Práctica 1: Microprocesador segmentado

NOTA: Se recomienda encarecidamente leer detenidamente el enunciado entero de cada ejercicio antes de comenzar con la codificación.

El objetivo de esta práctica es implementar la versión segmentada del microprocesador RISC V cuyos detalles se pueden encontrar en el libro de referencia de la asignatura:
"Computer Organization and Design RISC-V edition", por David A. Patterson y John L. Hennessy (chapter 4)
"Digital Design and Computer Architecture : RISC-V Edition", Sarah L. Harris autor (Chapter 6 and 7)
"Guia práctica del RISC V" David Patterson and Andrew Waterman (http://riscvbook.com/)

Esta bibliografía está disponible en la biblioteca de la EPS. Se recomienda seguir el libro para realizar esta práctica. En concreto, se sigue el modelo segmentado del RISC.

Generalidades del diseño

Se pide implementar el microprocesador RISC V en su versión uniciclo el cual servirá de base para su posterior segmentación No es necesario que el procesador soporte el juego de instrucciones completo de RISC V, sino las siguientes instrucciones: **add, addi, and, andi, auipc, beq, bne, j, jal, jalr, li, lw, lui, sw, xor**, cuyos códigos de operación y descripción se incluyen más abajo. En cualquier caso, tras la segmentación, la instrucción *beq*, que implica riesgos de control por ser un salto, funcionará "anómalamente" en esta versión básica del microprocesador.

Para facilitar la realización de este ejercicio, se proporciona una versión simplificada del procesador RISC-V en versión uniciclo con los diferentes sub-bloques del procesador ya definidos: el banco de registros, la unidad aritmética lógica (ALU), la unidad de control (óvalo "Control" en las figuras 2a, 2b y 3), el bloque que genera los códigos de control para la ALU (óvalo "ALU control"), generador de operando inmediato ("Imm Gen") y el propio procesador. En el ejercicio 3, los alumnos deberán añadir el código VHDL necesario para segmentar el procesador.

Por otra parte, para verificar el funcionamiento del procesador se entrega un fichero VHDL completo con un testbench muy simple que instancia el micro y las memorias de instrucciones y datos, para las cuales se entrega también completo un fichero VHDL.

La relación jerárquica en el diseño es pues la siguiente (ver figura 1):

- El testbench (*processorR5_tb*), instancia al procesador (*processor*) y a las 2 memorias (*memory_ **).
- El procesador (*processorRV*) instancia el banco de registros (*reg_bank*), la ALU (*aluRV*), la unidad de control (*control_unit*), el generador de inmediatos (imm_gen) y el bloque para el control de la ALU (*alu_control*). Adicionalmente la declaración de constantes se concentra en un paquete (RISCV_pack).

El fichero runsim_arq.do que permite lanzar la simulación invocando este script Modelsim/Questasim desde la consola del simulador. Este fichero compila, simula y abre las formas de onda descriptas en wave_arq.do

Todos los registros utilizados deben usar el flanco de subida de reloj y *reset* asíncrono activo a nivel alto. Por otra parte, para la codificación del procesador se deberá considerar lo siguiente:

- No crear componentes adicionales: hacer el contador de programa, los registros del pipeline, multiplexores, etc. como código dentro de la arquitectura de *processorRV*.
- Utilizar asignaciones concurrentes (simples y condicionales).

El material se entrega con la siguiente estructura, que deberá mantenerse:

- Directorio *rtl/* : contiene el código del procesador:

processorRV.vhd	Procesador uniciclo, a completar para segmentar
alu_RV.vhd	*ALU para RISC-V, completa*
reg_bank.vhd	*Banco de registros, completo*
control_unit.vhd	*Unidad de control, completo*
alu_control.vhd	*Control de la ALU, completo*
imm_gen.vhd	*Generador de inmediato, completo*
RISCV_pack.vhd	*Package con definiciones comunes*

- Directorio *sim/* : contiene lo necesario para simular el procesador (todos los ficheros están completos salvo *wave_arq.do*, donde los alumnos podrán grabar la configuración de ondas que deseen):

processorR5_tb.vhd	Testbench
datos.txt	*Fichero de datos para la memoria de datos*
Instrucciones.txt	*Fichero de datos para la memoria de instrucciones*
memory_datos.vhd	*Modelo simple de memoria síncrona de datos*
memory_instr.vhd	*Modelo simple de memoria síncrona de instrucciones*
riscv_pruBasico.asm	*Código fuente de un programa ensamblador de prueba*
riscv_Fibonacci.asm	*Código fuente de un programa ensamblador de prueba*
programa_test.asm	*Código fuente de un programa ensamblador de prueba*
runsim_arq.do	*Script de simulación para ModelSim/Questa*
wave.do	*Script de configuración de ondas paras ModelSim/Questa*

El testbench instancia las memorias, las cuales leen, en una fase de inicialización en tiempo = 0, sus datos desde los ficheros "*instrucciones.txt*" y "*datos.txt*" (nótese que estas memorias son modelos que simplifican lo que sería un escenario con memorias reales). Estos dos ficheros resultan del ensamblado del programa ensamblador. Para generar el contenido de las instrucciones y datos utilizaremos el simulador RARS *(RISC-V Assembler and Runtime Simulator)*

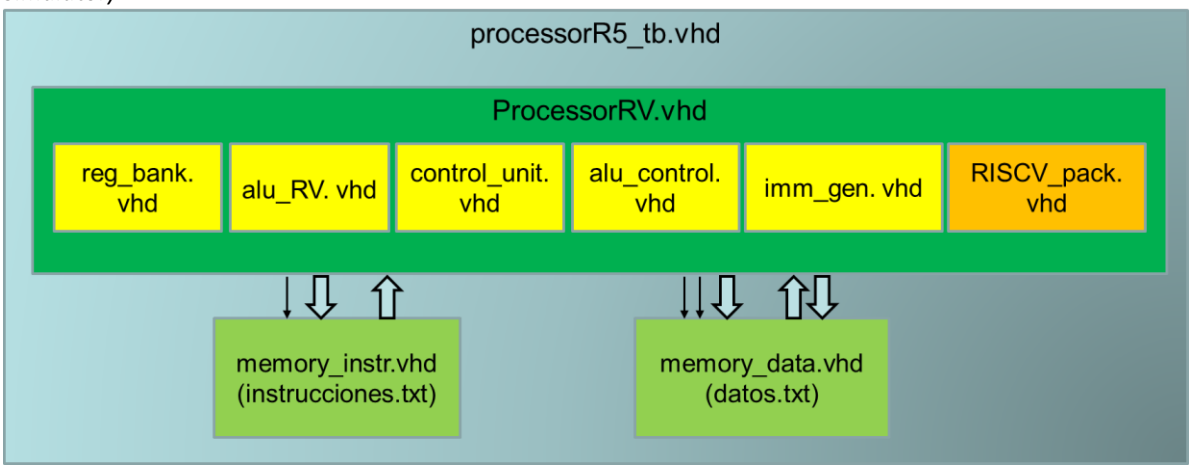
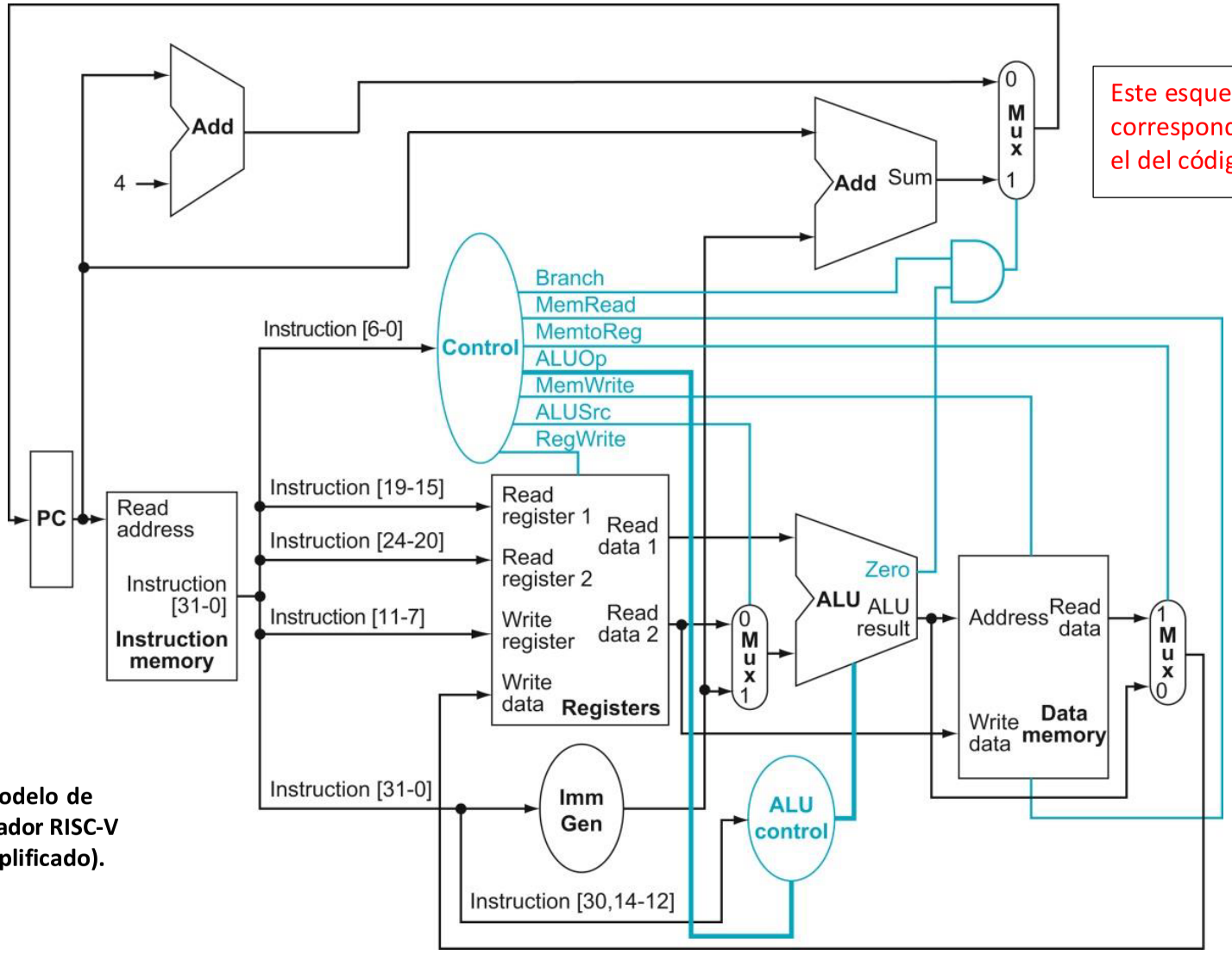


Figura 1. Jerarquía de ficheros VHDL en el diseño.

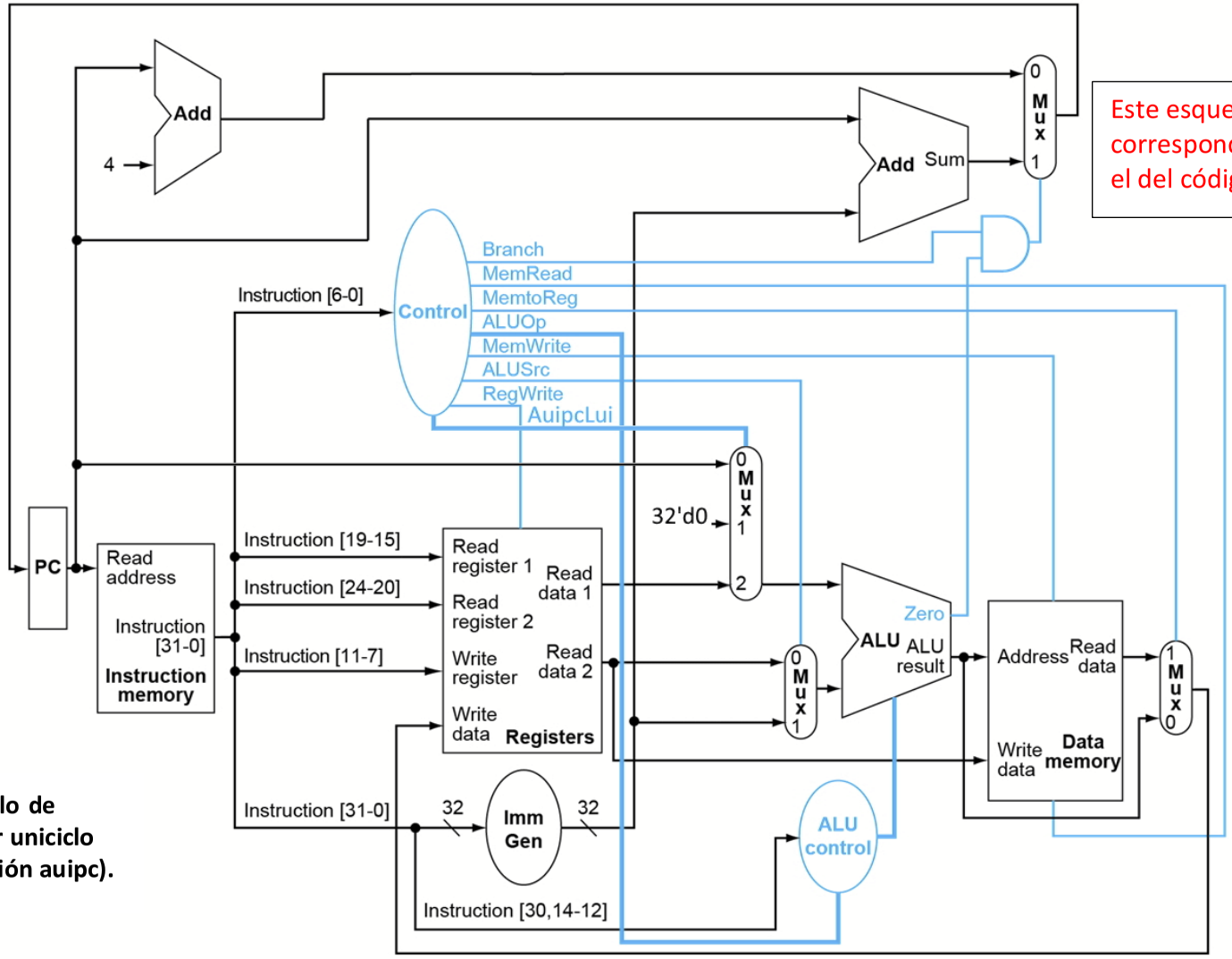
El esquema del procesador uniciclo a desarrollar es el indicado en la siguiente figura:

NOTA: Para la realización de este ejercicio se pueden revisar y reutilizar las prácticas realizadas a lo largo de la asignatura "Estructura de Computadores" (del segundo cuatrimestre del primer curso). Pero se deben utilizar los ficheros provistos.



Este esquema NO se corresponde exactamente con el del código entregado

Figura 2a. Modelo de microprocesador RISC-V uniciclo (simplificado).



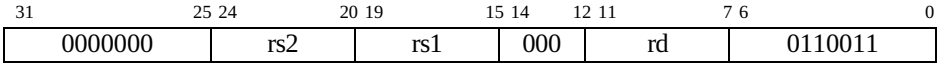
Este esquema NO se corresponde exactamente con el del código entregado

Figura 2b. Modelo de microprocesador uniciclo (incluye instrucción auipc).

Las instrucciones soportadas tienen la siguiente descripción:

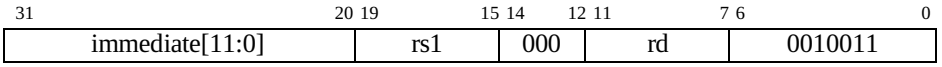
add rd, rs1, rs2 x[rd] = x[rs1] + x[rs2]

Add. Tipo R, RV32I y RV64I.
Suma el registro x[*rs2*] al registro x[*rs1*] y escribe el resultado en x[*rd*]. Overflow aritmético ignorado.
Formas comprimidas: **c.add** rd, rs2; **c.mv** rd, rs2



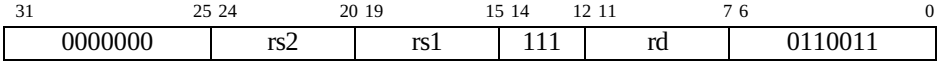
addi rd, rs1, immediate `x[rd] = x[rs1] + sext(immediate)`

Add Immediate. Tipo I, RV32I y RV64I.
Suma el *inmediato* sign-extended al registro x[*rs1*] y escribe el resultado en x[*rd*]. Overflow aritmético ignorado.
Formas comprimidas: **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm



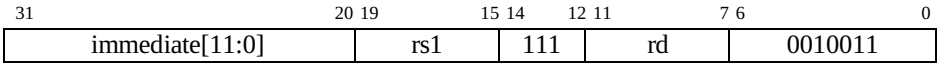
and rd, rs1, rs2 x[rd] = x[rs1] & x[rs2]

AND. Tipo R, RV32I y RV64I.
Calcula el AND a nivel de bits de los registros x[*rs1*] y x[*rs2*] y escribe el resultado en x[*rd*].
Forma comprimida: **c.and** rd, rs2



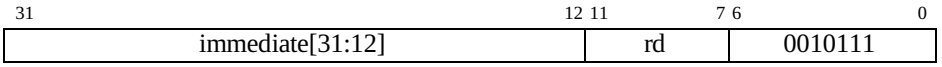
andi rd, rs1, immediate `x[rd] = x[rs1] & sext(immediate)`

AND Immediate. Tipo I, RV32I y RV64I.
Calcula el ANDa nivel de bits del *inmediato* sign-extended y el registro x[*rs1*] y escribe el resultado en x[*rd*].
Forma comprimida: **c.andi** rd, imm



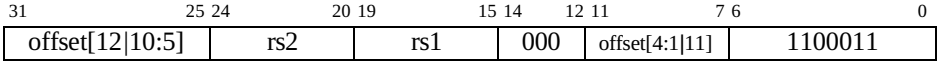
auipc rd, immediate `x[rd] = pc + sext(immediate[31:12] << 12)`

Add Upper Immediate to PC. Tipo U, RV32I y RV64I.
Suma el *inmediato* sign-extended de 20 bits, corrido a la izquierda por 12 bits, al *pc*, y escribe el resultado en x[*rd*].



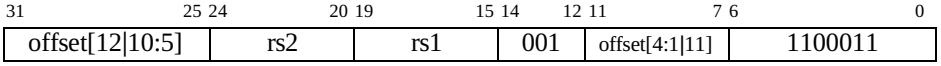
beq rs1, rs2, offset `if (rs1 == rs2) pc += sext(offset)`

Branch if Equal. Tipo B, RV32I y RV64I.
Si el registro x[*rs1*] es igual al registro x[*rs2*], asignar al *pc* su valor actual más el *offset* sign- extended.
Forma comprimida: **c.beqz** rs1,offset



bne rs1, rs2, offset `if (rs1 = rs2) pc += sext(offset)`

Branch if Not Equal. Tipo B, RV32I y RV64I.
Si el registro x[*rs1*] no es igual al registro x[*rs2*], asignar al *pc* su valor actual más el *offset* sign-extended.
Forma comprimida: **c.bnez** rs1, offset



j offset

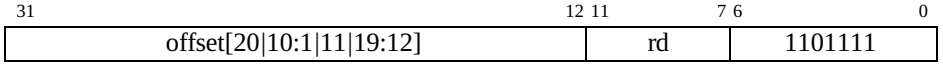
`pc += sext(offset)`

Jump. Pseudoinstrucción, RV32I y RV64I.
Escribe al *pc* su valor actual más el *offset* extendido en signo. Se extiende a **jal** x0, offset.

jal rd, offset

`x[rd] = pc+4; pc += sext(offset)`

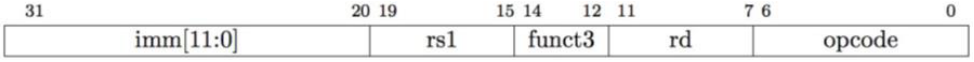
Jump and Link. Tipo J, RV32I y RV64I.
Escribe la dirección de la siguiente instrucción (*pc*+4) en x[*rd*], luego asigna al *pc* su valor actual más el *offset* extendido en signo. Si *rd* es omitido, se asume x1.
Formas comprimidas: **c.j** offset; **c.jal** offset



jalr rd, rs1,offset

`x[rd] = pc+4; pc = x[rs1] + sext(offset)`

Jump and Link Register. Tipo I, RV32I y RV64I.
Escribe la dirección de la siguiente instrucción (*pc*+4) en x[*rd*], luego asigna al *pc* el valor de un registro (*rs1*) más el *offset* extendido en signo. Si *rd* es omitido, se asume x1.



li rd, immediate

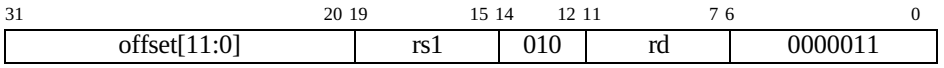
`x[rd] = immediate`

Load Immediate. Pseudoinstrucción, RV32I y RV64I.
Carga una constante en x[*rd*], usando tan pocas instrucciones como sea posible. Para RV32I, se extiende a **lui** y/o **addi**; para RV64I, es tan largo como **lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**.

lw rd, offset(rs1)

`x[rd] = sext(M[x[rs1] + sext(offset)][31:0])`

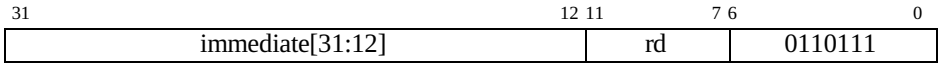
Load Word. Tipo I, RV32I y RV64I.
Carga cuatro bytes de memoria en la dirección x[*rs1*] + *sign-extend*(*offset*) y los escribe en x[*rd*]. Para RV64I, el resultado es extendido en signo.
Formas comprimidas: **c.lwsp** rd, offset; **c.lw** rd, offset(rs1)



lui rd, immediate

`x[rd] = sext(immediate[31:12] << 12)`

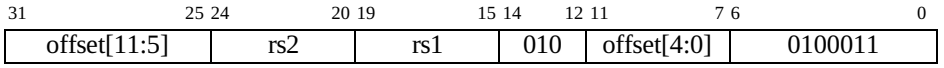
Load Upper Immediate. Tipo U, RV32I y RV64I.
Escribe el *inmediato* de 20 bits extendido en signo, corrido a la izquierda por 12 bits, en x[*rd*], volviendo cero los 12 bits más bajos.
Forma comprimida: **c.lui** rd, imm



sw rs2, offset(rs1)

`M[x[rs1] + sext(offset)] = x[rs2][31:0]`

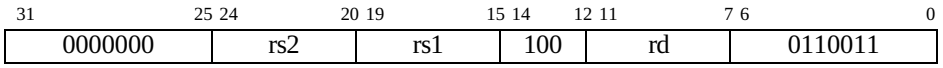
Store Word. Tipo S, RV32I y RV64I.
Almacena los cuatro bytes menos significativos del registro x[*rs2*] a memoria en la dirección x[*rs1*] + *sign-extend*(*offset*).
Formas comprimidas: **c.swsp** rs2, offset; **c.sw** rs2, offset(rs1)



xor rd, rs1, rs2

x[rd] = x[rs1] ^ x[rs2]

Exclusive-OR. Tipo R, RV32I y RV64I.
Calcula el OR exclusivo a nivel de bits de los registros x[*rs1*] y x[*rs2*] y escribe el resultado en x[*rd*].
Forma comprimida: **c.xor** rd, rs2



La instrucción NOP. La instrucción NOP no viene definida en el juego de instrucciones del RISC V, sino que se trata de una psudo-instruccion que se traduce en un addi x0, x0, 0

Ejercicio 1 (0.5 puntos)

Se pide revisar el microprocesador RISC V en su versión uniciclo. El diseño entregado como punto de partida, debería soportar el set de instrucciones descripto (**add, addi, and, andi, auipc, beq, bne, j, jal, jalr, li, lw, lui, sw, xor**). ¿están todas las instrucciones soportadas?

Ejercicio 2 (1.5 puntos)

Se pide realizar un programa en ensamblador simple que sea capaz ordenar una lista de 10 números enteros que se encuentran en memoria. El resultado ordenado debe quedar en otra posición de memoria. Puede usar cualquier algoritmo de ordenación. Compruebe la ejecución con RARS y la simulación RTL.

Ejercicio 3 (8 puntos)

Se pide implementar el microprocesador RISC V en su versión segmentada. **El resultado debe ser un micro capaz de realizar en el caso ideal (sin riesgos) una instrucción por ciclo de reloj**. Para el ejercicio básico, no es necesario que el modelo soporte riesgos (salvo el estructural de acceso a memorias separadas de instrucciones y datos).

Todos los registros utilizados para la segmentación han de cumplir las siguientes características:

1. Funcionar por flanco de subida del reloj (`rising_edge(clk)`).
2. *Resetearse* **asíncronamente** utilizando la señal "`Reset`".

Se recomienda utilizar como guía el esquema reflejado en la siguiente figura (ligeramente diferente al del libro).

Convenios de codificación:

Los siguientes convenios de codificación son de carácter obligatorio. Se pueden utilizar criterios alternativos los cuales deben ser justificados y/o convenidos con el profesor de su grupo de prácticas.

Cod1. Separación etapas, tanto procesos, asignaciones, así como instanciaciones.

Utilizar línea de comentario ("----------…") separando etapas del pipeline (Fetch, Decode,…). Ejemplo:

```
-------------------------------------------------------------------------
-- IF stage
PC_next <= . . .
. . .
-------------------------------------------------------------------------
--  Pipeline reg: IF/ID

process (Clk,…
. . .
```

Cod2. Nomenclatura señales en cada etapa de la segmentación (Pipeline).

Llamar a los registros del pipeline *<siguiente_etapa>. P.ej.: PC_plus4_ID para un registro de IF/ID, PC_plus4_EX para un registro de ID/EX…

Para las señales generadas en cierta etapa (entre registros de pipeline). Usar el mismo criterio de nombrarles con el sufijo de esa etapa.

Nota: Se recomienda enfáticamente disponer de una versión impresa del esquema para realizar anotaciones sobre el dibujo (o la correspondiente aplicación digital) donde apuntar los nombres de señales que se vayan generando.

Este esquema NO se corresponde exactamente con el del código entregado

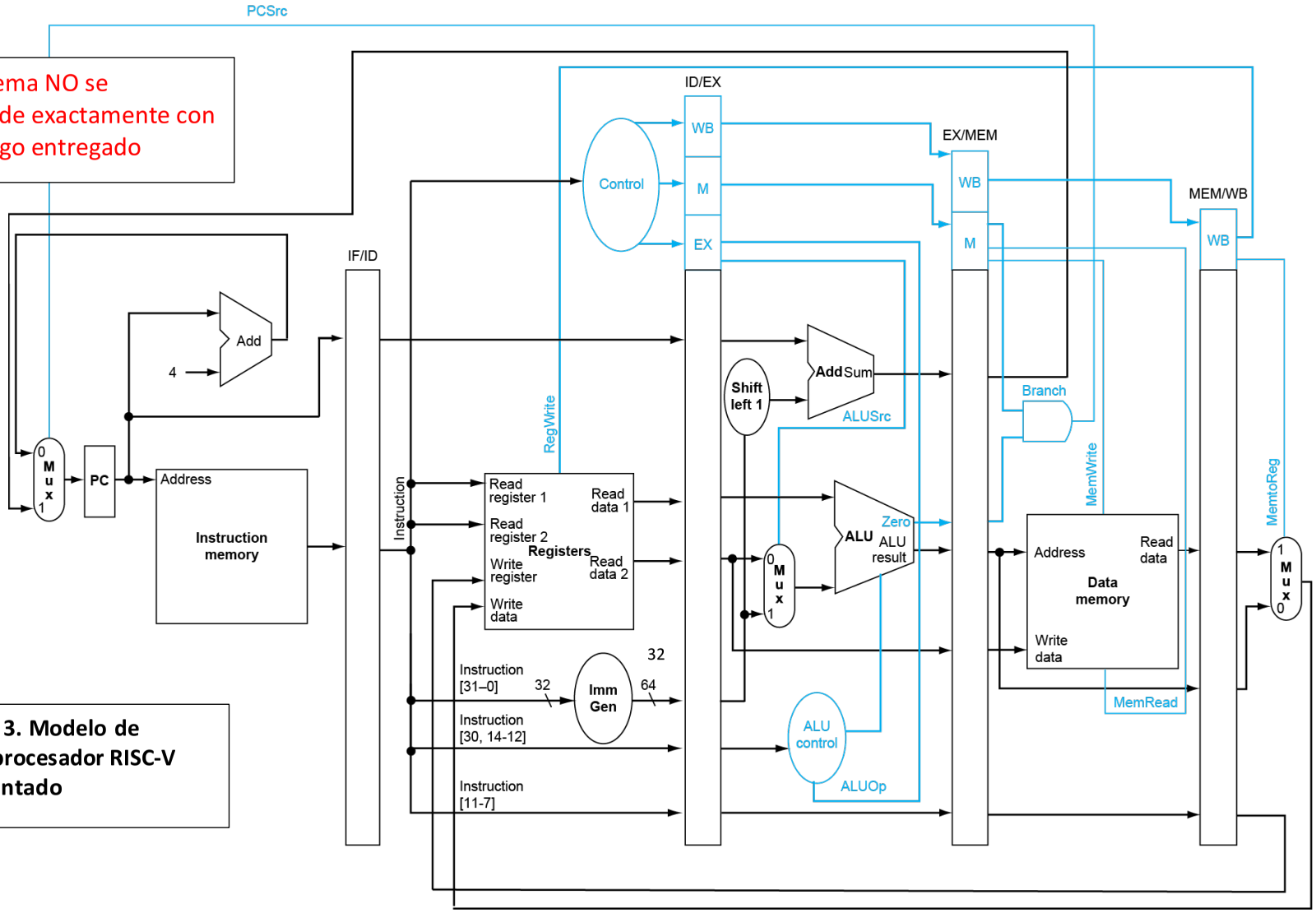


Figura 3. Modelo de microprocesador RISC-V segmentado

MATERIAL A ENTREGAR (IMPORTANTE RESPETAR EL FORMATO)

Los ficheros VHDL del ejercicio 3 y el ensamblador del ejercicio 2.
Un fichero de texto plano con nombre P1_grupoxxxx.txt que indique número de pareja, integrantes y cualquier aclaración que considere necesario para poder corregir la práctica.

Organizarlo en carpetas, una por los ejercicios 1-2 y otra para el ejercicio 3.

La entrega se realizará a través de Moodle, con fecha tope el viernes de la última semana asignada a esta práctica hasta las 23:59 de la noche. (consultar en Moodle fecha entrega)

*El profesor de cada grupo de prácticas podrá requerir una defensa, la cual es parte de la nota de la práctica

AYUDAS Y AVISOS

El compilador y emulador RARS (ejecutable Java) permite ensamblar y ejecutar código RISC-V en un entorno emulado. El contenido de la memoria de datos e instrucciones que se exporta es directamente aceptado por memorias VHDL en la simulación.

Los ficheros "`instrucciones.txt`" y "`datos.txt`" que contienen las memorias de instrucciones y datos respectivamente deben localizarse en el mismo directorio desde donde se lance la simulación del proyecto. Si no fuera así, se puede dar la ruta completa de dichos ficheros cambiando el script *runsim_arq.do*.