MIPS Pipeline CPU Simulator

Student: Boca Ioan-Doru

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

Introduction

1.1 Context

The **C**entral **P**rocessing **U**nit (CPU) is the foundational component of any computer system, and understanding its internal workings is crucial in Computer Science. The **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages (MIPS) architecture is a classic academic model used to teach core concepts of moder processor design.

The aim of this project is to develop a software simulator for a MIPS pipelined CPU. This program will provide a virtual, interactive environment to execute a variety of MIPS instruction, allowing for the visualization and analysis of the processor's behavior. Such a tool is invaluable for studying the principles of computer architecture, data paths and performance gains achived through pipelined execution.

1.2 Objectives

The simulation will be designed in Unity using C# language containg the internal schema for a MIPS Pipeline architecture and a GUI for programming and testing custom MIPS instruction and code.

The GUI will contain multiple menues with different purpose:

- Instruction Memory Interface
- Data Memory Interface
- Register File Interface

Bibliographic Research

2.1 The Role of CPU Simulation

The **Central Processing Unit** (CPU) is the core component responsible for executing program instructions in a computer. Its design is a complex interplay of logic, data paths, and control signals. To study these designs, **CPU simulation** is an indispensable tool. A simulator is a software program that models the behavior of a hardware component, providing a virtual environment to observ its internal state step-by-step without the cost and complexity of building physical hardware.

2.2 The MIPS Architecture

The architecture to be simulated is **MIPS** (Microprocessor without Interlocked Pipeline Stages). MIPS is a fundational example of a **RISC** (**R**educed **I**nstruction **S**et **C**omputer) architecture [1]. The RISC architecture prioritizes simplicity by using a small, highly optimized set of instructions.

Key characteristics of MIPS:

- 32 bit instruction length.
- "load-store" architecture, only **load** and **store** instructions access memory, the rest of instructions uses registers.
- small number of instruction types: **R**-type, **I**-type, **J**-type.

2.3 Pipeline Architecture

Instead of waiting for one instruction to complete all its steps before starting the next, the pipeline allows multiple instructions to be in different stages of completion at the same time. The classic MIPS architecture uses a five-stage pipeline[2], which will be the model for this simulator:

MIPS Pipeline Stages:

- **IF** (**I**nstruction **F**etch): Fetches the next instruction from memory using the address in the **P**rogram **C**ounter (PC).
- **ID** (**I**nstruction **D**ecode): Decodes the instruction, identifies the operation, and reads the required values from the register file.
- **EX** (**Ex**ecute): The Arithmetic Logic Unit (ALU) performs the required calculation.
- **MEM** (**Mem**ory Access): The instruction reads from or writes to data memory. This stage is only active for load and store instructions.
- **WB** (**W**rite **B**ack): The final result is written back into the register file.

2.4 Pipeline Hazard

While pipelining improves speed, it introduces potential problems known as **hazards**, which occur when the next instruction cannot execute correctly in the following clock cycle [2]. A functional simulator must be able to detect and manage these hazards. They fall into three categories.

2.4.1 Data Hazards

Data hazards occures when an instruction needs a result from the previous instruction that has not yet been written back to the register file.

```
add $r3, $r1, $r2

sub $r4, $r3, $r2
```

The result from **add** instruction will not been written to **R3** before is required in the next instruction resulting in the usage of the previous value of **R3** in the **sub** instruction. This hazard will produce incorrect results. The primary solution is implementing a forwording unit that will get the result from **EX stage** before the **WB stage**.

2.4.2 Control Hazards

Control hazards are caused by branch and jump instructions. The processor does not know the outcome of a branch (i.e., whether to take it or not) until the **EX stage**. However, it must decide which instruction to fetch in the **IF stage**, which is two cycles earlier.

A common solution, and the one to be simulated, is to **assume the branch is not taken** and fetch the next sequential instruction. If the branch is later determined to be taken, the incorrectly fetched instruction must be **flushed** from the pipeline, and the PC is updated with the correct branch target address.[2]

2.4.3 Structural Hazards

A structural hazard occurs if two instructions attempt to use the same hardware resource at the same time. The simple MIPS architecture is designed to minimize these. For instance, by having separate instruction and data memorie, an instruction fetch (IF stage) can occur at the same time as a data access (MEM stage) without conflict.

Project Analysis

This section provides a deeper analysis of the simulator's implementation, combining the project's academic goals with the practical C# and Unity code provided.

3.1 Use Case Scenario

A computer architecture student wants to understand **data hazards** and **forwarding**. Using the simulator's GUI , they load the following MIPS code into the Instruction Memory:

add $r3, $r1, $r2

sub $r4, $r3, $r2

As the student steps through the simulation, they watch the first instruction (add) move through the five-stage pipeline (IF, ID, EX, MEM, WB). When the sub instruction reaches the ID stage, it needs the value of $r3. However, the add instruction is only in the EX stage and has not yet written its result back to the register file.

The student can *visually* observe this data hazard. They would then see the forwarding unit activate, taking the result directly from the add instruction's EX/MEM pipeline register and feeding it into the sub instruction's EX stage, bypassing the WB stage and preventing a stall. This provides a clear, interactive visualization of a core concept in pipelined architectures

3.2 Basic Functionalities

The simulator is designed to model the classic five-stage MIPS pipeline:

- **Instruction Fetch (IF):** Reads from the instruction memory (modeled by ROM.cs) using the Program Counter.
- **Instruction Decode (ID):** Decodes the instruction, reads from the RegisterFile.cs, and generates control signals via MainControl.cs.
- **Execute (EX):** The ALU (represented by components like Adder.cs and ShiftUnit.cs) performs calculations.
- **Memory Access (MEM):** Active for LW and SW instructions, reading from or writing to data memory.

- **Write Back (WB):** Writes the final result back into the RegisterFile.cs.

The project also accounts for key pipeline challenges, including data hazards (solved with forwarding) and control hazards from branches (solved by assuming "not taken" and flushing the pipeline if wrong).

3.3 Supported Instruction Set

The MainControl.cs script clearly defines the supported subset of the MIPS instruction set by its opCode. The control unit generates the necessary control signals for the following instructions:

- **R-type** (e.g., add, sub): opCode 0x00
- **ADDI** (Add Immediate): opCode 0x08
- **LW** (Load Word): opCode 0x23
- **SW** (Store Word): opCode 0x2B
- **BEQ** (Branch on Equal): opCode 0x04
- **BNE** (Branch on Not Equal): opCode 0x05
- **BGEZ** (Branch on Greater than or Equal to Zero): opCode 0x01
- **BGTZ** (Branch on Greater than Zero): opCode 0x07
- **J** (Jump): opCode 0x02
- **JAL** (Jump and Link): opCode 0x03
- **JR** (Jump Register): opCode 0x1A

3.4 Core Implementation: Clock and Data Flow

The true innovation of this project lies in its simulation engine, which is masterfully adapted to the Unity environment.

3.4.1 The Custom Clock (Clock.cs)

Instead of relying on Unity's FixedUpdate() physics loop, the project is driven by custom *Clock.cs* component. This class runs in Unity's *Update()* loop and simulates a clock signal by toggling a *Signal* value every *numberOfUpdates* frames.

Crucially, this component provides boolean properties for *RisingEdge* and *FallingEdge*. This is the most important part of the design, as it allows the simulator to precisely model the edge-triggered behavior of real hardware, resolving the synchronization problems and race conditions found in a simple *FixedUpdate* model.

3.4.2 Asynchronous vs. Synchronous Components

With this new clock, the simulator's components are divided into two categories, perfectly modeling real hardware:

1. **Asynchronous (Combinational) Logic:** These are components that "instantly" calculate new values based on their inputs. Their logic is placed in Unity's *Update()* method to ensure they are always calculating and "settling" the values for the next clock tick. Those components are:
    - **ALU.cs**
    - **AluControl.cs**
    - **Mux.cs**
    - **SignalSplitter.cs**
    - **SignalAdder.cs**
    - **OperationUnit.cs**(AND, OR, NOT Gates, ADD unit, SHIFT unit)
2. **Synchronous (Sequential) Logic:** These are components that must be synchronized to the clock. They represent memory elements (registers and RAM) that only change state on a clock edge. Their logic is placed inside an *if (clock.RisingEdge)* block. Those components are:
    - **RAM.cs**
    - **Register.cs**
    - **RegisterFile.cs**

3.4.3 The Signal and BitArray Abstraction

The most critical components for data flow are the Signal.cs and BitArray.cs classes.

- **Signal as a "Wire":** The Signal class is a ScriptableObject. This is a clever Unity pattern. Instead of components having direct C# references to each other (e.g., adder.output = alu.input), they all reference a shared Signal asset. This Signal asset *is* the wire. A developer can visually "wire" the CPU in the Unity editor by dragging the same Signal asset into the output slot of one component and the input slot of another.
- **BitArray as the Data Packet:** This project uses a **custom BitArray class**, not the standard System.Collections.BitArray. This custom class is the foundation of the simulator's data handling. Its key features are:
    1. **Underlying Data:** It stores data as a simple bool[] _bits array.

2. **Explicit Type Casting:** The class's power comes from its **explicit conversion operators**. It allows code to seamlessly cast between the BitArray and standard C# numeric types.

This enables highly readable and realistic code in the hardware components. For example:

- The **Adder** can perform a natural C# **sum**:

  s.Value = (BitArray)((uint)a.Value + (uint)b.Value);

- The **ExtensionUnit** can interpret the same 16-bit data as either signed or unsigned:

  Extend((short)input.Value); or Extend((ushort)input.Value);

- The **Mux** can use a signal's value directly as an **array index**:

  uint sel = (uint)selection.Value; output.Value = inputs[sel].Value;

- The **ShiftUnit** uses a custom-defined **<< operator** for the BitArray class:

  output.Value = input.Value << 2;

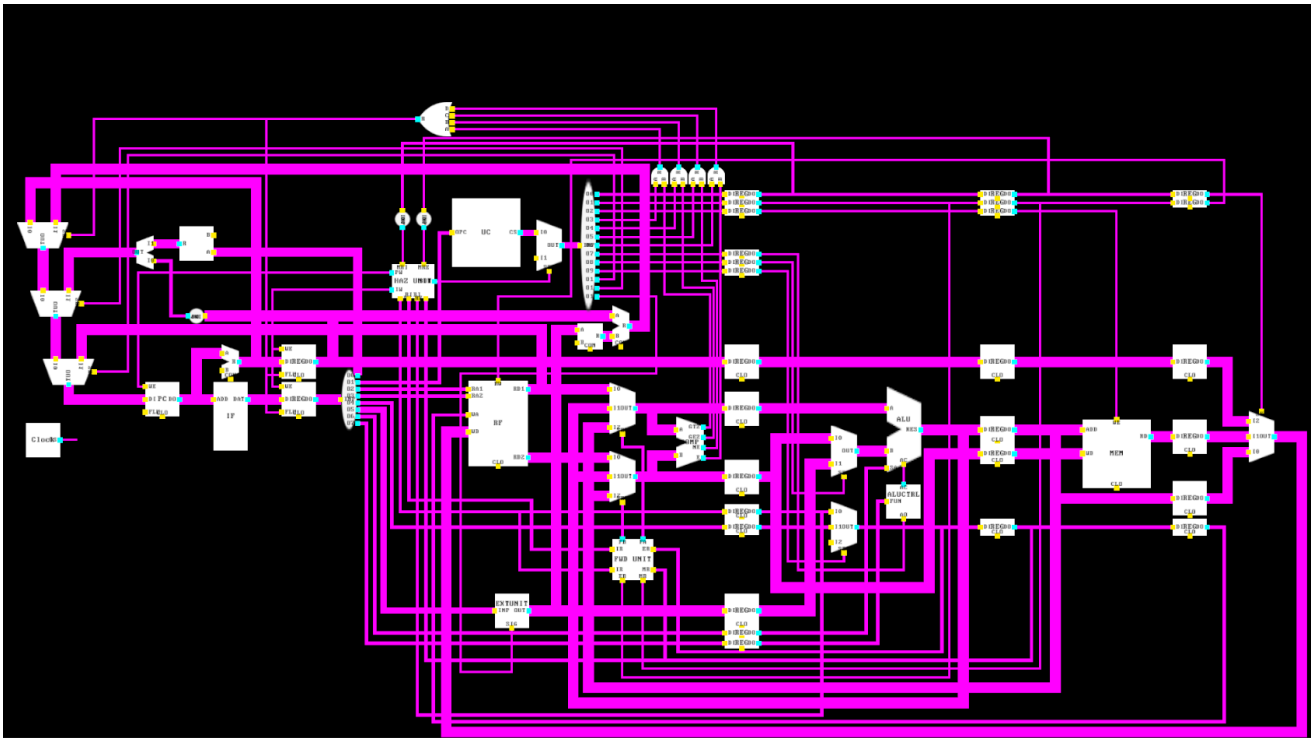
This Signal/BitArray combination provides a powerful, flexible, and type-safe abstraction that perfectly models how electrical signals (as bundles of bits) are transported on wires and re-interpreted by different hardware components.

Another critical component in this abstraction is the **SignalSplitter.cs** script. This class models the physical "wiring" that splits a main signal bus into smaller, dedicated buses. Its Split method uses bitwise shifting and masking to extract a specific range of bits from the input BitArray. In the simulator, this is used for example in the **Instruction Decode (ID)** stage. It takes the 32-bit instruction as a single signal bus and splits it into its functional sub-signals: the 6-bit **opCode** is routed to the **MainControl.cs**, the 5-bit register addresses (**rs**, **rt**, **rd**) are routed to the **RegisterFile.cs**, and the 16-bit **immediate** value is routed to the **ExtensionUnit.cs**. This is a direct simulation of how an instruction bus is physically broken apart and routed to different datapath components.

Design

Based on the functional requirements and the analyzed components, the simulator is designed as a classic 5-stage MIPS pipeline. The communication between components is managed via Signal objects, which act as data buses. The entire pipeline is clocked by Unity's FixedUpdate loop, with pipeline registers latching data between stages on each tick.

A high-level block diagram of the MIPS datapath shows how these components are interconnected.



The principal macro-components identified from the C# scripts are as follows in the next subsections.

4.1 Instruction Fetch (IF)

This stage is responsible for fetching the current instruction from memory.

- **Program Counter (PC):** An instance of Register.cs that holds the address of the *next* instruction.
- **Instruction Memory:** A ROM.cs component that stores the user's program. It takes the address from the PC and outputs the 32-bit instruction.

- **PC Adder:** An Adder.cs component that calculates PC + 4 to point to the next sequential instruction.

4.2 Instruction Decode (ID)

This stage decodes the fetched instruction and reads the required operands from the register file.

- **Instruction Splitter:** An instance of SignalSplitter.cs. It acts as a literal bus splitter, receiving the 32-bit instruction from the IF/ID register and splitting it into multiple output signals (opCode[31:26], rs[25:21], rt[20:16], rd[15:11], immediate[15:0], shift ammount[10:6], function[5:0]).
- **Main Control:** The MainControl.cs component. It receives the 6-bit opCode signal from the splitter and then decodes it to generate the primary control signals (e.g., RegWrite, ALUSrc, MemRead, MemWrite, Branch, Jump, etc.) for all other stages.
- **Register File:** The RegisterFile.cs component. It receives the readAddress1 (rs) and readAddress2 (rt) signals from the splitter. It outputs the data from these two registers (readData1, readData2). It also has a writeAddress port for the WB stage.
- **Sign Extension Unit:** The ExtensionUnit.cs component. It receives the 16-bit immediate value from the splitter and extends it to 32 bits, either with sign-extension (for addi, beq, lw, sw) or zero-extension, based on the signed control signal.
- **Branch Logic:** This stage also finalizes branches. As seen in the MIPS diagram, a series of logic gates(simulated by *OperationUnit.cs* components) use the *Branch* control signals and the *flags* from the *ALU* to decide if a branch should be taken. This result is used to control the *MUX* for the Program Counter in the IF stage.

4.3 Execute (EX)

This stage is the computational core of the pipeline and runs asynchronously in *Update()*. It receives values (ReadData1, ReadData2, extended immediate, etc.) and control signals from the ID/EX pipeline Registers.

- **ALU Input Mux:** An instance of *Mux.cs* selects the second operand for the *ALU*. Guided by the *ALUSrc* control signal from *MainControl.cs*, is chooses

between *readData2* from *RegisterFile.cs* and the 32-bit *Extended Immediate* from *ExtensionUnit.cs*.

- **AluControl:** The *AluControl.cs* component runs in *Update()*. It takes the *ALUOp* from *MainControl* and the instruction's *Func* field from from *SignalSplitter* and generates the final 3-bit *ALUCtrl* signal. This signal tells the *Alu* precisely which operation to perform.
- **ALU:** The *ALU.cs* component is the main workhorse. It runs in *Update()* taking the final *ALUCtrl* singal and two data inputs, and performs the specified operation. It outputs the 32-bit result and 2-bit *flags* signal for *Zero* and *Greater Than Zero* used by branch logic.

4.4 Memory Access (MEM)

This stage handles all communication whit data memory and resolves branch instructions.

- **Data Memory (RAM):** The *RAM.cs* component is the primary component here. It has a dual-mode design:
    - **Read (Asynchronous):** The *Read()* method runs continously in *Update()*. It takes the addres from the *ALU result* and constantly puts the corresponding data onto the *readData* signal
    - **Write (Synchronous):** The *Update()* method checks for the *clock.RisingEdge*. Only on rising edge, and only if the *writeEnable* signal is true, will it write the *writeData* to memory.

4.5 Write Back (WB)

This is the final stage, where the result of the operation is written back into the register file.

- **Write Back Mux:** A key *Mux.cs* component selects the final data to be written. Guided by the *MemtoReg* control signal, it chooses between:
    - **Input 00:** The ALU result.
    - **Input 01:** The Memory data.
    - **Input 10:** The PC+4.
- **Register File (Write):** The *RegisterFile.cs* component recives the data from the Mux. Its *Update()* method checks for *clock.FallingEdge*. On a falling edge (to prevent conflicts with reads), and only if the *registerWrite* signal is true, it writes the data into the register specified by *writeAddress*.

4.6 Pipeline Registers

To separate the five stages, four pipeline registers are used, all of which are instances of the *Register.cs* class:

- **IF/ID** Registers
- **ID/EX** Registers
- **EX/MEM** Registers
- **MEM/WB** Registers

Each register holds all data and control signals being passed from its stage to the next, synchronized by the *Clock.cs* clock.

4.7 Hazard Management & Architecture Design

This project implements a sophisticated **Branch-in-ID** pipeline architecture, an optimization over the standard textbook design[2] Cristian Vancea. *Arhitectura calculatoarelor Curs 8*.

4.7.1 Architectural Choice: Branch in ID

In a standard MIPS pipeline, the branch decision (Compare & Branch) often happens in the **MEM** stage. This results in a "Control Hazard" penalty of **3 clock cycles** (flushing IF, ID, and EX) whenever a branch is taken.

- To optimize performance, this simulator moves the branch logic to the **ID (Instruction Decode)** stage (referencing architecture from[2] Cristian Vancea. *Arhitectura calculatoarelor Curs 8*.**Comparison**: A Comparator unit checks equality (Rs == Rt) inside the ID stage.
- **Target Calculation**: An Adder inside the ID stage computes the Branch Target Address.
- **Benefit**: The flush penalty is reduced to **1 clock cycle** (flushing only the IF stage), significantly improving throughput for branch-heavy programs.

4.7.2 Challenges: Forwarding to ID

Moving the branch decision to ID introduces complex **Data Hazards**. The instruction in the ID stage might need data that is currently being calculated in the EX stage or loaded in the MEM stage.

- *Challenge*: The standard Forwarding Unit only sends data to the EX stage (for the ALU).
- *Solution*: A dedicated **Forwarding Unit (ID)** was implemented. It monitors the ID/EX (previous instruction) and EX/MEM (2nd previous instruction) pipeline registers. If it detects a write to the registers needed for the branch comparison, it forwards the data **backwards** to the ID stage comparator.

4.7.3 Challenges: Load-Use Stalls

A critical edge case arises when a Branch depends on a LW (Load Word) instruction immediately preceding it.

- *Scenario*: LW $t1, 0(...) followed by BEQ $t1, ...
- *Problem*: The data for $t1 is in memory. Even with forwarding, it cannot reach the ID stage in time.
- *Solution*: The **Hazard Unit** implements a stricter **2-cycle stall**. It detects if a Load instruction is in the EX or MEM stage targeting the branch operands. It freezes the PC and IF/ID registers until the data is available.

Component Class Descriptions

This section details the role of each C# script used in the simulation.

5.1 Core Simulation Components

- **Signal.cs**: A *ScriptableObject* that acts as a "wire" or "bus." It holds a *BitArray* value. Components read from and write to Signal assets to communicate, allowing for visual "wiring" in the Unity editor.
- **BitArray.cs**: A custom class that stores data as a bool[] array. It provides powerful explicit conversion operators to and from numeric types (like uint, int, byte, etc.), making hardware-level bit manipulation simple and readable (e.g., (uint)mySignal.Value).
- **Clock.cs**: The simulation's "heartbeat". It runs in *Update()* and toggles a Signal value at a fixed rate, simulating a clock. It provides *RisingEdge* and *FallingEdge* booleans, which are used by synchronous components to latch data.

5.2 Synchronous (Clock-Driven) Components

- **Register.cs**: Simulates a single pipeline register. It has a single data input and output. On the *clock.RisingEdge*, it writes its latched value to its output, and then in *Update()* it continuously reads its input to latch the value for the *next* clock tick.
- **RegisterFile.cs**: Simulates the 32 MIPS registers. It performs asynchronous reads in *Update()* (always outputting the data for two read addresses) and a synchronous write on the *clock.FallingEdge* only if *registerWrite* is true.
- **RAM.cs**: Simulates the Data Memory. Similar to the RegisterFile, it performs asynchronous reads in *Update()* and synchronous writes on the *clock.RisingEdge* only if *writeEnable* is true.

5.3 Asynchronous (Combinational) Components

- **ROM.cs**: Simulates the Instruction Memory. It is read-only and runs asynchronously in *Update()*, continuously outputting the 32-bit instruction at the address specified by the Program Counter.
- **MainControl.cs**: The primary decoder. Runs in *Update()*. It reads the 6-bit *opCode* from the instruction and generates the main 16-bit *controlSignal* that will be pipelined through the entire CPU.
- **AluControl.cs**: The secondary decoder for the *ALU*. Runs in *Update()*. It reads the *func* field of the instruction and an *ALUOp* signal from *MainControl* to generate the final 3-bit *ALUCtrl* signal for the *ALU*.
- **ALU.cs**: The *Arithmetic Logic Unit*. Runs in *Update()*. It takes the *ALUCtrl* signal and performs the specified operation (ADD, SUB, AND, etc.) on its inputs, outputting a result and flags.
- **Mux.cs**: Simulates a *multiplexer*. Runs in *Update()*. It reads a selection signal and passes the data from one of its inputs to its output.
- **ExtensionUnit.cs**: Simulates the *Sign/Zero Extender*. Runs in *Update()*. It reads a 16-bit immediate value and a 1-bit signed signal, outputting a 32-bit extended value.
- **Comparator.cs**: A specialized combinational logic unit used for branch decisions. Inputs: Two 32-bit values (A and B). Logic: Performs a subtraction (A - B) to determine relationships. Outputs: Generates boolean flags for *Equal*, *NotEqual*, *Greater*, and *GreaterOrEqual*. These flags are used by the Control
- **ForwardingUnit.cs**: This unit manages Data Hazards by bypassing the Register File. Located in the ID stage to support the "Branch-in-ID" architecture. It compares the source registers of the current instruction against the destination registers of the previous two instructions. It strictly prioritizes the **EX Stage** result (most recent) over the **MEM Stage** result.

  - If (Write_EX && Dest_EX == Src) -> Forward form EX
  - Else If (Write_MEM && Dest_MEM == Src) -> Forward from MEM

  Controls the multiplexers that feed the Comparator.

- **HazardUnit.cs:** This unit is the traffic cop of the processor. It resides in the ID stage and detects conditions requiring pipeline intervention. It monitors the MemRead signals from the EX and MEM stages. If a Load instruction is detected that conflicts with the current instruction's source registers (rs, rt), it triggers a stall.
  - pcWrite: Disables PC updates (freezes the fetch).
  - ifIdWrite: Disables IF/ID writes (freezes the current instruction).
  - bubble: Injects a NOP (all zeros) into the ID/EX register to clear the pipeline gap.

5.4 Utility Components

- **OperationUnit.cs**: A *generic, reusable calculation component*. Runs in *Update()*. It can be configured via an enum to perform one of many operations (ADD, AND, SHIFT, etc.) on its inputs.
- **SignalAdder.cs**: A *bus-concatenation utility*. Runs in *Update()*. It takes an array of inputs signals and combines them into a single, larger output signal.
- **SignalSplitter.cs**: A *bus-splitting utility*. Runs in *Update()*. It takes a single input signal and splits it into multiple output signals based on Interval definitions (e.g., splitting an instruction into opcode, rs, rt, etc.).

5.5 Visualization & Editor Components

- **ComponentFactory.cs**: An Editor script that provides a menu for instantiating hardware components with pre-configured ports and layout.
- **CircuitNode.cs**: Represents an input/output pin on a component. It manages its position relative to the component body and serves as an anchor for wires.
- **WireVisual.cs**: Renders the physical wire connecting two ports. It reads the Signal value and updates the wire's texture color to visually represent the binary data flow (Green/Black).

5.6 GUI Components

The simulation now includes a robust set of UI components to visualize and interact with the processor's state in real-time.

- **InstructionWindow.cs & InstructionRow.cs**: Manages the visualization of the Instruction Memory (ROM). It allows users to view code in Assembly or Binary, edit instructions on the fly, and visualize which instruction is currently in which pipeline stage (IF, ID, EX, MEM, WB) via dynamic tags. It also handles file I/O for saving and loading programs.
- **DataWindow.cs & DataRow.cs**: Provides an interface for the Data Memory (RAM). It displays memory contents in Hexadecimal and allows direct editing. It visually highlights memory access, turning rows green for reads and blue for writes based on active control signals.

- **RegisterFileWindow.cs & RegisterFileRow.cs**: Visualizes the 32 general-purpose registers. It shows register names (e.g., $t0, $sp) and their current values. It uses color-coding to show active ports: Green/Cyan for Read Address 1/2 and Blue for Write Address when the RegWrite signal is active.
- **InstructionHelper.cs**: A static utility class acting as the project's Assembler and Disassembler. It contains logic to convert 32-bit uint binary instructions into MIPS Assembly strings (Disassemble) and parse Assembly strings back into binary (Assemble), handling all R-Type, I-Type, and J-Type formats.
- **ResizableWindow.cs**: A UI utility that allows the various interface panels to be collapsed, expanded, and resized by the user, improving workspace management within the Unity Game View.

Circuit Editor & Visualization

To enhance usability and provide visual feedback, a custom editor system has been implemented. This allows users to instantiate components, create input/output ports, and connect them with visual wires directly within the Unity Editor, mimicking a digital logic circuit designer.

6.1 Component Factory

The ComponentFactory is a custom Unity Editor Window that streamlines the creation of simulation components. Instead of manually creating GameObjects and adding scripts, users can use a menu to instantiate fully configured prefabs.

- **Automated Setup:** Methods like CreateALU, CreateRegister, etc., automatically create the GameObject, attach the specific simulation script (ALU.cs, Register.cs), and add the necessary UI elements (Images, RectTransforms).
- **Port Generation:** The factory automatically generates input and output ports (CircuitNode) for each component at standard positions (e.g., "OpCode" on the left, "Control Signal" on the right for MainControl).
- **Signal Initialization:** When creating an output port, the factory automatically creates a new Signal asset and a corresponding WireVisual to visualize it, ensuring every component is ready to be connected immediately.

6.2 Circuit Ports

The CircuitNode component represents a connection point (pin) on a hardware component.

- **Anchoring:** It handles the logic for positioning the port relative to its parent component. The normalizedPortPosition allows ports to be placed consistently (e.g., (-1, 0) for left-center) regardless of the component's size.
- **World Position:** It provides a helper method GetPortWorldPosition() to assist the wire system in drawing lines exactly to the center of the port.

6.3 Visual Wiring

The WireVisual script brings the data flow to life. It renders a connection between two CircuitNode points and visualizes the Signal's data in real-time.

- **Dynamic Rendering:** It uses a Unity LineRenderer to draw the wire. The script automatically updates the line's start and end positions to follow the CircuitNode objects if the components are moved in the editor.
- **Data Visualization:** As discussed in the design phase, this component reads the Signal's BitArray value. It generates a 1D texture where each pixel represents a bit (Green for 1, Black for 0) and applies this texture to the line. This allows users to see the exact binary value traveling through the wire at a glance (e.g., a 1010 pattern appears as Green-Black-Green-Black).
- **Editor Integration:** It includes an [ExecuteInEditMode] attribute and UpdateLinePosition logic, allowing wires to be manipulated and visualized even when the simulation is not running.

Graphical User Interface (GUI)

To transition from a "black box" simulation to a fully interactive educational tool, a comprehensive Graphical User Interface has been implemented. This GUI allows users to inspect the internal state of the processor, modify programs, and visualize the data flow across the pipeline.

7.1 Instruction Memory Interface

The Instruction Window provides a direct view into the ROM component.

- **Dual-Mode Editing:** Users can toggle between **Binary** mode (viewing raw 32-bit machine code) and **Assembly** mode. The InstructionHelper class automatically translates between the two, allowing users to write "ADD $t0, $t1, $t2" and immediately see the corresponding binary generated.
- **Pipeline Stage Tracking:** The interface monitors the Program Counter inputs for every pipeline register. It dynamically assigns tags (IF, ID, EX, MEM, WB) to the specific instructions currently residing in those stages, effectively visualizing the pipeline flow.
- **Persistence:** Users can save their written programs to text files and load them back later, facilitating the creation of a library of test cases.

7.2 Data Memory Interface

The Data Window allows for inspection and manipulation of the system RAM.

- **Hexadecimal View:** Memory is displayed in 32-bit words formatted as Hexadecimal strings (e.g., 0000001A), which is the standard for low-level memory inspection.
- **Visual Debugging:** The window subscribes to the RAM's control signals (address, write, writeData). When a memory operation occurs, the specific row being accessed lights up (Green for Read, Blue for Write), providing immediate visual confirmation of load/store instructions.

7.3 Register File Interface

The Register File Window provides a real-time dashboard for the CPU's 32 general-purpose registers.

- **Live Monitoring:** Displays the current value of every register, updated every clock cycle.

  Port Visualization: To help students understand the hardware structure, the UI highlights which registers are currently being accessed. It connects to the ra1, ra2, and wa signals of the RegisterFile component. If an instruction is reading $t0 and $t1, those specific rows will highlight. If a result is being written back to $s0, that row will highlight in a different color.[2] Cristian Vancea. *Arhitectura calculatoarelor Curs 8*.

User Manual: Running the Simulation

This section provides a guide on how to interact with the running simulation to execute MIPS programs and visualize the pipeline.

8.1 Control Interface

The simulation is controlled via a main toolbar located at the top of the Game View:

- **Start/Pause**: Toggles the execution of the processor. When paused, the clock stops, allowing for static analysis of the current state.
- **Reload**: Resets the Program Counter (PC) to 0 and clears all pipeline registers, effectively restarting the current program from the beginning without clearing memory.
- **Clock Slider**: Adjusts the simulation speed. Moving the slider changes the frequency of the Clock signal, allowing users to slow down execution to watch individual data movements or speed it up to finish programs quickly.

8.2 Visualization Windows

The interface consists of three primary interactive windows:

1. **Instruction Memory Window**:
    - **View**: Shows the program code. The currently executing instructions are tagged with their pipeline stage (IF, ID, EX, MEM, WB).
    - **Edit**: Users can write assembly code directly (e.g., ADD $t0, $t1, $t2). The simulator automatically assembles this into binary.
    - **Restart**: A dedicated button clears the instruction memory.
2. **Data Memory Window**:
    - **View**: Displays the contents of the RAM in Hexadecimal.
    - **Interaction**: Rows highlight **Green** when read from and **Blue** when written to.
    - **Restart**: Clears all data memory to zero.
3. **Register File Window**:

- **View**: Shows the real-time values of all 32 General Purpose Registers.
- **Visualization**: Highlights active ports—**Green/Cyan** for registers being read (Rs, Rt) and **Blue** for the register being written (Rd/Rt) during the Write-Back stage.

Developer Manual: Editing the Architecture

The simulator is built to be modular, allowing users to modify the datapath or create new architectures directly within the Unity Editor.

9.1 Adding Components

New hardware components can be added to the scene via the custom context menu:

- Navigate to **GameObject** > **Components**.
- Select the desired component (e.g., ALU, Register, Mux, Hazard Unit).
- The component will appear in the scene with pre-configured ports.

9.2 Wiring Components

Data flow is defined by connecting ports using visual wires:

1. **Ports**: Every component has input (Yellow) and output (Cyan) nodes.
2. **Wiring**: To create a connection, the simulator uses a Signal asset system. In the Wire Inspector, drag the Input Port of one component and the Output Port of another in the specifed fields.
3. **Visual Feedback**: A visual wire will automatically render between the connected nodes. This wire changes color (Green/Black) in real-time to show the binary data flowing through it.

9.3 Customizing Components

- **Signal Length**: The bit-width of buses (e.g., 32-bit) is determined by the Signal assets.
- **Adding Ports**: For dynamic components like **Multiplexers** or **Signal Splitters/Adders**, you can add extra ports via the menu:
    - Select the component in the hierarchy.

    - Go to **GameObject > Components > Add Input Port** (or **Add Output Port**).
- **Manual Adjustment**: The visual placement of ports and wires is rudimentary. Users may need to manually adjust the RectTransform or CircuitNode positions in the Inspector to ensure wires are neat and do not overlap. Also the wire nodes positions need to be added manual by the user.

9.4 Limitations

- **Logic Coding**: While wiring is visual, the internal logic of a component (e.g., how an ALU calculates a result) must be written in C#. New component types require creating a new script inheriting from MonoBehaviour.
- **Layout**: There is no auto-layout engine. Moving a component requires manually moving its connected wires or nodes if they desync visually.

Bibliography

[1] Patterson, D. A., & Hennessy, J. L. (2013). *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann.

[2] Cristian Vancea. *Arhitectura calculatoarelor Curs 8*.