Documentation Développeur
Simulation d'une machine virtuelle

Architecture des ordinateurs – MIDO L2 2025/2026
Dahan Benjamin et Baudouin Rémy

Abstract

This technical documentation details the complete design and implementation of a 32-bit Virtual Machine (VM) and its associated Assembler, developed entirely in C. The project simulates a Von Neumann architecture, handling the full Fetch-Decode-Execute cycle without relying on high-level abstractions.

We first establish a robust memory management system simulating a 64KB RAM with Little Endian addressing, manual byte reconstruction, and strict boundary checks to prevent segmentation faults. The core CPU simulation relies on a custom Arithmetic Logic Unit (ALU) that handles bitwise operations, sign extensions, and register manipulation (R_0 to R_{31}) through low-level masks and shifts.

Furthermore, to bridge the gap between human-readable code and machine execution, we implemented a two-pass assembler. This module resolves symbolic labels in a first pass before encoding instructions into 32-bit hexadecimal machine code in the second. The resulting codebase emphasizes modularity, memory safety, and algorithmic efficiency.

Keywords : Virtual Machine, Assembly, C Programming, Computer Architecture, Little Endian, Bitwise Operations, Von Neumann.

Table des matières

Introduction

Bienvenue dans la documentation développeur du projet de simulation d'une machine virtuelle programmée en C, celle-ci est destinée aux développeurs voulant comprendre plus en profondeur l'implémentation du projet, ainsi que les raisons et idées derrière nos choix.

Dans ce projet, nous avons programmé en C un assembleur et un simulateur pour une machine virtuelle fictive composée d'une mémoire (64 Ko) et d'un microprocesseur. Notre programme traduit un fichier assembleur en code machine hexadécimal 32 bits, puis exécute ce programme instruction par instruction en simulant le fonctionnement du processeur et de ses registres.
Voir user.pdf pour plus d'indications concernant son utilisation.

Ce document sera séparé en 4 parties :
- L'architecture du Projet.
- L'implémentation du Traducteur Assembleur.
- L'implémentation de l'Executeur.
- Les difficultés rencontrées et les potentielles améliorations.

Nous vous souhaitons une bonne lecture.

Utilisation de l'IA

La documentation a été écrite à la main, et l'IA a été utilisée pour corriger les différentes fautes, améliorer la fluidité certains paragraphes et créer les illustrations en LaTeX.

1 Organisation du Projet

Cette courte partie sera dédiée à l'explication de nos choix concernant l'organisation du code. Dans le but de maximiser la lisibilité du code, nous avons séparés notre programme en un main et différents headers (avec leur fichier.c associé), chacun contenant les fonctions nécessaires à une partie précise du programme. Nous allons donc décrire ici ce qu'ils contiennent :

- Constantes : Les constantes globales utilisées dans le projet.
- Fonctions_outils : Les fonctions outils utilisées dans tout le projet.
- Structures_et_Fonctions_etiquettes : Les fonctions et structures concernant le 1er passage et la collecte des étiquettes.
- Structures_et_Fonctions_instructions : Les fonctions et structures concernant le 2ème passage et la collecte des instructions.
- Conversion_Binaire_Hexa : Les fonctions concernant la conversion des instructions en format binaire et hexadécimal.
- Gestion_memoire : L'ensemble des fonctions permettant de gérer la mémoire.

> **Remarque** — Bibliothèque
>
> Pour la majorité des fonctions outils utilisés dans notre projet, il doit certainement exister une version plus efficace dans une bibliothèque. Mais par souci de comprendre autant que possible notre programme, nous avons décidé de les recréer pour la plupart afin d'utiliser le moins possible de fonctions "magiques".

2 Documentation Developpeur du Traducteur Assembleur

Dans cette partie, nous avons programmé un traducteur de langage assembleur vers une représentation hexadecimale 32 bits (une instruction par ligne), selon le format d'encodage défini dans le sujet. Il lit un fichier source assembleur, vérifie sa validité (syntaxe et sens), résout les étiquettes et overflows, puis produit un fichier `hexa.txt`.

2.1 Organisation générale : stratégie en deux passes puis traduction

Le fichier source est parcouru deux fois :
— Passage 1 : collecte et validation des étiquettes (nom + adresse).
— Passage 2 : collecte et validation des instructions (avec résolution des sauts).
— Traduction des instructions en binaire puis hexadécimal et création du fichier.
Cette stratégie garantit que lors du passage 2, toutes les adresses des étiquettes sont déja connues.

2.2 Constantes, structures, variables globales

2.2.1 Constantes de format

Nous avons fixé un cadre strict via des constantes définies globalement : entre autres la longueur maximale d'une ligne, nombre maximal d'instructions, nombre maximal d'etiquettes... (détaillées dans le pdf user). Nous avons également défini une constante `ERREUR` qui comme son nom l'indique, va agir comme un indicateur d'erreurs.

> **Remarque** — Choix de robustesse
>
> Lorsqu'une erreur est detectée sur une ligne, le programme ne s'arrête pas immédiatement (sauf erreur critique) : il termine le passage en cours afin de remonter un maximum d'erreurs en une seule exécution.

2.2.2 Structures de données

Les informations sont stockées dans deux structures :
— `Etiquette` : contient un pointeur vers une chaine de caractères `etiquette` (nom du label) et un `unsigned int adresse` (adresse associée).

> **Remarque** — Choix de type
>
> Le champ `adresse` est de type `unsigned int` afin de pouvoir stocker les adresses comprises entre 0 et 65535.

— `Instruction` : contient trois `long Rd`, `Rn` et `Src2` (registres et opérande), ainsi qu'un int `Int` prenant comme valeur 0 ou 1 et précisant si `Src2` est une valeur immédiate ou non.

> **Remarque** — Choix de type
>
> Les champs `rd`, `rn` et `Src2` sont stockés en `long` pour permettre une conversion robuste et une détection des overflows.

2.2.3 Variables globales principales

Afin de stocker les instructions et étiquettes, on défini globalement deux tableaux de tailles respectives `MAX_ETIQ` et `MAX_INSTR` contenant des pointeurs vers les structures associées, ainsi que deux `int` : `nbEtiq` et `nbInstr` initialisés à 0 représentant leurs tailles. On définit également

deux `unsigned int ligne_courante` et `adresse_courante`, que l'on initialise à 0 au début de chaque passage et qui permettent respectivement de fournir la ligne où se situe l'erreur dans les messages d'erreur, et de donner les adresses des étiquettes/instructions.

> **Remarque** — Adresses vs numero de ligne
>
> On serait tenter d'utiliser uniquement la variable `ligne_courante` et de calculer l'adresse de la sorte : `(ligne-1) * 4`, mais cela attribuerait une adresse aux commentaires et lignes vides. Utiliser deux variables nous permet d'allouer une adresse mémoire qu'aux instructions effectives rencontrées.

2.3 Passage 1 : Collecte et validation des etiquettes

Chaque ligne est lue dans un buffer de taille `MAX_LIGNE_LONG` et subit des tests dans le main :
— Détection des lignes vides/commentaires que l'on ignore.
— Détection des lignes trop longues via les fonctions `occurences_char` et `feof`.
— Détection des étiquettes via `:` .
— Taille de l'étiquette si étiquette il y a.

> **Remarque** — Tests réalisés dans le `main`
>
> Ces vérifications sont effectuées dans le `main` car elles sont générales et s'appliquent à toutes les lignes lues indépendamment de la présence ou non d'une étiquette. Le test de la taille de l'étiquette constitue une exception puisqu'il n'est pertinent que lorsqu'une étiquette est détectée. Il a néanmoins été placé dans le `main` par souci de simplicité : l'étiquette étant stockée dans un tableau de caractères avant d'être transmise à la fonction de traitement, il est nécessaire de vérifier au préalable que sa taille respecte les contraintes imposées.

L'étiquette est ensuite stockée puis envoyée dans la fonction `ajouter_etiquette` qui s'attelle à détecter de nouvelles erreurs :
— Nombre d'étiquettes.
— Doublons d'étiquettes.
— Caractères interdits.

Si tous ces tests ressortent sans erreur, l'étiquette est alors stocké avec son adresse dans le tableau à cet usage, et sa taille est incrémentée. Une fois le fichier parcouru, si aucune erreur n'a été détectée, on peut alors passer au deuxième passage.

2.4 Passage 2 : Collecte et validation des instructions

Les variables `ligne_courante` et `adresse_courante` sont remis à 0 pour une deuxième lecture du fichier.

> **`decoupe_mot_virgules`**
>
> Cette fonction est cruciale dans le processus de collecte des instructions et a été relativement difficile à programmer, car son bon fonctionnement devait être irréprochable. Elle prend en entrée une chaîne de caractères représentant les opérandes d'une instruction, puis en extrait les différents arguments en ignorant les virgules éventuelles. Elle renvoie ensuite un tableau de chaînes de caractères, dans lequel chaque élément correspond à un argument de l'instruction. A noter que les espaces et tabulations séparant les opérandes ne sont pas enlevés, ce sera le travail de la fonction `enlever_espace`.

Pour chaque ligne contenant une instruction : on place notre pointeur sur le début de l'opcode après l'éventuelle étiquette en tête (on vérifie préalablement si une étiquette se trouve à cette adresse avec la fonction `chercher_adresse_dans_etiquettes`). On envoie alors notre chaine de caractères à la fonction `ajouter_instruction` qui va d'abord isoler l'opcode puis transformer les opérandes grâce aux fonctions `decouper_mots_virgules` et `enlever_espace`, qui permettent respectivement de séparer les opérandes en fonction des virgules, puis d'éliminer les espaces et tabulations sur les cotés de chaque opérandes. On obtient alors l'opcode sous forme d'une chaine de caractères, et un tableau de chaine de caractères, où chaque élément est un opérande de l'instruction. La fonction vérifie ensuite :

- Si la quantité d'instructions maximum est atteinte.
- Si l'instruction est vide.
- La validité de l'opcode ainsi que du nombre d'arguments fournis grace aux fonctions `numero_opcode` et `opcode_valide_arg` renvoyant respectivement le numéro opératoire de l'instruction et le nombre d'arguments attendus en fonctions de l'opcode.

Si aucune erreur n'est détectée à ce stade, la fonction va alors initialiser une Instruction avec tous les champs à 0, et va procéder par disjonction de cas en fonction de l'opcode (et son nombre d'arguments). Pour traiter les champ à 5 bits censé contenir les registres, on met en place les fonctions `conversion_char_long_reg` et `verif_reg` qui agissent en combinaison et permettent de :

- S'assurer du bon format de déclaration des registres.
- Isoler les caractères désignant le numéros des registres à l'aide de la fonction `tranche_str`.
- Convertit les numéros de registres (encore en format caractères) en décimaux via `strtol`.
- Détecter les caractères invalides.
- Gérer l'overflow et ramener la valeur sur 5 bits non signés.

Pour traiter le champ à 16 bits, on utilise les fonctions `conversion_char_long_src` et `verif_src` qui fonctionnent sur les mêmes bases et :

- S'assurent du bon format de déclaration de la valeur immédiate (si valeur immédiate il y a).
- Convertissent les numéros des valeurs immédiates en décimaux ou hexadécimales via `strtol`.
- Vérifient et renvoie l'adresse de l'étiquette dans le cas d'une instruction de saut grâce à `chercher_etiquette`.
- Détectent les caractères invalides
- Gèrent l'overflow et ramènent la valeur sur 16 bits signés.

> **Remarque** — Overflow
>
> Étant donné que nous stockons les valeurs des différents champs d'une instruction dans des variables de type `long`, nous distinguons deux types d'overflows :
> - Overflow critique : lorsque la valeur est strictement supérieure à 2 000 000 000 ou strictement inférieure à $-2\,000\,000\,000$.
> - Overflow classique : lorsque la valeur n'est pas en overflow critique, mais n'est pas représentable dans le format attendu (par exemple 5 bits non signés pour les registres, ou 16 bits signés pour le champ `Src2`).
>
> Dans le cas d'un overflow critique, la valeur est considérée comme invalide et les fonctions renvoient une erreur. Dans le second cas, nous avons choisi de ramener la valeur dans l'intervalle représentable par un simple modulo 2^k, où k correspond à la taille du champ. Pour plus d'indications sur le pourquoi du comment de l'overflow critique, voir 4)-Difficultés Rencontrées - Gestion de l'overflow critique.

Cas spécifique :
Voici comment l'on gère les instructions de transferts, qui attendent un format de la sorte : (rd)S ou (rn)S.
On s'assure tout d'abord du bon format de l'instruction en vérifiant que le premier caractère est `'('`. On parcourt la chaine avec un pointeur jusqu'à la première `')'` rencontrée (si l'on n'en trouve pas : Format invalide). On appelle `tranche_str` sur ce qui succède à `')'` qu'on traite ensuite comme une valeur immédiate classique, et `tranche_str` sur ce qui précède à `')'` (en partant du 2ème caractère pour ne pas prendre en compte `'('`), que l'on traite comme un registre classique.

Si erreurs il y a eu à n'importe quelle étape, la fonction `ajouter_instruction` renvoie `NULL`, sinon un pointeur vers la structure dont les champs contiennent :
- 0 si le champ n'est pas utilisé par l'instruction.
- La valeur rentrée par l'utilisateur si le champ est utilisé.

L'instruction est stocké dans le tableau à cet usage, et la taille du tableau est incrémentée. Une fois cette collecte finie sans erreurs, on peut finalement passer à la dernière étape.

2.5 Conversion binaire puis hexadécimale

A ce stade du programme, si tout c'est bien déroulé, nous nous retrouvons avec un tableau de pointeur, pointant vers des instructions contenant des champs sous forme de valeurs décimales. Pour les traduire en hexadécimales, nous allons d'abord les mettre sous format binaire. Les fonctions `conversion_binaire_sans_signe` et `conversion_binaire_signe` permettent de convertir une valeur décimales en une chaine de caractères binaire sur un certain nombre de bits, en complément à deux ou non. Pour chaque instruction, on convertit opcode, registres sur 5 bits non signés, Imm sur 1 bit non signé et Src2 sur 16 bits signés.

> **Remarque** — Complément à deux
>
> Le fait que Src2 soit signé en complément à deux n'empêche pas le programme d'interpréter des sauts ou des valeurs représentables sur 16 bits non signés mais non représentables sur 16 bits signés comme jmp #65535 par exemple (totalement légitime car notre machine virtuelle comporte 65535 cases mémoires). Il ne fera que la ramener par modulo 65536 sur la bonne plage de valeurs : -1. Or puisque c'est en complément à deux, -1 et 65535 s'écrivent de la même façon en binaire, et l'éxecuteur interprétera bien la bonne valeur.

En allouant une chaine à 32 bits en mémoire, on peut alors créer notre instruction en binaire selon le bon format en utilisant la fonction `copier_char_index` prenant deux chaines de caractères en entrée et un index, et copiant l'entièreté de l'une des chaine de caractères dans l'autre à partir de l'index.

> **Remarque** — Gestion des erreurs
>
> A ce stade, la façon dont on a géré les overflows nous assure que chaque champ de chaque instruction est bien représentable sur leur nombre de bits associé. De plus, les fonction s'assurant de la conversion en binaire renvoient une chaine de caractères de la taille du nombre de bits précisée. Ainsi, dès lors que les fonctions `copier_char_index` sont correctement implémentées et que l'échec éventuel d'une allocation dynamique est systématiquement traité, tout problème de taille incompatible entre les 5 chaînes de caractères et la chaîne de 32 bits est exclu.

Finalement, on utilise la fonction `Binaire_vers_hexa` pour convertir notre chaîne binaire en hexadécimal. Cette fonction s'appuie sur le fait qu'il est très simple de passer du binaire à l'hexadécimal en procédant par groupes de 4 bits. On découpe ainsi l'instruction binaire de 32 bits en 8 sous-chaînes de 4 bits, que l'on convertit une à une en hexadécimal, puis que l'on concatène dans une chaîne de 8 caractères, préalablement allouée en mémoire.
Pour créer notre fichier, on procède alors de la manière suivante :

- On ouvre un fichier `hexa.txt`.
- On parcourt notre tableau d'instructions une par une.
- On convertit notre instructions en binaire puis en hexadécimal.
- On écrit l'héxadécimal dans le fichier `hexa.txt` puis on saute une ligne.

Finalement, on ferme le fichier puis on libère l'espace mémoire alloué à toutes nos instructions et étiquettes.

2.6 Exemple

Illustrons le déroulement du programme :

Source assembleur

```
ici:  in r1
      jzs fin
      sub r1, r0, r1
      out r1
      jmp ici
fin:  hlt
```

1er passage : étiquettes

Adresse	Étiquette
0	ici
20	fin

2e passage : instructions

Adr	Op	Rd	Rn	Imm	Src2
0	28	1	0	0	0
4	22	0	0	1	20
8	4	1	0	0	1
12	29	1	0	0	0
16	21	0	0	1	0
20	31	0	0	0	0

Encodage binaire (32 bits)

```
11100 00001 00000 0 0000000000000000
10110 00000 00000 1 0000000000010100
00100 00001 00000 0 0000000000000001
11101 00001 00000 0 0000000000000000
10101 00000 00000 1 0000000000000000
11111 00000 00000 0 0000000000000000
```

Encodage hexadécimal

```
E0400000
B0010014
20400001
E8400000
A8010000
F8000000
```

FIGURE 1 – Exemple de traduction : assembleur → étiquettes → instructions → binaire → hexadécimal.

3 Documentation développeur du simulateur

Le programme correspondant à la partie du simulateur représente une machine virtuelle de 32 bits dont le but est d'exécuter les instructions de l'utilisateur, préalablement traduites en hexadécimal par l'assembleur.

3.1 Organisation générale : chargement et cycle d'exécution

L'architecture du simulateur repose sur deux étapes distinctes :

1. Initialisation de la mémoire
   Le programme commence par initialiser les composants matériels à zéro (à l'exception du flag `running`). Il parcourt ensuite le fichier `hexa.txt` généré par l'assembleur. Chaque ligne sous forme d'instruction encodée est découpée et chargée octet par octet dans le tableau représentant la mémoire centrale, reproduisant ainsi le processus de chargement d'un exécutable binaire en RAM.
2. Le cycle Lecture, Décodage et Action (Fetch,Decode,Execute)
   Une fois la mémoire initialisée, le processeur entre dans son cycle d'exécution infini :
   — Lecture (Fetch) : Le processeur récupère le mot de 32 bits situé à l'adresse mémoire pointée par le registre `PC`.
   — Décodage (Decode) : L'instruction brute est décomposée via des outils arithmétiques (masques et décalages simulés par divisions/modulos) pour isoler le code opération et les différents opérandes (registres sources, destination, valeurs immédiates).
   — Action (Execute) : Un sélecteur central redirige le flux d'exécution vers la logique appropriée (calcul arithmétique, transfert mémoire, branchement). Cette étape met à jour l'état de la machine, notamment les registres généraux et les drapeaux d'état (`Z`, `N`, `C`), avant d'incrémenter le `PC` pour l'instruction suivante.

Cette structure permet une isolation stricte où le simulateur ne manipule que l'état de la mémoire et des registres, sans aucune connaissance préalable du code source assembleur d'origine.

3.2 Fondements mathématiques de la manipulation Binaire

Pour réaliser ce simulateur, nous avons choisi de manipuler les instructions et les données en revenant aux propriétés de la représentation positionnelle des nombres entiers. Tout nombre entier N s'exprime dans une base B sous la forme d'un polynôme :

$$N = \sum_{i=0}^{k} d_i \cdot B^i$$

Dans notre architecture binaire ($B = 2$), nous utilisons cette définition pour isoler ou déplacer les composantes (d_i) du nombre. L'objectif est de ne pas utiliser d'opérateurs préconçus, mais de reconstruire la logique par le calcul arithmétique.

3.2.1 Extraction de données par l'arithmétique Modulaire

L'opération de base pour isoler une partie d'un nombre (comme récupérer un code opération ou un octet de donnée) est la division euclidienne. Soit un nombre N. La division euclidienne par une puissance de 2, notée 2^k, s'écrit :

$$N = q \cdot 2^k + r \quad \text{avec} \quad 0 \leq r < 2^k \tag{1}$$

Ici, le reste r correspond mathématiquement aux k bits de poids faible du nombre N. C'est sur ce principe que reposent nos fonctions d'extraction :

$$\texttt{masque}(N, k) = N \pmod{2^k}$$

Cette opération filtre la partie du nombre supérieure ou égale à 2^k (la partie $q \cdot 2^k$), ne conservant que les bits inférieurs.

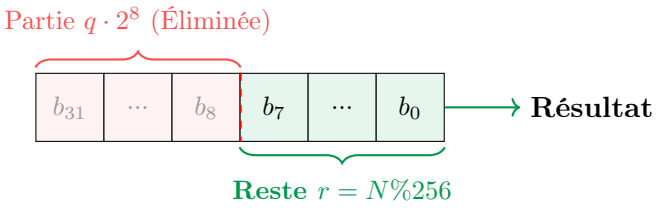


FIGURE 2 – L'opération Modulo vue comme un filtre arithmétique

3.2.2 Implémentation des masques spécifiques

Nous avons implémenté ce principe à travers quatre fonctions utilitaires, adaptées aux différentes tailles de données manipulées par le processeur.

`masque_1bit` — Modulo 2

Objectif : Isoler un bit unique (drapeau ou booléen).
Le bit 16 de l'instruction détermine le type d'opérande (Immédiat ou Registre). Mathématiquement, la parité d'un nombre (pair/impair) dépend uniquement de son bit de poids 0.

$$b_0 = N \pmod 2$$

Exemple : Si $N = 13$ (binaire `1101`), alors $13 \pmod 2 = 1$. Le dernier bit est isolé.

`masque_5bits` — Modulo 32

Objectif : Adressage des registres (R_0 à R_{31}).
Le processeur disposant de 32 registres, l'index doit impérativement être compris entre 0 et 31.

$$\text{Index} = Val \pmod{2^5}$$

Exemple : Si une instruction erronée demande le registre 33 (`100001`), l'opération $33 \pmod{32}$ renvoie 1. On accède ainsi au registre R_1, évitant un dépassement de tableau.

`masque_octet` — Modulo 256

Objectif : gestion de la mémoire (byte-addressing).
La mémoire étant un tableau de `unsigned char`, les valeurs doivent être comprises entre 0 et 255. Lors du stockage, il est nécessaire de tronquer la partie supérieure à 2^8.
Exemple : Pour l'instruction `STB` avec une valeur $N = 300$ (`0x12C`), le calcul $300 \pmod{256}$ donne 44 (`0x2C`). Seul l'octet de poids faible est écrit en mémoire.

`masque_16bits` Modulo 65536

Objectif : Extraction des valeurs immédiates.
Les constantes sont codées sur 16 bits. Cette fonction élimine les 16 bits de poids fort de l'instruction pour ne conserver que la valeur brute S. Cette étape permet de manipuler la valeur non signée avant d'appliquer l'extension de signe logicielle requise pour les registres 32 bits.
Exemple : Pour une valeur -1 (codée `0xFFFF`), ce masque assure que l'on récupère `65535` avant l'extension de signe.

3.2.3 Déplacement de bits par changement d'échelle

Une fois les bits isolés, il est souvent nécessaire de changer leur position (leur poids). Dans un système positionnel, cela correspond à une multiplication ou une division par la base.

3.2.4 Décalage à droite : La division entière

La fonction `decalage_droite` permet de ramener des bits de poids fort vers les poids faibles (par exemple, pour lire le code opération situé en début d'instruction). Mathématiquement, la division entière d'un nombre N par 2 décale tous ses chiffres d'un rang vers la droite :

$$\left\lfloor \frac{\sum b_i 2^i}{2} \right\rfloor = \sum b_i 2^{i-1}$$

Le bit de poids 2^0 se retrouve avec un poids 2^{-1}. Comme nous travaillons sur des entiers, cette partie fractionnaire disparaît (c'est le reste de la division).

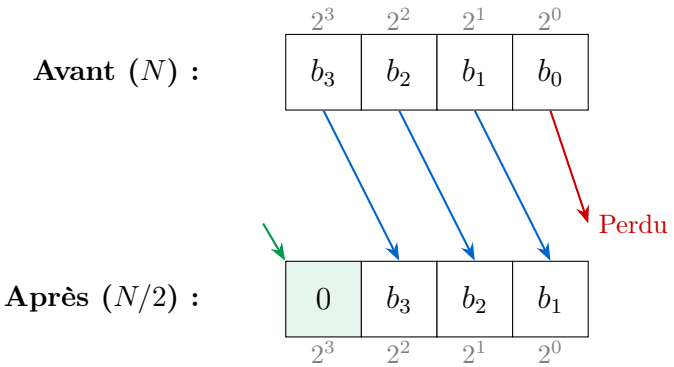


FIGURE 3 – Visualisation de la division par 2 : réduction de poids des bits

3.2.5 Décalage à gauche : la multiplication

L'opération inverse, `decalage_gauche`, est réalisée via une multiplication par 2. Cette opération est indispensable pour la reconstruction de la mémoire *Little Endian* (voir section 2). Elle permet de déplacer un octet lu à l'adresse $N+1$ (poids faible) vers une position plus haute (poids 2^8) dans le registre final.

Mathématiquement, multiplier le polynôme par 2 augmente l'exposant de chaque terme :

$$2 \times \left(\sum b_i 2^i \right) = \sum b_i 2^{i+1}$$

Le bit qui était en position 2^i se retrouve en position 2^{i+1}. Un zéro apparaît nécessairement en position 2^0.

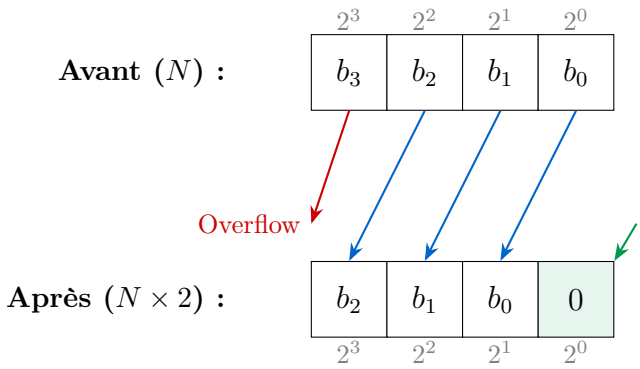


FIGURE 4 – Visualisation de la multiplication par 2 : augmentation de poids des bits

3.3 Architecture mémoire et modélisation des données

Pour simuler la mémoire de l'ordinateur, nous ne pouvons pas nous reposer sur les types de haut niveau classiques. La mémoire est une séquence brute d'octets sans type prédéfini. Cette section explique comment nous avons structuré cet espace et comment nous gérons les problèmes d'adressage.

3.3.1 Modélisation physique : le tableau linéaire

Conformément à l'architecture de Von Neumann, les instructions et les données cohabitent dans le même espace. Pour respecter la contrainte d'une mémoire de 64 Ko (65 536 octets) adressable à l'octet, nous avons modélisé cet espace par un tableau statique en C :

```
unsigned char memoire[65536];
```

Nous avons spécifiquement choisi le type `unsigned char` pour deux raisons. D'une part, sa taille est exactement de 1 octet (8 bits), ce qui correspond à l'unité adressable de la machine. D'autre part, le type `unsigned` (valeurs 0 à 255) nous évite les erreurs d'interprétation liées aux bits de signe lors des opérations logiques.

3.3.2 Initialisation et chargement du programme

Avant de lancer l'exécution, le simulateur doit charger le contenu du fichier hexadécimal (`hexa.txt`) dans notre tableau mémoire. Cette étape réalise l'inverse de la lecture, nous devons découper un mot de 32 bits en 4 octets distincts pour les stocker.

Notre fonction `charger_programme` lit chaque instruction brute et l'écrit en mémoire selon le format *Little Endian*. Nous utilisons nos fonctions arithmétiques pour effectuer cette décomposition, l'octet de poids faible est isolé via `masque_octet` et stocké à l'adresse courante A, tandis que les octets suivants sont obtenus par des divisions successives $(2^8, 2^{16}, 2^{24})$. Enfin, nous incrémentons l'adresse d'écriture de 4 pas pour passer à l'instruction suivante.

```
// Principe de décomposition dans charger_programme
memoire[adr]   = masque_octet(instruction);
memoire[adr+1] = masque_octet(decalage_droite(instruction, 8));
memoire[adr+2] = masque_octet(decalage_droite(instruction, 16));
memoire[adr+3] = masque_octet(decalage_droite(instruction, 24));
```

Cela permet de garentir que les données restent intègres, peu importe l'architecture de la machine sur laquelle tourne le simulateur.

3.3.3 Lecture Little Endian et reconstruction

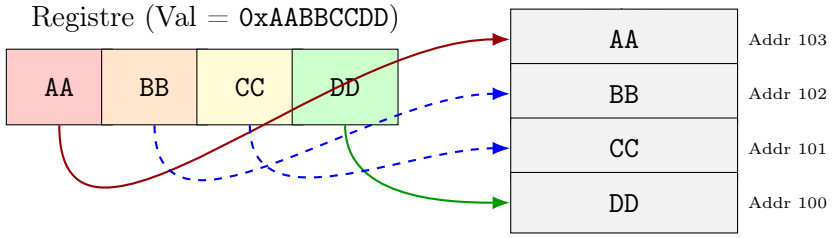
Une fois le programme en mémoire, le processeur doit pouvoir relire les instructions. L'énoncé impose le format **Little Endian**, ce qui signifie que l'octet de poids faible se trouve à l'adresse basse (A) et l'octet de poids fort à l'adresse haute ($A+3$).

Pour lire un mot de 32 bits, notre fonction `lire_mot_memoire` doit reconstruire la valeur complète à partir de ces 4 octets dispersés ce que nous faisons par le calcul :

$$V = M[A] + (M[A+1] \times 2^8) + (M[A+2] \times 2^{16}) + (M[A+3] \times 2^{24}) \tag{2}$$

Concrètement, ces multiplications par des puissances de 2 sont effectuées par notre fonction `decalage_gauche`.

1. Stockage en mémoire (Little Endian)



2. Lecture et reconstruction arithmétique

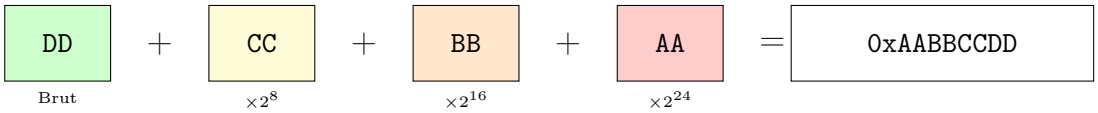


FIGURE 5 – Cycle de vie d'une donnée : Stockage éclaté en RAM (Haut) et Reconstruction mathématique lors de la lecture (Bas).

3.3.4 Alignement et compteur ordinal (PC)

Puisque chaque instruction est codée sur 32 bits (4 octets), le processeur doit avancer de 4 cases mémoire à chaque cycle *Fetch*. C'est pourquoi nous incrémentons systématiquement le compteur ordinal (PC) après chaque lecture :

```
PC = PC + 4;
```

Cela implique une contrainte d'alignement, les adresses des instructions seront toujours des multiples de 4 ($0, 4, 8, 12...$).

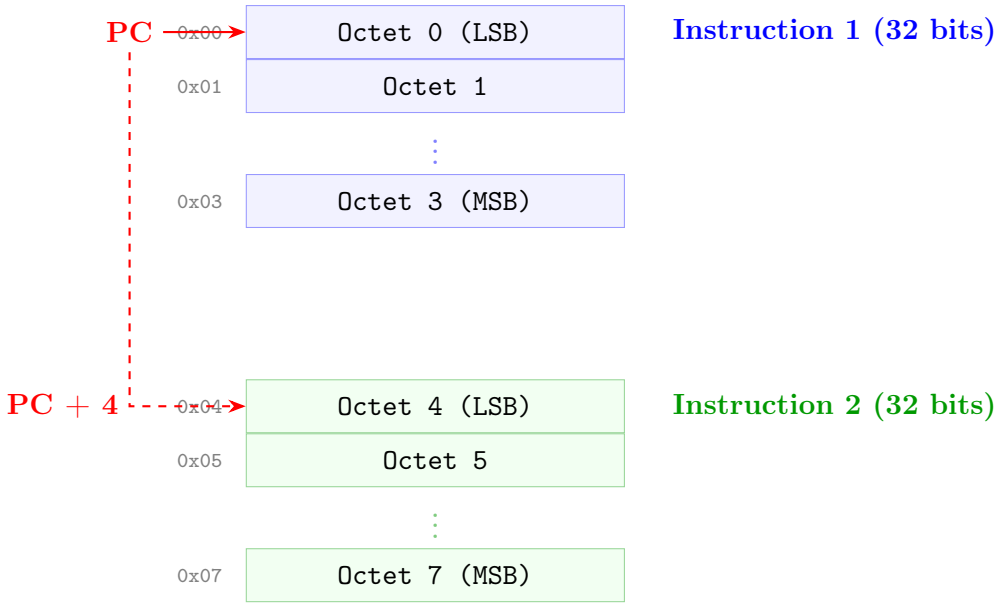


FIGURE 6 – Alignement mémoire : Le PC avance par pas de 4 octets.

3.3.5 Typage et extension de signe

La gestion des valeurs négatives est un point critique en C. Si nous nous contentons d'extraire une valeur immédiate de 16 bits comme `0xFFFF` (-1) pour la placer dans un `int` de 32 bits, le résultat sera `65535` (positif) car le bit de signe n'est pas propagé.

Pour corriger cela, nous utilisons une technique de "double cast" qui force le compilateur à effectuer l'extension de signe :

```
// Le cast (short) force l'interprétation signée sur 16 bits
valeur_etendue = (int)(short)valeur_16_bits;
```

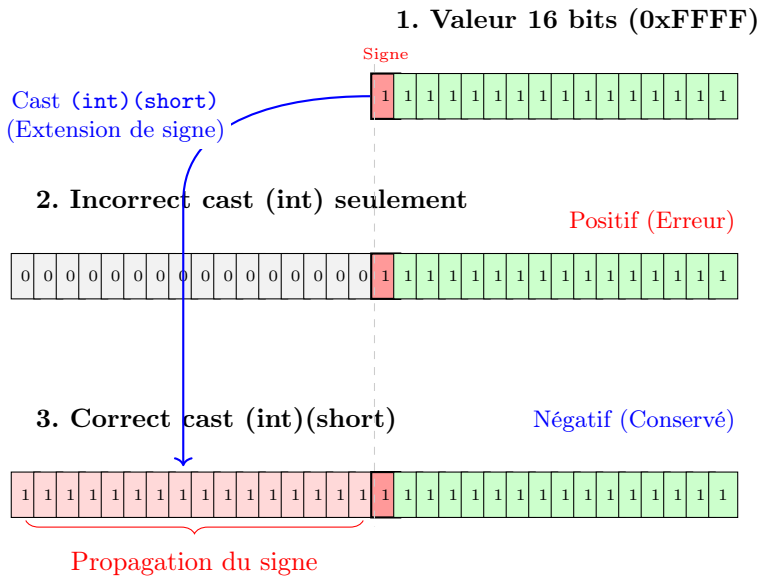


FIGURE 7 – Mécanisme de l'extension de signe : comparaison entre zéro-extension (incorrect pour les nombres négatifs) et extension de signe.

Ce mécanisme n'est pas limité aux valeurs immédiates. Il est également nécessaire pour les instructions de chargement partiel (`LDB` et `LDH`) afin de garantir que les données chargées

conservent leur signe une fois placées dans les registres 32 bits.

3.3.6 Robustesse et sécurité

Pour éviter tout arrêt brutal (*Segmentation Fault*), nous avons sécurisé chaque accès mémoire (`LDB`, `STW`, etc.) par une vérification des bornes.

```
if (adresse < 0 || adresse > 65532) { running = 0; break; }
```

3.4 Le cycle d'instruction

Une fois la mémoire initialisée, le simulateur entre dans sa boucle principale. Cette boucle infinie reproduit le fonctionnement cyclique de l'architecture de Von Neumann. À chaque itération, nous simulons un cycle d'horloge complet qui enchaîne séquentiellement le chargement de l'instruction, son décodage et enfin son exécution.

3.4.1 Phase de chargement (fetch)

La première étape consiste à récupérer l'instruction courante. Nous utilisons pour cela le registre PC (Program Counter) qui pointe vers l'adresse mémoire à lire. L'appel à notre fonction `lire_mot_memoire` nous renvoie l'instruction complète de 32 bits, correctement reconstruite.

Dès la lecture terminée, nous faisons avancer le compteur ordinal de 4 octets ($PC \leftarrow PC + 4$). Cette mise à jour immédiate est nécessaire pour garantir que, sauf indication contraire, le processeur passera bien à l'instruction suivante au prochain tour de boucle.

3.4.2 Phase de décodage (decode)

Le processeur dispose maintenant d'un mot brut de 32 bits qu'il doit interpréter. Pour identifier l'action à réaliser, nous devons isoler les différents champs binaires (code opération, registres, immédiat). Nous utilisons ici les opérations arithmétiques de décalage et de masquage définies dans la première partie.

3.4.3 Extraction des champs

Pour récupérer une information spécifique, comme le code opération situé sur les 5 bits de poids fort, nous procédons toujours de la même manière. Nous décalons d'abord les bits vers la droite pour les amener en position basse, puis nous appliquons un masque pour éliminer tout ce qui ne concerne pas le champ visé. Nous répétons cette logique pour extraire les identifiants des registres source et destination.

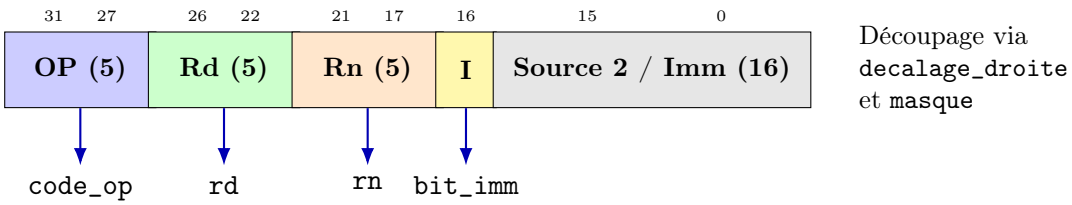


FIGURE 8 – Le Décodage, l'instruction brute est découpée bit par bit pour remplir les variables de contrôle.

3.4.4 Résolution de l'opérande "S"

Le traitement du second opérande (source 2) est particulier car sa nature change selon le contexte. Pour simplifier le code de la phase d'exécution, nous avons choisi de résoudre cette ambiguïté dès le décodage en analysant le bit 16.

Si ce bit indique une valeur immédiate, nous extrayons les 16 bits de poids faible et nous leur appliquons l'extension de signe pour obtenir un entier valide sur 32 bits. Dans le cas contraire, nous utilisons les 5 bits pour lire directement la valeur contenue dans le registre correspondant. Cette étape nous permet de fournir à la phase d'exécution une variable unifiée `S_valeur`, prête à l'emploi quel que soit son type d'origine.

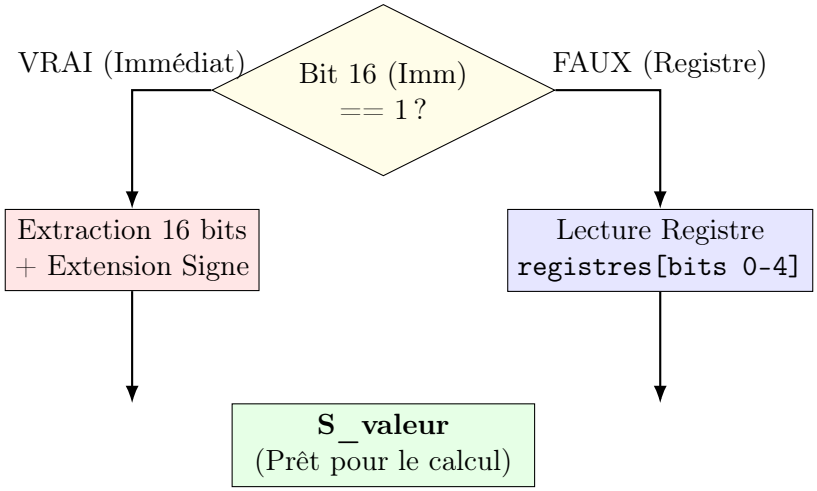


FIGURE 9 – Logique de l'opérande S : Le bit 16 agit comme un multiplexeur pour choisir la source de la donnée.

3.5 Phase d'exécution (execute)

Pour diriger le flux d'exécution, nous utilisons une structure de contrôle `switch` basée sur le code opération. Nous avons organisé les instructions en trois catégories principales.

Les opérations arithmétiques et logiques (codes 0 à 7) effectuent les calculs et stockent le résultat dans le registre destination. C'est également ici que nous traitons les cas d'erreurs mathématiques comme la division par zéro. Les instructions de transfert mémoire (codes 12 à 17) gèrent quant à elles les échanges entre les registres et la RAM, en vérifiant systématiquement la validité des adresses calculées. Enfin, les instructions de saut (codes 21 à 27) permettent de rompre la séquentialité du programme en modifiant directement la valeur du PC si la condition demandée est remplie.

3.5.1 Gestion du registre d'etat (Flags)

Nous avons pris le parti de ne pas modifier les drapeaux au cœur des opérations de calcul, mais de centraliser leur mise à jour à la fin du cycle d'instruction. Cette approche nous assure que l'état du processeur reste cohérent. Les drapeaux Z (Zéro) et N (Négatif) sont ainsi déduits de la valeur finale du registre destination.

Le drapeau de retenue C nécessite cependant une attention particulière. Si les opérations arithmétiques définissent naturellement sa valeur, les instructions de transfert et d'entrées-sorties sont censées affecter le registre d'état sans pour autant produire de retenue mathématique. Pour éviter de conserver par erreur une retenue provenant d'un calcul antérieur, nous forçons sa remise à zéro pour toutes les opérations non-arithmétiques. L'instruction RND fait exception et laisse les drapeaux intacts.

3.6 Robustesse et gestion des cas limites

Concevoir un simulateur ne consiste pas uniquement à implémenter le fonctionnement normal du processeur. Il est tout aussi important de garantir que le programme ne plantera pas si

l'utilisateur lui fournit du code assembleur erroné. Cette section décrit les sécurités que nous avons intégrées pour rendre notre machine virtuelle stable et capable de gérer proprement les erreurs d'exécution.

3.6.1 Sécurité d'accès mémoire (segmentation fault)

Notre architecture repose sur une mémoire de 64 Ko. En langage C, les tableaux ne possèdent pas de mécanisme de protection natif : si le programme tente de lire ou d'écrire en dehors des cases allouées, cela provoque une erreur de segmentation qui arrête brutalement le simulateur.

Pour pallier ce problème, nous avons ajouté une vérification systématique avant chaque instruction d'accès mémoire (`LDB`, `LDW`, `STB`, `STW`). Le principe est simple, si l'adresse calculée dépasse les bornes autorisées, nous interrompons l'exécution proprement et affichons un message d'erreur explicite, plutôt que de laisser le simulateur "crasher".

Implémentation (Exemple pour un accès mot 32 bits) :

```
if (adresse < 0 || adresse > 65532) {
    printf("Erreur : Tentative de lecture d'instruction hors de la
       memoire (Adresse %d invalide)\n", adresse);
    return 0;
}
```

3.6.2 Robustesse arithmétique

Les calculs mathématiques peuvent eux aussi provoquer des arrêts inattendus. Nous avons identifié et sécurisé plusieurs cas limites au niveau des fonctions de calcul.

Le premier cas concerne l'instruction de division (`DIV`). Nous vérifions systématiquement que le diviseur n'est pas nul avant de lancer l'opération. Nous traitons également un cas spécifique lié à la représentation en complément à deux : la division de la valeur minimale (`INT_MIN`) par -1. Théoriquement, le résultat devrait être positif, mais il dépasse la capacité d'un entier signé sur 32 bits, ce qui provoque généralement un arrêt immédiat du programme par le système. Nous détectons ce couple de valeurs pour éviter l'arrêt du simulateur.

3.6.3 Décalages complexes (SHR)

L'instruction de décalage logique à droite (`SHR`) est plus subtile qu'elle n'y paraît, car le nombre de bits de décalage est variable. Nous avons dû gérer deux situations particulières pour rester fidèles au comportement matériel attendu.

Premièrement, si l'utilisateur demande un décalage supérieur à 32 bits, le résultat est forcé à 0. Cela nous évite les comportements indéfinis du C sur les grands décalages. Deuxièmement, si la valeur de décalage est négative, l'opération doit s'inverser et devenir un décalage à gauche. Notre code détecte ce signe négatif et redirige dynamiquement l'exécution vers la fonction `decalage_gauche`.

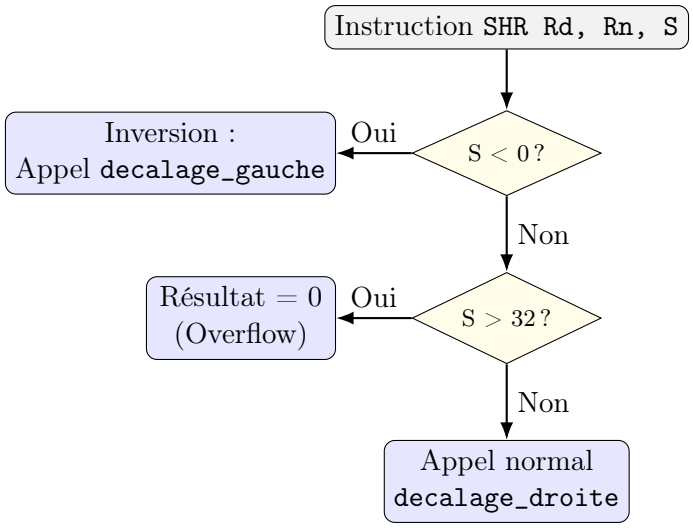


FIGURE 10 – Logique décisionnelle de l'instruction SHR traitant les décalages négatifs et les débordements de capacité.

3.6.4 Gestion "Propre" du registre d'etat

Un problème subtil s'est posé avec le drapeau de retenue C (Carry). Normalement la retenue n'est modifiée que par les calculs arithmétiques. Or, les spécifications indiquent que les instructions de transfert (`LD*`, `ST*`) mettent à jour le registre d'état, sans pour autant générer de retenue mathématique.

Notre première implémentation ne touchait simplement pas au Carry lors des transferts mémoire. Donc après un `LDW` suivi d'un `JCS`, le processeur pouvait sauter en se basant sur une retenue provenant d'une addition effectuée trois instructions plus tôt. L'état devenait "sale" et les sauts conditionnels devenaient imprévisibles. Nous forçons alors la remise à zéro du Carry pour toutes les instructions non-arithmétiques.

4 Difficultés rencontrées et potentielles améliorations

4.1 Difficultées rencontrées

4.1.1 Fuite mémoire

Lors de l'implémentation du traducteur assembleur, nous avons rencontré plusieurs problèmes liés aux fuites de mémoire. En effet, nous utilisions à de nombreuses reprises la fonction `malloc`, notamment dans les fonctions `decouper_mots_virgules`, `tranche_str` ou `conversion_binaire_signe`, sans jamais libérer ensuite la mémoire allouée. Ce problème a également été rencontré lors du stockage dynamique des étiquettes et des instructions dans des tableaux de pointeurs.

Nous avons identifié ces fuites assez tard dans le développement, ce qui nous a ammenés à revoir en profondeur une partie du code afin de nous assurer que chaque allocation était correctement suivie d'un `free` correspondant.

4.1.2 Gestion de l'Overflow critique

`Rappel :`
Étant donné que nous stockons les valeurs des différents champs d'une instruction dans des variables de type `long`, nous distinguons deux types d'overflows :

— Overflow critique : lorsque la valeur est strictement supérieure à 2 000 000 000 ou strictement inférieure à $-2\,000\,000\,000$.
— Overflow classique : lorsque la valeur n'est pas en overflow critique, mais n'est pas représentable dans le format attendu (par exemple 5 bits non signés pour les registres, ou 16 bits signés pour le champ `Src2`).

Dans le cas d'un overflow critique, la valeur est considérée comme invalide et le programme s'arrête. Dans le second cas, nous avons choisi de ramener la valeur dans l'intervalle représentable par un simple modulo 2^k, où k correspond à la taille du champ.

L'implémentation de l'overflow classique se fait relativement simplement à l'aide d'une boucle. En revanche, la gestion de l'overflow critique s'est révélée plus subtile.

Tout d'abord, le problème s'est présenté sous la forme suivante :
Étant donné que nous convertissons les chaînes de caractères en `long` à l'aide de la fonction `strtol`, comment détecter qu'une valeur saisie par l'utilisateur n'est pas stockable dans un `long` ? #LONG_MAX + 1 par exemple.

À première vue, une solution pourrait consister à stocker temporairement la valeur dans un type plus grand afin de repérer les dépassements. Cependant, cela ne ferait que déplacer le problème, puisque des valeurs pourraient également dépasser la capacité de ce type élargi.

Au cours de nos recherches, nous avons découvert l'utilisation de la bibliothèque`errno` qui, couplé à `strtol`, permet de détecter directement un dépassement de capacité. Toutefois, son fonctionnement nous paraissait peu mystique, et nous avons choisi de ne pas l'utiliser.

Nous avons alors observé que lorsque `strtol` reçoit en entrée une chaîne représentant un nombre non stockable dans un `long`, la fonction renvoie `LONG_MAX` si le nombre est positif, et `LONG_MIN` dans le cas contraire. Nous avons donc décidé de fixer une limite d'overflow critique strictement comprise dans l'intervalle stockable par un `long` (ici $[-2\,000\,000\,000;\ 2\,000\,000\,000]$), afin de pouvoir détecter ces dépassements tout en refusant certaines valeurs immédiates pourtant représentables. Ce choix repose sur l'hypothèse qu'un utilisateur n'a que très peu de raisons de saisir des valeurs de cette taille dans le cadre de notre assembleur.

4.1.3 Reconstruction arithmétique de la logique binaire

La principale difficulté a résidé dans la non-utilisation des opérateurs binaires natifs du langage C (`«`, `»`, `&`, `|`). Habituellement, la simulation d'un processeur repose massivement sur ces opérateurs pour le décodage des instructions. Nous avons dû repenser chaque opération logique sous sa forme arithmétique, un décalage devient une multiplication ou une division par une puissance de 2, et un masque binaire devient une opération modulo. Cette approche nous a cependant permis de revenir aux fondamentaux, et de comprendre plus en profondeur la gestion des bits d'un point de vue mathématiques.

4.1.4 Gestion du signe et extension de signe

Un autre problème concernait la manipulation des nombres signés lors des transferts entre la mémoire (8 bits) et les registres (32 bits). En langage C, la conversion d'un type plus petit vers un type plus grand peut parfois interpréter la valeur comme positive (unsigned) alors qu'elle devrait être négative. Par exemple, lors de l'exécution d'une instruction immédiate ou d'un saut relatif négatif, il a fallu s'assurer que l'extension de signe s'effectuait correctement. Nous avons dû utiliser explicitement des conversions de type (casts) tels que `(signed char)` ou `(short)` avant de stocker les valeurs dans les registres `int`, afin de garantir que la valeur hexadécimale `0xFF` soit bien interprétée comme `-1` et non comme `255` par l'UAL. Heureusement c'est une difficulté que nous avions anticipée avant la conception du programme, donc il n'y a pas eu besoin de tout remodifier, seulement de se creuser la tête pour savoir comment la résoudre.

4.1.5 Lisibilité du code

Pendant une grande partie du développement du projet, nous avions regroupé l'ensemble des fonctions ainsi que le `main` dans un seul et même fichier. Cependant au cours des interminables débugages, nous avons constaté que travailler sur un programme d'environ 1300 lignes, avec de fréquents allers-retours entre le `main` et les différentes fonctions, nous faisait perdre un temps considérable et nuisait énormément à la lisibilité du code.

Nous avons alors décidé de structurer le projet en plusieurs fichiers et `headers`. Cette structuration fut difficile : à partir d'un programme en un seul bloc, il a fallu identifier les dépendances, regrouper les fonctions par catégories, et répartir correctement les déclarations entre les différents modules.

4.2 Potentielles améliorations

4.2.1 Lisibilité

Malgré nos efforts via les `headers`, le `main` reste relativement conséquent et n'est pas toujours très lisible. Il aurait sans doute été possible de le réduire davantage en implémentant plus de fonctions dédiés, par exemple pour gérer la vérification de la taille des étiquettes ou l'exécution. L'utilisation des bitwise dont nous avons parlé précédemment aurait aussi amelioré considérablement la lisibilité du code et peut etre meme sa robustesse, mais nous avons fais le choix de ne pas les utiliser car nous voulions comprendre le plus possible la logique derrière les operations sur les bits.

4.2.2 Pertinence des erreurs

Dans certains cas, quand l'utilisateur saisit une entrée vraiment incorrecte, le programme échoue bien à la compilation et affiche une erreur, mais le message retourné n'est pas toujours le plus pertinent. Cela vient du fait que l'on essaie de déterminer précisément la cause du problème, mais lorsque l'entrée est trop incohérente, il n'y a parfois pas assez d'informations pour faire un diagnostic fiable. Une amélioration serait donc de prévoir plusieurs niveaux d'erreurs : un message général dans les cas extrêmes, puis un message plus précis dès que c'est possible.

4.2.3 Utilisation de strtol

L'utilisation de la fonction `strtol` de la bibliothèque standard `stdlib.h` permettant de convertir une chaîne de caractères en `long` en base décimale ou hexadécimale, représentait pour nous un choix de simplicité. Toutefois dans une démarche visant à limiter au maximum l'usage de fonctions déjà existantes, il aurait été judicieux de réimplémenter cette conversion nous-mêmes.

4.2.4 Robustesse

Le fait d'avoir cherché à limiter au maximum l'utilisation des fonctions issues des bibliothèques standards pose toutefois des questions de sécurité. En effet, certaines de nos fonctions outils ne sont pas suffisamment robustes si on les utilise en dehors du cadre précis pour lequel elles ont été conçues dans ce projet. Une amélioration possible serait donc de renforcer leur fiabilité afin de pouvoir les réutiliser plus facilement dans d'autres projets, quel que soit le contexte.

4.2.5 Fonctionnalités délaissées

Voici quelques fonctionnalités en vrac que l'on comptait ajouter et qui nous semblaient plutôt simples à implémenter, mais qui n'ont finalement pas pu être intégrées pour plusieurs raisons, notamment la difficulté de mise en place et la peur de déclencher des erreurs en cascade.

— Détection et Warning des étiquettes non utilisées par la Traduction Assembleur.

— Autoriser les commentaires sur la même ligne que les instructions (actuellement, une ligne entière doit leur être réservée).
— Renvoyer exactement la ligne et l'instruction qui fait planter le programme pour le simulateur (par exemple pour la boucle infini).

Documentation Utilisateur
Simulation d'une machine virtuelle

Architecture des ordinateurs – MIDO L2 2025/2026

Dahan Benjamin et Baudouin Rémy

Table des matières

Introduction

Bienvenue dans la documentation Utilisateur du projet de simulation d'une machine virtuelle programmée en C, celle-ci est destinée à tout public voulant simplement exécuter notre programme sur leur machine.

Dans ce projet, nous avons codé en C un assembleur et un simulateur pour une machine virtuelle fictive composée d'une mémoire (64 Ko) et d'un microprocesseur. Notre programme traduit un fichier assembleur en code machine hexadécimal 32 bits, puis exécute ce programme instruction par instruction en simulant le fonctionnement du processeur et de ses registres.
Voir dev.pdf pour plus d'indications concernant son implémentation et développement.

Ce document sera séparé en 2 parties :
- Compilation et exécution.
- Langage Assembleur et règles.

Tout usage à des fins commerciales est autorisé, mais nous nous éxonérons de toute responsabilités en cas de problème (explosion du processeur...).
Nous vous souhaitons bonne lecture.

Utilisation de l'IA

Une IA a été utilisée dans la documentation pour créer les figures et tableaux en LaTeX.

1 Compilation et exécution

Ce programme a été entièrement codé en C, il vous faudra donc vous munir d'un environnement C sur votre machine pour l'utiliser. Le programme est séparé en plusieurs fichiers (avec leurs headers respectifs), stockez les tous au même endroit. Ensuite, compilez les tous en même temps dans le terminal.
Vous pouvez utiliser la commande ci dessous :
gcc -Wall main.c Constantes.c Fonctions_outils.c Structures_et_Fonctions_etiquettes.c Structures_et_Fonctions_instructions.c Conversion_Instruction_Hexa.c gestion_memoire.c -o simulateur
Puis exécutez le avec :
./simulateur fichier.txt
ou fichier.txt est le fichier contenant les instructions en langage assembleur.
Attention, à l'issu de son exécution, le programme va créer un fichier hexa.txt à l'endroit endroit où le projet est stocké, risquant d'écraser les donnés du fichier hexa.txt si il en existe déja.

2 Instructions

2.1 Format des instructions

Le fichier texte contenant le programme assembleur doit respecter les règles suivantes :
- Une seule instruction par ligne.
- Une ligne peut commencer par des espaces et/ou des tabulations.
- Une étiquette peut apparaître en début de ligne. Elle doit se terminer par `:` et peut contenir des lettres (majuscules ou minuscules), des chiffres et le caractère `_`.
- Après une éventuelle étiquette, des espaces et/ou des tabulations peuvent apparaître.
- Le code opération doit être écrit en minuscules.
- Les opérandes doivent être séparées par des virgules (des espaces et/ou tabulations peuvent éventuellement les entourer).
- Il ne faut pas mettre de virgule après la dernière opérande.

2.2 Plage de valeurs acceptées dans les instructions

Voici les plages de valeurs autorisées pour les opérandes :
- Un registre doit être compris entre 0 et 31.
- Une adresse de saut doit être comprise entre 0 et 65535.
- Une valeur immédiate utilisée dans un calcul arithmétique doit être comprise entre -32768 et 32767.
- Toute valeur immédiate ou numéro de registre strictement supérieur à 2 000 000 000 ou strictement inférieur à -2 000 000 000 provoque une erreur (Overflow critique).

Si une valeur est en dehors des plages indiquées, mais ne correspond pas à un Overflow critique, le comportement du programme peut devenir imprévisible et aucun avertissement ne sera affiché (voir `dev.pdf` pour plus d'informations sur la gestion des overflows).

2.3 Règles à suivre

- La longueur maximale d'une étiquette est de 51 caractères.
- Le nombre maximal d'instructions autorisées est de 4096.
- Le nombre maximal d'étiquettes autorisées est de 1024.
- La longueur maximale d'une ligne est de 256 caractères.
- Une ligne contenant uniquement une étiquette (sans instruction) n'est pas acceptée.
- Un commentaire doit être indiqué par le caractère `;` et une ligne entière doit lui être réservée.
- Si une étiquette porte le même nom qu'un registre (par exemple `r30`), il ne sera plus possible d'utiliser une instruction de saut sur l'adresse contenue dans le registre (il sera interprété comme une référence à l'étiquette).

3 Dictionnaire Assembleur

Les bits Z, C et N évoqués dans la partie `instructions de saut` ci dessous sont mis à jour après l'exécution d'une instruction, suivant le résultat de l'instruction :
- Z est mis à 1 si le résultat est nul, à 0 sinon.
- C est mis à 1 s'il y a une retenue, à 0 sinon.
- N est la recopie du bit de poids fort du résultat.

Seules les instructions de saut n'affectent pas les bits Z,N,C.

Arithmétique et logique

Forme de S : $S \in \{rm,\ \#n,\ \#hn\}$

Syntaxe	Définition / effet
`or rd, rn, S`	`rd` ← `rn` OR `S`.
`and rd, rn, S`	`rd` ← `rn` AND `S`.
`xor rd, rn, S`	`rd` ← `rn` XOR `S`.
`add rd, rn, S`	`rd` ← `rn` + `S`.
`sub rd, rn, S`	`rd` ← `rn` - `S`.
`mul rd, rn, S`	`rd` ← `rn` × `S`
`div rd, rn, S`	`rd` ← `rn` / `S` (division entière).
`shr rd, rn, S`	`rd` ← décalage de `rn` de `S` bits à droite (à gauche si `S`<0).

Transferts mémoire

Forme de S : $S \in \{rm,\ \#n,\ \#hn\}$ (adresse effective : `rn+S` ou `rd+S` selon l'instruction)

Syntaxe	Définition / effet
`ldb rd, (rn)S`	Charge 1 octet depuis l'adresse `rn+S` dans `rd`
`ldh rd, (rn)S`	Charge 2 octets depuis `rn+S` dans `rd`
`ldw rd, (rn)S`	Charge 4 octets depuis `rn+S` dans `rd`.
`stb (rd)S, rn`	Stocke 1 octet : mémoire[`rd+S`] ← octet faible de `rn`.
`sth (rd)S, rn`	Stocke 2 octets : mémoire[`rd+S`] ← 16 bits faibles de `rn`.
`stw (rd)S, rn`	Stocke 4 octets : mémoire[`rd+S`] ← `rn`.

Sauts

Forme de S : $S \in \{rm,\ \#n,,\ \#hn,$ `etiq`$\}$

Syntaxe	Définition / effet
`jmp S`	Saut inconditionnel : `PC` ← `S`.
`jzs S`	Saut si Z=1 : `PC` ← `S`.
`jzc S`	Saut si Z=0 : `PC` ← `S`.
`jcs S`	Saut si C=1 : `PC` ← `S`.
`jcc S`	Saut si C=0 : `PC` ← `S`.
`jns S`	Saut si N=1 : `PC` ← `S`.
`jnc S`	Saut si N=0 : `PC` ← `S`.

Entrées / sorties

Forme de S : aucune (pas de champ Src2)

Syntaxe	Définition / effet
`in rd`	Lit une valeur au clavier et la place dans `rd`. Met à jour Z,C,N.
`out rd`	Affiche en décimal la valeur de `rd`.

Divers

Forme de S : dépend de l'instruction (voir ci-dessous)

Syntaxe	Définition / effet
`rnd rd, rn, S`	*Forme de S :* $S \in \{rm,\ \#n,\ \#hn\}$. Met dans `rd` un entier aléatoire entre `rn` et `S`-1 inclus. Met à jour Z,C,N.
`hlt`	Termine l'exécution du programme.

3.1 Exemples

```
; Programme 1 : Opposé d'une valeur (arrêt si 0)
ici: in r1
jzs fin
sub r1, r0, r1
out r1
jmp ici
fin: hlt
```

```
; Programme 2 : Affiche les entiers de 1 à 10
add r1,r0, #1
add r2, r0, #10
loop: out r1
add r1, r1, #1
sub r2, r2, #1
jzs fin
jmp loop
fin: hlt
```

FIGURE 1 – Exemples de programmes en langage Assembleur

4 Fonctionnement du Simulateur

Une fois le programme lancé via la commande `./simulateur`, le processus de simulation commence. Cette section décrit les interactions possibles et les messages que vous pouvez rencontrer.

4.1 Cycle de vie de l'exécution

Dès le lancement du programme, le simulateur va entrer dans le cycle suivant :

— Traduction : Il lit votre fichier source et génère un fichier intermédiaire `hexa.txt` contenant le code machine.

— Chargement : Il charge ce code machine dans la mémoire virtuelle de 64 Ko.

— Exécution : Il place le compteur ordinal (PC) à 0 et commence l'exécution instruction par instruction.

L'exécution s'arrête automatiquement lorsque l'instruction `hlt` est rencontrée :

```
-- FIN DU PROGRAMME --
```

4.2 Interactions (Entrées / Sorties)

Pour l'instruction `in` le programme communique avec vous l'utilisateur afin de saisir un entier dans un repertoire grace notamment à l'instruction `in rd`, tant que vous n'entrez rien, le programme est à l'arret. Si par erreur vous avez mis des caractères non numériques par exemple O au lieu de 0, le programme affichera :

```
Saisie invalide. R<x> mis a 0 par defaut
```

Avec x le numero du registre, l'exécution continue alors avec la valeur 0.

L'instruction `out rd` affiche quant à elle le contenu du registre spécifié en entier signé.

Le programme renvoie alors :

```
OUT R<x> : <valeur>
```

4.3 Sécurités et Gestion des erreurs

Contrairement à une vraie machine qui pourrait planter silencieusement, notre simulateur détecte les comportements dangereux et arrête l'exécution proprement pour vous avertir.

Liste des erreurs d'exécution (Runtime Errors)

— Boucle Infinie : Pour éviter que le simulateur ne tourne indéfiniment (par exemple sur un saut `jmp` mal configuré), une limite de sécurité est fixée à 1 000 000 de cycles. Au-delà, le programme s'arrête.

```
> Erreur : Temps d'execution depasse (> 1 000 000 cycles). Probable boucle infinie.
```

— Sortie de piste du PC : Se déclenche si le Compteur Ordinal (PC) dépasse la taille physique de la mémoire (64 Ko), souvent causé par l'oubli d'une instruction `hlt` à la fin du code.

```
> Erreur : PC hors limites (65536)
```

— Accès mémoire invalide : Lors d'une instruction de transfert (`ldb`, `stw`, etc.), l'utilisateur a tenté de lire ou d'écrire à une adresse illégale. L'adresse fautive est affichée.

```
> Erreur : Acces memoire invalide a l'adresse -4 (LDB)
```

— Erreurs Arithmétiques : Deux cas critiques sont gérés.

1. Division par zéro : L'instruction `div` avec un second opérande nul provoque un arrêt immédiat.
2. Overflow critique : Une protection spécifique empêche le crash lors de la division de l'entier minimal (`INT_MIN`) par -1.

— Instruction Inconnue : Si le processeur rencontre un code opératoire qu'il ne reconnaît pas (fichier corrompu ou saut vers une zone de données), il s'arrête.

```
> Erreur : Instruction inconnue ou invalide (Code : 42) a l'adresse 1024
```

Avertissements et Comportements Non-Bloquants

— Saisie invalide (Instruction `in`) : Si l'utilisateur entre des lettres ou des caractères spéciaux alors que le programme attend un nombre, le simulateur intercepte l'erreur, nettoie la saisie et attribue la valeur 0 au registre destination par défaut. L'exécution continue. (On a déjà vu celui là mais je le remets pour faire une liste complète)

```
> Saisie invalide. R[x] mis a 0 par defaut.
```

— Intervalle aléatoire incorrect (Instruction `rnd`) : Si vous demandez un nombre aléatoire avec une borne Min supérieure ou égale à Max, le calcul est impossible. Le simulateur attribue la valeur de la borne Min au registre pour éviter une erreur mathématique.

```
> Avertissement : Intervalle RND invalide...
```

— Dépassement de capacité (Overflow Arithmétique) : Si un calcul dépasse la valeur maximale d'un entier signé sur 32 bits ($> 2\,147\,483\,647$), le simulateur ne plante pas. La valeur "boucle" vers les négatifs selon la logique standard du Complément à 2 (ex : $MAX + 1$ devient MIN). C'est le comportement normal d'un processeur.

— Immuabilité du registre R0 : Toute tentative d'écriture dans le registre `r0` (ex : `add r0, r1, r2`) est ignorée silencieusement. L'instruction est exécutée, mais la valeur de `r0` est immédiatement forcée à 0 à la fin du cycle.

— Décalage excessif (Instruction `shr`) : Tout décalage supérieur à 32 bits vide complètement le registre (le résultat est 0), sans provoquer d'erreur système.

— Programme trop volumineux (Au chargement) : Si le fichier binaire dépasse la taille de la mémoire simulée (64 Ko), le chargement s'arrête net une fois la mémoire pleine. Le simulateur affiche une alerte puis exécute la partie du programme qui a pu être chargée.

```
> Attention : Le programme depasse la taille memoire (64ko). Troncation.
```

4.4 Fichiers génerés

L'exécution du simulateur produit un fichier `hexa.txt` dans le répertoire courant (seulement en cas de succès de la traduction evidemment).

Ce fichier contient la traduction hexadécimale de votre programme assembleur avec une instruction par ligne. Il est utile pour le débogage si vous souhaitez vérifier comment vos instructions ont été encodées par l'assembleur.