3TC31 (ex. INF107)

Exam

2024–2025

Name: .....

Instructions

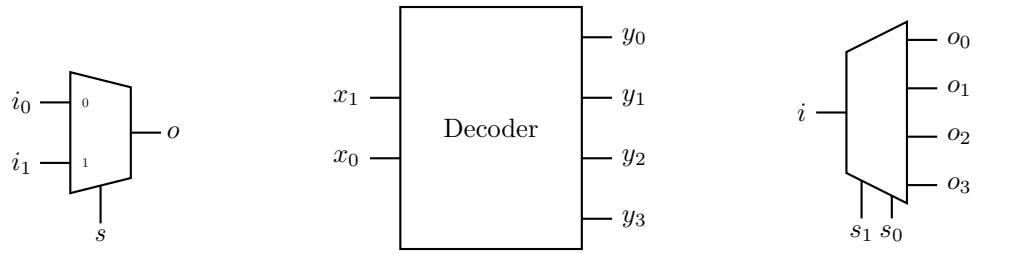
- Duration: 90 minutes
- No documents are allowed
- Calculators, mobile phones, and computers are prohibited
- You can answer in French or English
- You will find the signature of some useful C functions at the end of the exam sheet
- There are 3 independent parts:
  - Part 1 (6 points): Questions 1 and 2 on Combinatorial logics, Questions 3 and 4 on RISC-V processor
  - Part 2 (8 points): Questions 5 to 10 on the C programming language
  - Part 3 (6 points): Question 11 on Files and Questions 12 and 13 on Synchronization

Part 1 (6 points / 25 minutes)

Combinatorial Logics

In the lecture, a basic combinatorial circuit has been introduced, called the *multiplexer*. The 2-to-1 multiplexer selects among the input signals i_0 and i_1. Depending on the value of the select input s, the output o will take on the value of i_0 or i_1.

Another basic circuit is the *decoder*. In general, it has N inputs $x_0, \ldots, x_{N-1}$ and 2^N outputs $y_0, \ldots, y_{2^N-1}$. Its functionality can be described as follows: There is always exactly one active output (whose value is 1), where its index corresponds to the unsigned integer represented by the input vector $(x_{n-1} \ldots x_0)$. Below in Figures 1a and 1b are the circuit symbols of the described circuits.



(a) 2-to-1 multiplexer (b) 2-to-4 decoder (c) 1-to-4 demultiplexer

Figure 1: Circuit symbols

We now introduce a new circuit, the *demultiplexer*. It can be thought of as the inverse of the multiplexer. Depending on a select input s, the value of input i will be redirected to one of the outputs o_j, where j corresponds to the value of the select inputs interpreted as an unsigned integer value. All other outputs o_k with $k \neq j$ are equal to 0. Figure 1c shows the circuit symbol for a 1-to-4 multiplexer.

Question 1 (1 point)

Complete the truth table for the outputs of the 1-to-4 demultiplexer (*note: you can answer directly on this sheet but do not forget to write your name on the first page*):

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

Question 2 (1 point)

We can actually realise the 1-to-4 demultiplexer using a 2-to-4 decoder and some additional logic gates. Draw the corresponding circuit diagram.

RISC-V Processor

During the lecture and the lab exercises, we have studied a simple implementation of a RISC-V processor. Figure 2 on the following page shows the data path implementing parts of the RISC-V base instruction set, including register-to-register instructions, immediate instructions, loads, and stores.

Question 3 (2 points)

We assume that the *instruction memory* contains two instructions:

Address	**Instruction**	**Machine code**
0	`sub x3, x2, x1`	`0x401101b3`
4	`lw x4, 8(x3)`	`0x0081a203`

We assume that the *data memory* contains the following 32 bit words, noted in decimal:

Address	**Data**
0	10
4	20
8	40
12	80

We assume that the *registers* contains the following data, noted in decimal:

Register	**Data**
0	0
1	4
2	8
3	12
4	16

Complete the table on the following page with the values of the signals when executing these instructions. You can use operator symbols $+$, $-$… for the ALU operation (`op`). Note that the appendix contains specifications of the used instructions. If you make a mistake, cross out the line in the table and use one of the extra lines.

Note: you can answer directly on this sheet but do not forget to write your name on the first page.

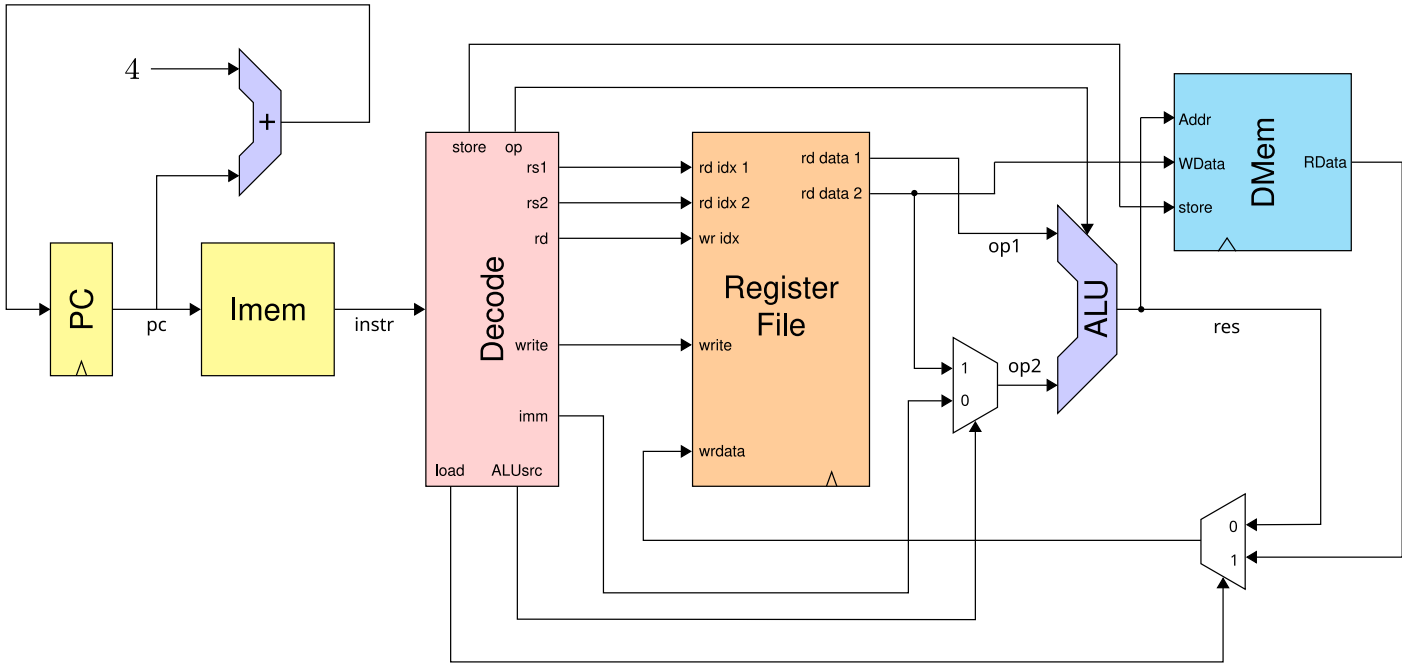


Figure 2: RISC-V data path for R-type, immediate and memory instructions

pc	instr	rs1	rs2	rd	op	ALUsrc	imm	op1	op2	res / Addr	WData	RData	write	load	store	wrdata
0	0x401101b3															
4	0x0081a203															

In the RISC-V ISA, the instruction `jal` (jump and link) is used to realise *function calls*: It stores the return address (current program counter plus 4) in the given register, and then jumps to the given address. By convention, we use the register `ra` to store the return address. One thing that we need to take care of before calling a function is *saving registers*: The called function might overwrite register values, so we need to write them to the memory and *restore* them after the function has returned. As you should know by now, we use the *stack* for this. By convention, the address of the next available memory cell on the stack is found in the register `sp` (*stack pointer*). We assume further that the stack grows from top to bottom (i.e. the stack pointer decreases on pushing and increases on popping).

Question 4 (2 points)

Complete the following assembler program in order to *save and restore* the two registers `s0` and `s1` on the stack before and after the function call, respectively. Give the instruction sequences that need to be inserted at positions (1) and (2).

```
foo:
        ...          // do something interesting
        jr ra        // jump back to return address

main:
                     // (1) push registers on the stack
        jal ra, foo  // call function foo
                     // (2) pop registers from stack
```

Part 2 (8 points / 40 minutes)

During the TP you have stored a data structure representing stars inside a linked list.
This exercise is concerned with representing and manipulating arithmetic expressions, e.g., `4 + 5`, in a *tree*. Contrary to a linked list, the binary operators of an arithmetic expression (here: `+`) have two sub-expressions (here: `4` and `5`).

Question 5 (1 point)

```
#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;
}
```

- What is the type of the variable `b` (line L1)?
- What is returned by the operator `&` (line L1)?
- What does line L2 do?
- What is printed on the standard output when this program is executed?

Question 6 (1 point)

Define an enumeration `op_e` with the following symbols: `PLUS`, `MINUS`, `MULT`, `DIV`, and `NUM`, and define a type alias `op_t` to be used instead of `enum op_e`.

Question 7 (1 point)

Implement the function `char_to_op`, which takes a single character symbol (`char`) as input.
The function should convert (return) the input characters `'+'`, `'-'`, `'*'`, and `'/'` to the respective values of `op_t` (`PLUS`, `MINUS`, `MULT`, and `DIV`).
For other characters that are not valid operators print the error message `"Invalid␣operator:␣"` to `stderr` followed by the invalid character symbol and a newline, then terminate the program using `exit` with an appropriate argument.
The function is never supposed to return `NUM`.

Question 8 (1 point)

Define a structure `expr_s`, representing a node of the arithmetic expression tree.
A node holds the following information:

- `op`:
  One of the operator symbols (`PLUS`, `MINUS`, `MULT`, or `DIV`) for binary operators or `NUM` when the expression is an integer number.
- `num`:
  A signed integer number when `op` has the value `NUM`, or `0` otherwise.
- `l` and `r`:
  Pointers to sub-expressions (left and right) when `op` represents an operator, or `NULL` otherwise.

Choose an appropriate type for each structure member and define a type alias `expr_t` to be used instead of `struct expr_s`.

Question 9 (2 points)

We suppose that we have defined 5 global variables of type `expr_t` with the names `n4`, `n5`, `n6`, `p`, and `m`, forming the expression shown in Figure 3, as follows:

```
expr_t n4 = {NUM, 4, NULL, NULL};
expr_t n5 = {NUM, 5, NULL, NULL};
expr_t n6 = {NUM, 6, NULL, NULL};
expr_t m = {MULT, 0, &n4, &n5};
expr_t p = {PLUS, 0, &n6, &m};
```

Write a recursive function `eval_expr`, with the signature: `int eval_expr(const expr_t *e)`.
This function should return the value of the expression `e`.

- When a node represents a number, return its value.
- When a node represents a binary operation, first evaluate the two sub-expressions (`l` and `r`). Then apply the respective binary operator of the C language to the two values obtained for the sub-expressions, and return the result.

You can assume that the expression passed to the function is correct. So, no error handling is required.
Example: The result of calling `eval_expr(p)` should compute 26.

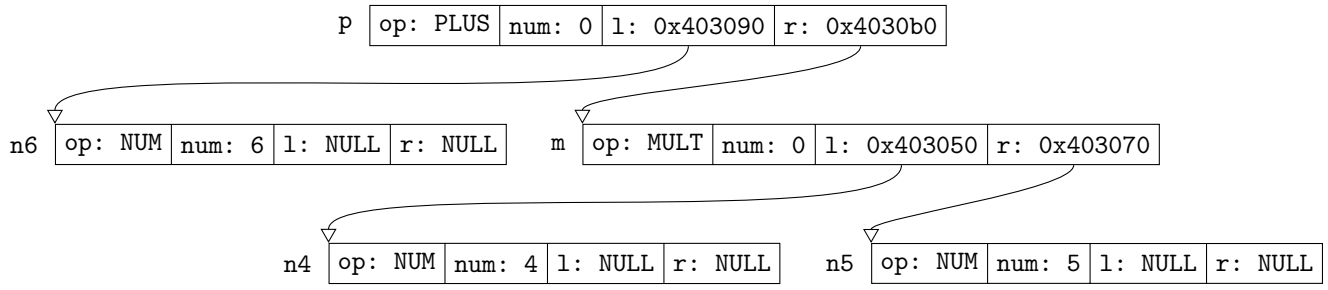


Figure 3: The expression tree represented by variable `p`.

Question 10 (2 points)

Now that the data structure is in place, we want to implement a function that is able to construct an expression tree from a string.

Complete the function `parse_expr` from below that takes a string as input and returns a special data structure `parse_result_t`.

```
typedef struct
{
  const char *str;    // Pointer to remaining characters of string to be parsed
  expr_t *e;          // Node of the expression tree, parsed so far
} parse_result_t;

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(          );
      parse_result_t r = parse_expr(          );

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

```
      // TODO (2): assign struct members of result.e (using n)




      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);
  }
}
```

The function inspects the first character of the input string, distinguishing the following cases.

- If the end of the string has been reached (`case '\0'`):
  An error message is printed to `stderr` and the program is terminated.
- If an invalid character (neither a number or operator) was encountered (`default:`):
  An error message is printed to `stderr` and the program is terminated.
- If a digit was encountered (`case '0':` … `case '9':`):
  A new object of type `expr_t` is allocated on the heap and initialized accordingly. The newly allocated object and a string pointer is returned (see `result`).
- If an operator character was encounter (`case '+':` … `case '/':`):
  Recursive calls to `parse_expr` parse the left and right sub-expressions respectively. A new object of type `expr_t` is allocated on the heap and initialized accordingly. The newly allocated object and a string pointer is returned (see `result`).

Example: Parsing the string `"+6*45"` should be parsed as $6 + (4 \times 5)$ and should yield an expression tree with the same shape as the one shown in Figure 3. In addition, Figure 4 illustrates the recursive calls to `parse_expr` while parsing this string.

TODO: Complete the code of the function shown above at the TODO markers (1) through (4).

1. Allocate a new object of type `expr_t` on the heap and assign the obtained pointer to `result.e`. If the allocation fails, print an error message using `perror` and terminate the program using `exit`.
2. Assign a value to every structure member of `result.e`, which is a pointer to an `expr_t`. Notably use the local variable `n` defined a couple of lines above.
3. Provide arguments to the recursive calls of `parse_expr`. These arguments have to be a sub-string of the original input string `str`. For the left sub-expression the argument is simply the rest of `str` without the first character. For the right sub-expression the argument is obtained from the return value of the left sub-expression. Use pointer arithmetic as needed.
4. Assign a value to `result.str`. The value has to be a sub-string of the original input string `str`, which was not yet parsed. In other words, `result.str` should point to the first character after the sub-string representing the right operand of the binary operator. See Figure 4 for an illustration.

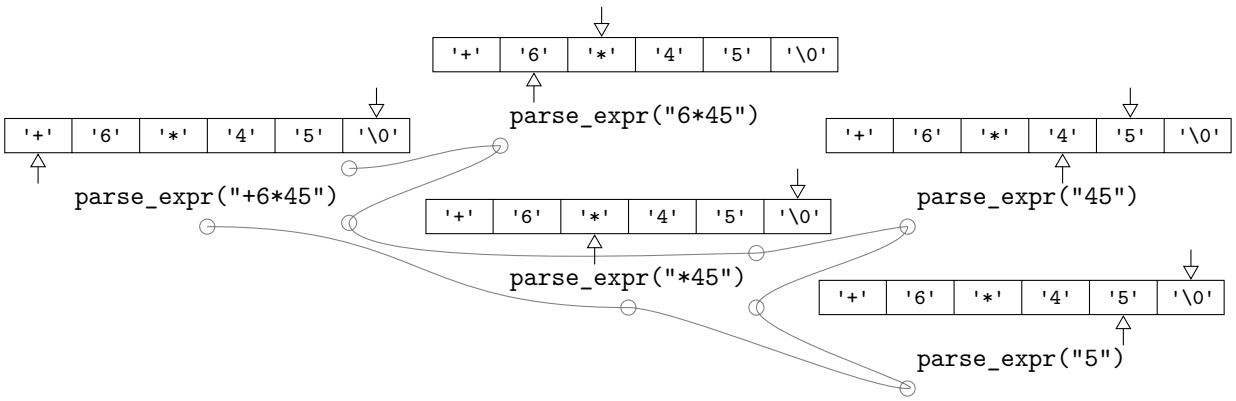


Figure 4: Recursive calls to `parse_expr` while parsing the string `"+6*45"`. The function arguments illustrate the remaining characters to parse using arrows (↑) below the boxes. The arrows (↓) above the boxes indicate the string pointer returned.

Part 3 (6 points / 25 minutes)

Question 11 (3 points)

We want to implement a program that reverses a file as the `reverse` would do in Python. The first byte is swapped with the last, the second with the next to last, and so on. A file containing "12345" becomes a file containing "54321". This exercise is very close to what was implemented in TP12 (fs-edit-file). We give an outline of the code below to start with. The `swap_bytes` function swaps two bytes in the file, the first being the byte number `offset` from the beginning of the file and the second being the byte number `offset` from the end of the file. If the parameter `offset` exceeds half the file size, the `swap_bytes` function returns 0; otherwise it returns 1. Complete the code below.

```
#include <assert.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int swap_bytes(int fd, int offset);

int main(int argc, char *argv[]) {
  int fd;
  if (argc != 2) return;

  // TODO

  for (int offset = 1; swap_bytes(fd, offset); offset++);

  // TODO

}

int swap_bytes(int fd, int offset) {
    int rv;
    char left, right;

    // TODO




    return 0;
```

```
    // TODO
```

```
    return 1;
}
```

Question 12 (1 point)

(Synchronization problem (1).) Two concurrent threads, T_1 and T_2, execute, respectively, computations C_1 and C_2 at some point during their execution. We want to make sure that computation C_2 *does not begin before* C_1 *has finished.*

Complete the following pseudo-code skeletons with the correct use of synchronization tools (semaphores and/or mutexes) to ensure that is the case.

Note: the annex contains a list of function signatures to manipulate semaphores and mutexes.

```
initialization() {
    // Write here initialization code (e.g., semaphores/mutex initialization,
    // but also global variables) shared by all participating threads.
```

```
}

T1() {
```

```
    C1;
```

```
}

T2() {
```

```
    C2;
```

```
}
```

Question 13 (2 points)

(Synchronization problem (2).) Two concurrent threads, T_1 and T_2, execute 3 computations: T_1 executes C_1 and later C_3; T_2 executes C_2. We want to make sure that the 3 computations execute in the order: C_1, C_2, C_3 where *each computation does not start before the previous has finished.* Additionally, $T2$ *should wait for* $C3$ *to complete* before terminating (so that T_1 and T_2 will terminate at around the same time).

Complete the following pseudo-code skeletons with the correct use of synchronization tools (semaphores and/or mutexes) to ensure that is the case.

Note: the annex contains a list of function signatures to manipulate semaphores and mutexes.

```
initialization() {
    // Write here initialization code (e.g., semaphores/mutex initialization,
    // but also global variables) shared by all participating threads.




}

T1() {


    C1;


    C3;


    print("Bye.");
}

T2() {


    C2;


    print("Bye.");
}
```

Annex: signatures of useful C functions

- `void *memset(void *ptr, int value, size_t num);`
  Fills the first `num` bytes of the memory area pointed to by `ptr`, returns `ptr`.
- `int memcmp(const void *ptr1, const void *ptr2, size_t num);`
  Compares `num` bytes of the memory areas `prt1` and `ptr2`; returns `0` when all the bytes are the same.
- `void *memcpy(void *dst, const void *src, size_t num);`
  Copies `num` bytes from memory area `src` to memory area `dst`; returns `dst`.
- `char *strncpy(char *dst, const char *src, size_t sz);`
  Copies up to `sz` characters from the string `src` to `dst`, stopping at the null character; returns `dst`.
- `size_t strlen(const char *s);`
  Calculates the length of the string pointed to by `s`.
- `void *malloc(size_t size);`
  Allocates `size` bytes on the heap and returns a pointer to the allocated memory.

- `void *realloc(void *ptr, size_t size);`
  Changes the size of the memory block pointed to by `ptr` to `size` bytes. If `ptr` is `NULL` behaves like `malloc`. May invalidate the pointer `ptr` and, in all cases, returns a pointer to the *reallocated* memory region. The **content** of the memory area originally pointed to by `ptr` is **preserved**.
- `void free(void *ptr);`
  Frees the memory space pointed to by `ptr`, which has to be allocated by `malloc` or `realloc` beforehand.

- `void perror(const char *s);`
  Produces a message on `stderr` describing the last error encountered during a call to a C library function.
- `void exit(int status);`
  Causes normal process termination with an exit code `status`.

- `ssize_t read(int fildes, void *buf, size_t nbyte);`
  Attempt to read `nbyte` bytes from the file associated with the file descriptor `fildes` into the buffer pointed to by `buf`; returns the number of bytes actually read, `0` when the end-of-file (EOF) was reached, or `-1` in case of an error.
- `ssize_t write(int fildes, const void *buf, size_t nbyte);`
  Attempt to write `nbyte` bytes from the buffer pointed to by `buf` to the file associated with the file descriptor `fildes`; returns the number of bytes actually written or -1 in case of an error.
- `int pipe(int fildes[2]);`
  Create a pipe and place two file descriptors into `fildes[0]` and `fildes[1]`; return `0` on success or `-1` on error.
- `int open(const char *path, int oflag, ...);`
  The file name specified by `path` is opened for reading and/or writing, as specified by the argument `oflag`; the file descriptor is returned to the calling process.
- `int close(int fd);`
  The close() call deletes the descriptor `fd` from the per-process object reference table.
- `off_t lseek(int fd, off_t offset, int whence);`
  The lseek() function repositions the offset of the file descriptor `fd` to the argument `offset`, according to the directive `whence`. Upon successful completion, lseek() returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of `-1` is returned. lseek() repositions the file pointer as follows:
  - If `whence` is SEEK_SET, the offset is set to `offset` bytes.
  - If `whence` is SEEK_CUR, the offset is set to its current location plus `offset` bytes.
  - If `whence` is SEEK_END, the offset is set to the size of the file plus `offset` bytes.

- `pid_t fork(void);`
  Create a new process by copying the current process; returns `0` in the child process; returns the process id of the child process or `-1` in case of an error in the parent process.
- `pid_t wait(int *status);`
  Blocks the current process until one of its child processes terminates; returns the process id of the child or `-1` in case of an error. Also returns the exit status of the child via the pointer `status`. If `status` is `NULL` no exit code is returned.
- `int execvp(const char *file, char *const argv[]);`
  Replace the current process image with a new process image loaded from the executable file with name `file`, passing the table `argv` as command-line arguments to the new process image; returns `-1` in case of an error.

Annex: (pseudo-code) signatures of synchronization tool functions

```
/* functions available on a semaphore sem */
init_sem(semaphore sem, unsigned int n);
wait(semaphore sem);
signal(semaphore sem);
```

```
/* functions available on a mutex mtx */
init_mtx(mutex mtx);  // initially unlocked
lock(mutex mtx);
release(mutex mtx);
```

Annex: RISC-V Instructions

add (addition)

Format `add rd, rs1, rs2`

Description Adds the register `rs2` to `rs1` and stores the result in `rd`.

addi (immediate addition)

Format `addi rd, rs1, imm`

Description Adds the register `rs1` to the sign extended immediate value `imm` and stores the result in `rd`.

sub (subtraction)

Format `sub rd, rs1, rs2`

Description Subtracts the register `rs2` from `rs1` and stores the result in `rd`.

lw (load word)

Format `lw rd, offs(rs1)`

Description Loads a 32-bit value from memory and stores it in register `rd`. The address is computed as the sum of register `rs1` and constant offset `offs`.

sw (store word)

Format `sw rs2, offs(rs1)`

Description Stores a 32-bit value from register `rs2` to the memory. The address is computed as the sum of register `rs1` and constant offset `offs`.

jal (jump and link)

Format `jal rd, offs`

Description Stores the return address (`PC` + 4) in register `rd` and jumps to the target address by adding `offs` to the current `PC`.