INF107

Exam

2023–2024

Instructions

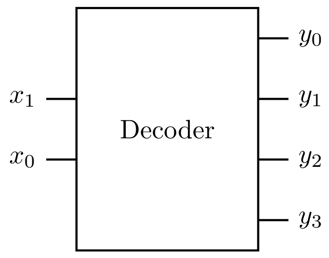
- Duration: 90 minutes
- No documents are allowed
- Calculators, mobile phones, and computers are prohibited
- You can answer in French or English
- You will find the signature of some useful C functions at the end of the exam sheet
- There are 3 independent parts:
  - Part 1: Questions 1 and 2 on Combinatorial logics, Questions 3 and 4 on RISC-V processor
  - Part 2: Questions 6 to 9 on the C programming language
  - Part 3: Question 10 on Processes, Question 11 on Synchronization and Question 12 on Scheduling

Part 1 (6 points / 25 minutes)

Combinatorial Logics

In the lecture, a basic combinatorial circuit has been introduced, called the *decoder*. In general, a decoder has n inputs (named x_0 to x_{n-1}) and 2^n outputs (named y_0 to y_{2^n-1}). Only one of its outputs is 1 and all the others are 0, where the *index* of the output being 1 corresponds to the value of the inputs, when interpreted as an unsigned number (with x_0 the least significant bit). Here is the truth table and the circuit symbol for a 2-input decoder:

x_1	x_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



Question 1 (1 point)

Give the Boolean equations for the four outputs y_0 to y_3 of the decoder.

Question 2 (1 point)

We can realise any n-input Boolean function using only an n-input decoder and one or several OR gates. Consider the 2-input Boolean function XNOR, i.e. Boolean equality. Its output is 1 if and only if the inputs are equal. Give the truth table of the XNOR function (with inputs x_0 and x_1 and output z) and a circuit implementation using a 2-input decoder and an OR gate.

RISC-V Processor

During the lecture and the lab exercises, we have studied a simple implementation of a RISC-V processor. Figure 1 shows the data path implementing parts of the RISC-V base instruction set, including register-to-register instructions, immediate instructions, loads, and stores.

We also recall the phases of the execution of an instruction: *fetch*, *decode*, *execute*, *write-back*, and *next instruction*. In our implentation, all of the above phases are executed within a single clock cycle.

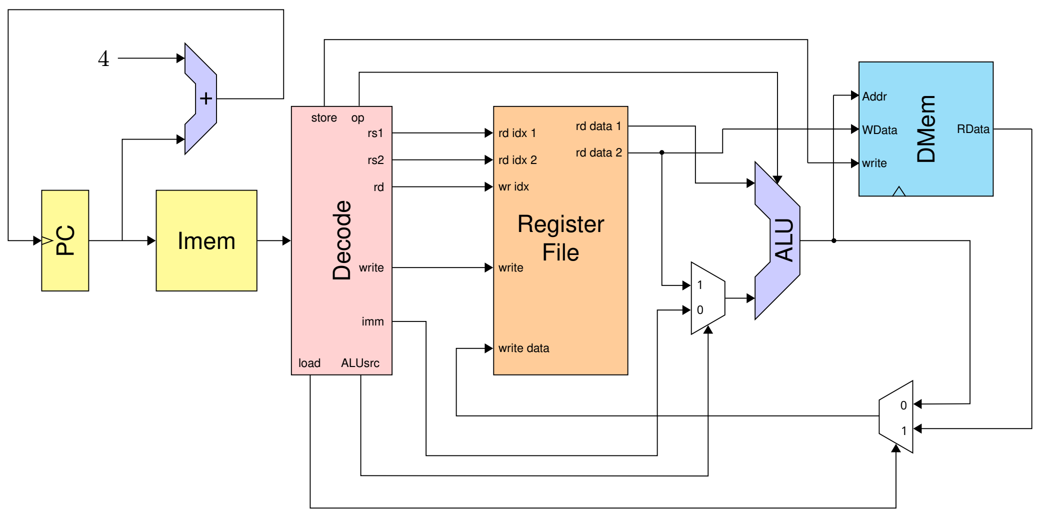


Figure 1: Data path of the RISC-V processor

Question 3 (2 points)

Consider the execution of a *load word* instruction corresponding to the assembler code `lw x2, 4(x1)`. For each phase of its execution, describe briefly what happens in the processor implementation, which circuit elements are involved, and what their input values are.

Question 4 (1 point)

The above implementation follows the *Harvard architecture*, i.e. we consider two separate memories for instructions and for data. As you have learned, code and data are usually stored in one shared memory. In the following, we will consider that the two memory interfaces Imem and Dmem are connected to a *single global memory* with the following properties:

- Read access is combinatorial, i.e. the read data is available in the same clock cycle
- Write access is synchronous, i.e. takes place at the next rising edge of the clock
- There can be only a single access (read or write) in one clock cycle

Explain why the proposed implementation no longer works with such a shared memory. Which are the instructions that cannot be executed? Give an example of an instruction and the specific phases that cause the problem.

Question 5 (1 point)

How can we change our implementation of the RISC-V base instruction set, in order to adapt it to a single shared memory? Explain briefly the changes that need to be made, without giving a concrete circuit diagram.

Part 2 (6 points / 25 minutes)

During the TP of INF107, you have stored structures representing stars inside an array and inside a linked list. In this part, you will implement functions to manipulate a **vector**, which is a data structure combining the advantages of lists and arrays:

- all the elements are stored contiguous in memory
- still, the data structure can also grow dynamically when you insert new elements.

The advantage of a vector is that accessing an element is fast. Since all the element are contiguous and we know the address of the first element, it easy to compute the address of the element to access. The disadvantage is that adding an element to a full vector may take some time as it is necessary to allocate new memory and copy the existing data to this new memory space.

The vector you will implement should store elements of type `star_t`, given below:

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

Question 6 (1 point)

Provide the definition of the structure `vector` that represents a vector and should contain the following fields:

- `base`:
  A pointer to a memory space that can store one or several elements of type `star_t`
- `size`:
  A number (`>= 0`) that stores the **number of elements** (not their sizes!) currently stored inside the vector
- `capacity`:
  A number (`>= 0`) that stores the **number of elements** (not their sizes!) that can be stored in the currently allocated memory space

Choose the most appropriate type for each structure member and define a type alias `vector_t` to be used instead of `struct vector`.

Question 7 (1 point)

Implement the function `init_vector` with the following signature:

```
void init_vector(vector_t *vec)
```

This function receives the address of a vector (`vec`) and initializes it to an empty vector. This empty vector should have no memory allocated yet to store elements. Make sure to assign a meaningful value to all members of the structure.

Question 8 (1.5 points)

Implement the function `get_element` with the following signature:

```
star_t *get_element(vector_t *vec, unsigned int element_idx)
```

This function should return the address of the element stored inside the vector `vec`, at the position indicated by `element_idx`. If no such element exists, `NULL` should be returned.

Question 9 (2.5 points)

Implement the function `ensure_capacity` with the following signature:

```
void ensure_capacity(vector_t *vec, unsigned int new_capacity)
```

This function should make sure that the memory space allocated for the vector elements is sufficiently large to hold at least `new_capacity` elements. If the current capacity is sufficiently large the vector should not be modified. Otherwise, you should *reallocate* the memory space with the new larger capacity.

If the memory reallocation fails, you should report an error message using `perror` and terminate the program with an appropriate exit code.

The content of the vector, i.e., its elements, and its size should not be modified.

Part 3 (8 points / 40 minutes)

Question 10 (3 points)

We want to implement a basic shell, similar to what we did in the lab session on processes and threads. As a reminder, we describe its simplified behavior.

The shell reads the command after the prompt `$`. We assume the command line takes no arguments after the command. For instance, we accept `"ls"` but neither `"ls -l"` nor `"ls &"`. For each command the shell always creates a new process and executes the command in it. Remember that the command may not exist or be visible. The shell then waits for the command completion (or the completion of the new process).

Complete the code below and implement the behaviour described above (*do not recopy the full source code on your paper, just write the pieces of code to be added after TODO 1 to 4*).

```
/* #include-s omitted for brevity */
int main() {
  char cmd_line[64];
  while (1) {
    printf("$␣");
    fflush(stdout);

    /* Read the command line.
     * We assume it is short and contains no argument (single word)
     */
    if (read(0, cmd_line, 64) <= 1)
      continue;

    int rv;

    /* TODO 1 */

    switch (rv) {
    case 0: /* Provide code to handle such a situation */

      /* TODO 2 */

    case -1: /* Provide code to handle such a situation */

      /* TODO 3 */

    default: /* Provide code to handle such a situation */

      /* TODO 4 */
    }
  }
}
```

Question 11 (3 points)

(Synchronization problem.) A file is shared among several people. It can be either edited (i.e., written to) or read from. If one person (a writer) is editing the file, no one else should be reading from or writing to it at the same time. If one person (a reader) is reading the file, then others might read it at the same time.

Using semaphores and/or mutex locks as synchronization tools, complete the pseudo-code below to model a correct solution for this problem (*do not answer on the examn subject*).

Multiple readers and writers will arrive continuously, each one of them executing the incomplete functions below. You can assume for simplicity that there is no uninterrupted flow of readers, that is, eventually a writer will always have its chance to write the file.

Note: the annex contains a list of functions usually available on semaphores and mutexes.

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

    // TODO
}

reader(file) {

    // TODO

    read(file);

    // TODO
}
```

```
writer(file) {

    // TODO

    write(file, "... something ...");

    // TODO
}
```

Question 12 (2 points)

Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	**Burst duration**	**Priority**
P_1	5	4 (lowest)
P_2	3	1 (highest)
P_3	1	2
P_4	8	2
P_5	4	3

Arrival order is P_1, P_2, P_3, P_4, P_5, all at time 0. Consider the scheduling algorithms: FCFS, nonpreemptive priority scheduling, and round robin (RR) with $q = 4\,\text{ms}$. What are the waiting times of each process in each case? (Reminder: the waiting time of a process is the amount of time spent waiting in the ready queue.) Recopy and fill the following table with your answers:

	Waiting times (ms)				
Algorithm	P1	P2	P3	P4	P5
FCFS					
Priority					
RR, q=4					

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.

- `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);
```