The Simpletron[*]

Machine-Language Programming

Algoritmos y Programación I (95.11/75.02)

7 de junio de 2018

1. Objetivo

El objetivo del presente trabajo es el diseño y desarrollo de una aplicación que permita interpretar y ejecutar un lenguaje de máquina inventado. A este intérprete lo llamaremos *Simpletron*. Utilizando el lenguaje inventando y el *Simpletron* se puede escribir cualquier programa.

Nota: El trabajo práctico está basado en un ejercicio del capítulo de punteros del libro de Deitel & Deitel, mencionado en la bibliografía de la materia. Se recomienda leer dicho ejercicio.

2. Alcance

Mediante el presente TP se busca que el/la estudiante aplique, conocimientos sobre los siguientes temas:

- Directivas al preprocesador C
- Programas en modo consola
- Tipos enumerativos
- Funciones
- Salida de datos con formato
- Modularización
- Arreglos/Vectores
- Memoria dinámica
- Arg. en línea de comandos
- Estructuras
- Archivos
- Punteros a función
- Modularización
- Tipos de Dato Abstracto, particularmente contenedores

[*]basado en un ejercicio del libro *C How to program* de Deitel & Deitel

3. Introducción

Se ha de crear una "computadora" a la cual llamaremos *Simpletron*[1]. La *Simpletron* ejecuta programas escritos en un lenguaje diseñado específicamente para este problema, el *Lenguaje de Máquina de la* Simpletron, LMS. También llamado el *assembly* de la *Simpletron*. Este lenguaje y este ejercicio brindarán un mayor entendimiento de lo que ocurre en los lenguajes de programación más cercanos al hardware.

4. Desarrollo

4.1. *Simpletron*

Toda la información que utiliza la *Simpletron* se maneja en *palabras*. A diferencia del trabajo anterior, aquí las palabras se leerán de archivos que podrán tener los formatos especificados en la sección 4.2.

Le *Simpletron* tiene espacio para almacenar una cantidad fija—en el sentido de que no se modificará su tamaño después de su creación—y finita de palabras, la *memoria*. En esta memoria se pueden almacenar instrucciones del programa, un dato o nada. Todas las palabras son enteros de, al menos, 16 bits.

Todas las instrucciones se deben cargar en memoria, por lo que las instrucciones son palabras. Todas las instrucciones están compuestas por un código de operación, *opcode*, y un operando. En la tabla 1 se resumen los códigos de operación que debe soportar el programa. En la representación binaria, de los 16 bits, los 7 de mayor significancia serán para almacenar opcode y los 9 siguientes almacenan el operando. Una representación gráfica se muestra en la figura 1.

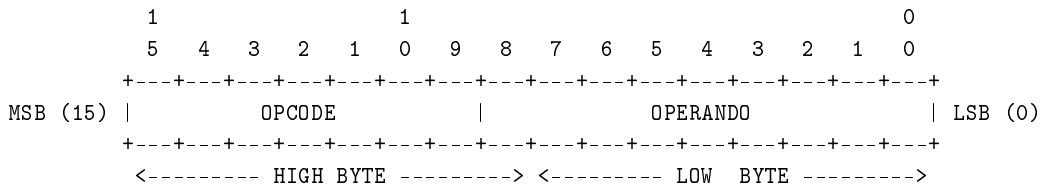

Figura 1: Representación binaria de una palabra

Operación	*OpCode*	Descripción
Op. de Entrada/Salida:		
`LEER`	10	Lee una palabra de *stdin* a una posición de memoria
`ESCRIBIR`	11	Imprime por *stdout* una posición de memoria
Op. de movimiento:		
`CARGAR`	20	Carga una palabra de la memoria en el acumulador
`GUARDAR`	21	Guarda una palabra del acumulador a la memoria
`PCARGAR`	22	Idem `CARGAR` pero el operando es *puntero*
`PGUARDAR`	23	Idem `GUARDAR` pero el operando es *puntero*
Op. aritméticas:		

[1] dado que es la computadora, o el computador, o el sistema, o la máquina, etc. la *Simpletron* se llamará indistintamente la o el o le *Simpletron*

Operación	*OpCode*	Descripción
`SUMAR`	30	Suma una palabra al acumulador
`RESTAR`	31	Resta una palabra al acumulador
`DIVIDIR`	32	Divide el acumulador por el operando
`MULTIPLICAR`	33	Multiplica el acumulador por el operando
Op. control:		
`JMP`	40	Salta a una ubicación de memoria
`JMPNEG`	41	Idem sólo si el acumulador es negativo
`JMPZERO`	42	Idem sólo si el acumulador es cero
`JNZ`	43	Idem sólo si el acumulador NO es cero
`DJNZ`	44	Decrementa el acumulador y salta si NO es cero
`HALT`	45	Finaliza el programa

Tabla 1: Códigos de operación—*opcodes*—del LMS

Además, la *Simpletron* posee un *acumulador*, `ACC`, un *registro* en el cual la información se coloca antes de ser utilizada por las distintas instrucciones que soporta la máquina.

Antes de ejecutar un programa escrito en lenguaje LMS, el mismo debe ser cargado en memoria. **Siempre**, la primera instrucción de cada programa se carga en la posición `0`.

4.2. Formatos de archivos

Los archivos a procesar por el programa para obtener los códigos podrán ser texto o binarios.

Formato de texto

Las palabras se almacenan en archivos de texto en forma *similar* al trabajo anterior: una palabra es un número entero de siete dígitos *con signo*, como son `+0950011`, `-7502325`, `+0666066`, `-2330153`, etcétera).

En este formato, los 3 primeros dígitos corresponden al opcode y los siguientes al operando; o bien todos los números corresponden a un dato a almacenar. Dado que las palabras se almacenarán en variables de 16 bits, se deben comprobar los valores límites a almacenar antes de guardarlos en la memoria.

Además, considere que el carácter `;` indica el comienzo de un comentario que termina al finalizar la línea. Todos los espacios en blanco al comienzo de la línea o al final de código útil son descartados, lo mismo que los comentarios. De esta forma, las líneas

```
+0440015 ; Decrementa el acumulador y, si el mismo no es 0,
         ; vuelve al comienzo del ciclo
```

son válidas, y equivalentes a

```
0440015
```

Ejemplos de código LMS

suma.lms

```
0  +0100009 ; LEER en la posición de memoria 09
1  +0100010 ; LEER en la posición de memoria 10
2  +0200009 ; CARGAR la posición de memoria 09 al acumulador
3  +0300010 ; SUMAR al acumulador la posición de memoria 10
4  +0210011 ; GUARDAR el acumulador en la posición de memoria 11
5  +0110011 ; ESCRIBIR por pantalla el contenido de la memoria 11
6  +0450000 ; HALT
7  +0000000 ; nop
8  +0000000 ; nop
9  +0000000 ; Variable A
10 +0000000 ; Variable B
11 +0000000 ; Resultado
```

resta.lms

```
0  +0100009
1  +0100010
2  +0200009
3  +0310010
4  +0410007 ; JMPNEG salta a la pos. 07 si el acumulador es negativo
5  +0110009
6  +0400008 ; JMP salta a la posición de memoria 08
7  +0110010
8  +0450000
9  +0000000
10 +0000000
11 +0000000
```

maximo.lms

```
0  +0100017
1  +0200018
2  +0310017
3  +0420015
4  +0200018
5  +0300021
6  +0210018
7  +0100019
8  +0200020
9  +0310019
10 +0410012
11 +0400001
12 +0200019
13 +0210020
14 +0400001
15 +0110020
16 +0450000
17 +0000000
18 +0000000
19 +0000000
20 +0000000
21 +0000001
```

promedio.lms

```
+0100022 ; LEER en la posición de memoria 22
+0200022 ; CARGAR la posición de memoria 22 al acumulador
+0310019 ; RESTAR al acumulador la posición de memoria 19
+0420012 ; JMPZERO salta a la pos. 12 si el acumulador es cero
+0200019 ; CARGAR la posición de memoria 19 al acumulador
+0300020 ; SUMAR el contenido de la pos. 20 al acumulador
+0210019 ; GUARDAR el contenido del acumulador en la posición 19
+0100018 ; LEER en la posición de memoria 18
+0200017 ; CARGAR la posición de memoria 17
+0300018 ; SUMAR la posición de memoria 18
+0210017 ; GUARDAR el contenido del acumulador en la posición 17
+0400001 ; JMP salta a la posición 01
+0200017 ; CARGAR la posición de memoria 17
+0320022 ; DIVIDIR el acumulador por el contenido de 22
+0210021 ; GUARDAR el acumulador en la posición 21
+0110021 ; ESCRIBIR por pantalla el contenido de 21
+0450000 ; HALT
0000000  ; Pos. 17 (valor acumulado)
0000000  ; Pos. 18 (variable)
0000000  ; Pos. 19 (contador)
0000001  ; Pos. 20 (incrementos de a uno)
0000000  ; Pos. 21
0000000  ; Pos. 22 (cantidad)
```

Formato binario

En este formato, las palabras se encuentran almacedas como enteros de 16 bits en formato Big-Endian. Esto es, primero se encuentra el byte más signficativo y luego el menos significativo, a continuación de éste se encuentra una nueva palabra. *Sugerencia:* leer cada byte por separado y rearmar la palabra una vez cargados dichos datos.

Dado que los datos ya se encuentran en 16 bits, no es necesario analizar y validar su contenido.

5. Simulador del *Simpletron*

Al igual que en el trabajo anterior, será necesario implementar un módulo que ejecute el código LMS cargado en la memoria del *Simpletron*. A continuación se repiten las especificaciones de la estructura que almacena la *Simpletron*.

Como mencionamos antes, la *Simpletron* posee un *acumulador* por donde pasan **todos** los datos. La *Simpletron* deberá iterar por las distintas posiciones de memoria, por lo que será necesario contar con una variable que nos permita llevar la cuenta de dicha posición, *program counter*. Con la variable antes mencionada podemos obtener el *registro de instrucción*, que equivale a la palabra almacenada en la memoria indicada por el contador antes mencionado. De este *registro de instrucción* podemos obtener tanto el código de operación, *opcode*, como el *operando*.

Finalizada la ejecución del programa, se hará un volcado de la aplicación, mostrando todos sus parámetros y el estado de la memoria, como se muestra en el ejemplo 1. Este volcado se conoce como *dump*, por su traducción del idioma

inglés. El volcado puede ser en formato de texto o en formato binario. El formato de texto se muestra en el ejemplo mencionado. El formato binario será una secuencia con los valores cargados en la memoria en formato Big-Endian, en forma análoga al formato de lectura especificado en la sección 4.2, es decir, no se almacenan los valores del acumulador, el *program counter*, etc.

```
REGISTROS:
     acumulador:     F20E
program counter:        8
    instruccion: +0450000
         opcode:       45
       operando:      000

MEMORIA:
000: 1409 140A 2809 3E0A 5207 1609 5008 160A  ....(.>.R...P...
010: 5A00 007D 0E6F 0000 0000 0000 0000 0000  Z..}.o..........
020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0b0: 0000 0000                                ....
```

Ejemplo 1: Ejecución y *dump* textual en español

A diferencia del trabajo anterior, en este caso se podrán procesar varios archivos de código de máquina de la *Simpletron*. Para ello, se deberán cargar en memoria, utilizando una lista, todos los archivos de entrada que reciba el programa y luego se ejecutarán.

5.1. Implementación

A diferencia del libro, usaremos una estructura para almacenar el estado del *Simpletron*. La misma debe contener todas las variables antes mencionadas y, además, la memoria.

Además, es requisito separar la funcionalidad del *Simpletron* de la del simulador. En este caso, se recomienda consultar a los docentes sobre las ideas para llevar a cabo la modularización.

Para el almacenamiento de la memoria, es requisito crear un tipo de dato abstracto de tipo contenedor, TDA Vector.

Para la carga de las distintas instancias del *Simpletron* es requisito utilizar el tipo de dato abstracto TDA Lista (polimórfica, simplemente enlazada).

Se preferirá un programa con alta cohesión en los módulos y bajo acoplamiento entre los mismos.

5.2. Argumentos del programa

La aplicación desarrollada debe ser invocable por línea de comandos de acuerdo con el siguiente ejemplo

```
./run [-h] [-m N] [-f FMT] [-|[ARCHIVO1 [ARCHIVO2 ...]]]
```

El programa carga los archivos de entrada `ARCHIVO1`, `ARCHIVO2`, ...

Archivos de entrada

Cada `ARCHIVO` es una cadena que contiene el nombre de un archivo a procesar. Los archivos pueden comenzar con un carácter, `t` o `b`, que indica si se deben leer en formato texto o binario, respectivamente. Estos carácteres deben estar separados del nombre del archivo por el carácter `:`. Son válidas, entonces, las siguientes expresiones: `archivo`, `t:archivo` o `b:archivo`. Si no se indica el formato, se interpreta como texto.

Si el archivo de entrada es el carácter `-`, el programa se carga de `stdin` y no puede haber otro archivo a cargar.

-h, --help

Muestra una ayuda

-m `N`, **--memoria** `N`

Genera una memoria de `N` **palabras** en la simpletron. Por omisión, N vale 50.

-f `FMT`, **--formato** `FMT`

Indica el formato de la salida. Si `FMT` es `txt`, el formato debe ser texto. Si `FMT` es `bin`, el formato debe ser binario. Por omisión, el formato es texto.

5.3. Ejecución del programa

Suponiendo que el programa se llama `run`, algunas variantes de ejecución son:

```
$ # Argumentos posicionales:
$ ./run -h
$ ./run -m 40 -f bin -
$ ./run -m 25 -f txt b:promedio.sac
$ cat maximo.lms | ./run -m 80
$ # Argumentos no posicionales:
$ ./run -h
$ ./run -f bin -m 25 b:promedio.sac t:suma.lms
$ ./run -m 25 -f txt b:resta.sac t:promedio.lms t:maximo.lms
```

6. Testing

Para probar la correcta ejecución del programa se deben ejecutar los códigos LMS que se encuentran en este enunciado. Algunos de los programas se encuentran documentados, otros no: documentarlos.

6.1. Fugas de memoria

Las fugas de memoria de la aplicación pueden ser advertidas utilizando la aplicación *valgrind*. Para que la misma indique dónde se producen fugas de memoria, es necesario compilar nuestra aplicación en modo debug.

Una vez compilada la aplicación, *valgrind* se ejecuta de la siguiente manera:

```
valgrind ./mi_aplicacion
valgrind ./mi_aplicacion argumento1 argumento2 ...
```

Al ejecutar correctamente el comando **y** si el programa no tiene fallas, se ve una leyenda similar a la siguiente:

```
==23390== HEAP SUMMARY:
==23390==     in use at exit: 0 bytes in 0 blocks
==23390==   total heap usage: 89,408 allocs, 89,408 frees, 5,057,256
    bytes allocated
==23390==
==23390== All heap blocks were freed -- no leaks are possible
==23390==
==23390== For counts of detected and suppressed errors, rerun with:
   -v
==23390== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
   from 0)
```

Ejemplo 2: Resultado de ejecución de valgrind sin fugas de memoria

Si en cambio el programa tiene fugas de memoria, el mensaje es:

```
==25854== HEAP SUMMARY:
==25854==     in use at exit: 456,722 bytes in 3,932 blocks
==25854==   total heap usage: 89,408 allocs, 85,476 frees, 5,057,256
    bytes allocated
==25854==
==25854== LEAK SUMMARY:
==25854==    definitely lost: 456,722 bytes in 3,932 blocks
==25854==    indirectly lost: 0 bytes in 0 blocks
==25854==      possibly lost: 0 bytes in 0 blocks
==25854==    still reachable: 0 bytes in 0 blocks
==25854==         suppressed: 0 bytes in 0 blocks
==25854== Rerun with --leak-check=full to see details of leaked
   memory
==25854==
==25854== For counts of detected and suppressed errors, rerun with:
   -v
==25854== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
   from 0)
```

Ejemplo 3: Resultado de ejecución de valgrind con fugas de memoria

7. Restricciones

La realización de los programas pedidos está sujeta a las siguientes restricciones:

- Debe realizarse en grupos de **3 (tres)** integrantes.
- No está permitida la utilización de `scanf()`, `gets()`[2], `fflush(stdin)`[3],la biblioteca `conio.h`[4] (`#include <conio.h>`), etc.
- Debe recurrirse a la utilización de funciones mediante una adecuada parametrización.
- Deben utilizarse punteros a función en la ejecución de los opcodes.
- Deben utilizarse TDAs para el vector y las listas, por lo menos.
- No está permitido en absoluto tener hard-codings:

```
...
int opcode;
...
if (opcode == 40)                /* ¡¡hard-coded!!*/
    printf("%s","xxxxxxxx");     /* ¡¡hard-coded!!*/
else if(opcode == 41)            /* ¡¡hard-coded!!*/
    simpletron_jmpneg(state);
...
```

  sino que debe recurrirse al uso de `ETIQUETAS`, `CONSTANTES SIMBÓLICAS`, `MACROS`, etc.

  Los ejemplos no son exhaustivos, sino que existen otros hard-codings y tampoco son aceptados.

- Hay ciertas cuestiones que no han sido especificadas intencionalmente en este Requerimiento, para darle al/la desarrollador/a la libertad de elegir implementaciones que, según su criterio, resulten más convenientes en determinadas situaciones. Por lo tanto, se debe explicitar cada una de las decisiones adoptadas, y el o los fundamentos considerados para las mismas.

8. Entrega

Fecha límite: 3 de julio de 2018.

No se requiere entrega en papel, pero se acepta. Deberá realizarse una entrega digital, a través del campus de la materia, de un único archivo cuyo nombre debe seguir el siguiente formato:

`YYYYMMDD_apellido1-apellido2-apellido3-N.tar.gz`

donde `YYYY` es el año (2018), `MM` el mes y `DD` el día en que uno de los integrantes sube el archivo, `apellido-1a3` son los apellidos de los integrantes ordenados alfabéticamente, `N` indica el número de vez que se envía el trabajo (`1`, `2`, etc.), y `.tar.gz` es la extensión, que no necesariamente es `.tar.gz`.

El archivo comprimido debe contener los siguientes elementos:

[2] obsoleta en C99 [3], eliminada en C11 [4] por fallas de seguridad en su uso.
[3] comportamiento indefinido para flujos de entrada ([3],[4]). Definida en estándar POSIX.
[4] biblioteca no estándar, con diferentes implementaciones y licencias, y no siempre disponible.

- La correspondiente documentación de desarrollo del TP (en formato pdf), siguiendo la numeración siguiente, incluyendo:

  1. Carátula del TP. Incluir una dirección de correo electrónico.
  2. Enunciado del TP.
  3. Estructura funcional de los programas desarrollados.
  4. Explicación de cada una de las alternativas consideradas y las estrategias adoptadas.
  5. Resultados de la ejecución (corridas) de los programas, captura de las pantallas, bajo condiciones normales e inesperadas de entrada.
  6. **Archivos de prueba utilizados.**
  7. Reseña sobre los problemas encontrados en el desarrollo de los programas y las soluciones implementadas para subsanarlos.
  8. Bibliografía (ver sección 9).
  9. Indicaciones sobre la compilación de lo entregado para generar la aplicación.
     NOTA: Si la compilación del código fuente presenta mensajes de aviso (warning), notas o errores, los mismos deben ser comentados en un apartado del informe.

  NOTA: El Informe deberá ser redactado en *correcto* idioma castellano.

- Códigos fuentes en formato de texto plano (.c y .h), *debidamente documentados*.

NOTA: Se debe generar y subir un único archivo (comprimido) con todos los elementos de la entrega digital. **NO usar RAR.** La compresión RAR no es un formato libre, en tanto sí se puede utilizar *ZIP*, *GUNZIP*, u otros (soportados, por ejemplo, por la aplicación de archivo *TAR*).

Si no se presenta cada uno de estos ítems, será rechazado el TP.

9. Bibliografía

Debe incluirse la referencia a toda bibliografía consultada para la realización del presente TP: libros, artículos, URLs, etc., citando:

- Denominación completa del material (Título, Autores, Edición, Volumen, etc.).
- Código ISBN del libro (opcional: código interbibliotecario).
- URL del sitio consultado. No poner Wikipedia.org o stackexchange.com, sino que debe incluírse un enlace al artículo, hilo, etc. consultado.

Utilizando LaTeX, la inclusión de citas/referencias es trivial. Los editores de texto gráficos de las suites de ofimática, como LibreOffice Write o MS Word, admiten plugins que facilitan la inclusión.

Ejemplo de referencias

[1] B.W. Kernighan y D.M. Ritchie. *The C Programming Language*. 2.ª ed. Prentice-Hall software series. Prentice Hall, 1988. ISBN: 9780131103627.

[2] P. Deitel y H. Deitel. *C How to Program*. 7.ª ed. Pearson Education, 2012. ISBN: 9780133061567.

[3] ISO/IEC. *Programming Languages – C*. ISO/IEC 9899:1999(E). ANSI, dic. de 1999, págs. 270-271.

[4] ISO/IEC. *Programming Languages – C*. INCITS/ISO/IEC 9899:2011. INCITS/ISO/IEC, 2012, pág. 305.