Compte-rendu TP n°3, MULTI, 2019-2020

Aymeric Agon-Rambosson

Mercredi 19 février 2020

Cette semaine, on va commencer à modéliser le fonctionnement du cache L1.

Application logicielle

Question C1

La fonction main est située au début du segment seg_code.

La première instruction est donc à l'adresse 0x00400000.

La première instruction de la boucle est à l'étiquette loop, donc quatre (en effet, la est une macro assembleur, qui est étendue à deux instructions après compilation) mots plus loin, soit en 0x00400010.

Question C2

Les trois tableaux A, B et C sont situés vers le début du segment seg_data.

Au début du segment seg_data, on a l'adresse de la fonction main sur 4 octets, puis un rembourrage de 124 octets.

On a donc l'adresse de base de A à 0x01000080.

A est un tableau d'entiers, chacun occupant un mot. On a donc 80 octets occupés par le tableau A, on a ensuite un rembourrage de 48 octets.

On a donc l'adresse de base de B à 0x01000100.

B est un tableau d'entiers, chacun occupant un mot. On a donc 80 octets occupés par le tableau B, on a ensuite un rembourrage de 48 octets.

On a donc l'adresse de base de C à 0x01000180

Question C3

On admet qu'on parle d'un MIPS32 avec 5 étages de pipeline, et donc un delayed slot.

Le sw doit être exécuté à chaque itération de la boucle, pour que la sémantique souhaitée soit respectée.

Il se trouve que ce sw est exécuté que à chaque itération de boucle : ce processeur a un delayed slot, ce qui fait qu'une instruction après chaque branchement est exécutée de manière inconditionnelle. On aurait pu mettre un nop, mais placer ici le sw permet de nous épargner un cycle gaspillé.

Question C4

On a pas de cycle de gel, donc on aura besoin de 7 cycles pour exécuter ces 7 instructions, si on suppose comme la consigne un système mémoire parfait.

Fonctionnement du cache instruction

Question D1

Les caches sont à correspondance directe, donc NWAY égale 1.

On a 16 octets par ligne, on a donc besoin de log_2(16) bits pour identifier ces octets, soit 4.

Le champ BYTE fait donc 4 bits.

Pour minimiser la probabilité de miss de conflits, on s'arrange pour maximiser la distance minimale entre deux adresses devant aller dans le même emplacement.

Si on donne m le nombre d'emplacements du cache, cette distance minimale est maximale si on assigne à chaque adresse tronquée de l'offset (soit 28 bits dans notre cas) l'emplacement i, pour adresse % m = i (pour des emplacements étiquetés de 0 à m-1, naturellement).

Selon la valeur de m, le calcul de ce modulo devra faire intervenir un plus ou moins grand nombre des bits de poids faible de cette adresse tronquée. Dans notre cas m = 8, donc on aura 3 bits à regarder pour savoir où mettre ou chercher une adresse. Ces trois bits identifient exactement dans quel emplacement la donnée se trouve.

Donc SET a 3 bits.

TAG sera donc constitué des 25 bits de poids fort.

Question D2

Voilà l'état du cache d'instructions à la fin de la première itération (on donne les adresses complètes, pas les données, c'est beaucoup plus clair).

Reconstituons l'histoire de ce cache (en commençant du début de la fonction main).

Le processeur commence par demander l'adresse 0x00400000. Le cache fait miss, les mots d'adresse 0x00400000, 0x00400004, 0x00400008 et 0x0040000x sont chargés dans le cache (soient les mots qui correspondent aux instructions juste avant la boucle).

L'adresse de base de main est 0x00400000, et 0x00400000 tronqué des 4 bits de poids faible, soit 0x0040000 % 8 = 0, ce qui fait que les 4 premiers mots de main vont dans le premier set.

On aura donc SET = 0 = 0b000.

En entrant dans le boucle, le processeur demande l'adresse 0x00400010. Le cache fait miss, les mots d'adresse 0x00400010, 0x00400014, 0x00400018 et 0x0040001c sont chargés dans le cache (soient les 4 premières instructions de la boucle).

L'adresse de base de loop est 0x00400010, et 0x00040001 % 8 = 1, ce qui fait que les 4 premiers mots de loop vont dans le deuxième set.

On aura donc SET = 0 = 0b000.

En arrivant à l'instruction d'adresse 0x00400020 (add t4,t2,t3), le cache fait miss, les mots d'adresse 0x00400020, 0x00400024, 0x00400028 et 0x0040002c sont chargés dans le cache (soient les 3 dernières instructions de la boucle, et la première de print).

On n'aura plus de cache miss avant la fin de la boucle, donc voilà l'état du cache (on s'est permis de rajouter quelques colonnes pour expliciter)

TAG + SET (28b)	TAG (25b)	SET (3b)	V	WORD3	WORD2	WORD1	WORD0
0x0040000	0x0008000	0x0	1	0x0040000c	0x00400008	0x00400004	0x00400000
0x0040010	0x0008000	0x1	1	0x0040001c	0x00400018	0x00400014	0x00400010
0x0040020	0x0008000	0x2	1	0x0040002c	0x00400028	0x00400024	0x00400020
			0				
			0				
			0				
			0				

TAG + SET (28b)	TAG (25b)	SET (3b)	V	WORD3	WORD2	WORD1	WORD0
			0				

Question D3

À la fin de la 20ème itération, le cache est dans le même état, on boucle sur les mêmes instructions qui sont déjà dans le cache :

TAG + SET (28b)	TAG (25b)	SET (3 b)	V	WORD3	WORD2	WORD1	WORD0
0x0040000	0x0008000	0x0	1	0x0040000c	0x00400008	0x00400004	0x00400000
0x0040010	0x0008000	0x1	1	0x0040001c	0x00400018	0x00400014	0x00400010
0x0040020	0x0008000	0x2	1	0x0040002c	0x00400028	0x00400024	0x00400020
			0				
			0				
			0				
			0				
			0				

On ne comprend pas très bien de la consigne sur quelle section de code précise on doit calculer le taux de miss.

On va donc en calculer plusieurs.

Si on ne considère que la boucle du main, on ne fait que deux miss sur les 20 itérations, soient 2 miss sur 20 x 7 = 140 instructions, ce qui fait un taux de miss de 1.42 % (1/70).

Si on considère le main en entier, sans les instructions de l'appel de fonction tty_puts, on a 1 miss dans main, 2 dans loop, puis 2 dans print et 0 dans suicide.

Soient 5 miss pour 4 + 7 x 20 + 6 + 2 = 152 instructions, ce qui fait un taux de miss de 3,29 % (5/152).

Question D4

L'état de cache MISS_SELECT est nécessaire quand il est besoin de désigner un emplacement à vider de son contenu pour accueillir la nouvelle donnée qu'on veut faire rentrer dans le cache. Quand on n'a de toute façon qu'un seul emplacement possible, la désignation est vite faite. On a donc besoin de cet état uniquement quand le degré d'associativité est strictement supérieur à 1.

On en a pas besoin dans notre cas, on est en direct mapping.

Question D5

Noeud IDLE

A = IREQ.IUNC.IMISS
B = IREQ.IMISS.C(IUNC)
C = C(IREQ) + IREQ.C(IMISS)

On a C.A = 0 et C.B = 0

A.B = 0 aussi.

L'orthogonalité est vérifiée.

A + B = IREQ.IMISS

A + B + C = IREQ.IMISS + C(IREQ) + IREQ.C(IMISS) = C(IREQ) + IREQ.(IMISS + C(IMISS)) = C(IREQ) + IREQ = 1

La complétude est vérifiée de même.

Noeud MISS_WAIT

F = VALID.C(ERROR)
G = VALID.ERROR
H = C(VALID)

H est orthogonal avec F et G, on l'élimine.

F et G sont orthogonaux.

On a l'orthogonalité.

F + G = VALID.C(ERROR) + VALID.ERROR = VALID
H + F + G = C(VALID) + VALID = 1

La complétude est vérifiée de même.

Noeud UNC_WAIT

J = C(VALID)
K = VALID.ERROR
L = VALID.C(ERROR)

J est orthogonal avec K et L, on l'élimine.

K et L sont orthogonaux.

On a l'orthogonalité.

K + L = VALID.ERROR + VALID.C(ERROR) = VALID
K + L + J = VALID + C(VALID) = 1

La complétude est vérifiée de même.

Noeuds UNC_GO, MISS_SELECT, MISS_UPDT et ERROR

M = 1
O = 1
I = 1
F = 1

On sort de ces quatre états de manière inconditionnée vers un unique noeud dans chaque cas. On a l'orthogonalité et la complétude par construction.

Question D6

Cet automate est forcé dans l'état IDLE lors de l'activation du signal RESETN. Ce dernier a pour autre effet d'invalider tout le cache d'instructions.

Fonctionnement du cache de données

Question E1

A[0] est à l'adresse 0x01000080, comme on l'a vu plus haut. BYTE correspond aux 4 bits de poids faible, donc 0. SET correspond aux trois bits suivants, sera donc à 0 aussi. TAG sera les 25 bits de poids fort, soit 0x0020001.

B[0] est à l'adresse 0x01000100, comme on l'a vu plus haut. BYTE correspond aux 4 bits de poids faible, donc 0. SET correspond aux trois bits suivants, sera donc à 0 aussi. TAG sera les 25 bits de poids fort, soit 0x0020002.

Rappelons le déroulement du programme pour trouver les miss de données :

Lors du premier lw, on a un miss sur l'adresse 0x01000080, on charge donc dans le premier emplacement du cache de données les mots d'adresse l'adresse 0x01000080, 0x01000084, 0x01000088 et 0x0100008c.

Lors du deuxième lw, on a un miss sur l'adresse 0x01000100, on charge donc dans le premier emplacement du cache de données les mots d'adresse l'adresse 0x01000180, 0x01000184, 0x01000188 et 0x0100018c. Ce faisant, on évince les 4 mots qui s'y trouvaient déjà.

À la fin de la première itération de la boucle, le cache de données ressemble à ça.

TAG + SET (28b)	TAG (25b)	SET (3b)	V	WORD3	WORD2	WORD1	WORD0
0x0100010	0x0020001	0x0	1	0x0100010c	0x01000108	0x01000104	0x01000100
			0				
			0				
			0				
			0				
			0				
			0				
			0				

Question E2

Pour la première itération, on a un miss compulsif pour A[0], puis un miss de conflit pour B[0]

Pour les trois itérations suivantes, on a deux miss de conflits : les éléments de A et ceux de B se chassent mutuellement.

Pour la cinquième itération, on est à nouveau dans le même cas que dans la première question, sauf qu'on va se battre pour le deuxième emplacement.

Et ainsi de suite.

On a donc exactement deux miss par itération, soit en fait un muss par lw, ce qui fait un taux de miss en lecture de 100% (40/40).

Voilà l'état du cache de données à la fin de la 20ème itération. Seuls les 5 premiers emplacements sont utilisés, et sont remplis des éléments de B, ceux-ci ayant chassé les éléments de A.

TAG + SET (28b)	TAG (25b)	SET (3b)	V	WORD3	WORD2	WORD1	WORD0
0x0100010	0x0020001	0x0	1	0x0100010c	0x01000108	0x01000104	0x01000100
0x0100011	0x0020001	0x1	0	0x0100011c	0x01000118	0x01000114	0x01000110
0x0100012	0x0020001	0x2	0	0x0100012c	0x01000128	0x01000124	0x01000120
0x0100013	0x0020001	0x3	0	0x0100013c	0x01000138	0x01000134	0x01000130
0x0100014	0x0020001	0x4	0	0x0100014c	0x01000148	0x01000144	0x01000140
			0				
			0				
			0				

Question E3

Noeud IDLE

A = DREQ.C(WRITE).DMISS.DUNC
B = DREQ.C(WRITE).DMISS.C(DUNC)

C = C(DREQ) + DREQ.C(DMISS).C(WRITE)

D = DREQ.WRITE.C(DMISS)
E = DREQ.WRITE.DMISS

A et B sont orthogonaux, et A + B = DREQ.C(WRITE).DMISS

D et E sont orthogonaux, et D + E = DREQ.WRITE

A + B et D + E sont orthogonaux.

C est orthogonal avec A + B (premier terme à cause de DREQ, et deuxième à cause de DMISS).
C est orthogonal avec D + E (premier terme à cause de DREQ, et deuxième à cause de WRITE).

On a l'orthogonalité par développement.

C + D + E = C(DREQ) + DREQ.C(DMISS).C(WRITE) + DREQ.WRITE
C + D + E = C(DREQ) + DREQ.(C(DMISS).C(WRITE) + WRITE)

A + B + C + D + E = DREQ.C(WRITE).DMISS + C(DREQ) + DREQ.(C(DMISS).C(WRITE) + WRITE)
A + B + C + D + E = C(DREQ) + DREQ.(C(DMISS).C(WRITE) + WRITE + C(WRITE).DMISS)
A + B + C + D + E = C(DREQ) + DREQ.(WRITE + C(WRITE).(C(DMISS) + DMISS))
A + B + C + D + E = C(DREQ) + DREQ.(WRITE + C(WRITE).(1))
A + B + C + D + E = C(DREQ) + DREQ.(1)
A + B + C + D + E = 1

On a la complétude de même.

Noeud MISS_WAIT

F = VALID.C(ERROR)
G = VALID.ERROR
H = C(VALID)

On a H orthogonal avec F et G, et F et G orthogonaux entre eux.

On a donc bien l'orthogonalité.

F + G = VALID.C(ERROR) + VALID.ERROR = VALID
F + G + H = VALID + C(VALID) = 1

On a la complétude de même.

Noeud UNC_WAIT

J = C(VALID)
K = VALID.ERROR
L = VALID.C(ERROR)

On a J orthogonal avec K et L, et K et L orthogonaux entre eux.

On a donc bien l'orthogonalité.

K + L = VALID.C(ERROR) + VALID.ERROR = VALID
K + L + J = VALID + C(VALID) = 1

On a la complétude de même.

Noeuds WRITE_UPDT, UNC_GO, ERROR, MISS_SELECT, MISS_UPDT

On quitte ces états de manière inconditionnée, à chaque fois vers un autre état unique.

On a donc :

M = 1
P = 1
I = 1
N = 1

Sur ces quatre noeuds, on a l'orthogonalité et la complétude de manière évidente.

Question E4

Comme le dit la consigne, les états successeurs de l'état WRITE_REQ sont les mêmes que les états successeurs de l'état IDLE.

La différence est l'évaluation du signal WOK. On ne sort de cet état que si le tampon d'écritures postées n'est pas plein.

Donc la boucle de WRITE_REQ vers WRITE_REQ (nommons la E') a comme condition C(WOK) + WOK.E (on reste soit si le tampon d'écriture postée est plein, soit pour les mêmes raisons qu'on va de IDLE vers WRITE_REQ, en vérifiant que le tampon d'écritures postées n'est pas plein).

Toutes les autres transitions de WRITE_REQ, soient A' (vers UNC_WAIT), B' (vers MISS_SELECT), C' (vers IDLE) et D' (vers WRITE_UPDT) pourront s'écrire :

A' = WOK.A
B' = WOK.B
C' = WOK.C
D' = WOK.D

En développant, on garde bien la complétude et l'orthogonalité.

Accès au PIBUS

Question F1

Ici, on peut répondre différemment selon les hypothèses qu'on fait sur le tampon d'écritures postées.

Quand on fait une lecture de données, prend-on la peine de vérifier le tampon d'écritures postéées, pour voir si par hasard la dernière version de la donnée ne s'y trouverait pas ?

Si non, alors ne pas rendre les écritures prioritaires sur le bus revient à immédiatement violer la consistance mémoire. On est obligé de rendre les écritures absolument prioritaires, simplement pour garantir la consistance.

Si en revanche on suppose que cette vérification est faite, on peut trouver aussi des raisons certes un peu plus faibles :

Le tampon d'écritures postées est de petite taille par rapport au cache (conséquent nécessaire de la vérification du tampon d'écritures postées !), il se remplit donc plus vite. Pour cette raison, on pourrait avoir envie de le rendre prioritaire.

Les programmes ont aussi tendance à faire les lw avant les sw, ce qui fait qu'un programme a déjà du matériau en réserve pour travailler quand le tampon d'écritures se met à monopoliser le bus. On espère que le prochain miss arrivera le plus tard possible après le début de cette monopolisation. D'une certaine manière, les effets vicieux de cette monopolisation sont un peu mitigés par l'ordre "naturel" des programmes.

L'inconvénient est évident, ce sont les famines potentielles pour les caches d'instructions et de données qui veulent accéder au bus. Ces famines sont ceci étant dit limitées d'elles-même. Si on ne peut plus chercher des instructions, le tampon d'écriture postées va se vider très vite, puisque rien ne peut plus le venir remplir.

Question F2

Comme pour tous les maîtres, d'après le schéma général du PIBUS, ICACHE_ISM et DCACHE_ISM demanderont le bus à l'aide du signal REQ.

Comme pour tous les maîtres, le maître répondra dans le même cycle avec le signal GNT.

Ici, on considère le PIBUS_FSM comme une espèce de serveur. Mais la question fait ici une supposition erronée, ce n'est pas le "serveur" (soit le PIBUS) qui signale au "client" que les données sont disponibles, c'est la cible directement (c'est la limite de cette analogie : le "serveur" est en fait plus un routeur !) ! La cible signale au maître que les données sont disponibles avec le signal ACK.

Question F3

Sur une requête d'écriture transmise sur le bus par le tampon d'écritures postées, il y a effectivement une réponse ! Celle-ci est transmise par la cible, via le signal ACK. Cette réponse est vitale, elle sert à savoir si la requête était valide. Si elle ne l'était pas, il faut bien le dire au processeur pour qu'il puisse déclencher l'interruption qui aboutira très probablement à la mort du processus fautif.

Pour répondre à la question, le PIBUS_FSM ne signale pas qu'une requête est terminée, parce que ce n'est pas son rôle ! C'est à la cible de le faire.

Question F4

Noeud IDLE

X = GNT.(ROK + SC)
Y = GNT.C(ROK).C(SC).(IUNC + IMISS + DUNC + DMISS)
Z = C(GNT)

X et Y sont orthogonaux.
Z est orthogonal avec X et Y.

On a donc l'orthogonalité.

X + Y = GNT.(ROK + SC) + GNT.C(ROK).C(SC).(IUNC + IMISS + DUNC + DMISS)
X + Y = GNT.(ROK + SC + C(ROK).C(SC).(IUNC + IMISS + DUNC + DMISS))

X + Y = GNT.(ROK.(IUNC + IMISS + DUNC + DMISS) +
ROK.(C(IUNC).C(IMISS).C(DUNC).C(DMISS)) +
SC.(IUNC + IMISS + DUNC + DMISS) +
SC.(C(IUNC).C(IMISS).C(DUNC).C(DMISS)) +
C(ROK).C(SC).(IUNC + IMISS + DUNC + DMISS))

X + Y = GNT.((IUNC + IMISS + DUNC + DMISS).(ROK + SC + C(ROK).C(SC)) +
ROK.(C(IUNC).C(IMISS).C(DUNC).C(DMISS)) +
SC.(C(IUNC).C(IMISS).C(DUNC).C(DMISS)))

X + Y = GNT.((IUNC + IMISS + DUNC + DMISS).(1) +
ROK.(C(IUNC).C(IMISS).C(DUNC).C(DMISS)) +
SC.(C(IUNC).C(IMISS).C(DUNC).C(DMISS)))

X + Y = GNT.((IUNC + IMISS + DUNC + DMISS) + (ROK + SC).(C(IUNC).C(IMISS).C(DUNC).C(DMISS)))

IUNC et ROK peuvent être considérés comme des événements disjoints : quand ROK est positionné, on ne tient aucun compte de IUNC, c'est comme si ce signal n'existait pas (c'est le sens de la priorité fixe).

Donc si ROK et IUNC sont des événements disjoints, on a ROK.C(IUNC) = ROK.

De même avec SC...

On peut donc simplifier X + Y comme ceci :

X + Y = GNT.(IUNC + IMISS + DUNC + DMISS + ROK + SC)

Les événements IUNC, IMISS, DUNC, DMISS, ROK, SC sont les seuls possibles dans le modèle qu'on s'est donné. Ils constituent l'univers, on a donc

IUNC + IMISS + DUNC + DMISS + ROK + SC = 1

Soit X + Y = GNT.

Donc X + Y + Z = C(GNT) + GNT = 1.

On a bien la complétude.

Question F5

Voilà comment le cache se comporte, si on suppose que le instruction fetch du MIPS termine sans instruction chargée à la fin de l'étage I au cycle n.

Cycle n : Le processeur demande une instruction, il ne l'a pas à la fin de son Instruction Fetch, le cache est informé à la fin de ce cycle, le cache est en position MISS_SELECT et l'emplacement à vider est choisi.
Cycle n + 1 : Le cache se met en position MISS_WAIT. Le cache demande le bus, il l'obtient à la fin du cycle si tout se passe bien.

Cycle n + 2 : Le cache envoie la première adresse de la rafale.
Cycle n + 3 : Le cache reçoit la première donnée, et envoie la deuxième adresse.
Cycle n + 4 : Le cache reçoit la deuxième donnée, et envoie la troisième adresse.
Cycle n + 5 : Le cache reçoit la troisième donnée, et envoie la quatrième adresse.
Cycle n + 6 : Le cache reçoit la quatrième donnée.
Cycle n + 7 : ?
Cycle n + 8 : Le cache met à jour l'emplacement qu'il avait choisi au cycle n + 1.
Cycle n + 9 : Le cache envoie l'instruction au processeur, qui la reçoit à la fin du cycle.
Cycle n + 10 : Le processeur fait rentrer l'instruction dans son pipeline.

Le processus est sensiblement le même pour le miss de données.

Le processeur est gelé des cycles n + 1 à n + 9 compris, soient 9 cycles de gel.

Le cycle n + 7 correspond à un cycle qu'on a constaté dans la trace, mais pour lequel on n'a pas vraiment d'explication pour le moment.

On trouvera le chronogramme en annexe.

Question F6

On a donc 9 cycles de gel par miss d'instruction ou de données.

Sur 20 itérations, on a 2 miss d'instructions, soient 18 cycles de gel liés aux miss d'instructions.

Sur 20 itérations, on a 40 miss de données, soient 40 x 9 = 360 cycles de gel liés aux miss de données.

Soient 378 cycles de gel liés aux miss toutes catégories confondues.

On a 140 instructions, un cycle par instruction.

Ce qui fait un CPI théorique de 518 / 140 = 3.7.

Expérimentation par simulation

Question G1

Si on numérote les cycles à compter du cycle 0 (comme la trace), la première instruction rentre dans le pipeline au cycle 10.

Cette instruction est la première instruction de reset, la partie du code du GIET devant être exécutée à la mise sous tension de la machine.

Si on excepte le premier miss, le coût d'un miss sur le cache d'instructions est de 9 cycles.

Question G2

D'après la trace, la première instruction du main, d'adresse 0x00400000, rentre dans le pipeline au cycle 57.

Question G3

Un miss sur le cache de données occasionne 9 cycles de gel pendant la première itération, puis 13 sur la prochaine et la suivante.

Le processeur fait rentrer la dernière instruction de la première itération dans son pipeline au cycle 108, sachant qu'il avait fait rentrer la première au cycle 71, ce qui donne 37 cycles.

Question G4

La seconde itération et la suivante durent 31 cycles, soit moins que la première. Cette état de chose s'explique par les miss compulsifs sur les instructions lors de la première instruction.

Ceci étant dit, le coût du miss de données est plus élevé : il occasionne 9 cycles de gel à la première itération, et 13 aux deuxième et troisième.

Ceci s'explique par l'entrée en jeu du tampon d'écritures postées, qui commence à utiliser de manière prioritaire le bus à partir de la toute fin de la première itération.

Question G5

On a en tout 40 lw dans les 20 itérations, on fait miss sur chacune d'entre elle, soit un taux de miss de 100 %.

La dernière instruction de la dernière itération de main entre dans le pipeline au cycle 696, et la première instruction de la première itération entre dans le pipeline au cycle 71.

On a donc 625 cycles pour toute la boucle, soit un CPI constaté de 4.46 (la différence avec le CPI théorique s'explique par la non prise en compte du tampon d'écriture postée).

Optimisation

Dans la manière dont est codée cette application, on s'est arrangé pour maximiser le nombre de miss sur le cache de données en maximisant les miss de conflits. Pour ce faire, les adresses des données nécessaires au même moment ont été exprès alignées sur des blocs de 128 octets, ce qui représente littéralement le pire agencement possible (taux de miss de 100 %).

Une manière simple de grandement optimiser ce programme et de briser cet alignement en supprimant le rembourrage de 48 octets à la fin du premier tableau.

Comme ceci :

```
################################################################################
# File : main.s
# Author : Aymeric Agon-Rambosson
#       Date : 16/02/2020
################################################################################
#       This is a very simple application directly written in MIPS32
# assembly language, in order to precisely control the memory mapping.
# The sections names are specific to control the linker.
################################################################################

  .section    .mydata

  .word   main
  .space  124

A :   .word     1,  2,  3,  4,  5,  6,  7,  8,  9, 10
  .word    11, 12, 13, 14, 15, 16, 17, 18, 19, 20

B :   .word   101,102,103,104,105,106,107,108,109,110
  .word   111,112,113,114,115,116,117,118,119,120
  .space 96

C :   .word     0,  0,  0,  0,  0,  0,  0,  0,  0,  0
  .word     0,  0,  0,  0,  0,  0,  0,  0,  0,  0

  .section    .mycode

  .set noreorder
main :
  la   $8,    A       # $8 <= &A[0]
  li   $7,    20          # $7 <= 20
```

```
  li   $6,    0       # $7 <= 0

loop :
  lw   $10,   0($8)       # $10 <= A[i]
  lw   $11,   80($8)      # $11 <= B[i]
  addi $6,    $6,     1   # i <= i+1
  addi $8,    $8,     4   # $8 <= &A[i+1]
  add  $12,   $10,    $11 # $12 <= A[i]+B[i]
  bne  $6,    $7,     loop    # fin de boucle ?
  sw   $12,   252($8)         # C[i] <= $12
print:
  la   $4, message
  addi $29,       $29,     -4
  jal  tty_puts
  nop
  addi $29,       $29,     +4
suicide:
  jal  exit
  nop
message:
  .asciiz   "\n!!! vector sum completed !!!\n"
```

Avec cette nouvelle organisation de la mémoire, on aura seulement 10 miss compulsifs sur le cache de données sur la totalité de la boucle, et aucun miss de conflit : on n'évince une ligne qu'après avoir fini d'en avoir besoin.

On a 30 cache miss en moins, soient un peu plus de 9 x 30 = 270 cycles de gel en moins (tous les cache miss ne coûtent pas 9 cycles, comme on l'a vu).

Le nombre de cycles pour l'exécution totale du programme sera donc un peu supérieur à 518 - 270 = 248 cycles, soient environ 12.4 cycles par itération, soit un CPI théorique de 1.77.

Le plus simple est encore de simuler.

Après réassemblage, on a la dernière instruction de la dernière itération de la boucle qui rentre dans le pipeline au cycle 336.

La première instruction de la première itération de la boucle était rentrée dans le pipeline au cycle 71.

Ce qui nous fait 336 - 71 = 265 cycles, ce qui est assez proche de ce qu'on avait prévu.

On a 265 cycles pour 140 instructions, soit un CPI constaté de 1.9, ce qui est bien mieux (la différence s'explique encore une fois par le tampon d'écriture postée, mais elle est plus faible que dans l'exécution non-optimisée parce qu'on a volontairement sous-estimé l'effet de nos optimisations en prenant un coût en cycles de gel moyen de 9 alors qu'il est en fait plus proche de 13).