Cheat Sheet | Compilation

2025-12-30

F. B
Universite de Montpellier
2025
M1

```
=== CHEAT SHEET INSTRUCTIONS RTL / ERTL (MIPS) ===

------------------------------------------------------------
1. BRANCHEMENTS SUR ZERO (Test 1 Registre -> Label)
------------------------------------------------------------
Instruction   | Signification (Anglais)       | Condition Mathématique
--------------|-------------------------------|-----------------------
beqz %r, L    | Branch Equal Zero             | if (%r == 0) goto L
bnez %r, L    | Branch Not Equal Zero         | if (%r != 0) goto L
blez %r, L    | Branch Less Equal Zero        | if (%r <= 0) goto L
bgez %r, L    | Branch Greater Equal Zero     | if (%r >= 0) goto L
bltz %r, L    | Branch Less Than Zero         | if (%r <  0) goto L
bgtz %r, L    | Branch Greater Than Zero      | if (%r >  0) goto L

------------------------------------------------------------
2. BRANCHEMENTS COMPARAISON (Test 2 Registres -> Label)
------------------------------------------------------------
Instruction       | Signification             | Condition Mathématique
------------------|---------------------------|-----------------------
beq %r1, %r2, L   | Branch Equal              | if (%r1 == %r2) goto L
bne %r1, %r2, L   | Branch Not Equal          | if (%r1 != %r2) goto L
bge %r1, %r2, L   | Branch Greater Equal      | if (%r1 >= %r2) goto L
ble %r1, %r2, L   | Branch Less Equal         | if (%r1 <= %r2) goto L
bgt %r1, %r2, L   | Branch Greater Than       | if (%r1 >  %r2) goto L
blt %r1, %r2, L   | Branch Less Than          | if (%r1 <  %r2) goto L

------------------------------------------------------------
3. ARITHMÉTIQUE (Destination, Source1, Source2)
------------------------------------------------------------
Instruction       | Description               | Exemple
------------------|---------------------------|-----------------------
add %d, %s1, %s2  | Addition                  | %d = %s1 + %s2
sub %d, %s1, %s2  | Soustraction              | %d = %s1 - %s2
mul %d, %s1, %s2  | Multiplication            | %d = %s1 * %s2
div %d, %s1, %s2  | Division                  | %d = %s1 / %s2
addi %d, %s, N    | Add Immediate (Constante) | %d = %s + 5
addiu %d, %s, N   | Add Imm. Unsigned (Idem)  | %d = %s + 5
```

```
------------------------------------------------------------
METTRE TOUJOURS PROLOGUE / EPILOGUE POUR ERTL
------------------------------------------------------------
PROLOGUE (AU DEBUT)
f_entry: newframe -> f_save
f_save:  move %save_ra $ra -> f_save_s  ; OBLIGATOIRE si appel de fonction
f_save_s: move %save_s0 $s0 -> f_arg    ; OBLIGATOIRE si on utilise $s0
f_arg:   move %0 $a0 -> f_body          ; Récupération de l'argument
------------------------------------------------------------
EPILOGUE/NETOYYAGE (A LA FIN)
f_ret:   move $v0 %res -> f_rest_s      ; Mettre le résultat dans $v0
f_rest_s: move $s0 %save_s0 -> f_rest_ra; Restaurer $s0 (si sauvegardé au
début)
f_rest_ra: move $ra %save_ra -> f_del   ; RESTAURER $ra (Crucial !)
f_del:   delframe -> f_jr
f_jr:    jr $ra -> f_end
------------------------------------------------------------
```

Qu'est-ce qui change d'une fonction à l'autre ? Il y a deux choses qui peuvent changer le prologue/épilogue :

A. As-tu des appels de fonction (call) ?

- OUI (Cas le plus fréquent en exam) : Tu DOIS sauvegarder ra.
    - Pourquoi ? Parce que l'instruction call écrase ra. Si tu ne le sauves pas dans une variable temporaire (%save_ra), tu ne pourras jamais faire le jr ra final pour rentrer chez toi.
- NON (Fonction feuille) : Tu n'es pas obligé de sauver ra. Exemple : Une fonction add(a,b) qui fait juste return a+b.

B. As-tu besoin de registres qui survivent (s_0) ?

- OUI : Si tu as besoin d'une variable (comme n) qui doit être conservée pendant un appel récursif, tu vas la mettre dans s_0. Du coup, tu DOIS sauvegarder le vieux s_0 au début et le restaurer à la fin.
- NON : Si tu n'utilises que des registres temporaires, pas besoin de toucher à s_0.

Pour ne pas prendre de risques à l'examen, utilise toujours le **Modèle Complet** :

1. `newframe`
2. **Sauve** `$ra` (dans une variable `%tmp1`)
3. **Sauve** `$s0` (dans une variable `%tmp2` - seulement si tu t'en sers)
4. **Récupère** `$a0`
5. ... Ton code ...
6. **Remplis** `$v0`
7. **Restaure** `$s0` (depuis `%tmp2`)
8. **Restaure** `$ra` (depuis `%tmp1`)
9. `delframe`
10. `jr $ra`

```
------------------------------------------------------------
4. MOUVEMENT DE DONNÉES
------------------------------------------------------------
Instruction       | Description               | Exemple
------------------|---------------------------|-----------------------
li %d, N          | Load Immediate (Entier)   | %d = 10
move %d, %s       | Copie de registre         | %d = %s

------------------------------------------------------------
5. CONTRÔLE DE FLUX (Spécifique Graphe RTL)
------------------------------------------------------------
Instruction       | Description
------------------|--------------------------
call %r f(%a)     | Appelle f avec arg %a, met résultat dans %r
-> L              | Saut inconditionnel vers L (GOTO)
-> L1, L2         | Saut conditionnel (Vrai -> L1, Faux -> L2)
exit final        | Point de sortie (retourne le registre résultat)

------------------------------------------------------------
6. REGISTRES PHYSIQUES (MIPS - Les "Vrais" Registres)
------------------------------------------------------------
Registre | Nom     | Rôle (Convention Standard)
---------|---------|-------------------------------------
$a0 - $a3| Args    | Arguments de la fonction (Input)
$v0      | Value   | Valeur de retour (Output)
$ra      | Return  | Return Address (Adresse de retour) *CRUCIAL*
$s0 - $s7| Saved   | Callee-Save (Doivent être remis à l'état initial)
$t0 - $t9| Temp    | Caller-Save (Peuvent être écrasés)
$sp      | Stack   | Stack Pointer (Pointeur de pile)
$fp      | Frame   | Frame Pointer (Pointeur de bloc)
```

```
------------------------------------------------------------
7. INSTRUCTIONS SPÉCIALES ERTL (Gestion de Pile)
------------------------------------------------------------
Instruction      | Description
-----------------|------------------------------------------
newframe         | Crée l'espace sur la pile (Prologue)
delframe         | Libère l'espace sur la pile (Épilogue)
jr $ra           | "Jump Register" : Retourne réellement à l'appelant
                 | (C'est le vrai "return" physique)

------------------------------------------------------------
8. LE "SQUELETTE" TYPE D'UNE FONCTION ERTL (A apprendre !)
------------------------------------------------------------
Pour une fonction f, voici l'ordre OBLIGATOIRE des blocs :

--- 1. PROLOGUE (Sauvegardes) ---
f0: newframe -> f1            ; 1. Ouvrir la frame
f1: move %save_ra $ra -> f2   ; 2. SAUVER L'ADRESSE DE RETOUR (Obligatoire
si appel récursif)
f2: move %save_s0 $s0 -> f3   ; 3. Sauver les registres "Saved" si on les
utilise
f3: move %arg0 $a0 -> f4      ; 4. Récupérer les arguments dans des
variables virtuelles

--- 2. CORPS DE LA FONCTION ---
... Logique, calculs, tests ...
... Si appel de fonction :
    move $a0, %valeur         ; a. Mettre l'arg dans le registre physique
    call f(1)                 ; b. Appeler
    move %res, $v0            ; c. Récupérer le résultat depuis le registre
    physique

--- 3. ÉPILOGUE (Restaurations) ---
f_end: move $v0, %res -> r1   ; 1. Mettre le résultat final dans $v0
r1:    move $s0, %save_s0 -> r2; 2. Restaurer les registres "Saved"
r2:    move $ra, %save_ra -> r3; 3. RESTAURER L'ADRESSE DE RETOUR (Vital !)
r3:    delframe -> r4         ; 4. Fermer la frame
r4:    jr $ra                 ; 5. Sauter à l'adresse de retour
```

```
============================================================
===   PARTIE 2 : COMPILATION LISP VERS VM (AUTOMATES)   ===
============================================================

------------------------------------------------------------
1. REGISTRES VM (Conventionnelle pour cet exercice)
------------------------------------------------------------
Registre | Rôle typique
---------|--------------------------------------------------
R0       | La "Bande" (Liste d'entrée restante)
R1       | Le "Symbole Courant" (Caractère lu via CAR)
R2       | Debug / État courant (Optionnel mais utile)
------------------------------------------------------------

------------------------------------------------------------
2. JEU D'INSTRUCTIONS VM (Instruction Set)
------------------------------------------------------------
Instruction           | Description
----------------------|-------------------------------------
(LABEL label)         | Définit une étiquette (Point de saut)
(MOVE src dest)       | Copie une valeur.
                      | Ex: (MOVE (:CONST a) R1) ou (MOVE R0 R1)
(CAR src dest)        | Prend la TÊTE de la liste src -> dest
(CDR src dest)        | Prend la QUEUE de la liste src -> dest
(BNULL reg label)     | Si le registre est vide (nil), saut vers label
                      | *Crucial pour détecter la fin du mot*
(CMP reg val)         | Compare reg avec une valeur (:CONST x)
(JEQ label)           | Saut si la comparaison précédente était ÉGALE
(JMP label)           | Saut inconditionnel (Toujours)
(HALT)                | Arrête la machine

------------------------------------------------------------
3. LOGIQUE DE TRADUCTION (Automate -> Code)
------------------------------------------------------------
Concept Automate      | Traduction en Code VM
----------------------|-------------------------------------
État (Rond)           | (LABEL nom_etat)
État Initial          | (LABEL start) + (JMP etat_initial)
Transition (Flèche)   | 1. (CMP R1 (:CONST symbole))
                      | 2. (JEQ etat_cible)
Lire un caractère     | (CAR R0 R1) suivi de (CDR R0 R0)
Mot vide / Fin        | (BNULL R0 ...) au début de chaque état
```

```
État Final (Double)   | Si BNULL est vrai -> (JMP accepter)
État Non-Final        | Si BNULL est vrai -> (JMP refuser)

------------------------------------------------------------
4. SQUELETTE TYPE D'UN ÉTAT (A apprendre par cœur !)
------------------------------------------------------------
(LABEL etat_N)
  ; 1. Vérifier si fini
  (BNULL R0 label_sortie)  ; label_sortie = 'accepter' si final, sinon
  'refuser'

  ; 2. Lire
  (CAR R0 R1)              ; R1 = char
  (CDR R0 R0)              ; R0 = reste

  ; 3. Tester les transitions
  (CMP R1 (:CONST a))      ; Est-ce un 'a' ?
  (JEQ etat_X)             ; Si oui -> aller à l'état X

  (CMP R1 (:CONST b))      ; Est-ce un 'b' ?
  (JEQ etat_Y)             ; Si oui -> aller à l'état Y

  ; 4. Rejet par défaut (si aucun caractère ne correspond)
  (JMP refuser)

------------------------------------------------------------
5. SYNTAXE LISP GÉNÉRATIVE (Backquote)
------------------------------------------------------------
Symbole | Nom             | Usage
--------|-----------------|--------------------------------
`       | Backquote       | Début du "patron" de code
,       | Comma (Virgule) | "Évalue cette variable ici"
        |                 | Ex: `(LABEL ,nom-variable)
        |                 | -> (LABEL etat1)
------------------------------------------------------------
```