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 una serie de mejoras sobre el microprocesador RISC-V (visto en clase de teoría). En esta práctica se resolverán los riesgos (*"hazards"*) generados al segmentar el procesador RISC-V. El punto de partida es una versión segmentada del microprocesador RISC-V cuyos detalles se pueden encontrar en los libros de referencia de la asignatura:
"Digital Design and Computer Architecture: RISC-V Edition", Sarah L. Harris autor (Chapter 6 and 7)
"Computer Organization and Design RISC-V edition", por David A. Patterson y John L. Hennessy (chapter 4)
"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

El punto de partida es una implementación del microprocesador RISC-V en su versión pipeline (segmentada) sin resolver los riesgos de datos y control (también se provee como referencia la versión uniciclo que varía levemente de la utilizada en la asignatura previa de Estructura de Computadores - EC)

El procesador no soporta el juego de instrucciones completo de RISC V, sino las siguientes instrucciones: **add, addi, and, xor**, **andi, auipc, beq, bne, blt, bge, j, jal, jalr, li, lw, lui, sw,** cuyos códigos de operación y descripción se incluyen más abajo. En cualquier caso, tras la segmentación, las instrucciones de salto **beq, bne, blt, bge**, que implica riesgos de control por ser un salto, funcionará "anómalamente" en esta versión básica del microprocesador.

Como punto de partida, 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 ("Control Unit" en las figuras 2a, 2b y 3), el bloque que genera los códigos de control para la ALU ("ALU control"), generador de operando inmediato ("Imm Gen") y el propio procesador.

Y una versión segmentada (figura 2b) totalmente funcional para el mismo set de instrucciones y usando los mismos componentes que la versión uniciclo que no tiene resuelto los riesgos de datos y control. Es decir, ante un salto (Jump o Branch), un riesgo RAW o un Load-Use el comportamiento puede ser erróneo

Por otra parte, para verificar el funcionamiento del procesador se entrega un fichero VHDL con un banco de pruebas (testbench) muy simple que instancia el microprocesador RISC-V, la memoria de instrucciones y la de datos, para las cuales se entrega también el correspondiente 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).

Los fichero runsim_arq_*.do (runsim_arq_uniciclo.do, runsim_arq_pip_Norisgo.do, runsim_arq_pipe.do) que permiten 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

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

- Directorio asm/ : contiene el código del procesador:

riscv_pruBasico.asm	*Código fuente de un programa ensamblador de prueba*
riscv_fibonacci.asm	*Código fuente de un programa ensamblador de prueba*
riscv_test.asm	*Código fuente de un programa ensamblador de prueba*

- Directorio rtl/ : contiene el código del procesador:

processorRV_uniciclo.vhd	Procesador uniciclo (top Level)
processorRV_pipeNoRiesgos.vhd	Procesador Pipeline sin soporte de riesgos
processorRV_Pipe.vhd	Procesador Pipeline para completar por el estudiante
alu_control.vhd	*Control de la ALU, completo*
alu_RV.vhd	*ALU para RISC-V, completa*
control_unit.vhd	*Unidad de control, completo*
hazard_Unit.vhd	*Unidad de riesgos a completar por el estudiante*
imm_gen.vhd	*Generador de inmediato, completo*
reg_bank.vhd	*Banco de registros, 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*
runsim_arq_pipe.do	*Script de simulación para Questasim Procesador pipeline*
runsim_arq_pipNoriesgo.do	*Script de simulación Procesador pipeline sin solución risgos*
runsim_arq_unic.do	*Script de simulación para QuestaSim Procesador uniciclo*
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*" (estas memorias son modelos que simplifican 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)*

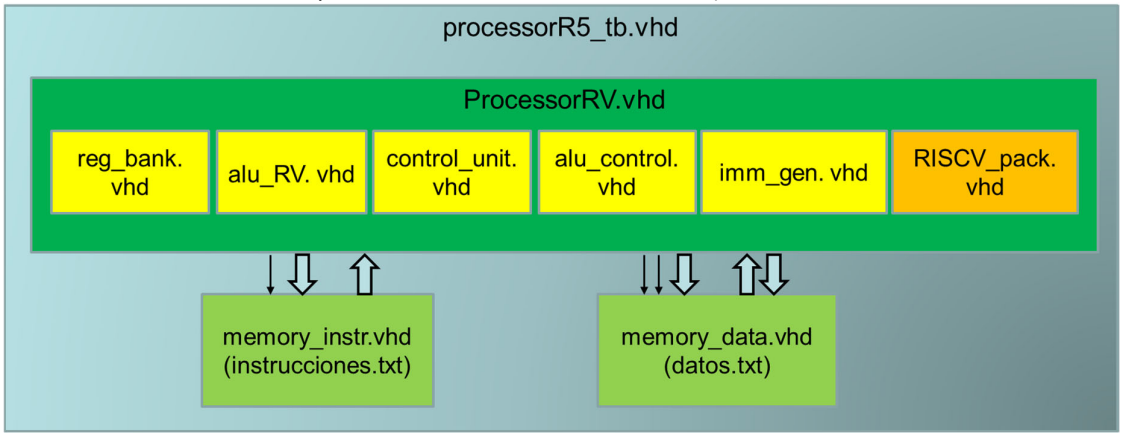


Figura 1. Jerarquía de ficheros VHDL en el diseño.

El esquema del procesador uniciclo y pipeline sin soporte de riesgos se presentan en las figuras 2a y 2b respectivamente

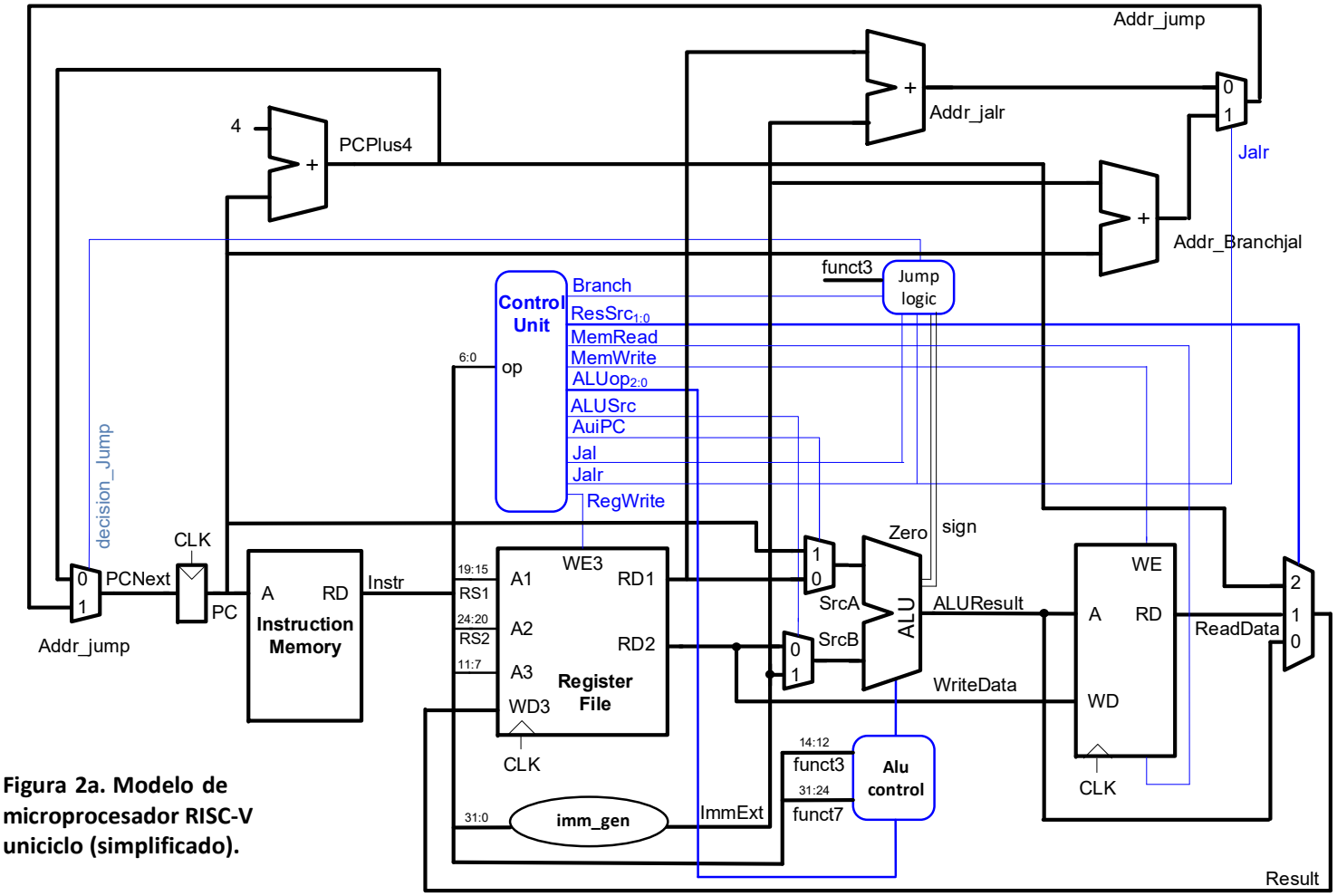


Figura 2a. Modelo de microprocesador RISC-V uniciclo (simplificado).

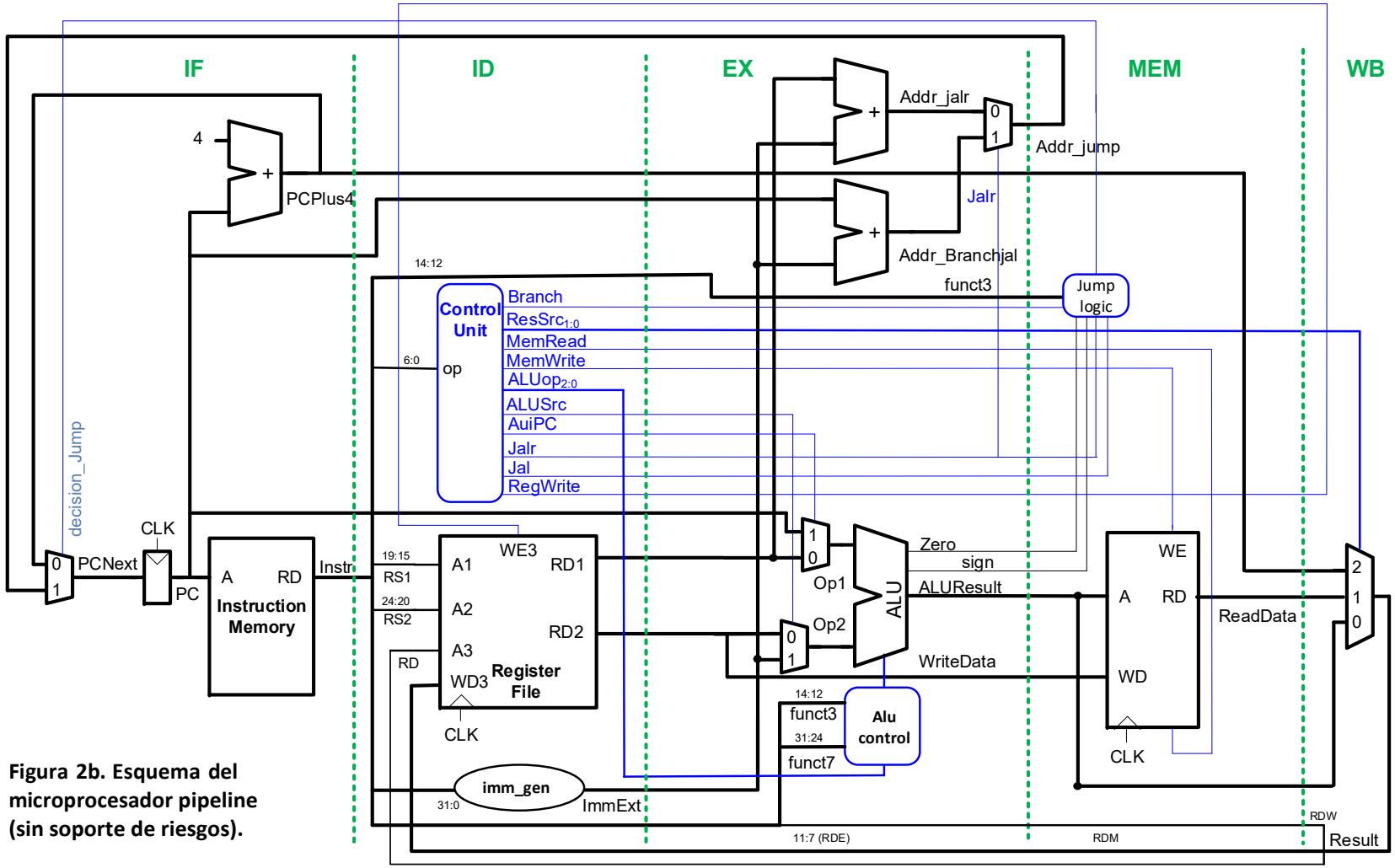
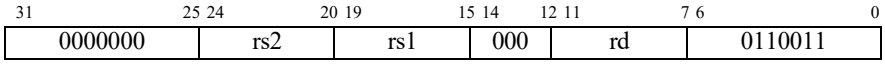


Figura 2b. Esquema del microprocesador pipeline (sin soporte de riesgos).

Las instrucciones soportadas tienen la siguiente descripción:

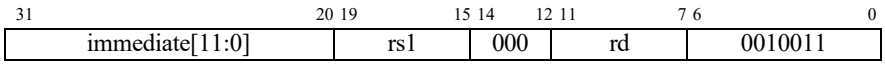
add rd, rs1, rs2 x[rd] = x[rs1] + x[rs2]

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



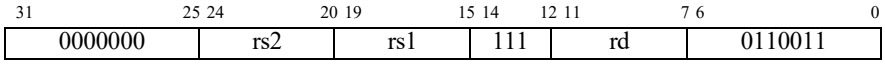
addi rd, rs1, immediate `x[rd] = x[rs1] + sext(immediate)`

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



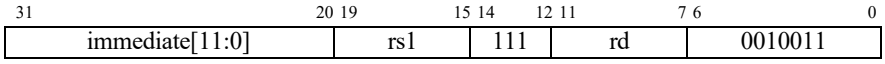
and rd, rs1, rs2 x[rd] = x[rs1] & x[rs2]

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



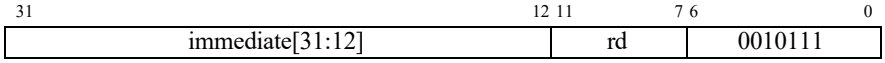
andi rd, rs1, immediate `x[rd] = x[rs1] & sext(immediate)`

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



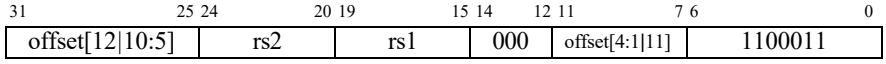
auipc rd, immediate `x[rd] = pc + sext(immediate[31:12] << 12)`

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*].



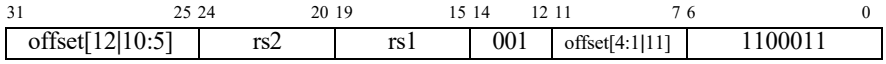
beq rs1, rs2, offset `if (rs1 == rs2) pc += sext(offset)`

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

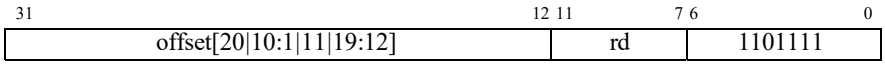


j offset `pc += sext(offset)`

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

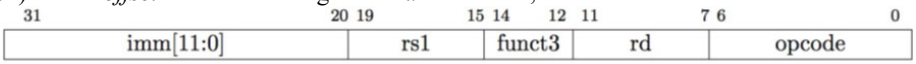
jal rd, offset `x[rd] = pc+4; pc += sext(offset)`

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.

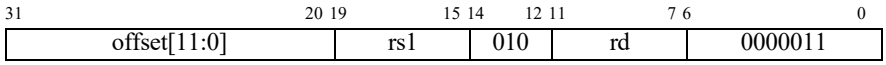


li rd, immediate `x[rd] = immediate`

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**.

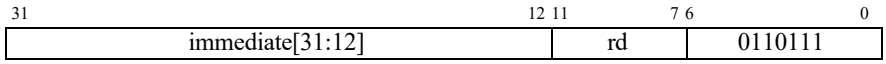
lw rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][31:0])`

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)



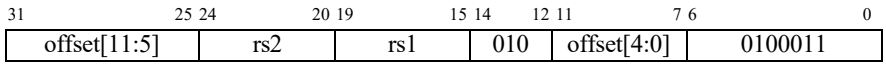
lui rd, immediate `x[rd] = sext(immediate[31:12] << 12)`

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



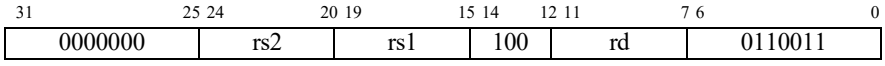
sw rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][31:0]`

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 (2 puntos) Puesta en marcha y pruebas de ensamblador.

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 del procesador uniciclo.

Realice las modificaciones necesarias al programa en ensamblador para que pueda ejecutarse correctamente en el procesador pipeline sin soporte de riesgos. Es decir, resuelva los riesgos de datos y control por software.

Ejercicio 2 (6 puntos) resolución riesgos de datos

Las modificaciones se harán en el fichero denominado "rtl/procesadorRV_pipe.vhd" y se usará el script de simulación provisto bajo el nombre "sim/runsim_arq_pipe.do". Un esquema simplifacado puede verse en la figura 5.

1. ***Forwarding de datos hacia la ALU.***

Sobre el diseño de microprocesador anterior se añadirán los caminos de adelantamiento de datos ("forwarding") mostrados en la siguiente figura, que adelantarán el resultado de la instrucción anterior (desde MEM) o desde dos anteriores (desde WR) para uso en la etapa "EX".

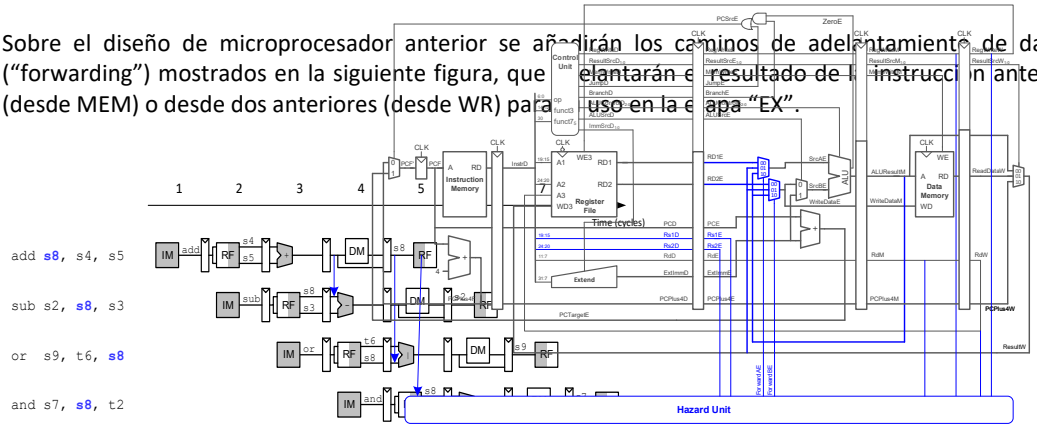


Figura 3. Descripción del problema a resolver con el adelantamiento y esquema de solución.

2. ***Detección del caso en que una instrucción LW carga un registro que es utilizado por la instrucción que le sigue (LW-Use).***

En este caso no es posible adelantar datos. Debe generarse un ciclo de detención ("stall"), repitiendo las etapas IF e ID actuales e insertando una "burbuja" (a modo de instrucción *nop*) en las etapa EX

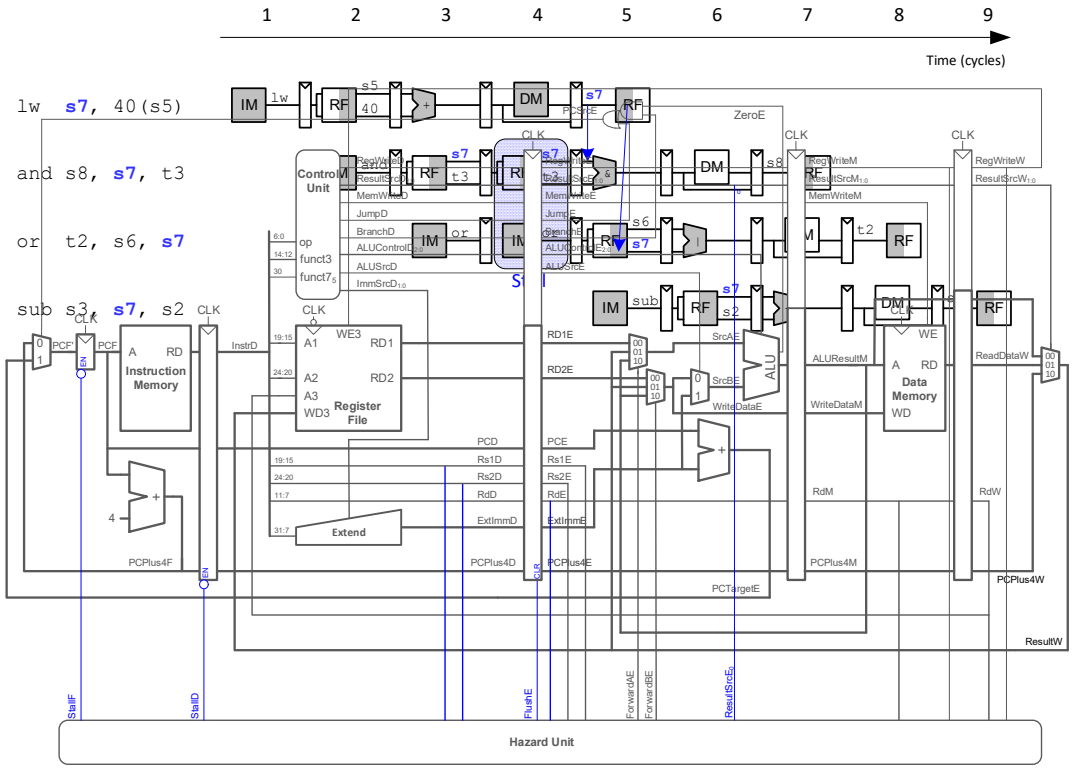
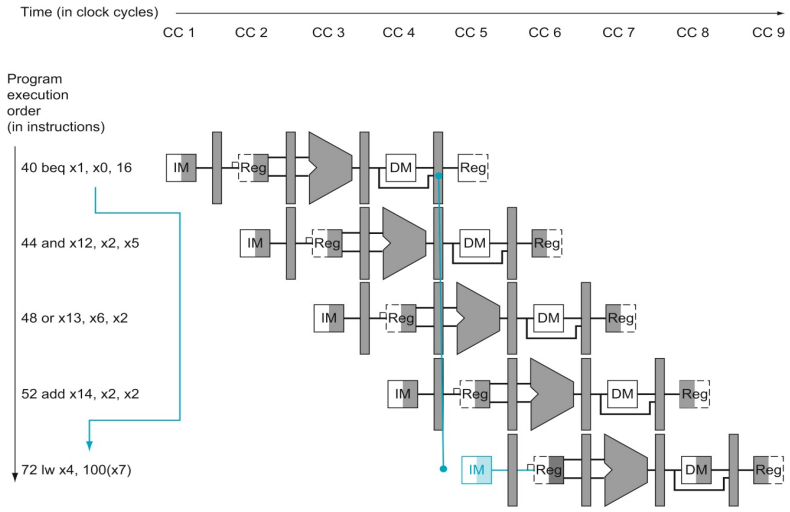


Figura 4. Riesgo por LW-USE y esquemas de solución.

Ejercicio 3: (2 puntos) Riesgos en Control

En diseño de partida no se han tenido en cuenta los riesgos producidos por las instrucciones de salto (*branch*). En este ejercicio el procesador debe ser modificado para ejecutar correctamente la instrucción *branch* ante cualquier condición del programa. La implementación provista resuelve el salto en la etapa MEM (la descripción del libro de Harris lo hace en EX). El procesador ante un salto efectivo tiene que eliminar las 3 instrucciones subsiguientes.



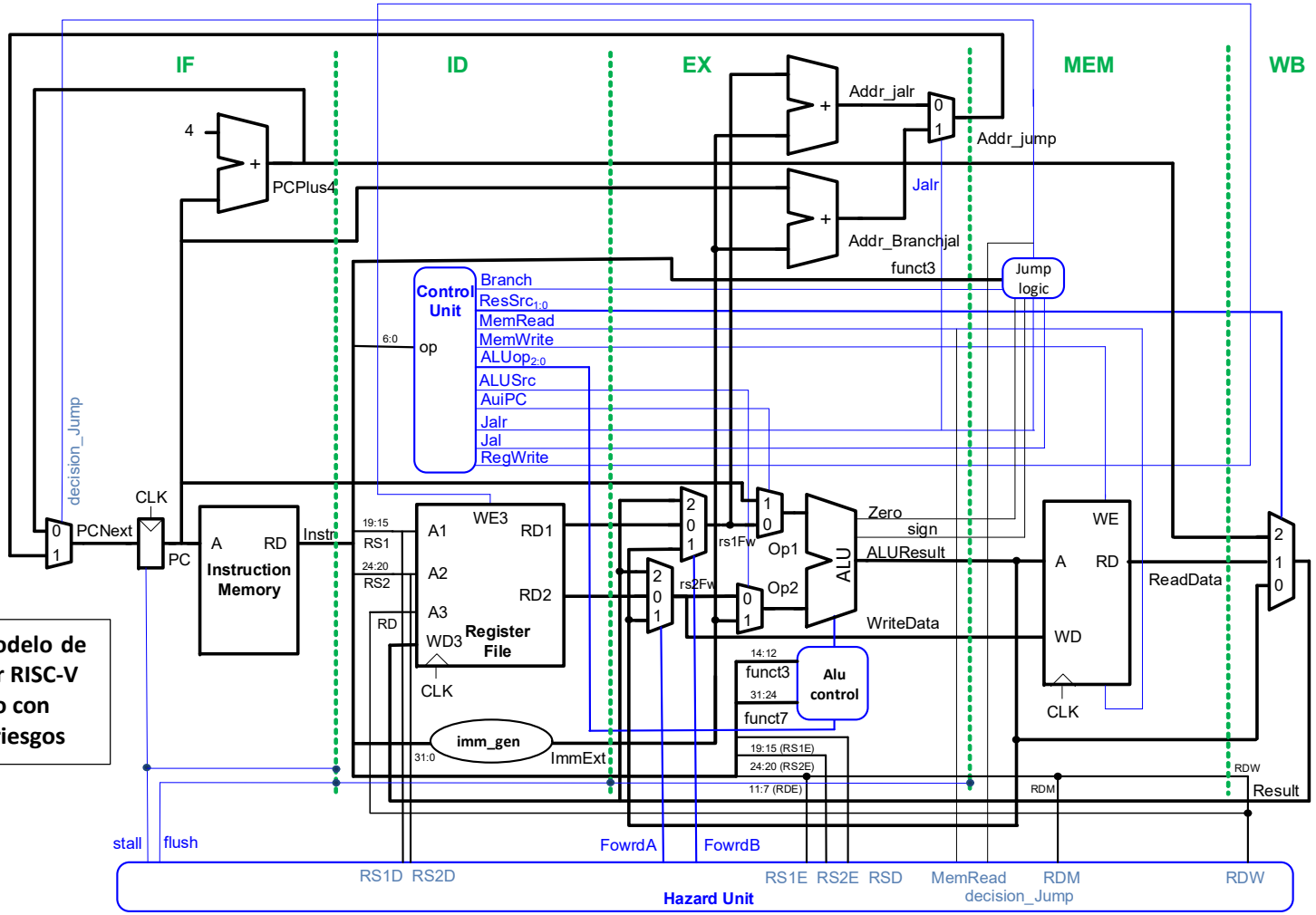


Figura 5. Modelo de µprocesador RISC-V segmentado con soporte de riesgos

Ejercicio 4. opcional: (1 puntos) reducir penalización en saltos

Solo se corrige si el ejercicio 3 está correctamente realizado.
Modificar el diseño del procesador para que ejecute el salto en la etapa EX en vez de MEM ahorrando un ciclo en los saltos efectivos.
Generar un nuevo fichero de microprocesador denominado "rtl/procRV_pipe_branchEX.vhd" y crear el script de simulación el nombre "sim/runsim_arq_branchEX.do"

MATERIAL A ENTREGAR (IMPORTANTE RESPETAR EL FORMATO)

Los ficheros VHDL del ejercicio 2 y 3 y el ensamblador del ejercicio 1.
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.

Entregar la misma organización de carpetas que el provisto. Es decir /asm, /rtl, /sim.

La entrega se realizará a través de Moodle, con fecha límite 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*.