Architecture et organisation des ordinateurs

GUIDE DE L'ÉTUDIANTE ET DE L'ÉTUDIANT
S4 – APP3i

Hiver 2024

Département de génie électrique et de génie informatique
Faculté de génie
Université de Sherbrooke

Historique des modifications

Date	Responsables	Description
2 Février 2024	Marc-André Tétrault	Version initiale 2024.

Auteur : Marc-André Tétrault, Sébastien Roy, Daniel Dalle
Version : 2024.1 (2 Février 2024)

Ce document est réalisé avec l'aide de LaTeX et de la classe `gegi-app-guide`.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

1 ÉNONCÉ DE LA PROBLÉMATIQUE

Le maillage des circuits logique en processeur efficace

Le Bureau Administratif de la Recherche Nationale en Aérospatiale du Québec (BARNAQ[1]) développe des satellites d'observation météo pour étudier le réchauffement climatique. L'équipe de R&D souhaite renforcer leur lien de communication numérique à l'aide de l'algorithme de correction d'erreur Viterbi. Cependant, l'ajout de cet algorithme ne laisse aucune marge d'exécution pour les quelques tâches connexes également gérées par le processeur embarqué, processeur à architecture MIPS doté d'une mémoire DRAM. L'équipe de R&D doit donc explorer différentes solutions d'accélération logicielles et/ou matérielles pour permettre à la fois les calculs existants et l'ajout de la correction d'erreur. Heureusement, la partie de l'algorithme la plus exigeante en calcul est bien connue.

mandat

Votre chef d'équipe propose de diviser le mandat en 4 parties : 1) traduire en assembleur MIPS la boucle de calcul matriciel de l'algorithme de correction d'erreur. 2) Évaluer quantitativement la différence de performance entre une organisations MIPS unicycle et pipeline. 3) Calculer et valider les performances d'une organisation mips unicycle avec trois agencements différents de mémoires : SRAM, DRAM et DRAM+Cache. 4) Proposer un système hypothétique permettant d'exécuter rapidement et effifacement la boucle de correction d'erreur en catégorisant les choix de conception par niveau d'effort requis et par gain de performance anticipé.

L'algorithme L'algorithme de Viterbi est un concept largement utilisé pour la correction d'erreurs dans les communications numériques et pour le stockage de données sur médium magnétique (disques durs). L'algorithme, déjà disponible et validé en langage C, revient essentiellement à une recherche du plus court chemin dans un graphe directif structuré, appelé *treillis*. Le but ultime est de corriger de potentielles erreurs de réception en déterminant quel trajet à travers tout le treillis obtient le coût (somme de métriques) minimum.

La boucle à optimiser constitue le coeur de l'algorithme de Viterbi, et consiste au calcul des trajets survivants à l'aide d'additions, de comparaisons et de la sélection de la plus petite valeur (ACS). En plus d'une version validée de l'algorithme de calcul des survivants, le code de référence fourni en C inclut également un message typique qui peut servir pour la vérification.

1. Les Cyniques, https://fr.wikipedia.org/wiki/Les_Cyniques

Votre travail débutera par l'analyse et la transcription de cette boucle en langage assembleur MIPS standard. Vous devrez inclure un test unitaire validant votre transcription du code. L'ensemble du code assembleur doit également respecter les conventions de rédaction MIPS pour garantir sa compatibilité avec d'autres projets qui exploiteront ces fonctions. Vous avez accès à l'outil de développement *MARS* pour appuyer votre travail de développement et de tests logiciels. De plus, *MARS* offre différents modules d'analyse statistique utiles à la quantification des performances.

Le processeur Vous étudierez ensuite deux variantes d'organisations matérielles de processeurs : MIPS unicycle et MIPS pipeline. Ce travail se fera entièrement en VHDL, avec des simulations fonctionnelles (simulations VHDL dans Xilinx/Vivado) pilotées par des codes de tests en assembleur préparés sur *MARS*. La BARNAQ a déjà prévu une activité de formation (laboratoire 1) sur la modification des organisations, au cas où vous devrez ajouter des fonctionalités manquantes, et vous fournira ensuite un code de référence VHDL plus élaboré. L'équipe a également en main un prototype d'organisation avec pipeline, mais sans unité d'avancement ou de détection des aléas (*hazards* en anglais). Votre équipe sait que ces omissions détérioreront significativement les performances et a déjà réservé une période pour mettre en place l'unité d'avancement (laboratoire 3).

Pour faciliter la comparaison des performances entre ces deux organisations, l'équipe logicielle vous fournira un code de référence standardisé de calcul matriciel. Grâce à cela, même s'il manque l'unité de détection des aléas au pipeline, vous pourrez rapidement quantifier par calcul théorique la différence de performance des organisations.

Suite à des discussions avec votre chef d'équipe et compte tenu des fonctions futures, vous savez déjà que le code du calcul des survivants devra se compléter en trois fois moins d'instructions exécutées, sans tenir compte des accès mémoire. Votre intuition remet à votre esprit le concept d'*extensions* spéciales qui permettent de traiter plusieurs données en un seul cycle d'horloge (SIMD, *Single Instruction Multiple Data*). Vous croyez que des ajouts ciblés de ce type pourraient offrir une accélération intéressante de la boucle de calcul ACS. Votre chef d'équipe est intéressé par l'idée et vous demande de commencer le travail par une étude théorique de la performance. Cette étude inclut l'analyse des performances sans mémoire cache, le choix des extensions et un court code d'étude de cas.

Si l'étude promet de bons résultats, vous aurez ensuite à développer une preuve de concept basée sur l'organisation unicycle à SRAM (sans mémoire cache), plus simple à déverminer. Les livrables à ce niveau sont une mise à jour du plan de vérification (pour les extensions), leur implémentation en VHDL, l'adaptation du code assembleur ACS pour exploiter ces

nouvelles extensions originales, et enfin une validation système par simulation fonctionnelle (dans Xilinx/Vivado).

Optimisation de la cache L'implémentation du processeur utilise une très grande mémoire externe de type DRAM avec un coût d'accès de 10 cycles d'horloge, et où chaque accès fourni séquentiellement 2 mots voisins de 32-bits (*2-word burst*) alignés sur l'adresse inférieure. Ce coût d'accès ralentit significativement la performance de toute forme de code. L'ajout de mémoire SRAM rapide est la solution idéale mais beaucoup trop coûteuse. L'ajout de petites mémoires caches distinctes pour les instructions et les données devient donc essentiel à l'atteinte de bonnes performances.

Heureusement, la compagnie possède un générateur de cache paramétrable ultra efficace en énergie et en vitesse. La configuration de la cache d'instructions est déjà fixée et sera testée dans une période prévue (laboratoire 2), test restreint à l'organisation unicycle. À l'aide du code de référence standardisé de calcul matriciel, vous devrez valider mathématiquement les temps de simulation obtenus pour les trois agencements de mémoire.

Également en utilisant le code de référence de calcul matriciel fourni, vous devrez finaliser la configuration pour la *cache de données*, incluant sa taille totale, son organisation (nombre de blocs, taille des blocs) et son type d'adressage (direct, associatif par ensemble, associatif intégral) pour tirer profit à la fois de la localité spatiale et temporelle des caches. La priorité ici sera de maximiser le taux de succès dans le code de référence, avec en deuxième plan minimiser la taille de la cache en nombre de mots "utiles".

Gains de performance et intégration Les travaux précédents étudients séparémment l'impact de chaque aspect sur les performances, il reste donc à évaluer leur impact cumulatif. De plus, l'équipe de gestion a besoin d'une analyse quantitative détaillée pour juger du rapport *gain de performance* vs *effort de développement* dans le contexte du calcul des survivants. En construisant sur les bases théoriques d'analyses que contiendra votre rapport, vous devez prédire la performance d'un système final *hypothétique* combinant les différentes stratégies d'accélération (ajout de SIMD, migration vers le pipeline, ajout de cache de données, peut-être optimisations logicielles et/ou algorithmiques). Vous devrez commenter de l'impact de chacune des modifications, lorsqu'entremêlées, possiblement différent que lorsque qu'analysés individuellement. Ce rapport permettra de débloquer les budgets de la BARNAQ pour la phase avancée de recherche et de développement.

2 GUIDE DE LECTURE

2.1 Références essentielles à consulter

1. *Computer Organization and Design*, 5e édition de D. Patterson et J. Hennessy [1], référé par l'abbréviation *COD5*. Depuis le campus ou RPV, disponible ici.
2. *Concepts de base de la programmation en assembleur MIPS* de J. Rossignol [2] (voir site web)
3. *Digital Systems Design Using VHDL* de C. H. Roth, Jr. et L. K. John [3].

2.2 Documents d'accompagnement de la problématique

— Carte de référence MIPS, tiré de du manuel [1], disponible à l'examen.
— *Lire un chemin de données*, de J. Rossignol (sur le site web de l'APP).

2.3 Autres références

Ces références sont fournies à des fins de documentation complémentaire et ne sont pas indispensables pour résoudre la problématique.

— Ressources complémentaires du manuel [1].
— Capsules vidéo basées sur les figures du manuel, par David Black-Schaffer, Université d'Uppsala, Suède. Lessons d'intérêt :
  — Lesson 3 - langage machine et assembleur.
  — Lesson 6 - sur le unicycle.
  — Lesson 7 - sur le pipeline.
  — Lesson 9 - pipeline : pénalités de bulles et branchements.
  — Lesson 11 - sur les caches.
— *See MIPS Run* de D. Sweetman, [4], disponible à la bibliothèque de l'université (QA76.9A73S93 2007).
— Site web de la firme MIPS, http://www.mips.com.
— *MIPS RISC Architeccture* de Gerry Kane et Joe Heinrich [5].
— *The Designer's Guide to VHDL* de Peter J. Ashenden [6].
— Steve Rhoads, projet PLASMA (coeur compatible MIPS 32 en VHDL à source ouverte), http://opencores.org/projects/plasma.

2.4 Guide de lectures

2.4.1 Préparation au procédural 1 et au laboratoire 1

Lectures essentielles (inclusivement) :

- Abstraction et technologies de l'ordinateur : [1] — section 1.3.
- Assembleur et langage machine :
  - Dans le guide d'assembleur [2], jusqu'à 3.3 ;
  - Dans COD5 [1] — Chapitre 2, sections 2.1 à 2.7, 2.10 et 2.19 ; la page 97 (de 2.8).
  - Dans COD5 [1] — Annexe A.2
  - Carte de référence MIPS (dans [1], disponible sur le site de l'APP)
- Arithmétique des ordinateurs [1] — chapitre 3, 3.1–3.2.
- Architecture interne d'un microprocesseur [1]—- chapitre 4, 4.1–4.4.

Lectures complémentaires :

- "Lire un chemin de données", de J. Rossignol (sur le site web de l'APP)
- Dans COD5 [1] — Annexe A, A.10 (Assembleur) ;
- Descriptions comportementales et structurales en VHDL — pp. 85–94 de [3]
- Dans COD5 [1], construction d'une unité arithmétique — Annexe B, section B.5 .

2.4.2 Préparation au procédural 2

Lectures essentielles :

- Appels de fonction et convention MIPS :
  - Dans le guide d'assembleur [2], de 3.4 jusqu'à la fin.
  - Dans COD5 [1] — Chapitre 2, section 2.8 ;
  - Dans COD5 [1] — Annexe A, section A.6.
- Mesure de la performance : [1] — sections 1.6 et 1.10.
- SIMD, vecteurs et parallélisme
  - Dans COD5 [1] — Chapitre 6, section 6.3 ;

2.4.3 Références pour le devoir

Références essentielles :

- Dans le guide d'assembleur [2], de 3.4 jusqu'à la fin.
- Dans COD5 [1] — Chapitre 2, section 2.8 et 2.13 ;
- Code de Viterbi en version C (fourni sur le site web).
- Annexe sur l'algorithme de Viterbi (site web de l'APP), sections 2.3, 3.2, 3.3

Lectures complémentaires :

- Section restantes de l'annexe sur l'algorithme de Viterbi.

2.4.4 Préparation au procédural 3 et au laboratoire 2

Lectures essentielles :

- Hiérarchies de mémoire, mémoires caches (directes, associatives, associatives par ensemble)
  - Dans COD5 [1] — Chapitre 5, sections 5.1–5.3, 5.9, 5.15.

2.4.5 Préparation au procédural 4

Lectures essentielles :

- Organisation en pipeline, aléas (données et contrôles), unités d'avancement : [1] — sections 4.5–4.8 ; 4.14.
- Calculs de performance : [1] — sections 1.6, 5.4, 5.8

3 LOGICIELS ET MATÉRIEL

3.1 Logiciels requis

— Simulateur MARS (MIPS Assembler and Runtime Simulator). En Java (compatible Windows, Mac et Linux).
https://courses.missouristate.edu/KenVollmar/MARS/
— Vivado Design Suite de Xilinx, pour le développement et la simulation en VHDL.
— Meld, logiciel de comparaison de fichiers sources
https://meldmerge.org

3.2 Logiciels d'intérêt

— Éditeur pour les fichiers sources (Notepad++, Crimson Editor, Emacs, ...)
— Visual C++ sur Windows ou GNU C++ sur Unix ou Mac OS pour valider les algorithmes en C. http://www.edumips.org/
— EDA Playground, logiciel web de prototypage de programmes VHDL (ainsi que d'autres langages) : https://www.edaplayground.com/.

4 SOMMAIRE DES ACTIVITÉS

Semaine 1

— Première rencontre de tutorat.
— Étude personnelle et exercices.
— Procédural sur l'assembleur et une organisation unicycle.
— Laboratoire sur l'organisation unicycle.

Semaine 2

— Remise du devoir.
— Lecture sur les caches.
— Procédural sur les caches.
— Amorce du rapport d'APP.
— Laboratoire sur les performances des mémoires.
— Lectures sur l'organisation en pipeline.
— Procédural sur l'organisation en pipeline.

Semaine 3

— Remise du plan de vérification formatif.
— Laboratoire sur l'unité d'avancement dans une organisation pipeline.
— Remise du rapport.
— Validation des instructions SIMD.
— Amorce et développement de la problématique.

Semaine 4

— Remise du plan de vérification.
— Remise et validation des livrables d'APP (VHDL, figure d'organisation SIMD, assembleur SIMD).
— Tutorat de fermeture.
— Évaluation formative théorique et pratique .
— Consultation pré-examen.
— Évaluation sommative théorique et pratique.

5 PRODUCTIONS À REMETTRE

— Les productions se font par équipe de 2, sauf lorsque indiqué autrement.
— L'identification des membres des équipes doit être faite sur la page web de l'unité à la fin de la journée du procédural 2.

Avertissement

Dans le contexte de cette activité d'apprentissage par problème, une collaboration entre équipes est légitime pour élaborer une méthode et des pistes de solution. Cependant, les travaux et documents soumis pour évaluation doivent être une production originale individuelle ou déquipe, selon le type de travail. Le non-respect des règlements de l'Université concernant lintégrité intellectuelle peut entraîner des sanctions académiques ou disciplinaires.

Retards

Tout retard sur la remise de livrable entraîne une pénalité de 20 % par jour.

Sommaire

Production	Date de remise	Type de livrable	Type d'évaluation
Devoir	lundi 12 Février 2024, 10h00	asm	Sommative
Plan de vérification	lundi 19 Février 2024, 9h00	pdf	Formative
Rapport	mardi 20 Février 2024, 17h00	zip	Sommative
	Texte du rapport Code viterbi du devoir (corrigé) Code viterbi SIMD Code de référence SIMD	pdf asm asm asm	
Devis de conception	lundi 26 Février 2024, 9h00	pdf	Sommative
	Spécifications SIMD sommaires Plan de vérification Figure d'organisation		
Validation	lundi 26 Février 2024, 13h00	zip	Sommative
	Démonstration en labo	projet Vivado	

5.1 Devoir (individuel)

L'objectif du devoir est de développer très tôt vos compétences individuelles en programmation MIPS et de vous fournir une rétroaction détaillée sur les exigences à ce niveau pour cet APP. Plutôt que de vous demander un code quelconque (ce qui aurait pu être le cas), on vous demande ici de traduire le calcul central à la problématique. **Il s'agit d'un travail individuel** : des copies aux similitudes marquées provenant d'un binôme en équipe se verront amputées

de 50% de la note totale du devoir. Une fois le devoir complété et la rétroaction (rapide) reçue, vous pourrez ensuite choisir un des deux codes pour le projet, et même comparer la différence de temps d'exécution, ce qui pourrait être intéressant au rapport.

Vous avez à remettre comme devoir individuel une version MIPS de la procédure calculant les survivants, tel que définie dans le code *DecodageViterbi.c* fourni sur le site de l'APP (voir la section 2.4.3 pour connaître les lectures essentielles et/ou complémentaires à faire). En plus des points d'entrée pour les deux fonctions d'intérêt (CalculSurvivants et acs, conservez ces noms exacts), votre code assembleur doit contenir un point d'entrée "main" qui effectue un ou des tests unitaires sur le calcul des survivants. Ce ou ces test(s) sont à votre choix, mais ils doivent confirmer l'équivalence entre le code C et votre implémentation en assembleur. La fonction *genmetric* n'est pas demandée.

Notez que vous pouvez vous servir du code C pour préalablement générer des métriques et les placer à l'avance dans le segment de données. L'ensemble du code doit respecter les conventions de programmation en assembleur MIPS tel que défini dans le guide assembleur de l'APP [2, voir site web]. Vous devez également inclure des commentaires expliquant sommairement les étapes réalisées par votre code, et organiser clairement le contenu du fichier.

Vous remettrez votre devoir individuel sous forme de fichier assembleur (.asm) avant lundi 12 Février 2024, 10h00. Le code sera évalué selon la grille de la table 6.1. Le code sera visuellement inspecté puis fonctionnellement validé dans un environnement de vérification automatisé. La rétroaction sera rapide, afin de vous permettre de mettre à jour le code avant la remise du rapport. Notez que la grille, après légères adaptations, sera utilisée lors des évaluations sommatives subséquentes portant sur la rédaction de code assembleur.

Suggestion : pour vous aider à tester votre respect des normes MIPS, imaginez que vous devez remplacer votre fonction CalculSurvivant assembleur par celle de votre coéquipier, sans lui parler de vos registres utilisés. Cela pourrait vous aider à mieux comprendre des cas de figures de normes et les corriger avant la remise. Enfin, tout code généré avec un compilateur se verra attribué la note 0(oui oui, ça se voit facilement !).

5.2 Plan de vérification formatif

Le but de cette activité est de valider vos choix d'instructions et de vous offrir une rétroaction rapide avant la rédaction du code VHDL. Dans votre document de maximum 3 pages, présentez brièvement les instructions SIMD que vous aurez choisies (vous pouvez renvoyer aux annexes) et votre plan de vérification vis-à-vis de ces instructions. À ce niveau, imaginez au moins un test par instruction qui tente de prouver qu'il existe une erreur dans le futur

code VHDL de vos collègues. C'est plus efficace de prendre le point de vue de l'avocat du diable... Votre plan sera évalué sommativement à la validation.

Remettez votre plan de validation en format PDF avant le lundi 19 Février 2024, 9h00. Le nom du fichier doit suivre le format *cip1-cip2.pdf* où *cip1* et *cip2* sont les CIP de chaque étudiant de l'équipe. Il ne faut pas oublier d'insérer le tiret entre les deux CIP. Le système n'acceptera pas le dépôt si le CIP de l'usager effectuant le dépôt n'est pas indiqué dans le nom du fichier.

5.3 Rapport

Le rapport d'APP est à remettre le mardi 20 Février 2024, 17h00. Il doit être remis par binôme (équipe de 2) et contient deux parties : 1- un document manuscrit (PDF) et 2- trois fichiers de code (voir en 5.3.2). Les deux parties seront évaluées selon la grille de la table 6.2.

5.3.1 Document manuscrit

Le rapport manuscrit doit répondre aux questions posées par l'énoncé de la problématique : **relire la problématique**. Plusieurs indications et indices y sont écrits et beaucoup moins mystérieux à ce stade qu'au premier tutorat. En complément au texte de la problématique, les quelques précisions ci-bas :

1. **Performances d'organisations** : À l'aide du code de référence de calcul matriciel, documentez sous forme d'équations algébriques et calculez son temps d'exécution **en cycles d'horloge** pour l'organisation unicycle. De même, identifiez et calculez les pénalités et retards causés par deux organisation en pipeline (procédural 4), soit l'organisation avec branchement au 4e étage (COD5 figure 4.51) et l'organisation avec branchement au 2e étage, unité d'avancement et unité de détection des aléas (COD5 figure 4.65). Enfin, calculez le temps d'exécution en vous basant sur les vitesses d'opération des organisations suivantes :
   — 25 ns pour l'organisation unicycle.
   — 10 ns pour l'organisation pipeline.
2. **Performance SIMD** : Dans le code de référence, identifiez les instructions qui seraient à convertir en SIMD, puis calculez le nouveau temps d'exécution en cycles d'horloge, pour enfin le comparer avec celui en unicycle. Est-ce que le gain de performance permet d'anticiper qu'il est possible d'atteindre les objectifs du projet ?
3. **Performances des mémoires sur processeur unicycle** : À l'aide du code de référence de calcul matriciel sans SIMD, et en vous basant sur une organisation unicycle, calculez le nombre de coups d'horloge supplémentaires due **aux données** pour le cas de la

DRAM de la problématique, et ensuite pour le cas d'une cache de données attachée à cette DRAM utilisant 256 blocs, 2 mots de 32-bits par bloc et l'écriture à la DRAM en *write-through*. Documentez votre calcul sous forme d'équations algébriques avant de produire le résultat numérique. Notez que cet exercice est possible avec la version en C du code.

Répétez ensuite l'exercice pour les pénalités d'accès **aux instructions**, pour le cas avec DRAM seule et cas avec DRAM+cache d'instructions de 256 blocs, 2 mots de 32-bits par bloc. Dans ces cas-ci le code assembleur est nécessaire.

4. **Configuration des caches** : Définissez et justifiez votre choix pour la configuration de la cache de données minimisant à la fois sa taille et les pénalités d'accès aux données (en coups d'horloge).

   Note importante : une étude uniquement faite à l'aide de l'outil dans MARS produira automatiquement un niveau de 1/4 pour cette partie du rapport.

5. **Intégration** : En partant cette fois d'une organisation unicycle avec mémoire DRAM (comme celui utilisé au labo 2), proposez un *système* optimal et priorisez les modifications par leurs gains effectifs sur la performance et leur temps de développement pressenti. (ex : extensions SIMD, restructuration du code assembleur, etc.). Choisir et justifier un changement à laisser tomber si le calendrier de développement devait être devancé (temps limité).

Chacun de ces éléments doit être accompagné d'une description adéquate pour en faciliter la compréhension et pour justifier vos calculs ou choix de conception. Le rapport doit suivre le guide de rédaction remis en S1 et vous avez jusqu'à 8 pages pour le contenu. Ces 8 pages excluent la page titre, la table des matières, l'introduction, la conclusion et les références. Votre rapport doit être **en format PDF**, et le nom du fichier doit suivre le format *cip1-cip2.pdf* où *cip1* et *cip2* sont les CIP de chaque étudiant de l'équipe.

5.3.2 Codes assembleur de la problématique

Suite à la rétroaction du devoir, corrigez au besoin votre code assembleur Viterbi. Dérivez de ce code une nouvelle version exploitant les instructions SIMD que vous avez choisies d'implémenter et documentées dans votre plan de vérification formatif. Remettez également la version SIMD du code de référence. Ces trois codes sont considérés comme faisant partie du rapport. Vous devez donc les remettre dans la boite de dépôt Moodle prévue. Le nom des fichiers doit être :

— cip1-cip2.viterbi.asm

— cip1-cip2.viterbi.simd.asm
— cip1-cip2.reference.simd.asm

5.4 Validation sommative

5.4.1 Code VHDL d'organisation

Vous devez produire une organisation unicycle avec extension SIMD. Les détails sur différentes approches possibles sont dans l'annexe D. Vous devez déposer votre archive Vivado avant le lundi 26 Février 2024, 13h00, juste avant la période de validation.

5.4.2 Devis de conception

Le devis de conception sommatif doit contenir la liste finale des instructions SIMD choisies par l'équipe, leur spécification sommaire et une figure démontrant comment vous avez implémenté vos instruction SIMD. Une approche possible est de modifier la figure COD5-4.17 avec un logiciel de dessin.

Le document doit également inclure un plan de vérification. Il doit décrire les points à surveiller lors des simulations pour confirmer le bon fonctionnement des instructions SIMD, et une liste de tests unitaires par instruction. Vous pouvez vous inspirer du tableau 11.1 pour la disposition. L'important est que l'information soit regroupée par instructions.

Remettez votre document sous format .pdf pour le lundi 26 Février 2024, 9h00. Vous avez 1/2 page pour la liste et la spécification sommaire, 1 page pour la figure et maximum 2 pages pour le plan de vérification. Le document servira à accompagner la validation, et sera à ce moment évalué selon la grille de la table 6.4.

5.4.3 Validation en laboratoire

Le but de cette activité, d'une durée de 4 min, est de

1. faire une démonstration en temps réel (*live*) de votre méthodologie pour la transformation de votre code assembleur SIMD en code machine. L'évaluateur doit pouvoir la reproduire lui-même s'il prenait le contrôle du poste de travail.
2. expliquer vos choix d'implémentation SIMD en vous basant sur la figure remise avec le devis de conception.
3. démontrer le bon fonctionnement des extensions par simulation fonctionnelle (Vivado) dans une organisation MIPS unicycle.

Pour la simulation dans Vivado, utilisez votre code Viterbi avec SIMD. Vous aurez à repérer dans votre chronogramme au moins 2 exécutions d'instructions SIMD structurellement distinctes au choix de l'évaluateur. Des exemples typiques sont le lwv, addv ou movenv. Il

est possible que votre implémentation SIMD ne fonctionne pas à 100% au moment de la présentation. Dans ce cas, utilisez un code assembleur intégrant vos tests unitaires, incluant l'instruction qui échoue. À ce stade, il est attendu au minimum que vous ayez en place les principaux éléments du chemin de données, ainsi que l'infrastructure de simulation permettant de les vérifier. Vous devez guider l'évaluateur à l'aide de votre plan de vérification pour indiquer le stade de développement où vous êtes rendus. Fiez-vous à la grille d'évaluation de la table 6.4.

6 ÉVALUATIONS

6.1 Évaluation du devoir

Rappel : Un code généré avec un compilateur se verra attribué la note 0.

	Activité	*devoir*	*devoir*	*devoir*
	Compétence	*GIF310_2*	*GIF310_2*	*GIF310_2*
	Qualité	*Connaissances en génie*	*Connaissances en génie*	*Connaissances en génie*
	Rubrique	*Assembleur*	*Assembleur*	*Assembleur*
	Critère	*Fonctionalité et validation*	*Convention MIPS*	*Clarté*
Niveaux	Pondération	20.00	15.00	5.00
Excellent	100.00%	Le calcul des survivants fonctionne sans erreurs, et est validé avec un cas de test pertinent.	Le code respecte toutes les conventions MIPS pour les appels de fonction, les registres et leur protection minimale sur la pile.	Le code est clair, lisible, bien compartimenté en fonctions et bien documenté.
Cible	85.00%	Le calcul des survivants fonctionne, mais est accompagné d'un cas de test trop simple ou mal documenté/commenté.	Le code respecte les conventions MIPS pour les appels de fonction, mais comporte quelques erreurs mineures concernant les registres et leur protection sur la pile.	Le code est relativement lisible et/ou partiellement documenté et assez bien compartimenté en fonctions.
Seuil	60.00%	Le calcul des survivants produit un résultat erroné, mais est accompagné d'un cas de test pertinent, ou le code fonctionne mais n'est pas accompagné d'un cas de test pertinent.	Le code respecte les conventions MIPS pour les appels de fonction, mais comporte plusieurs erreurs concernant les registres et leur protection sur la pile.	Le code est moyennement lisible, est éparpillé ou n'est pas documenté.
Non satisfaisant	25.00%	Le calcul des survivants produit un résultat erroné et n'est pas accompagné d'un cas de test pertinent ou, le code s'assemble mais cause une exception de mémoire ou une boucle infinie à l'exécution.	Le code contient des appels de fonction correctement gérés mais ne respecte aucune des conventions MIPS concernant les registres.	Le code est difficilement lisible, avec une documentation portant à confusion ou sans documentation.
Non initié	0.00%	Code non fourni ou qui ne s'assemble pas.	Il n'y a pas de structure d'appel et de retour de fonction dans le code fourni.	Le code est désorganisé.

Tableau 6.1 Grille d'évaluation du devoir

6.2 Évaluation du rapport

Le rapport fera l'objet d'une correction unique et la même note sera attribuée aux deux étudiants de l'équipe en fonction de la table 6.2. Cependant, la qualité de communication n'est pas découplée et peut influencer cette note. En effet, les phrases mal construites, les idées floues, les nombres sans unités ou les graphiques sans indications adéquates sont des éléments qui nuisent à la bonne compréhension des détails techniques et qui occasionnent des pénalités. C'est à vous de vous assurer de bien vous faire comprendre. L'idée la plus géniale n'a pas beaucoup de valeur si elle ne peut être communiquée clairement. La précision et la concision sont donc de mise. L'évaluation de la qualité de la communication sera faite selon la grille donnée à la table 6.3. L'atteinte du niveau *cible* dans cette grille implique que la note du rapport demeurera inchangée. Si ce seuil n'est pas atteint, jusqu'à un maximum de 15% des points peuvent être retranchés.

L'évaluation du rapport d'APP contribue à l'évaluation des éléments de compétence de l'unité. Il s'agit donc d'une évaluation sommative, c'est-à-dire que le résultat de l'évaluation sera consigné au dossier scolaire de l'étudiant et utilisé dans le calcul de sa note finale. Toutefois, pour permettre à l'étudiant d'apprendre de ses erreurs, son rapport corrigé lui sera remis, avec une grille lui permettant d'apprécier son niveau de compétence.

Il est important de noter que l'étudiant signataire du rapport est responsable de s'assurer de l'exactitude et de l'adéquation de chaque élément de solution, de même que de la qualité et de l'uniformité de l'ensemble du contenu de son rapport. N'oubliez pas que, lors de l'évaluation sommative (examen à la fin de l'unité), vous allez être évalués de façon individuelle sur les compétences mises en oeuvre pour l'élaboration de ce rapport. Vous êtes donc réputé pouvoir résoudre l'ensemble de la problématique de façon individuelle, de même que tout problème relié aux connaissances nouvelles à acquérir durant cette unité. Même si les travaux sont remis en équipe de 2 personnes, vous devez individuellement comprendre tous les aspects de la solution de la problématique pour pouvoir réussir les examens.

	Compétence	*GIF310_1*	*GIF310_3*	*GIF310_1*	*GIF310_2*
	Qualité	*Analyse de problèmes*	*Conception*	*Conception*	*Connaissances en génie*
	Rubrique	*Performance*	*Mémoires cache*	*Performance*	*Assembleur*
	Critère	*Analyser quantitativement les performances des architectures de processeur proposées*	*Proposer un système de mémoire cache à taille minimale avec impact maximal pour l'algorithme non-SIMD*	*Analyse conceptuelle d'un système optimisée pour un calcul de référence.*	*Développer un code assembleur exploitant des extensions SIMD*
Niveaux	Pondération	20,00	30,00	10,00	30,00
Excellent	100,00%	L'analyse de performance est clairement définie, quantifiée et expliquée pour chaque scénario. Toutes les configurations sont couvertes.	L'analyse du cas de référence et la solution proposée démontrent une pleine compréhension des mécanismes de mémoire cache. La solution est optimale.	Classifie et met rigoureusement en relation les configurations en fonction de leurs impacts cumulatifs sur la performance d'exécution du code de référence.	Implémente un algorithme fonctionnel sans et avec extensions SIMD qui respectent complètement les normes de codage et de présentation.
Cible	85,00%	L'analyse de performance est bien définie et quantifiée, mais son découpage ou les explications pourraient être un peu plus détaillés. Toutes les configurations sont couvertes.	L'analyse du cas de référence démontre une bonne compréhension de plusieurs mécanismes de mémoire cache, mais l'interprétation ne mène pas à la solution optimale, ou la solution optimale est proposée sans être entièrement justifiée.	Classifie et met correctement en relation les configurations en fonction de leurs performance d'exécution du code de référence. L'analyse sur les impacts cumulatifs pourrait être amélioré.	Implémente un algorithme fonctionnel sans et avec extensions SIMD , mais présentant des lacunes mineures au niveau du respect des spécifications ou des normes de codage et de présentation.
Seuil	60,00%	L'analyse de performance est définie et quantifiée, mais son découpage ou les explications pourraient être un peu plus détaillés. La performance de certaines configurations ne sont pas rapportées.	Peu importe que la solution soit optimale ou non, l'analyse du cas de référence aborde correctement des mécanismes de mémoire cache, mais leur mise en relation est superficielle ou justifie mal le choix de configuration de cache.	Classifie et met en relation les configurations, mais les explications manquent de détails ou encore les liens avec le code de référence laissent à désirer.	Implémente un algorithme qui suit partiellement les spécifications fonctionnelles ou respecte peu les normes de codage et de présentation.
Non satisfaisant	25,00%	L'analyse de performance est mal quantifiée, mal définie ou mal expliquée.	Peu importe que la solution soit optimale ou non, l'analyse du cas d'utilisation de la cache est générique et son explication démontre une compréhension superficielle des mécanismes de cache.	Classifie les configurations mais la mise en relation de leurs performances est vague ou n'a aucun lien avec le code de référence à optimiser.	Implémente un algorithme qui ne rencontre pas les spécifications fonctionnelles ou les normes de codage et de présentation.
Non initié	0,00%	Il n'y a pas d'analyse de performance.	La proposition est soit absente, soit complètement invalide.	Relève les différentes configurations matérielles mais n'arrive pas à les mettre en relation.	Algorithme non fourni ou invalide ou sans structure d'appel de fonction.

TABLEAU 6.2 Grille d'évaluation du rapport d'APP

Activité	*communication_rapport*	*communication_rapport*	*communication_rapport*
Qualité	*Communication écrite*	*Communication écrite*	*Communication écrite*
Rubrique	*Organisation*	*Présentation*	*Communication*
Critère	*Organiser et présenter de l'information pertinente*	*Présenter des communications graphiques et/ou un support visuel de qualité*	*Communiquer dans une langue de qualité, à l'écrit comme à l'oral*
Pondération	10,00	10,00	10,00
Excellent	Organise efficacement l'information pour en faciliter la compréhension. L'information est pertinente et complète au regard des objectifs de la communication. Le sujet est bien délimité et présenté avec concision	Présente des communications graphiques qui appuient le texte, sont faites selon l'état de l'art et sont accompagnées d'un titre évocateur. Recourt à un support visuel qui appuie efficacement ses propos et veille à ce que son support soit sobre, visible, équilibré et d'allure professionnelle	Fait des phrases complètes et bien structurées. Respecte les règles d'orthographe, de syntaxe et de grammaire élémentaires, en plus d'utiliser une terminologie et un vocabulaire tout à fait appropriés
Cible	Organise adéquatement l'information pour en permettre la compréhension. L'information est pertinente et complète au regard des objectifs de la communication	Présente des communications graphiques qui appuient le texte et qui sont généralement faites selon l'état de l'art. Recourt à un support visuel qui appuie bien ses propos et veille à ce que son support soit sobre, visible et équilibré	Fait généralement des phrases complètes et bien structurées. Respecte les règles d'orthographe, de syntaxe et de grammaire élémentaires, en plus d'utiliser une terminologie et un vocabulaire généralement appropriés
Seuil	Organise l'information pour en permettre la compréhension. L'information est pertinente au regard des objectifs de la communication, mais des éléments importants sont mal présentés	Présente des communications graphiques qui appuient le texte, mais celles-ci ne sont pas toujours faites selon l'état de l'art. Recourt à un support visuel qui appuie minimalement ses propos	Fait encore des erreurs d'orthographe, de syntaxe et de grammaire, mais celles-ci ne nuisent pas à la compréhension de son texte ou discours. Utilise une terminologie et un vocabulaire minimalement appropriés
Non satisfaisant	Organise minimalement l'information pour en permettre la compréhension. L'information est pertinente au regard des objectifs de la communication, mais quelques éléments importants sont absents	Présente des communications graphiques qui appuient minimalement le texte. Recourt à un support visuel dont l'allure devrait être améliorée	Fait généralement des phrases complètes et bien structurées, mais celles-ci comportent des erreurs d'orthographe, de syntaxe ou de grammaire qui nuisent parfois à la compréhension de son texte ou discours.
Non initié	N'est pas en mesure d'organiser l'information pour en permettre la compréhension. Éprouve de la difficulté à sélectionner l'information pertinente au regard des objectifs de la communication	Ne présente pas des communications graphiques qui appuient le texte. Recourt à un support visuel qui n'est pas adapté	Éprouve de la difficulté à faire des phrases complètes et bien structurées, de même qu'à respecter les règles grammaticales élémentaires, ce qui nuit à la compréhension de son texte ou discours. N'utilise pas une terminologie et un vocabulaire appropriés

Tableau 6.3 Grille d'évaluation de la communication écrite

6.3 Évaluation à la validation

	Activité	*Validation*	*Validation*	*Validation*	*Validation*
	Compétence	*GIF310_3*	*GIF310_2*	*GIF310_3*	*GIF310_2*
	Qualité	*Investigation*	*Connaissances en génie*	*Conception*	*Utilisation d'outils d'ingénierie*
	Rubrique	*Spécifications et vérification*	*Assembleur*	*Processeur*	*Assembleur*
	Critère	*Élaborer un plan de vérification pour des spécifications d'extensions SIMD données.*	*Transformation d'un code assembleur avec extensions SIMD en langage machine pour un microprocesseur modifié.*	*Spécifier et implémenter des extensions SIMD dans une organisation unicycle*	*Simulation fonctionnelle et vérification d'extensions SIMD dans l'outil Vivado.*
Niveaux	Pondération	20,00	10,00	30,00	10,00
Excellent	100,00%	En lien avec une spécification sommaire, le plan de vérification définit les objectifs, cible tous les tests unitaires pertinents avec critère de réussite ou d'échec clairs. La présentation résume au moins un test unitaire par instruction.	Maîtrise, exécute et explique intégralement la méthodologie de conversion des instructions.	Maîtrise l'ensemble des concepts liés à la modification d'une organisation de processeur et l'applique avec cohérence. Toutes les extensions SIMD sont spécifiées, clairement documentées (texte et figure) et fonctionnent correctement.	Simule les tests unitaires du plan de vérification dans l'environnement Vivado. La présentation suit de près le plan de test, affiche au moins un test unitaire par objectif et explique clairement les succès/échecs.
Cible	85,00%	En lien avec une spécification sommaire, le plan de vérification définit les objectifs et cible pour chaque objectif la plupart des tests unitaire pertinent avec critère de réussite ou d'échec. La présentation résume au moins un test unitaire par instruction.	Exécute la procédure de conversion des instructions mais a quelques difficultés à l'expliquer.	Maîtrise l'ensemble des concepts liés à la modification d'une organisation de processeur. Toutes les extensions SIMD sont spécifiées, documentées (texte et figure). Quelques erreurs subsistent dans leur implémentation, ou encore certaines explication manquent de détails.	Simule les tests unitaires du plan de vérification dans l'environnement Vivado. La présentation affiche au moins un test unitaire par instruction et explique les succès/échecs.
Seuil	60,00%	En lien avec une spécification sommaire, le plan de vérification définit les objectifs, au moins un test unitaire simpliste avec un critère de réussite ou d'échec. La présentation mentionne les tests unitaires sans les expliquer.	Connait la procédure de conversion des instructions mais a de la difficulté à l'exécuter, ou à expliquer des étapes cruciales.	Connais adéquatement les concepts liés à la modification d'une organisation de processeur. Toutes les extensions SIMD sont listées, mais leur spécification reste floue et des erreurs subsistent dans leur implémentation.	Simule certains tests unitaires du plan de vérification dans l'environnement Vivado. La présentation ne démontre pas au moins un test unitaire par instruction, ou montre des tests ne corresondants pas au plan de vérification.
Non satisfaisant	25,00%	En lien avec une spécification sommaire, le plan de vérification ne propose pas suffisamment d'objectifs, ou encore les critères de réussite ou d'échec sont vagues, insuffisants ou absents.	Connaît mal la procédure de conversion, les explications sont incomplètes ou l'exécution de l'encodage comporte des erreurs fondamentales.	Démontre une connaissance superficielle des concepts liés à la modification d'une organisation de processeur. Les extensions SIMD visées sont identifiées mais leur implémentation comporte des faiblesses apparentes.	Difficultés apparentes à simuler le projet ou à manipuler Vivado. La présentation est confuse sur l'interprétation des signaux.
Non initié	0,00%	Le plan de vérification est vague ou absent.	Ne connais pas la procédure de conversion.	Ne connais pas la démarche.	Pas de simulations présentées, ou aucun lien n'est possible entre les tests et le plan de vérification.

Tableau 6.4 Grille d'évaluation de la validation

6.4 Qualités de l'ingénieur

Voir le guide de référence pour les informations concernant les qualité de l'ingénieur.

Activité	*rapport*	*examen final*	*examen final*
Compétence	*GIF310_1*	*GIF310_2*	*GIF310_3*
Qualité	*Analyse de problèmes*	*Connaissances en génie*	*Analyse de problèmes*
Indicateur	*Analyser et interpréter les résultats obtenus*	*Connaissances procédurales (comment)*	*Définir le problème*
Rubrique	*Performance*	*Assembleur*	*Processeur*
Critère	*Analyser quantitativement les performances des architectures de processeur proposées*	*Comment construire un programme assembleur selon les normes MIPS*	*Analyser un chemin de données*
excellent	Est capable d'analyser correctement les résultats obtenus et témoigne d'une compréhension fine de leurs limites et de leur portée	Est capable d'appliquer en totalité ou presqueles méthodes et procédures nécessaires à la résolution d'un problème lié aux mathématiques,aux sciences naturelles ou aux sciences du génie	Identifie clairement et de façon exhaustive le problème d'ingénierie : reconnaît sa nature, identifie l'information connue et celle manquante, émet des hypothèses et énonce les résultats escomptés
cible	Est capable d'analyser correctement les résultats obtenus et d'en identifier les limites et la portée	Est capable d'appliquer l'essentiel des méthodes et procédures nécessaires à la résolution d'un problème lié aux mathématiques, aux sciences naturelles ou aux sciences du génie	Identifie clairement le problème d'ingénierie : reconnaît sa nature, identifie l'information connue et celle manquante
seuil	Est capable d'analyser sommairement les résultats obtenus et d'en identifier les limites et la portée	Est capable d'appliquer au niveau minimal acceptable les méthodes et procédures nécessaires à la résolution d'un problème lié aux mathématiques, aux sciences naturelles ou aux sciences du génie	Identifie convenablement, mais avec quelques lacunes mineures ou imprécisions, le problème d'ingénierie
insuffisant	Est capable d'analyser partiellement les résultats obtenus et d'en identifier les limites et la portée	Est incapable d'appliquer auniveauminimal acceptableles méthodes et procédures nécessaires à la résolution d'un problème lié aux mathématiques, aux sciences naturelles ou aux sciences du génie	Cerne suffisamment, mais avec plusieurs lacunes mineures ou imprécisions, le problème d'ingénierie pour en permettra la résolution
non_initie	Est incapable d'analyser les résultats obtenus et d'en identifier les limites et la portée	Est incapable d'appliquer les méthodes etprocédures nécessaires à la résolution d'un problème lié aux mathématiques, aux sciences naturelles ou aux sciences du génie	N'arrive pas à cerner le problème d'ingénierie pour permettre sa résolution

Tableau 6.5 Grille des qualités BCAPG mesurées

7 PRATIQUE PROCÉDURALE 1

Objectifs

- Connaitre les principales caractéristiques des processeurs RISC en général et de l'architecture MIPS en particulier.
- Se familiariser avec le langage assembleur :
  - Structure d'un programme.
  - Éléments de base : instructions, opérandes, indentation, commentaires, directives, étiquettes.
- Se familiariser avec le langage assembleur de l'architecture MIPS :
  - Instructions natives et pseudo-instructions.
  - Classes d'instructions.
- Analyser un programme en assembleur ou en langage machine.
- Concevoir un programme en assembleur.
- Exploiter une architecture au niveau du langage machine.
- Analyser une organisation unicycle simplifiée et la modifier pour étendre ses capacités.

P1.E1 (GIF310-2) Exploration du langage assembleur

Soit le programme C suivant qui calcule la somme des 15 premiers entiers positifs en stockant les résultats intermédiaires dans un tableau :

```
1  int32 A[16];
2
3  int main() {
4      int32 n, s;   /* n est l'index et s est la somme */
5      s=0;          /* initialisation de la somme */
6      A[0] = 0;
7      for (n=1; n < 16; n++) {
8          s=s+n;    /* ajout de l'index à la somme */
9          A[n]=s;   /* résultat intermédiaire stocké */
10     }
11 }
```

Voici un programme effectuant la même tâche, en langage assembleur MIPS :

```
1  .data 0x10010000
2  Ici:   .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
3
4  .text 0x400000
5
6  .globl main
7
8  main:
9      add $t0, $zero, $zero
10     la $t1, Ici
11     sw $zero, 0($t1)
12     addi $t2, $zero, 1
13     addi $t3, $zero, 16
14
15 boucle:
16     add $t0, $t0, $t2
17     addi $t1, $t1, 4
18     sw $t0, 0($t1)
19     addi $t2, $t2, 1
20     blt $t2, $t3, boucle
21
22     li $v0, 10                  # fin du programme
23     syscall
```


On cherche ici à établir le vocabulaire entourant le langage assembleur MIPS. En utilisant l'extrait de code ci-haut :

(a) Identifiez les segments de données et d'instructions de ce programme.

(b) Expliquez le rôle des éléments suivants : `.data`, `.word`, `.text`, `.globl`, `main:`, `boucle`, `Ici`

(c) À quelle adresse mémoire sont chargées les instructions de ce programme ? 0x400000

(d) Quels mécanismes peuvent être utilisés en assembleur pour représenter une variable ?
registre pour mettre des variables, .data pour créer des variable global

(e) Quels éléments du programme assembleur correspondent aux variables de la version en C (`n`, `s`, le tableau `A`, et une donnée indexée dans ce tableau) ?

(f) Expliquer le rôle et le fonctionnement de la commande `syscall`.

(g) Quelle est la dimension en octets de ce programme ?

P1.E2 (GIF310-2) Instructions et adressage

(a) Pouvez-vous identifier les pseudo-instructions dans le code assembleur de la question P1.E1 ?

(b) Que fait la pseudo-instruction `bleu $t0, $t1, etiquette` ? Comment un compilateur pourrait décomposer cette instruction en plusieurs instructions natives ?

(c) Il existe 3 formats d'instructions dans l'architecture MIPS pour ce qui est de la manière et l'ordre de déclinaison des opérandes de source et de destination.

(i) Donnez la spécification de ces 3 formats.

(ii) Déterminez le registre de destination dans les instructions suivantes :

(1) `lw $s1, (10)$s2`

(2) `add $t1, $t2, $t3`

(3) `addi $t2, $t1, 10`

P1.E3 (GIF310-2) Concevoir un programme en assembleur

Soit le code C suivant :

```
1  int valeurs[12]={4, 20, 6, 2, 1, 1, 10, 15, 8, 36, 5, 7};
2  int bornes[2]={2, 5};
3  int main() {
4      int n, s;       /* n est l'index et s est la somme */
5      s=0;            /* initialisation de la somme */
6      n=bornes[0];    /* initialisation de l'index */
7      while (n <= bornes[1]) {
8          s=s+valeurs[n]; /* incr. par la nième valeur */
9          n=n+1;      /* incrémentation de l'index */
10     }
11 }
```

(a) Quelle devrait être la valeur de `s` à la fin de l'exécution ?

(b) Proposez un programme assembleur équivalent.

(c) On veut maintenant transformer la fonction *int main()* en sous-fonction *int calcul(int valeurs[], int bornes[])*. Que faut-il modifier (normes mips simplifiée pour appel de fonction) ?

P1.E4 (GIF310-3) Langage machine MIPS

On observe en mémoire le segment de code machine suivant :

Adresse	Instruction
0x00400000	0x3c01ffff
0x00400004	0x342b000a
0x00400008	0x3c011001
0x0040000c	0x34240004
0x00400010	0x3c011001
0x00400014	0x00200821

— Interpréter les instructions en code machine MIPS pour en dériver le segment de code assembleur correspondant

P1.E5 (GIF310-3) Modification d'une organisation (prélaboratoire)

Soit le chemin de données illustré à la figure 7.1, ne supportant que des instructions MIPS de format R et sans mémoire de données.

1. Nous souhaitons modifier cette organisation pour *ajouter* la possibilité de l'addition avec une valeur immédiate sur 16-bits (`addi $rt, $rs, valeur`). Pour ce faire, commencez par analyser et comprendre les rôles joués par les modules déjà existants avec une instruction `add $rd, $rs, $rt` en commençant par le type d'encodage. Recommencez la procédure avec un `addi` en identifiant quelles parties peuvent être réutilisés, quelles parties sont manquantes, et ajoutez les modules pour réaliser les actions manquantes. Dessinez les changements directement sur la figure 7.1, cela vous sera utile pour le laboratoire 1. Note : En cas de difficulté, vous pouvez vous inspirer des figures de la section 4.4 du COD5, mais tentez quand même le travail avant de vous y référer.
2. À partir de votre réponse, répétez l'exercice avec `beq`. Vous verrez que vous pouvez réutiliser des pièces ajoutées pour le `addi`.

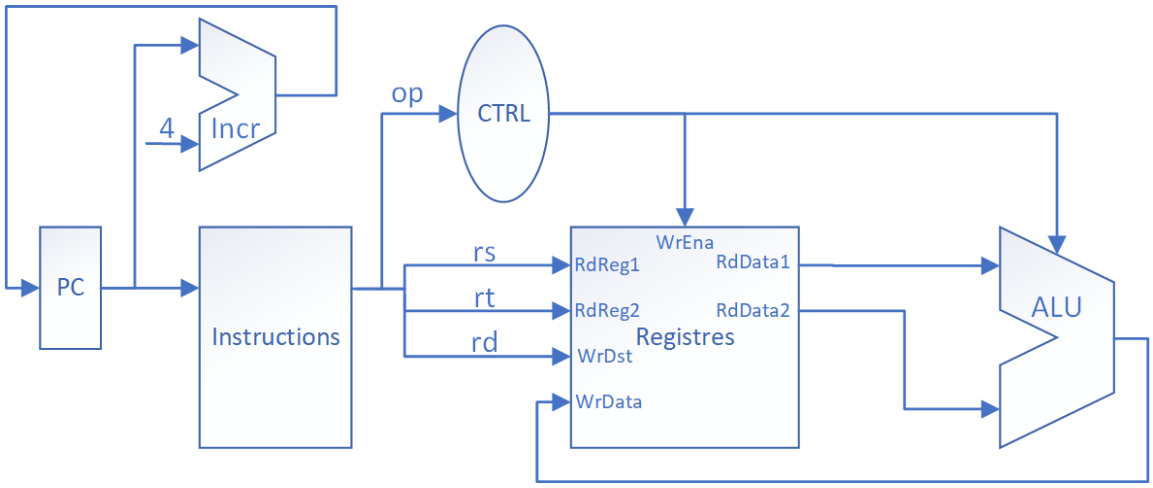


FIGURE 7.1 Architecture supportant seulement des opérateurs R (de registres)

8 PRATIQUE EN LABORATOIRE 1

Objectifs

— Premier contact avec une organisation unicycle avec mémoires d'instructions et données en SRAM.
— Modifier, développer et exécuter du code assembleur dans l'environnement MIPS / MARS.
— Modifier les chemins de données et de contrôle d'une organisation MIPS unicycle simplifiée.
— Élargir les capacités d'une unité arithmétique par la conception en VHDL et la simulation.
— Application de plans de validation.

8.1 (GIF310-2) Introduction au simulateur MARS (max 30 min)

Le simulateur MARS sera votre premier point de développement pour rédiger, tester et convertir vos codes en langage assembleur MIPS. Cet exercice vise à vous familiariser avec l'environnement et les procédures que vous aurez à répéter au cours de l'activité d'APP.

8.1.1 Exécution d'un code

1. Créer un répertoire vide qui servira d'espace de travail.
2. Télécharger le programme crypto_simple.asm du site web de l'APP vers ce répertoire de travail.
3. Démarrer le simulateur *MARS* (java), aussi disponible sur le site web.
4. Charger le programme crypto_simple.asm. Vous n'avez pas à comprendre les détails de ce code, il ne sert que d'exemple.
5. Cliquer sur la commande *Run / Assemble*.
6. Exécutez le programme pas à pas (touche *F7*) et suivez l'évolution des registres et de la mémoire dans l'interface *MARS*.
7. Notez dans le simulateur les données avant et après l'exécution du code. Avez-vous remarqué quels registres ont changé de valeur ? Quelles adresses mémoire on changé de valeur ?

8.1.2 Encodage du programme en assembleur

Les instructions exécutées par une architecture doivent être encodées en valeurs binaires (ou hexadécimales) avant d'être placées dans la mémoire d'instruction. Le simulateur *MARS* supporte cette fonction :

1. Dans le simulateur *MARS*, chargez le code `codeSimpleLabo1.asm` (voir site web).
2. Cliquez sur *Run / Assemble*, exécutez le code et vérifiez le fonctionnement.
3. Choisir dans le menu *File -> dump memory...*, conservez le choix du segment .text et changez le mode pour *Hexadecimal Text* (dans la problématique, pour exporter le segment de données, choisir plutôt le .data.)
4. Enregistrez avec le nom de votre choix (.txt). Comparez le résultat avec la colonne *Code* du simulateur *MARS*.

Nous allons utiliser cette procédure pour inclure les instructions dans vos simulations VHDL. Il faut cependant modifier les lignes afin de les rendre compatibles avec la syntaxe hexadécimale du VHDL. Pour être efficace, utilisez le "block mode" de l'éditeur Vivado, avec un également disponible dans notepad++ pour modifier plusieurs lignes en même temps :

1. Copiez-collez vos instructions hexadécimales dans un fichier texte quelconque dans Vivado (VHDL ou autre).
2. Activez le mode "colonnes" de l'éditeur texte (voir figure 8.1), puis cliquez-glissez devant le début des lignes désirées.
3. Un curseur apparaît maintenant devant chaque ligne. Tapez au clavier "`X"`" (X-guillemet).
4. Répétez la procédure mais à la fin de la ligne pour ajouter `",` (guillemet et virgule) à la fin de chaque ligne.
5. Chaque ligne devrait ressembler à –> `X"12345678",`

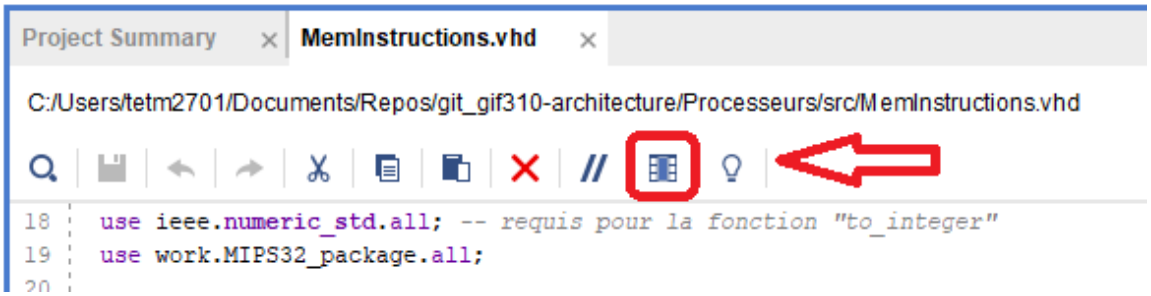


FIGURE 8.1 Mode d'édition de colonnes dans *Vivado*

8.2 (GIF310-3) Introduction à l'organisation unicycle

Informations pratiques

Fichier de package Lorsque des types de variables ou des valeurs constantes sont utilisées dans plusieurs fichiers VHDL, il devient très pratique de les déclarer dans un fichier de *package* afin de les définir qu'à un seul endroit dans tout le code, à la manière des fichiers `#include` en C/C++. Le fichier de *package* n'apparaît par contre pas dans l'arborescence de fichiers de projet dans *Vivado*. Il faut plutôt le retrouver dans l'onglet *library*, tel qu'illustré à la figure 8.2.

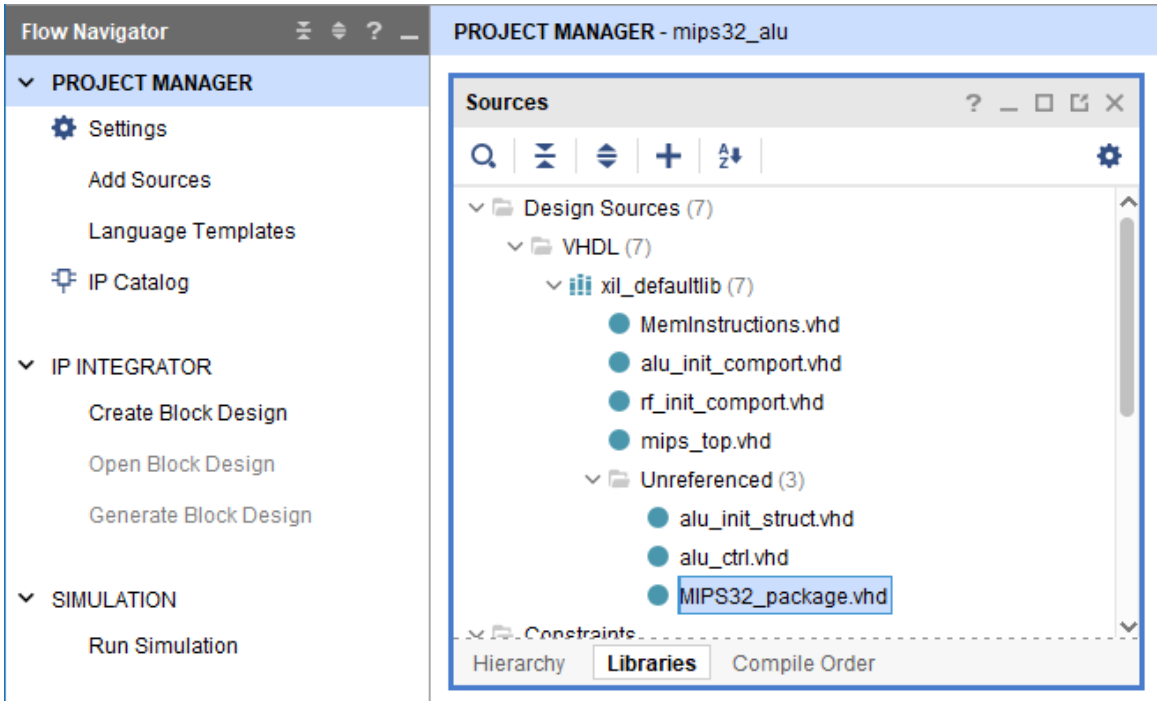


FIGURE 8.2 Localisation du fichier *package* dans *Vivado*

Figures du livre et VHDL La figure C.1 de l'annexe C présente la correspondance entre le nom des fichiers VHDL fournis et les principaux blocs d'une organisation unicycle. Le nom exact de certains fichiers auront des variantes au cours de l'APP, selon les particularités sous études (unicycle, pipeline, SIMD), mais la figure restera pertinente tout au long du cours.

Nomenclature VHDL Plusieurs pratiques de nomenclature sont répandues dans l'industrie pour nommer les signaux dans une architecture VHDL. À la base, les modules fournis dans l'APP utilisent une nomenclature d'étiquette ou noms axés sur des préfixes pour distinguer les types, noms d'état, noms de signaux ou de variables par rapport au fichier où ils se trouvent :

```
clk      -- signal d'horloge (une seule dans l'APP)
reset    -- reset synchrone
i_*      -- pour les entrées du module (entity/component)
o_*      -- pour les sorties du module (entity/component)
c_*      -- pour les constantes
r_*      -- pour les registres/bascules
s_*      -- pour les signaux combinatoires
v_*      -- pour les variables (bancs de test)
```

Bonnes pratiques d'organisation Pour éviter tout problème et pour maximiser les chances d'avoir du code synthétisable, observer les principes suivants :

— Évitez d'utiliser des `variables` et prendre uniquement le type `signal`, cela favorise un style de codage plus près du matériel. En simulation, les deux sont presque à comportement identique si on utilise judicieusement la commande "wait".

— Utilisez toujours l'entête suivant, sauf indication contraire :

```
library ieee;
use ieee.std_logic_1164.all; -- pour std_logic,
                             -- unsigned et signed
use ieee.numeric_std.all;    -- conversion to_integer()
use work.MIPS32_package.all; -- constantes du projet
```

— Pour tout signal de nature arithmétique, utilisez les types `signed` et `unsigned`. Vous pouvez également faire des *projections* (*cast* en anglais). Allez voir l'ALU pour d'autres cas de figure.

```
r_Registre1 <= unsigned(i_BusEntree);
o_BusSortie <= std_logic_vector(r_Registre1);
```

8.2.1 Simulation du processeur dans Vivado (30 min)

L'archive `mips_labo1_uniycle.zip` contient l'organisation MIPS 32-bits vue au procédural 1 (figure 7.1), une version très allégée de la figure COD5/4.17. Seules des instructions de *format* 'R' sont actuellement supportées par ce code VHDL, (voir les modules `controleur_labo1.vhd` et `alu.vhd`, sections avec un `case`). Notez aussi que les blocs "*Control*" et "*ALU control*" de la figure COD5/4.17 ont été fusionnées en un seul bloc, et le seront pour toutes les organisations VHDL fournies à cet APP. Pour cette étape du laboratoire :

1. Explorez et parcourez les fichiers sources en VHDL, en particulier le fichier `MIPS32_package.vhd`

2. Ouvrez le projet dans *Vivado* et simulez le projet sans modifications à partir du banc de test fourni.
   (a) De façon similaire à MARS, le banc de test arrêtera le simulateur si `$v0` contient la valeur 10 lors du `syscall`. Cependant, la fenêtre basculera automatiquement au point d'arrêt dans le VHDL. Simplement choisir à nouveau la fenêtre de simulation pour voir le chronogramme.
   (b) Le chronogramme n'a qu'une constante (période d'horloge) et 2 signaux (reset, horloge) d'affichés par défaut. Préparez une configuration de chronogramme (.wcfg) affichant des éléments pertinents dans les modules du processeur (PC, registres, ports de l'alu).
   (c) Un module de supervision (monitor) instancié dans le banc de test exploite des fonctionalités spécifiques aux bancs de tests pour reformatter certaines informations, par exemple en affichant sous forme de texte l'instruction et l'opération de l'alu. Il procure également un signal spécifique à l'appel des instructions `syscall` pour facilement les identifier, ainsi que l'appel de fin de programme. Ce module aura des ajouts adaptés aux différents laboratoires, à revisiter à chaque fois...
3. Vérifiez que le programme modifie les registres dans la même séquence et avec les mêmes résultats que vus dans *MARS* avec le code `codeSimpleLabo1.asm`.
4. Dans *MARS*, faites quelques modifications mineures au code `codeSimpleLabo1.asm` et régénérez le code machine en hexadécimal, comme à la section 8.1.2.
5. Dans le module `MemInstructions`, remplacez les instructions par celles que vous venez de régénérer. Laisser la dernière ligne automatiquement combler les adresses d'instruction inutilisée (ligne avec `others`).
6. Relancez la simulation et vérifiez que l'évolution des registres VHDL correspond à l'exécution dans *MARS*.
   — **Il se peut que la modification au bloc mémoire ne soit pas détectée par Vivado.** Si le comportement n'a pas changé suite à vos modifications, fermez la simulation, puis faire un clique-droit sur "simulation" et réinitialisez avant de relancer le processus.

8.2.2 Expansion d'une architecture de processeur, partie 1 (1h)

Comme vu au procédural, vous allez ajouter l'instruction `addi` comme première instruction de *format* immédiat (I) dans cette organisation. Le découpage du travail et un schéma conceptuel sont donc déjà disponibles dans vos cahiers de notes. L'extension de signe en VHDL peut se faire à même le module de chemin de donnée ou dans un sous-module, à votre choix. Vous pouvez aussi vous inspirer de la figure COD5/4.17 pour nommer et assigner les

signaux de contrôle. Vérifiez votre modification à l'aide du plan suggéré à la table 8.1 pour une plage dynamique des immédiats sur 16-bits. Pour des valeurs sur 32-bits, voir la section 8.3.1 ci-bas. **Astuce** : Utilisez l'outil *RTL Analysis* de *Vivado* pour générer automatiquement un schéma. Ceci devrait vous aider à comparer votre travail avec votre propre dessin fait en procédural 1.

Objectif ciblé		**Test du addi**	
Test	Action	Résultats attendus	
Instructions existante fonctionnelles	Exécuter code connu	Même qu'avec MARS	[ ]
Sommes simples avec immédiat 16-bits en complément-2	$zero + Immédiat immédiat = min(16-bits)	La somme donne min(16-bits)	[ ]
	$zero + Immédiat immédiat = max(16-bits)	La somme donne max(16-bits)	[ ]
	registre et immédiat positifs registre = ?; immédiat = ?	La somme donne ?	[ ]
	registre et immédiat négatifs registre = ?; immédiat = ?	La somme donne ?	[ ]
	registre positif, immédiat négatif registre = ?; immédiat = ?	La somme donne ?	[ ]
	registre négatif, immédiat positif registre = ?; immédiat = ?	La somme donne ?	[ ]

TABLEAU 8.1 Plan de vérification proposé pour le addi

8.2.3 Expansion d'une architecture de processeur, partie 2 (30 min)

Après l'addition de valeurs immédiate, vous allez ajouter une première instruction de contrôle de flux de programme, le `beq`. Comme pour le addi, il s'agit d'ajouter des composants de complexité moyenne (additionneur, muliplexeur) avec un peu de logique combinatoire ou de contrôle. Comparez votre plan de développement à la figure 4.11 du COD5 si nous êtes incertains face aux modifications à faire. Vérifiez votre implémentation VHDL à l'aide d'un code assembleur simple couvrant les deux cas de figures, soit la branche prise et la branche non-prise (plan de vérification de la table 8.2), par exemple avec une boucle d'incrémentation simple gérée par un `beq` en test de fin de boucle.

8.3 Modifications formatives complémentaires

Vous pourriez ajouter progressivement toutes les instructions MIPS à votre code, mais ceci n'est pas l'objectif du laboratoire et prends du temps. Cependant, il y a quelques autres modifications intéressantes à comprendre avant de voir le code de référence qui sera fourni.

Objectif ciblé		**Test du beq**	
Test	Action	Résultats attendus	
Instructions existante fonctionnelles	Exécuter code connu	Même qu'avec MARS	[ ]
La branche est prise lorsque les registres sont égaux	boucle infinie avec beq $zero, $zero, bcle	PC revient à l'adresse bcle	[ ]
La branche n'est pas prise lorsque les registres sont inégaux	Appels à Beq avec arguments toujours différents	PC incrémente de 4	[ ]

TABLEAU 8.2 Plan de vérification proposé pour le beq

Vous n'avez donc pas à faire ces modifications en VHDL, mais il est recommandé d'imaginer une solution comme au dernier numéro du procédural 1. Comparez ensuite votre solution au code de référence fourni pour le projet.

8.3.1 Chargement d'immédiat de 32-bits

Les instructions de type I en assembleur MIPS sont limitées à des valeurs de 16-bits. Malgré cela, MARS permet de spécifier une addition avec des valeurs de 32-bits. Comment fait-il pour légalement faire un addi avec des grandes valeurs ? MARS subdivise l'opération en deux intruction, souvent un lui et un ori.

Commencez par ajouter l'instruction `lui` (suggestion : utilisez l'alu en mode SLL16). Ajoutez ensuite l'instruction `ori`. Pour validez votre réflexion, appliquez le cas limite où on souhaite additionner 32769 ($2^{15} + 1$) à un registre quelconque. Comment le module d'extension de signe influencera ce résultat si laissé tel quel ? Que dit le guide sur l'assembleur ou la carte résumée MIPS à ce propos ? Consultez le code de référence une fois disponible pour voir si votre réponse y est similaire.

8.3.2 Fichier de registres

On vous a fourni une description VHDL préliminaire pour le banc de registres de l'architecture MIPS dans le fichier
`registres_labo1.vhd`, aussi représenté à la figure 8.3. Comment modifieriez-vous ce code VHDL pour que le registre `$zero` ne puisse jamais être modifié (toujours zéro) ? Repérez d'abord la modification à faire sur la figure 8.3 avant d'imaginer une modification au VHDL. Comment vérifieriez-vous cette modification ?

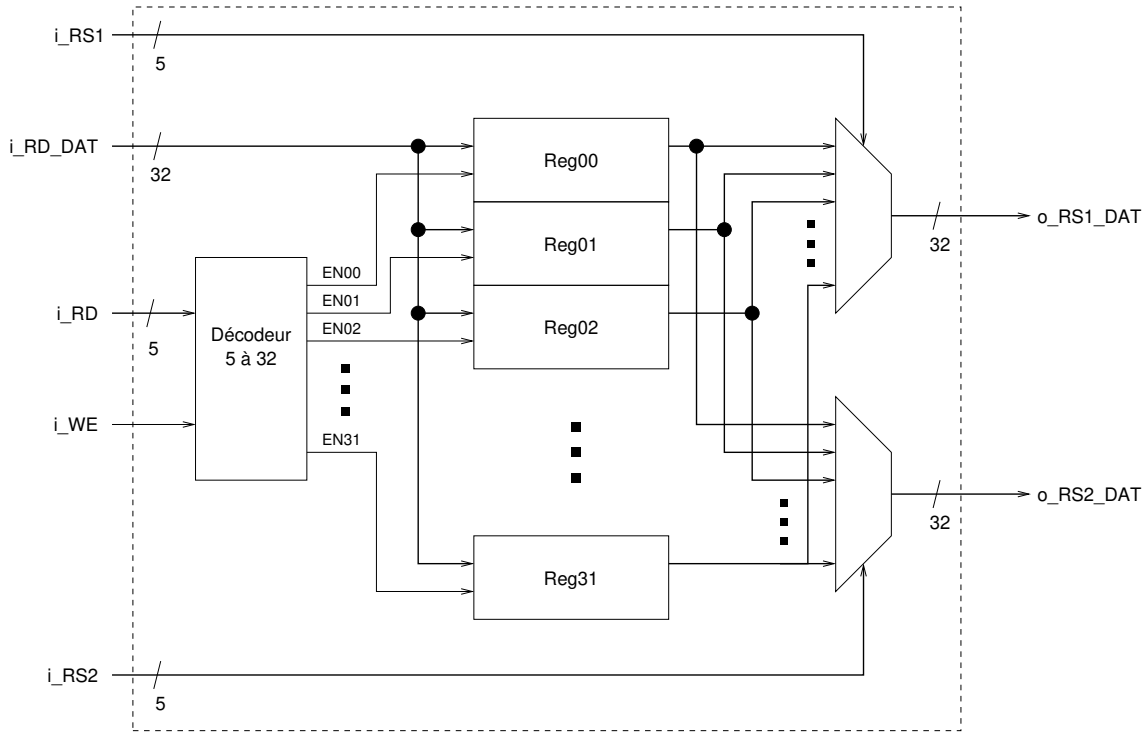


FIGURE 8.3 Structure interne du fichier de registre

9 PRATIQUE PROCÉDURALE 2

Objectifs

— Gérer le passage d'arguments par le biais de la pile
— Mettre en oeuvre des appels de fonction en langage machine
— Analyser l'organisation et le fonctionnement d'une mémoire cache en lecture
— Écrire un programme simple pour une architecture parallèle de type SIMD
— Implémenter une instruction originale dans une organisation unicycle.

9.1 EXERCICES

P2.E1 Appel de fonction

Soit la fonction suivante en C :

```
1 int moy8 (int x1, int x2, int x3, int x4, int x5, int x6,
     int x7, int x8)
2 {
3     return ( x1+x2+x3+x4+x5+x6+x7+x8 ) / 8;
4 }
```

a. Proposer un code assembleur équivalent qui suit la convention MIPS telle que décrite dans le guide [2] disponible sur le site de l'APP.
b. Quelle est la valeur contenue par le registre `$sp` avant le démarrage d'une fonction *main* ?

P2.E2 Calculs de performance et SIMD

Soit le segment de code C suivant et son équivalent en assembleur :

```
1 for (i=0; i < 200; i++)
2     for (j=0; j < 150; j++)
3         X[i][j]=Y[i][j]+100;
```

```
1  .data 0x10010000
2  X:   .space 120000                  # 150 x 200 x 4
3  Y:   .space 120000                  # (on assume 4 octets par
       donnée)
4
5  .text 0x400000
6
7  main:
8      la $t0, X              # adresse de base de X
9      la $t1, Y              # adresse de base de Y
10     li $t2, 200            # borne de fin de boucle externe
11     li $t3, 150            # borne de fin de boucle interne
12     add $t4, $0, $0        # indice de boucle i=0
13
14 L1: beq $t4, $t2, L4       # Si i=200, on saute à L4
15     add $t5, $0, $0        # indice de boucle j=0
16 L2: beq $t5, $t3, L3       # Si j=150, on saute à L3
17     lw $s0, 0($t1)         # On récupère Y[i][j]
18     addi $s0, $s0, 100     # On fait Y[i][j]+100
19     sw $s0, 0($t0)         # On entrepose dans X[i][j]
20     addi $t5, $t5, 1       # j=j+1
21     addi $t0, $t0, 4       # incrément. addr. X
22     addi $t1, $t1, 4       # incrément. addr. Y
23     j L2                   # retour au début de boucle interne
24 L3: addi $t4, $t4, 1       # i=i+1
25     j L1                   # retour au début de boucle externe
26
27 L4: li $v0, 10             # fin
28     syscall
```

(a) Quel est le temps d'exécution de ce code assembleur en cycles d'horloge ?

(b) Proposer des extensions SIMD au processeur MIPS en assumant que l'on souhaite travailler en blocs de 10 données en même temps. Modifiez le code assembleur pour exploiter ces nouvelle instructions.

(c) Quel est le facteur d'accélération observé en supposant un mémoire SRAM interne (sans pénalité d'accès) ?

P2.E3 Création d'une instruction originale - S4B

Soit le processeur unicycle représenté à la figure 4.17 COD5. On souhaite ajouter une nouvelle instruction "*soustraire quatre et branchement conditionnel si égal zéro*", documenté comme suit :

```
1 S4B $rt, incr # modifier PC de "incr" instructions si £rt
     actuel == zéro.
2                # Peu importe ce résultat, également faire £rt
                     = £rt - 4
```

Inspirez-vous de la procédure utilisée au numéro 5.5 (procédural 1), ou plus brièvement :

(a) Proposer un encodage en langage machine. Définir la valeur des champs au besoin.

(b) Détailler les opérations élémentaires et leur cheminement dans l'organisation.

(c) Quelles modifications faut-il apporter au chemin de données et, le cas échéant, de contrôle ?

9.2 EXERCICES SUPPLÉMENTAIRES

P2.S1 Analyse de code et temps d'exécution

Soit la fonction miroir32 ci-bas (également disponible au site web de l'APP) qui renverse l'ordre des bits et qui sert de code de référence pour le laboratoire 2. Quelle est la durée d'exécution (en nombre de cycles) de cette fonction sur un processeur à organisation unicycle ?

```
1  miroir32:                    # fonction renverser $a0 -> $v0
2                               # renverser l'ordre des bits
3                               # de $a0. (32 bits)
4                               # résultat dans $v0
5      add $t2, $a0, $0         # valeur a renverser dans $t2
6      add $v0, $0, $0          # resultat a construire dans $v0
7      addi $t1, $0, 32         # compteur = N-1
8      addi $t4, $0, -1         # cste -1
9      # li $t3, 0x80000000     # (pseudo instruction,
           changement manuel)
10     ori $t3, $zero, 0x8000
11     sll $t3, $t3, 16
12 bclerv32:                    # boucle k = 1 a N (k implicite)
13     beq $t1, $zero, finMirroir32
14     srl $v0, $v0, 1
15     and $t0, $t2, $t3        # masque du bit parvenu a
           position 2**N
16     or $v0, $v0, $t0         # place le bit dans le résultat
17     sll $t2, $t2, 1          # decalage gauche k position: 1,
           2,... N
18     addi $t1, $t1, -1        # compteur--
19     j bclerv32
20 finMirroir32:
21     jr $ra # fin miroir
```

Réponses (sans solution) à la section E.2.

10 PRATIQUE PROCÉDURALE 3

Mémoires cache

— Analyser l'organisation et le fonctionnement d'une mémoire cache en lecture

10.1 EXERCICES

P3.E1 Analyser une mémoire cache de type direct

On souhaite implémenter une architecture de mémoire cache pour prototypage dans un FPGA. Pour ce faire, il faut préparer un schéma-bloc à haut niveau (Entrées/sorties, multiplexeurs, comparateurs, bus, etc). Les contraintes imposées par le projet sont les suivantes :

— La mémoire est adressée en octets (1 LSB = incrément d'un octet).
— L'architecture du processeur définit les mots comme étant des données de 32 bits.
— Les blocs de la cache ont chacun 64 bits.
— La mémoire cache contient 128 octets de données (excluant les informations requises pour la gestion de la mémoire cache).
— La plage de mémoire système est définie sur 24 bits.

a. Dessiner un schéma grossier de cache directe. Plusieurs figures du chapitre 5 peuvent servir d'inspiration.
b. Basé sur la figure dessinée, décrire la nature des 3 types d'informations situées dans les mémoires cache directes.
c. Pour une cache à adressage direct comme décrite ci-haut, retrouvez les 4 catégories de subdivision du champ d'adresse lors d'un accès dans la cache.
d. Comment faire pour savoir si une donnée contenue à une adresse spécifique se trouve déjà dans la cache directe ?
e. En présumant que cette mémoire cache directe est initialement vide et branchée à une mémoire principale, identifier les réussites et échecs d'accès aux données demandées par la séquence d'adresses de lecture suivante :
   1. 0x101A55
   2. 0x7480D3
   3. 0x101A52
   4. 0x7480F3

P3.E2 Intégration des pénalités d'accès mémoire

On considère le code assembleur de la question P2.E2, sans SIMD, sur une organisation unicycle.

a. Calculez son temps d'exécution total si les données sont hébergées dans une mémoire DRAM avec une pénalité d'accès de 10 coups d'horloge pour chaque mot. Les instructions sont dans une SRAM sans pénalité d'accès. Réutiliser la réponse de P2.E2-a pour l'exécution des instructions.
b. Calculez son temps d'exécution total si les données sont hébergées dans une mémoire DRAM avec une pénalité d'accès de 10 coups d'horloge pour chaque mot, mais où la cache de données décrite ci-bas sert d'intermédiaire. Les instructions sont dans une SRAM sans pénalité d'accès.
c. Considérez maintenant que la DRAM fonctionne en mode "burst-8", où la pénalité s'applique à chaque demande de bloc de 8 mots consécutifs plutôt qu'à chaque mot. Les blocs demandés débutent toujours par addr(2 downto 0) == "000". Proposez deux configurations de cache directe permettant d'améliorer les performances du programme. Quel serait le temps d'exécution sur cette nouvelle configuration de mémoire ? Les instructions sont dans une SRAM sans pénalité d'accès.

Cache directe de 512k-octets, blocs de 1 mots, 32-bits par mots.

— L'architecture du processeur définit les mots comme étant des données de 32 bits.
— Les blocs de la cache ont chacun 32 bits, ou 1 mot.
— La mémoire cache contient 512k octets de données (excluant les informations requises pour la gestion de la mémoire cache).
— Le mode d'écriture est "write-through".

10.2 EXERCICES SUPPLÉMENTAIRES

P3.S1 Cache associative

— Refaire la question P3.E1-c) pour une cache associative
— Refaire la question P3.E1-e) pour une cache associative, après avoir pris soin de relire la section sur la méthode de remplacement du bloc le moins récemment utilisé (*least recently used*, ou LRU).

Réponses (sans solution) à la section E.1.

11 PRATIQUE EN LABORATOIRE 2

Performance des mémoires

— Premier contact pratique avec la mémoire DRAM et une mémoire cache, sans processeur.
— Configurer une cache à adressage direct.
— Visualiser les pénalités d'accès à la mémoire.
— Calculer ces pénalités d'accès avec DRAM et avec cache.
— Valider le calcul par simulation VHDL.
— Comprendre le rôle du tampon en mode écriture vers la mémoire principale (DRAM).
— Configurations possibles :
  — unicycle + DRAM-I + DRAM-D
  — unicycle + cache-I + DRAM-D
  — unicycle + cache-I + cache-D
  — unicycle + DRAM-I + cache-D (possible mais pas faite)

Activités à faire

— Ajuster le contrôle d'une organisation unicycle pour tenir compte des délais d'accès d'une mémoire DRAM.
— Évaluer l'impact sur la performance des mémoires de type DRAM dans une architecture de processeur.
— Configurer un modèle de cache direct et la vérifier dans un test unitaire.
— Tester et évaluer les performances de cette cache sur des courts programmes (test d'intégration).

Différentiation de fichiers de code

Pour comparer deux fichiers de code similaires, utilisez un logiciel de comparaison de fichiers. Dans les laboratoires, vous pouvez utiliser le logiciel *Meld* ou encore un des sous-modules dans les IDE de JetBrains (PyCharm, IntelliJ). Dans ce deuxième cas, placer les deux fichiers à comparer dans le même répertoire, ouvrir ce répertoire en tant que projet, sélectionner les deux fichiers concernés dans la vue projet, puis cliquer avec le bouton de droite de la souris et sélectionner *compare files*.

Sur vous ordinateurs personnels, vous pouvez utiliser une vaste gamme de logiciels dont Meld (dédié à cette fonctionnalité) ou encore un module de TortoiseSVN/TortoiseGIT. Vous devrez trouver comment faire pour le logiciel de votre choix, où Google est votre meilleur

ami.
https://meldmerge.org
https://tortoisesvn.net/downloads.html
https://tortoisegit.net/downloads.html
https://sourceforge.net/projects/npp-compare/

Projet VHDL pour le labo

Le projet *mips32_unicycle_reference* vous est fourni à l'APP comme point de départ pour la problématique. Le projet contient une description VHDL d'un processeur MIPS en organisation unicycle inspiré de la figure 4.24 du COD5 (page 271). L'organisation supporte la plupart des opérations arithmétiques (formats R et I), les accès à la mémoire de données (lw et sw), une instruction de type branchement conditionnel (BEQ) et des instructions de saut (J, JAL et JR). Le projet est préchargé avec le programme `test_mirroir.asm`, vu au numéro complémentaire P2.S1.

Le projet du laboratoire, `mips32_labo2_cache`, ajoute au chemin de données unicycle des éléments pour la réalisation de ce laboratoire mais est équivalent au projet de référence dans son état initial après décompression. Utilisez un outil de différentiation de fichiers et de *répertoires* (Meld ou IntelliJ) pour voir facilement et plus clairement les modifications comparativement au projet de référence. Le but de cette comparaison est d'établir vos repères.

11.1 Prélab : Simulation du processeur unicycle SRAM (5 min)

Le projet dans son état initial permet de simuler le code assembleur `test_mirroir.asm` directement à son ouverture dans Vivado. Pour comparer le temps d'exécution déterminé par analyse du code au numéro complémentaire P2.S1 vs une simulation :

- Récupérez votre formule de calcul et votre réponse au numéro P2.S1.
- Ouvrez le projet `mips32_labo2_cache` dans Vivado.
- La configuration par défaut définit une organisation identique au unicycle de référence, similaire à celle illustrée à la figure 4.24 (COD5). Au niveau des mémoires, elle utilise des mémoires SRAM pour les instructions et les données, tel qu'illustrée à la figure 11.1 (la mémoire de données y est omise par simplification).
- Le projet vient préchargé avec le programme `TestMirroir.asm`, disponible sur la page web de l'APP. Simulez son exécution dans Vivado.
- Ajoutez ici également les signaux du module de veille (*inst_Monitor*), différent du laboratoire 1.
- Vérifiez l'exécution des 10 premières instructions après le *reset* en comparant avec MARS (mode pas-à-pas). Notez l'évolution du compteur de programme (r_PC), les

mots/valeurs des instructions qui défilent (voir *inst_Monitor*) et les changements de valeur du banc de registres durant cette période. Suggestion : prenez une capture d'écran de cette plage de temps.

— Simulez le programme de test jusqu'à son appel à syscall, indiqué par le signal flag_syscall du module *inst_moniteur* du banc de test.

— Notez la durée de la simulation jusqu'au syscall et comparez avec votre réponse au numéro P2.S1.

— Fermez la simulation (enregistrez la configuration des traces pour sauver du temps plus tard).

Jusqu'ici, les simulations du projet unicycle de référence et du laboratoire sont identiques.

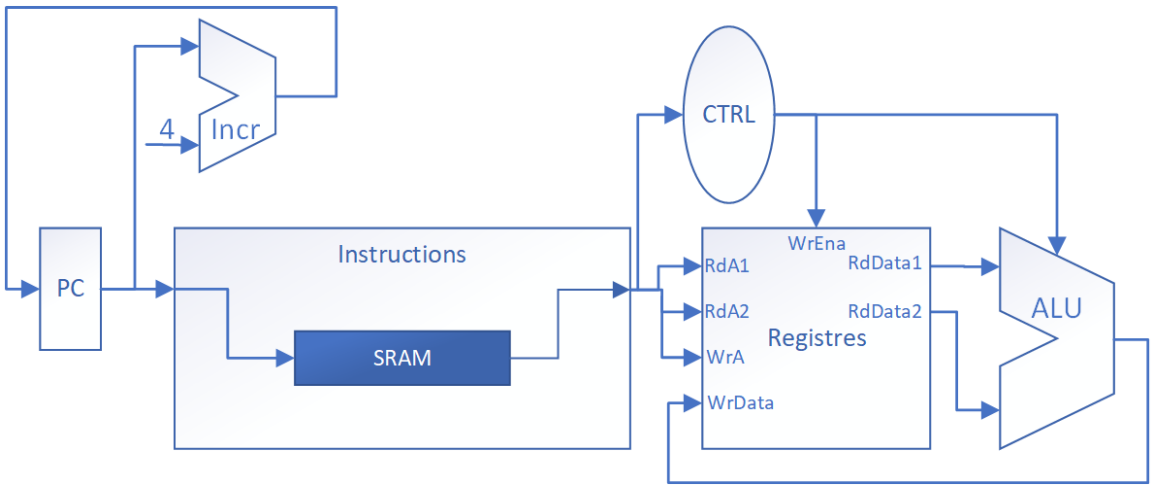


FIGURE 11.1 Schéma d'organisation unicycle simplifiée de la figure 4.17 (COD5), avec une mémoire SRAM pour les instructions. Les nouvelles valeurs sont immédiatement disponibles après un changement d'adresse, comme un circuit combinatoire.

11.2 Simulation du processeur unicycle avec DRAM (25 min)

Contrairement aux mémoires SRAM à lecture quasi instantanée, les mémoires DRAM exigent des séquences particulière pour accéder aux données qui y sont entreposées. Ces séquences sont normalement gérées par un contrôleur (MEF) de complexité variable (figure 11.2), aujourd'hui souvent fourni dans une partie préconçue et paramétrable des FPGA. Le séquençage DRAM implique par contre des délais dans l'obtention ou le remplacement d'une valeur dans la mémoire DRAM, délais qui varie selon l'agilité du contrôleur. Nous allons ici utiliser un contrôleur simplifié et déterministe où le délai est de 10 coups d'horloge après la requête pour les accès en lecture (figure 11.3). Le contrôleur permet aussi de demander des lectures consécutives sans avoir à remettre le signal de requête à '0' (figure 11.4), où la nouvelle valeur

ne sera valide qu'au prochain *Read Acknow* suivant. Enfin, vous allez seulement utiliser ce contrôleur pour les instructions alors il n'implémente pas les écritures des données.

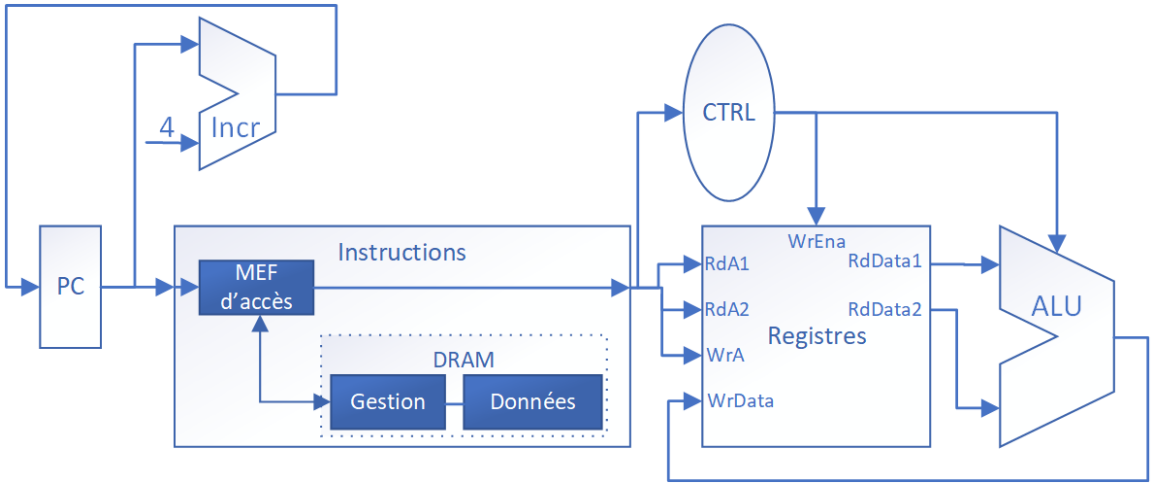


FIGURE 11.2 Organisation unicycle simplifiée de la figure 4.17 (COD5), avec mémoire DRAM pour les instructions. Il faut une MEF à la fois dans le processeur et la DRAM pour gérer l'échange des données.

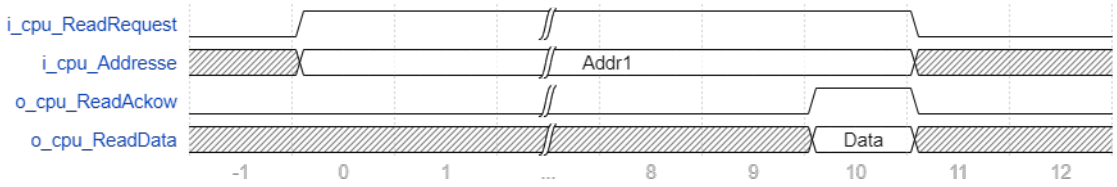


FIGURE 11.3 Chronogramme du modèle de contrôleur DRAM pour un seul accès en lecture.

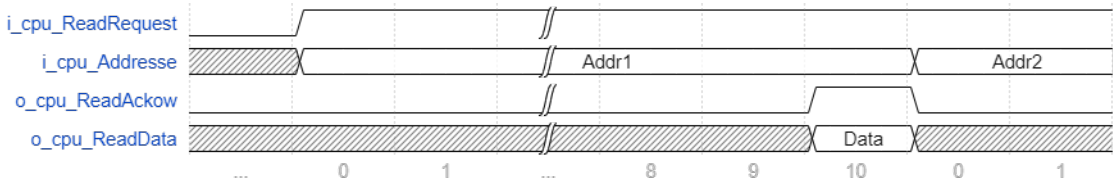


FIGURE 11.4 Chronogramme du modèle de contrôleur DRAM pour des accès consécutifs en lecture.

Poursuivez maintenant le laboratoire à partir du projet ouvert dans Vivado :

— À la déclaration de l'entité du chemin de données (mips_datapath_unicycle) à la ligne 24-26, changez le paramètre générique "g_useInstructionMem" à false, puis "g_useInstructionDram" à true. L'organisation est changée pour correspondre à la figure 11.2.

— Le contrôleur DRAM (MEF) fait maintenant le pont entre le processeur et la mémoire d'instructions et synchronize la réception des données en fonction du délai de lecture.
— Ajoutez au chronogramme les signaux affichés à la figure 11.3 et relancez la simulation.
— Comparez maintenant l'évolution du compteur de programme (r_PC) et du banc de registres dans les 100 premières nanosecondes.

Que s'est-il passé ? En fait, le compteur de programme (r_PC) n'est pas au courant du délai requis par la mémoire DRAM et continue d'incrémenter tel que décrit dans le VHDL. Il faut alors paralyser le processeur en attendant l'arrivée du nouveau code d'instruction. Il y a trois éléments à figer en attendant la réponse de la mémoire pour ne pas causer d'erreurs d'exécution. r_PC en est un, quels sont les autres modules/composants qui ne doivent pas changer d'état ou de valeur ? Suivez le signal s_InstructionStall dans le fichier mips_datapath_unicycle_labo2.vhd pour des indices. Suggestion : **Dessinez vos réponses sur la figure 4.17 du COD5 pour vous aider à visualiser ces modifications**.

Une fois vos modifications implémentées, le processeur peut s'exécuter correctement en attendant entre chaque accès mémoire :

— Relancez la simulation, validez que l'exécution du programme est redevenu équivalent à MARS et notez le nouveau temps d'exécution requis (plus de 20 μs).
— Ajustez votre formule du numéro P2.S1 avec la pénalité de la DRAM en vous inspirant du tout début de la section 5.4 de COD5, puis recalculez le nombre de coups d'horloge.
— Est-ce que votre nouveau calcul et le temps simulé correspondent ? Votre simulation devrait arriver au même résultat plus la durée du reset d'amorce.

11.3 Mise en oeuvre d'une cache (2h)

11.3.1 Introduction

Pour mitiger la perte de performance induite par la mémoire DRAM et son contrôleur, nous allons intégrer une cache pour la mémoire d'instructions (figure 11.5). Un contrôleur en cours de rédaction est fourni dans le projet de ce laboratoire et doit être complété. Les blocs implémentant la détermination des succès/échec et l'accès à l'espace mémoire de la cache seront pratiquement identiques à la représentation dans la figure 5.12 de COD5, **mais avec un nombre de blocs et de mots par blocs différents**.

Ce sera ici une cache directe de 16 blocs, avec 2 mots de 32-bits par blocs, avec adresses sur 32-bits. En cas de succès d'une requête en lecture, la bonne valeur sera immédiatement présentée au processeur avec uniquement des circuits combinatoires. En cas d'échec de lecture, une MEF amorcera alors une séquence de chargement de bloc de données en provenance de la DRAM suivant le chronogramme montré aux figures 11.6 et 11.7 ajoutant une pénalité de 10

coups d'horloge par échec d'accès. Les accès en écriture seront discutés plus loin à la section 11.4.

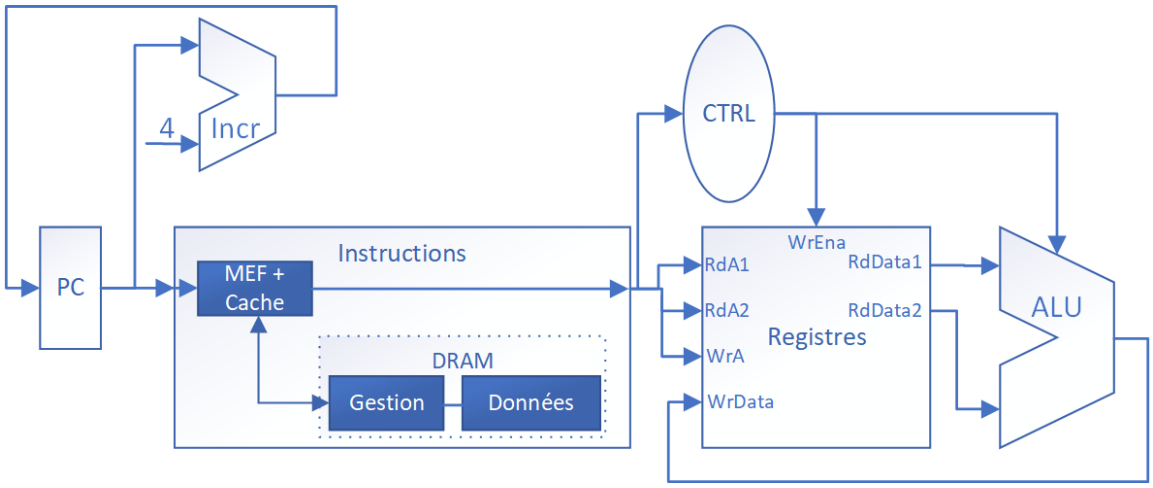


FIGURE 11.5 Organisation unicycle simplifiée de la figure 4.17 (COD5), avec une mémoire cache. La cache agit aussi à titre de MEF pour les accès à la DRAM.

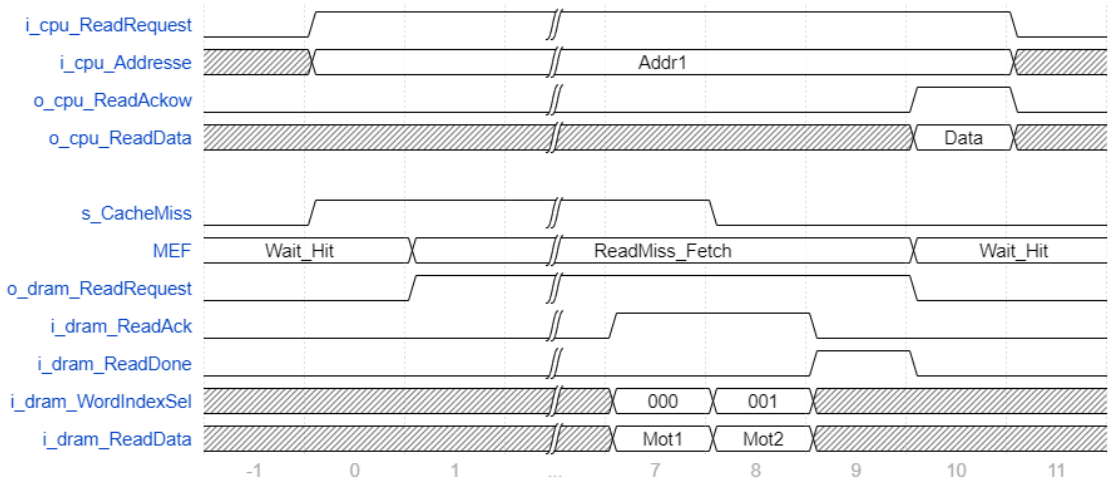


FIGURE 11.6 Chronogramme du contrôleur de cache accédant à la DRAM après un échec en lecture. Cet exemple montre le cas où il y a une pause avant et après l'accès à la mémoire.

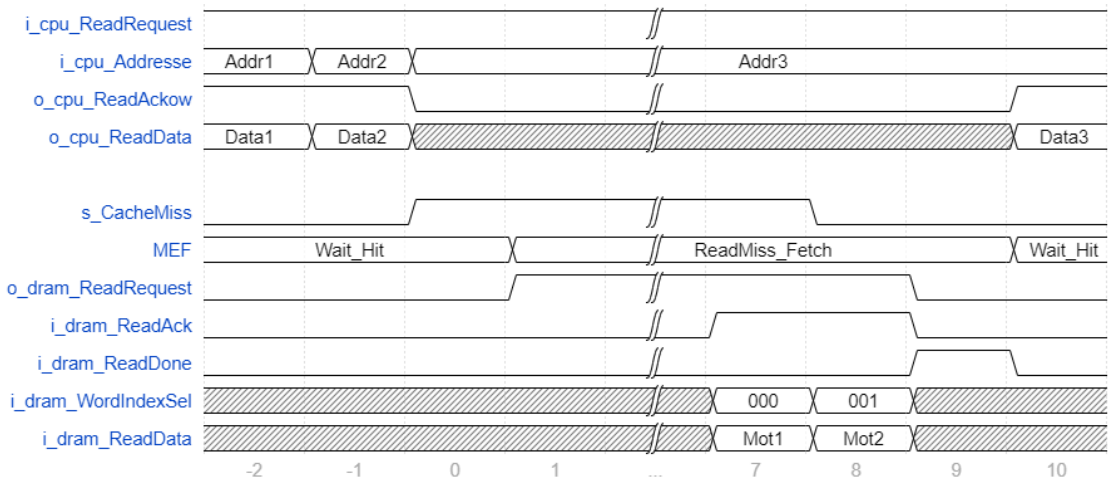


FIGURE 11.7 Chronogramme du contrôleur de cache avec requêtes en continu, avec 2 succès suivis d'un échec.

11.3.2 Plan de vérification

Avant de plonger dans les modifications du code, passez en revue la proposition de plan de vérification visant le fonctionnement de votre configuration de mémoire cache d'instructions à adressage direct (lecture seulement). Ce point de vue alternatif pourrait vous aider à recadrer les fonctionalités recherchées.

	Objectif ciblé		**Lecture avec cache (8 blocs de 2 mots)**	
ID	Test	Action	Résultats attendus	
L1	Lecture	Lire une adresse correspondant à un bloc vide.	La MEF déclare un échec. La MEF garni le bloc avec les données de la mémoire DRAM. Pénalité de 10 coups d'horloge	[ ]
L2		Lire une adresse liée à un bloc garni, mais différent de la plage mémoire enregistrée dans le bloc.	La MEF déclare un échec. La MEF garni le bloc avec les données de la mémoire DRAM. Pénalité de 10 coups d'horloge	[ ]
L3		Lire une adresse accessible en cache.	La MEF déclare une réussite. La MEF ne fait rien.	[ ]
P1	Paramètres : tous les blocs sont accessibles et utilisables.	Lire 34 mots à des adresses consécutives, puis relire les 31e et 1er mots.	Taux d'échec de %50 aux 34 premières lectures, réussite à la relecture du 31e mot, échec pour le premier mot.	[ ]

TABLEAU 11.1 Plan de vérification proposé pour la cache à adressage direct (lecture seulement)

11.3.3 Configuration et simulation de la cache en lecture

À cette étape du laboratoire, vous allez vous concentrer sur la configuration de la cache et sa vérification en mode lecture seule (cache d'instructions). La partie écriture viendra plus tard.

— Dans Vivado, ouvrez le fichier CachePourDram_labo2.vhd.
— Repérez les commentaires commençant par "- - ???". Ils indiquent tous des lignes à compléter ou modifier. Révisez les notions du procédural 2 pour trouver les bonnes valeurs à inscrire.
— Dans Vivado, ouvrez le banc de test ciblant spécifiquement le contrôleur de cache (cache_testbench.vhd). Il contient déjà des séquences correspondants aux éléments du plan de vérification de la table 11.1.
— Pour lancer en simulation ce banc de test, ne pas oublier de faire dans Vivado l'opération "set as top" dans la section simulation de l'onglet "design" (voir figure 11.8).
— Utilisez le code VHDL de test fourni dans le banc de test pour vérifier la cache. Vous pouvez également modifier ces séquences.
— Passez à l'étape suivante une fois que tous les tests passent avec succès.

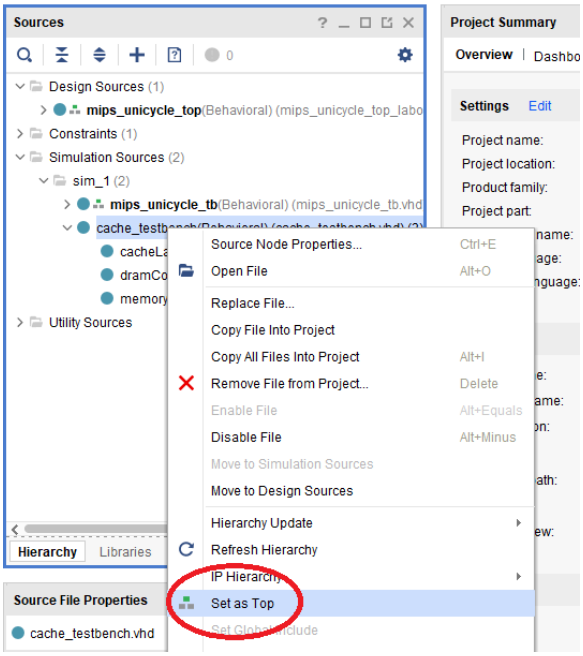


FIGURE 11.8 Changer le banc de test à exécuter à l'aide de la fonction "Set as Top".

11.3.4 Test d'intégration

Avec la cache validée seule en mode lecture, vous allez simuler la cache avec le processeur unicycle et évaluer le gain de performance par rapport à la DRAM sans cache :

— Réassignez le banc de test principal comme module de simulation (*set as top*, figure 11.8).
— Activez le mode avec mémoire cache dans l'organisation unicycle : à la déclaration de l'entité mips_datapath_unicycle (au début du fichier), changez le paramètre générique "g_useInstructionDram" à false, puis "g_useInstructionCache" à *true*. Votre contrôleur de cache fait maintenant le pont entre le processeur et la mémoire d'instructions.
— Simulez le processeur accéléré par la cache d'instructions.
— Est-ce que le programme termine avec les bons résultats ?
— Quel est le nouveau temps d'exécution du programme de test ? Les résultats attendus sont à la section E.3, mais un rien pourrait vous faire diverger de la réponse.
— Comment prédire analytiquement la durée ? Il faut ici faire une analyse manuelle des échecs et succès à partir du code assembleur, en n'oubliant pas que les instructions sont récupérées 2 à 2.
    — L'usage d'une feuille Excel est fortement recommandée pour prédire les performances par section de code.

11.4 Vérification en mode écriture (30 min)

Maintenant que vous avez complété et vérifié votre contrôleur de cache en lecture (cache d'instructions), il faut poursuivre sa vérification en considérant également l'écriture des données. Dans ce cas, le contrôleur du laboratoire 2 écrit la donnée dans une mémoire tampon (mode *write buffer*). Puis, si le bloc est déjà dans la cache, la donnée est également écrite localement au bon endroit. Dans le cas contraire, aucune autre action n'est faite ou requise. La situation à surveiller vient qu'en cas d'échec en *lecture*, si le tampon n'est pas vide, il doit être vidé avant la lecture et le remplacement du nouveau bloc de cache afin d'assurer la cohérance des données (*cache coherancy*). La pénalité sera donc de 10 cycles d'horloge pour chaque donnée toujours en attente d'écriture, en plus de la pénalité de lecture pour remplacer le bloc. Consultez la fin de la section 5.3 du COD5 pour une brève discussion sur l'écriture en mode *write buffer* et sur les autres modes possibles (*write through*, *write back*).

La vérification de l'écriture des données en simulation complexifie le travail d'analyse en raison du tampon d'écriture. Par exemple il faut aussi vérifier (et donc provoquer) le vidange du tampon en cas d'échec en lecture et d'autres cas limite. Le plan de vérification à la table 11.2 propose de nouvelles conditions et nouveaux tests à ceux déjà réalisés. Il n'est pas nécessaire d'implémenter tous ces tests pendant ce laboratoire, en particulier ceux déjà réalisés à la table 11.1 (L1-L3, lorsque le tampon d'écriture est vide).

Pour ajouter les tests en mode écriture :

— Choisir à nouveau comme *Top* le banc de tests ciblant spécifiquement le contrôleur de cache (cache_testbench.vhd, comme à la figure 11.8).
— Lisez bien les commentaires suivant les séquences en lecture sur comment encoder une écriture ou une lecture.
— Développez les séquences d'écriture/lecture pour les tests E2, L5 et E3.
— Les autres tests sont facultatifs, mais utile à réfléchir comment les faire. Par exemple, pour E1, il faut partir d'un bloc vide.

Si la cache est bien configurée, le code en écriture fonctionne correctement.

TABLEAU 11.2 : Plan de vérification proposé pour la cache à adressage direct (lecture et écriture)

	Objectif ciblé		**Lecture avec cache (8 blocs de 2 mots)**	
ID	Test	Action	Résultats attendus	
E1	Écriture	Écrire un mot dans un bloc vide.	La MEF ne déclare pas d'échec. Aucun changement à la cache. La cache écrit éventuellement la donnée en DRAM.	[ ]
E2		Écrire un mot dans un bloc occupé par une étiquette différente de la destination d'écriture.	La MEF ne déclare pas d'échec. Aucun changement à la cache. La cache écrit éventuellement la donnée en DRAM (tampon).	[ ]
E3		Écrire un mot dans un bloc de la cache où l'ancienne valeur existe.	La MEF ne déclare pas d'échec. La valeur en cache est mise à jour immédiatement. La cache écrit éventuellement la donnée en DRAM (tampon).	[ ]

E4		Écrire N fois de suite à la même adresse.	Les valeurs sont écrites à la même adresse de la DRAM, espacés par la pénalité d'accès DRAM. Seulement la dernière valeur reste en mémoire DRAM. La présence ou non de cette adresse en cache n'a pas d'importance.	[ ]
L1 L2 L3	Lectures **Condition** : Le tampon d'écriture est vide.	Idem à la table 11.1	Idem à la table 11.1.	[ ]
L4	Lectures **Condition** : Le tampon d'écriture n'est pas vide.	Lire une adresse correspondant à un bloc vide.	La MEF déclare un échec. La MEF attends que le tampon d'écriture se vide (pénalité variable). La MEF garni le bloc avec les données de la mémoire DRAM (pénalité de 10 coups d'horloges)	[ ]
L5		Lire une adresse liée à un bloc garni, mais différent de la plage mémoire enregistrée dans le bloc.	La MEF déclare un échec. La MEF attends que le tampon d'écriture se vide (pénalité variable). La MEF garni le bloc avec les données de la mémoire DRAM (pénalité de 10 coups d'horloges)	[ ]
L6		Lire une adresse accessible en cache.	La MEF déclare une réussite. La MEF ne fait rien.	[ ]

S1	Surcharge du tampon d'écriture (plus de 15 valeurs en attente)	Provoquer un débordement **Remarque** : La mise en pause du processeur n'est pas implémentée.	Ajout de pauses permettant au tampon de descendre à 15 valeurs ou moins.	[ ]

12 PRATIQUE PROCÉDURALE 4

Objectifs

— Analyser le fonctionnement d'un pipeline au sein d'un processeur.
— Identifier et classer les aléas de l'organisation en pipeline.
— Calculer la performance d'un code assembleur.
— Déterminer la performance moyenne d'une architecture en pipeline.

12.1 EXERCICES

P4.E1 Chemin de données, aléas et dépendances dans un pipeline

Étant donné le code de la fonction `miroir32` donnée en prélaboratoire du procédural 2 (P2.S1), considérer que la portion après l'étiquette `bclerv32` s'exécute dans un pipeline MIPS à 5 étages sans unité d'avancement ou de vidange, et avec les branchements au 4e étage (voir [1, fig. 4.46, p. 301]). Dans la figure 12.1 de la page suivante, suivez l'évolution des instructions dans le pipeline à partir de la ligne 14 (`srl`).

a) Déterminez les dépendances de données qui peuvent causer des aléas.
b) Dans la figure COD5-4.46, où se trouve le circuit logique responsable de la décision de branchement ? Combien d'instructions sont déjà en cours d'exécution ?

P4.E2 Performance d'une organisation à pipeline

On considère un processeur MIPS doté d'un pipeline à 5 étages. Chaque étage du chemin de données est caractérisé par les délais combinatoires suivants :

IF	ID	EX	MEM	WB
220 ps	200 ps	335 ps	250 ps	200 ps

(a) Quelle devrait être la fréquence d'horloge maximale pour
  (i) un processeur unicycle ?
  (ii) un processeur avec pipeline à 5 étages ?
(b) Déterminer et comparer le temps d'exécution d'une instruction arithmétique pour les deux types de processeurs (pas d'aléas de données pour le pipeline).
(c) Déterminer et comparer le temps d'exécution d'une séquence de 10 instructions arithmétique pour les deux types de processeurs (jamais d'aléas de données pour le pipeline).
(d) S'il était possible de diviser en deux parties égales le délai combinatoire d'un des étages, lequel choisiriez-vous et quelle serait la nouvelle période d'horloge du processeur ?

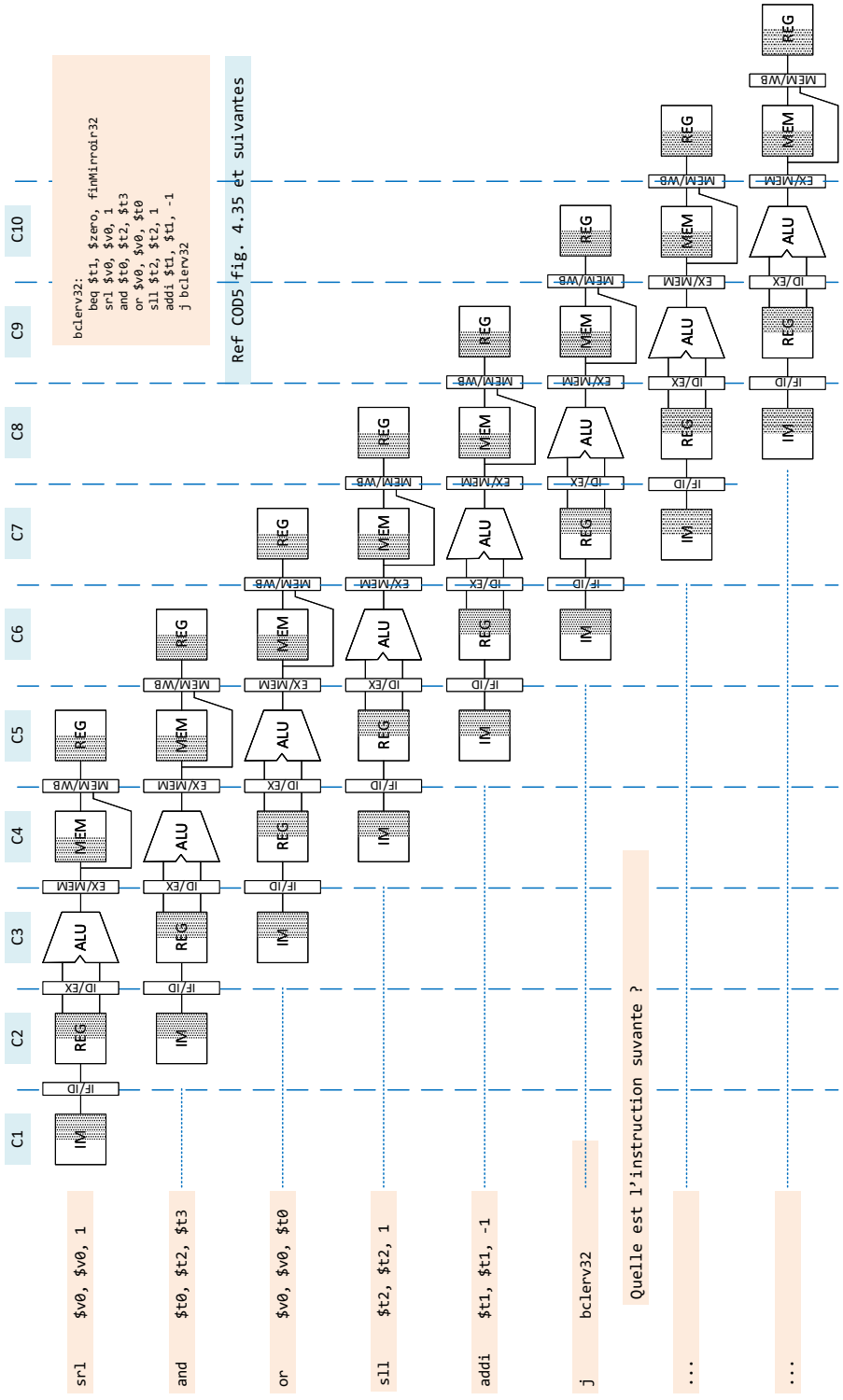


FIGURE 12.1 Évolution des instructions dans le pipeline avec le temps

P4.E3 Performance d'une organisation à pipeline avec aléas

Soit le segment de code suivant :

```
1          addiu $t9, $t9, 4
2          lw $t1, 8($a0)
3          and $t2, $t1, $t3
4          beq $t2, $s2, etiquette # on présume l'égalité
               fausse ici.
5          addu $t1, $t2, $s2
6          or $s2, $t2, $t1
7          j fin
8  fnct2:  add $v0, $a0, $a1
9          jr $ra
10 fin:    li $v0, 10
11         syscall
```

(a) Quelles sont les situations causant des aléas dans ce code, si on considère une organisation en pipeline sans gestion des aléas et les branchements faits à l'étage MEM (comme à la figure 4.51, p.304). Indiquer en commentaire dans ce code le type d'aléa, le registre impliqué et son écart avec l'instruction en conflit.

(b) Pour cette même organisation, résoudre les situations en introduisant le bon nombre de `nop` avant ou après les aléas identifiés pour garantir une exécution valide en tout temps. Calculez le temps d'exécution en nombre de cycles d'horloge incluant l'appel à syscall et le vidange du pipeline.

(c) Sans modifier l'organisation, que peut-on faire pour réduire le temps d'exécution par un cycle ?

(d) Refaire le même exercice qu'en (b) en assumant cette fois qu'il y a une unité d'envoi telle qu'à la figure 4.57, p. 312 dans COD5. Calculez le temps d'exécution en nombre de cycles d'horloge incluant un appel à syscall (pas illustré).

(e) Considérez maintenant que les branchements sont faits au 2e étage plutôt qu'au 4e, comme à la figure 4.65, p.325 de COD5. Identifiez dans la figure le bloc qui décide (ou non) du branchement conditionnel, vérifier ses interconnexions et corrigez les nop en conséquence.

(f) Jusqu'à présent, vous avez ajouté à la main les nops dans le code, mais ceci ralentira aussi son exécution dans une organisation unicycle alors qu'ils n'ont pas besoin de ces délais additionnels. Comme schématisé dans le bloc "Hazard detection unit" à la figure 4.65 de COD5, les nops sont normalement gérés automatiquement par le matériel sous forme de bulle pour insérer des retards (aléa de données), ou sous forme

de vidange pour écraser des instructions qu'il ne faut pas exécuter (aléa de contrôle). Considérant cette insertion intelligente de bulles et vidanges, réannotez la solution de (e) et calculez le nombre de cycles requis par le programme pour l'organisation à la figure 4.65 avec une **prédiction de branches non-prises**. Incluez ici aussi l'appel à syscall (pas illustré) et le fin d'exécution du programme.

(g) Si on considère que l'extraction de la première instruction (`addiu`) a lieu au cycle 1, déterminer l'état du pipeline au cycle 9.

(h) (À faire soi-même, réponses en E.4) Si la période d'horloge est de 350 ps sans unité d'avancement (UA) (question b, figure 4.51, p.304), de 420 ps avec unité d'avancement mais gestion manuelle des bulles/vidanges (question d et e), et enfin de 420 ps avec gestion matérielle des bulles et vidanges et l'UA (question f, figure 4.65, p.325), déterminer le temps d'exécution du segment de code dans les quatre cas, jusqu'à la fin de l'exécution du programme.

P4.E4 Calcul de CPI moyen - systèmes multicycles

On considère une organisation MIPS ultra-compacte, mais qui utilise un nombre variable de cycles d'horloge pour exécuter différentes catégories d'instructions. Les instructions arithmétiques prennent 4 coups d'horloge, les sauts et branchements 3 coups d'horloges, les accès à la mémoire 5 coups d'horloge et les autres instructions 8 coups d'horloge.

Math	Branch	Jump	`lw`	`sw`	autres
50%	14%	12%	10%	4%	10%

a) Sachant que les programmes exécutés sur cette organisation suivent une distribution d'instructions suivant le tableau ci-bas, quelle est en moyenne le nombre de cycles par instructions (CPI) ?

b) Supposant un programme de 6M (million) d'instructions et une période d'horloge de 1.7 GHz, quelle est la durée d'exécution du programme ?

c) Quel est le CPI d'une organisation unicycle ?

P4.E5 Calcul de CPI moyen - pipeline et cache

On considère une organisation MIPS avec pipeline à 5 étages, avec détection des aléas, unité d'envoi (*forwarding unit*) et prédiction de branchement non-pris telle qu'illustrée dans [1, fig. 4.65, p. 325]. Les programmes qui y sont exécutés suivent une distribution des instructions identique à la question précédente.

a) Supposez que les mémoires d'instructions et de données sont logées dans une SRAM locale, donc sans pénalité d'accès. Proposez une formule générale pour le calcul du

CPI moyen des organisations en pipeline. Autrement dit, d'où vient la dégradation de la performance ?

b) Calculez le CPI moyen de cette organisation pipeline, où aucun aléas de données ne génère de bulle (excellent compilateur) et où 17% des branches sont non prises.

c) Calculez maintenant le CPI moyen pour un *système* utilisant cette dernière organisation pipeline mais avec les mémoires d'instructions et de données dans des caches, branchées avec une DRAM commune, le tout avec les particularités suivantes :
   — Taux d'échec d'instructions de 20%.
   — Taux d'échec de lecture de données de 8%.
   — Taux d'échec d'écriture de données de 5%.
   — La DRAM a une pénalité d'accès de 12 cycles.
   — Écriture gérée par un tampon d'écriture (*write-buffer*, comme le laboratoire).
   — Le tampon ne sature jamais.
   — Le temps de vidange du tampon est négligeable.

d) (À faire soi-même, réponse à E.5) Répéter c, mais cette fois avec une cache de données avec écriture en mode *write-through* (**toujours** écrire à la fois dans la mémoire et la cache pour chaque `sw`).

12.2 EXERCICES SUPPLÉMENTAIRES

P4.S1 Analyse d'un programme sur une architecture à pipeline

Soit le code MIPS suivant (Réponses numériques du numéro à la section E.6.) :

```
1      lui $s1, 0x1001       # adresse tableau x[] (type math)
2      add $t3, $0, $0       # s = 0
3      addi $t6, $0, 0x80    # c: compteur initialisé
4      addi $t2, $0, 4       # k: compteur initialisé
5
6  L1:
7      lw $t1, 0($s1)
8      add $t3, $t3, $t1
9      addi $t2, $t2, -1
10     beq $t2, $0, L2
11     sw $0, 0x200 ($s1)
12     j L3
13
14 L2:
15     sw $t3, 0x200 ($s1)
16     add $t3, $0, $0
17     addi $t2, $0, 4
18
19 L3:
20     addi $s1, $s1, 4
21     addi $t6, $t6, -1
22     bne $t6, $0, L1
23
24 fin:
25     addi $v0, $zero, 10   #
26     syscall               # (type autres)
```

(a) Analyser ce segment de code et déterminer ce que fait le programme.

(b) Déterminer le nombre de cycles nécessaires pour exécuter ce segment de code dans une organisation MIPS unicycle telle qu'illustrée dans [1, fig. 4.24, p. 271].

(c) Déterminer le nombre de cycles nécessaires pour exécuter ce segment de code dans une organisation MIPS multicycles suivant la catégorisation des instructions indiquées

à la table ci-bas.

Math	Branch	Jump	`lw`	`sw`	autres
4	3	2	5	4	2

(d) Déterminer le nombre de cycles supplémentaires requis pour exécuter ce segment de code dans une organisation MIPS avec pipeline à 5 étages, avec gestion matérielle des aléas et unité d'envoi (*forwarding unit*) telles qu'illustrées dans [1, fig. 4.65, p. 325].

(e) Pour l'organisation à pipeline, si on considère que l'extraction de la première instruction (`lui`) a lieu au cycle 1, déterminer l'état du pipeline au cycle 21.

13 PRATIQUE EN LABORATOIRE 3

Pipeline et unité d'avancement

— Analyser le comportement d'un code MIPS dans un simulateur de pipeline.
— Comparer les organisations unicycle et pipeline fournies.
— Ajouter une unité d'avancement pour l'ALU.
— Accélérer le pipeline en devançant l'action des branches et jump au 2e étage.
— Développer et exécuter des plans de vérification pour les modifications proposées.

13.1 Préparation au laboratoire

Rappel de l'organisation unicycle

Une organisation unicycle MIPS complète est mise à votre disposition sur le site de l'APP. Elle est inspirée (mais non identique) à la figure 4.24 du COD5. L'organisation fournie a été vérifiée avec le code assembleur *TestMirroir.asm*. Le chemin de données supporte un sous-groupe essentiel d'instructions (calculs, LW, SW, BEQ, J, JAL, JR). Elle vous servira pour la problématique, ainsi que de point de référence pour ce laboratoire.

Retour sur le banc de registres : théorie vs pratique

Dans le chapitre 4 de [1] et dans la figure 12.1 (1er numéro du procédural 3), on stipule que la lecture des registres se produit au front d'horloge descendant (moitié droite en gris) alors que l'écriture se produit au front montant (moitié gauche en gris). Ceci permet de sauver un coup d'horloge dans le pipeline lors d'aléas de données. Cependant, le banc de registre fourni pour l'organisation unicycle de référence encode la lecture comme un simple multiplexeur asynchrone et ne fonctionnera pas avec l'organisation pipeline. Le banc de registres fourni pour ce laboratoire est donc légèrement différent et change les valeurs de sorties du banc de registre en milieu de cycle. Cela change ainsi l'allure des chronogrammes par une apparence de décalage d'une demi-période d'horloge, mais le code fourni fonctionne correctement.

13.1.1 Ajouts au chemin de données

Commencez par comparer le chemin de données des organisations unicycle et pipeline à l'aide d'un utilitaire de différence (par exemple, le programme Meld). Repérez les registres de pipeline et l'endroit où ils remplacent les signaux combinatoires. Êtes-vous en mesure de lier ces changements avec les figures 4.17 et 4.51 du COD5 ?

13.2 Ajout d'une unité d'avancement

Une organisation MIPS en pipeline basée sur l'organisation unicycle complète vous est fournie, très similaire à la figure 4.51 de COD5. En fait, cette organisation est un découpage de l'unicycle où on a enclavé les sections entre des colonnes de bascules D (plus explicitement,

les colonnes en bleu dans la figure 4.35 de COD5). L'organisation de la figure 4.51 implémente exactement le même jeu d'instructions que l'organisation unicycle complète, mais n'offre pas de très bonnes performances à cause des aléas de données et de contrôle. Vous devez modifier cette organisation en pipeline pour gérer les aléas en cours d'exécution et pouvoir bénéficier des gains potentiels promis par ce type d'organisation.

13.2.1 Analyse initiale

La mémoire d'instructions est préchargée avec un code sans nop. Observez les problèmes d'exécutions, puis remplacez les instructions par la version équivalente avec des nop (disponible sur le site web) qui corrigent les aléas de données et de contrôle. Validez que le résultat dans MARS pour les codes sans et avec nop sont identiques, ainsi que le résultat dans Vivado en utilisant le code incluant les nop. Notez le temps d'exécution jusqu'au syscall en microsecondes (assez long).

13.2.2 Implémentation

Comme vous avez pu voir dans le code *TestMirroir_pipeline.asm*, il faut manuellement ajouter des nop pour éviter les aléas de données et de contrôle. Ceci veut dire qu'un code fonctionnant correctement dans *MARS* ou dans le processeur unicycle causera des erreurs dues à des aléas de données et de contrôle dans l'organisation pipeline fournie. Votre prochaine tâche consiste à ajouter une unité d'avancement pour les aléas de données. **La conception est entièrement détaillée à la section 4.7 du COD5.**

Utilisez une approche de conception complète avant de vous lancer dans le VHDL (définition des entrées/sorties, liste des pièces présentes et celles à ajouter). Faites-vous aussi un plan de vérification pour l'unité d'avancement, ce qui vous aidera également à faire l'analyse de la logique combinatoire avant la rédaction du VHDL. Pour les tests, écrivez votre propre code assembleur avec les aléas de données causées par des instructions de format R. Si vous incluez des instructions de branchements, prenez soin d'ajouter les nop requis puiqu'il n'y a pas de module de gestion des bulles et vidanges.

13.2.3 Gain de performance

Vérifiez que l'exécution du code et son résultat sont conformes à Mars, puis notez le nouveau temps d'exécution dans Vivado. Quel est le gain de performance ?

13.3 Partie optionnelle : Réduction des délais de branchements

L'organisation en pipeline fournie effectue les *jump* et les branchements (`j`, `beq`) à l'étage *MEM* (4e étage) avec une stratégie de prédiction de branche non prise, mais il est possible de devancer ces actions vers l'étage *ID*. Ceci réduit significativement l'insertion des délais de branchement et augmente d'autant la vitesse d'exécution des boucles while et for. En vous

inspirant des explications aux pages 318-320 de COD5, déplacez la décision de branchements à l'étage *ID*, mais il n'est pas requis de gérer l'insertion automatique des `nop` devant les `beq` avec aléa de données (faites ceci à la main au besoin). Validez votre changement avec un code assembleur utilisant un seul `nop` après les branches et jump. Testez aussi le cas d'un `lw` suivi d'un `beq` utilisant le registre mis à jour par ce `lw`. Combien faut-il de nop dans ce cas après avoir mis `beq` au 2e étage ?

Vérifiez que l'exécution du code *TestMirroir* modifié en conséquence reste conformes à Mars, puis notez le nouveau temps d'exécution dans Vivado. Quel est le gain de performance ?

A Complémentaire : Altorithme de Viterbi - concept

A.1 Introduction

L'algorithme de Viterbi a été découvert à plusieurs reprises dans différents domaines. Andrew Viterbi, cependant, l'a développé en 1967 dans le contexte des communications numériques pour le décodage de codes convolutifs [7]. Lesdits codes avaient été proposés en 1955 par Elias et si on disposait déjà de méthodes pour les décoder (décodage séquentiel, décodage à seuil), il a fallu attendre l'algorithme de Viterbi pour disposer d'une méthode de décodage optimale au sens de la séquence transmise la plus probable (*maximum likelihood*).

D'une manière générale, c'est un algorithme qui permet, à partir d'observations, de déduire la séquence d'états par laquelle passe une machine à états cachée. Connaissant la structure de la machine à états en question ainsi que l'état de départ et en observant uniquement ses sorties, l'algorithme peut déduire la séquence d'états la plus probable par laquelle passe la machine, et ce même si les observations sont perturbées par du bruit ou incomplètes.

A.2 Encodeur convolutif

Dans le contexte des communications numériques, la machine à états cachée consiste en un encodeur qui prend les bits à transmettre et les entre un à un, à mesure qu'ils arrivent, dans une ligne de délais à K positions. Au lieu de transmettre les bits du message, on transmet des bits codés obtenus en faisant des ou-exclusifs à partir des bits à différentes positions de la ligne de délai. Comme la génération des bits codés se fait à partir d'un ensemble de K bits du message, ceci introduit des corrélations entre les bits et on comprend intuitivement qu'il devient alors possible, étant donné ces corrélations, de détecter et de corriger un bit quelconque en erreur. La figure A.1 illustre un encodeur convolutif de longueur $K = 3$ et de taux $r = \frac{1}{3}$, avec un message u en entrée et sa version encodée en sortie p sur trois bits. Le taux est simplement défini comme le ratio du nombre de bits k entrant à chaque étape au nombre de bits sortant correspondant n, c.-à-d.

$$r = \frac{k}{n}. \tag{A.1}$$

Habituellement, on entre seulement 1 bit à la fois dans la chaîne de délai, ce qui fait que $k = 1$. Pour l'encodeur de la figure A.1, le taux illustre le fait que 3 bits sont transmis pour chaque bit utile, ce qui fait que le potentiel de correction d'erreur s'accompagne d'un coût significatif en largeur de bande. Toutefois, il est possible de concevoir des codes puissants dont le taux r est très proche de 1.

Les éléments d'addition dans la figure sont en fait des additionneurs modulo 2 et peuvent être réalisés tout simplement en employant des portes ou exclusif. L'indice m indique que c'est le $m$ième bit du message et on assume que les bascules opèrent à la fréquence des bits entrant, avec trois bits sortants par coups d'horloge. On peut aussi envisager d'avoir un nouveau bit entrant à tous les 3 coups d'horloges pour sérialiser les bits sortants. Dans ce cas, le circuit disposerait d'une horloge plus rapide en plus d'un signal "strobe" en enable qui lui est à la fréquence des bits. On note que les bits originaux du message ne se trouvent pas dans les

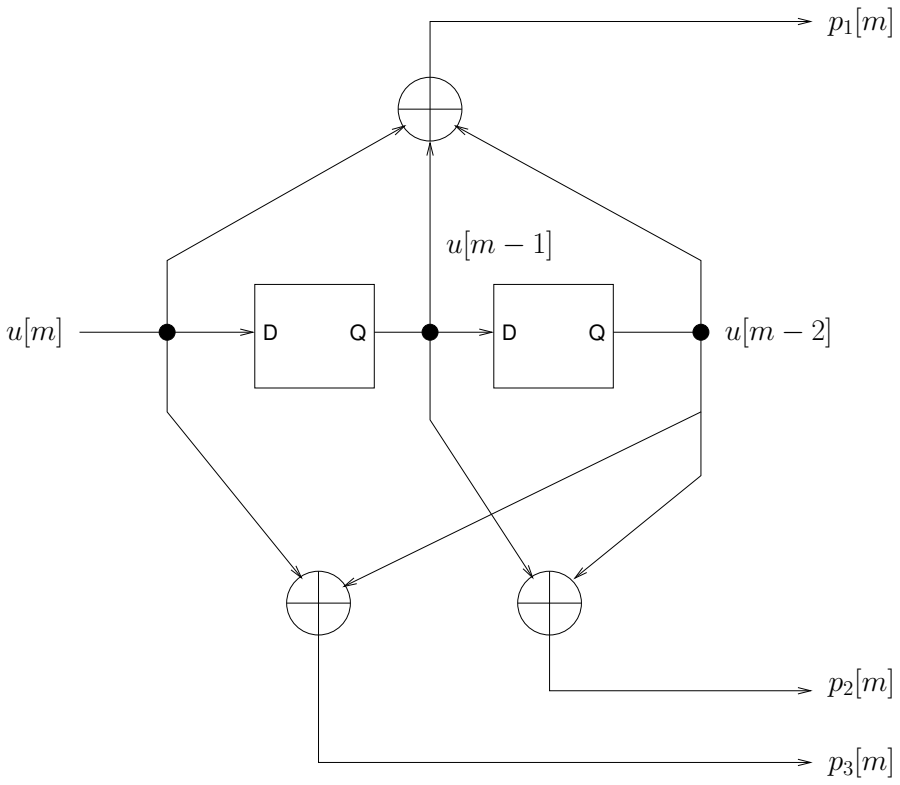


FIGURE A.1 Encodeur convolutif avec longueur de contrainte $K = 3$ et de taux $r = \frac{1}{3}$.

bits transmis. Les 3 bits sortants sont transmis soit en séquence (donc à un taux 3 fois plus rapide que le taux des bits en entrée), soit en parallèle si le schème de transmission le permet (modulation multiniveaux, porteuses distinctes, etc.).

Comme l'encodeur n'a de mémoire que des 2 derniers bits, il s'agit d'une machine à 4 états. La structure de l'automate épouse en fait le paradigme de Mealy. Le graphe correspondant est illustré à la figure A.2, où u est l'entrée, la sortie suit la convention $p = p_1p_2p_3$, et l'encodage de l'état suit exactement l'état des bascules.

Clairement, comme il y a un seul bit entrant à chaque étape, il n'y a que deux transitions sortantes pour chaque état. Si la machine démarre à l'état 00 au début d'une trame, on peut représenter les transitions d'états possibles par un *diagramme en treillis* tel qu'illustré à la figure A.3. Une flèche pleine correspond à une transition résultant d'un bit du message à '0', tandis qu'une flèche pointillée correspond à la transition résultat d'un bit à '1'. À la fin de la trame, on peut ajouter deux zéros successifs au message pour ramener la machine à l'état 00. On verra que le décodage selon l'algorithme de Viterbi correspond à trouver le chemin le plus court dans un tel treillis de l'état 00 au début de la trame à l'état 00 à la fin de la trame. Ceci implique cependant la notion de *distance* ou de *coût* associée à chaque branche.

A.3 L'algorithme de Viterbi comme décodeur optimal

Il faut donc déterminer une métrique pour chaque branche qui est inversement proportionnelle à la probabilité que la machine est bien passée par cette transition d'états à ce moment

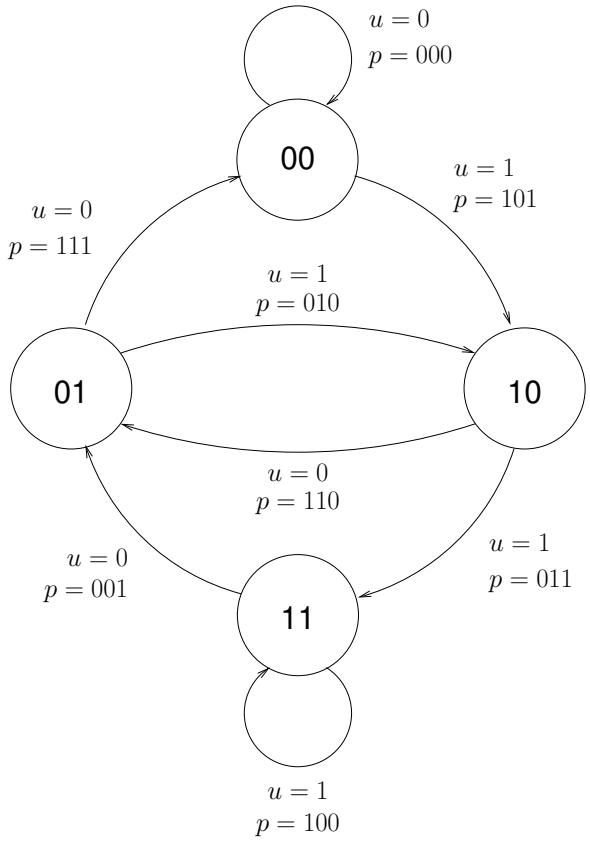


FIGURE A.2 Graphe d'états de l'encodeur convolutif de la figure A.1.

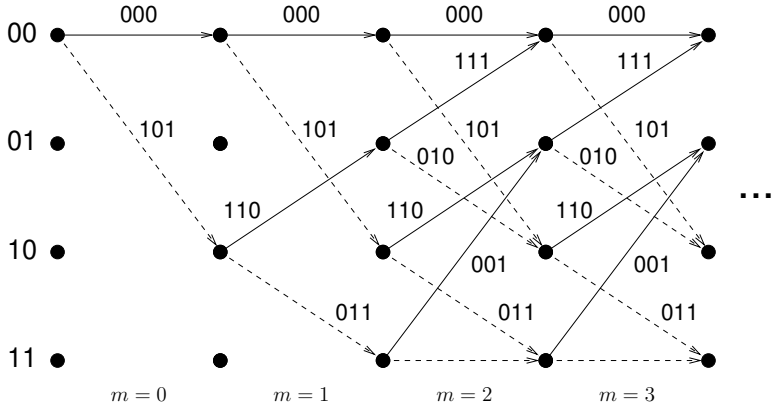


FIGURE A.3 Représentation en treillis de transitions d'états possibles à partir du début d'une trame accompagné du code transmis sur les transitions.

précis. C'est cette métrique probabiliste qui tient lieu de distance. Dans le contexte des communications numériques, on observera l'information reçue correspondant aux trois bits p_1, p_2 et p_3. Les paramètres reçus du canal qui sont associés à ces bits sont en fait des valeurs analogiques bruitées (typiquement des tensions à la sortie d'un filtre) qu'on appellera *variables de décision*. Dans le cas d'un décodage à décision dure, on prend une décision sur la

valeur des bits reçus avant d'effectuer le décodage. Ainsi, si une variable de décision pour p_1 est supérieure à zéro, on décide qu'il s'agit d'un '1' et sinon qu'il s'agit d'un '0'. Pour chaque branche de l'étage de treillis correspondant, on compare les 3 bits ainsi reçus aux 3 bits qu'on aurait dû recevoir idéalement pour cette branche. Le nombre de bits différents entre les deux constitue une mesure de distance appropriée et peut être utilisé comme métrique. Il s'agit de la *distance de Hamming*.

Toujours avec l'encodeur de la figure A.1, supposons que les quatre premiers bits à transmettre sont 1011. Comme la machine démarre à l'état 00, la séquence d'états suivie sera 00, 10, 01, 10, 11 et les bits encodés et transmis correspondants sont 101-110-010-011. Supposons ensuite qu'après l'étape de décision dure, les bits 4 et 9 sont en erreur. Le décodeur reçoit donc la série de bits 101-010-011-011. À partir de cette suite de code, on calcule la distance (de Hamming) pour chaque transition et on obtient le treillis de la figure A.4 (vérifiez que vous arrivez aux mêmes distances pour les 2 premières transitions du treillis). Du début à la fin du treillis, le chemin avec la distance totale la plus petite est celui qui passe par la bonne suite d'états car sa somme est de 0+1+1+0=2. Donc, dans ce cas, le décodeur serait capable de récupérer les bits transmis malgré les deux erreurs dans le canal. En effet, si on connait la séquence d'états suivie, on connait nécessairement les bits transmis puisque ce sont ces derniers qui induisent les choix de transition.

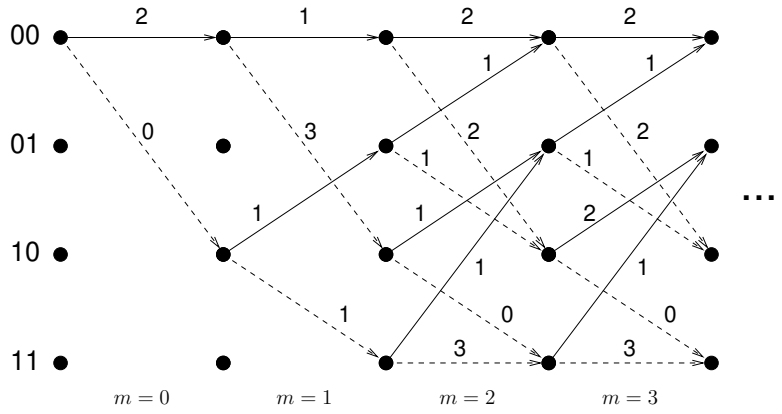


FIGURE A.4 Treillis avec métriques calculées selon la distance de Hamming pour le message erroné 101-010-011-011.

Si le signal reçu n'était pas bruité (aucune erreur), le chemin optimal aurait une distance totale nulle mais le décodeur ne servirait à rien. On voit que si on regarde qu'un étage isolé du treillis et qu'il y a une erreur sur un bit, on pourrait aisément choisir la mauvaise branche. La puissance du décodeur vient du fait d'observer une séquence complète d'une certaine longueur.

L'algorithme de Viterbi permet donc de trouver le plus court chemin dans un tel treillis et ce, sans examiner TOUS les chemins possibles ce qui deviendrait rapidement très lourd (particulièrement pour un nombre d'états plus élevé). En fait, on peut éliminer la majorité des chemins en cours de route sans problème. Si on considère l'étage m du treillis dans notre exemple, il y a un total de $2 + 4 + 8^{(m-1)}$ chemins. Le nombre de chemins croît donc exponentiellement avec le nombre d'étages parcourus. Cependant, à un étage donné, il n'est

nécessaire de considérer qu'un seul chemin arrivant à chacun des états. Ainsi, il y aura après réception du $m$ième bit, un certain nombre de chemins qui aboutissent à l'état 00. Mais seul le chemin le plus court parmi ces derniers est pertinent, parce que si la séquence optimale passe par l'état 0 à cet étage-là, elle passe nécessairement par le chemin le plus court pour cette portion du treillis.

Les seules informations nécessaires à la fin de l'étage $m-1$ sont les chemins les plus courts qui aboutissent à chacun des états (dans notre exemple, il y en aurait 4) ainsi que leurs longueurs. Ces chemins, dénotés *chemins survivants* ou simplement *survivants*, sont entièrement définis par la séquence d'états suivie et la longueur cumulée. À l'étage m, on prend les survivants de l'état précédent et on y ajoute les branches appropriées de l'étage courant. On obtiendra, pour notre exemple, un total de 8 chemins, soit 2 chemins aboutissant à chacun des 4 états. On calcule les longueurs de ces 8 chemins en ajoutant pour chacun la métrique de la nouvelle branche à la longueur du chemin survivant auquel elle est rattachée) et on choisit le plus court pour chacun des 4 états, définissant ainsi les nouveaux survivants.

Cette étape de mise à jour des survivants à chaque étage du treillis constitue le coeur de l'algorithme de Viterbi. C'est ce qu'on appelle l'opération d'addition-comparaison-sélection, ou ACS. Comme l'illustre le diagramme conceptuel de la figure A.5, l'algorithme complet comporte trois fonctions qui interviennent en séquence à chaque étage du treillis, soit :

1. le calcul des métriques pour chacune des branches ;
2. la mise à jour des survivants via le bloc ACS ;
3. l'entreposage en mémoire des parcours et longueurs des survivants.

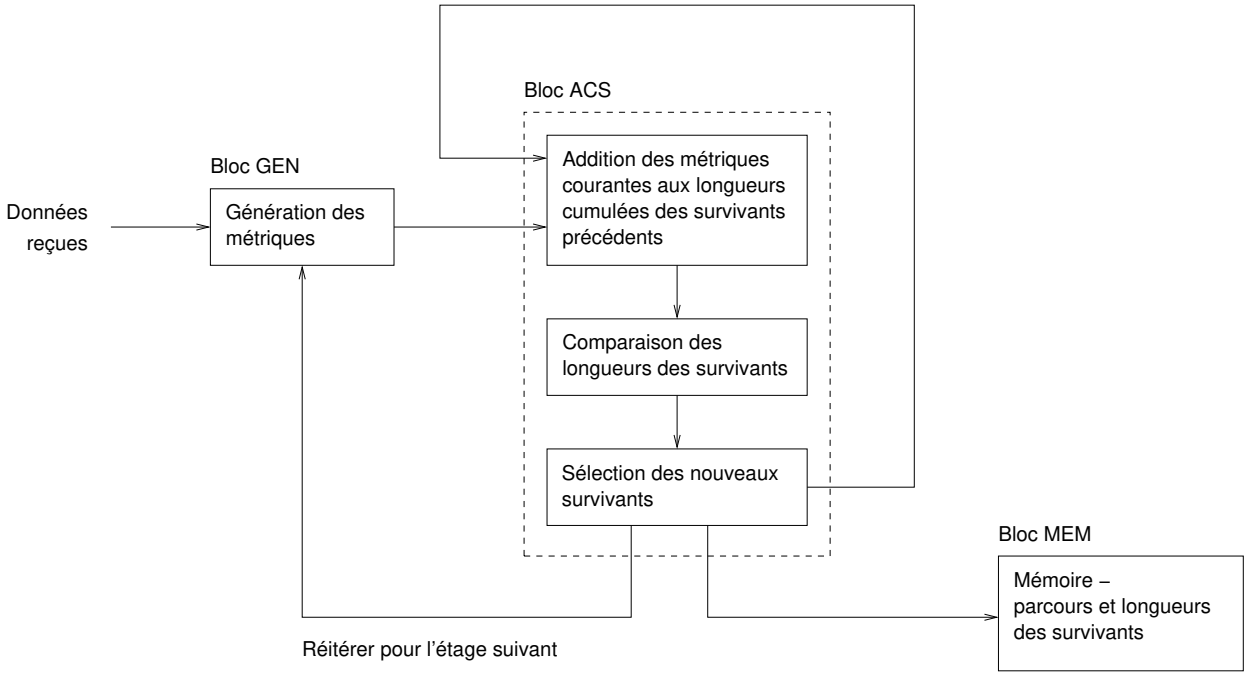


FIGURE A.5 Diagramme conceptuel de l'algorithme de Viterbi

A.4 Gestion de la mémoire

Pour une trame courte, il est judicieux d'attendre la fin de la trame pour choisir le chemin le plus court. Pour une trame très longue par contre, on ne voudra pas nécessairement garder en mémoire les métriques depuis le début de la trame. Il faudra alors tronquer les mémoires de parcours à une longueur acceptable δ. Ceci implique qu'il n'est plus possible d'attendre la fin de la trame pour décider de la séquence complète. On prendra plutôt des décisions partielles en cours de route. Ainsi, la décision finale pour l'instant $m - \delta$ doit être prise à l'instant m afin qu'on puisse se départir de la portion de mémoire associée. Ceci peut se faire sans affecter outre mesure la performance de l'algorithme étant donné que les survivants à l'instant m finissent par se fusionner en un seul lorsqu'ils sont retracés suffisamment loin dans le passé. Il convient donc de choisir un δ assez grand pour que les survivants soient fusionnés à l'instant $m - \delta$ avec une grande probabilité. Le critère généralement accepté est le suivant :

$$\delta \geq 5K. \tag{A.2}$$

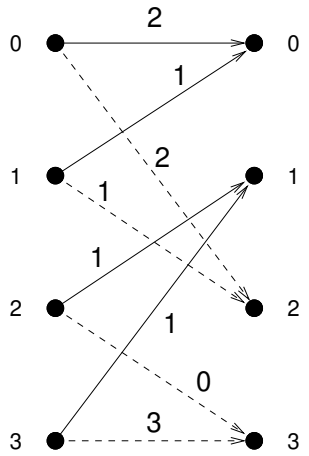
ce qui correspond à $\delta \geq 15$ dans notre situation spécifique. Dans les rares cas où les survivants ne concordent pas, un choix arbitraire n'entraînera qu'une dégradation négligeable de la performance.

A.5 Forme matricielle de l'algorithme

A.5.1 Reformulation en termes de matrices et vecteurs

L'étape d'addition-comparaison-sélection de l'algorithme de Viterbi peut s'accomplir par le biais d'une opération matrice-vecteur caractérisée par la même forme que la multiplication matrice-vecteur. Cet homomorphisme est intéressant parce que la multiplication matrice-vecteur se prête bien au parallélisme au niveau des données.

On peut représenter l'étage m du treillis par une matrice $\mathbf{T}_m$ dont l'élément $t_m(i, j)$ contient le poids (donc la métrique) de la branche qui part de l'état i pour aboutir à l'état j (voir figure A.6). On assigne un poids infini (ou un très grand nombre) aux éléments pour lesquels il n'existe pas de branche dans le treillis.



$$\mathbf{T}_m = \begin{bmatrix} 2 & 1 & \infty & \infty \\ \infty & \infty & 1 & 1 \\ 2 & 1 & \infty & \infty \\ \infty & \infty & 0 & 3 \end{bmatrix}$$

FIGURE A.6 Représentation matricielle d'un étage du treillis

Étant donné le vecteur $\mathbf{s}_m$ de N éléments contenant les longueurs des survivants immédiatement avant le $m$ième transition. La mise à jour des survivants peut alors s'écrire

$$\mathbf{s}_{m+1} = \mathbf{T}_m \otimes \mathbf{s}_m, \tag{A.3}$$

où l'opérateur $\otimes$ se définit comme suit :

$$s_{m+1}(i) = \min_{j\in[0,\cdots,N-1]} \left(T_m(i,j) + s_m(j)\right), \tag{A.4}$$

ce qui est analogue à la définition de la multiplication matrice-vecteur, c-à-d

$$s_{m+1}(i) = \sum_{j\in[0,\cdots,N-1]} \left(T_m(i,j) \times s_m(j)\right), \tag{A.5}$$

ce qui correspond également à

$$\mathbf{s}_{m+1} = \mathbf{T}_m\mathbf{s}_m. \tag{A.6}$$

En effet, on observe que la forme est la même, mais que la multiplication a été remplacée par une addition et la sommation par l'opérateur minimum.

Il ne suffit pas cependant de conserver les longueurs des parcours, mais également la suite d'états par laquelle chacun est passé. Il faut donc à chaque étage m et pour chaque nouveau survivant identifier et retenir l'état d'où provient la dernière branche qu'on vient d'ajouter. Mathématiquement, pour le survivant i, on peut écrire cette opération comme suit :

$$e_i[m] = \operatorname*{argmin}_{j\in[0,\cdots,N-1]} \left(T_m(i,j) + s_m(j)\right), \tag{A.7}$$

ce qui signifie que $e_i(m)$ (le $m$ième état par lequel passe le survivant i) prend la valeur de j qui, substituée dans $(T_m(i,j) + s_m(j))$, donne le résultat minimum.

A.5.2 Concept du treillis serré

Étant donné que la connectivité à chaque étage du treillis est relativement basse pour la plupart des codes convolutifs, beaucoup de positions dans les matrices $\mathbf{T}_m$ se voient assigner une valeur infinie (ou très grande). Il s'ensuit que la réalisation d'une telle opération matricielle sur un processeur quelconque résultera en une utilisation inefficace du matériel puisque plusieurs opérations seront effectuées pour des branches qui en fait n'existent pas. Pourrait-on alors imaginer une transformation du problème telle que le treillis serait au préalable pleinement connecté ?

Une telle transformation a été proposée en 1989 par Chang et Yao [**?**] et sera dénotée ici **treillis serré** (*strongly-connected trellis*). Le concept est très simple : il s'agit de combiner un certain nombre r d'étages successifs du treillis en un seul pour former un nouveau treillis à haute connectivité. Il y a en effet une correspondance un à un entre chaque branche du treillis serré et chaque de chemin de r branches dans le treillis original (voir figure A.7).

On observe dans la figure A.7 que pour l'encodeur de taux $\frac{1}{3}$ de la figure A.1, il suffit pour déterminer la métrique d'une branche du treillis serré de concaténer les codes des deux branches originales correspondantes et de comparer les 6 bits obtenus avec les 6 bits reçus correspondants.

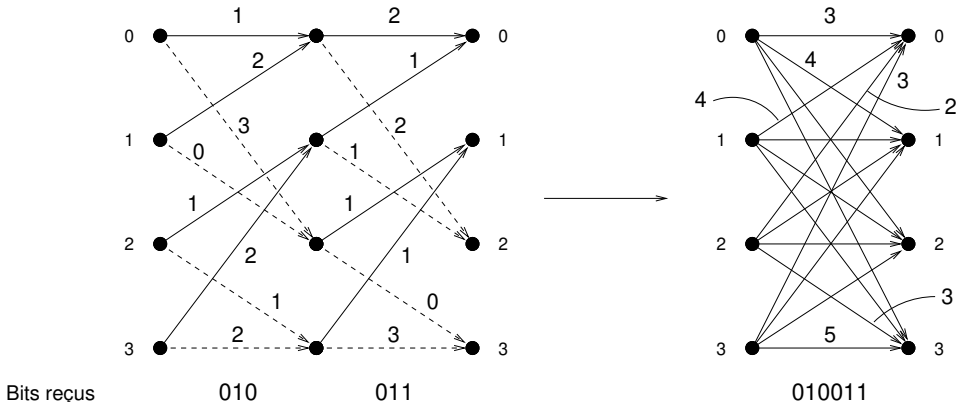


FIGURE A.7 Deux étages du treillis original pour l'exemple d'un encodeur de taux $\frac{1}{3}$ et l'étage de treillis serré correspondant. Quelques métriques sont données pour le treillis serré, correspondant à la somme des deux branches correspondantes dans les deux étages originaux.

A.6 Codes fournis

Il vous faut réaliser en code assembleur MIPS la partie critique de l'algorithme de Viterbi (section B.2), soit l'étape ACS (Addition-Comparaison-Sélection) de l'algorithme et ce, seulement pour le code convolutionnel à 4 états donné en exemple dans cette annexe. Vous validerez votre code assembleur à l'aide du simulateur MARS. Vous n'avez pas à réaliser l'encodeur. Toutefois, à titre informatif et expérimental, un code VHDL réalisant cet encodeur sous forme de testbench vous est fourni (voir site web). Le résultat d'une simulation dans Vivado avec ce code est donné à la figure A.8. Le message binaire original dans cette simulation est "1011 0001 1010" et la simulation montre que la séquence d'états correspondante est 0–2–1–2–3–1–0–0–2–3–1–2–1. De même, les bits transmis (codés) suivent la séquence 5–6–2–3–1–7–0–5–3–1–2–6, où chaque chiffre correspond à un vecteur de 3 bits. Ainsi, les 9 premiers bits transmis seraient "101-110-010." Tout ceci peut se vérifier à la main à partir de la théorie expliquée aux sections précédentes, et particulièrement du graphe de la machine à états de l'encodeur (figure A.2).

On remarque ici qu'on a choisi une fréquence d'horloge arbitraire mais élevée afin de faciliter l'affichage des résultats de simulation. Toutefois, un tel encodeur fonctionnerait normalement dans une vraie application à la fréquence de transmission des bits, à choisir selon la situation spécifique.

A.6.1 Explications du code C

Le code fourni sur la page web de l'APP (*DecodageViterbi.c*) constitue un code C de référence pour le décodeur de Viterbi de la problématique. Il s'agit d'un décodeur quasicomplet, comprenant la génération des métriques mais pas l'entreposage des parcours survivants. Il ne fait que mettre à jour les *longueurs* des parcours survivants à chaque étage et est basé sur le concept de treillis serré qui permet une réalisation sous forme matricielle de l'étape

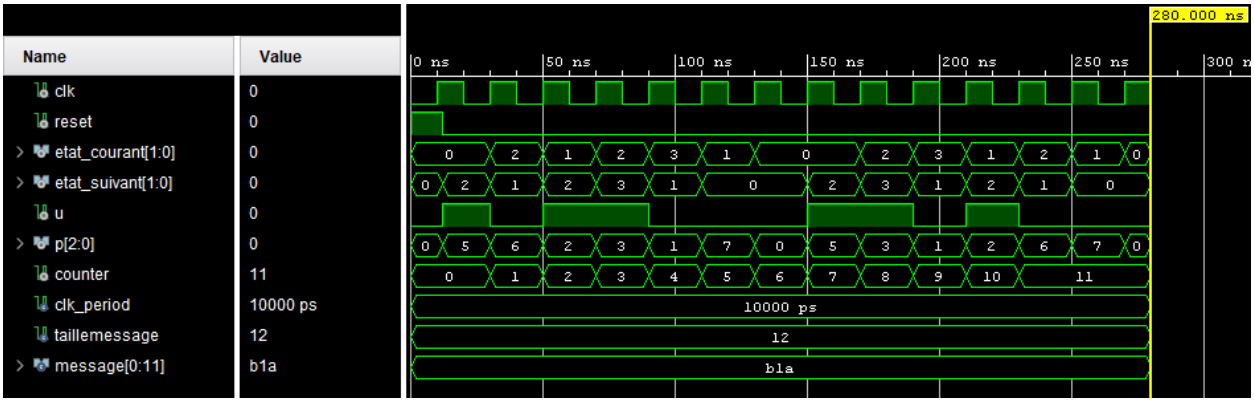


FIGURE A.8 Simulation avec le banc d'essai `encod_convol.vhd`

ACS. Par ailleurs, le code est spécifique à l'encodeur de taux $\frac{1}{3}$ décrit dans cet annexe. En effet, un code plus général aurait été plus difficile à comprendre et ce n'est pas nécessaire pour résoudre la problématique. On note également que la séquence de bits codés fournie en entrée correspond à la séquence produite par l'encodeur en VHDL, tel que discuté plus haut.

Il a été vérifié que ce code compile correctement (aucune erreur ni avertissement) avec gcc version 9.3.0 (Ubuntu 20.04). Par ailleurs, il tourne correctement et donne le résultat attendu, soit des longueurs de survivants de 4, 0, 4 et 2. Tel qu'illustré à la figure A.7, il faut combiner 2 étages du treillis original en un seul pour former un étage de treillis pleinement connecté. Le programme en C se sert des matrices `pr1` et `pr2` pour déterminer les métriques dans un étage du treillis serré. Par exemple, la branche du treillis serré qui va de l'état 0 à l'état 3 correspond dans le treillis original à une première branche de 0 à 2 (laquelle est associée aux bits codés 0b101 ou 5 en décimal) et à une deuxième branche de 2 à 3 (associée à 0b011 ou 3 en décimal). Ainsi, on constate qu'à la position (0, 3) de la matrice `pr1`, on retrouve 5, soit les bits codés de la première branche associée à cette transition. De même, à la même position de la matrice `pr2`, on retrouve 3, soit les bits codés de la deuxième branche du treillis original pour cette transition dans le treillis serré. On voit donc que les deux matrices sont indexées selon (état de départ, état d'arrivée) pour la branche dans le treillis serré qu'on souhaite examiner.

Comme on doit examiner les bits de deux branches successives du treillis original, le code principal `main` vérifie d'abord que l'index `i` est impair et calcule les métriques sur une paire de branches successives (c-à-d une branche du treillis serré). La distance de Hamming est définie comme le nombre de bits qui diffèrent. C'est pourquoi le calcul des métriques dans `genmetrique` s'appuie sur un ou-exclusif des bits reçus avec les bits attendus pour la branche examinée. On utilise ensuite la fonction `popcount` pour compter le nombre de bits à '1'. Une implémentation très simple de `popcount` est proposée ici puisque les valeurs sont limitées à 3 bits avec le code considéré.

Pour vérifier sommairement le bon fonctionnement du code, on note que les bits codés ne contiennent pas d'erreur ce qui implique qu'un parcours survivant à la fin devrait être de longueur nulle. C'est bien le cas et il s'agit du survivant qui se termine à l'état 1. Comme noté plus haut, c'est bien l'état où la machine se trouve à la fin de la séquence considérée. Par

ailleurs, on peut également vérifier que les longueurs des autres survivants sont correctes en notant que tous les survivants fusionnent à un étage de treillis serré dans le passé. Ceci découle de la simplicité relative de ce code (longueur de contrainte faible) qui fait que deux transitions dans l'encodeur suffisent pour passer de n'importe quel état à n'importe quel autre état. Il s'ensuit que les 4 parcours survivants ont une longueur nulle jusqu'aux 2 dernières branches (puisqu'ils suivent la séquence correcte d'états jusque-là). La fin de la séquence d'états de l'encodeur est 1-2-1. Pour l'hypothèse où la séquence se terminerait à l'état 0 (correspondant au survivant 0), la fin de la séquence deviendrait 1-0-0 et les bits correspondants seraient 111-000. Or, les bits reçus pour 1-2-1 sont 010-110. On voit que la distance de Hamming entre les deux séries de 6 bits est de 4, tout comme la longueur du survivant 0 donnée par le programme.

De même, dans l'hypothèse où la séquence se termine à l'état 2, les 3 derniers états seraient 1-0-2 au lieu de 1-2-1. Les bits codés correspondants sont 111-101, ce qui donne une distance de Hamming de 4 à nouveau, tout comme la longueur du survivant 2 donnée par le programme. Finalement, dans l'hypothèse où la séquence de terminerait à l'état 3, les 3 derniers états seraient 1-2-3 au lieu de 1-2-1. Les bits codés ici seraient 010-011, résultant en une distance de Hamming de 2, également vérifiée par le programme.

B Algorithme de Viterbi - Devoir

B.1 Travail à faire

Pour le devoir, vous devez transformer le calcul des survivants (2 fonctions) en assembleur ainsi qu'implémenter un test unitaire pour cette fonction. Vous pouvez vous servir du code C pour prégénérer les métriques et les placer à l'avance dans le segment de données. Il n'est pas nécessaire de garder en mémoire les séquences d'états. Il suffit de mettre à jour les survivants. Surtout, ne traduisez pas les autres fonctions du code C comme *genmetrique* à moins que vous souhaitiez le faire comme exercice supplémentaire. Le devoir vise la rédaction des deux fonctions principales et une fonction main effectuant un test unitaire.

B.2 Revue accélérée du code

Les explications sur le code fourni sont détaillées à la section A.6.1. Pour amorcer une compréhension du code, il est possible de passer par une analyse de calcul matriciel. Par exemple, Si on souhaite effectuer en C une multiplication matrice-vecteur de taille N (c-à-d que la matrice est de taille $N \times N$ et le vecteur de taille $N \times 1$) définie comme suit :

$$\mathbf{y} = \mathbf{A}\mathbf{x}, \tag{B.1}$$

on peut procéder par le biais de deux boucles imbriquées pour réaliser l'opération élément par élément conformément à

$$y(i) = \sum_{j \in [0,\cdots,N-1]} \left(A_m(i,j) \times x(j)\right), \tag{B.2}$$

ce qui donne

```
1         for (i = 0; i < N; i++) {
2                 y[i]=0;
3                 for (j = 0; j < N; j++) {
4                         y[i]=y[i]+A[i][j] * x[j];
5                 }
6         }
```

De même, si on souhaite réaliser l'opération d'addition-comparaison-sélection sous forme matricielle définie comme

$$\mathbf{y} = \mathbf{A} \otimes \mathbf{x}, \tag{B.3}$$

ce qui correspond à

$$y(i) = \min_{j \in [0,\cdots,N-1]} \left(A(i,j) + x(j)\right), \tag{B.4}$$

on pourra procéder comme suit :

```
1         for (i = 0; i < N; i++) {
2                 y[i]=250;
```

```
3                   for (j = 0; j < N; j++) {
4                           temp=A[i][j]+x[j];
5                           if (temp < y[i])
6                           y[i] = temp;
7                   }
8           }
```

On note que la valeur de départ de `y[i]` correspond en théorie à l'infini, mais doit dans la pratique simplement être suffisamment grande par rapport aux longueurs de chemins calculées. Dans le cas du code convolutif étudié, les longueurs de parcours en l'absence d'erreur de détection ne peuvent pas dépasser 6. En supposant une probabilité d'erreur de 1% et étant donné qu'il y a 6 bits par étage de treillis serré, il faudrait avoir une trame de 24400 bits pour arriver à dépasser la valeur de départ de 250. Ceci ne se produirait jamais dans une vraie application, même avec une probabilité d'erreur plus élevée à cause du critère de troncation (A.2). Pour le code étudié, cela limite la longueur des parcours à des séquences de 45 bits (3×5 étages de 3 bits).

Pour les besoins de la problématique, il suffit de savoir que 250 est une valeur de départ adéquate.

Peut-on éviter d'utiliser une structure conditionnelle `if` et suivre de plus près le gabarit de code pour la multiplication matrice-vecteur donné plus haut ? C'est effectivement possible, en utilisant l'opérateur conditionnel `?` comme suit :

```
1           for (i = 0; i < N; i++) {
2                   y[i]=250;
3                   for (j = 0; j < N; j++) {
4                           temp=A[i][j]+x[j];
5                           y[i] = temp < y[i] ? temp : y[i];
6                   }
7           }
```

Notez enfin que le code fourni sépare les deux boucles imbriquées en deux fonctions (CalculSurvivants et acs). Pour le devoir, on vous demande de conserver cette distinction.

```
1   void acs(unsigned *met, int *sinput, int *soutput)
2   {
3           unsigned int temp, j;
4           for(j=0; j< N; j++) {
5                   temp = met[j]+sinput[j];
6                   *soutput = temp < *soutput ? temp :
                        *soutput;
7           }
8   }
9   void CalculSurvivants( unsigned int *met, int *
        sinput, int *soutput)
10  {
11          unsigned int i;
12          for (i=0; i< N; i++) {
13                  soutput[i]=250;
14                  acs(&met[i*N], sinput, &soutput[i]);
15          }
16  }
```

B.3 Test unitaire

Votre code de devoir doit inclure un test unitaire documenté (dans les commentaires). Vous pouvez rédiger le vôtre, ou encore vous inspirer du code C pour trouver des valeurs pertinentes. Le choix de ce test est important pour justement trouver des erreurs dans votre code, erreurs pas toujours apparentes, surtout si le test est trop simple.

C Correspondance VHDL - COD5

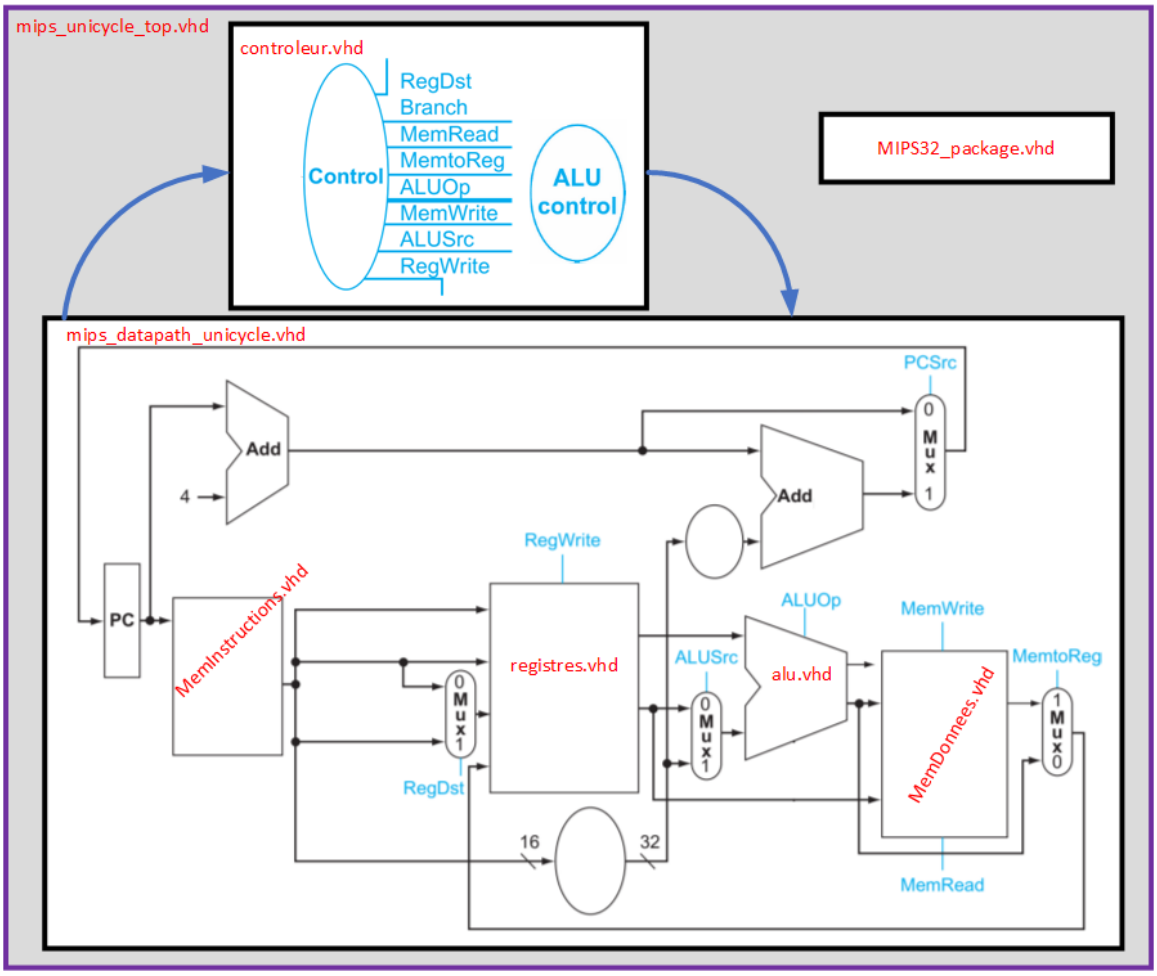


FIGURE C.1 Chronogramme du contrôleur de cache avec requêtes en continu, avec 2 succès suivis d'un échec.

D Annexe sur l'implémentation SIMD en VHDL

Le projet de l'APP4 est une entreprise d'intégration système, c'est-à-dire un cas où il faut maitriser à la fois les outils d'ingénierie (Xilinx/Vivado et Mars), le problème à résoudre (l'algorithme) et le moyen de résolution (chemins de données et de contrôle en VHDL). Le véritable défi vient de la mise en relation des trois, où l'ingénieur doit imaginer des solutions à partir de ces trois éléments. Dans le contexte de l'APP, il faut donc commencer par comprendre séparément. Xilinx/Vivado et le VHDL ont déjà étés vus aux APPs 1 et 2, une très bonne chose. L'outil MARS est assez simple et exploré au laboratoire 1. Ensuite, l'algorithme sert à la rédaction du devoir (un deux-pour-un qui fait gagner du temps). Il reste donc à comprendre les chemins de données, étude répartie dans les lectures, procéduraux et laboratoires. La problématique demande l'ajout d'extension SIMD à une organisation unicycle, travail qui viendra cristalliser votre compréhension des interactions entre les modules d'un processeur.

La procédure encouragée se décline en 4 grandes étapes (①-②-③-④). ① Revoir et modifier votre code assembleur (comme au procédural 2, problème 2). Cette première analyse fera ressortir une liste d'instructions à ajouter à l'organisation MIPS, et doit être remis avec le rapport. Après la sélection des instructions suivra ② la conception des modifications à faire sur le chemin de données et de contrôle de l'organisation unicycle (procédural 1 problème 5 et procédural 2 problème 3). Il sera très profitable de bien étoffer ce plan pour minimiser les itérations dans le code VHDL et ainsi économiser du temps précieux. Un schéma (par exemple dessiné par-dessus une figure du livre de référence) pourra vous servir de guide et aider à partager le travail entre les membres d'équipe. ③ Ces modifications seront ensuite validées suivant un plan de vérification, de complexité égale à l'instruction testée. Enfin, une fois que vous serez confiants que votre organisation fonctionne correctement, vous pourrez passer à ④ l'intégration système, c'est-à-dire d'exécuter et vérifier votre code assembleur exploitant les nouvelles instructions SIMD sur votre organisation originale.

D.1 Organisation unicycle de référence

Une organisation unicyle MIPS partielle vous est fournie comme point de départ, organisation calquée sur les figures de votre livre de référence. Vous aurez donc avantage à alterner entre vos lectures et l'utilisation du code pour raffiner votre compréhension de cette organisation. Un effort particulier a été investi pour conserver le plus possible les mêmes noms de signaux VHDL avec les figures du livre. Un document en français est également disponible et offre une revue alternative des mêmes concepts. Enfin, notez que ce ne sont pas toutes les instructions du jeu MIPS 32-bit qui sont implémentées dans le unicycle de référence. Référez-vous au fichier *MIPS32_package.vhd* pour la liste des instructions disponibles. Vous avez ainsi le choix d'ajouter des instructions manquantes vous-même au besoin, ou encore de modifier votre code assembleur pour faire fonctionner votre programme.

D.2 Pistes d'évolution pour l'algorithme ACS

D.2.1 Revue de l'algorithme

Pour notre encodeur à 4 états, on réalisera une extension SIMD de taille 4, c.-à-d. qu'il faudra gérer les opérations en blocs de 4 valeurs de 32 bits pour les registres, les arithmétiques, les

accès mémoire, etc. On pourra choisir d'ajouter 3 unités arithmétiques à celle de base utilisée par les instructions scalaires (normale, 1 mot de 32-bits à la fois). En alternative, comme les unités arithmétiques additionnelles peuvent être plus simples (elles n'ont qu'à supporter un sous-ensemble d'opérations spécifiques), des versions simplifiées pourraient être mises en place, et l'unité arithmétique de base ne servant alors qu'aux instructions scalaires.

Le code C fourni comme référence est conçu pour être très proche de la version accélérée par des extensions SIMD. Au même titre que le problème 2 du procédural 2, vous aurez à ajouter des instructions `lwv`, `swv` et `addv`. En plus de ces instructions, il sera judicieux d'exploiter des optimisations un peu plus subtiles. Dans un premier temps, les instructions supportées dans l'organisation fournie traduisent l'opération de sélection par des branchements conditionnels, comme un (`if`) en C. Cependant, dans le même esprit de l'opérateur conditionnel `?` du C/C++, il existe une paire d'instructions (`movz` et `movn`) permettant de conditionnellement copier un registre, sauvant quelques coups d'horloge.

Pour l'appliquer au cas SIMD, il faut d'abord se souvenir que le concept SIMD s'articule essentiellement comme plusieurs d'unités d'exécutions (dans ce cas-ci des unités arithmétiques) pilotées par un seul décodeur d'instructions. Dans ce cas, le test de condition s'effectuera sur des registres vectoriels plutôt qu'un registre unique. Le résultat de la condition sera vrai pour certaines unités et faux pour d'autres. Mais comme toutes les unités doivent exécuter les mêmes instructions en même temps, il faudra penser à une stratégie pour gérer les différences instantanées. La philosophie derrière les instructions `movz`/`movn` devient un bon guide pour résoudre ce problème, ce qui mènera à l'implémentation d'une instruction `movnv` (move-when-not-zero vector) ou `movzv` (move-when-zero vector). Afin de complémenter cette extension évoluée, vous aurez aussi besoin d'une instruction pour faire la comparaison, par exemple `sltv` (set less than vector) ou `sltuv`. Ceci nous mène à une liste d'extensions SIMD possibles :

- `lwv rt, decalage(rs)` — Charge le registre vectoriel `rt` avec les données situées à l'adresse mémoire obtenue par la somme du contenu du registre *scalaire* `rs` plus la constante *décalage*.
- `swv rt, decalage(rs)` — Écrit le contenu du registre vectoriel `rt` à l'adresse mémoire obtenue par la somme du contenu du registre *scalaire* `rs` plus la constante *décalage*.
- `addv rd, rs, rt` — addition vectorielle où `rd`, `rs` et `rt` sont tous trois des registres vectoriels ;
- `sltv rd, rs, rt` — *set less than unsigned* vectoriel : effectue la même opération que slt, mais en parallèle sur chaque mot des registres vectoriels `rd`, `rs` et `rt` ;
- `movnv rd, rs, rt` — *move if* vectoriel : élément par élément, transfère conditionnel du contenu des mots du registre vectoriel `rs` au registre vectoriel `rd`, si le mot corresondant dans le registre vectoriel `rt` ne vaut pas 0.
- `movzv rd, rs, rt` — *move if* vectoriel : élément par élément,transfère conditionnel du contenu des mots du registre vectoriel `rs` au registre vectoriel `rd`, si le mot corresondant dans le registre vectoriel `rt` vaut 0.

Cette liste n'est pas exhaustive et se veut un bassin de suggestions de départ. À vous de déterminer la liste exacte.

D.2.2 Réorganisation des calculs

Les instructions données ci-haut forment un sous-ensemble partiel pour réaliser le bloc ACS sous forme parallèle sur 4 unités d'exécution. Il y a plusieurs façons de répartir le travail sur les 4 unités en question et selon le choix, il faudra peut-être ajouter au moins une instruction au sous-ensemble donné.

Stratégie directe

Dans l'annexe sur l'algorithme de Viterbi, on voit que le bloc ACS peut se réaliser sous forme matricielle selon

$$y(i) = \min_{j \in [0,\cdots,N-1]} \left(A(i,j) + x(j) \right), \tag{D.1}$$

où **x** est le vecteur des longueurs des survivants de l'étage précédent (*sinput* dans le code C) et **y** est le vecteur des longueurs de survivants mis à jour pour l'étage courant (*soutput* dans le code C). L'équation D.1 est visuellement distribuée pour le cas de N = 4 à la figure D.1. On voit que pour une valeur donnée $y(i)$, il y a 4 sommes et 3 comparaisons à faire, effectués par le code C avec la boucle i=[0..3] à l'horizontale et la boucle j=[0..3] à la verticale. Autrement, les calculs peuvent être faits dans n'importe quel ordre.

$$\begin{array}{cccc}
y(0) = A(0,0) + x(0) & y(1) = A(1,0) + x(0) & y(2) = A(2,0) + x(0) & y(3) = A(3,0) + x(0) \\
y(0) = A(0,1) + x(1) & y(1) = A(1,1) + x(1) & y(2) = A(2,1) + x(1) & y(3) = A(3,1) + x(1) \\
y(0) = A(0,2) + x(2) & y(1) = A(1,2) + x(2) & y(2) = A(2,2) + x(2) & y(3) = A(3,2) + x(2) \\
y(0) = A(0,3) + x(3) & y(1) = A(1,3) + x(3) & y(2) = A(2,3) + x(3) & y(3) = A(3,3) + x(3) \\
\\
y(0) = min(colonne) & y(1) = min(colonne) & y(2) = min(colonne) & y(3) = min(colonne)
\end{array}$$

FIGURE D.1 Opérations atomiques réalisées par une boucle de calcul ACS.

Dans cette optique, il y a plusieurs façons de redistribuer ces calculs sur 4 unités d'exécution en parallèle. Si on assigne le calcul de $y(i)$ à la $i$ième unité d'exécution et que les calculs se font dans le même ordre sur les 4 unités, on obtient la répartition de travail illustrée à la figure D.2. Chaque ligne (ou étape) est calculée en même temps. Cette approche a l'avantage de ne pas nécessiter de mouvements de données pendant le calcul. Cependant, il faut au départ que les valeurs du vecteur **x** soient dans des registres scalaires (et non vectoriels) et il faudra implanter une instruction d'addition vecteur-scalaire `addvs` (une valeur 32-bit dirigée vers une des entrées de chaque unité arithmétique). En fin de calcul les valeurs du vecteur **y** (qui doit devenir le vecteur **x** pour la prochaine itération) se trouvent dans un registre vectoriel et il faudra donc éventuellement les déplacer d'une façon ou d'une autre vers un ou des registres scalaires. Au lieu de cela on pourrait réaliser l'instruction `addvs` de manière à

ce qu'elle puise son scalaire dans un des champs d'un registre vectoriel au lieu d'un registre scalaire (ajout d'un multiplexeur). Bref, il y a plusieurs chemins possibles, à vous de choisir.

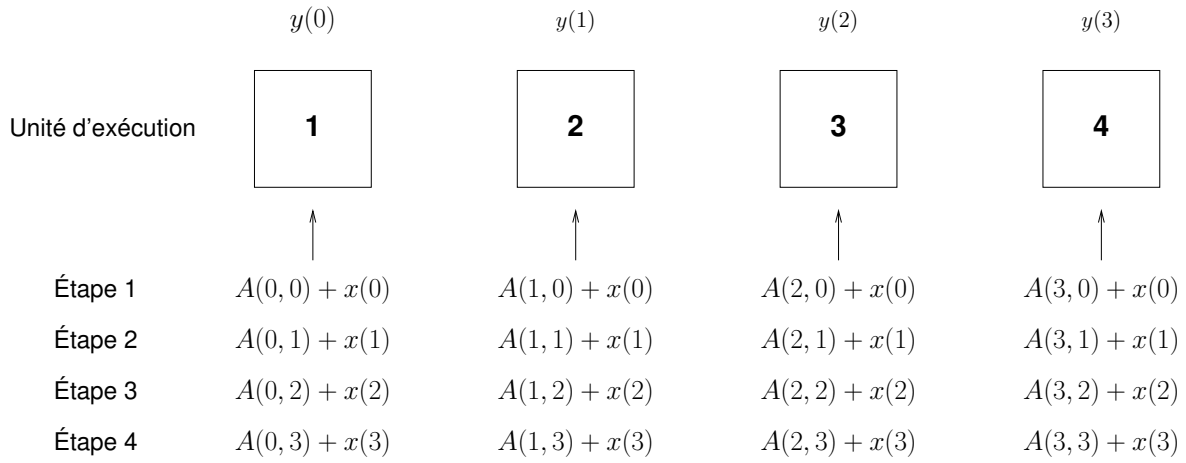


FIGURE D.2 Répartition des calculs par la méthode vectorielle directe.

Stratégie par décalage

On peut aussi réordonner le calcul de manière à pouvoir à chaque étape additionner le vecteur $\mathbf{x}$ au complet en utilisant `addv`. On obtiendra alors une rotation dans l'ordre des étapes d'une unité d'exécution à l'autre. Cette organisation du travail, dite "décalée," est illustrée à la figure D.3. Avantageusement, le vecteur des longueurs $\mathbf{x}$ n'a qu'à être présent dans un registre vectoriel au début des calculs et les éléments du vecteur résultant $\mathbf{y}$ se retrouveront aux mêmes endroits en fin de calcul, déjà prêts pour la prochaine itération. Cependant, il faut en cours de calcul à chaque étape effectuer une *rotation* des positions des valeurs de $\mathbf{x}$ à même le registre vectoriel. Il faudra pour ce faire supporter une extension additionnelle `rotv`. Dans ce cas-ci, le vecteur de sortie $\mathbf{y}$ reste en place tout au long du calcul alors que le vecteur $\mathbf{x}$ effectue une rotation d'un mot à chaque étape. Les couleurs identifient les valeurs du vecteur $\mathbf{x}$.

À l'opposé, on peut garder le vecteur $\mathbf{x}$ en place tout au long du calcul et faire plutôt tourner le vecteur de sortie $\mathbf{y}$. Parmi les 3 organisations, celle-ci est la seule où les résultats intermédiaires pour $\mathbf{y}$ doivent se déplacer. Cette organisation, désignée *entrelacée*, est illustrée à la figure D.4. Dans ce cas-ci, les couleurs identifient la position des valeurs intermédiaires du vecteur $\mathbf{y}$. Ici aussi, il faudra supporter une instruction effectuant la rotation des champs d'un registre vectoriel, soit `rotv`.

D.2.3 Notes utiles

On voit donc que l'organisation décalée nécessite moins de manipulations / déplacements de données, mais requiert une instruction originale différente. Il est donc très utile d'estimer quelle alternative offre le meilleur potentiel de gain de performance à haut niveau (calcul de performance prédictif). Les deux stratégies exigent aussi de revoir l'ordre des données dans le tableau de métriques $\mathbf{A}$: il est permis et même encouragé de changer la disposition initiale des données du tableau si cela accélère la vitesse du calcul. Ceci n'est pas une tâche que le

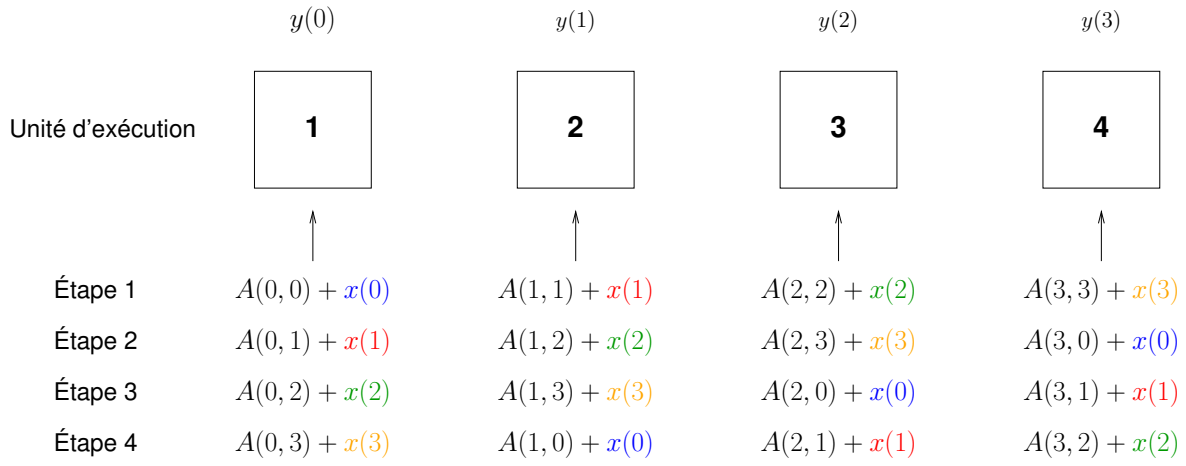


FIGURE D.3 Répartition des calculs par la méthode vectorielle décalée.

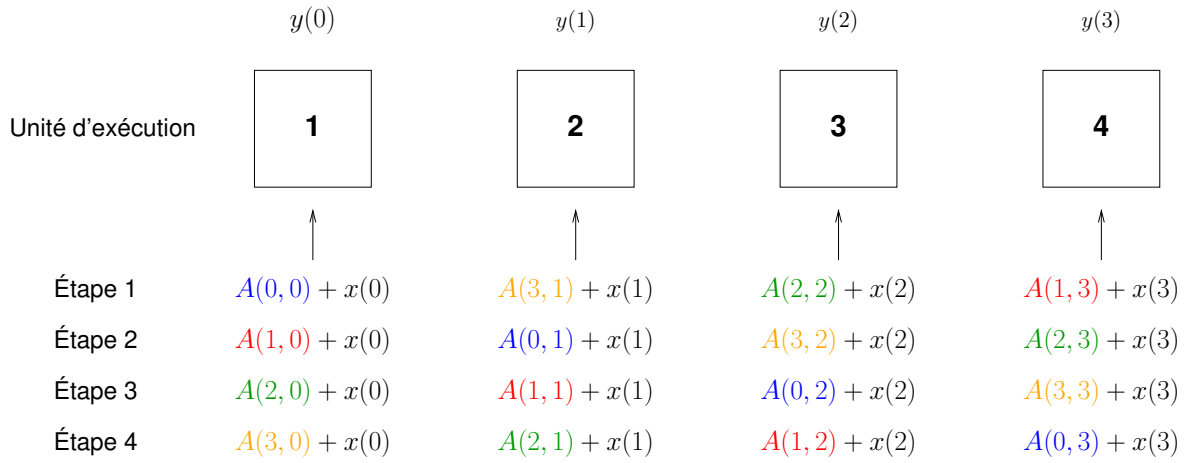


FIGURE D.4 Répartition de travail entrelacée sur architecture SIMD à 4 unités d'exécution.

logiciel assembleur doit accomplir, mais bien un réarrangement de l'ordre des données en mémoire dans le segment de données au niveau du code assembleur SIMD. Enfin, si vous trouvez des modifications ou réorganisations intéressantes à faire directement dans le code assembleur, cela viendra bonifier le gain obtenu par les extensions SIMD.

D.3 Ajout des extensions

Pour cette étape, revenez aux modifications d'organisation vues en procédural et à vos laboratoires : identifiez bien les rôles des instructions à ajouter ; leur encodage en langage machine ; les éléments essentiels, existants ou non, dans le chemin de données ; les champs requis et leurs connexions existantes dans le chemin de données. À présent s'ajoute aussi la possibilité de récupérer des pièces existantes, par exemple l'ALU. Il est tout à fait stratégique d'utiliser 4 instances de ce module plutôt que d'en écrire un spécifiquement pour les ALU supplémentaires.

L'ordre suggéré est de commencer par `lwv` et `swv`, puis `addvs` ou `addv`, avant de s'attaquer aux instructions plus exigeantes. Pour le choix de l'encodage des instructions, soyez straté-

giques : utilisez la carte résumée MIPS pour identifier soit des instructions similaires non implémentées, ou encore des codes libres où placer vos extensions. Tout est permis, pourvu que le jeu d'instructions déjà en place reste fonctionnel.

Mémoire à accès large bus Pour vous donner un coup de pouce avec les instructions `lwv` et `swv`, vous avez accès sur le site web de l'APP à un module de mémoire offrant à la fois un chemin d'accès de 32-bit et un autre de 128-bits (les deux chemins accèdent aux mêmes données). Son implémentation exige que les données récupérées par le bus de 128-bits soient référées dans l'assembleur par des adresses alignées sur des blocs de 4 mots (128 bits / 32 bits = ratio de 4). C'est généralement une exigence pour ce genre de bloc mémoire dit "dual port" avec largeur en multiples, très commun dans les FPGAs d'aujourd'hui.

Ensuite, à vous de voir comment modifier le banc de registre : modifier celui existant et y ajouter des registres vectoriels, ou ajouter une nouvelle instance modifiée et gérée par de nouveaux signaux de contrôle ? Les deux approches sont valides, pourvu que le jeu d'instructions déjà en place reste fonctionnel.

Mémoires cache

N'ajoutez pas de mémoires cache à votre implémentation SIMD, conservez l'approche simple des SRAM internes. Cependant, gardez en tête l'impact théorique des DRAM et des caches lors de vos réflexions à la rédaction du rapport.

D.4 Tests unitaires

Effectuez des tests unitaires ciblés sur vos instructions après chaque ajout : tout déverminer en un coup deviendra rapidement un mal de tête (je parle par expérience !). Cela vous permettra aussi de faire des archives intermédiaires de votre progrès avec une copie de retour en arrière. Encore mieux : servez-vous de git ou svn si vous êtes à l'aise avec ces outils.

Faire les tests unitaires à mesure, et les étoffer pour couvrir différents cas de figure placera rapidement les jalons pour la validation et les éléments qui y sont demandés, en plus de vous sauver beaucoup de temps de déverminage.

D.5 Intégration finale

Si les tests unitaires sont bien faits, alors les erreurs qui surviendront à l'intégration proviendront exclusivement de votre code assembleur avec SIMD. Il peut s'agir d'une erreur dans la séquence des instructions, de l'organisation des données dans les tableaux, de l'encodage en langage machine. Les erreurs peuvent aussi être très subtiles. Soyez méthodiques : si une première inspection rapide ne porte pas fruit, établissez clairement les valeurs attendues à différents moments avant l'arrivée du bogue et suivez le problème jusqu'à la source. Je serai présent lors de périodes de support en laboratoire prévues pour cela.

D.6 Conclusion

Restez positifs et ne vous découragez pas ! Parfois un seul bogue nous bloque pendant longtemps, alors que tout le reste fonctionne bien. J'ai plusieurs exemples à partager si vous voulez des exemples. Ils n'étaient pas évidents sur le moment (3 jours à chercher le bug, 2 lignes à modifier), mais maintenant je peux en rire.

E Réponses aux exercices complémentaires

E.1 Suppl. Procédural 2 : Cache associative

c) Pour cache associative : étiquette (21 bits), mots (1 bit), octets (2 bits).
e) Pour cache associative : échec, échec, réussite, échec.

E.2 Suppl. Procédural 2 : Analyse de code et temps d'exécution

232 cycles, pour le segment affiché dans le guide étudiant.
Avec le code fourni sur la page web pour le laboratoire 2, +14 pour les instructions du *main* pour un total de 246 cycles.
Vous pouvez également confirmer le résultat avec les outils d'analyse statistiques de *MARS*.

E.3 Laboratoire 2 : gain avec caches

Les durées de simulation incluent le reset assez long, requis par le modèle de simulation du buffer d'écriture de cache.

Mémoire Instructions	Mémoire Données	Temps du syscall	Temps calculé
SRAM	SRAM	2,460 ns	2,460 ns
DRAM	SRAM	27,040 ns	27,060 ns
Cache + DRAM	SRAM	3,860 ns	3,860 ns == SRAM + 1400 ns == SRAM + 14 échecs I
Cache + DRAM	Cache + DRAM	3,960 ns	== SRAM + 14 échecs I + 1 échec D

E.4 Suppl. Procédural 3 : Performance d'une organisation à pipeline avec aléas

Réponses pour h) :
- b) Sans unité d'envoi, 8750 ps.
- d) Avec unité d'envoi, branchement 4e étage, nop manuels : 8400 ps.
- e) Avec unité d'envoi, branchement 2e étage, nop manuels : 7560 ps.
- f) Avec unité d'envoi, branchement 2e étage, bulles/vidanges automatiques : 7140 ps.

E.5 Suppl. Procédural 3 : Calcul de CPI moyen - pipeline

d) 4.2362

E.6 Suppl. Procédural 3 : Analyse d'un programme sur une architecture à pipeline

a) Lalgorithme parcours un tableau x[ ] et fait la somme de ses composantes par groupe de 4. La somme de chaque groupe de 4 valeurs est réécrite dans un tableau de même dimension y[ ] pour le quelle les valeur sont nulles sauf une valeur sur 4 qui contient la somme des 4 valeurs de x qui précèdent.
b) Unicycle : 1190 cycles.
c) Multicycle : 4438 cycles.

d) Pipeline : 899 de plus que le unicycle, pour un total de 2089.
e) IF:lw ; ID:vidange ; EX:bne ; MEM:bulle ; WB:bulle.

F Modifications sur rétroactions

Proposé en	Suggestions/remarques	Modification(s)
H2023	Ajouter des figures montrant les résultats attendus pour certains laboratoires	○ En cours.
H2023	Allonger la durée de la validation	○ À voir si l'espace-temps le permet.
H2023	Relation entre les blocs des figures et le code VHDL	✓ Nouvelle figure C.1.
Tuteur		L'APP4GI est maintenant sur 4 semaines au lieu de 3. ✓ Remaniement de l'horaire des activités et des livrables. Ajustement des livrables. ○ Ajustement des barêmes du rapport et de la validation.
E2022	Capsules vidéo d'erreurs fréquentes affichées après le devoir.	○ Bonne idée ! À faire.
H2022 E2021	Améliorer le rapport, moins stimulant que la problématique. Certains éléments à mieux cerner. Plus de temps pour faire le rapport.	✓ Pour E2022 : nouveau code de référence matriciel fourni pour simplifier, uniformiser et permettre d'amorcer la rédaction du rapport plus tôt. ✓ H2023 : Inversion de l'ordre du dépôt du rapport et du code VHDL.
H2022	Informations en double	○ Ok, mais où sont-elles ? Svp me les envoyer par courriel si vous les voyez.
H2022	Ajouter un index des figures pertinentes	✗ Aucune figure ne montre tout, il y a toujours un certain niveau d'abstraction. Les figures pertinentes sont toujours fournie aux examens formatif/-sommatif/final. Leur lecture est essentielle vs le par coeur.
H2022	Laboratoires longs	○ En surveillance. Grande variabilité sur ce point selon les groupes.
H2022	Donner accès aux corrigés des procéduraux.	✗ Non pour les solutions, mais ok pour les réponses si le temps a manqué.
E2021	Retravailler la solution de formatif-Q1.	✓ Retravaillé la solution de formatif-Q1 en H2022.

E2021	Trop d'instructions pour le laboratoire 2.	✓ Le labo 2 est une révision majeure du concept de hiérarchie de la mémoire. Révisions mineures en H2022. ✓ Revue du VHDL pour E2022, plus forte cohésion avec le manuel de référence.
E2021	Reformuler les tests de cache en écriture (labo 2).	✓ Modifié en H2022.
E2021	Procéduraux très longs (idem à H2020/E2020). Mieux gérer le temps vs la quantité de matière. Allonger les procéduraux (30 min) ?	✗ Problème reconnu, mais difficile de choisir quoi enlever (également l'avis des étudiants passés). En réflexion au niveau de la session. ✓ Certains numéros de procéduraux mis en supplémentaires ou révision avec réponses fournies dans le guide. ✗ Pas de rallonge des procéduraux. ✓ H2023 : Nouvel horaire sur 4 semaines, 4 procéduraux au lieu de 3.
H2021	Ajouter certaines réponses numériques pour exercices.	✓ Nouvelle section à la fin du guide étudiant.
H2021	Améliorer les laboratoires.	✓ Ajouté des plans de vérification pour la laboratoire 1 et 2 (nouveau en H2021). ✗ Plans de vérification volontairement absent pour le laboratoire 3.
H2021	Exercices supplémentaire pour les caches et les calculs de performance.	✓ Ajustements en E2021 et H2022. Ajustement au projet/rapport E2022.
H2021	Codes entièrement en anglais pour uniformité.	○ Pas encore fait, pas prioritaire.
A2020	Voir l'impact des rétroactions sur les APP.	✓ Ce chapitre.
E2020	Fournir l'annexe de la problématique plus tôt.	✓Annexe maintenant disponible dès le début.
E2020	Matière sur la cache mal couverte.	✓ Nouveau laboratoire sur le sujet ; ✓ Remplace la matière sur l'organisation multicycle ; ✓ Nouveau numéro procédural.
H2020 E2020	Procéduraux très longs.	✓ Le retrait du multicycle a permis de dégager du temps et rebalancer les procéduraux.
H2020 E2020	Comprendre les instructions SIMD plus tôt dans l'APP.	✓ Numéro déplacé au procédural 2.

E2020	Lien entre le rapport et le cours pas clair.	✓ Retravaillé les consignes du rapport ; ✓ Vidéo sur les objectifs pédagogiques. ✓ Clarification des consignes de rapport.
H2020	Erreurs dans les fichiers code des laboratoires.	✓ Révision complétée pour laboratoires 1 et 3. ✓ Le labo 2 entièrement refait pour H2021. Retouches mineures faites pour E2021.
H2020	(problématique) Très très lourd à faire bien que les étudiants ont eu une semaine de relâche.	✓ Rééquilibrage progressif de l'APP en E2020 et H2021.

LISTE DES RÉFÉRENCES

[1] David Patterson et John Hennessy, *Computer Organization and Design, The Hardware/-Software Interface*, 5e édition, Burlington, MA : Elsevier / Morgan Kaufmann, 2014, 793 pages.

[2] Julien Rossignol, "Concepts de base de la programmation en assembleur MIPS," 20 janvier 2020.

[3] Charles Roth Jr., Lizy John, *Digital Systems Design Using VHDL*, 3e édition, Farmington Hills : MI, Cengage Learning, 2017, 592 pages.

[4] Dominic Sweetman, *See MIPS Run*, 2e édition, Burlington, MA : Elsevier / Morgan-Kaufmann, 2007, 492 pages.

[5] Gerry Kane et Joe Heinrich, *MIPS RISC architecture*, 2e édition, Englewood Cliffs, N.J. : Prentice Hall, 1992, 544 pages.

[6] Peter J. Ashenden and Jim Lewis, *The Designer's Guide to VHDL*, 3e édition, Burlington, MA : Elsevier / Morgan Kaufmann, 2008, 936 pages.

[7] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269.