PROJET ARCHITECTURE et SYSTEME

Claire Hanen et Marie-Pierre Gervais

Objectif général du projet

On considère un langage d'assemblage pour une architecture de type RISC pipelinée à 32 registres et mots mémoire de 32 bits. Le jeu d'instruction comprend les instructions suivantes :

syntaxe	**effet**	**commentaire**
LDIR Rd Imm	Rd←Mem[Imm]	Chargement d'un mot mémoire dans un registre par adressage direct
SDIR Rd Imm	Mem[Imm] ← Rd	Rangement d'un mot mémoire dans un registre par adressage direct
IADD Rd Ra Rb	Rd←Ra+Rb	Addition entière. Registre condition positionné
ISUB Rd Ra Rb	Rd←Ra-Rb	Soustraction entière. Registre condition positionné
COMP Ra Rb	Ra-Rb	Comparaison, RCC positionné
IMUL Rd Ra Rb	Rd←Ra*Rb	Multiplication de deux entiers, RCC positionné
Bccc dep	Si ccc PC←PC+ ES(dep)	Branchement conditionnel
JUMP dep	PC←PC+ ES(dep)	Branchement inconditionnel
NOOP		Ne fait rien
SETI Rd Imm	Rd←ES(Imm)	Affecte une valeur immédiate entière à un registre.

La micro-architecture de la machine est pipelinée, mais ne dispose pas de moyens très efficaces d'anticipation. En cas de dépendance, les instructions sont bloquées jusqu'à ce que le résultat puisse être utilisé. De plus le pipeline est de différente longueur selon la nature de l'instruction.

Pour cette raison, le compilateur a intérêt à réorganiser le code de sorte que deux instructions dépendantes soient suffisamment éloignées l'une de l'autre, quitte à insérer quelques instructions NOOP.

Par ailleurs un algorithme de gestion de mémoire virtuelle est mis en place à l'exécution pour gérer l'espace mémoire.

TRAVAIL À RÉALISER

Ce projet à pour but d'écrire un simulateur permettant, étant donnés quelques paramètres de l'architecture (latence minimale entre deux opérations dépendantes selon leur type, durée de chaque type d'instruction, taille des pages et autres paramètres de la mémoire virtuelle), le réordonnancement d'un code et la simulation de son exécution, afin de mesurer le nombre de cycles utilisés.

Organisation du projet

Le projet est divisé en trois parties.

1) **Génération du graphe des dépendances**
2) **Réordonnancement des instructions à partir du graphe des dépendances**
3) **Simulation de l'exécution avec gestion de la mémoire virtuelle**

Deux binômes seront regroupés pour réaliser l'un 1) et 3) l'autre 2) et 3) et permettre ensuite le regroupement des codes. Il faudra, bien entendu, que les groupes de binômes se mettent d'accord sur les structures de données employées.

Important : Pour simplifier le projet, les points 1) et 2) seront traités pour des ***programmes ne comportant pas d'instruction de branchement***. On vous demande en complément de réfléchir sur papier à une solution de ce même problème lorsqu'il y a des branchements correspondant à des alternatives et des boucles d'un langage structuré.

Éléments fournis

Vous disposez en entrée d'un fichier texte qui inclut plusieurs jeux d'essai séparés par le caractère #
Un jeu d'essai inclut :

- Un numéro d'identification
- La taille de la mémoire physique (donc le nombre de cases de la mémoire physique sera une variable dans votre programme)
- L'adresse virtuelle de la première instruction
- La suite des instructions.

NB : dans la mesure où le langage cible ne comporte que de l'adressage direct pour les accès mémoire, les adresses virtuelles des données seront exprimées directement dans le texte du programme à l'aide de nombres entiers.

Utilisation du fichier fourni

Ce fichier s'utilise aussi bien dans la partie 1 (génération de graphe) que dans la partie 3 (gestion de mémoire virtuelle) du projet.
Vous devez écrire une fonction qui permet de lire un jeu d'essai et de le ranger dans une structure qui sera utilisée comme base dans les parties 1 et 3.

Cette structure comprend :

- Le numéro d'identification du jeu,
- La taille de la mémoire,
- L'adresse virtuelle de la première instruction, et
- Une liste simplement chaînée dont les maillons portent les champs suivants :
    - ♦ Identifiant (numéro)
    - ♦ Instruction : texte de l'instruction
    - ♦ Virtuelle : le cas échéant, valeur entière de l'adresse virtuelle de la donnée (pour un LDIR ou SDIR)
    - ♦ Deppred : pointeur sur une liste de dépendances (voir plus loin)
    - ♦ Depsucc : id
    - ♦ Suivant : pointeur sur le maillon suivant.

    Le type d'un maillon sera nommé **sommet**
    Dans un premier temps, les champs Deppred et Depsucc seront mis à 0.

Partie 1 : Génération du graphe des dépendances

On se donne plusieurs tableaux résumant les informations sur les différentes instructions dont on donne le détail plus loin.

Les dépendances

On distingue dans le code trois types de dépendance :

La dépendance directe. Une instruction produit un résultat dans un registre qui est utilisé comme donnée pour une autre instruction :

Exemple : LDIR R01 1051
IADD R02 R01 R05

NB : l'instruction qui utilise R01 n'est pas nécessairement placée juste en dessous dans le code.
Ainsi, il est nécessaire que, sans mécanisme d'anticipation, l'écriture de R01 se fasse avant la lecture de R1 par IADD, et même avec de tels mécanismes, il y a une latence minimum à respecter entre les deux instructions. Si par exemple la latence est 3, on pourra lancer le LDIR au cycle t et l'IADD à t+3 ou plus tard.

On dispose d'un tableau qui résume les latences inter-opérations pour les dépendances directes :

DIRECT	SDIR	IADD, ISUB	IMUL
LDIR	10	8	8
IADD,ISUB	5	3	5
IMUL	7	5	7
SETI	7	7	7

L'anti-dépendance. Une instruction lit un registre qui est écrit par une autre un peu plus loin. En cas de réordonnancement des instructions, il est bien évident que la deuxième instruction ne saurait écrire sur R1 avant la première le lise

IADD R02 R01 R05
LDIR R01 1051

Dans ce cas, le load ne peut commencer que quelques cycles avant éventuellement, si l'écriture se fait par exemple au cycle 7 de l'instruction et la lecture au cycle 3, alors on pourra lancer l'instruction LDIR au cycle t puis IADD au cycle t+3, ainsi la lecture de R01 par IADD se fera à t+6 et l'écriture par LDIR à t+7. On aura alors –3 comme valeur dans la table des anti-dépendances.

ANTIDEP	LDIR	IADD, ISUB	IMUL	SETI
SDIR	-3	-4	-6	-4
IADD,ISUB	-1	-3	-5	-3
IMUL	-5	-4	-5	-4

La dépendance en assignation. Une instruction écrit un registre écrit par un autre un peu plus loin. Là, il s'agit, de conserver l'ordre des écritures, même si les opérations qui en dépendent sont indépendantes et peuvent être réordonnées à loisir. D'autres techniques existent, qui consistent pour l'essentiel à renommer les registres, et qui sortent du cadre de ce projet.

IADD R01 R02 R05
…..
LDIR R01 1051

NB : dans le cas le plus fréquent, entre les deux écritures il y a une lecture et donc en fait une dépendance directe suivie d'une anti-dépendance. Cela n'est pas le cas lorsque par exemple l'opération IADD est juste utilisée pour positionner le RCC.
Dans tous ces cas, en admettant que l'écriture d'un registre par une opération se fait toujours dans son dernier cycle, il s'agit de s'assurer que dans le nouveau code, la première écriture finisse avant la seconde, c'est à dire que si la première instruction dure d1 et la seconde d2, dans tous les codes produits si la première démarre au cycle t, la seconde démarre au plus tôt au cycle t+d1-d2+1

On donne donc une table des durées des différentes instructions.
Tableau des durées :

Instruction	Durée
Accès mémoire (SDIR,LDIR)	10
IADD , ISUB	7
IMULT	10
SETI	8

Génération du graphe

Le but de la génération du graphe des dépendances est de compléter la structure de données des « **sommets** » en ajoutant les informations relatives aux dépendances. Chaque sommet x comporte et deux listes d' « **arcs** » de dépendance qui lui correspondent (les arcs sortants Depsucc et les arcs entrants Deppred).
La liste des arcs sortant est une liste de dépendances décrites plus loin dont l'instruction de x est l'origine, de même la liste des arcs entrants est une liste des pointeurs sur des dépendances dont l'instruction x est l'extrémité.

Chaque dépendance contient les informations suivantes, calculées à l'aide des tables et de l'analyse du code :
- Type de la dépendance (directe, anti-dépendance, en assignation).
- Nom du registre concerné.
- Pointeur sur le sommet x origine de la dépendance.
- Pointeur sur le sommet y correspondant à l'instruction qui dépend de celle que l'on traite. (y dépend de x)
- Valeur de latence estimée, lat(x,y), de sorte que si l'instruction x est lancée au cycle t, vis à vis de cette dépendance, l'instruction y sera lancée à un cycle supérieur ou égal à t+lat(x,y).
  Dans certains cas cette latence est négative.

Indication :
Pour générer cette structure, il faut parcourir la liste des sommets déjà générée du programme en analysant son contenu de l'instruction et compléter les dépendances des sommets déjà parcourus. Pour cela, il est utile de mémoriser, pour chacun des 32 registres, sous forme de pointeur, le dernier sommet rencontré qui l'a lu, ainsi que le dernier sommet rencontré qui l'a écrit.

Résultats attendus

1) Un dossier d'analyse incluant la description des structures de données utilisées et une présentation en français de(s) l'algorithme(s) programmé(s).

2) Pour vérifier vos résultats, on vous demande une sortie fichier du graphe de dépendance avec la structure suivante :
- Nombre de sommets
- Pour chaque sommet
- Identificateur
- Instruction
- Nombre de dépendances succ
- Pour chacune
- sommet extrémité / latence/ type (D/N/A)
- Nombre de dépendances pred
- Pour chacune
- Sommet origine / latence /type

Partie 2 : Réordonnancement des instructions

Dans cette partie, on suppose que l'on dispose d'une structure de graphe de dépendance correspondant à un code initial avec des paramètres particuliers. Il s'agit de produire sous forme d'une liste chaînée de sommets un nouveau code réordonnancé de sorte que toutes les latences associées aux dépendances soient respectées, en insérant éventuellement au besoin des instructions NOP si nécessaire, de sorte qu'une instruction de ce nouveau programme puisse être lancée à chaque cycle en produisant exactement les mêmes résultats que le programme initial.

La solution algorithmique permettant d'optimiser la durée totale d'exécution (c'est à dire le nombre d'instructions NOP ajoutées) est laissée à votre initiative.

Résultats attendus

1) Un dossier d'analyse incluant la description des structures de données utilisées et une présentation en français de(s) l'algorithme(s) programmé(s), ainsi que quelques éléments de justification.
2) Pour cette partie, dans la mesure où les jeux d'essai finaux ne seront pas prêts, puisque les personnes qui traitent la génération du graphe n'ont pas achevé de traiter leur programme, on vous propose de travailler sur un fichier intermédiaire correspondant à la sortie de la partie 1) et représentant un graphe. Il faudra donc écrire une fonction permettant de générer le graphe à partir d'un tel fichier. La sortie attendue est aussi à faire sous forme fichier : il s'agira de reconstituer un jeu d'essai valide pour la partie 3.

Partie 3 : Gestion de la mémoire virtuelle et simulation.

Dans cette partie, on suppose que l'on dispose d'une mémoire virtuelle paginée. L'espace d'adressage d'un utilisateur est découpé en pages et la mémoire principale en cases, chaque case capable de contenir une page. Les pages et les cases ont la même taille. L'unité d'une page ou d'une case est un mot de 32 bits. Pages et cases ont la même taille qui est de 1024 mots.
L'espace virtuel de l'utilisateur contient à la fois des instructions et des données. Chaque instruction avec ses arguments (cf. le jeu d'instructions) occupe un mot.

Objectif du travail à réaliser

Écrire un programme de simulation du fonctionnement d'un gestionnaire de mémoire virtuelle paginée (monoprogrammation).
Ce gestionnaire doit fonctionner quels que soient les jeux d'essai fournis et pour un nombre quelconque de jeux d'essai. Il exécute séquentiellement les jeux d'essai en bouclant. C'est l'utilisateur du programme de simulation qui décide de l'arrêt de la simulation, qui ne peut s'arrêter que si tous les jeux d'essai de la simulation ont été traités au moins une fois.

Hypothèses de travail

- Le système au sein duquel ce gestionnaire est mis en œuvre est monoprogrammation.
- Au début de l'exécution d'un jeu d'essai, toutes les cases mémoire sont libres.
- Pour chaque jeu d'essai, deux algorithmes de remplacement de pages sont appliqués: FIFO et l'algorithme de la 2^{e} chance.
- Les adresses virtuelles sont en format `int`. Le gestionnaire doit donc prendre en charge la traduction en format (numéro de page, déplacement) des adresses.
- Un chargement de page introduit un délai et gel des instructions en cours de 10 cycles.
- Un déchargement de pages introduit un délai et gel des instructions en cours de 10 cycles.
- Par définition, un défaut de pages est constitué d'un déchargement suivi d'un chargement ! !

Résultats attendus

1) Un dossier d'analyse incluant la description des structures de données utilisées et une présentation en français de l'algorithme programmé.

2) Une trace du fonctionnement du gestionnaire (quand une simulation a été exécutée) qui se présentera de la façon suivante, pour chaque jeu d'essai faisant partie de cette simulation :

Numéro du jeu d'essai :
Temps total d'exécution :
Algorithme appliqué :
Nombre total de défauts de page :

Type d'action	numéro de la page concernée	numéro de la case concernée
xxxxxx	xx	xx
yyyyyyy	yy	yy

Les types d'action sont :
- Chargement de page. Dans ce cas, le numéro de page est celui de la page chargée, le numéro de case est celle de la case occupée par la page chargée.
- Défaut de page. Dans ce cas, le numéro de page est celui de la page volée, le numéro de case est celle de la case libérée par la page volée.

3) Une statistique sur les défauts de page rencontrés. Pour chaque jeu d'essai, le nombre de défauts de page sera indiqué. Lors de l'arrêt d'une simulation, la moyenne des défauts de page pour l'ensemble des jeux d'essai de la simulation sera affichée.