PF1 — Principes de Fonctionnement des machines binaires

Jean-Baptiste Yunès
Jean.Baptiste.Yunes@univ-paris-diderot.fr

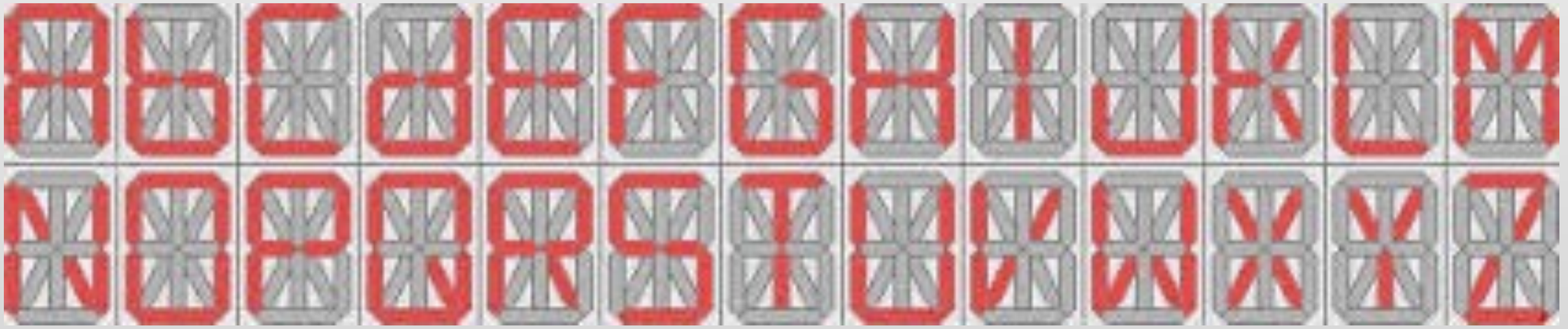
Version 1.0

Q: l'afficheur à 7 segments pour les chiffres décimaux codés **en ASCII** :

écrire les fonctions associés à chaque segment sous la FND

simplifier les fonctions par la méthode de Karnaugh

Q: même question pour les chiffres hexadécimaux :



Q: résoudre la question avec le code ASCII et un afficheur 14 segments (lettres et chiffres). On ne distinguera pas les majuscules des minuscules…

Q: résoudre la question avec un afficheur 7 segment double (deux affichages possibles) pour les nombres de 0 à 99 codés en DCB (BCD)

Q: Fabriquer un circuit permettant d’additionner deux nombres de deux bits.

Ajouter un indicateur de dépassement de capacité

Retour sur l’addition et la multiplication binaire

Prenons la table d’addition des chiffres de la base 2

+	0	1
0	0	1
1	1	(1)0

C’est un connecteur d’arité 2 (si on oublie la retenue)

Il s’agit même du $\oplus$ (xor)

Pour la table de multiplication on a

.	0	1
0	0	0
1	0	1

C’est aussi un connecteur d’arité 2

Il s’agit du $\wedge$ (et)

Le problème de la retenue pour l'addition…

Il suffit de voir l'addition binaire com
deux fonctions d'arité 3. L'une calcu
le résultat et l'autre la retenue (suiva
en utilisant les deux bits et une rete
(précédente)

a	b	r	+
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

a	b	r	R
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Programmation Assembleur

De quoi sont constitués (principalement pour ce qui nous concerne) les ordinateurs ?

d'un **processeur** effectuant des opérations élémentaires

d'une **mémoire**

les deux s'échangeant des informations continument…

Les ordinateurs sont des circuits complexes mais dont le principe de fonctionnement est assez simple

- le cœur est le processeur dont le rôle est d'exécuter des instructions en provenance de la mémoire
  - une instruction est **lue** depuis la mémoire et interprétée en : lisant des valeurs depuis la mémoire, **sélectionnant** la partie du circuit permettant de réaliser la fonction booléenne correspondante, **écrivant** le résultat en mémoire
    - les instructions sont codées, le code permet par l'intermédiaire d'un décodeur de sélectionner le circuit qui réalisera le calcul

Les codes machines sont des mots binaires

par exemple pour un processeur de la famille i386, l'addition de l'octet de valeur 37 avec le registre AL se code sur 16 bits par :

0000010000011001 (très sensible aux erreurs)

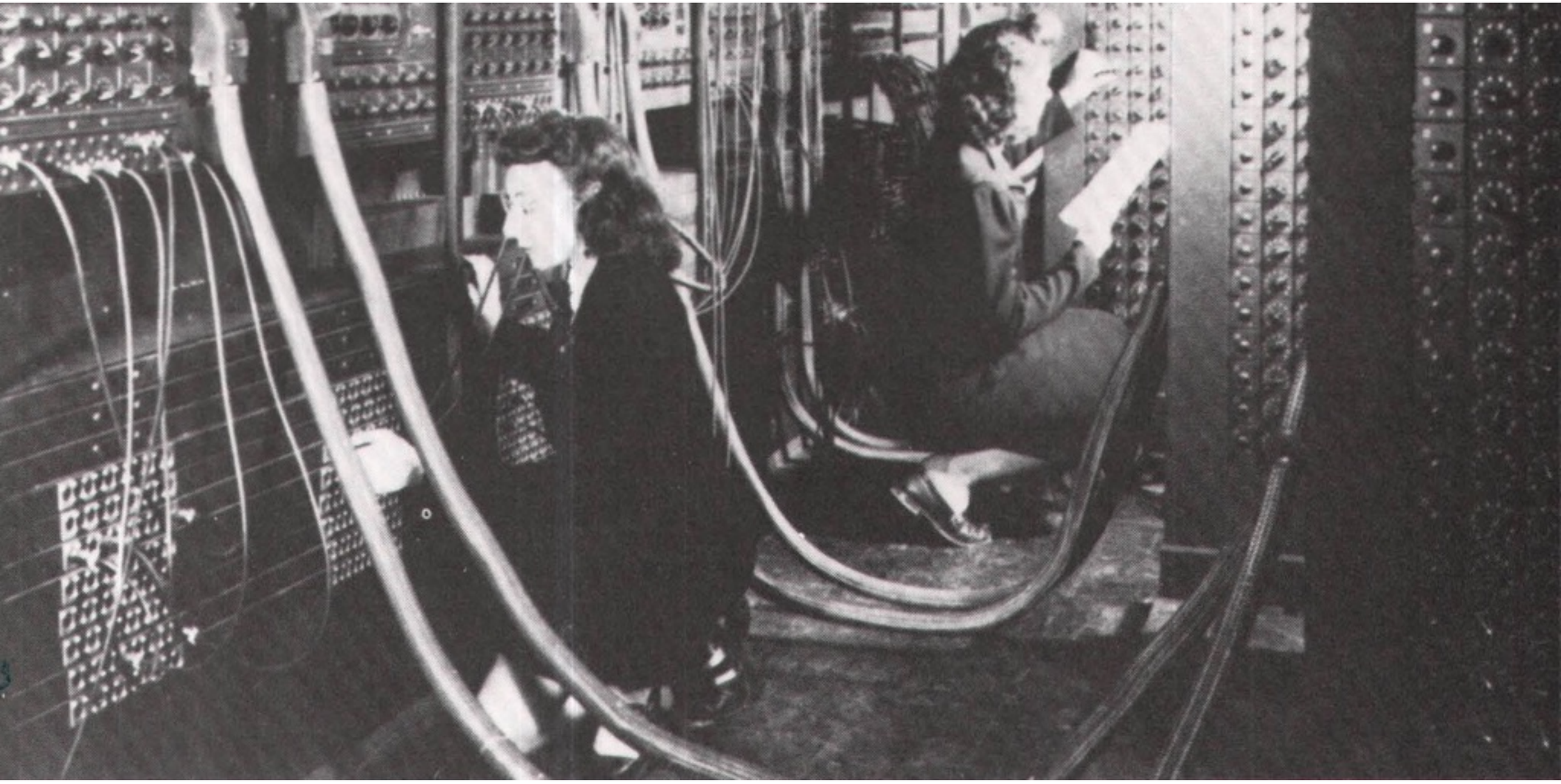
1049 (pas très parlant)

soit 0x04 0x25 (à peine plus lisible), ou 4 37

hèque Haverford

nombre dans l’ENIAC…

hèque Haverford



n programme dans l’ENIAC…

L’assembleur est un langage plus agréable pour les humains, la même instruction s’écrit

```
ADD AL,25h
```

c’est un peu ésotérique, mais on comprend déjà qu’il s’agit d’une ADDition faisant intervenir le registre AL et la valeur 25 en hexadécimal

La source de « l’ésotérisme » est qu’à ce niveau là, ne ne savons pas grand chose des processeurs de la famille i386, ni grand chose des processeurs en général

Les unités de mémorisation

Les ordinateurs actuels ont de nombreuses unités de mémorisation (vives) dont :

Les **registres** : très rapides (1ns) mais en nombre ridicule (<100)

Les **caches** (cache) : relativement rapide (10ns) mais de taille assez faible (~Ko-Mo)

La **mémoire vive** (RAM) : relativement lente (50ns) mais plus volumineuse (~Go)

La **mémoire vive** :

L’unité est l’octet

Chaque octet a une **adresse** qui lui est propre

Les **registres**

L’unité est le mot (couramment 64 bits)

Chaque registre a un **nom** qui lui est propre

Le langage machine

une instruction du langage machine est directement interprétable par le processeur

la collection des instructions disponibles est appelé **jeu d'instructions**

une **instruction** contient plusieurs informations codées :

le **code de l'instruction** à réaliser (correspond essentiellement à la désignation des circuits nécessaires)

les **adresses des opérandes** dans les unités de mémoire

L’exécution d’une instruction consiste en les opérations :

(**fetch**) aller chercher l’instruction situé à l’adresse contenu dans le compteur ordinal (registre de nom PC)

L'exécution d'une instruction consiste en les opérations :

(**decode**) exécuter l'instruction après décodage en :

chargeant depuis les unités de mémoire les opérandes dans l'unité de mémoire des opérandes (registres ou pile)

(**execute**) sélectionnant les parties nécessaires de la circuiterie de sorte que

les opérandes en entrée soient chargées dans l'unité de mémoire des opérandes

les circuits réalisent la fonction booléenne en lisant les

les sorties des circuits soient injectées dans l'unité de mémoire des opérandes

chargeant éventuellement les opérandes de sortie situées dans l'unité de mémoire des opérandes dans les unités de mémoire concernées

L’exécution d’une instruction consiste en les opérations :

passer à l’instruction suivante (généralement passage en séquence, mais parfois saut "if-then")

Les unités de mémoire des opérandes peuvent être de trois types :

registre

accumulateur (registre d’usage particulier)

pile (structure de donnée qui sera étudiée plus tard)

Le rôle de l'assembleur (l'outil pas le langage)

Une instruction Java comme x = y+z; est traduite en langage machine par des instructions du genre :

(pile) `push y; push z; add; pop x;`

(accumulateur) `load y; add z; store x;`

(registres) `load r1,y; load r2,z; add r1,r2; store x,r1`

Le rôle de l’assembleur (l’outil pas le langage)

Dans ces exemples x, y et z restent sous forme symbolique. Dans la réalité il s’agit d’emplacements en mémoire, des adresses…

la spécification de ces adresses peut être faite de différentes manières…

Il existe différents modes d'adressages en mémoire (différentes façons de désigner une valeur) :

le mode **immédiat** : la valeur de l'opérande est la valeur qui devra être utilisée

additionne 3 et 4

les constantes dans les programmes

Il existe différents modes d'adressages en mémoire (différentes façons de désigner une valeur) :

le mode **implicite** : l'emplacement où est stockée la valeur est implicite : le sommet de la pile, l'accumulateur, peut importe… n'importe quoi de fixé à l'avance

ajoute 3 à l'accumulateur

les calculs intermédiaires

Il existe différents modes d'adressages en mémoire (différentes façons de désigner une valeur) :

- le mode **registre** : la valeur de l'opérande est le numéro d'un registre dans lequel la valeur est stockée

  additionne le contenu du registre R_0 et 3, range le tout dans R_1

- les calculs intermédiaires

Il existe différents modes d'adressages en mémoire (différentes façons de désigner une valeur) :

- le mode **direct** : la valeur de l'opérande est l'adresse d'un emplacement en mémoire dans lequel la valeur est stockée

  range le contenu de R0 en mémoire à l'adresse 0xBADBEEF

- les variables

Il existe différents modes d'adressages en mémoire (différentes façons de désigner une valeur) :

le mode **indexé** ou relatif : la valeur de l'opérande est l'adresse relative d'un emplacement mémoire dans lequel la valeur est stockée. Le point d'origine est en général contenu dans un registre spécial (dit registre d'index)

range la valeur 3 à l'adresse située à R_0 emplacements à partir de l'adresse 0x30000

les tableaux…

Il existe différents modes d'adressages en mémoire (différentes façons de désigner une valeur) :

le mode **indirect** : la valeur de l'opérande est l'adresse d'un emplacement dans lequel est stockée l'adresse d'un emplacement dans lequel est stockée la valeur

range 3 à l'adresse contenue à l'adresse 0x3000

pointeurs (C/C++), références (Java)

Les instructions d'une machine peuvent être rangées dans les catégories suivantes :

- instructions **arithmétiques**
- instructions **logiques**
- instructions de **lecture/écriture** en mémoire
- instructions de **contrôle** (déroutement du flux normal)
- instructions de **gestion de pile** (attendre le cours TO2 - CI2)

L’assembleur et la machine LC-3

LC-3 : Little Computer 3

un assembleur pédagogique

œuvre de Yale Patt et Sanjay Patel

un livre « *Introduction to Computing Systems: From Bits to C and Beyond* », Mac Graw-Hill

Un simulateur + assembleur de LC-3 existe :

http://www.cis.upenn.edu/~milom/cse240-Fall05/handouts/lc3guide.html

Une machine à mots de 16-bits avec 8 registres de 16 bits et une mémoire vive (RAM), on dit, espace d'adressage de 2^{16} mots.

Les registres sont nommés R0-R7

Une instruction est toujours codée sur 16 bits (un mot)

Arithmétique signée en complément a deux (pas de flottants)

La machine possède un registre d'état (CC)

des indicateurs positionnés par certaines instructions lorsque certaines conditions sont rencontrées (on verra plus loin)

N (valeur négative produite)

P (valeur positive produite)

Z (valeur nulle produite)

La machine possède un pointeur d’instruction (PC) contenant à tout instant l’adresse de la prochaine instruction à exécuter

Mon premier programme

n’est pas "Hello world" (un poil trop complexe en assembleur pour l’instant)

additionne deux nombres en mémoire!

impressionnant non ?

Le programme Java correspondant serait :

```
class Add {
d main(String args[]) {
nt i = 23;
nt j = 78;
  int k;
k = i+j;
   }
  }
```

```
        .ORIG x3000
        LD R0,memi  ; direct
        LD R1,memj  ; direct
        ADD R2,R0,R1
        ST R2,memk  ; direct
        HALT
memi    .FILL #23
memj    .FILL #78
memk    .BLKW 1
        .END
```

Un peu barbare…

Pour l’exécuter :

lancer le simulateur

charger le système (pas nécessaire mais c’est mieux)

load lc3os.obj

assembler le code

as add.asm

charger le code

load add.asm

commencer l’exécution : continue (sans arrêt jusqu’à la fin) ou next (pas-à-pas)

L’assembleur permet d’utiliser des symboles pour désigner des adresses mémoire (données ou instructions)

L’assembleur fournit des pseudos-instructions permettant de déclarer des données, des adresses spéciales, etc

.ORIG : adresse de chargement en machine du programme

.FILL : adresse d’un mot de 16 bits (variable)

.BLKW : adresse d’un groupe de mots de 16 bits

.STRINGZ : adresse d’une chaîne de caractères

.END : fin du code source du programme assembleur

.ORIG x3000

pseudo-instruction permettant de charger le programme assemblé à partir de l’adresse 0x3000

il s’agit de l’adresse standard de chargement dans la machine LC-3

vari .FILL x1234

bidule .FILL xFFFF

déclaration de mots mémoire (16 bits), initialisés respectivement aux valeurs 0x1234 et 0xFFFF

pourront être manipulés via leurs symboles associés vari et bidule

mazone .BLKW #25

déclaration d’une zone de mémoire de 25 mots

note : en LC3, on peut utiliser des valeurs décimales (commençant par un #) ou hexadécimales (préfixées par x)

hello .STRINGZ "Bonjour"

permet de déclarer une zone initialisée avec les caractères de la chaîne.

un caractère NUL est inséré à la fin pour marquer celle-ci

un caractère par mot de 16-bits

Il existe d’autres pseudo-instructions

GETC

OUT

PUTS

IN

PUTSP

HALT : qui permet de stopper la machine

Les instructions de l'assembleur (et qui correspondent donc à des instructions machine) sont

ADD

AND

BR

JMP

JSR, JSRR

LD, **LDI**, **LDR**, **LEA**

NOT

RET, RTI

ST, **STI**, **STR**

TRAP

Addition

ADD dR, sR_1, sR_2

dR = sR_1+sR_2

ADD dR, sR, valeur (où -16≤valeur<16)

dR = sR+valeur

Positionne les indicateurs N,Z,P

Exemples

ADD R0, R1, R2

ADD R0, R0, R0

ADD R0, R1, #10

et bit-à-bit…

AND dR, sR_1, sR_2

$dR = sR_1$ AND sR_2

AND dR, sR, valeur (où -16≤valeur<16)

dR = sR AND valeur

Positionne N, P, Z en accord avec la valeur du résultat

Branchement conditionnel

BRx adresse (où x est n, z, p, nz, np, zp, nzp)

le branchement est réalisé si l'une des conditions demandées est réalisée, PC=adresse

BR adresse

branchement inconditionnel, PC=adresse

Exemples

BRnz OK, saut vers OK si le résultat est négatif ou nul

Branchement via un registre

JMP R

PC=R

Lecture mémoire / Chargement valeur

LD dR, adresse

dR = mem[adresse]

LDI dR, adresse

dR = mem[mem[adresse]]

LDR dR, bR, dep

dR = mem[bR+dep]

LEA dR, adresse

dR = adresse

positionnent N,Z,P

Exemples :

LEA R0, HELLO

LD R1, HELLO

LDI R2, HELLOA

NON bit-à-bit

NOT dR, sR

dR = NOT(sR)

positionne N,Z,P

Exemple :

NOT R2, R0

Écriture mémoire

ST sR, adresse

mem[adresse] = sR

STI sR, adresse

mem[mem[adresse]] = sR

STR sR, bR, depl

mem[bR+depl] = sR

pas de modification de CC

Exemples

ST R0, vari

STI R0, variadresse

STR R0, R1, #5

Appel système

TRAP nt (où nt vaut x20, x21, x22, x23, x24 ou x25)

x20 c’est GETC

x21 OUT

x22 PUTS

x23 IN

x24 PUTSP

x25 HALT

nécessite le chargement préalable du système…

Un exemple :

```
        .orig x3000
        LD R2, Zero
        LD R0, M0
        LD R1, M1
Loop    BRz Done
        ADD R2, R2, R0
        ADD R1, R1, #-1
        BR Loop
Done    ST R2, Res
        HALT
Res     .FILL x0000
Zero    .FILL x0000
M0      .FILL x0007
M1      .FILL x0003
        .END
```

Que fait ce programme ?

```
        .orig x3000
        LD R2, Zero          ; R2=0
        LD R0, M0            ; R0=7
        LD R1, M1            ; R1=3
Loop    BRz Done             ; IF R1==0 GOTO DONE
        ADD R2, R2, R0       ; R2=R2+R0
        ADD R1, R1, #-1      ; R1—
        BR Loop              ; GOTO LOOP
Done    ST R2, Res           ; RES=R2 ; 7*3
        HALT
Res     .FILL x0000
Zero    .FILL x0000
M0      .FILL x0007
M1      .FILL x0003
        .END
```

Que fait ce programm

```
            .ORIG x3000
            AND R6,R6,#0
            AND R7,R7,#0
            LD R1, DEND
            LD R2, ZOR
            NOT R3,R2
            ADD R3, R3,#1
LOOP1       ADD R7,R7,#1
            ADD R1,R1,R3
            BRN INF0
            BRZ ZERO
            BRP LOOP1
INF0        ADD R7,R7,#-1
            ADD R6,R1,R2
ZERO        HALT
DEND        .FILL #25
ZOR         .FILL #4
            .END
```