Design and Implementation of Three-stage RISC-V Microprocessor for Education

Jan Ber
The Faculty of Computer Science, Electronics and Telecommunications
AGH University of Krakow
Krakow, Poland

***Abstract*—In many computer science and embedded systems courses, processor architecture is typically introduced using either the classical five-stage RISC pipeline or the two-stage AVR pipeline. While these architectures provide a strong introduction to bare-metal programming and computer organization, they do not necessarily represent the most practical or accessible approach for teaching modern processor architectures. In particular, traditional five-stage pipelines often introduce considerable control and forwarding complexity, while compact microcontroller architectures may obscure fundamental pipeline behavior and instruction flow.**

This paper presents X3S, a work-in-progress 32-bit pipelined RISC-V softprocessor core originally developed as author's engineering thesis. The design adopts a simple three-stage pipeline intended to balance architectural clarity, implementation complexity, and achievable operating frequency on Field Programmable Gate Array (FPGA) devices. X3S implements the RV32I base integer instruction set together with the M extension for multiplication and division operations. Multi-cycle arithmetic accelerators are integrated into the execution pipeline using a handshake-based interface.

***Index Terms*—RISC-V, RV32I, soft processor, microprocessor, FPGA, pipelined processor**

I. Introduction

Effective software development, particularly for students in embedded systems education and computer science, requires more than familiarity with high-level programming abstractions. In practical systems, performance, determinism, and resource usage are directly determined by the underlying hardware architecture. Consequently, understanding how software maps onto hardware execution is critical for writing efficient and reliable bare-metal code. This is especially relevant in embedded systems education, where students must reason about execution on in-order processors, constrained memory and compute resources, and timing-sensitive peripherals.

However, educational approaches to processor architecture do not always reflect this requirement in a balanced way. In many curricula, instruction is based either on compact microcontroller architectures such as AVR or on classical five-stage RISC pipelines. These two approaches represent opposite ends of the design space in terms of complexity and visibility of internal execution.

AVR-based systems offer a very accessible programming model with minimal architectural overhead, which makes them suitable for introducing low-level programming concepts [1]. However, their execution model and memory organization, including the separation of program memory, data memory, and peripheral address spaces (e.g., memory-mapped I/O such as GPIO), provide limited insight into timing behavior and execution dynamics present in more recent embedded processors [2]. In particular, instruction execution is largely sequential with overlap limited to fetch and execution, which makes for highly predictable timing and abstracts away effects such as variable memory latency and wait states. Consequently, the relationship between instruction execution, memory access, and peripheral interaction is partially obscured, as illustrated in Fig.1.

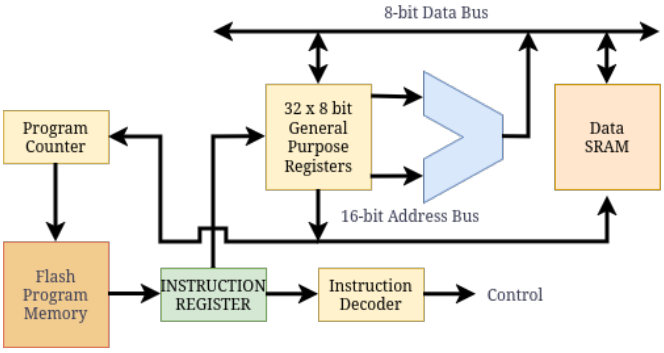

Fig. 1. Simplified dataflow in AVR processors.

In contrast, classical five-stage RISC pipelines are often used as a teaching model to explain instruction-level parallelism and pipelined execution. While this abstraction is useful for teaching the principles of pipelining, it does not directly correspond to many contemporary embedded processor architectures. Modern microcontroller cores, such as those in the ARM Cortex-M family, typically employ pipeline organizations that differ from the classical five-stage model. For example, Cortex-M3 and Cortex-M4 cores use a simple three-stage pipeline (fetch, decode, execute), while larger designs such as Cortex-M7 introduce deeper pipelines with additional buffering, speculation, and dual instruction issue [3].

Author's intention in this paper is to argue that a three-stage pipeline provides a more effective educational balance between architectural clarity and implementation complexity than either compact microcontroller models or classical five-stage RISC pipelines. To this end, X3S processor is used as a case study to demonstrate this design point.

Manuscript received May 24, 2026; revised August 10, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Short
DOI: doi.org/10.64552/wipiec.v12i2.150

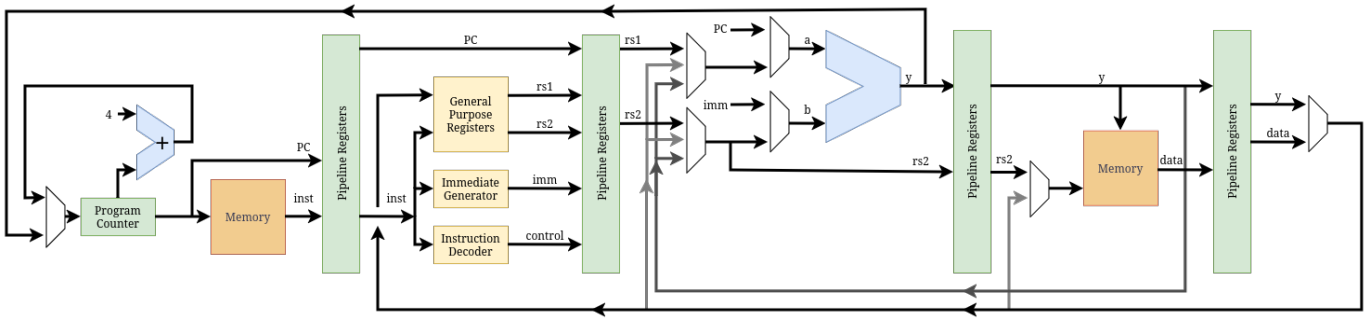

Fig. 2. Dataflow of a typical five-stage RISC-V processor.

TABLE I
EXECUTION TIMELINE IN A 3-STAGE PIPELINE (E.G., CORTEX-M3/M4-CLASS CORES).

Cycle	IF	ID	EX
N	inst0		
N+1	inst1	inst0	
N+2	inst2	inst1	inst0
N+3	inst3	inst2	inst1
N+4	inst4	inst3	inst2

II. FIVE-STAGE PIPELINE

The classical five-stage pipeline is a well-established abstraction used to introduce instruction-level parallelism and pipelined execution. While it has historically been a common design model, it is now primarily used as a teaching reference rather than an accurate representation of most contemporary embedded processor architectures. In this work, it serves as a baseline model for explaining pipeline concepts and for comparison with simpler pipeline organizations.

This section uses a standard RISC-V five-stage datapath, as commonly presented in literature [4], consisting of instruction fetch (IF), instruction decode (with register read) (ID), execute (EX), memory access (MEM), and write-back (WB), as illustrated in figure 2. The model represents an in-order pipeline where multiple instructions are processed concurrently at different stages as shown on Table II.

TABLE II
EXECUTION TIMELINE IN A FIVE-STAGE RISC-V PIPELINE.

Cycle	IF	ID	EX	MEM	WB
N	inst0				
N+1	inst1	inst0			
N+2	inst2	inst1	inst0		
N+3	inst3	inst2	inst1	inst0	
N+4	inst4	inst3	inst2	inst1	inst0
N+5	inst5	inst4	inst3	inst2	inst 1

A. Data hazards and forwarding

While instruction pipelining improves throughput, it introduces several classes of hazards that can prevent the next instruction from executing in the intended cycle. These hazards arise from resource conflicts (such as register file ports or execution units), control flow changes, and most importantly in the context of this paper, data dependencies between instructions.

A *data hazard* occurs when an instruction depends on the result of a previous instruction that has not yet completed its passage through the pipeline. In a classic five-stage pipeline, this is particularly relevant when a register is read during the instruction decode (ID) stage before a preceding instruction has written its result back in the write-back (WB) stage. As a result, the required operand may still be in flight, and the value read from register file is invalid.

A common example is a read-after-write (RAW) dependency:

```
inst1: add x1, x2, x3
inst2: sub x4, x1, x5
```

Here, `inst2` requires the value of `x1` produced by `inst1`. In a non-pipelined design, this dependency is trivially resolved because `inst1` completes before `inst2` begins. However, in a pipelined processor, `inst2` may reach the ID stage before `inst1` has written its result back to the register file. This creates a hazard where the decode stage would read a stale value of `x1` unless a forwarding mechanism is used.

In a five-stage pipeline, implementing data forwarding requires additional multiplexing logic to route results between pipeline stages. Three forwarding multiplexers are typically required: two in the EX stage and one in the MEM stage.

In RISC-V processors, forwarding logic must be integrated with the operand selection already present in the ALU datapath. ALU inputs are selected between register operands (`rs1`, `rs2`), immediate values, and the program counter (PC). Supporting forwarding therefore either increases the width of the operand selection multiplexers or introduces additional multiplexing stages in series. Although the required hardware is modest, wider multiplexers and cascaded selection logic increase combinational delay and extend the processor critical path.

B. Do we need WB stage?

The WB stage is primarily responsible for selecting between the ALU result and memory read data before writing the selected value back to the register file. Functionally, this stage often consists only of a small multiplexer together with the register file write interface. On FPGAs designs, memory

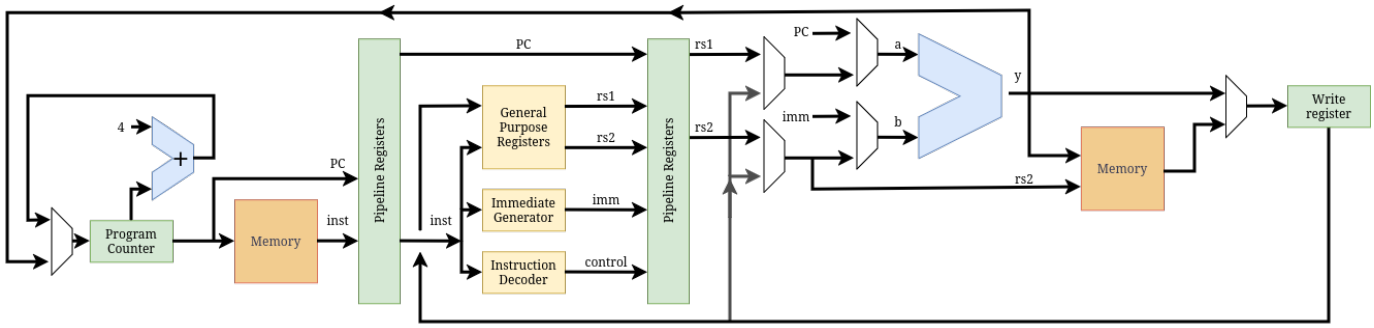

Fig. 3. Proposed pipeline organization with combined EX-MEM-WB stages.

access is fast compared to wide multiplexer networks, making it practical to merge the write-back selection into the MEM stage. This reduces the number of forwarding sources, making forwarding multiplexers in EX stage shallower since data only needs to be forwarded from a combined MEM/WB stage.

C. Do we need a MEM stage?

A similar argument can also be made for the MEM stage, memory access occupies a dedicated stage positioned between execute and write-back. However, load instructions already require the pipeline to stall whenever memory data is not immediately available, meaning that the MEM stage does not necessarily provide fully independent pipeline progress.

For load instructions, the EX stage can compute the effective address and issue the memory request by storing the address in a register connected to the memory interface. The pipeline then stalls until the requested data is returned. Once available, the loaded value can be written directly into a result register used both for forwarding and for register file write-back.

Store instructions can be handled similarly by storing the computed address together with the value to be written into dedicated registers associated with the memory interface. Since store operations do not immediately produce values required by subsequent instructions, execution may continue while the memory subsystem completes the write transaction.

Under this organization, the EX stage effectively becomes the final active stage of the pipeline. Arithmetic results and completed memory responses are written into a shared result register that feeds both the forwarding network and the register file. Proposed organization is shown on figure 3.

III. X3S softprocessor

X3S is a 32-bit RISC-V soft processor core implementing the RV32I base integer instruction set together with the M extension for integer multiplication and division. The design is released as an open-source project and was originally developed as engineering thesis of the author [5].

X3S processor implements an in-order pipeline organized into three stages: instruction fetch (IF), instruction decode (ID), and execute (EX). The design follows a strict single-issue model in which the IF stage is capable of fetching one instruction per clock cycle, provided that older stages aren't stalled. The ID stage performs instruction decoding, register file access, and immediate generation, but does not participate in execution or memory interaction beyond operand preparation. As a result, the ID stage is purely combinational in nature from a datapath perspective, with state changes occurring only through pipeline registers between stages.

The EX stage represents the architectural center of the processor and is responsible for all operations that modify architectural state. This includes arithmetic and logical computation, branch condition evaluation, address generation, and memory transactions. Importantly, only the EX stage is permitted to update the program counter, meaning that control-flow redirection (jumps and taken branches) is resolved exclusively at this stage. This introduces a fundamental control hazard: when a branch or jump is taken, the instructions already fetched in IF and decoded in ID must be invalidated, resulting in at least three cycle control penalty.

Memory operations are mediated entirely by the EX stage using a request–response handshake with an external memory system. Store instructions are issued in the same cycle in which the effective address and write data become available. If the memory interface does not assert backpressure, the pipeline may continue to accept consecutive store instructions at a rate of one per cycle. Load instructions, in contrast, always introduce a stall condition and hold pipeline progress until valid data is returned from memory.

A. Instruction Fetch (IF)

The instruction fetch stage maintains the program counter (PC), issues instruction memory requests, and forwards fetched instructions to the decode stage. In normal operation, next PC is incremented by four each cycle, corresponding to the fixed 32-bit instruction width, fetching one instruction per cycle.

On a pipeline stall, the PC and IF output registers are frozen, and no new memory requests are issued. The most recently fetched instruction and its PC are held at the IF–ID boundary, allowing the decode stage to continue operating on a valid instruction until the stall is released. Execution then resumes without re-fetching or re-decoding.

Control-flow redirection is resolved using a branch signal from the execute stage. When a branch or jump is taken, the PC

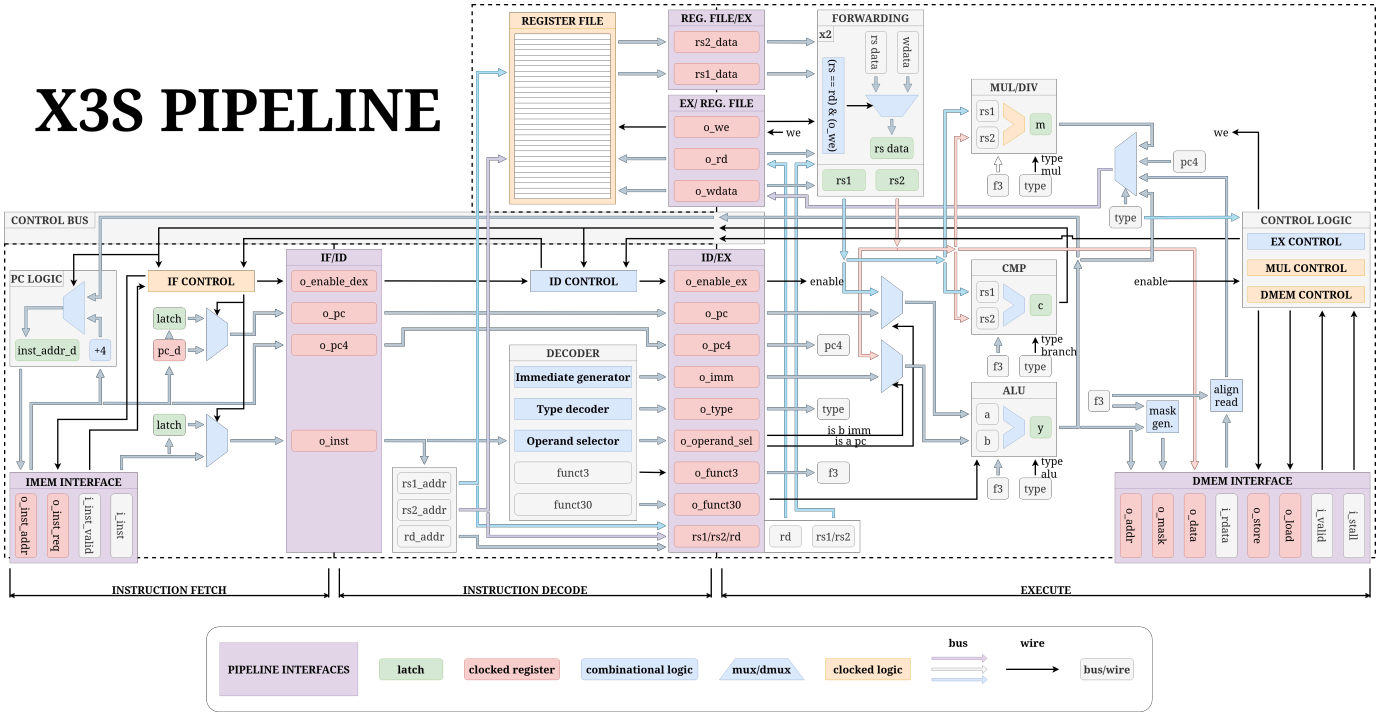

Fig. 4. Dataflow in the X3S processor.

is updated with the target address and any previously fetched instructions are invalidated.

B. Execute (EX)

The Execute stage (EX) acts as the integration point for all non-trivial datapath behavior in the X3S processor. Rather than introducing additional pipeline stages, EX aggregates several functional subsystems and resolves their interactions through internal control logic. Its primary architectural role is therefore not only computation, but also coordination of dependent execution units.

EX contains the operand forwarding network used to resolve read-after-write hazards. This forwarding logic is coupled with the operand selection stage and ensures that dependent instructions can execute without introducing additional pipeline stalls, provided that results are already available within the stage. The execution datapath also includes the Arithmetic Logic Unit(ALU) and comparator (CMP) logic used for arithmetic, logical and address generation, and branch condition evaluation respectively.

EX also acts as the exclusive owner of all external execution-side interface, the data memory interface is fully controlled within this stage. Both load and store transactions are initiated from EX, and all associated control signals (address, write data, and byte enable masks) are generated locally. Load operations are additionally governed by an internal control state machine that tracks request issuance and response validity, allowing the pipeline to be stalled only when required by memory latency.

EX also integrates the multiplication and division unit (M-unit). This block is decoupled from the main combinational datapath and interacts with EX through a handshake protocol based on start and completion signals. From the perspective of EX, the M-unit behaves as a latency-variable execution accelerator. When used, EX stalls the pipeline until completion is signaled, which ensures correct serialization at expense of throughput (since under this model, M-unit cannot be pipelined).

Overall, EX serves as the execution stage in which all architectural effects are resolved, including forwarding, memory transactions, and accelerator-style execution units within a single execution unit.

C. Verification

The X3S processor was verified using both simulation and FPGA implementation. Simulation-based verification was performed using `Icarus Verilog` and `vvp`. The testbench instantiates the processor together with a behavioral memory model and supports automated execution of RISC-V ISA tests [12]. Test completion and status are signaled via memory-mapped writes, enabling batch verification.

Hardware validation was carried out on a *Spartan-7 XC7S50-1* FPGA as part of the X3S-Hydrogen microcontroller, which integrates the core with memory and Universal Asynchronous Receiver Transmitter (UART). X3S-Hydrogen has been synthesized using Vivado Design Suite, and verified on FPGA with clock frequencies up to 100 [MHz]. The maximum clock frequency is limited by the critical path is in the EX stage and is caused by operand selection and forwarding multiplexers

before the ALU. Resource usage by X3S processor is shown on the Table III.

TABLE III
RESOURCE USAGE BY X3S PROCESSOR IMPLEMENTED IN X3S-HYDROGEN.

Module	Slice LUTs	Slice Registers	BRAM	DSP
X3S-Hydrogen	1697	1176	8	3
Pipeline	1574	935	–	3
IF	128	227	–	–
ID	600	127	–	–
EX	721	517	–	3
MUL	245	134	–	3
DIV	325	202	–	–
PeripheralBus	4	55	–	–
UART	74	126	–	–

IV. Conclusion and further work

This paper presented X3S, a 32-bit RISC-V soft processor implementing a simplified three-stage pipeline designed for educational use. The architecture was motivated by the limitations of both classical five-stage pipelines and minimal process models in teaching. By merging and reorganizing pipeline stages, X3S reduces structural complexity while preserving essential concepts such as instruction-level parallelism, data hazards, and control flow effects. The implemented design demonstrates that a three-stage organization can achieve a practical balance between clarity and hardware efficiency on FPGA platforms.

Further improvements to the X3S processor could be made by relocating parts of the operand selection logic from the EX stage to the ID stage, which would leave only the forwarding multiplexers in the execution stage. In addition, introducing an extra adder in the ID stage would allow address calculations for jumps and memory operations to be performed one cycle earlier, simplifying the EX-stage control logic. This change would require operand forwarding within the ID stage; however, this can be supported by implementing the register file using BRAMs, whose access time is typically comparable to or faster than the additional combinational adder, ensuring that register reads and address generation do not become the critical timing bottleneck.

References

[1] Microchip Technology Inc., *Avr instruction set manual*, 8-bit AVR architecture reference, n.d. Accessed: May 21, 2026. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf

[2] Microchip Technology Inc., *Atmega328p datasheet*, AVR microcontroller documentation, n.d. Accessed: May 21, 2026. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega328P_Datasheet.pdf

[3] Arm Ltd., *Armv7-m architecture reference manual*, Cortex-M architecture reference, n.d. Accessed: May 21, 2026. [Online]. Available: https://developer.arm.com/documentation/ddi0403/latest/

[4] S. L. Harris and D. M. Harris, *Digital Design and Computer Architecture: RISC-V Edition*, 2nd ed. Morgan Kaufmann, 2021.

[5] Jan Ber. "X3s risc-v soft processor core." Open-source FPGA soft-core implementation, Accessed: May 21, 2026. [Online]. Available: https://codeberg.org/jber/little-risc-v-softcore