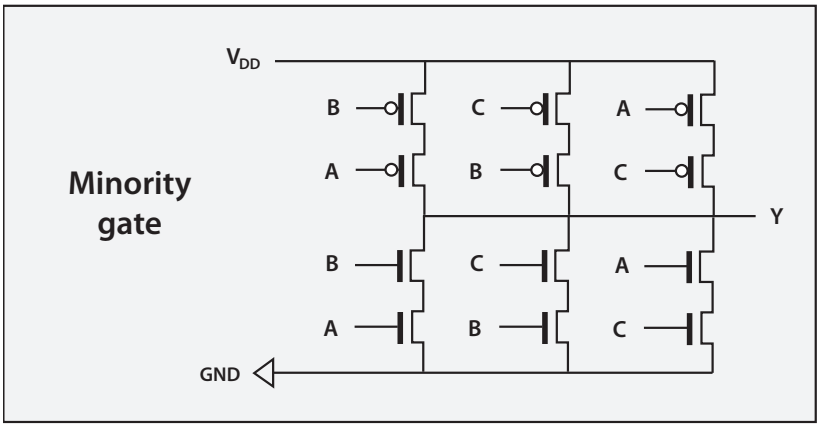
Computer Architecture
Midterm exam – 24 April 2020
From 8:45pm to 9:55pm (China time)

Exercise 1.

The minority gate is a logical gate with three inputs A, B, C and one output Y which satisfies the following specification:

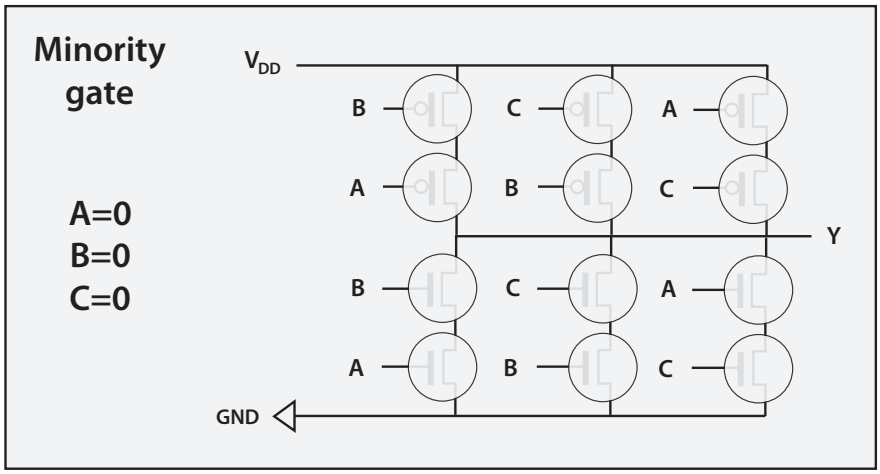
- the value of the output Y is 0 when there is a minority of input ports (zero or one) whose value is equal to 0,
- the value of the output Y is 1 when there is a minority of input ports (zero or one) whose value is equal to 1.

We want to show that the CMOS circuit depicted below does indeed implement the minority gate:

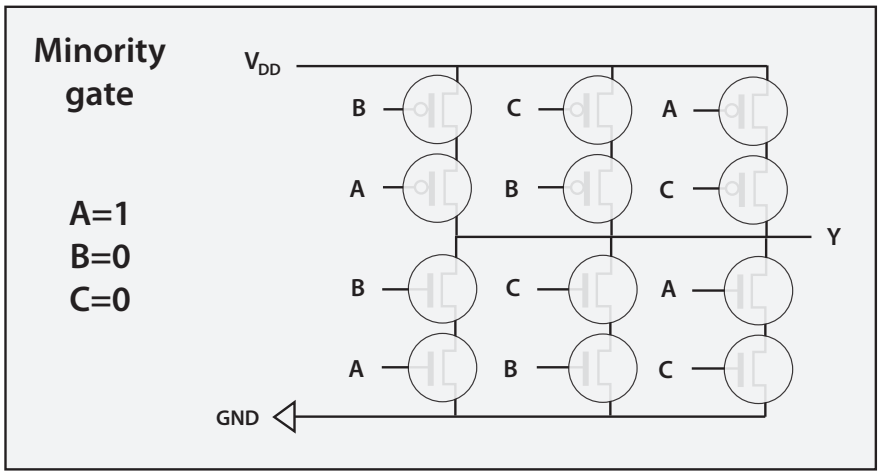


To that purpose, we test the CMOS circuit on four different inputs. Please explain using the diagrams below what is the value of the output Y for each input. You can use the diagrams to indicate which NMOS and PMOS transistors are ON and OFF, and whether the output Y is connected to the ground GND or to the drain voltage V_{DD}.

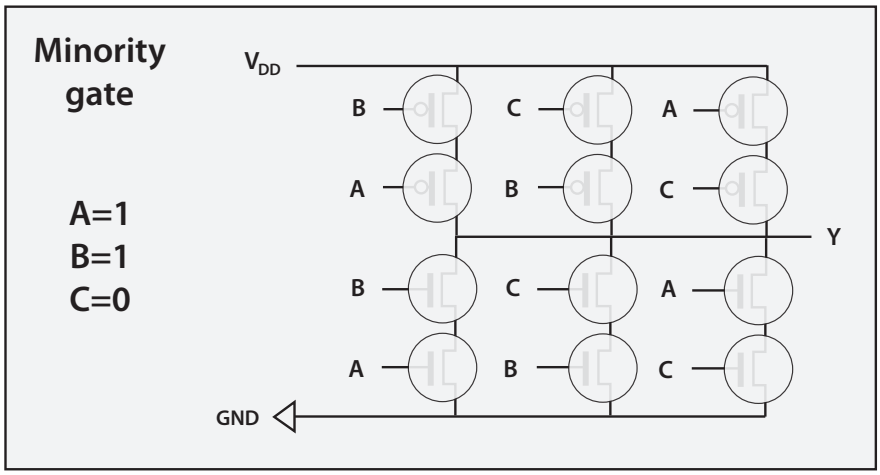
§1a. We start by testing the CMOS circuit with the inputs $A = 0$, $B = 0$ and $C = 0$.



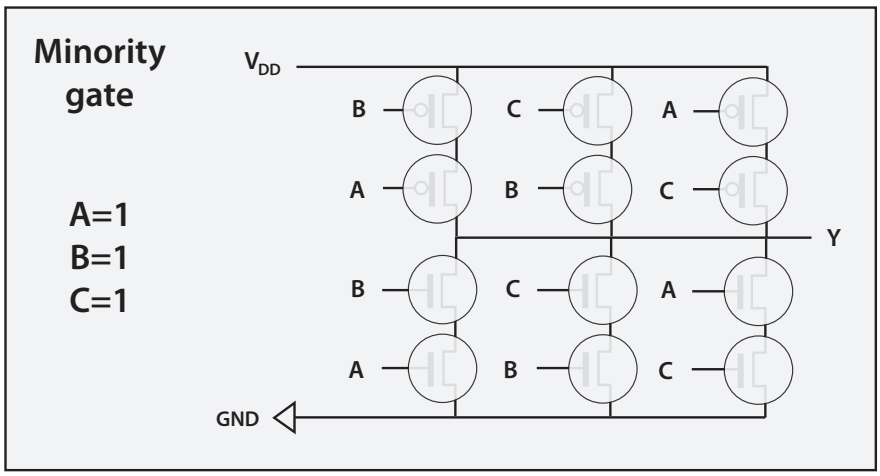
§1b. We then test the CMOS circuit on the inputs $A = 1$, $B = 0$ and $C = 0$.



§1c. We carry on and test the CMOS circuit on the inputs $A = 1$, $B = 1$, $C = 0$.



§1d. We conclude by testing the CMOS circuit on the inputs $A = 1$, $B = 1$, $C = 1$.



§1e. Explain why it is sufficient to know the output for the four inputs in order to know the behavior of the CMOS circuit for all possible inputs. Conclude that the CMOS circuit does indeed implement the minority gate.

Exercise 2.

§2a. Translate the following decimal numbers into binary numbers of length 8 bits:

7 70 99 127

§2b. Translate the same decimal numbers into hexadecimal numbers

7 70 99 127

§2c. Translate the decimal numbers into signed binary numbers (two's complement) of length 8 bits:

-1 -7 -70 -99 -127

Exercise 3.

Ben Bitdiddle and Alyssa P. Hacker are having an argument. Ben says, « All integers greater than zero and exactly divisible by six have exactly two 1's in their binary representation. » Alyssa disagrees. She says, « No, but all such numbers have an even number of 1's in their representation. » Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 4.

Ben Bitdiddle and Alyssa P. Hacker are having another argument. Ben says, « I can get the two's complement of a number by subtracting 1, then inverting all the bits of the result.» Alyssa says, « No, I can do it by examining each bit of the number, starting with the least significant bit. When the first 1 is found, invert each subsequent bit.» Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 5

Suppose that one declares the variable `myptr` in the following way

```
int *myptr
```

at the beginning of a function.

§5a. Describe in a few words the type of the variable `myptr`.

§5b. What is the size of the memory region reserved (or allocated) to the variable `myptr` during the function call?

§5c. Explain the meaning of the two notations

`myptr`[3] `*(myptr+3)`

and explain why they may be considered as equivalent.

§5d. Can you give one good reason of the fact that every pointer in C is not just a pointer, but a pointer to a specific type?

Exercise 6.

§6a. Consider the following C function

```
1.      int *myaddress()
2.      {
3.          int n;
4.          int *myptr;
5.
6.          myptr = &n;
7.          return myptr;
8.      }
```

and describe in a few words the type of the local variables `n` and `myptr`.

§6b. What is the size of memory allocated to each of the two variables `n` and `myptr` during the function call?

§6c. Explain what the assignment instruction in line 6 does, and why it makes sense despite the fact that the variable `n` has been declared but not initialized at this point.

§6d. Explain what the function `myaddress` does and what it could be used for. Indicate in particular if the pointer returned by the function `myaddress` can be safely used by the caller in order to store the value of an integer.

Exercise 7.

In all this exercise, we consider the following structured type `cell` whose purpose is to represent a cell in a linked list of character strings:

```
struct cell {
    char *key;
    struct cell *next;
};
```

Note that the structured type `cell` is the same as in Homework 5.

We use the function `lengthof` which computes the number of letters appearing in a string. Typically,

```
lengthof("Hello")
```

is equal to 5, the number of letters in the word Hello.

§7a. Depict with a diagram the typical organisation in memory of a linked list of character strings.

§7b. Explain why it makes more sense to think of a linked list as a *pointer* to a structure of type `cell`, rather than (more simply) as a structure of type `cell`.

§7c. The following C function is designed to clone a character string in memory.

```
char *clonestring(char *string) {
    char *newstring;
    int n, length;
    length = lengthof(string);
    newstring = malloc(length);
    for (n=0; n<length; n++) {
       newstring[n] = string[n];
    }
    return newstring;
}
```

Unfortunately, the code is not correct. Can you detect the bug, explain what could go wrong at run time, and then suggest how to correct the code.

§7d. A small and pretty stupid bug is hidden in the code of the function `cons` whose intended purpose is to append a character string `string` to a linked list `list` of character strings. Can you detect the bug, explain what undesired effect could happen at run time, and suggest how to correct it ?

```
struct cell *cons(char *string, struct cell *list) {
    struct cell *newlist;
    newlist = malloc(sizeof(struct cell *));
    newlist->key = clonestring(string);
    newlist->next = list;
    return newlist;
}
```

§7e. A pretty serious bug is hidden in the code below. The intended purpose of the `freelist` function is to deallocate a linked list `list` of character strings. Can you detect the bug, explain what undesired effect could happen at run time, and suggest how to correct it ?

```
int freelist(struct cell *list){
    struct cell *nodeptr;
    while (list != NULL){
       nodeptr = list;
       list = list->next;
       free(nodeptr);
    }
    return 0;
}
```

§7f. **[more difficult and if you have time]** One bug is hidden in the code of the function `clonelist` below. The intended purpose of the function is to « clone » a linked list of character strings, that is, to produce a linked list `clonelist(list)` entirely separated in memory from `list`. First, explain how the program works at run time, and describe in particular the role of the variables `cloned` and `nodeptr`. Then, detect the bug, the undesired effect which could happen at run time, and suggest how to correct it. Do not be afraid to draw pictures with cells and pointers to explain the situation...

```
struct cell *clonelist(struct cell *list){
   struct cell *cloned;
   struct cell *nodeptr;

// when the list is empty the function
// returns the NULL pointer
   if (list == NULL){
     cloned = NULL;
   }

// when the list is nonempty
   else {
// cloned is initialized
// with a malloc to the first node
     cloned = malloc(sizeof(struct cell));
// cloned->key is initialized
// with the value of list->key
     cloned->key = list->key;
// nodeptr is initialized
// with the address of the cloned list
     nodeptr = cloned;
// the list is shifted to the next node of the list
     list = list -> next;
// the while loop is executed
// as long as the list is nonempty
     while (list != NULL){
       nodeptr->next = malloc(sizeof(struct cell));
       nodeptr = nodeptr->next;
       nodeptr->key = list->key;
       list = list->next;
     }
// end of the while loop
// one should not forget to store
// the NULL pointer in the field next
// of the last node of the cloned list
     nodeptr->next = NULL;
   }

// return the pointer of the cloned list
   return cloned;
}
```

Exercise 8.

§8a. Can you explain in a few words the difference between a CISC and a RISC architecture?

§8b. What are the three different formats of MIPS instructions? Can you say in a few words what distinguishes them fundamentally?