Facultad de Estudios Superiores

Acatlán

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROPUESTA DE HERRAMIENTAS PARA MEJORAR LA COMPRENSIÓN DE LA ARQUITECTURA DE COMPUTADORAS Y SU PROGRAMACIÓN

TESIS Y EXAMEN PROFESIONAL QUE PARA OBTENER EL TÍTULO DE

LICENCIADO EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA:

RUBÉN DANIEL GUTIÉRREZ CRUZ

ASESORA: DRA. MARÍA DEL CARMEN GONZÁLEZ VIDEGARAY

ABRIL 2015

Santa Cruz Acatlán, Estado de México

Quiero dedicar este espacio para mencionar y agradecer a todas aquellas personas que formaron parte de este logro.

En primer lugar quiero agradecer al Padre por el gran regalo de la vida y por hacer este logro parte de su plan.

A mis padres Rubén y Sandra por el constante apoyo que siempre me dieron, por sus lecciones de vida y porque a pesar de que nadie enseña a ser padres, ellos han sido los mejores. Gracias por ese amor, esa paciencia, por cada vez que se sentaron a platicar conmigo sobre lo bueno y malo de la vida, y sobretodo por el excelente ejemplo de vida.

A mi hermano, Luis, quien siempre estuvo motivándome para alcanzar este logro.

A Jonatan, por su constante apoyo y motivación, por creer en mí en esos momentos difíciles, y por ser una persona verdaderamente maravillosa con quien siempre he podido contar para todo.

A quien fuera mi profesor de matemáticas, Andrés Gutiérrez, quien gracias a su excelente labor como docente, adquirí esa pasión por las matemáticas.

A Soraya, por todas esas charlas y consejos, por todos estos años de amistad, por estar siempre en los buenos y malos momentos y por ser una gran persona.

A mi asesora MariCarmen por su apoyo incondicional, su paciencia, sus consejos, y por su maravillosa labor docente que ha dejado una gran huella en mi. Si no fuera por ella este trabajo no habría comenzado.

Les agradezco de todo corazón.

Índice general

Hipótesis

La creación de un conjunto de herramientas de desarrollo que se basen en una máquina simple ayudará a quien las use a comprender de mejor manera la arquitectura de computadoras y su programación.

Objetivo

Demostrar que la creación de un conjunto de herramientas de desarrollo con base en una máquina simple ayudará en la correcta comprensión de la arquitectura de computadoras y su programación, independientemente de los lenguajes existentes.

Introducción

En este trabajo se hace una propuesta de herramientas con un fin didáctico. Para poder entender las razones de esta propuesta, es necesario entrar en contexto tanto histórico como técnico. A continuación se exponen dichos contextos.

Contexto histórico

Hoy en día, la gran mayoría de las personas estamos completamente familiarizados con el enorme avanze de la tecnología. Sin embargo, pocos conocen los detalles de esa historia. Y menos son las personas que conocen las decisiones que se tomaron para llegar a los niveles tecnológicos de hoy en día.

Para esto expone un pequeño resumen de los eventos históricos que resultan relevantes para este trabajo.

La historia de la computación es larga e interesante, pero quisiera partir desde un evento muy importante: la invención del primer microprocesador. El primer microprocesador se le atribuye a Intel con su Intel 4004 [Bre01]. Obviamente para llegar a esta etapa hubo muchas más invenciones y descubrimientos como los transistores, los circuitos integrados, etcétera.

A partir de este primer microprocesador vinieron muchos otros. Por parte de Intel siguieron el 8008, el 8080, y los célebres x86 entre muchos otros. Otras compañias también entraron en el mercado de los procesadores como Motorola, Zilog, Fairchild, entre otros.

Básicamente, cada nuevo procesador resolvía problemas que tenían los anteriores y aparte venían con mayor velocidad, capacidad, alguna nueva tecnología, entre otras cosas.

Desde la creación del primer microprocesador se empezaron a usar los primeros lenguajes ensambladores que no eran más que un conjunto de instrucciones que realizaban una acción sobre el microprocesador.

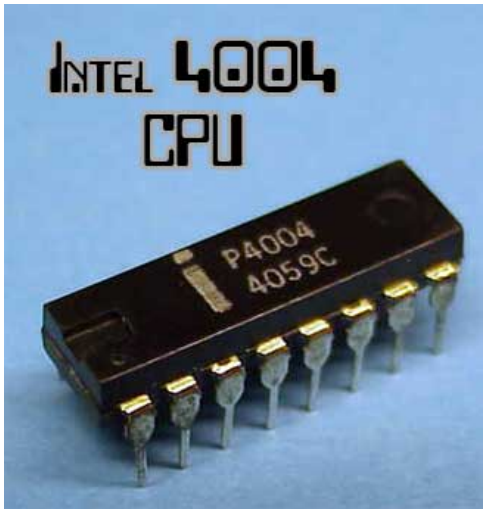


Figura 1: Procesador Intel 4004. Fuente: Sitio web Cultura Apple

Con el tiempo se necesitaron programas más grandes y complejos, y el lenguaje ensamblador no fue suficiente para poder crear programas bien ordenados, estructurados y sobretodo sin errores. Entonces se crearon los primeros lenguajes de programación de alto nivel. Entre estos lenguajes podemos encontrar a Pascal, Fortran, C, y otros.

Ya con nuevos lenguajes que nos ayudan a programar mejor viene un bum en el desarrollo del software, y con más software también viene más hardware (y viceversa). Llegó un momento en que se volvió muy complicado hacer un programa que funcionara igual en todas o por lo menos, la mayoría de las computadoras, entonces a problemas grandes, soluciones simples: Se crean los primeros sistemas operativos.

Los sistemas operativos en un principio no eran más que una capa que administraba el hardware de una computadora. Sin embargo, como el uso de un sistema operativo se volvió tan común y finalmente necesario, muchas compañías tomaron la oportunidad de lucrar con este tipo de software, el software del sistema. Entonces, cada compañía empezó a agregar más software a su sistema operativo que lo diferenciara de los otros. Siguiendo ese camino es como llegamos a los sistemas operativos de hoy, que más que ser una capa de abstracción, ya son un paquete completo de software y utilidades para que el usuario no tenga que preocuparse por el funcionamiento de su computadora.

Para poder lidiar con el crecimiento de la necesidad de software se han

seguido inventando nuevos lenguajes de más alto nivel como C++, Python, Java, entre muchos otros, nuevas metodologías, incluso surgió la ingeniería de software que estudia precisamente las diferentes metodologías para crear software.

Hasta aquí termina el resumen ya que es más que suficiente para poner en contexto el objetivo de este trabajo. Y para lograrlo, hay que mencionar lo siguiente.

- Cada vez que la complejidad llega a cierto límite, se crea una nueva capa de abstracción.
- Con cada nueva capa de abstracción los desarrolladores de software nos alejamos más del hardware.
- Entre más alejados están los desarrolladores del hardware menos saben de las verdaderas capacidades de una computadora.

Citando a algunos autores:

> La comprensión de la interacción entre el hardware y el software y sus compensaciones es necesario para cualquier especialista en computación [HVZ02].

> La tecnología de los computadores moderna necesita que los profesionales de todas las especialidades de la informática conozcan el hardware y el software. La interacción entre estos dos aspectos a diferentes niveles ofrece, al mismo tiempo, un entorno para la compresión de los fundamentos de la computación. Independientemente de que su interés principal sea el hardware o el software, la informática o la electrónica, las ideas centrales de la estructura y el diseño del computador son las mismas [PH11].

El problema está en que cada vez se aprovechan menos las posibilidades de una computadora. Por ejemplo en [ZD04] los autores mencionan que hay que saber optimizar tanto en el nivel alto(lenguajes de alto nivel) como en el bajo nivel (refiriéndose a usar ensamblador).

> El hardware de hoy en día esta organizado de manera más eficiente. Sin embargo, uno puede llevar al límite una tarjeta gráfica de alta gama muy facilmente. En conclusión, aun tiene sentido hacer ciertas optimizaciones tanto en el alto nivel como en el bajo en ciertos lugares. [ZD04].

Por otro lado, en el área de seguridad informática los autores de [VM03] nos dicen lo siguiente:

> Cuando se trata de escribir software fiable, hay cuatro tipos de programadores:
>
> - Aquellos que constantemente escriben código con errores.
> - Aquellos que escriben código razonable a partir de ejemplos y motivación.
> - Aquellos que escriben buen código la mayor parte del tiempo, pero no toman en cuenta sus limitaciones.
> - Aquellos que realmente entienden el lenguaje, la arquitectura de la computadora, ingeniería de software, el área de aplicación, y que pueden escribir código en libros de texto de manera regular.

En este trabajo se busca un nuevo acercamiento entre el software y el hardware de la computadora. El objetivo no es cerrar esa brecha, ya que tiene una muy fuerte razón de existir, sino mostrar que comprendiendo mejor la parte física de una computadora, se puede mejorar la manera en que la programamos y de igual manera sacar mejor provecho de ésta.

Para esto, se diseñó un conjunto de herramientas que funcionaran para cierta máquina relativamente sencilla y que cada decisión tomada en su diseño fuera en función de este acercamiento con el hardware.

Trabajos similares

El centro de investigación en artes, tecnología, educación y enseñanza (CRATEL en inglés) ofreció en 2008 un curso de ensamblador usando el Gameboy como su máquina simple. Ellos usaron un modelo construccionista para enseñar el curso en contraste con los modelos instruccionistas donde el conocimiento parte del profesor. Se puede ver el resultado de este curso así como algunas conclusiones y comentarios de sus instructores en `http://vimeo.com/1264117`.

Existen, sin embargo, otros proyectos similares al que se desarrolló en este trabajo, por ejemplo el proyecto Arduino. Citando el sitio oficial de arduino (`http://www.arduino.cc/es/`):

> Arduino es una plataforma de electrónica abierta para la creación de prototipos basada en software y hardware flexibles y fáciles de

usar. Se creó para artistas, diseñadores, aficionados y cualquiera interesado en crear entornos u objetos interactivos.

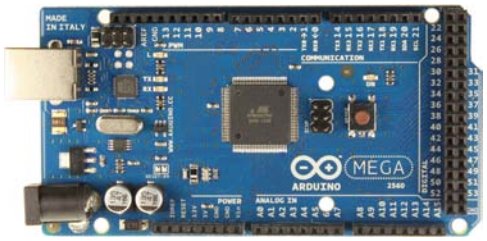


Figura 2: Foto de un Arduino MEGA

El objetivo del proyecto Arduino es bastante similar al de este trabajo ya que busca ese acercamiento entre el hardware y el software que en dicho proyecto ha tenido bastante exito.

Otro proyecto similar es Raspberry Pi. Raspberry Pi es una computadora simple pero completa en una sola placa que tiene como objetivo estimular la enseñanza de las ciencias de la computación en las escuelas.

Figura 3: Foto de una placa Raspberry Pi

Como se expuso, la hipótesis de este trabajo es la siguiente:

> **La creación de un conjunto de herramientas de desarrollo que se basen en una máquina simple ayudará a quien las use a comprender de mejor manera la arquitectura de computadoras y su programación.**

Se entiende como arquitectura de computadoras el diseño teórico o conceptual de las partes de una computadora, su organización estructural y su

implementación física. Tanto Patterson y Hennessy como Hamacher et al dan mejores definiciones del término en sus libros [PH11] y [HVZ02] respectivamente.

Para probar la hipótesis, se realizó una propuesta didáctica donde se desarrolló un conjunto de herramientas con el objetivo de explotar las capacidades de una máquina simple, que se explica más adelante, y entender cómo algunas decisiones en el diseño de esta máquina influyen en la manera en que debemos programarla.

Fue así como se estableció el objetivo de este trabajo que se cita a continuación:

> **Demostrar que la creación de un conjunto de herramientas de desarrollo con base en una máquina simple ayudará en la correcta comprensión de la arquitectura de computadoras y su programación, independientemente de los lenguajes existentes.**

En este trabajo se explican las decisiones tomadas asi como su justificación para desarrollar estas herramientas y la elección de la máquina simple. El trabajo consta de ocho capítulos diferentes que se describen a continuación:

1. **La máquina simple:** En este capítulo exploro diferentes posibilidades para la elección de la máquina simple. Una vez elegida describo sus características más importantes y relevantes.

2. **Las herramientas:** Ya que está bien definida la máquina que usaremos, entonces podemos discutir las diferentes posibles herramientas hasta llegar al conjunto definitivo de éstas.

3. **El ensamblador:** En este capítulo describo el diseño y algunos detalles de la implementación del ensamblador para Game Boy.

4. **El lenguaje genérico:** En este capítulo explico las ideas y las desiciones que tomé para el diseño de un nuevo lenguaje de alto nivel llamado lenguaje genérico estructurado.

5. **El compilador:** En este capítulo describo de manera resumida la metodología que usé para construir un compilador para el lenguaje genérico estructurado.

6. **El emulador:** En este capítulo resumo algunos puntos teóricos importantes sobre emulación y explico las técnicas utilizadas para el diseño y la implementación de un emulador de Game Boy.

7. **Otras herramientas:** En este capítulo describo un conjunto de herramientas menores que complementan a las de los capítulos anteriores. Es importante mencionar que aquí describo el desarrollo de un kit básico de programación para el Game Boy.

8. **Pruebas y resultados:** En este capítulo explico el diseño de un experimento para poner a prueba las herramientas. Al final de este capítulo muestro las conclusiones del experimento realizado.

Capítulo 1

La máquina simple

La idea de una máquina simple se refiere a una computadora que sea lo suficientemente completa para poder extrapolar lo que se expone en este trabajo a arquitecturas más complejas y que al mismo tiempo no sea tan difícil de analizar.

Una computadora completa debe tener todas las partes de una arquitectura de von Neumann (Figura 1.1), es decir, debe tener las siguientes partes como mínimo:

- Unidad Central de Procesamiento o CPU.
- Memoria principal.
- Sistema de entrada y salida.

Existe también la arquitectura Harvard, sin embargo, la gran mayoría de las computadoras modernas están basadas en la arquitectura de von Nuemann por lo que serña la que se utilizará en este trabajo.

Ahora veamos en que debe consistir cada una de estas partes en nuestra máquina simple.

1.1. CPU

La unidad central de procesamiento (CPU) es el corazón de nuestra computadora. Existen muchos fabricantes de procesadores de todo tipo, por ejemplo Atmel, Motorola, Intel. Sin embargo, los procesadores de Intel han adquirido un enorme terreno en el negocio de las computadoras personales. Una característica importante que influye mucho en el objetivo de este trabajo es la elección del tipo de CPU bajo la siguiente clasificación:

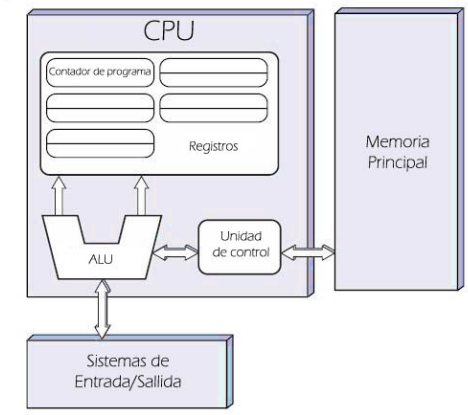


Figura 1.1: Arquitectura de von Neumann. Imagen de David Strigoi

- Procesadores RISC.
- Procesadores CISC.

En resumen, los procesadores tipo CISC (computadora con conjunto de instrucciones complejas) tienen, irónicamente, instrucciones mas simples y de mas bajo nivel que los procesadores RISC(computadora con conjunto reducido de instrucciones) donde las instrucciones realizan tareas mas complejas. Se puede consultar [HVZ02] para más información sobre las diferencias de estos tipos de procesadores.

1.2. Memoria principal

La memoria principal de una computadora no es más que el conjunto de memoria ROM y RAM de esta.

1.2.1. Memoria ROM

La memoria de sólo lectura (ROM) es un tipo de memoria donde una vez escrita la información, esta permenece indefinidamente.

Existen gran variedad de formas para almacenar nuestros programas. Por ejemplo, tenemos cartuchos con uno o más circuitos que incluyen chips de memoria ROM que contienen nuestro programa. Los famosos CD-ROMs también son memoria ROM que se usa bastante para almacenar programas y otros datos. Existen variaciones de la memoria ROM como la PROM,

EPROM, EEPROM, que básicamente son memorias ROM con la posibilidad de ser reprogramadas. Los discos duros y memorias externas no son parte de la memoria principal. Estos normalmente están dentro de la clasificación de memoria secundaria.

1.2.2. Memoria RAM

La memoria de acceso aleatorio (RAM), nos sirve para guardar datos generados por nuestros programas. Haciendo una pequeña analogía, podríamos decir que es nuestra mesa de trabajo. En algunas computadoras también se usa para almacenar temporalmente uno o más programas y así su acceso sea mucho más rápido para el CPU. Su característica importante es que al apagar la computadora, su contendio se pierde a diferencia de la memoria ROM.

1.3. Sistema de entrada y salida

Como el nombre indica, es la parte de la computadora que administra las entradas del usuario y del mundo real y a su vez canaliza las respuestas a través de una o más salidas. Ejemplos clásicos de entradas son el teclado, el mouse, botones, microfonos, etcétera. En cuanto a salidas existe una gran variedad, pero los más usados son un monitor o pantalla LCD, bocinas, entre otros.

1.4. Otras máquinas simples

Existen otros proyectos similares aunque con objetivos diferentes. Algunos de estos proyectos podrían ser utilizados para demostrar la hipótesis expuesta. A continuación se explica brevemente por qué no se usan estas alternativas. Los siguientes proyectos son los que al momento de escribir esto han estado en auge.

1.4.1. Arduino

La ventaja de Arduino es que es un proyecto tanto de hardware libre como de software libre. Esto quiere decir que toda la especificación del hardware está disponible al público. Incluso en su página oficial, existen guías de como construir tu propio Arduino. El software que se usa para programarlo también es libre y gratuito. En general, Arduino usa microprocesadores Atmel por su equilibrio entre potencia y precio.

Sin embargo, la arquitectura de Arduino ya no entra en la categoría de máquina simple. Esto se debe a que su microprocesador es demasiado complejo y por lo tanto, se usa C/C++ para programarlo. Entonces, se pierde el objetivo de este trabajo. Podríamos usar ensamblador, pero la curva de aprendizaje sería demasiado pronunciada. Por lo tanto, esta no es una buena alternativa.

Existen otros proyecto equivalentes a Arduino como las placas Parallax o los Basic Programmable Microcontrollers. Todos estos buscan ser pequeñas computadoras programables que puedan comunicarse facilmente con el mundo exterior sin necesidad de que el programador conozca los detalles del hardware.

1.4.2. Raspberry Pi

Raspberry Pi es una potente computadora del tamaño de una tarjeta de crédito. Su proposito es más general que Arduino y otros. El problema es el mismo que con Arduino y sus equivalentes. El objetivo no es acercarse al hardware, sino tener computadoras pequeñas capaces de hacer una gran variedad de cosas sin que el programador tenga que meterse en los detalles internos del hardware.

Lo que se busca, es una máquina donde se pueda escudriñar el hardware y aprender de las desiciones tomadas para su diseño. La solución surgió del estudio del hardware de las consolas portátiles de videojuegos.

Se encontró que la que tiene uno de los mejores diseños y una simpleza que puede hasta considerarse elegante es el conocido Game Boy y su sucesor directo Game Boy Color. En la siguiente sección se describen las características del Game Boy que dieron pauta a ser elegido como nuestra máquina simple.

1.5. El Game Boy Color

El Game Boy Color es una consola de videojuegos portatil desarrollada por Nintendo. Originalmente sólo existía el Game Boy el cual contaba con un LCD en blanco y negro y una bocina con sonido estereo. Nintendo desarrolló más tarde el Game Boy Color el cual, a parte de añadir algunas mejoras al hardware, se caracterizó por su nueva pantalla LCD a color. En este trabajo se usa de manera específica el Game Boy Color (Figura 1.2). Sus especificaciones técnicas son las siguientes:

- CPU personalizado similar al Z-80 de Zilog y al Intel 8080 de 8 bits.

Figura 1.2: Game Boy Color

- Velocidad de 4 Mhz con posibilidad de aumentarla a 8 Mhz.
- Memoria RAM interna de 32KB
- Memoria RAM de video de 16KB.
- Pantalla LCD con resolución de 160 x 144 pixeles.

Otras consolas de videojuegos como el Nintendo Entretainment System, Atari, entre otros no tienen un diseño tan elegante como el Game Boy.

La primer ventaja es su CPU. Es hasta cierto punto compatible con el procesador Intel 8080 y con los Zilog Z80. Aparte, el procesador que usa es bastante sencillo y sin embargo muy potente. Muchas de sus características y modo de programación también se aplican a los procesadores actuales.

Por otro lado cuenta poca memoria, sin embargo puede ser aumentada usando memoria externa. Esto es gracias a que el Game Boy usa cartuchos.

Con el uso de cartuchos podemos meter mas memoria dentro del circuito y así aumentar las capacidades.

Otra ventaja es que esta computadora ya viene con una pantalla LCD integrada al igual que con sonido estereo de 4 canales.

Al no tener suficiente memoria, y una velocidad relativamente baja, un compilador de C añadiría mucha sobrecarga a los programas, por lo que normalmente no se usan, a pesar de que si existen. Sin embargo, su conjunto de instrucciones es muy completo y no es dificil de aprender, por lo que se pueden hacer programas en ensamblador con relativa facilidad.

Se puede consultar [AGF+99], donde viene todas las especificaciones. Cabe señalar que se citará constantemente este manual para explicar características de las herramientas que aquí se desarrollaron.

Capítulo 2

Las herramientas

Uno de los puntos importantes de este trabajo es el conjunto de herramientas. A continuación se explica de dónde surge este conjunto así como algunas de sus características principales que las hacen únicas.

2.1. El software del sistema

En las computadoras modernas existe un conjunto de software llamado software del sistema.

> El software del sistema consiste en una variedad de programas que apoyan la operación de una computadora. Este software hace posible que el usuario se enfoque en una aplicación o en resolver algún problema sin necesidad de saber los detalles de cómo funciona la máquina internamente [Bec97].

Existe aun un poco de discrepancia sobre qué programas pertenecen al software del sistema. Lo que si se puede decir es que se trata de la principal capa de abstracción. La mayoría de las fuentes incluyen los siguientes programas:

- Sistema operativo.
- Ensamblador.
- Cargador.
- Enlazador.
- Compilador.

Leland Beck en su libro [Bec97] agrega algunos más como el procesador de macros o incluso el editor de textos. Este conjunto de software, al ser la capa principal entre el hardware y el programador, es precisamente lo que nos aleja del hardware. La idea es adelgazar esta capa y trabajar sobre ella para crear herramientas que sólo oculten los detalles del hardware suficientes sin que afecten la manera en que este se explota.

Para esto, algunos de los programas mencionados son escenciales. Sin un ensamblador sería muy difícil y tardado hacer un programa. Por otro lado, el compilador no es tan esencial ya que el lenguaje ensamblador que usa el Game Boy es bastante accesible y se considera suficiente. El cargador es totalmente inecesario ya que no existe un sistema operativo. En cuanto al enlazador, va a depender de la manera en que programemos.

2.2. La elección de las herramientas

La elección de las herramientas se basó en el conjunto de programas que conforman el software del sistema. Esto fue porque es precisamente la capa de abstracción la que interesa analizar y explotar para cumplir con el objetivo de este trabajo. La idea es quitar los programas que nos alejen del hardware, pero conservar los que nos ayuden a entenderlo. Bajo este criterio, las herramientas elegidas fueron las siguientes:

- Un ensamblador.
- Un compilador.

2.2.1. Un ensamblador

Como se mencionó, el ensamblador es una parte esencial para poder programar una computadora donde no hay un compilador. Para usar el lenguaje ensamblador es necesario conocer la arquitectura de la computadora. La ventaja principal, es que tenemos el control total de su funcionamiento y del tamaño de los programas. Aparte, con la práctica y la experiencia se pueden realizar programas verdaderamente eficientes que pueden llegar a superar las optimizaciones de un compilador, sobretodo cuando se trabaja con procesadores CISC.

En este trabajo se detalla la construcción de un ensamblador y más que nada la manera de usarlo para cumplir con el objetivo.

2.2.2. Un compilador

Hay varios puntos que hacen al compilador útil para este trabajo: que no sea un lenguaje de alto nivel, y que nos deje incrustar código ensamblador. Con estas características podremos tener un lenguaje estructurado pero con la mayoría de las ventajas del lenguaje ensamblador. Para esto, en este trabajo se detalla el diseño de un lenguaje de un nivel bajo y posteriormente el diseño del compilador de este nuevo lenguaje.

2.2.3. Sobre el enlazador

Un enlazador (o linker en inglés) es una herramienta esencial en las computadoras modernas. Esto se debe a la tendencia de construir programas a partir de diferentes módulos que se compilan por separado en archivos de código objeto. El enlazador junta estos objetos y los enlaza para crear un único programa ejecutable. Cuando existe un sistema operativo, el enlazador también añade código que tanto el cargador como el sistema operativo utilizarán para poder ejecutar el programa.

Ahora bien, no se incluye un enlazador como tal dentro del conjunto de herramientas que propongo. Esta decisión está fundamentada en la manera en que el Game Boy ejecuta sus programas. En resumen, utiliza un chip externo para intercambiar bancos de memoria ROM y RAM entre otras cosas. Cada banco tiene una capacidad de 16KB. Sin un chip externo, el Game Boy puede utilizar máximo 2 bancos, es decir, 32KB. Para que funcione el intercambio de bancos, éstos deben ser en gran medida independientes. Por lo que una manera de programarlos es como módulos separados.

Esta funcionalidad, que sería parte de un enlazador, la incluyo directamente en el ensamblador. De esta manera, ya no es necesario desarrollar una herramienta por separado que podría complicar la comprensión de las herramientas y cómo se relacionan con la arquitectura.

2.3. Herramientas complementarias

Existe otro grupo de herramientas fuera del software del sistema que se consideraron útiles y complementarias a las expuestas. La principal herramienta complementaria es sin duda un emulador.

2.3.1. Un emulador

La definición de emulador o emulación es muy general. "Emulación es la acción de imitar o incluso mejorar las acciones ajenas", según la Real

Academia de la Lengua Española. Pueden existir muchos tipos de emuladores. Por ejemplo hardware que imita otro hardware, software que imita las funciones de un sistema operativo, software que imita hardware, u otras combinaciones. Llevandolo al contexto de este trabajo. Un emulador es un programa que imita las funciones de un dispositivo en otro por medio de software.

Esto resultaría útil, ya que con un emulador en software del Game Boy se puede probar, o incluso depurar programas antes de introducirlos en un chip ROM y ejecutarlos en el hardware real.

En este trabajo se describirá la construcción de un emulador y su funcionamiento como herramienta. Construir un emulador, o por lo menos entender los algoritmos involucrados en su programación es un excelente ejercicio para aprender y dominar la arquitectura de una computadora. Por eso mismo, se hará incapié en las técnicas usadas para emular el Game Boy, de esta manera otros pueden basarse en ese material, para construir emuladores.

2.3.2. Utilidades

Finalmente se explicará la necesidad de otras herramientas menores. Estas herramientas se pueden clasificar de la siguiente manera:

- Herramientas de hardware
- Herramientas de software.

En este caso, no todas las herramientas fueron desarrolladas para este trabajo, pero se consideran bastante útiles para complementar las que son propias de este trabajo.

Una de las herramientas de hardware que se consideró de gran importancia es un kit básico de programación para el Game Boy. Este kit sirve para poder probar programas directamente en el hardware físico. Por el lado de las herramientas de software, se menciona el uso de un depurador, herramienta que ayudará a escribir programas correctos antes de ser implementados en el hardware.

Capítulo 3

El ensamblador

Como se mencionó anteriormente, una de las herramientas fundamentales del software del sistema es el ensamblador. Esta herramienta es la que va a permitir traducir código ensamblador en lenguaje máquina.

3.1. Lenguaje máquina

Básicamente, el lenguaje máquina es el nivel de programación más bajo que maneja una computadora. Este lenguaje está compuesto por ceros y unos, es decir, código binario. En realidad, estos ceros y unos representan diferentes niveles de voltage dentro de un circuito. Los niveles de voltage dependen del tipo de lógica digital que se maneje y del diseño del circuito lógico.

Una forma conveniente de representar el código binario de una computadora es mediante bits. Un bit es la unidad más pequeña de datos de una computadora. Un bit puede sólo representar uno de dos estado: cero o uno. Al juntar varios bits obtenemos otras unidades como los nibbles, los bytes, y otros. Sin embargo, la representación binaria es más difícil de manejar por personas, por lo que normalmente se usan otras notaciónes. Una de ellas es a través de números octales. Esto se debe a que un número octal puede ser representado en su totalidad por 3 bits. La otra notación, que es la que usaremos de aquí en adelante es la hexadecimal. A diferencia de los números octales, un número hexadecimal puede ser representado en su totalidad por 4 bits, es decir, un nibble. Sin embargo, la mayor ventaja es que dos números hexadecimales representan un byte. Esto quiere decir que existe una correspondencia entre la numeración hexadecimal y el número de bits que tiene un byte, es decir, 8 bits. Aunque un byte no siempre tuvo 8 bits, hoy en día

es una medida estandar y ampliamente usada en la computación.

Teniendo esto en cuenta, se puede decir que se tiene código máquina representado con números hexadecimales.

3.2. Conjunto de instrucciones

Una abstracción que se usa en el diseño de microprocesadores para organizar el código máquina es lo que se conoce como conjunto de instrucciones. Cada instrucción de este conjunto se representa con un código binario único que al recibirlo el microprocesador ejecuta una serie de acciones que al final producen un resultado específico. La cantidad de instrucciones, asi como su tamaño es variable dependiendo del microprocesador.

En el caso del Game Boy, las instrucciones son de 8 bits, sin embargo existe un pequeño conjunto de intrucciones que usan 16 bits.

3.3. Lenguaje ensamblador

El lenguaje ensamblador surgió a partir de que se decidió dar nombres mnemotécnicos (nombres fáciles de recordar) a las intrucciones en codigo máquina. Esto facilita la comprension y la memorización del conjunto de instrucciones. En realidad no existe un solo lenguaje ensamblador, ni siquiera un estandar que lo defina. Existen varias corrientes sobre el formato del código ensamblador aunque al final, este dependerá completamente de quien programe el ensamblador.

Una característica importante del lenguaje ensamblador es que tiene una correspondencia de uno a uno con las intrucciones de código máquina de una computadora, es decir, a cada instrucción en lenguaje ensamblador le corresponde una instrucción en código máquina.

3.4. Programa ensamblador

Un programa ensamblador es aquel que traduce las instrucciones dadas en algun código mnemotécnico a codigo máquina, es decir, código binario representado en hexadecimal. Sin embargo, y por lo general, los programas ensambladores incluyen un conjunto extra de pseudo-instrucciones que nos ayudan a controlar la manera en que se ensambla nuestro programa. Estas pseudo-instrucciones no agregan codigo máquina a nuestro programa, sólo modifican la manera en que se ensambla y ejecuta.

3.5. Ensamblador de Game Boy

El microprocesador de un Game Boy tiene aproximadamente 200 instrucciones diferentes. En [AGF+99] se menciona que el microprocesador del Game Boy es similar tanto a un Intel 8080 como a un Zilog Z80. También muestra un listado completo de las intrucciones, su significado y funcionamiento. Esto sirve tanto para la creación del programa ensamblador como del emulador que se verá más adelante.

En este trabajo no se detallan las intrucciones, pero sí el proceso de construcción del programa ensamblador. Para poder conocer con detalle cada una de las instrucciones se puede consultar [AGF+99].

La idea básica para construir un ensamblador es la traducción directa de las instrucciones. La forma más práctica de programarlo es usando tablas de traducción.

Para este trabajo se programaron 2 ensambladores, uno hecho en C y el otro en Python. Se expondrá la versión de Python ya que es mucho más legible, compacta y completa. Esto se debe al excelente manejo de listas y tuplas que tiene Python. Tanto el código como los programas compilados se pueden descargar de la página `http://biogb.sourceforge.net`

3.5.1. Tabla de traducción

Como se mencionó anteriormente, la construcción del ensamblador se basó en una tabla de traducción. En el listado 3.1 se muestra un fragmento de la tabla de traducción que se usa:

```
opcode_table = [
    . . . .
    ["add", "a", 0x87],
    ["add", "b", 0x80],
    ["add", "c", 0x81],
    ["add", "d", 0x82],
    ["add", "e", 0x83],
    ["add", "h", 0x84],
    . . .
    ]
```

Listing 3.1: Fragmento de la tabla de traducción.

En la primer columna viene el nombre de la instrucción (opcode ó código de operación). En la segunda columna vienen los parámetros que lleva esa instrucción, en este caso a, b, c, d, e y h son los nombres de los registros del

microprocesador. Para saber más sobre los registros de un microprocesador, se puede consultar a [Abe96].

En el caso de intrucciones con parámetros directos como las del listado 3.1, no hay ningún problema para hacer la traducción. Sin embargo, tambien existen otras instrucciones con parametros variables, por ejemplo:

```
add, 0x0F
add, 0x10
call routine
ld a,0x5
```

Listing 3.2: Parametros varibles.

Como vemos en el listado 3.2, el parametro de estas instrucciones puede ser tanto un número como un identificador. Para resolver este problema se usaron las siguientes expresiones.

- $*$ Un byte.
- $**$ Dos bytes.
- $-$ Sin parametro.
- n Un número entre 0 y 7.
- *dir* Un identificador para una dirección.

Con estas sencillas expresiones, se pueden abarcar todos los parametros del conjunto de instrucciones. Sólo existe aun un conflicto. La diferencia entre n y $*$. Sin embargo, no existen instrucciones que usen ambos parametros, por lo que en realidad nunca surge el conflicto. Aun así hay que tener en cuenta que n sólo puede ser un número entre 0 y 7, por lo que se debe añadir una comprobación.

3.5.2. Resolución de saltos y rutinas

Otra de las características del ensamblador es el uso de nombres en vez de direcciones físicas. Esta característica en realidad es una manera de abstraer las direcciones de memoria. Esto da una mayor legibilidad al código ensamblador sin afectar las decisiones en cuanto a la ubicación del código.

Para lograr esto, se tiene que usar un algoritmo de 2 pasadas. En la primera pasada se traducen todas las instrucciones directas usando la tabla de traducción. Se deben crear dos tablas:

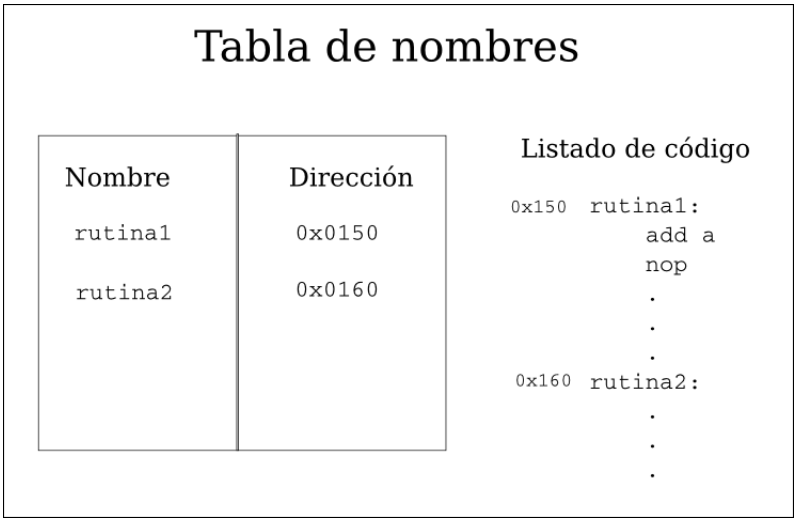


Figura 3.1: Tabla de nombres

La tabla de nombres (Figura 3.1) donde se van añadiendo todos los nombres de las rutinas en el orden en que se van encontrando y la dirección que les corresponde dentro del código.

La tabla de llamadas donde se van añadiendo las direcciones físicas de los lugares donde se hacen las llamadas a las rutinas o los saltos.

En la segunda pasada ya se tienen las 2 tablas completas, por lo que sólo se van escribiendo las direcciones físicas que hay en la tabla de nombres en los lugares que se hicieron las llamadas o los saltos, y esta información la encontramos en la tabla de llamadas.

Con esto se tiene la traducción de todas las instrucciones en ensamblador, sin embargo, los programas del Game Boy tienen una cabecera que contiene información sobre la forma en que se debe ejecutar el programa. Por lo tanto, debe existir un mecanismo donde se pueda indicar los valores que debe llevar esta cabecera.

3.5.3. Opciones del ensamblador

El mecanismo que se usó para poder establecer los valores de la cabecera de un programa fue con pseudo-instrucciones. Estas pseudo-instrucciones no generan código ensamblador, en su lugar, le indican al programa ensamblador que valores debe llevar la cabecera. En el listado 3.3 podemos ver un ejemplo de estas pseudo-instrucciones.

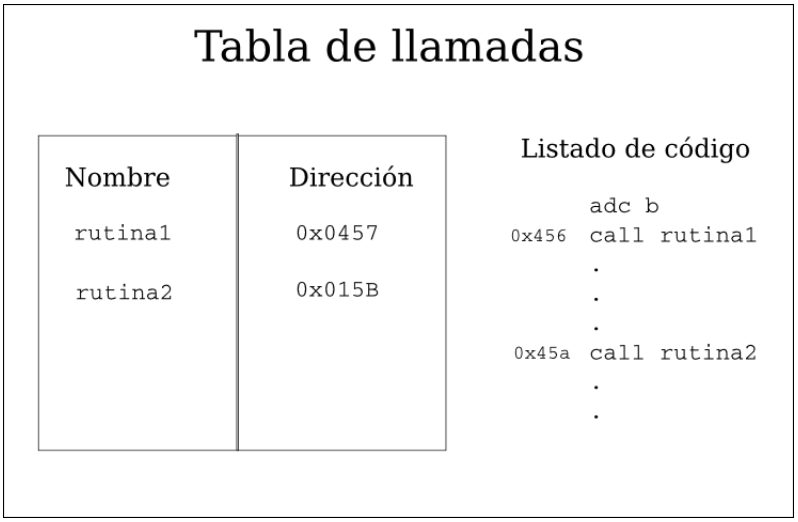


Figura 3.2: Tabla de llamadas

```
.title Nombre_Programa
.color
.main
```

Listing 3.3: Ejemplos de pseudo-instrucciones.

En el listado 3.3, se tienen 3 ejemplos de pseudo-instrucciones. Como se puede observar, todas comienza con un punto. De esta manera es más sencillo para el programa ensamblador reconocerlas y procesarlas. En el caso de *.title*, sirve para establecer el nombre del programa. *.color* le va indicar al programa ensamblador que escriba en la cabecera que nuestro programa debe ejecutarse en un Game Boy Color y por lo tanto están disponibles las capacidades extras. *.main* va a indicar donde comienza nuestro programa.

Para ver todas las pseudo-instrucciones que soporta el ensamblador, se puede consultar la documentación que viene junto al programa ensamblador.

El mecanismo de pseudo-instrucciones se puede aprovechar aun más. En este caso, se agreguó una pseudo-instrucción que nos ayuda a definir valores del tamaño de un byte dentro de nuestro código. La instrucción se llama *.db byte*. Esta sólo escribe el byte en el lugar donde se encuentra la instrucción. De esta manera se pueden agregar una gran cantidad de instrucciones que nos ayuden en las tareas de programación. Algunos ejemplo de pseudo-instrucciones que no se han implementado son:

- Bloques de datos con nombres.

- Macros.
- Constantes.
- Segmentos.
- Etcetera.

3.5.4. Bancos de memoria

Una característica importante del Game Boy y sus programas es el uso de bancos de memoria. Esto se debe a la escasa memoria que puede direccionar el Game Boy a partir de su BUS de direcciones. Los diseñadores de hardware de varias compañías idearon una manera de poder tener acceso a más ROM en un Game Boy. Este mecanismo se describe ampliamente en [AGF+99]. Sin embargo, para el programa ensamblador nos interesa saber como integrar estos bancos de memoria a nuestro programa.

Para esto, nuevamente se hace uso del mecanismo de pseudo-instrucciones descrito anteriormente. El funcionamiento es el siguiente: Cada banco de memoria debe estar en un archivo de código fuente diferente. Cada uno de estos archivos debe tener la pseudo-instrucción *.rombank n* donde *n* es el número de banco que se quiere usar para ese archivo. En caso de no ponerlo, se asumirá que se trata del banco 0. En caso de tener bancos repetidos, se mostrará un mensaje de error en el programa ensmablador, y se omitirá ese banco en el programa. Es importante notar que el programa ensamblador tiene la habilidad de indicar cuando un banco se satura. De esta manera se podrá saber cuando se debe agregar otro banco al programa.

Los algoritmos y técnicas descritas anteriormente fueron ideadas sin ninguna referencia inicial, sin embargo, en [Bec97] se pueden encontrar ciertas similitudes con los algoritmos aquí desarrollados.

Capítulo 4

El lenguaje genérico

Una de las principales ventajas de programar en ensamblador es el completo control sobre la máquina y la cercanía que hay con el hardware. Una vez que se haya adquirido habilidad, se pueden crear programas bastante eficientes y de menor tamaño. La desventaja principal es su bajo nivel de abastracción, es decir, al crear programas de mayor tamaño es fácil perderse entre todo el código, sobretodo si no está correctamente documentado. Normalmente los programas ensambladores agregan diferentes mecanismos para poder mejorar este aspecto, por ejemplo, el uso de nombres de rutinas, macros, entre otros.

La gran mayoría de los programas para Game Boy han sido escritos en ensamblador. Sin embargo, para este trabajo se considera importante la necesidad de tener un lenguaje de mayor nivel que el lenguaje ensamblador ya que la finalidad de éste es precisamente el acercamiento al hardware y ver la relación que tienen los lenguajes de alto nivel con respecto al código ensamblador y el funcionamiento de la máquina.

4.1. Características del nuevo lenguaje

Una vez dicha la razón por la cual se ha decidido diseñar un lenguaje de alto nivel, hay que mencionar las características principales que rigieron a la hora del diseño.

- Debe ser un lenguaje estructurado.
- Debe ser suficientemente minimalista para no sobrecargar las capacidades del Game Boy
- Debe poder aceptar código ensamblador en linea.

4.1.1. Estructurado

La idea de la programación estructurada fue originada entre 1968 y 1969, y fue Edsger Dijkstra uno de los primeros en introducir el nuevo término de *programación estructurada* [McC04].

En esencia la programación estructurada tiene 3 componentes [McC04] que se describen a continuación:

Secuencua: Es un conjunto de sentencias que se ejecutan en orden.

Selección: Son un conjunto de estructuras de control que ejecutan un grupo de sentencias de manera selectiva.

Iteración: Son un conjunto de estructuras de control que ejecutan un grupo de sentencias múltiples veces.

Dadas estas definiciones, se sabe entonces qué elementos debe llevar nuestro lenguaje para que sea estructurado. Estos elementos se describirán más adelante con mayor detalle.

4.1.2. Minimalista

Con minimalista se refiere a que el lenguaje debe tener los mínimos elementos necesarios para ser estructurado y nada más. De esta manera se pueden omitir algunas de las estructuras de control típicas de los lenguajes estructurados que pueden ser simuladas con otras.

Las estructuras de control deben ser suficientes para poder hacer un programa de manera estructurada y completo.

4.1.3. Ensamblador en linea

Una de las características necesaria para que este nuevo lenguaje sea verdaderamente útil es la inclusión de código ensamblador dentro de las estructuras de control que maneja. De esta manera se pueden abstraer algunas partes como saltos condicionales o las estructuras iterativas y concentrar el código ensamblador en lo importante. Steve McConnell nos menciona en [McC04] cómo la combinación de un lenguaje de alto nivel con otro de bajo nivel puede resultar en grandes optimizaciones si se hace adecuadamente.Por esta razón se consideró de gran importancia no dejar fuera el código ensamblador aun cuando el lenguaje sea de alto nivel.

4.2. Diseño del lenguaje

El primer paso para diseñar un lenguaje de programación es ponerle nombre. En este caso, dado que se trata de un lenguaje genérico estructurado, se ha decidido llamarlo LGE. De ahora en adelante se hará referencia al lenguaje como LGE.

A continuación se expone el análisis hecho para diseñar el LGE y sus palabras reservadas.

4.2.1. Sentencias

El componente principal de un lenguaje estructurado es la sentencia. Las sentencias son los elementos básicos en el código de un programa y la característica principal es que siempre realizan una acción dentro de este. Las sentencias pueden ser sencillas que al agruparse forman una ***secuencia***. También pueden ser estructuradas, que dependiendo de su estructura pueden ser tanto de ***selección*** como de ***iteración.***

Sin embargo, primero hay que definir la sentencia simple. Para LGE las sentencias deben ser sencillas y que debe haber el menor número de simbolos y palabras reservadas que aprender. Por lo tanto una forma sencilla es definir una sentencia en LGE como una linea de código que ejecuta una acción. La linea debe terminar con un salto de linea. En otras palabras, cada sentencia debe estar escrita en una linea diferente a diferencia de lenguajes como C donde con el simple hecho de escribir un símbolo de punto y coma (;) da por entendido que ahí termina una sentencia, y por lo tanto pueden existir varias sentencias en una sóla línea. En el siguiente capítulo se verá que el compilador permite varias sentencias en una sola línea para ciertos casos.

Para LGE sólo se definirán los siguientes tipos de sentencias básicas:

- Sentencias de asignación.
- Sentencias de llamadas.
- Sentencias de incremento y decremento.

Como se verá más adelante, las sentencias de asignación van a asignar valores a una variable. Las sentencias de llamadas van a a llamar a un procedimiento o rutina y finalmente las sentencias de incremento y decremento harán uso de los operadores con los mismos nombres respectivamente $(++, --)$. Dentro de estos dos tipos de sentencias básicas pueden existir expresiones.

4.2.2. Expresiones

Las expresiones se forman al unir los elementos atómicos de nuestro lenguaje: datos y operadores [Sch01]. Por lo tanto, primero se necesita establecer cuales van a ser nuestros datos.

El Game Boy tiene un BUS de datos de 8 bits, aunque algunas de sus instrucciones pueden trabajar con datos de 16 bits. Basandose en estas características, se introdujeron en LGE 2 tipos de datos únicamente: *byte* y *word* donde su tamaño es de 8 y 16 bits respectivamente. Para poder usar valores más grandes se tendrá que hacer uso de algunos operadores como se mostrará más adelante.

Se tomó la sencillez de la definición de variables que usan los lenguajes modernos como C o Java tomando en cuenta que son de tipado fuerte, es decir, todas las variables deben tener un tipo establecido el cual siempre se mantiene estático. La declaración de variables en LGE siguen la convención donde primero se escribe el tipo seguido de un número indeterminado de espacios y luego el nombre de la variable como se muestra en el listado 4.1:

```
byte miNumero
word otraVariable
```

Listing 4.1: Ejemplos de declaración de variables.

El otro elemento para poder definir expresiones son los operadores. Al igual que en prácticamente todos los lenguajes de programación, existen varios tipos de operadores. A continuación se enlistan los operadores soportados:

- Operadores aritméticos: suma, resta, multiplicación, módulo y división entera.
- Operadores de incremento ($++$) y decremento ($--$).
- Operadores relacionales.
- Operadores lógicos.
- Operadores de bits.
- Operador de indexación

Operadores aritméticos

Basicamente LGE soporta todos los operadores aritméticos comunes:

- Suma +
- Resta −
- Multiplicación ∗
- División entera /
- Módulo %
- Parentesis ()

4.2.3. Operadores relacionales

LGE también soporta todos los operadores realcionales que se usan comunmente en los lenguajes de programación modernos.

- Mayor que >
- Mayor o igual que >=
- Menor que <
- Menor o igual que <=
- Igual que ==
- Distinto que ! =

Operadores lógicos

LGE también tiene soporte de todos los operadores lógicos comunes.

- Y *and*
- OR *or*
- NOT *not*

Operadores de bits

Una característica escencial en un lenguaje de bajo nivel es la manipulación de bits por lo que LGE tiene el mismo soporte que el lenguaje C para la manipulación de bits.

- Y &
- O |
- XOR ^
- Complemento
- Desplazamiento a la derecha >>
- Desplazamiento a la izquierda <<

Operador de indexación

En LGE es posible la creación de listas o arreglos de bytes, por lo que se necesita un operador para poder trabajar con los elementos individuales de estas estructuras.

- Operador de indexación []

Existen muchos otros operadores que no son soportados por esta primera versión de LGE, entre ellos están todos los involucrados en punteros ya que LGE no soporta punteros. Tampoco soporta operadores de conversión de tipos por lo que habrá que tener cuidado al intentar asignar una variable de 16 bits a una de 8 bits.

Regresando a la formación de expresiones, ya tenemos los dos elementos atómicos para formarlas. Una característica principal de las expresiones es que estas siempre devuelven un valor.

De la misma manera, las sentencias pueden estar compuestas por diferentes expresiones como podemos ver en el listado 4.2:

```
byte miNumero = 5+6
word otraVariable = (miNumero >> 2)
Procedimiento(otraVariable & 7)
```

Listing 4.2: Ejemplos de declaración de variables.

En el primer ejemplo se tiene una sentencia donde le asignamos el valor que regresa la expresión $5 + 6$ a la variable *miNumero*. En el segundo ejemplo se tiene otra sentencia de asignación en donde ahora se utiliza el valor de la variable *miNumero* como parte de la expresión que regresa un valor y lo asigna en *otraVariable*. Finalmente en el tercer ejemplo se observa la llamada a un procedimiento o función donde como parámetro se pasa una expresión que al ser evaluada dará como resultado un valor que será el que recoja la función.

Con estos bloques fundamentales ya se tiene gran parte de la definición de nuestro lenguaje, sin embargo, aun faltan 2 tipos de estructuras.

4.2.4. Sentencias de selección

Las sentencias de selección son las que van a permitir elegir entre ejecutar un conjunto de sentencias u otro, pero no los dos. La sentencia más conocida y usada, y la única que soporta LGE es la sentencia *if..else* Al igual que las sentencias simples, su delimitación es el salto de linea, por lo que no hay necesidad de usar paréntesis como se da en el lenguaje C o Java. En el listado 4.3 se muestra un ejemplo de la forma de la sentencia if...else.

```
if expresion
{
    sentencia1
    sentencia2
}
else
{
    sentencia3
    sentencia4
}
```

Listing 4.3: Ejemplos de declaración de variables.

Con este ejemplo se pueden definir algunos detalles. En LGE no existen los tipos booleanos, por lo que por convención se eligió el valor de cero como verdadero y cualquier otro número como falso. Esto es necesario para que las expresiones condicionales puedan funcionar. Otro detalle es el uso de llaves para definir bloques de sentencias. A diferencia de C, C++ o Java, las llaves sólo pueden ser usadas en sentencias de selección y de iteración, nunca fuera de estas. Finalmente, se muestra la carencia de paréntesis en la expresión de la sentencia *if*.

Cabe mencionar que LGE si soporta la anidación de estructuras de selección, pero no soporta más de una expresión condicional que en otros lenguajes se da con la instrucción *else if* o *elif*. Como se dijo en la sección de operadores, los paréntesis también son operadores, por lo que la expresión puede ir encerrada entre parentesis sin alterar la semántica del programa.

4.2.5. Sentencias de iteración

Las sentencias de iteración son las que permiten ejecutar un grupo o bloque de sentencias de manera iterada, es decir, múltiples veces. LGE soporta sólo una sentencia de iteración: *while* En el listado 4.4 se muestra un ejemplo de la estructura *while*

```
while expresion
{
    sentencia1
    sentencia2
}
```

Listing 4.4: Ejemplos de declaración de variables.

Al igual que la sentencia de selección de LGE, la sentencia de iteración soporta anidación. El uso de llaves sigue las mismas reglas al igual que el uso de paréntesis en la expresión.

4.2.6. Definición de rutinas y funciones

Otra de las características de un lenguaje estructurado es la posibilidad de ***refactorizar*** código que se repite constantemente en una rutina que se pueda invocar. En otras palabras, si se tiene un conjunto de sentencias que se repiten a menudo dentro de un código, se pueden remplazar por la llamada a una rutina que ejecute esas instrucciones. Esto vuelve el código mucho más legible y fácil de mantener. A diferencia de una función, una rutina no devuelve valores.

LGE soporta tanto rutinas como funciones. Hay que tener especial cuidado, ya que los valores de retorno dependerán del tamaño que resta dentro de la pila de memoria en el Game Boy. En el listado 4.5 se puede ver un ejemplo de una función.

```
byte miFuncion(parametros)
{
    sentencia1
```

```
    sentencia2
    return resultado
}

routine miRutina(parametros)
{
    sentencia3
    sentencia4
}
```

Listing 4.5: Ejemplos de declaración de variables.

Como no existe un tipo de datos *void*, se usa la sentencia *routine* para poder definir una rutina. Es importante notar que si una función no va a regresar un valor, se use mejor una rutina. Esto se debe a que las rutinas producen mucho menos código ensamblador que las funciones y por esto mismo sobrecargan menos el Game Boy.

4.2.7. Estructura general de un programa

Finalmente para unir todas las estructuras ya analizadas, se necesita definir la estructura general de un programa. Aquí es donde radica la diferencia del lenguaje LGE. A continuación se enlistan las partes que debe tener en orden un programa correctamente escrito.

1. Definición de cabecera.
2. Definición de funciones, rutinas y variables globales.
3. Instrucción *.main*
4. Sentencias de nuestro programa.
5. Instrucción de fin de programa .end

Dentro de la definición de la cabecera se establecen en general los mismos datos que se pueden establacer en el ensamblador. Para información detallada sobre las opciones de cabecera se puede consultar el archivo de texto que viene junto al compilador que peude ser descargado en `http://biogb.sourceforge.net`. Es importante mencionar que en esta sección del programa se indican los nombres de los archivos que serán los diferentes bancos de memoria.

La definición de rutinas y funciones en LGE debe ser siempre antes del programa principal, es decir, antes de la instrucción *.main*. En esta sección también se puede hacer la declaración y definición de variables globales. Otro punto importante que vale la pena mencionar aquí es que dentro de las rutinas y funciones así como despúes de la instrucción *.main* la definición de variables va al principio del cuerpo del bloque. Una vez que se escribe una sentencia que no sea de definición o declaración de variables ya no se podrá declarar ni definir ninguna variable.

La instrucción *.main* indica el comienzo de nuestro programa. Es importante señalar que dentro de todos los bancos, sólo el banco cero debe tener una instrucción *.main*. Los demás bancos no deben tenerla.

En la sección de sentencias viene el cuerpo del programa. Siempre hay que recordar que cada banco tiene un tamaño limitado, por lo que hay que cuidar la cantidad y complejidad del código.

Finalmente hay que incluir una instrucción *.end* para indicar que el programa ha terminado. De esta manera tanto el compilador y el ensamblador pueden realizar una serie de optimizaciones que pueden resultar en código más compacto.

Ya se tienen los elementos de nuestro nuevo lenguaje, pero necesitamos una forma formal de establacerlos.

4.3. Gramática de LGE

Para poder expresar el nuevo lenguaje de una manera formal se necesita algún tipo de notación ya establecida. Una de las notaciones más usadas para definir lenguajes formales es la Backus Naur Form (BNF). Esta notación es la que se usará para expresar formalmente la gramática del lenguaje LGE. En el siguiente capítulo, en la sección de Análisis sintáctico se describe el concepto de gramática independiente de contexto así como sus propiedades importantes. En esta sección sólo se expondrán algunas partes del lenguaje LGE en notación BNF. En la sección de anexos se puede encontrar la definición completa de la gramática usada en este trabajo.

Una de las características de las gramáticas libres de contexto es la posibilidad de construir reglas a partir de otras reglas. Es importante mencionar que todas las reglas deben tener una relación con las demás y que al final exista un camino hasta la regla inicial, la raiz.

En la gramática LGE la regla inicial es la siguiente:

```
<goal> ::= <programa>
```

El ejemplo anterior no da ningún tipo de información excepto que al final se debe tener un programa. Las siguientes reglas dan mucha más información sobre la estructura de un programa:

```
<programa> ::= <cabecera> <def-globales> ".main"
               <cuerpo-programa> ".end"
<programa> ::= <cabecera> ".main" <cuerpo-programa> ".end"
```

En este caso se tienen dos reglas con el mismo nombre. En la primer regla dicta que un programa consta de una cabecera seguida por una sección de definiciones globales, luego la palabra reservada **.main**, después el cuerpo del programa y finalmente la palabra reservada **.end**.

Cada una de las reglas que están al lado derecho (las expresiones entre signos $<$ y $>$) van definiendo las diferentes partes de un programa. En algunos casos se hace uso de recursividad como en la declaracion de funciones:

```
<def-funciones> ::= <funcion> <def-funciones>
<def-funciones> ::= <rutina> <def-funciones>
<def-funciones> ::= <funcion>
<def-funciones> ::= <rutina>
```

En este último ejemplo se define la sección de definición de funciones como una función seguida de más definiciones de funciones. Esta es una de las ventajas del uso de las gramáticas independientes de contexto. También es importante mencionar cómo la notación BNF ayuda a que nuestra gramática sea legible. Existen varias consideraciones que se deben tomar en cuenta a la hora de construir una gramática independiente de contexto como por ejemplo la recursividad puede ser por la derecha o por la izquierda. Estos detalles afectan tanto en el rendimiento del compilador así como en la interpretación de algunas expresiones. Hopcroft et al. dan un muy buen tratamiento a estas cuestiones en [HMU07].

Capítulo 5

El compilador

El compilador es una parte del software del sistema. Un compilador da la oportunidad de usar lenguajes de alto nivel para escribir programas en nuestra máquina. Citando a Cooper y a Torczon se tiene esta excelente definición de compilador.

> Los compiladores son programas de computadora que traducen un programa escrito en un lenguaje de programación en otro programa escrito en otro lenguaje [CT12].

En otras palabras, un compilador es un traductor que a diferencia de un ensamblador no necesariamente traduce el código fuente a código máquina sino a cualquier otro lenguaje.

La ventaja principal de tener un compilador para Game Boy es poder tener disponibles las estructuras de un lenguaje estructurado definidas en el capitulo anterior junto con las características de un lenguaje de bajo nivel. En este capítulo se describen las decisiones que tomadas a la hora de diseñar el compilador así como las técnicas usadas.

5.1. Características del compilador

La mayoría de las características de un compilador son definidas a través de la definición del lenguaje fuente. En el capítulo anterior se definió el LGE como el lenguaje fuente para este trabajo. De esta definición se obtuvo una gramática libre de contexto tipo $LR(1)$ la cual fue la que se utilizó para construir el compilador. Aparte de estas características también se exponen las siguientes:

- El compilador traducirá el lenguaje LGE a lenguaje ensamblador del procesador del Game Boy.
- Cada archivo de código en LGE producirá un banco de ROM diferente para el ensamblador.

La razón por la cual se tomó la decisión de que el compilador produjera código ensamblador en vez de código máquina directamente es la siguiente: Las herramientas descritas en este trabajo tienen el objetivo de ayudar a entender los conceptos de arquitectura de computadoras; al producir código ensamblador desde el compilador podemos analizar qué es lo que realmente está sucediendo en el programa y de esta manera comprender los mecanismos internos de la computadora al ejecutar el programa. La idea es que haya un acceso fácil al código ensamblador producido por el compilador.

Otro punto importante es que la resolución de nombres de rutinas es delegada al enamblador en vez de ser resuelta por el compilador. Esto es por la misma razón expuesta en el párrafo anterior. El compilador producirá código ensamblador utilizando los nombres de rutinas originales y esto facilitará el análisis del código producido.

Por otro lado, al limitar cada archivo a un banco de ROM se puede obtener mejor información de espacio para cada banco. Sin embargo, los algoritmos para cambiar bancos dependerán totalmente del programador. Esto ayudará a no sobrepasar los límites de memoria que tiene cada banco de memoria ROM.

5.2. Partes del compilador

En esta sección se describe de manera resumida los algoritmos y técnicas que usados en cada una de las partes del compilador. Esta sección no es una descripción exhaustiva sobre el tema de compiladores. Para estudiar el tema en profundidad existe una gran cantidad de textos de los cuales se recomiendan [ASU90], [Hol90], [CT12], entre otros.

5.2.1. Análisis léxico

El primer paso para desarrollar un compilador es programar un analizador léxico. La función de un analizador léxico es leer caracter por caracter el código fuente y producir los denominados ***tokens***. En escencia un token es una unidad léxica o lexema descrita por una expresión regular. En términos más sencillos, es una palabra de nuestro lenguaje fuente.

Una de las herramientas teóricas que más se usa para el análisis léxico son las expresiones regulares. Citando a Grune et al.:

> Una **expresión regular** es una fórmula que describe un posible conjunto infinito de cadenas. [GBJL07].

Las expreisones regulares hacen uso de patrones para poder definir grupos de cadenas, por ejemplo, en el listado 5.1 se muestra la expresión regular para el conjunto de todas las cadenas que forman un nombre de variable válido en el lenguaje C y el conjunto de todos los números enteros.

```
N -> 0, 1, 2 ... 9
L -> a, b, c ... z, A, B, C ... Z

variable = (_|L)(L|N)*

enteros = (-NN*)|(NN*)
```

Listing 5.1: Expresiones regulares

En primer instancia se definen dos símbolos, N y L que representan los números naturales incluyendo el cero y las letras del abecedario respectivamente. En el primer ejemplo, se establece que la cadena debe comenzar con una letra o con un guión bajo. Después puede ir cualquier cantidad de símbolos tanto númericos como alfabéticos. En el segundo ejemplo, se establece que un número entero puede ser tanto positivo como negativo, y consta de un símbolo numérico seguido por cero o más números naturales.

Las expresiones regulares son la parte teórica para realizar el análisis léxico. Para poder ponerlas en práctica normalmente se implementan usando ***autómatas finitos*** o más específicamente ***máquinas de estados finitos.***

Citando a Hopcroft et al. un autómata finito es:

> Es aquel que puede estar en un único estado después de leer cualquier secuencia de entradas [HMU07].

En realidad, es un formalismo matemático que permite reconocer lenguajes regulares a partir de expresiones regulares. Estos se pueden representar mediante diagramas como se muestran en la figura 5.1

En esta figura se muestra el diagrama de las dos expresiones regulares mostradas anteriormente. Los símbolos $q0$ y $q1$ son nombres de estados de transición. $Variable$ y $Numero$ también son estados que se caracterizan por ser finales aceptables, es decir, si el autómata se detiene en uno de ellos significa que ha reconocido un *token* válido.

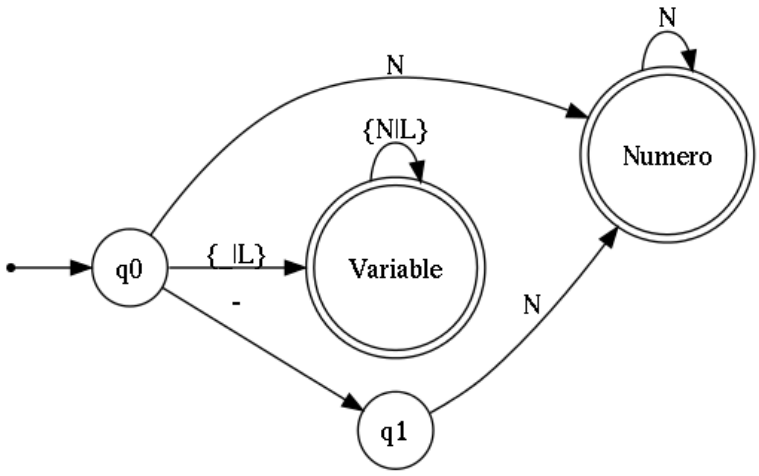


Figura 5.1: Diagrama de una máquina de estados finitos.

Ya teniendo este modelo, se puede diseñar el algoritmo que implemente la máquina de estados. En el listado 5.2 se muestra un posible algoritmo.

```
while state != final state
    character = read next character from source
    token.append(character)
    state = statesMatrix[state, character]
```

Listing 5.2: Algoritmo de un analizador léxico.

Este algoritmo es la base para programar el analizador léxico. Evidentemente requiere pasos adicionales como la creación de la matriz de estados, control de errores, entre otras cosas. La matriz de estados se puede generar a partir de una ánlisis de las expresiones regulares. Existe una variedad de algorítmos que pueden generar máquinas de estados finitos (las matrices) a partir de una serie de expresiones regulares. Para el compilador de este trabajo se generaron los valores de la matriz de manera manual. Se puede revisar el código fuente del compilador para poder observar este hecho.

El algoritmo expuesto es el más general y sencillo de implementar, pero existen otros algoritmos con algunas ventajas y desventajas al compararlo con el expuesto. En [CT12] se pueden encontrar algunos de estos algoritmos así como sus propiedades que los hacen únicos.

5.2.2. Análisis sintáctico

Algunos textos como [GBJL07] consideran el analizador léxico y sintáctico como una sola parte del compilador. Sin embargo, en este trabajo se toman como partes separadas para analizarlos de manera independiente.

Recordando, el analizador léxico lee el código fuente como entrada y produce *tokens* como salida. La función del analizador sintáctico es leer estos *tokens* y crear una representación del programa. Normalmente esta representación es en forma de árbol sintáctico. Sin embargo, existen compiladores que no requieren crear un árbol sintáctico. Para este trabajo, se decidió implementarlo ya que este facilitará la explicación y comprensión de la siguiente parte del compilador: el análisis semántico y la generación de código.

La herramienta teórica principal que ayudará a construir un analizador sintáctico es una extensión del concepto de expresiones regulares y se conocen como ***gramáticas independientes de contexto***. Se les llaman independientes de contexto porque las estructuras sintácticas tendrán siempre el mismo significado sin importar el contexto que las rodee en un programa. Tanto las expresiones regulares como las gramáticas en general son ampliamente estudiadas como parte del tema de lenguajes formales en el área de matemáticas discretas. En [HMU07] se puede encontrar un excelente estudio de estos dos temas.

Citando a Hopcroft et al. tenemos la siguiente definición.

> Una gramática independiente de contexto es una notación formal que sirve para expresar las definiciones recursivas de los lenguajes [HMU07].

Una de las características que diferencían a las gramáticas independientes de contexto de las expresiones regulares es la posibilidad de incluir definiciones recursivas, es decir, poder definir reglas a partir de otras reglas de la misma gramática.

Un ejemplo clásico de una gramática libre de contexto es la que se muestra en el listado 5.3.

```
Goal -> Expresion
Expresion -> Factor + Expresion
Expresion -> Factor - Expresion
Factor -> Term * Factor
Factor -> Term / Factor
Term -> (Expresion)
```

```
Term -> N
```

Listing 5.3: Gramática independiente de contexto.

Con este ejemplo se pueden definir algunos conceptos:

Producciones: También llamadas reglas son las definiciones individuales de una gramática donde $A- > B$ se lee "A produce B". En el ejemplo anterior cada linea es una producción.

No terminales: Son los símbolos gramaticales que derivan otros símbolos gramaticales. En el ejemplo anterior y en todas las gramáticas independientes de contexto todos los símbolos a la izquierda del símbolo de producción son no terminales.

Terminales: Son aquellos símbolos gramaticales que no derivan en otros símbolos gramaticales. En el ejemplo anterior, $+, -, *, /, (,)$ y N son terminales.

En el capítulo anterior se definió el lenguaje LGE. Se puede encontrar la gramática completa de este lenguaje en la sección de anexos.

Existen varios algoritmos y métodos para realizar el análisis sintactico, pero se pueden clasificar en dos categorías: ascendentes y descententes. Los algorítmos de análisis ascendente construyen el árbol sintáctico a partir de sus hojas creando nodos intermedios hasta llegar a la producción inicial de la gramática como nodo raiz. Los métodos descendentes normalmente parten del símbolo inicial y van construyendo una derivación hasta llegar al símbolo encontrado.

No todos los métodos siempre funcionan ya que dependen mucho de la naturaleza de la gramática. Para la gramática de LGE se utilizó el método ascendente $LR(1)$. Esto se debió a que los métodos ascendentes son mucho más eficientes y compactos, aparte, la gramática LGE es del tipo LR(1), esto quiere decir que se debe de leer de izquierda a derecha (L) donde siempre se hace la derivación por la derecha primero (R) y necesita un símbolo de antelación para poder realizar el análisis sin ambigüedad.

Tanto [ASU90] como [CT12] dedican una extensa sección al estudio de los algorítmos para el análisis sintactico.

En general, los algoritmos de análisis sintáctico ascendentes utilizan un autómata de pila. Éste normalmente se representa con una matriz de transiciones y una pila. Y para facilitar su entendimiento se dividió el método en dos partes.

En una primera parte se hace el análisis de la gramática lo cual producirá la matriz de transiciones. En la segunda parte se utiliza la matriz junto con una pila para construir el árbol sintáctico.

La primera parte puede ser separada totalmente del compilador como un programa independiente. Esto fue precisamente lo que se hizo. Se implementó el análisis de la gramática en un programa independiente que genera un archivo que contiene la matriz de transiciones. Este archivo es posteriormente cargado por el compilador para poder realiza el análisis sintáctico.

Lo que hace el programa analizador de gramáticas es construir un autómata de pila donde los estados son las posibles derivaciones que puede generar una gramática. Este método es bastante extenso y complejo como para describirlo completamente en este trabajo, pero se puede analizar el código del programa escrito en Python en el cual se modifican algunos de los algoritmos que exponen Cooper y Torczon en [CT12]. Por ejemplo, en el listado 5.4.

Listing 5.4: Algoritmo de clausura.

```
closure(s)
  while (s is still changing)
      for each item [A → β • Cδ, a] ∈ s
        for each production C → γ ∈ P
          for each b ∈ FIRST(δa)
            s ← s ∪ [C → •γ, b]
```

Este algoritmo fue implementado usando una cola en vez de checar si s había cambiado. El código de la implementación de este algoritmo es bastante extenso como para mostrarlo aquí, sin embargo, se puede descargar y analizar el código del analizador de gramáticas en `http://biogb.sourceforge.net`.

Al final, dentro del compilador, ya usando la matriz de transiciones dividida en dos partes: *Action* y *Goto* se implementó el algoritmo que sugieren tanto Cooper y Torczon así como Grune et al. [GBJL07] entre otros. En el listado 5.5 se muestra dicho algoritmo.

Listing 5.5: Algoritmo de clausura.

```
push EOF
push start state, s0;
while (true) do
  state ← top of stack
  if Action[state,word] = "reduceA → β" then begin
    pop 2 × |β| symbols
```

```
    state ← top of stack
    push A
    push Goto[state, A]
  end
  else if Action[state, word] = "shifts_i" then begin
    push word
    push s_i
    word ← NextWord()
  end
  else if Action[state, word] ="accept"
    then break
  else Fail()
end
report success
```

En resumen el algoritmo lee un *token* del analizador léxico y realiza una de las siguientes acicones:

- Mete el token a la pila junto con un estado nuevo que encontramos en la matriz Goto.
- Saca varios elementos de la pila y hace una reducción, esto es, crea un nodo padre que enlaza a los nodos hijos que fueron sacados de la pila.
- Muestra un mensaje de error porque no se pudo realizar ninguna de las acciones anteriores.

El algorítmo se le conoce como *"shift and reduce"(reducción y desplazamiento)* y es ampliamente usado en todo tipo de compiladores.

5.2.3. Análisis semántico

El análisis semántico es una parte del compilador que en algunas implementaciones va muy ligado tanto al análisis sintáctico como a la generación de código. Es una de las partes más flexibles que tiene el compilador, sin embargo existen varias funciones con las que debe cumplir forzosamente:

- Verificación de tipos.
- Verificación de número de parámetros en rutinas y funciones.
- Administración de tabla de símbolos.

Existen varios mecanísmos para realizar el análisis semántico, entre ellos está el uso de las gramáticas de atributos y el análisis ad hoc (o directo). Ambas se basan en agregar información a los nodos del árbol que va produciendo el analizador sintáctico. Sin embargo, la mayoría de las veces estos árboles contienen una gran cantidad de nodos inecesarios por lo que otra de las funciones del analizador semántico es crear una o varias representaciones intermedias que al final nos servirán para generar el código final.

La técnica de uso de gramáticas de atributos, aunque poderosa, genera una gran complejidad tanto en la gramática como en el compilador. Por esta razón se decidió usar un método más sencillo comunmente llamado análisis ad hoc.

En realidad, el análisis ad hoc es una versión simplificada del uso de gramáticas de atributos. Como su nombre indica, las gramáticas de atributos asignan atributos a cada nodo del árbol sintáctico y al mismo tiempo asignan acciones semánticas a cada regla gramatical. La diferencia radica en que en el uso de gramáticas de atributos, estos (los atributos) pueden ser construidos a partir de nodos hijos y nodos padres, en el método ad hoc, estos sólo se construyen a partir de los nodos hijos. De esta manera, el análisis se vuelve una función recursiva. Y en el caso de las gramáticas de atributos, se debe realizar un ordenamiento topológico entre los nodos y posteriormente ejecutar las acciones semánticas relacionadas con cada nodo.

Las acciones semánticas son funciones que toman como parámetros los atributos de los nodos hijos y pueden crear un nuevo nodo con la información condensada de los nodos hijos o simplemente asignar un nuevo atributo al nodo en cuestión. En si las acciones semánticas son las que construyen las diferentes representaciones intermedias. En el caso del compilador de este trabajo, una acción semántica es un bloque de código que se va a ejecutar al reducir cierta producción. Una forma típica de implementarlo es con una tabla *switch...case*, sin embargo, Python no soporta esta estructura, por lo que la implementación se hace con estructuras *if...else*.

El listado 5.6 muestra algunos ejemplos en pseudo-código de acciones semánticas para el compilador de LGE. Como se puede observar, el código que hay dentro del primer *if* es la acción semántica ligada a la regla gramatical **expresión-primaria**. En este caso la acción semántica regresa nodos nuevos con la información de los nodos hijos. En si, este es un fragmento del código que realiza la simplificación y anotación del árbol sintáctico.

Listing 5.6: Acciones semánticas.

```
if Grammar Rule == "expresion-primaria":
    if FirstChild.Name == "identificador":
```

```
            return CreateNode(FirstChild.Value)
        elif FirstChildren.Name == "valor":
            if FirstChild.Value <= 0xFF:
                return CreateNode(FirstChild.Value,
                                  "byte")
            else:
                return CreateNode(FirstChild.Value,
                                  "word")
        else:
            return SecondChild
```

Existen una gran variedad de representaciones intermedias, y gran parte de la investigación sobre compiladores radica en esa área (vease [CT12]). La gran mayoría tienen como objetivo realizar algún tipo de optimización o simplificación del código final. En este caso se usará una representación intermedia en forma de árbol. Este árbol es una forma compacta y anotada (con atributos) del árbol sintáctico.

La parte final del análisis semántico es recorrer este nuevo árbol. Al recorrerlo de forma recursiva se van almacenando los nombres de funciones, rutinas y variables en una tabla de símbolos y se va checando que su uso sea correcto: que las variables sean del tipo correcto, que el número de parámetros sea el adecuado, que las variables estén definidas antes de usarse, etcétera. Como el árbol se recorre en preorden, este análisis se vuelve bastante trivial. Sin embargo, parte de la dificultad de implementar un analizador semántico es la programación de estructuras de datos y algoritmos que administren el control de acceso a variables, la tabla de símbolos, la asignación de direcciones de memoria para variables, entre otras. Muchas de las técnicas que se usan hoy en día se pueden encontrar en los diferentes textos sobre compiladores como [GBJL07], [ASU90] [CT12] y [Hol90].

Para este trabajo se utilizaron métodos sencillos y legibles, pero siguen siendo muy extensos para mostrarse aquí, por lo que se puede ver el código del compilador en el sitio anteriormente mencionado para analizar su funcionamiento.

5.2.4. Generación de código

El generador de código normalmente forma parte del analizador semántico ya que en varias implementaciones éste genera una representación intermedia en forma de código. El generador de código es la parte más variable de un compilador. Cuando un generador de código genera una repre-

sentación intermedia, entonces esta y todas las partes anteriores del compilador pertenecen a lo que se conoce como ***frontend*** o *interfaz*. Cuando el compilador es diseñado de esta forma, debe existir una parte más que traduzca la representación intermedia en el código máquina correspondiente. La ventaja es que la interfaz puede ser totalmente portable y se pueden tener diferentes traductores que puedan producir código ensamblador para diferentes máquinas.

En el caso opuesto, donde el generador de código genera el código ensamblador directamente, este va a pertenecer al ***backend*** o *motor*. La principal ventaja es que no se necesita una nueva parte del compilador que traduzca el código, por lo que tenemos una mayor eficiencia.

Para este trabajo el generador de código produce código ensamblador directamente ya que el compilador es de propósito específico y no está diseñado para compilar para diferentes arquitecturas. Aunque en este trabajo no se realizan, otra de las principales ventajas de este diseño es que se pueden implementar algunas optimizaciones extra en el código generado, lo cual resultaría en código ensamblador más robusto y eficiente.

La implementación del generador de código se realizó sobre el analizador semántico. Al ir recorriendo el árbol simplificado, hay nodos que tienen la indicación de llamar al generador de código para que en conjunto con las tablas de símbolos y de direcciones genere el código final que será ensamblado.

Hay mucho que decir también sobre la generación de código ensamblador ya que existen problemas como la asignación óptima de registros, el uso eficiente de la memoria, la selección de instrucciones, etcétera. Para cada problema existe todo un abanico de soluciones, sin embargo, este tema sale del objetivo de este trabajo por lo cual no profundizo mas en este. Aun así es importante mencionar que existe un blog en línea donde se ha ido describiendo de manera mucho más detallada la construcción del compilador referente a este trabajo. Hasta el día en que se escribió esto, el blog aun no está terminado, pero se pueden ir consultado las diferentes entradas en `http://instru-dospro.blogspot.mx/`.

Capítulo 6

El emulador

Teniendo ya las herramientas disponibles lo común sería probar y ejecutar diferentes programas en el Game Boy, la máquina simple. Sin embargo, una importante característica del Game Boy es la del uso de cartuchos. Esta característica se convierte en una desventaja a la hora de querer desarrollar programas ya que es necesario programar un chip ROM, montarlo en un circuito e insertarlo en la ranura del Game Boy. Este proceso se tendría que repetir completamente cada vez que quisieramos probar un programa. Además, no dispondríamos de ningun método eficiente de depuración o retroalimentación.

Anteriormente lo que se usaba eran kits especiales de desarrollo. En ese entonces Nintendo, dentro de las licencias que vendía a los diferentes desarrolladores, incluía también un kit que constaba del hardware de un Game Boy conectado a otras placas y circuitos que se conectaban a una computadora y por la cual se podía recibir información de depuración entre varias cosas. Desafortunadamente este kit sólo estaba disponible para los que pudieran pagar una licencia de desarrollo a Nintendo, creador del Game Boy. A pesar de esto, hubo personas que lograron construir su propio kit de desarrollo. En el siguiente capítulo se hablará un poco más de estos kits.

Con el rápido avance en las capacidades de las computadoras, una nueva opción para el desarrollo en máquinas más sencillas fue la emulación. Aunque existen diferentes tipos de emulación, la que ha resultado más útil y la que se utilizará en este trabajo es la emulación por software. A diferencia de la emulación por hardware, el emulador por software es un programa que imita el funcionamiento de una máquina e incluso puede mejorar algunas funciones o dar información extra sobre el programa. Más adelante se describen las diferentes posibilidades que se tienen con un emulador por software.

Básicamente, un emulador de Game Boy es un programa que se va a ejecutar en nuestro sistema operativo de preferencia (Windows, Linux, iOS, Android, etcétera) y que va a recibir como entrada un programa desarrollado con las herramientas descritas en los capítulos pasados. El emulador va a ejecutar dicho programa como si se tratara de un Game Boy. En este capítulo se describen las ideas que están detrás del desarrollo de un emulador de Game Boy.

6.1. Intérpretes y recompiladores

Dentro de los emuladores por software existen dos clases generales: los intérpretes y los recompiladores. Los intérpretes, como su nombre indica, interpretan cada instrucción de nuestro programa y ejecutan la acción correspondiente dentro de los registros y áreas de memoria virtuales en el emulador. En cuanto a los recompiladores o JITs (Just in Time Compilation), existen varios tipos. La idea central es que el emulador reconoce bloques de código máquina y los traduce en unidades de ejecución del sistema anfitrión, todo esto antes de comenzar la ejecución del programa. Para estudiar a detalle las diferentes técnicas de emulación se recomienda ampliamente [dB01].

Las principales ventajas de los intérpretes son las siguientes:

- Son sencillos de implementar.
- Pueden ser escritos con código portable.
- Es mucho más sencillo agregar opciones de depuración y retroalimentación.
- Pueden ser extremadamente precisos.

La principal desventaja de un intérprete es la velocidad. Algunos estudios revelan que se necesita una computadora anfitrión hasta 10 veces más potente que la emulada para que la emulación se realice sin problemas [dB01].

En cuanto a los recompiladores, su principal ventaja es precisamente la gran velocidad de ejecución que pueden generar. Sin embargo esta ventaja viene con un costo caro:

- Son bastante complejos de implementar.
- No siempre es posible implementarlos.
- Rara vez se puede lograr que sean portables.

- Dificilmente pueden ser precisos.

Considerando que el Game Boy tiene un microprocesador de 8 bits a 4Mhz y que la memoria está en el orden de los kilobytes, la opción definitiva es desarrollar un intérprete. A continuación se describe el desarrollo del emulador de Game Boy.

6.2. Diseño y características del emulador

Diseñar y programar un emulador no es nada sencillo ya que se necesita contar con información muy específica del hardware la cual no siempre está disponible. Aparte de tener la información, es necesario comprender los detalles del funcionamiento de la máquina a emular. Afortunadamente el Game Boy tiene un diseño y una arquitectura sencilla y elegante lo que hace fácil comprenderla rápidamente. Por lo tanto, para este trabajo se desarrolló un emulador que imita de manera precisa las funciones del hardware del Game Boy. A continuación se enlistan las características que tiene así como su justificación.

- Portable.
- Eficiente.
- Código legible.
- Expandible.

6.2.1. Portabilidad

Una de las características prioritarias del emualdor de Game Boy es que sea portable. Esto se debe a que como herramienta de aprendizaje y de desarrollo, debe estar disponible en los diferentes sistemas que existen. Para lograr esta portabilidad, se consideraron los lenguajes de programación: C, C++, Java y Python. Otra parte importante que se consideró para la portabilidad del emulador fueron las librerías gráficas, de sonido y de entrada. Más adelante se mencionan las que utilizadas y por qué.

En primer instancia, se dejó de considerar Python como posible lenguaje para implementar un emulador. Esto se debe a que los demás candidatos tienen un mejor manejo de datos y operaciones a bajo nivel, característica necesaria en un emulador. En segunda instancia se decidió no implementar el emulador en Java, ya que a pesar de que Java es un lenguaje portable

gracias a su máquina virtual, no todos los sistemas tienen una, o en dado caso tienen versiones reducidas o con muchos errores.

6.2.2. Eficiencia

Otro de los aspectos importantes fue la eficiencia. Afortunadamente, las computadoras de hoy en día son suficientemente poderosas para poder correr emuladores de máquinas muy poderosas, sin embargo vuelve a salir el punto anterior. Las herramientas están pensadas para ser ejecutadas en una amplia variedad de sistemas. Por lo tanto, no se puede suponer que siempre se tendrá una computadora con características de sobra para correr el emulador. Por esta razón se decidió no utilizar tecnologías como Action Scripts dentro de Flash, o Javascript en un navegador web, entre otras. Los lenguajes C y C++ se caracterizan por producir código bastante eficiente por lo que siguen dentro de la lista de candidatos.

6.2.3. Legibilidad

Antes de que se empezara a desarrollar el emulador de Game Boy, se analizaron códigos de otros emuladores publicados en internet. Una característica común fue que su código era terriblemente ilegible e indocumentado. Fueron pocos los emuladores que tenían algún tipo de documentación.

Por otra parte, se busca también que el emulador sea expandible. Para lograr esto, es preciso que el código sea entendible y pueda ser modificado eficientemente sin tener efectos secundarios a la hora de la compilación o la ejecución. Hay mencionar que un tiempo después de la publicación del emulador de Game Boy que expuesto, Alex Radocea, un estuiante de Ciencias de la Computación e Ingeniería en Rensselaer Polytechnic University se interesó en el proyecto y lo utilizó agregándole algunas características para un trabajo sobre ingeniería inversa. Su trabajo se puede encontrar en el siguiente URL: (`http://codegate.org/renew/_file/board/dr_2688410.ppt`).

6.2.4. Expansibilidad

Como ya se mencionó, una de las características importantes es que los usuarios puedan agregar funcionalidades al emulador de manera sencilla. Esta característica fue la que impulsó a utilizar C++ y el modelo de programación orientado a objetos. Gracias a la abstracción que se puede generar con este modelo, un usuario no necesita comprender todo el código para poder hacer una modificación. En la mayoría de los casos bastará con heredar una clase para agregar la funcionalidad correspondiente.

Una vez establecidas las características del emulador se exponen las siguientes desiciones de diseño:

- Lenguaje de programación C++.
- Paradigma orientado a objetos.
- Simple Direct Layer (SDL) como librería de gráficos, sonido y entrada.
- Aplicación de consola (no ventanas o interfaces de usario).

La librería elegida para emular los gráficos, el sonido y los dispositivos de entrada fue la SDL (Simple Direct Layer). La razón de esta elección, es que SDL es un librería totalmente portable, sencilla y eficiente. Aparte, en una sóla librería se tienen los 3 componentes incluidos: gráficos, sonido y dispositivos de entrada.

Para poder diseñar el emulador en un contexto orientado a objetos se hizo la abstracción de los siguientes componentes:

- Módulo de la Unidad Central de Procesamiento (CPU).
- Módulo de memoria y BUSes de datos y direcciones.
- Módulo de gráficos.
- Módulo de sonido.
- Módulo de entradas.

Analizando las funciones de cada módulo se decidió unir el módulo de CPU con el de memoria. Esto simplifica su implementación sin perder mucha legibilidad. A continuación se describe de manera resumida el funcionamiento de cada módulo y al final como trabajan juntos para lograr la emulación.

6.3. Componentes del emulador

Es importante mencionar que los módulos fueron implementados como clases en C++. De ahora en adelante se usará el termino clase en vez de módulo para los componentes del emulador.

6.3.1. CPU y memoria

Esta clase es la más importante de todas ya que todas los demás dependen del buen funcionamiento de esta. El tipo de relación que hay entre esta clase y las demás es de composición, es decir, la clase de CPU y memoria está compuesta por objetos de las demás clases que representan los dierentes módulos. Dentro de esta clase se tienen los siguientes comportamientos que serán implementados por diferentes métodos.

- Cargar ROM (programa)
- Inicializar todos los objetos de la composición.
- Realizar un cíclo de emulación.

Estas son los principales comportamientos públicos de la clase CPU y memoria. Dentro de los privados están la interfaz de memoria y los mecanismos que emulan cada una de las instrucciones.

Para la emulación de la memoria, se utilizó un arreglo de bytes que representa todo el mapa de memoria del Game Boy, incluyendo los posibles bancos. Junto con el arreglo, se implementaron dos métodos importantes, uno para leer un byte del mapa de memoria y otro para escribirlo. La razón por la que se implementaron estas 2 funciones como métodos es que realizan gran parte de la comunicación con el hardware emulado. Esto se debe a que el hardware esta "mapeado" en el mapa de memoria. En otras palabras, existen localidades en el mapa de memoria que nos sirven para comunicarnos con el hardware en general. En el listado 6.1 se muestra un fragmento de pseudo-código de las funciones descritas anteriormente.

```
readByte( address )
        if address is in ROM
                return ROM byte in memoryMap[address]
        else if address in RAM
                return RAM byte in memoryMap[address]
        else if address is IO hardware
                return hardware response

writeByte( address, value )
        if address is in ROM
                perform ROM banking
        else if address in RAM
                write value in RAM[address]
```

```
        else if address is IO hardware
                return hardware response
```

Listing 6.1: Leer y escribir un byte.

Otra parte importante de esta clase es el método que ejecuta un ciclo de emulación. Este método es la función más importante del emulador ya que pone en funcionamiento todas sus partes. Sin embargo, este método sólo realiza un ciclo de ejecución. Para mantener la emulación es necesario ejecutar este método de manera iterativa. Este diseño permite pausar la emulación en el momento que lo necesitemos así como incrustar la ejecución dentro de algún framework como el del S.O. Android o Windows. El pseudocódigo de este método se muestra en el listado 6.2

```
doCycle
  read opcode from ROM
  execute opcode
  update timers
  update graphics

  read inputs
  paint graphics
  update sound
```

Listing 6.2: Ejecución de ciclo.

6.3.2. Gráficos, sonido y entradas

Las tres clases de esta sección tienen un diseño similar, por eso se describen las tres en esta sección. La función de las tres es recibir los mensajes correspondientes de la clase CPU y memoria, y construir salidas compatibles con el sistema anfitrión. En el caso de la clase de entradas, también recibe mensajes del sistema anfitrión los cuales son traducidos y devueltos a la clase de CPU y memoria. Dentro del diseño orientado a objetos, se decidió que estas clases fueran abstractas, es decir, no todos sus métodos están implementados. Con esta característica del diseño se puede implementar la producción de sonido y gráficos con la librería y el método que se quiera al igual que el procesamiento de entradas. Sólo basta heredar una de las clases e implementar los métodos que muestran la salida del emulador. En el listado 6.3 se da un ejemplo del diseño explicado.

```
class emuGraphics
```

```
  data

  interface:
    init
    recieve message
    *abstract draw screen

class SDLGraphics : emuGraphics
  data

  interface:
    draw screen
```

Listing 6.3: Diseño oerientado a objetos.

6.3.3. Trabajo conjunto de las clases

Teniendo todas las clases implementadas correctamente, se vuelve bastante sencillo realizar un programa que las utilice y haga la emulación. En el listado 6.4 se muestra un fragmento de pseudo-código típico para correr el emulador.

```
class CPUMemory
  GraphicsObject
  SoundObject
  InputObject

  interface:
    Init
    Do Cycle
    Exit


main:
  CPUMemoryObject

  CPUMemoryObject.Init

  while(emulator is running)
    CPUMemoryObject.Do Cycle
```

```
CPUMemoryObject.Exit
```

Listing 6.4: Programa principal

La programación de emuladores es todo un campo de actividad que tiene su aplicación en el área de virtualización. En este trabajo sólo se describió de manera resumida algunos aspectos importantes a la hora de programar el emulador de Game Boy así como un posible modelo orientado a objetos. Para conocer más acerca de la emulación se puede consultar [dB01]. También se puede consultar la documentación del Game Boy en [AGF+99] para poder profundizar en su funcionamiento. Finalmente, se puede consultar el código del emulador de Game Boy desarrollado para este trabajo en `http://biogb.sourceforge.net`.

Capítulo 7

Otras herramientas

Para poder sacar el máximo provecho tanto de las herramientas como del curso que se expone en el siguiente capítulo, se consideró necesario incluir un grupo extra de programas y subproyectos que complementan perfectamente las herramientas ya descritas.

Las herramientas se dividieron en dos grupos: herramientas de hardware y herramientas de software.

7.1. Herramientas de hardware

Las herramientas de hardware para este trabajo son varios subproyectos de electrónica diseñados y construidos para poder probar los programas de manera física en un GameBoy. En realidad todo este conjunto de proyectos junto con partes de software que se describen más adelante conforman un kit de programación para Game Boy. En esta sección se expone la construcción y conformación de dicho kit.

7.1.1. Comunicación con el Game Boy

El primer problema a enfrentar fue la comunicación con el Game Boy. Afortunadamente Martin Korth en su recopilación de documentos sobre el funcionamiento del Game Boy describe brevemente los conectores de sus cartuchos[1]. También Alex dentro de la documentación que describe su proyecto *GBCartRead: Adruino based Game Boy cart reader* hace una excelente de-

[1] `http://problemkaputt.de/pandocs.htm#externalconnectors`

scripción del funcionamiento de estos conectores[2].

Por otro lado, también se necesita un lugar donde almacenar el programa para que pueda ser leído por el Game Boy. Para esto se usó un chip ROM. Sin embargo este chip debía cumplir varias características; una de ellas era precisamente que fuera compatible con la velocidad de lectura del microprocesador del Game Boy. Afortunadamente en México fue sencillo encontrar un chip ROM que cumpliera con esta característica: el **Am29F010B**.

El chip Am29F010B puede contener programas de hasta un megabit lo cual es más que suficiente para los objetivos de este trabajo. AMD, su fabricante, proporciona excelente documentación para el manejo de este chip la cual se utilizó para la elaboración del proyecto.

Con estas dos partes de información se diseñó un circuito eléctrico básico que pudiera conectarse directamente al Game Boy y poder leer el programa desde el chip ROM. En la figura 7.1 se muestra el diagrama de este circuito. En la figuras 7.2 y 7.3 se muestran fotos del circuito ya construido.

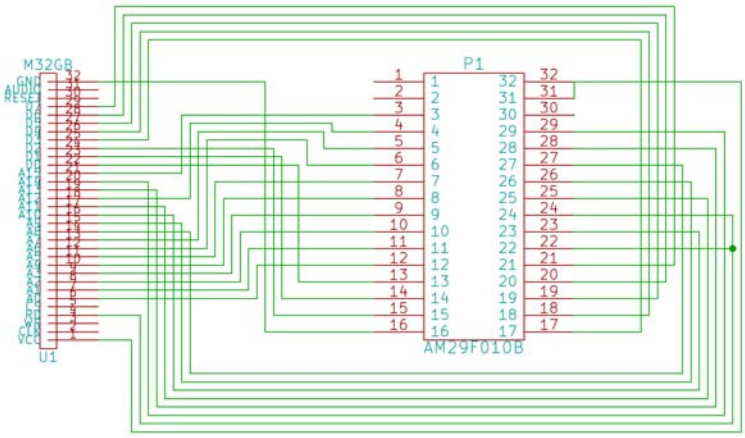


Figura 7.1: Esquema del circuito.

Por otro lado, aun falta poder almacenar el programa en el chip ROM antes de integrarlo en el circuito. Para esto existen varios equipos comerciales llamados *programadores* que hacen un excelente trabajo. En la mayoría de los casos esta es una excelente opción, pero algo costosa. En este caso se decidió utilizar una placa Arduino. La principal razón de esto es que ya se contaba con una de estas placas. De esta manera no se tendría que gastar en un programador y por otro lado el método que usado es aplicable

[2] `http://www.insidegadgets.com/2011/03/19/gbcartread-arduino-based-gameboy-cart-reader-%E2%80%93-part-1-read-the-rom/`

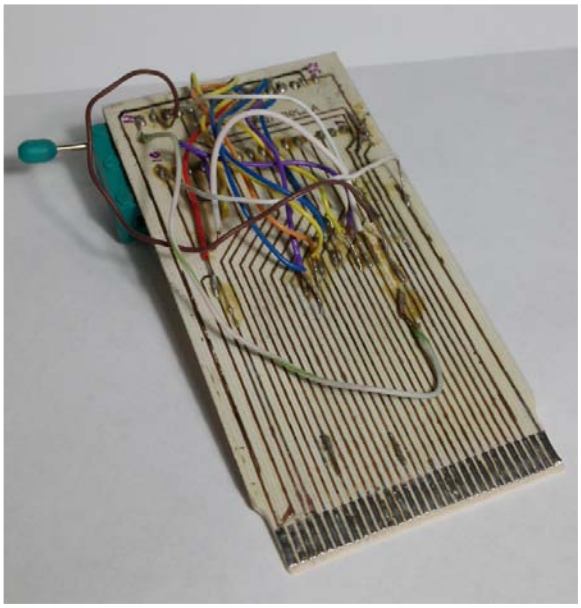


Figura 7.2: Foto del circuito construido.

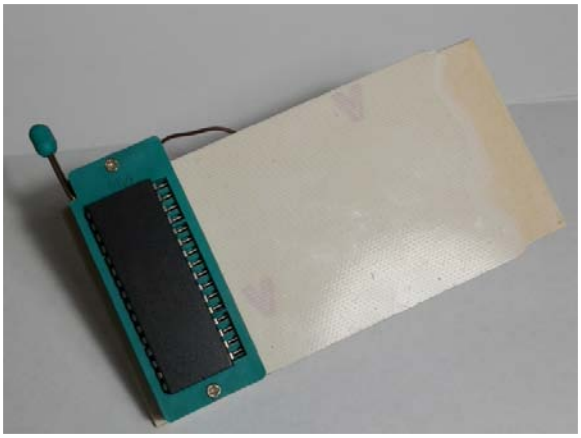


Figura 7.3: Foto del circuito construido.

a prácticamente todos los clones de Arduino y placas similares que son en general menos costosas que un programador. Cabe mencionar que no se encontró suficiente información tanto en linea como en libros que guiara en la elaboración de este método. Existen algunos métodos similares que funcionan para otros chips ROMs pero no para el Am29F010B. A continuación se

describe el método que desarrollado:

7.1.2. Programación del chip ROM

La placa de arduino más común es la Arduino UNO que se muestra en la figura 7.4.

Figura 7.4: Foto de un Arduino UNO

Para usar una de estas placas es necesario adquirir unos circuitos integrados llamados *shift registers* de 8 bits. Esto se debe a que el Arduino UNO no tiene los suficientes puertos para poder comunicarse con el chip ROM. Con los shift registers se puede simular una comunicación en paralelo utilizando datos en serie que pasan a través de estos circuitos. Se puede encontrar bastante información sobre los circuitos shift en general en internet, pero se recomienda ampliamente el texto de Floyd [Flo08].

Una alternativa, la cual fue la que se usó, fue usar la placa Arduino MEGA que se muestra en la figura 7.5.

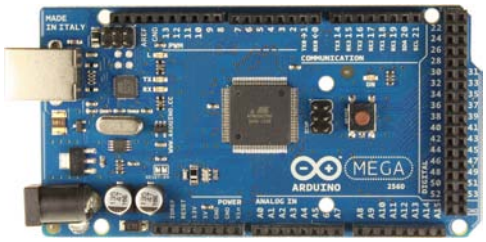


Figura 7.5: Foto de un Arduino MEGA

Esta placa tiene bastantes puertos. Para el chip Am29F010B se necesitan 30 puertos para poder escribir un programa correctamente en el chip. El Arduino MEGA tiene 52 puertos digitales de uso general.

En la figura 7.6 se muestra el diagrama del chip Am29F010B el cual tiene 32 pines de comunicación de los cuales dos, marcados como NC, no tienen uso.

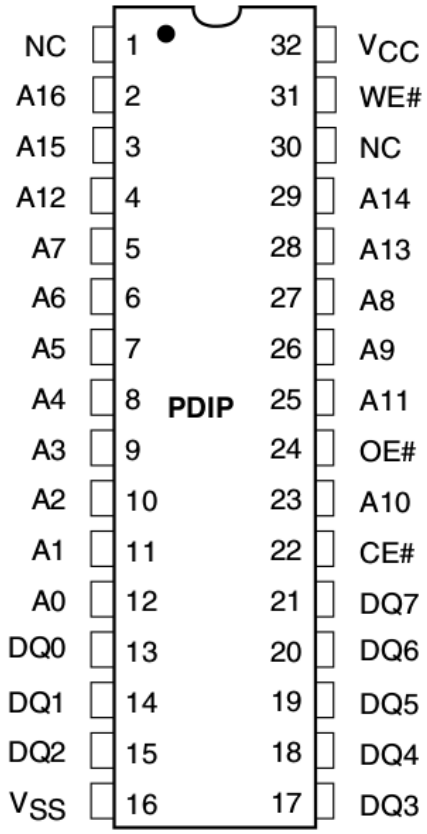


Figura 7.6: Diagrama del Am29F010B

La idea es conectar los 30 pines restantes con puertos del Arduino MEGA. Es importante notar que el pin VCC debe ir conectado al puerto que proporcione los 5 volts mientras que el pin VSS debe ir conectado al puerto tierra GND. Fuera de estas restricciones, los demás pines pueden ir conectados en cualquiera de los puertos digitales del Arduino. En la figura 7.7 se muestra una foto de como se vería el circuito montado en una placa de prototipos (protoboard).

Volviendo a la figura 7.6, hay que notar los diferentes nombres de los pines del chip. Estos se pueden agrupar en tres categorías:

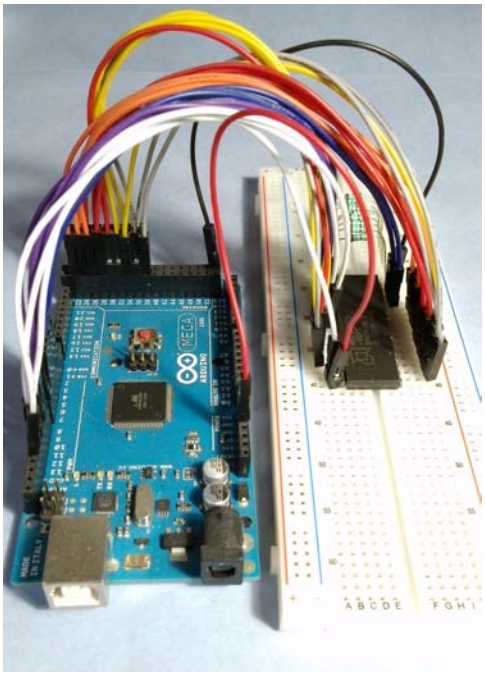


Figura 7.7: Conexión del Arduino MEGA con el chip Am29F010B

- Direcciones (A0 - A16).
- Datos (DQ0 - DQ7)
- Control (CE#, OE#, WE#)

La primer categoría representa el BUS de direcciones que en este caso es de 17 bits. La segunda categoría representa el BUS de datos de 8 bits. Ambos BUSes junto con los bits de control son los que van a permitir la comunicación con el chip ROM.

Analizando la documentación de AMD se puede encontrar la siguiente información:

> El dispositivo es totalmente compatible con el conjunto de comandos del estándar JEDEC para Flash de una sola fuente.

Esto significa que cada operación es activada por medio de un comando. El chip reconoce una amplia variedad de comandos, sin embargo, los que interesan son tres: lectura, escritura y borrado. Cada uno de estos comandos

se activan por medio de pulsos digitales en los pines de control. Arduino puede producir estos pulsos fácilmente por medio de sus puertos digitales. Por ejemplo, para leer un byte de una dirección dentro del chip se utiliza el algoritmo que se muestra en el listado 7.1

```
set_OE(HIGH);
set_CE(LOW);
set_WE(HIGH);

setAddressBUS(address);
set_OE(LOW);

data = ((digitalRead(BUS_DQ7) << 7) |
     (digitalRead(BUS_DQ6) << 6) |
     (digitalRead(BUS_DQ5) << 5) |
     (digitalRead(BUS_DQ4) << 4) |
     (digitalRead(BUS_DQ3) << 3) |
     (digitalRead(BUS_DQ2) << 2) |
     (digitalRead(BUS_DQ1) << 1) |
     digitalRead(BUS_DQ0));

set_OE(HIGH);
```

Listing 7.1: Comando de lectura

En este listado, las instrucciones set_OE, set_CE y set_WE mandan señales binarias a los pines con sus respectivos nombres, HIGH siendo un impulso equivalente a uno y LOW equivalente a cero. Después setAddressBUS escribe en el bus de direcciones la dirección de donde se desea leer el byte. Finalmente se lee el byte agrupando todos los bits del BUS de datos. Esto se logra con las operaciones de Ó lógico y desplazamiento de bits. La secuencia de impulsos altos y bajos en los pines de control son necesarios y deben ser ejecutados en ese orden ya que así lo dicta la documentación del chip. Se puede consultar dicha documentación para ver los detalles del funcionamiento del sistema de comandos.

Este algoritmo junto con otros para la escritura y el borrado del chip se implementan en un programa en C que se ejecutará en la placa Arduino. De esta manera se dan las instrucciones al microprocesador del Arduino para comunicarse con el chip ROM. Finalmente, el kit de desarrollo de Arduino proporciona librerías de comunicación serial las cuales fueron utilizadas para poder transferir el programa de Game Boy desde la computadora al Arduino

y desde el Arduino escribirlo en el chip ROM.

Esta es la operación elemental para poder transferir un programa al chip ROM, sin embargo es importante implementar algoritmos que revisen que la transferencia se haya hecho de manera correcta, es decir, sin errores. La implementación de estos algoritmos queda fuera del objetivo de este trabajo por lo que no se exponen aquí. Se puede consultar la documentación de AMD sobre el chip Am29F010B para conocer los diferentes métodos que usa para la comprobación de datos.

Ya que se tiene el chip ROM con el programa, sólo queda montarlo en el circuito. Este circuito puede conectarse correctamente al Game Boy gracias a su diseño. Una vez que el circuito está conectado; al encender el Game Boy, este debe ejecutar el programa.

Con esto queda concluido el kit básico de programación para Game Boy usando una placa Arduino MEGA.

7.2. Herramientas de software

Todas las herramientas desarrolladas en los capítulos anteriores han sido en software, sin embargo, existe un sin fin de posibles herramientas en software que nos pueden ayudar en la programación para Game Boy. Para este trabajo se consideraron dos de gran importancia. En primer lugar, un editor de gráficos y en segundo lugar un depurador.

7.2.1. Editor de gráficos

La importancia del editor de gráficos radica en que el Game Boy no usa un formato convencional para almacenar y pintar gráficos en pantalla. A pesar de que ya existen varias herramientas que ayudan con esta tarea, se decidió programar la propia.

La idea básica del editor de gráficos es tener una interfaz sencilla que permita dibujar patrones de manera intuitiva y que al final el programa pueda producir código ensamblador que genere el mismo patrón dentro del Game Boy.

La programación del editor se basó en el la documentación del Game Boy [AGF+99]. En la sección 2.8 de dicho documento viene la especificación del formato para los gráficos de Game Boy. En el editor se utilizó esta información para poder producir el código ensamblador necesario.

Se recomienda descargar y leer el código fuente de dicho programa para

profundizar sobre el algoritmo que se utilizó[3].

7.2.2. Depurador

Finalmente, la segunda herramienta expuesta y la que se considera de mayor importancia es el depurador. A continuación se mencionan las ventajas de tener un depurador.

Un depurador va a ayudar a encontrar errores y comportamientos indeseables en los programas. También proporciona información sobre el contenido de la memoria RAM, VRAM, etcétera. Uno de los mejores depuradores que encontrados es no$gmb desarrollado por Martin Korth. Este software aparte de ser un excelente emulador de Game Boy, incluye un gran conjunto de opciones de depuración, desde breakpoints y tracepoints hasta un poderoso desensamblador.

Todo este conjunto de herramientas va a facilitar la programación y el entendimiento de la arquitectura del Game Boy, la máquina simple.

[3] `http://biogb.sourcefoge.com`

Capítulo 8

Pruebas y resultados

En los capítulos anteriores se explicó el desarrollo de las herramientas y la elección de la máquina simple. A través de cada uno de estos capítulos se fueron fundamentando y justificando las decisiones que se tomaron para cumplir con el objetivo. Sin embargo, se consideró necesario realizar pruebas que puedan confirmar o refutar la hipótesis planteada.

En este caso, dada la restricción de recursos de tiempo y espacios para poder realizar un experimento cuantitativo, se optó por un experimento cualitativo, el cual, como se verá a continuación, arrojó resultados interesantes.

El experimento diseñado consistió en un curso de prueba que se explica a continuación.

8.1. Diseño del curso de prueba

En este curso se hizo una prueba de las herramientas. El experimento consistió en dar un curso a un grupo de estudiantes donde se mostraron las herramientas desarrolladas para este trabajo. También se expuso la arquitectura de la máquina así como el funcionamiento del emulador. El curso consistió de las siguientes secciones:

1. Exposición de la arquitectura del Game Boy.

2. Curso de ensamblador.

3. Curso de lenguaje genérico estructurado.

4. Evaluación.

8.1.1. Objetivo

El objetivo de este curso consistió en mostrar las herramientas y describir la arquitectura de la máquina para que los alumnos por si sólos desarrollen programas en ensamblador y en el lenguaje genérico estructurado.

La idea fue que los alumnos se fueran familiarizando con los conceptos de arquitectura de computadoras mientras desarrollaban programas en los lenguajes de bajo nivel. Al final se hizo una evaluación para obtener una retroalimentación del uso de las herramientas y del curso. A continuación se describen las etapas del curso como se diseñaron.

8.1.2. Exposición de la arquitectura del Game Boy

Esta sección tiene una duración total de una hora.

Se da un panorama general de la arquitectura del Game Boy. No es necesario explicar todos los detalles ya que esos deben ser descubiertos por los estudiantes a la hora de programar. Los temas que se tratan son los siguientes:

- Describir el mapa de memoria.
- Explicar brevemente el sistema de entrada y salida.
- Explicar el proceso de arranque de un programa.
- Explicar el funcionamiento del display y sus características.

Los conocimientos adquiridos en esta introducción son suficientes para que los alumnos estén preparados para la parte del lenguaje ensamblador.

8.1.3. Curso de ensamblador

El curso tiene tener una duración mínima de 2 horas.

En este curso básico de ensamblador se enseña un subconjunto de las instrucciones del microprocesador. Para eso se usará la siguiente clasificación:

1. Comandos de movimiento de datos de 8 bits.
2. Comandos de saltos y rutinas.
3. Comandos de aritmética de 8 bits.
4. Comandos de movimiento de datos de 16 bits.
5. Comandos de aritmética de 16 bits.

6. Comandos de manipulación de bits.

7. Comandos de control del microprocesador.

En este caso el orden es importante ya que permite realizar programas sofisticados desde las primeras clasificaciones.

Los alumnos por si sólos deben descubrir las instrucciones faltantes. No es necesario que los alumnos memoricen todas las instrucciones. De hecho, se sugiere que cada quien tenga a la mano un listado de ellas para consultarlas en cualquier momento.

La exposición de las instrucciones debe ser en conjunto con algoritmos básicos en lenguaje ensamblador para la manipulación de toda la computadora.

Al final de este curso los alumnos deben ser capaces de realizar programas en lenguaje ensamblador, usar las herramientas desarrolladas en este trabajo y entender la arquitectura de un Game Boy.

8.1.4. Curso de lenguaje genérico estructurado

Esta sección tiene una duración mínima de una hora.

El objetivo de este curso es mostrar las estructuras de LGE para que los alumnos logren realizar programas estructurados y legibles manteniéndose cerca del hardware y aprovechando todas las capacidades de la computadora.

Para este curso se da por hecho que los alumnos saben programar en algún lenguaje estructurado de alto nivel, por ejemplo, C, C++, Java, o similares. Los temas a tratar son lo siguientes:

1. Mostrar el funcionamiento del compilador de LGE.

2. Mostrar las estructuras de LGE.

3. Mezclar LGE con ensamblador.

Esta sección no es imprescindible para lograr el objetivo del curso en general pero es un buen complemento para crear una mejor perspectiva de la relación entre los lenguajes de programación y el hardware de una computadora.

8.1.5. Evaluación

En esta parte se hace una evaluación a todos los alumnos que participaron en el curso.

Lo ideal sería que los alumnos desarrollaran algún proyecto donde usaran todas o la mayoría de las capacidades del Game Boy, pero por el tiempo que esto implica se deja fuera de la evaluación.

Por un lado se realiza un pequeño examen sobre conceptos generales de arquitectura de computadoras. Este examen contenie conceptos generalizados, no propios del Game Boy para que se muestre la extrapolación de los temas.

En segundo lugar se aplica una encuesta sobre las herramientas desarrolladas en este trabajo para tener una medida subjetiva de su eficacia.

Finalmente se comenta el trabajo con los profesores encargados de sus respectivos grupos.

En el anexo de este trabajo se pueden encontrar tanto el examen aplicado como la encuesta que se realizó.

8.2. Resultados

8.2.1. Preparación del curso

Para lograr realizar el experimento mencionado se solicitó apoyo tanto al programa de Matemáticas Aplicadas y Computación como a los maestros que imparten la materia de Arquitectura de Computadoras. Afortunadamente se mostraron muy interesados en el proyecto, por lo que decidieron apoyarlo.

Se acordó impartir el curso en 4 grupos de la materia de Arquitectura de Computadoras de la carrera de Matemáticas Aplicadas y Computación. Los grupos fueron los siguientes:

- (A) Grupo 2652 con el profesor Mauricio Rico Castro.
- (B) Grupo 2651 con el profesor Pablo H. González Videgaray.
- (C) Grupo 2601 con el profesor Pablo H. González Videgaray.
- (D) Grupo 2602 con la profesora Nanette De Hoyos Esparza.

De aquí en adelante se hará referencia a los grupos por su letra asignada.

El tiempo acordado para dar el curso fue de 6 horas (3 clases) con excepción de los grupos B y C en donde sólo se contó con 4 horas (2 clases). Dada esta limitación de tiempo al igual que otras limitaciones en cuanto a recursos como laboratorios, se decidió omitir la sección del lenguaje LGE y sólo enfocar el curso en las secciones de ensamblador y el emulador.

8.2.2. Desarrollo del curso

El curso se desarrolló en los 4 grupos sin problemas importantes. Afortunadamente los grupos B y C fueron los últimos en recibir el curso, por lo que la experiencia adquirida en los otros 2 grupos ayudó significativamente en cubrir el material en menos tiempo.

Algunos de los inconvenientes que surgieron fueron los siguientes:

- Los alumnos no llegaban puntualmente a la clase, en varias ocasiones se empezó hasta 30 minutos después de la hora indicada.
- No todas las computadoras de los laboratorios tenían el software necesario (Python) para ejecutar los programas.
- En un par de ocasiones no se contó con conexión a Internet para descargar las herramientas.

Estos inconvenientes fueron resueltos en su momento gracias al apoyo de los profesores que imparten la materia y que estuvieron presentes durante el curso. El curso se dividió en 2 partes: una sección teórica y otra práctica. En la sección teórica se explicó la arquitectura del Game Boy en general. En la sección práctica se realizaron algunos programas en lenguaje ensamblador que pudieron ejecutar y visualizar su salida.

Para la sección práctica se elaboró un paquete donde se incluían todas las herramientas necesarias para el curso. Este paquete podía ser obtenido de Internet donde aun puede ser descargado. El paquete consiste en las siguientes herramientas:

- Ensamblador gbz80asm.
- Emulador BioGB4.
- Emulador y depurador no$gmb.
- Documentación del lenguaje ensamblador.

Finalmente, al terminar el curso, se aplicó la evaluación y la encuesta a cada uno de los grupos. La evaluación consistió en 2 partes: una teórica con 12 reactivos de opción múltiple y una práctica con 2 preguntas abiertas. Sin embargo, a los grupos B y C también se les aplicó la parte teórica de la evaluación antes de comenzar el curso. Se dio un total de 30 minutos para resolver la evaluación la cual se puede encontrar en los anexos de este trabajo. También hay que señalar que la segunda evaluación de los grupos B y C fue realizada en linea. En todos los grupos la encuesta fue realizada en línea. En los anexos se puede encontrar la encuesta aplicada.

8.2.3. Resultados de las evaluaciones

Para poder analizar los resultados de las evaluaciones se tomo en cuenta lo siguiente: La calificación final de cada alumno no se incluyó en el análisis ya que fue ponderada de manera favorable para los alumnos. Se dio un valor de 90 % para la parte teórica y 40 % para la parte práctica. De ésta manera los alumnos podían llegar a obtener hasta 130 de calificación lo cual se tomaba como 100.

Lo que se usó como datos para el análisis fue el número de aciertos. Es importante mencionar que la parte práctica constaba de 9 aciertos de los cuales 4 fueron de la primer pregunta y 5 de la segunda. De la misma manera, la parte teórica constaba de 12 aciertos donde a cada reactivo le correspondía un acierto.

Uno de los inconvenientes que se dio en los grupos B y C, donde se aplicó la parte teórica de la evaluación, fue que hubo alumnos que sólo hicieron una de las evaluaciones y no ambas. Para este caso se dan las siguientes medidas:

- El promedio general de los que presentaron la primera evaluación.
- El promedio general de los que presentaron la segunda evaluación.
- Los dos promedios de los que presentaron ambas evaluaciones.

En los grupos A y D sólo se aplicó una evaluación por lo que ahí no hubo ningún inconveniente y las únicas medidas son los promedios generales. En la figura 8.1 se puede ver la gráfica de los grupos B y C donde se comparan los aciertos de las 2 evaluaciones.

En esta gráfica se expone el promedio de aciertos de los alumnos que presentaron ambas evaluaciones. Como se puede observar hay un incremento de aproximadamente 2 aciertos al realizar la segunda evaluación. Ahora se analiza la siguiente gráfica en la figura 8.2:

En esta gráfica de barras se observan los promedios generales de cada grupo en cada evaluación. En el caso del grupo B y C se tomaron en cuenta todos los alumnos que hicieron la primera evaluación y todos lo que hicieron la segunda, no sólo ambas. De esta manera se pueden comparar los resultados con los grupos A y D. Cabe señalar que en la segunda evaluación hay una fuerte consistencia en los resultados, es decir, los 4 grupos obtuvieron un promedio muy similar en aciertos.

Los resultados expuestos dan una idea del efecto que tuvo el curso en el número de aciertos de los alumnos. Con estos resultados podemos tener una perspectiva de lo que se busca en nuestra hipótesis.

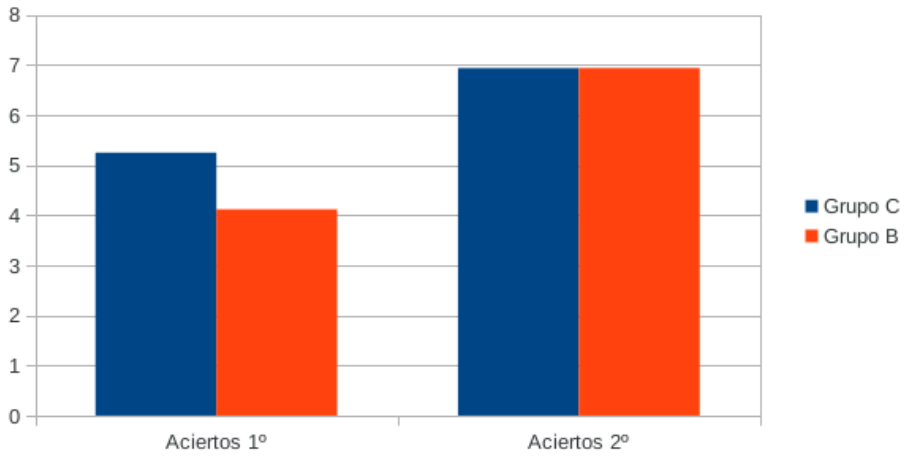


Figura 8.1: Promedio de aciertos de los que presentaron ambas evaluaciones. Grupo B y C

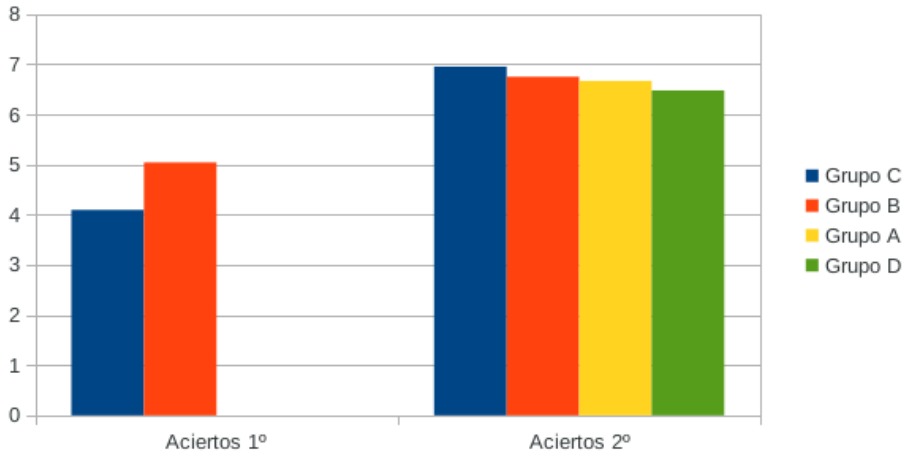


Figura 8.2: Promedio de aciertos en cada grupo.

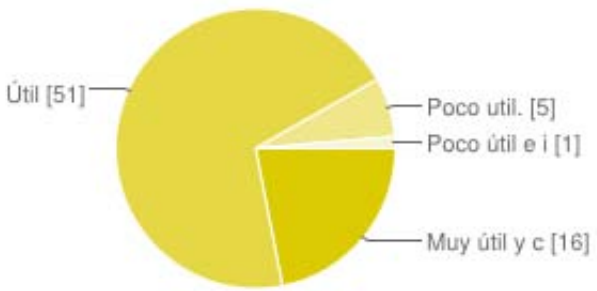


Figura 8.3: ¿Qué tan útil es el ensamblador?

8.3. Comentarios

Otra parte que se consideró importante para poder interpretar los resultados dados anteriormente fue la opinión tanto de los alumnos como de los profesores de la materia. En el caso de los alumnos se aplicó una encuesta, la cual se puede encontrar en los anexos de este trabajo.

8.3.1. Sobre las encuestas

La encuesta consistió en un cuestionario con 11 preguntas de opción múltiple y un apartado para comentarios generales. Todas las encuestas se realizaron en linea. En general se pidió a los alumnos evaluar tanto el curso como las herramientas usadas (ensamblador y emulador). A continuación se exponen las estadísticas mas relevantes de la encuesta (para ver los porcentajes exactos vea el anexo):

En la Figura 8.3 se preguntó qué tan útil consideraban el ensamblador que se usó en el curso. En la Figura 8.4 se preguntó que tan útil consideraban el emulador que se usó en el curso. En la Figura 8.5 se preguntó si consideraban que el uso de las herramientas del curso hayan mejorado su comprensión de la arquitectura de computadoras. En la Figura 8.6 se preguntó como creían que sería el conocimiento adquirido en caso de no haber usado las herramientas.

En los anexos de este trabajo se pueden encontrar las estadísticas de las demás preguntas de la encuesta. A continuación se pone una lista con el resumen de los comentarios más relevantes por su incidencia:

- Una gran mayoría de los encuestados comentó que el curso fue muy corto para la cantidad de información que se quería enseñar.

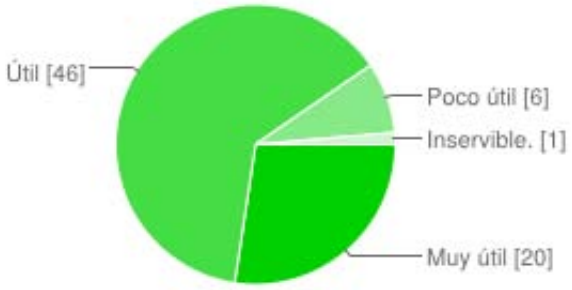


Figura 8.4: ¿Qué tan útil es el emulador?

Figura 8.5: ¿Las herramientas ayudaron a comprender la AC?

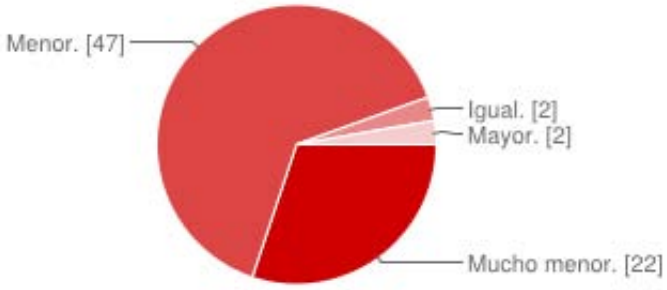


Figura 8.6: Conocimiento en caso de no usar las herramientas.

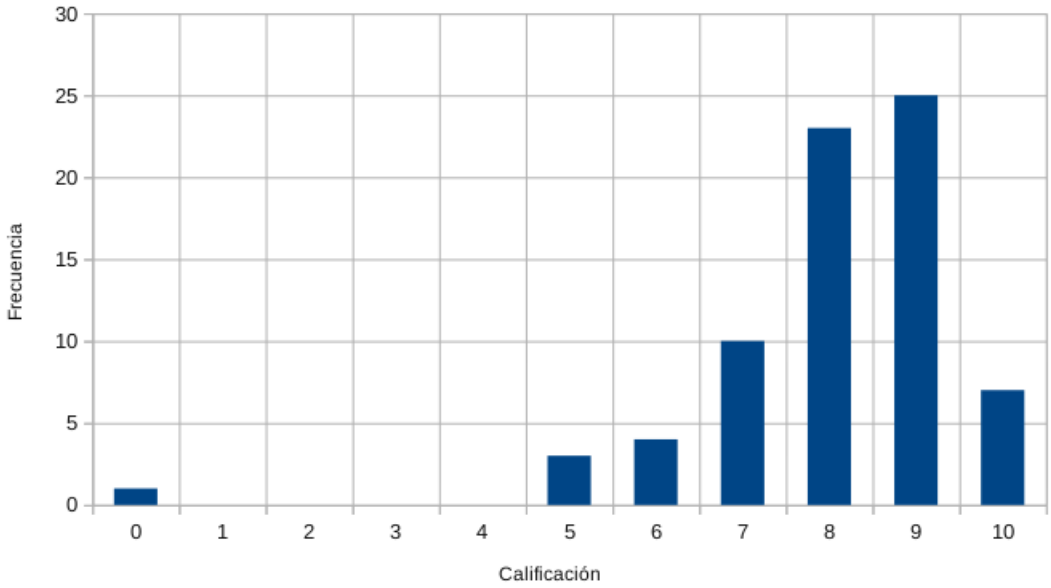


Figura 8.7: Calificación del curso.

- Otro comentario común fue la falta de prácticas durante el curso.
- Hubo algunos comentarios sobre las fallas técnicas en los laboratorios.
- También recibí comentarios muy positivos sobre el curso. Algunos hicieron la sugerencia de que se volviera a dar pero con mas tiempo.
- De la misma manera hubo algunos comentarios negativos donde la queja principal fue la misma que los primeros 2 puntos.

Se pueden encontrar las respuestas inéditas en la sección de los anexos.

Finalmente tanto en la figura 8.7 como en la figura 8.8 se exponen las distribuciones donde se pidió a los encuestados calificar tanto el curso como las herramientas.

La gran mayoría calificó tanto el curso como las herramientas de manera positiva dando calificaciones entre 8 y 9. Con esto se puede concluir que hubo un buen impacto en los alumnos que tomaron el curso.

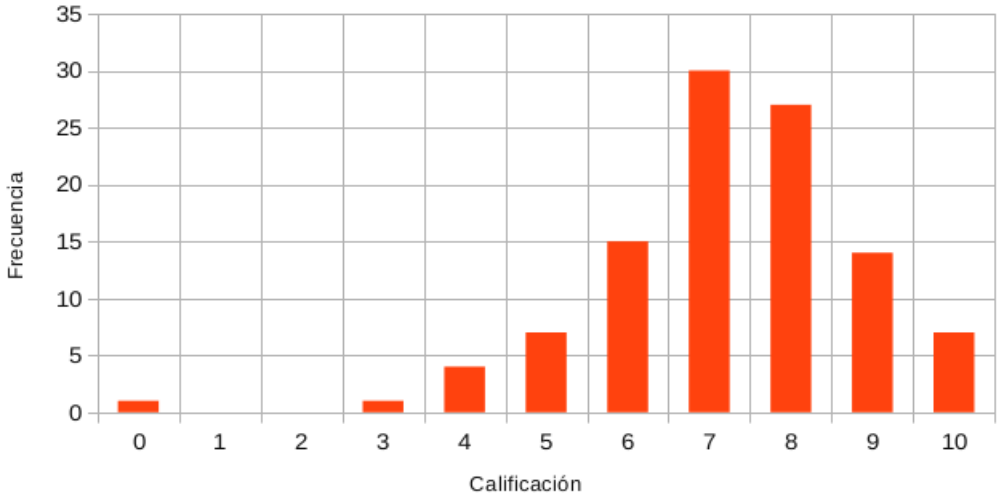


Figura 8.8: Calificación de las herramientas.

8.3.2. Comentarios de los profesores de la materia

Como se mencionó anteriormente, los profesores de la materia de Arquitectura de Computadoras estuvieron presentes durante el curso y al finalizar también hicieron varios comentarios que son relevantes para este trabajo.

El profesor Mauricio Rico al final comentó que le agradó bastante el curso. Sugirió que las herramientas estuvieran disponibles tanto para los alumnos como para los maestros. Incluso comentó que le hubiera gustado que el curso fuera un poco más extenso.

El profesor Pablo H. González también comentó sobre su agrado del curso. Al final de la primera sesión hizo notar un error conceptual que se tuvo durante el curso el cual se hizo la corrección en la segunda sesión. Al final del curso hizo una comparación entre el Game Boy y la máquina virtual que propone Andrew S. Tanenbaum en su libro [Tan12] que es la que normalmente utiliza en su curso de la materia. Se habló de las ventajas de una y otra, sin embargo, después de exponer las ventajas que ya se han mencionado en este trabajo, se concluyó que la elección del Game Boy como máquina simple había sido una buena decisión.

En el caso de la profesora Nanette de Hoyos, hay que mencionar que en un principio no se iba a dar el curso en su grupo, sin embargo, ella se mostró bastante interesada en el material del curso y ofreció que se diera

en su grupo. Al final, mencionó su interés por las herramientas y sobretodo por el curso. Incluso sugirió grabar la clase en video para poder mostrar el material nuevamente en otros semestres.

En general, los comentarios de los profesores fueron muy positivos, y se aprovecha este espacio para agradecerles por la oportunidad de dar el curso en el tiempo y espacio de su materia.

Capítulo 9

Conclusiones

Se comenzó este trabajo hablando sobre cómo a través de la historia los especialistas en las diferentes áreas de la computación han creado diferentes tipos de abstracciones o capas para poder ocultar algún nivel de complejidad. Ciertamente estas abstracciones han ayudado enormemente a incrementar la complejidad tanto del software como del hardware. Sin embargo, en esta, tal vez excesiva, sucesión de capas que hay entre el software y hardware se ha dejado de prestar atención en lo que hay en los niveles mas bajos de abstracción. Como consecuencia muchas veces se ha dejado de sacar provecho a todas las posibilidades y características del hardware y nos enfocamos sólo con lo que nos ofrecen las capas superiores. Es por esta razón que en este trabajo se hizo una propuesta de herramientas diseñadas específicamente para reintroducir tanto a estudiantes como a gente interesada en la arquitectura de computadoras, a los niveles mas bajos de abstracción y así obtener una mejor comprensión del funcionamiento interno de las computadoras en general.

En primer instancia se seleccionó una máquina que fuera sencilla y con una arquitectura interna elegante. La máquina elegida fue el Game Boy.

Una vez elegida la máquina, se hizo un análisis sobre las herramientas elegidas para poder cumplir con el objetivo. Las herramientas elegidas se basaron en un subconjunto de lo que se conoce como software del sistema. Este subconjunto lo fue elegido con base tanto en las capacidades del Game Boy así como en el objetivo de este trabajo. Las herramientas seleccionadas fueron: un ensamblador, un compilador de un nuevo lenguaje estructurado y un emulador.

En cada capítulo se describió una metodología para la implementación de los algoritmos de estas herramientas. En el caso de algoritmos complejos

como los de la implementación del compilador se mencionaron múltiples referencias tanto a otros textos como al código del compilador programado para este trabajo.

Finalmente se mencionó un segundo conjunto de herramientas complementarias, entre ellas, el kit de programación y un depurador. Estas herramientas sirven como un excelente complemento a las desarrolladas en el trabajo.

Es importante mencionar también el lenguaje genérico estructurado, ya que su diseño se basó en el objetivo de este trabajo. A pesar de que existen otros lenguajes similares o de mayor potencia, se usaron metodologías de diseño de gramáticas en beneficio de una amplia comprensión de las herramientas y por lo tanto de la arquitectura de computadoras.

Como se podrá ver, muchas de las decisiones sobre el diseño de las herramientas no se basaron en la eficiencia ni en producir mejor código compilado. Mas bien, se hizo un gran énfasis en la legibilidad del código y en la claridad de las metodologías de implementación. De esta manera no se pierde el enfoque hacia la arquitectura de computadoras.

El experimento cualitativo mostró resultados interesantes, donde hubo un incremento en el número medio de aciertos que obtuvieron las alumnos después de tomar el curso. Dentro de las encuestas y la opinión de los maestros a cargo de los grupos hubo un buen recibimiento tanto del curso como de las herramientas.

En conclusión se puede decir que en este trabajo, con base a los resultados obtenidos, quedó demostrado como un grupo de herramientas junto con el curso propuesto pueden ayudar a estudiantes a mejorar la comprensión sobre la arquitectura de computadoras, su funcionamiento y su programación. Queda mucho por hacer; desde pulir y mejorar las herramientas, explorar las diferentes posibilidades dentro del curso propuesto, hasta realizar experimentos que arrojen datos mas objetivos. El tema aun queda abierto para su investigación y profundización.

Anexos

Gramática LGE

A continuación se puede encontrar la gramática LGE completa. Las líneas que comienzan con el símbolo % son comentarios que nombran conjuntos de reglas.

```
<Goal> ::= <programa>

<programa> ::= <cabecera> <def-globales> ".main"
    <cuerpo-programa> ".end"
<programa> ::= <cabecera> ".main" <cuerpo-programa>
    ".end"

% Definicion de la cabecera

<cabecera> ::= <cabecera> "opcion_asm"
<cabecera> ::= <cabecera> "opcion_asm" "valor"
<cabecera> ::= <cabecera> "opcion_asm" "identificador"
<cabecera> ::= "opcion_asm"
<cabecera> ::= "opcion_asm" "valor"
<cabecera> ::= "opcion_asm" "identificador"

% Definiciones globales

<def-globales> ::= <def-variables> <def-funciones>
<def-globales> ::= <def-funciones>
<def-globales> ::= <def-variables>

% Funciones
```

```
<def-funciones> ::= <funcion> <def-funciones>
<def-funciones> ::= <rutina> <def-funciones>
<def-funciones> ::= <funcion>
<def-funciones> ::= <rutina>


<rutina> ::= "routine" "identificador" <parametros>
   "{" <cuerpo-rutina> "}"
<rutina> ::= "routine" "identificador" "{"
   <cuerpo-rutina> "}"
<funcion> ::= "function" <tipo> "identificador"
   <parametros> "{" <cuerpo-funcion> "}"
<funcion> ::= "function" <tipo> "identificador" "{"
   <cuerpo-funcion> "}"

<parametros> ::= <tipo> "identificador" ","
   <parametros>
<parametros> ::= <tipo> "identificador"

<tipo> ::= "byte"
<tipo> ::= "word"

<cuerpo-rutina> ::= <def-variables> <sentencias>
<cuerpo-rutina> ::= <sentencias>
<cuerpo-funcion> ::= <def-variables> <sentencias>
   "return" <expresion>
<cuerpo-funcion> ::= <sentencias> "return" <expresion>
<cuerpo-funcion> ::= "return" <expresion>


% Definicion de variables globales

<def-variables> ::= <tipo> "identificador" "="
   <expresion> <def-variables>
<def-variables> ::= <tipo> "identificador" "="
   <expresion>
<def-variables> ::= <tipo> "identificador"
   <def-variables>
<def-variables> ::= <tipo> "identificador"
```

```
% Cuerpo del programa

<cuerpo-programa> ::= <def-variables> <sentencias>
<cuerpo-programa> ::= <sentencias>

<sentencias> ::= <sentencia-asignacion> <sentencias>
<sentencias> ::= <sentencia-llamada> <sentencias>
<sentencias> ::= <sentencia-seleccion> <sentencias>
<sentencias> ::= <sentencia-iteracion> <sentencias>
<sentencias> ::= <sentencia-incdec> <sentencias>
<sentencias> ::= <sentencia-asm> <sentencias>
<sentencias> ::= <sentencia-asignacion>
<sentencias> ::= <sentencia-llamada>
<sentencias> ::= <sentencia-seleccion>
<sentencias> ::= <sentencia-iteracion>
<sentencias> ::= <sentencia-incdec>
<sentencias> ::= <sentencia-asm>
<sentencias> ::= "pass"

<sentencia-asignacion> ::= "identificador" "="
   <expresion>
<sentencia-llamada> ::= "identificador" "("
   <paso-parametros> ")"
<sentencia-llamada> ::= "identificador" "(" ")"

<paso-parametros> ::= <expresion> ","
   <paso-parametros>
<paso-parametros> ::= <expresion>

<sentencia-seleccion> ::= "if" <expresion> "{"
   <sentencias> "}" "else" "{" <sentencias> "}"
<sentencia-seleccion> ::= "if" <expresion> "{"
   <sentencias> "}"

<sentencia-iteracion> ::= "while" <expresion> "{"
   <sentencias> "}"

<sentencia-incdec> ::= <expresion> "++"
<sentencia-incdec> ::= <expresion> "--"
```

```
<sentencia-asm> ::= ".asm" "{" "ensamblador" "}"

% Expresiones

<expresion> ::= <expresion-logica-or>

<expresion-logica-or> ::= <expresion-logica-and> "or"
   <expresion-logica-or>
<expresion-logica-or> ::= <expresion-logica-and>

<expresion-logica-and> ::= <expresion-or> "and"
   <expresion-logica-and>
<expresion-logica-and> ::= <expresion-or>

<expresion-or> ::= <expresion-xor> "|" <expresion-or>
<expresion-or> ::= <expresion-xor>

<expresion-xor> ::= <expresion-and> "^"
   <expresion-xor>
<expresion-xor> ::= <expresion-and>

<expresion-and> ::= <expresion-igualdad> "&"
   <expresion-and>
<expresion-and> ::= <expresion-igualdad>

<expresion-igualdad> ::= <expresion-relacional> "=="
   <expresion-igualdad>
<expresion-igualdad> ::= <expresion-relacional> "!="
   <expresion-igualdad>
<expresion-igualdad> ::= <expresion-relacional>

<expresion-relacional> ::= <expresion-corrimiento>
   "<" <expresion-relacional>
<expresion-relacional> ::= <expresion-corrimiento>
   ">" <expresion-relacional>
<expresion-relacional> ::= <expresion-corrimiento>
   "<=" <expresion-relacional>
<expresion-relacional> ::= <expresion-corrimiento>
   ">=" <expresion-relacional>
```

```
<expresion-relacional> ::= <expresion-corrimiento>

<expresion-corrimiento> ::= <expresion-aditiva> ">>"
   <expresion-corrimiento>
<expresion-corrimiento> ::= <expresion-aditiva> "<<"
   <expresion-corrimiento>
<expresion-corrimiento> ::= <expresion-aditiva>

<expresion-aditiva> ::= <expresion-multiplicativa>
   "+" <expresion-aditiva>
<expresion-aditiva> ::= <expresion-multiplicativa>
   "-" <expresion-aditiva>
<expresion-aditiva> ::= <expresion-multiplicativa>

<expresion-multiplicativa> ::= <expresion-unaria> "*"
   <expresion-multiplicativa>
<expresion-multiplicativa> ::= <expresion-unaria> "/"
   <expresion-multiplicativa>
<expresion-multiplicativa> ::= <expresion-unaria> "%"
   <expresion-multiplicativa>
<expresion-multiplicativa> ::= <expresion-unaria>

<expresion-unaria> ::= <expresion-primaria> "["
   <expresion> "]"
<expresion-unaria> ::= <expresion-primaria>
<expresion-unaria> ::= "not" <expresion-primaria>

<expresion-primaria> ::= "identificador"
<expresion-primaria> ::= "valor"
<expresion-primaria> ::= "(" <expresion> ")"
```

Examen del Curso

Marque el inciso que considere correcto de acuerdo al curso.

1. Tipo de BUS donde se usan pistas en paralelo dentro de un circuito donde cada pista tiene una función fija.

   a) BUS de datos.

   b) BUS de direcciones.

c) BUS en serie.

d) BUS en paralelo

2. El mapa de memoria es la abstracción de los siguientes componentes:

   a) CPU, BUS de datos y BUS de direcciones.

   b) CPU, código ensamblador y memoria RAM.

   c) Interrupciones, memoria RAM y memoria ROM.

   d) CPU, memoria RAM y sistema de entrada y salida.

3. El mapa de memoria direcciona los siguientes componentes.

   a) CPU, memoria RAM, memoria ROM.

   b) Memoria RAM, memoria ROM y sistema de entrada y salida.

   c) CPU, BUS de datos y BUS de direcciones.

   d) CPU, BUS en serie y BUS en paralelo.

4. Un opcode es:

   a) El conjunto de parámetros de una instrucción en ensamblador.

   b) El nombre mnemotécnico de una instrucción en código máquina.

   c) Una instrucción en código máquina con una representación en ensamblador.

   d) Una interrupción.

5. Es una localidad de memoria de alta velocidad localizada dentro del CPU.

   a) Registro.

   b) Interrupción.

   c) Celda de memoria RAM.

   d) Celda de memoria ROM.

6. Sucede cuando un dispositivo externo solicita tiempo del CPU para procesar datos del sistema de entrada y salida.

   a) Registro.

   b) Proceso de arranque.

   c) Desborde del temporizador.

d) Interrupción

7. Sección de memoria donde se almacenan las direcciones de retorno al ejecutar instrucciones call o al haber un interrupción.

   a) Puertos de entrada y salida.

   b) Banco 0 de ROM.

   c) Pila en RAM

   d) Memoria de atributos de sprites.

8. Bits dentro de uno o más registros específicos que sirven para controlar la ejecución de un programa y que interactúan directamente con las instrucciones en código máquina.

   a) Registros de control.

   b) Bits de control.

   c) Banderas de control.

   d) Banderas de registros.

9. ¿Dónde se puede acceder a los dispositivos externos en nuestro programa?

   a) A través de la memoria ROM.

   b) A través de de registros externos.

   c) A través de los puertos vacios del mapa de memoria.

   d) A través de los puertos de entrada y salida del mapa de memoria.

10. Proceso en el cual el código enamblador es traducido a código máquina.

    a) Ensamblar.

    b) Compilar.

    c) Codificar.

    d) Decodificar.

11. Proceso en el cual código en un lenguaje de programación de alto nivel es traducido a código ensamblador o código máquina.

    a) Ensamblar.

    b) Compilar.

c) Codificar.

d) Decodificar.

12. Área del mapa de memoria donde se encuentra nuestro programa.

    a) Memoria RAM externa.

    b) Memoria VRAM.

    c) Memoria RAM interna.

    d) Memoria ROM.

13. Elabore un programa en ensamblador que haga la división entera de 2 números contenidos en los registros a y b.

14. Elabore un programa en ensamblador que utilice la interrupción v-blank para pintar un cuadrado en la parte superior izquierda de la pantalla.

Respuestas

1 - d, 2 - a, 3 - b, 4 - c, 5 - a, 6 - d,
7 - c, 8 - c, 9 - d, 10 - a, 11 - b, 12 - d

Encuesta

Encuesta sobre el curso de ensamblador.

Esta encuesta servirá para evaluar el curso de ensamlador que se dió en los grupos de arquitectura de computadoras. Los resultados se darán a conocer una vez que todos los grupos hayan realizado la encuesta, También se dará a conocer el análisis de los resultados dentro del trabajo de tesis para el cual se hizo este curso.

1. Considera que el curso de ensamblador que recibió fue:

    a) Excelente

    b) Bueno

    c) Regular

    d) Malo

2. Las explicaciones de los temas fueron:

    a) Muy claras

b) Claras

c) Poco Claras

d) Confusas

3. El ensamblador que se usó en el curso lo considera:

   a) Muy útil y completo

   b) Útil

   c) Poco útil

   d) Poco útil e incompleto

4. El emulador que se usó en el curso, lo considera:

   a) Muy útil

   b) Útil

   c) Poco útil

   d) Inservible

5. Considera que el uso de las herramientas del curso hayan mejorado su comprensión de la arquitectura de computadoras.

   a) Si ayudaron

   b) No ayudaron

6. Cree que las herramientas mostradas en el curso le vayan a servir en un futuro.

   a) Si

   b) No

   c) Tal vez

7. Cree que los conocimientos adquiridos en el curso le vayan a ser útiles eventualmente?:

   a) Si

   b) No

8. Sin la parte práctica del curso, el conocimiento adquirido sería:

   a) Mucho menor

b) Menor

c) Igual

d) Mayor

9. La documentación del paquete del curso la considera:

a) Muy adecuada

b) Adecuada

c) Poco adecuada

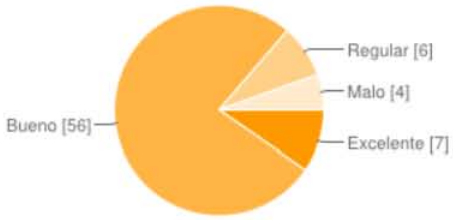
d) Inadecuada

10. Califique el curso (del 0 al 10) según su criterio ___.

11. Califique las herramientas que usó en el curso ___.

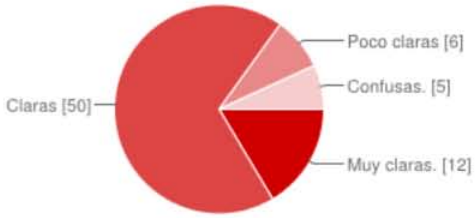
Respuestas

Considera que el curso de ensamblador que recibió fue:



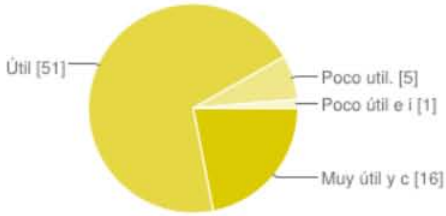
Excelente	**7**	10%
Bueno	**56**	77%
Regular	**6**	8%
Malo	**4**	5%

Las explicaciones de los temas fueron:



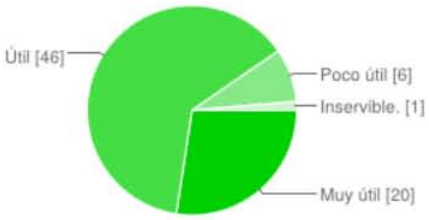
Muy claras.	**12**	16%
Claras	**50**	68%
Poco claras	**6**	8%
Confusas.	**5**	7%

El ensamblador que se usó en el curso lo considera:



Muy útil y completo	**16**	22%
Útil	**51**	70%
Poco util.	**5**	7%
Poco útil e incompleto.	**1**	1%

El emulador que se usó en el curso, lo considera:

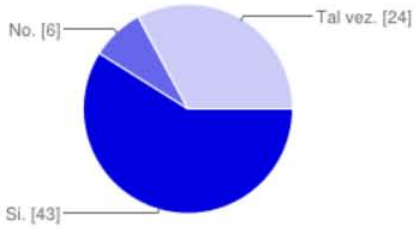


Muy útil	**20**	27%
Útil	**46**	63%
Poco útil	**6**	8%
Inservible.	**1**	1%

Considera que el uso de las herramientas del curso hayan mejorado su comprensión de la arquitectu

Si ayudaron.	**64**	88%
No ayudaron.	**9**	12%

Cree que las herramientas mostradas en el curso le vayan a servir en un futuro.

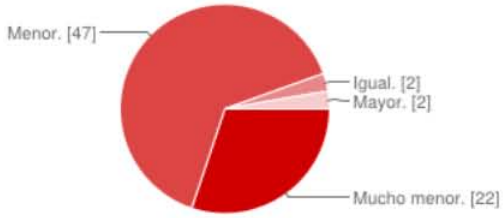


Si.	**43**	59%
No.	**6**	8%
Tal vez.	**24**	33%

Cree que los conocimientos adquiridos en el curso le vayan a ser útiles eventualmente?:

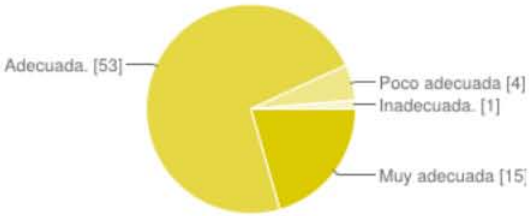
Si.	**57**	78%
No.	**16**	22%

Sin la parte práctica del curso, el conocimiento adquirido sería:



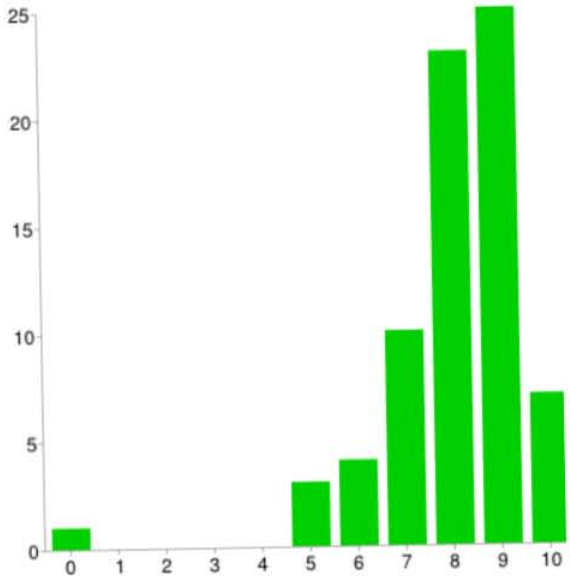
Mucho menor.	**22**	30%
Menor.	**47**	64%
Igual.	**2**	3%
Mayor.	**2**	3%

La documentación del paquete del curso la considera:



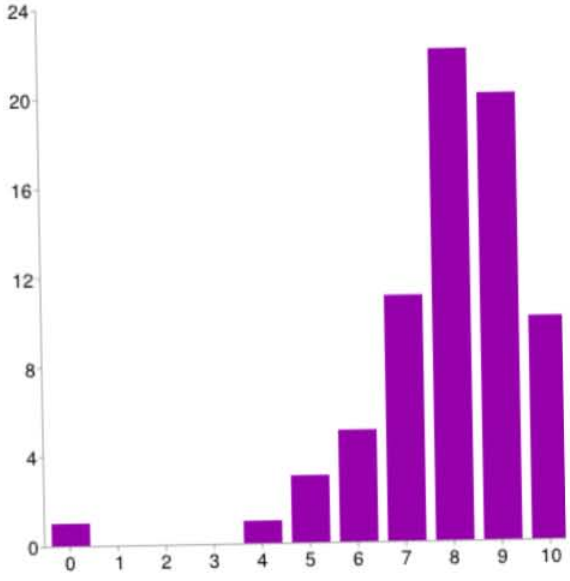
Muy adecuada.	**15**	21%
Adecuada.	**53**	73%
Poco adecuada.	**4**	5%
Inadecuada.	**1**	1%

Califique el curso (del 0 al 10) según su criterio.



0	**1**	1%
1	**0**	0%
2	**0**	0%
3	**0**	0%
4	**0**	0%
5	**3**	4%
6	**4**	5%
7	**10**	14%
8	**23**	32%
9	**25**	34%
10	**7**	10%

Califique las herramientas que usó en el curso.



0	**1**	1%
1	**0**	0%
2	**0**	0%
3	**0**	0%
4	**1**	1%
5	**3**	4%
6	**5**	7%
7	**11**	15%
8	**22**	30%
9	**20**	27%
10	**10**	14%

Comentarios

En esta sección pongo de manera textual los diferentes comentarios que expusieron los participantes del curso.

- Demasiado corto para toda la información que se trató de dar
- Fue un curso muy accesible y util
- Siento que falto tiempo para aclarar posibles dudas pero en general el curso estuvo bien planeado.
- Seria mejor si hubiera mas horas del curso.
- Me pareció que tiene muchos conocimientos y sabe de lo que habla, mi principal problema fue que las clases las sentí muy apresuradas y a las carreras dejando poco tiempo a la comprensión de los temas. Si tuviera oportunidad de tomar un curso con más tiempo, si duda alguna lo tomaría.
- Falto más información entendible y realización de prácticas
- el tiempo fue limitado, y la fallas técnicas no ayudan en nada
- buen curso
- el curso esta bien, solo que el problema único fue que en las maquinas no había python instalado.
- Me pareció muy bueno el curso, aunque a mi parecer creo que le falto más tiempo,ya que los temas los daba muy rápido
- En general, me parece un buen curso que contiene información interesante que sería mejor aplicarla en más prácticas. Lamentablemente no se pudieron hacer más prácticas, por lo que no pude completar y comprender el tema al 100 %. Es necesario tener bastante tiempo para poder aplicar los temas, jugar programando y complementar la teoría.
- Me hubiese gustado que el curso durara mas tiempo ya que se me hizo muy interesante.
- Pues es muy corto el curso eso afecta a que no se vea a detalle todas sus herramientas y pues si enseñas bien, lo que te falto es tiempo.
- Falto tiempo para hacer mas practicas.

- Mucha informacion y poco tiempo para diluirla pero bien explicado y con ejercicios simples
- Fue un curso muy rapido para todo el gran contenido , me hubiera gustado que fuera por mas dias, aun que aprendi mucho , en tan poco tiempo , fuiste muy claro, gracias y espero se abra este curso aparte para poder completarlo y no quedarnos con lo basico
- :)
- Pues para el poco tiempo que duró el curso aprendimos bastante
- Creo que el unico problema del curso es la duración del mismo, pero como curso introductorio es muy bueno dado que da una idea en general sobre lenguaje ensamblador y algunas caracteristicas de la arquitectura del gameboy.
- La parte práctica fue muy rápida y por los problemas con el python varios compañeros nos atrasamos y nos fue imposible llegar a la completa comprensión
- El curso me parecio bueno e interesante, el unico problema es que el laboratorio no contaba con el software necesario para elaborar la parte practica.
- El curso me pareció demasiado bueno, ya que por lo menos para mi fue algo nuevo, el único defecto fue que los equipos de cómputo no se prestaron para lograr adquirir al 100 % los conocimientos del curso!
- Fue un curso muy sensillo.
- estuvo completo pero breve , lo que no se puede tener un seguimiento , ademas de que no podia atender las dudas de cada alumno , pero en si explico los temas de manera clara , aunque un poco apresurado
- Aprendí más en el curso que en mi clase de Arquitectura de Computadoras...
- Pese a que no me interesa mucho el tema, me parecio interesante el curso. Pero seria bueno que se eliminara la clase teórica y se diera simultaneamente con la práctica.
- percatarse que este bien instalado y un poco mas de practica

- Estuvo muy interesante y el ponente que lo llevó a cabo sabe bastante acerca del curso, lo único malo fue el poco tiempo del curso.
- Falto una muestra de tu propio trabajo que estas haciendo o que hiciste.
- Me gustó el curso porque fue dinámico y creo que aprendí y comprendí algunas cosas que se me dificultaban
- Un poco más de practicas.
- Buena exposición... :)
- Antes de dar las clases deberian de tener todo listo, o pedirnos que traigamos nuestras laptops y con los archivos q necesitabamos para poder realizar los ejercicios
- Me hubiera gustado que se realizaran más prácticas
- La explicación fue muy rápida, si se hubiera contado con más tiempo habriamos obtenido mejores conocimientos. ¡FELICIDADES!
- Creo que hizo falta más tiempo para la parte práctica.
- El curso fue muy rápido, me hubiese gustado que durara mas
- Hazlo menos aburrido
- Me hubiera gustado tomar el curso completo. Deberías abrir un taller de 1-3 :)
- muy bueno
- algunas maquinas no contaban con las herramientas adecuadas, no obstante se entendía con el proyector. se podría mejorar el entendimiento durante la practica con algún material de apoyo un triptico o una presentación para revisar con anterioridad y agilizar la practica.
- Deberían de darle más tiempo, no considero que haya sido suficiente sólo 3 clases.
- mas tiempo y con mas expliaciones
- Fue muy corto y no alcance a aterrizar todos los conceptos.

- EL curso fue bueno, pero se dío poco tiempo para lo extenso que puede ser
- muy bueno
- me hubiera gustado que no existieran los temblores y pues la verdad fue muy poco tiempo de la clase
- fue muy claro y buenos ejemplos, falto tiempo y problemas con las maquinas fuera de eso todo bien

Bibliografía

[Abe96] Peter Abel. *Lenguaje Ensamblador y Programación para PC IBM y Compatibles*. Pearson Education, tercera edición edition, 1996.

[AGF+99] Anthrox, GABY, Marat Fayzullin, Pscal Felber, Paul Robson, Martin Korth, kOOPa, and Bowser. *Game Boy CPU Manual*. Nintendo CO., 1999.

[ASU90] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compiladores. Principios, técnicas y herramientas.* Pearson. Addison Wesley Longman., 1990.

[Bec97] Leland L. Beck. *System Software. An Introduction to Systems Programming*. Addison Wesley Longman, 3rd edition edition, 1997.

[Bre01] Barry B. Brey. *Los Microprocesadores Intel. Arquitectura, programación e interfaz de los procesadores 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro u Pentium II*. Prentice Hall, 5º edición edition, 2001.

[CT12] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kauffman, second edition, 2012.

[dB01] Victor Moya del Barrio. *Study of the techniques for emulation programming*. Computer Science Engenieering – FIB UPC, 2001.

[Flo08] Thomas L. Floyd. *Digital Fundamentals*. Prentice Hall, 10 º edition, 2008.

[GBJL07] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen. *Diseño de Compiladores Modernos*. McGraw Hill, 2007.

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Teoría de autómatas lenguajes y computación*. Pearson-Addison Wesley, 2007.

[Hol90] Allen I. Holub. *Compiler Design in C*. Prentice Hall, 1990.

[HVZ02] Carl Hamacher, Zvonko Vranesic, and Safwat Zaky. *Computer Organization*. McGraw-Hill, fifth edition edition, 2002.

[McC04] Steve McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004.

[PH11] David A. Patterson and John L. Hennessy. *Estructura y Diseño de Computadores. La Interfaz Hardware / Software*. Editorial Reverté, 2011.

[Sch01] Herbert Schildt. *C. Manual de Referencia.* McGraw Hill, 4th edition, 2001.

[Tan12] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 6th edition, 2012.

[VM03] John Viega and Matt Messier. *Secure Programming Cookbook for C and C++*. O'Reilly, 2003.

[ZD04] Stefan Zerbst and Oliver Düvel. *3D Game Engine Programming*. Thomson Course Technology PTR, 2004.