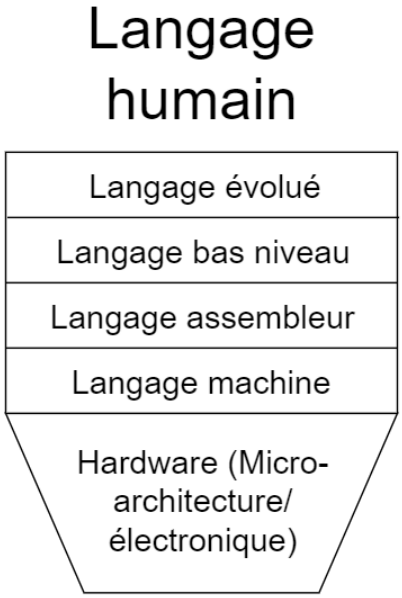
Université Sétif 1
Faculté des Sciences
Département d'Informatique

2-ième année(année2019/20)
Architecture des Ordinateurs
Enseignant Kara Abdelaziz

Solution Série 2
(notions sur les instructions d'un ordinateur)

Exercice 01 :

1. Les différents niveaux de programmation (ou la pile software) sont comme suite :



- Le langage humain est en dehors de la machine (pour l'instant)
- Les langages évolués sont plus proche de l'humain et loin de la machine, comme C++, Java ou Python.
- Les langages bas niveau sont des langages proche de la machine et loin de l'humain, comme le langage C.
- Les langages assembleurs sont des langages quasi-machine, les instructions sont des substituées des instructions machine. Chaque architecture a son propre langage machine, comme x86, ARM, MIPS, 6502.
- Le langage machine est le langage en binaire (0 et 1) qui commande réellement la machine, ça varie aussi d'une architecture à une autre.
- Le hardware c'est l'électronique de la machine, c'est la micro-architecture de la machine.

2. L'optimisation du code logiciel se réalise généralement lors du processus de compilation, ou dans l'une des étapes pour passer du code évolué vers le code machine. On peut citer quelques technique d'optimisations comme suite :

Élimination du code mort : ça peut arriver dans les applications complexes où un test ou une boucle ne s'exécute jamais.

exemple :

```
x = abs(y) ;
…
…
if(x < 0)
{
     … // cette partie du code ne va jamais s'exécuter
     …
}
```

Élimination des sous expressions communes : le compilateur factorise les expressions communes afin de réduire le nombre d'opérations dans le calcul.

exemple :

```
x = (cos(y)-sin(y))/exp(z) ;
y = (cos(y)-sin(y))/log(z) ;
```

devient :

```
t = cos(y)-sin(y) ;
x = t/exp(z) ;
y = t/log(z) ;
```

Pre-évaluation des constantes : le pré-calcul des constantes une seul fois avant exécution.

exemple :

```
const int x = (1+2)/3; // x = 1 ; avant l'exécution
```

Réduction des opérations : utiliser les opérations légères au lieu des opérations lourdes comme la multiplication et la division.

exemple :

```
x = Y * 4 ; // la multiplication utilise plusieurs cycles.
```

devient :

```
x = Y << 2 ; // le shift utilise un seul cycle d'horloge.
```

L'utilisation de l'inline : pour certaines fonctions de petite taille et à grande fréquence d'appel, il est plus optimal de ne pas mettre (encapsuler) le code dans des fonctions, évitant ainsi la mécanique de passage de paramètre et le retour d'appel qui peut alourdir le programme à grande fréquence.

exemple : (`inline` est une commande du C++)

```
inline  int fct(int x, int y)  // fonction inline
{    return x + y ;
}
```

lors de son appel :

```
…
z = fct(1 , 2) ; // les fonctions inline n'ont pas d'appel
…
```

ça devient :

```
…
z = 1 + 2 ; // le code de la fonction est remplacée
…
```

Transformation des appels terminaux en branchements : rassembler 2 fonctions en un seul appel de fonction au niveau assembleur, pour minimiser l'utilisation du mécanisme d'appel. L'appel de la deuxième fonction doit être à la fin (le retour) de la première.

Ré-arrangement du code : optimiser la gestion des registres à l'intérieur du processeur en effectuant des changement dans l'ordre des instructions au niveau assembleur, afin de minimiser le va-et-vient de l'information entre le processeur et la RAM.

Évaluation partielle : c'est de prendre des calculs simples pour des cas particuliers d'un calcul plus générique.

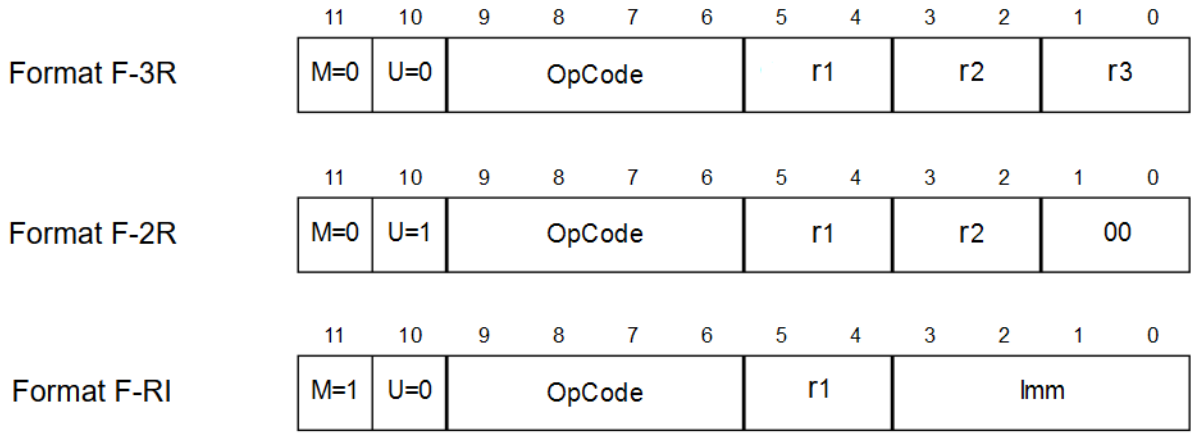
exemple : calculer x^2 par `x*x` au lieu d'appeler la fonction `power(double x,double n)`, car la fonction `power` qui calcule la puissance d'un réel est beaucoup plus complexe qu'une simple opération de multiplication.

Exercice 02 :

Le Jeu d'instructions :

Instruction	OpCode	Format	Syntaxe	Fonctionnement
ADD	0000	ALU1(F-3R)	ADD r1 , r2 , r3	r1 ← r2 + r3
SUB	0001	ALU1(F-3R)	SUB r1 , r2 , r3	r1 ← r2 - r3
AND	0010	ALU1(F-3R)	AND r1 , r2 ,r3	r1 ← r2 & r3
OR	0011	ALU1(F-3R)	OR r1 , r2 , r3	r1 ← r2 | r3
CMP	0100	ALU2(F-2R)	CMP r1 , r2	SR ← r1 -r2
NOT	0101	ALU2(F-2R)	NOT r1 , r2	r1 ← ~r2
SHL	0110	ALU2(F-2R)	SHL r1 , r2	r1 ← r2 << 1
SHR	0111	ALU2(F-2R)	SHR r1 , r2	r1 ← r2 >> 1
LOAD	1000	MEM(F-RI)	LD r1 , [Imm]	r1 ← RAM[Imm]
STR	1001	MEM(F-RI)	STR r1 , [Imm]	RAM[Imm]← r1

Format d'instruction :



Remarque 1: Les 2 derniers bits 10 et 11 sont le code du Format.

A. On a l'instruction suivante : `x ←(y+z)div2 ;//adresse x=5, adresse y=1, adresse z=3`

le code assembleur équivalent à cette instruction est :

```
LD R0 , [1]        // charger y dans R0
LD R1 , [3]        // charger z dans R1
ADD R2 , R0 , R1   // R2<- R0 + R1
SHR R3 , R2        // Le shift à droite est une division par 2
STR R3 , [5]       // retourner le resultat vers x dans la RAM
```

le code machine pour ces 5 instructions est comme suite :

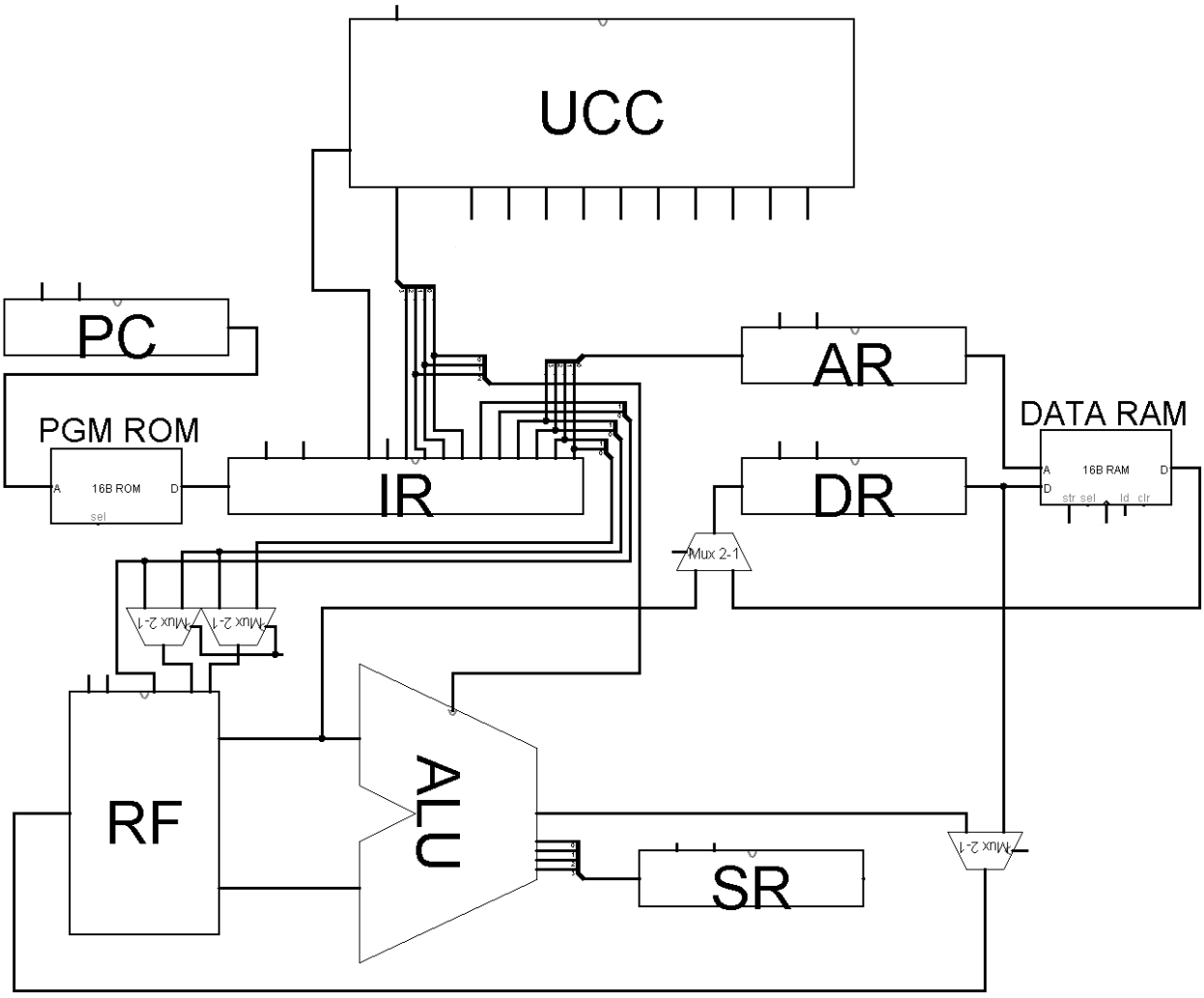
Instruction assembleur	Code machine	Représentation hexadécimal
`LD R0 , [1]`	`10|1000|00|0001`	0xA01
`LD R1 , [3]`	`10|1000|01|0011`	0xA13
`ADD R2 , R0 , R1`	`00|0000|10|00|01`	0x021
`SHR R3 , R2`	`01|0111|11|10|00`	0x5F8
`STR R3 , [5]`	`10|1001|11|0101`	0xA75

Remarque : Le préfixe 0x sur un nombre signifie dans la majorité des langage moderne une représentation du nombre en hexadécimal.

La représentation du code machine de notre programme dans la mémoire en représentation Little Endian et Big Endian est comme sur le table en bas. Sachant que la taille d'une cellule dans notre mémoire est de 4 bits, on aurait besoin de 15 cellules pour nos 5 instructions.

Adresse de la cellule	Little Endian	Big Endian
14	A	5
13	7	7
12	5	A
11	5	8
10	F	F
9	8	5
8	0	1
7	2	2
6	1	0
5	A	3
4	1	1
3	3	A
2	A	1
1	0	0
0	1	A

B. Le datapath :



1. Le mappeur (mapper en Anglais) est un circuit combinatoire qui permet à l'UCC de reconnaître l'adresse du début de la séquence de micro-code dans la ROM de contrôle à partir de l'OpCode de l'instruction. Notre mappeur est décrit sur le tableau en bas.

OpCode de l'instruction	Adresse dans la ROM de contrôle
ALU1	00000
ALU2	00100
CMP	01000
LOAD	10000
STR	10100
Fetch	11000
Direct	11100

Ce qui nous permet de construire le mappeur en utilisant la méthode 5 étapes :

Étape 1 : Schéma global

Étape 2 : Table de Vérité

Op_3	Op_2	Op_1	Op_0	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0
0	1	0	0	0	1	0	0	0
0	1	0	1	0	0	1	0	0
0	1	1	0	0	0	1	0	0
0	1	1	1	0	0	1	0	0
1	0	0	0	1	0	0	0	0
1	0	0	1	1	0	1	0	0
1	0	1	0	-	-	-	-	-
1	0	1	1	-	-	-	-	-
1	1	0	0	-	-	-	-	-
1	1	0	1	-	-	-	-	-
1	1	1	0	-	-	-	-	-
1	1	1	1	-	-	-	-	-

Étape 3 + Étape 4 : Fonctions Canoniques Disjonctives et minimisation

Par observatrion :

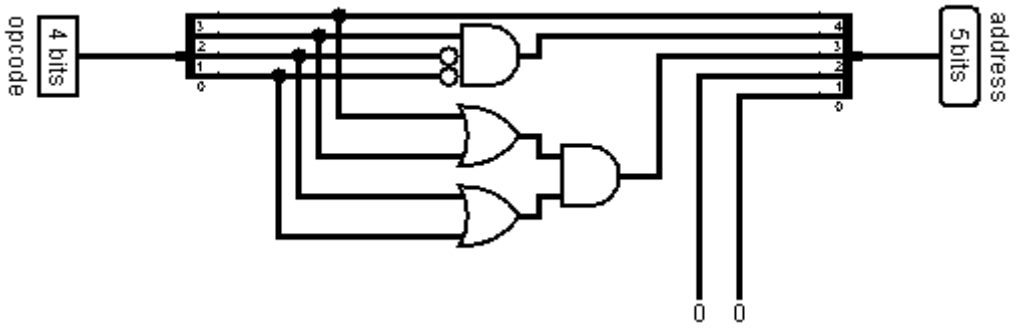
$A_4(Op_3, Op_2, Op_1, Op_0) = Op_3$
$A_1(Op_3, Op_2, Op_1, Op_0) = 0$
$A_0(Op_3, Op_2, Op_1, Op_0) = 0$

Op_1Op_0 \ Op_3Op_2	00	01	11	10
00	0	1	-	0
01	0	0	-	0
11	0	0	-	-
10	0	0	-	-

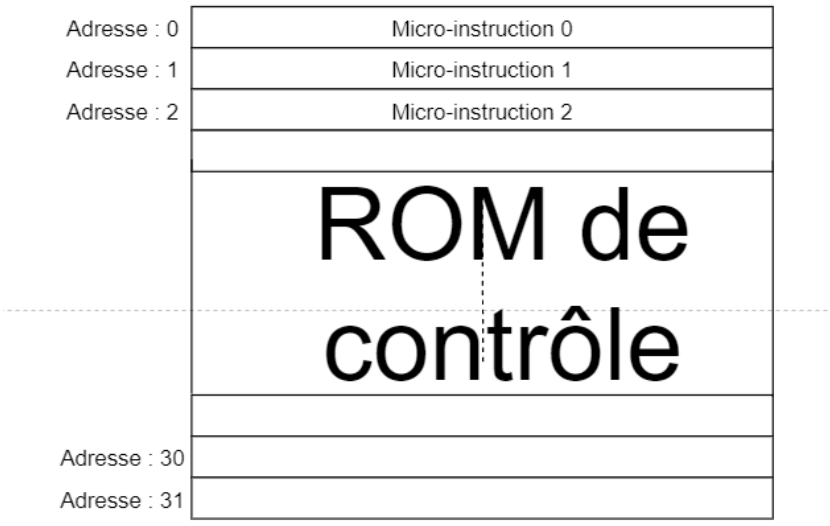
$A_3(Op_3,Op_2,Op_1,Op_0) = Op_2 \cdot \overline{Op_1} \cdot \overline{Op_0}$

Op_1Op_0 \ Op_3Op_2	00	01	11	10
00	0	0	-	0
01	0	1	-	1
11	0	1	-	-
10	0	1	-	-

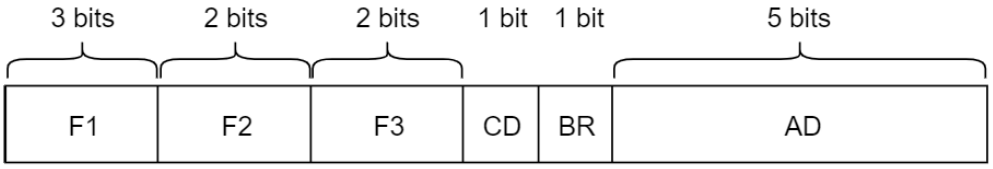
$A_2(Op_3,Op_2,Op_1,Op_0) = (Op_3+Op_2)\cdot(Op_1+Op_0)$



2. L'UCC se constitue à l'intérieur de ce qu'on appel un séquenceur, le rôle d'un séquenceur est d'exécuter une séquence de micro-instructions (appelées micro-code), sauvegarder sur ce qu'on appel une ROM de contrôle, elle détient les séquences de micro-code pour chaque type d'instruction, on peut voir une représentation de cette ROM sur diagramme en bas.



Une instruction est généralement subdivisée en petites parties appelées les micro-instructions, chacune d'elle fait progresser le flux d'information de l'instruction à travers son chemin sur le datapath jusqu'à sa complétion. Pour notre processeur la micro-instruction se compose de 6 champs, visibles sur la figure en bas. Les 3 premiers champs sont les fonctions de commandes F1, F2, F3 destinés à commander les composants du datapath, et 2 signaux BR (Branchement) et CD (Condition) utilisés pour le contrôle d'exécution du séquenceur, et un champ AD qui contient une adresse de branchement dans la ROM de contrôle.



La description des champs d'une micro-instruction est illustrée sur les tables en bas :
La table F1 :

Code			Fonction	Description
0	0	0	nop	aucune opération
0	0	1	$ALU_x \leftarrow IR[3,2]$	commande l'UAL de sortir r2 dans X
0	1	0	$ALU_y \leftarrow IR[1,0]$	commande l'UAL de sortir r3 dans Y
0	1	1	$ALU_y \leftarrow IR[5,4]$	commande l'UAL de sortir r1 dans Y
1	0	0	$IR[5,4] \leftarrow ALU$	commande l'RF d'écrire le résultat de l'UAL dans r1
1	0	1	$IR[5,4] \leftarrow DR$	commande l'RF d'écrire la donnée de DR dans r1
1	1	0	$SR \leftarrow ALU_{flags}$	commande SR d'écrire les flags de l'UAL
1	1	1	$IR \leftarrow ROM[PC]$	commande IR d'écrire l'instruction de la ROM avec l'adresse PC

La table F2 :

Code		Fonction	Description
0	0	nop	aucune opération
0	1	$AR \leftarrow IR[3,0]$	commande AR d'écrire l'immédiate (Imm)
1	0	$RAM[AR] \leftarrow DR$	commande la RAM avec l'adresse AR d'écrire la donnée de DR
1	1	$ALU_{op} \leftarrow IR[8,6]$	commande l'UAL avec le code d'opération dans IR

La table F3 :

Code		Fonction	Description
0	0	nop	aucune opération
0	1	$DR \leftarrow RAM[AR]$	commande DR d'écrire la donnée de la RAM avec l'adresse AR
1	0	$DR \leftarrow IR[5,4]$	commande DR d'écrire de registre r1 de RF
1	1	PC++	commande le PC à s'incrémenter

Remarque : les registres r1, r2 et r3 dans les tables désignent les champs de 2 bits des registres dans le format d'instruction.

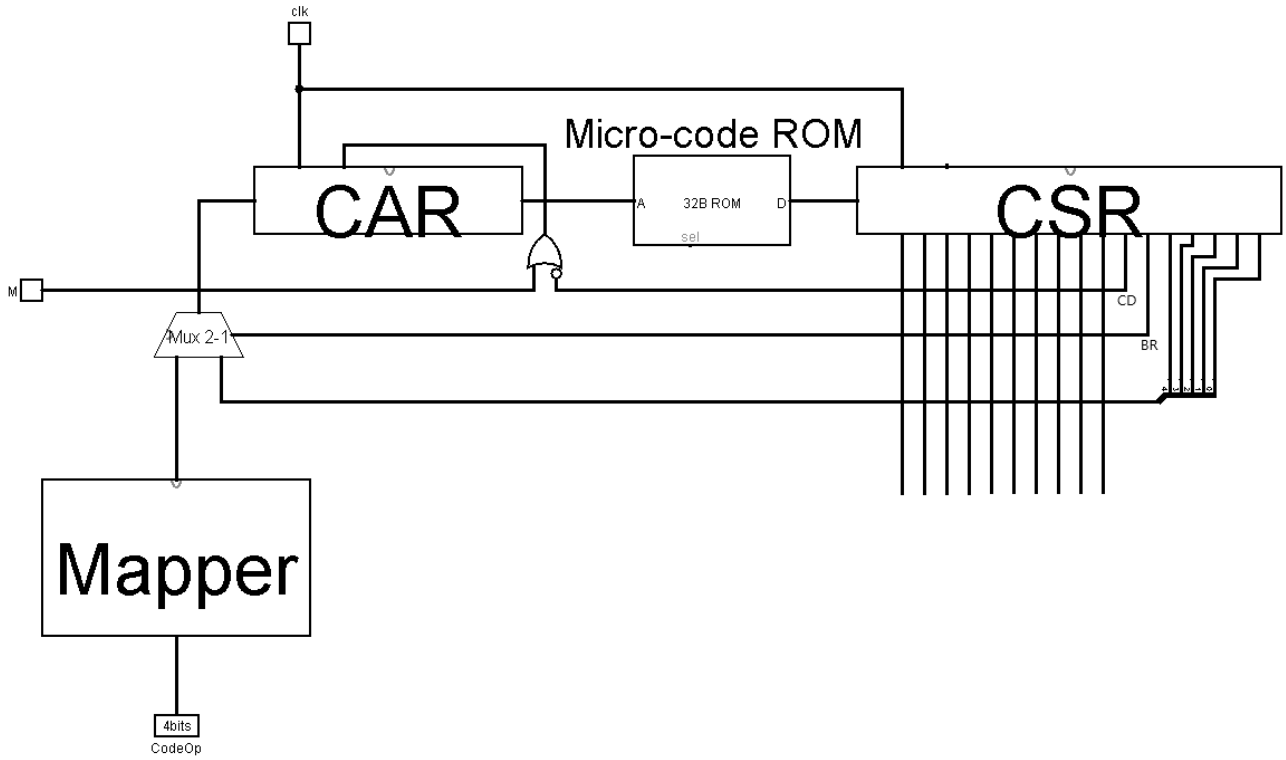
Une représentation interne pour chaque instruction dans la ROM est comme suite :

Instruction	Micro-instructions	F1	F2	F3	CD	BR	AD
Fetch	IR ← ROM[PC]	111	00	00	0	0	11001
	PC++	000	00	11	0	0	11010
	if M==1 CAR ← @Direct else CAR++	000	00	00	1	0	11100
	CAR ← Map	000	00	00	0	1	00000
Direct	AR ← IR[3,0] , CAR ← @Fetch+3	000	01	00	0	0	11011
ALU1	ALU_x ← IR[3,2]	001	00	00	0	0	00001
	ALU_y ← IR[1,0] , ALU_{op} ← IR[8,6]	010	11	00	0	0	00010
	IR[5,4] ← ALU	100	00	00	0	0	00011
	SR ← ALU_{flags} , CAR ← @Fetch	110	00	00	0	0	11000
ALU2	ALU_x ← IR[3,2]	001	00	00	0	0	00101
	ALU_{op} ← IR[8,6]	000	11	00	0	0	00110
	IR[5,4] ← ALU	100	00	00	0	0	00111
	CAR ← @Fetch	000	00	00	0	0	11000
CMP	ALU_x ← IR[3,2]	001	00	00	0	0	01001
	ALU_y ← IR[5,4] , ALU_{op} ← IR[8,6]	011	11	00	0	0	01010
	SR ← ALU_{flags}	110	00	00	0	0	01011
	CAR ← @Fetch	000	00	00	0	0	11000
LOAD	DR ← RAM[AR]	000	00	01	0	0	10001
	IR[5,4] ← DR	101	00	00	0	0	10010
	CAR ← @Fetch	000	00	00	0	0	11000
STR	DR ← IR[5,4]	000	00	10	0	0	10101
	RAM[AR] ← DR	000	10	00	0	0	10110
	CAR ← @Fetch	000	00	00	0	0	11000

Remarque 1: Le fonctionnement de l'UCC dans le TD est hypothétique (n'est pas réel) utilisé que pour des fin pédagogique, l'implémentation réel d'un datapath qui peut suivre l'UCC en question est complexe de surcroît. Dans le TP en va implémenter un UCC différent plus pratique.

Remarque 2: L'implémentation de la comparaison est aussi un petit peu différente dans le TP que celle du TD.

Le séquenceur comme sur le schéma en bas se compose principalement de 2 registres CAR (Control Address Register) de 5 bits et CSR (Control Signal Register) de 14 bits (même taille qu'une micro-instruction) et d'une ROM de commande de 32x14bits. Le registre CAR pointe sur la micro-instruction en cours d'exécution dans la ROM, le CSR quant à lui sauvegarde le code de la micro-instruction pour faire passer la liste des signaux de commande au datapath. On peut aussi voir sur le schéma le mappeur, il permet de décoder l'OpCode de l'instruction et de pointer directement sur l'adresse du début de la séquence de micro-code d'une l'instruction dans la ROM. 3 opérations de séquencement du prochain micro-code sont supportées par notre séquenceur, il peut chercher le micro-code dans l'adresse suivante CAR++, ou dans l'adresse du champ AD (les 5 premier bits dans CSR) ça faire un branchement, ou directement du mappeur. Ces 3 choix sont contrôlés par les 2 bits CD et BR dans CSR plus l'entrée M, qui vont influencer le multiplexeur et l'entrée Write de CAR pour exercer le séquencement adéquat.



Remarque : Il est important de faire noter qu'il ne faut pas faire la confusion en par exemple R1 en majuscule qui est le deuxième registre dans RF et r1 en minuscule qui est le champ de 2 bit dans le format d'instruction (le champ IR[5,4]).

Exercice 03 :

A. On un programme avec 1700 instructions, avec pour chaque instruction un temps d'exécution différent, la table en bas résume le temps d'exécution pour chaque type instruction et sa proportion en pourcentage dans le programme :

Type d'instruction	Temps d'exécution (en cycle d'horloge)	Pourcentage dans le programme (%)	Nombre d'instructions dans le programme
load	5	25	25/100 x 1700 = 425
store	5	10	10/100 x 1700 = 170
branche	3	11	11/100 x 1700 = 187
jump	3	2	2/100 x 1700 = 34
autres	5	52	52/100 x 1700 = 884

1. Le calcul du temps d'exécution pour un processeur de 5 GHz :

on a :

1 cycle d'horloge = 1/5 GHz = 1/5 000 000 000 Hz = $2 \cdot 10^{-10}$ seconde = 0,2 nanoseconde

à partir du nombre d'instructions dans le tableau plus haut et le temps d'exécution de chaque instruction, on peut calculer le temps total comme suite :

temps d'exécution = $(425x5 + 170x5 + 187x3 + 34x3 + 884x5) \times 2 \cdot 10^{-10}$
= 1611,6 nanoseconde
= 1,6116 microseconde

2. Le calcul du temps d'exécution pour un processeur de 5 GHz en utilisant le CPI (Clock cycle Per Instruction), qui représente réellement une moyenne de temps pour toutes les instructions :

on a CPI = 4 donc on aura :

temps d'exécution = nombre d'instructions x CPI x temps du cycle = $1700 \times 4 \times 2 \cdot 10^{-10}$
= $1700 \times 4 \times 2 \cdot 10^{-10}$
= 1360 nanoseconde
= 1,360 microseconde