Trabajo de inserción profesional

Qsim Web: Simulador de arquitectura Q para el browser

Alumnos

Pablo Pissi
pablopissi@gmail.com

Francisco Pérez Ramos
franperez21896@gmail.com

Director

Lic. Federico Martínez
federicoemartinez@gmail.com

Febrero de 2021

DEPARTAMENTO DE CIENCIA Y TECNOLOGÍA
TECNICATURA UNIVERSITARIA EN
PROGRAMACIÓN INFORMÁTICA

Universidad Nacional de Quilmes

Resumen

La arquitectura Q fue creada en la Universidad Nacional de Quilmes en la materia Organización de Computadoras, con el propósito de facilitar la enseñanza y el aprendizaje del funcionamiento conceptual de las computadoras, las partes que la componen y sus mecanismos. Esta es una arquitectura conceptual, es decir, no existe una máquina física que funcione sobre dicha arquitectura. Al ser Organización de Computadoras una materia inicial, los estudiantes encuentran una dificultad en el aprendizaje, marcado principalmente por la poca abstracción con la que cuentan estos lenguajes, sumando en nuestro caso, la imposibilidad de ejecutar sus programas. En este sentido, QSim, un trabajo de inserción profesional presentado en la misma universidad, aborda el problema brindando un entorno de ejecución para el lenguaje Q. Dicho proyecto fue encarado con una aplicación de escritorio escrita en Java que permitía ver los componentes de la arquitectura Q en funcionamiento. Algunos de los problemas que se encontraron a esta implementación fueron la baja performance de la herramienta y la dificultad que encuentran los estudiantes para configurar el entorno que requiere la aplicación para funcionar.

Es por eso que en este trabajo se busca brindar una solución a este problema: un simulador de la arquitectura Q que permita visualizar de manera didáctica el proceso de ejecución de un lenguaje de bajo nivel pero realizada en un entorno web, lo cual evita las complejidades accidentales encontradas y nombradas anteriormente y agrega portabilidad para diversos dispositivos como celulares o tablets. Además, en contexto de la pandemia por COVID-19, donde la educación en la Universidad tuvo un viraje a la educación virtual, es fundamental contar con una herramienta que pueda ser accedida mediante Internet. El trabajo brinda una librería escrita en javascript que implementa el lenguaje Q y una interfaz visual que utiliza esa librería para mostrar de manera didáctica el proceso de ejecución. Esta separación permite realizar cambios en la interfaz de usuario sin afectar a la lógica del lenguaje.

Abstract

The Q architecture was created at the Universidad Nacional de Quilmes in the context of the course of study Computer Organization, with the purpose of helping to teach and learn the conceptual operation of the computers, its parts and mechanisms. It's a conceptual architecture, meaning that there is not a physical computer that works with it. Students often face a learning difficulty caused mainly by the lack of abstraction that these languages have, adding in our case the fact that Computer Organization is an initial subject and the impossibility of executing their programs. Given this, QSim, a TIP presented at the same university, addresses the problem by providing an execution environment for the Q language. The project was developed as a Java desktop application, which made it resource hungry and hard to set up.

Therefore this project tries to provide a solution to this problem: a tool that allows to didactically visualize the process of executing a low-level language while running on a web environment, which avoids the accidental complexities found in QSim and that also adds portability for various devices such as cell phones or tablets. Moreover, in the context of the COVID-19 pandemic, and universities embracing virtual learning, an online tool becomes even more significant. This project provides a library written in javascript that implements the Q language and a visual interface that uses said library to display the execution process in a more didactic way. This separation allows changes to be made to the user interface without altering the language's logic.

Índice

1 Introducción **2**

2 Arquitectura *Q* **5**

2.1 Características generales de la arquitectura **5**

2.2 Capas de la arquitectura **6**

3 Qsim web **11**

3.1 Interfaz de usuario **12**

3.1.1 Tecnologías utilizadas 21

3.1.2 Accesibilidad 22

3.2 Librería **24**

3.2.1 Tecnologías usadas 26

3.2.2 `Parser` 26

3.2.3 `Translator` 28

3.2.4 Computer 32

4 Devoluciones de estudiantes **39**

5 Conclusiones **44**

5.1 Trabajos futuros **44**

6 Anexos **46**

6.1 Anexo: Detalles de la arquitectura *Q* **46**

6.2 Anexo: Detalles de implementación **49**

6.2.1 Parser 49

6.2.2 Translator 52

6.2.3 Ensamblado 56

6.3 Anexo: Implementación de una interfaz grafica usando QLib **57**

6.3.1 Utilizando un parser distinto 60

Agradecimientos

- A Federico Martínez por sus grandes aportes conceptuales, por su buena onda y predisposición para ayudar.
- Al equipo docente de Organización de Computadoras por brindarnos su perspectiva y apoyo sobre el diseño de la herramienta.
- A Miguel Barraza por su buena predisposición a probar la herramienta y ayudarnos a que sea mas accesible.

Agradecimientos de Francisco

- A Pablo, por confiar en mí y ofrecerme trabajar con él en este hermoso proyecto.
- A Fede, por acompañarnos siempre, aguantarnos y compartir de su humor y sabiduría.
- A mis compañeros, amigos y familia, por siempre estar ahí apoyándome y escuchándome.
- A la Universidad Nacional de Quilmes, por el maravilloso clima que siempre genera, haciendo que sin importar quién esté en ella, se sienta cómodo y bienvenido.

Agradecimientos de Pablo

- A Fede por invitarme a colaborar con la materia y darme libertad para crecer en la docencia.
- A Tati por escucharme hablar de mis cursadas como si fuese algo novedoso para ella.
- A la Universidad Nacional de Quilmes, por darme un espacio lleno de compañerismo y por la calidad de los y las docentes que la componen.
- A la dirección de la carrera, en especial Gabi, que siempre estuvo para darme una mano.
- A mis compañeros y compañeras, que me ayudaron a llegar hasta donde estoy, muchas veces en sentido literal.
- A mi familia y amigos por escuchar mis explicaciones con más amor que entendimiento.

Capítulo 1

Introducción

La materia Organización de Computadoras es dictada en el ciclo básico de la Tecnicatura Universitaria en Programación Informática de la Universidad Nacional de Quilmes y brinda contenidos que aportan un panorama general sobre arquitectura de computadoras.

Una parte importante de la materia consiste en introducir ideas de arquitectura y organización de computadoras, en particular utilizando el modelo de Von Neumann. Para eso se utiliza una arquitectura teórica denominada Q, la cual provee un lenguaje de programación, también teórico. Esta tesis busca proveer un simulador para dicho lenguaje que sea fácil de usar, sin requisitos de instalación y accesible.

A continuación presentaremos las características básicas de una arquitectura de Von Neumman y de su enseñanza en Organización de Computadoras en la Universidad de Quilmes.

Arquitectura de Von Neumann

La arquitectura de Von Neumann, también conocida como arquitectura Princeton, es una arquitectura de computadoras basada en la descrita en 1945 por el matemático y físico John von Neumann.

Formalmente, una computadora es una máquina de Von Neumann si:

- Tanto los programas como los datos se almacenan en una memoria en común.
- Cada celda de memoria de la máquina se identifica con un número único, llamado dirección.
- Las diferentes partes de la información (las instrucciones y los datos) tienen diferentes modos de uso, pero la estructura no se representa en memoria de manera codificada. Es decir que a priori no hay forma de distinguir si algo almacenado en memoria es un dato o una instrucción.
- Cada programa se ejecuta de forma secuencial de a una instrucción por vez.

Nuestras computadoras utilizan una arquitectura basada en el modelo de de Von Neumann, la cual consta de una Unidad Central de Procesamiento (CPU por sus siglas en inglés) y una memoria donde se almacenarán tanto los datos como los programas. En esta arquitectura se utiliza el sistema binario para representar información, es decir que tanto los datos como los programas deben poder representarse usando solamente $1s$ y $0s$.

El proceso de ejecución en la arquitectura de Von Neumann consta básicamente de 3 etapas: *fetch*, *decode*, *execute*. La etapa fetch consiste en buscar las instrucciones previamente ensambladas en memoria. Una vez que se encontraron las instrucciones en memoria, estas son decodificadas en la etapa de decode, donde se resuelven los valores de sus operandos, si hubiese alguno. Como paso

final, se ejecuta la instrucción en la etapa execute. Dicha ejecución puede llevar a cabo cambios en el estado de la computadora.

Programar utilizando solamente binario es una tarea extremadamente ardua, por lo que para interactuar con la arquitectura se utilizan lenguajes de programación de bajo nivel, como puede ser el lenguaje ensamblador. Estos lenguajes permiten escribir programas de un modo más amigable para el ser humano. El código escrito en ensamblador es luego traducido a binario mediante un programa denominado también ensamblador. Este paso es necesario, ya que la máquina solo puede utilizar binario. Al proceso de convertir código fuente de un lenguaje como ensamblador a código binario que puede ser ejecutado por la máquina, se lo denomina *ensamblado*.

Enseñanza en la UNQ

En la materia Organización de Computadoras se creó una arquitectura conceptual llamada Q, utilizando el modelo de la Arquitectura de Von Neumann. Dicha arquitectura se define como conceptual ya que no existe una máquina física que funcione en ella. A su vez, esta arquitectura cuenta con el lenguaje homónimo para su utilización. Está pensada para la enseñanza de arquitecturas de computadoras en un nivel inicial. En 2 se presenta a la arquitectura Q en profundidad.

Como la materia es dictada en el ciclo básico, muchos y muchas estudiantes encuentran la complejidad de no poder ver el resultado de sus programas, ya que deben comprender los conceptos para poder simularlos manualmente.

En este sentido, QSim (2013) [1], un simulador de la arquitectura Q que fue desarrollado hace algunos años en el contexto de otro trabajo de inserción profesional, realiza un aporte significativo, brindando una interfaz visual para la ejecución de programas Q.

Sin embargo, dadas las tecnologías y el paradigma de diseño de aplicaciones utilizado, resulta dificultoso para los y las estudiantes su uso actual. Algunos de los inconvenientes encontrados son, por ejemplo, que quienes quieren usarlo deben tener instalado un conjunto de herramientas, como son Java Runtime Environment (JRE) en la versión 7 ya que no es compatible con las versiones siguientes. Además debe saber ejecutar scripts como archivos .bat o .sh, lo cual aporta complejidades accidentales que podrían ser evitadas. Estas herramientas son necesarias porque la tecnología utilizada para la construcción de QSim es el lenguaje Java y la herramienta está pensada para ser una aplicación de escritorio.

Otra desventajas son que no puede ser utilizado en dispositivos móviles, ya que las tecnologías utilizadas y la solución propuesta no contemplaban el uso masivo que vemos actualmente; falta de accesibilidad para personas que utilicen lectores de pantalla y problemas de rendimiento, como la baja velocidad.

Es por eso que se decidió realizar una implementación del lenguaje Q utilizando otras tecnologías que nos permitan evitar o minimizar las dificultades encontradas en la herramienta mencionada. El propósito de este trabajo es crear un simulador que permita ejecutar programas en la arquitectura Q. De esta manera las y los estudiantes de la materia tendrán la posibilidad de escribir programas en dicho lenguaje y ver el impacto de los mismos en las partes que componen a la arquitectura, como son registros, memoria, flags y registros especiales. Esta herramienta debe cumplir las siguientes propiedades:

- Poder ejecutar cualquier programa de la arquitectura Q.
- Permitir la visualización de los resultados o los errores de ejecución.
- No requerir instalación.
- Poder usarse desde dispositivos móviles.
- Ser accesible para estudiantes con disminución visual o daltonismo.

A su vez, este trabajo tiene como objetivo la creación de una librería escrita en javascript, que pueda ser utilizada por otras herramientas que busquen un objetivo similar, separando la lógica del lenguaje de su visualización, permitiendo así, mantener actualizadas las herramientas visuales sin tener que actualizar la lógica del lenguaje.

Resumen del trabajo

Esta trabajo se organiza de la siguiente manera:

El capítulo 1 es esta introducción, que pretende establecer el concepto de Arquitectura de computadoras de manera básica, la arquitectura Q planteada en la universidad y las motivaciones que llevan al desarrollo de QSim Web, incluyendo un análisis de las herramientas existentes.

El capítulo 2 explicará más en detalle la estructura de la arquitectura Q, definiendo sus capas y los conceptos que se introducen en cada una de ellas.

El capítulo 3 realiza una descripción de la totalidad del trabajo, diferenciando la interfaz de usuario de la librería QLib.

El capítulo 4 cuenta las devoluciones de los y las estudiantes luego de haber utilizado la herramienta durante un cuatrimestre.

El capítulo 5 describe las conclusiones desarrolladas a partir de la realización del trabajo.

El capítulo 6 se encuentran los anexos. El anexo 6.1 presenta la especificación completa de la arquitectura Q. El anexo 6.2 profundiza los detalles técnicos de la implementación del `Parser`, `Translator` y el proceso de ensamblado. El anexo 6.3 describe una posible implementación de una interfaz de usuario diferente a la entregada por este trabajo, utilizando la librería entregada por el mismo. El anexo 6.4 describe una posible implementación de otro parser, traduciendo las instrucciones al lenguaje español.

Capítulo 2

Arquitectura *Q*

En la materia Organización de Computadoras se creó una arquitectura de computadoras teórica llamada *Q* [11], utilizando el modelo de la arquitectura de Von Neumann. A su vez, esta arquitectura cuenta con el lenguaje homónimo para su utilización. Está pensada para la enseñanza de conceptos de organización y arquitecturas de computadoras en un nivel inicial.

Su diseño está dividido en 5 capas llamadas *Q1*, *Q2*, *Q3*, *Q4*, *Q5*. Cada capa contiene los elementos conceptuales de la capa anterior, agregando nuevas funciones a la misma. Esto permite que un programa escrito en *Q1* pueda ser ejecutado en *Q5*, obteniendo el mismo resultado. Esta división en capas permite la introducción paulatina de los conceptos que se abordarán en la materia evitando una curva de aprendizaje muy pronunciada.

2.1. Características generales de la arquitectura

Las computadoras sólo pueden manejar valores binarios, por lo tanto para interactuar con ellas deberíamos utilizar código binario, también llamado código máquina. Esta interacción, a medida que los programas se complejizan representa un trabajo demasiado engorroso y es necesario generar abstracciones que permitan evitar el trabajo directo con código máquina.

Estas abstracciones en la arquitectura *Q* son las instrucciones. Las mismas representan operaciones que la computadora puede realizar y cada una de ellas se traduce a un código máquina específico y único. Algunas de estas instrucciones utilizan parámetros, llamados operandos.

Como los programas que realizamos necesitan guardar datos temporalmente para poder cumplir con su función, se dota a las computadoras de un conjunto de registros de uso general. Cuando nuestros programas necesitan almacenar más datos, se utiliza una memoria conocida como memoria principal. Se denomina modo de direccionamiento a la forma en la que se accede a un operando de una instrucción, por ejemplo hay un modo de direccionamiento para indicar que un operando está en un registro, o para indicar que está en una celda específica de la memoria.

La arquitectura de Von Neumann establece que tanto el programa como los datos que este utiliza se almacenan en la memoria principal. El proceso de transformar código fuente, es decir el que escribe un programador utilizando las instrucciones del lenguaje, en código máquina que se puede guardar en memoria, se denomina ensamblado. Para poder conocer el código binario de una instrucción debemos conocer su operación, los modos de direccionamiento de sus operandos (si tuviese) y los operandos concretamente. Cada una de estas partes tiene una traducción única a binario. Estos binarios se almacenarán en memoria, ordenados según el formato de la instrucción que se está analizando. En el caso de *Q*, la arquitectura define 5 formatos de instrucción que serán descriptos a más adelante.

Además de los registros de uso general, la computadora necesita almacenar algunos datos para su funcionamiento, para lo que utiliza registros de uso específico. Algunos de estos datos son la celda donde se encuentra la siguiente instrucción a ejecutar, almacenado en el registro *PC*; las

direcciones de retorno de las rutinas invocadas, almacenadas en la pila, una sección reservada de la memoria principal; la dirección de memoria donde se guardará la próxima llamada a rutina, almacenada en el *SP*; la instrucción que se está ejecutando actualmente, almacenada en el IR; la dirección de memoria que se está leyendo o escribiendo, almacenada en el *MAR*; el valor de la dirección de memoria que indica el *MAR*, almacenado en el *MBR*.

El *PC* se incrementa al ejecutarse una instrucción, apuntando su valor a la dirección donde comienza la siguiente. Existen contextos de uso donde puede ser necesario alterar el valor del *PC* de modo que la próxima instrucción a ejecutar no sea la que está ensamblada inmediatamente luego de la que se está ejecutando. Por ejemplo, se puede querer repetir un bloque de código para lo que se debe poner en *PC* la dirección de la celda donde comienza a estar ensamblado dicho bloque. Esto se logra con instrucciones de salto, instrucciones cuyo efecto es alterar el valor del *PC* y por lo tanto establecer cuál es la próxima instrucción a ejecutar. Existen saltos incondicionales, es decir instrucciones cuya ejecución siempre modifica el valor de *PC*; y saltos condicionales, que según el estado de la máquina alteran o no el valor del contador de programa. Este tipo de saltos son fundamentales para codificar estructuras condicionales, es decir bloques de código que se ejecutan o no dependiendo del estado de la máquina. Estas instrucciones de saltos condicionales están asociados a registros especiales denominados *flags*, los cuales son registros de un bit cuyo valor es establecido por la máquina en base al resultado de la ejecución de instrucciones aritméticas.

Como los programas o rutinas son ensamblados en memoria, muchas veces resulta difícil determinar en qué posición de memoria se ensamblarán. El lenguaje *Q* provee un mecanismo de etiquetas que permite abstraer la dirección de memoria donde se ensamblan los programas o rutinas, permitiendo definirlas asociadas a determinadas instrucciones y utilizarlas como operandos de otras. Estas etiquetas son asociadas a direcciones de memoria en el momento del ensamblado. Sus definiciones no pueden repetirse ya que funcionan como identificadores.

2.2. Capas de la arquitectura

Como fue mencionado anteriormente, la arquitectura *Q* cuenta con 5 capas conceptuales y en esta sección se detallarán los objetivos teóricos y conceptuales de cada una.

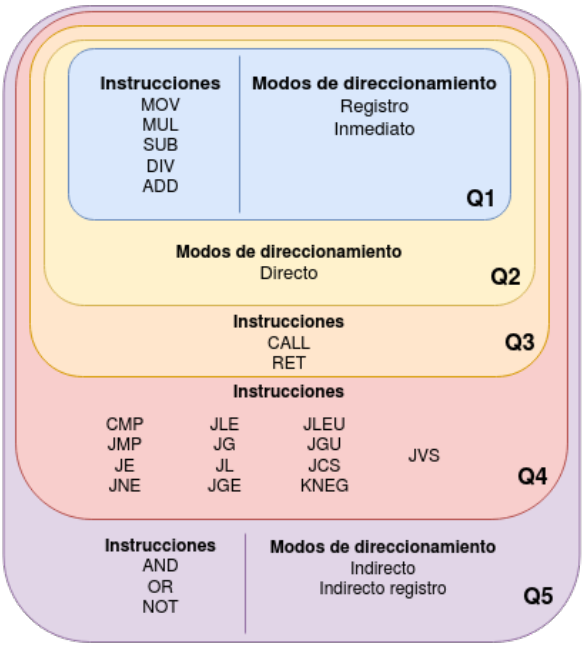


Figura 2.1: Capas de la arquitectura como conjuntos y superconjuntos.

Q1

El objetivo de esta capa es introducir algunos conceptos básicos de la programación de arquitecturas de bajo nivel, como son las instrucciones, registros y el modo de direccionamiento inmediato.

Sus características son: 8 registros de uso general, nombrados de R0 a R7; un modo de direccionamiento para trabajar con constantes llamado modo *inmediato* y 5 instrucciones de dos operandos cada una, MOV, ADD, SUB, DIV, MUL.

Los registros tienen 16 bits de almacenamiento. Los operandos del modo *inmediato* tienen 16 bits de longitud y se escriben en hexadecimal, es decir base 16, con la sintaxis: $0\text{x}N_1N_2N_3N_4$ donde N_i es un dígito en hexadecimal.

Un ejemplo de instrucción válida es *ADD R7, 0x23FA* que suma el valor *0x23FA* al valor contenido en *R7* guardando el resultado en dicho registro.

Un ejemplo de instrucción inválida es *ADD 0x23FA, R7* porque una constante no puede almacenar el resultado de una operación.

Un ejemplo de ensamblado de la instrucción *ADD R6, 0x2323* en la celda *0x00F0*

Celda	Valor
...	...
0x00F0	0010 100110 000000
0x00F1	0010 0011 0010 0011
...	...

Q2

El objetivo de esta capa es brindar la posibilidad de interactuar con la memoria, algo que se omite en *Q1* por simplicidad.

Para ello, se agrega un modo de direccionamiento llamado Modo Directo el cual especifica la dirección de memoria donde se encuentra el valor del operando. La dirección del operando se escribe entre corchetes, por ejemplo: *[0x23AB]*.

Un ejemplo de instrucción válida es *ADD R7, [0x23FA]* que suma el valor de la celda *0x23FA* al valor contenido en *R7* guardando el resultado en dicho registro.

Un ejemplo de ensamblado de la instrucción *ADD R6, [0x2323]* en la celda *0x00F0*

Celda	Valor
...	...
0x00F0	0010 100110 001000
0x00F1	0010 0011 0010 0011
...	...

Q3

El objetivo de esta capa es brindar la posibilidad de escribir rutinas, que permiten explicar los conceptos de reutilización, modularización y documentación.

En esta capa toman relevancia los registros *PC*, *SP*, *IR*, *MAR*, *MBR* y los conceptos de pila y etiqueta.

Además de las características de las capas anteriores, en esta capa se agregan dos instrucciones: *CALL* y *RET* que permiten interactuar con rutinas.

CALL: Invoca a una rutina, para ello guarda el *PC* en la pila, decrementa el *SP* en 1 y copia el *PC* a la dirección de memoria donde está ensamblada la primera instrucción de dicha rutina. Un ejemplo de uso es:

CALL calcularPromedio

RET: Finaliza la ejecución de una rutina, para ello incrementa el *SP* y pone en el *PC* el valor

de la pila. Su uso es:

RET

Un ejemplo de ensamblado de la instrucción *CALL promedio* en la celda *0x00F0*, asumiendo que *promedio* se encuentra ensamblada en *0xAFFF*.

Celda	Valor
...	...
0x00F0	1011 000000 000000
0x00F1	1010 1111 1111 1111
...	...

Un ejemplo de ensamblado de la instrucción *RET* en la celda *0x00F0*.

Celda	Valor
...	...
0x00F0	1100 000000000000
...	...

Q4

El objetivo de esta capa es introducir los conceptos de ejecución condicional y repeticiones. Para ello se introducen los conceptos de flags y saltos. Los flags son registros de 1 bit que se calculan al ejecutarse instrucciones aritméticas.

- Z: es uno cuando el resultado de la operación es cero.
- N: es uno cuando el resultado de la operación es negativo.
- C: es uno cuando el resultado de la operación tiene carry (acarreo) en el caso de la suma o borrow (pedir prestado) en el caso de la resta, en el bit más significativo.
- V: es uno cuando el resultado de la operación tiene overflow en complemento a dos. Se considera que una operación en complemento a dos tiene overflow cuando se cumple alguna de las siguientes condiciones: Se suman dos números positivos y el resultado es negativo; Se suman dos números negativo y el resultado es positivo; Se resta un número negativo a un positivo y el resultado es negativo; Se resta un número positivo a un negativo y el resultado es positivo;

Las características de esta capa son, además de las mencionadas anteriormente, saltos condicionales relativos y saltos incondicionales absolutos. Para realizar un salto, estos modifican el registro *PC*, que indica cuál es la siguiente instrucción a ejecutar. Existen dos categorías de saltos: Una de ellas se basa en si existe una condición de salto, es decir, salta si se cumple dicha condición

y, por otro lado, según la forma de indicar el cambio en el *PC*. La primera categoría divide a los saltos en condicionales e incondicionales. Un ejemplo de condición podría ser que una determinada cuenta dio como resultado cero. La segunda categoría divide a los saltos en absolutos y relativos. Los saltos absolutos indicarán exactamente el nuevo valor del *PC*. Los saltos relativos en cambio, indicarán el desplazamiento que se le aplicará al *PC*, basándose en su valor actual.

La instrucción salto incondicional absoluto es *JMP*, que copia el *PC* a la dirección de memoria indicada por su etiqueta. Un ejemplo de uso es:

```
JMP esNumeroPositivo
```

La ejecución de las instrucciones de saltos condicionales relativos tiene como efecto incrementar o decrementar el *PC* dado un desplazamiento, si su condición de salto se cumple. Utilizando un desplazamiento negativo se puede saltar hacia celdas de dirección menor mientras que usando un desplazamiento positivo se salta a celdas con dirección mayor al valor actual del *PC*. Para ver el listado completo de estas, ver el anexo 6.1.

Q5

En esta última capa se definen las instrucciones AND, OR y NOT y dos modos de direccionamiento indirectos, que permiten el trabajo con recorridos de arreglos e iteraciones. Si bien es técnicamente posible realizar lo mencionado sin estos modos, agregaría una complejidad no buscada en la materia. Estos modos de direccionamiento son *indirecto* e *indirecto registro*.

El modo de direccionamiento *indirecto* especifica la dirección de memoria donde se encuentra la dirección de memoria que contiene el valor del operando. La dirección del operando se escribe entre doble corchetes, por ejemplo: *[[0x23AB]]*.

El modo de direccionamiento *indirecto registro* especifica el registro donde se encuentra la dirección de memoria que contiene el valor del operando. El registro se escribe entre corchetes, por ejemplo: *[R5]*.

Un ejemplo de instrucción es *ADD [R6], [[0x2323]]* que suma el contenido de la celda a la que apunta el valor de la celda *0x2323* al valor de la celda a la que apunta el valor del registro *R6*.

Un ejemplo de ensamblado de la instrucción *ADD [R6], [[0x2323]]* en la celda *0x00F0*:

Celda	**Valor**
...	...
0x00F0	0010 110110 011000
0x00F1	0010 0011 0010 0011
...	...

Capítulo 3

Qsim web

Una vez presentada la problemática que buscamos resolver asi como el contexto necesario para comprender a la arquitectura *Q*, presentaremos en este capitulo la solución construida en este trabajo.

QSim web es una aplicación web que permite, desde un enfoque educativo, realizar la ejecución de programas escritos en el lenguaje Q permitiendo visualizar los cambios realizados en los componentes de la arquitectura Q, es decir, registros especiales, registros de uso general, memoria, flags y pila.

Consideramos que la interfaz de usuario y la lógica del lenguaje Q son partes separadas. Esta concepción permite que la lógica del lenguaje pueda ser reutilizada para otras interfaces de usuario. Esta idea de reutilización de la lógica del lenguaje, dio pie a la construcción de una librería. Se considera una librería a un software específico para determinada función, que provee una interfaz para su utilización por parte de otros/as desarrolladores/as.

La entrega de este trabajo cuenta con una librería que implementa la especificación de la arquitectura Q, la cual será descripta en la sección 3.2 y una interfaz de usuario que permite hacer uso de la librería para construir un entorno que pueden utilizar los estudiantes, que se describe en 3.1.

Esta implementación está subida a un servidor utilizando Heroku [12], en el siguiente link: https://qweb-unq.herokuapp.com/.

Su código fuente está subido a Gitlab y puede verse en el siguiente link: https://gitlab.com/qsim-web-project/qsim-web.

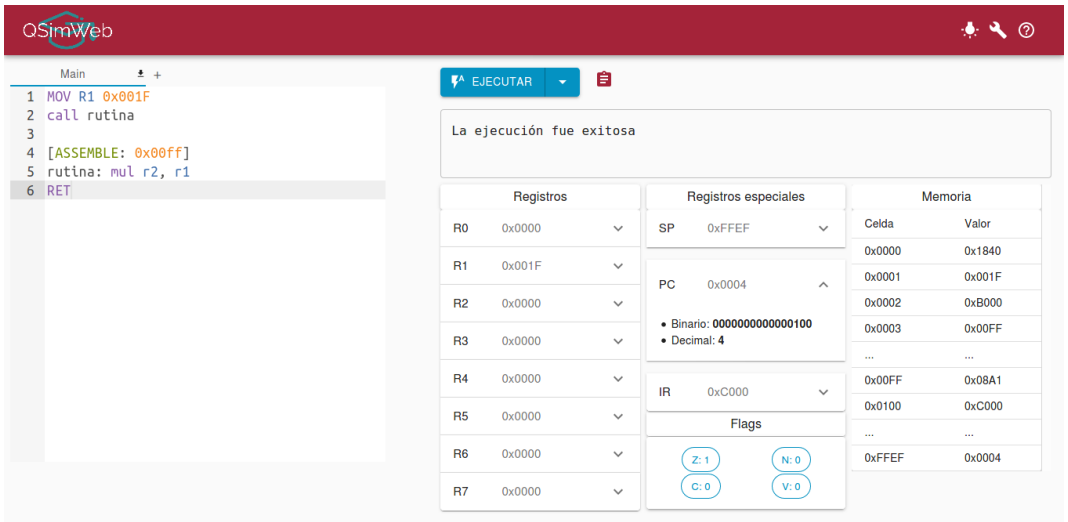


Figura 3.1: Vista previa de QSim web.

3.1. Interfaz de usuario

La interfaz cuenta con una pantalla inicial donde se pueden escribir y ejecutar programas Q, permitiendo visualizar los resultados de la ejecución. Buscamos construir una interfaz amigable, simple de utilizar y accesible para usuarios que pudieran tener dificultades visuales.

La interfaz de usuario se encarga de coordinar los pasos requeridos para ejecutar un programa utilizando la librería, es decir, parseo, traducción, ensamblado y ejecución del código.

Las secciones nombradas anteriormente se describen a continuación:

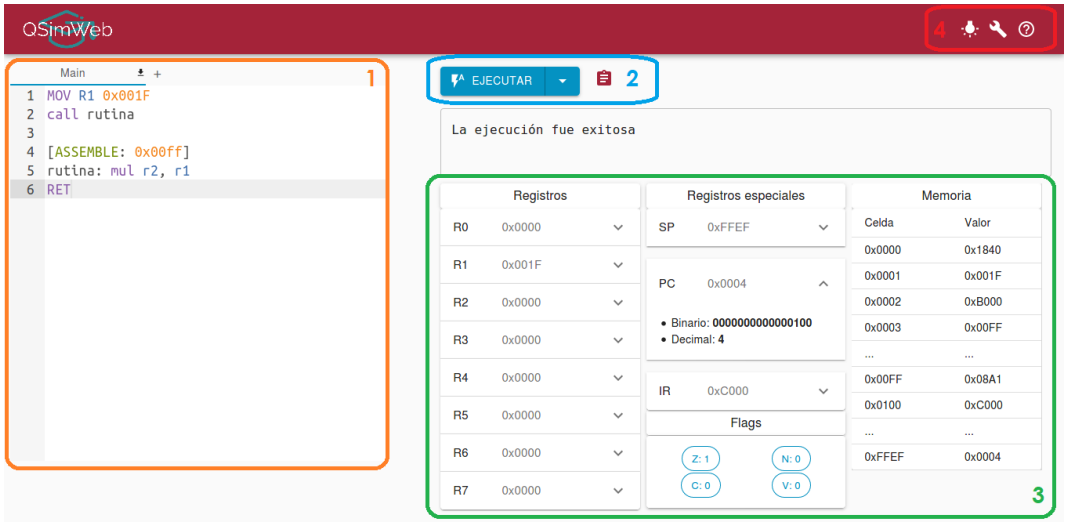


Figura 3.2: Componentes de la interfaz de usuario.

Esta sección presenta un recorrido sobre las distintas partes de la interfaz de usuario y sus funcionalidades.

1. Editor

Los programas se escriben sobre un editor de texto que cuenta con resaltado de sintaxis y errores. Estos programas pueden ser escritos tanto en mayúsculas como en minúsculas. Cuando las instrucciones tienen más de un operando, estos pueden ser separados mediante uno o más espacios o una coma. El editor fue desarrollado utilizando la librería react-ace.

Figura 3.3: Editor de texto con resaltado de sintaxis.

Uno de los contenidos de la materia es el uso de rutinas, que permiten reutilizar código y asignarle un nombre mediante etiquetas. Se permiten crear rutinas, ensamblándolas en celdas específicas utilizando [assemble: X], donde X es un inmediato representando la celda de memoria donde se ensamblará la rutina. Además, su etiqueta es definida posteriormente con la forma `nombreEtiqueta:`.

```
1 MOV R1 0x0002
2 CALL sumar1
3
4 [ASSEMBLE: 0x00F0]
5 sumar1: ADD R1, 0x0001
6 RET
```

Figura 3.4: Ejemplo del uso de ASSEMBLE: la rutina `sumar1` se ensamblará a partir de la celda *0x00F0*.

La herramienta además permite la realización de comentarios que no afectarán la ejecución como se muestra en la figura 3.5. Un comentario comienza con # y termina al final de la línea.

```
1 MOV R1 0x0002
2 CALL sumar1
3
4 #Assemble sirve para definir dónde se
5 #ensambla la rutina.
6 [ASSEMBLE: 0x00F0]
7 sumar1: ADD R1, 0x0001 #Sumo uno a R1
8 RET
```

Figura 3.5: Ejemplo del uso de comentarios.

Para facilitar la organización del código e incluso fomentar a las y los estudiantes a compartirlo con sus compañeros/as, cuenta también con la posibilidad de dividir el código en distintas pestañas, importación de archivos .txt y descarga de los programas. Esto puede verse en la figura 3.6.

Figura 3.6: Funcionalidades extra del editor.

2. Botón de ejecución

Cuando se quiere probar el código escrito, se debe ejecutar, haciendo click en el botón de ejecución mostrado en la figura 3.7. La interfaz da la opción de ejecutar un programa de 3 modos distintos: Ejecución completa, Ejecución por instrucción y Ejecución por instrucción detallada.

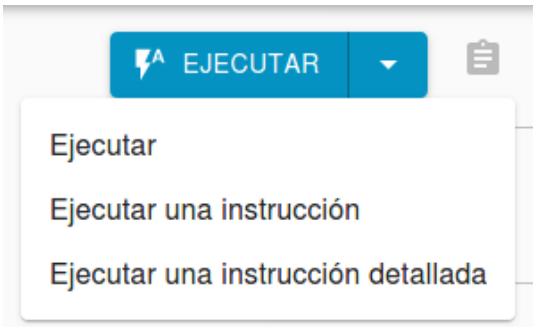


Figura 3.7: Opciones de ejecución.

El modo Ejecución completa realiza la ejecución de la totalidad de las instrucciones del programa mostrando, al final, los cambios en la sección de resultados.

Ejecución por instrucción realiza la ejecución de una instrucción y muestra los cambios en la sección de resultados, al hacer click nuevamente, ejecutará la siguiente instrucción si existe. Si no existe se comenzará a ejecutar el programa nuevamente.

Para el modo de Ejecución por instrucción detallada, mostrará mensajes con las acciones producidas en cada paso de la ejecución, como se muestra en la figura 3.8.

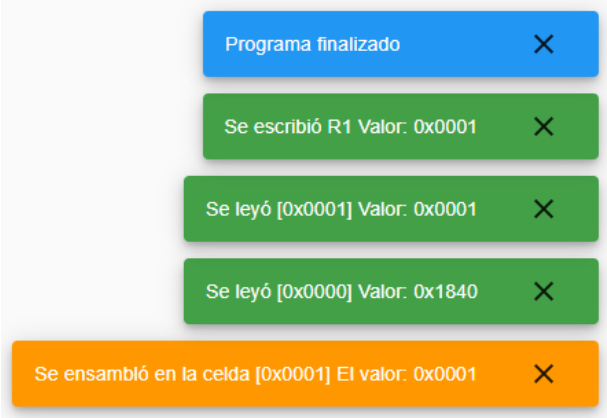


Figura 3.8: Ejecución detallada.

En todos los modos de ejecución, dichas acciones quedan registradas y disponibles para su revisión como se muestra en la imagen 3.9 Se puede acceder a este registro haciendo click en el ícono que se encuentra a la derecha del boton de ejecución, como ve en la imagen 3.7

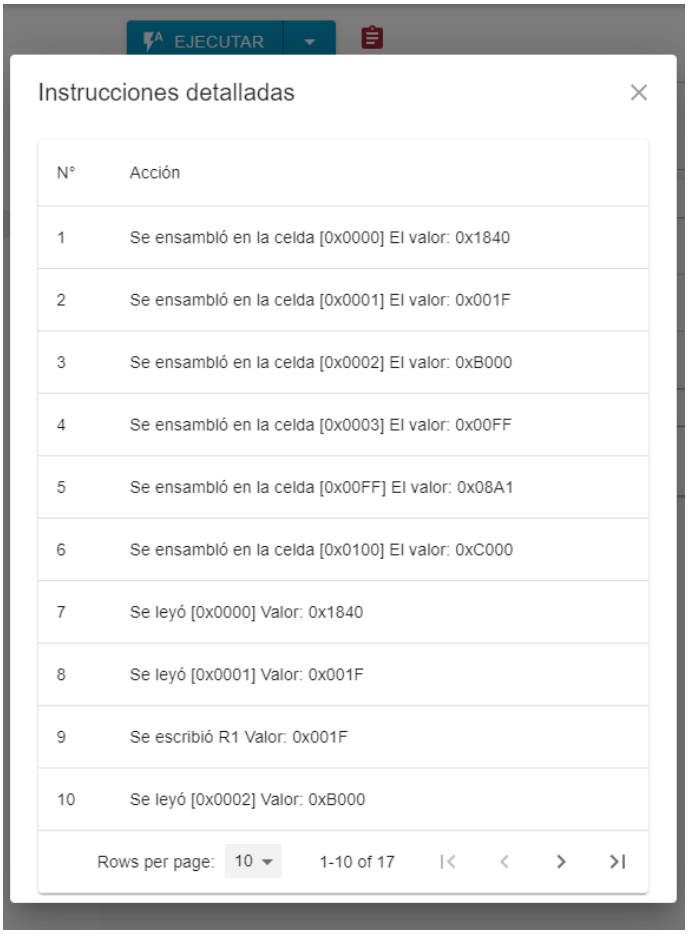


Figura 3.9: Acciones de la ejecución.

3. Resultados

Una vez finalizada la ejecución, con o sin errores, se actualizará la sección de resultados.

Los resultados se componen de los valores de registros, registros especiales, flags y memoria como puede verse en la figura 3.10. Los valores de registros y registros especiales pueden ser visualizados en hexadecimal, binario y decimal.

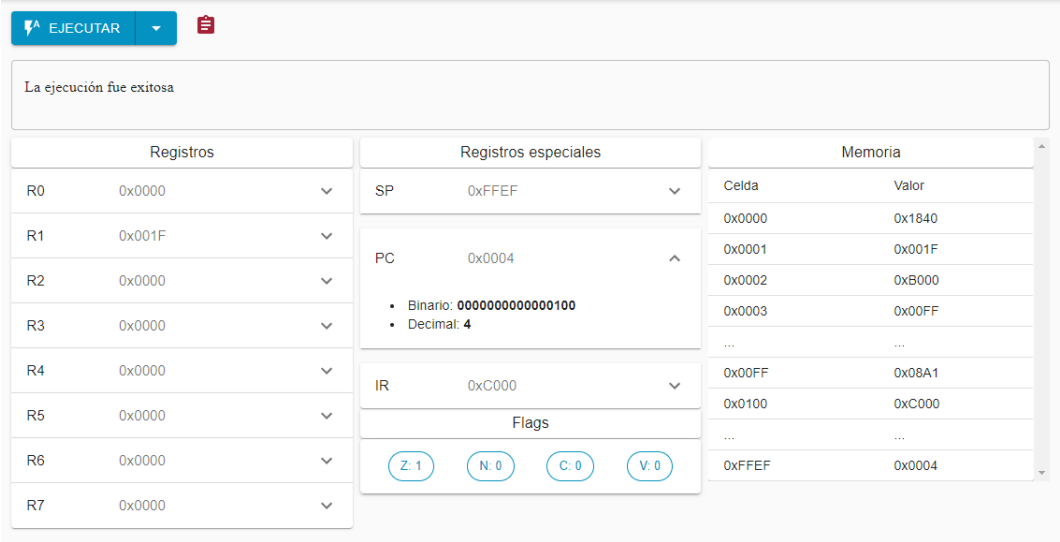


Figura 3.10: Resultados de la ejecución, separados en registros, registros especiales y memoria.

Cuando la memoria contiene celdas inicializadas que no son contiguas se resumen las celdas intermedias colocando "...", como puede verse en la figura 3.11.

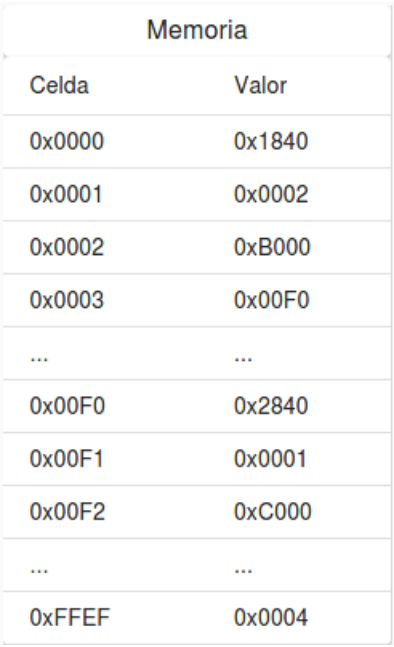


Figura 3.11: Visualización de la memoria.

Si durante la ejecución se produjo un error, este será mostrado en pantalla indicando, de ser posible, acciones a tomar para su corrección. Un ejemplo de error puede verse en la figura 3.12.

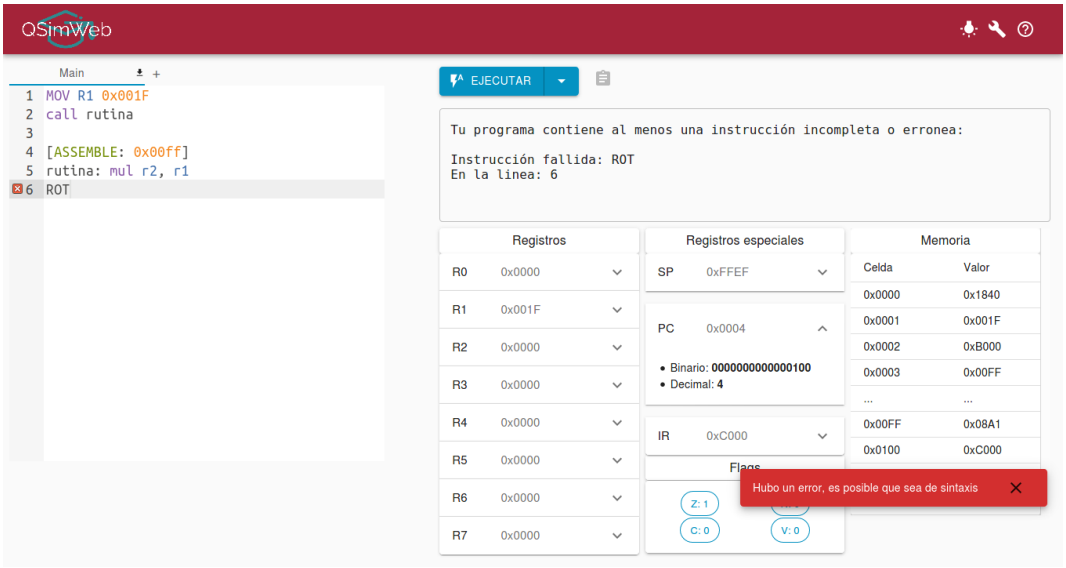


Figura 3.12: Error de sintaxis ocurrido al presionar ejecutar.

4. Configuraciones y ayuda

La ventana principal cuenta además con un menú que permite configurar parámetros de la ejecución.

Figura 3.13: Panel de configuraciones.

En la configuración se permite cambiar la cantidad de registros, el comportamiento de MUL respecto al registro R7, los valores por defecto de las celdas de memoria y registros, los modos de direccionamiento e instrucciones habilitadas. Es decir, todas las configuraciones que se especificarán en QConfig (ver 3.2.4). Además permite cargar un archivo de configuración para simplificar el armado de la misma.

Estas preferencias mencionadas se guardan en el navegador del usuario, permitiendo que no se pierdan al finalizar su uso. Además se permite importar un archivo de configuración, donde se pueden especificar las configuraciones nombradas anteriormente. Como se presentó anteriormente, la arquitectura Q define capas con diferentes caracteristicas: instrucciones disponibles y modos de direccionamiento. En nuestro repositorio[1] en GitLab se encuentran archivos que permiten fácilmente importar las configuraciones de las distintas Q_i.

Para brindar una ayuda en términos sintácticos a quienes utilizan la herramienta, se permite además ver un manual de uso donde se aclara el uso de etiquetas, la definición de rutinas ensambladas en celdas específicas, uso de comentarios y todas las instrucciones y modos de direccionamiento de la herramienta.

[1] https://gitlab.com/qsim-web-project/qsim-web/-/tree/develop/src/qweb/configs

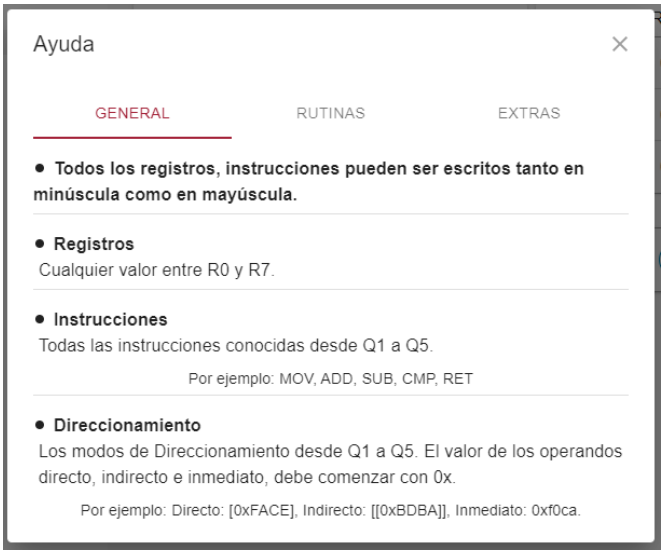


Figura 3.14: Panel de ayuda.

Dado que uno de los objetivos es que la herramienta sea accesible, esta implementa un modo oscuro, que facilita el uso por parte de personas con daltonismo.

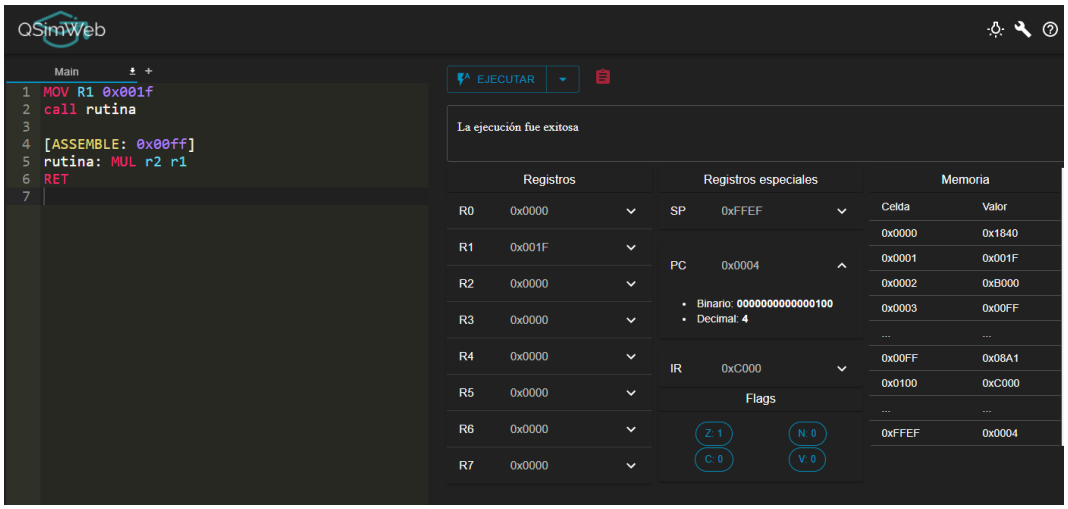


Figura 3.15: Modo oscuro.

3.1.1. Tecnologías utilizadas

La construcción de la interfaz de usuario fue realizada en JavaScript, utilizando React con la librería Material-UI [2] y react-ace [3] para el editor.

3.1.2. Accesibilidad

Una de las funcionalidades esperadas era brindar la posibilidad de que la herramienta pueda ser usada con lectores de pantalla. En este sentido se hicieron evaluaciones de accesibilidad utilizando herramientas estandarizadas y una prueba de campo con un estudiante no vidente de la tecnicatura.

Una vez lograda una interfaz de usuario funcional utilizamos el analizador de sitios web *WAVE* Web Accessibility Evaluation Tool [4] el cual nos permitió detectar problemas o incompatibilidades para lectores de pantalla utilizados por usuarios no videntes. La herramienta es muy útil porque no solo indica problemas sino que además brinda detalles sobre cómo resolverlos.

Los problemas más comunes que la herramienta nos marcó estaban asociados a la falta de etiquetas en los componentes visuales y a problemas de contraste en estos mismos. Definir las etiquetas es muy importante a la hora de usar un lector de pantalla ya que con estas, el usuario puede identificar rápidamente el propósito o la descripción del elemento que esté señalando con el mouse. Así también, los problemas de contraste pueden ser contraproducentes para usuarios con baja visión.

En la siguiente imagen, se puede apreciar como en el recuadro A, *WAVE* nos señala el motivo del problema y una posible solución. En el recuadro B, se ve el elemento que tiene un problema y cuál es este.

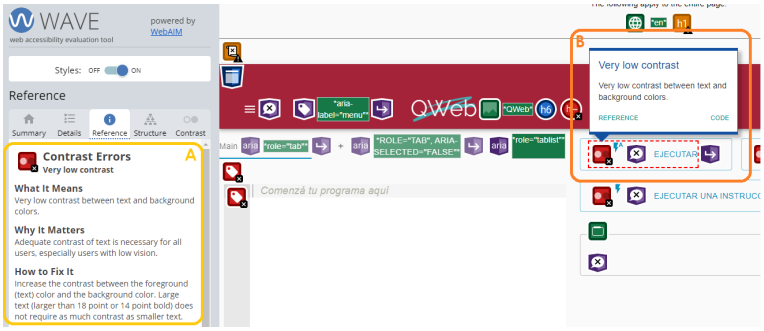


Figura 3.16: Herramienta WAVE sobre una versión antigua de QSim web.

Luego de haber resuelto los principales problemas detectados con la herramienta, recurrimos a Miguel Barraza, un alumno no vidente de la carrera, a quien le pedimos que escribiera y ejecutara un programa en la interfaz y nos comentara su experiencia y que en particular nos indicara problemas u oportunidades de mejora si es que las encontraba. Desde un principio pudo utilizar la herramienta pero con ciertas dificultades. Él, al ser un usuario con experiencia y que ha trabajado con páginas web, pudo identificar dónde estaban los problemas y hasta nos brindó documentación y consejos para poder mejorar la experiencia de uso. Dentro de la documentación aportada por Miguel resaltamos los estándares y pautas propuestos por W3C [6] en su iniciativa *WAI* Web Accessibility Initiative [5] y el apartado de accesibilidad que aporta Google en Web Fundamentals [**?**]accesibilidad-google).

Luego de dos encuentros con Miguel, se realizaron varias mejoras, como por ejemplo agregar etiquetas a todos los botones y componentes visuales que la herramienta no marcó pero Miguel pudo identificar con el uso. El atributo html utilizado es `aria-label`, *ARIA* son las iniciales de

Accessible Rich Internet Applications (Aplicaciones de Internet enriquecidas y accesibles) y brinda varios mecanismos para agregarles etiquetas y descripciones a elementos. El atributo `aria-label` puede ser usado cuando se tiene algún tipo de indicación visual del objetivo de un elemento, como un botón que use un gráfico en lugar de texto, pero igualmente se necesita aclarar el objetivo para quien no pueda acceder a la indicación visual, como un botón que use solo una imagen para indicar su propósito; agregar encabezados a la sección de resultados, para permitir su navegación y lectura, lo cual en un principio no era posible; corrección de autofoco en los modales, para que una vez que se accediera al modal, el foco esté dentro de este y no por fuera. Esto dificultaba la experiencia de uso ya que no se podía reconocer simplemente que se había logrado acceder al modal; se nombró correctamente el lenguaje de la aplicación, ya que React pone por defecto el lenguaje en inglés. Esto no es de ayuda para los lectores de pantalla ya que este estaría esperando texto en inglés cuando en realidad se encuentra en español. Esto se logró simplemente cambiando el valor por defecto del atributo "$lang$" `<html lang=`"en"`>` por "$es - AR$"; e hicimos que el contenido de las ventanas emergentes se lea automáticamente por el lector de pantalla. Sin esto, sería difícil que el lector note el cambio y se logró añadiendo atributos $ARIA$ a la ventana emergente.

A partir de los encuentros mencionados, donde Miguel identificaba problemas de accesibilidad para su posterior corrección, logramos nuestra meta de hacer de QSim web una herramienta más accesible que puede ser usada satisfactoriamente por usuarios con problemas de visión. Además, nos ayudó a comprender y darnos cuenta de cómo con pequeñas modificaciones en nuestro código, mayormente atributos html, se puede mejorar la experiencia de uso de aplicaciones web.

3.2. Librería

El trabajo tiene dos entregables principales, la herramienta Qsim web que puede ser usada en clase por los alumnos y una librería que implementa la lógica necesaria para trabajar con la arquitectura *Q*. En esta sección se presentarán las características de la librería así como los detalles de su implementación.

La librería creada por este trabajo recibe el nombre de *QLib*. El objetivo de la creación de la librería es brindar una base de código que cuente con las características de la arquitectura *Q* separadas de su representación visual. Está escrita en javascript utilizando el paradigma de programación orientada a objetos y permite su integración con aplicaciones web.

Además de ser considerada una buena práctica, la existencia de una librería así permite separar nuestro modelo de las vistas, reduciendo el número de dependencias entre ellos. En consecuencia un cambio hecho en una de las capas tiene un impacto posible muy bajo sobre las otras. Esta separación además mejora la reusabilidad del código. Por ejemplo, podemos crear varias representaciones visuales para el mismo modelo (temas cambiables), dando la posibilidad de extender o modificar fácilmente la interfaz entregada en este trabajo.

En la siguiente figura, se puede apreciar cómo podrían convivir dos representaciones visuales utilizando la misma librería.

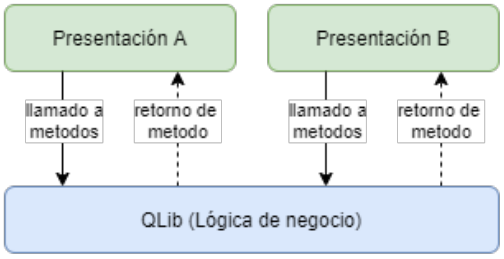


Figura 3.17: Separación de la lógica y la presentación.

En el anexo 6.3 puede encontrarse una explicación detallada sobre cómo utilizar la librería para implementar una interfaz de usuario diferente a la presentada en este trabajo.

QLib está compuesta por tres capas principales: `Parser`, `Translator` y `Computer`. Cada una de ellas será explicada en las secciones 3.2.2, 3.2.3 y 3.2.4 respectivamente.

Estas capas interactúan con la interfaz de usuario y entre si para permitir escribir y ejecutar programas en *Q*. La figura 3.18 ilustra dichas interacciones.

Primero se escribe un código *Q* en el editor de texto en la interfaz. Esta se comunica con la librería brindando como entrada el código del usuario. La librería analiza sintácticamente el texto que representa al programa, esta tarea es realizada por el `Parser`. El `Parser` indica si ocurre algún error si existiera. En caso de no haber errores, se procede a traducir el resultado del parseo a instrucciones de la librería, tarea realizada por el `Translator`. Finalmente, se procede a ensamblar y ejecutar el programa, ambas tareas realizadas por `Computer`.

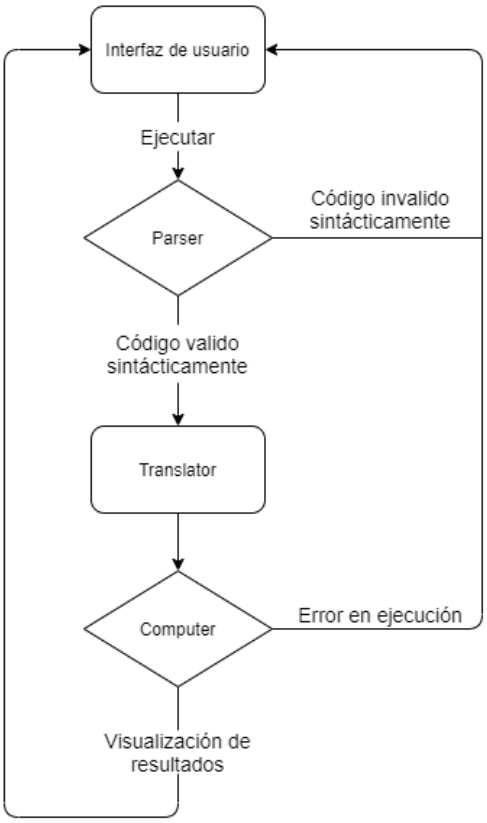


Figura 3.18: Proceso de ejecución separado por componentes.

Las funcionalidades que ofrece *QLib* son:

- Implementación del lenguaje Q: todas las instrucciones de Q, modos de direccionamiento, registros, flags, PC, SP, IR, MAR, MBR, etiquetas, pila están implementadas.
- Manejo de etiquetas: se proveen mecanismos para definir etiquetas, asociarlas a una dirección y utilizarlas en el ensamblado de programas.
- Manejo del ensamblado: métodos para pasar de un programa Q a la representación binaria de las instrucciones y viceversa.
- Ejecución completa, paso a paso y paso a paso detallada.
- Parseo del código Q.
- Capa de traducción entre el parseo y las instrucciones de la librería (detallada en 3.2.3).
- Opciones de configuración de la librería: modificar la cantidad de registros, valores por defecto de celdas y registros, modos de direccionamiento e instrucciones habilitadas en la ejecución.
- Exposición de valores del estado a lo largo de la ejecución.
- Exposición de acciones realizadas a lo largo de la ejecución.

3.2.1. Tecnologías usadas

La construcción de la librería fue realizada utilizando javascript. El parser utiliza la librería Nearley [7].

El testing unitario y de integración fue realizado con mocha [8] y cypress.io [9].

3.2.2. `Parser`

El parser es la unidad encargada del análisis sintáctico para determinar si el programa es correcto de acuerdo a la gramática del lenguaje *Q*. Una gramática se define como el conjunto de reglas del lenguaje que determinan si una entrada es válida o no. Recibe como entrada una cadena de texto y devolverá objetos javascript en caso de ser la entrada sintácticamente válida o un error en caso contrario. Dicha salida será la entrada del `Translator`, el cual es explicado en detalle en 3.2.3.

Para la definición de la gramática y su análisis sintáctico se utilizó la librería nearley.js, la cual provee una sintaxis para la definición de gramáticas basadas en *Extended Backus-Naur Form* [10]. Los detalles de EBNF escapan al desarrollo de este trabajo por lo que es suficiente con pensar que este funciona como un metalenguaje, es decir, un lenguaje que permite la construcción de otros lenguajes.

Para utilizar nearley.js se genera un archivo de gramática con la extensión .ne, donde se define la idea de instrucción, operando, tipos de instrucción y etiquetas. Luego, dicho archivo se compila a javascript utilizando la librería y esto permite el uso de la gramática en navegadores.

Una vez que la gramática está compilada, se utilizará la clase `Parser` provista por la librería para ingresar las instrucciones en forma de texto. Si son sintácticamente correctas, se devolverá un objeto preparado para traducirse a las instrucciones del lenguaje *Q*.

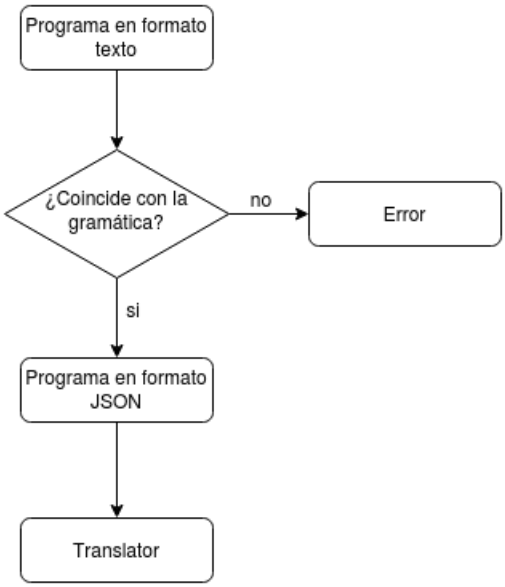


Figura 3.19: Proceso de parseo del código.

Veamos un ejemplo de cómo se parsea la instrucción ADD R3 0x3200:

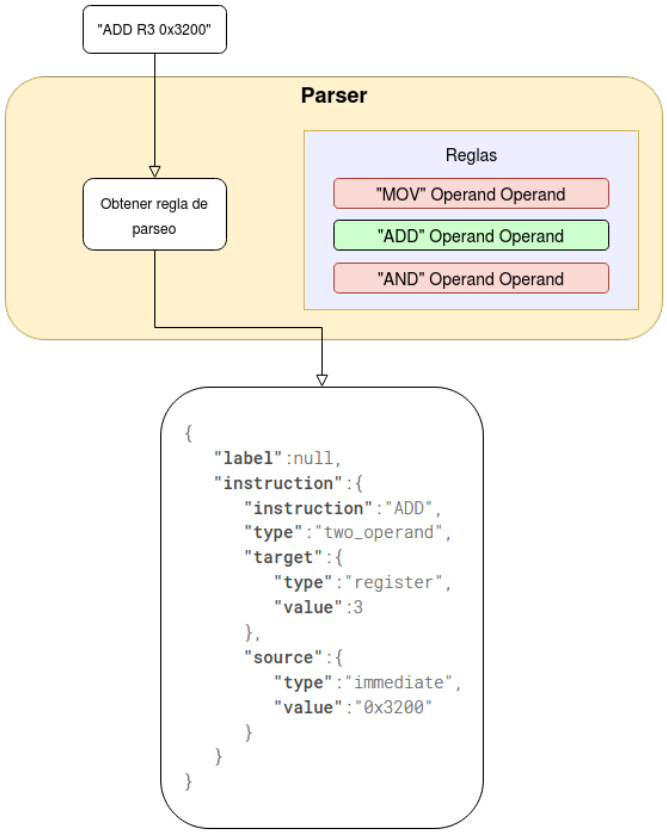


Figura 3.20: Ejemplo de parseo de la instrucción ADD R3 0x3200. Resaltada en verde la regla de parseo correspondiente a la operación ADD.

Este diseño que separa la etapa de parseo de la etapa de traducción a las clases del lenguaje permite la utilización de un parser distinto, siempre y cuando su salida respete el formato esperado por el `Translator`. Es decir, si no se quiere utilizar el parser provisto por *QLib*, se podría construir un parser distinto siempre y cuando su salida sea igual a la vista en la figura 3.20. Esta salida funciona como entrada del `Translator`, descripto a continuación. Más allá que el desacoplamiento sea una buena práctica de desarrollo de software, creemos que en este caso podría ser útil para traducir las instrucciones de *Q* al español, por ejemplo teniendo la instrucción *Saltar* en vez de *JMP*, quitando de esta manera una posible fuente de fricción para los estudiantes. En la sección 6.4 se encuentra un ejemplo de cómo se podría traducir el lenguaje Q a una versión en castellano.

Los detalles de implementación del parser pueden ser encontrados en la sección 6.2.1 del anexo.

3.2.3. `Translator`

El translator es otra parte fundamental de *QLib* acuando de interfaz entre los resultados brindados por el parser y la clase `Computer` para ejecutar el programa.

La clase `Translator` es la que permite que se pueda desacoplar el parseo del código de la logica de ejecución en Q. En ella se definen un conjunto de reglas que determinan cómo mapear las posibles entradas de formato JSON a las instancias de instrucciones soportadas por *QLib*. Estas reglas se dividen en 3 grupos: *reglas para instrucciones*, para *tipos de instrucciones* y *para operandos*.

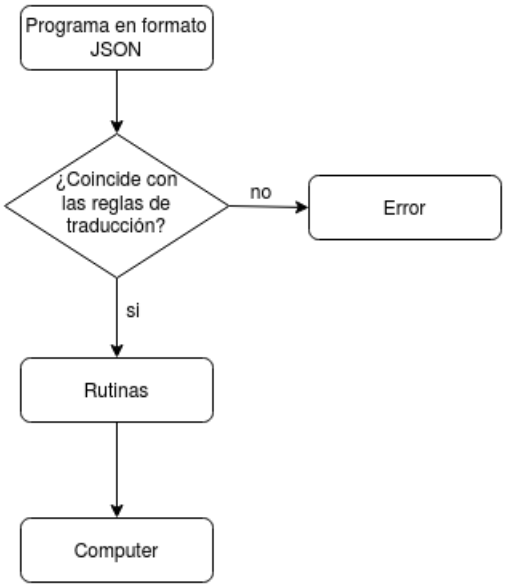


Figura 3.21: Proceso de traducción del código.

Cuando se quiere traducir un programa, se realiza el siguiente proceso línea por línea:

En base a la línea que se está traduciendo, se busca una instrucción que coincida con el nombre de la misma en las *reglas para instrucciones*. Una vez encontrada la instrucción, se busca el tipo de la instrucción en las *reglas de tipos*. El tipo de la instrucción sirve para determinar cómo se traducirán los operandos. Luego se obtienen los operandos de acuerdo al tipo de instrucción y el nombre del operando en las *reglas para operandos*, obteniendo así, una instancia de operando de *QLib*. Con estas búsquedas realizadas se puede instanciar la instrucción de *QLib* y así armar las rutinas resultantes. Una rutina, instancia de la clase `Routine` es un conjunto de instrucciones, asociadas a una dirección de comienzo. Dichas instrucciones se ensamblan a partir de la dirección de comienzo permitiendo así ser referenciadas en distintas partes del código, mediante etiquetas. Los detalles de implementación del `Translator` y de la clase `Routine` pueden verse en el anexo 6.2.2.

Siguiendo con el ejemplo visto en la figura 3.20, el proceso de traducción sería el siguiente:

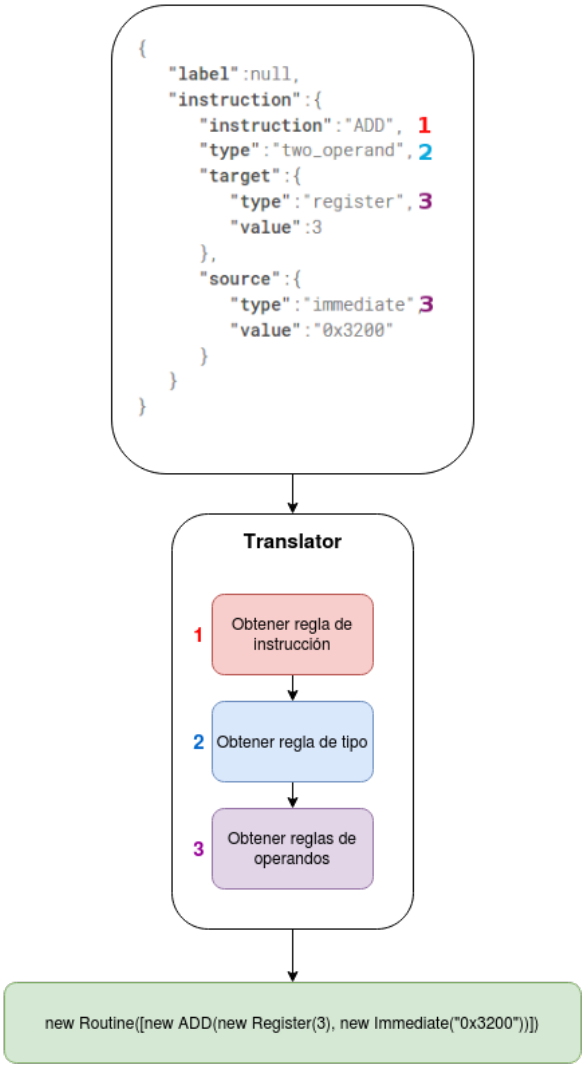


Figura 3.22: Proceso de traducción de la instrucción ADD R3 0x3200.

La vista en detalle del proceso realizado en el translator es el siguiente:

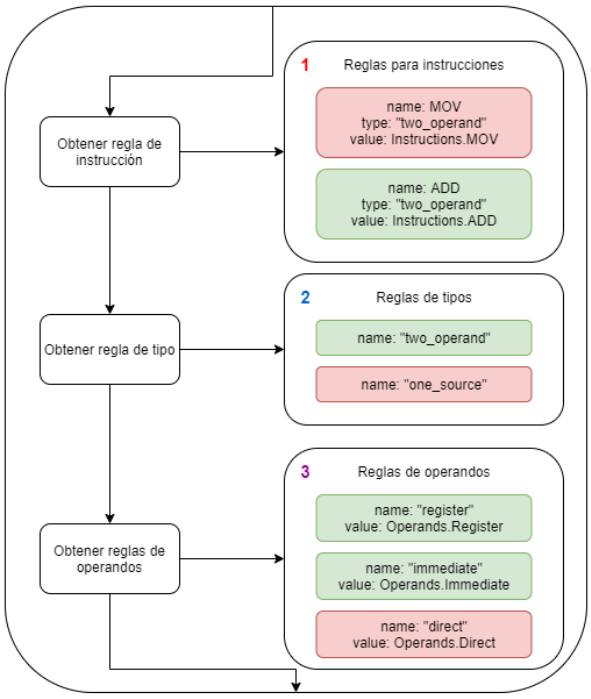


Figura 3.23: Proceso detallado de traducción de la instrucción ADD R3 0x3200. Resaltadas en verde las reglas de traducción asociadas a la misma.

3.2.4. Computer

Esta capa tiene como funcionalidades principales el manejo del ensamblado, que permite obtener el código binario de las instrucciones para luego poder ejecutarlo; y la ejecución que se encarga de decodificar el código previamente ensamblado y ejecutarlas en orden. Los detalles de estas etapas serán explicados a continuación.

Ensamblado

El ensamblado permite, utilizando las rutinas que genera el `Translator` (ver 3.2.3), obtener el código binario de las instrucciones para luego poder ejecutarlo. En el proceso de ensamblado intervienen principalmente 4 clases: `Computer`, `Instruction`, `State` y `Memory`. En esta etapa, la clase `Computer` se encarga de recibir las rutinas a ensamblar y delega en las instrucciones de cada rutina la obtención del código binario para entregarlo a la clase `State`. Esta última se encarga de guardar en memoria el código binario de cada una de las instrucciones utilizando la clase `Memory`.

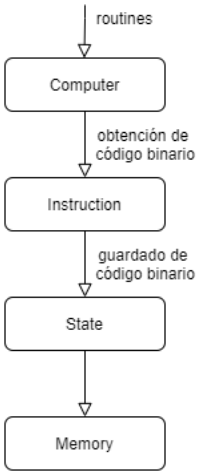


Figura 3.24: Proceso de ensamblado.

Como las instrucciones de una rutina pueden tener etiquetas o hacer uso de las mismas como operando, se definieron las clases `Label` y `LabelReference` para su manejo. Una rutina entonces, puede contener en su lista de instrucciones tanto elementos de tipo `Instruction` como de tipo `Label`. El objetivo de la etapa de ensamblado es convertir las etiquetas en direcciones de memoria ya que las etiquetas son texto y este no puede ser ensamblado en memoria. Para esto, el primer paso es calcular las direcciones de memoria donde se encuentren las definiciones de las etiquetas, es decir instancias de la clase `Label`. Luego, se deberán reemplazar los usos de esas etiquetas por operandos de tipo inmediato.

Para calcular las direcciones de memoria de las etiquetas se ven involucradas las clases `Instruction` y `Label` ya que son los elementos que puede tener una rutina y el PC, que permite conocer la dirección de memoria actual. La clase `Instruction`, al solo representar instrucciones y no etiquetas,

tiene la responsabilidad de avanzar el PC. Label en cambio, tiene la responsabilidad de guardar el PC actual y su identificador en una lista de etiquetas y delegar en su instrucción el avance del PC.

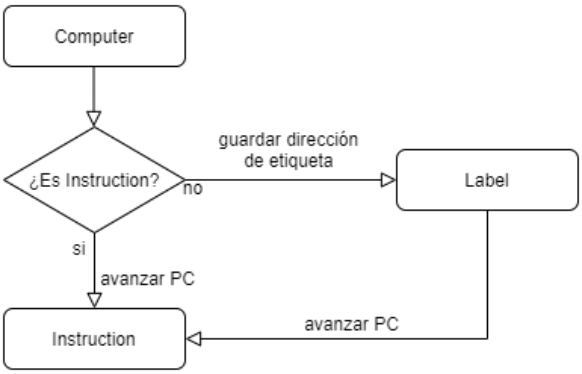


Figura 3.25: Cálculo de etiquetas.

Ejecución

La etapa de ejecución se encarga de decodificar el código previamente ensamblado, traducirlo a instrucciones Q y ejecutarlas en orden. De esta manera se da la noción de ejecución de un programa o rutina. Las instrucciones son decodificadas y ejecutadas de a una a la vez y esto permite que el programa o rutina pueda mutar a lo largo de su ejecución.

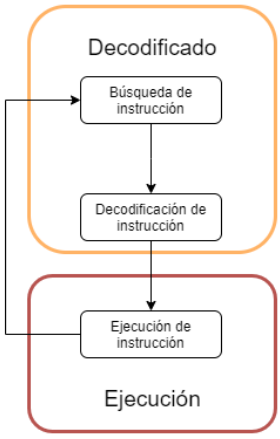


Figura 3.26: Diagrama de ejecución.

Para ello, `Computer` lee la dirección de memoria indicada por el PC, cuyo contenido será decodificado. El decodificado del código máquina queda delegado en `Instruction`, ya que cada instrucción puede determinar si un código binario le corresponde y sabe cómo decodificar sus operandos.

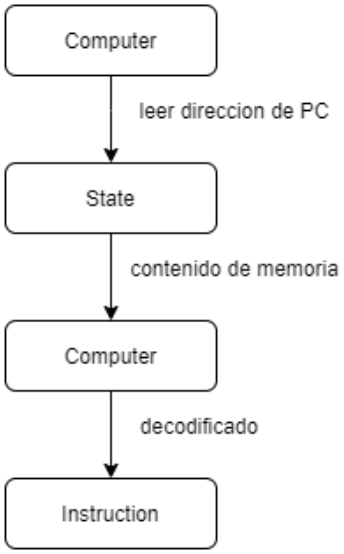


Figura 3.27: Proceso de decodificado.

Una vez que la instrucción fue decodificada junto con sus operandos, se procede a ejecutarla. La ejecución de una instrucción recibe el estado actual y lo modifica acorde a su efecto. Cada instrucción fue modelada con una clase, la cual conoce su efecto y lo aplica al ejecutarse. Este enfoque nos permite agregar nuevas instrucciones fácilmente siguiendo la interfaz brindada.

Estas modificaciones pueden incluir cambios en los flags, en el operando destino, en el PC o en la pila según la instrucción que se esté ejecutando, y es por esto que la ejecución recibe el estado, ya que este es el que se encarga realmente de manejar y almacenar estos valores.

Durante esta etapa pueden darse lecturas o escrituras de los operandos de origen y/o destino. Estos operandos pueden estar en modo inmediato, registro, directo, indirecto o registro indirecto. Para conocer o modificar sus valores, los operandos dependen de los valores almacenados en el estado, quien guarda los valores de registros y conoce a la memoria. Es por ello que se definieron cinco clases para representar cada una de estas posibilidades, las cuales conocen la forma de obtener valores del estado y guardarlos en él.

Como algunas instrucciones modifican los flags mientras que otras no lo hacen, esta tarea es delegada a cada instrucción, la cual conoce qué flags modifica y cómo lo hace.

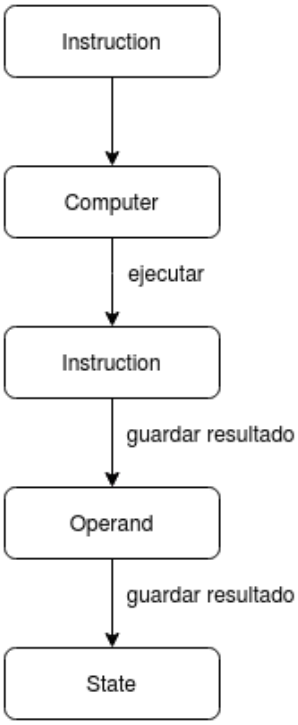


Figura 3.28: Proceso de ejecución.

Los detalles de implementación de la etapa de ensamblado y de ejecución se encuentran en la sección 6.2.3 del anexo.

La ejecución desde una mirada educativa

En la definición de Q no existe la diferenciación de programa y rutina, como así tampoco la idea de finalización de ejecución. La computadora está constantemente ejecutando alguna instrucción, ya sea una instrucción con efecto o sin efecto.

Dado que QLib es una librería que simula una arquitectura de computadoras enfocada en la enseñanza, permite la creación de programas y rutinas. Estos se diferencian entre sí, porque los programas no finalizan con una instrucción RET mientras que las rutinas si.

En la arquitectura Q no existe la idea de finalización de ejecución. Esto puede ser confuso para los/las estudiantes, ya que se esperaría que el programa tenga un inicio y un resultado final. Es por eso que consideramos que la librería debía proveer un método para identificar que la ejecución finalizó.

Para ello, analizamos las siguientes condiciones:

Al iniciar una ejecución las celdas que no hayan sido escritas, no tendrán valor. Por lo tanto el

PC eventualmente llegará a una celda no inicializada, esto puede darse por dos razones:

- Un salto o una llamada a rutina en una celda vacía.
- Incremento orgánico del PC. Se considera que el PC se incrementó orgánicamente cuando su incremento no es causado por un offset o por una asignación directa.

Analizando dichas razones:

- Si se llega a una celda no inicializada mediante un salto o una llamada a rutina, asumimos que es un error de programación. Por ejemplo:

  MOV R1 R2
  JMP 0x0fff
  En este caso se salta a la celda 0x0fff que no fue previamente inicializada.

  MOV R3 R0
  CALL 0xAFFF
  En este caso se llama a la rutina ensamblada en la celda 0xAFFF que no fue previamente inicializada.

- Si se llega a una celda no inicializada mediante un incremento orgánico del PC pueden darse dos casos:

  La pila no está vacía, por lo tanto asumimos un error de programación. Posiblemente por el olvido de un RET.

  MOV R3 R0
  CALL rutina
  MOV R0 R1

  [ASSEMBLE: 0x00ff]
  rutina: MOV R2 R3

  La pila está vacía, puede haber dos razones:

  La etapa de ejecución no es *FETCH* [2].

  MOV R3 [R0]
  En este caso, se lee el indirecto registro R0, pero la celda a la que apunta R0 no fue inicializada. Si bien esto no es un error, el resultado puede ser inesperado.

  La etapa de ejecución es FETCH, por lo tanto el programa finalizó.

  MOV R3 R0
  CALL rutina
  MOV R0 R1

  [ASSEMBLE: 0x00ff]
  rutina: MOV R2 R3
  RET

[2] FETCH es la etapa de la ejecución donde se busca en memoria la próxima instrucción a ejecutar.

Concluyendo, se asume la finalización de la ejecución cuando:

- La celda leída está no inicializada.
- El PC no fue modificado mediante un salto o una llamada a rutina.
- La pila se encuentra vacía.
- La etapa de ejecución es FETCH.

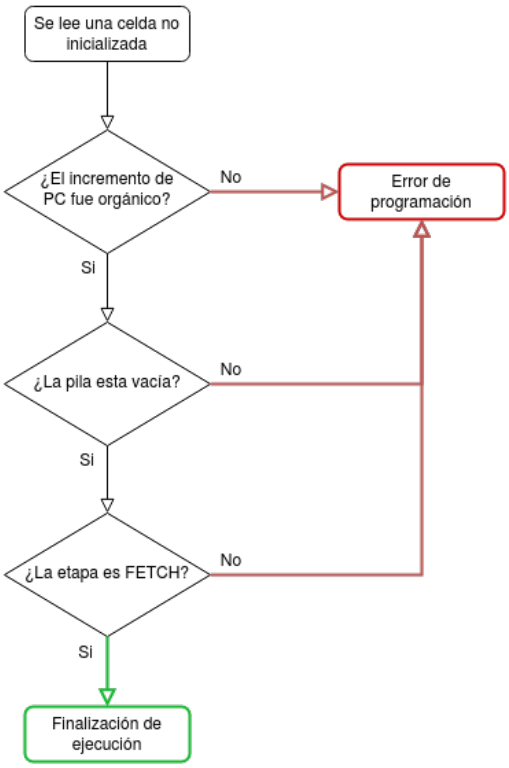


Figura 3.29: Condiciones de finalización.

Antes de definir un mecanismo de detección de finalización basado en la evaluación de condiciones, tuvimos en cuenta la posibilidad de agregar una instrucción de fin, donde por ejemplo, el usuario podría hacer uso de esta para definir cuándo o dónde el programa terminaría su ejecución.

Ejemplo de programa con instrucción de fin:

```
MOV R3 R0
ADD R0 R1
FIN
```

Con esta instrucción ya no sería necesario identificar una condición de finalización.

Optamos por definir un mecanismo de detección de finalización basado en la evaluación condiciones ya que esta facilita el entendimiento de un alumno o alumna de cuál es el resultado del programa que esté ejecutando abstrayendo la idea de que la computadora nunca deja de ejecutar. Por otro lado se evita agregar una instrucción que no se encuentra en Q.

QConfig

Existen distintos aspectos en QSim Web que pueden ser configurados con el objetivo de lograr que la herramienta tenga un enfoque más educativo y a su vez, para poder crear subconjuntos del lenguaje. QConfig es la clase encargada de guardar y modificar la configuración elegida. Existen las siguientes configuraciones:

- Cantidad de registros: Permite modificar la cantidad de registros entre 1 y 8 (R0-R7). Si se intenta usar un registro deshabilitado, la herramienta arrojará un error especificando lo sucedido.
- Comportamiento de la instrucción MUL: Permite elegir si se quiere que los 16 bits más significativos de una multiplicación se guarden en R7 o no. Esto es interesante ya que al principio resulta difícil de entender por los alumnos y que esté habilitado agregaría una complejidad esencial no buscada en los comienzos de la materia.
- Valor por defecto en celdas: Permite elegir si el valor por defecto de una celda será cero; un error, si se intenta acceder al valor de una celda no inicializada la herramienta lanzará una excepción; aleatorio, todas las celdas de memoria comenzarán con un valor aleatorio, ayudando así a simular que la memoria siempre puede contener valores inesperados.
- Valor por defecto en registros: Permite elegir si el valor por defecto de un registro será cero o aleatorio.
- Modos de direccionamiento: Permite elegir qué modos de direccionamiento estarán habilitados en la ejecución. Si se intenta usar un modo de direccionamiento deshabilitado, la herramienta lanzará una excepción indicando lo sucedido.
- Instrucciones: Permite elegir qué instrucciones estarán habilitadas en la ejecución. Si se intenta usar una instrucción deshabilitada, la herramienta lanzará una excepción indicando lo sucedido.

Por ejemplo, si solo se requiere de Q1, basta con habilitar únicamente las instrucciones MOV, MUL, ADD, SUB y DIV y los modos de direccionamiento Inmediato y Registro. O bien únicamente cargar un archivo pre-configurado con las opciones de Q1 como se describe en 3.1.

Capítulo 4

Devoluciones de estudiantes

Una vez desarrollada la herramienta quisimos ponerla a prueba con estudiantes reales. Para eso, gracias al apoyo de los docentes de Organización de Computadoras, compartimos la herramienta en las comisiones de dicha asignatura durante el segundo cuatrimestre del 2020.

Al finalizar la cursada fue enviada una encuesta para conocer las opiniones de los y las estudiantes acerca de la herramienta para poder saber qué tan útil resultó la misma y conocer ideas de posibles mejoras. La encuesta se realizó utilizando Google Forms [13] y era anónima aunque los alumnos tenían la posibilidad de dejar su email si estaban interesados en participar de futuros desarrollos relacionados con la herramienta.

Primero presentamos cuatro preguntas generales sobre la herramienta, uso y percepción por parte de los alumnos

1. ¿Qué tan útil te resultó a lo largo de la materia?
2. ¿Cómo calificaría el uso de la herramienta?
3. ¿Su uso fue intuitivo?
4. ¿Cuáles son las posibilidades de que la recomiendes a compañeros que vayan a cursar en un futuro?

En estos casos se daba una escala de uno a cinco, siendo uno "*poco*" y 5 "*mucho*", es decir, mayor puntaje significaba que el alumno estaba satisfecho con la herramienta.

Luego incluimos una serie de afirmaciones más en detalle sobre la herramienta. En este caso las respuestas iban de uno a cinco, siendo uno una valoración negativa: por ejemplo, los mensajes de errores eran poco claros, la ayuda era poco útil; y cinco una valoración positiva: los mensajes de error eran muy claros, la ayuda era muy útil.

Dichas afirmaciones fueron:

1. *Los mensajes de error fueron.*
2. *Las configuraciones de QSim Web fueron.*
3. *La ayuda de QSim Web fue.*

Finalmente, teníamos una serie de preguntas de texto libre para que los y las estudiantes pudieran dar sus devoluciones. Estas preguntas apuntaban a conocer su opinión acerca de posibles mejoras de la herramienta, nuevas funcionalidades, funcionalidades que más gustaron y mejoras en la visualización de resultados.

1. ¿Cómo mejorarías la herramienta? ¿Qué le agregarías?

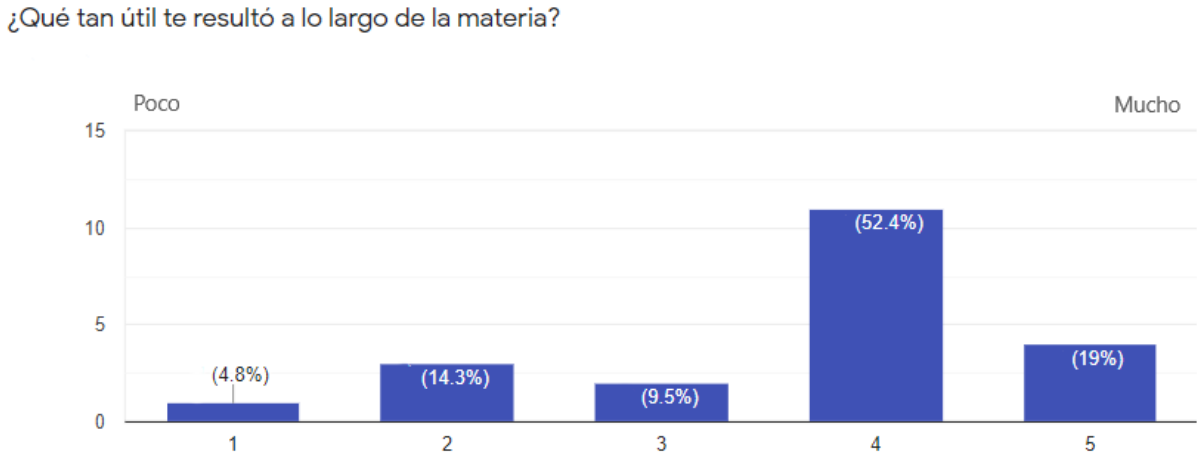
2. ¿Qué fue lo que más te gustó?
3. Al ejecutar un programa, se muestran los valores de registros, registros especiales y memoria. ¿Te parece claro? ¿Lo cambiarías de alguna forma?
4. ¿Te gustaría participar en futuros desarrollos de la herramienta (es decir programarla)? Si es así, dejanos tu mail.

La última pregunta responde a nuestro deseo de poder continuar el desarrollo de la herramienta ya que sabemos que quedan numerosas oportunidades de mejora (Ver 5.1).

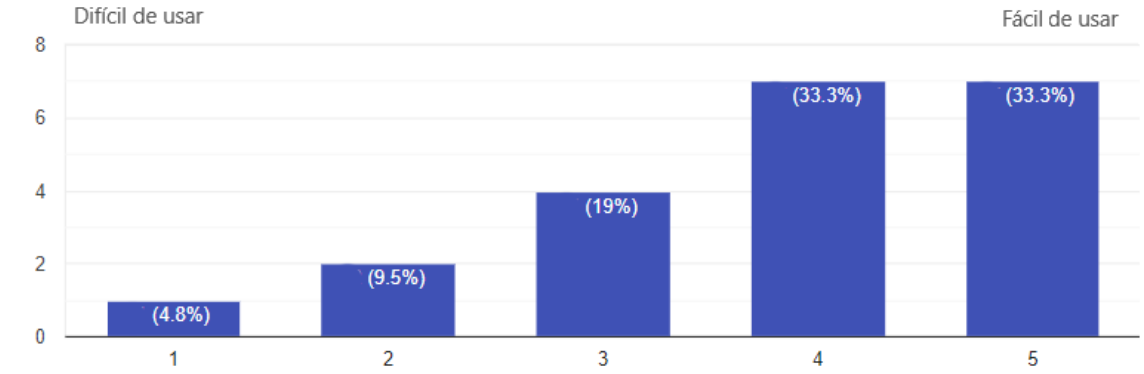
Resultados

Se recibieron y procesaron veintiún respuestas. A continuación se muestran los resultados. Los datos se encuentran disponibles en Google Drive [15].

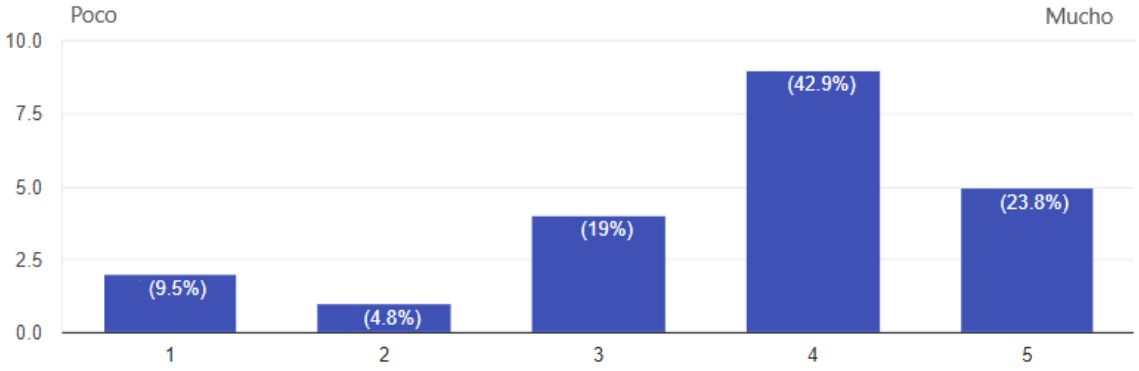
A continuación presentamos los resultados de la encuesta:



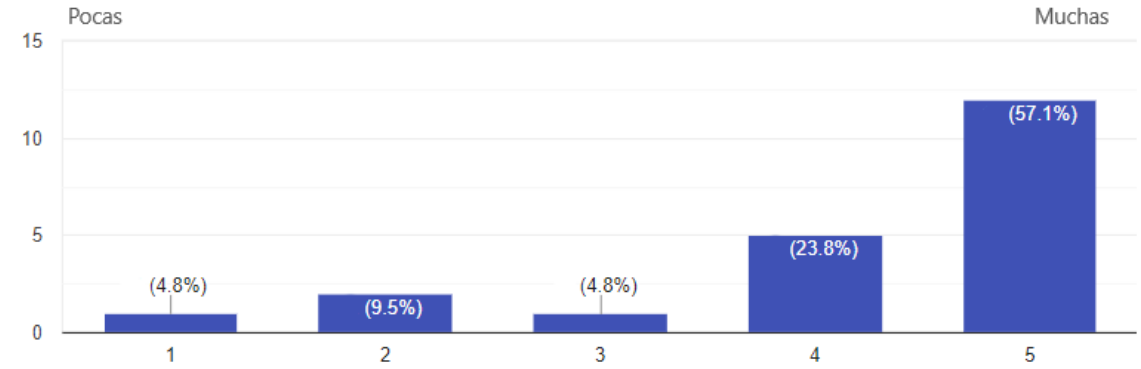
¿Cómo calificaría el uso de la herramienta?



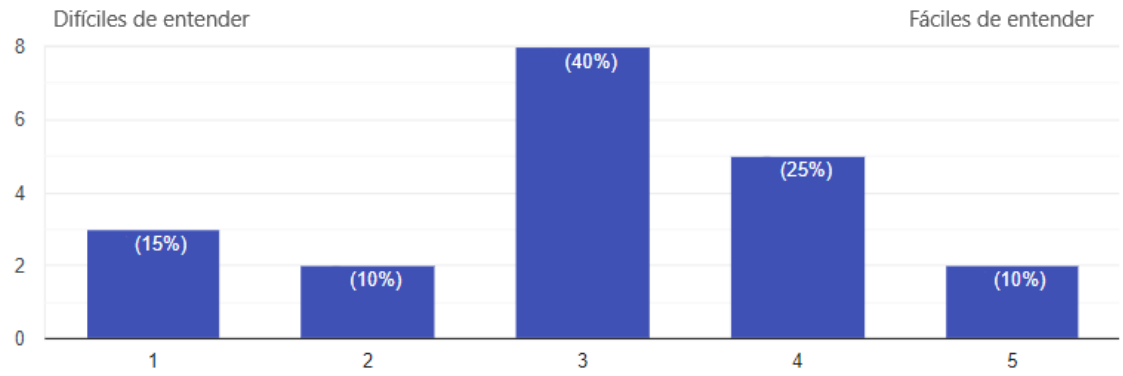
¿Su uso fue intuitivo?



¿Cuáles son las posibilidades de que la recomiendes a compañeros que vayan a cursar en un futuro?

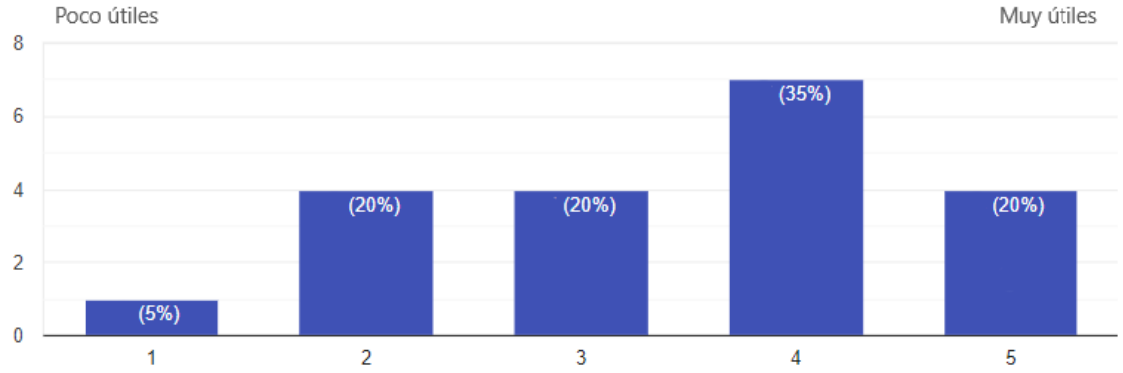


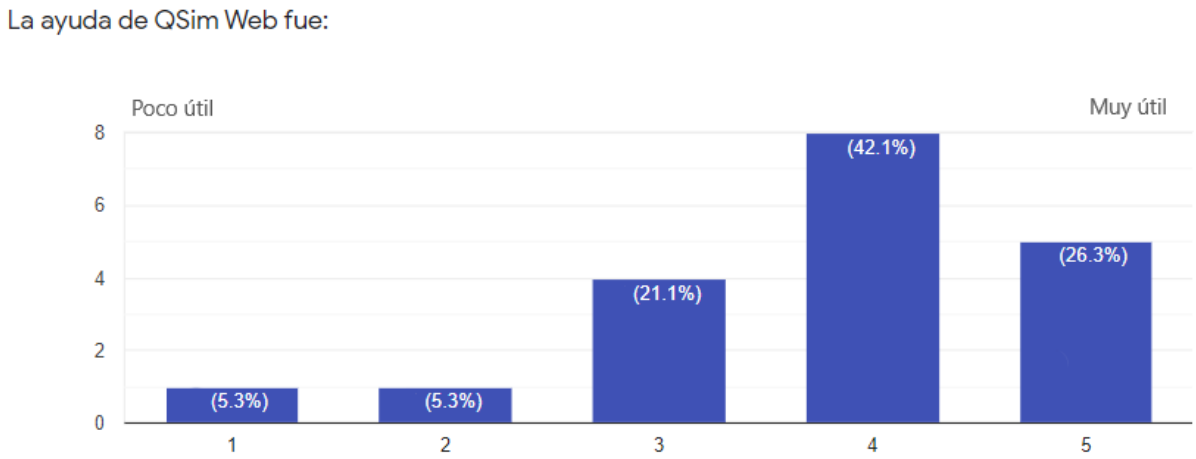
Los mensajes de error fueron:



Aquí podemos observar, más allá de que el resultado no fue negativo, que los mensajes de error no fueron suficientemente declarativos.

Las configuraciones de QSim Web fueron:





Análisis de los resultados

En general los resultados de la encuesta fueron muy satisfactorios.

A pesar de que al comienzo del semestre los alumnos usaron una versión incompleta de QSim Web se puede observar, que la herramienta cumplió con su cometido, les fue útil, fácil de usar e intuitiva.

La configurabilidad de la herramienta se destaca un punto fuerte. En este apartado fuimos sorprendidos, pues al estar la mayoría de las funcionalidades habilitadas por defecto, esperábamos que aquí el gráfico tendiera a "Poco útiles", ya que no la usarían. Si bien nos parece muy positivo el resultado, nos queda la duda de si la pregunta se entendió correctamente.

En general las respuestas fueron positivas y pudimos sacar varias conclusiones e ideas de nuevas funcionalidades para realizar a futuro, esto principalmente de las preguntas de respuesta tipo texto libre.

Es importante notar que varios comentarios no fueron realizados sobre la última versión de la herramienta, ya que la misma crecía con el pasar del cuatrimestre, y algunas de las funcionalidades pedidas ya se encontraban implementadas.

De los encuestados, nueve alumnos manifestaron interés en participar de futuros desarrollos de la herramienta, lo cual es muy alentador para el futuro de Qsim web.

Capítulo 5

Conclusiones

El resultado de este trabajo es la construcción de un simulador de la arquitectura Q que corre en un navegador y que puede ser usado por los y las estudiantes en el contexto de la materia Organización de Computadoras; permitiéndoles probar los programas que escriben, sin tener que simularlos en papel.

Desde el desarrollo de la herramienta notamos un aprendizaje en términos didácticos ya que habitualmente el software que realizamos en industria no es apuntado a la enseñanza. Sabemos que queda mucho por mejorar en este aspecto pero quienes hicimos este trabajo tenemos la intención de continuarlo para poder dar mejores funcionalidades tanto a estudiantes como a docentes.

Uno de los desafíos era la construcción de la idea de lenguaje, mediante el uso de gramáticas y parseo, aprendimos lo necesario para poder completar el trabajo y seguramente sigamos aprendiendo a lo largo de la licenciatura.

Construimos una librería pensando en evitar siempre el acoplamiento a nuestra interfaz de usuario e incluso a nuestro parser, creemos que puede ser realmente el puntapié inicial para que otros/as estudiantes de la carrera puedan intervenir y mejorar el trabajo realizado.

Mediante las evaluaciones realizadas por los alumnos notamos que fomentó la curiosidad en términos de programación, ya que realizaron programas por fuera de las prácticas y pusieron a prueba sus conocimientos.

5.1. Trabajos futuros

Encontramos durante el desarrollo del trabajo algunas ideas que pueden incluirse a futuro, que permitirán mejorar y extender las funcionalidades provistas hasta el momento. Las mismas no fueron implementadas ya sea porque escapaban de los objetivos de un trabajo de inserción profesional o por no haberlas considerado prioritarias para esta entrega. Catalogamos dichas mejoras basándonos en su origen: ideas propias originadas durante el desarrollo la herramienta, mejoras de accesibilidad originadas en el análisis realizado y mejoras propuestas por los y las estudiantes.

Mejoras propuestas por nosotros

- Manejo de instrucciones I/O. Actualmente la arquitectura *Q* no define estas instrucciones pero podría ser una extensión que aporte interactividad a los programas. Creemos que esta mejora podría ser buena porque agregaria a la herramienta la posibilidad de realizar programas que varíen su comportamiento de acuerdo a acciones del usuario, por ejemplo, se podrían hacer juegos sencillos que tengan respuesta visual en un "monitor"que se muestre en pantalla. Este desarrollo no es trivial, dado que por ejemplo no hay en Q mecanismos definidos para entrada y salida, por lo que debería resolverse eso primero.

- Permitir arquitecturas de computadoras distintas a Q, por ejemplo, poder variar el tamaño del bus de datos o direcciones. Esto es interesante ya que existen ejercicios de la práctica donde se trabajan con otras arquitecturas.
- Incluir algún método de revisión docente, como por ejemplo deep link o envío de mails.
- Instrucción NOOP, para poder terminar programas sin necesariamente realizar una acción como última instrucción. En muchos casos en las prácticas de la materia se incluyen programas que contienen alguna instrucción `JMP fin` donde la etiqueta *fin* no tiene ninguna instrucción definida. Este tipo de programas suele causar confusión en los y las estudiantes y este ítem apunta a resolverlo.
- Permitir la inicialización de registros y memoria antes de la ejecución.

Mejoras de accesibilidad

- Mejorar el editor en términos de accesibilidad: permitir salir del mismo usando la tecla tab, agregar una etiqueta para el mismo.
- El atributo "*role*" permite informar a los lectores de pantalla acerca de cambios realizados dinámicamente en la pantalla. Es por esto que surge como mejora agregar dicho atributo al resultado de ejecución y a los resultados en los registros para que los lectores de pantalla reconozcan el cambio.

Mejoras propuestas por estudiantes

- Mejorar la sección de ayuda. Por ejemplo agregando ejemplos de programas y links útiles.
- Mejorar descripción de errores de sintaxis. En este aspecto ya se tomaron medidas, ahora el editor muestra la línea que contiene un error de sintaxis y cuál es este pero aún creemos que se puede mejorar detectando posibles soluciones al problema.
- Mejorar resaltado de sintaxis.
- Visualización de resultados en *complemento a dos*.
- Dar la posibilidad de ejecutar instrucciones paso a paso automáticamente con un tiempo de diferencia entre cada una.
- Poder detener la ejecución.
- Resaltado de la línea de código que se está ejecutando.
- Poder modificar valores de registros, memoria y flags mientras se está ejecutando un programa.
- Poder seleccionar los distintos niveles de Q desde la web y no tener que cargar un archivo de configuración.
- Que la página se adapte mejor al ser usada desde un dispositivo con pantalla pequeña como un celular, dado que si bien tomamos medidas para mejorar este aspecto desde la primera versión, creemos que aún hay trabajo para mejorarla.

Capítulo 6

Anexos

6.1. Anexo: Detalles de la arquitectura *Q*

En este anexo se detallarán los aspectos técnicos de la arquitectura Q profundizando lo explicado en el capítulo 2

Q1

Las instrucciones de esta capa son:

Operación	**Cod Op**	**Efecto**	**Ejemplo de uso**
ADD	0010	$destino \leftarrow destino + origen$	ADD R7, 0x23FA
DIV	0111	$destino \leftarrow destino \% origen$	DIV R4, 0x23FA
MOV	0001	$destino \leftarrow origen$	MOV R4, R2
MUL	0000	$destino \leftarrow destino \times origen$	MUL R5, 0x0ABC
SUB	0011	$destino \leftarrow destino - origen$	SUB R2, [0x0ABC]

Todas estas instrucciones comparten el siguiente formato:

Cod Op	**Modo Destino**	**Modo Origen**	**Destino**	**Origen**
4 bits	6 bits	6 bits	16 bits	16 bits

Los modos de direccionamiento son:

Modo	**Codificación**
Inmediato	000000
Registro	100rrr [3]

Q2

El nuevo modo de direccionamiento es:

Modo	**Codificación**
Directo	001000

Q3

Las nuevas instrucciones de esta capa son:

Operación	**Cod Op**	**Efecto**	**Ejemplo de uso**
CALL	1011	$[SP] \leftarrow PC; SP \leftarrow SP - 1; PC \leftarrow origen$	CALL unaRutina
RET	1100	$SP \leftarrow SP + 1; PC \leftarrow [SP]$	RET

El formato de instrucción de *CALL* es el siguiente:

Cod Op	**Relleno**	**Modo Origen**	**Origen**
4 bits	6 bits	6 bits	16 bits

El formato de instrucción de *RET* es el siguiente:

Cod Op	**Relleno**
4 bits	12 bits

Q4

Las nuevas instrucciones de esta capa son:

El salto absoluto incondicional *JMP*:

Operación	**Cod Op**	**Efecto**	**Ejemplo de uso**
JMP	1010	$PC \leftarrow origen$	JMP unaRutina

Y los saltos condicionales relativos:

Modo	Codificación
Indirecto	011000
Indirecto Registro	110rrr

Operación	Cod Op	Descripción	Condición de salto	Ejemplo de uso
JE	0001	Igual / Cero	Z	JE etiqueta
JNE	1001	No igual	not Z	JNE etiqueta
JLE	0010	Menor o igual	Z or (N xor V)	JLE etiqueta
JG	1010	Mayor	not (Z or (N xor V))	JG etiqueta
JL	0011	Menor	N xor V	JL etiqueta
JGE	1011	Mayor o igual	not (N xor V)	JGE etiqueta
JLEU	0100	Menor o igual sin signo	C or Z	JLEU etiqueta
JGU	1100	Mayor sin signo	not (C or Z)	JGU etiqueta
JCS	0101	Carry / Menor sin signo	C	JCS etiqueta
JNEG	0110	Negativo	N	JNEG etiqueta
JVS	0111	Overflow	V	JVS etiqueta

El formato de instrucción de *JMP* es:

Cod Op	Relleno	Modo Origen	Origen
4 bits	6 bits	6 bits	16 bits

El formato de instrucción de los saltos condicionales es:

Prefijo	Cod Op	Desplazamiento
4 bits	4 bits	8 bits en complemento a dos

Q5

Los nuevos modos de direccionamiento son:

Las nuevas instrucciones de esta capa son:

Operación	Cod Op	Efecto	Ejemplo de uso
AND	0100	$destino \leftarrow destino \wedge origen$	AND R7, 0x23FA
OR	0101	$destino \leftarrow destino \vee origen$	OR R2, [0x0ABC]
NOT	1001	$destino \leftarrow \neg destino$	NOT R2

El formato de instrucción de NOT es:

Cod Op	Relleno	Modo Destino	Destino
4 bits	6 bits	6 bits	16 bits

6.2. Anexo: Detalles de implementación

6.2.1. Parser

Uno de los objetivos del `Parser` es validar sintácticamente el código que queremos ejecutar. Esta validación se realiza utilizando la librería Nearley [7], la cual provee una forma de construir gramáticas y utilizarlas para parsear texto. La construcción de dicha gramática depende del lenguaje que se quiera parsear y es aquí donde entra la sintaxis del lenguaje *Q*.

El lenguaje *Q* hace referencia a varios tipos de instrucciones basándose en el formato de las misma como se describe en el anexo 6.1. Uno de los tipos es el de las instrucciones con dos operandos. Este contiene a las instrucciones *MOV*, *ADD*, *AND*, *OR*, *SUB*, *MUL*, *DIV* y *CMP* y su regla de parseo es la siguiente:

```
1 InstructionT1 -> OperationT1 _ Operand ["," | _]:+ Operand
```

Este tipo de instrucciones son llamadas `InstructionT1` (pues en la materia organización de computadoras se las denomina instrucciones de tipo 1) y se componen de: `OperationT1` seguido de uno o más espacios, un `Operand` seguido de una coma y/o uno más espacios y luego el segundo `Operand`. `OperationT1` se define como cualquiera de las instrucciones nombradas anteriormente y su regla de parseo es la siguiente:

```
1   OperationT1 -> "MOV"i
2                | "ADD"i
3                | "AND"i
4                | "OR"i
5                | "SUB"i
6                | "MUL"i
7                | "DIV"i
8                | "CMP"i
```

`Operand` representa a cada uno de los modos de direccionamiento de la arquitectura *Q* y su regla de parseo es la siguiente:

```
1   Operand -> Register
2            | Immediate
3            | Direct
4            | Indirect
5            | IndirectRegister
```

Las reglas de parseo de los modos de direccionamiento son las siguientes:

```
1   Register         -> "R"i [0-7]
2   Immediate        -> ("0x"i) hexdigit hexdigit hexdigit hexdigit
3   Direct           -> "[" Immediate "]"
4   Indirect         -> "[" Direct "]"
5   IndirectRegister -> "[" Register "]"
6
7   hexdigit         -> [a-fA-F0-9]
```

Otro de los tipos es el de las instrucciones con un operando de origen. Este contiene a las instrucciones *CALL* y *JMP* y su regla de parseo es la siguiente:

```
InstructionT2 -> OperationT2 _ OperandLabel
```

Este tipo de instrucciones es llamado `InstructionT2` y se compone de: `OperationT2` seguido de uno o más espacios y un `OperandLabel`. `OperationT2` se refiere a las instrucciones *CALL* y *JMP* y su regla de parseo es la siguiente:

```
OperationT2 -> "CALL"i
             | "JMP"i
```

`OperandLabel` representa a cada uno de los modos de direccionamiento de la arquitectura *Q* y a las etiquetas, esto se consigue reutilizando la clase `Operand` y definiendo la clase `Label`. Su regla de parseo es la siguiente:

```
OperandLabel -> Operand | Label
Label        -> ([a-zA-Z0-9]):+
```

El resto de las reglas se encuentran definidas en el archivo de gramáticas ubicado en el repositorio [14].

QLib define una clase `Parser` la cual hace uso de *nearley* y de la gramática mencionada anteriormente. Esta provee la función `parse_code` que se encarga de devolver un listado de objetos JSON que representan al código introducido como se explica en la sección 3.2.2. Algunos ejemplos de JSON devueltos por el `Parser` son:

para la instrucción: `ciclo: CMP [0x0323] 0x0000` su JSON será el siguiente:

```
{
  label: {
    value: "ciclo",
    type: "label"
  },
  instruction: {
    instruction: "CMP",
    type: "two_operand",
    target: {
      type: "direct",
      value: "0x0323"
    },
    source: {
      type: "immediate",
      value: "0x0000"
    }
  }
}
```

para la instrucción: `multiplicar: CALL mulPorDos` su JSON será el siguiente:

```
{
  label: {
    value: "multiplicar",
    type: "label"
  },
  instruction: {
    instruction: "CALL",
    type: "one_source",
```

```
      source: {
        type: "label",
        value: "mulPorDos"
      }
    }
  }
```

para la instrucción: `JE fin` su JSON será el siguiente:

```
  {
    label: null,
    instruction: {
      instruction: "JE",
      type: "relative_jump",
      offset: {
        type: "label",
        value: "fin"
      }
    }
  }
```

para la instrucción: `fin: RET` su JSON será el siguiente:

```
  {
    label: {
      value: "fin",
      type: "label"
    },
    instruction: {
      instruction: "RET",
      type: "no_operands"
    }
  }
```

para la instrucción: `negar: NOT R1` su JSON será el siguiente:

```
  {
    label: {
      value: "negar",
      type: "label"
    },
    instruction: {
      instruction: "NOT",
      type: "one_target",
      target: {
        type: "register",
        value: 1
      }
    }
  }
```

En caso de error, puede lanzar dos tipos de errores que serán explicados a continuación.

InvalidInstructionError que es lanzado cuando la instrucción esta incompleta y se encontró un salto de linea, que representa el final de una instrucción. Por ejemplo:

```
  MOV R1    #La instruccion esta incompleta pero lo escrito hasta ahora es valido.
  RET
```

ParserError que es lanzado cuando el texto no puede representar una instrucción valida y no es una instrucción incompleta. Por ejemplo:

```
1   MOV R9   #MOV esta esperando algo de tipo Operand y R9 no es de ese tipo.
```

6.2.2. Translator

Los objetivos del `Translator` son instanciar las instrucciones de *QLib* y por otro lado, desacoplar al `Parser` de esta tarea.

Para instanciar dichas instrucciones recibe un listado de objetos JSON que representan a las instrucciones a instanciar. El formato de este JSON está ejemplificado en la figura 3.20 y definido en mayor detalle en la sección 6.2.1. Estas instrucciones estarán asociadas a instancias de la clase `Routine` y un listado de estas instancias será lo devuelto por el `Translator`.

Se definieron 3 tipos de reglas para poder instanciar las instrucciones: *reglas para instrucciones*, para *tipos de instrucciones* y *para operandos*. Cada uno de estos tipos hace referencia a una porción del JSON devuelto como se ve en la figura 6.1.

```
{
   "label":null,
   "instruction":{
      "instruction":"MOV", 1
      "type":"two_operand", 2
      "target":{
         "type":"register", 3
         "value":2
      },
      "source":{
         "type":"register", 3
         "value":5
      }
   }
}
```

Figura 6.1: JSON devuelto por el `Parser`.

Las reglas para instrucciones contienen a cada de una de las instrucciones de *Q* asociando su nombre a un tipo de instrucción y a la clase de *QLib* que la representa. Estas reglas hacen referencia mediante sus propiedades `name` y `type` al JSON mostrado en la figura 6.1, en los items 1 y 2 respectivamente.

Algunos ejemplos son:

```
1   {
2     name: 'mov',              //Nombre de la instruccion. Coincide con el nombre
          que se encuentra en el JSON.
3     type: 'two_operand',      //Instrucciones de dos operandos.
4     value: Instructions.MOV   //Clase de QLib.
5   },
```

```
6   {
7     name: 'call',
8     type: 'one_source',       //Instrucciones de un operando origen.
9     value: Instructions.CALL
10  },
11  {
12    name: 'ret',
13    type: 'no_operands',      //Instrucciones sin operandos.
14    value: Instructions.RET
15  },
16  {
17    name: 'not',
18    type: 'one_target',       //Instrucciones de un operando destino.
19    value: Instructions.NOT
20  },
21  {
22    name: 'je',
23    type: 'relative_jump',    //Instrucciones de salto relativo.
24    value: Instructions.JE
25  }
```

Las reglas para tipos de instrucciones definen a cada uno de los tipos de instrucciones de *Q*. Se relacionan con los tipos definidos en el ejemplo anterior y definen una forma de extraer los operandos del JSON que se recibió como entrada, esta función se llama `get_operands` y su valor de retorno es la lista de operandos. Estas reglas hacen referencia mediante su propiedad `name` al JSON mostrado en la figura 6.1, en el item 2.

Dichas reglas son:

```
1   {
2     name: "two_operand",
3     get_operands: (parsed_line, operands) => {
4       //return [target, source]
5     }
6   },
7   {
8     name: "one_source",
9     get_operands: (parsed_line, operands) => {
10      //return [source]
11    }
12  },
13  {
14    name: "no_operands",
15    get_operands: (parsed_line, operands) => {
16      //return []
17    }
18  },
19  {
20    name: "one_target",
21    get_operands: (parsed_line, operands) => {
22      //return [target]
23    }
24  },
25  {
26    name: "relative_jump",
27    get_operands: (parsed_line, operands) => {
28      //return [offset]
29    }
30  }
```

Las reglas para operandos definen a cada uno de los modo de direccionamiento de *Q*. Asocian el nombre que se encuentra en el JSON con la clase de *QLib* que representa dicho modo de direccionamiento. Estas reglas hacen referencia mediante su propiedad `name` al JSON mostrado en la figura 6.1, en el item 3.

```
{
  name: 'register',
  value: Register              //Clase de QLib
},
{
  name: 'immediate',
  value: Immediate
},
{
  name: 'direct',
  value: Direct
},
{
  name: 'indirect',
  value: Indirect
},
{
  name: 'indirect_register',
  value: IndirectRegister
}
```

Como las etiquetas pueden ser un operando, se definió una regla extra que contempla el uso de las mismas:

```
{
  name: 'label',
  value: LabelReference
}
```

Las instancias de la clase `Routine` contienen una dirección de memoria, desde donde son ensambladas y un listado de instrucciones. El `Translator` recibe en el método `translate_code` un listado de instrucciones con sus respectivas direcciones de memoria de inicio que luego servirán para instanciar las `Routine`.

Por ejemplo, si el método `translate_code` recibe el siguiente listado:

```
[
  {
    from_cell: '0x3232',
    instructions: [
      {
        label: null,
        instruction: {
          instruction: "MOV",
          type: "two_operand",
          target: {
            type: "register",
            value: 1
          },
          source: {
            type: "register",
            value: 2
          }
        }
      }
```

```
      ]
    }
  ]
```

su resultado será el siguiente:

```
[
  Routine('0x3232', [MOV(Register(1), Register(2))])
]
```

Ahora bien, puede que alguna de las instrucciones que reciba el `Translator` contenga una etiqueta, como es el caso siguiente:

```
[
  {
    from_cell: '0x3232',
    instructions: [
      {
        label: {
          type: "label",
          value: "ciclo" //una etiqueta ciclo
        },
        instruction: {
          instruction: "MOV",
          type: "two_operand",
          target: {
            type: "register",
            value: 1
          },
          source: {
            type: "register",
            value: 2
          }
        }
      }
    ]
  }
]
```

en este caso, el resultado de la función `translate_code` será el siguiente:

```
[
  Routine('0x3232', [Label("ciclo", MOV(Register(1), Register(2)))])
  //la instancia dentro de la lista es de la clase Label.
]
```

Es decir, el listado llamado de *instructions* dentro de `Routine` puede contener tanto instancias de `Instruction` como instancias de `Label`. Estas últimas servirán para el cálculo de etiquetas durante la etapa de ensamblado (descripto en 6.2.3). Las instancias de las clases `Instruction` y `Label` comparten interfaz respecto a los métodos de cálculo de etiquetas, es decir, son polimórficas respecto a los mismos. Esto permite manejar una única lista sin tener que diferenciar si se trata de instrucciones con o sin etiquetas.

6.2.3. Ensamblado

En este apartado se entrará en detalle sobre la implementación de la etapa de ensamblado, mencionada en la sección 3.2.4. Esta etapa comienza luego de la traducción del resultado del parseo a instrucciones de *QLib*. Recibirá por lo tanto, una lista de `Routine` como fue explicado en la sección 3.2.3. Se realiza llamando al método `load_many` de la clase `Computer`.

Este proceso se divide en 3 partes: cálculo de etiquetas, actualización de etiquetas y ensamblado de instrucciones.

El cálculo de etiquetas es el proceso mediante el cuál se le asigna a cada etiqueta una dirección de memoria.

Consideremos el siguiente programa ensamblado a partir de la celda 0x0002:

```
1           MOV R0 R1
2           JMP fin
3   fin:    RET
```

En este caso, la etiqueta *fin* será asociada al 0x0005 ya que *MOV R0 R1* ocupa una celda y *JMP fin* ocupa 2 celdas.

Este proceso es realizado por el método `calculate_labels` e involucra a las instancias de las clases `Label` e `Instruction`. Dicho método recibirá un listado de etiquetas y las direcciones de memoria donde están definidas. Si la instancia es de la clase `Instruction` el listado permanecerá igual. Si la instancia es de la clase `Label` se agregará un elemento a la lista con la dirección de memoria donde se encuentra dicha etiqueta. Para conocer dicha dirección de memoria se necesita de la clase `State`, quedando la firma del método `calculate_labels` de la siguiente forma:

```
1   class Instruction{
2     ...
3     calculate_label(labels, state){
4       //Se devuelve la lista tal como llego
5     }
6   }
7
8   class Label{
9     ...
10    calculate_label(labels, state){
11      //Se agrega una etiqueta a la lista labels
12    }
13  }
```

La actualización de etiquetas es el proceso por el cual se reemplazan las referencias a etiquetas por las celdas de memoria calculadas en el paso anterior. En nuestro caso, *JMP fin* contiene una referencia a una etiqueta, que será reemplazada por 0x0005 quedando *JMP 0x0005*.

El método encargado de actualizar las etiquetas es `update_labels` e involucra instancias de las clases `Label` e `Instruction`. Dicho método recibirá el listado de etiquetas y sus direcciones nombrado anteriormente. Si la instancia es de la clase `Label`, simplemente delegará la actualización en su instrucción. Si la instancia es de la clase `Instruction` pueden darse dos situaciones: la instrucción acepta operandos de tipo etiqueta o no lo hace. Si no acepta este tipo de operandos, simplemente continúa el proceso. Si acepta este tipo de operandos, como puede ser el caso de `CALL`,

deberá reemplazar su etiqueta por un inmediato cuyo valor es el valor guardado en la lista de etiquetas.

```
class MOV{ //MOV no acepta operandos de tipo LabelReference
  ...
  update_label(labels, state){
    //Continua el proceso
  }
}

class Label{
  ...
  update_label(labels, state){
    //Se delega en su instruccion
  }
}

class CALL{ //CALL acepta operandos de tipo LabelReference
  ...
  update_label(labels, state){
    //Reemplaza su LabelReference por un Immediate
  }
}
```

El último paso consiste en convertir cada una de las instrucciones en su respectivo código binario y guardarlo en memoria. Este proceso es delegado en cada una de las instrucciones, las cuales conocen sus códigos binarios y delegan en sus operandos la misma tarea. Para más detalles ver la sección 2.

Siguiendo con el ejemplo utilizado en la sección de **El cálculo de etiquetas**, la traducción de dicho programa será la siguiente:

```
Routine(from_cell: 0x0002, instructions: [
  MOV(Register(0), Register(1)),
  JMP(LabelReference("fin")),
  Label(value: "fin", instruction: RET())
])
```

Es decir, la llamada a `load_many` recibirá como argumento la traducción anterior e iterará por todas las instrucciones consiguiendo sus respectivos códigos binarios. Luego guardará en memoria dichos binarios concluyendo el proceso.

6.3. Anexo: Implementación de una interfaz grafica usando QLib

En este anexo se detallan los métodos expuestos por QLib que permitirán la construcción de interfaces de usuario.

Una interfaz de usuario minimalista debería contar con, al menos, algún mecanismo para que el usuario pueda proveer código Q, alguna forma de disparar el ensamblado y ejecución de dicho código, y la capacidad de mostrar los resultados.

El texto que representa al código Q, deberá ser parseado para poder ser utilizado por la librería.

Para esto se cuenta con la función `parse_code` que expone la librería mediante su `Parser`. Este método recibe el texto a parsear y devuelve una lista JSON que servirán como entrada para el `Translator`.

Su forma de utilización es la siguiente:

```
1 import parser from './qweb/language/parser'
2 ...
3 const code_to_parse = get_written_code() //obtener el codigo desde la UI
4 const parsed_code = parser.parse_code(code_to_parse)
```

En la variable *parsed_code* se cargará la lista JSON mencionada anteriormente.

La función `parse_code` puede lanzar 2 tipos de errores: `InvalidInstructionError` y `ParserError`. Es por esto que el código debe poder capturar dichos errores para manejarlos apropiadamente:

```
1   import parser from './qweb/language/parser'
2   ...
3   const code_to_parse = get_written_code() //obtener el codigo desde la UI
4   try{
5     const parsed_code = parser.parse_code(code_to_parse)
6   }
7   catch(error){
8     //manejo de errores de parseo
9   }
```

En la línea 8 deberán manejarse los errores de parseo. Las propiedades de cada clase de error pueden consultarse en el apartado 6.2.1 donde se detallan sus implementaciones.

Una vez que tenemos el código parseado, debe ser traducido, para poder tener las instancias de instrucciones de QLib. Este proceso se realiza utilizando la función `translate_code` de la clase `Translator`. El mismo recibe una lista JSON y devuelve una lista de `Routine` que servirán como entrada para la clase `Computer`. Siguiendo con el ejemplo anterior, el código para traducir el resultado del parseo es el siguiente:

```
1   import translator from './qweb/language/translator'
2   ...
3   const routines = translator.translate_code(parsed_code)
```

En la variable *routines* se cargará la lista de `Routine` mencionada anteriormente.

Luego de traducir el código, debe cargarse en la `Computer`, es decir, ensamblarlo para su posterior ejecución. Esta acción es realizada mediante el método `load_many` de la clase `Computer`.

```
1   import computer from './qweb/qcomputer'
2   ...
3   computer.load_many(routines)
```

El método `load_many` puede lanzar 2 tipos de errores: `UndefinedLabel` y `ImmediateAsTarget`. Es por esto que el código debe poder capturar dichos errores para manejarlos apropiadamente:

```
1  import computer from './qweb/qcomputer'
2  ...
3  try{
4    computer.load_many(routines)
5  }
6  catch(error){
7    //manejo de errores de carga
8  }
```

En la línea 7 deberán manejarse los errores de carga. Las propiedades de cada clase de error pueden consultarse en la sección 6.2.3 donde se detallan sus implementaciones.

Luego de haber ensamblado el código en la `Computer` podemos ejecutarlo utilizando la función `execute` de la clase `Computer`. Esta función devuelve una lista de `Action` que detallan lo sucedido durante la ejecución en términos de la arquitectura: qué celdas o registros se accedieron, si hubo acceso a pila y más.

```
1  import computer from './qweb/qcomputer'
2  ...
3  computer.execute()
```

La función `execute` puede lanzar varios tipos de errores. Es por esto que el código debe poder capturar dichos errores para manejarlos apropiadamente:

```
1  import computer from './qweb/qcomputer'
2  ...
3  try{
4    computer.execute()
5  }
6  catch(error){
7    //manejo de errores de ejecucion
8  }
```

En la línea 7 deberán manejarse los errores de ejecución. Las propiedades de cada clase de error, así como de la clase `Action` pueden consultarse en la sección 6.2.3 donde se detallan sus implementaciones.

Para acceder a los resultados de la ejecución debemos hacerlo mediante el estado de `Computer`:

```
1  import computer from './qweb/qcomputer'
2  ...
3  computer.get_memory() //valores en la memoria
4  computer.state.registers //registros de uso general
5  computer.state.PC //valor del registro especial PC
6  computer.state.SP //valor del registro especial SP
7  computer.state.IR //valor del registro especial IR
8  computer.state.C //valor del flag C
9  computer.state.N //valor del flag N
10 computer.state.V //valor del flag V
11 computer.state.Z //valor del flag Z
```

Concluyendo, el código del ejemplo completo es el siguiente:

```
1  import parser from './qweb/language/parser'
2  import translator from './qweb/language/translator'
3  import computer from './qweb/qcomputer'
```

```
4
5   let parsed_code
6
7   const code_to_parse = get_written_code() //obtener el codigo desde la UI
8   try{
9     parsed_code = parser.parse_code(code_to_parse)
10  }
11  catch(error){
12    //manejo de errores de parseo
13  }
14
15  const routines = translator.translate_code(parsed_code)
16
17  try{
18    computer.load_many(routines)
19  }
20  catch(error){
21    //manejo de errores de carga
22  }
23
24  try{
25    computer.execute()
26  }
27  catch(error){
28    //manejo de errores de ejecucion
29  }
30
31  computer.get_memory() //valores en la memoria
32  computer.state.registers //registros de uso general
33  computer.state.PC //valor del registro especial PC
34  computer.state.SP //valor del registro especial SP
35  computer.state.IR //valor del registro especial IR
36  computer.state.C //valor del flag C
37  computer.state.N //valor del flag N
38  computer.state.V //valor del flag V
39  computer.state.Z //valor del flag Z
```

6.3.1. Utilizando un parser distinto

QLib permite su utilización con un parser ajeno a la librería, las bondades de esta implementación están descriptas en la sección 3.2.2.

Si se desea utilizar un parser diferente al de la librería, esto podría hacerse siguiendo la explicación anterior sin incluir la parte del parseo. Nuestro nuevo parser debe cumplir con la interfaz esperada por el `Translator`. Los detalles de dicha interfaz están en el apartado 6.2.1.

En este caso, el codigo una vez que obtenemos la entrada para el Translator, el código es igual al ejemplo de como construir una interfaz de usuario . La principal diferencia es como a partir del código se obtiene la entrada para el translator.

```
1   import parser from './our/magic/parser' //se utiliza un parser distinto al de
        QLib
2   import translator from './qweb/language/translator'
3   import computer from './qweb/qcomputer'
4
5   let parsed_code
6
7   const code_to_parse = get_written_code() //obtener el codigo desde la UI
```

```
8   try{
9     parsed_code = parser.do_magic(code_to_parse) //la funcion utilizada es la
          especifica de nuestro nuevo parser.
10  }
11  catch(error){
12    //manejo de errores de parseo
13  }
14
15  const routines = translator.translate_code(parsed_code)
16
17  try{
18    computer.load_many(routines)
19  }
20  catch(error){
21    //manejo de errores de carga
22  }
23
24  try{
25    computer.execute()
26    ...
27  }
28  catch(error){
29    //manejo de errores de ejecucion
30  }
```

6.4. Anexo: Implementación de un parser con instrucciones en español

La separación entre el `Parser` y el `Translator` explicada en la sección 3.2 tiene como uno de sus objetivos desacoplar la etapa de parseo de la etapa de instanciación de las instrucciones de *QLib*.

Otro de sus objetivos es simplificar los cambios en el lenguaje, por ejemplo, para traducirlo a castellano. En este anexo se detallarán los cambios necesarios para poder traducir el lenguaje Q a una versión en castellano.

Las partes involucradas en estos cambios son el `Parser`, específicamente en la gramática y `QConfig` explicada en la sección 3.2.4.

A continuación implementaremos un parser rudimentario que permitirá parsear una instrucción con la forma MOVER R_n, R_m siendo `n` y `m` números entre 0 y 7.

Llamaremos a la clase que represente a nuestro nuevo parser `SimpleParser`, la cuál tendrá un método `parse` que recibirá el texto a parsear y devolverá los objetos que requiere el `Translator` de *QLib* (ver 3.2.2).

```
1   class SimpleParser {
2     constructor() {
3       this.grammar = //Definicion de la gramatica
4     }
5     parse(code) {
6       //Logica de parseo
7     }
8   }
```

Como queremos parsear texto de la forma MOVER R_n, R_m, utilizaremos la expresión regular

$$/(mover) + (R[0-7])(,|?) + (R[0-7])/$$

```
class SimpleParser {
  constructor() {
    this.grammar = new RegExp(/(mover) +(R[0-7])(,| ?)+(R[0-7])/, 'gmi')
  }
  parse(code) {
    //Logica de parseo
  }
}
```

Si el código coincide con nuestra expresión regular, entonces devolvemos el JSON correspondiente a la instrucción MOV R_n, R_m. Caso contrario, nuestro parser lanza un error.

Como la instrucción tiene la forma MOV R_n, R_m, el parser debe devolver el JSON siguiente:

```
{
  label: null,
  instruction: {
    instruction: "MOV",
    type: "two_operand",
    target: {
      type: "register",
      value: n //el valor de n
    },
    source: {
      type: "register",
      value: m //el valor de m
    },
  }
}
```

Por lo tanto, el método `parse` quedará definido de la siguiente forma:

```
class SimpleParser {
constructor() {
  this.grammar = new RegExp(/(mover) +(R[0-7])(,| ?)+(R[0-7])/, 'gmi')
}
parse(code) {
  if (code.match(this.grammar)) { //Si coincide con la gramatica devuelvo el JSON
    return {
      label: null,
      instruction: {
        instruction: "MOV",
        type: "two_operand",
        target: {
          type: "register",
          value: this._get_value_of_register(code, 1)
        },
        source: {
          type: "register",
          value: this._get_value_of_register(code, 2)
        },
      }
    }
  }
  throw new Error("I'm a very simple parser :(") //Caso contrario lanzo un error.
}
```

```
25
26    _get_value_of_register(code, operandNumber) {
27      return code.split(/r/)[operandNumber + 1].charAt(0)
28    }
29  }
```

La forma de utilizarlo es la siguiente:

```
1
2  const parser = new SimpleParser()
3
4  console.log(parser.parse('mover r2 r5'))
5  /*
6  Imprime:
7  {
8    label: null,
9    instruction: {
10     instruction: "MOV",
11     type: "two_operand",
12     target: {
13       type: "register",
14       value: 2
15     },
16     source: {
17       type: "register",
18       value: 5
19     },
20   }
21 }
22 */
```

El siguiente componente a modificar es `QConfig`. La configuración contiene el listado de instrucciones disponibles para *QLib* y *su valor legible*. Un ejemplo de instrucción en `QConfig` es el siguiente:

```
1    {
2     "name":"MOV",
3     "enabled":true,
4     "display_name":"MOV"
5    }
```

Si modificamos el atributo *display_name* lograríamos que *QLib* lo utilice como *valor legible*, quedando:

```
1    {
2     "name":"MOV",
3     "enabled":true,
4     "display_name":"MOVER"
5    }
```

Conclusión: Este anexo muestra cómo es posible utilizar otro parser y traducir el lenguaje al castellano. Nos parece que no son demasiados cambios a implementar y *QLib* muestra flexibilidad para los mismos.

Referencias

[1] Susana Rosito Sayago y Tatiana Molinari, *Qsim: Simulador de Arquitecturas Q*, Universidad Nacional de Quilmes.

[2] *Material UI*, `https://material-ui.com`

[3] *react ace*, `https://github.com/securingsincity/react-ace`

[4] *Web Accessibility Evaluation Tool*, `https://wave.webaim.org/`

[5] *Web Accessibility Initiative*, `https://www.w3.org/WAI/standards-guidelines/`

[6] *World Wide Web Consortium*, `https://www.w3.org/`

[7] Nearley, `https://nearley.js.org/docs/how-to-grammar-good`

[8] mocha.js, `https://mochajs.org/`

[9] cypress.io, `https://www.cypress.io/`

[10] Richard E. Pattis, *EBNF: A Notation to Describe Syntax*, University of California, Irvine.

[11] *Especificación - Q*, `http://orga.blog.unq.edu.ar/wp-content/uploads/sites/5/215/08/Especificacion-Q.pdf0`

[12] Heroku, `https://www.heroku.com/`

[13] Google Forms, `https://docs.google.com/forms/`

[14] Repositorio de código, `https://gitlab.com/qsim-web-project/qsim-web`

[15] Resultados de la encuesta de QSim web, `https://docs.google.com/spreadsheets/d/1Z4wlrnGznYcn94HLcA7vu`

[16] Google - Web Fundamentals, `https://developers.google.com/web/fundamentals/accessibility?hl=es`