Programmation en Langage d'Assemblage : Des Concepts Fondamentaux aux Appels de Fonctions

Introduction : Du Problème à l'Algorithme

Contextualisation

Avant de plonger dans l'écriture de code, il est impératif de se remémorer les fondations sur lesquelles repose toute l'informatique moderne. Nous opérons toujours dans le cadre du modèle de Von Neumann, où instructions et données coexistent dans une même mémoire. Chaque action que nous demandons à l'ordinateur de réaliser passe par le cycle d'exécution d'instructions : Fetch (Recherche), Decode (Décodage), Execute (Exécution), et Store (Stockage).[1] Garder ces principes à l'esprit est essentiel, car ils dictent la nature séquentielle de nos programmes et la manière dont nous devons les structurer.[1]

Le Processus de Résolution de Problèmes

La première étape, et la plus cruciale, de la programmation n'est pas d'écrire du code, mais de traduire un problème du monde réel en un algorithme, une séquence finie et non ambiguë d'opérations.[1] Pour la programmation à bas niveau, comme en langage d'assemblage, cette traduction prend souvent la forme d'un organigramme (flowchart). Cet outil visuel nous force à décomposer une tâche complexe en une série d'étapes simples et de décisions logiques, définissant même l'assignation des registres avant qu'une seule ligne d'assembleur ne soit écrite.[1]

Introduction aux Architectures Étudiées

Au cours de cette session, nous explorerons deux architectures de jeu d'instructions (ISA) : le LC-3 et le MIPS. Bien qu'elles présentent des similarités, elles servent des objectifs pédagogiques distincts.[1] Le LC-3 est une architecture simplifiée, conçue pour l'enseignement. Sa transparence nous permet de nous concentrer sur la mécanique fondamentale des boucles et des branchements sans être submergés par la complexité.[1] Le MIPS, quant à lui, est une architecture plus réaliste et puissante, utilisée dans de nombreux

systèmes réels. Il nous introduira à des concepts plus avancés, indispensables à la programmation moderne, comme les conventions d'appel de fonction.[1]

Cette progression n'est pas un hasard. Elle constitue un parcours d'apprentissage délibéré, allant de l'universel à l'implémentation concrète. Nous commencerons par les concepts de programmation qui transcendent tout langage, puis nous les appliquerons sur une machine simple (LC-3) pour en maîtriser la logique. Nous ajouterons ensuite une couche d'interaction avec le système (les instructions TRAP), nous confronterons à la réalité inévitable des erreurs (le débogage), et enfin, nous passerons à une architecture plus puissante (MIPS) pour aborder la pierre angulaire du génie logiciel : les fonctions modulaires. Maîtriser les exemples en LC-3 est donc un prérequis pour comprendre les solutions plus complexes en MIPS.[1]

Partie 1 : Les Blocs de Construction de la Programmation

Tout algorithme, quelle que soit sa complexité, peut être construit à partir de trois structures de contrôle fondamentales. Le jeu d'instructions d'un processeur est spécifiquement conçu pour permettre la mise en œuvre

de ces trois blocs de construction.[1]

La Structure Séquentielle

La structure séquentielle est la plus simple. Elle consiste à exécuter une série de sous-tâches les unes après les autres, dans un ordre prédéfini.[1] C'est le comportement par défaut du modèle de Von Neumann, où le Compteur de Programme (PC) s'incrémente automatiquement pour exécuter l'instruction suivante en mémoire, à moins qu'un ordre de branchement ne vienne briser cette séquence.[1]

La Structure Conditionnelle

La structure conditionnelle permet à un programme de prendre des décisions. Elle exécute une sous-tâche parmi plusieurs possibles en fonction de la véracité d'une condition, mais jamais toutes.[1] C'est le mécanisme qui nous permet d'implémenter la logique

if-then-else ou les instructions switch-case. Le flux d'exécution n'est plus linéaire ; il est altéré en fonction des données traitées.[1]

La Structure Itérative

La structure itérative, ou boucle, est utilisée lorsqu'une sous-tâche doit être répétée plusieurs fois.[1] Elle combine une condition de continuation (ou de sortie) et un corps de boucle. La sous-tâche est exécutée tant que la condition est vraie. Les boucles

for, while, et do-while sont toutes des implémentations de cette structure fondamentale.[1]

Partie 2 : Premier Programme en LC-3 : Somme d'un Tableau d'Entiers

Appliquons maintenant ces concepts avec un premier programme concret sur l'architecture LC-3.

Définition du Problème

La tâche consiste à écrire un programme qui calcule la somme de 12 entiers. Ces entiers sont stockés dans des adresses mémoires contiguës, allant de $x3100$ à $x310B$.[1]

L'Algorithme et l'Organigramme

Avant d'écrire le code, nous définissons l'algorithme via un organigramme.[1]

1. **Initialisation :** Nous devons préparer nos registres. Nous assignons des rôles spécifiques : R1 servira de pointeur vers l'adresse mémoire de l'entier courant, R3 sera notre accumulateur pour stocker la somme, et R2 agira comme un compteur, initialisé avec le nombre d'entiers à additionner (12).[1]
2. **La Boucle Itérative :** C'est le cœur de notre algorithme.
    - **Condition de Sortie :** À chaque début d'itération, nous testons si le compteur R2 est égal à zéro. Si c'est le cas, cela signifie que nous avons traité tous les entiers, et la boucle doit se terminer.[1]
    - Corps de la Boucle : Si R2 n'est pas nul, nous exécutons le corps de la boucle :
      a. Charger l'entier depuis l'adresse mémoire pointée par R1.
      b. Ajouter cette valeur à l'accumulateur R3.
      c. Incrémenter le pointeur R1 pour qu'il pointe vers l'entier suivant.
      d. Décrémenter le compteur R2.1

Pseudo-code de l'Algorithme de Sommation

Extrait de code

```
PROCEDURE SumArray(start_address, count)
  pointer_R1 = start_address
  sum_R3 = 0
  counter_R2 = count

  LOOP
    IF counter_R2 == 0 THEN
      BREAK LOOP
    END IF

    current_value = MEMORY
    sum_R3 = sum_R3 + current_value
    pointer_R1 = pointer_R1 + 1
    counter_R2 = counter_R2 - 1
  END LOOP

  RETURN sum_R3
END PROCEDURE
```

Implémentation en Assembleur LC-3

La traduction de cet algorithme en code assembleur LC-3 est directe. Chaque étape de l'organigramme correspond à une ou plusieurs instructions.[1]

Tableau 1 : Correspondance Algorithme-Code LC-3

Étape du Pseudo-code	Instruction(s) LC-3	Rôle de l'Instruction
pointer_R1 =...	LEA R1, x3100	Initialise le pointeur de données.
sum_R3 = 0	AND R3, R3, #0	Met à zéro le registre accumulateur.
counter_R2 =...	ADD R2, R2, #12	Initialise le compteur de boucle à 12.
IF counter_R2 == 0	BRz END_LOOP	Teste la condition de fin de boucle.
current_value =...	LDR R4, R1, #0	Charge la donnée depuis la mémoire.

sum_R3 =...	ADD R3, R3, R4	Ajoute la valeur à la somme courante.
pointer_R1 =...	ADD R1, R1, #1	Pointe vers l'entier suivant.
counter_R2 =...	ADD R2, R2, #-1	Décrémente le nombre d'itérations.
LOOP	BRnzp LOOP_START	Branchement inconditionnel au début.

Ce tableau illustre parfaitement comment les concepts abstraits de l'algorithme sont matérialisés par des instructions machine concrètes, comblant le fossé entre la pensée algorithmique et le codage à bas niveau.[1]

Partie 3 : Interaction avec le Système : L'Instruction TRAP et le Comptage de Caractères

Nos programmes ne vivent pas en vase clos. Ils doivent interagir avec le monde extérieur, comme le clavier et l'écran.

L'Instruction TRAP : Une Porte vers le Système d'Exploitation

En LC-3, l'instruction TRAP est le mécanisme standardisé par lequel un programme utilisateur demande un service au système d'exploitation.[1] Il s'agit d'une forme d'interruption logicielle. L'instruction

TRAP est suivie d'un numéro de 8 bits, appelé trap vector, qui identifie le service demandé. Le système d'exploitation prend alors le contrôle, exécute le service requis (par exemple, lire une touche du clavier), puis rend la main au programme.[1]

Tableau 2 : Vecteurs TRAP Courants en LC-3

Vecteur TRAP	Mnémonique	Description du Service
$x23$	GETC	Lit un seul caractère depuis le clavier.
$x21$	OUT	Affiche un seul caractère sur la console.
$x25$	HALT	Arrête l'exécution du programme.
$x22$	PUTS	Affiche une chaîne

		de caractères terminée par null.

Algorithme de Comptage d'Occurrences de Caractères

Utilisons les TRAP dans un programme plus complexe qui compte les occurrences d'un caractère dans un fichier. Cet algorithme met en œuvre les trois structures de contrôle.[1]

La logique est la suivante :

1. Initialiser un compteur à zéro.
2. Demander à l'utilisateur de saisir le caractère à rechercher via un TRAP GETC.
3. Entrer dans une boucle qui lit un fichier caractère par caractère. Ce fichier est stocké en mémoire.
4. La boucle se termine lorsqu'un caractère spécial, appelé sentinelle (par exemple, EOT - End of Text), est rencontré.
5. À l'intérieur de la boucle, chaque caractère lu est comparé au caractère cible. Si c'est le même, le compteur est incrémenté.
6. Une fois la boucle terminée, le résultat (le compte) est converti en un caractère affichable et imprimé à l'écran via un TRAP OUT.

7. Le programme se termine avec un TRAP HALT.[1]

Pseudo-code du Comptage de Caractères

Extrait de code

```
PROCEDURE CountCharacter(file_start_address, EOT_char)
  count = 0
  pointer = file_start_address
  target_char = TRAP(GETC) // Lit un caractère au clavier

  LOOP
    current_char = MEMORY[pointer]

    IF current_char == EOT_char THEN
      BREAK LOOP
    END IF

    IF current_char == target_char THEN
      count = count + 1
    END IF
```

```
    pointer = pointer + 1
  END LOOP

  // Convertit le compte en ASCII pour l'affichage
  display_char = count + ASCII_OFFSET('0')
  TRAP(OUT, display_char) // Affiche le résultat
  TRAP(HALT)
END PROCEDURE
```

Partie 4 : L'Art du Débogage

Écrire du code est une chose, s'assurer qu'il fonctionne correctement en est une autre. Les erreurs, ou "bugs", sont inévitables. Le débogage est le processus méthodique pour les trouver et les corriger.[1]

Principes du Débogage Interactif

Les débogueurs modernes fournissent des outils puissants pour analyser l'exécution d'un programme.[1]

- **Traçage (Tracing) :** Consiste à suivre la séquence d'instructions exécutées et à observer l'état des registres et de la mémoire après chaque étape.

- **Exécution Pas-à-Pas (Single-Stepping) :** Permet d'exécuter le programme une seule instruction à la fois (step). C'est l'outil ultime pour isoler le moment précis où le comportement du programme dévie de ce qui est attendu.[1]
- **Points d'Arrêt (Breakpoints) :** Un point d'arrêt est une instruction à laquelle on demande au débogueur de pauser l'exécution. Cela permet d'inspecter l'état du programme à des endroits stratégiques (par exemple, au début de chaque itération d'une boucle) sans avoir à parcourir manuellement toutes les instructions intermédiaires.[1]

Étude de Cas : Le Programme de Multiplication Bogué en LC-3

Analysons un exemple concret : un programme LC-3 censé multiplier le contenu du registre R4 (valeur 10) par celui de R5 (valeur 3). Le résultat attendu est 30, mais le programme produit 40.[1] Cet exemple est riche d'enseignements, car il contient non pas un, mais deux bugs distincts qui illustrent deux classes différentes d'erreurs de programmation.

- **Erreur n°1 - Erreur de Logique de Boucle :** Le premier bug est une erreur classique de type "off-by-one". En traçant l'exécution, on découvre que

la boucle s'exécute une fois de trop. La cause est une mauvaise instruction de branchement. Le code utilise BRzp (Branch if Zero or Positive), ce qui signifie que la boucle s'exécute une dernière fois lorsque le compteur R5 atteint zéro. La correction consiste à utiliser BRp (Branch if Positive), de sorte que la boucle s'arrête dès que le compteur n'est plus strictement positif.[1] C'est une erreur mécanique dans l'implémentation de la condition de sortie de l'algorithme.

- **Erreur n°2 - Oubli d'un Cas Limite (Corner Case) :** Le second bug est plus subtil. Que se passe-t-il si la valeur initiale de R5 est 0? Le programme, même corrigé, ne fonctionne pas. Il exécuterait le corps de la boucle une fois avant de tester la condition, produisant un résultat de 10 au lieu de 0.[1] Ceci révèle une défaillance de spécification. Le programmeur n'a pas envisagé tous les cas de figure possibles, en particulier les cas limites ou "corner cases".[1]

Cette étude de cas démontre que le développement de logiciels robustes ne consiste pas seulement à traduire correctement un algorithme. Il exige d'adopter un état d'esprit contradictoire, de se demander constamment : "Comment ce code pourrait-il échouer?". Le débogage n'est pas une tâche corrective, mais une partie intégrante

du processus de conception, qui doit inclure la gestion des erreurs de logique et la couverture exhaustive des cas limites.

Tableau 3 : Table de Traçage pour le Débogage de la Multiplication

Itération	PC	R2 (Résultat)	R5 (Compteur)	Codes (N,Z,P)	Action (Buggy BRzp)	Action (Correct BRp)
1	LOOP	10	2	P	Continue	Continue
2	LOOP	20	1	P	Continue	Continue
3	LOOP	30	0	Z	**Continue (Bug!)**	**Sort (Correct)**
4	LOOP	40	-1	N	Sort	-

Partie 5 : Structures de Contrôle et Manipulation de Tableaux en MIPS

Après avoir maîtrisé les bases sur LC-3, nous sommes

prêts à aborder l'architecture MIPS, plus complexe et plus proche du monde réel.

Implémentation des Structures Conditionnelles (if-else)

En MIPS, pour implémenter une condition if (i == j), on utilise une stratégie de "test inversé". On utilise l'instruction bne (Branch on Not Equal) pour sauter par-dessus le bloc if si la condition est fausse. Pour une structure if-else complète, on combine ce branchement conditionnel avec un saut inconditionnel (j). Si la condition if est vraie, son bloc est exécuté, puis un j saute par-dessus le bloc else.[1]

Implémentation des Boucles (while, for)

La logique est similaire à celle du LC-3. Une boucle while ou for est typiquement implémentée avec un branchement conditionnel en haut de la boucle pour tester la condition de sortie, et un saut inconditionnel en bas pour retourner au début de la boucle.[1] MIPS offre également des instructions de comparaison plus sophistiquées comme

slt (Set on Less Than), qui place 1 ou 0 dans un registre de destination en fonction du résultat de la comparaison. Ce résultat peut ensuite être testé pour contrôler la boucle.[1]

Accès et Manipulation de Tableaux en MIPS

- **Le Problème de l'Adresse 32-bits :** Les instructions MIPS ont une taille fixe de 32 bits. Il est donc impossible d'encoder une adresse mémoire complète de 32 bits comme une valeur immédiate dans une seule instruction.[1]
- **La Solution LUI/ORI :** La solution standard est une séquence de deux instructions. LUI (Load Upper Immediate) charge les 16 bits de poids fort de l'adresse dans un registre. Ensuite, ORI (OR Immediate) applique un masque pour insérer les 16 bits de poids faible, construisant ainsi l'adresse complète de 32 bits dans le registre.[1]
- **Calcul de l'Adresse d'un Élément :** Une fois l'adresse de base du tableau dans un registre, l'adresse d'un élément spécifique est calculée avec la formule : Adresse_Élément = Adresse_Base + (Index * Taille_Élément). Comme la multiplication est une opération coûteuse, on utilise une astuce : si la

taille de l'élément est une puissance de deux (par exemple, 4 octets pour un mot), on la remplace par une instruction de décalage de bits, sll (Shift Left Logical), qui est beaucoup plus rapide.[1]

Partie 6 : Mécanismes et Conventions d'Appel de Fonctions en MIPS

Nous arrivons au sommet de notre parcours : la construction de programmes modulaires et réutilisables grâce aux fonctions.

Les Six Étapes d'un Appel de Fonction

Conceptuellement, un appel de fonction se déroule en six étapes :

1. L'appelant (caller) place les arguments là où l'appelé (callee) peut les trouver.
2. L'appelant transfère le contrôle à l'appelé.
3. L'appelé acquiert les ressources de stockage dont il a besoin.
4. L'appelé exécute sa tâche.
5. L'appelé place la valeur de retour là où l'appelant peut la trouver.

6. L'appelé retourne le contrôle à l'appelant à l'instruction qui suit l'appel.

Support Matériel : Les Instructions jal et jr

MIPS fournit un support matériel minimal pour ce processus.[1]

- jal (Jump and Link) : L'appelant utilise cette instruction. Elle effectue deux actions simultanément : elle saute à l'adresse de la fonction appelée et sauvegarde l'adresse de retour (l'instruction suivante, PC+4) dans un registre dédié, $ra (return address).
- jr $ra (Jump Register) : L'appelé utilise cette instruction pour retourner. Elle saute simplement à l'adresse contenue dans le registre $ra.

Le Problème de la Récursivité et des Appels Imbriqués

Ce mécanisme matériel simple a une limite fondamentale. Si une fonction A appelle une fonction B, l'instruction jal vers B écrase la valeur de $ra qui contenait l'adresse de retour pour A. Si B essayait de retourner, elle le ferait

correctement, mais A aurait perdu son propre chemin de retour.[1]

La Pile (The Stack) comme Solution

La solution à ce problème est une structure de données logicielle : la pile (stack). C'est une zone de mémoire gérée selon le principe LIFO (Last-In, First-Out). Avant qu'une fonction n'effectue un appel imbriqué (un appel à une autre fonction), elle doit sauvegarder la valeur actuelle de $ra (et tout autre registre important qu'elle doit préserver) sur la pile. Une fois que l'appel imbriqué est terminé, elle restaure la valeur de $ra depuis la pile avant d'exécuter son propre jr $ra.[1]

Le mécanisme d'appel de fonction en MIPS est bien plus qu'un simple ensemble d'instructions ; c'est un contrat, une "interface de programmation" [1] qui permet à du code écrit par différents programmeurs de fonctionner ensemble de manière fiable. Le matériel fournit les crochets (

jal, jr), mais tout le système de passage d'arguments, de valeurs de retour et de sauvegarde des registres est une **convention logicielle** que tous les programmeurs doivent respecter pour que la modularité fonctionne.

C'est la solution au chaos. Cette convention distingue les registres que l'appelé peut modifier librement (temporaires, ou "caller-saved", $t0-$t9$) de ceux qu'il a la responsabilité de sauvegarder et de restaurer s'il les utilise (sauvegardés, ou "callee-saved", $s0-$s7$). Cette répartition des responsabilités est la clé qui permet de construire des logiciels vastes et complexes à partir de briques indépendantes et fiables.[1]

Tableau 4 : Résumé des Conventions de Registres MIPS

Nom du Registre	Numéro	Usage	Préservé par l'appelé?
$zero	0	Constante 0	N/A
$v0-$v1	2-3	Valeurs de retour de fonction	Non
$a0-$a3	4-7	Arguments de fonction	Non
$t0-$t9	8-15, 24-25	Registres temporaires (caller-saved)	Non
$s0-$s7	16-23	Registres	**Oui**

		sauvegardés (callee-saved)	
$sp	29	Pointeur de pile (Stack Pointer)	**Oui**
$ra	31	Adresse de retour	**Oui**

Conclusion : Synthèse et Prochaines Étapes

Au cours de cette session, nous avons entrepris un voyage complet, partant des principes les plus abstraits de la programmation pour arriver à l'implémentation concrète d'appels de fonction complexes sur une architecture réaliste. Nous avons vu que tout programme se construit à partir de trois structures de base : séquentielle, conditionnelle et itérative.[1] Nous avons appliqué ces concepts sur une architecture pédagogique, le LC-3, pour en maîtriser la logique fondamentale.[1]

Nous avons ensuite confronté la réalité inévitable des erreurs de programmation, en apprenant que le débogage est un art qui exige une pensée contradictoire

et une attention particulière aux cas limites.[1] Enfin, en passant à l'architecture MIPS, nous avons découvert que la construction de logiciels modulaires et à grande échelle ne repose pas seulement sur le matériel, mais sur un ensemble de conventions logicielles rigoureuses qui forment un contrat entre les différentes parties d'un programme.[1]

Vous possédez désormais les connaissances fondamentales pour comprendre comment les fonctionnalités des langages de haut niveau, telles que les fonctions, la récursivité et les variables locales, sont finalement traduites en ces opérations élémentaires de bas niveau. Le "miracle" de l'abstraction logicielle n'est plus magique ; c'est une pyramide de concepts bien définis, dont vous venez de maîtriser la base.[1]

Sources des citations

1. transcription.txt