nécessite d'abord un effacement. Ce dernier peut être effectué sur un secteur ou sur toute la puce, mais pas sur une autre zone. L'effacement a pour effet de mettre à 1 tous les bits. Après, l'écriture mettra à zéro les bits nécessaires. Le cycle effacement / écriture est aussi appelé programmation.

Notre mémoire flash, l'AS29LV800[27], fait 1 méga-octets (512k x 16). La table 4.1 illustre son organisation en secteurs.

Secteur	Addresses	Taille (ko)
0	0x00000 - 0x03FFF	16
1	0x04000 - 0x05FFF	8
2	0x06000 - 0x07FFF	8
3	0x08000 - 0x0FFFF	32
4	0x10000 - 0x1FFFF	64
5	0x20000 - 0x2FFFF	64
6	0x30000 - 0x3FFFF	64
7	0x40000 - 0x4FFFF	64
8	0x50000 - 0x5FFFF	64
9	0x60000 - 0x6FFFF	64
10	0x70000 - 0x7FFFF	64
11	0x80000 - 0x8FFFF	64
12	0x90000 - 0x9FFFF	64
13	0xA0000 - 0xAFFFF	64
14	0xB0000 - 0xBFFFF	64
15	0xC0000 - 0xCFFFF	64
16	0xD0000 - 0xDFFFF	64
17	0xE0000 - 0xEFFFF	64
18	0xF0000 - 0xFFFFF	64

TAB. 4.1 – Organisation des secteurs dans la mémoire flash

Jelie est également utilisé pour programmer cette mémoire. Il s'occupe de l'effacement préalable et de l'écriture, à partir d'une adresse quelconque et même sur plusieurs secteurs. Jelie fournit donc une méthode simple, extensible et propre. Le manuel utilisateur de Jelie, disponible en annexe 1, documente précisement la syntaxe des commandes relatives à la mémoire flash.

4.5 Charger un programme sans système d'exploitation

Comme le montre la figure 4.1, il y a deux possibilités pour charger un programme sans système d'exploitation :

- en utilisant GDB
- ou directement Jelie.

L'exemple d'un petit programme de test Mubus[3] va illustrer tout le chemin entre le code source et l'exécution d'un programme sans l'utilisation d'un système d'exploitation. La figure 4.3 contient le code source d'exemple.

```
unsigned int *Mubus =(unsigned int *) 0x14000000;

int main() {
        int i=1;

        while(1) {
                int j;

                if (Mubus[0] & 1) {
                        i = i << 1;
                        if ( i &0xFFFF0000)
                                i=1;
                } else {
                        i = i >> 1;
                        if ( i==0)
                                i = 0x00008000;
                }

                Mubus[0] = i&0xFF;
                Mubus[1] = i >>8;

                // wait a bit
                for ( j=0; j<0x8000; j++);
        }
}
```

FIG. 4.4 – Programme d'exemple qui utilise le Mubus sans système d'exploitation.

La première étape consiste à créer un exécutable ARM au format ELF[4] en utilisant le compilateur GCC. Pour cela, il faut compiler le fichier `mubus.c` pour obtenir un fichier objet (`mubus.o`) avec la commande suivante :

```
arm-uclibc-gcc -g -c -o mubus.o mubus.c
```

Les options de GCC sont résumées par le paragraphe 4.1.2 page 17.

Ensuite, quelques lignes d'assembleur serviront à initialiser une pile puis à appeler la fonction `main` – le fichier `bm.S` (fig. 4.4) est un exemple d'un tel méchanisme d'initialisation. Voici comment l'assembler :

```
arm-uclibc-gcc -g -c -o bm.o bm.S
```

L'édition de liens est la phase qui produira l'exécutable. Elle est un peu particulière puisqu'il faut forcer GCC à ne pas utiliser la bibliothèque standard C. La bibliothèque *libgcc* est nécessaire pour que le code généré par GCC fonctionne correctement. Pour ces raisons, il faut passer à GCC ces options : `-nostartfiles -nostdlib -lgcc`.

Il faut également préciser que le point d'entrée dans le programme est `_start` dans `bm.S`. L'option `-Ttext 0xA0000000` sert à placer le segment de code dans la mémoire SDRAM à l'adresse 0xA0000000. Voici comment obtenir le fichier ELF à partir des fichiers objets `bm.o` et `mubus.o` :

[3]Ce test suppose qu'un afficheur Mubus doté d'interrupteurs est branché sur Armonie. Ce type d'afficheur, fabriqué au LAMI, est encore très répandu au LAP.

[4]*Executable and Linking Format.*

```
. text
.global  _start
_start :
        mov r0, #0
        mov r1, #0
        mov r2, #0
        mov r3, #0
        mov r4, #0
        mov fp, #0
        mov sp, #0xa4000000   @ Hard coded stack address
        bl  main
        b _start          @ if  main() returns :  restart   it .
```

FIG. 4.5 – Programme d'initialisation de pile et d'appel de la fonction `main`.

```
arm-uclibc-gcc -g -nostartfiles -nostdlib -Ttext 0xa0000000 \
-e _start -lgcc mubus.o bm.o -o mubus
```

La commande `make` permet d'automatiser ce processus grâce au fichier `Makefile` qui contient les règles de compilation pour Armonie.

Le fichier `mubus` contient à présent une liste de segments à charger à des adresses précises, avec un point d'entrée et des informations de débogage : tout ce qu'il faut à GDB pour démarrer le programme, en pilotant Jelie.

Avant d'appeler GDB, il faut préparer la carte Armonie et démarrer Jelie qui devrait répondre plus ou moins comme ceci :

```
./jelie -p
This is jelie version 1.0
Julien Pilet & Stephane Magnenat, 6.2002 - 3.2003.
JTAG control pp used
idcode 0x39264013:
PXA250 version 3 detected
446 instructions read from debugHandler/debug_handler.bin
440 on 446 instructions loaded. adding 2 nops
446 instructions successfully loaded.
(target ready at PC=000000BC)
r>
```

A ce stade, il y a deux possibilités pour lancer le programme : avec ou sans GDB.

4.5.1 Exécuter un programme en utilisant GDB

Tout est maintenant prêt pour GDB. Voilà comment l'invoquer :

```
arm-linux-insight mubus
```

Une fenêtre apparait. Depuis le menu `File` puis `Target Settings`, il faut configurer GDB pour la communication avec Jelie. La fenêtre de configuration est illustrée par la figure 4.5. Pour que tout fonctionne, il faut choisir sous *Target* l'option *Remote/TCP*. Le champ *hostname* contient le nom de l'ordinateur sur lequel tourne Jelie, typiquement `localhost`. Le port par défaut de Jelie, qui peut être modifié par l'option `-p`, est 3141.

Une fois la connection avec Jelie configurée, il suffit de cliquer sur l'icône *Run* représentée par un bonhomme qui court pour que le programme commence son exécution.

On peut voir l'environnement graphique de débogage sur la figure 4.6 page précédente.

FIG. 4.6 – Fenêtre de configuration de GDB pour la communication avec Jelie.

Lors de l'invocation de Jelie, lui passer comme option de ligne de commande `-d` aura pour effet d'afficher le contenu de la communication avec GDB. Cela nous a permis de résoudre facilement les problèmes d'interaction entre Jelie et GDB.

4.5.2 Exécuter un programme en utilisant Objcopy

Le programme objcopy permet de produire un fichier binaire à charger directement en mémoire à partir d'un programme exécutable ELF. Jelie est ensuite capable de charger un tel fichier et de lancer son exécution.

Voici la commande qui produit le fichier binaire :

```
arm-uclibc-objcopy -O binary mubus mubus.bin
```

Il faut encore trouver l'adresse du symbole `_start`, contenue dans le fichier `bm.S`. On utilisera le programme `arm-uclibc-nm` sur le fichier ELF `mubus`. Celui-ci liste les symboles d'un programme et en donne les adresses.

Il n'y a plus qu'à dire à Jelie de charger le programme, en entrant dans la ligne de commande Jelie :

```
load mubus.bin
```

La dernière étape, l'exécution, consiste à sauter à l'adresse du symbole `_start`, 0xa00000e0 dans cet exemple. Du point de vue de Jelie, il faut modifier le registre R15 (le PC) pour qu'il contienne cette adresse, puis continuer l'exécution, comme si un programme était déja en cours de débogage. C'est ce que font ces deux commandes :

```
register 15 a00000e0
continue
```

4.6 Changer les fréquences du processeur et du bus

Les scripts et programmes fournis avec Jelie permettent de changer les fréquences du processeur et du bus (*PXbus*) et de passer en mode turbo. Cette section explique comment utiliser ces fonctions. La section 2.3.2 donne plus d'information sur les fréquences possibles, alors que la section 6.8 explique comment ces programmes réalisent les changements de fréquences.

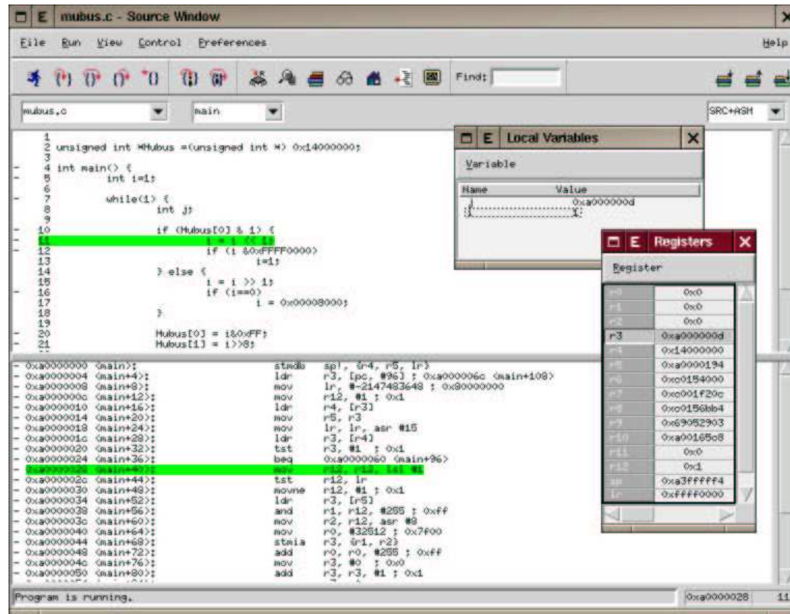

FIG. 4.7 – Environnement graphique de débogage d'un programme tournant sur Armonie sans système d'exploitation.

Les scripts de changement de fréquence appellent des programmes qui, selon la documentation d'Intel, ne sont pas sensés marcher. Pourtant, ils fonctionnent très bien sur les cartes que nous avons réalisées. Nous ne garantissons donc pas leur fonctionnement sur tous les processeurs de la famille XScale.

Les scripts Jelie suivants sont disponibles :

goturbo charge et exécute `turbo.bin`. Passe le processeur en mode turbo.

loadlinux_turbo passe le processeur en mode turbo avant de charger Linux (voir la section 4.7).

chspeed charge et exécute `fcs.bin`, ce qui change la vitesse du coeur à 200 MHz et la vitesse du PXbus à 100 MHz.

loadlinux_fcs appel le script `chspeed` puis charge Linux.

4.7 Guide d'utilisation de Linux

4.7.1 Prérequis

Voici ce qu'il faut préparer pour pouvoir faire démarrer Linux sur Armonie :
- une carte Armonie connectée par port JTAG à l'ordinateur de développement,
- le programme Jelie,
- une image compressée du noyau : `armlinux/zImage`,
- une image compressée d'un système de fichier : `armlinux/initrd.gz`,
- un script Jelie de démarrage contenant des chemins corrects : `armlinux/loadlinux`,
- un terminal branché sur le port série 0 (J2).

Pour utiliser le réseau via USB, il faut en plus brancher la carte Armonie à l'ordinateur de développement qui devra tourner sous Linux, avec le pilote `CONFIG_USB_USBNET` installé (voir la section 7.2.9 page 45).

La figure **??** illustre ceci.

4.7.2 Démarrage

Une fois les préparatifs terminés, il faut effectuer les opérations suivantes :

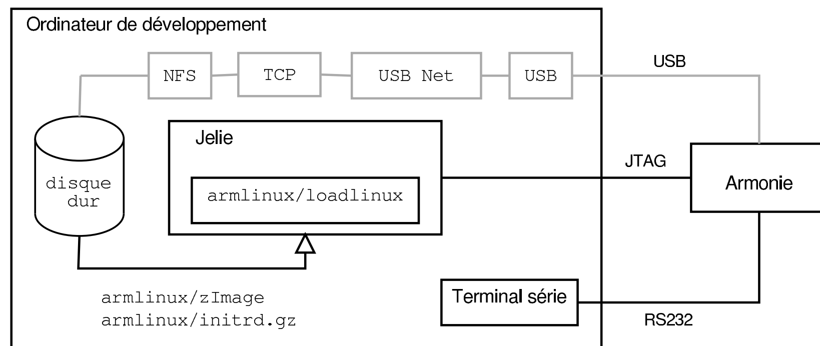

FIG. 4.8 – Prérequis pour utiliser Linux sur Armonie. La partie grisée est facultative.

1. lancer Jelie,
2. appeler le script `loadlinux`.

A ce moment, Linux et l'image disque sont chargés. Si tout va bien, le noyau démarre. Pour configurer le réseau via USB, les commandes suivantes sont nécessaires :

1. sur le PXA250 : `ifconfig usbf 192.168.0.2`
2. sur le PC : `ifconfig usb0 192.168.0.1`

Le réseau devrait maintenant fonctionner. La commande `ping` permet de le vérifier.

4.7.3 Travailler depuis l'ordinateur de développement

Une fois le réseau à travers USB en place, il est possible de monter un disque dur distant comme s'il était local grâce au système NFS (Network File System [14]).

L'ordinateur de développement doit dans ce cas exporter un système de fichier, par exemple le répertoire `armlinux/nfsarm`. Pour ce faire, il faut éditer le fichier `/etc/exports` et rajouter la ligne :

```
/armlinux/nfsarm 192.168.0.2(rw, no_root_squash)
```

Ensuite, la commande suivante montera le système de fichier sur Armonie :

```
mount -t nfs -o nolock 192.168.0.1 :/armlinux/nfsarm /nfs
```

Le répertoire `/nfs` devient sur Armonie le même que `/armlinux/nfsarm` sur l'ordinateur de développement.

4.7.4 Une distribution GNU/Linux complète

Nous avons mis la distribution GNU/Linux pour ARM PsiLinux[18]. Si tout fonctionne, on y peut accéder dans le répertoire `/nfs/psilinux`.

La commande `chroot /nfs/psilinux` permet alors de remplacer le répertoire racine par PsiLinux : l'environnement change et d'autres programmes sont à disposition. En particulier, il devient possible d'exécuter des programmes à édition de liens dynamique.

4.7.5 Contrôler l'écran LCD

Nous avons écrit le programme fbcontrol pour activer ou désactiver l'affichage sur l'écran et l'affichage du curseur. Voici les différentes possibilités pour l'appeler :

- `fbcontrol c` – active l'affichage du curseur
- `fbcontrol C` – désactive l'affichage du curseur
- `fbcontrol 0` – active l'affichage de l'écran
- `fbcontrol 1` – l'affichage à l'écran sera désactivé dans une minute (le DMA d'affichage sera désactivé).

Chapitre 5

Chaîne d'outils GNU

Ce chapitre explique comment nous avons adapté les outils de développement croisé GNU pour un bon fonctionnement avec Armonie.

Nous avons utilisés quatres paquetages principaux :

– Binutils qui contient un assembleur, un éditeur de liens et des outils de manipulation de fichiers objets ;
– GCC [7], le compilateur croisé C et C++ ;
– μClibc[17], une bibliothèque C pour système embarqué ;
– GDB / Insight [8], un débogueur. C'est une version de GDB doté d'une interface graphique.

Tous ces programmes doivent être configurés au moment de la compilation. Il faut spécifier le type d'architecture – ARM dans notre cas. Il faut également préciser la famille de processeur : XScale. Le système d'exploitation et la librairie C à utiliser sont les deux derniers choix : μClibc sous Linux.

5.1 Automatisation de la création des outils

La création de la chaîne d'outils est grandement simplifiée grâce à un script d'Erik Andersen, du projet μClibc. Voici les étapes pas à pas pour construire les outils depuis le CVS de μClibc :

1. Aller à la page `http://uclibc.org/cgi-bin/cvsweb/toolchain/`.
2. Cliquer sur *Download tarball* en bas de la page. Ceci télécharge une archive du script de création.
3. Décompresser cette archive.
4. Changer le répertoire courant pour celui contenant les scripts correspondants à la version de GCC désirée.
5. Editer le fichier `Makefile`, et effectuer les changements suivants :
   – Changer le répertoire `BASEDIR` vers le répertoire de destination désiré.
   – Changer l'architecture cible en `ARCH :=arm`.
   – Si une archive de la chaîne compilée est souhaitée, écrire `BUILD_TARBALL :=true`.
   – Activer le support spécifique XScale par `EXTRA_GCC_CONFIG_OPTIONS=-with-cpu=xscale`.
6. Compiler et installer avec `make`.

Afin de simplifier ce processus, nous avons fourni dans la distribution l'archive du compilateur déjà prêt. Il suffit de la décompresser comme ceci :

```
tar -xvjf arm-uclibc-toolchain.tar.bz2 -C /
```

Ainsi, le compilateur croisé C et C++ sera installé dans `/usr/local/toolchain`.

Pour l'utiliser, il faut rajouter dans la variable d'environement `$PATH` le répertoire `/usr/local/toolchain/bin`.

5.2 GDB

Nous avons adapté GDB 5.3 pour effectuer du débogage croisé sur Armonie. La version Insight Debugger de la suite d'outils GNUPro faite par Redhat ajoute une interface graphique agréable à utiliser.

Puisque GDB n'a aucune idée de comment communiquer avec un PXA250, il doit passer par Jelie qui sait interprèter le protocole de communication série de GDB[9]. La communication entre Jelie et GDB passe par TCP/IP. Jelie exécute les commandes fournies par GDB en utilisant l'interface JTAG et le gestionnaire de débogage (*debug handler*). La figure 5.1 illustre cette organisation.

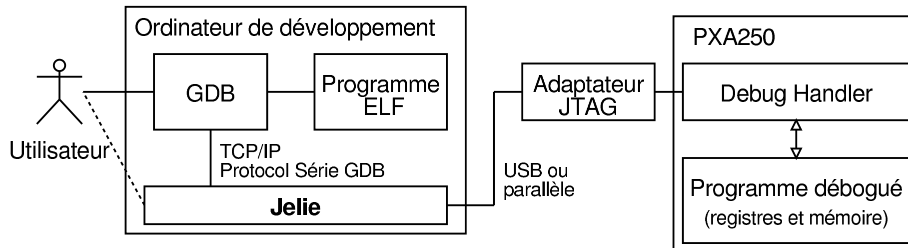

FIG. 5.1 – Comment GDB a accès au PXA250 grâce à Jelie.

GDB, par défaut, ne correspond pas au PXA250. En effet, ce processeur ne possède pas de méthode matérielle pour exécuter un programme pas à pas. Le débogueur doit donc calculer en logiciel quelle sera la prochaine instruction à exécuter puis y placer un point d'arrêt, et ceci à chaque pas d'exécution.

Les modifications que nous avons apportées à GDB sont très simples : activer le pas à pas logiciel (*software single stepping*) et choisir l'instruction de point d'arrêt (*breakpoint*) `bkpt #0`. Le fichier `tm.h` contient ces deux modifications, comme montré sur la figure 5.2.

Pour compiler gdb, les étapes suivantes sont nécessaires :

1. télécharger et décompresser l'archive ;
2. exécuter le script suivant :

```
../insight-5.3/configure --prefix=/usr/local/toolchain/ --target=arm-linux \
--with-headers=/home/jpilet/linux/linux/ \
--with-cpu=xscale \
--enable-threads=single \
"--enable-languages=c,c++" \
--disable-shared \
--disable-nls \
--enable-sim \
--quiet \
--enable-tui \
--enable-gdbtk
```

3. tapper la commande `make` ;
4. puis, en root si nécessaire, tapper la commande `make install`.

Les commandes `arm-linux-gdb` et `arm-linux-insight` sont alors disponibles. Une version compilée est fournie dans la distribution, dans l'archive `tools/arm-uclibc-toolchain.tar.bz2`.

```
/* Definitions to target GDB to ARM embedded systems.
   Copyright 1986, 1987, 1988, 1989, 1991, 1993, 1994, 1995, 1996, 1997,
   1998, 1999, 2000 Free Software Foundation, Inc.

   This file is part of GDB.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA 02111-1307, USA. */

#ifndef TM_ARMEMBED_H
#define TM_ARMEMBED_H

/* Include the common ARM definitions. */
#include "arm/tm-arm.h"

/* The remote stub should be able to single-step. */
//#undef SOFTWARE_SINGLE_STEP_P
//#define SOFTWARE_SINGLE_STEP_P() 0

#undef ARM_LE_BREAKPOINT
#define ARM_LE_BREAKPOINT {0x70,0x00,0x20,0xE1}

/* The first 0x20 bytes are the trap vectors. */
#undef LOWEST_PC
#define LOWEST_PC 0x20

/* Override defaults. */

#undef THUMB_LE_BREAKPOINT
#define THUMB_LE_BREAKPOINT {0xbe,0xbe}
#undef THUMB_BE_BREAKPOINT
#define THUMB_BE_BREAKPOINT {0xbe,0xbe}

/* Functions for dealing with Thumb call thunks. */
#define IN_SOLIB_CALL_TRAMPOLINE(pc, name) arm_in_call_stub (pc, name)
#define SKIP_TRAMPOLINE_CODE(pc)           arm_skip_stub (pc)
extern int arm_in_call_stub (CORE_ADDR pc, char *name);
extern CORE_ADDR arm_skip_stub (CORE_ADDR pc);

#undef IN_SIGTRAMP
#define IN_SIGTRAMP(pc, name) 0

#endif /* TM_ARMEMBED_H */
```

FIG. 5.2 – Le fichier `tm.h` utilisé pour compiler GDB 5.2.1.

Chapitre 6

Jelie

Le chapitre 4 donne une vue d'ensemble des outils de développement et introduit Jelie. Le manuel de l'utilisateur, annexé en 1, en décrit l'utilisation. Le présent chapitre décrit son fonctionnement interne. Enfin, le manuel du programmeur (annexe 2) documente les classes et méthodes de Jelie.

Nous avons réalisé Jelie pour pouvoir utiliser Armonie. Jelie pilote un processeur XScale par son port JTAG. Nous ne l'avons testé que sur des PXA250, mais il fonctionne probablement sur d'autres circuits de la famille XScale[1].

Nous avons choisi de le programmer sous Linux, avec une interface en ligne de commande. Cet environnement nous convient et permet une grande efficacité d'utilisation, particulièrement quand il s'agit d'automatiser des processus.

Jelie s'utilise aussi comme couche d'accès au PXA250 pour GDB. La communication TCP/IP avec GDB que nous avons mise en oeuvre offre une utilisation performante et souple.

Jelie est conçu pour être portable et fonctionne aussi bien sur des systèmes petit-boutiens que gros-boutiens[2] *(little endian* et *big endian*). L'utilisation des programmes automake et autoconf facilite la compilation sur des systèmes différents en adaptant le code source aux ressources disponibles.

Jelie se nomme ansi parce qu'il relie l'ordinateur de développement au PXA250. Il s'appelait lors de notre projet de semestre *jtag-usb*, nom bien maladroit puisque Jelie peut s'utiliser par port parallèle.

6.1 Conception objet

L'objectif de Jelie est de piloter un XScale par port JTAG. En plus, Jelie doit communiquer avec l'utilisateur par une ligne de commande, et avec GDB par TCP/IP. Un adaptateur est nécessaire entre l'ordinateur de développement et le port JTAG du PXA250. Puisque plusieurs adaptateurs existent (parallèle ou USB), il faut une abstraction de ce port.

Pour réaliser Jelie, nous avons adopté une conception orientée objet dont la suite de cette section donne une vue d'ensemble.

Une classe d'abstraction de port JTAG (`JTAGControl`) permet de travailler sans se soucier du support physique qui relie l'ordinateur de développement au port JTAG du PXA250.

La classe `JTAGpxa250` contient les commandes abstraites JTAG nécessaires pour piloter un PXA250. C'est cette dernière qu'utilisent les classes de ligne de commande et de communication avec GDB (`CmdLine` et `GdbRemote`) pour exécuter les ordres de l'utilisateur. Cette structure est illustrée par la figure 6.1.

Les classes `PPJTAG` et `EzUSBJTAG` implémentent l'interface `JTAGControl`. Ces deux classes représentent des contrôleurs de port JTAG matériel, l'un via un adaptateur branché sur le port parallèle, et l'autre

[1] par exemple PXA210, PXA261, PXA262, IXP425, IXP2400, IXP2800, 80200.

[2] Architecture informatique où l'octet le plus significatif des valeurs numériques multioctets est stocké en premier. L'image est empruntée aux "Voyages de Gulliver" de Jonathan Swift, dans lequel les Gros-Boutiens, partisans de la théorie selon laquelle il faut casser un oeuf par le gros bout, s'opposent aux Petits-Boutiens qui, eux, soutiennent qu'il faut, au contraire, casser un oeuf par le petit bout. Définition tirée de : `http://iquebec.ifrance.com/hapax/glossaire.htm`.

via une carte EzUSB branchée sur port USB.

La classe `EventSelector` centralise puis redistribue les événements provenant de ces différentes sources :

- flux d'entrée standard provenant de l'utilisateur (ligne de commande),
- port TCP/IP par lequel GDB communique,
- port JTAG d'où proviennent les commandes du gestionnaire de débogage embarqué sur le PXA250.

Les événements sont ensuite redistribués aux classes `CmdLine`, `GdbRemote` et `JTAGpxa250` en vue de leur traitement.

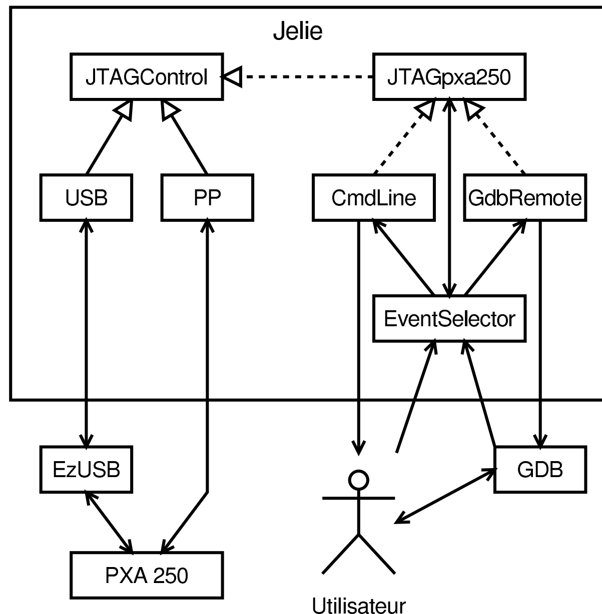

FIG. 6.1 – Structure interne de Jelie.

6.2 Fonctions JTAG

Le PXA250 possède des fonctions JTAG prévues pour mettre un oeuvre un débogueur. Nous utilisons principalement les deux fonctions suivantes :

- chargement du mini-cache d'instructions,
- communication avec un logiciel embarqué (le gestionnaire de débogage).

Les sections 5.4 et 5.5 du rapport de notre projet de semestre (annexe 6) détaillent le processus que Jelie utilise pour charger le mini-cache d'instructions. C'est pouquoi la présente section passe directement à la communication avec le gestionnaire de débogage.

Deux regitres JTAG de données de 32 bits et un registre de contrôle[3] permettent au gestionnaire de débogage et à l'ordinateur de développement de se synchroniser et de communiquer.

Quand le PXA250 veut envoyer un mot de 32 bits par port JTAG, il attend que le registre de communication soit libre, en lisant le bit `TR` (*TX Register Ready Bit*) du registre de contrôle de communication (`TXRXCTRL`). Il écrit ensuite la donnée dans le registre `TX`. La matériel va ensuite mettre à 1 un bit signalant au destinataire que les données contenues dans `TX` sont valides. L'opération de lecture va remettre ce bit à 0.

La communication dans l'autre direction – de l'ordinateur de développement vers le PXA250 – s'effectue d'une manière similaire, avec le registre `RX` et le bit `RR` du registre de contrôle. Un détail change pourtant : l'hôte n'est pas forcé, pour envoyer une donnée, d'attendre que le registre `RX` soit libre. Il peut envoyer une

[3]ces registres s'appellent `TX`, `RX` et `TXRXCTRL`.

séquence de mots. Il y a dans ce cas sur le PXA250 un mécanisme qui permet de détecter s'il n'arrive pas à lire les données suffisamment vite.

Ce système permet de gagner pas mal de temps. En effet, si, pour envoyer un mot, il faut attendre l'information qui indique si le registre et libre ou pas, beaucoup de temps est perdu car il faut traverser deux fois le système (ce qui est vraiment critique dans le cas de l'USB). L'autre possibilité consiste à envoyer les données en rafales, sans attendre de réponse. Cette solution permet d'utiliser beaucoup mieux la bande passante disponible.

La figure 6.2 illustre ces deux cas. Du côté ordinateur de développement, ce sont les méthodes `readTX`, `writeRX` et `fastWriteRX` de la classe `JTAGpxa250` qui se chargent de la communication. La fonction `writeRX` effectue une écriture avec relecture du registre JTAG, alors que `fastWriteRX` écrit des données en rafale.

Les fonctions de communication du côté cible sont `readrx` et `writetx`[4]. On peut trouver plus de détails dans le manuel de l'utilisateur [2, § 10.7, page 10-11].

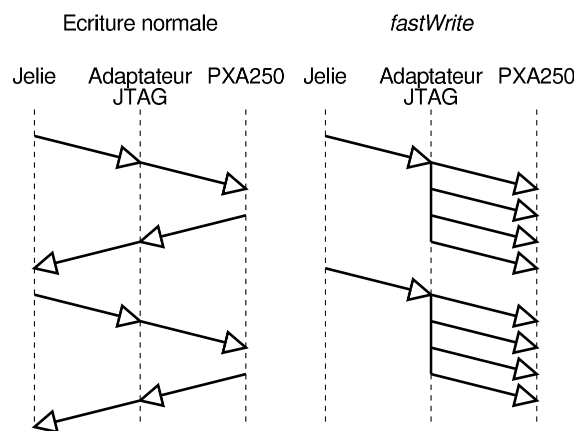

FIG. 6.2 – Deux manières d'envoyer des données par port JTAG : en relisant le contenu du registre ou en rafale.

L'écriture de données ne modifie aucun registre, et la lecture ne modifie que `R13`. Cette charactéristique permet au gestionnaire de débogage d'utiliser ce système pour sauvegarder les registres sur l'hôte afin de ne pas modifier le programme débogué.

Plus de détail à ce sujet, ainsi que les protocoles utilisés sont abordés par la section 6.5.3.

6.3 Contrôleur port parallèle

C'est le contrôleur le plus simple, il est implémenté dans la classe `PPJTAG`. Il ne fonctionne que sous Linux avec l'architecture x86. Il communique directement avec le port parallèle grâce aux macros `inb` et `outb`, qui donnent accès aux instructions assembleurs du même nom. Sur le port parallèle, un simple adaptateur de tension bidirectionnel[5] convertit les données du port vers le cable JTAG.

6.4 Contrôleur USB

Ce contrôleur se base sur une petite carte à microcontrôleur EzUSB développée au LAP. Ce microcontrôleur comprend le protocole USB et possède quelques entrées/sorties. Nous les utilisons pour implémenter le protocole JTAG.

[4]dans le fichier `Jelie/debugHandler/jtag_com.S`.

[5]Dans notre cas, le Wiggler de Macgraigor Systems. Mais pour ce qu'il contient, il est bien plus économique d'en faire un soi-même. L'adaptateur Alteraprog réalisé au LAP fournit presque les fonctionalités requises. Seul le redémarrage du processeur est impossible.

L'utilisation de l'USB est intéressante, car elle permet d'être indépendant de l'architecture et du système d'exploitation de la station de développement. En effet, la bibliothèque libusb[13] permet de faire abstraction du mécanisme interne de gestion de l'USB et présente au programmeur une interface claire et unifiée.

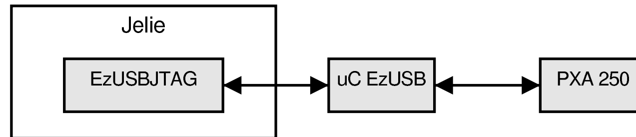

FIG. 6.3 – Chaîne de transmission JTAG par USB

Ce contrôleur est plus complexe que la version port parallèle. En effet, comme l'illustre la figure 6.3, toute commande JTAG doit d'abord passer le microcontrôleur, qui doit l'interpréter, effectuer les manipulations sur ses entrées/sorties, reconstruire le paquet USB, et l'envoyer. Or, la plupart des commandes JTAG sont à la fois une écriture et une lecture. Ainsi, devoir traverser deux fois la couche USB du PC et le code du microcontrôleur induit une latence importance, qui limite la vitesse de transfert réelle à une valeur bien inférieure du maxima théorique de l'USB.

L'optimisation décrite précédemment – qui consiste à ne faire que des écritures JTAGnon suivies de lectures – améliore l'efficacité d'un facteur deux, sans toutefois atteindre une vitesse comparable au port parallèle.

6.4.1 Détail du protocole

Dans sa version USB, Jelie communique avec la carte EzUSB par un protocole simple. On pourrait l'associer à une version primitive d'appel de fonction distante. Le PC initie la transmission en envoyant en mode *bulk* un appel de fonction. Le premier octet contient le numéro de la fonction. Le reste contient le contenu du message. Le microcontrôleur interprête le message, et, s'il y a lieu, renvoie une réponse. Le tout se fait à travers l'*endpoint* 2.

Les différents appels sont répertoriés ci-dessous. Entre parenthèses figure le nombre d'octets reçus et renvoyés par l'EzUSB. L'octet contenant le numéro de la fonction est inclus. Ils sont définis dans le fichier `jtag_order.h`.

JO_SELF_TEST (1, 1)

Teste la transmission et le microcontrôleur, renvoie un octet (0xBC) si tout est correct et génère un petit chenillard sur le port C de l'EzUSB.

JO_IREG (2, 1)

Ecrit et lit le registre `IREG` du JTAG. Ce registre contient 5 bits. Ainsi, un octet suffit pour le transférer.

JO_DREG (2+dataLength, dataLength)

Ecrit et lit le registre `DREG` du JTAG. Dans le cas du PXA250, ce registre contient de 1 à 64 bits. Il faut donc de 1 à 8 octets pour stocker cette donnée. La longueur en bits est donnée juste après le numéro de fonction. La taille du paquet dépend de dataLength, la longueur en octets, c'est à dire 1/8 de la longueur en bits, arrondie à l'entier supérieur.

JO_DREG_WRITE_ONLY (2+dataLength, 0)

Ecrit le registre `DREG` du JTAG. Dans le cas du PXA250, ce registre contient de 1 à 64 bits. Il faut donc de 1 à 8 octets pour stocker cette donnée. La longueur en bits est donnée juste après le numéro de fonction. La taille du paquet dépend de dataLength, la longueur en octet, c'est à dire 1/8 de la longueur en bits, arrondie à l'entier supérieur (il faut pouvoir stocker tous les bits).

JO_JTAG_RESET (1, 0)

Met la machine d'états JTAG dans l'état *idle*, quel que soit l'état dans lequel elle se trouve, en envoyant un 0 puis cinq 1 puis de nouveau 0 sur la ligne TMS.

JO_CPU_RESET (2, 0)

Active ou désactive le mode *reset* du CPU. L'octet juste après le numéro de fonction spécifie le nouvel état.

JO_TRST (2, 0)

Active ou désactive le ligne TRST du JTAG. L'octet juste après le numéro de fonction spécifie le nouvel état.

JO_IDLE (3, 0)

Génére des coups d'horloge JTAG. Les deux octets (petit boutiens) suivant le numéro de fonction spécifient le nombre de coups d'horloge à générer.

6.5 *Debug handler*

Le gestionnaire de débogage (*debug handler*) est un petit programme chargé par Jelie dans le minicache. Il a quatre utilités :

1. Tout d'abord, il initialise le processeur et les périphériques minimaux comme le contrôleur de mémoire vive.
2. Ensuite, il permet à l'ordinateur de développement de modifier l'état de la mémoire et des registres, par exemple pour charger un programme.
3. Il peut aussi, en coopération avec Jelie, lancer un programme, l'interrompre et reprendre son exécution de manière transparente.
4. Son dernier rôle est de signaler à l'utilisateur une exception en faisant clignoter des diodes.

Lorsque le gestionnaire de débogage interrompt l'exécution d'un programme, la mémoire n'est pas modifiée et les registres sont sauvegardés sur l'ordinateur de développement de manière à ne pas altérer le comportement du programme examiné. Ceci est possible grâce à un système – *Special Debug State* – qui, sous certaines contraintes, permet d'accéder à tous les registres, même banqués. Les sections 6.5.2 et 6.5.3 abordent ce sujet plus en détail.

Puisque le gestionnaire de débogage ne doit pas modifier la mémoire, il n'a pas de pile. Il est entièrement programmé en assembleur, et toutes les adresses de retour de fonction sont stockées dans des registres.

6.5.1 La table de vecteurs d'exceptions

Le gestionnaire de débogage installe en cache d'instruction une table de vecteurs d'exceptions ainsi que les routines pour les traiter, comme illustré par la table 6.1.

L'exception de démarrage (*reset*), à l'adresse zéro, sert aussi d'exception de débogage. La routine d'exception qui lui correspond utilise le bit d'activation global du registre de configuration et de status de débogage (`DCSR`, [2, §10.3]) pour déterminer si l'exception est un démarrage ou un événement de débogage.

Lors d'une première invocation de cette routine, avant de collaborer avec l'hôte, le gestionnaire de débogage active le bit d'activation globale de débogage et initialise les périphériques – phase qu'il sautera lors des appels suivants.

Les autres exceptions sont traitées de manière triviales : une routine s'occupe de faire clignoter une diode. On peut alors obtenir plus d'informations avec Jelie en interrompant la routine d'exception, puis en lisant dans le registre `R5` quelle est l'exception, et dans le registre `R14` l'adresse qui a causé l'exception[6].

[6] Pour plus d'information, le lecteur se reportera à la page A2-13 de l'ARM ARM [1].

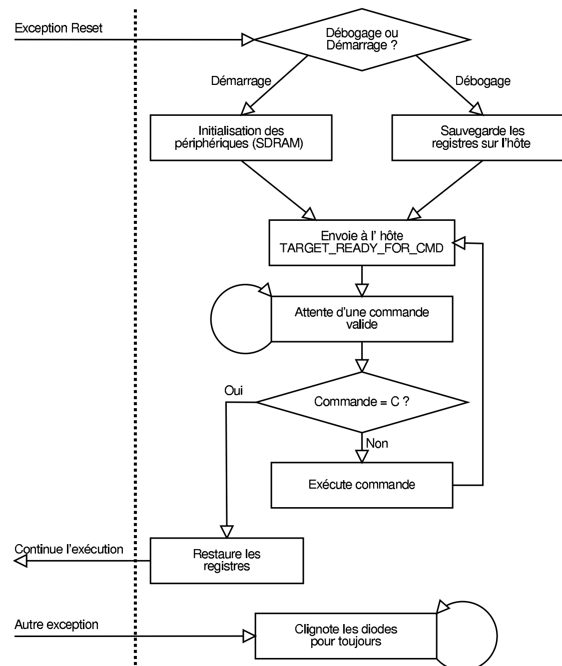

FIG. 6.4 – Diagramme de fonctionnement du gestionnaire de débogage.

Type d'exception	Mode	Addresse	Action
Reset / Debug	Superviseur	0x00000000	Initialisation / gestionnaire de debogue
Undefined instructions	Indéfini	0x00000004	Clignotement des diodes
Software interrupt (SWI)	Superviseur	0x00000008	Clignotement des diodes
Prefetch Abort	Abort	0x0000000C	Clignotement des diodes
Data Abort	Abort	0x00000010	Clignotement des diodes
IRQ (interrupt)	IRQ	0x00000018	Clignotement des diodes
FIQ (interrupt)	FIQ	0x0000001C	Clignotement des diodes

TAB. 6.1 – Table de vecteur d'exception ARM

Une amélioration possible consisterait à communiquer à Jelie, puis à GDB le fait qu'une interruption a eu lieu, par exemple à l'aide d'une intruction `bkpt` à l'intérieur des routines d'interruptions.

6.5.2 *Special Debug State*

Le mode SDS complète le mini-cache d'instructions pour permettre la réalisation complète d'un débogueur par port JTAG. Il permet :
- d'accéder à tous les registres de tous les modes de fonctionnement (superviseur, utilisateur, etc...),
- de changer de mode de fonctionnement,
- de contourner les mesures de protections de la mémoire et des registres des coprocesseurs.

Ce mode est activé après une exception de débogage, comme par exemple un démarrage en cache d'instructions géré par une connection JTAG. Il ne se configure pas via le registre de status (`CPSR`) comme les autres modes (superviseur, utilisateur, etc...). Par contre, il est quitté comme une interruption conventionnelle, grâce au bit S des instructions ARM, par exemple avec :

```
subs pc, lr, #0.
```

Dans ce mode, écrire dans le registre de status `CPSR` n'a qu'un seul effet : choisir les registres correspondants au mode choisi dans la banque de registres. Ce mécanisme permet d'accéder à tous les registres de tous les modes du processeur (utilisateur, interruption rapide, interruption, superviseur, annulation et système).

En mode SDS, le processeur ne fonctionne pas de manière habituelle. La documentation à ce sujet[7] laisse beaucoup de doutes et est modifiée par quelques erratas[8]. En particulier, les instructions d'accès multiples en mémoire ne fonctionnent pas, et deux instructions d'accès en mémoire ne peuvent pas se suivre directement.

6.5.3 Communication entre l'hôte et le gestionnaire de débogage

Le gestionnaire de débogage et Jelie peuvent communiquer à travers le port JTAG, par un mécanisme prévu à cet effet (voir section 6.2 page 31). La communication est bidirectionnelle. Cette section décrit les ordres que donne le gestionnaire de débogage à l'ordinateur de développement. La direction opposée sera abordée ensuite.

Jelie attend des événements provenant du gestionnaire de débogage via JTAG. Cette opération se fait par scrutation : Jelie lit le registre de communication JTAG périodiquement. De cette manière la cible peut donner des ordres à Jelie pour signaler une erreur, pour signaler à l'hôte que le gestionnaire de débogage est prêt ou pour sauvegarder et restaurer des registres sur l'hôte. Le fichier `debugHandler/target_to_host.h` décrit exhaustivement ce protocole alors que la table 6.2 en fait un résumé.

Partie haute bits 31-24	Partie basse bits 23-0	Nom	Signification
0x80	0 - 16	`SAVE_ON_HOST`	demande à l'hôte de sauvegarder un registre
0x40	0 - 16	`LOAD_FROM_HOST`	charge un registre depuis l'hôte
0x20	ignorée	`TARGET_READY_FOR_CMD`	le gestionnaire est prêt et attend un ordre
0x00	'I' (0x49)	`INVALID_COMMAND`	la cible à reçu un ordre invalide

TAB. 6.2 – Protocole de communication du gestionnaire de débogage vers Jelie.

Une fois que le gestionnaire de débogage a démarré, il lit un mot de 32 bits dans le registre `RX`. Il attend quatres commandes différentes de l'hôte sous la forme de la valeur ASCII des charactères suivants :

`c` continue l'exécution à l'adresse contenue dans le registre `R15` ;

`g` envoie un certain nombre de mots de 16 ou de 32 bits à une adresse fournie par l'hôte ;

[7] voir § 10.14.2.2 du manuel de l'utilisateur [2].

[8] voir entre autres errata 108 et 109 dans le document *Intel PXA250 and PXA210 Processors Specification Update*

`p` écrit en mémoire un certain nombre de mots de 16 ou de 32 bits donnés par l'hôte ;

`b` permet à l'ordinateur de développement d'écrire dans les registres du coprocesseur 15 (`IBCR0`, `IBCR1`, `DBR0`, `DBR1` et `DBCON` [2, § 10.2]).

Ces commandes sont suffisantes pour implémenter un débogueur. Le seul problème réside dans le fait que le PXA250 n'a aucun moyen d'exécuter un programme en pas à pas. Le débogueur doit, en logiciel, calculer l'adresse de l'instruction suivante et y mettre un point d'arrêt.

6.6 GNU automake et autoconf

Jelie utilise les outils GNU automake et autoconf pour préparer les sources avant la compilation. Ces outils donnent à Jelie beaucoup de portabilité et rendent sa compilation aisée.

Cette suite d'outils permet d'automatiser la détection du système et la configuration des sources en créant un script (`configure`) qui devra être exécuté juste avant la compilation. La figure 6.5 montre les programmes et fichiers impliqués dans la création de ce script.

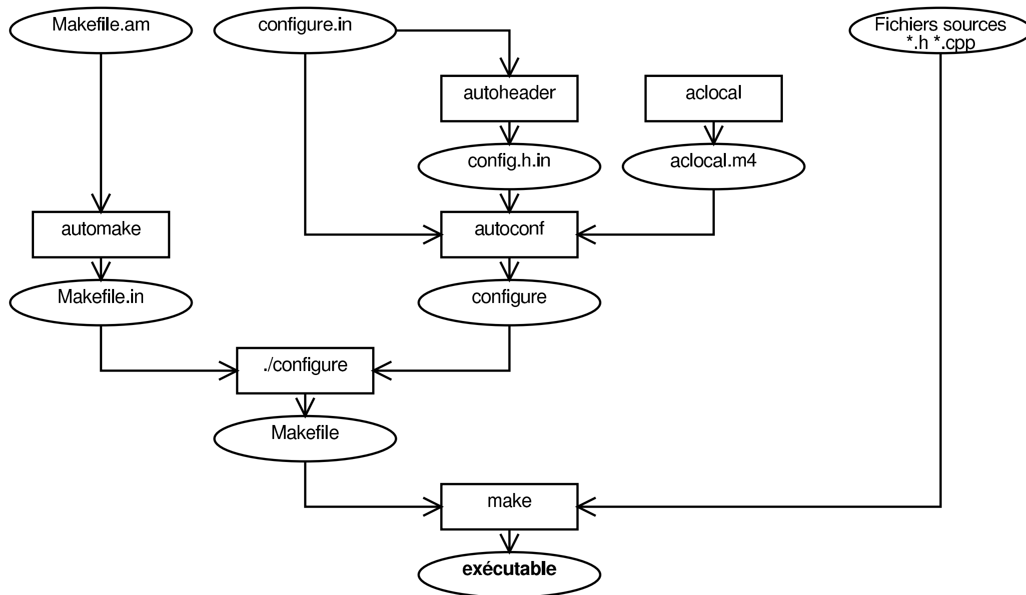

FIG. 6.5 – Organisation des fichiers utilisés par automake et autoconf.

Avant la compilation, il faut générer le script `configure` en appelant la commande :

```
./bootstrap
```

Ce script sera appelé par :

```
./configure
```

Cet appel aura les effets suivants :
- chercher un compilateur C++ ;
- si un compilateur croisé ARM est installé, prévoir les règles de compilation du gestionnaire de débogage embarqué sur le PXA250 ;
- si sdcc, un compilateur croisé pour microcontrôleur 8051, est disponible ; prévoir les règles de compilation du programme embarqué sur l'EzUSB ;
- configurer les sources en gros-boutien ou petit-boutien ;
- configurer la compilation du contrôleur USB si la bibliothèque `libusb` est présente ;
- configurer la compilation du contrôleur sur port parallèle si c'est possible ;
- configurer l'utilisation de la bibliothèque `libreadline` si elle est présente ;

– déterminer quel sera le répertoire d'installation pour que Jelie sache où trouver les programmes embarqués compilés ;
– effectuer quelques vérifications mineures.

Tout ces tests sont décrits dans le fichier `configure.in`. Du côté code de cette chaîne, c'est le fichier `config.h` qui contient, sous forme de définition de macros, une partie des résultats de l'exécution du script `configure`. La partie la plus importante est dans les fichiers `Makefile` qui contiennent les règles de compilation de Jelie.

Une fois les fichiers `Makefile` créés, il est possible de compiler Jelie par la commande :

```
make
```

Il est aussi possible d'installer Jelie sur le système, le rendant accessible à tous les utilisateurs, par :

```
make install
```

Il est possible de générer une distribution. Celle-ci sera composée des sources et du fichier `configure`. Ainsi, l'ordinateur de destination n'aura besoin que d'un interpréteur de commande compatible bourne, et évidemment des bons compilateurs et bibliothèques.

La distribution se présente sous la forme d'un fichier `.tar.gz`. Elle est crée en exécutante la commande :

```
make dist
```

Une documentation plus générale sur automake et sur autoconf est disponible sur internet [10, 11, 12].

6.7 Convivialité et ligne de commande

Nous pensons que convivialité et ligne de commande ne sont pas incompatibles. C'est pour cette raison que Jelie utilise la bibliothèque readline qui ne sert qu'à faciliter la vie de l'utilisateur.

Cette bibliothèque configure le terminal au mieux pour l'édition d'une commande sur une ou plusieurs lignes et fournit des fonctionnalités très pratiques : par exemple, la touche de flèche vers le haut inscrit sur la ligne la dernière commande saisie. Quand l'utilisateur appuie sur la touche tabulation après avoir écrit le début d'un nom de fichier, la bibliothèque readline va tenter de compléter ce nom en regardant les fichiers existants. Jelie affiche une invite qui indique si le gestionnaire de débogage est prêt ou non.

Si cette bibliothèque n'est pas présente ou ne fonctionne pas au moment de la compilation, le script de configuration des sources va automatiquement désactiver son utilisation. Dans ce cas, Jelie restera utilisable, bien que moins agréable.

L'utilisation de readline est désactivée au moment de l'exécution si le flux d'entrée standard n'est pas un terminal. Ceci permet l'utilisation de Jelie dans des scripts qui redirigent son entrée vers un fichier ou un programme.

Le manuel de Jelie (annexe 1) détaille l'utilisation de la ligne de commande.

6.8 Changements de fréquences

Nous distribuons avec Jelie des scripts et programmes conçus dans le but de changer les fréquences d'utilisation du coeur et du bus interne du PXA250.

Il y a deux méthodes principales pour changer les fréquences de fonctionnement. La première, relativement complexe, est la séquence de changement de fréquence (FCS, [3, §3.4.7]). La documentation d'Intel [3, §3.4.7.4] dit que lors de cette séquence, le processeur a besoin d'un événement externe pour se réveiller. Pourtant, sur les versions que nous avons testées, cet événement n'est pas nécessaire et le changement de fréquence se fait facilement.

La deuxième façon de changer la fréquence du coeur est plus simple ; elle est prévue pour une adaptation rapide à un changement de charge calcul. C'est le mode turbo. Il permet de multiplier la fréquence du coeur par un facteur de 1, 1.5, 2 ou 3.

Toutes les combinaisons possibles de fréquences sont résumées dans le tableau 6.3.

Les programmes du répertoire `jelie/freq` modifient le registre de configuration d'horloge (CCCR *Core Clock Configuration Register* [3, page 3-33]) avant d'appliquer les modifications via le registre 6 du co-processeur 14 (CCLKCFG [3, §3.7.1]). Trois rapports de fréquences sont à configurer : le facteur L entre le crystal et la fréquence de la mémoire, le facteur M entre la fréquence de la mémoire et celle du mode normal (*run mode*) et enfin le facteur N entre le mode normal et le mode turbo.

Pour le facteur L, une valeur de 27 est adaptée. Pour M, on peut choisir entre 1 et 2 pour un coeur à 100 MHz ou à 200 MHz. Pour les versions du processeur que nous avons, qui ne vont pas au-delà de 200 MHz, on ne peut pas choisir plus que 2 comme facteur de multiplication du mode turbo.

L	M	Turbo Mode Frequency (MHz) for Values "N" and Core Clock Configuration Register (CCCR[15:0]) programming for Values of "N":				PXbus Frequency	MEM, LCD Frequency (MHz)	SDRAM max Freq
		1.00 (Run)	1.50	2.00	3.00			
27	1	99.5 @.85v	—	199.1 @1.0 V	298.6 @1.1v	50	99.5	99.5
32	1	118.0 @1.0v	—	235.9 @1.1 V	353.9 @1.3v	59	118.0	59.0
36	1	132.7 @1.0v	—	265.4 @1.1v	398.1 @1.3v	66	132.7	66
40	1	147.5 @1.0v	—	294.9 @1.1v	—	74	147.5	74
45	1	165.9 1.0v	—	331.8 1.3v	—	83	165.9	83
27	2	199.1 @1.0v	298.6 @1.1v	398.1 @1.3v	—	99.5	99.5	99.5
32	2	235.9 @1.1v	—	—	—	118	118.0	59.0
36	2	265.4 @1.1v	—	—	—	132.7	132.7	66
40	2	294.9 @1.1v	—	—	—	147.5	147.5	74
45	2	331.9 @1.3v	—	—	—	165.9	165.9	83

TAB. 6.3 – Configurations de fréquences du PXA250 (tiré du tableau 3-1, page 3-5 du manuel du développeur [3]).

6.9 Améliorations possibles

Bien que Jelie soit totalement fonctionnel, il reste toujours de la place pour des évolutions futures. Comme Jelie est codé proprement en bon orienté objet (C++), il est relativement aisé d'effectuer des changements. Nous allons faire un tour d'horizon de ce qui pourrait être amélioré.

6.9.1 Contrôleurs JTAG

On pourrait facilement concevoir une interface physique JTAG plus rapide. Par exemple, il y a au LAP un analyseur logique USB, créé par Sebastian Gerlach, contenant une logique programmable et le même microcontrôleur EzUSB que dans la solution USB actuelle. Pour attaquer le port JTAG, il serait tout à fait possible d'implémenter un registre à décalage dans la logique programable pour alléger le travail du microcontrôleur. Cette solution améliorerait grandement la vitesse de transfert.

Chapitre 7

Linux sur Armonie

Ce chapitre, séparé en deux parties, présente la compilation et le fonctionnement du noyau Linux sur Armonie. Puis il illustre l'organisation d'un système d'exploitation complet basé sur ce noyau.

7.1 La communauté Linux ARM

Il y a une communauté très active dans le domaine de l'embarqué avec Linux, notamment sur l'architecture ARM.

Le site web The ARM Linux Project[1] est une source précieuse de codes, de documentations, de conseils et d'expérience. Il fournit entre autres des listes de dicussions (*mailing lists*) de grande qualité, où la plupart des problèmes que nous avons rencontrés ont été discutés. Ces listes nous ont donc été d'une grand aide. Elle sont :

- linux-arm-announce@arm.linux.org.uk, annonces ;
- linux-arm-kernel@arm.linux.org.uk, discussion sur les questions liées au noyau ;
- linux-arm@arm.linux.org.uk, discussion générale.

Le site suivant donne plus d'information :

`http://lists.arm.linux.org.uk/mailman/listinfo/`

7.2 Le noyau Linux

Nous utilisons la version 2.4.19 de Linux sur Armonie. Sur cette version, nous avons appliqué les modifications de Russell King (rmk4) et de Nicolas Pitre (pxa2), respectivement pour avoir un meilleur support ARM et un support PXA250. La version résultante se nomme linux-2.4.19-rmk4-pxa2.

Comme point de départ, nous avons pris les fichiers conçus pour la plateforme de développement d'Intel, dont le nom de code est *lubbock*. Nous leurs avons appliqué nos modifications personnelles afin d'avoir un support adapté à Armonie.

7.2.1 Démarrage du noyau, plan mémoire et MMU

Pour démarrer Linux, il faut effectuer les opérations suivantes :

1. charger l'image compressée du noyau en mémoire,
2. charger l'image compressée du système de fichier en mémoire,
3. préparer les registres avec les bons paramètres pour le noyau,
4. sauter à la première instruction du noyau compressé.

[1] `http://www.arm.linux.org.uk/`

A ce moment, l'image du noyau est décompressée puis appelée. Le script Jelie `loadlinux` montré sur la figure 7.1 exécute les quatres points précédents. La figure 7.2 résume l'organisation de la mémoire vive lors d'un démarrage de Linux.

Quand le noyau commence à s'initialiser, il va tout d'abord activer l'unité de gestion de mémoire (MMU). Ceci a pour effet de rendre impossible le débogage avec Jelie puisque Linux déplace et remplace les vecteurs d'interruptions, entre autres celui correspondant aux événements de débogage.

```
# reload the debug handler
stop
loadic debugHandler/debug_handler.bin 0 mini
boot

# set to 0 the usb connect because we are not ready to handle the usb
# connection yet.
put 40e00028 10

# load the compressed kernel image
load /home/jpilet/linux/linux/arch/arm/boot/zImage a1000000

# load the minimalist filesystem image
load /home/pxa/work/linux/initrd.ext2-busybox.gz a2000000

# set the correct parameters for the kernel
register 0 0
# it's an armonie machine (0x108 = 264)
register 1 108
# filesystem address
register 2 a2000000

# boot the kernel
register 15 a1000000
continue
```

FIG. 7.1 – Exemple de script Jelie qui démarre Linux sur Armonie.

7.2.2 Sources et compilation

Les sources du noyau se trouvent dans le répertoire `armlinux/linux-2.4.19-rmk4-pxa2-armonie1`. Plusieurs fichiers et sous-répertoires sont à remarquer :

`Makefile` configure la compilation et contient entre autres le nom du compilateur croisé à utiliser.

`.config` contient la configuration actuelle du noyau. A éditer avec `make menuconfig`.

`arch/arm/mach-pxa` ce répertoire contient le code spécifique au PXA250.

`arch/arm/mach-pxa/armonie.c` code d'initialisation spécifique à la carte Armonie.

`include/asm-arm/arch-pxa` fichiers d'en-tête spécifiques au PXA250, dont `armonie.h`.

`drivers/video/pxafb.c` pilote LCD pour PXA250.

`drivers/pcmcia/pxa/` pilote CompactFlash et PCMCIA pour PXA250 (ne fonctionne actuellement pas avec Armonie).

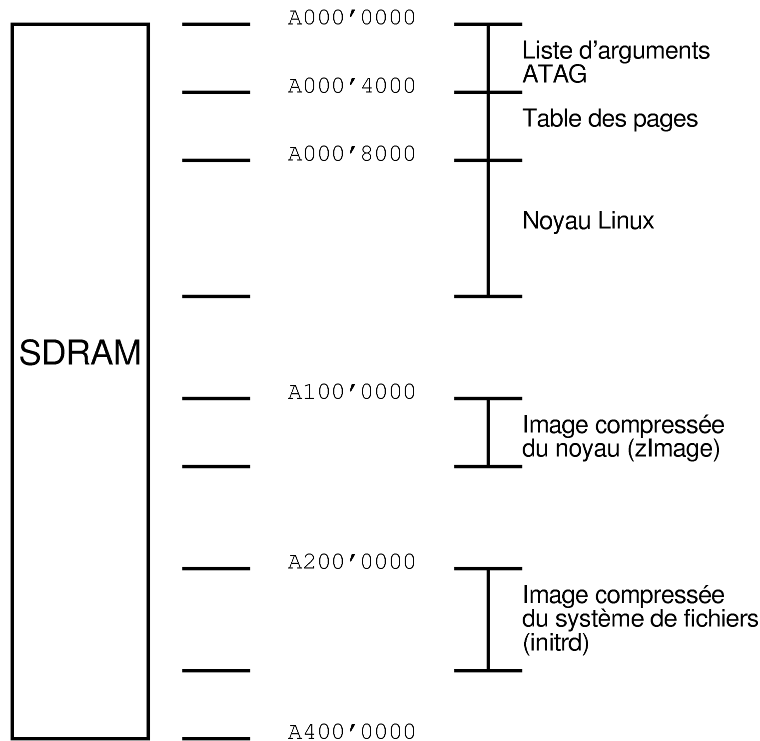

FIG. 7.2 – Plan mémoire au chargement de Linux

7.2.3 Les modifications apportées au noyau

Dans le monde ARM Linux, chaque machine possède un numéro unique[2]. Armonie a le numéro 264. Grâce à ce numéro, il est possible de définir une nouvelle machine pour Linux. C'est ce que nous avons fait, dans le fichier `armonie.c`. Ce fichier décrit l'assignation des GPIO, leurs éventuels fonctionnements en mode alternatif, la carte de la mémoire, et d'autres paramètres dépendants non pas du processeur, mais de la machine.

Le nom de nos modifications sur le noyau Linux est armonie1. Ainsi, le nom complet de notre version de Linux est linux-2.4.19-rmk4-pxa2-armonie1.

7.2.4 Configuration et compilation

Les explications des pilotes seront relatives à `make menuconfig`, car c'est une solution à la fois simple, intuitive et efficace à la configuration.

Pour que Linux puisse afficher des informations de debogage et pour trouver l'image d'un système de fichier, les informations relatives à la console, à la mémoire et à la position du `initrd` doivent être mise dans une chaîne de commande. Pour ce faire, il faut, lors du `make menuconfig`, aller dans *General Setup*, et éditer l'entrée nommée *Default kernel command string*. Pour Armonie, la chaîne de commande suivante est adaptée :

```
console=ttyS0,115200 mem=64M initrd=0xa2000000,223458 root=/dev/ram0
```

Cette chaîne contient les commandes suivantes :
- `console=ttyS0,115200` active la première UART en 115200 bps,
- `mem=64M` taille de la mémoire SDRAM,

[2]La liste complète peut être consultée à l'adresse suivante `http://www.arm.linux.org.uk/developer/machines/`

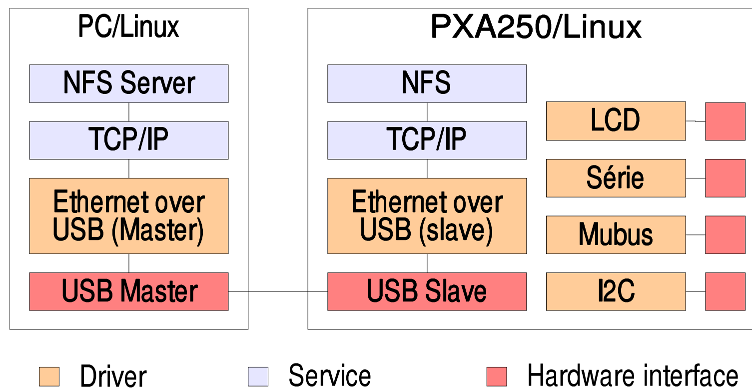

FIG. 7.3 – Linux sur Armonie

- `initrd=0xa2000000,223458` emplacement et taille de l'initrd (image d'un système de fichier en mémoire vive),
- `root=/dev/ram0` utilise l'initrd comme système de fichier racine.

Après la configuration, il suffit de taper `make dep zImage` pour construire une image compressée utilisable.

En plus du système de base, qui nous permet d'avoir une ligne de commande par le port série, nous voulons supporter un certain nombre des périphériques intégrés au PXA250. Les sections qui suivent décrivent la création, la configuration et l'utilisation de ces derniers.

7.2.5 Le Mubus

Le PXA250 fournit plusieurs signaux de sélection de circuit (*chip select*) qui permettent d'interfacer des périphériques visibles en mémoire. Afin de tester ceci, et aussi d'avoir à disposition une interface très simple largement utilisée au LAP, nous avons implémenté une interface Mubus sur Armonie.

Le manuel du pilote Mubus pour Linux (annexe 3) explique comment compiler le module `mubus.o` et comment l'insérer dans le noyau. Il documente également son utilisation du point de vue du programmeur (API d'entrée/sortie de bas niveau).

7.2.5.1 Fonctionnement d'un pilote sous Linux

Sous Linux, tout accès à un périphérique se fait à travers un fichier. Pour écrire ou lire le Mubus, il faut au préalable ouvrir le fichier `/dev/mubus` avec l'appel système *open*. Une fois qu'il est ouvert, il est possible d'écrire dedans avec *write*, de lire avec *read*, et de le fermer avec *close*[3].

L'accès à un périphérique par *write* et *read* est facile et intuitif, mais nécessite un appel de fonction pour chaque accès. Cela peut devenir très lent s'il faut faire un grand nombre d'accès. L'appel système *mmap* permet de rendre visible une zone de mémoire périphérique dans l'espace d'addressage d'un processus. Ainsi, l'accès est direct et ne nécessite pas l'intervention du système d'exploitation. Le pilote Mubus fournit cette fonctionnalité.

Pour plus d'informations sur le déploiement du pilote Mubus, veuillez consulter le manuel du pilote Mubus pour Linux, en annexe 3 de ce document.

7.2.5.2 Fonctionnement interne du pilote Mubus

Le pilote n'est pas très complexe. Il commence, lors d'une phase d'initialisation, par rendre accessible la zone de mémoire physique correspondant au Mubus depuis la mémoire virtuelle, en désactivant le cache – grâce à la fonction `ioremap_nocache()`.

[3] Pour plus d'information sur les différents appels système, il suffit, sous Linux, d'utiliser la commande man. Par exemple, `man 2 open` pour l'appel *open*. Le manuel du pilote Mubus (annexe 3) décrit précisement l'utilisation de l'interface Mubus sous Linux.

Ensuite, dès qu'un programme utilisateur fait une lecture ou une écriture, le pilote échange les données entre l'espace noyau, depuis lequel le Mubus est accessible, et l'espace utilisateur où le Mubus n'est pas visible. Les fonctions `copy_to_user()` et `copy_from_user()` effectuent cette tâche.

Pour rendre le Mubus visible à une application, l'appel système *mmap* invoque la fonction du noyau `remap_page_range()`. Une fois cette opération effectuée, le programme utilisateur peut accéder au Mubus sans intervention du pilote.

7.2.6 Le bus I^{2}C

Le PXA250 contient un contrôleur maître I^{2}C. Par chance, le pilote Linux I^{2}C pour le PXA250 est déjà écrit. L'I^{2}C est un bus, il connecte donc des périphériques au processeur.

Au démarrage, le pilote I^{2}C énumère les périphériques présents sur le bus. Or, ces périphériques ont eux aussi besoin de pilotes. Linux fournit deux méthodes pour ces derniers :

1. Il est possible d'avoir un pilote en mode noyau, qui est initialisé à l'énumération du bus, et qui fournit d'autres services. Nous n'avons pas exploré cette voie car nous n'avions pas de périphériques ayant un pilote spécifique.
2. Une interface (*I2C device interface*) permet à un programme en espace utilisateur d'accéder aux périphériques présents sur le bus. Tout d'abord, l'application ouvre le bus (`/dev/i2c-0`) avec l'appel système *open*. Puis, avec l'appel système *ioctl*, avec comme premier paramètre `I2C_SLAVE` et comme second l'addresse, elle sélectionne le périphérique désiré. Puis elle peut écrire et lire des données avec les appels systèmes *write* et *read*.

Le bus I^{2}C a été testé sous Linux avec un registre parallèle (PCF8574 de Philips) et fonctionne bien. Une petite application de test a été écrite à cet effet. Nommée testi2c.c, elle permet d'écrire puis de relire un octet depuis le registre parallèle. Son code est montré par la figure **??**.

Pour activer l'I^{2}C lors de la configuration du noyau correctement modifié, il faut aller dans *Character device - I2C support*, puis activer *I2C support*, *PXA I2C Algorithm*, *PXA I2C Adapter*, *I2C device interface*, *I2C /proc interface*. Le noyau, une fois compilé, contiendra le support I^{2}C. Le noyau distribué avec ce rapport contient le support I^{2}C.

7.2.7 Le CompactFlash

Le compact flash ne fonctionne pas. Le temps nous a manqué pour terminer le pilote Linux pour Armonie. Le terrain est quand même défriché : le fichier `drivers/pcmcia/pxa/armonie.c` contient les bases des modifications nécessaires. Actuellement, nous ignorons où se trouve le problème.

Des modifications matérielles seraient souhaitables. Pouvoir contrôler l'alimentation du port CompactFlash semble utile. L'annexe 11 propose des modifications.

7.2.8 L'écran LCD

Linux fournit un pilote pour le contrôleur LCD du PXA250, ce dernier peut-être exploité par la carte d'extension eMediaKit comme expliqué au chapitre 8, dans la section 8.2.5.

Le pilote de ce contrôleur est implémenté dans les fichiers `pxafb.c` et `pxafb.h`, présents dans le répertoire `drivers/video` des sources modifiées.

Néanmoins, ce pilote code en dur la synchronisation temporelle des signaux en fonction de la carte. Nous avons donc dû rajouter les informations relatives à notre écran (320x240x16 bpp). Cela consiste à écrire un certain nombre de `#define` dans le `.h`, afin de caractériser nos *timings*. Ils sont activés si la macro `CONFIG_ARCH_ARMONIE` est vraie, c'est à dire si l'on est en train de compiler un noyau pour Armonie. Idéalement, le fichier `.c` ne devrait pas être modifié. Malheureusement, la premiere version du pilote n'implémentait qu'un seul type d'écran, et n'utilisait même pas les `#define`. Pour garder la compatibilité ascendante, il y a aussi un test de la macro dans le `.c`.

Il est intéressant de constater que ce pilote ne fait pas grand chose. Essentiellement, il met les constantes définies par le programmeur dans les quatre registres principaux de configuration du LCD du PXA250. De

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>

int main(int argc, char *argv[])
{
        int file;
        int addr;
        char val;
        char buf[1];

        if ((file = open("/dev/i2c-0",O_RDWR)) < 0)
        {
                /* ERROR HANDLING; you can check errno to see what went wrong */
                perror("/dev/i2c-0");
                exit(1);
        }

        if (argc == 1)
        {
                addr=0x40; /* The I2C address */
                val=0x55;
        }
        else if (argc == 2)
        {
                addr=atoi(argv[1]);
                val=0x55;
        }
        else
        {
                addr=atoi(argv[1]);
                val=atoi(argv[2]);
        }

        buf[0]=val;

        if (ioctl(file,I2C_SLAVE,addr) < 0)
        {
                /* ERROR HANDLING; you can check errno to see what went wrong */
                perror("/dev/i2c-0 : ioctl");
                exit(2);
        }
        printf("Successfully selected slave 0x%x\n", addr);

        if (write(file, buf, 1) != 1)
        {
                /* ERROR HANDLING: i2c transaction failed */
                perror("/dev/i2c-0 : write");
                exit(3);
        }
        printf("Successfully written %c (%d)\n", buf[0], buf[0]);

        if (read(file, buf, 1) != 1)
        {
                /* ERROR HANDLING: i2c transaction failed */
                perror("/dev/i2c-0 : read");
                exit(4);
        }
        printf("Successfully read %c (%d)\n", buf[0], buf[0]);
}
```

FIG. 7.4 – Programme de test de l'I^{2}C

plus, il fait l'interface avec le reste du noyau, notamment en ce qui concerne la gestion d'énergie (désactivation de l'écran après un temps donné).

Pour activer l'écran LCD lors de la configuration du noyau correctement modifié, il faut aller dans *Console drivers* - *Frame-buffer support*, puis activer *Support for frame buffer devices* et *PXA LCD support*. A noter que ces options n'apparaissent que si *Character devices* - *Virtual terminal* est activé au préalable.

Linux a un système d'économie d'énergie qui éteint un terminal virtuel lorsqu'il n'est pas utilisé pendant un certain temps – 5 minutes par défaut. Nous avons écrit un programme, `fbcontrol`, qui permet de changer ce temps. Il permet aussi d'activer et de désactiver l'affichage du curseur. Ce programme fonctionne en écrivant dans le fichier `/dev/ttyS0` des séquences de carractères qui contrôlent la console.

Lorsque l'affichage sur l'écran est désactivé, le DMA n'est pas utilisé et la zône mémoire qui contenait l'image à afficher n'est plus lue. La section **??** décrit l'utilisation de fbcontrol.

7.2.9 Réseau via USB

Le PXA250 possède un contrôler USB esclave. Avec le concours de deux pilotes, un sur le PC et un sur le PXA250, Linux est capable de générer un lien Ethernet à travers l'USB en créant une interface réseau de chaque côté.

Pour que cela fonctionne, il faut effectuer quelques petites modifications :

- Tout d'abord, côté PC, il faut inverser deux valeurs, car il y a un changement subtile dans le contrôleur esclave du PXA250 par rapport au StrongARM, base sur laquelle le pilote PC a été écrit. Dans le fichier `drivers/usb/usbnet.c`, la structure `linuxdev_info` relative à l'ARM (avec comme description "Linux Device", ligne 703 du noyau 2.4.20) contient deux valeurs entières, `.in = 2` et `.out = 1`, qu'il faut inverser de façon à avoir `.in = 1` et `.out = 2`.
- Ensuite, il faut légèrement modifier le code du contrôleur USB esclave du PXA250. Il s'agit du fichier `sb_ctl.c`, dans le répertoire `arch/arm/mach-pxa` des sources de Linux. Armonie est capable, grâce à une GPIO, de simuler une déconnection sur l'USB. Le contrôleur supportait le concept, mais l'appel était fait au mauvais moment. Nous l'avons donc changé et nous avons rajouté le support Aromonie.

Une fois ces deux modifications effectuées, tout le code est en place pour activer le réseau par USB. Pour l'activer, il faut taper la commande suivante sur Armonie :

```
ifconfig usbf 192.168.0.2
```

Et sur le PC :

```
ifconfig usb0 192.168.0.1
```

7.3 Le système d'exploitation

Avoir un noyau n'est pas suffisant pour pouvoir travailler. Il faut aussi un ensemble de programmes fonctionnants en mode utilisateur, apportants les fonctionnalités voulues. Ceci est possible grâce à une image d'un disque en mémoire vive. Sous Linux, le démarrage à partir d'image disque en mémoire s'appelle `initrd`[15].

7.3.1 Busybox

Il est agréable que l'image compressée contienne un minimum de fonctionnalités d'un Unix standard. Si les versions standards GNU de ces logiciels étaient utilisées, comme c'est le cas avec un GNU/Linux normal, l'image serait vite grosse. C'est pourquoi busybox a été créé[16].

L'idée est d'avoir un seul exécutable regroupant plusieurs commandes, qui a un comportement différent suivant le nom par lequel il est appelé (`argv[0]` en C). Ainsi, en faisant des liens symboliques des différents programmes vers busybox, il est possible de simuler un système Unix complet.

Busybox a besoin d'une bibliothèque C (*lib c*). Néanmoins, il peut être lié avec une version minimale, μClibc[17], afin d'optimiser la taille résultante. Dans notre cas, busybox est lié à μClibc de façon statique.

L'exécutable de busybox que nous utilisons fait 385.1 ko et sait se comporter comme presque 100 commandes. Celles-ci sont listées sur la figure 7.4.

```
BusyBox v0.60.5 (2002.11.15-12:19+0000) multi-call binary

Usage: busybox [function] [arguments]...
   or : [ function ] [ arguments]...

        BusyBox is a multi-call binary that combines many common Unix
         utilities into a single executable. Most people will create a
        link to busybox for each function they wish to use, and BusyBox
        will act like whatever it was invoked as.

Currently defined functions:
        [, ash, basename, busybox, cat, chgrp, chmod, chown, chroot, chvt,
        clear, cp, cut, date, dd, df, dirname, dmesg, du, echo, env, false,
        find, free, grep, gunzip, gzip, halt, head, hostid, hostname,
        id, ifconfig, init, kill, killall, klogd, linuxrc, ln, logger,
        ls, lsmod, makedevs, md5sum, mkdir, mknod, modprobe, more, mount,
        mv, nslookup, pidof, ping, poweroff, ps, pwd, reboot, reset, rm,
        rmdir, route, sed, sh, sleep, sort, sync, syslogd, tail, tar,
        telnet, test, tftp, top, touch, traceroute, true, tty, umount,
        uname, uniq, uptime, vi, wc, wget, which, whoami, xargs, yes,
        zcat
```

FIG. 7.5 – Liste des commandes supportées par busybox.

7.3.2 Démarrage

Notre image contient donc busybox, lié avec μClibc, configuré pour fournir la majorité des commandes de base.

Au démarrage, l'image compressée du noyau se décompresse, se charge, initialise les périphériques, décompresse et charge l'image compressée du système de fichier puis exécute `/sbin/init`.

`/sbin/init` va lire le contenu du fichier `/etc/inittab` qui spécifie de façon standardisée quels programmes doivent être lancés et arrêtés, et à quel moment.

Ensuite, un script de démarrage va initialiser le réseau à travers l'USB, monter `/proc` afin d'avoir des informations sur le système et lancer une invitation de ligne de commande.

Cette dernière, ainsi que tous les messages du noyau, sont accessibles sur la console, c'est-à-dire le premier port série (connecteur J2 sur Armonie).

Chapitre 8

Système multimédia embarqué

Ce chapitre présente l'application système multimédia embarqué et détaille sa conception.

8.1 Présentation

Une fois Armonie fonctionnelle, nous avons eu au cours de ce projet l'idée de réaliser un système multimédia embarqué qui se présente sous la forme d'une boîte de la taille d'un livre. Ce système autonome possède des ports destinés à l'enseignement ou à de futures extensions.

Les critères de choix de conception et de dimensionnement sont les suivants :

- durée honnête d'utilisation sur batterie (10 h sans rétro-éclairage de l'écran LCD et 4h avec rétro-éclairage 50% du temps),
- modularité,
- simplicité de conception,
- solidité,
- fonctionnalité,
- poids et mobilité.

Le système final se compose de trois cartes, comme le montre la figure 8.2 :

- carte processeur Armonie (PXA250) ou RokEPXA (ARM + FPGA) ;
- carte d'alimentation intelligente, avec chargeur de batteries [1] ;
- carte d'extension multimédia eMediaKit, avec écran LCD TFT couleur 320x240, surface tactile et boutons, entrées/sorties son, réseau et contrôleur USB maître.

Un boitier viendra compléter le tout. Il permettra d'avoir un vrai système embarqué, comme illustré sur la figure 8.1.

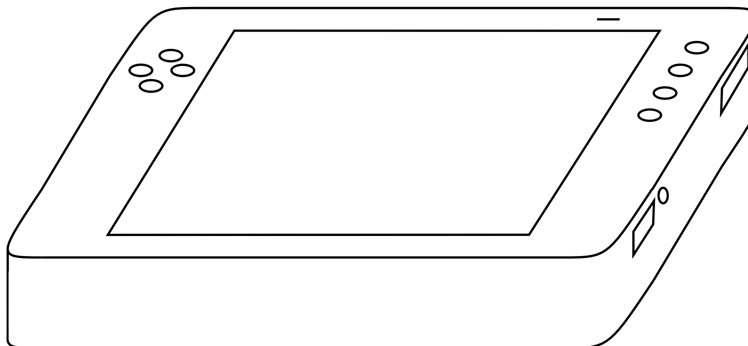

FIG. 8.1 – Dessin du système multimédia embarqué construit

[1] Cette carte est développée au LAP par le projet de semestre d'Adrian Spycher

Actuellement, tous les circuits imprimés existent. La carte Armonie est fonctionnelle. La carte d'alimentation est capable d'alimenter les cartes processeurs. La carte eMediaKit fonctionne partiellement.

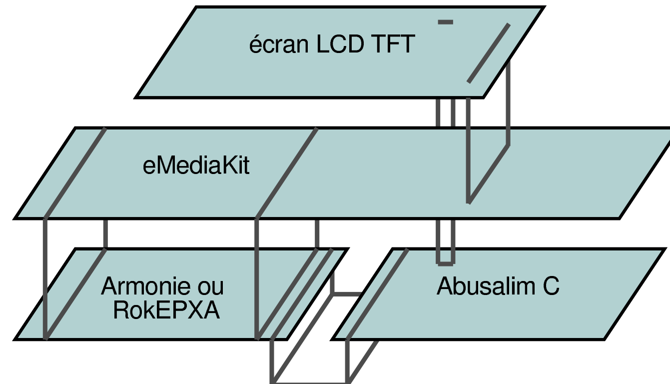

FIG. 8.2 – Schéma bloc des circuits imprimés composant le système multimédia embarqué

8.2 Conception eMediaKit

La carte eMediaKit a été développée afin de pouvoir connecter un écran LCD et d'avoir un riche ensemble de périphérique sur le bus Milli-BUS. La figure 8.3 en présente une vue synthétique.

Dans notre cas, ces périphériques s'intégrent dans l'espace mémoire d'Armonie comme le montre la figure 8.4.

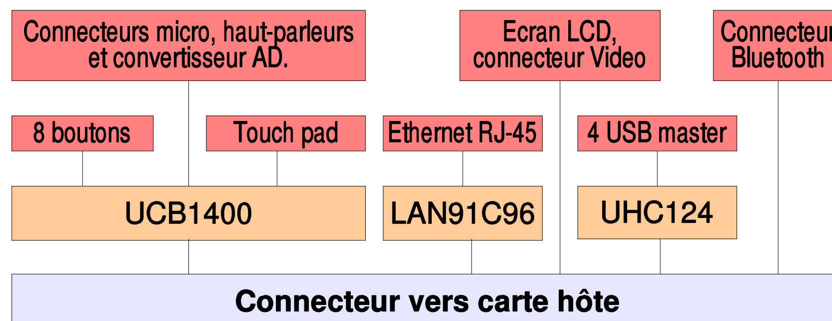

FIG. 8.3 – Schéma bloc eMediaKit

8.2.1 Disposition physique

La partie supérieure du système multimédia embarqué est principalement occupée par l'écran tactile horizontal. La croix directionelle est située à gauche, les boutons génériques sont situés à droite. A gauche, il y a aussi le connecteur Ethernet, les 4 connecteurs USB maître, et l'entrée son. A droite, il y a le connecteur pour la sortie vidéo, la sortie son et le port d'extension analogique. En haut, il y a le microphone et le port UART, permettant de connecter un module Bluetooth.

La figure **??** montre l'organisation des connecteurs sur la carte eMediaKit.

8.2.2 Contrôleur Ethernet

La carte eMediaKit possède un contrôleur ethernet SMSC LAN91C96[19], connecté sur 16 bits de large.

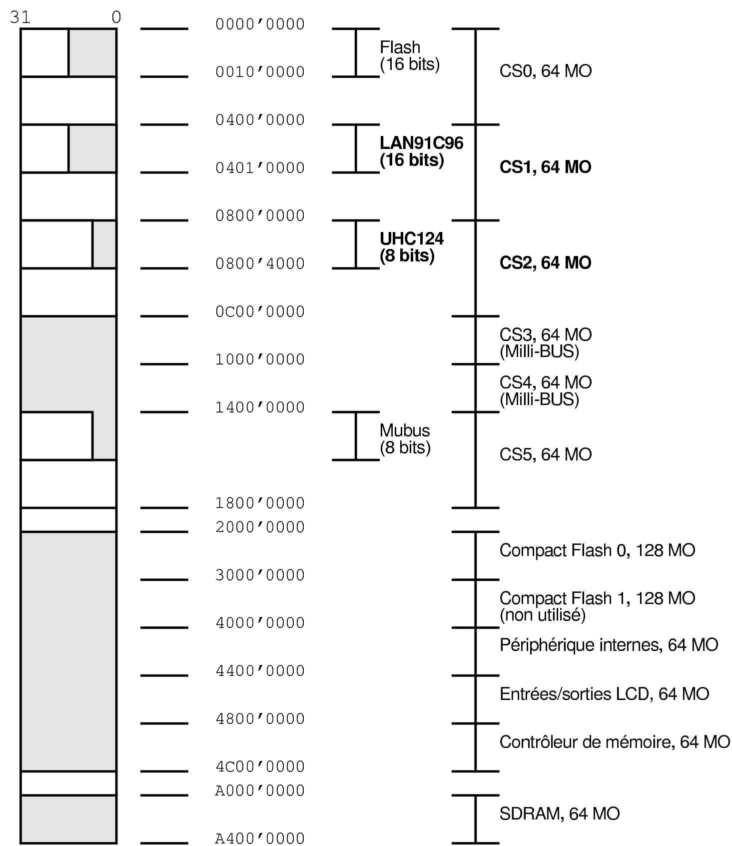

FIG. 8.4 – Carte de l'espace mémoire d'Armonie avec eMediaKit.

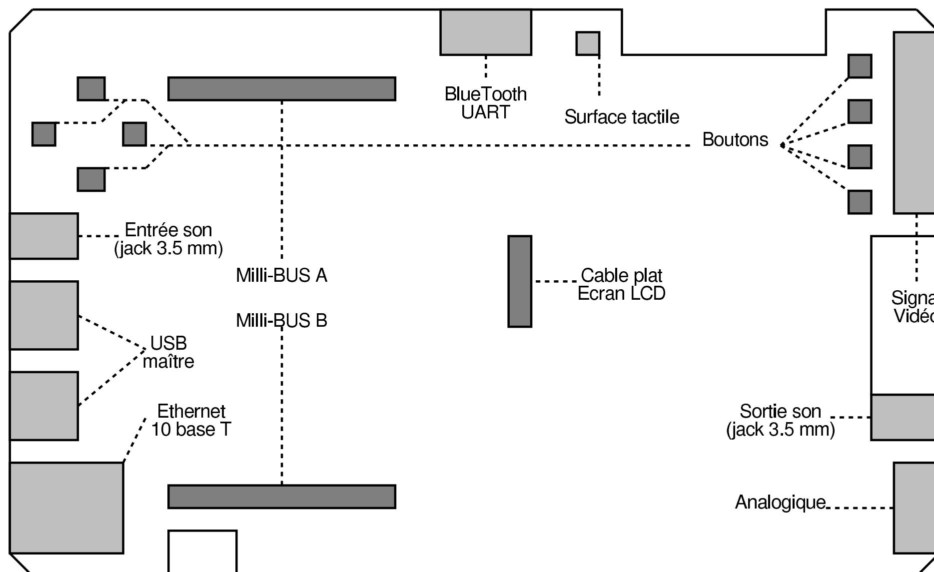

FIG. 8.5 – Disposition des connecteurs sur la carte eMediaKit.

Il est intéressant à plus d'un titre. En effet, il permet de se connecter à la carte afin d'avoir une ligne de commande, d'utiliser un disque par le réseau, de servir des pages web et permet de multiples autres applications.

Par contre, comme tout périphérique, il consomme du courant. Il est donc important que lorsqu'il n'est pas utilisé (système en mode autonome débranché du réseau et de l'alimentation secteur), il consomme le moins possible. C'est pourquoi nous avons choisi ce modèle.

En fonctionnement actif, il consomme, sous 3.3 V, typiquement 20 mA, et en mode veille (activable par logiciel), seulement 1.5 mA. Malheureusement, ce modèle ne fonctionne qu'à 10 Mbps. Nous avons aussi hésité avec le modèle à 100 Mbps de la même famille (LAN91C111[20]), mais comme ce dernier consomme trop (100 mA en mode actif et 36 mA en mode veille, soit plus que la version 10 Mbps en mode actif), nous nous sommes limité à une vitesse plus faible.

L'interfaçage physique est effectué par un connecteur bel 0810-1XX1-03, intégrant les filtres nécessaires.

Le pilote Linux pour ce circuit existe, mais nous n'avons eu le temps ni de l'adapter ni de le tester.

8.2.3 Puce AC97

Pour gérer le son et les interactions homme-machines, nous utilisons la puce UCB1400[21]. Cette dernière se connecte à Armonie grâce au bus AC97 (intégré dans le Milli-BUS), qui est un bus série commun dans le monde PC.

L'UCB fournit les fonctionnalités suivantes :

- Audio : entrée / sortie son stéréo, qualité CD (20 bits). L'entrée possède en outre un pré-amplificateur microphone ;
- Gestion d'une surface tactile : 4 entrées analogiques (10 bits) / sorties numériques, permettant de piloter une surface tactile résistive ;
- Entrées analogiques : sur le même convertisseur, 4 entrées analogiques peuvent être connectées ;
- 10 entrées / sorties numériques à usage général.

Nous n'avons pas eu le temps de tester l'UCB1400.

8.2.4 Son

La fonctionnalité audio fournie par l'UCB est exploitée par deux prises jacks 3.5mm stéréo : une faisant office de sortie son, l'autre d'entrée. Il est possible de souder un micro sur eMediaKit, et ce dernier sera utilisé si l'entrée n'est pas connectée.

8.2.5 Ecran et surface tactile

L'écran utilisé est le modèle LQ057Q3DC02[22] de Sharp. C'est un écran LCD matrice active couleur, d'une résolution de 320x240 pixels, acceptant jusqu'à 18 bits de couleurs (6 bits par composante). Il possède un rétro-éclairage efficace mais très gourmant en courant électrique (3.5 W). Ce dernier doit être alimenté avec 1000 V alternatif. Ce signal est généré à partir du +5 V par le circuit oscillant CXA-L0505-NJL[23].

Nous avons choisi cet écran surtout pour sa disponibilité, mais aussi pour sa taille et son interfaçage aisé (l'écran se syncronise facilement).

La spécification Milli-BUS suppose un bus de donnée pour l'écran de 16 bits de large (16 bits par pixel). L'organisation des couleurs dans le 16 bits est laissée au bon soin du créateur de la carte écran. La figure 8.5 illustre notre solution pour passer de 16 bits à 18 bits.

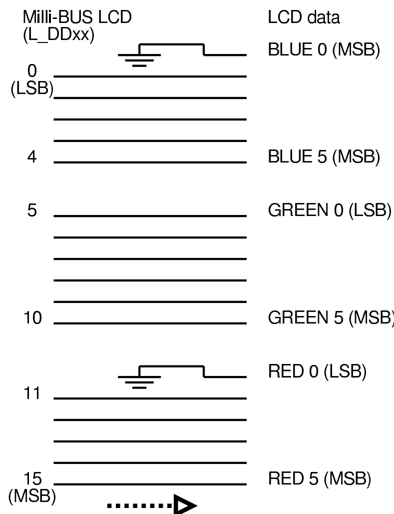

FIG. 8.6 – Connection de l'écran 18 bits au Milli-BUS LCD 16 bits

La surface tactile est une ATP-057, de type résistante[24, 25]. Elle s'interface armonieusement avec l'UCB1400, qui est clairement conçu pour gérer ce type de surface.

Une connecteur sortant les signaux écran est présent sur eMediaKit. Ainsi, avec un convertisseur externe et des réglages temporels appropriés, il est possible de sortir un signal compatible VGA.

8.2.6 Périphériques d'interaction homme-machine

L'entrée d'information se fait par plusieurs voies. Tout d'abord l'écran tactile. Il est connecté à l'UCB1400, qui communique avec la processeur par le bus AC97 intégré au Milli-BUS.

L'interaction se fait par trois voies, qui sont utilisées de façon combinée :

– écran tactile à pression variable (3 axes),
– croix directionnelle (4 boutons),
– boutons génériques (4 boutons).

Ainsi, les périphériques d'entrée présentent une richesse appréciable, malgré leur relative simplicité.

8.2.7 Contrôleur maître USB

La carte eMediaKit possède aussi un contrôleur maître USB. Ce dernier permet de connecter le système multimédia embarqué à une foule de périphériques, comme des claviers, des souris ou des caméras. Ainsi, avec le connecteur de sortie écran, le système peut être utilisé comme un ordinateur complet.

Le contrôleur choisi est l'UHC124[26]. Le choix a été simple ; c'était le seul facilement disponible.

L'UHC est connecté sur un bus de donnée de 8 bits. Or, la spécification Milli-BUS ne fournit pas un bus de 8 bits (le PXA250 ne le supportant pas), il est donc connecté sur le poids faible du bus de 32 bits, avec les 2 bits de poids faible d'addresses non connectés. Ainsi, il est possible de l'accéder en 32 bits avec un pilote approprié, qui ignore systématiquement les 24 bits de poids fort de chaque mot.

Ce contrôleur ne possède actuellement aucun pilote pour Linux, mais la complète disponibilité des spécifications permettrait de l'écrire.

Chapitre 9

Consommation et performances

L'économie d'énergie fait partie des objectifs d'Armonie. Cette carte doit pouvoir adapter sa consommation à la charge de calcul qui lui est demandée. Ce chapitre décrit les conditions de tests et résume les mesures de performances et de consommation effectuées.

Tous les tests ont été effectués sous Linux[1] qui active l'unité de gestion de mémoire et les caches de données et d'instructions.

Sept différents tests permettent d'évaluer les performances et la consommation de la carte :

1. inactivité,
2. unité arithmétique et logique (ALU),
3. multiplications entières,
4. accès mémoire SDRAM en lecture,
5. accès mémoire SDRAM en écriture,
6. accès mémoire cache en lecture,
7. accès mémoire cache en écriture.

Les mesures, pour les sept charges d'utilisation, ont été faites dans trois modes d'exécution différents :

1. processeur à 100 MHz, bus interne au processeur (*PXbus*) à 50 MHz alimenté à 0.9 V ;
2. processeur à 200 MHz, bus interne à 50 MHz, alimenté à 1.0 V ;
3. processeur à 200 MHz, bus interne à 100 MHz, alimenté à 1.3 V.

Le paragraphe 4.6 page 23 donne de plus amples explications à propos de ces différents modes d'exécution.

De plus, dans une optique d'utilisation d'Armonie avec un écran LCD, nous avons mesuré la consommation avec et sans le DMA qui envoie les pixels depuis la mémoire principale vers l'interface LCD.

Pour ces sept tests et dans ces six conditions, nous avons mesuré la consommation du coeur, la consommation de l'alimentation 3.3V et le nombre moyen d'itérations par seconde de chaque test. Les relevés mesurés sont disponibles (annexe 12) et sont présentés plus loin sous forme de graphiques.

9.1 Les tests de performances

Les sources du programme contenant tous les tests décrits ci-après sont disponibles dans la distribution logicielle acompagnant ce rapport, dans le répertoire `armlinux/nfsarm/bench/bench-compat.c`.

[1] Le noyau `linux-2.4.19-rmk4-pxa2-armonie1` est distribué avec ce projet.

9.1.1 Unité arithmétique et logique

Ce logiciel a pour but d'évaluer l'efficacité d'un processeur devant calculer beaucoup d'opérations logiques comme les additions, les soustractions, les décalages et les opérations bit-à-bit.

Les accés en mémoire sont limités au minimum : le programme et les données tiennent dans les caches de mémoire.

```
unsigned benchALU(void)
{
        unsigned i = ITER_COUNT;
        register unsigned a, b, c;

        a = 4234235;
        b = 124322;
        c = 390435;

        do
        {
                a += (c »2);
                c −= a ^ (b | ~c);
                b += ((7« c ) ^ ( c »2) ^  a) + 19;
        }
        while (−−i);

        loopCount = ITER_COUNT;
        return a+b+c;
}
```

9.1.2 Multiplications entières

Ce logiciel est très similaire au précédent : il teste le coeur du processeur, plus précisement l'unité de multiplications entières. La mémoire n'est pas concernée par ce test car le programme est suffisamment petit pour tenir en cache mémoire.

```
unsigned benchMULT(void)
{
        unsigned i = ITER_COUNT;
        register unsigned a, b, c;

        a = 4234235;
        b = 124322;
        c = 390435;

        do
        {
                a += c ∗ a;
                c −= a ∗ (b ∗ c);
                b += ((a∗c) − (c∗b) ∗ a) + 19;
        }
        while (−−i);

        loopCount = ITER_COUNT;
        return a+b+c;
}
```

9.1.3 Lectures en mémoire

Cette portion de programme lit une grande zone de mémoire de manière consécutive. Bien que la taille de cette fenêtre soit largement plus grande que le cache de donnée, celui-ci intervient quand même puisque les accès sont séquentiels.

```
unsigned benchRAMRead(void)
{
        unsigned val = 0;
        unsigned *data=(unsigned *)malloc(MALLOC_SIZE*sizeof(unsigned));
        unsigned *mem=data;
        int i;

        i = MALLOC_SIZE;
        do
        {
                val |= mem[0];
                val |= mem[1];
                val |= mem[2];
                val |= mem[3];
                val |= mem[4];
                val |= mem[5];
                val |= mem[6];
                val |= mem[7];
                val |= mem[8];
                val |= mem[9];
                val |= mem[10];
                val |= mem[11];
                val |= mem[12];
                val |= mem[13];
                val |= mem[14];
                val |= mem[15];
                i-=16;
                mem+=16;
        }
        while (i>0);

        free(data);

        loopCount = MALLOC_SIZE;
        return val;
}
```

9.1.4 Ecritures en mémoire

Ce test évalue les performances d'écriture en mémoire en écrivant séquentiellement une zone de mémoire largement plus grande que le cache de mémoire.

```
unsigned benchRAMWrite(void)
{
        unsigned val = 0;
        unsigned *data=(unsigned *)malloc(MALLOC_SIZE*sizeof(unsigned));
        unsigned *mem=data;
        int i;
```

```
        i = MALLOC_SIZE;
        do
        {
                mem[0] = val;
                mem[1] = val;
                mem[2] = val;
                mem[3] = val;
                mem[4] = val;
                mem[5] = val;
                mem[6] = val;
                mem[7] = val;
                mem[8] = val;
                mem[9] = val;
                mem[10] = val;
                mem[11] = val;
                mem[12] = val;
                mem[13] = val;
                mem[14] = val;
                mem[15] = val;
                i-=16;
                mem+=16;
        }
        while (i>0);

        free(data);

        loopCount = MALLOC_SIZE;
        return val;
}
```

9.1.5 Lectures en cache

Ce test effectue de nombreuses lectures dans une zone de mémoire suffisament petite pour tenir en cache.

```
unsigned benchCacheRead(void)
{
        unsigned i = ITER_COUNT;
        volatile unsigned mem[16];
        unsigned val = 0;

        do
        {
                val |= mem[0];
                val |= mem[1];
                val |= mem[2];
                val |= mem[3];
                val |= mem[4];
                val |= mem[5];
                val |= mem[6];
                val |= mem[7];
                val |= mem[8];
                val |= mem[9];
                val |= mem[10];
```

```
            val |= mem[11];
            val |= mem[12];
            val |= mem[13];
            val |= mem[14];
            val |= mem[15];
        }
        while (--i);

        loopCount = ITER_COUNT;
        return val;
}
```

9.1.6 Ecritures en cache

Ce test effectue des écritures dans une zone de mémoire suffisamment petite pour tenir en cache. Cependant, sur le PXA250, Linux active le cache en écriture directe (*write through*) à cause d'un problème matériel[2] pouvant corrompre la mémoire. Ce test est donc très similaire au test d'écriture en mémoire, puisque le cache n'est pas utilisable correctement sur les versions du processeur que nous possédons.

Il y a quand même une différence : lorsqu'une écriture en mémoire manque le cache, le PXA250 lit une ligne de cache complète avant d'effectuer l'écriture. Le test d'écriture en mémoire cause donc autant de lectures que d'écritures vers les mémoires SDRAM, alors que le test d'écritures en cache ne cause que des écritures, donc deux fois moins d'accès.

```
unsigned benchCacheWrite(void)
{
        unsigned i = ITER_COUNT;
        volatile unsigned mem[16];
        unsigned val = 0;

        do
        {
            mem[0] = val;
            mem[1] = val;
            mem[2] = val;
            mem[3] = val;
            mem[4] = val;
            mem[5] = val;
            mem[6] = val;
            mem[7] = val;
            mem[8] = val;
            mem[9] = val;
            mem[10] = val;
            mem[11] = val;
            mem[12] = val;
            mem[13] = val;
            mem[14] = val;
            mem[15] = val;
        }
        while (--i);

        loopCount = ITER_COUNT;
        return val;
```

[2]voir errata n° 120 du document *Intel PXA250 and PXA210 Processors Specification Update*.

}

9.2 Consommation

Trois graphiques donnent une bonne idée de la consommation d'Armonie :
- la consommation du coeur,
- la consommation de l'alimentation 3.3V,
- la consommation totale dans les différentes conditions de test.

9.2.1 Consommation du coeur

La figure 9.1 montre la consommation du coeur du PXA250. On constate que l'utilisation ou non du DMA pour afficher une image sur un écran n'influence presque pas la consommation du coeur.

Dans des conditions d'inactivité, on voit que la consommation n'augmente pas beaucoup entre le mode normal et le mode turbo, mais fait plus que doubler quand le bus interne passe de 50 MHz à 100 MHz. Le coeur arrive visiblement à éviter de dissiper trop d'énergie inutilement quand il ne fait rien, contrairement aux autres composants cadencés par l'horloge du bus interne[3].

Le mode turbo avec un bus à 50 MHz semble adapté à une tâche qui demande peu d'accès en mémoire et beaucoup de puissance processeur. La figure 9.5 expliquée plus loin confirme cette hypothèse : les tests de multiplications et d'unité arithmétique et logique ont un meilleur rapport entre performance et puissance en mode turbo.

9.2.2 Consommation de l'alimentation 3.3 V

L'alimentation 3.3V alimente principalement les quatres circuits de mémoire SDRAM. La mémoire flash y est aussi connectée, mais n'est pas utilisée dans nos tests ; elle consomme une énergie constante. Le PXA250 utilise du 3.3V pour les entrées sorties CMOS, pour le bus mémoire et pour les pattes de l'interface PCMCIA. La consommation 3.3V est donc fortement liée à l'activité de la mémoire SDRAM.

La première constatation qu'on peut faire en regardant la figure 9.2 qui montre la puissance consommée par l'alimentation 3.3V, est que l'utilisation ou non du DMA a une grande influence sur les résultats. Dans les tests où la mémoire ne joue pas un grand rôle – ALU, inactivité, multiplication et lecture de cache – les fréquences du coeur et du bus interne importent peu, c'est le DMA qui influence la consommation. Par contre, du moment qu'un programme accède à la mémoire, ajouter le DMA pour afficher une image ne change pas massivement la consommation.

De nombreux accès aux mémoires dynamiques ne coûtent pas beaucoup plus que peu d'accès, c'est en supprimant tous les accès qu'on économise jusqu'à 300 mW. Les cache de mémoires ont donc une grande importance en matière de consommation. Cet effet est aussi visible en comparant les tests d'écritures et de lectures en mémoire : à chaque écriture, le contrôleur doit accéder aux mémoires puisque le cache est configuré en écriture directe (*write-through*). En lecture séquentielle, par contre, le contrôleur remplit une fois sur 8 une ligne de cache, les 7 accès suivants n'ont pas besoin d'accéder aux mémoires.

On voit, en comparant ces deux cas sur les figures 9.1 et 9.2 que le cache de mémoire en lecture économise entre 50 mW et 100 mW sur l'alimentation 3.3V, pour une augmentation de la consommation du coeur d'environ 20 mW dans le pire des cas. Un programme optimisé pour utiliser au mieux les ressources du cache de mémoire dépensera moins d'énergie.

Bien que configuré pour fonctionner à 100 MHz, le contrôleur cadence les mémoires SDRAM à 50 MHz. On comprend donc que la consommation reste similaire quelles que soient les fréquences du coeur et du bus interne.

[3] voir page 3-3 du manuel du developpeur [3].

9.2.3 Consommation totale

Le schéma 9.3 montre la somme des consommations des alimentations du coeur et 3.3V. On voit que la partie la plus significative de la dépense d'énergie d'Armonie est tirée de l'alimentation 3.3V, conséquence de la présence de quatres circuits de mémoire SDRAM sur la carte[4].

La plage de consommation totale se trouve environ entre 220 mW et 750 mW, les logiciels ont donc une grande responsabilité en matière de dépense énergétique. Pour diminuer la consommation d'Armonie, il est plus intéressant de limiter les accés en mémoires que de ralentir le processeur. Une autre possibilité serait, si 32 méga-octets de mémoire suffisent, de n'utiliser que deux des quatres circuits de SDRAM sur un bus de 16 bits de large. Le temps manque pour tester cette solution.

9.3 Performances

Trois graphiques donnent une bonne idée des performances relatives et absolues d'Armonie :
- les performances,
- le rapport performance / consommation,
- les performances comparées avec des Pentiums.

9.3.1 Performances

La figure 9.4 montre les performances du PXA250 dans les différentes conditions de test. Les résultats sont normalisés à 1 afin de bien montrer les effets des changements de fréquences.

Nous constatons que pour ce qui est du calcul pur (ALU et multiplication) et qui ne sort pas du coeur (lecture du cache), le doublement de la fréquence du coeur entraine exactement le doublement des performances. Dans le cas des tests accédant à la mémoire SDRAM, c'est surtout la fréquence du bus interne qui est déterminante. Le DMA a une influence significative, bien qu'assez faible, dans les tests accédant à la mémoire. Evidemment, quand seul le coeur travaille, son influence est nulle.

On voit dans ce graphique les effets du cache lors d'écritures : à chaque écriture, si la ligne correspondante n'est pas dans le cache, le PXA250 va faire une lecture de 32 octets pour remplir une ligne en plus d'effectuer l'écriture en mémoire[5]. Le test d'écritures séquentielles force donc le processeur à lire 32 octets avant de faire 32 écritures consécutives. Puisqu'une écriture dans une zone déjà cachée ne nécessite pas cette opération de remplissage de cache, il y a deux fois moins de trafic vers la mémoire dans le test d'écriture en cache que dans le test d'écriture en mémoire.

9.3.2 Rapport performances / consommation

La figure 9.5 qui montre le rapport normalisé des performances sur la puissance consommée, est plus intéressante encore.

En effet, dans le domaine de l'embarqué, ce n'est pas tellement la puissance pure qui est importante, mais plutôt son rapport avec la consommation. Pour lire ce graphique, il ne faut pas comparer les tests avec et sans DMA, mais plutôt les trois différentes combinaisons de fréquences de bus et de coeur.

Dans les tests n'accédant pas à la mémoire, on voit que la situation la plus efficace est une grande vitesse du coeur avec un bus le plus lent possible, ce qui est logique, puisqu'il est peu sollicité.

Dans les tests qui font en moyenne environ un accès mémoire par instruction, c'est-à-dire les tests de lectures en mémoire et d'écritures en cache[6], on constate une amélioration des performances quand le coeur passe de 100 MHz à 200 MHz. Cela signifie qu'à 100 MHz le coeur n'est pas capable d'exploiter la bande passante du bus interne et de la mémoire. Néanmoins, cette amélioration est moins marquée lors de lectures que lors d'écritures. On peut en déduire que, pour le coeur, effectuer une écriture est plus compliqué qu'une lecture.

[4] Ces quatres circuits permettent une largeur de bus de 32 bits, et donc un gain de performances intéressant par rapport à un bus de seulement 16 bits.

[5] le bit de modification (*dirty bit*) n'est alors pas mis. Voir pages 6-5 et 6-6 du manuel de l'utilisateur [2].

[6] le paragraphe 9.1.6 explique les effets d'une écriture à une adresse contenue dans le cache.

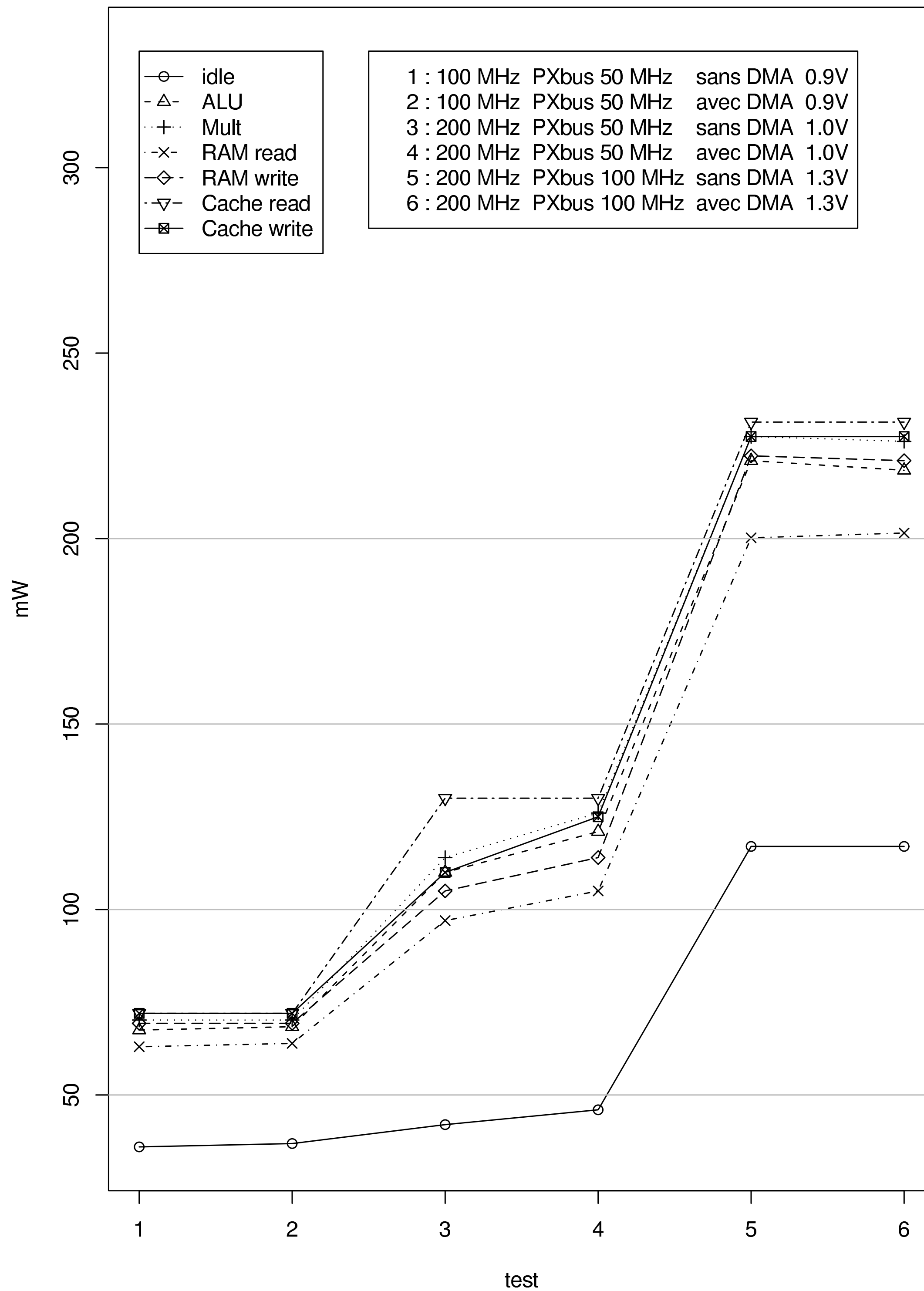

FIG. 9.1 – Consommation du coeur

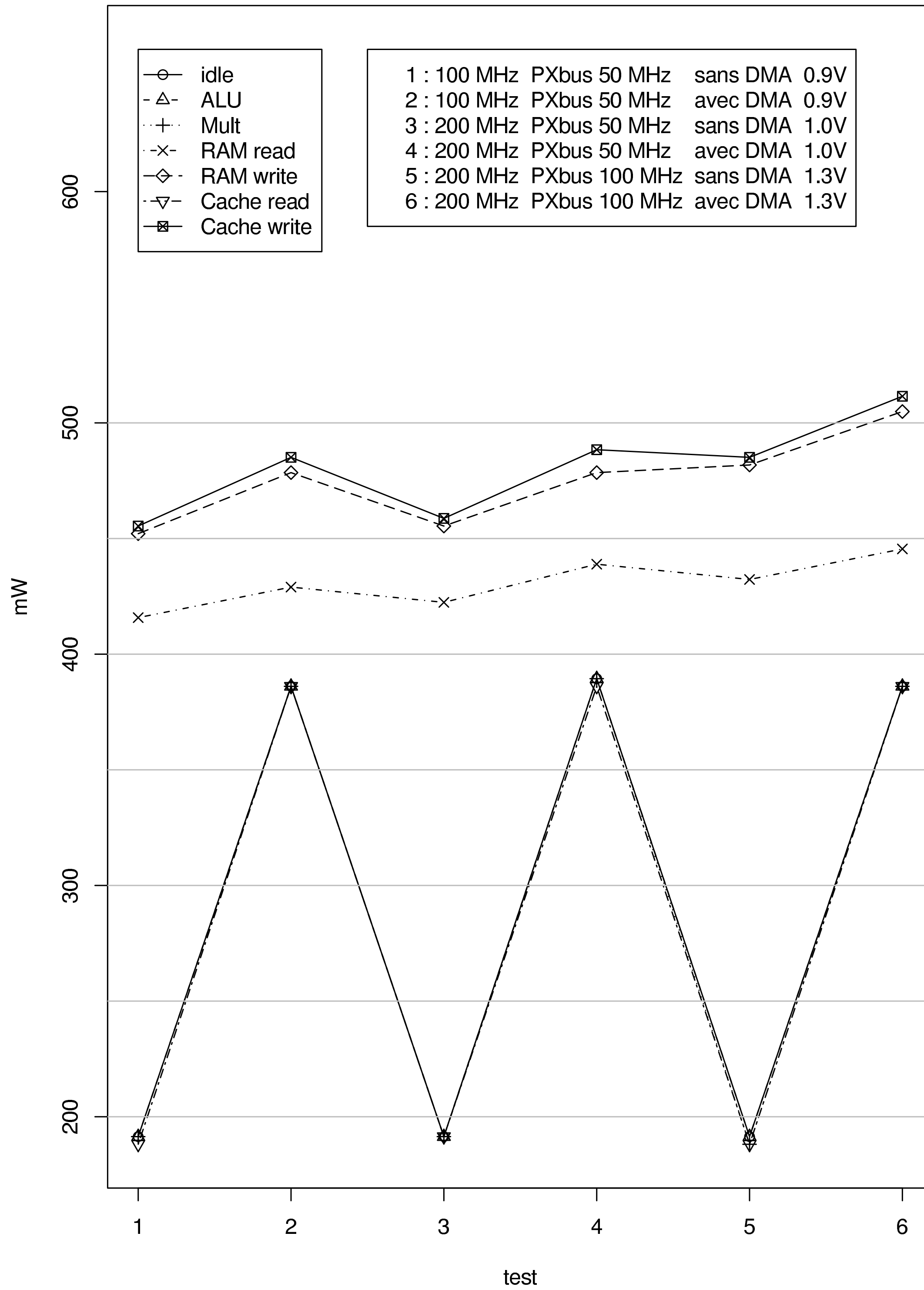

FIG. 9.2 – Consommation de l'alimentation 3.3V

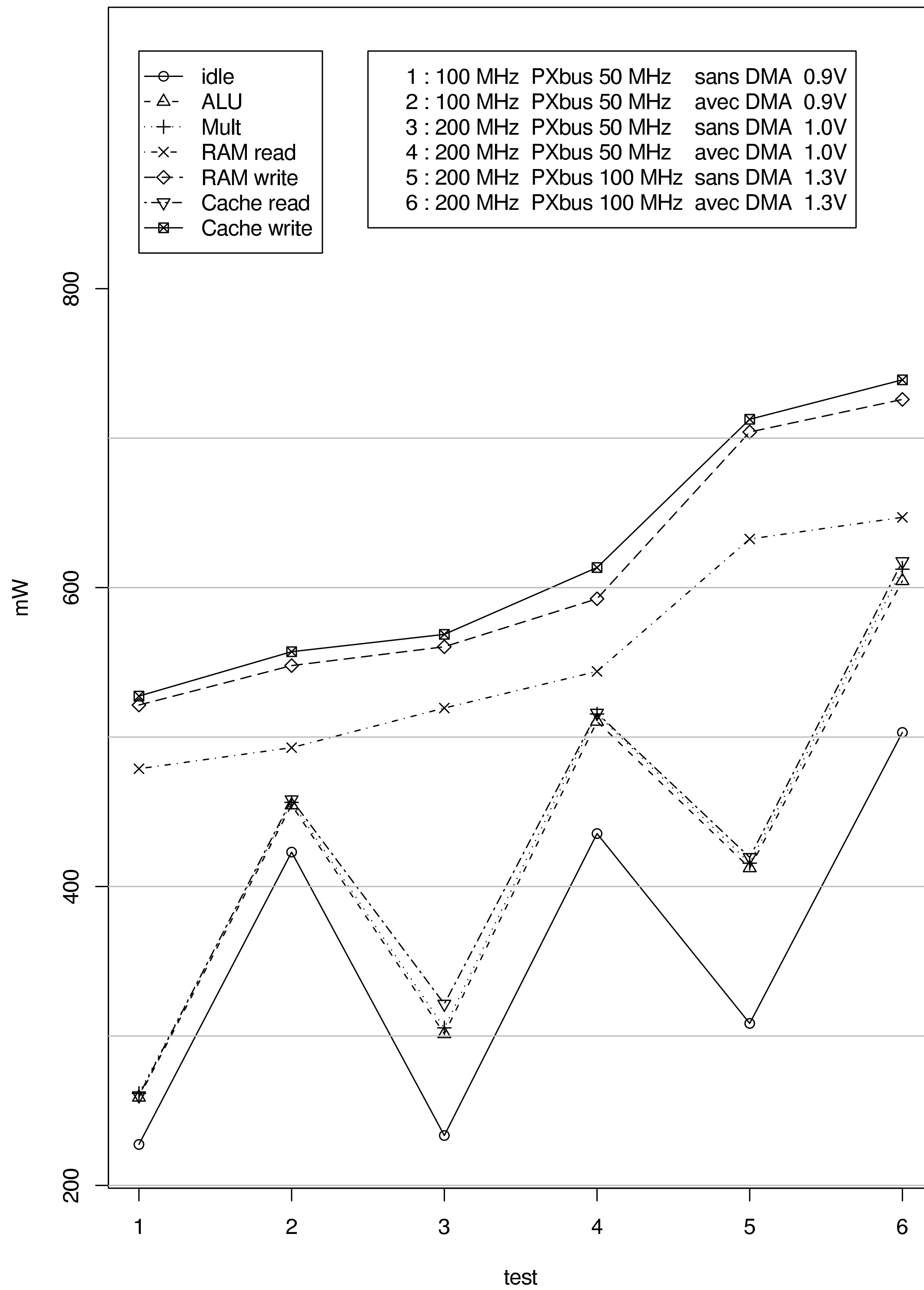

FIG. 9.3 – Consommation totale d'Armonie

D'autre part, dans le test d'écritures séquentielles en mémoire, qui cause en moyenne deux accès mémoire par instruction, cette amélioration n'est presque pas visible : le coeur arrive dans ce test à exploiter toute la bande passante vers la mémoire.

9.3.3 Performances par rapport à un PC

La figure 9.6 permet une comparaison avec des processeurs de PC.

Nous constatons que dans les tâches pour lesquelles le PXA250 a été conçu (calcul en virgule fixe), il est efficace. En lecture de cache, le rapport de vitesse correspond au rapport des fréquences. Il est intéressant de constater qu'en écriture en mémoire, la vitesse est comparable à celle d'un Pentium III à 800 Mhz.

Il est amusant de voir que pour deux tests sur six (multiplication et lecture mémoire), le Pentium III à 800 Mhz est plus rapide, et ce de façon significative, que le Pentium 4 à 1.7 GHz.

9.4 Conclusion

Après ces analyses, on se rend compte qu'il est très intéressant d'avoir un bus assez lent suivant les calculs à faire, meme si cette solution semble à priori inefficace.

Nous constatons que la consommation de la mémoire (alimentation 3.3V) est un facteur important à prendre en compte lors de la conception d'un système embarqué. En effet, si lors d'un calcul n'utilisant que le coeur et le cache à 200 Mhz, elle consomme légèrement moins que le coeur, lors d'un accès intensif à 100 Mhz, elle peut consommer jusqu'à 6 fois plus (450 mW contre 75 mW pour le coeur).

En matière de consommation, ces résultats montrent l'importance du cache de mémoire que les logiciels devront exploiter au mieux pour économiser l'énergie.

Pour limiter la consommation, il est important de connaître la puissance de calcul requise par un logiciel afin d'adapter la fréquence et la tension d'alimentation. Une autre solution consiste à mesurer les besoins de puissance de calcul afin de l'adapter automatiquement.

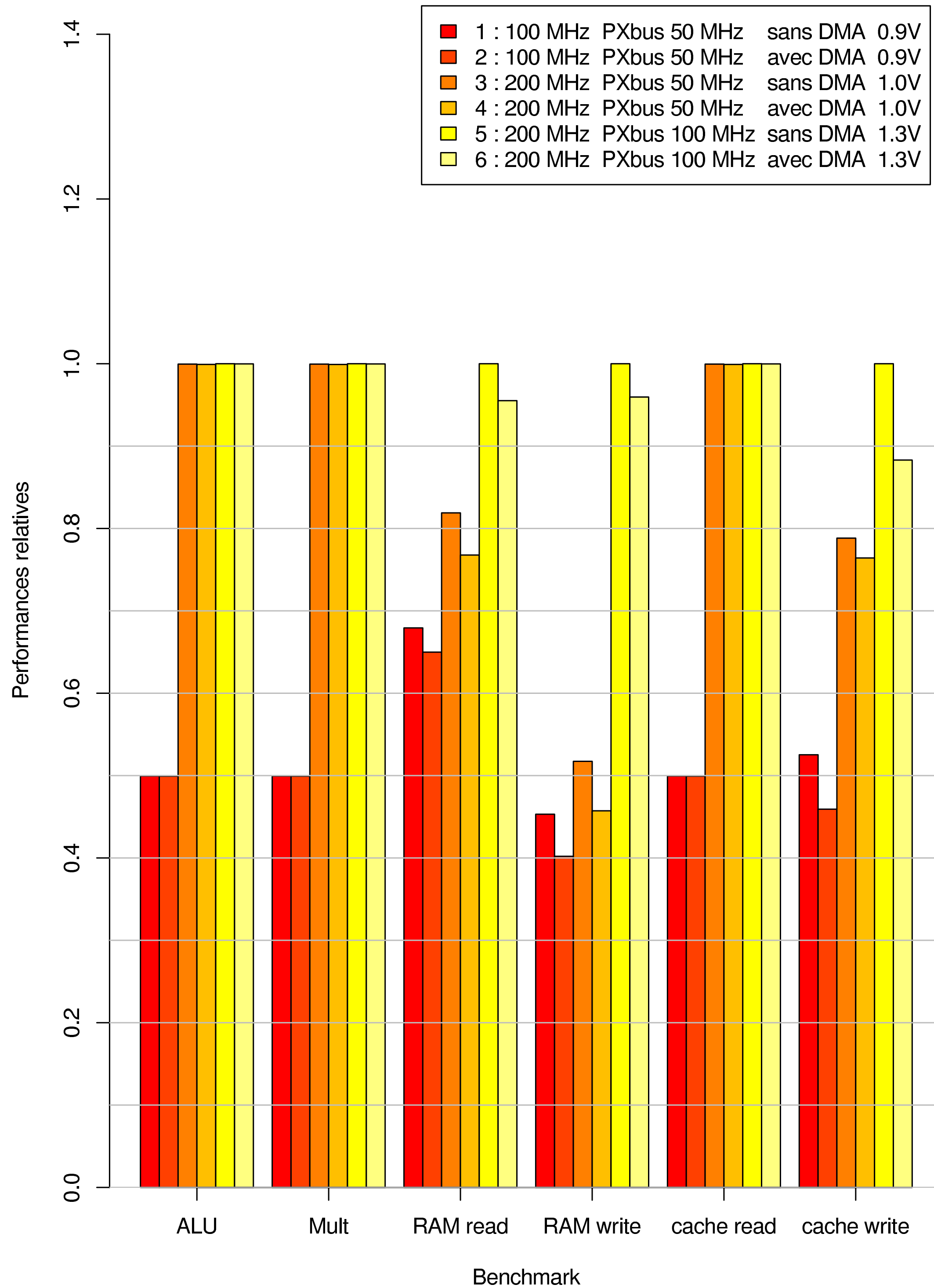

FIG. 9.4 – Comparaison des performances relatives

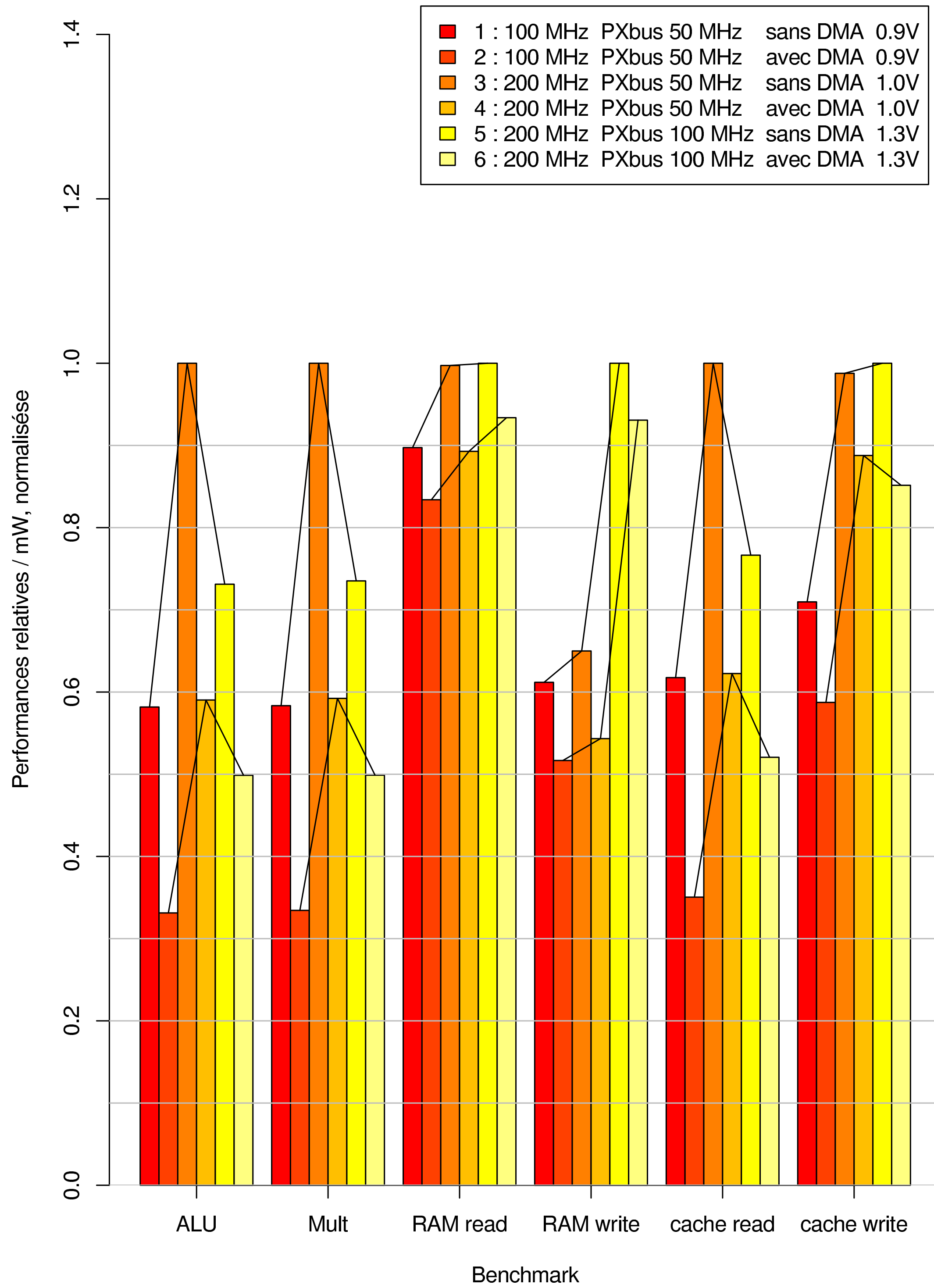

FIG. 9.5 – Performances relatives / consommation totale

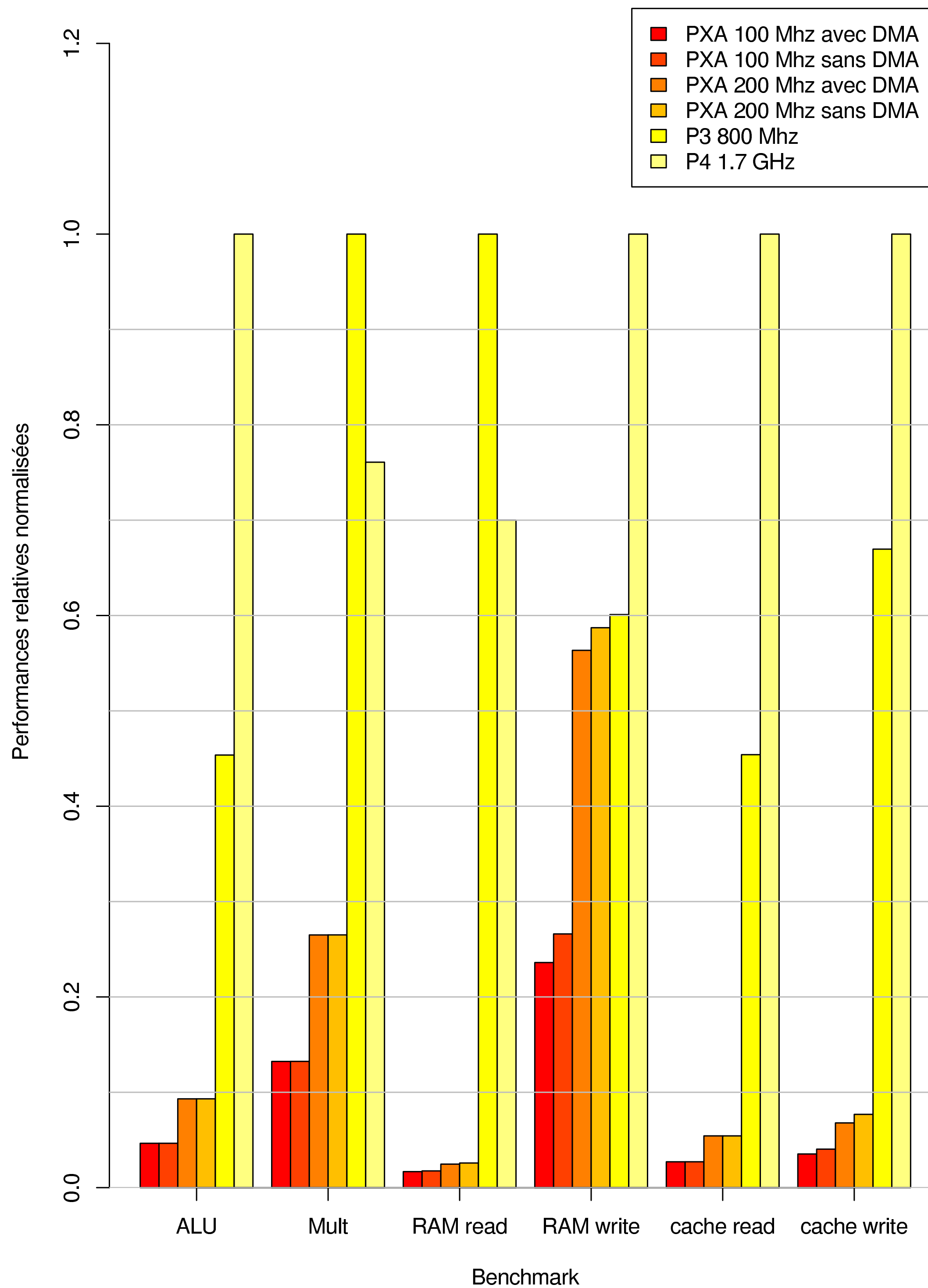

FIG. 9.6 – Comparaison des vitesses entre le PXA250 et des Pentium III et IV

Chapitre 10

A propos de ce projet

Ce chapitre pose un regard sur ce projet. Il présente les outils utilisés et conclut ce travail.

10.1 Difficultés

Au cours de ce projet, nous avons rencontrés plusieurs difficultés que nous énumérons ici.

10.1.1 Processeurs partiellement défectueux

La première version de processeur avec laquelle nous avons travaillé était le PXA250, *stepping* A. Nous avons passé pas mal de temps à comprendre pourquoi l'horloge n'oscillait pas, pour nous rendre compte que le pilote du quartz de cette version était défectueux, et qu'il fallait introduire l'horloge depuis l'extérieur, en utilisant un oscillateur.

Une fois ce problème d'horloge résolu, le JTAG ne fonctionnait pas. Nous avons passé beaucoup de temps à analyser la séquence générée par nos outils à l'analyseur logique, pour finalement observer que même avec une séquence très simple, le comportement du JTAG du PXA250 était incohérent. En effet, la machine d'état JTAG ne fonctionnait pas non plus !

La version *stepping* B a amélioré la situation, mais la liste d'errata reste très longue. Nous n'avons testé ni le *stepping* C ni la version PXA255, sensée corriger de nombreux problèmes.

Pour ces raisons, il est difficile de dire comment fonctionneront nos outils de développement sur les version futures du PXA250.

10.1.2 Débogage outils, logiciel et matériel

Dans ce projet, nous avons créé le matériel, Armonie ; et le logiciel, Jelie. Jelie fonctionne sur plusieurs niveaux : dans le coeur du PXA250 avec le gestionnaire de debogue, sur le microcontrôleur USB et sur le PC. Il était parfois très difficile de trouver un problème, car chaque pièce du système était en développement ou pouvait être boguée. Il nous fallait même mettre le processeur en doute, car nous n'avions que des échantillons au début du projet.

Nous avons surmonté ces difficultés, et nous avons maintenant une chaîne d'outils plutôt stable.

10.1.3 Documentation floue ou erronée

La documentation d'Intel nous a parfois posé des problèmes.

Il y a une grande différence entre la procédure de changement de fréquence (FCS) décrite dans la documentation et ce qu'il faut faire en réalité : aucun événement externe n'est nécessaire, contrairement à ce que spécifie le manuel du développeur.

Le mode spécial de débogage (SDS) impose des restrictions considérables au logiciel qui l'utilise. Or, la documentation à ce sujet est floue et sujette à des erratas. Beaucoup d'essais ont été nécessaires pour écrire le gestionnaire de débogage.

Le contrôleur de SDRAM devrait pouvoir fonctionner à 100 MHz aussi bien qu'à 50 MHz. Pourtant, malgré l'utilisation du code généré par le site web d'Intel, nous n'avons pas réussi à effectuer ce changement de fréquence. Seul le bus interne passe à 100 MHz.

10.2 Outils utilisés et télé-travail

10.2.1 Présentation de l'espace de travail

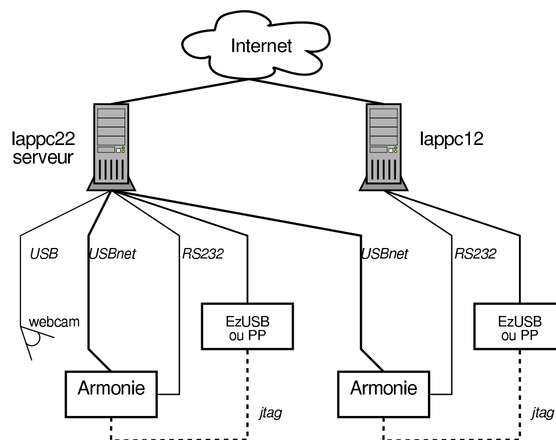

FIG. 10.1 – Environnement de travail

Pour ce projet, notre environnement de travail était composé d'un PC serveur (*lappc22*) et d'un client (*lappc12*), tous deux sous Debian GNU/Linux[4]. La figure 10.1 page 67 présente cette organisation. Le serveur, *lappc22*, fournit les services suivants :

- Serveur de fichier NFS pour le PC client.
- Serveur CVS.
- Serveur de fichier NFS pour les deux cartes Armonie, à travers l'USB.
- Caméra, pour pouvoir observer Armonie depuis Internet.

De plus, chaque station de travail possède une connexion série et JTAG (par USB ou port parallèle) avec sa carte Armonie attitrée. L'idée derrière cette organisation était de pouvoir travailler aussi bien sur son poste que sur celui du voisin, et même à distance à travers Internet. Ce système fonctionne très bien et nous a rendu de grands services.

10.2.2 Debian

Les PC tournent sous le système d'exploitation Debian GNU[4], avec un noyau Linux 2.4. Ce système est à la fois sûr, stable, libre et gratuit. Il nous a largement satisfait durant ce projet.

10.2.3 CVS, copies de sauvegarde et serveur de fichier

Nous utilisons CVS[6] qui est un système de contrôle de versions, permettant à plusieurs personnes de travailler au même moment sur les mêmes sources, et effectuant automatiquement, dans les limites du possible, la fusion entre les différentes versions. De plus, toute version antérieure du programme peut être récupérée, que ce soit en spécifiant la date ou un identifiant de version.

lappc22 sert les fichiers du projet pour *lappc12*. Toutes les nuits un script crée une archive contenant nos fichiers de travail et ceux du CVS et la copie sur le serveur du laboratoire. Ce script porte les numéros du jour et du mois où la sauvegarde a été effectuée. Ainsi, nous pouvons retrouver l'état de notre travail du dernier mois (même s'il n'est pas dans CVS).

10.2.4 Caméra

Au début du développement logiciel, avant d'avoir un Linux qui tourne sur Armonie, nous utilisions souvent les quelques diodes lumineuses et l'interface Mubus disponibles afin d'avoir une idée de l'état interne de notre programme.

Malheureusement, ceci nous obligeait à être physiquement à l'EPFL pour pouvoir travailler. Comme cette contrainte nous semblait indigne de la technologie du XXI$^{\text{ème}}$ siècle, nous avons décidé de pointer une caméra sur une des cartes, et d'utiliser le serveur HTTP de *lappc22* pour regarder Armonie. Cette approche s'est avérée utile, puisqu'elle nous a permis de travailler à distance sans handicap lors d'un cas de maladie.

10.2.5 Doxygen

Documenter le fonctionnement d'un logiciel n'est pas chose facile. Néanmoins, c'est un travail nécessaire, surtout dans le contexte universitaire, où les projets changent de main fréquemment.

Comme nous avons espoir que Jelie continuera a être utilisé après notre projet, nous voulons qu'il soit correctement documenté. Nous utilisons Doxygen, un outil de documentation automatique à partir de marques dans le code source, un peu comme Javadoc.

Cet outil, à partir du source, génère la documentation des fonctions, des méthodes, dessine les graphes de dépendances et de collaboration, et produit du HTML et grâce à LaTeX, du PostScript et du PDF. La version LaTeX est annexée à ce document, et la version html est disponible sur internet à l'addresse suivante :
`http://lappc22.epfl.ch/pxa250/html/`

10.2.6 Outils utilisés pour le développement.

Nous avons utilisés les outils suivants de la chaîne d'outils GNU, à la fois pour compiler du code x86 destiné au PC, et pour compiler du code ARM destiné au PXA250 :
- binutils, assembleur et gestionnaire de fichiers objets,
- GCC (GNU Compiler Collection), compilateur C/C++,
- GDB (GNU Debugger),
- make, gestionnaire de compilation.

De plus, pour éditeur nos fichier sources, nous avons utilisé l'éditeur de texte vim, et d'autres outils Unix standards (grep, sed, ...).

10.2.7 Outils utilisés pour le rapport et la présentation.

Pour la publication, nous avons utilisés les outils suivants :
- R, environnement d'analyse statistique et de graphique (`http://www.r-project.org/`)[1].
- LaTeX, processeur de texte et compilateur de document (`http://www.latex-project.org/`).
- LyX, interface graphique pour LaTeX (`http://www.lyx.org/`).
- OpenOffice.org, suite office libre (`http://www.openoffice.org/`).
- Dia, éditeur de diagramme (`http://www.lysator.liu.se/~alla/dia/`).
- XFIG, éditeur de dessin vectoriel (`http://www.xfig.org/`).
- Graphviz, générateur de graph (`http://www.research.att.com/sw/tools/graphviz/`).

[1] La décision d'utiliser R plutôt qu'un simple tableur est motivée par l'envie de connaître un vrai language de statistique, qui nous permettra dans l'avenir d'être efficaces, même avec des données complexes. Ainsi, nous avons investi un peu de temps durant ce projet pour apprendre à utiliser R.

10.3 Continuations possibles

Bien que nous avons eu l'occasion de développer plein de choses au cours de ce projet, il reste une foule de travaux intéressants à effectuer.

10.3.1 Plateforme pédagogique

Les outils de développement sous Linux fonctionnent bien, mais peuvent être améliorés. Il serait possible d'utiliser l'analyseur logique de Sebastian Gerlach comme interface JTAG. Il serait aussi intéressant de supporter d'autres processeurs ARM, voire d'autres architectures.

10.3.2 Robot cyclope

La voie du robot cyclope n'a pas été explorée dans ce projet. Maintenant que la carte d'alimentation d'Adrian Spycher fonctionne, il est intéressant de mettre Armonie sur le robot et d'adapter une caméra couleur. Il serait aussi judicieux de regarder quels algorithmes peuvent être implémentés sur Armonie, et lesquels tirent avantageusement profit de sa grande puissance de calcul.

10.3.3 Système multimédia embarqué

Ce système peut être développé plus en avant. En effet, il faudrait finir de tester tous les périphériques, adapter ou écrire les pilotes de périphériques pour Linux. Il faudrait aussi trouver pourquoi le Compact-Flash ne fonctionne pas. Le boitier n'est pas non plus construit. De plus, il serait bon de tester différentes applications, et étudier leur impact du point de vue de la consommation.

10.3.4 La gestion d'énergie sous Linux

Un système de gestion d'énergie, utilisant la capacité de certains processeurs de pouvoir changer à la fois leurs fréquences et leurs voltages, a été développé au LAP. Ce système fonctionne avec Linux sur StrongARM[2] et pourrait être adapté au PXA250, pour autant qu'une carte d'alimentation fournisse les services matériels de changement de voltage et de fréquence[3]. Une telle carte a été faites par Adrian Spycher comme projet de semestre. Cette carte fonctionne et permettrait de décroître de façon significative la consommation d'énergie d'Armonie.

10.3.5 Ensemble d'outils JTAG

Cédric Gaudin, lors de son projet, a développé un analyseur JTAG sur une logique programmable (FPGA). Avec Jelie, ils pourraient former un ensemble d'outils JTAG complet, comprenant un analyseur et un programme de pilotage, tous deux génériques. Cela impliquerait la modification de Jelie pour supporter plusieurs types de composants – tâche relativement aisée puisque la conception objet de Jelie prévoit la séparation du code spécifique.

Du point de vue matériel, il serait possible d'utiliser l'analyseur logique que Sebastian Gerlach a développé il y a quelques années au LAMI. Cette carte existe en nombre suffisant et possède un microcontrôleur EzUSB, une mémoire statique et une logique programmable. La logique pourrait implémenter un registre à décalage pour accélérer les accès au port JTAG. La mémoire fournirait un tampon suffisant pour utiliser au mieux la bande passante USB.

Ces trois éléments réunis – analyseur JTAG, analyseur logique et Jelie – fourniraient un outils JTAG complet, rapide et performant.

[2] La famille StrongARM précède la famille XScale.

[3] Selon la documentation d'Intel, le PXA250 a besoin d'un évenement externe de réveil lors du changement de fréquence ou de tension.

10.4 Résumé du travail effectué

Au cours de ce projet, nous avons réalisé les travaux suivantes :
- Jelie, un programme de pilotage de port JTAG d'environ 10'000 lignes de C++ et d'assembleur ARM, abondamment documenté ;
- Adaptation des outils de développement GNU et réalisation d'un environnement de développement C, C++ et assembleur, avec ou sans système d'exploitation ;
- Port de Linux sur Armonie ;
- Pilote Mubus et programmes de tests pour Linux ;
- Conception d'un système multimédia autonome ;
- Utilisation de l'écran LCD sous Linux ;
- Mesures de performances et de consommation d'Armonie.

10.5 Collaboration

10.5.1 Entre étudiants

Nous avons enchaîné un projet de semestre et notre travail de diplôme au LAP. Cédric Gaudin et Damien Baumann, étudiants de notre volée, ont fait de même. Comme nous travaillions sur des projets assez proches (construction de cartes, puis de systèmes complets avec processeur ARM), nous avons pu collaborer étroitement. Adrian Spycher qui a construit une alimentation de qualité pour nos cartes, nous a fourni une aide importante.

Nous avons également travaillé avec Dario Suarez Gracia qui a programmé un support ARM pour le compilateur SUIF.

C'est dans un esprit de camaraderie, où l'entraide côtoyait le sourire, que nous avons eu le plaisir de passer la dernière année. De plus, comme nous partageons la vision d'une informatique libre, tous nos schémas, codes et documents étaient à disposition des autres, et nous avons plusieurs fois créé des choses en commun. A-BUS, Milli-BUS, AbusAlim et eMediaKit sont les fruits d'une coopération efficace. Le résultat est un travail effectué important et de qualité, qui – nous l'espèrons – sera profitablement exploité dans le futur.

10.5.2 Au sein du laboratoire

Nous avons aussi mis en place un serveur Debian GNU/Linux, *lappc22*, qui a été utile à une dizaine de personnes dans le laboratoire. Ce serveur contient notamment plusieurs version de GCC pour ARM qui ont rendu service à une partie de l'équipe de recherche du laboratoire qui travaille sur cette architecture.

10.5.3 Avec l'extérieur

Nous avons aussi collaboré avec le K-Team, une entreprise de la région spécialisée en robotique. Nous avons échangé schémas et savoir faire. Le K-Team utilise Jelie, et comme ce dernier est en GPL, les améliorations du K-Team profiteront à toute la communauté.

10.6 Conclusions

Après tout ce travail, nous déconseillons l'utilisation des processeurs PXA250 d'Intel. Bien qu'ils semblent à priori peu couteux et riches en fonctionnalités, ils sont commercialisés trop tôt et sont trop bogués.

Malgré ceci, ce projet nous a beaucoup apporté ; que ce soit au niveau technique, évidemment, mais aussi au niveau humain (travail en groupe et en équipe).

Nous sommes heureux d'avoir pu faire ce projet dans une ambiance aussi agréable, entourés de gens aussi compétants.

10.7 Remerciements

Nous remercions la communauté du logiciel libre, notamment GNU, pour ces merveilleux outils de développement et pour en faire profiter l'humanité, et aussi le projet Debian, pour avoir inventé le plus beau système de paquetage du monde.

Nous tenons à remercier le LAP, et tout particulièrement René Beuchat, pour nous avoir fourni matériel, support, expérience et bonne humeur tout au long des projets de semestre et de diplôme.

Nous remercions aussi Cédric, Damien, Adrian et Dario, pour leur agréable compagnie et pour leurs conseils pertinants. Merci également à Judith Strauser pour les nombreuses corrections orthographiques.

Bibliographie

[1] ARM Architecture Reference Manual, David Seal, Addison-Wesley (ISBN 0-201-73719-1)

[2] Intel XScale Microarchitecture for the PXA250 and PXA210 Applications Processors User's Manual

[3] Intel PXA250 and PXA210 Applications Processors Developer's Manual, order number 278522-001

[4] Le noyau Linux, Daniel P. Bovet & Marco Cesati, édition française, O'Reilly (ISBN 2-84177-141-5)

[5] Linux Device Drivers, Alsessandro Rubini & Jonathan Corbet, O'Reilly (ISBN 0-596-0000-1)

[6] Linux Internals, Moshe Bar, McGraw-Hill, (ISBN 0-07-212598-5)

[7] Système d'exploitation Debian GNU/Linux http://www.debian.org

[8] Générateur de documentation Doxygen http://www.stack.nl/~dimitri/doxygen/

[9] Système de contrôle de version CVS http://www.cvshome.org/

[10] GNU Compiler Collection (GCC) http://gcc.gnu.org/

[11] The Insight Debugger (GDB) http://sources.redhat.com/insight/

[12] Protocole série de GDB http://sources.redhat.com/gdb/current/onlinedocs/gdb_32.html#SEC631

[13] GNU autoconf, automake and libtool http://sources.redhat.com/autobook/

[14] GNU autoconf manual http://www.gnu.org/manual/autoconf/

[15] GNU automake manual http://www.gnu.org/manual/automake/

[16] Bibliothèque d'accès à l'USB libusb http://libusb.sourceforge.net/

[17] Documentation NFS (NFS HOWTO) http://nfs.sourceforge.net/nfs-howto/

[18] Working with Ramdisk, Initrd, and Filesystemshttp://www.handhelds.org/handhelds-faq/filesystems.html

[19] Busybox – The Swiss Army Knife of Embedded Linux http://www.busybox.net/

[20] uClibc – a C library for embedded systems http://www.uclibc.org/

[21] PsiLinux – Linux for Psion Computers http://linux-7110.sourceforge.net/

[22] LAN91C96 – Non-PCI Single-Chip Full Duplex Ethernet Controller with Magic Packet http://www.smsc.com/main/catalog/lan91c96.html

[23] LAN91C111 – 10/100 Non-PCI Ethernet Single Chip MAC + PHY http://www.smsc.com/main/catalog/lan91c111.html

[24] UCB1400 – Audio codec with touch screen controller and power management monitor http://www.semiconductors.philips.com/pip/ucb1400.html

[25] LQ057Q3DC02 – Color TFT LCD Module http://www.sharp.co.jp/products/device/lcd/pdf/LQ057Q3DC02_LCY99073B.pdf

[26] CXA-L0505-NJL – LCD backlight inverter http://www.tdk.co.jp/tefe02/ea415_cxal05.pdf

[27] ATP-057 – Resistiv Touchscreen http://www.datamodul.com/pdf/Touch/atp057.pdf

[28] Resistiv touch-panels http://www.triangledigital.com/man2020f/ch7touch.htm

[29] UHC124 – UHC Host Controller http://www.transdimension.com/html/products_uhc124.html

[30] AS29LV800 – 3V 1M x 8/512K x 16 CMOS Flash EEPROM http://www.gaw.ru/doc/Alliance/as29lv800.pdf

[31] EzUSB – AN2131 – Final Datasheet http://www.cypress.com/cfuploads/img/products/AN2131SC.pdf