

UniversidadeVigo

Desarrollo de un ensamblador para ARM/THUMB

Johan Sebastián Villamarín Caicedo

Trabajo de Fin de Grado
Escuela de Ingeniería de Telecomunicación
Grado en Ingeniería de Tecnologías de Telecomunicación

Tutor
Martín Llamas Nistal

Curso 2021/2022

Contents

1	Introducción	2
1.1	Tecnologías	2
1.2	Arquitectura	2
2	Objetivos y metodología de trabajo	3
3	Resultados	4
3.1	Simulador ARM Thumb	4
3.1.1	Ensamblador	4
3.1.2	ARM CPU	5
3.1.3	Tests unitarios	5
3.2	Interfaz Web	6
3.2.1	Code Editor	7
3.2.2	Programa	8
3.2.3	Registros y flags	8
3.2.4	Memoria	10
4	Conclusiones y líneas futuras	11
5	Referencias	12
	Anexos	13
A	Subconjunto de intrucciones ARM Thumb	13

List of Figures

1	ARM Test Code Example	6
2	CPU Expected State Example	6
3	ARM Code Editor	7
4	Error Compiling Code	7
5	Loaded Program	8
6	Registers Menu	9
7	Change Register Value	9
8	Memory Menu	10
9	Initial Data Example	10
10	Memory Edit Example	11

1 Introducción

El desarrollo a bajo nivel de software para una arquitectura específica es siempre de gran complejidad, tanto por el propio lenguaje ensamblador como por la necesidad de disponer del hardware específico para probarlo. Debido a esto se recurre a crosscompilers, para compilar/ensamblar código para una arquitectura que no es la del host, y emuladores como QEMU para ejecutar y depurar el programa. Existen entornos como **Xilinx Vitis** que provee de lo necesario para desarrollar y depurar programas integrando QEMU como herramienta de simulación. En muchos casos estos entornos resultan pesados de correr y difíciles de configurar.

Para la arquitectura ARM existe **QtARMSim**, un simulador para ARM programado en python, hace uso de un crosscompiler para ensamblar el código y después emular la ejecución del mismo. Estos entornos tienen en muchos casos dependencias necesarias para su funcionamiento y en el caso de Xilinx Vitis ocupan una gran cantidad de almacenamiento.

Como solución a esto, y para el caso concreto de ARM Thumb, se decide desarrollar un **Simulador Web para ARM Thumb**. Al tratarse de una plataforma web se simplifica enormemente la instalación dado que bastaría con disponer de cualquier navegador y conexión a internet. Además esto hace que se pueda usar desde cualquier sistema operativo.

Actualmente existen distintas alternativas para este propósito como son QtARMSim, ARMThumbSim (web) o OakSim (web). Estos entornos ofrecen también la posibilidad de simular código ARM Thumb incluso en el caso de ARMThumbSim un display. Estas plataformas sin embargo no permiten la modificación de los valores de la cpu como si hace QtARMSim. Así este proyecto busca integrar ambas partes, la facilidad de uso e interfaz de usuario de una plataforma web, con el potencial de modificar aspectos de la cpu como permite QtARMSim.

1.1 Tecnologías

Para el desarrollo de este proyecto se emplearon las siguientes tecnologías:

1. **React**: JavaScript Framework para crear interfaces de usuario
2. **React-Redux**: States. Permite mantener un estado común para toda la aplicación
3. **TypeScript**: React con typescript. Módulo del simulador enteramente escrito en typescript.
4. **NodeJS**: JavaScript Engine, para ejecutar el servidor de desarrollo de React
5. **Docker**: Contenedores. Para despliegue ágil e instalación de dependencias
6. **Git**: Controlador de versiones. Repositorios en Github con CI/CD gracias a Github Workflows y Github Pages

1.2 Arquitectura

Los procesadores ARM, a excepción de los basados en ARMv6-M y ARMv7-M, tienen un total de 37 registros, en algunos casos pueden tener 3 adicionales para distintos usos. Un total de 31 registros de 32 bits para uso general y 6 registros de estado. Estos registros no están siempre accesibles al mismo tiempo, dependiendo del modo de ejecución del procesador están disponibles unos u otros. Los distintos modos de ejecución son usuario y sistema, FIQ (Fast Interrupt), supervisor, abort, IRQ (Interrupt) y undefined. El modo sistema es un modo usuario privilegiado para los sistemas operativos. En todos los modos hay disponibles 16 registros de uso general, uno de estado para los flags de la CPU (CPSR) y dependiendo del modo otro más de estado SPSR (Saved Program Status Register).

En este proyecto se considera que la ejecución será siempre en modo usuario por lo que tenemos 16 registros de uso general y un registro de estado CPSR. Estos procesadores tienen dos modos de operación, ARM (32-bits word-aligned instructions) y Thumb (16-bit halfword-aligned instructions) que será el modo a emular. Como indica la documentación oficial de ARM en modo Thumb están disponibles los registros inferiores (del 0 al 7) y los registros SP (Stack Pointer o r13), LR (Long Return o r14), PC (Program Counter o r15) y CPSR (sin acceso directo). El lenguaje ensamblador tiene acceso limitado a los registros superiores (del 8 al 15) para ser utilizados como un medio temporal de almacenamiento.

El registro CPSR contiene los flags de condiciones, por ejemplo si después de una suma hubiese overflow el bit C de acarreo (bit 29) se pondría a 1 así como el modo actual de ejecución. De los 32 bits de estado los 4 más significativos son los bits N (negativo), Z (cero flag), C (acarreo flag), V (signed overflow). Estos 4 bits son los empleados a la hora de hacer saltos condicionales, es decir para continuar la ejecución del programa en otro punto dependiendo de si se cumple o no una condición (por ejemplo si una resta da 0). Dado que el modo de ejecución será siempre Thumb se podrán ignorar los demás bits a la hora de emular la CPU.

Tanto en modo ARM como Thumb soporta los siguientes tipos de datos:

1. **Palabras:** Valor de 32-bits. Tiene que estar alineado a 4 en la memoria.
2. **Medias palabras:** Valor de 16-bits. Alineado a 2.
3. **Bytes:** Valor de 8 bits.

2 Objetivos y metodología de trabajo

El objetivo principal de este trabajo es desarrollar una plataforma web de software libre, inspirada en CREATOR , que emule la arquitectura ARM Thumb. En la que los usuarios puedan escribir en ensamblador, ensamblar y probar el programa de forma interactiva. Pudiendo ver en todo momento el estado de la cpu, sus registros y flags de estado, así como la memoria y su contenido.

Interpretar el código ensamblador en caso de que no contenga errores, de haberlos se le mostrará al usuario un mensaje de error así como la línea en la que se ha encontrado.

Funcionalidades mínimas del sistema:

1. **Comprobación de errores**
2. **Simulación del programa**
3. **Editor de código:** Editor de texto con syntax highlighting para ARM Thumb.
4. **Visualizador de registros:** Panel con los registros de la cpu y su valor. Así como el estado de los flags.
5. **Visualizador de programa:** Lista de instrucciones ensamblador que el usuario ha cargado a la cpu.
6. **Visualizador de memoria:** Panel con la memoria de la cpu.

Como objetivo adicional se busca que la aplicación se lo mas intuitiva posible y permita modificar el estado de la cpu y la memoria que se está simulando. Con facilidades para modificar los valores de ambos.

A la hora del desarrollo se dividió el proyecto en dos subproyectos, por un lado el emulador y por otro la plataforma web, que integrará siempre la última versión del emulador. Para ambos subproyectos se emplea un controlador de versiones, en este caso **git**, con sendos repositorios en github (ARM Thumb Simulator y Web ARM Thumb Simulator). Haciendo uso de **Github Workflows** para CI/CD, ejecutándose los tests en cada push al repositorio del simulador y para despliegue automático de la web gracias a github pages.

La distribución de trabajo fue también en dos fases, la fase inicial se centró en el desarrollo del emulador y crear un entorno de trabajo con tests unitarios. Para crear el entorno de trabajo se utilizó **yarn** como gestor de paquetes y el paquete **jest** para crear una test suite para el emulador. También se creó una interfaz web mínima que integrase el emulador.

La segunda fase se centró en corrección de errores, tests e integración de la plataforma web con el emulador. Por último se abordó el apartado visual empleando **react bootstrap**.

3 Resultados

Se llega a una plataforma web que permite escribir código en ensamblador, parsea y comprueba errores y en caso de que no hubiese simula la ejecución del programa. Muestra al usuario los registros y su contenido, los flags de estado, la memoria disponible y el programa cargado. Soporta la edición de registros y memoria, se puede introducir un valor específico para cambiar el contenido de estos.

3.1 Simulador ARM Thumb

Módulo en TypeScript con todas las funciones necesarias para la simulación de código escrito en ARM Thumb. Este módulo se desarrolló de forma separada para su posterior integración en el sistema final. Gracias a esta división se realizaron pruebas unitarias de cada instrucción implementada para asegurar que el comportamiento del simulador es el esperado.

Puede ser empleado por cualquier otra aplicación que use TypeScript. Está dividido en tres grandes partes:

1. **Ensamblador**: Comprobación de errores.
2. **CPU**: Contiene los registros, flags y la memoria.
3. **Tests Unitarios**: Conjunto de tests para cada operación de la cpu.

3.1.1 Ensamblador

Implementa la lógica necesaria para la comprobación de errores, interpretar las directivas de ensamblador y crear para cada operación un objeto con la información necesaria para su ejecución. Gracias a TypeScript se definen los siguientes tipos:

1. **CompilerError**. Contiene la causa del error y la línea.
2. **Operand**. Argumento de una operación, consta de tipo (LowRegister, DecImmediate, ...) y valor.
3. **Instruction**. Operación a ejecutar. Campos: operation, operands, break y label.
4. **Program**. Programa ensamblado. Consta de un array de instrucciones y un posible error.

Los datos iniciales del programa especificados mediante las directivas de ensamblador (.byte, .asciz, ...) son almacenados en un array de números enteros para después cargarse en la cpu junto con el programa. Se exporta una única función **compileAssembly** que recibe una string con el código ensamblador y retorna un objeto correspondiente al programa ensamblado y la memoria inicial que este necesita, el programa podrá tener un error para su gestión posterior. Consultar anexo para ver la lista de operaciones y directivas.

3.1.2 ARM CPU

Para simular la cpu se define un objeto javascript con los siguientes campos:

Registros (regs)

Objeto javascript script que contiene un par clave valor por cada registro. Para ARM Thumb desde r0 a r15 inicializados a 0, menos el registro SP (r13) que se inicializa con la dirección de memoria del stack y se actualiza al realizar operaciones en él.

CPSR (Z, N, C, V)

Por simplicidad, y dado que se asume que la ejecución es siempre en el mismo, como se indica previamente, se simulan únicamente los 4 bits más significativos (Z, N, C y V), cada uno es miembro del objeto cpu de tipo boolean.

Memoria (memory)

Simulada mediante un array de número enteros, tipo number. Dado que javascript emplea 32 bits para representar número enteros se inicializa un array de un tamaño por defecto (128), la mitad del array como memoria para almacenamiento y la otra mitad para el stack. El tamaño de esta memoria es modificable a la hora de inicializar la CPU.

Programa (program)

Se define un tipo de TypeScript **Instruction**, este objeto tiene el tipo de operación a ejecutar (MOV, ADD, SUB, etc), los argumentos de la operación y una posible label. El programa cargado en la CPU se define como un array de estas instrucciones.

Error del ensamblador (error)

Se define el tipo **CompilerError** que contiene la causa del error y la línea en la que se ha encontrado. Previamente a cargar el código ensamblado se comprueba si el proceso se ha detenido debido a un error.

Además de estos campos cuenta con los métodos:

1. **run()**: Ejecuta el programa cargado. Retorna al finalizar la ejecución o al encontrar un breakpoint.
2. **step()**: Ejecuta la siguiente instrucción indicada por el registro PC.
3. **execute(ins: Instruction)**: Empleado por los dos anteriores, recibe la instrucción a ejecutar y realiza la operación en cuestión. Implementa la lógica de cada operación del subconjunto de operaciones de ARM Thumb. Activando o desactivando flags según corresponda y modificando los registros o memoria.
4. **loadAssembly(assembly: string)**: Llama a **compileAssembly** para ensamblar y posteriormente cargar el código ensamblador, en caso de que no haya errores. Inicializa la memoria con los valores de la memoria inicial que devuelve el ensamblador. En caso de haber algún error lo almacena en el correspondiente campo de la cpu y retorna sin cargar el programa ni la memoria.

3.1.3 Tests unitarios

Durante todo el desarrollo del simulador se tuvo presente la necesidad de realizar tests para comprobar el correcto funcionamiento. Por ello se mantuvo en todo momento un conjunto de tests para cada operación.

Cada test tiene un archivo con el código ARM Thumb a ensamblar y ejecutar, como primer paso se ensambla el código mediante un crosscompiler para ARM. De no haber error se hace mediante el simulador, se comprueba que no hayan ocurrido errores y por último se ejecuta.

```

src > _tests_ > asm > ASM mov.S
1  .text
2      mov r1, #1      @ r1 = 1
3      mov r2, #0x04   @ r2 = 4
4      mov r3, r1      @ r3 = 1
5      mov r8, r2      @ r8 = 4
6      mov r1, r8      @ r1 = 4
7      mov r9, r8      @ r9 = 4
8      mov sp, r8      @ sp = 4
9      mov r5, sp      @ r5 = 4
10     @ r1 = 4, r2 = 4, r3 = 1, r8 = 4, r9 = 4

```

Figure 1: ARM Test Code Example

Una vez finalizada la ejecución se comprueba si existe el archivo correspondiente con el estado esperado de la cpu. De no existir el estado de la cpu tras la ejecución es volcado a un archivo temporal con el nombre del test (add.json.tmp). De este modo si el estado es correcto basta con eliminar la extensión .tmp para que en los siguientes tests este sea el estado esperado.

```

src > _tests_ > asm > {} mov.S.json > ...
1  {
2      "regs": {
3          "r0": 0,
4          "r1": 4,
5          "r2": 4,
6          "r3": 1,
7          "r4": 0,
8          "r5": 4,
9          "r6": 0,
10         "r7": 0,
11         "r8": 4,
12         "r9": 4,
13         "r10": 0,
14         "r11": 0,
15         "r12": 0,
16         "r13": 4,
17         "r14": 0,
18         "r15": 16
19     },
20     "z": false,
21     "n": false,
22     "c": false,
23     "v": false,
24     "memory": [],
25     "memSize": 0,
26     "stackSize": 0,
27     "program": [

```

Figure 2: CPU Expected State Example

Cada test comprueba una instrucción de forma independiente, salvo en algunos casos donde se prueban variantes de una sola en un mismo test (ej: ldr, ldrb, ldrh) u otros donde es necesaria otra operación. Por ejemplo para realizar un salto condicional es necesario utilizar una operación previa, por ejemplo de comparación. Además de asegurar que el código en ensamblador es correcto mediante el crosscompiler el estado final de la cpu se comprobó manualmente mediante el simulador **QtARMSim**.

3.2 Interfaz Web

Como interfaz de usuario se desarrolla una plataforma web usando React-TypeScript, el objeto cpu se incluye como parte del estado de la cpu gracias a React-Redux lo que permite mantener un estado global común para los distintos componentes. El estado global de la aplicación consta de tres campos:

1. **cpu**: El objeto CPU del emulador.
2. **assembly**: El código ensamblador que el usuario ha introducido.
3. **error**: Mensaje de error para mostrar en la interfaz, es el mismo error de la cpu pero se mantiene en el estado para más rápido acceso, en caso de haber error.

Este estado es utilizado por los distintos componentes de la interfaz para mantener siempre la información que se muestra actualizada. Cada componente selecciona los campos del estado que necesita acceder y es recargado de forma automática si hay algún cambio.

La estética de la interfaz está inspirada en un simulador web que permite simular distintas arquitecturas así como crear una propia (CREATOR)

3.2.1 Code Editor

Editor de texto con highlighting para ARM Thumb. Hace uso de React CodeMirror que tiene soporte para distintos lenguajes. En este caso se extiende la librería para soportar ARM Thumb. Además incorpora múltiples atajos de teclado para mover, cortar o seleccionar texto.

```

1 ; This a simple example in arm thumb!
2
3 .text ; Start of .text section. This is where the code will be placed.
4     mov r0, #2    ; r0 = 2
5     add r0, #1    ; r0 = 3
6     add sp, #8    ; sp = sp + 8
7     add sp, #4    ; sp = sp + 4 = 12
8
9     mov r1, #0x2  ; r1 = 2
10    add r1, r1    ; r1 = 4
11    mov r9, r1    ; r9 = 4
12    mov r8, r9    ; r8 = 4

```

Figure 3: ARM Code Editor

Este componente muestra en todo momento el valor de **assembly** del estado de la aplicación y es el encargado de actualizarlo con la entrada del usuario. Una vez escrito el programa el usuario puede cargarlo en la CPU mediante el botón de **Load Program** en la parte superior de la interfaz.

De haber algún error en el código se notificaría al usuario a través de un PopUp indicando la causa y la línea del error.

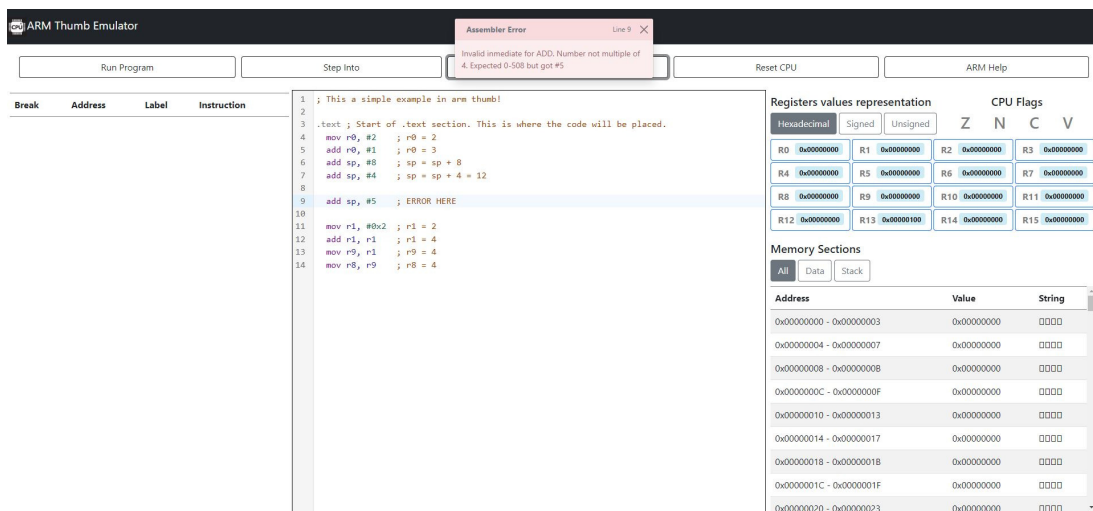


Figure 4: Error Compiling Code

3.2.2 Programa

Una vez cargado el programa este estará en el panel izquierdo, donde se podrá ver la siguiente instrucción a ejecutar, los breakpoints del programa así como todas las demás instrucciones cargadas.



Break	Address	Label	Instruction
	0x00		mov r0, #2
	0x02	label_example	add r0, #1
	0x04		add r13, #8
	0x06		add r13, #4
	0x08	here	mov r1, #0x2
	0x0a		add r1, r1
	0x0c		mov r9, r1
	0x0e	end	mov r8, r9

Figure 5: Loaded Program

Además se pueden poner/quitar breakpoints con un simple click. Para ejecutar el programa una vez cargado tenemos dos opciones:

1. **Run Code:** Ejecuta el programa hasta que se llega al final o a un breakpoint.
2. **Step Into:** Ejecuta instrucción a instrucción el programa.

3.2.3 Registros y flags

Este componente se encarga de mostrar en todo momento el valor de cada registro de la cpu. En este caso los 16 registros de uso general, además también muestra el estado de los flags.

Obtiene del estado de la aplicación, gracias a react-redux, el objeto cpu. Este estado se actualiza tras cada operación. De estos 16 registros cabe destacar los registros especiales:

1. **R13 (SP):** Stack Pointer. Se actualiza automáticamente al hacer push o pop
2. **R14 (LR):** Long Return Register. Se actualiza automáticamente al hacer un salto largo.
3. **R15 (PC):** Program Counter Register. Se actualiza automáticamente con la dirección de la próxima instrucción

Registers values representation				CPU Flags			
<div>Hexadecimal</div> <div>Signed</div> <div>Unsigned</div>				Z	N	C	V
R0 0x00000000	R1 0x00000000	R2 0x00000000	R3 0x00000000				
R4 0x00000000	R5 0x00000000	R6 0x00000000	R7 0x00000000				
R8 0x00000000	R9 0x00000000	R10 0x00000000	R11 0x00000000				
R12 0x00000000	R13 0x00000100	R14 0x00000000	R15 0x00000000				

Figure 6: Registers Menu

Como se puede ver en la figura 6 cada registro aparece por defecto en formato hexadecimal. Además se puede cambiar el formato entre hexadecimal, signed y unsigned con un simple click.

El valor de cada registro puede ser modificado en cualquier momento. Clickando en el registro que queremos cambiar abriremos un menú de modificación donde podremos ver el valor actual en todos los formatos disponibles así como introducir un nuevo valor.

Register R8

Hex.	0x00000000
Signed	0000000000
Unsigned	0000000000

Register value

0x00000000

Not a valid 32 bits number. Not a Number
New value of register. A 32 bits number.

Save

Registers values representation

Hexadecimal Signed Unsigned

R0 0x00000000	R1 0x00000000	R2 0x00000000	R3 0x00000000
R4 0x00000000	R5 0x00000000	R6 0x00000000	R7 0x00000000
R8 0x00000000	R9 0x00000000	R10 0x00000000	R11 0x00000000
R12 0x00000000	R13 0x00000100	R14 0x00000000	R15 0x00000000

Memory Sections

All Data Stack

Figure 7: Change Register Value

Además se comprueba en todo momento que la entrada del usuario sea un n^o de 32 bits válido. Ya sea en formato hexadecimal o decimal, no se admiten números negativos, y se habilita el botón de guardar solo si la entrada es correcta.

En la parte superior derecha tenemos los flags de la cpu. Estos flags se activan en algunas operaciones como las de comparar dependiendo del resultado de dicha operación. En caso de ejecutar el programa sin ir paso a paso el estado de los flags sería siempre el que hay al final de la ejecución, igual que los registros. Los flags activos se representan en rojo y en gris cuando están a 0.

3.2.4 Memoria

Debajo de los registros está la memoria de la cpu, se inicializa por defecto a 64 palabras (4 bytes) como zona libre y 64 más para el stack. Se puede visualizar toda la memoria a la vez, sólo la zona de datos o sólo el stack.

Address	Value	String
0x00000000 - 0x00000003	0x00000000	□□□□
0x00000004 - 0x00000007	0x00000000	□□□□
0x00000008 - 0x0000000B	0x00000000	□□□□
0x0000000C - 0x0000000F	0x00000000	□□□□
0x00000010 - 0x00000013	0x00000000	□□□□
0x00000014 - 0x00000017	0x00000000	□□□□
0x00000018 - 0x0000001B	0x00000000	□□□□
0x0000001C - 0x0000001F	0x00000000	□□□□

Figure 8: Memory Menu

El usuario puede utilizar directivas de ensamblador para inicializar la memoria con cualquier valor que necesite durante la ejecución. En caso de que el programa necesite más memoria de la disponible esta crece de forma automática, esto puede pasar en operaciones como push o en programas que soliciten guardar muchos datos iniciales.

ARM Thumb Emulator

Run Program

Step Into

Emulator Message

Code compiled without errors

Load Program

Reset CPU

ARM Help

Break	Address	Label	Instruction
	0x00		mov r0, #2
	0x02		add r0, #1
	0x04		add r13, #8
	0x06		add r13, #4
	0x08		mov r1, #0x2
	0x0a		add r1, r1
	0x0c		mov r9, r1
	0x0e		mov r8, r9

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

This is a simple example in arm thumb!

.text : Start of .text section. This is where the code will be placed.

mov r0, #2 ; r0 = 2

add r0, #1 ; r0 = 3

add sp, #8 ; sp = sp + 8

add sp, #4 ; sp = sp + 4 = 12

mov r1, #0x2 ; r1 = 2

add r1, r1 ; r1 = 4

mov r9, r1 ; r9 = 4

mov r8, r9 ; r8 = 4

.data

.ascii "Hello World"

.align 2

.byte 0xFF

.align 2

.hword 0xAAD0

.align 2

.word 0xB8888888

Registers values representation

CPUs Flags

Hexadecimal

Signed

Unsigned

Z

N

C

V

R0 0x00000000

R1 0x00000000

R2 0x00000000

R3 0x00000000

R4 0x00000000

R5 0x00000000

R6 0x00000000

R7 0x00000000

R8 0x00000000

R9 0x00000000

R10 0x00000000

R11 0x00000000

R12 0x00000000

R13 0x00000100

R14 0x00000000

R15 0x00000000

Memory Sections

All

Data

Stack

Address	Value	String
0x00000000 - 0x00000003	0x6C6C548	Hello
0x00000004 - 0x00000007	0x6F57206F	o Wo
0x00000008 - 0x0000000B	0x0646C72	rdD
0x0000000C - 0x0000000F	0x000000FF	yDD
0x00000010 - 0x00000013	0x00000A00	DD
0x00000014 - 0x00000017	0xB8888888	----
0x00000018 - 0x0000001B	0x00000000	DDDD

Figure 9: Initial Data Example

Como se puede ver en este ejemplo hay diferentes directivas para almacenar desde strings, bytes, medias palabras o palabras. Así como directivas para alinear el próximo dato.

Además del mismo modo que para los registros los valores de la memoria son fácilmente modificables, para ello basta con clicar en la zona de memoria cuyo valor deseamos modificar y se nos abrirá un menú similar al visto para modificar un registro. Al igual que para los registros la entrada del usuario se comprueba antes de habilitar el botón de guardar para evitar errores.

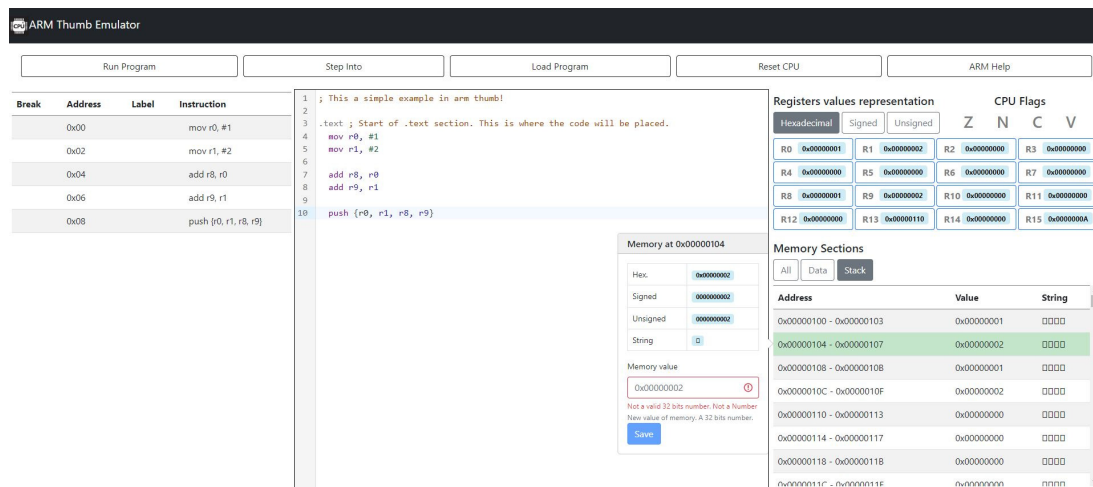


Figure 10: Memory Edit Example

4 Conclusiones y líneas futuras

Desde el inicio del proyecto se hizo presente la necesidad de implementar pruebas unitarias, dada la necesidad de probar cada operación de la cpu. Además de eso la calidad de las pruebas es un factor muy importante en todo proyecto ya que de ello depende que se encuentren todos los posibles errores.

Por otra parte el potencial que presenta la disponibilidad de una plataforma web con la capacidad de simular código ensamblador de una arquitectura específica es muy elevado, desde para hacer pequeñas pruebas de rutinas a por ejemplo el uso desde el hogar sin el equipo del laboratorio.

Una vez finalizado el trabajo se alcanza el objetivo principal, se obtiene una plataforma web capaz de interpretar código ensamblador para ARM Thumb, centrada en la simulación del mismo ya que no se obtiene el código objeto en ARM Thumb. Permitiendo a los usuarios programar y simular la ejecución de la cpu. Con comprobación de errores y mensajes de error lo más específicos posibles para cada error. Desplegada automáticamente en github pages: ARM Thumb Emulator. Además el desarrollo modular permite la integración del simulador por parte de cualquier otra aplicación.

Cómo líneas futuras, esta aplicación ganaría aún mas potencial de ser capaz de generar un archivo ejecutable para la arquitectura ARM, permitiendo también obtener el código objeto de cada operación. Mediante el desarrollo de un backend y el uso de un crosscompiler se podría enviar el código a ensamblar y recibir el archivo ejecutable listo para cargar en el equipo donde se va a ejecutar y con herramientas como **objdump** obtener la información necesaria de dicho archivo (secciones de datos, código objeto de cada operación, ...). Esto abre además la posibilidad de permitir registrar usuarios, guardar y cargar programas.

5 Referencias

1. **ARM Developer Documentation.** <https://developer.arm.com/documentation/ddi0210/c/Programmer-s-Model/Registers/The-Thumb-state-register-set>, <https://developer.arm.com/documentation/ddi0210/c/Programmer-s-Model/Operating-modes>, <https://developer.arm.com/documentation/ddi0210/c/Introduction/Instruction-set-summary/Thumb-instruction-summary?lang=en>
2. **CREATOR.** <https://creatorsim.github.io/creator/>, <https://github.com/dcamarmas/creator>
3. **Xilinx Vitis.** <https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html>
4. **QtARMSim.** <http://lorca.act.uji.es/project/qtarmsim/>
5. **OakSim.** <https://wunkolo.github.io/OakSim/>, <https://github.com/Wunkolo/OakSim>
6. **React Documentation.** <https://es.reactjs.org/>
7. **React-Redux.** <https://react-redux.js.org/>, <https://react-redux.js.org/using-react-redux/usage-with-typescript>
8. **React CodeMirror.** <https://uiwjs.github.io/react-codemirror/>
9. **CodeMirror Documentation.** <https://codemirror.net/docs/>, <https://github.com/codemirror/legacy-modes>
10. **React Bootstrap.** <https://react-bootstrap.github.io/>

Anexos

A Subconjunto de instrucciones ARM Thumb

Conjunto de instrucciones ARM Thumb (1/2)

Operación		Ensamblador	Acción	Actualiza	Notas
Mover	Inmediato 8 bits	mov Rd, #Inm8	Rd ← Inm8	N Z	Rango Inm8: 0–255.
	Lo a Lo	mov Rd, Rm	Rd ← Rm	N Z	
	Hi a Lo, Lo a Hi, Hi a Hi	mov Rd, Rm	Rd ← Rm		
Sumar	Inmediato 3 bits	add Rd, Rn, #Inm3	Rd ← Rn + Inm3	N Z C V	Rango Inm3: 0–7.
	Inmediato 8 bits	add Rd, Rd, #Inm8	Rd ← Rd + Inm8	N Z C V	Rango Inm8: 0–255.
	Lo a Lo	add Rd, Rn, Rm	Rd ← Rn + Rm	N Z C V	
	Hi a Lo, Lo a Hi, Hi a Hi	add Rd, Rd, Rm	Rd ← Rd + Rm		
	Valor a SP	add SP, SP, #Inm	SP ← SP + Inm		Rango Inm: 0–508 (alineado a palabra).
	Crear dirección desde SP	add Rd, SP, #Inm	Rd ← SP + Inm		Rango Inm: 0–1020 (alineado a palabra).
Restar	Inmediato 3 bits	sub Rd, Rn, #Inm3	Rd ← Rn – Inm3	N Z C V	Rango Inm3: 0–7.
	Inmediato 8 bits	sub Rd, Rd, #Inm8	Rd ← Rd – Inm8	N Z C V	Rango Inm8: 0–255.
	Lo a Lo	sub Rd, Rn, Rm	Rd ← Rn – Rm	N Z C V	
	Valor de SP	sub SP, SP, #Inm	SP ← SP – Inm		Rango Inm: 0–508 (alineado a palabra).
	Negar	neg Rd, Rn	Rd ← –Rn	N Z C V	
Multiplicar	Multiplica	mul Rd, Rm, Rd	Rd ← Rm * Rd	N Z	
Comparar	Compara	cmp Rn, Rm	Act. flags según Rn – Rm	N Z C V	Cualquier registro a cualquier registro.
	Compara con negado	cmn Rn, Rm	Act. flags según Rn + Rm	N Z C V	
	Inmediato	cmp Rn, #Inm8	Act. flags según Rn – Inm8	N Z C V	
Lógicas	AND	and Rd, Rm	Rd ← Rd AND Rm	N Z	
	AND NOT (borrar bits)	bic Rd, Rm	Rd ← Rd AND NOT Rm	N Z	
	OR	orr Rd, Rm	Rd ← Rd OR Rm	N Z	
	XOR (or exclusiva)	eor Rd, Rm	Rd ← Rd XOR Rm	N Z	
	NOT	mvn Rd, Rm	Rd ← NOT Rm	N Z	
	Comprueba bits	tst Rn, Rm	Act. flags según Rn AND Rm	N Z	
Desplazar	Lógico a la izquierda	lsl Rd, Rm, #Shift	Rd ← Rm << Shift	N Z C	Rango Shift: 0–31
		lsl Rd, Rd, Rs	Rd ← Rd << [Rs] _{7:0}	N Z C	
	Lógico a la derecha	lsr Rd, Rm, #Shift	Rd ← Rm >> Shift	N Z C	Rango Shift: 0–31
		lsr Rd, Rd, Rs	Rd ← Rd >> _l [Rs] _{7:0}	N Z C	
	Aritmético a la derecha	asr Rd, Rm, #Shift	Rd ← Rm >> _a Shift	N Z C	Rango Shift: 0–31
		asr Rd, Rd, Rs	Rd ← Rd >> _a [Rs] _{7:0}	N Z C	
	Rotación a la derecha	ror Rd, Rd, Rs	Rd ← Rd ROR [Rs] _{7:0}	N Z C	
Directivas del ensamblador					
.align N	Siguiente dato empieza en dir. múltiplo de 2 ^N .	.equiv símbolo, expr	Como .equ , pero da error si existe símbolo.		
.ascii "cadena"	Inicializa una zona de memoria con los caracteres UTF-8 de cadena.	.eqv	Equivalente a .equiv .		
.asciz "cadena"	Idem que la anterior, pero termina con 0.	.hword valor	Inicializa una media palabra a valor.		
.balign N	Siguiente dato empieza en dir. múltiplo de N.	.quad valor	Inicializa una doble palabra a valor.		
.byte valor	Inicializa un byte a valor.	símbolo .req rd	Define símbolo como un alias para rd.		
.text	Ensambla lo que sigue en la zona de código.	.set	Equivalente a .equ .		
.data	Ensambla lo que sigue en la zona de datos.	.space N	Reserva N bytes de memoria a 0.		
.end	No hay más instrucciones	símbolo .unreq símbolo	Cancela el alias símbolo.		
.equ símbolo, expr	Asigna el valor de expr a símbolo.	.word valor	Inicializa una palabra a valor.		

Conjunto de instrucciones ARM Thumb (2/2)

Operación		Ensamblador	Acción	Notas
Cargar	Con desp. inm., palabra	ldr Rd, [Rn, #Inm]	Rd ← [Rn + Inm]	Rango Inm: 0–124, múltiplos de 4.
	media palabra	ldrh Rd, [Rn, #Inm]	Rd ← ZeroExtend([Rn + Inm] _{15:0})	Bits 31:16 a 0. Rango Inm: 0–62, pares.
	byte	ldrb Rd, [Rn, #Inm]	Rd ← ZeroExtend([Rn + Inm] _{7:0})	Bits 31:8 a 0. Rango Inm: 0–31.
	Con desp. en registro, palabra	ldr Rd, [Rn, Rm]	Rd ← [Rn + Rm]	
	media palabra	ldrh Rd, [Rn, Rm]	Rd ← ZeroExtend([Rn + Rm] _{15:0})	Bits 31:16 a 0.
	media palabra con signo	ldrsh Rd, [Rn, Rm]	Rd ← SignExtend([Rn + Rm] _{15:0})	Bits 31:16 igual al bit 15.
	byte	ldrb Rd, [Rn, Rm]	Rd ← ZeroExtend([Rn + Rm] _{7:0})	Bits 31:8 a 0.
	byte con signo	ldrnb Rd, [Rn, Rm]	Rd ← SignExtend([Rn + Rm] _{7:0})	Bits 31:8 igual al bit 7.
Almacenar	Relativo al PC	ldr Rd, [PC, #Inm]	Rd ← [PC + Inm]	Rango Inm: 0–1020, múltiplos de 4.
	Relativo al SP	ldr Rd, [SP, #Inm]	Rd ← [SP + Inm]	Rango Inm: 0–1020, múltiplos de 4.
	Con desp. inm., palabra	str Rd, [Rn, #Inm]	[Rn + Inm] ← Rd	Rango Inm: 0–124, múltiplos de 4.
	media palabra	strh Rd, [Rn, #Inm]	[Rn + Inm] _{15:0} ← Rd _{15:0}	Rd _{31:16} se ignora. Rango Inm: 0–62, pares.
	byte	strb Rd, [Rn, #Inm]	[Rn + Inm] _{7:0} ← Rd _{7:0}	Rd _{31:8} se ignora. Rango Inm: 0–31.
	Con desp. en registro, palabra	str Rd, [Rn, Rm]	[Rn + Rm] ← Rd	
	media palabra	strh Rd, [Rn, Rm]	[Rn + Rm] _{15:0} ← Rd _{15:0}	Rd _{31:16} se ignora.
	byte	strb Rd, [Rn, Rm]	[Rn + Rm] _{7:0} ← Rd _{7:0}	Rd _{31:8} se ignora.
Apilar	Relativo al SP	str Rd, [SP, #Inm]	[SP + Inm] ← Rd	Rango Inm: 0–1020, múltiplos de 4.
Desapilar	Apilar	push <loreglist>	Apila registros en la pila	
	Apilar y enlazar	push <loreglist+LR>	Apila LR y registros en la pila	
Saltar	Desapilar	pop <loreglist>	Desapila registros de la pila	
	Desapilar y retorno	pop <loreglist+PC>	Desapila registros y salta a la dirección cargada en el PC	
Extender	Salto condicional	b {cond} <label>	Si {cond}, PC ← label (rango salto: –252 a +258 bytes de la instrucción actual).	
	Salto incondicional	b <label>	PC ← label (rango salto: ±2 KiB de la instrucción actual).	
	Salto largo y enlaza	bl <label>	LR ← dirección de la siguiente instrucción, PC ← label	
(Instrucción de 32 bits. Rango salto: ±4 MiB de la instrucción actual).				
Extender	Con signo, media a palabra	sxth Rd, Rm	Rd _{31:0} ← SignExtend(Rm _{15:0})	
	Con signo, byte a palabra	sxtb Rd, Rm	Rd _{31:0} ← SignExtend(Rm _{7:0})	
	Sin signo, media a palabra	uxth Rd, Rm	Rd _{31:0} ← ZeroExtend(Rm _{15:0})	
	Sin signo, byte a palabra	uxtb Rd, Rm	Rd _{31:0} ← ZeroExtend(Rm _{7:0})	

{cond}	EQ	Igual	Z	CS	Mayor o igual sin signo	C	LT	Menor que	Nv o nV	VS	No desbordamiento	V
	HI	Mayor sin signo	Cz	GE	Mayor o igual	NV o nv	PL	Positivo o cero	n	VC	Desbordamiento	v
	GT	Mayor que	z y (NV o nv)	MI	Negativo	N	LS	Menor o igual sin signo	C o Z			
	NE	Distinto	z	CC	Menor sin signo	c	LE	Menor o igual que	Nv o nV o Z			