Simulador de Arquitecturas Q

Susana Rosito Tatiana Molinari

11 de noviembre de 2013

Índice

Resumen

Una de las primeras asignaturas que debe recorrer un estudiante de la Tecnicatura Universitaria en Programación Informática es **Organización de Computadoras**. En esta materia los estudiantes descubren los componentes funcionales que conforman un sistema de cómputos, con el fin de comprender un modelo de ejecución de programas que está presente hoy en día en la mayoría de las computadoras personales.
Este trabajo es el desarrollo de una herramienta que permite simular la ejecución de programas en una arquitectura teórica desarrollada por el equipo docente de la materia.

Parte I
Contexto

1. Sobre la materia Organización de Computadoras

Si bien los conceptos fundamentales de la ejecución de programas son independientes de las arquitecturas de computadoras comerciales, es conveniente explicar los mismos dando un marco específico. Por otro lado, las diferentes arquitecturas que subyacen los numerosos modelos disponibles en el mercado, incluyen un basto conjunto de herramientas y recursos de lenguaje para el control del funcionamiento de una computadora, pero que agregan complejidad innecesaria a la comprensión de los conceptos funcionales y la didáctica de la materia.

La propuesta de la asignatura Organización de Computadoras es utilizar la arquitectura QArq, una arquitectura *assembly-like* teórica (esto es, que no existe una computadora real que la implemente) basada en el modelo de ejecución de Von Neumann y cuya característica principal es la de ser minimalista y presentarse en 'capas'. Este enfoque permite ir incorporando los conceptos de manera gradual a partir de versiones escalonadas de la arquitectura, denominadas Q_1, Q_2, Q_3, Q_4, Q_5 y Q_6.

Actualmente, la arquitectura se presenta a los estudiantes mediante especificaciones formales que deben analizar y comprender para utilizar el lenguaje de programación y resolver los problemas que se les plantean en la práctica. Entendemos que les resulta de gran utilidad incorporar una herramienta que les permita probar sus ejercicios de una manera automatizada y es por eso que se desarrolló un simulador para esta arquitectura.

2. Conceptos importantes

2.1. Enfoque de Von Neumann

El matemático John Von Neumann en el año 1945 se encontraba colaborando en el proyecto ENIAC (*Electronic Numerical Integrator And Computer*, primer computadora electrónica de propósito general, diseñada para ser utilizada por el

ejército norteamericano. La ENIAC podía ser programada para realizar operaciones complejas e incluso decisiones, interacciones y subrutinas, pero la tarea de resolver un problema y volcarlo en la máquina era tan complejo que podía tomar semanas. Luego que el programa era diseñado en papel, el proceso de representarlo en la máquina ENIAC mediante la manipulación de cables e interruptores tomaba varios días. Entonces Von Neumann se comenzó a interesar por la problemática que significaba la la necesidad de reconfigurar la máquina para cada nueva tarea y tan sólo cuatro años más tarde propone y desarrolla una solución a este problema que se basaba en almacenar la información sobre las operaciones a realizar en la misma memoria utilizada para los datos, a partir de su codificación en código binario al igual que los datos.

Además, este enfoque generaliza la organización de las computadoras distinguiendo en tres partes interconectadas: La CPU (con la unidad aritmético-lógica o ALU y la unidad de control) la memoria, y un módulo de entrada/salida. La interconexión es llevada a cabo por un bus de sistema que proporciona un medio de transporte de los datos entre las distintas partes.

Con la propuesta de este modelo, Von Neumann incorpora el concepto de **programa almacenado** en memoria. Con esta idea, el programa se codifica de cierta manera para que pueda ser almacenado en memoria principal y posteriormente pueda ser ejecutado quizás múltiples veces. De esta manera, la lógica del programa puede ser "recordada" y el programa toma un valor mayor, a diferencia de lo que ocurría hasta entonces, donde el programa se reflejaba en un conjunto de configuraciones de cables aplicadas a los equipos. Esto implica una separación entre el mecanismo de ejecución (el *hardware*) y la lógica de computo o instrucciones (el *software*). La codificación en binario de las instrucciones de un programa se denomina **código máquina**.

Por otro lado este tipo de diseño, que permite un programa almacenado, también da la posibilidad de que la ejecución de las instrucciones modifique el código máquina del mismo u otro programa. Por ejemplo un programa podría modificar o incrementar las referencias a las direcciones de memoria que tenga en algunas instrucciones y luego volver a ejecutar dichas instrucciones con el fin de procesar celdas diferentes de memoria cada vez. Esta característica es potente pero presenta un alto riesgo pues las modificaciones en los programas podía ser algo perjudicial, por accidente o por diseño.

2.2. Organización de la computadora

La CPU (Unidad Central de Procesamiento del inglés: Central Processing Unit), es el componente principal y el encargado ejecutar los programas y procesar los datos. La CPU contiene otros componentes de importancia tales como la Unidad de Control, algunos registros de uso específico como el contador de programa (PC o *Program Counter*), el registro de instrucción (IR - *Instruction Register*) y el Puntero de pila (SP - *Stack pointer*), otros registros de uso general y la Unidad Aritmético-Lógica (ALU).

La Unidad de Control dirige el ciclo de ejecución de cada instrucción, pidiendo la lectura de celdas de memoria donde esta alojada, decodificándola y

ejecutándola luego en colaboración con los otros componentes del sistema: si es una operación lógica o aritmética le ordena a la ALU su ejecución, si es de movimiento de datos colabora con la memoria ó el módulo de Entrada/Salida.

Entre los registros de uso específico, los más importantes son el **Contador de programa**, el **registro IR**, el **Puntero de pila** y los **Flags**. El Contador de programa es un registro que indica la posición de memoria donde estará la siguiente instrucción que debe ejecutarse. Luego de completar el ciclo de ejecución de una instrucción, el PC se incrementa en función de la cantidad de celdas que ocupa el código máquina de esta. El registro de instrucción, **registro IR**, contiene el código máquina de la instrucción actual una vez que la misma es leída de memoria para luego decodificarla y ejecutarla. El Puntero de pila es un registro que indica la posición de memoria donde está el dato que representa a la última direccion de memoria que fue guardada al llamar a una sub-rutina, para que cuando esta termine, el programa pueda seguir ejecutandose a partir de ese punto. Los Flags representan carateristicas del último computo realizado como si el resultado fue negativo, cero, etc.

El diseño de cada arquitectura ofrece un conjunto diferente de registros de uso general para ser usados en los programas. Estos registros son elementos de memoria de alta velocidad y poca capacidad que pueden ser utilizados como variables en los programas. Es importante marcar que pueden almacenar tanto datos como direcciones de memoria.

La Unidad Aritmético-Lógica, recibe su nombre de las siglas en inglés de *Arithmetic and Logic Unit*. La ALU es un circuito digital que lleva a cabo operaciones aritméticas (suma, resta, multiplicación, división) y las operaciones lógicas como la negación, disyunción, conjunción, etc, entre dos cadenas binarias que son interpretadas como números o valores lógicos.

La memoria es un conjunto de celdas numeradas. La numeración de cada celda la identifica inequívocamente por lo cual a esta numeración se le llama dirección. En cada celda de la memoria se pueden almacenar datos o instrucciones en forma de cadenas binarias y este contenido puede leerse y modificarse. En la memoria es donde se alojan los programas que luego serán ejecutados.

El bus de sistema es el encargado de transferir los datos entre los componentes de la computadora..La unidad de control al pedir un contenido de una dirección de memoria lo hace a través del bus, y similarmente mismo cuando desea escribir en memoria, y lo hace a partir de que la Unidad de Control pide la lectura o escritura de celdas de memoria o puertos de entrada/salida.

2.3. Ejecución de un programa

La función de una computadora es la ejecución de programas. Los programas se encuentran almacenados en memoria y consisten en una secuencia de instrucciones y es la Unidad de Control es quien se encarga de ejecutar dichas instrucciones implementando un **ciclo de ejecución de instrucción**. Para ser almacenadas en memoria, las instrucciones deben codificarse en cadenas binarias (secuencias de ceros y unos) que no son legibles para las personas pero son tales que la Unidad de Control las puede interpretar y traducir en acciones. Por

eso para saber de qué instrucción se trata, y cuáles son los valores o variables (celdas de memoria o registros) que estan involucrados, la Unidad de Control toma el código máquina de la instrucción y verifica los códigos de operación y modos de direccionamiento. La ejecución de instrucciones se divide en tres etapas importantes:

1. búsqueda de instrucción
2. decodificación de la instrucción
3. ejecución de la instrucción

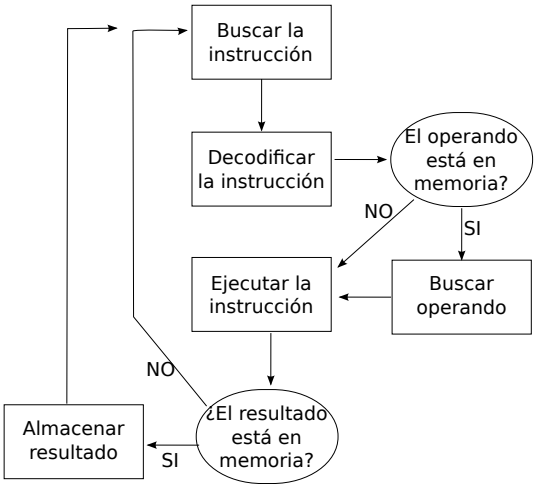

Figura 1: ciclo de ejecución de instrucción

Al principio de cada ciclo de ejecución de instrucciónse lleva a cabo la búsqueda de instrucción durante la cual se leen las celdas que contienen el código máquina de la instrucción y para lo cual se utiliza el Contador de programa que mantiene la dirección de la siguiente instrucción a ejecutar. El código máquina de la instrucción leída que está en la forma de cadena binaria se carga dentro de otro registro de la CPU, llamado registro de instrucción (IR).

Durante la decodificación de la instrucción, la Unidad de Control determina que operación se debe llevar a cabo, y finalmente durante la ejecución de la instrucciónla Unidad de Controlrealiza el efecto esperado para esa operación, buscando los operandos y modificando la memoria o los registros como resultado final y el ciclo vuelve a comenzar

3. Estado del arte

A través de los años de la carrera Tecnicatura Universitaria en Programación Informática, en la materia Organización de Computadoras se analizaron distintos enfoques y herramientas para desarrollar los conceptos relacionados a la ejecución de programas en una computadora.

Inicialmente se utilizó un simulador de código abierto para la arquitectura Intel 8085, que ofrecía una funcionalidad bastante completa, pero una interfaz que no resultaba del todo intuitiva. Este simulador se eligió por tratarse de un lenguaje assembler mas reducido, con un repertorio de instrucciones y modos de direccionamiento mas pequeño que contaba con un entorno de prueba (el simulador propiamente dicho) para facilitar la didáctica de la programación en lenguaje ensamblador, pero posteriormente se entendió que las características de la arquitectura no eran las adecuadas para la enseñanza de los contenidos y se descartó.

Entonces el equipo docente eligió definir una arquitectura teórica que proveyera solamente lo necesario para cumplir con los objetivos de la materia e inspirados en un caso similar de la Universidad de Buenos Aires, se definió la arquitectura **QARQ**, muy similar a la que se presenta en la sección A

Posteriormente, el equipo docente propuso un cambio en la secuencia didáctica que requirió la división de la especificación de la arquitectura QARQ en varias partes, donde cada una recibe el nombre de **Arquitectura Qi** y agrega una nueva funcionalidad (instrucciones o modos de direccionamientos) a la versión anterior, construyendo una arquitectura en capas. Sin embargo todos los ejercicios de las arquitecturas Qi se siguen haciendo en papel.

Actualmente se busca incorporar el **Simulador Qsim** en la materia con el objetivo de que los alumnos puedan visualizar el funcionamiento de una computadora al mas mínimo detalle, a través de la ejercitación del ciclo de ejecución de instrucción.

4. Arquitecturas Q

Las versiones de la arquitectura Q están pensadas para incorporar funcionalidades de manera que la curva de aprendizaje sea adecuada para los alumnos, siendo paulatina e incremental, es decir, cada arquitectura Q_{i+1} agrega más funcionalidad (ya sean instrucciones nuevas o modos de direccionamiento) a la arquitectura Q_i anterior.

Parte II
Simulador QSim

El simulador desarrollado es una herramienta de utilización sencilla ya que se asume que los alumnos de esta materia están en la etapa inicial de la carrera y se pretende transmitir los conceptos sin distraerlos con detalles de uso y configuración.

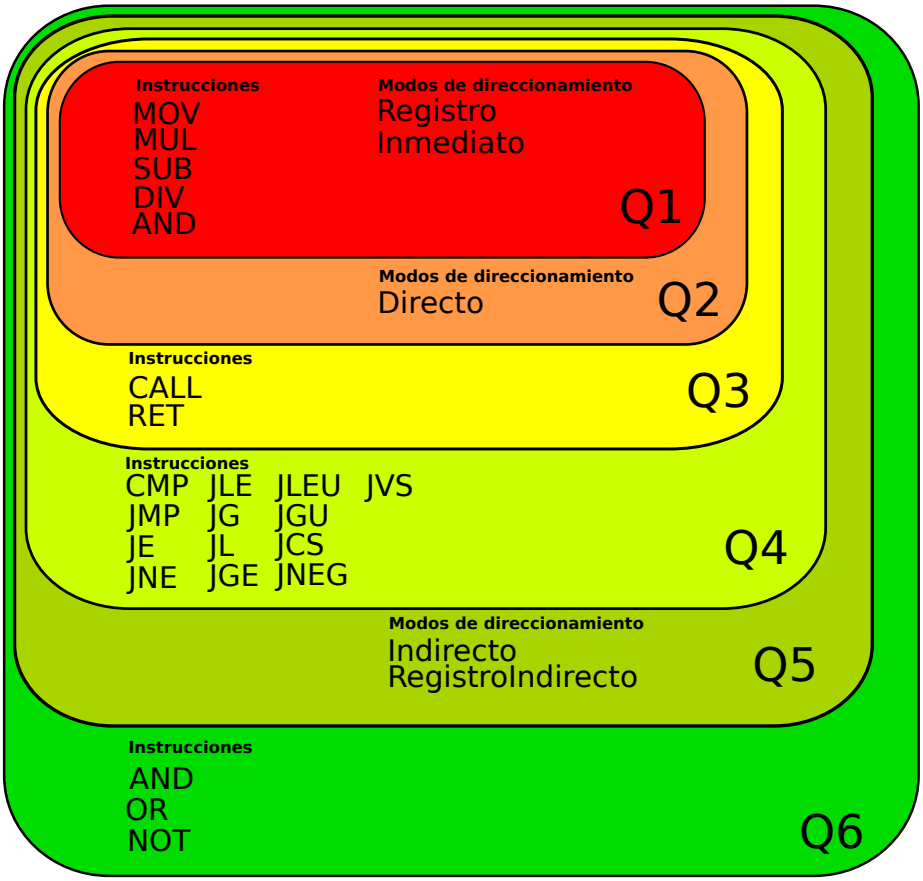

Figura 2: Cebolla Qi

5. Funcionalidad del simulador

La funcionalidad del simulador puede caracterizarse mediante las siguientes partes importantes:

- Chequeo de sintaxis de los programas escritos en el lenguaje Q
- Ensamblado del código fuente de un programa en su correspondiente código máquina
- Cargado en memoria del código máquina
- Ejecución paso a paso de un programa cargado en memoria

5.1. Chequeo de sintaxis

El simulador provee al alumno de un editor de texto en el cual escribirá el programa en un lenguaje Qi, que desea cargar en memoria y ejecutar. Una vez que el usuario haya terminado la escritura, al momento de cargar el programa,

el simulador utilizará un *parser*[1] para detectar errores de sintaxis, tales como la falta de una coma o un corchete, o la presencia de símbolos que no pertenecen al lenguaje (como por ejemplo signos de pregunta y símbolos matemáticos); o bien errores semánticos como la combinación incorrecta de elementos del lenguaje, por ejemplo: modos de direccionamiento mal ubicados. El parser solo revisará lo escrito por el alumno y de acuerdo a las gramática del lenguaje, mostrará alguno de los siguientes estados:

OK Este mensaje se obtiene cuando no hubo ningún error de sintaxis. Si se da este resultado, es posible continuar con el ensamblado y cargado en memoria.

SyntaxError Este mensaje de error se obtiene cuando en alguna línea del programa se detectó algún error de sintaxis o de semántica, como se describió arriba. Cuando ocurre este error se lo acompaña con una descripción lo mas detallada posible para que el alumno detecte donde ocurrió y pueda corregirlo. Un programa con errores no puede ser ensamblado y cargado en memoria.

5.2. Ensamblado

Una vez que el programa es sintácticamente válido es posible traducir el código fuente del programa en código máquina (representado en cadenas binarias). Para esto se respeta un formato de instrucción que indica cómo se codifica cada operación y los operandos.
Mas detalle al respecto de este proceso en el apéndice A.

5.3. Cargado en memoria

Una vez ensamblado, la representación binaria (o código máquina) del programa será cargado en memoria a partir de una ubicación (celda de memoria) que el alumno puede elegir. Esto permite visualizar el contenido de la memoria (con el programa cargado) y el estado de los registros de la CPU.

Durante la carga del programa en memoria puede ocurrir que el programa no cuenta con el espacio suficiente a partir de la ubicación elegida ya que ocupa más celdas que las que se encuentran disponibles, ya que como se especifica en el apéndice A, la memoria disponible tiene un tamaño limitado y por este motivo la alocación en memoria del código máquina puede exceder el espacio disponible a partir de la celda inicial anteriormente elegida. Si por el contrario, no se produce este error, el alumno podrá ver el programa cargado en memoria exitosamente.

5.4. Ejecución paso a paso

Se provee la funcionalidad de la ejecución paso a paso ya que se desea que el alumno pueda experimentar y así comprender los pasos del ciclo de ejecución. Además puede ejercitarse situaciones que se denominan "errores conceptuales de programación" Esto es a lo que llamamos Errores conceptuales, entre los cuales es posible mencionar:

[1] Un parser (o analizador sintáctico) es una de las partes del compilador que transforma su entrada de texto plano en este caso a objetos del modelo.

- Tomar un dato de un sector de memoria equivocado.
- Que el programa sobrescriba su mismo código máquina.
- Permitir que la ejecución continue una vez procesadas las instrucciones del programa cargado en memoria.

El paso a paso que provee el simulador consiste en las siguientes etapas pertenecientes al ciclo de ejecución de instrucción:

1. **Búsqueda de instrucción:** El alumno podrá visualizar el valor que contiene PC (Program counter) donde se encuentra la dirección de la celda en memoria que contiene la próxima instrucción a ejecutar (por ejemplo, en caso de ser la primer instrucción del programa recién cargado, el pc tendrá la dirección de memoria elegida por el alumno para iniciar el cargado del programa en memoria). El simulador, toma de la memoria el código maquina correspondiente a la instrucción que comienza en esa dirección tomada de PC (una instrucción puede ocupar más de una celda de memoria) y los guarda en el registro IR (*Instruction Register*). Será observable también para el alumno el incremento del registro PC, tantas como celdas ocupe la instrucción actual, lo que conceptualmente es, preparar el contexto de ejecución para tomar la siguiente instrucción.
2. **Decodificación:** En la decodificación el Interprete se encarga de desensamblar el código máquina (abreviado en hexadecimal) que ya fue ubicado en el registro IRpara mostrar el código fuente de la instrucción actual con sus respectivos operandos. Si el programa escrito por el alumno es sintacticamente y conceptualmente correcto, este paso le permite comprobar que la instrucción actual es la que él mismo escribió y no otra, visualizandola en pantalla. En esta etapa se provee también la oportunidad de que el alumno aprecie otros conceptos, tales como los errores conceptuales mencionados antes.
3. **Ejecución** El execute ejecuta los efectos de la instrucción y muestra en pantalla los cambios en el estado de ejecución: memoria, puertos, registros y flags. Dentro de esta misma etapa se lleva a cabo el almacenamiento de resultados que, cuando sea necesario, guardará el valor resultante de la operación descripta por la instrucción en el operando destino. Esto cambiará el valor de una celda de memoria o de un registro y será visto en pantalla por el alumno.

6. Implementación

6.1. Tecnología utilizada

En la presente sección se indica la tecnología elegida para la implementación del simulador, justificando dichas elecciones en cada caso.

- **Lenguaje Scala.** Elegimos el lenguaje Scala para realizar el simulador ya que durante las cursadas de las asignaturas de TPI no tuvimos la oportunidad de profundizar el dominio de este lenguaje ni aprovechar las ventajas que este ofrecía al combinar el manejo de objetos y las características de un

lenguaje funcional. Es por ello que descartamos la elección de un lenguaje con el que estábamos mas familiarizadas, como por ejemplo Java.

- **Framework Arena.** Utilizamos el framework Arena para realizar la interfaz de usuario del simulador porque es una herramienta de código abierto que tuvimos la oportunidad de conocer en la materia Diseño de Interfaces de Usuario. Al poder ser combinado con **Scala** nos pareció una buena oportunidad de explotar lo que nos ofrecía para simplificar la definición de la interfaz de usuario, permitiéndonos así enforcarnos en la implementación del modelo.

- **Eclipse.** Se eligió utilizar el entorno de programación Eclipse ya que es una herramienta multiplataforma, lo que nos permitió trabajar en diferentes sistemas operativos y con la cual estábamos familizadas. Por otro lado, la comunidad provee pluggins para manejar proyectos para Scala.

- **Git** Elegimos git como repositorio externo para trabajar colaborativamente durante el desarrollo, dado que es una herramienta que muy extendida en los desarrollos de software libre.

6.2. Diseño Orientado a Objetos

6.2.1. ALU

Como se observa en la figura 2 la ALU tiene toda la responsabilidad en la ejecución de operaciones matemáticas y lógicas, además del cómputo de los flags luego de cada operación.

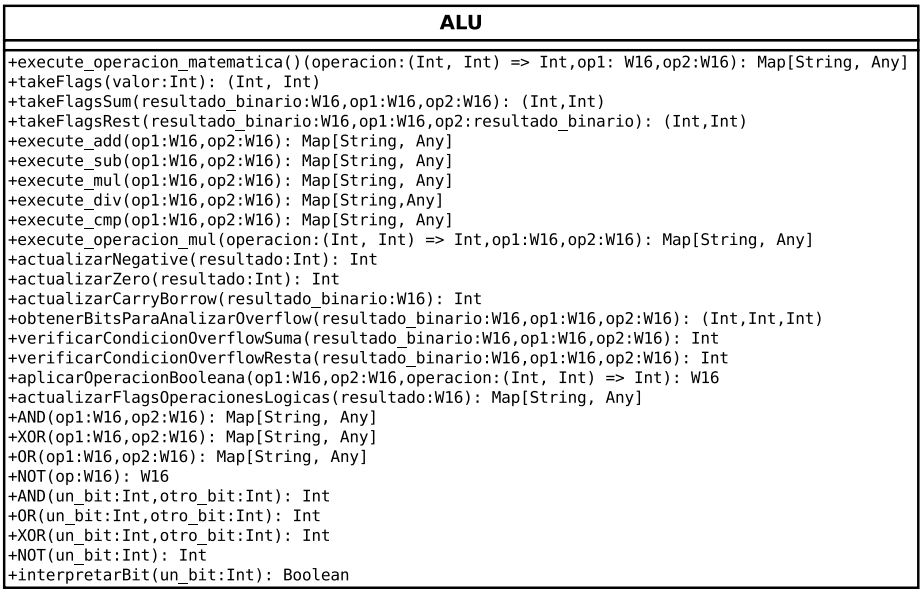

Figura 3: Diagrama de clase de la Unidad Aritmético-Lógica

6.2.2. Bus de entrada y salida, memoria y puertos

Como se observa en la figura 3 el Bus de entrada y salida tiene la responsabilidad de derivar según donde corresponda (Memoria o Puertos) la escritura o lectura de un dato. Para ello conoce a una instancia de la clase Memoria y a otra de la clase CeldasPuertos. Ambas clases conocen una colección de instancias de la clase Celda, y cada Celda a su vez conoce un dato: una instancia de la clase W16 que representa al dato almacenado en una celda de memoria o en un puerto.

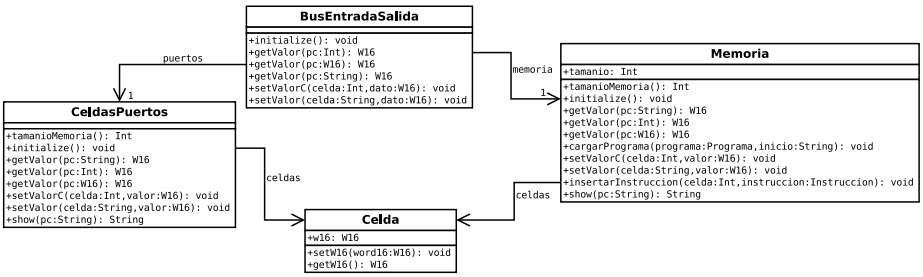

Figura 4: Diagrama de las clases del la BusEntradaSalida, Memoria, CeldasPuertos y Celda

6.2.3. CPU

Como se observa en la figura 4, la CPU conoce a la ALU, contiene los registros IR y PC, los flags (V,Z,C,N) y los ocho registros de uso general (R0...R7). La responsabilidad de la CPU es actualizar los flags, los registros, actualizar el PC y el IR, y ser la conexión con la ALU.

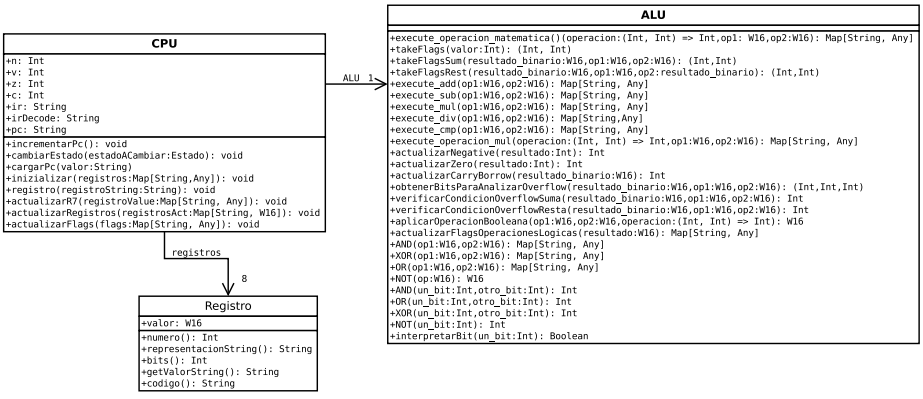

Figura 5: Diagrama de clase de la CPU

6.2.4. Intérprete

Como se observa en la figura 5 el Interprete es un *singleton* que tiene la entera responsabilidad de construir un objeto que representa una instrucción

determinada a partir de la decodificación de las celdas de memoria que ocupa su código máquina. Se ocupa de la decodificación de la instrucción.

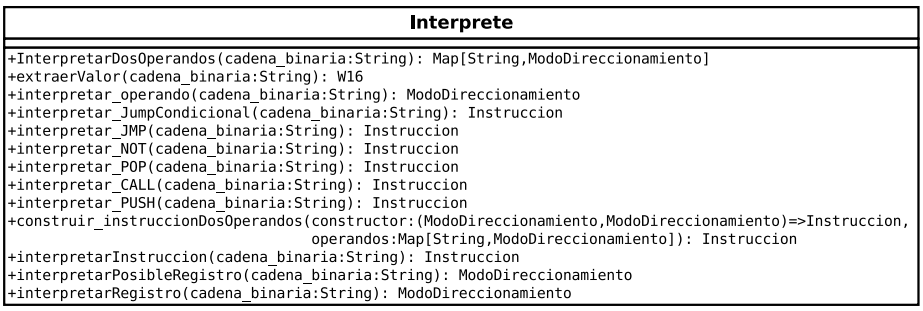

Figura 6: Diagrama de clase del Interprete

6.2.5. Modos de direccionamiento y W16

Esta jerarquía de clases permite controlar el acceso a los operandos, mediante un objeto ModoDireccionamiento que controla el acceso a un objeto W16, que es quien contiene el valor propiamente dicho. Como se observa en la figura 6 los modos de direccionamiento extienden del trait ModoDireccionamiento, donde se encuentran declarados mensajes necesarios para manejar todas las subclases de manera polimórfica. Entre estos mensajes podemos enumerar:

- **representacionString:** Devuelve la representación en string como código fuente, por ejemplo la representación de un objeto **ADD(R0,R7)**, sería: `ADD R0, R7` este mensaje se utiliza en la etapa de desensamblado.
- **codigo:** Retorna el string que representa al código del modo de direccionamiento, por el ejemplo, el código de modo de direccionamiento del R7 es `100111`. este mensaje se utiliza en la etapa de ensamblado.
- **getValorString:** Retorna el dato almacenado en el operando. En el caso de un Inmediato que sea `FF56`, devolverá el string "FF56", y en el caso de cualquier registro, retornara el valor que represente su W16.

La clase W16 que también esta en la figura 6, representa el dato que es guardado en memoria. Tiene la capacidad de incrementarse, decrementarse, sumar una entero, devolver su representación binaria y su valor en decimal.

Los modos de direccionamiento diferentes a Inmediato y Registro, conocen otro modo de direccionamiento que encapsula al objeto **W16** según corresponda, es decir:

- **RegistroIndirecto** conoce una instancia de Registro.
- **Directo** conoce conoce una instancia de Inmediato.
- **Indirecto** conoce conoce una instancia de Directo.

La clase **Etiqueta** representa las etiquetas creadas por el alumno cuando realiza el programa. Cuando el mismo es cargado en memoria, en función de cual sea la celda de inicio y cuanto ocupen las instrucciones, se calcula la dirección de memoria a la que hace referencia y luego se la descarta reemplazándola por un modo de direccionamiento Inmediato.

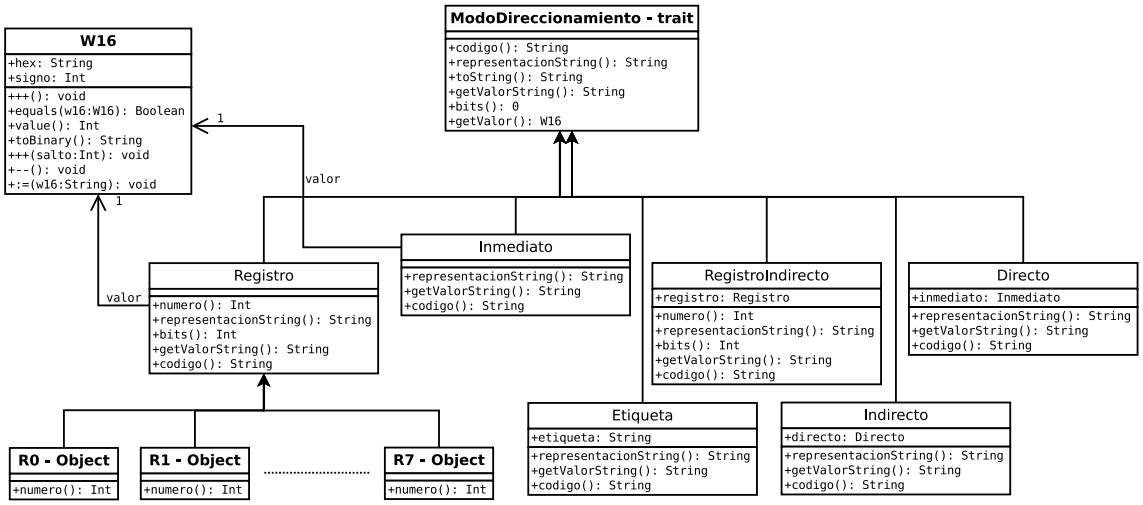

Figura 7: Diagrama de clase de la jerarquía de los modos de direccionamiento

6.2.6. Instrucciones

Como se observa en la figura 7 las Instrucciones están jerarquizadas en:

- Instrucciones de un operando.
- Instrucciones de dos operandos.
- Instrucciones de sin operandos.
- Saltos condicionales.

Se realizó dicha jerarquía para permitir la fácil incorporación de nuevas instrucciones como subclases de la clase que corresponda ya que de ese modo se reutiliza comportamiento tal como la manera de mostrarse (en términos de código fuente) y de codificarse (en términos de código máquina).

Instrucciones sin operandos Como se observa en la figura 8 la única instrucción sin operandos implementada en la arquitectura Q es la instrucción `RET`. A pesar de esto, se eligió hacer una jerarquía para que luego se facilite la escalabilidad del modelo, permitiendo la inserción de nuevas instrucciones sin operandos.

Instrucciones de un operando Como se observa en la figura 9 y siguiendo con el criterio mencionado antes sobre la jerarquía de instrucciones, puede haber dos tipos de instrucciones de un operando:

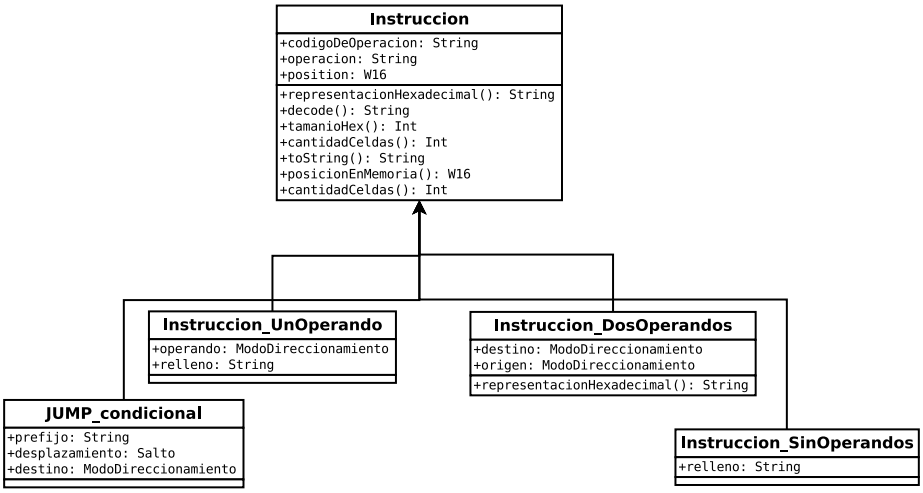

Figura 8: Diagrama de clase de la Instrucción

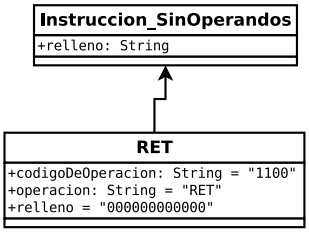

Figura 9: Detalle de clase Instruccion_SinOperandos

- Un operando origen.
- Un operando destino.

Ambas clases de instrucciones tienen un solo operando. La diferencia entre ellas es la lógica de ejecución y la manera en la que se ensamblan y desensamblan ya que sus formato de instrucción difiere (ver apéndice A).

Instrucciones de dos operandos Como se observa en la figura 10 la jerarquía de clases de instrucciones de dos operandos es la más amplia por tener mayor cantidad de instrucciones. Todas comparten el comportamiento para la decodificación e interpretación, además de la lógica de impresión.

6.2.7. Programa

Como se observa en la figura 11 la clase Programa conoce un grupo de Instrucciones (las instrucciones que lo componen). Las instancias son creadas por el *parser* (ver sección 5.1), luego se calculan las etiquetas (si es que las tiene) y finalmente cuando es cargado en memoria la instancia de programa es desechada ya que no vuelve a usarse en ningún momento de la ejecución.

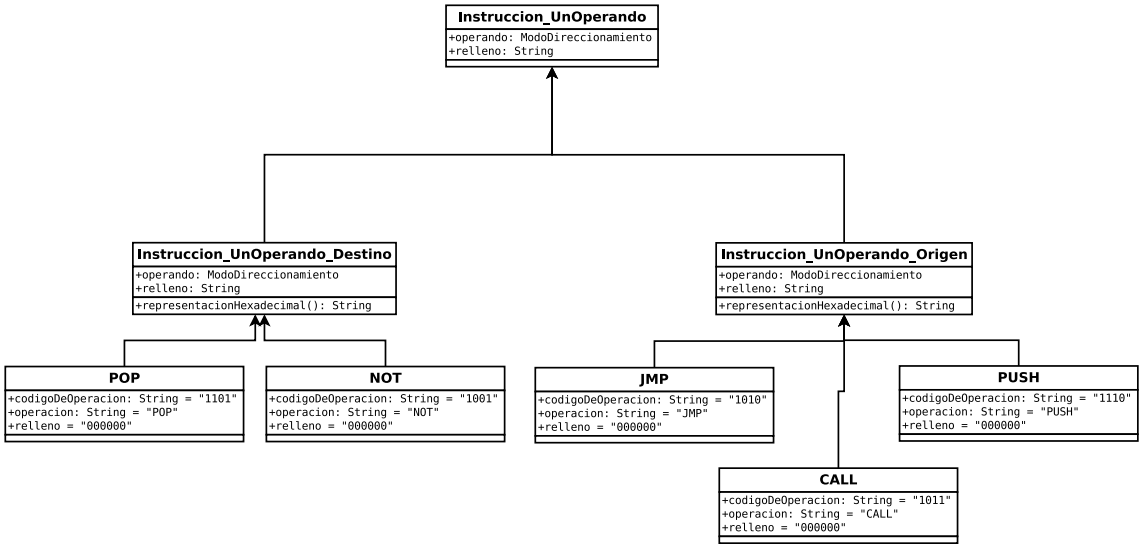

Figura 10: Diagrama de clase de la Instruccion_UnOperando

6.2.8. Simulador

Esta e la clase principal del modelo y la encargada de coordinar la ejecución del programa paso a paso. Como se observa en la figura 12 la clase **Simulador** conoce a una instancia de la clase **CPU**, a una instancia de la clase **BusEntradaSalida** y a una intancia de la clase **Instruccion**, que representa a la instrucción que se esta ejecutando en ese momento.

La clase Simulador tiene la responsabilidad de obtener el código máquina de la siguiente instrucción, colaborando con el Interprete (ver **??**), calcular las etiquetas de un programa, cargar el programa en memoria y los datos en registros, ejecutar las instrucciones o delegar su ejecución al objeto **ALU** que conoce a través de la **CPU** según corresponda, y guardar datos en memoria o registros (almacenamiento de resultados).

Parte III
Evaluación del desarrollo

7. Dificultades encontradas

7.1. Dificultades presentadas por el dominio

Las dificultades del dominio estuvieron relacionadas a la comprensión no solamente del modelo de arquitectura Q si no también a su propósito didáctico, ya que el objetivo del simulador no es solamente la simulación de la arquitectura y la ejecución de programas, sino también el proveer a los alumnos la capacidad de ejercitar situaciones conceptualmente erróneas, por lo que se requería

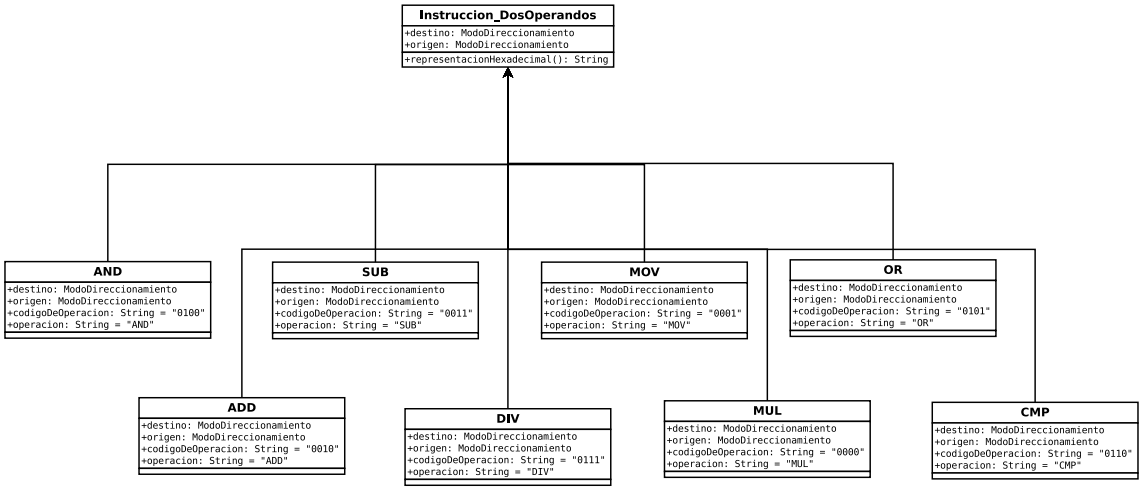

Figura 11: Diagrama de clase de la Instruccion_DosOperandos

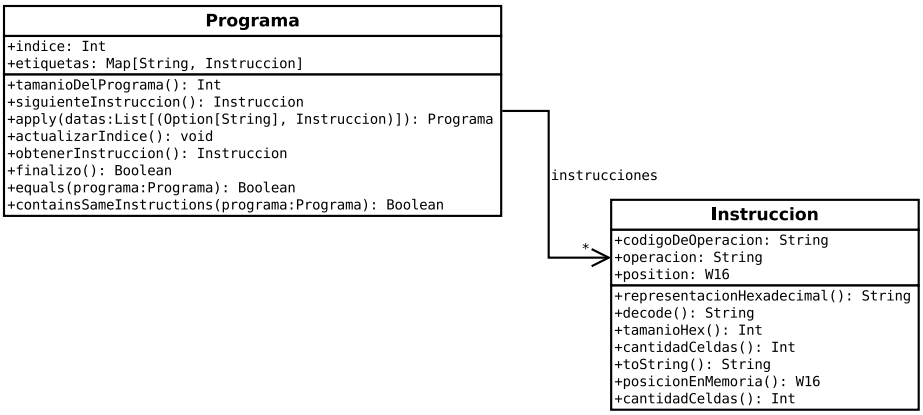

Figura 12: Diagrama de clase de la Programa

una comprensión didáctica del problema más allá de la especificación de las arquitecturas Q.

7.2. Dificultades de diseño

En primera instancia se optó por implementar un modelo de objetos que utilizaba un objeto de la clase **Programa** a lo largo de toda la ejecución.Esto implica que no leía de la memoria principal las instrucciones a ejecutar, sino que solicitaba la siguiente instrucción al objeto instancia de la clase Programa, sobreviviendo así las distintas etapas una vez que fue creado por el *parser*. Este enfoque evitaba la creación de un objeto que tuviera que interpretar el código máquina alojado en la memoria, evitaba la nueva creación de instancias de la clase **Instrucción** y simplificaba en gran medida el modelo ya que la memoria era sólo una clase que contenía datos que se reflejaban en pantalla y no se extraían datos de celdas en ningún momento. Luego entendimos que un programa

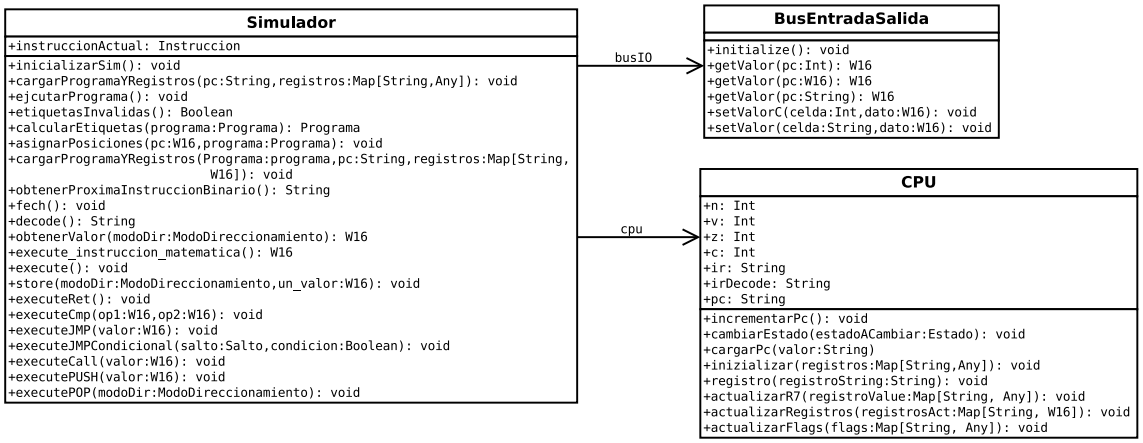

Figura 13: Diagrama de clase de la Simulador

no sólo podía modificar su entorno al ser ejecutado (otras celdas de memoria que no ocupen su código maquina, celdas de puertos, registros, etc) si no que también podría sobrescribir su código maquina (ya sea con ese propósito o sólo por un error conceptual), o bien, que el alumno debía tener la posibilidad de seguir ejecutando más allá del código máquina alojado en memoria o más. A partir de eso fue necesario corregir gran parte del modelo agregando una clase denominada **Intérprete**, cuya responsabilidad es interpretar la siguiente instrucción alojada en memoria para que luego sea ejecutada, otorgando más responsabilidad a la clase **Memoria** y descartando el objeto instancia de **Programa** una vez que éste es cargado en memoria con éxito.

Por otro lado, tuvieron que solicitarse extensiones al equipo de desarrolladores de Arena para poder realizar la interfaz con dicho framework:

- FileSelector para que los alumnos puedan cargar el archivo .Qsim en el cual se encuentra su programa
- CodeEditor (O actualmente llamado KeywordTextArea). Es el espacio donde se visualiza el programa Qi que realizaron los alumnos una vez que esta cargado.
- Bindings contra el background de componentes y celdas de una tabla, para permitir la visualización de cambios de colores en la memoria luego de las etapas de ejecución y de realizar algún cambio en la misma.
- TexBox multiLine (TextArea) con scroll para ser utilizado como consola de devolucion.
- Icono para la aplicación, solo por cuestiones estéticas.

8. Casos de prueba (revisar redaccion!!)

En esta sección se describen los casos de prueba de la aplicación.[2]

[2]Falta explicar porqué es importante esta sección

8.1. Chequeo de sintaxis en la distintas Qi

Para chequear la sintaxis de cada **Arquitectura Q**, primero escribimos un programa Qi en un archivo con extension .qsim luego este archivo lo recibe el objeto Parser que se encargara de chequear la sintaxis. Para eso realizamos dos casos de prueba por cada Qi:

1. **Chequear programa Qi valido**

   Agarramos el programa Qi válido[3] y se lo pasamos al Parser, cuando termina de chequear devuelve un resultado, como es de esperar devuelve un objeto programa con la lista de instrucciones. Tomamos ese resultado y como ultimo paso comparamos si el programa resultado es igual al programa esperado.[4]

2. **Chequear programa Qi es invalido**

   En este caso escribimos un programa Qi con errores de sintaxis para ver como reacciona el parser.

   Agarramos el programa Qi invalido y se lo pasamos al Parser, cuando termina de chequear devuelve un error, como es de esperar devuelve una Exception **SyntaxErrorException** porque dentro del archivo del programa hay una linea que tiene errores de sintaxis, el mensaje de la excepción te muestra el numero de linea y la propia linea para que veas en donde te equivocaste. Capturamos la Excepción y tomamos como resultado el mensaje y como ultimo paso comparamos el mensaje resultado con el mensaje esperado.

8.2. Ensamblado

(Objetos –+ Código Maquina)[5] Para poder verificar que el Ensamblado se realizo correctamente tomamos el programa Qi mencionado anteriormente y como sabemos que el resultado del parser nos devuelve un objeto de la clase **Programa** que contiene un conjunto de objetos de la clase **Instruccion**, el ensamblado simplemente lo realiza cada instrucción, esto quiere decir que cada instrucción tiene el comportamiento para generar su propio código máquina. De esta forma realizamos el ensamblado del programa y obtenemos una lista de código maquina como resultado y como ultimo paso comparamos la lista de código maquina resultado con la lista código maquina esperado.

8.3. Decodificación

Código Maquina –+ Objetos (Interprete)[6]

La Decodificación se verifica correctamente a la inversa del Ensamblado tomando el ensamblado del programa Qi que como sabemos es la lista de código maquina, la cual se itera para pasar cada elemento al objeto Interprete que es el encargado de Decodificar. Como resultado puede devolver dos cosas:

[3] Un programa Qi valido es un programa en cualquier sintáticamente correcto en alguna arquitectura Q.

[4] OJO con la redacción

[5]

[6]

1. **Lista de de Instrucciones (decodificación)**

   El resultado correcto es la lista de objetos instrucciones, por cada código maquina el interprete verifica los códigos de operación para poder crear las instrucciones correctas y como ultimo paso de la verificación comparamos la lista resultante con la lista esperada.

2. **Error**

   El resultado puede ser un error porque puede pasar que el código de operación o el código de algún modo de direccionamiento dentro del formato de cada instrucción sea invalido. La excepción que puede tirar tiene el nombre de **CodigoInvalidoException**.

8.4. Ejecución

Para verificar que la ejecución de un programa Qi con el **Ciclo de Instrucción** se realice correctamente lo que hacemos es cargar el programa en la memoria a través del objeto **Simulador**, de esta forma cuando se encuentra cargado esta listo para la ejecución. Los pasos que se verifican son 3:

- **Fetch**

  Para verificar el fetch se toma la instrucción siguiente a ejecutar y se compara el valor que se guarda en el registro ir con el valor esperado. Ademas se verifica que el registro especial pc apunte a la siguiente instrucción a ejecutar.

- **Decode**

  Para verificar el decode, se toma el valor del registro ir para que lo reciba el interprete que dará como resultado el objeto instrucción correspondiente. Este objeto instrucción sabe mostrarse quien es y por eso la responsabilidad de decodificar se la delega a la instrucción. Al obtener la decodificación de cada instrucción podemos comparar la decodificacion resultante con la esperada.

- **Execute**

  Para verificar el efecto de cada instrucción se considero armar un caso de prueba por cada instrucción donde cada uno tiene un estado inicial y luego de realizar la ejecución de dicha instrucción termina con un estado final modificado en el cual verificamos el resultado obtenido con el resultado esperado.

Las siguiente secciones son pasos que se realizan de acuerdo al efecto de la instrucción

8.4.1. Operaciones de ALU

La ALU es la encargada de ejecutar operaciones aritméticas/lógicas. Si la instrucción a ejecutar tiene un efecto aritmético/lógico se le delega la ejecución de la operación. Para poder verificar todas las operaciones creamos un caso de prueba por cada operación. Tanto las operaciones aritméticas como las lógicas se analizan de la misma forma:

1. Se toma cada valor de los operandos(la búsqueda se detalla el la siguiente instrucción)
2. Se realiza la operación aritmética/lógica.
3. Se verifica el resultado obtenido con el resultado esperado.
4. Algunas operaciones (la mayoría) modifican los flags. la ALU tiene mucha relación con los flags ya que por cada cuenta realizada actualiza los mismos. Estos también se se verifican comparando el resultado obtenido con el valor esperado.

8.4.2. Búsqueda de Operandos

Para verificar la búsqueda de operandos, tomamos una instrucción cualquiera y probamos todas las combinaciones de modos de direccionamiento para cada operando, teniendo en cuenta que son invalidas todas las combinaciones de 1/2 operando/s que tenga el operando destino el modo de direccionamiento Inmediato. Al tener la instrucción elegida procedemos a obtener el valor de cada operando, este valor se lo compara con el valor esperado.

8.4.3. Almacenamiento de Operandos

Para verificar el almacenamiento de operandos, salvo algunas instrucciones todas pasan por la etapa de store. Tomamos cada una de las instrucciones de un programa Qi y realizamos el ciclo de ejecución mencionado anteriormente. Cuando estamos en la etapa de ejecución, cada una tiene un operando destinado a guardar el resultado de su efecto. En esta instancia sabemos que dicho operando tiene el valor guardado, a este valor lo comparamos con el valor esperado.

9. Ejemplos de uso (Revisar!!)

En esta sección vamos a mostrar las diferentes situaciones que pueden ocurrir a la hora de utilizar el simulador.

9.1. Para Experimentar

Esta sección describe situaciones particulares que los alumnos pueden experimentar.

- Supongamos que un alumno escribe el siguiente programa Qi:

  1.ADD R0, [0x0002]
  2.MUL R4, 0x0001
  3.SUB [0x0003], 0x000A
  4.MOV R5, 0x0056
  5.MOV [0x0005], etiqueta
  6.CALL [0x0005]
  7.etiqueta: ADD R0, 0x0002
  8.RET

Lo interesante es que luego de ejecutar el CALL se ejecuta la instrucción ADD R0, 0x0002 y el RET. Cuando vuelve para ejecutar la próxima instrucción el PC se encuentra en la linea 7, donde vuelve a ejecutar la instrucción ADD antes mencionada. Ya lo ultimo que le queda es ejecutar el RET. Vamos a hacer un mapa del estado del registro SP antes de seguir: Antes de ejecutar el RET el registro SP tiene el valor inicial que es FFEF, luego en la ejecución lo primero que se hace es incrementar el SP osea que ahora tiene el valor FFFO y luego buscar el valor de esa dirección para actualizar el registro PC. Para informarles la dirección FFFO es un puerto de E/S. La conclusión es que en esta instancia el flujo de ejecución del programa depende del valor que tiene ese puerto, puede pasar que el valor sea 0x0000 y se actualice el PC nos lleve al inicio de la memoria donde se encuentra inicializado otro programa y empiece a ejecutar desde allí.

- Para los que son curiosos, supongamos que un alumno escribe el siguiente programa Qi:

```
ADD R0, [0x0002]
MUL R4, 0x0001
SUB [0x0003], 0x000A
```

Pensemos que el PC se posiciona en la linea 3 y realizamos el ciclo de instrucción(FETCH - DECODE - EXECUTE). Como estado final tenemos el efecto de la ultima instrucción y el valor de PC apuntando a la siguiente instrucción a ejecutar. Como verán el programa que escribió el alumno termino en la la linea 3 pero el simulador no para de ejecutar, por ende te permite realizar las veces que quiera el ciclo de instrucción. Es re interesante que los chicos experimenten este tipo de cosas.

La decodificación con desensamblado permite al alumno experimentar otros escenarios y efectos laterales, entre los cuales podemos enumerar:

- Si la ejecución paso a paso excede los límites del programa, pueden tomarse instrucciones de otra rutina y procesarse como una nueva instruccion.
- Si en cambio, se intenta ejecutar el contenido de una celda con datos (y no una instrucción) podrá ocurrir que se encuentre una instrucción invalida (por ejemplo, una combinación incorrecta de modos de direccionamiento y códigos de operación) y el alumno verá el mensaje de error pertinente.

ACOMODAR!!!

10. Trabajo Futuro

En esta sección se describirán características y funcionalidades que deseamos agregar al Simulador QSim en el futuro.

- Habilitar las instrucciones PUSH y POP.

  Estas instrucciones permiten el manejo de la Pila como estructura de datos disponible para el programador. PUSH tiene como efecto agregar el valor

del operando origen a la pila. POP permite sacar el primer elemento de la pila y guardarlo en el operando destino. Actualmente estas instrcucciones se encuentran implementadas pero no habilitadas en ninguna gramática Qi.

- Entrada y Salida.

  Que el simulador admita la interacción con dispositivos de E/S. Para eso tenemos que modelar dispositivos como el teclado, impresora, monitor, etc.

Parte IV
Apéndices

A. Especificación de la arquitectura Q

A.1. Características generales

La Arquitectura Q tiene 8 registros de uso general de 16 bits, denominados R0..R7, registros especiales de 16 bits tales como PC - *Program counter*, SP[7] - *Stack Pointer* y los Flags de un bit como Negative, oVerflow, Carry, Zero. También tiene un conjunto de instrucciones que detallaran mas adelante. Todas las instrucciones Alteran los flags excepto MOV, CALL, RET, JMP, Jxx. De las instrucciones que alteran los Flags, todas dejan C y V en 0 a excepción de ADD, SUB y CMP.

Por ultimo tiene una Memoria que tiene direcciones de 16 bit, donde el tamaño de cada celda también es de 16 bit. La Memoria tiene un tamanio de 65536 celdas.

A.2. Modos de direccionamiento

Los siguientes son los modos de direccionamiento implementados en la Arquitectura Q.

1. **Inmediato** Representa un operando que denota un valor constante. Es importante notar que este modo direccionamiento es admitido en el operando origen pero no el operando destino. La codificación de este modo se indica en la tabla 1.

   Ejemplos:

   - **0x0000** denota el modo de direccionamiento inmediato cuyo valor es cero.
   - **0x000F** denota el modo de direccionamiento inmediato cuyo valor es 15.

[7] Comienza en la dirección FFEF.

Modo	Codificación
Inmediato	000000
Directo	001000
Indirecto	011000
Registro	100rrr
Registro indirecto	110rrr

Cuadro 1: Tabla de códigos de los modos de direccionamiento (Nota: rrr describe el número de registro)

2. **Directo** Con este modo de direccionamiento se denota un operando alojado en una dirección de memoria o de puertos. La codificación de este modo se indica en la tabla 1.

   Ejemplos:

   - [**0x0000**] denota un operando cuyo valor se encuentra en la celda de memoria cuya dirección es **0x0000**.
   - [**0x000F**] denota un operando cuyo valor se encuentra en la celda de memoria cuya dirección es **0x000F**.

3. **Indirecto** Con este modo de direccionamiento se denota un operando alojado en una celda de memoria cuya dirección está almacenada en otra celda de memoria. La codificación de este modo se indica en la tabla 1.

   Ejemplos:

   - [[**0x0000**]] denota un operando cuyo valor se encuentra en la celda de memoria cuya dirección esta guardada como dato en la celda de memoria cuya dirección es **0x0000**
   - [[**0x000F**]] denota un operando cuyo valor se encuentra en la celda de memoria cuya dirección esta guardada como dato en la celda de memoria cuya dirección es **0x000F**

4. **Registro** Con este modo de direccionamiento se denota un operando alojado en un registro de uso general (R0 a R7). La codificación de este modo se indica en la tabla 1.

   Ejemplos: **R0** denota un operando almacenado en el registro R0. **R7** denota un operando almacenado en el registro R7.

5. **Registro Indirecto** De manera similar al modo indirecto, con este modo de direccionamiento se denota un operando alojado en una celda de memoria cuya dirección está almacenada en el registro indicado. La codificación de este modo se indica en la tabla 1.

   Ejemplos: [**R0**] denota un operando almacenado en una celda de memoria cuya dirección está en el registro **R0**. [**R7**] denota un operando almacenado en una celda de memoria cuya dirección está en el registro **R7**.

A.3. Repertorio de instrucciones

En esta sección se detalla cómo se construye el código máquina de las instrucciones de la arquitectura.

A.3.1. Instrucciones de 2 operandos

A continuación se muestra la codificación (formato) de las instrucciones de dos operandos:

Codigo de Operación (4b)	Modo Destino (6b)	Modo Origen (6b)	Destino (16b)	Origen (16b)

Las instrucciones de dos operandos descriptas a continuación son instrucciones aritméticas o lógicas donde se asume que el resultado de la operación se almacena en uno de los dos operandos de entrada, y por lo tanto se lo denomina **operando destino**.

1. **MUL destino, origen** Código de operación: 0000 Esta instrucción describe la multiplicación entre los datos de los dos operandos. Esta operación es la única que cuyo resultado puede ser 32 bits, que son lo que ocuparía más de una celda de memoria en código binario, por lo que los primeros 16 bits, es decir, la primer mitad, es guardada en el registro **R7** y la segunda en el operando destino.

2. **ADD destino, origen** Código de operación: 0010 Esta instrucción describe la suma entre los datos de los dos operandos. El resultado de la ejecución de la suma es guardado en el operando destino.

3. **SUB destino, origen** Código de operación: 0011 Esta instrucción describe la resta entre los datos de los dos operandos. El resultado de la ejecución de dicha resta es guardado en el operando destino.

4. **DIV destino, origen** Código de operación: 0111 Esta instrucción describe la división entre el dato en el operando destino como dividendo y el dato en el operando origen como divisor. El resultado de la ejecución de la división es guardado en el operando destino.

5. **MOV destino, origen** Código de operación: 0001 Esta instrucción describe la copia de datos del dato alojado en el operando origen al operando destino. El resultado de la ejecución del MOV es el dato guardado en el operando origen ahora también guardado en el operando destino.

6. **AND destino, origen** Código de operación: 0100 Esta instrucción describe la operación lógica ”z”bit a bit entre los datos de los dos operandos. El resultado de la ejecución de esta operación es guardado en el operando destino.

7. **CMP destino, origen** Código de operación: 0110 Esta instrucción describe la resta entre dos operandos, sin guardar el resultado. Su único efecto es la actualización de flags en la cpu.

8. **OR destino, origen** Código de operación: 0101 Esta instrucción describe la operación lógica .º”bit a bit entre los datos de los dos operandos. El resultado de la ejecución de esta operación es guardado en el operando destino.

A.3.2. Instrucciones de 1 operando origen (falta revisar Mara)

El formato de instrucción de un operando origen es el siguiente:

4Formato de Instrucción CodOp (4b)	Relleno (000000)	Modo Origen (6b)	Operando Origen (16b)

1. **CALL origen** Código de operación: 1011 El efecto del CALL es guardar la dirección de memoria en la celda de la dirección que se encuentra guardada en el SP (Stack pointer) aumentar el SP y guardar en el PC (Program Counter) el dato que se encuentra guardado en el operando origen ya que describe el llamado a una subrutina que comienza en la celda de memoria cuya dirección esta guardada en el operando origen.
2. **JMP origen** Código de operación: 0110 El efecto del JMP es cambiar el PC (Program Counter) por el dato que esta guardado en el operando origen ya que esta operación describe el salto a otra parte de la memoria para continuar con la ejecución del programa.

A.3.3. Instrucciones de 1 operando destino (falta revisar Mara)

El formato de instrucción de un operando destino es el siguiente:

4Formato de Instrucción CodOp (4b)	Modo Origen (6b)	Relleno (000000)	Operando Origen (16b)

1. **NOT destino** Código de operación: 1001 Esta instrucción describe la operación lógica "negación"bit a bit en el datos del operando destino. El resultado de la ejecución de esta operación es guardado en la misma celda o registro de donde es leído el dato inicialmente.

A.3.4. Instrucciones sin operandos (falta revisar Mara)

El formato de instrucción sin operandos es el siguiente:

2Formato de Instrucción CodOp (4b)	Relleno (000000000000)

1. **RET** Código de operación: 0110 El efecto del ret es cambiar el pc por el dato que esta guardado en la celda de memoria cuya direccion se encuentra en el SP (Stack pointer) y decrementar el SP ya que describe la finalización de la ejecución de una subrutina y la ejecución del resto del programa.

A.3.5. Instrucciones de salto condicional (falta revisar Mara)

El formato de instrucción de salto condicional es el siguiente:

Cod_Op (8)	Desplazamiento(8)

donde los primeros cuatro bits del campo `Cod_Op` es la cadena 1111. Si **al evaluar la condición de salto** el resultado es 1, se le suma al PC el valor del desplazamiento, representado en 8. En caso contrario la instrucción no hace nada.

El efecto de cualquier salto condicional es aumentar el PC (Program Counter) en la cantidad de celdas que indique el desplazamiento si sólo la condición que cada salto condicional tiene da como resultado 1, lo cual es interpretado como verdadero.

1. **JE desplazamiento** Código de operación: 0001 La condición del salto es que el flag **Z** (Cero) sea 1, es decir la ultima operación matemática dió como resultado el número cero.

2. **JNE desplazamiento** Código de operación: 1001 La condición del salto es que el flag **Z** (Cero) sea 0, es decir la ultima operación matemática no dió como resultado el número cero.

3. **JLE desplazamiento** Código de operación: 0010 La condición del salto es el resultado de la siguiente operación lógica **Z OR (N XOR V)**, es decir la ultima operación matemática es menor o igual con signo.

4. **JG desplazamiento** Código de operación: 1010 La condición del salto es el resultado de la siguiente operación lógica **NOT (Z OR (N XOR V))**, es decir la ultima operación matemática es mayor con signo.

5. **JL desplazamiento** Código de operación: 0011 La condición del salto es el resultado de la siguiente operación lógica **N XOR V**, es decir la ultima operación matemática es menor con signo.

6. **JGE desplazamiento** Código de operación: 1011 La condición del salto es el resultado de la siguiente operación lógica **NOT (N XOR V)**, es decir la ultima operación matemática es mayor o igual con signo.

7. **JLEU desplazamiento** Código de operación: 0100 La condición del salto es el resultado de la siguiente operación lógica **C OR Z**, es decir la ultima operación matemática es menor o igual sin signo.

8. **JGU desplazamiento** Código de operación: 1100 La condición del salto es el resultado de la siguiente operación lógica **NOT (C OR Z)**, es decir la ultima operación matemática es mayor sin signo.

9. **JCS desplazamiento** Código de operación: 0101 La condición del salto es que el flag **C** sea 1, es decir la ultima operación matemática es menor sin signo.

10. **JNEG desplazamiento** Código de operación: 0101 La condición del salto es que el flag **N** sea 1, es decir si el último resultado de una operación dio negativo.

11. **JVS desplazamiento** Código de operación: 0111 La condición del salto es que el flag **V** sea 1, es decir si el último resultado de una operación dio overflow.

B. Como utilizar el simulador

En esta sección debemos mostrar como se arranca la aplicación y como se carga un programa .qsim con algunos pantallazos
https://github.com/molinarirosito/QSim
https://github.com/molinarirosito/QSim-UI

C. Errores comunes

C.1. Errores de Sintaxis

En esta seccion se detallan las diferentes situaciones que pueden dar como resultado un mensaje de error, que informará en que línea del programa se encuentra, durante la etapa de ensamblado.

- Dado el siguiente programa Qi:

```
ADD R0, 0x0002
MUL R4, 0x01
SUB R5, 0x000A
MOV R5, 0x0056
MOV R2, R3
ADD R1, R7
```

  En la linea numero 2 el modo de direccionamiento inmediato esta incompleto, (le faltan dos dígitos). Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up[8] el siguiente mensaje:

  "Ha ocurrido un error en la linea 2 : MUL R4, 0x01"

- Dado el siguiente programa Qi:

```
MOV 0x0006, 0x0056
ADD R2, R3
SUB R1, R7
```

  En la linea numero 1 el operando destino es inmediato lo cual es invalido (El operando nunca puede tener como modo de direccionamiento un inmediato). Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

  "Ha ocurrido un error en la linea 1 : MOV 0x0006, 0x0056

- Dado el siguiente programa escrito en Q1:

[8] Ventana emergente.

```
1.ADD R0, [0x0002]
2.MOV R4, R0
```

En la linea numero 1 el operando origen es directo lo cual es invalido en la arquitectura Q1 (El modo de direccionamiento Directo se incorpora en las arquitecturas Qi desde la arquitectura Q2 en adelante). Cuando el alumno quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1 : ADD R0, [0x0002]

- Dado el siguiente programa Qi:

```
CMP R3, [0xA000]
MOV R4 R0
\begin{verbatim}

En la línea numero 2 entre los operandos no se encuentra la coma que los separa (La s

\mensaje{"Ha ocurrido un error en la linea 2 : MOV R4, R0"} \\

\item Dado el siguiente programa Qi:\\

\begin{verbatim}
sub [[0x0004]], [0xA000]
ADD R4, R0
```

En la linea numero 1 la instrucción sub esta escrita en minúscula esto es inválido (La sintáxis define que los nombres de las instrucciones son estrictamente en mayúscula). Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1: sub [[0x0004]], [0xA000]

- Dado el siguiente programa Qi:

```
MUL [R6], r4
ADD [0xF0F0], R0
```

En la linea numero 1 el operando origen es un registro escrito con minúscula, esto es invá lido(La sintáxis define que los registros empiezan estrictamente con R mayúscula). Cuando se quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1: MUL [R6], r4

Dado el siguiente programa Qi:

```
AND R2, R8
OR [0xF0F0], R0
```

En la linea numero 1 el operando origen el numero del registro es invalido (Los registros deben estar dentro del rango R0..R7). Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1: AND R2, R8

Dado el siguiente programa Qi:

```
MUL R7, R4
AND R5, [R3]
```

Para la instrucción MUL es inválido utilizar como destino el registro R7 por lo que, la primer línea es inválida. Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1: MUL R7, R4

Dado el siguiente programa Qi:

```
inicio: MUL R7, R4
AND R5, [R3]
JMP incio
```

En la linea numero 3 la etiqueta anteriormente declarada en la línea número 1 esta incompleta (Le faltan los dos puntos). Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 3: JMP incio"

Dado el siguiente programa Qi :

```
ADD [0x9000], R4
NOT 0x0004
```

En la linea numero 2 el operando destino de la instrucción NOT no puede ser un inmediato, esto es inválido (Los operandos destinos no pueden ser inmediatos). Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 2: NOT 0x0004

Dado el siguiente programa Qi:

```
ADD [9000], R4
NOT R2
```

En la linea numero 1 el operando origen no tiene el prefijo "0x", es una expresión invalida. Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1: ADD [0009], R4

Dado el siguiente programa Qi :

```
ADD [0x900000000000], R4
NOT R2
```

En la linea numero 1 el operando destino tiene mas dígitos que los permitidos, (Un inmediato tiene el prefijo 0x y luego sólo 4 dígitos hexadecimales). Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1: ADD [0x900000000000], R4

Dado el siguiente programa Qi :

```
ZDD [0x900000000000], [R5]
NOT R2
```

En la linea numero 1 el nombre de la operación es invalida (No existe la instrucción ZDD).Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1: ZDD [0x900000000000], R4"

Dado el siguiente programa Qi :

```
SUB [], 0x000A
```

En la linea numero 1 el operando destino no esta incompleto (El modo de direccionamiento directo debe escribirse como un valor inmediato encerrado entre corchetes). Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1: SUB [], 0x000A

Dado el siguiente programa Qi :

```
JMP
```

En la linea numero 1 sólo esta escrito el nombre de la instruccion JMP pero falta a continuación su el operando origen. Cuando se quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla se mostrará mediante un pop-up el siguiente mensaje:

"Ha ocurrido un error en la linea 1: JMP

Referencias

[1] Williams Stallings, *Computer Organization and Architecture*, octava edición, Editorial Prentice Hall, 2010.

[2] A. Tanenbaum, *Organización de Computadoras*, cuarta edición, Editorial Pearson.

[3] Hennessy, Patterson. *Arquitectura de Computadores - Un enfoque cuantitativo*, primera edición, Editorial Mc Graw Hill.

[4] Sitio oficial de la materia Organización de Computadoras: `http:\orga.blog.unq.edu.ar` (2013)