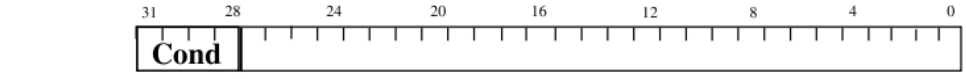
The Condition Field



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

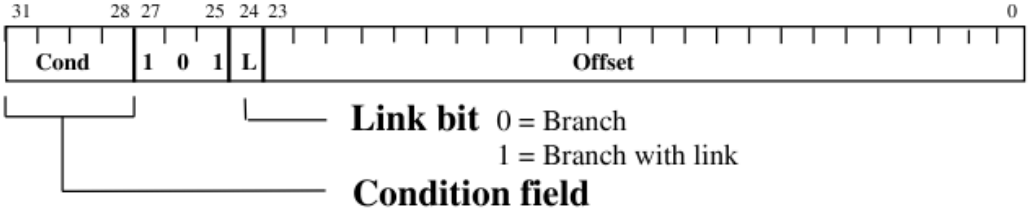
1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

Branch instructions (1)

* **Branch :** `B{<cond>} label`
* **Branch with Link :** `BL{<cond>} sub_routine_label`



B, BAL, BRANCH sont des synonymes

BCC et BCS selon l'état de C

Principales conditions : EQ/NE, CS/CC, MI/PL, VS/VC, HI/LS(lowe orr same), GE/LT et GT/LE

Arithmetic Operations

* **Operations are:**
  - ADD operand1 + operand2
  - ADC operand1 + operand2 + carry
  - SUB operand1 - operand2
  - SBC operand1 - operand2 + carry -1
  - RSB operand2 - operand1
  - RSC operand2 - operand1 + carry - 1
* **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
* **Examples**
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

RSB est l'inverse de SUB ➔ operande 2 – operande 1
SUB et RSB peuvent influencer le C

Comparisons

* **The only effect of the comparisons is to**
  - ***UPDATE THE CONDITION FLAGS.*** Thus no need to set S bit.
* **Operations are:**
  - CMP operand1 - operand2, but result not written
  - CMN operand1 + operand2, but result not written
  - TST operand1 AND operand2, but result not written
  - TEQ operand1 EOR operand2, but result not written
* **Syntax:**
  - <Operation>{<cond>} Rn, Operand2
* **Examples:**
  - CMP r0, r1
  - TSTEQ r2, #5

CMP r0, r7 ➔ lève les drapeaux en fonction de r0-r7

Logical Operations

* **Operations are:**
  - AND operand1 AND operand2
  - EOR operand1 EOR operand2
  - ORR operand1 OR operand2
  - BIC operand1 AND NOT operand2 [ie bit clear]
* **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
* **Examples:**
  - AND r0, r1, r2
  - BICEQ r2, r3, #7
  - EORS r1,r3,r0

Data Movement

* **Operations are:**
  - MOV operand2
  - MVN NOT operand2

  Note that these make no use of operand1.
* **Syntax:**
  - <Operation>{<cond>}{S} Rd, Operand2
* **Examples:**
  - MOV r0, r1
  - MOVS r2, #10
  - MVNEQ r1,#0

MVN est move negate ➔ en inversant les bits

Multiplication Instructions

* **The Basic ARM provides two multiplication instructions.**
* **Multiply**
  - MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs
* **Multiply Accumulate - does addition for free**
  - MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn
* **Restrictions on use:**
  - Rd and Rm cannot be the same register

Load / Store Instructions

* **The ARM is a Load / Store Architecture:**
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.
* **This might sound inefficient, but in practice isn't:**
  - Load data values from memory into registers.
  - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
  - Store results from registers out to memory.
* **The ARM has three sets of instructions which interact with main memory. These are:**
  - Single register data transfer (LDR / STR).
  - Block data transfer (LDM/STM).
  - Single Data Swap (SWP).

LDR rd, #immediate ➔ valeur de taille limitée (16 bits je crois)
LDR r0, [r1] ➔ charge la valeur à l'adresse indiquée par r1
STR r0, [r1] ➔ écrit ……
LDR r0, [rn, #offset] ➔ charge ce qui se trouve à rn+offset
LDR r0, [rn], #offset ➔ charge ce qui est à rn et rn = rn + offset

Stacks and Subroutines

* **One use of stacks is to create temporary register workspace for subroutines. Any registers that are needed can be pushed onto the stack at the start of the subroutine and popped off again at the end so as to restore them before return to the caller :**

```
STMFD sp!,{r0-r12, lr}          ; stack all registers
........                        ; and the return address
........
LDMFD sp!,{r0-r12, pc}          ; load all the registers
                                ; and return automatically
```

BL label ➔ appel de la routine label
POP liste de registres
PUSH liste de registres ➔ push et pop peuvent être conditionnels, mais ce n'est pas une bonne idée !
STMFD et LDMFD s'appliquent à des piles descendantes (D) et Full Meaning.
STM et LDM sont pour load/store multiple et permettent de manipuler plusieurs valeurs. Dans l'exemple, la commande STMFD emplie les 13 registres usagers et le link register. À la sortie de la routine, on dépile le lr dans pc (program counter) et les 13 registres.

Une routine devrait donc toujours débuter par
STMFD sp!, {liste des registres à empliler et lr} et se terminer par :
LDFFD sp!, {liste des registres empilés et lr}
C'est préférable d'utiliser STMFD et LDFFD que de gérer manuellement (MOV pc,lr).

Déclarations de constantes

```
Xyz     DCB 34   ; déclare une constante « byte »
abc     DC  1234   ; déclare une constante « halfword » ➔ 16 bits
pqr     DCD 0x3f  ; déclare une constante 32 bits
yz      DCB  34, 'A', 99, 0xA ; déclare un tableau de 4 octets
yz      DCB ''hello'', 0 ; déclare une null terminated string
.equ  n  5 ; déclare la constante n qui vaut 5
```

Déclarations de variables ➔ .data

```
unTableau :     DS32 8 ; un tableau de 8 mots ➔ des entiers
unAutre :       DS8  64 ; un tableau de 64 octets
libre : .space 32 ; réserve 32 octets        .
```

Table 4. SWI I/O operations (0x00 - 0xFF)

Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x00	Display Character on Stdout	r0: the character		SWI_PrChr
swi 0x02	Display String on Stdout	r0: address of a null terminated ASCII string	(see also 0x69 below)	
swi 0x11	Halt Execution			SWI_Exit
swi 0x12	Allocate Block of Memory on Heap	r0: block size in bytes	r0:address of block	SWI_MeAlloc
swi 0x13	Deallocate All Heap Blocks			SWI_DAlloc
swi 0x66	Open File (mode values in r1 are: 0 for input, 1 for output, 2 for appending)	r0: file name, i.e. address of a null terminated ASCII string containing the name r1: mode	r0:file handle If the file does not open, a result of -1 is returned	SWI_Open
swi 0x68	Close File	r0: file handle		SWI_Close
swi 0x69	Write String to a File or to Stdout	r0: file handleor Stdout r1: address of a null terminated ASCII string		SWI_PrStr

Table 4. SWI I/O operations (0x00 - 0xFF)

Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x6a	Read String from a File	r0: file handle r1: destination address r2: max bytes to store	r0: number of bytes stored	SWI_RdStr
swi 0x6b	Write Integer to a File	r0: file handle r1: integer		SWI_PrInt
swi 0x6c	Read Integer from a File	r0: file handle	r0: the integer	SWI_RdInt
swi 0x6d	Get the current time (ticks)		r0: the number of ticks (milliseconds)	SWI_Timer

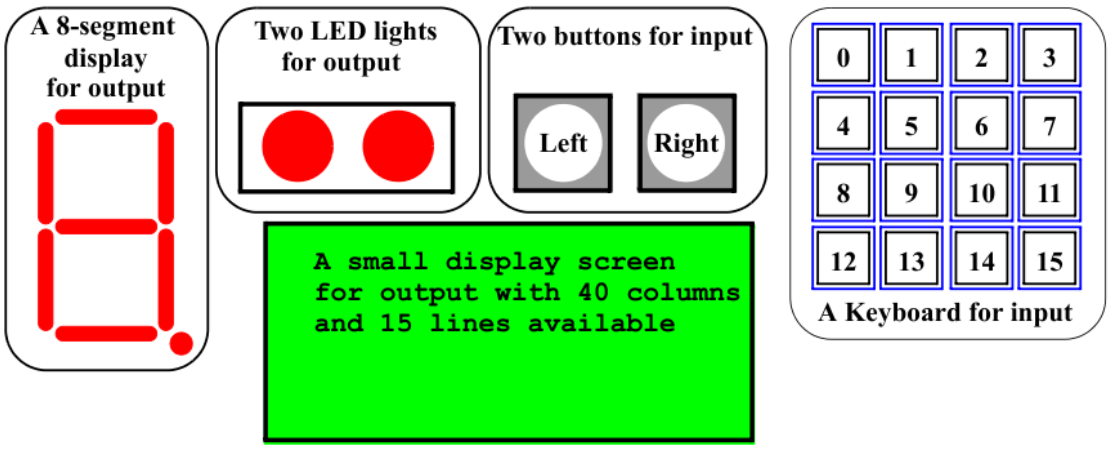


Figure 14. A diagrammatic view of the available controls and displays on the Embest Board View

Opcode	Description and Action	Inputs	Outputs
`swi 0x200`	Light up the 8-Segment Display.	r0: the 8-segment Pattern (see below in Figure 15 for details)	The appropriate segments light up to display a number or a character
`swi 0x201`	Light up the two LEDs .	r0: the LED Pattern, where: Left LED on = `0x02` Right LED on = `0x01` Both LEDs on = `0x03` (i.e. the bits in position 0 and 1 of r0 must each be set to 1 appropriately)	Either the left LED is on, or the right, or both
`swi 0x202`	Check if one of the Black Buttons has been pressed.	None	r0 = the Black Button Pattern, where: Left black button pressed returns r0 = `0x02;` Right black button pressed returns r0 = `0x01;` (i.e. the bits in position 0 and 1 of r0 get assigned the appropriate values).
`swi 0x203`	Check if one of the Blue Buttons has been pressed.	None (see below in Figure 19 for details)	r0 = the Blue Button Pattern (see below in Figure 19).
`swi 0x204`	Display a string on the LCD screen	r0: x position coordinate on the LCD screen (0-39); r1: y position coordinate on the LCD screen (0-14); r2: Address of a null terminated ASCII string. Note: (x,y) = (0,0) is the top left and (0,14) is the bottom left. The display is limited to 40 characters per line.	The string is displayed starting at the given position of the LCD screen.

Figure 15. The Pattern for the 8-Segment Display

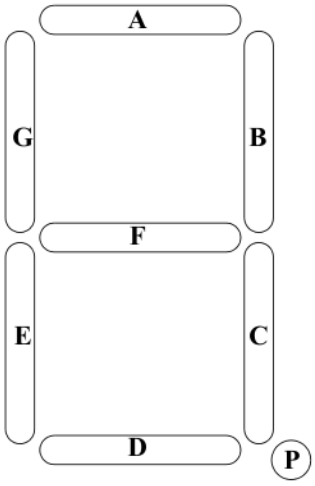
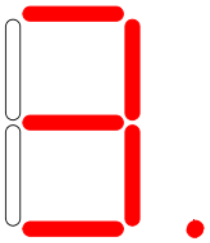


Table 6: Segmen t byte values

Display byte values	
A	0x80
B	0x40
C	0x20
P	0x10
D	0x08
E	0x04
F	0x02
G	0x01

Example: number "3" plus "dot"



As an example, the number 3 plus the right hand dot would have a pattern value computed as the logical OR of the values of the segments "A,B,C,D,F,P" to form the integer: `0x80 | 0x40 | 0x20 | 0x08 | 0x02 | 0x10 = 0xFA`, to be assigned to r0.

Use "`.equ`" statements to set up the byte value of each segment of the Display.

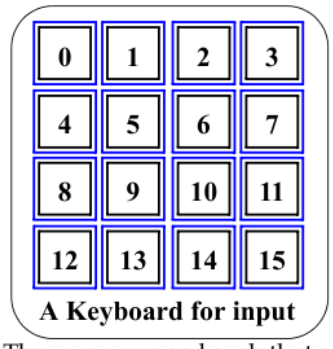
```
.equ    SEG_A,0x80
.equ    SEG_B,0x40
.equ    SEG_C,0x20
.equ    SEG_D,0x08
.equ    SEG_E,0x04
.equ    SEG_F,0x02
.equ    SEG_G,0x01
.equ    SEG_P,0x10
```

Figure 16. Possible data declaration for byte values for segments

A possible data declaration for an array of words which can be indexed to obtain the appropriate value for a number {0,...,9} to be displayed.

```
Digits:
   .word    SEG_A|SEG_B|SEG_C|SEG_D|SEG_E|SEG_G @0
   .word    SEG_B|SEG_C             @1
   .word    SEG_A|SEG_B|SEG_F|SEG_E|SEG_D @2
   .word    SEG_A|SEG_B|SEG_F|SEG_C|SEG_D @3
   .word    SEG_G|SEG_F|SEG_B|SEG_C @4
   .word    SEG_A|SEG_G|SEG_F|SEG_C|SEG_D @5
   .word    SEG_A|SEG_G|SEG_F|SEG_E|SEG_D|SEG_C @6
   .word    SEG_A|SEG_B|SEG_C       @7
   .word    SEG_A|SEG_B|SEG_C|SEG_D|SEG_E|SEG_F|SEG_G @8
   .word    SEG_A|SEG_B|SEG_F|SEG_G|SEG_C @9
   .word    0                       @Blank display
```

Figure 17. Possible data declaration forinteger patterns



They are arranged such that each button has a corresponding bit position in the 16-bit lower portion of a word in register R0 after the call to swi 0x203 to poll the keypad. The "*number labels*" placed in the figure, which do not appear on the real keypad, also represent the corresponding bit position as returned in R0. When a button is pressed, the corresponding bit is set.

For example, when the button in position "1" is pressed, the `swi 0x203` instruction returns `r0 = 0x02,` that is,

`r0 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0`

in binary, where the bit in position "1" has been set.

Figure 19. The Pattern for the Blue Buttons

11.4 Example: Subroutine to implement a wait cycle with the 32-bit timer

```
@ Wait(Delay:r2) wait for r2 milliseconds
Wait:
     stmfdsp!, {r0-r1,lr}
     swi  SWI_GetTicks
     mov  r1, r0            @ R1: start time
WaitLoop:
     swi  SWI_GetTicks
     subs r0, r0, r1        @ R0: time since start
     rsbltr0, r0, #0        @ fix unsigned subtract
     cmp  r0, r2
     blt  WaitLoop
WaitDone:
     ldmfdsp!, {r0-r1,pc}
```

Table 5. SWI operations greater than 0xFF as currently used for the Embest board Plug-In

Opcode	Description and Action	Inputs	Outputs
`swi 0x205`	Display an integer on the LCD screen	r0: x position coordinate on the LCD screen (0-39); r1: y position coordinate on the LCD screen (0-14); r2: integer to print. Note: (x,y) = (0,0) is the top left and (0,14) is the bottom left. The display is limited to 40 characters per line	The string is displayed starting at the given position of the LCD screen.
`swi 0x206`	Clear the display on the LCD screen	None	Blank LCD screen.
`swi 0x207`	Display a character on the LCD screen	r0: x position coordinate on the LCD screen (0-39); r1: y position coordinate on the LCD screen (0-14); r2: the character. Note: (x,y) = (0,0) is the top left and (0,14) is the bottom left. The display is limited to 40 characters per line	The string is displayed starting at the given position of the LCD screen.
`swi 0x208`	Clear one line in the display on the LCD screen	r0: line number (y coordinate) on the LCD screen	Blank line on the LCD screen.

TP ARM à effectuer pour le 9 novembre.

Introduction

Nous allons construire ici une application pour un processeur ARM simulé dans ARCSIM. L'application consiste à simuler une machine distributrice.

Pour construire notre distributrice, nous aurons à :

- Déclarer des constantes
- Déclarer des variables
- Déclarer et programmer des routines utilisant 0, un paramètre ou deux paramètres
- Écrire un code principal
- Utiliser les périphériques fournis par **ArmSim** (ex. écran, clavier, bouton)

Bref, ce sera un petit programme très instructif à écrire.

Consignes

À chaque étape, vous devez fournir un fichier source. Ce fichier devrait indiquer en commentaire :

- toutes les modifications apportées (et les raisons) par rapport à l'étape précédente
- l'utilité de chacune des lignes de code.

Je dis de remettre un fichier .s, mais ce pourrait aussi être un fichier Word ou autre avec de la couleur, des caractères gras et/ou italiques afin d'attirer l'attention sur les éléments importants.

Il pourrait aussi y avoir un texte explicatif.

J'ai subdivisé la tâche en une dizaine d'étapes. Je suggère des opérations à effectuer à chaque étape. Vous devez suivre assez fidèlement celles-ci. Toutefois, j'accepterais une conception ou des choix différents à condition que les fonctionnalités et le comportement de l'application soient les mêmes.

Nous effectuerons quelques étapes en classe afin de nous faire la main.

Étape #1 Partir de quelque chose qui fonctionne

Utiliser le programme de la figure 20 du tutoriel de ArcSim comme point de départ. Additionnez directement R0 et R1 au lieu d'appeler une routine externe.

Testez votre code et sauvegardez-le dans P01.s

```
@ File: AddMain.s
        .text
        .global _start
```

```
        .extern myAdd
_start:
        LDR   R0,=Num1
        LDR   R0,[R0]  @ first parameter passed in R0
        LDR   R1,=Num2
        LDR   R1,[R1]  @ second parameter passed in R1
        ADD   R0, R0, R1
        LDR   R4, =Answer
        STR   R0,[R4]  @ result est écrit en mémoire
        SWI   0x11     @ arrêter le programme

        .data
Num1:  .word 537
Num2:  .word -237
Answer: .word 0
        .end
```

Étape #2 : Déclarez un tableau de chaines de caractères

L'écran de ArmSim fournit un écran simulé de 15 lignes de 40 caractères. Déclarez les chaînes suivantes qui décrivent les produits disponibles dans la distributrice.

0123456789012345678901234567890123456789 // Cette ligne sert à identifier les colonnes

CODE	DESCRIPTION	PRIX	DISP
1	Chips	1.25	10
2	Chocolat	1.50	20
3	Fromage	2.95	10
4	Gateau	1.60	8
5	Yogourt	1.25	12
6	Lait	1.40	6
7	Muffin	1.80	8
8	Arachides	2,00	20
9	Bonbons	1.25	15

Chaque produit a un code, une description, un prix et le nombre d'unités disponibles. Puisque le nombre d'unités disponibles peut changer, vous devrez soit allouer vos chaînes dans la zone data (afin de pouvoir en changer le contenu), soit créer des constantes chaînes sans les disponibilités et ajouter ces valeurs par la suite à l'affichage. Personnellement, j'opterais pour la seconde solution.

Testez votre code pour vous assurer que vos déclarations sont légales et sauvegardez-le dans p02.s

```
@ File: AddMain.s
        .text
```

```
        .global _start
        .extern myAdd
_start:
        LDR   R0,=Num1
        LDR   R0,[R0]  @ first parameter passed in R0
        LDR   R1,=Num2
        LDR   R1,[R1]  @ second parameter passed in R1
        ADD   R2, R0, R1
        LDR   R4, =Answer
        STR   R2,[R4]  @ result was returned in R2
        MOV   r0,#0
        mov   R1,#0
        LDR   R2,=produit
        LDR   R2,[r2]
        SWI   0x204
        SWI   0x11

        .data

@ Déclaration des produits
p0:     .asciz "Code   Description  Prix   Disp."
p1:     .asciz "1          Chips        1.25"
p2:     .asciz "2          Chocolat    1.50"
p3:     .asciz "3          Fromage    2.95"
p4:     .asciz "4          Gateau      1.60"
p5:     .asciz "5          Yogourt     1.25"
p6:     .asciz "6          Lait          1.40"
p7:     .asciz "7          Muffin      1.80"
p8:     .asciz "8          Arachides   2.00"
p9:     .asciz "9          Bonbons     1.25"

@ Déclaration d'un tableau de chaines pour le 9 produits
produit:
        .word p1
        .word p2
        .word p3
        .word p4
        .word p5
        .word p6
        .word p7
        .word p8
        .word p9
```

@ Déclaration du tableau contenant les quantités disponibles
disp:
.word 10,20,10,8,12,6,8,20,15

@ Déclaration du tableau des prix des produits

prix:
.word 125, 150, 295, 160, 125, 140, 180, 200, 125

Num1: .word 537
Num2: .word -237
Answer: .word 0

Étape #3 : Déclaration des autres variables et tableaux

Nous aurons besoin de plusieurs variables pour écrire notre application. Commençons par déclarer les plus évidentes :

- Le **solde** : l'usager insèrera de la monnaie et fera des achats. Il faut comptabiliser l'argent inséré et tenir compte des achats. Pour faire simple, nous représenterons la solde en cents. Le solde ne peut dépasser 9,95$ soit 955. Toute pièce de monnaie qui amènerait le solde au-delà de cette somme doit être refusée.
- Le **tableau des quantités** : Pour chaque item nous avons besoin de suivre les quantités en inventaire. Les quantités initiales sont indiquées à l'étape #2. Mais, au fur et mesure des achats, il faut tenir à jour ces quantités. Nous déclarons donc un tableau de 9 entiers que nous initialisons tout de suite ou plus tard (dans le code d'initialisation).
- Le **tableau des prix** : même si nous avons les prix dans les chaînes de caractères, il serait plus pratique de les maintenir (aussi ?) dans un tableau d'entiers.
- Le **nombre de produits** : ici c'est 9.

Testez votre code pour vous assurer que vos déclarations sont légales et sauvegardez-le dans p03.s

solde: .word
np: .word 9
.end

Étape #4 : Écriture du code principal

Le code principal du programme ressemble à ceci :

1 – Initialiser mes variables, ports, etc.
2 - Afficher l'inventaire ➔ afficher les produits et les quantité disponibles
3 – Afficher le solde à la ligne 12 de l'écran

4 – Tant qu'il reste quelque chose en inventaire

4.1 – Y a-t-il une touche du clavier pesée ?

OUI : traiter la touche

4.2 – Le bouton #1 a-t-il été pesé?

OUI : Traiter la commande en cours

4.3 – Le bouton #2 a-t-il été pesé?

OUI : Retourner la monnaie

Écrivez et testez votre code pour vous assurer que celui-ci est valide et sauvegardez-le dans p04.s

Notez que vous ferez appel à des routines pour les étapes 1, 2, 3 et les sous étapes de l'étape #4. Ici, ne déclarez que les routines qui retournent immédiatement (sans rien faire).

```
@ File: AddMain.s
        .text
        .global _start
        .extern myAdd

_start:

        @ r10 contient le code la touche pesée --> 0 au départ
        BL    initialise
loop:   @ boucle principale qui tourne tant que nProd > 0
        @ 1 - Y a-t-il une touche pesée?
        swi   0x203       @ vérifier si un touche a été pesée
        CMP   r0, #0
        BEQ   bouton1   @ NON --> passer aux boutons
        BL    keypressed  @ OUI, traiter la touche
        b     findeboucle

bouton1:
        swi   0x202     @ Un bouton a été pesé?
        cmp   r0, #0    @ NON --> boucler
        BEQ   findeboucle
        cmp   r0, #1    @ OUI : Est-ce le bouton #1 ?
        BNE   bouton2  @ NON ==> bouton2 a été pesé --> donner le change
        bl    proceedcommande @ OUI : traiter la commande
        b     findeboucle
bouton2:
        b     change    @ donner la monnaie

findeboucle:
        @ A-t-on fini?
        LDR   r1, =nProd @ Charger le nombre de produits en inventaire
```

```
        LDR   r1, [r1]
        CMP   r1, #0     @ S'il n'y a plus rien, quitter
        BNE loop

        SWI   0x11  @ arrêter le programme

@ ================================================
@ routine qui procède à l'initialisation des variables
@ La seule variable à initialiser est le nombre de produits
@    en inventaire (disp). Pour ce faire, il faut additionner
@    les inventaires de chaque produit.
initialise:

        STMFD sp!,{lr}
        @ Le code de la routine
        LDMFD sp!,{pc}

@ ======================================
@ routine qui traite une touche du clavier
@ Ici, on détermine quelle touche a été pesée.
@ Si c'est un chiffre de 1 à 9, on inscrit le chiffre dans r10.
@ Si c'est de la monnaie, on incrit le montant dans r10
@ Si c'est une touche illégale, on met r10 à 0
keypressed:
        STMFD sp!,{lr}
        MOV   r10, #1   @ pour le moment
        LDMFD sp!,{pc}
```

Étape #5 : Écrire une routine qui affiche l'inventaire

À l'étape #2 du programme principal, vous appelez une routine qui :

1. Efface l'écran
2. Affiche la ligne d'entête
3. Affiche chacun des produits avec sa quantité

Pour la sous-étape #3, il faudrait une routine qui prend en paramètre un numéro de produit, qui affiche ses infos et lui ajoute la quantité en inventaire.

Écrivez donc la routine AfficheProduit(no) et la routine AfficheInventaire() qui fait appel à cette routine.
Testez votre code et sauvegardez-le dans p05.s

Étape #6 : Écrire une routine qui affiche le solde à la ligne 12 de l'écran

À la suite de l'inventaire, on doit afficher le solde. Celui-ci est, à l'interne, en cents (un entier en mémoire). Toutefois, on voudrait un affichage du style :

VOTRE SOLDE EST DE 2.25$

Il vous faut donc écrire le code permettant de générer une chaine à partir d'un entier représentant des cents.

Écrivez donc la routine AfficheSolde() .
Testez votre code et sauvegardez-le dans p06.s

Pour vous aider :

```
; ============= UDiv ====================================
; Routine qui effectue une division entière (non-signée).
; Cette routine ne perturbe le contenu d'aucun registre à
; l'exception de
;       r0 : le reste de la division
;       r1 : le quotient
;       r2 : code d'erreur (1 --> division par zéro)
; Pour utiliser la routine, placer le dividende dans r0 et
; le diviseur dans r1
UDiv:
        STMFD sp!, {r4, lr}
        MOV         r2, #0          ; par défaut, pas d'erreur
        MOVS        r1, r1          ; tester si le diviseur = 0
        BNE         DivOK
        MOV         r2, #1          ; mettre le code d'erreur à 1
        BAL         EndDiv
DivOK:      MOV     r4, #0    ; init le quotient à 0

PasFini:
        ADD         r4, r4, #1      ; incrémenter le quotient
        SUBS        r0, r0, r1      ; Dividende = dividende – diviseur
        BCS         PasFini         ; Dividende >= 0 ==> pas fini!
        ADD         r0, r0, r1      ; On a soustrait une fois de trop ==> restaurer
        SUB         r4, r4, #1  ;   On corrige le quotient
        MOV         r1, r4          ; le quotient est mis dans r1

EndDiv:
        LDMFD sp!, {r4, pc}
```

Étape #7 : traiter les touches du clavier

Si une touche a été pesée, il peut s'agir d'un nombre de 1 à 9 (touches 0,1,2,4,5,6,8,9,10) ou d'une des touches 3, 7, 11, 15.

Les touches 1 à 9 permettent à l'usager de faire un achat d'un produit.

Les touches 3, 7, 11, 15 permettent respectivement de simuler l'insertion de 2,00$, 1,00$, 0,25 et 0,10.

Les autres touches sont illégales (il faudrait afficher un message pendant un court moment).

Écrivez la routine TraiterLaTouche(???) qui vérifie quelle touche a été pesée, la valide et stocke le caractère dans une variable.

Testez votre code et sauvegardez-le dans p07.s

Étape #8 : Le bouton #1 a-t-il été pesé?

Le bouton #1 confirme l'achat d'un item ou l'insertion de monnaie. Il faut, si un caractère a été entrée et qu'il était valide (voir étape précédente) procéder à l'achat ou à l'insertion d'argent.

Pour un achat, vous devez, si le solde était suffisant, diminuer de 1 l'inventaire du produit sélectionné et diminuer le solde du prix du produit. Vous devez ensuite réafficher votre inventaire et le nouveau solde. Si le solde était insuffisant, vous devez afficher un message approprié pendant quelques secondes.

Si de l'argent est inséré, vous devez calculer le nouveau solde et l'afficher. Pour faire simple, nous représenterons la solde en cents. Le solde ne peut dépasser 9,95$. Toute pièce de monnaie qui amènerait le solde au-delà de cette somme doit être refusée. Vous devez afficher un message approprié pendant quelques secondes.

Vous pouvez choisir de traiter les touches illégales ici au lieu de l'étape précédente.

Écrivez les routines Button1Pressed() qui sera appelée par le programme principal ainsi que les routines Achat(int noProduit) et Insert(int montant). Validez vos routines et sauvegardez votre code dans p08.s

Étape #9 : Le bouton #2 a-t-il été pesé?

Le bouton #2 demande de rendre la monnaie. Donc, s'il a été pesé, vous mettez le solde à 0 et réaffichez le solde.

Écrivez le code de la routine Button2Pressed() et sauvegardez votre programme dans p09.s

Étape #10 : Valider le tout

Normalement, votre code est terminé et fonctionne.

Validez l'ensemble de votre programme et indiquez les corrections que vous y avez apportées le cas échéant. Sauvegarder le code final dans p10.s

À remettre :

- **Les différents fichiers de code.**
- **Un rapport indiquant votre cheminement.**
- **Le tout pour le 9 novembre.**
- **Prendre rendez-vous, à compter du 9 novembre, pour la correction.**
- **Le travail peut être fait en équipe de deux personnes. Si vous avez reçu une aide quelconque, vous devez l'indiquer dans le rapport d'étape.**

Toute collaboration non avouée sera considérée comme du plagiat.