

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)

“Guía 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, 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 (“Inm Gen”) y el propio procesador. Los alumnos deberán completar estos ficheros escribiendo el código de las correspondientes arquitecturas VHDL.

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 (*Inm_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 *runsimsim_arq.do* que permite lanzar la simulación invocando este script Modelsim/Questasim desde la consola de modelsim. Este fichero compila, simula y abre las formas de onda descritas 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 recomienda:

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

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

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

<i>processorRV.vhd</i>	procesador, a completar
<i>alu_RV.vhd</i>	ALU para RISC-V, completa
<i>reg_bank.vhd</i>	Banco de registros, completo
<i>control_unit.vhd</i>	Unidad de control, a completar
<i>alu_control.vhd</i>	Control de la ALU, a completar
<i>Inm_generator.vhd</i>	Generador de inmediato, completo
<i>RISCV_pack.vhd</i>	Package con definiciones comunes

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

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

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 de *programa.asm*. Para generar el contenido de las instrucciones y datos utilizaremos el simulador RARS (*RISC-V Assembler and Runtime Simulator*)

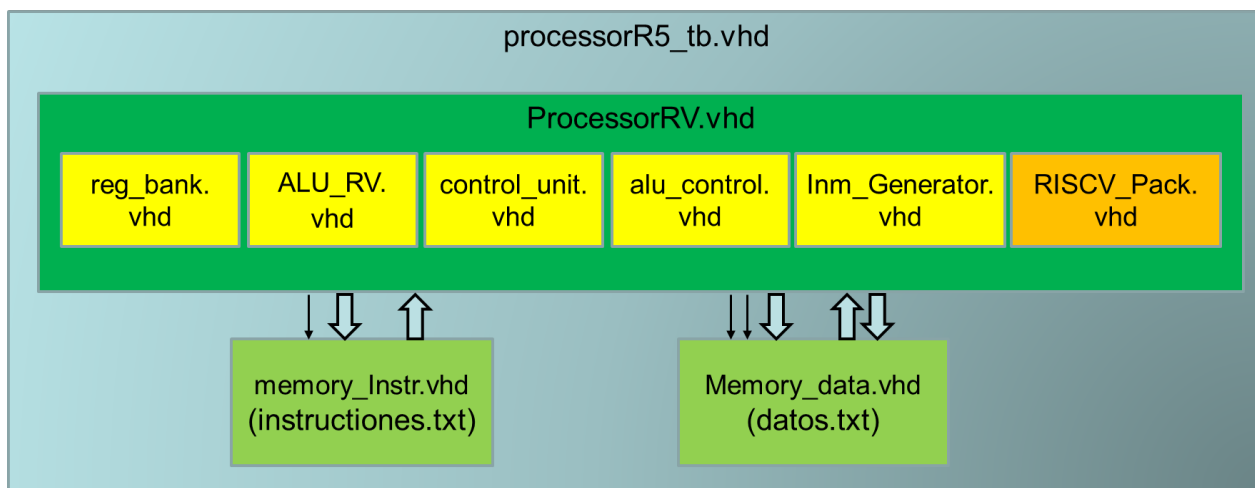


Figura 1. Jerarquía de ficheros VHDL en el diseño.

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.

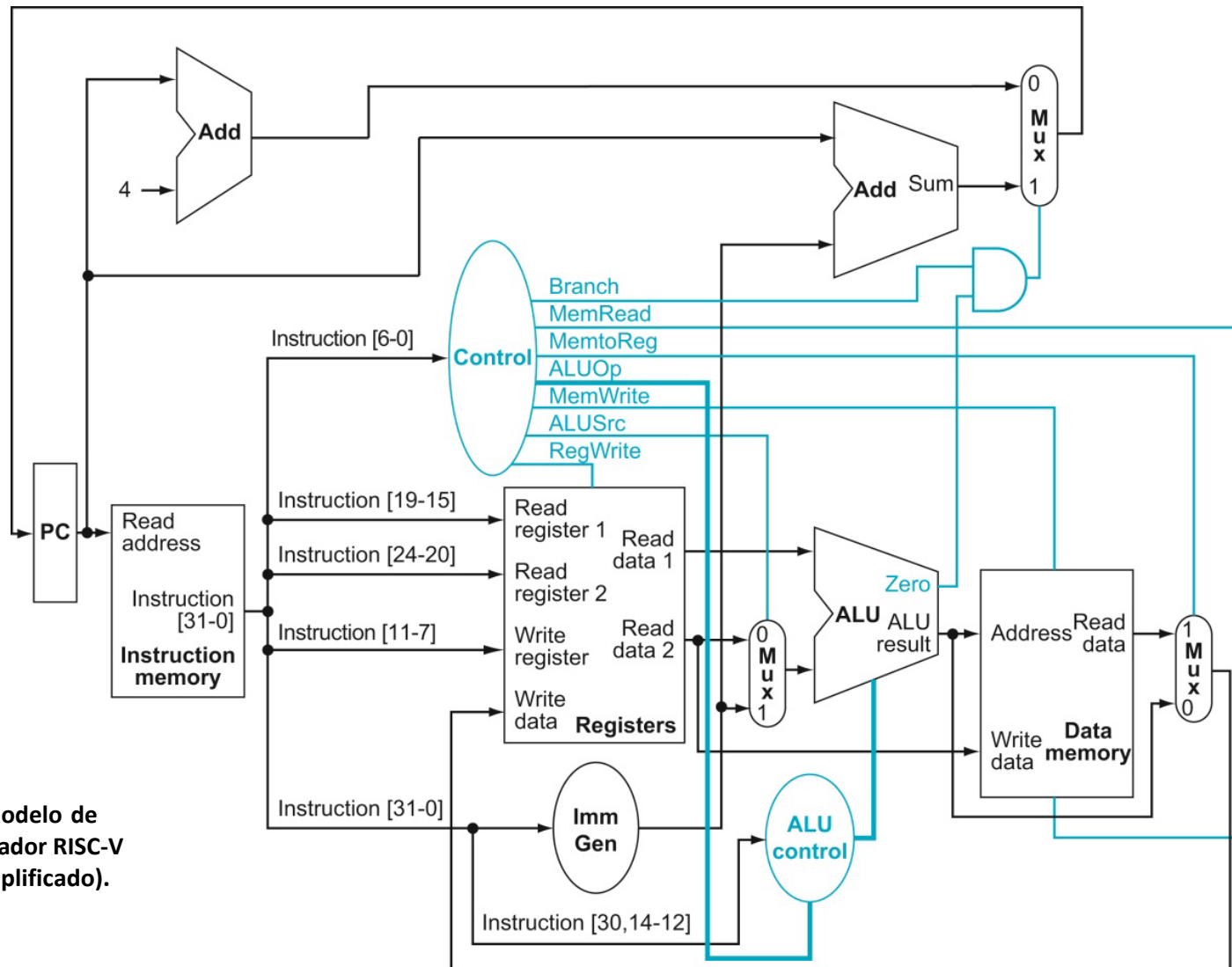


Figura 2a. Modelo de

 microprocesador RISC-V

 unicycle (simplificado).

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

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	000	rd	0110011	

addi rd, rs1, immediate $x[rd] = x[rs1] + \text{sext}(\text{immediate})$

Add Immediate. Tipo I, RV32I y RV64I.

Suma el *immediato* 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

31	20 19	15 14	12 11	7 6	0
immediate[11:0]	rs1	000	rd	0010011	

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

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	111	rd	0110011	

andi rd, rs1, immediate $x[rd] = x[rs1] \& \text{sext}(\text{immediate})$

AND Immediate. Tipo I, RV32I y RV64I.

Calcula el AND a nivel de bits del *immediato* sign-extended y el registro $x[rs1]$ y escribe el resultado en $x[rd]$.

Forma comprimida: **c.andi** rd, imm

31	20 19	15 14	12 11	7 6	0
immediate[11:0]	rs1	111	rd	0010011	

auipc rd, immediate $x[rd] = pc + \text{sext}(\text{immediate}[31:12] \ll 12)$

Add Upper Immediate to PC. Tipo U, RV32I y RV64I.

Suma el *immediato* sign-extended de 20 bits, corrido a la izquierda por 12 bits, al *pc*, y escribe el resultado en $x[rd]$.

31	12 11	7 6	0
immediate[31:12]	rd	0010111	

beq rs1, rs2, offset $\text{if } (rs1 == rs2) \text{ pc} += \text{sext}(\text{offset})$

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

31	25 24	20 19	15 14	12 11	7 6	0
offset[12:10:5]	rs2	rs1	000	offset[4:1:11]	1100011	

bne rs1, rs2, offset $\text{if } (rs1 \neq rs2) \text{ pc} += \text{sext}(\text{offset})$

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

31	25 24	20 19	15 14	12 11	7 6	0
offset[12:10:5]	rs2	rs1	001	offset[4:1:11]	1100011	

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*.

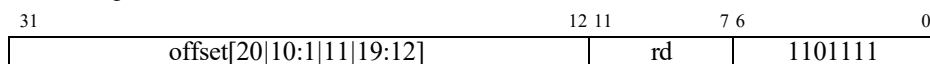
jal rd, offset

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

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*



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**.

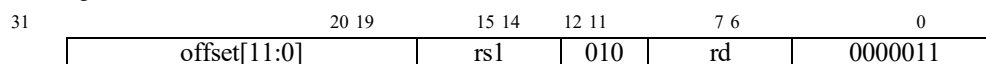
lw rd, offset(rs1)

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

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*)



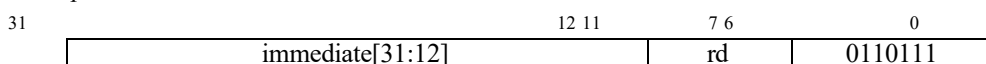
lui rd, immediate

$$x[rd] = sext(immediate[31:12] \ll 12)$$

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*



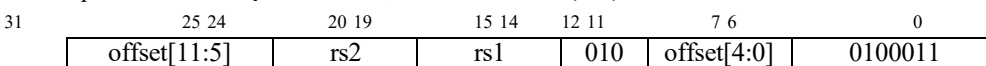
sw rs2, offset(rs1)

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

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*)



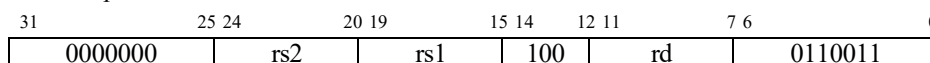
xor rd, rs1, rs2

$$x[rd] = x[rs1] \oplus x[rs2]$$

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 pseudo-instrucción que se traduce en un **addi** *x0*, *x0*, 0

Ejercicio 1 (2 puntos)

Se pide completar el microprocesador RISC V en su versión uniciclo. El diseño entregado como punto de partida, no soporta el set de instrucciones descripto (**add, addi, and, andi, auipc, beq, bne, j, jal, li, lw, lui, sw, xor**).

Ejercicio 2 (1 puntos)

Se pide realizar un programa en ensamblador simple que sea capaz de probar las instrucciones que puedan haber quedado sin probar en el programa provisto como punto de partida. ¿cuales no se prueban en el programa de partida?

Ejercicio 3 (7 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" de la entidad "processor_core".

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

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.

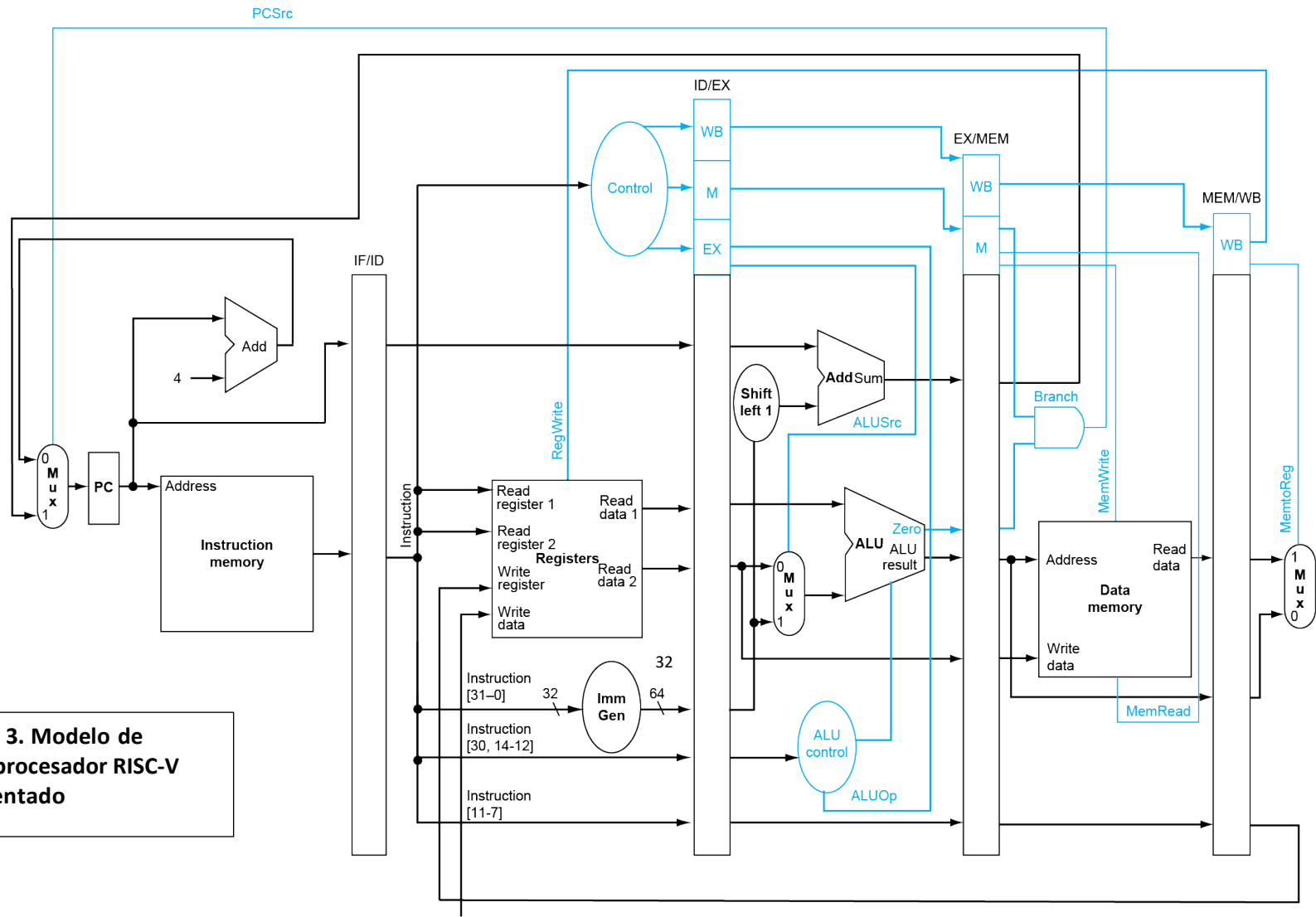


Figura 3. Modelo de microprocesador RISC-V segmentado

MATERIAL A ENTREGAR (**IMPORTANTE RESPETAR EL FORMATO**)

Los ficheros VHDL de los ejercicios 1 y 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 3 carpetas, uno por cada ejercicio.

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

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, en el fichero “procesadorR5_TB.vhd” se puede dar la ruta completa de dichos ficheros cambiando las líneas de código:

```
C_ELF_FILENAME    => "instrucciones",  
...  
C_ELF_FILENAME    => "datos",
```

Por otras donde indique la ruta completa a ambos ficheros:

```
C_ELF_FILENAME    => "X:\nombredirectorio\instrucciones",  
...  
C_ELF_FILENAME    => "X:\nombredirectorio\datos",
```

El compilador y emulador RARS (ejecutable Java) permite ejecutar 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.