Compilation avancée – Projet N°2 – Rapport

Vincent Havinh

William Sergeant

# Objectif du projet

Notre objectif pour ce projet est d’implémenter un ensemble de fonctions permettant de traiter du code assembleur. Ainsi, nous structurerons le code en délimitant les blocs de bases puis nous analyserons les dépendances entre les instructions. Enfin, nous apporterons des améliorations au code source telle que l’optimisation de l’ordonnancement des instructions et le renommage des registres afin de limiter les dépendances.

# Fonctions implémentées

## compute\_basic\_bloc

Afin de délimiter les blocs de bases, nous parcourons chaque ligne du code correspondant à une instruction et analysons son type :

Si c’est une directive alors elle est ignorée et nous passons à l’instruction suivante

Si c’est un label il s’agit du début d’un nouveau bloc de base (en considérant que tous les labels du code sont utilisés). Ainsi on ferme le bloc de base courant en lui donnant pour fin la ligne précédente et un nouveau est ouvert.

Si c’est une instruction et qu’elle est du type branchement alors c’est la fin du bloc courant, le delayed slot suivant le saut est ajouté et le bloc de base est fermé.

## compute\_CFG

La fonction compute\_CFG étant dors-et-déjà implémentée dans le code du programme de test, nous n’avons pas effectué de nouveau cette implémentation.

## compute\_succ\_pred

Cette fonction sert à définir les liens de succession et précédence entre les blocs de base. Nous parcourons ainsi tous les blocs définis grâce à la fonction compute\_basic\_bloc et calculons ses successeurs potentiels.

Afin de savoir si celui-ci en possède un, deux ou aucun nous analysons sa dernière instruction :

* Si c’est un saut inconditionnel, un appel de fonction ou qu’il n’y a simplement pas de saut, il aura un successeur qui sera le bloc suivant.
* Si c’est un saut conditionnel alors il aura deux successeurs : la cible du saut si la condition est respectée, le bloc suivant sinon.
* Si c’est un saut indirect ou qu’il s’agit du dernier bloc de base du programme (la dernière instruction étant JR R31), alors ils n’ont pas de successeur.

## compute\_dom

Nous utilisons l’algorithme vu en cours pour calculer les relations de dominance entre les blocs de bases. Nous itérons sur ceux-ci tant qu’un changement de dominance à lieu dans les prédécesseurs d’un bloc.

A noter que la racine se domine elle-même.

## comput\_pred\_succ\_dep

Nous cherchons ici à définir les dépendances entre deux instructions au sein d’un même bloc de base.

Des prédicats servant à tester une dépendance donnée entre deux instructions nous permet de déterminer aisément s’il en existe et le cas échéant, de trouver son type. Nous pouvons ainsi dresser la table des dépendances RAW1-2, WAR1-2 et MEM au sein du bloc.

En outre il faut ajouter les dépendances de contrôles avec les branchements s’ils sont présents.

La subtilité en ce point est de faire un bon usage des dépendances WAW, elles sont utilisées afin de stopper la recherche de dépendance avec l’instruction courante. En effet dans le cas d’une telle réécriture c’est la dernière instruction qui sera considérée pour les futures recherches de dépendance par rapport au registre destination.

## nb\_cycles

Le but est à présent de calculer le nombre de cycles nécessaire à l’éxécution d’un bloc de base. On s’interesse alors au nombre de cycle de chaque instruction constituant le bloc.

A cet effet nous utiliserons la formule suivante :

Soit k l’instruction courante, soit p l’instruction prédécesseur de k

cycle(k) = max(cycle(k-1)+1, max(cycle(p)+delai(p,k))

Le tableau de délai entre les instructions nous a été fourni en annexe.

L’enjeu est de considérer les phases d’attente active des instructions soumises à dépendance avec d’autre (les cycles de gel).

## compute\_use\_def

Afin de connaître les registres vivant en différents points du programme, nous calculons les domaines DEF et USE de chaque bloc de base. Lorsqu’un registre est utilisé comme destination d’une instruction, il appartient au domaine DEF du bloc. Aussi, si un registre apparaît comme source d’une instruction, il appartiendra au domaine USE seulement s’il n’a pas été défini par une instruction antérieure dans le bloc.

En outre certaines spécificités apparaissent lors de l’appel de fonction :

Le registre $2 contiendra le résultat de retour de la fonction (toutes les fonctions retournant un résultat), les registres $4,$5,$6,$7 contiennent les paramètres de l’appel de la fonction (les quatre ne sont pas nécessairement utilisés). De plus le registre $31 (adresse de retour de la fonction) sera également vivant en sortie d’appel. Ces registres ne sont pas explicitement utilisés dans les instructions mais doivent être pris en compte afin de correctement définir les domaines DEF et USE.

## compute\_live\_var

Cette fonction utilise les domaines DEF et USE fournis par la fonction précédente afin d’analyser la vivacité des registres en entrée et sortie de bloc.

Un bloc aura comme liste de registres vivants Live-Out tous les registres présents dans la liste Live-In de son successeur. Si un bloc ne possède pas de successeur mais qu’il s’agit du bloc de fin du main (terminant par un saut indirect) ou du bloc de sortie d’une fonction, les registres $2 (contenant la valeur de retour de la fonction) et le registre $29 (pointeur de pile) seront vivants en sortie de bloc et doivent être pris en compte.

La liste Live-In d’un bloc est construite en y ajoutant :

* Les registres présents dans son Live-Out à l’exception de ceux définis dans le bloc
* Les registres appartenant au domaine USE du bloc.

On itère cette méthode sur tous les blocs de bases en commençant par ceux sans successeurs.

Le cas particuliers des arcs retour nous contraint à effectuer plusieurs passes afin de propager les changements des Live-In et Live-Out dans tous les blocs. L’itération s’arrête après une ultime passe où aucun changement n’est constaté.

## compute\_def\_liveout

Dans le cas où un registre est vivant en sortie de bloc et que ce registre appartient au domaine DEF du bloc, nous cherchons à savoir quelle est la dernière instruction à l’avoir défini.

Ainsi, après avoir repérer les registres remplissant ces deux conditions, nous parcourons les instructions et on note celle qui définit le registre concerné. Si l’on rencontre une instruction le définissant à nouveau, l’information est mise à jour. Alors après une passe c’est bien la dernière instruction ayant défini le registre qui est retenue.

## reg\_rename

Cette fonction de re-nommage concerne les registres définis puis utilisés dans le bloc mais non vivants en sortie.

Nous cherchons d’abord à définir quels registres sont re-nommables :

Pour cela une liste est construite contenant les registres absent du Live-In/Live-Out du bloc mais présent dans le domaine DEF du bloc et utilisés par celui-ci.

Ensuite, nous itérons sur cette liste en repérant la première définition du registre à renommer puis nous effectuons le changement de nom dans celle-ci. Enfin nous parcourons le reste des instructions en renommant les occurrences ultérieures des utilisations du registre jusqu’à atteindre la fin du bloc ou une nouvelle définition du registre.

Analyses expérimentales Soit le bloc de base suivant et le nombre de cycle nécessaire à son exécution (on utilise la table de délais initiale):

main:

**i0 lw $4,0($6)**

**i1 lw $2,0($4)**

**i2 add $5,$14,$2**

**i3 ori $10,$6,0**

**i4 sw $5,0($10)**

**i5 lw $2,65524($10)**

**i6 addi $5,$2,4**

**i7 bne $5,$2,$l5**

**i8 add $0,$0,$0**

|  |  |  |
| --- | --- | --- |
| Instruction | Code | Cycle de sortie |
| 0 | lw $4,0($6) | 1 |
| 1 | lw $2,0($4) | 3 |
| 2 | add $5,$14,$2 | 5 |
| 3 | ori $10,$6,0 | 6 |
| 4 | sw $5,0($10) | 7 |
| 5 | lw $2,65524($10) | 8 |
| 6 | addi $5,$2,4 | 10 |
| 7 | bne $5,$2,$l5 | 12 |
| 8 | add $0,$0,$0 | 13 |

Total de cycles : 13

RENOMMAGE

|  |  |  |
| --- | --- | --- |
| Instruction | Code | Cycle de sortie |
| 0 | lw $9,0($6) | 1 |
| 1 | lw $2,0($9) | 3 |
| 2 | add $11,$14,$2 | 5 |
| 3 | ori $10,$6,0 | 6 |
| 4 | sw $11,0($10) | 7 |
| 5 | lw $2,65524($10) | 8 |
| 6 | addi $5,$2,4 | 10 |
| 7 | bne $5,$2,$l5 | 12 |
| 8 | add $0,$0,$0 | 13 |

Total de cycles : 13

SCHEDULING

|  |  |  |
| --- | --- | --- |
| Instruction | Code | Cycle de sortie |
| 0 | lw $4,0($6) | 1 |
| 3 | ori $10,$6,0 | 5 |
| 1 | lw $2,0($4) | 2 |
| 2 | add $5,$14,$2 | 3 |
| 5 | lw $2,65524($10) | 7 |
| 4 | sw $5,0($10) | 6 |
| 6 | addi $5,$2,4 | 8 |
| 7 | bne $5,$2,$l5 | 10 |
| 8 | add $0,$0,$0 | 11 |

Total de cycles : 11

RENOMMAGE & SCHEDULING

|  |  |  |
| --- | --- | --- |
| Instruction | Code | Cycle de sortie |
| 0 | lw $9,0($6) | 1 |
| 3 | ori $10,$6,0 | 5 |
| 1 | lw $2,0($9) | 2 |
| 2 | add $11,$14,$2 | 3 |
| 5 | lw $2,65524($10) | 7 |
| 4 | sw $11,0($10) | 6 |
| 6 | addi $5,$2,4 | 8 |
| 7 | bne $5,$2,$l5 | 10 |
| 8 | add $0,$0,$0 | 11 |

On observe bien un gain de cycle avec l’opération de re-nommage & re-ordonnacement. Cependant l’opération de re-nommage seule n’a pas eu d’impact sur ce résultat, en effet le nombre de cycle nécessaires restant identique à l’original. De plus on retrouve la même optimisation en n’utilisant que le re-ordonnacement. (Peut-être un problème avec notre fonction de renommage?)

Soit le bloc de base suivant et le nombre de cycle nécessaire à son exécution (on utilise la table deuxième table de délais):

main:

**i0 lw $4,0($6)**

**i1 lw $2,0($4)**

**i2 add $5,$14,$2**

**i3 ori $10,$6,0**

**i4 sw $5,0($10)**

**i5 lw $2,65524($10)**

**i6 addi $5,$2,4**

**i7 bne $5,$2,$l5**

**i8 add $0,$0,$0**

|  |  |  |
| --- | --- | --- |
| Instruction | Code | Cycle de sortie |
| 0 | lw $4,0($6) | 1 |
| 1 | lw $2,0($4) | 4 |
| 2 | add $5,$14,$2 | 7 |
| 3 | ori $10,$6,0 | 8 |
| 4 | sw $5,0($10) | 10 |
| 5 | lw $2,65524($10) | 11 |
| 6 | addi $5,$2,4 | 14 |
| 7 | bne $5,$2,$l5 | 17 |
| 8 | add $0,$0,$0 | 18 |

Total de cycles : 18

RENOMMAGE

|  |  |  |
| --- | --- | --- |
| Instruction | Code | Cycle de sortie |
| 0 | lw $9,0($6) | 1 |
| 1 | lw $2,0($9) | 4 |
| 2 | add $11,$14,$2 | 7 |
| 3 | ori $10,$6,0 | 8 |
| 4 | sw $11,0($10) | 10 |
| 5 | lw $2,65524($10) | 11 |
| 6 | addi $5,$2,4 | 14 |
| 7 | bne $5,$2,$l5 | 17 |
| 8 | add $0,$0,$0 | 18 |

Total de cycles : 18

SCHEDULING

|  |  |  |
| --- | --- | --- |
| Instruction | Code | Cycle de sortie |
| 0 | lw $4,0($6) | 1 |
| 3 | ori $10,$6,0 | 7 |
| 1 | lw $2,0($4) | 2 |
| 2 | add $5,$14,$2 | 4 |
| 5 | lw $2,65524($10) | 9 |
| 4 | sw $5,0($10) | 8 |
| 6 | addi $5,$2,4 | 11 |
| 7 | bne $5,$2,$l5 | 14 |
| 8 | add $0,$0,$0 | 15 |

Total de cycles : 15

RENOMMAGE & SCHEDULING

|  |  |  |
| --- | --- | --- |
| Instruction | Code | Cycle de sortie |
| 0 | lw $9,0($6) | 1 |
| 3 | ori $10,$6,0 | 7 |
| 1 | lw $2,0($9) | 2 |
| 2 | add $11,$14,$2 | 4 |
| 5 | lw $2,65524($10) | 9 |
| 4 | sw $11,0($10) | 8 |
| 6 | addi $5,$2,4 | 11 |
| 7 | bne $5,$2,$l5 | 14 |
| 8 | add $0,$0,$0 | 15 |

Total de cycles : 15

Encore une fois, on observe bien une optimisation du nombre de cycle avec l’opération de re-nommage & re-ordonnancement, mais le re-nommage ne semble pas avoir d’incidence sur ces résultats.