Silicon-Level Design and FPGA Implementation of an 8-bit Five-Stage Pipelined RISC Microprocessor

Roberto Ibáñez Mingarro
Electronics & Embedded Systems Engineering (Double Degree: INSA Toulouse & Universitat Jaume I)

Abstract— This work presents the end-to-end definition, RTL implementation, synthesis-driven refinement and FPGA deployment of a fully pipelined 8-bit RISC microprocessor. The core follows a strict Load/Store execution model and a fixed-width instruction encoding, mapped onto a five-stage synchronous pipeline (LI, DI, EX, MEM, ER). The design is structurally decomposed into a hazard-aware register file, a flag-producing ALU, and synchronous instruction/data memories, with control logic engineered for deterministic stage behavior. Verification was performed via waveform-level simulation and validated through physical FPGA implementation, emphasizing synthesizability, timing robustness and architectural clarity.

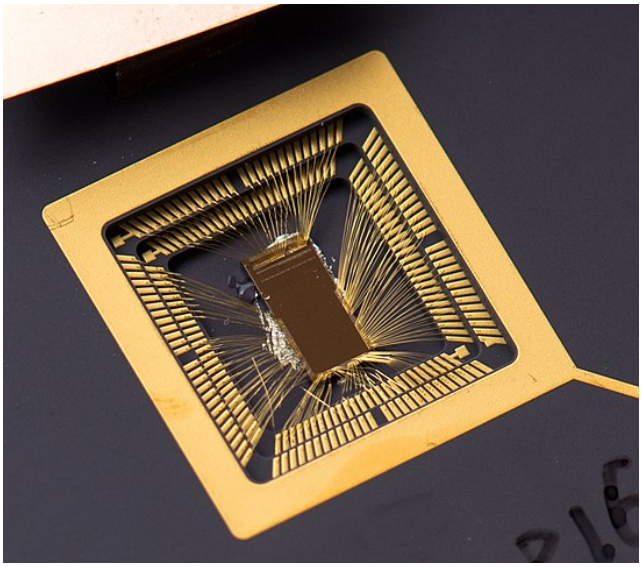

Figure 1: Commercial microprocessor die illustrating silicon-level integration density and implementation constraints.

1 Design Philosophy and Architectural Intent

The processor was designed with a pragmatic microarchitecture mindset: define a clean RISC datapath that is structurally regular, synthesizable without ambiguity, and capable of predictable timing closure on FPGA.

Instead of treating the design as a purely functional model, the implementation was approached as a hardware product: signal ownership is stage-local, pipeline boundaries are explicit, and all state elements are synchronous.

Figure 1 anchors the physical perspective: real processors are ultimately constrained by silicon integration density, routing, clock distribution and verification scale. This framing reinforces why pipeline regularity and hazard correctness matter even in compact cores.

2 FPGA Deployment Platform and Rationale

The processor was synthesized and deployed on a Xilinx Artix-7 FPGA (Basys 3 board), selected as a balanced validation vehicle for CPU-class RTL: sufficient fabric to map a pipelined datapath cleanly, native DSP resources to implement multiplication efficiently, and deterministic clocking infrastructure to validate synchronous stage behavior.

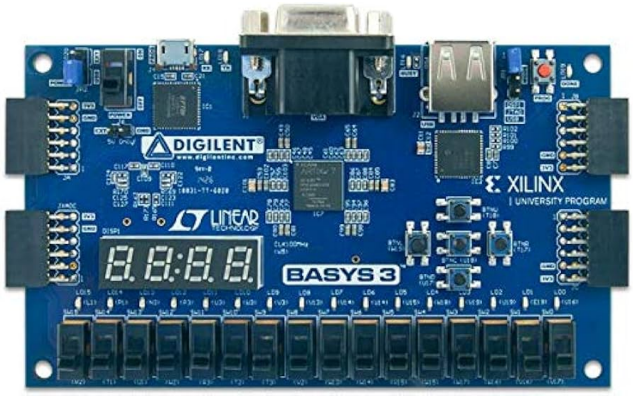

Figure 2: Basys 3 (Artix-7) FPGA platform used for synthesis-driven implementation and hardware validation.

This hardware grounding is critical: it forces the design to survive synthesis, placement and routing—exposing real critical paths, fanout pressure, and clock-domain assumptions that simulation alone does not fully surface. The chosen platform is shown in Figure 2.

Key FPGA primitives leveraged:

- **DSP slices** for hardware multiplication with predictable latency and reduced LUT pressure
- **Block RAM / distributed RAM** for memory structures and synthesis-friendly storage inference
- **Clock management** to ensure stable, low-jitter clocking and well-defined synchronous timing
- **Fine-grain LUT fabric** for control decode, hazard logic and datapath glue logic

Implementation constraints were applied to preserve stable clocking assumptions and deterministic pipeline behavior under full toolflow.

3 Pipeline Microarchitecture Overview

The architecture follows a five-stage pipeline partitioning:

- **LI** — instruction fetch and stage-local instruction

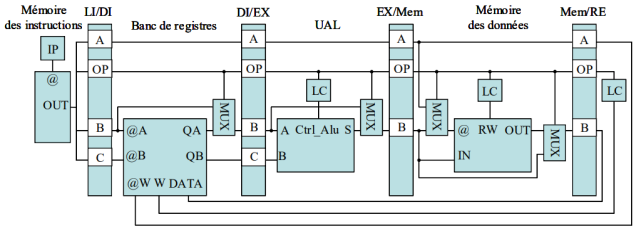

Figure 3: Global five-stage pipeline organization and functional datapath partitioning.

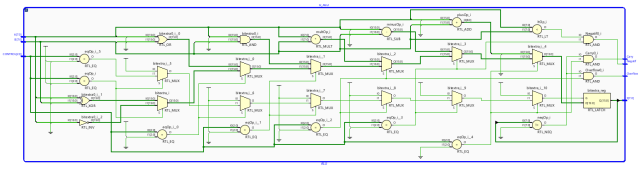

Figure 4: Structural ALU implementation with arithmetic/logical functions and status flag generation.

capture

- **DI** — decode and register operand acquisition
- **EX** — ALU execution and flag generation
- **MEM** — data memory access (Load/Store boundary)
- **ER** — register write-back commit

The global datapath integration is illustrated in Figure 3. This figure acts as the system-level "contract": it defines what each stage owns, what crosses stage boundaries, and where hazards must be addressed.

Each stage boundary is isolated by explicit pipeline registers, ensuring that timing closure is dominated by intra-stage combinational depth rather than multi-stage combinational leakage.

4 Execution Core: ALU and Flag Semantics

The ALU is a structurally decomposed execution block supporting arithmetic and logical operations with flag generation for architectural visibility.

The implementation, shown in Figure 4, includes:

- **ADD / SUB** with carry-aware behavior
- **MUL** mapped to FPGA DSP resources to avoid LUT-based multipliers
- **AND / OR / XOR / NOT** as single-cycle logic functions

Status flags are produced deterministically:

- **N** — negative result indication (sign semantics over 8-bit result)
- **O** — multiplication overflow indication (result width exceeds 8 bits)
- **C** — carry indication for addition

From a timing standpoint, carry propagation and decode-controlled muxing are typical contributors to the EX-stage critical path, which is why the ALU boundary was kept stage-local.

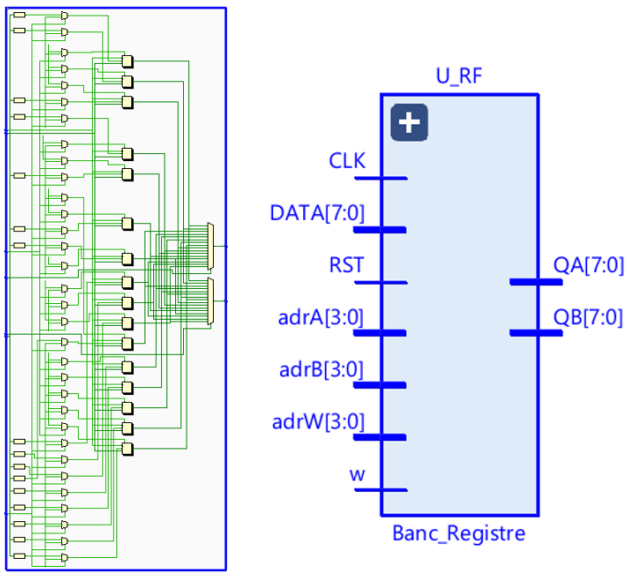

Figure 5: Register file with dual-read ports, synchronous write, and integrated bypass (D→Q).

5 Register File: Dual-Read, Single-Write with Bypass

The register file implements 16 × 8-bit storage with two simultaneous read ports and one synchronous write port.

Its structural implementation is shown in Figure 5. The key feature is a built-in bypass (D→Q) path, which resolves the common read-after-write collision case without injecting bubbles:

$$if(W = 1) \wedge (@Read = @Write) \Rightarrow Q \leftarrow DATA$$

This local bypass reduces hazard pressure at the pipeline level, simplifying control logic and improving throughput for back-to-back dependent operations.

6 Memory Subsystem: Synchronous DMEM and ROM-Like IMEM

A strict Load/Store model was enforced: only LOAD and STORE interact with data memory, while arithmetic and logical operations remain register-to-register.

6.1 Data Memory (DMEM)

The data memory is synchronous and supports controlled read/write behavior, as shown in Figure 6. This structure ensures deterministic latency and eliminates asynchronous read assumptions that often degrade timing predictability.

6.2 Instruction Memory (IMEM)

Instruction memory is treated as ROM-like during execution: program contents are stable and read synchronously, as shown in Figure 7. This clean separation

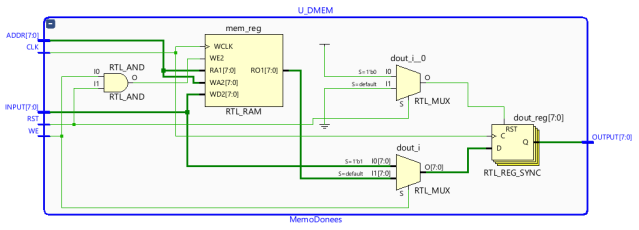

Figure 6: Data memory implementation with synchronous behavior suitable for FPGA inference.

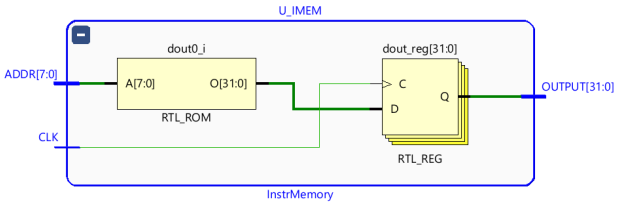

Figure 7: Instruction memory (ROM-like) with synchronous output register.

improves verification clarity and aligns with Harvard-style separation at the architectural level.

7 Hazards, Stalls, and Deterministic Control

Pipeline correctness is primarily threatened by data hazards. The design addresses hazards through a layered approach:

- **Local bypassing** inside the register file for same-cycle collisions
- **Inter-stage hazard detection** comparing destination/source fields across adjacent stages
- **Bubble injection (NOP)** and selective pipeline inhibition when forwarding is insufficient

The guiding principle is deterministic behavior: when a hazard is detected, the pipeline response is explicit and stage-safe, avoiding accidental multi-write, metastability assumptions, or partially updated state.

8 Verification: Waveform-Level Proof of Behavior

Functional verification was performed using a dedicated VHDL testbench, verifying:

- correct stage propagation
- correct write-back commit timing
- correct memory semantics for LOAD/STORE
- correct hazard resolution behavior

The waveform evidence is shown in Figure 8. This is where microarchitecture intent becomes observable: control decisions, stage timing, and register visibility can be proven cycle-by-cycle.

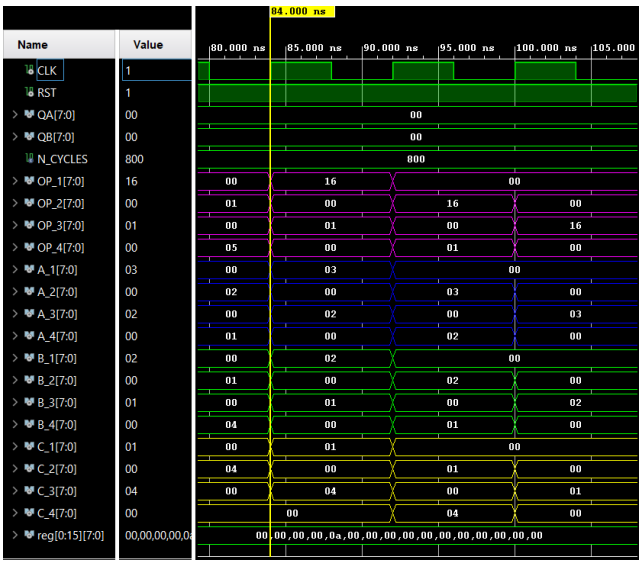

Figure 8: Testbench waveform showing pipeline activity, register updates, and instruction progression.

Figure 9: Post-implementation FPGA mapping (region view 1), highlighting slice utilization and routing density.

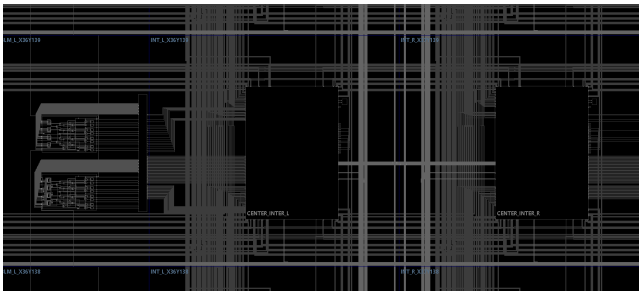

Figure 10: Post-implementation FPGA mapping (region view 2), illustrating placement distribution and interconnect complexity.

9 Implementation: FPGA Mapping and Physical Realization

Synthesis and implementation were pushed through the full FPGA toolflow, transforming RTL into placed-and-routed hardware.

The resulting resource mapping is shown in Figures 9 & 10. These views provide a physical sanity check: routing congestion, locality, and region utilization reflect whether the design is structurally clean or tool-hostile.

EX-stage depth (carry + muxing) and decode fanout dominate timing, making hardware implementation the true indicator of frequency limits.