# QWeb

Francisco Perez Ramos - Pablo Nicolás Pissi

## Introducción

QWeb nace con la idea de poder simular el lenguaje de programación Q, utilizado en la Universidad Nacional de Quilmes. Este simulador es web para evitar todas las complejidades accidentales inherentes a la instalación de herramientas para su uso, como pueden ser compiladores o herramientas específicas de lenguajes. El código está escrito enteramente en Javascript y es ejecutado en el navegador por lo que no requiere de servidores una vez que este fue cargado en la computadora cliente.

## Descripción

En esta sección se detallan las diferentes partes que componen el simulador y las interacciones entre ellas.

### Frontend web

Se trata del componente desde el cual se puede ejecutar código Q, escrito por el usuario.

Permite escribir en varias líneas el programa que se debe ejecutar. Luego este programa se ejecutará proveyendo una devolución acorde al resultado del programa, si este no tiene errores de sintaxis, o mostrará los errores que la ejecución del programa haya devuelto. Permite 3 modos de ejecución:

1. Ejecución total
2. Ejecución de a una instrucción
3. Ejecución de a una instrucción con detalle

#### Ejecución total

En este modo, la totalidad del programa escrito es ejecutado, es decir, todas las instrucciones que componen al programa son ejecutadas secuencialmente o de acuerdo a la especificación de la misma. Los valores de registros de uso general, registros especiales y memoria son plasmados en la interfaz de usuario de acuerdo a la totalidad del programa.

#### Ejecución de a una instrucción

En este modo, se ejecuta una instrucción por cada vez que el usuario lo indique. Los valores mostrados representan el estado de la computadora hasta la instrucción que se haya ejecutado, siendo este acumulado para la siguiente instrucción. Este modo permite ver los efectos de las instrucciones, pudiendo analizarse de a una.

#### Ejecución de a una instrucción con detalle

Como QWeb es una herramienta pensada para la enseñanza, este modo permite ver separadas las etapas de Fetch, Decode, Execute por cada instrucción. Es decir, el usuario puede ver los efectos de realizar la búsqueda de la instrucción, luego los efectos de la decodificación y posteriormente de la ejecución para cada una de las instrucciones que componen el programa.

Para obtener los efectos, la ejecución detallada de la clase Computer devuelve un conjunto de efectos que se va llenando a medida que realiza la ejecución. El formato propuesto para estos efectos es el especificado en la clase Action dentro de la libreria qweb:

class Action(name, data)

A su vez, el frontend web define 7 tipos de Acciones:

- Assemble
- ReadMemory
- WriteMemory

- ReadRegister
- WriteRegister
- WriteStack
- AssignPC

El tipo Assemble y WriteMemory son en definitiva, el mismo tipo pero en distinto momento del ciclo de instrucción: durante el ensamblado y durante la ejecución respectivamente pero con fines pedagógicos se separaron. Esta separación es configurable mediante la clave *actions_mode* presente en el archivo config.json.

El tipo WriteStack y WriteMemory son en definitiva, el mismo tipo pero con distintos niveles de detalle: si el *actions_mode* es ultra-verbose se mostraran las escrituras en pila de manera diferenciada.

Los valores posibles para esta configuración son:

- **normal**, en este modo las acciones. Assemble y WriteMemory son un mismo tipo de efecto.
- **verbose**, en este modo Assemble y WriteMemory son diferentes tipos de efecto.
- **ultra-verbose**, en este modo Assemble y WriteMemory son diferentes tipos de efecto. Además se diferencian las escrituras a memoria si se tratan de celdas reservadas a la pila y se muestran los cambios en el PC realizados por CALL, RET y JMP.

Para tener un manejo más prolijo del *actions_mode* se creó la clase ActionMode, la cual relaciona los valores "normal", "verbose", "ultra-verbose" con su respectiva clase.

Las subclases de ActionMode son:

- ActionModeNormal
- ActionModeVerbose
- ActionModeUltraVerbose

Cada una de ellas define una lista de tipos de acción válidos mediante la función *valid_action_types*. Además, como hay tipos de acciones que varian su representacion en la UI según el modo activado, se definieron relaciones entre los tipos cuando es necesario cambiar la representación. Por ejemplo, una acción de tipo *assemble* se mapea a una acción de tipo *write_memory* si el *actions_mode* es "normal". Estas relaciones se definen en la función *mappings*.

Es decir, el mecanismo para obtener las acciones a mostrarse en pantalla sería el siguiente:

- Obtengo las acciones de la ejecución.
- Mapeo las acciones a sus posibles relaciones basadas en el ActionMode. Si no hay relación valida la acción queda igual.
- Filtro las acciones según los tipos de acciones válidos basados en el ActionMode.

### Carga de archivos

Para facilitar y fomentar la división en rutinas, el frontend cuenta con la posibilidad de manejar varios archivos de código simultaneamente. Estos archivos se pueden cargar desde la computadora del usuario y su contenido se visualiza en la pestaña correspondiente al mismo. Cuando un archivo se agrega, es incluído en la ejecución como un programa único. Permite además cerrar pestañas de archivos para evitar que sean incluidos en la ejecución.

Cuenta con 3 validaciones a la hora de cargar archivos:

- Que un archivo con el mismo nombre no esté cargado actualmente.
- Que el tamaño del archivo sea menor a 1Mb.
- Que el tipo de archivo sea txt.

## Parser

El parser es el componente encargado de la traducción del texto escrito en el Frontend web, en objetos que representan las instrucciones del lenguaje. Su tarea es determinar qué cadenas de texto pueden ser convertidas en instrucciones y que cadenas no. Para hacerlo utlizamos la librería nearley.js la cual permite escribir la gramática en un archivo con extensión .ne y compilarla a javascript para poder ser usada como un módulo dentro del navegador. [1]

Por ejemplo, la siguiente cadena de texto:

```
ciclo:  CMP R1, 0x0000
```

Que representa a una etiqueta ciclo apuntando a la instrucción CMP con los operandos R1 y 0x0000, será parseada de la siguiente forma:

```
{
  "label": {
        "type": "label",
        "value": "ciclo"
  },
  "instruction": {
    "instruction": "CMP",
    "type": "two_operand",
    "target": {
        "type": "register",
        "value": "1"
    },
    "source": {
        "type": "immediate",
        "value": "0x0000"
    }
  }
}
```

## Translator

Una vez que el Parser convierte el texto en objetos de javascript, el Translator se encargará de instanciar las instrucciones propiamente del lenguaje, diferenciando etiquetas, instrucciones, subrutinas. Para ello, cuenta con:

1. Un mapeo entre los tipos de operandos del parser con los modos de direccionamiento de la arquitectura Q.
2. Un mapeo entre las instrucciones devueltas por el parser con las instrucciones de la arquitectura Q.
3. Un mapeo entre los tipos de instrucciones y la cantidad de parametros y los tipos de los mismos.

En resumen, convertirá el json anteriormente descripto, en un objeto de la siguiente forma:

```
new Label("ciclo", new CMP(new Register(1), new Immediate("0x0000")))
```

Estos objetos, agrupados en rutinas son comprendidos por la representación de la computadora Q y por lo tanto pueden ser ejecutados.

## Computer

Es la clase principal que representa el funcionamiento de la arquitectura Q. Sus tareas principales son

- Analizar las etiquetas presentes en el programa y convertirlas a direcciones de memoria o saltos relativos.
- Ensamblar el resultado del analisis de etiquetas
- Ejecutar las instrucciones en memoria hasta que encuetre un fin de ejecución, cambiando su estado.

[1] Se tuvo que modificar la librería para poder usarla con heroku, se subieron esas modificaciones a un repositorio propio y se hizo que la dependencia se instale desde este repositorio en lugar de utilizar el repositorio de npm.

### Instrucciones

Representas las instrucciones presentes en la arquitectura Q, conteniendo cada clase la definición de cómo se modifica el estado basandose en sus operandos, cuando aplica. Ellas son:

`MUL, MOV, ADD, SUB, CMP, DIV, CALL, RET, JE, JNE, JLE, JG, JL, JGE, JLEU, JGU, JCS, JNEG, JVS, JMP, AND, OR, NOT`

Cada una de estas instrucciones está representada por una clase de javascript. A su vez, cada instrucción sabe como ensamblarse y desensamblarse.

### Estado

Es la clase que contiene todos los valores que son modificados por la ejecución. En ella se encuentran los registros, la memoria y los flags. Provee metodos para cambiar el estado que son accedidos por los operandos de las instrucciones.

### Operandos

Son los encargados de definir cómo se modifica el estado:

- Modo Inmediato: no modifica el estado ya que representa a un valor literal.
- Registro: modifica el registro asociado con su valor, dentro del estado.
- Directo: modifica la dirección de memoria asociada a su valor, dentro del estado.
- Indirecto Registro: modifica la dirección de memoria apuntada por el registro asociado con su valor, dentro del estado.
- Indirecto: modifica la dirección de memoria apuntada por la dirección de memoria asociada a su valor, dentro del estado.

Cada uno de estos operandos está representado por una clase de javascript. A su vez, cada operando sabe como ensamblarse y desensamblarse.