Índice

Condicionales y bucles

Área de Arquitectura y Tecnología de Computadores – Versión 1.0.898, 18/02/2022

Objetivos

Esta sesión prosigue el objetivo de comprender la traducción de programas escritos en lenguaje de alto nivel (C++) al lenguaje ensamblador del Computador Teórico. Se estudiará en esta sesión la traducción de sentencias y bucles.

Conocimientos y materiales necesarios

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

- Conocer los sistemas de codificación binario natural y complemento a 2, así como tener soltura en la conversión a hexadecimal.
- Conocer el lenguaje de programación C++.
- Conocer el lenguaje ensamblador del Computador Teórico.
- Conocer conceptos básicos de traducción de sentencias de alto nivel a instrucciones de la máquina, tales como sentencias de asignación, aritmético-lógicas, condicionales, y bucles.
- 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. Condicionales y bucles

- **Ejercicio 8.** Traduce al ensamblador del Computador Teórico el siguiente fragmento de C++

```
if (a==b)     // a en r0, b en r1
  c=1;        // c en r2
else
  c=2;
d=c;          // d en r3
```

  teniendo en cuenta las asignaciones de variables a registros que se indican en los comentarios. Si las variables `a` y `b` son iguales, ¿qué valor debería aparecer al final en `d`? Responde en el cuestionario: pregunta 1.

  Para comprobarlo inicializa en el simulador web los registros que almacenan dichas variables `a` y `b` con dos valores iguales. Ejecuta tu código y comprueba que al final el registro que almacena la variable `d` tiene el valor que has predicho. ¿Y si los valores iniciales de `a` y `b` fuesen diferentes? ¿Cuál sería en este caso el valor final de `d`? Responde en el cuestionario: pregunta 2. Compruébalo de nuevo en el simulador, ejecutando el mismo código pero con otros valores iniciales para `a` y `b`.

  Nota: Si marcas la opción "Obtener trazado de cada instrucción", verás en la traza que junto a las instrucciones de salto condicional aparece una pequeña marca, que puede ser "✓" o "✗". La primera (✓) indica que la condición del salto se cumple y por tanto se saltará a la etiqueta de destino (en la traza las etiquetas ya no son visibles y han sido sustituidas por números, que indican cuántas instrucciones se saltarán para llegar al destino. En los saltos hacia atrás este número es negativo, y cuenta a la propia instrucción de salto). La segunda (✗) indica que la condición no se cumple y por tanto no se saltará a la etiqueta, sino que se continuará la ejecución por la instrucción siguiente al salto. Estas marcas son una mera ayuda, ya que tú mismo podrías averiguar si la condición se cumple o no mirando los valores que los bits de estado tienen en ese momento (los cuales también salen en la traza).

- **Ejercicio 9.** Considera el siguiente fragmento de código en C++, que asigna cero a la variable `a` si es mayor que `b`. Las dos variables almacenan números naturales.

```
if (a > b)  // a en r0, b en memoria (500h)
  a=0;
```

  Escribe las instrucciones de ensamblador que producen el mismo resultado, usando la asignación de variables indicada en los comentarios.

  Comprueba si funciona correctamente, poniendo en la variable `b` (es decir, en la dirección de memoria correspondiente) el valor `7`, y haz tres pruebas. En la primera, la variable `a` será menor (por ejemplo, vale `3`), en la segunda será igual (vale `7`) y en la tercera será mayor (por ejemplo, vale `12`). Comprueba que al finalizar tu programa la variable `a` (es decir, el registro que la contiene) conserva su valor inicial en las dos primeras pruebas, pero es cambiado por `0000h` en la última.

- **Ejercicio 10.** El siguiente código en C++ calcula la suma de los 100 primeros números naturales, dejando el resultado en la variable `a`:

```
a=0;                  // a en r0
for (i=0; i<100; i++)    // i en r1
  a=a+i;
```

  ¿Cuál será el resultado? Responde en el cuestionario: pregunta 3. Puedes obtener la respuesta escribiendo un pequeño programa en C++ como el anterior, que imprima el valor de `a` al finalizar, y ejecutándolo. O evaluar en un intérprete python la expresión `sum(range(100))`. Pásalo a hexadecimal para comparar con la salida del simulador (en python tienes la función `hex()`).

  Escribe el equivalente en ensamblador, usando la asignación de variables a registros indicada en los comentarios. Comprueba que el valor final de `a` (es decir, del registro en que se almacena) es el esperado, pero ten en cuenta que el simulador web te dará el resultado en hexadecimal. Si no sale lo que esperas, puedes activar la traza, pero en este caso mejor reduces el número de iteraciones del bucle hasta que localices el problema. Una vez corregido, lo vuelves a 100, pero desactiva la traza. ¡De lo contrario la traza tendría cientos de líneas!

- **Ejercicio 11.** El siguiente fragmento de código C++ va contando cuántos elementos distintos de cero hay en un array dado. Por no complicar el código, se supone que el array es de tamaño suficiente y siempre se encuentra un cero que finaliza el bucle antes de "salirse" de este tamaño.

```
contador=0;
i=0;
while (datos[i]!=0) {
  contador++;
  i++;
}
```

  Para traducir el código anterior a ensamblador, usa `r0` para el `contador`, `r1` para el índice `i` del array, y `r2` para contener la dirección de memoria en la que comienza el array, que será `0740h`. Necesitarás otro registro auxiliar que puede ser `r4` para contener la dirección del dato concreto al que se accede en cada iteración del bucle, el cual se calcula como la suma de la dirección en la que comienza el array más el índice `i`. También necesitarás un registro que contenga cero para comparar con el dato que estás procesando.

  Para probar tu código, pon en las direcciones `0740h`, `0741h` y `0742h` tres valores cualesquiera distintos de cero. Al terminar tu programa, el `contador` debe valer 3, ya que habrá encontrado tres valores consecutivos distintos de cero en el array. Prueba a poner un cero en `0741h` y ejecutar de nuevo, y el contador deberá ahora valer `1`, ya que tras contar el primer dato ya encuentra un cero. Finalmente prueba a poner un cero en `0740h` y comprueba que en este caso el contador vale cero.

---

2. Ejercicios adicionales

Ejercicios adicionales, por si te ha sobrado tiempo.

- **Ejercicio 12.** Considera el siguiente fragmento de código en C++, que asigna cero a la variable `a` si detecta que está fuera de un rango pre-establecido por las variables `b` y `c`. Las tres variables almacenan números naturales.

```
if ((a<b) || (a>c))  // a en r0, b en r1, c en r2
  a=0;
```

  Escribe las instrucciones de ensamblador que producen el mismo resultado, usando la asignación de variables a registros indicada en los comentarios.

  Comprueba si funciona correctamente, poniendo en las variables `b` y `c` (es decir, en los registros correspondientes a estas variables) los valores `5` y `10`, respectivamente, y haz tres pruebas. En la primera, la variable `a` está dentro del rango (por ejemplo, vale `7`), en la segunda se sale del rango por debajo (por ejemplo, vale `2`) y en la tercera se sale del rango por encima (por ejemplo vale `12`). Comprueba que al finalizar tu programa la variable `a` (es decir, el registro que la contiene) conserva su valor inicial en la primera prueba, pero es cambiado por `0000h` en las otras dos.

- **Ejercicio 13.** Cuando necesitamos saber si una variable contiene un valor par o impar, habitualmente usamos en C++ o en python el operador "módulo" que da el resto de la división, dividiendo por 2. Si ese resto es 0, es que el número era par y si es 1 es que era impar. Así por ejemplo:

```
if (a % 2 == 0)  // Si a es par, incrementar b
   b++;
```

  Sin embargo, traducir directamente eso a código máquina no es trivial, ya que no tenemos una instrucción máquina que nos de el resto de una división, y si tuviéramos que calcular ese resto mediante restas sucesivas sería muy ineficiente.

  En lugar de eso, suele utilizarse otra estrategia. Si te das cuenta, al expresar un número en binario, se ve claramente que si el número es par su último bit será cero, mientas que si el último bit es uno, eso implica que el número es impar. De este modo evitamos el tener que hacer divisiones. ¿Cómo comprobar entonces si el último bit es 1 ó 0? En código máquina resulta muy sencillo. Basta hacer la operación `AND` entre el dato en cuestión y el valor `0001h`. Si el resultado de esa operación es `0000h`, es que el último bit del dato era cero y por tanto el número par. Y este caso se detecta fácilmente a través del flag Z.

  Así pues, podríamos reescribir en C el programa anterior haciendo uso de esta propiedad, en lugar de usar divisiones o módulos. El lenguaje C también tiene el operador `AND` bit a bit, que se denota por &.

```
if (a & 0x0001 == 0)  // Si a es par, incrementar b
  b++;
```

  Traduce a ensamblador el listado anterior, haciendo `a=R0`, `b=R1` y comprueba su funcionamiento con el simulador web. Prueba con diferentes valores de `a` (pares e impares) para ver si en cada caso `b` se incrementa adecuadamente o no. Ten cuidado de que al finalizar el programa no haya resultado modificada la variable `a`, que debe terminar con el mismo valor con el que comenzó.

- **Ejercicio 14.** Modifica el ejercicio 10 haciendo uso del "truco" que has aprendido en el ejercicio anterior, para que sume los números impares comprendidos entre 0 y 99. Comprueba si obtienes el resultado correcto (que es 2500 o `09C4h`).