

Universidad Carlos III de Madrid
Escuela Politécnica Superior
Grado en Ingeniería Informática



Universidad
Carlos III
de Madrid

Trabajo Fin de Grado

WepSIM: Simulador de un procesador elemental con unidad de control microprogramada

Autor: Javier Prieto Cepeda
Tutor: Félix García Carballeira

Leganés, Madrid, España
Junio 2017

*If you would not be forgotten
As soon as you are dead and rotten,
Either write things worth reading,
Or do things worth the writing.*

Benjamin Franklin

Agradecimientos

Es muy difícil incluir en tan pocas palabras a todas aquellas personas que han puesto su granito de arena a lo largo de estos años para que a día de hoy, haya logrado llegar hasta aquí. Es por ello que si me dejo a alguien sin incluir, me disculpe y no dude en sentirse parte de estas palabras.

En primer lugar, quiero agradecer de corazón a mis padres, Tomás y Raquel, el haberme educado y enseñado a elegir aquel camino que me hiciera feliz de verdad, dándome la oportunidad de estudiar con su esfuerzo y sacrificio. Aquí también estás tú, David. Pese a llegar unos años después de mí, no paras de enseñarme día a día nuevas lecciones. De los pequeños también se aprende. Gracias.

También debo de dar las gracias a la persona que ha tenido que aguantar estos cinco años las consecuencias de estudiar esta carrera. Míriam, gracias por tu paciencia, tus consejos y tus ánimos.

Mis abuelos también han sido parte de esas personas que han sumado su granito de arena enseñándome y aconsejándome en todo momento, ejerciendo de guías. Sé que vosotros habéis sufrido también mucho en estos años, pero aquí está el premio.

Por otro lado, debo de agradecer a dos personas el darme la oportunidad de realizar este proyecto. Félix y Álex, gracias por este tiempo en el que tanto he podido aprender y en el que tanto me habéis ayudado.

También son parte de este proyecto los grandes amigos que he hecho en la travesía por el grupo de investigación ARCOS. Uno de ellos además, después de compartir asiento en clase. Saúl, gracias por tu ayuda y por los buenísimos momentos que hemos compartido. Carlos, nos quedan más carreras por hacer juntos. Fran, Estefanía, Silvina, Alberto, Jesús Cristina, Garci, Pablo, David, Rafa y Alfredo, gracias por acogerme tan bien en el laboratorio y por ayudarme en todo momento.

Agradecer también a Jesús, Javi Blas y al resto de personas de componen el grupo de investigación ARCOS la ayuda prestada durante este tiempo.

Por otro lado, también quiero destacar a aquellos compañeros de prácticas y amigos que he hecho a lo largo de estos años en la carrera, y que siempre recordaré: Sergio, Planet, Rubén, Álvaro, Álex, Guille, Juanlu, Sandra y Marin.

**WepSIM: Simulador de un procesador elemental con unidad de control
micropogramada**

por

Javier Prieto Cepeda

Resumen

WepSIM es un simulador de un procesador elemental con unidad de control micropogramada basado en el procesador diseñado por el personal del grupo de investigación ARCOS del Departamento de Informática de la Universidad Carlos III de Madrid para la docencia en la asignatura Estructura de Computadores. Con esta herramienta se ofrece una visión integrada de la micropogramación de un computador y la programación en lenguaje ensamblador, dando la posibilidad de especificar distintos juegos de instrucciones y ofreciendo un nivel de detalle a nivel de ciclo de reloj. Gracias al diseño modular propuesto en el cual está basado este simulador, es posible añadir, modificar o quitar elementos existentes al modelo hardware diseñado. Debido a que solo precisa de un navegador web, es posible utilizarlo en casi cualquier momento y dispositivo. Todas estas características, hacen de WepSIM el primer simulador docente que unifica la micropogramación y la programación en lenguaje ensamblador permitiendo una fácil personalización del modelo hardware a simular y el uso de diferentes juegos de instrucciones. Mediante este simulador se busca facilitar la enseñanza y aprendizaje en el área docente de Estructura y Arquitectura de Computadores.

Palabras clave: MIPS · Simulación · Ensamblador · Micropogramación

Tutor: Félix García Carballeira
Catedrático de Universidad

Índice general

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos	3
1.3	Estructura del documento	3
2	Estado del arte	5
2.1	Simuladores para la enseñanza de la Estructura y Arquitectura de Computadores	5
2.1.1	Simuladores para microprogramación	6
2.1.2	Simuladores para programación en ensamblador	7
2.1.3	Propuesta de simulación unificada	12
2.2	Tecnologías web	13
2.2.1	HTML5	13
2.2.2	Frameworks	15
2.3	Procesador elemental WepSIM	16
3	Análisis	19
3.1	Descripción del proyecto	19
3.2	Requisitos	20
3.2.1	Requisitos de usuario	23
3.2.2	Modelo de casos de uso	29
3.2.3	Requisitos funcionales	38
3.2.4	Requisitos no-funcionales	47
3.3	Marco regulador	50
3.3.1	Aspectos legales	50

4 Diseño	53
4.1 Estudio de la solución final	53
4.1.1 Solución elegida	55
4.2 Arquitectura de WepSIM	55
4.2.1 Modelo hardware	57
4.2.2 Modelo software	59
4.2.3 Kernel del simulador	62
5 Implementación y despliegue	63
5.1 Implementación	63
5.2 Despliegue	73
6 Verificación, validación y evaluación	75
6.1 Verificación y validación	75
6.1.1 Pruebas de verificación	76
6.1.2 Pruebas de validación	87
7 Planificación y presupuesto	99
7.1 Planificación	99
7.1.1 Justificación de la metodología	99
7.1.2 Ciclo de vida	100
7.1.3 Tiempo estimado	101
7.2 Presupuesto	102
7.2.1 Coste del proyecto	102
7.2.2 Oferta de Proyecto Propuesta	105
7.3 Entorno socio-económico	106
8 Conclusiones y trabajos futuros	109
8.1 Conclusiones	109
8.1.1 Contribuciones	110
8.1.2 Publicaciones	111
8.2 Trabajos futuros	112
Anexos	113

A Summary	115
B Manual de usuario	127
Glosario	160
Acrónimos	160
Bibliografía	161

Índice de figuras

2-1 Ejemplo de definición de microcódigo en simulador Tanenbaum.	6
2-2 Interfaz SPIM.	7
2-3 Interfaz QtSPIM.	8
2-4 Interfaz MARS.	10
2-5 Interfaz em88110.	10
2-6 Interfaz WebMIPS.	11
2-7 Arquitectura CPU WepSIM	16
2-8 Arquitectura unidad de control WepSIM.	17
3-1 Diagrama modelo casos de uso 1.	29
3-2 Diagrama modelo casos de uso 2.	30
4-1 Arquitectura de WepSIM.	56
4-2 Modelado del hardware.	57
4-3 Ejemplo de modelado de una puerta triestado.	58
4-4 Ejemplo de modelado de un registro.	59
4-5 Ejemplo de formato de instrucción.	60
4-6 Ejemplo de código fuente en ensamblador.	60
4-7 Formato de instrucción descrito en el microcódigo y ejemplo de su traducción en binario.	61
4-8 Arquitectura del kernel del simulador.	62
5-1 Estructura de ficheros.	64
5-2 Dependencias entre ficheros.	65
6-1 Verificación y validación software.	76

7-1	Modelo en Espiral (Boehm, 2000).	101
7-2	Diagrama Gantt.	102
A-1	CPU Architecture WepSIM.	117
A-2	Control unit architecture in WepSIM.	118
A-3	WepSIM Architecture.	121
B-1	Pantalla principal: opción de carga de juego de firmware.	128
B-2	Pantalla firmware: cuadro de texto con firmware a cargar.	128
B-3	Pantalla firmware: firmware finalmente compilado.	129
B-4	Pantalla principal: opción de carga de código ensamblador.	130
B-5	Pantalla ensamblador: cuadro de texto con ensamblador a cargar.	130
B-6	Pantalla ensamblador: código finalmente compilado.	131
B-7	Pantalla principal: visualización de la memoria de control.	132
B-8	Pantalla principal: visualización del código en la memoria principal.	132
B-9	Pantalla principal: opciones para la ejecución.	133
B-10	Pantalla principal: visualización de la unidad de control.	134

Índice de tablas

2.1 Comparación de simuladores de ensamblador y microcódigo	12
3.1 Plantilla para la especificación de requisitos.	22
3.2 Requisito de usuario UR-C01.	23
3.3 Requisito de usuario UR-C02.	23
3.4 Requisito de usuario UR-C03.	24
3.5 Requisito de usuario UR-C04.	24
3.6 Requisito de usuario UR-C05.	24
3.7 Requisito de usuario UR-C06.	25
3.8 Requisito de usuario UR-C07.	25
3.9 Requisito de usuario UR-C08.	25
3.10 Requisito de usuario UR-C09.	26
3.11 Requisito de usuario UR-R01.	26
3.12 Requisito de usuario UR-R02.	27
3.13 Requisito de usuario UR-R03.	27
3.14 Requisito de usuario UR-R04.	27
3.15 Requisito de usuario UR-R05.	28
3.16 Requisito de usuario UR-R06.	28
3.17 Requisito de usuario UR-R07.	28
3.18 Plantilla de caso de uso.	31
3.19 Caso de uso UC-01.	31
3.20 Caso de uso UC-02	32
3.21 Caso de uso UC-03.	32
3.22 Caso de uso UC-04.	33

3.23 Caso de uso UC-05.	33
3.24 Caso de uso UC-06.	34
3.25 Caso de uso UC-07.	34
3.26 Caso de uso UC-08.	35
3.27 Caso de uso UC-09.	35
3.28 Caso de uso UC-10.	36
3.29 Caso de uso UC-11.	36
3.30 Caso de uso UC-12.	37
3.31 Caso de uso UC-13.	37
3.32 Requisito funcional SR-F-F01.	38
3.33 Requisito funcional SR-F-F02.	38
3.34 Requisito funcional SR-F-F03.	39
3.35 Requisito funcional SR-F-F04.	39
3.36 Requisito funcional SR-F-F05.	39
3.37 Requisito funcional SR-F-F06.	40
3.38 Requisito funcional SR-F-F07.	40
3.39 Requisito funcional SR-F-F08.	41
3.40 Requisito funcional SR-F-F09.	41
3.41 Requisito funcional SR-F-F10.	42
3.42 Requisito funcional SR-F-F11.	42
3.43 Requisito funcional SR-F-F12.	42
3.44 Requisito funcional SR-F-F13.	43
3.45 Requisito funcional SR-F-F14.	43
3.46 Requisito funcional SR-F-F15.	43
3.47 Requisito funcional SR-F-F16.	44
3.48 Requisito funcional SR-F-F17.	44
3.49 Requisito funcional SR-F-F18.	45
3.50 Requisito funcional SR-F-F19.	45
3.51 Requisito funcional SR-F-F20.	45
3.52 Requisito funcional SR-F-F21.	46
3.53 Requisito funcional SR-F-F22.	46
3.54 Requisito no-funcional SR-NF-PL01.	47

3.55 Requisito no-funcional SR-NF-PL02.	47
3.56 Requisito no-funcional SR-NF-PL03.	48
3.57 Requisito no-funcional SR-NF-PL04.	48
3.58 Requisito no-funcional SR-NF-PL05.	48
3.59 Requisito no-funcional SR-NF-UI01.	49
3.60 Requisito no-funcional SR-NF-P01.	49
3.61 Matriz de trazabilidad de requisitos.	51
 4.1 Comparación de frameworks y bibliotecas.	54
 6.1 Plantilla de pruebas de verificación.	77
6.2 Test de verificación VET-01.	78
6.3 Test de verificación VET-02.	78
6.4 Test de verificación VET-03.	79
6.5 Test de verificación VET-04.	79
6.6 Test de verificación VET-05.	80
6.7 Test de verificación VET-06.	80
6.8 Test de verificación VET-07.	81
6.9 Test de verificación VET-08.	82
6.10 Test de verificación VET-09.	82
6.11 Test de verificación VET-10.	83
6.12 Test de verificación VET-11.	83
6.13 Test de verificación VET-12.	84
6.14 Matriz de trazabilidad de pruebas de verificación.	86
6.15 Plantilla para test de validación.	87
6.16 Test de validación VAT-01.	88
6.17 Test de validación VAT-02.	89
6.18 Test de validación VAT-03.	90
6.19 Test de validación VAT-04.	91
6.20 Test de validación VAT-05.	92
6.21 Test de validación VAT-06.	93
6.22 Test de validación VAT-07.	94
6.23 Test de validación VAT-08.	94

6.24 Test de validación VAT-09.	95
6.25 Test de validación VAT-10.	95
6.26 Test de validación VAT-11.	96
6.27 Matriz de trazabilidad de pruebas de validación.	97
7.1 Información del Proyecto.	103
7.2 Costes de recursos humanos.	103
7.3 Costes de equipamiento.	104
7.4 Otros costes directos.	105
7.5 Resumen de costes.	105
7.6 Oferta propuesta.	106

Capítulo 1

Introducción

El primer capítulo introduce brevemente el objetivo del proyecto, incluyendo las características clave del proyecto y su motivación (Sección 1.1, *Motivación*), los objetivos del proyecto (Sección 1.2, *Objetivos*), y toda la estructura del documento (Sección 1.3, *Estructura del documento*).

1.1 Motivación

La enseñanza de la arquitectura de un computador es una parte básica y fundamental en la formación de los estudiantes de Ingeniería Informática, mediante la cual logran obtener una visión y comprensión del comportamiento a bajo nivel de un computador. Para lograr que los estudiantes comprendan y asienten correctamente los fundamentos teóricos, es necesario el uso de clases prácticas, en donde pueda ser capaz de interactuar con un computador de arquitectura igual o similar a la explicada en teoría y además logre extrapolar los fundamentos teóricos al comportamiento real del computador.

Uno de los principales problemas a la hora de diseñar estas clases prácticas es lograr obtener los recursos necesarios para que los estudiantes puedan hacer uso de un computador similar al visto en las clases teóricas. Actualmente, existen distintas herramientas que permiten simular diferentes componentes de un computador, pero no existe una herramienta que unifique la simulación de estos componentes, siendo un problema a la hora de obtener una visión global del funcionamiento de un computador.

Estas herramientas, pueden ser de dos tipos: emuladores y simuladores. Un emulador es un software que se encarga de imitar el comportamiento de una computadora de forma que

programas pensados para una arquitectura concreta, puedan ser ejecutados en otra diferente. Por otro lado, un simulador es un software que trata reproducir el comportamiento de un computador, pero con un menor nivel de realismo que los emuladores, puesto que estos se encargan de modelar de forma precisa el dispositivo de manera que los programas ejecutados sobre él funcionen como si estuvieran siendo ejecutados en el dispositivo original.

Hay distintos simuladores que se pueden utilizar para trabajar con los principales aspectos que se tratan en las asignaturas de Estructura y Arquitectura de Computadores: ensamblador, caché, etc. La mayoría de estos simuladores se centran en un tipo de simulación concreto, lo que implica una pérdida de la visión del conjunto al necesitar el uso de diferentes simuladores. Por lo tanto, es mucho mejor tener siempre una visión centralizada de cualquier servicio, siendo más favorable el uso de una herramienta que reúna las máximas funcionalidades posibles.

Hay otro problema no menos importante: la mayoría de los simuladores están pensados para PC. Uno de los objetivos que nos planteamos con WepSIM es que pudiera ser utilizado en dispositivos móviles (*smartphones* o *tablets*), para ofrecer al estudiante una mayor flexibilidad en su uso.

Además de tener un simulador portable a distintas plataformas, el simulador ha de ser lo más autocontenido posible, de manera que integre la ayuda principal para su uso (no como un documento separado que sirva de manual de uso para ser impreso) permitiendo al usuario hacer un uso completo de la aplicación sin la necesidad de salir de ella.

Por todo ello, proponemos ofrecer un simulador que sea simple y modular, y que permita integrar la enseñanza de la microprogramación con la programación en ensamblador. En concreto, puede utilizarse para microprogramar un juego de instrucciones y ver el funcionamiento básico de un procesador, y para crear programas en ensamblador basados en el ensamblador definido por el anterior microcódigo. Este planteamiento permite una mejor comprensión de la programación de sistemas dado que es posible ver cómo interactúa el software en ensamblador con el hardware en el tratamiento de interrupciones. Finalmente, desde el punto de vista de la docencia, nuestro simulador ofrece una visión global de lo que pasa en hardware y software, evitando además el tiempo extra que supone el aprendizaje de distintas herramientas.

1.2 Objetivos

El objetivo principal de este proyecto, es desarrollar un simulador, que a diferencia de los existentes, pueda simular de forma completa el comportamiento de un procesador elemental permitiendo comprobar el estado de los componentes en cada ciclo de reloj, de manera que ayude a los estudiantes a comprender y asimilar de forma sencilla y visual el funcionamiento de un procesador. Los objetivos secundarios, derivados del objetivo principal, son:

- **O1:** Simular la ejecución del juego de instrucciones especificado en un computador denominado WepSIM desde el punto de vista de la microprogramación y la programación en ensamblador.
- **O2:** Permitir la especificación de diferentes juegos de instrucciones.
- **O3:** Permitir unificar la microprogramación en un computador y la programación en lenguaje ensamblador.
- **O4:** Permitir al usuario la visualización en cada ciclo de reloj del estado y comportamiento del computador simulado.

1.3 Estructura del documento

El documento contiene los siguientes capítulos:

- Capítulo 1, *Introducción*, presenta una breve descripción del contenido del documento. También incluye la motivación y los objetivos del proyecto.
- Capítulo 2, *Estado del arte*, incluye una descripción de los diferentes tipos de simuladores (de lenguaje ensamblador y de microcódigo), presentando el trabajo relacionado y una descripción de las tecnologías web actuales.
- Capítulo 3, *Análisis*, describe brevemente el proyecto, establece los requisitos y presenta el marco regulador del proyecto.
- Capítulo 4, *Diseño*, explica la solución elegida y detalla el diseño del sistema, incluyendo todos sus componentes.

- Capítulo 5, *Implementación y despliegue*, incluye los detalles de implementación de las partes principales del software desarrollado y las características necesarias para la implementación de la aplicación.
- Capítulo 6, *Verificación, validación y evaluación*, detalla una verificación y validación completa del proyecto.
- Capítulo 7, *Planificación y presupuesto*, presenta los conceptos relacionados con la planificación seguida, descompone todos los costes del proyecto y describe el entorno socio-económico.
- Capítulo 8, *Conclusiones y trabajos futuros*, incluye las contribuciones del proyecto, explica las principales conclusiones del proyecto y presenta los trabajos futuros.

Capítulo 2

Estado del arte

Este capítulo presenta el estado del arte, la última y más avanzada etapa de las tecnologías relacionadas con nuestra aplicación. En primer lugar, se presentan los diferentes simuladores existentes en el ámbito docente tanto para microprogramación (Sección 2.1.1) como para la programación en código ensamblador (Sección 2.1.2), realizando una comparación de nuestro trabajo con el contexto actual de los distintos simuladores expuestos previamente (Sección 2.1.3). Después, se presentan las diferentes tecnologías web más utilizadas en la actualidad (Sección 2.2). Por último se describe el procesador en el cual se basa WepSIM (Sección 2.3).

2.1 Simuladores para la enseñanza de la Estructura y Arquitectura de Computadores

Son muchos los tipos de simuladores existentes en el ámbito docente para la explicación de las distintas partes que componen un computador. Estos simuladores, pueden ser de diferentes ámbitos como, por ejemplo, cachés, pipeline, circuitos integrados, microprogramación, programación en ensamblador, etc.

En esta sección, nos centramos en los tipos de simuladores que tienen relación directa con la propuesta realizada, como lo son los simuladores para la microprogramación y los simuladores para la programación en ensamblador.

2.1.1 Simuladores para microprogramación

En esta sección, se explican las diferentes herramientas existentes para el aprendizaje de microprogramación a través de simulaciones. Cabe destacar que, actualmente, existen pocos simuladores que nos proporcionen esta funcionalidad, principalmente debido a que una gran parte de ellos son utilizados de forma interna por las empresas para verificar el correcto funcionamiento de sus unidades en la fase de diseño y desarrollo.

P8080E [1] es un simulador desarrollado por el grupo de investigación DATSI en la Facultad de Informática de la Universidad Politécnica de Madrid. Este simulador, no consta de una interfaz gráfica portable e interactiva que se ajuste a las tecnologías actuales. Para poder realizar el desarrollo del microcódigo, el simulador requiere el binario completo de cada ciclo, de forma que resulta un tanto complejo su uso. Pese a ser un simulador bastante completo, no está orientado para la enseñanza de ensamblador y microprogramación con la misma herramienta, puesto que no permite la definición del juego de instrucciones con un formato diferente al definido por defecto.

SOURCE LINE

```
0: mar := pc; rd;
1: pc := pc + 1; rd;
2: ir := mbr;
3: tir := lshift(ir + ir);
4: tir := lshift(ir);
5: alu := tir; if n then goto 9;

6: mar := ir ; rd;
7: rd;
8: ac := mbr; goto 0;

9: mar := ir ; mbr := ac; wr;
10: wr; goto 0;
```

Figura 2-1: Ejemplo de definición de microcódigo en simulador Tanenbaum.

En [2], se describe el desarrollo y la implementación de un simulador para una arquitectura de microprogramación específica (Figura 2-1). Fue publicado en el año 1986 y está basado en la arquitectura definida en el entonces popular libro de ingeniería de computadores de *Tanenbaum et al.* [3]. El simulador, requería de la definición de cada uno de los ciclos de ejecución de las instrucciones para la posterior generación de la memoria de control. Esta definición, se realizaba mediante la sintaxis definida en el libro, lo que hacía de este simulador una herramienta bastante completa para aprender junto con el libro.

2.1.2 Simuladores para programación en ensamblador

En esta sección, se explican los diferentes simuladores existentes para la programación en ensamblador. Los simuladores más conocidos para labores docentes son SPIM, MARS y WebMIPS.

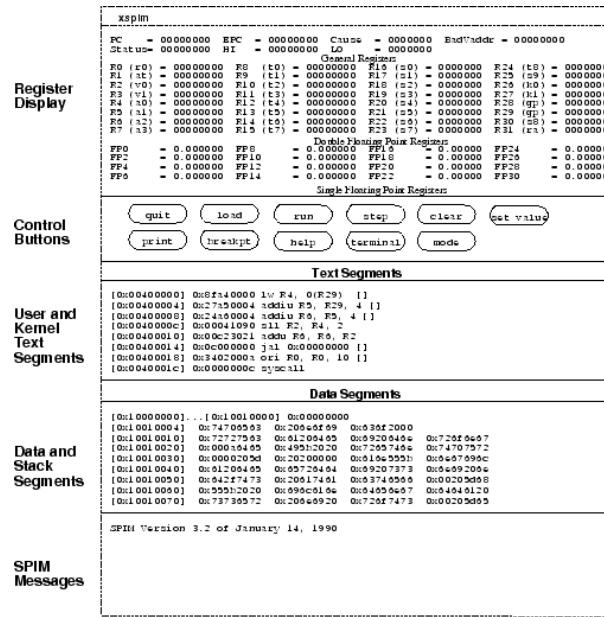


Figura 2-2: Interfaz SPIM.

SPIM [4] es un simulador de un procesador MIPS de 32 bits desarrollado a principios de 1990. En un primer momento implementaba el juego de instrucciones MIPS-1, utilizado por los computadores MIPS R2000/R3000. Actualmente, implementa la arquitectura MIPS32 más reciente y sus instrucciones adicionales. Permite ejecutar programas en ensamblador para esta arquitectura. Además, también permite depurar el código implementado, de forma que el estudiante pueda corregir con mayor facilidad los errores cometidos. Este simulador, fue creado por James R. Larus y tiene versiones compiladas para Windows, Mac OS X y Unix/Linux e incluso tiene una versión básica para Android, aunque el diseño de su interfaz gráfica no está pensada para dispositivos móviles. Puesto que el simulador está totalmente enfocado al procesador MIPS, posee el juego completo de instrucciones para la versión de 32 bits del procesador y todas las directivas de las que consta el lenguaje. Otra característica que hace de SPIM un potente simulador, es proveer de un pequeño sistema operativo que soporta las principales llamadas al sistema mediante la instrucción *syscall*.

SPIM es un simulador que permite múltiples configuraciones mediante su interfaz de usu-

rio (Figura 2-2), de manera que se puede indicar:

- Activación del uso de pseudoinstrucciones.
 - Simulación de predicción de saltos y accesos a memoria, con la latencia correspondiente.
 - Activación del uso del manejador de la señal trap y la carga del manejador personalizado.
 - Activación de la visualización de los mensajes en caso de ocurrir excepciones.
 - Activación del uso de “*memory-mapped IO*”.

SPIM, además cuenta con una versión más actualizada del simulador, que posee una interfaz gráfica más potente (Figura 2-3). Este simulador se llama QtSPIM [5], y cuenta con todas las características anteriormente mencionadas de SPIM.

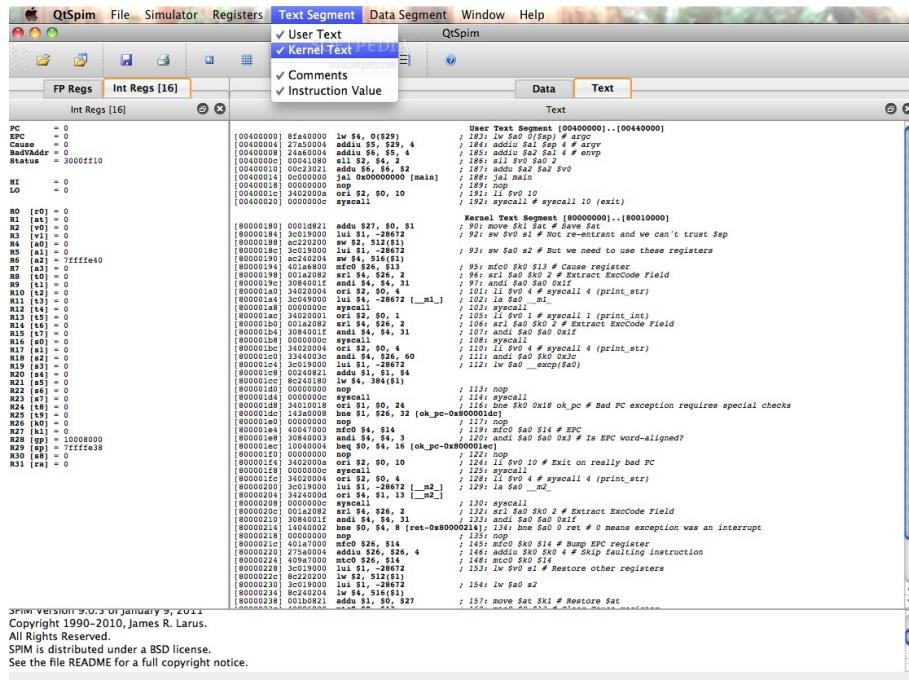


Figura 2-3: Interfaz QtSPIM.

MARS [6] es un simulador con interfaz gráfica desarrollado en JAVA para el lenguaje ensamblador MIPS. Fue creado en el año 2005 como una alternativa del simulador SPIM y diseñado específicamente para las necesidades limitadas de grado. Está basado en la arquitectura MIPS RISC, que tiene un número pequeño de elementos del lenguaje e instrucciones. Consta de operaciones de entrada/salida, saltos, condiciones y operaciones aritmético-lógicas tanto enteras

como en coma flotante. Alguna de las mejoras que incorpora MARS con respecto a SPIM son [7]:

- Depuración más sencilla y cómoda, debido a constar de una interfaz gráfica de usuario (GUI) (Figura 2-4). Esto se debe, a que para añadir un punto de ruptura, únicamente hay que pulsar sobre el elemento que tiene la instrucción.
- Los programas pueden ejecutarse de forma inversa, lo que permite volver a un estado anterior del computador en la ejecución de un programa.
- La velocidad de ejecución puede modificarse de forma que, si se ralentiza, el usuario puede observar cómo cada instrucción se resalta cuando está siendo ejecutada y ver los cambios que se producen en los valores del banco de registros y de las ubicaciones de memoria.
- La memoria y los registros se pueden modificar de una manera “*What You See Is What You Get*”(WYSIWYG).
- Incorpora un editor de texto, eliminando dependencias externas para el uso del simulador.
- Permite que los desarrolladores puedan suministrar nuevos complementos que, por ejemplo, sirvan para extender la arquitectura o mostrar los datos de forma intuitiva.

P88110 es un emulador descrito en [8] que se basa en la emulación de un procesador superescalar. El propósito fundamental de este emulador, es mostrar el funcionamiento de este tipo de procesadores, haciendo visible el funcionamiento del *pipeline* y las dependencias existentes (Figura 2-5). Está basado en una arquitectura específica (MC88110) y pese a integrar el efecto de un procesador superescalar no integra la microprogramación, que haría de él un emulador completo.

WebMIPS [9], es un simulador desarrollado en la Facultad de Ingeniería de la Información de Sienna, Italia. Está escrito en el lenguaje de programación ASP.NET y se utiliza mediante una página web (Figura 2-6), de forma que no requiere de su instalación en computador local. No soporta el juego completo de instrucciones de MIPS, sino que consta de un juego reducido de instrucciones que sirve de introducción al estudio de la arquitectura del computador. Con respecto a los anteriores simuladores mencionados, difiere al mostrar el esquema arquitectónico,

10 WepSIM: Simulador de un procesador elemental con unidad de control microprogramada

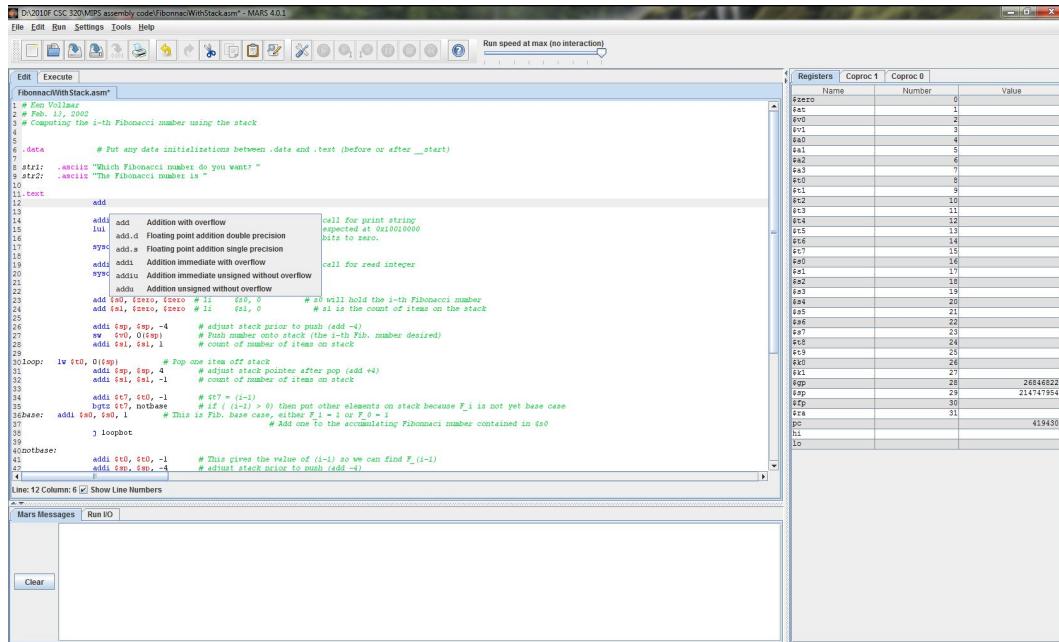


Figura 2-4: Interfaz MARS.

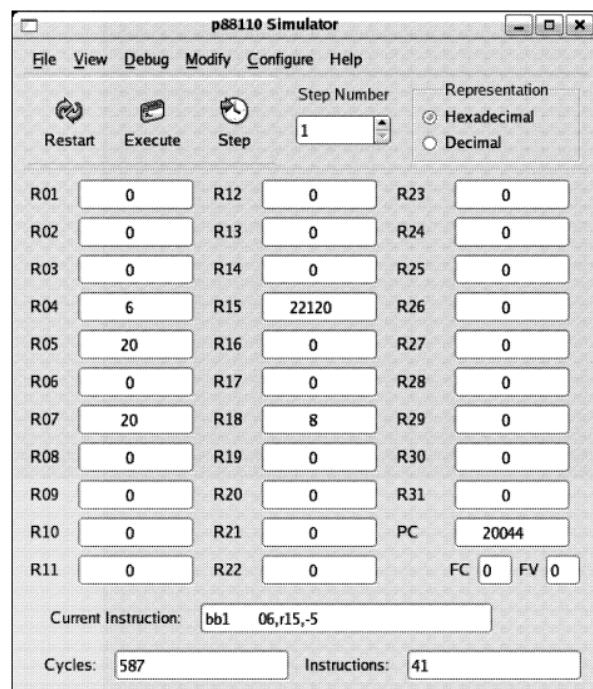


Figura 2-5: Interfaz em88110.

indicando las cinco etapas de *pipeline* de las que consta el computador que simula. Otra característica, es la indicación del número de ciclos de reloj que consume la ejecución del programa. Tiene como objetivo principal la demostración de la ejecución del juego de instrucciones básico explicado durante la asignatura “Arquitectura de Computadores” impartida por los creadores del simulador.

Entre las diferentes características a destacar de este simulador, se encuentran:

- Verificación del código ensamblador, comprobando que no existen errores en la definición el código.
- Dispone de códigos simples de ejemplo (“*load-and-play*”) que facilitan la comprensión del funcionamiento del procesador.
- Permite diferentes visualizaciones del estado de la memoria y los registros del procesador.
- Consta de dos modos de ejecución: paso a paso y ejecución completa del programa.

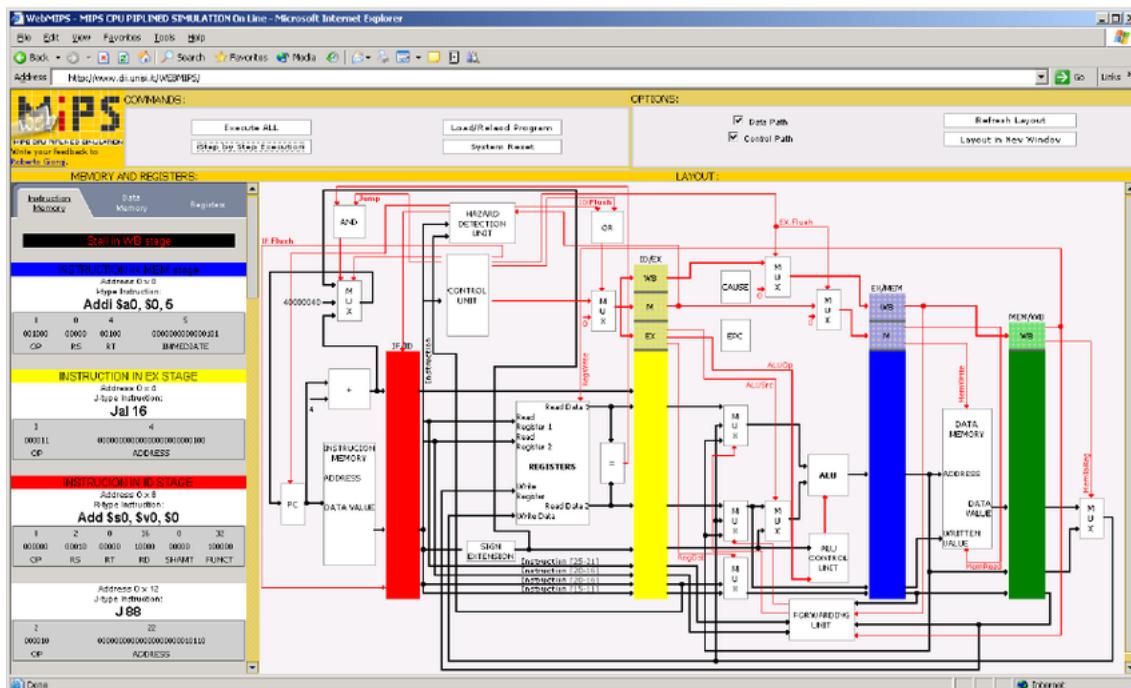


Figura 2-6: Interfaz WebMIPS.

2.1.3 Propuesta de simulación unificada

Hay distintas herramientas muy útiles para labores docentes relacionadas con la asignatura “Estructura de Computadores”, aunque como se comentó previamente, no existe una herramienta que nos proporcione el desarrollo y simulación tanto a nivel de microcódigo como a nivel de ensamblador (Figura 2.1). WepSIM, une ambos aspectos ofreciendo:

- Visión interrelacionada de la microprogramación y la programación en ensamblador.
- Flexibilidad en la plataforma usada, pensando en dispositivos móviles.
- Herramienta autocontenido, de forma que incorpora en sí misma tanto la ayuda como distintos ejemplos que favorecen la comprensión del simulador.
- Consta de un modelo hardware que puede ser modificado o ampliado en caso de ser necesario.
- Permite definir un amplio conjunto de instrucciones máquina.
- Puede ser usado como herramienta de microprogramación o de programación en ensamblador.

Tabla 2.1: Comparación de simuladores de ensamblador y microcódigo.

Simulador	SPIIM	QtSPIM	MARS	PC88110	WebMIPS	WepSIM	P8080E	MicMac
Ensamblador	✓	✓	✓	✓	✓	✓		
Microprogramación						✓	✓	✓
Multiplataforma	✓	✓			✓	✓		
Interactivo			✓			✓		
Juego de instrucciones personalizable						✓		

De este modo, con WepSIM se pretende crear un simulador unificado, abarcando los aspectos más relevantes de la asignatura “Estructura de Computadores” y permitiendo que todas las prácticas de la asignatura puedan realizarse en la misma plataforma, evitando la pérdida de tiempo y dificultad que supone para el estudiante el habituarse a diferentes entornos para cada una de las prácticas.

WepSIM se puede definir por tanto como un simulador de programación en ensamblador y microprogramación, flexible puesto que permite la personalización del juego de instrucciones

del computador, multiplataforma puesto que permite su uso desde diferentes tipos de dispositivos, e interactivo al poder modificar la configuración del computador en tiempo de ejecución.

2.2 Tecnologías web

Actualmente, gran parte de los recursos digitales disponibles son ofrecidos a través de Internet y su distribución es posible gracias a las tecnologías web. Estas tecnologías permiten la construcción de las páginas web haciendo uso de las tecnologías para el desarrollo de páginas web y las tecnologías de interconexión de computadores permitiendo a los usuarios el intercambio en formato de hipertexto de todo tipo de datos y de aplicaciones software.

2.2.1 HTML5

Cuando se realiza el desarrollo de una página web, son tres las tecnologías que se emplean con mayor frecuencia: HTML, CSS y JavaScript. Todas estas tecnologías vienen definidas y estandarizadas por el organismo internacional W3C, y conforman el lenguaje de marcado HTML5. Cada una de ellas tiene una función concreta en el funcionamiento de la página web:

- **HTML:** Es un lenguaje de marcas con el que se realiza la estructuración de la página web. Es un estándar que sirve de referencia para el software que conecta con la elaboración de páginas web en sus diferentes versiones. HTML define una estructura básica y un código (denominado código HTML) para la definición de contenido de una página web, como texto, imágenes, videos, juegos, entre otros.

El lenguaje HTML basa su filosofía de desarrollo en la diferenciación. Para añadir un elemento externo a la página (imagen, video, script, entre otros.), éste no se incrusta directamente en el código de la página, sino que se hace una referencia a la ubicación de dicho elemento mediante texto. De este modo, la página web contiene solamente texto mientras que recae en el navegador web (interpretador del código) la tarea de unir todos los elementos y renderizar la página final. Al ser un estándar, HTML busca ser un lenguaje que permita que cualquier página web escrita en una determinada versión, pueda ser interpretada de la misma forma (estándar) por cualquier navegador web actualizado.

En cambio, a lo largo de las diferentes versiones, se han añadido y eliminado diferentes características con la finalidad de hacer el lenguaje más eficiente y facilitar el desarrollo de

páginas web con diferentes plataformas y navegadores. Un navegador web desactualizado no será capaz de interpretar correctamente una página web escrita en una versión superior de HTML a la que pueda interpretar.

- **CSS:** Es un lenguaje de diseño gráfico utilizado para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado, como puede ser HTML. Su principal finalidad es establecer el diseño visual de las páginas web e interfaces de usuario escritas en los lenguajes HTML o XHTML. Este lenguaje, además, permite aplicar estilos no visuales como pueden ser las hojas de estilo audibles.

Esta tecnología web es usada por la mayoría de sitios web para crear páginas visualmente atractivas, interfaces de usuario para aplicaciones web, y GUIs para muchas de las aplicaciones móviles existentes en el mercado. Con ella se pretende separar el contenido de la página web de la presentación, buscando mejorar la accesibilidad del documento y dando una mayor flexibilidad y control en la especificación de las características de la presentación, permitiendo que varios documentos HTML compartan un mismo estilo, usando una única hoja de estilos reduciendo la complejidad y la repetición de código en la estructura del documento.

- **JavaScript:** Es un lenguaje de programación interpretado orientado a las páginas web, que surgió de la necesidad de ampliar las posibilidades del HTML. Su principal función en las páginas web es realizar tareas y operaciones en el marco de la aplicación cliente como la mejora de la interfaz de usuario y la generación de páginas web dinámicas, aunque también es utilizado en el lado del servidor .

Este lenguaje fue diseñado con una sintaxis similar a la del lenguaje de programación C, aunque adopta nombres y convenciones del lenguaje de programación Java. En cambio, Java y Javascript tienen propósitos y semánticas distintos.

En la actualidad, todos los navegadores realizan la interpretación del código JavaScript integrado en las páginas web. Para poder interactuar con una página web, el lenguaje está provisto de una implementación del DOM que facilita el control y manejo de las interacciones del usuario.

2.2.2 Frameworks

Como se indica en [10] las aplicaciones web se han convertido en los últimos años en complejos sistemas software con interfaces de usuario cada vez más parecidas a las aplicaciones de escritorio, dando servicio a procesos de negocio de considerable envergadura y estableciéndose sobre ellas requisitos estrictos de accesibilidad y respuesta. Para ayudar al desarrollo de estas aplicaciones web, se hace uso de diferentes *frameworks* y bibliotecas que añaden funcionalidad al entorno de trabajo. Los *frameworks* son una estructura conceptual para el desarrollo e implementación de aplicaciones web, teniendo como objetivos principales la aceleración del proceso de desarrollo, reutilización de código ya existente y promover buenas prácticas de desarrollo, como el uso de patrones. Entre los *frameworks* más utilizados en la actualidad, se encuentran:

- **Angular.js:** Es un *framework* MVC (Modelo-Vista-Controlador) de JavaScript de código abierto mantenido por Google para el desarrollo Web Front End, que permite crear aplicaciones *Single-Page*. Este *framework* ha supuesto un avance puesto que al implementar el patrón MVC ayuda al desarrollador a separar conceptos. De este modo, adapta y amplía el HTML tradicional para servir mejor contenido dinámico a través de un *data binding* bidireccional que permite la sincronización automática de modelos y vistas. Además facilita la creación de test unitarios, lo cual resulta de gran ayuda para los desarrolladores.
- **jQuery.js:** Es una biblioteca de JavaScript rápida, pequeña y característica. Ofrece una infraestructura que proporciona una mayor facilidad para la creación de aplicaciones complejas del lado del cliente. De esta forma, al utilizar esta biblioteca, el desarrollador se asegura que funcionará correctamente en los navegadores de los usuarios.
- **BootStrap:** Es un *framework* de código abierto para el diseño de aplicaciones web, en el cual se incluyen diferentes plantillas de diseño con tipografía, formularios, botones, menús de navegación y otros elementos de diseño basados en HTML y CSS que proporcionan el desarrollador ayuda a la hora de implementar la interfaz de usuario de su aplicación. Además, es compatible con la mayoría de los navegadores web.
- **Dojo:** Es un *framework* que contiene APIs que facilitan el desarrollo de aplicaciones web que utilizan la tecnología AJAX, conteniendo un sistema de empaquetado inteligente, efecto de interfaz de usuario, abstracción de eventos, almacenamiento de APIs en el cliente e interacción de APIs con AJAX.

2.3 Procesador elemental WepSIM

WepSIM es un procesador elemental con unidad de control microprogramada diseñado por el personal del grupo de investigación ARCOS de la Universidad Carlos III de Madrid para la docencia en la asignatura “Estructura de Computadores”.

En la Figura 2-7 se muestra la estructura del Procesador Elemental WepSIM. WepSIM, consta de un módulo de memoria, un dispositivo teclado, un dispositivo pantalla y un dispositivo de E/S genérico que se puede utilizar para trabajar con interrupciones.

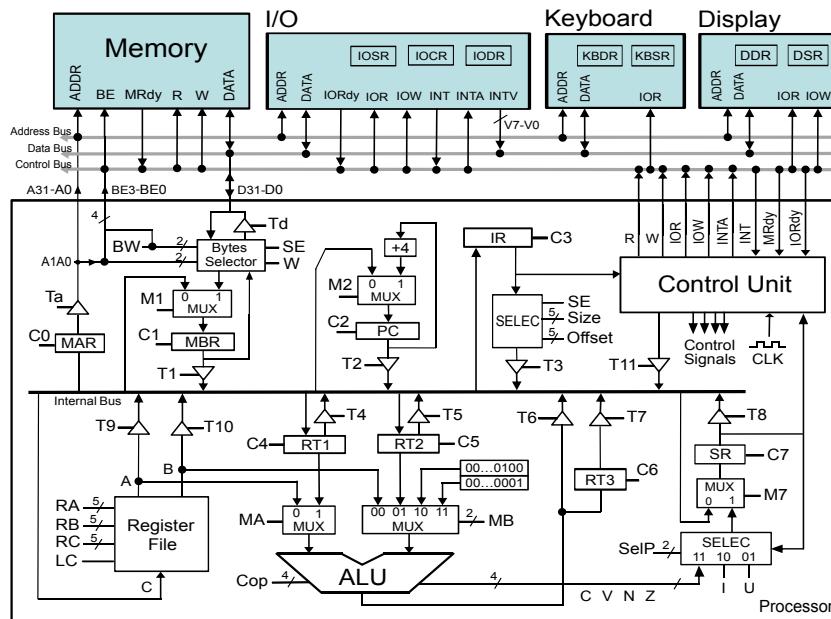


Figura 2-7: Arquitectura CPU WepSIM .

WepSIM es un procesador de 32 bits que direcciona la memoria por bytes, que cuenta con un banco de 32 registros y dos registros adicionales ($RT1$ y $RT2$) que no son visibles para el programador de ensamblador, pero que permiten el almacenamiento temporal de datos para la realización de operaciones intermedias. Desde los registros, es posible enviar los valores para operar en una ALU que dispone de las 15 operaciones aritmético-lógicas más comunes. El registro PC (contador de programa) tiene su propio operador de sumar cuatro, de forma que no es necesario hacer uso de la ALU para esta operación. El resultado de las operaciones realizadas en la ALU puede ser almacenado en un registro temporal ($RT3$) que también es invisible para el programador de ensamblador, o ser enviado directamente al bus interno a través del correspondiente triestado.

El registro de estado (*SR*) puede ser actualizado con los *flags* resultantes de la última opera-

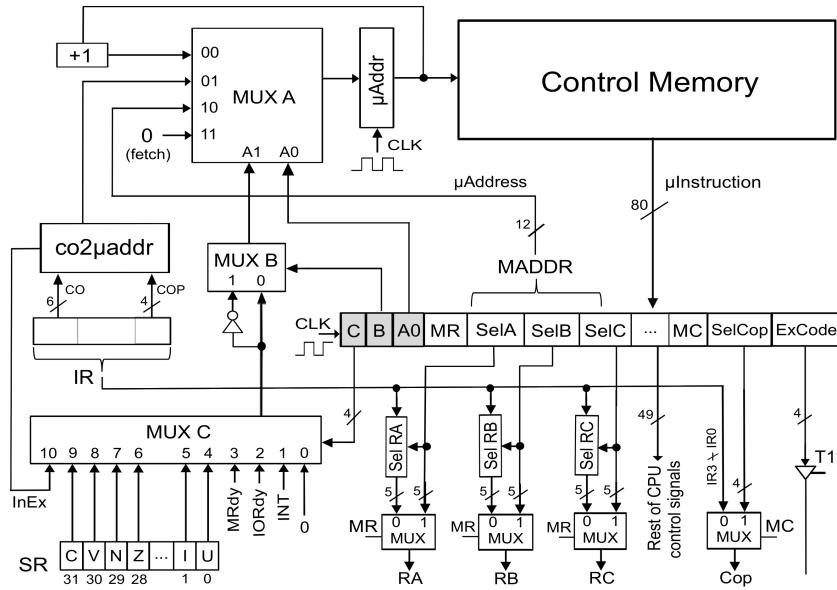


Figura 2-8: Arquitectura unidad de control WepSIM.

ción de la alu (O , N y Z). Para ello, $SELEC/SelP$ representa un bloque de circuitos que permite indicar qué parte del registro de estado (SR) debe ser actualizado. A la derecha de $SELEC/SelP$ llegan los bits del registro de estado SR como entrada ($Input=O\ N\ Z\ I\ U$) y $SelP$ permite seleccionar qué grupo de estos bits se actualizará en el registro de estado: los bits O , N y Z con los valores procedentes de la ALU , el bit I con el valor indicado o el bit U con el valor indicado para el mismo.

El registro de instrucción (IR) tiene asociado un módulo selector (circuito de más alto nivel que un multiplexor, etc.) que permite seleccionar un segmento del valor binario almacenado en el registro de instrucción que pasará hacia $T3$.

En concreto, se indica la posición (*Offset*, donde 0 represente el bit menos significativo del registro IR) inicial y el número de bits (*Size*) a tomar a partir de dicha posición inicial, así como si se desea hacer extensión de signo (*SE*) antes de pasar el valor a la entrada de $T3$.

Los registros MAR y MBR se usan para almacenar la dirección y el contenido asociado a esta dirección en las operaciones de lectura/escritura con la memoria. La memoria está diseñada para un funcionamiento síncrono o asíncrono. Actualmente funciona de forma síncrona, pero dispone de la señal $MRdy$ para en un futuro trabajar de forma asíncrona. El circuito de selección permite indicar qué porción de la palabra de memoria es la que se desea (un byte, dos bytes o una palabra completa de cuatro bytes).

Se dispone también de tres dispositivos de E/S: un teclado, una pantalla y un dispositivo

genérico que se puede configurar para generar diversos tipos de interrupciones.

Finalmente, la Unidad de Control genera las señales de control para cada ciclo de reloj. La Figura 2-8 muestra la Unidad de Control con mayor detalle. Se trata de una unidad de control microprogramada con secuenciamiento implícito. Las señales de control para el ciclo de reloj actual se almacenan en el registro de micro-instrucción (aquel con los campos *A*₀, *B*, *C*, *SelA*, etc.). El contenido de este registro proviene de la memoria de control, concretamente del contenido en la posición a la que apunta el registro de micro-dirección. La micro-dirección almacenada en este registro puede modificarse usando el multiplexor “MUX”. Hay cuatro opciones: la microdirección actual más uno, una micro-dirección indicada en la propia micro-instrucción (que se solapa con *SelA*, *SelB* y parcialmente con *SelE*), la primera micro-dirección asociada al campo de código de operación de la instrucción del registro *IR*, y finalmente el valor cero, que es la dirección de la memoria de control donde se almacena la microrutina correspondiente al *fetch*. La microdirección se puede seleccionar de forma condicional, para ello se usa el multiplexor “MUX C” que permite seleccionar los bits del registro de estado (*SR*) o valores de las señales de control de E/S.

Para generar los valores correspondientes a las señales selectoras del banco de registros *RA*, *RB* y *RE* se usan los circuitos selectores *SelRA*, *SelRB* y *SelRE*. Estos selectores toman como entrada los 32 bits del registro de instrucción (*IR*) por un lado y el campo *SelA*, *SelB* y *SelE* por otro, de forma que toman *SelX* como el desplazamiento dentro del registro de instrucción (de 0 a 32) desde donde tomar los siguientes 5 bits correspondientes a las señales *RA*, *RB* o *RE*. Permiten por tanto seleccionar 5 bits consecutivos de los 32 bits del registro de instrucción.

El multiplexor “MR” permite indicar si las señales *RA*, *RB* y *RE* serán literalmente los valores almacenados en *SelA*, *SelB* y *SelE* (MR=1) o bien si *SelA*, *SelB* y *SelE* indican el desplazamiento dentro de la instrucción donde están los valores a utilizar para *RA*, *RB* y *RE*. Esto último permite que en la instrucción se pueda indicar los registros a usar como operandos en el banco de registros, en lugar de indicarlos desde la micro-instrucción.

Para la señal *Cop* (código de operación en la *ALU*) la señal *MC* permite tomar el valor *SelCop* de la micro-instrucción (MC=1) o bien los 4 bits menos significativos del registro de instrucción (MC=0), es decir *IR3-IR0*.

En [11] se puede encontrar una descripción más detallada del procesador descrito anteriormente.

Capítulo 3

Análisis

El objetivo principal de este capítulo, es describir el proyecto mediante la obtención y especificación de los requisitos del simulador, que puede proporcionar información suficiente para un análisis detallado que, por lo tanto, puede servir para continuar diseñando e implementando (Capítulos 4, *Diseño*; y 5, *Implementación y despliegue*) un software que cumpla con esos requisitos.

Con el fin de obtener los requisitos del sistema, el tutor ha desempeñado el papel del cliente en diferentes reuniones, mientras que el estudiante ha desempeñado los roles de analista, diseñador, programador y *tester*.

La Sección 3.1 resume brevemente la descripción del proyecto. La Sección 3.2 especifica los requisitos del sistema, empezando con los requisitos de usuario y finalizando con los requisitos funcionales y no-funcionales. La Sección 3.2.2 especifica los caso de uso del sistema. Finalmente, la Sección 3.3 indica el conjunto de leyes y regulaciones para la gestión del software.

3.1 Descripción del proyecto

El objetivo de este proyecto es construir una herramienta que permita simular con realismo el comportamiento de un procesador basado en la arquitectura indicada por el el tutor del proyecto, de forma que sirva como única herramienta para los estudiantes a lo largo de la asignatura Estructura de Computadores.

Los simuladores actuales para la enseñanza de Estructura de Computadores, están focalizados a una función concreta, como puede ser la simulación de cachés, la simulación de código ensamblador o la simulación de microcódigo entre otros, pero a la hora de unificar todas estas

funcionalidades en una única herramienta, existe un vacío que genera una pérdida de tiempo en el aprendizaje de cada herramienta y la no posibilidad de ver una simulación completa.

El reto al que se enfrenta cualquier simulador educativo es el de ser capaz de simular fielmente el funcionamiento de un dispositivo, permitiendo al estudiante poder observar con el mayor detalle posible el comportamiento éste.

El sistema que se propone debe ser capaz de simular con realismo el comportamiento del procesador, permitiendo la definición del juego de instrucciones a utilizar para el posterior desarrollo de código ensamblador y su ejecución en el simulador con un alto nivel de detalle en cada uno de los ciclos de ejecución. De esta forma, los estudiantes podrán comprender fácilmente los contenidos teóricos expuestos en la asignatura y serán capaces de realizar todas las prácticas en una misma herramienta, pudiendo ver de forma incremental el desarrollo de software de bajo nivel sobre un procesador elemental.

3.2 Requisitos

Esta sección proporciona una descripción detallada de los requisitos de la aplicación. Para la tarea de la especificación de requisitos, se han seguido las prácticas recomendadas por IEEE [12]. De acuerdo con estas prácticas, una buena especificación debe abordar la funcionalidad del software, los problemas de rendimiento, las interfaces externas, otras características no funcionales y las limitaciones de diseño o implementación. Además, la especificación de los requisitos debe ser:

- **Completa:** el documento refleja todos los requisitos de software importantes.
- **Consistente:** los requisitos no deben generar conflictos entre sí.
- **Correcta:** cada requisito debe ser cumplido por el software según las necesidades del usuario.
- **Modificable:** la estructura de la especificación permite cambios en los requisitos de una manera simple, completa y consistente.
- **Clasificación basada en la importancia y la estabilidad:** cada requisito debe indicar su importancia y su estabilidad.

- **Trazable:** el origen de cada requisito es claro y se puede hacer referencia fácilmente en otras etapas.
- **Inequívoco:** cada requisito tiene una sola interpretación.
- **Verificable:** Cada requisito debe ser verificable, es decir, existe algún proceso para verificar que el software cumple con cada requisito.

A partir de los requisitos de los usuarios, que constituyen una referencia informal al comportamiento del producto que el cliente espera, derivamos los requisitos de software (en este caso, requisitos funcionales y no funcionales) que guiaron el proceso de diseño con información específica sobre la funcionalidad del sistema y otras características. Los requisitos recuperados se estructuraron de acuerdo con el siguiente esquema:

1. Requisitos de usuario

- (a) **Capacidad:** el requisito describe la funcionalidad esperada del sistema como en casos de uso.
- (b) **Restricción:** el requisito especifica las restricciones o condiciones que el sistema debe cumplir.

2. Requisitos de software

(a) Funcionales

- i. **Funcional:** el requisito describe la funcionalidad básica y el propósito del sistema mientras se minimiza la ambigüedad.
- ii. **Inverso:** el requisito limita la funcionalidad de la aplicación para aclarar su alcance.

(b) No-Funcionales

- i. **Rendimiento:** el requisito se relaciona con el rendimiento mínimo requerido del sistema resultante.
- ii. **Interfaz:** el requisito está relacionado con la interfaz de usuario de la aplicación.
- iii. **Escalabilidad:** el requisito está relacionado con la capacidad del sistema para adaptarse a cargas de trabajo cada vez mayores.

- iv. **Plataforma:** el requisito especifica las plataformas subyacentes de software y hardware en las que funcionará el sistema.

La Tabla 3.1 proporciona la plantilla utilizada para la especificación de requisitos. Es necesario tener en cuenta que para los requisitos del usuario, el formato de identificación será UR-XYY, donde X indica el subtipo de requisito: requisitos de capacidad (C) o restricciones (R). YY corresponde al número de requisito en su subcategoría. Para los requisitos de software, se utilizará el formato de identificación SR-X-YZZ, donde X indica si es un requisito funcional (F) o no funcional (NF), e Y representa su subcategoría: funcional (F), inversa (I), Rendimiento (P), interfaz (UI), escalabilidad (S) o plataforma (PL). ZZ corresponde al número de requisito en su subcategoría.

Tabla 3.1: *Plantilla para la especificación de requisitos.*

ID	Requisito ID.
Nombre	Nombre del requisito.
Tipo	Indica la categoría en la que se colocaría el requisito de acuerdo con el esquema descrito anteriormente.
Origen	Constituye la fuente del requisito. Puede ser el usuario, otro requisito u otros actores involucrados en el proyecto.
Prioridad	Indica la prioridad del requisito según su importancia. Un requisito puede ser identificado como <i>esencial</i> , <i>condicional</i> or <i>opcional</i> .
Estabilidad	Indica la variabilidad del requisito a través del proceso de desarrollo, definido como <i>estable</i> or <i>inestable</i> .
Descripción	Explicación detallada del requisito.

3.2.1 Requisitos de usuario

Esta subsección especifica los requisitos de usuario.

Tabla 3.2: *Requisito de usuario UR-C01.*

ID	UR-C01
Nombre	Simulación del modelo hardware propuesto
Tipo	Capacidad
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta simulará el comportamiento de la arquitectura definida para la asignatura Estructura de Computadores.

Tabla 3.3: *Requisito de usuario UR-C02.*

ID	UR-C02
Nombre	Definición de juego de instrucciones del simulador
Tipo	Capacidad
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir la definición del juego de instrucciones a utilizar por el usuario siguiendo el formato especificado por el tutor.

Tabla 3.4: Requisito de usuario UR-C03.

ID	UR-C03
Nombre	Operaciones con el juego de instrucciones
Tipo	Capacidad
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir al usuario las siguientes operaciones con el juego de instrucciones: cargar desde fichero, exportar a un fichero y generar el microcódigo.

Tabla 3.5: Requisito de usuario UR-C04.

ID	UR-C04
Nombre	Definición de juego del código ensamblador
Tipo	Capacidad
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir la definición del código ensamblador a simular siguiendo el formato utilizado en la asignatura Estructura de Computadores y el juego de instrucciones cargado previamente.

Tabla 3.6: Requisito de usuario UR-C05.

ID	UR-C05
Nombre	Operaciones con el código ensamblador
Tipo	Capacidad
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir al usuario las siguientes operaciones con el código ensamblador: cargar desde fichero, exportar a un fichero y generar el binario asociado.

Tabla 3.7: *Requisito de usuario UR-C06.*

ID	UR-C06
Nombre	Tipos de simulación
Tipo	Capacidad
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir tres tipos diferentes de simulación: simulación ciclo a ciclo de reloj, simulación instrucción a instrucción y simulación completa del código.

Tabla 3.8: *Requisito de usuario UR-C07.*

ID	UR-C07
Nombre	Configuración de la velocidad de las simulaciones
Tipo	Capacidad
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir al usuario seleccionar la velocidad de la simulación.

Tabla 3.9: *Requisito de usuario UR-C08.*

ID	UR-C08
Nombre	Esquemas de la arquitectura
Tipo	Capacidad
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe mostrar la arquitectura de la CPU y la Unidad de Control que forman el simulador.

Tabla 3.10: *Requisito de usuario UR-C09.*

ID	UR-C09
Nombre	Información a mostrar
Tipo	Capacidad
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe mostrar al usuario la información del estado del simulador en cada paso de la simulación.

Tabla 3.11: *Requisito de usuario UR-R01.*

ID	UR-R01
Nombre	Multiplataforma
Tipo	Restricción
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	El simulador no debe ser instalado en el dispositivo del usuario.

Tabla 3.12: *Requisito de usuario UR-R02.*

ID	UR-R02
Nombre	Navegadores web
Tipo	Restricción
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	El simulador debe de funcionar en los siguiente navegadores web: Microsoft Edge 30+, Mozilla Firefox 45+, Google Chrome 50+ y Safari 10+.

Tabla 3.13: *Requisito de usuario UR-R03.*

ID	UR-R03
Nombre	Interfaz de usuario
Tipo	Restricción
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La interfaz de usuario debe de ser compatible para uso en PCs y smartphones.

Tabla 3.14: *Requisito de usuario UR-R04.*

ID	UR-R04
Nombre	Tiempo por ciclo de reloj
Tipo	Restricción
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	El tiempo de ejecución en media de un ciclo de reloj del simulador no debe de sobrepasar 0,1 segundos.

Tabla 3.15: *Requisito de usuario UR-R05.*

ID	UR-R05
Nombre	Ejecución en local
Tipo	Restricción
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	Las simulaciones serán realizadas en el dispositivo del usuario, siendo únicamente necesario el servidor para el acceso a la herramienta.

Tabla 3.16: *Requisito de usuario UR-R06.*

ID	UR-R06
Nombre	Conexión a internet
Tipo	Restricción
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta podrá ser ejecutada sin conexión a internet.

Tabla 3.17: *Requisito de usuario UR-R07.*

ID	UR-R07
Nombre	Tecnologías de desarrollo
Tipo	Restricción
Origen	Usuario
Prioridad	Esencial
Estabilidad	Estable
Descripción	El simulador debe de ser desarrollado mediante el lenguaje de programación HTML5.

3.2.2 Modelo de casos de uso

Un caso de uso representa un uso típico que realizará alguno de los futuros usuarios del sistema desarrollado. El diagrama que se muestra a continuación representa los casos de uso de un usuario que utilice esta herramienta desarrollada. Se ha partido el diagrama en dos para hacerlo mucho más legible. A continuación, se muestra la primera parte del diagrama (Figura 3-1). En la página siguiente se muestra la segunda parte (Figura 3-2).

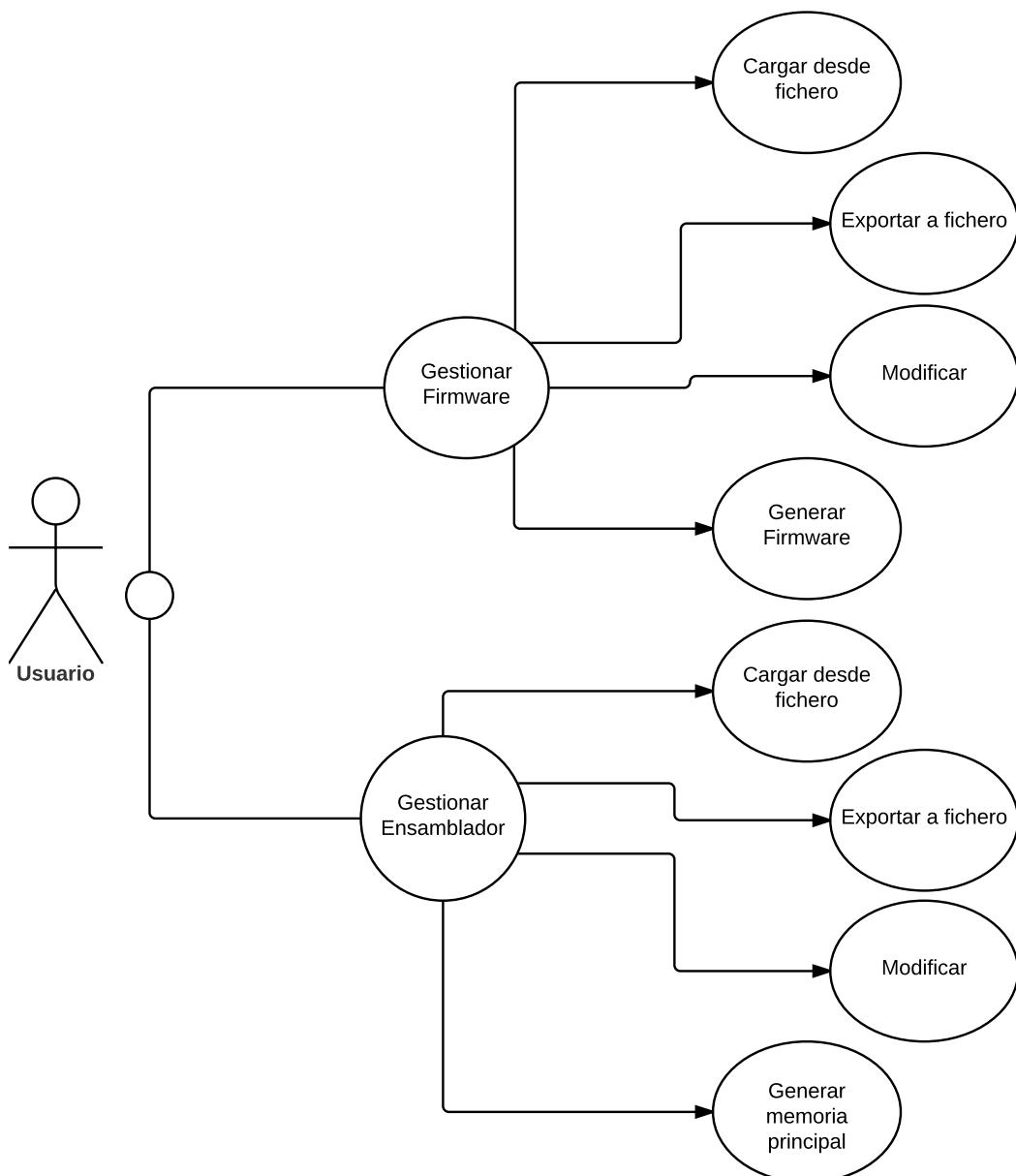


Figura 3-1: Diagrama modelo casos de uso 1.

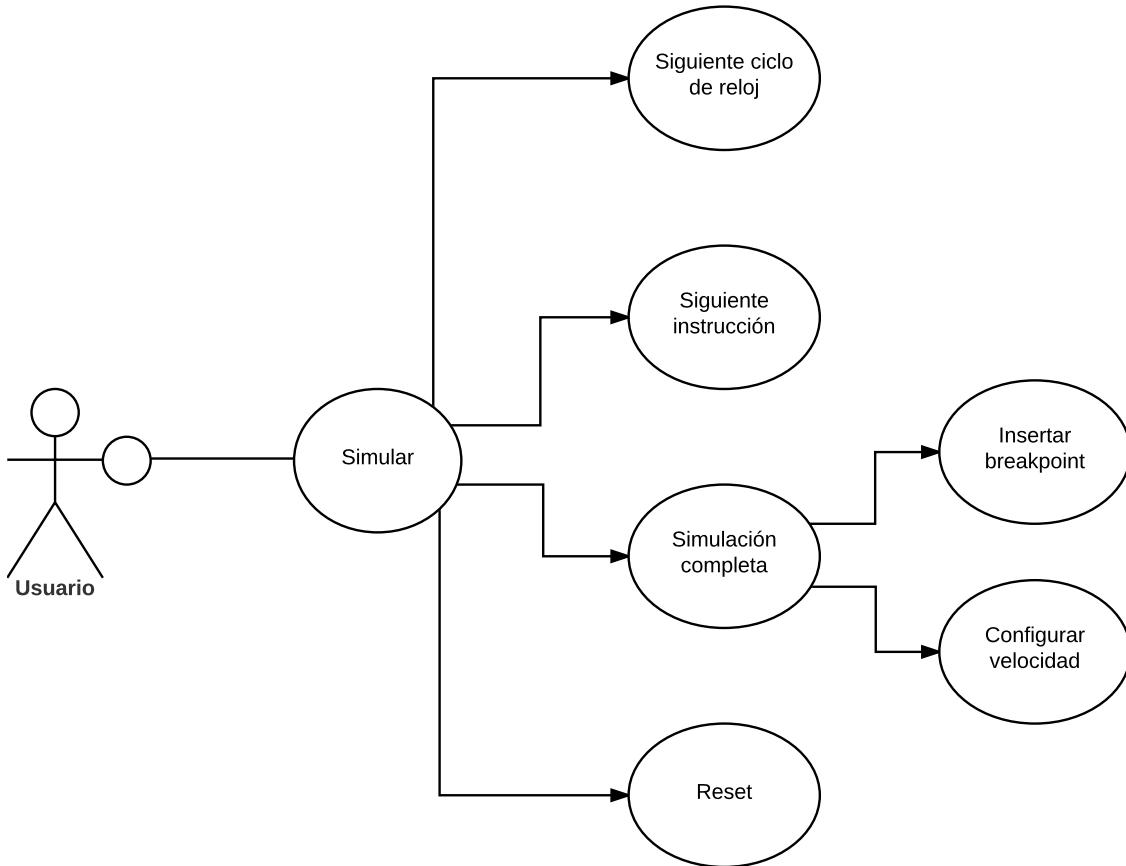


Figura 3-2: *Diagrama modelo casos de uso 2.*

Para realizar la especificación clara, completa y detallada de cada uno de los casos de uso recogidos en el diagrama anterior se utilizará una tabla donde se recogerán los siguientes campos de información.

La Tabla 3.18 proporciona la plantilla utilizada para la especificación de casos de uso. Es necesario tener en cuenta que el formato de identificación será UC-XX, donde XX corresponde al número del caso de uso.

A continuación, se exponen los casos de uso.

Tabla 3.18: *Plantilla de caso de uso.*

ID	Caso de uso ID.
Nombre	Nombre del caso de uso.
Actores	Describe el actor o los actores que intervienen en la realización del caso de uso.
Objetivo	Describe textualmente el cometido concreto del caso de uso.
Precondiciones	Muestra el estado del sistema que debe darse para que se pueda realizar el caso de uso.
Postcondiciones	Presenta el estado del sistema tras la realización del caso de uso.
Escenario Básico	Especifica la secuencia de pasos principales que se efectúan para realizar el caso de uso.

Tabla 3.19: *Caso de uso UC-01.*

ID	UC-01
Nombre	Cargar juego de instrucciones
Actores	Usuario
Objetivo	Cargar en la herramienta un nuevo juego de instrucciones desde un fichero que indica el usuario.
Precondiciones	Ninguna.
Postcondiciones	El juego de instrucciones se carga en la herramienta, mostrándose en el editor de texto correspondiente al microcódigo.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña firmware. ■ Pulsa el botón cargar firmware. ■ Selecciona el fichero deseado y acepta.

Tabla 3.20: *Caso de uso UC-02*

ID	UC-02
Nombre	Exportar juego de instrucciones
Actores	Usuario
Objetivo	Exportar a un fichero el juego de instrucciones cargado en la memoria de control.
Precondiciones	La memoria de control ha sido generada.
Postcondiciones	Se genera un fichero en el dispositivo del usuario con el juego de instrucciones cargado previamente en la memoria de control.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña firmware. ■ Pulsa el botón guardar firmware. ■ Selecciona el nombre de fichero deseado y acepta.

Tabla 3.21: *Caso de uso UC-03.*

ID	UC-03
Nombre	Modificar juego de instrucciones
Actores	Usuario
Objetivo	Editar el juego de instrucciones desde la herramienta.
Precondiciones	Ninguna.
Postcondiciones	Las modificaciones realizadas por el usuario en el juego de instrucciones son mostradas en el editor de texto correspondiente al microcódigo.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña firmware. ■ Realiza las modificaciones deseadas en el editor de texto.

Tabla 3.22: *Caso de uso UC-04.*

ID	UC-04
Nombre	Generar microcódigo
Actores	Usuario
Objetivo	Generar la memoria de control a partir del juego de instrucciones definido en la herramienta.
Precondiciones	El juego de instrucciones debe estar definido en la herramienta.
Postcondiciones	Se genera la memoria de control asociada al juego de instrucciones definido por el usuario.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña firmware. ■ Define el juego de instrucciones deseado en la herramienta. ■ Pulsa el botón generar firmware.

Tabla 3.23: *Caso de uso UC-05.*

ID	UC-05
Nombre	Cargar ensamblador
Actores	Usuario
Objetivo	Cargar en la herramienta un nuevo código ensamblador desde un fichero que indica el usuario.
Precondiciones	Ninguna.
Postcondiciones	El código ensamblador se carga en la herramienta, mostrándose en el editor de texto correspondiente al ensamblador.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña ensamblador. ■ Pulsa el botón cargar ensamblador. ■ Selecciona el fichero deseado y acepta.

Tabla 3.24: Caso de uso UC-06.

ID	UC-06
Nombre	Exportar ensamblador
Actores	Usuario
Objetivo	Exportar a un fichero el código ensamblador cargado para la simulación.
Precondiciones	El código ensamblador ha sido cargado en el simulador.
Postcondiciones	Se genera un fichero en el dispositivo del usuario con el código ensamblador cargado previamente en el simulador.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña ensamblador. ■ Pulsa el botón guardar ensamblador. ■ Selecciona el nombre de fichero deseado y acepta.

Tabla 3.25: Caso de uso UC-07.

ID	UC-07
Nombre	Modificar ensamblador
Actores	Usuario
Objetivo	Editar el código ensamblador desde la herramienta.
Precondiciones	Ninguna.
Postcondiciones	Las modificaciones realizadas por el usuario en el juego de instrucciones son mostradas en el editor de texto correspondiente al microcódigo.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña ensamblador. ■ Realiza las modificaciones deseadas en el editor de texto.

Tabla 3.26: Caso de uso UC-08.

ID	UC-08
Nombre	Generar memoria principal
Actores	Usuario
Objetivo	Cargar el código ensamblador definido por el usuario en el simulador, generando el contenido de la memoria principal correspondiente al ensamblador.
Precondiciones	El código ensamblador debe estar definido en la herramienta.
Postcondiciones	Se genera la memoria principal asociada al código ensamblador definido por el usuario.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña ensamblador. ■ Define el código ensamblador deseado en la herramienta. ■ Pulsa el botón compilar.

Tabla 3.27: Caso de uso UC-09.

ID	UC-09
Nombre	Siguiente ciclo de reloj
Actores	Usuario
Objetivo	Avanzar un ciclo de reloj en la simulación.
Precondiciones	La memoria de control y la memoria principal deben de estar generadas.
Postcondiciones	La ejecución de la simulación avanza un ciclo de reloj, siendo modificado el estado del simulador.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña simulador. ■ Pulsa el botón siguiente micro-instrucción.

Tabla 3.28: Caso de uso UC-10.

ID	UC-10
Nombre	Siguiente instrucción
Actores	Usuario
Objetivo	Avanzar una instrucción en la simulación.
Precondiciones	La memoria de control y la memoria principal deben de estar generadas.
Postcondiciones	La ejecución de la simulación avanza una instrucción, siendo modificado el estado del simulador.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña simulador. ■ Pulsa el botón siguiente instrucción.

Tabla 3.29: Caso de uso UC-11.

ID	UC-11
Nombre	Insertar breakpoint
Actores	Usuario
Objetivo	Insertar un punto de detención en una instrucción del código ensamblador a simular.
Precondiciones	La memoria de control y la memoria principal deben de estar generadas.
Postcondiciones	Se añade un breakpoint en la simulación.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña simulador. ■ Selecciona la pestaña Assembly Debugger. ■ Pulsa sobre la instrucción en la que añadir el breakpoint.

Tabla 3.30: *Caso de uso UC-12.*

ID	UC-12
Nombre	Configurar velocidad
Actores	Usuario
Objetivo	Configurar la velocidad de la simulación.
Precondiciones	La memoria de control y la memoria principal deben de estar generadas.
Postcondiciones	Se modifica la velocidad de la simulación.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña simulador. ■ Modifica la velocidad mediante la barra deslizable de velocidad.

Tabla 3.31: *Caso de uso UC-13.*

ID	UC-13
Nombre	Reinicio del simulador
Actores	Usuario
Objetivo	Reiniciar la simulación.
Precondiciones	La memoria de control y la memoria principal deben de estar generadas.
Postcondiciones	El simulador queda reiniciado y preparado para el comienzo de la simulación.
Escenario Básico	<ul style="list-style-type: none"> ■ El usuario selecciona la pestaña simulador. ■ Pulsa sobre el botón reset.

3.2.3 Requisitos funcionales

Esta subsección especifica los requisitos funcionales.

Tabla 3.32: *Requisito funcional SR-F-F01.*

ID	SR-F-F01
Nombre	Arquitectura de 32 bits
Tipo	Funcional
Origen	UR-C01
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe simular una arquitectura de 32 bits, implicando un banco de 32 registros, memoria principal con tamaño de 2^{32} bits y registros y buses de 32 bits.

Tabla 3.33: *Requisito funcional SR-F-F02.*

ID	SR-F-F02
Nombre	Unidad de control microprogramable
Tipo	Funcional
Origen	UR-C01
Prioridad	Esencial
Estabilidad	Estable
Descripción	La unidad de control debe de ser microprogramable, con secuenciamiento implícito, saltos a nivel de microdirección y microsaltos condicionales.

Tabla 3.34: *Requisito funcional SR-F-F03.*

ID	SR-F-F03
Nombre	Datos en complemento a dos (<i>Ca2</i>)
Tipo	Funcional
Origen	UR-C01
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta operará con una ALU de números enteros en complemento a dos.

Tabla 3.35: *Requisito funcional SR-F-F04.*

ID	SR-F-F04
Nombre	Definición de juego de instrucciones
Tipo	Funcional
Origen	UR-C02
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir al usuario la definición del juego de instrucciones a utilizar.

Tabla 3.36: *Requisito funcional SR-F-F05.*

ID	SR-F-F05
Nombre	Formato del juego de instrucciones
Tipo	Funcional
Origen	UR-C02
Prioridad	Esencial
Estabilidad	Estable
Descripción	El juego de instrucciones debe seguir el formato utilizado en la asignatura Estructura de Computadores, definiendo instrucciones del lenguaje ensamblador, su microcódigo asociado, nombrado de los registros y pseudoinstrucciones.

Tabla 3.37: *Requisito funcional SR-F-F06.*

ID	SR-F-F06
Nombre	Modificación del juego de instrucciones
Tipo	Funcional
Origen	UR-C03
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir la modificación del juego de instrucciones a utilizar por el usuario.

Tabla 3.38: *Requisito funcional SR-F-F07.*

ID	SR-F-F07
Nombre	Carga de juego de instrucciones
Tipo	Funcional
Origen	UR-C03
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir la carga de un fichero que contenga el juego de instrucciones.

Tabla 3.39: *Requisito funcional SR-F-F08.*

ID	SR-F-F08
Nombre	Exportar juego de instrucciones
Tipo	Funcional
Origen	UR-C03
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir exportar el juego de instrucciones en un fichero.

Tabla 3.40: *Requisito funcional SR-F-F09.*

ID	SR-F-F09
Nombre	Generación microcódigo del simulador
Tipo	Funcional
Origen	UR-C03
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe de generar el microcódigo simulado a partir del juego de instrucciones definido y cargado por el usuario.

Tabla 3.41: *Requisito funcional SR-F-F10.*

ID	SR-F-F10
Nombre	Definición del código ensamblador
Tipo	Funcional
Origen	UR-C04
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir al usuario la definición del código ensamblador a utilizar en la simulación.

Tabla 3.42: *Requisito funcional SR-F-F11.*

ID	SR-F-F11
Nombre	Formato del código ensamblador
Tipo	Funcional
Origen	UR-C04
Prioridad	Esencial
Estabilidad	Estable
Descripción	El código ensamblador a simular debe de seguir el formato utilizado en la asignatura Estructura de Computadores, utilizando el lenguaje definido en el juego de instrucciones.

Tabla 3.43: *Requisito funcional SR-F-F12.*

ID	SR-F-F12
Nombre	Edición del código ensamblador
Tipo	Funcional
Origen	UR-C05
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir modificar el código ensamblador.

Tabla 3.44: *Requisito funcional SR-F-F13.*

ID	SR-F-F13
Nombre	Carga del código ensamblador
Tipo	Funcional
Origen	UR-C05
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir la carga de un fichero que contenga el código ensamblador.

Tabla 3.45: *Requisito funcional SR-F-F14.*

ID	SR-F-F14
Nombre	Exportar código ensamblador
Tipo	Funcional
Origen	UR-C05
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe permitir exportar a un fichero el código ensamblador cargado en la herramienta.

Tabla 3.46: *Requisito funcional SR-F-F15.*

ID	SR-F-F15
Nombre	Generación ejecutable del código ensamblador
Tipo	Funcional
Origen	UR-C05
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe de generar el binario ejecutable asociado al código ensamblador definido para la simulación.

Tabla 3.47: *Requisito funcional SR-F-F16.*

ID	SR-F-F16
Nombre	Simulación ciclo a ciclo de reloj
Tipo	Funcional
Origen	UR-C06
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe de permitir la simulación ciclo a ciclo de reloj del código ensamblador cargado por el usuario.

Tabla 3.48: *Requisito funcional SR-F-F17.*

ID	SR-F-F17
Nombre	Simulación instrucción a instrucción
Tipo	Funcional
Origen	UR-C06
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe de permitir la simulación instrucción a instrucción del código ensamblador cargado por el usuario.

Tabla 3.49: *Requisito funcional SR-F-F18.*

ID	SR-F-F18
Nombre	Simulación completa
Tipo	Funcional
Origen	UR-C06
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe de permitir la simulación completa del código ensamblador cargado por el usuario.

Tabla 3.50: *Requisito funcional SR-F-F19.*

ID	SR-F-F19
Nombre	Velocidad de simulación
Tipo	Funcional
Origen	UR-C07
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe de permitir al usuario elegir la velocidad de simulación del código ensamblador cargado.

Tabla 3.51: *Requisito funcional SR-F-F20.*

ID	SR-F-F20
Nombre	Esquema CPU
Tipo	Funcional
Origen	UR-C08
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe de mostrar el esquema de la CPU del simulador, modificando el color de las señales activas y buses de datos actualizados en cada ciclo de reloj.

Tabla 3.52: *Requisito funcional SR-F-F21.*

ID	SR-F-F21
Nombre	Esquema unidad de control
Tipo	Funcional
Origen	UR-C08
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe de mostrar el esquema de la unidad de control del simulador, modificando el color de las señales activas y buses de datos actualizados en cada ciclo de reloj.

Tabla 3.53: *Requisito funcional SR-F-F22.*

ID	SR-F-F22
Nombre	Información de la simulación
Tipo	Funcional
Origen	UR-C09
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe de mostrar durante la ejecución el estado de los registros del simulador.

3.2.4 Requisitos no-funcionales

Esta subsección especifica los requisitos no-funcionales.

Tabla 3.54: *Requisito no-funcional SR-NF-PL01.*

ID	SR-NF-PL01
Nombre	Multiplataforma
Tipo	Plataforma
Origen	UR-R01
Prioridad	Esencial
Estabilidad	Estable
Descripción	El simulador debe de ser diseñado como una aplicación web.

Tabla 3.55: *Requisito no-funcional SR-NF-PL02.*

ID	SR-NF-PL02
Nombre	Navegadores web
Tipo	Plataforma
Origen	UR-R02
Prioridad	Esencial
Estabilidad	Estable
Descripción	El simulador debe poder ser ejecutado en los navegadores: Microsoft Edge 30+, Mozilla Firefox 45+, Google Chrome 50+ y Safari 10+.

Tabla 3.56: *Requisito no-funcional SR-NF-PL03.*

ID	SR-NF-PL03
Nombre	Plataforma de ejecución
Tipo	Plataforma
Origen	UR-R05
Prioridad	Esencial
Estabilidad	Estable
Descripción	Las ejecuciones de la herramienta deben ser realizadas en el dispositivo del usuario.

Tabla 3.57: *Requisito no-funcional SR-NF-PL04.*

ID	SR-NF-PL04
Nombre	Internet
Tipo	Plataforma
Origen	UR-R06
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta podrá ser ejecutada sin conexión a internet.

Tabla 3.58: *Requisito no-funcional SR-NF-PL05.*

ID	SR-NF-PL05
Nombre	Lenguaje de programación HTML5
Tipo	Plataforma
Origen	UR-R07
Prioridad	Esencial
Estabilidad	Estable
Descripción	La herramienta debe ser desarrollada en el lenguaje de programación HTML5 (HTML + CSS + JavaScript).

Tabla 3.59: Requisito no-funcional SR-NF-UI01.

ID	SR-NF-UI01
Nombre	Interfaz de usuario
Tipo	Interfaz
Origen	UR-R03
Prioridad	Esencial
Estabilidad	Estable
Descripción	La interfaz de usuario del simulador debe de ser compatible tanto con PCs como con plataformas móviles (tablets, smartphones, etc).

Tabla 3.60: Requisito no-funcional SR-NF-P01.

ID	SR-NF-P01
Nombre	Tiempo medio por ciclo de reloj
Tipo	Rendimiento
Origen	UR-R04
Prioridad	Esencial
Estabilidad	Estable
Descripción	El tiempo medio de ejecución de un ciclo de reloj no debe exceder 0,1 segundos.

La matriz de trazabilidad de entre requisitos de usuario y requisitos de software (Tabla 3.61) determina que todos los requisitos de usuario quedan reflejados en los requisitos de software.

3.3 Marco regulador

Esta sección discute las restricciones necesarias teniendo en cuenta el marco regulador. En concreto, se especifican las restricciones legales aplicables al simulador.

3.3.1 Aspectos legales

Para el uso de la mayoría de aplicaciones web, los usuarios deben de registrarse, y las bases de datos de las mismas manejan información confidencial de los usuarios, por lo que es necesario garantizar que terceros no puedan acceder a esa información. Una solución es cifrar la información transmitida mediante algún protocolo criptográfico. En España, este requisito es especificado en el artículo 104 del RD 1720/2007 [13], que se ocupa de la Ley Española de Protección de Datos.

A diferencia de lo anteriormente citado, la aplicación desarrollada no utiliza datos privados de los usuarios y tampoco transmite información confidencial a terceros, ya que es un simulador que únicamente utiliza los códigos generados por el usuario para ejecutarlos de forma local en el dispositivo del usuario.

Por otro lado, es crucial que el simulador esté disponible como software de código abierto. Queremos que cualquiera pueda redistribuir el código o modificarlo por los términos de la Licencia Pública General Menor de GNU (LGPL) [14]. Para ello, nuestro simulador está disponible en los siguientes sitios web:

<https://www.arcos.inf.uc3m.es/wepsim/>.

<https://github.com/wepsim/wepsim>.

Tabla 3.61: Matriz de trazabilidad de requisitos.

Requisitos	UR-C01	UR-C02	UR-C03	UR-C04	UR-C05	UR-C06	UR-C07	UR-C08	UR-C09	UR-R01	UR-R02	UR-R03	UR-R04	UR-R05	UR-R06	UR-R07
SR-F-F01	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F02	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F03	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F04	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F05	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F06	-	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F07	-	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F08	-	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F09	-	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F10	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F11	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-	-
SR-F-F12	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-
SR-F-F13	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-
SR-F-F14	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-
SR-F-F15	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-
SR-F-F16	-	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-
SR-F-F17	-	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-
SR-F-F18	-	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-
SR-F-F19	-	-	-	-	-	-	✓	-	-	-	-	-	-	-	-	-
SR-F-F20	-	-	-	-	-	-	-	✓	-	-	-	-	-	-	-	-
SR-F-F21	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-	-
SR-F-F22	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
SR-NF-PL01	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
SR-NF-PL02	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-
SR-NF-PL03	-	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-
SR-NF-PL04	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	-
SR-NF-PL05	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓
SR-NF-UI01	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-
SR-NF-P01	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-

Capítulo 4

Diseño

En este capítulo se realiza una descripción completa del simulador desarrollado, incluyendo la arquitectura interna y los diferentes componentes software, que componen la herramienta descritos con anterioridad en [15].

La Sección 4.1 discute el estudio de la solución final, considerando las diferentes alternativas existentes para el diseño y desarrollo del simulador. En la Sección 4.1.1 se indica la solución elegida y la compara con las alternativas consideradas. La Sección 4.2 describe cada uno de los módulos que componen el simulador.

4.1 Estudio de la solución final

Debido a los requisitos de usuario establecidos en este proyecto (Sección 3.2.1), en donde se especificaba que el software no debía de ser instalado en el dispositivo del usuario, y debía de poder ser ejecutado en los navegadores web indicados, se ha decidido que el diseño del simulador esté enfocado como una aplicación web.

Existen diferentes modelos arquitectónicos a la hora de realizar el diseño de una aplicación web. La primera opción, es el modelo cliente-servidor en el cual las tareas se reparten entre dos roles: un proveedor que proporciona recursos o servicios que es llamado servidor, y un consumidor que contacta con el servidor con el objetivo de hacer uso de los recursos que este provee. Es un modelo distribuido en el que la lógica de negocio y el cómputo recaen en el servidor, y el cliente provee la interfaz de usuario, realizando validaciones y procesos menores de la lógica de negocio. La segunda opción, es un modelo en el que toda la aplicación es ejecutada en el cliente, siendo únicamente necesario el servidor para el acceso al código de

la aplicación. Este modelo, posibilita incluso que el servidor no sea necesario en el caso de existir el código de la aplicación web en el cliente, pudiendo ser ejecutada sin la necesidad de internet. Ambos modelos, poseen una alta compatibilidad al hacer que las aplicaciones no sean dependientes del sistema operativo del usuario, sino del navegador web que este utiliza.

Debido a los requisitos de usuario establecidos en este proyecto (Sección 3.2.1), la segunda opción es la que cumple con la necesidad de no necesitar acceso a internet, de forma que la funcionalidad total del simulador quede en el dispositivo del usuario.

Una vez decidido el modelo de diseño, es necesario tener en cuenta las diferentes tecnologías a utilizar. Como se indicó en (Sección 3.2.1), es necesario que la aplicación sea diseñada siguiendo el lenguaje de marcado HTML5, que está formado por HTML, CSS y JavaScript. De este modo, es necesario contemplar los diferentes frameworks y bibliotecas existentes que ayudan a la hora de realizar el diseño de la aplicación, posibilitando el funcionamiento de esta tanto en ordenadores como en dispositivos móviles. Estas tecnologías descritas anteriormente en el Apartado 2.2, proporcionan distintas funcionalidades y enfoques a la hora de diseñar la aplicación, permitiendo la compatibilidad de la aplicación con la amplia mayoría de los navegadores web. En la Tabla 4.1, se analizan las funcionalidades y compatibilidades que proporcionan estas tecnologías, valorando si cumplen con los requisitos establecidos en este proyecto.

Tabla 4.1: Comparación de frameworks y bibliotecas.

Framework	Angular.js	jQuery.js	BootStrap	Dojo
Funcionalidad requerida	✓	✓	✓	✓
Ejecución en cliente	✓	✓	✓	
Compatibilidad navegadores requeridos	✓	✓	✓	✓
Soporte plataformas móviles	✓	✓	✓	

De este modo, se puede observar como pese a que *Angular.js* nos proporciona la mayoría de funcionalidades requeridas, tiene una curva de aprendizaje bastante mayor que *jQuery*, lo cual hace que *jQuery* se imponga a la hora de su elección debido a que las ventajas que supondría utilizar *Angular.js* quedan disueltas por este inconveniente. Por otro lado, sería interesante el uso de *Dojo* a la hora de utilizar una tecnología que nos proporcione funcionalidad y efectos en la interfaz de usuario y nos ayude a la capturación de eventos, pero queda descartado al estar enfocado a aplicaciones que utilizan *AJAX*. De esta forma, *BootStrap* será la tecnología

elegida para dotar a la aplicación de la funcionalidad requerida a nivel de interfaz de usuario y complementando tanto a JavaScript como a jQuery.

4.1.1 Solución elegida

Para que los profesores de la asignatura Estructura de Computadores puedan hacer uso de una herramienta que sirva de ayuda para la explicación de los conceptos teóricos de la asignatura, y los estudiantes puedan utilizarla para comprender estos conceptos y realizar posteriormente las prácticas de la asignatura, se propone el diseño e implementación de una herramienta web que simule con realismo en funcionamiento de un procesador elemental con unidad de control microprogramable.

Este simulador, será desarrollado como una herramienta web debido a la portabilidad que proporciona, ya que podrá ser ejecutado sobre un gran número de diferentes dispositivos independientemente del sistema operativo que utilice, puesto que únicamente necesita un navegador web para su correcto funcionamiento. De esta forma, los profesores y estudiantes podrán hacer uso de la herramienta sin depender de su instalación en el dispositivo a utilizar, incluso pudiendo los estudiantes realizar las prácticas sobre dispositivos móviles.

Para lograr dicha portabilidad, el simulador ha sido desarrollado en HTML5 (HTML + JavaScript + CSS) haciendo posible su ejecución en cualquier plataforma (*smartphones, tablets, PCs, etc.*) que pueden ejecutar Microsoft Edge, Mozilla Firefox, Google Chrome o Safari. Además, la herramienta depende de los siguientes frameworks/bibliotecas: JQuery, JQueryUI, JQuery Mobile, Knockout y BootStrap.

Por tanto, la solución elegida es capaz de unificar en una misma herramienta todas las funcionalidades requeridas para la enseñanza de Estructura de computadores con un alto nivel de detalle, con alta disponibilidad al facilitarse su uso como una herramienta web, y con una gran portabilidad puesto que podrá ser ejecutada sobre un gran número de diversos dispositivos, buscando en todo momento una solución sin la necesidad de servidor.

4.2 Arquitectura de WepSIM

La arquitectura de la solución presentada en este trabajo consta de tres elementos principales:

- **Modelo hardware:** permite definir el hardware a usar.

- **Modelo software:** permite definir el juego de instrucciones a utilizar.
- **Kernel de simulación:** simula el funcionamiento del hardware ejecutando el microcódigo/lenguaje máquina definido con anterioridad.

El modelo hardware permite definir los distintos elementos típicos de un computador (memoria principal, procesador, etc.) de una forma modular. La forma de definir estos elementos equilibra dos objetivos contrapuestos: es suficientemente completa como para imitar los principales aspectos de la realidad, pero es lo suficientemente mínima para facilitar su uso. Ante todo se persigue que sea una herramienta didáctica.

El modelo software permite definir el microcódigo y el ensamblador basado en este microcódigo de la forma más intuitiva posible. El ensamblador a usar viene dado por un conjunto de instrucciones que puede ser definido por el usuario e intenta ser lo suficientemente flexible como para poder definir diferentes tipos y juegos de instrucciones, como por ejemplo MIPS o ARM.

El tercer elemento de la arquitectura propuesta es un kernel que toma como entrada el modelo hardware descrito y el modelo software de trabajo, y se encarga de mostrar el funcionamiento del hardware con el software dado.

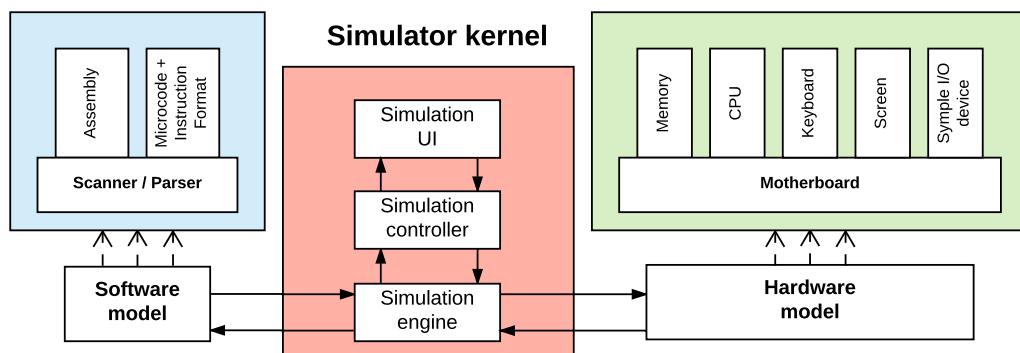


Figura 4-1: Arquitectura de WepSIM.

La Figura 4-1 resume la arquitectura de WepSIM. El punto de inicio es el modelo hardware que describe el procesador a ser simulado. Ello incluye el procesador, la memoria y algunos dispositivos de E/S: teclado, pantalla y un dispositivo de E/S simple que genera interrupciones. El modelo hardware describe el estado global del procesador. A partir del estado global del procesador, el kernel de simulación actualiza el estado en cada ciclo de reloj.

La unidad de control simulada almacena las señales de control de cada ciclo en una memoria

de control. La memoria de control tiene todos los microprogramas para las instrucciones con las que trabaja el procesador, y el fetch para leer la instrucción de memoria y decodificarla.

El microcódigo (el contenido de la memoria de control) junto con el formato de cada instrucción (campos de la instrucción y su longitud) se describe en un fichero de texto. El modelo software lee este fichero, lo traduce a binario y lo carga en el procesador. La definición del lenguaje ensamblador a utilizar se describe junto con el microcódigo, y el modelo software permite traducir a binario programas escritos en dicho ensamblador.

El kernel de simulación pregunta al subsistema del modelo software por el microcódigo definido, la descripción del formato de instrucción y el contenido de la memoria principal. Los binarios se cargan en los elementos del modelo hardware, y a continuación el kernel de simulación actualiza el estado global en cada ciclo de reloj.

WepSIM dispone de un controlador de simulación que se encarga de actualizar el ciclo de reloj y mostrar el estado global. El subsistema de interfaz de simulación actualiza la interfaz de usuario. Cuando el usuario usa la interfaz de usuario para solicitar una operación, el subsistema de interfaz de simulación traslada la petición al controlador de simulación. De este modo, se usa un Modelo- Vista-Controlador (MVC) básico para la arquitectura de WepSIM.

4.2.1 Modelo hardware

El modelo hardware que usa WepSIM permite definir los distintos elementos típicos de un computador (memoria principal, procesador, etc.) de una forma modular y de manera que sea posible añadir, quitar o modificar estos elementos.

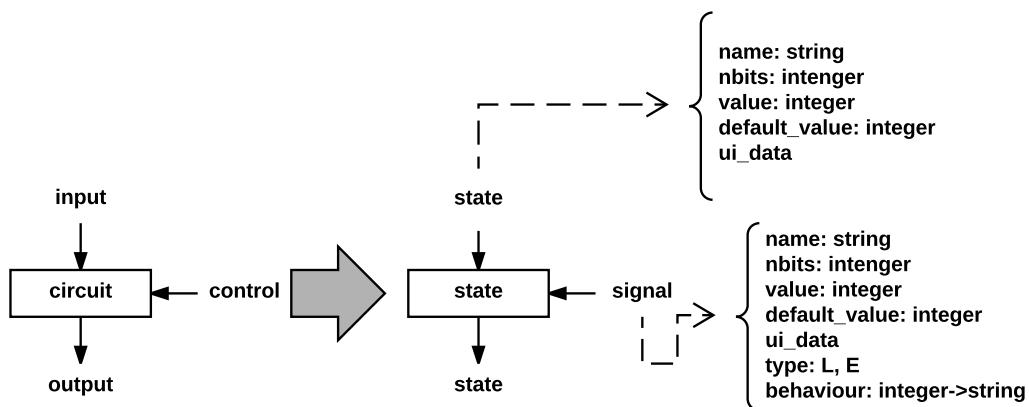


Figura 4-2: Modelado del hardware.

La figura 4-2 introduce el modelo propuesto. Cada elemento del circuito se describe como

una caja negra con posibles entradas, posibles salidas y señales de control (que controlan las posibles transformaciones de las entradas a las salidas). El subsistema del modelo hardware transforma esta caja negra en dos conjuntos de objetos: estados y señales. Un estado tiene un identificador (el nombre), el valor (un valor entero) y un valor inicial (el valor por defecto). Los valores que puede tomar son valores naturales dentro de un rango, dado por el número de bits con los que se representa el estado. Una señal es un estado especial que controla el valor de otros estados o señales. Hay dos atributos asociados a las señales (y no a los estados): el tipo de señal (por nivel o por flanco) y su comportamiento. Para cada valor de señal una cadena de caracteres describe en un lenguaje simple lo que la señal mueve o transforma. Este lenguaje simple se compone principalmente de instrucciones que representan las operaciones elementales.

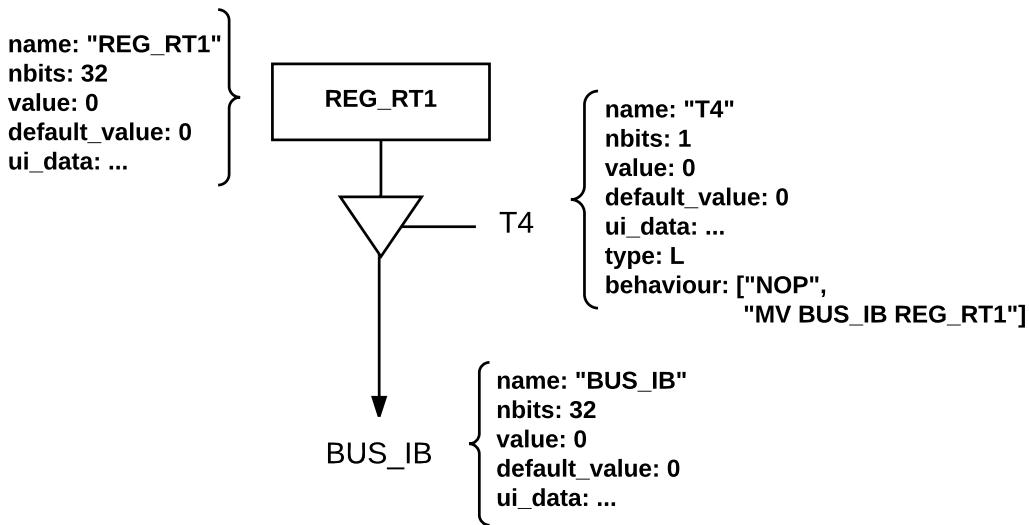


Figura 4-3: Ejemplo de modelado de una puerta triestado.

Las Figuras 4-3 y 4-4 muestran dos ejemplos: un triestado y un registro. El triestado controla dos estados: el estado del bus al que se conecta *BUS IB* y el estado del registro de entrada, *REG RT1* en este caso. Ambos representan el valor a la salida de la puerta (*BUS IB*) y el valor del registro *RT1* (*REG RT1*). La señal *T4* se encarga de indicar cuándo el valor del registro *RT1* se envía a la salida. Esta señal *T4* es una señal por nivel (tipo: L), con valor cero no tiene efecto (comportamiento “NOP”). Cuando el valor de la señal es uno entonces el comportamiento es el de copiar el valor del registro *RT1* a la salida (comportamiento “*MV BUS IB REG RT1*”).

El ejemplo con el registro (Figura 4-4) es similar. En este caso se trabaja con dos estados: el

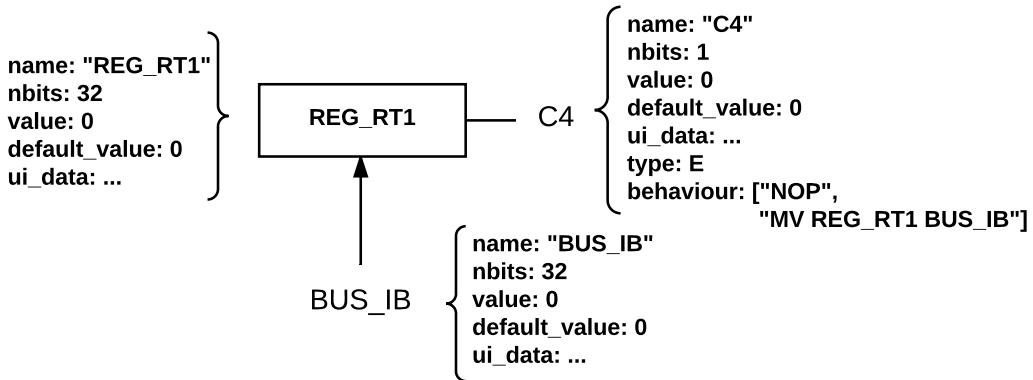


Figura 4-4: Ejemplo de modelado de un registro.

contenido del registro *RT1* y el contenido situado a la entrada (*BUS IB*). La señal *C4* controla cuándo se almacena en el registro *RT1* el valor que hay en la entrada. La diferencia está en el tipo de señal: *C4* es una señal por flanco de bajada (tipo: *E*), por lo que al final del ciclo de reloj (pasa de uno a cero) si la señal vale uno entonces el comportamiento es el de copiar el valor situado a la entrada al registro (comportamiento “*MV REG RT1 BUS IB*”).

El lenguaje simple usado para definir los comportamientos añade a las operaciones elementales otras operaciones necesarias. Por ejemplo disparar una señal (“*FIRE C4*”) que ayuda a propagar el efecto de una señal al re-evaluar la señal inmediata que podría verse afectada. Otro ejemplo lo encontramos en dos operaciones que pueden ser muy útiles a la hora de depurar: imprimir el valor de un estado (“*PRINT E BUS IB*”) e imprimir el valor de una señal (“*PRINT S C4*”).

4.2.2 Modelo software

Una vez definido el procesador elemental usando el modelo hardware propuesto, toca describir el conjunto de instrucciones que es capaz de ejecutar así como el microcódigo que lo orquesta. En un fichero de texto se define el formato de las instrucciones máquina junto con el cronograma asociado a la ejecución de cada una de las instrucciones máquina. La Figura 4-5 muestra un ejemplo de definición para la instrucción *li* (*load immediate*), que almacena un valor inmediato en un registro.

El fichero con el cronograma de fetch y todos los cronogramas de las instrucciones define el microcódigo para la plataforma WepSIM. El simulador permite la definición de diferentes

```

li reg val
{
    co=000010,
    nwords=1,
    reg=reg(25,21),
    val=inm(15,0),
    {
        (SE=0, OFFSET=0, SIZE=10000, SE=1, T3=1,
         LE=1, MR=0, SELE=10101, A0=1, B=1, C=0)
    }
}

```

Figura 4-5: Ejemplo de formato de instrucción.

juegos y formatos de instrucciones. Inicialmente se ha implementado un subconjunto de las instrucciones del MIPS, pero es posible definir instrucciones de otros conjuntos de forma similar. En este fichero se pueden asignar códigos simbólicos a los registros del banco de registros, lo que permite que en los programas escritos en ensamblador se puedan usar dichos símbolos (por ejemplo, registro `$t3` en la Figura 4-6).

```

.text
main: li $t3 8
      li $t5 10
      add $t6 $t3 $t5

```

Figura 4-6: Ejemplo de código fuente en ensamblador.

El campo “`co`” identifica el código de instrucción máquina, que es un número binario de 6 bits. Esto permite definir hasta 64 instrucciones distintas. Dado que los últimos 4 bits de la instrucción pueden usarse para seleccionar la operación en la ALU, es posible seleccionar hasta 16 operaciones aritmético-lógicas con un mismo código de instrucción, por lo que se podrían tener 79 (63+16) instrucciones en total.

Cuando WepSIM carga el microcódigo, cada código de instrucción tiene asociado una dirección de comienzo en la memoria de control donde se almacena el cronograma asociado. Esta tabla con dos columnas (el código de instrucción y su dirección de comienzo asociada en la memoria de control) se carga en la ROM `co2microAddr` mostrada en la Figura 2-8.

El campo “`nwords`” define cuantas palabras precisa la instrucción para su definición y carga en memoria. Una palabra en WepSIM son 4 bytes.

Para cada campo de la instrucción se define el bit inicial, el bit final (ambos incluidos) y el

tipo de campo (registro, valor inmediato, dirección absoluta y dirección relativa a PC). Una vez definido el formato, se definen todas las microinstrucciones que necesita la instrucción máquina definida para su ejecución. Todas las microinstrucciones se encuentran encerradas entre llaves y cada microinstrucción está formada por una lista de tuplas (señal, valor) encerradas entre paréntesis. Para la instrucción definida en la Figura 4-5 se precisa de una sola microinstrucción, en la que se indican qué señales se activan durante un ciclo de reloj. Para las señales no indicadas se asume que su valor es 0 durante el ciclo de reloj correspondiente.

Una vez cargado el microcódigo en WepSIM, es posible cargar cualquier fichero ensamblador que haya sido codificado usando las instrucciones máquina definidas anteriormente en el microcódigo.

En la Figura 4-6 se muestra un ejemplo de código fuente en ensamblador que se puede usar en WepSIM. Este ejemplo en particular muestra un código estilo MIPS. Para que un programa en ensamblador pueda utilizar la instrucción de carga inmediata *li* (*load immediate*) y de suma *add* (*addition*), deben haber sido definidas previamente en el microcódigo. WepSIM puede comprobar los errores de sintaxis y construir el binario mediante el relleno de los campos descritos en la definición del microcódigo correspondiente a la instrucción. La Figura 4-7 muestra un ejemplo de traducción a binario para la instrucción *li \$2 5* en función del formato definido en la Figura 4-5. También se debe haber definido en el fichero de microcódigo el valor del registro asociado a la etiqueta *\$2* (00100 en este caso).

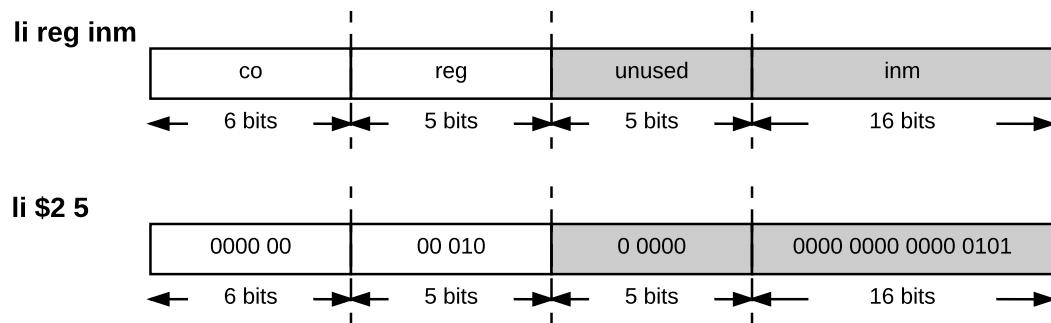


Figura 4-7: Formato de instrucción descrito en el microcódigo y ejemplo de su traducción en binario.

Una de las grandes ventajas del simulador WepSIM es que no está limitado a un conjunto de instrucciones concreto. Se puede definir un amplio conjunto de instrucciones de procesadores reales o inventados. Se puede usar para añadir por ejemplo, a un conjunto de instrucciones MIPS, otras instrucciones diferentes no incluidas en dicho conjunto de instrucciones.

4.2.3 Kernel del simulador

El kernel del simulador, es el componente que integra la interfaz de usuario del simulador, el controlador del simulador, y el motor de simulación, siendo el encargado de hacer las funciones de controlador de la herramienta gestionando las comunicaciones de la interfaz de usuario con los modelos hardware y software.

De esta forma cuando el usuario interacciona con la interfaz gráfica, el módulo encargado de la interfaz genera eventos que identifican las acciones que el usuario realiza. Estos eventos son generados a modo de peticiones al controlador de la simulación, que se encarga de identificar el evento generado, comprobar el contexto en el cual se encuentra, y enviar la acción correspondiente a realizar al motor del simulador. El motor del simulador, se encarga de comunicarse con los modelos hardware y software, de forma que estos puedan ejecutar correctamente la acción solicitada (Figura 4-8).

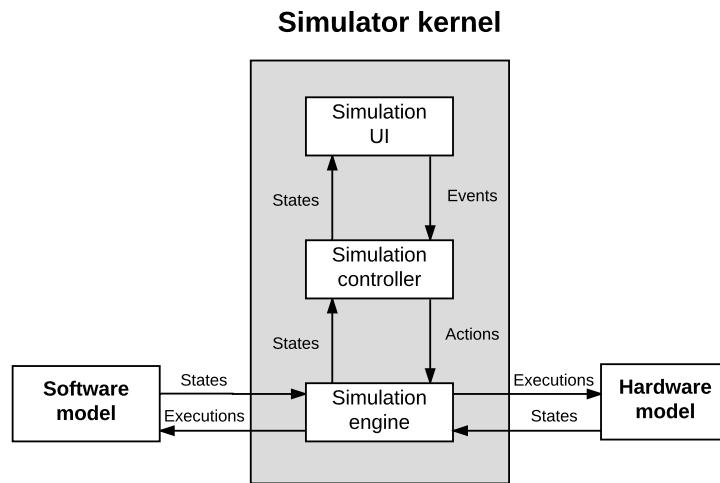


Figura 4-8: Arquitectura del kernel del simulador.

Por ejemplo, si el usuario selecciona la ejecución de un ciclo de reloj, la interfaz de usuario genera el evento correspondiente. El controlador del simulador, se encarga de identificar el evento, verificar que se cumplen todas las condiciones para poder ejecutarse un ciclo de reloj comprobando diferentes valores del estado del simulador. En caso de poder ser realizada la ejecución del ciclo de reloj, envía la orden al motor del simulador que se encarga de comunicarse con el modelo hardware, realizando la ejecución de un ciclo de reloj actualizando los componentes del modelo hardware. Por último, la interfaz de usuario consulta el estado del modelo hardware actualizando la información necesaria para ser mostrada al usuario.

Capítulo 5

Implementación y despliegue

Este capítulo trata la implementación y despliegue del software. En cuanto a la implementación del sistema, se explican las partes más complicadas del código en (Sección 5.1, *Implementación*). Por otro lado, explicamos los pasos necesarios para desplegar el sistema final (Sección 5.2, *Despliegue*)

5.1 Implementación

Como hemos explicado en el Capítulo 3, *Análisis*, hemos implementado el simulador utilizando el lenguaje de programación JavaScript junto con HTML, CSS y las bibliotecas/frameworks JQuery, JQueryUI, JQuery Mobile, Knockout y BootStrap. El motor de simulación es el encargado de ejecutar cada uno de los ciclos de reloj del simulador, tomando como entradas tanto el modelo hardware como el modelo software, pero el desarrollador ha debido de diseñar e implementar el algoritmo que posibilita esta ejecución.

Además, hemos trabajado en conseguir una herramienta que sea capaz de generar la memoria de control mediante la definición del juego de instrucciones por parte del usuario, y de generar el código binario asociado al código ensamblador definido por el usuario; el cual depende del juego de instrucciones definido previamente. Para ello, se han diseñado e implementado dos compiladores diferentes, capaces de generar los binarios correspondientes además de las estructuras de datos necesarias para ayudar al motor de simulación a lo largo de la ejecución.

Como resultado de este diseño e implementación, el código fuente de la herramienta está compuesto por una serie de ficheros que definen las funcionalidades y componentes anteriormente citados. Por ello, es necesario explicar la estructura de ficheros que componen el simu-

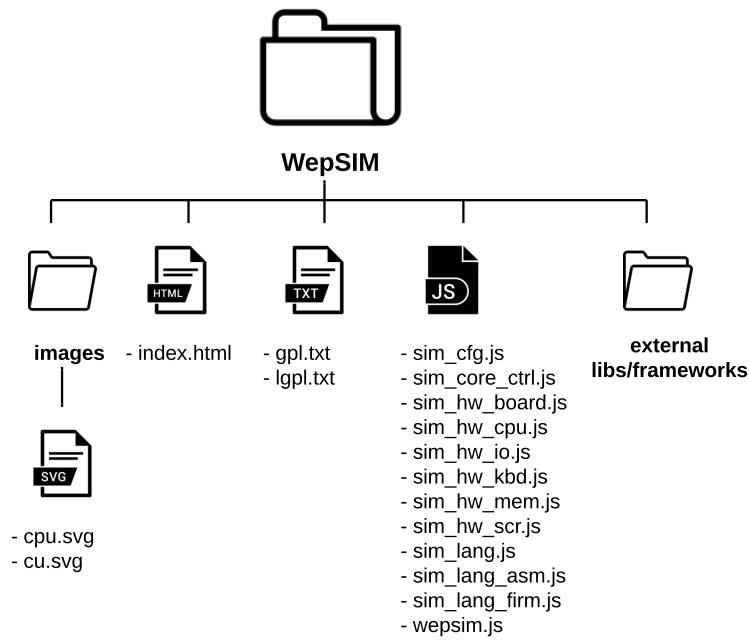


Figura 5-1: Estructura de ficheros.

lador y las dependencias que existen entre sí. De esta forma, en la Figura 5-1 podemos ver los ficheros que componen la herramienta web, los cuales tienen una serie de dependencias indicadas en la Figura 5-2.

Los ficheros que componen WepSIM son detallados a continuación:

- **index.html:** este fichero se encarga de la vista de la herramienta, generando la interfaz de usuario de la herramienta.
- **gpl.txt y lgpl.txt:** estos ficheros contienen la especificación de las licencias del software.
- **sim_cfg.js** este fichero contiene las estructuras de datos de la configuración de la herramienta.
- **sim_core_ctrl.js:** este fichero contiene la implementación del motor de simulación de la herramienta.
- **sim_core_ui.js:** este fichero contiene la implementación del motor de la interfaz de usuario de la herramienta.
- **sim_hw_cpu.js:** este fichero contiene la definición del modelo hardware de la cpu del simulador.

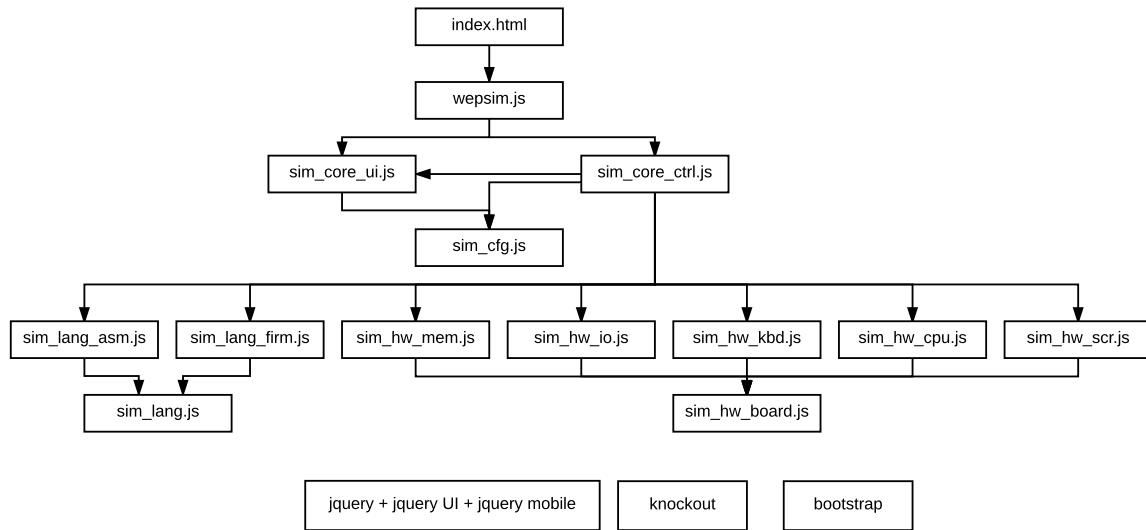


Figura 5-2: Dependencias entre ficheros.

- **sim_hw_io.js:** este fichero contiene la definición del modelo hardware del módulo de generación de interrupciones del simulador.
- **sim_hw_kbd.js:** este fichero contiene la definición del modelo hardware del módulo del teclado del simulador.
- **sim_hw_scr.js:** este fichero contiene la definición del modelo hardware de la memoria principal del simulador.
- **sim_hw_mem.js:** este fichero contiene la definición del modelo hardware de la pantalla del simulador.
- **sim_lang.js:** este fichero contiene la implementación de las funciones principales del parser de ficheros de la herramienta.
- **sim_lang_asm.js:** este fichero contiene la implementación del compilador de código ensamblador de la herramienta.
- **sim_lang_firm.js:** este fichero contiene la implementación del compilador de microcódigo de la herramienta.
- **images/cpu.svg:** este fichero contiene la definición de la imagen vectorial de la cpu de la herramienta.

- **images/cpu.svg:** este fichero contiene la definición de la imagen vectorial de la unidad de control de la herramienta.
- **external folder:** este directorio contiene las bibliotecas y frameworks que necesita el simulador para su correcto funcionamiento, como son JQuery, JQueryUI, JQuery Mobile, Knockout y BootStrap.

Para comprender funcionamiento tanto de los dos compiladores utilizados (compilador de microcódigo y compilador de ensamblador) como del kernel del simulador, se puede observar en el Pseudocódigo 5.1, mientras que en el Pseudocódigo 5.2 podemos ver el proceso de compilación del código ensamblador definido por el usuario en función de la memoria de control previamente generada, detallando el proceso de compilación del segmento de datos en el Pseudocódigo 5.3 y el proceso de compilación del segmento de texto en el Pseudocódigo 5.4. En el Pseudocódigo 5.5 podemos ver como se realiza una simulación, realizando la comprobación de segmentos de memoria, tipo de simulación, y ejecución del ciclo o instrucción correspondiente.

El proceso de compilación del juego de instrucciones (Pseudocódigo 5.1) se realiza mediante un análisis sintáctico basado en la técnica *LEX and YACC*, que queda perfectamente descrita en [16]. De este modo, el proceso de compilación queda dividido en tres bloques importantes: instrucciones, nombrado de registros y pseudoinstrucciones. En primer lugar, se realiza la lectura del bloque de instrucciones, en donde se comprueba cada uno de los campos y parámetros que la conforman para posteriormente generar el binario asociado a cada ciclo de reloj. Después, se realiza la lectura del banco de registros, donde se forma el par *id,nombre*, pudiendo realizar el nombrado deseado de los distintos registros que lo componen. Por último, se realiza la lectura de pseudoinstrucciones. Estas instrucciones no pertenecen al lenguaje, sino que están compuestas por instrucciones del lenguaje, pudiendo el usuario utilizar técnicas de alto nivel en donde con una única línea de código, genere más de una instrucción.

Pseudocódigo 5.1 Proceso de compilación del juego de instrucciones.

```
1: function COMPILE_FIRMWARE(context )
2:   token = getToken();
3:   firmware = null;
4:   instructions= null;
5:   registers = null;
6:   pseudoInstructions = null;
7:   while (!is_registers(token) and token != null) do
8:     instruction = null;
9:     instructions[signature]=readSignature(getNextToken());
10:    instructions[co]=readCo(getNextToken());
11:    if (getNextToken() == COP) then
12:      instruction[cop]=readCop(getNextToken());
13:    end if
14:    while (is_Field(getNextToken()) do
15:      instructions[field].add=readField(getToken());
16:    end while
17:    while (is_microcode(getNextToken()) do
18:      instructions[microcode].add=readCycle(getToken());
19:    end while
20:    firmware.add(instructions);
21:  end while
22:  if (is_registers(getToken()) then
23:    registers = readRegisters(getToken());
24:  end if
25:  if (is_pseudoInstructions(getToken()) then
26:    pseudoInstructions = readPseudoInstructions(getToken());
27:  end if
28:  firmware.add(instructions);
29:  firmware.add(registers);
30:  firmware.add(pseudoInstructions);
31:  Return firmware;
32: end function
```

El proceso de compilación del código ensamblador (Figura 5.1) está basado, al igual que el compilador del juego de instrucciones, en la técnica de compilación *LEX and YACC*. Este proceso, está dividido en dos etapas: segmentos de datos y segmento de texto.

En la etapa de compilación del segmento de datos, se realiza el proceso de lectura y compilación de las diferentes directivas definidas por el usuario. En primer lugar, se realiza la identificación de las diferentes etiquetas que identifican a las directivas y que serán utilizadas posteriormente en el segmento de texto. Una vez identificadas estas etiquetas, se procede a la generación del binario asociado a este segmento, el cual depende directamente del tipo de directiva utilizado y del tamaño de la directiva.

Pseudocódigo 5.2 Proceso de compilación de código ensamblador.

```
1: function COMPILE_ASSEMBLY( context,datosCU)  
2:   dataSegment = null;  
3:   textSegment = null;  
4:   mainMemory = null;  
5:   if (!is_data_segment(getToken())) then  
6:     Return;  
7:   end if  
8:   dataSegment = read_data_segment(context[dSegment]);  
9:   if (!is_text_segment(getToken())) then  
10:    Return;  
11:   end if  
12:   textSegment = read_text_segment(context[tSegment]);  
13:   mainMemory.add(dataSegment);  
14:   mainMemory.add(textSegment,dataSegment.labels);  
15:   Return mainMemory;  
16: end function
```

En la etapa de compilación del segmento de texto, se realiza el proceso de compilación de las diferentes instrucciones ensamblador definidas por el usuario. Por ello, es necesario el uso del juego de instrucciones definido previamente, ya que la generación del binario de la instrucción depende de la definición previa. En primer lugar, se realiza la búsqueda de las diferentes etiquetas empleadas en este segmento, identificando las nuevas etiquetas definidas a las anteriores definidas en el segmento de datos, para la cambiar la etiqueta por el valor de la dirección de memoria asociada a la etiqueta. Una vez realizado este proceso, se procede a la

compilación de cada una de las instrucciones, buscando para cada instrucción del segmento de texto una instrucción definida en el juego de instrucciones que cumple con la sintaxis. En caso de existir la instrucción, se procede a la generación del binario asociado, comprobando el valor de cada uno de los campos. Este proceso se realiza para cada una de las operaciones definidas en este segmento. Una vez traducidas todas las instrucciones, se genera la memoria principal asociada al código definido.

Por último, se indica el proceso de simulación de la herramienta. Este proceso es realizado en el kernel del simulador como ha sido explicado anteriormente (Sección 4.2.3). Como podemos observar en el Pseudocódigo 5.5, en primer lugar se realiza la petición de ejecución. El controlador del simulador, comprobaría el contexto (*breakpoints*, final de segmento, etc.) verificando si la ejecución es posible. En caso de ser posible, identificaría el tipo de simulación solicitada, llamando a la función correspondiente. Si la ejecución solicitada es de una microinstrucción, el motor de simulación únicamente generaría un ciclo de reloj, enviándolo al modelo hardware para su ejecución y actualizando posteriormente los valores mostrados en la interfaz de usuario. La ejecución de una instrucción completa se realiza ejecutando en bucle la secuencia de microinstrucciones correspondientes hasta que el registro de microdirección apunta a la dirección de comienzo del ciclo *FETCH*, que indicaría el comienzo de la próxima instrucción, o apunta a una dirección de memoria fuera del rango del segmento de texto, lo cual indica que la simulación del código ensamblador ha finalizado.

En [17] se presenta el manual completo de usuario de la herramienta, que incluye la especificación del modelo hardware implementado y la explicación de uso del simulador, indicando algunos ejemplos docentes para aprender el uso de esta herramienta y poder comprender mejor el funcionamiento de los distintos componentes que la forman.

Pseudocódigo 5.3 Proceso de compilación de código ensamblador: segmento de datos.

```
1: function READ_DATA_SEGMENT(context)
2:   labels = null;
3:   token = getToken();
4:   while (!is_End(getNextToken()) do                                ▷ Check Labels
5:     if (isLabel(getToken())) then
6:       labels.add(getLabel(getToken()));
7:     end if
8:   end while
9:   dataSegment[labels].add(labels);
10:  token = getFirstToken();
11:  for (i in context[directives]) do                               ▷ Read Directives
12:    possible_datatype = i;
13:    if (possible_datatype == ".byte") then dataSegment[directives].add(readByte(i));
14:    end if
15:    if (possible_datatype == ".half") then dataSegment[directives].add(readHalf(i));
16:    end if
17:    if (possible_datatype == ".word") then dataSegment[directives].add(readWord(i));
18:    end if
19:    if (possible_datatype == ".space") then dataSegment[directives].add(readSpace(i));
20:    end if
21:    if (possible_datatype == ".ascii") then dataSegment[directives].add(readAscii(i));
22:    end if
23:    if (possible_datatype == ".asciiz") then dataSegment[directives].add(i);
24:    end if
25:    if (possible_datatype == ".align") then dataSegment[directives].add(readAlign(i));
26:    end if
27:  end for
28:  Return dataSegment
29: end function
```

Pseudocódigo 5.4 Proceso de compilación de código ensamblador: segmento de texto.

```
1: function READ_TEXT_SEGMENT(context,labels)
2:   for (i in labels) do                                ▷ Read Labels
3:     replaceLabel(context, i);
4:   end for
5:   for (i in context[instructions]) do                ▷ Read Instructions
6:     auxInstruction = fillFields(i);
7:     auxBinary = null;
8:     if (!isInstruction(auxInstruction[0])) then      ▷ Check if instruction exist
9:       Return;
10:    end if auxBinary = generateBinary(auxInstruction);
11:    if (auxBinary == null) then
12:      Return;
13:    end if textSegment.add(auxBinary);
14:   end for
15:   Return textSegment
16: end function
```

Pseudocódigo 5.5 Proceso de simulación.

```
1: function RUN_SIMULATION( )
2:   if (!possible_execute()) then
3:     Return;
4:   end if
5:   change_play_button();
6:   execute_simulation_inchain();
7: end function

8: function EXECUTE_SIMULATION_INCHAIN( )
9:   if (end_code()) then
10:    Return;
11:   end if
12:   if (is_breakpoint()) then
13:     Return;
14:   end if
15:   if (execute_mode == microInstruction) then
16:     execute_microInstruction();
17:   end if
18:   if (execute_mode == instruction) then
19:     execute_instruction();
20:   end if
21: end function

22: function EXECUTE_MICROINSTRUCTION( )
23:   if (!possible_execute()) then
24:     Return;
25:   end if
26:   compute_general_behaviour(CLOCK);           ▷ Execute Cycle
27:   update_UI();
28: end function

29: function EXECUTE_INSTRUCTION( )
30:   if (!possible_execute()) then
31:     Return;
32:   end if
33:   do
34:     compute_general_behaviour(CLOCK);           ▷ Execute Cycle
35:     while (reg_microaddr!=0 and mem_control[microAddr]!=undefined) ▷ Check Next Cycle
36:     update_UI();                                ▷ Update User Interface
37:   end function
```

5.2 Despliegue

En esta sección se presenta el despliegue de la herramienta, el cual queda dividido en tres partes: especificaciones técnicas recomendadas, el proceso de despliegue del software para que sea accesible, y por último el proceso de ejecución de la herramienta.

Las especificaciones técnicas recomendadas para que el usuario final obtenga la mejor experiencia posible con la herramienta:

- **Sistema Operativo:** Ubuntu 16.04.2 LTS (*Linux distribution*) /Windows 10 / MacOS 10.12.5.
- **Procesador:** Intel(R) Core(TM) i3 CPU 6300 @3.8GHz o superior.
- **Random-Access Memory (RAM):** 2 GB o superior.
- **Almacenamiento:** 1 GB de espacio libre en el disco duro (recomendado para el navegador web).
- **Red:** La conexión a internet no es necesaria para la ejecución de la herramienta, únicamente para el acceso a ella.
- **Software:** Los siguientes navegadores web son los recomendados para el uso de la herramienta:
 1. Mozilla Firefox 45+.
 2. Google Chrome 50+.
 3. Microsoft Edge 30+.
 4. Safari 10+.

Para el proceso de despliegue de la herramienta, se requiere que el usuario disponga de un servidor web *Apache* o *Nginx* instalado en su computador para que el software sea accesible. Una vez instalado el servidor web, se deben de seguir los siguientes pasos:

1. Acceder al directorio establecido en el servidor web para alojar el simulador.
2. Clonar el repositorio del simulador para obtener el código fuente del simulador:
`git clone https://github.com/wepsim/wepsim.git`
3. Reiniciar el servicio del servidor web para que realice los cambios pertinentes.

Por último, el usuario podrá acceder a la herramienta mediante un navegador web de los citados anteriormente. Para ello, el usuario deberá de indicar en el navegador la dirección en la cual ha sido publicada la herramienta, no siendo necesaria la conexión a internet en caso de estar publicada en el mismo computador (*localhost*).

En caso de no desear el usuario realizar este proceso, puede acceder a la herramienta de forma inmediata accediendo a la dirección:

[https://wepsim.github.io/wepsim/.](https://wepsim.github.io/wepsim/)

Capítulo 6

Verificación, validación y evaluación

Este capítulo detalla la verificación, validación y evaluación del proyecto. En primer lugar, presentamos la verificación y validación del simulador (Sección 6.1, *Verificación y validación*), y detallamos una serie de pruebas que nos permitieron verificar que habíamos cumplido todos los requisitos establecidos en el Capítulo 3(*Análisis*). Después de esto, mostramos la validación de los resultados de las simulaciones, demostrando que el simulador realiza simulaciones precisas y realistas.

Para la realización de las simulaciones, hemos tomado la definición de un juego de instrucciones base proporcionado por el coordinador de la asignatura Estructura de Computadores, pudiendo así validar que el funcionamiento del simulador es correcto al obtener los resultados esperados en cada una de las instrucciones y códigos simulados en la herramienta.

6.1 Verificación y validación

El principal objetivo de esta sección es verificar que todos los requisitos detallados en el Capítulo 3 (*Análisis*) han sido cumplidos. Además, validamos los resultados obtenidos con WepSIM, comparándolos con los resultados teóricos esperados de la definición del juego de instrucciones de la asignatura Estructura de Computadores y los ejercicios especificados en el libro de la asignatura [11].

En la Ingeniería del Software, la verificación y validación son los procesos de comprobar que un sistema de software cumple con las especificaciones y que cumple con su propósito. Como se explica en el Capítulo 3 (*Análisis*), el cliente fija inicialmente los requisitos deseados para el producto final (requisitos del usuario). A partir de ahí, los analistas especifican los requisitos de

software (requisitos funcionales y no funcionales). Para verificar que se cumplen los requisitos del proyecto, se necesitan procesos de verificación y validación (ver Figura 6-1).

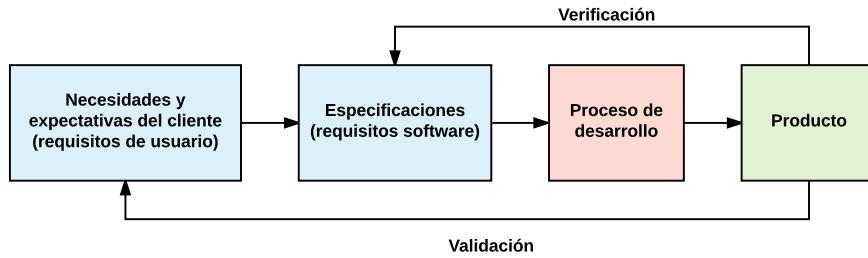


Figura 6-1: *Verificación y validación software*.

Verificación Software Es el proceso de evaluación de los productos de trabajo (no el producto final real) de una fase de desarrollo para determinar si cumplen con los requisitos especificados para esa fase (los requisitos de software). *Validación software* es el proceso de evaluación del producto final al final del proceso de desarrollo para determinar si satisface los requisitos especificados por el usuario al inicio del proyecto [18].

6.1.1 Pruebas de verificación

Con el fin de realizar las pruebas de verificación, hemos seguido un proceso dinámico durante la fase de desarrollo del software. Con estas pruebas queríamos responder a la pregunta: “*¿Estamos construyendo el producto correctamente?*”. La Tabla 6.1 proporciona la plantilla utilizada para los test de verificación. Es necesario tener en cuenta que el formato del atributo ID es VET-XX, donde XX indica el número de test de verificación.

Tabla 6.1: *Plantilla de pruebas de verificación.*

ID	Test ID.
Nombre	Nombre del test.
Requisitos	Requisitos de software cumplidos con este test.
Descripción	Descripción del test.
Precondiciones	Las condiciones que siempre deben ser verdad antes de realizar el test.
Procedimiento	Una secuencia fija, paso a paso, de las actividades realizadas por el test.
Postcondiciones	Las condiciones que siempre deben ser verdaderas justo después de realizar el test.
Evaluación	<i>OK o Error.</i>

A continuación, se especifican las pruebas de verificación.

Tabla 6.2: *Test de verificación VET-01.*

ID	VET-01.
Nombre	Plataforma.
Requisitos	SR-NF-PL01, SR-NF-PL02
Descripción	Verificar que el software puede ser utilizado en los navegadores especificados en los requisitos.
Precondiciones	<ol style="list-style-type: none"> 1. Utilizar un computador con sistema operativo Ubuntu 16.04 / Windows 10 / MacOS 10.2.5 . 2. Tener instalados los navegadores Microsoft Edge 30+, Mozilla Firefox 45+, Google Chrome 50+ o Safari 10+.
Procedimiento	<ol style="list-style-type: none"> 1. Abrir el software en cualquiera de los navegadores web especificados. 2. Generar microcódigo. 3. Generar memoria principal. 4. Ejecutar simulación.
Postcondiciones	La simulación ha sido completada satisfactoriamente sin ningún error de ejecución debido a la plataforma .
Evaluación	OK

Tabla 6.3: *Test de verificación VET-02.*

ID	VET-02.
Nombre	Ejecución local.
Requisitos	SR-NF-PL03, SR-NF-PL04
Descripción	Verificar que el software es ejecutado en local sin la necesidad de conexión a internet.
Precondiciones	<ol style="list-style-type: none"> 1. No tener acceso a Internet. 2. Tener el código fuente de la herramienta descargado en local.
Procedimiento	<ol style="list-style-type: none"> 1. Abrir el software en cualquiera de los navegadores web especificados. 2. Generar microcódigo. 3. Generar memoria principal. 4. Ejecutar simulación.
Postcondiciones	La simulación ha sido completada satisfactoriamente sin ningún error de ejecución debido a la conexión de red.
Evaluación	OK

Tabla 6.4: *Test de verificación VET-03.*

ID	VET-03.
Nombre	Lenguaje de programación HTML5.
Requisitos	SR-NF-PL05
Descripción	Verificar que el software ha sido desarrollando utilizando HTML5.
Precondiciones	1. Tener el código fuente de la herramienta descargado en local.
Procedimiento	1. Comprobar el código de los archivos fuente del simulador.
Postcondiciones	El simulador ha sido desarrollado mediante el lenguaje de programación HTML5.
Evaluación	OK

Tabla 6.5: *Test de verificación VET-04.*

ID	VET-04.
Nombre	Interfaz de usuario.
Requisitos	SR-NF-UI01
Descripción	Verificar que la interfaz del simulador es compatible tanto con pc como con plataformas móviles.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un pc y de un dispositivo móvil. 2. Los dispositivos deben tener instalados los navegadores web especificados previamente.
Procedimiento	<ol style="list-style-type: none"> 1. Abrir el software en cualquiera de los navegadores web especificados. 2. Generar microcódigo. 3. Generar memoria principal. 4. Ejecutar simulación.
Postcondiciones	La interfaz del simulador es compatible tanto como PC como con dispositivos móviles, visualizándose correctamente en cualquier pantalla de la herramienta.
Evaluación	OK

Tabla 6.6: *Test de verificación VET-05.*

ID	VET-05.
Nombre	Tiempo medio por ciclo de reloj.
Requisitos	SR-NF-P01
Descripción	Verificar que el tiempo medio por ciclo de reloj del simulador no supera 0,1 segundos.
Precondiciones	1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente.
Procedimiento	1. Abrir el software en cualquiera de los navegadores web especificados. 2. Generar microcódigo. 3. Generar memoria principal utilizando un código ensamblador que utiliza cada una de las partes de la arquitectura del simulador. 4. Ejecutar simulación midiendo el tiempo de ejecución.
Postcondiciones	El tiempo de ejecución medio por ciclo de reloj excede 0,1 segundos.
Evaluación	OK

Tabla 6.7: *Test de verificación VET-06.*

ID	VET-06.
Nombre	Arquitectura del simulador.
Requisitos	SR-F-F01, SR-F-02, SR-F-03
Descripción	Verificar que el software simula una arquitectura de 32 bits con unidad de control microprogramable (con secuenciamiento implícito, saltos a nivel de microdirección y microsaltos condicionales).
Precondiciones	1. Tener el código fuente de la herramienta descargado en local. 2. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente.
Procedimiento	1. Comprobar el código de los archivos fuente del simulador que contienen el modelo hardware a simular (<code>sim_hw_*.js</code>). 2. Generar microcódigo que define instrucciones de acceso a memoria, operaciones con el banco de registros, uso de los buses de la arquitectura del simulador, saltos a nivel de microdirección y microsaltos condicionales. 3. Generar memoria principal utilizando un código ensamblador que utiliza cada una de las partes de la arquitectura del simulador y las instrucciones de prueba definidas. 4. Verificar que los resultados obtenidos de la ejecución son los resultados teóricos esperados.
Postcondiciones	El modelo hardware ha sido definido correctamente, obteniendo los resultados esperados tras la simulación.
Evaluación	OK

Tabla 6.8: *Test de verificación VET-07.*

ID	VET-07.
Nombre	Definición de juego de instrucciones.
Requisitos	SR-F-F04, SR-F-F05, SR-F-F09, SR-F-F10, SR-F-F11, SR-F-F15
Descripción	Verificar que el software permite la definición del juego de instrucciones en el formato utilizado en la asignatura Estructura de Computadores.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente. 2. Disponer de un juego de instrucciones base en el formato utilizado en la asignatura Estructura de Computadores. 3. Disponer de un código ensamblador que utiliza el juego de instrucciones utilizado en la asignatura Estructura de Computadores.
Procedimiento	<ol style="list-style-type: none"> 1. Generar microcódigo mediante el juego de instrucciones proporcionado. 2. Generar memoria principal utilizando el código ensamblador proporcionado. 3. Ejecutar simulación. 4. Verificar que los resultados obtenidos de la ejecución son los resultados teóricos esperados.
Postcondiciones	La memoria de control y la memoria principal han sido generadas correctamente, realizándose la simulación de forma satisfactoria.
Evaluación	OK

Tabla 6.9: *Test de verificación VET-08.*

ID	VET-08.
Nombre	Importación y exportación.
Requisitos	SR-F-F07, SR-F-F08, SR-F-F13, SR-F-F14
Descripción	Verificar que el software permite tanto la importación como la exportación de/a fichero del juego de instrucciones y del código ensamblador.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente. 2. Disponer de un juego de instrucciones base en el formato utilizado en la asignatura Estructura de Computadores en un fichero. 3. Disponer de un código ensamblador que utiliza el juego de instrucciones utilizado en la asignatura Estructura de Computadores en un fichero.
Procedimiento	<ol style="list-style-type: none"> 1. Importar juego de instrucciones mediante el fichero proporcionado. 2. Generar microcódigo. 3. Exportar juego de instrucciones. 4. Importar código ensamblador mediante el fichero proporcionado. 5. Generar memoria principal. 6. Exportar código ensamblador. 7. Verificar que la memoria de control y la memoria principal han sido generadas correctamente. 8. Verificar que los ficheros exportados han sido generados correctamente.
Postcondiciones	La memoria de control, la memoria principal y los dos ficheros exportados han sido generados correctamente.
Evaluación	OK

Tabla 6.10: *Test de verificación VET-09.*

ID	VET-09.
Nombre	Edición de texto.
Requisitos	SR-F-F06, SR-F-F12
Descripción	Verificar que el software permite la edición del juego de instrucciones y del código ensamblador.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente.
Procedimiento	<ol style="list-style-type: none"> 1. Definir un juego de instrucciones mediante la herramienta. 2. Generar microcódigo. 3. Definir código ensamblador mediante la herramienta. 4. Generar memoria principal.
Postcondiciones	La memoria de control, la memoria principal han sido generadas correctamente, pudiéndose editar desde la herramienta.
Evaluación	OK

Tabla 6.11: *Test de verificación VET-10.*

ID	VET-10.
Nombre	Tipos de simulaciones.
Requisitos	SR-F-F16, SR-F-F17, SR-F-F18, SR-F-F22
Descripción	Verificar que el software permite los tres tipos de simulación: ciclo a ciclo de reloj, instrucción a instrucción y simulación completa.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente. 2. Tener la memoria de control y la memoria principal de la herramienta generadas.
Procedimiento	<ol style="list-style-type: none"> 1. Realizar la simulación del código cargado microinstrucción a microinstrucción. 2. Reiniciar la simulación. 3. Realizar la simulación del código cargado instrucción a instrucción. 4. Reiniciar la simulación. 5. Realizar la simulación completa del código cargado.
Postcondiciones	Las simulaciones han sido realizadas correctamente, obteniéndose los resultados teóricos esperados en ellas.
Evaluación	OK

Tabla 6.12: *Test de verificación VET-11.*

ID	VET-11.
Nombre	Velocidad de simulación.
Requisitos	SR-F-F19
Descripción	Verificar que el software permite la configuración de la velocidad de simulación.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente. 2. Tener la memoria de control y la memoria principal de la herramienta generadas.
Procedimiento	<ol style="list-style-type: none"> 1. Realizar la simulación completa del código cargado. 2. Reiniciar la simulación. 3. Modificar velocidad de simulación. 4. Realizar la simulación completa del código cargado.
Postcondiciones	Las simulaciones han sido realizadas correctamente, obteniéndose diferentes tiempos en función de la velocidad de simulación establecida.
Evaluación	OK

Tabla 6.13: *Test de verificación VET-12.*

ID	VET-12.
Nombre	Información de las simulaciones.
Requisitos	SR-F-F20, SR-F-21, SR-F-22
Descripción	Verificar que el software muestra los esquemas de la CPU y la Unidad de control, modificando el color de las señales activas y buses actualizados y muestra la información del estado de los registros.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente. 2. Tener la memoria de control y la memoria principal de la herramienta generadas.
Procedimiento	<ol style="list-style-type: none"> 1. Realizar la simulación del código cargado ciclo a ciclo de reloj.
Postcondiciones	Las simulaciones han sido realizadas correctamente, cambiándose el color de las señales activas y buses de datos actualizados en cada ciclo de reloj y mostrándose la información de los registros.
Evaluación	OK

La matriz de trazabilidad de las pruebas de verificación (Tabla 6.14) determina que todos los requisitos de software se han verificado durante la fase de desarrollo del proyecto.

Tabla 6.14: Matriz de trazabilidad de pruebas de verificación.

Requisitos	VET-01	VET-02	VET-03	VET-04	VET-05	VET-06	VET-07	VET-08	VET-09	VET-10	VET-11	VET-12
SR-F-F01	-	-	-	-	-	✓	-	-	-	-	-	-
SR-F-F02	-	-	-	-	-	✓	-	-	-	-	-	-
SR-F-F03	-	-	-	-	-	✓	-	-	-	-	-	-
SR-F-F04	-	-	-	-	-	-	✓	-	-	-	-	-
SR-F-F05	-	-	-	-	-	-	✓	-	-	-	-	-
SR-F-F06	-	-	-	-	-	-	-	-	✓	-	-	-
SR-F-F07	-	-	-	-	-	-	-	-	✓	-	-	-
SR-F-F08	-	-	-	-	-	-	-	-	✓	-	-	-
SR-F-F09	-	-	-	-	-	-	✓	-	-	-	-	-
SR-F-F10	-	-	-	-	-	-	✓	-	-	-	-	-
SR-F-F11	-	-	-	-	-	-	✓	-	-	-	-	-
SR-F-F12	-	-	-	-	-	-	-	-	✓	-	-	-
SR-F-F13	-	-	-	-	-	-	-	✓	-	-	-	-
SR-F-F14	-	-	-	-	-	-	-	✓	-	-	-	-
SR-F-F15	-	-	-	-	-	-	✓	-	-	-	-	-
SR-F-F16	-	-	-	-	-	-	-	-	-	✓	-	-
SR-F-F17	-	-	-	-	-	-	-	-	-	✓	-	-
SR-F-F18	-	-	-	-	-	-	-	-	-	✓	-	-
SR-F-F19	-	-	-	-	-	-	-	-	-	-	✓	-
SR-F-F20	-	-	-	-	-	-	-	-	-	-	-	✓
SR-F-F21	-	-	-	-	-	-	-	-	-	-	-	✓
SR-F-F22	-	-	-	-	-	-	-	-	-	✓	-	✓
SR-NF-PL01	✓	-	-	-	-	-	-	-	-	-	-	-
SR-NF-PL02	✓	-	-	-	-	-	-	-	-	-	-	-
SR-NF-PL03	-	✓	-	-	-	-	-	-	-	-	-	-
SR-NF-PL04	-	✓	-	-	-	-	-	-	-	-	-	-
SR-NF-PL05	-	-	✓	-	-	-	-	-	-	-	-	-
SR-NF-UI01	-	-	-	✓	-	-	-	-	-	-	-	-
SR-NF-P01	-	-	-	-	✓	-	-	-	-	-	-	-

6.1.2 Pruebas de validación

Para realizar las pruebas de validación, hemos comprobado el software final, comparándolo con las necesidades del usuario especificadas en el Capítulo 3 *Análisis*). Con estas pruebas queremos responder a la pregunta: “*¿Hemos construido el producto adecuado?*”. La Tabla 6.15 proporciona la plantilla utilizada para los test de validación. Es necesario tener en cuenta que el formato del atributo ID es VAT-XX, donde XX indica el número de test de validación.

Tabla 6.15: *Plantilla para test de validación.*

ID	Test ID.
Nombre	Nombre del test.
Requisitos	Requisitos del usuario cumplidos con esta prueba.
Test de verificación	Pruebas de verificación que nos ayudan a validar esta prueba.
Descripción	Descripción de la prueba.
Precondiciones	Las condiciones que siempre deben ser verdad antes de realizar la prueba.
Procedimiento	Una secuencia fija, paso a paso, de las actividades realizadas por la prueba.
Postcondiciones	Las condiciones que siempre deben ser verdaderas justo después de realizar la prueba.
Evaluación	<i>OK o Error.</i>

A continuación, especificamos las pruebas de validación.

Tabla 6.16: *Test de validación VAT-01.*

ID	VAT-01.
Nombre	Juego de instrucciones.
Requisitos	UR-C02, UR-C03
Descripción	Validar que el software permite la definición del juego de instrucciones en el formato utilizado en la asignatura Estructura de Computadores mediante la carga de un fichero y la edición desde la herramienta, generando la memoria de control asociada y permitiendo la exportación del juego de instrucciones a un fichero.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente. 2. Disponer de un juego de instrucciones base en el formato utilizado en la asignatura Estructura de Computadores.
Procedimiento	<ol style="list-style-type: none"> 1. Importar juego de instrucciones desde fichero. 2. Editar el juego de instrucciones desde la herramienta. 3. Generar memoria de control utilizando el juego de instrucciones cargado. 4. Exportar el juego de instrucciones cargado a fichero.
Postcondiciones	La memoria de control ha sido generada correctamente, permitiéndose la carga desde fichero, la edición desde la herramienta y la exportación del juego de instrucciones a fichero.
Evaluación	OK

Tabla 6.17: *Test de validación VAT-02.*

ID	VAT-02.
Nombre	Código ensamblador.
Requisitos	UR-C04, UR-C05
Descripción	Validar que el software permite la definición del código ensamblador en el formato utilizado en la asignatura Estructura de Computadores mediante la carga de un fichero y la edición desde la herramienta, generando la memoria principal asociada y permitiendo la exportación del código ensamblador a un fichero.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente. 2. Disponer de un juego de instrucciones base en el formato utilizado en la asignatura Estructura de Computadores cargado en el simulador. 3. Disponer de un código ensamblador base en el formato utilizado en la asignatura Estructura de Computadores.
Procedimiento	<ol style="list-style-type: none"> 1. Importar código ensamblador desde fichero. 2. Editar el código ensamblador desde la herramienta. 3. Generar memoria principal utilizando el código ensamblador cargado. 4. Exportar el código ensamblador cargado a fichero.
Postcondiciones	La memoria principal ha sido generada correctamente, permitiéndose la carga desde fichero, la edición desde la herramienta y la exportación del código ensamblador a fichero.
Evaluación	OK

Tabla 6.18: *Test de validación VAT-03.*

ID	VAT-03.
Nombre	Simulación del modelo hardware propuesto.
Requisitos	UR-C01
Descripción	Validar que el software permite la simulación del modelo hardware propuesto.
Precondiciones	<ul style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente. 2. Disponer de un juego de instrucciones base en el formato utilizado en la asignatura Estructura de Computadores cargado en el simulador. 3. Disponer de un código ensamblador base en el formato utilizado en la asignatura Estructura de Computadores cargado en el simulador.
Procedimiento	<ul style="list-style-type: none"> 1. Comprobar el código de los archivos fuente del simulador que contienen el modelo hardware a simular (sim_hw_*.js). 2. Ejecutar la simulación del código ensamblador.
Postcondiciones	El modelo hardware definido se corresponde con el modelo propuesto, siendo el resultado de la simulación el resultado teórico esperado.
Evaluación	OK

Tabla 6.19: *Test de validación VAT-04.*

ID	VAT-04.
Nombre	Tipos de simulaciones.
Requisitos	UR-C06
Descripción	Verificar que el software permite los tres tipos de simulación: ciclo a ciclo de reloj, instrucción a instrucción y simulación completa.
Precondiciones	<ol style="list-style-type: none">1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente.2. Tener la memoria de control y la memoria principal de la herramienta generadas.
Procedimiento	<ol style="list-style-type: none">1. Realizar la simulación del código cargado microinstrucción a microinstrucción.2. Reiniciar la simulación.3. Realizar la simulación del código cargado instrucción a instrucción.4. Reiniciar la simulación.5. Realizar la simulación completa del código cargado.
Postcondiciones	Las simulaciones han sido realizadas correctamente, obteniéndose los resultados teóricos esperados en ellas.
Evaluación	OK

Tabla 6.20: *Test de validación VAT-05.*

ID	VAT-05.
Nombre	Velocidad de simulación.
Requisitos	UR-C07
Descripción	Verificar que el software permite la configuración de la velocidad de simulación.
Precondiciones	<ul style="list-style-type: none"> 1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente. 2. Tener la memoria de control y la memoria principal de la herramienta generadas.
Procedimiento	<ul style="list-style-type: none"> 1. Realizar la simulación completa del código cargado. 2. Reiniciar la simulación. 3. Modificar velocidad de simulación. 4. Realizar la simulación completa del código cargado.
Postcondiciones	Las simulaciones han sido realizadas correctamente, obteniéndose diferentes tiempos en función de la velocidad de simulación establecida.
Evaluación	OK

Tabla 6.21: *Test de validación VAT-06.*

ID	VAT-06.
Nombre	Información de las simulaciones.
Requisitos	UR-C08, UR-C09
Descripción	Verificar que el software muestra los esquemas de la CPU y la Unidad de control, modificando el color de las señales activas y buses actualizados y muestra la información del estado de los registros.
Precondiciones	<ol style="list-style-type: none">1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente.2. Tener la memoria de control y la memoria principal de la herramienta generadas.
Procedimiento	<ol style="list-style-type: none">1. Realizar la simulación del código cargado ciclo a ciclo de reloj.
Postcondiciones	Las simulaciones han sido realizadas correctamente, cambiándose el color de las señales activas y buses de datos actualizados en cada ciclo de reloj y mostrándose la información de los registros.
Evaluación	OK

Tabla 6.22: *Test de validación VAT-07.*

ID	VAT-07.
Nombre	Plataforma.
Requisitos	UR-R01, UR-R02
Descripción	Verificar que el software puede ser utilizado en los navegadores especificados en los requisitos.
Precondiciones	<ol style="list-style-type: none"> 1. Utilizar un computador con sistema operativo Ubuntu 16.04 / Windows 10 / MacOS 10.2.5 . 2. Tener instalados los navegadores Microsoft Edge 30+, Mozilla Firefox 45+, Google Chrome 50+ o Safari 10+.
Procedimiento	<ol style="list-style-type: none"> 1. Abrir el software en cualquiera de los navegadores web especificados. 2. Generar microcódigo. 3. Generar memoria principal. 4. Ejecutar simulación.
Postcondiciones	La simulación ha sido completada satisfactoriamente sin ningún error de ejecución debido a la plataforma
Evaluación	OK

Tabla 6.23: *Test de validación VAT-08.*

ID	VAT-08.
Nombre	Interfaz de usuario.
Requisitos	UR-R03
Descripción	Verificar que la interfaz del simulador es compatible tanto con pc como con plataformas móviles.
Precondiciones	<ol style="list-style-type: none"> 1. Disponer de un pc y de un dispositivo móvil. 2. Los dispositivos deben tener instalados los navegadores web especificados previamente.
Procedimiento	<ol style="list-style-type: none"> 1. Abrir el software en cualquiera de los navegadores web especificados. 2. Generar microcódigo. 3. Generar memoria principal. 4. Ejecutar simulación.
Postcondiciones	La interfaz del simulador es compatible tanto como pc como con dispositivos móviles, visualizándose correctamente en cualquier pantalla de la herramienta.
Evaluación	OK

Tabla 6.24: *Test de validación VAT-09.*

ID	VAT-09.
Nombre	Tiempo medio por ciclo de reloj.
Requisitos	UR-R04
Descripción	Verificar que el tiempo medio por ciclo de reloj del simulador no supera 0,1 segundos.
Precondiciones	1. Disponer de un dispositivo con una versión de navegador web igual o superior a los indicados anteriormente.
Procedimiento	1. Abrir el software en cualquiera de los navegadores web especificados. 2. Generar microcódigo. 3. Generar memoria principal utilizando un código ensamblador que utiliza cada una de las partes de la arquitectura del simulador. 4. Ejecutar simulación midiendo el tiempo de ejecución.
Postcondiciones	El tiempo de ejecución medio por ciclo de reloj excede de 0,1 segundos.
Evaluación	OK

Tabla 6.25: *Test de validación VAT-10.*

ID	VAT-10.
Nombre	Ejecución local.
Requisitos	UR-R05, UR-R06
Descripción	Verificar que el software es ejecutado en local sin la necesidad de conexión a internet.
Precondiciones	1. No tener acceso a Internet. 2. Tener el código fuente de la herramienta descargado en local.
Procedimiento	1. Abrir el software en cualquiera de los navegadores web especificados. 2. Generar microcódigo. 3. Generar memoria principal. 4. Ejecutar simulación.
Postcondiciones	La simulación ha sido completada satisfactoriamente sin ningún error de ejecución debido a la conexión de red.
Evaluación	OK

Tabla 6.26: *Test de validación VAT-11.*

ID	VAT-11.
Nombre	Tecnologías de desarrollo.
Requisitos	UR-R07
Descripción	Verificar que el software ha sido desarrollando utilizando HTML5.
Precondiciones	1. Tener el código fuente de la herramienta descargado en local.
Procedimiento	1. Comprobar el código de los archivos fuente del simulador.
Postcondiciones	El simulador ha sido desarrollado mediante el lenguaje de programación HTML5.
Evaluación	OK

La matriz de trazabilidad de las pruebas de validación (Tabla 6.27) determina que todos los requisitos de usuario han sido validados con el producto final.

Tabla 6.27: Matriz de trazabilidad de pruebas de validación.

Capítulo 7

Planificación y presupuesto

Este capítulo presenta una planificación detallada del proyecto (Sección 7.1, *Planificación*). Luego, se explican los costes del proyecto (Sección 7.2, *Presupuesto*). Al final del capítulo, se explica el entorno socio-económico del proyecto (Sección 7.3, *Entorno socio-económico*).

7.1 Planificación

Esta sección incluye la planificación completa del proyecto. En primer lugar, se describe la metodología de desarrollo de software utilizada. Después, se detalla la duración de cada fase del proyecto, indicando todos los tiempos en un diagrama Gantt.

7.1.1 Justificación de la metodología

Debido a sus características, hemos dividido nuestro proyecto en tres iteraciones:

- **Modelo hardware:** la primera iteración ha consistido en lograr el modelado del hardware que compone el procesador WepSIM. El objetivo de esta fase, ha sido lograr el modelado y funcionamiento básico de cada componente hardware por separado.
- **Modelo software:** esta fase ha consistido en lograr el modelo software utilizado en la asignatura Estructura de computadores para la definición del juego de instrucciones y el lenguaje ensamblador. El objetivo de esta fase, ha sido lograr interpretar el juego de instrucciones definido y código ensamblador definidos por el usuario, generando la memoria de control y memoria principal en el orden correspondiente.

- **Kernel del simulador:** esta fase ha consistido en lograr unir el modelo hardware y modelo software, realizando el kernel del simulador. El objetivo de esta fase, ha sido diseñar e implementar el kernel de simulador encargado de unir el modelo hardware y el modelo software, generando la simulación y la vista del simulador.

Era necesario contar con una metodología iterativa para desarrollar cada una de las fases de manera independiente para unirlas todas en la última etapa y obtener el producto final. Para ello, hemos analizado tres metodologías de desarrollo de software diferentes: Prototipado de software [19], el Modelo en Cascada [20] y el Modelo en Espiral [21]. El Prototipado de Software no encaja muy bien en nuestro proyecto, puesto que requiere la construcción de un prototipo de software en poco tiempo. El Modelo en Cascada es un proceso de diseño secuencial, utilizado en procesos de desarrollo de software en el cual el progreso es visto como un flujo constante hacia abajo (como una cascada) a través de diferentes fases. El problema con esta metodología es que no permite iteraciones dentro del desarrollo del software. Finalmente, el Modelo en Espiral permitió dividir el proyecto en diferentes iteraciones. Este modelo, combina las fortalezas de los otros dos modelos (simplicidad y flexibilidad), utilizando además un proceso iterativo. Aunque este modelo es más lento que los dos anteriores, nos permitió aplicar diferentes iteraciones por lo que decidimos aplicarlo a todo el proceso.

7.1.2 Ciclo de vida

El proceso de desarrollo del ciclo de vida del proyecto ha seguido el Modelo de ciclo de vida en Espiral [21]. La Figura 7-1 muestra el Modelo en Espiral usando un esquema.

El modelo en Espiral tiene cuatro fases, que se repiten durante las diferentes iteraciones del modelo. Estas fases son:

- **Planificación** (*"Determine objectives"* en Figura 7-1): Se reúnen los requisitos de los usuarios, se realiza un estudio de factibilidad del sistema y se determinan los objetivos de iteración.
- **Análisis** (*"Identify and resolve risks"* en Figura 7-1): Se realiza un análisis completo de los requisitos y se identifican los posibles riesgos. Esta fase termina con un diseño básico.
- **Desarrollo y Pruebas** (*"Development and Test"* en Figura 7-1): Se lleva a cabo la implementación del código. Se realizan los casos de prueba y los resultados de la prueba.

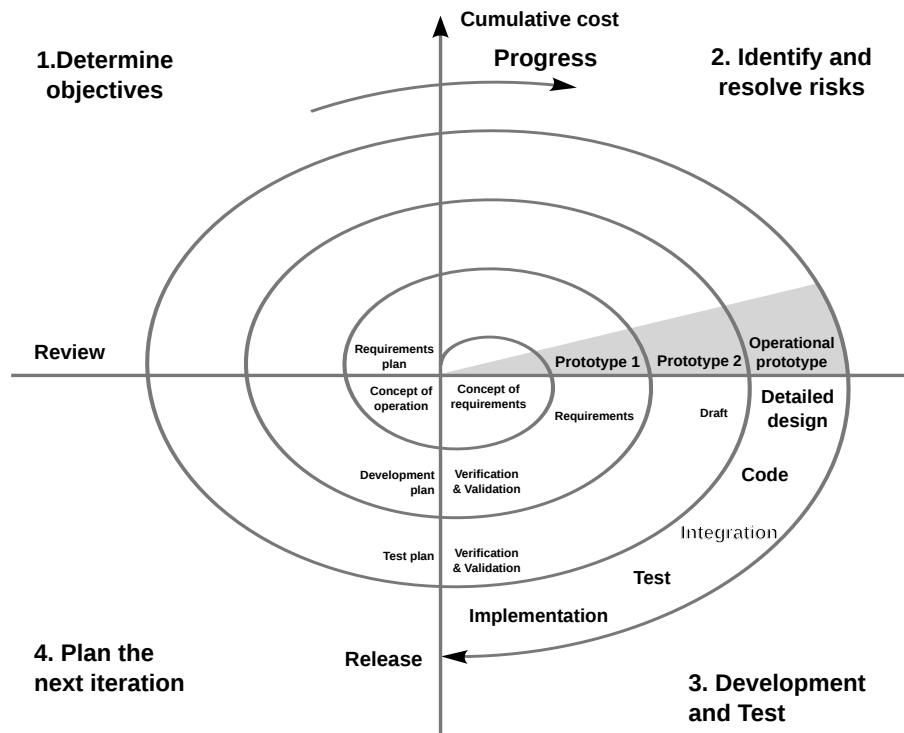


Figura 7-1: *Modelo en Espiral* (Boehm, 2000).

- **Evaluación** (“*Plan the next iteration*” en Figura 7-1): Los clientes evalúan el software y proporcionan sus comentarios. En este caso, el estudiante intenta obtener la aprobación del supervisor. Esta es la *tarea crítica* del ciclo de vida, ya que solo podemos pasar a la siguiente iteración del Modelo de ciclo de vida Espiral si esta tarea es aprobada.

Cada fase comienza con un objetivo de diseño y termina con el cliente (el supervisor) revisando el progreso hasta ahora. Como se explicó anteriormente, hemos dividido el desarrollo de software en tres iteraciones: modelo hardware, modelo software, y kernel del simulador. En la última iteración, el software completo debe someterse a pruebas exhaustivas para validar el simulador.

7.1.3 Tiempo estimado

El diagrama de Gantt (Figura 7-2) muestra todas las tareas realizadas durante el desarrollo del proyecto. El proyecto comenzó el 2 de noviembre de 2015 y finalizó el 30 de diciembre de 2016, lo que supone un total de casi 13 meses de trabajo (305 días laborables). Durante este tiempo, he trabajado de lunes a viernes, durante cuatro horas al día, haciendo un total de 1.220

horas trabajadas.

El diagrama de Gantt muestra todas las tareas realizadas en cada iteración del modelo de ciclo de vida espiral. Recuerde que las tres iteraciones fueron: Modelo hardware, Modelo software y kernel del simulador. Además de las tareas (fases) mencionadas anteriormente (Planificación, Análisis, Desarrollo y Pruebas, y Evaluación), hemos incluido la tarea de Documentación al final de cada iteración. La tarea de documentación ha consistido principalmente en la redacción de este trabajo fin de grado.

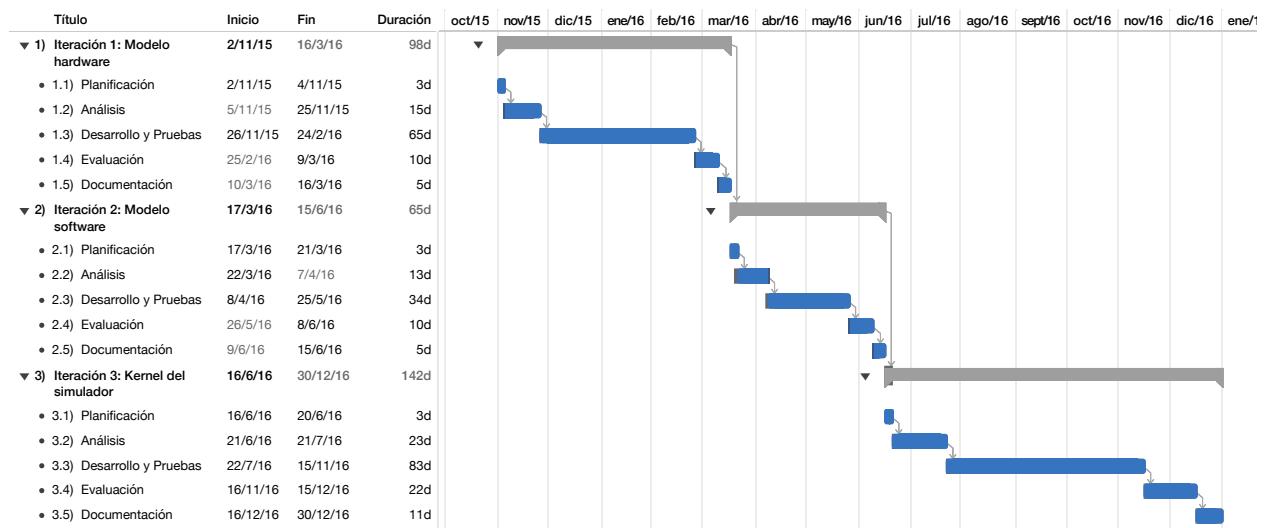


Figura 7-2: Diagrama Gantt.

7.2 Presupuesto

Esta sección detalla el presupuesto general del proyecto. Por un lado, presentamos los costes del proyecto y, por otro lado, damos a conocer la oferta presentada al cliente.

7.2.1 Coste del proyecto

La Tabla 7.1 resume las principales características del proyecto, incluido el presupuesto total.

Tabla 7.1: *Información del Proyecto.*

<i>Información del Proyecto</i>	
Título	WepSIM: Simulador de procesador elemental con unidad de control microprogramada
Autor	Javier Prieto Cepeda
Departamento	Departamento de Informática
Fecha de inicio	2 de noviembre de 2015
Fecha de finalización	30 de diciembre de 2016
Duración	13 meses
Ratio de costes indirectos	20 %
Presupuesto total	51.860,96

A continuación se desglosa el presupuesto total del proyecto.

Costes Directos

En esta parte se presentan los costes directos del proyecto. La Tabla 7.2 muestra los costes directos causados por los costes de personal, sobre la base de la planificación presentada en la sección anterior. El tutor y el estudiante han desempeñado los siguientes roles:

- **Tutor:** Jefe de Proyecto.
- **Estudiante:** Analista, Desarrollador, Tester.

Tabla 7.2: *Costes de recursos humanos.*

Categoría	Coste por hora (€)	Horas	Total (€)
Jefe de Proyecto	60	65	3.900
Analista	35	269	9.415
Desarrollador	35	523	18.305
Tester	25	428	10.700
Total			42.320,00

La Tabla 7.3 muestra los costes directos causados por la adquisición y uso del equipo. El coste amortizado, C, es calculado utilizando la siguiente fórmula:

$$C = \frac{d \cdot c \cdot u}{D} \quad (7.1)$$

Donde:

- **C:** Coste amortizado. Es equivalente al valor amortizado.
- **d:** Tiempo que el equipamiento ha sido utilizado.
- **c:** Coste del equipamiento.
- **u:** Dedicación al proyecto. Porcentaje del tiempo que el equipamiento ha sido utilizado.
- **D:** Periodo de amortización del equipamiento.

Tabla 7.3: *Costes de equipamiento.*

Concepto	Coste, c (€)	Dedicación, u (%)	Dedicación, d (meses)	Amortización, D (meses)	Coste amortizado, C (€)
PC sobremesa	799,99	100	13	36	288,89
Portátil	529,99	25	13	36	47,85
ARCOS Mirlo	1.979,99	50	7	60	115,50
Impresora	79,99	5	4	60	0,27
Total					452,51

Además, el equipo presentado en la Tabla 7.3 es detallado a continuación:

- **PC sobremesa:** All in One - Asus Z220ICUK, 21.5", i5-6400T, 8GB, 1TB)
- **Portátil:** Toshiba L50D-C-19D, A10-8700P, 8GB RAM and 1TB.
- **ARCOS Mirlo:** Servidor utilizado por el grupo de investigación ARCOS. 32GB RAM y 8 procesadores i7 a 2.67GHz cada uno.
- **Impresora:** Brother DCP-1510.

Otros costes directos se muestran en la Tabla 7.4. Estos costes consisten en material de oficina, un tóner para la impresora y el abono transporte mensual. El material de oficina incluye: lápices, bolígrafos, cuadernos, papel y marcadores.

Tabla 7.4: *Otros costes directos.*

Concepto	Coste (€)
Material de oficina	112,98
Toner (x1)	71,99
Abono transporte (x13)	260
Total	444,97

Resumen de costes

La Tabla 7.5 muestra el resumen completo de los costes del proyecto. Los costes indirectos (20 % de los costes directos) consisten en las facturas de electricidad y agua, teléfono, acceso a internet, etc.

Tabla 7.5: *Resumen de costes.*

<i>Resumen de costes</i>	
Recursos humanos	42.320,00
Equipamiento	452,51
Otros costes directos	444,97
Costes indirectos	8.643,50
Presupuesto total	51.860,98

El presupuesto total de este proyecto asciende a **51.860,98 € (cincuenta y uno mil ochocientos sesenta euros y noventa y ocho céntimos).**

7.2.2 Oferta de Proyecto Propuesta

La Tabla 7.6 muestra la propuesta de oferta detallada. Esta oferta incluye los riesgos estimados (20 %), los beneficios esperados (15 %), y el Impuesto de Valor Agregado (Impuesto Sobre el

Valor Añadido (IVA)), que corresponde al 21 % [22]. Después de aplicar todos estos conceptos, la cantidad final para este proyecto en caso de venta a un cliente de terceros es **86.597,46 €** (**Ochenta y seis mil quinientos noventa y siete euros y cuarenta y seis céntimos**).

Tabla 7.6: *Oferta propuesta.*

<i>Oferta propuesta</i>				
Concepto	Incremento (%)	Valor Parcial (€)	Coste agregado (€)	
Costes del proyecto	-	51.860,98	51.860,98	
Riesgos	20	10.372,20	62.233,17	
Beneficios	15	9.334,98	71.568,15	
IVA	21	15.029,31	86.597,46	
Total				86.597,46

7.3 Entorno socio-económico

Como se ha comentado en capítulos anteriores, WepSIM puede apoyar la enseñanza en Estructura y Arquitectura de Computadores. Esto significa que los profesores de estas asignaturas pueden utilizar esta herramienta como refuerzo en las explicaciones del temario para mostrar de forma visual y práctica a los estudiantes el comportamiento de un procesador elemental. Además, sirve para que los estudiantes puedan desarrollar las prácticas de estas asignaturas sin la necesidad de utilizar diferentes herramientas, evitando la pérdida de tiempo que esto supone y permitiendo a los estudiantes dedicar el mayor tiempo posible a la realización de los ejercicios. Este ahorro de tiempo, tiene un impacto en los estudiantes a la hora de minimizar el tiempo invertido en la realización de prácticas, de forma que la mayor parte del tiempo dedicado a la asignatura sea útil y sirva para comprender mejor el funcionamiento de un computador, sirviendo de base en el resto de la trayectoria académica de los estudiantes y mejorando su rendimiento.

Debido al diseño modular realizado de este simulador, WepSIM permite la especificación tanto de diferentes juegos de instrucciones como de diferentes modelos hardware. Esto supone un gran avance en el área docente, puesto que permite el uso de una misma herramienta para

la enseñanza y simulación de diferentes dispositivos. Anteriormente se invertía mucho dinero para que los estudiantes pudieran realizar prácticas con diferentes tipos de procesadores, ya que se debían de adquirir los diferentes componentes hardware necesarios para la realización de las prácticas. Con WepSIM este concepto cambia, puesto que modelando el comportamiento del computador deseado, puede ser utilizado para diferentes tipos de prácticas, evitando costes de adquisición y mantenimiento y permitiendo una mayor variedad de opciones a la hora de diseñar las prácticas.

Gracias al uso de este simulador en el ámbito docente, se puede mostrar el impacto energético que supone una programación eficiente. Esto es debido a que WepSIM es un simulador de microprogramación y programación en ensamblador, pudiendo hacer visible el coste a bajo nivel de unas malas prácticas de programación. El objetivo, es concienciar a los estudiantes de la importancia que tiene la optimización y eficiencia en el uso de los recursos de un computador, puesto que uno de los retos de la comunidad científica es diseñar sistemas cada vez más eficientes energéticamente, de forma que en un futuro estas técnicas sean llevadas a cabo realizando sistemas más eficientes.

Por último, es necesario destacar el impacto que supone que WepSIM tenga licencia de código abierto. Esto supone que las universidades y centros docentes que quieran utilizar la herramienta, puedan hacerlo sin ningún tipo de coste, con el ahorro económico que supone la utilización de una herramienta de estas características. Además, gracias a este tipo de licencia, se favorece un entorno de colaboración entre diferentes universidades, de forma que se puedan añadir nuevos módulos a la herramienta sin coste económico para la comunidad docente.

Capítulo 8

Conclusiones y trabajos futuros

En este capítulo se presentan las conclusiones del trabajo, se revisan los objetivos establecidos al principio de este documento, y se incluyen algunas conclusiones personales. Además, se discuten las principales contribuciones de nuestro trabajo, indicando también las publicaciones resultantes de este trabajo. Finalmente, se discute el trabajo futuro.

8.1 Conclusiones

En este trabajo se ha descrito el diseño de WepSIM, un simulador de un procesador elemental con unidad de control microprogramada. Este trabajo presenta un nuevo simulador que resulta intuitivo, portable y extensible, sirviendo como complemento docente para la docencia en Estructura de Computadores. Este simulador, permite definir diferentes juegos de instrucciones y ejecutar y depurar código fuente que use el conjunto de instrucciones definido. También permite definir el comportamiento del procesador mediante microprogramación.

WepSIM, permite a los estudiantes entender el funcionamiento de un procesador elemental de una forma sencilla, pudiendo ser usado desde un dispositivo móvil o un ordenador con un navegador Web moderno, sin la necesidad de ser instalado. De esta forma, los estudiantes pueden interactuar con el simulador aprendiendo y comprendiendo el funcionamiento del procesador elemental WepSIM, incluyendo los mecanismos de interacción con el software de sistema e integrando en una misma herramienta tanto la microprogramación como la programación en ensamblador.

El objetivo principal de este proyecto era desarrollar un simulador, que a diferencia de los existentes, pudiera simular de forma completa el comportamiento de un procesador elemental

permitiendo comprobar el estado de los componentes en cada ciclo de reloj, de manera que ayudáse a los estudiantes a comprender y asimilar de forma sencilla y visual el funcionamiento de un procesador. También hemos cumplido con todos los demás objetivos presentados en la introducción del documento:

- **O1**, Se ha diseñado una herramienta que simula la ejecución del juego de instrucciones especificado en un computador llamado WepSIM, desde el punto de vista de la microprogramación y la programación en ensamblador.
- **O2**, La herramienta permite la especificación de diferentes juegos de instrucciones.
- **O3**, La herramienta unifica la microprogramación de un computador y la programación en lenguaje ensamblador.
- **O4**, La herramienta permite al usuario visualizar en cada ciclo de reloj el estado y el comportamiento del computador simulado.

A nivel personal, este trabajo me ha ayudado a adentrarme en el mundo de la investigación científica. He logrado aplicar una gran cantidad de los conocimientos adquiridos a lo largo del grado. Además, he aprendido importantes técnicas de modelado de hardware, compilación y simulación, las cuales tienen una gran utilidad y complejidad y me han servido parar profundizar aún más en los conocimientos adquiridos en el grado. Por todo ello, es muy satisfactorio ver el resultado final obtenido, puesto que he logrado superar todos los problemas que han surgido a lo largo del proyecto.

8.1.1 Contribuciones

El proyecto llevado a cabo durante este Trabajo Fin de Grado encaja con muchas de las asignaturas estudiadas en el Grado en Ingeniería Informática de la Universidad Carlos III de Madrid, destacando los siguientes temas en particular:

- **Tecnología de Computadores** (asignatura obligatoria, Primer curso) en donde se introducen los componentes hardware y la lógica binaria.
- **Estructura de Computadores** (asignatura obligatoria, Segundo curso) en donde se introducen las bases de la estructura y funcionamiento de un computador.

- **Teoría de Autómatas y Lenguajes Formales** (asignatura obligatoria, Segundo curso) en donde se introducen las bases acerca de los lenguajes y gramáticas formales.
- **Sistemas Operativos** (asignatura obligatoria, Segundo curso) en donde se introducen las bases del funcionamiento del sistema operativo.
- **Arquitectura de Computadores** (asignatura obligatoria, Tercer curso) en donde se introducen las bases de la arquitectura de un computador.
- **Diseño de Sistemas Operativos** (asignatura obligatoria, Tercer curso) en donde se introducen las bases del diseño de los distintos módulos de un sistema operativo.
- **Dirección de proyectos de desarrollo de software** (asignatura obligatoria, Tercer curso) en donde se introducen las bases para la dirección y gestión de un proyecto de desarrollo de software.

8.1.2 Publicaciones

Este Trabajo Fin de Grado ha permitido realizar una importante contribución al campo de la docencia en Estructura y Arquitectura de Computadores. Además, se han conseguido publicar los siguientes artículos científicos:

- **A. Calderón, F. García-Carballeira, and J. Prieto**, "WepSIM: Simulador modular e interactivo de un procesador elemental para facilitar una visión integrada de la microprogramación y la programación en ensamblador", *Enseñanza y aprendizaje de ingeniería de computadores*, vol. 6, 35-53,2016. [15]
- **J. Prieto, A. Calderón, F. García-Carballeira, and S. Alonso-Monsalve**, "WepSIM: simulador integrado de microprogramación y programación en ensamblador", *Jornadas sarteco 2016*. [23]

Además, en el momento de la entrega de este documento, también otro artículo enviado a la espera de su aceptación.

8.2 Trabajos futuros

Actualmente, hay varias líneas de trabajos futuros en las cuáles estamos trabajando.

- En cuanto a mejoras en el modelo hardware:
 1. Introducir más elementos hardware, como por ejemplo una caché, de forma que se amplíen los contenidos de la asignatura incluídos en la herramienta.
 2. Introducir un modelo hardware basado en *pipeline*, permitiendo el uso de la herramienta en aquellas asignaturas que utilizan este modelo de arquitectura.
- En cuanto a mejoras en el modelo software:
 3. Añadir revisión semántica del código, permitiendo identificar y notificar los errores de programación al usuario.
 4. Añadir nuevos juegos de instrucciones a la herramienta como por ejemplo el ensamblador ARM, permitiendo la utilización de diferentes lenguajes en la herramienta.
 5. Estudiar el ensamblador de MIPS/ARM generado con GCC/Clang de forma que pueda ser usado directamente en WepSIM.
- En cuanto a mejoras en la herramienta:
 6. Añadir un módulo de corrección automática de prácticas a la herramienta, de forma que los estudiantes puedan practicar con ella y comprobar la validez de sus ejercicios.
 7. Migrar la herramienta como aplicación móvil mediante el *plugin* Apache Cordova, de forma que la herramienta no quede ligada al uso mediante navegador web.

Anexos

Apéndice A

Summary

Introduction

Teaching the topics related to the architecture of a computer is a basic and fundamental part of the training of computer science students. In these subjects, students achieve a low-level vision and understanding of the components of a computer. In order to facilitate the correctly understanding of the theoretical foundations, it is necessary to use practical classes, where the student should interact with the architecture of a computer in a similar way to that one explained in theory and also manage to extrapolate the theoretical foundations to the actual behavior.

One of the main problems when preparing these practical classes is to emulate the necessary resources seen in the theoretical classes. Currently, there are different tools that enable the emulation of different components of a computer. However, there is a lack of tools that unify the simulation of these components, being a problem when trying to obtain a global view of the operation of a computer.

These tools can be classified into two main types: emulators and simulators. An emulator is a software that imitates the behavior of a computer, so that programs designed for a particular architecture can be executed in a different one. On the other hand, a simulator is a software that aims to reproduce the behavior of a computer, but with a lower level of realism than the emulators.

There are different simulators that can be used in order to work with the main aspects treated in the subjects of "Computer Architecture and Computer Structure" such as assembler, cache, etc. Most of these simulators are focused on a specific type of simulation, which implies

that, to obtain a global vision on the architecture of the computer several simulators are needed. Therefore, it is much better to count with a tool that offers a complete simulation of the architecture and combines the maximum possible functionalities.

We identify another important problem: most of the simulators are designed for running as PC applications. One of the goals we put forward with WepSIM is that it could be used in smartphones or tablets in order to offer greater flexibility in its usage to the student.

In addition to having a portable simulator, it must also be as self-contained as possible. Therefore, it must integrate support for its use inside the tool(not as a separate document that serves as a user manual to be printed), allowing the end-user to make full use of the application without the need of getting out of it.

In conclusion, we have considered how to offer a simulator that is simple and modular, and that at the same time allows to integrate the teaching of the microprogramming language with programming in assembly. In particular, it can be used to microprogram a customized instruction set and observe the basic operation of a processor, as well as to create assembler programs based on the assembler defined by the above microcode. This is of great help, for example, for system programming, since it is possible to follow how the software interacts in assembly with the hardware in the treatment of interruptions. The idea is to offer a simulator that offers a global vision of a computer, integrating both hardware and software components and also avoiding the extra time involved in learning different tools.

Objectives

The main objective of this project is to design and develop a simulator, which, unlike the existing ones, can fully simulate the behavior of an elementary processor allowing to check the state of the components in each clock cycle, so that it helps the students to understand and assimilate in a simple and visual way the operation of a processor. The secondary objectives, derived from the main objective, are as follow:

- **O1:** Simulate the execution of the set of instructions specified in a computer called Wep-SIM from the point of view of microprogramming and programming in assembly.
- **O2:** Allow the specification of different and customized sets of instructions.

- **O3:** Allow unified microprogramming in a computer and programming in assembly language.
- **O4:** Allow the user to display in each clock cycle the status and behavior of the simulated computer.

WepSIM Elemental Processor

WepSIM is an elementary processor with an integrated microprogrammed control unit designed by the staff of the research group ARCOS of the University Carlos III of Madrid. This processor design is used for teaching in the subject “Computer Structure”.

In Figure A-1, the structure of the WepSIM Elemental Processor is shown. WepSIM consists of a memory module, a keyboard device, a display device, and a generic I/O device that can be used to work with interrupts.

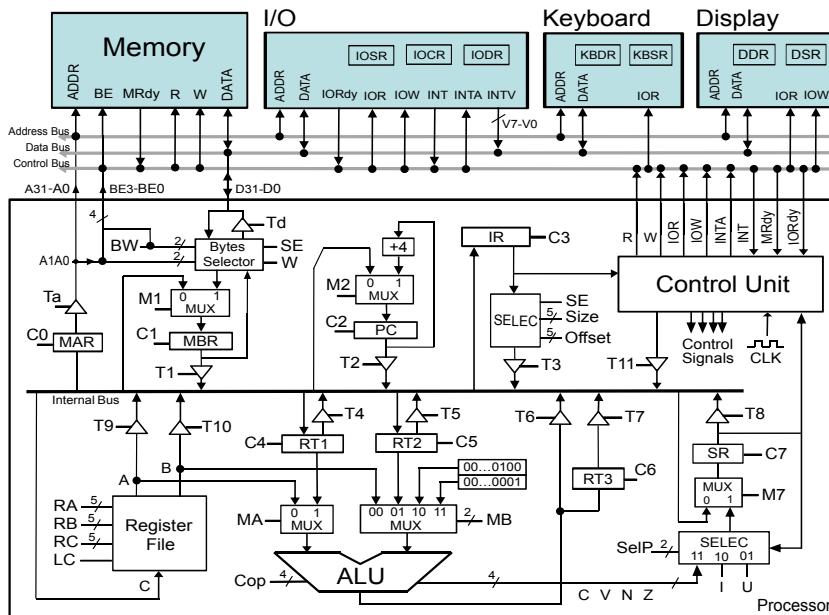


Figura A-1: *CPU Architecture WepSIM.*

WepSIM incorporates a 32bit processor that includes a byte addressable memory, a bank of 32 registers and two additional registers (RT1 and RT2) that are not visible to the assembly programmer but allow temporary storage of data for the carrying out of intermediate operations. From the registers, it is possible to forward the values to operate in an ALU that implements the 15 most common arithmetic-logic operations. The PC registry has its own 4-add operator,

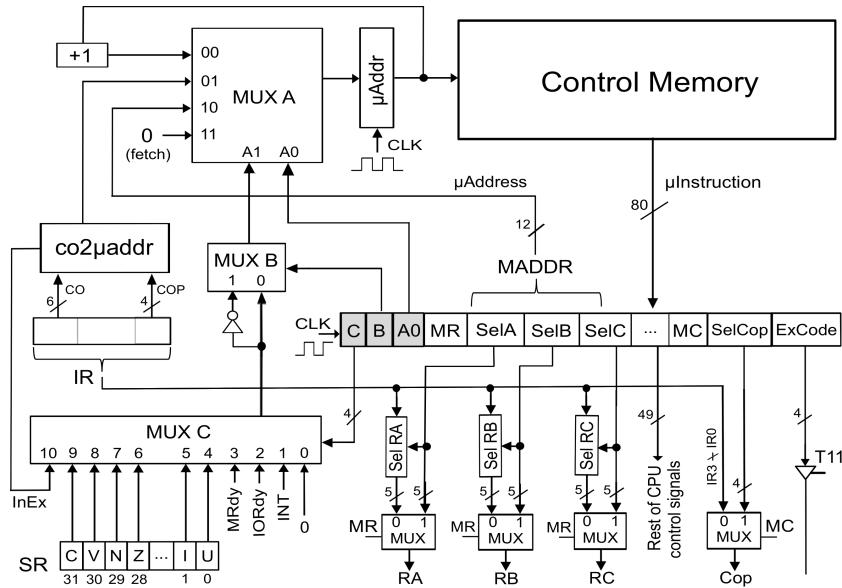


Figura A-2: Control unit architecture in WepSIM.

so it is not necessary to make use of the ALU for this operation. The result of the operations performed in the ALU can be stored in a temporary register (RT3) that is also invisible to the assembly programmer, or sent directly to the internal bus through the corresponding tristate.

The status register (SR) can be updated with the resulting flags of the last operation of the ALU (O, N and Z). To do this, SELEC/SelP represents a circuit block that allows indicating which part of the state register (SR) must be updated. **To the right of SELEC/SelP the bits of the SR status register enter (Input = ONZIU) and SelP allows to select which group of these bits will be updated in the status register: bits O, N and Z with the values coming of the ALU, the bit I with the indicated value or the bit U with the indicated value for the same.**

The instruction register (IR) is associated with a selector module (higher level circuit than a multiplexer, etc.), which allows the selection of a segment of the binary value stored in the instruction register that will pass to T3.

In particular, the position (offset, where 0 represents the least significant bit of the IR register) and the number of bits (Size) to be taken from that initial position is indicated, and whether if sign extension (SE) is desired before passing the value to the T3 input.

The MAR and MBR registers are used to store the address and content associated with this address in read/write operations with memory. The memory is designed for synchronous or asynchronous operation. It currently works synchronously, but has the MRdy signal for working asynchronously in the future. The selection circuit allows the user to indicate which

portion of the memory word is the desired one (one byte, two bytes, or a complete four-byte word).

There are also three I / O devices: a keyboard, a display and a generic device that can be configured to generate various types of interrupts.

Finally, the Control Unit generates the control signals for each clock cycle. The Figure A-2 shows the Control Unit in more detail. It is a microprogrammed control unit with implicit sequencing. The control signals for the current clock cycle are stored in the micro-instruction register (the one with the fields A0, B, C, SelA, etc.). The contents of this record come from the control memory, specifically from the content in the position to which the micro-address register points. The micro-address stored in this register can be modified using the "MUX" multiplexer. There are four options: the current micro-address plus one, a micro-address indicated in the micro-instruction itself (which overlaps with SelA, SelB and partially with SelE), the first micro-address associated with the operation code field of the instruction of the IR register, and finally the value zero, which is the address of the control memory where the microprogram corresponding to the fetch is stored. The micro address can be selected conditionally by using the "MUX C" multiplexer that allows to select the status register bits (SR) or values of the I/O control signals.

The SelRA, SelRB and SelRE selector circuits are used to generate the values corresponding to the selector signals of the register bank RA, RB and RE. These selectors take as input the 32 bits of the instruction register (IR) on one side and the field SelA, SelB and SelE on the other, so that they take SelX as the displacement within the instruction register (from 0 to 32) from where Take the next 5 bits corresponding to the RA, RB or RE signals. They thus allow to select 5 consecutive bits of the 32 bits of the instruction register.

The MR multiplexer is used to indicate whether the RA, RB and RE signals are literally the values stored in SelA, SelB and SelE (MR = 1) or if SelA, SelB and SelE indicate the offset within the instruction where the values to be used for RA, RB and RE. The latter allows the instruction to indicate the registers to be used as operands in the register bank, instead of indicating them from the micro-instruction.

For the Cop signal (operation code in the ALU), the MC signal can be used to take the SelCop value of the micro-instruction (MC = 1) or the 4 least significant bits of the instruction register (MC = 0), ie IR3 -IR 0.

Simulator design

In order for the teachers of the Computer Structure subject to use a tool that helps to explain the theoretical concepts of the subject, and that, at the same time, students can employ to understand the main concepts and to carry out the subject's practices later we propose the design and implementation of a web tool that realistically simulates an elementary processor with a microprogrammable control unit.

This simulator will be developed as a web tool due to the portability it provides, since it can be executed on a large number of different devices regardless of the operating system use, only needing a web browser for its correct operation. In this way, teachers and students will be able to use the tool without depending on their installation in the device to be used, even allowing the students to practice on mobile devices.

To achieve this portability, the simulator has been developed in HTML5 (HTML + JavaScript + CSS) making it possible to run on any platform (smartphones, tablets, PC, etc.) being compatible with Microsoft Edge, Mozilla Firefox, Google Chrome or Safari. In addition, the tool depends on the following frameworks or libraries: JQuery, JQueryUI, JQuery Mobile, Knockout, and BootStrap.

Therefore, the chosen solution is able to unify in a single tool all the features required for the teaching of Computer Structure with a high level of detail, with high availability by facilitating it as a web tool, and being portable, since it can be executed on a large number of different devices, **always looking for a solution without the need for a server**.

Simulator Architecture

The architecture of the solution presented in this work consists of three main elements:

- **Hardware model:** allows the user to define the hardware to use.
- **Software model:** allows the user to define the set of instructions to use.
- **Simulation kernel:** simulates hardware operation by running the microcode / machine language previously defined.

The hardware model allows the user to define the typical elements of a computer (main memory, processor, etc.) in a modular way. The way in which these elements are defined balances

two opposing objectives: it must be complete enough to imitate the main aspects of reality, but also minimal enough to facilitate its use. Above all, it is intended to be a didactic tool.

The software model allows defining the microcode and assembler based on this microcode as intuitively as possible. The assembly used is given by a set of instructions that can be defined by the user. These instructions try to be flexible enough to be able to define different types and sets of instructions, such as MIPS or ARM.

The third element of the proposed architecture is a kernel that takes as input the described hardware model and the working software model, and is responsible for showing the operation of the hardware with the given software.

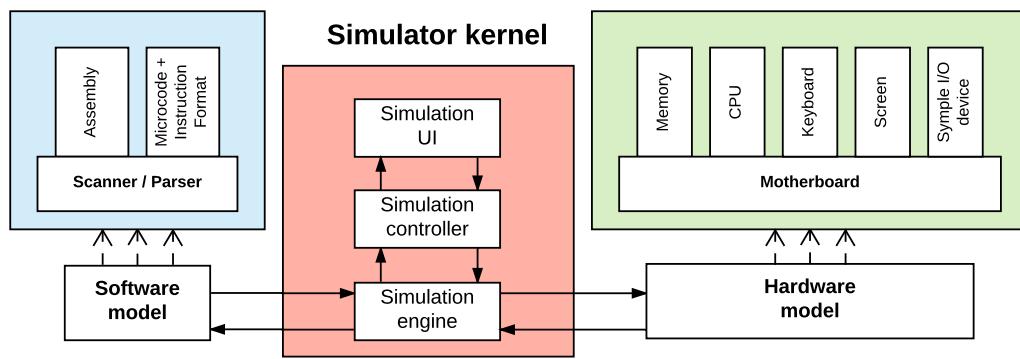


Figura A-3: *WepSIM Architecture*.

Figure A-3 summarizes the architecture of WepSIM. The starting point is the hardware model that describes the processor to be simulated. This includes the processor, memory, and some I/O devices such as keyboard, display, and a single I/O device that generates interrupts. The hardware model describes the overall state of the processor. From the processor's overall state, the simulation kernel updates the state in each clock cycle.

The simulated control unit stores the control signals of each cycle in a control memory. The control memory has all the microprograms for the instructions with which the processor works, and the fetch instruction to be able to read the instructions from the memory and to decode them.

The microcode (the contents of the control memory) together with the format of each instruction (fields of the instruction and its length) is described in a text file. The software model reads this file, translates it to binary and loads it into the processor. The definition of the assembly language to be used is described together with the microcode, and the software model allows to translate programs written in said assembler to binary.

The simulation kernel asks the subsystem of the software model for the defined microcode, the description of the instruction format and the contents of the main memory. The binaries are loaded into the elements of the hardware model, and then the simulation kernel updates the global state in each clock cycle.

WepSIM has a simulation controller that is responsible for updating the clock cycle and displaying the global status. The simulation interface subsystem updates the user interface. When the user uses the user interface to request an operation, the simulation interface subsystem moves the request to the simulation controller. In this way, a basic Model-View-Controller (MVC) design is used for the WepSIM architecture.

Conclusions and future work

This section presents the conclusions obtained from the project, reviews the objectives set out at the beginning of this document, and includes some personal observations. In addition, we discuss the main contributions of our work, also indicating the publications resulting from this project. Finally, future work is discussed.

Conclusions

In this work, we have described the design of WepSIM, an elementary processor simulator with a microprogrammed control unit. This work presents a new simulator that is intuitive, portable and extensible, serving as a teaching complement for teaching in Computer Structure. This simulator enables the definition of different sets of instructions and the execution and debugging of source code. It also allows to define the behavior of the processor by microprogramming instructions, also defined by the end-users.

WepSIM allows students to understand the operation of an elementary processor in a simple way, can be used from a mobile device or a computer with a modern Web browser without the need of being installed. In this way, students can interact with the simulator by learning and understanding the operation of the WepSIM elementary processor, including the mechanisms of interaction with the system software and integrating in the same tool both the microprogramming and the programming in assembler.

The main objective of this project was to develop a simulator, which, in contrast to existing

ones, could completely simulate the behavior of an elementary processor, allowing to check the state of the components in each clock cycle, so that students understand and assimilate in a simple and visual way the operation of a processor. We have also met all the other objectives presented in the introduction of the document:

- **O1**, A tool has been designed that simulates the execution of the instruction set specified in a computer called WepSIM, from the point of view of microprogramming and assembly programming.
- **O2**, The tool allows the specification of different instructions.
- **O3**, The tool unifies the microprogramming of a computer and programming in assembly language.
- **O4**, The tool allows the user to display in each clock cycle the state and behavior of the simulated computer.

On a personal level, this work has helped me to enter the world of scientific research. I have been able to apply a great deal of the knowledge acquired throughout the degree. In addition, I have learned important techniques of hardware modeling, compilation and simulation, which have a great utility and complexity and have served me to deepen even more in the knowledge acquired in the degree. For all this, it is very satisfying to see the final result obtained, since I have managed to overcome all the problems that have arisen throughout the project.

Contributions

The project carried out during this Bachelor Thesis fits with many of the subjects studied in the Degree in Computer Engineering of the University Carlos III of Madrid, highlighting the following topics in particular:

- **Computer technology** (Compulsory subject, First course) Where the hardware components and binary logic are introduced.
- **Computer structure** (Compulsory subject, Second course) Where the bases of the structure and operation of a computer are introduced.
- **Formal languages and Automata theory** (Compulsory subject, Second course) Where the bases are introduced about the formal languages and grammars.

- **Operating systems** (Compulsory subject, Second course) Where the bases of the operation of the operating system are introduced.
- **Computer architecture** (Compulsory subject, Third course) Where the bases of the architecture of a computer are introduced.
- **Operating systems design** (Compulsory subject, Third course) Where the bases of the design of the different modules of an operating system are introduced.
- **Software development projects management** (Compulsory subject, Third course) Where the bases for the management and management of a software development project are introduced.

Publications

This bachelor thesis has permitted to make an important contribution to the field of teaching in Structure and Computer Architecture. In addition, the following scientific articles have been published:

- **A. Calderón, F. García-Carballeira, and J. Prieto**, "WepSIM: Simulador modular e interactivo de un procesador elemental para facilitar una visión integrada de la microprogramación y la programación en ensamblador", *Enseñanza y aprendizaje de ingeniería de computadores*, vol. 6, 35-53, 2016. [15]
- **J. Prieto, A. Calderón, F. García-Carballeira, and S. Alonso-Monsalve**, "WepSIM: simulador integrado de microprogramación y programación en ensamblador", *Jornadas sarteco 2016*. [23]

In addition, at the time of delivery of this document, there is also another item sent waiting for its acceptance.

Future works

Currently, there are many lines of future work in which we are working on:

- Regarding improvements in the hardware model:

1. Introduce more hardware elements, such as a cache, in order to expand the contents of the subject included in the tool.
 2. Introduce a hardware model based on a pipeline, allowing the use of the tool in those subjects that employ this architecture model.
- Regarding improvements in the software model:
 3. Add semantic revision of the code, allowing to identify and notify the programming errors to the user.
 4. Add new sets of instructions to the tool such as the ARM assembler, allowing the use of different languages in the tool.
 5. Study the MIPS / ARM assembler generated with GCC/Clang so that it can be used directly in WepSIM.
 - Regarding improvements in the tool:
 6. Add an automatic correction module for practices to the tool, so that students can practice with it and check the validity of their exercises.
 7. Migrate the tool to a mobile application using the Apache Cordova plugin, so that the tool is not limited to its usage in a web browser.

Apéndice B

Manual de usuario

Este anexo presenta un manual de usuario detallado de WepSIM. En primer lugar, se indica el ciclo de ejecución en el simulador, explicando la gestión del microcódigo, el código ensamblador y la realización de una simulación. Después, se indica el formato de definición del microcódigo. Tras ello, se indica el formato del código ensamblador. Por último, se indica la definición de un juego de instrucciones y un código ensamblador base.

Ejecución en WepSIM

El ciclo de trabajo para el usuario en WepSIM supone (normalmente) los siguientes pasos:

- Cargar un juego de instrucciones de trabajo.
- Cargar un programa en ensamblador que utilice el juego de instrucciones de trabajo.
- Ir a la pantalla principal del simulador para realizar la ejecución del programa ensamblador que utilice el juego de instrucciones de trabajo.

Gestión de microcódigo en WepSIM

Un fichero de texto con las tres secciones antes comentadas (microcódigo, nombrado de registro y definición de pseudo-instrucciones) se carga en la pantalla de Firmware. Para acceder a dicha pantalla se ha de ir al menú (parte superior derecha) e indicar la opción Firmware (Figura B-1).

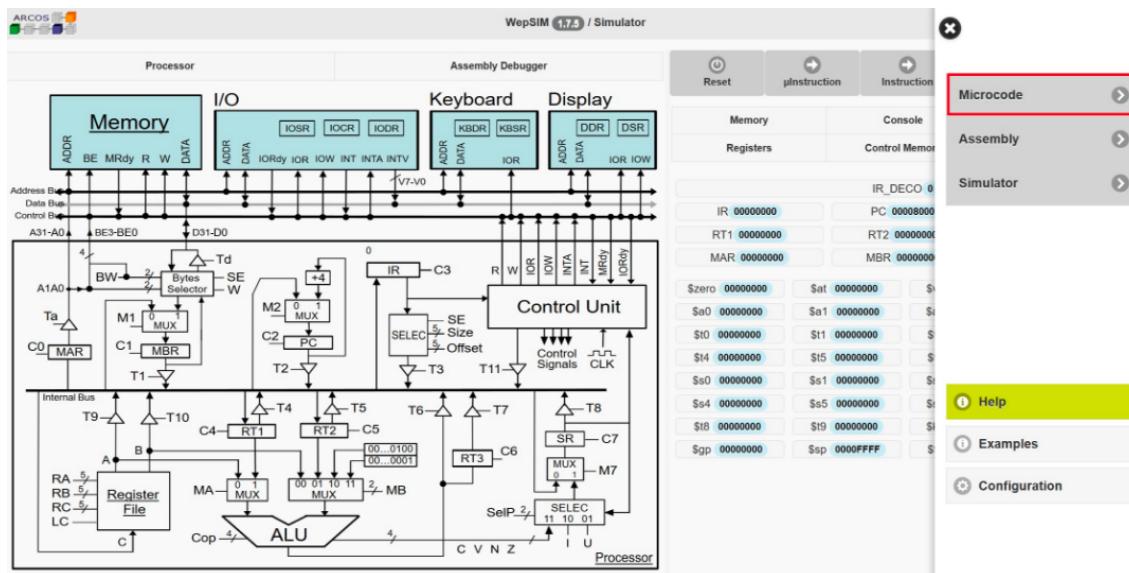


Figura B-1: Pantalla principal: opción de carga de juego de firmware.

A continuación aparecerá una pantalla con un cuadro de texto que permite indicar el firmware (las tres secciones antes comentadas). Es posible cargar un firmware existente usando el botón "Load" de la barra inferior, modificar un firmware anteriormente cargado o salvar el firmware en curso con el botón "Save" de la barra inferior. Una vez se indique el firmware es preciso hacer clic en el botón "micro-compile" para pasar a binario y cargar en la memoria de control dicho firmware, como se muestra en la Figura B-2.

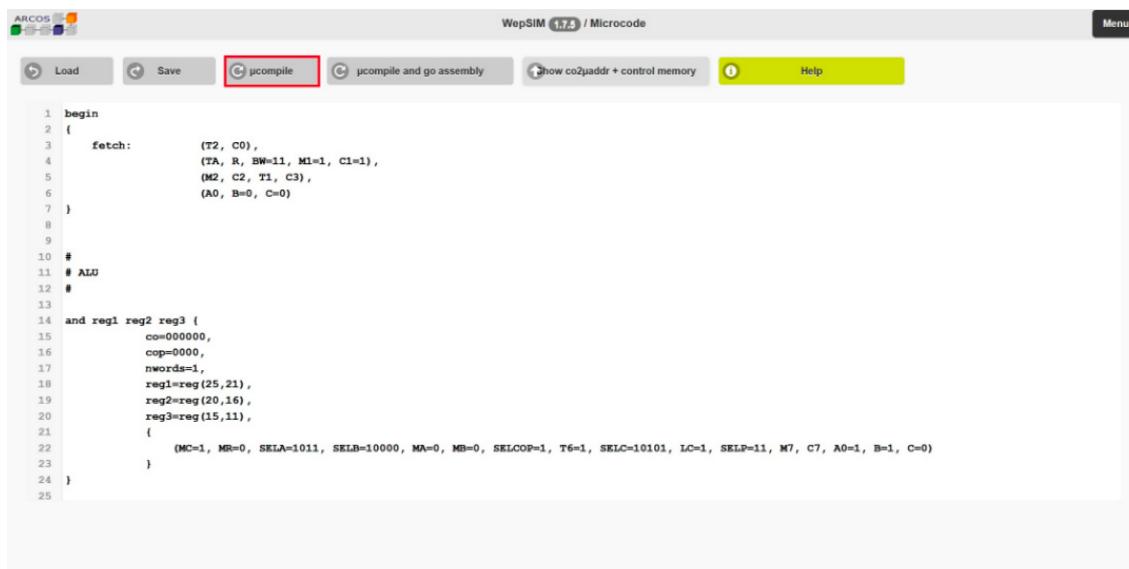
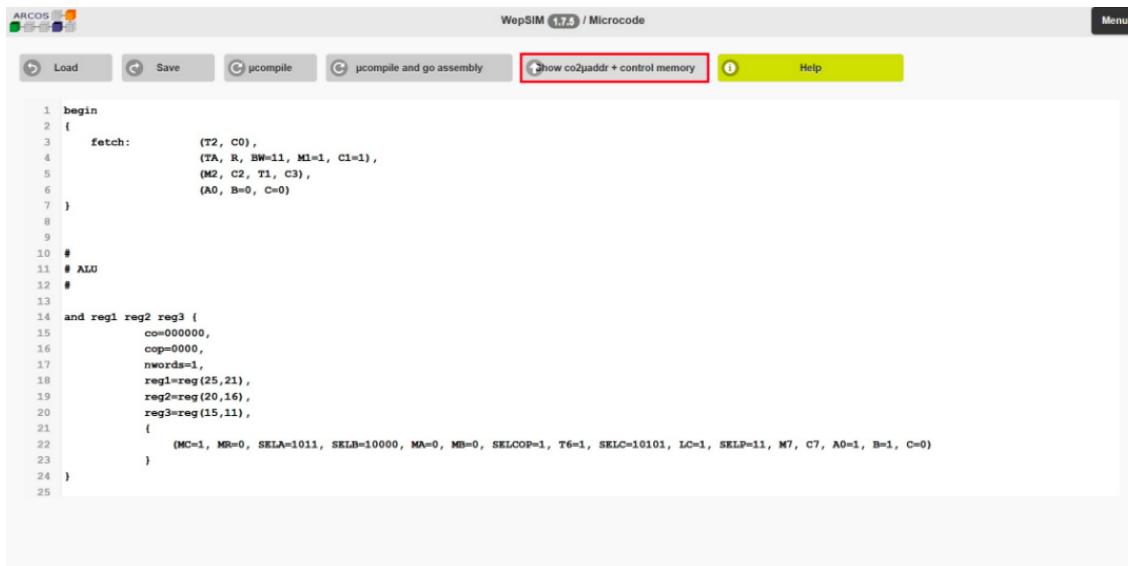


Figura B-2: Pantalla firmware: cuadro de texto con firmware a cargar.

Una vez cargado el nuevo firmware correctamente aparecerá la pantalla mostrada en la

Figura B-3 en la que se muestra la memoria de control con los valores que almacena.



The screenshot shows the WepSIM 1.7.5 / Microcode interface. At the top, there are buttons for Load, Save, µcompile, µcompile and go assembly, and a red-highlighted button labeled "show co2jaddr + control memory". Below these are buttons for Help and Menu. The main area contains assembly code:

```

1 begin
2 {
3     fetch:      (T2, C0),
4             (TA, R, BW=11, M1=1, C1=1),
5             (M2, C2, T1, C3),
6             (A0, B=0, C=0)
7 }
8
9
10 #
11 # ALU
12 #
13
14 and reg1 reg2 reg3 {
15     cov=000000,
16     cop=0000,
17     nwords=1,
18     reg1=reg(25,21),
19     reg2=reg(20,16),
20     reg3=reg(15,11),
21     {
22         (MC=1, MR=0, SELA=10111, SELB=10000, MA=0, MB=0, SELCOP=1, T6=1, SELC=10101, LC=1, SELP=11, M7, C7, A0=1, B=1, C=0)
23     }
24 }
25

```

Figura B-3: Pantalla firmware: firmware finalmente compilado.

El siguiente paso es la carga del programa ensamblador que permita probar este firmware, para ello podemos pulsar el botón “Assembly” de la barra inferior.

Gestión de código ensamblador en WepSIM

Un fichero de texto con las dos secciones antes comentadas (datos y código) se carga en la pantalla de Ensamblador. Para acceder a dicha pantalla se ha de ir al menú (parte superior derecha) e indicar la opción Assembly (B-4).

A continuación aparecerá una pantalla con un cuadro de texto que permite indicar el código en ensamblador. Es posible cargar un código existente usando el botón "Load" de la barra superior, modificar un código anteriormente cargado o salvar el código actualmente en cargado con el botón "Save" de la barra inferior.

Una vez se indique el código es preciso hacer clic en el botón "Compile" para pasar a binario y cargar en la memoria de principal el binario resultante, tal y como se muestra en la Figura B-5.

Una vez compilado, se pasará a la pantalla indicada en la Figura B-6 donde se mostrará el contenido de la memoria principal en binario.

El siguiente paso es ir a la pantalla principal para ejecutar la combinación de firmware y ensamblador cargado, para lo que ha de pulsar el botón "Simulador" de la barra inferior.

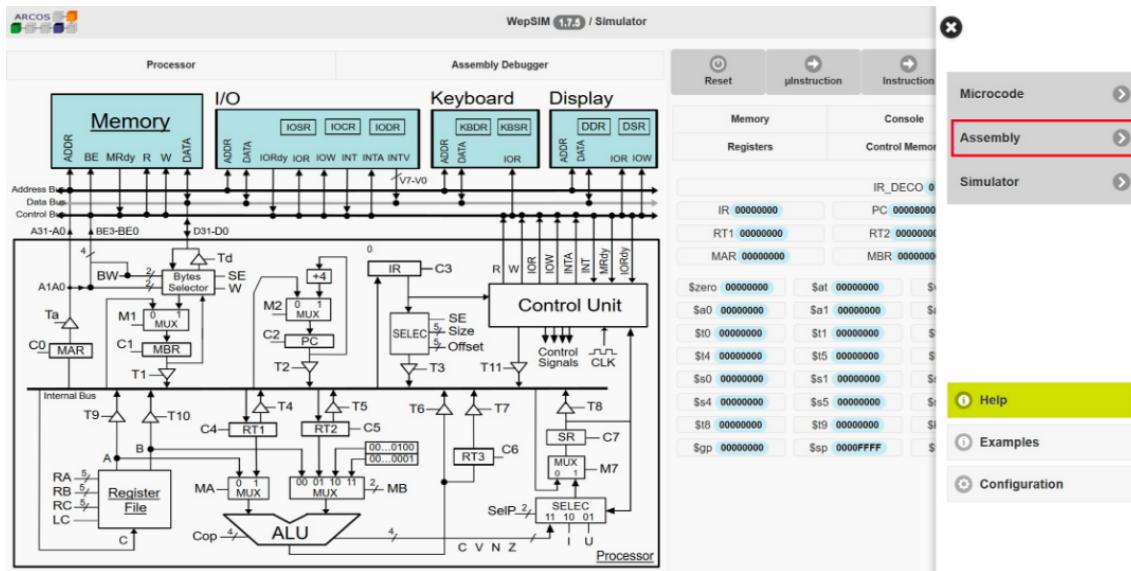


Figura B-4: Pantalla principal: opción de carga de código ensamblador.

```

1 .kdata
2     vector: .word rt_i0
3     .word rt_i1
4     .word rt_i2
5
6     msgi0: .asciiz "INT: 0\n"
7     msgi1: .asciiz "INT: 1\n"
8     msgi2: .asciiz "INT: 2\n"
9
10 .ktext
11 sys_print: lb $27 ($t0)
12     li $1 0
13     beq $27 $1 fin1
14     out $27 0x1000
15     li $1 1
16     add $t0 $t0 $1
17     b sys_print
18 fin1: reti
19
20 rt_i0: # 1.- interruption
21     la $t0 msgi0
22     b sys_print
23
24 rt_i1: # 2.- interruption
25     la $t0 msgi1

```

Figura B-5: Pantalla ensamblador: cuadro de texto con ensamblador a cargar.

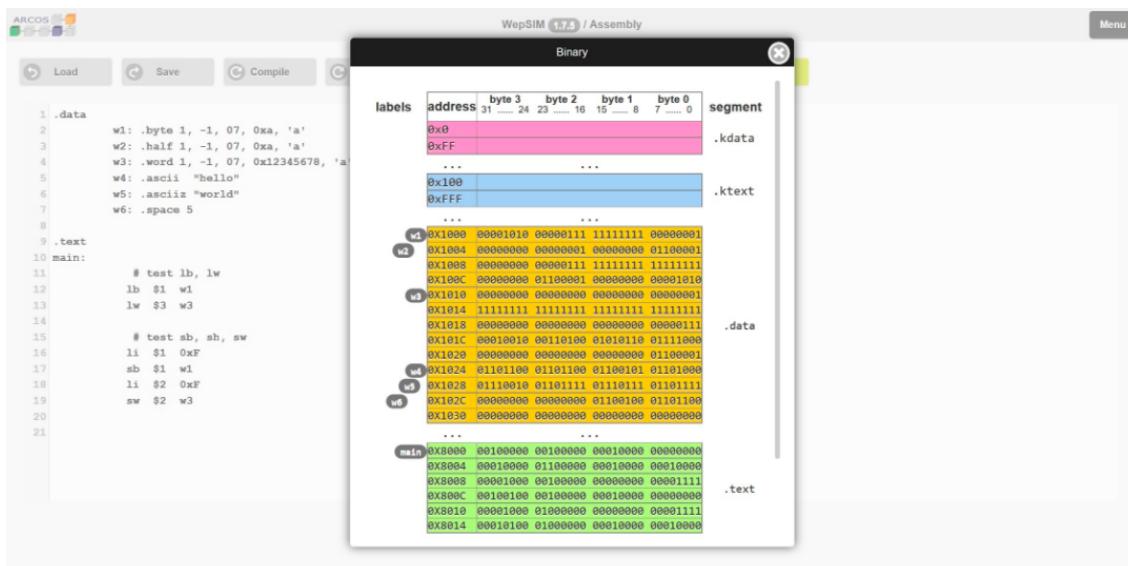


Figura B-6: Pantalla ensamblador: código finalmente compilado.

Simulación en WepSIM

Estando en la pantalla principal es posible visualizar:

- El contenido de la memoria de control (Figura B-7), con las señales que están asociadas a cada ciclo. Para ello ha de pulsar el botón “Control Memory” en la barra de botones situada en la parte superior derecha de la pantalla principal.
- Se destaca las señales que en el presente ciclo de reloj están activadas en azul con letra algo más grande.
- Se dispone de una barra de desplazamiento para poder inspeccionar todo el contenido de la memoria de control.
- El contenido de la memoria principal (Figura B-8), con las instrucciones en ensamblador a ejecutar. Para estas, es posible establecer un punto de ruptura haciendo clic en la columna breakpoints. Al establecer un punto de ruptura aparecerá un icono en dicha columna.

Estando en la pantalla principal es posible ejecutar:

- Microinstrucción a microinstrucción pulsando el botón “micro-Instrucción” (Figura B-9), de forma que se pasará al siguiente ciclo de reloj y se generarán las señales de control asociadas.

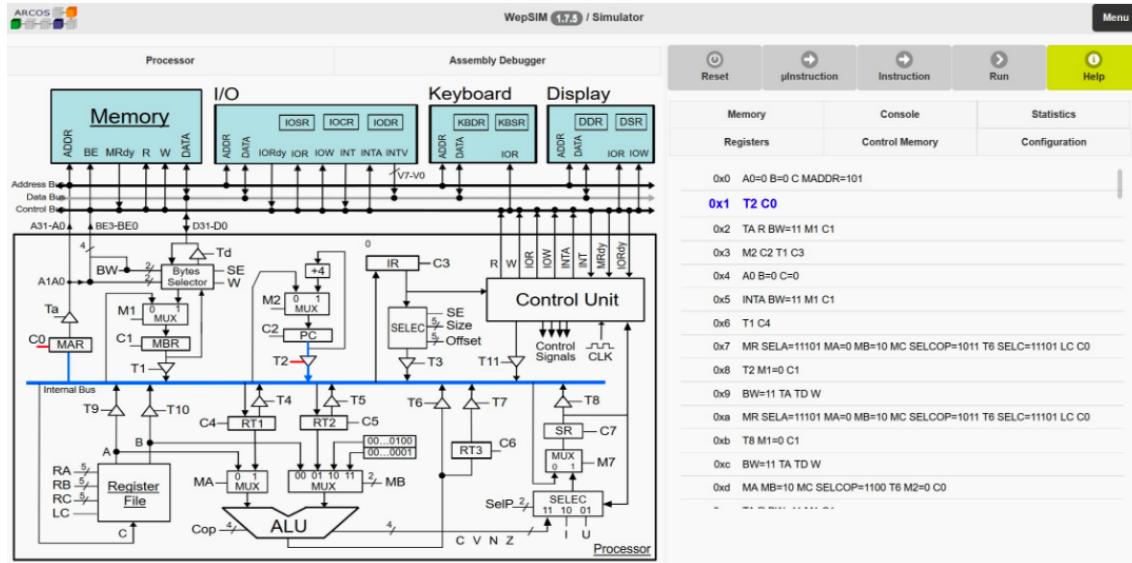


Figura B-7: Pantalla principal: visualización de la memoria de control.

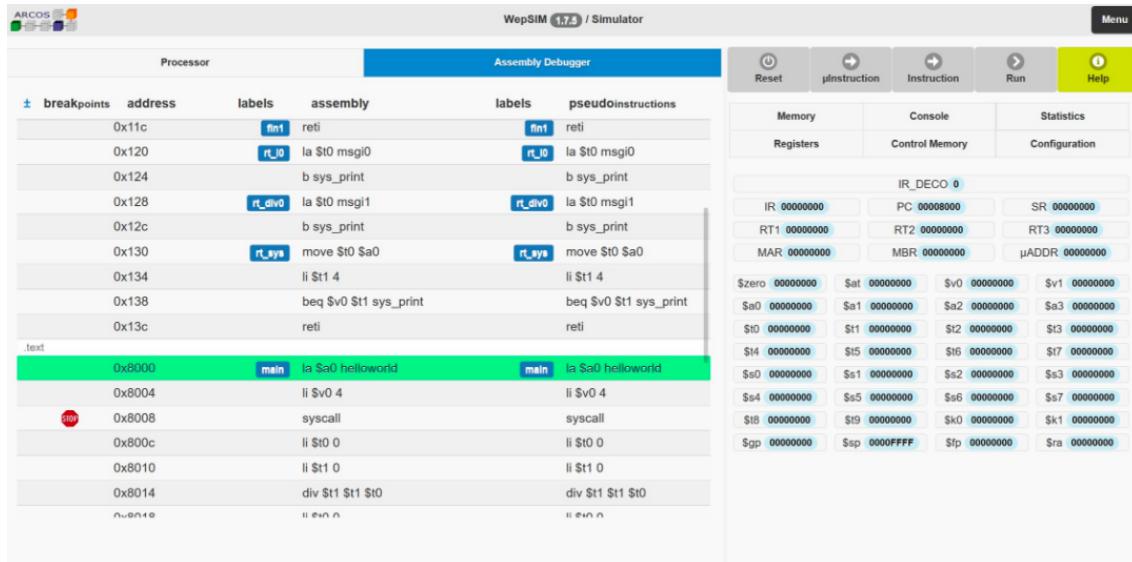


Figura B-8: Pantalla principal: visualización del código en la memoria principal.

- Instrucción a instrucción pulsando el botón “Instrucción” (Figura 26 B-9) de forma que se generarán todos los ciclos de reloj asociados al microprograma de la instrucción, parando al principio del fetch.

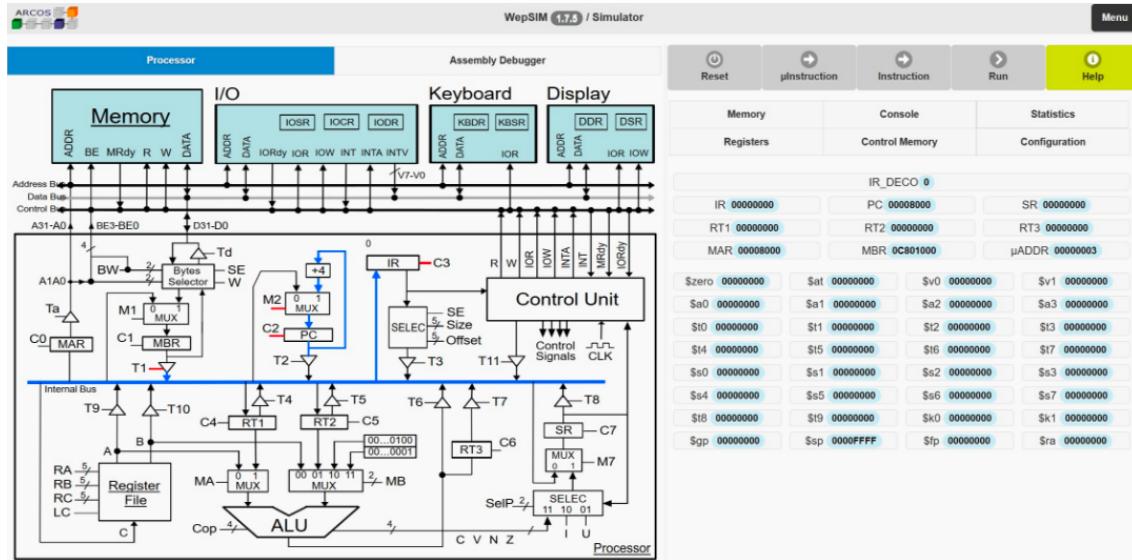


Figura B-9: Pantalla principal: opciones para la ejecución.

Dando al botón “Registers” en la barra de botones situada en la parte superior derecha de la pantalla principal (Figura B-9) es posible ver cómo los valores de los registros son modificados durante la ejecución.

Es posible visualizar también la unidad de control como se muestra en la Figura B-10.

Así como también es posible reiniciar la ejecución haciendo clic al botón “Reset” situado en la barra de botones que aparece en la parte superior de la pantalla.

Formato del microcódigo

El microcódigo se carga a través de un fichero de texto que tiene tres secciones:

1. Lista de microprogramas.
2. Nombrado de registros.
3. Pseudo-instrucciones.

La lista de microprogramas comienza con el código del fetch. Un ejemplo de microprograma de fetch básico sería:

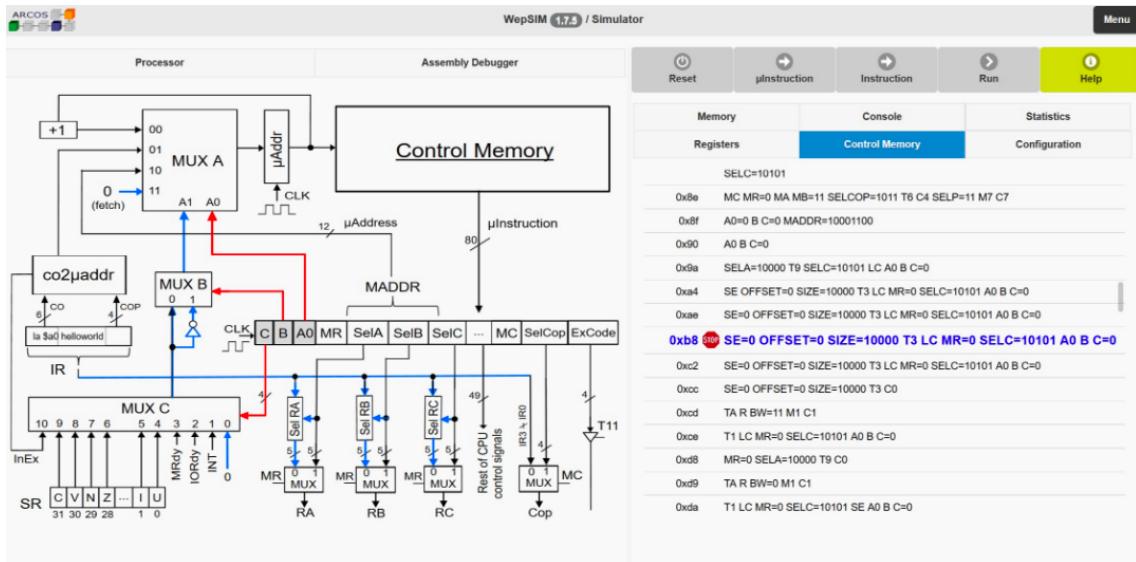


Figura B-10: Pantalla principal: visualización de la unidad de control.

```

begin
{
fetch: (T2, C0=1),
(Ta, Td, R, BW=11, C1),
(M2, C2, T1, C3),
(A0, B=0, C=0000)
}

```

Las señales de control situadas entre los paréntesis (y) se corresponden con las señales a activar en un ciclo de reloj. Así el fetch necesita cuatro ciclos de reloj, el último se corresponde con la decodificación (incluida en dentro del fetch). A continuación se encuentran el resto de microprogramas asociados a cada instrucción. Cada microprograma sigue el siguiente formato:

```
inst1 campo1 campo2 campo3
```

```
{
co=000000,
nwords=1,
campo1=reg(25,21),
campo2=reg(20,16),
campo3=reg(15,11),
{
```

```
(Cop=1001, SelP=11, C7, T6, LC, SelA=10100, SelB=01111, SelC=10111,  
A0=1, B=1, C=0)  
}  
}
```

Donde la primera línea describe el nombre de la instrucción (inst1) y los parámetros que tiene (registros, valores inmediatos, etc.). A continuación se abre un bloque con llaves para describir dicha instrucción.

El primer campo del ejemplo mostrado (co) indica los 6 bits que identifica únicamente la instrucción. Esto no es del todo cierto para las instrucciones aritmético-lógicas puesto que es posible compartir el mismo código de operación y según el valor del campo cop diferenciarse. Un ejemplo de este caso sería:

```
inst1 campo1 campo2 campo3  
{  
    co=000000,  
    cop=0000,  
    nwords=1,  
    campo1=reg(25,21),  
    campo2=reg(20,16),  
    campo3=reg(15,11),  
    {  
        (Cop=1001, SelP=11, C7, T6, LC, SelA=10100, SelB=01111, SelC=10111,  
        A0=1, B=1, C=0)  
    }  
}
```

Los siguientes campos indican para cada uno de los parámetros qué tipo es (registro, inmediato o dirección) así como el bit de inicio y final (valor de 0 a 31) entre los cuales está el valor de dicho parámetro. El tipo de parámetro se indica con:

- parametro1 = reg(bit-final, bit-inicio), para un registro, siendo los 5 bits que identifican el registro a usar.
- parametro1 = inm(bit-final, bit-inicio), para un valor inmediato.

- parametro1 = addr(bit-final, bit-inicio)rel, para una dirección relativa con respecto al PC.
- parametro1 = addr(bit-final, bit-inicio)abs, para una dirección absoluta.

A continuación se abre un bloque donde se indicará el microprograma correspondiente a la instrucción. Las señales de cada ciclo están entre paréntesis y se separan dichos ciclos por coma. Si solo hay un ciclo no es necesario la coma. Dentro de los paréntesis se indicarán las señales y el valor correspondiente. Si es una señal de un bit, con solo poner el nombre de la señal se entenderá que su valor es uno. El valor de la señal se ha de indicar en binario, usando tantos bits como tenga asociado la señal. Las señales y su valor correspondientes se separan usando una coma. Para el nombrado de registros se precisa indicar la etiqueta que se usará para cada uno de los 32 registros del banco de registros. Un ejemplo de esta sección es:

```
registers
```

```
{
    0=$zero,
    1=$at,
    2=$v0,
    3=$v1,
    4=$a0,
    5=$a1,
    6=$a2,
    7=$a3,
    .
    .
    .
    24=$t8,
    25=$t9,
    26=$k0,
    27=$k1,
    28=$gp,
    29=$sp (stack_pointer),
    30=$fp,
```

```
31=$ra  
}
```

En la que se indican los nombres usados en la arquitectura MIPS32. En este ejemplo el registro 29 etiquetado con \$sp tiene el atributo “(stack_pointer)” para indicar que será usado como puntero de pila.

Por último es posible definir pseudo-instrucciones. Un ejemplo de pseudoinstrucción sería:

```
Pseudoinstructions  
{  
    li reg num  
    {  
        lui reg sel(31,16,num) ;  
        ori reg reg sel(15,0,num)  
    }  
}
```

Formato del código ensamblador

El código en ensamblador se describe en un fichero de texto con una primera sección de datos (.data) y una segunda sección para el código (.text).

En la sección de datos es posible definir las variables y constantes que se alojarán en el segmento de datos de la memoria principal. Dicha sección comienza con la directiva .data. Las directivas que permiten especificar los distintos tipos de datos que pueden definirse son:

- **.ascii:** va seguida de la cadena de caracteres, instruyendo al ensamblador para crear una zona de memoria con datos, y almacenar en ella la cadena que se indique.
- **.asciiz:** va seguida de la cadena de caracteres, instruyendo al ensamblador para crear una zona de memoria con datos, y almacenar en ella la cadena que se muestra terminado por un byte con valor cero.
- **.byte:** va seguida de uno o más valores que formarán parte del valor de la variable. En caso de varios valores, dichos valores se separarán por coma. Para los valores se puede usar: carácter, octal, hexadecimal y decimal.

- **.half:** va seguida de uno o más valores que formarán parte del valor de la variable. En caso de varios valores, dichos valores se separarán por coma. Para los valores se puede usar: octal, hexadecimal y decimal.
- **.word:** va seguida de uno o más valores que formarán parte del valor de la variable. En caso de varios valores, dichos valores se separarán por coma. Para los valores se puede usar: octal, hexadecimal y decimal.
- **.space:** va seguida del número de bytes, en formato decimal, que el usuario desea reservar en memoria.

El formato de un valor de los tipos de datos comentados es:

- **Cadena de caracteres:** secuencia de caracteres entre comillas dobles.
Por ejemplo: "hola 123\n"
- **Carácter:** carácter entre comillas simples.
Por ejemplo: 'c'.
- **Octal:** un número que comienza por cero y sus dígitos son menores que ocho.
Por ejemplo: 012.
- **Hexadecimal:** un número que comienza por el prefijo 0x y sus dígitos son del cero al nueve y las letras a, b, c, d, e y f.
Por ejemplo: 0x12.
- **Decimal:** un número que no está en formato octal o hexadecimal con dígitos comprendidos entre el cero y el nueve (ambos incluidos).
Por ejemplo: 12.

En la sección de código es posible definir las subrutinas que se alojarán en el segmento de código de la memoria principal. Dicha sección comienza con la directiva .text.

Es posible usar comentarios de línea usando el carácter #. Todo lo que hay a continuación de este carácter hasta el final de línea será ignorado por el ensamblador.

Un ejemplo programa sería:

```
.data

    age1: .word 0x12345678, 20

    age2: .word 20 , 10

    resultado: .word 0

    # 32-bit word initialized with decimal

    texto: .ascii "Hola\t"

    texto2: .asciiz "Hola\t"

    hueco: .space 16

.text

.globl main

main: li $3 2

        li $4 1

        li $5 0
```

Ejemplos

Juego de instrucciones

```

begin
{
    (A0=0, B=1, C=1, MADDR=fetch),
    # push PC
    (MR=1, SELA=11101, MA=0, MB=10, MC=1, SELCOP=1011, T6=1, SELC=11101, LC=1, C
     (T2=1, M1=0, C1),
     (BW=11, TA=1, TD=1, W=1)
    # push SR
    (MR=1, SELA=11101, MA=0, MB=10, MC=1, SELCOP=1011, T6=1, SELC=11101, LC=1, C
     (T8=1, M1=0, C1),
     (BW=11, TA=1, TD=1, W=1),
    # MBR <- DB <- INTV
    (INTA, BW=11, M1=1, C1=1),
    # RT1 <- MBR
    (T1=1, C4=1),
    # MAR <- RT1*4
    (MA=1, MB=10, MC=1, SELCOP=1100, T6, M2=0, C0),
    # MBR <- MP[MAR]
    (TA=1, R=1, BW=11, M1=1, C1=1),
    # PC <- MAR
    (T1, M2=0, C2),
    fetch: (T2, C0),
    (TA, R, BW=11, M1=1, C1=1),
    (M2, C2, T1, C3),
    (A0, B=0, C=0)
}

```

```
#  
# INT  
#  
  
reti {  
    co=111110,  
    nwords=1,  
    {  
        # pop SR  
        (MR=1, SELA=11101, T9, CO),  
        (MR=1, SELA=11101, MA=0, MB=10, MC=1, SELCOP=1010, T6=1, SELC=11101, LC=1),  
        (TA=1, R=1, BW=11, M1=1, C1),  
        (T1=1, M7=0, C7),  
        # pop PC  
        (MR=1, SELA=11101, T9, CO),  
        (MR=1, SELA=11101, MA=0, MB=10, MC=1, SELCOP=1010, T6=1, SELC=11101, LC=1),  
        (TA=1, R=1, BW=11, M1=1, C1),  
        (T1=1, M2=0, C2, A0=1, B=1 ,C=0)  
    }  
}  
  
#  
# ALU  
#  
  
and reg1 reg2 reg3 {  
    co=000000,  
    cop=0000,  
    nwords=1,  
    reg1=reg(25,21),  
    reg2=reg(20,16),
```

```
reg3=reg(15,11),
{
  (MC=1, MR=0, SELA=1011, SELB=10000, MA=0, MB=0, SELCOP=1, T6=1, SELC=10101, LC=1,
}
}

or reg1 reg2 reg3 {
  co=000000,
  cop=0001,
  nwords=1,
  reg1=reg(25,21),
  reg2=reg(20,16),
  reg3=reg(15,11),
{
  (MC=1, MR=0, SELA=1011, SELB=10000, MA=0, MB=0, SELCOP=10, T6=1, SELC=10101, LC=1,
}
}

not reg {
  co=000000,
  cop=0010,
  nwords=1,
  reg=reg(25,21),
{
  (MC=1, MR=0, SELA=10101, MA=0, SELCOP=11, T6=1, SELC=10101, LC=1, SELP=11, M7, C,
}
}

xor reg1 reg2 reg3 {
  co=000000,
  cop=0011,
  nwords=1,
```

```
    reg1=reg(25,21),  
    reg2=reg(20,16),  
    reg3=reg(15,11),  
    {  
        (MC=1, MR=0, SELA=1011, SELB=10000, MA=0, MB=0, SELCOP=100, T6=1, SELC=10101, LC=0)  
    }  
}  
  
add reg1 reg2 reg3 {  
    co=000000,  
    cop=1001,  
    nwords=1,  
    reg1=reg(25,21),  
    reg2=reg(20,16),  
    reg3=reg(15,11),  
    {  
        (MC=1, MR=0, SELA=1011, SELB=10000, MA=0, MB=0, SELCOP=1010, T6=1, SELC=10101, LC=0)  
    }  
}  
  
sub reg1 reg2 reg3 {  
    co=000000,  
    cop=1010,  
    nwords=1,  
    reg1=reg(25,21),  
    reg2=reg(20,16),  
    reg3=reg(15,11),  
    {  
        (MC=1, MR=0, SELB=1011, SELA=10000, MA=0, MB=0, SELCOP=1011, T6=1, SELC=10101, LC=0)  
    }  
}
```

```
mul reg1 reg2 reg3 {
    co=000000,
    cop=1011,
    nwords=1,
    reg1=reg(25,21),
    reg2=reg(20,16),
    reg3=reg(15,11),
    {
        (MC=1, MR=0, SELA=1011, SELB=10000, MA=0, MB=0, SELCOP=1100, T6=1, SELC=10101, I
    }
}

div reg1 reg2 reg3 {
    co=000000,
    cop=1100,
    nwords=1,
    reg1=reg(25,21),
    reg2=reg(20,16),
    reg3=reg(15,11),
    {
        (MC=1, MR=0, SELB=1011, SELA=10000, MA=0, MB=0, SELCOP=1101, T6=1, SELC=10101, I
    }
}

#
# MV/L*
#
move reg1 reg2 {
    co=000001,
    nwords=1,
```

```
    reg1=reg(25,21),
    reg2=reg(20,16),
{
    (SELA=10000, T9, SELC=10101, LC, A0=1, B=1, C=0)
}
}

li reg val {
    co=000010,
    nwords=1,
    reg=reg(25,21),
    val=inm(15,0),
{
    (SE=1, OFFSET=0, SIZE=10000, SE=1, T3=1, LC=1, MR=0, SELC=10101, A0=1, B=1, C=0)
}
}

liu reg val {
    co=111100,
    nwords=1,
    reg=reg(25,21),
    val=inm(15,0),
{
    (SE=1, OFFSET=0, SIZE=10000, SE=0, T3=1, LC=1, MR=0, SELC=10101, A0=1, B=1, C=0)
}
}

la reg addr {
    co=000011,
    nwords=1,
    reg=reg(25,21),
    addr=address(15,0)abs,
```

```
{  
    (SE=0, OFFSET=0, SIZE=10000, T3=1, LC=1, MR=0, SELC=10101, A0=1, B=1, C=0)  
}  
  
}  
  
la reg addr {  
    co=111111,  
    nwords=2,  
    reg=reg(25,21),  
    addr=address(63,32)abs,  
{  
    (SE=0, OFFSET=0, SIZE=10000, T3=1, LC=1, MR=0, SELC=10101, A0=1, B=1, C=0)  
}  
}  
  
lw reg addr {  
    co=000100,  
    nwords=1,  
    reg=reg(25,21),  
    addr=address(15,0)abs,  
{  
    (SE=0, OFFSET=0, SIZE=10000, T3=1, C0=1),  
    (TA=1, R=1, BW=11, M1=1, C1=1),  
    (T1=1, LC=1, MR=0, SELC=10101, A0=1, B=1, C=0)  
}  
}  
  
lb reg1 (reg2) {  
    co=100101,  
    nwords=1,  
    reg1 = reg(25,21),  
    reg2 = reg(20,16),
```

```
{  
    (MR=0, SELA=10000, T9=1, C0),  
    (TA=1, R=1, BW=00, M1=1, C1=1),  
    (T1=1, LC=1, MR=0, SELC=10101, SE=1, A0=1, B=1, C=0)  
}  
}  
  
lbu reg1 (reg2) {  
    co=101111,  
    nwords=1,  
    reg1 = reg(25,21),  
    reg2 = reg(20,16),  
    {  
        (MR=0, SELA=10000, T9=1, C0),  
        (TA=1, R=1, BW=00, M1=1, C1=1),  
        (T1=1, LC=1, MR=0, SELC=10101, SE=0, A0=1, B=1, C=0)  
    }  
}  
  
lw reg1 (reg2)  
{  
    co=100000,  
    nwords=1,  
    reg1 = reg(25,21),  
    reg2 = reg(20,16),  
    {  
        (MR=0, SELA=10000, T9, C0),  
        (TA=1, R=1, BW=11, M1=1, C1=1),  
        (T1=1, LC=1, MR=0, SELC=10101, A0=1, B=1, C=0)  
    }  
}
```

```
sw reg addr {
    co=000101,
    nwords=1,
    reg=reg(25,21),
    addr=address(15,0)abs,
{
    (SE=0, OFFSET=0, SIZE=10000, T3=1, C0=1),
    (MR=0, SELA=10101, T9=1, M1=0, C1=1),
    (BW=11, TA=1, TD=1, W=1, A0=1, B=1, C=0)
}
}

lb reg addr {
    co=001000,
    nwords=1,
    reg=reg(25,21),
    addr=address(15,0)abs,
{
    (SE=0, OFFSET=0, SIZE=10000, T3=1, C0=1),
    (TA=1, R=1, BW=00, SE=1, M1=1, C1=1),
    (T1=1, LC=1, MR=0, SELC=10101, A0=1, B=1, C=0)
}
}

sb reg addr {
    co=001001,
    nwords=1,
    reg=reg(25,21),
    addr=address(15,0)abs,
{
    (SE=0, OFFSET=0, SIZE=10000, T3=1, C0=1),
    (MR=0, SELA=10101, T9=1, M1=0, C1=1),
```

```
(BW=0, TA=1, TD=1, W=1, A0=1, B=1, C=0)
}

}

#  
# IN/OUT  
#  
  
in reg val {  
    co=001010,  
    nwords=1,  
    reg=reg(25,21),  
    val=inm(15,0),  
    {  
        (SE=0, OFFSET=0, SIZE=10000, T3=1, C0=1),  
        (TA=1, IOR=1, M1=1, C1=1),  
        (T1=1, LC=1, MR=0, SELC=10101, A0=1, B=1, C=0)  
    }  
}  
  
out reg val {  
    co=001011,  
    nwords=1,  
    reg=reg(25,21),  
    val=inm(15,0),  
    {  
        (SE=0, OFFSET=0, SIZE=10000, T3=1, C0=1),  
        (MR=0, SELA=10101, T9=1, M1=0, C1=1),  
        (TA=1, TD=1, IOW=1, A0=1, B=1, C=0)  
    }  
}
```

```
#  
# b*  
  
#  
  
b offset {  
    co=001100,  
    nwords=1,  
    offset=address(15,0)rel,  
    {  
        (T2, C4),  
        (SE=1, OFFSET=0, SIZE=10000, T3, C5),  
        (MA=1, MB=1, MC=1, SELCOP=1010, T6, C2, A0=1, B=1, C=0)  
    }  
}  
  
beq reg reg offset {  
    co=001101,  
    nwords=1,  
    reg=reg(25,21),  
    reg=reg(20,16),  
    offset=address(15,0)rel,  
    {  
        (T8, C5),  
        (SELA=10101, SELB=10000, MC=1, SELCOP=1011, SELP=11, M7, C7),  
        (A0=0, B=1, C=110, MADDR=bck2ftch),  
        (T5, M7=0, C7),  
        (T2, C4),  
        (SE=1, OFFSET=0, SIZE=10000, T3, C5),  
        (MA=1, MB=1, MC=1, SELCOP=1010, T6, C2, A0=1, B=1, C=0),  
        bck2ftch: (T5, M7=0, C7),  
    }
```

```
(A0=1, B=1, C=0)
}

}

bne reg reg offset {
    co=001110,
    nwords=1,
    reg=reg(25,21),
    reg=reg(20,16),
    offset=address(15,0)rel,
{
    (T8, C5),
    (SELA=10101, SELB=10000, MC=1, SELCOP=1011, SELP=11, M7, C7),
    (A0=0, B=0, C=110, MADDR=bck3ftch),
    (T5, M7=0, C7),
    (T2, C4),
    (SE=1, OFFSET=0, SIZE=10000, T3, C5),
    (MA=1, MB=1, MC=1, SELCOP=1010, T6, C2, A0=1, B=1, C=0),
    bck3ftch: (T5, M7=0, C7),
    (A0=1, B=1, C=0)
}
}

bge reg reg offset {
    co=001111,
    nwords=1,
    reg=reg(25,21),
    reg=reg(20,16),
    offset=address(15,0)rel,
{
    (T8, C5),
    (SELA=10101, SELB=10000, MC=1, SELCOP=1011, SELP=11, M7, C7),
```

```

        (A0=0, B=0, C=111, MADDR=bck4ftch),
        (T5, M7=0, C7),
        (T2, C4),
        (SE=1, OFFSET=0, SIZE=10000, T3, C5),
        (MA=1, MB=1, MC=1, SELCOP=1010, T6, C2, A0=1, B=1, C=0),
        bck4ftch: (T5, M7=0, C7),
        (A0=1, B=1, C=0)
    }

}

blt reg reg offset {
    co=010000,
    nwords=1,
    reg=reg(25,21),
    reg=reg(20,16),
    offset=address(15,0)rel,
{
    (T8, C5),
    (SELA=10101, SELB=10000, MC=1, SELCOP=1011, SELP=11, M7, C7),
    (A0=0, B=1, C=111, MADDR=bck5ftch),
    (T5, M7=0, C7),
    (T2, C4),
    (SE=1, OFFSET=0, SIZE=10000, T3, C5),
    (MA=1, MB=1, MC=1, SELCOP=1010, T6, C2, A0=1, B=1, C=0),
    bck5ftch: (T5, M7=0, C7),
    (A0=1, B=1, C=0)
}
}

bgt reg reg offset {
    co=010001,
    nwords=1,

```

```
    reg=reg(25,21),
    reg=reg(20,16),
    offset=address(15,0)rel,
{
    (T8, C5),
    (SELA=10101, SELB=10000, MC=1, SELCOP=1011, SELP=11, M7, C7),
    (A0=0, B=0, C=111, MADDR=bck6ftch),
    (A0=0, B=0, C=110, MADDR=bck6ftch),
    (T5, M7=0, C7),
    (T2, C4),
    (SE=1, OFFSET=0, SIZE=10000, T3, C5),
    (MA=1, MB=1, MC=1, SELCOP=1010, T6, C2, A0=1, B=1, C=0),
    bck6ftch: (T5, M7=0, C7),
    (A0=1, B=1, C=0)
}
}

ble reg reg offset {
    co=010010,
    nwords=1,
    reg=reg(25,21),
    reg=reg(20,16),
    offset=address(15,0)rel,
{
    (T8, C5),
    (SELA=10101, SELB=10000, MC=1, SELCOP=1011, SELP=11, M7, C7),
    (A0=0, B=0, C=111, MADDR=ble_ys),
    (A0=0, B=0, C=110, MADDR=ble_ys),
    (T5, M7=0, C7),
    (A0=1, B=1, C=0),
    ble_ys: (T5, M7=0, C7),
    (T2, C4),
```

```
(SE=1, OFFSET=0, SIZE=10000, T3, C5),  
(MA=1, MB=1, MC=1, SELCOP=1010, T6, C2, A0=1, B=1, C=0)  
}  
}  
  
#  
# j*  
#  
  
j addr {  
    co=010011,  
    nwords=1,  
    addr=address(15,0)abs,  
    {  
        (SE=0, OFFSET=0, SIZE=10000, T3=1, M2=0, C2=1, A0=1, B=1, C=0)  
    }  
}  
  
jal addr {  
    co=010100,  
    nwords=1,  
    addr=address(15,0)abs,  
    {  
        (T2, SELC=11111, MR=1, LC),  
        (SE=0, OFFSET=0, SIZE=10000, T3=1, M2=0, C2=1, A0=1, B=1, C=0)  
    }  
}  
  
jr reg1 {  
    co=010101,  
    nwords=1,
```

```
reg1=reg(25,21),  
{  
    (SELA=10101, T9=1, C2=1, A0=1, B=1, C=0)  
}  
}  
  
#  
# Misc  
#  
  
nop {  
    co=010110,  
    nwords=1,  
    {  
        (A0=1, B=1, C=0)  
    }  
}  
  
srl reg1 reg2 val {  
    co=010111,  
    nwords=1,  
    reg1=reg(25,21),  
    reg2=reg(20,16),  
    val=inm(5,0),  
    {  
        (SE=1, OFFSET=0, SIZE=110, T3=1, C4=1),  
        (MC=1, MR=0, SELA=10000, SELB=10000, MA=0, MB=0, SELCOP=1, T6=1, SELC=10101),  
        loop9: (A0=0, B=0, C=110, MADDR=bck2ftch),  
        (MC=1, MR=0, SELA=10101, SELB=10101, MA=0, MB=0, SELCOP=101, T6=1, LC=1, SELC=10101),  
        (MC=1, MR=0, MA=1, MB=11, SELCOP=1011, T6=1, C4=1, SELP=11, M7, C7),  
        (A0=0, B=1, C=0, MADDR=loop9),  
    }  
}
```

```
bck9ftch: (A0=1, B=1, C=0)
}

}
```

```
registers
```

```
{
    0=$zero,
    1=$at,
    2=$v0,
    3=$v1,
    4=$a0,
    5=$a1,
    6=$a2,
    7=$a3,
    8=$t0,
    9=$t1,
    10=$t2,
    11=$t3,
    12=$t4,
    13=$t5,
    14=$t6,
    15=$t7,
    16=$s0,
    17=$s1,
    18=$s2,
    19=$s3,
    20=$s4,
    21=$s5,
    22=$s6,
    23=$s7,
    24=$t8,
```

```
25=$t9,  
26=$k0,  
27=$k1,  
28=$gp,  
29=$sp (stack_pointer),  
30=$fp,  
31=$ra  
}  
  
pseudoinstructions
```

```
{  
    lii reg num  
    {  
        li reg sel(31,16,num) ;  
        li reg sel(15,0,num)  
    }  
}
```

Código ensamblador

```
.data

matrix: .word 1, 0, 3, 0, 0, 1
        .word 1, 2, 0, 1, 1, 0

.text

counting:
# result = 0
li $v0 0

# for ($t0=0; $t0 < $a1; ...
li $t0 0
b1: bge $t0 $a1 f1

# for ($t1=0; $t1 < $a2; ...
li $t1 0
b2: bge $t1 $a2 f2

# $t2 = ($t0*$a2 + $t1) * 4
mul $t2 $t0 $a2
add $t2 $t2 $t1
li $t3 4
mul $t2 $t2 $t3

# elto = *( $a0 + $t2 )
add $t2 $a0 $t2
lw $t2 ($t2)
```

```
bne $t2 $0 nozero
li $t3 1
add $v0 $v0 $t3
```

nozero:

```
# ... $t1++)
li $t3 1
add $t1 $t1 $t3
b b2
```

f2:

```
# ... $t0++)
li $t3 1
add $t0 $t0 $t3
b b1
```

f1:

```
# return
jr $ra
```

main:

```
# counting (matrix, 2, 6)
la $a0 matrix
li $a1 2
li $a2 6
jal counting
```


Bibliografía

- [1] "P8080e simulator." https://www.datsi.fi.upm.es/docencia/Estructura/U_Control/#HERRAMIENTAS. Accessed: 2017-04-26.
- [2] R.-F. Yen and Y. Kim, "Development and implementation of an educational simulator software package for a specific microprogramming architecture," *IEEE Transactions on Education*, no. 1, pp. 1–11, 1986.
- [3] A. S. Tanenbaum, *Structured Computer Organization 2nd*. ACM, 1984.
- [4] J. R. Larus, *Spim s20: A mips r2000 simulator*. Center for Parallel Optimization, Computer Sciences Department, University of Wisconsin, 1990.
- [5] I. Aguilar Juárez and J. R. Heredia Alonso, "Simuladores y laboratorios virtuales para ingeniería en computación," 2013.
- [6] K. Vollmar and P. Sanderson, "Mars: an education-oriented mips assembly language simulator," in *ACM SIGCSE Bulletin*, vol. 38, pp. 239–243, ACM, 2006.
- [7] S. R. Vegdahl, "Mipspilot: A compiler-oriented mips simulator," *Journal of Computing Sciences in Colleges*, vol. 24, no. 2, pp. 32–39, 2008.
- [8] M. I. Garcia, S. Rodríguez, A. Pérez, and A. García, "p88110: A graphical simulator for computer architecture and organization courses," *IEEE Transactions on Education*, vol. 52, no. 2, pp. 248–256, 2009.
- [9] I. Branovic, R. Giorgi, and E. Martinelli, "Webmips: a new web-based mips simulation environment for computer architecture education," in *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, p. 19, ACM, 2004.
- [10] J. C. Garrido, "Arquitectura y diseño de sistemas web modernos," *InforMAS, Revista de Ingeniería Informática del CIIRM*, no. 1, 2004.
- [11] J. C. PEREZ, F. G. CARBALLEIRA, J. D. G. Sánchez, and D. E. Singh, *Problemas resueltos de estructura de computadores*. Ediciones Paraninfo, SA, 2015.
- [12] Institute for Electrical and Electronics Engineers, "IEEE Recommended Practice for Software Requirements Specifications," *IEEE*, pp. 830–1998, 1998.
- [13] BOE, 19 de enero de 2008, "Real Decreto 1720/2007, de 21 de diciembre, por el que se aprueba el Reglamento de desarrollo de la Ley Orgánica 15/1999, de 12 de diciembre, de protección de datos de carácter personal," *Boletín Oficial del Estado*, 17:4103-4136.

- [14] "The GNU Lesser General Public License." <http://www.gnu.org/licenses/licenses.html>, Last visited, June 2016.
- [15] A. C. Mateos, F. G. Carballeira, and J. P. Cepeda, "Wepsim: Simulador modular e interactivo de un procesador elemental para facilitar una visión integrada de la microprogramación y la programación en ensamblador," *Enseñanza y aprendizaje de ingeniería de computadores: Revista de Experiencias Docentes en Ingeniería de Computadores*, no. 6, pp. 35–53, 2016.
- [16] J. P. Bennett, *Introduction to compiling techniques: a first course using ANSI C, LEX and YACC*. McGraw-Hill, Inc., 1996.
- [17] ARCOS Research Group , "WepSIM user manual." <https://wepsim.github.io/wepsim-manual.pdf>, Last visited, June 2017.
- [18] Software Testing Fundamentals, "Verification vs Validation." <http://softwaretestingfundamentals.com/verification-vs-validation/>, Last visited, Mar. 2016.
- [19] Accelerated Technologies, Inc, "The Human Condition: A Justification for Rapid Prototyping," *Time Compression Technologies*, vol. 3 no. 3, p. 1, May 1998.
- [20] Benington, Herbert D., "Production of Large Computer Programs," *IEEE Annals of the History of Computing (IEEE Educational Activities Department)*, p. 350–361, Oct. 1983.
- [21] B. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer* 21, 5, pp. 61–72, 1988.
- [22] BOE, 6 de agosto de 2012, "Resolución de 2 de agosto de 2012, de la Dirección General de Tributos, sobre el tipo impositivo aplicable a determinadas entregas de bienes y prestaciones de servicios en el Impuesto sobre el Valor Añadido," *Boletín Oficial del Estado*, 187:56055-56060.
- [23] J. Prieto-Cepeda, F. Garcia-Carballeira, A. Calderon, and S. Alonso-Monsalve, "WepSIM: simulador integrado de microprogramación y programación en ensamblador," *XXVII Jornadas de Paralelismo (JP2016)*, September 2016.