Inf2C Computer Systems

Tutorial 3, Week 6

Solutions

1. **C and fun with pointers.**

   (a) What will the following piece of C code do and why?

   ```
   int *a = 10; *a = 100;
   ```

   Answer *Most likely, it will make the program crash. The reason is that the statement* `int *a = 10;` *assigns a value to the pointer, making it point to address* `0x0000000A` *in memory. However, this memory has not been allocated (or at least, not that we know) and is not accessible by the user's program. Then, statement* `*a = 100;` *tries to change that piece of memory and give it a value of* `100`*, which results in the program crashing.*

   (b) If there is something wrong with the previous C code, can you fix it?

   Answer *Basically make the pointer point to something on the heap, either through* `malloc()`*, or by using an additional integer variable and assigning its address to* `a`*, i.e., either:*

   ```
   int *a = (int *) malloc(sizeof(int));
   ```

   or

   ```
   int x; int *a = &x;
   ```

   A comment on the return type of `malloc`*. It is the special type of generic pointers* `void*`*. C allows casting between this type and any other pointer type. Indeed, the explicit cast is optional – C will insert the appropriate type conversion if it is left out. That is, the following two statements are equivalent:*

   ```
   int *a = (int *) malloc(sizeof(int));
   int *a = malloc(sizeof(int));
   ```

   (c) Given the following declaration:

   ```
   int **array;
   ```

   allocate a triangular array of `n` rows. That is, row `0` should have 1 column, row `1` should have 2 columns and so on. Each cell of a column should have an initial value equal to the current row, *i.e.*, `array[0][0] == 0`, `array[1][0] == array[1][1] == 1` and so on.

   Answer *Here's a sample implementation:*

   ```
   int n;
   int **array;
   n = ...;

   int i, j;
   array = (int **) malloc(n*sizeof(int *));
   // iterate over rows
   for (i = 0; i < n; i++) {
      array[i] = (int *) malloc((i+1)*sizeof(int));
      for (j = 0; j <= i; j++)
         array[i][j] = i;  // or *(*(array+i)+j)
   }
   ```

   The fist `malloc()` *call allocates as many pointers to integers as there are rows in the array, the second one allocates as many integers as there are columns in each row.*

(d) How would you de-allocate the array you allocated in the previous question?

Answer *Memory is de-allocated in inverse allocation order:*

```
for (i = 0; i < n; i++)
   free(array[i]);    // de-allocate each row
free(array);          // de-allocate the table
```

2. **Ripple Carry Adder**

Assume that the propagation delay in each gate in a 32-bit ripple carry adder is 100ps. How fast can a 32-bit addition be performed? Assuming that the adder delay is the major limiting factor on the clock speed, how fast can we clock the processor?

Answer *The longest path is from carry in at bit 0 to sum 31. For the full adders at bit positions 0-31, the carry propagates through 2 gates: one AND and one OR (cout = ab + ac + bc) For the last full adder the slowest path is to the sum output, not the carry output. The delay is 3 gates: inverter, AND, OR. ($s = \overline{a} \cdot \overline{b} \cdot c + \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b}\overline{c} + a \cdot b \cdot c$) So total 31 full-adders * 2 gates each + 3 gates for the last adder = 65 gate delays * 100ps = 6.5ns.*

If the carry-in for bit 0 is always 0, an optimisation is to replace the bit 0 full adder by a half adder ($c_{out} = ab$, $s = \overline{a} \cdot b + a \cdot \overline{b}$.) In this case, the longest delay is 64 gate delays = 6.4ns.

In standard synchronous digital logic designs, the clock period must be longer than the maximum delay through the combinational logic, in order to ensure that the clocked sequential elements following the combinational logic always capture the expected values. Therefore the maximum clock frequency is 1/6.5ns = 154 MHz.

3. **Design with basic logic elements.**

Adapted from Dr. Ian Wassell's Digital Electronics Lecture Notes.

You are tasked with designing a fuel control system to be installed in a heating boiler. The boiler monitors a number of factors through dedicated sensors which produce a single output value out of {*go*, *stop*} to control the fuel valve. The sensors used are:

(i) house temperature above temperature range `t`, (ii) fuel level `f`, (iii) boiler chimney clear `c`, and (iv) pilot light `p`.

(a) The fuel valve should be enabled when all sensors output *go*. What basic logic circuit would you employ to control the fuel valve based on all the sensor outputs?

(b) The boiler employs multiple temperature sensors, each of which checks if the house temperature is in the specified range. The system has 4 such sensors:

i. `s0` that outputs *go* if the house temperature is below or equal to 16C and *stop* otherwise.

ii. `s1` that outputs *go* if the house temperature is between 16C and 18C (inclusive) and *stop* otherwise.

iii. `s2` that outputs *go* if the house temperature is between 18C and 21C (inclusive) and *stop* otherwise.

iv. `s2` that outputs *go* if the house temperature is between 21C and 25C (inclusive) and *stop* otherwise.

An operator chooses which temperature sensor's output should be used to control the boiler at any given time. Which basic logic circuit can help you choose between the outputs of the 4 temperature sensors?

Answer *The sensor outputs are binary; go or stop.*

(a) The fuel valve should be enabled when ALL sensors output go. This requires AND logic which takes the sensor outputs as an input (AND gate with >2 inputs or a cascade of AND gates).

(b) A 4-input multiplexer is required to select between the outputs of multiple temperature sensors. The multiplexer takes in all the temperature sensor outputs and inputs and the temperature preference can be used as the 2-bit select signal.