Tester par soi-même apporte plus de réponses que mille questions théoriques.

Aucune question permise

Résumé

L'innovation en programmation repose sur la capacité à transcender les limitations des langages existants. Ces derniers, bien que puissants en terme d'expressivité, peuvent restreindre la créativité des développeur.euse.s en imposant des structures rigides. Cette note propose la création d'un langage assembleur basé sur l'arithmétique de Presburger, une théorie mathématique offrant un cadre décidable pour la manipulation des entiers.

Architecture du processeur

Cycle du pipeline . Chaque instruction suit un cycle de traitement en un maximum de cinq étapes, selon le schéma suivant :

1. `IF (Instruction Fetch)` - Chargement de l'instruction depuis la mémoire.
2. `ID (Instruction Decode)` - Décodage de l'instruction pour identifier les opérandes et registres.
3. `EX (Execution)` - Exécution des calculs arithmétiques, logiques ou autres opérations.
4. `MEM (Memory Access)` - Accès à la mémoire pour lire ou écrire des données.
5. `WB (Write Back)` - Écriture des résultats dans les registres.

Registres . Le processeur dispose de 5 registres R_0, R_1, R_2, R_3, R_4 non signés de quatre (4) octets ainsi que certains registres spéciaux :

— `pc (Program Counter)` - Registre pointant vers l'instruction en cours d'exécution.
— `sp (Stack Pointer)` - Registre pointant vers le sommet de la pile.
— `lr (Link Register)` - Registre de sauvegarde de l'adresse de retour.

Pile . Le processeur dispose d'une pile P bornée à 32 éléments de quatre (4) octets non signés chacun.

Modèle de mémoire

La mémoire visible par un programme assembleur est segmentée en deux parties distinctes, soit la mémoire en lecture seule (ROM) et la mémoire en écriture seule (RAM). La mémoire est adressée par des entiers non signés de quatre (4) octets. Soit les mots-clés suivants :

— `MEM (mémoire)` - Tableau d'octets représentant la mémoire en lecture seule (ROM).
— `MAP (mappage)` - Tableau d'octets représentant la mémoire en écriture seule (RAM).

Jeu d'instructions

Instruction	*Description*	*IF*	*ID*	*EX*	*MEM*	*WB*
Pile						
`PUSH Rx`	$P[sp] \leftarrow R_x; \quad sp \leftarrow sp + 1$	x	x		x	
`POP Rx`	$R_x \leftarrow P[sp]; \quad sp \leftarrow sp - 1$	x			x	x
`TOP Rx`	$R_x \leftarrow P[sp]$	x			x	x
`SWAP`	$P[sp] \leftrightarrow P[sp-1]$	x				x
Arithmétique						
`ADD Rx, Ry, Rz`	$R_x \leftarrow R_y + R_z$	x	x	x		x
`MOV Rx, Ry`	$R_x \leftarrow R_y$	x	x			x
`INC Rx`	$R_x \leftarrow Rx + 1$	x	x	x		x
`ZER Rx`	$R_x \leftarrow 0$	x				x
Mémoire						
`LDR Rx, Ry, Rz`	$R_x \leftarrow MEM[R_y : R_y + min(R_z, 4)]$	x	x	x	x	x
`STR Rx, Ry`	$MAP[R_x : R_x + 4] \leftarrow MAP[R_x : R_x + 4] \mid R_y$	x	x	x	x	
Contrôle						
`JLE Rx, Ry, label`	$pc \leftarrow$ label si $x \leq y$	x	x	x		x
`CALL label`	$lr \leftarrow pc; \quad pc \leftarrow$ `label`	x				x
`RET`	$pc \leftarrow lr$	x	x			x
`NOP`	*Ne fait rien*	x				

Ainsi, l'objectif est de démontrer que le langage assembleur minimal présenté est suffisamment puissant pour permettre l'expression d'un calcul permettant l'assemblage de blocs deux dimensions à l'aide d'un manuel d'instructions.

Il existe deux formats de manuel d'instructions pouvant être interprété par le langage L' :

1. *premier format*
   - `1er octet` : largeur (4 bits) et hauteur (4 bits) de la pièce.
   - `2e octet` : décalage horizontal (4 bits) et vertical (4 bits) de la pièce.
   - `3e octet` : couleur de la pièce.

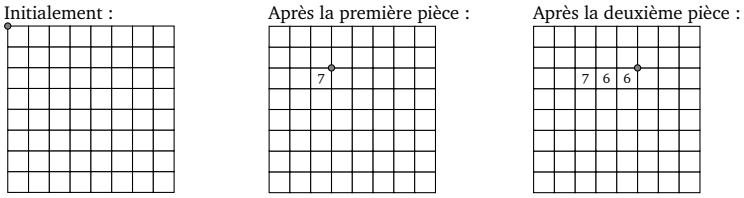
   Soit par exemple `MEM`:=$[0x11, 0x22, 0x07, 0x21, 0x10, 0x06]$ pouvant être traduit par :
   1. Placer une pièce 1×1 avec un déplacement de $(2, 2)$ et de couleur $0x07$.
   2. Placer une pièce 2×1 avec un déplacement de $(1, 0)$ et de couleur $0x06$.
2. *deuxième format*
   - `1er octet` : largeur de la pièce.
   - `2e octet` : hauteur de la pièce.
   - `3e octet` : décalage horizontal de la pièce.
   - `4e octet` : décalage vertical de la pièce.
   - `5e octet` : couleur de la pièce.

   Soit par exemple `MEM`:=$[0x01, 0x01, 0x02, 0x02, 0x07, 0x02, 0x01, 0x01, 0x00, 0x06]$ pouvant être traduit par :
   1. Placer une pièce 1×1 avec un déplacement de $(2, 2)$ et de couleur $0x07$.
   2. Placer une pièce 2×1 avec un déplacement de $(1, 0)$ et de couleur $0x06$.

Le programme assembleur doit être capable de lire un manuel d'instructions (se trouvant dans MEM) dans l'un des deux formats et d'assembler les blocs de manière à former une construction en deux dimensions.

Le positionnement suit une logique cumulative : initialement à $(0, 0)$, soit le coin supérieur gauche, chaque pièce est placée relativement à la précédente. Son ancrage est fixé à son coin supérieur gauche. Pour que cela reste décidable, la taille de la grille servant à y placer les pièces (`MAP`) est fixée à 8×8. Lorsqu'un déplacement occasionne un dépassement de la grille, le programme doit repositionner son ancrage à la position 0 de cette dimension. C'est-à-dire que si une pièce subit un déplacement de $(4, 3)$ et que l'on se trouve à la position $(5, 6)$, la pièce sera placée à la position $((5 + 4) \mod 8, (6 + 3) \mod 8) = (1, 1)$. Cette même logique est appliquée lorsqu'une partie de la pièce dépasse la grille. Le point d'ancrage finale après la mise en place d'une pièce est donnée par la formule suivante : $((p_{ix} + w + dx) \mod 8, (p_{iy} + h + dy - 1) \mod 8)$ où p_{ix} et p_{iy} sont les coordonnées du coin supérieur gauche du point d'ancrage initial, w et h sont respectivement la largeur et la hauteur de la pièce et dx et dy sont les déplacements horizontal et vertical de la pièce.

L'illustration ci-dessous montre l'évolution de la grille après le placement des pièces pour la mémoire `MEM`:=$[0x11, 0x22, 0x07, 0x21, 0x00, 0x06]$ (premier format) :



L'objectif est donc dans un premier temps de traduire un manuel d'instructions pour en faire une construction dans la grille `MAP`. Dans un second temps, nous conjecturons qu'il existe plusieurs agencements d'instructions permettant de réaliser un programme assembleur minimal L' capable de réaliser cette tâche. Il faudrait donc optimiser le programme pour que ce dernier utilise le moins de cycle possible avant de terminer la construction.

Outil de déverminage

Un outil rudimentaire a été mis sur place pour aider à la conception de programmes en langage assembleur. Cet outil permet de visualiser l'état du programme à chaque étape de son exécution. Ce programme nommé *asm* s'utilise comme suit :

```
./asm [-p <path>] [-m <path>] [-b breakpoints]
```

Où

- `-p <path>` : spécifie le chemin vers le programme assembleur à exécuter.
- `-m <path>` : spécifie le chemin vers le manuel d'instructions.
- `-b breakpoints` : spécifie les points d'arrêt séparés par des virgules. On doit écrire les numéros de lignes faisant référence au fichier assembleur.

Exemple d'utilisation : `./asm -p program.asm -m manual.txt -b 3,6,42`

Visuel du programme. Le programme de déverminage est composé de six zones distinctes ainsi que six commandes pour interagir avec le programme. Voici sous forme de tableaux les différentes zones et commandes disponibles :

Commande	*Utilité*
`q`	Quitter le programme.
`i`	Changer la zone sélectionnée.
`w`	Défiler vers le haut la zone sélectionnée.
`x`	Défiler vers le bas la zone sélectionnée.
`s`	Exécuter une instruction.
`c`	Continuer l'exécution jusqu'au prochain point d'arrêt.

Liste des commandes pour interagir avec le dévermineur.

Zone	*Description*
`Instructions`	Liste des instructions du programme.
`Processor`	Information sur la pile et les registres.
`Memory`	Contenu de la mémoire ROM (manuel d'instructions).
`Map`	Contenu de la mémoire RAM (grille de construction).
`Pipeline`	État du pipeline d'exécution.

Liste des zones affichées par le dévermineur.

Pour la zone `Map`, les couleurs seront représentées non pas par des valeurs hexadécimales, mais par leur équivalent en couleur.

$0 \Rightarrow$ Blanc, $1 \Rightarrow$ Bleu, $2 \Rightarrow$ Vert, $3 \Rightarrow$ Cyan, $4 \Rightarrow$ Rouge, $5 \Rightarrow$ Magenta, $6 \Rightarrow$ Jaune, $7 \Rightarrow$ Noir

Manuels d'instructions. Différents manuels d'instructions sont fournis pour tester vos programmes. Voici comment ils sont structurés :

- *première ligne* : numéro de format du manuel (1 ou 2).
- *lignes suivantes* : chaque ligne contient une instruction sous la forme `<largeur> <hauteur> <décalage_horizontal> <décalage_vertical> <couleur>`.

On peut prendre pour acquis que les instructions ont des valeurs valides et minimales. Par exemple, un décalage horizontal ne dépassera jamais la largeur - 1 de la grille.

Pointage

Soit l'ensemble des tests ayant été fournis de base pour le projet :

- `manuals_f1/` : dossier contenant les manuels d'instructions avec le premier format pour les instructions.
- `manuals_f2/` : dossier contenant les manuels d'instructions avec le deuxième format pour les instructions.

❗ Un (1) seul fichier par format d'instructions sera utilisé pour les tests. Puisqu'il y a deux formats, vous aurez deux fichiers à soumettre. Si plusieurs (plus de deux) fichiers sont soumis, un fichier sera choisi aléatoirement pour chaque format.

Notation . Pour chacun des manuels (des dossiers `manuals_f1` et `manuals_f2`), un test sera effectué pour vérifier que le programme donne bien le résultat attendu (dessin dans la grille `MAP`).

Attribution des points pour chaque manuel :

- Si c'est le bon résultat attendu :
  L'équipe ayant fait l'implémentation utilisant le moins de cycle d'horloge aura un (1) point pour les manuels `m1_x`, deux (2) points pour les manuels `m2_x` ainsi de suite. L'équipe ayant fait l'implémentation la plus lente aura trois quarts (0.75) de point de moins que le maximum 0.25 pour les manuels `m1_x`, 1.25pour les manuels `m2_x`, ainsi de suite. Les autres équipes ayant réussi auront un pointage distribué de manière linéaire entre ces deux bornes.
- Sinon :
  Un pointage variant entre zéro (0) et un dixième (0.1) de point sera attribué selon l'avancement de la construction.
- Si le programme est hardcodé :
  Un pointage de 0.25 sera attribué.

Le pointage final sera composé de la sommation des points de chaque manuel.