Universidad de Vigo

Desarrollo de una 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

List of Figures

1 Introducción

Hoy en día la programación es cada vez una competencia más demandada, con la automatización de tareas siempre surge la necesidad de un software ya sea específico o no.

Debido a esto cada vez obtienen mas interés los estudios relacionados con cualquier tipo de desarrollo software (WEB, Android, etc). Entre los distintos lenguajes de programación están los conocidos como **de bajo nivel o ensamblador**. En estos el programador tiene pleno control sobre los registros de la cpu, habiendo uno específico para cada arquitectura y soportado solo por esta.

Por ello se usa comunmente simuladores para el desarrollo sin necesidad de tener el hardware o incluso en la docencia de este tipo de lenguajes. Permitiendo estos escribir y probar código para una arquitectura que no es la del equipo. Sin embargo, estos simuladores pueden depender de librerías de terceros o programas por lo que su instalación y uso puede no ser siempre tan rápida. Además no todos son intuitivos y/o configurables.

Como solución a todo esto, y para el caso concreto de ARM Thumb, se decide desarrollar un **Ensamblador y Simulador Web para ARM/THUMB**. Al tratarse de una plataforma web simplificamos 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.

Las tecnologías empleadas para esta plataforma son:

1. **React**: JavaScript Framework
2. **NodeJS**: JavaScript Engine
3. **TypeScript**: React - TypeScript
4. **React-Redux**
5. **Docker**

2 Objetivos

El objetivo principal de este trabajo es crear una plataforma web en la que los usuarios puedan escribir código en lenguaje ARM/THUMB, compilar 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.

Módulos de la aplicación web:

1. **Editor de código**: Editor de texto con soporte para ARM/Thumb con syntax highlighting.
2. **Visualizador de registros**: Panel con los registros de la CPU y su valor. Así como el estado de los flags.
3. **Visualizador de programa**: Lista de instrucciones ensamblador que el usuario ha cargado a la CPU.
4. **Visualizador de memoria**: Panel con la memoria de la CPU.
5. **ARM Help**: Mínima ayuda sobre las instrucciones ARM disponibles en el simulador.

Como objetivo adicional se busca que la aplicación se lo mas intuitiva posible y ofrezca control absoluto sobre la CPU y la memoria que se está simulando. Con facilidades para modificar los valores de los registros o de la memoria.

3 Resultados

Plataforma web integrando un ensamblador y simulador para ARM Thumb.

3.1 ARM Thumb Emulator

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 también se realizaron pruebas unitarias de cada instrucción implementada para asegurar que el comportamiento del simulador es el esperado si se ejecutase en una CPU de ARM real.

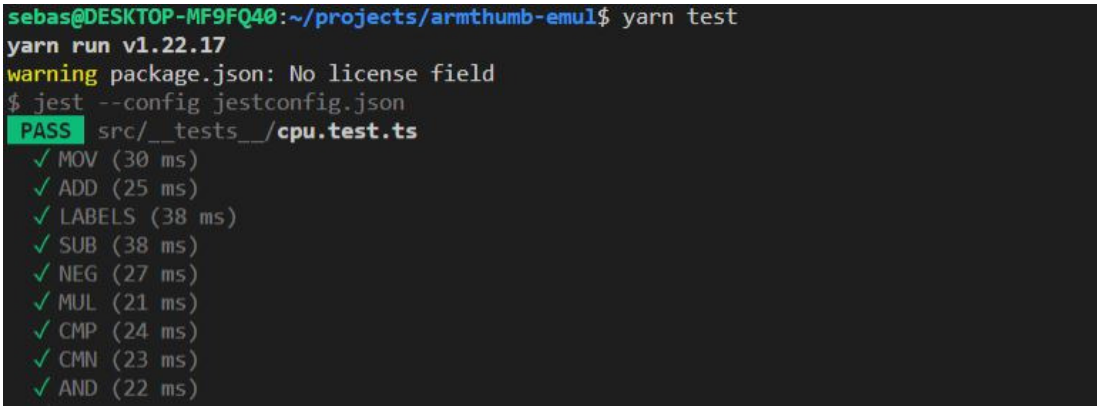

Figure 1: Running Test Suite

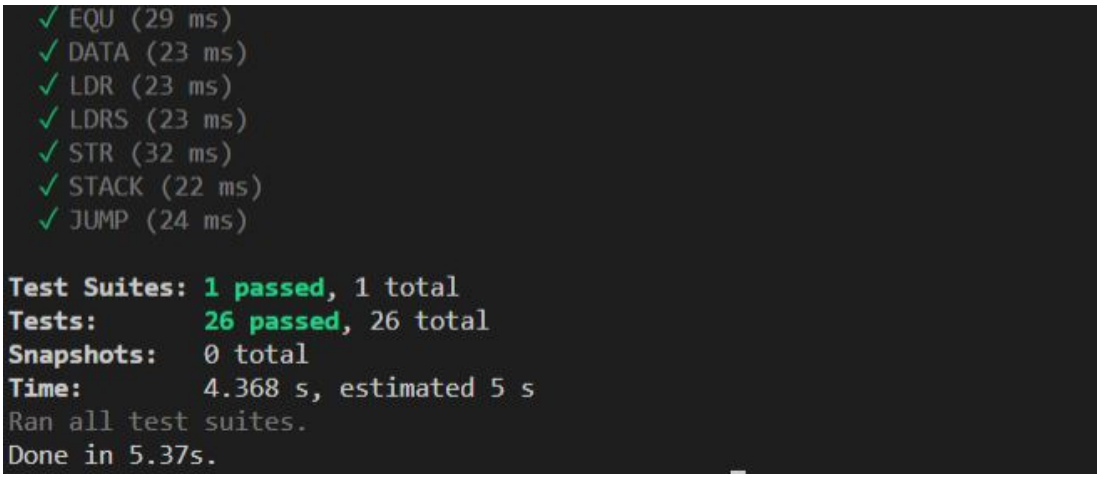

Figure 2: Finished Test Suite

Para cada test hay un código de ARM/Thumb cuyo resultado es conocido y siempre el mismo. De este modo podemos asegurar que todo sigue funcionando como debe tan solo con ejecutar las pruebas unitarias.

El estado de la CPU tras la ejecución del código es volcado a un archivo temporal (add.json.tmp) de no existir. 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 de la CPU.

Cada test simboliza una instrucción en ARM Thumb, salvo en algunos casos donde se prueban variantes de una sola en un mismo test (ldr, ldrb, ldrh). Cada test tendrá su correspondiente archivo con el estado esperado de la cpu con el que se comprobará que la ejecución sea correcta.

src > __tests__ > 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 3: ARM Test Code

src > __tests__ > asm > {} mov.S.json > ...


```
{
  "regs": {
    "r0": 0,
    "r1": 4,
    "r2": 4,
    "r3": 1,
    "r4": 0,
    "r5": 4,
    "r6": 0,
    "r7": 0,
    "r8": 4,
    "r9": 4,
    "r10": 0,
    "r11": 0,
    "r12": 0,
    "r13": 4,
    "r14": 0,
    "r15": 16
  },
  "z": false,
  "n": false,
  "c": false,
  "v": false,
  "memory": [],
  "memSize": 0,
  "stackSize": 0,
  "program": [
```


Figure 4: CPU Expected State

La CPU emulada es básicamente un objeto de javascript con los registros, flags, memoria y todas las funciones necesarias para compilar y ejecutar el código.

Para inicializar la cpu basta con llamar a la función defaultCPU() que exporta el módulo. Esta función devuelve un objeto cpu con los valores por defecto, se puede modificar el tamaño de la memoria y del stack por parámetros.

```typescript
type armCPU_T = {
  regs: { [key: string]: number };
  z: boolean;
  n: boolean;
  c: boolean;
  v: boolean;
  memory: number[];
  memSize: number;
  stackSize: number;
  program: Instruction[];
  error?: CompilerError;

  // Methods
  run: () => void;
  step: () => void;
  reset: () => void;
  load: (program: Program) => void;
  loadAssembly: (assembly: string) => void;
  execute: (ins: Instruction) => void;
  setFlag: (flag: Flags, value: boolean) => void;
};
```

Figure 5: ARM CPU Type

3.2 Web Interface

Como interfaz de usuario se desarrolla una plataforma web que integra el emulador. Desarrollada en TypeScript con el framework React. Gracias a la librería React-Redux se comparte un estado global a todos los componentes de la interfaz. Este estado consta de:

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, 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.

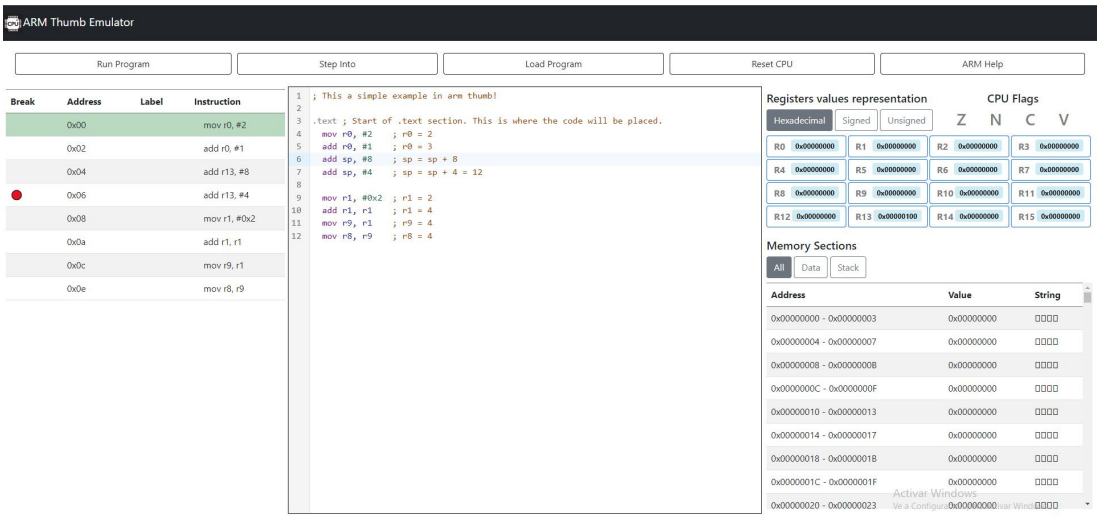

Figure 6: User Interface

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 7: 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.

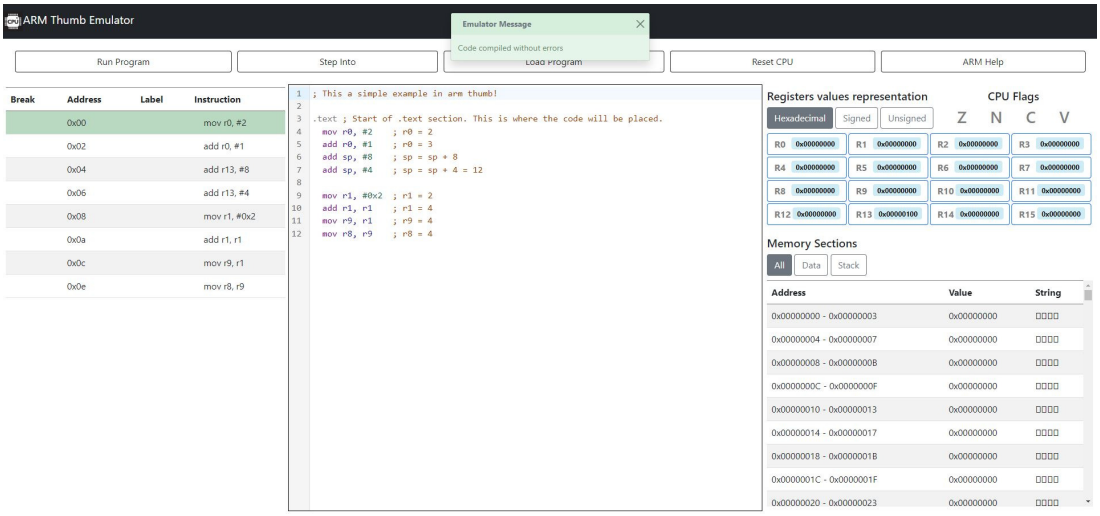

Figure 8: Compiled Code

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

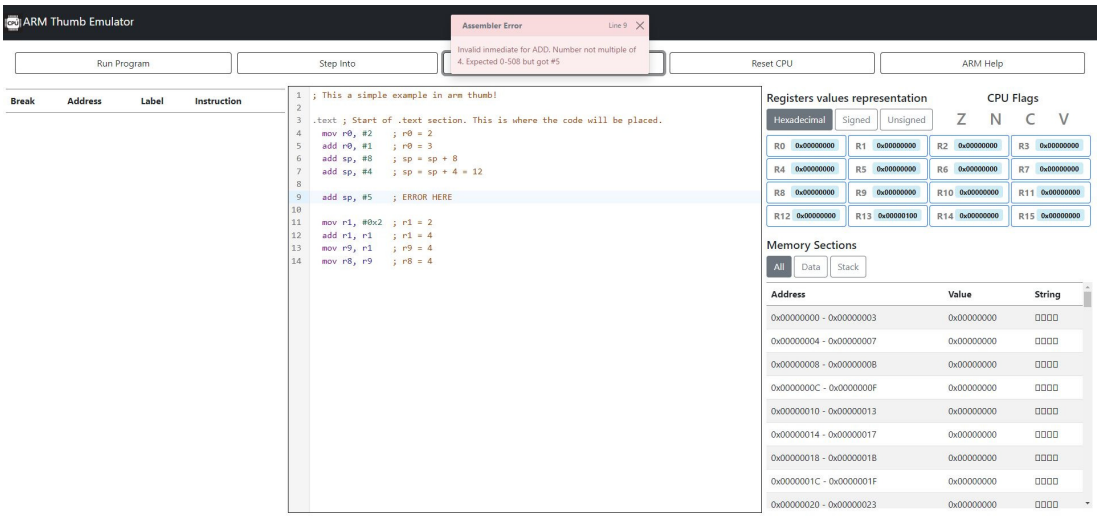

Figure 9: Error Compiling Code

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 en la CPU.

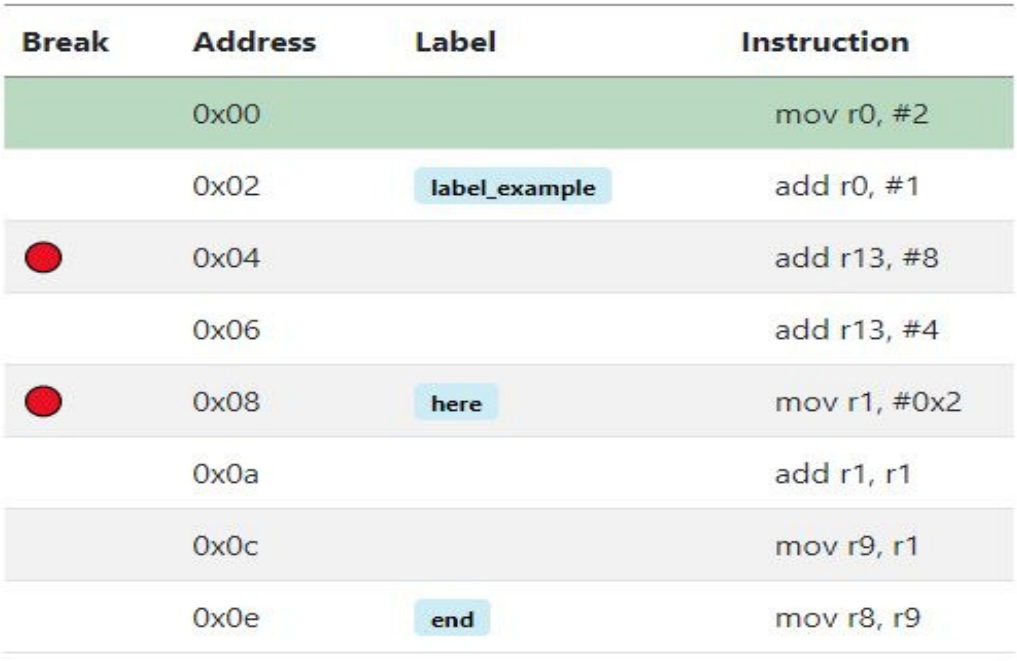

Figure 10: Loaded Program

Además se pueden poner/quitar breakpoints con un simple click. Del mismo modo para ejecutar el programa tras su carga tenemos dos botones:

1. **Run Code**: Ejecuta todo el código cargado en la CPU.
2. **Step Into**: Ejecuta instrucción a instrucción el programa cargado.

3.2.3 Registros y flags

En ARM Thumb tenemos acceso a 16 registros, la mitad conocidos como **Low Registers** y la mitad superior **High Registers**. Estos registros están siempre representados en pantalla junto con los flags de la CPU.
Los registros inferiores son los del 0 al 7, mientras que los superiores son del 8 al 15. Algunos de los registros superiores son registros con un propósito específico. Estos son:

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

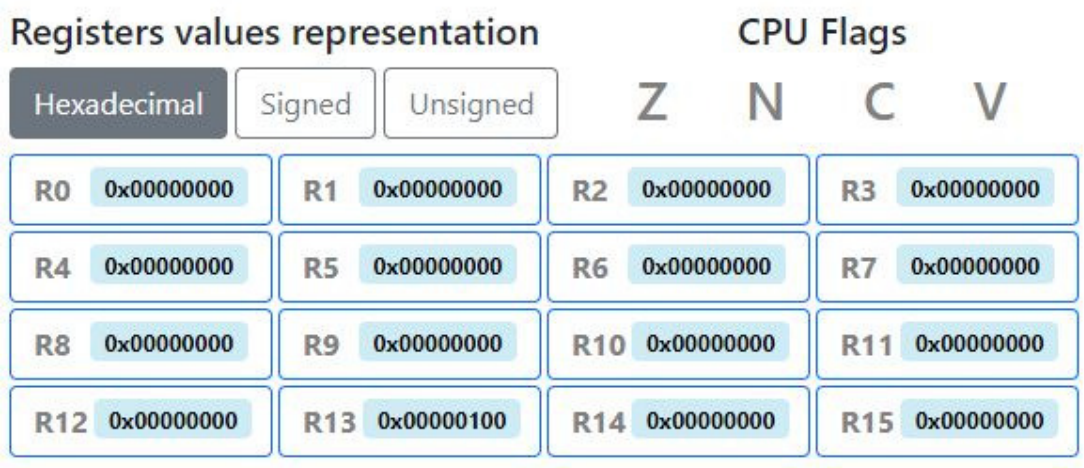

Figure 11: Registers Menu

Como se puede ver en la figura 11 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.

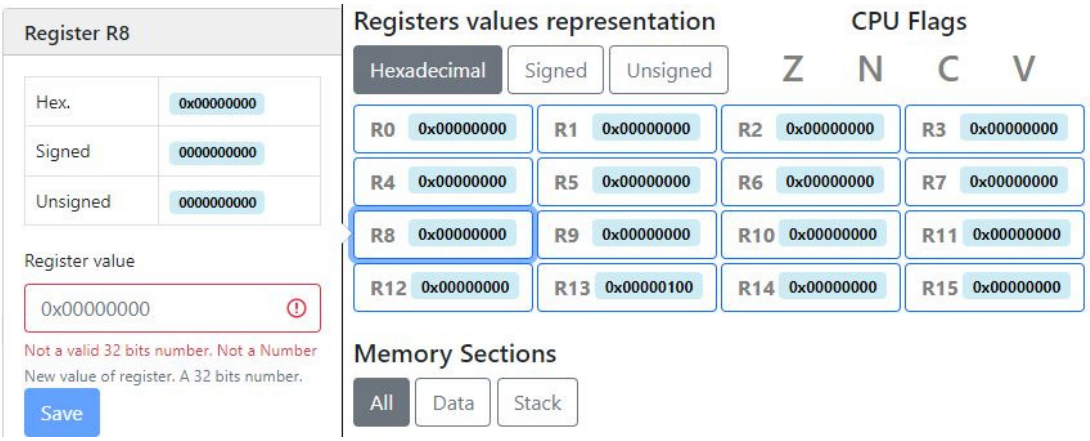

Figure 12: Change Register Value

Además se comprueba en todo momento que la entrada del usuario sea un nº de 8 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.

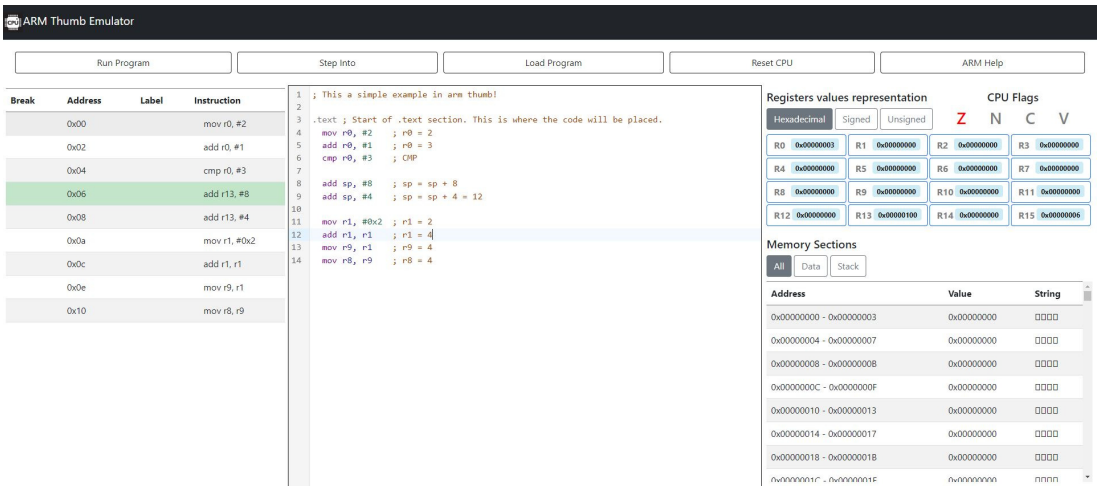

Figure 13: Flags Example

En este ejemplo se compara r0 con un 3. Al ser la comparación una resta y dar esta cero se activa el flag correspondiente Z. Los distintos flags son:

1. **Z**: Resultado igual a 0
2. **N**: Resultado negativo
3. **C**: Bit de acarreo
4. **V**: Overflow con signo

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.

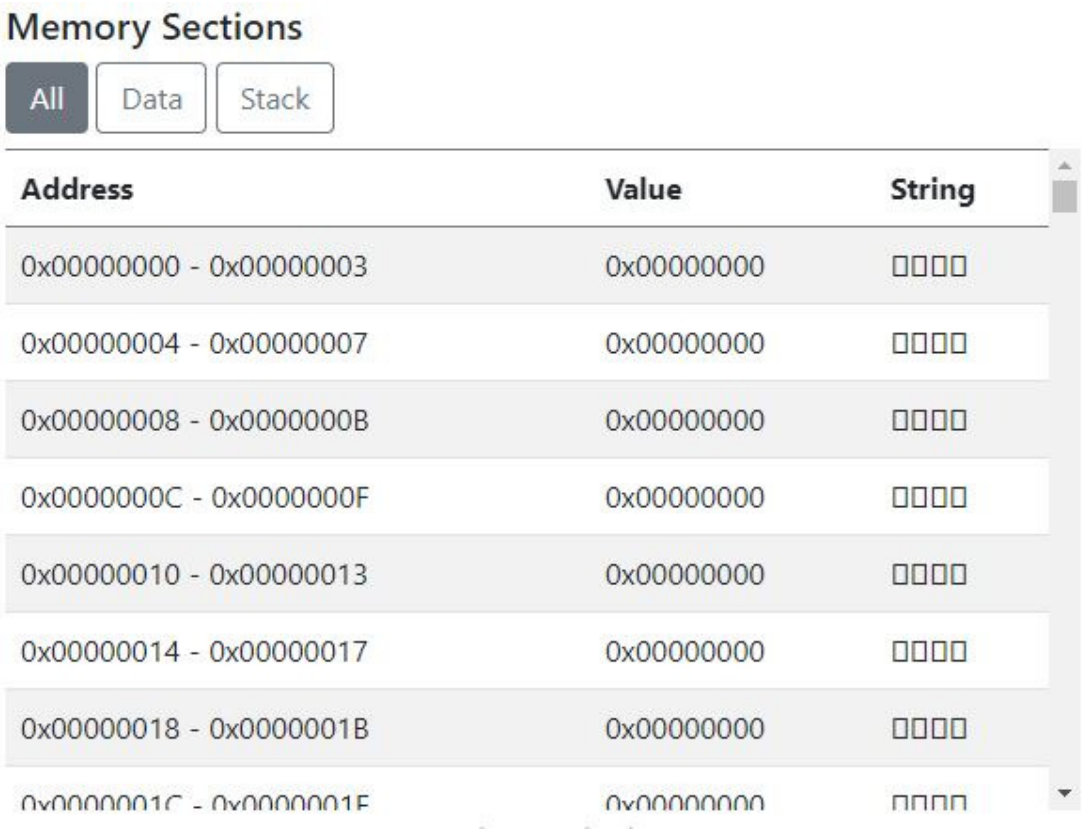

Figure 14: Memory Menu

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.

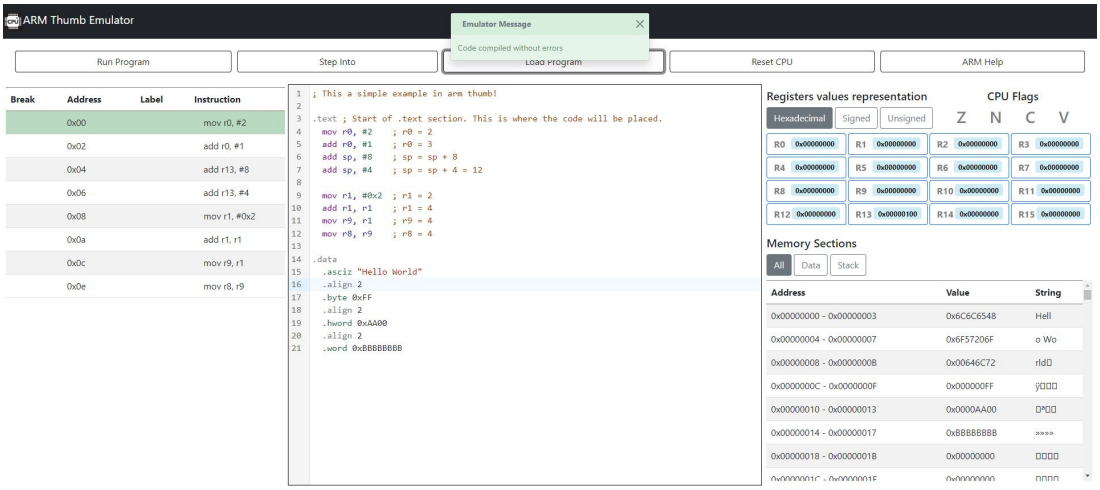

Figure 15: Memory Data Example

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 clickar en la zona de memoria cuyo valor deseamos modificar y se nos abrirá un menú similar al visto para modificar un registro.

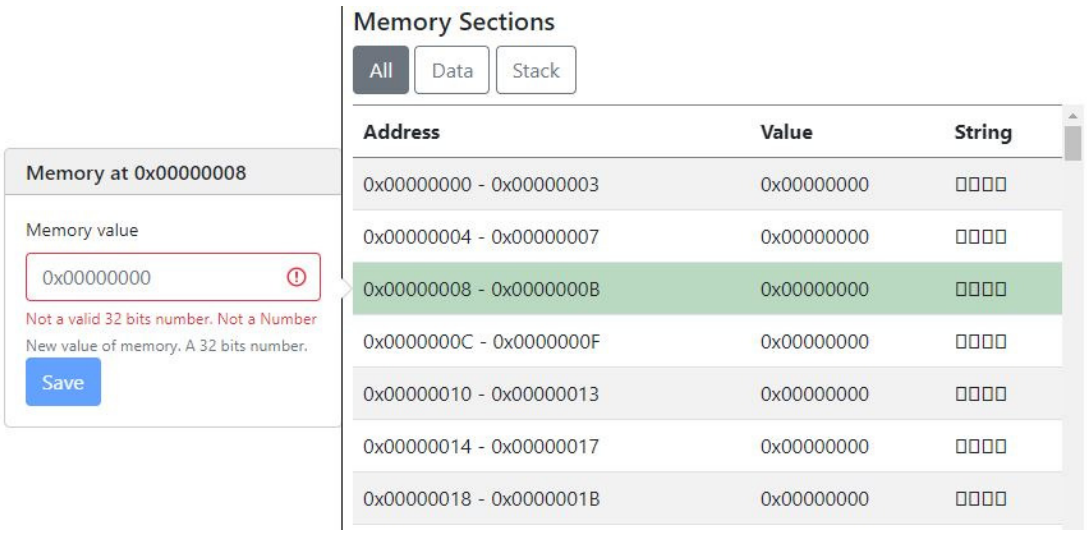

Figure 16: Change Memory Value

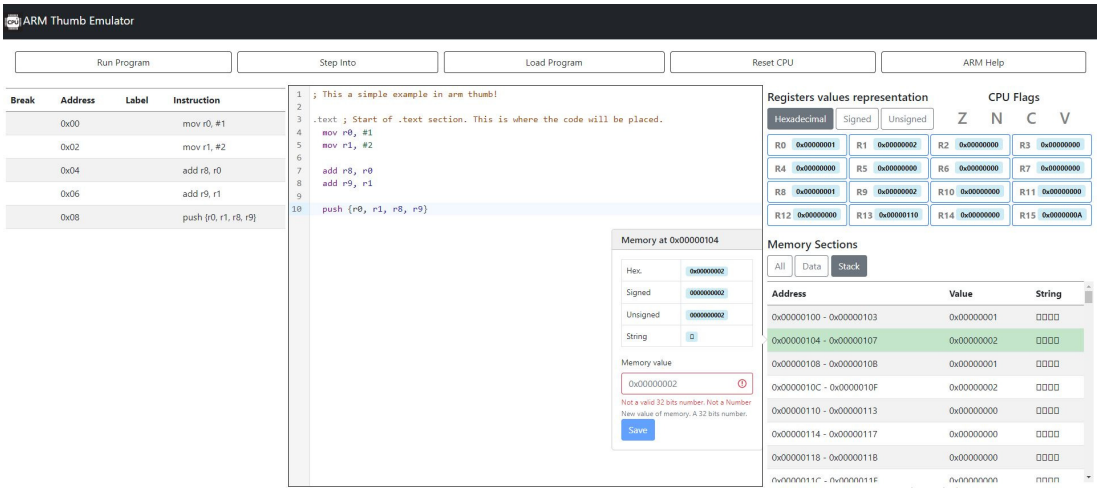

Figure 17: Stack Example

4 Conclusiones

5 Bibliografía