3TC31 (ex. INF107)

Examen final
Éléments de correction

2025–2026

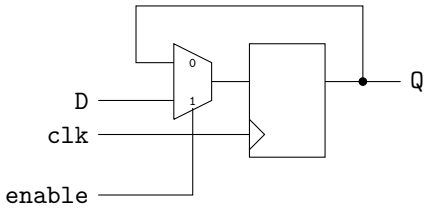
Partie 1 (3 points / 15 minutes)

Question 1 (0.5 points)

La valeur, en décimal, du nombre représenté en complément à 2 sur 5 bits par la valeur 10000 est -16.

En effet, on se souvient que la valeur d'un nombre représenté en complément à 2 sur n bits $(a_{n-1}a_{n-2}\dots a_1a_0)$ est $-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$ soit ici $-1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = -16$.

Question 2 (1 point)



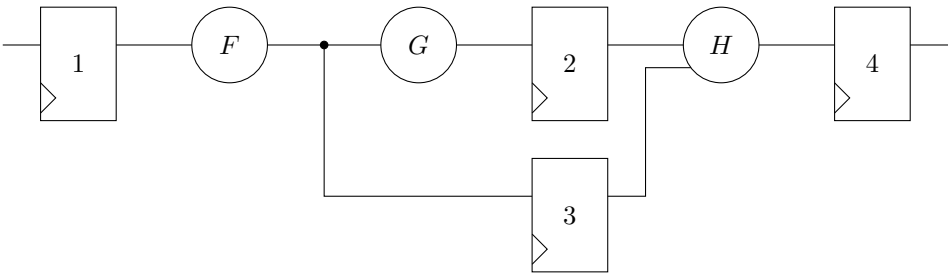
Lorsque le signal `enable` vaut 1, le signal `D` est présenté directement en entrée de la bascule D. Donc si un front montant se produit sur l'entrée d'horloge `clk`, la sortie `Q` de la bascule prendra la valeur de `D`.

Si `enable` vaut 0, la valeur actuelle de la sortie `Q` est présentée en entrée de la bascule. Donc si un front montant se produit sur l'entrée d'horloge `clk`, la sortie `Q` prendra la valeur de l'entrée de la bascule, c'est-à-dire la valeur actuelle de `Q`. La sortie `Q` conservera donc sa valeur.

Le reste du temps (en l'absence de front montant sur l'horloge), la sortie `Q` de la bascule conserve sa valeur (fonctionnement normal d'une bascule).

Note : Une autre réponse revient très souvent. Elle consiste à ajouter une porte logique ET (AND) dont les entrées sont l'horloge `clk` et le signal `enable` et d'utiliser la sortie de cette porte comme horloge d'une bascule classique. On s'interdit explicitement cette construction dans le cadre de ce cours (l'utilisation de logique combinatoire sur le chemin de l'horloge entraîne un décalage des fronts montants utilisés pour déclencher les bascules au sein du circuit et des risques de faux fronts causés par le temps de propagation au sein de la logique).

Question 3 (1 point)



On rappelle que toutes les bascules fonctionnent sur la même horloge. Au moment d'un front montant d'horloge, chaque bascule va échantillonner son entrée et recopier cette valeur sur sa sortie, avec un petit temps de retard : le temps de propagation t_{co}. Cette valeur va ensuite se propager dans la logique combinatoire en aval de la bascule, ce qui va prendre du temps (le temps de propagation dans la logique combinatoire rencontrée). Il faut enfin arriver en entrée de la bascule suivante au plus tard t_{su} avant le front montant d'horloge suivant pour que cette bascule échantillonne correctement le résultat du calcul.

Sur le schéma, si on considère les bascules 1 et 2 : lors d'un premier front montant, la bascule 1 recopie son entrée sur sa sortie. Donc après t_{co}, la sortie de la bascule 1 (donc l'entrée de la fonction combinatoire F) est stable. Il faut ensuite t_F pour que la sortie de F (et donc l'entrée de G) soit stable. Il faut ensuite t_G pour que la sortie de G soit stable. Cette sortie, qui est l'entrée de la bascule 2 doit être stable au plus tard t_{su} avant le front d'horloge suivant pour que la bascule 2 échantillonne un résultat correct. Donc en considérant uniquement les bascules 1 et 2, la période d'horloge T_{clk} doit être supérieure ou égale à $t_{co}+t_F+t_G+t_{su} = 1+4+4+1 = 10$ ns.

Ce raisonnement doit être étendu à tout le circuit en considérant tous les chemins partant d'une sortie d'une bascule à l'entrée de la bascule suivante, c'est-à-dire dans notre cas :

— de la bascule 1 à la bascule 2 : $T_{clk} \geq t_{co} + t_F + t_G + t_{su}$ (10 ns)
— de la bascule 1 à la bascule 3 : $T_{clk} \geq t_{co} + t_F + t_{su}$ (6 ns)
— de la bascule 2 à la bascule 4 : $T_{clk} \geq t_{co} + t_H + t_{su}$ (5 ns)
— de la bascule 3 à la bascule 4 : $T_{clk} \geq t_{co} + t_H + t_{su}$ (5 ns)

Le chemin critique (le chemin combinatoire le plus long entre la sortie d'une bascule et l'entrée de la bascule suivante), qui est donc celui qui contraint le plus la période d'horloge (et donc sa fréquence) est le chemin entre la bascule 1 et la 2.

La période minimale d'horloge est donc 10 ns, et donc la fréquence maximale de celle-ci est de 100 MHz.

Question 4 (0.5 points)

L'instruction assembleur RISC-V `addi x1, x2, 42` additionne le contenu du registre `x2` avec la valeur immédiate 42 et stocke le résultat dans le registre `x1`.

La valeur 42 est un immédiat. Elle est stockée directement dans certains des 32 bits de l'instruction (les bits 31 à 20 pour être précis).

Partie 2 (3 points / 15 minutes)

Question 5 (0.5 points)

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[]) {
5    int * p;
6    *p = 42;
7    printf("*p␣=␣%d\n", *p);
8    return EXIT_SUCCESS;
9  }
```

Le programme va planter avec une erreur de segmentation. C'était le piège de l'examen. En effet, ligne 5, on définit un pointeur. Le compilateur va donc allouer de la mémoire pour stocker la valeur de ce pointeur (dans le cas présent sur la pile puisqu'il s'agit d'une variable locale). Cependant, le pointeur n'est pas initialisé (aucune valeur explicite n'est affectée au pointeur). Sa valeur est donc non définie (en pratique, comme c'est une variable locale, il vaudra la valeur précédente stockée dans la pile) et peut valoir n'importe quoi.

Or dans la ligne suivante, on déréférence le pointeur, c'est-à-dire que l'on prend sa valeur (qui est n'importe quoi) et on l'utilise comme adresse, et on stocke la valeur 42 à cette adresse en mémoire. On va donc écrire la valeur 42 un peu n'importe où en mémoire, et donc très certainement à une adresse non allouée ou à laquelle on n'a pas le droit d'écrire, ce qui va déclencher une erreur de segmentation.

Moralité : on ne déréférence jamais un pointeur dans lequel on n'a pas mis une valeur sensée (adresse d'une variable allouée, résultat d'un `malloc`...).

Question 6 (1 point)

Le programme affiche : `a=1 b=2`. En effet, le passage des arguments aux fonctions en C se fait par copie. La fonction `swap` reçoit donc une copie des valeurs passées par `main`. Elle va donc intervertir la valeur de ces copies, copies qui vont disparaître à la fin de la fonction `swap`. La fonction `main` ne voit donc pas de modifications.

Pour palier ce problème, on va modifier la fonction `swap` pour qu'elle reçoive non pas la valeur de ses arguments mais les adresses auxquelles ils sont stockés.

```
#include <stdlib.h>
#include <stdio.h>
```

```c
void swap(int *a, int *b) {
  int c = *a;
  *a = *b;
  *b = *c;
}

int main(int argc, char *argv[]) {
  int a = 1;
  int b = 2;
  swap(&a, &b);
  printf("a=%d␣b=%d\n", a, b);
  return EXIT_SUCCESS;
}
```

Question 7 (1.5 points)

```c
void display(struct node *head) {
  if (head == NULL) {
    printf("Liste␣vide\n");
    return;
  }

  struct node *p = head;
  while (*p != NULL) {
    printf("%d\n", p->value);
    p = p->next;
  }
}
```

Remarque : Une liste chaînée est représentée par un pointeur vers son premier maillon (donc ici un type `struct node *`). Si ce pointeur vaut `NULL`, cela indique que la liste ne contient même pas de premier maillon, donc qu'elle est vide.

Part 3 (14 points / 60 minutes)

Exercice 1 : Entrées/Sorties bas niveau

Question 8 (4 points)

```c
typedef struct {
  int id;
  char name[32];
  double value;
} object_t;

void read_object_at(int fd, size_t index, object_t *obj) {
  off_t offset = index * sizeof(object_t);

  if (lseek(fd, offset, SEEK_SET) == -1)
    exit(EXIT_FAILURE);

  ssize_t n = read(fd, obj, sizeof(object_t));
  if (n != sizeof(object_t))
    exit(EXIT_FAILURE);
}

void write_object_at(int fd, size_t index, object_t *obj) {
  off_t offset = index * sizeof(object_t);

```

```
21    if (lseek(fd, offset, SEEK_SET) == -1)
22      exit(EXIT_FAILURE);
23
24    ssize_t n = write(fd, obj, sizeof(object_t));
25    if (n != sizeof(object_t))
26      exit(EXIT_FAILURE);
27  }
```

Les objets sont stockés les uns à la suite des autres dans le fichier (que l'on suppose déjà ouvert et représenté par le descripteur de fichier `fd`). Chaque objet est stocké dans le fichier directement tel qu'il est représenté en mémoire. Chaque objet prend donc `sizeof(object_t)` octets dans le fichier.

Quand l'on souhaite lire ou écrire l'objet numéro `index`, il faut donc dans un premier temps se positionner correctement dans le fichier. Les objets étant numérotés à partir de zéro, l'objet `index` commence à partir de l'octet `index * sizeof(object_t)` (lignes 8 et 19). On appelle ensuite `lseek` pour changer la position courante dans le fichier vers l'octet en question (il s'agit donc d'une position calculée à partir du début du fichier d'où la valeur `SEEK_SET` en troisième argument de l'appel à `lseek`). En cas d'erreur (par exemple si la position est au delà de la fin du fichier), `lseek` retourne la valeur -1. Dans ce cas, on appelle `exit` pour arrêter immédiatement l'exécution (lignes 11 et 22).

Une fois bien positionné, on appelle `read` (ou `write`) pour lire (ou écrire) `sizeof(object_t)` octets, depuis la positon courante, vers (depuis) l'adresse fournie par le pointeur `obj` en mémoire. `read` retourne le nombre d'octets lus (qui peut être inférieur au nombre d'octets demandés) ou -1 en cas d'erreur. Le seul cas où tout s'est bien passé est quand `read` renvoie `sizeof(object_t)` d'où le test ligne 14 et 25.

Exercice 2 : Processus et parallélisme

Question 9 (2 points)

Voir ci-dessous

Question 10 (1 point)

```
1  void process_partition(int fd, off_t first, off_t last) {
2      if (last - first <= 2)
3          return;
4
5      off_t pivot = partition_file(fd, start, end);
6
7      // Question 9
8      pid_t left = fork();
9      if (left == 0) {
10         process_partition(fd, first, pivot);
11         exit(EXIT_SUCCESS);
12     }
13
14     pid_t right = fork();
15     if (right == 0) {
16         process_partition(fd, pivot + 1, last);
17         exit(EXIT_SUCCESS);
18     }
19
20     // Question 10
21     wait(NULL);
22     wait(NULL);
23 }
```

Ligne 8, on créé un premier processus enfant grâce à `fork`. On a maintenant deux processus qui sortent de l'appel à `fork` (le processus originel, processus parent, et le nouveau processus, le processus enfant). Dans le processus parent, `fork` renvoie une valeur non nulle correspondant à l'identifiant du processus enfant. Dans le processus enfant, `fork` renvoie 0. Donc seul le processus enfant entre dans le corps du `if` lignes 10 et 11.

Le processus parent, et uniquement lui, ne rentre pas dans le `if` et continue ligne 14. À ce moment là, un second processus enfant est créé par le processus parent. Le second processus enfant exécute le corps du `if` lignes 16 et 17.

Le processus parent quant à lui va ensuite exécuter deux fois `wait`. Chaque appel à `wait` va bloquer jusqu'à ce que l'un des deux processus enfants se termine (événement qui peut s'être produit avant l'appel à la fonction).

Exercice 3 : Ordonnancement

Question 11 (3 points)

Temps d'attente :

	Temps d'attente (ms)				
Algorithme	P1	P2	P3	P4	P5
FCFS	0	5	8	9	17
SJF	8	1	0	13	4
RR, q=3	10	3	6	13	15

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 $t = 17$)
Le plus court d'abord (sans notion de priorité) :
— $t = 0$ ms, P3 débute son exécution, P1, P2, P4 et P5 attendent
— $t = 1$ ms, P3 se termine en n'ayant jamais attendu, P2 débute son exécution, P1, P4 et P5 attendent
— $t = 4$ ms, P2 se termine en ayant attendu au total 1 ms (entre $t = 0$ et $t = 1$), P5 débute son exécution, P1 et P4 attendent
— $t = 8$ ms, P5 se termine en ayant attendu au total 4 ms (entre $t = 0$ et $t = 4$), P1 débute son exécution, P4 attend
— $t = 13$ ms, P1 se termine en ayant attendu au total 8 ms (entre $t = 0$ et $t = 8$), P4 débute son exécution
— $t = 21$ ms, P4 se termine en ayant attendu au total 13 ms (entre $t = 0$ et $t = 13$)
Tourniquet, sans notion de priorité, avec quantum de temps de 3 ms :
— $t = 0$ ms, P1 débute son exécution, P2, P3, P4 et P5 attendent
— $t = 3$ ms, P1 est mis en attente sans avoir fini (il lui reste 2 ms), P2 débute son exécution, P1, P3, P4 et P5 attendent
— $t = 6$ ms, P2 se termine en ayant attendu au total 3 ms (entre $t = 0$ et $t = 3$), P3 débute son exécution, P1, P4 et P5 attendent
— $t = 7$ ms, P3 se termine en ayant attendu au total 6 ms (entre $t = 0$ et $t = 6$), P4 débute son exécution, P1 et P5 attendent
— $t = 10$ ms, P4 est mis en attente sans avoir fini (il lui reste 5 ms), P5 débute son exécution, P1 et P4 attendent
— $t = 13$ ms, P5 est mis en attente sans avoir fini (il lui reste 1 ms), P1 reprend son exécution, P4 et P5 attendent
— $t = 15$ ms, P1 se termine en ayant attendu au total 10 ms (entre $t = 3$ et $t = 13$), P4 reprend son exécution, P5 attend
— $t = 18$ ms, P4 est mis en attente sans avoir fini (il lui reste 2 ms), P5 reprend son exécution, P4 attend
— $t = 19$ ms, P5 se termine en ayant attendu au total 15 ms (entre $t = 0$ et $t = 10$ et entre $t = 13$ et $t = 18$), P4 reprend son exécution
— $t = 21$ ms, P4 se termine en ayant attendu au total 13 ms (entre $t = 0$ et $t = 7$, entre $t = 10$ et $t = 15$ et entre $t = 18$ et $t = 19$)

Exercice 4 : Synchronisation

Question 12 (2 points)

```
init() {
    init_sem(A_has_arrived, 0);
    init_sem(B_has_arrived, 0);
}
```

```
barrier_A() {
    signal(A_has_arrived);
    wait(B_has_arrived);
}

barrier_B() {
    signal(B_has_arrived);
    wait(A_has_arrived);
}
```

Il s'agit ici directement d'une extension de l'exemple vu en cours (une action doit être faite par un premier processus avant qu'un second puisse faire une autre action). On utilise deux sémaphores, initialisés à 0 (donc pour l'instant, un appel à `wait` sur n'importe lequel des deux va bloquer). Dès qu'un des deux processus arrive à la barrière de synchronisation, il va faire un appel à `signal` sur le sémaphore indiquant qu'il est arrivé, puis faire un `wait` sur le sémaphore indiquant que l'autre processus est arrivé. Si l'autre processus n'est pas encore arrivé, l'appel à `signal` fait passer le compteur du sémaphore passe à 1 (et donc l'appel ultérieur à `wait` sur ce sémaphore ne bloquera pas). Si l'autre processus était déjà arrivé (il est donc actuellement bloqué sur le `wait`), l'appel à `signal` va le libérer et il pourra sortir du `wait` et franchir la barrière de synchronisation.

Question 13 (2 points)

```
#define TOTAL_THREADS N;
init() {
  unsigned int count = 0;  // Nombre de threads déjà arrivés
  init_mtx(mutex);  // Protège la variable partagée count
  init_sem(all_arrived, 0);
}

barrier() {
  lock(mutex);
  count++;
  if (count == N) {  // Si je suis le dernier processus à arriver, je libère tout le monde
    for i in 0 <= i < N {  // Appel à signal N fois
      signal(all_arrived);
    }
  }
  unlock(mutex);
  wait(all_arrived); // Chaque thread attend (le dernier s'est libéré lui même)
}
```

L'attente des processus va se faire sur un sémaphore (`all_arrived`) initialisé à 0 pour que dès le début, un appel à `wait` dessus bloque. Le dernier processus arrivé (le $N^{\text{ième}}$) doit faire une action particulière. Il faut donc savoir combien de processus sont déjà en attente pour savoir si on est le dernier. Il n'est pas possible de consulter le compteur interne d'un sémaphore (en tout cas dans l'abstraction donnée). Il est donc nécessaire d'avoir une variable globale pour compter le nombre de processus en attente. Or, dès que l'on a une variable globale qui sera manipulée par les différents threads, nous devons la protéger par un verrou, d'où le mutex.

Donc quand un processus arrive au point de sychronisation, il verrouille le mutex. Il incrémente ensuite la variable indiquant le nombre de processus en attente. Si moins de N processus sont en attente, il déverrouille le mutex puis fait un `wait` sur le sémaphore qui va le bloquer. Si N processus sont maintenant en attente, il fait N appel à `signal`, qui vont débloquer les $N-1$ processus actuellement bloqués dans `wait`, et le compteur du sémaphore arrive à 1 (et donc le dernier `wait` ne bloquera pas le dernier processus).