UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

# DISEÑO E IMPLEMENTACIÓN DE UN SOC EN UN FPGA BASADO EN EL ISA DE RISC-V: INSTRUCCIONES DE EXTENSIÓN ATÓMICA E INTERFAZ DE USUARIO.

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELÉCTRICO

MARCELO IVÁN URRUTIA SALAZAR

PROFESOR GUÍA:
FRANCISCO RIVERA SERRANO

MIEMBROS DE LA COMISIÓN:
GIANLUCA D'AGOSTINO MATUTE
JOSÉ GONZALEZ GARCIA

SANTIAGO DE CHILE
2023

# Resumen

En esta memoria se continúa el desarrollo del SoC diseñado por Gianluca Vincenzo D'Agostino Matute en su memoria de título del año 2021. Este sistema es capaz de trabajar con el ISA (*Instruction Set Architecture*) libre de RISC-V y posee: el conjunto de instrucciones base I para números enteros, la extensión de instrucciones M para multiplicaciones y divisiones de enteros, y la extensión de instrucciones F de punto flotante precisión simple.

El trabajo incorpora al sistema la extensión de instrucciones A o atómicas en el SoC desarrollado. Esta permite la sincronización en la memoria de datos. En el trabajo se propone un rediseño del SoC ya implementado, cambiando la estructura de algunos módulos existentes, e incorporando otros nuevos. El SoC obtenido, es un sistema que realiza correctamente la lectura de las nuevas instrucciones atómicas. Para comprobar su correcto funcionamiento, se realizan simulaciones dentro del *software* que se ha utilizado durante el trabajo.

# Tabla de Contenido

# Índice de Tablas

# Índice de Ilustraciones

# Capítulo 1
# Introducción

## 1.1. Contexto

En esta Memoria se continúa el desarrollo del SoC diseñado por Gianluca Vincenzo D'Agostino Matute en su memoria de título del año 2021. Este fue desarrollado para el ISA (*Instruction Set Architecture*) libre de RISC-V y posee: el set de instrucciones base I para números enteros, el set de instrucciones M para multiplicaciones y divisiones de enteros, y el set de instrucciones F de punto flotante precisión simple.

## 1.2. Motivación

En los últimos años la industria tecnológica ha vivido la aparición de RISC-V, una arquitectura y conjunto de instrucciones de bajo nivel libre desarrollado en la Universidad de California en Berkeley que, al ser libre y con el objetivo de abarcar instrucciones simples a diferencia de otras arquitecturas, da como resultado el desarrollo de hardware simple y de bajo consumo energético para su ejecución. Esto último ha despertado el interés de diversos actores en la industria para fomentar su desarrollo y crecimiento. [5] [2]

Empresas como *Nvidia*, *AMD*, *SiFive* y *Huawei*, son algunas de las muchas empresas interesadas en esta arquitectura e instrucciones [5], por lo que adentrarse en la comprensión de estas crea dos objetivos pedagógico interesantes que son una motivación para este y futuros trabajos.

1. Tener el SoC para que los estudiantes puedan ejecutar, probar y comprender el funcionamiento de las instrucciones de RISC-V en el curso Arquitectura de Computadores [5].
2. Usar el SoC como un espacio de desarrollo escalable en proyectos como memorias o trabajos de título, que sea capaz de incorporar nuevas instrucciones con cada mejora usando conocimiento y herramientas entregadas en cursos como [3], [4] y [5]. Este trabajo es la primera incorporación de nuevas instrucciones desde su primer desarrollo.

## 1.3. Objetivos del trabajo

El objetivo general es incluir, en el desarrollo del SoC, el diseño e implementación del conjunto de instrucciones "atómicas", denominada "A". Este conjunto contiene instrucciones

que leen, modifican y escriben "atómicamente" (indivisiblemente) la memoria para admitir la sincronización entre múltiples RISC-V *harts* (o *threads*) que se ejecutan en el mismo espacio de memoria.

Para el desarrollo del objetivo general, se desarrollarán distintos objetivos específicos:

1. Clasificar las funciones atómicas, reconociendo sus estructuras y utilidad.
2. Estudiar la incorporación de hardware y la nueva estructura de este para el uso de funciones atómicas
3. Diseñar e implementar el hardware necesario para el correcto funcionamiento del nuevo SoC y del entorno de ejecución.
4. Poner en práctica la metodología de diseño *Top-Down* en el desarrollo de sistemas digitales.
5. Utilizar una Tarjeta de Desarrollo FPGA (Nexys A7 de Digilent) como plataforma de prototipado y para la implementación final.
6. Dominio y uso de HDL (*Hardware Description Language*), el cual se utiliza para sintetizar todos los diseños digitales realizados sobre el FPGA y para realizar las simulaciones.
7. Creación de pruebas de funcionamiento de los diseños obtenidos, previo a la implementación en hardware sobre el FPGA.
8. Obtener el nuevo SoC, haciendo mejoras en la interfaz de usuario, y corroborar su correcto funcionamiento.

# Capítulo 2

# Marco Teórico

En esta sección, se presentan los fundamentos teóricos básicos para abordar el problema. Primeramente, se enseñan los fundamentos generales del trabajo, entendiendo que es un sistema digital, o un computador y sus partes. Luego, comprender a grandes rasgos la arquitectura e ISA de RISC-V, el cual se utiliza en el trabajo, y explicando las extensiones ya aplicadas en el SoC. Finalmente, se explica en detalle qué es la extensión atómica y qué hacen sus instrucciones. Cabe destacar que las definiciones entregadas con excepción de la extensión atómica en esta sección están basadas en el trabajo de Gianluca [2], así mismo, estas están basadas en los cursos [3], [4] y [5].

## 2.1. Sistemas Digitales

Un sistema digital es un dispositivo electrónico que utiliza lógica binaria para ejecutar tareas. Prácticamente, cualquier función que pueda realizar una máquina puede ser controlada mediante un sistema digital. Los sistemas digitales pueden ser:

1. Combinacionales o secuenciales. Los circuitos combinacionales no dependen de un reloj, es decir, las salidas dependen solo de las señales de entradas. Los circuitos secuenciales además de depender de las señales de entrada, dependen de un reloj que determina cuando se definen las salidas. Es importante destacar que los circuitos combinacionales forman parte de los secuenciales.
2. Si es una máquina de estados (*Finite State Machine* o FSM), puede ser de Moore, donde las salidas solo dependen del estado actual, o de Mealy, donde también dependen de las entradas. Una máquina de estados es secuencial.
3. Ser de propósito general, como un microcontrolador o un computador, o específico, como una calculadora o un sistema de alarmas.

## 2.2. Arquitectura de Computadores

Un computador es un sistema digital de propósito general, el cual es programable y debe interactuar con el usuario. Este tipo de sistemas, en general, se componen de los siguientes elementos:

1. **La CPU:** o procesador, se encarga de procesar toda la información del sistema.

- Tiene una arquitectura, es la estructura general del sistema. Depende de un conjunto de instrucciones (ISA: *Instruction Set Architecture*) que el procesador debe ser capaz de leer y ejecutar. Dichas instrucciones se codifican en binario y se almacenan en memoria. Según la filosofía de diseño del conjunto de instrucciones, puede ser simples o complejas:
  - **RISC** (*Reduced Instruction Set Computer*): Busca simplificar al máximo el conjunto de instrucciones requeridas.
  - **CISC** (*Complex Instruction Set Computer*): Busca acercar instrucciones a los lenguajes de alto nivel, es decir, tienen mayor complejidad.
- Para ejecutar una instrucción, la CPU posee un *Datapath*, que es un conjunto de unidades funcionales (unidades lógicas que realizan tareas específicas como la ALU, Muxes, etc) cuyo objetivo es realizan operaciones de procesamiento de datos, acceso a registros y manejo de buses de datos. Este *Datapath* posee una unidad de control que indica qué unidades funcionales están en uso. El *Datapath* puede poseer diferentes etapas de ejecución para realizar una instrucción, siendo las siguientes 5 etapas las más típicas:
  - **Fetch:** etapa en la que se lee en la memoria la instrucción a ejecutar.
  - **Decode:** etapa en la que se decodifica la instrucción en distintas señales y se leen los registros necesarios para su ejecución.
  - **Execute:** etapa en la que se ejecuta la operación aritmética/lógica de la instrucción.
  - **Memory:** etapa en la que se accede a la memoria si la instrucción lo necesita, tanto para guardar como para cargar valores.
  - **Write Back:** etapa en la que se guarda, si corresponde, el resultado de la instrucción en un registro.
- Puede ser *Single-Cycle*, *Multi-Cycle* o *Pipelined*:
  - Un procesador **Single-Cycle** (figura 2.1) funciona de tal manera que la instrucción se ejecuta en un solo ciclo de reloj, pasando por todo el *Datapath* antes de seguir con el siguiente ciclo. Debido a esto, el *Datapath* es más complejo.
  - El procesador **Multi-Cycle** presenta un *datapath* simplificado, esto gracias a que las unidades funcionales para cada etapa de la ejecución de la instrucción son la misma dado a que la instrucción se ejecuta en múltiples ciclos de reloj.
  - El procesador **Pipelined** (figura 2.2, es el estándar para un procesador de alto rendimiento. Su *datapath* está basado en el de un procesador *Single-Cycle*, pero es capaz de aumentar la frecuencia del reloj al separar el *datapath* en las distintas etapas (*fetch, decode, execute, memory y write back*) las cuales ahora se pueden ejecutar de forma paralela. La separación se hace con registros para guardar los resultados de cada etapa hasta el siguiente ciclo de reloj.

    Al tener distintas instrucciones ejecutándose de manera paralela, si una instrucción requiere algún dato que aún no se ha definido en su etapa, pero que debía cambiar por una instrucción anterior(ejemplo, estar leyendo un registro

que en teoría debería tener una suma, pero esa suma también se está ejecutando), es necesario recuperar ese valor de manera adelantada. A esto se le llama *Data Hazard*, y se define como la dependencia de datos entre etapas. De la misma manera, si se predice que se debe ejecutar una instrucción cuando en realidad no es así (debido a un salto de dirección o algo similar) se ejecuta un *Control Hazard*, el cual anula las instrucciones mal predichas.

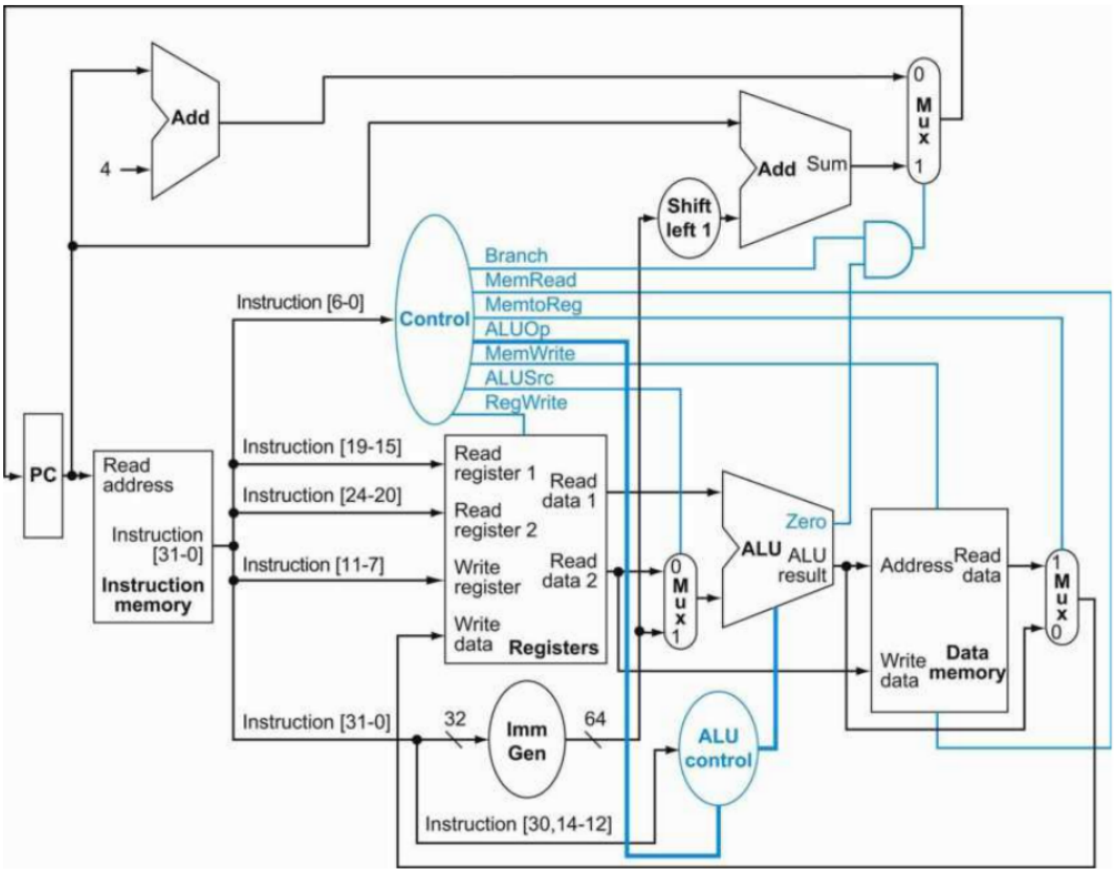


Figura 2.1: Ejemplo del *datapath* de un *Single Cycle*. Fuente: [7]

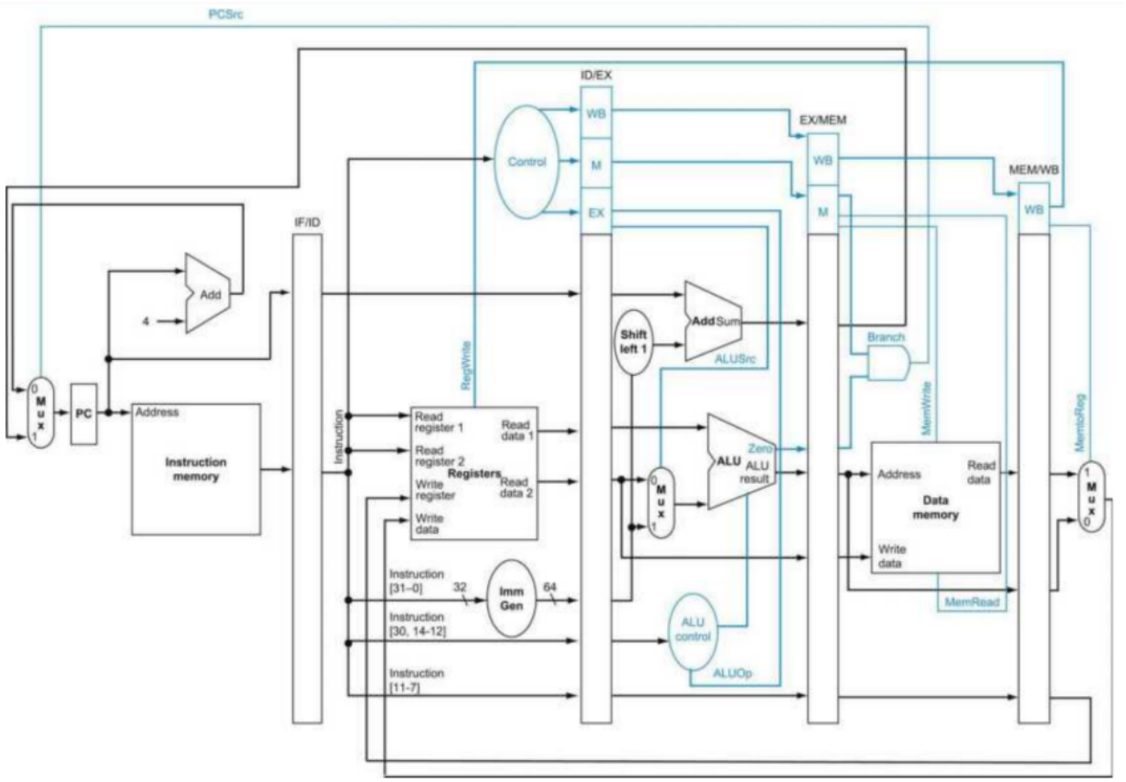


Figura 2.2: Ejemplo *datapath Pipeline*. Fuente: [7]

2. **Sistema de memoria:** la memoria se encarga de almacenar la información con la que trabaja todo el sistema, ya sean datos y/o instrucciones. Hay distintos tipos, clasificados por velocidad, y en consecuencia, por costos.
   - **Caché:** Memoria más cercana a la CPU (integrada en la cpu misma), la más veloz y costosa.Al quitar la energía de esta, se pierde información.
   - **RAM:** Memoria un poco más económica, por ende, con mayor almacenamiento y más lenta que la anterior. Al quitar la energía de esta, se pierde información.
   - **Disco duro:** Mecánico o sólido, el disco duro es más barato que la Memoria RAM, de mayor almacenamiento, pero velocidad mucho menor. Si es capaz de mantener la información luego de quitarle la alimentación (por un periodo considerablemente largo de tiempo).
3. **Dispositivos de entrada/salida (E/S):** estos son el medio por el cual el sistema recibe información e interactúa con el mundo real

## 2.3. FPGA y Targeta de Desarrollo a utilizar

Un FPGA (figura 2.3) (*Field Programmable Gate Arrays*)[3], hecha de silicio, es una pieza de hardware reprogramable físicamente, es decir, se puede cambiar su estructura interna reconectando los transistores que posee dentro. Esto permite implementar diferentes diseños de hardware en el mismo chip y actualizarlos de ser necesario.

La tarjeta utilizada corresponde a la Nexys A7, fabricada por Digilent y basada en el chip FPGA Artix-7 100T de Xilinx. Esta tarjeta es una plataforma de desarrollo de circuitos digitales completa que, en general, permite un desarrollo sin necesidad de otros componentes. Entre los componentes que posee se destacan [6]:

- 16 interruptores de usuario.
- 16 LED de usuario.
- 2 LED tri-color.
- 2 pantallas de 4 dígitos de 7 segmentos.
- 5 pulsadores + 1 de reinicio CPU + 1 de reinicio de configuración FPGA
- Puerto USB UART/JTAG compartido.
- Micrófono (PDM).
- Conector VGA de 12 bits.
- Conector de audio (+ salida de audio PWM).
- Conector Ethernet.
- Conector de host USB.
- Sensor de temperatura.
- Acelerómetro de 3 ejes.
- 16MB de Cellular RAM.
- Conector de tarjeta Mirco SD.
- 128 MiB DDR2
- puertos Pmod para señales XADC.
- puertos Pmod para I/O.

Por otra parte, es importante destacar que el Artix-7 100T cuenta con:

- 15.850 logic slides, cada uno con cuatro 6-input LUTs (look-up tables) y 8 flip-flops.
- 1.188 Kbits de fast block RAM.
- 240 DSP slices (Digital Signal Processing).
- Un analog-to-digital converter (XADC) en el chip.
- Velocidades de reloj interno superiores a 450MHz.
- 6 clock management tile (CMT), cada uno con phase-locked loop (PLL).

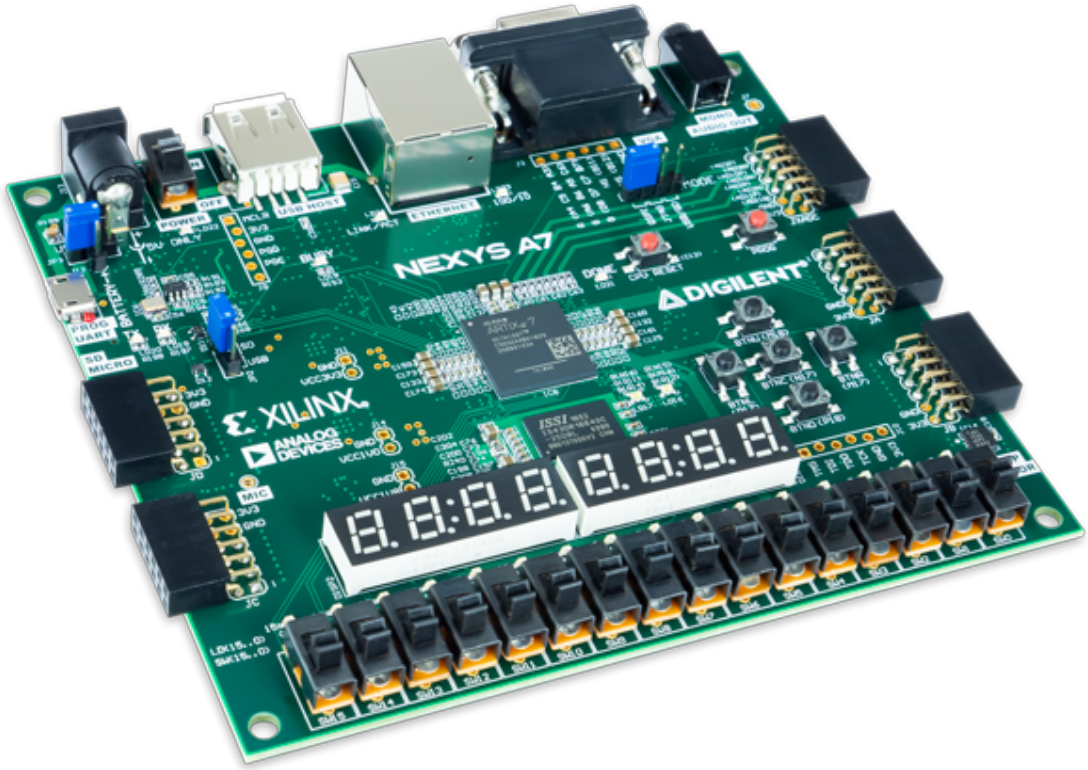


Figura 2.3: Tarjeta *NEXYS A7*. Fuente: [6]

## 2.4. RISC-V

La implementación se realizará utilizando la arquitectura e ISA de RISC-V [8], un procesador de Hardware Libre desarrollado en la University of California, Berkeley. A continuación, se presentan los principales objetivos del proyecto RISC-V:

- El desarrollo de un ISA completo, abierto y gratuito
- El desarrollo de un ISA útil para la implementación directa de hardware nativo, no solo para simulación o traducción binaria.
- El desarrollo de un ISA sencillo y eficiente, que evite el desarrollo de arquitecturas demasiado complicadas.
- Desarrollar un ISA segmentado, es decir, compuesto por un ISA básico con las instrucciones más simples, pero completamente funcional por si solo y por extensiones estandarizadas de este ISA base, que son útiles para el desarrollo de software de propósito general.
- Soporte para el estándar 2008 IEEE-754 para el cálculo de punto flotante.
- El desarrollo de las variantes en 32 bits (*word*) y 64 bits (*double*).

El nombre asignado al ISA base es I (con el prefijo RV32 o RV64, según la variante elegida), y se compone por instrucciones de cálculo de números enteros, cargas de números enteros, almacenamiento de números enteros e instrucciones de flujo de control. A continuación, se listan las extensiones estándares más importantes:

- M: extensión de multiplicación y división de enteros. Agrega instrucciones para multiplicar y dividir valores contenidos en los *integer registers*.
- A: extensión de instrucciones *atomic*. Agrega instrucciones que leen, modifican y escriben de forma "atómica" (sin división, evita la pérdida de información) la memoria para la sincronización entre *harts*.
- F: extensión de punto flotante de precisión simple. Agrega registros de punto flotante, instrucciones de precisión simple, cargas y almacenamiento de precisión simple.
- D: extensión de punto flotante de precisión doble. Expande los registros de punto flotante y agrega instrucciones computacionales de precisión doble, y cargas y almacenamiento de precisión doble.
- C: la extensión de instrucción comprimida proporciona formas más estrechas (16 bits) de instrucciones comunes.

## 2.5. Instrucciones Atómicas

La extensión de instrucción atómica estándar [8], denominada A, contiene instrucciones que leen, modifican y escriben atómicamente la memoria para admitir la sincronización entre

múltiples RISC-V *harts* que se ejecutan en el mismo espacio de memoria. Las dos formas de instrucciones atómicas proporcionadas son las instrucciones *load-reserved/store-conditional* y las instrucciones *atomic fetch-and-op memory (AMO)*. Ambos tipos de instrucciones atómicas admiten varios ordenamientos de consistencia de memoria que incluyen semántica desordenada, adquirida, liberada y secuencialmente consistente y permiten el uso de *word* ( instrucciones para 32 *bits*) y de *double* (para 64 *bits*). Los ordenamientos de consistencia de memoria y la utilización de esta función para *double* están fuera del alcance del trabajo, por lo que no serán abordados.

Las instrucciones atómicas son de tipo R. Esto define su estructura, otras instrucciones de este tipo son, por ejemplo, *add, xor y sub* pertenecientes al ISA base I [2]. Lo anterior significa que las instrucciones atómicas tienen una misma estructura en su datagrama (tabla 2.1), la cual, para una instrucción atómica de tipo *word* es la siguiente:

| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-0 |
|---|---|---|---|---|---|---|
| Funct5 | Funct2<br>aq-rl | rs2 | rs1 | Funct3<br>010 | rd | Opcode<br>0101111 |

Tabla 2.1: Datagrama de Instrucción Atómica [8]

En la estructura se puede ver que las instrucciones son de 32 bits, donde el *opcode* y el *funct3* son fijos, de hecho, el *opcode* es la información con la que se les distingue como extensión de RISC-V, y el *funct3* se utiliza para distinguir entre *word* y *double*.

Junto a lo anterior, se rescatan valores como el *funct5*, el cual sirve para distinguir qué instrucción atómica se está usando. El *funct2*, que no se ocupará en este trabajo. Y rd, rs1 y rs2, valores que corresponden a las direcciones de 5 *bits* de los registros (enteros): registro de destino (desde ahora x[rd]), registro 1 (desde ahora x[rs1]) y registro 2 (desde ahora x[rs2]) respectivamente.

Las operaciones atómicas de memoria complejas (llamadas así porque dependen la una de la otra) se realizan con las instrucciones *load-reserved* (LR.W) y *store-conditional* (SC.W), y es con ambas instrucciones que se realiza la sincronización, ya que con la carga-reserva y la carga condicional se accede a un espacio de memoria sabiendo que no hubo interrupciones.

LR.W carga en x[rd] un espacio de memoria correspondiente a la dirección almacenada en x[rs1] (desde ahora M(x[rs1])), luego reserva este mismo espacio de memoria.

SC.W verifica que la reserva del espacio x[rs1] siga vigente, de ser así, se escribe un 0 en x[rd] y se guarda el valor de x[rs2] en la memoria M(x[rs1]). En caso contrario, no guarda el valor en memoria, y escribe un valor distinto de 0 en x[rd].

Las instrucciones de operación de memoria atómica (con siglas en inglés AMO), realizan operaciones de lectura, modificación y escritura para la sincronización del multiprocesador. Estas instrucciones AMO cargan atómicamente un valor de la memoria M(x[rs1]) en el registro de destino x[rd]. Además de lo anterior, se aplica un operador binario al valor cargado con el valor del registro x[rs2], y el resultado de esa operación se almacena en la memoria en M(x[rs1]).

A continuación se muestran las instrucciones, explicadas una a una, con una ecuación que resume su funcionamiento y su datagrama en específico:

- **lr.w rd, (rs1):** carga el *word* de memoria de la dirección x[rs1], y se escribe el signo extendido de este en x[rd]. Junto a esto, se registra una reserva de ese *word* de memoria. (No se utiliza el registro x[rs2])

$$x[rd] = M(x[rs1]) \tag{2.1}$$

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| 00010 | aq | rl | 00000 | rs1 | 010 | rd | 0101111 |

Tabla 2.2: Distribución de bits lr.w. Fuente: [1]

- **sc.w rd, rs2, (rs1)** almacena en la dirección x[rs1] de memoria el *word* del registro x[rs2], siempre y cuando haya una reserva de load en esa dirección de memoria. Escribe 0 en x[rd] si tuvo éxito, o un código de error distinto de cero en caso contrario.

$$\begin{aligned} if(isReserved(M(x[rs1]))) : x[rd] = 0, else : x[rd] != 0 \\ M(x[rs1]) = M(x[rs2]) \end{aligned} \tag{2.2}$$

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| 00011 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 |

Tabla 2.3: Distribución de bits sc.w. Fuente: [1]

- **amoswap.w rd, rs2, (rs1)** atómicamente, se escribe el valor del *word* en memoria de la dirección x[rs1] en el registro x[rd], y se asigna x[rs2] a la dirección x[rs1] de la memoria.

$$\begin{aligned} x[rd] = M(x[rs1]) \\ M(x[rs1]) = x[rs2] \end{aligned} \tag{2.3}$$

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| 00001 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 |

Tabla 2.4: Distribución de bits amoswap.w. Fuente: [1]

- **amoadd.w rd, rs2, (rs1)** atómicamente, se escribe el valor del *word* en memoria de la dirección x[rs1] en el registro x[rd], y se asigna x[rs2]+x[rd] a la dirección x[rs1] de la memoria.

$$\begin{aligned} x[rd] &= M(x[rs1]) \\ M(x[rs1]) &= x[rd] + x[rs2] \end{aligned} \tag{2.4}$$

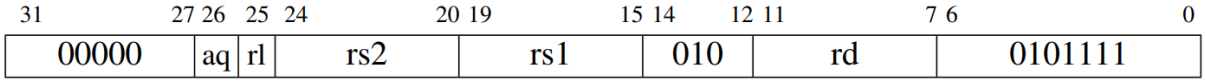


Tabla 2.5: Distribución de bits amoadd.w. Fuente: [1]

- **amoxor.w rd, rs2, (rs1)** atómicamente, se escribe el valor del *word* en memoria de la dirección x[rs1] en el registro x[rd], y se asigna x[rs2] XOR x[rd] a la dirección x[rs1] de la memoria.

$$\begin{aligned} x[rd] &= M(x[rs1]) \\ M(x[rs1]) &= x[rd] XOR x[rs2] \end{aligned} \tag{2.5}$$

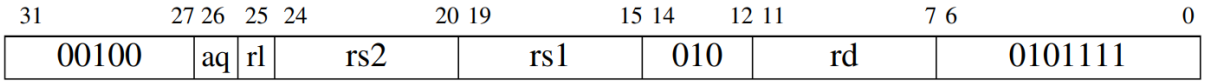


Tabla 2.6: Distribución de bits amoxor.w. Fuente: [1]

- **amoand.w rd, rs2, (rs1)** atómicamente, se escribe el valor del *word* en memoria de la dirección x[rs1] en el registro x[rd], y se asigna x[rs2] AND x[rd] a la dirección x[rs1] de la memoria.

$$\begin{aligned} x[rd] &= M(x[rs1]) \\ M(x[rs1]) &= x[rd] AND x[rs2] \end{aligned} \tag{2.6}$$

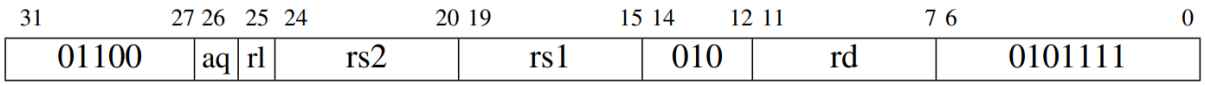


Tabla 2.7: Distribución de bits amoand.w. Fuente: [1]

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| 01000 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 |

Tabla 2.8: Distribución de bits amoor.w. Fuente: [1]

- **amoor.w rd, rs2, (rs1)** atómicamente, se escribe el valor del *word* en memoria de la dirección x[rs1] en el registro x[rd], y se asigna x[rs2] OR x[rd] a la dirección x[rs1] de la memoria.

$$\begin{aligned} x[rd] &= M(x[rs1]) \\ M(x[rs1]) &= x[rd]ORx[rs2] \end{aligned} \tag{2.7}$$

- **amomin.w rd, rs2, (rs1)** atómicamente, se escribe el valor del *word* en memoria de la dirección x[rs1] en el registro x[rd], y se asigna el mínimo, en complemento de dos, entre x[rs2] y x[rd] a la dirección x[rs1] de la memoria.

$$\begin{aligned} x[rd] &= M(x[rs1]) \\ M(x[rs1]) &= x[rd]MINx[rs2] \end{aligned} \tag{2.8}$$

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| 10000 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 |

Tabla 2.9: Distribución de bits amomin.w. Fuente: [1]

- **amomax.w rd, rs2, (rs1)** atómicamente, se escribe el valor del *word* en memoria de la dirección x[rs1] en el registro x[rd], y se asigna el máximo, en complemento de dos, entre x[rs2] y x[rd] a la dirección x[rs1] de la memoria.

$$\begin{aligned} x[rd] &= M(x[rs1]) \\ M(x[rs1]) &= x[rd]MAXx[rs2] \end{aligned} \tag{2.9}$$

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| 10100 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 |

Tabla 2.10: Distribución de bits amomax.w. Fuente: [1]

- **amominu.w rd, rs2, (rs1)** atómicamente, se escribe el valor del *word* en memoria de la dirección x[rs1] en el registro x[rd], y se asigna el mínimo, en comparación sin signo, entre x[rs2] y x[rd] a la dirección x[rs1] de la memoria.

$$\begin{aligned} x[rd] &= M(x[rs1]) \\ M(x[rs1]) &= x[rd] MINU x[rs2] \end{aligned} \tag{2.10}$$

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| 11000 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 |

Tabla 2.11: Distribución de bits amominu.w. Fuente: [1]

- **amomaxu.w rd, rs2, (rs1)** atómicamente, se escribe el valor del *word* en memoria de la dirección x[rs1] en el registro x[rd], y se asigna el máximo, en comparación sin signo, entre x[rs2] y x[rd] a la dirección x[rs1] de la memoria.

$$\begin{aligned} x[rd] &= M(x[rs1]) \\ M(x[rs1]) &= x[rd] MAXU x[rs2] \end{aligned} \tag{2.11}$$

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| 11100 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 |

Tabla 2.12: Distribución de bits amomaxu.w. Fuente: [1]

# Capítulo 3

# Estado del Arte

## 3.1. Diseño e implementación de un SoC en un FPGA basado en el ISA de RISC-V

- Memoria del año 2021 que se continúa en este trabajo de título. Escrita por Gianluca Vincenzo D'Agostino Matute.
- Utiliza la metodología Top-Down. Metodología que se vuelve a utilizar en esta memoria de título.
- Obtiene un SoC bastante completo, logrando implementar un juego de instrucciones RV32IMF, es decir, un SoC con las instrucciones bases de 32 bits y sus respectivas extensiones M y F.
- Completa el estudio con ejemplos para ver las capacidades del SoC.
- Se le pueden incorporar extensiones como la Atómica (A), la flotante de precisión Doble (D), y la extensión de instrucciones Comprimidas (C), estas, junto a las ya incorporadas, son las extensiones estándares más importantes.

Estas últimas extensiones no incorporadas son el eje fundamental para el desarrollo del proyecto ahora desarrollado.

## 3.2. Diseño de un Core RISC-V en SiFive

SiFive es una compañía de semiconductores especializada en la chips, procesadores, y SoC basados en el ISA de RISC-V. Entre sus desarrollos más importantes está la opción de comprar un *core* completamente personalizado por el usuario, donde se pueden elegir diversos procesadores base, donde cambia la arquitectura, o si son de 32 *bits* o 64 *bits*, entre otras opciones, y junto a ello, incorporar distintas extensiones a su diseño entre las cuales la instrucción atómica está incluida.

# Capítulo 4

# Diseño Propuesto

Para el diseño, o rediseño del sistema a implementar, se parte, como base, de la estructura propuesta en [2], donde el diagrama de bloques simplificado se ve de la siguiente manera (figura 4.1:

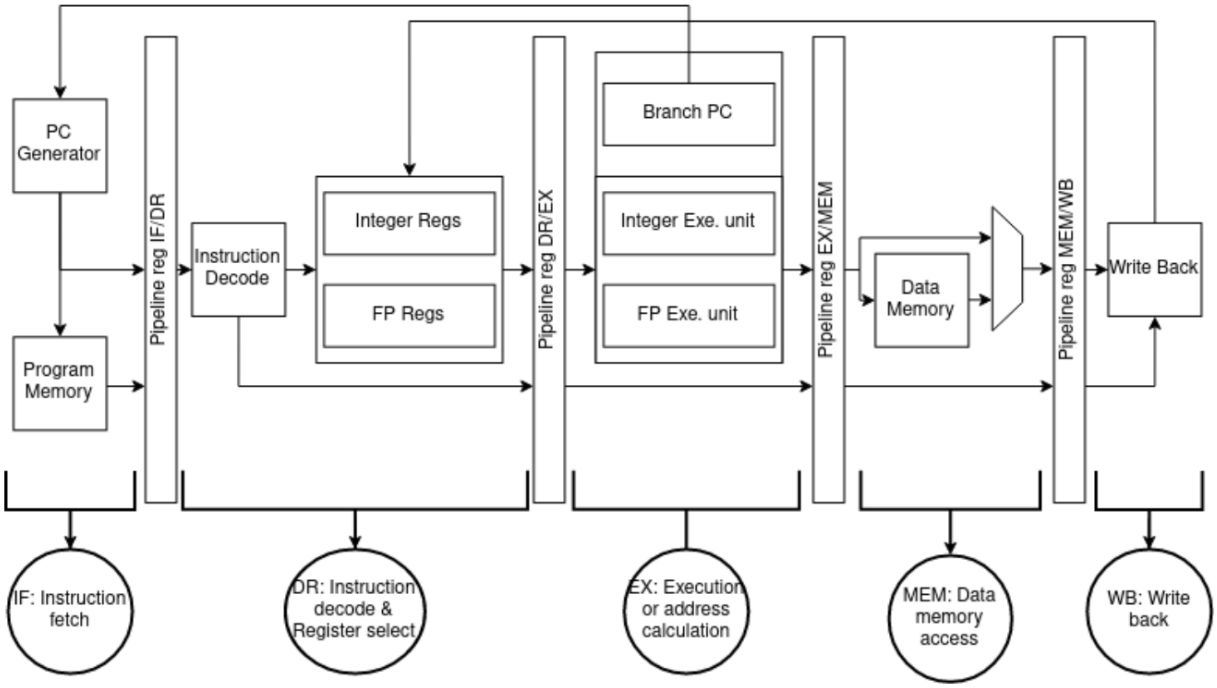


Figura 4.1: Diagrama de Bloques simplificado inicial. Fuente: [2]

En la imagen 4.1 se puede ver el diagrama simplificado de un pipeline, donde se realiza lo siguiente en cada etapa:

- IF: *Instruction fetch* Carga de la memoria de programa la instrucción a ejecutar, junto a esto también se entrega la dirección de memoria de programa PC (*program counter*).
- DR: *Instruction Decode & Register Select* Se separa la instrucción en distintas señales con la unidad *Instruction Decoder*, y se cargan y guardan los valores de registros correspondientes en la unidad de registros.
- EX: *Execution or Address calculation* Etapa de cálculos de dirección o de cálculos en la ALU.
- MEM: *Data memory access* Etapa de carga y guardado de datos en memoria.
- WB: *Write back* Etapa para, como dice su nombre, escribir de vuelta los valores obtenidos de las etapas EX y MEM en los registros de DR.

- Cabe destacar que al ser este un diagrama simplificado no muestra señales específicas como la unidad de control del pipeline, pero que si está presente en el diseño propuesto.

## 4.1. Load-Reserved y Store-Conditional

Junto a la restricción de mantener la estructura anterior, se definen condiciones que deben cumplir las instrucciones LR.W y SC.W, estas, basadas en [8] son:

1. La reserva de un espacio de memoria hecha por LR.W puede ser arbitrariamente grande, siempre que esta incluya todos los bytes del *word* en memoria a reservar.
2. La instrucción SC.W puede ser pareada únicamente con el último LR.W realizado según el orden del programa.
3. la instrucción SC.W no se puede parear si hay otra instrucción SC.W entre ella y la última instrucción LR.W en programa.

Con todo lo anterior, la solución propuesta es la inclusión de dos nuevos registros llamados *atomic_register_lr* y *atomic_addr_lr* dentro de la unidad de registros para almacenar el valor de memoria y su dirección respectivamente. También se incorporan señales que le informarán al sistema de control si la instrucción debe hacer una carga del espacio de memoria, haciendo una reserva en los nuevos registros. Y una señal para indicar si la reserva se mantiene para poder hacer un almacenamiento en la memoria. Cabe destacar que se ha decidido hacer solo dos registros y no una estructura más grande ya que la verificación se hace únicamente con la última reserva realizada. Resultado de lo anterior es el diagrama simplificado de la figura 4.2.

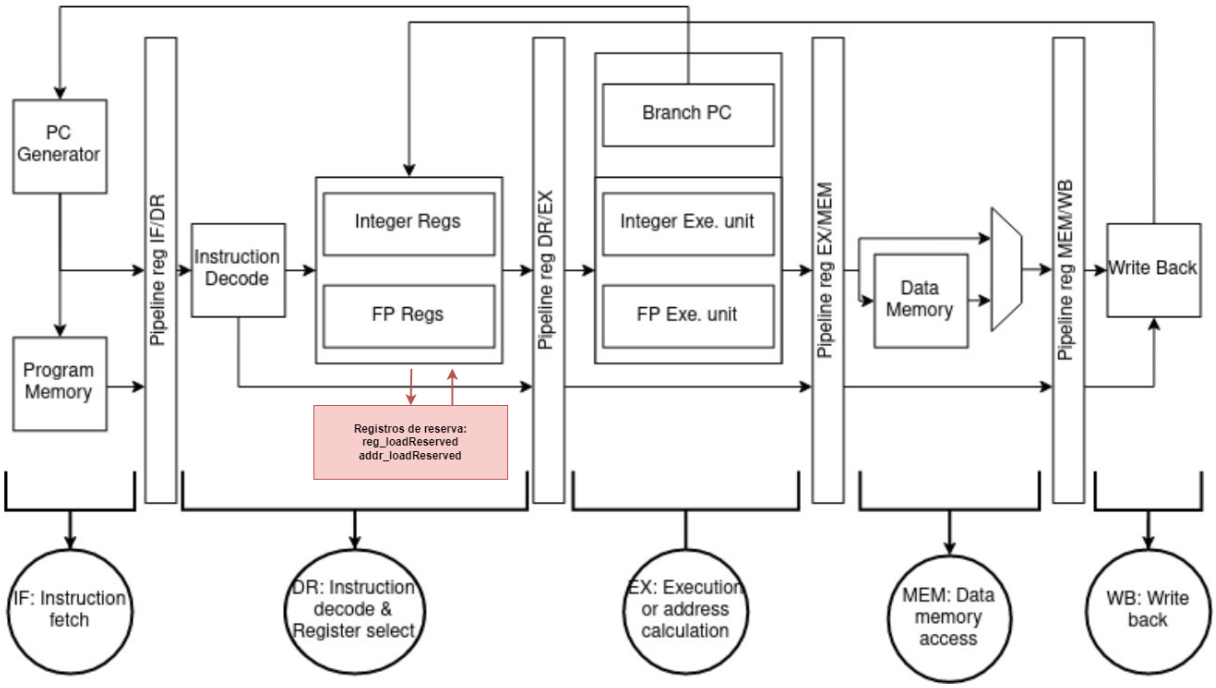


Figura 4.2: Diagrama de Bloques simplificado propuesto.

### 4.1.1. Load Reserved

Para el desarrollo y funcionamiento se han reutilizado partes del sistema ya desarrollado en [2], donde la instrucción LW ya permite guardar un *word* en un espacio de memoria indicado, pero no realiza reserva. Es por esto que al realizar un LR.W el sistema toma acción como si se tratase de un LW. Para junto a esto se incorpora un bit de control denominado *is_loadReserved*, el cual indica que se debe hacer una carga de un valor en memoria. Además, al recibir *is_loadReserved_WB* (el mismo bit en etapa WB), se le indica a la unidad de registros que se debe cargar el valor de memoria en el registro de destino x[rd], y hacer la reserva. Esto es, guardar el valor en el nuevo registro *atomic_register_lr*, y guardar la dirección de memoria x[rs1] en *atomic_addr_lr*. Lo anterior se puede ver explicado en el diagrama de la figura 4.3.

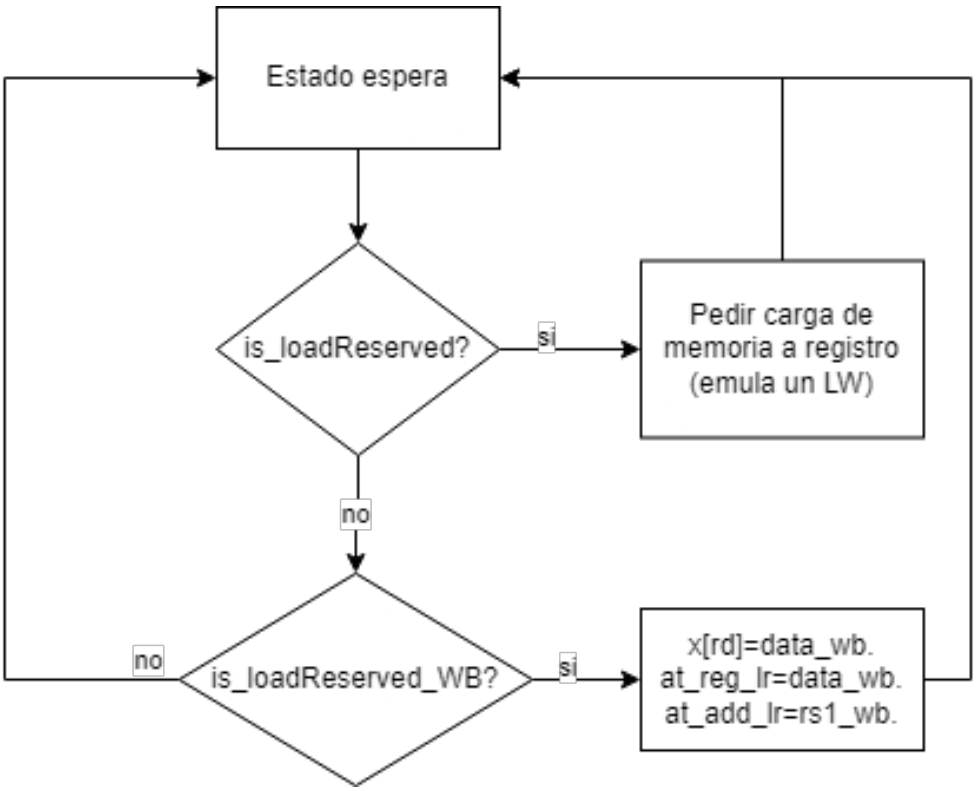


Figura 4.3: Diagrama de Flujo simplificado Load Reserved.

### 4.1.2. Store Conditional

Al igual que con la instrucción LR.W, la intención de SC.W es utilizar el sistema ya implementado en [2], pero, al tener que verificar la atomicidad del espacio de memoria, el proceso es más largo que el resto de instrucciones implementadas, esto es debido a que se debe cargar primeramente el valor para verificar que no ha sido modificado, y luego, si corresponde, se hace el proceso de guardado en memoria. Es decir, mientras que una instrucción normalmente pasa por las etapas (IF-DR-EX-MEM-WB), o (IF-DR-EX-MEM) en caso de una instrucción *store* (tipo S), la instrucción SC.W pasa por las etapas (IF-DR-EX-MEM-WB-(EX-MEM)), debiendo repetir EX y MEM si la reserva es verificada. Para esto se implementa, al igual que en *load reserved*, dos bits de control *is_storeConditional* para indicar que se debe hacer una carga de memoria, *is_storeConditional_WB* para indicar que se deben cargar los valores en los registros, y se agrega una nueva variable de control *store_conditions*, que indica si la reserva es verificada y por ende, se debe guardar en memoria.

Como la reserva de LR.W nos entrega la dirección de memoria y el valor de memoria reservada. SC.W parte cargando en el registro de destino x[rd] el valor de memoria de la dirección a guardar (M(x[rs1])), y comparando, si la dirección donde se hará el almacenamiento (x[rs1]) y el valor cargado en x[rd] de esa dirección es igual a los valores reservados en LR.W, *atomic_addr_lr* y *atomic_register_lr* respectivamente. Estas comparaciones, añadidas a que se reciba la señal *is_storeConditional_WB*, definen la variable *store_conditions*.

De cumplirse *store_conditions*, se cambia el valor del registro de destino x[rd] por 0 y se envía la acción de guardado del valor x[rs2] al espacio de memoria x[rs1] (M(x[rs1])) como indica la instrucción. En caso contrario guarda en x[rd] un valor distinto de 0 indicando un error (en el caso de este trabajo 32'b010101010...01), y no guarda el valor en memoria. También, se deben cambiar los valores de *atomic_addr_lr* y *atomic_register_lr* para que no sigan reservando el espacio de memoria, debido a que solo lo pueden reservar para el primer SC.W que chequee la condición (ambos se cambian al valor 32'b111...11). Lo anterior se puede ver explicado en el diagrama de la figura 4.4.

Cabe destacar, que al tener que hacer un guardado en memoria, y volver a pasar por las etapas EX y MEM, el *pipeline* debe activar un control hazard que anule las instrucciones que estaban ocurriendo luego de SC.W y retomar el PC (*program counter*) en la instrucción siguiente a la misma.

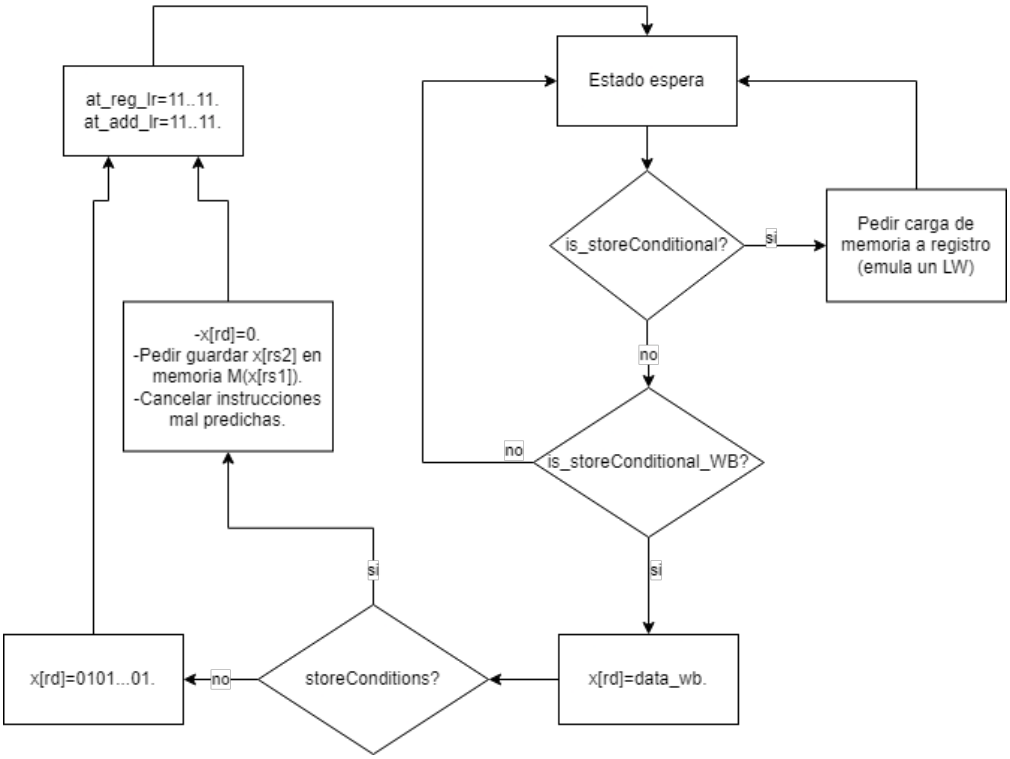


Figura 4.4: Diagrama de Flujo simplificado Store Conditional.

## 4.2. Instrucciones AMO

Para las instrucciones AMO el desarrollo es similar al de *store conditional*, donde se cargará inicialmente el el valor de memoria en el registro de destino x[rd], pero en vez de que este valor cambie dependiendo si el espacio de memoria está o no reservado, este se mantendrá con el valor cargado, debido a que no hay reserva previa, y se manda a guardar en el espacio de memoria x[rs1] (M(x[rs1])) el nuevo valor del registro de destino x[rd] operado con el valor del registro x[rs2] como indique la función AMO en ejecución. El resultado de este diseño se puede ver en el diagrama de la figura 4.5

Para estas instrucciones, si bien no hay que verificar reserva, al tener que cargar operar y guardar en una sola instrucción, también es necesario activar el mismo control hazard que activa *store conditional* para anular las instrucciones en el *pipeline* y recuperar el PC a la instrucción siguiente de la AMO.

Para la implementación de la operación se ha decidido hacer una pequeña ALU dentro del modulo de registros, en donde se modifica la salida rs2 del modulo por el valor necesario según la instrucción AMO en ejecución. Esto se hizo por complejidad, para evitar el cambio dentro de la ALU principal, así como cambiar el bus de datos debido a la necesidad de operar y guardar después de cargar.

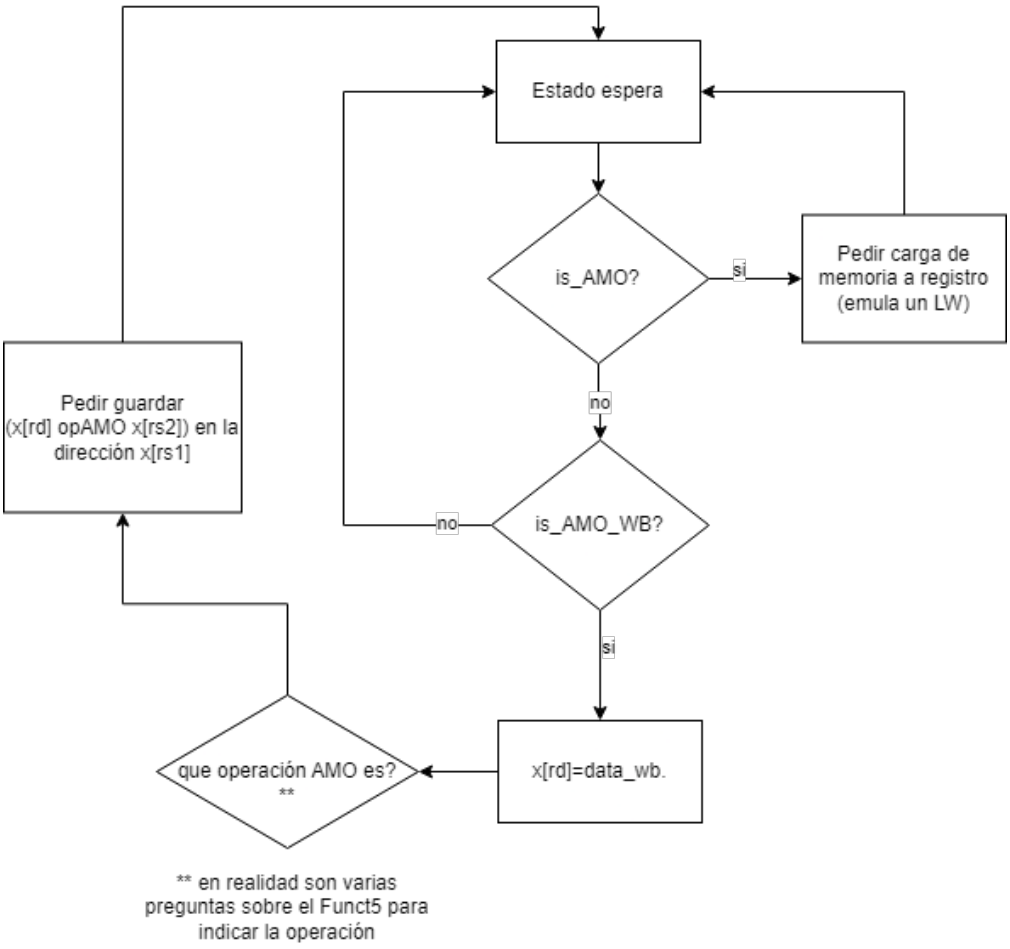


Figura 4.5: Diagrama de Flujo simplificado AMO.

# Capítulo 5

# Implementación

Para la implementación se partió con el SoC desarrollado en [2], al cual se le modificaron secciones ya implementadas, así como también se le incorporaron nuevas secciones para el manejo de las nuevas instrucciones del sistema.

## 5.1. Diseño

### 5.1.1. Módulo de Instrucciones Atómicas

Para reconocer una instrucción atómica, se ha incorporado un módulo (figura 5.1) en la etapa *DR: Instruction decode & Register select* del *pipeline*. Es un modulo combinacional en el que se toma de entradas: el $format\_type$, el $sub\_format\_type$ y el *Funct5*. Estas son tres variables obtenidas del *Instruction Decoder* que sirven para reconocer la instrucción que se está ejecutando. Tiene como salidas: *is_storeConditional*, *is_loadReserved*, *is_Atomic* y *is_AMO* que indican lo que dicen sus nombres. Si la instrucción que está en la etapa es atómica, es una instrucción de carga con reserva, un guardado condicional o una instrucción AMO.

Las variables de salida de este módulo se utilizan junto al *Funct5* en el modulo de registros para saber si reservar, cargar y guardar u operar atómicamente, así como también en el control del pipeline para indicar si la memoria debe cargar, o guardar un valor, lo que implica, por lo visto en 4, que también se utilizan como variables del *Control Hazard* que necesitan las instrucciones atómicas.

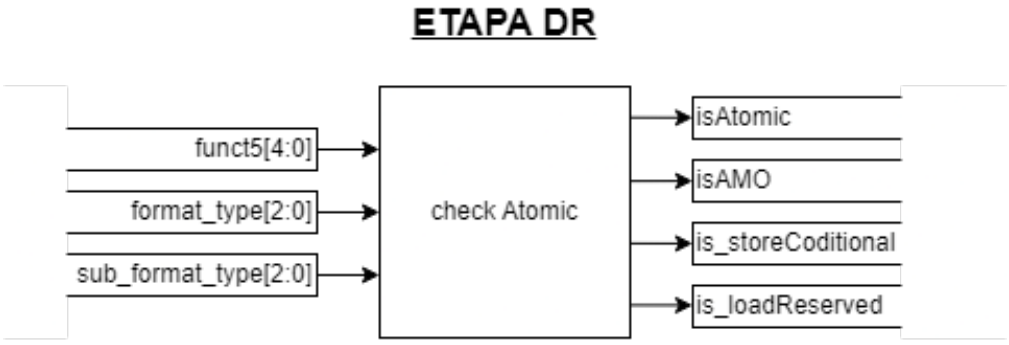


Figura 5.1: Diagrama de bloques del modulo *checkAtomic*.

### 5.1.2. Compilador de Assembler a Binario

Para poder ejecutar las instrucciones atómicas dentro del SoC, se ha desarrollado e implementado de un nuevo decodificador de *assembler* a lenguaje de máquina, ya que el anteriormente usado en [2], que está dentro de RARS, no es capaz de compilar las instrucciones atómicas.

Este nuevo decodificador es desarrollado en *Python*, y realiza las siguientes tareas:

1. Simplificar el código de *assembler*, eliminando comentarios y espacios en blanco innecesarios para la conversión.
2. Reemplazar los nombres de saltos de linea por sus valores en memoria.
3. Convertir el código *assembler* a binario.
4. Convertir al formato de lectura del FPGA (modificación del código *python* de [2] que realiza esta acción).

Este compilador tiene la limitación de no permitir las pseudoinstrucciones de RISC-V, y de permitir estructuras más básicas para las instrucciones, es decir, no permite varias formas de escribir una misma instrucción en *assembler*. El código de este compilador se encuentra en el anexo A.

## 5.2. Rediseño

Las modificaciones hechas al sistema anterior [2] son: mejorar la interfaz de lectura de registros del SoC en el FPGA (interfaz de usuario), incorporar las funciones atómicas a la lectura del *Instruction Decoder*, cambiar la estructura del modulo de registros y cambiar el control del *pipeline* y estructura del mismo para incorporar las nuevas partes al sistema que permiten ejecutar las nuevas instrucciones.

### 5.2.1. Interfaz de Usuario

En el anterior desarrollo [2] la lectura de cada par de *bytes* del registro a leer o ingresar se hacia mediante hexadecimal, por lo que un valor de 0 a 15 pasaba a leerse de 0 a F (llegando a 9 y luego pasando a las letras A a F), pero en este todos los valores del abecedario estaban en mayúsculas, por lo que en valores como B (=11) o D (=13) al leerlos en el *display* de 7 segmentos del FPGA se veían como un 8 o un 0, respectivamente. La única diferencia es un punto que tenían los valores mayores a 9 para distinguir que eran letras.

Para obtener una mejor lectura, en la que sea más fácil distinguir una B de un 8 y una D de un 0, se han modificado las interpretaciones de dichos valores en el *display* de 7 segmentos, pasando de una B mayúscula a una b minúscula, y de una D mayúscula a una d minúscula. Igualmente, se decidió mantener el punto del *display* para distinguir los valores mayores a 9, pero con esta nueva lectura que evita confusiones al momento de revisar el *display*.

Ejemplos de este cambio son las figuras 5.2 y 5.3

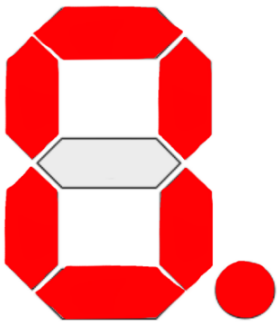

Figura 5.2: Cómo se desplegaba una D en el *display* de 7 segmentos.

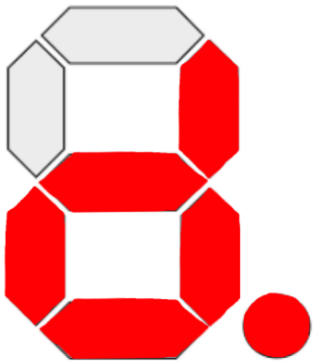

Figura 5.3: Cómo se despliega ahora una D en el *display* de 7 segmentos.

### 5.2.2. Instruction Decoder

En el *Instruction Decoder* desarrollado en [2] clasifica las distintas instrucciones en diferentes grupos y subgrupos que dependen del tipo de instrucción del ISA. Las instrucciones atómicas, al ser instrucciones de tipo R deben pertenecer al grupo de este tipo, pero deben diferenciarse entre las demás (las instrucciones de tipo R que corresponden al ISA base (I), o a la extensión flotante simple (F)), por lo que se debe definir su propio sub grupo. Para esta nomenclatura que define el *Instruction Decoder*, se ha utilizando el *opcode* de las funciones atómicas, que, al ser siempre el mismo (tabla 2.1), nos permite definir el $format\_type$ (R) y el $sub\_format\_type$ (A).

Además, se ha incorporado una nueva variable de salida en este módulo, el *Funct5*. Esta nueva lectura corresponde a los bits del 31 al 26 de cada instrucción y son estos bits los que distinguen las distintas instrucciones atómicas entre si.

Las variables aquí incorporadas son las utilizadas en el módulo checkAtomic (figura 5.1) explicado anteriormente.

### 5.2.3. Módulo de Registros

Este módulo fue modificado para ser capaz de reservar, cargar y guardar espacios de memoria de manera atómica como se explicó en el capitulo 4. Para esto se incorporan entradas y registros extras que activarán la carga de datos en registros, o el almacenamiento de valores en memoria. Las nuevas entradas son *is_storeConditional_WB*, *is_loadReserved_WB*, *is_Atomic_WB*, *is_AMO_WB* y *Funct5_WB* que pertenecen a la etapa WB del pipeline. Los registros extras son *reg_loadReserved* y *addr_loadReserved*. Junto a lo anterior, también se agrega un *output* nuevo para levantar un *flag* si se cumplen o no las condiciones de guardado en memoria *store_conditions*.

Las instrucciones Atómicas parten siempre con la carga del valor de memoria en el registro de destino, por lo que si en WB se detecta que es una función atómica (*is_Atomic_WB*), se realizará una carga el registro de destino. Luego, si se recibe una entrada *is_loadReserved_WB*, el sistema a demás de cargar el valor de memoria en el registro de destino, lo hará también en el nuevo registro *reg_loadReserved*. También guardará el valor de x[rs1] en *addr_loadReserved* (en específico se guarda el valor WB de x[rs1]).

Si es *is_storeConditional_WB* se cargará en el registro de destino x[rd] el valor de memoria, pero inmediatamente se comparará el valor cargado, así como su dirección de memoria (WB de x[rs1]), con los valores de los registros de reserva. Si son iguales ambos valores, en el registro de destino se guarda un 0 y se levanta el *flag store_conditions* para indicar que se debe guardar el valor x[rs2] en el espacio de memoria x[rs1] (nuevamente son los valores obtenidos en WB).

De manera similar al de *is_storeConditional_WB*, si se recibe un *is_AMO_WB*, el registro de destino cargará el valor de memoria. Luego, sin modificar el registro de destino y se levanta el *flag* de salida *store_conditions* para indicar que se debe guardar en memoria. Pero en vez de guardarse el valor x[rs2], se guarda el valor x[rs2] operado con el valor x[rd]. La operación entre estos dos valores se define con *Funct5_WB* y se almacena en la memoria con dirección x[rs1] (M(x[rs1])).

Para mayor entendimiento ver el anexo C, el módulo de registros *registers_files*.

## 5.3. Control

Se incorpora en el control del pipeline condiciones nuevas para saber cuando guardar en memoria, o cargar en los registros, así como también control para cancelar instrucciones y volver en el *Program Counter* (PC) a esas instrucciones canceladas. Esto se hace debido a la estructura que tienen las instrucciones atómicas en el pipeline, que deben hacer a la carga y almacenamiento en una sola instrucción (exceptuando LR.W).

En específico, se han modificado señales como:

1. *write_reg* del modulo *registers_files*. Es una señal de entrada que indica cuando se debe hacer, o no una carga de valores en el registro. Inicialmente se guardaban valores en los registros para todas las instrucciones con excepción de las instrucciones S y B, que corresponden a guardado en memoria y saltos en el PC, así como en las interrupciones. Como *store_conditions* es una señal que indica que la instrucción atómica puede guardar su valor en memoria, tampoco debe almacenar el valor en registro cuando llega a la etapa WB, por lo que se agrega como nueva condición la señal *store_conditions_WB*.
2. *rw_data_mem* y *access_to_mem* del módulo de memoria de datos. Estas variables indican si el valor de entrada al modulo de memoria es de escritura o no, y si el acceso al uso de la memoria es valido o no. Es por esto, que a *rw_data_mem* se le incorpora *store_conditions* para indicar que el valor si se debe guardar en memoria, y a *access_to_mem* se le agrega *is_Atomic* para indicar que debe ejecutarse una acción en memoria.
3. Las diferentes interrupciones del SoC se calculaban en la etapa DR, misma en donde se obtiene *store_conditions*, por lo que, se ha incorporado a la definición de las interrupciones esta variable, ya que al volver a la instrucción atómica, la interrupción queda invalida.

Junto a lo anterior, se incorporan los registros *pipeline* que permiten pasar las señales diseñadas en este trabajo por el *datapath*.

Por último, la modificación que permite que todas las instrucciones atómicas puedan pasar 2 veces por el pipeline sin generar quiebres en la ejecución del código, es la siguiente (figura 5.4):

Donde se puede ver que, todas las etapas del pipeline se anulan (los registros se hacen 0) exceptuando EX, la cual almacena la instrucción de guardar el valor atómico en memoria. También se puede ver el calculo de *PC_IF=PC_WB+4*, lo que vuelve el *program counter* a la instrucción que debe ejecutar.

Todas estas modificaciones se pueden ver en el anexo B.

```
else if(set_regs) begin // set
    if (store_conditions) begin
        PC_IF = PC_WB+32'b100;

        {PC_DR, inst_DR} = 0;

        {PC_EX, imm_EX, rs1_EX, rs2_EX, rs3_EX, rd_add_EX, rs1_add_EX, rs2_add_EX, rs3_add_EX,
         funct7_out_EX, funct3_EX, format_type_EX, sub_format_type_EX, reg_access_option_EX,
         is_imm_valid_EX, acces_to_fpu_EX, acces_to_mem_EX, not_valid_EX, is_ebreak_EX, is_ecall_EX,
         is_loadReserved_EX, is_storeConditional_EX, store_conditions_EX    , is_Atomic_EX  , is_AMO_EX ,funct5_EX}
        =
        {PC_WB, 32'b0 , rs1_DR, rs2_DR, 32'b0 , 5'b0, 5'b0, 5'b0, 5'b0,
        4'b1111, 3'b010, 3'b111, 3'b111, 2'b00,
        1'b0, 1'b0, acces_to_mem_DR, 1'b0, 1'b0, 1'b0,
        1'b0, 1'b0, store_conditions, 1'b0, 1'b0,5'b11111};

        // PipelineReg_EX2MEM
        {PC_MEM, final_res_MEM, operand3_MEM, rd_add_MEM, format_type_MEM, sub_format_type_MEM, funct3_MEM,
         reg_access_option_MEM, acces_to_mem_MEM, not_valid_MEM, is_ebreak_MEM, is_ecall_MEM,
         is_loadReserved_MEM, is_storeConditional_MEM, store_conditions_MEM    , is_Atomic_MEM  , is_AMO_MEM     ,funct5_ME

        {PC_WB,  rd_WB,  rd_add_WB,  format_type_WB,  reg_access_option_WB,  not_valid_WB,  is_ebreak_WB,  is_ecall_WB ,
         is_loadReserved_WB, is_storeConditional_WB, store_conditions_WB    , is_Atomic_WB  , is_AMO_WB     ,funct5_WB,
         rs1_WB, rs2_WB} = 0;

    end
```

Figura 5.4: Código del *Control Hazard* y nuevo cálculo de PC.

# Capítulo 6

# Verificación y Pruebas

Para la verificación del sistema se utilizaron principalmente cuatro códigos que se encuentran en el proyecto:

1. *tb_riscv32imf_top.sv*. Código *test bench* desarrollado por Gianluca que permite simular los códigos en el *datapath pipeline*, y luego recuperar los valores de registros y memorias.
2. *pruebaLW_SC.txt*. Código *assembler* con una prueba sencilla de reserva y guardado de datos en memoria con las instrucciones LR.W y SC.W.
3. *pruebaLW_SC_2.txt*. Código *assembler* con una prueba sencilla de reserva fallida y, por ende, no guardado de datos en memoria con las instrucciones LR.W y SC.W.
4. *pruebaAMO.txt*. Código *assembler* con una prueba sencilla de una instrucción atómica tipo AMO. El código posee todas las instrucciones comentadas excepto una, para hacer la prueba de todas.

Entre los principales resultados obtenidos están los registros y memorias de una reserva fallida por parte del código *pruebaLW_SC_2.txt* como se ve en las figuras 6.1 y 6.2

El código del fallo define el registro 0b como 1, el 0c como 4, y el 0d como 5, luego, guarda el 1 en memoria. Finalmente, reserva el espacio y carga el valor en 0a, para terminar el código verificando la reserva pero en una dirección errónea, lo que hace fallar el guardado condicional. Es por esto que el valor en memoria no cambia, y en el registro 0a se carga un 55555555, indicando una falla en la reserva.

Otro resultado obtenido en *pruebaLW_SC.txt* se puede ver en las figuras 6.3 y6.4

El código del define el registro 0b como 1 y el 0c como 4, luego, guarda el 1 en memoria. Finalmente reserva el espacio y carga el valor en 0a, para terminar el código verificando la reserva correctamente, lo que permite el guardado condicional. Es por esto que el valor en memoria cambia a un 4, y en el registro 0a se carga un 0, indicando una reserva y guardado exitoso.

Ambos códigos se encuentran en los anexos E y F, donde se puede ver más al detalle el funcionamiento.

Otro punto que verifica el correcto funcionamiento del sistema es el valor almacenado en el registro 11, que corresponde a una a (=10). Esto demuestra que, siendo una instrucción que ocurre luego de las cargas y guardados, se ejecuta correctamente en el sistema.

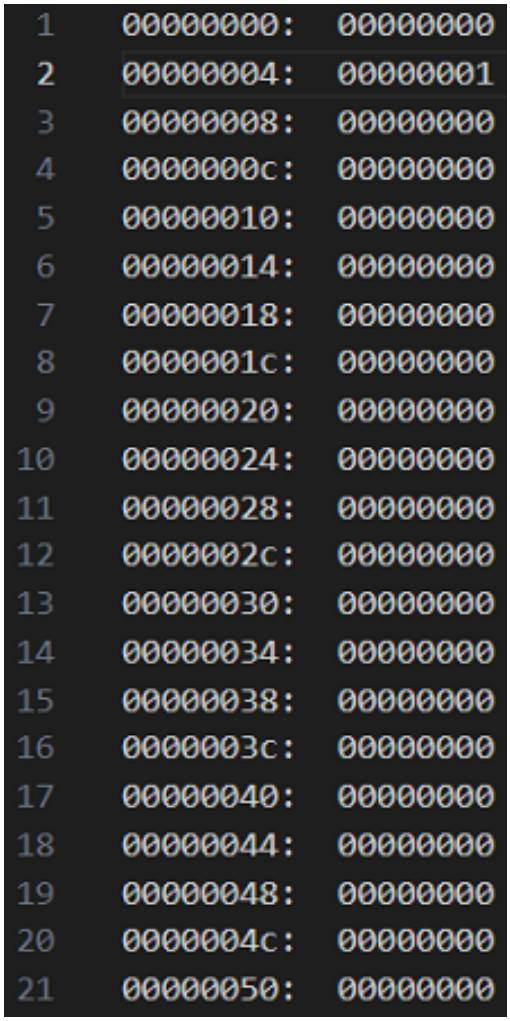


Figura 6.1: Imagen de la memoria obtenida en la Simulación.

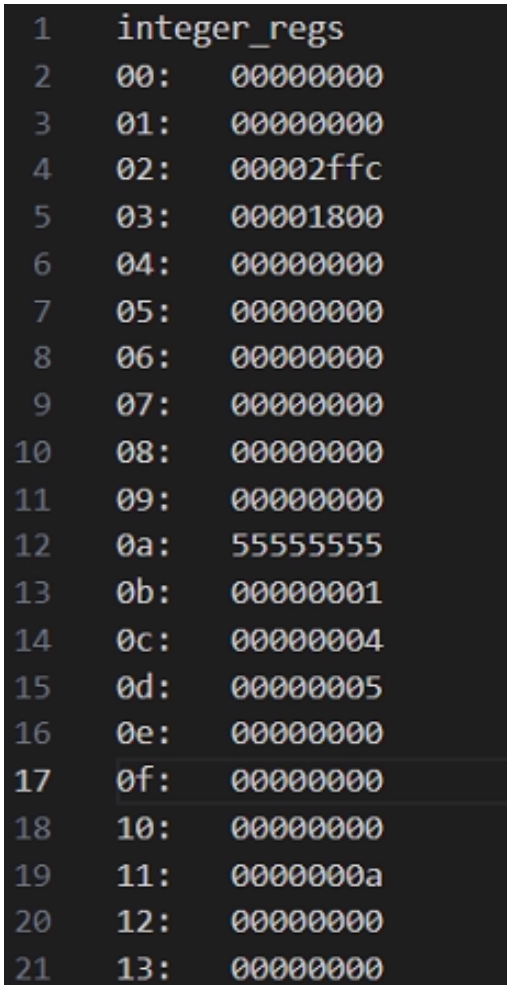


Figura 6.2: Imagen de los registros obtenida en la Simulación.

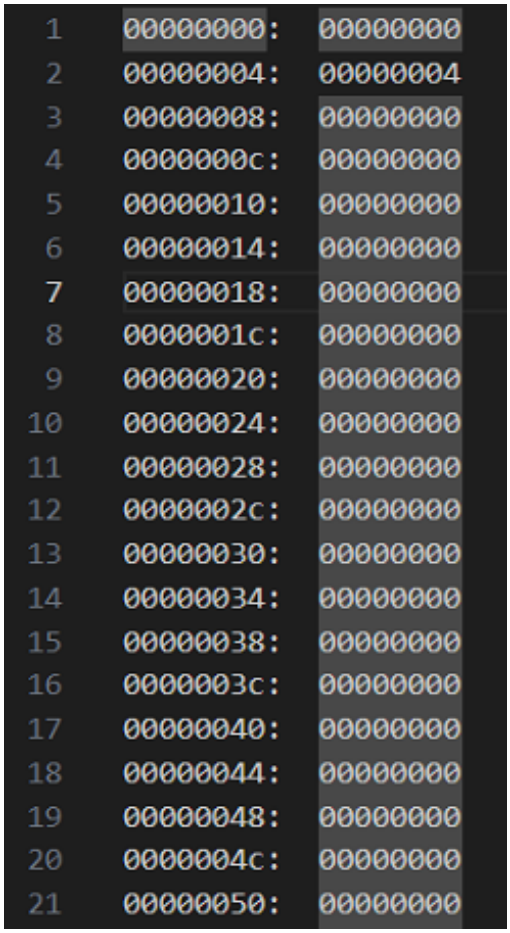


Figura 6.3: Imagen de la memoria obtenida en la Simulación.

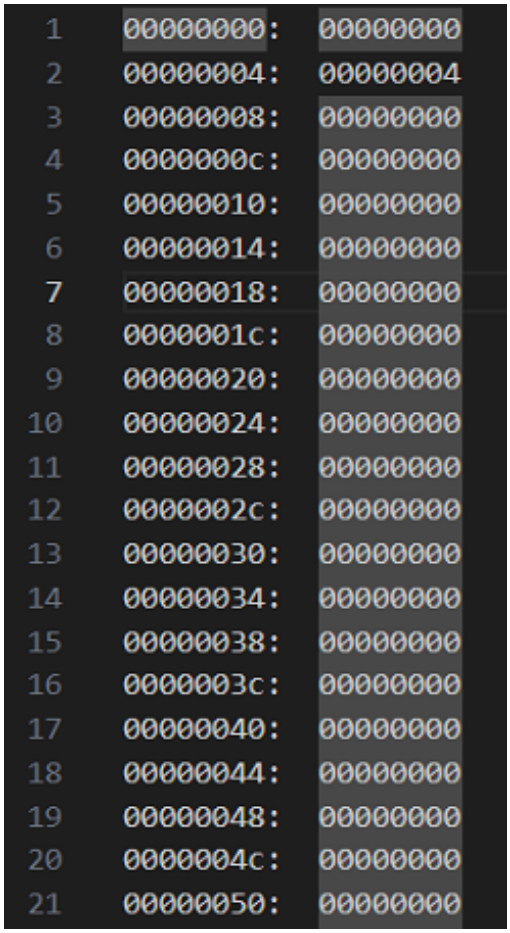


Figura 6.4: Imagen de los registros obtenida en la Simulación.

# Capítulo 7

# Conclusiones

Luego de realizado el trabajo, se pueden concluir diversos puntos. Primeramente, es interesante como planteamiento pedagógico para el estudiante de ingeniería, el adentrarse en un trabajo ya realizado para crear capas de desarrollo sobre él. Una actividad no muy cotidiana en el desarrollo como estudiante, pero que puede ser una experiencia muy cotidiana en el mundo laboral, por lo que, vivirlo de primera mano como un proyecto de memoria te prepara a afrontar este tipo de desafíos a futuro.

Junto a lo anterior, tanto la concepción, comprensión y desarrollo del SoC, así como su uso para la implementación de acciones con *assembler* RISC-V en él, ayudan bastante en el fortalecimiento de conocimientos aprendidos en la Universidad. Estos conocimientos son herramientas de gran interés para la industria de la electrónica, tanto por el uso del lenguaje de descripción de hardware como RISC-V, que se ve reflejado en el interés de cientos de empresas de gran renombre en el rubro sobre el desarrollo de la tecnología RISC-V. Generando con este sistema, una muy buena herramienta para los estudiantes que se quieran adentrar en el mundo.

El sistema quizá es mejorable en cuanto al uso del bus de información, debido a la complejidad del funcionamiento y a la poca información encontrada, se tomaron decisiones que aminoraban la carga del trabajo, pero aumentaban el movimiento de datos dentro del *Datapath*, lo que significa una mayor energía utilizada. En cuanto al rendimiento, no se ve fuertemente afectado si se considera el guardado condicional o la instrucción AMO como una instrucción no muy recurrente, ya que el *pipeline* funciona correctamente, pero tiene un *Control Hazard* grande al momento de realizar el almacenamiento en memoria.

En cuanto a los resultados obtenidos, se puede afirmar un correcto funcionamiento de las instrucciones atómicas en el SoC, lo que permite crecimientos a futuro en el sistema como el desarrollo de un procesador *multicore*, donde estas mismas instrucciones son la piedra angular para la sincronización de los múltiples *cores* que podría tener.

De la misma forma, se puede seguir ampliando el trabajo a otras instrucciones de RISC-V, aprovechando su segmentación y sus diversas extensiones que aún no se han implementado, como las instrucciones comprimidas (C) o flotantes de presición doble (D).

# Bibliografía


[1] Andrew Waterman D. A. Patterson. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon LLC, Berkeley, California, Estados Unidos, 2017.

[2] G. D'Agostino Matute. *Diseño e implementación de un SoC en un FPGA basado en el ISA de RISC-V*. Universidad de Chile - Facultad de Ciencias Físicas y Matemáticas, Santiago, Chile, 2021. https://repositorio.uchile.cl/handle/2250/183965.

[3] Departamento de Ingeniería Eléctrica. *"Sistemas Digitales" notas de clases para EL4002-1*. Universidad de Chile, Santiago, Chile, Otoño, 2020.

[4] Departamento de Ingeniería Eléctrica. *"Seminario de Sistemas Digitales" notas de clases para EL7039-1*. Universidad de Chile, Santiago, Chile, Otoño, 2021.

[5] Departamento de Ingeniería Eléctrica. *"Arquitectura de Computadores" notas de clases para EL4102-1*. Universidad de Chile, Santiago, Chile, Primavera, 2020.

[6] Digilent. *Nexys A7 Reference Manual*. RISC-V International, 1300 Henley Court, July 10, 2019. https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual.

[7] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, The Hardware Software/Interface: RISC-V Edition*. Morgan Kaufmann, Cambridge, Estados Unidos, 2018.

[8] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-*. RISC-V International, 2019. https://five-embeddev.com/riscv-isa-manual/latest/riscv-spec.html.

# ANEXOS

## Anexo A

El siguiente código corresponde al compilador desarrollado en *python* para la lectura del ISA RISC-V IMAF.

```
import sys, re
import os

registers = {
  "zero": "00000",
  "ra": "00001",
  "sp": "00010",
  "gp": "00011",
  "tp": "00100",
  "t0": "00101",
  "t1": "00110",
  "t2": "00111",
  "s0": "01000",
  "s1": "01001",
  "a0": "01010",
  "a1": "01011",
  "a2": "01100",
  "a3": "01101",
  "a4": "01110",
  "a5": "01111",
  "a6": "10000",
  "a7": "10001",
  "s2": "10010",
  "s3": "10011",
  "s4": "10100",
  "s5": "10101",
  "s6": "10110",
  "s7": "10111",
  "s8": "11000",
  "s9": "11001",
  "s10": "11010",
  "s11": "11011",
  "t3": "11100",
  "t4": "11101",
  "t5": "11110",
  "t6": "11111"
}

registers_float = {
    "ft0": "00000",
  "ft1": "00001",
  "ft2": "00010",
  "ft3": "00011",
  "ft4": "00100",
  "ft5": "00101",
```

```
  "ft6": "00110",
  "ft7": "00111",
  "fs0": "01000",
  "fs1": "01001",
  "fa0": "01010",
  "fa1": "01011",
  "fa2": "01100",
  "fa3": "01101",
  "fa4": "01110",
  "fa5": "01111",
  "fa6": "10000",
  "fa7": "10001",
  "fs2": "10010",
  "fs3": "10011",
  "fs4": "10100",
  "fs5": "10101",
  "fs6": "10110",
  "fs7": "10111",
  "fs8": "11000",
  "fs9": "11001",
  "fs10": "11010",
  "fs11": "11011",
  "ft8": "11100",
  "ft9": "11101",
  "ft10": "11110",
  "ft11": "11111"
}

def is_binary(num_str):
  return len(num_str) >= 2 and num_str[0:2] == "0b"

def is_hex(num_str):
  return len(num_str) >= 2 and num_str[0:2] == "0x"

def is_int(num_str):
  for chr in num_str:
    if not(chr.isdigit() or chr == '-'):
      return False

  return True

def num_str_to_bin(num_str, base, pad, signed):
  bytes_num = int(num_str, base).to_bytes(4, byteorder="big", signed=
   signed)
  bin_num = "".join(format(x, '08b') for x in bytes_num)

  return bin_num[-pad:]

def val_to_bin(num_str, pad, signed, ref_dict=None):
  if is_binary(num_str):
    return num_str_to_bin(num_str[2:], 2, pad, signed)
  elif is_hex(num_str):
    return num_str_to_bin(num_str[2:], 16, pad, signed)
  elif is_int(num_str):
    return num_str_to_bin(num_str, 10, pad, signed)
  elif ref_dict and num_str in ref_dict:
```

```
    return ref_dict[num_str]

  raise ValueError(f"{num_str} can not be converted to binary")

def reg_to_bin(num_str):
  if num_str[0] == 'x':
    return num_str_to_bin(num_str[1:], 10, 5, False)
  else:
    return val_to_bin(num_str, 5, False, ref_dict=registers)

def reg_float_to_bin(num_str):
  if num_str[0] == 'f' and num_str[1]!='a' and num_str[1]!='s' and num_str
    [1]!='t':
    return num_str_to_bin(num_str[1:], 10, 5, False)
  else:
    return val_to_bin(num_str, 5, False, ref_dict=registers_float)


def parse_reg_imm(cmd, data):
  groups = re.match(r"([^,]+),(.+)", cmd).groups()
  data.rd = groups[0]
  data.imm = groups[1]

def parse_reg_off_reg(cmd, data):
  groups = re.match(r"([^,]+),([^(]+)\(([^)]+)\)", cmd).groups()
  data.rd = groups[0]
  data.imm = groups[1]
  data.rs1 = groups[2]

def parse_reg_reg_imm(cmd, data):
  groups = re.match(r"([^,]+),([^,]+),(.+)", cmd).groups()
  data.rd = groups[0]
  data.rs1 = groups[1]
  data.imm = groups[2]

def parse_ecall(cmd, data):
  #groups = re.match(r"([^,]+),([^,]+),(.+)", cmd).groups()
  data.rd = '00000'
  data.rs1 = '00000'
  data.imm = '000000000000'

def parse_ebreak(cmd, data):
  #groups = re.match(r"([^,]+),([^,]+),(.+)", cmd).groups()
  data.rd = '00000'
  data.rs1 = '00000'
  data.imm = '000000000001'

def parse_reg_reg_reg(cmd, data):
  groups = re.match(r"([^,]+),([^,]+),(.+)", cmd).groups()
  data.rd = groups[0]
  data.rs1 = groups[1]
  data.rs2 = groups[2]

def parse_reg_reg_reg_reg(cmd, data):
  groups = re.match(r"([^,]+),([^,]+),([^,]+),(.+)", cmd).groups()
  data.rd = groups[0]
```

```
156   data.rs1 = groups[1]
157   data.rs2 = groups[2]
158   data.rs3 = groups[3]
159 
160 def parse_reg_reg(cmd, data):
161   groups = re.match(r"([^,]+),([^,]+)", cmd).groups()
162   data.rd = groups[0]
163   data.rs1 = groups[1]
164   data.rs2 = '00000'
165 
166 
167 def parse_reg_reg_1(cmd, data):
168   groups = re.match(r"([^,]+),([^,]+)", cmd).groups()
169   data.rd = groups[0]
170   data.rs1 = groups[1]
171   data.rs2 = '00001'
172 
173 
174 
175 
176 
177 def ex_rtype(data):
178   rd_bin = reg_to_bin(data.rd)
179   rs1_bin = reg_to_bin(data.rs1)
180   rs2_bin = reg_to_bin(data.rs2)
181 
182   return data.funct7 + rs2_bin + rs1_bin + data.funct3 + rd_bin + data.
      opcode
183 
184 def ex_r4type(data):
185   rd_bin = reg_float_to_bin(data.rd)
186   rs1_bin = reg_float_to_bin(data.rs1)
187   rs2_bin = reg_float_to_bin(data.rs2)
188   rs3_bin = reg_float_to_bin(data.rs3)
189 
190   return rs3_bin + '00' + rs2_bin + rs1_bin + data.funct3 + rd_bin + data.
      opcode
191 
192 def ex_rtype_float(data):
193   rd_bin = reg_float_to_bin(data.rd)
194   rs1_bin = reg_float_to_bin(data.rs1)
195   rs2_bin = reg_float_to_bin(data.rs2)
196 
197   return data.funct7 + rs2_bin + rs1_bin + data.funct3 + rd_bin + data.
      opcode
198 
199 def ex_rtype_float_int(data):
200   rd_bin = reg_float_to_bin(data.rd)
201   rs1_bin = reg_to_bin(data.rs1)
202   rs2_bin = reg_float_to_bin(data.rs2)
203 
204   return data.funct7 + rs2_bin + rs1_bin + data.funct3 + rd_bin + data.
      opcode
205 
206 def ex_rtype_int_float(data):
207   rd_bin = reg_to_bin(data.rd)
```

```
208   rs1_bin = reg_float_to_bin(data.rs1)
209   rs2_bin = reg_float_to_bin(data.rs2)
210
211   return data.funct7 + rs2_bin + rs1_bin + data.funct3 + rd_bin + data.
       opcode
212
213 def ex_itype(data):
214   rd_bin = reg_to_bin(data.rd)
215   rs1_bin = reg_to_bin(data.rs1)
216   imm_bin = val_to_bin(data.imm, 12, True)
217
218   return imm_bin + rs1_bin + data.funct3 + rd_bin + data.opcode
219
220 def ex_itype_float(data):
221   rd_bin = reg_float_to_bin(data.rd)
222   rs1_bin = reg_to_bin(data.rs1)
223   imm_bin = val_to_bin(data.imm, 12, True)
224
225   return imm_bin + rs1_bin + data.funct3 + rd_bin + data.opcode
226
227 def ex_sitype(data):
228   rd_bin = reg_to_bin(data.rd)
229   rs1_bin = reg_to_bin(data.rs1)
230   imm_bin = val_to_bin(data.imm, 5, False)
231
232   return data.funct7 + imm_bin + rs1_bin + data.funct3 + rd_bin + data.
       opcode
233
234 def ex_stype(data):
235   rs1_bin = reg_to_bin(data.rs1)
236   rs2_bin = reg_to_bin(data.rd)
237   imm_bin = val_to_bin(data.imm, 12, True)
238
239   return imm_bin[-12:-5] + rs2_bin + rs1_bin + data.funct3 + imm_bin[-5:]
       + data.opcode
240
241 def ex_stype_float(data):
242   rs1_bin = reg_to_bin(data.rs1)
243   rs2_bin = reg_float_to_bin(data.rd)
244   imm_bin = val_to_bin(data.imm, 12, True)
245
246   return imm_bin[-12:-5] + rs2_bin + rs1_bin + data.funct3 + imm_bin[-5:]
       + data.opcode
247
248 def ex_btype(data):
249   rd_bin = reg_to_bin(data.rd)
250   rs1_bin = reg_to_bin(data.rd)
251   rs2_bin = reg_to_bin(data.rs1)
252   imm_bin = val_to_bin(data.imm, 13, True)
253
254   return imm_bin[-13] + imm_bin[-11:-5] + rs2_bin + rs1_bin + data.funct3
       + imm_bin[-5:-1] + imm_bin[-12] + data.opcode
255
256 def ex_utype(data):
257   rd_bin = reg_to_bin(data.rd)
258   imm_bin = val_to_bin(data.imm, 20, True)
```

```
259
260   return imm_bin + rd_bin + data.opcode
261
262 def ex_jtype(data):
263   rd_bin = reg_to_bin(data.rd)
264   imm_bin = val_to_bin(data.imm, 21, True)
265
266   return imm_bin[-21] + imm_bin[-11:-1] + imm_bin[-12] + imm_bin[-20:-12]
       + rd_bin + data.opcode
267
268
269 class CommandData:
270   def __init__(self, opcode, funct3=None, funct7=None):
271     self.opcode = opcode
272     self.funct3 = funct3
273     self.funct7 = funct7
274     self.rd = None
275     self.rs1 = None
276     self.rs2 = None
277     self.rs3 = None
278     self.imm = None
279
280
281 class CommandHandler:
282   def __init__(self, parser, executor, data):
283     self.parser = parser
284     self.executor = executor
285     self.data = data
286
287   def parse(self, command):
288     self.parser(command, self.data)
289
290   def execute(self):
291     return self.executor(self.data)
292
293 handlers = {
294   "lui":    CommandHandler(parse_reg_imm, ex_utype, CommandData("0110111")
       ),
295   "auipc":  CommandHandler(parse_reg_imm, ex_utype, CommandData("0010111")
       ),
296   "jal":    CommandHandler(parse_reg_imm, ex_jtype, CommandData("1101111")
       ),
297   "jalr":   CommandHandler(parse_reg_off_reg, ex_itype, CommandData("
       1100111", "000")),
298   "beq":    CommandHandler(parse_reg_reg_imm, ex_btype, CommandData("
       1100011", "000")),
299   "bne":    CommandHandler(parse_reg_reg_imm, ex_btype, CommandData("
       1100011", "001")),
300   "blt":    CommandHandler(parse_reg_reg_imm, ex_btype, CommandData("
       1100011", "100")),
301   "bge":    CommandHandler(parse_reg_reg_imm, ex_btype, CommandData("
       1100011", "101")),
302   "bltu":   CommandHandler(parse_reg_reg_imm, ex_btype, CommandData("
       1100011", "110")),
303   "bgeu":   CommandHandler(parse_reg_reg_imm, ex_btype, CommandData("
       1100011", "111")),
```

```
304   "lb":     CommandHandler(parse_reg_off_reg, ex_itype, CommandData("
       0000011", "000")),
305   "lh":     CommandHandler(parse_reg_off_reg, ex_itype, CommandData("
       0000011", "001")),
306   "lw":     CommandHandler(parse_reg_off_reg, ex_itype, CommandData("
       0000011", "010")),
307   "lbu":    CommandHandler(parse_reg_off_reg, ex_itype, CommandData("
       0000011", "100")),
308   "lhu":    CommandHandler(parse_reg_off_reg, ex_itype, CommandData("
       0000011", "101")),
309   "sb":     CommandHandler(parse_reg_off_reg, ex_stype, CommandData("
       0100011", "000")),
310   "sh":     CommandHandler(parse_reg_off_reg, ex_stype, CommandData("
       0100011", "001")),
311   "sw":     CommandHandler(parse_reg_off_reg, ex_stype, CommandData("
       0100011", "010")),
312   "addi":   CommandHandler(parse_reg_reg_imm, ex_itype, CommandData("
       0010011", "000")),
313   "slti":   CommandHandler(parse_reg_reg_imm, ex_itype, CommandData("
       0010011", "010")),
314   "sltiu":  CommandHandler(parse_reg_reg_imm, ex_itype, CommandData("
       0010011", "011")),
315   "xori":   CommandHandler(parse_reg_reg_imm, ex_itype, CommandData("
       0010011", "100")),
316   "ori":    CommandHandler(parse_reg_reg_imm, ex_itype, CommandData("
       0010011", "110")),
317   "andi":   CommandHandler(parse_reg_reg_imm, ex_itype, CommandData("
       0010011", "111")),
318   "add":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "000", "0000000")),
319   "sub":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "000", "0100000")),
320   "sll":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "001", "0000000")),
321   "slt":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "010", "0000000")),
322   "sltu":   CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "011", "0000000")),
323   "xor":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "100", "0000000")),
324   "srl":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "101", "0000000")),
325   "sra":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "101", "0100000")),
326   "or":     CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "110", "0000000")),
327   "and":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
       0110011", "111", "0000000")),
328   "slli":   CommandHandler(parse_reg_reg_imm, ex_sitype, CommandData("
       0010011", "001", "0000000")),
329   "srli":   CommandHandler(parse_reg_reg_imm, ex_sitype, CommandData("
       0010011", "101", "0000000")),
330   "srai":   CommandHandler(parse_reg_reg_imm, ex_sitype, CommandData("
       0010011", "101", "0100000")),
331
332
```

```
333   "ecall":  CommandHandler(parse_ecall  , ex_itype, CommandData("1110011",
        "000")),
334   "ebreak":   CommandHandler(parse_ebreak , ex_itype, CommandData("1110011
        ", "000")),
335
336
337   "mul":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
        0110011", "000", "0000001")),
338   "mulh":   CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
        0110011", "001", "0000001")),
339   "mulhsu":   CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
        0110011", "010", "0000001")),
340   "mulhu":  CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
        0110011", "011", "0000001")),
341   "div":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
        0110011", "100", "0000001")),
342   "divu":   CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
        0110011", "101", "0000001")),
343   "rem":    CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
        0110011", "110", "0000001")),
344   "remu":   CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData("
        0110011", "111", "0000001")),
345
346
347   "flw":    CommandHandler(parse_reg_off_reg, ex_itype_float, CommandData(
        "0000111", "010")),
348   "fsw":    CommandHandler(parse_reg_off_reg, ex_stype_float, CommandData(
        "0100111", "010")),
349
350
351   "fmadd.s":  CommandHandler(parse_reg_reg_reg_reg, ex_r4type, CommandData
        ("1000011", "111")),
352   "fmsub.s":  CommandHandler(parse_reg_reg_reg_reg, ex_r4type, CommandData
        ("1000111", "111")),
353   "fnmadd.s": CommandHandler(parse_reg_reg_reg_reg, ex_r4type, CommandData
        ("1001111", "111")),
354   "fnmsub.s": CommandHandler(parse_reg_reg_reg_reg, ex_r4type, CommandData
        ("1001011", "111")),
355
356
357   "fadd.s": CommandHandler(parse_reg_reg_reg, ex_rtype_float, CommandData(
        "1010011", "111", "0000000")),
358   "fsub.s": CommandHandler(parse_reg_reg_reg, ex_rtype_float, CommandData(
        "1010011", "111", "0000100")),
359   "fmul.s": CommandHandler(parse_reg_reg_reg, ex_rtype_float, CommandData(
        "1010011", "111", "0001000")),
360   "fdiv.s": CommandHandler(parse_reg_reg_reg, ex_rtype_float, CommandData(
        "1010011", "111", "0001100")),
361   "fsqrt.s":  CommandHandler(parse_reg_reg  , ex_rtype_float, CommandData(
        "1010011", "111", "0101100")),
362   "fsgnj.s":  CommandHandler(parse_reg_reg_reg, ex_rtype_float,
        CommandData("1010011", "000", "0010000")),
363   "fsgnjn.s": CommandHandler(parse_reg_reg_reg, ex_rtype_float,
        CommandData("1010011", "001", "0010000")),
364   "fsgnjx.s": CommandHandler(parse_reg_reg_reg, ex_rtype_float,
        CommandData("1010011", "010", "0010000")),
```

```
365   "fmin.s": CommandHandler(parse_reg_reg_reg, ex_rtype_float, CommandData(
       "1010011", "000", "0010100")),
366   "fmax.s": CommandHandler(parse_reg_reg_reg, ex_rtype_float, CommandData(
       "1010011", "001", "0010100")),
367   "fcvt.w.s": CommandHandler(parse_reg_reg  , ex_rtype_int_float,
       CommandData("1010011", "111", "1100000")),
368   "fcvt.wu.s":CommandHandler(parse_reg_reg_1  , ex_rtype_int_float,
       CommandData("1010011", "111", "1100000")),
369   "fmv.x.w":  CommandHandler(parse_reg_reg  , ex_rtype_int_float,
       CommandData("1010011", "000", "1110000")),
370   "feq.s":  CommandHandler(parse_reg_reg_reg, ex_rtype_int_float,
       CommandData("1010011", "010", "1010000")),
371   "flt.s":  CommandHandler(parse_reg_reg_reg, ex_rtype_int_float,
       CommandData("1010011", "001", "1010000")),
372   "fle.s":  CommandHandler(parse_reg_reg_reg, ex_rtype_int_float,
       CommandData("1010011", "000", "1010000")),
373   "fclass.s": CommandHandler(parse_reg_reg  , ex_rtype_int_float,
       CommandData("1010011", "001", "1110000")),
374   "fcvt.s.w": CommandHandler(parse_reg_reg  , ex_rtype_float_int,
       CommandData("1010011", "111", "1101000")),
375   "fcvt.s.wu":CommandHandler(parse_reg_reg_1  , ex_rtype_float_int,
       CommandData("1010011", "111", "1101000")),
376   "fmv.w.x":  CommandHandler(parse_reg_reg  , ex_rtype_float_int,
       CommandData("1010011", "000", "1111000")),
377
378
379    "lr.w":       CommandHandler(parse_reg_reg  , ex_rtype, CommandData("
      0101111", "010", "0001000")),
380    "lr.w.aq":      CommandHandler(parse_reg_reg  , ex_rtype, CommandData(
      "0101111", "010", "0001010")),
381    "lr.w.rl":      CommandHandler(parse_reg_reg  , ex_rtype, CommandData(
      "0101111", "010", "0001001")),
382    "lr.w.aq.rl":     CommandHandler(parse_reg_reg  , ex_rtype,
      CommandData("0101111", "010", "0001011")),
383    "sc.w":       CommandHandler(parse_reg_reg_reg, ex_rtype, CommandData(
      "0101111", "010", "0001100")),
384    "sc.w.aq":      CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0001110")),
385    "sc.w.rl":      CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0001101")),
386    "sc.w.aq.rl":     CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0001111")),
387    "amoswap.w":    CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0000100")),
388    "amoswap.w.aq":   CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0000110")),
389    "amoswap.w.rl":   CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0000101")),
390    "amoswap.w.aq.rl":  CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0000111")),
391    "amoadd.w":     CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0000000")),
392    "amoadd.w.aq":    CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0000010")),
393    "amoadd.w.rl":    CommandHandler(parse_reg_reg_reg, ex_rtype,
      CommandData("0101111", "010", "0000001")),
```

```
394     "amoadd.w.aq.rl":   CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0000011")),
395     "amoxor.w":     CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0010000")),
396     "amoxor.w.aq":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0010010")),
397     "amoxor.w.rl":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0010001")),
398     "amoxor.w.aq.rl":   CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0010011")),
399     "amoand.w":     CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0110000")),
400     "amoand.w.aq":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0110010")),
401     "amoand.w.rl":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0110001")),
402     "amoand.w.aq.rl":   CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0110011")),
403     "amoor.w":      CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0100000")),
404     "amoor.w.aq":     CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0100010")),
405     "amoor.w.rl":     CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0100001")),
406     "amoor.w.aq.rl":  CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "0100011")),
407     "amomin.w":     CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1000000")),
408     "amomin.w.aq":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1000010")),
409     "amomin.w.rl":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1000001")),
410     "amomin.w.aq.rl":   CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1000011")),
411     "amomax.w":     CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1010000")),
412     "amomax.w.aq":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1010010")),
413     "amomax.w.rl":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1010001")),
414     "amomax.w.aq.rl":   CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1010011")),
415     "amominu.w":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1100000")),
416     "amominu.w.aq":   CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1100010")),
417     "amominu.w.rl":   CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1100001")),
418     "amominu.w.aq.rl":  CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1100011")),
419     "amomaxu.w":    CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1110000")),
420     "amomaxu.w.aq":   CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1110010")),
421     "amomaxu.w.rl":   CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1110001")),
```

```
422     "amomaxu.w.aq.rl":  CommandHandler(parse_reg_reg_reg, ex_rtype,
    CommandData("0101111", "010", "1110011"))
423 }
424
425
426
427 if __name__ == "__main__":
428
429   Texto= open(input("Ingrese nombre del archivo: "))
430   with open("resultMedio","w") as file:
431     count=0
432     for line in Texto:
433
434       if(line!='\n'):
435         if(line.split("#")[0]!=''):
436           count=count+1
437           file.write(line.split("#")[0].strip("\t")+'\n')
438
439   Texto= open("resultMedio")
440   dictionary={}
441   count=0
442   for line in Texto:
443     if(line!='\n'):
444       if(':'in line):
445         dictionary[line.split(":")[0].strip(" ")]=count
446       else:
447         count=count+1
448
449   with open("result","w") as file:
450     Texto= open("resultMedio")
451     count=-1
452     for line in Texto:
453       if(line!='\n'):
454         if(not ':'in line):
455           line=line.replace('\t',' ')
456           count=count+1
457           for head in dictionary:
458             if (head in line):
459               line=line.replace(head,str((dictionary[head]-count)*4))
460
461           file.write(line)
462   Texto=''
463   os.remove('resultMedio')
464
465
466   with open("text_in","w") as file:
467     Texto= open("result")
468     for line in Texto:
469       lineVal=line.strip('\n')
470       lineVal=lineVal.strip().lower()
471       parts = lineVal.split(" ")
472       title = parts[0]
473       tail = "".join(parts[1:])
474       handler = handlers[title]
475       handler.parse(tail)
476       output = hex(int(handler.execute(),2))
```

```
      output2=output.split('x')[1]
      while (len(output2)<8):
        output2='0'+output2

      file.write(output2+'\n')

  Texto=''
  os.remove('result')

  open("data_in","w")


def singleWordPerLine2newFormat(words_per_line, number_of_lines,
    name_file_in, name_file_out):
    fr = open(name_file_in, 'r')
    fw = open(name_file_out, 'w')
    new_line = [0] * words_per_line
    for i in range(number_of_lines):
        for j in range(words_per_line):
            new_line[words_per_line - 1 - j] = fr.read(8)
            if new_line[words_per_line - 1 - j] == '':
                new_line[words_per_line - 1 - j] = '00000000'
            fr.read(1)
        for j in range(words_per_line):
            fw.write(new_line[j])
        fw.write('\n')
    fr.close()
    fw.close()

# 3906 lineas de 4 palabras => total_words = 3906*4 = 15624
singleWordPerLine2newFormat(4, 3906, "text_in", "text_in_formatted")
singleWordPerLine2newFormat(4, 3906, "data_in", "data_in_formatted")
```

# Anexo B

Archivo *Datapath* modificado para incorporar las instrucciones atómicas en el Soc.

```
/*
autor: Gianluca Vincenzo D'Agostino Matute
Santiago de Chile, Septiembre 2021

co-autor: Marcelo Urrutia
Santiago de Chile, Noviembre 2023
*/
`timescale 1ns / 1ps

module riscv32imf_pipeline(
    input clk, rst, start, acces_to_registers_files,
   is_wb_data_fp_fromEEI, do_wb_fromEEI, is_rs1_fp_fromEEI,
   is_rs2_fp_fromEEI,
    input acces_to_prog_mem, prog_rw_fromEEI, prog_valid_mem_fromEEI,
   prog_is_load_unsigned_fromEEI,
    input acces_to_data_mem, data_rw_fromEEI, data_valid_mem_fromEEI,
   data_is_load_unsigned_fromEEI,
    input [31:0] initial_PC, prog_addr_fromEEI, prog_in_fromEEI,
   data_addr_fromEEI, data_in_fromEEI, wb_data_fromEEI,
    input [1:0] prog_byte_half_word_fromEEI,
   data_byte_half_word_fromEEI,
    input [4:0] rs1_add_fromEEI, rs2_add_fromEEI, wb_add_fromEEI,
    output prog_ready_toEEI, prog_out_of_range_toEEI, data_ready_toEEI,
    data_out_of_range_toEEI,
    output reg ready,
    output [1:0] exit_status,
    output [31:0] prog_out_toEEI, data_out_toEEI, rs1_toEEI, rs2_toEEI,
    PC
    /*
        exit_status[1:0] =
            00 -> ecall
            11 -> ebreak
            01 -> program out of range
            10 -> memory out of range
    */
    );
    typedef enum {waiting_state, loop_state_1, loop_state_2_normal,
   loop_state_2_chazzard, loop_state_3, prog_error_state, final_state}
   state_type;
    state_type actual_state, next_state;
    reg set_regs, is_the_beginning, enable_execution, not_valid_EX,
   not_valid_MEM, not_valid_WB, control_bubble, acces_to_fpu_EX,
        acces_to_mem_EX, acces_to_mem_MEM, is_imm_valid_EX, is_ecall_EX
   , is_ecall_MEM, is_ebreak_EX, is_ebreak_MEM, is_ecall_WB,
   is_ebreak_WB;
```

```
33    reg is_storeConditional_EX, is_loadReserved_EX, is_Atomic_EX,
      is_AMO_EX;      //ComentarioAtomic
34    reg is_storeConditional_MEM, is_loadReserved_MEM,is_Atomic_MEM,
      is_AMO_MEM; //ComentarioAtomic
35    reg is_storeConditional_WB, is_loadReserved_WB, is_Atomic_WB,
      is_AMO_WB;      //ComentarioAtomic
36    reg store_conditions_EX, store_conditions_MEM, store_conditions_WB;
            //ComentarioAtomic
37    reg [4:0] rd_add_EX, rd_add_MEM, rd_add_WB, rs1_add_EX, rs2_add_EX,
      rs3_add_EX, funct5_EX, funct5_MEM, funct5_WB;  //ComentarioAtomic
38    reg [3:0] funct7_out_EX;
39    reg [2:0] format_type_MEM, sub_format_type_MEM, format_type_EX,
      sub_format_type_EX, format_type_WB, funct3_EX, funct3_MEM;
40    reg [1:0] reg_access_option_EX, reg_access_option_MEM,
      reg_access_option_WB;
41    reg [31:0] new_PC, PC_IF, PC_DR, PC_EX, PC_MEM, PC_WB, inst_DR,
      imm_EX, rs1_EX, rs2_EX, rs3_EX, final_res_MEM, operand3_MEM, rd_WB;
42    reg [31:0] rs1_WB, rs2_WB,rs1_MEM, rs2_MEM;
43
44    wire is_storeConditional, is_loadReserved, is_Atomic, is_AMO,
      store_conditions; //ComentarioAtomic
45    wire is_ecall_DR, is_ebreak_DR, acces_to_fpu_DR, acces_to_mem_DR,
      not_valid_DR, is_imm_valid_DR, ready_fpu, enable_fpu, exit, done,
46          ready_prog_mem, out_of_range_prog_mem, ready_data_mem,
      out_of_range_data_mem, rw_data_mem, is_load_unsigned_data_mem,
      branch_condition, control_hazard;
47    wire [6:0] ignored_signals;
48    wire [4:0] rd_add_DR, rs1_add_DR, rs2_add_DR, rs3_add_DR,
      alu_option, fpu_option, funct5_DR;     //ComentarioAtomic
49    wire [3:0] funct7_out_DR;
50    wire [2:0] format_type_DR, sub_format_type_DR, funct3_DR, rm2fpu;
51    wire [1:0] reg_access_option_DR, byte_half_word_data_mem;
52    wire [31:0] jalr_pc, branch_PC, rs1_DR, rs2_DR, rs3_DR, imm_DR,
      rd_MEM, operand1_EX, operand2_EX, operand3_EX, final_res_EX, inst_IF
      , res_alu, res_fpu, data_out_data_mem;
53
54
55    // interface con EEI
56    assign {prog_ready_toEEI, prog_out_of_range_toEEI    ,
      data_ready_toEEI   , data_out_of_range_toEEI    , prog_out_toEEI,
      data_out_toEEI     , rs1_toEEI , rs2_toEEI } =
57          {ready_prog_mem   , out_of_range_prog_mem      ,
      ready_data_mem     , out_of_range_data_mem      , inst_IF        ,
      data_out_data_mem  , rs1_DR     , rs2_DR     };
58    assign PC = (exit_status[1]^exit_status[0]) ? (exit_status[0] ?
      PC_IF : PC_MEM) : PC_WB;
59    // exit_status_selection
60    assign exit_status = is_ecall_WB
                    ? 2'b00 : (
```

```
61                          is_ebreak_WB
                        ? 2'b11 : (
62                          (acces_to_mem_MEM & out_of_range_data_mem) ? 2'b10 : (
63                          (out_of_range_prog_mem & ~control_hazard)    ? 2'b01 : (
64                          2'b00))));
65      // data_mem_options_selection
66      assign rw_data_mem = (format_type_MEM==3'b011 | store_conditions_MEM) ? 1'b1 : 1'b0; // only 1 in S type instruction and atomic instructions ComentarioAtomic
67      assign is_load_unsigned_data_mem = (format_type_MEM==3'b010 & funct3_MEM[2:1]==2'b10) ? 1'b1 : 1'b0; // only 1 in I type instruction and funct3= 4 o 5
68      assign byte_half_word_data_mem = (funct3_MEM[1:0]==2'b00 & ~store_conditions_MEM) ? 2'b10 : // funct3= 0 o 4 -byte
69                                        ((funct3_MEM[1:0]==2'b01 & ~store_conditions_MEM) ? 2'b01 : // funct3= 1 o 5 -half
70                                                                                                  2'b00);
        // funct3= 2     -word (default)
71      // final_alu_fpu_res_selection
72      assign final_res_EX =
73          (({format_type_EX, sub_format_type_EX}==6'b000_000 & {funct7_out_EX, funct3_EX}!=7'b1011_000 & {funct7_out_EX, funct3_EX}!=7'b1100_000) | // se excluyen fmv.x.w y fmv.w.x
74          format_type_EX==3'b001) ? res_fpu     : // R (fp)    o R4
75          ((({format_type_EX, sub_format_type_EX}==6'b000_000 & (funct7_out_EX==4'b1011 | funct7_out_EX==4'b1100))|is_Atomic_EX | store_conditions_EX) ? operand1_EX : // fmv.x.w y fmv.w.x  tiotiotio
76          (({format_type_EX, sub_format_type_EX}==6'b010_011 | format_type_EX==3'b110) ? PC_EX+32'b100 : // I (jalr) o J
77          ((format_type_EX==3'b101)     ?    ( operand2_EX + ( sub_format_type_EX[0] ? PC_EX : 32'b0) )    : // U
78          res_alu))); // default
                            // default
79      // mux rd
80      assign rd_MEM = ((format_type_MEM==3'b010 & ~sub_format_type_MEM[2] & ~sub_format_type_MEM[0]) | is_Atomic_MEM) ? data_out_data_mem : final_res_MEM; //ComentarioAtomic
81      // sub_decoder
82      assign is_ecall_DR  = (format_type_DR==3'b010 & sub_format_type_DR==3'b100 & imm_DR==32'b0) ? 1'b1 : 1'b0;
83      assign is_ebreak_DR = (format_type_DR==3'b010 & sub_format_type_DR==3'b100 & imm_DR==32'b1) ? 1'b1 : 1'b0;
84      assign acces_to_fpu_DR = (
85              format_type_DR==3'b001 | ( // R4
86              {format_type_DR, sub_format_type_DR}==6'b0 & {funct7_out_DR, funct3_DR}!=7'b1011_000 & {funct7_out_DR, funct3_DR}!=7'b1100_000 ) // R (fp) - {fmv.x.w, fmv.w.x}
87          ) ? 1'b1 : 1'b0;
```

```
88      assign acces_to_mem_DR = (format_type_DR==3'b011 | (format_type_DR
        ==3'b010 & ~sub_format_type_DR[2] & ~sub_format_type_DR[0]) |
        is_Atomic | store_conditions) ? 1'b1 : 1'b0;  //ComentarioAtomic
89      assign not_valid_DR = format_type_DR==3'b111 ? 1'b1 : 1'b0;
90      // next_PC_selection
91      assign control_hazard = ( ({format_type_EX, branch_condition}==4'
        b100_1 | format_type_EX==3'b110) | ({format_type_EX,
        sub_format_type_EX}==6'b010_011) ) ? 1'b1 : 1'b0;      //preguntar
92      assign jalr_pc = operand1_EX+operand2_EX;
93      assign branch_PC = ({format_type_EX, branch_condition}==4'b100_1 |
        format_type_EX==3'b110) ? PC_EX+operand2_EX : // B | J
94                         (({format_type_EX, sub_format_type_EX}==6'
        b010_011)                    ? {jalr_pc[31:1], 1'b0} : // I (jalr)
95
96                         32'b0);
97      // control_signals
98      assign exit = (is_ecall_WB | is_ebreak_WB |
99                     (out_of_range_prog_mem & ~control_hazard) |
100                    (acces_to_mem_MEM & out_of_range_data_mem) ) ? 1'b1 :
        1'b0;
101     assign done = (((ready_prog_mem & ~out_of_range_prog_mem) |
        control_hazard) &
102                    ((acces_to_fpu_EX & ready_fpu) | ~acces_to_fpu_EX) &
103                    ((acces_to_mem_MEM & ready_data_mem & ~
        out_of_range_data_mem) | ~acces_to_mem_MEM)) ? 1'b1 : 1'b0;
104     // FSM controller
105     always_comb
106         case (actual_state)
107             default: begin
108                 {is_the_beginning, ready}                    = 2'b00;
109                 {enable_execution, control_bubble, set_regs} = 3'b000;
110                 next_state = actual_state;
111             end
112             waiting_state: begin
113                 {is_the_beginning, ready}                    = {start,
        1'b0};
114                 {enable_execution, control_bubble, set_regs} = 3'b000;
115                 next_state = start ? loop_state_1 : waiting_state;
116             end
117             loop_state_1: begin
118                 {is_the_beginning, ready}                    = 2'b00;
119                 {enable_execution, control_bubble, set_regs} = 3'b100;
120                 next_state = start ? (exit ? final_state ://(
        store_conditions? loop_state_3 :
121                       (done ? (control_hazard ? loop_state_2_chazzard :
        loop_state_2_normal)
122                       : loop_state_1)) : waiting_state;
123             end
124             loop_state_2_normal: begin
125                 {is_the_beginning, ready}                    = 2'b00;
```

```
126                 {enable_execution, control_bubble, set_regs} = 3'b101;
127                 next_state = loop_state_3;
128             end
129             loop_state_2_chazzard: begin
130                 {is_the_beginning, ready}                    = 2'b00;
131                 {enable_execution, control_bubble, set_regs} = 3'b111;
132                 next_state = loop_state_3;
133             end
134             loop_state_3: begin
135                 {is_the_beginning, ready}                    = 2'b00;
136                 {enable_execution, control_bubble, set_regs} = 3'b000;
137                 next_state = loop_state_1;
138             end
139             final_state: begin
140                 {is_the_beginning, ready}                    = 2'b01;
141                 {enable_execution, control_bubble, set_regs} = 3'b100;
142                 next_state = start ? final_state : waiting_state;
143             end
144         endcase
145      always_ff @(posedge(clk)) begin // el orden SI importa
146         new_PC = is_the_beginning ? initial_PC : (set_regs ? (
    control_bubble ? branch_PC : PC_IF+32'b100) : PC_IF); //
    PC_generator
147
148
149         if(is_the_beginning) begin // reset
150             // PipelineReg_MEM2WB
151             {PC_WB, rd_WB, rd_add_WB, format_type_WB,
    reg_access_option_WB, not_valid_WB, is_ebreak_WB, is_ecall_WB ,
152              is_loadReserved_WB, is_storeConditional_WB,
    store_conditions_WB    , is_Atomic_WB   , is_AMO_WB      ,funct5_WB,
153              rs1_WB, rs2_WB} = 0;
154             // PipelineReg_EX2MEM
155             {PC_MEM, final_res_MEM, operand3_MEM, rd_add_MEM,
    format_type_MEM, sub_format_type_MEM, funct3_MEM,
156              reg_access_option_MEM, acces_to_mem_MEM, not_valid_MEM,
    is_ebreak_MEM, is_ecall_MEM,
157              is_loadReserved_MEM, is_storeConditional_MEM,
    store_conditions_MEM    , is_Atomic_MEM   , is_AMO_MEM      ,
    funct5_MEM,
158              rs1_MEM, rs2_MEM} = 0;
159
160             {PC_EX, imm_EX, rs1_EX, rs2_EX, rs3_EX, rd_add_EX,
    rs1_add_EX, rs2_add_EX, rs3_add_EX,
161              funct7_out_EX, funct3_EX, format_type_EX,
    sub_format_type_EX, reg_access_option_EX,
162              is_imm_valid_EX, acces_to_fpu_EX, acces_to_mem_EX,
    not_valid_EX, is_ebreak_EX, is_ecall_EX,
163              is_loadReserved_EX, is_storeConditional_EX,
    store_conditions_EX    , is_Atomic_EX   , is_AMO_EX ,funct5_EX} = 0;
```

```
164             // PipelineReg_IF2DR
165             {PC_DR, inst_DR} = 0;
166              PC_IF = new_PC;
167         end
168 
169 
170         else if(set_regs) begin // set
171             if (store_conditions) begin
172                 PC_IF = PC_WB+32'b100;
173 
174                 {PC_DR, inst_DR} = 0;
175 
176                 {PC_EX, imm_EX, rs1_EX, rs2_EX, rs3_EX, rd_add_EX,
    rs1_add_EX, rs2_add_EX, rs3_add_EX,
177                  funct7_out_EX, funct3_EX, format_type_EX,
    sub_format_type_EX, reg_access_option_EX,
178                  is_imm_valid_EX, acces_to_fpu_EX, acces_to_mem_EX,
    not_valid_EX, is_ebreak_EX, is_ecall_EX,
179                  is_loadReserved_EX, is_storeConditional_EX,
    store_conditions_EX     , is_Atomic_EX   , is_AMO_EX ,funct5_EX}
180                 =
181                 {PC_WB, 32'b0 , rs1_DR, rs2_DR, 32'b0 , 5'b0, 5'b0, 5'
    b0, 5'b0,
182                 4'b1111, 3'b010, 3'b111, 3'b111, 2'b00,
183                 1'b0, 1'b0, acces_to_mem_DR, 1'b0, 1'b0, 1'b0,
184                 1'b0, 1'b0, store_conditions, 1'b0, 1'b0,5'b11111};
185 
186                 // PipelineReg_EX2MEM
187                 {PC_MEM, final_res_MEM, operand3_MEM, rd_add_MEM,
    format_type_MEM, sub_format_type_MEM, funct3_MEM,
188                  reg_access_option_MEM, acces_to_mem_MEM, not_valid_MEM
    , is_ebreak_MEM, is_ecall_MEM,
189                  is_loadReserved_MEM, is_storeConditional_MEM,
    store_conditions_MEM     , is_Atomic_MEM   , is_AMO_MEM      ,
    funct5_MEM} = 0;
190 
191                 {PC_WB,  rd_WB,  rd_add_WB,  format_type_WB,
    reg_access_option_WB,  not_valid_WB,  is_ebreak_WB,  is_ecall_WB  ,
192                  is_loadReserved_WB, is_storeConditional_WB,
    store_conditions_WB     , is_Atomic_WB   , is_AMO_WB      ,funct5_WB,
193                  rs1_WB, rs2_WB} = 0;
194 
195             end
196 
197             else begin
198                 // PipelineReg_MEM2WB
199                 {PC_WB,  rd_WB,  rd_add_WB,  format_type_WB,
    reg_access_option_WB,  not_valid_WB,  is_ebreak_WB,  is_ecall_WB  ,
200                  is_loadReserved_WB  , is_storeConditional_WB  ,
    store_conditions_WB   , is_Atomic_WB   , is_AMO_WB      ,funct5_WB,
```

```
201                 rs1_WB, rs2_WB}
202                 =
203                 {PC_MEM, rd_MEM, rd_add_MEM, format_type_MEM,
    reg_access_option_MEM, not_valid_MEM, is_ebreak_MEM, is_ecall_MEM ,
204                  is_loadReserved_MEM, is_storeConditional_MEM,
    store_conditions_MEM , is_Atomic_MEM , is_AMO_MEM ,funct5_MEM,
205                  rs1_MEM, rs2_MEM};
206                 // PipelineReg_EX2MEM
207                 {PC_MEM, final_res_MEM, operand3_MEM, rd_add_MEM,
    format_type_MEM, sub_format_type_MEM, funct3_MEM,
208                  reg_access_option_MEM, acces_to_mem_MEM, not_valid_MEM
    , is_ebreak_MEM, is_ecall_MEM ,
209                  is_loadReserved_MEM, is_storeConditional_MEM,
    store_conditions_MEM , is_Atomic_MEM , is_AMO_MEM ,funct5_MEM,
210                  rs1_MEM, rs2_MEM}
211                 =
212                 {PC_EX, final_res_EX, operand3_EX, rd_add_EX,
    format_type_EX, sub_format_type_EX, funct3_EX,
213                  reg_access_option_EX, acces_to_mem_EX, not_valid_EX,
      is_ebreak_EX, is_ecall_EX ,
214                  is_loadReserved_EX, is_storeConditional_EX,
    store_conditions_EX , is_Atomic_EX , is_AMO_EX ,funct5_EX,
215                  operand1_EX, operand2_EX};
216
217
218                 if(control_bubble) begin // reset (c. hazard)
219                     // PipelineReg_DR2EX
220                     {PC_EX, imm_EX, rs1_EX, rs2_EX, rs3_EX, rd_add_EX,
    rs1_add_EX, rs2_add_EX, rs3_add_EX,
221                      funct7_out_EX, funct3_EX, format_type_EX,
    sub_format_type_EX, reg_access_option_EX,
222                      is_imm_valid_EX, acces_to_fpu_EX, acces_to_mem_EX,
     not_valid_EX, is_ebreak_EX, is_ecall_EX,
223                      is_loadReserved_EX, is_storeConditional_EX,
    store_conditions_EX , is_Atomic_EX , is_AMO_EX ,funct5_EX} = 0;
224                     // PipelineReg_IF2DR
225                     {PC_DR, inst_DR} = 0;
226                 end
227                 else if(~control_bubble) begin // set (c. hazard)
228                     // PipelineReg_DR2EX
229                     {PC_EX, imm_EX, rs1_EX, rs2_EX, rs3_EX, rd_add_EX,
    rs1_add_EX, rs2_add_EX, rs3_add_EX,
230                      funct7_out_EX, funct3_EX, format_type_EX,
    sub_format_type_EX, reg_access_option_EX,
231                      is_imm_valid_EX, acces_to_fpu_EX, acces_to_mem_EX,
     not_valid_EX, is_ebreak_EX, is_ecall_EX,
232                      is_loadReserved_EX , is_storeConditional_EX,
    store_conditions_EX , is_Atomic_EX , is_AMO_EX ,funct5_EX}
233                     =
```

```
234                     {PC_DR, imm_DR, rs1_DR, rs2_DR, rs3_DR, rd_add_DR,
    rs1_add_DR, rs2_add_DR, rs3_add_DR,
235                      funct7_out_DR, funct3_DR, format_type_DR,
    sub_format_type_DR, reg_access_option_DR,
236                      is_imm_valid_DR, acces_to_fpu_DR, acces_to_mem_DR,
     not_valid_DR, is_ebreak_DR, is_ecall_DR,
237                      is_loadReserved     , is_storeConditional    ,
    store_conditions       , is_Atomic      , is_AMO     ,funct5_DR};
238                     // PipelineReg_IF2DR
239                     {PC_DR, inst_DR} = {PC_IF, inst_IF};
240                     PC_IF = new_PC; // PC_generator
241                 end
242             end
243         end
244
245
246         actual_state = rst ? waiting_state : next_state;
247
248
249     end
250     // prog_memory
251     memory #(.initial_option(2)) prog_memory_unit(
252         .clk(clk), .rst(rst),
253         // inputs
254         .rw(              (acces_to_prog_mem & ~start) ?
    prog_rw_fromEEI                 : 1'b0),
255         .valid(           (acces_to_prog_mem & ~start) ?
    prog_valid_mem_fromEEI          : enable_execution),
256         .addr(            (acces_to_prog_mem & ~start) ?
    prog_addr_fromEEI               : PC_IF),
257         .data_in(         (acces_to_prog_mem & ~start) ?
    prog_in_fromEEI                 : 32'b0),
258         .byte_half_word(  (acces_to_prog_mem & ~start) ?
    prog_byte_half_word_fromEEI     : 2'b0),
259         .is_load_unsigned((acces_to_prog_mem & ~start) ?
    prog_is_load_unsigned_fromEEI : 1'b1),
260         // outputs
261         .ready(ready_prog_mem), .out_of_range(out_of_range_prog_mem), .
    data_out(inst_IF)
262     );
263     // data_memory
264     memory #(.initial_option(1)) data_memory_unit(
265         .clk(clk), .rst(rst),
266         // inputs
267         .rw(              (acces_to_data_mem & ~start) ?
    data_rw_fromEEI                 : rw_data_mem),
268         .valid(           (acces_to_data_mem & ~start) ?
    data_valid_mem_fromEEI          : acces_to_mem_MEM & enable_execution)
    ,
```

```
269         .addr(              (acces_to_data_mem & ~start) ?
    data_addr_fromEEI               : final_res_MEM),
270         .data_in(           (acces_to_data_mem & ~start) ?
    data_in_fromEEI                 : operand3_MEM),
271         .byte_half_word(    (acces_to_data_mem & ~start) ?
    data_byte_half_word_fromEEI     : byte_half_word_data_mem),
272         .is_load_unsigned((acces_to_data_mem & ~start) ?
    data_is_load_unsigned_fromEEI : is_load_unsigned_data_mem),
273         // outputs
274         .ready(ready_data_mem), .out_of_range(out_of_range_data_mem), .
    data_out(data_out_data_mem)
275     );
276     // Instruction_Decoder
277     Instruction_Decoder Instruction_Decoder_unit(
278         .in(inst_DR),
279         .is_imm_valid(is_imm_valid_DR), .imm(imm_DR),
280         .rd_add(rd_add_DR), .rs1_add(rs1_add_DR), .rs2_add(rs2_add_DR),
     .rs3_add(rs3_add_DR),
281         .funct7_out(funct7_out_DR),.funct5(funct5_DR), .format_type(
    format_type_DR), .sub_format_type(sub_format_type_DR),
282         .funct3(funct3_DR), .funct2(ignored_signals[6:5]), .
    reg_access_option(reg_access_option_DR)
283     );
284
285     checkAtomic checkAtomic_unit(
286         .format_type(format_type_DR), .sub_format_type(
    sub_format_type_DR),
287         .funct5(funct5_DR),
288         .is_storeConditional(is_storeConditional), .is_loadReserved(
    is_loadReserved), .is_Atomic(is_Atomic), .is_AMO(is_AMO)
289     );
290     // registers_files
291     registers_files registers_files_unit(
292         .clk(clk),
293         // inputs
294         .rs1_add( (acces_to_registers_files & ~start) ? rs1_add_fromEEI
     : rs1_add_DR),
295         .rs2_add( (acces_to_registers_files & ~start) ? rs2_add_fromEEI
     : rs2_add_DR),
296         .rs3_add( rs3_add_DR ),
297         .wb_add(  (acces_to_registers_files & ~start) ? wb_add_fromEEI
     : rd_add_WB),
298         .wb_data( (acces_to_registers_files & ~start) ? wb_data_fromEEI
     : rd_WB),
299         .write_reg( (acces_to_registers_files & ~start) ? do_wb_fromEEI
     :
300             ((format_type_WB==3'b011 | format_type_WB==3'b100 |
    not_valid_WB | is_ecall_WB | is_ebreak_WB|store_conditions_WB) ? 1'
    b0 : 1'b1) ),
```

```
301         .is_wb_data_fp( (acces_to_registers_files & ~start) ? is_wb_data_fp_fromEEI : reg_access_option_WB[0]),
302         .is_rs1_fp( (acces_to_registers_files & ~start) ? is_rs1_fp_fromEEI : reg_access_option_DR[1]),
303         .is_rs2_fp( (acces_to_registers_files & ~start) ? is_rs2_fp_fromEEI : (reg_access_option_DR[1] | reg_access_option_DR[0])),
304         .is_loadReserved_WB(is_loadReserved_WB),
305         .is_storeConditional_WB(is_storeConditional_WB),
306         .isAMO_wb(is_AMO_WB),          //indica si el acceso a reg es AMO en wb
307         .funt_5_wb(funct5_WB),  // funct 5 wb
308         .wb_rs1(rs1_WB),     // data rs1 wb
309         .wb_rs2(rs2_WB),     // data rs2 wb
310         // outputs
311         .rs1(rs1_DR), .rs2(rs2_DR), .rs3(rs3_DR), .store_conditions_out(store_conditions)
312      );
313      // forwarding_unit
314      forwarding_unit forwarding_unit(
315         .store_conditions_EX(store_conditions_EX), .store_conditions_MEM(store_conditions_MEM), .store_conditions_WB(store_conditions_WB),
316         .is_imm_valid_EX(is_imm_valid_EX), .ready_data_mem(ready_data_mem), .valid_data_mem(acces_to_mem_MEM),
317         .reg_access_option_EX(reg_access_option_EX), .reg_access_option_MEM(reg_access_option_MEM), .reg_access_option_WB(reg_access_option_WB),
318         .format_type_EX(format_type_EX), .format_type_MEM(format_type_MEM), .format_type_WB(format_type_WB), .sub_format_type_MEM(sub_format_type_MEM),
319         .rs1_add_EX(rs1_add_EX), .rs2_add_EX(rs2_add_EX), .rs3_add_EX(rs3_add_EX), .rd_add_MEM(rd_add_MEM), .rd_add_WB(rd_add_WB),
320         .rs1_EX(rs1_EX), .rs2_EX(rs2_EX), .rs3_EX(rs3_EX), .rd_MEM(rd_MEM), .rd_WB(rd_WB), .imm_EX(imm_EX),
321         .enable_fpu(enable_fpu), .operand1_EX(operand1_EX), .operand2_EX(operand2_EX), .operand3_EX(operand3_EX)
322      );
323      // alu_op_selection
324      alu_op_selection alu_op_selection_unit(
325         .imm_11_5(operand2_EX[11:5]), .funct7_out(funct7_out_EX),
326         .format_type(format_type_EX), .sub_format_type(sub_format_type_EX), .funct3(funct3_EX),
327         .alu_option(alu_option)
328      );
329      // ALU
330      ALU ALU_unit(
331         .in1(operand1_EX), .in2((format_type_EX==3'b100) ? operand3_EX : operand2_EX), // is B type?
332         .operation(alu_option),
```

```
333         .res(res_alu), .boolean_res(branch_condition)
334     );
335     // fpu_op_selection
336     fpu_op_selection fpu_op_selection_unit(
337         .rs2_add(rs2_add_EX), .funct7_out(funct7_out_EX),
338         .format_type(format_type_EX), .sub_format_type(
    sub_format_type_EX), .funct3(funct3_EX), .rm_from_fcsr(3'b000),
339         .rm2fpu(rm2fpu), .fpu_option(fpu_option)
340     );
341     // FPU
342     FPU FPU_unit(
343         .start(acces_to_fpu_EX & enable_fpu & enable_execution), .rst(
    rst | ~enable_execution), .clk(clk), .rm(rm2fpu),
344         .option(fpu_option), .in1(operand1_EX), .in2(operand2_EX), .in3
    (operand3_EX),
345         .NV(ignored_signals[4]), .NX(ignored_signals[3]), .UF(
    ignored_signals[2]), .OF(ignored_signals[1]), .DZ(ignored_signals
    [0]),
346         .ready(ready_fpu), .out(res_fpu)
347     );
348 endmodule
349 
350 module forwarding_unit(
351     input store_conditions_EX, store_conditions_MEM,
    store_conditions_WB,
352     input is_imm_valid_EX, ready_data_mem, valid_data_mem,
353     input [1:0] reg_access_option_EX, reg_access_option_MEM,
    reg_access_option_WB,
354     input [2:0] format_type_EX, format_type_MEM, format_type_WB,
    sub_format_type_MEM,
355     input [4:0] rs1_add_EX, rs2_add_EX, rs3_add_EX, rd_add_MEM,
    rd_add_WB,
356     input [31:0] rs1_EX, rs2_EX, rs3_EX, rd_MEM, rd_WB, imm_EX,
357     output enable_fpu,
358     output [31:0] operand1_EX, operand2_EX, operand3_EX
359     );
360     /*
361         reg_access_option:
362         |code| rd | rs1 | rs2 |
363         | 00 | I  | I   | I   |
364         | 01 | FP | I   | FP  |
365         | 10 | I  | FP  | FP  |
366         | 11 | FP | FP  | FP  |
367         note: rs3 is always FP
368     */
369     wire [31:0] final_rs1, final_rs2, final_rs3;
370     //tiotiotio
371     // checking rs1, rs2, rs3: se debe checkear MEM primero y despues
    WB (debido a que MEM el la inst inmediatamente precedente)
```

```
372     //                    calza la direcci n          no se trata de x0
                                      es un write back por realizar
                                           son del mismo tipo (integer
    /fp)
373     assign final_rs1 = ( rs1_add_EX==rd_add_MEM & ~( rd_add_MEM==5'b0 &
     ~reg_access_option_MEM[0] ) & format_type_MEM!=3'b011 &
    format_type_MEM!=3'b100 & ~store_conditions_MEM &
    reg_access_option_EX[1]==reg_access_option_MEM[0] ) ? rd_MEM : (
374                         ( rs1_add_EX==rd_add_WB  & ~( rd_add_WB ==5'b0 &
     ~reg_access_option_WB [0] ) & format_type_WB !=3'b011 &
    format_type_WB !=3'b100 & ~store_conditions_WB &
    reg_access_option_EX[1]==reg_access_option_WB[0]  ) ? rd_WB  : (
375                           rs1_EX));
376     assign final_rs2 = ( rs2_add_EX==rd_add_MEM & ~( rd_add_MEM==5'b0 &
     ~reg_access_option_MEM[0] ) & format_type_MEM!=3'b011 &
    format_type_MEM!=3'b100 & ~store_conditions_MEM & ((
    reg_access_option_EX==2'b0 & ~reg_access_option_MEM[0]) | (
    reg_access_option_EX!=2'b0 & reg_access_option_MEM[0])) ) ? rd_MEM :
     (
377                         ( rs2_add_EX==rd_add_WB  & ~( rd_add_WB ==5'b0 &
     ~reg_access_option_WB [0] ) & format_type_WB !=3'b011 &
    format_type_WB !=3'b100 & ~store_conditions_WB & ((
    reg_access_option_EX==2'b0 & ~reg_access_option_WB[0] ) | (
    reg_access_option_EX!=2'b0 & reg_access_option_WB[0] )) ) ? rd_WB  :
     (
378                           rs2_EX));
379     assign final_rs3 = ( rs3_add_EX==rd_add_MEM & ~( rd_add_MEM==5'b0 &
     ~reg_access_option_MEM[0] ) & format_type_MEM!=3'b011 &
    format_type_MEM!=3'b100 & ~store_conditions_MEM &
    reg_access_option_MEM[0] ) ? rd_MEM : (
380                         ( rs3_add_EX==rd_add_WB  & ~( rd_add_WB ==5'b0 &
     ~reg_access_option_WB [0] ) & format_type_WB !=3'b011 &
    format_type_WB !=3'b100 & ~store_conditions_WB &
    reg_access_option_WB [0] ) ? rd_WB  : (
381                           rs3_EX));
382     // final outputs (analogo al caso single cycle)
383     assign operand1_EX = final_rs1;
384     assign operand2_EX = is_imm_valid_EX ? imm_EX : final_rs2;
385     assign operand3_EX = (format_type_EX==3'b011 | format_type_EX==3'
    b100 | store_conditions_EX) ? final_rs2 : final_rs3;
386     // permite la ejecuci n de la FPU, es importante en las
    instrucciones load cuando hay data hazard pues debe esperar a que se
     obtenga el dato de la memoria
387     assign enable_fpu = ( // si alguno de los operandos depende de
    rd_MEM
388           ( (rs1_add_EX==rd_add_MEM & format_type_EX!=3'b101 &
    format_type_EX!=3'b110) | // no aplica para U y J
389             (rs2_add_EX==rd_add_MEM & (~is_imm_valid_EX |
    format_type_EX==3'b011 | format_type_EX==3'b100)) | // aplica si imm
     es inv lido o para B o S   |store_conditions_EX
```

```
390             (rs3_add_EX==rd_add_MEM & format_type_EX==3'b001) // aplica
    para R4
391           ) & (((~sub_format_type_MEM[2] & ~sub_format_type_MEM[0] &
    format_type_MEM==3'b010) ) & valid_data_mem) // y es una
    instrucci n de tipo load (en MEM) | (format_type_MEM == 3'b000 &
    sub_format_type_MEM == 3'b010)
392           & ~( rd_add_MEM==5'b0 & ~reg_access_option_MEM[0] ) // y no
    se trata de x0=zero
393           ) ? ready_data_mem : 1'b1; // se espera a que el dato sea
    valido
394 endmodule
```

# Anexo C

Archivo *Instruction Decoder* modificado para incorporar las instrucciones atómicas en el Soc.

```
/*
autor: Gianluca Vincenzo D'Agostino Matute
Santiago de Chile, Septiembre 2021

co-autor: Marcelo Urrutia
Santiago de Chile, Noviembre 2023
*/
`timescale 1ns / 1ps

module Instruction_Decoder(
    input [31:0] in,
    output is_imm_valid,
    output [31:0] imm,
    output [4:0] rd_add, rs1_add, rs2_add, rs3_add, funct5,
    output [3:0] funct7_out,
    output [2:0] format_type, sub_format_type, funct3,
    output [1:0] reg_access_option, funct2
    );
    wire [6:0] opcode, funct7;
    assign {opcode, funct7   , funct3   , funct5   , funct2   , rd_add
     , rs1_add   , rs2_add   , rs3_add  }
        = {in[6:0], in[31:25], in[14:12], in[31:27], in[26:25], in
   [11:7], in[19:15], in[24:20], in[31:27]};
    /*
        Format Group Classifier:
        opcode   -> Type
        0110011 -> R
        1010011 -> R (FP)
        0101111 -> R (A)
        100 00 11 -> R4 (FP- fmadd.s)
        100 01 11 -> R4 (FP- fmsub.s)
        100 11 11 -> R4 (FP- fnmadd.s)
        100 10 11 -> R4 (FP- fnmsub.s)
        0100011 -> S
        0100111 -> S (FP)
        1100011 -> B
        1101111 -> J
        0110111 -> U (lui)
        0010111 -> U (auipc)
        0010011 -> I (addi, xori, ori, andi, slli, srli, srai, slti,
   sltiu)
        0000011 -> I (lb, lh, lw, lbu, lhu)
        1100111 -> I (jalr)
        1110011 -> I (ecall, ebreak)
```

```
42          0000111 -> I (FP)
43          format_type[2:0]:
44          000 -> R type
45          001 -> R4 type
46          011 -> S type
47          100 -> B type
48          101 -> U type
49          110 -> J type
50          010 -> I type
51          111 -> Not valid type
52       */
53       assign format_type = ((opcode[4:0]==5'b10011 & (opcode[6]^opcode
         [5]))|opcode==7'b0101111) ? 3'b000 : // R
54                            (({opcode[6:4], opcode[1:0]}==5'b10011)
          ? 3'b001 : // R4
55                            (({opcode[6:3], opcode[1:0]}==6'b010011)
          ? 3'b011 : // S
56                            ((opcode==7'b1100011)
          ? 3'b100 : // B
57                            ((opcode==7'b1101111)
          ? 3'b110 : // J
58                            (({opcode[6], opcode[4:0]}==6'b010111)
          ? 3'b101 : // U
59                            ((opcode[1:0]==2'b11&(opcode[6:2]==5'b100|
         opcode[6:2]==5'b0|opcode[6:2]==5'b1
60                            |opcode[6:2]==5'b11001|opcode[6:2]==5'b11100))
          ? 3'b010 : // I
61
         3'b111))))))); // Not valid
62       /*
63           Sub-Format Group Classifier
64               sub_format_type[2:0]:
65                   if format_type[2:0] = 000
66                   -000 -> R (FP)
67                   -001 -> R
68                   -010 -> R (A)
69                   if format_type[2:0] = 001
70                   -000 -> R4 (FP- fmadd.s)
71                   -001 -> R4 (FP- fmsub.s)
72                   -010 -> R4 (FP- fnmsub.s)
73                   -011 -> R4 (FP- fnmadd.s)
74                 100 00 11 -> R4 (FP- fmadd.s)
75                 100 01 11 -> R4 (FP- fmsub.s)
76                 100 10 11 -> R4 (FP- fnmsub.s)
77                 100 11 11 -> R4 (FP- fnmadd.s)
78                   if format_type[2:0] = 010
79                   -000 -> I (FP)
80                   -001 -> I (addi, xori, ori, andi, slli, srli, srai,
         slti, sltiu)
81                   -010 -> I (lb, lh, lw, lbu, lhu)
```

```
82                  -011 -> I (jalr)
83                  -100 -> I (ecall, ebreak)
84                  if format_type[2:0] = 011
85                  -000 -> S (FP)
86                  -001 -> S
87                  if format_type[2:0] = 101
88                  -000 -> U (lui)
89                  -001 -> U (auipc)
90                  else
91                  -111
92      */
93      assign sub_format_type = (format_type==3'b000 & opcode[5:4]==2'b01)
      ? 3'b000 : // R (FP)
94                               ((format_type==3'b000 & opcode[5:4]==2'b11)
      ? 3'b001 : // R (I)
95                               ((format_type==3'b000 & opcode[5:4]==2'b10)
      ? 3'b010 : // R (A)
96                               ((format_type==3'b011) ? (opcode[2] ? 3'
    b000 : 3'b001) : // S
97                               ((format_type==3'b101) ? (opcode[5] ? 3'
    b000 : 3'b001) : // U
98                               ((format_type==3'b001) ? opcode[4:2]
                 : // R4
99                               ((format_type==3'b010&opcode[6:2]==5'b1)
        ? 3'b000 : // I (FP)
100                              ((format_type==3'b010&opcode[6:2]==5'b0)
        ? 3'b010 : // I (lb, lh, lw, lbu, lhu)
101                              ((format_type==3'b010&opcode[6:2]==5'b100)
        ? 3'b001 : // I (addi, xori, ori, andi, slli, srli, srai, slti,
    sltiu)
102                              ((format_type==3'b010&opcode[6:2]==5'b11001
    )  ? 3'b011 : // I (jalr)
103                              ((format_type==3'b010&opcode[6:2]==5'b11100
    )  ? 3'b100 : // I (ecall, ebreak)
104                                 3'b111))))))))));
105     /*
106         Opration Classifier:
107         if  Funct7[6:0]=0000000: Funct7-out[3:0]=0000
108         if  Funct7[6:0]=0000001: Funct7-out[3:0]=0001
109         if  Funct7[6:0]=0100000: Funct7-out[3:0]=0010
110         if  Funct7[6:0]=0000100: Funct7-out[3:0]=0011
111         if  Funct7[6:0]=0001000: Funct7-out[3:0]=0100
112         if  Funct7[6:0]=0001100: Funct7-out[3:0]=0101
113         if  Funct7[6:0]=0101100: Funct7-out[3:0]=0110
114         if  Funct7[6:0]=0010000: Funct7-out[3:0]=0111
115         if  Funct7[6:0]=0010100: Funct7-out[3:0]=1000
116         if  Funct7[6:0]=1101000: Funct7-out[3:0]=1001
117         if  Funct7[6:0]=1100000: Funct7-out[3:0]=1010
118         if  Funct7[6:0]=1110000: Funct7-out[3:0]=1011
119         if  Funct7[6:0]=1111000: Funct7-out[3:0]=1100
```

```
120         if  Funct7[6:0]=1010000: Funct7-out[3:0]=1101
121         else:
122          Funct7-out[3:0]=1111
123     */
124     assign funct7_out = (funct7==7'b0000000) ? 4'b0000 : // 0x00
125                         ((funct7==7'b0000001) ? 4'b0001 : // 0x01
126                         ((funct7==7'b0100000) ? 4'b0010 : // 0x20
127                         ((funct7==7'b0000100) ? 4'b0011 :
128                         ((funct7==7'b0001000) ? 4'b0100 :
129                         ((funct7==7'b0001100) ? 4'b0101 :
130                         ((funct7==7'b0101100) ? 4'b0110 :
131                         ((funct7==7'b0010000) ? 4'b0111 :
132                         ((funct7==7'b0010100) ? 4'b1000 :
133                         ((funct7==7'b1101000) ? 4'b1001 :
134                         ((funct7==7'b1100000) ? 4'b1010 :
135                         ((funct7==7'b1110000) ? 4'b1011 :
136                         ((funct7==7'b1111000) ? 4'b1100 :
137                         ((funct7==7'b1010000) ? 4'b1101 :
138                          4'b1111))))))))))))));
139     /*
140         Registers selection options
141         if format-type[2:0]=000 (R):
142           if sub-format-type[2:0]=000 (FP):
143             if Funct7-out[3:0]=1010 o 1011 o 1101 ( fwt.w(u).s fmv.x.w
    fclass.s feq.s flt.s fle.s )
144               reg_access_option[1:0]=10
145             if Funct7-out[3:0]=1001 o 1100 ( fwt.s.w(u) fmv.w.x )
146               reg_access_option[1:0]=01
147             else:
148               reg_access_option[1:0]=11
149
150           if sub-format-type[2:0]=001 (no FP):
151             reg_access_option[1:0]=00
152         if format-type[2:0]=010 (I):
153           if sub-format-type[2:0]=000 (FP):  reg_access_option[1:0]=01
154           if sub-format-type[2:0]=001 o 010 o 011 o 100 (no FP):
    reg_access_option[1:0]=00
155         if format-type[2:0]=011 (S):
156           if sub-format-type[2:0]=000 (FP):  reg_access_option[1:0]=01
157           if sub-format-type[2:0]=001 (no FP):  reg_access_option
    [1:0]=00
158         if format-type[2:0]=001 (R4):
159           reg_access_option[1:0]=11
160         if format-type[2:0]=100 (B):
161           reg_access_option[1:0]=00
162         if format-type[2:0]=101 (U) o 110 (J):
163           reg_access_option[1:0]=00 o 10=00
164         reg_access_option[1:0] =
165             00: rd, rs1, rs2 son registros Integer
166             01: rs1 es reg Integer y rd, rs2 son FP
```

```
167             10: rd es reg Integer y rs1, rs2 son FP
168             11: rd, rs1, rs2 son registros FP
169         reg_access_option:
170         |code| rd | rs1 | rs2 |
171         | 00 | I  | I   | I   |
172         | 01 | FP | I   | FP  |
173         | 10 | I  | FP  | FP  |
174         | 11 | FP | FP  | FP  |
175         note: rs3 is always FP
176      */
177      assign reg_access_option = (format_type[2:1]==2'b10|format_type==3'
    b110) ? 2'b00 : // B-U-J
178                                 ((format_type==3'b001)
           ? 2'b11 : // R4
179                                 ((format_type==3'b011&sub_format_type==3'
    b000) ? 2'b01 : // S (FP)
180                                 ((format_type==3'b011&sub_format_type==3'
    b001) ? 2'b00 : // S (no FP)
181                                 ((format_type==3'b010&sub_format_type==3'
    b000) ? 2'b01 : // I (FP)
182                                 ((format_type==3'b010&sub_format_type!=3'
    b111) ? 2'b00 : // I (no FP)
183                                 ((format_type==3'b000&sub_format_type==3'
    b001) ? 2'b00 : // R (no FP)
184                                 ((format_type==3'b000&sub_format_type==3'
    b010) ? 2'b00 : // R (A)
185                                 ((format_type==3'b000&sub_format_type==3'
    b000&(funct7_out==4'b1010|
186                                  funct7_out==4'b1011|funct7_out==4'b1101
       )) ? 2'b10 : // R (FP) ( fwt.w(u).s fmv.x.w fclass.s feq.s flt.s
    fle.s )
187                                 ((format_type==3'b000&sub_format_type==3'
    b000&(
188                                  funct7_out==4'b1001|funct7_out==4'b1100
       )) ? 2'b01 : // R (FP) ( fwt.s.w(u) fmv.w.x )
189                                 ((format_type==3'b000&sub_format_type==3'
    b000&(
190                                  funct7_out!=4'b1111
       )) ? 2'b11 : // R (FP) ( ~ )
191                                  2'b00))))))))));
192      /*
193          if format-type[2:0] = 010 (I type):
194           imm[31:0] = Inst[31]*ones[19:0] : Inst[30:20]
195           is_imm_valid = 1
196          if format-type[2:0] = 011 (S type):
197           imm[31:0] = Inst[31]*ones[19:0] : Inst[30:25] : Inst[11:7]
198           is_imm_valid = 1
199          if format-type[2:0] = 100 (B type):
200           imm[31:0] = Inst[31]*ones[19:0] : Inst[7] : Inst[30:25] : Inst
    [11:8] : 0
```

```
201          is_imm_valid = 1
202         if format-type[2:0] = 101 (U type):
203          imm[31:0] = Inst[31:12] : zeros[11:0]
204          is_imm_valid = 1
205         if format-type[2:0] = 110 (J type):
206          imm[31:0] = Inst[31]*ones[11:0] : Inst[19:12] : Inst[20] :
    Inst[30:21] : 0
207          is_imm_valid = 1
208         else:
209          imm[31:0] = dont_care[31:0]
210          is_imm_valid = 0
211     */
212     assign is_imm_valid = (format_type[2:1]==2'b01|format_type[2:1]==2'
    b10|format_type==3'b110) ? 1'b1 : 1'b0;
213     assign imm = (format_type==3'b010) ? 32'(signed'(in[31:20]))
                                        : // I
214                  ((format_type==3'b011) ? 32'(signed'({in[31:25], in
    [11:7]}))                           : // S
215                  ((format_type==3'b100) ? 32'(signed'({in[31], in[7], in
    [30:25], in[11:8], 1'b0}))    : // B
216                  ((format_type==3'b101) ? {in[31:12], 12'b0}
                                        : // U
217                  ((format_type==3'b110) ? 32'(signed'({in[31], in
    [19:12], in[20], in[30:21], 1'b0})) : // J
218                   32'bxxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx))));
219 endmodule
220
221 module registers_files(
222     input clk,              // clock del sistema
223     input [4:0] rs1_add,    // direcci n reg source 1
224     input [4:0] rs2_add,    // direcci n reg source 2
225     input [4:0] rs3_add,    // direcci n reg source 3
226     input [4:0] wb_add,     // direcci n reg para wb
227     input [31:0] wb_data,   // data para wb
228     input write_reg,        // pin que indica si realizar wb
229     input is_wb_data_fp,    // indica si el reg de wb es fp o no
230     input is_rs1_fp,        // indica si el reg source 1 es fp o no
231     input is_rs2_fp,        // indica si el reg source 2 es fp o no
232     input is_storeConditional_WB,  //indica si el acceso a reg es un sc
    .w en wb
233     input is_loadReserved_WB,      //indica si el acceso a reg es un lr
    .w en wb
234     input isAMO_wb,           //indica si el acceso a reg es AMO en wb
235     input [4:0] funt_5_wb,    // funct 5 wb
236     input [31:0] wb_rs1,      // data rs1 wb
237     input [31:0] wb_rs2,      // data rs2 wb
238     output [31:0] rs1,        // data de reg source 1
239     output [31:0] rs2,        // data de reg source 2
240     output [31:0] rs3,        // data de reg source 3
241     output store_conditions_out
```

```
242     );
243     reg[31:0] int_registers[31:0];
244     reg[31:0] fp_registers[31:0];
245     reg[31:0] atomic_register_lr;           //Registro para guardar el
    ultimo valor de memoria reservado
246     reg[31:0] atomic_addr_lr;               //registro del ultimo
    address de memoria reservado
247
248
249     wire store_conditions;
250
251     initial for(int i=0; i<32; i++) begin
252         if(i == 2)      {int_registers[i], fp_registers[i]} = {32'h2FFC
    , 32'b0}; // sp
253         else if(i == 3) {int_registers[i], fp_registers[i]} = {32'h1800
    , 32'b0}; // gp
254         else if(i == 32) {int_registers[i], fp_registers[i],
    atomic_register_lr ,atomic_addr_lr} = '0;
255         else            {int_registers[i], fp_registers[i]} = '0;
256     end
257
258
259     assign store_conditions= (is_storeConditional_WB & atomic_addr_lr==
    wb_rs1 & atomic_register_lr==wb_data)? 1'b1 : 1'b0;  //
    ComentarioAtomic
260
261     assign store_conditions_out = (store_conditions | isAMO_wb);
262
263     assign rs1 = is_rs1_fp ? fp_registers[rs1_add] : (
    store_conditions_out? wb_rs1 :int_registers[rs1_add]);
      //ComentarioAtomic
264
265     assign rs2 = is_rs2_fp ? fp_registers[rs2_add] :
266                 ((store_conditions | (isAMO_wb & funt_5_wb[0] == 1'b1))
    ? wb_rs2 :
267                 ((isAMO_wb & funt_5_wb == 5'b00000)     ? wb_rs2 +
    wb_data :
268                 ((isAMO_wb & funt_5_wb[4:2] == 3'b001)  ? wb_rs2 ^
    wb_data :
269                 ((isAMO_wb & funt_5_wb[4:2] == 3'b011)  ? wb_rs2 &
    wb_data :
270                 ((isAMO_wb & funt_5_wb[4:2] == 3'b010)  ? wb_rs2 |
    wb_data :
271                 ((isAMO_wb & funt_5_wb[4:2] == 3'b100)  ? ((signed'(
    wb_rs2) <signed'(wb_data)) ? wb_rs2: wb_data):
272                 ((isAMO_wb & funt_5_wb[4:2] == 3'b101)  ? ((signed'(
    wb_rs2) <signed'(wb_data)) ? wb_data: wb_rs2):
273                 ((isAMO_wb & funt_5_wb[4:2] == 3'b110)  ? ((unsigned'(
    wb_rs2) <unsigned'(wb_data)) ? wb_rs2: wb_data) :
```

```
274                 ((isAMO_wb & funt_5_wb[4:2] == 3'b111)  ? ((unsigned'(
    wb_rs2) <unsigned'(wb_data)) ? wb_data: wb_rs2) :
275                 int_registers[rs2_add])))))))));                //
    ComentarioAtomic
276
277
278     assign rs3 = fp_registers[rs3_add];
279
280     always @ (negedge clk)
281         if(write_reg)
282             if(is_wb_data_fp) fp_registers[wb_add] <= wb_data;
283             else begin
284                 if (is_loadReserved_WB) begin
285
286                     {int_registers[wb_add], atomic_register_lr,
    atomic_addr_lr}
287                     = {((wb_add == 5'd0) ? 32'd0 : wb_data), ((wb_add
    == 5'd0) ? 32'd0 : wb_data),  wb_rs1};
288
289                 end
290                 else if(is_storeConditional_WB) begin
291
292                     if(store_conditions) int_registers[wb_add] <= 32'b0
    ;
293                     else int_registers[wb_add] <= 32'
    b01010101010101010101010101010101;
294                     {atomic_register_lr, atomic_addr_lr}
295                     ={32'b11111111111111111111111111111111,32'
    b11111111111111111111111111111111};
296                     //else if(atomic_addr_lr==wb_rs1) int_registers[
    wb_add] <= 32'b10101010101010101010101010101010;
297                 end
298                 else int_registers[wb_add] <= (wb_add == 5'd0) ? 32'd0
    : wb_data;
299             end
300 endmodule
301
302 module checkAtomic(
303
304     input [2:0] format_type, sub_format_type,
305     input [4:0] funct5,
306     output is_storeConditional, is_loadReserved, is_Atomic, is_AMO
307 );
308
309     wire isAtomic_w;
310
311     assign isAtomic_w   = (format_type == 3'b000 & sub_format_type ==
    3'b010) ? 1'b1 : 1'b0;
312
```

```
313     assign is_AMO          = (isAtomic_w  & funct5[1] == 1'b0) ? 1'b1: 1'
        b0;
314     assign is_loadReserved      =   ((isAtomic_w)&(funct5[1:0] ==2'b10)
        )?  1'b1 : 1'b0;
315     assign is_storeConditional  =   ((isAtomic_w)&(funct5[1:0] ==2'b11)
        )?  1'b1 : 1'b0;
316     assign is_Atomic            =   isAtomic_w;
317
318 endmodule
```

# Anexo D

Archivo *TOP (SoC)* modificado para incorporar las instrucciones atómicas en el Soc.

```
/*
autor: Gianluca Vincenzo D'Agostino Matute
Santiago de Chile, Septiembre 2021
*/
`timescale 1ns / 1ps
// set_param pwropt.maxFaninFanoutToNetRatio 1000
module TOP(
    input clk, rst_asin, start_asin, resume_asin, // rst, start, resume deben ser pulsadores (OJO, en todo el resto del dise o start se comporta de otra forma)
    input [15:0] in,
    output [5:0] color_leds,
    output [15:0] leds, display
    );
    typedef enum {
        waiting_state,
        final_state_error_prog, final_state_error_data, start_cpu_state,
        ebreak_state, ecall_state, ecall_default_state,
        ecall_print_integer_state, ecall_print_fp_state,
        ecall_get_integer_state_1, ecall_get_integer_state_2, ecall_get_integer_state_3,
        ecall_get_fp_state_1, ecall_get_fp_state_2, ecall_get_fp_state_3,
        final_state
    } state_type;
    state_type actual_state, next_state;
    reg rst_PC, set_PC, set_first_input_half, set_second_input_half,
        start_to_cpu, sel_info_out, enable_info_out,
        acces_to_registers_files, is_fp, do_wb_fromEEI, rst_riscv;
    reg [3:0] info_state;
    reg [1:0] input_indicator;
    /*
        input_indicator =
            00 o 11 - no input indicator
            01 - first half indicator
            10 - second half indicator
        exit_status[1:0] =
            00 -> ecall     (se lee registro x17 y seg n eso se toma decici n)
            11 -> ebreak    (pausa ejecuci n)
            01 -> program out of range
            10 -> memory out of range
    */
```

```
39      wire ready, prog_ready_toEEI, prog_out_of_range_toEEI,
    data_ready_toEEI, data_out_of_range_toEEI, rst, start, resume;
40      wire [1:0] exit_status;
41      wire [15:0] second_input_half, first_input_half;
42      wire [31:0] prog_out_toEEI, data_out_toEEI, rs1_toEEI, rs2_toEEI,
    PC, PC_reg, info_out, input_from_user;
43      assign info_out = enable_info_out ? (sel_info_out ? rs2_toEEI :
    PC_reg) : 32'b0;
44      assign input_from_user = {second_input_half, first_input_half};
45      assign leds = (input_indicator[1]^input_indicator[0]) ? in : 16'b0;
46      // color_leds_decoder
47      assign color_leds =  (input_indicator==2'b01 & info_state==4'h8) ?
    6'b000_101 : // get integer (PURPLE)
48                           ((input_indicator==2'b10 & info_state==4'h8) ?
    6'b101_000 : // get integer (PURPLE)
49                           ((input_indicator==2'b01 & info_state==4'h9) ?
    6'b000_110 : // get fp (YELLOW)
50                           ((input_indicator==2'b10 & info_state==4'h9) ?
    6'b110_000 : // get fp (YELLOW)
51                           ((info_state==4'h0) ? 6'b001_001 : // A -> --B
    |--B = BLUE      - waiting state
52                           ((info_state==4'h2) ? 6'b011_011 : // B -> -GB|-
    GB = CYAN      - cpu working
53                           ((info_state==4'h3) ? 6'b111_111 : // C -> RGB|
    RGB = WHITE     - ebreak
54                           ((info_state==4'h4) ? 6'b100_001 : // D -> R
    --|--B = RED/BLUE - prog error
55                           ((info_state==4'h5) ? 6'b001_100 : // E -> --B|R
    -- = BLUE/RED - data error
56                           ((info_state==4'h6) ? 6'b101_101 : // F -> R-B|R
    -B = PURPLE    - print integer
57                           ((info_state==4'h7) ? 6'b110_110 : // G -> RG-|
    RG- = YELLOW    - print fp
58                           ((info_state==4'h8) ? 6'b000_000 : // H ->
    ---|--- = NO COLOR - get integer
59                           ((info_state==4'h9) ? 6'b000_000 : // I ->
    ---|--- = NO COLOR - get fp
60                           ((info_state==4'hA) ? 6'b010_010 : // J -> -G-|-
    G- = GREEN     - final state
61                           ((info_state==4'hB) ? 6'b100_100 : // K -> R--|R
    -- = RED       - ecall default
62                             6'b000_000))))))))))));//rgb_rgb
63      // FSM controller
64      always_ff @(posedge(clk)) actual_state <= rst ? waiting_state :
    next_state;
65      always_comb
66          case (actual_state)
67              default: begin
68                  {input_indicator, info_state}                    = 6'
    b00_0000; // A
```

```
69                  {set_first_input_half, set_second_input_half}    = 2'
    b00;
70                  {sel_info_out, enable_info_out}                 = 2'
    b00;
71                  {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b000;
72                  {rst_riscv, start_to_cpu, set_PC, rst_PC}       = 4'
    b00_00;
73                  next_state = actual_state;
74              end
75              waiting_state: begin
76                  {input_indicator, info_state}                   = 6'
    b00_0000; // A
77                  {set_first_input_half, set_second_input_half}    = 2'
    b00;
78                  {sel_info_out, enable_info_out}                 = 2'
    b00;
79                  {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b000;
80                  {rst_riscv, start_to_cpu, set_PC, rst_PC}       = 4'
    b10_01;
81                  next_state = start ? start_cpu_state : waiting_state;
82              end
83              start_cpu_state: begin
84                  {input_indicator, info_state}                   = 6'
    b00_0010; // B
85                  {set_first_input_half, set_second_input_half}    = 2'
    b00;
86                  {sel_info_out, enable_info_out}                 = 2'
    b00;
87                  {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b000;
88                  {rst_riscv, start_to_cpu, rst_PC} = 3'b01_0; set_PC =
    ready;
89                  next_state = ready ? ((exit_status == 2'b11) ?
    ebreak_state :
90                                       ((exit_status == 2'b01) ?
    final_state_error_prog :
91                                       ((exit_status == 2'b10) ?
    final_state_error_data :
92                                       ecall_state))) : start_cpu_state;
93              end
94              ebreak_state: begin
95                  {input_indicator, info_state}                   = 6'
    b00_0011; // C
96                  {set_first_input_half, set_second_input_half}    = 2'
    b00;
97                  {sel_info_out, enable_info_out}                 = 2'
    b01;
```

```
98                  {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b000;
99                  {rst_riscv, start_to_cpu, set_PC, rst_PC}        = {
    resume, 3'b0_00};
100                 next_state = resume ? start_cpu_state : ebreak_state;
101             end
102             final_state_error_prog: begin
103                 {input_indicator, info_state}                    = 6'
    b00_0100; // D
104                 {set_first_input_half, set_second_input_half}    = 2'
    b00;
105                 {sel_info_out, enable_info_out}                  = 2'
    b01;
106                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b000;
107                 {rst_riscv, start_to_cpu, set_PC, rst_PC}        = 4'
    b00_00;
108                 next_state = resume ? waiting_state :
    final_state_error_prog;
109             end
110             final_state_error_data: begin
111                 {input_indicator, info_state}                    = 6'
    b00_0101; // E
112                 {set_first_input_half, set_second_input_half}    = 2'
    b00;
113                 {sel_info_out, enable_info_out}                  = 2'
    b01;
114                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b000;
115                 {rst_riscv, start_to_cpu, set_PC, rst_PC}        = 4'
    b00_00;
116                 next_state = resume ? waiting_state :
    final_state_error_data;
117             end
118             ecall_print_integer_state: begin
119                 {input_indicator, info_state}                    = 6'
    b00_0110; // F
120                 {set_first_input_half, set_second_input_half}    = 2'
    b00;
121                 {sel_info_out, enable_info_out}                  = 2'
    b11;
122                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b100;
123                 {rst_riscv, start_to_cpu, set_PC, rst_PC}        = {
    resume, 3'b0_00};
124                 next_state = resume ? start_cpu_state :
    ecall_print_integer_state;
125             end
126             ecall_print_fp_state: begin
```

```
127                 {input_indicator, info_state}                       = 6'
    b00_0111; // G
128                 {set_first_input_half, set_second_input_half}    = 2'
    b00;
129                 {sel_info_out, enable_info_out}                     = 2'
    b11;
130                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b110;
131                 {rst_riscv, start_to_cpu, set_PC, rst_PC}           = {
    resume, 3'b0_00};
132                 next_state = resume ? start_cpu_state :
    ecall_print_fp_state;
133             end
134             ecall_get_integer_state_1: begin
135                 {input_indicator, info_state}                       = 6'
    b01_1000; // H
136                 {set_first_input_half, set_second_input_half}    = 2'
    b10;
137                 {sel_info_out, enable_info_out}                     = 2'
    b00;
138                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b100;
139                 {rst_riscv, start_to_cpu, set_PC, rst_PC}           = 4'
    b00_00;
140                 next_state = start ? ecall_get_integer_state_2 :
    ecall_get_integer_state_1;
141             end
142             ecall_get_integer_state_2: begin
143                 {input_indicator, info_state}                       = 6'
    b10_1000; // H
144                 {set_first_input_half, set_second_input_half}    = 2'
    b01;
145                 {sel_info_out, enable_info_out}                     = 2'
    b00;
146                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b100;
147                 {rst_riscv, start_to_cpu, set_PC, rst_PC}           = 4'
    b00_00;
148                 next_state = resume ? ecall_get_integer_state_3 :
    ecall_get_integer_state_2;
149             end
150             ecall_get_integer_state_3: begin
151                 {input_indicator, info_state}                       = 6'
    b00_1000; // H
152                 {set_first_input_half, set_second_input_half}    = 2'
    b00;
153                 {sel_info_out, enable_info_out}                     = 2'
    b00;
154                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b101;
```

```
155                 {rst_riscv, start_to_cpu, set_PC, rst_PC}          = 4'
    b10_00;
156                 next_state = start_cpu_state;
157             end
158             ecall_get_fp_state_1: begin
159                 {input_indicator, info_state}                      = 6'
    b01_1001; // I
160                 {set_first_input_half, set_second_input_half}      = 2'
    b10;
161                 {sel_info_out, enable_info_out}                    = 2'
    b00;
162                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b110;
163                 {rst_riscv, start_to_cpu, set_PC, rst_PC}          = 4'
    b00_00;
164                 next_state = start ? ecall_get_fp_state_2 :
    ecall_get_fp_state_1;
165             end
166             ecall_get_fp_state_2: begin
167                 {input_indicator, info_state}                      = 6'
    b10_1001; // I
168                 {set_first_input_half, set_second_input_half}      = 2'
    b01;
169                 {sel_info_out, enable_info_out}                    = 2'
    b00;
170                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b110;
171                 {rst_riscv, start_to_cpu, set_PC, rst_PC}          = 4'
    b00_00;
172                 next_state = resume ? ecall_get_fp_state_3 :
    ecall_get_fp_state_2;
173             end
174             ecall_get_fp_state_3: begin
175                 {input_indicator, info_state}                      = 6'
    b00_1001; // I
176                 {set_first_input_half, set_second_input_half}      = 2'
    b00;
177                 {sel_info_out, enable_info_out}                    = 2'
    b00;
178                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b111;
179                 {rst_riscv, start_to_cpu, set_PC, rst_PC}          = 4'
    b10_00;
180                 next_state = start_cpu_state;
181             end
182             final_state: begin
183                 {input_indicator, info_state}                      = 6'
    b00_1010; // J
184                 {set_first_input_half, set_second_input_half}      = 2'
    b00;
```

```
185                 {sel_info_out, enable_info_out}                    = 2'
    b01;
186                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b000;
187                 {rst_riscv, start_to_cpu, set_PC, rst_PC}          = 4'
    b00_00;
188                 next_state = resume ? waiting_state : final_state;
189             end
190             ecall_state: begin
191                 {input_indicator, info_state}                      = 6'
    b00_1011; // K
192                 {set_first_input_half, set_second_input_half}      = 2'
    b00;
193                 {sel_info_out, enable_info_out}                    = 2'
    b00;
194                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b100;
195                 {rst_riscv, start_to_cpu, set_PC, rst_PC}          = 4'
    b00_00;
196                 next_state = (rs1_toEEI==31'b01010) ? final_state :
197                              ((rs1_toEEI==31'b00001) ?
    ecall_print_integer_state :
198                              ((rs1_toEEI==31'b00010) ?
    ecall_print_fp_state :
199                              ((rs1_toEEI==31'b00101) ?
    ecall_get_integer_state_1 :
200                              ((rs1_toEEI==31'b00110) ?
    ecall_get_fp_state_1 :
201                                  ecall_default_state))));
202             end
203             ecall_default_state: begin
204                 {input_indicator, info_state}                      = 6'
    b00_1011; // K
205                 {set_first_input_half, set_second_input_half}      = 2'
    b00;
206                 {sel_info_out, enable_info_out}                    = 2'
    b01;
207                 {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'
    b000;
208                 {rst_riscv, start_to_cpu, set_PC, rst_PC}          = {
    resume, 3'b0_00};
209                 next_state = resume ? start_cpu_state :
    ecall_default_state;
210             end
211         endcase
212     //riscv32imf_singlecycle
213     riscv32imf_pipeline
214     riscv32imf(
215         // inputs
```

```
216         .clk(clk), .rst(rst | rst_riscv), .start(start_to_cpu), .
    acces_to_registers_files(acces_to_registers_files),
217         .is_wb_data_fp_fromEEI(is_fp), .do_wb_fromEEI(do_wb_fromEEI),
218         .is_rs1_fp_fromEEI(1'b0), .is_rs2_fp_fromEEI(is_fp),
219         .acces_to_prog_mem(1'b0), .prog_rw_fromEEI(1'b0),
220         .prog_valid_mem_fromEEI(1'b0), .prog_is_load_unsigned_fromEEI
    (1'b0),
221         .acces_to_data_mem(1'b0), .data_rw_fromEEI(1'b0),
222         .data_valid_mem_fromEEI(1'b0), .data_is_load_unsigned_fromEEI
    (1'b0),
223         .initial_PC(PC_reg),
224         .prog_addr_fromEEI(32'b0), .prog_in_fromEEI(32'b0),
225         .data_addr_fromEEI(32'b0), .data_in_fromEEI(32'b0),
226         .wb_data_fromEEI(input_from_user),
227         .prog_byte_half_word_fromEEI(2'b00), .
    data_byte_half_word_fromEEI(2'b00),
228         .rs1_add_fromEEI(5'b10001), .rs2_add_fromEEI(5'b01010), .
    wb_add_fromEEI(5'b01010),
229         // outputs
230         .ready(ready),
231         .prog_ready_toEEI(prog_ready_toEEI), .prog_out_of_range_toEEI(
    prog_out_of_range_toEEI),
232         .data_ready_toEEI(data_ready_toEEI), .data_out_of_range_toEEI(
    data_out_of_range_toEEI),
233         .exit_status(exit_status),
234         .prog_out_toEEI(prog_out_toEEI), .data_out_toEEI(data_out_toEEI
    ), .rs1_toEEI(rs1_toEEI), .rs2_toEEI(rs2_toEEI), .PC(PC)
235     );
236     generic_register #(.width(32)) PC_in_reg(
237         .clk(clk), .reset(rst_PC), .load(set_PC), .data_in(PC+32'b100),
238         .data_out(PC_reg)
239     );
240     generic_register #(.width(16)) first_input_half_reg(
241         .clk(clk), .reset(1'b0), .load(set_first_input_half), .data_in(
    in),
242         .data_out(first_input_half)
243     );
244     generic_register #(.width(16)) second_input_half_reg(
245         .clk(clk), .reset(1'b0), .load(set_second_input_half), .data_in
    (in),
246         .data_out(second_input_half)
247     );
248     debounce rst_debounce   (.clk(clk), .signal_in(~rst_asin),    .
    signal_out(rst));
249     debounce start_debounce (.clk(clk), .signal_in(start_asin),   .
    signal_out(start));
250     debounce resume_debounce(.clk(clk), .signal_in(resume_asin), .
    signal_out(resume));
251     full_display full_display_unit(
```

```
252         .clk(clk), .enable( (input_indicator[1]^input_indicator[0]) |
    enable_info_out ),
253         .show_options(input_indicator), .in( (input_indicator[1]^
    input_indicator[0]) ? {in, in} : info_out ),
254         .out(display)
255     );
256 endmodule
257
258 module generic_register #(parameter width=32)(
259     input clk, reset, load,
260     input [width-1:0] data_in,
261     output reg[width-1:0] data_out
262     );
263     always_ff @(negedge clk) data_out <= load ? data_in : (reset ? 0 :
    data_out);
264 endmodule
265
266 module debounce(input clk, signal_in, output reg signal_out);
267     typedef enum {
268         waiting_state, counter_state, hold_state, response_state
269     } state_type;
270     state_type actual_state, next_state;
271     reg signal_in_sync, start_counter, counter_ready;
272     reg [15:0] counter;
273     always_ff @(posedge clk) {signal_in_sync, actual_state} <= {
    signal_in, next_state}; // Se al de entrada sincronizada y
    actualizaci n de estado
274     always_ff @(posedge clk) // debounce counter
275         if(start_counter) {counter, counter_ready} <= {counter + 16'b1,
     (counter == 16'hFFFF) ? 1'b1 : 1'b0};
276         else              {counter, counter_ready} <= 17'b0;
277     always_comb // FSM controller
278         case (actual_state)
279             default: begin
280                 {start_counter, signal_out} = 2'b00;
281                 next_state = actual_state;
282             end
283             waiting_state: begin
284                 {start_counter, signal_out} = 2'b00;
285                 next_state = signal_in_sync ? counter_state :
    waiting_state;
286             end
287             counter_state: begin
288                 {start_counter, signal_out} = 2'b10;
289                 next_state = signal_in_sync ? (counter_ready ?
    hold_state : counter_state) : waiting_state;
290             end
291             hold_state: begin
292                 {start_counter, signal_out} = 2'b00;
```

```
293                 next_state = signal_in_sync ? hold_state :
    response_state;
294             end
295             response_state: begin
296                 {start_counter, signal_out} = 2'b01;
297                 next_state = waiting_state;
298             end
299         endcase
300 endmodule
301
302 module full_display(
303     input clk, enable,
304     input [1:0] show_options,
305     input [31:0] in,
306     output [15:0] out // enable7, enable6, enable5, enable4, enable3,
    enable2, enable1, enable0, CA, CB, CC, CD, CE, CF, CG, DP
307     );
308     /*
309         show_options =
310                 00 o 11 -> full: 8 bytes
311                 01      -> half: first 4 bytes
312                 10      -> half: last 4 bytes
313     */
314     reg [2:0] select;
315     reg [15:0] counter;
316     wire show_first_half, show_last_half;
317     wire [7:0] character0, character1, character2, character3,
    character4, character5, character6, character7;
318     assign {show_first_half, show_last_half} = {(show_options==2'b10) ?
    1'b0 : 1'b1, (show_options==2'b01) ? 1'b0 : 1'b1};
319     assign out = enable ?
320                 ((select==3'h0) ? {show_first_half ? 8'b11111110 : 8'
    hFF, character0} :
321                 ((select==3'h1) ? {show_first_half ? 8'b11111101 : 8'
    hFF, character1} :
322                 ((select==3'h2) ? {show_first_half ? 8'b11111011 : 8'
    hFF, character2} :
323                 ((select==3'h3) ? {show_first_half ? 8'b11110111 : 8'
    hFF, character3} :
324                 ((select==3'h4) ? {show_last_half  ? 8'b11101111 : 8'
    hFF, character4} :
325                 ((select==3'h5) ? {show_last_half  ? 8'b11011111 : 8'
    hFF, character5} :
326                 ((select==3'h6) ? {show_last_half  ? 8'b10111111 : 8'
    hFF, character6} :
327                 ((select==3'h7) ? {show_last_half  ? 8'b01111111 : 8'
    hFF, character7} :
328                     16'hFFFF))))))) : 16'hFFFF;
329     always_ff @(posedge clk)
330         if(enable) begin
```

```
331             if(counter==16'hFFFF) {select, counter} = { unsigned'(
    select)+3'b1, 16'b0 };
332             else {select, counter} = { select, unsigned'(counter)+16'b1
    };
333         end
334         else {select, counter} = 0;
335     display_character display_character_unit0(
336         .in(in[ 3: 0]), .out(character0) // CA, CB, CC, CD, CE, CF, CG,
    DP
337     );
338     display_character display_character_unit1(
339         .in(in[ 7: 4]), .out(character1) // CA, CB, CC, CD, CE, CF, CG,
    DP
340     );
341     display_character display_character_unit2(
342         .in(in[11: 8]), .out(character2) // CA, CB, CC, CD, CE, CF, CG,
    DP
343     );
344     display_character display_character_unit3(
345         .in(in[15:12]), .out(character3) // CA, CB, CC, CD, CE, CF, CG,
    DP
346     );
347     display_character display_character_unit4(
348         .in(in[19:16]), .out(character4) // CA, CB, CC, CD, CE, CF, CG,
    DP
349     );
350     display_character display_character_unit5(
351         .in(in[23:20]), .out(character5) // CA, CB, CC, CD, CE, CF, CG,
    DP
352     );
353     display_character display_character_unit6(
354         .in(in[27:24]), .out(character6) // CA, CB, CC, CD, CE, CF, CG,
    DP
355     );
356     display_character display_character_unit7(
357         .in(in[31:28]), .out(character7) // CA, CB, CC, CD, CE, CF, CG,
    DP
358     );
359 endmodule
360
361 module display_character(
362     input [3:0] in,
363     output [7:0] out // CA, CB, CC, CD, CE, CF, CG, DP
364     ); // Dot On (0 l gico) => letra | Dot Off (1 l gico) => n mero
365     assign out = (in==4'h0) ? 8'b00000011 : // 0
366                  ((in==4'h1) ? 8'b10011111 : // 1
367                  ((in==4'h2) ? 8'b00100101 : // 2
368                  ((in==4'h3) ? 8'b00001101 : // 3
369                  ((in==4'h4) ? 8'b10011001 : // 4
370                  ((in==4'h5) ? 8'b01001001 : // 5
```

```
371                 ((in==4'h6) ? 8'b01000001 : // 6
372                 ((in==4'h7) ? 8'b00011111 : // 7
373                 ((in==4'h8) ? 8'b00000001 : // 8
374                 ((in==4'h9) ? 8'b00001001 : // 9
375                 ((in==4'hA) ? 8'b00010000 : // A
376                 ((in==4'hB) ? 8'b11000000 : // B
377                 ((in==4'hC) ? 8'b01100010 : // C
378                 ((in==4'hD) ? 8'b10000100 : // D
379                 ((in==4'hE) ? 8'b01100000 : // E
380                 ((in==4'hF) ? 8'b01110000 : // F
381                 7'b1111111))))))))))))))); // default
382 endmodule
```

# Anexo D

Código *Test Bench* para la verificación del SoC.

```
/*
autor: Gianluca Vincenzo D'Agostino Matute
Santiago de Chile, Septiembre 2021

para: testing_code_imf.s
*/
`timescale 1ns / 1ps

module tb_riscv32imf_top;
    reg clk, rst, start, acces_to_registers_files, is_wb_data_fp_fromEEI, do_wb_fromEEI, is_rs1_fp_fromEEI, is_rs2_fp_fromEEI;
    reg acces_to_prog_mem, prog_rw_fromEEI, prog_valid_mem_fromEEI, prog_is_load_unsigned_fromEEI;
    reg acces_to_data_mem, data_rw_fromEEI, data_valid_mem_fromEEI, data_is_load_unsigned_fromEEI;
    reg [31:0] initial_PC, prog_addr_fromEEI, prog_in_fromEEI, data_addr_fromEEI, data_in_fromEEI, wb_data_fromEEI;
    reg [1:0] prog_byte_half_word_fromEEI, data_byte_half_word_fromEEI;
    reg [4:0] rs1_add_fromEEI, rs2_add_fromEEI, wb_add_fromEEI;
    wire ready, prog_ready_toEEI, prog_out_of_range_toEEI, data_ready_toEEI, data_out_of_range_toEEI;
    wire [1:0] exit_status;
    wire [31:0] prog_out_toEEI, data_out_toEEI, rs1_toEEI, rs2_toEEI, PC;
    //riscv32imf_singlecycle
    riscv32imf_pipeline
    riscv32imf(
        // inputs
        .clk(clk), .rst(rst), .start(start), .acces_to_registers_files(acces_to_registers_files),
        .is_wb_data_fp_fromEEI(is_wb_data_fp_fromEEI), .do_wb_fromEEI(do_wb_fromEEI),
        .is_rs1_fp_fromEEI(is_rs1_fp_fromEEI), .is_rs2_fp_fromEEI(is_rs2_fp_fromEEI),
        .acces_to_prog_mem(acces_to_prog_mem), .prog_rw_fromEEI(prog_rw_fromEEI),
        .prog_valid_mem_fromEEI(prog_valid_mem_fromEEI), .prog_is_load_unsigned_fromEEI(prog_is_load_unsigned_fromEEI),
        .acces_to_data_mem(acces_to_data_mem), .data_rw_fromEEI(data_rw_fromEEI),
        .data_valid_mem_fromEEI(data_valid_mem_fromEEI), .data_is_load_unsigned_fromEEI(data_is_load_unsigned_fromEEI),
        .initial_PC(initial_PC),
```

```
31          .prog_addr_fromEEI(prog_addr_fromEEI), .prog_in_fromEEI(
    prog_in_fromEEI),
32          .data_addr_fromEEI(data_addr_fromEEI), .data_in_fromEEI(
    data_in_fromEEI), .wb_data_fromEEI(wb_data_fromEEI),
33          .prog_byte_half_word_fromEEI(prog_byte_half_word_fromEEI), .
    data_byte_half_word_fromEEI(data_byte_half_word_fromEEI),
34          .rs1_add_fromEEI(rs1_add_fromEEI), .rs2_add_fromEEI(
    rs2_add_fromEEI), .wb_add_fromEEI(wb_add_fromEEI),
35          // outputs
36          .ready(ready),
37          .prog_ready_toEEI(prog_ready_toEEI), .prog_out_of_range_toEEI(
    prog_out_of_range_toEEI),
38          .data_ready_toEEI(data_ready_toEEI), .data_out_of_range_toEEI(
    data_out_of_range_toEEI),
39          .exit_status(exit_status),
40          .prog_out_toEEI(prog_out_toEEI), .data_out_toEEI(data_out_toEEI
    ), .rs1_toEEI(rs1_toEEI), .rs2_toEEI(rs2_toEEI), .PC(PC)
41      );
42      always #5 clk = ~clk;
43      initial begin
44          int prog_out_file, data_out_file, registers_file;
45          string buff_str;
46          {clk, rst, start, acces_to_registers_files,
    is_wb_data_fp_fromEEI, do_wb_fromEEI, is_rs1_fp_fromEEI,
    is_rs2_fp_fromEEI,
47          acces_to_prog_mem, prog_rw_fromEEI, prog_valid_mem_fromEEI,
    prog_is_load_unsigned_fromEEI,
48          acces_to_data_mem, data_rw_fromEEI, data_valid_mem_fromEEI,
    data_is_load_unsigned_fromEEI,
49          initial_PC, prog_addr_fromEEI, prog_in_fromEEI,
    data_addr_fromEEI, data_in_fromEEI, wb_data_fromEEI,
50          prog_byte_half_word_fromEEI, data_byte_half_word_fromEEI,
    rs1_add_fromEEI, rs2_add_fromEEI, wb_add_fromEEI} = 0;
51          $display("\nBEGIN\n");
52          prog_out_file = $fopen("prog_out.mem","w");
53          data_out_file = $fopen("data_out.mem","w");
54          registers_file = $fopen("registers.mem","w");
55          if(prog_out_file) $display("prog_out.mem - OK: %0d",
    prog_out_file); else $display("prog_out.mem - FAILED: %0d",
    prog_out_file);
56          if(data_out_file) $display("data_out.mem - OK: %0d",
    data_out_file); else $display("data_out.mem - FAILED: %0d",
    data_out_file);
57          if(registers_file) $display("registers.mem - OK: %0d",
    registers_file); else $display("registers.mem - FAILED: %0d",
    registers_file);
58          $display("\n");
59          // Se ejecuta el programa
60          #10 start = 1; while(~ready) #10;
61          $display("\nexit_status: %b\n", exit_status);
```

```
62          $display("\nPC: %h\n", PC+32'b100);
63          #10 start = 0;
64          // Se guarda el estado de la memoria de programa
65          acces_to_prog_mem = 1'b1;
66          while(1) begin
67              #10;
68              prog_valid_mem_fromEEI = 1'b1;
69              while(~prog_ready_toEEI & ~prog_out_of_range_toEEI) #10;
70              if(prog_out_of_range_toEEI == 1) break;
71              $fwriteh(prog_out_file, prog_addr_fromEEI); $fwrite(
    prog_out_file, ":  ");
72              $fwriteh(prog_out_file, prog_out_toEEI); $fwrite(
    prog_out_file, "\n");
73              #10;
74              prog_valid_mem_fromEEI = 1'b0;
75              prog_addr_fromEEI += 4;
76          end
77          {prog_addr_fromEEI, acces_to_prog_mem, prog_valid_mem_fromEEI}
    = 0;
78          // Se guarda el estado de la memoria de datos
79          acces_to_data_mem = 1'b1;
80          while(1) begin
81              #10;
82              data_valid_mem_fromEEI = 1'b1;
83              while(~data_ready_toEEI & ~data_out_of_range_toEEI) #10;
84              if(data_out_of_range_toEEI == 1) break;
85              $fwriteh(data_out_file, data_addr_fromEEI); $fwrite(
    data_out_file, ":  ");
86              $fwriteh(data_out_file, data_out_toEEI); $fwrite(
    data_out_file, "\n");
87              #10;
88              data_valid_mem_fromEEI = 1'b0;
89              data_addr_fromEEI += 4;
90          end
91          {data_addr_fromEEI, acces_to_data_mem, data_valid_mem_fromEEI}
    = 0;
92          // se guarda el estado del register files
93          acces_to_registers_files = 1'b1;
94          $fwrite(registers_file, "integer_regs\n");
95          {is_rs1_fp_fromEEI, is_rs2_fp_fromEEI} = 2'b00;
96          {rs1_add_fromEEI, rs2_add_fromEEI} = 10'b00000_00001;
97          for(int i = 0; i < 16; i++) begin
98              #10;
99              $fwriteh(registers_file, rs1_add_fromEEI); $fwrite(
    registers_file, ":   ");
100             $fwriteh(registers_file, rs1_toEEI); $fwrite(registers_file
    , "\n");
101             $fwriteh(registers_file, rs2_add_fromEEI); $fwrite(
    registers_file, ":   ");
```

```
102             $fwriteh(registers_file, rs2_toEEI); $fwrite(registers_file, "\n");
103             #10;
104             rs1_add_fromEEI += 5'b10;
105             rs2_add_fromEEI += 5'b10;
106             #10;
107         end
108         $fwrite(registers_file, "fp_regs\n");
109         {is_rs1_fp_fromEEI, is_rs2_fp_fromEEI} = 2'b11;
110         {rs1_add_fromEEI, rs2_add_fromEEI} = 10'b00000_00001;
111         for(int i = 0; i < 16; i++) begin
112             #10;
113             $fwriteh(registers_file, rs1_add_fromEEI); $fwrite(registers_file, ":    ");
114             $fwriteh(registers_file, rs1_toEEI); $fwrite(registers_file, "\n");
115             $fwriteh(registers_file, rs2_add_fromEEI); $fwrite(registers_file, ":    ");
116             $fwriteh(registers_file, rs2_toEEI); $fwrite(registers_file, "\n");
117             #10;
118             rs1_add_fromEEI += 5'b10;
119             rs2_add_fromEEI += 5'b10;
120             #10;
121         end
122         {rs1_add_fromEEI, rs2_add_fromEEI, is_rs1_fp_fromEEI, is_rs2_fp_fromEEI, acces_to_registers_files} = 0;
123         // fin
124         $display("\nEND\n");
125         $fclose(prog_out_file); $fclose(data_out_file); $fclose(registers_file); #10 $finish;
126     end
127 endmodule
```

# Anexo E

Código con funciones LR y SC y reserva valida, para la verificación del SoC.

```
addi    a1, x0, 1       # a1=1
addi    a2, x0, 4       # a2=4
sw      a1, 0(a2)       # se guarda en el espacio de memoria 4 el valor 1 (M[a2]=a1)

lr.w    a0, a2          #se carga el valor del espacio de memoria 4 en a0, tambi n se reserva ese espacio
sc.w    a0, a2, a2      #si el espacio sigue reservado, se guarda un 4 en el espacio de memoria y se carga un 0 en a0


addi x17, zero, 10      # terminar programa
ecall
```

# Anexo F

Código con funciones LR y SC y fallo en reserva, para la verificación del SoC.

```
#Para probar los 2 errores de SC.W, se debe usar el c digo como est , o descomentar las lineas sw a2, 0(a2) y
#sc.w a0, a2, a3. Y comentar la linea sc.w a0, a3, a2

addi    a1, x0, 1       # a1=1
addi    a2, x0, 4       # a2=4
addi    a3, x0, 5       # a3=5
sw      a1, 0(a2)       # se guarda en el espacio de memoria 4 el valor 1 (M[a2]=a1)

lr.w    a0, a2          #se carga el valor del espacio de memoria 4 en a0, tambi n se reserva ese espacio

#sw     a2, 0(a2)       # se guarda en el espacio de memoria 4 el valor 4 (M[a2]=a2)
#sc.w   a0, a2, a3      #sc erroneo por modificaci n de memoria

sc.w    a0, a3, a2      #sc erroneo modificaci n de direcci n

addi x17, zero, 10      # terminar programa
ecall
```

# Anexo G

Código con funciones AMO para la verificación del SoC.

## Prueba AMO

```
#Para probar los 2 errores de SC.W, se debe usar el c digo como est , o descomentar las lineas sw a2, 0(a2) y
#sc.w a0, a2, a3. Y comentar la linea sc.w a0, a3, a2

addi    a1, x0, 1       # a1=1
addi    a2, x0, 4       # a2=4
addi    a3, x0, 5       # a3=5
sw      a1, 0(a2)       # se guarda en el espacio de memoria 4 el valor 1 (M[a2]=a1)

lr.w    a0, a2          #se carga el valor del espacio de memoria 4 en a0, tambi n se reserva ese espacio

#sw     a2, 0(a2)       # se guarda en el espacio de memoria 4 el valor 4 (M[a2]=a2)
#sc.w   a0, a2, a3      #sc erroneo por modificaci n de memoria

sc.w    a0, a3, a2      #sc erroneo modificaci n de direcci n

addi x17, zero, 10      # terminar programa
ecall
```

# Anexo H

A continuación se explican los pasos para configurar el SoC en *Vivado* y el procedimiento para compilar y cargar programas en el mismo:

1. Descargar los archivos del *Github* del proyecto.
2. Instalar la librería para la tarjeta *FPGA Nexys A7*. En el repositorio de *Digilent*, *vivado-boards*, se hallan estas librerías junto con las instrucciones para su instalación en Vivado.
3. Luego, se abre *Vivado 2020.2* y haciendo clic en *Create Project*, en el apartado *Quick Start*, se comienza la creación de un nuevo proyecto. Después de oprimir *Next* en la primera ventana, se selecciona el nombre del proyecto y el directorio del mismo, tal como se aprecia en la figura

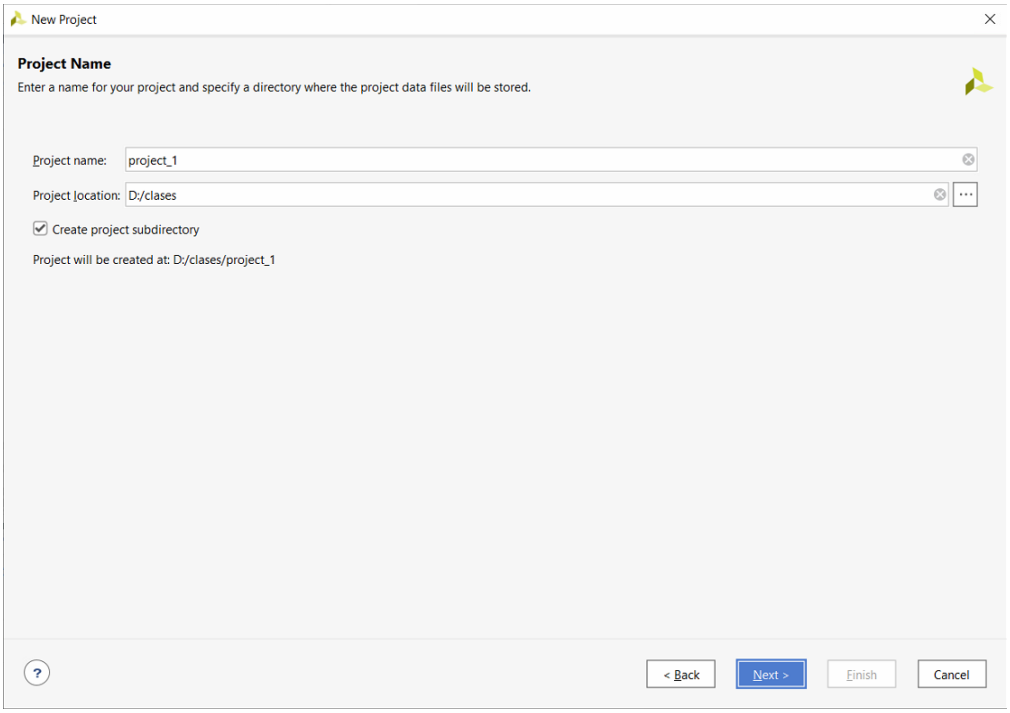


Figura 7.1: Configurando el nombre y directorio del nuevo proyecto de *Vivado*.

4. Se presiona *Next* y en la pestaña de tipo de proyecto se elige *RTL Proyect* y se aprieta en *Next*.
5. La siguiente ventana te permite subir los archivos del SoC. Para esto hay que presionar el botón *Add files*, el cual abrirá la ventana de la figura 7.2. Se deben elegir todos los archivos ubicados en la carpeta *sources* dentro del proyecto. Una vez elegidos todos los archivos, se presiona el botón *Ok*.
6. Al volver a la ventana de selección de archivos, se aprieta el botón *Next*, verificando antes que la única casilla marcada con un *ticket* es *Copy sources into project*. Figura 7.3.

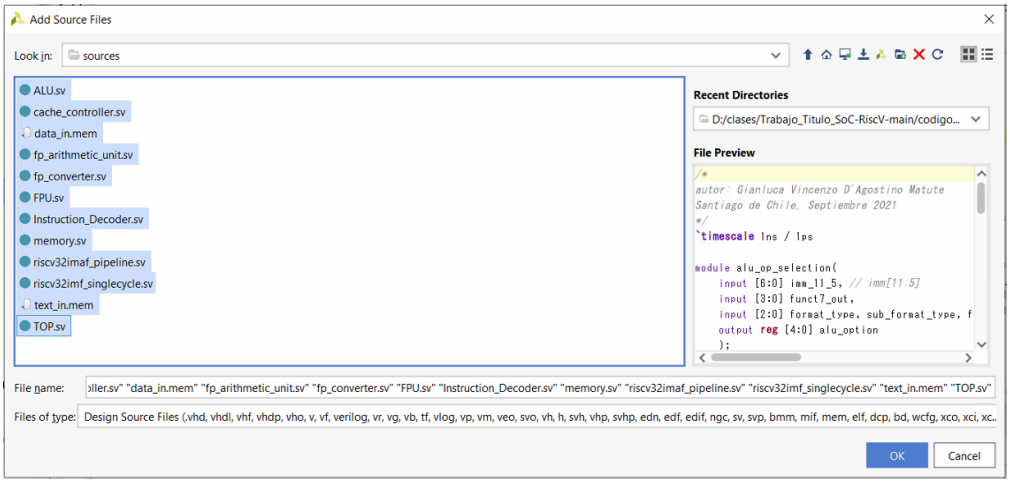


Figura 7.2: : Selección de los archivos a importar cómo código fuente.

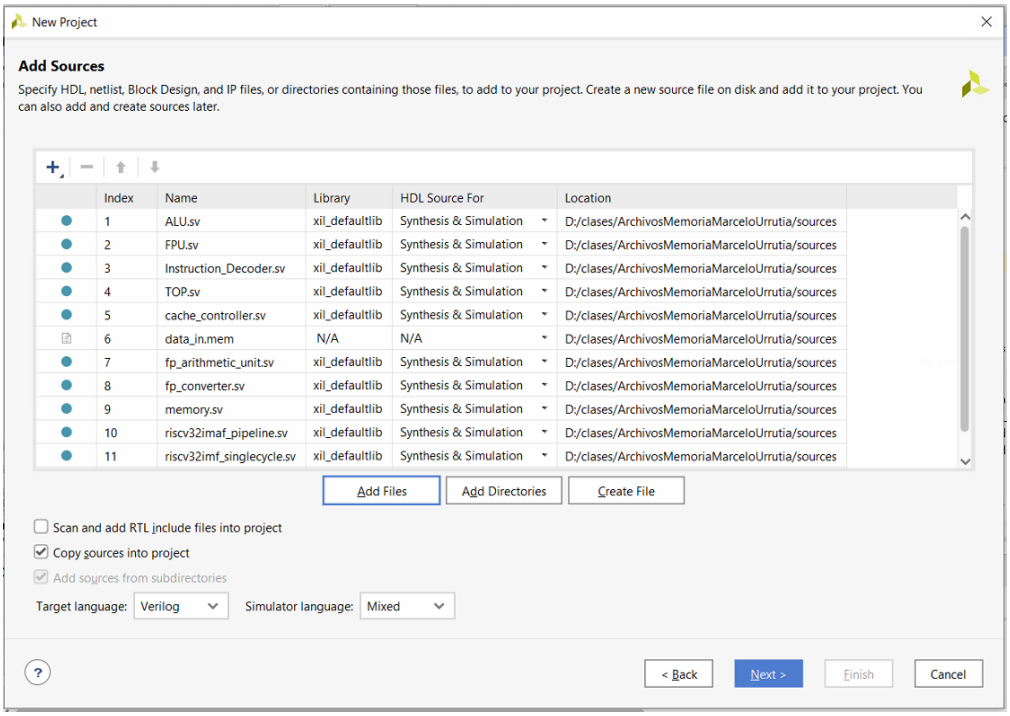


Figura 7.3: : Final de la ventana de selección.

7. Se repite el proceso anterior, pero esta vez con el archivo *top_constraints.xdc* que se encuentra en la carpeta *Constraints* del proyecto. Figura 7.4.

8. En la siguiente pestaña, en la parte superior izquierda, se debe cambiar la sección de *Parts* por la sección *Boards*. Una vez dentro, se busca en el listado la Tarjeta a utilizar, es decir, *Nexys-A7 100T*. Figura 7.5

9. Una vez abierto el proyecto, se debe escribir lo siguiente en la *Tlc Console: set_param pwropt.maxFaninFanoutToNetRatio 1000*. Además de esto, se debe definir el archivo TOP.sv como el archivo principal, para esto se le da un clic derecho, y se elige la opción *Set as Top*.

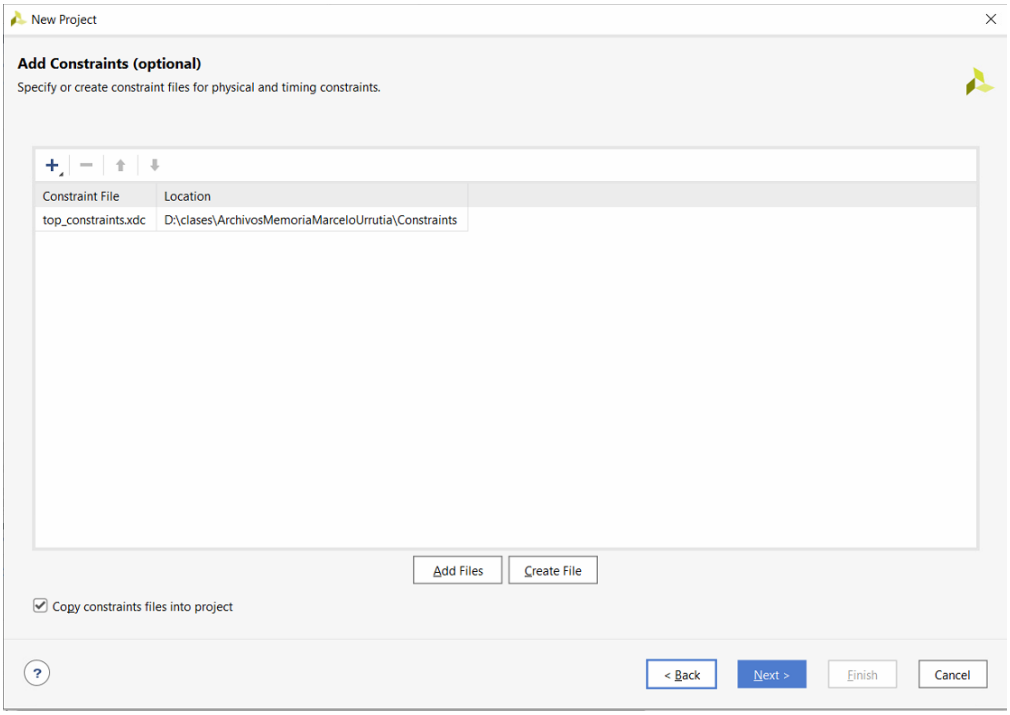


Figura 7.4: : Final de la ventana de selección *constraints*.

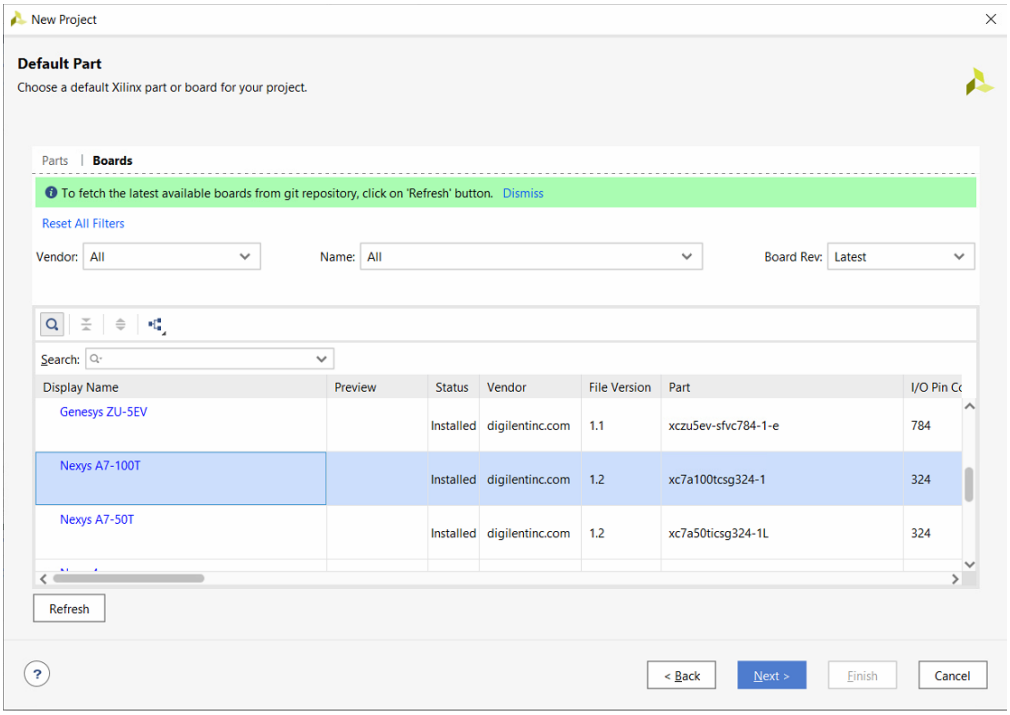


Figura 7.5: : Ventana de selección de tarjeta.

10. Para utilizar el SoC, se debe compilar un código de *assembler* en la memoria del SoC (*text_in.mem*). Para esto se debe ejecutar el archivo *AssemblyDecoderFull.py* con *python 3.10*. El archivo se encuentra en el *Github* del proyecto, en la carpeta *CompiladorAssembler*.

11. Luego el archivo pedirá ingresar el nombre de un archivo, este es el código *assembler* a compilar. El archivo debe estar en la misma carpeta en la que se ubica el archivo *python*. Al ingresar el nombre en la consola, si el archivo se compila, se crearán 4 nuevos archivos, *text_in, data_in, text_in_formatted y data_in_formatted*. En caso contrario saltará un error.

12. Los archivos terminados en *formatted* corresponden al archivo a cargar en las memorias del SoC. Para cargar se copia todos los datos del documento (puede hacerlo con CTRL+A) y se pega en los archivos *data_in.mem, y text_in.mem* según corresponda dentro del proyecto de *Vivado*.