Estudio parcial del procesador SERV: desarrollo inicial de una variante orientada a la educación

Òscar Mor Navarro

2 de julio de 2024

Resumen– Este trabajo se centra en el estudio parcial del procesador SERV (procesador serie a nivel de bits), y el rediseño de ciertos módulos de este, haciendolos más fáciles y asequibles para poder ser utilizados en un entorno educativo. Mas concretamente se han estudiado y rediseñado el módulo de la ALU, el módulo reordenador del valor inmediato, y los dos módulos bufreg y bufreg2. Para el desarrollo del proyecto, se ha utilizado una metodología cascada, realizando un desarrollo de forma secuencial. A su vez, y para obtener una visión global, se ha aplicado una aproximación Top-Down. Una vez rediseñados los módulos se ha verificado su funcionamiento con *test benches* utilizando Verilator. Por lo tanto, se concluye que el procesador SERV es una herramienta valiosa para la enseñanza de la arquitectura de procesadores, RISC-V y los lenguajes de descripción hardware, como Verilog.

Palabras clave– Procesador, SERV, Verilog, Verilator, Quartus, RISC-V, ISA, Instrucción, Simulación, ALU, Memória, Decodificación, Test bench.

Abstract– This work focuses on the partial study of the SERV processor (bit-level serial processor) and the redesign of certain modules, making them easier and more accessible for use in an educational environment. Specifically, the ALU, the immediate value generator, and the two modules bufreg and bufreg2 have been studied and redesigned. For the development of the project, a cascada methodology was used, carrying out development sequentially. Additionally, to obtain a global view, a Top-Down approach was applied. Once the modules were redesigned, their functionality was verified with test benches using Verilator. Therefore, it is concluded that the SERV processor is a valuable tool for teaching processor architecture, RISC-V, and hardware description languages such as Verilog.

Keywords– Processor, SERV, Verilog, Verilator, Quartus, RISC-V, ISA, Intruction, Simulation, ALU, Memory, Decodification, Test bench.

1 Introducción - Contexto del trabajo

En el mundo informático actual, caracterizado por la constante evolución de la tecnología, los procesadores juegan un papel fundamental en el rendimiento y la eficiencia de los sistemas informáticos. Usualmente, cuando pensamos en procesadores, se nos viene a la mente, el componente esencial encargado de ejecutar las instrucciones de programas almacenados en la memoria de nuestros ordenadores o smartphones. Pero el uso de estos está mucho más distribuido, yendo desde centros de computación de altas prestaciones como clústers, hasta sistemas informáticos menos exigentes computacionalmente, como son los sistemas embebidos.

Este proyecto se centra en el estudio de ciertos módulos de un procesador RISC-V de bajo rendimiento computacional que fue diseñado con la intención de reducir al máximo el coste hardware de este. Esto se consigue mediante la serialización bit de las distintas operaciones que realiza la CPU.

Procesadores como estos, están presentes en el mercado, como sería el caso de "SERV". El problema de estos procesadores es que tanto el diseño como el código Verilog es relativamente complejo de entender. Para ello, y con fines

• E-mail de contacte: oscar.mor@autonoma.cat
• Mención realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Joaquin Saiz Alcaine (Microelectronics and Electronic Systems)
• Curs 2032/2024

pedagógicos, se ha simplificado el diseño de los distintos módulos, intentando que el coste hardware no se vea incrementado en exceso.

El procesador SERV [1], diseñado para trabajar con la ISA de RISC-V, específicamente el conjunto de instrucciones RV32I, ha sido concebido para admitir una compilación completa y ser compatible con los sistemas operativos modernos. RV32I, con su conjunto de instrucciones, está diseñado para minimizar el hardware necesario en una implementación básica, lo que lo hace adecuado para entornos donde se requiere eficiencia en términos de recursos. Este conjunto de instrucciones, junto con el conjunto de registros de 32 bits, proporciona un entorno ágil y eficaz para la ejecución de programas y operaciones de procesamiento de datos.

2 Estado del arte

En el contexto actual de los procesadores serie [2], hay una arquitectura que destaca sobre las demás, SERV. Cuando hablamos de SERV, estamos hablando de la CPU más pequeña diseñada a partir de la ISA de RISC-V. Para hacer un buen uso de esta arquitectura, este, debe combinarse con otros componentes como memorias, aceleradores o controladores de periféricos. Ello da como resultado diferentes implementaciones que se encuentran actualmente en el mercado, como son:

- Servant [3]: Se trata de un SoC muy básico que contiene lo justo para ejecutar Zephyr RTOS. Está pensado para su implementación en FPGAs. A fecha de este trabajo, se ha llegado a implementar exitosamente en 20 FPGAs distintas.
- Subservient [4]: es un pequeño SoC basado en SERV independiente de la tecnología y destinado a implementaciones ASIC junto con una SRAM de puerto único.
- Litex [5]: Es un marco de trabajo basado en Python para crear SoCs FPGA. SERV es uno de los más de 30 núcleos compatibles. Se ha utilizado un SoC generado por Litex para ejecutar DooM en SERV.

3 Objetivos

Para poder llevar a cabo tanto el estudio, la simplificación como el desarrollo parcial de una variante del procesador, es necesario establecer ciertos objetivos.

El primer objetivo será estudiar el funcionamiento del procesador SERV para tener una visión y comprensión global del funcionamiento del mismo. Ello conllevará también consolidar los conocimientos relativos a la arquitectura ISA RISC-V.

Una vez obtenida esa visión global, el objetivo es realizar el rediseño de distintos módulos de dicho procesador, bajo la arquitectura RISC-V RV32I, de forma que se facilite al máximo su comprensión, con vistas a su uso pedagógico.

Este objetivo puede ser desglosado en los distintos módulos a ser estudiados y rediseñados. En concreto se abordará el análisis del módulo ALU, del módulo de procesamiento de valor inmediato y de los módulos multifuncionales bufreg y bufreg2. De esa forma, y gracias a la modularidad, se facilita la realización de pruebas exhaustivas de verificación funcional de cada módulo rediseñado mediante *test benches*.

A su vez, y como el proyecto tiene también un fin pedagógico, la implementación de fácil comprensión deberá verse reforzada por la elaboración de una documentación de apoyo para cada módulo rediseñado (esquema lógico, descripción/explicación de entradas y salidas, y código verilog sobrecomentado).

4 Metodología y Planificación

La metodología [6] de desarrollo seleccionada es la cascada, también conocida como desarrollo secuencial, caracterizada por organizar el proceso de manera lineal, donde cada fase se completa antes de poder pasar a la siguiente.

Cada fase del proyecto es una continuación de la anterior, y, por lo tanto, esto hace que se adapte muy bien al diseño de los distintos módulos a estudiar y implementar.

Para tener una mejor comprensión del procesador, se ha decidido utilizar una de las metodologías más comunes en el desarrollo hardware, la metodología Top-Down.

Se utilizará la metodología Top-Down para entender mejor la arquitectura SERV, ya que nos ofrece una visión general del diseño del procesador, y nos facilita la descomposición y comprensión del diseño de módulos clave.

A su vez, se realizarán iteraciones continuas para refinar el diseño y realizar las pruebas de verificación en cada uno de los módulos.

Para realizar la planificación, se ha distribuido la carga de trabajo en las diferentes semanas/entregas, con el fin de distribuir de forma más óptima la carga de trabajo a lo largo del proyecto. Por ello, y teniendo en cuenta que la metodología de planificación escogida es la cascada, como se puede observar en la figura (fig. 1) del anexo, se ha separado en 4 grandes bloques de desarrollo, separados por los 4 módulos a entregar y las entregas a realizar:

- Módulo ALU
- immdec
- bufreg2
- bufreg

5 Entorno de desarrollo

Para el desarrollo de este trabajo, es necesario entender y utilizar las tecnologías que mejor se adapten a este.

La responsabilidad del diseño lógico del procesador recae en el lenguaje de descripción de hardware Verilog[7]. En los últimos años, Verilog, junto con SystemVerilog [8], ha ganado mayor relevancia en el mercado, en comparación con otros lenguajes de descripción de hardware como VHDL[9].

En el desarrollo del proyecto, se ha optado por trabajar con Quartus para la realización de las simulaciones preliminares de los módulos, y los *test benches* han sido llevados a cabo con Verilator.

6 DESARROLLO

En esta sección, se explicarán las distintas fases del desarrollo del proyecto, desde la ampliación de conocimiento sobre RISC-V en la fase previa o la arquitectura de SERV, así como los criterios que se han utilizado para el rediseño de los distintos módulos del procesador, y el porqué de las decisiones tomadas.

6.1. Fase previa

El trabajo realizado en la fase previa al inicio del proyecto se basa principalmente en profundizar los conocimientos de la ISA de RISC-V y, de manera específica, del núcleo RV32I. Asimismo, en esta fase se inicia el estudio de la microarquitectura del procesador SERV.

6.1.1. RISC-V

Risc-V [10] es una arquitectura de conjunto de instrucciones abierta, diseñada para ser escalable y flexible, permitiendo su implementación en una amplia gama de aplicaciones. Está formada por distintos conjuntos de instrucciones, que permiten realizar desde implementaciones más complejas, hasta implementaciones más simples, como sería el caso de SERV, utilizando el conjunto de instrucciones RV32i.

RV32i es la implementación base de la arquitectura de 32 bits e incluye un total de 47 instrucciones esenciales (ver fig. 3 del anexo) para operaciones aritméticas, lógicas, de control de flujo y acceso a memoria, siendo suficiente para construir un procesador básico. Estas instrucciones se distribuyen en 6 grupos según su formato de codificación, como se puede observar en la fig. 8.

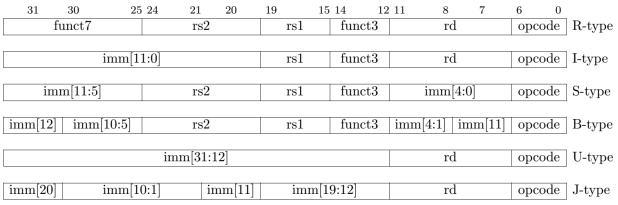


Fig. 8: RISC-V formato de instrucción

Al analizar la distribución de los campos dentro de cada formato de registro de instrucción destacan 3 grandes componentes:

- ***identificador:*** Conjunto de números binarios que identifican la instrucción a realizar, formado por los campos opcode, funct3 y funct7.
- ***rs1:*** Indica el índice del registro dentro del banco de registros, donde se encuentra el primer operando de la operación a realizar.
- ***rs2:*** Indica el índice del registro dentro del banco de registros, donde se encuentra el segundo operando de la operación a realizar.
- ***inmediato:*** Presente en las instrucciones que operan con valor inmediato. . En determinados formatos dicho valor es el segundo operando de la operación a realizar.

6.1.2. SERV

SERV es un procesador serie a nivel de bits. Esto permite que sea capaz de operar con las instrucciones de RISC-V (RV32I), mediante la reutilización de la lógica de los distintos módulos. A modo de ejemplo, para realizar ona operación AND entre dos operandos de 32 bits, se empieza a operar desde el bit menos significativo, y en cada ciclo de reloj se van operando los distintos grupos de dos bits (un bit de cada operando) hasta llegar al el bit más significativo, obteniendo el resultado deseado.

El ciclo de vida de una instrucción en SERV comienza cuando se emite una solicitud de una nueva instrucción, y termina cuando el PC se actualiza con la dirección de la siguiente instrucción a ejecutar.

Diferenciamos entre dos tipos de instrucción, aquellas que requieren una etapa (32 ciclos de reloj), y las que requieren dos etapas (32 ciclos + 32 ciclos y en ciertos casos algunos ciclos adicicionales). Más concretamente, las instrucciones de branch, shifts, slt y los loads/store requieren dos etapas mientras el resto de las instrucciones se ejecutan en una etapa.

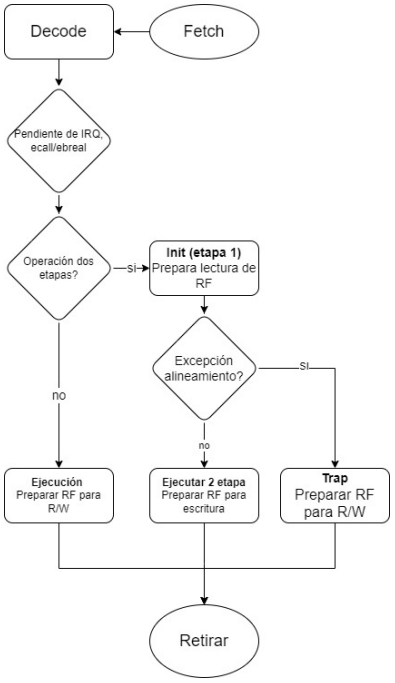


Fig. 9: Flujo de ejecución de instrucción

En la fig. 9 se muestra un diagrama que resume el flujo de ejecución de una instrucción.

Es interesante reseñar que en SERV el banco de registros (*Register File*, RF) no forma parte del núcleo del procesador, ofreciéndose libertad en lo que a su implementación se refiere. En concreto, la versión por defecto de SERV realiza una implementación del banco de registros basada en el uso de memoria, ayudándose para ello de un módulo específico que actúa a modo de interfaz.

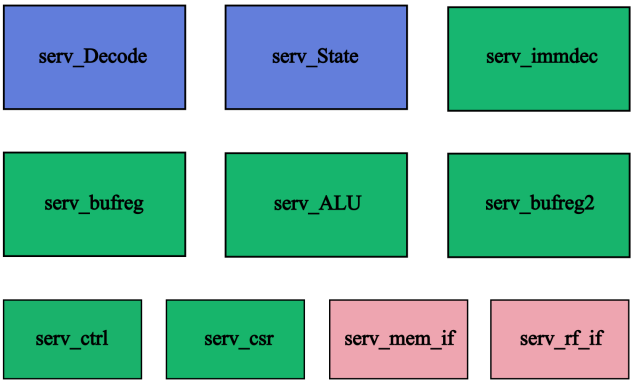


Fig. 10: Módulos de SERV

La fig. 10 muestra los módulos que conforman el procesador SERV. En azul se puede observar los módulos de control, serv_decode y serv_state, encargados de generar la mayoría de las señales de control. En verde se presentan los módulos encargados del procesamiento: serv_immdec, serv_bufreg, serv_ALU_serv_bufreg2, serv_ctrl y serv_csr. De este grupo, los cuatro primeros módulos han sido estudiados y se ha diseñado variantes para ellos. Finalmente, los módulos en rojo son módulos que actúan como interfaz con el banco de registros y la memoria (serv_rf_if y serv_mem_if, respectivamente).

6.2. Modulo ALU

El módulo de la ALU (fig. 11), es una de las partes centrales de una CPU, debido a que este módulo es el encargado de realizar las operaciones aritméticas, como sumas y restas (los shifts, se encargan de realizarlo los módulos de bufreg y bufreg_2). El módulo ALU, también se encarga de realizar las operaciones lógicas como serían las operaciones OR, XOR, o AND. Asimismo realiza operaciones en las instrucciones SLT y branches.

La ALU se encarga de, a partir de las instrucciones recibidas, y mediante los registros de entrada, procesar las instrucciones, para posteriormente devolver el resultado al registro de salida, realizando las operaciones aritmético-lógicas pertinentes.

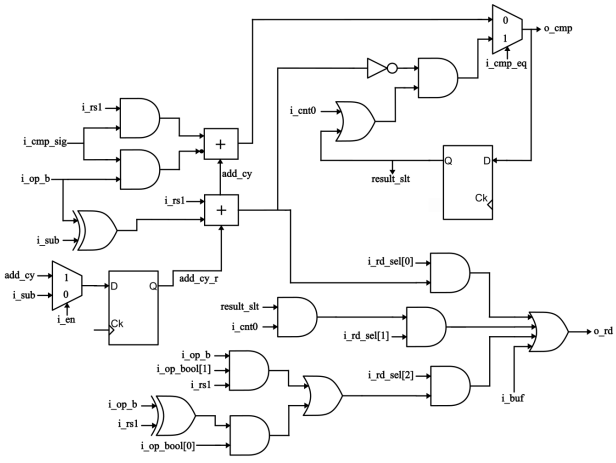


Fig. 11: Esquema ALU - SERV

Para entender el funcionamiento de la ALU, es imprescindible entender las señales con las que se trabajará. Esto nos permite obtener una mejor comprensión de como interactúan entre ellas, y, por lo tanto, como se realizará las distintas operaciones aritmético-lógicas.

En este caso, y como se puede ver en fig. 11, los principales puertos de entrada de datos:

- ***i_rs1:*** Bit de rs1.
- ***i_op_b:*** Bit de rs2, o del valor inmediato.
- ***i_buf:*** Conexión directa a rd.

En cuanto a los puertos de entrada de control, encontramos los siguientes:

- ***i_en:*** Indica si estamos dentro de una etapa de ejecución.
- ***i_cnt0:*** Indica el primer ciclo de reloj de una etapa.
- ***i_cmp_sig:*** Indica si las operaciones aritméticas a realizar serán con signo.
- ***i_cmp_eq:*** Indica si se realizará una comparación de igualdad o desigualdad.
- ***i_bool_sel [1:0]:*** Indica qué operación lógica realizar (ver tabla 1).

XOR	00
OR	10
AND	11

TABLA 1: VALORES BOOL-SEL

- ***i_rd_sel [2:0]*** Señal de 3 bits, para indicar a que salida conectar.

Como puertos de salida, hay dos señales, una la salida principal (rd), y la segunda como flag de status , que permite hacer operaciones como *branch* y *SetifLessThan*:

- ***o_rd:*** Salida principal, según la señal de control *in_rd_sel*, permite conectar la salida para que realice una operación lógica, aritmética (suma/resta), operaciones tipo slt, o conectar directamente al valor del puerto de entrada *i_buf*, ya comentado anteriormente.
- ***o_cmp:*** Flag de status generado por la ALU, que informa si se cumplen dos posibles condiciones. Estas dos posibles condiciones son:
  - El resultado de una operación igual a cero.
  - Si el operando 1 (rs1) es menor que el operando 2.

6.2.1. Variante del diseño de la ALU

Para facilitar la comprensión del módulo ALU de serv, se ha modificado tanto la salida rd como la selección de las operaciones booleanas (fig. 12), ya que la implementación de estas se realiza mediante dos funciones combinacionales relativamente complejas, substituyendo estas por dos multiplexores.

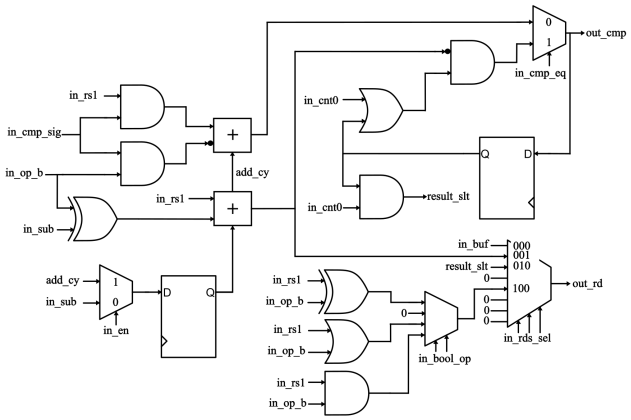


Fig. 12: Esquema ALU modificado

6.2.2. Verificación ALU

Para comprobar el correcto funcionamiento del módulo, se han realizado un conjunto de simulaciones funcionales, una por cada una de las instrucciones que requieren el uso de la ALU (add, sub, and, or, xor, beq, btl, btlu, stl). Con ello comprobamos que el funcionamiento está siendo el correcto. Se ha comprobado el resultado obtenido con los valores obtenidos con el código de serv, siendo estos idénticos. Como se puede ver en la figura del anexo (fig. 4), se realiza una simulación de una operación AND, dode los operandos toman los siguientes valores:

- rs1: 10101010101010101010101010101010
- op_b:11001100110011001100110011001100
- rd: 10001000100010001000100010001000

En la fig. 2, se puede observar el inicio del código c++ realizado para verificar a partir de la herramienta de simulación Verilator.

6.3. Mòdulo de reordenación del valor imediato (immdec)

Uno de los principales aspectos a considerar al trabajar con las instrucciones de valor inmediato en RISC-V, como se puede observar en la fig. 8, es comprender cómo se distribuye este valor en los distintos bits de un registro de instrucción.

Por lo tanto, este módulo (fig. 13) se encarga de ensamblar las diferentes partes inmediatas de la palabra de instrucción y organizarlas correctamente para su salida. Esto se realiza mediante el uso de registros *superpuestos* de desplazamiento que están conectados de manera óptima, con el fin de ensamblar el valor inmediato por completo, y de forma ordenada. A su vez, este módulo, también es el encargado de obtener las direcciones de rs1, rs2 y rd haciendo uso de los mismos registros citados anteriormente.

Los registros internos guardan los valores de los datos bits [31:7] de la instrucción procedente del bus de instrucciones, entre los cuales se distribuyen los valores inmediatos, dependiendo del tipo de instrucción a realizar. Con el uso de señales de control, es posible formar y ordenar los valores inmediatos necesarios, de forma completa.

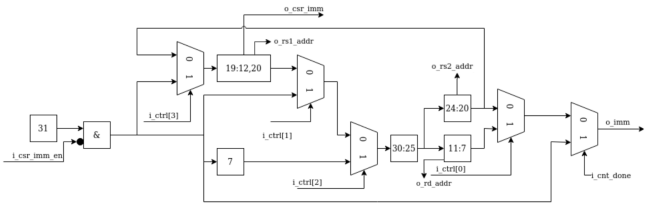


Fig. 13: Esquema decodificación inmediato - SERV

6.3.1. Diseño módulo valor imediato

Para facilitar la comprensión de este módulo y de cómo interactúan las señales se ha decidido simplificar el diseño, modificando el diseño de la lógica, a pesar de que el coste hardware incremente.

La principal diferencia recae en los registros de este módulo. En el original se trabaja con registros donde se realiza una carga paralela de los distintos bloques de bits, que contienen información del valor inmediato (y otros datos como las direcciones de rs1, rs2 y rd), ordenando los bits de tal forma que permitan formar el valor inmediato. Ello se hace conectando adecuadamente esos registros entre sí y realizando desplazamientos, todo ello regido mediante ciertas señales de control.

Como alternativa, se ha propuesto modificar los registros internos, en lugar de representar bloques de bits, estos son cargados de forma paralela con los bits del valor inmediato ya ordenados. Consecuentemente, y como se realiza en el diseño de SERV, únicamente es necesario desplazar estos bits de forma secuencial hacia la salida (fig. 14). Asimismo, se han añadido tres registros de cinco bits con el fin específico de almacenar las direcciones de rs1, rs2 y rd.

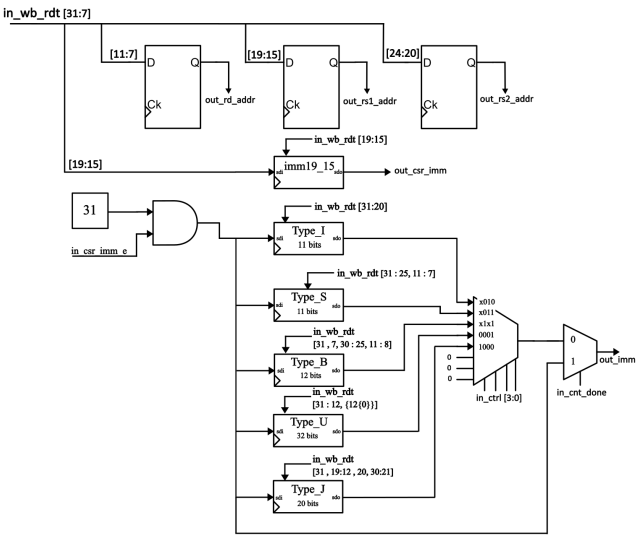


Fig. 14: Esquema decodificación inmediato simplificado

Para facilitar la explicación de los puertos de entrada, estos se han dividido en dos grupos. El primero está formado por las señales de entrada encargadas del control de los registros internos del módulo. El segundo grupo está formado por las señales de entrada (tanto de datos como de control) que son mostradas en es el esquema de la fig. 14. Empecemos comentando las señales de este segundo grupo:

- ***in_clk:*** Señal de reloj.
- ***in-csr-imm-en:*** Extensor de signo, e indicador de sí se está trabajando con una instruccion crs.
- ***in_ctrl[3:0]:*** Anteriormente encargada de concatenar los distintos bloques según el tipo de instrucción a realizar. Actualmente señal de un solo multiplexor, el cual conecta el tipo de instrucción a realizar con la salida (out_imm).
- ***in_cnt-done:*** Señal de control donde si toma el valor 0, se conecta el contenido de los registros internos, si toma el valor 1, se conecta el bit de signo.
- ***in_wb_rdt [31:7]:*** Señal de entrada, que contiene bits del 31 al 7 de la instrucción con la que se trabajará. Obteniendo en este rango de bits, el valor inmediato y las direcciones de rs1, rs2, rd, dependiendo del tipo de instrucción.

A su vez, encontramos los puertos de entrada de control de los registros internos (las señales del primer grupo antes referido), como se puede observar en la fig. 15.

Estos registros tienen dos funciones principales. La primera es cargar directamente el valor correspondiente de los bits que se requiere guardar manteniendo ese valor. La segunda se encarga de realizar el shift entre los bits, desplazándolos para sacar por la salida el bit menos significativo en cada ciclo de reloj.

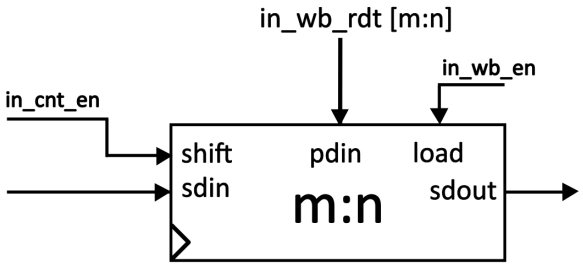


Fig. 15: Registro interno imm-dec

Estas señales de control son:

- ***in_cnt_en:*** Indica que se está en una etapa.
- ***in_wb_en:*** Permite realizar la carga de los bits de in-wb-rdt, en los registros internos si está en alta.

En cuanto a los puertos de salida, encontramos dos, visibles en la fig. 14, siendo estos:

- ***out_imm:*** Salida secuencial del valor inmediato ya ordenado. En cada ciclo de reloj se saca un bit, desde el bit menos significativo al más significativo.
- ***out_csr_imm:*** Salida del valor inmediato de las instrucciones csr, situado en los bits [19:15].

6.3.2. Verificacióndel mòdulo immdec

Para verificar el correcto funcionamiento del módulo se han realizado una serie de simulaciones para cada una de las salidas, comprobándolas con las salidas obtenidas con el módulo original.

Para la salida out_csr, se ha realizado una carga de los bits [19:15], obteniendo la salida secuencial esperada.

Para la salida out_imm, se ha realizado una simulación para cada uno de los tipos de instrucción. En la fig. 6 del anexo, se puede ver un ejemplo de simulación para una instrucción de tipo I (addi x5, x1, 0x1BFA000). En todos los casos, se han obtenido una salida igual a la original.

6.4. Módulo bufreg2

El módulo bufreg2 (fig. 16), es un registro de buffer, el cual se encarga de realizar las operaciones de store, load y además, es el encargado de realizar el conteo de bits para las operaciones de shift.

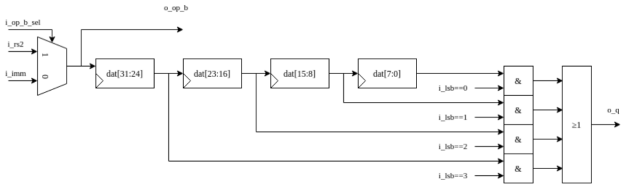


Fig. 16: serv - bufreg2

Las entradas de este módulo son:

- ***i_clk:*** Señal de reloj.
- ***i_en:*** Indica que estamos dentro de una etapa.
- ***i_init:*** Indica que estamos en la primera etapa.
- ***i_cnt_done:*** Indica el último ciclo de una etapa.
- ***i_lsb:*** Permite seleccionar los bytes dentro del registro interno.
- ***i_byte_valid:*** Señal para asegurar el alineamiento en memoria.
- ***i_op_b_sel:*** Permite escoger entre cargar secuencial rs2, o imm.
- ***i_shift_op:*** Indica si se realiza un desplazamiento de bits.
- ***i_rs2:*** Contiene un bit de rs2.
- ***i_imm:*** Contiene un bit inmediato.
- ***i_load:*** Indica si se realiza una carga paralela en el registro interno.
- ***in_dat [31:0]***: Señal de entrada de 32 bits, que realiza una carga paralela desde el bus de datos al registro.

Por puertos de salida, encontramos:

- ***o_sh_done:*** Indican si el desplazamiento de bits ha terminado.
- ***o_sh_done_r:*** Indican si el desplazamiento de bits ha terminado (con un ciclo de retardo respecto a o_sh_done)
- ***o_op_b:*** Salida secuencial con el valor de rs1 o el valor inmediato.

- ***o_dat [31:0]:*** Salida paralela, desde el registro interno del módulo al bus de datos.

Como se ha comentado anteriormente, bufreg2, es el módulo encargado de realizar las operaciones de load, store y shift. Operaciones que, para su procesamiento, es necesario realizarlas en dos etapas.

- ***Store***: En la primera etapa (mientras la señal init está activa), se realiza el desplazamiento de los bits de rs2 al registro para posteriormente enviarlo a memoria por el bus de datos. En la segunda etapa es necesario incrementar el PC.
- ***Load***: Se carga el dato proveniente de la memoria de manera paralela y se procesa según los dos últimos bits de la dirección de memoria (entrada i_lsb). En la segunda etapa se realiza un desplazamiento de bits hacia rd.
- ***Shifts***: La forma en la que se tratan los shifts (fig. 17 es un tanto especial debido a que trabaja conjuntamente con el módulo bufreg. Durante la primera etapa y mientras i_init es igual a 1 (primeros 32 ciclos de reloj), bufreg2 se encarga de cargar el número de desplazamientos a realizar en el registro, ya sea por rs2 o por el valor inmediato. Una vez se ha cargado el valor en los 6 bits menos significativos, se inicia el conteo de forma descendiente, este llega a 0 y envía una señal a bufreg, para indicar que el conteo ha terminado.

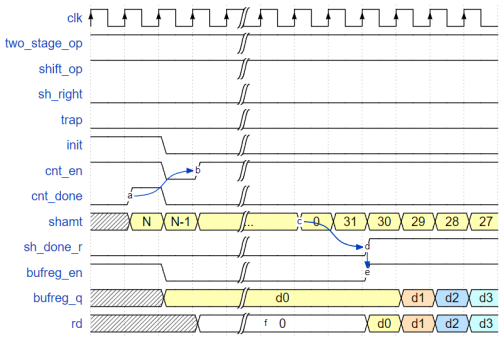


Fig. 17: shift_waveform, ref:SERV

6.4.1. Variante de diseño bufreg2

Para modificar el diseño de bufreg2 (fig. 18), se ha decidido separar/duplicar los últimos 6 bits del registro (encargados del conteo), utilizadas cuando in_shift_op es igual a 1. Por lo tanto, ahora se realiza el conteo decreciente en un contador externo al registro, facilitando de esa manera la comprensión de este.

6.4.2. Verificación del módulo bufreg2

Para verificar el correcto funcionamiento de bufreg2, se han realizado simulaciones sobre loads, store y shifts con distintos valores de desplazamiento, y se ha obtenido resultados idénticos al módulo original. En la fig. 7 del anexo, se puede observar una instrucción load, donde se carga el registro con el valor 0xAAAAAAAA.

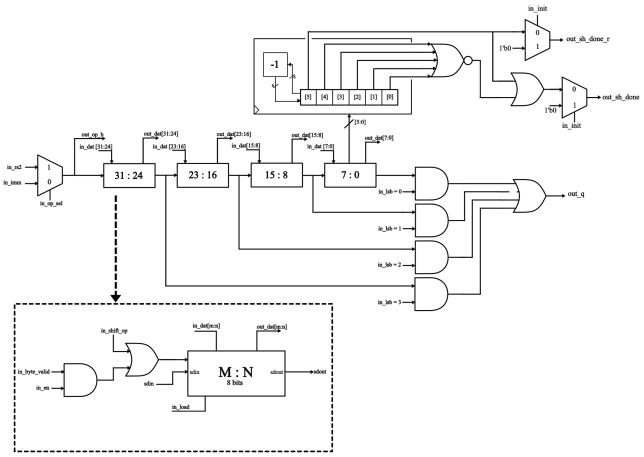


Fig. 18: Variante módulo bufreg2

6.5. Módulo bufreg

Bufreg (fig. 19) es el módulo encargado de mantener los datos entre etapas, en aquellas instrucciones que requieren de dos etapas de ejecución. En la primera etapa, se encarga de cargar de forma secuencial el valor de rs1, el valor inmediato, o la suma de estos dos. Ello se hace con el fin de, a partir del pc y los valores anteriormente nombrados, obtener la dirección efectiva de la posición de memoria a leer, escribir o a la cual saltar.

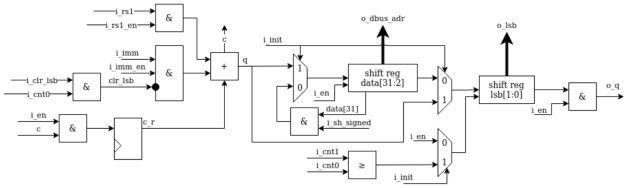


Fig. 19: serv buffreg

Como se ha comentado anteriormente, este módulo se encarga de operar con las instrucciones que requieren dos etapas, siendo estas jal, jalr, branches, loads, stores y shifts. Sus entradas y salidas son las siguienetes:

- ***in_cnt0:*** Indica el primer ciclo de reloj.
- ***in_cnt1:*** Indica el segundo cilo de reloj.
- ***in_en:*** Indica si estamos dentro de una etapa,
- ***in_init:*** Indica si estamos dentro de la primera etapa.
- ***in_rs1_en:*** Indica si se opera con rs1.
- ***in_imm_en:*** Indica si se opera con el valor inmediato.
- ***in_clr_lsb:*** Si está activa, pone a 0 el bit menos significativo del valor inmediato.
- ***in_rs1:*** Valor secuencial de rs1.
- ***in_imm:*** Valor secuencial de imm.
- ***out_lsb[1:0]:*** Los dos bits menos significativos, para comprobar errores de alineamiento de memoria.
- ***out_dat[31:0]:*** Salida paralela del registro de datos.

- ***out_q:*** Salida secuencial de rs1, imm, o rs1 + imm.

Un aspecto peculiar de este módulo es el tratamiento especial que se da a los dos bits menos significativos, los cuales son almacenados en un registro separado. Como se puede observar en la fig. 19, se realiza la carga de los dos bits menos significativos en los dos primeros ciclos de reloj, permitiendo así comprobar errores de alineamiento de memoria lo más pronto posible.

6.5.1. Variante de diseño Bufreg

A partir de aquí, hay que identificar que operaciones trabajan conel valor inmediato, con rs1, o con la suma de estos dos. Ya que estos valores serán los almacenados de forma secuencial en el registro de desplazamiento interno, para su posterior salida por el puerto de salida de forma secuencial.

En las instrucciones nombradas anteriormente, el módulo (fig. 20) tiene un comportamiento similar en todas ellas, a excepción de las operaciones srl y sll.

Ello se debe a que estas operaciones se ejecutan trabajando en conjunto con el módulo bufreg2. Como ya se ha explicado en el punto anterior, el módulo bufreg2 cuando concluye el conteo de desplazamientos envía la señal sh_done_r al módulo de control serv_state. Será serv_state quien entonces reactivará la señal de control que llega a la entrada i_en que posibilitará la finalización de la realización de la operación de desplazamiento.

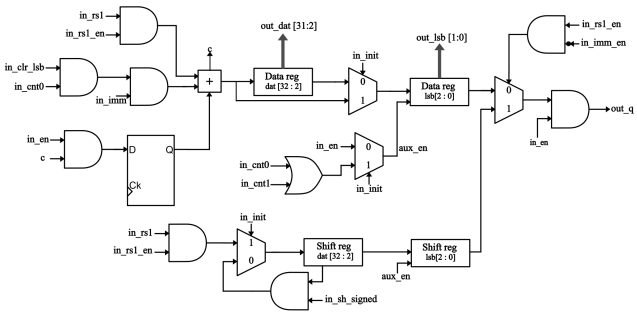


Fig. 20: Variante módulo bufreg

Otro aspecto que también distingue las operaciones de shift es el hecho de que no trabajan con direcciones de memoria con lo que resulta innecesario el tratamiento diferenciado de los dos últimos bits. Por todo ello, y para facilitar la comprensión del módulo bufreg, en la variante que se ha elaborado se ha añadido un registro específico para la implementación de las operaciones de shift.

6.5.2. Verificación Bufreg

Para la verificación del módulo bufreg, se han realizado verificaciones utilizando Verilator, sobre cada una de las instrucciones que se ejecutan en este (JAL, JALR, branches, load/store, sll, srl), Obteniendo unos resultados correctos, respecto a la funcionalidad del módulo. En la fig. 8 se puede observar la simulación de una instrucción *jal rd, 0x1baed*,

7 PRESENTACIÓN Y DISCUSIÓN DE RESULTADOS

7.1. Comprensión del funcionamiento global de SERV

Para comprender el funcionamiento general del procesador SERV, es esencial entender la interacción de las distintas señales con los diferentes módulos. En particular, dos módulos, serv_decode y serv_state, son de gran relevancia para conocer la finalidad y el significado de un amplio conjunto de señales de control y su relación con el funcionamiento global del procesador. De estos módulos es necesario conocer la finalidad de las señales que generan si bien no es imprescindible conocer su estructura interna.

El módulo serv_decode se encarga de la decodificación de las instrucciones, interpretándolas y generando las señales de control necesarias para determinar qué operaciones se realizan, cómo se manejan los datos y sobre qué registros trabajan. Las siguientes señales son generadas por serv_decode y utilizadas por los módulos estudiados.

Para la ALU:

- ***o_alu_sub:*** Indica si la operación a realizar es una resta.
- ***o_alu_bool_op[1:0]:*** Selector de la operación lógica a realizar (and, or, xor).
- ***o_alu_cmp_eq:*** Indica que se está realizando una instrucción *branch* (beq, bne).
- ***o_alu_sig:*** Indica que se está realizando un *branch* con números signados (beq, bne, blt o bge).
- ***o_alu_sel [2:0]:*** Indica la operación a realizar, add/sub (bit 0), slt (bit 1), o booleana (bit 2).

Para el módulo immdec:

- ***o_immdec_ctrl[3:0]:*** codifica el tipo de instrucción: bit 3 (1: J, 0: U), bit 2 (1: B), bit 1 (1: I/S, 0: U/J), bit 0 (1: B/S, 0: J/I)
- ***o_imm_en[3:0]:*** bit 3 (1 para B, I, J, S, U), bit 2 (1 para I, J, U), bit 1 (1 para J, U), bit 0 (1 para B, S).

Para el módulo bufreg2

- ***o_shift_op:*** Indicador de shift.
- ***o_two_strage_op:*** Indicador de instrucción de dos etapas.

Para el módulo bufreg

- ***o_bufreg_rs1_en:*** Habilitador del uso de rs1 en bufreg.
- ***o_bufreg_imm_en_strage_op:*** Habilitador del uso del valor inmediato en bufreg.
- ***o_bufreg_clr_lsb:*** Indicador de sí el módulo deberá borrar el bit menos significativo del valor inmediato.
- ***o_sh_signed:*** Indica que se está realizando un shift aritmético.

Las instrucciones deben ser controladas durante su ejecución, manteniendo un seguimiento de la etapa en la que se encuentran. Esto se realiza mediante el módulo serv_state, que rige y refleja el estado actual del procesador. Algunas de las señales generadas por serv_state son:

- ***o_init:*** Indica primera etapa en una instrucción de dos etapas.
- ***o_cnt_en:*** Indica que se está dentro de una etapa.
- ***o_cnt_done:*** Indica el último ciclo de una etapa.
- ***o_cnt0, o_cnt1, o_cnt2, o_cnt3, o_cnt7, o_cnt0to3, o_cnt12to31:*** Indican un ciclo específico de una etapa.

7.2. ALU

El diseño original de la ALU de SERV presenta una estructura lógica que dificulta la realización de modificaciones significativas. Por ello, se ha decidido modificar el diseño para hacerlo lo más comprensible posible, centrándose en dos aspectos principales.

La submodularidad, que permite separar partes esenciales del diseño (lo cual se refleja tanto a nivel de esquema como a nivel de código) facilitando la comprensión de cada una de las funciones principales del módulo ALU. De esta manera, al separar cada parte, es más fácil entender cómo interactúan entre sí.

Además, para gestionar tanto la salida de out_rd, como la selección de las operaciones lógicas, las cuales eran gestionadas mediante funciones lógicas complejas con el fin de reducir el coste hardware al máximo, estas, se han substituido por multiplexores.

Con estas mejoras, es más fácil comprender este módulo, manteniendo resultados idénticos al módulo original.

7.3. imm_dec

Para el módulo imm_dec de SERV, nos encontramos con una situación diferente a la de la ALU. El diseño de este, a pesar de ser relativamente complejo, permite realizar modificaciones sustanciales, haciendo este más fácil de entender. immdec trabaja con registros *superpuestos*. Esto permite que un mismo registro se utilice para distintos tipos (formatos) de instrucción. Gracias a esto, y al uso de señales de control, permite formar los valores inmediatos de cada instrucción ya ordenados a un coste hardware mínimo.

Una vez alcanzado el conocimiento de la funcionalidad y código de dicho módulo, se decidió implementar un nuevo diseño para facilitar su comprensión.

En este nuevo diseño, se generan registros de diferentes tamaños que realizan una carga paralela de la información del puerto de entrada in_wb_rdt. Estos registros efectúan un desplazamiento de bits en cada ciclo de reloj, y mediante un multiplexor, se selecciona el tipo de instrucción que se conectará a la salida, permitiendo una reordenación más clara y estructurada. Esta implementación no solo mejora la comprensión del diseño (y la legibilidad del código), sino que también mantiene la funcionalidad del módulo original.

7.4. Bufreg2

Las posibilidades de facilitar este módulo están muy limitadas en cuanto se refiere a las operaciones load y store, pero no en operaciones de tipo shift.

En la versión original del módulo en SERV, los últimos 6 bits del registro son los encargados de realizar la cuenta regresiva para indicar cuando se termina el desplazamiento. De tal forma, se ha decidido rediseñar las operaciones de shift separando el conteo descendiente, es decir, los 6 bits menos significativos ([5:0]) del registro de 32 bits. En la nueva variante diseñada se realiza dicha operación mediante un contador (separado del registro) en el que se realiza una carga paralela de estos valores en el último ciclo, cuando in_cnt_done es igual a 1, haciendo posible realizar el conteo a partir del ciclo de reloj pertinente.

Consecuentemente, hay que modificar las señales de salida relacionadas con las operaciones shift.

Estas señales se gestionan mediante el bit 6 del contador. Este bit, durante los primeros 32 ciclos es forzado a tener el valor 0, gracias a la señal in_init. Una vez cargado el valor shamt (shift-amount), se realiza el conteo descendiente hasta que este llegue al valor binario 000000. Consecuentemente, en el siguiente ciclo se restará 1 al valor del contador de nuevo, produciendo esto que todos los bits del contador tomen el valor 1, incluido el sexto bit (es decir, 111111). Por lo tanto, primero se activará la señal combinacional out_sh_done (cuando el contador tenga el valor 000000) y un ciclo después se activará la señal secuencial out_sh_done_r (cuando el contador valga 111111). Estas señales son generadas de la siguiente manera:

- ***out_sh_done:*** Para out_sh_done, se ha cambiado la conexión combinacional original por el uso de una nor de 5 entradas. La salida tomará el valor 1 cuando los 5 bits menos significativos del contador se encuentren a 0.
- ***out_sh_done_r:*** Para la salida out_sh_done_r, se ha conectado directamente al sexto bit (el de más peso) del contador, de tal forma, cuando este valga 1, out_sh_done_r tomara el valor 1.

Se gestiona la salida de ambas señales con un multiplexor y la señal de control in_init.

7.5. Bufreg

Bufreg presenta varias opciones de modificación, de las cuales se han implementado dos. La primera consiste en separar el tratamiento de aquellas instrucciones que realizan una suma (entre imm y rs1), de las que no. En este caso es necesario añadir un registro adicional, para la implementación del tratamiento de las instrucciones que no requieren una suma.

La segunda variante, presentada en el apartado Desarrollo, es separar las operaciones de shift (sll, srl) del resto ya que estas añaden mayor dificultad a la hora de comprender su funcionamiento en este módulo. Todo ello justificado por la sinergia con bufreg2, y la lógica adicional para mantener los signos durante las operaciones de shift. Además, los dos bits menos significativos de este tipo de operaciones no son necesarios para comprobar el alineamiento de memoria, a

diferencia del resto de operaciones. Se ha considerado que esta versión facilitara más la comprensión del módulo, por contraparte a la versión inicial modificada (separar según se haga la suma o no).

8 Conclusiones

El procesador SERV, tiene un gran potencial para ser utilizado en el ámbito pedagógico universitario por distintas razones.

En primer lugar, está basado en una de las ISAs de mayor crecimiento y relevancia en el mercado: RISC-V. Esta arquitectura está ganando importancia cada año, lo que la convierte en una herramienta fundamental para el aprendizaje de los principios del diseño de procesadores y la arquitectura de computadoras. Al ser una arquitectura abierta, RISC-V permite a los estudiantes adquirir habilidades directamente aplicables en el entorno profesional.

En segundo lugar, al tratarse de una arquitectura de procesamiento en serie a nivel de bits, permite comprender conceptos fundamentales de procesamiento, ofreciendo una perspectiva distinta a las implementaciones tradicionales. Esto facilita una comprensión más profunda de los principios básicos del procesamiento y su aplicación en diversas situaciones.

No obstante, a pesar de estas ventajas, nos enfrentamos a ciertos desafíos significativos, como las características intrínsecas del procesador, el elevado grado de optimización interna del propio SERV, la complejidad del código con que está descrito y la limitada documentación disponible sobre este. Por ello, se ha decidido modificar la implementación de cuatro módulos específicos con el fin de hacerlos más fácilmente comprensibles.

El primer módulo, la ALU, se ha modificado ligeramente para mejorar algunos aspectos de su comprensión. El segundo, el módulo de decodificación de instrucciones (immdec), ha sido rediseñado para simplificar la lógica y facilitar su entendimiento. El tercer módulo, bufreg2, se centra en las operaciones de lectura y escritura en memoria, así como en operaciones shift, donde también se han realizado ajustes en la lógica separando las operaciones de tipo shift con ayuda de un contador aparte. Finalmente, para el módulo bufreg, las instrucciones de shift, requerían de lógica adicional no utilizada por el resto de instrucciones, y por lo tanto, se ha decidido gestionar de forma independiente estas operaciones de shift.

Estas modificaciones no solo hacen que el diseño y su código asociado sea más accesible, sino que también permiten a los estudiantes comprender mejor las características fundamentales de un procesador y, en concreto, de un procesador serie.

Como se ha comentado anteriormente, uno de los principales desafíos es la documentación disponible, la cual es bastante limitada y carece de explicaciones detalladas. Esto puede representar un obstáculo significativo para el aprendizaje. Por ello, se ha elaborado una documentación extensa de cada módulo, explicando su función y cómo las señales interactúan para generar la salida deseada. Esta documentación detallada pretende facilitar la comprensión y el estudio de los módulos, mejorando así la experiencia educativa y permitiendo un aprendizaje más profundo y efectivo. En el anexo se puede observar la documentación para el módulo immdec (fig. 5).

9 Agradecimientos

Quiero dar las gracias a mi tutor del proyecto, Joaquín Saiz, por darme la oportunidad de poder realizar este trabajo, acompañándome, supervisando y guiando en el transcurso de este. También me gustaría agradecer a mi familia y amigos por el apoyo a lo largo de este proyecto.

Referencias

[1] https://github.com/olofk/serv?tab=readme-ov-file

[2] S. Dick, V. Gaudet and H. Bai, "Bit-serial arithmetic: A novel approach to fuzzy hardware implementation,"NAFIPS 2008 - 2008 Annual Meeting of the North American Fuzzy Information Processing Society, New York, NY, USA, 2008, pp. 1-6, doi: 10.1109/NAFIPS.2008.4531258. keywords: Hardware;Field programmable gate arrays;Fuzzy logic;Digital arithmetic;Fuzzy systems;Silicon;Prototypes;Computer architecture;Fuzzy control;Throughput

[3] https://serv.readthedocs.io/en/latest/servant.html

[4] https://github.com/olofk/subservient/?tab=readme-ov-file

[5] https://github.com/enjoydigital/litex/blob/master/README.md

[6] Restrepo, S. M. V., Montoya, J. D. V., Adasme, M. E. G., Zapata, E. J. R., Pino, A. A., & Marín, S. L. (2019). Una revisión comparativa de la literatura acerca de metodologías tradicionales y modernas de desarrollo de software. Revista CINTEX, 24(2), 13-23. https://doi.org/10.33131/24222208.334

[7] https://marceluda.github.io/rp_dummy/EEOF2018/verilog.pdf

[8] https://ece.uah.edu/gaede/cpe526/SystemVerilog_3.1a.pdf

[9] L. Tarés, Y. Torroja, S. Ocaz, & E. Villar (1997). VHDL: *Lenguaje estándar de diseño electrónico*. McGraw-Hill.

[10] https://github.com/riscv/riscv-isa-manual/blob/main/src/rv32.adoc

[11] https://chat.openai.com/

Anexo

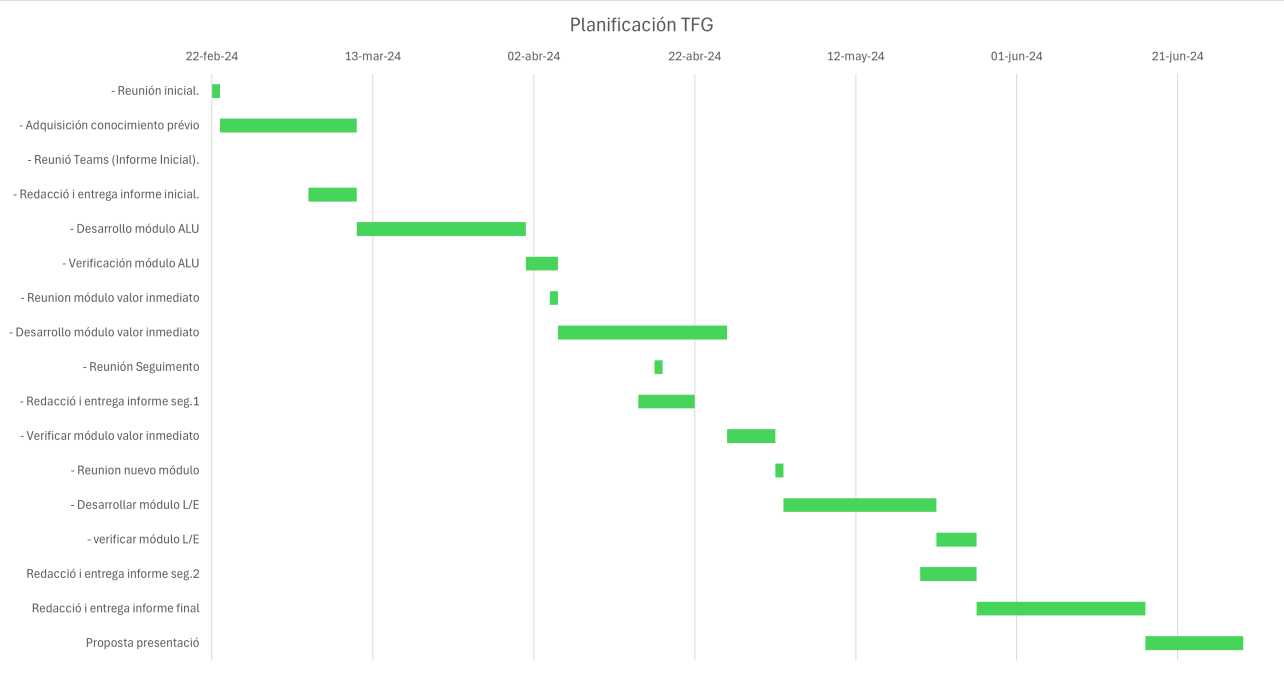


Fig. 1: Planificación

```cpp
#include "VALU.h"
#include "verilated.h"
#include "verilated_vcd_c.h"

vluint64_t sim_time = 0;
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);
    // Instancia del módulo Verilog
    VALU* top = new VALU;
    // Inicializa las entradas de simulación
    top->in_clk = 0;
    top->in_en = 0;
    top->in_cnt0 = 0;
    top->in_sub = 0;
    top->in_bool_op = 3;
    top->in_cmp_eq = 0;
    top->in_cmp_sig= 0;
    top-> in_rd_sel = 4;
    top->in_rs1 = 0;
    top->in_op_b= 0;
    top-> in_buf =0;
    // Inicializa la traza
    Verilated::traceEverOn(true);
    VerilatedVcdC* tfp = new VerilatedVcdC;
    top->trace(tfp, 99); // Traza 99 niveles de jerarquía
    tfp->open("ALU_and_waveform.vcd");

    // Paso de simulación 1 (Flanco de bajada del reloj)
    top->in_clk = 0;
    top->eval();
    tfp->dump(sim_time);
    // Paso de simulación 2 (Flanco de subida del reloj)
    top->in_clk = 1;
    top->in_en = 1;
    top->in_cnt0 = 1;
    top->in_sub = 0;
    top->in_bool_op = 3;
    top->in_cmp_eq = 1;
    top->in_cmp_sig= 1;
    top-> in_rd_sel = 4;
    top-> in_buf =0;
    top->in_rs1 = 1;
    top->in_op_b = 0;

    top->eval();
    tfp->dump(++sim_time);
```

Fig. 2: Simulación AND Verilator

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Fig. 3: Conjunto de instrucciones RV32i

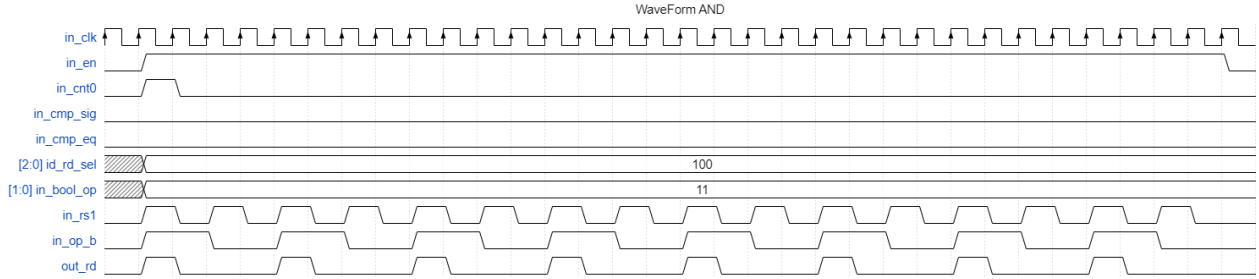
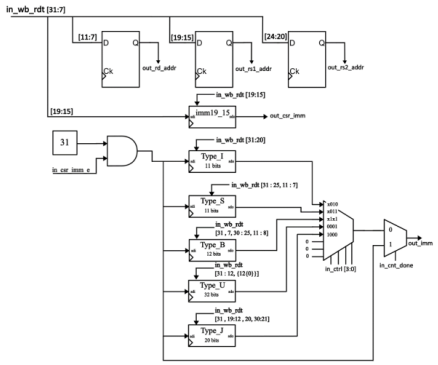


Fig. 4: Simulación AND

Documentación módulo Imm_dec

Este módulo tiene tres funcionalidades principales, de las cuales dos de ellas, se encargan de sacar el valor inmediato de los registros de instrucción. La primera de estas, se encarga de formar los valores de las instrucciones Tipo I, S, B,U y J, sacándolos de forma inmediata. Mientras que la segunda se encarga de sacar el valor inmediato de las instrucciones de Control Status Register (csr). Como tercera funcionalidad, encontramos que este módulo es el encargado de sacar las direcciones de los registros utilizados en cada operación, tanto de rs1, rs2 y rd, en caso de que estas operaciones las utilicen.



Código Verilog comentado

```
module immdec
 //#(parameter SHARED_RFADDR_IMM_REGS = 1)
 (
  input wire        in_clk,
  //State
  input wire        in_cnt_en, //active durante 32 ciclos
  input wire        in_cnt_done, //active en ciclo 32
  //Control
  input wire        in_csr_imm_en,
  input wire [3:0]  in_ctrl,
  output reg [4:0] out_rd_addr,//
  output reg [4:0] out_rs1_addr,
  output reg [4:0] out_rs2_addr,
  //Data
  output wire       out_csr_imm,
  output wire       out_imm,
  //External
  input wire        in_wb_en,
  input wire [31:7] in_wb_rdt);
  reg aux_out_imm;
  wire sign_bit;

  /*registros para la carga y el desplazamiento hacia la salida,
  de los distinotos tipos de instrucción.*/
  reg imm_31;
  reg [10:0]  Type_I;
  reg [10:0]  Type_S ;
  reg [11:0]  Type_B;
  reg [31:0]  Type_U;
  reg [19:0]   Type_J;
  reg [4:0] imm19_15;
  //Assignación salida del inmediato de la instrucción csr
   assign out_csr_imm = imm19_15[0];
   assign sign_bit =  imm_31 &  ~in_csr_imm_en;

  always@(posedge in_clk) begin

  /* cuando in_wb_en esta activa, se cargan los registros con las
  posibles posiciones donde se encuentran lso valores inmediatos,
  y las direcciones de los rgistros */
       if(in_wb_en) begin

           imm19_15 <= in_wb_rdt[19:15];

           out_rd_addr <= in_wb_rdt[11:7];
           out_rs1_addr <= in_wb_rdt[19:15];
           out_rs2_addr <= in_wb_rdt[24:20];

          Type_I <= in_wb_rdt[30:20];
          Type_S <= {in_wb_rdt[30:25], in_wb_rdt[11:7]};
          Type_B <= {in_wb_rdt[7], in_wb_rdt[30:25], in_wb_rdt[11:8], 1'b0};
          Type_U <= {in_wb_rdt[31:12], {12{1'b0}}};
          Type_J <= {in_wb_rdt[19:12], in_wb_rdt[20], in_wb_rdt[30:21], 1'b0};


          imm_31 <= in_wb_rdt[31];
       end
       else begin
       /* En cada ciclo de reloj y cuando la señal in_cnt_en esta activa,
       se realiza un desplazamiento hacia la derecha,
       añadiendo al bit más significativo, el bit de signo */
           if (in_cnt_en ) begin
               Type_I <= {sign_bit, Type_I[10:1]};
               Type_S <= {sign_bit, Type_S[10:1]};
               Type_B <= {sign_bit, Type_B[11:1]};
               Type_U <= {sign_bit, Type_U[31:1]};
               Type_J <= {sign_bit, Type_J[19:1]};
               imm19_15 <= {1'b0, imm19_15[4:1]};
           end
       end
  end

  /*multiplexor encargado de decidir que tipo de instrucción conectar a la salida*/
  always@(*) begin
       casex (in_ctrl)
           4'bx010:  aux_out_imm <= Type_I[0];
           4'bx011:  aux_out_imm <= Type_S[0];
           4'bx1x1:  aux_out_imm <= Type_B[0];
           4'b000x:  aux_out_imm <= Type_U[0];
           4'b1000:  aux_out_imm <= Type_J[0];
           default: aux_out_imm <= 1'b0;
       endcasex
   end

  /*salida secuencial del valor inmediato, cuando cnt_done vale 0,se conecta a la salida la salida es el
valor inmediato, cuando vale 1, es el bit de signo.*/
  assign out_imm = in_cnt_done ? sign_bit : aux_out_imm;
  endmodule
```

Descripción de puertos de entrada:

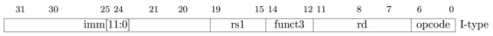
- ***In_clk:*** Señal de reloj, para sincronizar el resto de señales y módulos dentro del procesador. ´
- ***In_csr_imm_en:*** Extensor de signo, e indicador de sí se está trabajando con una instrucción crs.
- ***In_ctrl[3:0]:*** Señal de control de un multiplexor, la cual se encarga de conectar el tipo de instrucción a realizar con la salida out_imm)
- ***In_cnt_done:*** Señal de control que indica el ciclo31, donde si toma el valor 0 se conecta el contenido de los registros internos a out_imm, si toma el valor 1, se conecta el bit de signo.
- ***In_wb_rdt [31:7]:*** Señal de entrada, que contiene bits del 31 al 7 de la instrucción con la que se trabaja. Obteniendo en este rango de bits, el valor inmediato, rs1, rs2 o rd, dependiendo del tipo de instrucción.
- ***In_cnt_en:*** Señal de conteo de una etapa, está en alta durante 32 ciclos.
- ***In_wb_en:*** Permite realizar la carga de los bits de in_wb_rdt, en los registros internos, si está en alta.

Descripció de puertos de salida:

out_imm

Este puerto de salida, es el encargado de ordenar y sacar los valores inmediatos para las operaciones de los siguientes tipos:

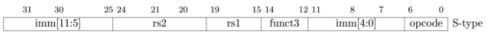
- ***Type I:*** Como se puede observar en la siguiente imagen, el valor inmediato, se distribuye entre los bits 20 y 31, por lo tanto, solo se ven representados por 12 bits. Para llenar los 10 bits restantes (32 en total), se realiza mediante la extensión del signo, tomando el valor del bit más significativo.



En el código Verilog se puede observar que cuando in_wb_en está activa el registro encargado de gestionar las instrucciones Type I, se carga con los valores los bits [30:20] (in_wb_rdt), y realiza el desplazamiento correspondiente hacia la salida out_imm, mediante la señal de control del multiplexor in_ctrl con valor 4'bx010.

```
Type_I <= in_wb_rdt[30:20];
```

- ***Type S:*** Para este tipo de instrucciones, es prácticamente igual, con unos pequeños cambios a la hora de concatenar los valores. Ya que estos están separados en distintas posiciones dentro del registro de instrucción, como se puede observar en la siguiente imagen.

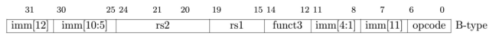


Por lo tanto, para cargar el valor inmediato en el registro interno Type_S, es necesario realizarlo de la siguiente forma en código Verilog.

```
Type_S <= {in_wb_rdt[30:25], in_wb_rdt[11:7]};
```

A partir de aquí, solo es necesario, conectar la salida a Type_S (in_ctrl = x011), y desplazarlo secuencialmente.

- ***Type B:*** Aquí volvemos a encontrar una pequeña diferencia respecto al caso anterior, ya que el bit 0 del valor inmediato, no esta representado en el mismo valor inmediato dentro del registro de instrucción, ya que este no es significativo.

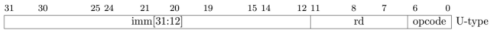


Por lo tanto, lo iniciamos a 0. La declaración queda de la siguiente forma:

```
Type_B <= {in_wb_rdt[7], in_wb_rdt[30:25], in_wb_rdt[11:8], 1'b0};
```

La salida se gestiona con la señal in_ctrl = x1x1

- ***Type U:*** En este caso, los valores del valor inmediato, se distribuyen del 31:12.

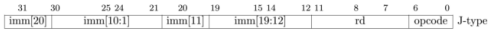


Por lo tanto, para rellenar los bits del 11:0, se han inicializado a 0.

```
Type_U <= {in_wb_rdt[31:12], {12{1'b0}}};
```

La salida se gestiona con la señal in_ctrl = 000x

- ***Type J:*** Como se puede observar en la imagen, los valores inmediatos, organizado de la siguiente forma.



Con el fin de ensamblarlos, se ha realizado las siguiente asignación.

```
Type_J <= {sign_bit, Type_J[19:1]};
```

Fig. 5: Documentación immdec

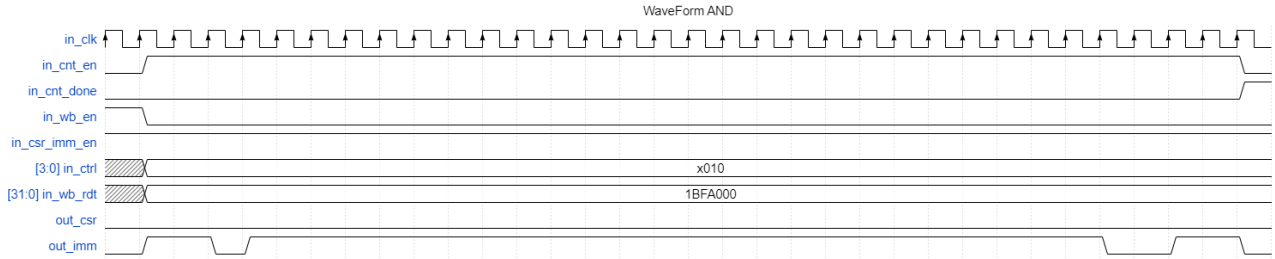


Fig. 6: Simulación imm-dec_Type_I

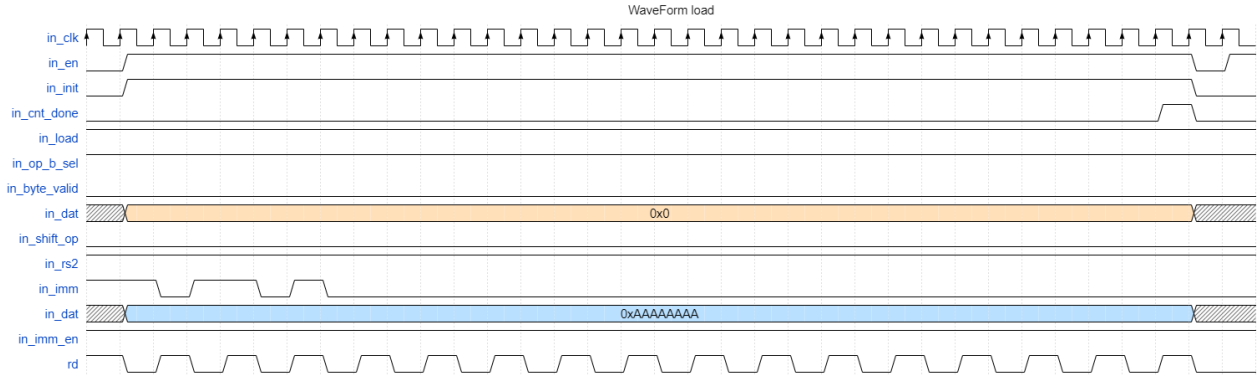


Fig. 7: Simulación load bufreg2

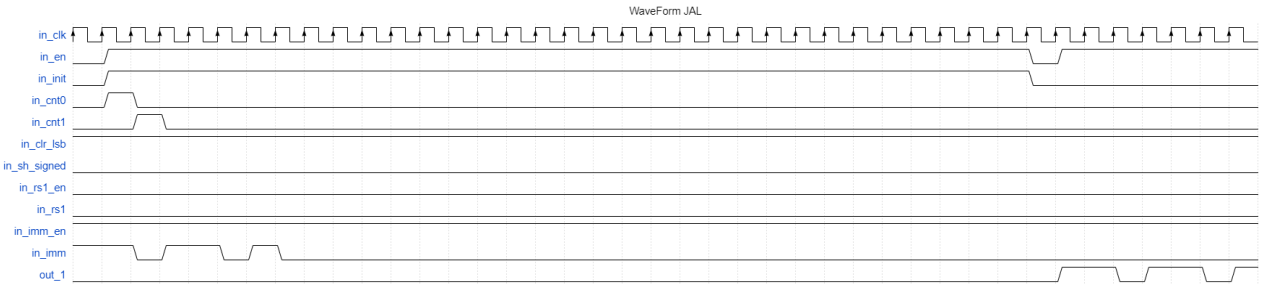


Fig. 8: Simulación jal bufreg