



## Simulador de Arquitecturas Q

Susana Rosito      Tatiana Molinari

10 de noviembre de 2013

# Índice

<b>I Contexto</b>	<b>4</b>
1. Sobre la materia Organización de Computadoras(susi)	4
2. Conceptos importantes	4
2.1. Enfoque de Von Neumann . . . . .	4
2.2. Organización de la computadora . . . . .	5
2.3. Ejecución de un programa . . . . .	6
3. Estado del arte	7
4. Arquitecturas Q (Susi)	7
4.1. Q1 . . . . .	8
4.2. Q2 . . . . .	8
4.3. Q3 . . . . .	8
4.4. Q4 . . . . .	9
4.5. Q5 . . . . .	9
4.6. Q6 . . . . .	10
<b>II Simulador QSim</b>	<b>10</b>
5. Funcionalidad del simulador	10
5.1. Chequeo de sintaxis . . . . .	10
5.2. Ensamblado . . . . .	11
5.3. Cargado en memoria (falta revisar Mara) . . . . .	11
5.4. Ejecución paso a paso (falta revisar Mara) . . . . .	11
6. Implementación	12
6.1. Tecnología utilizada (tati) . . . . .	12
6.2. Diseño Orientado a Objetos (falta revisar Mara) . . . . .	13
6.2.1. ALU . . . . .	13
6.2.2. Bus de entrada y salida, memoria y puertos . . . . .	13
6.2.3. CPU . . . . .	14
6.2.4. Interprete . . . . .	14
6.2.5. Modos de direccionamiento y W16 . . . . .	15
6.2.6. Instrucciones . . . . .	16
6.2.7. Programa . . . . .	17
6.2.8. Simulador . . . . .	17
<b>III Evaluación del desarrollo</b>	<b>17</b>
7. Dificultades encontradas	18
7.1. Dificultades presentadas por el dominio . . . . .	18
7.2. Dificultades de diseño . . . . .	18

<b>8. Casos de prueba</b>	<b>21</b>
8.1. Chequeo de sintaxis en la distintas Qi . . . . .	21
8.2. Ensamblado (Objetos $\rightarrow$ Código Maquina) . . . . .	21
8.3. Decodificación Código Maquina $\rightarrow$ Objetos (Interprete) . . . . .	21
8.4. Ejecución . . . . .	22
8.4.1. Operaciones de ALU . . . . .	22
8.4.2. Búsqueda de Operandos . . . . .	23
8.4.3. Almacenamiento de Operandos . . . . .	23
<b>9. Ejemplos de uso</b>	<b>23</b>
9.1. ERROR DE SINTAXIS . . . . .	23
9.2. PARA EXPERIMENTAR . . . . .	29
<b>10. Trabajo Futuro</b>	<b>30</b>
<b>A. Especificación de la arquitectura Q</b>	<b>30</b>
A.1. Características generales . . . . .	30
A.2. Modos de direccionamiento . . . . .	30
A.3. Repertorio de instrucciones . . . . .	32
A.3.1. Instrucciones de 2 operandos . . . . .	32
A.3.2. Instrucciones de 1 operando origen (falta revisar Mara) . . . . .	33
A.3.3. Instrucciones de 1 operando destino (falta revisar Mara) . . . . .	33
A.3.4. Instrucciones sin operandos (falta revisar Mara) . . . . .	33
A.3.5. Instrucciones de salto condicional (falta revisar Mara) . . . . .	33

## Resumen

Este trabajo...

# Parte I

## Contexto

1. Sobre la materia Organización de Computadoras(susi)
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, iteraciones 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 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** y el **registro IR**. 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 contiene el código máquina de la instrucción actual una vez que la misma es leída de memoria para luego decodificarla y ejecutarla. **Falta hablar del SP y los flagsssss**

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 están 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

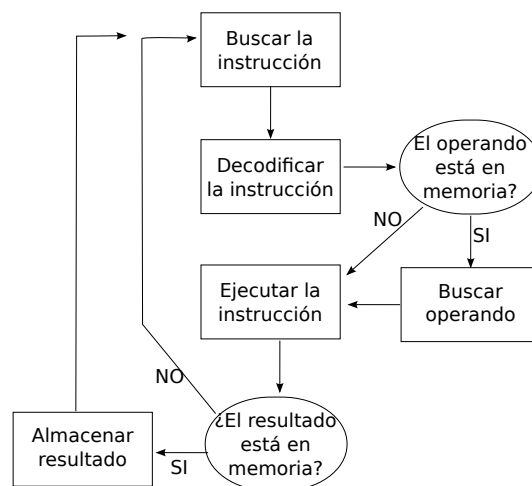


Figura 1: ciclo de ejecución de instrucción

Al principio de cada ciclo de ejecución de instrucción se 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 instrucción la Unidad de Control realiza 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 más reducido, con un repertorio de instrucciones y modos de direccionamiento más 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, son como capas de cebolla. 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 más mínimo detalle, a través de la ejercitación del ciclo de ejecución de instrucción.

## 4. Arquitecturas Q (Susi)

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 Qi agrega más fun-

cionalidad (ya sean instrucciones nuevas o modos de direccionamiento) a las arquitecturas Qi anteriores.

Incorporar lo que sigue a un gráfico de la estructura en cebolla<sup>1</sup>

#### 4.1. Q1

##### Modos de direccionamiento

1. Inmediato
2. Registro

##### Instrucciones

1. MOV
2. SUB
3. DIV
4. ADD
5. MUL

#### 4.2. Q2

##### Modos de direccionamiento

1. Modos de direccionamiento **Q1**
2. Directo

##### Instrucciones

1. Instrucciones **Q1**

#### 4.3. Q3

##### Modos de direccionamiento

1. Modos de direccionamiento **Q2**

##### Instrucciones

1. Instrucciones **Q2**
2. CALL
3. RET

---

<sup>1</sup>



#### 4.4. Q4

##### Modos de direccionamiento

1. Modos de direccionamiento **Q3**

##### Instrucciones

1. Instrucciones **Q3**
2. CMP
3. JMP
4. JE
5. JNE
6. JLE
7. JG
8. JL
9. JGE
10. JLEU
11. JGU
12. JCS
13. JNEG
14. JVS

#### 4.5. Q5

##### Modos de direccionamiento

1. Modos de direccionamiento **Q4**
2. Indirecto
3. RegistroIndirecto

##### Instrucciones

1. Instrucciones **Q4**

## 4.6. Q6

### Modos de direccionamiento

#### 1. Modos de direccionamiento Q5

#### Instrucciones

##### 1. Instrucciones Q5

2. AND

3. OR

4. NOT

## Parte II

# Simulador QSim

## 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 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 (falta revisar Mara)

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. 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 instrucción.
- 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 (combinacion de modos, codigos, incorrecta) y el alumno verá el mensaje de error pertinente.

Durante la carga del programa en memoria puede producirse un `OutOfMemoryError?`, que significa que el programa no entra en la ubicación elegida en memoria ya que ocupa más celdas que las que se encuentran disponibles debajo de ella ya que como se mencionó en la sección ??, la memoria disponible tiene un tamaño limitado y por este motivo la alocaión 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 (falta revisar Mara)

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:

- Tomar un dato de un sector de memoria equivocado.
- Que el programa sobrescriba su mismo código máquina.

- Permitir que la ejecución continúe una vez terminado el programar cargado en memoria.

■

El paso a paso que provee el simulador consiste en las siguientes etapas pertenecientes al ciclo 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 primera 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 máquina 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 Interpretador se encarga de ensamblar el código máquina (abreviado en hexadecimal) que ya fue ubicado en el registro IR para mostrar el código fuente de la instrucción actual con sus respectivos operandos. Si el programa escrito por el alumno es sintácticamente y conceptualmente correcto, este paso le permite comprobar que la instrucción actual es la que él mismo escribió y no otra, visualizándola 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 intérprete 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 descrita 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 (tati)

- **Scala** Elegimos el lenguaje Scala para realizar el simulador ya que, en la materia llamada Objetos III, lo utilizamos un pequeño lapso de tiempo y creímos que era una buena oportunidad para, en vez de elegir un lenguaje que hayamos utilizado más en la carrera como Java, profundizar en la utilización de Scala y aprovechar las ventajas que este ofrecía al combinar el manejo de objetos y las características de un lenguaje funcional.
- **Arena** Utilizamos el framework Arena para realizar la interfaz de usuario del simulador porque es un framework de código abierto, que también

pudimos utilizar en una de las materias y al poder ser conuinado con Scala nos pareció una buena oportunidad de explotar lo que nos ofrecia para que este simulador en su totalidad sea de codigo abierto.

- **Eclipse** Se eligio utilizar el entorno de programación Eclipse ya que ambas trabajamos en diferentes sistemas operativos para los cuales Eclipse es funcional y puede tener los mismos pluggins que hace que pueda sopotar proyectos MVN y Scala, además de que, al ser el entorno de programación que más hemos usado nos sentiamos cómodas con esa elección.
- **Git** Elegimos git como repositorio externo para sincronizar nuestro codigo ya que promueve el codigo abierto (ACA QUIERO PONER LO DE QUE ESTA AHI ARRIBA Y NO LE CEDEMOS DERECHOS NI NADA A NADIE PERO NO ME SALE!!!)

## 6.2. Diseño Orientado a Objetos (falta revisar Mara)

### 6.2.1. ALU

Como se observa en la figura 2 la ALU tiene toda la responsabilidad en la ejecución de operaciones matematicas y logicas, además del analisis sobre los flags luego de cada operación.

ALU
<pre> +execute_operacion_matematica()(operacion:(Int, Int) =&gt; Int,op1: W16,op2:W16): Map[String, Any] +takeFlags(valor:Int): (Int, Int) +takeFlagsSum(resultado_binario:W16,op1:W16,op2:W16): (Int,Int) +takeFlagsRest(resultado_binario:W16,op1:W16,op2:resultado_binario): (Int,Int) +execute_add(op1:W16,op2:W16): Map[String, Any] +execute_sub(op1:W16,op2:W16): Map[String, Any] +execute_mul(op1:W16,op2:W16): Map[String, Any] +execute_div(op1:W16,op2:W16): Map[String,Any] +execute_cmp(op1:W16,op2:W16): Map[String, Any] +execute_operacion_mul(operacion:(Int, Int) =&gt; Int,op1:W16,op2:W16): Map[String, Any] +actualizarNegative(resultado:Int): Int +actualizarZero(resultado:Int): Int +actualizarCarryBorrow(resultado_binario:W16): Int +obtenerBitsParaAnalizarOverflow(resultado_binario:W16,op1:W16,op2:W16): (Int,Int,Int) +verificarCondicionOverflowSuma(resultado_binario:W16,op1:W16,op2:W16): Int +verificarCondicionOverflowResta(resultado_binario:W16,op1:W16,op2:W16): Int +aplicarOperacionBooleana(op1:W16,op2:W16,operacion:(Int, Int) =&gt; Int): W16 +actualizarFlagsOperacionesLogicas(resultado:W16): Map[String, Any] +AND(op1:W16,op2:W16): Map[String, Any] +XOR(op1:W16,op2:W16): Map[String, Any] +OR(op1:W16,op2:W16): Map[String, Any] +NOT(op:W16): W16 +AND(un_bit:Int,otro_bit:Int): Int +OR(un_bit:Int,otro_bit:Int): Int +XOR(un_bit:Int,otro_bit:Int): Int +NOT(un_bit:Int): Int +interpretarBit(un_bit:Int): Boolean </pre>

Figura 2: 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 modificación de una celda o el leer un dato. Para ello conoce a una instancia de la clase

Memoria y a otra de la clase CeldasPuertos. Ambas clases conocen muchas instancias de la clase Celda, y cada Celda a su vez conoce un dato: una instancia del W16, que representa a los datos guardados en memoria o en los puertos.

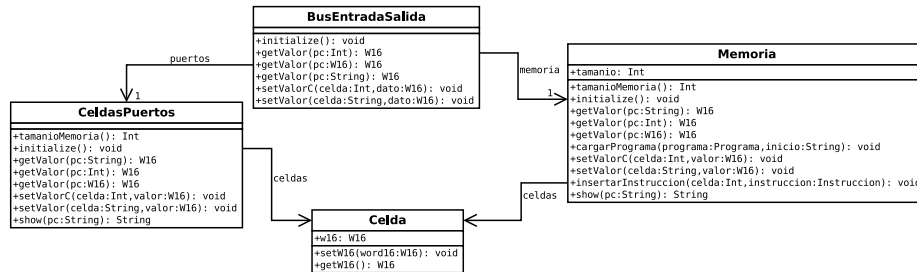


Figura 3: Diagrama de clase del la BusEntradaSalida\_Memoria\_CeldasPuertos

### 6.2.3. CPU

Como se observa en la figura 4, la CPU conoce a la ALU, contiene los registros IR y PC, y los flags (V,Z,C,N) y los ocho registros (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.

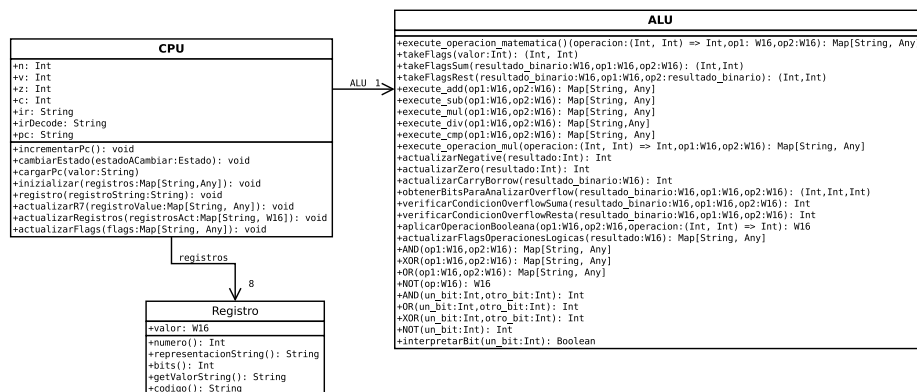


Figura 4: Diagrama de clase de la CPU

### 6.2.4. Interprete

Como se observa en la figura 5 el Interprete es un singleton tiene la entera responsabilidad de recibir los datos leídos desde, una celda de memoria hasta tres celdas, y bit a bit, según los códigos de operación y de modo de direccionamiento saber de que instrucción se trata y cuales son sus operandos si es que los tiene y devolver el objeto que la representa. Se ocupa del decode de la instrucción.

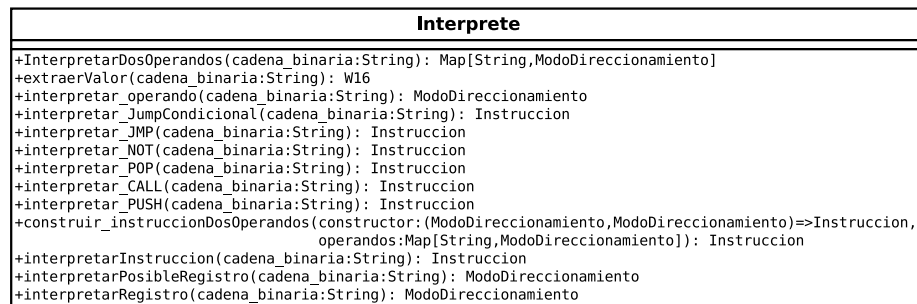


Figura 5: Diagrama de clase del Interprete

### 6.2.5. Modos de direccionamiento y W16

Como se observa en la figura 6 los modos de direccionamiento extienden del trait `ModoDireccionamiento`, donde se encuentran declarados mensajes necesarios para majear todas las subclases poliformicamente. Entre los cuales se encuentran los mensajes:

- **representacionString** Que devuelve la representación en string como codigo fuente, por ejemplo la representación de un `ADD(R0,R7)`, sería: `ADD R0, R7`
- **codigo** Retorna el string que representa al código único de modo de direccionamiento, por el ejemplo, el código de modo de direccionamiento del R7 es 100111.
- **getValorString** Retorna el string que representa al dato que posee el modo de direccionamiento. En el caso de un `Inmediato` que sea `FF56`, devolvera el string `"FF56"`, y en el caso de cualquier registro, retornara el valor que represente su W16.

Los modos de direccionamiento diferentes a `Inmediato` y `Registro`, conocen en vez de un W16 otro modo de direccionamiento según corresponda:

- `RegistroIndirecto` conoce un a instancia de `Registro`.
- `Directo` conoce un a instancia de `Inmediato`.
- `Indirecto` conoce un a instancia de `Directo`.

Esto se implemento de esta manera para que el leer datos de memoria, puertos o registros, o guardarlos en los mismos sea más sencillo ya que se delega en el modo de direccionamiento que conoce.

La clase `Etiqueta` representa la etiqueta creada por el alumno cuando realiza el programa. Cuando el mismo es cargado en memoria, según cual sea la celda de inicio y cuanto ocupen las instrucciones, se calcula la dirección de memoria que representa la etiqueta y luego se descarta reemplazandola por un modo de direccionamiento `Inmediato`.

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

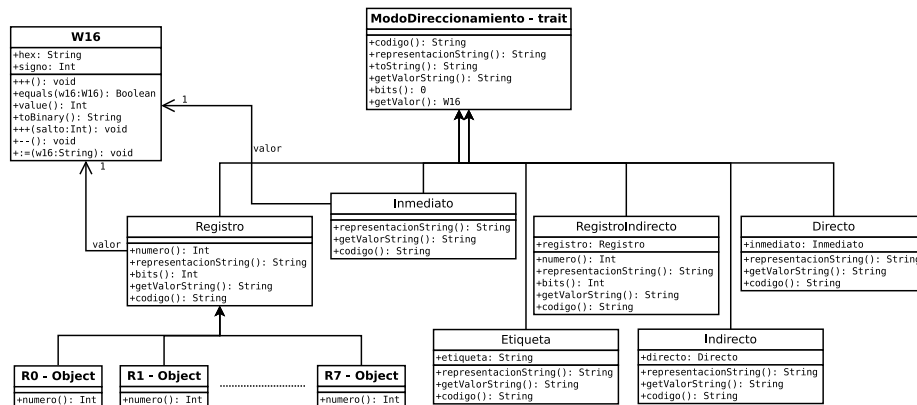


Figura 6: Diagrama de clase de la jerarquía de los modos de direccionamiento

### 6.2.6. Instrucciones

Como se observa en la figura 7 las Instrucciones estan modeladas en jerarquias que se dividen en:

- Un operando.
- Dos operandos.
- Sin operandos.
- JumpConducional.

Se realizo dicha jerarquía para permitir la fácil agregación de nuevas instrucciones siendo subclases de la clase que corresponda ya que comparten comportamiento tal como manera de mostrarse y de codificarse.

**Instrucciones sin operandos** Como se observa en la figura ?? la unica instrucción sin operandos implementada es la instruccion RET, a pesar de esto, se eligió hacer una jerarquía para que luego se facilite **la inserción al modelo de nuevas instrucciones sin operandos**<sup>2</sup>.

**Instrucciones de un operando** Como se observa en la figura ?? y concordando en lo mencionado sobre las Instrucciones anteriormente, puede haber dos tipos de instrucciones de un operando:

- Un operando origen.
- Un operando destino.

Ambas clases de instrucciones tienen un solo operando. La diferencia entre ellas es la logica de ejecución y la manera en la que se ensamblan y desensamblan ya que sus formato de instruccion difiere en donde esta colocado el relleno.

<sup>2</sup>la escalabilidad del modelo, permitiendo la inserción de nuevas instrucciones sin operandos



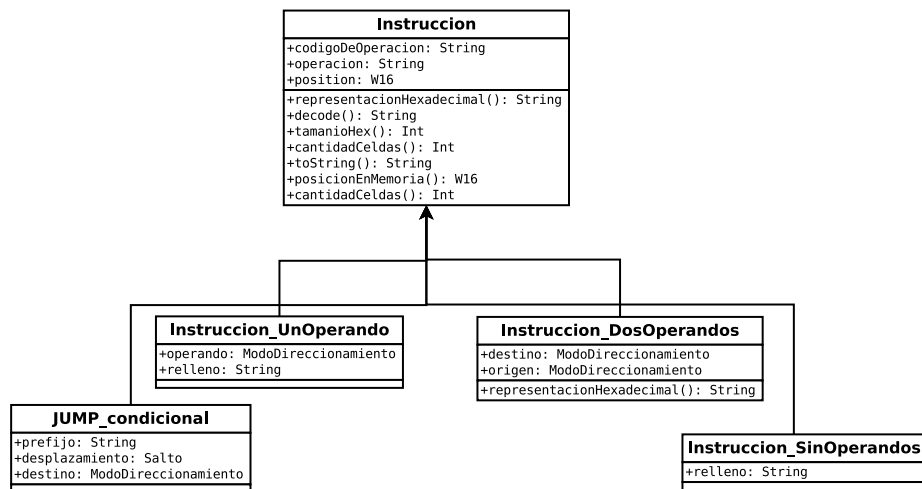


Figura 7: Diagrama de clase de la Instrucción

**Instrucciones de dos operandos** Como se observa en la figura ?? la jerarquía de clases de instrucciones de dos operandos es la más amplia por tener mayor cantidad de instrucciones. Todas comparten la manera de 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, 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.

#### 6.2.8. Simulador

Como se observa en la figura 12 la clase Simulador conoce a una instancia de CPU, a una instancia de BusEntradaSalida y a una instancia de Instruccion que es la instruccion que se esta ejecutando en ese momento. La clase Simulador tiene la responsabilidad de obtener el codigo maquina a traves del Interprete de la siguiente instruccion (fech y decode), calcular las etiquetas de un programa, cargar el programa en memoria, asi mismo, los datos en registros, ejecutar las instrucciones o delegar su ejecución a la ALU que conoce a traves de la CPU según corresponda (realizar el execute), y guardar datos en memoria o registros (Store). Es la clase principal del modelo y la encargada de coordinar la ejecución del programa paso a paso.

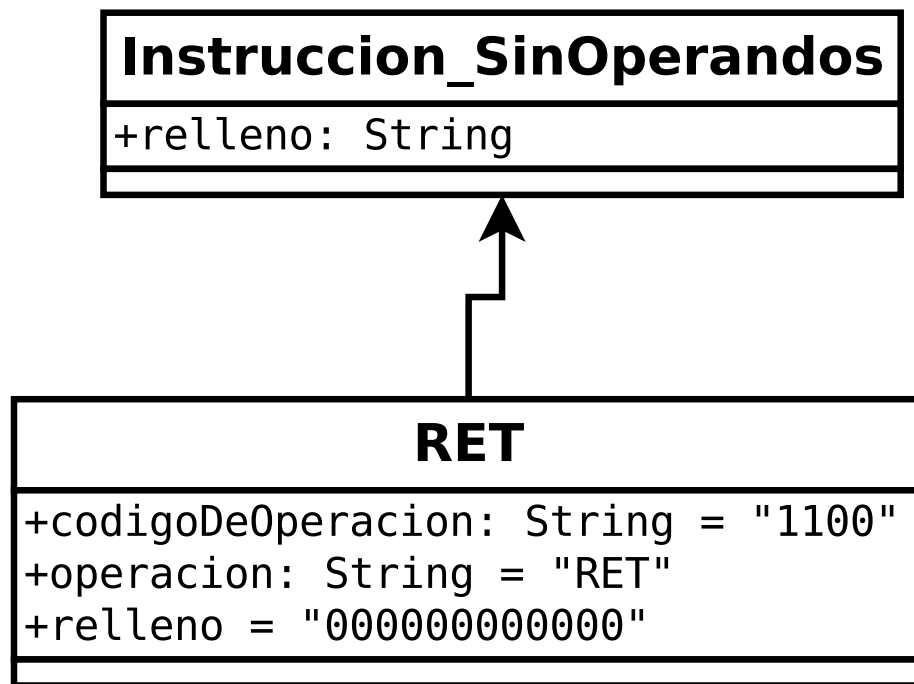


Figura 8: Detalle de clase InstruccionSinOperandos

### 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 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. Procesaba las instrucciones no leyendo de la matriz memoria, si no, pidiendo la siguiente instrucción al objeto instancia de la clase Programa, sobreviviendo así las distintas etapas una vez que fue creado y evitando la creación de un objeto cuya responsabilidad sea interpretar el código máquina alojado en la memoria, evitando la

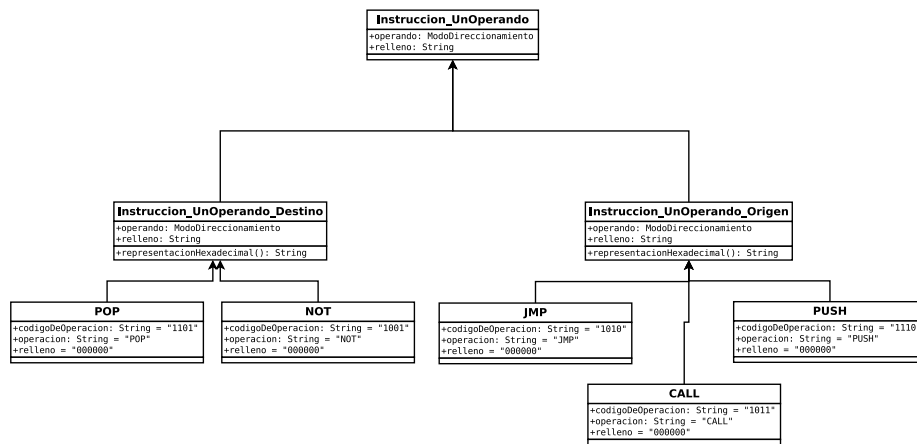


Figura 9: Diagrama de clase de la InstruccionUnOperando

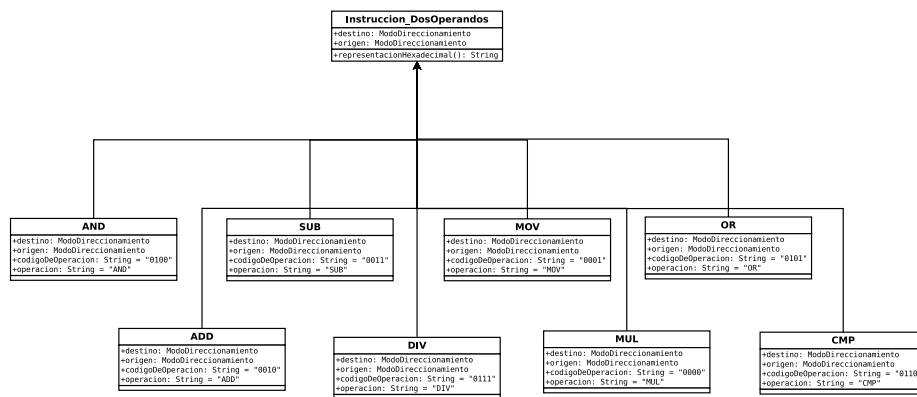


Figura 10: Diagrama de clase de la InstruccionDosOperandos

nueva creacion de instancias de la clase **Instruccion** y haciendo mucho más sencillo al modelo ya que la memoria era sólo una clase que contenia datos que se reflejaban en pantalla y no se extraian datos de celdas en ningun momento. Luego entendimos que un programa 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, inevitablemente se necesitó 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.

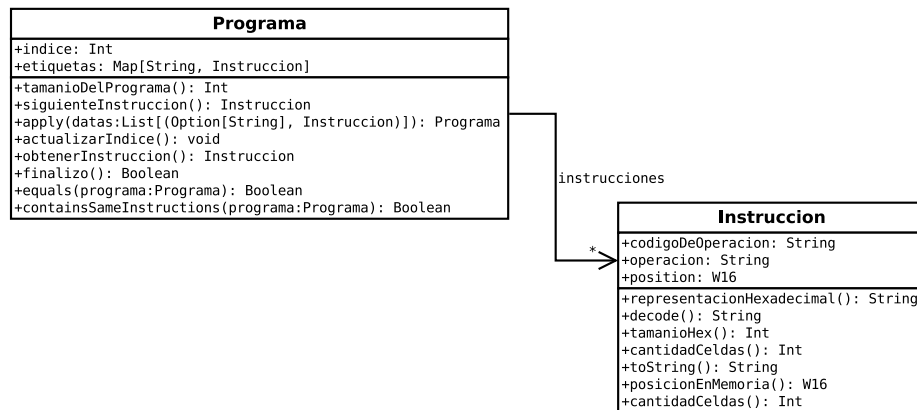


Figura 11: Diagrama de clase de la Programa

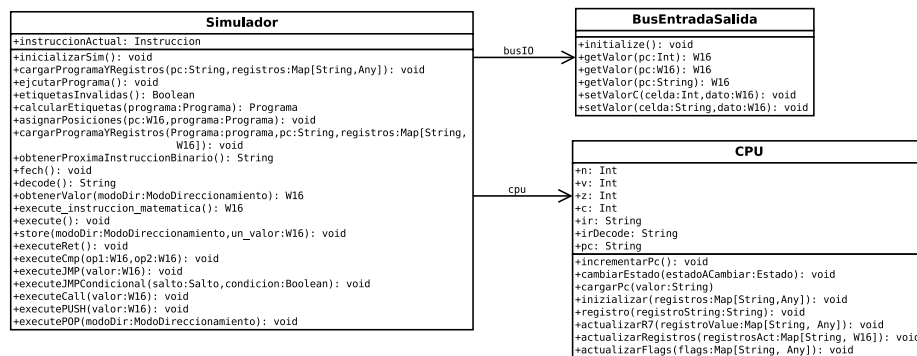


Figura 12: Diagrama de clase de la Simulador

Tuvieron que ser solicitadas además 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 visualizacion de cambios de colores en la memoria luego de las etapas de ejecución y de realizar algun cambio en la misma.
- TextBox multiLine (TextArea) con scroll para ser utilizado como consola de devolucion.
- Icono para la aplicacion, solo por cuestiones estéticas.

## 8. Casos de prueba

### Descripción de cada Caso de Prueba

#### 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 valido 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.

*(esto va en nota al pie)programa Qi valido = Es un programa en cualquier arquitectura Q escrito sintaticamente correcto.*

##### 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 excepcion te muestra el numero de linea y la propia linea para que veas en donde te equivocaste. Capturamos la Excepcion y tomamos como resultado el mensaje y como ultimo paso comparamos el mensaje resultado con el mensaje esperado.

#### 8.2. Ensamblado (Objetos $\rightarrow$ Código Maquina)

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 programa que contiene un conjunto de objetos intrucciones, el ensamblado simplemente lo realiza cada instruccion, esto quiere decir que cada instruccion sabe ensamblarse. De esta forma realizamos el ensamblado del programa y obtenemos una lista de codigo maquina como resultado y como ultimo paso comparamos la lista de codigo maquina resultado con la lista codigo maquina esperado.

#### 8.3. Decodificacion Código Maquina $\rightarrow$ Objetos (Interprete)

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

### 1. Lista de de Instrucciones (decodificacion)

El resultado correcto es la lista de objetos instrucciones, por cada codigo maquina el interprete verifica los codigos de operacion para poder crear las instrucciones correctas y como ultimo paso de la verificacion comparamos la lista resultante con la lista esperada.

### 2. Error

El resultado puede ser un error porque puede pasar que el codigo de operacion o el codigo de algun modo de direccionamiento dentro del formato de cada instruccion sea invalido. La excepcion que puede tirar tiene el nombre de **CodigoInvalidoException**.

## 8.4. Ejecucion

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

#### ■ Fetch

Para verificar el fetch se toma la instruccion 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 instruccion a ejecutar.

#### ■ Decode

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

#### ■ Execute

Para verificar el efecto de cada instruccion se considero armar un caso de prueba por cada instruccion donde cada uno tiene un estado inicial y luego de realizar la ejecucion de dicha instruccion 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 instruccion

### 8.4.1. Operaciones de ALU

La ALU es la encargada de ejecutar operaciones aritmeticas/logicas. Si la instruccion a ejecutar tiene un efecto aritmetico/logico se le delega la ejecucion de la operacion. Para poder verificar todas las operaciones creamos un caso de prueba por cada operacion. Tanto las operaciones aritmeticas como las logicas se analizan de la misma forma:

1. Se toma cada valor de los operandos (la búsqueda se detalla en 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 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 inválidas 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

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

### 9.1. ERROR DE SINTAXIS

- Enfoquémonos en esta situación, que pasaría si un alumno escribe el siguiente programa Qi:
  - 1.ADD R0, 0x0002
  - 2.MUL R4, 0x01
  - 3.SUB R5, 0x000A
  - 4.MOV R5, 0x0056
  - 5.MOV R2, R3
  - 6.ADD R1, R7

En la línea número 2 el modo de direccionamiento inmediato está incompleto, (le faltan dos dígitos). Cuando el alumno quiera ensamblar este programa, el ensamblador detectará este error y en la pantalla recibirá el error descriptivo como el siguiente:

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

- Otra situacion seria si un alumno escribe el siguiente programa Qi:

```
1.MOV 0x0006, 0x0056
2.ADD R2, R3
3.SUB R1, R7
```

En la linea numero 1 el operando destino es inmediato lo cual es invalido<sup>3</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Una situacion peculiar puede ser si un alumno escribe el siguiente programa Q1:

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

En la linea numero 1 el operando destino es directo lo cual es invalido en Q1<sup>4</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Una situacion peculiar puede ser si un alumno escribe el siguiente programa Qi:

```
1.CMP R3, [0xA000]
2.MOV R4 R0
```

En la linea numero 2 entre los operandos no se encuentra la coma que los separa, eso es invalido<sup>5</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

”Ha ocurrido un error en la linea 2 : MOV R4, R0”

- Algo comun que podria ocurrir es si un alumno escribe el siguiente programa Qi:

---

<sup>3</sup>El operando nunca puede tener como modo de direccionamiento un inmediato

<sup>4</sup>El modo de direccionamiento no es parte de Q1

<sup>5</sup>las instrucciones de dos operandos tiene que tener la coma para separarlos



```
1.sub [[0x0004]], [0xA000]
2.ADD R4, R0
```

En la linea numero 1 la instruccion sub se escribio en minuscula esto es invalido<sup>6</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- veamos que pasa si un alumno escribe el siguiente programa Qi:

```
1.MUL [R6], r4
2.ADD [0xF0F0], R0
```

En la linea numero 1 el operando destino es un registro que empieza con minuscula, esto es invalido<sup>7</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Otra situacion que se podria dar es si un alumno escribe el siguiente programa Qi:

```
1.AND R2, R8
2.OR [0xF0F0], R0
```

En la linea numero 1 el operando destino el numero del registro es invalido<sup>8</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Este es un error de concepto que se le puede escapar a los alumnos, que pasa si un alumno escribe el siguiente programa Qi:

```
1.MUL R7, R4
2.AND R5, [R3]
```

---

<sup>6</sup>Los nombres de las instrucciones son estrictamente en mayuscula

<sup>7</sup>Los registros empiezan estrictamente con Mayuscula

<sup>8</sup>Los registros estan en el rango R0..R7, por lo tanto R8 no pertenece al rango por eso es invalido

En la linea numero 1 cuando uno escribe la instruccion MUL no es valido utilizar como destino el registro numero 7 por lo tanto es invalida<sup>9</sup> esa linea. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Este es otro error que se les puede escapar a los alumnos, que pasa si un alumno escribe el siguiente programa Qi:

```
1.inicio: MUL R7, R4
2.AND R5, [R3]
3.JMP inicio
```

En la linea numero 3 cuando uno escribe una instruccion utilizando etiquetas no es correcto escribir el nombre incompleto de la etiqueta, por eso es invalido. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Otro error con etiquetas, que pasa si un alumno escribe el siguiente programa Qi:

```
1.MOV [0x0005], etiq
2.CALL [0x0005]
3.etiqueta: ADD R0, 0x0002
```

En la linea numero 1 ocurre el mismo error antes mencionado donde el nombre de la etiqueta esta incompleto, esta situacion es distinta ya que en la anterior uno define primero la etiqueta y luego se utiliza, aca es alreves primero se utiliza en la linea 1 y luego se define en la linea 3. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

"Ha ocurrido un error en la linea 1: MOV [0x0005], etiq"

- Relacionado con las etiquetas puede pasa que se olviden de algun simbolo, que pasa si un alumno escribe el siguiente programa Qi:

```
1.inicio SUB [0x9000], R4
2.MUL R5, [R7]
```

<sup>9</sup>El resultado de la multiplicacion se parte en 2. La primer parte va a R7 y la segunda parte va al operando destino, no pueden ser los mismo porque si no se perderia una parte del resultado.

## 3.JMP inicio

En la linea numero 1 la etiqueta inicio no es una etiqueta valida<sup>10</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

”Ha ocurrido un error en la linea 1: inicio SUB [0x9000], R4”

- Esto es un error de concepto, que pasa si un alumno escribe el siguiente programa Qi :

```
1.ADD [0x9000], R4
2.NOT 0x0004
```

En la linea numero 2 el operando destino de la instruccion NOT no puede ser un inmediato, esto es invalido<sup>11</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Esto es un error que se les puede escarpar, que pasa si un alumno escribe el siguiente programa Qi :

```
1.ADD [9000], R4
2.NOT R2
```

En la linea numero 1 el operando destino no tiene el prefijo 0x por lo tanto es una expresion invalida. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Una cosa interesante puede ocurrir que pasa si un alumno escribe el siguiente programa Qi :

```
1.ADD [0x90000000000000], R4
2.NOT R2
```

En la linea numero 1 el operando destino tiene mas digitos que los permitidos, por esto es invalido<sup>12</sup>. Cuando el alumno quiera ensamblar este

<sup>10</sup>Las etiquetas pueden empezar con mayuscula o minuscula pero al final siempre tienen (;).

<sup>11</sup>La instruccion NOT no puede recibir un inmediato como destino porque necesita guardar el efecto que genera y un inmediato es una constante.

<sup>12</sup>Un inmediato tiene el prefijo 0x y luego 4 digitos hexadecimales.

programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Esto es un error que se les puede escarpar, que pasa si un alumno escribe el siguiente programa Qi :

```
1.ZDD [0x900000000000], [R5]
2.NOT R2
```

En la linea numero 1 el nombre de la operacion es invalida<sup>13</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- Este error no es tan habitual, que pasa si un alumno escribe el siguiente programa Qi :

```
1.SUB [], 0x000A
```

En la linea numero 1 el operando destino no contiene su valor, por eso es invalido<sup>14</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

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

- otro error no tan habitual, que pasa si un alumno escribe el siguiente programa Qi :

```
1.JMP
```

En la linea numero 1 el operando origen de la instruccion JMP no existe y por ende es invalido<sup>15</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

”Ha ocurrido un error en la linea 1: JMP”

- otro error no tan habitual, que pasa si un alumno escribe el siguiente programa Qi :

<sup>13</sup>Ese nombre es invalido porque no es parte del conjunto de instrucciones, no existe la instruccion ZDD.

<sup>14</sup>El modo de direccionamiento directo siempre contiene un inmediato como valor.

<sup>15</sup>Todas las instrucciones son validas si tienen a su operandos, en este caso JMP le falta su operando.

```
1.ADD R7, [[]]
```

En la linea numero 1 el valor el operando origen no existe, esto es invalido<sup>16</sup>. Cuando el alumno quiera ensamblar este programa, el ensamblador detectara este error y en la pantalla recibira el error descriptivo como el siguiente:

”Ha ocurrido un error en la linea 1: ADD R7, [[]]”

## 9.2. PARA EXPERIMENTAR

Esta seccion 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 instruccion ADD R0, 0x0002 y el RET. Cuando vuelve para ejecutar la proxima instruccion el PC se encuentra en la linea 7, donde vuelve a ejecutar la instruccion 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 ejecucion lo primero que se hace es incrementar el SP osea que ahora tiene el valor FF00 y luego buscar el valor de esa direccion para actualizar el registro PC. Para informarles la direccion FF00 es un puerto de E/S. La conclusion es que en esta instancia el flujo de ejecucion 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 alli.

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

```
1.ADD R0, [0x0002]
2.MUL R4, 0x0001
3.SUB [0x0003], 0x000A
```

Pensemos que el PC se posiciona en la linea 3 y realizamos el ciclo de instruccion(FETCH - DECODE - EXECUTE). Como estado final tenemos el efecto de la ultima instruccion y el valor de PC apuntando a la

<sup>16</sup>El modo de direccionamiento indirecto necesita su valor inmediato.

siguiente instrucción a ejecutar. Como verán el programa que escribí el alumno terminó en la línea 3 pero el simulador no para de ejecutar, por ende te permite realizar las veces que quieras el ciclo de instrucción. Es interesante que los chicos experimenten este tipo de cosas.

## 10. Trabajo Futuro

Incorporar push y pop/ mejorar el editor de programas Qi/Entrada y salida: modelar dispositivos como el teclado, etc<sup>17</sup>

### 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<sup>18</sup> - *Stack Pointer* y los Flags de un bit como Negative, overflow, Carry, Zero. También tiene un conjunto de instrucciones que detallaré más 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. La instrucción DIV tiene como efecto [destino (- destino %<sup>19</sup> origen)]. La instrucción MUL<sup>20</sup> tiene como efecto [destino (- destino \* origen)]. Por último tiene una Memoria que tiene direcciones de 16 bit, donde el tamaño de cada celda también es de 16 bit. La Memoria tiene un tamaño 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 de 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.

---

<sup>17</sup>

<sup>18</sup>Comienza en la dirección FFEF.

<sup>19</sup>El carácter % denota el cociente de la división entera.

<sup>20</sup>El resultado de la operación **MUL** ocupa 32 bits, almacenándose los 16 bits menos significativos en el operando destino y los 16 bits más significativos en el registro **R7**.

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 está 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 está 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. **RegistroIndirecto** 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:

Código de Operación (4b)	Modo Destino (6b)	Modo Origen (6b)	Destino (16b)	Origen (16b)
-----------------------------	----------------------	---------------------	------------------	-----------------

Las instrucciones de dos operandos descritas 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**.<sup>21</sup>, 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 "y" 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.

<sup>21</sup>(mara)No entiendo!. Asi esta mejor??



8. **OR destino, origen** Código de operación: 0101 Esta instrucción describe la operación lógica "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:

CodOp + relleno + Modo origen + Origen (4 bits) (000000) (6 bits) (16 bits)

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:

CodOp + Modo origen + relleno + Origen (4 bits) (6 bits) (000000) (16 bits)

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:

CodOp + relleno (4 bits) (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 que 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 es el siguiente: prefijo + CodOp + desplazamiento (1111) (4 bits) (8 bits)

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.

## 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)