

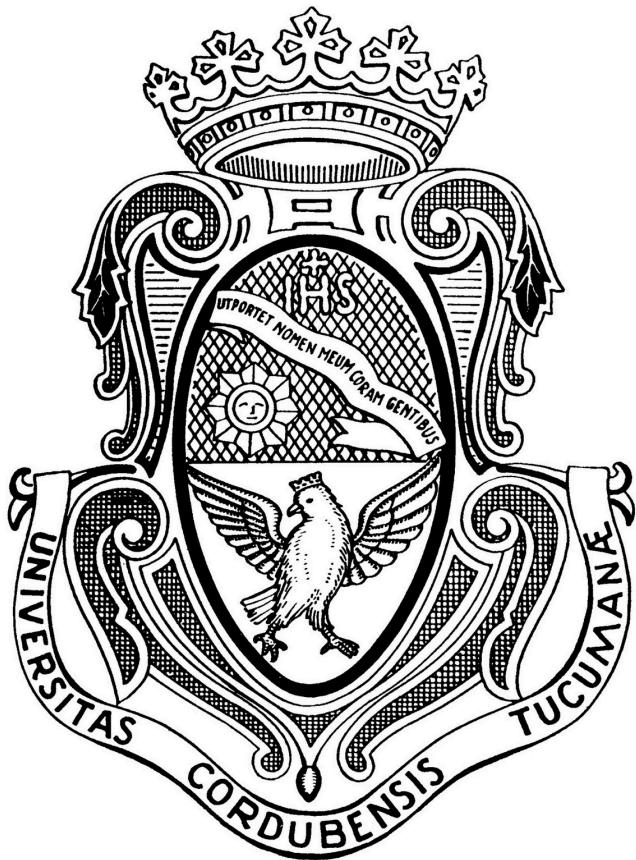
Arquitectura de Computadoras
Trabajo Final

PIPELINE PROCESADOR (RISC-V)

Autores

BORGATELLO, Ignacio
DALLARI LARROSA, Gian Franco

2026



Consigna.....	4
Marco teórico.....	4
Etapas.....	4
Riesgos.....	4
Campos de las instrucciones.....	5
Requerimientos.....	5
Instrucciones.....	5
Implementación.....	6
Introducción.....	6
IF.....	7
Program counter.....	7
Memoria de instrucciones.....	7
Integración.....	8
ID.....	9
Generador de inmediatos.....	9
Registros.....	10
Unidad de control.....	11
Integración.....	12
EX.....	13
ALU Control.....	13
ALU.....	14
Integración.....	14
MEM.....	15
Memoria de datos.....	15
Integración.....	16
WB.....	17
Registros intermedios.....	18
Forwarding unit.....	19
Hazard detection unit.....	21
CPU TOP.....	23
Debug unit.....	27
Interfaz del módulo.....	27
UART RX (entrada desde PC).....	27
UART TX (salida hacia PC).....	27
Señales CPU ↔ DEBUG.....	28
Entradas desde CPU (observabilidad).....	28
Salidas hacia CPU (control de ejecución).....	28
Reprogramación de memoria de instrucciones.....	28
Lectura de estado interno (stream debug).....	28
FSM.....	29
Estados principales.....	29
Política.....	29
Comandos soportados.....	29

Secuencia Run/Stop/Dump.....	30
Secuencia Step/Drain/Dump.....	30
FSM del dump.....	30
Formato del stream.....	31
1) Header (4 bytes).....	31
2) PC (4 bytes, little-endian).....	31
3) Registros (32 × 4 bytes).....	31
4) Memoria de datos (ventana de bytes).....	31
GUI.....	32
Conexión de la placa.....	33
Conexión desde la GUI.....	33
Cargar el programa en la IMEM.....	33
Ejecución completa.....	34
UART.....	34
Simulaciones.....	37
Clock.....	38
Programas de ejemplo.....	40
Programa 1 — ALU + shifts + branches + store/load.....	40
Programa 2 — Memoria completa (SB/SH/SW + LB/LBU/LH/LHU/LW).....	41
Programa 3 — Control de flujo (loop con BNE + BEQ) + JAL link.....	42
Resultados.....	43
Repositorio de GitHub.....	43
Referencias.....	43

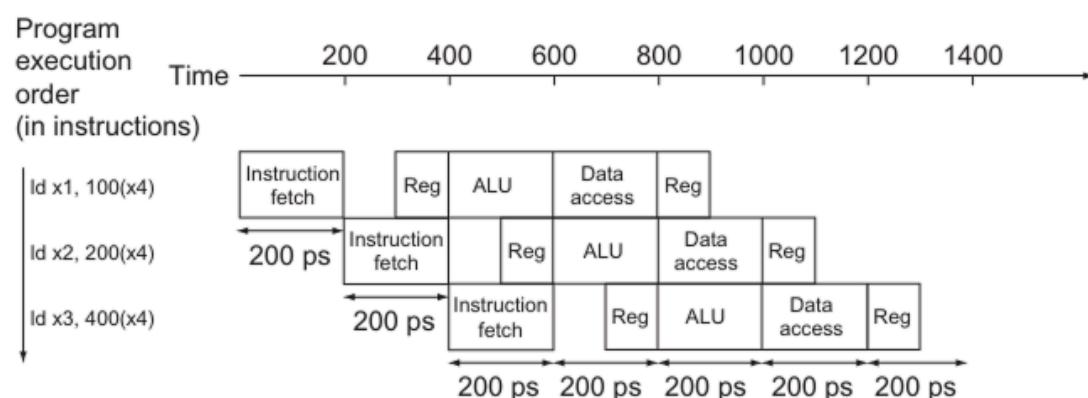
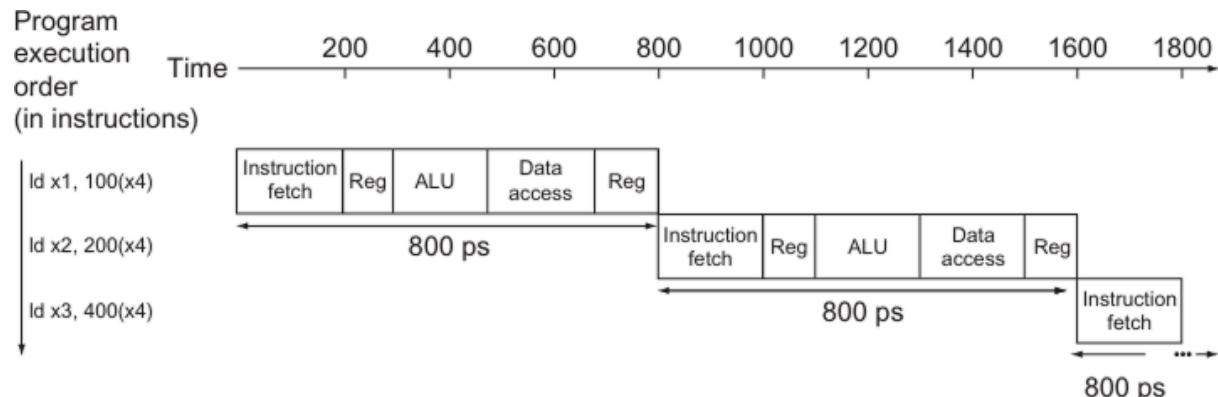
Consigna

1. Implementar el pipeline del procesador RISC-V
2. Implementar una Debug Unit que permite enviar y recibir información al procesador mediante protocolo UART.
3. Una interfaz para visualizar los datos e interactuar con la Debug Unit (CLI, GUI y/o TUI).

Marco teórico

Etapas

- IF (Instruction Fetch): Búsqueda de la instrucción en la memoria de programa.
- ID (Instruction Decode): Decodificación de la instrucción y lectura de registros.
- EX (Execute): Ejecución de la instrucción propiamente dicha.
- MEM (Memory Access): Lectura o escritura desde/hacia la memoria de datos.
- WB (Write back): Escritura de resultados en los registros.



Riesgos

- Estructurales: Se producen cuando dos instrucciones tratan de utilizar el mismo recurso en el mismo ciclo.

- De datos: Se intenta utilizar un dato antes de que esté preparado. Mantenimiento del orden estricto de lecturas y escrituras.
- De control: Intentar tomar una decisión sobre una condición todavía no evaluada.

Campos de las instrucciones

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Here is the meaning of each name of the fields in RISC-V instructions:

- **opcode**: Basic operation of the instruction, and this abbreviation is its traditional name.
- **rd**: The register destination operand. It gets the result of the operation.
- **funct3**: An additional opcode field.
- **rs1**: The first register source operand.
- **rs2**: The second register source operand.
- **funct7**: An additional opcode field.

Requerimientos

- El procesador debe ser capaz de programarse a través de comandos de UART.
- El clock no debe verse intervenido en ninguna parte del proyecto.
- Hay elementos que requieren de su ingenio y toma de decisiones, documenten estas decisiones y su porqué.
- Ser creativos al momento de mostrar los datos e interactuar con ellos (GUI, TUI, CLI).

Instrucciones

Instrucciones a implementar

- R-type (Registro a Registro)
 - add, sub, sll, srl, sra, and, or, xor, slt, sltu
- I-Type (Inmediato/Carga)
 - lb, lh, lw, lbu, lhu, addi, andi, ori, xori, slti, sltiu, slli, srli, srai, jalr
- J-Type (Salto Incondicional)
 - jal
- S-Type (Almacenamiento)
 - sb, sh, sw
- B-Type (Ramificación Condicional)
 - beq, bne
- U-Type (Inmediato Superior)

- lui

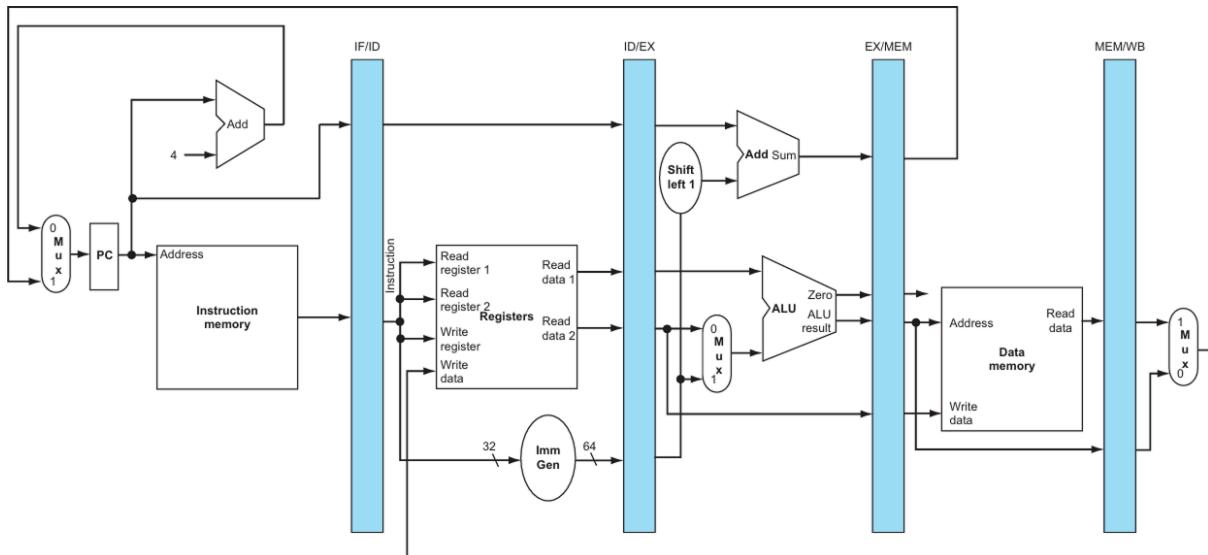
Implementación

Introducción

La arquitectura RISC-V se basa en los procesadores *Reduced Instruction Set Computer* (RISC), cuyo objetivo es simplificar las instrucciones y optimizar su ejecución mediante técnicas de paralelismo. Una de las técnicas fundamentales para mejorar el rendimiento en este tipo de arquitecturas es el pipeline de instrucciones.

El pipeline consiste en dividir la ejecución de una instrucción en varias etapas, de modo que distintas instrucciones puedan encontrarse simultáneamente en procesamiento. De esta forma, mientras una instrucción se está ejecutando, otras pueden estar siendo decodificadas o cargadas desde memoria, aumentando el *throughput* del procesador sin necesidad de incrementar la frecuencia del reloj.

El uso de un pipeline introduce nuevos desafíos, conocidos como riesgos (hazards), que pueden afectar la correcta ejecución del programa. Estos riesgos pueden ser de tipo estructural, de datos o de control, y requieren mecanismos adicionales como *stall*, *flush* o *forwarding* para ser resueltos. A pesar de esta complejidad adicional, el pipeline permite obtener una mejora significativa en el rendimiento general del procesador.



A continuación, se detallan los distintos módulos desarrollados en Verilog, describiendo su función dentro de la arquitectura del procesador y la forma en que interactúan entre sí para implementar el pipeline RISC-V.

IF

Program counter

El módulo `pc_reg` implementa el contador de programa (PC) del procesador y se actualiza de forma síncrona en el flanco positivo del reloj.

Las principales características del módulo son:

- Inicialización por reset.
- Carga forzada desde la Debug Unit, mediante la señal `dbg_load_pc`, permitiendo modificar el flujo de ejecución de manera externa.
- Actualización habilitada, controlada por la señal `en`, que permite implementar `stall` del pipeline sin intervenir el reloj.

```
always @ (posedge clk) begin
    if (reset) begin
        pc <= {XLEN{1'b0}};
    end else if (dbg_load_pc) begin
        pc <= dbg_pc_value;
    end else if (en) begin
        pc <= next_pc;
    end
end
```

Memoria de instrucciones

El módulo `imem_simple` implementa una memoria de instrucciones utilizada durante la etapa *Instruction Fetch* del pipeline.

Sus aspectos principales son:

- Memoria de palabras de 32 bits.
- Inicialización con instrucciones NOP.
- Lectura combinacional para el procesador, permitiendo acceder a la instrucción sin latencia adicional.
- Interfaz de escritura síncrona para la Debug Unit, que posibilita la programación y reprogramación del código en tiempo de ejecución.
- Direcciones en bytes, compatibles con el modelo de direccionamiento de RISC-V.

```

initial begin
    for (i = 0; i < DEPTH; i = i + 1)
        mem[i] = 32'h0000_0013; // NOP

    if (MEM_FILE != "")
        $readmemh(MEM_FILE, mem);
end

wire [AW-1:0] cpu_word = addr[AW+1:2];
wire [AW-1:0] dbg_word = dbg_addr[AW+1:2];

// Lectura combinacional (CPU)
assign instr = mem[cpu_word];

// Escritura síncrona (Debug)
always @(posedge clk) begin
    if (dbg_we) begin
        mem[dbg_word] <= dbg_wdata;
    end
end

```

Dirección (PC)	Instrucción (32 bits)

0x00000000	instr_0 (NOP / ADD / LW / ...)
0x00000004	instr_1
0x00000008	instr_2
0x0000000C	instr_3
...	
0x00000FFC	instr_1023

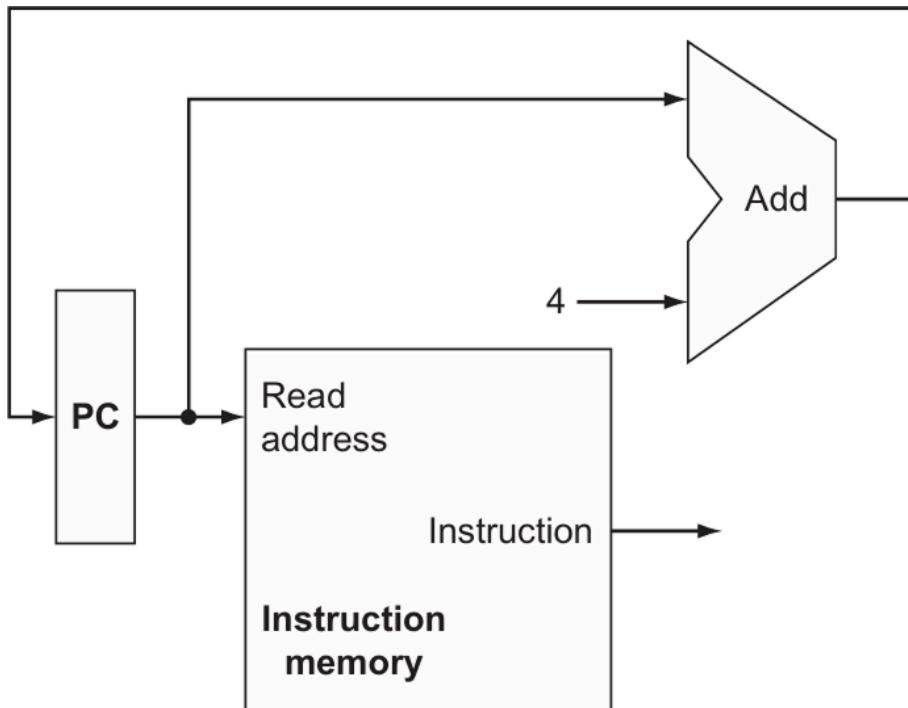
Integración

La etapa IF (Instruction Fetch) se implementó como un módulo que integra el registro de PC y la memoria de instrucciones para entregar, en cada ciclo, la instrucción correspondiente y las direcciones asociadas al avance del programa.

En esta integración:

1. Se calcula PC + 4 como dirección secuencial por defecto.
2. Se selecciona el próximo PC mediante un multiplexor controlado por `pcsrc`, permitiendo elegir entre secuencia normal (PC+4) o salto/branch usando `branch_target`.
3. El avance del PC se habilita con `pc_en`, lo que permite mantener el PC en un stall sin tocar el reloj.

4. Se conecta la memoria de instrucciones para que la salida `instr` refleje la instrucción ubicada en la dirección actual del `pc`.
5. Además, se incorporó soporte de depuración: la Debug Unit puede cargar un PC arbitrario (`dbg_load_pc`) y reprogramar la memoria de instrucciones vía señales de escritura (`imem_dbg_we`, `imem_dbg_addr`, `imem_dbg_wdata`).



ID

Generador de inmediatos

Este módulo se encarga de extraer y construir el inmediato a partir del `instr[31:0]`, siguiendo los formatos de RV32I. En la etapa de *Decode* es clave porque provee el operando inmediato para:

- instrucciones tipo I (OP-IMM, LOAD, JALR) → `instr[31:20]` con sign-extension
- STORE (tipo S) → combinación `instr[31:25] + instr[11:7]`
- BRANCH (tipo B) → reordenamiento de bits + `imm[0]=0` (alineación)
- LUI/AUIPC (tipo U) → `instr[31:12] << 12`
- JAL (tipo J) → reordenamiento + `imm[0]=0`

Registros

El `regfile` implementa el banco de 32 registros de RISC-V, con dos puertos de lectura y uno de escritura, usado en la etapa de *Decode* para obtener `rs1` y `rs2`.

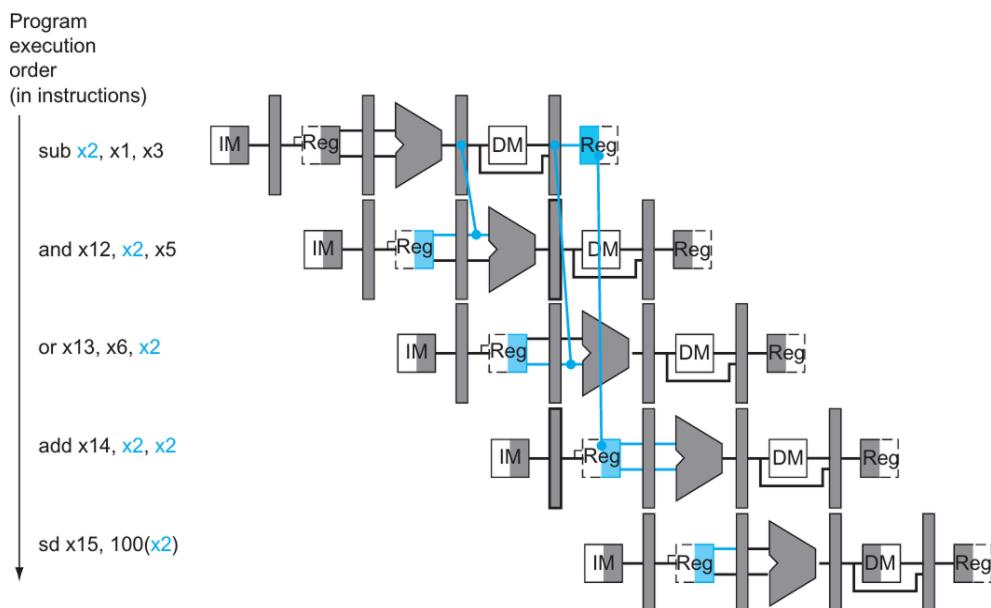
- Escritura síncrona en flanko positivo (`we`, `rd`, `wd`).
- Lecturas combinacionales, para entregar `rd1` y `rd2` sin esperar un ciclo extra.
- Incluye write-first (bypass interno): si en el mismo ciclo se escribe un registro que se está leyendo, se devuelve `wd` (evita un hazard simple dentro del propio regfile).
- Interfaz de debug: permite leer cualquier registro con `dbg_reg_addr` → `dbg_reg_data`, útil para volcado por UART.

```
// Escritura síncrona
always @(posedge clk) begin
    if (reset) begin
        // Para que la simulación sea limpia y repetible
        for (i = 0; i < 32; i = i + 1) begin
            regs[i] <= {XLEN{1'b0}};
        end
    end else begin
        // x0 NO se escribe
        if (we && (rd != 5'd0)) begin
            regs[rd] <= wd;
        end
        // opcional: forzar x0 a 0 por seguridad (no es estrictamente necesario)
        regs[0] <= {XLEN{1'b0}};
    end
end

// Lecturas combinacionales con write-first logic
assign rd1 = (rs1 == 5'd0) ? {XLEN{1'b0}} :
            (we && (rd == rs1) && (rd != 5'd0)) ? wd : regs[rs1];

assign rd2 = (rs2 == 5'd0) ? {XLEN{1'b0}} :
            (we && (rd == rs2) && (rd != 5'd0)) ? wd : regs[rs2];

assign dbg_reg_data = regs[dbg_reg_addr];
```



Unidad de control

Este módulo decodifica el **opcode** y genera las señales de control que gobiernan el datapath/pipeline..

- reg_write: habilita escritura en **rd**
- mem_read / mem_write: control de loads/stores
- mem_to_reg: selección de dato que vuelve al registro (ALU vs memoria)
- alu_src: selecciona si la ALU usa **rs2** o **imm**
- branch / jump / jalr: control de cambios de PC
- wb_sel_pc4: soporta link de **JAL/JALR** (escribir **PC+4**)
- alu_op [1:0]: codificación “macro” para que la ALU/ALU-control determine la operación final (R-type, I-type, branch, suma, etc.)

La lógica se implementa con valores por defecto “seguros” y un **case(opcode)**, lo que facilita depuración y evita señales flotantes.

	reg_wri te	mem_re ad	mem_w rite	mem_to _reg	alu_src	branch	jump	jalr	wb_sel _pc4	alu_op
R-type	1	0	0	0	0	0	0	0	0	10
I-type	1	0	0	0	1	0	0	0	0	11
lw	1	1	0	1	1	0	0	0	0	00
sw	0	0	1	0	1	0	0	0	0	00
beq/bne	0	0	0	0	0	1	0	0	0	01
JAL	1	0	0	0	0	0	1	0	1	00
JALR	1	0	0	0	1	0	0	1	1	00
LUI	1	0	0	0	1	0	0	0	0	00
AUIPC	1	0	0	0	1	0	0	0	0	00

Integración

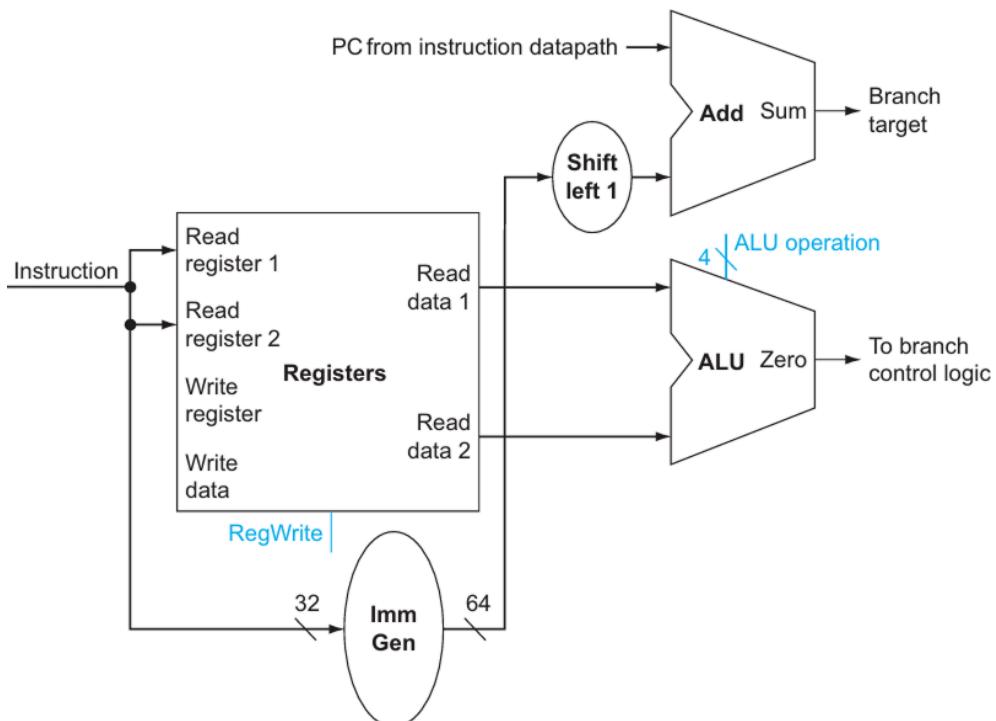
El módulo `id_stage` implementa la etapa de decodificación (Instruction Decode) del pipeline y concentra toda la lógica necesaria para transformar la instrucción proveniente de IF/ID en operandos, inmediato y señales de control para las etapas siguientes.

En términos de integración, esta etapa se arma con tres bloques principales mencionados anteriormente y un extractor de campos.

```
// ----- Field Extractor -----
assign opcode = instr_in[6:0];
assign rd      = instr_in[11:7];
assign funct3 = instr_in[14:12];
assign rs1     = instr_in[19:15];
assign rs2     = instr_in[24:20];
assign funct7 = instr_in[31:25];
```

Como salida adicional, `pc_out` propaga el `pc_in` para mantener disponible el PC asociado a la instrucción decodificada (útil para calcular saltos o para instrucciones tipo `jal/auipc`).

Finalmente, el módulo incluye registros de debug (`dbg_reg_addr / dbg_reg_data`) que permite inspeccionar el contenido del banco de registros desde la unidad sin interferir con la ejecución.



EX

ALU Control

Este módulo traduce las señales de control “macro” (`alu_op`) generadas en ID, junto con los campos `funct3` y `funct7`, a un código concreto `alu_ctrl[3:0]` que selecciona la operación exacta de la ALU.

- `alu_op = 00`: fuerza ADD (típico para load/store, AUIPC, JALR → cálculo de dirección).
- `alu_op = 01`: fuerza SUB (típico para branch, permitiendo comparar vía resta/zero).
- `alu_op = 10`: decodifica R-type usando `funct3` y `funct7[5]` (ej. add/sub, srl/sra).
- `alu_op = 11`: decodifica I-type ALU (addi/andi/ori/xori, shifts, slt/sltu).

```
// Loads / Stores / AUIPC / JALR
2'b00: alu_ctrl = ADD;
// Branches (comparación vía resta)
2'b01: alu_ctrl = SUB;
// R-type
2'b10: begin
    case (funct3)
        3'b000: alu_ctrl = (funct7[5]) ? SUB : ADD; // add /
    sub    3'b111: alu_ctrl = AND;
        3'b110: alu_ctrl = OR;
        3'b100: alu_ctrl = XOR;
        3'b010: alu_ctrl = SLT;
        3'b011: alu_ctrl = SLTU;
        3'b001: alu_ctrl = SLL;
        3'b101: alu_ctrl = (funct7[5]) ? SRA : SRL;
        default: alu_ctrl = ADD;
    endcase
end
// I-type ALU
2'b11: begin
    case (funct3)
        3'b000: alu_ctrl = ADD;    // addi
        3'b111: alu_ctrl = AND;   // andi
        3'b110: alu_ctrl = OR;    // ori
        3'b100: alu_ctrl = XOR;   // xori
        3'b010: alu_ctrl = SLT;   // slti
        3'b011: alu_ctrl = SLTU;  // sltiu
        3'b001: alu_ctrl = SLL;   // slli
        3'b101: alu_ctrl = (funct7[5]) ? SRA : SRL; //
    srl/sra default: alu_ctrl = ADD;
    endcase
end
```

ALU

La `alu` recibe los operandos `a` y `b`, el código `alu_ctrl`, y produce:

- `result`: salida de 32 bits con el resultado de la operación.
- Flags útiles para control:
 - `zero`: se activa si el resultado es cero (muy útil para branches tipo BEQ/BNE según el datapath).
 - `lt`: comparación signed (`$signed(a) < $signed(b)`)
 - `ltu`: comparación unsigned (`a < b`)

Integración

El módulo `ex_stage` implementa la etapa de ejecución del pipeline, donde se resuelven tres cosas principales: operación ALU, cálculo de direcciones/targets, y decisión de branch.

- Selección de operando B (ALUSrc): se incluye un MUX que elige si la ALU opera con `rs2` o con el inmediato `imm`, según `alu_src_in`. El operando A queda fijo en `rs1`.
- Decodificación de operación (ALU control): `alu_control` transforma `alu_op_in + (funct3_in, funct7_in)` en `alu_ctrl`, que indica la operación concreta (add/sub/and/orshifts/slts...).
- Ejecución ALU y flags: la ALU produce `alu_result` y genera banderas (`zero, lt, ltu`) que se exponen para debug y para futuras extensiones de branches.
- Cálculo de destinos de salto/branch:
 - `branch_target_out = pc_in + imm_in` (B-type)
 - `jal_target_ex = pc_in + imm_in`
 - `jalr_target_ex = (rs1 + imm) & ~1` (alineación según jalr)
- Decisión de branch: se compara `rs1` y `rs2` para formar `eq`, se evalúa `cond_true` según `funct3` (000/001), y se genera `branch_taken_out = branch_in & cond_true`.
- Interfaz hacia EX/MEM: se entregan `alu_result_out` (resultado principal) y `rs2_pass_out` (dato a almacenar en caso de `sw`), además del target de branch y la señal de tomado.

MEM

Memoria de datos

El módulo `dmem_rv32` implementa la memoria de datos RV32 del procesador, con soporte para accesos byte/half/word y sus variantes con o sin signo, según el campo `funct3`. Se diseñó con un modelo de direccionamiento por byte.

Organización y direccionamiento

- La memoria se parametriza en tamaño total con `BYTES` (por ejemplo 4096 B = 4 KB).
- Internamente se organiza en palabras de 32 bits (`WORDS = BYTES/4`), y a partir de `addr` se obtiene:
 - `waddr = addr[... :2]` → índice de palabra
 - `boff = addr[1:0]` → offset de byte dentro de la palabra

Implementación por “byte lanes” LittleEndian

Para soportar escrituras parciales (SB/SH) de forma limpia en FPGA, la memoria se implementa como cuatro bancos de 8 bits (`mem0..mem3`), que representan los 4 bytes de una palabra:

- `mem0`: byte menos significativo (LSB)
- `mem3`: byte más significativo (MSB)

Además se sugiere a Vivado implementarla como distributed RAM mediante el atributo `ram_style`.

Lecturas (loads)

- La lectura es combinacional: se arma primero la palabra alineada `word_aligned` y luego se selecciona byte o halfword según el offset.
- Según `funct3`, se entrega:
 - LB/LH con sign-extension
 - LBU/LHU con zero-extension
 - LW palabra completa

Escrituras (stores)

- La escritura es sincrónica en flanco positivo (`clk`), y depende de `funct3`:
 - SB: escribe solo un byte según `boff`
 - SH: escribe 2 bytes (asumiendo alineación a 2 bytes)
 - SW: escribe los 4 bytes

Se agrega una salida `dbg_byte_data` que permite leer un byte arbitrario usando `dbg_byte_addr`, sin interferir con el CPU..

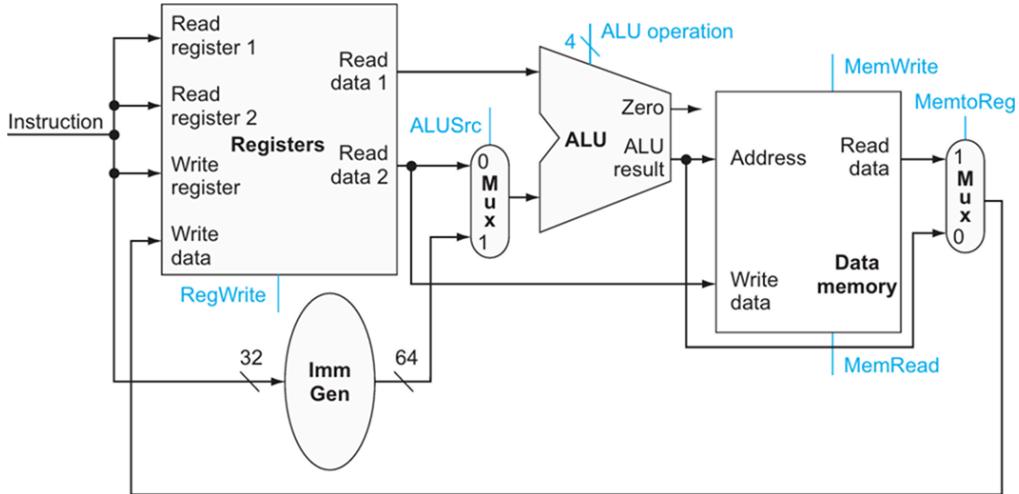
Palabra N			
mem3 ->	Byte 3	MSB	(addr + 3)
mem2 ->	Byte 2		(addr + 2)
mem1 ->	Byte 1		(addr + 1)
mem0 ->	Byte 0	LSB	(addr + 0)

Integración

El módulo `mem_stage` implementa la etapa de acceso a memoria del pipeline y actúa como “wrapper” que conecta el resultado de EX con la memoria de datos RV32.

- Recibe desde EX/MEM el `alu_result_in`, que en este punto se interpreta típicamente como dirección efectiva para loads/stores (o como resultado ALU que debe propagarse).
- Según las señales de control `mem_read` y `mem_write`, habilita la operación correspondiente dentro de `dmem_rv32`.
- El campo `funct3` se utiliza para seleccionar el tipo de acceso: byte/halfword/word y extensión con/sin signo en el caso de loads.
- La salida `mem_read_data` entrega el dato leído ya extendido a 32 bits, listo para la etapa WB.
- En paralelo, `alu_result_out` se mantiene como *passthrough*, para que WB pueda elegir entre “dato de memoria” o “resultado de ALU” según la instrucción.

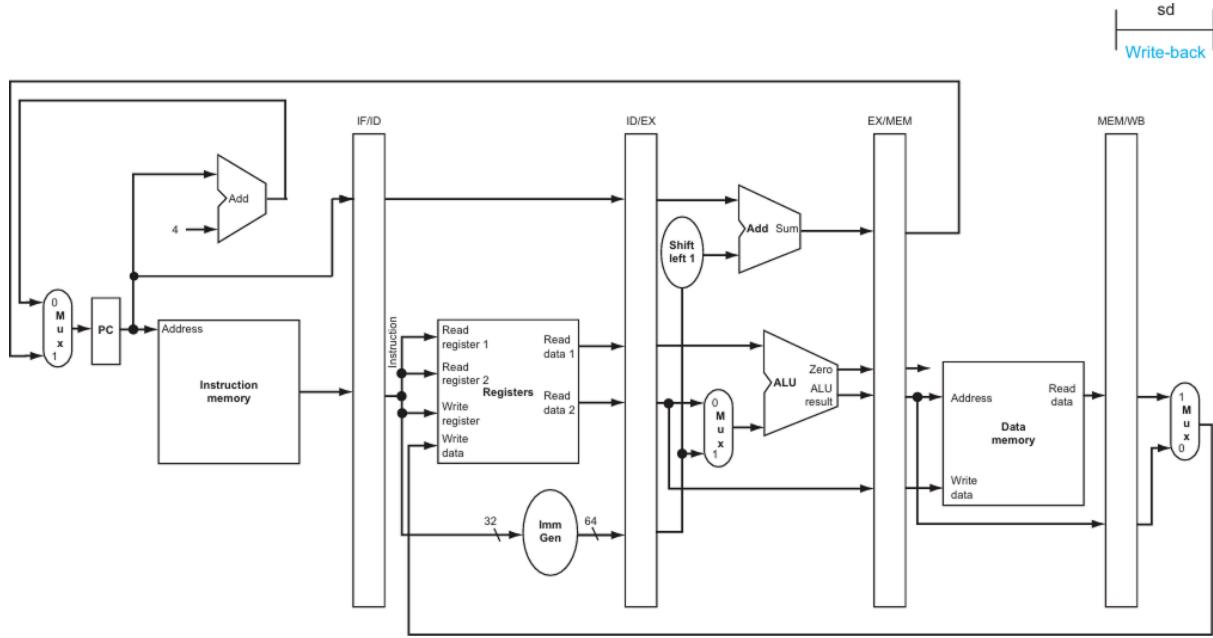
Además se incluye un canal de debug por byte (`dbg_byte_addr` → `dbg_byte_data`) que permite inspeccionar el contenido de memoria sin interferir con el flujo normal del procesador.



WB

El módulo `wb_stage` implementa la etapa final del pipeline, encargada de escribir el resultado de la instrucción en el banco de registros. En esta etapa no se realizan cálculos nuevos, sino que se selecciona y propaga el dato correcto hacia la interfaz de escritura del `regfile`.

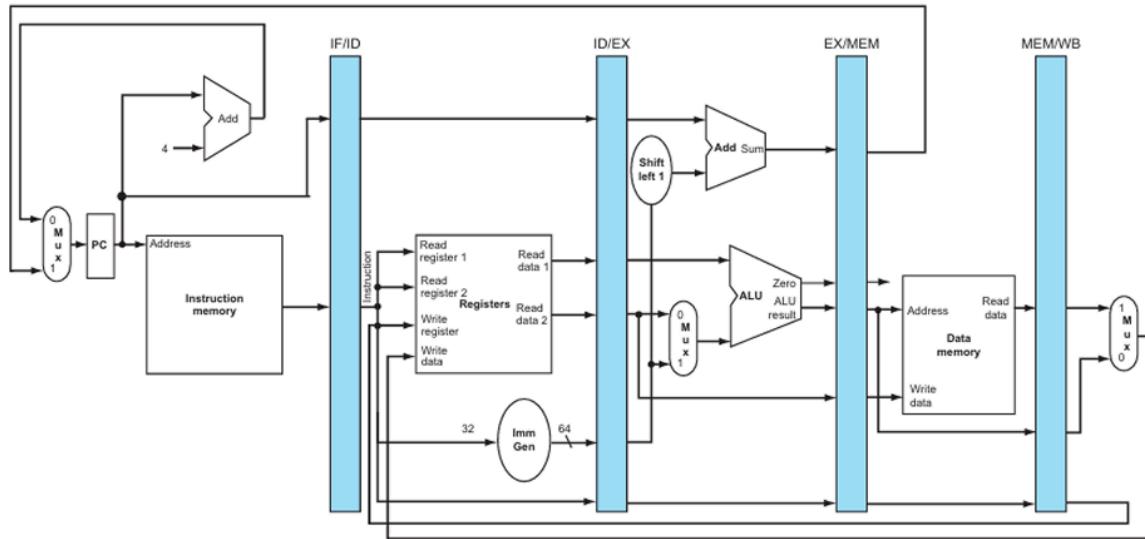
- La señal `wb_wd` se obtiene mediante un multiplexor que selecciona entre:
 - `pc_plus4_mwb`, para instrucciones de salto con link (`jal` / `jalr`);
 - `mem_read_data`, para instrucciones de carga (`load`);
 - `alu_result`, para operaciones aritmético-lógicas y otras instrucciones que escriben el resultado de la ALU.
- La señal `wb_sel_pc4_mwb` tiene prioridad para garantizar que, en instrucciones de salto, el valor escrito en `rd` sea siempre `PC + 4`.
- La señal `mem_to_reg` determina si el dato proviene de memoria o de la ALU cuando no se trata de un salto.
- `reg_write_in` se propaga directamente como `wb_we`, habilitando o no la escritura en el banco de registros.
- El registro destino `rd_in` se envía como `wb_rd` hacia la etapa ID, cerrando el ciclo de ejecución de la instrucción.



Registros intermedios

En un procesador segmentado, cada etapa trabaja en paralelo sobre instrucciones distintas. Para lograrlo, se usan registros de pipeline que almacenan, en cada flanco de reloj, los datos y señales de control producidas por una etapa para que la siguiente las use en el próximo ciclo. En este proyecto, todos los registros comparten una misma filosofía:

- **write_en**: permite stall (si es 0, el registro mantiene su valor).
- **flush**: inserta una burbuja (NOP) o “limpia” la etapa siguiente cuando hay un cambio de flujo (branch/jump) o se desea invalidar una instrucción en vuelo.
- **valid_in/valid_out**: marca si el contenido del registro corresponde a una instrucción válida (útil para debug y para evitar efectos laterales en burbujas).



```

always @(posedge clk) begin
    if (reset) begin
        pc_out      <= 32'b0;
        pc_plus4_out <= 32'b0;
        instr_out    <= NOP;
        valid_out    <= 1'b0;
    end else if (flush) begin
        pc_out      <= 32'b0;
        pc_plus4_out <= 32'b0;
        instr_out    <= NOP;
        valid_out    <= 1'b0;
    end else if (write_en) begin
        pc_out      <= pc_in;
        pc_plus4_out <= pc_plus4_in;
        instr_out    <= instr_in;
        valid_out    <= valid_in;
    end
end

```

Forwarding unit

El módulo **forwarding_unit** implementa la lógica de reenvío de datos (data forwarding) del pipeline, cuyo objetivo es resolver hazards de datos tipo RAW sin introducir stalls innecesarios. Su función es detectar cuándo una instrucción en la etapa EX necesita un

operando que aún no fue escrito en el banco de registros, pero que ya fue calculado por una instrucción previa.

La unidad compara los registros fuente de la instrucción en ID/EX (`idx_rs1`, `idx_rs2`) con los registros destino de las instrucciones que se encuentran en etapas posteriores:

- EX/MEM (resultado más reciente, mayor prioridad)
- MEM/WB (resultado más antiguo)

A partir de estas comparaciones genera dos señales de control:

- `forward_a`: controla el origen del operando A de la ALU
- `forward_b`: controla el origen del operando B de la ALU

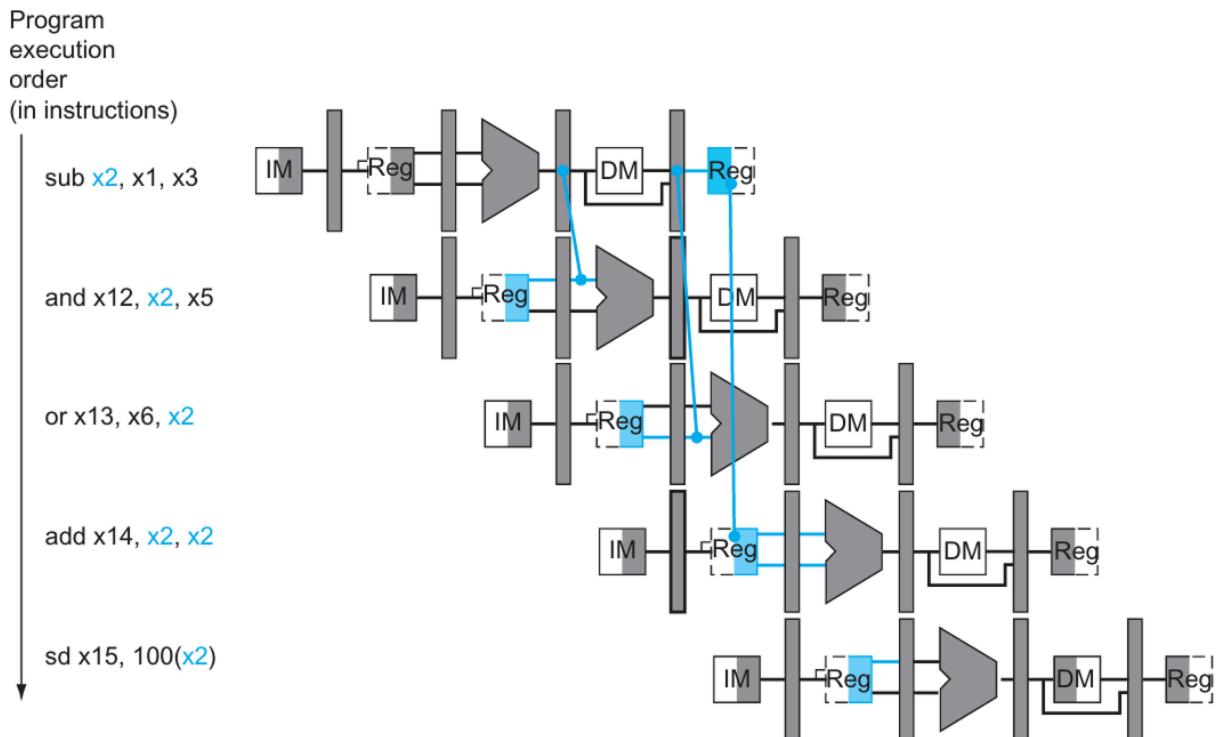
Criterios de decisión

- Si `exmem_reg_write` está activo y `exmem_rd` coincide con `idx_rsX` (y no es `x0`), se selecciona forward desde EX/MEM (`2'b10`).
- En caso contrario, si `memwb_reg_write` está activo y `memwb_rd` coincide con `idx_rsX`, se selecciona forward desde MEM/WB (`2'b01`).
- Si no hay coincidencias, se utiliza el valor leído originalmente del banco de registros (`2'b00`).

La prioridad otorgada a EX/MEM sobre MEM/WB asegura que siempre se use el dato más reciente disponible.

```
// ----- Forward A (rs1) -----
if (exmem_reg_write && (exmem_rd != 5'd0) && (exmem_rd == idx_rs1)) begin
    forward_a = 2'b10; // desde EX/MEM
end else if (memwb_reg_write && (memwb_rd != 5'd0) && (memwb_rd == idx_rs1)) begin
    forward_a = 2'b01; // desde MEM/WB
end

// ----- Forward B (rs2) -----
if (exmem_reg_write && (exmem_rd != 5'd0) && (exmem_rd == idx_rs2)) begin
    forward_b = 2'b10; // desde EX/MEM
end else if (memwb_reg_write && (memwb_rd != 5'd0) && (memwb_rd == idx_rs2)) begin
    forward_b = 2'b01; // desde MEM/WB
end
```



Hazard detection unit

El módulo `hazard_detection_unit` implementa la detección de hazards de datos tipo load-use, que no pueden resolverse únicamente mediante forwarding. Este caso ocurre cuando una instrucción en la etapa EX está realizando una carga desde memoria (load) y la instrucción siguiente necesita inmediatamente el valor cargado.

La unidad analiza las siguientes señales:

- `index_mem_read`: indica que la instrucción en ID/EX es un load.
- `index_rd`: registro destino del load.
- `ifid_rs1, ifid_rs2`: registros fuente de la instrucción siguiente (en IF/ID).

Condición de *stall*

Se genera un *stall* cuando:

- la instrucción en EX realiza una lectura de memoria,
- y dicho registro coincide con alguno de los registros fuente de la instrucción siguiente.

Acciones ante un *stall*

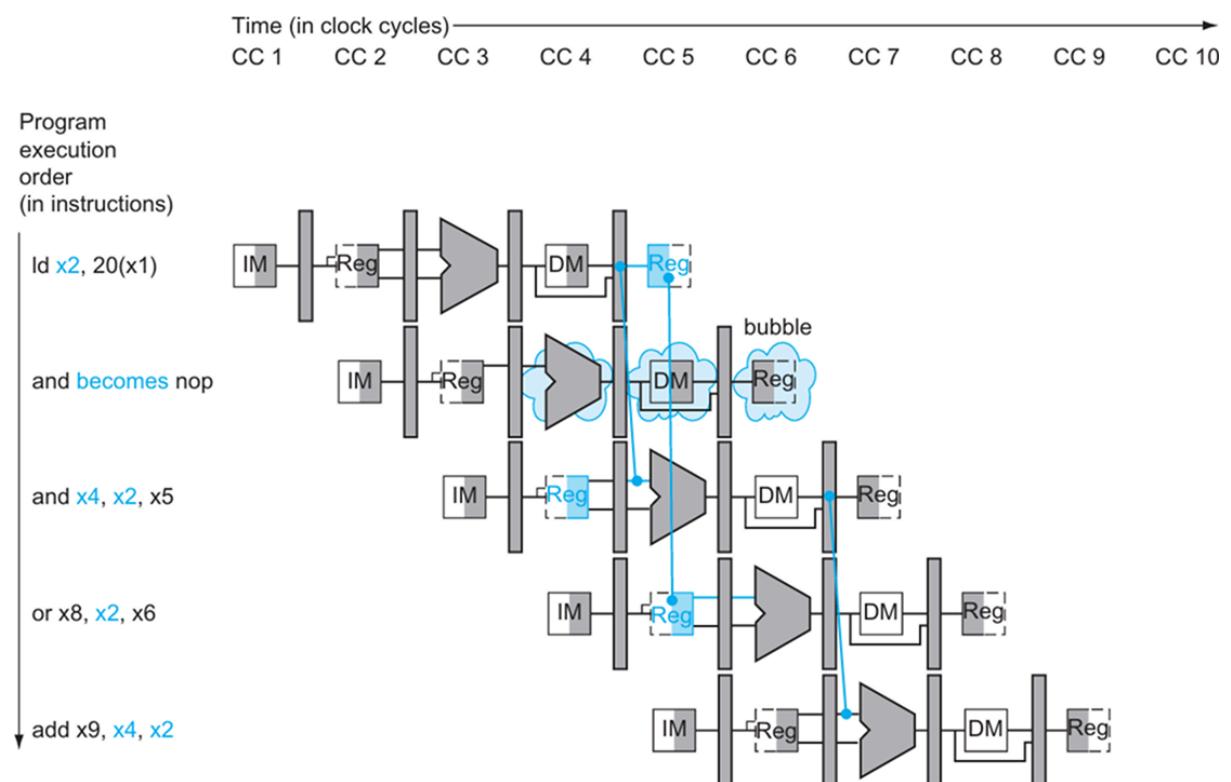
Cuando se detecta esta condición, la unidad genera automáticamente las señales necesarias para mantener la coherencia del pipeline:

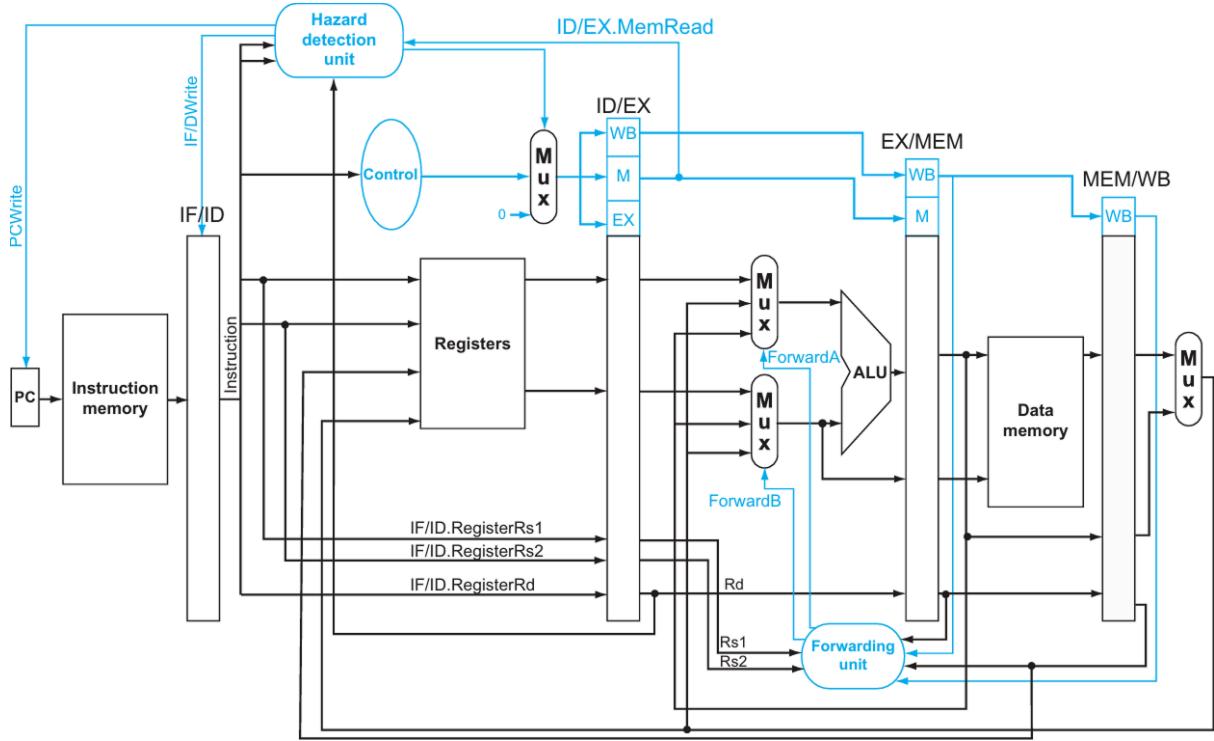
- `pc_en = 0`: se congela el PC, evitando avanzar a la siguiente instrucción.
- `ifid_write_en = 0`: se congela el registro IF/ID, manteniendo la instrucción actual.
- `idx_flush = 1`: se inserta una burbuja en ID/EX, anulando las señales de control de la instrucción dependiente.

De esta manera, se introduce un stall de un ciclo, suficiente para que el dato cargado desde memoria esté disponible en la etapa siguiente y pueda ser utilizado correctamente.

```
assign stall =
    idx_mem_read &&
    (idx_rd != 5'd0) &&
    ((idx_rd == ifid_rs1) || (idx_rd == ifid_rs2));

// Si stall=1: congelar PC e IF/ID, y flushear ID/EX
assign pc_en          = ~stall;
assign ifid_write_en = ~stall;
assign idx_flush     = stall;
```





CPU TOP

El módulo `cpu_top` integra todas las etapas del pipeline (IF, ID, EX, MEM, WB), sus registros inter-etapa y los mecanismos de forwarding, detección de hazards load-use, flush/stall, y un conjunto de señales de debug que permiten ejecutar en modo continuo, paso a paso o drenado, sin intervenir el reloj.

Control global: run / step / drain / freeze

El avance del procesador se controla mediante un habilitador lógico global `cpu_ce`, construido a partir de señales externas de depuración. Para habilitar el modo `step`, se detecta el flanco ascendente de `dbg_step` y se genera un pulso `step_pulse`.

```

always @(posedge clk) begin
    if (reset) dbg_step_q <= 1'b0;
    else        dbg_step_q <= dbg_step;
end

assign step_pulse = dbg_step & ~dbg_step_q;
assign step_fire  = step_pulse & ~dbg_run;

assign cpu_ce = (dbg_run | step_fire | dbg_drain) & ~dbg_freeze;

```

Con este esquema:

- `dbg_run` habilita ejecución continua.
- `dbg_step` avanza un ciclo aislado cuando no se está en run.
- `dbg_drain` permite vaciar el pipeline.
- `dbg_freeze` anula cualquier avance.

Este mecanismo cumple el requisito de “no intervenir el clock”: el reloj es el mismo, y lo que se controla son los enables de escritura y el avance del PC.

Señales de estado: `halt_seen` y `pipe_empty`

Para depuración se incluye detección de `EBREAK` y un flag latcheado `dbg_halt_seen`. La detección se realiza sobre la instrucción en IF/ID cuando es válida:

```
localparam [31:0] INSTR_EBREAK = 32'h0010_0073;
wire halt_id = valid_ifid && (instr_ifid == INSTR_EBREAK);

reg halt_seen_r;
always @(posedge clk) begin
    if (reset) begin
        halt_seen_r <= 1'b0;
    end else if (dbg_flush_pipe || dbg_load_pc) begin
        halt_seen_r <= 1'b0;
    end else if (halt_id) begin
        halt_seen_r <= 1'b1;
    end
end
assign dbg_halt_seen = halt_seen_r;
```

Además, el estado “pipeline vacío” se determina con los flags `valid` de cada registro inter-etapa:

```
assign dbg_pipe_empty = ~(valid_ifid | valid_idex | valid_exmem |
    valid_memwb);
```

Implementación de las etapas

El PC no avanza si hay stall por hazard, si el debug congela, o si se está drenando. El enable efectivo del PC queda:

```
assign pc_en = hdu_pc_en & cpu_ce & ~dbg_drain;
```

Esto implica que durante `dbg_drain`:

- el pipeline puede seguir avanzando (si `cpu_ce=1`)
- pero el PC no incorpora instrucciones nuevas, permitiendo “vaciar” lo que ya estaba en vuelo.

La etapa IF se instancia con soporte de:

- selección de PC (pcsrc/branch_target),
- reprogramación de IMEM,
- carga forzada de PC desde debug.

El registro IF/ID se puede:

- congelar (stall) con `write_en`,
- invalidar (flush) insertando un NOP y `valid=0`.

La habilitación de escritura en IF/ID combina hazard y debug:

```
assign write_ifid = hdu_ifid_we & cpu_ce;
```

El `flush_ifid` se activa ante cambio de flujo (branch/jump), flush externo o drain:

```
assign flush_ifid = ((pcsrc_ex & cpu_ce) | dbg_flush_pipe) | dbg_drain;
```

La validez de la instrucción que entra al registro se liga al avance real del PC:

```
.valid_in(pc_en & ~dbg_drain),
```

La unidad de hazards compara el `rd` de un load en ID/EX contra `rs1/rs2` de la instrucción en IF/ID, y genera un stall clásico:

- se congela PC e IF/ID,
- se flushea ID/EX para insertar una burbuja.

La etapa ID recibe la interfaz de WB, pero la escritura al regfile se habilita solo si el CPU está “avanzando” en ese ciclo:

```
.wb_reg_write(wb_we & cpu_ce),
```

Además, se expone una lectura de registros para debug:

```
.dbg_reg_addr(rf_dbg_addr),  
.dbg_reg_data(rf_dbg_data)
```

El registro ID/EX se flushea cuando:

- hubo cambio de flujo resuelto en EX (`pcsrc_ex`),
- la HDU solicita burbuja (`hdu_idex_flush`),
- debug flush,
- drain.

```
assign flush_idex = (((pcsrc_ex | hdu_idex_flush) & cpu_ce) | dbg_flush_pipe)  
| dbg_drain;  
assign write_idex = cpu_ce;
```

Esto asegura que, ante branch/jump, las instrucciones “mal fetcheadas” no continúen avanzando.

La unidad de forwarding decide de dónde alimentar los operandos A/B de EX.

Y el multiplexado se realiza así:

```
case (forward_a)  
  2'b01: rs1_fwd = wb_wd;  
  2'b10: rs1_fwd = alu_result_exmem;  
  default: rs1_fwd = rs1_data_idex;  
endcase  
  
case (forward_b)  
  2'b01: rs2_fwd = wb_wd;  
  2'b10: rs2_fwd = alu_result_exmem;  
  default: rs2_fwd = rs2_data_idex;  
endcase
```

Con prioridad EX/MEM sobre MEM/WB, se utiliza siempre el valor más reciente.

La etapa EX produce `branch_taken_ex` y targets de salto. En `cpu_top` se unifica el control de cambio de PC:

```
assign pc_branch_ex =  
  (jalr_idex) ? jalr_target_ex :  
  (jump_idex) ? jal_target_ex :  
  branch_target_ex;
```

Este bloque define la política de prioridad y concentra la decisión final de control de flujo.

El acceso a memoria se enmascara con `cpu_ce` para evitar efectos laterales en freeze/pausa:

```
.mem_read (mem_read_exmem & cpu_ce),  
.mem_write(mem_write_exmem & cpu_ce),
```

La etapa WB alimenta el regfile mediante `wb_we/wb_wd/wb_rd`, y la escritura efectiva ya quedó controlada en ID con `wb_we & cpu_ce`.

La MEM expone una interfaz independiente de lectura por byte para inspección:

```
.dbg_byte_addr(dmem_dbg_addr),  
.dbg_byte_data(dmem_dbg_data)
```

Esto permite monitorear la memoria desde la PC/UART sin interferir con el datapath principal.

Debug unit

La unidad `debug_unit_uart` implementa una interfaz de depuración basada en UART que permite controlar la ejecución del procesador, reprogramar la memoria de instrucciones y extraer el estado interno (PC, registros y una ventana de memoria de datos) en forma de un stream de bytes. El módulo opera completamente en el mismo dominio de reloj que el CPU (`clk`), por lo que las señales de control se generan de forma síncrona y determinista.

El diseño se estructura en dos partes principales:

1. FSM de comandos (RX → control del CPU / programación / solicitudes de dump)
2. FSM de transmisión del dump (TX → stream secuencial con índice `dump_idx`)

Interfaz del módulo

UART RX (entrada desde PC)

- `rx_done_tick`: pulso de 1 ciclo que indica la llegada de un byte.
- `rx_dout[7:0]`: byte recibido.

Estos bytes representan comandos ASCII y/o payload binario en el caso de programación de IMEM.

UART TX (salida hacia PC)

- `tx_start`: pulso de 1 ciclo para iniciar el envío de un byte.
- `tx_din[7:0]`: byte a transmitir.
- `tx_done_tick`: pulso de fin de transmisión de byte (handshake).

La lógica evita iniciar un nuevo byte mientras haya uno “en vuelo”, mediante `tx_inflight`.

Señales CPU ↔ DEBUG

Entradas desde CPU (observabilidad)

- `dbg_pc`: valor del PC actual (exportado por `cpu_top`).
- `dbg_pipe_empty`: indica que no hay instrucciones válidas en pipeline (para finalizar drain).
- `dbg_halt_seen`: flag latcheado por EBREAK, usado para detener RUN.

Salidas hacia CPU (control de ejecución)

- `dbg_freeze`: congela el avance del CPU (modo pausa).
- `dbg_run`: ejecución continua.
- `dbg_step`: pulso de 1 ciclo para “avanzar” (step por ciclo).
- `dbg_drain`: permite vaciar el pipeline sin incorporar nuevas instrucciones (según lógica del `cpu_top`).

Además se incluyen señales de eventos:

- `dbg_flush_pipe`: pulso para invalidar el pipeline (insertar burbujas).
- `dbg_load_pc + dbg_pc_value`: pulso y dato para forzar el PC (soft reset / salto manual).

Reprogramación de memoria de instrucciones

La unidad permite reprogramar IMEM mediante:

- `imem_dbg_we`: pulso de escritura
- `imem_dbg_addr`: dirección (byte-addressed, tipo PC)
- `imem_dbg_wdata`: palabra de 32 bits (una instrucción)

Este mecanismo se activa con el comando “P” y luego recibe 8 bytes: 4 de dirección + 4 de dato.

Lectura de estado interno (stream debug)

Para extraer el estado del CPU sin “copiar” memorias completas por hardware, la debug unit usa dos puertos de lectura:

- Regfile: `rf_dbg_addr[4:0]` selecciona registro, `rf_dbg_data[31:0]` devuelve su contenido.
- Data memory: `dmem_dbg_addr[11:0]` selecciona byte, `dmem_dbg_data[7:0]` devuelve el byte.

Estos puertos son usados por la FSM de dump para formar un stream secuencial.

FSM

La FSM principal recibe comandos por RX y activa modos o eventos sobre el CPU.

Estados principales

```
ST_IDLE: espera comando, CPU congelado por defecto.  
ST_P_ADDR: recibe 4 bytes de dirección para programación.  
ST_P_DATA: recibe 4 bytes de data y ejecuta imem_dbg_we.  
ST_RUN: habilita ejecución continua hasta detectar dbg_halt_seen.  
ST_STEP: emite dbg_step por 1 ciclo.  
ST_STEP_WAIT: ciclo intermedio para estabilizar el avance.  
ST_DRAIN: vacía pipeline hasta dbg_pipe_empty.  
ST_DUMP: congela CPU y delega el stream a la FSM de TX
```

Política

Dentro del bloque secuencial, se implementa una política de reset por defecto para señales evento:

```
dbg_step      <= 1'b0;  
dbg_flush_pipe <= 1'b0;  
dbg_load_pc    <= 1'b0;  
imem_dbg_we    <= 1'b0;
```

Comandos soportados

En `ST_IDLE`, al recibir `rx_dout` se selecciona acción:

- "P": programación de IMEM
Pasa a `ST_P_ADDR` y acumula 4 bytes de dirección + 4 bytes de dato.
- "R": reset de ejecución (PC=0 + flush)
Genera simultáneamente:
 - `dbg_pc_value = 0`

- `dbg_flush_pipe = 1`
 - `dbg_load_pc = 1`
- "T": freeze inmediato
Fuerza `dbg_freeze=1`.
- "D": dump manual
Selecciona `dump_type=MANUAL` y pasa a `ST_DUMP`.
- "S": step + dump
Selecciona `dump_type=STEP`, ejecuta `ST_STEP → DRAIN → DUMP`.
- "G": run continuo
Pasa a `ST_RUN`.

Secuencia Run/Stop/Dump

En `ST_RUN`, el CPU queda en ejecución continua:

```
dbg_freeze <= 1'b0;
dbg_run     <= 1'b1;
```

Cuando el CPU detecta `EBREAK` y levanta `dbg_halt_seen`, la debug unit:

- detiene `dbg_run`,
- activa `dbg_drain`,
- y finalmente realiza un `dump_type=RUN_END`.

Secuencia Step/Drain/Dump

La secuencia para step es:

1. `ST_STEP`: `dbg_step = 1` por un ciclo (avance controlado).
2. `ST_STEP_WAIT`: baja el pulso y entra en modo drain.
3. `ST_DRAIN`: espera `dbg_pipe_empty` para garantizar estado estable.
4. `ST_DUMP`: congela y transmite el estado.

Esta decisión asegura que el dump se capture con el pipeline en un estado consistente.

FSM del dump

La transmisión se implementa con un índice `dump_idx` que recorre el total del stream:

- Header: 4 bytes
- PC: 4 bytes
- Regs: 32×4 bytes
- Mem: `DM_DUMP_BYTES` bytes

```
localparam integer DUMP_TOTAL =
  4 + 4 + (32*4) + DM_DUMP_BYTES;
```

La FSM usa `tx_inflight` para garantizar que sólo se emita un byte nuevo cuando el anterior finalizó (`tx_done_tick`).

Formato del stream

El dump transmitido por UART sigue este layout:

1) Header (4 bytes)

Byte	Contenido
0	<code>0xD0</code> (magic)
1	<code>dump_type</code> (1=STEP, 2=RUN_END, 3=MANUAL)
2	flags <code>{pipe_empty, halt_seen}</code>
3	padding <code>0x00</code>

2) PC (4 bytes, little-endian)

```
dbg_pc[7:0], dbg_pc[15:8], dbg_pc[23:16], dbg_pc[31:24]
```

3) Registros (32 × 4 bytes)

Se recorren `x0..x31`, enviando cada registro en little-endian.

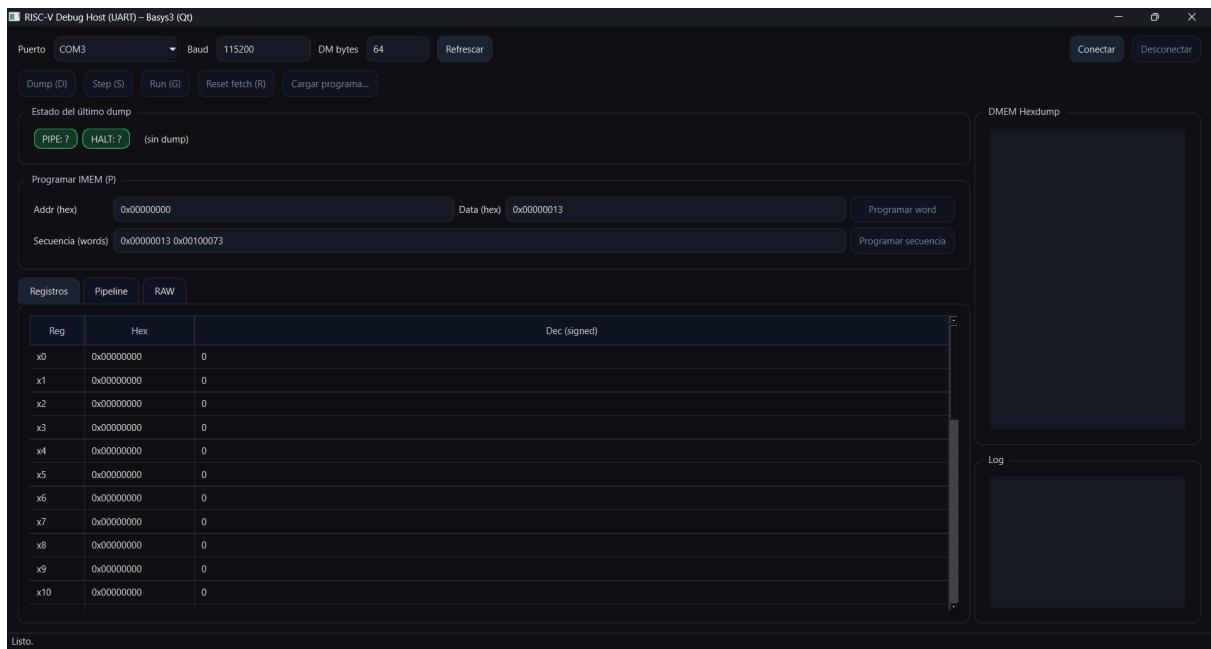
La dirección se genera con:

```
reg_idx = (dump_idx - 8) >> 2;
reg_byte = (dump_idx - 8) & 3;
rf_dbg_addr <= reg_idx;
```

4) Memoria de datos (ventana de bytes)

Se transmiten `DM_DUMP_BYTES` bytes desde `dmem_dbg_addr=0..DM_DUMP_BYTES-1`.

GUI



Este instructivo describe cómo cargar, ejecutar y verificar un programa simple en el procesador RISC-V implementado en la FPGA, utilizando la unidad de debug por UART.

El programa sirve para validar:

- escritura de registros (`addi`)
- ejecución secuencial
- detección de `EBREAK`
- funcionamiento correcto del pipeline y del debug unit

```
addi x1, x0, 7
addi x2, x0, 3
add x3, x1, x2
ebreak
```

Instrucción	Hex
addi x1,x0,7	0x00700093
addi x2,x0,3	0x00300113
add x3,x1,x2	0x002081B3
ebreak	0x00100073

Creá un archivo de texto, por ejemplo:

```
prog.mem
```

con el siguiente contenido (una instrucción por línea):

```
00700093  
00300113  
002081B3  
00100073
```

Este archivo representa la IMEM comenzando desde la dirección `0x00000000`.

Conexión de la placa

1. Programa la Basys 3 con el bitstream del proyecto.
2. Conectá el cable USB.
3. Identifica el puerto serie (COMx en Windows / `/dev/ttyUSBx` o `/dev/ttyACMx` en Linux).
4. Abrí la GUI de debug en Python.

Conexión desde la GUI

En la GUI:

1. Seleccioná el puerto serie
2. Baudrate: 115200
3. DM bytes: 64
4. Presioná Conectar

En el log deberías ver algo como:

```
[INFO] Conectado a COM5 @ 115200
```

Cargar el programa en la IMEM

1. Presioná “Cargar programa...”
2. Seleccioná el archivo `prog.mem`
3. Esperá a que finalice la carga

En el log deberías ver:

```
[INFO] Words a programar: 4  
[OK] Programa cargado. Rango: 0x00000000 .. 0x0000000c
```

Antes de ejecutar, es importante forzar el PC a 0.

1. Presioná “Reset fetch (R)”

Esto hace:

- $PC \leftarrow 0$
- flush del pipeline
- limpia estados internos del debug

Ejecución completa

1. Presioná “Run (G)”

El procesador:

- ejecuta instrucciones
- detecta EBREAK
- drena el pipeline
- genera automáticamente un dump RUN_END

En la tabla de registros deberías observar:

Registro	Valor esperado
x0	0x00000000
x1	0x00000007
x2	0x00000003
x3	0x0000000A
x4..x31	0x00000000

UART

Para el envío de datos a través del UART, fue desarrollado el generador de Baud Rate, esencial para sincronizar la comunicación. Se implementó un divisor de frecuencia basado en un contador para ajustar la tasa de transmisión a un valor definido, por ejemplo, 19200 baudios.

```

`timescale 1ns / 1ps

module mod_m_counter
#(
    parameter N = 4,
    parameter M = 10
)
(
    input wire clk,
    input wire reset,
    output wire max_tick,
    output wire [N-1:0] q
);

reg [N-1:0] r_reg;
wire [N-1:0] r_next;

always @(posedge clk)
    if (reset)
        r_reg <= 0;
    else
        r_reg <= r_next;

assign r_next = (r_reg == (M-1)) ? 0 : r_reg + 1;
assign max_tick = (r_reg == (M-1)) ? 1'b1 : 1'b0;
assign q = r_reg;
endmodule

```

Teniendo en cuenta que $\frac{Clock}{BaudRate * 16} = Ticks$

Al trabajar con la frecuencia de reloj predeterminada de 100 MHz, especificada en la restricción (constraint) de la placa, entonces, la cantidad de ticks que debemos definir en el mod_m_counter es la siguiente: $\frac{100\text{ MHz}}{19200 * 16} = 325$

La comunicación UART se implementó mediante dos módulos complementarios: el receptor, que detecta el bit de inicio, realiza el muestreo de los bits de datos y comprueba la validez de la información a través del bit de paridad, y el transmisor, encargado de enviar los datos de manera secuencial respetando el protocolo UART; en ambos casos se emplearon máquinas de estados finitos (FSM) para coordinar las distintas etapas del proceso, garantizando un funcionamiento correcto y una temporización adecuada tanto en la recepción como en la transmisión de los datos.

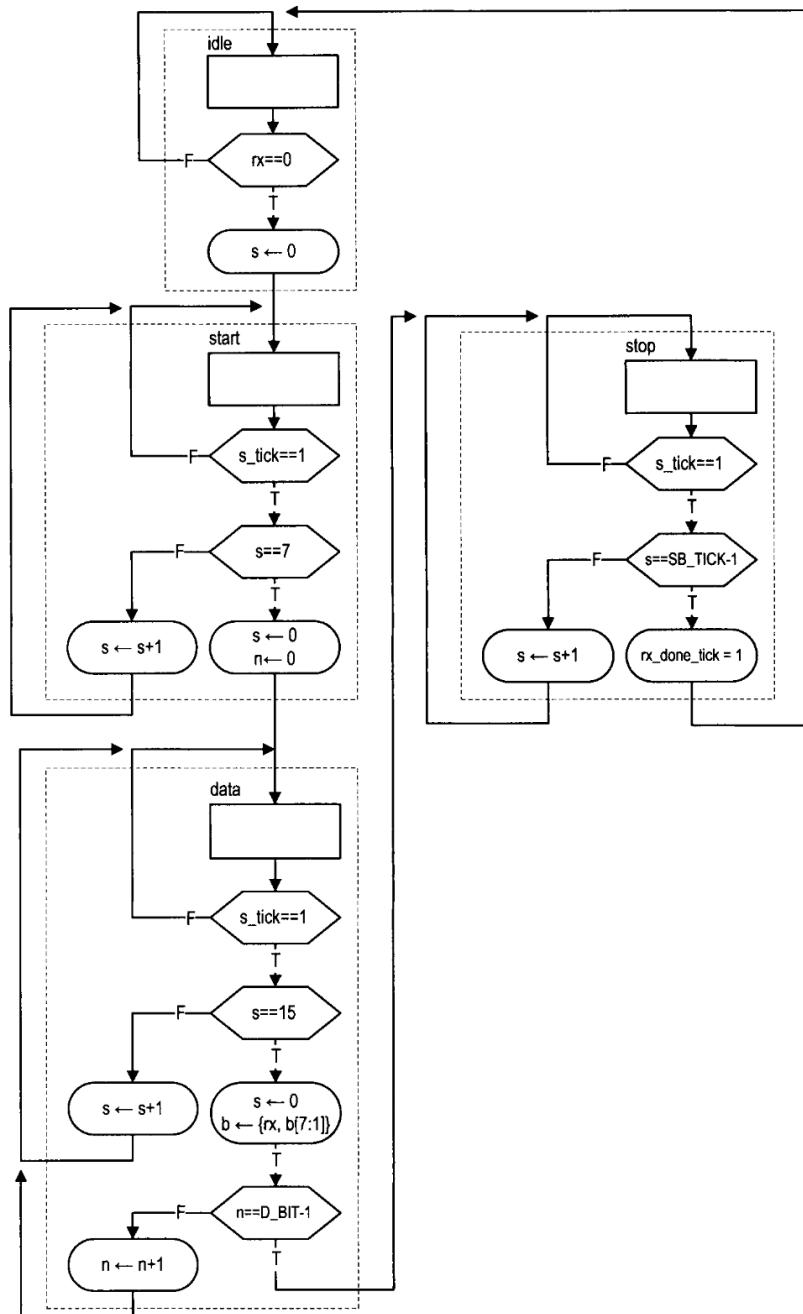
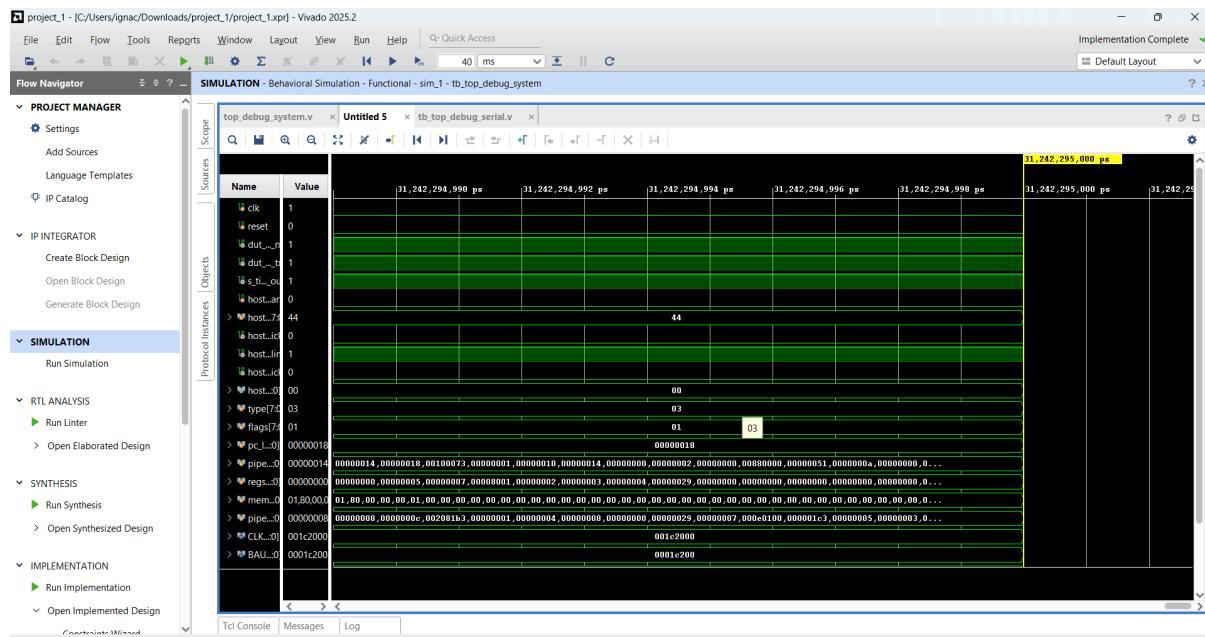
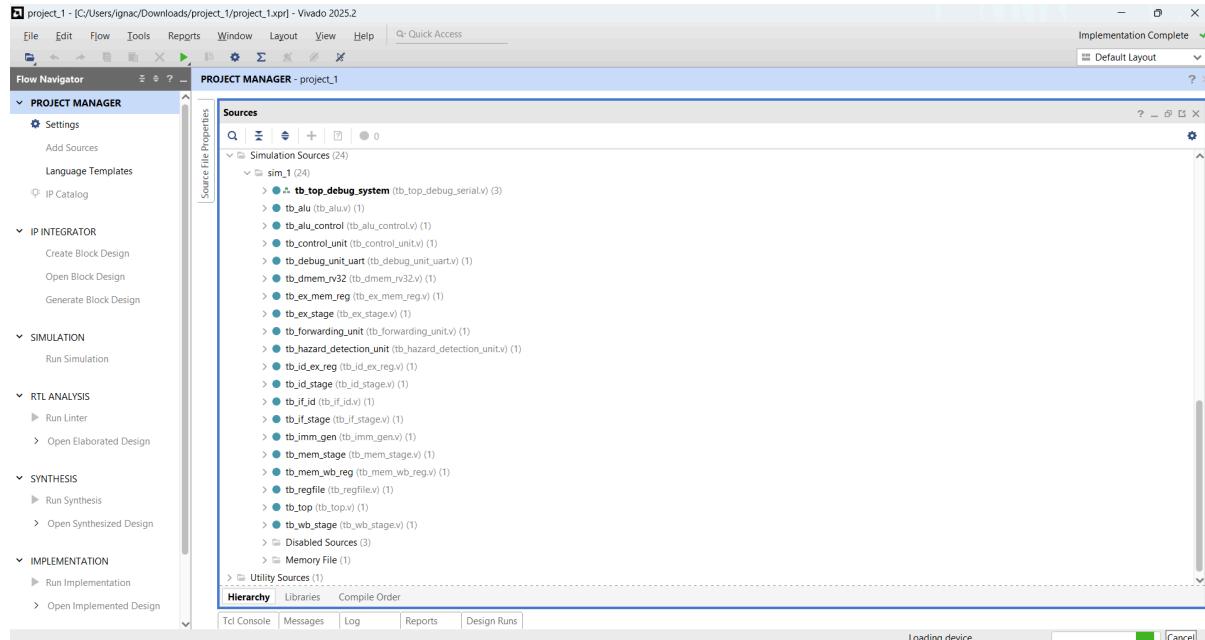


Figure 8.3 ASMD chart of a UART receiver.

Simulaciones

Durante el desarrollo del proyecto se realizaron simulaciones funcionales de cada módulo en forma individual, utilizando testbenches específicos para validar su comportamiento esperado. Una vez verificado el funcionamiento aislado, se avanzó con simulaciones de integración, comprobando la correcta interacción entre los distintos bloques del sistema, en particular el pipeline del procesador, la unidad de control y la Debug Unit con comunicación UART.



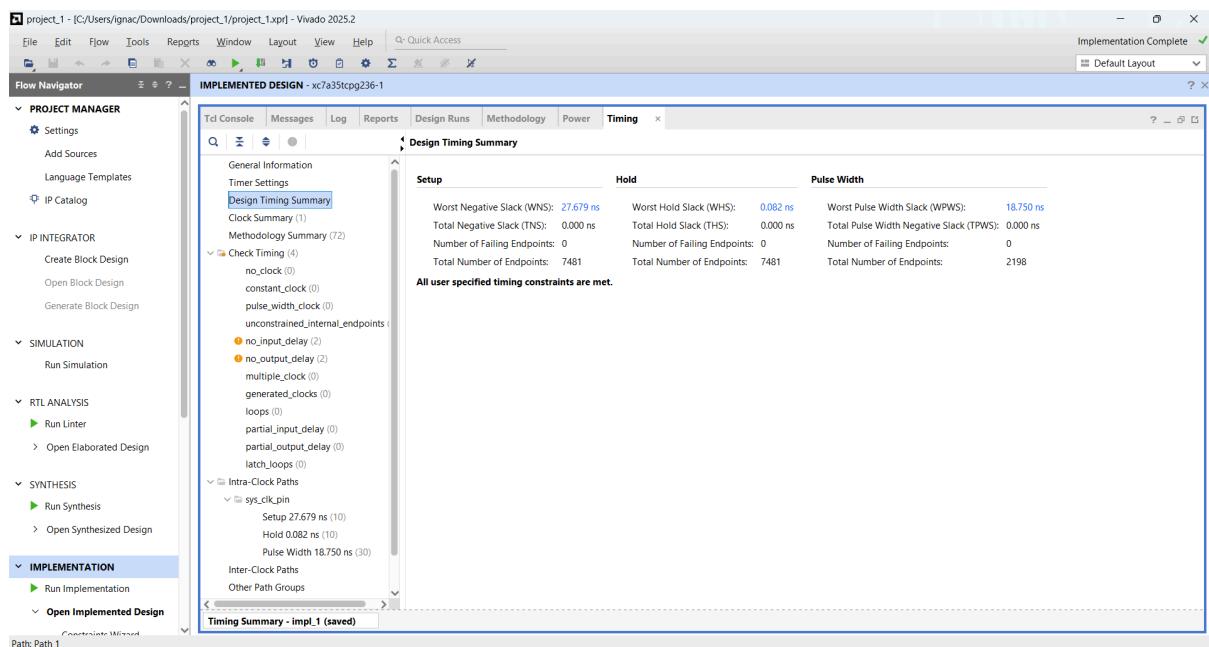
Clock

Llegada la integración deben preguntarse.

- ¿Cuál es el camino crítico de mi sistema?
- ¿Este camino crítico genera Skew en mi sistema? ¿Qué consecuencias tiene esto?

De encontrarse Skew:

- Encontrar la frecuencia de funcionamiento óptima de mi sistema.
- Generar métricas de funcionamiento utilizando las herramientas de Vivado.
- Aplicar la frecuencia de funcionamiento en mi sistema.



El Worst Negative Slack (WNS) representa el margen temporal mínimo del diseño, correspondiente al camino crítico. Un WNS positivo indica que todas las rutas cumplen con las restricciones de temporización, mientras que un WNS negativo implica violaciones de setup. En este diseño se obtuvo un WNS de +0.263 ns, lo que confirma el correcto cierre de temporización para la frecuencia seleccionada.

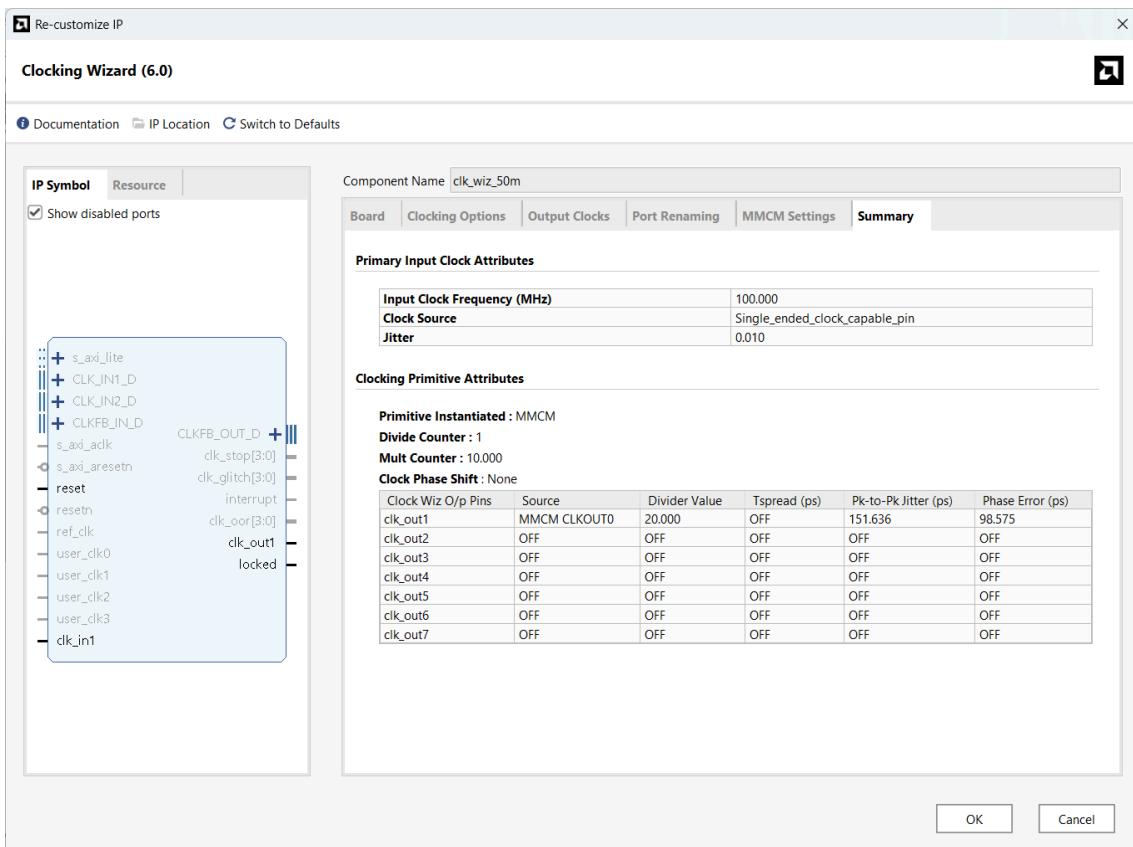
Se realizó un barrido de temporización post-implementación evaluando WNS para múltiples frecuencias objetivo. A partir de los resultados se seleccionó una frecuencia de operación según un criterio de robustez (margen mínimo de slack). Luego, para aplicar dicha frecuencia en hardware, se instanció un MMCM mediante Clock Wizard, generando un clock interno a la frecuencia seleccionada sin detener el clock del sistema. Finalmente, el baud rate generator de la UART fue calculado utilizando la nueva frecuencia de clock, manteniendo compatibilidad con el baud rate configurado en la PC.

Frecuencia analizada	Período de clock (ns)	WNS (ns)	Interpretación
----------------------	-----------------------	----------	----------------

100 MHz	10.00	+0.263	Funciona, margen muy reducido
80 MHz	12.50	+1.984	Margen adecuado
50 MHz	20.00	+8.531	Margen alto (frecuencia seleccionada)
25 MHz	40.00	+27.679	Margen excesivo

El aumento del período de clock incrementa el margen temporal disponible para el camino crítico, lo cual se refleja en un mayor WNS. A partir de este análisis se seleccionó una frecuencia de 50 MHz por ofrecer un margen amplio sin comprometer el desempeño del sistema.

Dado que la placa provee un oscilador físico de 100 MHz, se implementó un generador de clock interno mediante el Clock Wizard de Vivado, utilizando un MMCM, con el objetivo de derivar un clock de menor frecuencia sin detener el clock del sistema. El clock generado se distribuye a todo el sistema (CPU, unidad de depuración y UART), mientras que la señal **locked** del MMCM se utiliza para garantizar una liberación segura del reset, asegurando que la lógica opere únicamente cuando el clock es estable.



Programas de ejemplo

Programa 1 — ALU + shifts + branches + store/load

Qué prueba

- add/sub/xori/andi/ori
- slli/srli/srai
- slti
- beq/bne (una no tomada y una tomada)
- sw/lw para dejar resultados “visibles”

Qué deberías ver

- DMEM:
 - [0x20] = 0x0000000f (15)
 - [0x24] = 0x0000007fc
 - [0x28] = 0x000001ff
- Registros (si mirás al final):
 - x3=15, x8=0x7FC, x9=0x1FF, x12=1, x13..x15 reflejan lo cargado.

02000513
00500093
00a00113
002081b3
40110233
0f01c293
0ff2f313
10036393
00239413
00145493
4014d493
00622593
00058463
00100613
00059463
06300613
00352023
00852223
00952423
00052683

00452703
00852783
00100073

Programa 2 — Memoria completa (SB/SH/SW + LB/LBU/LH/LHU/LW)

Qué prueba

- Stores byte/half/word: `sb/sh/sw`
- Loads con extensión de signo y zero-extend: `lb/lbu/lh/lhu/lw`
- Casos con bytes “negativos” (0xD4, 0xFF) para verificar sign-extend.

Qué deberías ver (principal)

- `x11 = 0xA1B2C3D4`
- `x12 (lw) = 0xA1B2C3D4`
- `x13 (lb) = 0xFFFFFD4`
- `x14 (lbu)= 0x000000D4`
- `x15 (lh) = 0xFFFFC3D4`
- `x16 (lhu)= 0x0000C3D4`
- `x21 (lb 0xFF)= 0xFFFFFFFF, x22 (lbu 0xFF)= 0x000000FF`

02000513
a1b2c5b7
3d458593
00b52023
00b50223
00b502a3
00b51323
00052603
00050683
00054703
00051783
00055803
00150883
00154903
00452983
fff00a13
01450423
00850a83
00854b03

07f00b93
017504a3
00950c03
00000c93
00852c83
00100073

Programa 3 — Control de flujo (loop con BNE + BEQ) + JAL link

Qué prueba

- Loop con `bne` (suma 1..10)
- `beq` tomado para “camino OK”
- `jal` guardando link ($x_5 = PC+4$)
- Stores/loads para dejar evidencia en DMEM

Qué deberías ver

- DMEM:
 - $[0x20] = 55$ (0x00000037)
 - $[0x24] = x_5 \text{ link} = 0x00000034$ (el `jal` está en $PC=0x30 \rightarrow link=0x34$)
 - $[0x28] = 7$
- Registros:
 - $x_2=55, x_5=0x34, x_6=7, x_9=0x34, x_{11}=7$

02000513
00000093
00000113
00a00193
00108093
00110133
fe309ce3
00252023
00052203
03700393
00720463
00100413
008002ef
07b00313
00700313

00552223
00652423
00452483
00852583
00000013
00000013
00000013
00100073

Resultados

Subir capturas de la GUI y fotos de la placa funcionando.

Repositorio de GitHub

https://github.com/nachoborgatello/tp3_risc_v

Referencias

1. Chu PP (2008) FPGA prototyping by VHDL Examples: Xilinx Spartan-3 version. Wiley, New York
2. Patterson, D. A., & Hennessy, J. L. (2018). *Computer organization and design: The hardware/software interface (RISC-V ed.)*. Morgan Kaufmann.