3TC31 (ex. INF107)

Examen final
Éléments de correction

2024–2025

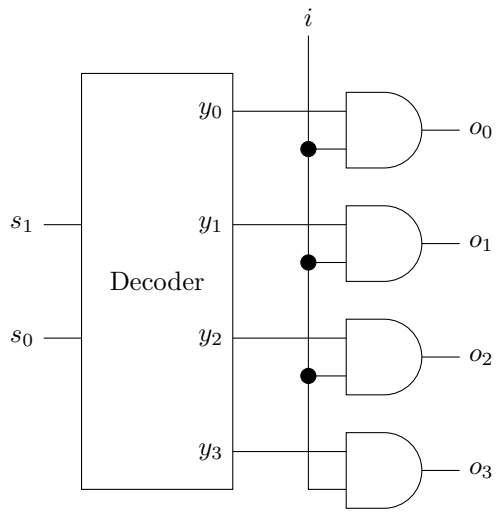
Part 1 (6 points / 25 minutes)

Combinatorial Logics

Question 1 (1 point)

i	s_1	s_0	o_0	o_1	o_2	o_3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Question 2 (1 point)



RISC-V Processor

Question 3 (2 points)

pc	instr	rs1	rs2	rd	op	ALUsrc	imm	op1	op2	res / Addr	WData	RData	write	load	store	wrdata
0	0x401101b3	2	1	3	−	1	x	8	4	4	x	x	1	0	0	4
4	0x0081a203	3	x	4	+	0	8	4	8	12	x	80	1	1	0	80

Question 4 (2 points)

```
1  foo:
2          ...          // do something interesting
3          jr ra        // jump back to return address
4
5  main:
6          addi sp, sp, -8
7          sw s0, 8(sp)
8          sw s1, 4(sp)
9
10         jal ra, foo // call function foo
11
12         lw s1, 4(sp)
13         lw s0, 8(sp)
14         addi sp, sp, 8
```

Explications : avant l'appel à la fonction `foo` (`jal` ligne 10), il faut sauvegarder le contenu des registres `s0` et `s1` dans la pile. Le registre `sp` contient l'adresse du sommet de la pile (plus précisément la case libre à utiliser pour stocker la prochaine donnée dans la pile). La première étape est de réserver 8 octets dans la pile (pour y stocker `s0` et `s1` chacun sur 4 octets). Comme la pile croît vers les adresses basses, cette opération se fait en décrémentant `sp` de 8 (ligne 6). On réalise cette opération en premier pour s'assurer qu'à tout moment, `sp` pointe bien vers une case non utilisée dans la pile (pour assurer un bon fonctionnement en cas d'interruption par exemple). Les deux cases libres sont maintenant aux adresses `sp` + 4 et `sp` + 8. On stocke donc (lignes 7 et 8) le contenu des registres `s0` et `s1` à ces deux adresses.

Une fois la fonction `foo` terminée, il faut relire le contenu de la pile pour restaurer l'état des registres `s0` et `s1` (lignes 12 et 13), puis de libérer ces deux cases en incrémentant `sp` de 8 (ligne 14).

Part 2 (8 points / 40 minutes)

Question 5 (1 point)

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

int main(int argc, char *argv[]) {
  int a = 42;
  int *b = &a;            // L1
  *b = 43;                // L2
  printf("a␣=␣%d\n", a); // L3
  return EXIT_SUCCESS;
}
```

- `b` est un pointeur, de type `int *`
- `&` retourne l'adresse de son argument, ici l'adresse de la variable `a`
- La ligne L2 stocke la valeur 43 à l'adresse contenu dans le pointeur `b` (c'est-à-dire dans la variable `a`)
- Le programme affiche `a = 43`

Question 6 (1 point)

```c
enum op_e { PLUS, MINUS, MULT, DIV, NUM };
typedef enum op_e op_t;
```

Question 7 (1 point)

```c
op_t char_to_op(char c)
{
  switch (c)
  {
    case '+': return PLUS;
    case '-': return MINUS;
    case '*': return MULT;
```

```
    case '/': return DIV;
    default:
      fprintf(stderr, "Invalid operator: %c.\n", c);
      exit(EXIT_FAILURE);
  }
}
```

Question 8 (1 point)

```
struct expr_s
{
  op_t op;           // operator
  int num;           // value when operator if N, 0 otherwise
  struct expr_s *l;  // left expression or NULL
  struct expr_s *r;  // right expression or NULL
};

typedef struct expr_s expr_t;
```

Question 9 (2 points)

```
int eval_expr(const expr_t *e)
{
  if (e->op == NUM)
    return e->num;
  else
  {
    int l = eval_expr(e->l);
    int r = eval_expr(e->r);

    switch (e->op)
    {
      case PLUS:  return l + r;
      case MINUS: return l - r;
      case MULT:  return l * r;
      case DIV:   return l / r;

      default:
        exit(EXIT_FAILURE);
    }
  }
}
```

Question 10 (2 points)

```
parse_result_t parse_expr(const char *str)
{
  parse_result_t result;

  // Process first character of remaining string to be parsed.
  switch (*str)
  {
    // An operator was detected ...
    case '+': case '-':
    case '*': case '/':
    {
      // Continue parsing the left and right child sub-expression using recursive calls:
      // TODO (3): provide the arguments to the recursive calls
```

```
      parse_result_t l = parse_expr(str + 1);
      parse_result_t r = parse_expr(l.str);

      // Return value: str points to the character after the right operand
      // TODO (4): assign a value to result.str
      result.str = r.str;

      // Return value: e represents the parsed expression
      result.e = malloc(sizeof(expr_t));
      result.e->op = char_to_op(*str);
      result.e->n = 0;
      result.e->l = l.e;
      result.e->r = r.e;

      return result;
    }

    // The beginning of an integer number was detected ...
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    {
      // Convert character to integer number
      int n = *str - '0';

      // Return value: str points to the character after the digit
      result.str = str + 1;

      // Return value: e represents the parsed expression

      // TODO (1): allocate a new expression on the heap and store it in result.e
      result.e = malloc(sizeof(expr_t));
      if (result.e == NULL)
      {
        perror("Allocate NUM");
        exit(EXIT_FAILURE);
      }

      // TODO (2): assign struct members of result.e (using n)
      result.e->op = NUM;
      result.e->num = n;
      result.e->l = NULL;
      result.e->r = NULL;

      return result;
    }

    // Unexpected end of string: display an error
    case '\0':
      fprintf(stderr, "Unexpected end of expression\n");
      exit(EXIT_FAILURE);

    default:
      fprintf(stderr, "Invalid character in expression: %c.\n", *str);
      exit(EXIT_FAILURE);
  }
}
```

Part 3 (6 points / 25 minutes)

Question 11 (3 points)

```
1  #include <assert.h>
2  #include <fcntl.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  int swap_bytes(int fd, int offset);
7
8  int main(int argc, char *argv[]) {
9      if (argc != 2) return;
10     int fd = open(argv[1], O_RDWR);
11     for (int offset = 1; swap_bytes(fd, offset); offset++);
12     close(fd);
13 }
14
15 int swap_bytes(int fd, int offset) {
16     int rv;
17     char left, right;
18
19     rv = lseek(fd, offset - 1, SEEK_SET);
20     assert(0 <= rv);
21     rv = read(fd, &left, sizeof(left));
22     assert(rv == 1);
23
24     rv = lseek(fd, -offset, SEEK_END);
25     assert(0 <= rv);
26
27     if (rv < offset) return 0;
28
29     rv = read(fd, &right, sizeof(right));
30     assert(rv == 1);
31
32     rv = lseek(fd, offset - 1, SEEK_SET);
33     assert(0 <= rv);
34
35     rv = write(fd, &right, sizeof(right));
36     assert(rv == 1);
37
38     rv = lseek(fd, -offset, SEEK_END);
39     assert(0 <= rv);
40
41     rv = write(fd, &left, sizeof(left));
42     assert(rv == 1);
43
44     return 1;
45 }
```

Quelques remarques. On notera ici l'utilisation de `assert`, une macro qui prend en paramètre une expression et qui termine le programme en affichant un message d'erreur si cette expression est évaluée à une valeur fausse (égale à 0).

On notera également que `offset`, tel qu'il est utilisé dans la boucle `for` ligne 11, commence par la valeur 1 (et non 0). D'où l'utilisation de `offset - 1` ligne 19 lors de l'appel à `lseek`.

En cas de succès, `lseek` renvoie la nouvelle position dans le fichier, exprimée à partir du début du fichier. C'est ainsi que l'on peut élégamment savoir si `offset` est plus grand que la moitié du fichier ligne 27.

Enfin, on remarquera que le corps de la boucle `for` ligne 11 est vide. Ce n'est pas une erreur, en effet, tout est fait dans la condition d'itération, à la fois l'appel à `swap_bytes` et, en fonction de sa valeur de retour, le test pour savoir si on a dépassé la moitié du fichier. On peut réécrire plus lisiblement `swap_bytes(fs, offset) != 0`.

Question 12 (1 point)

```
initialization() {
    init_sem(c1_finished, 0);
}

T1() {
    C1;
    signal(c1_finished);
}

T2() {
    wait(c1_finished);
    C2;
}
```

Il s'agit exactement de l'exemple contenu dans le cours.

Si le thread `T2` arrive à `wait` avant que le processus `T1` ait fini `C1` (donc avant qu'il n'ait appelé `signal`, le compteur associé au sémaphore vaut 0 et donc la fonction `wait` va bloquer le thread `T2` jusqu'à ce que le thread `T1` ait appelé `signal`.

Si le thread `T1` fini `C1` et appelle `signal` avant que `T2` n'ait appelé `wait`, le compteur du sémaphore passe à 1. Donc quand `T2` fera le `wait`, la fonction ne bloquera pas et il pourra poursuivre pour faire `C2`.

Question 13 (2 points)

```
initialization() {
    init_sem(c1_finished, 0);
    init_sem(c2_finished, 0);
    init_sem(c3_finished, 0);
}

T1() {
    C1;
    signal(c1_finished);
    wait(c2_finished);
    C3;
    signal(c3_finished);
    print("Bye.");
}

T2() {
    wait(c1_finished);
    C2;
    signal(c2_finished);
    wait(c3_finished);
    print("Bye.");
}
```

Il s'agit d'une simple extension de la question précédente.