



UNIVERSIDAD  
DE MÁLAGA



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA  
GRADO EN INGENIERÍA DE SOFTWARE

**Desarrollo de un simulador de WebAssembly con  
aplicación a la docencia de Arquitectura de  
Computadores**

**Development of WebAssembly simulator with application  
to Computer Architecture learning**

Realizado por  
**Daniel Castillo Sánchez**

Tutorizado por  
**Eladio Gutiérrez Carrasco**

Departamento  
**Arquitectura de computadores**

UNIVERSIDAD DE MÁLAGA  
MÁLAGA, SEPTIEMBRE DE 2020



# Resumen

El principal propósito de este Trabajo de Fin de Grado es lograr entender el funcionamiento interno de WebAssembly y lograr verlo de forma visual mediante una aplicación web que ilustre su funcionamiento.

WebAssembly es un formato binario de instrucciones que representa un paso intermedio entre un lenguaje de programación y código nativo y cuyo propósito es ser lo más portátil posible, es decir, que se pueda ejecutar en el mayor número de dispositivos posibles sin perder demasiado rendimiento.

Todos los navegadores actuales implementan de una forma o de otra una máquina virtual que permite leer y traducir ficheros fuentes de WebAssembly y traducirlos a código máquina para permitir la ejecución de código eficiente. Este proceso permite que se ejecute ciertas partes de una aplicación web con una eficiencia cercana a las aplicaciones nativas, cosa que con JavaScript no se consigue normalmente.

Para demostrar las capacidades de WebAssembly se ha desarrollado un simulador o interprete simple en el lenguaje de programación Rust, que se compilará al formato binario de WebAssembly y se utilizará sus funciones dentro de un navegador web. Este simulador se comunicará con la interfaz web, que permitirá controlarlo y mostrar información relevante de su funcionamiento.

**Palabras clave:** Interprete, WebAssembly, Rust, Web, Ensamblador.



# Abstract

The main intention of this Final Degree Project is to understand the internal operations of WebAssembly and to create a web application which will show it visually.

WebAssembly is a binary instruction format which represents the middle point between a programming language and native code. It's designed to be the most portable possible, which means that it will run in a wide variety of devices without losing performance.

All of the modern web browsers implement in some a virtual machine which allows to read and translate WebAssembly code files to machine code to support the execution of code efficiently. This process allows us to run some parts of a web application much more efficient than Javascript.

A WebAssembly simulator has been developed in the programming language Rust to prove the WebAssembly capabilities, which will compile into the WebAssembly binary format and it will use its functions inside the virtual machine in the web browser. This simulator will communicate with the web interface, allowing to manage it and to show relevant information of its internals.

**Keywords:** Simulator, Interpreter, WebAssembly, Web, Assembly



# Índice

<b>Resumen</b> .....	<b>1</b>
<b>Abstract</b> .....	<b>1</b>
<b>Índice</b> .....	<b>1</b>
<b>Tabla de Ilustraciones</b> .....	<b>3</b>
<b>Introducción</b> .....	<b>1</b>
<b>1.1</b> <b>Ámbito</b> .....	<b>1</b>
<b>1.2</b> <b>Motivación</b> .....	<b>2</b>
<b>1.3</b> <b>Tecnologías y herramientas utilizadas</b> .....	<b>2</b>
<b>1.4</b> <b>Estructura de la memoria</b> .....	<b>3</b>
<b>WebAssembly</b> .....	<b>5</b>
<b>2.1</b> <b>¿Qué es WebAssembly?</b> .....	<b>5</b>
2.1.1 Casos de uso .....	5
2.1.2 Flujo de funcionamiento en el navegador .....	6
2.1.3 Figma: Un caso real de uso de WebAssembly .....	7
<b>2.2</b> <b>Elementos de WebAssembly</b> .....	<b>8</b>
2.2.1 Tipos de datos .....	8
2.2.2 Estructura de un módulo .....	10
2.2.3 Estructura de ejecución .....	14
2.2.4 Instrucciones .....	18
2.2.5 Control de flujo .....	24
<b>2.3</b> <b>Otras características importantes</b> .....	<b>27</b>
2.3.1 Formatos de fichero disponibles .....	27
2.3.2 Validación de un módulo .....	28
<b>Rust</b> .....	<b>31</b>
<b>3.1</b> <b>¿Qué es Rust?</b> .....	<b>31</b>
<b>3.2</b> <b>¿Por qué Rust?</b> .....	<b>31</b>
<b>3.3</b> <b>Relación con WebAssembly</b> .....	<b>34</b>
<b>Estructura del proyecto</b> .....	<b>37</b>
<b>4.1</b> <b>Consideraciones iniciales</b> .....	<b>37</b>
4.1.1 Elección de librerías .....	37
4.1.2 Planificación inicial .....	38
4.1.3 Organización de las distintas partes del proyecto .....	39
<b>4.2</b> <b>Estructura final del proyecto</b> .....	<b>41</b>
4.2.1 Planificación final .....	41
4.2.2 Modo de conexión de las partes del proyecto .....	42
<b>4.3</b> <b>Fases de desarrollo</b> .....	<b>46</b>
4.3.1 Estructuras de un módulo .....	46
4.3.2 Conversión de un fichero de WebAssembly .....	48

4.3.3 Estructuras de WebAssembly en tiempo de ejecución .....	50
4.3.4 Implementación del simulador.....	50
4.3.5 Conexión entre las distintas partes del proyecto.....	54
<b>4.4 Limitaciones del simulador.....</b>	<b>54</b>
<b>Ejecución de pruebas .....</b>	<b>57</b>
5.1 Modo de ejecución de pruebas .....	57
5.2 Alcance de las pruebas.....	58
5.3 Implementación de las pruebas.....	59
<b>Conclusiones y líneas futuras .....</b>	<b>61</b>
6.1 Conclusiones .....	61
6.2 Líneas futuras .....	62
<b>Referencias.....</b>	<b>65</b>
<b>Manual de instalación.....</b>	<b>67</b>
1. Instalación de las herramientas.....	67
2. Preparación de las carpetas del proyecto .....	68
3. Compilación y ejecución .....	69
4. Ejecución de pruebas .....	69
<b>Manual de Usuario de la vista web.....</b>	<b>71</b>
<b>Requisitos del proyecto .....</b>	<b>75</b>



# Tabla de Ilustraciones

Ilustración 1	Tabla de compatibilidades, de: <a href="https://developer.mozilla.org/en-US/docs/WebAssembly">https://developer.mozilla.org/en-US/docs/WebAssembly</a>	5
Ilustración 2	Comparación de tiempo entre las dos funciones, de <a href="https://developers.google.com/web/updates/2018/04/loading-wasm">https://developers.google.com/web/updates/2018/04/loading-wasm</a>	6
Ilustración 3	Comparación de tiempos de carga entre asm.js y WebAssembly, de <a href="https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/">https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/</a>	7
Ilustración 4	Comparación de tamaños de ficheros generados por asm.js y WebAssembly, de <a href="https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/">https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/</a>	7
Ilustración 5	Elementos de un módulo, de <a href="https://webassembly.github.io/spec/core/syntax/modules.html">https://webassembly.github.io/spec/core/syntax/modules.html</a>	10
Ilustración 6	Estructura de un Func, de <a href="https://webassembly.github.io/spec/core/syntax/modules.html">https://webassembly.github.io/spec/core/syntax/modules.html</a>	11
Ilustración 7	Estructura de un Element Segment, de <a href="https://webassembly.github.io/spec/core/syntax/modules.html">https://webassembly.github.io/spec/core/syntax/modules.html</a>	11
Ilustración 8	Estructura de un Data Segment, de <a href="https://webassembly.github.io/spec/core/syntax/modules.html">https://webassembly.github.io/spec/core/syntax/modules.html</a>	12
Ilustración 9	Estructura de un global, de <a href="https://webassembly.github.io/spec/core/syntax/modules.html">https://webassembly.github.io/spec/core/syntax/modules.html</a>	12
Ilustración 10	Estructura de un tipo Export, de <a href="https://webassembly.github.io/spec/core/syntax/modules.html">https://webassembly.github.io/spec/core/syntax/modules.html</a>	13
Ilustración 11	Estructura de un Import, de <a href="https://webassembly.github.io/spec/core/syntax/modules.html">https://webassembly.github.io/spec/core/syntax/modules.html</a>	13
Ilustración 12	Estructura del tipo Store, de <a href="https://webassembly.github.io/spec/core/exec/runtime.html">https://webassembly.github.io/spec/core/exec/runtime.html</a>	14
Ilustración 13	Formato de los tipos Addresses, de <a href="https://webassembly.github.io/spec/core/exec/runtime.html">https://webassembly.github.io/spec/core/exec/runtime.html</a>	14
Ilustración 14	Estructura de un Module Instance, de <a href="https://webassembly.github.io/spec/core/exec/runtime.html">https://webassembly.github.io/spec/core/exec/runtime.html</a>	15
Ilustración 15	Estructura de un Function Instance, de <a href="https://webassembly.github.io/spec/core/exec/runtime.html">https://webassembly.github.io/spec/core/exec/runtime.html</a>	15
Ilustración 16	Estructura de un Frame, de <a href="https://webassembly.github.io/spec/core/exec/runtime.html">https://webassembly.github.io/spec/core/exec/runtime.html</a>	16
Ilustración 17	Estructura de un Table Instance, de <a href="https://webassembly.github.io/spec/core/exec/runtime.html">https://webassembly.github.io/spec/core/exec/runtime.html</a>	17
Ilustración 18	Estructura de un Memory Instance, de <a href="https://webassembly.github.io/spec/core/exec/runtime.html">https://webassembly.github.io/spec/core/exec/runtime.html</a>	17
Ilustración 19	Estructura de un Global Instance, de <a href="https://webassembly.github.io/spec/core/exec/runtime.html">https://webassembly.github.io/spec/core/exec/runtime.html</a>	17

Ilustración 20 Estructura de un Export Instance, de <a href="https://webassembly.github.io/spec/core/exec/runtime.html">https://webassembly.github.io/spec/core/exec/runtime.html</a>	18
Ilustración 21 Estructura de las instrucciones numéricas, de <a href="https://webassembly.github.io/spec/core/syntax/instructions.html">https://webassembly.github.io/spec/core/syntax/instructions.html</a>	19
Ilustración 22 Estructura de las instrucciones numéricas, de <a href="https://webassembly.github.io/spec/core/syntax/instructions.html">https://webassembly.github.io/spec/core/syntax/instructions.html</a>	20
Ilustración 23 Diagrama de funcionamiento de la instrucción Select	20
Ilustración 24 Estructura de las instrucciones de variables	21
Ilustración 25 Estructura de instrucciones de manejo de memoria	22
Ilustración 26 Estructura de las instrucciones de control de flujo, de <a href="https://webassembly.github.io/spec/core/syntax/instructions.html#control-instructions">https://webassembly.github.io/spec/core/syntax/instructions.html#control-instructions</a>	23
Ilustración 27 Diferencia entre los distintos bloques	24
Ilustración 28 Diferentes ejecuciones de la instrucción BR	25
Ilustración 29 Demostración de un bloque con Function Type	26
Ilustración 30 Ejemplo del formato textual, de <a href="https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format">https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format</a>	28
Ilustración 31 Diferencia entre especificación y código escrito	32
Ilustración 32 Fragmento de código que no compilará, de <a href="https://doc.rust-lang.org/stable/book/ch04-02-references-and-borrowing.html">https://doc.rust-lang.org/stable/book/ch04-02-references-and-borrowing.html</a>	33
Ilustración 33 Implementación del operador Local.Get	33
Ilustración 34 Implementación de las instrucciones de Reinterpret	34
Ilustración 35 Llamada de una función de WebAssembly en Javascript directamente desde el fichero .wasm, de <a href="https://developer.mozilla.org/en-US/docs/WebAssembly&gt;Loading_and_running">https://developer.mozilla.org/en-US/docs/WebAssembly&gt;Loading_and_running</a>	35
Ilustración 36 Ejemplo de uso de Wasm-bindgen, de <a href="https://github.com/rustwasm/wasm-bindgen">https://github.com/rustwasm/wasm-bindgen</a>	35
Ilustración 37 Tablero de Trello con algunas tareas.	39
Ilustración 38 Diagrama de comunicación directa entre las distintas partes	40
Ilustración 39 Diagrama de comunicación indirecta entre las distintas partes	40
Ilustración 40 Ejemplo de tarea con la nueva planificación	42
Ilustración 41 Comparación de los diferentes frameworks existentes para el desarrollo de una vista web, de <a href="https://github.com/flosse/rust-web-framework-comparison#frontend-frameworks">https://github.com/flosse/rust-web-framework-comparison#frontend-frameworks</a>	43
Ilustración 42 Diagrama de la arquitectura de Elm, de <a href="https://www.tutorialspoint.com/elm/elm_architecture.htm">https://www.tutorialspoint.com/elm/elm_architecture.htm</a>	44
Ilustración 43 Algunos mensajes utilizados en el desarrollo de la vista web	45
Ilustración 44 Ejemplo de una parte de la vista web	45
Ilustración 45 Comparación de estructuras de WebAssembly entre implementación y especificación	46
Ilustración 46 Parte de la estructura con las operaciones que muestra las encargadas del control de flujo	47
Ilustración 47 Ejemplo de implementación de instrucciones numéricas	47
Ilustración 48 Implementación de las operaciones unarias de punto flotante en el simulador	48
Ilustración 49 Ejemplo de transformación del tipo <i>Export</i>	49

Ilustración 50 Fragmento de código donde se muestran la conversión de las instrucciones del tipo I32	49
Ilustración 51 Comparación de la instancia de un módulo entre especificación e implementación	50
Ilustración 52 Estructura antigua del código del simulador	51
Ilustración 53 Estructura final del código del simulador	51
Ilustración 54 Diagrama de funcionamiento de la estructura ExecutionUnit	52
Ilustración 55 Implementación de la estructura StackUnwindVariant	53
Ilustración 56 Ejemplo de estado de pila durante la ejecución de una instrucción de salto en un bloque de tipo Block	53
Ilustración 57 Distintos asertos disponibles en el formato wast, de <a href="https://github.com/WebAssembly/spec/blob/master/interpreter/README.md#scripts">https://github.com/WebAssembly/spec/blob/master/interpreter/README.md#scripts</a>	58
Ilustración 58 Extracto del fichero de pruebas memory_grow.wast, de <a href="https://github.com/WebAssembly/spec/blob/master/test/core/memory_grow.wast">https://github.com/WebAssembly/spec/blob/master/test/core/memory_grow.wast</a>	59
Ilustración 59 Salida correcta del comando "rustup -V"	68
Ilustración 60 Ejemplo de ejecución de las pruebas	70
Ilustración 61 Vista inicial de la interfaz web	71
Ilustración 62 Vista de elección de función	72
Ilustración 63 Vista de control del simulador	72



# 1

## Introducción

### 1.1 Ámbito

En la última década ha habido una revolución en lo que a la web se refiere, permitiendo cada vez páginas y aplicaciones web más complejas. Ha llegado al punto de que ya hay aplicaciones web que permiten realizar funciones de software desarrollado para escritorio, sin el hastío de tener que instalarlo. El único factor limitante es el rendimiento ya que, aunque los motores de JavaScript de los navegadores actuales son muy avanzados y realizan muchas optimizaciones a nivel interno, hay ciertas situaciones en las que esto no es suficiente.

Mirando la historia de los navegadores se han ido desarrollando ideas que permitieran ejecutar código en el navegador web para suplir esta deficiencia. A continuación veremos algunos de estas ideas:

- **Java Applet:** Fue la respuesta de Oracle a intentar solucionar este problema, permitiendo ejecutar código Java directamente en el navegador web usando la máquina virtual de Java del propio sistema operativo del usuario. Actualmente está en desuso por graves problemas de seguridad [1].
- **Asm.js:** Desarrollado por un ingeniero de Mozilla, es un subconjunto estricto de JavaScript que permite a los motores de JavaScript de los navegadores web ejercer una optimización más agresiva consiguiendo un mayor rendimiento. La mayoría de los navegadores lo pueden utilizar, pero está siendo sustituido por WebAssembly.
- **Google Native Client (NaCl):** Es la respuesta de Google para este problema. Es una tecnología que permite ejecutar código nativo dentro del navegador desde una página web más allá de JavaScript. Comparte varias ideas con WebAssembly, pero el factor limitante es que es exclusivo de Google Chrome, el navegador web oficial de Google.

- **WebAssembly:** Es la respuesta de la unión de varias personas vinculadas a Mozilla, Microsoft, Google y Apple en un intento de estandarizar las ideas que se han ido presentando anteriormente. Su desarrollo ha sido y sigue siendo influenciado por asm.js y NaCl. Aunque está en serio desarrollo todavía, la World Wide Web Consortium (W3C) lo ha aceptado como el cuarto lenguaje que se permite ejecutar en el navegador, siendo los primeros HTML, CSS y JavaScript.

Esta lista no está completa ni mucho menos, hay otras tecnologías como Flash o Emscripten<sup>1</sup> que se han quedado fuera y la información proporcionada es escueta, pero permite tener una visión general de cómo se ha llegado a WebAssembly.

## 1.2 Motivación

La principal motivación a la hora de realizar este Trabajo de Fin de Grado es relacionar dos temas que para mí como persona me interesan bastante, que son el lenguaje de programación Rust y la tecnología de WebAssembly.

La explicación de la elección de Rust como lenguaje de programación para este proyecto se debe a que debido a su fuerte sistema de tipos ha permitido desarrollar este proyecto con relativa facilidad, sin necesidad de preocuparse de algunos problemas típicos que solo se encontrarían en tiempo de ejecución.

Sobre WebAssembly, llama bastante la atención la capacidad de poder ejecutar código nativo en un navegador web que tuviera casi la misma eficiencia que un programa nativo de sistema operativo.

Finalmente, para la principal temática del proyecto, que es un simulador de WebAssembly, la decisión viene del deseo de conocer cómo funciona esta tecnología por dentro y a su vez poder proporcionar una manera visual de ver su funcionamiento. Además, como es posible compilar código de Rust a WebAssembly, parece la mejor oportunidad para demostrar lo interesante que puede llegar a ser para el futuro próximo. Otra ventaja intrínseca de esto es que se permite distribuir el simulador de forma muy eficiente, ya que solo haría falta conexión a internet y un navegador compatible con esta tecnología.

## 1.3 Tecnologías y herramientas utilizadas

A lo largo del proyecto se han utilizado distintas herramientas y equipo que han ayudado de un modo u otro al desarrollo del proyecto. Entre ellas, podemos mencionar las siguientes:

- Lenguajes utilizados:
  - **Rust:** Principal lenguaje en el que está implementado la gran mayoría del proyecto, se describirá con más detalles en capítulos posteriores.
  - **HTML y CSS:** Los lenguajes *de facto* para el desarrollo web, han permitido implementar la parte web del proyecto.

---

<sup>1</sup> <https://emscripten.org/>

- Servicios web:
  - **Bitbucket**<sup>2</sup>: Se ha usado como servidor externo para mantener la gestión de código usando el sistema de control de versiones Git.
  - Trello: Servicio web que permite un control de tareas al estilo Kanban, se ha utilizado para gestionar todas las tareas del proyecto.
  - **Netlify**<sup>3</sup>: Servicio externo que ofrece hosting para distintos proyectos. Con él se ha podido subir el proyecto a la web para que pueda ser probado por cualquier persona.
- Utilidades de compilación:
  - **Rustup**<sup>4</sup>: Es un sistema de control para las distintas versiones de Rust, permite descargar e instalar diferentes compiladores para distintas plataformas.
  - Cargo: Gestor de paquetes por defecto de Rust, ha permitido crear y enlazar los distintos proyectos, así como instalar dependencias con extrema facilidad.
  - **Wasm-pack**<sup>5</sup>: Conjunto de herramientas que facilitan enormemente el desarrollo de aplicaciones de WebAssembly en Rust.
- Librerías:
  - **Wasm-bindgen**: Al compilar de Rust a WebAssembly, permite tener una interfaz común entre funciones de Rust y Javascript por medio de WebAssembly.
  - **Wasm-parser**: Librería en Rust que permite serializar ficheros binarios de WebAssembly a estructuras de datos adecuados para su posterior uso.
  - **Wat**: Librería que permite serializar ficheros textuales de WebAssembly.
  - **Wast**: Librería que permite serializar ficheros textuales que incluyen ciertas instrucciones usadas para la ejecución de pruebas.

## 1.4 Estructura de la memoria

El documento se va a organizar en varios capítulos que explicará de manera progresiva desde los aspectos teóricos de WebAssembly hasta la organización, implementación y pruebas del proyecto.

A continuación vamos a explicar de manera breve los temas que se van a tratar en cada capítulo:

---

<sup>2</sup> <https://bitbucket.org/>

<sup>3</sup> <https://www.netlify.com/>

<sup>4</sup> <https://rustup.rs/>

<sup>5</sup> <https://github.com/rustwasm/wasm-pack>

- **2. WebAssembly:** Se explicará que es WebAssembly, que implicaciones puede tener su uso en el desarrollo de una aplicación web y los elementos que conforman la estructura de un entorno de WebAssembly.
- **3. Rust:** En este capítulo se hablará brevemente de Rust, por qué se ha elegido como lenguaje de programación para este proyecto y su íntima relación con WebAssembly.
- **4. Estructura del proyecto:** Durante este capítulo se hablará en detalle de todo el proceso que ha llevado a la implementación del proyecto. Esto incluye las asunciones iniciales del proyecto, los problemas que han ocurrido durante todo el proyecto y algunos detalles importantes de la implementación del mismo.
- **5. Ejecución de pruebas:** En este capítulo se va a entrar en algunos detalles sobre el proceso de ejecución de pruebas para comprobar el correcto funcionamiento del simulador, así como detalles de la infraestructura de pruebas.
- **6. Conclusiones y líneas futuras:** Para terminar, en este capítulo se hablará de las posibles ampliaciones o proyectos derivados que podrían surgir de este, así como hablar del posible futuro que tiene WebAssembly en el desarrollo de aplicaciones web.



# 2

## WebAssembly

### 2.1 ¿Qué es WebAssembly?

WebAssembly es un conjunto de instrucciones binarias para una máquina virtual basada en pila [2]. Está diseñado para poder ejecutado en el mayor número de entornos posibles sin necesidad de compilar varias veces.

A fecha del año 2020, se puede observar en la Ilustración 1 que **la mayoría de navegadores web tienen soporte para esta tecnología**. Además, Node.js también está incluido en esta lista, por lo que es posible usarlo tanto en navegadores web como en algunas implementaciones de servidores.

	🖥️						📱					📄	
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js
WebAssembly	57	16	52 *	No	44	11	57	57	52 *	43	11	7.0	8.0.0

Ilustración 1 Tabla de compatibilidades, de: <https://developer.mozilla.org/en-US/docs/WebAssembly>

#### 2.1.1 Casos de uso

Una de las principales fortalezas es la **variedad de casos de usos** y de posibles desarrollos que puede llevar a la web, habiendo cosas que antes eran impensables de realizar dentro de un entorno tan limitado como lo es un navegador web. A continuación se lista unos posibles casos de usos [3]:

- Edición de imágenes/videos.

- Videojuegos con bastante carga de CPU.
- Reconocimiento de imágenes
- Visualización y simulación de aplicaciones científicas
- Aplicaciones basadas en Realidad Virtual y Realidad Aumentada

Estos casos de uso entre otros, eran aplicaciones que antes eran posibles solo utilizando herramientas muy específicas, situación que ha cambiado y que cambiará en el futuro con un adopción más generalizada de esta tecnología.

### 2.1.2 Flujo de funcionamiento en el navegador

Aunque WebAssembly se ejecuta en un entorno aparte al de Javascript en un navegador, aún es necesario usar funciones de la API de Javascript para poder ejecutar el módulo correspondiente.

Además, WebAssembly representa un lenguaje intermedio para el navegador. Esto es así ya que con las funciones anteriormente mencionadas los navegadores leen las instrucciones definidas y las convierten a su equivalente en la **máquina destino**. Esto es un gran avance respecto a Javascript, que es código interpretado, ya que es posible conseguir un **rendimiento casi nativo** en una aplicación web.

Los navegadores exponen dos funciones cuya utilidad principal es compilar estos ficheros fuente, y aunque parecidas, tienen un gran impacto. Estas dos funciones son **WebAssembly.compile()**<sup>6</sup> y **WebAssembly.compileStreaming()**<sup>7</sup>.

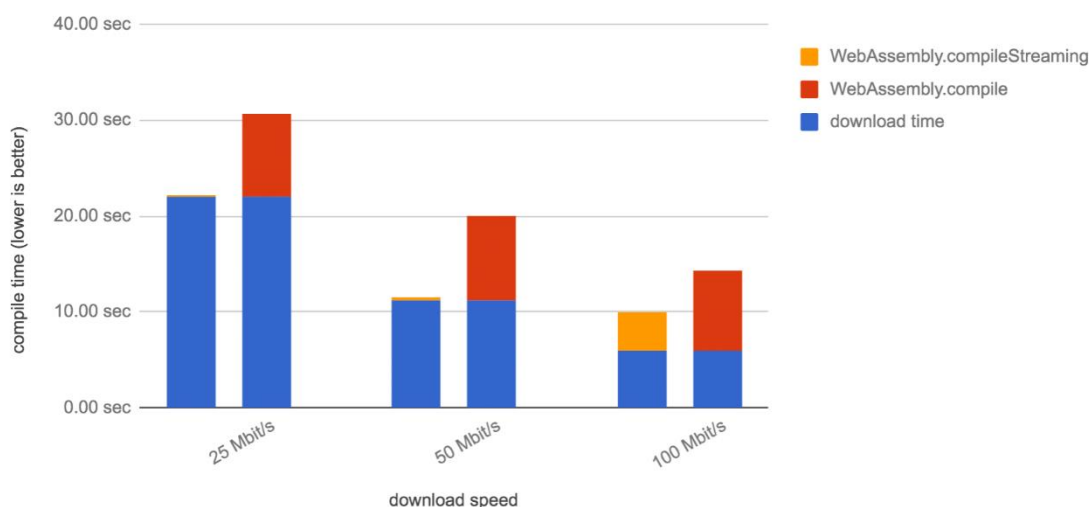


Ilustración 2 Comparación de tiempo entre las dos funciones, de <https://developers.google.com/web/updates/2018/04/loading-wasm>

Como vemos en la Ilustración 2, la función `WebAssembly.compileStreaming()` tiene un mejor rendimiento, esto es debido a que permite compilar el fichero a la vez que se va descargando por partes. Su contraparte, en cambio, necesita primero el fichero entero para poder compilarse, y este comportamiento podríamos decir que es el mismo para

<sup>6</sup> [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/WebAssembly/compile](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/compile)

<sup>7</sup> [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/WebAssembly/compileStreaming](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/compileStreaming)

un fichero de Javascript. El entorno de Javascript del navegador necesita descargar el fichero de Javascript entero, debido a su formato y al ser un lenguaje interpretado.

### 2.1.3 Figma: Un caso real de uso de WebAssembly

Veamos un caso real de la ventaja que supone esto: Figma es una de las empresas que adoptó de manera más temprana esta tecnología, lo que le supuso algunas ventajas respecto a la tecnología que usaban anteriormente. Para tener un poco de contexto, gran parte del producto de esta empresa está escrito en C++ y antes de WebAssembly usaban otra herramienta que permitía compilar este código a asm.js [4], un pequeño subconjunto de Javascript muy optimizado en los motores de Javascript de los navegadores. A pesar de tener un buen rendimiento, debido a que es necesario descargar el fichero completamente, los tiempos de carga se resentían.

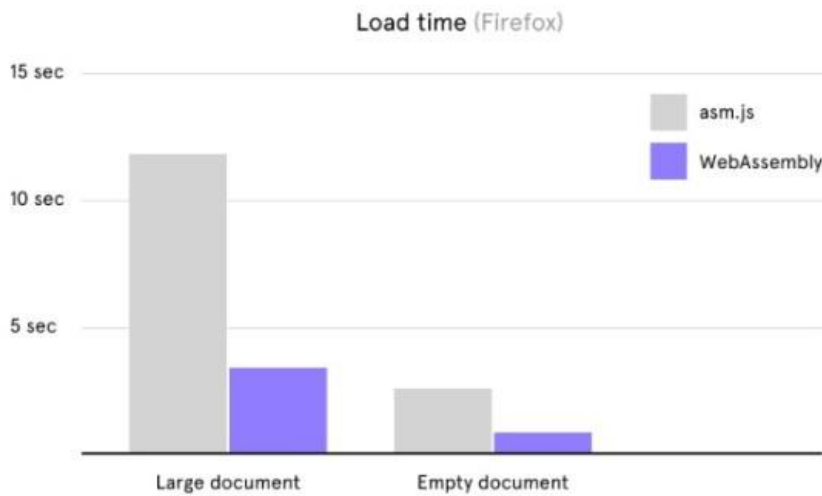


Ilustración 3 Comparación de tiempos de carga entre asm.js y WebAssembly, de <https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/>

Observando la Ilustración 3, observamos la enorme mejoría que supone el hecho de que WebAssembly se pueda compilar a la vez que se descarga.

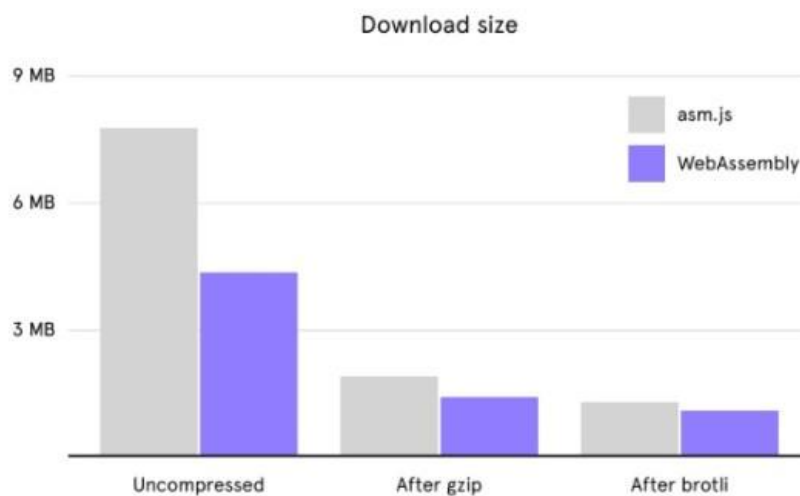


Ilustración 4 Comparación de tamaños de ficheros generados por asm.js y WebAssembly, de <https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/>

En la Ilustración 4 vemos otra clara ventaja respecto a WebAssembly. Y es gracias a que es un formato binario y no textual que es posible tener estas notables mejoras tanto en tiempo de carga como en rendimiento.

## 2.2 Elementos de WebAssembly

En esta sección vamos a comentar en detalle los elementos de los que se compone el lenguaje de WebAssembly, así como sus tipos de datos, elementos particulares y el entorno que lo ejecuta.

El lenguaje de WebAssembly se puede representar de varias formas, como pueden ser su formato binario y su formato textual. Para tener una estructura común, se representará su sintaxis en un formato común que se pueda aplicar a ambas representaciones.

Para mantener la coherencia entre este documento y el documento de especificación de WebAssembly, se dejarán los nombres de sus diferentes elementos en el lenguaje original, en este caso inglés.

### 2.2.1 Tipos de datos

Al ser un lenguaje intermedio cuyo objetivo final es compilarse a código de una máquina objetivo, se tienen ciertas estructuras de datos en un nivel más alto que ensamblador, permitiendo así abstraer ciertas funcionalidades de conjunto de instrucciones específico.

Nombre	Descripción
Value Types	<p>Representa los cuatro tipos de datos numéricos que existen en WebAssembly. Es importante destacar que este tipo <b>en ningún momento contiene ningún valor numérico.</b></p> <ul style="list-style-type: none"><li>• <b>Tipo de dato entero:</b> Son los tipos <b>i32</b> y <b>i64</b> representando valores enteros de 32 y 64 bits respectivamente. Aunque se puede pensar que son valores con signo, este se evalúa dependiendo de la instrucción que lo ejecute.</li><li>• <b>Tipo de dato de punto flotante:</b> Son los tipos <b>f32</b> y <b>f64</b> y permiten operar con elementos decimales. Su formato está definido respecto al estándar IEEE 754-2019 [5]</li></ul>

<b>Result Types</b>	Es una agrupación de <i>Value Types</i> que puede ser vacío.
<b>Function Types</b>	Se definen como una tupla de <i>Result Types</i> que representa los argumentos y la salida de las funciones <b>y otros tipos de elementos</b> .  <i>Result Type</i> → <i>Result Type</i>
<b>Limits</b>	Una estructura con dos valores numéricos: <ul style="list-style-type: none"> <li>• <b>Min:</b> Representa el mínimo tamaño de un elemento.</li> <li>• <b>Max:</b> Representa el máximo tamaño de un elemento. <b>Puede ser opcional.</b></li> </ul>
<b>Memory Types</b>	Representan el tamaño que puede la memoria de WebAssembly, su único valor es el tipo de dato <i>Limits</i> .  Este límite está dado en unidades de una constante definida, conocida como <i>page size</i> , cuyo valor numérico es <b>65536</b> .
<b>Table Types</b>	Representa el tamaño y el tipo de dato que puede contener una tabla en WebAssembly. A día de hoy, <b>las tablas solo pueden contener valores del tipo <i>Function Type</i></b> . En el futuro posiblemente se levante esta restricción.
<b>Global Types</b>	Representa la estructura que tendrá un registro global, detallando valores como el tipo de valor numérico que podrá contener ( <i>Value Type</i> ) y si podrá ser modificado o no.
<b>External Types</b>	Es una estructura de dato capaz de contener solo uno de los siguientes tipos: <ul style="list-style-type: none"> <li>• <i>Function Type</i></li> <li>• <i>Table Type</i></li> <li>• <i>Memory Type</i></li> <li>• <i>Global Type</i></li> </ul> Su principal uso es clasificar ciertos valores que pueden ser importados y exportados desde el entorno de WebAssembly. En secciones posteriores se hablará en detalle de este tema.

### 2.2.2 Estructura de un módulo

En WebAssembly, cada fichero se compone de distintas estructuras de datos, todas contenidas en una mayor, llamada módulo. Este elemento de WebAssembly contiene toda la información para poder inicializar de manera correcta el entorno. Es decir, a este nivel de abstracción no se ejecuta ni traduce ningún tipo de función contenida dentro del módulo.

```
module ::= { types vec(functype),  
             funcs vec(func),  
             tables vec(table),  
             mems vec(mem),  
             globals vec(global),  
             elem vec(elem),  
             data vec(data),  
             start start?,  
             imports vec(import),  
             exports vec(export) }
```

Ilustración 5 Elementos de un módulo, de <https://webassembly.github.io/spec/core/syntax/modules.html>

En la Ilustración 5 podemos observar de los tipos que se compone un módulo y que iremos describiendo a lo largo de esta sección. Como anotación adicional, la palabra *Vec* representa una agrupación de esos elementos, pudiendo ser 0 o más elementos del tipo que tiene entre los paréntesis.

### Índices

Son tipos de **datos numéricos** que referencian a elementos contenidos en las agrupaciones del módulo. Como práctica común en el diseño de lenguajes de programación, su primer valor es el valor 0.

- *Typeidx*
- *Funcidx*
- *Tableidx*
- *Memidx*
- *Globalidx*
- *Localidx*

Es importante mencionar que **este valor no se va a modificar nunca en tiempo de ejecución**, a diferencia de otro tipo de enumeración que veremos en la siguiente sección.

Tenemos otro tipo de índice, el tipo *labelidx*, cuya funcionalidad difiere de los anteriores. Su principal utilidad es dar información útil a ciertas instrucciones que permiten el control de flujo dentro del entorno de WebAssembly.

## Types

El componente *Types* de un módulo no es más que una agrupación de todos los *function types* que se utilizan en ese módulo. Se pueden referenciar mediante el índice de tipo *Typeidx*.

## Funciones o Functions

Es el componente del módulo que guarda una agrupación de tipos *func*, siendo una de las estructuras más importantes de WebAssembly que guardan los conjuntos de instrucciones que el elemento puede ejecutar. En el componente, es posible referenciarlos mediante el tipo *Funcidx*.

*func* ::= {type *typeidx*, locals *vec*(*valtype*), body *expr*}

Ilustración 6 Estructura de un Func, de <https://webassembly.github.io/spec/core/syntax/modules.html>

- El valor *type* nos da la información de las entradas y salidas referenciando a un *function type* que se encuentra en otra parte del módulo.
- Los *locals* son **registros** tipados **locales** a la función, es decir, otra función no podrá acceder directamente a ellos por ningún medio. En este caso, este valor nos está dando información **de qué tipo serán** los registros locales de esa función y no guardan ningún valor numérico. Es posible referenciar los registros locales usando el tipo *Localidx*.
- Por último, el valor *body* representa una agrupación de las distintas instrucciones que puede ejecutar esa función.

## Tables and Element Segments

El componente *tables* de un módulo no es más que una agrupación de las distintas definiciones de tablas que pueden existir en el módulo<sup>8</sup>. Esta estructura es simple, ya que solo tiene un valor con el tipo *Table Type*, el cual representa el número de filas mínimo y (opcionalmente) máximos que puede tener una tabla. En el componente, es posible referenciarlos mediante el tipo *Tableidx*.

Un componente del módulo fuertemente relacionado con estas son los *Element Segments*, que permiten inicializar la tabla con distintos valores.

*elem* ::= {table *tableidx*, offset *expr*, init *vec*(*funcidx*)}

Ilustración 7 Estructura de un Element Segment, de <https://webassembly.github.io/spec/core/syntax/modules.html>

---

<sup>8</sup> En la versión 1.1 de WebAssembly solo se permite tener una tabla por módulo, a pesar de ser representado como una agrupación de tablas.

- El valor `table` representa la tabla que se va a inicializar<sup>9</sup>.
- El valor `offset` representa el desplazamiento que se va a aplicar antes de inicializar la tabla. Es dado por un conjunto de instrucciones que se ejecutan y que devuelven un único valor.
- El valor `init` representa una agrupación de índices de *functions*. Al momento de inicializar la tabla, si el valor final del offset es X, y el tamaño del vector es Y, se inicializaran las filas con la posición X hasta X+Y de la tabla.

## Memories and Data Segments

El componente *mems* de un módulo no es más que una agrupación de la definición de los distintos espacios de memoria que puede haber en el módulo<sup>10</sup>. En su estructura nos encontramos el valor `type`, del tipo *memory type*, que nos dice el tamaño mínimo y máximo del espacio de memoria. En el componente, es posible referenciarlos mediante el tipo *Memidx*.

Como en el caso anterior, tenemos una estructura aparte que nos permite inicializar este espacio de memoria. En este caso hablamos de los *Data Segments*.

*data* ::= {*data memidx, offset expr, init vec(byte)*}

Ilustración 8 Estructura de un Data Segment, de <https://webassembly.github.io/spec/core/syntax/modules.html>

Vemos una estructura bastante similar a la de los *Element Segments* con algunos cambios menores.

- El valor `data` hace referencia al espacio de memoria del módulo que se va a inicializar.
- El valor `offset` tiene exactamente el mismo comportamiento que en el caso anterior.
- El valor `init` tiene casi el mismo comportamiento que en el caso anterior, solo se diferencia en que en vez de ser un agrupamiento de *funcidx*, es un bloque de memoria contigua que se inicializa a partir del valor calculado por el offset.

## Globals

El componente *globals* de un módulo representa una agrupación de *globals*, que son los registros globales que puede tener un módulo. En el componente, es posible referenciarlos mediante el tipo *Globalidx*.

*global* ::= {*type globaltype, init expr*}

Ilustración 9 Estructura de un global, de <https://webassembly.github.io/spec/core/syntax/modules.html>

Observamos que su primer valor es del tipo *globaltype*, que nos dice el tipo de dato numérico que va a guardar este registro y si será mutable o no. En el segundo valor, nos

<sup>9</sup> En la versión 1.1 de WebAssembly este valor siempre será 0.

<sup>10</sup> En la versión 1.1 de WebAssembly, solo es posible tener un único espacio de memoria por módulo.



encontramos un conjunto de instrucciones cuyo resultado será el valor inicial de ese registro global.

## Start

Este componente de un módulo hace referencia a la función que se ejecutará tras inicializar todas las estructuras necesarias para su ejecución (Se expondrá más información en la siguiente sección). Este campo es opcional.

## Exports

El componente *exports* de un módulo es una agrupación de elementos que definen las estructuras del módulo que serán accesibles desde fuera del entorno de WebAssembly.

```
export ::= {name name, desc exportdesc}  
exportdesc ::= func funcidx  
                | table tableidx  
                | mem memidx  
                | global globalidx
```

Ilustración 10 Estructura de un tipo Export, de <https://webassembly.github.io/spec/core/syntax/modules.html>

Como observamos en la Ilustración 10, cada uno de ellos se le asigna un nombre mediante una **cadena de caracteres**. Además, puede hacer referencia a distintos elementos del módulo: *Functions*, *Tables*, *Memories* o *Globals*.

## Imports

El componente *imports* de un módulo es una agrupación de elementos que definen las estructuras que se tendrán que proporcionar desde fuera del entorno de WebAssembly.

```
import ::= {module name, name name, desc importdesc}  
importdesc ::= func typeidx  
                | table tabletype  
                | mem memtype  
                | global globaltype
```

Ilustración 11 Estructura de un Import, de <https://webassembly.github.io/spec/core/syntax/modules.html>

En la Ilustración 11 vemos que su estructura es bastante parecida a la del caso anterior. Primero, tenemos una cadena de caracteres que definen el nombre y diferentes tipos de información que el entorno puede recibir. Se hablará más en profundidad del rol de este tipo de datos en la próxima sección.

### 2.2.3 Estructura de ejecución

WebAssembly presenta una serie de estructuras que son manejables, directa o indirectamente, en tiempo de ejecución por las distintas instrucciones. Los tipos de esta sección y la anterior tienen nombres similares, sin embargo, la principal diferencia es que las estructuras de la sección anterior servían como un esquema de cómo tienen que ser estas estructuras en ejecución.

#### Values

El tipo *Value*, es la estructura dinámica más pequeña que existe en un entorno de WebAssembly. A diferencia del tipo *Value Type*, este sí es capaz de guardar datos numéricos, además de tener marcado siempre el tipo de dato al que pertenece (i32, i64, f32 ó f64), evitando así comportamientos no deseados o fallos lógicos.

#### Store

El tipo *Store* es una estructura dinámica cuya principal finalidad es albergar las distintas estructuras inicializadas en memoria en un único acceso centralizado. Se le podría llamar también un estado global dentro de todo el entorno de WebAssembly.

```
store ::= { funcs   funcinst*,  
            tables tableinst*,  
            mems   meminst*,  
            globals globalinst* }
```

Ilustración 12 Estructura del tipo Store, de <https://webassembly.github.io/spec/core/exec/runtime.html>

En la Ilustración 12 se puede observar su estructura, y, aunque la nomenclatura ha cambiado, cada uno de sus componentes son agrupaciones de instancias<sup>11</sup> de distintas estructuras de las que se hablarán en esta sección.

WebAssembly tiene varios tipos definidos para poder hacer referencia a las estructuras dentro de estas agrupaciones, a las que llama *Addresses*.

```
addr      ::= 0 | 1 | 2 | ...  
funcaddr ::= addr  
tableaddr ::= addr  
memaddr  ::= addr  
globaladdr ::= addr
```

Ilustración 13 Formato de los tipos Addresses, de <https://webassembly.github.io/spec/core/exec/runtime.html>

---

<sup>11</sup> Instancia: Estructura inicializada en memoria con la que el entorno puede interactuar de una manera u otra.

A diferencia de los índices del apartado anterior, que eran valores estáticos que nunca iban a cambiar, los *addresses*, son valores que pueden cambiar entre ejecuciones de un mismo entorno de WebAssembly.

Un ejemplo sencillo de este caso es en el caso de que tengamos dos módulos. Dependiendo del orden en que los inicialicemos en el entorno de WebAssembly, el valor del *address* que referencia a una misma instancia varía en los distintos casos.

### Module Instance

Con la información del tipo *Module* y aplicando un largo proceso<sup>12</sup> obtenemos la estructura dinámica *Module Instance*.

```
moduleinst ::= { types      functype*,  
                 funcaddrs funcaddr*,  
                 tableaddrs tableaddr*,  
                 memaddrs  memaddr*,  
                 globaladdrs globaladdr*,  
                 exports    exportinst* }
```

Ilustración 14 Estructura de un Module Instance, de <https://webassembly.github.io/spec/core/exec/runtime.html>

Se puede observar en la Ilustración 14 que la mayoría de sus componentes son *addresses*, que hacen referencia a algún elemento del *Store*. Esto es debido a que en el proceso de inicialización (*instantiation*) se han inicializado todas las estructuras dinámicas que se encontraban dentro del tipo *Module* y como resultado están todas alojadas en el *Store*, en vez de en el propio *Module Instance*.

### Function Instance

La estructura *Function Instance* representa el tipo *Function* en tiempo de ejecución. Se puede decir que es el tipo *Function* con unos componentes extras que permiten que interactuar con los otros elementos en tiempo de ejecución.

```
funcinst ::= { type functype, module moduleinst, code func }  
           | { type functype, hostcode hostfunc }  
hostfunc ::= ...
```

Ilustración 15 Estructura de un Function Instance, de <https://webassembly.github.io/spec/core/exec/runtime.html>

En la Ilustración 15 se observa que la estructura *Function Instance* engloba a dos posibles tipos o estructuras:

---

<sup>12</sup> <https://webassembly.github.io/spec/core/exec/modules.html#instantiation>

- Una instancia de una función propia dentro del módulo de WebAssembly, es el comportamiento normal. Se observa que tiene el tipo *Module Instance* como componente, ya que gracias a esto es capaz de acceder a los demás elementos en tiempo de ejecución.
- Una instancia de una función que se le ha pasado al entorno de WebAssembly a través de la estructura *Import*. Gracias a esto WebAssembly puede ejecutar código desde fuera de su entorno, aunque tiene serias limitaciones<sup>13</sup> que quedan fuera del ámbito de este documento.

## Frame

Si recordamos de la sección anterior, la estructura *Function* tenía un componente que daba información sobre los distintos registros locales (*locals*) que una función iba a tener. Sin embargo, dentro de la estructura *Function Instance* no se encuentra ningún componente que diga esta información. Esto es debido que esos valores se encuentran en una estructura separada del *Module Instance* y del *Store* al que se llamará *Frame*.

Aunque esta estructura no tiene relevancia dentro de un *Module Instance*, es clave para el correcto de funcionamiento de las *Function Instances*.

*frame* ::= {*locals val\**, *module moduleinst*}

Ilustración 16 Estructura de un Frame, de <https://webassembly.github.io/spec/core/exec/runtime.html>

Dentro de la estructura tenemos la agrupación de *Values*, que representan los registros locales de una *Function Instance*, además del tipo *Module Instance* para acceder a las distintas estructuras en tiempo de ejecución.

## Stack

La mayoría de instrucciones interactúan con una pila o *stack*. A diferencia de las estructuras convencionales de pilas, donde solo se guardan valores numéricos, en la estructura definida por WebAssembly puede guardar tres tipos de datos:

- *Values*: El comportamiento definido en cualquier pila de otro sistema, son los datos numéricos con los que operan las instrucciones.
- *Frame*: Son estructuras donde se guarda el estado o contexto de una función que ha sido inicializada.
- *Labels*: Es un tipo de estructura que sirve para el control de flujo dentro de WebAssembly. Se verá con más detalle próximamente.

A pesar de que la especificación de WebAssembly se habla de la pila como una entidad única que guarda los tres tipos de datos. Es posible separar esta pila en tres entidades distintas o incluso manejar ciertos aspectos de otra manera. En este proyecto se ha optado por esta última opción.

---

<sup>13</sup> <https://webassembly.github.io/spec/core/exec/instructions.html#exec-invoke-host>

## Table Instance

Este tipo es la representación del tipo *Table* en tiempo de ejecución. En esta estructura ya si se puede interactuar con los distintos elementos que se pueden guardar en las filas de la tabla.

$$\begin{aligned} \text{tableinst} & ::= \{ \text{elem } \text{vec}(\text{funcelem}), \text{max } u32^? \} \\ \text{funcelem} & ::= \text{funcaddr}^? \end{aligned}$$

Ilustración 17 Estructura de un Table Instance, de <https://webassembly.github.io/spec/core/exec/runtime.html>

Como se mencionó anteriormente, actualmente esta estructura solo puede almacenar datos del tipo *Function Addresses*, que puede ser vacío o con un valor. Es posible cambiar los valores de la tabla mediante ciertas instrucciones o con ciertas herramientas que provea el entorno.

## Memory Instance

Al igual que los anteriores, este tipo es la representación en tiempo de ejecución del tipo *Memory*, siendo posible guardar e interactuar con el espacio de memoria lineal que se describe en el tipo anterior.

$$\text{meminst} ::= \{ \text{data } \text{vec}(\text{byte}), \text{max } u32^? \}$$

Ilustración 18 Estructura de un Memory Instance, de <https://webassembly.github.io/spec/core/exec/runtime.html>

El tamaño del componente *data* siempre debe ser divisible por el tamaño de página de WebAssembly, en otro caso no será válido. La memoria lineal puede ser modificada mediante instrucciones o herramientas que provea el entorno.

## Global Instances

Es la representación en tiempo de ejecución del tipo *Global*, de manera que ya permite almacenar un valor numérico y modificarlo (si lo permite) mediante instrucciones u otros medios que provea el entorno.

$$\text{globalinst} ::= \{ \text{value } \text{val}, \text{mut } \text{mut} \}$$

Ilustración 19 Estructura de un Global Instance, de <https://webassembly.github.io/spec/core/exec/runtime.html>

## Export instances

El tipo *Export Instance* es una representación en tiempo de ejecución del tipo *Export*. A diferencia de los demás tipos, estos son expuestos por el entorno para que puedan ser accedidos desde fuera.

```

exportinst ::= {name name, value externval}
externval ::= func funcaddr
                | table tableaddr
                | mem memaddr
                | global globaladdr

```

Ilustración 20 Estructura de un Export Instance, de <https://webassembly.github.io/spec/core/exec/runtime.html>

Como observamos, su estructura es idéntica al tipo *Export* con una notable diferencia: Mientras que en este último se utilizan índices para referenciar al contenido que se va a exportar dentro del módulo, en el *Export Instance* se utilizan *addresses* para referenciar las instancias del contenido a exportar.

#### 2.2.4 Instrucciones

En WebAssembly el tema de la ejecución es algo particular, ya que aunque tiene un conjunto de instrucciones como los procesadores, existen otras estructuras de más alto que permiten una mayor expresividad que los anteriores, sobre todo en el caso del control de flujo.

Existen una gran variedad de instrucciones que permiten abarcar los distintos casos que existen debido a la gran cantidad de variables existentes en WebAssembly, como hemos visto en las secciones anteriores.

La mayoría de las instrucciones interactuarán de alguna forma u otra con la pila del entorno, ya sea leyendo o modificando datos de ella. Además, al ser una pila, el modo de funcionamiento está restringido a que una instrucción solo puede acceder a los valores en lo alto de la pila.

La especificación de WebAssembly cataloga las distintas instrucciones en varios grupos relacionados: Instrucciones numéricas, instrucciones paramétricas, instrucciones de variables, instrucciones de manejo de memoria e instrucciones de control de flujo. A lo largo de esta sección se irán explicando con cierto detalle los distintos grupos mencionados.

#### Instrucciones numéricas

Al tener 4 tipos de datos numéricos, WebAssembly necesita una manera de trabajar con ellos de una manera eficiente y correcta. Por lo tanto, este grupo alberga el mayor número de instrucciones, que son necesarias para manejar y transformar de un tipo de dato a otro, entre otras funcionalidades importantes.

```

nn, mm ::= 32 | 64
sx ::= u | s
instr ::= inn.const inn | fnn.const fnn
          | inn.iunop | fnn.funop
          | inn.ibinop | fnn.fbinop
          | inn.itestop
          | inn.irelop | fnn.frelop
          | inn.extend8_s | inn.extend16_s | i64.extend32_s
          | i32.wrap_i64 | i64.extend_i32_sx | inn.trunc_fmm_sx
          | inn.trunc_sat_fmm_sx
          | f32.demote_f64 | f64.promote_f32 | fnn.convert_imm_sx
          | inn.reinterpret_fnn | fnn.reinterpret_inn
          | ...
iunop ::= clz | ctz | popcnt
ibinop ::= add | sub | mul | div_sx | rem_sx
          | and | or | xor | shl | shr_sx | rotl | rotr
funop ::= abs | neg | sqrt | ceil | floor | trunc | nearest
fbinop ::= add | sub | mul | div | min | max | copysign
itestop ::= eqz
irelop ::= eq | ne | lt_sx | gt_sx | le_sx | ge_sx
frelop ::= eq | ne | lt | gt | le | ge

```

Ilustración 21 Estructura de las instrucciones numéricas, de <https://webassembly.github.io/spec/core/syntax/instructions.html>

Se ha decidido mostrar las instrucciones en el formato que presenta la Ilustración 21 para compactar la información necesaria. A continuación se explicarán un par de puntos necesarios para poder entender correctamente este formato:

- A lo largo de la imagen nos encontramos con las cadenas *nn* y *mm* antes de las letras *[i]* ó *[f]*. Tanto *[nn]* como *[mm]* se les puede asignar indistintamente el valor 32 o 64, quedando *[i32, i64, f32, f64]* respectivamente.
- También nos encontramos con la cadena *[sx]*, cuyo valor puede ser *[u]* ó *[s]*. Este valor representa si el operador debe tomar el valor de su operando como un número con signo (*[s]igned*) o sin él (*[u]nsigned*).

Es importante mencionar que esto solo es una manera de listar las distintas instrucciones de un modo compacto, no estando estos valores presentes en ninguna parte del entorno de ejecución.

El funcionamiento de cada instrucción<sup>14</sup> es el típico que se observa en cualquier procesador salvo por los detalles de los múltiples tipos. Debido al gran número de instrucciones, se ha decidido no entrar en detalle en cada una de ellas, siendo la mayoría bastante obvias en su funcionamiento observando el nombre.

### Instrucciones paramétricas

Este grupo está destinado a las instrucciones que pueden interactuar con valores de cualquier tipo, a diferencia del anterior que estaban restringidos por tipos específicos.

```
instr ::= ...  
      | drop  
      | select
```

Ilustración 22 Estructura de las instrucciones numéricas, de <https://webassembly.github.io/spec/core/syntax/instructions.html>

Observando la Ilustración 22, se compone solo de dos instrucciones simples en su funcionamiento. Ambas sirven para manejar de cierta manera la pila del entorno de WebAssembly:

- Drop: Descarta un valor en lo alto de la pila, sin importar el tipo que sea.
- Select: Permite elegir uno de dos valores de la pila, basándose en el valor numérico de un tercer valor. En la Ilustración 23 podemos ver en un diagrama el funcionamiento básico de esta instrucción. Se mantendrá el valor a o b en la pila dependiendo del valor de c.

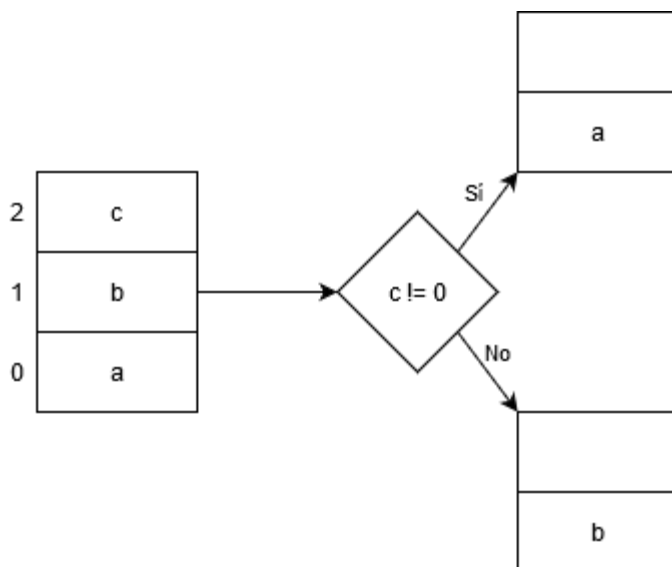


Ilustración 23 Diagrama de funcionamiento de la instrucción Select

<sup>14</sup> <https://webassembly.github.io/spec/core/exec/instructions.html>



## Instrucciones de variables

En este grupo nos encontramos una serie de instrucciones que nos permiten interactuar con los registros locales (*Locals*) y globales (*Globals*) que existen en WebAssembly.

```
instr ::= ...  
      | local.get localidx  
      | local.set localidx  
      | local.tee localidx  
      | global.get globalidx  
      | global.set globalidx
```

Ilustración 24 Estructura de las instrucciones de variables

A continuación se muestra una tabla con una breve explicación de lo que hace cada una de estas instrucciones:

Instrucción	Explicación
<b>local.get</b>	Inserta en lo alto de la pila el valor del <b>registro local</b> al que referencia el valor de <i>localidx</i> .
<b>local.set</b>	Elimina el valor de lo alto de la pila y se lo asigna al <b>registro local</b> referenciado por <i>localidx</i> .
<b>local.tee</b>	El funcionamiento es parecido a la instrucción <i>local.set</i> . La única diferencia es que esta no toca en ningún momento la pila.
<b>global.get</b>	Inserta en lo alto de la pila el valor del <b>registro global</b> al que referencia el valor de <i>globalidx</i> .
<b>global.set</b>	Elimina el valor de lo alto de la pila y se lo asigna al <b>registro global</b> referenciado por <i>globalidx</i> .

## Instrucciones de manejo de memoria

En este grupo nos encontramos con las instrucciones que permiten el manejo de la memoria lineal. Como muchos conjuntos de instrucciones, tiene las típicas instrucciones de *load* y *store*.

```

memarg ::= {offset u32, align u32}
instr  ::= ...
          | inn.load memarg | fnn.load memarg
          | inn.store memarg | fnn.store memarg
          | inn.load8_sx memarg | inn.load16_sx memarg | i64.load32_sx memarg
          | inn.store8 memarg | inn.store16 memarg | i64.store32 memarg
          | memory.size
          | memory.grow

```

Ilustración 25 Estructura de instrucciones de manejo de memoria

En la Ilustración 25 se puede observar que, al igual que en las instrucciones numéricas se comparte la sintaxis a la hora de representar ciertas instrucciones (nn, mm y sx). Tenemos instrucciones de *load* y *store* para cada tipo de dato numérico y, además, otras específicas para poder controlar la memoria en bloques menores de 32 bits. Por último, es necesario mencionar las dos últimas instrucciones:

- *Memory.size*: Devuelve en lo alto de la pila el tamaño de la memoria
- *Memory.grow*: Toma el valor en lo alto de la pila (Que debe ser del tipo *i32*) e intenta expandir el tamaño de la memoria con ese valor multiplicado por el tamaño de página. Es posible que falle, en cuyo caso pondrá en lo alto de la pila un -1.

## Instrucciones de control de flujo

En este grupo se encuentran las instrucciones que permiten manejar el flujo de un programa, pudiendo saltar a ciertas instrucciones o incluso llamar a otras funciones de WebAssembly.

```
blocktype ::= typeidx | valtype?  
instr      ::= ...  
            | nop  
            | unreachable  
            | block blocktype instr* end  
            | loop blocktype instr* end  
            | if blocktype instr* else instr* end  
            | br labelidx  
            | br_if labelidx  
            | br_table vec(labelidx) labelidx  
            | return  
            | call funcidx  
            | call_indirect typeidx
```

Ilustración 26 Estructura de las instrucciones de control de flujo, de <https://webassembly.github.io/spec/core/syntax/instructions.html#control-instructions>

En este caso, se explicarán solo algunas instrucciones pues las demás se explicarán con detalle junto al control de flujo en la próxima sección.

Instrucción	Explicación
<b>Nop</b>	Esta instrucción no tiene ningún efecto en ninguna parte del entorno, solo sirve para hacer que pase un ciclo de instrucción.
<b>Unreachable</b>	Hace que se lance una excepción incondicional.
<b>Call</b>	Permite llamar a otra función, creándose un <i>frame</i> nuevo en la pila.
<b>Call_indirect</b>	Permite llamar a otra función de manera indirecta a través del tipo <i>Table Instance</i> . La referencia de la función en la tabla se encuentra en lo alto de la pila.

### 2.2.5 Control de flujo

En esta sección vamos a explicar en detalle el resto de instrucciones de control de flujo que no se explicaron en la sección anterior. El control de flujo de WebAssembly difiere bastante de otros sistemas, principalmente por su particular forma de organizar bloques de instrucciones.

Vamos a abstraer por un momento la palabra de bloque o *Block*. **Un bloque no es más que una secuencia de instrucciones que tiene ciertas propiedades especiales.** Estos bloques a través de ciertas instrucciones son capaces de saltar fragmentos de código, elegir un bloque u otro o incluso repetir otra vez el bloque.

Anteriormente se ha mencionado las palabras *label* y el tipo *labelidx*, a continuación se explicará con detalle lo que es y para qué sirve. Un *label* no es más que una posición dentro del código que se asigna al definir un bloque de código. En cambio, el tipo *labelidx* representa una posición relativa a los bloques que haya anidado.

En WebAssembly existen tres tipos de bloques:

- **Block:** Contiene una sola secuencia de código y especifica la posición de su *label* **justo después de la última instrucción del bloque.**
- **Loop:** Contiene una sola secuencia de código y especifica la posición de su *label* **en la primera instrucción dentro de su conjunto de instrucciones.** Es importante mencionar que el **comportamiento de este bloque es igual al de Block**, solo varía la posición de su *label*. Esto es importante porque **su nombre puede dar lugar a confusión** y dar la sensación de que lo que está dentro de este bloque es un bucle que se va a repetir de manera indefinida y que solo se puede salir de él con una instrucción de control de flujo. Sin embargo, **es justo lo contrario**, para que se repita las instrucciones de este bloque se debe usar obligatoriamente una instrucción de salto que diga explícitamente eso.
- **If/Else:** Contiene dos secuencias de código separadas, mediante el valor en lo alto de la pila se elige qué secuencia se ejecutará. Su *label* se posiciona **justo después de la última instrucción de la segunda secuencia**, independientemente de cual se haya ejecutado.

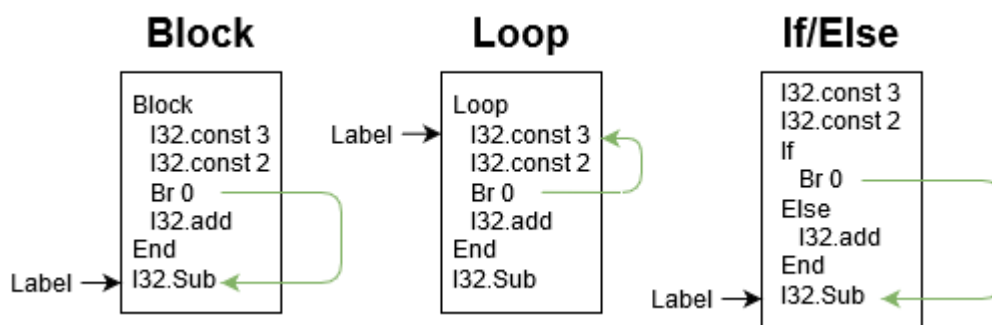


Ilustración 27 Diferencia entre los distintos bloques

En la Ilustración 27 podemos ver la diferencia visualmente entre los distintos bloques y donde se posiciona su *label*. Es importante mencionar que **los bloques**, al igual que las

funciones, pueden tener asignado un tipo *Function Type*, es decir, **permiten tener argumentos y un resultado**.

Ahora vamos a hablar de las instrucciones que permiten interactuar con estos bloques. La instrucción más importante de entender es *br*, pues de ella deriva el funcionamiento de todas las demás. Esta instrucción recibe un argumento de tipo *labelidx* que, como se dijo antes, representa una posición relativa a los bloques. Su valor mínimo es 0, y cuanto más alto sea, se saldrá de un número mayor de bloques anidados. En la Ilustración 28 se puede ver un ejemplo claro de este funcionamiento.

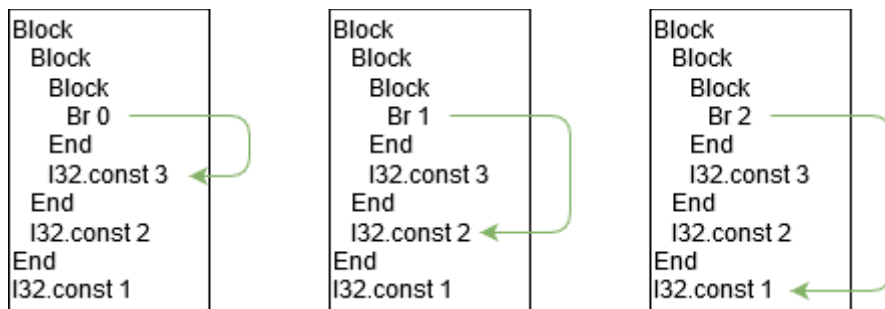


Ilustración 28 Diferentes ejecuciones de la instrucción BR

Con este conocimiento, podemos relacionar ciertos comportamientos de la instrucción BR con los distintos tipos de bloques:

- Block ó If/Else: Si se ejecuta un Br 0 dentro de este bloque, el comportamiento es similar a una sentencia **break** en otros lenguajes de programación.
- Loop: Si se ejecuta un Br 0 dentro de este bloque, el comportamiento es similar a una sentencia **continue** en otros lenguajes de programación.

Un tema importante con este tipo de instrucciones, es que cuando se hace un salto, ya sea condicional o incondicional, el estado de la pila se restaura a un estado similar al que tenía antes de entrar en el bloque. Se menciona que un estado similar debido a que aquí entra en juego el *Function Type* que tenga asignado el bloque.

Argumento usado
Resultado

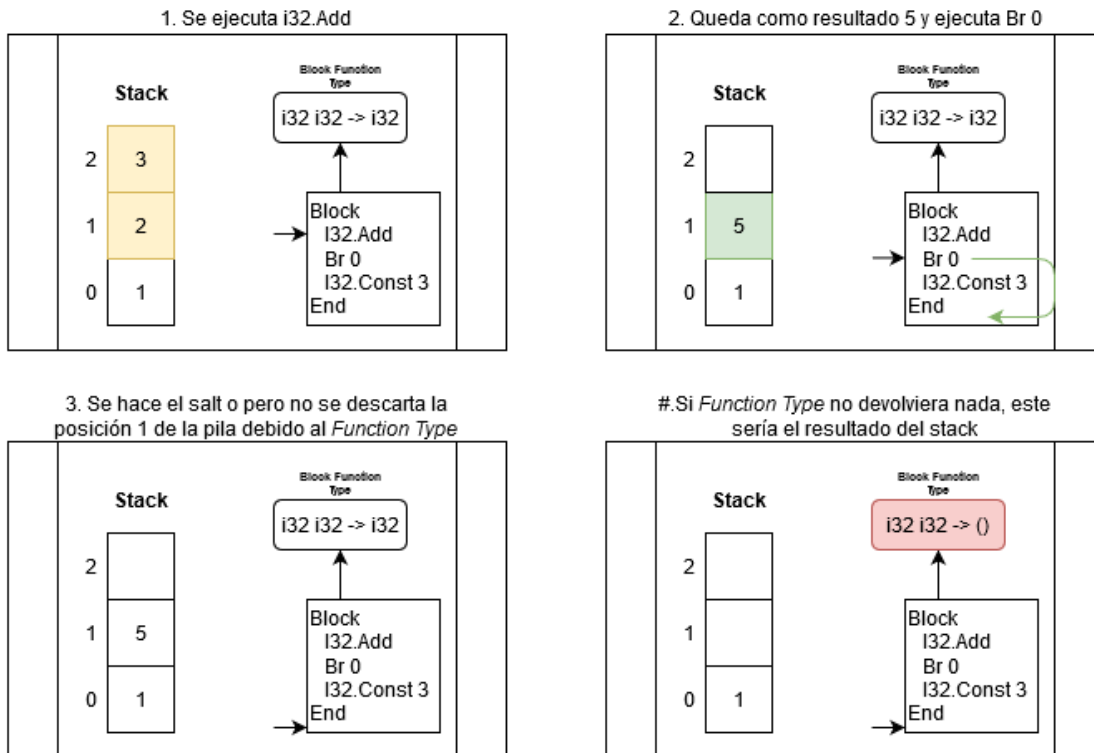


Ilustración 29 Demostración de un bloque con *Function Type*

En la Ilustración 29 podemos ver el estado de la pila conforme se avanza en la ejecución de un bloque con un *Function Type* asignado. En el cuarto paso se ve que sucedería en el caso de que no devolviera ningún tipo, se perdería el resultado de la operación hecha dentro de ese bloque.

Para terminar con esta sección vamos a hablar brevemente del resto de instrucciones de control de flujo:

Instrucción	Explicación
<b>Br_if <i>labelidx</i></b>	Permite realizar un salto condicional basándose en el valor de lo alto de la pila. Si el valor es <b>distinto de 0</b> , se ejecutará el salto condicional con el mismo comportamiento que la instrucción br.

<p><b>Br_table</b> <i>vec(labelidx) labelidx</i></p>	<p>Permite realizar un salto mediante el uso de una tabla.</p> <p>Un ejemplo de esta instrucción sería:  <code>Br_table [2,3,1] 0</code></p> <p>En lo alto de la pila se encuentra un valor que puede referenciar a algún valor del vector mediante su posición [0..]. Se realiza el salto condicional con el valor de la tabla con el mismo comportamiento que la instrucción br.</p> <p>Si el valor de la pila es <b>mayor que la longitud del vector</b>, se usa el valor por defecto que es el segundo argumento de esta instrucción.</p>
<p><b>Return</b></p>	<p>Esta instrucción permite un <b>salto incondicional</b> al <b>cuerpo de la función</b>, es decir, si existen muchos bloques anidados y se ejecuta esta instrucción dentro, <b>se saldrá todos los bloques</b>.</p> <p>Tiene exactamente el mismo comportamiento que la instrucción br, siendo una manera más directa de salir de todos los bloques, en vez de especificarlo manualmente.</p>

Un último punto a tratar son las excepciones. En WebAssembly existen varias instrucciones que al cumplirse una serie de condiciones de error, se lanzará una excepción que parará cualquier función en ejecución al momento y sin posibilidad de continuar. Un ejemplo típico de excepción es a la hora de intentar dividir por cero.

## 2.3 Otras características importantes

En este capítulo hemos hablado un poco de las particularidades de WebAssembly y de cómo se estructura internamente un entorno que puede ejecutarlo. Sin embargo, hay varios temas de los que no se han hablado porque no entran dentro del alcance de este documento. Aun así, se les hará un pequeño resumen y se expondrá fuentes para poder informarse más en caso de que el lector esté interesado en ampliar su conocimiento.

### 2.3.1 Formatos de fichero disponibles

Hasta ahora se ha hablado solo de ficheros sin especificar un formato en específico. Aunque lo normal es que solo tengamos un único formato, normalmente binario, WebAssembly cuenta con hasta un total de 3 formatos distintos: uno binario y dos textuales. Cada uno de ellos tienen una finalidad distinta, mientras que el binario es

utilizado para un mayor rendimiento, los formatos textuales permiten una lectura más clara del código y otras finalidades como un mejor soporte de ejecución de pruebas.

Una de las principales ventajas del formato binario es un menor tamaño de fichero y poder utilizar una de las capacidades más potentes de un navegador web, que es usar la función `WebAssembly.compileStreaming()` de la que se habló brevemente en este capítulo. En la especificación de `WebAssembly` nos encontramos una sección entera dedicada a este formato<sup>15</sup>, dando todos los detalles necesarios.

A diferencia del formato binario, el formato textual aboga por una lectura más sencilla del código a cambio de perder todas las ventajas mencionadas en el párrafo anterior. En la Ilustración 30 vemos una simple función de suma escrita en este formato, se puede apreciar que se permite el uso de ciertas palabras precedidas por un "\$" para mejorar la legibilidad del código. Al igual que en el caso anterior, la especificación de `WebAssembly` tiene una sección entera dedicada a este formato<sup>16</sup>.

```
1 (module
2   (func $add (param $lhs i32) (param $rhs i32) (result i32)
3     local.get $lhs
4     local.get $rhs
5     i32.add)
6   (export "add" (func $add))
7 )
```

Ilustración 30 Ejemplo del formato textual, de [https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding\\_the\\_text\\_format](https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format)

Es importante mencionar que existen herramientas<sup>17</sup> que permiten interactuar con los distintos formatos, ya sea convirtiendo de manera sencilla un fichero de un formato a otro o incluso optimizar el tamaño de un fichero en formato binario.

Para terminar, existe un segundo formato textual que no es más que una ampliación del primero pero destinado a la ejecución de pruebas, se hablará más de este tema en un capítulo dedicado enteramente a la verificación del funcionamiento del proyecto.

### 2.3.2 Validación de un módulo

Además de toda la estructura que hemos explicado de `WebAssembly`, existen una serie de normas lógicas que permiten verificar la estructura de un módulo en tiempo de compilación. Mayormente se centra en verificar que no haya inconsistencias entre las distintas interacciones de los elementos del módulo. Por ejemplo, comprueba que si una instrucción requiere que el valor de lo alto de la pila en ese momento sea del tipo `i32`, se pueda estar seguro que se va a cumplir esa condición antes de ejecutarlo.

En los navegadores se tiene una función [6] que permite validar de forma sencilla si un fichero de `WebAssembly` en formato binario pasa todas estas verificaciones

---

<sup>15</sup> <https://webassembly.github.io/spec/core/binary/index.html>

<sup>16</sup> <https://webassembly.github.io/spec/core/text/index.html>

<sup>17</sup> <https://github.com/WebAssembly/wabt>



correctamente. Además, la especificación de WebAssembly tiene una sección entera<sup>18</sup> dedicada a la verificación en caso de que uno quiera implementar un sistema propio.

En este proyecto por temas de tiempo se ha optado por dejar esta funcionalidad a los propios navegadores, ya que consumiría mucho tiempo comprobar que todo esté correctamente.

---

<sup>18</sup> <https://webassembly.github.io/spec/core/valid/index.html>



# 3

## Rust

En este capítulo se va a definir y explicar el lenguaje de programación en el que está implementado la gran mayoría de este proyecto.

### 3.1 ¿Qué es Rust?

Rust es un lenguaje de programación de propósito general desarrollado por Mozilla para su uso interno, que posteriormente se liberó al público. Su principal cualidad es que promete controlar ciertos elementos a bajo nivel (Como C o C++, por ejemplo) y rendimiento con una sintaxis de más alto nivel (Java, por ejemplo) y con ciertas garantías a la hora de manejar la memoria sin necesidad de recolector de basura o ninguna penalización en ejecución [2].

Una de las principales bazas de Rust es su seguridad a la hora de manejar la memoria, ya que a diferencia de otros lenguajes como C o C++, este fuerza a cumplir una serie de reglas en tiempo de compilación que evitan errores o ciertos hábitos que a la larga harán que surjan errores inesperados. El equipo de seguridad de Microsoft ha estado planteando usar Rust para paliar uno de los errores más extendidos en su sistema operativo Windows, los errores basados en el manejo incorrecto de la memoria [6].

### 3.2 ¿Por qué Rust?

De todos los lenguajes con soporte para compilar a un formato de WebAssembly, Rust es el que mejor y más comunidad tiene, permitiendo un desarrollo sin demasiadas complicaciones. Este proyecto no habría sido posible hacerlo a tiempo sin las herramientas y librerías proporcionadas por la organización Bytecode Alliance<sup>19</sup>, obviamente escritas en Rust.

---

<sup>19</sup> <https://bytecodealliance.org/>

Otro de las razones por las que se ha apostado por Rust en este proyecto es por potente y versátil sistema de tipos y que, a diferencia de otros lenguajes de programación, no apuesta por un lenguaje orientados a objetos de manera pura, **desechando por completo la herencia de clases**. A cambio, apuesta por un sistema de interfaces a las que llaman *Traits*<sup>20</sup>. De manera que gracias a todo esto, podemos escribir el código siendo su formato muy cercano a lo que se muestra en la especificación de WebAssembly.

```

module ::= { types vec(funcType),
            funcs vec(func),
            tables vec(table),
            mems vec(mem),
            globals vec(global),
            elem vec(elem),
            data vec(data),
            start start2,
            imports vec(import),
            exports vec(export) }

```

```

#[derive(Debug, Default, Clone)]
pub struct Module {
    pub types: Vec<FuncType>,
    pub funcs: Vec<Func>,
    pub tables: Vec<Table>,
    pub mems: Vec<Memory>,
    pub globals: Vec<Global>,
    pub elem: Vec<Segment<TableIdx, FuncIdx>>,
    pub data: Vec<Segment<MemIdx, u8>>,
    pub start: Option<FuncIdx>,
    pub imports: Vec<Import>,
    pub exports: Vec<Export>,
}

```

Ilustración 31 Diferencia entre especificación y código escrito

En la Ilustración 31 es posible ver la similitud que es posible conseguir entre especificación y código y que, con la excepción de algunos atributos de la estructura, hace posible razonar sobre el funcionamiento del proyecto si se ha entendido en parte la especificación de WebAssembly, o al menos las partes explicadas en el capítulo anterior.

Otro punto a favor de Rust para este proyecto es lo que se llama **su sistema de ownership**<sup>21</sup> de variables. Es un estricto sistema sobre quien puede poseer y utilizar las variables y se basa en varias reglas sencillas:

1. **Toda variable es inmutable** por defecto, a menos que se indique lo contrario explícitamente usando la sintaxis del lenguaje. Esto se aplica también a las referencias a esa variable.
2. Una variable solo puede tener un dueño definido dentro de su ámbito (Por ejemplo, dentro de una función). Es posible **leer y modificar** el valor de esta variable **mediante el uso de referencias**. Además, es posible **traspasar la pertenencia** de una variable de un ámbito a otro, en este caso la variable dejará de estar disponible en el antiguo ámbito.
3. Una variable puede tener o **infinitas referencias inmutables** o **solo una referencia mutable**. Cualquier intento de saltarse esta regla hará que el código no compile.

<sup>20</sup> <https://doc.rust-lang.org/stable/book/ch10-02-traits.html>

<sup>21</sup> <https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html>

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}", {}, r1, r2);
```

Ilustración 32 Fragmento de código que no compilará, de <https://doc.rust-lang.org/stable/book/ch04-02-references-and-borrowing.html>

En la Ilustración 32 podemos ver en juego dos de las tres reglas de las que hemos hablado. En la primera línea, la variable se define explícitamente como una variable que se puede modificar usando la palabra clave *mut*. En las dos siguientes líneas, se intenta obtener dos referencias mutables a un mismo elemento, lo que incumple la tercera regla, ya que solo se puede tener una única referencia mutable.

Gracias a este sistema se ha podido desarrollar el proyecto **con garantías** de que los únicos fallos que se pueden encontrar son a la hora de implementar la lógica y no fallos que ocurren por un efecto adverso de intentar modificar cierta variable.

```
pub(crate) fn local_get(
    stack: &mut StackValue,
    context: &FunctionContext,
    idx: LocalIdx,
) -> IntResult {
    let val = context.locals.get(idx as usize).ok_or(intError!(
        "Local get: Due to validation, local should exist"
    ))?;

    stack.push(val.clone());

    Ok(IntAction::Continue)
}
```

Ilustración 33 Implementación del operador Local.Get

Veamos un ejemplo claro de esto, en la Ilustración 33 vemos una función con 3 argumentos. Vamos a analizar cada argumento de forma separada:

- *Idx*: Es del tipo *LocalIdx* y en este caso, la función es dueña de esta variable, por lo que puede leer y modificar su valor sin ningún problema y sin afectar a otra parte del código.
- *Context*: Es una **referencia inmutable** a una estructura del tipo *FunctionContext*, y, aunque no es dueña de la variable a la que referencia, es posible leer sus datos y/o usar cualquier método que no requiera tener acceso mutable a ella.
- *Stack*: Es una **referencia mutable** a una estructura del tipo *StackValue*, de manera que tiene permisos tanto de lectura como de modificación. Sin embargo, la función no es dueña de la variable.

Para terminar, otra característica que ha sido de gran ayuda durante el desarrollo del proyecto y específicamente para un entorno de *WebAssembly* ha sido el uso de lo que se llaman **macros**. Una macro la podríamos definir como una función que permite escribir código. Una de sus principales utilidades es evitar la repetición de código.

```

macro_rules! impl_reinterpret_ops {
    ($from: ty, $to: ty) => {
        impl Reinterpret<$to> for $from {
            #[inline]
            fn reinterpret(self) -> $to {
                let bits = <$from>::to_le_bytes(self);
                <$to>::from_le_bytes(bits)
            }
        }
    };
}

impl_reinterpret_ops!(f32, i32);
impl_reinterpret_ops!(f64, i64);
impl_reinterpret_ops!(i32, f32);
impl_reinterpret_ops!(i64, f64);

```

Ilustración 34 Implementación de las instrucciones de Reinterpret

En la Ilustración 34 vemos un ejemplo sencillo de una macro usada en el proyecto. En tiempo de compilación, las llamadas a las macros (En Rust se distinguen por la exclamación (!) al final de la llamada) se sustituyen por el contenido de dentro de la macro. Esto permite **evitar la repetición de código** y, por lo tanto, posibles fallos al escribir el código que acaben en un error oculto en el código. Sin embargo, el uso intensivo de esta característica **aumenta en gran medida el tiempo de compilación**.

Los lenguajes de programación C y C++ también tienen un sistema de macro, sin embargo nos encontramos con una diferencia crucial entre sus sistemas y el de Rust: Rust verifica en tiempo de compilación los tipos y el resultado de las macros, mientras que en C/C++ en tiempo de compilación solo se sustituye el fragmento de código, sin realizar ningún tipo de comprobación.

### 3.3 Relación con WebAssembly

Además de las ventajas mencionadas en la sección anterior, otra es la gran sinergia que tiene con WebAssembly, teniendo **soporte nativo** para compilar desde Rust a WebAssembly<sup>22</sup>. Se complementa además, con una **gran variedad de herramientas y librerías** que facilitan más el desarrollo que en otros lenguajes de programación.

Wasm-pack<sup>23</sup> es un conjunto de herramientas que permiten generar proyectos en Rust con extrema facilidad. Este a su vez hace uso de una librería de Rust llamada Wasm-bindgen<sup>24</sup>, que se encarga de facilitar las funciones exportadas de un módulo de WebAssembly como si fueran funciones nativas de Javascript y viceversa. Otro aspecto muy importante es que se encarga de **manejar toda la lógica cuando queremos usar**

<sup>22</sup> <https://doc.rust-lang.org/nightly/rustc/platform-support.html>

<sup>23</sup> <https://github.com/rustwasm/wasm-pack>

<sup>24</sup> <https://github.com/rustwasm/wasm-bindgen>

una cadena de caracteres en alguna función de WebAssembly al no existir de manera nativa un tipo *String* dentro de WebAssembly.

```
1 request = new XMLHttpRequest();
2 request.open('GET', 'simple.wasm');
3 request.responseType = 'arraybuffer';
4 request.send();
5
6 request.onload = function() {
7     var bytes = request.response;
8     WebAssembly.instantiate(bytes, importObject).then(results => {
9         results.instance.exports.exported_func();
10    });
11 };
```

Ilustración 35 Llamada de una función de WebAssembly en Javascript directamente desde el fichero .wasm, de [https://developer.mozilla.org/en-US/docs/WebAssembly/Loading\\_and\\_running](https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running)

En la Ilustración 35 se puede apreciar cómo se llamaría a una función exportada sin el uso de Wasm-bindgen. En las líneas 1 a 4 se hace una petición para descargar el fichero binario de WebAssembly y en las líneas 6 a 11 se define el comportamiento que sucederá cuando se complete la petición entera. Es importante recalcar que **Javascript es asíncrono por naturaleza**, de manera que puede llegar a ser complicado manejar la lógica en un proyecto que abarque más que un simple ejemplo.

Import JavaScript things into Rust and export Rust things to JavaScript.

```
use wasm_bindgen::prelude::*;

// Import the `window.alert` function from the Web.
#[wasm_bindgen]
extern "C" {
    fn alert(s: &str);
}

// Export a `greet` function from Rust to JavaScript, that alerts a
// hello message.
#[wasm_bindgen]
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}
```

Use exported Rust things from JavaScript with ECMAScript modules!

```
import { greet } from "./hello_world";

greet("World!");
```

Ilustración 36 Ejemplo de uso de Wasm-bindgen, de <https://github.com/rustwasm/wasm-bindgen>

En cambio, en la Ilustración 36 podemos ver un ejemplo sencillo del uso de esta librería. En el primer fragmento de código se observa que en la primera función se está exponiendo la función de *window.alert* de Javascript en el entorno de WebAssembly (Como vimos anteriormente, se pueden **llamar a funciones externas al entorno** a través de los *imports*). Posteriormente, se define una función *greet* que recibe como parámetro un tipo *String*. Una vez se compila todo esto, *wasm-bindgen* se encarga de generar todo

el *glue code*<sup>25</sup> que conecta WebAssembly y Javascript y, como se ve en el segundo fragmento de código, es tan simple como llamar a una función de Javascript.

Para terminar, al contar con el soporte de Mozilla y otras entidades como Bytecode Alliance, existen librerías de Rust que permiten facilitar ciertas tareas relacionadas con el manejo de ficheros de WebAssembly. En concreto, para este proyecto se han usado varias librerías que han ahorrado tener que implementar la lógica que implica leer los ficheros de WebAssembly en sus distintos formatos y pasarlo a una estructura en memoria que permitan su manejo. Entre las distintas librerías que existen, se ha utilizado *Wasm-parser*<sup>26</sup> para el manejo de ficheros binarios y *wat*<sup>27</sup> para el manejo de ficheros en formato textual. Estas dos últimas librerías han sido desarrolladas y son mantenidas por Bytecode Alliance.

---

<sup>25</sup> *Glue code*: Código cuyo único propósito es ofrecer interoperabilidad entre dos sistemas que no son compatibles de por sí.

<sup>26</sup> <https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasmparser>

<sup>27</sup> <https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wat>



# 4

## Estructura del proyecto

En este capítulo se va a hablar de la estructura final del proyecto y del proceso que se ha seguido para llegar hasta ella.

### **4.1 Consideraciones iniciales**

Al inicio del proyecto se pensaron distintas maneras de planificarlo y sobre cuál sería su estructura, así como las distintas librerías que se podrían utilizar para facilitar el desarrollo. En esta sección se va a describir estos procesos y explicar por qué fallaron al final.

#### **4.1.1 Elección de librerías**

Durante las primeras semanas del proyecto se analizó el documento de especificación de WebAssembly para formar una fuente conocimiento segura que sirviera durante el desarrollo del proyecto. Tras tener un conocimiento básico de cómo funcionaba, lo siguiente era analizar el elenco de librerías que iban a permitir la lectura de ficheros de WebAssembly. Se descartaron algunas que no tenían las características que uno deseaba en el proyecto y hubo que tomar la decisión final entre dos:

Parity\_wasm<sup>28</sup>

Ventajas	Desventajas
<ul style="list-style-type: none"><li>• Tiene implementada su propia estructura interna de un módulo de WebAssembly, lo que ahorraría bastante trabajo.</li><li>• Es una librería que se usa junto con un intérprete de WebAssembly.</li></ul>	<ul style="list-style-type: none"><li>• El repositorio no se actualizaba desde hacía meses, aunque ahora ha vuelto a tener más actividad.</li><li>• En caso de encontrar un error en alguna estructura tendría que esperar que los desarrolladores lo corrigieran.</li></ul>

Wasmparser<sup>29</sup>

Ventajas	Desventajas
<ul style="list-style-type: none"><li>• Se puede implementar una estructura interna propia de WebAssembly, pudiéndolo adaptar a las necesidades particulares que se necesiten.</li><li>• Es posible realizar el desarrollo de la estructura interna de manera iterativa.</li></ul>	<ul style="list-style-type: none"><li>• No tiene una estructura definida en memoria, sino que a medida que va leyendo un fichero expone distintas estructuras que se destruyen al leer el siguiente elemento. Se tendría que implementar una estructura interna de WebAssembly.</li><li>• En caso de que no pudiera continuar usándola habría perdido bastante tiempo.</li></ul>

Habiendo analizado detenidamente cada librería, la opción más obvia sería utilizar Parity\_wasm para ahorrar tiempo. Sin embargo, durante el análisis de esta librería y mirando su documentación, se vieron algunas decisiones de diseño de algunas estructuras que iban a dificultar bastante el desarrollo del proyecto. Debido a todo esto, se optó por utilizar Wasmparser e **implementar desde cero** toda la estructura interna, **logrando más control del proyecto** y también un mayor entendimiento del funcionamiento de WebAssembly. Aunque a cambio se le ha tenido que dedicar menos tiempo a otras partes.

#### 4.1.2 Planificación inicial

Tras la elección definitiva de una librería para la lectura de ficheros, lo siguiente era planificar la forma en la que se iba a realizar el proyecto. Para ello, se definieron una serie de requisitos funcionales (Véase el Apéndice C)

---

<sup>28</sup> <https://github.com/paritytech/parity-wasm>

<sup>29</sup> <https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasmparser>

Tras definir los requisitos, se optó por ir implementando los requisitos uno a uno mediante una metodología iterativa parecida a Scrum.



Ilustración 37 Tablero de Trello con algunas tareas.

En la Ilustración 37 podemos ver un ejemplo de cómo se estaba organizando el proyecto con esta planificación. El principal problema detrás de este tipo de planificación es que buscaba tener hecho funcionalidades que dependían de ciertas estructuras de WebAssembly. Por ejemplo, para implementar el requisito 4.3.1 eran necesarias ciertas estructuras de WebAssembly que no estaban implementadas todavía.

El desarrollo avanzó sin demasiados problemas hasta que en un punto que, por el tipo de planificación que se estaba haciendo, se iban implementando las estructuras de WebAssembly a medida que hacían falta. Esto ocasionó varios problemas graves en la estructura del código que dieron lugar a la elección de empezar todo el proyecto desde cero y tomar otra estrategia. Por suerte, una gran parte del código ya implementado se pudo reutilizar sin demasiados cambios.

El principal problema que resalta en esta forma de planificar el proyecto es el descontrol que ocasiona el querer tener ciertas funcionalidades listas desde el inicio del proyecto. Esto en circunstancias normales no habría sido un problema, sin embargo, al tener dos partes separadas, el simulador y la vista web, era bastante complicado mantener las dos partes sincronizadas ya que cualquier mínimo cambio producía incompatibilidades entre ellas.

#### 4.1.3 Organización de las distintas partes del proyecto

Habiendo hablado ya de la elección de librerías y de la planificación inicial del proyecto, en esta sección nos vamos a centrar en las distintas opciones que se pensaron para organizar y conectar las dos partes principales de este proyecto. Por una parte tenemos el simulador escrito en Rust y por otra la vista web.

A la hora de conectar estas dos partes, existen una gran cantidad de maneras en la que es posible realizar esto, solo vamos a discutir un par de estas opciones y las que más estas estuvieron de implementarse.

### Conectar directamente el simulador con la vista web

En esta opción, se utiliza directamente la librería de wasm-bindgen para exponer ciertas funciones de Rust que permitan manejar el estado del simulador directamente desde la vista web. Sin embargo, wasm-bindgen tiene una **serie de limitaciones técnicas** que imposibilitan esta opción por detalles de implementación interna del proyecto.

Wasm-bindgen solo permite un rango **muy limitado** de tipos que puede exponer, de forma que habría que adaptar el simulador a estas limitaciones complicando de manera excesiva la implementación de ciertas partes.

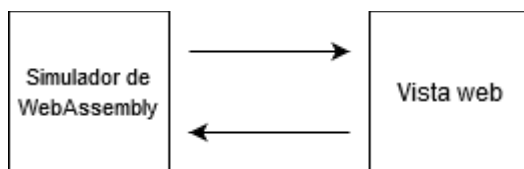


Ilustración 38 Diagrama de comunicación directa entre las distintas partes

### Conectar indirectamente el simulador con la vista web

Esta opción nace como respuesta a la anterior. Se plantea la posibilidad de que el proyecto tenga tres partes: El simulador, la vista web y una tercera parte que sirva como puente entre los dos primeros. En este caso se usaría la librería de wasm-bindgen para exponer las distintas estructuras del simulador, pero adaptadas a las limitaciones técnicas de la propia librería.

Esto soluciona el problema de la opción anterior, que nos limitaba los tipos que se podían utilizar en Rust. A cambio, añade un nivel más de complejidad, al tener que mantener los cambios sincronizados del simulador y este puente. Tras analizar la magnitud de esta opción se optó por no realizarla, pues aunque es la que más flexibilidad permite, también es la que mayor coste de mantenimiento tiene. Un simple cambio en el simulador podría complicar en exceso las cosas en el puente.

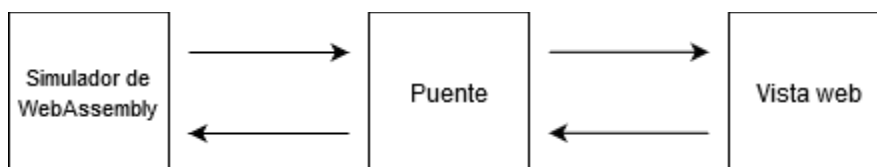


Ilustración 39 Diagrama de comunicación indirecta entre las distintas partes

### Resto de opciones

Finalmente, se implementó una tercera opción que, aunque limita bastante el diseño de la vista web, permite un mayor control del simulador dentro de la vista web. De esta opción se hablará más en detalle en la siguiente sección.

## 4.2 Estructura final del proyecto

En esta sección vamos a comentar cuales han sido las decisiones finales que se han tomado respecto al proyecto. En concreto, nos vamos a centrar en el modo de planificar el proyecto que se ha tomado al final, que difiere bastante del planteamiento inicial y también sobre cómo se han conectado las distintas partes del proyecto.

### 4.2.1 Planificación final

En el capítulo anterior mencionamos los principales defectos que se encuentra en la que podemos llamar planificación basada en funcionalidades, ya que una tarea estaba completa cuando se disponía de esta funcionalidad. Esto dio problemas a corto-medio plazo debido a la falta de consistencia entre los distintos tipos implementados de WebAssembly.

Para solucionar el problema, se optó por planificar el proyecto en base a los tipos de WebAssembly. La idea es seguir teniendo en cuenta los requisitos mencionados anteriormente, pero fijando tareas distintas a ellos. El objetivo principal de esta manera de planificar es tener una base sólida sobre la que implementar las funcionalidades, para ello primero es necesario implementar todos y cada uno de los tipos existentes en WebAssembly para evitar problemas de falta de coherencia como en el caso anterior.

En la Ilustración 40 se puede observar una tarjeta de Trello usada en el nuevo modo de planificación. Difiere bastante de las tareas mostradas anteriormente, ya que en este ejemplo se buscaba implementar todos los tipos existentes dentro de una estructura Module, lo que no lleva asociado ningún requisito.

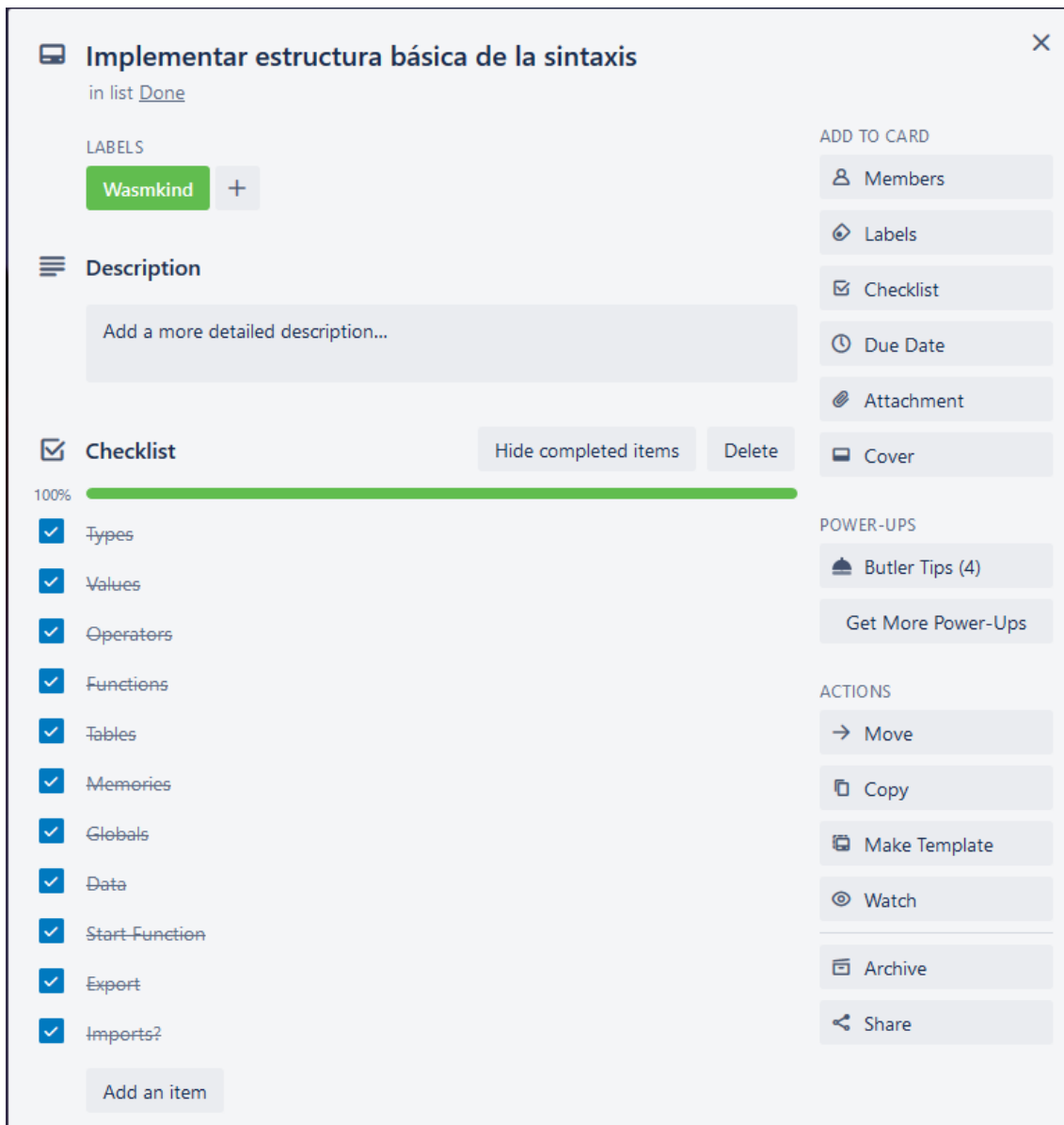


Ilustración 40 Ejemplo de tarea con la nueva planificación

#### 4.2.2 Modo de conexión de las partes del proyecto

La pregunta más importante que falta por responder es como se ha logrado que tanto el simulador escrito en Rust, como la vista web se puedan comunicar de manera que no sea tan difícil de mantener en el caso de que haya cambios en la estructura del código del simulador.

WebAssembly todavía no tiene acceso a las funciones nativas del navegador, pero se pueden llamar a funciones de Javascript mediante el uso de los *imports*. Tomando ventaja de esto, varias personas han empezado el desarrollo de varias librerías y *frameworks* que permiten manejar una vista web utilizando únicamente Rust, que posteriormente se compila a WebAssembly.

En el caso particular del proyecto, es la solución perfecta a la hora de conectar las dos partes del proyecto, pues como los dos son proyectos de Rust, es sencillo comunicar el uno con el otro. Además, gracias al fuerte tipado de Rust, se tiene la seguridad de que si algún cambio en el simulador causa incompatibilidades en la vista web el compilador no compilará y nos avisará de los errores.

Aunque no todo es bueno, el uso de estas librerías o *frameworks* plantean una serie de grandes limitaciones y desventajas que influyen a la hora de desarrollar una aplicación web compleja.

- Una de las limitaciones más notables es la **incapacidad de usar Javascript directamente** para modificar el comportamiento de la web y, aunque las librerías suelen tener implementada la mayoría de la API de los navegadores, no resultan tan intuitivas de usar.
- Otra limitación bastante evidente es que es común que usen **su propia sintaxis para definir la vista de la aplicación**, aunque por otra parte, dependiendo de la librería utilizada puede llegar a tener incluso más flexibilidad que otras soluciones existentes.

Con la mayoría de estos puntos en mente, se siguió optando por este método debido a que la vista web no es excesivamente complicada. Por lo tanto, el siguiente punto a tratar es la elección de la librería que nos permita implementar la vista web que deseamos con el menor esfuerzo posible.

Name	yew	stdweb	percy	dodrio	seed	draco	squark
License	MIT/Apache-2.0	MIT/Apache-2.0	MIT	MPL-2.0	MIT	MIT/Apache-2.0	WTFPL
Version	v0.17.3	v0.4.20	v0.0.1	v0.2.0	v0.7.0	v0.1.2	v0.7.1
Github Stars	13k	3k	1.4k	1.1k	1.2k	265	154
Contributors	188	59	17	11	37	4	4
Activity	657/year	20/year	38/year	40/year	460/year	210/year	23/year
Stable Rust	yes	yes	no	?	yes	yes	no
Base framework	wasm-bindgen (or stdweb)	-	wasm-bindgen	wasm-bindgen	wasm-bindgen	wasm-bindgen	wasm-bindgen
Virtual DOM	yes	?	yes	yes	yes	yes	yes

Ilustración 41 Comparación de los diferentes frameworks existentes para el desarrollo de una vista web, de <https://github.com/flosse/rust-web-framework-comparison#frontend-frameworks>

En la tabla mostrada en la Ilustración 41 tenemos algunos de los *frameworks* existentes ahora mismo en el ecosistema de Rust. Un detalle a tener en cuenta es que ninguno de ellos tiene una versión estable, por lo que el actualizar de una versión a otra podría hacer

que el proyecto de la vista web dejara de funcionar. Este punto sería una opción bastante importante a tener en cuenta en caso de tener un proyecto que se quisiera mantener a largo plazo, pues el coste de mantenimiento sería inviable en la mayoría de casos. Sin embargo, debido a la naturaleza de este proyecto es posible tomar este riesgo sin ningún problema, pues no es un proyecto que se va a mantener a lo largo de los años.

No vamos a comparar en detalle todos los *frameworks* mencionados en la tabla, pues no entra dentro del alcance de este documento, pero si vamos a hablar brevemente del que se ha utilizado finalmente y por qué.

Seed<sup>30</sup> es un *framework* que permite diseñar aplicaciones web de manera segura utilizando el lenguaje de programación Rust. Es una de las más populares debido a su particular forma de manejar el estado de la aplicación, basándose en el conocido lenguaje funcional Elm<sup>31</sup>. Una explicación rápida de la arquitectura de Elm es la siguiente: Existe un **modelo** o estado global de la aplicación y la **vista** web se muestra y cambia conforme a este estado; el único modo de cambiar el estado es mediante el envío de **mensajes** previamente definidos y que los recoge una función específica que **actualiza** el estado dependiendo del mensaje recibido.

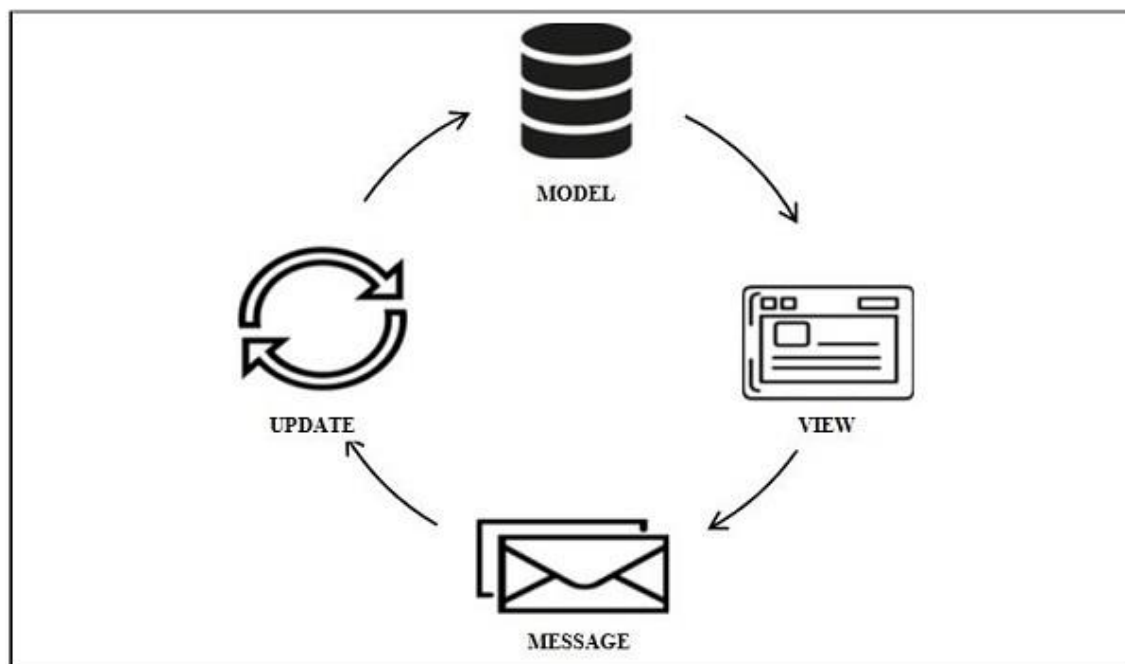


Ilustración 42 Diagrama de la arquitectura de Elm, de [https://www.tutorialspoint.com/elm/elm\\_architecture.htm](https://www.tutorialspoint.com/elm/elm_architecture.htm)

Seed adopta esta arquitectura de Elm para organizar su aplicación y, debido a esto, con una buena gestión es posible desarrollar una aplicación web eficiente y sin posibilidades de efectos secundarios no deseados, pues es imposible cambiar el estado de una forma que no haya sido definida.

<sup>30</sup> <https://seed-rs.org/>

<sup>31</sup> <https://elm-lang.org/>



```

94  #[derive(Clone)]
95  // `Msg` describes the different events you ca
96  pub(crate) enum Msg {
97      // File upload
98      FileChanged(Event),
99      FileUploaded(File),
100     FileBinary(Vec<u8>),
101     FileText(String),
102     FileError,
103     Reset,
104
105     // Simulator
106     InstanciateModule,
107     SelectExport(FuncAddr),
108     ArgChanged(usize, String, ValueType),
109     CreateInterprete,
110 }

```

Ilustración 43 Algunos mensajes utilizados en el desarrollo de la vista web

```

12  pub(crate) fn view(model: &Model) -> Node<Msg> {
13      div![
14          C!["columns"],
15          div![C!["column is-one-third"], panel_export_functions(model)],
16          div![
17              C!["column"],
18              functions_area(model),
19              div![
20                  C!["card"],
21                  show_arguments(model),
22                  button!["Execute", ev(Ev::Click, |_| Msg::CreateInterprete)]
23              ]
24          ]
25      ]
26  }

```

Ilustración 44 Ejemplo de una parte de la vista web

En la Ilustración 43 podemos ver la mayoría de mensajes utilizados en el proyecto de la vista web, es importante notar que los mensajes pueden llevar distintos tipos de datos. Por otro lado, en la Ilustración 44 tenemos un ejemplo de una parte de la vista web implementada usando Seed.

Una ventaja muy grande que Seed respecto a las otras librerías es la **flexibilidad** con la que te permite componer distintas vistas como si fueran funciones, por ejemplo, en las líneas 18 y 21 se llaman a funciones que devolverán otras vistas, permitiendo modularizar con facilidad el código. Además, en la línea 22 se define un botón y se aprecia que al interactuar con él mediante el ratón del ordenador lanzará un mensaje del tipo *CreateInterprete*. Este mensaje lo recogerá una función en otra parte del código y actualizará el estado de la manera definida.

## 4.3 Fases de desarrollo

En esta sección vamos a exponer en detalle las distintas fases de desarrollo por las que ha pasado el proyecto, así como explicar varios detalles de implementación que merecen la pena ser mencionados. El proyecto puede ser dividido en varias fases bien diferenciadas:

1. Estructuras de un módulo
2. *Parsing* de un fichero de WebAssembly
3. Estructuras de WebAssembly en tiempo de ejecución
4. Implementación del simulador
5. Desarrollo del entorno de pruebas
6. Conexión entre las distintas partes del proyecto.

### 4.3.1 Estructuras de un módulo

Como se ha mencionado anteriormente, siguiendo la planificación anteriormente mencionada el primer paso fundamental de todo el proyecto era tener una base sólida con todos los tipos existentes en WebAssembly. Para ello, se implementaron las estructuras de WebAssembly de la forma más parecida posible a como aparecen en la especificación, como vimos en la Ilustración 31.

```
119  #[derive(Debug, Clone)]
120  pub struct Table {
121      pub limits: Limits,
122      pub elem: ElemType,
123  }
124  #[derive(Debug, Clone, PartialEq)]
125  pub struct Global {
126      pub mutable: bool,
127      pub valtype: ValueType,
128  }
129
130  #[derive(Debug, Clone)]
131  pub enum Extern {
132      Func(FuncType),
133      Table(Table),
134      Memory(Memory),
135      Global(Global),
136  }
```

*tabletype ::= limits elemtype*

*globaltype ::= mut valtype*  
*mut ::= const | var*

*externtype ::= func functype | table tabletype | mem memtype | global globaltype*

Ilustración 45 Comparación de estructuras de WebAssembly entre implementación y especificación

En la Ilustración 45 se puede observar más ejemplos de lo intuitivo que resulta traducir el lenguaje formal de la especificación de WebAssembly a una estructura de código en Rust.

Una decisión de diseño importante que se tomó en cuenta en esta sección es en relación de cómo se implementaron las distintas instrucciones existentes en WebAssembly. La primera opción y la más sencilla aparentemente es tener una estructura que tenga todas y cada una de las instrucciones existentes en WebAssembly. Sin embargo, debido a su control de flujo y la repetición constante de algunas instrucciones muy similares, se decidió agruparlas en distintos tipos.

```

17     Block(BlockType, Vec<Operator>),
18     Loop(BlockType, Vec<Operator>),
19     End,
20     If(BlockType, Vec<Operator>, Vec<Operator>),
21     Br(Labelidx),
22     BrIf(Labelidx),
23     BrTable(Vec<Labelidx>, Labelidx),
24     Return,
25     Call(Funcidx),
26     CallIndirect(Typeidx),

```

Ilustración 46 Parte de la estructura con las operaciones que muestra las encargadas del control de flujo

En la Ilustración 46 se puede ver una decisión de diseño que facilitó enormemente la implementación del control de flujo en el simulador, tema que veremos en próximas secciones. En las operaciones de bloques (*Block*, *Loop* e *If*) tiene un tipo *BlockType* que no es más que un *Function Type* ligeramente modificado, y por otro lado nos encontramos que tiene un vector de instrucciones que representa las instrucciones que se encuentra dentro de ese bloque. Este simple detalle **facilita enormemente el manejo del control de flujo**, especialmente cuando hay varios **bloques anidados**. Si se hubiera implementado de otra manera habría dificultado el manejo de la situación anteriormente mencionada, pues tendría que haber una lógica específica para encontrar el final de un bloque específico.

```

45     // Numeric instructions
46     Const(Value),
47     IUnary(Int, IUnOp),
48     IBinary(Int, IBinOp),
49
50     FUnary(Float, FUnOp),
51     FBinary(Float, FBinOp),
52
98     #[derive(Debug, Clone)]
99     pub enum FUnOp {
100         Abs,
101         Neg,
102         Sqrt,
103         Ceil,
104         Floor,
105         Trunc,
106         Nearest,
107     }

```

*funop* ::= abs | neg | sqrt | ceil | floor | trunc | nearest

Ilustración 47 Ejemplo de implementación de instrucciones numéricas

Respecto al resto de instrucciones, se separaron en varios tipos o grupos, como lo hace la especificación de WebAssembly. En la Ilustración 47 podemos ver cómo se han implementado los diferentes tipos de instrucciones numéricas y específicamente un ejemplo de cómo se han implementado las instrucciones para el tipo numérico *float* con un solo argumento. La ventaja de esta de manera de organizar las instrucciones es que luego a la hora de implementar su comportamiento para el simulador es más complicado cometer algún fallo debido a la clara separación entre distintas instrucciones y sus tipos. En la Ilustración 48 podemos observar lo claro que queda el código a la hora de implementar las instrucciones para el ejemplo de la Ilustración 47.

```

292 fn unary_internal<T>(input: T, unary_op: &FUnaryOp) -> T
293 where
294     T: FloatOps<T>,
295 {
296     match unary_op {
297         FUnaryOp::Abs => T::abs(input),
298         FUnaryOp::Neg => T::neg(input),
299         FUnaryOp::Sqrt => T::sqrt(input),
300         FUnaryOp::Ceil => T::ceil(input),
301         FUnaryOp::Floor => T::floor(input),
302         FUnaryOp::Trunc => T::trunc(input),
303         FUnaryOp::Nearest => T::nearest(input),
304     }
305 }

```

Ilustración 48 Implementación de las operaciones unarias de punto flotante en el simulador

Obviando ciertos detalles de implementación, podemos asegurar que el tipo `T` de la función será siempre de los tipos `f32` o `f64`. El funcionamiento de esta función es simple, dependiendo del valor del parámetro `unary_op`, que se controla con el operador `match` de Rust, aplicará una función u otra al valor de entrada (parámetro `input`).

Todo el resto de instrucciones tienen una estructura bastante similar a los ejemplos mostrados, de manera que se obviará la explicación de cada una de ellas.

#### 4.3.2 Conversión de un fichero de WebAssembly

Tras haber implementado todas las estructuras necesarias para formar la estructura de un módulo dentro del proyecto, el siguiente paso es poder crear las estructuras dado un fichero de WebAssembly. En secciones anteriores se mencionó el uso de una librería de Rust que facilitaba enormemente esta tarea.

Esta librería tiene su propio método para leer e interpretar un fichero de WebAssembly binario (Esto es importante, pues la librería solo lee este tipo de archivos). La principal ventaja de esta librería respecto a otras es que no guarda ninguna estructura en memoria, al recorriendo el fichero va devolviendo ciertos tipos y estructuras que una vez se pasa al siguiente elemento se descarta completamente. Los tipos que devuelve son las estructuras típicas de WebAssembly que se implementaron en la sección anterior.

Por lo tanto, el mayor trabajo en esta fase del proyecto es implementar la lógica que nos permita transformar las estructuras de WebAssembly propias de la librería a las estructuras propias del proyecto. En algunos casos esta transformación es algo trivial (Véase la Ilustración 48), pues las estructuras son idénticas en ambos lugares, pero en otros casos las estructuras difieren bastante y es necesaria una lógica más complicada.

```

70 // Export type
71 impl From<ExtExport<'_>> for Export {
72     fn from(val: ExtExport) -> Self {
73         let desc = match val.kind {
74             ExternalKind::Function => ExportDesc::Func(val.index),
75             ExternalKind::Table => ExportDesc::Table(val.index),
76             ExternalKind::Memory => ExportDesc::Mem(val.index),
77             ExternalKind::Global => ExportDesc::Global(val.index),
78         };
79
80         Export {
81             name: String::from(val.field),
82             desc,
83         }
84     }
85 }

```

Ilustración 49 Ejemplo de transformación del tipo *Export*

En especial, una de las estructuras que más difieren son el cómo se representan las instrucciones, ya que Wasmparser un planteamiento diferente al que se ha tomado en el proyecto. En concreto, apuesta por tener una única estructura con todas las instrucciones posibles, de manera que la tarea de convertir las instrucciones desde la librería al proyecto fue bastante laboriosa, pues hubo que revisarla varias veces debido a que se encontraron errores.

```

200 // I32
201 ExtOperator::I32Eqz => Operator::Test(Int::I32, ITestOp::Eqz),
202 ExtOperator::I32Eq => Operator::IRel(Int::I32, IRelOp::Eq),
203 ExtOperator::I32Ne => Operator::IRel(Int::I32, IRelOp::Ne),
204 ExtOperator::I32LtS => Operator::IRel(Int::I32, IRelOp::LtS),
205 ExtOperator::I32LtU => Operator::IRel(Int::I32, IRelOp::LtU),
206 ExtOperator::I32GtS => Operator::IRel(Int::I32, IRelOp::GtS),
207 ExtOperator::I32GtU => Operator::IRel(Int::I32, IRelOp::GtU),
208 ExtOperator::I32LeS => Operator::IRel(Int::I32, IRelOp::LeS),
209 ExtOperator::I32LeU => Operator::IRel(Int::I32, IRelOp::LeU),
210 ExtOperator::I32GeS => Operator::IRel(Int::I32, IRelOp::GeS),
211 ExtOperator::I32GeU => Operator::IRel(Int::I32, IRelOp::GeU),

```

Ilustración 50 Fragmento de código donde se muestran la conversión de las instrucciones del tipo I32

En la Ilustración 50 se puede observar una pequeña parte de la función encargada de la conversión de las instrucciones. En concreto, en la imagen se ve las conversiones que se hacen para parte de las instrucciones del tipo I32. Aparte de esto, una de las partes más complicadas fue lograr el comportamiento esperado en la conversión de las instrucciones de control de flujo debido al diseño que se implementó para ellas en el proyecto.

Para terminar con esta sección, antes se ha mencionado que el proyecto tiene soporte para el formato textual también y se ha aclarado que la librería Wasmparser solo funciona con ficheros en formato binario. La solución fue utilizar otra librería que tomaba un fichero de WebAssembly en formato textual, lo convertía a formato binario y entonces ya podemos usar Wasmparser sin ningún tipo de problemas.

### 4.3.3 Estructuras de WebAssembly en tiempo de ejecución

Tras tener todas las estructuras necesarias para poder tener un módulo en memoria y tener la posibilidad de leer ficheros de WebAssembly, el siguiente paso es conseguir tener un módulo inicializado, y para ello es necesario implementar las estructuras de WebAssembly en tiempo de ejecución.

En esta sección la mayoría de la implementación ha sido una tarea bastante sencilla, al igual que en el caso anterior. Como se observa en la Ilustración 51, se vuelve a repetir el caso en el que el paso de traducir la especificación a código es una tarea trivial y que no requiere de ningún esfuerzo gracias. Esto es en parte gracias a la implementación progresiva de los tipos, empezando por los más simples y avanzando al más complicado.

```
moduleinst ::= { types      functype*,      217  #[derive(Debug, Clone, Default)]
                funcaddrs  funcaddr*,      218  pub struct ModuleInst {
                tableaddrs tableaddr*,      219      pub types: Vec<types::FuncType>,
                memaddrs   memaddr*,      220      pub funcaddrs: Vec<FuncAddr>,
                globaladdrs globaladdr*,    221      pub tableaddrs: Vec<TableAddr>,
                exports    exportinst* }    222      pub memaddrs: Vec<MemAddr>,
                                          223      pub globaladdrs: Vec<GlobalAddr>,
                                          224      pub exports: Vec<ExportInst>,
                                          225  }
```

Ilustración 51 Comparación de la instancia de un módulo entre especificación e implementación

La única parte complicada y en la que había que tener total seguridad de que estaba implementada correctamente era en el algoritmo de inicialización de un módulo<sup>32</sup>. Pues por características del lenguaje era imposible traducir directamente el algoritmo descrito en la implementación a código. No se probó directamente, pero en una fase de pruebas posterior se pudo asegurar que estaba bien implementado todo.

### 4.3.4 Implementación del simulador

Una vez se tienen todas las estructuras necesarias de WebAssembly ya lo único que queda es implementar la lógica que interactúe con todas estas estructuras y simule un entorno de WebAssembly. Cabe decir que esta parte ha sido la que ha consumido la mayor parte del tiempo del proyecto, pues hubo varios intentos de implementación fallidos y con la incorporación de la fase de pruebas se descubrieron algunos errores complicados de arreglar con la organización actual del código.

El primer objetivo del desarrollo era tener una estructura que permitiera ejecutar instrucciones sencillas, como las numéricas, de manera que se organizó el código para que se pudiera realizar esto de la manera más sencilla. Así pues, se continuó en la implementación del resto de instrucciones, algunas más complicadas que otras dejando las instrucciones de control de flujo las últimas. Esto último fue un error bastante grave, pues el control de flujo es una de las partes más complejas de WebAssembly, como hemos visto anteriormente, lo que ocasionó una total reestructuración del código del simulador, quedando intacta la implementación ya hecha de la mayoría de las instrucciones.

<sup>32</sup> <https://webassembly.github.io/spec/core/exec/modules.html#instantiation>

Una de las principales razones por las que hizo falta la reestructuración del código fue su estructura actual, como se aprecia en la Ilustración 52. En un principio, las instrucciones se controlaban todas agrupándolas todas en un único vector. Los *frames* de las distintas funciones tenían su propio vector dentro de la estructura principal del simulador, pero a su vez las estructuras que tenían la información del flujo de control (La estructura *Label*) estaba separado del resto de estructuras.

```

96  #[derive(Debug, Clone)]
97  pub struct Interpreter {
98      pub stack: StackValue,
99      pub frames: Vec<StackFrame>,
100     pub labels: Vec<Label>,
101     pub instructions: Vec<Operator>,
102     pub last_operator_state: IntResult,
103 }

74  #[derive(Debug, Clone)]
75  pub struct StackFrame {
76      pub(crate) module: Rc<ModuleInst>, // Reference to the module
77      pub(crate) locals: Vec<Value>,
78  }

89  #[derive(Debug, Clone)]
90  pub struct Label {
91      pub idx: usize,
92      pub value_stack_begin: usize,
93      pub op_stack_begin: usize,
94  }

```

Ilustración 52 Estructura antigua del código del simulador

El principal problema de esta estructuración del código es la **incapacidad de manejar de manera simple y sin errores el control de flujo del simulador**. Sin embargo, esta solución no se aprovechaba de la estructura particular que se les dio a las instrucciones de control de flujo, pues todas las instrucciones se agregaban en una única pila o vector, no controlándose en ningún momento si se mezclaban instrucciones de distintas funciones. Las instrucciones se iban consumiendo a medida que se iban usando, lo que daba un control preciso del control de flujo. Al menos hasta que se empezó a implementar el **bloque de tipo loop**, que con este enfoque era inviable pues no había forma alguna de volver a introducir las instrucciones de este bloque en la pila.

Tras intentar diseñar varias soluciones para el problema, se llegó a la conclusión de que lo mejor era la reestructuración completa del código del simulador hacia una solución que permitiera un mayor control del flujo de las instrucciones con una mayor facilidad. Se pensó en una estructura del código que se refinó varias veces antes de implementarlo, pues era importante conseguir un buen diseño que permitiera corregir los distintos errores que salieran de las pruebas sin demasiados problemas.

```

93  #[derive(Debug, Clone)]
94  pub struct Interpreter {
95      pub stack: StackValue,
96      pub call_stack: Vec<FunctionContext>,
97  }

517 #[derive(Debug, Clone)]
518 pub struct FunctionContext {
519     pub module_ref: Rc<ModuleInst>,
520     func_addr: FuncAddr,
521     functype: FuncType,
522     pub execution_unit: ExecutionUnit,
523     pub locals: Vec<Value>,
524 }

568 #[derive(Debug, Clone)]
569 pub struct ExecutionUnit {
570     nesting_level: usize,
571     stack_operators: Vec<Vec<Operator>>,
572     stack_pc: Vec<usize>,
573     stack_value_unwind: Vec<StackUnwindVariant>,
574 }

```

Ilustración 53 Estructura final del código del simulador

En la Ilustración 53 es posible ver el notorio cambio que se produjo, siendo una de las mayores diferencias la separación de código de las ejecuciones de funciones de



WebAssembly, que se guarda en la estructura *FunctionContext*. Esta estructura alberga todos los datos necesarios para la ejecución y el manejo de una función, tanto para el manejo de los registros locales como para la ejecución de código. La siguiente estructura a destacar es *ExecutionUnit*, que es el grueso principal que permite el manejo del control de flujo del entorno de WebAssembly y que, además, se beneficia enormemente de la implementación de las instrucciones de control de flujo.

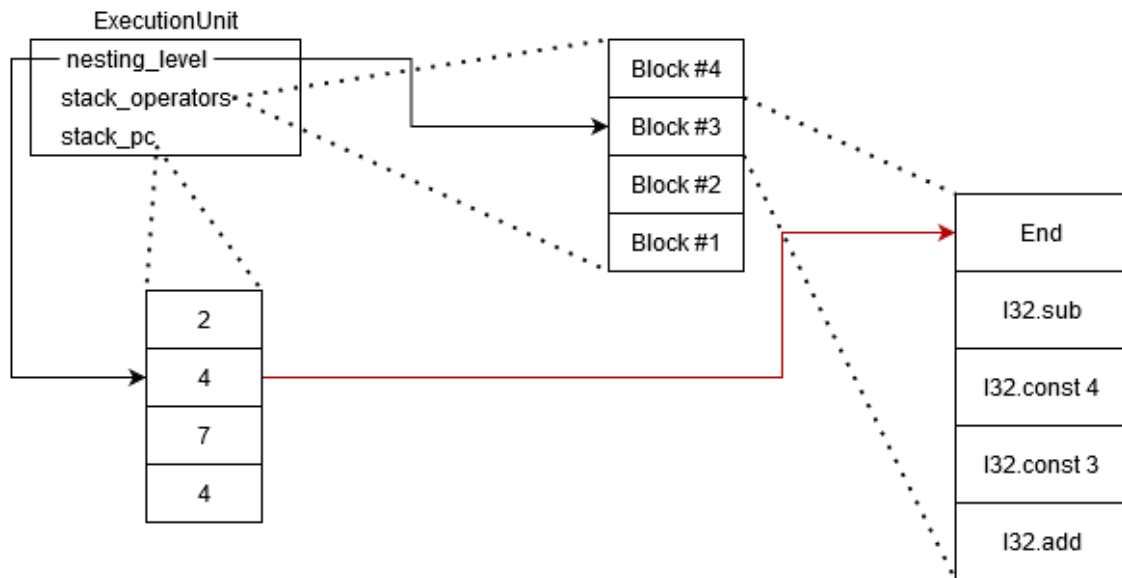


Ilustración 54 Diagrama de funcionamiento de la estructura *ExecutionUnit*

El funcionamiento de la estructura *ExecutionUnit* aunque a primera vista puede resultar compleja, su funcionamiento no es complicado como se puede apreciar en la Ilustración 54. Dentro de la estructura se encuentra el atributo **nesting\_level** que es de un tipo numérico<sup>33</sup> que representa dos cosas:

- El número de bloques anidados que existen en un momento específico, siendo su valor inicial 0.
- Como índice para los demás atributos existentes, que son todos vectores.

Luego está el atributo **stack\_operators** que es un **vector de vectores de instrucciones**. Esta peculiaridad permite que si, por ejemplo, la instrucción ejecutada es de un nuevo bloque (*Block*, *Loop* o *If/Else*), el código de este bloque esté totalmente separado del resto. Esto permite eliminar el bloque en cuestión si se ejecuta una instrucción de salto sin necesidad de lógica adicional.

El siguiente atributo es **stack\_pc**, que no es más que un **vector de índices** que indican la **posición de la instrucción que se va a ejecutar a continuación** en un bloque específico. El bloque y el índice queda determinado por el valor del atributo `nesting_level`.

<sup>33</sup> El tipo *usize* en Rust representa un puntero hacia una dirección de memoria. Tiene su propio tipo debido a que el tamaño en bits de este cambia dependiendo de la plataforma en la que se compile el código.



```

549 #[derive(Debug, Clone)]
550 pub enum StackUnwindVariant {
551     Block {
552         stack_init: usize,
553         results_size: usize,
554     },
555     Loop {
556         stack_init: usize,
557         params_size: usize,
558         results_size: usize,
559     },
560 }

```

Ilustración 55 Implementación de la estructura StackUnwindVariant

Para terminar, el atributo **stack\_value\_unwind** es un vector de una estructura de dato que guarda información necesaria para controlar el estado de la pila del entorno en el caso que se ejecuta una instrucción de salto. Los elementos son del tipo *StackUnwindVariant* y tiene guardado tanto la posición de la pila en el momento en el que se entra en el bloque (Se resta a ese valor el número de parámetros que tiene el bloque) y los tamaños de los argumentos y resultados del *Function Type* asociado a ese bloque. Esto se usa para calcular y desechar los elementos de la pila no necesarios en caso de que ocurra una instrucción de salto.

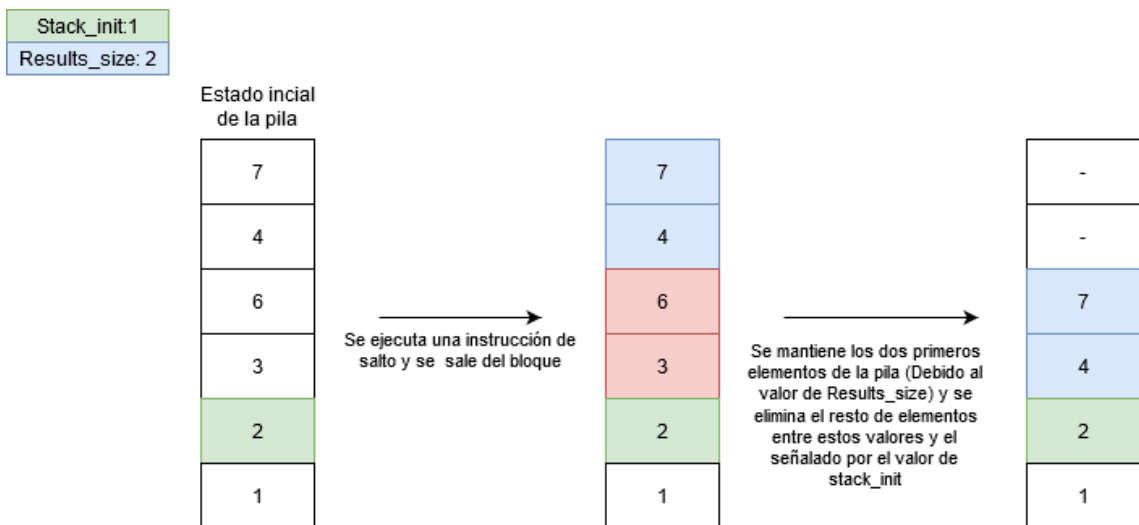


Ilustración 56 Ejemplo de estado de pila durante la ejecución de una instrucción de salto en un bloque de tipo Block

En la Ilustración 56 se puede ver un ejemplo en el que se usa la estructura *StackUnwindVariant*. Este es uno de los distintos comportamientos que está implementado en la aplicación, pues este variará dependiendo de varios factores como el tipo de bloque y la instrucción de salto ejecutada.

Para concluir con esta sección, es importante mencionar que, aunque esta solución permite un razonamiento simple de lo que hace el código, tiene un coste computacional mayor al añadir varias estructuras y varios niveles de indirección. Sin embargo, el objetivo de este proyecto no es implementar un simulador lo más eficiente posible, sino uno que permita entender la estructura de WebAssembly. Por lo tanto, se cree que este

diseño de simulador permite entender de forma sencilla el funcionamiento del control de flujo de WebAssembly a nivel de código.

#### 4.3.5 Conexión entre las distintas partes del proyecto

Tras tener listo el simulador y funcionando sin errores gracias a la fase de pruebas (A la que se le dedicará un capítulo entero próximamente), lo único restante es desarrollar la vista web que permita interactuar con este simulador con una interfaz sencilla. Para ello como se explicó en secciones anteriores se ha usado un framework de Rust llamado Seed.

Durante el desarrollo de la vista web, se encontraron varios problemas que ocasionaron que el desarrollo de esta fuera más lento de lo esperado, en concreto, a la hora de mostrar las instrucciones bien formateadas, ya que por defecto se mostraba la estructura del código de Rust. Por lo tanto, hubo que definir una cadena de caracteres a todas y cada una de las instrucciones implementadas dentro de la vista web.

Para terminar, se tenía pensado añadir bastantes más funcionalidades a la vista web, tales como poder ver la memoria o la estructura de un módulo de forma sencilla. Debido a los contratiempos ocasionados por el desarrollo del simulador, se tiene solo la funcionalidad básica que permite subir un fichero de WebAssembly, ya sea binario o textual, y poder seleccionar una función y ejecutarla.

#### 4.4 Limitaciones del simulador

En esta última sección vamos a hablar de las limitaciones que tiene el simulador de WebAssembly, ya que por prioridades se han decidido dejar fuera.

Una de las primeras limitaciones que se encuentran en el simulador es la incapacidad de poder usar ficheros de WebAssembly en los que haga falta el uso de los *imports*. Si el proyecto hubiera sido enfocado a un simulador de WebAssembly que se ejecutara en un sistema operativo no habría habido problemas en implementarlo. Sin embargo, al tener que ejecutarse en un navegador web, añade varias capas de complejidad que resultarían inabarcables en el tiempo total que se dispone.

Siguiendo la primera limitación, la segunda es precisamente la **incapacidad de ejecutar funciones de fuera del entorno de WebAssembly**. Estaba en mente la idea de que en la vista web se le pudieran pasar funciones específicas de Javascript, sin embargo, aunque era abaricable el problema, los navegadores web tienen muchas limitaciones para que no se pueda ejecutar código arbitrario, por lo que al final se acabó descartando esta idea.

Para terminar, el mayor factor limitante de este proyecto, aunque con ejemplos simples no se va a percibir, es la gran pérdida de rendimiento que supone ejecutar este simulador en el navegador. Recordemos que el simulador se ha compilado a WebAssembly, y es el propio entorno de WebAssembly del navegador el que está ejecutándolo. Aunque una ventaja de esto es la posibilidad de ejecutar cualquier fichero

de WebAssembly con suma facilidad (Teniendo en cuenta las limitaciones anteriormente mencionadas).



# 5

## Ejecución de pruebas

Una de las cosas más importantes para comprobar el correcto funcionamiento de un software es tener un conjunto de pruebas que lo corroboren. En este capítulo vamos a hablar de como se ha podido probar una parte del simulador de WebAssembly y el método de implementación que se ha llevado a cabo para ello.

### 5.1 Modo de ejecución de pruebas

Para comprobar el correcto funcionamiento de un software siempre es recomendable tener un conjunto de pruebas que permitan tener la certeza de que todo funciona correctamente. Afortunadamente, el equipo de desarrollo de WebAssembly tiene un extenso conjunto de pruebas<sup>34</sup> que abarca todo el conjunto de funcionalidades del que se compone WebAssembly.

Las pruebas vienen contenidas en ficheros del tipo wast, siendo un formato textual idéntico al que ya conocemos pero con unas funcionalidades extras que facilitan el proceso de ejecutar las pruebas. En concreto, viene con una serie de asertos que permiten comprobar el estado de las distintas partes de un entorno de WebAssembly, como puede ser la pila o un espacio de memoria.

---

<sup>34</sup> <https://github.com/WebAssembly/spec/tree/master/test/core>

```

assertion:
  ( assert_return <action> <result>* )      ;; assert action has expected results
  ( assert_trap <action> <failure> )        ;; assert action traps with given failure string
  ( assert_exhaustion <action> <failure> )  ;; assert action exhausts system resources
  ( assert_malformed <module> <failure> )   ;; assert module cannot be decoded with given failure string
  ( assert_invalid <module> <failure> )     ;; assert module is invalid with given failure string
  ( assert_unlinkable <module> <failure> )  ;; assert module fails to link
  ( assert_trap <module> <failure> )       ;; assert module traps on instantiation

```

Ilustración 57 Distintos asertos disponibles en el formato wast, de <https://github.com/WebAssembly/spec/blob/master/interpreter/README.md#scripts>

En la Ilustración 57 se puede observar una lista de todos los asertos disponibles. Todos tienen su utilidad, sin embargo, para el proyecto se ha decidido mantener el nivel de complejidad al mínimo debido a problemas que se comentarán más adelante. Además, en esa lista existe un par de asertos que permiten comprobar si un módulo está bien formado o es válido, no tendremos en cuenta este tipo de asertos a la hora de realizar pruebas pues este tipo de pruebas ya están cubiertas en las librerías que hemos usado.

## 5.2 Alcance de las pruebas

El equipo de WebAssembly nos provee de un **gran número de pruebas** que **nos garantiza el correcto funcionamiento de un entorno de WebAssembly** si las completa todas satisfactoriamente. Sin embargo, el lograr este hecho es un trabajo que solo puede ser logrado empleando mucho tiempo y esfuerzo o con la ayuda de diferentes personas. Es tal la complejidad de pasar satisfactoriamente estas pruebas que muchos entornos de WebAssembly desarrollados por la comunidad no las pasaron enteramente hasta que el desarrollo de ese proyecto llevaba meses y años de intenso trabajo por parte de un grupo de personas.

Debido a esto último y a una serie de problemas que se van a comentar a continuación, se ha elegido limitar el alcance de las pruebas ejecutadas en el simulador a un grupo reducido pero que permite tener cierta seguridad respecto a su funcionamiento. En concreto, todas las pruebas que tengan relación con las **instrucciones de control de flujo** se han añadido a este limitado conjunto de pruebas, pues es en gran parte lo que ha ayudado a diseñar y refinar la implementación del control de flujo de WebAssembly en el simulador.

Tras ello, en el limitado conjunto de pruebas entran también las que tengan relación con los tipos numéricos *i32* y *i64*. No se incluyen los tipos numéricos *f32* y *f64* debido a un pequeño detalle que no se tuvo en cuenta en el diseño del proyecto y que ahora sería extremadamente costoso cambiarlo. Esto es que, estos tipos numéricos aparte de tener un valor numérico pueden tener ciertos **valores especiales** que representa cierta información. En concreto, existen dos valores que representan si un número es **infinito** (Tanto positivo como negativo) y los valores que representan que esa secuencia de bits no es un número válido (**NaN** o Not a Number), también positivo y negativo.

El fallo de implementación fue que **se utilizaron los tipos f32 y f64 de Rust** para representar este tipo específico en el entorno de WebAssembly. Mientras que los tipos de Rust contiene un valor específico para representar el valor concreto NaN, no contienen ninguno para el valor -NaN. Este último valor está presente en la gran mayoría

de pruebas relacionadas con este tipo, por lo que se decidió no incluirlas en el conjunto de pruebas del proyecto.

El otro fallo grave de implementación no tiene que ver con el simulador, sino con la infraestructura que se ha desarrollado para la ejecución de estas pruebas. Se comentará más en detalle en la próxima sección, pero la infraestructura está diseñada de manera que cada aserto se ejecuta de manera independiente, de manera que para cada uno se creará un entorno de WebAssembly completamente aislado del resto y ahí se comprobará el resultado del aserto.

```
36 (module
37   (memory 0)
38   (func (export "grow") (param i32) (result i32) (memory.grow (local.get 0)))
39 )
40
41 (assert_return (invoke "grow" (i32.const 0)) (i32.const 0))
42 (assert_return (invoke "grow" (i32.const 1)) (i32.const 0))
43 (assert_return (invoke "grow" (i32.const 0)) (i32.const 1))
44 (assert_return (invoke "grow" (i32.const 2)) (i32.const 1))
45 (assert_return (invoke "grow" (i32.const 800)) (i32.const 3))
46 (assert_return (invoke "grow" (i32.const 0x10000)) (i32.const -1))
47 (assert_return (invoke "grow" (i32.const 64736)) (i32.const -1))
48 (assert_return (invoke "grow" (i32.const 1)) (i32.const 803))
```

Ilustración 58 Extracto del fichero de pruebas `memory_grow.wast`, de [https://github.com/WebAssembly/spec/blob/master/test/core/memory\\_grow.wast](https://github.com/WebAssembly/spec/blob/master/test/core/memory_grow.wast)

Por las razones anteriormente mencionadas y por el diseño de la infraestructura de las pruebas, resulta **imposible** incluir en el conjunto de pruebas del proyecto las relacionadas específicamente con el manejo de memoria. Como podemos observar en la Ilustración 58, en ese módulo solo se usa la instrucción `grow`, que acepta un parámetro y **devuelve el tamaño de la memoria** (en bloques de WebAssembly) antes de la ejecución de la instrucción (`O -1` en caso de que haya algún fallo). Si se presta atención al flujo que siguen las pruebas, se espera que todos los asertos se utilicen bajo un mismo entorno de WebAssembly.

Sin embargo, esto no quiere decir que las instrucciones de manejo de memoria se queden sin ningún tipo de verificación del funcionamiento, pues entre el conjunto de pruebas elegidos, indirectamente se prueba la gran mayoría de instrucciones de este tipo de instrucciones.

### 5.3 Implementación de las pruebas

Habiendo visto ya el formato de los ficheros de prueba, la siguiente cuestión a resolver es el cómo se van a pasar este tipo de pruebas al simulador. El formato tan concreto de estos ficheros hace que no se pueda pasar las pruebas con facilidad. Para solucionar este problema se ha investigado varios proyectos similares y se ha encontrado que hay dos principales formas de ejecutar este tipo de pruebas en un simulador o interprete de WebAssembly.

Una primera solución es trasladar el esfuerzo de la ejecución de pruebas fuera del propio proyecto. Un ejemplo de esta solución ha sido usada por los propios desarrolladores de WebAssembly<sup>35</sup> y es tan simple como adaptar el proyecto de manera que el ejecutable resultante de compilar permita manejar la ejecución especificando ciertos comandos en una línea de comandos. Esto permite dejar el trabajo de desglosar toda la información de las pruebas a un agente externo al proyecto (Por ejemplo, otro programa o script), concentrándose estrictamente en el resultado de ejecutar el simulador.

Opuesta a la anterior, la segunda solución consiste en incluir lógica que permita manejar el formato y la ejecución de los ficheros de prueba. Una ventaja clara respecto a la anterior solución es el evitar incoherencias en las pruebas que manejen datos numéricos de punto flotante (Como los tipos *f32* y *f64*). Sin embargo, la principal desventaja es el manejo manual del tipo de formato de ficheros, pues aunque no es complicado, lleva tiempo conseguir una lógica que funcione bien.

Para la infraestructura de pruebas de este proyecto se eligió la segunda opción, pues hay librerías en Rust que permiten el manejo de este tipo de ficheros, lo que suple la desventaja anteriormente mencionada.

No se va a entrar en detalles de cómo se ha implementado en Rust, pues para llegar a implementar esta infraestructura ha habido que usar funcionalidades avanzadas de Rust que no entran dentro del ámbito de este documento. La infraestructura a nivel de funcionamiento es sencilla. Tenemos los ficheros con la extensión *.wast* se encuentran en una carpeta localizada en el proyecto del simulador, de manera que a la hora de compilar se generan archivos de pruebas propios de Rust<sup>36</sup>. Con los ficheros generados ya es posible ejecutar las pruebas sin demasiado problemas gracias a una herramienta de Rust que permite la ejecución de este tipo de ficheros.

Por último, como se ha mencionado en secciones anteriores, solo se ha optado por implementar el aserto del tipo *assert\_return*. Esta decisión se ha tomado tras revisar un gran número de pruebas y ver que los otros tipos de asertos no añadirían nada al conjunto limitado de pruebas de la aplicación.

---

<sup>35</sup> <https://github.com/WebAssembly/spec/tree/master/test/core>

<sup>36</sup> [https://doc.rust-lang.org/rust-by-example/testing/unit\\_testing.html](https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html)



# 6

## Conclusiones y líneas futuras

### 6.1 Conclusiones

A lo largo de este documento se ha visto el proceso que ha llevado a la realización de un simulador de WebAssembly que, tras algunos problemas de considerable magnitud y otros percances, se ha conseguido desarrollar una versión limitada pero funcional.

El proyecto empezó como una idea para ver las capacidades de WebAssembly a la hora de ejecutar programas más complejos en un navegador web. Además de haber visto que es posible, aunque con ciertas limitaciones que seguramente se suplan en un futuro, se ha aprendido bastante sobre el funcionamiento interno de un entorno de WebAssembly.

Además, como resultado de todo este desarrollo, se ha conseguido un simulador o intérprete de WebAssembly capaz de compilarse a un ejecutable *standalone*, en el que se exponen algunas estructuras internas. Debido a esto es capaz de interactuar con la vista web y mostrar información del comportamiento interno del simulador, cosa que ningún otro intérprete o simulador de WebAssembly hace, de manera que permita incluso depurar el funcionamiento de algunos programas de WebAssembly. Por falta de tiempo la vista es sencilla, pero se podría desarrollar una vista web con una infinidad de funcionalidades útiles gracias a esto.

En el documento se ha visto en cierta profundidad el funcionamiento formal de este tipo de entornos y, además, se ha relatado el desarrollo del proyecto con detalles suficientes que permiten tener la certeza de las dificultades que han habido durante el desarrollo

de este. Si bien es cierto que a lo largo del proyecto no se ha mencionado en exceso sobre técnicas de desarrollo de software, el tener un conjunto de funcionalidades bastante reducida ha permitido evitar un descontrol en temas de la organización.

Para terminar, aunque WebAssembly no es extremadamente usado a día de hoy a nivel comercial por sus serías limitaciones, tengo la certeza de que conforme vaya mejorando y se vayan aceptando diferentes mejoras propuestas o *proposals*<sup>37</sup>, se irá abriendo cada vez más su mercado laboral.

## 6.2 Líneas futuras

Como se ha mencionado ya numerosas veces a lo largo de este documento, existen varias funcionalidades que se han tenido que dejar fuera por varios contratiempos ya mencionados. Debido a ello existen varios caminos que se podrían seguir a la hora de ampliar el proyecto o elegir un tema relacionado.

El primero de ellos es sencillamente **ampliar extensamente la funcionalidad de la vista web**, pues tal y como está ahora tiene funcionalidades básicas. Algunas mejoras que se podrían hacerle son:

- Añadir más funcionalidades que permitan interactuar con el entorno de WebAssembly simulado, como una vista que permita ver la memoria.
- Más funcionalidades que permitan depurar con certeza la ejecución de una función de WebAssembly.
- Una vista aparte que permita ver toda la información de un módulo de WebAssembly de una manera fácil y cómoda.
- Permitir la inclusión de *imports* a la hora de inicializar un módulo, de manera que se pueda definir el comportamiento de ciertas funciones directamente desde el navegador web. Esto último requeriría utilizar algunas funciones actualmente experimentales en los navegadores web.

Otro tema interesante sería **comparar el rendimiento de varias librerías y frameworks que utilicen Javascript y WebAssembly**, pero solamente cuando se acepte un *proposal*<sup>38</sup> específico, que permitirá suplir una de las carencias más notorias de WebAssembly. En concreto, esto permitiría a WebAssembly utilizar funciones nativas del navegador a pleno rendimiento y no como funciona a fecha de la escritura de este documento. Actualmente la única forma en que WebAssembly puede contactar con el navegador es importando funciones de Javascript a un módulo, que tiene un sobrecoste computacional notable en aplicaciones de alto rendimiento.

Para terminar, un tercer tema bastante decente y que demostraría la potencia de WebAssembly sería **obtener una aplicación medianamente compleja** escrita preferiblemente en C/C++ o Rust (Debido al gran soporte que tienen con WebAssembly) y **adaptarla para que se pudiera ejecutar directamente en el navegador**. Existen varias

---

<sup>37</sup> <https://github.com/WebAssembly/proposals>

<sup>38</sup> <https://github.com/WebAssembly/interface-types/blob/master/proposals/interface-types/Explainer.md>

empresas que han logrado esto en sus productos de gran complejidad como ya se mencionó anteriormente con la empresa Figma o Autodesk, que adaptaron su aplicación de escritorio a una aplicación web [9] gracias a esta tecnología.



# Referencias

- [1] Oracle, «Java Applet,» [En línea]. Available: <https://www.oracle.com/technetwork/java/applets-137637.html>.
- [2] «WebAssembly,» [En línea]. Available: <https://webassembly.org/>.
- [3] «Use Cases - WebAssembly,» [En línea]. Available: <https://webassembly.org/docs/use-cases/>.
- [4] E. Wallace, «WebAssembly cut Figma's load time by 3x,» 8 Junio 2017. [En línea]. Available: <https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/>.
- [5] IEEE, «754-2019 - IEEE Standard for Floating-Point Arithmetic,» 22 Julio 2019. [En línea]. Available: <https://ieeexplore.ieee.org/document/8766229>.
- [6] M. Contributors, «WebAssembly.validate(),» 7 Mayo 2020. [En línea]. Available: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/WebAssembly/validate](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/validate).
- [7] The Rust Core Team, «Rust-lang,» 15 Mayo 2015. [En línea]. Available: <https://blog.rust-lang.org/2015/05/15/Rust-1.0.html>.
- [8] A. Burch, «Using Rust in Windows,» 07 11 2019. [En línea]. Available: <https://msrc-blog.microsoft.com/2019/11/07/using-rust-in-windows/>.
- [9] L. E. Friedman, «Roundup: The AutoCAD Web App at Google I/O 2018,» 28 Mayo 2018. [En línea]. Available: <https://blogs.autodesk.com/autocad/autocad-web-app-google-io-2018/>.
- [10] «Rust,» [En línea]. Available: <https://www.rust-lang.org/>.



# Apéndice A

## Manual de instalación

Para lograr compilar y ejecutar el proyecto es necesario tener en cuenta una serie de pasos y herramientas.

### 1. Instalación de las herramientas

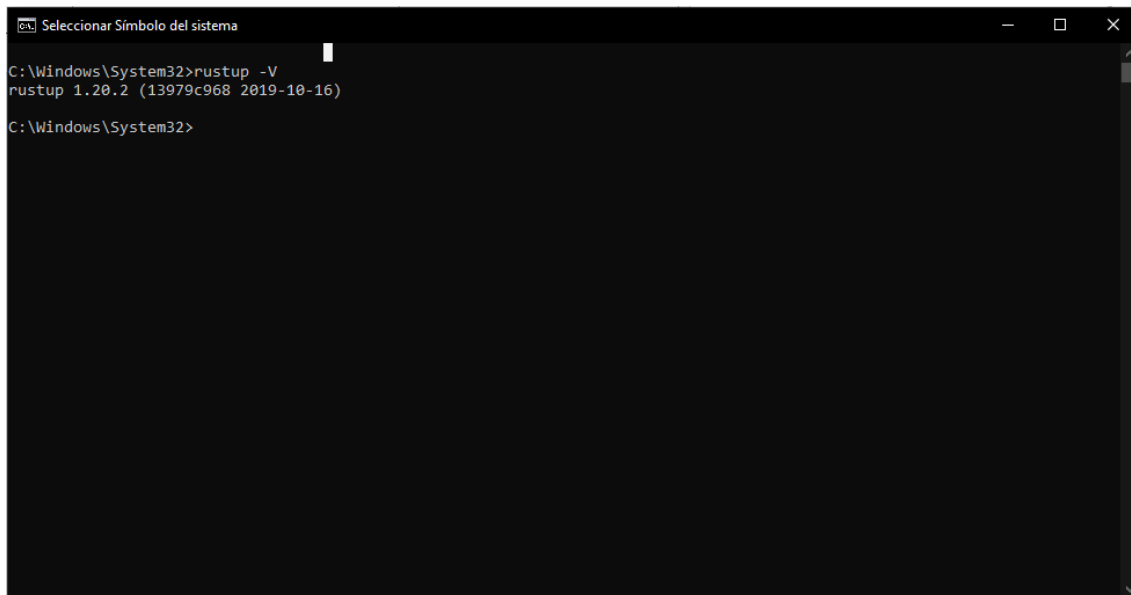
Antes de nada, es necesario tener instalado distintas herramientas relacionadas con el lenguaje de programación Rust. Entre ellas nos encontramos con Rustup, un gestor de versiones de Rust y Cargo, un gestor de proyectos.

Para instalar todas estas herramientas visite esta página: <https://www.rust-lang.org/tools/install> y siga los pasos de acuerdo al sistema operativo del equipo.

Para comprobar que todo el proceso de instalación se ha hecho correctamente, abra su terminal de preferencia de su sistema operativo y escriba este comando:

```
rustup -V
```

Y tendría que aparecer en la siguiente línea la versión de la herramienta Rustup.



```
C:\Windows\System32>rustup -V
rustup 1.20.2 (13979c968 2019-10-16)
C:\Windows\System32>
```

Ilustración 59 Salida correcta del comando "rustup -V"

A continuación necesitaremos varias herramientas específicas que instalaremos con el gestor de dependencias Cargo. Vamos a instalar Wasm-pack, que se encarga de compilar un código de Rust a un fichero binario de WebAssembly y Cargo-Make<sup>39</sup> que permite automatizar ciertas acciones a la hora de compilar.

Para ello, instalamos los siguientes comandos en una terminal:

```
cargo install wasm-pack
```

```
cargo install cargo-make
```

```
cargo install microserver
```

Este proceso de instalación puede durar un rato dependiendo de las especificaciones del equipo en el que se instala.

## 2. Preparación de las carpetas del proyecto

El TFG se encuentra dividido en dos proyectos de Rust, cuyos nombres son **wasmkind** y **wasmkind-web-viewer**.

El proyecto **wasmkind** contiene el código del simulador, mientras que el proyecto **wasmkind-web-viewer** contiene el código de la vista web. Este último tiene como **dependencia** al primero, de manera que no compilará si falta.

Ambos proyectos deben estar **alojadas en la misma carpeta**. Además, **bajo ninguna circunstancia debe cambiarse el nombre de la carpeta "wasmkind"**, pues esto hará que sea imposible el proceso de compilación.

---

<sup>39</sup> <https://github.com/sagiegurari/cargo-make>



### 3. Compilación y ejecución

Teniendo ya los proyectos perfectamente configurados y sin ningún error, el siguiente paso es compilar la vista web.

Vamos a compilar el proyecto de la vista web (wasmkind-web-viewer), para ello es necesario abrir una terminal en la carpeta de este proyecto y usar el siguiente comando:

```
cargo make build release
```

Se generará un binario de WebAssembly lo más optimizado posible, tanto en rendimiento como en el peso total del fichero. Si quisiera generar un binario que contenga información de depuración, tendría que ejecutar el siguiente comando:

```
cargo make build
```

Tras ello, para conseguir ejecutar la vista es necesario un servidor de ficheros estático, pues los navegadores web tienen serias limitaciones a la hora de cargar ficheros directamente desde el sistema de ficheros.

En pasos anteriores instalamos un servidor de ficheros estáticos sencillo de usar, para usarlo ejecutamos el siguiente comando:

```
cargo make serve
```

Con el servidor iniciado, ahora solo tienes que ir a <http://localhost:8000> y podrás acceder a la vista web.

### 4. Ejecución de pruebas

Si se desea ejecutar las diferentes pruebas que existen en el proyecto, estas se ejecutan en el proyecto del simulador, pues no tienen nada que ver con la vista web.

Abre una terminal en el proyecto de **wasmkind** y para ejecutar el suite de pruebas escribe el siguiente comando:

```
cargo test -q
```

Este proceso puede durar varios minutos dependiendo de las capacidades de su equipo y durante el proceso podrá ver algo similar a esto:

```
running 81 tests
.....
test result: ok. 81 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

running 139 tests
..... 100/139
test result: ok. 139 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

running 64 tests
.....
test result: ok. 64 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

running 103 tests
..... 100/103
test result: ok. 103 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

running 2500 tests
..... 100/2500
..... 200/2500
..... 300/2500
..... 400/2500
..... 500/2500
..... 600/2500
..... 700/2500
..... 800/2500
..... 900/2500
..... 1000/2500
..... 1100/2500
..... 1200/2500
..... 1300/2500
..... 1400/2500
..... 1500/2500
..... 1600/2500
..... 1700/2500
..... 1800/2500
..... 1900/2500
..... 2000/2500
```

Ilustración 60 Ejemplo de ejecución de las pruebas

Durante el proceso de ejecución no se ve el nombre de las diferentes pruebas debido al extenso número de pruebas similares que existen.

# Apéndice B

## Manual de Usuario de la vista web

El simulador se puede utilizar mediante una interfaz gráfica que permite manejar y visualizar su funcionamiento y estructura interna.

La vista web se puede utilizar con cualquier tipo de navegador que soporte WebAssembly, de manera que es sencillo distribuirlo y usarlo, pues no requiere de ninguna configuración previa.

Una vez accedes al dominio o dirección en la que se encuentra alojado la vista web, el usuario es recibido con una vista mínima que le indica que suba un fichero con la extensión `.wasm/.wat`.

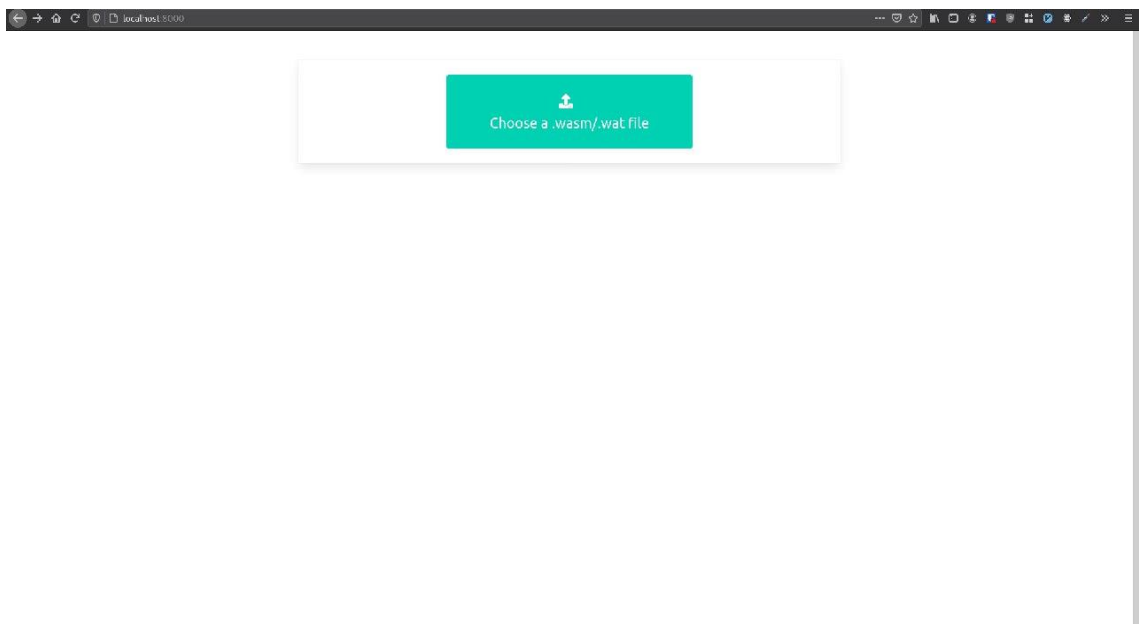


Ilustración 61 Vista inicial de la interfaz web

Tras elegir un fichero válido para el simulador, aparecerá un botón que permitirá instanciar o inicializar las distintas estructuras que se encuentran dentro del módulo, como se aprecia en la Ilustración 61.

Una vez se ha inicializado todas las estructuras pertinentes del módulo, se muestra una vista que muestra las distintas **funciones exportadas** del módulo, así como las instrucciones de las que se componen y la posibilidad de introducir valores para sus argumentos.

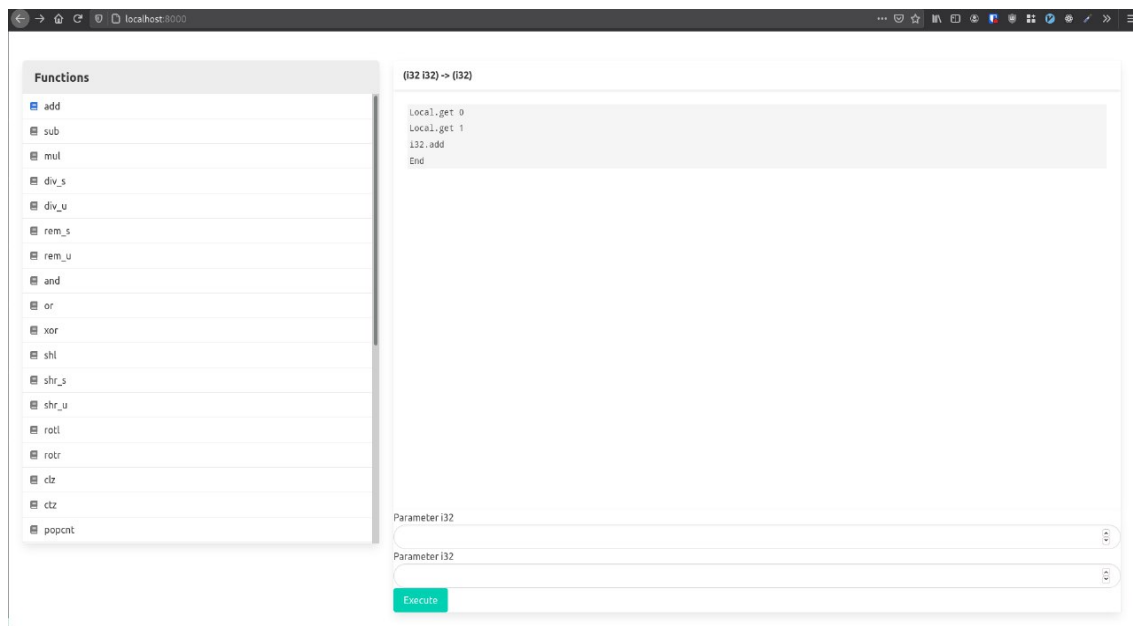


Ilustración 62 Vista de elección de función

Una vez se elige una función y sus argumentos, en caso de que lo tengan, a continuación se mostrará la última vista importante de la interfaz web, donde se muestra toda la información relevante a la ejecución de una función.

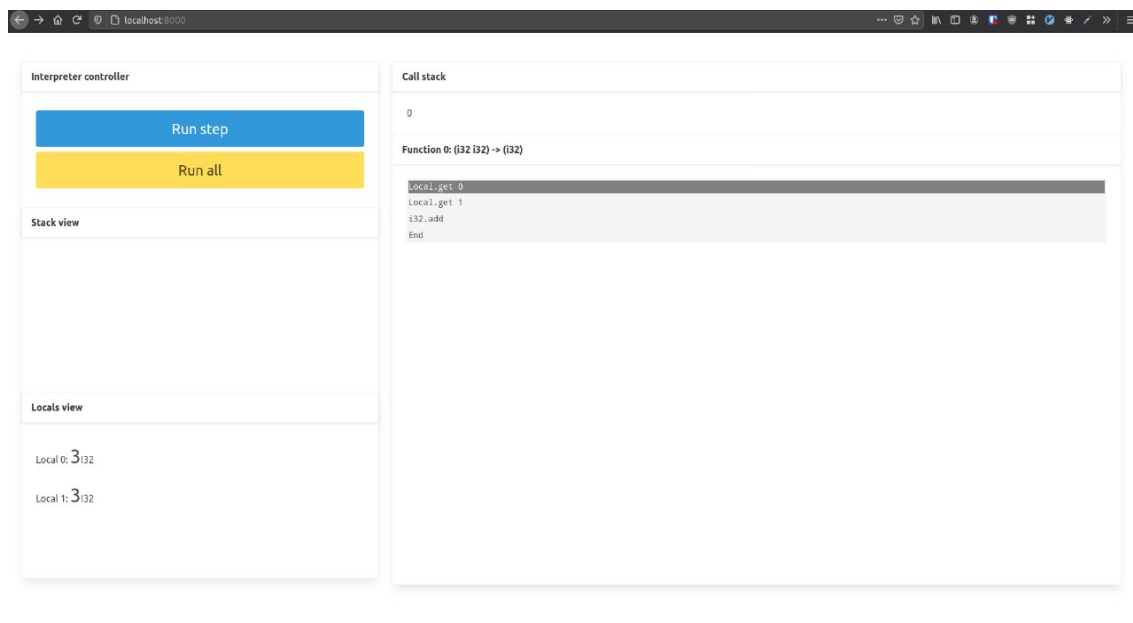


Ilustración 63 Vista de control del simulador

En la Ilustración 63 se observa varios paneles que muestran información específica de la ejecución:

- **Interpreter controller:** Se muestran los controles que permiten la ejecución instrucción a instrucción del simulador, así como una opción que permite ejecutar toda la función hasta su final.
- **Stack view:** Permite ver el estado actual de la pila del entorno de WebAssembly, de manera que muestra tanto el valor que guarda como su tipo.
- **Call stack:** Permite ver la pila de llamada de funciones del entorno de WebAssembly. Se muestra el *function address* de la función, pues aquí pueden haber funciones que no tengan un identificador mediante una cadena de caracteres.
- **Locals view:** Permite ver el estado de los registros locales para esa función en específica.
- En el panel que ocupa la mayor parte de la pantalla podemos ver los argumentos y la salida de la función que está en lo alto del *call stack*, así como las instrucciones de dicha función.

Es importante remarcar que la funcionalidad de esta interfaz es bastante limitada por temas de tiempo, en el posible caso de que eso sucediera, sería necesario actualizar debidamente este manual de usuario con las funcionalidades cambiadas y/o añadidas.



# Apéndice C

## Requisitos del proyecto

### 1. Parsing

- 1.1. El sistema permitirá generar una estructura interna a partir de un fichero binario de WebAssembly.
- 1.2. El sistema permitirá crear una estructura interna a partir de un fichero en formato textual de WebAssembly.

### 2. Elementos de la arquitectura

- 2.1. El sistema permitirá tener una representación interna de la lista de funciones exportadas en el fichero fuente.
  - 2.1.1. El sistema permitirá obtener una o varias funciones de esa lista.
- 2.2. El sistema deberá tener una representación interna de una pila de valores de WebAssembly.
  - 2.2.1. El sistema permitirá controlar el comportamiento de la pila mediante acciones de añadir (push) y eliminar (pop) valores de la pila.
  - 2.2.2. (Opcional) El sistema permitirá modificar cualquier posición de la pila en cualquier momento.
- 2.3. El sistema permitirá tener una representación interna de la memoria interna de WebAssembly.
  - 2.3.1. El sistema permitirá el manejo de esta memoria con precisión de bytes.
  - 2.3.2. El sistema permitirá ver el contenido de una parte de esta memoria dados un offset y un tamaño de memoria.
  - 2.3.3. El sistema permitirá el manejo y modificación de la memoria dado un offset.
- 2.4. El sistema permitirá tener una representación interna de la estructura de *Globals* de WebAssembly.
  - 2.4.1. El sistema permitirá exponer el estado de los *Globals* desde fuera del entorno.
- 2.5. El sistema permitirá tener una representación interna de la estructura de *Table* de WebAssembly.

### 3. Ejecución

- 3.1. El sistema debe permitir la ejecución de las **instrucciones numéricas** definidas en WebAssembly.
- 3.2. El sistema debe permitir la ejecución de las **instrucciones de manejo de memoria** definidas en WebAssembly.

- 3.3. El sistema debe permitir la ejecución de **instrucciones de control de flujo** definidas en WebAssembly.
- 3.4. El sistema permitirá la elección de distintos modos de ejecución del simulador.
  - 3.4.1. El sistema debe permitir la ejecución completa de un fichero de WebAssembly, mostrando al final el resultado de la operación.
  - 3.4.2. El sistema debe permitir la ejecución paso a paso de un fichero de WebAssembly, mostrando en cada paso el estado de las distintas estructuras.
4. Vista web
  - 4.1. El sistema permitirá cargar un fichero de WebAssembly mediante un formulario.
  - 4.2. El sistema mostrará los controles que permitan manejar el manejo de la ejecución del simulador.
  - 4.3. El sistema mostrará una lista con las funciones exportadas en el módulo cargado.
    - 4.3.1. El sistema permitirá seleccionar una función de la lista y mostrar su código fuente en un campo de texto.
  - 4.4. El sistema permitirá introducir los argumentos de una función mediante un formulario.
  - 4.5. El sistema mostrará una representación visual de los valores contenidos en la pila.







UNIVERSIDAD  
DE MÁLAGA

| [uma.es](http://uma.es)

E.T.S de Ingeniería Informática  
Bulevar Louis Pasteur, 35  
Campus de Teatinos  
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA