# *Manuel d'utilisation du simulateur ARM epater Révision 2*

*Marc-André Gardner, Yannick Hold-Geoffroy et Jean-François Lalonde*

*7 mars 2018*

## *Introduction*

Ce manuel présente l'utilisation du simulateur **épater**, qui émule un processeur à architecture ARMv4 (coeur ARM7), tel qu'étudié dans le cours *GIF-1001 Ordinateurs : Structure et Applications*. Il présente également succinctement le jeu d'instruction ARMv4. Des informations complémentaires concernant l'architecture ARM et la programmation en assembleur peuvent être obtenues dans les notes de cours ou dans les manuels de référence cités dans le plan de cours.

## *Prise en main*

### *Accès au simulateur*

Le simulateur est disponible à l'adresse `http://gif1001-sim.gel.ulaval.ca/`. Aucune installation n'est nécessaire. Le simulateur a été testé avec les navigateurs suivants :

— Google Chrome version 50 et plus, sur Windows, MacOS et Linux

— Mozilla Firefox version 44 et plus, sur Windows, MacOS et Linux

— Microsoft Edge, sur Windows 10

— Apple Safari, sur MacOS

D'autres navigateurs peuvent également être compatibles avec l'interface du simulateur, mais cette compatibilité n'est pas garantie. Si vous rencontrez des problèmes avec un navigateur alternatif, installez un des navigateurs énumérés plus haut.

***Note importante :*** Le simulateur opère automatiquement une sauvegarde **locale** de votre code. Aucune donnée n'est conservée sur le serveur : si les données de votre navigateur sont effacées ou perdues, vous **perdrez** votre code. Veillez à **toujours** télécharger le fichier contenant votre code, en utilisant le bouton "Sauvegarder", à chaque fois que vous quittez la session de simulation ou fermez votre navigateur. Il est de **votre** responsabilité de vous assurer de conserver ces fichiers en lieu sûr. L'équipe du simulateur ne pourra être tenue responsable de perte de données et **ne possède aucun moyen** de récupérer des données perdues.

*Table des matières*

*Introduction* 1
*Prise en main* 1
  *Accès au simulateur* 1
*Utilisation du simulateur* 4
  *Éditeur de code* 5
  *Contrôles du simulateur* 7
  *Vue des registres* 8
  *Vue des drapeaux* 9
  *Vue de la mémoire* 10
  *Description de l'instruction courante* 12
  *Sauvegarde des sessions de travail* 13
  *Contrôles d'enregistrement et de chargement* 14
  *Configuration* 15
*Instructions ARM supportées* 16
*Instructions de données* 17
  *Exemples d'utilisation* 17
*Assignations et allocations en mémoire* 22
*Instructions d'accès mémoire* 23
  *Accès mémoire de base* 23
  *Format d'une instruction LDR ou STR* 23
  *Exemples d'utilisation* 24
  *Accès mémoire multiples* 25
  *Exemples d'utilisation* 25
  *Échange mémoire/registre (swap)* 26
  *Exemples d'utilisation* 26
*Instructions de branchement* 27
  *Exemples d'utilisation* 27
*Instructions de décalage* 28
  *Exemples d'utilisation* 28
*Instructions d'accès au registre de contrôle et de statut* 29
  *Exemples d'utilisation* 29

*Instructions de déclenchement d'une interruption logicielle* 30
*Exemples d'utilisation* 30
*Assertions* 31
*Exemples d'utilisation* 31
*Note importante* 31
*Remerciements* 32

## *Utilisation du simulateur*

La page d'accueil du simulateur est présentée à la figure 1. Elle contient, sur la gauche, un menu permettant de choisir le type d'activité (démonstrations, exercices, travaux pratiques ou simulation libre) et, dans sa section principale, une liste des programmes disponibles.

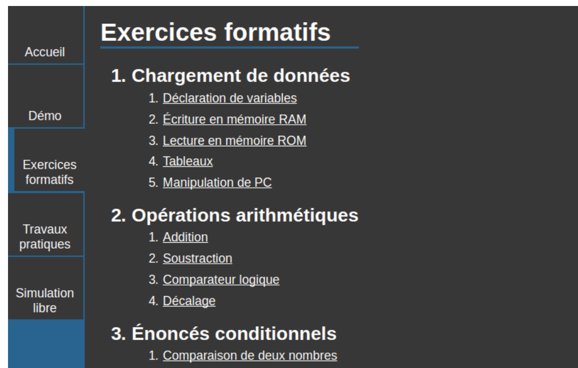


FIGURE 1: Page d'accueil du simulateur

Cliquer sur l'un de ces programmes ouvre l'interface de simulation à proprement parler. La figure 2 présente un exemple typique de cette interface, annotée. Les sous-sections suivantes présentent chaque zone de l'interface.

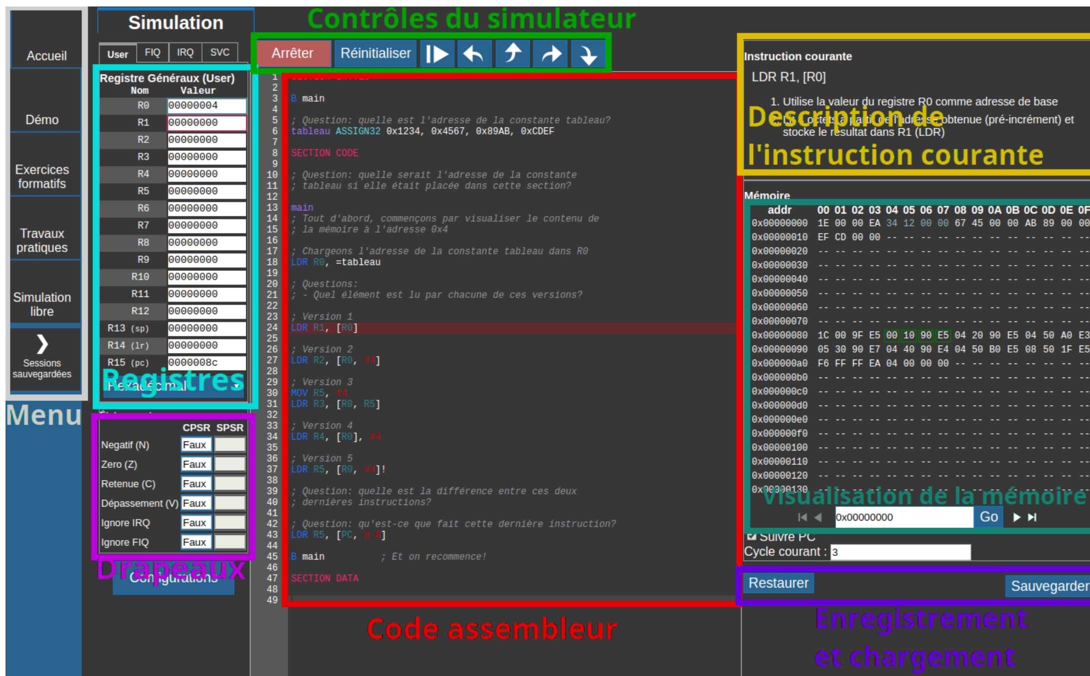


FIGURE 2: Vue d'ensemble de l'interface du simulateur.

*Éditeur de code*

L'éditeur de code constitue la zone principale du simulateur. C'est dans cet éditeur que vous pouvez créer votre programme. Lors de l'exécution, *l'éditeur passe en mode lecture seulement*, mais présente certaines informations comme la ligne en cours d'exécution.

Cet éditeur possède une *coloration syntaxique*, c'est-à-dire qu'il est capable d'analyser le code qu'il convient pour colorier différemment les diverses parties d'une instructions. Cela permet de faciliter la lecture et la détection des erreurs.

La colonne de gauche contient le numéro de ligne. Lorsqu'une instruction est erronée, un X blanc sur fond rouge y apparaît pour signaler le problème :

Figure 3: Vue typique de l'éditeur

```
22 LDR  R3, =_variable   ; Questions 11 et 12
23 STR  R0, [R1]         ; Question 13
24 MOV  R5 #4
25 LDR  R2, [R0], #4     ; Question 14
```

Figure 4: Exemple d'instruction erronée. Le X blanc sur fond rouge signale la ligne contenant l'instruction invalide.

Passer la souris sur ce X fait apparaître une infobulle contenant un texte explicatif de l'erreur :

```
23 STR  R0, [R1]         ; Question 13
24 MOV  R5 #4
Les registres et/ou constantes utilisés dans une opération doivent être séparés par une virgule
27 ADD  R0, R0, #4       ; Question 16
```

Figure 5: Le message expliquant l'erreur peut être consulté en passant la souris sur le X.

Il est possible de mettre en place des points d'arrêt (*breakpoints*) en cliquant sur le numéro d'une ligne. Un point d'arrêt force le simulateur à s'arrêter lorsqu'il l'atteint. Un point d'arrêt peut être désactivé en recliquant sur le même numéro de ligne. Par exemple, dans la figure suivante, les lignes 20 et 22 sont des points d'arrêt :

```
17 main
18 LDR  R0, =_tableau    ; Question 4
19 LDR  R1, =_pointeur   ; Question 5
20 LDR  R2, [PC, #-128]  ; Questions 6, 7 et 8
21 LDR  R4, [PC, #-128]  ; Questions 9, 10 et 11
22 LDR  R3, =_variable   ; Questions 11 et 12
23 STR  R0, [R1]         ; Question 13
24 MOV  R5, #4
```

Figure 6: Un point d'arrêt (*breakpoint*) peut être mis en place en cliquant sur le numéro de ligne.

***Note importante :*** un point d'arrêt peut être placé sur n'importe quelle ligne si le programme n'est *pas assemblé*. Une fois le programme *assemblé* (le bouton "Démarrer" pressé et le code assemblé sans erreur), un point d'arrêt peut seulement être placé sur une ligne *contenant une instruction* et un point d'arrêt invalide est désactivé ou

déplacé sur la prochaine ligne. Si l'utilisateur clique sur une ligne ne contenant pas d'instruction, le point d'arrêt sera placé sur la prochaine ligne contenant une instruction. Par exemple, à la figure 6, cliquer sur la ligne 17 placera un point d'arrêt à la ligne *18*.

Lors de l'exécution, l'éditeur passe en mode lecture seule. Il affiche cependant certaines informations. La ligne en cours d'exécution est surlignée en rouge bourgogne :

```
16
17  main
18  LDR    R0, =_tableau   ; Question 4
19  LDR    R1, =_pointeur  ; Question 5
20  LDR    R2, [PC, #-128] ; Questions 6, 7 et 8
21  LDR    R4, [PC, #-128] ; Questions 9, 10 et 11
```

Figure 7: L'instruction qui va être exécutée au prochain pas de temps est surlignée en rouge dans l'éditeur. Si aucune instruction n'est surlignée, c'est que le *Program Counter* (PC) ne se situe pas dans la mémoire d'instructions.

De même, lorsque l'instruction courante est un branchement ou un appel de fonction, l'éditeur affiche la destination du branchement en surlignant en noir la prochaine instruction. Par exemple, dans ce cas-ci, l'instruction en cours d'exécution (surlignée en rouge) est le *B main*, et la prochaine instruction sera celle de la ligne 18 (surlignée en noir) :

```
5   B        main    ; Question 1
6
7   ; Definitions de constantes en memoire ROM
8   _tableau                DC32    0x002938AE, 0x003EF391
9   _AdresseDePointeur      DC32    0x00100000
10  _AdresseDeTableau       DC32    0x00000004
11
12  ; La directive SECTION place le texte qui suit en ROM
13  ; La mémoire ROM commence à l'adresse 0x00000000, mais
14  ; sont réservés. Le code de la section .text commence
15  SECTION CODE
16
17  main
18  LDR    R0, =_tableau   ; Question 4
19  LDR    R1, =_pointeur  ; Question 5
20  LDR    R2, [PC, #-128] ; Questions 6, 7 et 8
```

Figure 8: La prochaine instruction à être exécutée après un branchement est surlignée en noir profond. Si le branchement est conditionnel, cet affichage en tient compte et affiche la prochaine instruction correspondant à la branche prise.

### *Contrôles du simulateur*

Les boutons de contrôle du simulateur permettent d'agir sur la simulation. Cette interface diffère selon le mode actuel. Dans le mode *édition* (figure 9), seul le bouton *Démarrer* est actif. Dans le mode *exécution*, que l'on peut accéder en pressant sur *Démarrer*, tous les boutons sont actifs, et le bouton *Démarrer* devient *Arrêter*.

Figure 9: Barre de contrôle en mode *édition*

Figure 10: Barre de contrôle en mode *exécution*

Lorsque le bouton *Démarrer* est pressé, le simulateur tente d'abord d'assembler le code. Si aucune erreur n'est détectée, il passe alors en mode exécution. Dans ce mode, tous les boutons sont actifs et sont, respectivement, de gauche à droite :

1. **Arrêt** : interrompt la simulation et revient en mode édition.
2. **Réinitialisation** : effectue l'équivalent d'une interruption *reset* sur le microprocesseur. La valeur de PC est mise à 0. Notez que cela n'affecte *pas* la valeur des autres registres et des drapeaux.
3. **Exécution en continu** : le simulateur exécute les instructions suivantes sans arrêt, jusqu'à ce qu'il rencontre une erreur ou un point d'arrêt. Notez que le simulateur est volontairement limité quant au nombre d'instructions qu'il peut exécuter d'un seul coup (10 000). Lorsque ce nombre est atteint, le simulateur arrête comme s'il avait rencontré un point d'arrêt. Il suffit d'appuyer à nouveau sur le bouton d'exécution en continu pour poursuivre.
4. **Retour en arrière** : retourne en arrière d'une instruction. L'état du simulateur (les registres, la mémoire et les drapeaux) retourne comme il était à l'instruction précédente puis arrête.
5. **Exécuter jusqu'à la sortie** : dans le cas où la ligne courante est dans une fonction, cette action exécute sans arrêt les instructions jusqu'à sortir de la fonction (appel de BX). Si la ligne courante n'est pas dans une fonction, cette action a le même effet qu'une exécution en continu.
6. **Exécuter ligne courante** : dans le cas où la ligne courante est une instruction autre que BL, cette action a le même effet que la précédente. Toutefois, dans le cas d'un appel de fonction (BL), cette action exécute l'entièreté de la fonction jusqu'à son retour.
7. **Exécuter instruction courante** : exécute l'instruction courante (celle qui est surlignée dans l'éditeur) et passe à la suivante puis arrête.

*Vue des registres*

La section de gauche de l'interface présente les valeurs contenues dans les registres du processeur. Les 16 registres que contient un processeur ARM (R0 à R15) sont affichés.

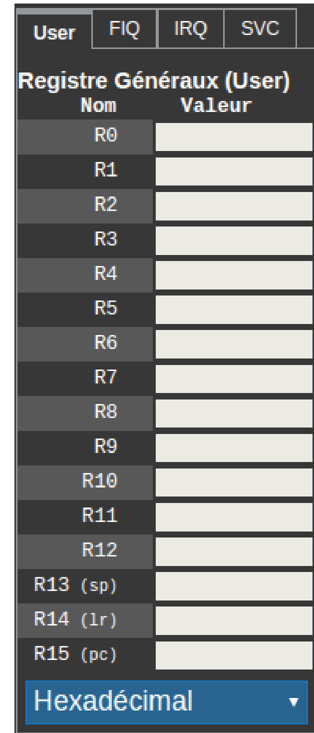


FIGURE 11: Vue des registres généraux

Par défaut, leur valeur est présentée en hexadécimal, mais il est possible de choisir différents modes d'affichage en cliquant sur le mode actuel pour faire apparaître le menu de sélection. Le mode décimal signé correspond à une interprétation complément-2 de la valeur du registre, alors que le mode non-signé correspond à une simple conversion vers le système décimal.

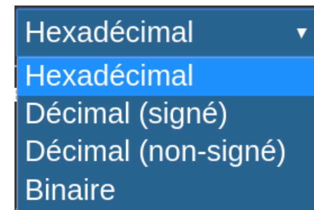


FIGURE 12: Menu de sélection du mode d'affichage

Les onglets au-dessus des registres permettent de choisir la *banque* de registre à visualiser. La plupart des banques de registre d'un processeur ARM sont présentes, incluant la banque IRQ (interruption), FIQ (interruption rapide) et SVC (interruption logicielle).

FIGURE 13: Onglets de sélection de la banque de registres à visualiser

La valeur d'un registre peut être modifiée en changeant sa valeur dans l'interface. Attention toutefois à respecter le type d'affichage sélectionné (par exemple, il est illégal d'écrire autre chose que 0 ou 1 en mode binaire).

Il est possible d'affecter un point d'arrêt à un registre, en mode lecture ou écriture. Un point d'arrêt lié à un registre met la simulation en pause lorsque le registre est écrit ou lu. Par exemple, dans la figure suivante, les registres R3 et R5 ont un point d'arrêt en écriture et le registre R2 un point d'arrêt en lecture :

| Nom | Valeur |
|---|---|
| R0 | 00000000 |
| R1 | 00000000 |
| R2 | 00000000 |
| R3 | 00000000 |
| R4 | 00000000 |
| R5 | 00000000 |
| R6 | 00000000 |

FIGURE 14: Points d'arrêt sur les registres R2, R3 et R5.

Un point d'arrêt en écriture peut être ajouté en cliquant sur le nom du registre tout en tenant la touche Ctrl ou Cmd enfoncée, respectivement sur Windows et MacOS. De même, un point d'arrêt en lecture peut être ajouté en cliquant tout en tenant la touche Maj (Shift) enfoncée. Un point d'arrêt précédemment créé peut être retiré en répétant la même opération. Il est possible de mettre à la fois un point d'arrêt en lecture et en écriture sur le même registre.

Lorsque l'instruction courante lit ou écrit un registre, sa valeur est entourée d'un rectangle turquoise (lecture) ou rouge (écriture). Par exemple, dans l'image suivante, l'instruction courante lit les valeurs de R0 et R5 et écrit dans R2 :

| Nom | Valeur |
|---|---|
| R0 | 00000008 |
| R1 | 00000100 |
| R2 | 002938ae |
| R3 | 00010004 |
| R4 | 00000004 |
| R5 | 00000006 |
| R6 | 00000000 |

### *Vue des drapeaux*

Les drapeaux de l'ALU sont présentés juste en dessous de la vue des registres.

Un drapeau ne peut prendre que deux valeurs : vrai ou faux (0 ou 1 en binaire). Il est possible de changer la valeur d'un drapeau en cliquant sur sa valeur. L'interface présente à la fois la vue pour le registre de statut courant (CPSR) et le registre de statut sauvegardé (SPSR), si applicable.

Le registre de statut sauvegardé (SPSR) est propre à chaque banque de registres, hormis la banque *User* qui n'en possède pas. Il permet de conserver le registre de statut du programme principal pendant une interruption. Dans la majorité des cas (exécution en mode *User*), les valeurs du SPSR ne sont pas définies ou modifiables.

Tout comme pour les registres, les drapeaux lus ou écrits par une instructions voient leur contour coloré de manière différente.

État courant

| | CPSR | SPSR |
|---|---|---|
| Negatif (N) | Vrai | |
| Zero (Z) | Faux | |
| Emprunt (C) | Faux | |
| Dépassement (V) | Faux | |
| Ignore IRQ | Faux | |
| Ignore FIQ | Faux | |

Figure 15: Vue des drapeaux de l'ALU

### *Vue de la mémoire*

La vue de la mémoire présente le contenu de la mémoire d'instructions et de données. Cette vue est arrangée en tableau, où chaque ligne fait 16 octets de largeur. Par exemple, pour retrouver la valeur à l'adresse 0x9A, il suffit d'aller au croisement de la ligne 0x90 et de la colonne 0xA. Les espaces mémoire non déclarés (qui ne se rapportent ni à une instruction, ni à une variable) sont indiqués par des tirets. L'affichage du contenu de la mémoire se fait en hexadécimal.

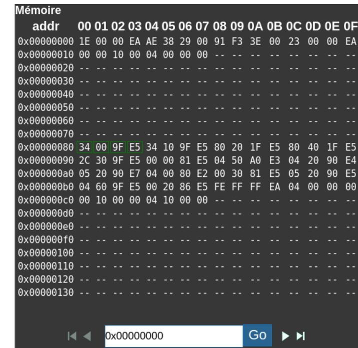


FIGURE 16: Vue de la mémoire

Tout comme pour les registres et les drapeaux, il est possible de modifier une valeur initialisée de la mémoire en cliquant. Les zones non déclarées (dont la valeur est identifiée par des tirets) ne peuvent être modifiées. La valeur doit être écrite en hexadécimal et ne peut excéder la capacité d'un octet (255, soit 0xFF).

Lors de l'exécution, les octets composant l'instruction courante (celle surlignée en rouge dans l'éditeur) sont entourés de vert, comme dans l'exemple suivant :

```
0x00000080 34 00 9F E5 34 10 9F E5 80 20 1F E5 80 40 1F E5
0x00000090 2C 30 9F E5 00 00 81 E5 04 50 A0 E3 04 20 90 E4
0x000000a0 05 20 90 E7 04 00 80 E2 00 30 81 E5 05 20 90 E5
0x000000b0 04 60 9F E5 00 20 86 E5 FE FF FF EA 04 00 00 00
0x000000c0 00 10 00 00 04 10 00 00 -- -- -- -- -- -- -- --
```

FIGURE 17: Affichage des octets composant l'instruction courante

Lorsque l'instruction est une opération agissant en mémoire, les cases mémoires lues ou écrites voient leurs valeurs colorées de manière différente (vert dans le cas d'une lecture, rouge dans le cas d'une écriture). Par exemple, dans l'image suivante, l'instruction courante lit les valeurs 0x10 à 0x13 inclusivement :

```
Mémoire
   addr    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x00000000 1E 00 00 EA AE 38 29 00 91 F3 3E 00 23 00 00 EA
0x00000010 00 00 10 00 04 00 00 00 -- -- -- -- -- -- -- --
```

FIGURE 18: Exemple du changement de couleur des valeurs d'une case mémoire lors d'un accès

Il est possible de placer des points d'arrêt en mémoire. Ceux-ci peuvent être de trois types :

— **Lecture** : le simulateur s'arrête lorsqu'un accès en lecture (par exemple via l'instruction LDR) est effectué sur cette case mémoire

— **Écriture** : le simulateur s'arrête lorsqu'un accès en écriture (par exemple via l'instruction STR) est effectué sur cette case mémoire

— **Exécution** : le simulateur s'arrête lorsque le *Program Counter* (PC) atteint cette valeur

Notons que ces trois types de points d'arrêt peuvent être combinés. Par exemple, dans l'image suivante, un point d'arrêt en lecture est présent pour les adresses 0x04 à 0x07, un point d'arrêt en écriture est

actif aux adresses 0x0C à 0x0F et un point d'arrêt en exécution est présent à l'adresse 0x8C :

**Mémoire**

| addr | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x00000000 | 1E | 00 | 00 | EA | AE | 38 | 29 | 00 | 91 | F3 | 3E | 00 | 23 | 00 | 00 | EA |
| 0x00000010 | 00 | 00 | 10 | 00 | 04 | 00 | 00 | 00 | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000020 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000030 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000040 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000050 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000060 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000070 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000080 | 34 | 00 | 9F | E5 | 34 | 10 | 9F | E5 | 80 | 20 | 1F | E5 | 80 | 40 | 1F | E5 |
| 0x00000090 | 2C | 30 | 9F | E5 | 00 | 00 | 81 | E5 | 04 | 50 | A0 | E3 | 04 | 20 | 90 | E4 |

FIGURE 19: Exemple de points d'arrêt en mémoire. Un point d'arrêt en lecture peut être ajouté en cliquant sur une case mémoire tout en pressant la touche Control (Ctrl). Un point d'arrêt en écriture peut être ajouté en cliquant et pressant la touche Maj (Shift). Un point d'arrêt en écriture peut être ajouté en cliquant et pressant la touche Alt.

Afin de faciliter l'utilisation du simulateur, un aide-mémoire succinct reprenant les informations présentées ici est disponible en bas de la vue mémoire :

**Instructions pour l'activation des breakpoints**
Écriture (W) : Ctrl/Cmd + Clic
Lecture (R) : Shift + Clic
Lecture et Écriture (RW) : Ctrl/Cmd + Shift + Clic
Exécution (E) : Alt + Clic
Instruction courante

FIGURE 20: Description des différents types de points d'arrêt en mémoire dans l'interface du simulateur

Finalement, en bas du visualisateur, on retrouve une interface permettant de naviguer dans la mémoire :

FIGURE 21: Contrôle de la position dans la mémoire

Les flèches permettent d'aller vers des adresses mémoire plus hautes ou plus basses. Le champ de texte central permet de spécifier directement une adresse, à laquelle on peut par la suite se rendre en pressant le bouton *Go*.

Finalement, notons que cette vue indique, par un soulignement blanc, la position en mémoire de la ligne couramment *sélectionnée* dans l'éditeur (celle sur laquelle se trouve le curseur). Si cette ligne est une instruction, elle couvrira 4 cases (soit 4 octets). S'il s'agit plutôt d'une constante ou d'une variable, elle indiquera toutes les cases mémoire appartenant à de celle-ci.

```
15  ; Comparons R0 et R1
16  CMP R0, R1
17
18  ; Cette instruction est équivalente à CMP R0, R1
19  SUBS R0, R0, R1
20
21  ; Lorsque la comparaison est effectuée, nous pouvons exécuter des instructions
22  ; conditionnelles. Ces instructions utilisent les drapeaux de l'ALU pour
23  ; déterminer si elles doivent être exécutées ou non.
24
25  ; R3 <- R1 seulement si les opérandes sont égales
26  ; (EQ = equal, ou si le drapeau Z est égal à 1)
27  MOVEQ R3, R1
28
29  ; R2 <- R2 + R1 seulement si les opérandes ne sont pas égales
30  ; (NE = not equal, ou si le drapeau Z est égal à 0)
31  ADDNE R2, R2, R1
32
33
34  B main
35
36  SECTION DATA
37
38
39
```

| addr | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x00000000 | 1E | 00 | 00 | EA | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000010 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000020 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000030 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000040 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000050 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000060 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000070 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x00000080 | 05 | 00 | A0 | E3 | 0A | 10 | A0 | E3 | 01 | 00 | 50 | E1 | 01 | 00 | 50 | E0 |
| 0x00000090 | 01 | 30 | A0 | 01 | 01 | 20 | 82 | 10 | F8 | FF | FF | EA | -- | -- | -- | -- |
| 0x000000a0 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x000000b0 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x000000c0 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x000000d0 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x000000e0 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| 0x000000f0 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |

FIGURE 22: Vue de la position de la ligne couramment sélectionnée. Dans ce cas-ci, la ligne 27 est sélectionnée ; la vue mémoire souligne en blanc les cases mémoires qui y correspondent, soit les adresses 0x90 à 0x93 inclusivement. La ligne en cours d'exécution est quant à elle surlignée en rouge et sa position en mémoire indiquée par des carrés verts autour des cases concernées.

### *Description de l'instruction courante*

Cette section présente des informations textuelles sur l'instruction courante. La première ligne présente l'instruction *désassemblée* : elle provient non pas de l'éditeur, mais du *bytecode* lui-même. Cela lui permet d'apporter de l'information supplémentaire dans deux circonstances :

1. Les étiquettes y sont remplacées par les décalages effectifs requis pour atteindre la case mémoire demandée
2. Dans le cas où l'on exécute des données présentes en mémoire, auxquelles aucun code assembleur n'est lié, cet affichage reste fonctionnel puisqu'il ne se base que sur la représentation binaire des instructions.

**Instruction courante**
LDR R2, [R0, R5]

1. Utilise la valeur du registre R0 comme adresse de base
2. Additionne le registre R5 à l'adresse de base
3. Lit 4 octets à partir de l'adresse obtenue (pré-incrément) et stocke le résultat dans R2 (LDR)

Figure 23: Affichage de l'instruction désassemblée et de sa description. En fonction de l'instruction, la description peut être plus ou moins longue.

Les lignes suivantes sont une description textuelle de l'instruction, générée automatiquement. Elles indiquent, dans l'ordre, les opérations effectuées pour obtenir le résultat final et leurs effets de bord s'il y a lieu. Dans le cas d'une instruction indéfinie, cette zone reste vide.

### *Sauvegarde des sessions de travail*

Les sessions sauvegardées permettent d'alterner entre plusieurs versions de code d'une même simulation. La session courante est sauvegardée automatiquement et restaurée au retour à la même simulation.

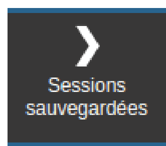


Figure 24: Boutton du menu latéral

Pour afficher le panneau des sessions sauvegardées, il faut survoler le bouton "Sessions sauvegardées" du menu latéral.

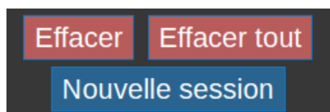


Figure 25: Contrôle des sessions sauvegardées

Trois bouton sont accessible pour modifier les sessions :

1. **Effacer** : la session courante est supprimée et la suivante est sélectionnée (son code est restauré).
2. **Effacer tout** : toutes les sessions sont supprimées puis le code par défaut est restauré.
3. **Nouvelle session** : la session courante est sauvegardée et une nouvelle session est ajoutée en tête de la liste. Le code par défaut est restauré.

De plus, cliquer sur une session sauvegarde la session courante et restaure la session sélectionnée. La session en cours est affichée en bleu pour la différencier.

Certaines informations sont affichées pour distinguer les différentes sessions. On y retrouve le numéro de la session, l'heure de la dernière sauvegarde, la taille du code (en nombre de caractères) et le nom personnalisé de la session. Un double-clic sur le nom permet de le modifier.

Figure 26: Vue des sessions sauvegardées avec la session numéro 2 de sélectionnée

***Note importante :*** bien que ce procédé permette de sauvegarder son code même après la fermeture de votre navigateur, **il est fortement recommandé d'enregistrer (télécharger) votre code**. Les sauvegardes de sessions sont enregistrées dans la mémoire de votre navigateur et il n'y a aucune garantie que celle-ci ne sera pas effacée. Si c'est le cas, l'équipe du simulateur ne possède aucun moyen de récupérer votre code, puisque aucune donnée n'est enregistrée sur le serveur.

### *Contrôles d'enregistrement et de chargement*

Ces contrôles, situés au bas de l'écran, vous permettent de télécharger le code présent dans l'éditeur et de charger un code présent sur votre ordinateur. Ces deux actions sont analogues à l'ouverture et l'enregistrement dans une application traditionnelle.

Figure 27: Interface d'ouverture et d'enregistrement

N'importe quel type de fichier texte brut peut être lu par le simulateur. Nous vous recommandons toutefois d'utiliser l'extension "txt" afin d'éviter la confusion avec d'autres formats de fichier. C'est d'ailleurs cette extension que comportent les fichiers téléchargés depuis le simulateur.

***Note importante :*** malgré que votre code est sauvegarder automatiquement dans votre navigateur, **il est fortement recommandé de sauvegarder votre code** avec ces contrôles.

## *Configuration*

La fenêtre de configuration peut être ouverte en pressant le bouton *Configurations* situé sous la vue des drapeaux. Une fois ouverte, la fenêtre ressemble à celle-ci :

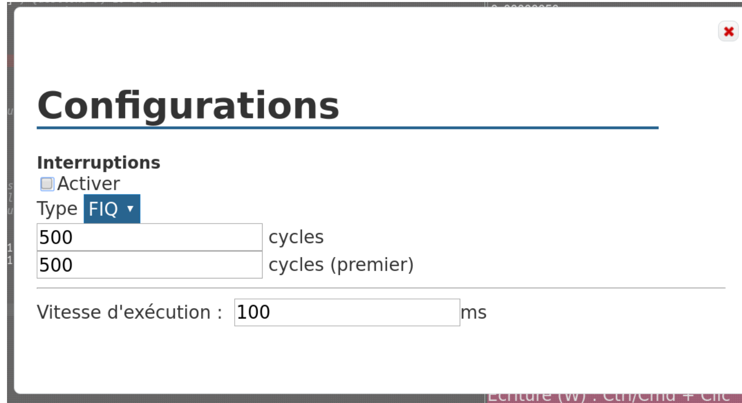


Figure 28: Interface de configuration des interruptions et du simulateur

Deux éléments principaux peuvent être configurés dans cette interface :

1. Une interruption temporelle peut être activée, en mode FIQ ou IRQ. Le nombre de cycles avant la première interruption, de même que le délai de répétition (encore là en nombre de cycles processeur) peuvent être configurés.
2. La vitesse d'exécution, qui correspond au délai minimum entre l'exécution de deux instructions consécutives. Par défaut, cette vitesse est de 0, c'est-à-dire que le simulateur exécute les instructions aussi vite que possible. Toutefois, il est possible d'augmenter ce délai afin de pouvoir visualiser l'exécution du programme. Dans tous les cas, le simulateur se met automatiquement en pause après avoir exécuté 10 000 instructions de suite.

## *Instructions ARM supportées*

Le simulateur supporte toutes les instructions ARMv4 à l'exception des instructions du co-processeur. Elles sont présentées dans les sections suivantes. Référez-vous aux notes de cours ou à la documentation d'ARM pour plus de détails sur chacune de celles-ci. La syntaxe à utiliser est celle présentée dans les spécifications et documentations techniques d'ARM, sauf pour les spécificités suivantes :

1. Toutes les mnémoniques (MOV, LDR, BL, etc.) doivent être écrites en **majuscules**
2. Tous les noms de registres doivent être écrits en **majuscules**
3. Les étiquettes peuvent contenir, majuscules, minuscules et chiffres, mais pas *commencer* par un chiffre.
4. L'indentation n'a pas d'importance. L'espace ou la tabulation après la mnémonique est obligatoire.

Toutes les instructions peuvent être exécutées *conditionnellement*, en ajoutant un des codes à 2 lettres suivants à la fin de la mnémonique :

| Code | Équation | Explication | Signé ? |
|---|---|---|---|
| **EQ** | $A == B$ | Teste l'égalité | N/A |
| **NE** | $A \neq B$ | Teste l'inégalité | N/A |
| **CS** | $A \geq B$ | Teste si un premier nombre est plus grand ou égal à un second | NON |
| **CC** | $A < B$ | Teste si un premier nombre est strictement plus petit qu'un second | NON |
| **HI** | $A > B$ | Teste si un premier nombre est strictement plus grand qu'un second | NON |
| **LS** | $A \leq B$ | Teste si un premier nombre est plus petit ou égal à un second | NON |
| **GE** | $A \geq B$ | Teste si un premier nombre est plus grand ou égal à un second | OUI |
| **LT** | $A < B$ | Teste si un premier nombre est strictement plus petit qu'un second | OUI |
| **GT** | $A > B$ | Teste si un premier nombre est strictement plus grand qu'un second | OUI |
| **LE** | $A \leq B$ | Teste si un premier nombre est plus petit ou égal à un second | OUI |
| **MI** | $A < 0$ | Teste si le résultat est négatif | OUI |
| **PL** | $A \geq 0$ | Teste si le résultat est positif | OUI |
| **VS** | N/A | Teste si la précédente opération a généré un *débordement* (overflow) | OUI |
| **VC** | N/A | Teste si la précédente opération **n'a pas** généré un *débordement* (overflow) | OUI |

TABLE 1: Codes de condition en ARM. Notons qu'une condition spéciale (AL) est implicitement ajoutée à n'importe quelle instruction ne comportant pas de condition. Cette condition signifie que l'instruction doit être inconditionnellement exécutée. La colonne *Signé ?* indique si la comparaison présume un nombre signé ou non. Par exemple, la condition CS considère que les nombres ne sont pas signés, si bien que comparer -1 (0xFFFFFFFF) et 0 (0x00000000) sera vrai, comme si $-1 \geq 0$. À l'opposé, la condition GE tient compte du bit de signe et sera fausse pour cette même comparaison.

## *Instructions de données*

Les instructions manipulant les données peuvent accepter deux ou trois arguments. Lorsque le dernier argument est un registre, il peut optionnellement être décalé par une constante ou un autre registre avant d'exécuter le reste de l'instruction (voir le tableau 2). Le résultat de l'exécution de l'instruction peut également affecter les drapeaux, en ajoutant le suffixe *S* à la mnémonique.

Les instructions de manipulation de données peuvent être divisées en quatre catégories :

— Les instructions arithmétiques : voir tableau 3
— Les instructions booléennes et logiques : voir tableau 5
— Les instructions arithmétiques utilisant la valeur du drapeau retenue C (*Carry*) : voir tableau 4
— Les instructions de comparaison (arithmétiques et booléennes) : voir tableau 6

### *Exemples d'utilisation*

`MOV R1, R2` Copie la valeur du registre R2 dans le registre R1.

`MOV R0, #8` Écrit la valeur 8 dans le registre R0.

`MOVEQ R3, R4` Copie la valeur du registre R4 dans le registre R3 seulement si la condition EQ est vraie.

`MOV R3, R4, LSL #4` Copie la valeur du registre R4 décalée de 4 bits vers la gauche dans le registre R3.

`MOV R3, R4, LSR R5` Copie la valeur du registre R4 décalée de *N* bits vers la droite, où *N* est la valeur de R5, dans le registre R3.

`ADD R8, PC, #0x10` Ajoute 16 (0x10) à PC, met le résultat dans R8.

`SUBS R0, R4, R5` Soustrait R4 et R5, stocke le résultat dans R0 et met à jour les drapeaux en fonction de ce résultat.

`ADD R0, R1, R2, LSL #1` Additionne R1 et la valeur de R2 décalée de une position vers la gauche, stocke le résultat dans R0.

`RSBNES SP, R8, R9, ASR R7` Soustrait la valeur de R9, décalée arithmétiquement vers la droite R7 fois, et R8, stocke le résultat dans SP et met à jour les drapeaux en fonction de ce résultat, mais seulement si la condition NE est rencontrée.

| Type | Format | Effet |
|---|---|---|
| **LSL** | `LSL N` | Décale les bits vers la **gauche** N fois. Des zéros sont insérés à droite pour remplacer les bits décalés. Le bit qui se retrouverait immédiatement à gauche du MSB devient la valeur du drapeau de retenue C (*Carry*) si le suffixe *S* est utilisé. |
| **LSR** | `LSR N` | Décale les bits vers la **droite** N fois. Des zéros sont insérés à gauche pour remplacer les bits décalés. Le bit qui se retrouverait immédiatement à droite du LSB devient la valeur du drapeau de retenue C (*Carry*) si le suffixe *S* est utilisé. |
| **ASR** | `ASR N` | Décale les bits vers la **droite** N fois. Les bits insérés à gauche ont la même valeur que le MSB actuel (bit 31). Le bit qui se retrouverait immédiatement à droite du LSB devient la valeur du drapeau de retenue C (*Carry*) si le suffixe *S* est utilisé. |
| **ROR** | `ROR N` | Effectue une rotation des bits vers la droite autant de fois que la valeur contenue dans REG. Les bits qui "débordent" à droite sont réinsérés à gauche. Le bit qui se retrouverait immédiatement à droite du LSB devient la valeur du drapeau C (*carry*) si le suffixe *S* est utilisé. |
| **RRX** | `RRX` | Opération spéciale qui décale tous les bits du registre vers la droite d'une position. Le drapeau C (*carry*) est inséré à la position du bit le plus significatif (MSB). Le bit éjecté devient la valeur du drapeau C (*carry*) si le suffixe *S* est utilisé. Ce décalage n'accepte pas de paramètre. |

Table 2: Format des différents modes de décalage acceptés pour une instruction de manipulation de données en ARMv4. *N* peut être soit un registre, soit une constante positive $\leq 32$. Toutes les figures (sauf RRX) montrent l'effet d'un décalage de 5.

| Code | Format | Description | Exemple |
|---|---|---|---|
| **MOV** | `MOV REG1, REG2[, DEC]` | Copie la valeur de REG2 dans REG1. Un décalage optionnel peut être appliqué sur REG2 (voir tableau 2). | `MOV R1, R2` |
| | `MOV REG1, #CONST` | Écrit la valeur CONST (en décimal ou hexadécimal précédé de `0x`) dans REG1 | `MOV R1, #0x42` |
| **ADD** | `ADD REG1, REG2, REG3[, DEC]` | Additionne REG2 et REG3, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `ADD R1, R2, R3` |
| | `ADD REG1, REG2, #CONST` | Additionne REG2 et la valeur CONST, stocke le résultat dans REG1 | `ADD R1, R2, #42` |
| **SUB** | `SUB REG1, REG2, REG3[, DEC]` | Effectue la soustraction REG2 - REG3, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `SUB R1, R2, R3` |
| | `SUB REG1, REG2, #CONST` | Effectue la soustraction REG2 - CONST, stocke le résultat dans REG1 | `SUB R1, R2, #42` |
| **RSB** | `RSB REG1, REG2, REG3[, DEC]` | Effectue la soustraction REG3 - REG2, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `RSB R1, R2, R3` |
| | `RSB REG1, REG2, #CONST` | Effectue la soustraction CONST - REG2, stocke le résultat dans REG1 | `RSB R1, R2, #42` |

TABLE 3: Instructions arithmétiques de manipulation de données

| Code | Format | Description | Exemple |
|---|---|---|---|
| **ADC** | `ADC REG1, REG2, REG3[, DEC]` | Additionne REG2 + REG3 + C, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `ADC R1, R2, R3` |
| | `ADC REG1, REG2, #CONST` | Additionne REG2 + CONST + C, stocke le résultat dans REG1 | `ADC R1, R2, #42` |
| **SBC** | `SBC REG1, REG2, REG3[, DEC]` | Effectue la soustraction REG2 - REG3 - 1 + C, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `SBC R1, R2, R3` |
| | `SBC REG1, REG2, #CONST` | Effectue la soustraction REG2 - CONST - 1 + C, stocke le résultat dans REG1 | `SBC R1, R2, #42` |
| **RSC** | `RSC REG1, REG2, REG3[, DEC]` | Effectue la soustraction REG3 - REG2 - 1 + C, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `RSC R1, R2, R3` |
| | `RSC REG1, REG2, #CONST` | Effectue la soustraction CONST - REG2 - 1 + C, stocke le résultat dans REG1 | `RSC R1, R2, #42` |

TABLE 4: Instructions arithmétiques de manipulation de données utilisant le drapeau Retenue (*Carry*). Dans les explications, *C* correspond à la valeur de ce drapeau (0 ou 1).

| Code | Format | Description | Exemple |
|---|---|---|---|
| **MVN** | `MVN REG1, REG2[, DEC]` | Effectue un NON logique sur REG2 et écrit le résultat dans R1. Un décalage optionnel peut être appliqué sur REG2 (voir tableau 2). | `MVN R1, R2` |
| | `MVN REG1, #CONST` | Effectue un NON logique sur CONST (inverse ses bits) et écrit le résultat dans REG1 | `MVN R1, #42` |
| **AND** | `AND REG1, REG2, REG3[, DEC]` | Effectue l'opération ET logique entre REG2 et REG3, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `AND R1, R2, R3` |
| | `AND REG1, REG2, #CONST` | Effectue l'opération ET logique entre REG2 et CONST, stocke le résultat dans REG1 | `AND R1, R2, #0xFF` |
| **ORR** | `ORR REG1, REG2, REG3[, DEC]` | Effectue l'opération OU logique entre REG2 et REG3, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `ORR R1, R2, R3` |
| | `ORR REG1, REG2, #CONST` | Effectue l'opération OU logique entre REG2 et CONST, stocke le résultat dans REG1 | `ORR R1, R2, #0xFF` |
| **EOR** | `EOR REG1, REG2, REG3[, DEC]` | Effectue l'opération logique REG2 XOR REG3, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `EOR R1, R2, R3` |
| | `EOR REG1, REG2, #CONST` | Effectue l'opération logique REG2 XOR CONST, stocke le résultat dans REG1 | `EOR R1, R2, #0xFF` |
| **BIC** | `BIC REG1, REG2, REG3[, DEC]` | Effectue l'opération logique REG2 ET NON REG3, stocke le résultat dans REG1. Un décalage optionnel peut être appliqué sur REG3 (voir tableau 2). | `BIC R1, R2, R3` |
| | `BIC REG1, REG2, #CONST` | Effectue l'opération logique REG2 et NON CONST, stocke le résultat dans REG1 | `BIC R1, R2, #0xFF` |

Table 5: Instructions booléennes de manipulation de données

| Code | Format | Description | Exemple |
|---|---|---|---|
| **CMP** | `CMP REG1, REG2[, DEC]` | Effectue la soustraction REG1 - REG2 et met à jour les drapeaux en fonction du résultat. Un décalage optionnel peut être appliqué sur REG2 (voir tableau 2). | `CMP R1, R2` |
| | `CMP REG1, #CONST` | Effectue la soustraction REG1 - CONST et met à jour les drapeaux en fonction du résultat. | `CMP R1, #18` |
| **CMN** | `CMN REG1, REG2[, DEC]` | Effectue l'addition REG1 + REG2 et met à jour les drapeaux. Un décalage optionnel peut être appliqué sur REG2 (voir tableau 2). | `CMN R1, R2` |
| | `CMN REG1, #CONST` | Effectue l'addition REG1 + CONST et met à jour les drapeaux en fonction du résultat. | `CMN R1, #18` |
| **TEQ** | `TEQ REG1, REG2[, DEC]` | Effectue l'opération logique REG1 XOR REG2 et met à jour les drapeaux en fonction du résultat. Un décalage optionnel peut être appliqué sur REG2 (voir tableau 2). | `TEQ R1, R2` |
| | `TEQ REG1, #CONST` | Effectue l'opération logique REG1 XOR CONST et met à jour les drapeaux en fonction du résultat. | `TEQ R1, #0x5F` |
| **TST** | `TST REG1, REG2[, DEC]` | Effectue l'opération logique REG1 AND REG2 et met à jour les drapeaux en fonction du résultat. Un décalage optionnel peut être appliqué sur REG2 (voir tableau 2). | `TST R1, R2` |
| | `TST REG1, #CONST` | Effectue l'opération logique REG1 AND CONST et met à jour les drapeaux en fonction du résultat. | `TST R1, #0x5F` |

Table 6: Instructions de comparaisons de données (arithmétique et logique). Ces instructions ont le même effet que d'autres, mais n'écrivent leur résultat nulle part – elles ne font qu'agir sur les drapeaux. Pour cette raison, le suffixe *S* est redondant et ne doit pas être utilisé.

## *Assignations et allocations en mémoire*

Écrire une instruction réserve implicitement 4 octets en mémoire. Toutefois, il est également possible de réserver ou assigner des valeurs arbitraires à n'importe quel endroit dans la mémoire. Deux mots clés sont disponibles à cette fin. Il est à noter que ces mots clés **ne sont pas des instructions** : ils ne génèrent aucun code ARM en tant que tel, mais se contentent d'initialiser ou de réserver certaines sections de la mémoire. Ces directives doivent être précédées d'une étiquette qui sert de nom de variable.

— Le mot clé `ALLOC` permet **d'allouer** une certaine quantité de mémoire, sans l'initialiser (c'est-à-dire que vous ne pouvez pas déterminer ses valeurs initiales). Sa syntaxe est `ALLOCT NOMBRE`, où `T` est un nombre déterminant la taille des éléments à allouer et `NOMBRE` le nombre d'éléments de cette taille à allouer. `T` est exprimé en bits, et peut prendre les valeurs 8, 16 ou 32.

  Par exemple, la ligne `mavariable ALLOC32 5` allouera 5 espaces de 32 bits (4 octets) chacun ; l'étiquette *mavariable* permettra d'y faire référence dans le code. De même `monoctet ALLOC8 1` allouera 1 espace de 8 bits, soit un octet au total, avec *monoctet* comme étiquette de référence.

— Le mot clé `ASSIGN` permet **d'assigner** une valeur à une ou plusieurs adresses mémoire. Sa syntaxe est `ASSIGNT VAL1, VAL2, ..., VALN`, où `T` est un nombre déterminant la taille des éléments en mémoire et `VAL1, VAL2, ..., VALN` une liste de valeurs séparées par des virgules de longueur arbitraire, qui seront écrites à la suite dans la mémoire. Tout comme pour `ALLOC`, `T` est exprimé en bits, et peut prendre les valeurs 8, 16 ou 32.

  Par exemple, la ligne `mavariable ASSIGN32 8, 0x12, 0, -1` assignera la valeur 8 (représentée sur 32 bits) aux 4 premiers octets, la valeur 0x12 aux 4 suivants, la valeur 0 aux 4 suivants et la valeur -1 aux 4 suivants, pour un total de 16 octets assignés.

  La directive `ASSIGN` accepte également des chaînes de caractères écrites entre guillemets. Par exemple, la ligne `machaine ASSIGN8 "ceci est du texte"` assignera un octet pour chaque caractère ASCII du texte contenu entre les guillements.

Notons qu'une étiquette ne peut être déclarée ou assignée **qu'une seule fois**. Par exemple, faire suivre `mavariable ALLOC32 5` de `mavariable ASSIGN32 8, 0x12, 0, -1` est une erreur, puisque `mavariable` est alors utilisée deux fois.

## *Instructions d'accès mémoire*

### *Accès mémoire de base*

Il existe deux types d'accès mémoire (avec leur mnémonique) :

- `LDR`, qui lit une donnée de la mémoire et l'écrit dans un registre ;
- `STR`, qui lit un registre et écrit sa valeur dans la mémoire ;

Plusieurs suffixes peuvent être ajoutés à ces opérations pour modifier leur comportement (en plus du suffixe conditionnel) :

- Le suffixe `B`, qui permet de ne lire qu'un seul octet (au lieu de 4)
- Le suffixe `H`, qui permet de ne lire que 2 octets (au lieu de 4)
- Dans le cas de LDR, les suffixes `SB` et `SH`, qui lisent respectivement 1 et 2 octets, mais en *étendant le signe de la valeur lues au reste du registre*. Par exemple, lire la valeur $0xFF$ en mode `B` écrira la valeur $0x000000FF$ dans le registre de destination. Faire la même chose en mode `SB` écrira plutôt $0xFFFFFFFF$ (le MSB de l'octet lu est dupliqué à gauche).

Note : l'utilisation des suffixes `H`, `SH` ou `SB` empêche l'utilisation des cas 3, 5, 6 et 7 de la liste suivante ou d'un décalage.

### *Format d'une instruction LDR ou STR*

Les instructions d'accès mémoire sont du format : `LDR REG, INFOMEM` où REG est le registre source (dans le cas de STR) ou de destination (dans le cas de LDR), et INFOMEM une expression déterminant une adresse mémoire. Celle-ci peut prendre sept formes :

1. [REGBASE] : la valeur contenue dans REGBASE est directement interprétée comme une adresse
2. [REGBASE, REGADD] : les valeurs de REGBASE et REGADD sont additionnées pour former une adresse. REGADD peut optionnellement être décalé, mais seulement par constante (voir tableau 2). Si le ] est suivi d'un point d'exclamation (!), la somme de REGBASE et REGADD est écrite dans REGBASE par après
3. [REGBASE, #CONST] : les valeurs de REGBASE et CONST sont additionnées pour former une adresse. Un point d'exclamation peut ici aussi être ajouté pour stocker la somme dans REGBASE
4. [REGBASE], REGADD : la valeur de REGBASE est utilisée comme adresse, mais REGADD est additionné à cette valeur *après* l'accès mémoire, et le résultat stocké dans REGBASE. REGADD peut optionnellement être décalé (voir tableau 2), mais seulement par constante (pas par registre)

5. [REGBASE], #CONST : la valeur de REGBASE est utilisée comme adresse, mais CONST est additionné à cette valeur *après* l'accès mémoire, et le résultat stocké dans REGBASE
6. LABEL : l'assembleur transforme l'étiquette LABEL en un accès relatif à PC. L'étiquette doit être définie ailleurs dans le programme.
7. =LABEL : l'assembleur transforme =LABEL en un accès relatif à PC vers *l'adresse* de l'étiquette LABEL (l'équivalent d'un pointeur en C). Cette étiquette doit être définie ailleurs dans le programme. Ce mode ne peut être utilisé qu'en conjonction avec `LDR` (il est interdit de l'utiliser avec une instruction `STR`).

*Exemples d'utilisation*

*`LDR R0, [R1]`* Copie dans R0 la valeur contenue dans l'adresse mémoire désignée par R1 et les 3 octets suivants (4 octets au total).

*`LDR R0, [R1, #4]`* Copie dans R0 la valeur contenue dans l'adresse mémoire calculée par R1+4 et les 3 octets suivants (4 octets au total).

*`LDR R0, [R1, R2]`* Copie dans R0 la valeur contenue dans l'adresse mémoire calculée par R1+R2 et les 3 octets suivants (4 octets au total).

*`LDR R0, [R1, R2, LSR #2]`* Copie dans R0 la valeur contenue dans l'adresse mémoire calculée par R1+N et les 3 octets suivants (4 octets au total), où N est la valeur de R2 décalée deux fois vers la droite.

*`LDR R0, [R1, R2]!`* Copie dans R0 la valeur contenue dans l'adresse mémoire calculée par R1+R2 et les 3 octets suivants (4 octets au total). Par la suite, stocke le résultat de R1+R2 dans R1.

*`LDRB R0, [R1]`* Copie dans R0 la valeur de l'octet contenu dans l'adresse mémoire désignée par R1.

*`LDRGEB R0, [R1], R2`* Copie dans R0 la valeur de l'octet contenu dans l'adresse mémoire désignée par R1. Additionne R1+R2 et stocke le résultat dans R1. Ces opérations ne sont effectuées que si la condition GE est rencontrée.

*`LDR R0, mavariable`* Copie dans R0 la valeur contenue dans l'adresse mémoire désignée par l'étiquette *mavariable* et les 3 octets suivant.

*`LDR R0, =var2`* Copie dans R0 *l'adresse de* l'étiquette *var2*.

`LDRH R0, [R1]` Copie dans R0 la valeur contenue dans l'adresse mémoire désignée par R1 et l'octet suivant (2 octets au total).

`STR R0, [R1]` Copie les 4 octets composant la valeur de R0 à l'adresse mémoire désignée par R1 (ainsi que les 3 octets suivants).

`STRB R0, [R1]` Copie l'octet le moins significatif de la valeur de R0 à l'adresse mémoire désignée par R1.

`STREQ R0, [R1], #8` Copie l'octet le moins significatif de la valeur de R0 à l'adresse mémoire désignée par R1. Additionne 8 à R1 et stocke le résultat dans R1. Ces opérations ne sont effectuées que si la condition EQ est rencontrée.

`STR R0, mavariable` Copie les 4 octets composant la valeur de R0 à l'adresse mémoire désignée par l'étiquette *mavariable*.

### *Accès mémoire multiples*

Il est également possible de lire ou écrire plusieurs espaces mémoire contigus simultanément. De manière générale, ce genre d'accès est surtout utilisé en conjonction avec une pile. Les instructions à utiliser sont `PUSH` et `POP`. Ces deux instructions prennent une liste de registre en paramètre, entourés d'accolages ({}). Les registre listés sont stockés dans la mémoire, de la manière suivante :

1. Le registre SP (R13) est décrémenté de 4 octets
2. La valeur du registre dont l'index est **le plus élevé** est stockée à cette adresse (par exemple, si on a R5, R8 et R11, c'est la valeur de R11 qui sera écrite).
3. Le registre SP est décrémenté de quatre octets, et le cycle recommence avec les registres restant à écrire.

L'utilisation de POP est similaire, mais la mémoire est lue au lieu d'être écrite et SP incrémenté au lieu d'être décrémenté. L'ordre des registres dans la liste n'a donc aucune importance, seule leur présence (ou absence) est prise en compte.

### *Exemples d'utilisation*

`PUSH {R0, R3-R7}` Stocke les registres R0, R3, R4, R5, R6 et R7 à partir de l'adresse pointée par SP (en descendant la mémoire)

`POPNE {R4, R5, R12}` Lit la mémoire et copie ses valeurs dans R4, R5 et R12 à partir de l'adresse pointée par SP (en remontant la mémoire), mais seulement si la condition NE est rencontrée.

### *Échange mémoire/registre (swap)*

Un dernier type d'accès, plus rare, est un accès *d'échange*, qui écrit et lit simultanément le même espace mémoire. L'instruction à utiliser est *SWP*, dont la syntaxe est la suivante : `SWP REG1, REG2, [REG3]`. REG3 doit contenir l'adresse où l'on souhaite lire et écrire, REG2 le contenu que l'on souhaite y écrire et REG1 le registre de destination pour la valeur lue. La valeur est lue dans REG1 avant d'être modifiée par l'écriture de REG2. Un suffixe `B` peut optionnellement être ajouté pour ne lire/écrire qu'un seul octet (au lieu de 4).

### *Exemples d'utilisation*

*`SWP R3, R8, [R9]`* Copie la valeur contenue à l'adresse mémoire contenue dans R9 dans R3, écrit la valeur de R8 à cette même adresse

*`SWPPLB R1, R2, [R10]`* Copie la valeur de l'octet contenu à l'adresse mémoire contenue dans R10 dans R1, écrit la valeur de l'octet le moins significatif de R8 à cette même adresse, mais seulement si la condition PL est rencontrée

## *Instructions de branchement*

Ces instructions permettent de modifier la valeur de PC afin de changer le fil d'exécution du programme. Elles sont souvent utilisées en conjonction avec une condition pour réaliser un bloc *IF*. Il existe trois types d'instruction de branchement :

1. `B LABEL`, qui assigne PC à l'adresse de l'étiquette LABEL
2. `BL LABEL`, qui opère la même opération, mais stocke de plus l'adresse de l'instruction *suivant le BL* dans le registre LR (R14). Cette instruction est souvent utilisée pour appeler une fonction en conservant l'adresse de retour.
3. `BX REG`, qui copie la valeur du registre REG dans PC. Cette instruction est souvent utilisée avec le registre LR pour revenir d'une fonction.

### *Exemples d'utilisation*

*`B main`* Branche inconditionnellement à l'adresse de l'étiquette *main*

*`BEQ maboucle`* Branche à l'adresse de l'étiquette *maboucle*, mais seulement si la condition EQ est remplie

*`BL mafonction`* Branche inconditionnellement à l'adresse de l'étiquette *mafonction*, écrit l'adresse de l'instruction suivant ce `BL` dans le registre LR

*`BX LR`* Copie la valeur de LR dans PC et poursuit l'exécution

*`BXLE R3`* Copie la valeur de R3 dans PC, mais seulement si la condition LE est rencontrée

### *Instructions de décalage*

On peut utiliser une instruction `MOV` pour réaliser n'importe quel type de décalage. Toutefois, pour faciliter la programmation, un ensemble de cinq *pseudo-instructions* sont fournies et permettent d'écrire directement une opération de décalage :

— `LSL REG1, REG2, REG3` ou `LSL REG1, REG2, #CONST` : décale REG2 de REG3 ou #CONST positions vers la gauche, stocke le résultat dans REG1. Équivalent à `MOV REG1, REG2, LSL REG3`.

— `LSR REG1, REG2, REG3` ou `LSR REG1, REG2, #CONST` : décale REG2 de REG3 ou #CONST positions vers la droite, stocke le résultat dans REG1. Équivalent à `MOV REG1, REG2, LSR REG3`.

— `ASR REG1, REG2, REG3` ou `ASR REG1, REG2, #CONST` : décale REG2 de REG3 ou #CONST positions vers la droite (en copiant la valeur du MSB sur tous les bits insérés), stocke le résultat dans REG1. Équivalent à `MOV REG1, REG2, ASR REG3`.

— `ROR REG1, REG2, REG3` ou `ROR REG1, REG2, #CONST` : décale REG2 de REG3 ou #CONST positions vers la droite en réinsérant les bits à la gauche du registre, stocke le résultat dans REG1. Équivalent à `MOV REG1, REG2, ROR REG3`.

— `RRX REG1, REG2` : décale REG2 de une position vers la droite en utilisant le drapeau Retenue (C) comme nouvelle valeur de MSB. Équivalent à `MOV REG1, REG2, RRX`.

Note : ces instructions étant des pseudo-instructions, leur encodage est strictement le même qu'une instruction MOV équivalente. Le tableau 2 présente en détails chaque type de décalage.

#### *Exemples d'utilisation*

*`LSL R9, R10, R4`* Décale R10 de R4 positions vers la gauche, stocke le résultat dans R9

*`LSREQ R1, R2, #6`* Décale R2 de 6 positions vers la droite, stocke le résultat dans R1, mais seulement si la condition EQ est rencontrée.

## *Instructions d'accès au registre de contrôle et de statut*

Deux instructions permettent d'accéder et de modifier les registres de contrôle et de statut (CPSR et SPSR). Ces instructions sont les suivantes :

— `MRS REG, STATUS` : transfère la valeur du registre STATUS (qui peut être CPSR ou SPSR) dans le registre REG

— `MSR STATUS, REG` : transfère la valeur du registre REG dans le registre STATUS (qui peut être CPSR ou SPSR)

— `MSR STATUS_flg, REG` ou `MSR STATUS_flg, #CONST` : transfère la valeur du registre REG ou de la constante CONST dans le registre STATUS (qui peut être CPSR ou SPSR), mais ne fait que agir sur les drapeaux (les autres bits du registre de statut ne sont pas modifiés)

Attention : l'accès au registre SPSR n'est valide que dans les modes qui en comportent un.

### *Exemples d'utilisation*

*`MRS R9, CPSR`* Transfère le contenu du registre CPSR dans le registre R9

*`MRS SPSR, R5`* Transfère le contenu du registre R5 dans le registre SPSR du mode courant

*`MRS CPSR_flg, #0xF0000000`* Transfère la constante 0xF0000000 dans le registre CPSR, mais en n'affectant que les drapeaux. Les drapeaux étant aux bits 28 à 31 du CPSR, cette instruction les assigne tous à 1.

## *Instructions de déclenchement d'une interruption logicielle*

Deux mnémoniques équivalentes permettent de déclencher une interruption logicielle : `SWI` et `SVC`. Ces instructions possèdent *exactement* le même comportement et prennent en paramètre une constante dont la valeur ne peut excéder $2^2$4. Cette valeur est ignorée par le processeur, mais peut être récupérée par l'interruption logicielle et être utilisée comme paramètre.

### *Exemples d'utilisation*

*`SVC #0xCAFE`* Déclenche une interruption logicielle

*`SWI #22`* Déclenche une interruption logicielle

*`SVCEQ #0xCAFE`* Déclenche une interruption logicielle seulement si la condition EQ est rencontrée

### *Assertions*

En plus des instructions ARM et des directives de déclarations mémoire, le simulateur supporte un mécanisme *d'assertion* qui permet de valider le fonctionnement d'un programme. Il se base sur une directive spéciale, `ASSERT`. Lorsque le simulateur rencontre une ligne contenant ce mot-clé, il évalue la ou les expressions qui suivent. Si ces expressions ne sont pas vraies, il arrête la simulation et écrit un message d'erreur indiquant les divergences entre le résultat prévu et obtenu. Si les expressions sont vraies, aucune action n'est prise et le programme se poursuit normalement. On peut combiner plusieurs expressions sur la même ligne en les séparant par des virgules et ces expressions peuvent être de trois types :

— Vérification d'un registre : `REG=VAL`, où `REG` est un registre et `VAL` la valeur qu'il doit posséder, en décimal ou hexadécimal.
— Vérification d'une adresse mémoire : `ADDR=VAL`, où `ADDR` est une adresse mémoire commençant par `0x` et `VAL` la valeur que cette case mémoire doit posséder. Si `VAL` est entre 0 et 255, seule l'octet pointé par cette adresse est comparé ; sinon, ce sont aussi les trois octets suivants qui sont comparés (pour une taille totale de 32 bits ou 4 octets)
— Vérification d'un drapeau : `DRAPEAU=VAL`, où `DRAPEAU` est une lettre *majuscule* désignant un drapeau (C, V, Z ou N) et `VAL` la valeur (0 ou 1) attendue

### *Exemples d'utilisation*

*`ASSERT R0=9`* Teste si R0 vaut bel et bien 9

*`ASSERT R2=0x18, R1=-5`* Teste si R2 vaut bel et bien 0x18 et R1 -5

*`ASSERT 0x16=24`* Teste si la valeur à l'adresse 0x16 de la mémoire est bien 24

*`ASSERT 0x1004=-1`* Teste si les valeurs aux adresses 0x1004 à 0x1007 inclusivement valent bien -1 (0xFFFFFFFF)

*`ASSERT C=1,Z=0`* Teste si le drapeau Retenue (*Carry*) est bien levé et le drapeau Zéro inactif

### *Note importante*

Les assertions ne sont pas des instructions ARM et n'apparaissent donc pas dans le bytecode. Elles ne peuvent pas être copiées d'un endroit à l'autre de la mémoire comme le peuvent être des instructions.

## *Remerciements*

Les auteurs tiennent à remercier les personnes suivantes pour l'aide qu'ils ont apporté à la réalisation de ce simulateur :

— Jessica Déziel, pour le design de l'interface graphique

— Jonathan Gilbert, pour les tests préliminaires du simulateur et la réalisation des exercices

— Étienne Dubeau, pour son travail à l'amélioration du simulateur et à l'implémentation de nouvelles fonctionnalités