1. Simulacion en QtARMSim para operaciónes con Strings, Recursividad y manejo de Arreglos

Andreu Lechuga, 202073595-6, andreu.lechuga@usm.cl

1. Resumen

Se desarrollo un programa simulación en QtARMSim usando lenguaje ARM con el propósito de realzar 3 operaciones sobre un conjunto de entradas. La entrada de datos es manual, directo en el código a través de un *.data*, y las operaciones involucran (1) manejo de Strings, descomposición y comparación, (2) recursividad y (3) manejo de Arreglos en Memoria RAM, almacenamiento y operaciones aritméticas.

Cada uno de los tres procesos nombrados quedo plenamente funcional y a lo largo de este informe se detallará el proceso de concepción y desarrollo del código entero, así como la lógica detrás de cada proceso.

2. Introducción

En este proceso se plantearon, conceptualizaron y desarrollaron tres problemas que involucraban distintos aspectos del área de la programación, contextualizados y resueltos dentro del lenguaje de Assembly ARM.

Para este propósito se utilizó la herramienta QtARMSim, simulador de ARM construido sobre Python y Ruby.

El comportamiento del programa consiste en recibir un conjunto de datos en formato *.data*, interpretarlos y desarrollar 1 de 3 procedimientos utilizando la información. Finalizando cada procedimiento, el programa retorna un valor entero dependiendo del modo, binario (1|0) en el primer modo y decimal (0+) para el segundo y el tercero.

La entrada del programa estuvo compuesta por

	.data
modo	ID operación
numero	Parámetro (a) recursión simple (b) comparación numérica
str1	String 1
str2	String 2
lens1	Largo de str1
lens2	Largo de str2
arr	Arreglo de numero enteros

Por otro lado, el programa contempla las siguientes subrutinas generales

main	Decide el modo y redirige
log_0	Muestra 0 por consola y termina el programa
log_1	Muestra 1 por consola y termina el programa
log_r0	Muestra el valor en r0 por consola y termina el programa
endsim	End simulation Termina el programa

Y las siguientes subrutinas especificas

Modo 1	
mod1	Compara lens1 y lens2. Prepara variables
inv_s	Invierte str2 y lo almacena en memoria RAM
cmp_s	Compara str1 y str2 letra por letra y retorna
loop1	Auxiliar de cmp_s
Modo 2	
mod2	Prepara variables
f_rec	Evalúa condición base. Prepara y retorna
rec	Realiza //2 y //3 y guarda resultados en memoria. Lee de memoria el siguiente núm. a evaluar
Modo 3	
mod3	Prepara variables
loop3	Accede, evalúa y opera los extremos del arreglo.

La memoria de un computador se trabaja con valores hexadecimales, tanto las direcciones como los valores almacenados. Esto significa que cualquier operación y/o manejo que involucre datos de tipo Char/String serán procesados como hexadecimal, principalmente al momento de almacenar.

En programación, la recursividad consiste en un programa o función que se llama a sí mismo, respetando una condición de termino que evita su infinita repetición. Suele ser útil al momento de modelar comportamientos repetitivos y estandarizados. Una ventaja de la programación recursiva es su simplicidad al momento de programar e implementar.

3. Desarrollo

El orden de creación de cada módulo fue inverso al propuesto, por lo que procederé a presentarlos en orden cronológico (y resumida en lo demás).

El primer modulo fue el 3ro, encargado del manejo del arreglo de números enteros y la administración de una suma aritmética M entre ellos, siguiendo las siguientes condiciones: Si el numero extraído del comienzo del arreglo X_i es mayor estricto que el segundo número X_{2N-i}, del final del arreglo, se resta la diferencia a valor de la suma M. En caso contrario, la diferencia se suma a M.

La esencia del 3er modulo reside en el *loop3*, donde itera un total de N/2 veces (con N el largo del arreglo) e implementando 2 índices independientes con comportamientos contrarios. Uno de ellos (almacenado en r0) tiene un comportamiento creciente, mientras que el otro (r1) es decreciente. Entre ambos es posible acceder, en una misma iteración, al primer y ultimo valor del arreglo simultáneamente, para luego comparar y sumar.

El loop3 termina cuando el valor del índice r0 es mayor a r1, significando que ya se han recorrido la mitad del arreglo.

Siguiente fue el 2do modulo, dedicado a la función recursiva. La solución final de esta etapa se logro solo parcialmente, con perfecta claridad del problema y de su solucionarlo. La explicación del problema surge naturalmente al momento de explicar la solución, siendo esta la siguiente.

El problema asociado es el siguiente: Dado un numero N, se evalúa si es igual a 0. Si lo es, retorna 1. Si no lo es, la función retorna la suma entre si misma evaluada en N//2 y si misma evaluada en N//3. La recursividad continua hasta que se cumpla la primera condición.

Al momento de plantearme la problemática a resolver me percate de la necesidad de un registro manual de cada ramificación de la función. En otras palabras, cada llamado recursivo a la función implica dos cosas, (1) que el número actual N es distinto de 0 y (2) habrá dos valores para las dos llamadas recursivas siguientes, productos de N//2 y N//3.

De esta manera, desarrolle un arreglo que fuese almacenando los valores ramificados a evaluar, además de ir evaluándolos. Esto se logro con dos índices independientes, llamémosles i (en r6) y j (en r7). La función de i era mantener la información del largo de arreglo, apuntando siempre a la siguiente casilla disponible para **cargar** (store). Por otro lado, j buscaba recorrer el arreglo desde el comienzo hasta encontrarse con un 0 (he aquí el problema, explicare más adelante), siempre apuntando a la siguiente casilla a **leer** (load). De esta manera, j itera por cada número del arreglo antes del primer 0, evaluándolos en la función *f_rec*. Cada llamada a la función genera 2 appends (//2 y //3) en el arreglo, actualizando el valor de i en dos unidades por iteración. Al final de la función se vuelve a llamar al siguiente valor, dado por j. Esta parte de la función *f_rec* termina cuando j apunta a 0.

Una vez la función se evalúa en 0, entra al condicional encargado de cortar la recursividad. Ya adentro el procedimiento es secuencial, dejando atrás la recursividad. Para conocer el valor de retorno, se restan ambos índices recién mencionados, con el objetivo de obtener la cantidad de elementos entre la posición actual (primer 0) y el final del arreglo. Se asume (incorrectamente en algunos casos) que todo el resto de los elementos son 0 y, por lo tanto, retornaran 1 a la suma. Así, al obtener la cantidad de posiciones restantes (j - i), obtendremos en consecuencia el resultado que buscamos y retornamos, mostrándolo por el LCD.

El problema de este método, que funciona en algunos casos, es que a medida que la lista se va creando puede pasar que existan valores en el arreglo por evaluar **luego** del primer 0 encontrado. Si es así, el valor de retorno de mi maquina solo otorgara valores iguales o menores a valor real del número.

En la siguiente figura, representando la función recursiva en forma de árbol, se puede apreciar el orden en que ingresan los valores al arreglo. Mismo orden de salida.

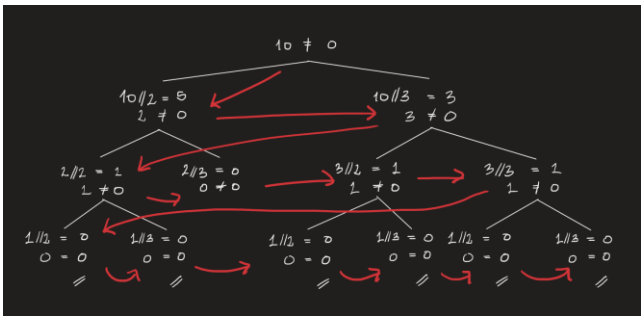

Figura 1: Árbol recursivo, ejemplo N = 10

En el caso de N = 10, al igual que N = 4 (ambos ejemplos dados por el departamento) el algoritmo funciona, puesto que una vez encuentra el primer 0, no quedan más números por evaluar. El primer numero positivo natural que falla, de acuerdo a esta lógica, es el 8 (retornando 7 en vez de 8, que es el resultado real del algoritmo para N = 8).

Dentro de los primeros 20 números positivos, el algoritmo falla en N = {8, 9, 12, 13, 15, 16, 17, 18, 19, 20}, acertando el resto. Estadísticamente acierta en un 50% dentro de un rango pequeño [0-20]. Sin embargo, el código tiene solución obvia y perfectamente posible de programar. Lo único que hacer es, luego de encontrar el primer 0, recorrer el resto de la lista utilizando un contador T. Por cada 0 que vea, sumara 1 a T. Por cada 1 que vea, sumara 2 a T, y así sucesivamente hasta el 7. Los valores de 0 a 7 son arbitrarios, considerando que en el 8 se produce la primera falla, y pueden verse en la Tabla 1 del anexo. Sin embargo, la lógica es sólida, puesto que mientras mas bajamos por las ramas, mas pequeños se hacen los valores de cada rama, simultáneamente. Esto asegura que, luego de encontrar el primer 0 en el arreglo, la posibilidad de encontrar otros números disminuye exponencialmente mientras más grande sea el número, siendo 7 muy improbable, incluso casi imposible.

La solución al problema no fue programada por problemas temporales y navideños, pero es una solución que se encuentra diagnosticada con claridad. Además es extremadamente fácil de programar.

Finalmente, el primero modulo se desarrolló de la siguiente manera. El trabajo mental fue menor, puesto que se siguieron las indicaciones del documento, utilizando las 3 subrutinas en seudocódigo descritas en él.

La subrutina principal seria *mod1*, en que suceden esencialmente dos cosas: (1) antes que nada, se comparan los largos de cada string para corroborar que son iguales. Si no lo son, se salta todo el modulo y retorna 0 por el LCD, usando la subrutina *log_0*.

Si son igual, procede a su segunda función (2) preparar las variables (registros) para llamar a la subrutina siguiente, *inv_s*. La siguiente subrutina llamada Invertidor De Strings (*inv_s*) almacena dos direcciones de memoria como punteros, la del str2 y la dirección de r13, para almacenaje en RAM. Luego, usando la información del largo del string, recorre str2 de atrás hacia adelante, almacenando el valor correspondiente a esa iteración en la memoria. La gracia es que, de esta manera, el string str2 queda des-invertido en la memoria.

Finalmente, la ultima subrutina llamada Comparador de String (*cmp_s*), utiliza un mismo índice i para recorrer ambos strings simultáneamente, comparando sus valores en casillas equivalentes dadas por la posición i. En caso de encontrar una discrepancia, retornara 0 usando *log_0*. En caso contrario, si se llega al final del string sin encontrar un par de valores distintos, entonces significa que str1 y str2 eran equivalentes y retorna 1, utilizando *log_1*.

Hasta aquí hemos descrito el proceso de concepción y realización de cada uno de los 3 módulos solicitados. En otras palabras, se podría decir que la lógica subyacente al código está cubierta. En la siguiente sección veremos un análisis en base a pruebas del código recién mencionado.

4. Resultados

En este apartado se presentarán las distintas pruebas de cada uno de los tres módulos creados para el funcionamiento del programa. Para esto se utilizarán los ejemplos corroborados que nos otorga el departamento, como parte del enunciado.

Ejemplo 1 – Modo 1

Valores

mode	1
numero	-
str1	“rata”
str2	“atar”
lens1	4
lens2	4
arr	-

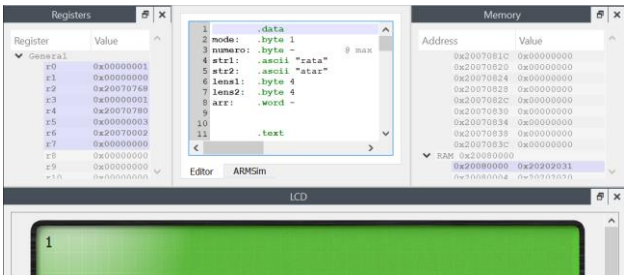

Figura 2: Consola LCD. Ejemplo 1.1

Ejemplo 2 – Modo 1

Valores

mode	1
numero	-
str1	“torre”
str2	“error”
lens1	5
lens2	5
arr	-

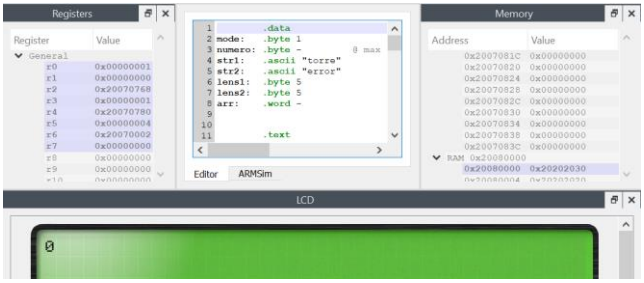

Figura 4: Consola LCD. Ejemplo 2.1

Ejemplo 3 – Modo 1

Valores

mode	1
numero	-
str1	“ola”
str2	“playa”
lens1	3
lens2	5
arr	-

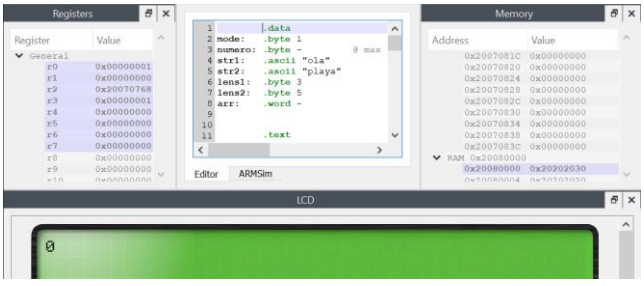

Figura 5: Consola LCD. Ejemplo 3.1

Ejemplo 1 – Modo 2

Valores

mode	2
numero	10
str1	“”
str2	“”
lens1	-
lens2	-
arr	-

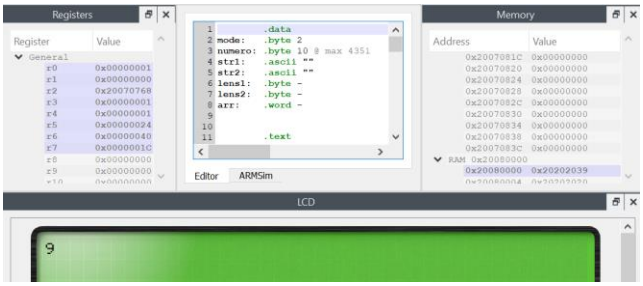

Figura 6: Consola LCD. Ejemplo 1.2

Ejemplo 2 – Modo 2

Valores

mode	4
numero	-
str1	“”
str2	“”
lens1	-
lens2	-
arr	-

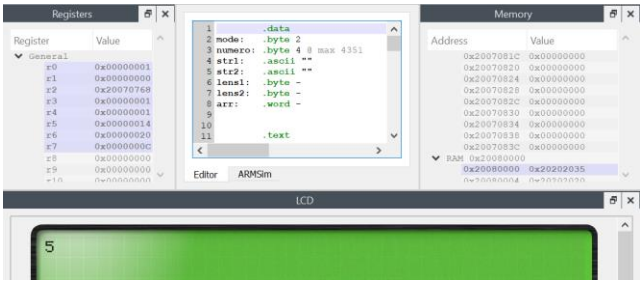

Figura 7: Consola LCD. Ejemplo 2.2

Ejemplo 1 – Modo 3

Valores

mode	3
numero	4
str1	“”
str2	“”
lens1	-

lens2	-
arr	1,2,3,4

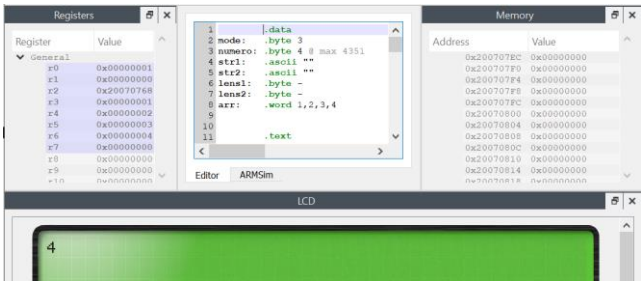

Figura 8: Consola LCD. Ejemplo 1.3

Ejemplo 2 – Modo 3

Valores

mode	3
numero	6
str1	“”
str2	“”
lens1	-
lens2	-
arr	1,2,3,4,5,6

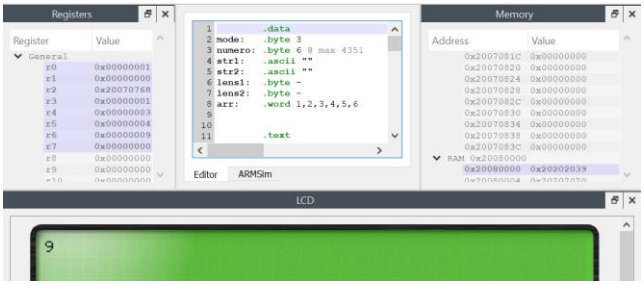

Figura 9: Consola LCD. Ejemplo 3.3

Ejemplo 3 – Modo 3

Valores

mode	3
numero	8
str1	“”
str2	“”
lens1	-
lens2	-
arr	8,6,4,2,2,4,6,8

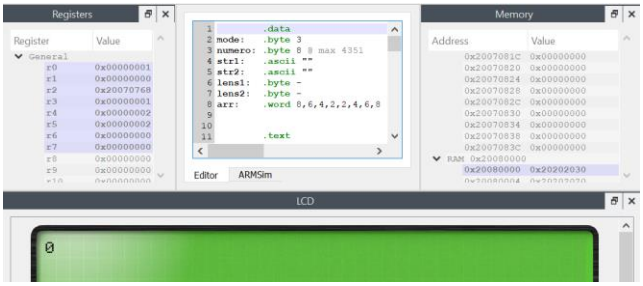

Figura 10: Consola LCD. Ejemplo 3.3

De esta manera dejamos demostrado el funcionamiento de los módulos. Pasamos a Análisis

5. Análisis

Honestamente, desde las primeras hasta la creación de cada uno de los módulos fue desafiante, como ningún otro lenguaje que haya visto. La lógica es tan rudimentaria, se siente torpe al comienzo, pero una vez comienza a entender, no el simulador, sino el lenguaje, la manera en que se almacenan los registros, la existencia de constantes que permiten comparaciones, el uso de memoria RAM y ROM entre tantas otras cosas todo comienza a caer junto.

El nivel del lenguaje es un desafío que, debo decir, me alegra haber tomado. No es fácil cambiar el paradigma interno. Aun así, luego de suficiente trabajo de prueba y error y de difícil documentación, fue posible, como todo, lograr un código estable y funcional.

6. Conclusión

Con el proyecto realizado se profundizo enormemente en el conocimiento sobre el Lenguaje ARM, adoptando conceptos y propios de un lenguaje assembly, tales como branch y comparaciones *cmp* en base a constantes del sistema y uso y tipos de memoria. Además, todo el conocimiento se vio a través del lente de la máquina, una vista propia del lenguaje y que de seguro vale mucho, en comparación con lenguajes de mayor nivel.

Se comprendió cuales son los recursos de la maquina y como los utiliza, que es la memoria, que tipos de memoria existen y como interactuar con ellos, como funcionan y que son los registros, entre otras cosas.

Mediante la superación, teórica y práctica, de bastantes imprevistos y dificultades, se completó con éxito el programa que se buscaba realizar.

7. Anexos

N	0	1	2	3	4	5	6	7	8
f_rec(N)	1	2	3	4	5	5	7	7	8/7

Tabla 1: Valores para N = {0,1,2,3,4,5,6,7,8}

8. Bibliografía

No hay bilbiografia