Arquitectura del Conjunto de Instrucciones Educativa para el Inicio en la Carrera de Ingeniería en Computación

Thiago A. Cruz [1], Hernan F. Kisiel [2], Brian E. Ryberg [1], Jonathan C. Meier [1], Matías G. Krujoski [1], Alicia B. Rendón [1] y Roberto E. Carballo [2]

[1] Facultad de Ingeniería, Universidad Nacional de Misiones (UNaM), Oberá, Misiones, Argentina

[2] IMAM, UNaM, CONICET, FI, Grupo de Investigación y Desarrollo en Electrónica (GIDE), Juan Manuel de Rosas #325, Oberá, 3360, Misiones, Argentina.

cruzthiagoagustin664@gmail.com, hernankisiel@gmail.com, ryberg.brian2@gmail.com, jonny.meier26@gmail.com, matias.krujoski@fio.unam.edu.ar, alibrendon@gmail.com, robertocarballo@fio.unam.edu.ar.

Resumen. En este trabajo se presenta el desarrollo de una arquitectura del conjunto de instrucciones (ISA - Instruction Set Architecture), diseñada especialmente para el uso en ámbito académico de los primeros años de la carrera de Ingeniería en Computación. El objetivo es proporcionar un modelo simple, que sirva como una introducción a los fundamentos de ISA's más complejas que se puedan desarrollar o estudiar a futuro, facilitando así la comprensión de conceptos básicos y un aprendizaje general de manera progresiva a lo largo de la carrera.

Para lograr el cometido, a continuación se presentan tres componentes esenciales: la arquitectura del conjunto de instrucciones (ISA), la microarquitectura asociada a la ISA y un simulador de código ensamblador, este último permite tanto el desarrollo como la prueba de códigos en lenguaje ensamblador por parte del alumno. Estos componentes en conjunto forman un primer escalón sencillo con el cual los estudiantes pueden introducirse al diseño de ISA's, microarquitecturas y, además, a la programación a bajo nivel.

Palabras Clave. Arquitectura del Conjunto de Instrucciones Educativo, Microarquitectura, Código Ensamblador, Fundamentos de Informática, Procesador de un Solo Ciclo

1. Introducción

Existen varios enfoques para introducir a los estudiantes al funcionamiento de las computadoras modernas en las carreras vinculadas a las ciencias de la computación, pudiéndose agrupar estos en dos clasificaciones, enfoques de abajo hacia arriba o de arriba hacia abajo (Yale N. Patt and Kevin J. Compton, 1997).

Los enfoques de abajo hacia arriba consisten en comenzar por enseñar los componentes de la computadora pero desde sus partes constitutivas, por lo que se suele comenzar por presentar en primer lugar el transistor MOSFET, el cual es la base para la construcción de circuitos CMOS y luego con estos cómo se construyen las compuertas lógicas, para seguir con el armado de circuitos combinacionales, circuitos aritméticos y memorias. De esta forma se van construyendo y agrupando los distintos niveles de abstracción que conforman las bases de las computadoras modernas, llegando hasta la microarquitectura de un procesador y su Arquitectura del Conjunto de Instrucciones (ISA – Instruction Set Architecture), e inclusive continuar con la enseñanza de lenguajes de programación assembly y de alto nivel. El objetivo del enfoque de abajo hacia arriba es que el estudiante construya su conocimiento sobre lo que ya sabe, aprendió o va aprendiendo en el transcurso del curso, minimizando los vacíos que puedan haber en el funcionamiento de cada componente y sus interacciones (Sanjay J. Patel et al., 2023).

Los distintos niveles de abstracción que se tienen desde el problema que se quiere resolver usando una computadora (o redes de computadoras), hasta los electrones que fluyen por los circuitos transistorizados de la mismas, se agrupan en lo que se denomina la jerarquía de transformación (Sanjay J. Patel et al., 2023), presentada en la Fig. 1.

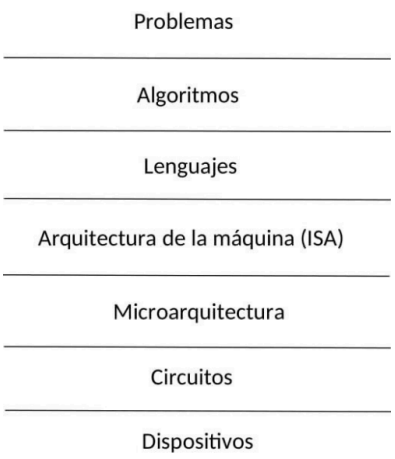

Fig. 1 Jerarquía de Transformación (Sanjay J. Patel et al., 2020).

A diferencia de los enfoques de abajo hacia arriba, los enfoques de arriba hacia abajo se basan en enseñar primero a programar en un lenguaje de alto nivel, con lo cual suele variar mucho el contenido dependiendo de qué lenguaje se enseña. El

argumento en contra del por qué no es bueno este tipo de enfoques es que el estudiante termina memorizando y tratando de aplicar patrones que aprendió en ejercicios previos a los problemas futuros, sin comprender el funcionamiento básico de un procesador y la memoria, lo cual lo lleva a fracasar en la resolución de nuevos problemas. El argumento a favor es que el tiempo para aprender a programar podría ser más reducido que con un enfoque de abajo hacia arriba, sacrificando un conocimiento profundo sobre los componentes de la computadora y su funcionamiento.

Por lo general en los enfoques de abajo hacia arriba se suele utilizar un modelo de máquina (descrito por su ISA) que permite al estudiante entender cómo funciona una computadora, como es la interacción memoria-procesador, la programación a bajo nivel, ya sea en código máquina o el lenguaje assembly de una máquina en particular. Los modelos utilizados generalmente son en base a arquitecturas RISC (*Reduced Instruction Set Computer*, Arquitectura del Conjunto de Instrucciones Reducida), ya que otorgan pocas instrucciones con las que el estudiante debe familiarizarse. Estos modelos suelen estar acompañados de herramientas para simular la ejecución del código, así es posible escribir algoritmos y probar su funcionamiento en un entorno de simulación.

Dentro de los posibles modelos de máquinas más simples para aprender el funcionamiento de las computadoras, se encuentra una propuesta del profesor Yale Patt, denominada LC-3 (*Little Computer 3*, Pequeña Computadora 3), para la cual definió una ISA educativa basada en arquitecturas RISC, con 14 instrucciones y 5 modos de direccionamiento.

Si bien la ISA propuesta por Patt Y. es simple y permite a los estudiantes introducirse en forma más amigable a conceptos como, código máquina, lenguaje assembly, microarquitectura de un procesador, en comparación a si lo tuvieran que hacer con MIPS, ARM o RISC-V (enfoques utilizados en libros de los autores (Harris S. y Harris D., 2007, 2015, 2021)), el estudio de la LC-3 requiere preestablecer las siguientes características de un procesador, las cuales no son tan intuitivas para un estudiante que recién comienza:

- Es necesario un banco de registros.
- Se debe aprender las características de instrucciones de cargar *(Load)* y guardar (*Store*) en 3 diferentes modos de direccionamiento: relativo al contador de programa, indirecto y *base + offset*.
- Se tienen varios campos dentro de los formatos de las instrucciones, ya que se cuenta con el *opcode*, las direcciones destino y fuente dentro del banco de registros, la dirección base en un banco de registros, el bit de dirección (bit 5 en instrucciones de procesamiento), los bits inmediatos (constantes), los bits NZP en instrucciones de salto condicional y los distintos offsets.
- Se deben aprender otras instrucciones que facilitan o permiten implementar funcionalidades que pueden considerarse “más avanzadas” que la implementación de algoritmos básicos, como el manejo de interrupciones (instrucción RTI), llamado a subrutinas (instrucciones JSR, JSRR, RET) y llamadas al sistema operativo (instrucción TRAP).

El hecho de tener que explicar por qué se utiliza un banco de registros presenta desafíos al enseñar estos conceptos a los estudiantes. Esto se debe a que es necesario introducir nociones de jerarquía de memoria, cachés y pipelines, que son las estructuras de hardware que permiten aprovechar las ventajas de incorporar bancos de registros en las máquinas modernas y, en conjunto, son las principales responsables de mejorar el rendimiento.

El objetivo de este trabajo es proponer un prototipo de máquina más simple, que opera directamente desde la memoria y con formatos de instrucción más reducidos que los de la LC-3, permitiendo lograr que en los tiempos de cursado que tiene la materia Fundamentos de Informática de la FIO (60 hs en un cuatrimestre), mejore el nivel de comprensión de los estudiantes en el funcionamiento de una computadora. Con esta motivación se propone un modelo de máquina más simple que la LC-3, que consista de: un acumulador, 7 instrucciones para operar, un formato simple para cada una de estas y solo dos modos de direccionamiento.

El modelo propuesto presenta un inconveniente: la imposibilidad de direccionar toda la memoria con una sola instrucción. Sin embargo, este problema se resuelve en la LC-3, lo que permitirá al estudiante abordarlo en la materia de Arquitectura de Computadoras. La ISA presentada en este trabajo está diseñada para ser el punto culminante del curso de Fundamentos de Informática, impartido en el primer año de la carrera. Parte del objetivo es que en el segundo año, la materia de Arquitectura de Computadoras pueda retomar y expandir naturalmente los conocimientos adquiridos, utilizando la LC-3 como una continuación lógica de lo aprendido previamente.

A continuación se presentará la ISA educativa propuesta junto a su microarquitectura, sumado a un simulador de programación en código *assembly* específico a la misma. Debido al reciente desarrollo e implementación en la cursada actual (2025), aún no se tienen resultados cuantificables, por lo que en las conclusiones se comenta la experiencia de los docentes obtenida hasta el momento y que se utilizará para medir el impacto que tiene la implementación de esta ISA en lugar de la LC-3.

Este trabajo se encuentra organizado de la siguiente forma: 1. Introducción, 2. ISA y su microarquitectura, 3. Simulador de código ensamblador, 4. Ejemplos de programa, 5. Conclusiones y 6. Referencias.

2. ISA educativa y su microarquitectura

2.1. Conjunto de instrucciones y sus formatos

La ISA propuesta en este trabajo ha sido nombrada como ESM (*Educational Simplified Machine*, Máquina Educativa Simplificada), desarrollada como una simplificación de la LC-3 y al igual que la misma, con fines educativos.

En primera instancia del desarrollo de la ESM se planificó contar con instrucciones de procesamiento, memoria y control de flujo de programa, ya que estos tres tipos de instrucciones son los que se encuentran en la LC-3. La diferencia principal con las instrucciones de la propuesta de Patt Y. es que las instrucciones de

memoria mueven datos entre memoria y el acumulador, y no entre memoria y un banco de registros. Esto facilita la operación entre memoria y procesamiento, ya que no se tienen que cargar registros antes de comenzar a operar, pero sacrificando simplicidad en el programa, ya que ahora se requieren más instrucciones de memoria para transferir datos de la misma al acumulador y viceversa.

Para mantener compatibilidad con las instrucciones de procesamiento de la LC-3, se plantea mantener las mismas instrucciones, ADD (suma aritmética), AND y NOT, estando la diferencia en que, debido a la estructura de acumulador con la que trabaja la ALU (*Arithmetic Logic Unit*, Unidad Aritmético Lógica), se tiene un solo operando. En la Fig. 2 se presenta la conexión simplificada de la ALU con el acumulador.

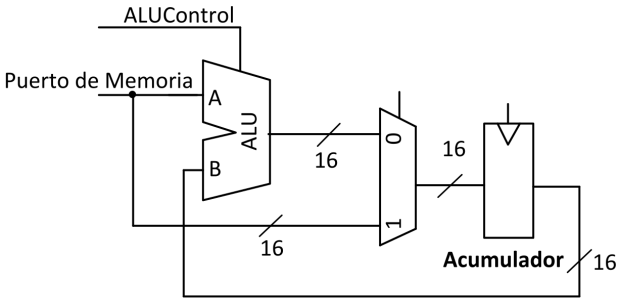

Fig. 2 Conexión de la ALU con la Memoria y el Acumulador

Aquí se puede observar que la ISA es *word addressable*, ya que al cargar el acumulador se operará sobre números de 16 bits.

El formato de las instrucciones de procesamiento se presenta en la Fig. 3, donde es posible observar el campo de opcode (13:15) (*operation code*, código de operación), el bit 12 que hace de bit de dirección (lo que se denomina como *steering bit*) y define el modo de direccionamiento en este caso, pudiéndose utilizar el modo relativo al contador de programa (bit 12 = 0) o inmediato (bit 12 = 1).

	15 14 13	12	11 10 9 8 7 6 5 4 3 2 1 0
ADD$^+$	0 0 0	0	PCoffset12
ADD$^+$	0 0 0	1	imm12
AND$^+$	0 0 1	0	PCoffset12
AND$^+$	0 0 1	1	imm12
NOTA$^+$	0 1 0	0	PCoffset12
NOTB$^+$	0 1 0	1	0 0 0 0 0 0 0 0 0 0 0 0

Fig. 3 Formato de las instrucciones de procesamiento

Las instrucciones ADD y AND cuentan con dos modos de direccionamiento, con bit 12=0 se utiliza el modo relativo al contador de programa, donde se opera el contenido del acumulador con lo que se encuentre en la dirección PC+1+PCOffset12, mientras que con bit12=1 se utiliza el modo inmediato, donde las instrucciones operan el contenido del acumulador con una constante de 12 bits escrita en el campo imm12.

La instrucción NOT tiene dos modos de direccionamiento con bit 12 definiéndolos, el relativo al contador de programa (NOTA) y el modo acumulador (NOTB), haciendo en este último NOT bit a bit sobre el contenido del acumulador directamente.

El multiplexor en la Fig. 2 es para utilizarlo con instrucciones de memoria, ya que se propone poder cargar el Acumulador directamente con lo que provenga de la memoria. Para esto se utilizaría la instrucción LD, que viene del término *load* (cargar). Se optó este nombre para mantener la similitud con la instrucción LD de la LC-3, la cual a diferencia de la ESM permite cargar un registro del banco de registros en la LC-3.

Completando las instrucciones de memoria tenemos ST, proviene de *store* (guardar), el cual transfiere el contenido del acumulador a la memoria en modo relativo al contador de programa.

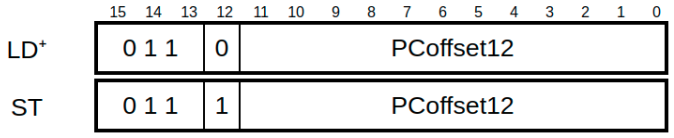

Fig. 4 Formato de las instrucciones de memoria

Notar que el bit 12 en el formato presentado en la Fig. 4 cambia la dirección de hacia dónde irían los datos, con bit 12 = 0 se lleva de memoria a acumulador, y con bit 12 = 1 de acumulador a memoria.

En las figuras Fig. 3 y Fig. 4 el superíndice con un signo + indica que esas instrucciones modifican los bits N, Z, P en el código de condiciones (CC), que se utiliza para comparar con los bits NZP de la instrucción de control de flujo de programa BR (la abreviación proviene de *branch*, ramificación), la cual se presenta su formato en la Fig. 5.

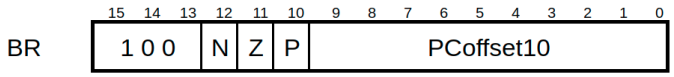

Fig. 5 Formato de las instrucciones de control de flujo de programa

La instrucción BR permite cambiar la dirección del PC (contador de programa) si existe una coincidencia en algunos de los bits NZP con los bits NZP del código de condiciones. En caso de coincidir se accede a la siguiente instrucción en ejecución dada por la dirección PC+1+PCOffset10; en caso contrario, a la correspondiente a PC+1. El PC apunta a un puerto de dirección de memoria, lo cual al leer esta se obtiene una instrucción, teniendo una capacidad de direccionamiento de 2^{16}, por lo que el PC puede valer números expresados en 16 bits.. Leer otros puertos de dirección de memoria mediante las instrucciones LD o ST permite obtener o almacenar datos en la memoria. Las conexiones relevantes se detallan en la siguiente sección, donde se presenta la microarquitectura del procesador que utiliza esta ISA educativa.

Además de estas instrucciones para el cómputo básico de datos que se encuentran en la memoria previo a la ejecución del programa, se agrega la instrucción TRAP para posibilitar subrutinas del sistema operativo que atiendan *system calls* del usuario, como por ejemplo captura de caracteres desde el teclado, impresión en pantalla de caracteres, u otras tareas que el usuario quisiera delegar al sistema operativo. Básicamente estas instrucciones funcionan como un salto incondicional, el cual accede a las primeras posiciones de memoria para saber a que subrutina acceder en particular. Para evitar superponer hardware complejo adicional al que ya se presentará en la próxima sección, no se añadirá el cableado de esta instrucción en el camino de datos, pero si se tiene en cuenta esta instrucción en el simulador de la ESM a describirse en la sección 3.

En la Fig. 6 se muestra el formato de esta, en el cual “trapvect10” nos indicará de qué subrutina se trata, siendo en esta ISA x23 para la entrada de un carácter y x21 para imprimir un carácter en consola (manteniendo estas direcciones de TRAP de la misma forma que se realiza en la LC-3).

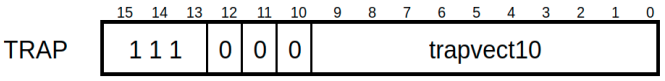

Fig. 6 Formato de las instrucciones de interrupción de programa

2.2. Microarquitectura

Para la microarquitectura del procesador se considera que toda la instrucción se ejecuta en un solo ciclo de reloj (lo que se conoce como procesador de un solo ciclo), pudiendo mapear cada instrucción de la ISA en forma directa al *hardware* que las ejecuta, obteniéndose la representación de la Fig. 7.

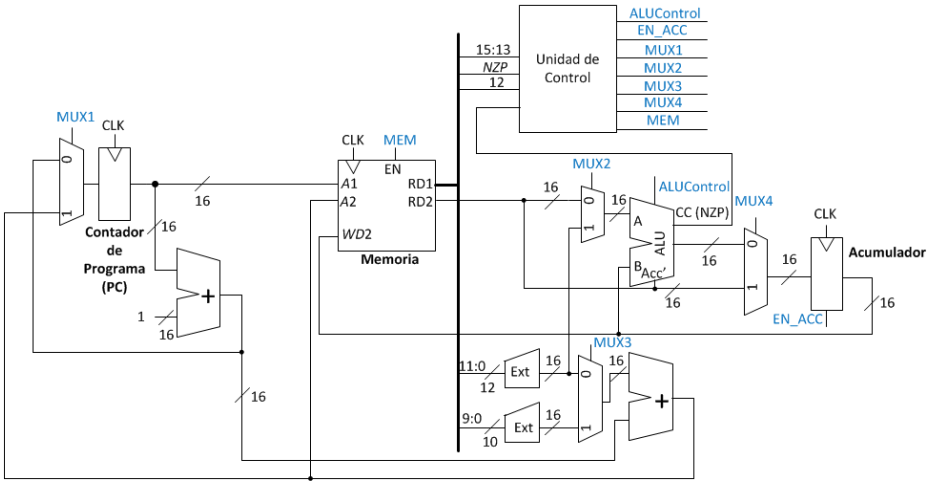

Fig. 7 Microarquitectura del procesador con implementación en un solo ciclo de reloj

Para comparar la simplicidad de la microarquitectura de la ESM con la de la LC-3, que también se implementa en un solo ciclo de reloj (siendo en realidad un subconjunto de la ISA educativa de la LC-3 que se aborda en la asignatura de Fundamentos de Informática), en la Fig. 8 se presentan el camino de datos y la unidad de control de la LC-3 con las siguientes instrucciones: ADD, AND, NOT, LD, ST, LDI, STI, LDR, STR, LEA, BR y JMP.

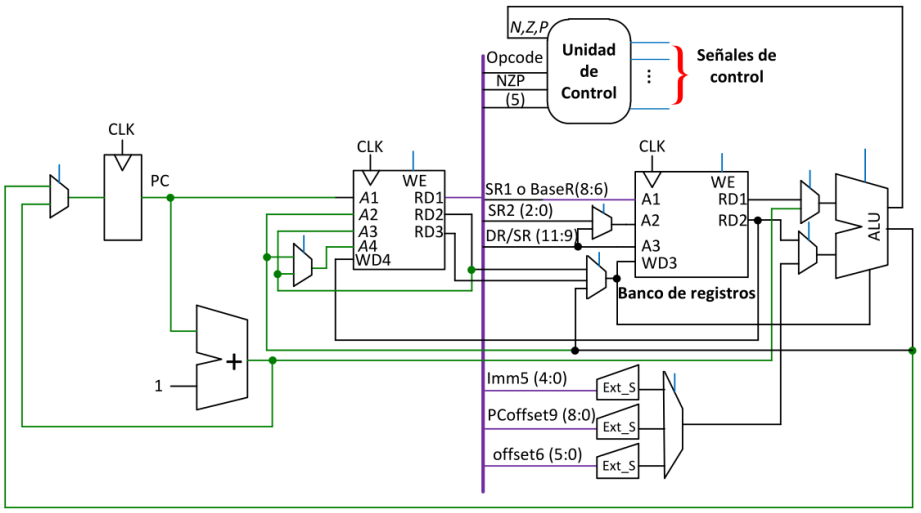

Fig. 8 Microarquitectura del subconjunto de instrucciones de la LC-3 con implementación en un solo ciclo de reloj

3. Simulador de código ensamblador

Para cumplir con el objetivo mencionado al principio, también es necesario crear una APP en la cual los estudiantes puedan experimentar con la programación con código ensamblador (David Salomon et al.,1993). Para ello se añaden las siguientes pseudo instrucciones de ensamblador a la APP para poder manipular parámetros como la ubicación en memoria y la reserva de datos, que se vuelven necesarias a la hora de buscar un flujo dinámico de programación:

- .ORIG: Define la dirección inicial de memoria desde donde comenzará la ejecución del programa. Esta directiva especifica la ubicación de carga del programa en la memoria
- .BLKW: Reserva espacio en memoria para futuros datos sin asignar un valor inicial, para su posterior utilización en el código al guardar datos calculados previamente.
- .FILL: Se emplea para almacenar un valor constante en una ubicación de memoria antes de que el código sea ensamblado. Este valor puede ser un número literal, una dirección de memoria o cualquier constante definida en el código fuente.
- .END: Indica el final del programa, señalando que no existen más instrucciones ni datos a procesar, lo que indica al ensamblador que debe cesar su procesamiento.

El desarrollo del simulador requiere tres componentes esenciales: un analizador léxico (implementado en LEX (Lesk, M. E. and E. Schmidt, 1975)), encargado de identificar los elementos léxicos del código; un analizador sintáctico (implementado en YACC (Johnson, Stephen C., 1975)), responsable de verificar la correcta estructuración de las instrucciones; y, por último, una interfaz gráfica de usuario (GUI) (realizada en python utilizando la librería tkinter (Tkinter - the Python interface for Tk, 2025)), que facilite la interacción con el usuario y proporcione un entorno intuitivo para la programación y depuración del código ensamblador.

Resulta necesario establecer un mecanismo que permita una correcta integración de los lenguajes basados en el lenguaje C (LEX;YACC) con la interfaz escrita en Python. Para lograr esta vinculación, se optó por el uso de *Shared Libraries* (Program Library HOWTO, 2024) (lib.dll en Windows (Creating a Resource-Only DLL, 2024) y lib.so en Linux (Robert A. Gingell, 1998) permitiendo así la compatibilidad en ambos sistemas operativos), estas permiten compilar los analizadores como bibliotecas dinámicas y cargarlas desde Python. Para esta tarea, se emplean módulos como ctypes y cffi, los cuales facilitan la llamada a funciones escritas en C desde el entorno de ejecución de Python.

Finalmente, todos los archivos mencionados se integran en un archivo principal denominado ESM.py, el cual tiene la función de unificar los distintos módulos dentro de la interfaz gráfica y gestionar las llamadas a las funciones implementadas en YACC para la ejecución del código ensamblador. Además, ESM.py se encarga del preprocesamiento del código, eliminando los elementos que pueden alterar la simulación o funcionamiento del mismo (como espacios en blanco, comentarios y saltos de líneas).

En cuanto al manejo de los TRAP se implementan como interrupciones manejadas por dispositivos de I/O, de manera que el usuario no puede observar la tarea del sistema operativo al atender el respectivo TRAP, solamente podrá ver que el programa continúa ejecutando cuando se provee el carácter en el teclado, o cuando se logra imprimir en pantalla.

Se adjunta el repositorio en el cual se encuentra el código fuente del simulador junto a los instaladores de la versión 19.4 del mismo.

https://github.com/thiagocruz664/ESM-Simulator

Cabe recalcar que esta no es la versión final y está sujeta a futuras actualizaciones, las cuales se distribuirán por GitHub.

4. Ejemplos de programa

Con el objetivo de ilustrar el funcionamiento de la aplicación desarrollada, se implementaron dos programas sencillos que permiten demostrar la ejecución de las instrucciones definidas en la arquitectura del conjunto de instrucciones educativa diseñada. El primer programa ejecuta la operación lógica XOR entre dos valores y el segundo realiza la multiplicación de dos números positivos de un dígito ingresados por teclado.

```
.ORIG x2000
0x2000    NOTA NUM1
0x2001    AND NUM2
0x2002    NOTB
0x2003    ST TEMP1
0x2004    NOTA NUM2
0x2005    AND NUM1
0x2006    NOTB
0x2007    ST TEMP2
0x2008    LD TEMP1
0x2009    AND TEMP2
0x200A    NOTB
0x200B    ST RESULT
0x200C    BR nzp RESULT
0x200D    NUM1 .FILL #2
0x200E    NUM2 .FILL #5
0x200F    TEMP1 .BLKW
0x2010    TEMP2 .BLKW
0x2011    RESULT .BLKW
.END
```

Fig. 9 Código de ejemplo 1

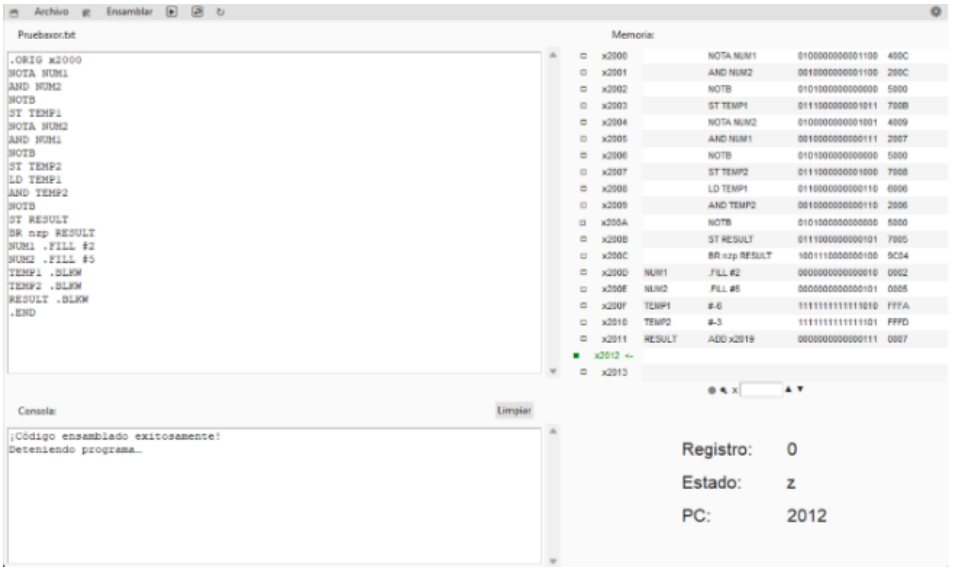

Fig. 10 Simulación de código de ejemplo 1

```
.ORIG x2000
0x2000    TRAP x23
0x2001    ADD #-48
0x2002    ST NUM1
0x2003    TRAP x23
0x2004    ADD #-48
0x2005    ST NUM2
0x2006    LD NUM2
0x2007    ST CONT
0x2008    REDO LD CONT
0x2009    ADD #-1
0x200A    ST CONT
0x200B    LD RESULT
0x200C    ADD NUM1
0x200D    ST RESULT
0x200E    LD CONT
0x200F    BR p REDO
0x2010    LD RESULT
0x2011    ADD #48
0x2012    TRAP x21
0x2013    BR nzp SALTAR
0x2014    NUM1 .BLKW
0x2015    NUM2 .BLKW
0x2016    RESULT .BLKW
0x2017    CONT .BLKW
0x2018    SALTAR .BLKW
.END
```

Fig. 11 Código de ejemplo 2

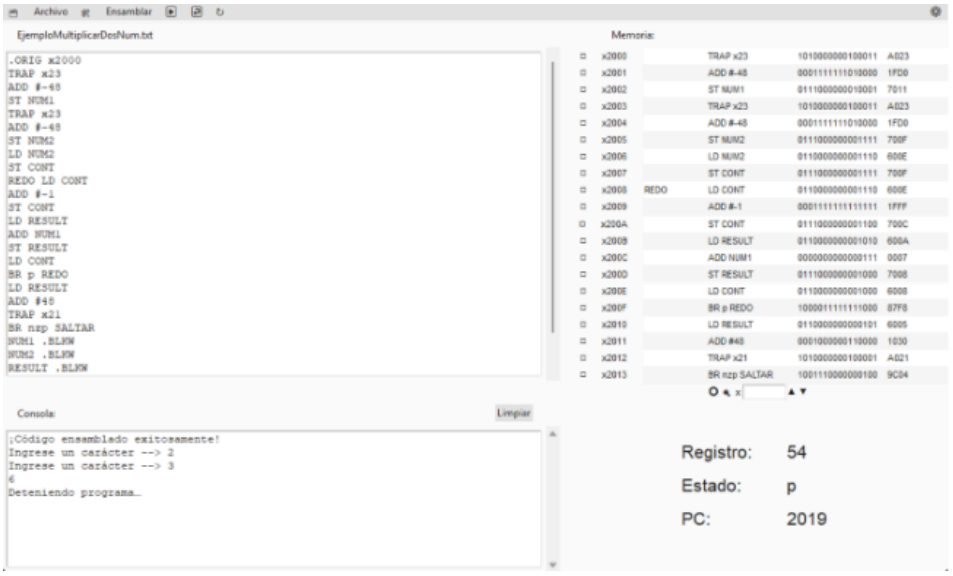

Fig. 12 Simulación de código de ejemplo 2

5. Conclusiones

Se ha presentado una arquitectura del conjunto de instrucciones (ISA) de un procesador educativo, denominada como ESM (Educational Simplified Machine), para ser utilizada en el curso de Fundamentos de Informática de la Carrera de Ingeniería en Computación de la FIO. La ISA contiene 7 instrucciones, 3 de procesamiento, 2 de memoria, 1 de salto condicional y 1 de *system call* con 2 vectores de interrupción respectivos, con las que es posible formular cualquier algoritmo simple de forma práctica, esperando que facilite la comprensión de las características de una ISA a estudiantes que recién se inician en la carrera.

Se contrastó la microarquitectura del procesador propuesto con el que surge del subconjunto de la LC-3 con la que se enseñó hasta el año 2024 en la materia Fundamentos de Informática. Si bien resulta significativamente más reducida la cantidad de conexiones, se puede apreciar que características como el contador de programa, memoria e interfaz con extensión de signo con la ALU siguen manteniéndose en el camino de datos, por lo que se puede considerar un paso previo en la comprensión de la microarquitectura de ISA's más complejos.

Se diseñó una GUI intuitiva en la cual los alumnos pueden desarrollar y testear código ensamblador, logrando comprender los conceptos más básicos de la programación a bajo nivel.

Con un conjunto de instrucciones simples y pedagógicas, una microarquitectura definida y un simulador funcional, la ESM pudo integrarse al cierre de la cursada de Fundamentos de Informática como una herramienta didáctica. Su uso permitió desarrollar al completo una ISA capaz de realizar cualquier tipo de cómputo, sin la necesidad de abordar temas complejos que llevan más tiempo del disponible para dictarlos correctamente sin asumir u obviar diversos conceptos. A impresión de los docentes, esto ayudó a transmitir los conceptos básicos de una manera más simple y comprensible para los alumnos presentes en el cursado.

Se espera poder cuantificar los resultados a partir de analizar datos de las evaluaciones en exámenes finales anteriores y posteriores a Julio del 2025, y con esto determinar si la tasa de aprobación ha mejorado con la implementación de la ESM en el cursado. A espera de estos resultados cuantificables, las primeras impresiones de los docentes han sido positivas y sugieren que el desarrollo avanza en la dirección correcta.

6. Referencias

Lesk, M. E. and E. Schmidt (1975). Lex – A Lexical Analyzer Generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey.

Johnson, Stephen C. (1975). Yacc: Yet Another Compiler Compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey.

Robert A. Gingell, Meng Lee, Xuong T. Dang and Mary S. Weeks (1988). Shared Libraries in SunOS. Sun Microsystems, Inc. 2550 Garcia Ave. Mountain View, CA 94043.

David Salomon (1993). Assemblers and Loaders. The Chicago Manual of Style.

Yale N. Patt and Kevin J. Compton (1997), Introduction to Computing --The Correct (Bottom-up) Approach, Department of Electrical Engineering & Computer Science University of Michigan, Ann Arbor.

Program Library HOWTO, https://tldp.org/HOWTO/Program-Library-HOWTO, last access 2024/11/24.

S. Harris and D. Harris (2007). Digital Design and Computer Architecture. Morgan Kaufmann.

John. L. Hennessy and David. A. Patterson (2012). Computer Architecture: A Quantitative Approach fifth edition. Morgan Kaufmann.

Paxson, Vern, Will Estes and John Millaway (2015). Lexical Analysis with Flex. University of California, Berkeley.

Donnelly, Charles and Richard Stallman (2015). Bison. Free Software Foundation.

S. Harris and D. Harris (2015) Digital design and computer architecture: arm edition. Morgan Kaufmann.

Creating a Resource-Only DLL Microsoft Developer Network Library, https://learn.microsoft.com/en-us/cpp/build/creating-a-resource-only-dll, last access 2024/11/24.

S. Harris and D. Harris (2021) Digital Design and Computer Architecture: RISC-V Edition. Morgan Kaufmann.

Tkinter - the Python interface for Tk, https://python-course.eu/tkinter, last access 2025/10/02.

Biblioteca de vínculos dinámicos (DLL) Microsoft Developer Network Library, https://learn.microsoft.com/es-es/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library, last access 2025/02/23.

Sanjay J. Patel and Yale. N. Patt (2023). Introduction to Computing Systems: from bits & gates to C/C++ & beyond. McGraw-Hill Education