**Bilan - Assembleur ARM**

**Le modèle de mémoire**

**Les registres**

15 registres de 32 bits (32/8=4 octets) : usage général, r0 à r14

1 registres de 32 bits (32/8=4 octets) : compteur de programme, r15

1 resgistre d’état : CPSR

1 octet = 8 bits

ex : 1010 0011 2^7 + 2^5 + 2^1 + 2^0 = 163 dans 1 octet

1 Ko = 1024 bits

code conditions représenté par :

bits de poids faible : mode de fonctionnement, jeu d’instructions, activations d’interruptions

bits de poids fort :

* N (négative) : N=1, la dernière opération qui a mis à jour les codes conditions a produit un résultat négatif
* Z (Zero) : Z=1, la dernière opération qui a mis à jour les codes conditions a prodiot un résultat nul
* C (Carry) : C=0, la dernière opération qui a mis à jour les codes conditions a généré une retenue sortante sur une opération arithmétique ou sur un décalage
* V (overflow) : V=1, la dernière opération qui a mis à jour les codes conditions a produit un débordement sur le bit de signe

Valeurs Caractères de remplacement : 10 A 11 B 12 C 13 D 14 E 15 F

**La mémoire**

un tableau linéaire d’octets numérotés de 0 à 2^32-1

mots : 4 octets = 32 bits

demi-mots : 2 octets = 16 bits

1 octet = 8 bits

gérée selon le mode little endian(de petit à grand)

**Le jeu d’instructions ARM**

ARM de type load-store

load : transférer une valeur depuis la mémoire vers un registre

store : inversement

**opérations arithmétiques**

ADD r0,r1,r2 @r0 <- r1 + r2 addtion

ADC r0,r1,r2 @r0 <- r1 + r2 + C addtion avec retenue

SUB r0,r1,r2 @r0 <- r1 - r2 soustraction

SBC r0,r1,r2 @r0 <- r1 - r2 + C - 1 soustraction avec retenue

RSB r0,r1,r2 @r0 <- r2 - r1 reverse substraction

RSC r0,r1,r2 @r0 <- r2 - r1 + C -1 renverse substraction

**Opérations logiques**

AND r0,r1,r2 @r0 <- r1 et r2 and

ORR r0,r1,r2 @r0 <- r1 ou r2 or

EOR r0,r1,r2 @r0 <- r1 ouexc1 r2 exclusive-or

BIC r0,r1,r2 @r0 <- r1 et non r2 bit clear

**Mouvements entre registre**

première opérande en entrée (il est omis et copient le seconde opérande vers la destination

MOV r0,r2 @r0 <- r2

MVN r0,r2 @r0<-non r2

**Comparaisons**

CMP r1,r2 @CPSR <- cc(r1 - r2) compare

CMN r1,r2 @CPSR <- cc(r1 + r2) compare negated

TST r1,r2 @CPSR <- cc(r1 et r2) test

TEQ r1,r2 @CPSR <- cc(r1 +O r2) test equal

**Opérandes registres décalés**

ADD r3,r2,r1,LSL, #3 @r3 <- r2+r1\*2^3

ADD r5,r5,r3,LSL,r2 @r5 <- r5+r3\*2^r2

LSL logical shift left

décalage de 0 à 31 positions vers la gauche avec introduction de 0

LSR logical shift right

décalage de 0 à 32 positions vers la droite avec introduction de 0

ASL arithmetic shift leftt

identique à LSL

ASR arithmetic shift right

décalage de 0

ROR rotate right

décalage de 0 à 43 positions circulaire vers la droite

RRX rotate right extended

décalage d’une position vers ma droite avec introduction du bit C

**Opérandes immédiats**

ADD r3,r3,#1 @r3 <- r3+1

AND r8,r7,#0xFF @r8 <- r7 @0x pour une constante en hexa, ob pour binaire

0xFF = 0000 0000 0000 0000 0000 0000 1111 1111

1111 = 2^3 + 2^2 + 2^1 + 2^0 = 15 donc F

si la valeur de la forme (0 à 255) \* 2 ^2n : MOV rd, #valeur

sinon : LDR rd, [r15, #depl], cste : .int <valeur>, depl = cste - etiq - 8

**Multiplications**

MUL r4,r3,r2 @r4 <- r3 \* r2

le seconde opérande ne peut pas être une valeur immédiate

le registre résultat ne peut pas être identique au premier registre opérande

si le bit S est positionné, le code V n’est pas modifié et le code C est non significatif

MLA r4,r3,r2,r1 @r4 <- r3\*r2+1

ADD r0,r0,r0,LSL,#2 @r0 <- r0 + r0 \* 2^2

RSB r0,r0,r0,LSL,#3 @r0 <- 7\*r0

**Mise à jour des codes condition**

ADDS r2,r2,r0 @retenue dans C

ADC r3,r3,r1 @ajouter la retenue au mot de poids fort

**Instruction de transfert de données**

load : mémoire -> registre store : registre -> mémoire

**Adressage indirect par registre**

LDR r0,[r1] @r0 <- mem[r1]

STR r0,[r1] @mem[r1] <- r0

**Initialisation d’un pointeur**

avant un transfert de donnée, initialiser un registre avec l’adresse correspondante

ex : .fill, .int, .byte

il n’est pas envisageable d’utiliser une instruction de mouvement MOV d’une valeur immédiate vers un registre car il est probable que l’adresse ne respecte pas les restrictions.

calculer l’adresse de la donné en ajoutant une constante d au compteur de programme (PC), PC est égal à l’adresse de l’instruction + 8 (traitement des instruction en pipeline)

ADD r3,pc,#12 @var = PC + 12 = début + 8 + 12

ADR <registre destination>,<étiquette>

ex : adr r3, var

**Adressage “base + déplacement”**

ajouter un déplacement pour calculer l’adresse du transfert

mode pré-indexé : LDR r0,[r1,#4] @r0 <- mem[r1+4]

mode auto-indexé : LDR r0,[r1,#4]! @r0 <- mem[r1+4], r1 <- r1+4

mode post-indexé : LDR r0,[r1],#4 @r0 <- mem[r1], r1<-r1+4

**Transfert d’octets**

LDRB r2,[r1] @prendre les deux dernières bits depuis le mémoire

STRB r2,[r1] @stocker les deux dernières bits dans le mémoire

**Transferts multiples**

transférer plusieurs registres de ou vers la mémoire

LDMIA r1,{r0,r2,r5} @r0 <- mem[r1], r2 <- mem[r1+4], r3 <- mem[r1+8]

**Adressage de pile**

une pile est une zone de mémoire allouée dynamiquement et gérée en mode “dernier entré, premier sorti”, le modèle full descending dans lequel ma pile grandit vers les adresses décroissantes et le pointeur de pile contient l’adresse du dernier élément rangé dans la pile

STMFD r13!,{r0,r1,r5} @empiler

LDMFD r13!,{r0,r1,r5} @dépiler