

Mémoire vidéo GBA, calculs d'adresses en assembleur Indicateurs de code condition NZCV

Modèle mémoire de l'écran

La mémoire vidéo de la GBA (VRAM) est accessible directement en lecture et en écriture à partir de l'adresse 0x06000000.

En mode 3, que nous utiliserons exclusivement en TD/TP et pour le projet, l'écran a une largeur de 240 pixels, une hauteur de 160 pixels, et chaque pixel est représenté par un demi-mot (2 octets) en mémoire vidéo selon le format BGR 15 bits.

Les pixels sont stockés comme un texte : de gauche à droite et de haut en bas.

On peut calculer l'adresse du pixel à lire ou écrire, à partir des coordonnées, selon la formule $adresse = 0x06000000 + 2*x + 480*y$

Calcul d'adresse pixel à partir de coordonnées : Un exemple de calcul en assembleur ARM

On souhaite réaliser ce calcul en assembleur ARM. On suppose que les coordonnées x et y sont stockées respectivement dans les registres r0 et r1. On obtiendra le résultat, adresse du pixel, dans le registre r2. Une première implémentation possible est donnée par le code suivant :

```

mov    r2,#0x06000000    @ r2 <- 0x06000000  initialisation de r2 avec l'adresse de base de la mémoire vidéo
mov    r3,#2             @ r3 <- 2      r3 prend la valeur 2 pour multiplier x ...
mul    r0,r3,r0          @ r0 <- r3 * r0  x(dans r0) est multiplié par 2: r0 vaut maintenant 2*x
add    r2,r2,r0          @ r2 <- r2 + r0  r2 vaut maintenant 0x06000000 + 2*x
mov    r3,#480           @ r3 <- 480    r3 prend la valeur 480 pour multiplier y ...
mul    r1,r3,r1          @ r1 <- r3 * r1  y(dans r1) est multiplié par 480: r1 vaut maintenant 480*y
add    r2,r2,r1          @ r2 <- r2 + r1  r2 vaut maintenant 0x06000000 + 2*x + 480*y
    
```

La documentation précise que l'instruction de multiplication `mul` nécessite l'emploi de registres comme opérandes. Il est donc impossible de multiplier directement par une constante : `mul r0,r0,#2 @ Impossible : n'est pas une instruction du microprocesseur ARM`
Ce qui explique l'emploi d'un registre supplémentaire (ici r3) comme stockage temporaire pour multiplier un autre registre par une constante.

En revanche les instructions `mov`, `add`, `sub`, `rsb` ... acceptent une constante en dernier opérande¹.

Ainsi il est possible d'ajouter directement une constante : `add r6,r7,#2 @ Possible : on ajoute 2 à r7, résultat dans r6`

En effectuant les opérations dans un ordre différent et en utilisant cette possibilité : proposez un code équivalent qui n'utilise que 6 instructions au lieu de 7, et si possible qui ne modifie pas le contenu de départ des registre r0 et r1 (r0 et r1 conservent leur valeur).

Rappels concernant les instructions usuelles sur registres :

<code>mov rd,<Oprnd2></code>	@ rd <- Oprnd2	Mouvement de donnée : le registre destination prend la valeur de l'opérande
<code>add rd,rn,<Oprnd2></code>	@ rd <- rn + Oprnd2	Opération d'addition
<code>sub rd,rn,<Oprnd2></code>	@ rd <- rn - Oprnd2	Opération de soustraction
<code>rsb rd,rn,<Oprnd2></code>	@ rd <- Oprnd2 - rn	Opération de soustraction en ordre inverse (<i>reverse sub</i>)
<code>mul rd,rm,rs</code>	@ rd <- rm * rs	Opération de multiplication : les 2 opérandes rm et rs sont nécessairement des registres

Multiplications rapides par décalages et addition ou soustraction

Un décalage à gauche de n bits correspond à une multiplication par 2ⁿ.

Un décalage à droite de n bits correspond à une division (quotient entier) par 2ⁿ.

Il est possible d'effectuer directement, sans instruction supplémentaire, un décalage à gauche (`lsl`) ou à droite (`lsr` ou `asr`) ou une rotation à droite (`ror`) sur un registre en dernier opérande pour les instructions `mov`, `add`, `sub`, `rsb`

Exemples :

```

mov    r4,r5,lsr #2      @ r4 prend la valeur de r5 décalé à droite de 2 bits
add    r6,r7,r8,lsl #3   @ r6 prend la valeur de r7 + ( r8 décalé à gauche de 3 bits )
    
```

Déterminer les opérations arithmétiques réalisées par les instructions ARM suivantes (on cherche l'expression de r0 en fonction de r1) :

```

mov r0,r1,lsl #5      add r0,r1,r1,lsl #6      rsb r0,r1,r1,lsl #3      add r0,r1,r1,lsl #16
    
```

Est-il possible d'implémenter le calcul de l'adresse d'un pixel en assembleur ARM sans utiliser d'instruction `mul` ?

En combien d'instructions ?

Sachant qu'une instruction `mul` consomme au moins 3 cycles horloge pour être exécutée alors que les autres instructions de calcul utilisées ici n'en prennent qu'un seul (même en utilisant un décalage), combien de cycle économise-t-on par rapport à la 1ère implémentation du TD ?

Indicateurs de Code Condition NZCV

Sur une instruction assembleur postfixée par S(*set*) ou une instruction `CMP`, les 4 indicateurs binaires NZCV sont modifiés en fonction du résultat.

N : résultat négatif (bit de signe à 1) Z : résultat nul, C (resp. V) : résultat tronqué en interprétation non signée (resp. interprétation signée).

Pour l'exercice nous travaillons avec un additionneur de 8 bits de large. **Compléter NZCV pour les 3 derniers exemples :**

Hexa	u8	s8	NZCV	Hexa	u8	s8	NZCV	Hexa	u8	s8	NZCV	Hexa	u8	s8	NZCV
0xBA	186	-70		0xCE	206	-50		0x32	50	50		0x32	50	50	
+ 0xA6	+ 166	+ -90		+ 0x3C	+ 60	+ 60		+ 0xCE	+ 206	+ -50		+ 0x78	+ 120	+ 120	
0x60	96	96	0011	0x0A	10	10	0x00	0	0	0xAA	170	-86

¹ Ces constantes numériques, préfixées par #, sont appelées « valeurs immédiates », valeurs qui doivent respecter certaines contraintes (sur ARM) :

De 0 à 256 : toutes valeurs possibles De 256 à 1024 : multiples de 4 De 1024 à 4096 : multiples de 16 ...

On peut vérifier que toutes les valeurs immédiates de cet exercice sont bien compatibles avec ces contraintes.