Teaching Reform and Practice of the Computer Systems Course Based on a RISC-V Pipelined Processor Design

Junxiang Gao and Lihua Bie*

College of Informatics, Huazhong Agricultural University, Wuhan, 430070, China
gao200@mail.hzau.edu.cn, *biebie@mail.hzau.edu.cn

Abstract. The computer systems course is a cornerstone of computer science education, bridging software and hardware knowledge and cultivating system-level engineering competence. Traditional teaching models often fragment content and lack depth in practical training. Thus, students may understand individual components such as arithmetic units and memory subsystems, yet struggle to see how software executes on hardware as a unified system. Moreover, the long-standing focus on the MIPS instruction set has diverged from current industry trends, which increasingly favor open architectures like RISC-V. This misalignment makes it difficult for traditional teaching to address modern engineering and practical skill needs. In response, this paper presents a comprehensive reform for the computer systems course that prioritizes the cultivation of system capabilities. Capabilities are structured hierarchically, with course and practical activities adjusted accordingly. The course features a project-based approach centered on designing and implementing a five-stage pipelined processor using the RISC-V RV32I instruction set. Students progress through the full process, including understanding the instruction set, designing modules, integrating the system, and verifying results. Outcomes show significant gains, as excellent course projects rose from 25% to 60%, and system capability attainment increased from 40% to 88%. These results confirm the reform's effectiveness in strengthening systematic thinking and practical skills, providing a valuable reference for similar course reforms.

Keywords: computer systems course, system capability cultivation, RISC-V instruction set, pipelined processor, teaching reform.

1 Introduction

The computer systems course is fundamental for computer-related majors. It covers machine-level data representation, arithmetic unit structure, hierarchical memory, instruction set architecture, and central processing unit design. Students must understand both individual hardware mechanisms and their interactions within the complete system, enabling them to explain program execution in detail. This course bridges programming skills and knowledge of system architecture.

Extensive teaching practice suggests that traditional lecture-centered delivery and fragmented experiments often leave students with weak system-level understanding

B. Kanoksilapatham et al. (eds.), *Proceedings of the 2026 2nd International Conference on Education Reform, Ideology and Politics (ERIP 2026)*, Advances in Social Science, Education and Humanities Research 1028,
https://doi.org/10.2991/978-2-38476-601-7_7

and limited ability to connect concepts across layers. Students may complete isolated tasks yet struggle to explain how instructions, data paths, and control interact in a running system. Common difficulties include treating the datapath, control, and memory subsystem as separate topics, and failing to connect timing with correctness when moving from diagrams to code [1]. In many offerings, course topics are compartmentalized: lectures cover datapaths and controllers in isolation, while laboratory work focuses on verifying single modules, which weakens students' ability to connect concepts across the full execution flow. This separation can leave students with "local correctness" in small units but limited confidence when they must integrate modules, reason about interfaces, and debug system-level behavior.

At the same time, rapid hardware–software co-evolution and industry demand increasingly require graduates to reason about complete system execution, not only component knowledge. This creates pressure to align course learning outcomes with practical, system-oriented engineering capability. As a result, graduates may be able to recite definitions but find it hard to implement a coherent subsystem or to explain why an integrated design fails under specific instruction sequences. The shift of computer architecture practice toward open, modern ecosystems also matters. MIPS, once common in teaching, is now less representative of industrial toolchains, whereas RISC-V's open, modular ISA has gained broad academic and industry attention and provides a realistic basis for architectural design and hands-on implementation.

To address these gaps, this paper reforms the computer systems course around a RISC-V pipelined processor as a unifying practical project, aiming to strengthen system cognition and hands-on implementation ability while keeping the teaching workload feasible. The reform emphasizes "learning-by-building": students repeatedly relate abstract concepts to concrete signals, timing, and observable behavior in the pipeline. Accordingly, the paper presents the reform from objectives and content organization to implementation and effect analysis, so that the motivation, teaching methods, and outcomes form a coherent narrative [2].

2 Objectives and Teaching Design of the Course Reform

2.1 Reform Objectives

This reform targets system-oriented competence: students should understand how a computer system executes programs end-to-end and be able to apply core principles in implementation and debugging, addressing the common disconnect between theory and practice. It also aims to cultivate disciplined engineering habits, including interface specification, incremental integration, and evidence-based debugging.

System cognition emphasizes building a coherent mental model of how instructions flow through the datapath and control, how memory and I/O are accessed, and how modules cooperate to deliver correct execution. The goal is for students to explain system behavior using architectural concepts and to diagnose errors by tracing execution across stages. In practice, students are encouraged to switch between architectural diagrams, RTL modules, and waveforms so that the same mechanism can be understood at multiple representations [3].

2.2 Reform Approach

Centered on these objectives, the course adopts a project-driven structure using a five-stage RISC-V pipeline as the "spine" that links instruction, datapath, control, and verification. Lectures and labs are reorganized to follow system execution rather than isolated topics [4]. Each lecture topic is mapped to a concrete module or behavior that will be implemented or tested in the corresponding lab, so that theory directly reduces project uncertainty.

In implementation, teaching alternates between concept explanation and incremental construction of modules, with frequent checkpoints and demonstrations. Students progress from understanding the overall pipeline to completing key components and integrating them into a working system. Checkpoints require students to submit intermediate artifacts (module code, simulation evidence, and brief explanations), which reduces last-minute integration and makes learning progress visible.

2.3 Organization of Teaching Content

Content is reorganized around capability cultivation: core concepts are retained but rearranged to match the execution flow of a pipelined processor, and laboratory work is aligned with the corresponding theory so that each topic immediately supports a concrete design task. The resulting structure retains the original knowledge coverage, but reduces repetition and emphasizes the causal chain from instruction semantics to hardware signals and to program-visible outcomes (Table 1).

Table 1. Mapping between course content and system capability cultivation

Teaching Content	Teaching Requirements	Capabilities Cultivated
Introduction of computer systems	Understand Amdahl's Law, locality, and the integration of hardware and software.	System cognition
Logic circuits and EDA technology	Design combinational and sequential circuits using EDA tools.	System cognition and design
Arithmetic methods and units	Learn data representations; Master arithmetic operations, exceptions, and floating-point.	System design and development
Instruction set architecture	Understand instruction formats and addressing modes; Compare instruction design principles.	System design and development
Central processing unit	Understand CPU components and control; Master pipelining and branch prediction.	System design, development, and application

3 Teaching Implementation Based on a RISC-V Processor

3.1 Overall Design of Practical Teaching

Practical teaching centers on implementing a five-stage RISC-V pipelined processor, which serves as the main thread for the course. Students complete modular design, integration, and testing, so that architectural concepts are learned through building a running system. The project begins with a reference top-level architecture and a minimal test harness, after which students implement or complete key modules in a prescribed order to reduce integration risk.

Compared with confirmatory, single-module labs, the project approach requires students to connect modules through interfaces and timing, and to reason about hazards and control. This supports a transition from "doing tasks" to explaining system behavior. It also encourages students to practice reading and writing interface specifications, since mismatched assumptions about widths, control timing, or reset behavior are common sources of failure in real designs. The project is not intended to maximize performance or add complex features. It is designed to make key pipeline mechanisms observable and debuggable, helping students form stable system intuition while keeping the scope manageable for a teaching schedule [5, 6]. By keeping the instruction set and peripheral scope limited, most student time can be spent on understanding the pipeline's essential invariants (e.g., what must hold true at each stage boundary) and on learning systematic verification.

3.2 Instruction Set Selection and Toolchain Configuration

The practical project uses the core RV32I integer subset to keep the instruction set small, stable, and well supported by tools. This choice reduces learning overhead while still covering representative control flow, arithmetic, and memory-access behaviors needed to illustrate pipeline design. The subset selection also simplifies compiler outputs, making it easier to relate generated assembly to the expected datapath actions during teaching and debugging.

A classic five-stage pipeline is used, with clearly separated fetch, decode, execute, memory, and write-back stages (Figure 1). This structure is simple enough for teaching while still enabling discussion of control, data hazards, and forwarding/stalling mechanisms. A small hazard-handling scheme (forwarding and stalling) is introduced so that students can see why hazards occur and how control resolves them without overwhelming design complexity.

For implementation, a mainstream FPGA development workflow is adopted (HDL design, synthesis, place-and-route, and on-board testing) to match common engineering practice and to ensure students can reproduce results in the lab environment. In the lab, students use the FPGA vendor toolchain for synthesis and implementation, together with waveform simulation for pre-board verification; this keeps the workflow consistent with typical digital design practice.

FPGA resource evaluation indicates that the selected teaching platform provides sufficient logic and memory resources for the pipeline design, allowing students to focus

on correctness, integration, and verification rather than resource constraints. For example, the Cyclone-V SoC teaching board (e.g., 5CSEMA5F31C6N class devices) offers ample logic elements, on-chip memory blocks, and I/O for the pipeline, basic peripherals, and test harnesses used in the course.

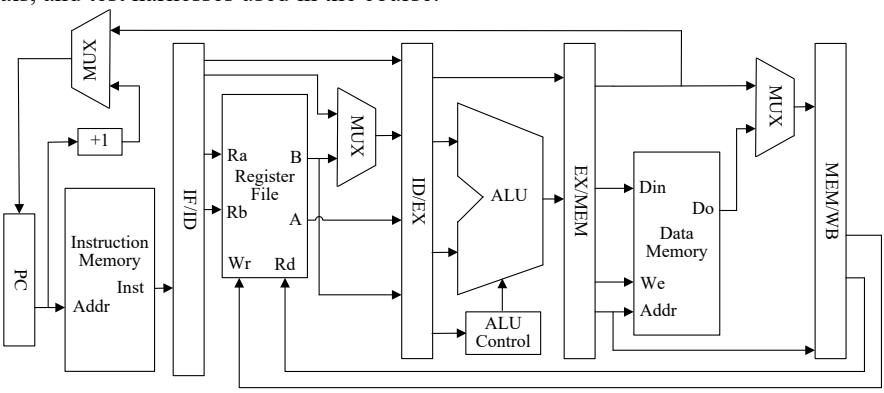

Fig. 1. Teaching schematic of a five-stage pipeline processor based on RISC-V.

3.3 System Implementation, Validation, and Teaching-Oriented Adaptation

Implementation proceeds module by module following the pipeline structure, with clear interface definitions and staged integration. Students first implement core datapath units, then add control logic and hazard handling, and finally run end-to-end programs to validate correctness. To control difficulty, optional extensions are provided as stretch goals, but the assessment focuses on a correct baseline pipeline implementation.

In decode, instructions are parsed to generate operands, immediates, and control signals. The design emphasizes clarity of control generation so that students can trace how each instruction type drives downstream stages. Students learn to distinguish combinational decoding from registered control, and to reason about how control must be delayed or aligned when it travels through pipeline registers.

In execute, the ALU performs arithmetic/logic operations and computes branch decisions and effective addresses. The course highlights how execution results and control outcomes must be aligned with pipeline timing. Branch resolution illustrates trade-offs between simplicity and performance, and provides a natural setting to explain how mispredictions or late decisions impact the instruction stream in earlier stages.

To make pipeline behavior observable, the course emphasizes hazard handling in the execute stage. Students implement basic forwarding paths and a stall/bubble mechanism for typical data conflicts, then validate that control and data signals remain aligned as instructions advance through the registers. Load-use cases are treated as representative corner scenarios, so students can relate incorrect results to a specific stage interaction.

In memory, load/store accesses are handled and remaining instructions pass their results forward. Students learn to manage data alignment and to ensure that write-back

receives the correct value under different instruction types. The write-back path is treated explicitly: students verify that destination register selection, write-enable control, and data selection match the instruction semantics across all supported operations.

Auxiliary modules such as the register file, instruction/data memory, and basic I/O are provided or implemented in simplified form. This keeps attention on the pipeline while still enabling complete program execution and observable outputs. Typical teaching peripherals include a simple serial interface for printing results, LED/switch I/O for quick checks, and a minimal memory-mapped interface so that software-visible behavior can be demonstrated in class.

For FPGA resource evaluation, the Cyclone-V SoC platform used in the course provides roughly 85k logic units, 128k registers, and 496KB of block RAM. After estimating the needs of key modules (ALU, register file, and instruction/data memories), the board is sufficient for a baseline RV32I five-stage pipeline; on-chip block RAM is used for instruction and data memory to reduce logic occupancy.

Debugging and verification follow a hierarchical strategy. Students first verify individual modules with targeted simulation tests, then validate stage-level behavior by inspecting pipeline registers and key control signals, and finally run integrated instruction sequences on the board to check hazards and corner cases. Representative programs include arithmetic sequences, load-store loops, and branch-heavy cases designed to trigger hazards and control transfers; students compare expected and observed outcomes to locate faults efficiently.

4 Analysis of the Effects of Teaching Reform

Effect evaluation combines quantitative and qualitative evidence. Quantitatively, we use students' completion of the processor project, lab performance, and assessment results to reflect engineering implementation ability and conceptual mastery across the course outcomes. Where possible, results are compared with prior cohorts using the traditional arrangement to provide a coarse reference for improvement, while recognizing differences in student background and cohort size. We also review typical error patterns in project checkpoints, such as pipeline hazard handling and interface timing, to identify where students most often struggle and to refine the sequencing of lectures and labs. This feedback loop helps the reform remain practical: evaluation is used not only to report outcomes, but to adjust scaffolding, documentation, and debugging guidance for subsequent cohorts.

Quantitatively, implementation capability is tracked through design scores, stage acceptance rates, and comprehensive test pass rates. Across cohorts, the excellence rate increased from 25% to 60%, and the system capability compliance rate increased from 40% to 88%. Questionnaire feedback also reports improved understanding of pipeline conflicts and debugging.

Qualitatively, project defenses and reports indicate three recurring changes. Students explain errors from a stage perspective (e.g., IF/ID/EX) and can connect symptoms to concrete pipeline interactions. They use dependency relationships to justify forwarding

versus stalls (especially for load-use cases), and some attempt small extensions with test evidence for correctness.

In classroom observation, students increasingly trace signals across stages and show stronger confidence when integrating modules and validating behavior, while prior HDL experience remains a key source of variation.

5 Conclusions

Focusing on the cultivation needs of the computer system course's system capability, this article explores and implements a practical teaching reform scheme based on the RISC-V architecture. By adopting a five-stage pipeline processor design as the main practice, the course integrates capabilities in system cognition, design, development, and application throughout the entire teaching process, effectively facilitating students' transition from component-level understanding to system-level cognition.

Teaching practice has shown that, while ensuring the correctness of the system architecture, introducing moderate engineering implementation details and applying pedagogical treatments can enhance the authenticity and operability of practical teaching. This teaching model is highly replicable for universities equipped with FPGA experimentation conditions and can serve as a reference for teaching reform in computer system courses in the context of new engineering education.

References

1. Zhang, J., Feng, X.: Research and practice of cultivating complex engineering problem-solving ability in software engineering majors. Software Guide 24(11), 229–236 (2025). https://doi.org/10.11907/rjdk.241774.
2. Wang, T.J., Wang, T., Zhao, Y.H.: Exploration and practice of cultivating system capabilities in the cloud computing introduction course. Education and Teaching Forum 41, 113–116 (2025). https://read.cnki.net/web/Journal/Article/JYJU202541029.html.
3. Kumar, S., Ray, K. C.: Micro-Architecture of LW Driven Bubble-Free Five-Stage Pipelined RISC-V Processor Core for Energy Constraint Low-End Application. IEEE Transactions on Circuits and Systems 73(1), 207-215 (2026). https://doi.org/10.1109/TCSI.2025.3629838
4. Liu, C., Wu, Y.J., Wu, J.Z.: Survey on RISC-V system architecture research. Journal of Software 32(12), 3992–4024 (2021). https://doi.org/10.13328/j.cnki.jos.006490.
5. Singh, A., Franklin, N., Gaur, N., et al.: Design and implementation of a 32-bit ISA RISC-V processor core using Virtex-7 and Virtex-UltraScale. In: Proceedings of the 5th International Conference on Computing Communication and Automation, pp. 126–130 (2020). https://doi.org/10.1109/ICCCA49541.2020.9250850.
6. Saussereau, J., Leroux, C., Begueret, J.B., et al.: AsteRISC: A size-optimized RISC-V core for design space exploration. In: IEEE International Symposium on Circuits and Systems, pp. 1–5 (2023). https://doi.org/10.1109/ISCAS46773.2023.10181330.