Objetivos

Esta práctica tiene como objetivo servir de introducción a los conceptos de C/C++ que serán necesarios en la asignatura de Fundamentos de Computadores y Redes. Se presenta una idea general de estos lenguajes, suponiendo que el alumno ya conoce Java. Los conceptos de paso de parámetros por valor y por referencia son los más importantes.

Los conocimientos relativos a C/C++ se utilizarán sobre todo en el tema 4, dedicado al lenguaje de la máquina, y son un prerrequisito para poder hacer el trabajo en grupo. Además, serán muy útiles en asignaturas posteriores de la carrera como Arquitectura de Computadores y Sistemas Operativos.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Tener conocimientos básicos de programación: concepto de variable, estructura de control, función y paso de parámetros.
- Conocer las estructuras básicas de Java.
- Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente cuestionario en el Campus Virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. Introducción

En esta asignatura se estudia cómo hace el computador para ejecutar los programas desarrollados en lenguajes de alto nivel. Python o Java son lenguajes que se ejecutan habitualmente sobre una máquina virtual que utiliza un lenguaje intermedio y entender la relación entre el código fuente en esos lenguajes y lo que ejecuta el computador resulta más complejo que comprenderla cuando se utiliza C/C++, pues no utilizan una máquina virtual. Por otro lado, C/C++ tienen una sintaxis muy parecida a Java (de hecho, Java se inspiró en C/C++), por lo que al nivel que se va a ver en esta asignatura la mayoría de los aspectos sintácticos no van a suponer ningún problema para el alumno. Por último, de los lenguajes que no utilizan una máquina virtual, el conjunto C/C++ es sin duda en la actualidad el más importante. Por todas estas razones, en esta asignatura usaremos C/C++.

El lenguaje C es un lenguaje imperativo creado por Dennis Ritchie entre 1969 y 1973. En los años 80 del siglo XX Bjarne Stroustrup desarrolló una extensión de C que facilitaba el desarrollo de programas orientados a objetos y la llamó C++. Es prácticamente un superconjunto de C (añade elementos a C), así que los programas de C suelen poder ser traducidos con compiladores de C++, pero no a la inversa.

2. Entorno de programacion Microsoft Visual Studio

En esta asignatura vamos a utilizar el compilador de C/C++ que viene incluido dentro de Visual Studio, un IDE (*Integrated Development Environment*) de Microsoft que también se puede utilizar para desarrollar en otros lenguajes. Existen varias versiones de Visual Studio. Vamos a usar la versión Community Edition, que se proporciona de manera gratuita a todo el mundo. Es posible que los ordenadores de la escuela tengan instalada también la versión Professional o Enterprise. Para comprobar cuál se está utilizando se puede ir a la opción **Ayuda › Acerca de**.

Visual Studio organiza el desarrollo en lo que denomina «Soluciones» que a su vez están compuestas de «Proyectos», que pueden ser de distintos tipos según el lenguaje, las bibliotecas y las plataformas a los que vayan dirigidos. Vamos a crear un nuevo proyecto:

> - Arranca el Visual Studio. Si te pide una cuenta, utiliza la de Uniovi, que está registrada con Microsoft. Si pregunta qué entorno de programación quieres utilizar, escoge C++.
> - Selecciona en el menú **Archivo › Nuevo › Proyecto**. Como en esta asignatura se van a utilizar solamente características de C/C++, de entre las plantillas de Win32 de C++, escoge **Proyecto vacío**.
> - Proporciona el nombre `0-1Ejemplo` como nombre de proyecto, cambia la ubicación del proyecto para que esté en un sitio donde la puedas encontrar fácilmente (pero no uses tu lápiz USB ya que la compilación será muy lenta en este dispositivo; copia tu trabajo al lápiz al final de la práctica), deselecciona la opción **Crear directorio para la solución** (nuestra solución sólo va a tener un proyecto, asi que no es necesario) y pulsa **Aceptar**.

Con estos pasos tendrás un proyecto vacío. En los siguientes apartados se va a ir añadiendo código y se va a explicar cómo compilar y ejecutar al mismo tiempo que se explican los aspectos más relevantes de C/C++.

3. Estructura de un programa en C

Los programas en C/C++ se organizan en dos tipos de archivo:

- Archivos de encabezado (o cabecera). Tienen la extensión `.h` (a veces `.hpp` en C++) y se suelen utilizar para definir constantes y tipos de datos (por ejemplo, clases en C++) y declarar prototipos de funciones. El prototipo de una función dice qué devuelve la función, cuál es su nombre y qué argumentos recibe.
- Archivos de código fuente. Tienen la extensión `.c` para programas en C y `.cpp` para programas en C++.

En los archivos de encabezado se colocan elementos que se utilizan en varios ficheros de código fuente. Por ejemplo, una función `CuentaPalabras` que pueden utilizar muchos otros ficheros tendrá su prototipo en un archivo `.h` que incluirán todos los archivos de código fuente que quieran utilizarla. Su implementación estará en un solo archivo de código fuente con extensión `.c`.

No vamos a entrar aquí a explicar cómo organizar correctamente el código C/C++ en varios archivos. Los programas que haremos en la asignatura tendrán todo el código en un solo archivo de código fuente, excepto los elementos que vamos a utilizar para imprimir y que forman parte de la biblioteca estándar de C++: un conjunto de funciones, clases y variables que permiten realizar operaciones comunes como escribir por pantalla, manejar cadenas o archivos.

> Puedes obetener más información consultando guías de estilo de programación en C++.

Vamos a hacer el primer programa:

> - Pulsa con el botón derecho sobre **Archivos de origen** en el explorador de soluciones de Visual Studio.

- Escoge la opción **Agregar › Nuevo elemento**.
- Dentro de plantillas de *Visual C++*, escoge las de *Código* a la izquierda y selecciona a la derecha *Archivo C++ (.cpp)*.
- Dale el nombre `Ejemplo` y pulsa **Agregar**.

Verás que se ha creado un nuevo archivo de código fuente, denominado `Ejemplo.cpp`. Visual Studio abre automáticamente el editor para que empieces a programar. Vamos a hacer un programa que imprima en la consola "Hello, World!". Copia el siguiente código:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!";

    return 0;
}
```

Como puedes comprobar, el código tiene cierto aire a Java pero con elementos que te serán desconocidos:

- La primera línea incluye un fichero de cabecera, `iostream`, que contiene el prototipo de los elementos de la biblioteca estándar de C++ para realizar entrada/salida mediante flujos (*input/output stream*, de ahí el nombre). Los `include` cumplen una función similar a los `import` de Java o de Python.

  > Fíjate que, a pesar de ser un fichero de cabecera, no lleva `.h`. Es un caso especial por motivos históricos.

- Todos los programas en C tienen que tener una función `main` y por ella empezará la ejecución del programa. Esto es similar al método público `main` que deben tener los programas de Java. Sin embargo, las funciones no están dentro de una clase en C, ya que no existen clases en C.
- Para imprimir se utiliza un flujo estándar denominado `std::cout` (de *character out*). Para enviar elementos al flujo para que se impriman se utiliza el operador `<<`, que envía al flujo de su izquierda lo que tenga a la derecha.

Comprueba que el programa funciona pulsando `Ctrl`+`F5`, que compila, enlaza y ejecuta el programa. Si se cierra la ventana sin que te dé tiempo a verla, añade `system("pause")` antes del `return` de la función `main()`.

El resultado es similar a lo que lograrías en Java con un `System.out.println`, excepto en que no hay un salto de línea al final. Para añadirlo, hay que enviar un salto de línea al flujo de salida. Los saltos de línea se indican mediante el elemento `std::endl`. Por lo tanto, debes cambiar la línea de impresión a esta:

```
    std::cout << "Hello, World!" << std::endl;
```

El `std` que antecede a `cout` y `endl` es el espacio de nombres (*namespace*) de la biblioteca estándar y sirve para evitar conflictos si otra biblioteca decide denominar a un elemento `cout` o `endl`. Como es engorroso añadir el `std` continuamente cuando no hay conflictos, C++ ofrece la posibilidad de indicar que ciertos identificadores serán siempre de un *namespace* utilizando la directiva `using`. Aquí tienes el programa completo usando esta directiva y, a modo de ejemplo de cómo imprimir varios valores, con una línea que imprime números y cadenas:

```
#include <iostream>

using std::cout; // cout will be the same as std::cout
using std::endl; // endl will be the same as std::endl
```

```
int main()
{
    cout << "Hello, World!" << endl; // std is not required
    cout << "Integer: " << 3 << " Floating-point: " << 4.5 << endl;

    return 0;
}
```

Para leer de consola se utiliza el flujo `std::cin` junto con el operador `>>`, como se muestra en este ejemplo:

```
    int i;

    cout << "Input an integer: ";
    cin >> i; // a 'using std::cin' at the beginning is assumed
```

4. Similitudes y diferencias básicas con Java

Al haberse inspirado Java en la sintaxis de C/C++, hay muchos aspectos comunes, pero también hay muchos diferentes. No se van a tratar aquí todos estos aspectos, sino sólo los más relevantes para la asignatura.

Entre los aspectos comunes, la sintaxis general, incluyendo el uso de punto y coma para separar sentencias, las llaves para delimitar bloques de código y la forma de declarar variables, es la misma. Las estructuras de control básicas (`if`, `while`, `for` y `switch`) son similares (hay pequeñas diferencias que no afectan a lo que vamos a ver aquí).

Los tipos de datos básicos de C/C++ son muy parecidos a los de Java, pero con algunas variaciones. Existen en ambos lenguajes `int` para enteros, `float` para reales y `double` para reales más largos. Una de las diferencias básicas es que en C el tamaño de estos tipos no está definido exactamente por el estándar, así que puede variar entre sistemas. En la actualidad, es bastante común que `int` sea un entero de 32 bits, como en Java. Existen los modificadores `short` y `long` que, añadidos a `int` (por ejemplo, `short int`), permite obtener enteros con menos o más bits en algunas arquitecturas. Los `float` suelen ser de 32 bits y los `double` de 64 bits, en ambos casos de manera similar a Java.

Otros modificadores que se pueden añadir a los enteros y que Java no posee son `signed` y `unsigned`. El primero permite definir números con signo y el segundo sin signo, de manera que si en una arquitectura los `int` son de 32 bits, `signed int` utilizará estos 32 bits para representar números positivos y negativos entre -2147483648 y 2147483647 (como en Java) y `unsigned int` utilizará los 32 bits para representar sólo números positivos, con lo que se podrán representar números mayores, en concreto, números entre 0 y 4294967295 (el doble de números positivos que en Java). Estos rangos surgen del sistema de codificacion usado, que se estudiará con detalle en la asignatura. Lo habitual si no se indica el tipo de número es que se suponga que es un número con signo.

Para comprobar los problemas a los que puede llevar esto, modifica el código de la función `main` para que sea el siguiente:

```
int main()
{
    unsigned int a = 23;
    int b = -5;

    cout << "a: " << a << " b: " << b << endl;

    if (a < b)
    {
```

```
        cout  << "a less than b" << endl;
    }

    return 0;
}
```

Compila y ejecuta el programa. Como verás, se imprime que `a` es menor que `b`, lo que no tiene sentido ya que antes se ha imprimido que `a` vale 23 y `b` vale -5. Las razones detalladas de este comportamiento las podrás comprender tras estudiar la asignatura.

> En cualquier caso, el compilador te puede ayudar a evitar este tipo de errores: mira en la ventana **Lista de errores** situada en la zona inferior. Verás mensaje de aviso (*warning*) que te informa de que estás comparando variables con tipos que no coinciden. En general, debes considerar los *warnings* como errores a no ser que comprendas exactamente por qué se producen y estés seguro de que la situación de la que te avisa el compilador no puede generar un error.

Otra diferencia con respecto a los tipos básicos entre Java y C/C++ es que utilizan sistemas de codificación distintos para los caracteres. En Java, un `char` son 16 bits codificados en Unicode, en concreto en UTF-16 (verás más adelante en la asignatura qué son estos sistemas de codificación). En C/C++ un `char` puede variar en distintas implementaciones pero lo mas habitual es que sea de 8 bits y se utilice un sistema de codificación derivado de ASCII como ISO Latin-1.

En esta asignatura no vamos a utilizar tipos complejos, como por ejemplo clases, pero debes saber que hay bastantes diferencias entre la forma de implementar los tipos complejos en Java y C/C++.

5. Paso de parámetros a funciones

La definición de funciones en C/C++ es similar a Java. Sin embargo, hay diferencias en el mecanismo de paso de parámetros. Como ya deberías saber, en Java los parámetros se pasan siempre por valor. En C también es así. En C++, en cambio, se pueden pasar por valor o por referencia. Cuando se pasan por valor, los cambios dentro de la función no afectan al valor fuera de la función; cuando se pasan por referencia, la función sí puede cambiar el valor que tiene la variable pasada fuera de la función.

Veamos primero un ejemplo de paso por valor:

```
#include <iostream>

using std::cout;
using std::endl;

void set23(int number)
{
    number = 23;
}

int main()
{
    int i = 10;

    set23(i);
    cout << "i now contains: " << i << endl;

    return 0;
}
```

¿Qué valor para `i` crees que imprimirá el código anterior? Responde en el cuestionario: pregunta 1. Introdúcelo en el Visual Studio y comprueba si tu respuesta es correcta. Como verás, una función así no tiene mucho sentido.

El paso anterior era por valor, al igual que en Java. En C/C++, para pasar por referencia un parámetro se debe utilizar en su definición el símbolo `&` (en español se llama *et*, pero en programación se usa habitualmente con su nombre inglés, *ampersand*). Modifica el programa anterior para que la definición de la función sea la siguiente:

```
void set23(int& number)
```

Fíjate que en la declaración de la función el parámetro ahora es `int&`, es decir, una referencia a un entero. Compila el programa y ejecútalo. ¿Qué se imprime para `i`? Responde en el cuestionario: pregunta 2. Como habrás comprobado, ahora la función modifica el parámetro que se le pasa.

Para entender la diferencia entre el paso por valor y por referencia, debes de tener claro que cuando se pasa por valor lo que se está haciendo es hacer una copia del valor original. Es decir, en el paso por valor del primer programa de esta sección, al hacer `set23(i)` se está copiando el valor de `i` en otra zona de memoria, en concreto, en la variable `number`. Sin embargo, cuando se hace un paso por referencia, no se realiza una copia del valor. En el segundo programa lo que se hace es que la variable `number` haga referencia a la misma dirección de memoria que `i`. En el tema 4 se explicará esto con más detalle.

6. Vectores

A un nivel básico, los vectores (*arrays*) son similares en Java y C/C++. En Java, los vectores son estructuras de datos que guardan varios elementos del mismo tipo y pueden ser accedidos por un índice entero. En C, además, se asegura que los elementos están contiguos en memoria.

Se pueden definir vectores en C/C++ de varias formas. En la asignatura vamos a utilizar sólo una que reserva memoria para el vector a la vez que lo declara. Para ello, simplemente se debe poner el tipo de los elementos del vector, el nombre del vector y, entre corchetes, cuántos elementos va a tener. Por ejemplo, el siguiente fragmento de código define y reserva memoria para un vector de enteros con cuatro elementos:

```
int v[4];
```

Como este tipo de vectores ya tiene memoria asignada, no es necesario, al contrario que en Java, llamar a `new` para reservar memoria. A continuación se muestra un ejemplo de definición y uso:

```
int v[4];

v[0] = 2131; // first element
v[3] = 112;  // last element
```

También se pueden inicializar los elementos de un vector durante su definición. En este caso, no es necesario indicar entre los corchetes el número de elementos del *array*: se tomará del número de elementos que se inicialicen. Por ejemplo:

```
int v[] = {300, 123, 12}; // 3-element array
```

Para pasar un vector a una función, se pone el tipo de los elementos, el nombre del parámetro y los corchetes vacíos, como en este ejemplo:

```
void f(int v[]); // function f receives an integer array as a parameter
```

El tipo de vectores de C/C++ explicado aquí no tienen la propiedad `length` que tienen los vectores en Java. Por lo tanto, cuando se pasa un vector a una función, es habitual pasarle en otra variable el tamaño, como en este ejemplo:

```
void f(int v[], int longV);
```

Este tipo de vectores se pasan siempre por referencia.

7. Cadenas

En C las cadenas son vectores de caracteres que acaban con un carácter especial denominado `null`, que se suele representar como `\0` y tiene el código todo ceros. Es necesario que el vector de caracteres donde se almacena la cadena tenga una posición para almacenar este carácter.

Cuando se escribe una cadena entre comillas en el código, el compilador añade automáticamente este carácter terminador. Por ejemplo:

```
char str[] = "abc"; // Character array with these four values:
                    // str[0]='a' str[1]='b' str[2]='c' str[3]=\0
```

Para imprimir cadenas por consola, se hace como se ha visto mediante `cout` y el operador `<<`. Para leer cadenas de la consola, sin embargo, no se puede utilizar directamente `cin` con el operador `>>`, ya que este operador no funciona sobre el tipo de vectores que estamos utilizando.

> C++ define una clase `String` que sí permite utilizar el operador `<<`. Sin embargo, no la utilizamos en la asignatura porque comprender cómo funciona a bajo nivel es muy complejo y, además, es fundamental entender cómo funcionan las cadenas de C ya que las utilizan el conjunto de funciones para desarrollar aplicaciones (API, de *Application Programming Interface*) que ofrecen los sistemas operativos.

Una posibilidad sería ir leyendo carácter a carácter en un bucle. Afortunadamente, el objeto `cin` ofrece una función para leer cadenas: `getline`, que recibe la cadena en la que debe dejar la secuencia de caracteres y el máximo número de caracteres que puede tener la cadena (incluyendo el terminador). Por ejemplo:

```
const unsigned int maxChars = 100; // const is used to define constants
char str[maxChars];
cin.getline(str, maxChars); // Reads at most 99 characters from console,
                            // or until return is pressed. The characters are
                            // stored in str, and \0 is added as the last
                            // element of the array
```

8. Ejemplo completo

Como ejemplo de todo lo visto hasta ahora, se va a hacer un programa que define e inicializa dos vectores de enteros y luego llama a una función que compara los elementos con el mismo índice de cada vector y, si son distintos, modifica el elemento del segundo vector para que tenga la suma de los dos elementos originales. Además, la función indica cuántos elementos ha modificado. El programa principal utiliza una función para imprimir el nombre del vector y sus elementos.

> - Descarga del Campus Virtual el fichero `0-1EjemploVector.zip`.
> - Descomprímelo (no basta con abrirlo: tienes que descomprimirlo a un nuevo directorio) y abre el fichero de solución (`.sln`) en Visual Studio. Si te pide que actualices el proyecto, acéptalo.

- Lee el código, incluyendo los comentarios, y complétalo para que funcione.
- Compílalo y ejecútalo para comprobar que está todo bien.

9. Para saber más: punteros

Una de las grandes ventajas de C/C++ es que incorpora un sistema de punteros que permite acceder directamente a posiciones de memoria, lo que es fundamental para trabajar a bajo nivel y también para desarrollar estructuras con un alto rendimiento. Como contrapartida, es fácil utilizar incorrectamente esta potencia y generar errores con consecuencias graves.

Un puntero es una variable que contiene una dirección de memoria. Se dice que el puntero apunta a lo que hay en la zona de memoria de la dirección que contiene. En C/C++, cuando se declara un puntero hay que indicar de qué tipo es lo que hay en la dirección a la que se apunta. Por ejemplo, de esta forma se declararían un puntero para apuntar a un `int` y otro para apuntar a un `float`:

```
int* pI;   // pI is a pointer to int
float* pF; // pF is a pointer to float
```

El asterisco después del tipo en la declaración es lo que indica que es un puntero. En general, se desaconseja definir varios punteros en la misma línea porque la sintaxis no es intuitiva:

```
int* a, b;   // a is a pointer to an int, b is an int (not a pointer)
int* c, * d; // both c and d are pointers to int. Not recommended
```

Una de las formas de uso de los punteros más habitual es utilizarlos para guardar la dirección de otra variable. Para obtener la dirección de una variable se utiliza el simbolo `&`. Por ejemplo:

```
int* p;           // p is a pointer to an int
int number = 35; // number is an int
p = &number;      // copy to p the address of number. p now points to number
```

La expresión `&i` se lee *dirección de i*. Otra forma alternativa de decir que un puntero apunta a algo es decir que referencia ese algo. Los punteros se pueden desreferenciar, es decir, obtener lo que hay en la memoria a la que apuntan. Por ejemplo:

```
int* p;           // p is a pointer to an int
int number = 35; // number is an int

cout << "number: " << number << endl;

p = &number; // copy to p the address of number. p now points to number
*p = 555;    // write 555 in the address pointed by p, i.e., in number

cout << "number: " << number << endl;
cout << "*p: " << *p << endl;
```

Ejecuta el código anterior, y comprueba que accediendo a la zona de memoria de la variable `number` con su nombre o con `*p` se muestra lo mismo. Si ejecutases esta instrucción a continuación:

```
number = 101;
```

¿Cuánto valdría `*p`? Responde en el cuestionario: pregunta 3. Comprueba tu respuesta añadiendo la instrucción anterior e imprimiendo después el valor de `*p`.

Fíjate que el operador asterisco se utiliza en dos contextos relacionados con punteros de manera muy distinta: en la definición de variables (por ejemplo, `int* p`) indica que se está declarando un puntero y en la desreferenciación (por ejemplo, `*p =555`) indica que se quiere acceder al valor que hay donde apunta el puntero.

Uno de los problemas de los punteros es que pueden no estar inicializados. Por ejemplo, este programa es incorrecto:

```
int* p;     // p is a pointer to an int
*p = -3400; // ERROR: It is unknown where p points
```

Como `p` no se ha inicializado, puede contener cualquier dirección. Al escribir en `*p`, ¡estamos escribiendo en cualquier dirección! Eso puede tener consecuencias desastrosas: imagínate que en un programa de un banco justo en esa dirección está lo que tiene un cliente en la cuenta; con esa línea, pasaría a tener -3400 euros, tuviese lo que tuviese antes. Probablemente, al cliente no le haría gracia… a no ser que tuviese una cantidad inferior, en cuyo caso al que no le haría gracia sería al banco.

Pero también puede que el *valor basura* que contiene `p` antes de ser inicializado coincida con una dirección del sistema operativo y, por lo tanto, escribas un -3400 una zona del sistema operativo. Como verás en la asignatura de Arquitectura de Computadores en 2º curso, existen mecanismos para proteger al sistema operativo que hacen que falle el programa al intentar acceder a estas direcciones en lugar de hacer fallar a todo el sistema.

A veces interesa definir un puntero a una zona de memoria sin saber qué se va a almacenar en esa zona de memoria, es decir, interesa poder tener un «puntero a cualquier cosa». En C/C++ eso se logra utilizando el tipo `void*`. Los punteros de este tipo no se pueden desreferenciar sin hacerles antes un *casting*, como muestra este ejemplo:

```
void* p; // p is a pointer to anything
int i = 23;
float f = 55.2;
p = &i;
int i2 = *((int*) p);
p = &f;
float f2 = *((float*) p);
```

Como ves, la sintaxis empieza a ser engorrosa. Para entender el `*((int*) p)` debes pensar que se parte de `p`, que es un `void*` (puntero a cualquier cosa) y se le hace un casting a `int*`, es decir, `((int*) p)` es un puntero a entero; el asterisco que se añade a la izquierda en `*((int*) p)` quiere decir desreferenciar `((int*) p)`, es decir, ir a la dirección que contiene `p`, suponer que ahí hay un entero y obtener el valor de ese entero.
Con `*((float*) p)` se hace lo mismo pero suponiendo que `p` está apuntando a un número en punto flotante.

Estos conceptos de punteros son complejos. Precisamente el estudio de cómo funciona el computador a bajo nivel que se hace en esta asignatura te permitirá entenderlos mucho más fácilmente.

10. Ejercicios

- Crea un programa que defina dos vectores de 6 enteros. Inicialízalos con valores aleatorios, la mitad positivos y la mitad negativos. El programa debe contener una función que reciba un vector de enteros y su longitud y ponga a cero los elementos negativos. El programa principal debe llamar a esta función para los dos vectores. Deben imprimirse los vectores antes y después de modificarlos.
- Crea un programa que defina dos cadenas e imprima el número de vocales y el número de consonantes que tienen. Para ello debe utilizar una función que realice el cálculo.

Apéndice A: Para saber más: printf

Como se vio anteriormente, en C++ se puede imprimir por pantalla mediante `cout`. Esto no funciona en C, donde la forma estándar de hacerlo es utilizando la función `printf` que está incluida en el fichero de cabecera `stdio.h`, que contiene el prototipo de las funciones de entrada/salida de la biblioteca estándar de C (*standard input/output*, de ahí el nombre). Por ejemplo, para imprimir `Hello, World!`, se haría así:

```
printf("Hello, World!\n");
```

El `\n` final produce un salto de línea, ya que `printf` no produce un salto de línea por defecto. No se puede utilizar `endl` para este cometido porque también es una característica de C++.

Debido a la forma de manejar las cadenas en C, imprimir varios valores es más complejo que en Java y en C++. No se puede utilizar el operador `+` para concatenar cadenas ni el operador `<<` para encadenar salidas por pantalla. La forma de imprimir varios valores es utilizando varios parámetros en la función `printf`. El primero es una cadena, denominada cadena de formato, que puede incluir unos indicadores de elementos que se sustituirán por los valores de los siguientes parámetros. La mejor forma de entenderlo es con un ejemplo:

```
printf("Integer1: %d. Integer2: %d. Float: %f\n", 23, 100, 45.3);
```

Si añades esto a un programa, lo compilas y lo ejecutas (recuerda que también debes incluir `stdio.h`), comprobarás que el primer indicador `%d` se ha sustituido por el segundo parámetro, el segundo indicador `%d` se ha sustituido por el tercer parámetro y el indicador `%f`, por el cuarto. Existen indicadores para distintos tipos de datos. En Internet puedes encontrar más información sobre `printf`. Es interesante conocerla porque la sintaxis de cadenas de formato ha pasado a muchas funciones de otros lenguajes. En Java, por ejemplo, está `String.format` y en Python se utiliza con el operador de cadenas `%`.

Objetivos

Esta práctica analizará la forma en la que se representan números naturales y enteros en un lenguaje de alto nivel, en concreto en C++. Además, se realizarán diversas operaciones con datos numéricos para entender las repercusiones que tienen los errores de desbordamiento en un programa.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Conocer los sistemas de codificación de números naturales y enteros (binario natural, hexadecimal, signo-magnitud y complemento a 2).
- Conocer el lenguaje de programación C++.

Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente cuestionario en el Campus Virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. Codificación de enteros y naturales

Para realizar esta práctica se va a utilizar como herramienta el *Visual Studio*. Para simplificar se proporciona un proyecto mínimo ya creado. Descarga del Campus Virtual el fichero `1-Enteros` y descomprímelo.

> - Abre el proyecto haciendo doble clic sobre el fichero de solución (extensión `sln`).
> - Una vez que está abierto el proyecto compila y ejecuta el programa pulsando la combinación de teclas `Ctrl`+`F5`.

La salida que proporciona este programa es sorprendente: dice que `a` vale 23 y `b`, -5, pero que `a` es menor que `b`. Para entender por qué ocurre esto, en primer lugar hay que codificar estos números. La codificación depende del tipo con el que estén declarados:

> - La variable `a` está declarada como `unsigned int`, lo que significa, en la versión de Visual Studio empleada en prácticas, que es un número natural de 32 bits codificado en binario natural. ¿Cuál deberá ser la codificación de `a`? Indica el resultado en hexadecimal. Responde en el cuestionario: pregunta 1.
> - La variable `b` está declarada como `int`, lo que significa, en la versión de Visual Studio empleada en prácticas, que es un número entero de 32 bits codificado en complemento a 2. ¿Cuál deberá ser la codificación de `b`? Responde en el cuestionario: pregunta 2.

Para analizar la codificación de las variables en memoria es necesario ejecutar el programa en modo depuración y pausar su ejecución. La ejecución de un programa se puede pausar utilizando puntos de interrupción, que permiten interrumpir la ejecución del programa en cualquier línea. Realiza los siguientes pasos:

- Sitúa el cursor en la línea en la que se imprime el mensaje por pantalla. Pulsa `F9` para colocar en esa línea un punto de interrupción. El Visual Studio indica que en esa línea hay un punto de interrupción dibujando un círculo rojo en el margen izquierdo. Pulsando con el botón izquierdo del ratón sobre el círculo rojo se elimina el punto de interrupción, y volviendo a pulsar se vuelve a crear.
- Ejecuta el programa en modo depuración pulsando `F5`. Fíjate que para ejecutar el programa en modo depuración no hay que pulsar la tecla `Control`.
- El programa se detendrá justo en la línea donde se colocó el punto de interrupción. La flecha amarilla en el margen izquierdo indica la próxima sentencia que se va a ejecutar.
- En este momento pasando el ratón por encima del nombre de una variable en el programa se puede obtener su valor.
- Otra forma de inspeccionar el valor de las variables es la ventana denominada «Automático», donde se muestran automáticamente variables utilizadas cerca del código ejecutado. También existe la posibilidad de escoger las variables que queremos ver utilizando la ventana «Inspección». Sigue estos pasos para abrir esta ventana: pulsa en el menú **Depurar › Ventanas › Inspección › Inspección 1**.
- Escribe en el inspector de variables el nombre de las variables `a` (en una línea) y `b` (en otra línea) para ver el valor que almacenan; podrás ver su valor en decimal. Pulsa con el botón derecho del ratón en cualquier parte de la ventana **Inspección 1** y selecciona la opción **Presentación hexadecimal**. El contenido de las variables debería coincidir con la codificación que habías indicado. Si no es así, repasa la codificación y pregunta a tu profesor si no encuentras el problema.

El programa ha determinado que `a` es menor que `b` debido a que al realizar la comparación se ha encontrado con que `a` es una variable sin signo y `b` con signo y ha decidido interpretar la codificación de los dos números como variables sin signo. Esto es así porque C hace tipados (*castings*) automáticos y en el caso de una expresión que mezcle números con y sin signo, decide interpretarlos ambos como números sin signo. Para ver qué valor tiene `b` interpretado sin signo, deberás tomar su codificación e interpretarla como binario natural en lugar de complemento a 2. Como es un número muy grande, vamos a utilizar la calculadora de Windows para comprobar su valor:

- Abre la calculadora en Windows.
- En el menú escoge **Ver › Programador**.
- Pon la calculadora en modo hexadecimal pulsando sobre `Hexa` en la parte izquierda.
- Escribe la codificación de `b`.
- Pulsa sobre `Dec` para que pase el número hexadecimal a decimal.

Como verás, es un número muy grande, mucho mayor que 23 (el valor de `a`) y por eso se escribe el mensaje de que `b` es mayor que `a`. En Visual Studio, recompila la aplicación con **Compilar › Recompilar solución** (no vale una compilación normal porque como no se ha cambiado el código, no vuelve a compilar) y muestra la ventana

denominada «Lista de errores» (**Ver › Lista de errores**). Podrás ver que, durante la compilación, se generó un aviso («warning») indicando que no coincidían los tipos de las dos variables comparadas. Recuerda que debes considerar los avisos como errores a no ser que sepas qué consecuencias tienen y que en ese caso concreto no son un error.

2. Almacenamiento en memoria

Vamos a ver cómo se almacena la codificación de estas variables en memoria. Para ello realiza los siguientes pasos:

- En primer lugar es necesario determinar la dirección de memoria en la que se almacenan las variables. Ejecuta el programa en modo depuración pulsando `F5` y añade al inspector de variables `&a` (en una nueva línea) y `&b` (en otra línea). El operador `&` se utiliza para calcular la dirección a partir de la cual está almacenada una variable. El prefijo `0x` indica que el número que aparece a continuación está representado en hexadecimal. ¿En qué dirección se almacena `a`? ¿Y `b`? Responde en el cuestionario: pregunta 3.
- A continuación se examinará el contenido de la memoria en las direcciones indicadas. Para ello pulsa en el menú **Depurar › Ventanas › Memoria › Memoria 1**. Esta acción abrirá el inspector de memoria.
- En el campo dirección del inspector de memoria introduce la dirección de la variable `a`. Los datos se muestran en hexadecimal agrupados por bytes, por tanto los 32 bits que contienen la codificación de la variable `a` son los cuatro primeros grupos (4 bytes son 32 bits). Sin embargo, para poder obtener el valor de la codificación hay que tener en cuenta que los datos en memoria se almacenan en orden inverso siguiendo el criterio *little-endian*. Esto quiere decir que si se quiere almacenar en memoria el dato `12AB34CD`, la secuencia de bytes que se observará será `CD 34 AB 12`. Teniendo esto en cuenta comprueba que la codificación de `a` coincide con lo que habías respondido previamente.
- Siguiendo el mismo procedimiento, comprueba que la codificación de `b` coincide con tu respuesta anterior.
- Para continuar con la ejecución del programa pulsa de nuevo `F5`. Como no hay más puntos de interrupción, el programa terminará su ejecución.
- En el código C++, cambia el valor que se asigna a la variable `a` por 64 y el valor que se asigna a la variable `b` por -1. Codifica manualmente estos datos y repite el proceso anterior para verificar que coinciden con la codificación que se hace de estas variables en el programa.

3. Desbordamiento

El desbordamiento se produce cuando se realiza una operación aritmética y el resultado no es representable para el tipo de formato y número de bits con el que se está trabajando. Realiza los siguientes pasos para ver un ejemplo:

- Modifica el programa anterior para que se lean por teclado dos números enteros (con signo), `a` y `b`.
- Suma `a` y `b`, almacenando el resultado en otra variable entera (con signo) denominada `c`. Imprime el valor de `c` por pantalla.
- Compila y ejecuta el programa pulsando `Ctrl`+`F5`. Introduce valores para `a` y `b`, y verifica que funciona correctamente.
- Vuelve a ejecutar el programa pero esta vez introduce el valor dos mil millones para `a` (un 2 y nueve 0) y lo mismo para `b`. ¿Cuál es el resultado? Responde en el cuestionario: pregunta 4.
- Razona el resultado que se ha producido. Si no tienes clara la causa deberías consultar con tu profesor.

Se podría pensar que una forma de evitar el problema del desbordamiento es utilizar un mayor número de bits. Sin embargo, siempre se podrían encontrar dos números tal que su suma no fuera representable para ese número de bits. Por tanto, **el desbordamiento es un problema que no se puede evitar, solo detectar**.

Para detectar el desbordamiento cuando se ha realizado una suma con números enteros basta con que algunas de las siguientes dos condiciones sea cierta:

1. Que `a` y `b` sean positivos y `c` sea negativo.
2. Que `a` y `b` sean negativos y `c` sea positivo (mayor o igual que cero).

- Modifica el programa anterior para que, en caso de que la suma provoque desbordamiento, no se imprima el resultado, sino un mensaje indicándolo.

El problema del desbordamiento en sí no se puede evitar. Sin embargo, hay ciertas operaciones que, dependiendo de la forma en la que se realizan, pueden o no generar problemas de desbordamiento. El cálculo de la media aritmética es una de ellas.

- Añade al programa anterior una nueva variable entera, `d`.
- Asigna a `d` la media entre `a` y `b` e imprime por pantalla el valor de `d`.
- Ejecuta el programa e introduce nuevamente tanto para `a` como para `b` el valor dos mil millones (un 2 y nueve 0).
- Si has calculado la media como (a+b)/2 el resultado te habrá salido incorrecto. El problema en este caso no es que el resultado de la operación genere desbordamiento, ya que la media de dos números iguales es ese mismo número y, por tanto, representable. El problema en este caso es que se produce desbordamiento en una operación intermedia, en concreto en la suma que se realiza antes de la división entre dos.
- Una forma alternativa de calcular la media sería mediante la operación a/2+b/2. Matemáticamente es equivalente a la forma de calcular la media anterior, pero en este caso se evita el problema del desbordamiento en las operaciones intermedias. Modifica el cálculo de la media con esta expresión.
- Ejecuta de nuevo el programa con los mismos datos y observa los resultados.

Utilizando los conocimientos adquiridos responde a la siguiente pregunta: ¿qué valor se imprimirá por pantalla al ejecutar el siguiente fragmento de código? Responde en el cuestionario: pregunta 5.

```
const int UN_MILLON = 1000000;
int contador = 0;
for (int i = 0; i < 3000*UN_MILLON; i++)
  contador++;
cout << "contador " << contador << endl;
```

> - Añade el fragmento de código al programa y verifica que tu respuesta fue correcta.

4. Ejercicios adicionales

- ✓ Crea un programa que realice la suma acumulada de todos los números entre 1 y un millón que son múltiplos de 3. Para acumular la suma, utiliza variables de los siguientes tipos: `int` (entero de 32 bits), `short` (entero de 16 bits) y `long long` (entero de 64 bits). Compara y analiza los resultados.
- ✓ Crea un programa que lea por pantalla dos números enteros e imprima su diferencia. En caso de que dicha operación aritmética produzca un desbordamiento imprime un mensaje de error indicándolo.

Objetivos

Esta sesión analizará la forma en la que se representan números reales y caracteres en un lenguaje de alto nivel, en concreto en C++. Además, se realizarán diversas operaciones con datos numéricos para entender las repercusiones que tienen los errores de redondeo y desbordamiento en un programa.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Conocer los sistemas de codificación numéricos de números reales (IEEE-754).
- Conocer los sistemas de codificación de caracteres.
- Conocer el lenguaje de programación C++.

Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente cuestionario en el Campus Virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. Codificación de números reales

Vamos a ver cómo se codifican números reales. En primer lugar, codifica el número -27.625 en el formato IEEE 754. ¿Cuál es la codificación en hexadecimal? Responde en el cuestionario: pregunta 1.

Vamos a comprobarlo con un programa. Sigue estos pasos:

> - Descarga el fichero `2-Reales` del Campus Virtual y descomprímelo. Abre el fichero de solución en Visual Studio.
> - Modifica el programa insertando al principio del `main` la declaración de una variable llamada `f` de tipo `float`.
> - Añade a continuación una sentencia que asigne a la variable `f` el valor `-27.625`.
> - Añade un mensaje que imprima por pantalla el valor de la variable.
> - Ejecuta el programa y verifica que funciona correctamente.
> - Inserta un punto de ruptura en la sentencia que muestra el mensaje e inicia la depuración.
> - Muestra la ventana de memoria e introduce en el campo dirección `&f`. En los cuatro primeros bytes deberás ver la codificación de f. Teniendo en cuenta que está guardada usando la convención *Little Endian*, comprueba que coincide con la codificación que habías hecho tú.

2. Redondeo

Los números reales se ven afectados por los mismos problemas que los números enteros en cuanto al desbordamiento, aunque la forma de detectarlo es diferente. Además, en la representación de números reales

aparece un nuevo problema: el problema del redondeo. Este problema ocurre cuando un número real no tiene una representación exacta.

Para comprobar en qué consiste el problema de redondeo convierte el número 0.1 a binario. ¿Cuál es su codificación? Responde en el cuestionario: pregunta 2. El número en binario no tiene una representación exacta con lo que se tendrá que trabajar con aproximaciones.

A continuación se van a comprobar los problemas que pueden generar este tipo de errores:

- En primer lugar comenta todo el código del `main`.
- Añade dos variables de tipo `float` `f1` y `f2`.
- Asigna el valor 0.1 a `f1` y 0.3 a `f2`.
- Imprime por pantalla el valor de ambas variables.
- Compila y ejecuta el programa pulsando `Ctrl`+`F5`.
- Aparentemente se imprime el valor exacto, pero eso se debe a que por defecto se imprimen pocos decimales. Para imprimir los números reales con más decimales en primer lugar hay que añadir al programa un nuevo fichero de cabecera:

```
#include <iomanip>
```

- A continuación se puede indicar a `cout` que se quieren imprimir más decimales de la siguiente forma:

```
cout << "f1: " << setprecision(15) << f1 << endl;
cout << "f2: " << setprecision(15) << f2 << endl;
```

- Ejecuta de nuevo el programa y observa los resultados.

Ciertamente los errores de redondeo son muy pequeños. Sin embargo, si no se tienen en cuenta se pueden generar graves problemas.

- Añade al programa una sentencia condicional que imprima si `f1·3` es igual a `f2` o si son distintos. Dado que estamos trabajando con números reales, todos los números de la expresión deben ser reales para que el compilador no aplique un tipado diferente: la expresión a utilizar debería ser `f1*3.0`.
- Ejecuta el programa y comprueba los resultados.

El problema es que los errores de redondeo cometidos al representar `f1` y hacer la operación `f1*3.0` son distintos a los errores cometidos al representar `f2`, lo que hace que las magnitudes comparadas sean ligeramente diferentes. Por este motivo para comparar dos números reales se utiliza siempre una expresión similar a la siguiente:

```
const float TOLERANCIA = 0.0000001;
if (fabs(f1*3.0 - f2) < TOLERANCIA)
  cout << "Son iguales" << endl;
else
  cout << "Son distintos" << endl;
```

- Añade el fragmento de código al programa y verifica que ahora la comparación da el resultado esperado. La función `fabs` calcula el valor

absoluto de un número y está definida en el archivo de cabecera `math.h`. El valor de la tolerancia depende del tipo de datos que se manejen en la aplicación.

Un error de redondeo muy pequeño puede dar lugar a un error muy grande si se realizan muchas operaciones con números reales. Los pequeños errores se van acumulando y finalmente el error empieza a ser significativo.

Supongamos que se almacena sobre una variable de tipo `float` el número de euros que una persona tiene en el banco. ¿Qué ocurriría si dicha persona realiza 20 millones de ingresos de 10 céntimos cada uno?

- Añade al programa el siguiente fragmento de código que simularía el caso anterior:

```
float euros = 0;
const int UN_MILLON = 1000000;
for (int i = 0; i < 20*UN_MILLON; i++)
  euros = euros + 0.1;
cout <<"euros = " << euros << endl;
```

- 20 millones multiplicado por 0.1 debería dar como resultado 2000000 (2 millones). Ejecuta el programa y observa el error que se produce.
- Piensa que ocurriría si en lugar de ingresar 0.1 euros se ingresase 1 euro cada vez. En este caso el número 1 tiene una representación exacta donde no se producen errores de redondeo. ¿Crees que el resultado será correcto?
- Haz los cambios oportunos y comprueba lo que ocurre en este caso.
- Nuevamente aparecen errores de redondeo. Aunque el número 1 tiene una representación exacta, no ocurre lo mismo con otros números que aparecen en los cálculos.
- Añade el siguiente código al programa:

```
float f3 = 19*UN_MILLON;
cout << "f3: " << setprecision(15) << f3 << endl;
f3 = f3+1;
cout << "f3 + 1: " << setprecision(15) << f3 << endl;
```

- Reflexiona sobre el resultado que va a producir el código que se acaba de añadir. Ejecuta el programa y analiza los resultados.

Por todos estos motivos, la utilización de números reales en los programas sin conocer sus limitaciones debidas a los errores de redondeo puede provocar grandes problemas. Es por tanto esencial para el desarrollo de programas robustos el conocimiento de la forma en la que se codifica la información.

3. Codificación de caracteres

Vamos a ver a continuación cómo se codifican caracteres en C++. Sigue estos pasos:

- Comenta el código de la función `main`.
- Define una variable `c1` de tipo `char` y asígnale el valor `'a'`. Fíjate que para indicar un carácter individual se utilizan las comillas simples.

- Define otra variable `c2` también de tipo `char` y haz que contenga el carácter `'ñ'`.
- Añade una sentencia que imprima los dos caracteres separados por un espacio.
- Ejecuta el programa y comprueba que la `a` se imprime correctamente pero la `ñ` no.

Vamos a analizar lo que está ocurriendo. En primer lugar, debes saber que el tipo `char` de C sirve para almacenar un byte. Vamos a ver en concreto qué secuencia de bits se almacena en las dos variables que hemos definido:

- Pon un punto de ruptura en la línea que imprime las variables e inicia la depuración con `F5`.
- Muestra la ventana de memoria y haz que muestre el valor de la variable `c1` en hexadecimal. ¿Cuál es? Responde en el cuestionario: pregunta 3. Como puedes comprobar si buscas en Internet la tabla de códigos ASCII (http://es.wikipedia.org/wiki/ASCII) ese valor hexadecimal se corresponde con la codificación ASCII de la letra `a`. Como es una letra del alfabeto inglés y forma parte del ASCII, también tendrá esa misma codificación en todas las extensiones del ASCII, incluyendo ISO-Latin-1, y en UTF-8.
- Muestra en la ventana de memoria el valor de la variable `c2` en hexadecimal. ¿Cuál es? Responde en el cuestionario: pregunta 4. Este no puede ser un código ASCII porque los códigos ASCII sólo van hasta 127 (7Fh). En realidad es un código que se corresponde con una de las extensiones de ASCII. MS-DOS, el sistema operativo de Microsoft antecesor de Windows, tradicionalmente usaba en España una extensión llamada página de códigos 850 (http://es.wikipedia.org/wiki/Página_de_códigos_850). Sin embargo, en la actualidad las aplicaciones usan distintas extensiones según estén configuradas, siendo una de las más comunes en España la denominada Windows-1252 (http://es.wikipedia.org/wiki/Windows-1252). ¿Cuál de estas dos extensiones asigna a la `ñ` el código que se encuentra almacenado en memoria? Responde en el cuestionario: pregunta 5.

Esa es la versión que está utilizando el editor de Visual Studio. Sin embargo, la consola que se muestra cuando ejecutas el programa está utilizando la otra extensión y, por esta razón, el carácter mostrado no se corresponde con la `ñ`, sino con el carácter que se corresponde con el número `F1h` en esa extensión.

Vamos a comprobar que es así:

- Finaliza la depuración.
- Modifica la sentencia de asignación de `c2` para que se le asigne el código hexadecimal correspondiente a la `ñ` en la extensión que utiliza la consola. ¿Cuál es? Responde en el cuestionario: pregunta 6. Para asignar directamente un código hexadecimal puedes poner su valor en hexadecimal antecediéndolo de 0x. Por ejemplo, si quieres asignar el valor hexadecimal `5a` a la variable `var` tendrías que poner `var = 0x5a`.

- Ejecuta el programa y comprueba que ahora se imprime la `ñ` correctamente.

4. Ejercicios adicionales

- ✓ Es importante darse cuenta de que los errores con números reales que se han observado en esta práctica ocurrirán en la mayoría de los programas que trabajen con datos numéricos. A modo de ejemplo, abre una hoja Excel y añade la siguiente información a sus celdas:
  - En la celda `A1` escribe 1,324.
  - En la celda `A2` escribe 1,319.
  - En la celda `A3` calcula la diferencia entre `A1` y `A2` de la siguiente forma: =A1-A2.
  - En la celda `A4` comprueba que la diferencia es 0.005 de la siguiente forma: =SI(A3=0,005;"Iguales";"Diferentes").
  - Interpreta el resultado. Deberías ser capaz de identificar el problema.

  ¿Se podría utilizar una variable de tipo `char` para almacenar cualquier carácter codificado con UTF-8? ¿Por qué? Responde en el cuestionario: pregunta 7.

Objetivos

En esta sesión se pretende introducir al alumno en la utilización de la herramienta *Digital* para la simulación de circuitos digitales. Esta herramienta permite diseñar un circuito digital y simular su comportamiento, introduciendo diferentes combinaciones en sus entradas y comprobando que sus salidas toman los valores esperados.

Para ilustrar el funcionamiento del simulador se implementará un detector de *overflow* para la suma de enteros en complemento a 2.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Conocer la función lógica desempeñada por un detector de *overflow*.
- Comprender el concepto de tabla de verdad de un circuito y ser capaz de escribir la tabla de verdad del detector de *overflow*.
- Llevar a clase una memoria USB, o dispositivo análogo, para almacenar los circuitos que se desarrollarán en las práctica.

Durante la sesión se plantearán una serie de preguntas que deben responderse en el correspondiente cuestionario del Campus Virtual. El cuestionario se puede abrir en otra pestaña del navegador pinchando en el enlace mientras se mantiene pulsada la tecla `Ctrl`.

1. Introducción al simulador de circuitos digitales

En las clases de laboratorio de sistemas digitales se empleará el simulador denominado *Digital*. Se trata de un programa desarrollado en Java y publicado como proyecto de software libre con licencia GPL v3.0. Se puede obtener la última versión en https://github.com/hneemann/Digital. Es recomendable que todos empleemos la misma versión, por lo que tu profesor de laboratorio te indicará dónde está ubicada la versión de *Digital* que debes descargar. En cualquier caso, para poder ejecutar el programa hace falta tener instalada la máquina virtual de Java.

Este simulador no es un producto orientado al mercado profesional, sino un simulador sencillo orientado a la docencia de sistemas digitales. No obstante, a pesar de su simplicidad tiene la potencia y flexibilidad necesarias para simular un computador educacional completo.

> - En primer lugar descarga el simulador desde la ubicación que te indicará el profesor. Se trata de un archivo de nombre `Digital.zip` que debes descomprimir. Una vez descomprimido verás entre otros un archivo `Digital.jar` que debes ejecutar haciendo doble click sobre él.

La primera vez que se ejecuta el simulador aparece un tutorial que debería seguirse para adquirir los principios de funcionamiento de la herramienta. Sin embargo, dependiendo de la versión del simulador, las puertas lógicas se representan usando cajas rectangulares en lugar de los símbolos habituales. Para conseguir la representación habitual deben seguirse estos pasos:

- Cancela el tutorial.
- En la barra de menú debes seleccionar **Editar › Ajustes**, a continuación en la pestaña **Básico** activa la casilla **Usa las formas IEEE 91-1984**.
- Aparece una ventana indicando que debe reiniciarse el programa; proceso que debes hacer manualmente cerrando y abriendo de nuevo el programa.

Una vez arranca de nuevo el programa ya no aparece la posibilidad de seguir el tutorial. Sin embargo se puede arrancar el tutorial.

- Selecciona la opción **Ver › Start Tutorial** para arrancar el tutorial.

> **Es imprescindible seguir el tutorial antes de continuar con la práctica.**

Durante la simulación guiada por el tutorial habrás observado que el color verde oscuro se emplea para representar un cero lógico, mientras que el color verde claro se emplea para el uno lógico. Estos colores no pueden cambiarse en la versión actual del simulador.

Si en algún momento tienes alguna duda sobre los diferentes componentes u opciones del programa, puedes dejar el puntero del ratón sobre el elemento correspondiente, aparecerá una descripción emergente (*tooltip*).

A continuación se proporcionan atajos útiles en el uso del simulador.

- Para seleccionar un componente del área de dibujo debe hacerse click con el botón izquierdo del ratón sobre el mismo. Si lo que se desea seleccionar es un cable, además hay que pulsar la tecla `Ctrl`.
- Se puede seleccionar una parte del circuito manteniendo pulsado el botón izquierdo del ratón, arrastrando y soltando. Una vez seleccionado, aparte de moverlo haciendo click sobre él de nuevo y arrastrando, se puede borrar `Supr`, copiar `Ctrl`+`C` y pegar `Ctrl`+`V`.
- Para configurar un componente situado en el circuito debe hacerse click sobre él con el botón derecho del ratón. Aparecerá un menú contextual.
- La tecla `Esc` puede usarse para cancelar operaciones.
- Para deshacer cualquier operación de edición puede pulsarse la combinación de teclas `Ctrl`+`Z` y para rehacer la combinación `Ctrl`+`Y`.
- Para rotar un componente seleccionado, o un área seleccionada, debe pulsarse la tecla `R`.
- El nivel de *zoom* puede modificarse pulsando la combinación teclas `Ctrl+` y `Ctrl-`, así como empleando la tecla `Ctrl` y la rueda de desplazamiento del ratón. También se puede ajustar el nivel de *zoom* al tamaño del circuito con `F1`.
- Para colocar en el circuito una copia del último componente empleado puede pulsarse la tecla `L`.

2. Diseño de un detector de *overflow*

Una vez conocemos el funcionamiento básico del simulador es momento de seguir avanzando y construir circuitos más complejos que el del tutorial.

El siguiente objetivo es construir un detector de *overflow* para las operaciones de suma de enteros. Se trata de un sistema digital con una única salida que se pone a uno cuando el signo de la suma no es coherente con el signo de los sumandos.

> - ¿En qué dos casos se produce *overflow*? Responde en el cuestionario: pregunta 1.

La figura 1 muestra una posible implementación del detector de *overflow* dentro del simulador *Digital*.

Figura 1. Implementación del detector de overflow

Como puedes observar, las puertas AND tienen tres entradas en lugar de las dos habituales y además tienen unos círculos en algunas de sus entradas. Estos círculos indican que la puerta emplea esas entradas negadas y son equivalentes a poner puertas NOT en dichas entradas.

> - Abre un nuevo circuito. Puedes emplear la combinación de teclas `Ctrl`+`N` o seleccionar **Archivo › Nuevo**. El ciruito del tutorial puedes descartarlo.
> - Coloca sobre el circuito todas las puertas lógicas, así como las entradas y salidas, tomando como referencia la figura 1. Las puertas AND tienen dos entradas no negadas, pero no hay problema, pues las configuraremos una vez colocadas en el circuito.
> - Configura las dos puertas AND con tres entradas y las entradas negadas apropiadas.
>
> 1. Pulsa sobre la puerta con el botón derecho del ratón, selecciona la pestaña **Básico** y a continuación selecciona 3 empleando el selector etiquetado como **Número de entradas**.
> 2. Pulsa el botón a la derecha de **Salidas invertidas** y activa las casillas de las entradas que se debes invertir. La figura 2 muestra la configuración de la puerta AND superior.
>
> *Figura 2. Configuración del número de entradas y su negación en una puerta*
>
> - Traza los cables necesarios entre todas las puertas, entradas y la salida tomando como referencia el circuito de la figura 1. Observa cómo cuando un cable comienza o termina en otro cable el simulador pone automáticamente un punto de conexión en la unión.

Las entradas y salidas del circuito están etiquetadas para su identificación. El etiquetado se consigue configurando los componentes correspondientes, pulsando sobre ellos con el botón derecho del ratón, como se muestra en la figura 3. En este caso se ha etiquetado la entrada A_{n-1}, usando el carácter `_` para crear un subíndice. Además del etiquetado se ha introducido una descripción de la entrada que aparecerá como un *tooltip* cuando se ponga sobre ella el cursor del ratón.

Figura 3. Etiquetado de entradas y salidas

- Etiqueta todas las entradas y salidas del circuito para que coincidan con las de la figura 1.

Para completar el dibujo del circuito mostrado en la figura 1 vamos a colocar la cadena `Detector de overflow para la suma`.

- Selecciona el componente **Texto** a través del menú **Componentes › Entrada-Salida**. En el apartado de descripción debes introducir el texto requerido. El componente **Texto** es un componente meramente descriptivo que no influye en la simulación.

Una vez hemos completado el circuito debes guardarlo.

- Guarda el archivo en tu directorio de trabajo con el nombre `overflow`, seleccionando **Archivo › Guardar**, o pulsando la combinación de teclas `Ctrl` + `S`.
- Observa desde el explorador de archivos el circuito guardado, verás que tiene la extensión `.dig`.

Vamos a proceder ahora a la simulación.

- Arranca la simulación pulsando la barra espaciadora, o haciendo click sobre el icono de la barra de herramientas. El estado inicial de todas las entradas es cero, pues es el estado por defecto de las mismas cuando se arranca la simulación. Además, cuando todas las entradas están a cero en este circuito, la salida y todos los cables están también a cero. Recuerda que el estado cero se representa con un color verde oscuro.
- Para cambiar una entrada de cero a uno o viceversa, basta con hacer click sobre ella. Prueba a poner las entradas A_{n-1} y B_{n-1} a uno, se pondrán en color verde claro, y observa cómo la salida de *overflow* se pone a uno. Esto es lógico, pues al estar A_{n-1} y B_{n-1} a uno y S_{n-1} a cero, se están sumando dos enteros negativos y el resultado es un entero positivo.
- Prueba todas las combinaciones de entradas y observa que el estado de la salida es el esperado.
- Finaliza la simulación pulsando de nuevo la barra espaciadora, o haciendo click sobre el icono de la barra de herramientas.

3. Comprobación automática de circuitos

Una de las características interesantes del simulador *Digital* es la posibilidad de comprobar de forma automática el correcto funcionamiento del circuito, esto es, sin tener que cambiar a mano durante la simulación el estado de las entradas, como se hizo en el apartado anterior.

- Coloca en el circuito el componente **Caso de prueba**, al cual puede accederse a través del menú **Componentes › Varios**. La ubicación del componente en el circuito es indiferente, pues no se conecta a otros componentes. Una vez situado en el circuito aparece un componente de nombre `Test` en su interior.
- Configura el componente `Test` haciendo click sobre él con el botón derecho del ratón y pulsando a continuación sobre el botón `Editar`. La pantalla de la aplicación será similar a la mostrada en la figura 4

Figura 4. Componente para la comprobación del circuito

En la figura anterior se ha mostrado además como ejemplo el caso de prueba para la combinación de entradas $A_{n-1} = 0$, $B_{n-1} = 0$ y $S_{n-1} = 0$ que debe dar Ov = 0.

> Para que el componente `Test` funcione correctamente, los nombres de las entradas y salidas de la primera línea deben coincidir con las etiquetas de las entradas y salidas en el circuito.

- Debes completar el dispositivo `Test` con las combinaciones de entradas restantes, empleando una línea adicional para cada una de ellas. Una vez completado, debes guardar su estado pulsando OK en las dos ventanas modales abiertas.
- Para realizar la comprobación del circuito puedes pulsar F8 o hacer click sobre el icono de la barra de herramientas. Aquellas combinaciones de entradas para las cuales las salidas son correctas son marcadas en verde en una ventana emergente.

Vamos a comprobar lo que ocurre cuando el resultado de algún *test* no es correcto.

- Cierra la venta emergente anterior, prueba a hacer algún cambio en los datos de la prueba y repite la prueba. Aquellas combinaciones cuyas salidas no coinciden en la tabla y en la simulación aparecen marcadas en rojo.
- Devuelve los valores correctos al componente de prueba.
- Guarda de nuevo el circuito para almacenar el circuito con el componente `Test`.

4. Creación de componentes

Una de las características imprescindibles en cualquier simulador digital es la capacidad de crear nuevos componentes a partir de circuitos. De esta forma, a partir de puertas lógicas u otros componentes simples se pueden crear de forma jerárquica componentes cada vez más complejos, hasta llegar por ejemplo a implementar un computador completo.

Para ejemplificar la capacidad del simulador para crear componentes a partir de circuitos se procederá a crear un componente nuevo, el detector de *overflow* para la suma.

- Abre un nuevo archivo en blanco seleccionando **Archivo › Nuevo**, o pulsando la combinación de teclas `Ctrl` + `N`. Para que este nuevo circuito pueda emplear el circuito `overflow.dig` anterior como componente debe estar en el mismo directorio, o en un directorio que lo contenga. Por esta razón guardaremos el nuevo circuito en el mismo directorio.
  Selecciona **Archivo › Guardar como** y guárdalo con el nombre `new-overflow.dig`.
- Selecciona en el menú la opción **Componentes › Personalizado** y elige el componente *overflow*. Observarás que aparece el componente *overflow* que construiste previamente.
- Sitúa el componente *overflow* en el área de dibujo del simulador. La disposición de las entradas y salidas, así como las dimensiones del componente son generadas por defecto por el simulador y son las mostradas en la figura 5 izquierda.

Figura 5. Diferentes disposiciones de entradas y salidas en el componente

- Prueba a poner el puntero del ratón sobre el componente que acabas de situar en el circuito y espera un instante. Observarás que aparece la descripción del componente que introdujiste durante su creación.
- Prueba ahora a poner el puntero del ratón sobre alguna de las entradas o la salida del componente. Observarás que aparece la descripción introducida durante el etiquetado.

Sin embargo, nuestro objetivo es que la disposición y forma sea similar a la que aparece en la misma figura a la derecha. Para conseguirlo debes realizar algunos cambios adicionales en el circuito del detector de *overflow*.

- Abre de nuevo el circuito haciendo click con el botón derecho del ratón sobre el componente y pulsando el botón `Abrir circuito`. Aparecerá una nueva ventana del simulador en la cual podemos modificar el circuito del detector de *overflow*.
- Modifica el circuito para que la disposición de las entradas sea como la que se muestra en la figura 6.

Figura 6. Detector de overflow con las entradas cambiadas

- Guarda el circuito del detector de *overflow* con el mismo nombre, pero no lo cierres, y vuelve al circuito con el componente, verás que nada ha cambiado.

Independientemente de la posición de las entradas y salidas en el circuito `overflow`, la disposición en el componente no cambia cuando el circuito está configurado con su forma por defecto, por lo que hay que cambiar de forma.

- Vuelve a la ventana del circuito del detector de *overflow*, sin cerrar la ventana actual, y selecciona **Editar › Ajustes específicos del circuito**. En la pestaña **Avanzado** selecciona la opción **Diseño para Forma**.

- Guarda el circuito para que los cambios tengan efecto. Si vuelves de nuevo a la ventana del circuito con el componente verás que la ubicación de las entradas ha cambiado y son las mostradas en la figura 5 central. Lo que ha ocurrido es que las entradas y salidas se sitúan ahora en el componente en el mismo orden y orientación que aparecen en el circuito.

Para finalizar el componente aún faltan un par de ajustes.

- En primer lugar vamos a ampliar el ancho mínimo de la caja para que pase de 3 a 5 unidades de rejilla. De nuevo hay que volver a la ventana del circuito del detector de *overflow* y seleccionar **Editar › Ajustes específicos del circuito**. En la pestaña **Básico** seleccionamos 5 en **Anchura**. Aprovecharemos además para introducir una descripción que aparecerá en forma de descripción emergente cuando pongamos el puntero del ratón sobre el componente. Estos ajustes se muestran en la figura 7.

Figura 7. Configuración del ancho y descripción emergente del componente

Si además del ancho mínimo se quisiese configurar el alto mínimo de la caja de componente podría hacerse dentro de la pestaña **Avanzado**.

- Guarda de nuevo el circuito del detector de *overflow* y ciérralo.
- Observa en el circuito `new-overflow` cómo la caja del detector de *overflow* coincide con la de la figura 5 derecha.

Ahora vamos a trabajar exclusivamente en el circuito `new-overflow` con el componente.

- Conecta las entradas y salidas como se indican en la figura 8.

Figura 8. Ejemplo de uso del componente overflow

- A continuación veremos la posibilidad de copiar de un circuito a otro.
  1. Abre el circuito `overflow` haciendo click con el botón derecho del ratón sobre el componente y pulsando el botón `Abrir circuito`.
  2. Selecciona el componente `Test` haciendo click sobre él, pulsa `Ctrl`+`C` y cierra el circuito `overflow`.
  3. Ya en el circuito `new-overflow` pulsa `Ctrl`+`V`.
  4. Ejecuta de nuevo la comprobación del circuito pulsando `F8` o .

> Para que el componente `Test` funcione en el nuevo circuito, las etiquetas de las entradas y salidas deben ser las mismas que en el circuito `overflow`.

5. Generación de cronogramas

Una de las características más interesantes de este simulador es su capacidad de generar cronogramas con las entradas y salidas durante la simulación.

- Añade al circuito el componente `Gráfica de datos`, seleccionado **Componentes › Entrada-Salida** y siguiendo la opción **Más**. Durante la simulación este componente mostrará un cronograma que se actualizará con cada cambio en alguna de las entradas o salidas del circuito.

Antes de probar la generación de cronogramas vamos a ordenar las entradas y salidas dentro del cronograma.

- Selecciona **Editar › Ordenar los valores de las medidas**. Aparecerá una ventana en la cual debes situar las entradas y salidas usando las flechas en el orden indicado en la figura 9.

Figura 9. Orden de las entradas y salidas en el cronograma

- Pulsa OK y guarda el circuito.

Ya está todo dispuesto para comenzar la simulación.

- Arranca la simulación pulsando la barra espaciadora o el icono de la barra de herramientas. Cada vez que cambies una entrada observarás cómo evoluciona el cronograma. En la figura 10 se muestra la evolución tras probar las dos combinaciones de entradas que producen *overflow*.

Figura 10. Cronograma durante la simulación

6. Ejercicios adicionales

Estos ejercicios adicionales permiten al alumno reforzar los conocimientos adquiridos durante la sesión práctica.

- En ocasiones resulta útil visualizar en los cronogramas no sólo las entradas y salidas de un circuito, sino además puntos intermedios dentro de un circuito o de uno de sus componentes. El simulador proporciona el componente **Sonda** que se encuentra en **Componentes › Entrada-Salida**. Guarda el circuito `new-overflow` como `sonda-overflow` y sobre este nuevo circuito conecta una puerta AND y una sonda a su salida, tal como se muestra en la figura 11. Se puede cambiar su nombre configurándola y cambiando su valor de etiqueta. Observa el efecto arrancando la simulación y modificando los valores de las entradas.

Figura 11. Conexión de sondas en circuitos

- Implementa en el simulador un multiplexor con dos canales de entrada. Comprueba su correcto funcionamiento añadiendo un componente `Test` que verifique todas las combinaciones de entradas.
- El simulador puede emplearse para comprobar cualquiera de los circuitos digitales de la asignatura, especialmente los que aparecen en los ejercicios. Por ejemplo, hay ejercicios en los que se pide el cronograma, cuyas soluciones se pueden comprobar usando el simulador **una vez resueltos**.

Objetivos

El objetivo de esta sesión práctica es comprender el funcionamiento de una ALU básica análoga a la empleada en el Computador Teórico (CT), un computador didáctico muy sencillo que se estudiará en la asignatura. Sobre esta ALU se llevarán a cabo las operaciones aritméticas de suma y resta, así como las operaciones lógicas AND, OR y XOR.

Como herramienta de prácticas se empleará el simulador de circuitos digitales introducido en la sesión anterior, el cual nos permitirá analizar la ALU no sólo como una "caja negra", sino a través del funcionamiento de sus componentes internos.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Tema 2 del libro *Fundamentos de Computadores y Redes*, *Sistemas Digitales*, en concreto el apartado 2.2.7, en el cual se describe el funcionamiento externo de la ALU. El alumno debe leer con detenimiento este apartado antes de asistir a la sesión práctica.
- Llevar a clase una memoria USB, o dispositivo análogo, para almacenar los circuitos que se desarrollarán en la práctica.

Durante la sesión se plantearán una serie de preguntas que deben responderse en el correspondiente cuestionario del Campus Virtual. El cuestionario se puede abrir en otra pestaña del navegador pinchando en el enlace mientras se mantiene pulsada la tecla `Ctrl`.

1. Introducción

La ALU es un componente fundamental del computador, pues es la unidad encargada de llevar a cabo las operaciones aritméticas y lógicas. Por ejemplo, cada vez que se suman dos números dentro del computador estos números se llevan como entradas a la ALU, la cual genera a la salida no sólo el resultado de la suma, sino además unos bits de estado denominados *flags* que proporcionan información sobre el resultado de la operación.

La figura 1 muestra las entradas y salidas de la ALU de 4 bits empleada en esta práctica. Como entradas recibe los operandos `A` y `B`, ambos de 4 bits, así como 4 bits que seleccionan la operación a realizar: `Op1`, `Op0`, `Cin` y `Compl-1`. Como salidas se genera el resultado de 4 bits, `S`, y 4 bits de estado: `ZF`, `CF`, `OF` y `SF`.

Figura 1. Entradas y salidas de una ALU de 4 bits

La tabla 1 indica la operación realizada en función del valor de los bits de operación.

Tabla 1. Tabla de operaciones de la ALU

Op1	Op0	Cin	Compl-1	Operación
0	0	0	0	`S = A AND B`

0	1	0	0	`S = A OR B`
1	0	0	0	`S = A XOR B`
1	1	0	0	`S = A + B`
1	1	1	0	`S = A + B + 1`
1	1	1	1	`S = A - B`

Para comprender el funcionamiento de la ALU se realizarán diferentes operaciones, siguiendo estos pasos:

- La operación planteada será realizada manualmente por el alumno, esto es, sin ayuda del simulador. Como resultado se obtendrán los 4 bits del resultado `S` y los 4 bits de estado: `ZF`, `CF`, `OF` y `SF`.
- Se realizará la operación en el simulador, para lo cual será necesario activar de forma adecuada las entradas con los bits necesarios. El resultado debería ser el mismo que el obtenido de forma manual.
- Con la ayuda del simulador se profundizará en el funcionamiento de la ALU observando el funcionamiento de sus componentes internos.

2. Funcionamiento de la ALU con operaciones de suma

En primer lugar, vamos a realizar la operación de suma con los operandos `A=0110` y `B=0111`.

> - Realiza a mano sobre el papel la operación de suma de los operandos `A` y `B`. ¿Qué valor toma `S`, el resultado de la suma? Responde en el cuestionario: pregunta 1.
> - ¿Qué valor deben tomar los bits de estado `ZF` y `SF`? ¿Por qué? Responde en el cuestionario: pregunta 2.
> - Interpretando los operandos `A`, `B` y el resultado `S` como números naturales, ¿el resultado de la suma es correcto o incorrecto? ¿Qué valor debe tomar entonces el bit `CF` de acarreo? Responde en el cuestionario: pregunta 3.
> - Interpretando ahora los operandos `A`, `B` y el resultado `S` como números enteros, ¿el resultado de la suma es correcto o incorrecto? ¿Por qué? ¿Qué valor debería tomar entonces el bit `OF` de *overflow*? Responde en el cuestionario: pregunta 4.

A continuación la operación de suma se realizará con la ayuda del simulador.

> - Descarga el archivo `alu4.zip` del Campus Virtual y descomprímelo en tu directorio de trabajo. Dentro del directorio `alu4` creado aparecen varios archivos. Uno de ellos es el archivo `alu4_layout.dig`, el cual debes abrir empleando el simulador de circuitos digitales. En el área de trabajo

observarás un componente de nombre `alu4` con sus entradas y salidas unidas a puntos de conexión, tal como se muestra en la figura 2.

Figura 2. La ALU de 4 bits en el simulador

- En el simulador, activa el modo simulación.
- Coloca en las entradas correspondientes los valores `1` y `0` adecuados para realizar la suma con los operandos `A=0110` y `B=0111`.
- Observa el resultado de la suma y el de los bits de estado. ¿Coinciden con los que obtuviste manualmente?
- Desactiva el modo simulación.
- Haz clic con el botón derecho del ratón sobre el componente `alu4`. En la ventana que aparece haz clic con el botón izquierdo en la opción **Abrir circuito**. El simulador muestra el interior de la ALU en el cual se observan sus componentes y los cables de conexión como se muestra en la figura 3.

Figura 3. Interior de la ALU de 4 bits

El resultado de la suma, `S`, es generado por el componente `alu4-nf`, el cual implementa la ALU de 4 bits sin *flags* (bits de estado). En la parte derecha aparecen 4 componentes que implementan los bits de estado. Debe tenerse en cuenta que el bit de estado `SF` se obtiene directamente del bit más significativo del resultado, pues es el que indica si el resultado es negativo (`SF`=1) o positivo (`SF`=0). Estos son los componentes que aparecen a la derecha:

- Una puerta NOR de 4 entradas. Esta puerta genera un uno a la salida sólo cuando todas sus entradas están a cero, es decir, cuando el resultado de la suma es `S=0000`. Este es justo el resultado esperado del bit de estado `ZF`.
- Una puerta XOR que genera el bit de acarreo `CF`. Durante las operaciones de suma la entrada de operación `compl-1` toma el valor 0, por lo que el bit de acarreo coincide con el proporcionado por la ALU de 4 bits sin *flags*, es decir, `CF = cout`. Si el bit `compl-1` fuese 1, como ocurre en el caso de las operaciones de resta, el bit de acarreo `CF` sería justo `cout` negado.
- Para la detección de *overflow* se emplea el generador de *overflow* para la suma, de nombre `gen_ov_s`, y la puerta XOR que está justo debajo. La función de la puerta XOR es invertir el bit más significativo del operando `B` cuando se hace una operación de resta, es decir, cuando `compl-1` es 1. En el caso de la suma no se invierte el bit de signo de `B`, por lo que `gen_ov_s` recibe como entradas el bit más significativo del operando `A`, el bit más significativo del operando `B` y el bit más significativo del resultado `S`. La salida `Ov` del generador de *overflow*, bit `OF`, debe reflejar la coherencia entre los signos de los operandos de entrada y el resultado. Es decir, indica si el resultado de la suma es correcto interpretando los operandos y el resultado como enteros en complemento a 2.

- En el nuevo circuito que has abierto, `alu4.dig`, activa el modo simulación.
- Vuelve a colocar en las entradas correspondientes los valores de `1` y `0` adecuados para realizar la suma con los operandos `A=0110` y `B=0111`.

- Observa el resultado y cómo se generan las salidas correspondientes a los bits de estado.
- En el caso del bit de estado `ZF`, ¿qué valores toman las entradas y salidas de la puerta NOR? Responde en el cuestionario: pregunta 5.
- En el caso del bit de estado `CF`, ¿qué valor toman los bits `cout` y `compl-1`? Responde en el cuestionario: pregunta 6.
- En el caso del bit de estado `OF`, ¿qué signos tienen `A`, `B` y `S`? ¿Es coherente el signo de `S` con el de los operandos `A` y `B`? Responde en el cuestionario: pregunta 7.

Vamos a profundizar un paso más en el diseño del circuito de la ALU.

- Desactiva el modo simulación en el circuito `alu4.dig`, si no lo has hecho ya.
- Haz clic con el botón derecho del ratón sobre el componente `alu4-nf`. En la ventana que aparece elige de nuevo la opción **Abrir circuito**. Aparecerá ahora un nuevo circuito que representa la estructura interna de la ALU de 4 bits sin bits de estado, como se muestra en la figura 4.
- En el nuevo circuito que has abierto, `alu4-nf.dig`, activa el modo simulación.
- Vuelve a colocar en las entradas correspondientes los valores de `1` y `0` adecuados para realizar la suma con los operandos `A=0110` y `B=0111`.
- Observa el resultado y cómo ahora solo estaría disponible el acarreo de salida de la ALU, `cout`.

Figura 4. Interior de la ALU de 4 bits sin flags

En el nuevo circuito, se observa cómo las operaciones que realiza la ALU se realizan bit a bit mediante circuitos más simples. Para entender el porqué de esta implementación, la figura 5 ilustra cómo la operación de suma anterior se puede llevar a cabo con 4 ALU de 1 bit, cada una de las cuales trabaja sobre una pareja de bits de `A` y `B`.

Figura 5. Ejemplo de suma de 4 bits

Así por ejemplo, la ALU de 1 bit menos significativa, la que está más a la derecha, realiza la operación con los bits A_0 y B_0, generando un resultado S_0 y un acarreo de salida `ci` que pasa a ser el acarreo de entrada `ci-1` de la ALU de 1 bit siguiente. El acarreo de salida de la ALU de 1 bit más significativa será el acarreo de salida `cout` de la ALU de 4 bits sin *flags*. De forma análoga, el acarreo de entrada de la ALU menos significativa es el acarreo de entrada `cin` de la ALU de 4 bits sin *flags*. Resulta conveniente reseñar que todas las ALU de 1 bit emplean los mismos bits de operación `Op1`, `Op0` y `compl-1`. Esta estructura es razonable si se piensa cómo se lleva a cabo por ejemplo una suma de 4 bits como la indicada en la figura 5.

- Desactiva el modo simulación en el circuito `alu4-nf.dig`, si no lo has hecho ya.
- Haz clic con el botón derecho del ratón sobre la ALU de 1 bit (componente `alu1`) más significativa, la que aparece más a la izquierda. En la ventana que aparece elige de nuevo la opción **Abrir circuito**. Aparecerá su estructura interna, como se muestra en figura 6.

Figura 6. Circuito interno de la ALU de 1 bit más significativa

El funcionamiento de la ALU de 1 bit es simple. Realiza 4 operaciones a la vez sobre los bits A_i y B_i, los cuales en nuestro caso coinciden con A_3 y B_3. Las operaciones son AND, OR, XOR y SUMA. Los resultados de las 4 operaciones llegan a un multiplexor de 4 canales de entrada. Con las líneas `Op1` y `Op0` se selecciona la operación deseada. Por ejemplo, para seleccionar la suma hay que poner en estas líneas la combinación `11` pues el resultado del sumador de 1 bit llega a la entrada e_3 del multiplexor.

Durante las operaciones de suma la entrada de operación `compl-1` toma el valor 0, por lo que a la entrada B_i del sumador llega B_3 (en caso contrario llegaría B_3 negada, como ocurre en las operaciones de resta).

El funcionamiento interno de los multiplexores, `mux2` y `mux4`, y del sumador de 1 bit `sum1` es conocido, por lo que no se profundizará dentro de ellos.

3. Funcionamiento de la ALU con operaciones de resta

En este apartado se realizará la operación de resta con los mismos operandos anteriores, `A=0110` y `B=0111`.

- Realiza a mano sobre el papel la operación de resta de los operandos `A` y `B`. Recuerda que para llevar a cabo la resta debes sumarle al operando `A` el complemento a 2 del operando `B`, tal como puedes observar en la figura 7. ¿Qué valor toma `S`, el resultado de la resta? Responde en el cuestionario: pregunta 8.

Figura 7. Ejemplo de resta de 4 bits realizada "manualmente"

- ¿Qué valor deben tomar los bits de estado `ZF` y `SF`? ¿Por qué? Responde en el cuestionario: pregunta 9.
- Interpretando los operandos `A` y `B` como números naturales, ¿cuáles serían respectivamente sus valores? ¿es posible llevar a cabo la operación de resta? ¿Por qué? Responde en el cuestionario: pregunta 10.
- Al sumar al operando `A` el complemento a 2 del operando `B`, ¿se ha producido acarreo? ¿cuál es el valor del resultado `S` interpretado como número natural? Responde en el cuestionario: pregunta 11. Cuando no se produce acarreo al hacer esta operación con el complemento a 2, la operación de resta original no es posible entre los operandos interpretados

como naturales. En caso contrario, si se genera acarreo en la suma con el complemento a 2, sí es posible la operación de resta original.

- Interpretando ahora los operandos `A` y `B` como números enteros, ¿cuáles serían respectivamente sus valores? ¿cuál es el valor del resultado `S` interpretado como número entero? ¿el resultado de la resta es correcto o incorrecto? ¿Por qué? ¿Qué valor debería tomar entonces el bit `OF` de overflow? Responde en el cuestionario: pregunta 12.

A continuación la operación de resta se realizará con la ayuda del simulador.

- Abre con el simulador de circuitos digitales el archivo `alu4.dig`, si no lo tienes abierto ya.
- Activa el modo de simulación. Coloca en las entradas correspondientes los valores de `1` y `0` adecuados para realizar la resta con los operandos `A=0110` y `B=0111`.
- ¿Qué valor toma el bit de acarreo a la salida de la ALU de 4 bits sin *flags*? Responde en el cuestionario: pregunta 13. ¿Qué valor toma el bit de acarreo a la salida de la ALU de 4 bits completa? Responde en el cuestionario: pregunta 14. Observa cómo en el caso de resta (`compl-1` es 1) el valor del bit de acarreo se invierte para indicar si el minuendo es menor que el sustraendo.
- Observa el resultado de la resta y el de los bits de estado. ¿Coinciden con los que obtuviste manualmente al sumar `A` más el complemento a 2 de `B`? Deberían coincidir todos excepto el bit de acarreo. La razón es que en el caso de resta la ALU genera el bit de acarreo para indicar si el minuendo es menor que el sustraendo, interpretados minuendo y sustraendo como naturales. Cuando el minuendo es menor que el sustraendo la ALU pone el bit de estado `CF` a 1 indicando un error en la resta de naturales. Cuando la ALU genera un bit de estado `CF` a 0 esto indica que la operación interpretada entre naturales es correcta; si se trata de suma el resultado está dentro del rango y si se trata de resta el minuendo es mayor que el sustraendo.
- Desactiva la simulación.

Vamos a repetir ahora la operación a nivel de los componentes internos de la ALU.

- Pulsa con el botón derecho sobre el componente `alu4-nf`, elige la opción **Abrir circuito**. El simulador muestra el interior de la ALU en el cual se observan sus cuatro ALU elementales y los cables de conexión.

El circuito digital de la ALU realiza la resta de forma diferente a como la hemos hecho sobre el «papel». La ALU para realizar la resta, suma el operando `A` con el negado del operando `B` (~`B`), que también se denomina complemento a 1 del operando `B`. Esta suma se realiza bit a bit en cada ALU elemental. Para completar la operación de resta, se suma `1` al bit memos significativo de los operandos. Esto equivale a activar la señal `ci-1` de la ALU elemental menos significativa, es decir, la de más a la derecha. El proceso puede verse

esquematizado en la figura 8. Como puedes deducir, el complemento a 2 que realizamos en la resta sobre el «papel» equivale a realizar el complemento a 1 + 1.

Figura 8. Ejemplo de resta de 4 bits realizada por la ALU

Vamos a centrarnos ahora en lo que sucede en la ALU elemental menos significativa.

- En el circuito `alu4-nf.dig`, observa cómo la línea de entrada `cin` entra solamente en el componente `alu1` situado más a la derecha, la menos significativa. En cambio las otras tres señales de control (`Op1`, `Op0` y `compl-1`) llegan a todos los componentes `alu1`.
- Pulsa con el botón derecho sobre el componente `alu1` situado más a la derecha en el circuito `alu4-nf`. Esta ALU es la que opera con los bits A_0 y B_0 de los operandos. Elige la opción **Abrir circuito**.
- Activa el modo de simulación. Coloca en las entradas A_i y B_i los valores correspondientes a los bits A_0 y B_0 de los operandos `A` y `B`. Coloca en las entradas de control los valores necesarios para ordenar la operación de resta: `Op1` = `1`, `Op0` = `1`, `compl-1` = `1` y `ci-1` = `1`.
- ¿Qué operación se está seleccionada dentro de la ALU de 1 bit? Responde en el cuestionario: pregunta 15.
- Observa lo que sucede. La señal `compl-1` llega al multiplexor de dos entradas haciendo que en lugar de tomar el valor del operando B_i tome el de su negado ~ (B_i). Es decir, la señal `compl-1` indica a las ALU elementales que obtengan el complemento a 1 del operando `B`.
- Observa que la señal `ci-1`, cuyo valor es `1`, entra al sumador de la ALU elemental menos significativa. Se sumará de esta forma `1` al valor del complemento a 1 del operando `B`. De esta forma la ALU realiza el complemento a 2 del número para realizar la resta.
- Desactiva la simulación y cierra el circuito `alu1.dig`.

En esta sesión se ha ilustrado el funcionamiento de la ALU con 4 bits. Como habrás podido observar, el funcionamiento es perfectamente escalable a cualquier número de bits encadenando componentes del tipo `alu1`. Cada componente `alu1` o ALU elemental opera con un bit de cada operando (A_i y B_i), teniendo en cuenta el valor del acarreo previo (`ci-1`) y produciendo un resultado (S_i) y un acarreo de salida (`ci`).

4. Ejercicios adicionales

- Realiza las operaciones recogidas en la tabla 2 «en papel», es decir, sin utilizar el simulador. En el caso de las operaciones lógicas el valor de los bits de estado *carry* y *overflow*, aunque la ALU los genera, no son significativos y deben ignorarse.

Tabla 2. Operaciones propuestas

A	B	Operación	Resultado	ZF	CF	OF	SF

0010	1100	OR					
0010	1100	AND					
0010	1100	XOR					
0010	1100	Suma					
0010	1100	Resta					

- Verifica que tus respuestas son correctas comparándolas con el resultado que ofrece el simulador.

- Elige, sobre el papel y de forma razonada, dos operandos cuya suma genere acarreo; calcula su suma y los bits de estado. A continuación, mediante simulación comprueba que el resultado de su suma y los bits de estado (*carry* y *overflow*) son iguales a los obtenidos sobre el papel.
- Busca dos operandos enteros positivos cuya suma genere acarreo. ¿Existen tales operandos? ¿Por qué?
- Busca dos operandos enteros negativos cuya suma no genere acarreo. ¿Existen tales operandos? ¿Por qué?
- Elige, sobre el papel y de forma razonada, dos operandos enteros positivos cuya resta dé un valor negativo. Calcula su resta y los bits de estado. A continuación, mediante simulación comprueba que el resultado de su suma y los bits de estado (*carry* y *overflow*) son iguales a los obtenidos sobre el papel.
- Busca dos operandos enteros positivos cuya resta genere *overflow*. ¿Existen tales operandos? ¿Por qué?

Objetivos

El objetivo de esta sesión práctica es la simulación por parte del alumno de operaciones aritméticas y lógicas dentro de la CPU del Computador Teórico (CT). Para no complicar en exceso la simulación se considera una CPU de sólo 4 bits, pero totalmente análoga a la del CT. Además, sólo se consideran los siguientes elementos de la CPU:

- Bus interno.
- ALU.
- Registro temporal de entrada.
- Registro temporal de salida.
- Registro de estado.
- Dos registros de propósito general.

Después de realizar la sesión práctica el alumno debería ser capaz de definir la secuencia de señales necesaria para realizar cualquier operación aritmética o lógica empleando la CPU del CT.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Comprender el funcionamiento de los biestables D, los *buffers* triestado, los registros y la ALU del CT.
- Llevar a clase una memoria USB, o dispositivo análogo, para almacenar los circuitos que se desarrollarán en la práctica.

Durante la sesión se plantearán una serie de preguntas que deben responderse en el correspondiente cuestionario del Campus Virtual. El cuestionario se puede abrir en otra pestaña del navegador pinchando en el enlace mientras se mantiene pulsada la tecla `Ctrl`.

1. Introducción

En esta sesión se pretende simular lo que se denomina «el camino de datos» de la CPU del CT, cuyo circuito se muestra en la figura 1. El camino de datos está compuesto por una serie de elementos que permitirán almacenar datos en unos registros denominados «registros de propósito general» y hacer operaciones aritméticas y lógicas con los datos utilizando la ALU. Para conectar los registros con la ALU se utilizará un bus, denominado «bus interno» (`IB`, de *Internal Bus*). Un bus es un conjunto de líneas que funcionan de forma coordinada. En nuestro caso, serán cuatro líneas ya que tanto los registros como la ALU serán de 4 bits. Como la ALU tiene dos entradas y no se pueden poner dos valores a la vez en un bus, será necesario introducir un registro temporal de entrada (`TMPE`) para almacenar uno de los datos. También será necesario un registro temporal de salida (`TMPS`) para almacenar el resultado de la ALU sin sobrescribir uno de los valores de entrada que se están utilizando. El último elemento que utilizaremos será otro registro, denominado «registro de estado» (`SR`, de *Status Register*) para almacenar los valores de los bits de estado (*flags*) calculados por la ALU.

Figura 1. Camino de datos del CT

Fíjate que, en este circuito, las entradas conectadas a las entradas de control de los registros no se han llamado `Read` y `Write`, sino con unos nombres que indican cómo se mueve la información al activar esa

entrada. Por ejemplo, la señal Read del registro `R0`, como vuelca su valor al `IB`, se ha llamado `R0-IB` y la señal `Write` de este registro, como escribe en el registro el valor que esté en `IB`, se ha llamado `IB-R0`. ¿Cómo se llama la entrada que hace que se almacene en el registro `TMPS` el valor generado por la ALU? Responde en el cuestionario: pregunta 1

Como verás en el tema 5, estas entradas son señales de control que no son generadas por un humano, sino por otro circuito, denominado «Unidad de Control».

2. Registros

Para almacenar información dentro de la CPU se utilizan registros, pero no todos son iguales:

- Los registros de propósito general (`R0` y `R1`) tienen líneas de datos d_i que actúan tanto de entrada como de salida y están conectadas al bus interno. Se utilizarán registros como los vistos en teoría.
- Los registros `SR` y `TMPS`, en cambio, tienen entradas y salidas separadas: las entradas están conectadas a salidas de la ALU (las de `SR` a los bits de estado y las de `TMPS` a las salidas de la operación) y sus salidas conectadas al bus interno.
- El registro `TMPE` tiene sus entradas y salidas separadas (las entradas conectadas al bus interno y las salidas a la entrada `A` de la ALU) pero, además, tiene la particularidad de que no hay otro elemento conectado a sus salidas (el resto de registros están conectados a un bus con otros registros), con lo que no necesita un *buffer* triestado en sus salidas.

En estas prácticas vamos a crear los registros de propósito general como los estudiados en teoría; los circuitos del resto de registros te serán proporcionados.

> - Descarga el archivo `datapath.zip` y descomprímelo a tu carpeta de trabajo.
> - Abre el simulador de circuitos digitales y crea un registro como el mostrado en la figura 2. Fíjate que usa biestables D (que puedes insertar a través de la opción **Componentes › Flip-Flops**) y *buffers* triestado (denominados «Drivers» en la opción **Componentes › Cables**). Como las entradas d_i tienen que actuar de entrada o salida en función de si la señal `Read` o `Write` está activa, tendrás que modificar sus opciones y en la solapa `Avanzado` seleccionar, tal y como se muestra en la figura 3, «Es una entrada de tres estados» y poner como valor por defecto «Z», que representa un estado de alta impedancia equivalente a no conectar ese elemento.
> - Cambia el orden de las entradas (**Editar › Ordenar las entradas**) para que sea `WRITE`, `READ`, d_3, d_2, d_1 y d_0.
> - Guarda el circuito con el nombre `reg4-tri.dig` en la carpeta donde has descomprimido `datapath.zip`.

Figura 2. Registro que usa líneas de datos que funcionan como entrada y salida.

Figura 3. Opciones para las entradas tri-estado.

3. Camino de datos

- Carga en el simulador el proyecto `datapath.dig`.
- Complétalo añadiendo los componentes necesarios. Fíjate en que, aparte del registro que acabas de crear (`reg4-tri.dig`), hay otros dos tipos de registros: `reg4_e_s.dig`, que tiene entradas y salidas separadas, y `reg4.dig`, que además de tener entradas y salidas separadas, no utiliza un *buffer* triestado. Debes seleccionar el adecuado para cada registro. Además, es posible que debas rotar alguno de estos registros.

Ahora vamos a hacer una operación de carga (escritura) de un registro y vamos a comprobar que almacena correctamente su valor. En concreto, vamos a almacenar el valor 5 en el registro de propósito general superior (`R0`):

- Lanza la simulación. Fíjate en que algunos cables están en gris: si sitúas el ratón sobre uno de ellos, podrás comprobar que se corresponde con el estado de alta impedancia (Z). Las entradas a las que están conectados estos cables pueden ahora cambiarse entre tres estados: 0, 1 y Z.
- Sitúa el valor 5, codificado en binario natural con cuatro bits, en las entradas IB_3, IB_2, IB_1 e IB_0.
- Cambia el valor de la entrada IB-R0 de 0 a 1 para que se produzca un flanco de subida en los biestables D del registro `R0`. Eso hará que este registro tome el valor en sus líneas de datos y lo almacene en sus biestables.
- Pon a cero la entrada `IB-R0` y en alta impedancia (color gris) las entradas IB_3, IB_2, IB_1 e IB_0 del bus interno. El bus en estos momentos no tiene ningún valor.
- Vamos a volcar el valor almacenado en `R0` al bus, es decir, vamos a leer el registro. Para ello, pon a 1 la entrada `R0-IB`, que está conectada a la entrada `Read` del registro R0, que a su vez está conectada a sus *buffers* triestado, con lo que estos van a llevar la salida de sus biestables a las líneas `d`. Deberías ver en las líneas del bus interno el valor 5 que habías almacenado en `R0`.
- Detén la simulación.

Fíjate que en este tipo de operaciones con circuitos digitales **síncronos**, el orden de las operaciones es muy importante. Si cometes cualquier error, tienes que empezar a repetir todos los pasos desde el principio.

A continuación vamos a hacer una operación de resta:

- Analiza cómo se haría la resta de los operandos A=5 y B=4 en papel, tal y como se muestra en la figura 4.

Figura 4. Resta de 5 - 4.

Seguidamente, vamos a simular esta operación con la herramienta. En este tipo de sesiones prácticas en las cuales se simulan circuitos digitales, es fundamental que trates de predecir el resultado de la simulación **antes** de llevarla

a cabo. A continuación, debes comparar tu predicción con el resultado proporcionado por la simulación y extraer tus propias conclusiones.

Por ejemplo, si tu predicción no coincide con el resultado de la simulación podría suceder que no tengas claros los conceptos teóricos relacionados, o bien que hayas diseñado mal el circuito. En cualquier caso, debes dedicar el tiempo que sea necesario hasta conseguir que tus predicciones coincidan con los resultados de simulación.

En particular, en esta sesión práctica debes predecir correctamente el resultado y bits de estado correspondientes a cualquier operación **antes de realizar simulación alguna**. Además, con cada operación debes predecir correctamente el estado que tendrán todos los elementos del circuito **justo después de llevar a cabo cualquiera de los pasos anteriores**.

En primer lugar vamos a cargar los valores a restar en `R0` y en `R1`, luego vamos a hacer la resta y vamos a llevar el resultado a `R0`, es decir, vamos a hacer `R0=R0-R1`. En cada uno de los pasos siguientes, si no se indica que una señal tiene que estar a 1 (activa), debería estar a 0. Si no lo haces así, se pueden producir cortocircuitos si se generan distintos valores sobre el mismo cable (pasaría por ejemplo si se activa `R0-IB` y `R1-IB` al mismo tiempo).

- Lanza la simulación.
- Carga, siguiendo los mismos pasos que hiciste antes, el valor 5 en `R0`. No hace falta que hagas la lectura.
- Carga el valor 4 en `R1`.
- Vuelca el valor de `R0` en el bus interno (`R0-IB`) y activa la entrada `IB-TMPE` para que se almacene en el registro `TMPE`. Fíjate que a la salida de este registro debería estar el mismo valor que se almacenó en `R0`.
- Vuelca el valor de `R1` en el bus interno y activa las señales, `Op1`, `Op0`, `compl-1` y `cin` de la ALU para que haga la operación de resta y, al mismo tiempo, activa las señales `ALU-TMPS` y `ALU-SR` para que se almacene el resultado y los bits de estado en `TMPS` y `SR` respectivamente. ¿Qué valor se almacena en `TMPS`? Responde en el cuestionario: pregunta 2 ¿Y en `SR`? Responde en el cuestionario: pregunta 3
- Ahora debes almacenar el valor de TMPS en `R0`. Para ello, vuelca el valor de `TMPS` al bus interno y lee ese valor desde el bus interno a `R0`.
- Por último, vuelca el valor de `R0` al bus interno para comprobar que se almacenó correctamente. ¿Qué valor aparece en el bus interno? Responde en el cuestionario: pregunta 4
- Detén la simulación.

4. Ejercicios adicionales

Estos ejercicios adicionales permiten al alumno reforzar los conocimientos adquiridos durante la sesión práctica.

- Para practicar la trasferencia entre registros, carga el registro `R0` con un valor a través de las entradas IB_0, IB_1, IB_2 e IB_3, y luego genera las señales

necesarias para copiar la información de `R0` a `R1`. Comprueba que `R1` tiene el valor adecuado.

- Realiza las operaciones recogidas en la tabla 1 «en papel», es decir, sin utilizar el simulador. En el caso de las operaciones lógicas el valor de los bits de estado *carry* y *overflow*, aunque la ALU los genera, no son significativos y deben ignorarse.

Tabla 1. Operaciones propuestas

R0	R1	Operación	Resultado	ZF	CF	OF	SF
-3	7	OR					
5	3	AND					
-6	-8	AND					
4	-6	Suma					
-3	7	Resta					
-1	-5	XOR					
5	-4	AND					

- Lleva a cabo las operaciones sobre el simulador y verifica que tus respuestas son correctas.
- Abre los circuitos de los registros `TMPE` y `TMPS` y observa cómo están hechos. Estudia en qué se diferencian entre sí y con el circuito para los registros de propósito general. Analiza qué diferencias hay en su funcionamiento.