INF107

Examen final
Éléments de correction

2023–2024

Part 1 (6 points / 25 minutes)

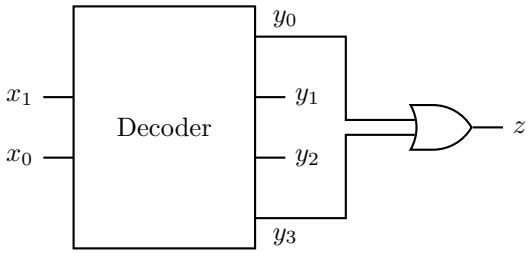
Combinatorial Logics

Question 1 (1 point)

$$y_0 = \overline{x_1} \cdot \overline{x_0}$$
$$y_1 = x_1 \cdot \overline{x_0}$$
$$y_2 = \overline{x_1} \cdot x_0$$
$$y_3 = x_1 \cdot x_0$$

Question 2 (1 point)

x_1	x_0	z
0	0	1
0	1	0
1	0	0
1	1	1



RISC-V Processor

Question 3 (2 points)

Fetch : L'adresse de l'instruction courante (sortie du bloc PC) est envoyée à la mémoire d'instructions (`Imem`).

Decode : La mémoire d'instructions envoie l'instruction à exécuter au bloc `Decode` qui génère les différents signaux de contrôle et notamment, le numéro du premier registre de source (`x1`), le numéro du registre de destination (`x2`), l'immédiat contenu dans l'instruction (`4`). Le banc de registres (`Register file`) renvoie la valeur du registre de source.

Execute : L'ALU calcule l'adresse de la donnée à lire en mémoire en additionnant le contenu du registre de source 1 et l'immédiat. Cette adresse est envoyée à la mémoire de données (`DMem`).

Write-back : La valeur lue en mémoire est stockée par le banc de registres dans le registre de destination.

Instruction suivante : La valeur de PC est incrémentée de 4.

Question 4 (1 point)

Les instructions *load* et *store* ne peuvent plus fonctionner car elles nécessitent deux accès mémoire pendant leur exécution : un accès pour récupérer l'instruction, et un autre pour réaliser l'accès mémoire demandé (écriture pour un *store* et lecture pour un *load*).

Question 5 (1 point)

En conservant les contraintes indiquées dans le sujet, il est nécessaire de diviser l'exécution d'une instruction en au moins deux cycles d'horloges : un cycle pour récupérer l'instruction et un autre cycle pour effectuer l'accès à la mémoire de données. La répartition des autres étapes (décodage, calcul par l'ALU, etc.) sur ces deux cycles est arbitraire, bien qu'un équilibrage entre les deux cycles soit souhaitable pour optimiser la fréquence d'horloge.

Admettons que le premier cycle permet juste de récupérer l'instruction et que le second cycle réalise tout le reste de l'exécution d'une instruction. Dans ce cas, il est nécessaire d'ajouter un registre stockant l'instruction reçue de la mémoire (reçue lors du premier cycle et utilisé lors du second), entre `Imem` et `Decode`. Nous devons également ajouter une petite machine à états pour connaître l'état actuel (nous avons besoin de deux états : *fetch* et *execute*). En fonction de l'état, l'adresse présentée à la mémoire est soit `PC` (dans l'état *fetch* pour récupérer l'instruction) ou la sortie de l'ALU (dans l'état *execute* pour gérer les *load* et *store*). Le `PC` ne doit être incrémenté que sur l'un des deux cycles, le banc de registre ne doit être mis à jour qu'au cycle *execute*, de même pour l'écriture en mémoire. Le registre contenant l'instruction ne doit être mis à jour que dans le cycle *fetch*.

Part 2 (6 points / 25 minutes)

```
typedef struct {
    /* Member fields omitted for brevity */
} star_t;
```

Question 6 (1 point)

```
struct {
    star_t *base;
    unsigned int size;
    unsigned int capacity;
} vector;
typedef struct vector vector_t;
```

Il est également possible d'utiliser une notation plus concise :

```
typedef struct {
    star_t *base;
    unsigned int size;
    unsigned int capacity;
} vector_t;
```

Question 7 (1 point)

```
void init_vector(vector_t *vec) {
    vec->base = NULL;
    vec->size = 0;
    vec->capacity = 0;
}
```

Notes : Le sujet indique qu'aucune mémoire n'est allouée pour les données de ce vecteur. La seule valeur sensible pour le vecteur `base` est donc `NULL` (tout autre valeur serait indistinguable de l'adresse d'une zone valide allouée). Le champ `size` vaut 0 (aucun élément n'est stocké dans le vecteur) et `capacity` vaut 0 car aucun espace n'est alloué (il est donc possible de stocker au maximum 0 éléments).

La structure représentant le vecteur est passée à la fonction par adresse (via un pointeur). Pour initialiser ses différents champs, il est donc nécessaire de déréférencer ce pointeur pour accéder aux champs. Cela peut se faire explicitement, exemple : `(*vec).base` (les parenthèses sont nécessaires à cause des priorités respectives des opérateurs `*` (déréférencement) et `.` (accès à un champ)), ou, plus élégamment, via l'opérateur `->`, exemple : `vec->base`.

Question 8 (1.5 points)

```
star_t *get_element(vector_t *vec, unsigned int element_idx) {
    if (element_idx >= vec->size)
        return NULL;

    return &vec->base[element_idx];
}
```

Note : Le test sur la ligne 2 permet de vérifier que l'indice de l'élément demandé ne dépasse pas le nombre d'éléments réellement stockés dans le vecteur. Les éléments du vecteur sont stockés les uns à la suite des autres à partir de l'adresse `vec->base`. L'élément d'indice `element_idx` est donc à l'adresse `vec->base + element_idx * sizeof(start_t)` (opération arithmétique). Il y a deux moyens d'effectuer ce calcul et de renvoyer cette adresse sous la forme d'un pointeur :

- Utiliser la notation tableau : l'élément concerné est `vec->base[element_idx]` et son adresse est `&vec->base[element_idx]`.
- Utiliser l'arithmétique des pointeurs : `vec->base + element_idx`

Question 9 (2.5 points)

```
void ensure_capacity(vector_t *vec, unsigned int new_capacity)
    if (vec->capacity < new_capacity) {
        vec->base = realloc(vec->base, new_capacity * sizeof(star_t));
        if (!vec->base) {
            perror("realloc");
            exit(EXIT_FAILURE);
        }
        vec->capacity = new_capacity;
    }
}
```

Note : le vecteur doit pouvoir stocker au moins `new_capacity` éléments. Si c'était déjà le cas (`capacity >= new_capacity`), il n'y a rien à faire. Dans le cas contraire, il faut augmenter la capacité de stockage du vecteur. Cela se fait avec la fonction `realloc`. Cette fonction prend l'adresse d'une zone mémoire déjà allouée (`vec->base`) et la nouvelle taille désirée pour cette zone. On doit pouvoir stocker `new_capacity` éléments, donc la taille doit être `new_capacity * sizeof(star_t)`. La nouvelle zone conserve les données de l'ancienne. L'adresse de la nouvelle zone est renvoyée par `realloc`. En cas d'échec, `realloc` renvoie `NULL`.

Part 3 (8 points / 40 minutes)

Question 10 (3 points)

Solution acceptée :

```
1  /* #include-s omitted for brevity */
2  int main() {
3    char cmd_line[64];
4    while (1) {
5      printf("$␣");
6      fflush(stdout);
7
8      /* Read the command line.
9       * We assume it is short and contains no argument (single word)
10      */
11     if (read(0, cmd_line, 64) <= 1)
12       continue;
13
14     int rv;
15
16     rv = fork();
17
18     switch (rv) {
```

```
19      case 0: /* Provide code to handle such a situation */
20
21        execvp(cmd_line, NULL);
22        exit(EXIT_FAILURE);
23
24      case -1: /* Provide code to handle such a situation */
25
26        exit(EXIT_FAILURE);
27
28      default: /* Provide code to handle such a situation */
29
30        wait(NULL);
31      }
32    }
33  }
```

Notes : une fois la commande lue, la première étape est de créer un nouveau processus. Cela se fait à l'aide de la fonction `fork` (ligne 16). Cette fonction crée un nouveau processus. Le nouveau processus (processus enfant) est une copie du processus d'origine (processus parent), y compris l'instruction en cours d'exécution. Le processus parent et le processus enfant sortent donc tous les deux de la fonction `fork`.

Dans le processus parent, `fork` renvoie l'identifiant du processus enfant, strictement positif. Ce processus parent exécute donc le cas `default` (ligne 28) du `switch`. Ce processus parent doit attendre que le processus enfant (qui exécute la commande) se termine. On appelle donc la fonction `wait` qui va bloquer jusqu'à ce que le processus enfant se termine. La fonction `wait` permet d'obtenir des informations sur le processus enfant qui s'est terminé (identifiant et cause de l'arrêt), ce qui n'est pas nécessaire ici.

Dans le processus enfant, `fork` renvoie 0. Ce processus enfant exécute donc le cas `0` (ligne 19). Il appelle `execvp` qui permet de remplacer le code du processus par le contenu d'un fichier exécutable (ici la commande à exécuter). En cas de succès `execvp` ne retourne pas (tout le code est remplacé par un autre). En cas d'échec `execvp` retourne et on arrête le processus enfant par un appel à `exit` (ligne 22). L'appel à `execvp` a été simplifié ici : la fonction doit prendre un tableau en deuxième argument, dont le premier élément est le nom de l'exécutable, les suivants les éventuels arguments supplémentaires et doit se terminer par la valeur `NULL`.

Si `fork` échoue, il retourne `-1`, ce qui permet de déclencher l'appel à `exit` ligne 26.

Question 11 (3 points)

```
1  initialization() {
2      int reader_count = 0;
3      semaphore mtx;  // protect reader_count
4      semaphore wrt;  // exclusive access (for writer)
5      init_semaphore(mtx, 1);
6      init_semaphore(wrt, 1);
7  }
8
9  writer(file) {
10     wait(wrt);
11     write(file, "...␣something␣...");
12     signal(wrt);
13 }
14
15 reader(file) {
16     wait(mtx);
17     reader_count++;
18     if (reader_count == 1)
19         wait(wrt);
20     signal(mtx);
21
22     read(file);
23
24     wait(mtx);
25     reader_count--;
26     if (reader_count == 0)
```

```
27          signal(wrt);
28      signal(mtx);
29  }
```

Note : il s'agit ici du problème classique lecteur/écrivain simplifié car nous supposons qu'il n'y a pas un flux ininterrompu de lecteurs, il n'y a donc pas besoin de gérer la famine des écrivains.

La première étape est de garantir l'accès exclusif : soit un écrivain, soit un (ou plusieurs) lecteur, mais pas les deux en même temps. Ce accès exclusif est assuré par le sémaphore `wrt`, initialisé à 1.

Si un écrivain arrive, il appelle `wait` sur `wrt` (ligne 10).

— Si le compteur du sémaphore est à 1, il passe à 0, la fonction `wait` retourne et l'écrivain peut ainsi faire son écriture (ligne 11). Une fois l'écriture terminée, l'écrivain appelle `signal` sur `wrt` (ligne 12), ce qui permet de débloquer éventuellement un lecteur ou un (premier) écrivain qui serait bloqué sur l'opération `wait` sur ce sémaphore.

— Si le compteur du sémaphore est inférieur ou égal à 0 (un écrivain ou un lecteur est déjà présent), la fonction `wait` bloque et l'écrivain ne peut pas faire l'écriture pour le moment. Il sera débloqué par un appel à `signal` sur ce sémaphore.

Côté lecteur, le comportement sera différent si on est le premier lecteur ou non.

Si on est le premier lecteur, il est nécessaire de garantir l'exclusion avec les écrivains. Le premier lecteur appelle donc `wait` (ligne 19) sur le sémaphore `wrt` (le même qu'utilisé précédemment). Si un écrivain est présent, `wait` va bloquer. Lorsque le dernier lecteur a terminé, le sémaphore peut être rendu (pour débloquer un éventuel écrivain arrivé entre temps) grâce à `signal` (ligne 27).

Pour savoir si un lecteur est le premier/dernier ou non, il est nécessaire d'avoir une variable partagée comptant le nombre de lecteurs (`reader_count`). Cette variable est incrémentée au début d'un lecteur (ligne 17) et décrémentée à la fin (ligne 25). Cependant, comme toute variable partagée, l'accès à celle-ci doit être protégé. Cela peut se faire à l'aide d'un *mutex* ou d'un sémaphore utilisé en verrou. Dans la solution ici, c'est cette dernière option qui est retenue. Le sémaphore verrou est `mtx`. On appelle `wait` dessus (lignes 16 et 24) avant toute manipulation de la variable et `signal` (lignes 20 et 28) après.

Question 12 (2 points)

	Waiting times (ms)				
Algorithm	P1	P2	P3	P4	P5
FCFS	0	5	8	9	17
Priority	16	0	3	4	12
RR, q=4	12	4	7	13	12

FCFS = Premier arrivé, premier servi (sans notion de priorité) :

— $t = 0$ ms, P1 débute son exécution, P2, P3, P4 et P5 attendent

— $t = 5$ ms, P1 se termine en n'ayant jamais attendu, P2 débute son exécution, P3, P4 et P5 attendent

— $t = 8$ ms, P2 se termine en ayant attendu au total 5 ms (entre $t = 0$ et $t = 5$), P3 débute son exécution, P4 et P5 attendent

— $t = 9$ ms, P3 se termine en ayant attendu au total 8 ms (entre $t = 0$ et $t = 8$), P4 débute son exécution, P5 attend

— $t = 17$ ms, P4 se termine en ayant attendu au total 9 ms (entre $t = 0$ et $t = 9$), P5 débute son exécution

— $t = 21$ ms, P5 se termine en ayant attendu au total 17 ms (entre $t = 0$ et $t = 17$)

Priorité :

— $t = 0$ ms, P2 débute son exécution, P1, P3, P4 et P5 attendent

— $t = 3$ ms, P2 se termine en n'ayant jamais attendu, P3 débute son exécution, P1, P4 et P5 attendent

— $t = 4$ ms, P3 se termine en ayant attendu au total 3 ms (entre $t = 0$ et $t = 3$), P4 débute son exécution, P1 et P5 attendent

— $t = 12$ ms, P4 se termine en ayant attendu au total 4 ms (entre $t = 0$ et $t = 4$), P5 débute son exécution, P1 attend

— $t = 16$ ms, P5 se termine en ayant attendu au total 12 ms (entre $t = 0$ et $t = 12$), P1 débute son exécution

— $t = 21$ ms, P1 se termine en ayant attendu au total 16 ms (entre $t = 0$ et $t = 16$)

Tourniquet (RR), sans notion de priorité, avec quantum de temps de 4 ms :

— $t = 0$ ms, P1 débute son exécution, P2, P3, P4 et P5 attendent

— $t = 4$ ms, P1 est mis en attente sans avoir fini (il lui reste 1 ms), P2 débute son exécution, P1, P3, P4 et P5 attendent

— $t = 7$ ms, P2 se termine en ayant attendu au total 4 ms (entre $t = 0$ et $t = 4$), P3 débute son exécution, P1, P4 et P5 attendent

— $t = 8$ ms, P3 se termine en ayant attendu au total 7 ms (entre $t = 0$ et $t = 7$), P4 débute son exécution, P1 et P5 attendent

— $t = 12$ ms, P4 est mis en attente sans avoir fini (il lui reste 4 ms), P5 débute son exécution, P1 et P4 attendent
— $t = 16$ ms, P5 se termine en ayant attendu au total 12 ms (entre $t = 0$ et $t = 12$), P1 reprend son exécution, P4 attend
— $t = 17$ ms, P1 se termine en ayant attendu au total 12 ms (entre $t = 4$ et $t = 16$), P4 reprend son exécution
— $t = 21$ ms, P4 se termine en ayant attendu au total 13 ms (entre $t = 0$ et $t = 8$ et entre $t = 12$ et $t = 17$)