Universidad de Buenos Aires
Facultad de Ingeniería

Especialización en Microelectrónica

Arquitectura de Sistemas Digitales

Diseño e Implementación de un Procesaador Simple de Arquitectura RISC

Alumno: Agustín Galdeman
Fecha: 22 de junio de 2025

Buenos Aires, Argentina

1. Introducción

El propósito de este trabajo es diseñar y testear en simulación un microprocesador de arquitectura sencilla, construido íntegramente en *SystemVerilog* y debugueado mediante *EDA Playground*. El proyecto persigue dos objetivos principales:

1. **Comprender el flujo completo de diseño digital**: desde la especificación de la *Instruction Set Architecture* (ISA) hasta la validación funcional mediante un banco de pruebas.
2. **Sentar las bases para extensiones futuras**, tales como introducción de etapas de *pipeline*, manejo de interrupciones o ampliación del set de instrucciones.

La ISA definida para este procesador emplea palabras de 32 bits con campos compactos que permiten, en un único ciclo de clock, ejecutar operaciones aritmético-lógicas, transferencias entre memorias y registros y saltos condicionales. Gracias a esta sencillez, el data-path puede implementarse en un solo nivel de lógica combinacional seguido de registros de salida, eliminando la complejidad de los *hazards* de *pipeline* para la versión inicial.

Para verificar la funcionalidad se ha desarrollado un **programa de prueba corto** que:

- Inicializa un registro con el valor 10,
- Ejecuta un bucle `STORE--LOAD--SUB--JNZ` que decrementa dicho registro mientras almacena los valores en memoria.
- Tras detectar *zero*, sale del bucle para realizar dos escrituras finales antes de entrar en un *idle loop*.

Este flujo pone a prueba los componentes críticos del diseño: decoder, ALU, banco de registros, acceso a memoria de datos, actualización de flags y lógica de salto. La sección siguiente detalla la arquitectura interna y la asignación de campos de la instrucción, seguida de un análisis paso a paso de la ejecución del programa de prueba y del estado final esperado del sistema.

2. Arquitectura del procesador

2.1. Vista general

El procesador se organiza en un único ciclo *fetch–decode–execute–writeback* y está formado por los módulos:

- **RegBank** — Banco de 16 registros de propósito general de 32 bits con dos puertos de lectura (`rsA`, `rsB`).
- **ALU** — Soporta `ADD`, `SUB`, `ABS`, `SHL`, operaciones lógicas (`AND`, `OR`, `XOR`) y negación de `B`. Genera los flags *zero* (`Z`) y *sign* (`S`) y sus versiones negadas.
- **PMEM** — Memoria de programa de $256x32$ bits accedida mediante el program counter (`PC`).
- **DMEM** — Memoria de datos 256 x 32bit orientada a bytes. Las instrucciones `LOAD` y `STORE` acceden a ella.
- **Unidad de control** (dentro de `core`) — Decodifica la instrucción de 32 bits (ver Tabla 1) y genera las señales de control: escritura de registro, escritura de memoria, fuente del operando B, selección de fuente de `rd`, escritura de flags y control de salto.

2.2. Formato de instrucción

La palabra de 32 bits se divide de la siguiente forma (Tab. 1):

bit 31	30	29	28	27	26	25–23	22–20
R	M	J	F	L	RS	cond	alu_op

19–12	11–8	7–4	3–0
$\texttt{Literal}_{[7:0]}$	rsB	rsA	rd

Cuadro 1: Campos de la instrucción.

`R` Habilita escritura en `RegBank`.

`M` Habilita escritura en `DMEM`.

`J` Escribe `PC` si la condición se cumple (`cond_ok`).

`F` Habilita la actualización de flags `Z`/`S`.

`L` Selecciona operando B: literal (`1`) o `rsB` (`0`).

`RS` Selección de fuente para `rd`: `0` ALU / `1` DMEM.

2.3. Circuito de control: máquina de estados implícita

El procesador cada instrucción recorre en un solo pulso:

1. **IF** Lectura de instrucción desde `PMEM[PC]`.
2. **ID** Decodificación y lectura de registros/literal.
3. **EX** Operación ALU o cálculo de dirección de memoria.
4. **MEM** Acceso a `DMEM` (si corresponde).
5. **WB** Escritura en `RegBank` y/o actualización de flags.

El salto condicional se resuelve en la misma etapa **WB** mediante `cond_ok`. No hay *pipeline*; por lo tanto, no aparecen inconvenientes de *hazards* al nivel actual de complejidad.

3. Programa de prueba cargado en `PMEM`

3.1. Listado

```
1  addr  instr                                  comentario
2  0     NOP                                    ; ciclo de relleno
3  1     MOVE  r1, #10                          ; R1      10
4  2     STORE [r1], r1                         ; DMEM[R1]      R1
5  3     LOAD  r2,  #5                          ; R2      DMEM[5]
6  4     SUB   r1, #1          ; F=1            ; R1      R1-1, actualiza Z/S
7  5     JNZ   2                                ; while (R1 != 0) repeat
8  6     STORE [0],  r2                         ; DMEM[0]      R2
9  7     STORE [11], r2                         ; DMEM[11]     R2
10 8     JUMP  8                                ; bucle infinito
```

3.2. Secuencia de ejecución

1. Inicialmente `R1` toma el valor 10.
2. El bucle *2–5* escribe `R1` en su propia dirección, decrementa el registro y vuelve a empezar mientras `Z = 0`. Tras 10 iteraciones:

$$\texttt{R1} = 0, \quad \texttt{DMEM[10:1]} = \{10, 9, \ldots, 1\}$$

3. Al salir del bucle, se copian dos valores de `R2` a memoria: `DMEM[0]` y `DMEM[11]`.
4. El salto incondicional en la dirección 8 mantiene al procesador en un *idle loop*.

3.3. Estado final esperado

`R1`	= 0	`DMEM[0]`	= 5
`R2`	= 5	`DMEM[1]`	= 1
`Z`	= 1	`DMEM[2]`	= 2
`S`	= 0	...	

3.4. Señales Medidas

La Figura 1 muestra la traza obtenida en EPWave para la ejecución del programa de prueba. Para que la correcta secuencia de eventos sea evidente y se pueda justificar cada transición del `PC`, se añadieron las siguientes señales:

1. **Reloj y programa**
   - `clk_i` — señal de clock.
   - `PC[7:0]` — dirección de instrucción.
   - `pmem_dt_i` — palabra de instrucción.
2. **Control principal**
   - `opcode_reg_we` (`R`)
   - `opcode_mem_we` (`M`)
   - `opcode_pc_we` (`J`)
   - `opcode_flg_we` (`F`)
   - `cond_ok` — resultado del comparador de salto.
3. **Banco de registros**
   - `reg_wr_i` — strobe de escritura.
   - `reg_addr_i` — `rd`.
   - `reg_dt_i` — dato escrito.
4. **ALU y flags**
   - `alu_dt` — resultado de la operación.
   - `alu_fZ` — flag `Z` combinacional.
   - `alu_fZ_r` — flag `Z` latcheada.
5. **Memoria de datos**
   - `dmem_wr_o` — strobe de escritura en DMEM.
   - `dmem_addr_o`— dirección de acceso.
   - `dmem_dt_o` — dato escrito.
   - `dmem_dt_i` — dato leído (LOAD).

Estas señales permiten comprobar:

- La transición `PC = 4` $\rightarrow$ `5` con `alu_dt = 0` y `alu_fZ = 1`, seguida de `cond_ok = 0`, lo que provoca que el PC avance a 6.
- Las escrituras consecutivas en `DMEM[10:1]` durante el bucle, y las escrituras finales en `DMEM[0]` y `DMEM[11]`.
- La evolución de `R1` ($10 \rightarrow 0$) y `R2` ($0 \rightarrow 5$).

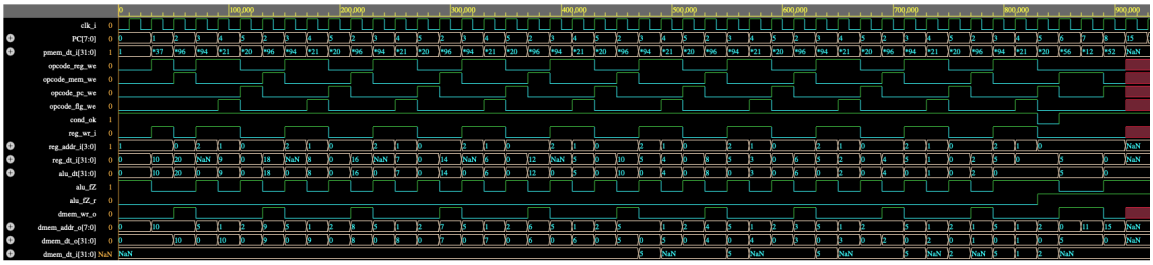

Figura 1: Señales del procesador.

Puede encontrarse la simulación en el siguiente link: Proyecto de EDA Playground