LEGv8, Raspberry Pi 3 y una vieja fórmula

Pablo Ferreyra[1], Agustín Laprovitta[1,3], Delfina Velez Ibarra[1,2], Gonzalo Vodanovic[1,4], Nicolás Wolovick[1]

[1] Universidad Nacional de Córdoba, Córdoba, Argentina.
[2] CONICET, Córdoba, Argentina.
[3] Universidad Católica de Córdoba, Córdoba, Argentina
[4] Universidad Tecnológica Nacional, Facultad Regional Villa María, Villa María, Argentina

Resumen En este trabajo se describe una experiencia sobre la mejora del aprendizaje del Lenguaje Ensamblador (LE) llevada a cabo en la materia Organización del Computador de la Carrera de Licenciatura en Ciencias de la Computación de la Facultad de Matemática, Astronomía, Física y Computación (FAMAF) de la Universidad Nacional de Córdoba (UNC). Los principales objetivos son los de mostrar al alumno la utilidad de adquirir una comprensión de las características y potencialidades de la programación a nivel de LE, y mostrar la factibilidad de incentivar al alumno en el uso del LE usando las capas bajas de abstracción y en contacto directo con el hardware. El trabajo surge a partir del cambio de arquitectura de conjunto de instrucciones (ISA) estudiadas en la materia. Se pasa de la arquitectura MIPS de 32 bits a la LEGv8 de 64 bits, propuesta en el último libro de Patterson-Hennessy. Se aprovecha el hecho que LEGv8 es un subconjunto propio de ARMv8 y la disponibilidad de plataformas Raspberry Pi 3 que incorporan esta ISA. Se diseña un laboratorio con un enunciado mínimo: elaborar una demo gráfica utilizando un *framebuffer* arrancando la máquina sin sistema operativo y ejecutando el código sin ninguna capa de abstracción. En base a los códigos entregados por los alumnos, se realizaron análisis y se generaron estadísticas. Los resultados obtenidos muestran que los objetivos fueron cumplidos, observando un aumento en la participación de la actividad respecto a otras instancias de evaluación.

Keywords: Organización del Computador, lenguaje ensamblador, LEGv8, ARMv8, Raspberry Pi 3

1. Introducción

La materia Organización del Computador (O.C). en la Licenciatura en Ciencias de la Computación de la FaMAF-UNC nace junto a la carrera en 1993. Durante el periodo 1993-1999 aproximadamente, su dictado está a cargo casi exclusivamente del ingeniero Carlos Alberto Marqués, jefe del Grupo de Desarrollo Electrónico e Instrumental (GDEI) de la FaMAF. Luego, en el periodo 2000-2014 aproximadamente, se desarrolla un trabajo conjunto entre el GDEI, los egresados de la Licenciatura en Computación (que empiezan a colaborar en el dictado de las materias) y otros profesionales, complementando el fuerte perfil electrónico inicial de O.C. con un enfoque más general, basado

en la arquitectura básica de un microprocesador y el estudio de su ISA. En la actualidad y desde hace 5 años aproximadamente, estamos en el periodo "moderno" de la cátedra, conformada principalmente por docentes-investigadores del GDEI y otros de la Sección de Computación. En este periodo se empieza a utilizar el libro de texto de Patterson-Hennessy [1] en base a la arquitectura MIPS [2]. Se realizan laboratorios utilizando el simulador MARS [3] que tiene un entorno de programación rico, además de incorporar dispositivos de I/O interesantes como *framebuffers*, generadores de sonido MIDI, entradas por teclado y acceso al sistema de archivos. Aunque la experiencia es positiva, se detecta que el uso de un emulador oculta la experiencia concreta de los alumnos con el hardware desnudo (*bare-metal*) y los aleja de los problemas reales de la programación de dispositivos embebidos, ya que disponen de un entorno de desarrollo que permite un ciclo de edición-ensamblado-ejecución muy rápido y eso sesga a la programación por prueba-error. También se nota que al ser tan clásico el libro y el simulador, la mayoría de las dudas están resueltas en plataformas de intercambio de conocimientos como Stack Overflow y esto reduce el compromiso de pensar los problemas e intentar resolverlos sin demasiada ayuda externa. En el periodo 2013 a 2015 se incorporan proyectos opcionales de programación de Raspberry Pi 1 en ARMv6 para sensar y actuar dispositivos digitales a través de GPIO, siguiendo las ideas de [4]. Aunque la experiencia es buena, los alumnos tienen que aprender ARMv6 por su cuenta a partir de los conocimientos de MIPS. A mediados del 2015 y gracias al Proyecto de Mejoramiento de la Enseñanza en Carreras de Informática (PROMINF) [5] de la Secretaría de Políticas Universitarias del Ministerio de Educación de la Nación, se adquieren 12 Raspberry Pi 3, que permiten tener una base de plataformas de desarrollo y producir un cambio de ISA en la Cátedra. Los elementos estaban dados por dos componentes fuertes, un libro clásico y de altísima calidad que fue portado de MIPS a LEGv8 [6] y una plataforma de desarrollo para los proyectos con capacidad de ejecutar ensamblador ARMv8 de manera *bare-metal*. Respecto a los recursos humanos disponibles, en la cátedra hay un Doctor especializado en Sistemas Embebidos, un Doctor en Ingeniería especializado en Sistemas Tolerantes a Fallas, un Doctor en Computación especializado en Computación de Alto Desempeño y dos Ingenieros Electrónicos también especialistas en Sistemas Embebidos, es decir, un equipo que es capaz de tomar el desafío de cambiar la ISA y que este cambio no impacte de manera negativa en los alumnos. Actualmente en esta materia se inscriben alrededor de 110 alumnos, pero el conjunto de trabajo real es de aproximadamente 80 alumnos.

2. Plataforma de Desarrollo

La idea de la computadora Raspberry Pi surge en el año 2006 luego que Eben Upton y sus colegas del Cambridge Computer Laboratory en Inglaterra detectan una caída en la comprensión del funcionamiento interno de la computadora entre los ingresantes a la carrera de Ciencias de la Computación. Los ingresantes de 1990 eran hobbistas de la programación con conocimientos profundos de organización y arquitectura de computadoras, mientras que los ingresantes del nuevo milenio apenas tenían alguna idea de diseño web [7].

Surge entonces la idea de re-crear el ecosistema de las microcomputadoras de los 80s que permitían una comprensión total del hardware/software y fomentaban la programación y modificación de la computadora, en vez del simple uso del software. El retroceso de la computación como ciencia en las escuelas estaba causando problemas en el ámbito universitario y había que hacer un esfuerzo para revertir la situación. En 2006 se propone una computadora de 25USD hecha con un microcontrolador ATmega644 con 512 KiB de RAM y salida de video compuesto de 320x200x8 generado por software [8], algo similar a proyectos del 2002 como la SX Game System que estaban basadas en microcontroladores PIC [9]. Aunque Eben Upton deja la academia y empieza a trabajar en Broadcom, la idea no desaparece, sino que muta con la aparición de systems-on-a-chip (SoC) cada vez más poderosos y de precio accesible que permitirían el uso de computadoras a chicas y chicos que inicialmente no solo quieren programar. En 2012 aparece Raspberry Pi 1 alrededor del BCM2835 (ARMv6Z), un SoC para *media players* (Roku por ejemplo) y, aunque de manera lenta, el progreso continúa con Raspberry Pi 2 en 2015, utilizando un BCM2836 (ARMv7A) y con Raspberry Pi 3 en 2016, con un BCM2837 (ARMv8-A) que es ya un procesador de 64 bits en contrapartida a los anteriores que eran de 32 bits. La potencia de cómputo de una Raspberry Pi 3 se equipara a la de un Pentium 4, pero casi dos órdenes de magnitud menos en disipación de calor. Todos los modelos mantuvieron el procesador de video, un Broadcom VideoCore IV que aunque incorpora una unidad de procesamiento SIMD cuádruple (QPU) y es capaz de procesar números de punto flotante de simple precisión a 28 GFLOPS, tiene un *framebuffer* de video fácilmente configurable. La plataforma ARM es tan popular que la mayoría de las distribuciones de GNU/Linux incorporan *cross-compilers*, *cross-assemblers* y *cross-debuggers* tanto para 32 como para 64 bits, `arm-none-eabi-*` y `aarch64-linux-gnu-*` respectivamente. Existen además pruebas de concepto que muestran cómo generar un sistema de archivos FAT32 sobre una tarjeta *microSD* para que la Raspberry Pi 3 cargue en la memoria `0x0000` un programa en código de máquina de ARMv8 que utiliza el *framebuffer* [10] y que se compila en la ubicua plataforma Linux/x86_64.

3. Arquitectura del Conjunto de Instrucciones LEGv8/ARMv8

La materia se divide en dos partes, la primera está a cargo de un Ingeniero Electrónico y produce un recorrido *bottom-up* que parte de compuertas básicas hasta el diseño de circuitos secuenciales, combinacionales y mapeos de memoria. La segunda parte está a cargo de un Científico de la Computación y tiene un recorrido *top-down*, que parte de la ISA de LEGv8, pasando por el código de máquina y finalizando en la implementación de la CPU que ejecuta ese conjunto de instrucciones a partir de todos los módulos vistos en la primera parte. El objetivo fundamental es que los estudiantes comprendan cómo funciona una CPU.

LEGv8 es una ISA de tipo RISC *three-operand/load-store* con un ancho fijo de instrucción de 32 bits, que es un subconjunto propio de ARMv8 con algunas modificaciones para facilitar la generación manual de código de máquina, como por ejemplo los operandos inmediatos de las instrucciones aritméticas y lógicas. Se evita también utilizar el *barrel shifter* para el tercer argumento. Hay más simplificaciones. No existen

los operadores de carga y almacenamiento en memoria escalados ni los pre y post incrementos en estos mismos operadores. La ISA es extremadamente sencilla, a tal punto que en una cartilla A5 se resume todo el conjunto de instrucciones, su semántica y la codificación en bits. Esto se conoce como la *green-sheet* y es la hoja de ruta que los estudiantes deben tener durante toda la segunda mitad de la materia. LEGv8 tiene 31 registros de 64 bits de propósito general (`X0` a `X30`), incorpora un registro de solo lectura que siempre está a cero (`XZR`), un registro de acceso limitado apuntando a la instrucción que se está ejecutando (`PC`) y un registro de estado (`PSR`) que solo se puede leer para controlar el flujo de ejecución a través de saltos condicionales y solo es escrito por las instrucciones aritmético lógicas con el postfijo `S` de *set flags*. El set de instrucciones tiene la típica división de los procesadores RISC: aritmético-lógicas, memoria y control. Las instrucciones aritmético-lógicas son las usuales tienen dos modificadores independientes de las que surgen cuatro variedades, estos modificadores son si la instrucción tiene operando inmediato y si la instrucción modifica las banderas de estado. También hay instrucciones de desplazamiento. Las instrucciones de transferencia de datos entre el banco de registros y la memoria RAM pueden ser de 64, 32, 16 y 8 bits que se corresponden con double-words, words, half-words y bytes con un registro base y un desplazamiento inmediato. Para modificar el flujo de control hay instrucciones de salto relativo de diferentes rangos con y sin condiciones. También incorpora una instrucción de cambio de `PC` con almacenamiento previo de la dirección de retorno en `X30` o *link register* para implementación de subrutinas con un nivel de anidamiento.

Instrucción	**Semántica**
`ADD X0, X1, X2`	$R[X0] = R[X1] + R[X2]$
`SUBI X0, X1, #42`	$R[X0] = R[X1] - 42$
`LSL X0, X1, #16`	$R[X0] = 2^{16} R[X1]$
`LDUR X0, [X1,#8]`	$R[X0] = M[R[X1] + 8]$
`STUR X0, [X1,#0]`	$M[R[X1] + 0] = R[X0]$
`CBNZ X0, 1024`	if $(X0 \neq 0)$ $PC = PC + 4096$ else $PC = PC + 4$
`B -8192`	$PC = PC - 32768$
`BL 6`	$R[X30] = PC + 4; PC = PC + 24$

Aunque no existen ensambladores y desensambladores de LEGv8, si se tiene el suficiente cuidado en las diferencias, se puede utilizar las herramientas de desarrollo del GNU Compiler Collection: `as`, `objdump`, `strip`, `gdb`, `gcc`. Un ejemplo típico para ensamblar y desensamblar:

```
$ echo "ADD X0, X1, X2" | aarch64-linux-gnu-as -- && aarch64-linux-gnu-strip ./a.
    ↪ out && aarch64-linux-gnu-objdump -d ./a.out

./a.out:     file format elf64-littleaarch64

Disassembly of section .text:
0000000000000000 <.text>:
   0:    8b020020     add    x0, x1, x2
```

La ISA es moderna, ya que fue anunciada en Octubre de 2011 y aún está penetrando en el mercado de los teléfonos inteligentes, *media players* y computadoras de bajo consumo. La perspectiva es que todos los dispositivos ARM de 32 bits, salvo las arquitecturas de procesadores orientadas a microcontroladores (Familia Cortex-M), se pasarán a esta ISA, por lo tanto el estudio de ARMv8 es una apuesta a futuro.

4. Proyecto

Con el fin que los alumnos puedan visualizar en pantalla los resultados de la ejecución del código ensamblador, se diseña un laboratorio donde se aprovecha el puerto HDMI de la Raspberry Pi 3 y, de esta forma, se prescinde de la incorporación de electrónica de interfaz. Para esto se hace uso del *framebuffer* a bajo nivel y se altera la secuencia de inicio del sistema, cargando el código generado en reemplazo del kernel del sistema operativo. La plataforma Raspberry Pi 3 soporta *framebuffer* para el manejo gráfico en su Video Core (VC). Para esto, hay que realizar una inicialización del VC por medio de un servicio implementado para comunicar el CPU con el VC llamado *mailbox*. Un *mailbox* es uno o varios registros ubicados en direcciones específicas del mapa de memoria (en zona de periféricos) cuyo contenido es enviado a los registros correspondientes de control/estado de algún periférico del sistema. Luego del proceso de inicialización del VC via *mailbox*, los registros de control y estado del VC pueden ser consultados, como por ejemplo la dirección de memoria donde se ubica el inicio del *framebuffer*.

Vale recordar que la plataforma Raspberry Pi 3 está basada en un SoC BCM2837, que incluye un CPU ARM Cortex A53 de cuatro núcleos. Para determinar cuál de los cuatro núcleos ejecutará nuestro código utilizamos el identificador del núcleo mediante el uso de un registro de función especial (`MPIDR`, *Multiprocessor Affinity Register*) al principio del programa y ponemos a ejecutar un lazo infinito todos los núcleos salvo el 0, donde correrá la demo. Dado que excede los contenidos de la materia, se entrega a los alumnos un código que realiza las tareas de inicialización explicadas anteriormente, junto con un programa de ejemplo a modo de plantilla. Este código inicializa el *framebuffer* en 512 píxeles por 512 píxeles y formato de color RGB de 16 bits. En el formato RGB (ver Fig. 1), el color negro se representa con el valor `0x0000`, el blanco con `0xFFFF`, el rojo con `0xF800`, el verde con `0x07E0`, y el azul con `0x001F`.

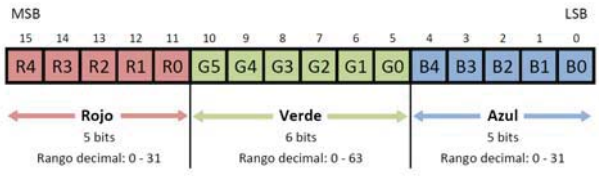

Figura 1: Formato de color RGB de 16 bits

En base a la configuración dada, el *framebuffer* queda organizado en palabras de 16 bits (2 bytes), cuyos valores establecen el color que tomará cada píxel de la pantalla. La palabra contenida en la primer posición del *framebuffer* determina el color del primer píxel (indicado con 0 en la Fig. 2), ubicado en el extremo superior izquierdo de la pantalla, incrementando su numeración hacia la derecha en eje X (columnas) hasta llegar al píxel 511. De esta forma, el píxel 512 representa el primer píxel de la segunda línea (fila 1).

Fila\Columna	0	1	2	...	...	509	510	511
0	0	1	2	...	...	509	510	511
1	512	513	514	...	...	1021	1022	1023
2	.			...				
...	.				...			
509	.					...		
510	.						...	
511	.							...

Figura 2: Distribución de píxeles en una pantalla con resolución de 512×512

Debido a que la palabra que contiene el color de cada píxel es de 16 bits, la dirección de memoria que contiene el estado del píxel N se calcula como se indica en Eq. 1, siendo N el número de píxel, mientras que X e Y indican el número de columna y fila en pantalla respectivamente.

$$\text{colorpixel}_N = M[\text{framebuffer}_{base} + 2N] \quad \text{donde } N = 512Y + X \tag{1}$$

El proyecto que se entrega a modo de ejemplo (Fig. 3), además de la configuración del *framebuffer*, contiene el código necesario para escribir la palabra `0xF800` en todas las posiciones de memoria correspondientes al *framebuffer*. Al compilar dicho código y renombrarlo como `kernel8.img` en la Raspberry, se genera una pantalla de 512×512 píxeles de color rojo:

```
.globl app
app:
    //---------------- EJEMPLO ------------------------------
    // Registro X0 contiene la direccion base del framebuffer

    mov w10, 0xF800     // X10[15..0] = 0xF800 -> color ROJO
    mov x2, 512         // X2 = 512 -> Tamano en Y (filas)
loop1:
    mov x1,512          // X1 = 512 -> Tamano en X (columnas)
loop0:
    sturh w10, [x0]     // M[X0] = X10[15..0] -> Se establece el color del pixel
    add x0, x0, 2       // X0 = X0 + 2 -> Se posiciona X0 en el proximo pixel
    sub x1, x1, 1       // X1 = X1 - 1 -> Decrementa el contador de columna
    cbnz x1, loop0      // Si no es el fin de la columna, salta a loop0
    sub x2,x2,1         // X2 = X2 - 1 -> Decrementa el contador de filas
    cbnz x2,loop1       // Si no es la ultima fila, salta a loop 1
    //-------------------------------------------------------
    // Loop Infinito
InfLoop:
    b InfLoop
```

Figura 3: Código de ejemplo para poner la pantalla en rojo

La primer consigna que se da a los alumnos, organizados en grupos de tres, es la de escribir un programa en assembler ARMv8 sobre el código de ejemplo dado, que pinte la pantalla completa dividida en tres secciones horizontales de igual tamaño, una roja, una verde y una azul. Con esta primer consigna simple se espera que los

alumnos comprendan y asimilen la organización y funcionamiento del *framebuffer*. El proyecto culmina con el desarrollo de un programa en ensamblador ARMv8 donde se espera que se genere un efecto gráfico llamativo con movimiento, con el espíritu de la *demoscene* de los 80s, una forma de expresión cultural a través de medios digitales [11]. Para evaluar, se introduce un mínimo de restricciones: la duración de la secuencia debe ser de al menos 10 segundos, y el patrón generado debe ser repetitivo y no aleatorio. Además, se considera deseable la utilización de la gama de colores completa (paleta de $2^{16} = 65536$ colores) y se valora la relación entre el efecto logrado y el tamaño del código generado.

5. Resultados

El laboratorio tuvo un total de 10 horas de clases asistidas por los docentes, donde los estudiantes trabajaban y consultaban. El periodo de trabajo fue de un mes. Las 12 plataformas Raspberry Pi 3 se pusieron a disposición, para que los estudiantes, además de las 10 horas disponibles de laboratorio, pudieran continuar trabajando en sus casas. La demanda fue grande y se tuvieron que prestar por turnos, siguiendo un esquema *round-robin* con un tiempo límite por grupo. Finalmente se presentaron 21 trabajos de grupos de hasta tres personas, donde aprobaron 57 estudiantes.

Se observó gran participación y motivación de los estudiantes. Los principales indicadores fueron la demanda de las 12 Raspberry Pi 3, el entusiasmo que se veía en las 10 horas de trabajo asistido por los docentes y sobre todo que la cantidad de estudiantes que presentaron el laboratorio (57) superó a la cantidad de estudiantes que rindieron el segundo parcial (49), que evaluaba habilidades similares respecto al lenguaje ensamblador. También se pudo ver la motivación en que al menos cinco estudiantes adquirieron la plataforma para desarrollar más cómodamente la demo y luego seguir trabajando en ella.

El análisis de los trabajos entregados muestran casos destacados, como la implementación de el Conway's Game of Life, la impresión de dígitos 7-segmentos para mostrar mensajes, impresión de dígitos 7-segmentos con el número binario del color del mensaje, barras de colores siguiendo la secuencia de Fibonacci y complejos dibujos bitmap como el logo de Raspberry Pi superpuesta a la demo gráfica. Algunas de ellas se pueden ver en la Fig. 4. Claramente estos casos superaron la espectativa y mostraron que el laboratorio capturó la metáfora *low-floor/high-ceiling/wide-walls* [12].

Se realizó un análisis cuantitativo del uso de la ISA a fin de buscar evidencia del desempeño del proceso de enseñanza/aprendizaje. En la Fig. 5 se muestra un desglose de las instrucciones de la ISA utilizadas, ordenadas por frecuencia de aparición, tanto para el primer ejercicio de la bandera tricolor como para el segundo de la demo. Se observan varios fenómenos, el primero es que en el segundo ejercicio se cubrió más la ISA, otro importante es que en este ejercicio aparecen picos en instucciones específicas: `add`, `bl`, `movk` y `lsl`. Todas menos `bl` tienen explicación en la necesidad de mayor cantidad de cálculos para lograr efectos más vistosos.

Este fenómeno también se puede ver en la Fig. 7 donde mostramos el uso de las instrucciones por tipo, dividido en aritmético/lógicas, memoria y flujo de control.

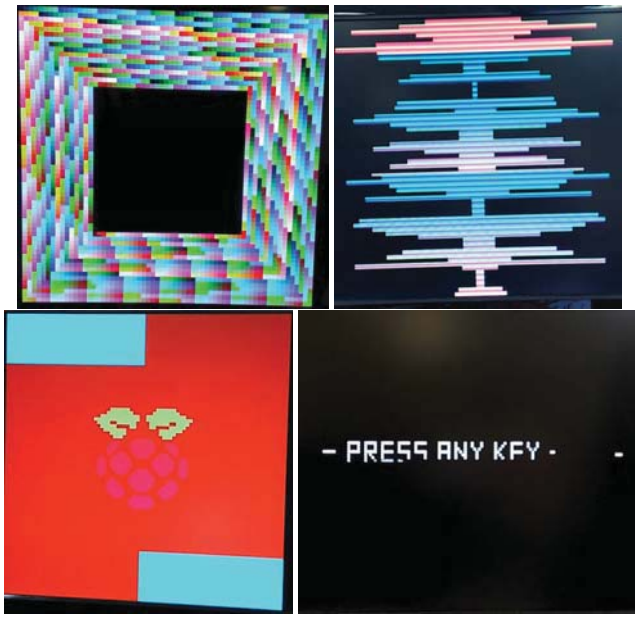

Figura 4: Cuatro demos en acción

En la Fig. 6 graficamos la diversidad de uso de la ISA, mostrando el porcentaje utilizado en los proyectos de instrucciones que no pertnenecen a LEGv8 y si a ARMv8. Aunque no fueron dadas en clases, las instrucciones de ARMv8 [13, 14] fueron investigadas y utilizadas por los estudiantes.

Finalmente notamos que el objetivo de la consigna "el efecto logrado vs. la cantidad de líneas de código", se logró a través del uso de modularización de código mediante el uso del par de instrucciones `bl` y `br X30` para saltar a una subrutina y retornar (Fig. 8). Resulta importante que la modularización surgió de manera natural como mecanismo para disminuir la cantidad de líneas de código y manejar la complejidad de la codificación.

6. Conclusiones

Para verificar el cumplimiento de los objetivos planteados inicialmente, todos los alumnos que realizaron el proyecto debieron pasar por una presentación y defensa oral del mismo. En dicha presentación, se realizaron preguntas para determinar si habían adquirido una comprensión de las principales características y potencialidades de la programación a nivel de LE y su utilidad. También se realizaron observaciones para determinar el grado de motivación y predisposición de los alumnos para usar de las capas bajas de abstracción y en contacto directo con el hardware. En base a éstas herramientas y el análisis de los códigos en LE desarrollados por los alumnos, estamos en condicio-

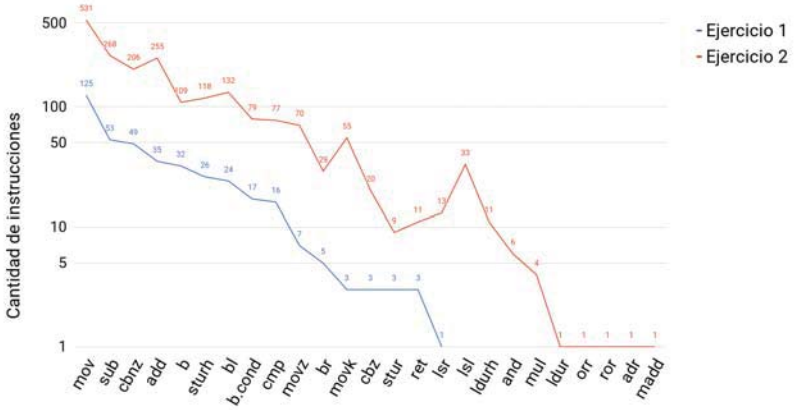

Figura 5: Histograma de cantidad de instrucciones ordenadas por frecuencia

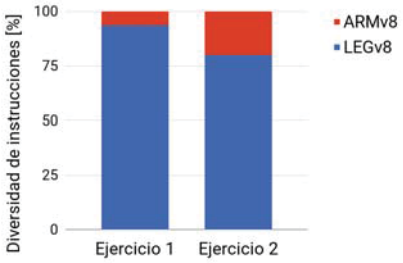

Figura 6: Diversidad en el uso de la ISA

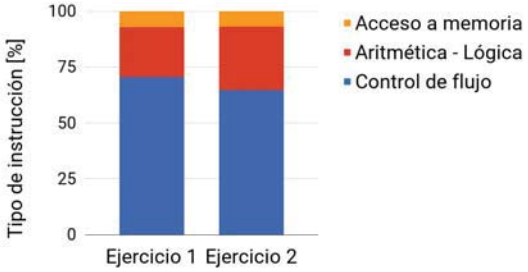

Figura 7: Tipos de instrucciones de la ISA

nes de afirmar que los objetivos mencionados se cumplieron. Otros aspectos a resaltar incluyen, pero no se limitan, a los siguientes:

- Se utilizó una nueva ISA con mucha perspectiva de aplicación a los sistemas embebidos actuales y futuros.
- Se trabajó por medio de experiencias prácticas y concretas, planteando un desafío interesante para los estudiantes.
- La plataforma de desarrollo también resulta altamente motivacional para los alumnos, porque para la mayoría de ellos resultó ser el primer contacto y experiencia con hardware corriendo directamente sobre el metal y a la vista, es decir, con un sistema embebido.

En perspectiva y en resumen, se alcanzó el mismo objetivo que planteaba Upton en su propuesta original de 2006 [8]: hacer una demostración gráfica en LE directamente sobre la CPU y sin el agregado de ninguna capa de software. Tal vez la razón del éxito es

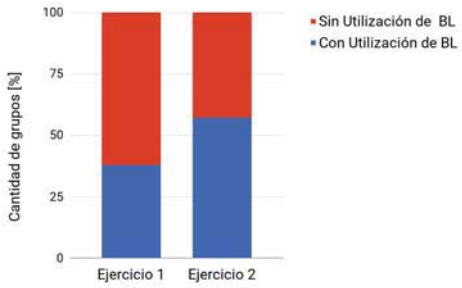

Figura 8: Uso de *branch-and-link* en los grupos

que la vieja receta sigue vigente, aún cuando usamos una ISA moderna como ARMv8 y una plataforma atractiva y poderosa como la Raspberry Pi 3.

7. Agradecimientos

A Martín Marcucci de UCC por la ayuda para compilar `gdb` para ARMv8 en X86_64. A Felipe Manzano que nos ayudó a desnudar los binarios de toda información de debugging. Al PROMINF [5] por los fondos para comprar las Raspberry Pi 3.

Referencias

1. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. 5th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2013)
2. Przybylski, S.A., Gross, T.R., Hennessy, J.L., Jouppi, N.P., Rowen, C.: Organization and VLSI implementation of MIPS. Technical report, Stanford, CA, USA (1984)
3. Missouri State University: MARS (MIPS assembler and runtime simulator). `http://courses.missouristate.edu/KenVollmar/mars/`
4. Clements, A.: ARMs for the poor: Selecting a processor for teaching computer architecture. (2010)
5. Ministerio de Educación de la Nación: Proyecto de mejoramiento de la enseñanza en carreras de informática (PROMINF). `http://www.educacion.gob.ar/secretaria-de-politicas-universitarias/seccion/168/mejora-de-la-ensenanza` (2016)
6. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design, ARM Edition: The Hardware/Software Interface. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2016)
7. Raspberry Pi Foundation: About an ARM GNU/Linux box for $25. Take a byte! `http://web.archive.org/web/20120220051631/https://www.raspberrypi.org/about/` (2012)
8. Raspberry Pi Foundation: Raspberry Pi – 2006 edition. `https://www.raspberrypi.org/blog/raspberry-pi-2006-edition/` (2017)
9. Gunée, R.: SX Game System. `http://www.rickard.gunee.com/projects/video/sx/gamesys.php` (2002)

10. Lemon, P.: Raspberry Pi Bare Metal Assembly Programming. https://github.com/PeterLemon/RaspberryPi/tree/master/HelloWorld/CPU (2017)
11. Polgár, T.: Freax, The History Of The Computer Demoscene. CSW-Verlag (2005)
12. Resnick, M., Silverman, B.: Some reflections on designing construction kits for kids. In Eisenberg, M., Eisenberg, A., eds.: IDC, ACM (2005) 117–122
13. ARM Limited: ARMv8 Instruction Set Overview (2011)
14. Franchin, M.: ARMv8-A A64 ISA Overview, 64-bit Android on ARM. Campus London (2015)