Instituto Tecnológico de Costa Rica

Área Académica de Ingeniería en Computadores

Programa de Licenciatura en Ingeniería en Computadores

Curso: CE4301 -Arquitectura de Computadores I

Informe de Proyecto 1

Realizado por:

Daniel Canessa Valverde, 201137483

Felipe Alberto Mejías Loría, 201231682

Profesor:

Jeferson González Gómez

Fecha: Cartago, Abril 7, 2016

Contenido

Documento de diseño de software 3
Requerimientos del cliente 3
Características del programa 5
Software del sistema 6
Ensamblador de código ARM 6
Archivo de texto “Lexical.flex” 7
Código de usuario 7
Directivas JLex 7
Reglas de expresiones regulares 7
Clase “LexicalAnalyzer” 10
Archivo de texto “Syntactic.cup” 12
Clase “SyntacticAnalyzer” 14
Clase “CodeGeneration” 16
Simulador de código ARM 21
Clase “Memory” 22
Clase “Register” 23
Clase “ConditionFlags” 24
Clase “ReserveInstructions” 25
Clase “HashLabel” 26
Clase “Operation” 27
Problemas semánticos 37
Problemas de desarrollo 38
Lista de chequeo de cumplimiento de los requerimientos 39
Metodología de diseño del sistema 40
Metodología de diseño 40
Elección del lenguaje de programación 41
Desarrollo de aplicación 43
Herramientas de ingeniería 50
NetBeans 50
VisualSim 50
JFlex 50
JCup 51
Referencias 52

Documento de diseño de software

Requerimientos del cliente

Se debe diseñar e implementar un simulador de la arquitectura ARMv4, que permita la escritura de código ensamblador, la simulación de este y pueda generar el ensamblado correspondiente.

Existen 3 modos de operación:

1. En este modo se permite la escritura de código ARMv4 en la interfaz gráfica, guardar y cargar archivos de programas con extensión “armv4”.
2. En este modo se genera el ensamblado del código cargado en la interfaz gráfica.
3. En este modo se realiza la simulación del código de la operación 1 y 2.

El sistema a implementar debe compilar el código previo a cualquier simulación o generación, debe realizar un análisis léxico, sintáctico y si es necesario semántico para mostrar errores. No existe limitación en utilizar alguna herramienta que se integre al sistema y realice esto.

En la tabla 1 se muestran las instrucciones que el generador del código y simulador soportan, además de los modos de direccionamiento que están disponibles para cada una.

Tipo de instrucción	Instrucción	Modo de direccionamiento
Procesamiento de datos	AND	• Registro: registro solo • Inmediato: con registro
	EOR	
	SUB	
	RSB	
	ADD	
	ADC	
	SBC	
	RSC	
	CMP	
	CMN	
	ORR	
	MOV	
	LSL	
	ASR	
	RRX	
	ROR	
	BIC	
	MVN	
	MUL	
	MLA	
Memoria	STR	• Registro: registro solo • Inmediato: con registro • Base: offset inmediato y de registro.
	LDR	
	STRB	
	LDRB	
Salto	B	• Relativo a PC
	BL	

Tabla 1. Instrucciones y modos de direccionamiento soportados por el simulador y generador de código.

Los tipos de datos que se deben de manejar son enteros de 8 y 32 bits. Los inmediatos se pueden expresar con números decimales o hexadecimales, siempre que inicia un inmediato decimal tiene el símbolo de #, si es hexadecimal inicia con #0x.

Se deben manejar 16 registros, enumerados del 0 al 15 (R0 – R15). Los registros 14 y 15 corresponden al pc+4 en caso de que se realice un branch y al pc respectivamente.

La memoria debe ser de 2KB, 1KB para el programa y 1KB para los datos. La primera dirección debe ser 0x00 y la última 0x7FF. Se debe proveer la posibilidad al usuario de modificar la memoria manualmente, cualquier introducción errónea en la memoria manualmente es responsabilidad del usuario.

La interfaz gráfica queda a criterio del grupo desarrollador, esta debe mostrar el banco de registros, la memoria, las banderas de la ALU, permitir de alguna manera introducir el código, cargarlo y guardarlo.

Características del programa

- El programa se desarrolló en el editor de código NetBeans 8.1.
- Es compatible con la versión de la máquina virtual de Java 8u71, las versiones de Windows: 7 Service Pack 1, 8, 8.1 y 10, y de Linux Ubuntu 15.04.
- El idioma del programa es inglés, además todo el código fue desarrollado en ese idioma, también se siguen las reglas del Javadoc.
- Se utilizan las bibliotecas open source: TextLineNumber, jflex y cup.

Software del sistema

El desarrollo de la aplicación se dividió en 2 etapas:

- Analizador léxico, sintáctico y generación de código ensamblado.
- Simulador de código ARM.

A continuación, se detalla cada una de estas etapas.

Ensamblador de código ARM

Los ensambladores son programas de computadora que traducen un lenguaje ensamblador a lenguaje máquina. Un ensamblador es un tipo de compilador, el cual toma como su entrada un programa escrito en su lenguaje fuente y produce un programa equivalente escrito en su lenguaje objetivo. El lenguaje fuente es un subconjunto del código ARM, mientras que el lenguaje objeto es el código máquina. El objetivo del ensamblador de código ARM es permitir la correcta traducción a lenguaje máquina del código en ensamblador.

Un compilador se compone de varias etapas que realizan distintas operaciones lógicas. Las fases de un compilador son las siguientes: el análisis léxico, el análisis sintáctico, el análisis semántico y la generación de código. Además estas etapas interactúan con una tabla de símbolos y con un manejador de errores.

Para poder realizar el ensamblador se debe entonces contar como mínimo con un analizador léxico, un analizador sintáctico, un generador de código y un manejador de errores.

Mapeando lo anterior al programa, se debe contar entonces con las siguientes clases:

- Una clase que represente al analizador léxico.
- Una clase que represente al analizador sintáctico.
- Una clase que represente al generador de código.

Además de las clases mencionadas anteriormente, se proponen:

- Una clase que se encargue de generar los analizadores léxicos y sintácticos.
- Un archivo de texto en lenguaje JLex que contiene la estructura abstracta del analizador léxico.

- Un archivo de texto en lenguaje Cup que contiene las reglas sintácticas del analizador sintáctico.
- Una clase Label que contiene el nombre de la etiqueta y el valor de la dirección en la que se encuentra.

Archivo de texto “Lexical.flex”

El archivo de texto contiene la especificación léxica del código ARM. Este archivo contiene las expresiones regulares, junto con las acciones que se tomarán cuando se iguale cada expresión. La especificación léxica está organizada en tres secciones, separadas por la directiva "%%", como se muestra a continuación [4]:

Código de usuario

%%

Directivas JLex

%%

Reglas de expresiones regulares

%%

La directiva "%%" separa las diferentes secciones y debe ser ubicada en el comienzo de la línea.

En la sección de código de usuario se indican: el paquete en el que se desea almacenar el analizador léxico generado y las clases que se van a utilizar, por lo que en esta parte se escribe el paquete en el que se guarda la clase del analizador léxico generado y se importa la clase Symbol, ésta clase es necesaria para que el analizador léxico se logre comunicar con el analizador sintáctico.

La sección de directivas JLex es la segunda parte del archivo de especificación léxica. En esta sección se declaran los diferentes tokens que conforman el lenguaje, un token es una secuencia de caracteres, por lo que cuando se declara cada uno de los tokens también se indica la expresión regular que lo define. Los tokens que se definen se muestran a continuación:

- Digit = [0-9]
- Number = {Digit} {Digit}*
- Letter = [A-Za-z]

- ANDINSTRUCTION = "AND"|"and"
- EORINSTRUCTION = "EOR"|"eor"
- SUBINSTRUCTION = "SUB"|"sub"
- RSBINSTRUCTION = "RSB"|"rsb"
- ADDINSTRUCTION = "ADD"|"add"
- ADCINSTRUCTION = "ADC"|"adc"
- SBCINSTRUCTION = "SBC"|"sbc"
- RSCINSTRUCTION = "RSC"|"rsc"
- CMPINSTRUCTION = "CMP"|"cmp"
- CMNINSTRUCTION = "CMN"|"cmn"
- ORRINSTRUCTION = "ORR"|"orr"
- MOVINSTRUCTION = "MOV"|"mov"
- LSLINSTRUCTION = "LSL"|"lsl"
- ASRINSTRUCTION = "ASR"|"asr"
- RRXINSTRUCTION = "RRX"|"rrx"
- RORINSTRUCTION = "ROR"|"ror"
- BICINSTRUCTION = "BIC"|"bic"
- MVNINSTRUCTION = "MVN"|"mvn"
- MULINSTRUCTION = "MUL"|"mul"
- MLAINSTRUCTION = "MLA"|"mla"
- STRINSTRUCTION = "STR"|"str"
- LDRINSTRUCTION = "LDR"|"ldr"
- STRBINSTRUCTION = "STRB"|"strb"
- LDRBINSTRUCTION = "LDRB"|"ldrb"
- BINSTRUCTION = "B"|"b"
- BGEINSTRUCTION = "BGE"|"bge"
- BLEINSTRUCTION = "BLE"|"ble"
- BLTINSTRUCTION = "BLT"|"blt"
- BGTINSTRUCTION = "BGT"|"bgt"
- BEQINSTRUCTION = "BEQ"|"beq"
- BNEINSTRUCTION = "BNE"|"bne"
- BLINSTRUCTION = "BL"|"bl"

- BLEQINSTRUCTION = "BLEQ"|"bleq"
- BLNEINSTRUCTION = "BLNE"|"blne"
- BLGEINSTRUCTION = "BLGE"|"blge"
- BLLTINSTRUCTION = "BLLT"|"bllt"
- BLGTINSTRUCTION = "BLGT"|"blgt"
- BLLEINSTRUCTION = "BLLE"|"blle"
- REGISTER_ID = "R" | "r"
- PC_ID = "PC" | "pc"
- LR_ID = "LR" | "lr"
- REGISTER_NUMBER = {REGISTER_ID} {Number}
- HEXADECIMAL_IMMEDIATE = ("#0x"|"#0X")[0-9|A-Fa-f]+
- DECIMAL_IMMEDIATE = "#"(-?)[0-9]+
- SPECIAL_CHARACTER = [<>*+.:$()'\"/;`%]
- LABEL_NAME= {Letter} {Letter}*
- SPACE = " "
- NewLine = \n|\r|\r\n

La sección de reglas de expresiones regulares contiene las reglas necesarias para el análisis léxico; las reglas contienen las siguientes partes: el nombre del token, y la acción asociada cuando se encuentra el token. La regla que se define en el archivo indica que en el momento en que se encuentra un token, el analizador léxico devuelva un objeto Symbol al analizador sintáctico. Un objeto Symbol está compuesto por el tipo de token, el número de línea en donde se encontró el token y la palabra que representa. Una regla se define de la siguiente manera:

{ADDINSTRUCTION} {return new Symbol(sym.ADD,yyline,yychar, yytext());}

Las reglas que están implementadas en la especificación léxica son similares a la anterior regla, la única diferencia es el tipo de token que retornan.

Una vez que se define el archivo de especificación léxica, se ejecuta una vez para generar la clase del Analizador Léxico.

Clase “LexicalAnalyzer”

Esta clase representa al analizador léxico, y se genera por medio del archivo de texto que contiene la especificación léxica. Esta clase es autogenerada, por lo que la clase generada contiene las tablas de DFA, un buffer de entrada, los estados léxicos de la especificación, un constructor, y el método de búsqueda de tokens con las acciones proporcionadas por el usuario. Las tablas de DFA se utilizan para ir verificando las secuencias de caracteres, los estados léxicos de la especificación son los distintos tokens que se asignaron en el archivo de especificación léxica. Los métodos y atributos que tienen el prefijo zz son utilizados internamente por la clase, mientras que los que tienen el prefijo yy son utilizados en las acciones que realiza el analizador cuando encuentra un token, y brinda información sobre el análisis léxico. En la figura 1, se puede observar el diagrama UML que define a esta clase.

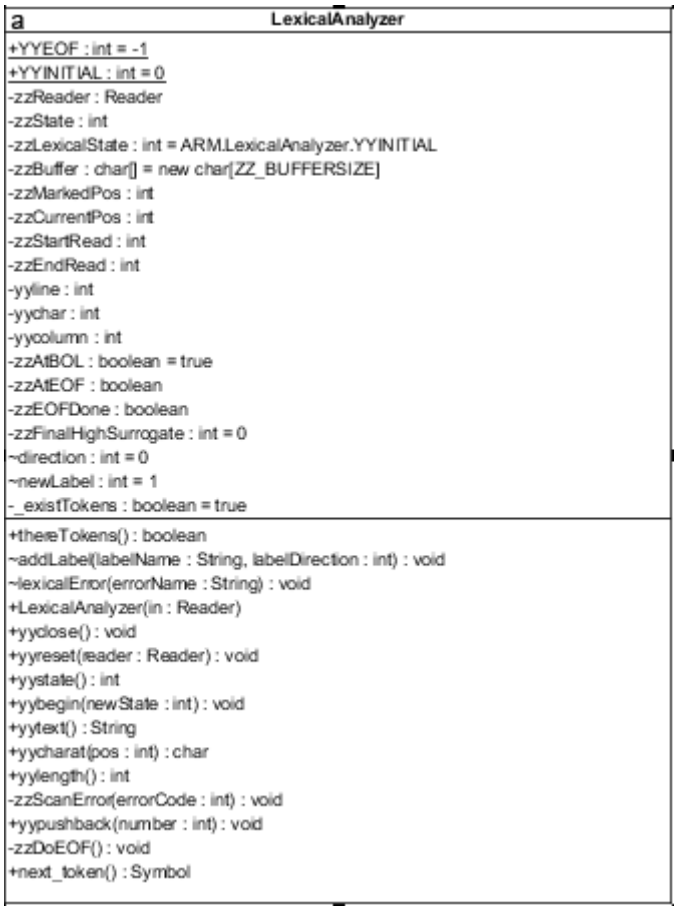

Figura 1. Diagrama UML, clase LexicalAnalyzer.

Los métodos que contiene esta clase se explican a continuación:

- thereTokens: método que no recibe entradas y da como salida un valor booleano que indica si todavía faltan tokens por analizar.
- addLabel: método que recibe como entrada el nombre de la etiqueta y la dirección en la que se encuentra. El método se encarga de crear un objeto llamado Label, el cual va a contener el nombre de la etiqueta y su dirección; luego la agrega a una lista que se encarga de almacenar las etiquetas que se encuentran en el programa.
- lexicalError: método que recibe como entrada el tipo de error léxico y la línea en la que ocurre el error. El método se encarga de almacenar el error en una lista que maneja los errores que ocurren con el programa.
- yyclose: método que no recibe entradas y se encarga de cerrar el flujo de entrada de datos, por lo que a todas las entradas siguientes el analizador va a mostrar fin de archivo.
- yyreset: método que reestablece el analizador léxico para leer de un nuevo flujo de entrada de datos.
- yystate: método que retorna el estado léxico actual del analizador.
- yybegin: método que recibe como entrada el estado léxico en el que debe empezar, por lo que el método cambia el estado léxico actual del analizador por el estado que recibe como parámetro el método.
- yytext: método que no recibe entradas y retorna el lexema perteneciente al token reconocido.
- yycharat: método que recibe como entrada la posición de un caracter de la palabra. El método se encarga de devolver el carácter que se encuentra en esa posición en el valor del token encontrado.
- yylength: método que no recibe entradas y devuelve el tamaño de la cadena que coincidió con el token.
- yypushback: método que recibe como entrada un valor que indica un n número de caracteres. El método considera los n últimos caracteres del token reconocido como no procesados. Estos caracteres no se tienen en cuenta para: yytext() e yylength().
- next_token: El método de búsqueda de tokens no recibe ninguna entrada y consiste en ir analizando cada uno de los caracteres de la entrada hasta

que corresponda con uno de los tokens en la especificación, de lo contrario se produce un error. Si la entrada coincide con un token, se ejecuta la acción correspondiente. Si se llega al final del archivo, el escáner realiza la acción EOF.

Archivo de texto “Syntactic.cup”

El archivo de texto contiene la especificación sintáctica del código ARM. Este archivo contiene las reglas gramaticales, junto con las acciones que se tomarán cuando se evalúe cada expresión. La especificación sintáctica está organizada en cinco secciones, como se muestra a continuación [4]:

- **Definición de paquetes y sentencias import**

  En esta parte de la especificación se declara el paquete en donde se va a guardar la clase generada correspondiente al analizador sintáctico, y se importa la clase java_cup.runtime.*, la cual es necesaria para que el análisis sintáctico se realice en tiempo de ejecución.
- **Sección de código de usuario**

  En esta parte de la especificación se agregan funciones que brindan operaciones complementarias al análisis sintáctico, en este caso se decide no agregar ningún código a esta sección para lograr modularidad en el código y de esta forma la implementación de cada una de las etapas del compilador siga siendo independientes.
- **Declaración de símbolos terminales y no terminales**

  En esta parte del archivo de especificación sintáctica se declaran los símbolos terminales y no terminales que forman parte de la gramática. Las reglas gramaticales determinan las cadenas legales de tokens por medio de derivaciones. Una derivación es una secuencia de reemplazos de nombres de estructura, los nombres de estructuras se refiere a los símbolos no terminales. Los símbolos terminales en la gramática se refieren a los tokens que se definieron en el analizador léxico, mientras que los símbolos no terminales están definidos por la regla gramatical. En esta sección se definieron los siguientes símbolos:

Símbolos terminales:

- AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,CMP,CMN,ORR,MOV
- LSL, ASR, RRX, ROR, BIC, MVN, MUL, MLA
- STR, LDR, STRB, LDRB
- B, BGE, BLE, BLT, BGT, BEQ, BNE
- BL, BLEQ, BLNE, BLGE, BLLT, BLGT, BLLE
- REGISTER;
- DECIMALIMMEDIATE,HEXIMMEDIATE;
- LABEL, PC, LR;
- COMA;
- CORCHETEIZQUIERDO,CORCHETEDERECHO, NEWINSTRUCTION;

Símbolos no terminales:

- assembly_code
- instruction
- register

- **Sección de reglas gramaticales**

  En esta parte del archivo de especificación sintáctica se declaran la gramática que se encarga de determinar las cadenas legales de tokens. La gramática que se utiliza es la siguiente:

```
start with assembly_code;
assembly_code::=instruction assembly_code | instruction;
instruction ::= AND register COMA register COMA register
| AND register COMA register COMA IMMEDIATE
| EOR register COMA register COMA register | EOR register COMA register COMA IMMEDIATE
| SUB register COMA register COMA register | SUB register COMA register COMA IMMEDIATE
| RSB register COMA register COMA register | RSB register COMA register COMA IMMEDIATE
| ADD register COMA register COMA register | ADD register COMA register COMA IMMEDIATE
| ADC register COMA register COMA register | ADC register COMA register COMA IMMEDIATE
| SBC register COMA register COMA register | SBC register COMA register COMA IMMEDIATE
| RSC register COMA register COMA register | RSC register COMA register COMA IMMEDIATE
| CMP register COMA register | CMP register COMA IMMEDIATE
| CMN register COMA register | CMN register COMA IMMEDIATE
| ORR register COMA register COMA register | ORR register COMA register COMA IMMEDIATE
| MOV register COMA register | MOV register COMA IMMEDIATE
| LSL register COMA register COMA register | LSL register COMA register COMA IMMEDIATE
| ASR register COMA register COMA register | ASR register COMA register COMA IMMEDIATE
| RRX register COMA register COMA register | RRX register COMA register COMA IMMEDIATE
| ROR register COMA register COMA register | ROR register COMA register COMA IMMEDIATE
| BIC register COMA register COMA register | BIC register COMA register COMA IMMEDIATE
| MVN register COMA register COMA register | MVN register COMA register COMA IMMEDIATE
| MUL register COMA register COMA register
| MLA register COMA register COMA register COMA register
| STR register COMA CORCHETEIZQUIERDO register COMA IMMEDIATE CORCHETEDERECHO
| STR register COMA CORCHETEIZQUIERDO register CORCHETEDERECHO COMA IMMEDIATE
| STR register COMA CORCHETEIZQUIERDO register COMA register CORCHETEDERECHO
| LDR register COMA CORCHETEIZQUIERDO register COMA IMMEDIATE CORCHETEDERECHO
| LDR register COMA CORCHETEIZQUIERDO register CORCHETEDERECHO COMA IMMEDIATE
| LDR register COMA CORCHETEIZQUIERDO register COMA register CORCHETEDERECHO
| STRB register COMA CORCHETEIZQUIERDO register COMA IMMEDIATE CORCHETEDERECHO
| STRB register COMA CORCHETEIZQUIERDO register CORCHETEDERECHO COMA IMMEDIATE
| STRB register COMA CORCHETEIZQUIERDO register COMA register CORCHETEDERECHO
| LDRB register COMA CORCHETEIZQUIERDO register COMA IMMEDIATE CORCHETEDERECHO
| LDRB register COMA CORCHETEIZQUIERDO register CORCHETEDERECHO COMA IMMEDIATE
| LDRB register COMA CORCHETEIZQUIERDO register COMA register CORCHETEDERECHO
| B LABEL | BGE LABEL | BLE LABEL | BLT LABEL | BGT LABEL | BEQ LABEL | BNE LABEL | BL LABEL | BLEQ LABEL
| BLNE LABEL | BLGE LABEL | BLLT LABEL
| BLGT LABEL | BLLE LABEL
| NEWINSTRUCTION
| LABEL
register::= REGISTER
```

Figura 2. Gramática del ensamblador

Una vez que se define el archivo de especificación sintáctica, se ejecuta una vez para generar la clase del Analizador Sintáctico.

Clase “SyntacticAnalyzer”

Esta clase representa al analizador sintáctico, y se genera por medio del archivo de texto que contiene la especificación sintáctica. Existen dos tipos de análisis sintáctico: el descendente y el ascendente. La clase autogenerada se encarga de realizar un análisis sintáctico ascendente con algoritmos LALR. Un análisis sintáctico ascendente es un método más potente y eficiente que los análisis descendentes, pero ésta eficiencia conlleva métodos más complejos. La clase generada contiene las tablas de producción, la tabla de acción y la tabla que se encarga de ir reduciendo los estados sintácticos de la especificación, estos tres atributos son utilizados por el analizador sintáctico para realizar el análisis correcto. Las tablas de producción proporcionan el número de símbolo de la izquierda no terminal, junto con la longitud del lado derecho, para cada regla gramatical. Las tablas de acción indican que acción se va a tomar conforme avanza el análisis. La clase generada proporciona una serie de métodos que se pueden utilizar para personalizar el analizador sintáctico. Los otros métodos que son generados son proporcionados por el generador para realizar el análisis

sintáctico. En la figura 3, se puede observar el diagrama UML que define a esta clase.

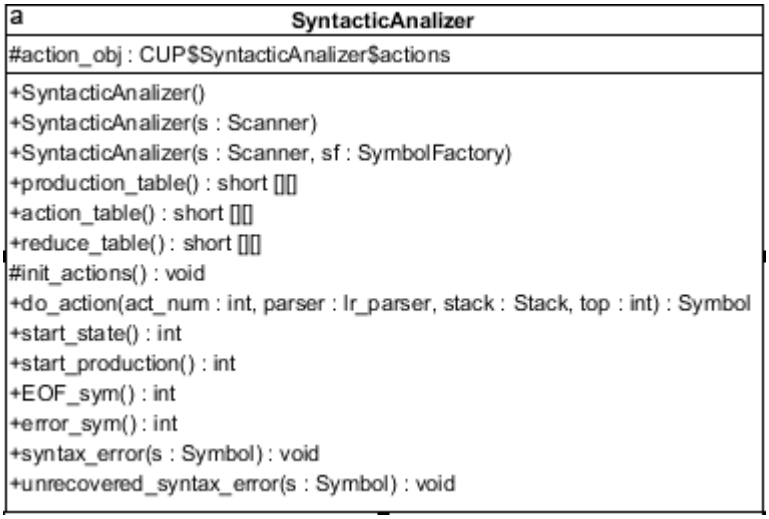

Figura 3. Diagrama UML, clase SyntacticAnalyzer.

Los métodos que contiene esta clase se explican a continuación:

- production_table: método que no recibe entradas y se utiliza para acceder la tabla de producciones.
- action_table: método que no recibe entradas y se utiliza para acceder a la tabla de acciones.
- reduce_table: método que no recibe ninguna entrada y se utiliza para acceder a la tabla que reduce los estados sintácticos.
- init_actions: método que no recibe entradas y es llamado por el analizador antes de pedir el primer token del análisis léxico.
- do_action: método que se encarga de llamar una de las acciones sintácticas definidas por el usuario.
- start_state: método que retorna el estado inicial del análisis.
- start_production: método que retorna la regla gramatical inicial.
- EOF_sym: método que no recibe entradas y retorna si se analizaron todos los token.
- error_sym: método que no recibe ninguna entrada y retorna el índice del símbolo que causó el error.
- syntax_error: método que retorna los errores encontrados.

- unrecovered_syntax_error: método que retorna que los errores no pudieron ser arreglados por el analizador.

Clase “CodeGeneration”

Esta clase se encarga de generar el código máquina de la instrucción en código ARM. En una primera versión del compilador esta clase no existía, la decodificación se realizaba en la clase del analizador sintáctico, sin embargo, esta clase era extensa y para mantener una mayor modularidad en el código se decidió en separarlo en una clase que se encargará únicamente de generar el código.

Consta de 9 instancias de clase:

- labelList: lista que contienen todas las etiquetas presentes en el código y sus direcciones.
- instructionList: lista que contiene cada una de las instrucciones en hexadecimal.
- errorList: lista que contiene cada uno de los errores que aparecen en los análisis.
- calculatingAddressesCompleted: esta variable se utiliza como bandera para indicar que ya se calcularon todas las direcciones de las etiquetas.
- lexicalError: esta variable se utiliza como bandera e indica que un error léxico ocurrió.
- syntacticError: esta variable se utiliza como bandera e indica que un error sintáctico ocurrió.
- semanticError: esta variable se utiliza como bandera e indica que un error semántico ocurrió.
- labelExist: esta variable se utiliza como bandera e indica si una etiqueta no existe.
- instruction_direction: esta variable se encarga de almacenar la dirección en la que se encuentra la instrucción.

Los métodos que contiene esta clase se explican a continuación:

- generateCode: este método no recibe ninguna entrada. Este método se encarga de realizar la generación de código, para esto empieza

inicializando cada una de las variables de clase, luego inicia el análisis léxico, para realizar el análisis léxico se cuenta con una instancia de la clase LexicalAnalyzer, si el análisis léxico es el correcto, procede a realizar el análisis sintáctico por medio de la instancia de la clase SyntacticAnalyzer, y si el análisis sintáctico se realiza adecuadamente entonces se llama a las funciones que se encargan de generar el código hexadecimal correspondiente para cada una de las diferentes instrucciones.

- createOutputFile: este método no recibe ninguna entrada y se encarga de crear el archivo de salida “out.txt”, para ello verifica primero si el archivo existe, de lo contrario crea el archivo.
- writeOutputFile: este método recibe como entrada un string que contiene la instrucción en hexadecimal. El método se encarga de escribir esta instrucción en el archivo de salida “out.txt”.
- generateHexInstruction: este método recibe como entrada un string que contiene a la instrucción decodificada en binario. El método se encarga de convertir el código binario en su código hexadecimal correspondiente, para ello primero verifica que no haya ocurrido ningún error léxico, sintáctico, ni semántico y luego procede a convertir el código binario a hexadecimal por medido de la función que ofrece Java llamada toHexString.
- generateCodeBranchInstructions: este método recibe como entrada los parámetros: cond, funct y la etiqueta a donde se realiza el salto. El método se encarga de generar el código binario de la instrucción de salto B y BL, para ello primero calcula el branch target address, luego verifica si la etiqueta existe, en caso de que la etiqueta no exista entonces se genera un error semántico, de lo contrario se convierte el número de los registros a su correspondiente valor binario, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.
- findDirection: este método recibe como entrada el nombre de la etiqueta y se encarga de iterar por toda la lista que contiene las etiquetas del programa para verificar que la etiqueta correspondiente existe. En el caso

en que no se encuentre la etiqueta se procede a generar un error semántico.

- generateCodeAddressingRegisterOnly: este método recibe como entrada los parámetros: cond, op, i, cmd, s, Rn, Rd, shamt5, sh y Rm. El método se encarga de generar el código binario de la instrucción de procesamiento de datos con direccionamiento por registros, para ello primero obtiene los números de los registros, luego convierte cada uno de estos números en su formato binario correspondiente, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.
- generateCodeAddressingRegisterShiftedRegister: este método recibe como entrada los parámetros: cond, op, i, cmd, s, Rn, Rd, Rs, sh y Rm. El método se encarga de generar el código binario de la instrucción de procesamiento de datos del tipo shift con direccionamiento por registros, para ello primero obtiene los números de los registros, luego convierte cada uno de estos números en su formato binario correspondiente, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.
- generateCodeMultiplicationInstr: este método recibe como entrada los parámetros: cond, cmd, s, Rd, Ra, Rm y Rn. El método se encarga de generar el código binario de la instrucción de procesamiento de datos del tipo multiplicación con direccionamiento por registros, para ello primero obtiene los números de los registros, luego convierte cada uno de estos números en su formato binario correspondiente, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.
- generateCodeMemoryRegisterOffset: este método recibe como entrada los parámetros: cond, p, u, b, w, l, Rn, Rd y Rm. El método se encarga de generar el código binario de la instrucción de transferencia de datos con direccionamiento por offset almacenado en un registro, para ello primero obtiene los números de los registros, luego convierte cada uno de estos números en su formato binario correspondiente, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.

- generateCodeMemoryInstrHexImmediate: este método recibe como entrada los parámetros: cond, p, w, b, l, Rn, Rd y el inmediato en hexadecimal. El método se encarga de generar el código binario de la instrucción de transferencia de datos con direccionamiento por inmediato hexadecimal, para ello primero obtiene los números de los registros, luego verifica el largo del inmediato, si el largo está entre el rango adecuado se procede a convertir cada uno de los números del registro en su formato binario correspondiente, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.
- generateCodeMemoryInstrDecImmediate: este método recibe como entrada los parámetros: cond, p, w, b, l, Rn, Rd y el inmediato en decimal. El método se encarga de generar el código binario de la instrucción de transferencia de datos con direccionamiento por inmediato decimal, para ello primero obtiene los números de los registros, luego verifica el largo del inmediato, si el largo está entre el rango adecuado se procede a convertir cada uno de los números del registro en su formato binario correspondiente, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.
- generateCodeDataProcessingInstrDecImmediate: este método recibe como entrada los parámetros: cond, cmd, Rn, Rd y el inmediato en decimal. El método se encarga de generar el código binario de la instrucción de procesamiento de datos con direccionamiento por inmediato decimal, para ello primero obtiene los números de los registros, luego verifica el largo del inmediato, si el largo está entre el rango adecuado se procede a calcular el valor del inmediato codificado y su valor de rotación. Si el inmediato no se puede codificar entonces se genera un error semántico, de lo contrario se retorna el valor del inmediato codificado y su rotación y luego convierte cada uno de estos números en su formato binario correspondiente, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.

- generateCodeShiftInstrDecImmediate: este método recibe como entrada los parámetros: cond, cmd, Rn, Rd y el inmediato en decimal. El método se encarga de generar el código binario de la instrucción de transferencia de datos con direccionamiento por inmediato decimal, para ello primero obtiene los números de los registros, luego verifica el largo del inmediato, si el largo está entre el rango adecuado se procede a calcular el valor del inmediato codificado y su valor de rotación. Si el inmediato no se puede codificar entonces se genera un error semántico, de lo contrario se retorna el valor del inmediato codificado y su rotación y luego convierte cada uno de estos números en su formato binario correspondiente, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.
- generateCodeDataProcessingInstrHexImmediate: este método recibe como entrada los parámetros: cond, cmd, Rn, Rd y el inmediato en hexadecimal. El método se encarga de generar el código binario de la instrucción de transferencia de datos con direccionamiento por inmediato hexadecimal, para ello primero obtiene los números de los registros, luego verifica el largo del inmediato, si el largo está entre el rango adecuado se procede a calcular el valor del inmediato codificado y su valor de rotación. Si el inmediato no se puede codificar entonces se genera un error semántico, de lo contrario se retorna el valor del inmediato codificado y su rotación y luego convierte cada uno de estos números en su formato binario correspondiente, y por último se genera la instrucción en código binario y se manda a llamar a la instrucción generateHexInstruction con el código binario generado.
- getRotationImmediateValues: este método recibe como entrada el inmediato en hexadecimal. El método se encarga de verificar si el inmediato se puede codificar, en caso de que se logre retorna el valor codificado y el valor de la rotación en caso contrario retorna un error semántico.

En la figura 4, se puede observar el diagrama UML que define a esta clase.

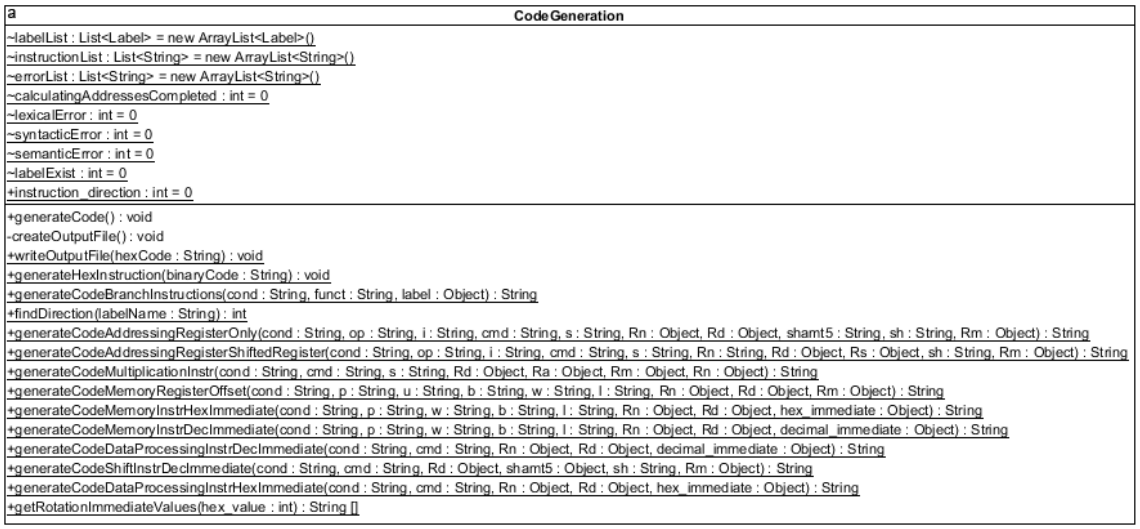

Figura 4. Diagrama UML, clase CodeGeneration.

Simulador de código ARM

El objetivo del simulador de código es emular el comportamiento de un procesador cuando ejecuta un programa en el lenguaje de programación ensamblador con la arquitectura ARMv4.

ARM por lo tanto es una arquitectura, es decir corresponde a un set de instrucciones. Esta arquitectura es una arquitectura del tipo RISC, que comparada con su contraparte CISC, tiene un diseño de hardware mucho más sencillo. Para poder realizar el simulador se debe entonces contar como mínimo con una memoria, un banco de registros, una ALU, una unidad extensora de signo, una unidad extensora de cero y una unidad de control.

Mapeando lo anterior al programa, se debe contar entonces con las siguientes clases:

- Una clase que represente a la memoria.
- Una clase que represente al banco de registros.
- Una clase que represente a la unidad de control.
- Una clase que permita realizar las operaciones de la ALU, y extensión de signo y cero.

Además de las clases mencionadas anteriormente, se proponen:

- Una clase que maneje las instrucciones soportadas por la arquitectura ARMv4.
- Una clase que contenga los labels que se encuentren en el código.
- Una clase de interfaz gráfica.

Como se puede observar con las clases propuestas anteriormente, se busca relevar cualquier tipo de lógica de negocio a la interfaz gráfica.

Clase “Memory”

Esta clase representa a la memoria del computador, se creó siguiendo el patrón de diseño singleton. Consta de una única instancia de clase denominada “memory” que consiste en un arreglo dinámico de strings, esto porque el contenido de la memoria son números en hexadecimal, que son representados como strings. Todos los métodos de la clase trabajan sobre esta instancia. El constructor de esta clase se encarga de inicializar la instancia “memory” con 2048 elementos, esto para poder acceder a cualquier ubicación de la memoria de 2KB (este tamaño de memoria es definido por los requerimientos del cliente). En la figura 5, se puede observar el diagrama UML que define a esta clase.

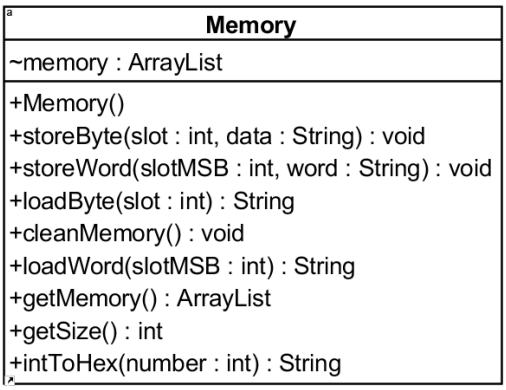

Figura 5. Diagrama UML, clase Memory.

Los métodos que contiene esta clase se explican a continuación:

- storeByte: método que recibe como entrada una posición de memoria, representada por un número entero y un string que contiene el dato que se quiere almacenar. Se encarga de almacenar en la posición de memoria dada, el string de entrada. La posición de memoria debe ser menor a 2048 y superior o igual a 0. El string debe estar compuesto por 2 caracteres.

- storeWord: método que recibe como entrada una posición de memoria, representada por un número entero y un string que contiene el dato que se quiere almacenar. A diferencia de storeByte, el string que recibe este método puede tener 8 caracteres. Este método divide el string en 4 partes de 2 caracteres cada una y llama al método storeByte con cada parte, la parte menos significativa es almacenada en la posición que recibe el método, entre más significativa la parte se suma 1 a la posición. La posición de entrada debe ser un valor entero divisible por 4, esto para asegurar que la memoria no se desalinee.
- loadByte: método que recibe como entrada una posición de memoria, representada por un número entero, y da como salida el valor de esa posición de la memoria.
- loadWord: método que recibe como entrada una posición de memoria, representada por un número entero divisible por 4, y da como salida una concatenación del dato en esa posición de memoria y las 3 posiciones siguientes.
- getMemory: método que retorna la instancia de la memoria.
- getSize: método que retorna el tamaño del arreglo de la memoria.

Clase “Register”

Esta clase representa al banco de registros, en la arquitectura ARM se cuenta con 15 registros de propósito general. Cada registro representa por lo tanto una instancia de clase, estas instancias son de tipo string y pueden contener máximo 8 caracteres, aunque esta verificación no se realiza en esta clase. Cada instancia cuenta con un método de tipo get y otro de tipo set, conocidos como getter and setter. El método de tipo get retorna el valor de la variable, el método de tipo set asigna un valor a la variable. Además, existe un método llamado findRegister, que recibe como entrada un registro y da como salida el valor de ese registro. También existe la contraparte a este método, con un método denominado setRegister que recibe como entrada un string con el nombre de un registro y un nuevo valor para asignarle, este método ubica el registro y le asigna el nuevo valor. En la figura 6, se puede observar el diagrama UML que define a esta clase.

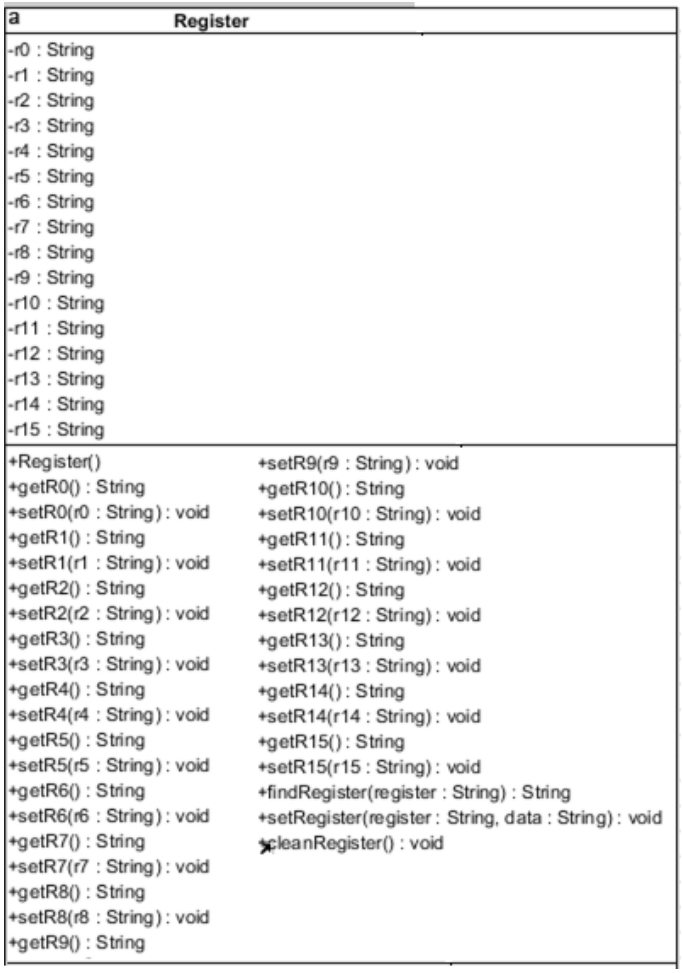

Figura 6. Diagrama UML, clase Register.

Clase “ConditionFlags”

Esta clase representa las banderas condicionales, si bien estas se pueden incluir en la clase de la unidad de control, se decidió separarlas para favorecer el alto acoplamiento y la baja cohesión. Consta de 4 instancias de clase de tipo booleano, cada una de tipo booleano: negative, zero, carry y oVerflow. Estas instancias son modificadas por el resultado de la ALU producido en las instrucciones cmp y cmn, la implementación de estas instrucciones se explica más adelante en este informe.

El constructor de esta clase inicializa las 4 instancias en false. Esta clase contiene como métodos los getter y setter para cada una de estas instancias, cabe destacar que la abreviatura get se cambia por is, ya que se retornan valores booleanos.

En la figura 7, se puede observar el diagrama UML que define a esta clase.

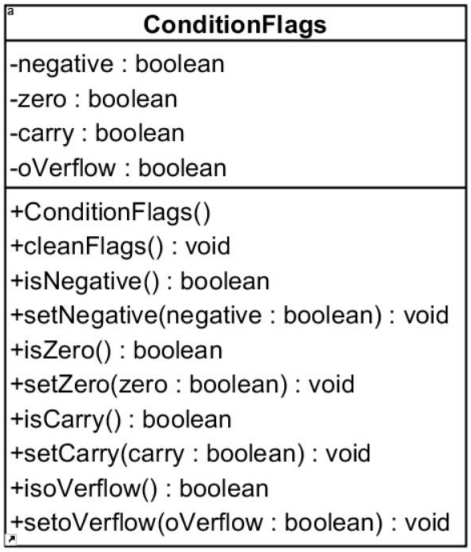

Figura 7. Diagrama UML, clase ConditionFlags.

Clase “ReserveInstructions”

Esta clase contiene las palabras reservadas o instrucciones de la arquitectura. En una primera versión del simulador esta clase no existía, todas las operaciones se realizaban directo en la clase que representaba a la unidad de control, sin embargo, esta clase era extensa y mezclaba operaciones que no tenían ninguna relación, es decir existía una baja cohesión. El crear esta clase permite aislar la detención del tipo de instrucción que se está manejando y además discernir entre instrucciones y etiquetas.

Consta de 4 instancias de clase:

- DataProcessingInstructions: arreglo dinámico de strings, que contiene todas las instrucciones del tipo data processing.
- MultiplyInstructions: arreglo dinámico de strings, que contiene todas las instrucciones del tipo multiply.
- MemoryInstructions: arreglo dinámico de strings, que contiene todas las instrucciones del memory.
- BranchInstructions: arreglo dinámico de strings, que contiene todas las instrucciones del tipo branch.

Existen 4 métodos encargados de llenar cada uno de estos arreglos dinámicos con las instrucciones correspondientes a cada grupo de instrucciones, estos son:

fillDataProcessingInstructions, fillMultiplyInstructions, fillMemoryInstructions y fillBranchInstructions. El constructor de esta clase se encarga de llamar a estos métodos.

Los otros 4 métodos restantes son para verificar el tipo de instrucción:

- isDataProcessingInstructions: este método recibe como entrada un string, retorna true si el string de entrada se encuentra contenido en el arreglo dinámico DataProcessingInstructions, false en cualquier otro caso.
- isMultiplyInstructions: este método recibe como entrada un string, retorna true si el string de entrada se encuentra contenido en el arreglo dinámico MultiplyInstructions, false en cualquier otro caso.
- isMemoryInstructions: este método recibe como entrada un string, retorna true si el string de entrada se encuentra contenido en el arreglo dinámico MemoryInstructions, false en cualquier otro caso.
- isBranchInstructions: este método recibe como entrada un string, retorna true si el string de entrada se encuentra contenido en el arreglo dinámico BranchInstructions, false en cualquier otro caso.

En la figura 8, se puede observar el diagrama UML que define a esta clase.

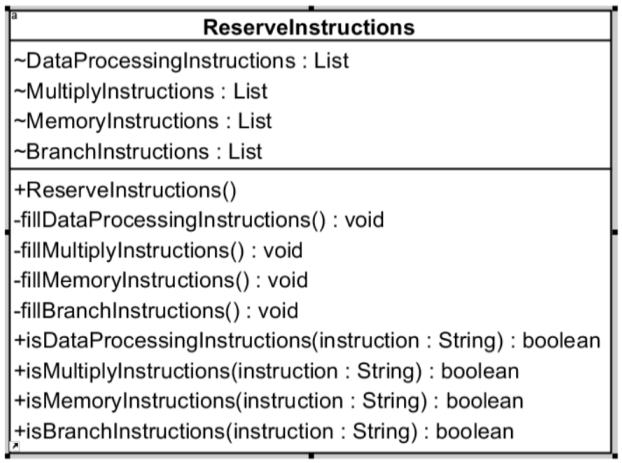

Figura 8. Diagrama UML, clase ReserveInstructions.

Clase “HashLabel”

Esta clase es la encargada de manejar las etiquetas en el código. Antes de simular el código este es recorrido línea por línea, para detectar las etiquetas y su posición. Para detectar una etiqueta se compara la línea con las instrucciones

almacenadas en la clase ReserveInstructions, en caso de no ser una instrucción y no contener espacios en blanco, la línea es agregada a la lista de etiquetas.

La clase es conformada por 2 instancias de clase:

- hashTableLabels: consiste en una lista de sublistas, la cual contiene en cada sublista la etiqueta y la posición donde está ubicada en el código.
- reserveInstructions: consiste en una instancia de la clase ReserveInstructions, es inicializada por el constructor.

Los métodos de esta clase son:

- addLabel: este método recibe como entrada un string y su posición, agrega este string y la posición a la lista de sublistas de etiquetas.
- fillHashTable: este método recibe como entrada un string y su posición, verifica que sea una etiqueta y en caso de cumplir llama al método addLabel.
- findLabel: este método tiene como entrada un string, se encarga de verificar si ese string se encuentra en la lista de etiquetas, en caso de cumplirse retorna la ubicación de la etiqueta, en otro caso retorna -1.

En la figura 9, se puede observar el diagrama UML que define a esta clase.

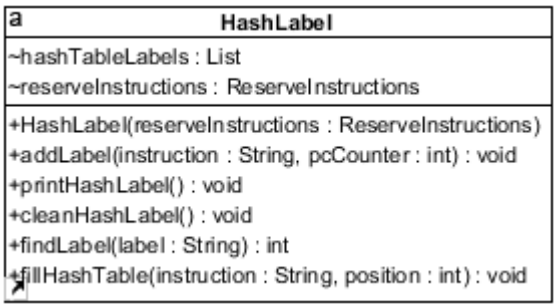

Figura 9. Diagrama UML, clase HashLabel.

Clase “Operation”

Esta clase reúne en su mayoría la lógica del simulador, funciona como la unidad de control detectando que instrucción es la que se debe ejecutar, aunque en esta clase también se ejecuta la instrucción. Tiene como instancias de clase, instancias de las otras clases específicamente:

- bankRegister: instancia de la clase Register.

- memory: instancia de la clase Memory.
- conditionFlag: instancia de la clase ConditionFlags.
- hashLabel: instancia de la clase HashLabel.
- pcCounter: como su nombre lo dice representa al PC Counter, es de tipo int. Tiene como función servir como puntero de la instrucción actual ejecutandose.
- reserveInstructions: instancia de la clase ReserveInstructions.

El constructor de esta clase recibe como entrada todas las instancias de clase anteriores y las asigna, la instancia pcCounter es inicializada en 0.

Los métodos de esta clase se describen a continuación:

- Operation: constructor de la clase, se encarga de asignar e inicializar las instancias de clase antes mencionadas.
- getPCCounter: método encargado de retornar el valor actual del pcCounter.
- verifyNegative: método que recibe un número binario en formato de string, como entrada, retorna true si el número es negativo, false en caso de sea positivo.
- verifyZero: método que recibe un número binario en formato de string, como entrada, retorna true si el número es cero, false en caso cualquier otro número diferente a cero.
- verifyCarry: método que recibe un entero con la longitud de un número, si este número es mayor que 32 retorna true en otro caso false.
- verifyOverflow: método que recibe 2 números binarios en formato de string rn y src2, retorna true en caso de que se dé un overflow, false en caso de que no.
- selectOperation: este método recibe como entrada un string con una instrucción, utilizando la variable reserveInstructions verifica que tipo de instrucción es y por medio de la variable hashLabel se descarta que la instrucción sea una etiqueta. En caso de no ser una instrucción ni una etiqueta se envía un error. Este método además es el encargado de incrementar el pcCounter en 1 cada vez que se lee una instrucción. La detención del tipo de instrucción se realiza de la siguiente manera:

- reserveInstructions.isDataProcessingInstructions: en caso de que esta verificación retorne true, se llama al método dataProcessingDecodeInstruction.
- reserveInstructions.isMultiplyInstructions: en caso de que esta verificación retorne true, se llama al método multiplyDecodeInstruction.
- reserveInstructions.isMemoryInstructions: en caso de que esta verificación retorne true, se llama al método memoryDecodeInstruction.
- reserveInstructions.isBranchInstructions: en caso de que esta verificación retorne true, se llama al método BranchDecodeInstruction.

- dataProcessingDecodeInstruction: este método recibe como entrada el string de la instrucción, lo decodifica a las partes rd, src2 y dependiendo de la instrucción rn. Para la decodificación: rd se mantiene como el registro destino, src2 puede ser un inmediato o un registro, en caso de que sea un registro se utiliza la variable bankRegister de la clase Register, se llama al método findRegister (se explicó en la clase Register) y se obtiene el valor del registro, para rn este debe ser un registro, por ende, se realiza lo mismo que para src2 en el caso de registros. Luego de decodificar a la instrucción se llama al método específico de la misma. Para “data processing” se tienen los siguientes métodos:
  - mov: recibe como entrada un string rd (registro destino) y un long src2, castea el src2 a hexadecimal y utiliza la variable bankRegister para llamar al método setRegister (este método se explicó en la clase register), esto con el objetivo de asignar al registro el nuevo valor.
  - add: este método recibe como entrada un string rd, un long rn y un long src2, suma rn con src2 y llama al método mov pasándole como parámetro rd y el resultado de la suma.
  - eor: este método recibe como entrada un string rd, un long rn y un long src2, aplica la operación XOR a rn con src2 y llama al método mov pasándole como parámetro rd y el resultado del XOR.

- orr: este método recibe como entrada un string rd, un long rn y un long src2, aplica la operación OR a rn con src2 y llama al método mov pasándole como parámetro rd y el resultado del OR.
- lsl: este método recibe como entrada un string rd, un long rn y un long src2, aplica la operación <<= a rn con src2 y llama al método mov pasándole como parámetro rd y el resultado del <<=.
- asr: este método recibe como entrada un string rd, un long rn y un long src2, aplica la operación >>= a rn con src2 y llama al método mov pasándole como parámetro rd y el resultado del >>=.
- and: este método recibe como entrada un string rd, un long rn y un long src2, aplica la operación AND a rn con src2 y llama al método mov pasándole como parámetro rd y el resultado del AND.
- bic: este método recibe como entrada un string rd, un long rn y un long src2, niega a src2 y luego aplica un and entre rn y src2 (negado), luego se llama al método mov pasándole como parámetro rd y el resultado del AND.
- mvn: este método recibe como entrada un string rd y un long src2, niega src2 y llama al método mov pasándole como parámetro rd y el resultado de de src2 negado.
- sub: este método recibe como entrada un string rd, un long rn y un long src2, le resta a rn, src2 y llama al método mov pasándole como parámetro rd y el resultado de la resta.
- sbc: este método recibe como entrada un string rd, un long rn y un long src2, le resta a rn, src2. Realizado esto se utiliza a la variable conditionFlag para llamar al método isCarry, en caso de que este retorne true al resultado de la resta se le resta un 1, en caso de que retorne false no se modifica el resultado de la resta. Por último, se llama al método mov pasándole como parámetro rd y el resultado de la resta.
- ror: este método recibe como entrada un string rd, un long rn y un long src2, aplica la operación ROR a rn con src2 y llama al método mov pasándole como parámetro rd y el resultado del ROR.
- cmn: este método recibe como entrada un long rn y un long src2, suma ambos números y realiza las asignaciones:

        - Llama al método setNegative de la variable conditionFlag y le pasa como parámetro la función verifyNegative que a su vez recibe como parámetro el resultado de la suma casteado a binario en formato de string.
        - Llama al método setZero de la variable conditionFlag y le pasa como parámetro la función verifyZero que a su vez recibe como parámetro el resultado de la suma casteado a binario en formato de string.
        - Llama al método setCarry de la variable conditionFlag y le pasa como parámetro la función verifyCarry que a su vez recibe como parámetro la cantidad de bits del resultado de la suma casteado a binario.
        - Llama al método setoVerflow de la variable conditionFlag y le pasa como parámetro la función verifyOverflow que a su vez recibe como parámetros rn y src2 casteados a binario en formato de string.
    - cmp: este método recibe como entrada un long rn y un long src2, resta ambos números y realiza las asignaciones:
        - Llama al método setNegative de la variable conditionFlag y le pasa como parámetro la función verifyNegative que a su vez recibe como parámetro el resultado de la resta casteado a binario en formato de string.
        - Llama al método setZero de la variable conditionFlag y le pasa como parámetro la función verifyZero que a su vez recibe como parámetro el resultado de la resta casteado a binario en formato de string.
        - Llama al método setCarry de la variable conditionFlag y le pasa como parámetro la función verifyCarry que a su vez recibe como parámetro la cantidad de bits del resultado de la resta casteado a binario.
        - Llama al método setoVerflow de la variable conditionFlag y le pasa como parámetro la función verifyOverflow que a su vez recibe como parámetros rn y src2 casteados a binario en formato de string.

- multiplyDecodeInstruction: este método recibe como entrada el string de la instrucción, lo decodifica a las partes rd, rn, rm y dependiendo de la instrucción ra. Para la decodificación: rd se mantiene como el registro destino, rn, rm y ra son registros, por ende, se utiliza la variable bankRegister de la clase Register, se llama al método findRegister (se explicó en la clase Register) y se obtiene el valor de cada registro. Luego de decodificar a la instrucción se llama al método específico de la misma. Para “multiply” se tienen los siguientes métodos:
    - mul: este método recibe como entrada un string rd, un long rn y un long rm, multiplica rn con rm y llama al método mov pasándole como parámetro rd y el resultado de la multiplicación.
    - mla: este método recibe como entrada un string rd, un long rn, un long rm y un long ra, multiplica rn con rm y suma al resultado ra, luego llama al método mov pasándole como parámetro rd y el resultado de la multiplicación.
- memoryDecodeInstruction: este método recibe como entrada el string de la instrucción, lo decodifica a las partes rd, src2 y dependiendo de la instrucción rn. Para la decodificación: rd se mantiene como el registro destino, src2 puede ser un inmediato o un registro, en caso de que sea un registro se utiliza la variable bankRegister de la clase Register, se llama al método findRegister (se explicó en la clase Register) y se obtiene el valor del registro, para rn este debe ser un registro, por ende, se realiza lo mismo que para src2 en el caso de registros. Luego de decodificar a la instrucción se llama al método específico de la misma. Para “memory” se tienen los siguientes métodos:
    - ldr: este método recibe un string rd que representa el registro destino, un long rn y un long srcs. Se suma rn con src2, se verifica que el número resultante se encuentre en el rango de 1024 y 2047, esto para cumplir con el requerimiento de que tanto la memoria de programa como la de datos deben ser de 1KB. Hecha la anterior verificación, se valida que la división entera del número por 4 sea 0, esto con el fin de asegurar que la memoria no se desalinee. Si las 2 revisiones anteriores se cumplen se utiliza la variable memory para llamar al método loadWord al que se le pasa como parámetro

la suma de rn y src2. Por último, se llama al método mov con el dato que devuelve loadWord y rd.

- ldrb: este método recibe un string rd que representa el registro destino, un long rn y un long srcs. Se suma rn con src2, se verifica que el número resultante se encuentre en el rango de 1024 y 2047, esto para cumplir con el requerimiento de que tanto la memoria de programa como la de datos deben ser de 1KB. Si la validación anterior se cumple se utiliza la variable memory para llamar al método loadByte al que se le pasa como parámetro la suma de rn y src2. Por último, se llama al método mov con el dato que devuelve loadByte y rd.
- str: este método recibe un string rd que representa el registro destino, un long rn y un long srcs. Se suma rn con src2, se verifica que el número resultante se encuentre en el rango de 1024 y 2047, esto para cumplir con el requerimiento de que tanto la memoria de programa como la de datos deben ser de 1KB. Hecha la anterior verificación, se valida que la división entera del número por 4 sea 0, esto con el fin de asegurar que la memoria no se desalinee. Si las 2 revisiones anteriores se cumplen se utiliza la variable memory para llamar al método storeWord al que se le pasa como parámetro la suma de rn y src2 y el valor del registro rd, este se obtiene utilizado la variable bankRegister de la clase Register por medio del método findRegister.
- strb: este método recibe un string rd que representa el registro destino, un long rn y un long srcs. Se suma rn con src2, se verifica que el número resultante se encuentre en el rango de 1024 y 2047, esto para cumplir con el requerimiento de que tanto la memoria de programa como la de datos deben ser de 1KB Si la validación anterior se cumple se utiliza la variable memory para llamar al método storeByte al que se le pasa como parámetro la suma de rn y src2 y el valor del registro rd, este se obtiene utilizado la variable bankRegister de la clase Register por medio del método findRegister.

- BranchDecodeInstruction: este método recibe como entrada el string de la instrucción, lo decodifica para obtener un string con el label y en caso de que sea una instrucción de tipo BL, activa una bandera booleana denominada link. Luego de decodificar a la instrucción se llama al método específico de la misma. Para “memory” se tienen los siguientes métodos:
    - b: este método recibe como entrada un valor booleano link y el string del label. Primero se obtiene por medio del método findLabel de la variable hashLabel, la posición en el código del label. Si la posición obtenida es diferente a -1, se asigna al pcCounter la posición en el código del label. Si la variable link se encuentra activa se asigna al registro r14 la posición previa al cambio del pcCounter.
    - bge: este método recibe como entrada un valor booleano link y el string del label. Utilizando la variable conditionFlag llama a los métodos isZero e isNegative, en caso de que isZero retorne true e isNegative false llama al método b, con los parámetros link y label.
    - ble: este método recibe como entrada un valor booleano link y el string del label. Utilizando la variable conditionFlag llama a los métodos isZero e isNegative, en caso de que isZero retorne true e isNegative true llama al método b, con los parámetros link y label.
    - blt: este método recibe como entrada un valor booleano link y el string del label. Utilizando la variable conditionFlag llama a el método isNegative, en caso de que retorne true llama al método b, con los parámetros link y label.
    - bgt: este método recibe como entrada un valor booleano link y el string del label. Utilizando la variable conditionFlag llama a los métodos isZero e isNegative, en caso de que tanto isZero como isNegative retornen false llama al método b, con los parámetros link y label.
    - beq: este método recibe como entrada un valor booleano link y el string del label. Utilizando la variable conditionFlag llama a el método isZero, en caso de que retorne true llama al método b, con los parámetros link y label.

- bne: este método recibe como entrada un valor booleano link y el string del label. Utilizando la variable conditionFlag llama a el método isZero, en caso de que retorne false llama al método b, con los parámetros link y label.

En la figura 10, se puede observar el diagrama UML que define a esta clase.

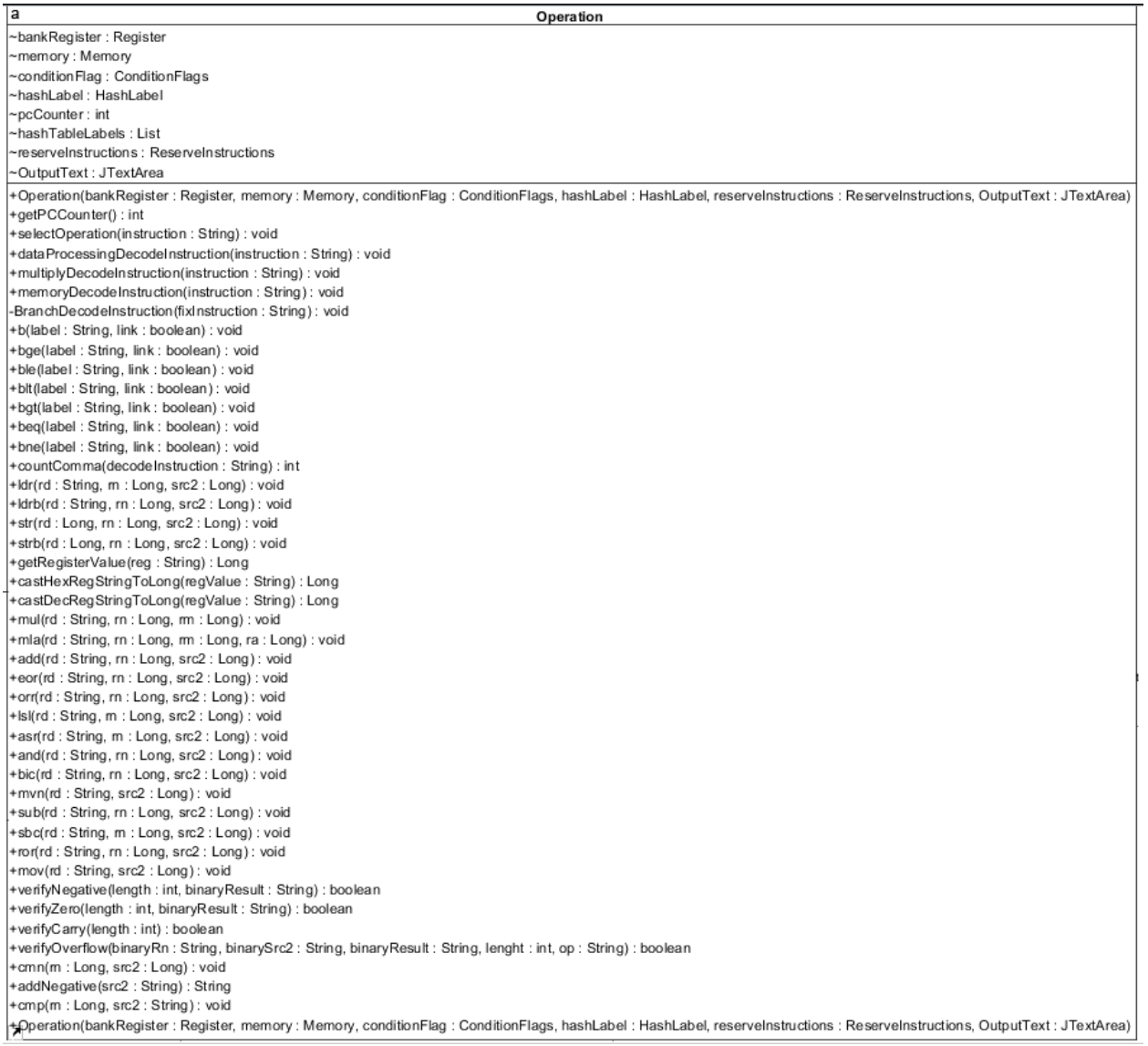

Figura 10. Diagrama UML, clase Operation.

Problemas semánticos

El simulador detiene su ejecución cuando alguno de los siguientes eventos ocurre:

- Se intenta leer o escribir una palabra en la memoria, con una posición desalineada.
- Se intenta escribir o leer en la memoria, en una posición no existente o asignada a la memoria de programa.
- Se trata de realizar un branch a un label no existente.

Además, en caso de que se trate de asignar valores a registros mayores a 8 bytes el simulador asigna los 32 bits menos significativos.

Problemas de desarrollo

El problema principal que se presentó fue con el tipado. No existe un tipo como tal en Java hexadecimal, por ende, este tipo de dato se debe manejar por medio de strings. Para realizar una operación sobre un número hexadecimal se debe realizar una conversión de tipo a integer o a long. Para decidir a qué tipo de dato, se debe recurrir a la capacidad de bits de cada uno, un integer soporta 31 bits, un long soporta 63 bits. Por los requerimientos cliente se necesitan 32 bits, por este motivo se optó por el tipo de dato long. Long posee métodos de conversión de hexadecimal (string) a long, para operar los números resulta muy valioso este tipo de conversión, sin embargo, cuando se pasa de long a hexadecimal, existe la posibilidad de que el número que se quiere operar sea negativo, esto genera que se complete el número con F's, pero recordando que long es de 63 bits, el número se completa con una mayor cantidad de F's lo que genera por ejemplo si se quiere detectar un carry que existan falsos positivos.

La solución que se encontró al problema anterior fue quitar el negativo del número long y completar manualmente el string con F's, esto hace que no existan falsos positivos en un carry.

Lista de chequeo de cumplimiento de los requerimientos

En la tabla 2 se puede observar la lista de chequeo de cumplimiento de los requerimientos.

Requerimiento		Estado
Permitir la escritura de código ARMv4 en la interfaz gráfica.		Implementado
Guardar y cargar archivos de programas con extensión “armv4”.		Implementado
Generar el ensamblado del código cargado en la interfaz gráfica.		Implementado
Realizar la simulación del código.		Implementado
Análisis léxico del código		Implementado
Análisis sintáctico del código		Implementado
Análisis semántico del código		Implementado
El simulador y generador de código debe ser capaz de ejecutar las instrucciones de procesamiento de datos, con los modos de direccionamiento: • Registro: registro solo • Inmediato: con registro	AND	Implementado
	EOR	Implementado
	SUB	Implementado
	RSB	Implementado
	ADD	Implementado
	ADC	Implementado
	SBC	Implementado
	RSC	Implementado
	CMP	Implementado
	CMN	Implementado
	ORR	Implementado
	MOV	Implementado
	LSL	Implementado
	ASR	Implementado
	RRX	Implementado
	ROR	Implementado
	BIC	Implementado
	MVN	Implementado
	MUL	Implementado
	MLA	Implementado
El simulador y generador de código debe ser capaz de ejecutar las instrucciones de memoria, con los modos de direccionamiento: • Registro: registro solo • Inmediato: con registro • Base: offset inmediato y de registro.	STR	Implementado
	LDR	Implementado
	STRB	Implementado
	LDRB	Implementado
El simulador y generador de código debe ser capaz de ejecutar las instrucciones de salto, con el modo de direccionamiento: • Relativo a PC	B	Implementado
	BL	Implementado
Los tipos de datos que se deben de manejar son enteros de 8 y 32 bits		Implementado
La memoria debe ser de 2KB, 1KB para el programa y 1KB para los datos. La primera dirección debe ser 0x00 y la última 0x7FF. Se debe proveer la posibilidad al usuario de modificar la memoria manualmente, cualquier introducción errónea en la memoria manualmente es responsabilidad del usuario.		Implementado
La interfaz gráfica queda a criterio del grupo desarrollador, esta debe mostrar el banco de registros, la memoria, las banderas de la ALU, permitir de alguna manera introducir el código, cargarlo y guardarlo.		Implementado

Tabla 2. Lista de chequeo de requerimientos.

Metodología de diseño del sistema

Metodología de diseño

La metodología de diseño utilizada es la de proceso unificado [1]. Esta metodología tiene 4 etapas: inicio, elaboración, construcción y transición.

- La etapa de inicio tiene como objetivo identificar todas las entidades que interactúan con el sistema y definir esas interacciones.
- La etapa de elaboración tiene como objetivo desarrollar la comprensión del dominio del problema y desarrollar el plan del proyecto. Al terminar esta fase se tiene un modelo de los requerimientos del problema.
- La etapa de construcción tiene como objetivo el diseño, la implementación y las pruebas del sistema.
- La etapa de transición en esta etapa se traslada el sistema desde el entorno de desarrollo al entorno real.

Cada etapa consta de iteraciones, estas iteraciones pueden tener un enfoque ya sea top-down o bottom-up. En el proyecto se empleó un enfoque top-down. Las etapas de inicio y de elaboración se realizaron por medio de reuniones del equipo de desarrollo en las cuales se definieron los requerimientos y la modalidad que se iba a utilizar para desarrollar el proyecto. Es importante que además de decidir utilizar una metodología de diseño de proceso unificado, se agregó la metodología de diseño modular. Para la etapa de construcción el diseño modular fue fundamental, ya que se dividió el sistema en módulos, esto se explicará más adelante.

En cada iteración se llevó el proceso de diseño, implementación y pruebas.

Algunas ventajas que se presentaron de haber utilizado esta metodología son:

- Se dio una mitigación del riesgo temprana.
- Se redujo el tiempo de integración y esfuerzo.
- Los cambios fueron más manejables.
- Y se produjo un alto nivel de reutilización.

La principal desventaja presentada fue el alto gasto de tiempo en la etapa de diseño, ya que al ser un enfoque top down, las clases debían ser muy genéricas, para prevenir una integración sencilla.

Elección del lenguaje de programación

Al ser libre la elección del lenguaje, se tomaron en cuenta 3 lenguajes de alto nivel: python, C# y java [2]. Lenguajes de bajo nivel como por ejemplo “C” fueron descartados. Este tipo de lenguajes, aunque son muy potentes, para los requerimientos del cliente no era necesario ir a tan bajo nivel, por ende, elegir un lenguaje de este tipo lo que provocaba era un código más extenso y una mayor complejidad del mismo, generando que factores como tiempo, alcance y costo se afectaran negativamente.

Para elegir el lenguaje entre estos 3, se analizaron varias características:

- Tanto C#, como Java son lenguajes que utilizan un paradigma de programación orientada a objetos, python por su parte es multiparadigma, es decir este lenguaje de programación soporta múltiples paradigmas. La programación orientada a objetos como su nombre lo dice utiliza objetos como elementos fundamentales en la construcción de la solución. Un objeto corresponde a una abstracción del mundo real, con atributos que representan sus características y métodos que emulan su comportamiento. Los objetos se encapsulan en clases, por lo tanto, una clase es una plantilla, que es representada por instancias de clase también conocidas como objetos.
- Estos 3 lenguajes son funcionales para desarrollar aplicaciones de escritorio, aunque también permiten generar aplicaciones web. En este proyecto se desarrollará una aplicación de escritorio, ya que estas permiten realizar un manejo más sencillo de la interfaz gráfica, en caso de aplicaciones web se deben involucrar elementos como bootstrap, angular js, java script, etc, los cuales no son pertinentes para los requerimientos del cliente.
- Java y python son lenguajes multiplataforma, C# es compatible con el sistema operativo Windows.

- C# y Java son lenguajes compilados, Python es interpretado. Generar el ejecutable de Java o C# es mucho más lento que ejecutar el código de Python, sin embargo, el tiempo de ejecución de Java y C# es mucho mayor al de Python.
- Python es débilmente tipado, mientras que Java y C# son fuertemente tipados. Para la aplicación que se debe desarrollar un lenguaje débilmente tipado es una desventaja fuerte, ya que en todo momento se debe tener claro con qué tipo de dato se está trabajando, si no se hace es muy posible que se incurra en resultados erróneos.
- Existe gran cantidad de herramientas para verificar la calidad de código en lenguajes como C# y Java, en Python no.
- Java y Python llevan mucho tiempo como lenguajes open source, C# lleva poco como lenguaje de este tipo.
- La extensión Swing de Java permite desarrollar una interfaz gráfica con rapidez e intuitivamente, por su lado Python tiene varios editores de interfaz como pygame (dejó de recibir soporte desde la versión 2.7 de Python), QT (es de pago) y TKinter (gestor gratuito, aunque se queda corto en comparación con Swing), C# utiliza Windows Form Designer, aunque para su desarrollo es necesario Visual Studio (es de pago).

En vista a las comparaciones presentadas anteriormente se eligió como lenguaje de programación a Java, por ser multiplataforma, fuertemente tipado, open source, tener soporte gratuito, gran cantidad de recursos en internet y por último permitir realizar una interfaz gráfica de una manera sencilla.

Desarrollo de aplicación

Al implementarse la aplicación bajo el paradigma de programación orientado a objetos, se utilizaron los patrones GRASP para la asignación de responsabilidades [3]. Estos son patrones generales de software, específicamente:

- Experto en información: la responsabilidad de la creación de un objeto o la implementación de un método, recae sobre la clase que conoce toda la información necesaria del mismo. Esto genera una mayor cohesión y encapsulamiento del código.
- Creador: la instanciación de un objeto es realizada por la clase que:
    - Tiene la información necesaria para realizar la creación del objeto.
    - Usa directamente la instancia del objeto.
    - Contiene a la clase.
    - Maneja varias instancias de clase.

  Esto permite bajo acoplamiento en el código.

- Controlador: se mantiene apartada la lógica de negocio de la presentación. Con esto se logra mayor reutilización de código, un aumento de la cohesión y reducción del acoplamiento.
- Alta cohesión: la información que almacenan las clases es coherente y está relacionada con la clase. Esto permite que si surge algún error o cambio es más fácil de solucionar o realizar respectivamente.
- Bajo acoplamiento: las clases se encuentran lo menos ligadas posible. Con esto se busca que si surge un cambio otras clases no se afecten.

Se propone dividir el sistema en módulos:

- Un módulo general que abarca todo el sistema.
- Sub módulos que representan las distintas tareas.

Cada módulo puede ser representado por un paquete, en la figura 11 se puede observar el diagrama de paquetes del sistema. El sistema de divide entonces en 2 grandes paquetes el del generador de código y el del simulador de código, realizar esta división del sistema en paquetes favorece el cumplimiento de los patrones GRASP.

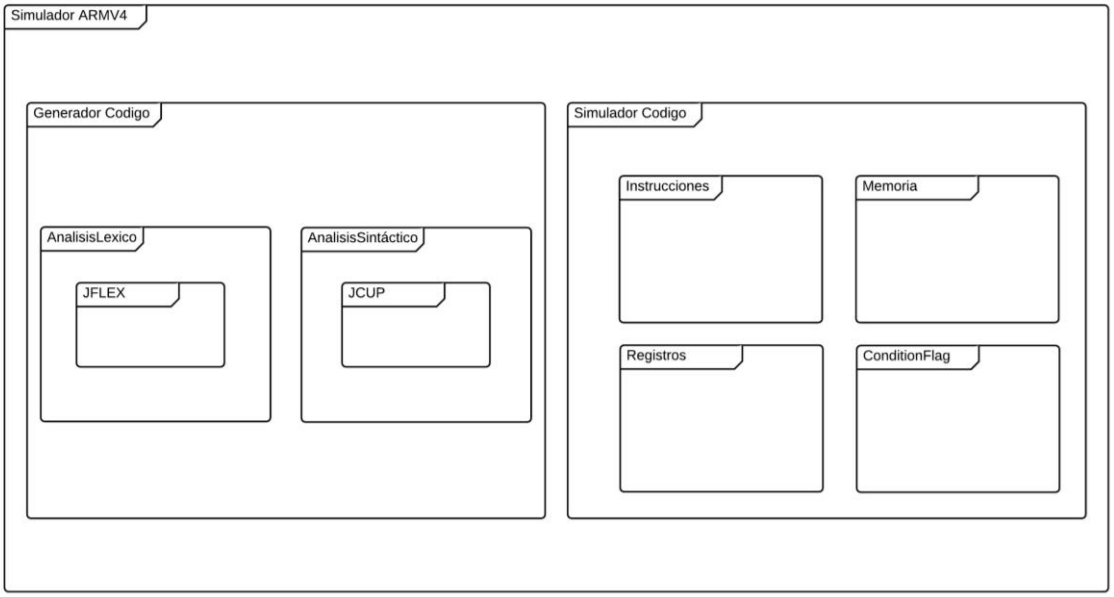

Figura 11. Diagrama de paquetes del sistema.

Propuesta 1: Diseño del ensamblador

Para el diseño del ensamblador se propone utilizar una sola clase que se encargue de realizar el análisis sintáctico y a la vez se encargue de realizar la generación de código máquina, por lo que dentro de esta clase se incluyen todos los métodos que se encargan de la generación del código. Sin embargo, al realizar esta implementación, la modularidad del código se pierde y además se hace muy robusto. El proceso de ensamblado del código se puede observar en el diagrama de alto nivel que se muestra en la figura 12.

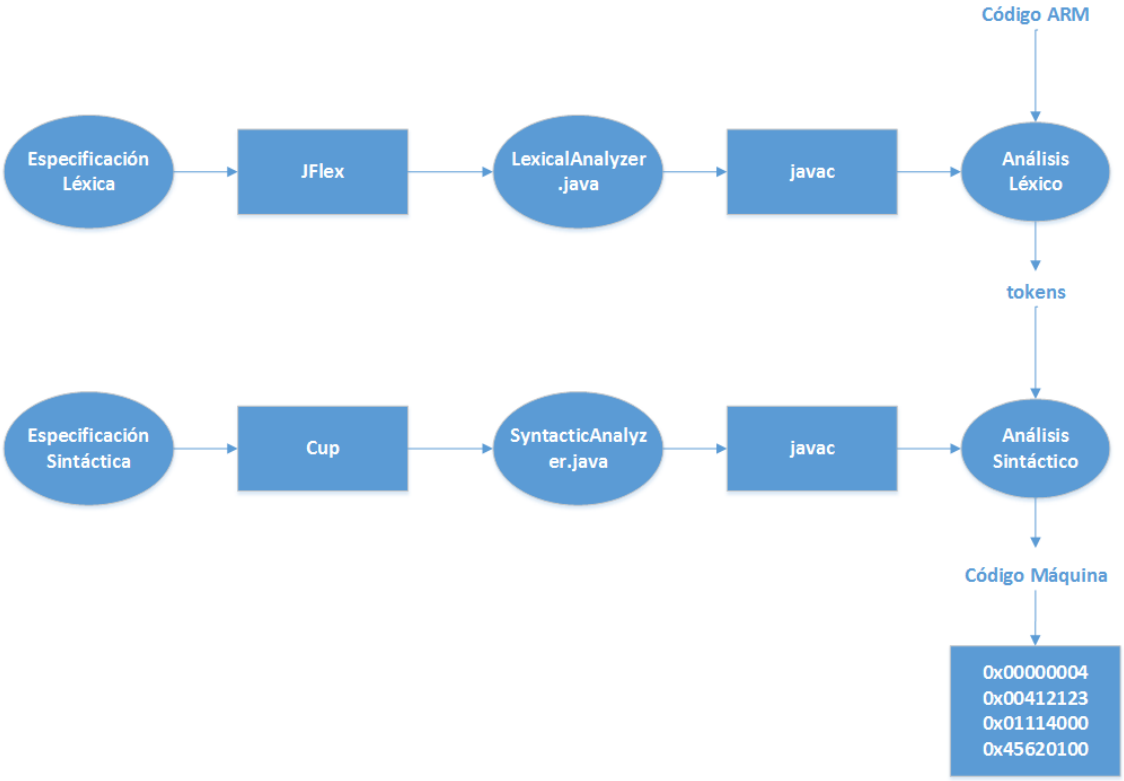

Figura 12. Diagrama de alto nivel, ensamblado de código.

Propuesta 2: Diseño del ensamblador

Para el diseño del ensamblador se propone separar la generación de código del análisis sintáctico, esto con el fin de mantener la modularidad en el código. Al realizar esta implementación se crea una clase nueva que se encarga solamente de generar el código. El proceso de ensamblado del código con esta implementación se puede observar en el diagrama de alto nivel que se muestra en la figura 13.

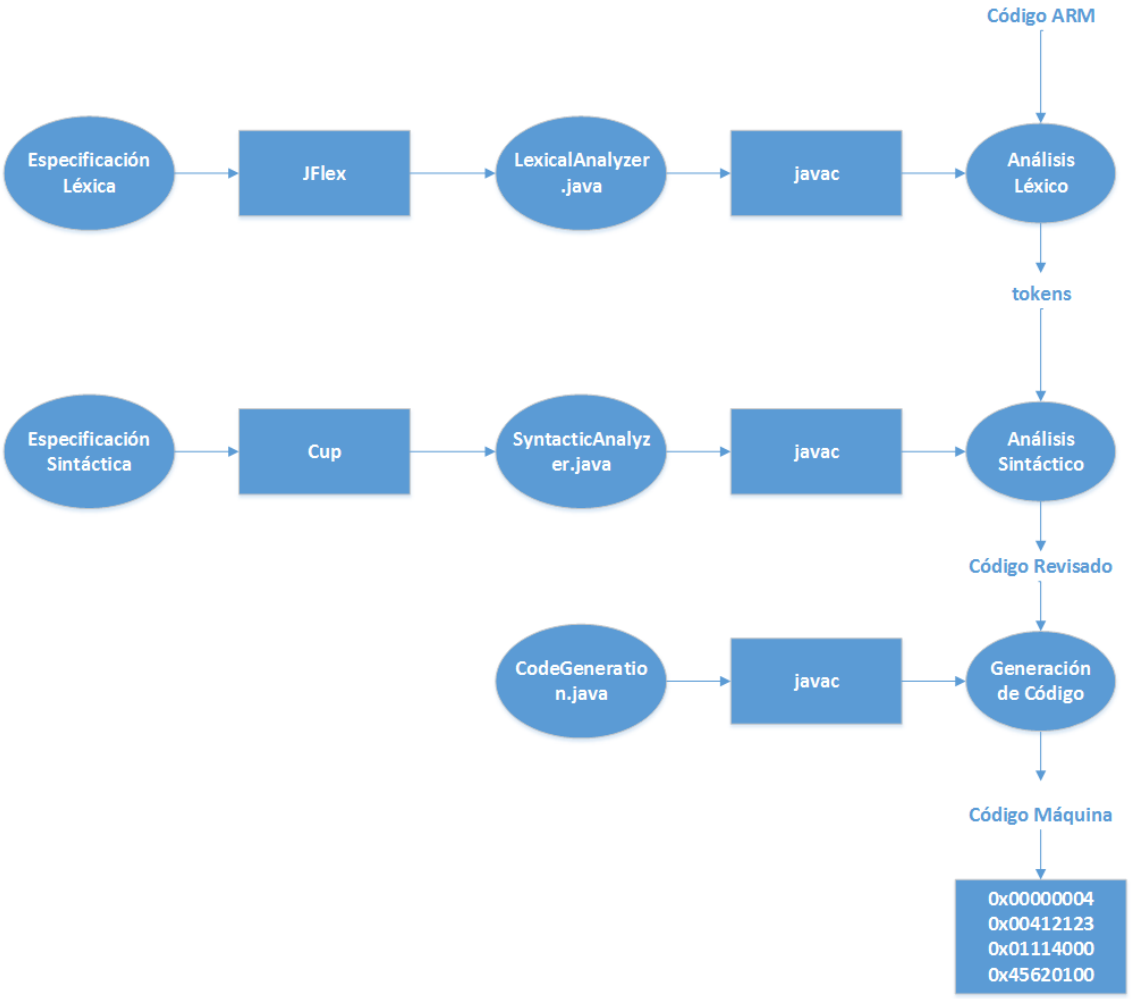

Figura 13. Diagrama de alto nivel, generación de código.

Elección de la propuesta del ensamblador

Se elige la propuesta número 2, esto porque:

- La propuesta número 1 encapsula en una clase los métodos que se encargan de generar pueden separar fácilmente, hacerlo de esta manera genera una clase difícil de modificar y de entender.
- A la propuesta número 2 se le pueden agregar más métodos en caso de que fuera necesario sin la necesidad de modificar la clase que se encarga de realizar el análisis sintáctico.

Propuesta 1: Diseño del simulador

Para el paquete “simulador de código”, se propone encapsular una clase que funcione como controlador, que reciba las instrucciones y dependiendo de cada instrucción la ejecute según como corresponde y de cómo salida los datos que

se deben mostrar en la interfaz gráfica. Esta clase controlador se encarga de manejar la clase de la memoria, registros y condition flag. El modelo que se refleja con esta propuesta es un MVC (modelo vista controlador), sin embargo, se debe hacer la salvedad que la vista que se devuelve a la interfaz gráfica son datos puros representados por listas en su mayoría, no se expande el encapsulamiento para realizar vistas puras. El proceso de simulación de código se puede observar en el diagrama de alto nivel que se muestra en la figura 14.

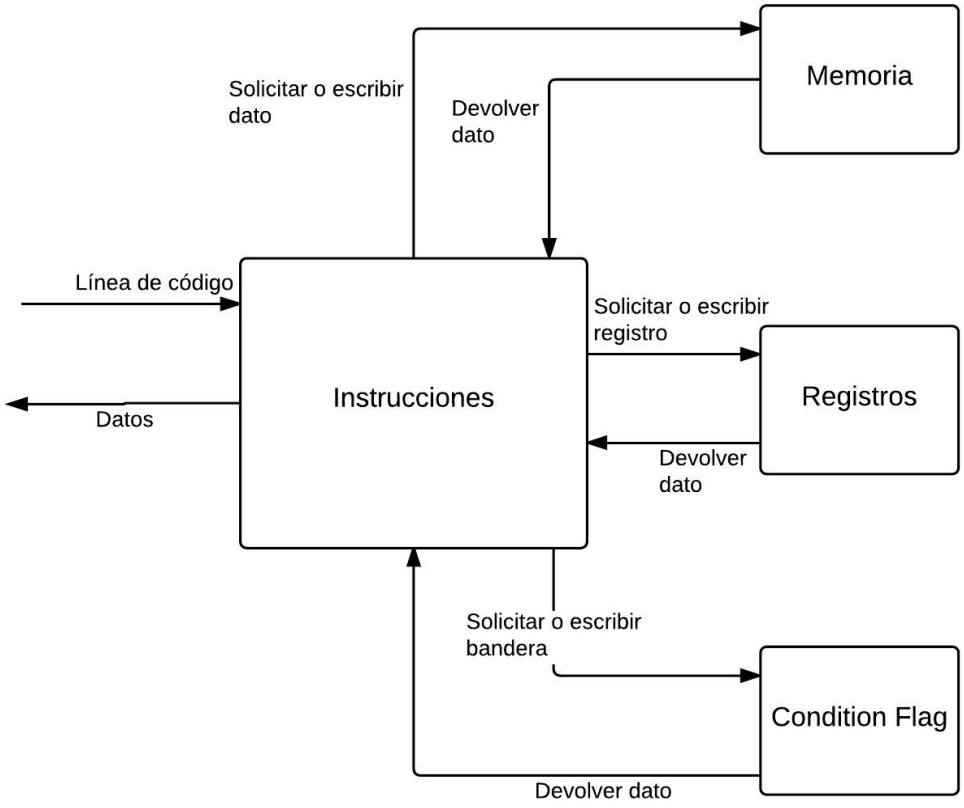

Figura 14. Diagrama de alto nivel, simulación de código.

El diagrama de alto nivel de todo el sistema propuesto a implementar se muestra en la figura 15.

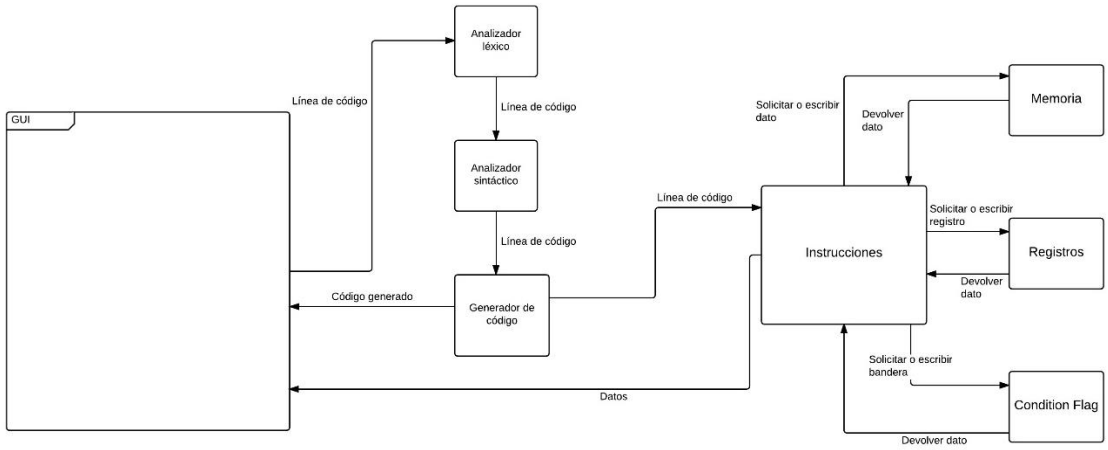

Figura 15. Diagrama de alto nivel, aplicación propuesta.

Propuesta 2: Diseño del simulador

Se propone utilizar una sola clase que encapsule a la memoria, el banco de registros, las banderas de la alu y las instrucciones. Esta clase se comunica con la interfaz. Esta propuesta se puede observar en la figura 16.

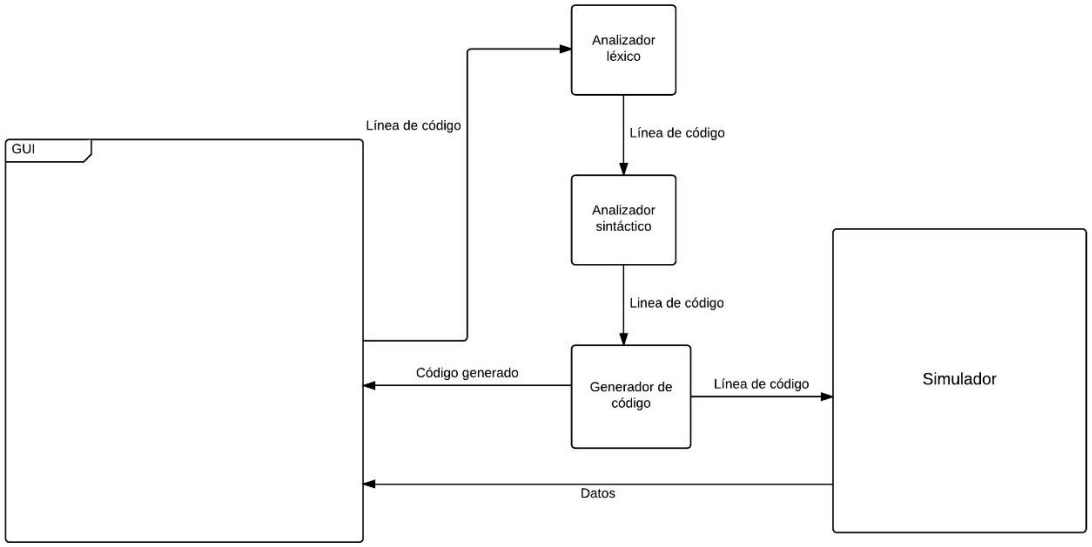

Figura 16. Diagrama de alto nivel, aplicación propuesta.

Elección de la propuesta del simulador

Se elige la propuesta número 1, esto porque:

- La propuesta número 2 encapsula en una clase muchas cosas que se pueden separar fácilmente, hacerlo de esta manera genera una super clase difícil de modificar y de entender.
- A la propuesta número 1 se le pueden aplicar patrones de diseño por la clara separación, a la 2 no.

Herramientas de ingeniería

NetBeans

NetBeans es un entorno de desarrollo integrado libre, hecho principalmente para el lenguaje de programación Java. Existe además un número importante de módulos para extenderlo. NetBeans IDE es un producto libre y gratuito sin restricciones de uso. La plataforma NetBeans permite que las aplicaciones sean desarrolladas a partir de un conjunto de componentes de software llamados módulos. Un módulo es un archivo Java que contiene clases de java escritas para interactuar con las APIs de NetBeans y un archivo especial (manifest file) que lo identifica como módulo. Las aplicaciones construidas a partir de módulos pueden ser extendidas agregándole nuevos módulos. Debido a que los módulos pueden ser desarrollados independientemente, las aplicaciones basadas en la plataforma NetBeans pueden ser extendidas fácilmente por otros desarrolladores de software.

VisualSim

Visual se ha desarrollado como una herramienta multiplataforma para hacer el lenguaje ensamblador ARM de aprendizaje más fácil. Permite la simulación de un subconjunto de instrucciones ARM, proporciona visualizaciones de conceptos claves únicos para programación en lenguaje ensamblador y por lo tanto ayuda a que la programación sea más accesible.

JFlex

JFlex es un generador de analizadores lexicográficos desarrollado por Gerwin Klein como extensión a la herramienta JLex desarrollada en la Universidad de Princeton. Está desarrollado en Java y genera analizadores en código Java. Es una reescritura de la herramienta JLex. JFlex se utiliza en conjunto con el generador de analizadores sintácticos LALR CUP. Los programas escritos para JFlex tienen un formato parecido a los escritos en PCLex; además todos los patrones regulares permitidos en Lex también son admitidos por JFlex. La instalación y ejecución de JFlex es la siguiente: se utiliza la biblioteca JFlex.jar, la cual sólo es necesario en tiempo de meta-compilación, siendo el analizador

generado totalmente independiente. La clase Main del paquete JFlex es la que se encarga de metacompilar el archivo con la especificación léxica .jflex de entrada; de esta manera, se llama de la siguiente forma: java JFlex.Main fichero.jflex lo que generará un fichero Yylex.java que implementa al analizador lexicográfico. [4]

JCup

Cup es un analizador sintáctico LALR desarrollado en el Instituto de Tecnología de Georgia (EE.UU.) que genera código Java y que permite introducir acciones semánticas escritas en dicho lenguaje. Utiliza una notación bastante parecida a la de PCYacc e igualmente basada en reglas de producción. Además Cup está escrito en Java, por lo que, para su ejecución, es necesario tener instalado algún JRE (Java Runtime Environment). [4]

Referencias

[1] I. Jacobson, El proceso unificado de desarrollo de software, 2000.

[2] A. Yakunin, «Quora,» 2003. [En línea]. Available: https://www.quora.com/Java-vs-Ruby-vs-Python-vs-c-which-one-should-be-chosen-for-development-on-the-server-side.

[3] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Prentice Hall, 2004.

[4] S. Gálvez and M. Mora, Java a tope: Compiladores. Traductores y compiladores con LEX/YACC, JFLEX/CUP y JAVACC. Málaga: Universidad de Málaga, 2005.