J3ea - Vol. 5 - Hors série N°2 (2006) CETSIS'2005, Nancy, 25-27 octobre 2005
DOI: 10.1051/j3ea:2006043

Simulateur Pédagogique d'Architecture Machine

Charles André[1]
[1]Université de Nice-Sophia Antipolis, Laboratoire I3S, BP 121, 06903, Sophia Antipolis cedex, andre@unice.fr

RESUME Les processeurs numériques utilisés en contrôle et traitement du signal ont fortement évolué. Une programmation en langage de haut niveau ne permet pas toujours de tirer le meilleur parti de ces machines. Leur utilisation efficace passe par une connaissance précise de leur architecture. Cet article décrit les grandes lignes de notre enseignement d'architecture machine en EEA. Il présente l'originalité de débuter par l'étude d'un simulateur pédagogique d'architectures de base que nous avons développé spécialement pour cette formation. Ce simulateur permet de s'assurer que les étudiants ont acquis les concepts de base, préalables à toute étude d'architectures évoluées.

Mots clés : processeurs numériques, architecture machine, programmation, simulation, autoformation.

1 INTRODUCTION

Les étudiants EEA de Licence et de Master 1 doivent acquérir des *connaissances précises* sur les *architectures des processeurs numériques*, nommés « machines » dans la suite. Les machines peuvent être spécialisées pour le contrôle (microcontrôleurs), pour le traitement de données (processeurs de traitement du signal – DSP) ou être d'usage général (ordinateurs). Les processeurs nouveaux exploitent plus largement les possibilités d'évolutions parallèles et combinent souvent plusieurs processeurs (architectures intégrant microcontrôleurs et DSP). Les architectures ont évolué pour répondre à des besoins de performance. Les solutions adoptées ne sont pas nécessairement révolutionnaires : il s'agit souvent de mise en œuvre d'idées anciennes rendues possibles par les progrès technologiques.

L'usage quotidien des ordinateurs pourrait laisser croire que les architectures elles-mêmes ont peu d'importance : une application peut être décrite en langage de haut niveau et son déploiement sur l'architecture confié à un compilateur. Cette vision n'est pas réaliste pour les applications à performances fortement contraintes (systèmes embarqués, systèmes temps réel). C'est justement à ce genre d'applications que les étudiants EEA sont confrontés. Pour les DSP et les microcontrôleurs, le langage de « haut niveau » est généralement le langage C. Une exploitation efficace des potentialités des processeurs passe souvent par l'écriture de procédures critiques en langage d'assemblage et par une organisation des données adaptée à l'architecture. Ces optimisations dépassent les possibilités des compilateurs usuels. C'est au programmeur de faire ces choix en fonction de l'architecture disponible. Une connaissance superficielle des architectures ne suffit pas. Il faut identifier les fonctionnalités des différentes unités, les chemins de données et de contrôle, ainsi que les contraintes liées à chaque type d'architecture.

Le retour d'expérience de travaux pratiques faits sur des machines pipeline et des machines VLIW (Very Large Instruction Word) montre qu'une grande partie des étudiants n'a pas suffisamment assimilé les concepts de base : jeu d'instructions, chemins de données et contrôle, allocation des données en mémoire et plus généralement l'application d'un modèle de calcul sur une architecture cible. Pour pallier ces lacunes, nous avons développé un *logiciel pédagogique de simulation* d'architectures séquentielles. Ce logiciel est utilisé dans la toute première séance de travaux pratiques pour illustrer les concepts, les mettre en œuvre et évaluer leur impact en termes de performance. Ce sont les caractéristiques de ce logiciel et leur exploitation en travaux pratiques qui sont décrites dans cette présentation.

2 CHOIX PEDAGOGIQUES

2.1 Contenu du cours

L'objectif principal du cours d'architecture est la connaissance des différentes architectures de machines, leurs avantages et inconvénients, l'étude de leur adéquation à certaines classes d'applications et leur mise en œuvre effective.

Le livre de W. Stallings intitulé « Computer Organization and Architecture » [1], qui en est à sa sixième édition, nous sert d'ouvrage de référence.

Dans une première partie du cours, nous décrivons rapidement l'*organisation générale des ordinateurs*. Une attention particulière est portée sur les mémoires internes et la notion de mémoire cache.

Nous introduisons ensuite les notions de *jeu d'instructions* et de *cycle d'instruction*. Le jeu d'instructions offre au programmeur une vision abstraite de sa machine et il a un impact direct sur l'exécution logique d'un programme. Le logiciel que nous présentons plus loin permet d'illustrer et d'approfondir ces notions.

La structure des *unités centrales de traitement* (CPU) est abordée par le biais des unités à un seul bus interne. Le microcontrôleur 68HC11 nous sert d'illustration pour ces structures de CPU. Nous introduisons ensuite le fonctionnement pipeline en prenant le processeur MIPS [2] comme exemple. Les processeurs super scalaires et les possibilités d'exécution dans le désordre sont évoqués en fin de cours.

Une étude plus détaillée est faite sur les *processeurs de traitements du signal* (DSP) en tant qu'*architecture spécialisée* et donc optimisée pour des missions particulières (le traitement du signal dans ce cas). Nous présentons d'abord un DSP travaillant en virgule fixe (famille TMS 320C5000) qui introduit des unités spécialisées (multiplieur, additionneur séparés et pouvant fonctionner en pipeline) et offre plus de chemins de données que les architectures classiques. Nous terminons avec un DSP bien plus performant (de la famille TMS320C6000 [3]). Ce DSP est équipé d'unités de calcul en virgule flottante et il est représentatif des architectures VLIW (Very Large Instruction Word). Ce DSP contient 8 unités de calculs qui travaillent effectivement en parallèle avec une grande richesse de chemins de données concurrents.

2.2 Les objectifs

La maîtrise des architectures passe par une bonne connaissance de l'organisation et l'architecture des machines de base (séquentielles). Le développement des applications demande également un minimum de discipline en matière de programmation. Il s'agit essentiellement de privilégier les phases d'analyse et de réutiliser un maximum de patrons de conceptions éprouvés pour un type d'architecture donnée. Nous avons donc retenus deux objectifs.

Le premier objectif est de permettre aux étudiants de se familiariser avec différentes vues d'une machine :

1. Une *vue architecturale*. La machine est considérée comme un ensemble d'unités fonctionnelles qui coopèrent et communiquent (échanges d'informations — données ou contrôle).
2. Une *vue programmation*. Elle s'appuie sur le jeu d'instructions. Ce modèle concerne l'écriture des programmes pour la machine étudiée.
3. Une *vue micromachine*. Elle explique les enchaînements d'opérations élémentaires et les chemins de donnée ou contrôle fins, qui réalisent les instructions.
4. Une *vue quantitative*. Des propriétés extra fonctionnelles sont prises en compte (taille mémoire et temps d'exécution).

Le second objectif est plus *méthodologique* : apprendre aux étudiants comment programmer correctement leurs applications, puis améliorer leur code pour des machines diverses.

2.3 Les moyens

Pour explorer les diverses vues d'une machine, nous avons développé un logiciel qui simule le fonctionnement d'une machine à différents niveaux d'abstraction.

Ce logiciel *a vocation pédagogique*. Il n'est en aucun cas destiné à créer de nouvelles architectures ou à développer des applications réelles. Il permet simplement d'illustrer le cours, d'expérimenter et de comparer des solutions. Il respecte les critères suivants :

1. *Autonomie* : Ce logiciel peut tourner sur toute plate-forme. Les étudiants peuvent sans problème l'installer sur leur machine personnelle et expérimenter à leur guise.
2. *Simplicité* : Seulement deux types de machines sont supportés : les machines à accumulateur et les machines à pile. Ces machines sont séquentielles. Les jeux d'instructions sont volontairement limités. Seules des opérations et des tests de base sont proposés. Les mémoires données et programmes sont séparées (architecture de type Harvard) et de taille très réduite.
3. *Rigueur* : La sémantique de chaque instruction est complètement définie. Elle est accessible en ligne sous forme de règle de transformation opérationnelle. Les registres non directement programmables (MAR : Memory Address Register ; MBR : Memory Buffer Register ; IR : Instruction Register) ainsi que le compteur programme (PC : Program Counter) et le registre d'état (Status Register) sont visualisés et modifiables. Certains de ces registres sont d'ordinaire non visibles par l'utilisateur. Le simulateur permet de révéler leur rôle dans le fonctionnement du processeur.
4. *Facilités de visualisation* : L'utilisateur peut choisir son mode d'affichage des données entières : décimal (signé ou non signé), hexadécimal ou binaire. Les mémoires d'instruction, de données et le contenu des différents registres sont visualisables. Les changements (données ou contrôle) sont mis en évidence après chaque exécution. Il est également possible d'ouvrir et de masquer les différentes fenêtres de l'application.
5. *Facilités (restreintes) d'évaluation de performance* : Le simulateur comptabilise les cycles et les accès mémoire lors de l'exécution d'un programme.
6. *Niveau d'abstraction* : Les simulations peuvent être faites selon trois modes : 1) instruction par instruction, 2) par succession de chargement d'instruction (instruction fetch) et exécution, 3) en distinguant les phases du cycle instruction (i-fetch, decode, data-fetch, execute, data-store).
7. *Absence de facilité d'assemblage* : Il s'agit encore d'un choix délibéré. Ce n'est pas un programme (l'assembleur) qui détermine les allocations en mémoire. La responsabilité de création des tables de symboles est laissée à l'utilisateur.

Le logiciel répond au premier des objectifs : la connaissance des architectures de base. En ce qui concerne l'amélioration de la qualité des codes développés, nous comptons surtout sur l'apprentissage. Durant les séan-

ces de travaux pratiques nous veillons à ce que les étudiants adoptent une approche systématique illustrée plus loin (Section 4).

2.4 Les raisons de ces choix

Les modèles sont simples, sans être simplistes. Ainsi, en un temps raisonnable, il est possible de *maîtriser le jeu d'instructions*. Le fait de ne pas utiliser d'assembleur permet à l'étudiant de prendre conscience qu'un processeur se contente d'exploiter les informations fournies, il ne les invente pas. Notre simulateur autorise l'usage d'opérandes et d'étiquettes symboliques. L'instruction « `load Alpha` » pour une machine à accumulateur, ne peut être interprétée correctement que si l'utilisateur a, au préalable, associé `Alpha` à une case mémoire précise (et de préférence l'a initialisé).

A côté de la simulation classique d'un programme instruction par instruction, le mode qui décompose les phases d'un cycle instruction se révèle utile pour expliquer le rôle exact joué par les registres non programmables. Cette compréhension est un préalable à l'étude des architectures pipeline que nous illustrons avec la machine MIPS. La machine MIPS est elle-même étudiée dans une autre séance de travaux pratiques avec le simulateur SPIM [4].

La visualisation de l'unité arithmétique et logique (ALU) et de la pile (dans le cas des machines à pile) révèle des *chemins de données*. Il s'agit de premiers apprentissages qui permettent de mieux comprendre le fonctionnement et les chemins de données des ALU pipeline qui réalisent les opérations MAC (Multiply Add aCcumulate) dans les DSP. La complexité est encore plus grande avec les architectures VLIW comme le TMS320C6711, qui est étudié en travaux pratique en tant que processeur de traitement du signal et en tant qu'architecture évoluée. L'aspect innovant de ces architectures ne peut être apprécié que lorsqu'on a bien assimilé le fonctionnement des processeurs « classiques ». La possibilité de comptabiliser le nombre de cycles et les accès mémoire requis par un programme permet de faire des premières *évaluations quantitatives* des architectures. Les évaluations quantitatives ont été popularisées par Patterson et Hennesy [5]. Les étudiants sont amenés à comparer les performances de programmes simples, solutions d'un même problème, écrits pour la machine à accumulateur et la machine à pile. Lorsque la machine à pile est programmée correctement (ce qui est rarement le cas), la diminution des accès mémoire améliore nettement les performances, même si le code est parfois plus long. Cette sensibilisation au caractère pénalisant des accès externes est exploitée dans le cours lors de l'étude de la hiérarchie mémoire.

3 LE SIMULATEUR D'ARCHITECTURE

Dans cette section nous présentons partiellement l'interface utilisateur et les fonctionnalités du logiciel appelé CPUSimulator. Une description complète est disponible dans la distribution [6].

3.1 Les principales fenêtres

Lors du lancement du logiciel, l'utilisateur choisit soit une machine à accumulateur, soit une machine à pile. Il apparaît alors le panneau de commande principal (Fig. 1). Celui-ci contient les registres liés aux chemins de données et de contrôle.

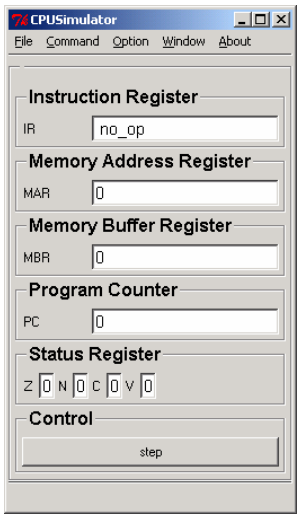

Fig. 1 : Panneau de commande principal

Suivant le type de machine choisi, l'unité arithmétique et logique associée est affichée dans une autre fenêtre (Figures 2 ou 3).

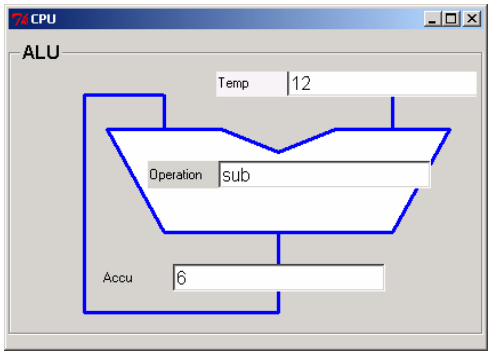

Fig. 2 : ALU pour la machine à accumulateur

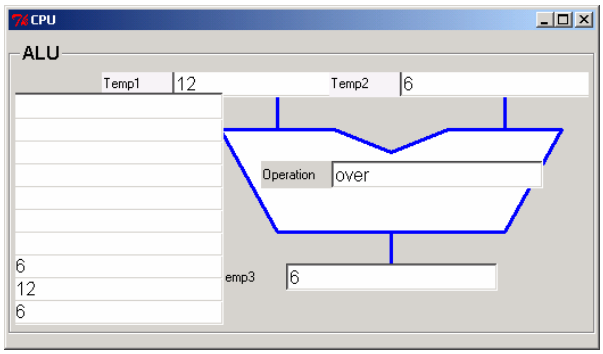

Fig. 3 : ALU pour la machine à pile

Une troisième fenêtre est également créée (Fig. 4). Elle représente le contenu des mémoires données et programme. La programmation se fait directement dans les cases mémoire et non pas par l'intermédiaire d'un éditeur de texte. La syntaxe des instructions (format BNF) est [étiquette':'] mnémonique [opérande | étiquette].
Le mnémonique doit être pris dans la liste des instructions (Figures 6 ou 7). Les étiquettes ou les opérandes sont généralement des identificateurs. Ils peuvent être des entiers interprétés comme des adresses ou des valeurs immédiates suivant l'instruction.

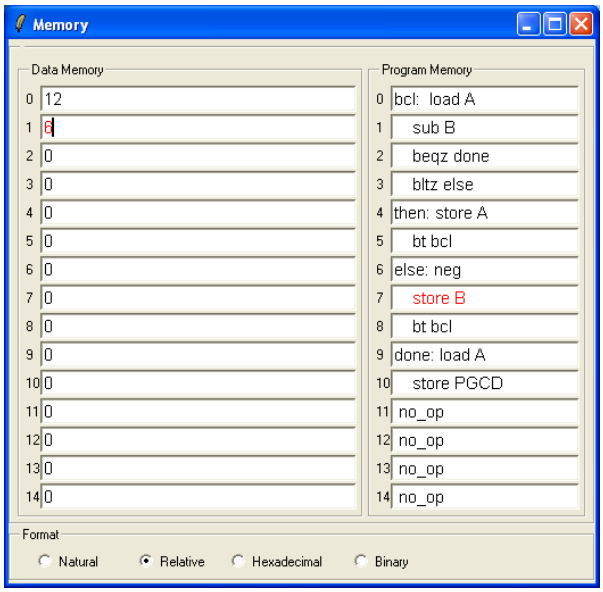

Fig. 4 : Exemple de programmation

Les associations identificateurs / cases mémoire se font par l'intermédiaire d'une fenêtre iconifiable (Fig. 5). Cette fenêtre est nommée Symbol Table. *L'utilisateur doit lui-même préciser* les associations identificateur / adresse mémoire. Il est clair que les associations étiquettes / adresses mémoire de programme auraient pu être aisément faites automatiquement. Nous avons préféré laisser ce travail à l'utilisateur afin qu'il se rendre bien compte des rôles différents joués par les adresses dans les deux types de mémoire.

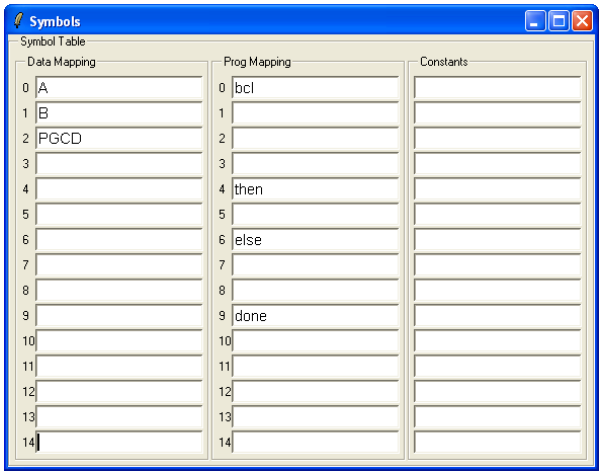

Fig. 5 : Association identificateur / mémoire

3.2 Exécution d'un programme

Un menu déroulant accessible à partir de la commande Option de la barre de menu permet de sélectionner le mode de simulation du programme :

Step :	instruction par instruction
Fetch/Execute :	2 phases : fetch et execute
Detail :	cycle instruction détaillé

La figure 4 montre le contenu des mémoires en phase de simulation. Les indications en rouge (gris plus clair sur les figures noir et blanc) sont les données qui viennent d'être modifiées (la case 1 en mémoire de données) ou l'instruction qui vient d'être exécutée (la case 7 en mémoire programme).

3.3 Sémantique des instructions

Sur la fenêtre contenant les instructions de la machine (Figures 5 ou 6), il suffit de positionner la souris pour que les effets de l'instruction (sa sémantique) apparaissent dans une bulle d'aide. Pour exprimer ces effets, les registres sont considérés comme des éléments d'un tableau Reg[ ], la mémoire de données est assimilée à un tableau Mem[ ]. L'adresse effective en mémoire de données obtenue après résolution des associations identificateur / adresse est notée EA (Effective Address). d dénote une adresse correspondant à une étiquette (adresse en mémoire programme) ou une valeur immédiate. Les modifications sont exprimées à l'aide des opérateurs du langage C. Le tableau 1 contient 3 illustrations relatives à la machine à accumulateur.

load	Reg[accu] ← Mem[EA]
lsr	C ← Reg[accu][0] Reg[accu] ← Reg[accu] >> 1
bltz	Reg[pc] ← if N then d else Reg[pc]+1

Tableau 1 : Sémantique d'instructions

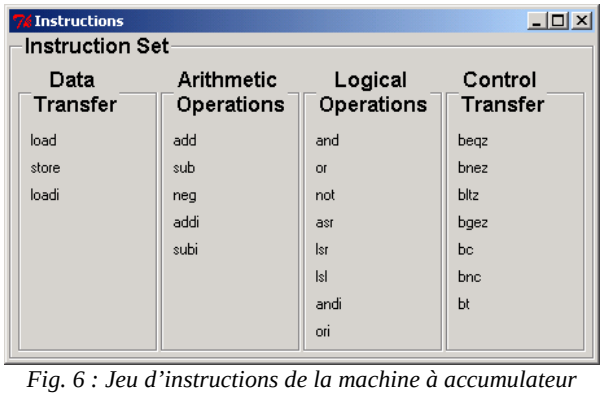

Fig. 6 : Jeu d'instructions de la machine à accumulateur

Dans le cas de la machine à pile, la pile est considérée comme une structure composée d'un tableau Stack[ ] et d'une variable Top qui contient l'indice indiquant la position courante du sommet de la pile dans le tableau.

A titre d'exemple, l'exécution de l'instruction over est décrite par les actions suivantes :

T ← Stack[Top – 1] ;
Top ++ ;
Stack[Top] ← T

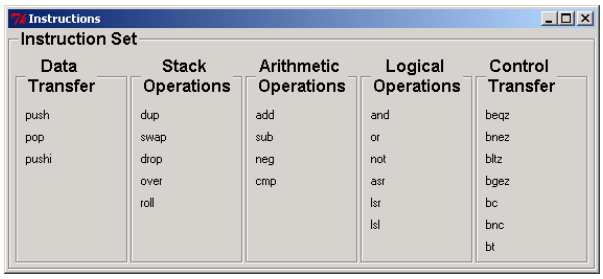

Fig. 7 : Jeu d'instructions de la machine à pile

3.4 Evaluation de performance

Il est possible de faire une évaluation des performances d'un programme. La fenêtre Statistics, qui est appelable à partir de la commande Window de la barre de menu, affiche trois valeurs entières : 1) le nombre d'instructions exécutées (Steps), 2) le nombre d'accès à la mémoire de données (Memory Accesses), 3) le nombre de microcycles (Micro-cycles). Ce dernier résultat est obtenu en comptant un microcycle pour chacune des phases i-fetch, decode, d-fetch, execute et d-store. Bien sûr, pour une instruction donnée toutes les phases ne sont pas forcément exécutées. La commande Reset accessible par la commande Command de la barre de menu remet à 0 tous ces compteurs.

3.5 Accès aux fichiers

La commande File de la barre de menu ouvre un menu déroulant qui permet de sauvegarder ou de charger des programmes. Les fichiers contiennent le code, la table des symboles et les éventuelles valeurs initiales. Il s'agit de scripts Tcl. Ils sont éditables mais aucune facilité n'est offerte à l'utilisateur pour les modifier.

3.6 Plates-formes

Le logiciel a été entièrement écrit en Tcl-Tk. Il faut avoir une version 8.4 ou ultérieure. La bibliothèque BWidget 1.7 est également utilisée. Tcl-Tk et BWidget sont disponibles pour de nombreuses plates-formes sur le site de SourceForge (http://sourceforge.net). Le simulateur lui-même est accessible sur le site Web de l'auteur (http://www.i3s.unice.fr/~andre rubrique Software). La procédure d'installation y est également donnée.

4 PROGRAMMATION

4.1 L'approche préconisée

Les étudiants sont amenés à programmer en langage C et dans des assembleurs dédiés aux différentes machines étudiées (les machines à accumulateur et à pile du simulateur, le Pentium, le MIPS et les DSP). Ceci fait beaucoup de langages, avec des syntaxes souvent très différentes. Il faut donc de la méthode et savoir réutiliser au maximum.

Nous insistons sur la nécessité d'une analyse préalable à haut niveau, avant toute programmation. Durant toutes les séances de travaux pratiques nous veillons à ce que les étudiants adoptent l'approche suivante :

1. Analyse soignée du problème et expression d'une solution sous forme algorithmique.
2. Écriture d'un programme en C (langage supposé bien maîtrisé) pour valider la solution (a-t-on répondu à la question posée ?).
3. Traduction en langage d'assemblage de la machine cible et test de correction. Pour la traduction des patrons ont été donnés aux étudiants.
4. Évaluation de performances et optimisations.

Cette démarche n'est pas linéaire. Elle peut nécessiter des retours aux étapes précédentes.

4.2 Les recommandations

Certaines recommandations relèvent de l'*activité de programmation en général*. Elles ont certainement été données par ailleurs. Il est toutefois utile de veiller à leur application. Nous insistons en particulier sur la bonne programmation des boucles. Pour chaque boucle, déterminer les préconditions, les postconditions et les invariants. Pour les boucles « tant que » sensées se terminer vérifier l'existence d'un variant monotone qui assure la sortie de boucle en un nombre fini d'itérations.

A ces recommandations générales nous ajoutons celles plus spécifiques à la *programmation de bas niveau*. Par exemple dans les boucles, lorsque l'indice d'itération n'est pas utilisé dans le corps de la boucle, l'usage d'un *décompteur* est préférable.

Le code doit être *documenté*. Le plus simple est l'usage de commentaires à condition qu'ils soient pertinents. Nous luttons contre les commentaires du genre :

load A ; accu <- le contenu de A

qui ne font que paraphraser l'instruction. Le commentaire utile lie l'instruction à l'algorithme ou bien il reflète une décision du concepteur. Pour les machines à pile, il est indispensable de monter le contenu de la pile, ou tout au moins de sa partie supérieure. C'est le meilleur moyen pour s'assurer que la structure de la pile est invariante à chaque itération dans une boucle.

4.3 Les exercices proposés

Dans la première séance de travaux pratiques qui utilise le *simulateur d'architecture*, les exemples sont forcément simples. Ils sont prétextes à l'écriture de structures de contrôles (itérations, tests), d'opérations arithmétiques simples et de manipulations de tableaux de bits. Les exercices proposés sont le calcul du PGCD (par le premier algorithme d'Euclide, utilisant la soustraction), des déplacements de bits (chenillards et ses variations), génération de nombres de la suite de Fibonnaci, transcodage, …

Une deuxième séance est consacrée au *Pentium*. Les étudiants travaillent sous Visual C++. Les programmes sont essentiellement écrits en C et l'environnement de développement permet d'accéder au code assembleur engendré par le compilateur. Dans cet exercice il s'agit surtout de comprendre comment sont exécutés à bas niveau les programmes C. La programmation en assembleur est ici une simple programmation par différence : les étudiants doivent optimiser le code généré par le compilateur (les programmes ont été compilés sans optimisation).

La troisième séance de travaux pratiques utilise le *simulateur SPIM* pour la machine MIPS. C'est pour les étudiants le premier contact avec une machine RISC et ses registres généraux. Ils doivent réécrire les solutions développées lors de la première séance. Cet exercice est très formateur car ils se rendent compte que le même algorithme se programme de façon très différentes sur les machines CISC (Complex Instruction Set Computer) et les machines RISC (Reduced Instruction Set Computers). La séance se termine par une initiation au pipeline. La simulation est exécutée en mode avec branchement retardé. Les programmes doivent être modifiés pour tenir compte de ce fait.

La quatrième séance porte sur le *microcontrôleur* HC11. Le travail consiste surtout à programmer correctement les entrées sorties, en simulation et ensuite pour commander effectivement un système clavier/afficheur.

Le dernier processeur étudié est un DSP. L'étude est faite sur une *carte de développement* à base du TMS320C6711 et en utilisant l'environnement de développement de Texas Instrument. La première étape est la programmation en C d'un produit scalaire. Il faut ensuite améliorer les performances du code en introduisant le parallélisme, puis en programmant le pipeline (l'ouvrage de R. Chassaing [7] est largement utilisé). La séance se termine par la programmation d'un filtre numérique à réponse impulsionnelle finie.

5 CONCLUSION

Nous avons présenté les grandes lignes de nos enseignements d'architecture de processeurs numériques en Licence et Master 1 EEA. Pour être utile à nos étudiants, un tel enseignement doit dépasser la simple classification ou typologie des architectures machines. Les évolutions architecturales ont eu un impact important sur les processeurs utilisés dans les applications embarquées (microprocesseurs, microcontrôleurs, processeurs de traitement du signal). Les performances, spécialement en rapidité d'exécution, ont été considérablement améliorées. L'enseignement a été conçu pour permettre à l'étudiant de *comprend*re ces évolutions et de savoir les *exploiter*. Cette compréhension et cette exploitation demandent une certaine aisance avec les notions d'unités fonctionnelles, de chemins de données et de contrôle, ainsi que de bonnes dispositions en matière de programmation. C'est pour faciliter l'assimilation de ces notions que nous avons développé le simulateur de machines de base décrit dans cet article. Ce simulateur est uniquement à vocation pédagogique et nous l'utilisons au tout début de l'enseignement d'architecture. Il permet de faire rapidement le point sur les acquis (et lacunes) des étudiants et de les préparer à l'étude d'architectures sophistiquées. Son usage peut être étendu à d'autres cursus. Nous souhaitons partager notre expérience avec d'autres collègues. En fonction de leurs réactions le produit peut évoluer. Il doit toutefois rester simple et se focaliser sur les concepts essentiels. Ce ne sera jamais un outil de développement. L'introduction d'un mode d'adresse indirect est par exemple une décision à débattre.

Bibliographie

1. Stallings, W. *Computer Organization and Architecture*. (6th edition) Prentice-Hall, Inc. Upper Saddle River (NJ), 2003.
2. Patterson, D.A., Hennesy, J.L. *Computer Organization & Design*. Morgan Kaufmann Publishers, Inc. San Francisco, 1994.
3. Dahnoun, N. *Digital Signal Processing Using the TMS320C6000 platform*, Prentice Hall, ISBN: 0-201-61916-4 (2000). Un CD ROM du même auteur, intitulé *TMSC6000 Teaching ROM*, est fourni gratuitement par Texas Instrument pour l'enseignement.
4. Larus, R.L. *Assemblers, Linkers, and the SPIM Simulator*. Appendix A in [1]. Simulateur disponible sur le site http://www.cs.wisc.edu/~larus/spim.html
5. Patterson, D.A., Hennesy, J.L. *Computer Architecture: A Quantitative Approach*. (2nd edition) Morgan Kaufmann Publishers, Inc. San Francisco, 1996.
6. André, C. *CPU Simulator*. Logiciel et documentation. Version 2.5, July 2005. http://www.i3s.unice.fr/~andre/software.html
7. Chassaing, R. *DSP: Applications using C and the TMS320C6x DSK*. John Wiley & sons, Inc, New York, 2002.