

Un Lenguaje de Bajo Nivel como Apoyo al Aprendizaje en el Primer Curso de Programación de las Carreras de Ingeniería

Federico Meza, Ruth Garrido y César Astudillo
[fmeza,rgarrido,castudillo]@utalca.cl
Departamento de Ciencias de la Computación
Facultad de Ingeniería
Universidad de Talca
Camino Los Niches Km. 1 - Curicó - CHILE

Resumen

En la enseñanza de la programación, durante el primer año de estudios, se omite o se estudia en forma superficial los detalles del computador subyacente. Esto provoca problemas de comprensión a los alumnos, por ejemplo, respecto al control de flujo, el uso de variables y punteros. Un enfoque alternativo consiste en comprender primero la arquitectura del computador, exponiendo sus características internas y estudiando la forma en que ejecuta los programas. En este artículo describimos nuestra experiencia siguiendo esta metodología. En nuestra asignatura los alumnos trabajan con el lenguaje ensamblador de un computador ficticio, que cuenta con un conjunto de instrucciones muy simple y limitado. Posteriormente, la mayor parte del curso se concentra en la programación en el lenguaje Java. Los resultados que hemos obtenido con el primer grupo de alumnos que se han sometido a este programa son alentadores.

Palabras clave. Primera asignatura, herramientas y ambientes de apoyo a la enseñanza y aprendizaje, metodologías de enseñanza, lenguaje ensamblador, introducción a la computación.

1. Introducción

La enseñanza de la programación en el primer año de las carreras universitarias ha sido por mucho tiempo un tema de intenso debate. La discusión se ha centrado, principalmente, en el paradigma de programación a emplear, así como en el lenguaje que se utilizará para escribir los programas. Así, podemos encontrar enfoques que utilizan lenguajes imperativos, funcionales, u orientados a objetos [1]. También es posible abstraerse de un lenguaje o paradigma particular, y concentrarse en el desarrollo de algoritmos y en la resolución de problemas a alto nivel.

En la actualidad las alternativas que predominan son las que utilizan lenguajes imperativos u orientados a objetos. En este contexto, un programa puede ser tratado como una entidad abstracta, que se puede construir y manipular en forma independiente de un computador, en la forma que lo propone Dijkstra [3]. Así, el objeto de estudio es el programa y el computador pierde relevancia. Alternativamente, de manera opuesta, el programa puede estudiarse como aquello que especializa una máquina de propósito general – llámese un computador. En este caso, el computador pasa a ser tan importante como los programas que se escriben para él.

Por lo general, en nuestras universidades los programas se estudian en estrecha relación con su comportamiento esperado en el computador, en oposición a lo propuesto por Dijkstra. Incluso, se acostumbra mostrar la ejecución de los programas durante las clases. Sin embargo, a pesar de la relación estrecha que existe entre estos dos componentes –programa y computador– por lo general, el último de ellos no se trata con la suficiente profundidad en el primer curso de programación. Muchos conceptos no se cubren con precisión, y algunos

de ellos se suponen ciertos o se presentan con excesiva simplicidad. Consideremos, por ejemplo, la ejecución secuencial de un programa, cuya comprensión es fundamental para el aprendizaje de la programación. Con un poco de conocimiento de arquitectura de computadores y del ciclo de ejecución de las instrucciones, el concepto de secuencialidad y las construcciones de control de flujo pueden ser asimilados con facilidad.

Otra herramienta, muy utilizada en los lenguajes de alto nivel, la constituye los punteros a memoria. Incluso lenguajes orientados a objetos, como Java, no pueden escapar de los comportamientos misteriosos de los punteros. Por ejemplo, en Java es posible comparar dos variables enteras para ver si sus contenidos son iguales, utilizando el operador de comparación. Sin embargo, no puede hacerse lo mismo con dos instancias de una clase de objetos, pues el nombre del objeto es, en la realidad del computador, un puntero a la dirección de memoria donde se guarda el objeto. Este tipo de inconsistencias semánticas confunden a quienes están aprendiendo a programar. En este caso, la comprensión de los punteros puede verse beneficiada si se ha estudiado a bajo nivel la estructura de la memoria, y la forma en que en ella se almacenan tanto los programas como los datos.

Harrison, *et. al.*, señala dos enfoques para abordar la enseñanza de la programación [7]. En el primero de ellos, conocido como la *caja negra*, el alumno utiliza y escribe programas para un computador sin interiorizarse de sus componentes internos o de la forma en que éstos interactúan. Así, la enseñanza de un lenguaje de alto nivel satisface las necesidades planteadas bajo estas premisas. El segundo enfoque, llamado la *caja blanca*, permite que el alumno vea el computador como la suma de varios componentes –memoria, registros, dispositivos de entrada y salida, etc.– los cuales interactúan de acuerdo a las instrucciones que componen un programa cargado en la memoria. En este caso, el mero estudio de un lenguaje de alto nivel no permite llegar al nivel de detalle requerido. Entonces, por lo general, se utiliza un lenguaje de bajo nivel – ensamblador, ya sea real o ficticio, que permite introducir los conceptos de nivel más bajo.

Nuestra experiencia nos ha enseñado que el desconocimiento de los detalles intrínsecos a la ejecución de los programas hace más difícil el aprendizaje de la programación. El problema es que, por lo general, el primer curso de programación se restringe al estudio de un lenguaje de alto nivel, y se da poco énfasis a los fundamentos de *hardware* que sustentan la ejecución de los programas. Creemos que si se va a estudiar programación relacionando los programas con su ejecución en un computador, entonces es necesario conocer con detalle la operación del computador. De esta forma se facilita el aprendizaje.

Los lenguajes de alto nivel tratan de ocultar el funcionamiento general del computador, ofreciendo al programador un alto grado de abstracción. Por el contrario, el uso de un lenguaje de bajo nivel más bien pone al descubierto el funcionamiento del computador. Utilizando un lenguaje de bajo nivel podemos desmitificar el computador, exponiendo sus componentes y la forma en que interactúan. Los alumnos trabajan con componentes concretos, como lo son la memoria y los registros.

Por otra parte, desde el comienzo, el alumno aprende a tratar con las limitaciones del computador, por ejemplo, en cuanto a tamaño de la memoria, rangos de los tipos de datos, y otros temas relacionados. Además, el alumno logra un completo entendimiento de la representación e interpretación de los datos en la memoria. Por ejemplo, los alumnos aprenden que una cadena de bits almacenada en la memoria puede ser interpretada de diferentes formas.

La idea de enseñar un lenguaje de bajo nivel durante el primer año de estudios no es novedosa ni desconocida. Quienes la han implementado han seguido cuatro estrategias distintas. Algunas veces se utiliza un enfoque bastante radical, en el que se limita la programación exclusivamente al lenguaje ensamblador, obviando los lenguajes de alto nivel en la primera asignatura [10]. Otras veces, el curso se divide en distintos módulos que se desarrollan en paralelo, dedicando uno de ellos a la programación de alto nivel, y otro al *hardware* y a la arquitectura de computadores [2, 8]. Alternativamente, el lenguaje de bajo nivel puede utilizarse durante la introducción del curso, para lograr una mejor comprensión de los fundamentos de la computación, y posteriormente dar lugar a la enseñanza de un lenguaje de alto nivel [9, 7]. Finalmente, algunas asignaturas no se concentran exclusivamente en la programación, sino que hacen una cobertura de las distintas áreas de la computación; sin embargo, estos cursos también pueden beneficiarse del uso de un lenguaje de bajo nivel para introducir a los alumnos algunos conceptos básicos de programación [11].

El uso de un lenguaje de bajo nivel puede complementarse con un emulador que permita la ejecución de los programas en un ambiente simulado. Esta es una práctica muy común en los cursos de arquitectura de computadores. Aunque muchas veces los emuladores recrean computadores muy complejos, algunos de ellos son lo suficientemente simples para ser utilizados en cursos introductorios [10, 12, 13, 4, 5].

Durante el primer año de estudios, los estudiantes de Ingeniería de la Universidad de Talca toman el curso *Computación y Programación*. Esto incluye los alumnos de las carreras de Ingeniería Civil en Computación, Ingeniería en Bioinformática e Ingeniería Civil Industrial. Para los alumnos de esta última carrera, Computación y Programación es el único curso de computación en su plan de estudios. En esta asignatura se entrega a los alumnos los fundamentos de la computación, incluyendo historia, estructura interna del computador, el concepto de programa, el ciclo de *fetching* y la interacción entre el *hardware* y el *software*. Posteriormente, y durante la mayor parte del curso, se les enseña a programar utilizando el lenguaje de programación Java.

A partir del presente año, hemos modificado la forma de introducir la programación, utilizando un lenguaje ensamblador para un computador ficticio, cuya arquitectura es de fácil comprensión. Contamos, además, con un emulador para la ejecución de los programas. Previamente, estudiamos algunos principios de arquitectura de computadores que permiten la comprensión de la operación fundamental del computador. Creemos que el aprendizaje se ha visto beneficiado con este enfoque, y hemos detectado importantes mejoras en la asimilación de la programación en Java.

En general, intentamos que los programas de bajo nivel que proponemos sean lo suficientemente sencillos como para no confundir al alumno con demasiadas o muy complejas construcciones. Además, es posible seguir la ejecución de los programas paso a paso, conociendo con todo detalle la forma en que el estado del computador cambia con la ejecución de cada instrucción. Creemos que este aspecto es de vital importancia para comprender lo que es un programa y cómo se ejecuta.

En este artículo ofrecemos los detalles de la forma en que incorporamos los conceptos básicos de arquitectura de computadores y la programación en lenguaje ensamblador en nuestro curso, así como algunos comentarios acerca de los resultados que hemos obtenido. El resto del artículo se organiza de la siguiente manera: Primero analizamos algunos aspectos que consideramos problemáticos en el proceso de aprendizaje de la programación, y la forma en que el uso de un lenguaje de bajo nivel ayuda a abordarlos de mejor manera. Posteriormente, describimos la estructura del curso y entramos en algunos detalles de importancia, incluyendo una descripción de la arquitectura del computador empleado y del lenguaje ensamblador con el que se escriben los programas. A continuación, incluimos los resultados de una encuesta que fue respondida por los alumnos del curso, respecto a la conveniencia del nuevo enfoque. Finalmente, presentamos nuestras conclusiones y enunciamos el trabajo futuro al que nos enfrentamos.

2. Beneficios de trabajar a bajo nivel

Durante el proceso de aprendizaje de la programación, hemos detectado una serie de aspectos que resultan difíciles de asimilar por parte de los alumnos.

Consideremos por ejemplo una instrucción simple de acumulación en un lenguaje de alto nivel, como Java: $i=i+2$. Un alumno sin experiencia que es enfrentado por primera vez a este tipo de instrucción, sufre una enorme confusión, a raíz del conocimiento que tiene de las igualdades matemáticas.

Por el contrario, en ensamblador, la variable i se manipula en forma más concreta, como una celda de la memoria. Esta celda cuenta con una dirección asociada que se utiliza para su lectura o escritura. Así, para llevar a cabo la acumulación es necesario efectuar la suma del contenido de la celda de memoria con 2, para luego guardar el resultado en la misma celda de memoria. Lo concreto de esta manipulación logra un mejor entendimiento por parte del alumno.

Entre los aspectos más problemáticos que hemos podido encontrar, al enseñar a programar utilizando un lenguaje de programación de alto nivel, podemos señalar:

- El concepto de programa y el flujo secuencial en la ejecución de las instrucciones. Los alumnos no asimilan con facilidad la propiedad de secuencialidad en un programa computacional. Esto afecta también el aprendizaje de las construcciones de control de flujo.
- El concepto de lenguaje de programación y de conjunto de instrucciones.
- Los componentes sintácticos y semánticos de un lenguaje de programación.
- El concepto de estado de un programa, el cual está ligado a la visión de un programa en ejecución como una máquina con un número finito de estados. La ejecución de las instrucciones provoca cambios de estado, los cuales están determinados por la semántica del lenguaje de programación.
- Las variables y las instrucciones de asignación, incremento y acumulación. Se confunden las variables de un programa y las instrucciones de asignación, con las variables algebraicas y las ecuaciones matemáticas, respectivamente.
- Las construcciones para ejecución condicional, como el *if* y el *switch*. La forma en que es posible cambiar el flujo de ejecución de las instrucciones es un concepto difícil de asimilar. En particular, la ejecución condicional de instrucciones.
- Las construcciones de iteración como el *while* y el *for*. Al igual que en el caso anterior, el concepto de repetición de un grupo de instrucciones presenta problemas en el proceso de aprendizaje.
- El uso de condiciones, asociadas con las construcciones de ejecución condicional y repetitiva.
- Los conceptos de entradas y salidas de un programa, las cuales le dan generalidad al programa.
- Los tipos de datos y sus rangos de valores son conceptos muy concretos, pues se derivan de la representación en el computador. Por este motivo, son tópicos muy difíciles de asimilar en forma abstracta.
- El concepto y la necesidad de traducción, llevada a cabo por un compilador o intérprete.

Estos y otros aspectos pueden abordarse utilizando un lenguaje con una sintaxis y una semántica muy simples, y con efectos muy cercanos al *hardware*, es decir, con un nivel bajo de abstracción. Aquí se vuelve valioso el uso de un ensamblador y el conocimiento detallado de la arquitectura del *hardware* sobre el que opera.

El uso de un computador ficticio es conveniente para mantener la simplicidad. Sin embargo, tanto su arquitectura como el conjunto de instrucciones que soporta, y los tipos de datos que manipula, deben ser realistas.

Los estudiantes se enfrentan a un computador con sus componentes expuestos –memoria y procesador. Las instrucciones de un programa y los datos que el programa utiliza se almacenan en la memoria. El ciclo de ejecución garantiza la recuperación, decodificación y ejecución de las instrucciones. El estado del programa se encuentra a la vista, con sólo mirar el contenido de la memoria y de los registros del procesador. A su vez, la semántica de los cambios de estado provocados por la ejecución de las instrucciones se vuelve visible por el mismo motivo. El conjunto limitado de instrucciones con que los alumnos pueden trabajar los hace comprender de mejor manera lo que es un lenguaje de programación.

El flujo de ejecución de las instrucciones se comprende a partir del entendimiento del ciclo de ejecución de un computador –ciclo de *fetching*. La manipulación a bajo nivel del contador de programa permite a los alumnos entender fácilmente las construcciones de control de flujo para ejecución condicional e iterativa. Las condiciones son simples, y el control de flujo se limita a la verificación de los valores de ciertas banderas representadas como bits en un registro.

Los datos que utiliza el programa son elementos concretos. Se trata de números binarios que se guardan en celdas de memoria. Para utilizarlos es necesario darles la interpretación apropiada, reforzando así el concepto de tipo de datos y comprendiendo la razón por la que existen rangos de valores para los distintos tipos. Las asignaciones también dejan de ser elementos oscuros, pues se trata de un simple flujo de bits entre celdas de memoria y los registros del procesador o viceversa.

Utilizando un lenguaje de bajo nivel, la acumulación $i=i+2$ se convierte en algo más concreto:

```
LOADI 2    % Carga el operando inmediato 2 al registro acumulador
ADD 0      % Suma el valor del registro con el valor de la celda 0 (la variable)
STORE 0    % Reescribe el resultado en la misma celda (dirección 0)
```

La comprensión de otros tópicos relacionados con lenguajes de programación y compiladores también se ve beneficiada con el uso de un lenguaje ensamblador que actúa sobre una arquitectura de *hardware* conocida para los alumnos. El mismo programa emulador que ejecuta los programas de bajo nivel puede ser escrito por los alumnos en la etapa más madura del curso. Esto se facilita pues existe una correspondencia uno a uno entre las instrucciones del ensamblador y el conjunto de instrucciones del computador.

Finalmente, el hecho de haber escrito programas de bajo nivel desde el inicio de sus carreras, hace que los alumnos no sientan temor de trabajar a niveles bajos de la jerarquía de los sistemas – llámese programación de sistemas [14], y no deban esperar hasta cursos de niveles superiores. A la vez, aprenden a valorar la importancia de los lenguajes de alto nivel.

3. Estructura del Curso

Nuestro curso se divide en tres partes, claramente identificables:

1. Introducción a la computación y a la arquitectura de computadores. Se estudia la historia de la computación, haciendo especial énfasis en el surgimiento del concepto de programa almacenado. Se presenta la Arquitectura de von Neumann y se estudia el ciclo de *fetching* de las instrucciones. La principal fuente de referencia para esta sección es el texto de Levine [6].
2. Introducción a la programación. Se trabaja con la arquitectura de un computador ficticio y su conjunto de 16 instrucciones, así como un lenguaje ensamblador para su programación.
3. Programación de alto nivel. Utilizando un lenguaje moderno y orientado a objetos, como Java, se introduce la programación de alto nivel. Inicialmente se trabaja únicamente con clases de objetos predefinidas y siguiendo un paradigma procedimental. Al final del curso se estudian los conceptos de abstracción de datos y se construyen nuevas clases de objetos.

El computador se presenta a los alumnos como una máquina de propósito general, contrastándolo con otros dispositivos existentes en la actualidad. Es fundamental el concepto de *programa almacenado*, y la forma en que un programa especializa la función del computador. Estos tópicos se tratan durante la presentación de la historia de la computación, de modo que es posible contrastar los primeros computadores de propósito específico, con los que tenemos en la actualidad.

La arquitectura de von Neumann es el fundamento de la programación a bajo nivel que llevan a cabo los alumnos (Fig. 1). Al nivel más alto, se presentan tres componentes enlazados por un bus: El procesador, la memoria, y el subsistema de entrada y salida. La memoria almacena los programas y los datos que éstos utilizan. Los alumnos la perciben como un arreglo unidimensional de celdas, cada una de las cuales tiene una dirección única que la identifica y permite su uso. Los alumnos aprenden conceptos de direccionamiento, tamaño de las direcciones, tamaño de palabra, y representación de distintos tipos de datos –enteros positivos y negativos, números en punto flotante y caracteres. Por su parte, el subsistema de entrada y salida se estudia desde la perspectiva de la interacción del computador con el mundo exterior.

Posteriormente se estudia en mayor detalle el procesador y sus componentes: (1) la unidad aritmético-lógica, o simplemente ALU, que lleva a cabo operaciones aritméticas –como sumas y multiplicaciones– y lógicas –como comparaciones, y (2) la unidad de control, encargada de ejecutar los programas repitiendo en forma constante el ciclo de *fetching* de las instrucciones (Fig. 2). Se estudia también la importancia en este proceso

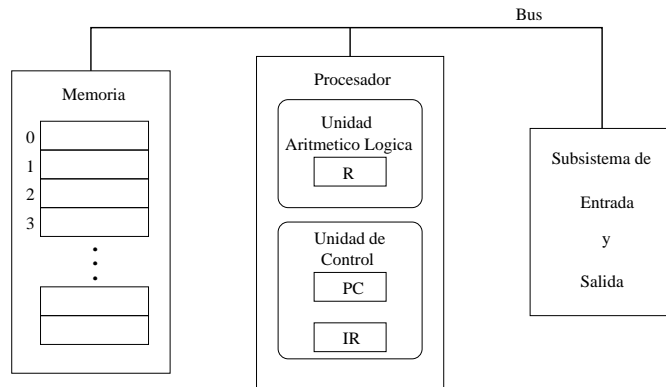


Figura 1: Arquitectura de von Neumann

de dos registros: El contador de programa –PC– que contiene la dirección de la celda de memoria en donde se encuentra la siguiente instrucción a ejecutar, y el registro de instrucción –IR– en donde se carga la instrucción para su ejecución. Los alumnos aprenden que el ciclo de *fetching* permite la integración del *software* con el *hardware*; en otras palabras, permite que el computador ejecute los programas. Por su parte, la ALU cuenta con registros acumuladores para guardar operandos y resultados de las operaciones que efectúa. Junto con el ciclo de *fetching*, se introduce a los alumnos al concepto de *conjunto de instrucciones*, y se tratan tópicos relacionados con el formato de las instrucciones, su representación, códigos de operación y operandos. Finalmente, se estudia el registro de banderas, o *flags*, que permite la ejecución de instrucciones de salto condicional y la señalización de ciertas situaciones de interés, como los desbordamientos en las operaciones aritméticas.

El computador ficticio para el que se escriben los programas trabaja únicamente con números enteros y con caracteres ASCII. Cuenta con una memoria de 256 celdas de 8 bits cada una, es decir, 256 bytes en total. Por esta razón, las direcciones requieren de 8 bits para su representación. Los programas se cargan en la memoria en la dirección 128, dejando las celdas con direcciones menores –desde 0 hasta 127– para guardar ahí los datos. Cuando se ejecuta un programa, la primera instrucción será la que se encuentra en la dirección 128.

El procesador cuenta con un único registro acumulador, al que llamamos R, de 8 bits. Por lo tanto, las operaciones aritméticas trabajan sobre operandos de 8 bits. Los números enteros varían entre -128 y 127, y los negativos se representan utilizando complemento a 2. Existe también un registro de banderas, con dos bits que son de interés: EQ y LT, que se utilizan en las instrucciones de comparación y salto condicional. También se considera el registro contador de programa, referido como PC.

El computador es capaz de comunicarse con su entorno a través de un dispositivo de entrada y uno de salida. La interacción se limita a números enteros y caracteres ASCII. Los alumnos visualizan estos dispositivos como si fueran un teclado y una impresora de caracteres, respectivamente.

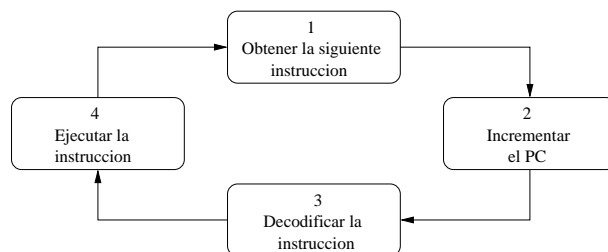


Figura 2: Ciclo de *fetching*

El código de operación requiere de 4 bits para su representación, dado que el conjunto de instrucciones del computador consta de 16 instrucciones. Sin embargo, se decidió utilizar 8 bits para poder ampliar la cantidad de instrucciones en el futuro. Así, los primeros 4 bits del código de operación serán siempre 0. El Cuadro 1 resume el conjunto de instrucciones del computador. Las instrucciones se agrupan en 6 categorías: Transferencia de datos, operaciones aritméticas, comparación, salto, entrada/salida y detención.

Opcode	Nemónico	Descripción
0000	LOAD X	Memoria[X] \rightarrow R
0001	STORE X	R \rightarrow Memoria[X]
0010	LOADI dato	dato \rightarrow R
0011	ADD X	R + Memoria[X] \rightarrow R
0100	INC X	Memoria[X] +1 \rightarrow Memoria[X]
0101	SUB X	R - Memoria[X] \rightarrow R
0110	DEC X	Memoria[X] -1 \rightarrow Memoria[X]
0111	CMP X	Si Memoria[X]=R, 1 \rightarrow EQ Si Memoria[X] \neq R, 0 \rightarrow EQ Si Memoria[X]<R, 1 \rightarrow LT Si Memoria[X] \geq R, 0 \rightarrow LT
1000	JMP X	X \rightarrow PC
1001	JMPLT X	Si LT= 1, X \rightarrow PC
1010	JMPEQ X	Si EQ= 1, X \rightarrow PC
1011	IIN X	Valor entero ingresado \rightarrow Memoria[X]
1100	CIN X	Caracter ingresado \rightarrow Memoria[X]
1101	IOUT X	Decimal(Memoria[X]) \rightarrow Salida
1110	COUT X	ASCII(Memoria[X]) \rightarrow Salida
1111	HALT	Termina la ejecución del programa

Cuadro 1: Conjunto de instrucciones del computador

La letra X denota una dirección de memoria, sobre la cual actúa una instrucción. En la descripción de las instrucciones, para referenciar el contenido de una celda de memoria se utiliza la notación Memoria[dirección]. Por ejemplo, Memoria[X] se refiere al contenido de la celda de memoria cuya dirección es X.

Inicialmente, los alumnos se enfrentan con programas muy simples, que llevan a cabo cálculos matemáticos y generan salidas. En este punto, intentamos enfatizar en los conceptos estudiados sobre el ciclo de *fetching* y la ejecución secuencial. Asimismo, reforzamos los conocimientos sobre el estado de un programa y sus transiciones, viendo la forma en que la ejecución de las instrucciones afecta el estado de la memoria y los registros. Además, se hace énfasis en la sintaxis y la semántica del lenguaje que se está utilizando. Respecto a la semántica, es fácilmente explicable a partir de las transiciones de estado.

La interpretación de los datos almacenados también puede ser cubierta con programas simples. Por ejemplo, un caracter podría ser tratado como un entero y llevar a cabo operaciones sobre él. Un ejemplo clásico es la conversión a mayúscula de una letra minúscula.

El concepto de asignación es introducido a partir de las instrucciones de carga y almacenamiento. Por otra parte, las instrucciones de entrada permiten generalizar los programas simples que se han escrito hasta ahora. Los estudiantes aprenden la estructura de un programa desde el punto de vista de un proceso que modifica sus entradas para generar salidas.

Puesto que el ensamblador no cuenta con instrucciones para llevar a cabo multiplicaciones o divisiones, los alumnos aprenden a efectuar estas operaciones de alguna manera ingeniosa. En primer lugar, se les enfrenta al problema de multiplicar un número por 2, lo cual llevan a cabo sumando consigo mismo el número en cuestión.

La instrucción de comparación y las de salto condicional permiten introducir el concepto de condición y de control de flujo. La ejecución condicional de instrucciones se enseña a partir de conceptos de bajo nivel, es decir, el cambio en el valor del contador de programa, en combinación con el funcionamiento normal del ciclo de *fetch*. Como primer ejemplo se escribe un programa que lee dos números como entrada y entrega como salida el mayor de ellos.

La ejecución de los programas se visualiza, paso a paso, construyendo tablas de estado en las que, en cada fila, se muestra el contenido de la memoria y los registros antes de ejecutar la siguiente instrucción, es decir, la que está siendo apuntada por el PC. Los alumnos ejercitan esta práctica constantemente, con el objetivo de que puedan seguir sin problema la ejecución de cualquier programa y determinar lo que hace.

Las iteraciones se introducen por necesidad, escribiendo programas en los que es necesario escribir varias veces las mismas instrucciones. Los alumnos se dan cuenta por sí mismos que es posible volver a ejecutar las instrucciones, efectuando un salto. Sin embargo, se hace énfasis en la importancia de una construcción estructurada, evitando llevar a cabo saltos innecesarios en los programas. Se estudia dos tipos de ciclos: Unos del tipo *while*, en los que existe una condición al inicio del ciclo que controla la iteración, y otros del tipo *for*, en los que se utiliza un contador para controlar la cantidad de veces que se debe iterar. También se enseña lo que es un acumulador y la importancia de su inicialización. Finalmente, se estudia las condiciones de término y el problema de los ciclos infinitos.

Con los conocimientos adquiridos hasta este punto, los alumnos pueden escribir programas de mayor complejidad, por ejemplo, calcular una potencia de 2, multiplicar dos números enteros, conteo diferenciado de valores de entrada, y otros.

Como apoyo al proceso de aprendizaje, diseñamos e implementamos un emulador para el lenguaje ensamblador de nuestro computador ficticio (Fig. 3). El emulador basa su funcionamiento en el ciclo de *fetching*, y permite que los alumnos analicen los cambios de estado que se producen como producto de la ejecución de las instrucciones que componen sus programas. Muchos de los componentes del computador son visibles, incluyendo la memoria, los registros PC y R, y el registro de banderas. El emulador fue programado en Java y se encuentra disponible para su uso como aplicación independiente o como *applet*.

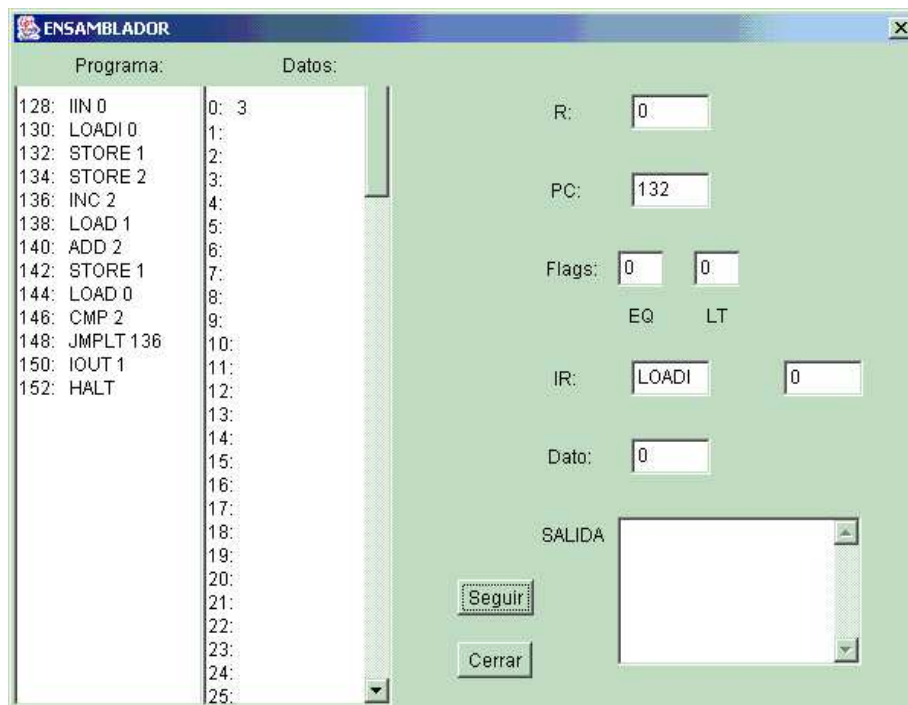


Figura 3: Vista del emulador

4. Primeros resultados

Durante el presente año incorporamos por primera vez la metodología descrita a los 200 alumnos inscritos en la asignatura. El grado de comprensión de la materia fue satisfactorio y los estudiantes trabajaron con entusiasmo en los ejercicios y tareas de programación. Los beneficios fueron evidentes al empezar a programar utilizando Java. En primer lugar, el tiempo requerido para estudiar algunos conceptos se vio disminuido considerablemente. Este es el caso de las condiciones, las instrucciones de decisión, los ciclos y los tipos de datos.

Para percibir el efecto que ha provocado el uso del ensamblador en los estudiantes, condujimos una encuesta muy simple, que contestaron 80 alumnos del curso.

El 46,75 % de los alumnos asegura que logró un nivel alto de comprensión de la materia, mientras que un 40,26 % dice haber alcanzado un nivel medio. El restante 12,99 % dice que su nivel fue bajo o nulo. Esto nos demuestra que el nivel no fue excesivamente alto y que los alumnos de primer año son capaces de comprender los fundamentos de arquitectura de computadores y de programación a bajo nivel que intentamos enseñarles.

Respecto a la influencia que tuvo la programación en ensamblador, respecto a la asimilación del concepto de programa secuencial y a la forma en que se ejecuta un programa, el 80,52 % de los encuestados dicen que se vieron favorecidos en un nivel medio o alto.

También se les consultó sobre la influencia que tuvo la programación en ensamblador para facilitar la comprensión de instrucciones de decisión y de iteración en Java. En estos casos, el 76,62 % y el 79,22 %, respectivamente, aseguraron que se vieron favorecidos en un nivel medio o alto.

Estos resultados nos alientan a pensar que las hipótesis que nos habíamos planteado, respecto al beneficio de introducir en primera instancia un lenguaje de bajo nivel, son acertadas.

5. Conclusiones y Trabajo Futuro

El uso de un lenguaje de bajo nivel se justifica desde el punto de vista de la comprensión del funcionamiento de los computadores. La programación en ensamblador se encuentra en la mayoría de los planes de estudio de las carreras de computación, pero no porque sea una destreza fundamental que deban adquirir, sino por la comprensión que se logra acerca del funcionamiento de los computadores.

Los resultados que hemos obtenido durante el primer año de uso de un lenguaje de bajo nivel en el primer curso de programación de nuestras carreras han sido prometedores. Creemos que hemos logrado una mejoría, respecto a los enfoques anteriores en los que los detalles de arquitectura de los computadores se trataban en forma muy superficial. Los principales beneficios se ven respecto a la comprensión de tópicos que son difíciles de asimilar en forma abstracta, por ejemplo, las variables, los tipos de datos y los punteros a memoria.

El tiempo necesario para estudiar ciertos conceptos en el lenguaje Java se ha visto reducido como producto del conocimiento previo que tienen los alumnos. Esto nos ha permitido tratar ciertos tópicos con mayor profundidad que en años anteriores.

Actualmente estamos analizando la posibilidad de construir un computador que ejecute el conjunto de instrucciones con el que hemos trabajado. La idea es mantener al máximo la visibilidad de los componentes y que los alumnos no sólo puedan identificarlos, sino que puedan también seguir la ejecución de sus programas en un computador real. Este es un proyecto que reportará beneficios a otros cursos de años superiores, por ejemplo, *Arquitectura de Computadores*.

A corto plazo, conduciremos un estudio más formal para tratar de cuantificar los beneficios reales de la nueva metodología. Esto podría llevarse a cabo con la ayuda de grupos de control. Asimismo, es necesario evaluar la influencia de variables externas que pudieran afectar los resultados obtenidos. Para este efecto, es importante mantener un registro histórico de las encuestas contestadas por los alumnos, así como de las variaciones metodológicas que se vayan presentando.

Finalmente, un comentario sobre el aprovechamiento del tiempo. Nuestra asignatura tiene la ventaja de extenderse a dos semestres, lo cual nos permite programar de mejor manera las tres secciones que cubrimos. Sin embargo, los cursos semestrales también pueden beneficiarse de este enfoque pues, como lo mencionamos antes, el conocimiento adquirido al estudiar la arquitectura del computador y la programación en ensamblador, ayuda a avanzar más rápido en las etapas posteriores, utilizando un lenguaje de alto nivel.

Referencias

- [1] Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) and the Association for Computing Machinery (ACM). Final Report of the Joint ACM/IEEE-CS Task Force on Computing Curricula 2001 for Computer Science, Dec. 2001.
- [2] David Cordes, Allen Parrish, Brandon Dixon, Richard Borie, Jeff Jackson, and Patrick Gaughan. An Integrated First-Year Curriculum for Computer Science and Computer Engineering. *IEEE Frontiers in Education Conference*, 1997.
- [3] Edsger W. Dijkstra. On the Cruelty of Really Teaching Computer Science. *Communications of the ACM*, 32(12):1398–1404, Dec 1989.
- [4] Barry Donahue. Using Assembly Language to Teach Concepts in the Introductory Course. *Proceedings of the 19th. SIGCSE Technical Symposium on Computer Science Education*, pages 158–162, Atlanta, Georgia, United States, 1988.
- [5] Jr. E. F. Ecklund. A “Non-Programming” Introduction to Programming Concepts. *Proceedings of the ACM SIGCSE-SIGCUE technical symposium on Computer science and education*, pages 62–64, 1976.
- [6] Guillermo Levine. *Computación y Programación Moderna: Perspectiva General de la Informática*. Pearson Educación, 2001.
- [7] Warren A. Harrison and Kenneth I. Magel. A Suggested Course in Introductory Computer Programming. *Proceedings of the 12th. SIGCSE Technical Symposium on Computer Science Education*, pages 50–56, St. Louis, Missouri, United States, 1981.
- [8] Allen Parrish, Richard Borie, David Cordes, Brandon Dixon, Jeff Jackson, and Russ Pimmel. An Integrated Course for Computer Science and Engineering. *IEEE Frontiers in Education Conference*, pages 11a3–12 – 11a3–17, San Juan, Puerto Rico, 1999.
- [9] Yale Patt. First Courses and Fundamentals. *ACM Computing Surveys (CSUR)*, 28(4es):99, 1996.
- [10] Jeff Penfold and J. Kelly Flanagan. A First Year Computer Organization Course on the Web: Make the Magic Disappear. *IEEE Workshop on Computer Architecture Education*, Vancouver, British Columbia, 2000.
- [11] Gabriela Salazar. El Primer Curso de Computación del Programa de Bachillerato en Computación e Informática de la Universidad de Costa Rica. *IV Congreso Chileno de Educación Superior en Computación*, Copiapó, Chile, 2002.
- [12] Robert W. Sebesta and James M. Kraushaar. TOYCOM - A Tool for Teaching Elementary Computer Concepts. *Proceedings of the 11th. SIGCSE Technical Symposium on Computer Science Education*, pages 58–62, Kansas City, Missouri, United States, 1980.
- [13] Ted Sjoerdsma. An Interactive Pseudo-Assembler for Introductory Computer Science. *Proceedings of the ACM SIGCSE-SIGCUE Technical Symposium on Computer Science and Education*, pages 342–349, 1976.
- [14] J. L. Wolfe. Reviving Systems Programming. *Proceedings of the 23rd. SIGCSE Technical Symposium on Computer Science Education*, pages 255–258, Kansas City, Missouri, United States, 1992.