Examen - Durée 3 heures

Exercice 1 - Testeur de réflexes – 8 points

50 minutes

On souhaite réaliser un dispositif permettant de tester les réflexes, utilisables après une soirée arrosée par exemple. Ce testeur est réalisé comme un automate de Moore. Outre le signal d'horloge H, le circuit possède trois entrées : *go*, *stop*, *reset* et cinq sorties : *On*, *T(richeur)*, *E(xcellent)*, *B(on)*, *F(aible)*.

Le principe consiste à « mesurer » le temps écoulé entre l'activation successive des signaux *go* et *stop*. L'unité de mesure est le temps de cycle de l'horloge *H*. Le comportement de ce circuit peut être décrit comme tel :

- On suppose que le circuit a été initialisé grâce au signal *reset* dans un état où toutes les sorties valent 0. Le signal reset est synchrone, c'est-à-dire que sa valeur n'est prise en compte qu'au moment du front de l'horloge H.
- La sortie *On* passe à 1 lorsqu'une personne (le passager d'une voiture par exemple) active le signal *go*.
- La sortie T passe à 1 si la personne active le signal stop avant ou en même temps qu'elle active le signal go.
- Si le signal stop est effectivement activé après le signal go,
  - E passe à 1, si stop est activé un cycle après go ;
  - B passe à 1, si stop est activé deux cycles après go ;
  - F passe à 1, si stop est activé plus de deux cycles après go.

On notera que F ne passe à 1 qu'à partir du moment où l'utilisateur appuie sur le bouton stop. Il faut activer le signal reset pour remettre à 0 les signaux de sortie et démarrer un nouveau test, et le signal reset n'est pris en compte que si l'un des signaux E, F, B ou T a la valeur 1.

1. (1 point) Représenter graphiquement l'automate de Moore à 8 états décrivant le comportement du testeur de réflexes.
2. (2 points) Décider d'un codage des états de l'automate, et construire le tableau de Karnaugh donnant, pour chaque état de l'automate, les transitions associées à celui-ci.
3. (1 point) Construire le tableau de Karnaugh donnant, pour chaque état, la valeur des 5 signaux de sortie.
4. (2 points) En supposant que l'on utilise une bascule D par bit codant l'état de l'automate, déduire, pour chacune des bascules, une expression booléenne donnant sa valeur au temps t en fonction de sa valeur au temps $t-1$ et des signaux d'entrée.
5. (2 points) Donner les expressions booléennes des 5 signaux de sortie.

Exercice 2 - MIPS – 5 points

30 minutes

Le but de cet exercice est d'écrire, en langage d'assemblage MIPS (dont les instructions ont été vues en TD), un code réalisant la fonction UPPERCASE remplaçant les lettres minuscules d'une chaîne de caractères par les lettres majuscules correspondantes. Attention : seules les lettres minuscules doivent être affectées. La fonction n'aura donc aucun impact sur les lettres déjà en majuscules, les chiffres, ou les caractères spéciaux. Par exemple, UPPERCASE("J'aime l'ASR1") = "J'AIME L'ASR1".

Pour écrire ce code, on prendra en compte les informations suivantes :

- les lettres minuscules ont un code ASCII décimal compris entre 97 ("a") et 122 ("z") ;
- si la minuscule d'une lettre a pour code ASCII décimal x, sa majuscule a pour code $x - 22$;
- le code ASCII signifiant la fin d'une chaîne ("\0") de caractères est 0.

1. (0.5 points) Donner un pseudo-code réalisant la fonction UPPERCASE.
2. (0.5 points) Décider d'une convention pour les registres MIPS à utiliser.
3. (4 points) Convertir le pseudo-code donné en 1. en un code en langage d'assemblage MIPS utilisant la convention définie en 2. Par simplicité, on considèrera que la chaîne de caractères à transformer est définie "en dur" dans le segment data `MyTXT` comme suit :

```
.data
  MyTXT: .asciiz "J'aime l'ASR1\0"
```

Exercice 3 - Pipeline – 7 points

40 minutes

Attention : lire l'intégralité de l'exercice avant de commencer. Les réponses se feront sur l'annexe 1 !

Soit le programme ci-dessous exprimé dans un language de haut-niveau et compilé en MIPS :

```
do {
  if ( VECTA[i] >= 0 ) {
    VECTB[i] = VECTA[i];
  } else {
    VECTB[i] = 0;
  {
  i++;
} while (i != N);
```

Pipeline basique

Considérons l'exécution d'une seule itération sur un processeur MIPS avec un pipeline 5 étages n'ayant **aucune optimisation**, et tel que, dans **50% des cas**, la condition (VECTA[i] >= 0) soit vraie.

Quelques précisions sur le pipeline sans optimisation :

- La mémoire est physiquement segmentée en instructions et données.

- Dans le cas d'un branchement ou saut, l'adresse de destination est calculée à l'étage EX et le PC à l'étage ME.

Dans les deux premiers tableaux fournis en annexe :

1. (2 points) Déterminer (en rouge) les dépendances de données (RAW) et (en bleu) les dépendances de contrôle (branchement, etc.). Pour cela, entourer les étages concernés, les relier avec une flèche modélisant le sens de la dépendance, et indiquer dans les colonnes de droite le(s) registre(s) concerné(s) si dépendance de données ou *CTRL* si dépendance de contrôle.
2. (1 point) Inscrire dans la colonne de gauche le nombre de *Stall* à insérer **avant l'instruction (ou entre l'étage IF et ID)** pour résoudre la dépendance.
3. (0.5 points) Déterminer le CPI (nombre de cycles par instruction) dans le cas de THEN, ELSE et total.

Pipeline optimisé

Considérons maintenant la même exécution (une seule itération) sur un processeur MIPS avec un pipeline 5 étages ayant les optimisations suivantes :

- Les registres peuvent être lus et écrits durant le même cycle d'horloge.
- Tous les bypass nécessaires ont été ajoutés.
- L'adresse de destination et le PC sont calculés durant l'étage ID (cas d'un branchement/saut).

Dans les deux derniers tableaux fournis en annexe :

1. (2 points) Déterminer les bypass résolvant les dépendances de données (en rouge) et les dépendances de contrôle (en bleu). Pour cela, entourer les étages concernés, et les relier avec une flèche modélisant le sens du bypass.
2. (1 point) Inscrire dans la colonne de gauche le nombre de *Stall* à insérer **avant l'instruction (ou entre l'étage IF et ID)** pour résoudre la dépendance.
3. (0.5 points) Déterminer le CPI (nombre de cycles par instruction) dans le cas de THEN, ELSE et total.

Exercice 4 - Code OpenCL – 6 points

30 minutes

Les questions ci-dessous font références aux codes disponibles en Annexe 2.

1. (2 points) A quoi sert la ligne 68 ? Expliquez.
2. (2 points) Expliquez le rôle des lignes 106 à 109.
3. (2 points) Que fait ce code ?

Exercice 5 - Synchronisation – 1.5 points

5 minutes

1. (1.5 points) Citez trois méthodes de synchronisation dans les systèmes d'exploitation.

Exercice 6 - Alea jacta est – 2.5 points

5 minutes

1. (2.5 points) Expliquez ce que sont les aléas de données dans un pipeline ?

Annexe 1 (à rendre)

Pipeline basique

Nb. Stalls THEN	Instruction	1	2	3	4	5	6	7	8	9	10	11	Deps. THEN
	`DO: lw $t2,VECTA($t6)`	IF	ID	EX	ME	WB							
	`slt $t0,$t2,$0`		IF	ID	EX	ME	WB						
	`bne $t0, $0, ELSE`			IF	ID	EX	ME	WB					
	`sw $t2,VECTB($t6)`				IF	ID	EX	ME	WB				
	`j INC`					IF	ID	EX	ME	WB			
	`INC:addi $t6,$t6,4`						IF	ID	EX	ME	WB		
	`bne $t6,$t7, DO`							IF	ID	EX	ME	WB	

Nb. Stalls ELSE	Instruction	1	2	3	4	5	6	7	8	9	10	Deps. ELSE
	`DO: lw $t2,VECTA($t6)`	IF	ID	EX	ME	WB						
	`slt $t0,$t2,$0`		IF	ID	EX	ME	WB					
	`bne $t0, $0, ELSE`			IF	ID	EX	ME	WB				
	`ELSE:sw $0,VECTB($t6)`				IF	ID	EX	ME	WB			
	`INC:addi $t6,$t6,4`					IF	ID	EX	ME	WB		
	`bne $t6,$t7, DO`						IF	ID	EX	ME	WB	

$CPI_{then} =$
$CPI_{else} =$
$CPI =$

Pipeline optimisé

Nb. Stalls THEN	Instruction	1	2	3	4	5	6	7	8	9	10	11
	`DO: lw $t2,VECTA($t6)`	IF	ID	EX	ME	WB						
	`slt $t0,$t2,$0`		IF	ID	EX	ME	WB					
	`bne $t0, $0, ELSE`			IF	ID	EX	ME	WB				
	`sw $t2,VECTB($t6)`				IF	ID	EX	ME	WB			
	`j INC`					IF	ID	EX	ME	WB		
	`INC:addi $t6,$t6,4`						IF	ID	EX	ME	WB	
	`bne $t6,$t7, DO`							IF	ID	EX	ME	WB

Nb. Stalls ELSE	Instruction	1	2	3	4	5	6	7	8	9	10
	`DO: lw $t2,VECTA($t6)`	IF	ID	EX	ME	WB					
	`slt $t0,$t2,$0`		IF	ID	EX	ME	WB				
	`bne $t0, $0, ELSE`			IF	ID	EX	ME	WB			
	`ELSE:sw $0,VECTB($t6)`				IF	ID	EX	ME	WB		
	`INC:addi $t6,$t6,4`					IF	ID	EX	ME	WB	
	`bne $t6,$t7, DO`						IF	ID	EX	ME	WB

$CPI_{then} =$
$CPI_{else} =$
$CPI =$

Annexe 2

exam.c

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <CL/cl.h>

#define SIZE 512

int main(int argc, char** argv) {

        size_t srcsize;

        cl_int error;
        cl_platform_id platform;
        cl_device_id device;
        cl_uint platforms, devices;

        error=clGetPlatformIDs(2, &platform, &platforms);
        if(error != CL_SUCCESS) {
                printf("Error clGetPlatformIds\n");
                return 1;
        }

        error=clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, &devices);
        if(error != CL_SUCCESS) {
                printf("Error clGetDeviceIds\n");
                return 1;
        }

        cl_context_properties properties[]={
                CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
                0};

        cl_context context=clCreateContext(properties, 1, &device, NULL, NULL, &error);
        if(error != CL_SUCCESS) {
                printf("Error clCreateContext\n");
                return 1;
        }

        cl_command_queue cq = clCreateCommandQueue(context, device, 0, &error);

  int A[SIZE*SIZE];
  int B[SIZE*SIZE];
  int C[SIZE*SIZE];
  int C_CPU[SIZE*SIZE];
  int size = SIZE;

  int i,j;
  for(i=0;i<SIZE;i++) {
    for(j=0;j<SIZE;j++) {
      A[i*size+j] = rand()%10;
      B[i*size+j] = rand()%10;
    }
  }

        char src[8192];
        FILE *fil=fopen("fct.cl","r");
        srcsize=fread(src, sizeof src, 1, fil);
        fclose(fil);

        const char *srcptr[]={src};
        cl_program prog=clCreateProgramWithSource(context,
                1, srcptr, &srcsize, &error);
        if(error != CL_SUCCESS) {
                printf("Error clCreateProgramWithSource\n");
                return 1;
        }

        error=clBuildProgram(prog, 0, NULL, "", NULL, NULL);
        if(error != CL_SUCCESS) {
                printf("Error clBuildProgram\n");

```

```
                char build_log[2048+1];
73              clGetProgramBuildInfo(prog, device,
                                CL_PROGRAM_BUILD_LOG, 2048,
75                              build_log, NULL);
                printf("%s\n",build_log);
77              return 1;
        }
79
        cl_mem memA, memB, memC;
81
        memA=clCreateBuffer(context, CL_MEM_READ_ONLY, size*size*sizeof(int), NULL, &error);
83      if(error != CL_SUCCESS) {
                printf("Error clCreateBuffer\n");
85              return 1;
        }
87
    memB=clCreateBuffer(context, CL_MEM_READ_ONLY, size*size*sizeof(int), NULL, &error);
89      if(error != CL_SUCCESS) {
                printf("Error clCreateBuffer\n");
91              return 1;
        }
93
        memC=clCreateBuffer(context, CL_MEM_WRITE_ONLY, size*size*sizeof(int), NULL, &error);
95      if(error != CL_SUCCESS) {
                printf("Error clCreateBuffer\n");
97              return 1;
        }
99
        cl_kernel kern=clCreateKernel(prog, "fct", &error);
101     if(error != CL_SUCCESS) {
                printf("Error clCreateKernel %d\n",error);
103             return 1;
        }
105
        clSetKernelArg(kern, 0, sizeof(size), &size);//size
107     clSetKernelArg(kern, 1, sizeof(memA), &memA);//A
        clSetKernelArg(kern, 2, sizeof(memB), &memB);//B
109     clSetKernelArg(kern, 3, sizeof(memC), &memC);//C

111     error=clEnqueueWriteBuffer(cq, memA, CL_FALSE, 0, size*size*sizeof(int), A, 0, NULL, NULL);
        if(error != CL_SUCCESS) {
113             printf("Error clEnqueueWriteBuffer %d\n",error);
                return 1;
115     }

117  error=clEnqueueWriteBuffer(cq, memB, CL_FALSE, 0, size*size*sizeof(int), B, 0, NULL, NULL);
        if(error != CL_SUCCESS) {
119             printf("Error clEnqueueWriteBuffer %d\n",error);
                return 1;
121     }

123  size_t *localWorkSize = NULL;
    size_t globalWorkSize[] = {size, size};
125
        error=clEnqueueNDRangeKernel(cq, kern, 2, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);
127     if(error != CL_SUCCESS) {
                printf("Error clEnqueueNDRangeKernel %d\n", error);
129             return 1;
        }
131
        error=clEnqueueReadBuffer(cq, memC, CL_FALSE, 0, size*size*sizeof(int), C, 0, NULL, NULL);
133     if(error != CL_SUCCESS) {
                printf("Error clEnqueueReadBuffer %d\n",error);
135             return 1;
        }
137
        error=clFinish(cq);
139     if(error != CL_SUCCESS) {
                printf("Error clFinish %d\n",error);
141             return 1;
        }
143
        return 0;
145 }
```

fct.cl

```
1
__kernel void fct
3     (const int size,
       __global const int* A,
5      __global const int* B,
       __global int* C)
7 {
      const uint i = get_global_id(0);
9     const uint j = get_global_id(1);
      uint k;
11
      C[i*size+j] = 0;
13    for(k=0;k<size;k++) {
        C[i*size+j] += A[i*size+k] * B[k*size+j];
15    }
}
```