JoSDC'24

المسابقة الوطنية لتصميم الشرائح الإلكترونية

Jordan National Semiconductors Design Competition

# Development Phase Report - JoSDC'24

| Team Name | | |
|---|---|---|
| SB2P | | |
| **Team Members** | | |
| # | Name | Email |
| **1** | Abdelrahman Samha | AbdelrahmanSamha@Outlook.com |
| **2** | Ahmad Alhaj | alhajahmad670@gmail.com |
| **3** | Nermeen Nedal | nermeennedal11@gmail.com |
| **4** | Ehab Sbieh | ehabsbaih9@gmail.com |



# Jordan National Semiconductor Design Competition (JOSDC'2024)

## MIPS PipelineX

By


| Abdelrahman Samha | Ahmad Alhaj | Nermeen Nedal | Ehab Sbieh |
|---|---|---|---|
| Computer Engineering | Computer Engineering | Computer Engineering | Computer Engineering |
| AbdelrahmanSamha@Outlook.com | alhajahmad670@gmail.com | nermeennedal11@gmail.com | ehabsbaih9@gmail.com |


Amman, Jordan

12/2024

# Acknowledgments

We would like to extend our sincere gratitude to all those who contributed to the success of this project. First, we thank our professor and mentor, Dr. Dheya Gazi, for his valuable guidance and support throughout the project.

We also appreciate the help of our contest mentors, Abdelrahman Alshanaq, Hassan Taqi Aldeen, and Essa Qandah, whose expertise and feedback were crucial in overcoming technical challenges and improving our design.

Additionally, we would like to acknowledge our former teammate, Raneem Al Samaky, for her contributions to the project before stepping down from the contest. Her efforts were much appreciated.

Finally, we thank everyone who provided assistance and feedback during the course of this project.

# Abstract

This project involves the design, optimization, and evaluation of a 5-stage pipeline MIPS processor, built to enhance throughput and frequency through iterative optimizations. Initially, a single-cycle RISC MIPS processor supporting 22 instructions was designed. Subsequently, a 5-stage pipeline was introduced, incorporating various forwarding mechanisms and a branch predictor, aimed at improving instruction throughput. The project focuses on evaluating the trade-offs between the single-cycle processor and the pipelined design, highlighting the speedup gain achieved through pipelining.

In addition to the hardware design, a cycle-accurate simulator was developed to aid in the verification process, providing a detailed visualization of the pipeline operation and data flow between stages. The simulator ensures that the design functions correctly and efficiently under real-world conditions. An assembler was also created to support the 22 instructions, including two pseudo-instructions that are not natively supported by the hardware, further bridging the gap between the hardware and software aspects of the project.

The entire design process, from simulation and hardware optimization to final evaluation, emphasizes the importance of iterative testing and refinement in achieving an efficient and reliable processor architecture.

## Executive Summary

The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture is a widely studied Reduced Instruction Set Computer (RISC) design known for its simplicity and efficiency. Its straightforward instruction format and load-store architecture make it an ideal foundation for exploring computer architecture concepts, including pipelining, hazard management, and branch prediction.

The **single-cycle processor design** serves as a foundational concept in computer architecture, offering a streamlined approach to executing instructions. It ensures that all stages of instruction execution—**fetch, decode, execute, memory access, and writeback**—are completed within a single clock cycle. This uniformity in instruction timing provides clarity and simplicity, making it an excellent starting point for understanding processor design principles.

A defining feature of this architecture is its fixed **Cycles Per Instruction (CPI)** of one, meaning that every instruction is guaranteed to complete in a single clock cycle. However, this simplicity comes with a significant trade-off: the clock cycle must be long enough to accommodate the slowest instruction. As a result, faster instructions are constrained by the same clock duration as slower ones, leading to inefficiencies and underutilization of hardware resources. This limitation is particularly problematic for complex instruction sets, where variability in execution times is common.

In our implementation, we designed a **single-cycle processor supporting 22 instructions**, ensuring functional coverage for a range of operations. The architecture employs a **word-addressable memory model starting at index 0**, providing a straightforward and efficient approach to memory access. While the single-cycle processor is an effective educational tool and a strong baseline for understanding fundamental concepts, its inherent limitations in scalability and performance highlighted the need to explore more advanced designs, such as pipelining.

To address the inefficiencies of the single-cycle design, we developed a **5-stage pipelined processor**. Pipelining mitigates the performance bottlenecks of single-cycle architectures by dividing instruction execution into discrete stages—**fetch, decode, execute, memory access, and writeback**—that operate concurrently. This parallelism allows multiple instructions to be processed simultaneously, significantly increasing throughput. Once the pipeline is filled, one instruction is completed per clock cycle, enabling the processor to achieve much higher efficiency and performance compared to the single-cycle design.

In our pipelined architecture, careful attention was given to optimizing **throughput** and **clock frequency** through an iterative design process. This involved improving the data path, implementing efficient **forwarding mechanisms** to resolve data hazards, and designing robust **hazard resolution units** to handle stalls

Two primary pipeline designs were developed, each with distinct advantages and trade-offs. The first design achieved a clock frequency of **99 MHz** but required the pipeline to stall in the event of a **load-word dependency**, where a subsequent instruction depends on the result of a load instruction. The second design operated at a slightly lower clock frequency of **90.7 MHz** but successfully addressed this dependency, eliminating the need for stalls in such cases.

To evaluate the trade-offs between these designs, we analyzed data from the **SPEC2000 benchmarks**, which revealed that approximately **35%–40%** of the instructions in the benchmarks were memory accesses. While specific data on the frequency of load-word dependencies was unavailable, we concluded that the **90.7 MHz**

**design** would outperform the **99 MHz design** if the executed program contained **9.5% or more load-word instructions** followed immediately by dependent instructions. This analysis highlights the intricate balance between clock frequency and pipeline efficiency in real-world scenarios and demonstrates the importance of tailoring designs to workload characteristics.

After selecting the **90.7 MHz design** as the foundation for further development, we implemented a **branch prediction mechanism** to improve pipeline efficiency. A review of academic literature revealed that most static branch prediction algorithms rely on the behavior of backward and forward branches. Backward branches, commonly encountered in loops, are highly likely to be **taken**, while forward branches, which are typically conditional, tend to be **not taken** more often.

In our design, we employed a static branch prediction strategy where **backward branches are predicted as taken**, keeping the pipeline full during loop iterations and only requiring a flush at the last iteration. For **forward branches**, we predicted them as **not taken**, in line with research suggesting that forward branches have a slightly higher probability of not being executed. This approach proves beneficial in loop-intensive programs, as it minimizes pipeline flushes and maintains high instruction throughput.

However, the implementation of branch prediction resulted in a slight reduction in overall frequency, lowering the design's clock speed to **84 MHz**. Despite this decrease, the performance gain from keeping the pipeline filled most of the time, especially in programs with heavy looping, compensates for the loss in frequency, ultimately leading to better overall performance in such cases.

One of the primary advantages of transitioning from a single-cycle processor to a pipelined design is the significant increase in processing speed. In our evaluation, the single-cycle processor operated at a frequency of 32 MHz, while the pipelined processor achieved a final frequency of 84 MHz. This increase in clock speed contributed to a substantial performance boost.

The actual performance improvement was measured as a **3.5x speed-up** between the two designs. The pipelined architecture benefits from parallelism, as multiple instructions are processed simultaneously at different stages, significantly increasing throughput. This allows the pipelined design to process more instructions in less time, eliminating the bottleneck present in the single-cycle design, where the entire instruction execution process must complete within one clock cycle.

While the pipelined design introduces additional complexity in terms of hazard management and branch prediction, the trade-off is well worth the performance gain, particularly for real-world applications where instruction-level parallelism is crucial.

To further validate and explore the behavior of the pipelined processor, a cycle-accurate simulator (CAS) was designed. This simulator models the pipelined processor's execution in a highly detailed and accurate manner, providing a step-by-step simulation of each instruction’s passage through the pipeline stages. By closely mimicking the behavior of the real hardware, the simulator allows for an in-depth analysis of how the processor handles various types of instructions, hazards, and dependencies in a real-world scenario. It helps to visualize the internal states of the processor during execution, making it an invaluable tool for understanding pipeline dynamics, optimizing performance, and debugging.

The CAS also integrates with an assembler designed specifically for the project. The assembler takes high-level MIPS assembly code and converts it into machine code that can be executed on the simulator. This integration ensures that the program flow from high-level code down to actual machine instructions is accurately

represented, allowing for precise testing and validation. With the simulator and assembler working in tandem, the design and optimization process of the pipelined processor becomes much more efficient, enabling quick iterations and thorough testing of various features, such as forwarding, hazard resolution, and branch prediction mechanisms.

Together, the cycle-accurate simulator and assembler enhance the development and evaluation process of the pipelined architecture, providing insights into potential optimizations and ensuring the correctness of the implementation before hardware deployment.

# Contents

# Introduction

## 1. Overview of the MIPS Instruction Set Architecture (ISA)

The **Instruction Set Architecture (ISA)** is a fundamental aspect of computer system design, defining the rules and structure for how a processor communicates with software. It specifies the set of operations a processor can perform, the encoding of instructions, the use of registers, and the methods for accessing memory and I/O devices. Essentially, the ISA serves as the boundary between hardware and software, enabling programs to execute on a processor in a predictable and standardized manner. By abstracting the complexities of hardware, the ISA ensures compatibility across different implementations of the same architecture, allowing developers to write software without needing to know the underlying hardware details.

### 1.1 Introduction to MIPS ISA

The **MIPS Instruction Set Architecture (ISA)** is a foundational design in computer science that follows the principles of **RISC (Reduced Instruction Set Computing)**. Developed in the early 1980s at Stanford University under the guidance of John L. Hennessy, MIPS (Microprocessor without Interlocked Pipeline Stages) was designed to simplify processor implementation while achieving high performance. Over time, it became a widely adopted ISA in both academia and industry, powering everything from educational tools to embedded systems.

### 1.2 Design Philosophy of MIPS ISA

**RISC Principles:**

The MIPS ISA embodies RISC principles, which emphasize simplicity and efficiency in instruction execution. Key features include:

- **Small, Fixed Instruction Set**: The ISA uses a limited number of simple instructions that can be executed within a one clock cycle.
- **Uniform Instruction Length**: All instructions are 32 bits long, simplifying the fetch and decode stages of a pipeline.
- **Load/Store Architecture**: Only load (lw) and store (sw) instructions access memory; other operations work with registers.
- **Fewer Addressing Modes**: This reduces hardware complexity and improves decoding speed.

### 1.3 Pipeline Optimization:

MIPS was explicitly designed for efficient pipelining. Key features that support pipelining include:

- A consistent instruction length.
- Simplified instruction decoding due to fixed field positions.
- Avoidance of complex instructions that require multiple cycles to execute.

## 1.4 Key Features of the MIPS ISA

- **Register-based Operations**: MIPS uses 32 general-purpose registers to minimize the reliance on slower memory access.
- **Efficient Branching**: The ISA supports conditional and unconditional branch instructions, allowing for precise control flow.
- **Hardware Simplicity**: The design minimizes interlocked pipeline stages, making implementations faster and more power efficient.
- **Scalability**: The simplicity of the ISA makes it easy to scale for different applications, from embedded systems to high-performance computing.

## 1.5 Significance of MIPS ISA

- **Educational Use:**
  MIPS is a cornerstone of computer architecture education. Its clean and minimalistic design makes it ideal for explaining concepts like instruction pipelines, registers, and memory organization.
- **Industrial Applications:**
  While it has largely been replaced by more advanced architectures like ARM in commercial markets, MIPS is still used in certain embedded systems, routers, and network devices.

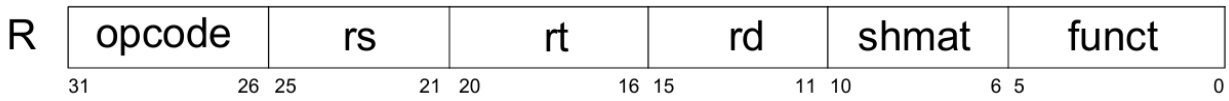
## 2. Instruction Formats

The MIPS Instruction Set Architecture (ISA) organizes its instructions into three main formats: **R-type (Register-type)**, **I-type (Immediate-type)**, and **J-type (Jump-type)**. These formats are designed to ensure consistency, simplify decoding, and optimize execution in a pipeline. Each instruction in MIPS is 32 bits long, and the layout of these bits varies depending on the type of instruction.

### 2.1 R-Type (Register-Type) Format

The R-type format is used for arithmetic, logical, and shift operations that work exclusively with registers.

- **Fields**:
  - **Opcode (6 bits)**: Specifies the instruction category (e.g., arithmetic or logical).
  - **Rs (5 bits)**: Source register 1.
  - **Rt (5 bits)**: Source register 2.
  - **Rd (5 bits)**: Destination register.
  - **shamt (5 bits)**: Shift amount (used for shift instructions).
  - **funct (6 bits)**: Specifies the exact operation (e.g., add, sub).
- **Example**: The $add\ \$t0, \$t1, \$t2$ instruction adds the contents of $t1 and $t2 and stores the result in $t0.
  Binary layout:

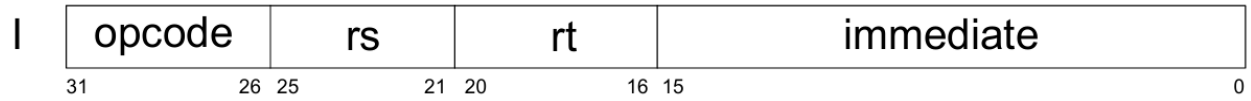



### 2.2 I-Type (Immediate-Type) Format

The I-type format is used for instructions that involve constants (immediate) or memory operations.

- **Fields:**
  - **Opcode (6 bits)**: Specifies the instruction category (e.g., load, store, branch).
  - **Rs (5 bits)**: Source register.
  - **Rt (5 bits)**: Destination register (or another operand for branches).
  - **Immediate (16 bits)**: A constant value or an offset for memory/branch operations.

- **Example**: The $addi\ \$t0, \$t1, 5$ instruction adds the immediate value 5 to the content of $t1 and stores the result in $t0.
  Binary layout:




### 2.3 J-Type (Jump-Type) Format

The J-type format is used for jump instructions, where the target address is a part of the instruction.

- **Fields**:
  - **Opcode (6 bits)**: Specifies the jump instruction type.
  - **Address (26 bits)**: Represents the jump target address (upper 4 bits are taken from PC).
- **Example**: The $j\ 0x00400000$ instruction sets the Program Counter (PC) to the target address 0x00400000.
  Binary layout:


## 3. Comparison of Instruction Formats

- **R-Type**: Used for operations entirely within the processor's registers.
- **I-Type**: Handles memory and immediate data, enabling interaction with memory and constants.
- **J-Type**: Directs the flow of control to specific addresses.

## 4. Instruction Categories

The MIPS Instruction Set Architecture (ISA) classifies its instructions into several categories based on their functionality. This categorization reflects the simplicity of MIPS and its adherence to RISC design principles. The key instruction categories are **Arithmetic and Logical Instructions**, **Data Transfer Instructions**, **Control Flow Instructions**, and **Other Instructions**. Each category plays a specific role in implementing programs efficiently.

### 4.1 Arithmetic and Logical Instructions

These instructions perform operations on data in registers, including basic arithmetic, bitwise logic, and shift operations. They exclusively operate on registers to maintain the load/store design of MIPS.

#### 4.1.1 Arithmetic Instructions:

Perform addition, subtraction, and related operations.

- Example:
  - $add\ \$t0, \$t1, \$t2$: Adds the values in $t1 and $t2, storing the result in $t0.
  - $sub\ \$t0, \$t1, \$t2$: Subtracts the value in $t2 from $t1.

#### 4.1.2 Logical Instructions:

Perform bitwise operations like AND, OR, XOR, and NOR.

- Example:
  - $and\ \$t0, \$t1, \$t2$: Performs bitwise AND on $t1 and $t2, storing the result in $t0.
  - $or\ \$t0, \$t1, \$t2$: Performs bitwise OR on $t1 and $t2.

#### 4.1.3 Shift Instructions:

Shift data in registers by a specified number of bits.

- Example:
  - $sll\ \$t0, \$t1, 2$: Shifts the value in $t1 left by 2 bits, storing the result in $t0.

## 4.1 Data Transfer Instructions

These instructions move data between registers and memory. They form the foundation of MIPS's load/store architecture.

#### 4.2.1 Load Instructions:

Read data from memory into a register.

- Example:

  $lw\ \$t0, 4(\$t1)$: Loads the 32-bit word from the memory address ($t1 + 4) into $t0.

#### 4.2.2 Store Instructions:

Write data from a register to memory.

- Example:

  $sw\ \$t0, 8(\$t1)$: Stores the 32-bit word in $t0 into the memory address ($t1 + 8).

## 4.2 Control Flow Instructions

These instructions direct the flow of execution by altering the program counter (PC).

#### 4.3.1 Branch Instructions:

Used for conditional control flow.

- Example:

  $beq\ \$t0, \$t1$, label: Branches to label if $t0 equals $t1.

  $bne\ \$t0, \$t1$, label: Branches to label if $t0 does not equal $t1.

#### 4.3.2 Jump Instructions:

Directly change the PC to a specific address for unconditional control flow.

- Example:

  **j label:** Jumps to the instruction at label.

### 4.3 Other Instructions

This category includes miscellaneous instructions, such as system calls and pseudo-instructions.

#### 4.4.1 System Call Instructions:

Trigger an operating system service, often used for input/output operations.

- Example:

  Syscall: Executes the system call specified by a value in register $v0.

#### 4.4.2 Pseudo-Instructions:

High-level instructions that are not part of the core ISA but are translated into one or more real instructions by the assembler.

- Example:

  $BGEZ\ \$t0, lable$ : (translated to $slt\ \$27, \$t0, \$zero$ and $beq\ \$27, \$t0, lable$ ).

## 5. Registers in MIPS

Registers are a critical part of the MIPS Instruction Set Architecture (ISA). They provide fast, direct access to data for the processor, ensuring high performance. MIPS uses 32 general-purpose registers, each 32 bits wide, following the RISC design principle of simplicity and efficiency. Additionally, there are a few special-purpose registers for specific tasks like multiplication, division, and program control.

### 5.1 General-Purpose Registers

MIPS defines 32 registers, conventionally referred to as $0 to $31. These registers are further classified based on their intended use:

- **Reserved and Fixed Registers**:

  $zero (R0): Constantly holds the value 0. This is useful for initializing registers or performing operations without additional memory access.
  Example: add $t0, $t1, $zero assigns the value of $t1 to $t0.

- **Temporary Registers**:

  $t0-$t9: Used for temporary storage of data during computation.
  Example: A program performing intermediate calculations might use these registers.

- **Saved Registers**:

  $s0-$s7: Preserve values across function calls. They are saved and restored by the programmer or compiler as needed.

- **Function Call Registers**:

  $a0-$a3: Pass arguments to functions.

  $v0-$v1: Store return values from functions.

- **Kernel and Reserved Registers**:
  - $k0-$k1: Reserved for operating system kernel operations.
- **Stack Pointer and Return Address**:
  - $Sp: Points to the current top of the stack in memory.
  - $Sa: Stores the return address for function calls.

| Name | Number | Usage Availability |
|---|---|---|
| $zero | 0 | N.A. |
| $at | 1 | No |
| $v0-$v1 | 2-3 | No |
| $a0-$a3 | 4-7 | No |
| $t0-$t7 | 8-15 | No |
| $s0-$s7 | 16-23 | Yes |
| $t8-$t9 | 24-25 | No |
| $k0 | 26 | No |
| $k1 | 27 | N.A. |
| $gp | 28 | Yes |
| $sp | 29 | Yes |
| $fp | 30 | Yes |
| $ra | 31 | no |

### 5.2 Special-Purpose Registers

MIPS includes additional registers for specific functions that enhance processor operations.

- **Program Counter (PC)**:

  Holds the address of the current instruction being executed. The PC automatically increments to the next instruction unless altered by a branch or jump instruction.

- **HI and LO Registers**:

  Used to store the results of multiplication and division operations.
  Example:

  After $mult\ \$t0, \$t1$, the product is stored in HI (upper 32 bits) and LO (lower 32 bits).

  $mflo\ \$t2$: Moves the value from LO to $t2.

### 5.3 Role of Registers in MIPS Programming

- **Reduced Memory Access**: Registers significantly reduce the need for slow memory operations, enhancing performance.
- **Pipeline Efficiency**: Using registers simplifies the design of pipelined processors, as data is readily accessible.
- **Simplified Programming Model**: A fixed set of registers with predefined roles streamlines assembly language programming.

## 6. Memory Addressing Modes

In the MIPS Instruction Set Architecture (ISA), memory addressing modes define how the processor identifies the memory locations of operands. MIPS employs a limited number of addressing modes, adhering to its RISC design philosophy, which prioritizes simplicity and efficiency. These modes primarily support load/store operations, as all arithmetic and logical instructions operate exclusively on registers.

### 6.1 Immediate Addressing

The operand is embedded directly within the instruction as an immediate value.

- **Usage**: Common in arithmetic operations involving constants.
- **Example**:

$$addi\ \$t0, \$t1, 10$$

Adds the constant 10 to the value in $t1 and stores the result in $t0.


### 6.2 Register Addressing

The operand is stored in a register, and the instruction directly references the register.

- **Usage**: Frequently used in arithmetic, logical, and shift instructions.
- **Example**:

$$add\ \$t0, \$t1, \$t2$$

Adds the values in $t1 and $t2 and stores the result in $t0.

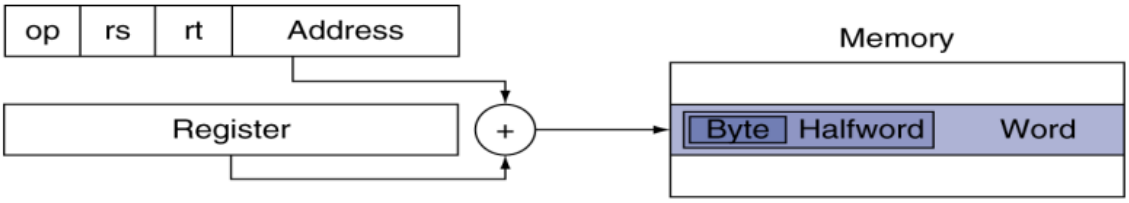
### 6.3 Base (Displacement) Addressing

A memory address is computed by adding a base register's value to a constant offset (displacement).

- **Usage**: Used for accessing data in memory.
- **Example**:

$$lw\ \$t0, 4(\$t1)$$

Loads the word from the memory address calculated as ($t1 + 4) into $t0.



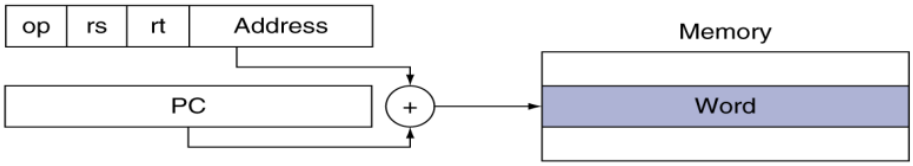

### 6.4 PC-Relative Addressing

The memory address is computed by adding a constant offset to the Program Counter (PC). This is primarily used in branch instructions.

- **Usage**: Supports control flow changes within a limited range.
- **Example**:

$$beq\ \$t0, \$t1, label$$

- If $t0 equals $t1, the program branches to the address calculated as (PC + offset).



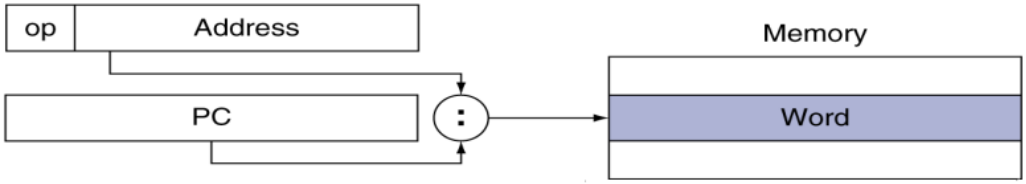

### 6.5 Pseudo-Direct Addressing

Used in jump instructions, where the target address is formed by combining a 26-bit address from the instruction with the upper bits of the PC.

- **Usage**: Supports control flow changes to distant locations.
- **Example**:

$$j\ label$$

Sets the PC to the target address specified by the instruction.



### 6.6 Comparison of Addressing Modes

| Addressing Mode | Operand Location | Typical Usage |
|---|---|---|
| Immediate Addressing | Operand is part of instruction | Arithmetic with constants |
| Register Addressing | Operand in a register | Arithmetic and logical operations |
| Base Addressing | Operand in memory (via base + offset) | Data transfer (load/store) |
| PC-Relative Addressing | Address = PC + offset | Branch instructions |
| Pseudo-Direct Addressing | Address = PC upper bits + 26-bit offset | Jump instructions |

- **Simplicity**: The limited number of modes reduces hardware complexity and enhances decoding speed.
- **Pipeline Efficiency**: Consistent formats and straightforward addressing simplify memory operations within the pipeline.
- **Flexibility**: Despite their simplicity, MIPS addressing modes accommodate a wide range of programming needs, from accessing local variables to controlling program flow.

## Supported Instructions by the Processor

1. **Introduction to Supported Instructions**

   **In this section, we explore the supported instructions in our custom MIPS processor. These instructions form the core functionality of the system, allowing arithmetic operations, memory access, control flow, and logical comparisons. Additionally, two pseudoinstructions are supported, simplifying programming tasks while being translated into hardware-compatible instructions by the assembler.**

2. **Detailed Explanation of Supported Instructions**

### Arithmetic and Logical Instructions

- **ADD: Adds two registers and stores the result in a destination register.**
  **Example: ADD $t0, $t1, $t2.**
- **SUB: Subtracts the value in the second register from the first register and stores the result.**
  **Example: *SUB* $t0, $t1, $t2.**
- **AND, OR, XOR, NOR: Perform respective bitwise operations between two registers and store the result in the destination register.**
  **Examples:**
  - ***AND* $t0, $t1, $t2**
  - **OR $t0, $t1, $t2**
- **SLT (Set on Less Than): Compares two registers and sets the destination register to 1 if the first is less than the second; otherwise, sets it to 0.**
  **Example: SLT $t0, $t1, $t2.**

### Data Transfer Instructions

- **LW (Load Word): Loads a 32-bit word from memory into a register. The memory address is computed as the sum of a base address in a register and an offset.**
  **Example: *LW* $t0, 4($t1)**
- **SW (Store Word): Stores a 32-bit word from a register into memory, with the address computed similarly to LW.**
  **Example: *SW* $t0, 4($t1)**

## Branch and Jump Instructions

- **BEQ (Branch if Equal): Branches to the specified label if the two registers contain equal values.**
  **Example: *BEQ* $*t0*, $*t1*, *label*.**

- **BNE (Branch if Not Equal): Branches to the specified label if the two registers are not equal.**
  **Example: *BNE* $*t0*, $*t1*, *label*.**

- **J (Jump): Jumps to the specified label.**
  **Example: *J label*.**

- **JR (Jump Register): Jumps to the address stored in a register. Typically used to return from a subroutine.**
  **Example: *JR* $*ra*.**

- **JAL (Jump and Link): Jumps to the specified label and stores the return address in the $ra register.**
  **Example: *JAL label*.**

## Shift Instructions

**SLL (Shift Left Logical) and SRL (Shift Right Logical):**
**These instructions shift the contents of a register to the left or right by a specified number of bits and store the result in the destination register.**

**Example: SLL $t0, $t1, 4.**

**Note on Implementation:**
**In our processor, the implementation of SLL and SRL is modified. The typical MIPS R-type instruction format requires a multiplexer to handle the Rs field. To avoid the additional hardware overhead, the assembler changes the usual placement of the Rs field during encoding, effectively bypassing the need for an extra multiplexer. This optimization maintains the semantics of the instructions while simplifying the hardware design.**

## Immediate Instructions

- **ADDI, ORI, XORI, ANDI: Perform respective arithmetic or bitwise operations between a register and an immediate value.**
  **Example: *ADDI* $*t0*, $*t1*, 10.**

- **SLTI (Set Less Than Immediate): Compares a register value with an immediate value and sets the destination register accordingly.**
  **Example: *SLTI* $*t0*, $*t1*, 20**

## Pseudo-Instructions in My Processor

**What Are Pseudo-instructions?**

**Pseudoinstructions are high-level assembly instructions that do not have direct hardware implementations. Instead, they are translated by the assembler into equivalent machine instructions. These instructions make assembly programming more intuitive and less error-prone.**

**In our processor, pseudoinstructions utilize register $27 as a temporary flag to store intermediate results or conditions for branching. The assembler ensures that no other instruction writes to $27, preserving its functionality for pseudoinstructions.**

- **Supported Pseudoinstructions**

1. **BGEZ (Branch if Greater Than or Equal to Zero):**
   **This instruction checks if a register value is greater than or equal to zero. If true, it branches to the specified label.**
   **Translation:**

$$\boldsymbol{SLT}\ \$\mathbf{27}, \$\boldsymbol{t0}, \$\boldsymbol{zero}$$

$$\boldsymbol{BEQ}\ \$\mathbf{27}, \$\boldsymbol{zero}, \boldsymbol{label}$$

- **BLTZ (Branch if Less Than Zero):**
  **This instruction checks if a register value is less than zero. If true, it branches to the specified label.**
  **Translation:**

$$\boldsymbol{SLT}\ \$\mathbf{27}, \$\boldsymbol{t0}, \$\boldsymbol{zero}$$

$$\boldsymbol{BNE}\ \$\mathbf{27}, \$\boldsymbol{zero}, \boldsymbol{label}$$

**Explanation:**
**For both pseudoinstructions, the first translated instruction compares the register value against zero, setting $27 to 1 if the condition is met. The second instruction performs a conditional branch based on the value of $27.**

## 3. Reference sheet

**At the end of this chapter, the Reference Sheet provides a comprehensive visual summary of all supported instructions in the processor. This sheet serves as a quick reference for programmers, consolidating essential details about instruction formats and examples.**

# SB2P Reference Data

**CORE INSTRUCTION SET**

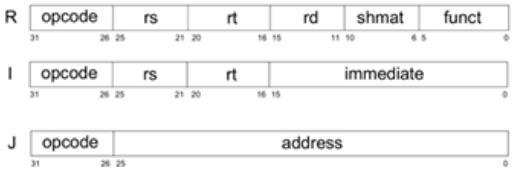
| Name | MNEMONIC- | FORMAT | Operation | | Opcode/ Funct (hex) |
|---|---|---|---|---|---|
| Add | ADD | R | R[rd] = R[rs] + R[rt] | (1) | 0 / 20 hex |
| Sub | SUB | R | R[rd] = R[rs] - R[rt] | (1) | 0 / 22 hex |
| And | AND | R | R[rd] = R[rs] & R[rt] | | 0 / 24 hex |
| Or | OR | R | R[rd] = R[rs] \| R[rt] | | 0 / 25 hex |
| Xor | XOR | R | R[rd] = R[rs] ^ R[rt] | | 0 / 26 hex |
| Nor | NOR | R | R[rd] = ~(R[rs] \| R[rt]) | | 0 / 27 hex |
| Slt | SLT | R | R[rd] = (R[rs] < R[rt]) ? 1 : 0 | | 0 / 2a hex |
| Load Word | LW | I | R[rt] = MEM[R[rs] + SignExtImm] | (2) | 23 hex |
| Store Word | SW | I | MEM[R[rs] + SignExtImm] = R[rt] | (2) | 2b hex |
| Branch Equal | BEQ | I | if (R[rs] == R[rt]) PC += SignExtImm | (4) | 4 hex |
| Branch Not Equal | BNE | I | if (R[rs] != R[rt]) PC += SignExtImm | (4) | 5 hex |
| Jump | J | J | PC = address | (3) | 2 hex |
| Jump Register | JR | R | PC = R[rs] | | 0 / 8 hex |
| Jump And Link | JAL | J | $ra = PC+4; PC = address | (3) | 3 hex |
| Add Immediate | ADDI | I | R[rt] = R[rs] + SignExtImm | (1)(2) | 8 hex |
| Or Immediate | ORI | I | R[rt] = R[rs] \| ZeroExtImm | (2) | d hex |
| Xor Immediate | XORI | I | R[rt] = R[rs] ^ ZeroExtImm | | e hex |
| And Immediate | ANDI | I | R[rt] = R[rs] & ZeroExtImm | | c hex |
| Sll | SLL | R | R[rd] = R[rs] << shamt | | 0 / 0 hex |
| Srl | SRL | R | R[rd] = R[rs] >> shamt | | 0 / 2 hex |
| Slti | SLTI | I | R[rt] = (R[rs] < SignExtImm) ? 1 : 0 | (2) | a hex |
| Sgt | SGT | R | R[rd] = (R[rs] > R[rt]) ? 1 : 0 | | 0 / 2C hex |

(1) Operands considered unsigned numbers (vs. 2s comp.)
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) JumpAddr = { PC[31:28], address, 2'b0 }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }

**Pseudo Instructions**

| NAME | MNEMOVIC | OPERATION |
|---|---|---|
| Branch >= Zero | BGEZ | slt $27, $rs, $zero;<br>beq $27, $rs, label |
| Branch < Zero | BLTZ | slt $27, $rs, $zero;<br>bne $27, $rs, label |

**BASIC INSTRUCTION FORMATS**




| Name | Number | Usage Availability |
|---|---|---|
| $zero | 0 | N.A. |
| $at | 1 | No |
| $v0-$v1 | 2-3 | No |
| $a0-$a3 | 4-7 | No |
| $t0-$t7 | 8-15 | No |
| $s0-$s7 | 16-23 | Yes |
| $t8-$t9 | 24-25 | No |
| $k0 | 26 | No |
| $k1 | 27 | N.A. |
| $gp | 28 | Yes |
| $sp | 29 | Yes |
| $fp | 30 | Yes |
| $ra | 31 | no |

## Single Cycle

# 1. Single-Cycle Processor Design

### 1.1 Introduction:

The single-cycle processor design is a cornerstone concept in computer architecture, offering a streamlined and efficient method to execute instructions. It achieves simplicity by ensuring that all stages of instruction execution—fetch, decode, execute, memory access, and write-back—are completed within a single clock cycle. This uniformity in instruction timing provides clarity and a straightforward implementation, making it an excellent starting point for understanding processor design principles.

A critical feature of the single-cycle architecture is its fixed Cycles Per Instruction (CPI) of one, which guarantees that each instruction takes exactly one clock cycle to complete. However, this approach comes with trade-offs: to accommodate the slowest instruction, the clock cycle must be sufficiently long, potentially limiting overall performance. This trade-off highlights the balance between simplicity and efficiency in processor design.

This section explores the architecture, design, and implementation of a single-cycle processor, delving into its instruction flow, Datapath components, and control unit logic. By examining the advantages, limitations, and practical applications of this design, the report provides insights into how foundational processor concepts pave the way for more advanced architectures.

### 1.2 Processor Overview

A single-cycle processor executes each instruction in a single clock cycle, encompassing all stages of instruction execution—fetch, decode, execute, memory access, and write-back. This architecture emphasizes simplicity and ease of understanding, making it ideal for educational purposes and foundational learning in computer architecture.

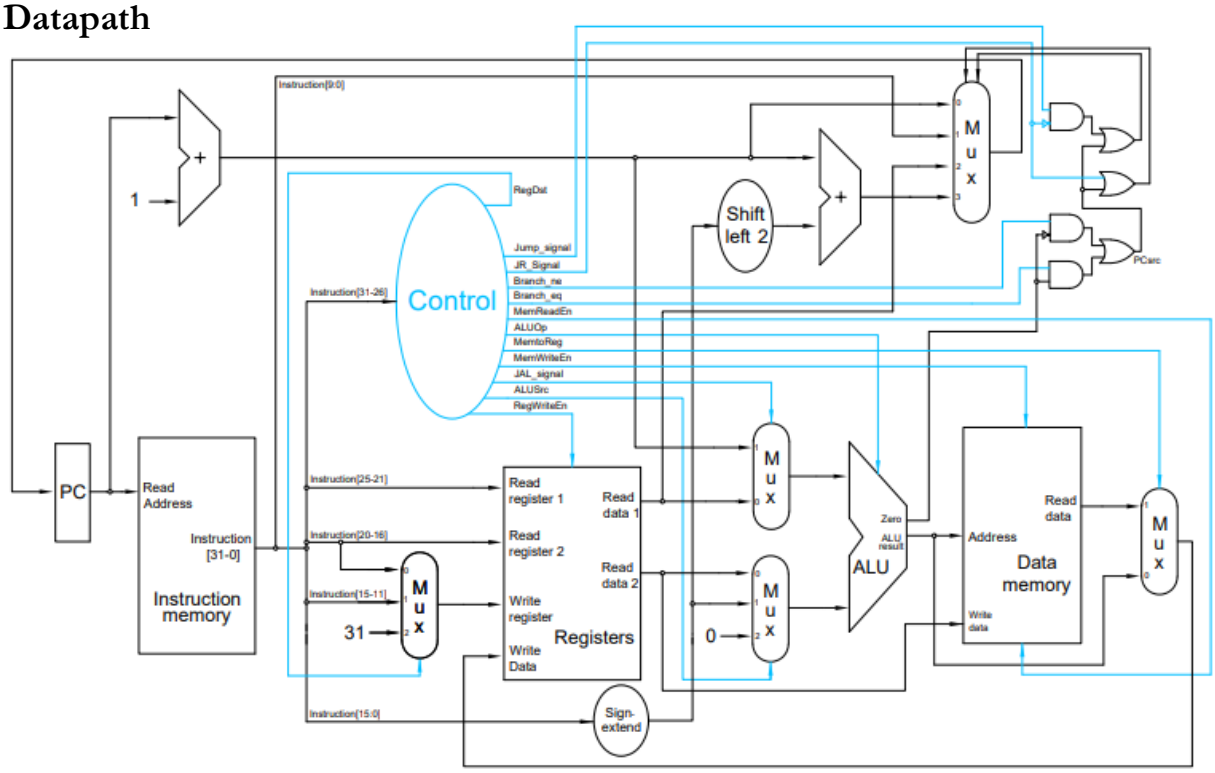
Key features of a single-cycle processor include:

- **Single Clock Cycle Per Instruction**: Each instruction is fully executed within one clock period.
- **Unified Datapath**: The Datapath integrates all required components, including the ALU, registers, and memory, to support instruction execution.
- **Simple Control Unit**: The control unit generates all necessary signals based on the instruction opcode, enabling efficient coordination among Datapath components.

### 1.3 Limitations:

- **Fixed Clock Cycle Duration**: The clock cycle must accommodate the execution of the longest instruction, leading to inefficiencies for simpler instructions.
- **Scalability Issues**: As instruction complexity increases, maintaining a single-cycle architecture becomes impractical.

## 2. Datapath

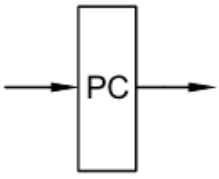



The single-cycle MIPS processor executes each instruction in a single clock cycle, combining instruction fetch, decode, execute, memory access, and write-back. Its datapath integrates key components like the **register file**, **ALU**, **instruction memory**, **data memory**, multiplexers, and sign extenders. These components work together, directed by control signals, to handle data flow and execute arithmetic, memory, and branch instructions efficiently. Diagrams below illustrate the operation of the Datapath for various instruction types.

The single-cycle Datapath is built around several key components, each playing a vital role in executing instructions. These components work together to fetch, decode, execute, and write back results, ensuring seamless data flow and functionality. Below is a detailed breakdown of each component and its purpose in the processor.

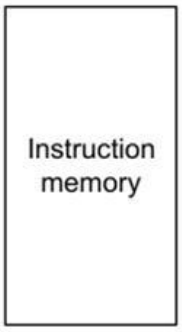
### Program Counter (PC):

The Program Counter keeps track of the address of the current instruction and updates sequentially to the next instruction. In the case of branches or jumps, it updates to the target address instead.



### Instruction Memory:

A dedicated memory component that stores the program's instructions. It is accessed using the address provided by the Program Counter (PC), ensuring the CPU fetches the correct instruction to execute at each step. Size




### Adders in the Datapath:

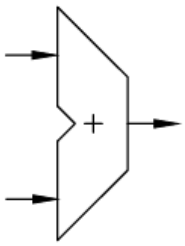
The Datapath includes two key adders that play distinct roles:

#### PC Adder:

This adder increments the Program Counter (PC) by 1 to compute the address of the next sequential instruction. Since the memory is word-addressable, this ensures proper alignment for instruction fetching.

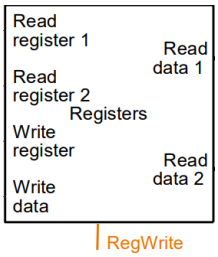
#### Branch Target Adder:

This adder calculates the target address for branch instructions. It adds the sign-extended, left-shifted immediate value to the incremented PC (PC + 1), enabling conditional branching.



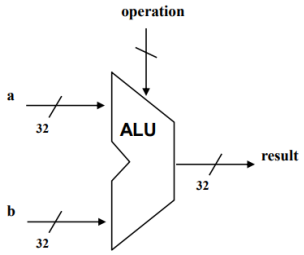

### Register File:

The register file is a key component of the single-cycle MIPS processor, consisting of 32 general-purpose registers, each 32 bits wide. It supports two simultaneous read operations and one write operation, controlled by the instruction's rs, rt, and rd fields, with the RegWrite signal enabling writes. The register file provides operands to the ALU and stores results from computations or memory operations, ensuring efficient instruction execution within a single clock cycle.



## Arithmetic Logic Unit (ALU):

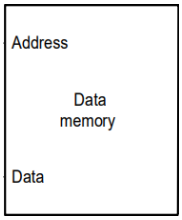
The Arithmetic Logic Unit (ALU) is a fundamental component of a computer processor, responsible for performing mathematical, logical, and decision-making operations.




The ALU is a critical element of the processor's Datapath, directly influencing the speed and efficiency of instruction execution. Its design is highly optimized to perform operations with minimal delay, ensuring smooth and reliable processing in both simple and complex computing systems.

## Data Memory:

Data memory is a vital part of the processor's datapath, storing data required during program execution, such as variables and intermediate results. It is accessed by lw (load word) and sw (store word) instructions in the MIPS architecture, enabling efficient read and write operations. Acting as a bridge between the CPU and main memory, it ensures rapid data retrieval and updates, directly impacting the processor's performance.




## Sign Extender:

The sign extender converts 16-bit immediate values to 32 bits by extending the most significant bit (MSB) to the upper 16 bits, preserving the value's sign. It is essential for I-type instructions like lw, sw, and addi, allowing immediate values to integrate correctly with the 32-bit datapath.


## Multiplexers:

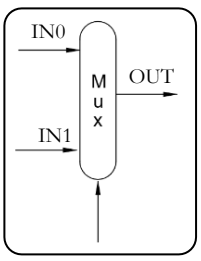
A multiplexer is a fundamental digital component used in logic circuits to select one of several input signals and forward it to a single output line.

- **Inputs**: All inputs in multiplexers are equal
- **Selector**: The number of selection lines determines how many input signals the multiplexer can handle, with $2^n$ inputs for n selection lines.
- **Output**: one output determined from input by selector value.

### 32-Bit 2×1 Mux:

A **32-bit 2-to-1 multiplexer (MUX)** is a digital circuit used to select between two 32-bit data inputs based on a single selection line (S):

- 2 inputs, each input 32-bit wide.
- One output, 32-bit wide



- Selector: 1-bit wide to select input 1 or input 2.

S

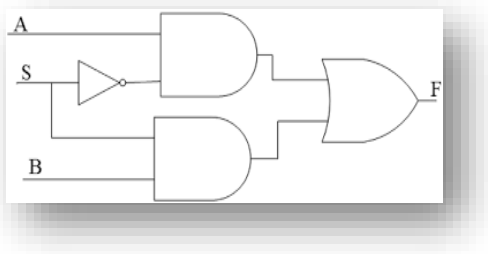
A multiplexer (MUX**)** can be implemented using basic logic gates such as **AND**, **OR**, and **NOT**. Have 2 inputs (A and B), 1 selector (S) and one output (Y).

$$0 \rightarrow Y = I_0;\ 1 \rightarrow Y = I_1;$$

Truth Table for 2-to-1 Mux:

| S | A | B | Y |
|---|---|---|---|
| **0** | **0** | × | **$I_0$=0** |
| **0** | **1** | × | **$I_0$=1** |
| **1** | × | **0** | **$I_1$=0** |
| **1** | × | **1** | **$I_1$=1** |

**Boolean expression:** $\boldsymbol{Y = (I_0 \cdot \bar{S}) + (I_1 \cdot S)}$




A 32-bit 2-to-1 MUX can be constructed using N individual 1-Bit 2-to-1 MUX circuits, each bit of the output is determined using the following equation:

$$Y[i] = (I_0[i] \cdot \bar{S}) + (I_1[i] \cdot S)$$

Repeat this logic for all 32 bits, with the same selection line S controlling all the bit-level multiplexers.

**32-Bit 4×1 Mux:**

A **32-bit 4-to-1 multiplexer (MUX)** is a digital circuit used to select between Four 32-bit data inputs based on a two-selection line **($S_0$, $S_1$)**:

- 4 Inputs, each input 32-bit wide.
- One output, 32-bit wide
- Selector: 2-bit wide to select one input from 4 input:

A multiplexer (MUX**)** can be implemented using basic logic gates such as **AND**, **OR**, and **NOT**. Have 4 input ($I_0$ , $I_1$ , $I_2$ , $I_3$), 2 selector ($S_0$, $S_1$) and one output (Y):

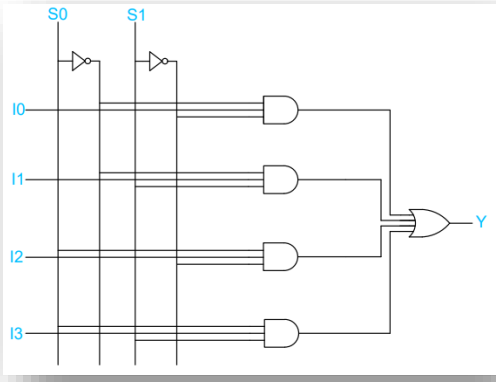
$0 \rightarrow Y = I_0$ ;  $1 \rightarrow Y = I_1$ ;  $2 \rightarrow Y = I_2$ ;  $3 \rightarrow Y = I_3$ ;

$$Y = (\bar{S_1} \cdot \bar{S_0} \cdot I_0) + (\bar{S_1} \cdot S_0 \cdot I_1) + (S_1 \cdot \bar{S_0} \cdot I_2) + (S_1 \cdot S_0 \cdot I_3)$$

| S1 | S0 | I0 | I1 | I2 | I3 | Y |
|---|---|---|---|---|---|---|
| **0** | **0** | **0** | **X** | **X** | **X** | **I0=0** |
| **0** | **0** | **1** | **X** | **X** | **X** | **I0=1** |
| **0** | **1** | **X** | **0** | **X** | **X** | **I1=0** |
| **0** | **1** | **X** | **1** | **X** | **X** | **I1=1** |

| | | | | | | |
|---|---|---|---|---|---|---|
| **1** | **0** | **X** | **X** | **0** | **X** | **I2=0** |
| **1** | **0** | **X** | **X** | **1** | **X** | **I2=1** |
| **1** | **1** | **X** | **X** | **X** | **0** | **I3=0** |
| **1** | **1** | **X** | **X** | **X** | **1** | **I3=1** |

Truth Table for 4-to-1 Mux:




A 32-bit 4-to-1 MUX can be constructed using N individual 1-Bit 4-to-1 MUX circuits, each bit of the output is determined using the following equation:

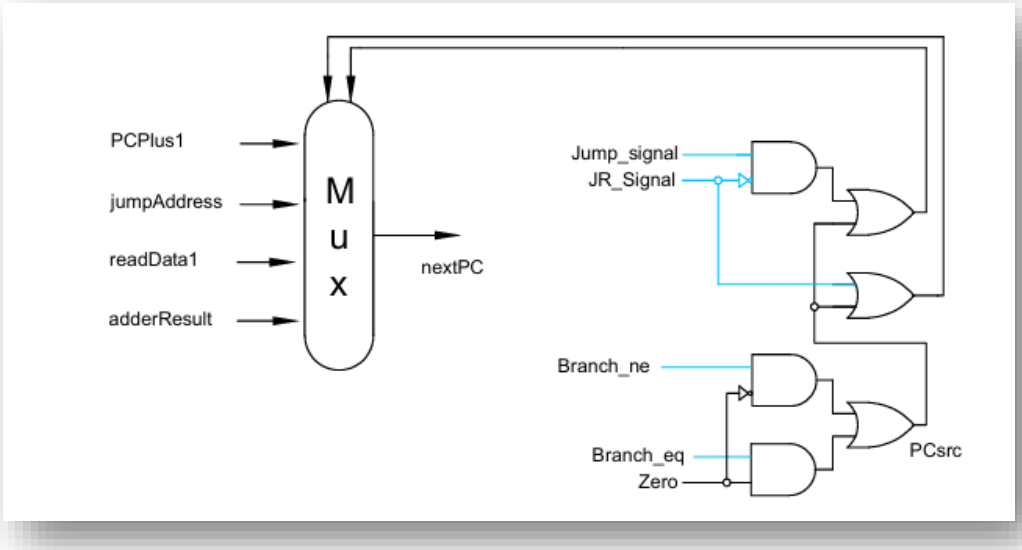
$$Y[i] = (\overline{S_1} \cdot \overline{S_0} \cdot I_0[i]) + (\overline{S_1} \cdot S_0 \cdot I_1[i]) + (S_1 \cdot \overline{S_0} \cdot I_2[i]) + (S_1 \cdot S_0 \cdot I_3[i])$$

Repeat this logic for all 32 bits, with the same selection line S controlling all the bit-level multiplexers.

## Next PC Determination Unit

### Introduction

This subsection explains the design and functionality of the unit responsible for determining the next PC value in the single-cycle processor. It provides the equations, logic, and truth table governing this unit and its impact on program execution.



**Equations and Explanation**

- **Next PC Selection Logic**
  The next PC value is selected by the mux4_1 multiplexer, which determines the appropriate source based on the control signals S0 and S1. These signals are computed using the control unit's outputs.

- **Control Signals**

  1. **PCsrc**: Activated to select the branch address (adder Result) based on branch conditions.
  2. **Jump signal**: Indicates the execution of a jump instruction.
  3. **JR Signal**: Indicates a jump to a register-specified address (readData1).

- **Branch Condition Logic**

$$ANDGate\ branchAnd1(.in1(zero),.in2(Branch\_eq),.out(Andout1));$$

$$ANDGate\ branchAnd2(.in1(\sim zero),.in2(Branch\_ne),.out(Andout2));$$

$$assign\ PCsrc\ =\ Andout1\ |\ Andout2;$$

The branch condition logic evaluates the zero flag alongside branch control signals (Branch_eq, Branch_ne) to determine if the branch address should be selected.

- **Selection Signal Logic**

$$assign\ S0\ =\ PCsrc\ |\ ((\sim JR\_Signal)\ \&\ Jump\_signal);$$

$$assign\ S1\ =\ JR\_Signal\ |\ PCsrc;$$

  1. **S0**: Determines the choice between the branch address, jump address, or default incremented PC.
  2. **S1**: Decides if the jump-register address is used.

- **Multiplexer Inputs**

  1. **IN0** (PCPlus1): Default next PC (PC incremented by 1).
  2. **IN1** (jump Address): Address for jump instructions.
  3. **IN2** (ReadData1): Register-specified address for jump-register instructions.
  4. **IN3** (adder Result): Address for branch instructions.

**Truth Table for Next PC Selection**

| S1 | S0 | Next PC | Description |
|---|---|---|---|
| **0** | 0 | PCPlus1 | Default: Next sequential instruction. |
| **0** | 1 | jumpAddress | Jump instruction. |
| **1** | 0 | readData1 | Jump-register instruction (JR). |
| **1** | 1 | adderResult | Branch instruction. |

**Final Explanation**

The mux4_1 multiplexer integrates control signals to determine the correct address for the following scenarios:

- **Default PC increment (PCPlus1)**: For sequential execution without control flow changes.
- **Jump (jumpAddress)**: For unconditional jump instructions (e.g., j).
- **Jump to Register (readData1)**: For register-based jump instructions (e.g., jr).
- **Branch (adderResult)**: For conditional branch instructions, based on Branch_eq and Branch_ne signals.

  This unit ensures smooth control flow by providing the correct address to the program counter (PC), driving instruction execution.

## 3. Instruction Flow

The instruction flow varies based on the type of instruction (R-type, I-type, J-type). Below is an explanation of how each instruction is executed:

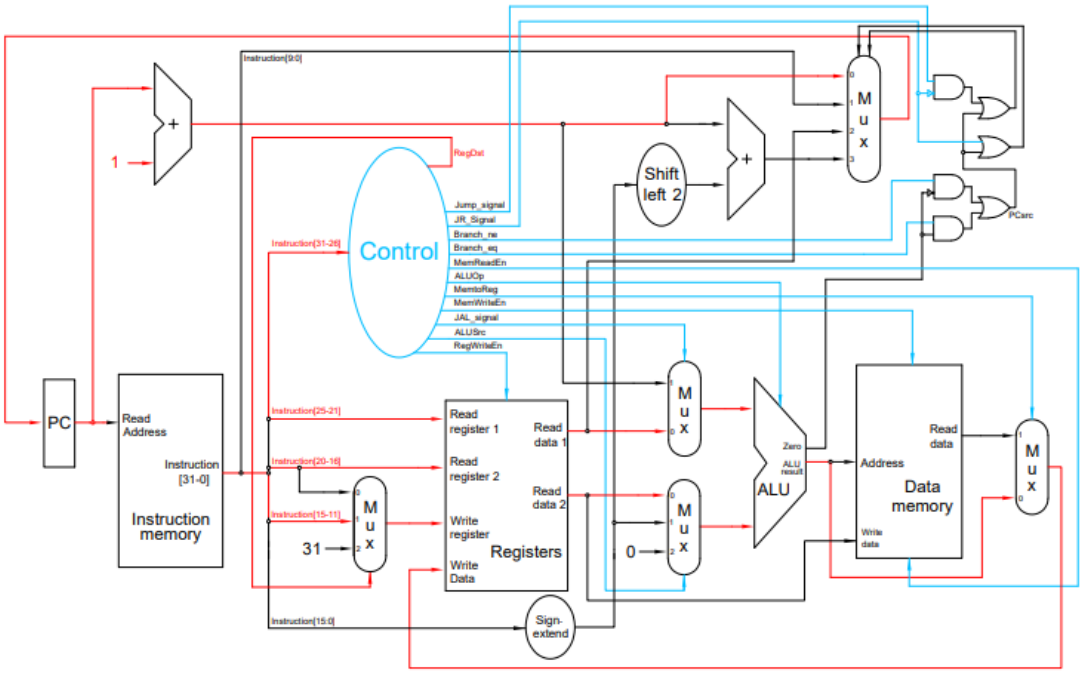
### R-Type Instruction Flow

**Example Instruction:** add $t1, $t2, $t3

**Steps:**

- Fetch the instruction from the instruction memory using the Program Counter (PC).
- Decode the instruction to identify the opcode and register operands.
- Read values from the specified registers ($t2 and $t3).
- Perform the ALU operation (addition in this case).
- Write the result back to the destination register ($t1).
- Update the PC to point to the next instruction.

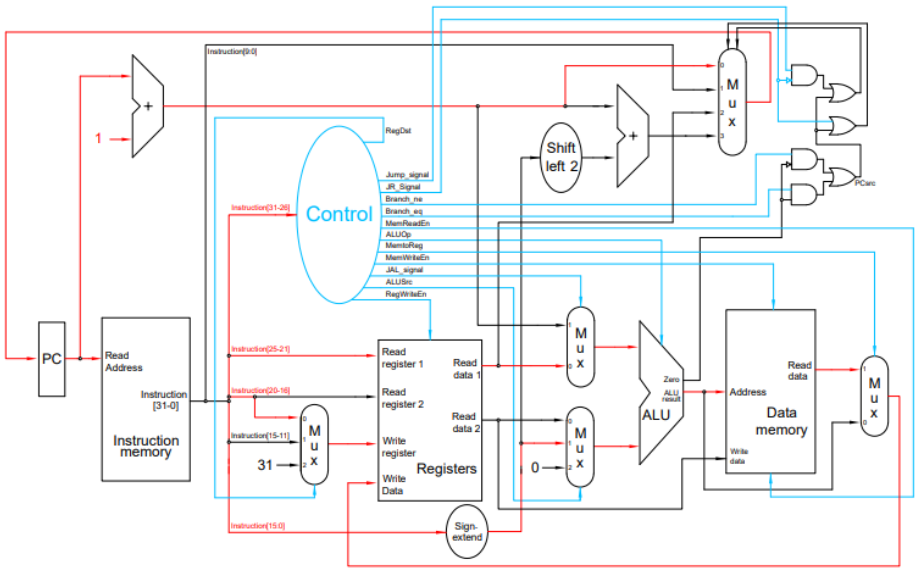
ADD $t1,$t2,$t3




## I-Type Instruction Flow

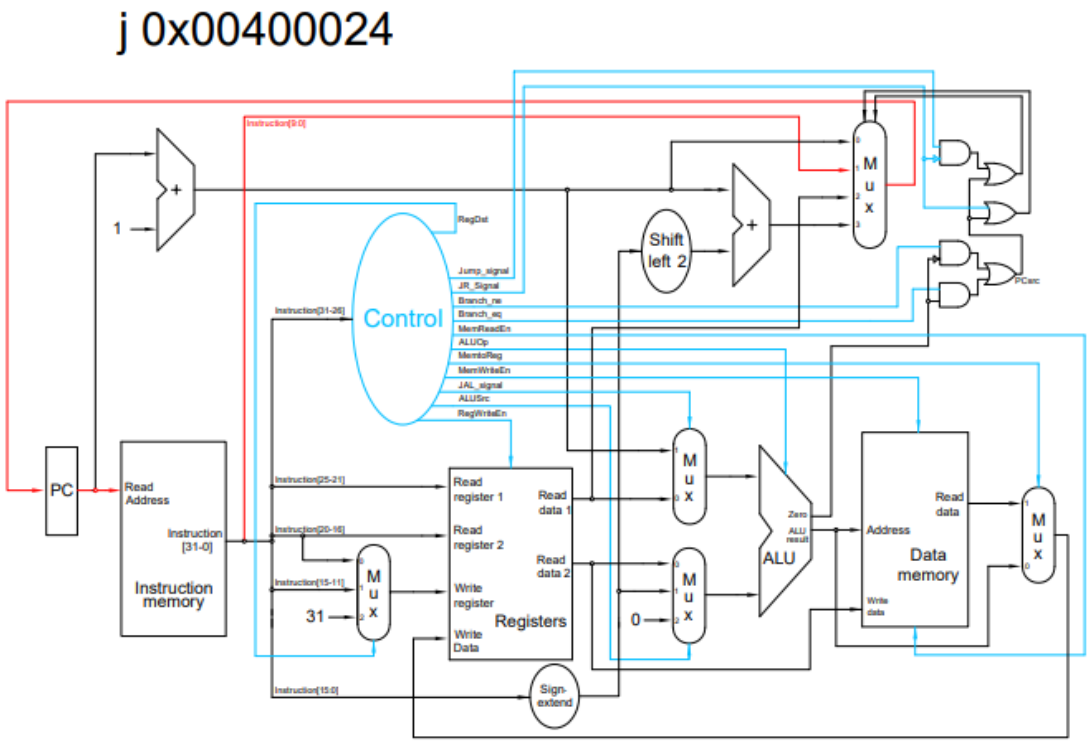
- **Example Instruction:** lw $t1, 4($t2)
- **Steps:**
  - Fetch the instruction from the instruction memory using the PC.
  - Decode the instruction to identify the opcode, base register, offset, and destination register.
  - Sign-extend the offset to 32 bits if necessary.
  - Add the base register value ($t2) to the offset (4) to compute the memory address.
  - Access the data memory to read the value at the computed address.
  - Write the retrieved value to the destination register ($t1).
  - Update the PC to point to the next instruction.

Lw $t1 , 4($t2)



## J-Type Instruction Flow

- **Example Instruction:** j 0x00400024
- **Steps:**
    1. Fetch the instruction from the instruction memory using the PC.
    2. Decode the instruction to extract the jump target address.
    3. Concatenate the upper bits of the current PC with the target address.
    4. Update the PC to the computed jump address.



# 4. ALU Design

The Arithmetic Logic Unit (ALU) is a core component of the single-cycle processor, responsible for performing arithmetic, logical, and shift operations as specified by the instructions. This section explores the design, operations, and testing methodology for the ALU.

### Overview of the ALU

The ALU is a critical component that executes operations required by various instructions in the processor. Its primary roles include:

- **Arithmetic operations**: Addition, subtraction, and comparison (e.g., set less than).
- **Logical operations**: AND, OR, and NOT.
- **Shift operations**: Shift left logical (SLL), shift right logical (SRL), and potentially others, depending on the instruction set.

By decoding control signals from the control unit, the ALU selects the appropriate operation to execute. It processes two inputs (operands) and generates one output, often accompanied by status flags like Zero or Overflow.

### ALU Design Approach

The ALU is designed using a modular approach to simplify implementation and testing. Its key components include:

- **Arithmetic Unit**: Includes an adder/subtractor to perform addition and subtraction. Subtraction is implemented by inverting the second operand and using the adder with a carry-in of 1.
- **Logical Unit**: Implements bitwise operations (AND, OR, NOR, XOR).
- **Shift Unit**: Performs shift operations (logical left shifts and logical right shifts).
- **Comparators**: Supports comparisons like set less than (SLT) and set greater than (SGT).
- **Multiplexers**: Select between the outputs of the arithmetic, logical, shift, and comparator units based on control signals.

### ALU Operations

The ALU supports the following operations:

| Control Signals | Operation | Description |
|---|---|---|
| **0000** | **ADD** | **Adds two inputs.** |
| **0001** | **SUB** | **Subtracts the second input from the first.** |
| **0010** | **AND** | **Bitwise AND of two inputs.** |
| **0011** | **OR** | **Bitwise OR of two inputs.** |
| **0100** | **NOR** | **Bitwise NOR of two inputs.** |
| **0101** | **XOR** | **Bitwise XOR of two inputs.** |
| **0110** | **SLT** | **Sets the output to 1 if the first input is less than the second; otherwise, 0.** |
| **0111** | **SLL** | **Shifts the first input left by the specified number of positions.** |
| **1000** | **SRL** | **Shifts the first input right by the specified number of positions.** |
| **1001** | **SGT** | **Sets the output to 1 if the first input is greater than the second; otherwise, 0.** |

The control unit generates a 4-bit control signal to select the appropriate operation. For example:

- 0000: The arithmetic unit is enabled to perform addition.
- `0110`: The comparator determines if the first operand is less than the second.

### Truth Table for ALU Operations

Below is the truth table mapping the control signals to the corresponding ALU operations and their behaviors:

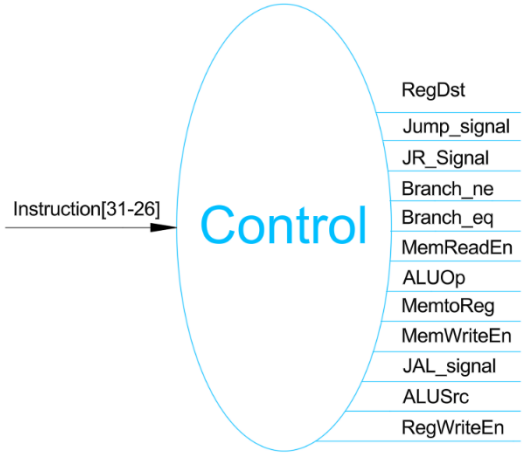
| ALU Control | Operation | Operand1 | Operand2 | Result | Status Flags |
|---|---|---|---|---|---|
| 0000 | ADD | A | B | A + B | Zero=1 if Result=0 |
| 0001 | SUB | A | B | A - B | Zero=1 if Result=0 |
| 0010 | AND | A | B | A AND B | Zero=1 if Result=0 |
| 0011 | OR | A | B | A OR B | Zero=1 if Result=0 |
| 0100 | NOR | A | B | NOT (A OR B) | Zero=1 if Result=0 |
| 0101 | XOR | A | B | A XOR B | Zero=1 if Result=0 |
| 0110 | SLT | A | B | 1 if A < B, else 0 | Zero=1 if Result=0 |
| 0111 | SLL | A | B | A shifted left by B[4:0] | Zero=1 if Result=0 |
| 1000 | SRL | A | B | A shifted right by B[4:0] | Zero=1 if Result=0 |
| 1001 | SGT | A | B | 1 if A > B, else 0 | Zero=1 if Result=0 |

## 5. Control Unit Design

The control unit is a fundamental component of the single-cycle processor, responsible for orchestrating the operation of the Datapath. It generates the necessary control signals to ensure that each instruction is executed correctly. This section explores the design and implementation of the control unit.

### Control Unit Overview

The control unit acts as the "brain" of the processor, directing the flow of data and the operation of components such as the ALU, memory, and registers. In a single-cycle processor, the control unit must generate all control signals within a single clock cycle to execute an instruction completely. These control signals dictate:




**Which operations the ALU performs.**

**The source and destination of data transfers.**

**Memory read and write operations.**

## The program counter (PC) update logic.

Due to its central role, the control unit directly impacts the correctness and efficiency of the processor's execution.

### Hardwired Control

The single-cycle processor typically employs hardwired control, where the control logic is implemented using combinational circuits. This approach ensures simplicity and speed but lacks flexibility compared to microprogrammed control. Hardwired control designs are based on the instruction format and the opcode field, which determine the specific control signals required for each instruction.

Key steps in designing a hardwired control unit include:

1. **Decoding the Instruction:**
    - The opcode and function code fields are extracted from the instruction.
    - These fields are used to determine the type of instruction (R-type, I-type, or J-type).
2. **Generating Control Signals:**
    - The decoded instruction drives the combinational logic that produces control signals.
    - For example, an add instruction will enable the ALU to perform addition, select the appropriate source registers, and write the result back to the destination register.
3. **Ensuring Signal Coordination:**
    - Control signals must be consistent and synchronized to avoid conflicts in the datapath.
4. **Inputs and Outputs of the Control Unit**

    **Inputs:**

    - **opcode: Specifies the type of instruction (6 bits).**
    - **funct: Used for further decoding in R-type instructions (6 bits).**

    **Outputs (control signals)**

    - **aluop: Specifies the ALU operation.**
    - **RegDst: Determines the destination register (used in R-type and JAL).**
    - **Branch: Signals a branch instruction.**
    - **MemReadEn and MemWriteEn: Control memory read and write operations.**
    - **MemtoReg: Specifies if data comes from memory or the ALU.**
    - **RegWriteEn: Enables register write.**

- **ALUSrc: Determines the ALU's second operand.**
- **ZERO, JAL_signal, Jump_signal, and JR_Signal: Control branching and jumping behavior.**

5. Control Signal Table

**The following table outlines the key control signals generated by the control unit for different instruction types:**

| Instruction Type | RegDst | ALUSrc | MemtoReg | RegWrite | MemRead | MemWrite | Branch_eq | Branch_ne | Jump | ALUOp |
|---|---|---|---|---|---|---|---|---|---|---|
| R-Type | 01 | 00 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Based on funct |
| ADDI | 00 | 01 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0000 |
| LW | 00 | 01 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0000 |
| SW | XX | 01 | X | 0 | 0 | 1 | 0 | 0 | 0 | 0000 |
| BEQ | XX | 00 | X | 0 | 0 | 0 | 1 | 0 | 0 | 0001 |
| JUMP | XX | XX | X | 0 | 0 | 0 | 0 | 0 | 1 | XXXX |

XX indicates "Don't Care" values, and X represents signals not used by the instruction.

This table shows how the various control signals behave for different instruction types, indicating when they are asserted or deserted to achieve the correct operation.

| Signal Name | Effect when Deasserted | Effect when Asserted |
|---|---|---|
| **RegDst** | **Default register destination (e.g., `rt`).** | **Selects specific destination registers (e.g., `rd` or `31` for JAL).** |
| **Branch_eq** | **No branch operation occurs.** | **Branch if equal.** |
| **Branch_ne** | **No branch operation occurs.** | **Branch if not equal.** |
| **MemReadEn** | **Memory is not read.** | **Enables memory read operation.** |
| **MemtoReg** | **ALU result written to register.** | **Memory data is written to register.** |
| **MemWriteEn** | **No memory write operation.** | **Enables memory write operation.** |
| **RegWriteEn** | **No data is written to the register file.** | **Enables writing data into the register file.** |
| **ALUSrc** | **Second ALU operand from register file.** | **Second ALU operand is an immediate or shift amount.** |
| **ZERO** | **Result of ALU comparison ignored.** | **Used for zero comparison (e.g., BEQ or BNE instructions).** |
| **JAL_signal** | **Normal operation, no jump.** | **Jump and link operation (saves return address to register 31).** |
| **Jump_signal** | **Normal PC progression.** | **PC is set to jump target address.** |
| **JR_Signal** | **Normal PC progression.** | **PC is set to the address in the register specified (e.g., for JR instruction).** |
| **aluop** | **Default ALU operation.** | **Specifies the ALU operation (e.g., ADD, SUB, OR).** |

## Pipelined Processor Design and Optimization

The single-cycle processor, while simple in design, is inherently limited by its execution model, where each instruction is executed in a single clock cycle. This results in significant inefficiencies, as the clock period must be long enough to accommodate the slowest instruction. To address these inefficiencies, a pipelined architecture was proposed, enabling overlapping execution of multiple instructions to increase throughput.

This section explores the transition from single-cycle to pipelined design, highlighting the challenges encountered and the iterative optimization process. By leveraging pipelining, we aim to balance the resolution of hazards with maintaining a high clock frequency, ultimately achieving a design that maximizes throughput and performance efficiency.

### 1. Overview of pipelining

Fundamentally, the typical life cycle of a processor consists of **fetching**, **decoding**, and finally **executing** instructions.

In a single-cycle processor, the entire life cycle of an instruction is completed within a single clock cycle. In other words, data flows from the program counter (PC) register through a series of combinational logic blocks and eventually reaches the register file—all in one clock period. hence the name Single Cycle.

However, a significant drawback of single-cycle processors is their inability to operate at high clock frequencies. Why?

The answer lies in **setup time constraints**.

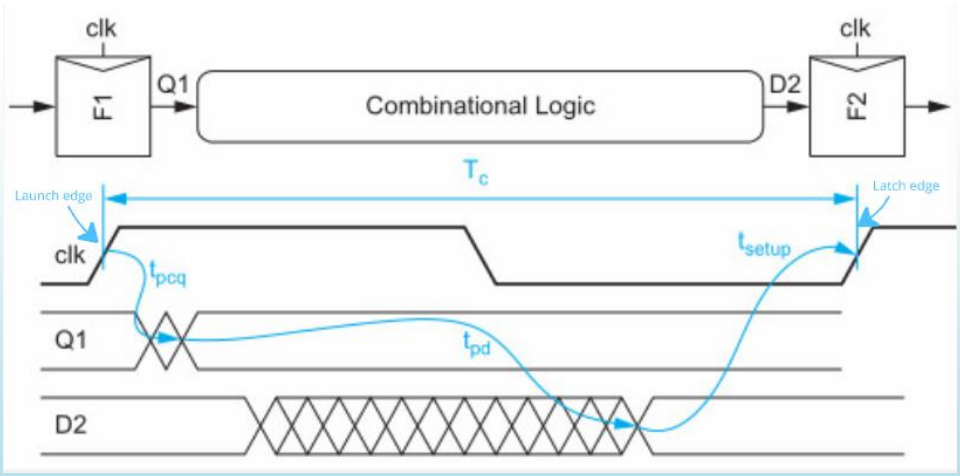



Figure 1 : Setup time

Setup time is defined as the minimum duration for which input data must remain stable **before** the triggering edge of the clock. This constraint ensures that data propagates correctly between registers.

In sequential circuits, this occurs between two registers over two subsequent clock cycles. In **Figure 1**, the rising edge of the first clock cycle is called the **launch edge**, which triggers the data to propagate from the source register. The rising edge of the second clock cycle is called the **latch edge**, which captures the data into the destination register.

For the system to function correctly, the data triggered at the launch edge must propagate through the combinational logic and reach the destination register before the latch edge occurs. If the data fails to arrive in time, a **setup time violation** occurs, leading to a metastable state where the data captured in the destination register is unpredictable, potentially causing errors in the system.

Digital processors consist of extensive sequential and combinational logic. The longest delay path for data to propagate between two registers is referred to as the **critical path**. This path determines the maximum clock frequency (Fmax) the design can operate at safely. The clock period must be long enough to accommodate the delay of the critical path.

Now In single-cycle processors, the critical path encompasses the entire instruction lifecycle: fetch, decode, execute, memory access, and writeback. For example, in **Figure 1**, if we assume the PC register is F1 and a register in the register file is F2, the combinational logic between them represents the critical path. Since the full lifecycle of an instruction must be completed within one clock cycle, the clock period must be long enough to accommodate the slowest instruction. This constraint inherently limits the clock frequency, reducing the overall performance of single-cycle designs.

Setup time constraints must be carefully respected when designing digital systems to ensure reliable and predictable operation.

In addition to setup time, there is another timing constraint known as **hold time**. Hold time is the minimum duration for which input data must remain stable *after* the triggering clock edge to ensure it is securely captured by the destination register. Unlike setup time, which directly determines the maximum clock frequency a processor can operate at, hold time is less of a concern in most designs because violations typically arise from excessively short delays, not from long critical paths. When hold time violations do occur, they are often addressed by adding small buffers to delay signals. While hold time is an important consideration for ensuring correct operation, our primary focus remains on setup time, as it directly limits the processor's performance and clock frequency.

To overcome the limitations of single-cycle processors, MIPS processors adopt a **pipelined architecture** that divides the instruction lifecycle into five distinct stages: **Fetch (IF)**, **Decode (ID)**, **Execute (EX)**, **Memory Access (MEM)**, and **Writeback (WB)** as illustrated in Figure2. Instead of executing one instruction at a time, the pipeline enables overlapping execution, where each stage processes a different instruction simultaneously during every clock cycle, as depicted in **Figure 3**.

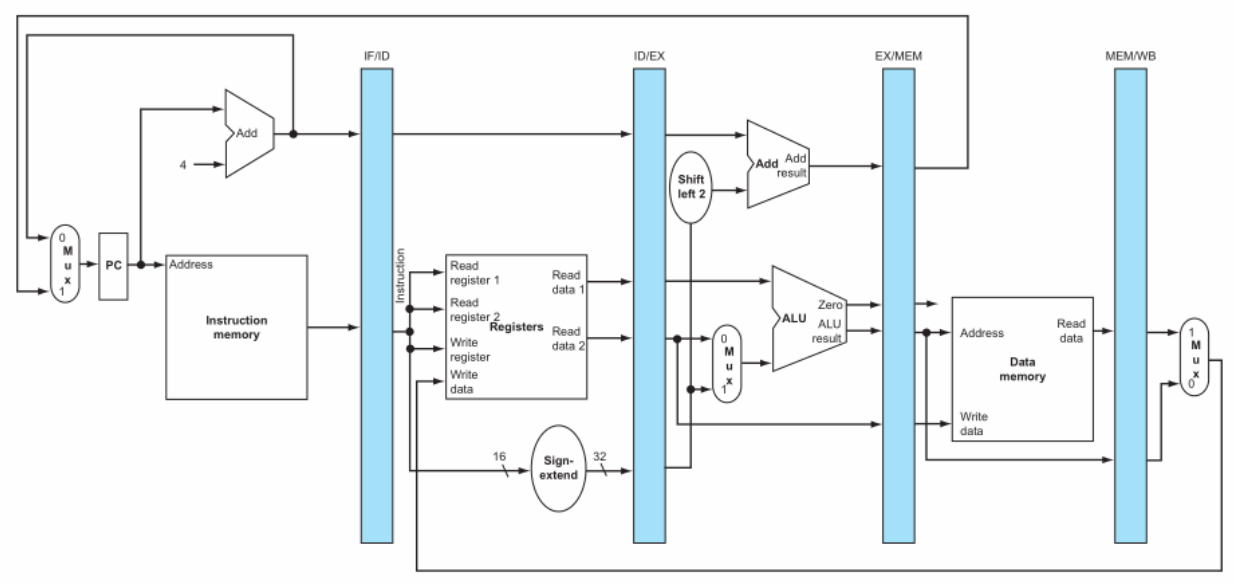



Figure 2 : Pipelined Datapath

This design significantly improves instruction throughput by ensuring that the processor is continuously active. For instance, while one instruction is being decoded, another instruction can be fetched, and yet another can be executed. The **segmentation of the instruction lifecycle** into these discrete stages reduces the critical path, allowing for much higher clock frequencies compared to a single-cycle processor, where the clock period must accommodate the slowest instruction.

In the MIPS pipeline:

- The **Fetch (IF)** stage retrieves the instruction from memory using the **Program Counter (PC)**, which increments sequentially, or updates based on branch decisions.
- The **Decode (ID)** stage interprets the opcode, reads from the **register file**, and performs sign extension if necessary.
- The **Execute (EX)** stage carries out arithmetic or logical operations using the **ALU** or calculates memory addresses for load and store instructions.
- The **Memory Access (MEM)** stage accesses data memory for loads and stores.
- The **Writeback (WB)** stage updates the destination register with results, completing the instruction lifecycle.

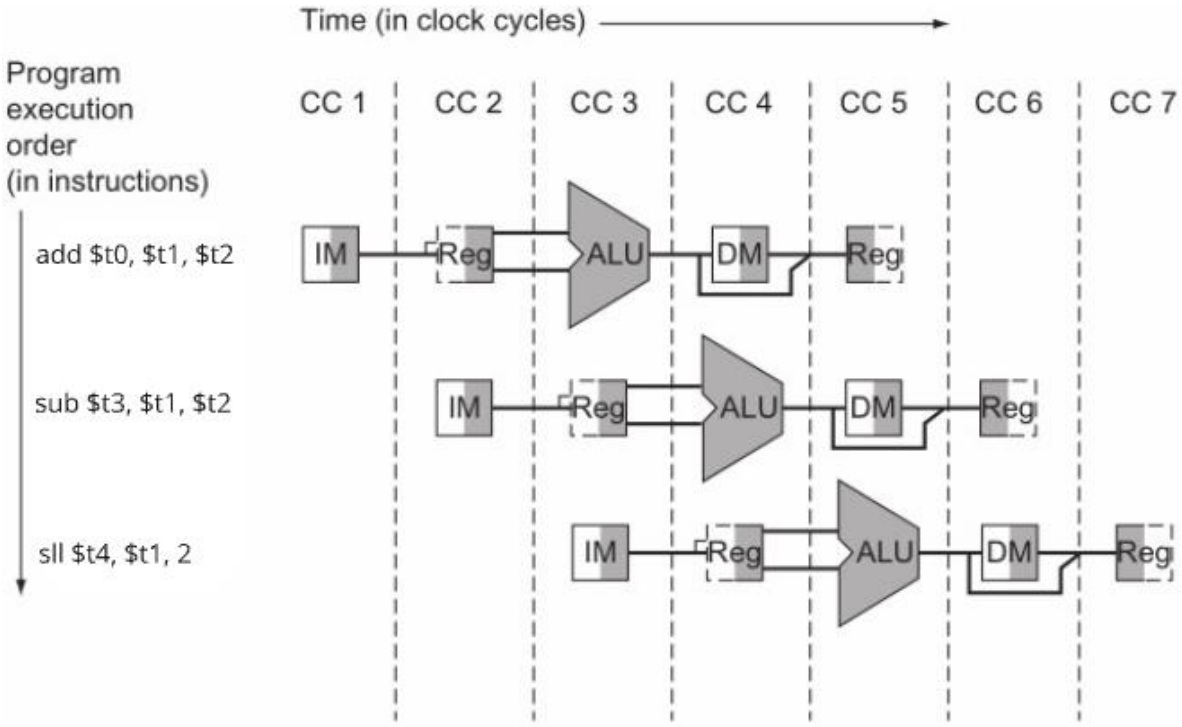



Figure 3 :Program execution

As shown in Figure 2, the MIPS pipelined data path contains inter-stage registers (e.g., **IF/ID**, **ID/EX**, etc.) that hold intermediate values passed between pipeline stages. These registers ensure that instructions move smoothly through the pipeline without overwriting or losing critical data.

This pipelined approach transforms the instruction flow into an assembly-line process, as shown in **Figure 3**. For example:

- In **Clock Cycle 3 (CC3)**:
    - The first instruction (add) is in the **Execute** stage.
    - The second instruction (sub) is in the **Decode** stage.
    - The third instruction (sll) is in the **Fetch** stage.

By overlapping these stages, the processor achieves an effective instruction throughput of one instruction per clock cycle once the pipeline is fully filled.

However, this improvement in throughput comes at a cost. The overlapping execution introduces challenges, such as **data hazards**, **control hazards**, and **structural hazards**, which can disrupt the smooth flow of instructions and complicate design. Addressing these issues requires mechanisms like forwarding, stalling, and hazard detection.

In Summary, the MIPS pipelined architecture remains a foundational design that enhances both performance and efficiency by balancing higher clock frequencies, by shortening the paths of which data moves from a register to another, with parallel instruction processing. The following sections will discuss

these challenges and their solutions, exploring how the architecture ensures proper execution while maximizing efficiency.

## Hazards

To ensure the efficient operation of a pipelined processor like MIPS, several challenges arise due to hazards. Hazards occur when the instructions in the pipeline interact in ways that can lead to incorrect or delayed execution. These hazards must be managed carefully to maintain the performance gains pipelining offers. The primary types of hazards in the MIPS pipeline are **data hazards**, **control hazards**, and **structural hazards**.

### Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. In other words, when a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

There are three main types of data hazards:

- **Read After Write (RAW)**: This is the most common type, where an instruction needs to read a register value that has not yet been written back by the previous instruction.
- **Write After Write (WAW)**: This occurs when two instructions write to the same register, and the order of writing is crucial.
- **Write After Read (WAR)**: This occurs when a write to a register happens before the previous instruction has read it, potentially causing incorrect data to be written.

Regarding the RAW data hazard, Consider the following instruction sequence:

```
add $s0, $t1, $t2

sub $s1 , $s0, $t3
```

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that the subtract instruction would have to wait in the decode stage till the add instructions writes to the register file to be able to read the correct data, thus waste three clock cycles in the pipeline.

Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard, By Adding extra hardware to retrieve the data early from later stages is called **forwarding** or **bypassing**.

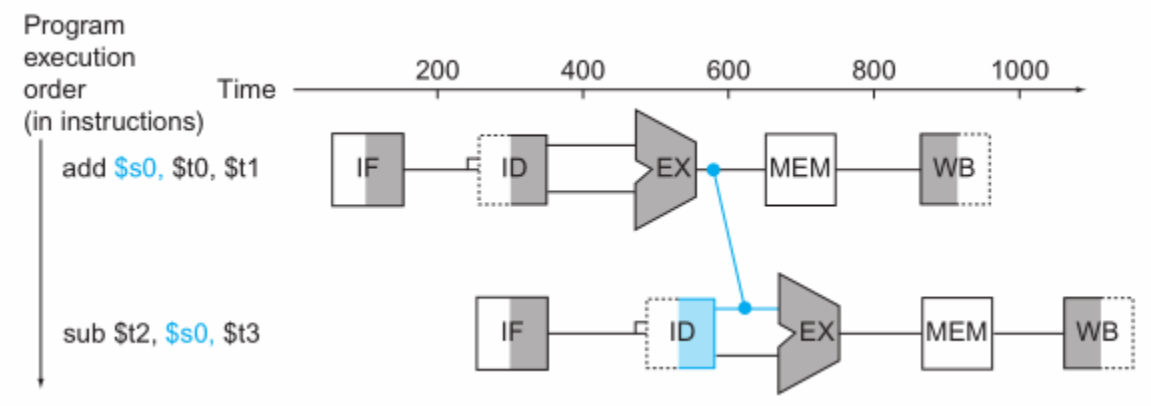



Figure 4 : Forwarding (Bypassing)

In MIPS, forwarding is used to solve RAW hazards. Forwarding allows the output of one stage, to be sent directly to a previous stage that requires it. This avoids the need to wait until the writeback stage.

Forwarding can be enhanced from this simple case, because in a real program the pipeline would be filled with instructions in each stage, there is a high chance that an instruction in the execute stage requires data from the memory or write back stage. Figure 5 shows the benefits of forwarding if applied in a way that

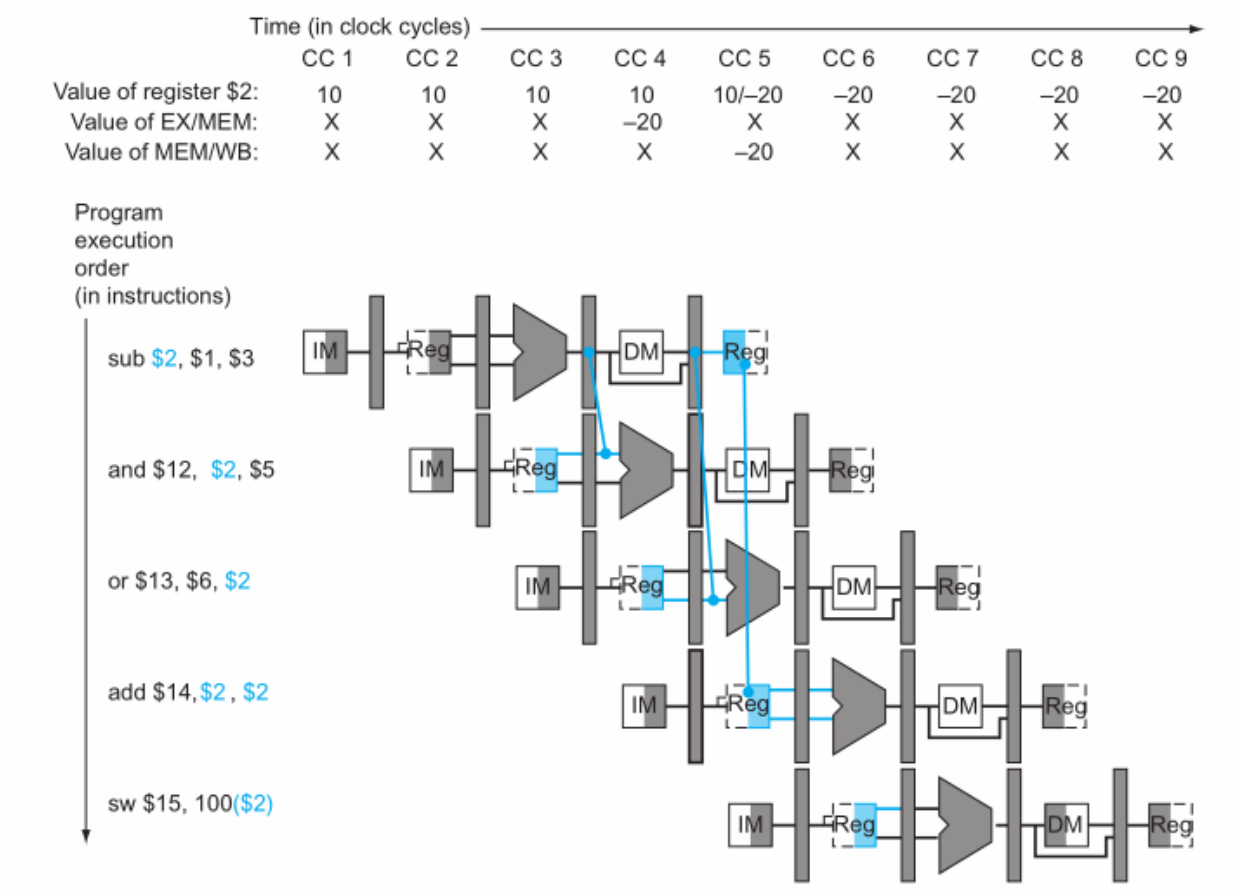



Figure 5 : Enhanced forwarding

we are able to propagate the data back the ALU to perform the necessary operations with the correct data.

Notice how it wasn't necessary to stall the pipeline while keeping the outcome correct. However to be able to propagate the data to an earlier stage requires more hardware to spot when exactly forwarding must happen. Detecting when to forward data can be done through the **Forwarding Unit**, its main job is to compare destination register index from the memory stage or the write back stage with the registers rs and rt in the execute stage, as they hold the operand values that are entering the ALU, and if there is a match then that means a RAW dependency is detected and data must be forwarded from the Mem or WB stages or even both, and that depends on the dependency itself within the sequence of the instructions.

While this explanation regarding the Forwarding Unit is brief, it describes how it works and most importantly when. In this report the aim was to find the optimal design for a pipelined processor, so different designs of the forwarding logic was implemented, some of which resolve all the dependencies between the stages and some that doesn't, and the reason is that *in principle* all the dependencies can be resolved in a pipelined processor, but that comes at a cost of increasing the critical path which decreases the maximum frequency. Because the one main reasons of pipelining was to split the stages in a way that the processor can operate at higher frequencies. In section 1.2.4 the Forwarding implementation will be described in more detail regarding the pros and cons for each design.

The in-order execution of the pipeline processor and the simple register access model in this report discards the problems that can arise from the WAW and WAR data hazards, but they are a concern if the execution model was out of order and if the register file access was more complex.

### The trade-offs

While it is true that there is great benefit from pipelining, in terms of throughput increase, Problems arise as described in the previous sections.

The question is : Can we solve all the problems in pipelining, and the answer is *in principle* yes, we can. But there is a limit. Solving dependencies by Bypassing (Forwarding) requires stretching out the data away from its typical path (Between two Pipe Registers) so we can send it to a previous stage.
And this means that We are violating the structure of our pipelining model and reducing its benefit.
Remember that by pipelining we are shortening the critical path in our Single Cycle processor, and we are dividing that path by number of stages.
So if we have 5 stage pipelined processor our critical path should be divided into 5 paths. Which means we are capable of achieving higher frequencies for our clock since we are making the critical path shorter, by reducing the setup time required between two registers.

Now when we Bypass data from a stage to a previous stage, we increase this critical path, meaning that by forwarding we reduce the overall frequency our design can run with.

And this Is the trade off, to solve dependencies but at the cost of frequency. And to figure out an optimal design multiple evaluations need to be made to find the balance.

The methodology of approaching this problem statement is as follows, trying different designs and evaluating their Fmax, and throughput by running benchmarks, to eventually conclude what kind of

problems we should solve and what not.
because a design that solves 90% of the problems in Pipelining can run at approximately 50Mhz, but a design that solves 50% can run at 80Mhz.

And a balance between the two is what we are after to reach the optimal design that has the most throughput, Because this is what really matters, no consumer cares of what their microprocessors look like or how it is designed, a consumer cares about how fast their processors can execute instructions in a given time frame.

# 1. Datapath Design

The Datapath in a 5-stage pipeline processor is the collection of hardware components and the interconnections between them that are responsible for the actual data processing. It consists of registers, arithmetic units, multiplexers, and memory, which work together to execute instructions as they move through the pipeline.

### Overview of the Datapath

The Datapath in a 5-stage pipeline processor is designed to handle different types of instructions, such as arithmetic operations, data memory operations, and control operations. The major components of the Datapath are the Program Counter (PC), instruction memory, register file, ALU (Arithmetic Logic Unit), data memory, and the pipeline registers that store intermediate results between stages.

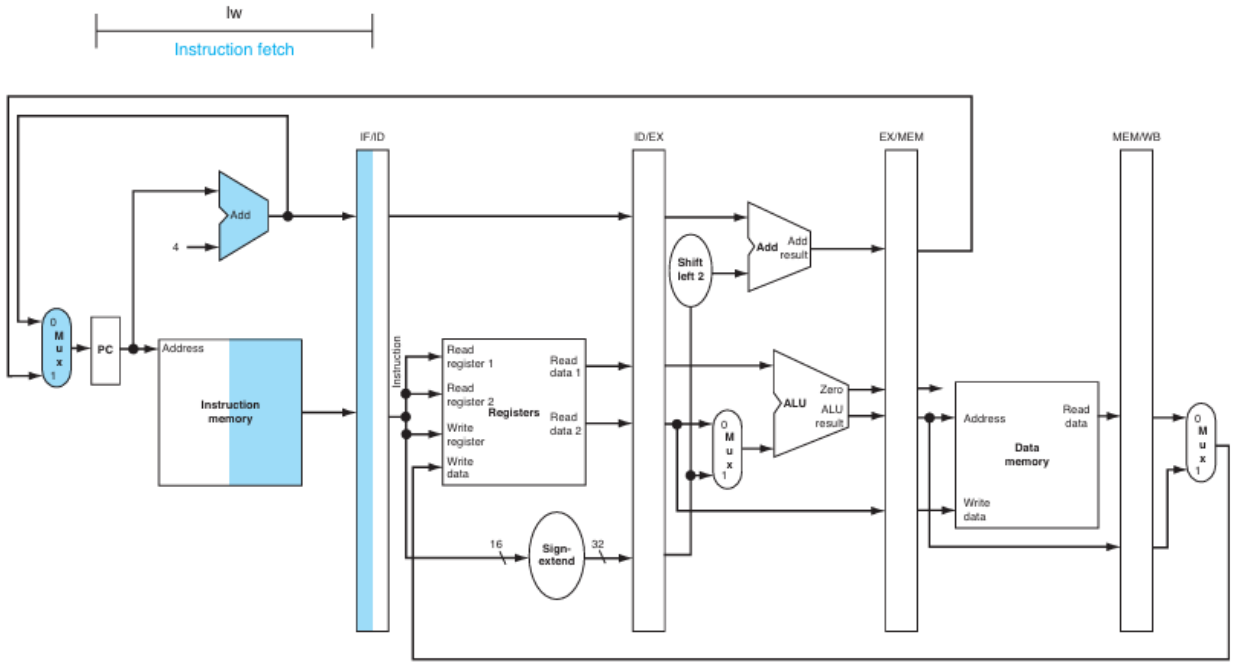
The primary function of the Datapath is to carry out the operations specified by the instruction in each stage of the pipeline. It does so by connecting these components and allowing data to flow through them as each instruction is processed in a given stage. The data path must be carefully designed to manage data forwarding, hazards, and synchronization between the stages.

### Design of Each Stage

#### Instruction Fetch (IF):

In the first stage, the instruction is fetched from memory using the address in the Program Counter (PC). The fetched instruction is placed in the IF/ID pipeline register, which stores the instruction for the next cycle. The PC is incremented by 4 to point to the next instruction, and this incremented value is also saved in the IF/ID pipeline register. This is important for instructions like branch (e.g., beq), which may need the PC value to compute the branch target.

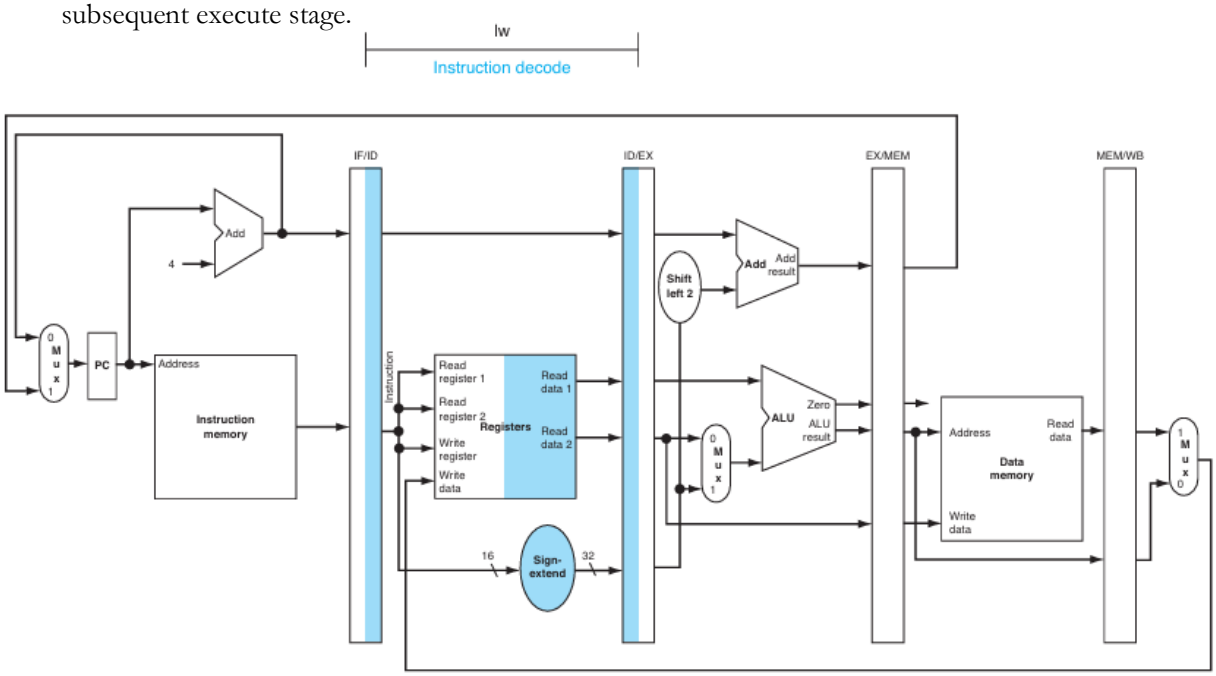
Since the processor does not know the type of instruction being fetched, it must be prepared to handle any type of instruction. Therefore, the information, including the incremented PC, is passed down the pipeline to later stages where it might be needed.




## Instruction Decode and Register File Read (ID):

The instruction stored in the IF/ID register is passed to the ID stage. In this stage, the instruction is decoded, and the relevant register addresses are identified. The 16-bit immediate field from the instruction is sign-extended to 32 bits. This extended immediate value and the register addresses are stored in the ID/EX pipeline register, along with the incremented PC address.

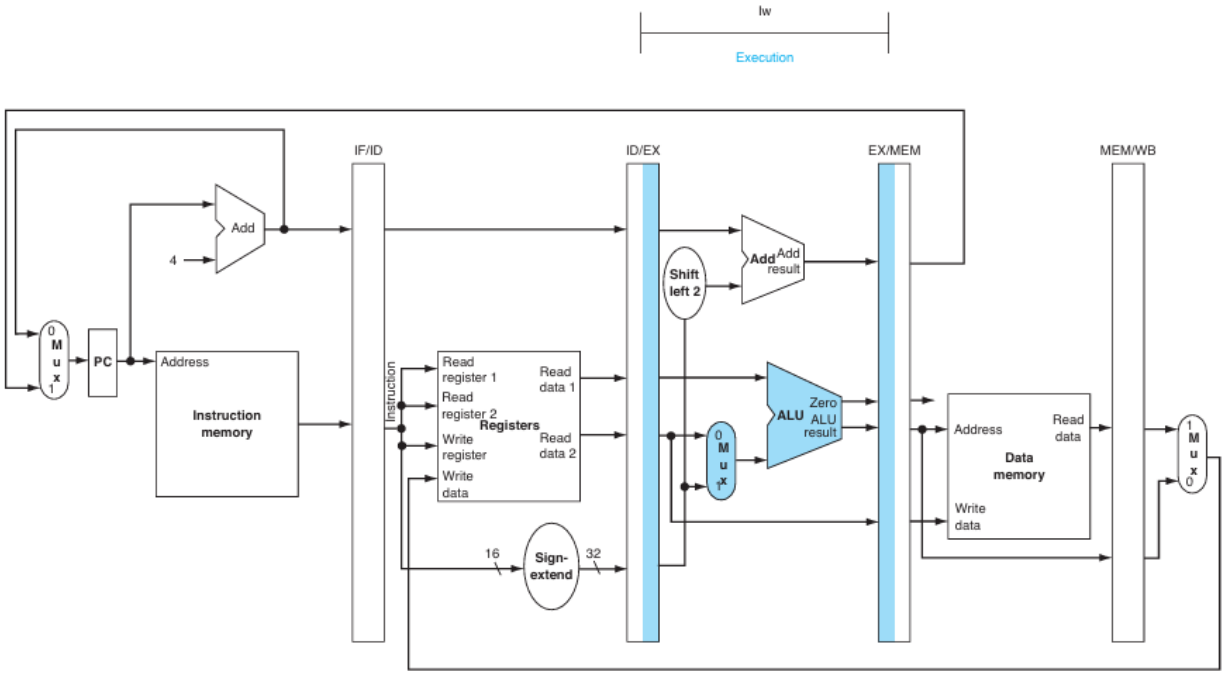
The register file is read to retrieve the operands needed for execution. These operands, along with other relevant data (such as the immediate value and PC), are prepared for use in the subsequent execute stage.



Execute (EX) or Address Calculation:

In the EX stage, the arithmetic or logical operation is performed. For example, in a load instruction, the ALU reads the contents of the register file and the sign-extended immediate value, then adds them to calculate the memory address. This calculated address, along with any other results, is stored in the EX/MEM pipeline register.

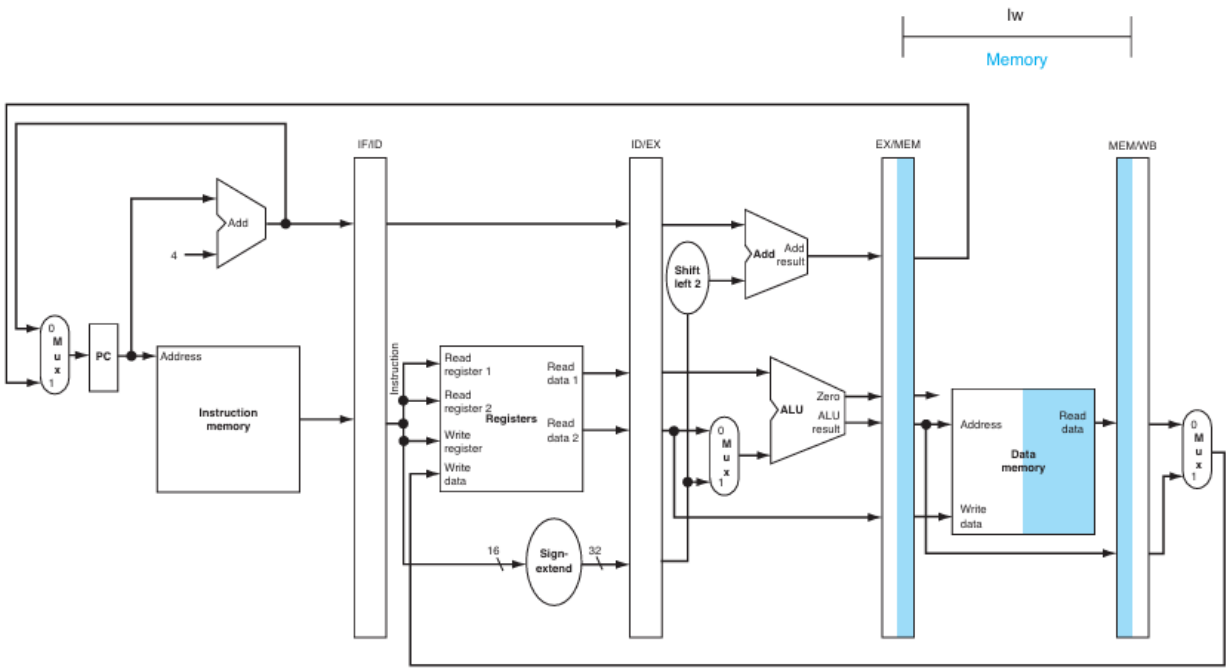
The EX stage is crucial for computing results and memory addresses, preparing the processor for the next step in the pipeline.




**Memory Access (MEM):**

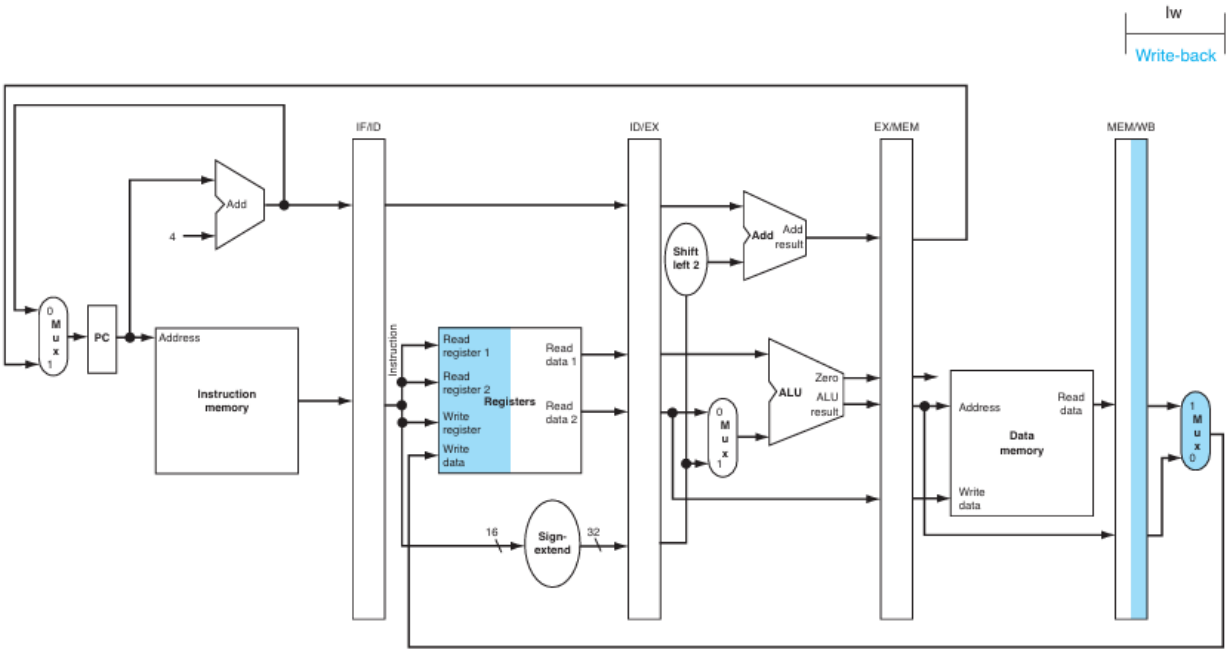
In the MEM stage, the processor accesses data memory. For a load instruction, the calculated address from the EX stage is used to read data from memory. The data fetched from memory is then stored in the MEM/WB pipeline register.

For store instructions, the data to be written to memory is placed in this stage, ready to be written to the data memory.

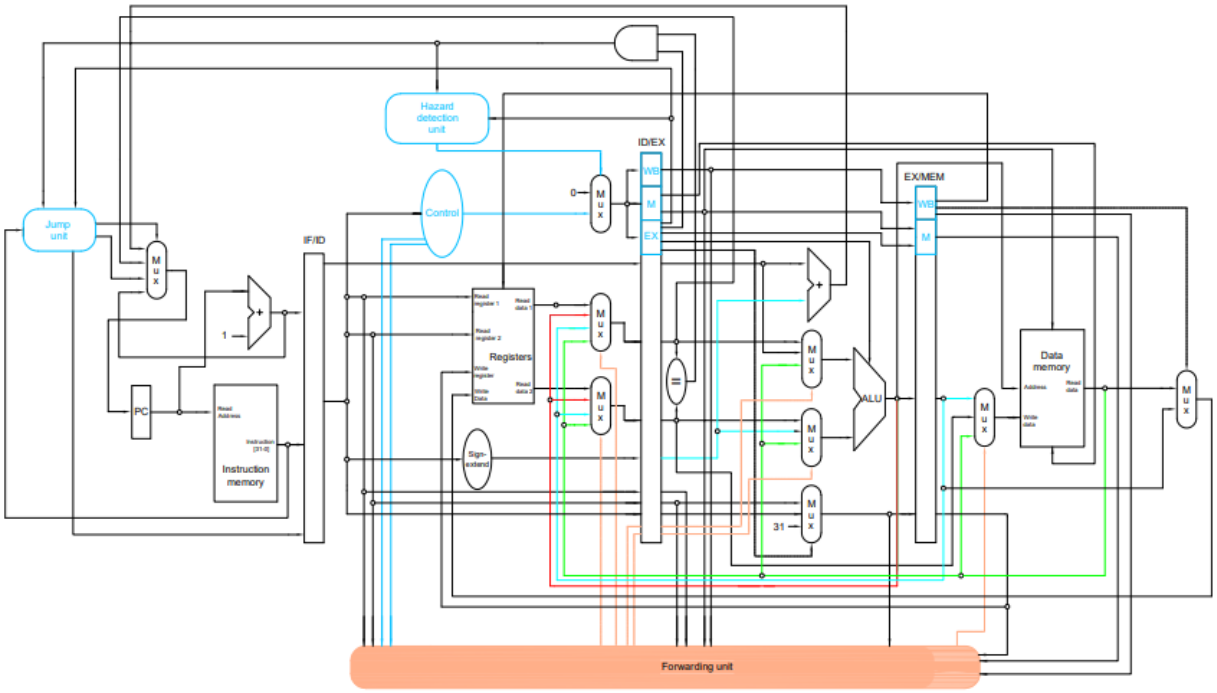



**Write-Back (WB):**

The final stage involves writing the results back into the register file. In the WB stage, the processor reads the data stored in the MEM/WB pipeline register (either the result from memory or the result from the ALU operation) and writes it to the appropriate register in the register file. This completes the instruction's execution.



## The Modified DataPath




### Units

As the design of the 5-stage pipeline processor evolves, new units are integrated to enhance its functionality and improve performance. These additional units are introduced to address various challenges in pipeline processing, such as handling control hazards, optimizing instruction flow, and ensuring efficient data forwarding. The inclusion of these new components aims to streamline the processor's operation by reducing pipeline stalls, improving the handling of complex instruction types like jumps and branches, and enabling better overall throughput. The following sections will detail the new units integrated into the design and explain their contributions to improving pipeline efficiency.

### Jump Unit

#### Overview

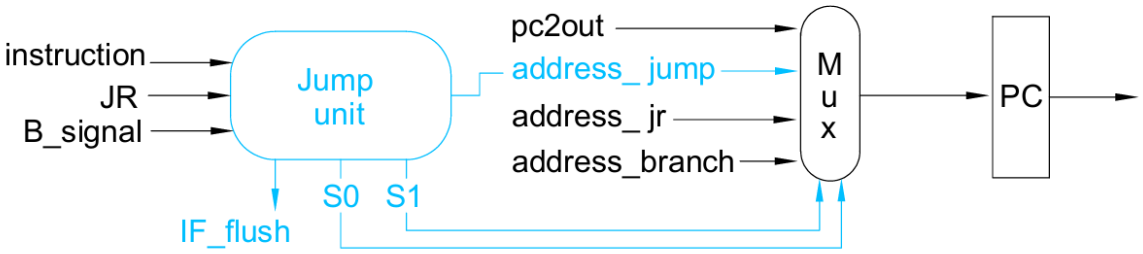
The Jump Unit is an integral part of the pipeline architecture, responsible for handling jump and branch instructions efficiently. By generating precise control signals, it ensures the pipeline correctly handles changes in control flow.

#### Placement of the Jump Unit

Normally, the Jump Unit resides in the **Execute Stage**. However, this placement introduces a delay of two cycles when handling jump instructions, as the Fetch Stage continues to process instructions sequentially until the jump address is resolved.

To eliminate these delays, the Jump Unit is moved to the **Fetch Stage**, allowing the pipeline to immediately redirect execution to the correct address. This optimization

prevents the two bubbles that would otherwise occur and significantly improves overall performance. By resolving jump instructions earlier, the processor maintains higher instruction throughput and minimizes control flow disruptions.




### Control Signal Generation

The control signals generated by the Jump Unit are critical for managing the program counter (PC) updates and maintaining pipeline integrity during control flow changes. These signals include the selector signals (**S0**, **S1**) and the flush signal (**IF flush**).

#### Selector Signal Logic

The selector signals, **S0** and **S1**, are used by the multiplexer in the Fetch Stage to determine the source of the next PC value (PC_next). The logic for these signals is as follows:

**Logic for S0:** $S0 = B\_signal \mid (\sim JR \,\&\, J)$

- **B_signal:** Indicates that a branch instruction is taken (e.g., BEQ when the condition is met).
- **JR:** Indicates a register-based jump instruction.
- **J:** Indicates an unconditional jump instruction.
- **S0:** is asserted for either a branch instruction or an unconditional jump (as long as it is not a register-based jump).

**Logic for S1:**

$$S1 = JR \mid B\_signal$$

**S1** is asserted when the next PC value should be determined by either a register-based jump (JR) or a branch instruction (B_signal).

### Flush Signal Logic

The flush signal, **IF_flush**, ensures that invalid instructions do not propagate through the pipeline during control flow changes. It is triggered by the Jump Unit in the following scenarios:

- A branch is taken (B_signal=1).
- A jump instruction is detected (JR=1or J=1).

The logic for the flush signal is as follows:

$$IF\_flush = JR \mid B\_signal$$

When **IF_flush** is asserted, the pipeline's IF/ID register is cleared, discarding incorrect instructions. This ensures the pipeline fetches the correct instruction in the next cycle.

## Changes to the Multiplexer Design

In the basic pipeline processor design, a **2-to-1 multiplexer** is typically used to select the next program counter (PC_next) value, allowing the processor to switch between sequential instruction fetching and a single alternate source, such as a branch target address. However, the inclusion of additional control flow instructions, such as jumps and register-based jumps, necessitated an upgrade to a **4-to-1 multiplexer** to accommodate multiple possible sources for PC_next.

- **The 4-to-1 Multiplexer Design**

In the updated design, the **4-to-1 multiplexer** handles four possible sources for the next program counter (PC_next) value, each corresponding to a specific control flow scenario.

- **Inputs:**
  - **IN0 (pc2out):** The address of the next sequential instruction (PC + 4 in a typical implementation).
  - **IN1 (address_jump):** The target address for unconditional jump instructions.
  - **IN2 (address_jr):** The target address for register-based jump instructions (e.g., JR).
  - **IN3 (address_branch):** The target address for branch instructions.
- **Outputs:**
  - **OUT (pc_next):** The selected address that determines the next instruction to fetch.
- **Selector Signals:**
  - **S0 and S1:** Control signals generated by the Jump Unit that determine the source of the next PC value.

- **Selection Logic**

  The 4-to-1 multiplexer uses the selector signals (**S0** and **S1**) to choose among the four input sources. The selection is as follows:

| S1 | S0 | Selected Input | Source Type |
|---|---|---|---|
| 0 | 0 | IN0 | Sequential instruction (PC + 1) |
| 0 | 1 | IN1 | Unconditional jump address |
| 1 | 0 | IN2 | Register-based jump address (JR) |
| 1 | 1 | IN3 | Branch target address |

- **Rationale for the Design Change**

The original 2-to-1 multiplexer design only supported switching between sequential execution and one alternate source, such as a branch address. However, with the integration of additional control flow instructions (e.g., unconditional jumps and JR), a more flexible multiplexer was required. The new **4-to-1 multiplexer** ensures that the pipeline can handle multiple types of control flow changes efficiently, including:

1. **Sequential Execution:** When no control flow instruction is encountered, the PC advances to the next instruction sequentially (PC+1).
2. **Unconditional Jumps:** For instructions such as J and JAL, the PC is updated with the calculated jump address.
3. **Register-Based Jumps:** For JR instructions, the PC is updated with the address stored in a register.
4. **Branches:** For conditional branch instructions (e.g., BEQ), the PC is updated with the branch target address when the branch condition is satisfied.

The modifications in the Fetch Stage, including the integration of the Jump Unit and the replacement of the 2-to-1 multiplexer with a 4-to-1 multiplexer, ensure that the pipeline can efficiently handle various control flow instructions. By resolving jump and branch targets early, the Fetch Stage eliminates unnecessary stalls and reduces pipeline bubbles, significantly improving overall performance. These enhancements allow the processor to maintain high throughput while preparing accurate instruction addresses for subsequent stages. With these optimizations, the Fetch Stage serves as a robust foundation for the smooth operation of the pipeline, seamlessly passing control and data to the Decode Stage for further processing.

## Data Hazards in the Decode Stage

Data hazards occur when instructions in the pipeline depend on the results of prior instructions that have not yet completed their execution. This dependency can cause incorrect data to be forwarded or delays in instruction processing, potentially stalling the pipeline. To address this, hazard detection mechanisms are essential in the Decode Stage to maintain pipeline integrity while minimizing performance degradation.

### Hazard Detection Unit

The Hazard Detection Unit is responsible for identifying scenarios where data hazards may arise and resolving them by controlling the flow of instructions. Key signals managed by this unit include:

- **NopSel (No-Operation Selector):** This signal determines inserting a no-operation (NOP) into the pipeline.

The **NopSel** signal is generated using the following logic:

$$NopSel = B\_taken\_signal \mid JR\_signal$$

- **B_taken_signal:** Indicates that a branch instruction is taken.
- **JR_signal:** Indicates a register-based jump instruction.

When either of these conditions is true, a stall is introduced to ensure the pipeline fetches the correct instruction and maintains data integrity.

### Forwarding Mechanism in Decode Stage

In the Decode Stage, data forwarding is employed to minimize the delays caused by data hazards. Instead of waiting for an instruction to complete execution, the pipeline forwards the required data from later stages back to the Decode Stage using a 4-to-1 multiplexer.

**Note:** The Forwarding Unit, which plays a critical role in implementing this mechanism, will be discussed in detail in a later section.

### Handling Data Dependencies

1. **Read After Write (RAW) Hazards:** When an instruction requires a value that a previous instruction is yet to write to a register, forwarding or stalling ensures the correct value is used.
2. **Control Hazards:** For branch or jump instructions, the Decode Stage ensures that control signals generated by the Control Unit accurately direct the pipeline, avoiding incorrect instruction fetches.

By incorporating these mechanisms, the Decode Stage mitigates the effects of data hazards, allowing the pipeline to operate smoothly with minimal disruptions. This ensures that even in the presence of dependencies, instructions can flow through the pipeline efficiently.

## Multiplexers in the Decode Stage

In a pipelined processor, multiplexers are critical components for managing hazards and directing data flow. In the Decode Stage, three multiplexers have been adapted to address specific needs, considering an enhanced design approach inspired by the fetch stage multiplexer changes.

### Control Unit Signal Multiplexer (MUX2_1)

This multiplexer ensures correct hazard management by introducing flexibility similar to the fetch stage's upgraded 4-to-1 multiplexer logic. The inclusion of additional control signals allows the handling of complex scenarios such as control-dependent instructions (e.g., branches and jumps).

**Inputs:**

- control_unit_signal: Control signals generated during normal operation.
- 14'h0: A default "no-operation" (NOP) value for stalling the pipeline.

**Selector Signal:**

- NopSel is calculated as:

$$NopSel = JR\_signal \mid B\_taken\_signal$$

**Output:**

- If NopSel=1, the output is 14'h0, halting further pipeline progress for hazard resolution.
- If NopSel=0, the multiplexer forwards the control signals to proceed with normal operation.

This updated design ensures the pipeline handles scenarios requiring a control override while maintaining minimal complexity.

### Forwarding Multiplexer 1 (MUX4_1 for Data1)

Inspired by the fetch stage's shift to a 4-to-1 multiplexer, this component selects the correct value for the first source operand (Data1) to resolve RAW hazards.

**Inputs:**

- data1: Original register data.
- Alu Result: ALU result from the Execute Stage.
- Alu-Result-Mem: ALU result from the Memory Stage.
- Data-MEM: Data read from memory in the Memory Stage.

**Selector Signals:**

- The ForwardA[1:0] signal determines the input selection:

| ForwardA[1:0] | Input Selected |
|---|---|
| 00 | data1 (default) |
| 01 | Alu-Result |
| 10 | Alu-Result-Mem |
| 11 | Data-MEM |

**Output:**
The selected data ensures correct forwarding of operands, similar to how the fetch stage multiplexer handles multiple sources for the next program counter.

**Note:** Forwarding mechanisms and their implementation will be explained in detail in a later section.

**Forwarding Multiplexer 2 (MUX4_1 for Data2)**

This multiplexer operates identically to Forwarding Multiplexer 1 but resolves hazards for the second source operand (Data2) using ForwardB[1:0] signals.

**Inputs:**

- Identical to those of Forwarding Multiplexer 1.

**Selector Signals:**

- The selection is based on ForwardB[1:0], with the same logic applied:

| ForwardB[1:0] | Input Selected |
|---|---|
| **00** | Data2 (default) |
| **01** | AluResult |
| **10** | AluResult_Mem |
| **11** | Data_MEM |

**Output:**
Forwarded data to the second operand of the Decode Stage.

**Note:** Forwarding mechanisms and their implementation will be elaborated upon in a dedicated section.

**Integration of Multiplexers in the Pipeline**

The enhancements to the Decode Stage multiplexers draw parallels to the fetch stage's updated 4-to-1 multiplexer, ensuring robust functionality in the following ways:

1. **Control Management with MUX2_1:**
    - Overcomes hazards by dynamically selecting NOP signals, akin to how the fetch stage multiplexer dynamically selects the next PC source.
2. **Data Forwarding with MUX4_1 Units:**
    - Provides seamless forwarding of data to operands, reducing pipeline stalls while leveraging multi-source input selection similar to the fetch stage's PC multiplexer design.

**Equations Summary**

**Control MUX:**

$$\boldsymbol{Output} = \begin{cases} \boldsymbol{14'h0} & \boldsymbol{if\ NopSel\ =\ 1} \\ \boldsymbol{Control\ Signals} & \boldsymbol{if\ NopSel\ =\ 0} \end{cases}$$

**Forwarding MUXs (1 and 2):**

$$\boldsymbol{Output} = \begin{cases} \boldsymbol{data1\ or\ data2} & \boldsymbol{if\ Selection\ =\ 00} \\ \boldsymbol{AluResult} & \boldsymbol{if\ Selection\ =\ 01} \\ \boldsymbol{AluResult\_Mem} & \boldsymbol{if\ Selection\ =\ 10} \\ \boldsymbol{Data\_MEM} & \boldsymbol{if\ Selection\ =\ 11} \end{cases}$$

This improved design ensures efficient hazard resolution and data management in the Decode Stage, aligning with the processor's broader architectural goals.

The modifications in the Decode Stage, including the handling of data hazards and the integration of three multiplexers, are critical for improving the pipeline's efficiency. The inclusion of mechanisms to detect and resolve data hazards ensures that the processor can continue to execute instructions without unnecessary stalls, even when data dependencies exist between consecutive instructions. Additionally, the three multiplexers in the Decode Stage enhance flexibility, enabling precise control over the selection of input values based on the instruction type and execution context. These changes significantly optimize the data flow and reduce delays, ensuring that the Decode Stage can effectively prepare the necessary control signals and operands for the Execute Stage. By minimizing stalls and ensuring accurate data routing, these enhancements contribute to a smoother, more efficient pipeline, leading to improved overall performance. With these updates, the Decode Stage serves as a crucial link in the pipeline, ensuring that instructions are properly prepared for the subsequent stages of execution.

## Comparator Design

The comparator in the pipelined processor is responsible for evaluating the equality or inequality of two 32-bit inputs. Its functionality is governed by a control signal (zero signal) that determines the type of comparison to perform.
The logic of the comparator can be expressed as:

$$\boldsymbol{Output} = \begin{cases} \mathbf{1}, & \boldsymbol{if}\ (\boldsymbol{zero_{signal}} = \mathbf{1}\ \boldsymbol{and}\ \boldsymbol{data1} = \boldsymbol{data2})\ \boldsymbol{or}\ (\boldsymbol{zero_{signal}} = \mathbf{0}\ \boldsymbol{and}\ \boldsymbol{data1} \neq \boldsymbol{data2}), \\ \mathbf{0}, & \boldsymbol{otherwise} \end{cases}$$

In essence, the output is high (1) when the condition defined by the control signal is satisfied.
**Placement of the Comparator**
Although the comparator could theoretically be placed in the decode stage, it is instead implemented in the execute stage. This design choice is crucial to optimizing the processor's performance. If placed in the decode stage, the comparator would increase the critical path length, as its inputs are required early in the pipeline. By positioning it in the execute stage, where the inputs remain the same, the design avoids creating a bottleneck and allows for higher operating frequencies.

### Purpose of B_taken Signal

The B_taken signal is used in the Fetch Stage to determine whether a branch instruction should alter the normal program flow. This signal combines two conditions:

- **B_signal (Branch Equal):** A control signal indicating that a branch instruction is active and should be considered for execution.
- **EQU (Equality Condition)**: A signal generated based on the comparison of two brands, confirming whether the branch condition is satisfied.

### AND Gate Logic

The AND gate implements the B_taken signal with the following equation:

**B_taken= B_signal & Zero_Signal**

**Explanation of Inputs:**

- **B_signal**: Asserted (set to 1) when the instruction being executed is a branch-equal instruction.
- **Zero_Signal**: Asserted (set to 1) if the two operands compared in the Decode Stage are equal.

**Output:**

- **B_taken = 1:** Indicates that the branch condition is met, and the program counter should jump to the branch address.
    - **B_taken = 0:** Indicates that the branch condition is not met, and the normal sequential flow of instructions continues.

## Multiplexers in the Execute Stage

In the Execute Stage, three multiplexers are used to manage data selection and control signals. Each multiplexer performs a specific role to ensure smooth operation while addressing potential hazards and optimizing the Datapath.

**Multiplexer for ALU Operand 1 (MUX3 for wire_alu_input1)**

This multiplexer determines the first operand for the ALU based on the forwarding conditions and the current program counter (PC).

### Inputs:

- Data1: The default register value.
- PC: The program counter, used in certain branch and jump instructions.
- EXEMEM_readdata: Data forwarded from the Memory Stage.

**Selector Signals:**
The input selection is based on the ForwardC[1:0] signal:

| ForwardC[1:0] | Selected Input |
|---|---|
| 00 | Data1 (default) |
| 01 | PC |
| 10 | EXEMEM_readdata |

**Output:**
The selected value is forwarded as wire_alu_input1 to the ALU.
Note: The forwarding logic and its implementation will be discussed in a separate section.

**Multiplexer for ALU Operand 2 (MUX3 for wire_alu_input2)**

This multiplexer selects the second Operand for the ALU, handling immediate values and forwarded data.

**Inputs:**

- Data2: The default register value.
- Immex: The sign-extended immediate value for immediate instructions.
- EXEMEM_readdata: Data forwarded from the Memory Stage.

**Selector Signals:**
The input selection is determined by the ForwardD[1:0] signal:

| **ForwardD[1:0]** | **Selected Input** |
|---|---|
| 00 | Data2 (default) |
| 01 | immex |
| 10 | EXEMEM_readdata |

**Output:**
The selected value is forwarded as wire_alu_input2 to the ALU.
Note: Forwarding mechanisms and their details will be explained in a dedicated section.

**Multiplexer for Destination Register Selection (MUX4 for dst_reg)**

This multiplexer determines which register will be the destination for the result of the ALU operation or specific instructions.

**Inputs:**

- rt: The default destination register for most instructions.
- rd: The destination register for R-type instructions.
- 5'b11111: A constant used for jump-and-link (JAL) instructions.
- 5'b00000: A constant used to signify no destination (e.g., NOP).

**Selector Signals:**
The input selection is controlled by Ex-signal[1:0]:

| Ex_signal[1:0] | Selected Input |
|---|---|
| 00 | Rt |
| 01 | Rd |
| 10 | 5'b11111 (JAL) |
| 11 | 5'b00000 |

**Output:**
The selected value is forwarded as dst_reg, indicating the destination register for write-back.

**Integration of Multiplexers in the Pipeline**

The execute stage multiplexers enable efficient data management and hazard resolution through dynamic input selection:

1. **ALU Operand Selection:**
   Both MUX3 units (wire_alu_input1 and wire_alu_input2) ensure that the ALU receives the correct operands, leveraging forwarding and bypassing mechanisms to minimize stalls.

2. **Destination Register Selection:**
   The MUX4 unit (dst_reg) supports flexible destination register assignment, adapting to various instruction formats and special cases like JAL.

   These multiplexers collectively contribute to the pipeline's performance by ensuring proper data flow and minimizing control complexity.

These enhancements in the Execute Stage contribute to reducing stalls, resolving data hazards dynamically, and improving overall pipeline efficiency. By providing flexible Operand selection and efficient control over the program flow, these modifications enable the processor to handle a wide range of instructions while minimizing pipeline delays. Consequently, the Execute Stage becomes a vital component in the overall pipeline, ensuring that instructions are executed accurately and efficiently, leading to improved performance across the processor.

### Multiplexer for Write Data Selection (MUX4 for RAM Write data)

This multiplexer determines the source of the data that will be written to the memory unit during the memory stage of the pipeline.

**Inputs:**

- **write_data**: The default write data provided by the execute stage.
- **writedata2**: Data forwarded from another pipeline stage to resolve hazards.
- **memory_out**: Data read back from memory for specific operations.
- **32'b0**: A constant value used to signify no data write (e.g., during idle or default conditions).

**Selector Signals:**

The input selection is controlled by the 2-bit signal ForwardE[1:0] as follows:

| ForwardE[1:0] | Selected Input |
|---|---|
| 00 | write_data |
| 01 | writedata2 |
| 10 | memory_out |
| 11 | 32'b0 (default idle) |

**Output:**

The selected input is forwarded as the RAM write data, which represents the data sent to the memory unit for write operations. This multiplexer is essential in resolving data dependencies and ensuring that the correct data is supplied to the memory stage, facilitating accurate and efficient memory write operations.

### Multiplexer for Write-Back Data Selection (MUX2 for write_data)

This multiplexer determines the source of the data that will be written back to the destination register during the write-back stage of the pipeline.

**Inputs:**

- **readdata_out**: The data read from memory in the memory stage.
- **aluresult_out**: The result of the arithmetic or logical operation performed by the ALU in the execute stage.

**Selector Signal:**

The input selection is controlled by the signal memtoreg_out:

| memtoreg_out | Selected Input |
|---|---|
| 0 | aluresult_out |
| 1 | readdata_out |

## Output:

The selected input is forwarded as write_data, representing the data sent to the register file for write-back.

**Purpose and Role:**

This multiplexer ensures that the appropriate data is written back to the destination register based on the type of instruction being executed:

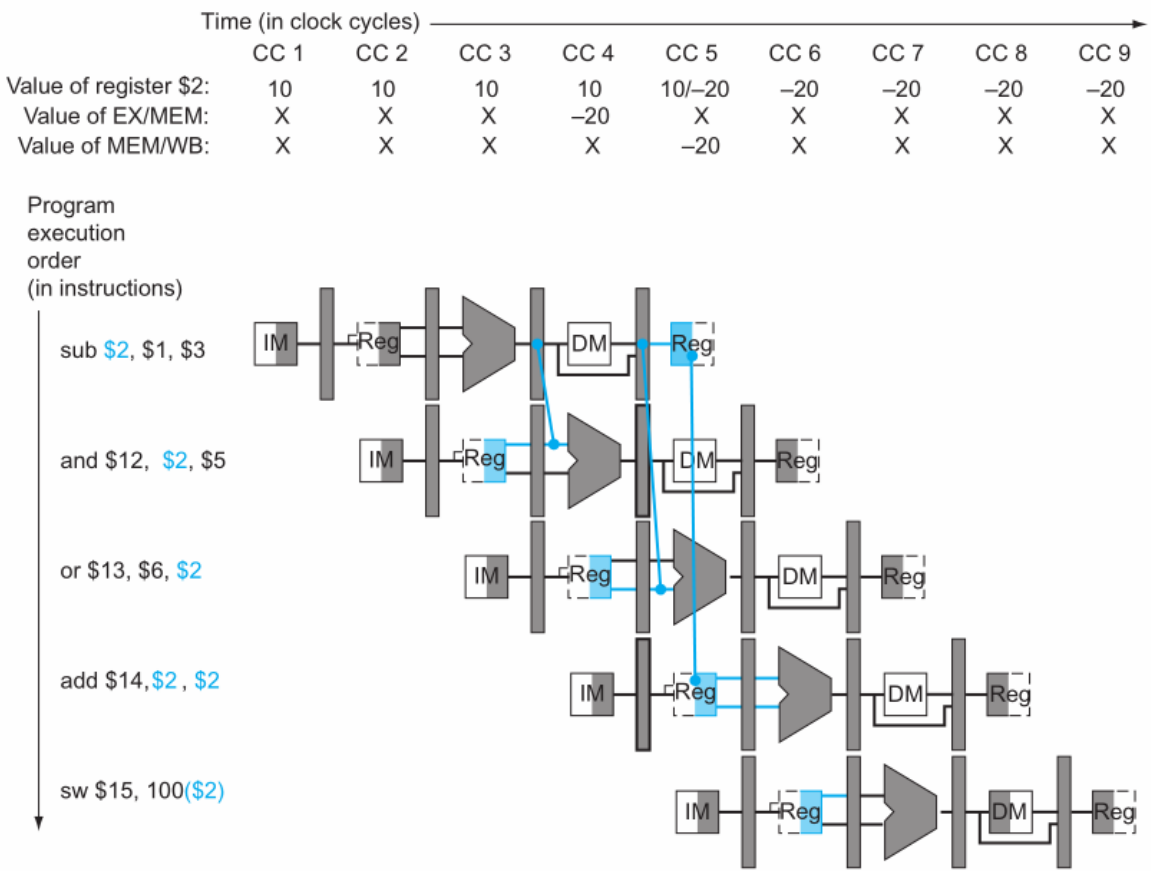
- For **arithmetic and logical operations**, the ALU result (aluresult_out) is written back.
- For **load instructions**, the memory read data (readdata_out) is written back.

By dynamically selecting the correct source of write-back data, the multiplexer ensures that the processor performs accurate operations while maintaining pipeline efficiency.

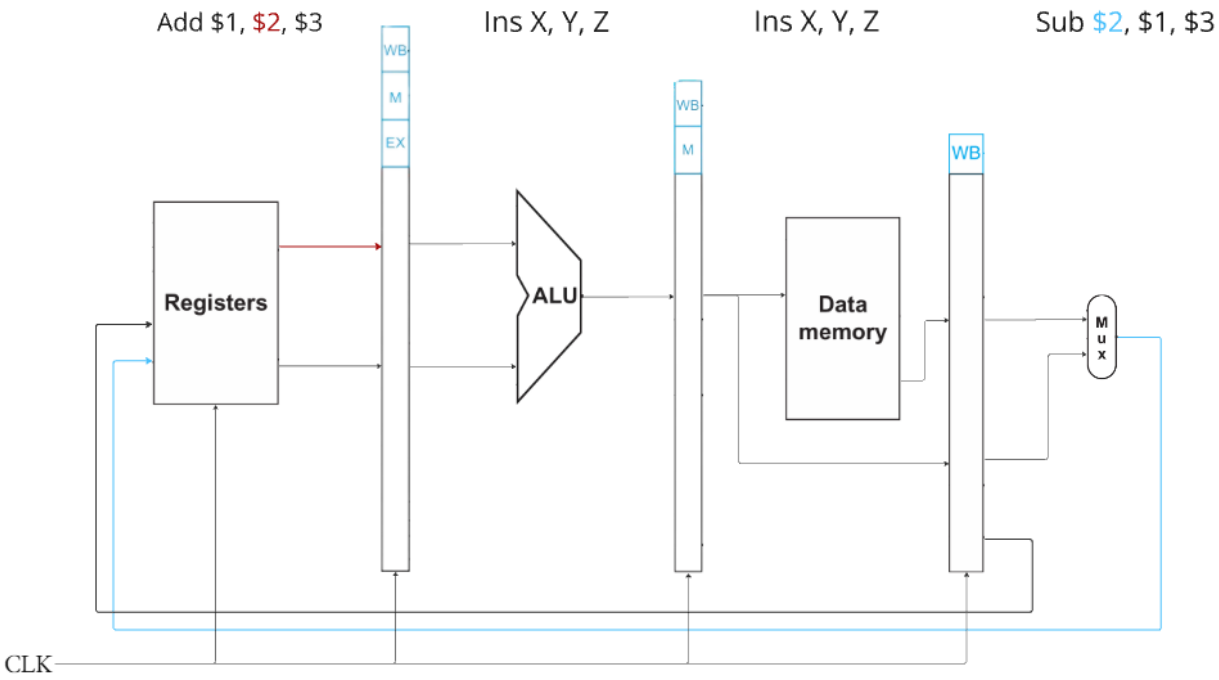
### Write-Back Stage:

In the context of our 5-stage pipelined processor, the goal is to ensure that instructions complete their lifecycle within 5 clock cycles. This includes the write-back operation, where the result from either the ALU (for R-type instructions) or Data Memory (for load instructions) is written to the register file. To achieve this, we explored two approaches to implementing the Write-Back (WB) stage and ultimately optimized the pipeline design by eliminating the need for an explicit WB pipeline register.

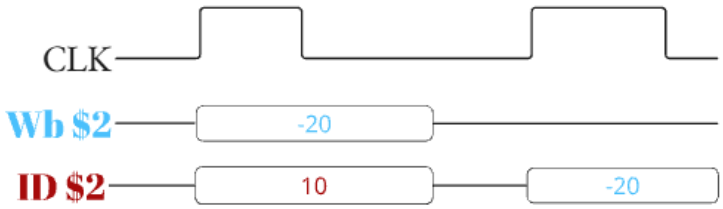
To further visualize the problem statement, consider the following figure:



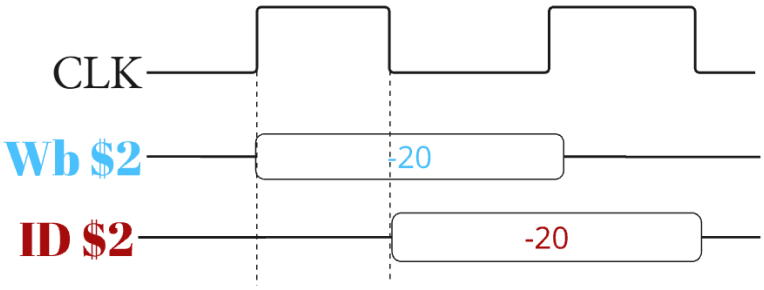
Since our registers are implemented using flip-flops, they require a clock cycle to latch any data at their inputs. If a Write-Back (WB) pipeline register is present, it outputs its data to the register file. However, the data will not be latched into the register file until the next clock cycle. This makes it impossible to achieve the result shown in the figure above, where the instruction is able to write to the register file while it is still in the WB stage.




Given that the Write-Back pipeline register is clocked on the positive edge of the clock and the registers in the register file are also clocked the same way, an instruction like Add $1, $2, $3 in the Decode stage will fetch the current value of $2, which is 10, instead of the updated value -20. The value of $2 will only be updated to -20 at the next clock cycle. This results in unexpected behavior, as illustrated in the figure below.

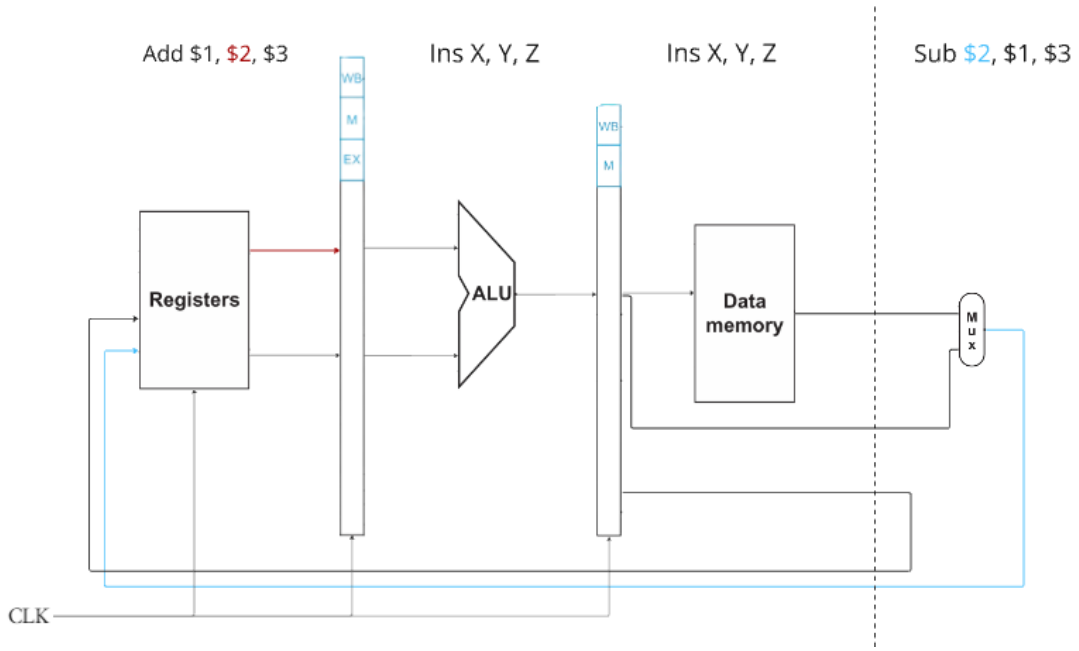


The first solution to this timing problem was to clock the register file with the negative edge of the clock which gives time for the data to propagate to the register file to be latched correctly and give an expected result. Although this configuration is functional, it limits the operating frequency of the design. The data

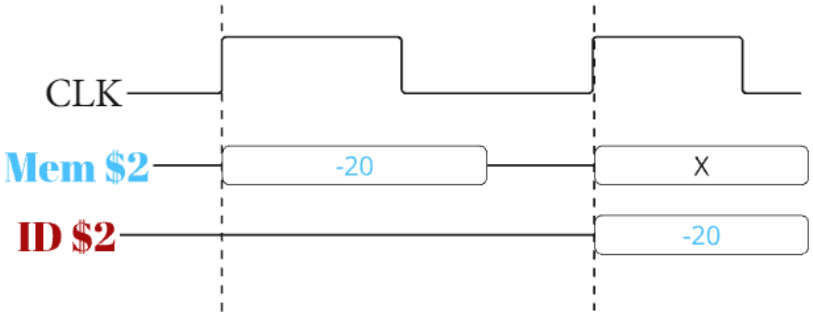



leaving the register file enters the combinational logic and has only half a clock cycle to propagate through it. This creates a new critical path, which significantly reduces the maximum operating frequency (Fmax) of the design.

We aim for a design where all registers are clocked on the positive edge of the clock. Therefore, we removed the Writeback pipeline register, as it introduced multiple issues without providing any tangible benefits. Additionally, it occupied 66 bits of unnecessary storage. By removing this pipeline register, we establish an implicit Writeback stage. In this configuration, the data leaves the Memory stage and is directed to the register file, where it gets latched in the next clock cycle. This next cycle corresponds to the 5th cycle of the instruction's execution, ensuring proper functionality within a 5-stage pipeline.



The visualization in the figure above creates the illusion of a structural hazard, where two instructions seem to access the register file simultaneously within the same clock cycle. However, this is not the case due to the flip-flop latching mechanism, which ensures that data is latched only at the positive edge of the clock. Specifically, when the Sub $2, $1, $3 instruction is in the Memory stage, the data is sent to the register file, and at the next positive clock edge, it is latched into register $2. Consequently, when the Add $1, $2, $3 instruction reaches the Decode stage, it correctly captures the updated value of $2. Furthermore, the data in the EX/MEM pipeline register corresponds to the instruction Ins X, Y, Z, ensuring proper operation without structural hazards. The implicit write-back stage shown in the figure, which holds the Subtract instruction, is included solely for visualization purposes and does not represent an additional explicit stage in the pipeline.



## Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is an essential part of a pipelined CPU, responsible for performing the arithmetic and logical operations required by various instructions. This section provides a detailed explanation of the ALU design.

**Overview of the ALU**

### 1. Inputs and Outputs:

- **Inputs**:
  - operand1 and operand2: These are the two operands on which the ALU performs the operation. They are usually fetched from registers or represent immediate values.
  - opSel: A control signal that determines which operation the ALU will perform (such as addition, subtraction, or logical operations).

- **Output**:
  - result: The result of the ALU operation. This output is used in the subsequent stages of the pipeline.

### 2. Parameterization:

- The ALU is designed to be flexible with parameters:
  - data_width (default: 32 bits): Specifies the bit width of the operands and the result.
  - sel_width (default: 4 bits): Specifies the width of the operation selector (opSel), which determines which operation the ALU should perform.

### 3. Supported Operations:

The ALU supports a variety of arithmetic and logical operations as determined by the opSel control signal. These operations are outlined in the table below:

| Operation | Control Code (opSel) | Description |
|---|---|---|
| ADD | 0000 | Adds operand1 and operand2. |
| SUB | 0001 | Subtracts operand2 from operand1. |
| AND | 0010 | Performs bitwise AND. |
| OR | 0011 | Performs bitwise OR. |
| NOR | 0100 | Performs bitwise NOR. |
| XOR | 0101 | Performs bitwise XOR. |
| SLT | 0110 | Sets result to 1 if operand1 is less than operand2, otherwise 0. |
| SLL | 0111 | Performs logical shift left. |
| SRL | 1000 | Performs logical shift right. |
| SGT | 1001 | Sets result to 1 if operand1 is greater than operand2, otherwise 0. |

## 4. ALU Design

The ALU uses a combinational logic block that operates based on the selected opSel. Each operation produces a result according to the following equations:

- **Arithmetic Operations**:
  - **Addition**: result = operand1 + operand2
  - **Subtraction**: result = operand1 - operand2
- **Logical Operations**:
  - **Bitwise AND**: result = operand1 & operand2
  - **Bitwise OR**: result = operand1 | operand2
  - **Bitwise NOR**: result = ~(operand1 | operand2)
  - **Bitwise XOR**: result = operand1 ^ operand2
- **Comparison Operations**:
  - **Set Less Than (SLT)**: result = (operand1 < operand2) ? 1 : 0
  - **Set Greater Than (SGT)**: result = (operand1 > operand2) ? 1 : 0
- **Shift Operations**:
  - **Logical Shift Left (SLL)**: result = operand1 << operand2[10:6]
  - **Logical Shift Right (SRL)**: result = operand1 >> operand2[10:6]

**Control Logic and Result Selection**

The ALU uses a case statement to select the operation based on the opSel control signal. The corresponding result is assigned to the result output. The selection logic is implemented as follows:

$$\text{result} = \begin{cases} \text{operand1} + \text{operand2}, & \text{if opSel} = 0000 \\ \text{operand1} - \text{operand2}, & \text{if opSel} = 0001 \\ \text{operand1\&operand2}, & \text{if opSel} = 0010 \\ \text{operand1}|\text{operand2}, & \text{if opSel} = 0011 \\ \sim(\text{operand1}|\text{operand2}), & \text{if opSel} = 0100 \\ \text{operand1} \oplus \text{operand2}, & \text{if opSel} = 0101 \\ (\text{operand1} < \text{operand2})?1:0, & \text{if opSel} = 0110 \\ \text{operand1} << \text{operand2}[10:6], & \text{if opSel} = 0111 \\ \text{operand1} >> \text{operand2}[10:6], & \text{if opSel} = 1000 \\ (\text{operand1} > \text{operand2})?1:0, & \text{if opSel} = 1001 \\ 32'b0, & \text{otherwise.} \end{cases}$$

This logic ensures that the correct operation is selected based on the opSel input.

## Pipeline Control Unit

The control unit in the pipeline design ensures that each stage of the pipeline executes its part of the instruction correctly. Control signals are generated during the **Instruction Decode (ID)** stage and are propagated to the subsequent pipeline stages through pipeline registers (ID/EX, EX/MEM, MEM/WB). These signals control various operations, such as ALU behavior, memory access, and register updates.

### Instruction Control Signals Table

The following table provides a detailed breakdown of control signals based on the instruction type. Each control signal determines the operation of specific components in the pipeline.

| Instruction | RegDst | Branch | MemRead En | Memto Reg | Mem Write En | Reg Write En | ALUSrc | aluop | JR_Signal | ZERO_s | JAL_signal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| R-type | 01 | 0 | 0 | 0 | 0 | 1 | 0 | Based on funct | 0 | 0 | 0 |
| ADDI | 00 | 0 | 0 | 0 | 0 | 1 | 1 | 0000 | 0 | 0 | 0 |
| ORI | 00 | 0 | 0 | 0 | 0 | 1 | 1 | 0011 | 0 | 0 | 0 |
| ANDI | 00 | 0 | 0 | 0 | 0 | 1 | 1 | 0010 | 0 | 0 | 0 |
| LW | 00 | 0 | 1 | 1 | 0 | 1 | 1 | 0000 | 0 | 0 | 0 |
| SW | 00 | 0 | 0 | - | 1 | 0 | 1 | 0000 | 0 | 0 | 0 |
| BEQ | - | 1 | 0 | - | 0 | 0 | 0 | 0001 | 0 | 1 | 0 |
| BNE | - | 1 | 0 | - | 0 | 0 | 0 | 0001 | 0 | 0 | 0 |
| SLTI | 00 | 0 | 0 | 0 | 0 | 1 | 1 | 0110 | 0 | 0 | 0 |
| J | - | 0 | 0 | - | 0 | 0 | - | - | 0 | - | 0 |
| JAL | 10 | 0 | 0 | 0 | 0 | 1 | 0 | - | 0 | - | 1 |
| JR | - | 0 | 0 | - | 0 | 0 | 0 | - | 1 | - | 0 |

### Explanation of Control Signals

- **RegDst**: Determines the destination register for the result.
    - 01 → Write to rd (R-type instructions).
    - 00 → Write to rt (I-type instructions).
    - 10 → Write to $ra (for JAL instructions).
- **Branch**: Signals a branch instruction.
    - 1 → Branch instructions (BEQ and BNE).
- **MemReadEn**: Enables reading from memory.
    - 1 → Load instructions (e.g., LW).

- **MemtoReg**: Determines if the value to be written back to the register comes from memory.
    - 1 → Value is from memory (LW).
    - 0 → Value is from ALU.
- **MemWriteEn**: Enables writing to memory.
    - 1 → Store instructions (e.g., SW).
- **RegWriteEn**: Enables writing to a register.
    - 1 → Instructions that modify a register (e.g., R-type, LW, ADDI, etc.).
- **ALUSrc**: Selects the second ALU operand.
    - 0 → Operand comes from a register (R-type).
    - 1 → Operand is an immediate value (I-type and memory instructions).
- **aluop**: Determines the operation performed by the ALU.
    - 0000 → Addition (ADD, ADDI, LW, SW).
    - 0001 → Subtraction (BEQ, BNE).
    - 0011 → OR (OR, ORI).
    - 0010 → AND (AND, ANDI).
    - 0110 → Set less than (SLT, SLTI).
    - 0111 → Shift left logical (SLL).
- **JR_Signal**: Specifies a jump to the address in a register (JR instruction).
- **ZERO_s**: Used for branch decisions.
    - 1 → Take branch if ZERO flag is set (e.g., BEQ).
    - 0 → Take branch if ZERO flag is not set (e.g., BNE).
- **JAL_signal**: Specifies a JAL instruction, ensuring the return address is saved in $ra.

## Pipeline Control Signals Behavior Table

This table shows how the various control signals behave for different instruction types, indicating when they are asserted or deasserted to achieve the correct operation.

| Signal Name | Effect when Deasserted | Effect when Asserted |
|---|---|---|
| RegDst | Default register destination (e.g., rt). | Selects specific destination registers (e.g., rd for R-type or 31 for JAL). |
| Branch_eq | No branch operation occurs. | Branches if the operands are equal (used for BEQ). |
| Branch_ne | No branch operation occurs. | Branches if the operands are not equal (used for BNE). |
| MemReadEn | Memory is not read. | Enables memory read operation (used for LW). |
| MemtoReg | ALU result written to the register. | Memory data is written to the register (used for LW). |
| MemWriteEn | No memory write operation. | Enables memory write operation (used for SW). |
| RegWriteEn | No data is written to the register file. | Enables writing data into the register file (used for R-type, I-type, and LW). |
| ALUSrc | Second ALU operand from the register file. | Second ALU operand is an immediate value or shift amount (used for I-type or shift instructions). |
| ZERO | Result of ALU comparison ignored. | Used for zero comparison (e.g., BEQ or BNE instructions). |
| JAL_signal | Normal operation, no jump. | Performs jump-and-link operation, saving the return address to register 31. |
| Jump_signal | Normal PC progression. | PC is set to jump target address (used for J or JAL). |
| JR_Signal | Normal PC progression. | PC is set to the address in the register specified (used for JR). |
| aluop | Default ALU operation (e.g., addition). | Specifies the ALU operation (e.g., ADD, SUB, OR, AND, SLT). |

# Forwarding Unit

Data hazards, as described in the overview section, can stall the pipeline and significantly degrade the throughput of a pipelined processor. To address this, we implemented data forwarding, which bypasses the need to stall by providing the required data directly to dependent instructions.

Through an iterative design process, discussed in the *Optimization and Evaluation* chapter, we determined that implementing multiple data forwarding paths provided the most efficient solution.

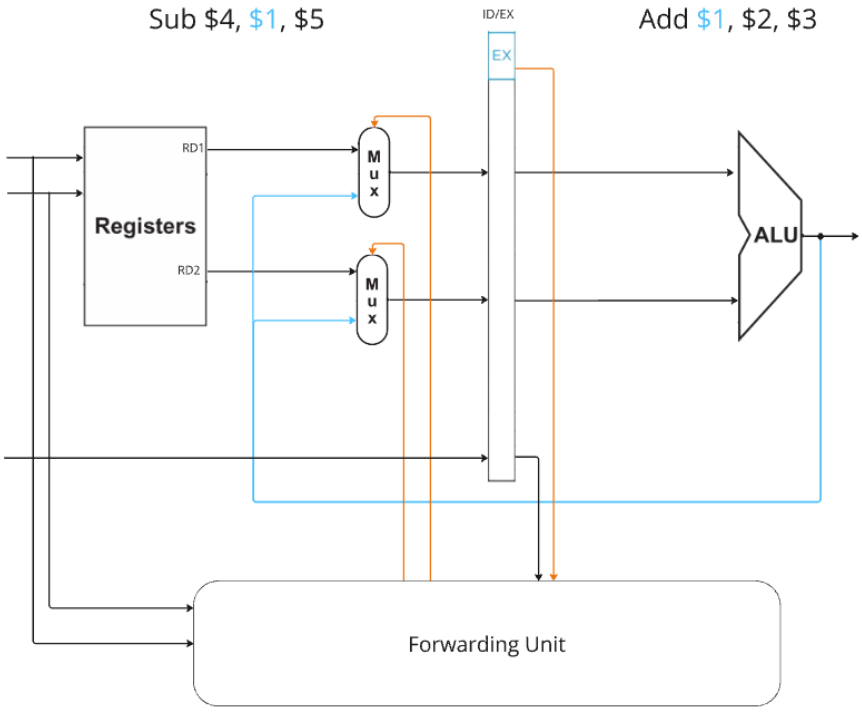
In our pipeline design, the Forwarding Unit resolves two primary cases of data dependency:

- When a subsequent stage requires the result from the Execute Stage (ALU output).
- When a subsequent stage requires data read from the Data Memory Stage.

A **first-order data dependency** arises when an instruction depends on the result of the immediately preceding instruction. This dependency is resolved by forwarding the ALU result from the Execute stage directly to the appropriate multiplexers in the Decode stage.

Add $1, $2, $3

Sub $4, $1, $5



The ALU result is forwarded to both multiplexers in the Decode stage to accommodate cases where the result is needed as either the first or second operand of the instruction currently being decoded.

The Forwarding Unit needs the meta data of the instructions to determine what we need to forward, So by inputting the Rs, and Rt indices from the decode Stage and the Rd index from the Execute Stage If a match was found then we select to forward the ALU result.

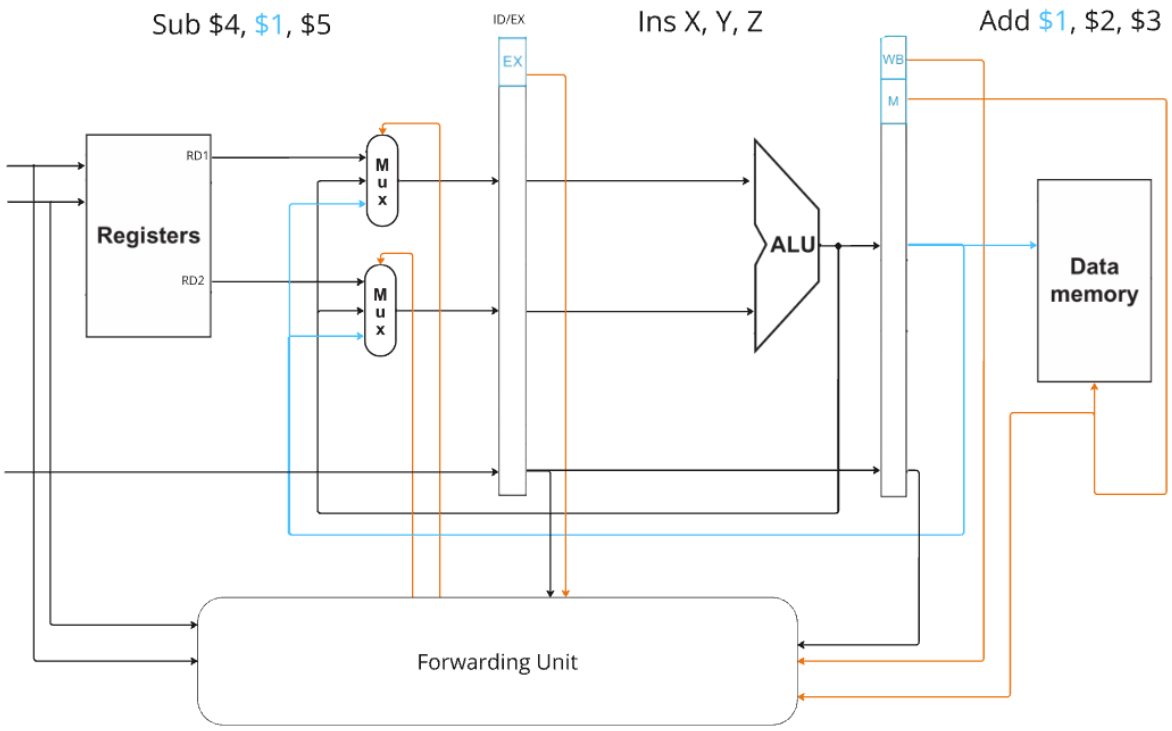
It also important not to forward unwanted data, and for that we need to check if the instruction in the execute stage is an instruction that writes to the register file, we input the Reg-Write Signal from the Execute Stage to the Forwarding Unit for that reason, And we need to check that Rd in the Execute Stage is not equal to zero.

A **second-order data dependency** arises when an instruction depends on the result of an instruction that is two stages ahead in the pipeline. This dependency is resolved by forwarding the data output from the Memory stage (memory address) directly to the appropriate multiplexers in the Decode stage.

Add $1, $2, $3

Ins X, Y, Z

Sub $4, $1, $5



Similar to the **first-order data dependency** we check for the decode stage **Rs** or **Rt** and see if it equals the memory stage **Rd** But there is an additional condition here that we must consider, which is the **Regwrite** signal and the **MemReadEn** signal and from the memory stage, in order to forward the **address** the **MemReadEn** must be 0.

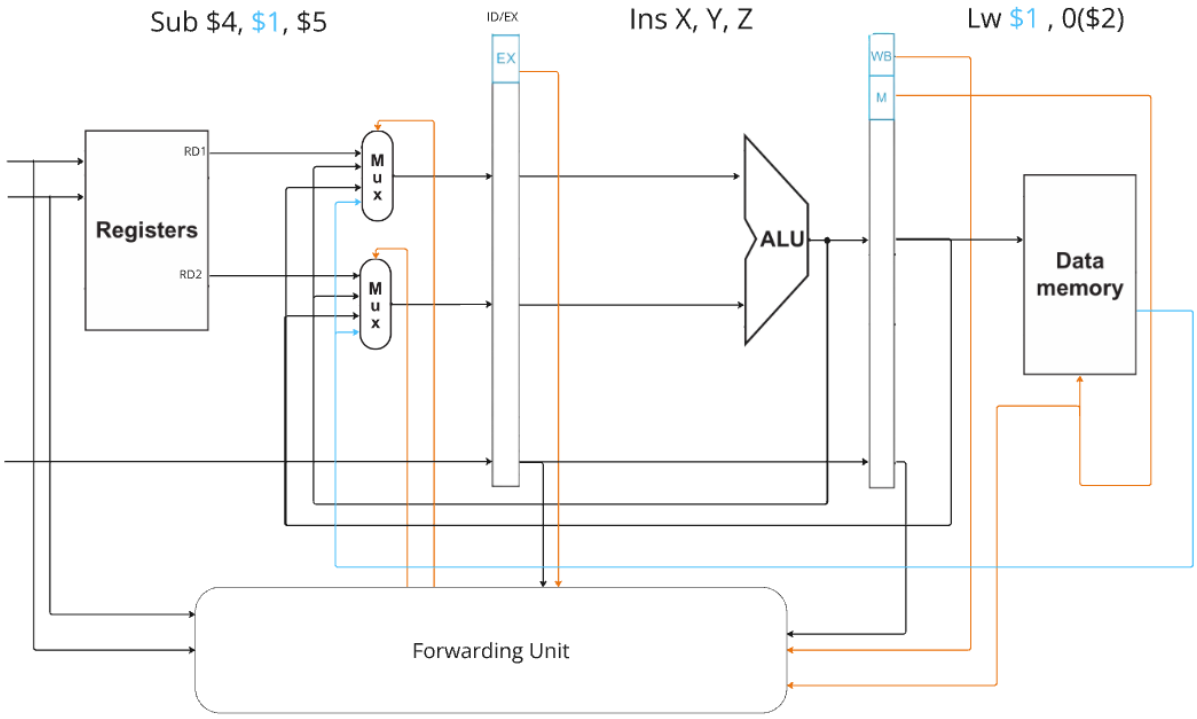
The second case of data dependencies involves loaded data from memory. When an instruction requires data that has just been loaded from memory by a previous instruction that is still in the pipeline, stalling the pipeline is undesirable. To address this, we forward the ReadData from the Data Memory stage back to the earlier stages of the pipeline, ensuring the dependent instruction can proceed without delays.

**Second-order load dependency**

Lw $1 , 0($2)

Ins X, Y, Z

Sub $4, $1, $5

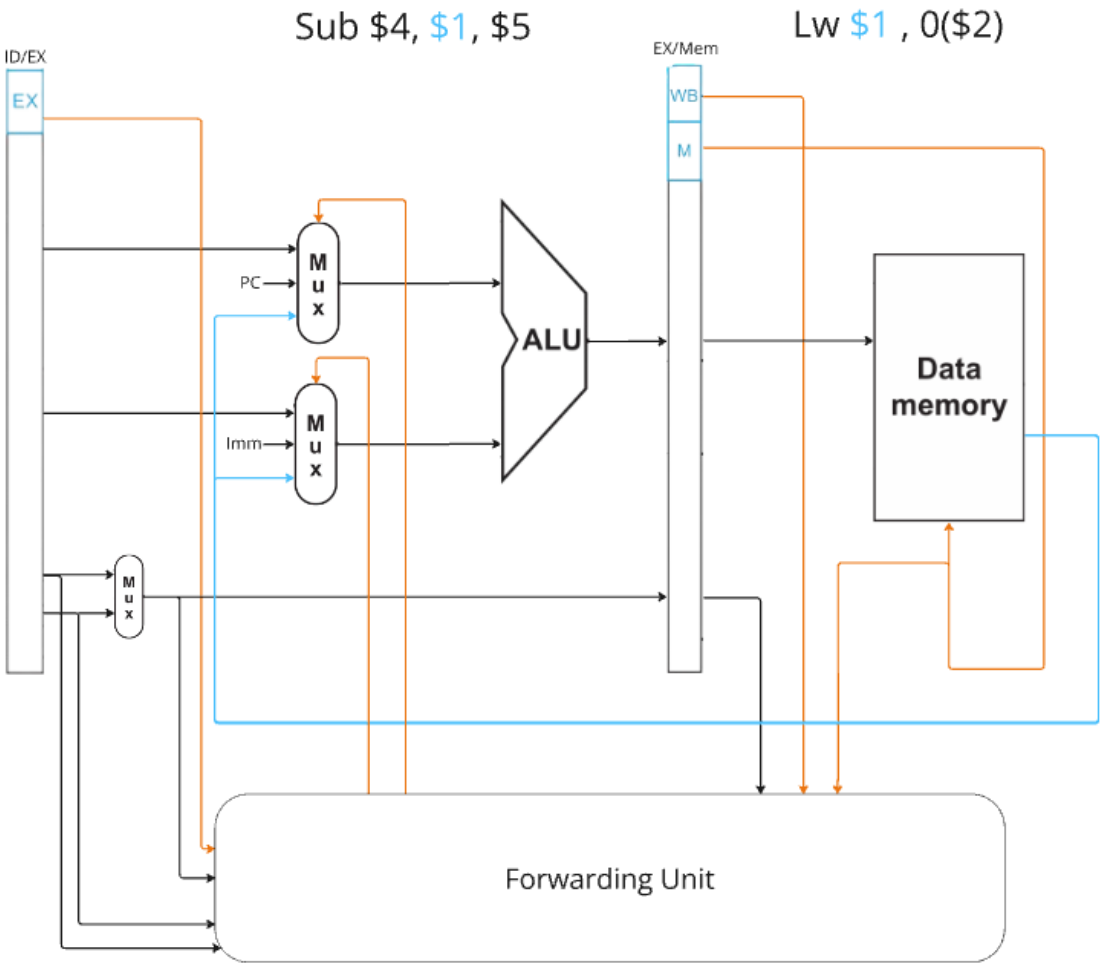



Checking for the decode stage **Rs** or **Rt** and see if it equals the memory stage **Rd,** In this case the **MemReadEn** is handy in determining that we need to forward the result data from the **DataMemory.**

To solve the **first-order load dependency** where an instruction is dependent on the previous instruction result, but the previous instruction is load word:

Lw $1 , 0($2)

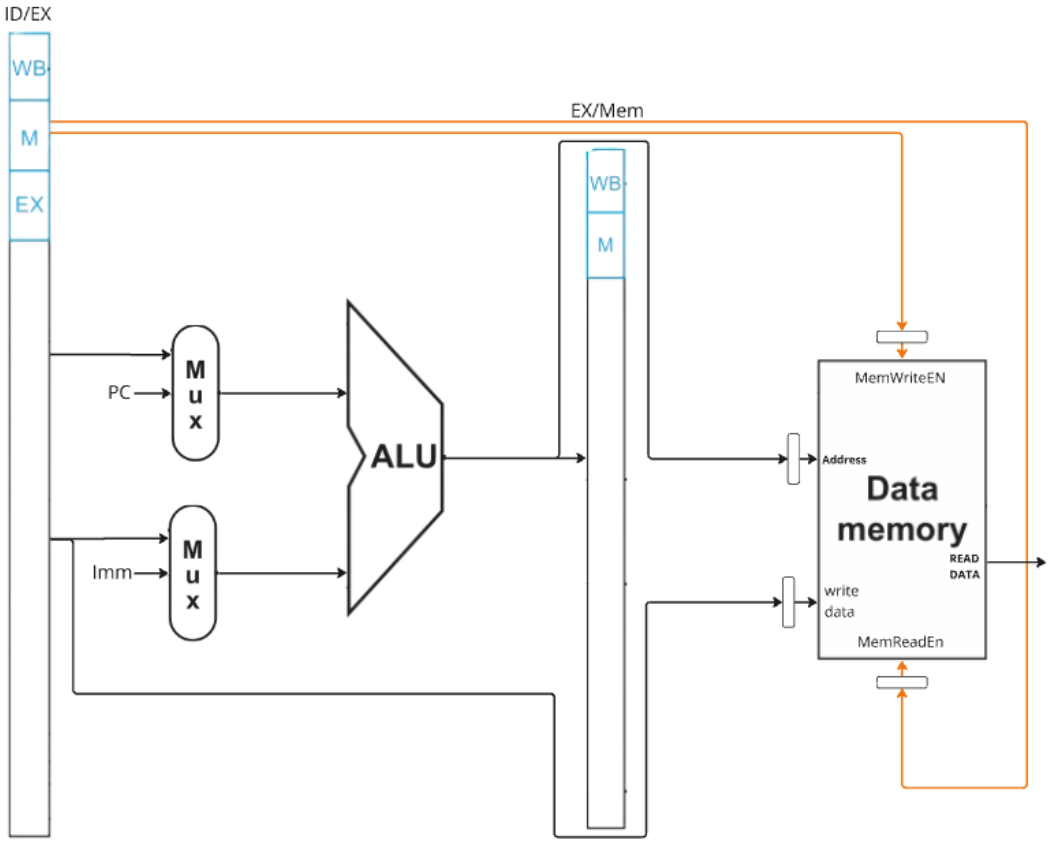
Sub $4, $1, $5




Designing this forwarding path presented several challenges, the first of which involved the Data Memory. In Quartus, we discovered that the Memory IP requires registered inputs, meaning the address value leaving the EX/MEM pipeline register cannot fetch the required memory location in the same cycle. This behavior is undesirable, as we want the Load Word instruction in the Memory stage to access the desired memory content within the same cycle.

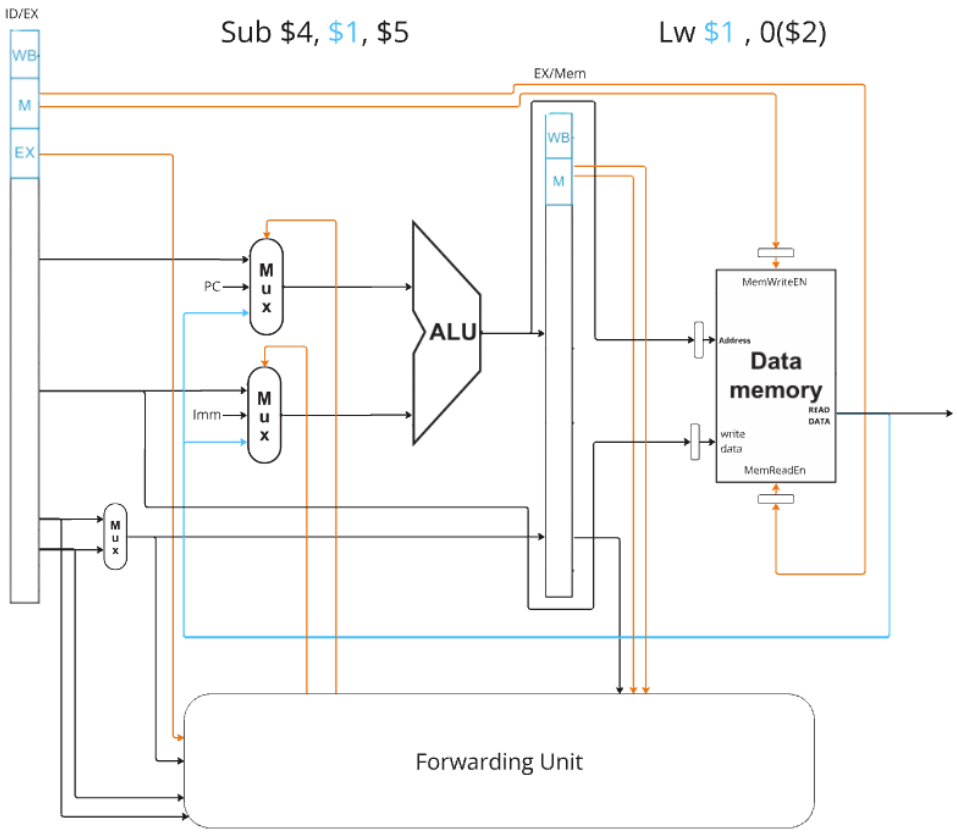
One potential solution was to clock the Data Memory with the negative edge of the clock. However, this approach introduced another issue: the signal leaving the Data Memory would have only half a clock cycle to propagate through the multiplexer preceding the ALU and the ALU itself. This significantly limits the maximum operating frequency of the design, making it impractical for high-performance applications.

A solution to this challenge was to treat the registered inputs of the Data Memory as an extension of the EX/MEM pipeline register. By directly connecting the necessary data from the Execute stage to the Data Memory inputs, we could clock the Data Memory with the positive edge of the clock. This ensures that the memory content is ready by the time the Load Word instruction reaches the Memory stage, eliminating the need for additional delays.




At first glance, it might appear that this design introduces a **Structural Hazard**. However, this is merely a visualization issue. By treating the registered inputs of the Data Memory as part of the EX/MEM pipeline register, we ensure that no two instructions access the memory simultaneously. This design effectively eliminates the potential for a structural hazard while maintaining proper pipeline functionality.

Now that we have ensured the safe operation of the Data Memory on the positive edge of the clock, and that the memory content is available in the same cycle when a Load Word instruction is in the Memory stage, we can integrate this solution with the previous design. Specifically, we can forward the content of the memory directly to the ALU, maintaining the efficiency of the forwarding paths without introducing pipeline stalls.

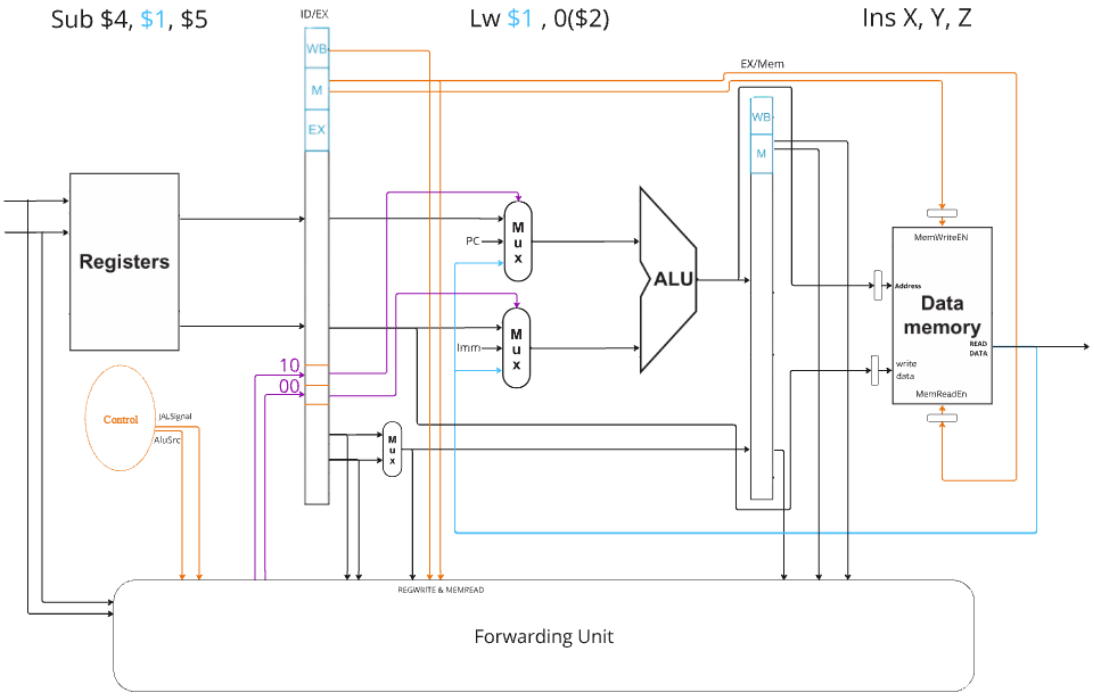



This design ensures that the content of memory is ready when a load instruction is in the MemoryStage, and since we can operate the datamemory at the positive edge of the clock then we can forward the memory content backwards to the ALU while ensuring that it has time to propagate safely at high frequencies.

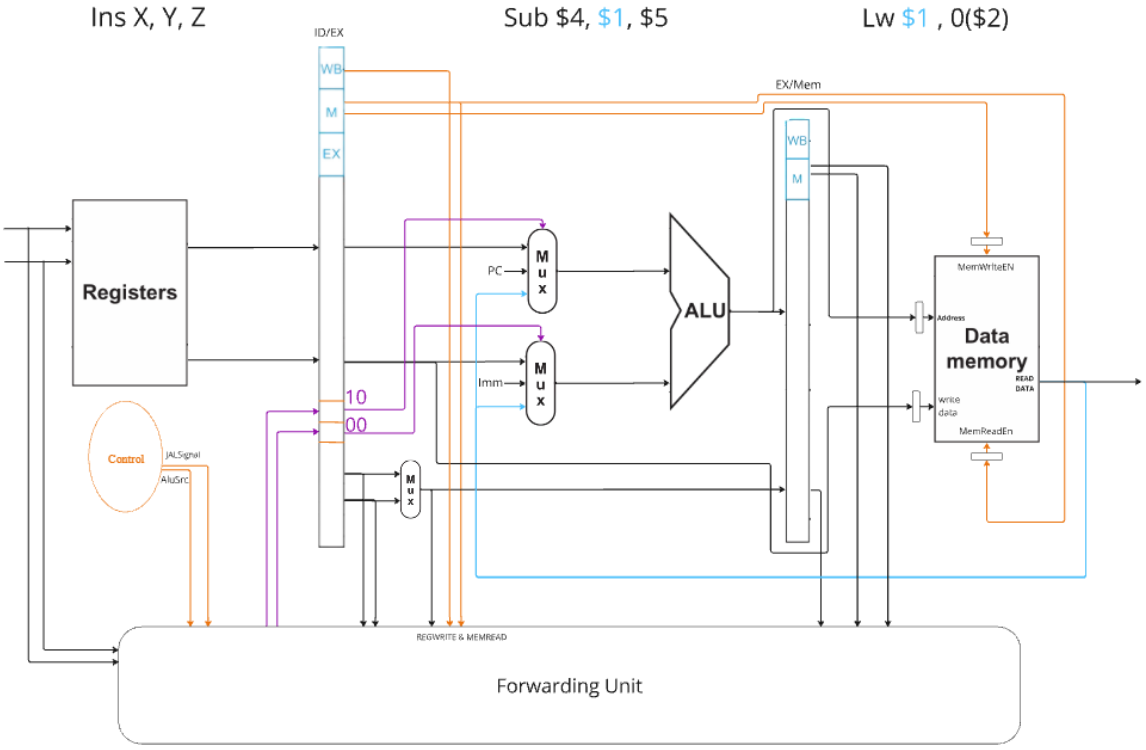
While we can now successfully forward memory content backward, designing the control signals for the multiplexers introduced another challenge. As the Forwarding Unit became more complex, the combinational logic required to determine the multiplexer inputs also grew. This increased complexity extended the critical path, as signals from the Memory stage must propagate through the Forwarding Unit logic to generate the multiplexer control signals. Subsequently, the forwarded data travels through the ALU and ultimately reaches the Data Memory address register. This longer critical path imposes constraints on the maximum operating frequency of the design, making it an important aspect to address during optimization.

To optimize this critical path, we decided to pipeline the multiplexer inputs in the Execute stage, requiring their values to be computed earlier in the Decode stage. Specifically, we check for matches between the Rd register in the Execute stage and the Rs or Rt registers in the Decode stage. However, additional signals must be considered to handle cases where forwarding is not required.

For example, if no forwarding is needed, we must determine whether to select RD1 or the PC value for the Operand1 multiplexer, and whether to choose RD2 or the Immediate value for the Operand2 multiplexer. To facilitate this, we pass these signal values to the Forwarding Unit, ensuring it has all the necessary information to compute the correct multiplexer inputs efficiently.



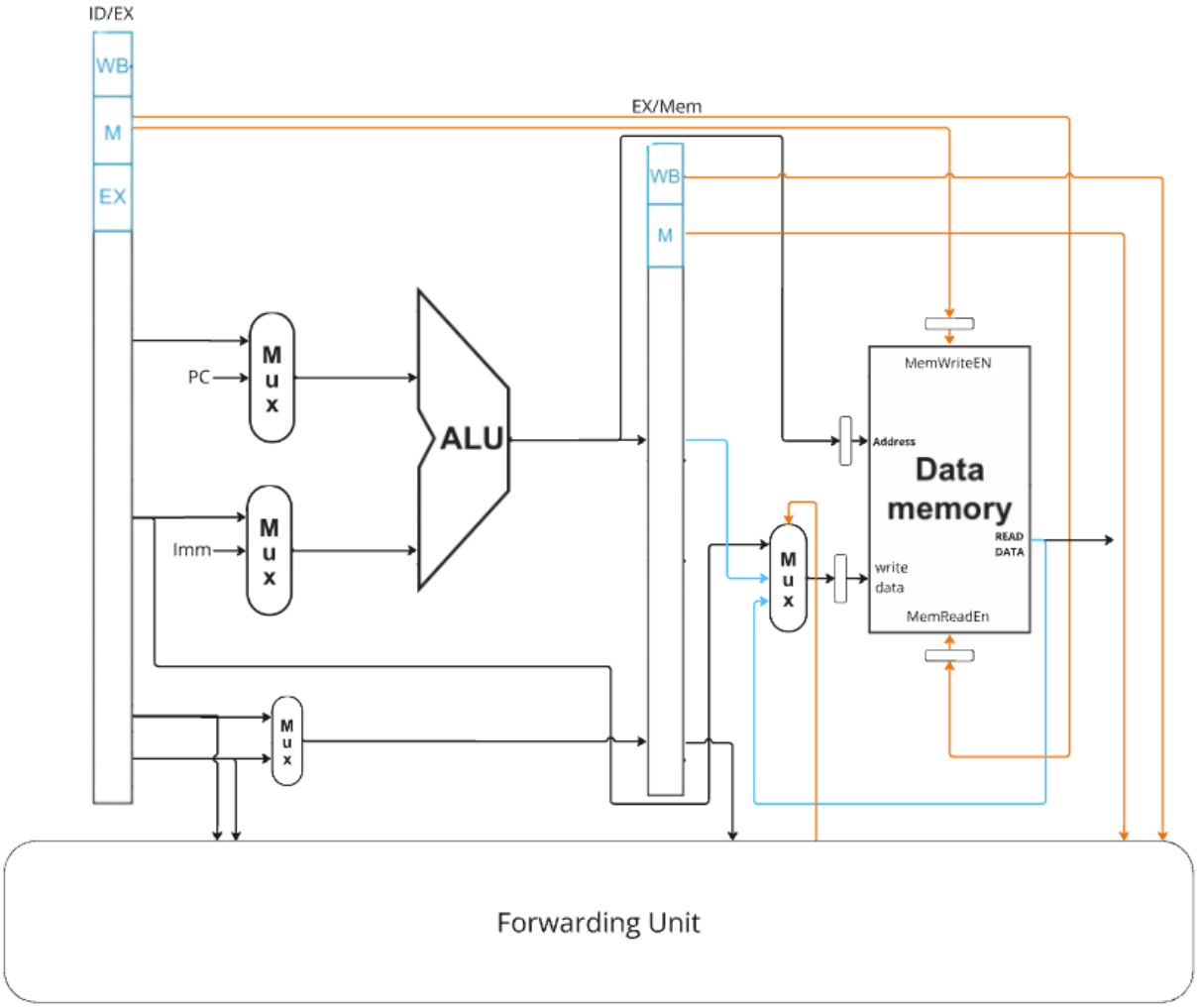

So by preparing the multiplexer selector values early in the Decode Stage we can safely use them in the execute stage to determine the necessary operands for this instruction ALU operation.



Lastly, there are multiple cases where we need another multiplexer in the memory stage that inputs to the write data register.

It's important to note that, for visualization purposes, this multiplexer is shown in the Memory stage. However, it should conceptually be treated as part of the Execute stage because the WriteData port of the Data Memory is registered.

In other words, any input before the Data Memory registers is effectively considered as Execute stage data. This approach aligns with our design principle, ensuring accurate stage representation and minimizing confusion during analysis.

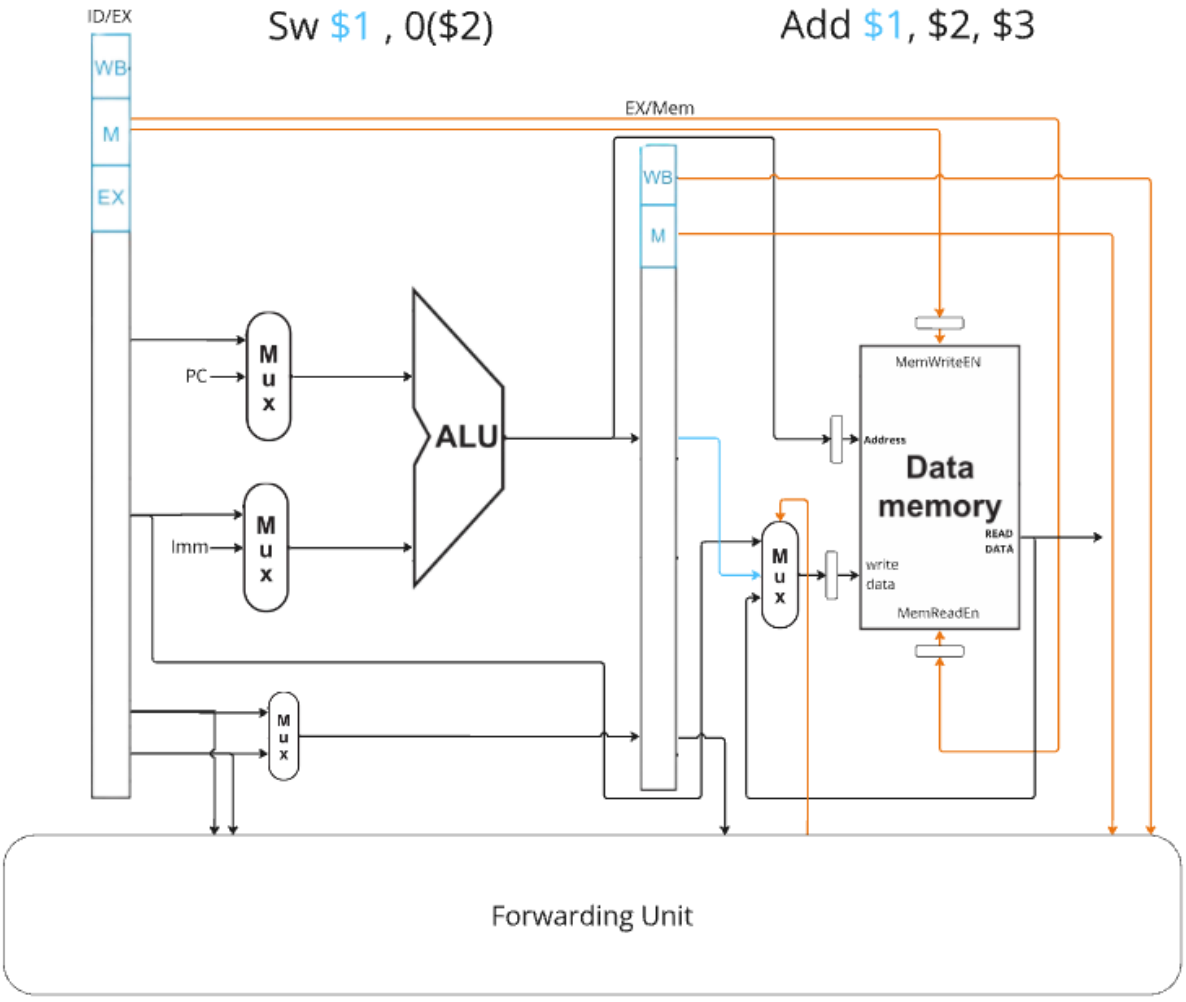


In normal execution if we encounter a **Sw** instruction in the execute stage we need to send the data to store to this Multiplexer. However there is some dependencies to consider.

In a case where we are storing a value that is yet in the pipeline :

Add $1, $2, $3

Sw $1 , 0($2)



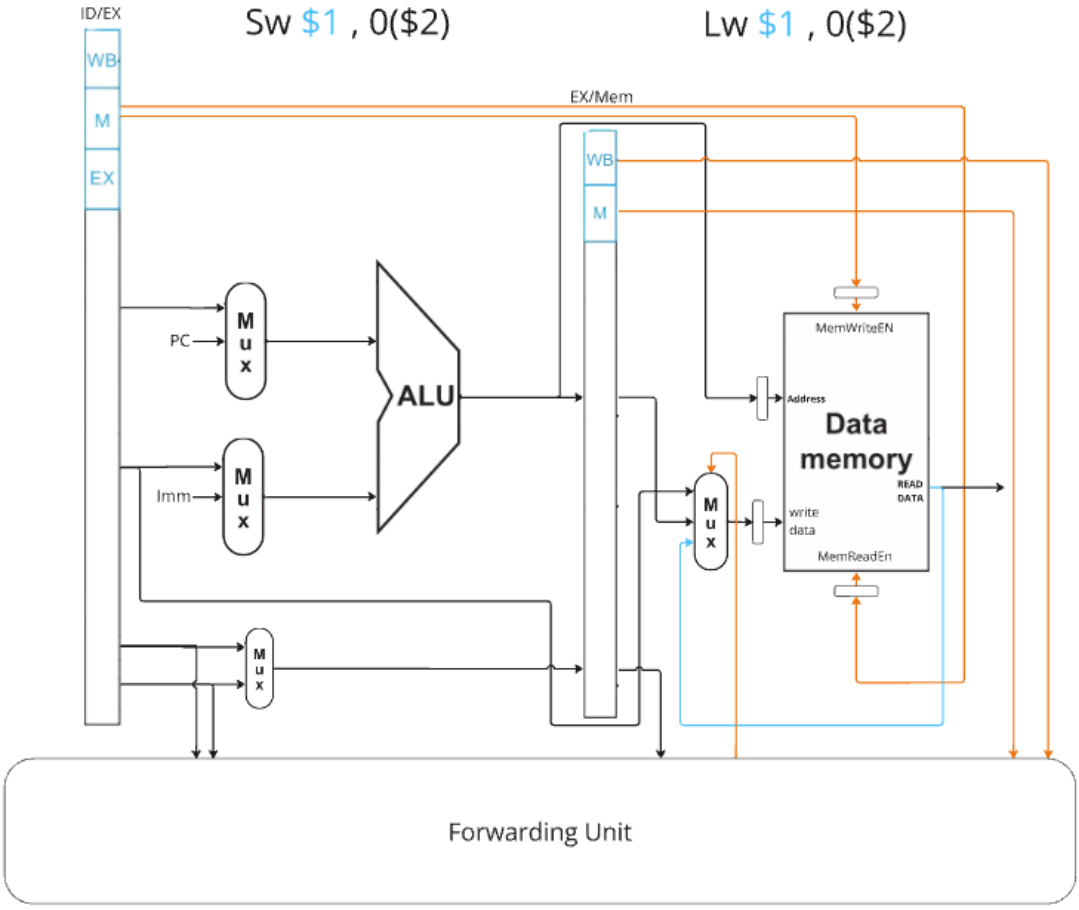
To store the value of $1, we forward it using the multiplexer shown in the figure. To determine the selector value for this multiplexer, we compare the destination register (Rd) in the Memory stage with Rt in the Execute stage. If a match is found, we then check the **RegWrite** signal in the Memory stage. If **RegWrite** is true and **MemReadEn** is false, the value is forwarded to be stored in the next clock cycle.

A similar but less common case must also be addressed to ensure a reliable and error-free design. Consider the following instruction sequence:

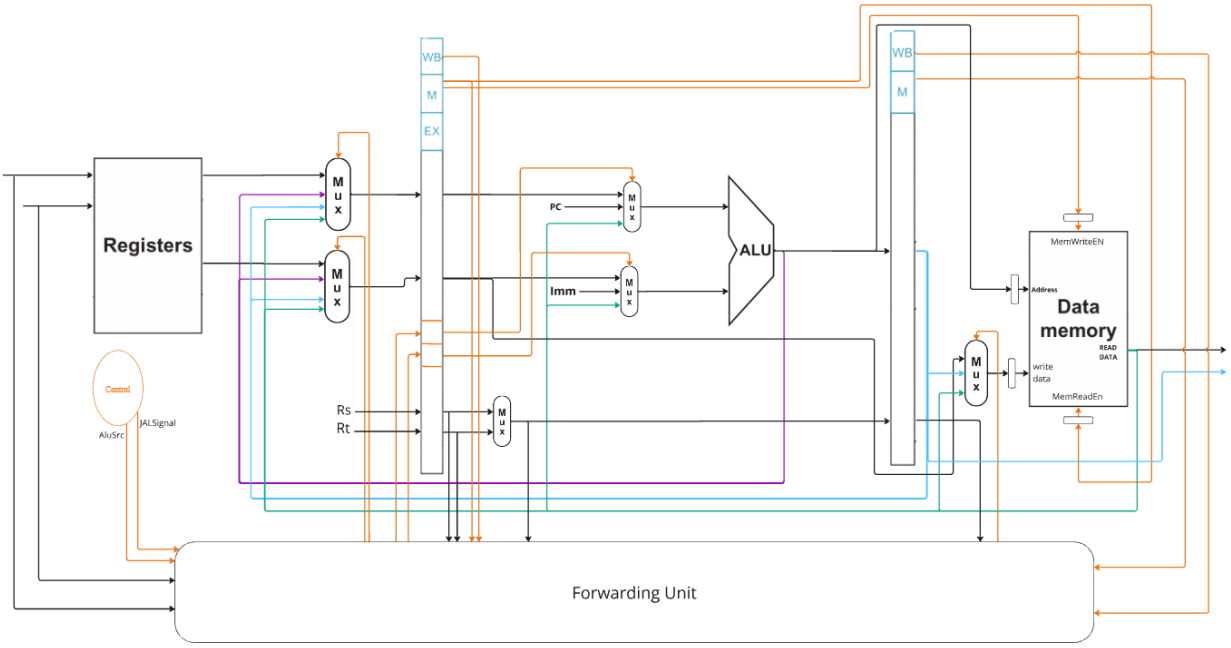
Lw $1 , 0($2)

Sw $1 , 1($2)

In this scenario, we need to forward the loaded data from memory to achieve the correct result. To handle this, we use the same conditions as the previous case: if Rt in the Execute stage matches Rd in the Memory



stage, we then check the **MemReadEn** signal. If **MemReadEn** is true, the loaded data is forwarded to be stored in the next clock cycle.




The forwarding mechanisms we implemented address an inefficiency in the Quartus memory IP. With this configuration, we not only resolved data dependencies and improved pipeline throughput but also reduced the size of the EX/MEM pipe register by 32 bits. Previously, the value of RD2 had to be stored in the pipe to be written to memory later. Now, we bypass this requirement by directly storing the value to the WriteData register port in the Data Memory.

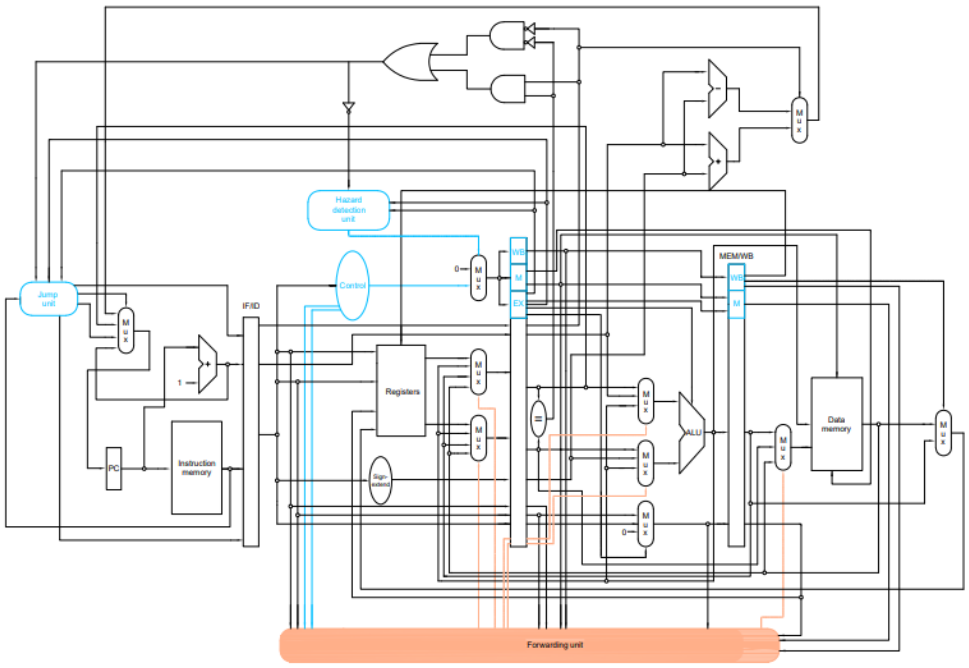
However, it is important to note that this optimization applies only to RD2. The other values in the EX/MEM pipe must still be stored, as they are required for either the WriteBack stage or the Forwarding Unit. While the inputs to the Data Memory are registered, they are write-only and cannot be read by any other unit, limiting their flexibility for further Usage.

Now that we addressed all the dependencies, we can safely say that there is no case where we need to stall the pipeline in order to resolve a data hazard given the ISA that is provided for this phase of the contest.

And by integrating all the solutions together as shown in the figure above we can ensure a consistent flow in execution without stalling the pipeline.

This configuration successfully optimized the critical path and reduced its overall latency. While we explored other configurations for our forwarding logic, as detailed in the Optimization and Evaluation chapter of this report, this design proved to be the most efficient solution.

## Branch Prediction:




To further improve the design efficiency, we decided to introduce branch prediction. While it is true that branch prediction reduced the overall Fmax of the design due to the added complexity, it significantly mitigates the NOP operations typically inserted when branch instructions are taken. In this design, the next instruction is always fetched regardless of the branch outcome. Since the result of a branch instruction is resolved in the Execute stage, taking a branch requires flushing two previously fetched instructions in the pipeline.

By implementing a prediction mechanism, we can reduce the number of pipeline flushes when the prediction is correct. Even in cases where the prediction is incorrect, no additional penalty is incurred compared to the original finished design without prediction. This enhancement helps balance the trade-off between performance improvement and a slight reduction in maximum operating frequency.

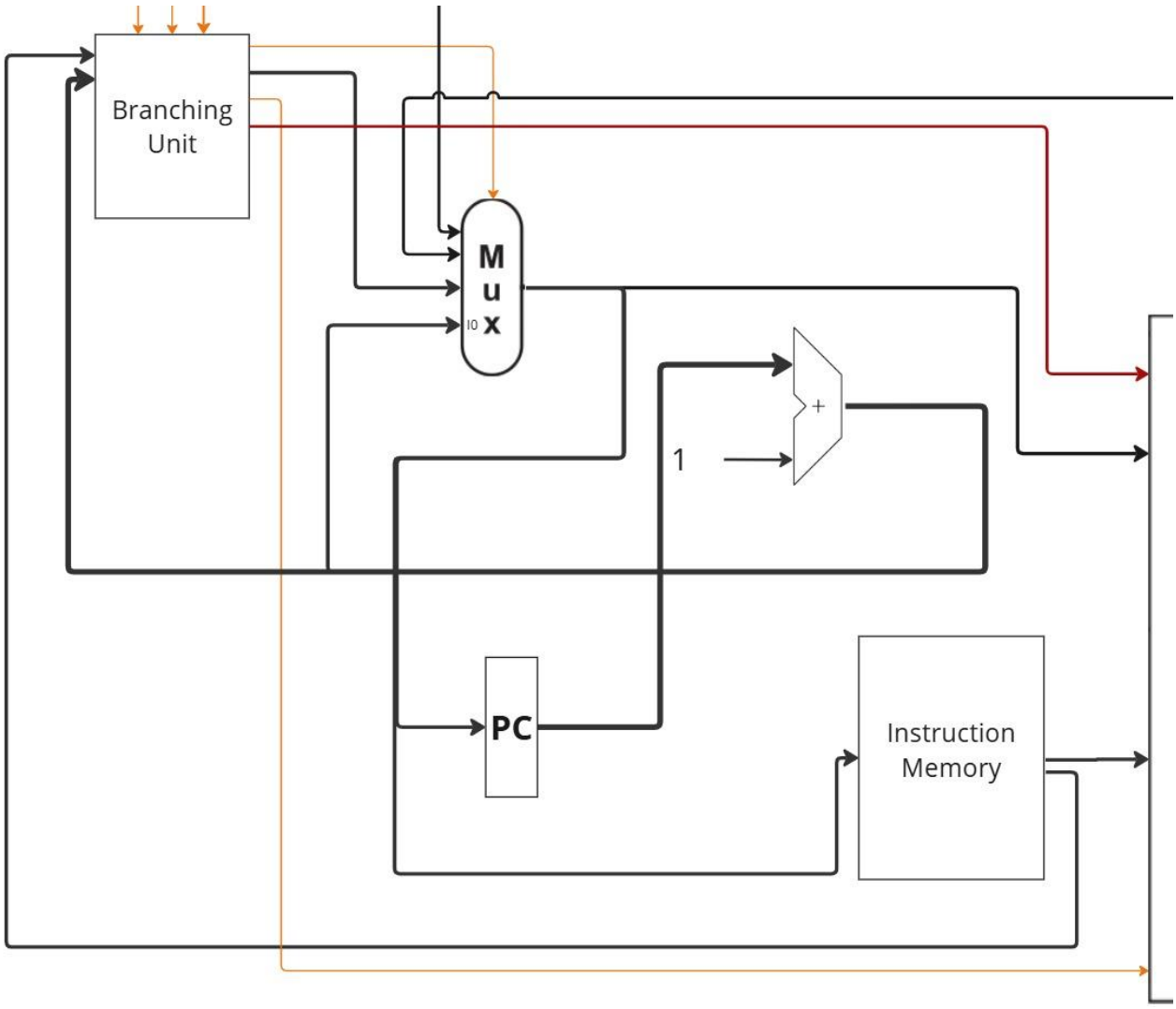
### Technique

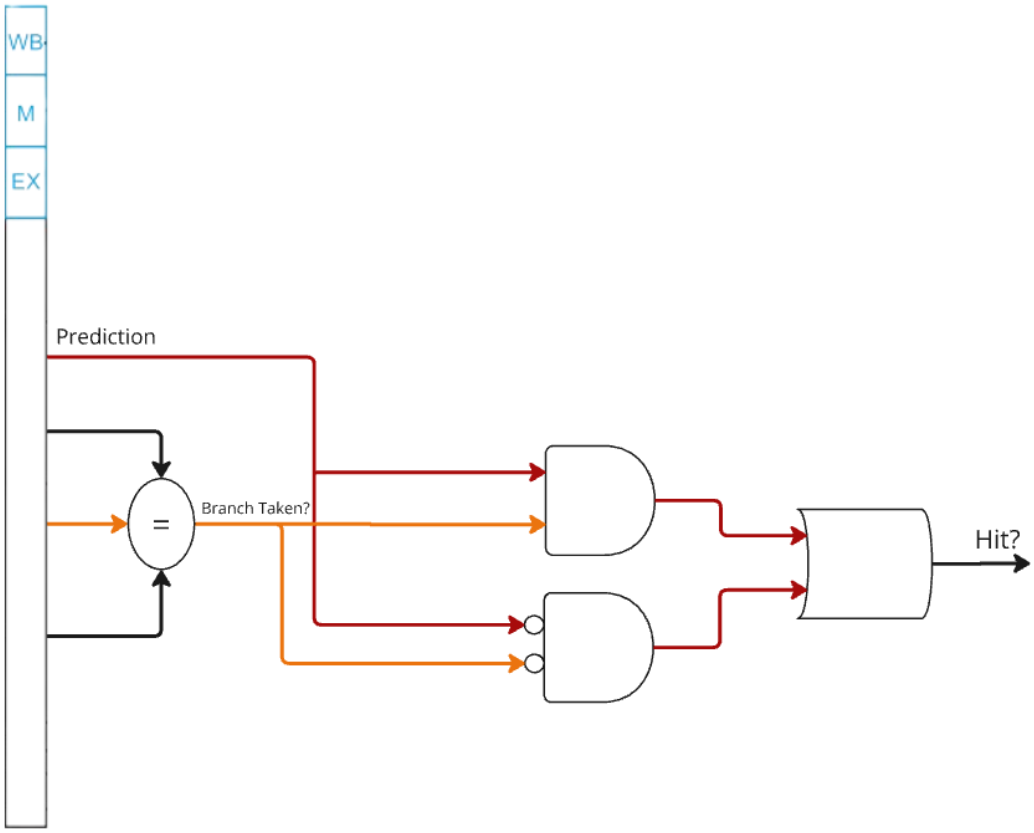
The "backward taken, forward not taken" static branch prediction design is a simple and efficient approach that assumes backward branches (typically associated with loops) are more likely to be taken, while forward branches (commonly conditional branches) are less likely to be taken. This design is effective in many cases because loops often involve backward branches that are indeed frequently taken, while conditionals are often not. The simplicity of this scheme leads to low hardware overhead and easy implementation, making it well-suited for resource-constrained systems. However, its limitations include inaccuracies for non-loop backward branches or frequently taken forward branches, which can result in unnecessary pipeline flushes and reduced performance. Despite these shortcomings, the approach aligns with typical program behavior, especially in loops, and is supported by empirical data that shows a high predictability of backward branches. While more sophisticated dynamic branch prediction techniques could improve accuracy, they would introduce added complexity and hardware requirements.

To implement the branch predictor in our design, several modifications were made to integrate the functionality smoothly. First, to determine whether the fetched instruction is a branch, we leveraged the previously designed JumpUnit. We then combined both the JumpUnit and the BranchPrediction Unit into a single unit, which we called the Branching Unit. This unit serves the dual purpose of identifying Jump instructions and making branch predictions.

Once we identify that the instruction is a branch, we need to determine if it is a forward or backward branch. This decision requires comparing the Program Counter (PC) with the target address of the branch. The branch target address is calculated by adding the offset from the branch instruction to the current PC value. By comparing the PC with the target address, we can predict whether the branch is likely to be taken (backward) or not taken (forward), following our "backward taken, forward not taken" prediction strategy.

If the branch is determined to be backward, we predict it as taken and update the PC to fetch the branch target address in the next clock cycle. Additionally, we set a flag that indicates the prediction outcome: a value of 1 if the prediction was "taken," and 0 if the prediction was "not taken." This prediction flag is then passed down the pipeline, reaching the execute stage where it can be used to determine the final outcome of the branch instruction.






When the branch instruction reaches the execute stage we must know if the prediction we made was correct or not, and by designing a simple circuit we can output a Hit Or miss value to be propagated back to the branching unit.

This circuit determines if our branch prediction was correct by comparing the predicted outcome with the actual result, which is provided by the taken signal. If the predicted value matches the taken signal, the prediction is considered a hit. To simplify this comparison, we leverage the Zero Unit, which outputs a value based on the type of branch instruction.

The Zero Unit is designed to calculate equality for the BEQ (branch if equal) instruction and inequality for the BNE (branch if not equal) instruction. This functionality helps streamline the design of the "hit" circuit by directly providing the necessary condition for a branch's success or failure, depending on whether the branch is a BEQ or BNE. As a result, the prediction hit logic can efficiently check if the prediction aligns with the actual branch outcome, improving the branch prediction accuracy and the overall performance of the design.

| Prediction | Actual | Hit |
|---|---|---|
| Not Taken | Not Taken | 1 |
| Not Taken | Taken | 0 |
| Taken | Not Taken | 0 |
| Taken | Taken | 1 |

Our prediction Signal is Design that its only true if and only if there is a backwards branch instruction, and it is false otherwise.

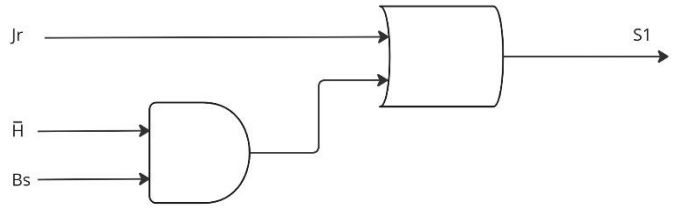
To ensure the correctness of the branch prediction mechanism and prevent potential issues, we need to propagate the Branch Signal back to the Branching Unit. This is crucial to avoid a situation where the prediction hit is true, but there is no actual branch instruction in the Execute stage. Without this propagation, the system could incorrectly process the branch prediction as valid when there is no branch operation to execute, leading to unpredictable behavior.

Additionally, we must retain the Jr (Jump Register) signal, which was previously part of the Jump Unit's functionality, to maintain the existing jump functionality. The Jr signal will continue to be propagated, ensuring that any jump instructions are handled appropriately, even with the integration of the Branching Unit. This allows us to support both branching and jumping within the same framework, improving the system's flexibility while preserving the expected behavior of both mechanisms.

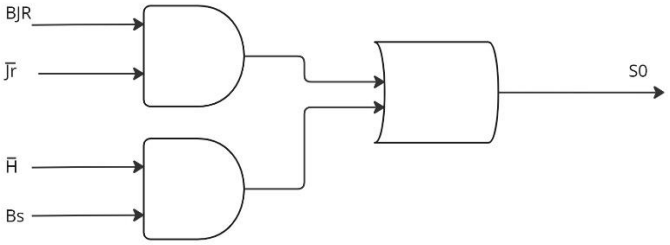
The following Table is what determines the selector lines for the fetch stage multiplexer:

| | H | Bs | JR | BJR | S1 | S0 | F |
|---|---|---|---|---|---|---|---|
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **1** | **0** | **0** | **0** | **1** | **0** | **1** | **0** |
| **2** | **0** | **0** | **1** | **0** | **1** | **0** | **1** |
| **3** | **0** | **0** | **1** | **1** | **1** | **0** | **1** |
| **4** | **0** | **1** | **0** | **0** | **1** | **1** | **1** |
| **5** | **0** | **1** | **0** | **1** | **1** | **1** | **1** |
| **6** | **0** | **1** | **1** | **0** | **X** | **X** | **X** |
| **7** | **0** | **1** | **1** | **1** | **X** | **X** | **X** |
| **8** | **1** | **0** | **0** | **0** | **0** | **0** | **0** |
| **9** | **1** | **0** | **0** | **1** | **0** | **1** | **0** |
| **10** | **1** | **0** | **1** | **0** | **1** | **0** | **1** |
| **11** | **1** | **0** | **1** | **1** | **1** | **0** | **1** |
| **12** | **1** | **1** | **0** | **0** | **0** | **0** | **0** |
| **13** | **1** | **1** | **0** | **1** | **0** | **1** | **0** |
| **14** | **1** | **1** | **1** | **0** | **X** | **X** | **X** |
| **15** | **1** | **1** | **1** | **1** | **X** | **X** | **X** |

S1: $S1 = J_r + \overline{H}.B_s$

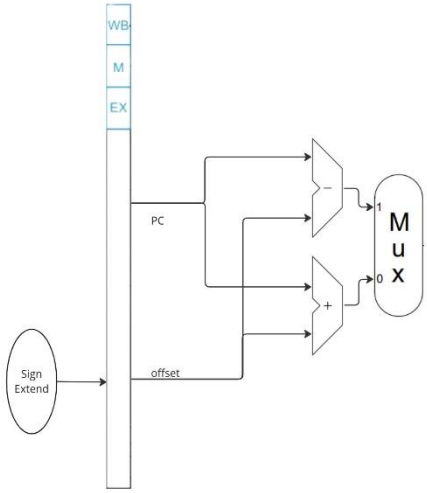


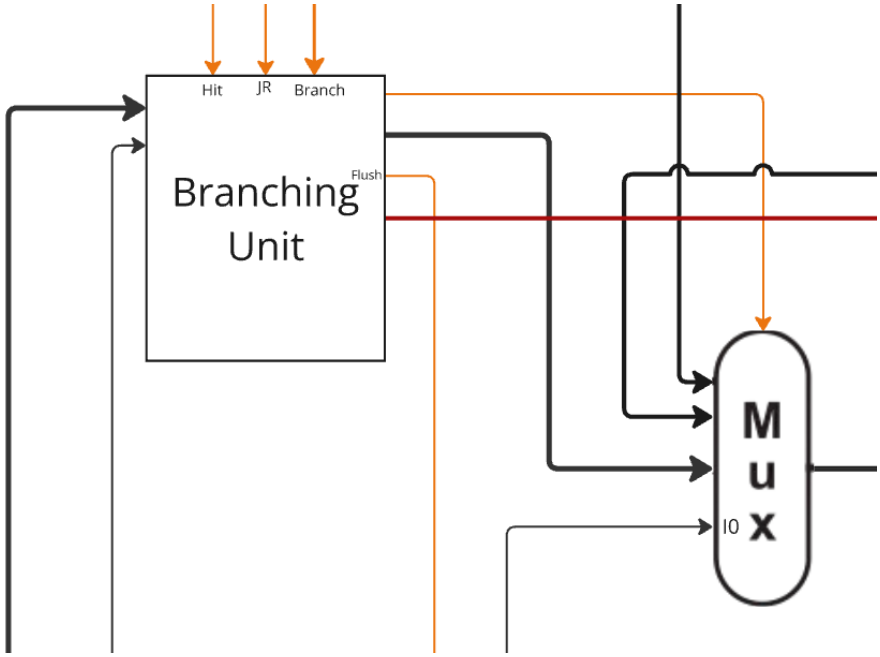

S0 : $S0 = \overline{J_r}.BJR + \overline{H}B_s$




When a branch prediction is missed, there are two cases to consider. One case occurs when the branch is taken and is forward. Initially, the assumption is made that the branch is not taken, so execution proceeds to PC + 1. However, if it is later determined that the branch should be taken, the target label address must be calculated. This is achieved using an adder that computes the target address with the formula (Next-PC + Offset). This ensures the program correctly redirects execution to the branch target address. In this case, the selector of the multiplexer is set to 0 to choose the value from the adder.

Another case occurs when the branch is not taken and is backward. Initially, the assumption is made that the branch is taken, causing execution to jump to the target label. However, if it is determined during the execute stage that the branch is not taken, the program must return to the next sequential address (PC + 1) following the branch instruction. To achieve this, a subtractor is used to compute the correct address using the formula (PC label - Offset). This ensures accurate redirection of the execution flow. In this case, the selector of the multiplexer is set to 1 to choose the value from the subtractor.



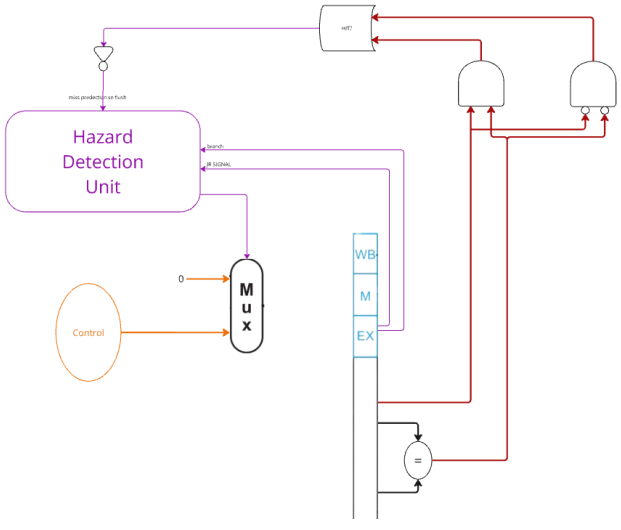



# HAZARRD

To handle branch mispredictions effectively, we need to modify the hazard detection unit. When a misprediction occurs, we must flush the ID/EX pipeline register to discard any instructions that are incorrectly speculated. This can be accomplished by inverting the Hit signal from the Branching Unit. When the Hit signal indicates a mismatch between the prediction and the actual branch outcome, the hazard detection unit detects the misprediction and initiates the pipeline flush.

However, it is also crucial to ensure that the hazard detection unit only flushes the pipeline in the case of an actual branch instruction. This requires the hazard unit to verify that there is a branch instruction in the Execute stage. If a branch is not present, the hazard detection unit should not flush the pipeline falsely.

Furthermore, we retain the Jr signal from the Jump Unit. If a JR (Jump Register) instruction is present in the Execute stage, we need to flush the instructions that were fetched before it, as jumping could alter the control flow in a way that makes previous instructions irrelevant. By maintaining the Jr signal, we ensure that the Jump functionality is correctly handled, alongside branch prediction and flush logic. This approach keeps the pipeline and control flow synchronized, ensuring the correct instructions are executed.



## Evaluation

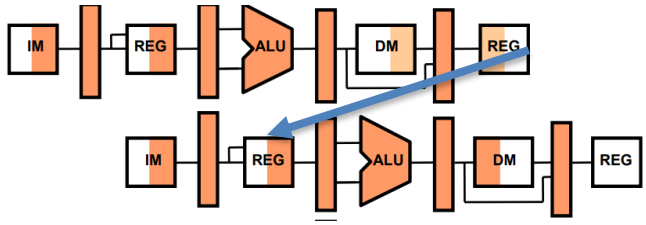
In this section we will outline the various steps taken to achieve the final design result, detailing the processes, considerations, and decisions that shaped its development.

Mainly, the pipeline has main issues like dependency between instruction, to solve this problem we have different unit like forwarding unit and hazard detection Unit. This unit help design to make the right decision when have different dependencies between instruction like (RAW dependency).

The main function of the forwarding unit in a pipeline is to resolve data hazards that occur when instructions depend on the results of previous instructions still in the pipeline. For example:

Add $t5, $t2, $5

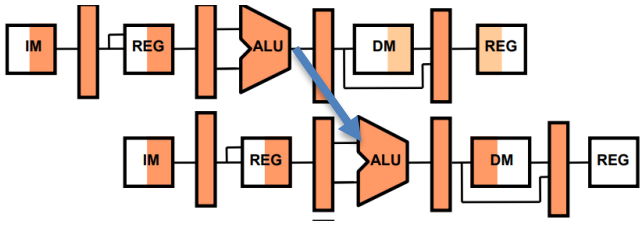
Addi $t6, $t5, 0xA




The given instructions, Add and Addi exhibit a data dependence in a pipeline because the second instruction depends on the result of the first. In a pipelined processor, this creates a read-after-write (RAW) hazard, as the second instruction cannot proceed until the first instruction has completed writing the result to $t5.

| *INST* | *T0* | *T1* | *T2* | *T3* | *T4* | *T5* | *T6* | *T7* |
|---|---|---|---|---|---|---|---|---|
| *Add* | F | D | EXE | MEM | WB | | | |
| *stall* | | nop | nop | nop | nop | nop | | |
| *stall* | | | nop | nop | nop | nop | nop | |
| *Addi* | | | | F | D | EXE | MEM | WB |

Without mechanisms like forwarding, the pipeline would need to stall until $t5 is ready, requiring the insertion of a bubble after fetching any new instruction that depends on the previous result. While this ensures correct execution, it increases the overall execution time of the process by introducing delays between dependent instructions.



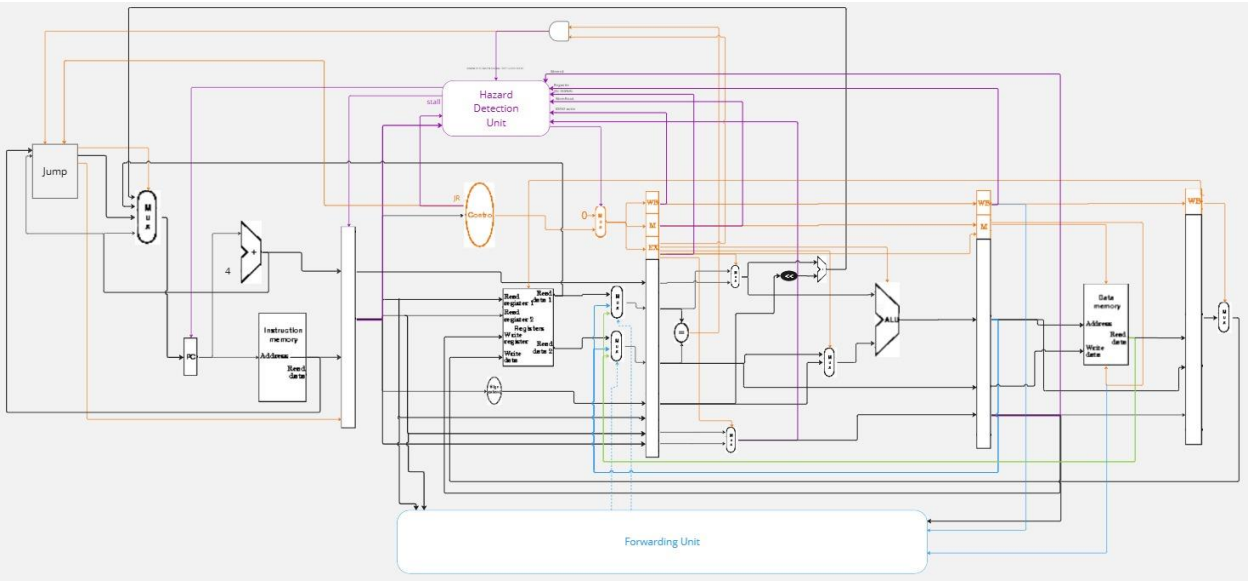

However, with forwarding, the pipeline can bypass the intermediate register write stage and directly provide the output of the first instruction to the input of the second, minimizing delays and maintaining high throughput.

To implement this approach, a reference line is typically used from the EXE_MEM pipeline register (for the ALU result) to the ALU input. The forwarding unit detects if the instruction in the execute stage requires data from the previous instruction and selects the appropriate input for the ALU through a multiplexer. For more details go to ()

Forwarding mux in decode:

In the beginning, the execute stage takes long time to end correct result, because have one or more critical path it plays a major role in calculating the frequency.

Moving the forwarding multiplexer to the decode stage can significantly reduce the execution stage time and increase the overall pipeline frequency. By handling data hazards earlier in the pipeline, the forwarding logic resolves dependencies during instruction decoding rather than waiting until execution. This reduces the complexity and critical path delay in the execute stage, enabling it to process instructions more quickly. As a result, the pipeline can achieve a higher clock frequency since the execute stage no longer dictates the longest timing path. This optimization improves overall performance by allowing faster processing cycles while maintaining accurate data flow between instructions.



This design achieves a frequency of 82.63 MHz without utilizing a forwarding line from the ALU but faces an issue with the I-Type instruction in the data path. To address this problem, it is crucial to first understand the function of the forwarding unit—specifically, how it handles data dependencies and maintains the correct flow of information between pipeline stages:

| *INST* | *T0* | *T1* | *T2* | *T3* | *T4* | *T5* | *T6* | *T7* |
|---|---|---|---|---|---|---|---|---|
| *Add $t5, $t2, $5* | F | D | EXE | MEM | WB | | | |
| *stall* | | nop | nop | nop | nop | nop | | |
| *And $t6, $t5, $2* | | | F | D | EXE | MEM | WB | |

Removing this line is necessary to raise the frequency; however, this change introduces an additional cycle loss to address the dependency problem effectively.

When testing this design, we encountered an issue with I-type instructions. To explain this problem, we need to analyze it step by step. The hazard detection unit identifies RAW dependencies by checking if the destination register (Rd) in the execute stage matches the source register (Rs or Rt) in the decode stage. For I-type instructions, the rt register serves as the destination.

When an I-type instruction is in the decode stage and does not depend on the value of rt being modified by an instruction in the execute stage, no bubble is needed. However, the current design mistakenly generates a bubble in this situation, causing unnecessary stalls in the pipeline.

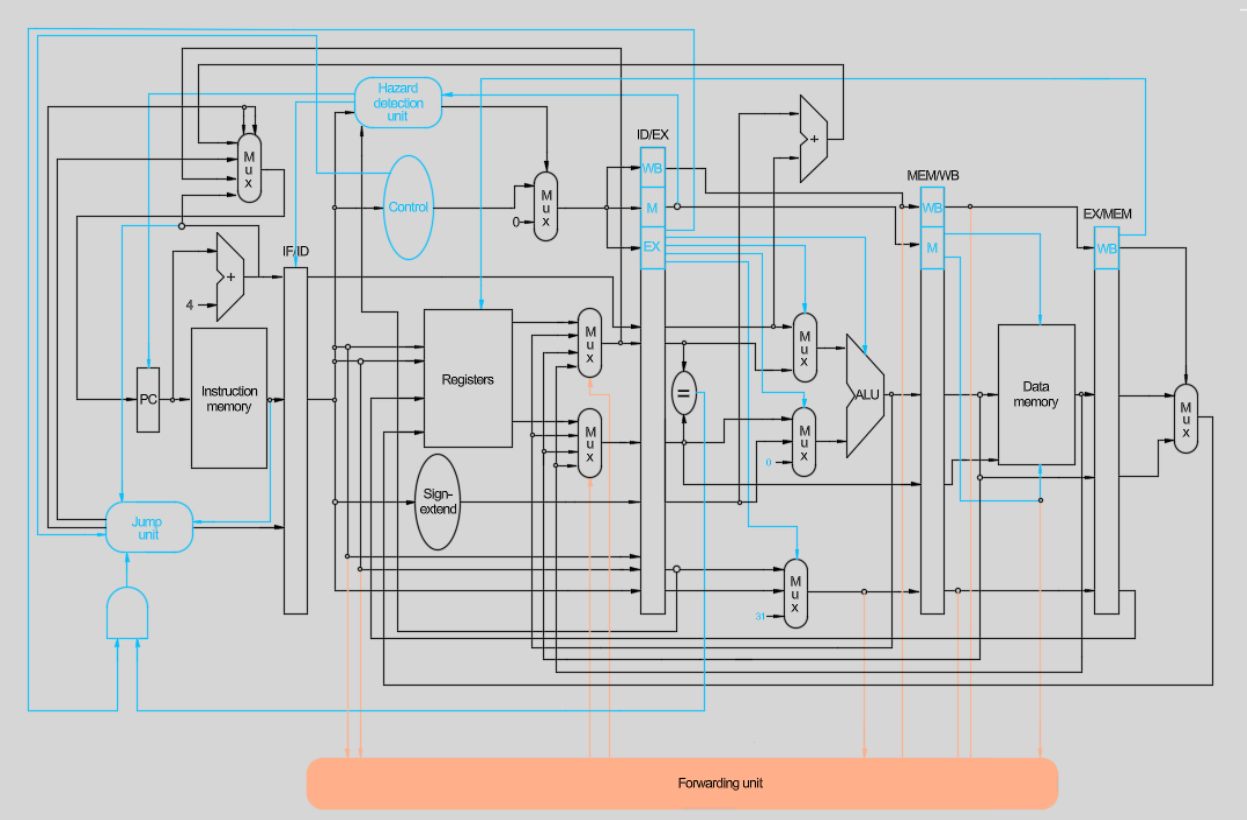
| *INST* | *T0* | *T1* | *T2* | *T3* | *T4* | *T5* | *T6* | *T7* |
|---|---|---|---|---|---|---|---|---|
| *Add $t5, $t2, $t3* | F | D | EXE | MEM | WB | | | |
| *stall* | | nop | nop | nop | nop | nop | | |
| *Addi $t5, $t6, 0xA* | | | F | D | EXE | MEM | WB | |

By default, there is no dependency between the two instructions, Add $t5, $t2, $t3 and Addi $t5, $t6, 0xA, because $t5 in the Addi instruction is the destination register. We are not concerned if $t5 is being modified by the instruction in the execute stage. However, the hazard detection unit incorrectly interprets $t5 in the decode stage as rt, not as a destination register. This misinterpretation leads to a faulty decision to send a bubble, resulting in an unnecessary cycle loss.

Due to this issue, we must disregard this design and focus on finding a solution to the problem while considering the frequency constraints, which will be explained later.

## Forwarding mux in decode and alu reference:

This design achieves a frequency of 86.48 MHz, the forwarding multiplexer in the decode stage and the ALU reference are key components for resolving data dependencies in the pipeline. By moving the forwarding logic to the decode stage, the processor can detect and address data hazards earlier, minimizing delays in later stages. Additionally, connecting the ALU output to the forwarding multiplexer in the decode stage helps resolve the issues discussed in the previous section.




## Memory negative edge clock:

The design in the previous section activates the Data-Memory at the negative edge of the clock to allow the Lw instruction to read data in the same clock cycle without any issues. However, the line connecting the Data-Memory output to the forwarding multiplexer forms a critical path in the design. This is because it only has half a clock cycle to complete, making it challenging to meet the timing requirements and synchronize with the next clock cycle for ID-EXE register pipeline.

In FPGA designs, RAM memory IP often uses registers for input to ensure stable and efficient operation. Registers act as a buffer, capturing the input signals at a specific clock edge, which helps synchronize data flow and eliminate issues like signal glitches or timing mismatches. By registering the inputs, the design achieves better timing control, ensuring that the memory operates reliably even at high clock frequencies. Additionally, using registers for inputs allows the memory to meet strict setup and hold time requirements, reducing the risk of errors during read and write operations. This practice is particularly beneficial in FPGA environments, where maintaining precise timing and predictable behavior is critical for overall system performance.

In this design, the EXE-MEM pipeline register connects its output to the Data-Memory, requiring two cycles to read data from memory. To optimize this, the Data-Memory is activated on the positive clock edge, allowing a single cycle to retrieve data. To achieve this, the input to the Data-Memory is connected directly from the execute stage. Additionally, we assume the input register for the Data-Memory is part of the EXE-MEM pipeline register, enabling the memory to complete the read operation within one cycle on the positive clock edge.

Additionally, we activates MEM_WB pipeline register at the negative edge clock, and register file at positive edge clock. This approach achieve frequency for design to **99.38MHz.**

## Execution Time:

In this section, we compare two designs: the first design, which serves as the main design discussed in the pipeline chapter, and the second design, described in the previous section.

Execution time in a pipeline refers to the total time required to process a series of instructions, leveraging the pipeline's staged structure to overlap the execution of multiple instructions. Unlike a non-pipelined system, where each instruction completes all stages sequentially, a pipelined system allows different instructions to be in different stages simultaneously. The execution time for a single instruction is determined by the duration of the longest stage (the pipeline's clock cycle). However, for multiple instructions, the execution time is significantly reduced due to concurrent stage operation once the pipeline is filled.
The efficiency of a pipeline depends on its ability to minimize stalls, handle dependencies effectively, and maintain a balanced workload across all stages. In this section, we calculate and compare the execution times for the two designs to evaluate their performance under these criteria.
To calculate the execution time for this design, we need to consider the number of stalls introduced for each instruction and the total number of instructions executed by the processor. Stalls occur due to hazards or dependencies between instructions, which can delay the pipeline.

In general, the equation to calculate the execution time:

$$execution_{time} = \frac{clock\ cycles}{frequancy}$$

$$clock\ cycles = (\#\ of\ cycle\ to\ fill\ pipeline) + (\#\ of\ instruction)$$

Clock cycle: refer to the number of cycle need to finish the program
frequency: frequency for the design

If we have stall in the program we need to add number of stalls to clock cycle:

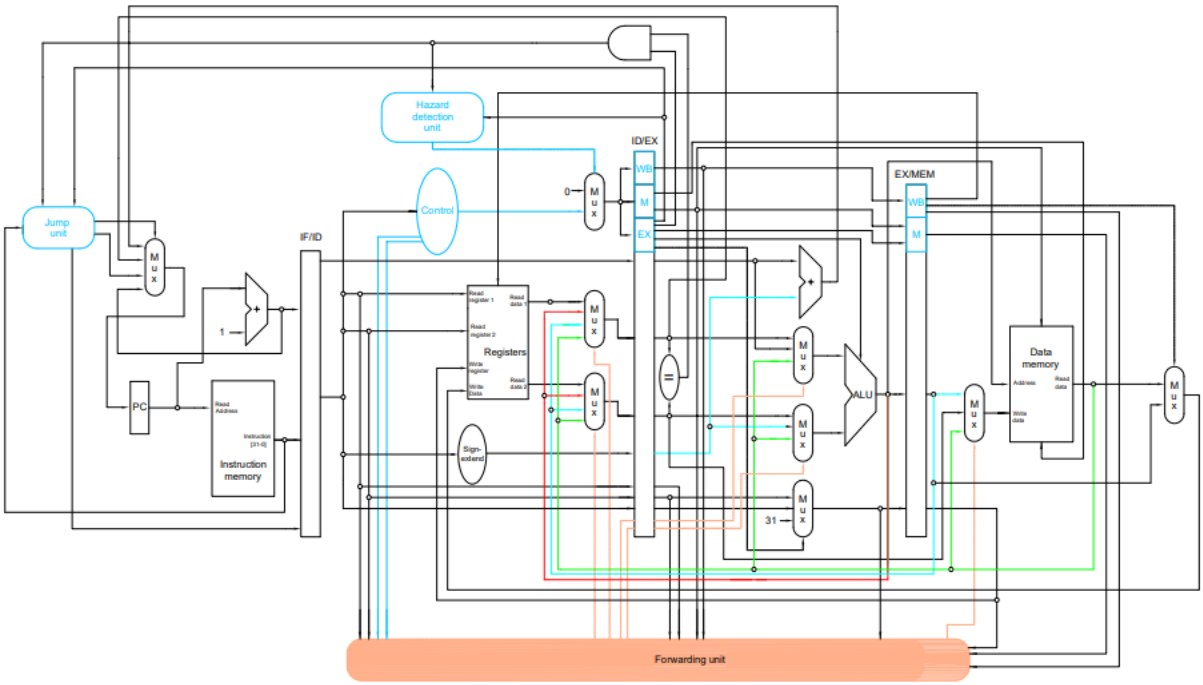
$$clock\ cycles = (\#\ of\ cycle\ to\ fill\ pipeline) + (\#\ of\ instruction) + (\#\ of\ bubble)$$

Finial equation:

$$execution_{time} = \frac{clock\ cycles}{frequancy} = \frac{(\#of\ cycle\ to\ fill\ pipeline) + (\#\ of\ instruction) + (\#of\ bubble)}{frequancy}$$

Now, we will calculate and compare the execution time for both designs to highlight the differences and determine which design performs better under specific conditions. By analyzing the number of stalls, instruction count, and pipeline behavior for each design, we can identify scenarios where one design outperforms the other. This comparison will demonstrate the strengths and weaknesses of each design, helping to clarify which is more efficient and under what circumstances.

**Design 1:**




The first design has a unique advantage over the second design, especially in programs that require frequent memory access. These programs often involve repeated memory operations, with subsequent instructions depending on the data read from memory. This creates a dependency between the memory access instruction and the instruction that follows, which relies on the retrieved data during the execute stage. The first design effectively addresses this dependency issue by resolving the need for bubbles in such cases, ensuring smoother execution and improved performance.

To calculate the execution time for the first design, we use a general equation format. However, it is essential to identify and explain the instructions that generate bubbles during execution. Bubbles occur when there are dependencies or hazards that temporarily stall the pipeline.

$$number\ of\ bubble = (Y)\ \times\ \alpha \times 2$$

2: number of bubble for each branch and JR instruction
Y = number of instructions in the program
α = percentage of branch and jump-register instructions in the program.

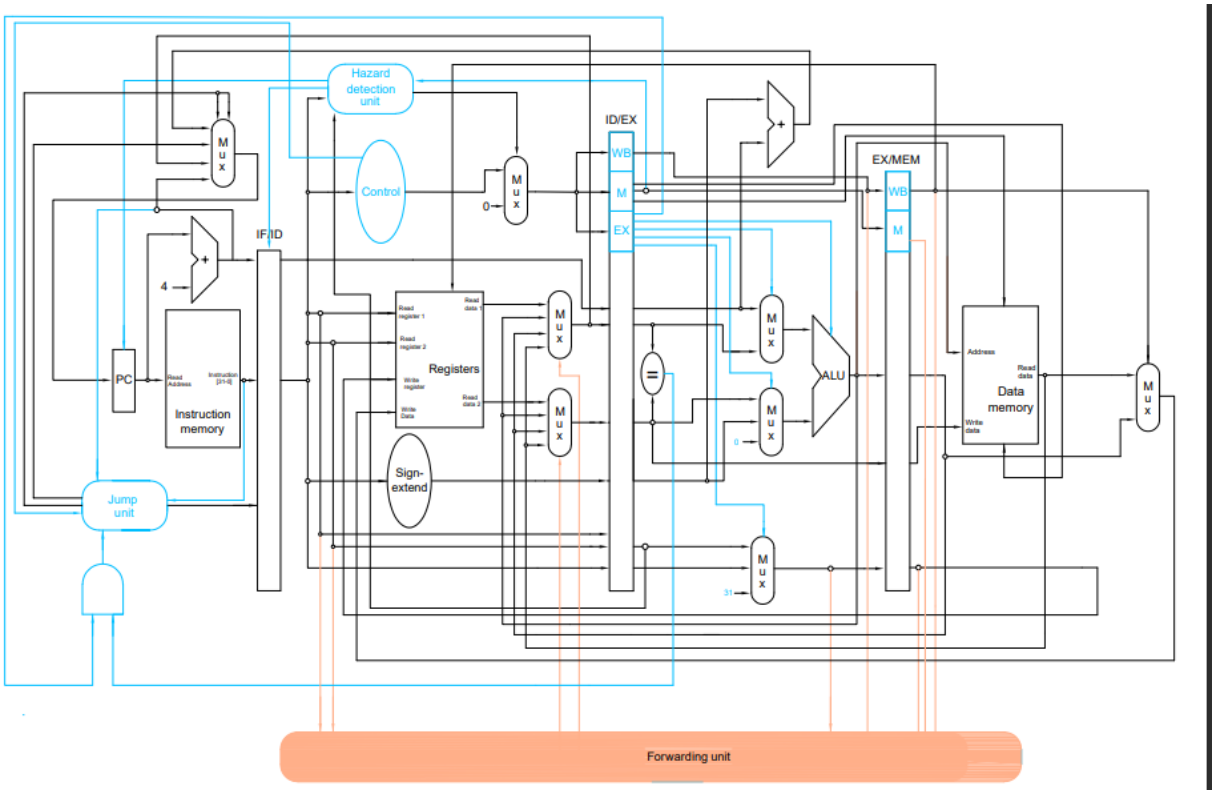
$$\alpha = \%of\ branch + \%of\ JR$$

To calculate α, we need to determine the number of branch and jump-register (JR) instructions in the program. For example, if a program consists of 100 lines, with 10 branch instructions and 2 JR instructions, the percentage of branch and JR instructions is calculated as follows: (10 + 2) / 100. This gives α as 12%.

The final equation for execution time for first design:

$$execution_{time} = \frac{clock\ cycles}{frequancy} = \frac{(\#of\ cycle\ to\ fill\ pipeline) + Y(1 + 2\alpha)}{frequancy}$$

**Design2:**




The second design is faster than the first, but it introduces more bubbles due to dependencies between instructions that rely on data fetched from memory. Specifically, when an instruction accesses data memory, it creates a dependency for subsequent instructions that need that data, leading to additional pipeline stalls.

To calculate the execution time for the first design, we use a general equation format. However, it is essential to identify and explain the instructions that generate bubbles during execution. Bubbles occur when there are dependencies or hazards that temporarily stall the pipeline.

$$number\ of\ bubble = (Y \times \alpha \times 2) + (Y \times \beta)$$

2: number of bubbles for each branch and JR instruction
Y = number of instructions in the program
α = percentage of branch and jump-register instructions in the program.
β =percentage of dependencies caused by load instructions in the program.

$$\beta = \%of\ lw\ instruction\ \times \%of\ lw\ dependency$$

To calculate β in this equation, we need to determine the number of load instructions in the program and the number of dependencies caused by these load instructions. For example, if a program contains 100 instructions, with 15 LW instructions, and 10 dependencies caused by load instructions, we first calculate the percentage of LW instructions as 15/100. Next, we calculate the percentage of LW dependencies as 10/15. Finally, β is calculated as (15 × 0.6666), which means that the number of bubbles caused by LW dependencies is 10 bubbles.

The final equation for execution time for first design:

$$execution_{time} = \frac{clock\ cycles}{frequancy} = \frac{(\#of\ cycle\ to\ fill\ pipeline) + Y(1 + 2\alpha + \beta)}{frequancy}$$

**Comparison:**

In both designs, branch and jump-register (JR) instructions are handled in the same way, resulting in two bubbles for each instruction. To simplify the comparison, we will assume these bubbles can be ignored in the equation. This equation is intended to demonstrate the conditions under which Design 1 performs better than Design 2.

$$E_1 < E_2 \rightarrow \frac{4 + X}{F_1} < \frac{4 + X(1 + \beta)}{F_2}$$

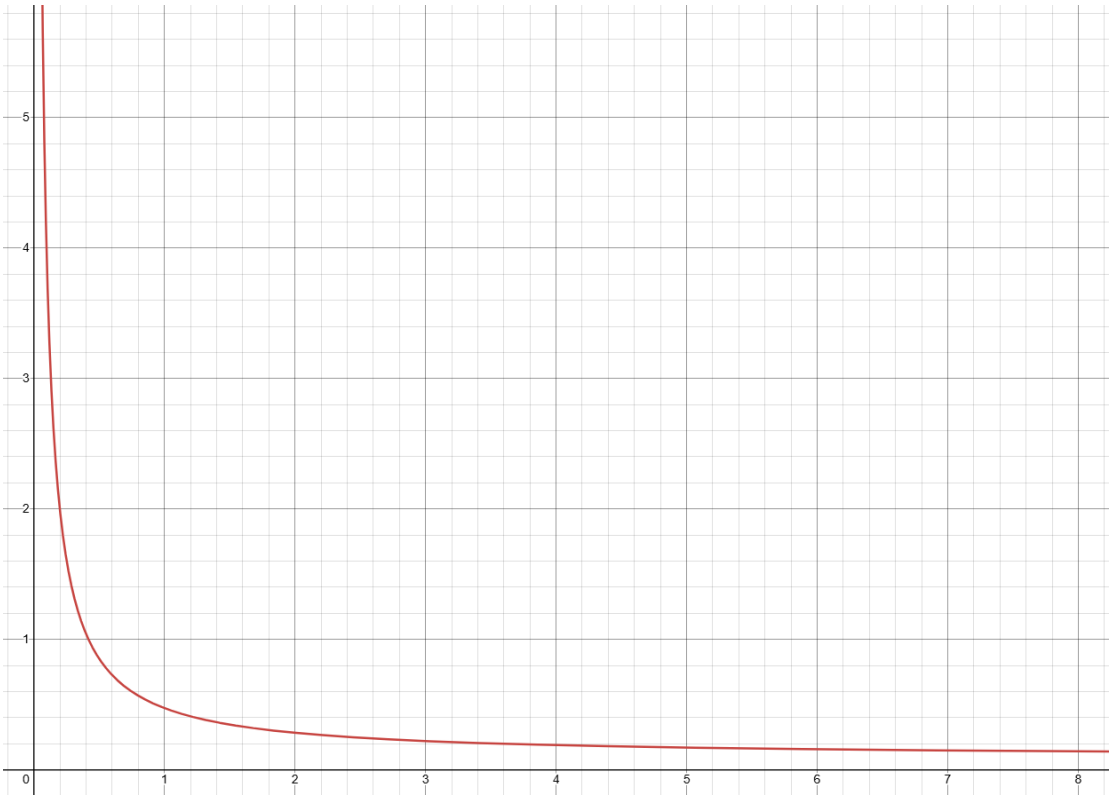
**$F_1$ =90.77, $F_2$ = 99.37**

At the point, where Design 1 performs better than Design 2, the comparison depends on the percentage of the load instruction dependencies (β). To determine this value, we need to calculate the point at which both designs have the same execution time. By finding this critical value of β, we can identify the threshold where the performance of the two designs is equal and determine under what conditions Design 1 becomes more efficient than Design 2.

$$\beta \geq \frac{0.379}{X} + 0.0947$$

To evaluate the limit as x→∞:

$$\beta = \lim_{X \to \infty} \left( \frac{0.379}{X} + 0.0947 \right) = 0.0947$$



## Speed Up

**Performance comparison : pipeline vs single-cycle execution**

**Amdahl's law** (or **Amdahl's argument**)

> ***" the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used "***

**Single cycle VS Pipeline processor**

**1 Single cycle**

A **single-cycle** processor executes an entire instruction—comprising several steps such as instruction fetch, decode, execute, memory access, and write-back—within one clock cycle. while straightforward in design, this architecture has significant limitations due to its reliance on the longest critical path, resulting in a slow clock speed and reduced efficiency for complex instructions

the time required to complete a single clock cycle in a single-cycle processor is determined by the **critical path**, which represents the longest sequence of operations needed to execute the most complex instruction. for example, an instruction that involves accessing memory or performing complex arithmetic may require significantly more time than simpler instructions. because the clock cycle must be long enough to accommodate the slowest instruction.

**1.1 performance challenges in single-cycle processors**

**1.1.1 low clock speed**: the processor's clock speed is limited by the longest I instruction in the instruction set.

**1.1.2 wasted potential**: simpler instructions, which could complete much faster, are forced to wait for the entire clock cycle to finish.

**1.2 limitations in clock speed and throughput**

**1.2.1 low clock speed**: the need to account for the longest instruction reduces the frequency of the clock, making the processor slower overall.

**1.2.2 limited throughput**: since the processor can only execute one instruction per clock cycle, its instruction throughput is inherently low.

**1.2.3 inefficiency in resource utilization**: resources like arithmetic units, memory, and registers are idle during parts of the cycle when not actively used by the current instruction.

## 2 pipeline processor

A **pipelined** processor is designed to execute multiple instructions **simultaneously** by dividing the instruction execution process into multiple stages. these stages typically include instruction fetch, decode, execute, memory access, and write-back. as one instruction moves through each stage, other instructions are simultaneously processed at different stages of the pipeline. this overlapping of instructions allows the processor to achieve **higher throughput** and **better resource utilization,** as multiple instructions can be in progress at the same time.

### 2.1 implications of pipeline complexity

while pipelining improves performance, it introduces additional complexity to the processor's design. the stages of the pipeline must be carefully managed, and coordination between stages is critical. several challenges arise from this complexity:

**2.1.1 pipeline hazards**: when instructions depend on the results of previous instructions or cause conflicts in execution, leading to **delays**. these hazards include:

**2.1.1.1 data hazards**: when an instruction depends on the result of a previous instruction that hasn't yet completed.

**2.1.1.2 control hazards**: caused by branch instructions that alter the flow of execution.

**2.1.1.3 structural hazards**: when insufficient resources are available to execute multiple instructions at the same time.

**2.1.2 pipeline overhead**: the additional hardware and control logic required to manage the pipeline stages and resolve hazards adds **extra complexity**. this overhead can reduce the overall performance improvement, especially when stalls or flushes are needed due to hazards.

### 2.2 limitations in performance

**2.2.1 complex design**: the need for managing multiple stages and handling hazards results in a more intricate design than a single-cycle processor.

**2.2.2 overhead**: the additional hardware and control mechanisms required for pipelining can consume resources, sometimes diminishing the benefits.

**2.2.3 hazard management**: resolving hazards may require inserting pipeline stalls or flushing the pipeline, which can reduce the overall speedup.

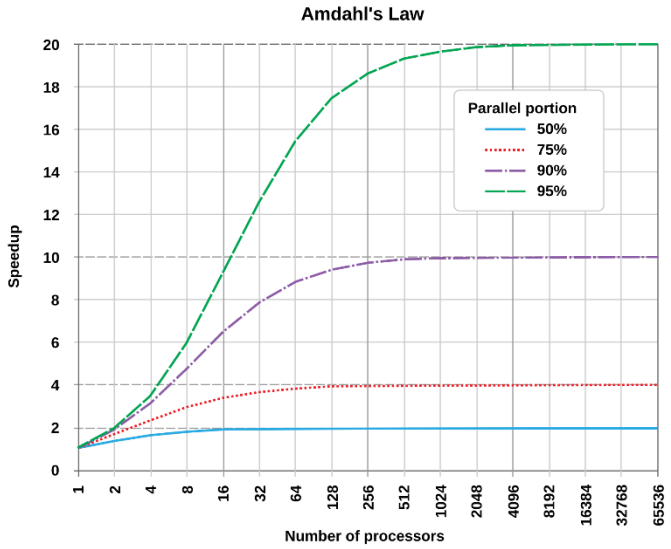
## Speed UP

**Speedup** is a key metric used to assess the performance improvement achieved by optimizing a system or adding additional resources, such as processors or memory. it is calculated as the ratio of the time required to complete a task on a single resource to the time required to complete the same task on multiple resources or an optimized system

The concept of **speedup** is particularly important in parallel computing, where the goal is to reduce execution time by dividing a task among multiple processors. in an ideal scenario, known as **linear speedup**, the performance improvement is directly proportional to the number of resources added. for example, doubling the number of processors would ideally halve the execution time

However, in practical applications, achieving linear speedup is rare due to factors like communication overhead, synchronization delays, and resource contention. these factors can lead to sublinear speedup, where the performance improvement is less than proportional to the resources added

understanding speedup is crucial for evaluating whether optimization efforts are effective and justifiable. it helps in identifying the limitations of a system and determining the point at which adding more resources yields diminishing returns this makes it an essential tool for designing and analyzing efficient computing systems




### Speedup Analysis

#### Formula and Calculation

$$speedup = \frac{execution\ time\ (single-cycle)}{execution\ time\ (pipelined)}$$

$$\text{execution time} = \frac{\#\ of\ cycles\ for\ the\ program}{Frequency}$$

**In the single cycle processor every instruction takes the same clock cycle time to execute :**

$$\text{execution time(single cycle)} = \frac{\#\ of\ instruction\ in\ the\ program}{Frequency\ (single\ cycle)}$$

---

**In the pipeline processor , after the pipeline is filled one instruction is completed per clock cycle :**

$$\text{execution time (pipelined)} = \frac{4+\ \#\ of\ instructions\ \ for\ the\ program}{Frequency\ (pipelined)}$$

---

### 1.2 Measure Execution Times

The first step is to determine the execution time for both the single-cycle and pipelined processors:

#### 1.2.1 Single-cycle execution time:

**Tsingle=Instruction Count × Tcycle_single**

Where **Tcycle_single** the clock cycle time of the single-cycle processor.

#### 1.2.2 Pipelined execution time:

**Tpipeline = Tfill + (Instruction Count−1) × Tcycle_pipeline + Tdrain**

where:

**Tfill** is the time taken to fill the pipeline (typically Pipeline Stages × Tcycle_pipeline )

**Tdrain** is the time to drain the pipeline (flushes or incomplete execution cycles).

**Tcycle_pipeline** is the clock cycle time of the pipelined processor.

---

### 1.3 Use the Speedup Formula

Once the execution times are known, practical speedup is calculated as:

$$Speedup = \frac{\text{Tsingle}}{\text{Tpipeline}}$$

Alternatively, when clock frequencies are used instead of cycle times:

$$Speedup = \frac{\text{F single} \times \text{IPC single}}{\text{F pipeline} \times \text{IPC pipeline}}$$

Where:

- **F pipeline** and **F single** are the clock frequencies of the pipelined and single-cycle processors, respectively.
- **IPC pipeline** and **IPC single** (Instructions Per Cycle) reflects the instruction throughput.

---

### 1.4 Evaluate Pipeline Utilization

Pipeline utilization is a critical metric for assessing practical speedup. It reflects how efficiently the pipeline stages are utilized:

$$\text{Utilization} = \frac{\text{Total CyclesUseful}}{\text{Instruction Executions}}$$

Low utilization (e.g., due to frequent stalls or flushes) reduces practical speedup.

---

## 2 Calculation

- **F single = 30.9 MHz**
- **F pipeline = 83.36 MHz**
- **Pipeline Stages = 5**
- **Branch Prediction Accuracy = 90%**
- **Forwarding Unit: Reduces data stalls by 70%**

### 2.1 Ideal Speedup

ideal speedup in a pipelined processor assumes that the processor achieves perfect utilization of all pipeline stages.

**Speedup (Ideal) = Pipeline Stages = 5**

---

### 2.2 Practical Speedup

**Practical speedup** reflects the real-world performance improvement of a pipelined processor compared to a single-cycle processor. It accounts for the effects of pipeline overhead, hazards, and branch penalties

- Pipeline overhead: Reduces clock speed slightly.
- Pipeline hazards: Introduces 10% stalls.
- Branch penalties: Causes a 10% slowdown due to mispredictions.

#### 2.2.1 Calculations for cycle time

$$\text{T cycle\_single} = \frac{1}{30.9\ x\ 10^{6}} \approx 32.36 \text{ ns}$$

$$\text{T pipeline} = \frac{1}{83.36\ x\ 10^{6}} \approx 12.00 \text{ ns}$$

**Thus, the pipelined processor has a shorter cycle time due to the higher clock frequency, but the overhead is included in this calculation**

---

#### 2.2.2 Pipeline Hazards

Pipeline hazards introduce stalls, reducing throughput.

- Data hazards: The forwarding unit reduces data stalls by **70%** but **30%** of data hazards still cause pipeline stalls. Assume **15%** of cycles are stalled due to data hazards.
- Control hazards: Branch prediction accuracy is **90%** so **10%** of branch instructions result in pipeline flushes. If **20%** of instructions are branches, the overall control stall rate is:

**Control Stall Rate = 0.10 × 0.20 = 0.02 (2% of cycles stalled)**

**total stall rate: Adding the stall rates**

**Total Stall Rate = 0.15 + 0.02 = 0.17 (17% of cycles stalled )**

The effective throughput of the pipelined processor is reduced by this stall rate**:**

**Throughput pipeline=Pipeline Stages×(1−Total Stall Rate)=5×(1−0.17) = 4.15 IPC (practical)**

---

### 2.2.3 Branch Penalties

Each branch misprediction causes the pipeline to flush, adding delays proportional to the number of pipeline stages. For a 5-stage pipeline and a 10% misprediction rate:

**Branch Penalty (cycles)=Pipeline Stages × Misprediction Rate = 50 x 10=0.5 cycles per branch**

Including this penalty, the throughput drops slightly further.

---

### 2.2.4 Practical Speedup Calculation

Using the adjusted values for throughput:

**2.2.4.1 Single-cycle execution time per instruction**:

$$\text{T instruction\_single} = \text{T cycle\_single} \times 1 = 32.36\text{ns.}$$

**2.2.4.2 Pipelined execution time per instruction (after filling)**:

$$\text{T instruction\_pipeline} = \text{T cycle\_pipeline} \times \frac{1}{\text{Throughput pipeline}}$$

$$\text{T instruction\_pipeline} = 12.00\text{ns} \times \frac{1}{4.151} \approx 2.89\text{ns.}$$

**2.2.4.3 Speedup**

$$\text{Speedup practical} = \frac{\text{T instruction\_pipeline}}{\text{T instruction\_single}}$$

$$\text{Speedup practical} = \frac{32.36}{2.89} \approx 3.5$$

This indicates that the pipelined processor, operating at a higher frequency, has the potential to achieve significantly faster execution times than the single-cycle processor.

## Conclusion:

In conclusion, the development of the pipelined processor, along with the cycle-accurate simulator and assembler, provides a solid foundation for exploring the complexities of modern processor design. By transitioning from the single-cycle architecture to a pipelined design, we were able to significantly improve performance, with the added branch prediction mechanism enhancing efficiency further in loop-intensive programs. The careful analysis of different pipeline designs and the use of performance benchmarks underscored the importance of balancing clock frequency and pipeline efficiency to meet the needs of real-world applications. The cycle-accurate simulator and assembler have proven to be indispensable tools in refining the processor design, enabling accurate testing, and ensuring the correctness of the implementation. This work lays the groundwork for future exploration of advanced processor features and optimizations, and demonstrates the power of pipelining in improving processor performance.

## References:

CMOS VLSI Design: A Circuits and Systems Perspective by Neil Weste, David Harris.

ComputerOrganizationAndDesign by David A. Patterson, John L. Hennessy.

Digital Design with an introduction to Verilog HDL by M. Morris Mano, Michael Ciletti.

Computer Architecture: A Quantitative Approach by John L. Hennessy, David A. Patterson.

Intel Quartus Timing analyzer user quide

A study of branch Prediction Strategies by JAMES E. SMITH.

Branch Predection For free by THOMAS BALL, JAMES R. LAWS

Branch Prediction Strategies and branch Target buffer desgin Johnny K. F. Lee Jay Smith,

University of California, Berkeley.

An Architectural Assessment of SPEC CPU Benchmark Relevance Benjamin C.

Lee Cache Performance for SPEC CPU2000 Benchmarks Mark D. Hill

Intel Quartus Timing analyzer user quide

Tools to visualize : AutoDesk AutoCad miro EDA Tools:

Intel Quartus Prime Simulation tools: ModelSim Intel FPGA

# APPENDIX A: Verification and Testing

**Methodologies for effective system testing**

comprehensive methodologies for verifying the correctness of single-cycle and pipeline processor designs

If it isn't tested, it doesn't work!

## 1. Objectives

- **functional correctness**: ensuring that all components of the Datapath and control unit function correctly in both single-cycle and pipelined designs, with special focus on instruction flow in pipelined systems.
- **timing validation**: verifying that all operations in both systems complete within the required clock period, ensuring single-cycle instructions fit within one cycle and pipeline stages complete without stalls.
- **signal integrity**: ensuring control and data signals propagate correctly, with no conflicts or inconsistencies, and managing data forwarding and hazards in pipelined processors.
- **resource utilization**: ensuring efficient use of hardware resources like registers, buses, and memory in both systems, avoiding contention in single-cycle processors and conflicts in pipelined systems.
- **error detection**: identifying and addressing design flaws in both systems, with additional focus on data and control hazards in pipelined processors to maintain reliable execution

## 2. Testing

### Single-Cycle Processor Testing:

**Theory**: A single-cycle processor executes each instruction in one clock cycle, meaning all components (ALU, registers, memory, etc.) must complete their operation in a single cycle. The main theory behind its design is simplicity and ease of testing.

**Testing Methods**: Initially, testing was relatively straightforward, with the focus being on verifying the correct operation of each instruction (e.g., load, store, arithmetic operations). Key testing approaches were:

**Functional Testing**: Verifying that each instruction executes correctly in one cycle.

**Boundary Testing**: Ensuring that extreme values or edge cases (such as zero or maximum integer values) are handled correctly.

**Timing Constraints**: Verifying that all components can operate within the time constraints of a single cycle.

**Challenges** The main challenge with single-cycle processors was that achieving high performance was difficult because all instructions were constrained to the same cycle time, regardless of their complexity

### Pipelined Processor Testing

**Theory**: Pipelining breaks down the execution of instructions into multiple stages (such as instruction fetch, decode, execute, memory access, and write-back). This allows multiple instructions to be processed simultaneously, increasing throughput. The theory behind pipelining is based on maximizing the use of the processor's resources and reducing idle time.

**Stages:** Each stage in the pipeline can be thought of as a mini-process, where testing needs to ensure that data flows correctly between these stages. Pipelined processors are typically tested by focusing on

**Data Hazards** : Ensuring data dependencies are correctly handled (e.g., forwarding or stalling to avoid conflicts).

**Control Hazards**: Managing branch instructions and ensuring correct program flow.

**Structural Hazards**: Ensuring that multiple instructions don't conflict for the same resources (e.g., memory or register access).

**Testing Methods**

- **Simulation-based Testing**: Testbenches are often used to simulate pipeline execution under various conditions, including checking for correct forwarding and hazard resolution.

- **Formal Verification**: Used to prove mathematically that the pipeline stages function correctly, particularly in complex pipelines with many stages.
- **Fault Injection**: Deliberately introducing faults (such as incorrect data forwarding or branch misprediction) to test how well the system can recover or handle errors.
- **Performance Testing**: Unlike single-cycle processors, where the test is mostly about correctness, pipelined processors require testing of throughput, latency, and pipeline stall conditions under different workloads.

**Challenges**: The main testing challenges with pipelined processors are:

- **Synchronization**: Ensuring that all stages work in harmony, especially when pipeline stages are not fully balanced in terms of timing.
- **Complexity**: Testing becomes more complex due to the interdependencies between pipeline stages, and the need to test multiple combinations of data, instructions, and hazards.
- **Stateful Behavior**: The need to track the state of multiple instructions simultaneously and ensure that the processor behaves correctly even when certain instructions are delayed or stalled.

**Key Milestones in Pipelined Processor Testing**

- **1970s** : Early pipelined processors (e.g., CDC 6600) used simple tests for hazard detection and data forwarding.

- **1980s-1990s**: As processors became more complex with deeper pipelines (e.g., Intel's P6 architecture), testing tools such as cycle simulators, automated testing frameworks, and formal verification methods became more common.

- **2000s-Present**: Advanced methods like model checking and RTL (register transfer level) simulation are employed for exhaustive verification of complex pipeline behaviors.

## Single Cycle Testing

Benchmark 1 – Data Manipulation Instruction and Control Flow Instructions

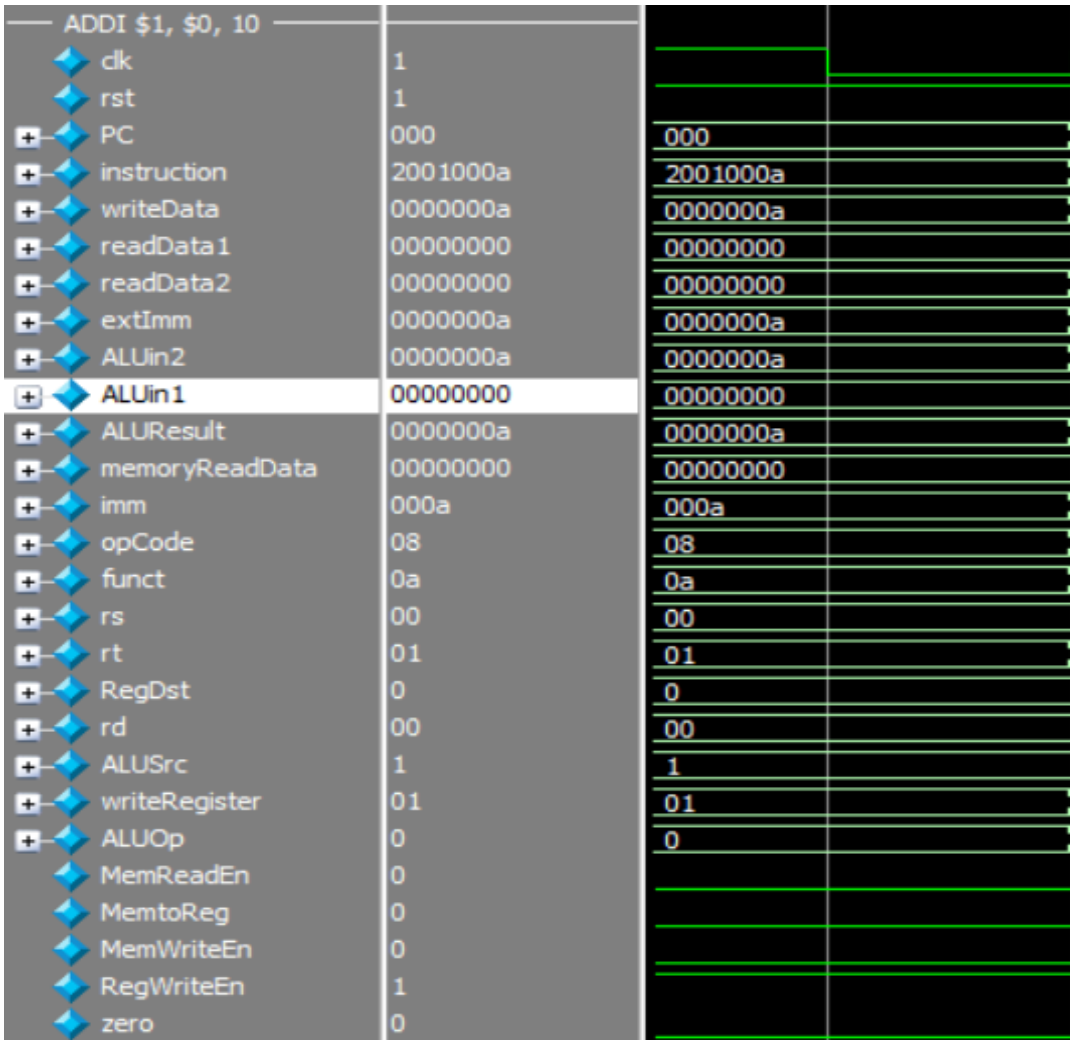
```
        .data
value:  .word 0x5          # sample data for loading
        .text
main:

# Immediate instructions to initialize registers
ADDI $1, $0, 0xA
ORI $2, $0, 0xB
XORI $3, $0, 0xC

# Basic ALU operations
ADD $4, $1, $2
SUB $5, $4, $3
AND $6, $1, $3
OR $7, $2, $3
NOR $8, $2, $3

# Comparison operations
SLT $10, $1, $2
SGT $11, $3, $1

# Shift operations
SLL $12, $1, 2
SRL $13, $2, 1

# Load and store word instructions
SW $4, 0x0($0)
LW $14, 0x0($0)

# Branch instructions

BEQ $1, $2, label_beq
BNE $2, $3, label_bne
```

```
label_beq:
ANDI $20 , $1 , 0x53
label_bne:
Nop

J label_jump          # Jump to label_jump
SLL $7,$1,3 # This will not execute due to jump
label_jump:
SLTI $10, $1, 0x13

ADDI $14,$14,0x1



# Jump Register (JR) instruction
JR $14                #Jump to the address in $14

# Jump and Link (JAL) instruction
JAL label_jal         # Jump to label_jal and store return address in $31

label_jal:
XOR $8, $0, $16

ANDI $20 , $1 , 0x53

#psudo instructions
BLTZ $1,  label_beq
BGEZ $2, label_bne
```

# Processor Testing

**Instruction 1 → ADDI $1, $0, 0xA**

**Object code → 2001000A H**



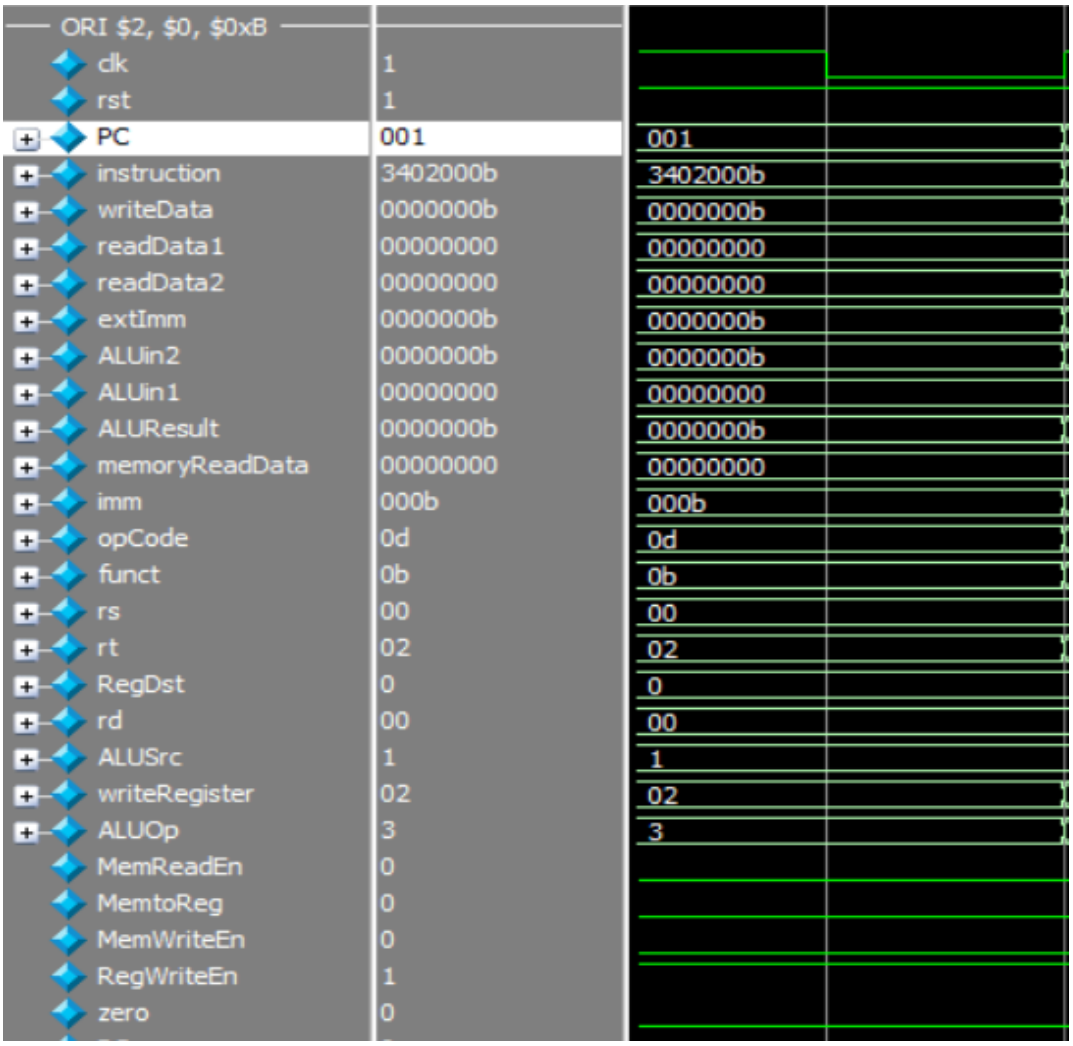

Register value after execution


When the instruction **ADDI $1, $0, 0xA** is executed in a single-cycle processor with the initial **PC** value of 0, the first step is the instruction fetch. The instruction is fetched from memory address 0, where the opcode **ADDI** is decoded. During the decode stage, the processor reads the value of register **$0** (which holds 0) and decodes the immediate value **0xA** (10 in decimal). At the positive clock edge, the ALU is configured to perform an addition operation, adding the value from register $0 (which is 0) to the immediate value 0xA. This results in the value 0xA.

In the execute stage at the positive clock edge, the ALU outputs the result 0xA, which is then passed to the write-back stage. During the write-back stage, the value **0xA** is written to register **$1**, as indicated by the RegWrite signal. The PC is incremented by 1 to point to the next instruction, so the new **PC** value becomes 1. Control signals such as **RegDst** are set to 0 (indicating the destination is specified by rt), and **ALUSrc** is set to 1, directing the ALU to use the immediate value as the second operand. No memory access occurs, so **Memory Read** and **Memory Write** signals are both disabled. The entire process is completed in a single cycle, with the relevant values updated as specified.

**Instruction 2 → ORI $2, $0, 0xB**

**Object code → 3402000B H**



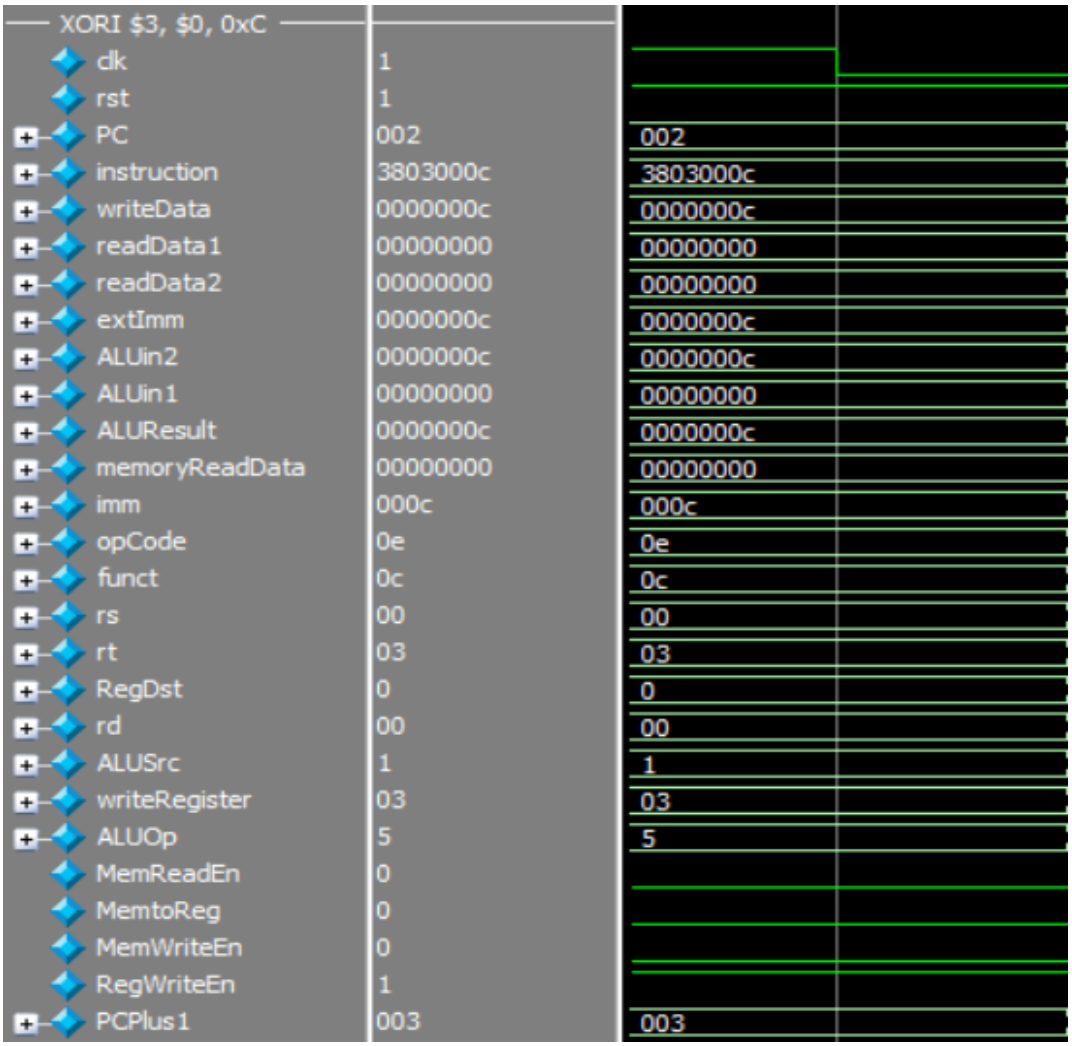

Register value after execution


When the instruction **ORI $2, $0, 0xB** is executed in a single-cycle processor with the initial **PC** value of 1, the first step is the instruction fetch. The instruction is fetched from memory address 1, where the opcode **ORI** is decoded. During the decode stage, the processor reads the value of register **$0** (which holds 0) and decodes the immediate value **0xB** (11 in decimal). At the positive clock edge, the **ALU** is configured to perform a bitwise **OR** operation, applying the OR between the value from register **$0** (which is 0) and the immediate value **0xB**. This results in the value **0xB**

.In the execute stage at the positive clock edge, the ALU outputs the result **0xB**, which is then passed to the write-back stage. During the write-back stage, the value **0xB** is written to register $2, as indicated by the RegWrite signal. The PC is incremented by 1 to point to the next instruction, so the new **PC** value becomes 2. Control signals such as **RegDst** are set to 0 (indicating the destination is specified by rt), and **ALUSrc** is set to 1, directing the **ALU** to use the immediate value as the second operand. No memory access occurs, so Memory Read and Memory Write signals are both disabled. The entire process is completed in a single cycle, with the relevant values updated as specified.

**Instruction 3 → XORI $3, $0, 0xC**

**Object code → 3803000C H**



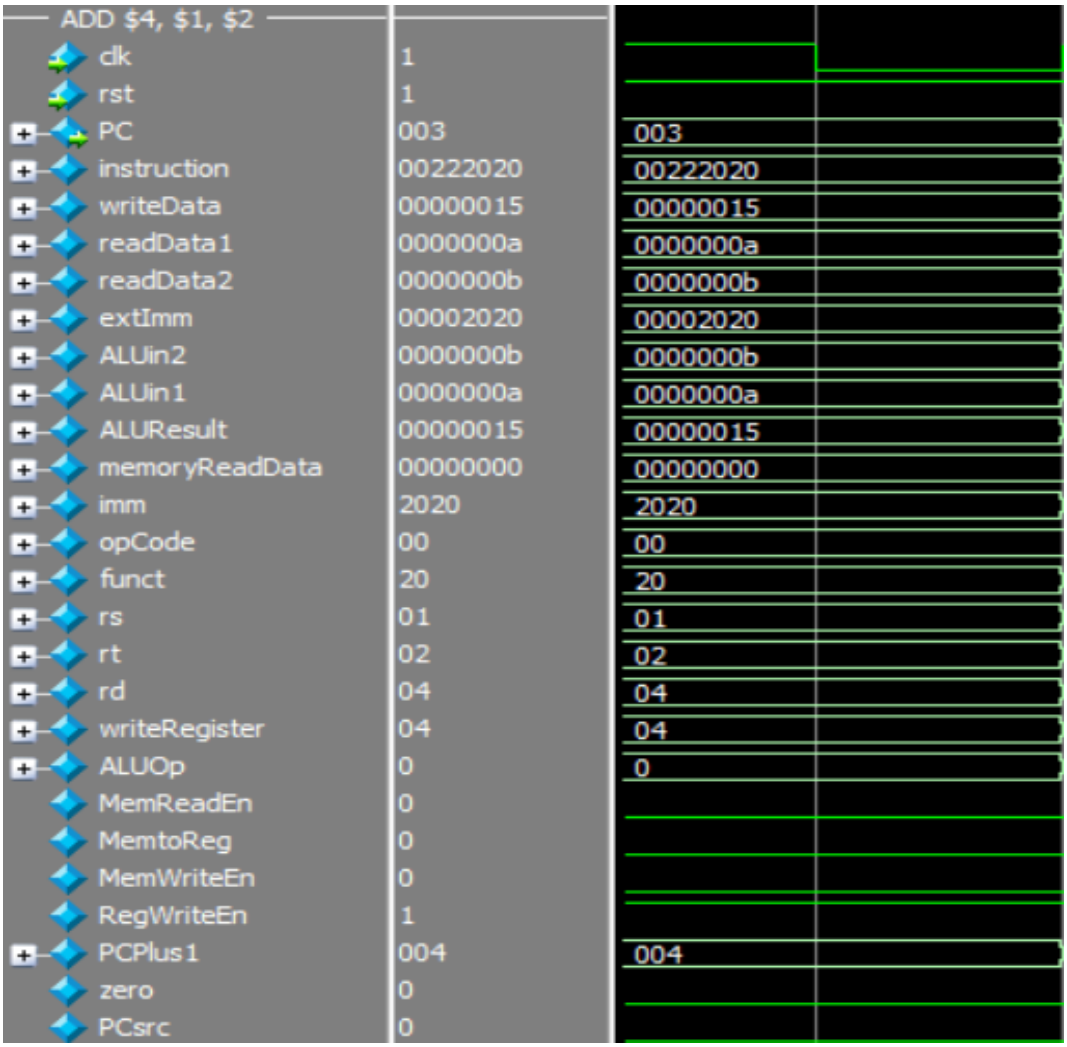

Register value after execution


When the instruction **XORI $3, $0, 0xC** is executed in a single-cycle processor with the initial **PC** value of 2, the first step is the instruction fetch. The instruction is fetched from memory address 2, where the opcode **XORI** is decoded. During the decode stage, the processor reads the value of register **$0** (which holds 0), and the immediate value **0xC** is decoded. At the positive clock edge, the **ALU** is then configured to perform a bitwise **XOR** operation, applying the **XOR** between the value from register **$0** (which is 0) and the immediate value **0xC**. This results in the value **0xC** (12 in decimal).

In the execute stage at the positive clock edge, the **ALU** outputs the result **0xC**, which is then passed to the write-back stage. During the write-back stage, the value **0xC** is written to register **$3**, as indicated by the **RegWrite** signal. The PC is incremented by 1 to point to the next instruction, so the new **PC** value becomes 3. Control signals such as **RegDst** are set to 0 (indicating the destination is specified by rt), and **ALUSrc** is set to 1, directing the **ALU** to use the immediate value as the second operand. No memory access occurs, so **Memory Read** and **Memory Write** signals are both disabled. The entire process is completed in a single cycle, with the relevant values updated as specified.

**Instruction 4 → ADD $4, $1, $2**

**Object code → 00222020 H**



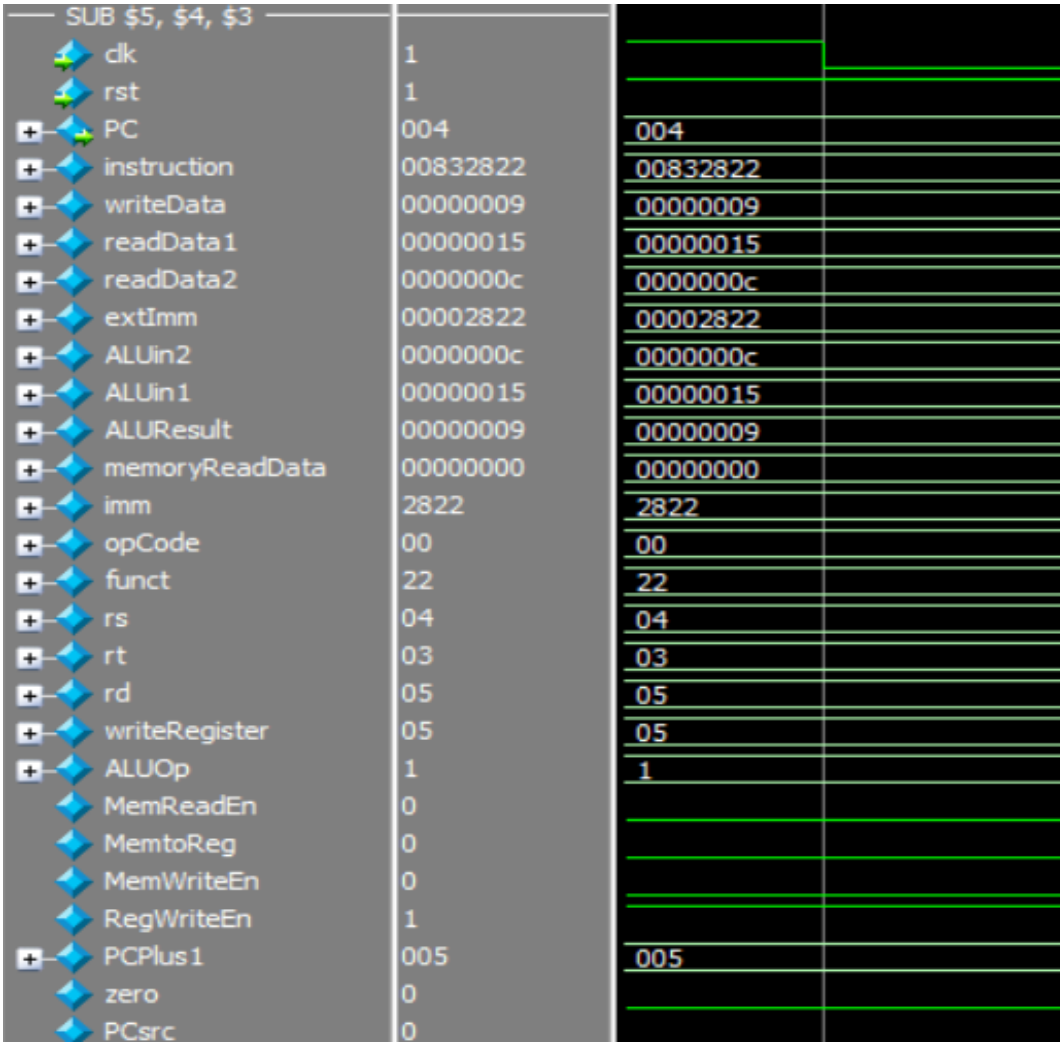

Register value after execution


When the instruction **ADD $4, $1, $2** is executed in a single-cycle processor with the initial **PC** value of 3, the first step is the instruction fetch. The instruction is fetched from memory address 3, where the opcode **ADD** is decoded. During the decode stage, the processor reads the values of registers **$1** (which holds **0xA**) and **$2** (which holds **0xB**). At the positive clock edge, the ALU is configured to perform an addition operation, adding the values from registers **$1** and **$2**. This results in the value **0x15** (21 in decimal).

In the execute stage at the positive clock edge, the ALU outputs the result **0x15**, which is then passed to the write-back stage. During the write-back stage, the value **0x15** is written to register **$4**, as indicated by the **RegWrite** signal. The PC is incremented by 1 to point to the next instruction, so the new **PC** value becomes 4. Control signals such as **RegDst** are set to 1 (indicating the destination is specified by rd), and ALUSrc is set to 0, directing the **ALU** to use the register value as the second operand. No memory access occurs, so **Memory Read** and **Memory Write** signals are both disabled. The entire process is completed in a single cycle, with the relevant values updated as specified.

**Instruction 5 → SUB $5, $4, $3**

**Object code → 00832822**



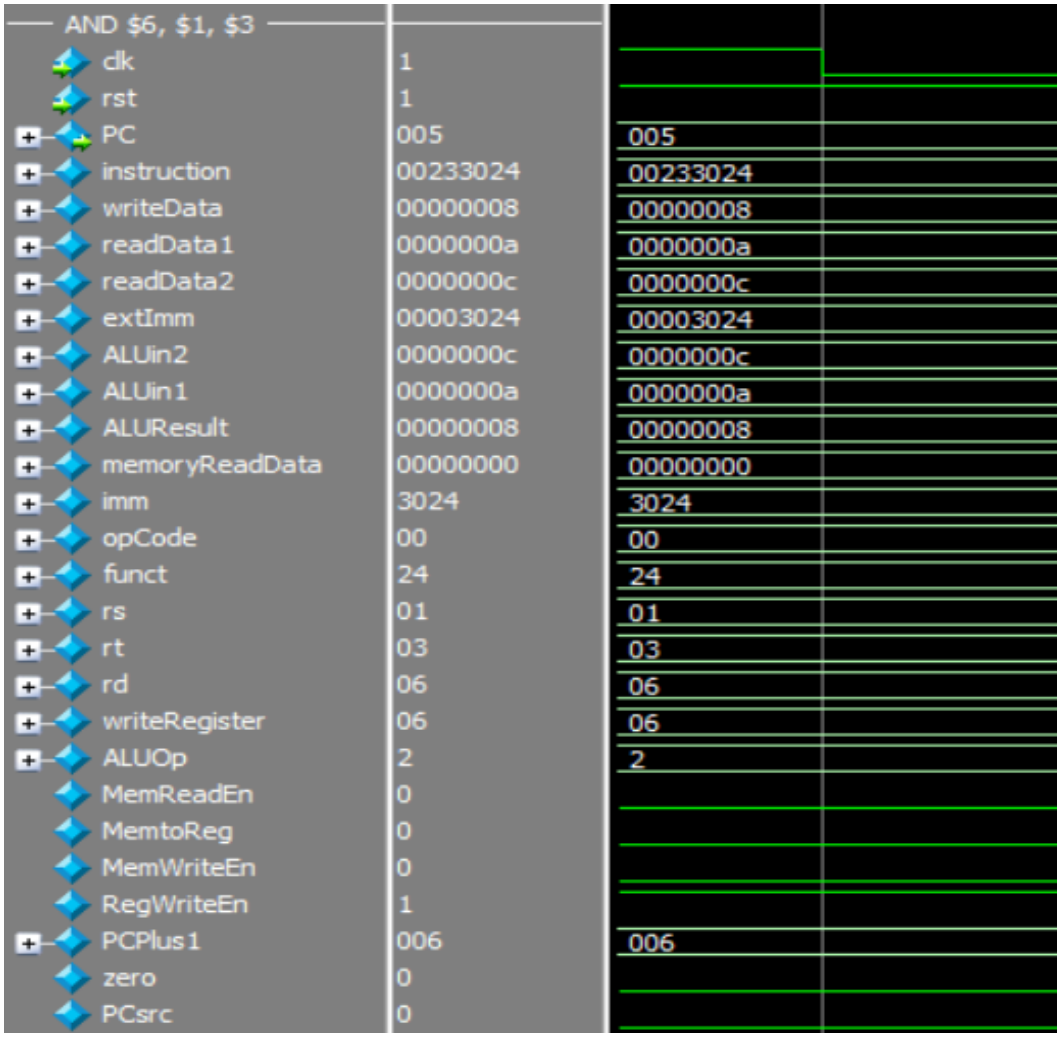

Register value after execution


When the instruction **SUB $5, $4, $3** is executed in a single-cycle processor with the initial PC value of 4, the first step is the instruction fetch. The instruction is fetched from memory address 4, where the opcode SUB is decoded. During the decode stage, the processor reads the values of registers **$4** (which holds 0x15) and **$3** (which holds 0xC). At the positive clock edge, the ALU is configured to perform a subtraction operation, subtracting the value in $3 from the value in **$4**. This results in the value **0x9** (9 in decimal).

In the execute stage at the positive clock edge, the ALU outputs the result 0x9, which is then passed to the write-back stage. During the write-back stage, the value 0x9 is written to register $5, as indicated by the RegWrite signal. The PC is incremented by 1 to point to the next instruction, so the new PC value becomes 5. Control signals such as RegDst are set to 1 (indicating the destination is rd), and ALUSrc is set to 0, directing the ALU to use the second register value as the operand. No memory access occurs, so Memory Read and Memory Write signals are both disabled. The entire process is completed in a single cycle, with the relevant values updated as specified.

**Instruction 6 → AND $6, $1, $3**

**Object code → 00233024**



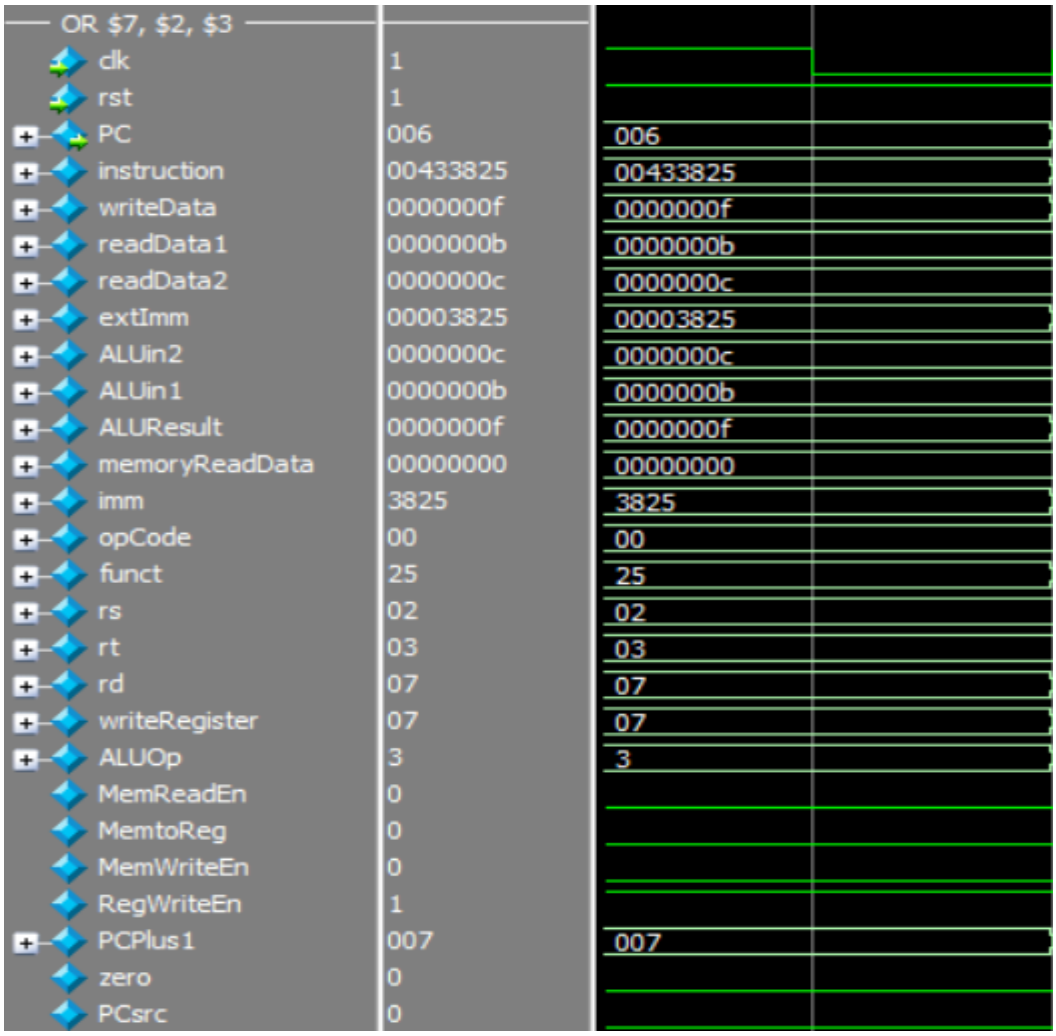

Register value after execution


When the instruction AND $6, $1, $3 is executed in a single-cycle processor with the initial PC value of 5, the first step is the instruction fetch. The instruction is fetched from memory address 5, where the opcode AND is decoded. During the decode stage, the processor reads the values of registers $1 (which holds 0xA) and $3 (which holds 0xC). At the positive clock edge, the ALU is configured to perform a bitwise AND operation between the values in $1 and $3. This results in the value 0x8 (8 in decimal).

In the execute stage at the positive clock edge, the ALU outputs the result 0x8, which is then passed to the write-back stage. During the write-back stage, the value 0x8 is written to register $6, as indicated by the RegWrite signal. The PC is incremented by 1 to point to the next instruction, so the new PC value becomes 6. Control signals such as RegDst are set to 1 (indicating the destination is specified by rd), and ALUSrc is set to 0, directing the ALU to use the register value as the second operand. No memory access occurs, so Memory Read and Memory Write signals are both disabled. The entire process is completed in a single cycle, with the relevant values updated as specified.

**Instruction 7 → OR $7, $2, $3**

**Object code → 00433825**



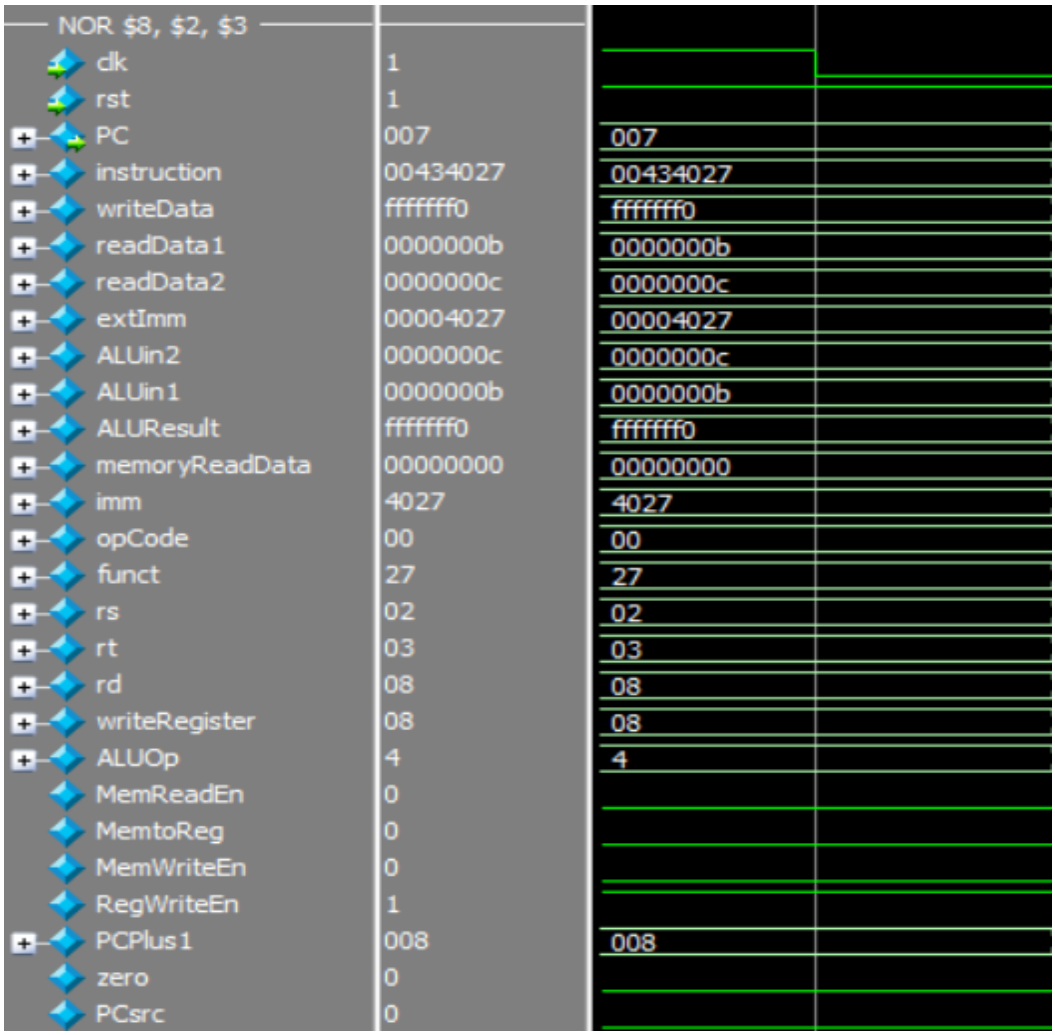

Register value after execution


When the instruction **OR $7, $2, $3** is executed in a single-cycle processor with the initial PC value of 6, the first step is the instruction fetch. The instruction is fetched from memory address 6, where the opcode **OR** (bitwise OR) is decoded. During the decode stage, the processor reads the values of registers $2 (which holds 0xB) and $3 (which holds 0xC). At the positive clock edge, the ALU performs the OR operation to compute the result as 0xB OR 0xC = 0xF.

In the execute stage at the positive clock edge, the result of the operation (0xF) is prepared for the write-back stage. During the write-back stage, the value 0xF is written to register $7, as indicated by the RegWrite signal. The PC is incremented by 1 to point to the next instruction, so the new PC value becomes 7. Control signals such as RegDst are set to 1 (indicating that the destination register is rd), and ALUSrc is set to 0, directing the ALU to use the two register values as operands. The Memory Read and Memory Write signals are both disabled since no memory access is required.

The entire process is completed in a single cycle, with the relevant values updated as specified. Register $7 now holds 0xF, and the PC points to the next instruction.

**Object code → 00434027**

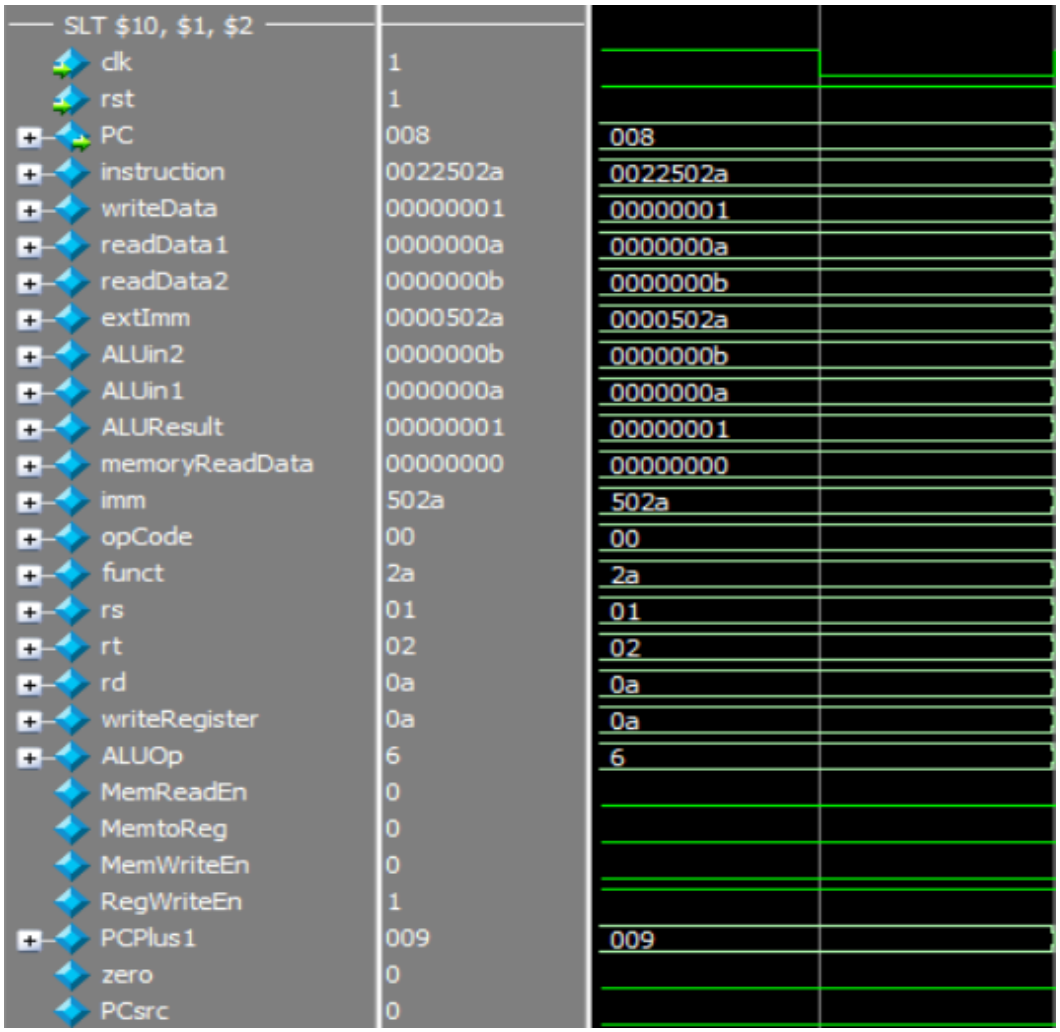



## Register value after execution


When the instruction **NOR $8, $2, $3** is executed in a single-cycle processor with the initial PC value of 7, the first step is the instruction fetch. The instruction is fetched from memory address 7, where the opcode **NOR** (bitwise NOR) is decoded. During the decode stage, the processor reads the values of registers $2 (which holds 0xB) and $3 (which holds 0xC). At the positive clock edge, the ALU performs the NOR operation to compute the result as ~(0xB OR 0xC) = ~0xF = 0xFFFFFFF0 (in 32-bit representation).

In the execute stage at the positive clock edge, the result of the operation (0xFFFFFFF0) is prepared for the write-back stage. During the write-back stage, the value 0xFFFFFFF0 is written to register $8, as indicated by the RegWrite signal. The PC is incremented by 1 to point to the next instruction, so the new PC value becomes 8. Control signals such as RegDst are set to 1 (indicating the destination register is rd), and ALUSrc is set to 0. The Memory Read and Memory Write signals remain disabled.

**Object code → 0022502A**



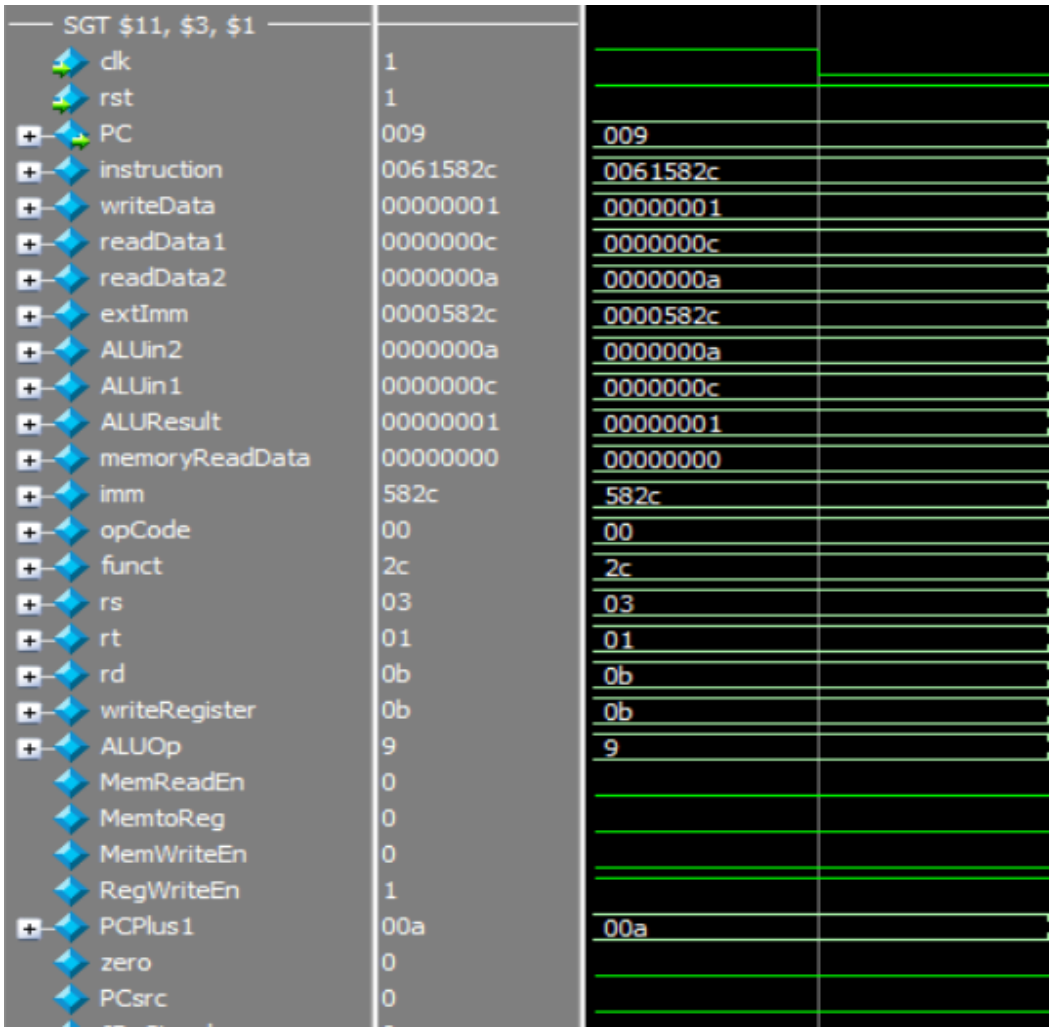

Register value after execution


When the instruction **SLT $10, $1, $2** is executed with the initial PC value of 8, the instruction is fetched from memory address 8, where the opcode **SLT** (Set on Less Than) is decoded. During the decode stage, the processor reads the values of registers $1 (which holds 0xA) and $2 (which holds 0xB). At the positive clock edge, the ALU performs the comparison and sets the result to 1 since 0xA < 0xB.

In the execute stage at the positive clock edge, the result of the comparison (1) is prepared for the write-back stage. During the write-back stage, the value 1 is written to register $10. The PC is incremented by 1, so the new PC value becomes 9. Control signals are set appropriately for the ALU operation, and no memory access is required.

**Instruction 10 → SGT $11, $3, $1**

**Object code → 0061582C**



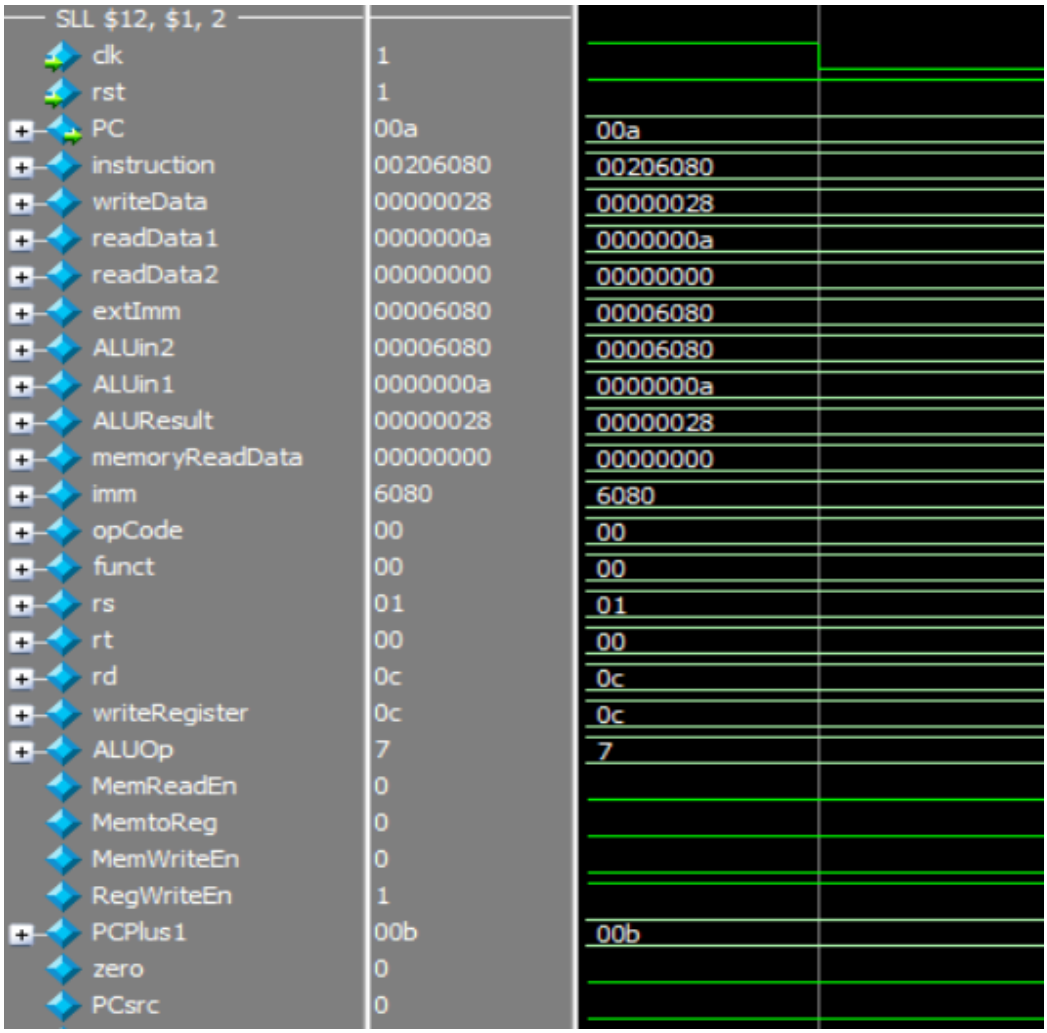

Register value after execution


When the instruction SGT $11, $3, $1 is executed with the initial PC value of 9, the instruction is fetched from memory address 9, where the opcode SGT (Set on Greater Than) is decoded. During the decode stage, the processor reads the values of registers $3 (which holds 0xC) and $1 (which holds 0xA). At the positive clock edge, the ALU performs the comparison and sets the result to 1 since 0xC > 0xA.

In the execute stage at the positive clock edge, the result of the comparison (1) is prepared for the write-back stage. During the write-back stage, the value 1 is written to register $11. The PC is incremented by 1, so the new PC value becomes 10. Control signals are set for the ALU operation, and no memory access is required.

**Instruction 11 → SLL $12, $1, 2**

**Object code → 00206080**



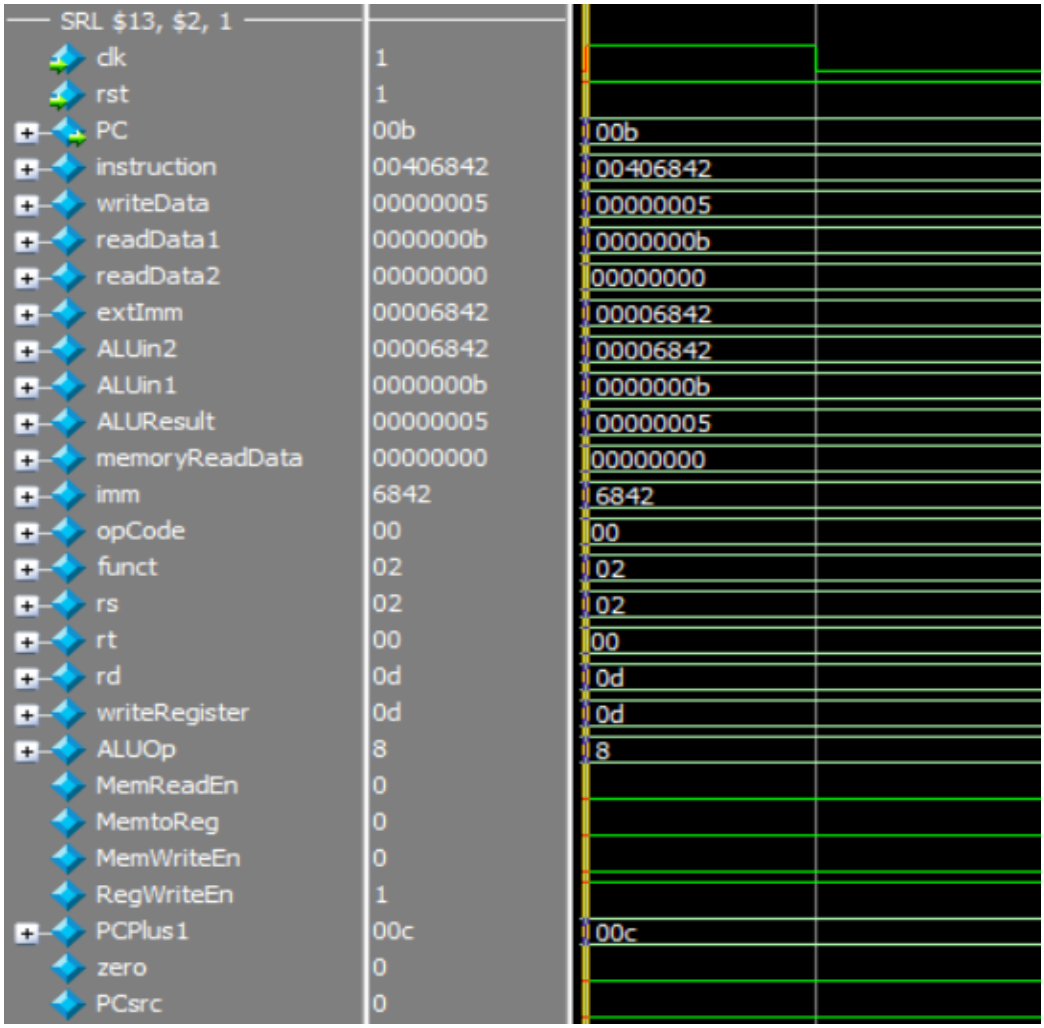

Register value after execution


When the instruction **SLL $12, $1, 2** is executed with the initial PC value of 10, the instruction is fetched from memory address 10, where the opcode **SLL** (Shift Left Logical) is decoded. During the decode stage, the processor reads the value of register $1 (which holds 0xA) and the shift amount (2). At the positive clock edge, the ALU performs the shift operation to compute the result as 0xA << 2 = 0x28.

In the execute stage at the positive clock edge, the result of the shift operation (0x28) is prepared for the write-back stage. During the write-back stage, the value 0x28 is written to register $12. The PC is incremented by 1, so the new PC value becomes 11. Control signals are set appropriately for the shift operation, and no memory access is required.

**Instruction 12 → SRL $13, $2, 1**

**Object code → 00406842**



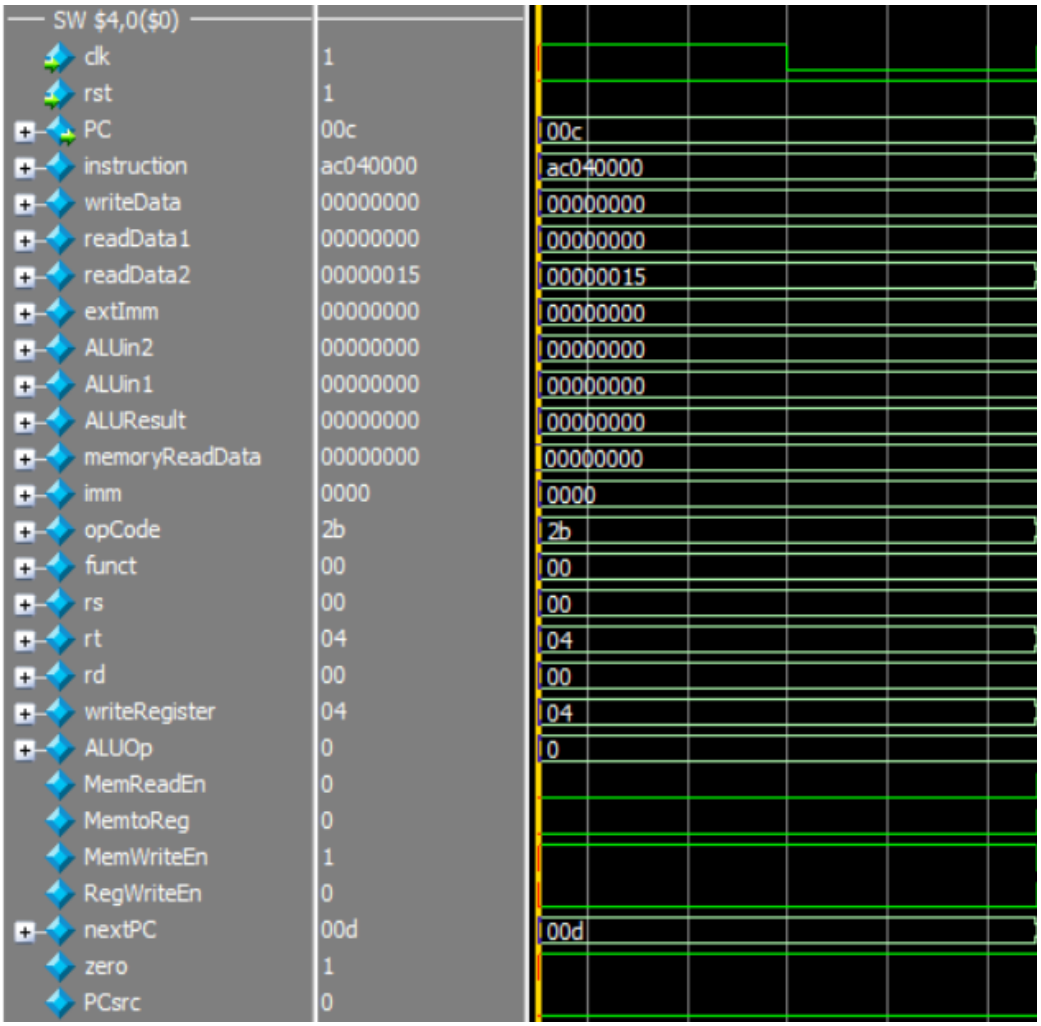

Register value after execution


When the instruction SRL $13, $2, 1 is executed with the initial PC value of 11, the instruction is fetched from memory address 11, where the opcode **SRL** (Shift Right Logical) is decoded. During the decode stage, the processor reads the value of register $2 (which holds 0xB) and the shift amount (1). At the positive clock edge, the ALU performs the shift operation to compute the result as 0xB >> 1 = 0x5.

In the execute stage at the positive clock edge, the result of the shift operation (0x5) is prepared for the write-back stage. During the write-back stage, the value 0x5 is written to register $13. The PC is incremented by 1, so the new PC value becomes 12. Control signals are set appropriately for the shift operation, and no memory access is required.

**Instruction 13 → SW $4, 0($0)**

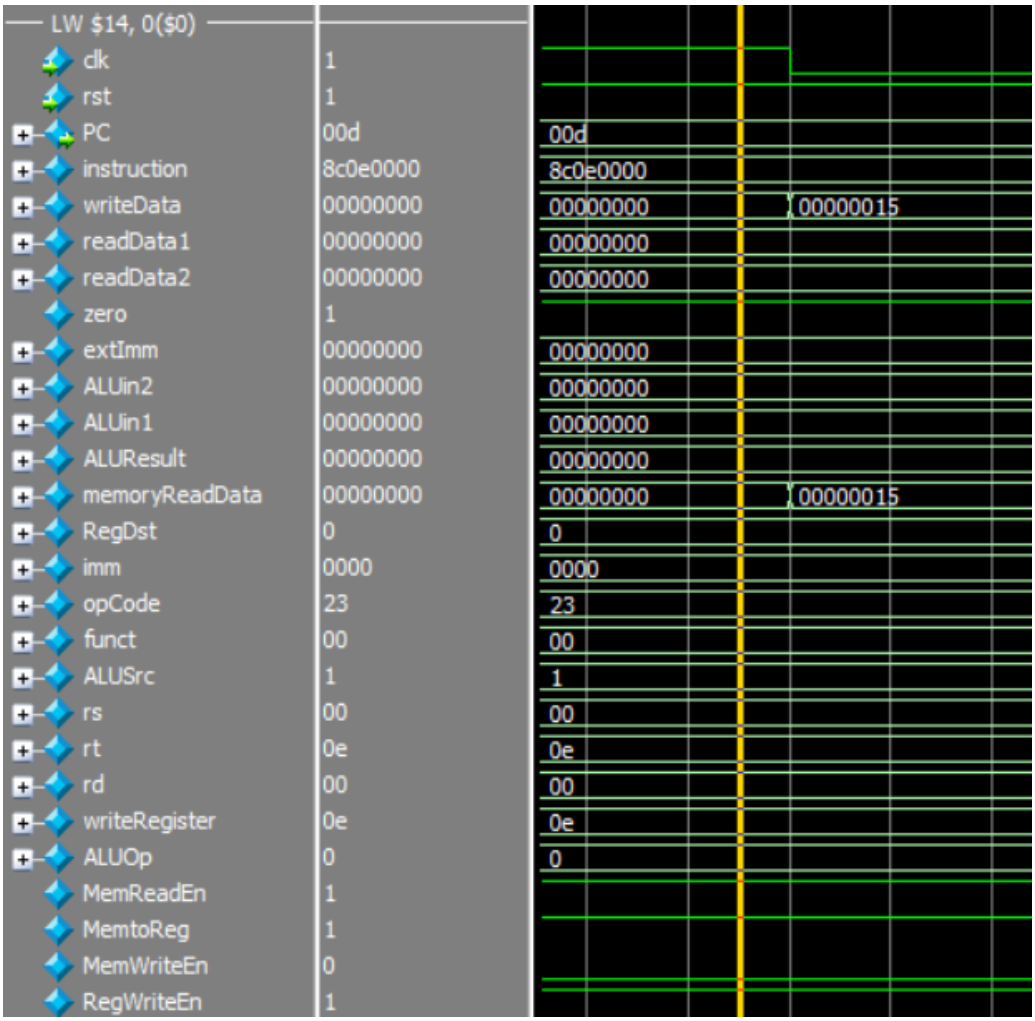
**Object code → AC040000**




When the instruction **SW $4, 0x0($0)** is executed with the initial PC value of 12, the instruction is fetched from memory at address 12, where the opcode **SW** (Store Word) is decoded. During the instruction decode stage, the base address is read from register $0 (which holds the value 0x0), the offset (0x0) is extracted from the instruction, and the value to be stored (0x15) is fetched from register $4. At the positive clock edge, the ALU computes the effective memory address as 0x0 + 0x0 = 0x0, with the **ALUSrc** signal set to direct the offset as the second operand for the addition operation specified by **ALUOp**. During the memory access stage, the **MemWrite** signal is enabled to allow the data from register $4 (0x15) to be stored at memory address 0x0, while the **MemRead** signal is disabled to prevent data being read from memory. Since this is a store operation, the **RegWrite** signal is disabled as no register is written back. The PC is incremented by 1 to point to the next instruction, updating the PC to 13

**Instruction 14 → LW $14, 0($0)**

**Object code → 8C0E0000**

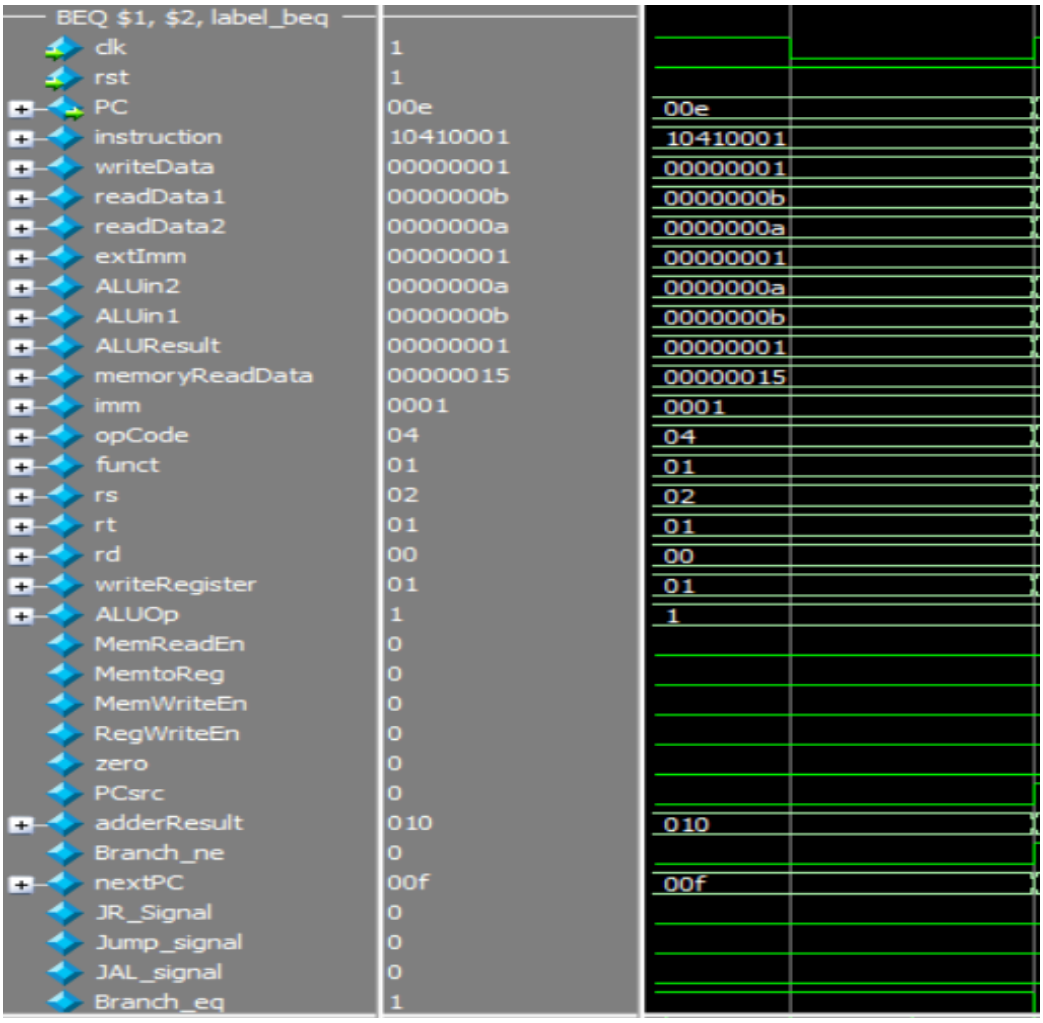



**Register value after execution**


When the instruction **LW $14, 0x0($0)** is executed with the initial PC value of 13, the instruction is fetched from memory at address 13, where the opcode **LW** (Load Word) is decoded. During the decode stage, the base address is retrieved from register $0 (which holds the value 0x0), and the offset (0x0) is extracted from the instruction. At the positive clock edge, the ALU computes the effective memory address as 0x0 + 0x0 = 0x0, with the **ALUSrc** control signal directing the ALU to use the offset as the second operand. In the memory access stage, the **MemRead** control signal is enabled to allow data to be read from the computed memory address (0x0), while the **MemWrite** signal remains disabled to prevent writing to memory. The memory at address 0x0 contains the value 0x15, which is read and passed to the write-back stage. During the write-back stage, the **RegWrite** signal is enabled to store the value 0x15 into register $14. This ensures the correct data is updated in the register file. Meanwhile, the **RegDst** signal selects the target register as specified in the instruction's rt field. The PC is incremented by 1 to point to the next instruction, updating its value to 14

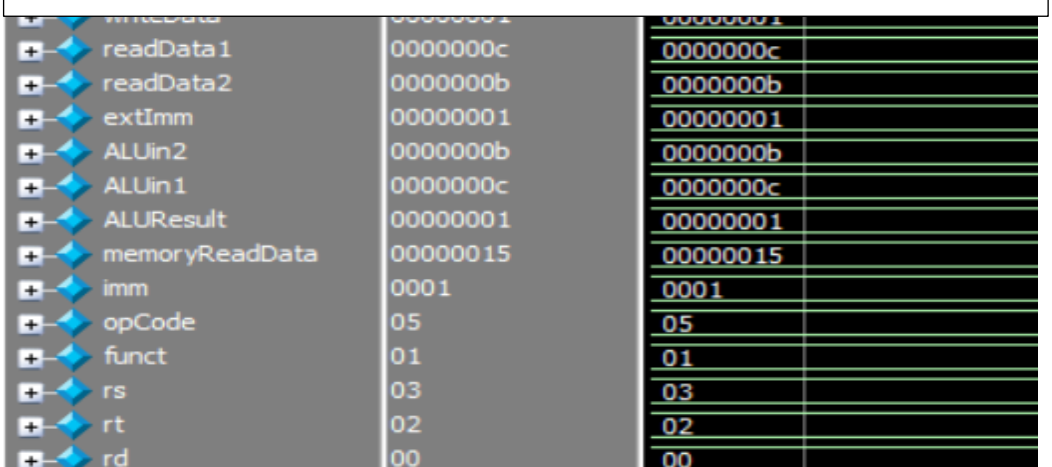
**Instruction 15 → BEQ $1, $2, label_beq**

**Object code → 10410001**



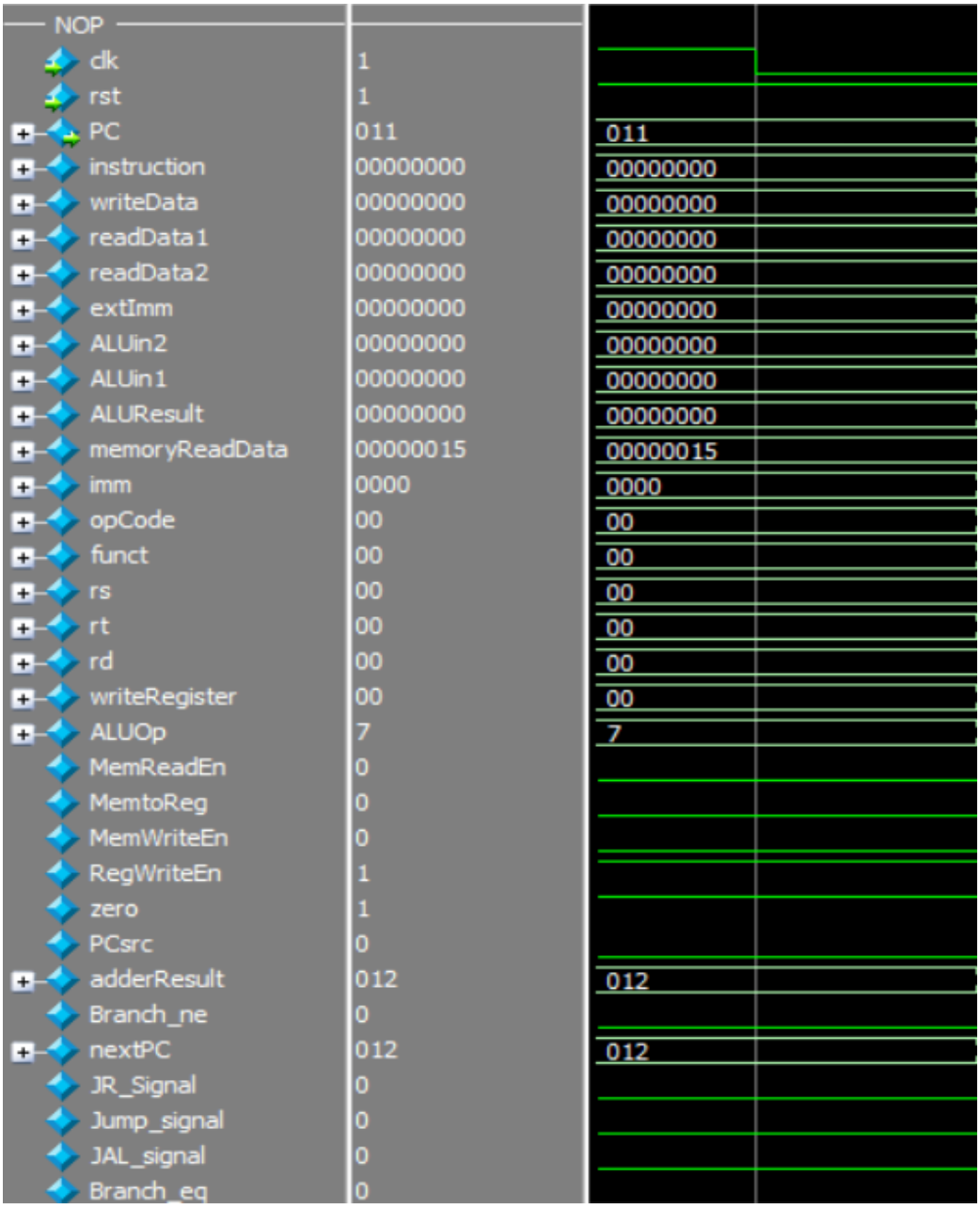

## Instruction 16 → BNE $2, $3, label_bne

When the instruction **BEQ $1, $2, label_beq** is executed in a single-cycle processor with the initial PC value of 14, the first step is the instruction fetch. The instruction is fetched from memory address 14, where the opcode BEQ is decoded. During the decode stage, the processor reads the values of registers $1 (which holds 0xA) and $2 (which holds 0xB). The processor then compares the two values, and since 0xA is not equal to 0xB, the **Zero flag** is set to **0** (indicating the values are not equal). At the positive clock edge, the branch is not taken because the **Zero flag** is not active. As a result, the **Branch** control signal is set to **0**. The PC is incremented by 1 to point to the next instruction, so the new PC value becomes **5**. Control signals such as **RegDst** and **ALUSrc** are set to their appropriate values, but since no register writes or memory access occurs, **RegWrite**, **Memory Read**, and **Memory Write** signals remain disabled. The entire process is completed in a single cycle, with the PC value updated as specified.



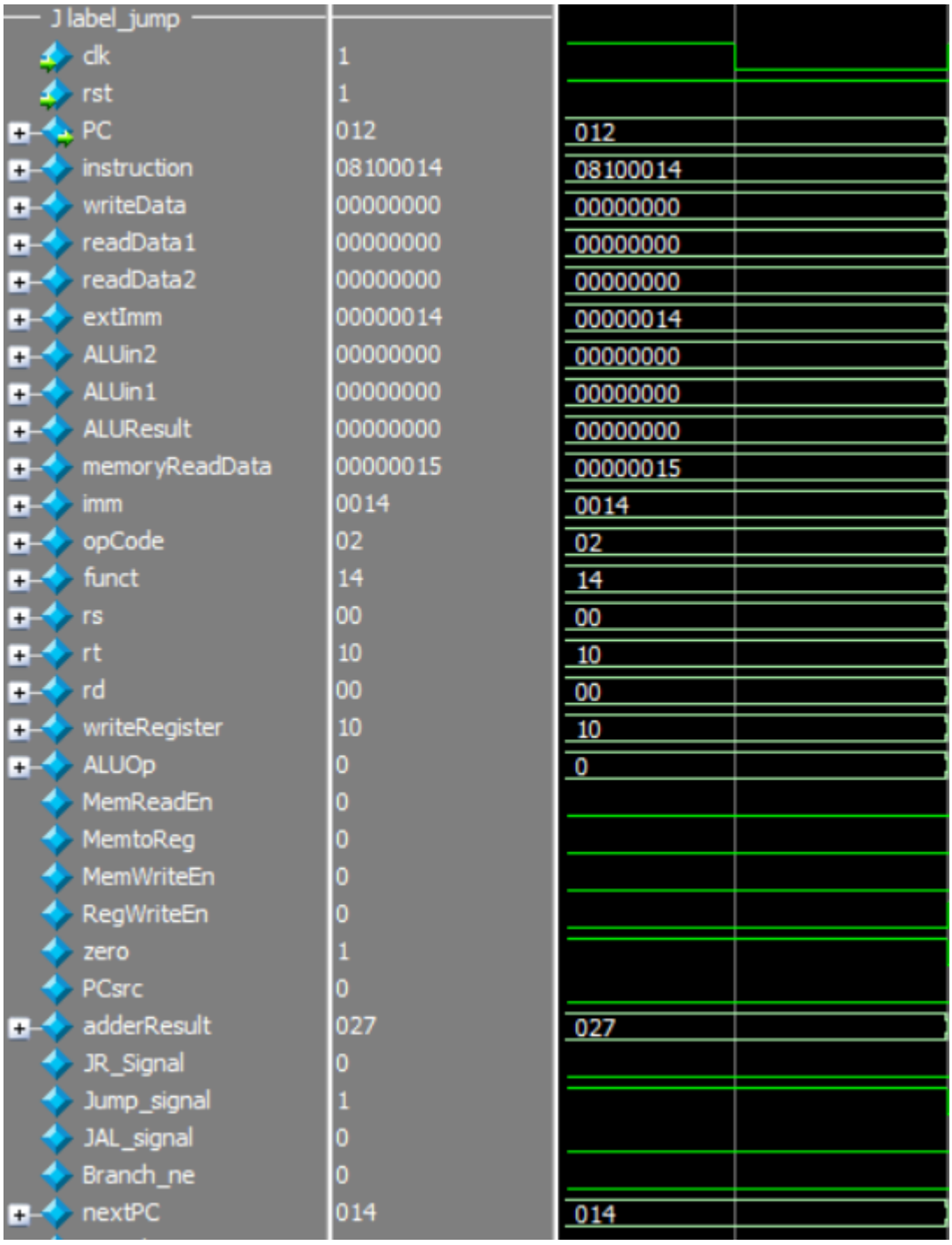
**NO Instruction → NOP**

**At label_bne, a NOP (no operation) is executed as a placeholder.**
**This instruction does nothing and is used here to indicate**
**no action is required at this label , PC is incremented by 1**



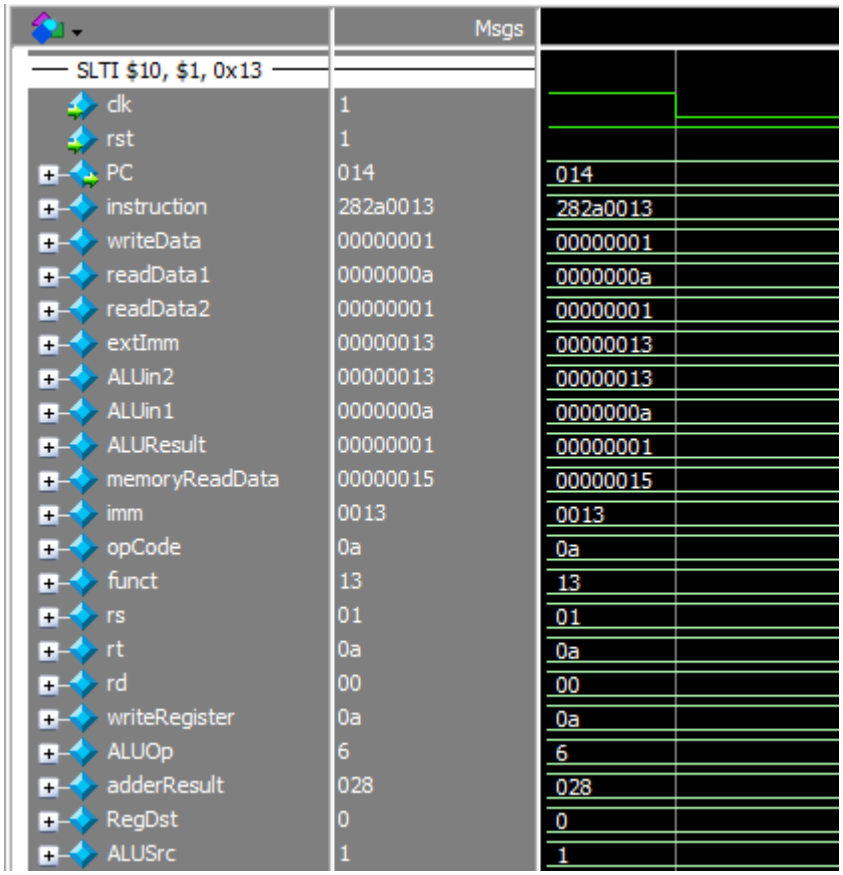

**Instruction 17 → J label_jump**

**Object code → 08100014**

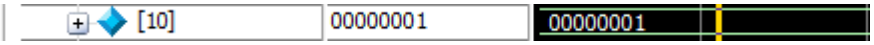



**Instruction 18 → SLTI $10, $1, 0x13**

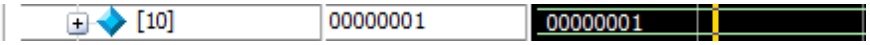
**Machine Code → 282A0013**




**Register value before execution**

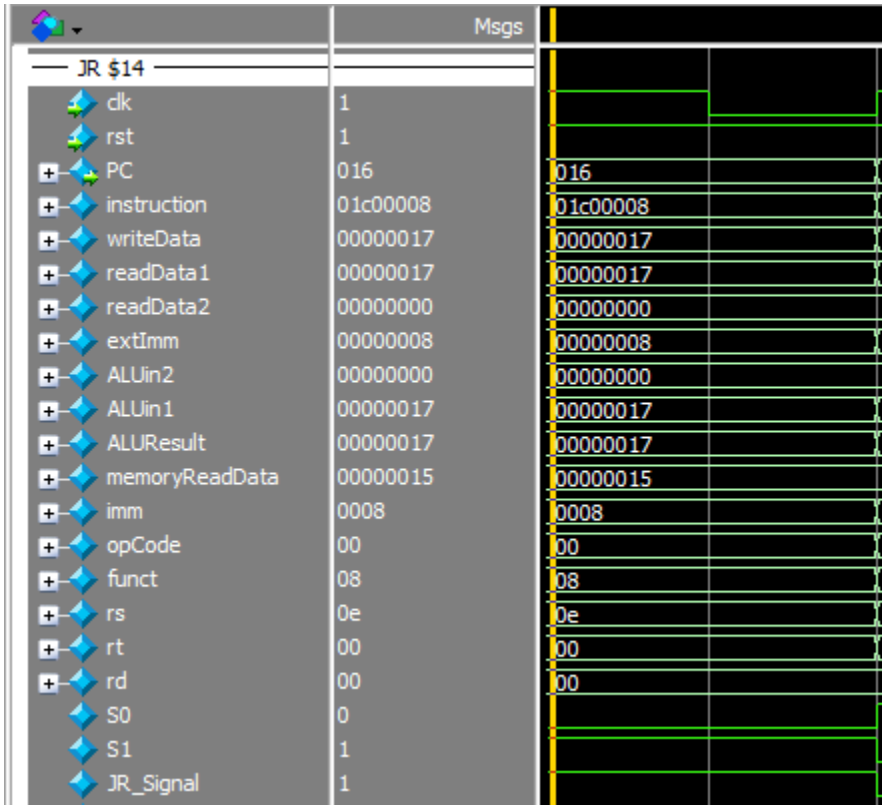



**Register value after execution**



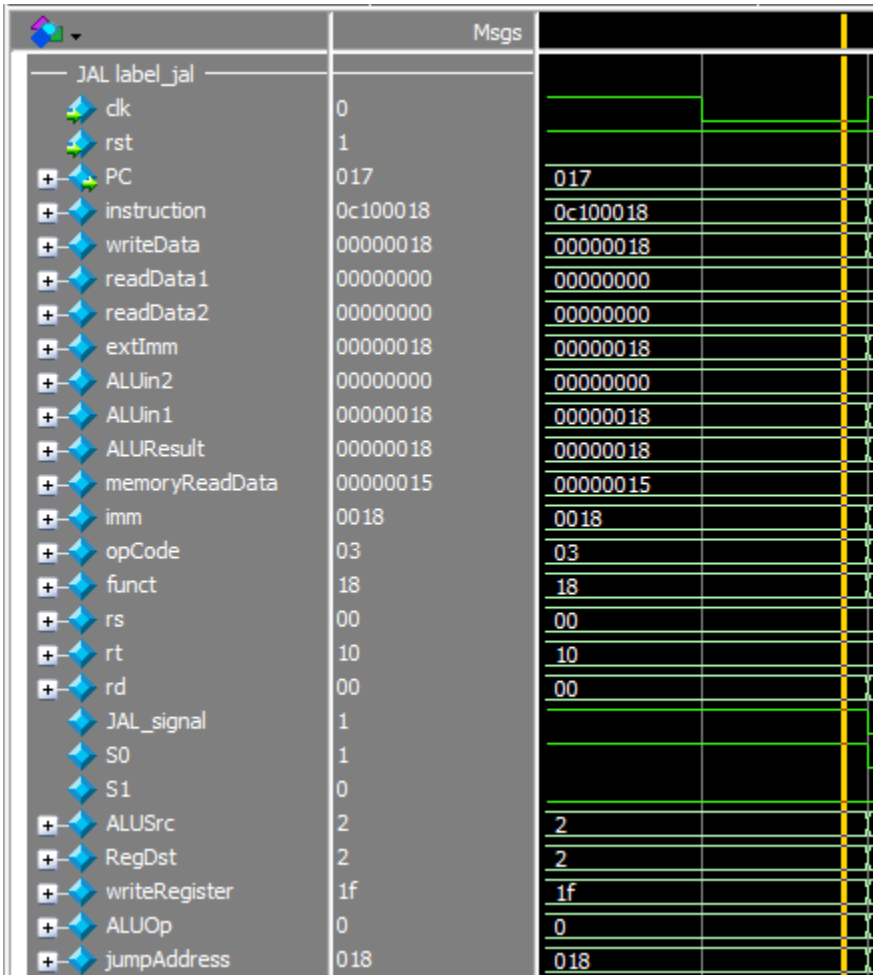

**Instruction 18 → JR $14**

**Machine Code → 01C00008**

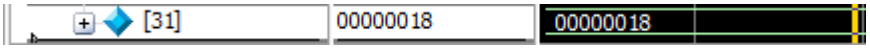



**Instruction 18 → JAL label_jal**

**Machine Code → 0C100018**




**Register value after execution**

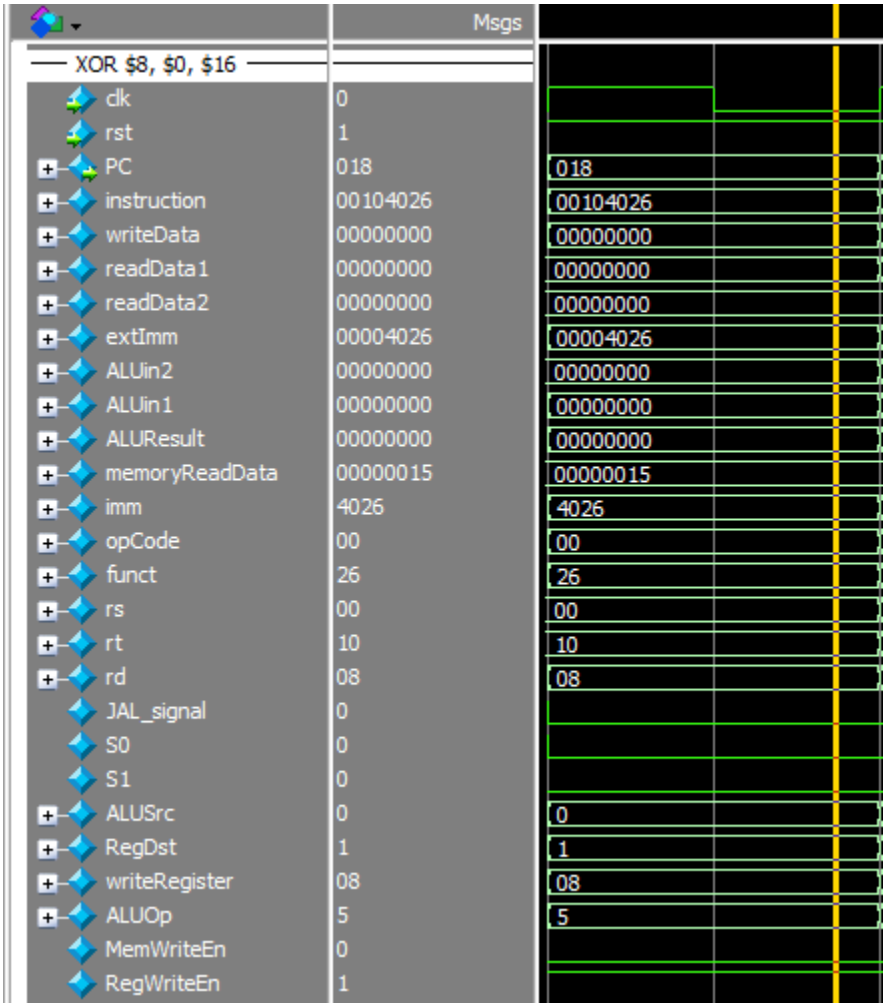



JAL Instructions edit value if PC and return next address to save in register 31

When the JAL (Jump and Link) instruction is executed, it performs two key operations: updating the value of the program counter (PC) to the target address specified by the instruction and saving the address of the next instruction in register $31 (the link register).

**Instruction 18 → XOR $8, $0, $16**

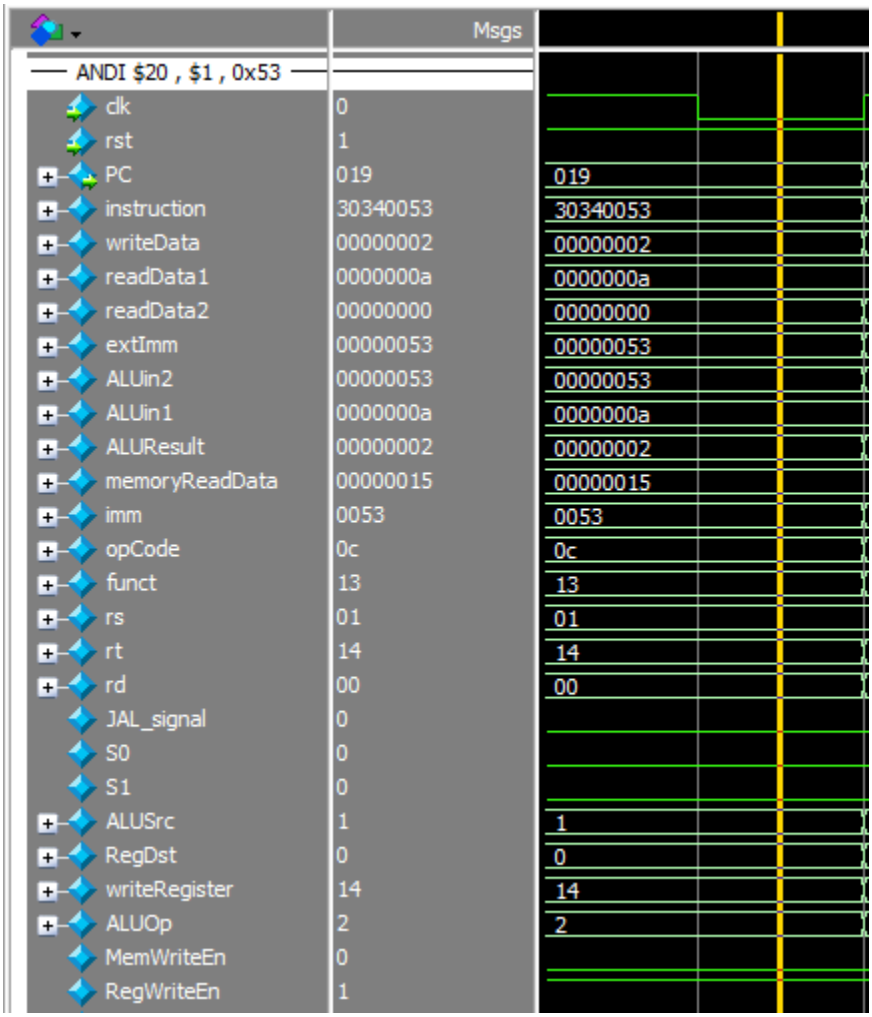
**Machine Code → 00104026**




**Register value after execution**


**Instruction 18 → ANDI $20 , $1 , 0x53**

**Machine Code → 30340053**

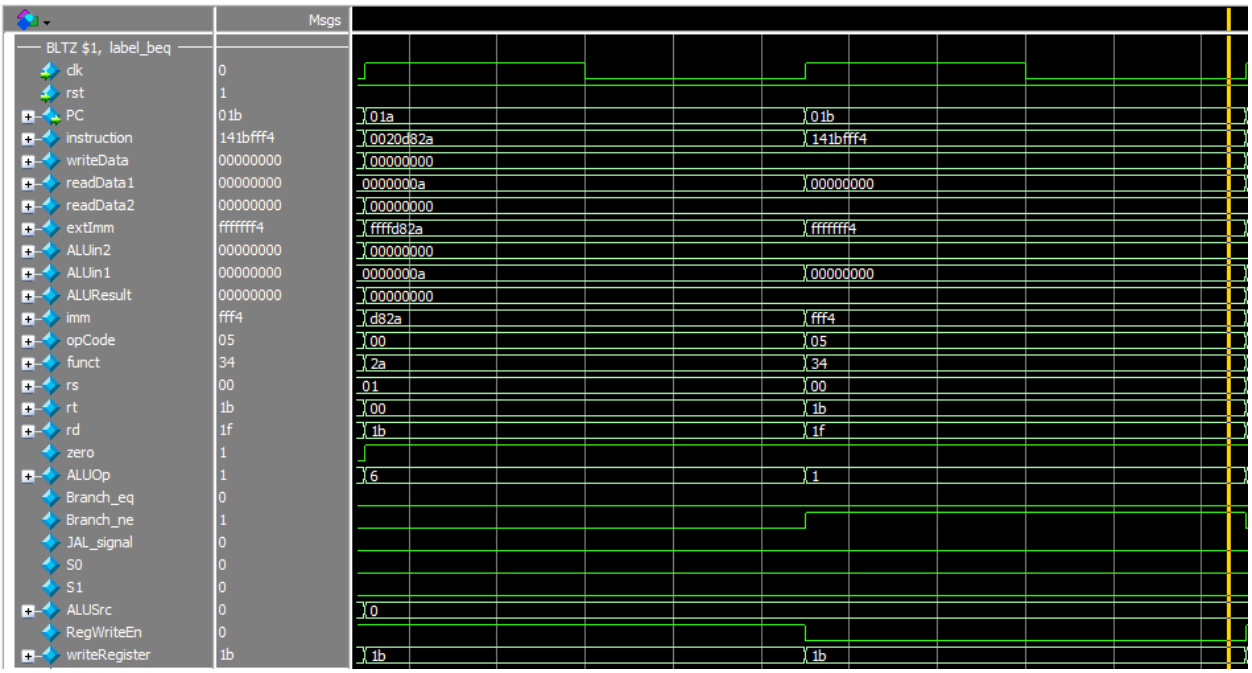



**Register value after execution**


**Instruction 18 → BLTZ $1, label_beq**

**Convert To :**

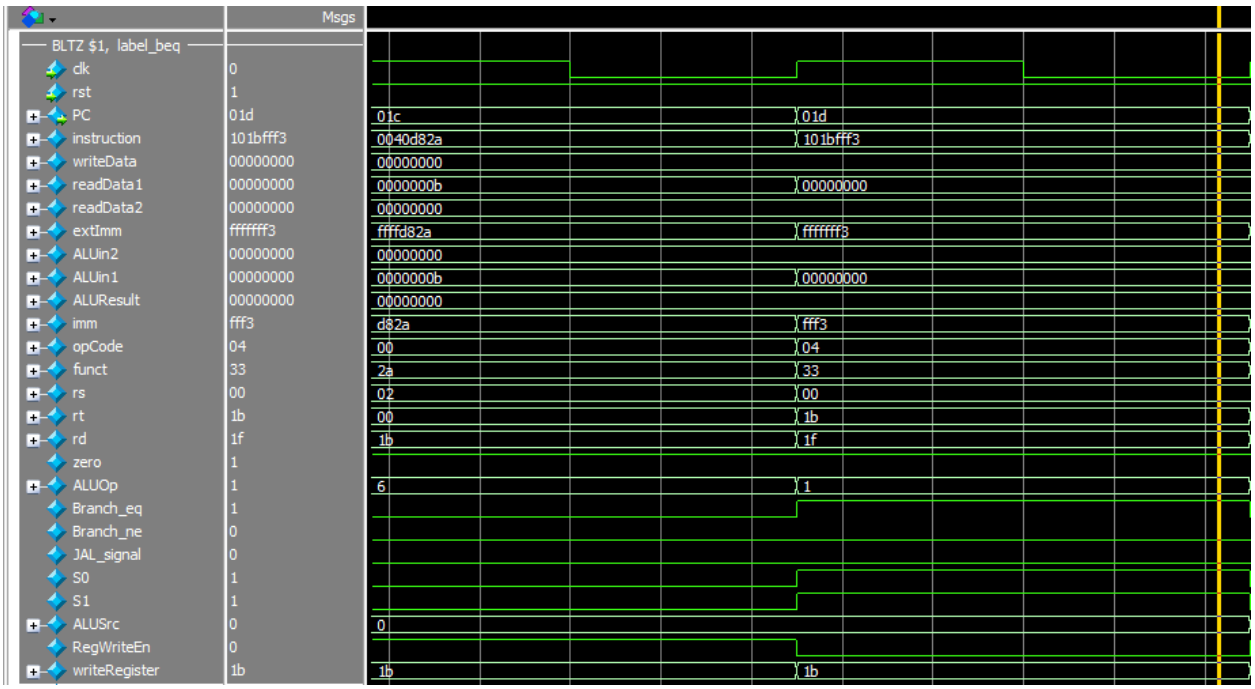
**slt $27, $1, $zero , bne $27, $zero, label_beq**




**Instruction 18 → BGEZ $2, label_beq**

**Convert To :**

**slt $27, $2, $zero , beq $27, $zero, label_bne**



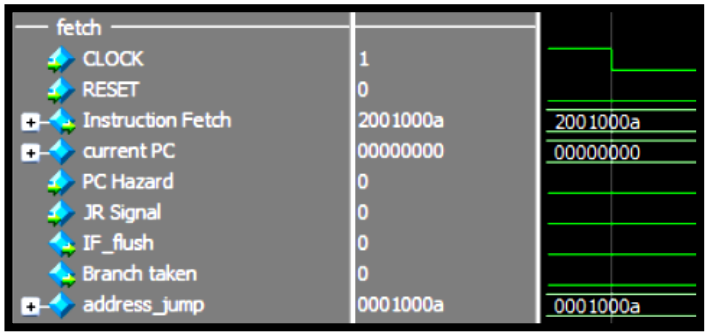

## Pipeline Testing

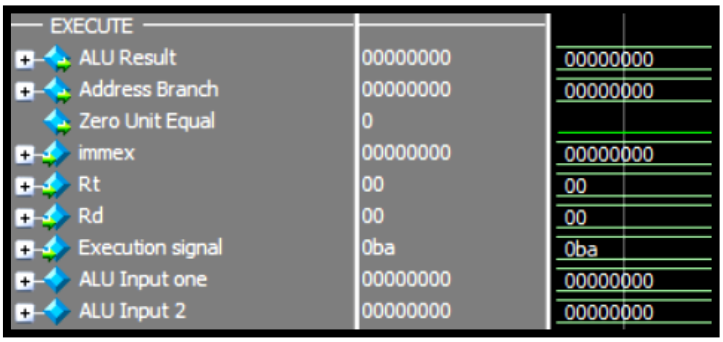
## Benchmark 1 – Data Manipulation Instruction and Control Flow Instructions

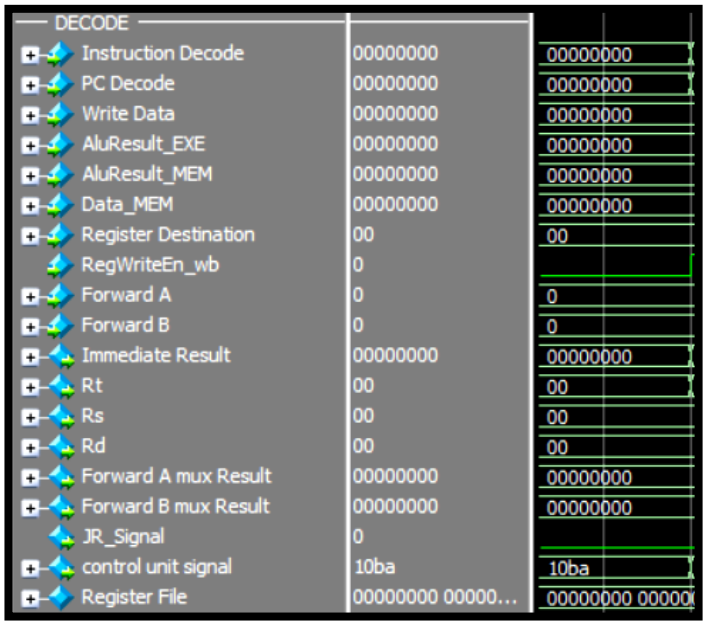
```
        .data
value:  .word 0x5          # sample data for loading
        .text
main:

# Immediate instructions to initialize registers
ADDI $1, $0, 0xA
ORI $2, $0, 0xB
XORI $3, $0, 0xC

# Basic ALU operations
ADD $4, $1, $2
SUB $5, $4, $3
AND $6, $1, $3
OR $7, $2, $3
NOR $8, $2, $3

# Comparison operations
SLT $10, $1, $2
SGT $11, $3, $1

# Shift operations
SLL $12, $1, 2
SRL $13, $2, 1

# Load and store word instructions
SW $4, 0x0($0)
LW $14, 0x0($0)

# Branch instructions

BEQ $1, $2, label_beq
BNE $2, $3, label_bne
```

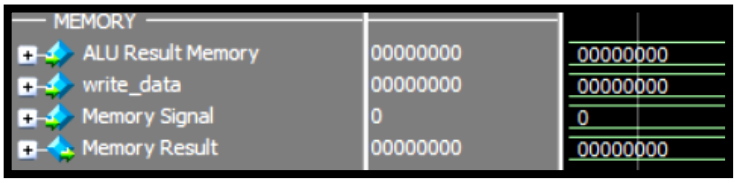
```
label_beq:
ANDI $20 , $1 , 0x53
label_bne:
Nop

J label_jump          # Jump to label_jump
SLL $7,$1,3 # This will not execute due to jump
label_jump:
SLTI $10, $1, 0x13

ADDI $14,$14,0x1



# Jump Register (JR) instruction
JR $14                #Jump to the address in $14

# Jump and Link (JAL) instruction
JAL label_jal         # Jump to label_jal and store return address in $31

label_jal:
XOR $8, $0, $16

ANDI $20 , $1 , 0x53

#psudo instructions
BLTZ $1,  label_beq
BGEZ $2, label_bne
```

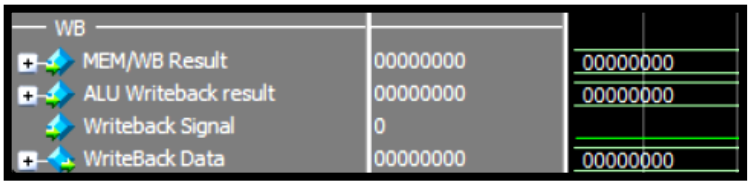
## CYCLE 1

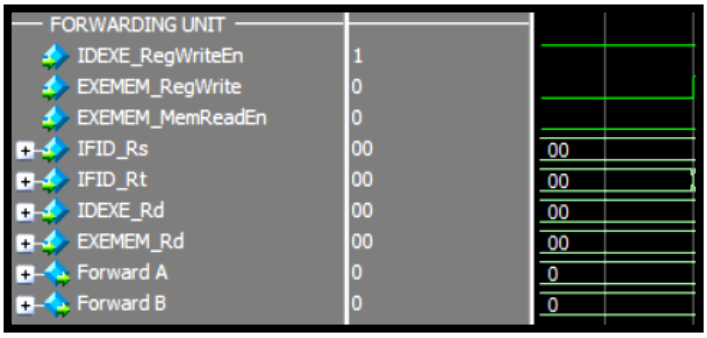

fetch
CLOCK 1
RESET 0
Instruction Fetch 2001000a
current PC 00000000
PC Hazard 0
JR Signal 0
IF_flush 0
Branch taken 0
address_jump 0001000a


EXECUTE
ALU Result 00000000
Address Branch 00000000
Zero Unit Equal 0
immex 00000000
Rt 00
Rd 00
Execution signal 0ba
ALU Input one 00000000
ALU Input 2 00000000

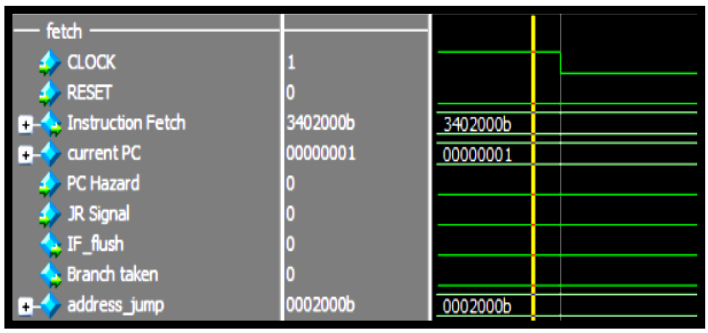

DECODE
Instruction Decode 00000000
PC Decode 00000000
Write Data 00000000
AluResult_EXE 00000000
AluResult_MEM 00000000
Data_MEM 00000000
Register Destination 00
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00000000
Rt 00
Rs 00
Rd 00
Forward A mux Result 00000000
Forward B mux Result 00000000
JR_Signal 0
control unit signal 10ba
Register File 00000000 00000...

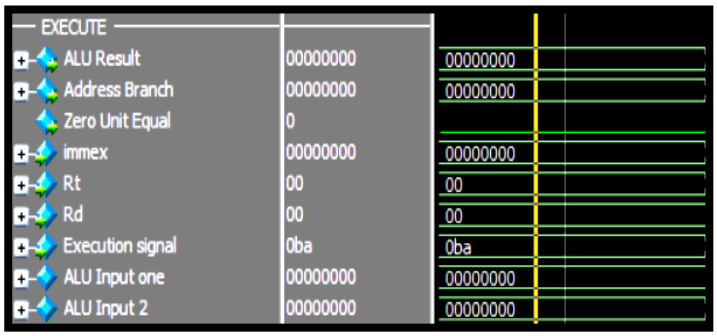

MEMORY
ALU Result Memory 00000000
write_data 00000000
Memory Signal 0
Memory Result 00000000

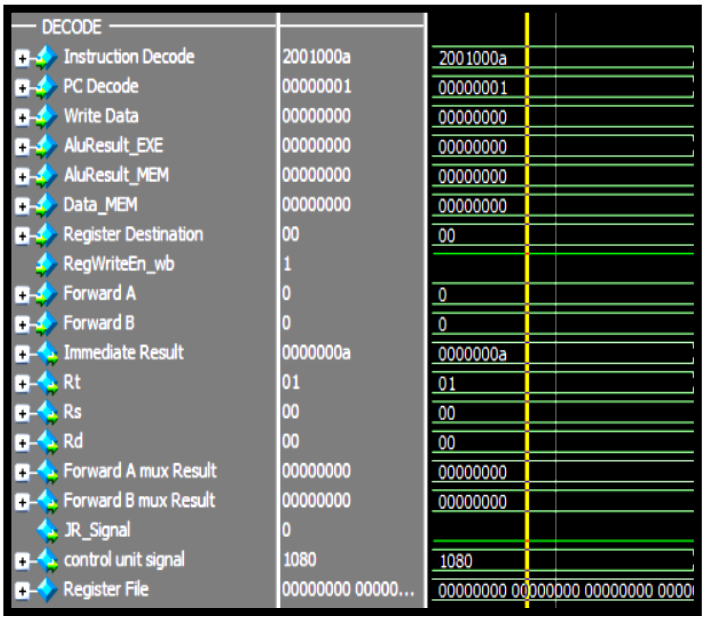

WB
MEM/WB Result 00000000
ALU Writeback result 00000000
Writeback Signal 0
WriteBack Data 00000000

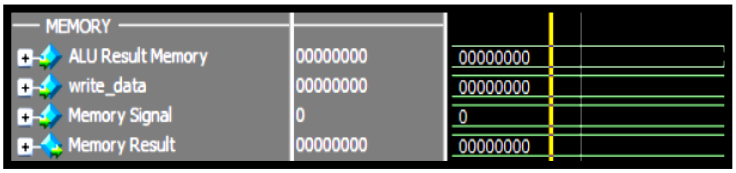

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 00
IFID_Rt 00
IDEXE_Rd 00
EXEMEM_Rd 00
Forward A 0
Forward B 0

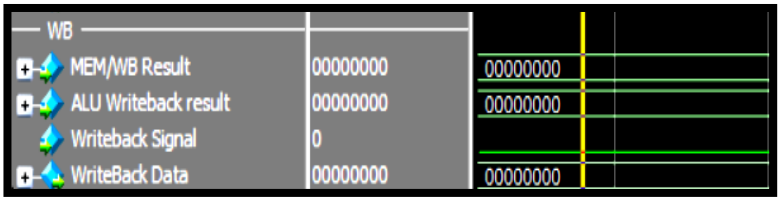
## CYCLE 2

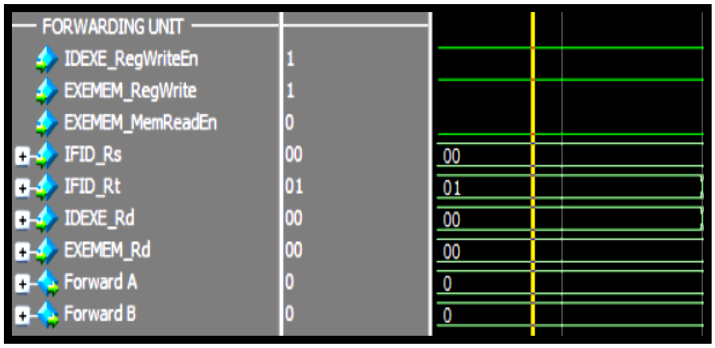

fetch
CLOCK 1
RESET 0
Instruction Fetch 3402000b 3402000b
current PC 00000001 00000001
PC Hazard 0
JR Signal 0
IF_flush 0
Branch taken 0
address_jump 0002000b 0002000b


EXECUTE
ALU Result 00000000 00000000
Address Branch 00000000 00000000
Zero Unit Equal 0
immex 00000000 00000000
Rt 00 00
Rd 00 00
Execution signal 0ba 0ba
ALU Input one 00000000 00000000
ALU Input 2 00000000 00000000

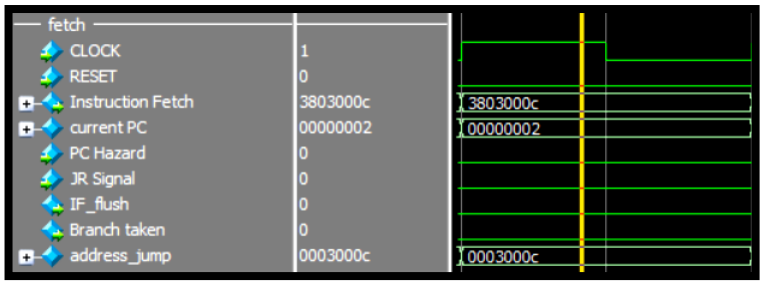

DECODE
Instruction Decode 2001000a 2001000a
PC Decode 00000001 00000001
Write Data 00000000 00000000
AluResult_EXE 00000000 00000000
AluResult_MEM 00000000 00000000
Data_MEM 00000000 00000000
Register Destination 00 00
RegWriteEn_wb 1
Forward A 0 0
Forward B 0 0
Immediate Result 0000000a 0000000a
Rt 01 01
Rs 00 00
Rd 00 00
Forward A mux Result 00000000 00000000
Forward B mux Result 00000000 00000000
JR_Signal 0
control unit signal 1080 1080
Register File 00000000 00000... 00000000 00000000 00000000 0000

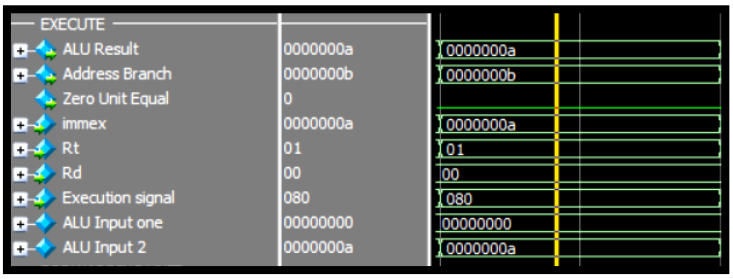

MEMORY
ALU Result Memory 00000000 00000000
write_data 00000000 00000000
Memory Signal 0 0
Memory Result 00000000 00000000

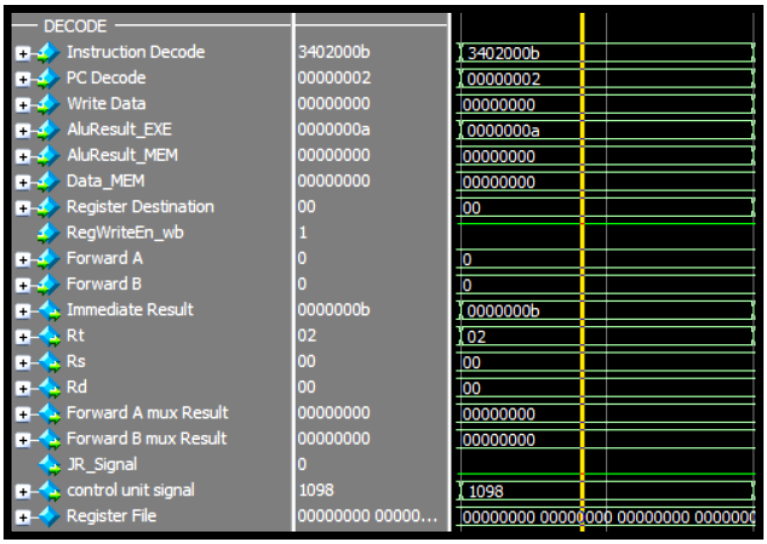

WB
MEM/WB Result 00000000 00000000
ALU Writeback result 00000000 00000000
Writeback Signal 0
WriteBack Data 00000000 00000000

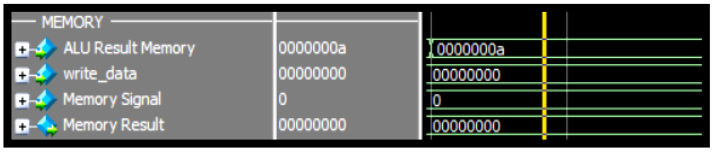

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 00 00
IFID_Rt 01 01
IDEXE_Rd 00 00
EXEMEM_Rd 00 00
Forward A 0 0
Forward B 0 0

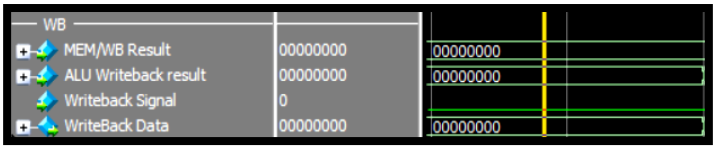
# CYCLE 3

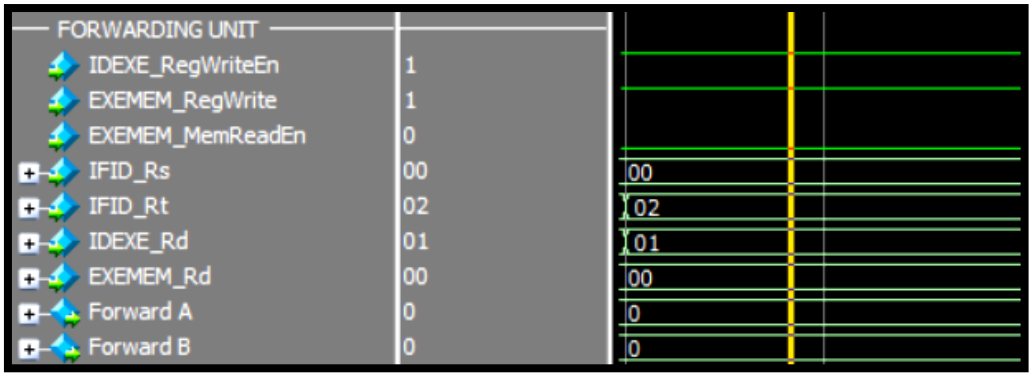


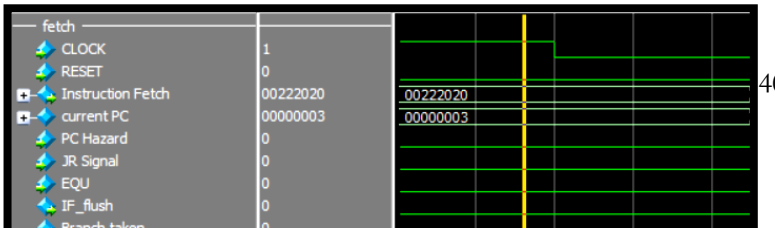



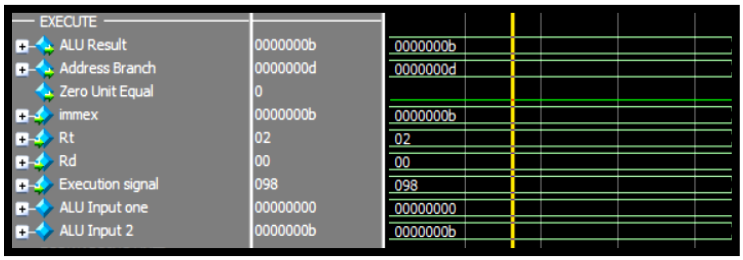



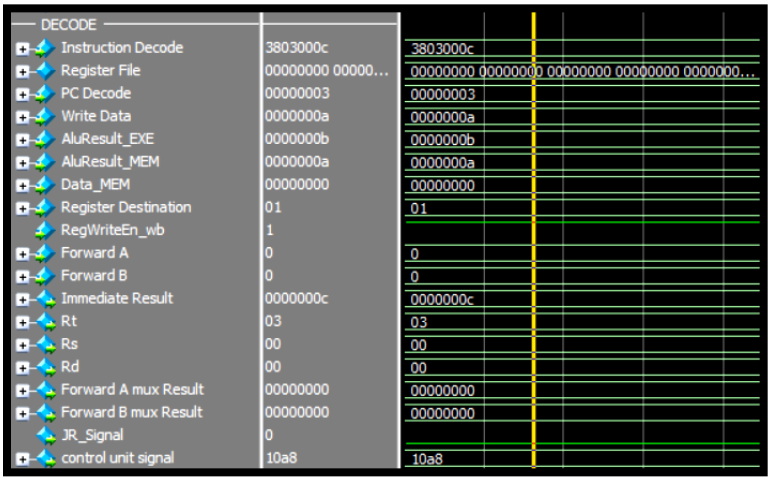



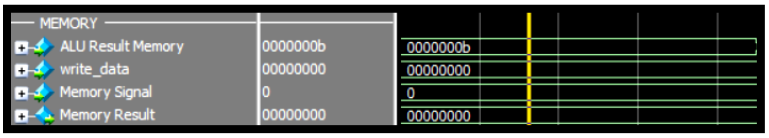



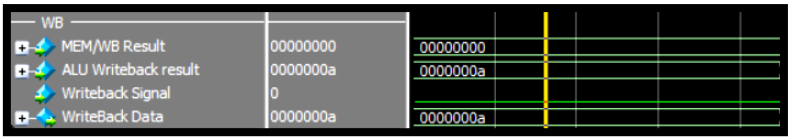

# CYCLE 4

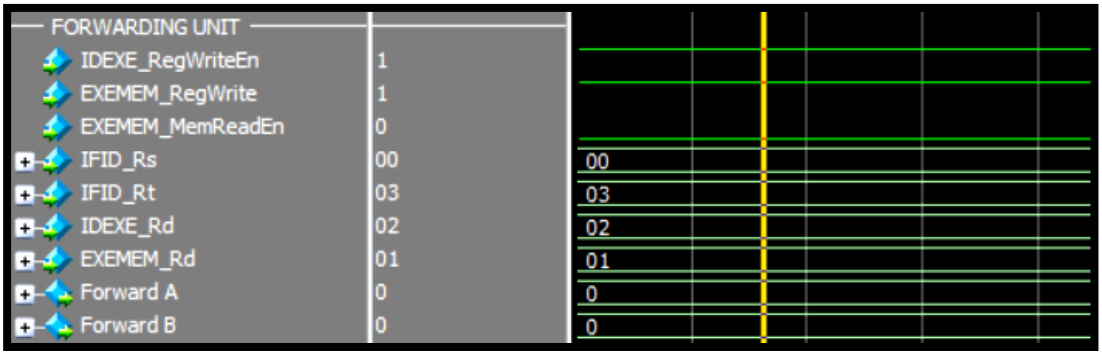



EXECUTE
ALU Result 0000000b
Address Branch 0000000d
Zero Unit Equal 0
immex 0000000b
Rt 02
Rd 00
Execution signal 098
ALU Input one 00000000
ALU Input 2 0000000b

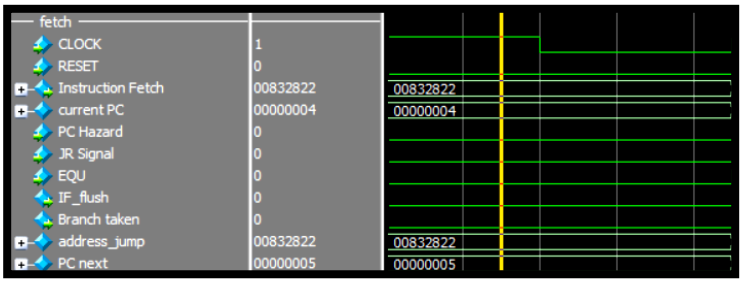

DECODE
Instruction Decode 3803000c
Register File 00000000 00000...
PC Decode 00000003
Write Data 0000000a
AluResult_EXE 0000000b
AluResult_MEM 0000000a
Data_MEM 00000000
Register Destination 01
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 0000000c
Rt 03
Rs 00
Rd 00
Forward A mux Result 00000000
Forward B mux Result 00000000
JR_Signal 0
control unit signal 10a8

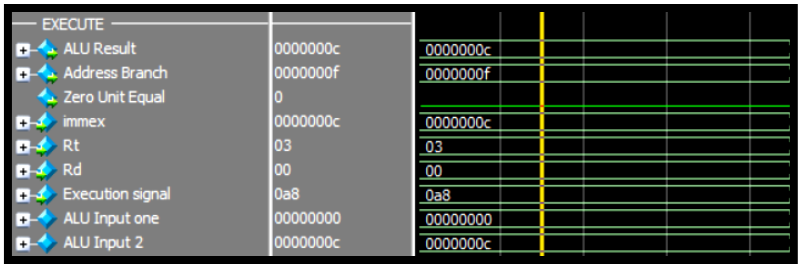

MEMORY
ALU Result Memory 0000000b
write_data 00000000
Memory Signal 0
Memory Result 00000000

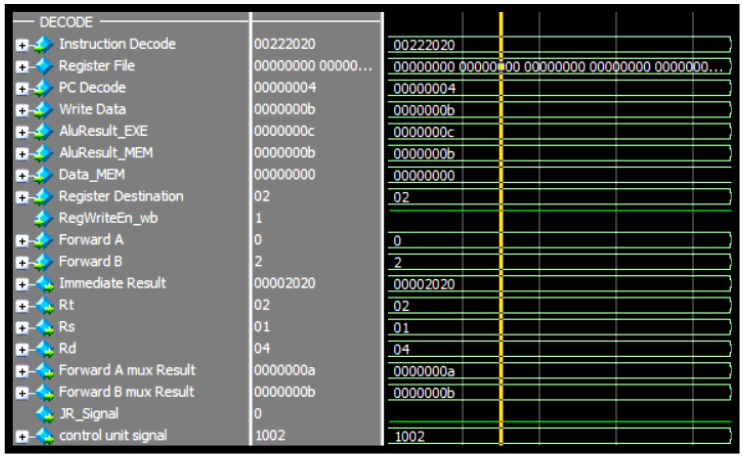

WB
MEM/WB Result 00000000
ALU Writeback result 0000000a
Writeback Signal 0
WriteBack Data 0000000a

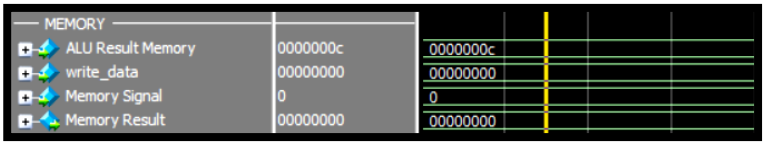

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 00
IFID_Rt 03
IDEXE_Rd 02
EXEMEM_Rd 01
Forward A 0
Forward B 0

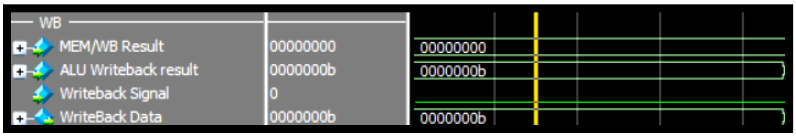
## Cycle 5

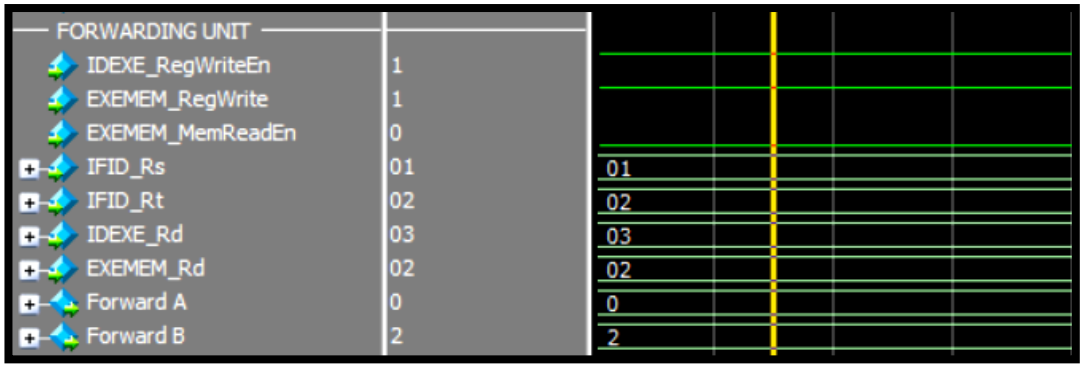

fetch
CLOCK 1
RESET 0
Instruction Fetch 00832822
current PC 00000004
PC Hazard 0
JR Signal 0
EQU 0
IF_flush 0
Branch taken 0
address_jump 00832822
PC next 00000005


EXECUTE
ALU Result 0000000c
Address Branch 0000000f
Zero Unit Equal 0
immex 0000000c
Rt 03
Rd 00
Execution signal 0a8
ALU Input one 00000000
ALU Input 2 0000000c

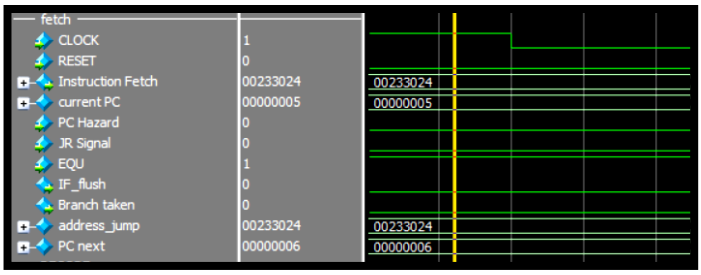

DECODE
Instruction Decode 00222020
Register File 00000000 00000...
PC Decode 00000004
Write Data 0000000b
AluResult_EXE 0000000c
AluResult_MEM 0000000b
Data_MEM 00000000
Register Destination 02
RegWriteEn_wb 1
Forward A 0
Forward B 2
Immediate Result 00002020
Rt 02
Rs 01
Rd 04
Forward A mux Result 0000000a
Forward B mux Result 0000000b
JR_Signal 0
control unit signal 1002

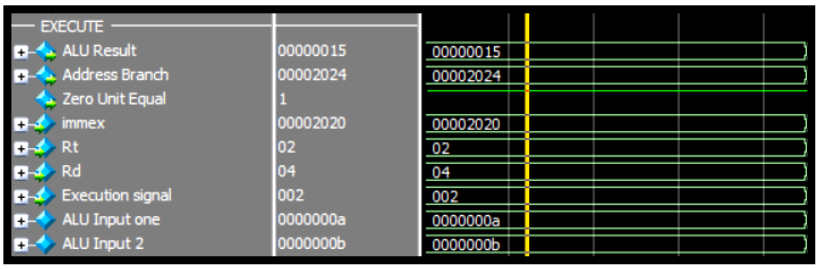

MEMORY
ALU Result Memory 0000000c
write_data 00000000
Memory Signal 0
Memory Result 00000000

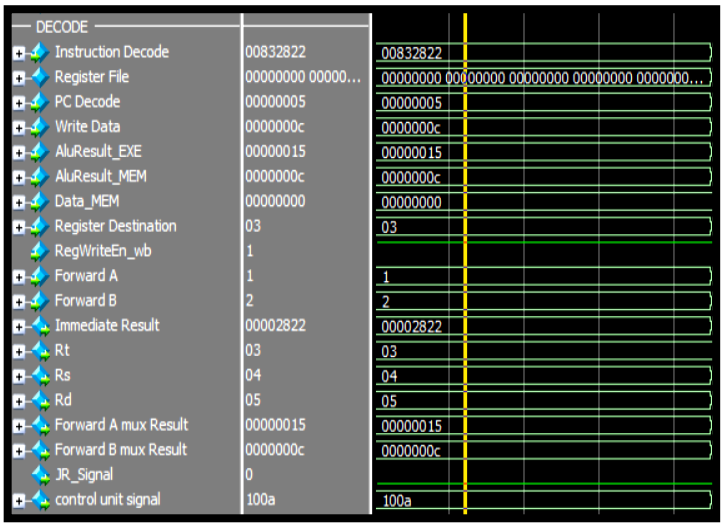

WB
MEM/WB Result 00000000
ALU Writeback result 0000000b
Writeback Signal 0
WriteBack Data 0000000b

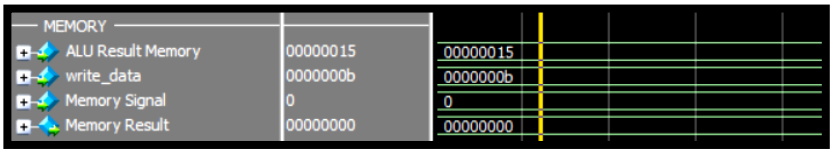

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 01
IFID_Rt 02
IDEXE_Rd 03
EXEMEM_Rd 02
Forward A 0
Forward B 2

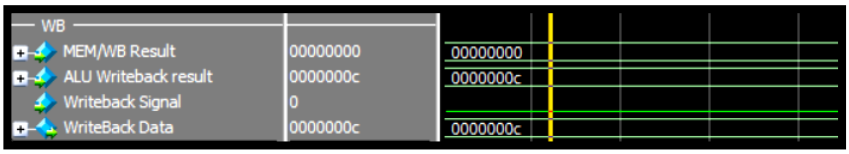
## Cycle 6

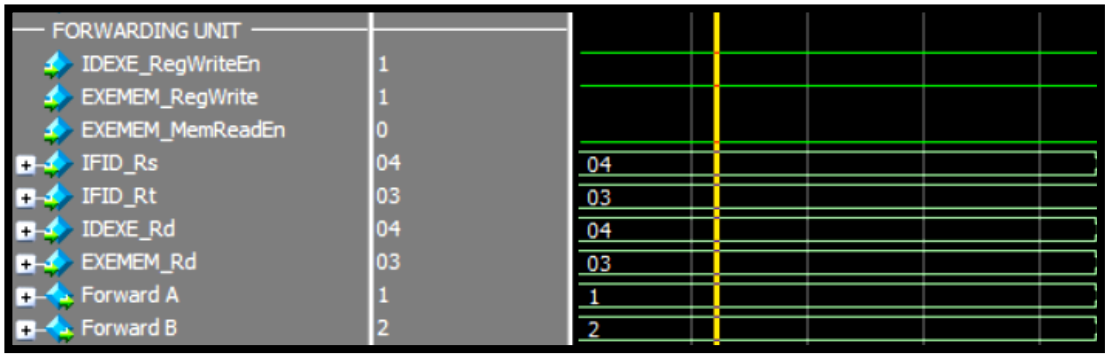

fetch
CLOCK 1
RESET 0
Instruction Fetch 00233024
current PC 00000005
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 00233024
PC next 00000006


EXECUTE
ALU Result 00000015
Address Branch 00002024
Zero Unit Equal 1
immex 00002020
Rt 02
Rd 04
Execution signal 002
ALU Input one 0000000a
ALU Input 2 0000000b

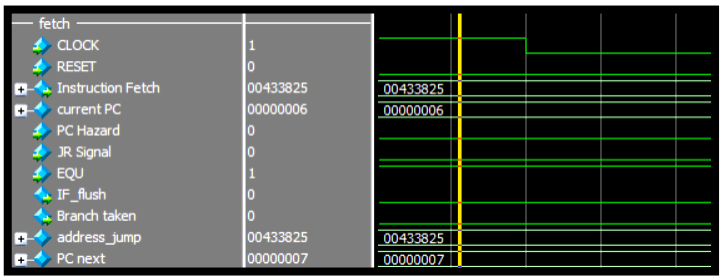

DECODE
Instruction Decode 00832822
Register File 00000000 00000...
PC Decode 00000005
Write Data 0000000c
AluResult_EXE 00000015
AluResult_MEM 0000000c
Data_MEM 00000000
Register Destination 03
RegWriteEn_wb 1
Forward A 1
Forward B 2
Immediate Result 00002822
Rt 03
Rs 04
Rd 05
Forward A mux Result 00000015
Forward B mux Result 0000000c
JR_Signal 0
control unit signal 100a

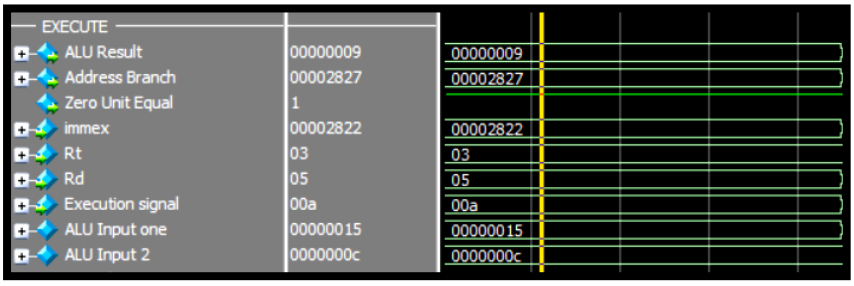

MEMORY
ALU Result Memory 00000015
write_data 0000000b
Memory Signal 0
Memory Result 00000000

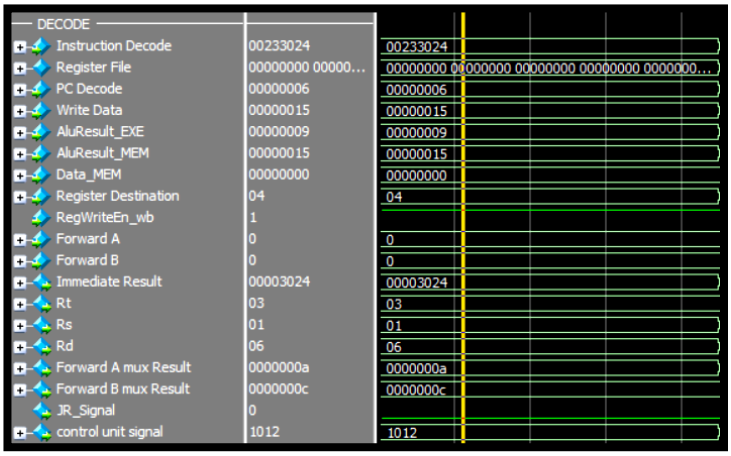

WB
MEM/WB Result 00000000
ALU Writeback result 0000000c
Writeback Signal 0
WriteBack Data 0000000c

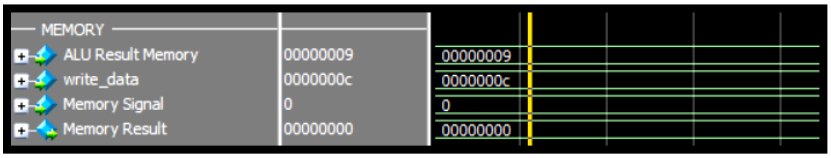

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 04
IFID_Rt 03
IDEXE_Rd 04
EXEMEM_Rd 03
Forward A 1
Forward B 2

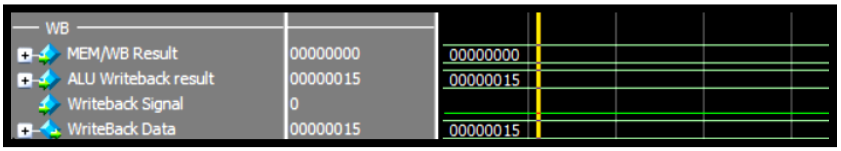
# Cycle 7

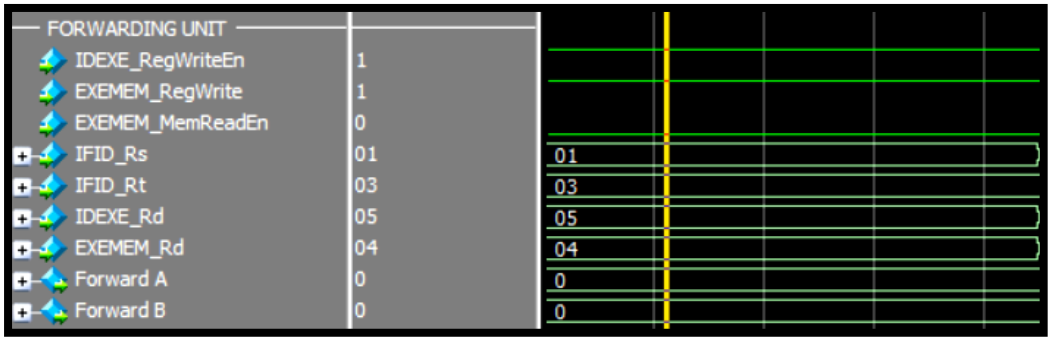

fetch
CLOCK 1
RESET 0
Instruction Fetch 00433825
current PC 00000006
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 00433825
PC next 00000007


EXECUTE
ALU Result 00000009
Address Branch 00002827
Zero Unit Equal 1
immex 00002822
Rt 03
Rd 05
Execution signal 00a
ALU Input one 00000015
ALU Input 2 0000000c

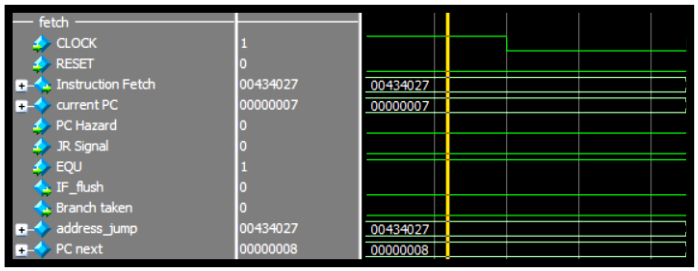

DECODE
Instruction Decode 00233024
Register File 00000000 00000...
PC Decode 00000006
Write Data 00000015
AluResult_EXE 00000009
AluResult_MEM 00000015
Data_MEM 00000000
Register Destination 04
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 00003024
Rt 03
Rs 01
Rd 06
Forward A mux Result 0000000a
Forward B mux Result 0000000c
JR_Signal 0
control unit signal 1012

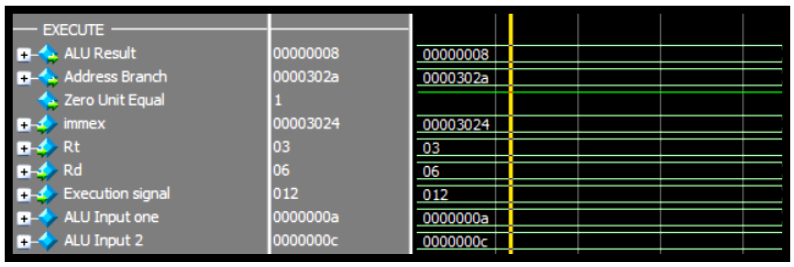

MEMORY
ALU Result Memory 00000009
write_data 0000000c
Memory Signal 0
Memory Result 00000000

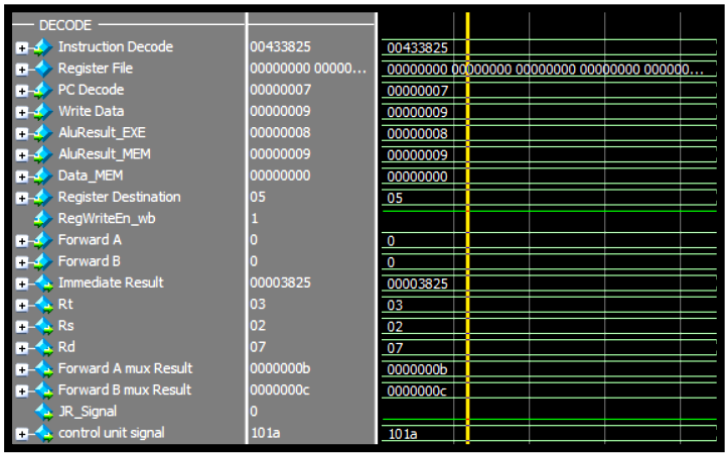

WB
MEM/WB Result 00000000
ALU Writeback result 00000015
Writeback Signal 0
WriteBack Data 00000015

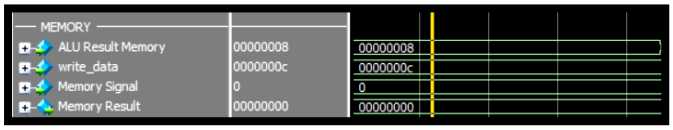

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 01
IFID_Rt 03
IDEXE_Rd 05
EXEMEM_Rd 04
Forward A 0
Forward B 0

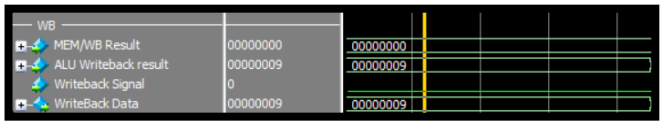
# Cycle 8

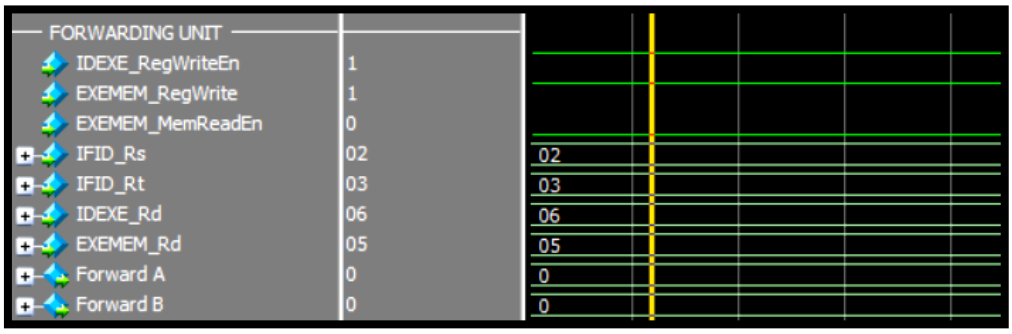

fetch
CLOCK 1
RESET 0
Instruction Fetch 00434027
current PC 00000007
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 00434027
PC next 00000008


EXECUTE
ALU Result 00000008
Address Branch 0000302a
Zero Unit Equal 1
immex 00003024
Rt 03
Rd 06
Execution signal 012
ALU Input one 0000000a
ALU Input 2 0000000c

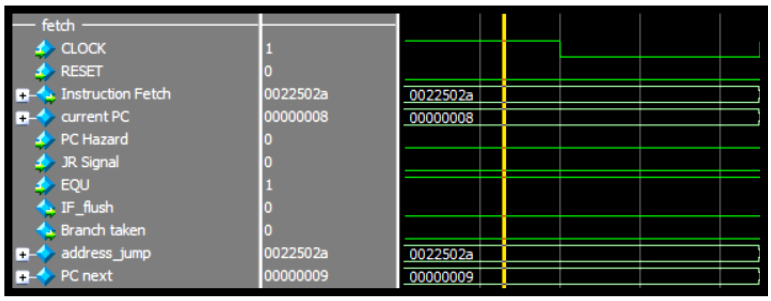

DECODE
Instruction Decode 00433825
Register File 00000000 00000...
PC Decode 00000007
Write Data 00000009
AluResult_EXE 00000008
AluResult_MEM 00000009
Data_MEM 00000000
Register Destination 05
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 00003825
Rt 03
Rs 02
Rd 07
Forward A mux Result 0000000b
Forward B mux Result 0000000c
JR_Signal 0
control unit signal 101a

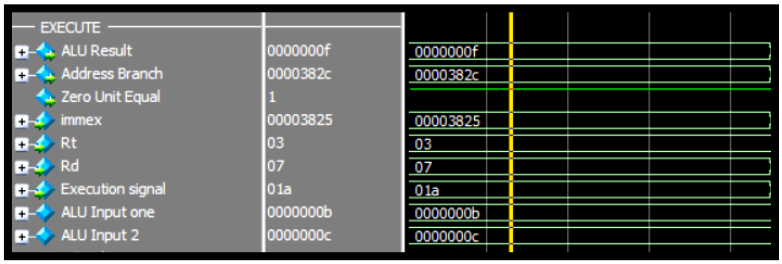

MEMORY
ALU Result Memory 00000008
write_data 0000000c
Memory Signal 0
Memory Result 00000000

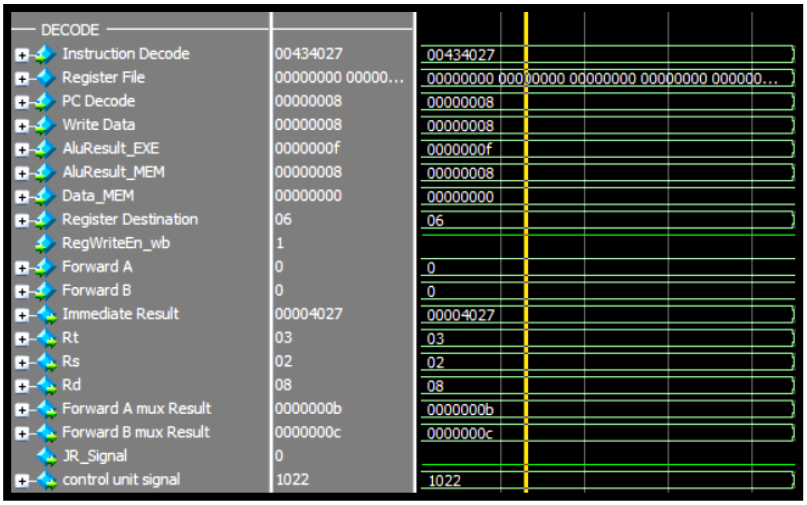

WB
MEM/WB Result 00000000
ALU Writeback result 00000009
Writeback Signal 0
WriteBack Data 00000009

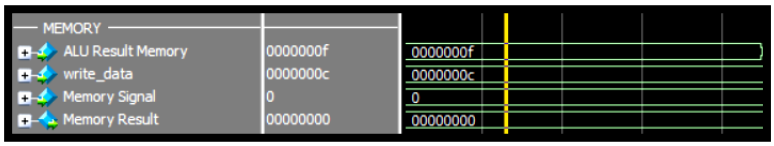

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 02
IFID_Rt 03
IDEXE_Rd 06
EXEMEM_Rd 05
Forward A 0
Forward B 0

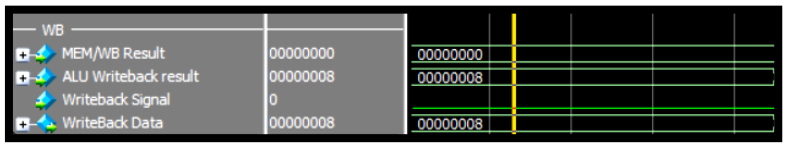
## Cycle 9

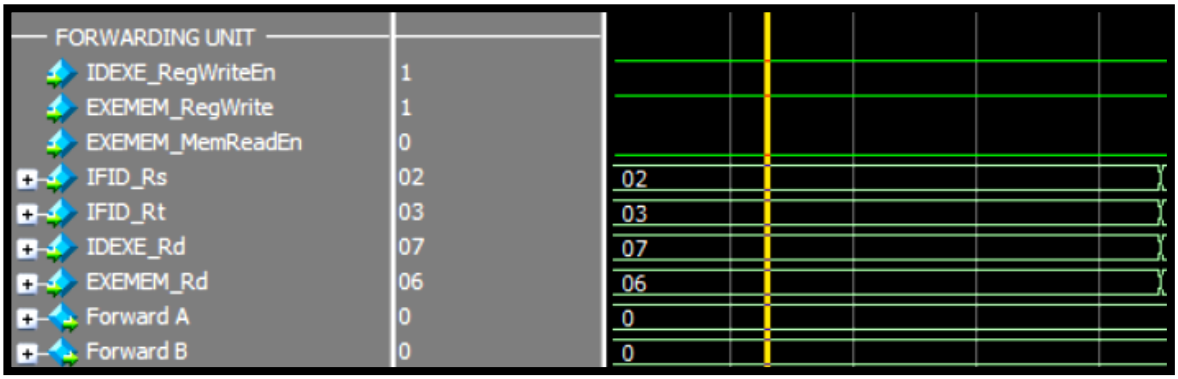

fetch
CLOCK 1
RESET 0
Instruction Fetch 0022502a
current PC 00000008
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 0022502a
PC next 00000009


EXECUTE
ALU Result 0000000f
Address Branch 0000382c
Zero Unit Equal 1
immex 00003825
Rt 03
Rd 07
Execution signal 01a
ALU Input one 0000000b
ALU Input 2 0000000c

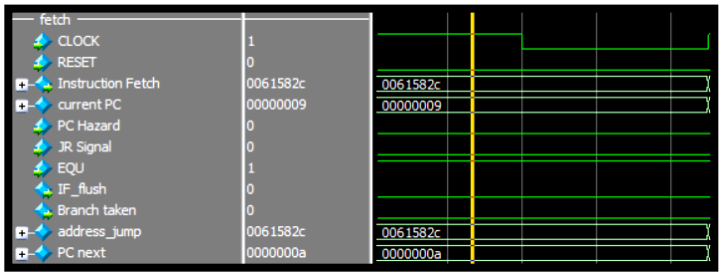

DECODE
Instruction Decode 00434027
Register File 00000000 00000...
PC Decode 00000008
Write Data 00000008
AluResult_EXE 0000000f
AluResult_MEM 00000008
Data_MEM 00000000
Register Destination 06
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 00004027
Rt 03
Rs 02
Rd 08
Forward A mux Result 0000000b
Forward B mux Result 0000000c
JR_Signal 0
control unit signal 1022

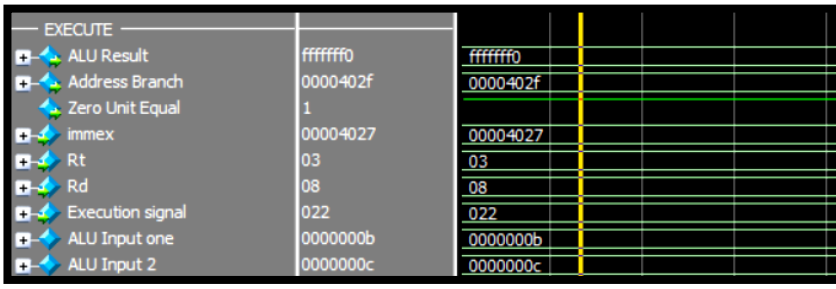

MEMORY
ALU Result Memory 0000000f
write_data 0000000c
Memory Signal 0
Memory Result 00000000

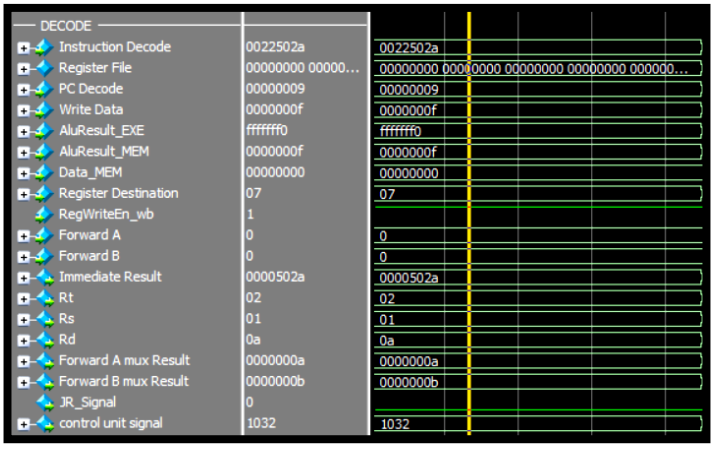

WB
MEM/WB Result 00000000
ALU Writeback result 00000008
Writeback Signal 0
WriteBack Data 00000008

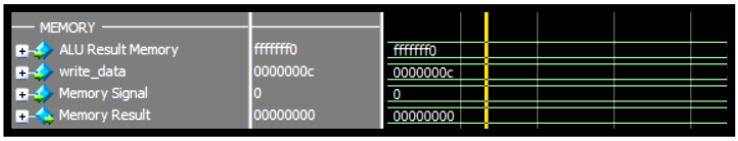

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 02
IFID_Rt 03
IDEXE_Rd 07
EXEMEM_Rd 06
Forward A 0
Forward B 0

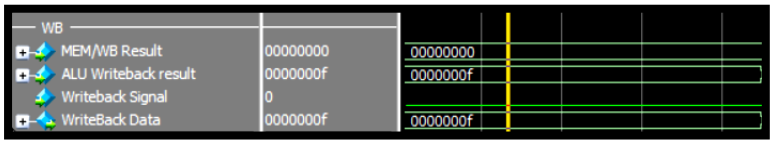
## Cycle 10

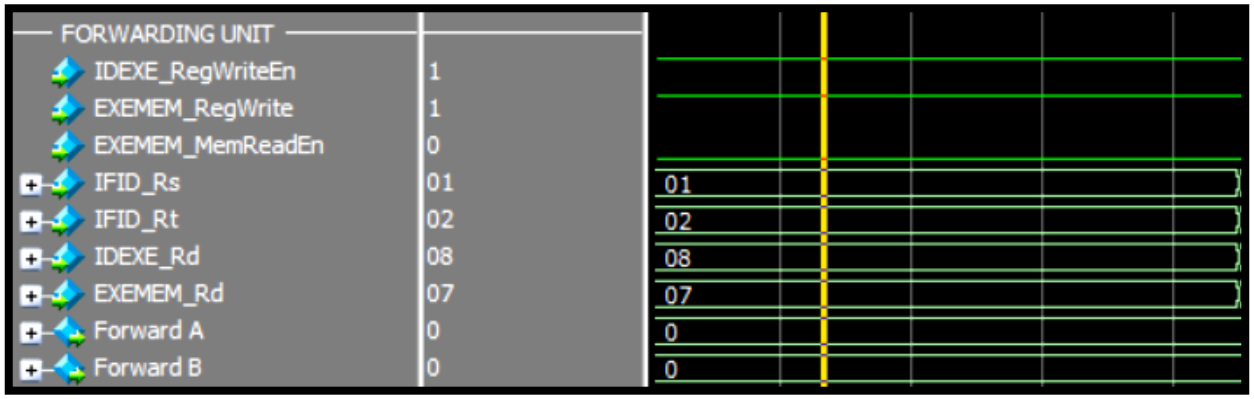

fetch
CLOCK 1
RESET 0
Instruction Fetch 0061582c
current PC 00000009
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 0061582c
PC next 0000000a


EXECUTE
ALU Result ffffffff0
Address Branch 0000402f
Zero Unit Equal 1
immex 00004027
Rt 03
Rd 08
Execution signal 022
ALU Input one 0000000b
ALU Input 2 0000000c

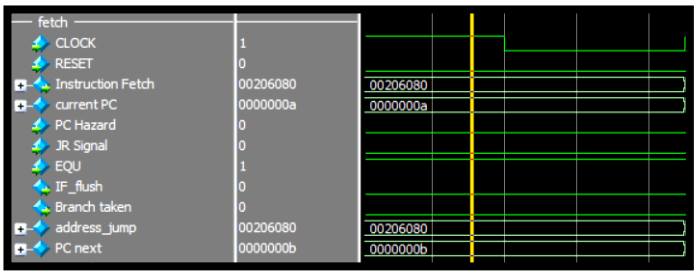

DECODE
Instruction Decode 0022502a
Register File 00000000 00000...
PC Decode 00000009
Write Data 0000000f
AluResult_EXE ffffffff0
AluResult_MEM 0000000f
Data_MEM 00000000
Register Destination 07
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 0000502a
Rt 02
Rs 01
Rd 0a
Forward A mux Result 0000000a
Forward B mux Result 0000000b
JR_Signal 0
control unit signal 1032

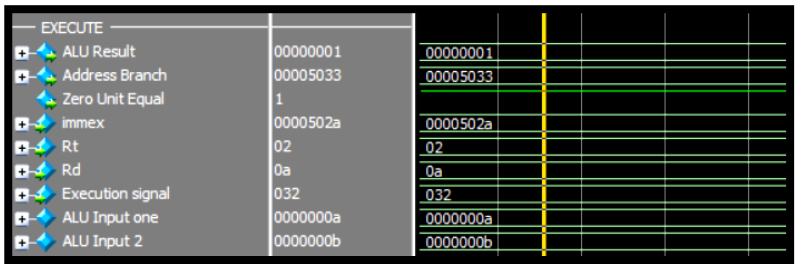

MEMORY
ALU Result Memory ffffffff0
write_data 0000000c
Memory Signal 0
Memory Result 00000000

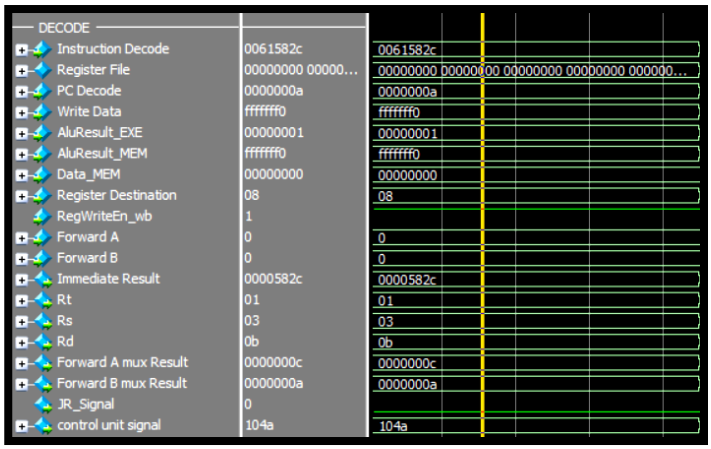

WB
MEM/WB Result 00000000
ALU Writeback result 0000000f
Writeback Signal 0
WriteBack Data 0000000f

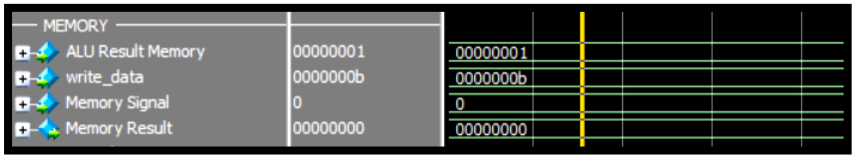

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 01
IFID_Rt 02
IDEXE_Rd 08
EXEMEM_Rd 07
Forward A 0
Forward B 0

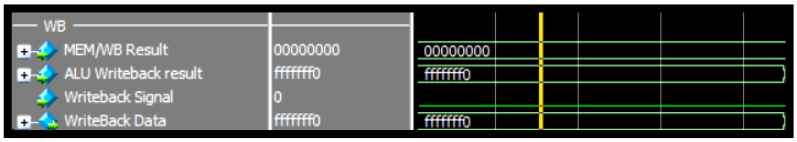
# Cycle 11

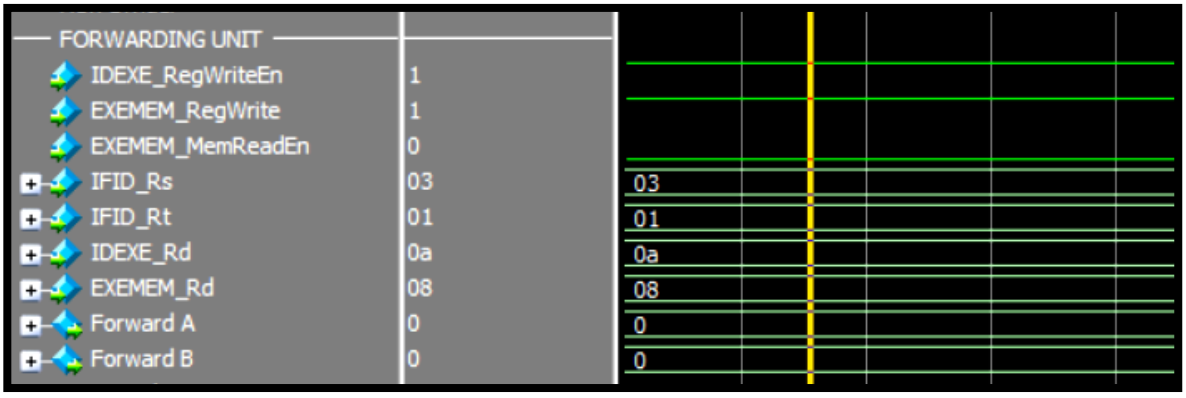

fetch
CLOCK 1
RESET 0
Instruction Fetch 00206080
current PC 0000000a
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 00206080
PC next 0000000b


EXECUTE
ALU Result 00000001
Address Branch 00005033
Zero Unit Equal 1
immex 0000502a
Rt 02
Rd 0a
Execution signal 032
ALU Input one 0000000a
ALU Input 2 0000000b

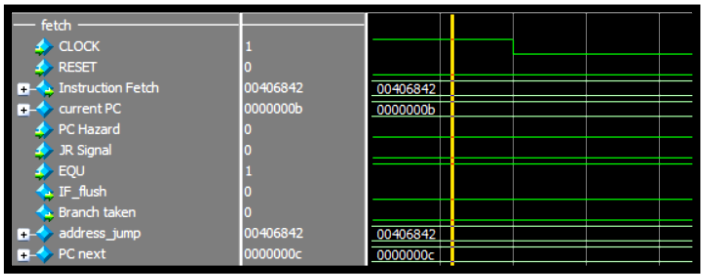

DECODE
Instruction Decode 0061582c
Register File 00000000 00000...
PC Decode 0000000a
Write Data ffffffff0
AluResult_EXE 00000001
AluResult_MEM ffffffff0
Data_MEM 00000000
Register Destination 08
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 0000582c
Rt 01
Rs 03
Rd 0b
Forward A mux Result 0000000c
Forward B mux Result 0000000a
JR_Signal 0
control unit signal 104a

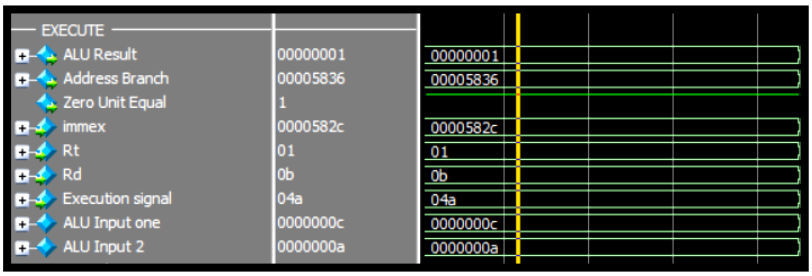

MEMORY
ALU Result Memory 00000001
write_data 0000000b
Memory Signal 0
Memory Result 00000000

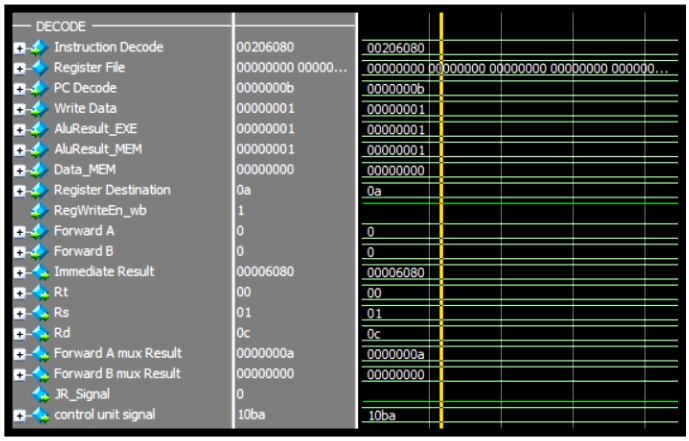

WB
MEM/WB Result 00000000
ALU Writeback result ffffffff0
Writeback Signal 0
WriteBack Data ffffffff0

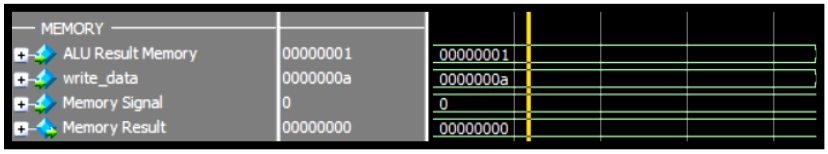

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 03
IFID_Rt 01
IDEXE_Rd 0a
EXEMEM_Rd 08
Forward A 0
Forward B 0

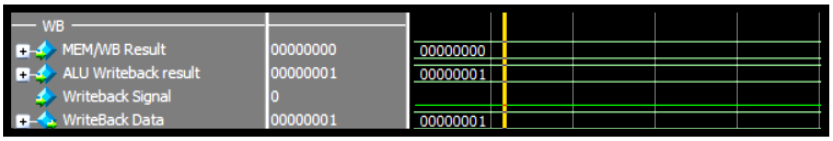
# Cycle 12

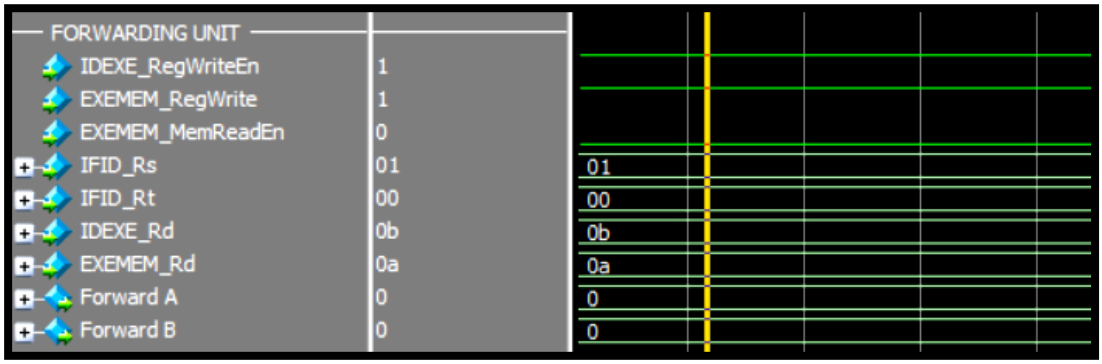

fetch
CLOCK 1
RESET 0
Instruction Fetch 00406842
current PC 0000000b
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 00406842
PC next 0000000c


EXECUTE
ALU Result 00000001
Address Branch 00005836
Zero Unit Equal 1
immex 0000582c
Rt 01
Rd 0b
Execution signal 04a
ALU Input one 0000000c
ALU Input 2 0000000a

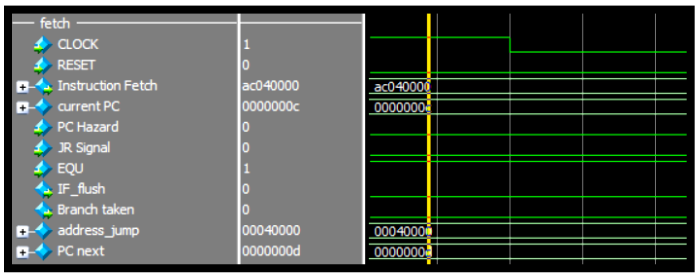

DECODE
Instruction Decode 00206080
Register File 00000000 00000...
PC Decode 0000000b
Write Data 00000001
AluResult_EXE 00000001
AluResult_MEM 00000001
Data_MEM 00000000
Register Destination 0a
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 00006080
Rt 00
Rs 01
Rd 0c
Forward A mux Result 0000000a
Forward B mux Result 00000000
JR_Signal 0
control unit signal 10ba

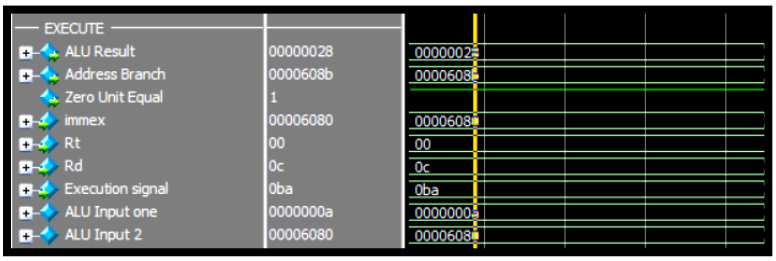

MEMORY
ALU Result Memory 00000001
write_data 0000000a
Memory Signal 0
Memory Result 00000000

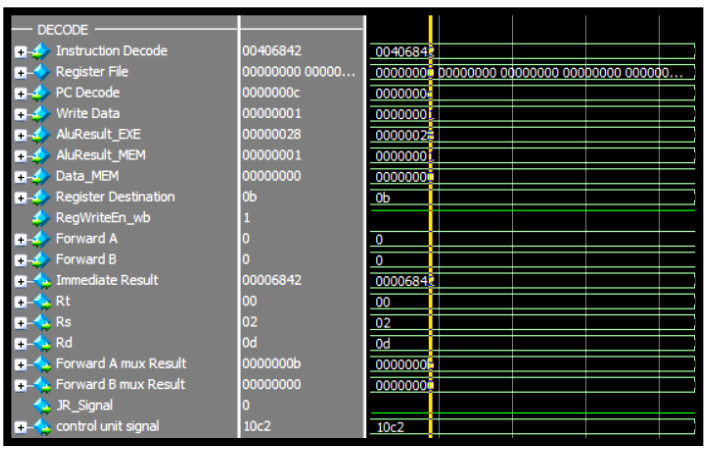

WB
MEM/WB Result 00000000
ALU Writeback result 00000001
Writeback Signal 0
WriteBack Data 00000001

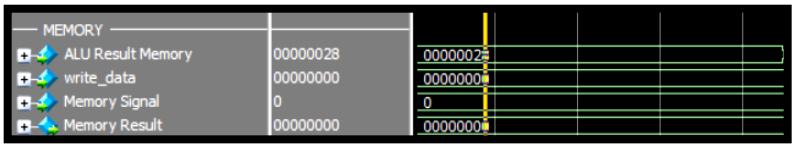

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 01
IFID_Rt 00
IDEXE_Rd 0b
EXEMEM_Rd 0a
Forward A 0
Forward B 0

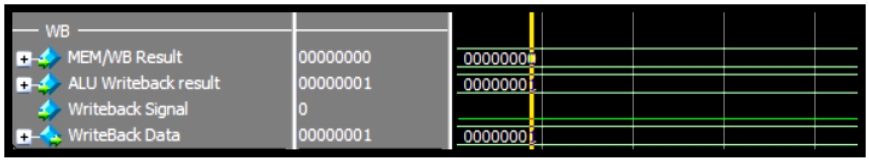
## Cycle 13

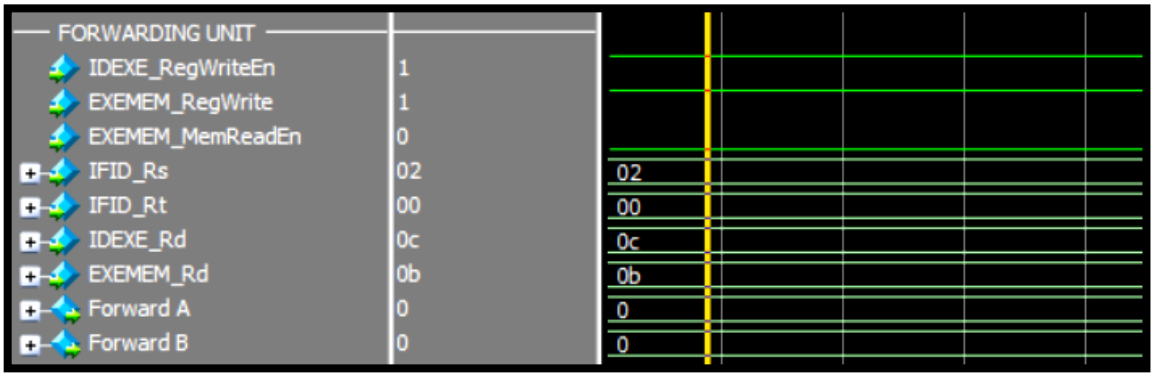

fetch
CLOCK 1
RESET 0
Instruction Fetch ac040000
current PC 0000000c
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 00040000
PC next 0000000d


EXECUTE
ALU Result 00000028
Address Branch 0000608b
Zero Unit Equal 1
immex 00006080
Rt 00
Rd 0c
Execution signal 0ba
ALU Input one 0000000a
ALU Input 2 00006080

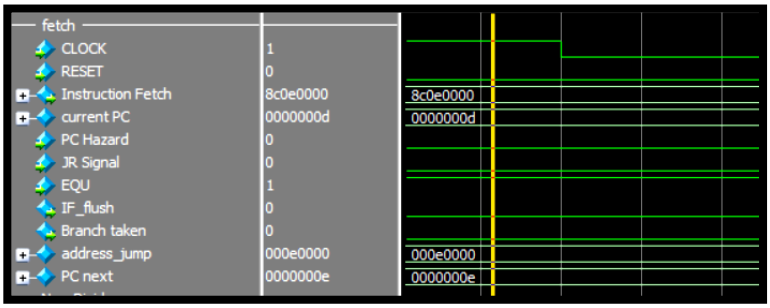

DECODE
Instruction Decode 00406842
Register File 00000000 00000...
PC Decode 0000000c
Write Data 00000001
AluResult_EXE 00000028
AluResult_MEM 00000001
Data_MEM 00000000
Register Destination 0b
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 00006842
Rt 00
Rs 02
Rd 0d
Forward A mux Result 0000000b
Forward B mux Result 00000000
JR_Signal 0
control unit signal 10c2

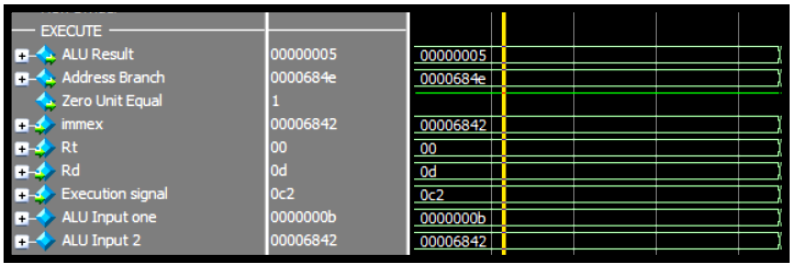

MEMORY
ALU Result Memory 00000028
write_data 00000000
Memory Signal 0
Memory Result 00000000

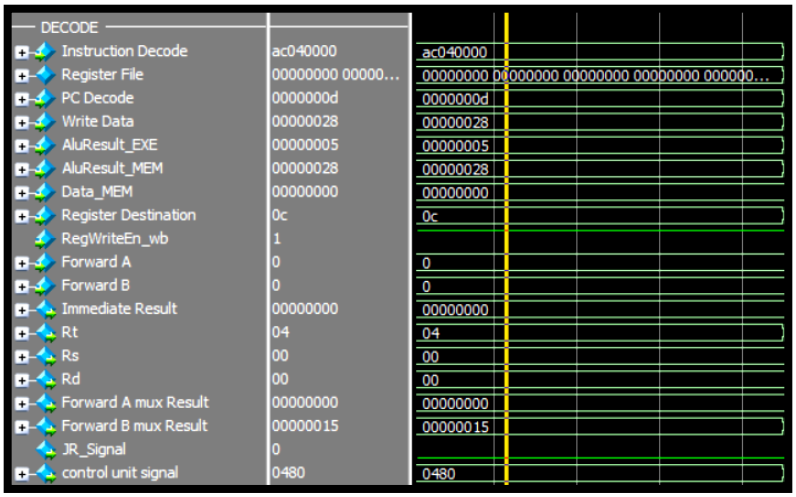

WB
MEM/WB Result 00000000
ALU Writeback result 00000001
Writeback Signal 0
WriteBack Data 00000001

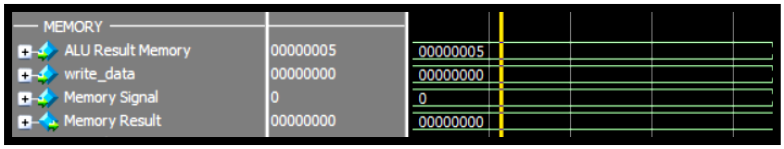

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 02
IFID_Rt 00
IDEXE_Rd 0c
EXEMEM_Rd 0b
Forward A 0
Forward B 0

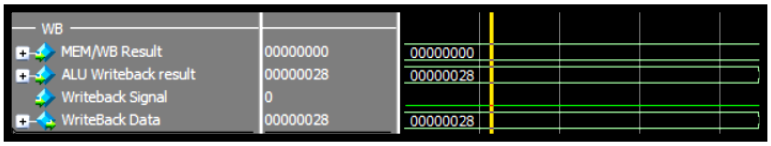
# Cycle 14

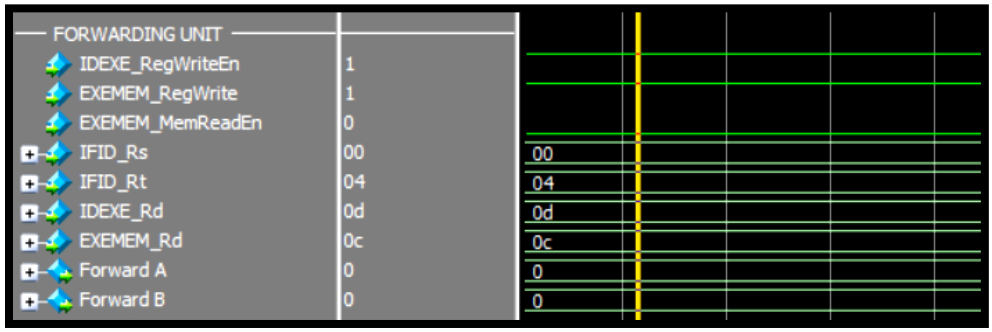

fetch
CLOCK 1
RESET 0
Instruction Fetch 8c0e0000
current PC 0000000d
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 000e0000
PC next 0000000e


EXECUTE
ALU Result 00000005
Address Branch 0000684e
Zero Unit Equal 1
immex 00006842
Rt 00
Rd 0d
Execution signal 0c2
ALU Input one 0000000b
ALU Input 2 00006842

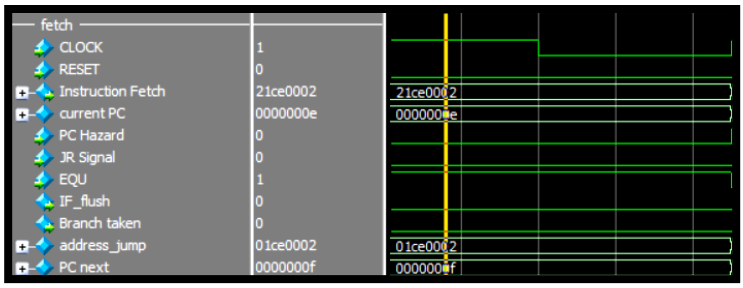

DECODE
Instruction Decode ac040000
Register File 00000000 00000...
PC Decode 0000000d
Write Data 00000028
AluResult_EXE 00000005
AluResult_MEM 00000028
Data_MEM 00000000
Register Destination 0c
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 00000000
Rt 04
Rs 00
Rd 00
Forward A mux Result 00000000
Forward B mux Result 00000015
JR_Signal 0
control unit signal 0480

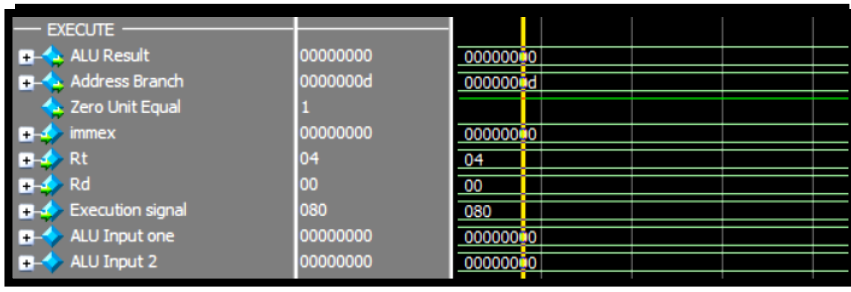

MEMORY
ALU Result Memory 00000005
write_data 00000000
Memory Signal 0
Memory Result 00000000

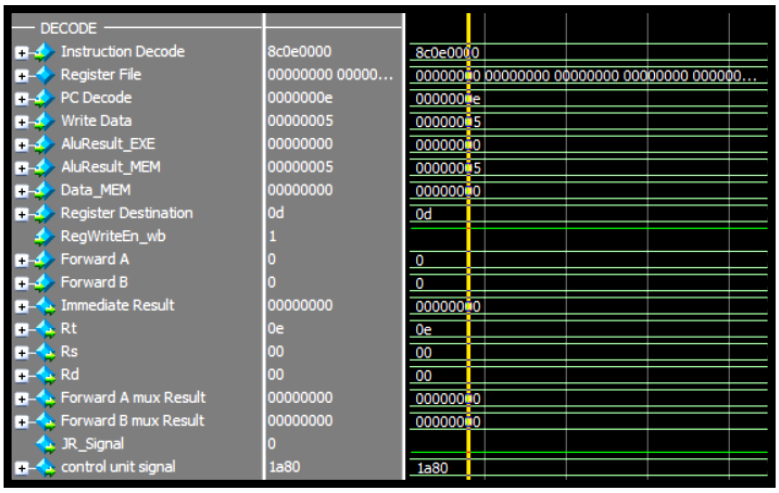

WB
MEM/WB Result 00000000
ALU Writeback result 00000028
Writeback Signal 0
WriteBack Data 00000028

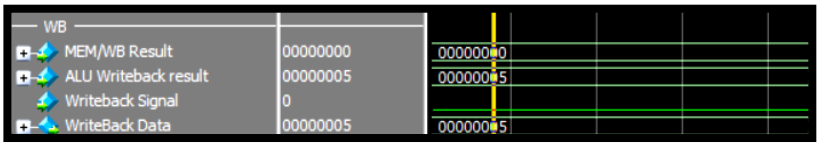

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 00
IFID_Rt 04
IDEXE_Rd 0d
EXEMEM_Rd 0c
Forward A 0
Forward B 0

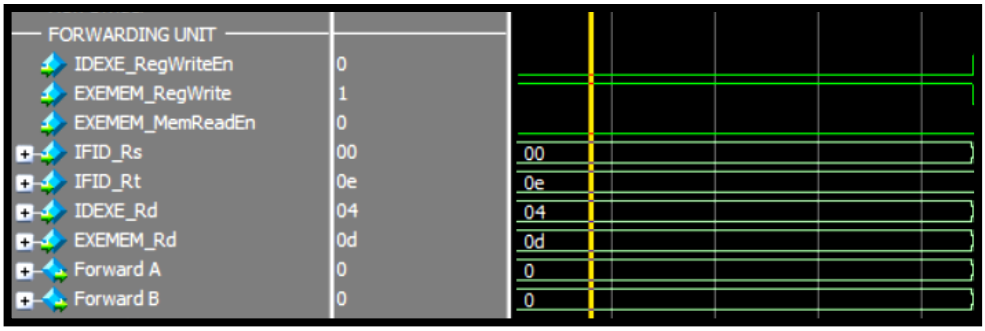
## Cycle 15


fetch
CLOCK 1
RESET 0
Instruction Fetch 21ce0002
current PC 0000000e
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 01ce0002
PC next 0000000f

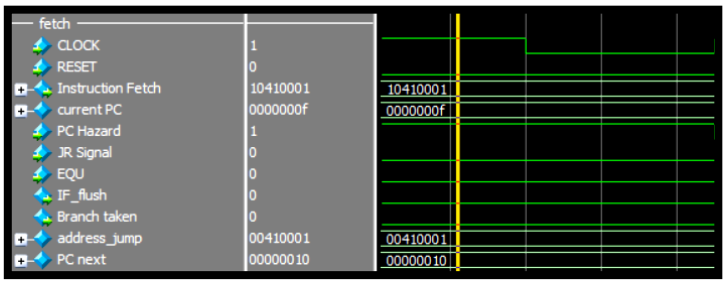

EXECUTE
ALU Result 00000000
Address Branch 0000000d
Zero Unit Equal 1
immex 00000000
Rt 04
Rd 00
Execution signal 080
ALU Input one 00000000
ALU Input 2 00000000

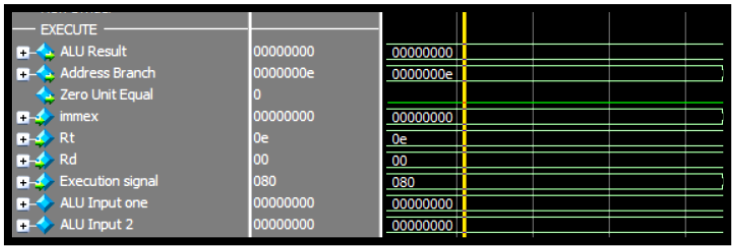

DECODE
Instruction Decode 8c0e0000
Register File 00000000 00000...
PC Decode 0000000e
Write Data 00000005
AluResult_EXE 00000000
AluResult_MEM 00000005
Data_MEM 00000000
Register Destination 0d
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 00000000
Rt 0e
Rs 00
Rd 00
Forward A mux Result 00000000
Forward B mux Result 00000000
JR_Signal 0
control unit signal 1a80

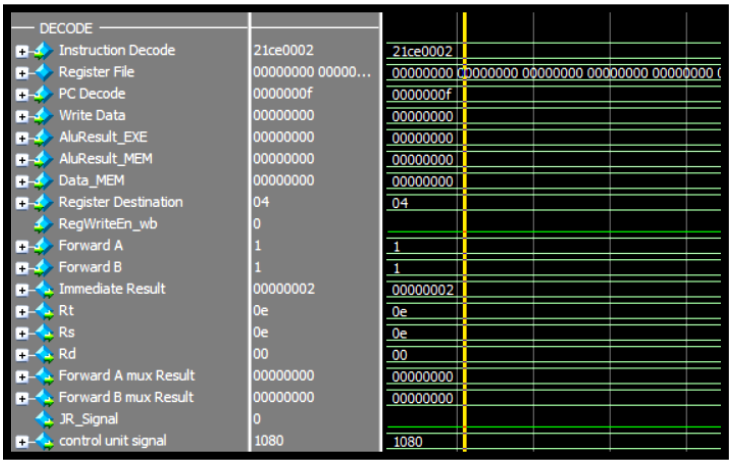

WB
MEM/WB Result 00000000
ALU Writeback result 00000005
Writeback Signal 0
WriteBack Data 00000005

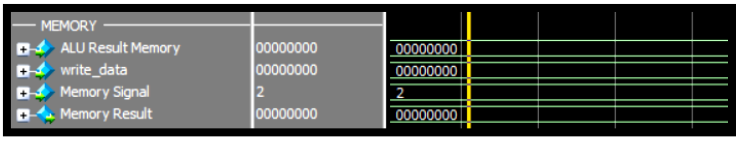

FORWARDING UNIT
IDEXE_RegWriteEn 0
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 00
IFID_Rt 0e
IDEXE_Rd 04
EXEMEM_Rd 0d
Forward A 0
Forward B 0

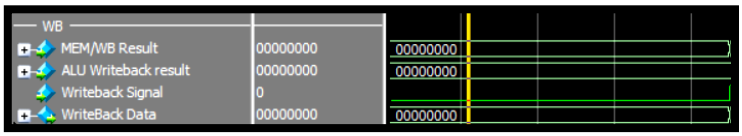
## Cycle 16

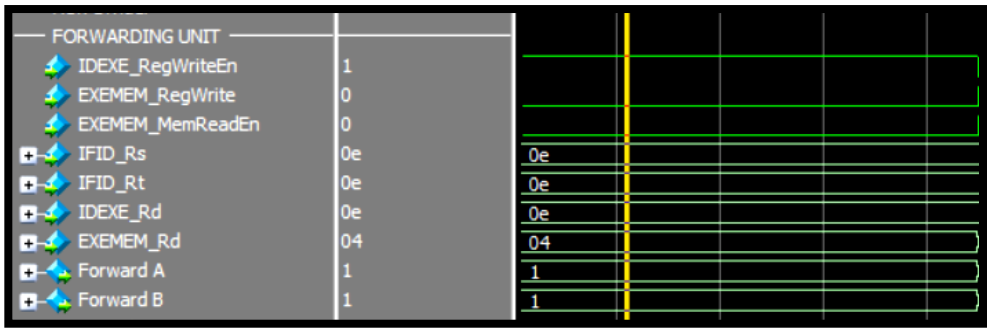

fetch
CLOCK 1
RESET 0
Instruction Fetch 10410001
current PC 0000000f
PC Hazard 1
JR Signal 0
EQU 0
IF_flush 0
Branch taken 0
address_jump 00410001
PC next 00000010


EXECUTE
ALU Result 00000000
Address Branch 0000000e
Zero Unit Equal 0
immex 00000000
Rt 0e
Rd 00
Execution signal 080
ALU Input one 00000000
ALU Input 2 00000000

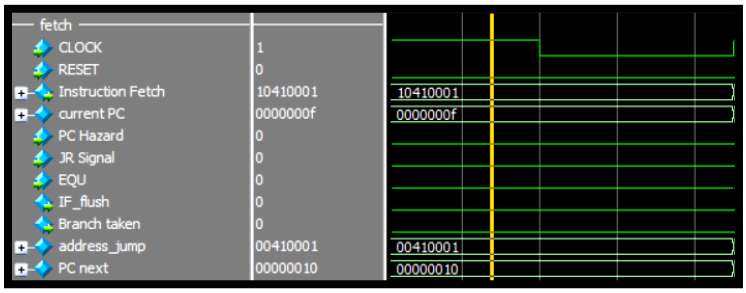

DECODE
Instruction Decode 21ce0002
Register File 00000000 00000...
PC Decode 0000000f
Write Data 00000000
AluResult_EXE 00000000
AluResult_MEM 00000000
Data_MEM 00000000
Register Destination 04
RegWriteEn_wb 0
Forward A 1
Forward B 1
Immediate Result 00000002
Rt 0e
Rs 0e
Rd 00
Forward A mux Result 00000000
Forward B mux Result 00000000
JR_Signal 0
control unit signal 1080

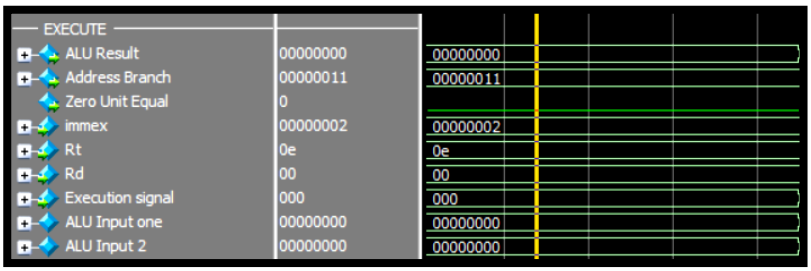

MEMORY
ALU Result Memory 00000000
write_data 00000000
Memory Signal 2
Memory Result 00000000

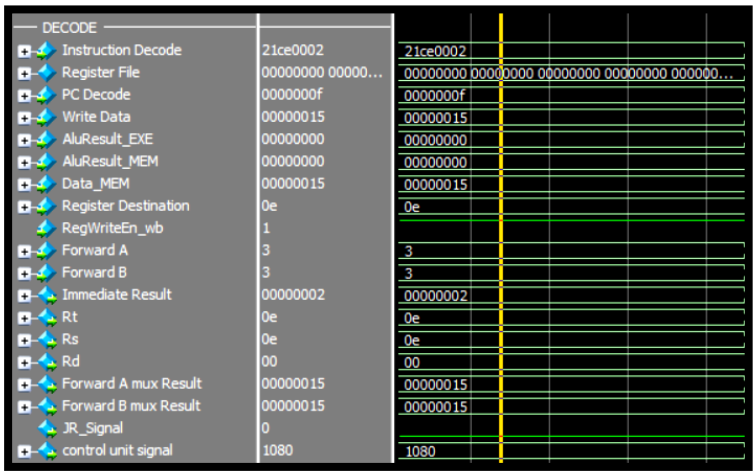

WB
MEM/WB Result 00000000
ALU Writeback result 00000000
Writeback Signal 0
WriteBack Data 00000000

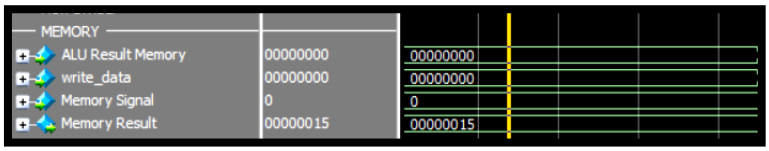

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 0e
IFID_Rt 0e
IDEXE_Rd 0e
EXEMEM_Rd 04
Forward A 1
Forward B 1

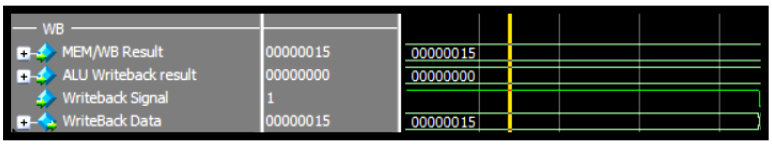
## Cycle 17

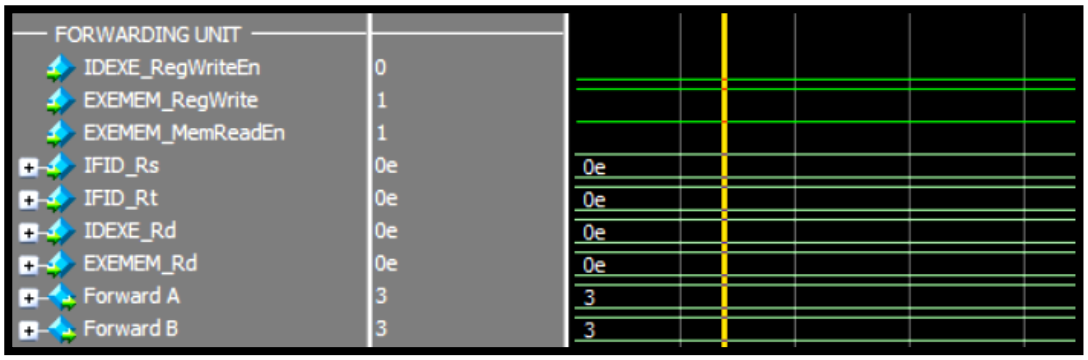

fetch
CLOCK 1
RESET 0
Instruction Fetch 10410001
current PC 0000000f
PC Hazard 0
JR Signal 0
EQU 0
IF_flush 0
Branch taken 0
address_jump 00410001
PC next 00000010


EXECUTE
ALU Result 00000000
Address Branch 00000011
Zero Unit Equal 0
immex 00000002
Rt 0e
Rd 00
Execution signal 000
ALU Input one 00000000
ALU Input 2 00000000

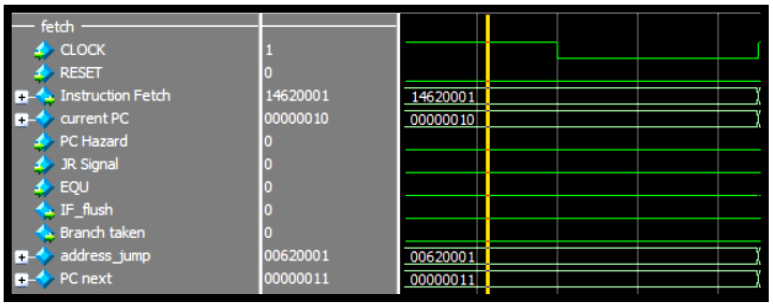

DECODE
Instruction Decode 21ce0002
Register File 00000000 00000...
PC Decode 0000000f
Write Data 00000015
AluResult_EXE 00000000
AluResult_MEM 00000000
Data_MEM 00000015
Register Destination 0e
RegWriteEn_wb 1
Forward A 3
Forward B 3
Immediate Result 00000002
Rt 0e
Rs 0e
Rd 00
Forward A mux Result 00000015
Forward B mux Result 00000015
JR_Signal 0
control unit signal 1080

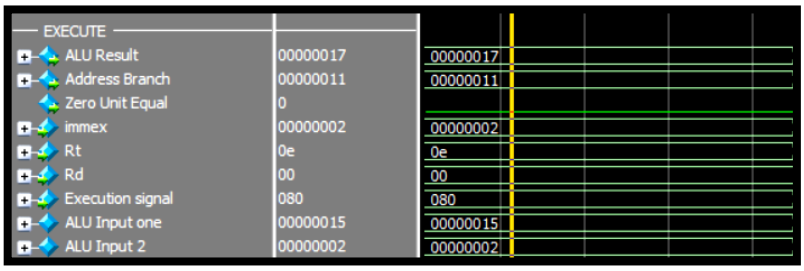

MEMORY
ALU Result Memory 00000000
write_data 00000000
Memory Signal 0
Memory Result 00000015

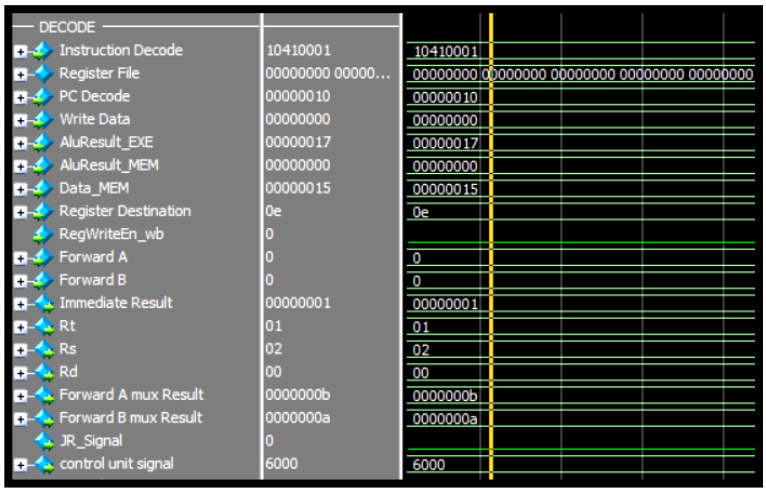

WB
MEM/WB Result 00000015
ALU Writeback result 00000000
Writeback Signal 1
WriteBack Data 00000015

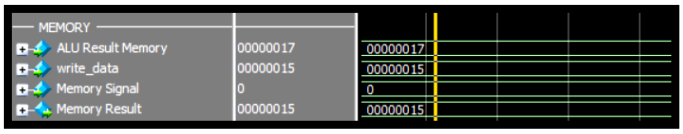

FORWARDING UNIT
IDEXE_RegWriteEn 0
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 1
IFID_Rs 0e
IFID_Rt 0e
IDEXE_Rd 0e
EXEMEM_Rd 0e
Forward A 3
Forward B 3

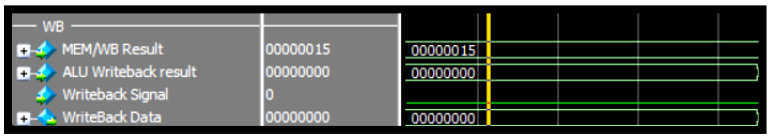
## Cycle 18

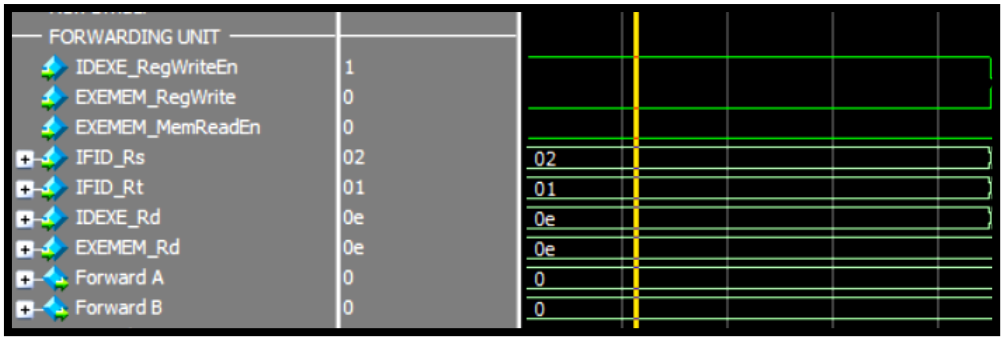

fetch
CLOCK 1
RESET 0
Instruction Fetch 14620001
current PC 00000010
PC Hazard 0
JR Signal 0
EQU 0
IF_flush 0
Branch taken 0
address_jump 00620001
PC next 00000011


EXECUTE
ALU Result 00000017
Address Branch 00000011
Zero Unit Equal 0
immex 00000002
Rt 0e
Rd 00
Execution signal 080
ALU Input one 00000015
ALU Input 2 00000002

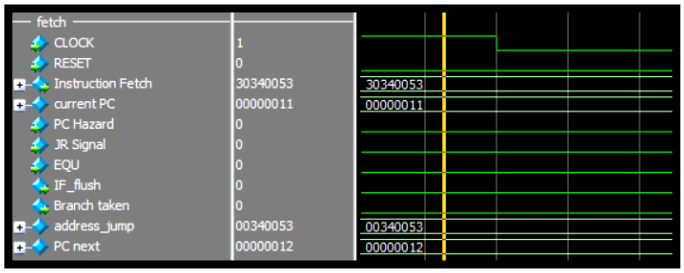

DECODE
Instruction Decode 10410001
Register File 00000000 00000...
PC Decode 00000010
Write Data 00000000
AluResult_EXE 00000017
AluResult_MEM 00000000
Data_MEM 00000015
Register Destination 0e
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00000001
Rt 01
Rs 02
Rd 00
Forward A mux Result 0000000b
Forward B mux Result 0000000a
JR_Signal 0
control unit signal 6000

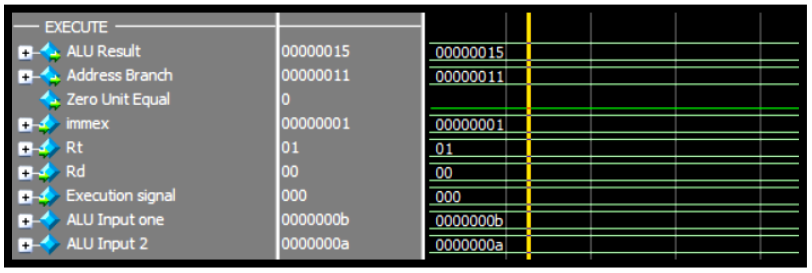

MEMORY
ALU Result Memory 00000017
write_data 00000015
Memory Signal 0
Memory Result 00000015

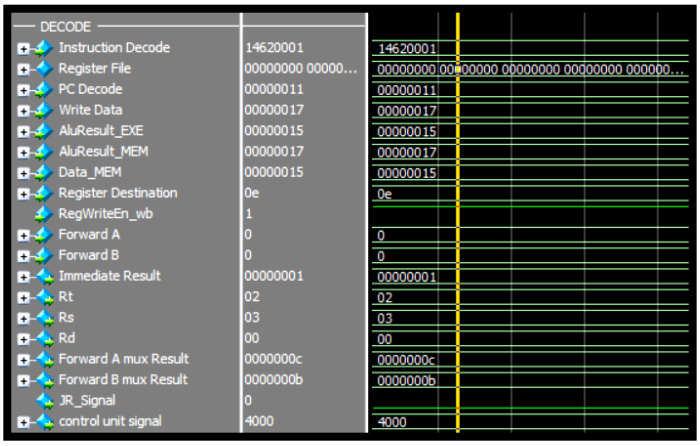

WB
MEM/WB Result 00000015
ALU Writeback result 00000000
Writeback Signal 0
WriteBack Data 00000000

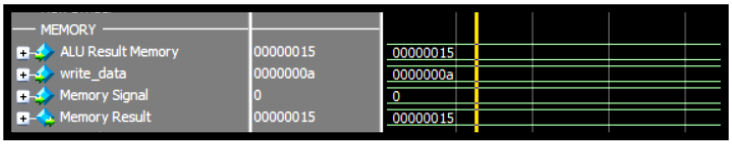

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 02
IFID_Rt 01
IDEXE_Rd 0e
EXEMEM_Rd 0e
Forward A 0
Forward B 0

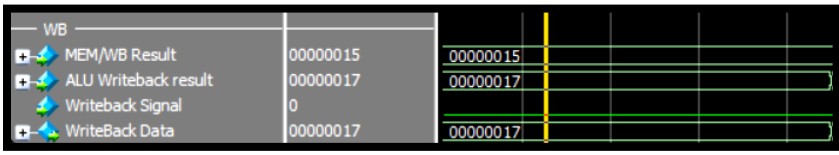
## Cycle 19

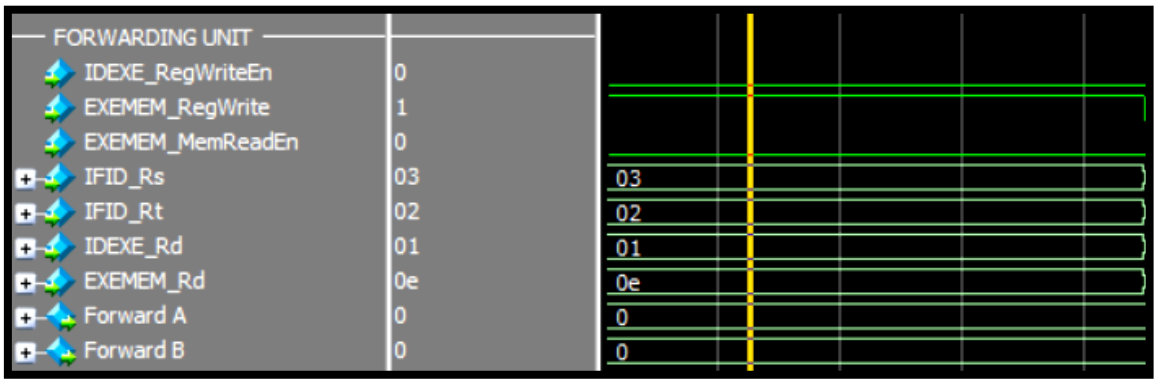

fetch
CLOCK 1
RESET 0
Instruction Fetch 30340053
current PC 00000011
PC Hazard 0
JR Signal 0
EQU 0
IF_flush 0
Branch taken 0
address_jump 00340053
PC next 00000012


EXECUTE
ALU Result 00000015
Address Branch 00000011
Zero Unit Equal 0
immex 00000001
Rt 01
Rd 00
Execution signal 000
ALU Input one 0000000b
ALU Input 2 0000000a

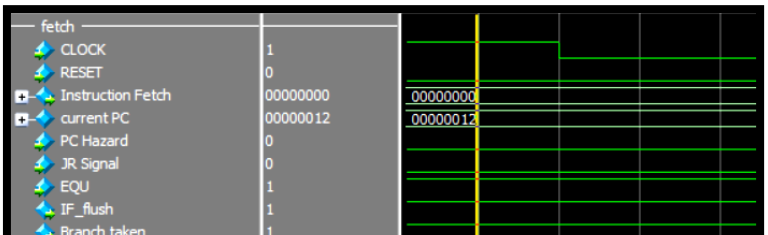

DECODE
Instruction Decode 14620001
Register File 00000000 00000...
PC Decode 00000011
Write Data 00000017
AluResult_EXE 00000015
AluResult_MEM 00000017
Data_MEM 00000015
Register Destination 0e
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 00000001
Rt 02
Rs 03
Rd 00
Forward A mux Result 0000000c
Forward B mux Result 0000000b
JR_Signal 0
control unit signal 4000

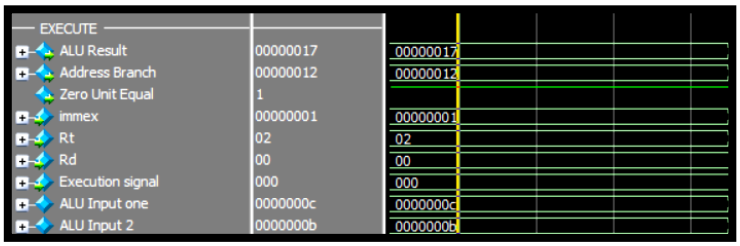

MEMORY
ALU Result Memory 00000015
write_data 0000000a
Memory Signal 0
Memory Result 00000015

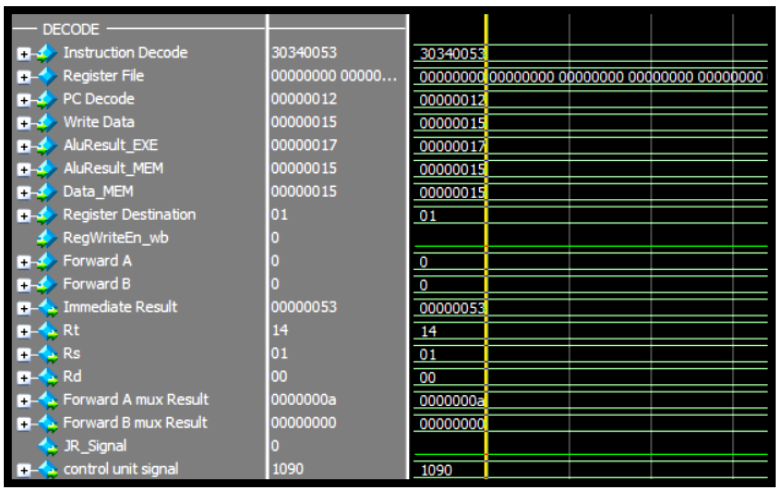

WB
MEM/WB Result 00000015
ALU Writeback result 00000017
Writeback Signal 0
WriteBack Data 00000017

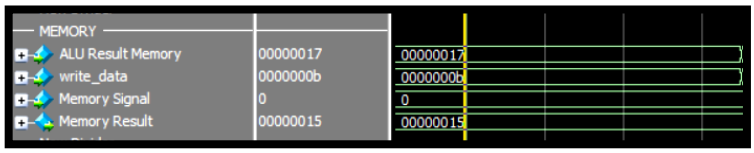

FORWARDING UNIT
IDEXE_RegWriteEn 0
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 03
IFID_Rt 02
IDEXE_Rd 01
EXEMEM_Rd 0e
Forward A 0
Forward B 0

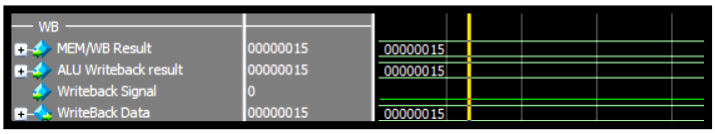
## Cycle 20

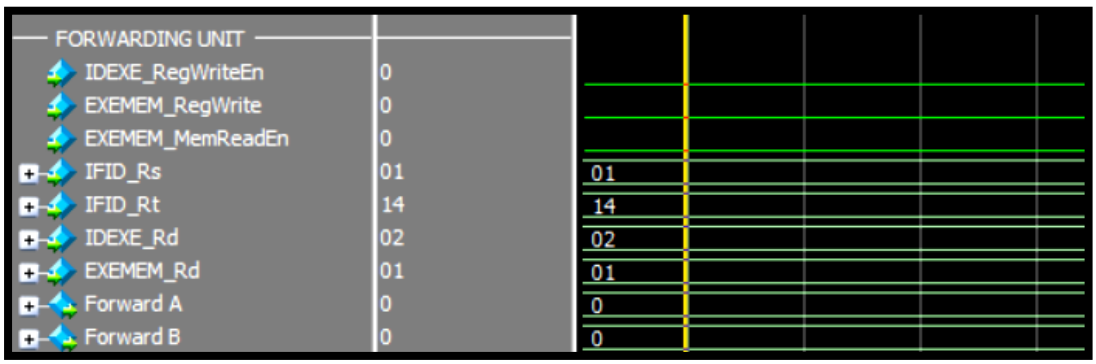

fetch
CLOCK 1
RESET 0
Instruction Fetch 00000000
current PC 00000012
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 1
Branch taken 1


EXECUTE
ALU Result 00000017
Address Branch 00000012
Zero Unit Equal 1
immex 00000001
Rt 02
Rd 00
Execution signal 000
ALU Input one 0000000c
ALU Input 2 0000000b

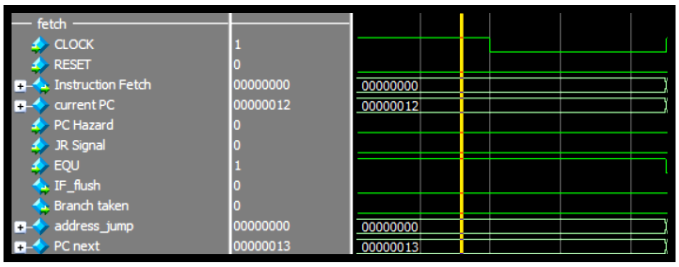

DECODE
Instruction Decode 30340053
Register File 00000000 00000...
PC Decode 00000012
Write Data 00000015
AluResult_EXE 00000017
AluResult_MEM 00000015
Data_MEM 00000015
Register Destination 01
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00000053
Rt 14
Rs 01
Rd 00
Forward A mux Result 0000000a
Forward B mux Result 00000000
JR_Signal 0
control unit signal 1090

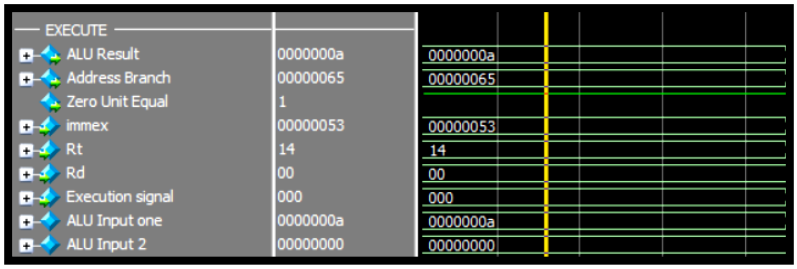

MEMORY
ALU Result Memory 00000017
write_data 0000000b
Memory Signal 0
Memory Result 00000015

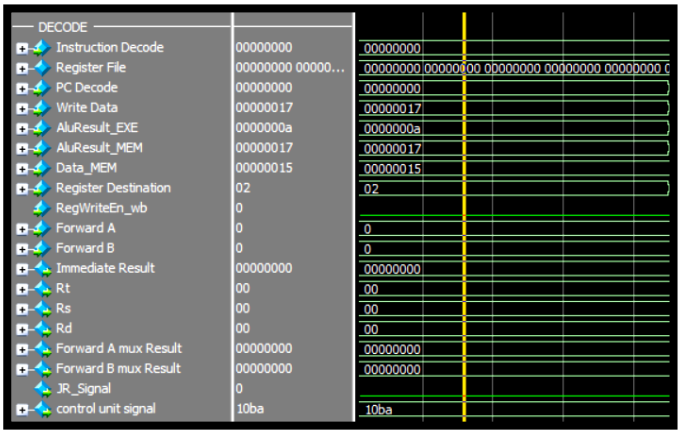

WB
MEM/WB Result 00000015
ALU Writeback result 00000015
Writeback Signal 0
WriteBack Data 00000015

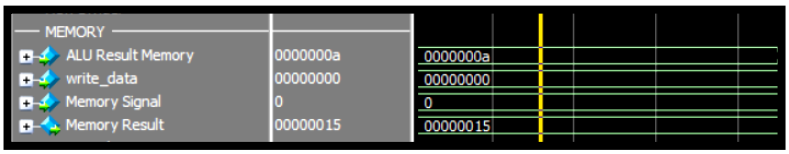

FORWARDING UNIT
IDEXE_RegWriteEn 0
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 01
IFID_Rt 14
IDEXE_Rd 02
EXEMEM_Rd 01
Forward A 0
Forward B 0

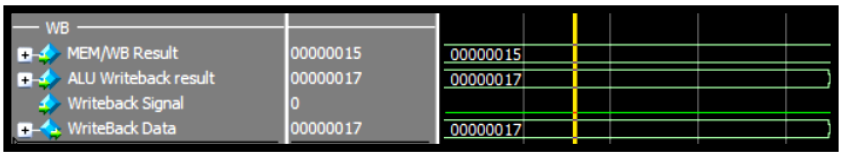
Cycle 21

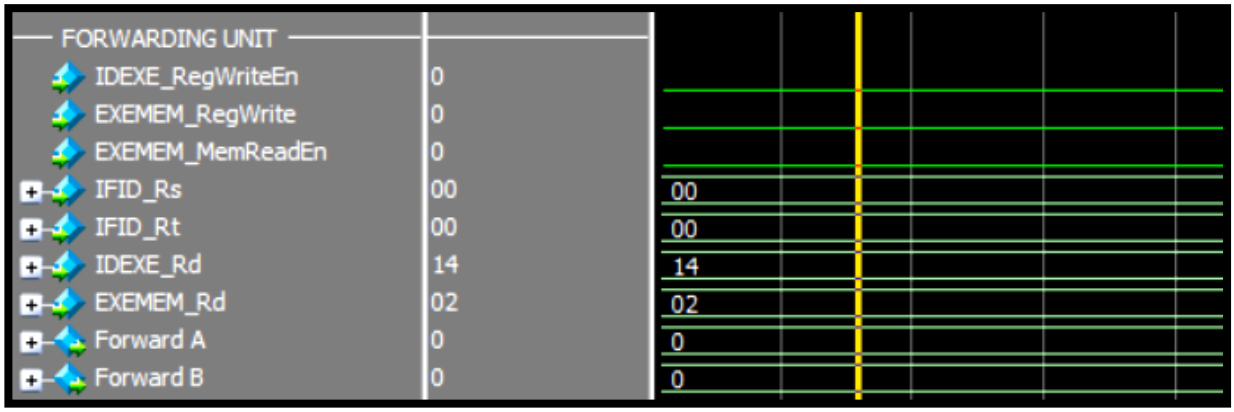

fetch
CLOCK 1
RESET 0
Instruction Fetch 00000000
current PC 00000012
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 00000000
PC next 00000013


EXECUTE
ALU Result 0000000a
Address Branch 00000065
Zero Unit Equal 1
immex 00000053
Rt 14
Rd 00
Execution signal 000
ALU Input one 0000000a
ALU Input 2 00000000

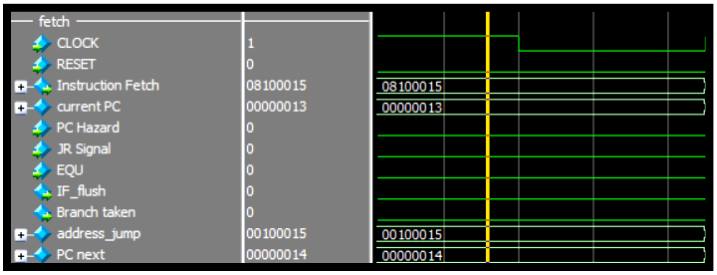

DECODE
Instruction Decode 00000000
Register File 00000000 00000...
PC Decode 00000000
Write Data 00000017
AluResult_EXE 0000000a
AluResult_MEM 00000017
Data_MEM 00000015
Register Destination 02
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00000000
Rt 00
Rs 00
Rd 00
Forward A mux Result 00000000
Forward B mux Result 00000000
JR_Signal 0
control unit signal 10ba

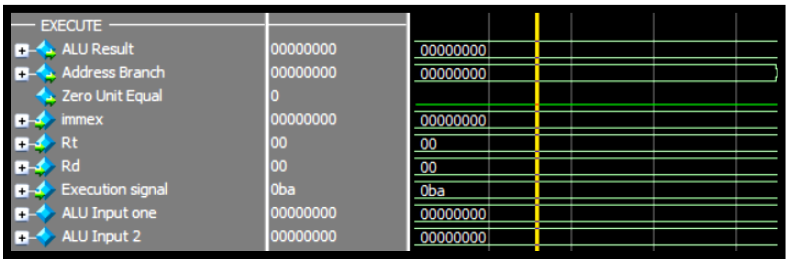

MEMORY
ALU Result Memory 0000000a
write_data 00000000
Memory Signal 0
Memory Result 00000015

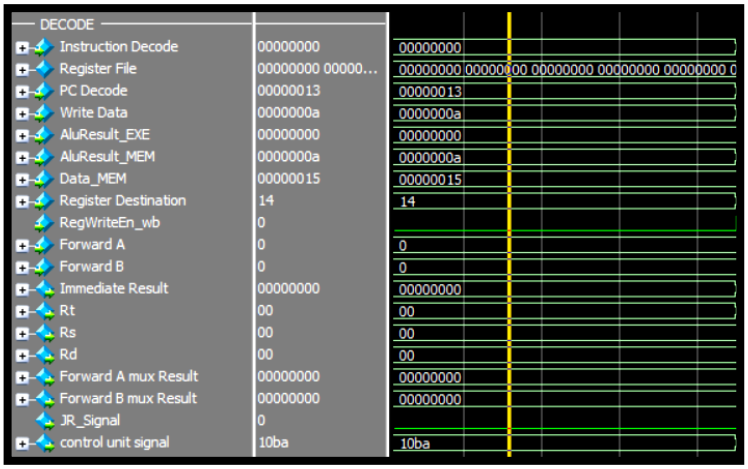

WB
MEM/WB Result 00000015
ALU Writeback result 00000017
Writeback Signal 0
WriteBack Data 00000017

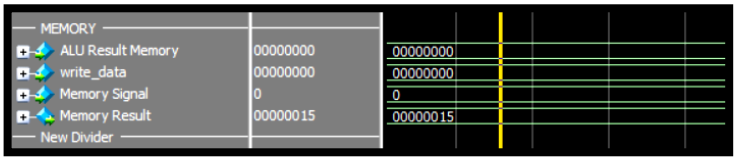

FORWARDING UNIT
IDEXE_RegWriteEn 0
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 00
IFID_Rt 00
IDEXE_Rd 14
EXEMEM_Rd 02
Forward A 0
Forward B 0

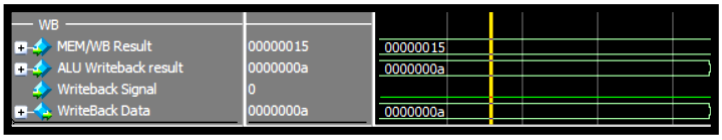
## Cycle 22

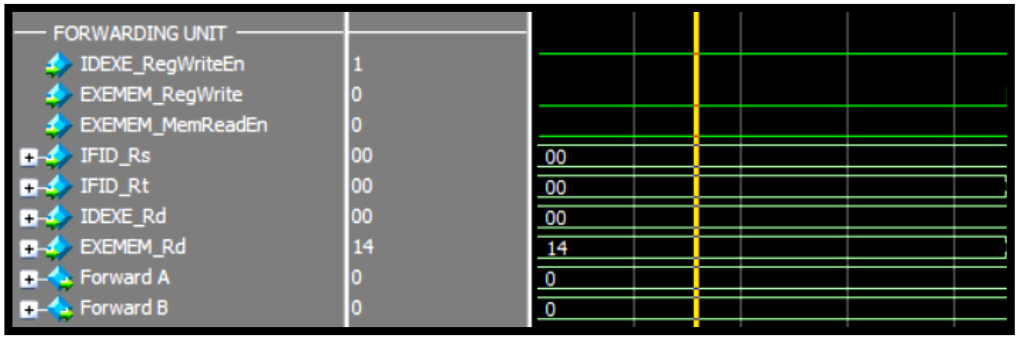

fetch
CLOCK 1
RESET 0
Instruction Fetch 08100015
current PC 00000013
PC Hazard 0
JR Signal 0
EQU 0
IF_flush 0
Branch taken 0
address_jump 00100015
PC next 00000014


EXECUTE
ALU Result 00000000
Address Branch 00000000
Zero Unit Equal 0
immex 00000000
Rt 00
Rd 00
Execution signal 0ba
ALU Input one 00000000
ALU Input 2 00000000

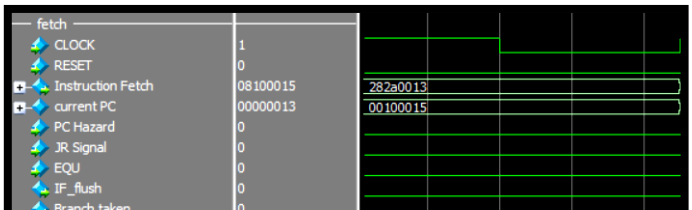

DECODE
Instruction Decode 00000000
Register File 00000000 00000...
PC Decode 00000013
Write Data 0000000a
AluResult_EXE 00000000
AluResult_MEM 0000000a
Data_MEM 00000015
Register Destination 14
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00000000
Rt 00
Rs 00
Rd 00
Forward A mux Result 00000000
Forward B mux Result 00000000
JR_Signal 0
control unit signal 10ba

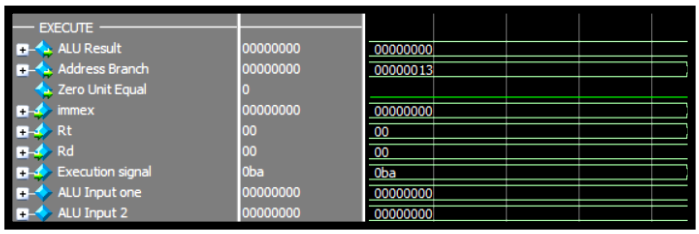

MEMORY
ALU Result Memory 00000000
write_data 00000000
Memory Signal 0
Memory Result 00000015
New Divider

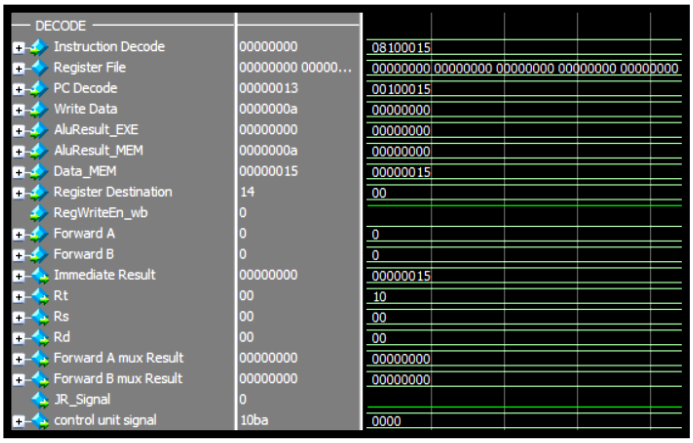

WB
MEM/WB Result 00000015
ALU Writeback result 0000000a
Writeback Signal 0
WriteBack Data 0000000a

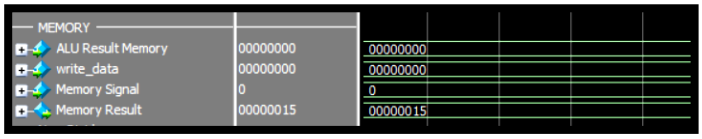

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 00
IFID_Rt 00
IDEXE_Rd 00
EXEMEM_Rd 14
Forward A 0
Forward B 0

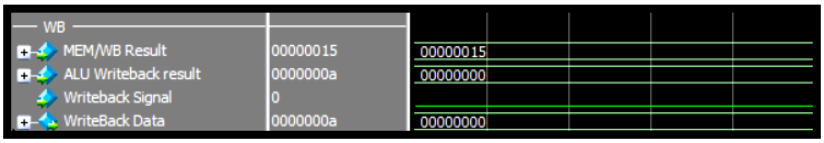
## Cycle 23

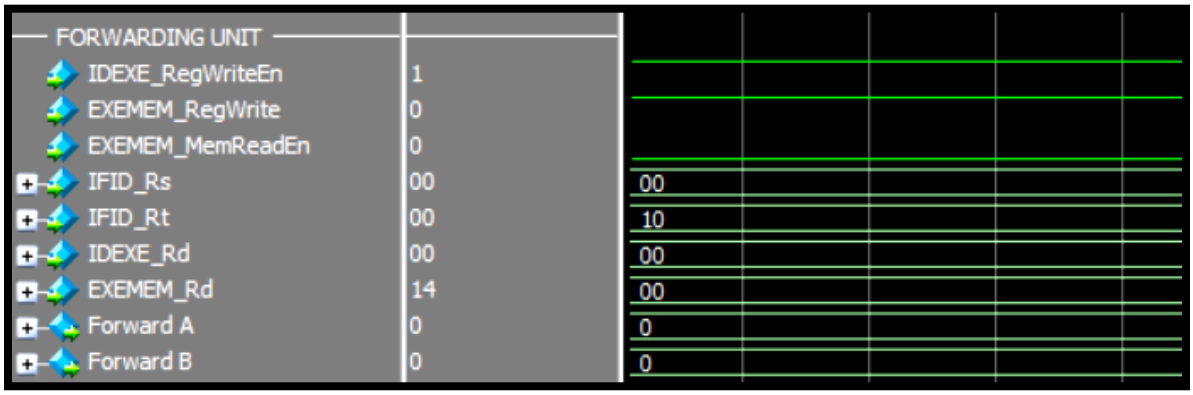

fetch
CLOCK 1
RESET 0
Instruction Fetch 08100015
current PC 00000013
PC Hazard 0
JR Signal 0
EQU 0
IF_flush 0
Branch taken 0


EXECUTE
ALU Result
Address Branch
Zero Unit Equal
immex
Rt
Rd
Execution signal
ALU Input one
ALU Input 2

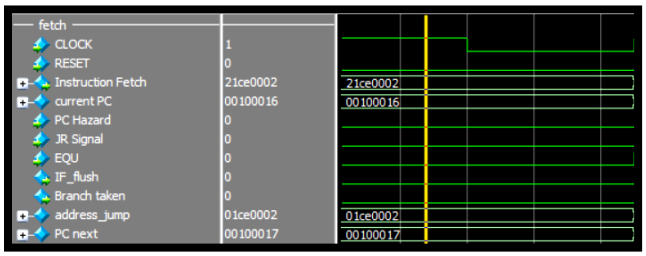

DECODE
Instruction Decode
Register File
PC Decode
Write Data
AluResult_EXE
AluResult_MEM
Data_MEM
Register Destination
RegWriteEn_wb
Forward A
Forward B
Immediate Result
Rt
Rs
Rd
Forward A mux Result
Forward B mux Result
JR_Signal
control unit signal

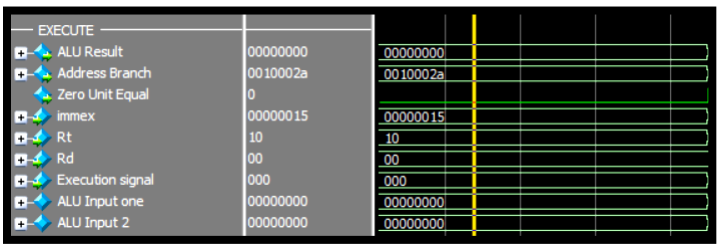

MEMORY
ALU Result Memory
write_data
Memory Signal
Memory Result

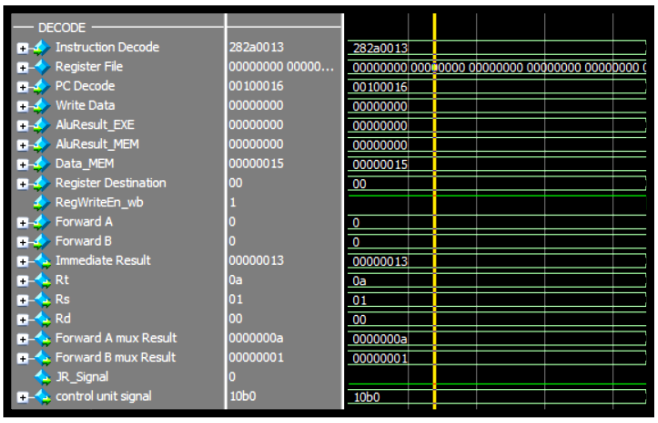

WB
MEM/WB Result
ALU Writeback result
Writeback Signal
WriteBack Data

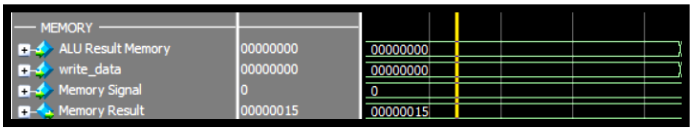

FORWARDING UNIT
IDEXE_RegWriteEn
EXEMEM_RegWrite
EXEMEM_MemReadEn
IFID_Rs
IFID_Rt
IDEXE_Rd
EXEMEM_Rd
Forward A
Forward B

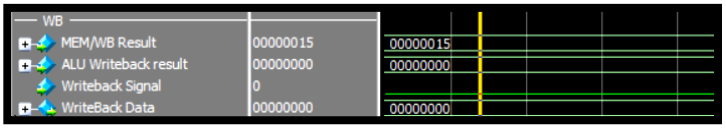
## Cycle 24

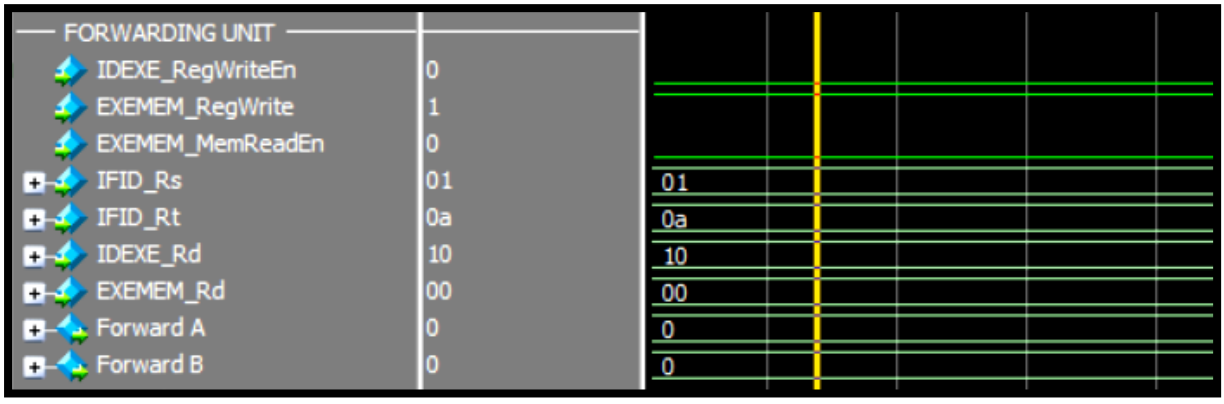

fetch
CLOCK 1
RESET 0
Instruction Fetch 21ce0002
current PC 00100016
PC Hazard 0
JR Signal 0
EQU 0
IF_flush 0
Branch taken 0
address_jump 01ce0002
PC next 00100017


EXECUTE
ALU Result 00000000
Address Branch 0010002a
Zero Unit Equal 0
immex 00000015
Rt 10
Rd 00
Execution signal 000
ALU Input one 00000000
ALU Input 2 00000000

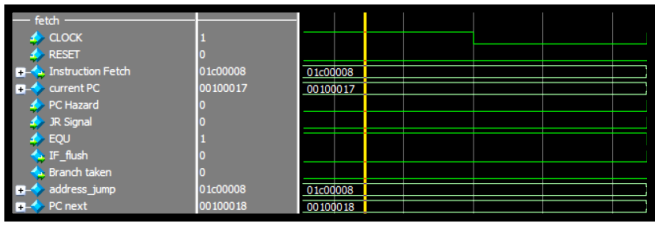

DECODE
Instruction Decode 282a0013
Register File 00000000 00000...
PC Decode 00100016
Write Data 00000000
AluResult_EXE 00000000
AluResult_MEM 00000000
Data_MEM 00000015
Register Destination 00
RegWriteEn_wb 1
Forward A 0
Forward B 0
Immediate Result 00000013
Rt 0a
Rs 01
Rd 00
Forward A mux Result 0000000a
Forward B mux Result 00000001
JR_Signal 0
control unit signal 10b0

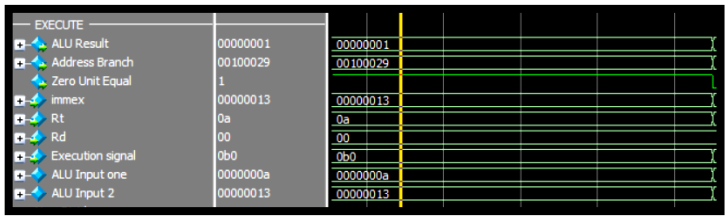

MEMORY
ALU Result Memory 00000000
write_data 00000000
Memory Signal 0
Memory Result 00000015

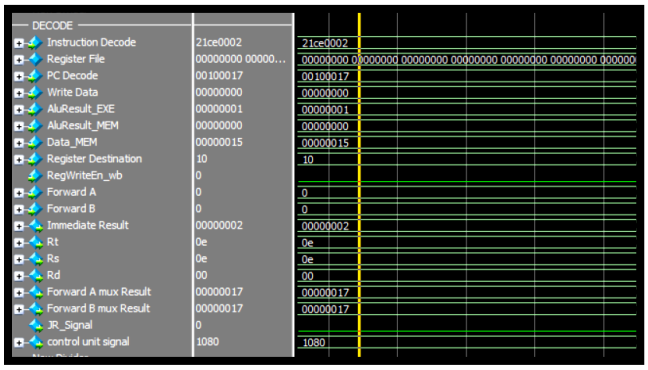

WB
MEM/WB Result 00000015
ALU Writeback result 00000000
Writeback Signal 0
WriteBack Data 00000000

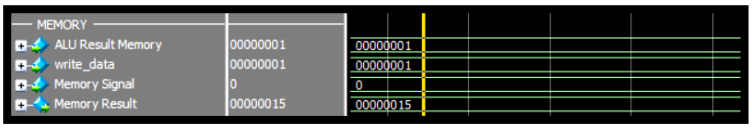

FORWARDING UNIT
IDEXE_RegWriteEn 0
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 01
IFID_Rt 0a
IDEXE_Rd 10
EXEMEM_Rd 00
Forward A 0
Forward B 0

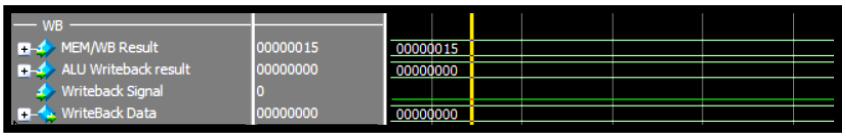
## Cycle 25

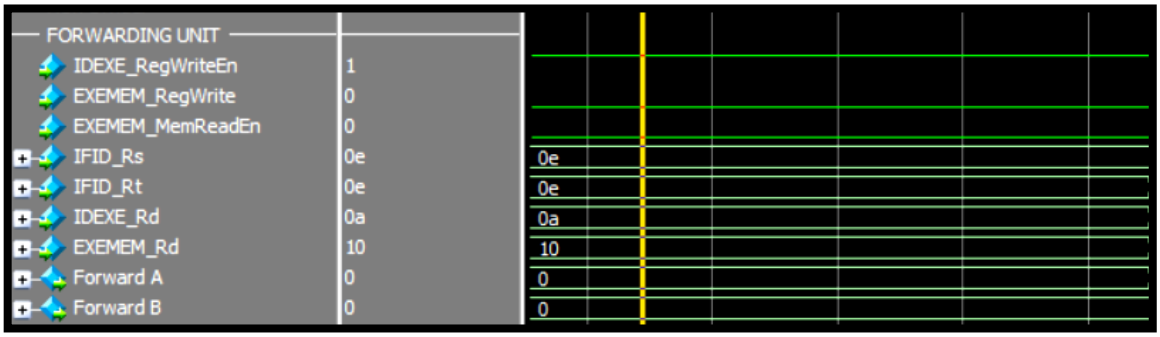

fetch
CLOCK 1
RESET 0
Instruction Fetch 01c00008
current PC 00100017
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 01c00008
PC next 00100018


EXECUTE
ALU Result 00000001
Address Branch 00100029
Zero Unit Equal 1
immex 00000013
Rt 0a
Rd 00
Execution signal 0b0
ALU Input one 0000000a
ALU Input 2 00000013

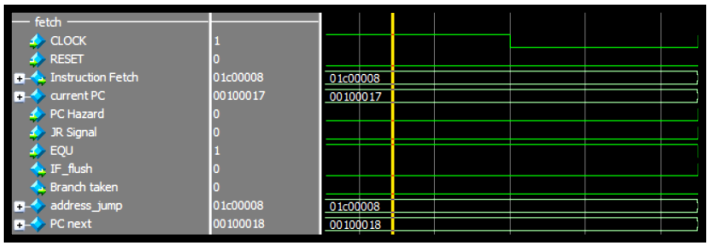

DECODE
Instruction Decode 21ce0002
Register File 00000000 00000...
PC Decode 00100017
Write Data 00000000
AluResult_EXE 00000001
AluResult_MEM 00000000
Data_MEM 00000015
Register Destination 10
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00000002
Rt 0e
Rs 0e
Rd 00
Forward A mux Result 00000017
Forward B mux Result 00000017
JR_Signal 0
control unit signal 1080

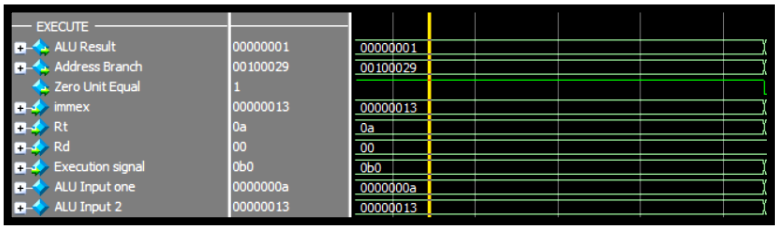

MEMORY
ALU Result Memory 00000001
write_data 00000001
Memory Signal 0
Memory Result 00000015

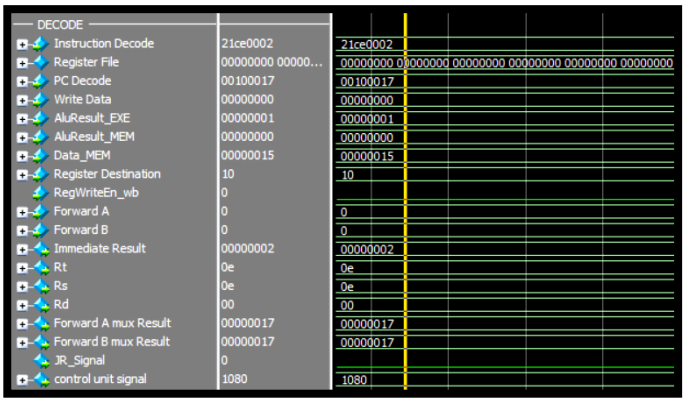

WB
MEM/WB Result 00000015
ALU Writeback result 00000000
Writeback Signal 0
WriteBack Data 00000000

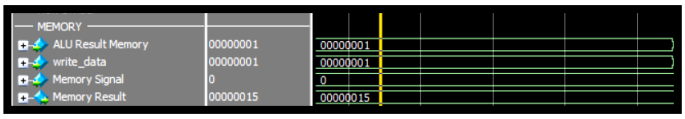

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 0e
IFID_Rt 0e
IDEXE_Rd 0a
EXEMEM_Rd 10
Forward A 0
Forward B 0

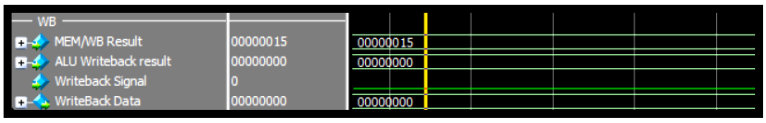
## Cycle 26

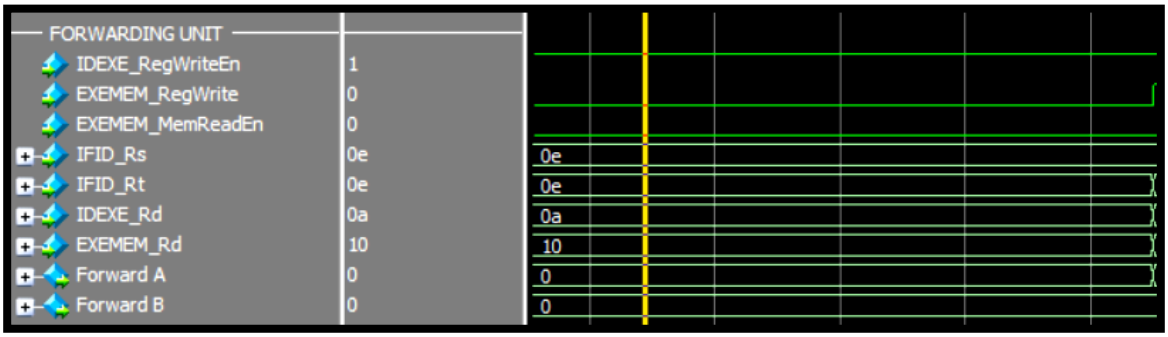

fetch
CLOCK 1
RESET 0
Instruction Fetch 01c00008
current PC 00100017
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 01c00008
PC next 00100018


EXECUTE
ALU Result 00000001
Address Branch 00100029
Zero Unit Equal 1
immex 00000013
Rt 0a
Rd 00
Execution signal 0b0
ALU Input one 0000000a
ALU Input 2 00000013

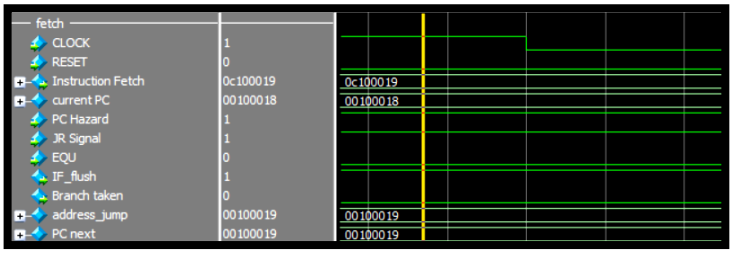

DECODE
Instruction Decode 21ce0002
Register File 00000000 00000...
PC Decode 00100017
Write Data 00000000
AluResult_EXE 00000001
AluResult_MEM 00000000
Data_MEM 00000015
Register Destination 10
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00000002
Rt 0e
Rs 0e
Rd 00
Forward A mux Result 00000017
Forward B mux Result 00000017
JR_Signal 0
control unit signal 1080

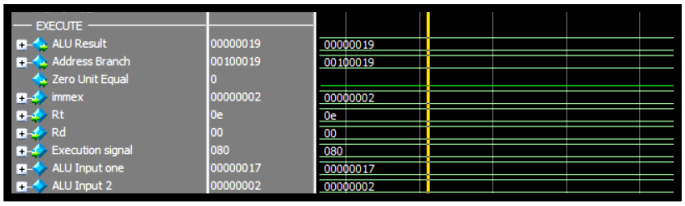

MEMORY
ALU Result Memory 00000001
write_data 00000001
Memory Signal 0
Memory Result 00000015

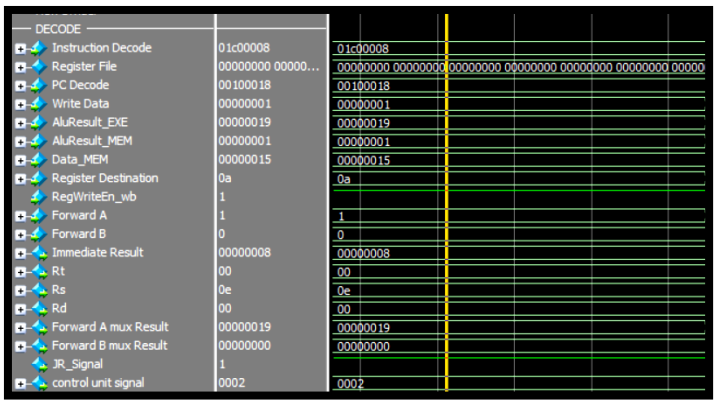

WB
MEM/WB Result 00000015
ALU Writeback result 00000000
Writeback Signal 0
WriteBack Data 00000000

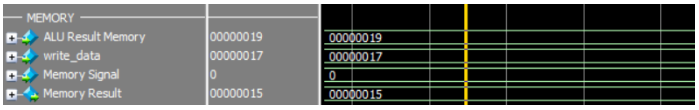

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 0e
IFID_Rt 0e
IDEXE_Rd 0a
EXEMEM_Rd 10
Forward A 0
Forward B 0

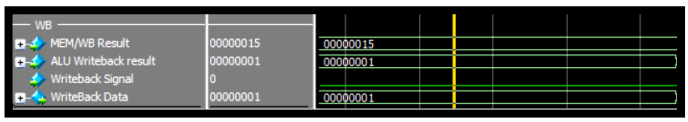
## Cycle 27


fetch
CLOCK 1
RESET 0
Instruction Fetch 0c100019
current PC 00100018
PC Hazard 1
JR Signal 1
EQU 0
IF_flush 1
Branch taken 0
address_jump 00100019
PC next 00100019

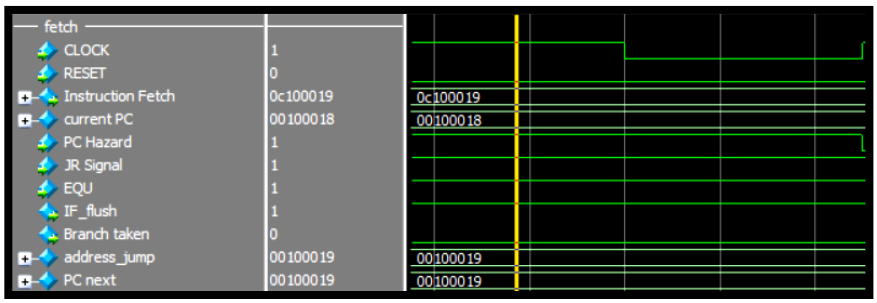

EXECUTE
ALU Result 00000019
Address Branch 00100019
Zero Unit Equal 0
immex 00000002
Rt 0e
Rd 00
Execution signal 080
ALU Input one 00000017
ALU Input 2 00000002

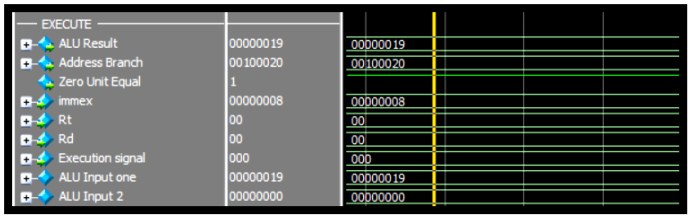

DECODE
Instruction Decode 01c00008
Register File 00000000 00000...
PC Decode 00100018
Write Data 00000001
AluResult_EXE 00000019
AluResult_MEM 00000001
Data_MEM 00000015
Register Destination 0a
RegWriteEn_wb 1
Forward A 1
Forward B 0
Immediate Result 00000008
Rt 00
Rs 0e
Rd 00
Forward A mux Result 00000019
Forward B mux Result 00000000
JR_Signal 1
control unit signal 0002

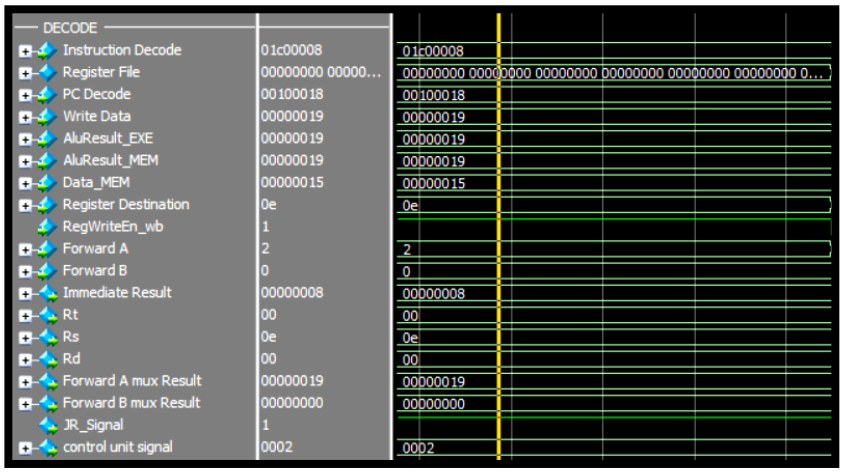

MEMORY
ALU Result Memory 00000019
write_data 00000017
Memory Signal 0
Memory Result 00000015

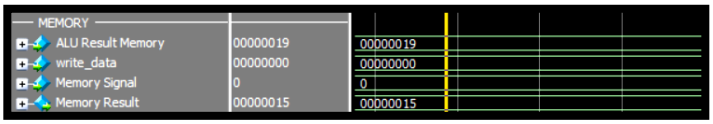

WB
MEM/WB Result 00000015
ALU Writeback result 00000001
Writeback Signal 0
WriteBack Data 00000001

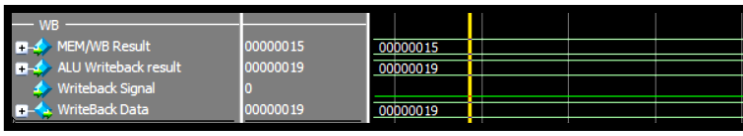
## Cycle 28

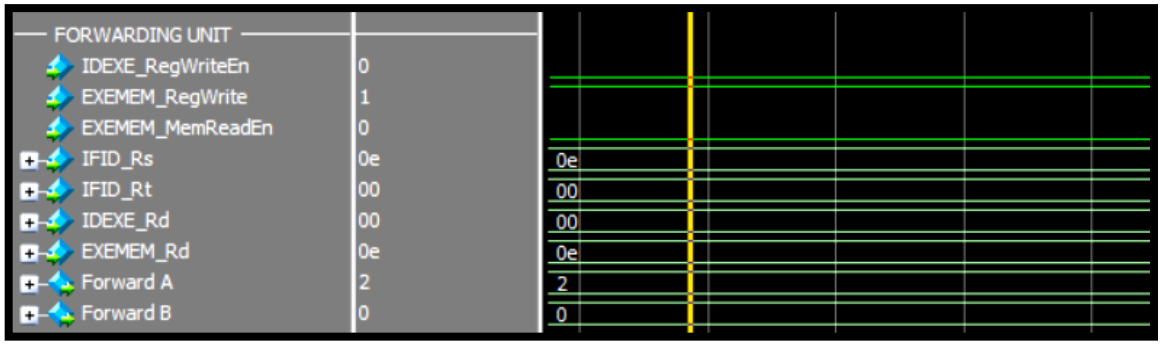

fetch
CLOCK 1
RESET 0
Instruction Fetch 0c100019
current PC 00100018
PC Hazard 1
JR Signal 1
EQU 1
IF_flush 1
Branch taken 0
address_jump 00100019
PC next 00100019


EXECUTE
ALU Result 00000019
Address Branch 00100020
Zero Unit Equal 1
immex 00000008
Rt 00
Rd 00
Execution signal 000
ALU Input one 00000019
ALU Input 2 00000000

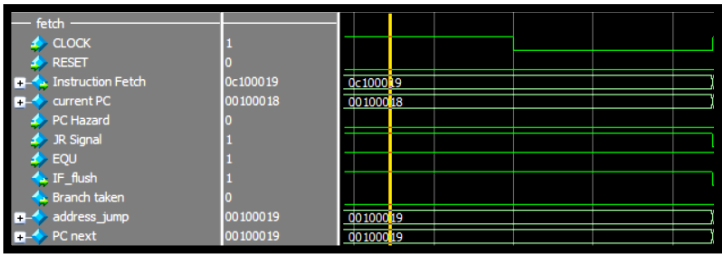

DECODE
Instruction Decode 01c00008
Register File 00000000 00000...
PC Decode 00100018
Write Data 00000019
AluResult_EXE 00000019
AluResult_MEM 00000019
Data_MEM 00000015
Register Destination 0e
RegWriteEn_wb 1
Forward A 2
Forward B 0
Immediate Result 00000008
Rt 00
Rs 0e
Rd 00
Forward A mux Result 00000019
Forward B mux Result 00000000
JR_Signal 1
control unit signal 0002

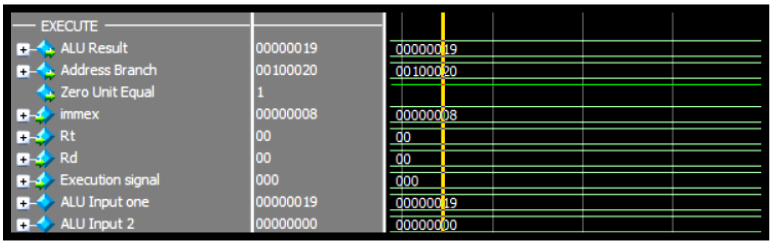

MEMORY
ALU Result Memory 00000019
write_data 00000000
Memory Signal 0
Memory Result 00000015

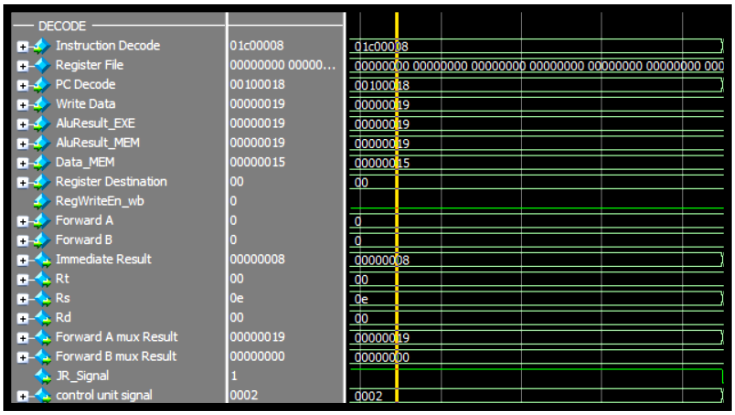

WB
MEM/WB Result 00000015
ALU Writeback result 00000019
Writeback Signal 0
WriteBack Data 00000019

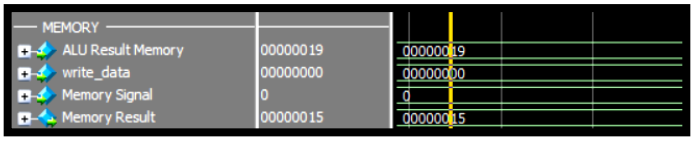

FORWARDING UNIT
IDEXE_RegWriteEn 0
EXEMEM_RegWrite 1
EXEMEM_MemReadEn 0
IFID_Rs 0e
IFID_Rt 00
IDEXE_Rd 00
EXEMEM_Rd 0e
Forward A 2
Forward B 0

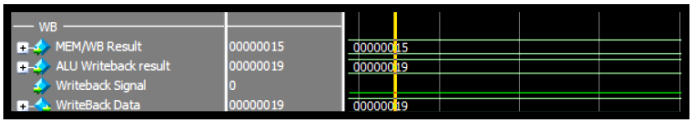
## Cycle 29

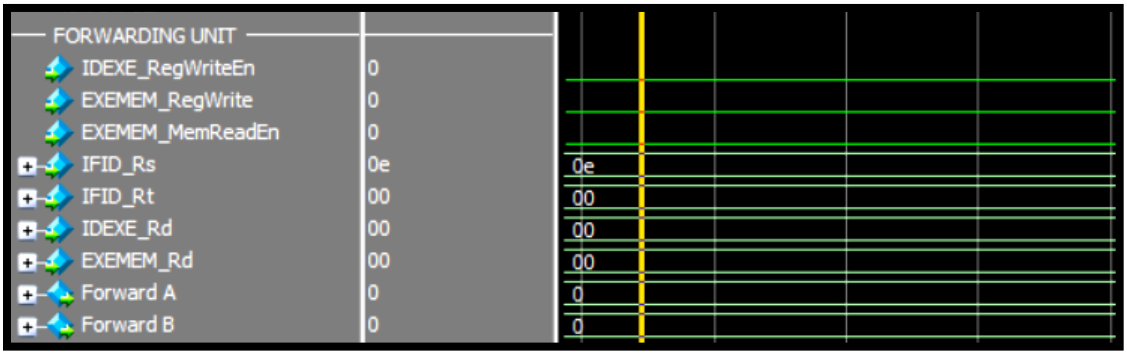

fetch
CLOCK 1
RESET 0
Instruction Fetch 0c100019
current PC 00100018
PC Hazard 0
JR Signal 1
EQU 1
IF_flush 1
Branch taken 0
address_jump 00100019
PC next 00100019


EXECUTE
ALU Result 00000019
Address Branch 00100020
Zero Unit Equal 1
immex 00000008
Rt 00
Rd 00
Execution signal 000
ALU Input one 00000019
ALU Input 2 00000000

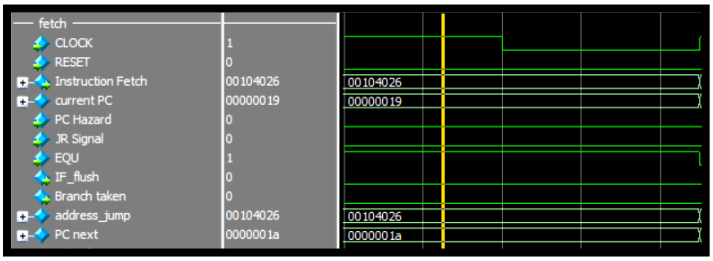

DECODE
Instruction Decode 01c00008
Register File 00000000 00000...
PC Decode 00100018
Write Data 00000019
AluResult_EXE 00000019
AluResult_MEM 00000019
Data_MEM 00000015
Register Destination 00
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00000008
Rt 00
Rs 0e
Rd 00
Forward A mux Result 00000019
Forward B mux Result 00000000
JR_Signal 1
control unit signal 0002

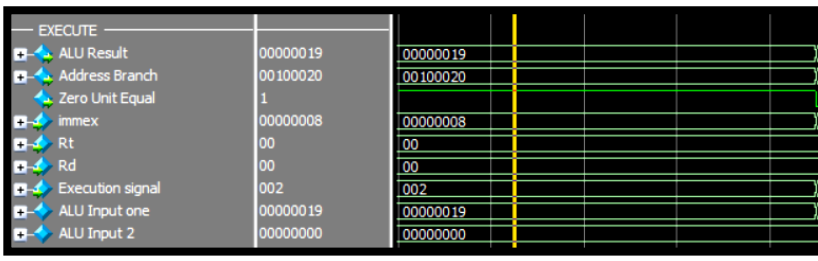

MEMORY
ALU Result Memory 00000019
write_data 00000000
Memory Signal 0
Memory Result 00000015

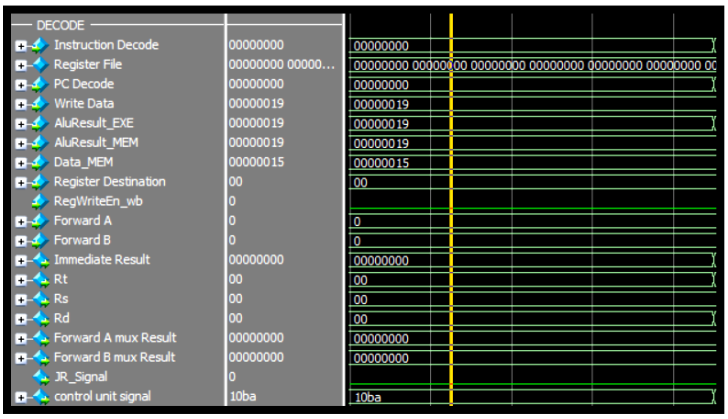

WB
MEM/WB Result 00000015
ALU Writeback result 00000019
Writeback Signal 0
WriteBack Data 00000019

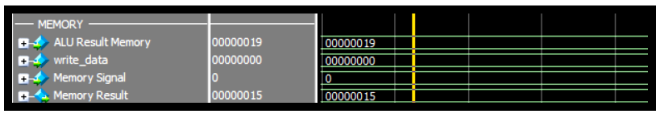

FORWARDING UNIT
IDEXE_RegWriteEn 0
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 0e
IFID_Rt 00
IDEXE_Rd 00
EXEMEM_Rd 00
Forward A 0
Forward B 0

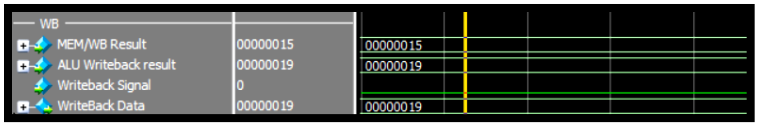
## Cycle 30

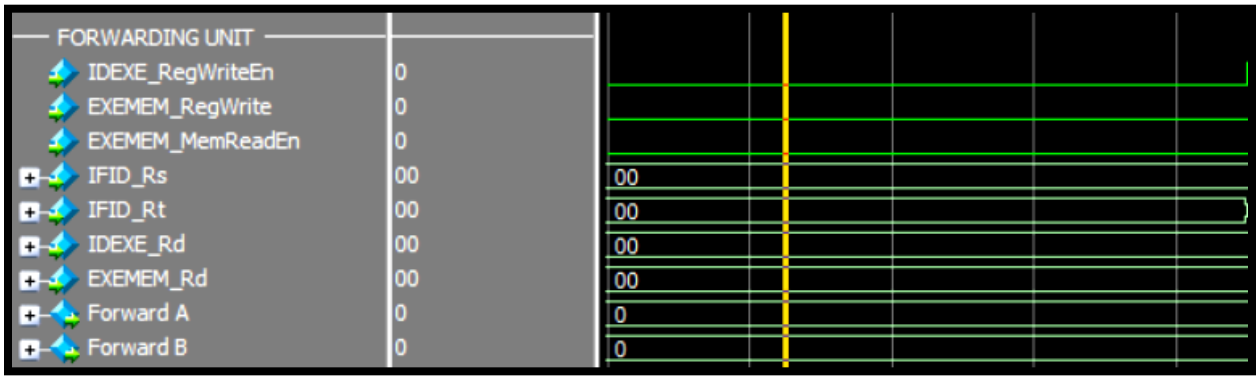

fetch
CLOCK 1
RESET 0
Instruction Fetch 00104026
current PC 00000019
PC Hazard 0
JR Signal 0
EQU 1
IF_flush 0
Branch taken 0
address_jump 00104026
PC next 0000001a


EXECUTE
ALU Result 00000019
Address Branch 00100020
Zero Unit Equal 1
immex 00000008
Rt 00
Rd 00
Execution signal 002
ALU Input one 00000019
ALU Input 2 00000000

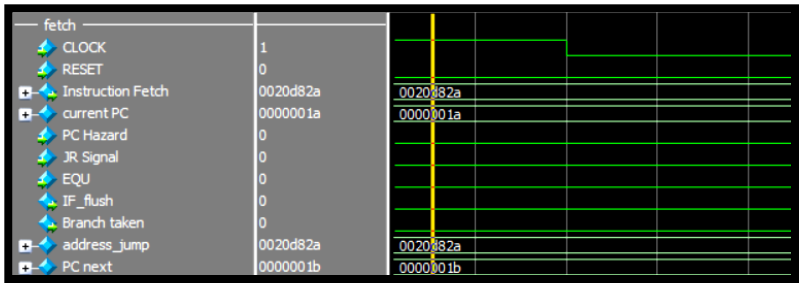

DECODE
Instruction Decode 00000000
Register File 00000000 00000...
PC Decode 00000000
Write Data 00000019
AluResult_EXE 00000019
AluResult_MEM 00000019
Data_MEM 00000015
Register Destination 00
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00000000
Rt 00
Rs 00
Rd 00
Forward A mux Result 00000000
Forward B mux Result 00000000
JR_Signal 0
control unit signal 10ba

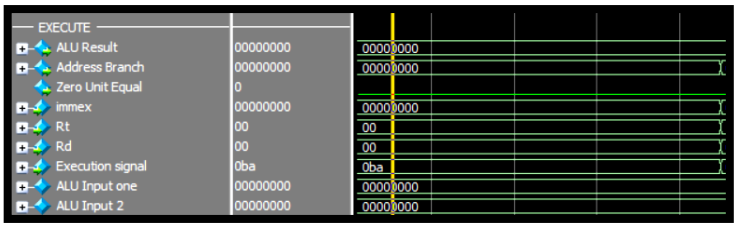

MEMORY
ALU Result Memory 00000019
write_data 00000000
Memory Signal 0
Memory Result 00000015

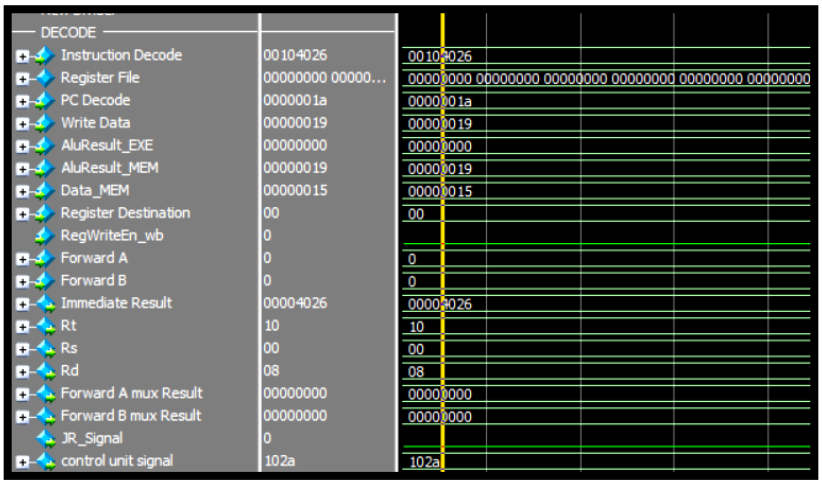

WB
MEM/WB Result 00000015
ALU Writeback result 00000019
Writeback Signal 0
WriteBack Data 00000019

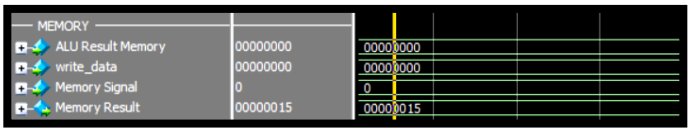

FORWARDING UNIT
IDEXE_RegWriteEn 0
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 00
IFID_Rt 00
IDEXE_Rd 00
EXEMEM_Rd 00
Forward A 0
Forward B 0

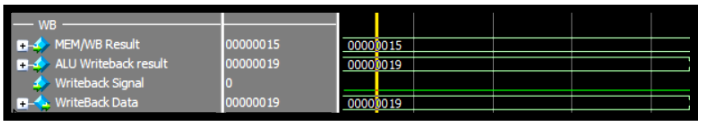
## Cycle 31

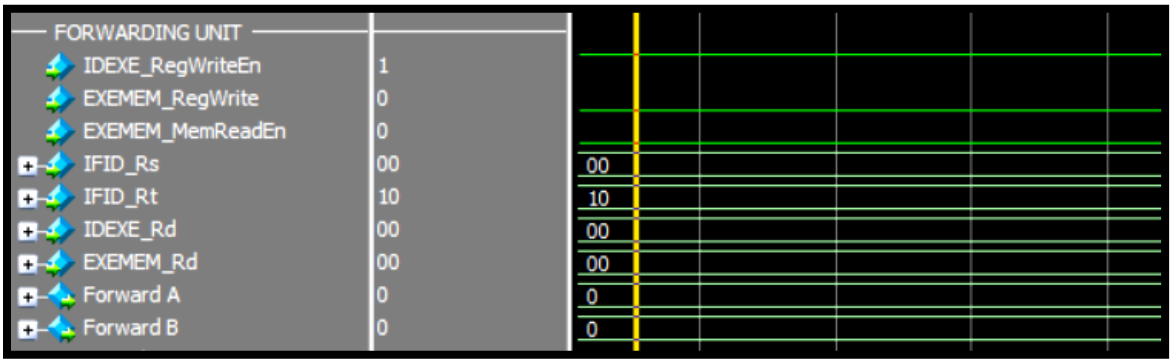

fetch
CLOCK 1
RESET 0
Instruction Fetch 0020d82a
current PC 0000001a
PC Hazard 0
JR Signal 0
EQU 0
IF_flush 0
Branch taken 0
address_jump 0020d82a
PC next 0000001b


EXECUTE
ALU Result 00000000
Address Branch 00000000
Zero Unit Equal 0
immex 00000000
Rt 00
Rd 00
Execution signal 0ba
ALU Input one 00000000
ALU Input 2 00000000

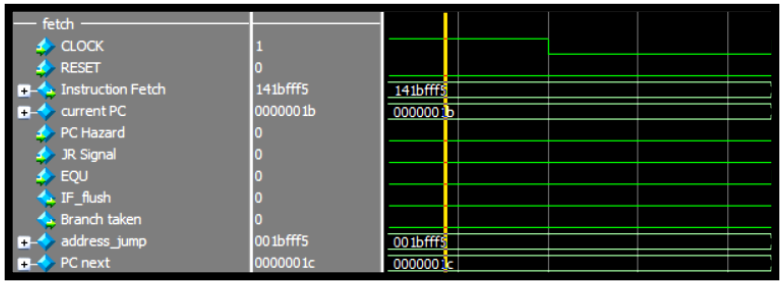

DECODE
Instruction Decode 00104026
Register File 00000000 00000...
PC Decode 0000001a
Write Data 00000019
AluResult_EXE 00000000
AluResult_MEM 00000019
Data_MEM 00000015
Register Destination 00
RegWriteEn_wb 0
Forward A 0
Forward B 0
Immediate Result 00004026
Rt 10
Rs 00
Rd 08
Forward A mux Result 00000000
Forward B mux Result 00000000
JR_Signal 0
control unit signal 102a

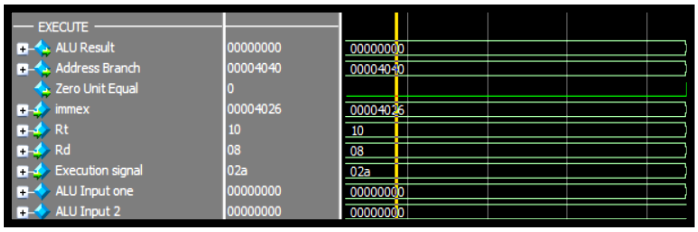

MEMORY
ALU Result Memory 00000000
write_data 00000000
Memory Signal 0
Memory Result 00000015

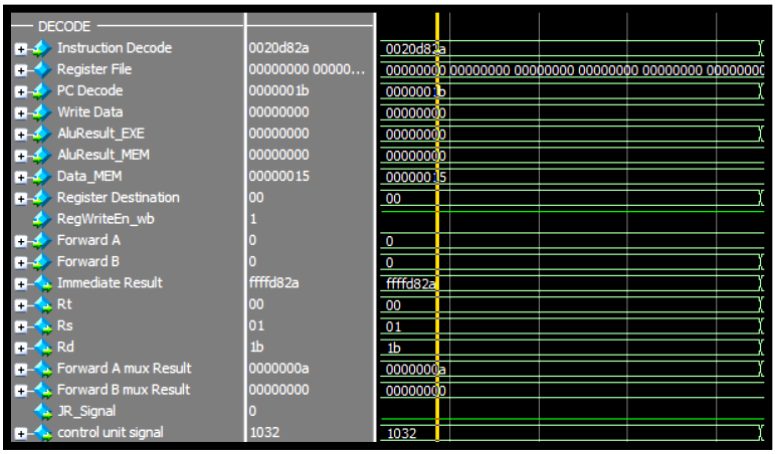

WB
MEM/WB Result 00000015
ALU Writeback result 00000019
Writeback Signal 0
WriteBack Data 00000019

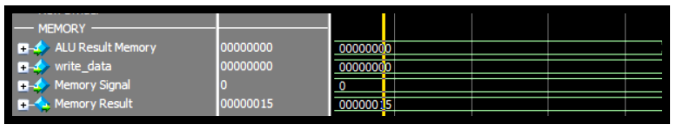

FORWARDING UNIT
IDEXE_RegWriteEn 1
EXEMEM_RegWrite 0
EXEMEM_MemReadEn 0
IFID_Rs 00
IFID_Rt 10
IDEXE_Rd 00
EXEMEM_Rd 00
Forward A 0
Forward B 0

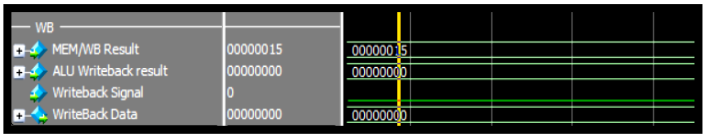
## Cycle 32

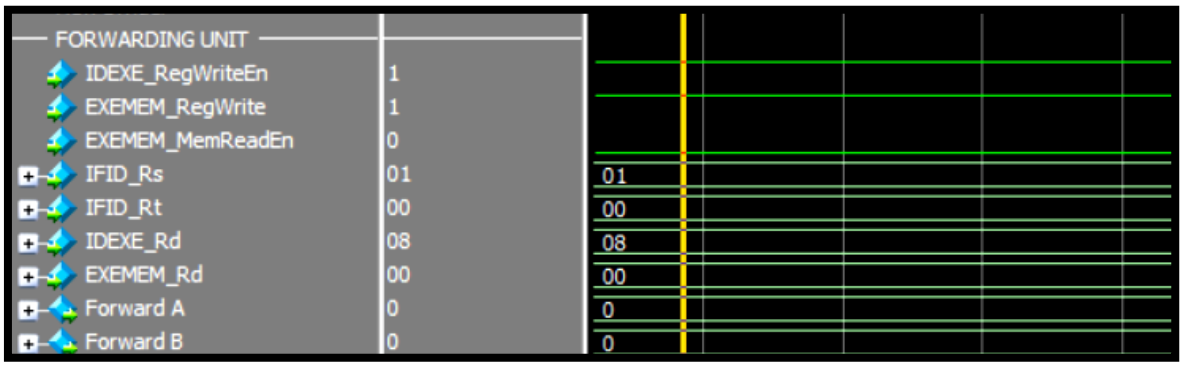


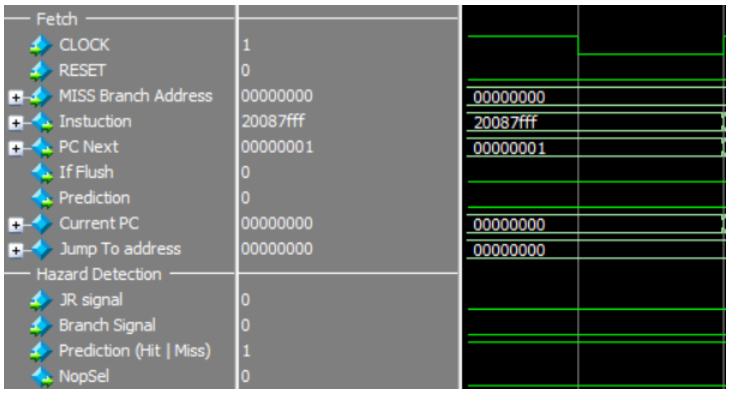



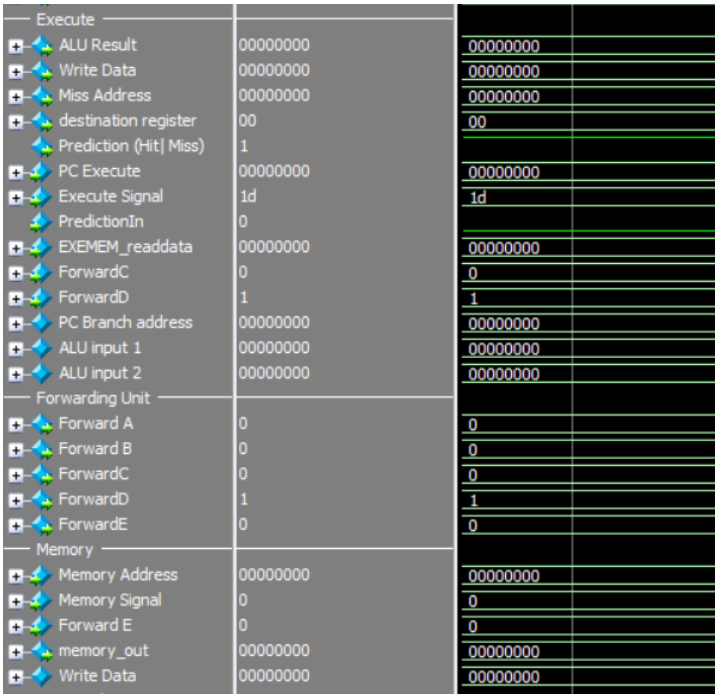

**Pipeline Testing2:**

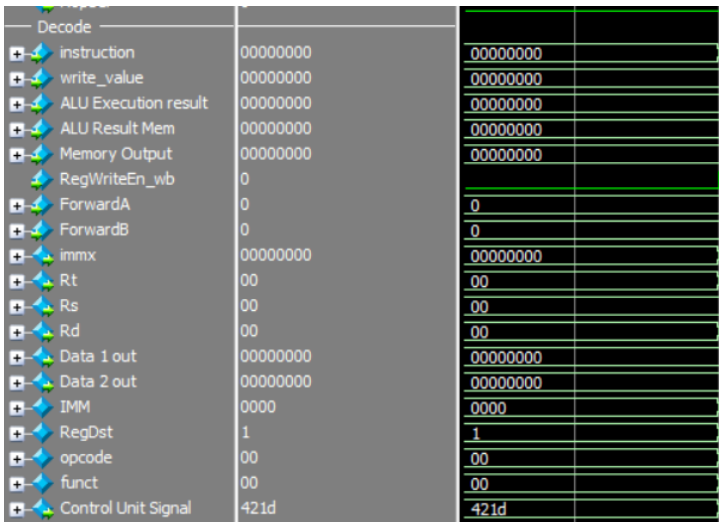
## Benchmark 2

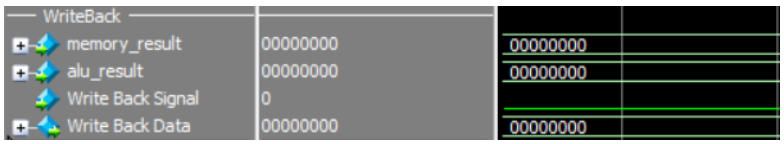
```
.text
addi $t0,$0, 0x7FFF
andi $t1,$0, 0x8000
add $t2, $t0, $t1
ori $t0, $0, 0x0
xori $t1,$0, 0xA
slti $t3,$t1,0xB
sll $t4,$t1,1
srl $t5,$t1,1
sgt $t6,$t1,$t0
bne_label:
sub $s0,$t1,$t0
and $s1,$t1,$t1
or $s2,$t1,$t4
xor $s3,$t1,$t4
nor $s4,$t1,$t4
slt $s5,$t1,$t4

nor $s4,$t1,$t4
slt $s5,$t1,$t4

j l
slt $s5,$t1,$t4
l:
addi $s7,$0,0x15
jr $s7
jal L1
L1:
beq $t1,$t0,beq_label
beq_label: bne $t1,$t0,bne_label
```

## CYCLE 1

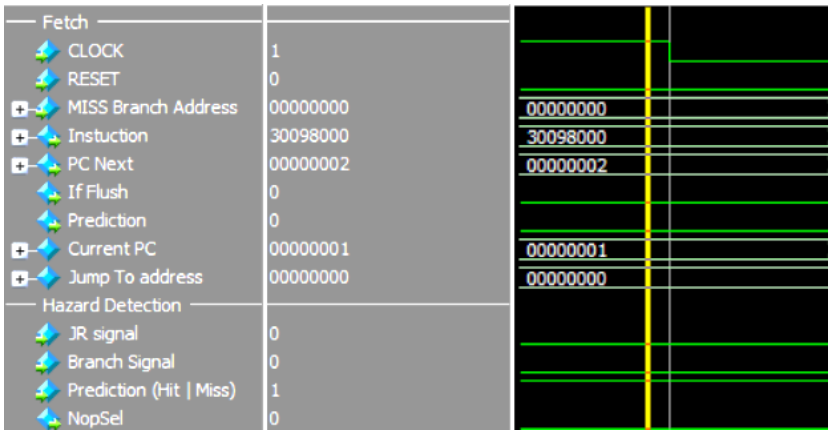

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00000000
Instuction 20087fff
PC Next 00000001
If Flush 0
Prediction 0
Current PC 00000000
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 1
NopSel 0

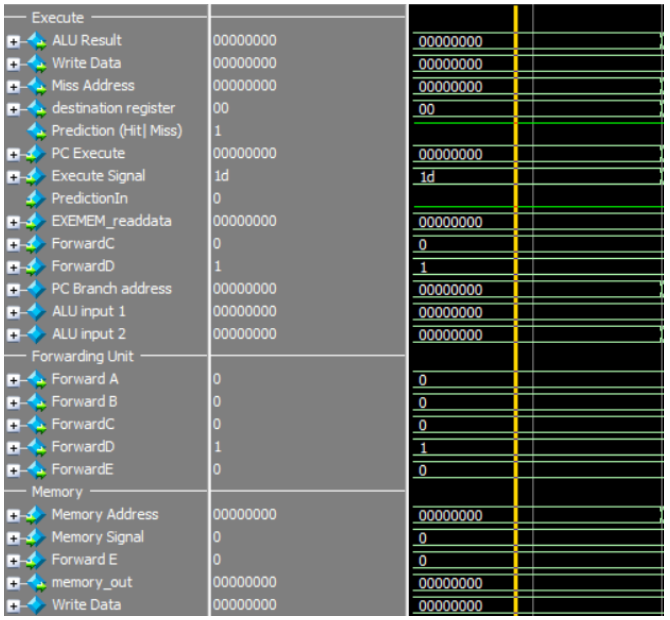

Execute
ALU Result 00000000
Write Data 00000000
Miss Address 00000000
destination register 00
Prediction (Hit| Miss) 1
PC Execute 00000000
Execute Signal 1d
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address 00000000
ALU input 1 00000000
ALU input 2 00000000
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 0
Memory
Memory Address 00000000
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

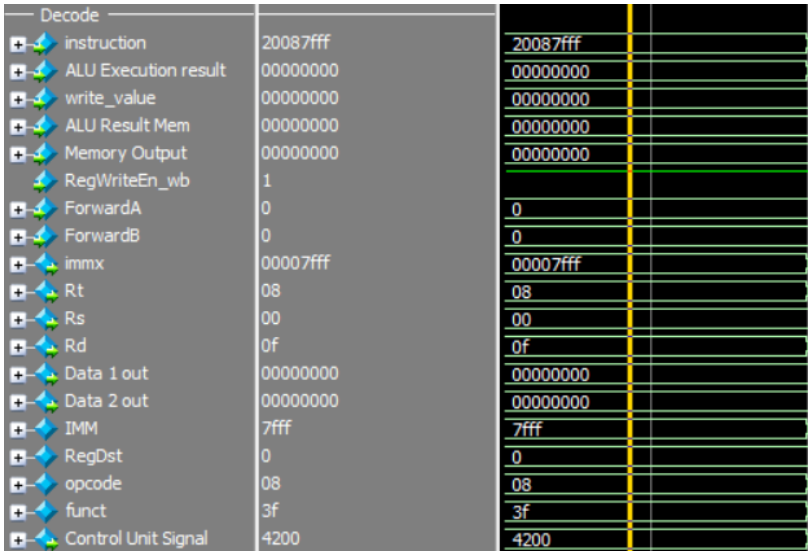

Decode
instruction 00000000
write_value 00000000
ALU Execution result 00000000
ALU Result Mem 00000000
Memory Output 00000000
RegWriteEn_wb 0
ForwardA 0
ForwardB 0
immx 00000000
Rt 00
Rs 00
Rd 00
Data 1 out 00000000
Data 2 out 00000000
IMM 0000
RegDst 1
opcode 00
funct 00
Control Unit Signal 421d

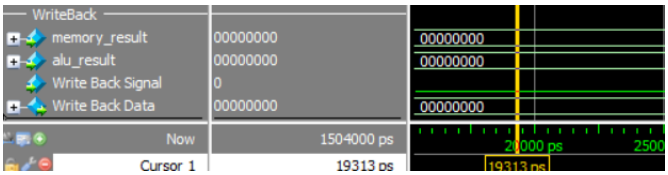

WriteBack
memory_result 00000000
alu_result 00000000
Write Back Signal 0
Write Back Data 00000000

## Cycle 2

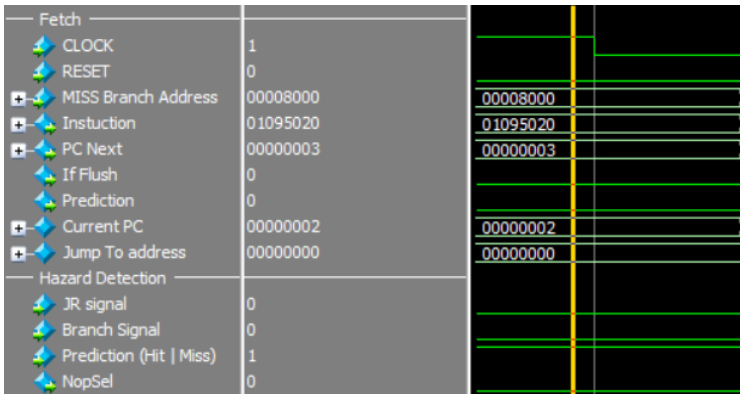

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00000000
Instuction 30098000
PC Next 00000002
If Flush 0
Prediction 0
Current PC 00000001
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 1
NopSel 0

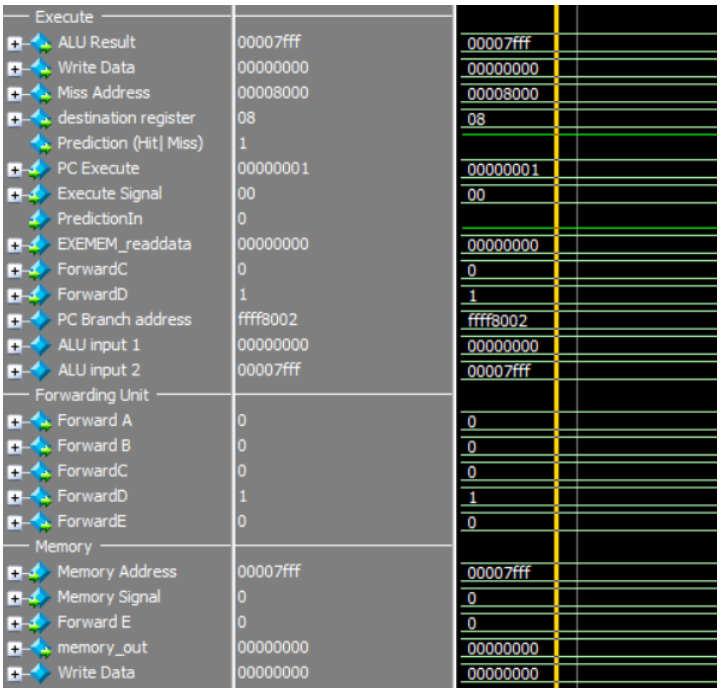

Execute
ALU Result 00000000
Write Data 00000000
Miss Address 00000000
destination register 00
Prediction (Hit| Miss) 1
PC Execute 00000000
Execute Signal 1d
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address 00000000
ALU input 1 00000000
ALU input 2 00000000
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 0
Memory
Memory Address 00000000
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

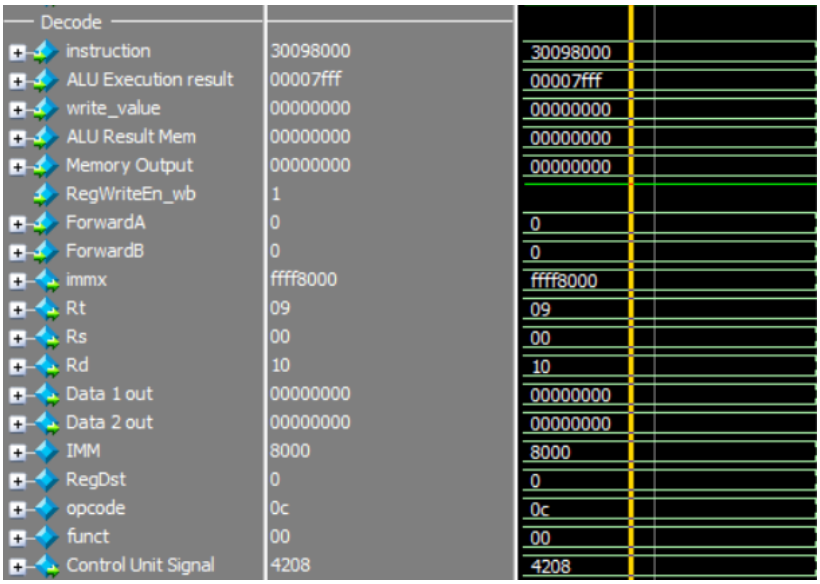

Decode
instruction 20087fff
ALU Execution result 00000000
write_value 00000000
ALU Result Mem 00000000
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx 00007fff
Rt 08
Rs 00
Rd 0f
Data 1 out 00000000
Data 2 out 00000000
IMM 7fff
RegDst 0
opcode 08
funct 3f
Control Unit Signal 4200

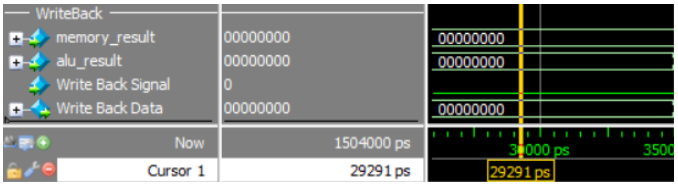

WriteBack
memory_result 00000000
alu_result 00000000
Write Back Signal 0
Write Back Data 00000000
Now 1504000 ps
Cursor 1 19313 ps

## Cycle 3

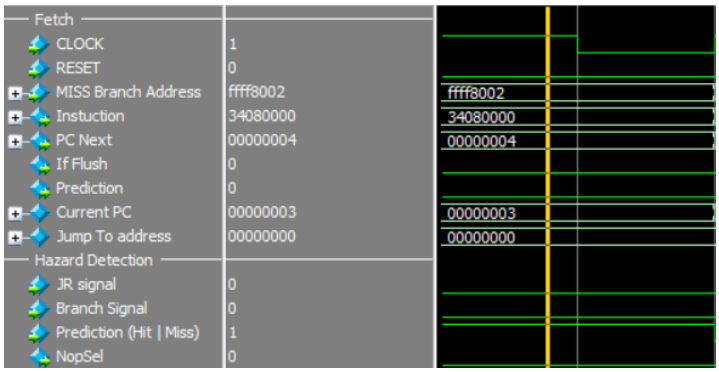

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00008000
Instuction 01095020
PC Next 00000003
If Flush 0
Prediction 0
Current PC 00000002
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 1
NopSel 0

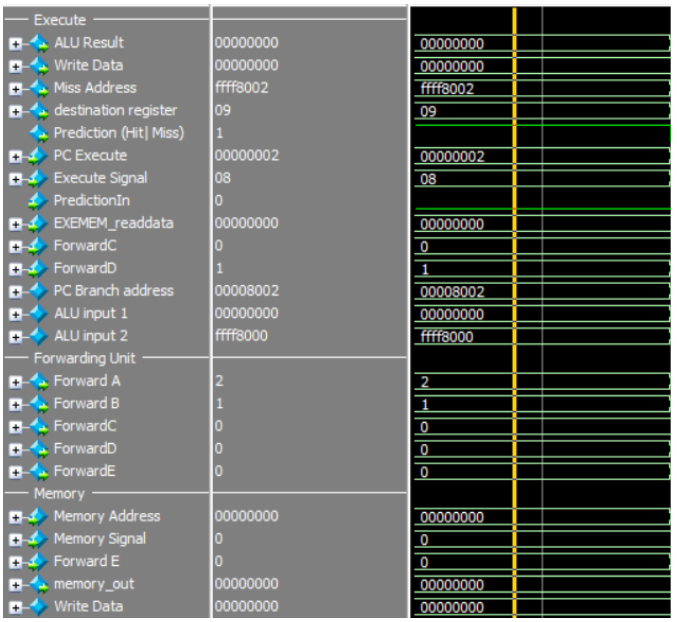

Execute
ALU Result 00007fff
Write Data 00000000
Miss Address 00008000
destination register 08
Prediction (Hit| Miss) 1
PC Execute 00000001
Execute Signal 00
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address ffff8002
ALU input 1 00000000
ALU input 2 00007fff
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 0
Memory
Memory Address 00007fff
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

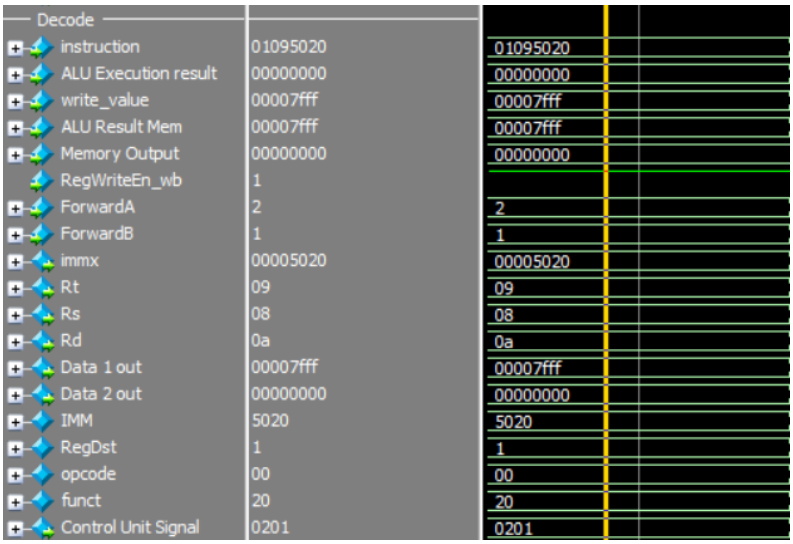

Decode
instruction 30098000
ALU Execution result 00007fff
write_value 00000000
ALU Result Mem 00000000
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffff8000
Rt 09
Rs 00
Rd 10
Data 1 out 00000000
Data 2 out 00000000
IMM 8000
RegDst 0
opcode 0c
funct 00
Control Unit Signal 4208


WriteBack
memory_result 00000000
alu_result 00000000
Write Back Signal 0
Write Back Data 00000000
Now 1504000 ps
Cursor 1 29291 ps
29291 ps

## Cycle 4

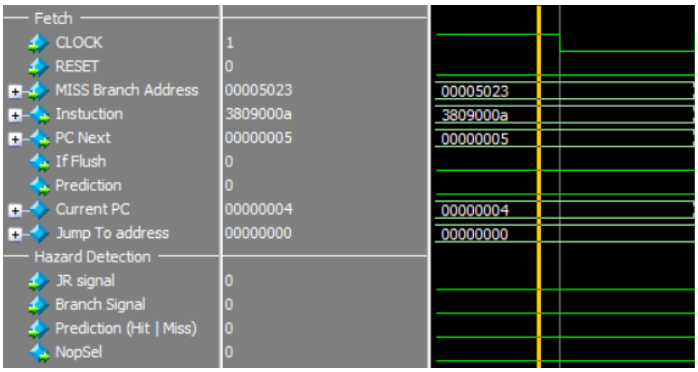

Fetch
CLOCK 1
RESET 0
MISS Branch Address ffff8002
Instuction 34080000
PC Next 00000004
If Flush 0
Prediction 0
Current PC 00000003
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 1
NopSel 0

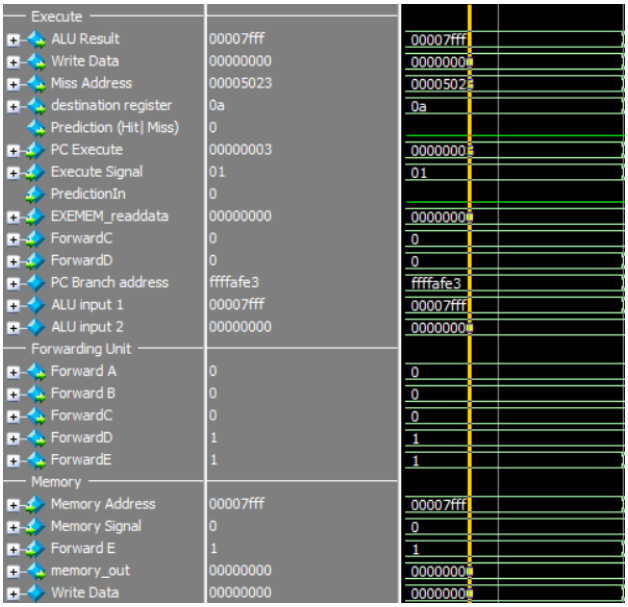

Execute
ALU Result 00000000
Write Data 00000000
Miss Address ffff8002
destination register 09
Prediction (Hit| Miss) 1
PC Execute 00000002
Execute Signal 08
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address 00008002
ALU input 1 00000000
ALU input 2 ffff8000
Forwarding Unit
Forward A 2
Forward B 1
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 00000000
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

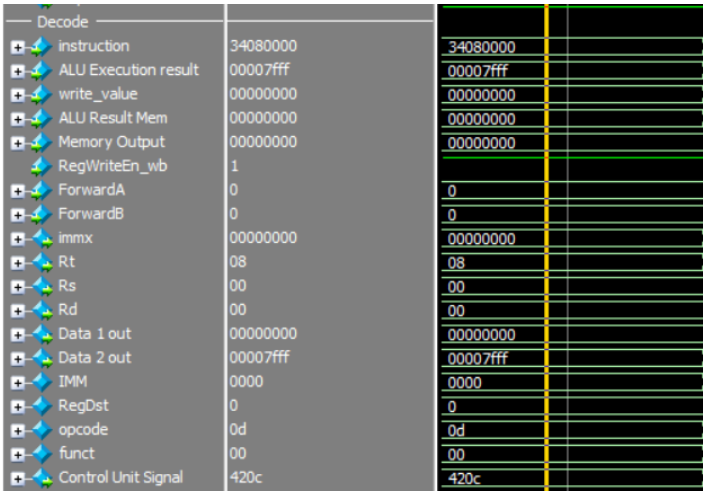

Decode
instruction 01095020
ALU Execution result 00000000
write_value 00007fff
ALU Result Mem 00007fff
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 2
ForwardB 1
immx 00005020
Rt 09
Rs 08
Rd 0a
Data 1 out 00007fff
Data 2 out 00000000
IMM 5020
RegDst 1
opcode 00
funct 20
Control Unit Signal 0201

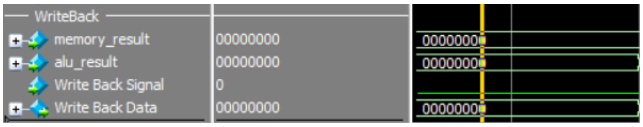

WriteBack
memory_result 00000000
alu_result 00007fff
Write Back Signal 0
Write Back Data 00007fff

## Cycle 5

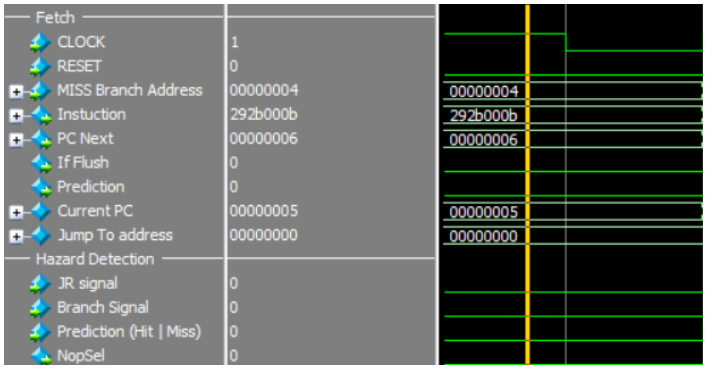

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00005023
Instuction 3809000a
PC Next 00000005
If Flush 0
Prediction 0
Current PC 00000004
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

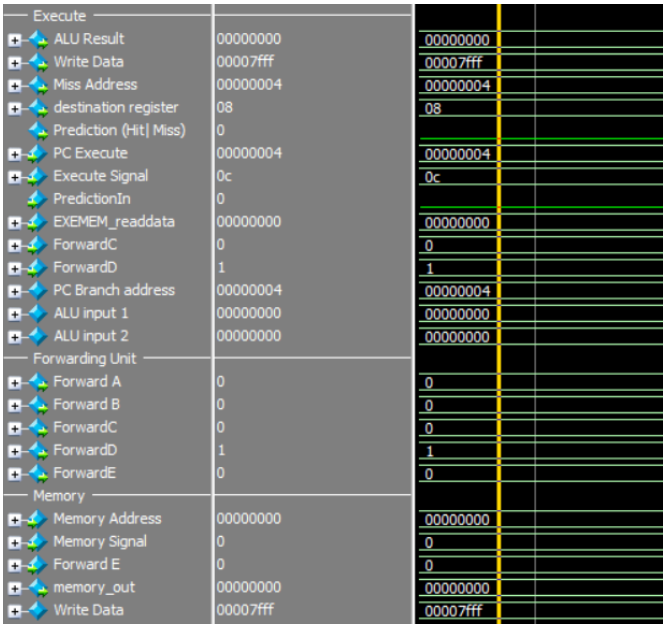

Execute
ALU Result 00007fff
Write Data 00000000
Miss Address 00005023
destination register 0a
Prediction (Hit| Miss) 0
PC Execute 00000003
Execute Signal 01
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address ffffafe3
ALU input 1 00007fff
ALU input 2 00000000
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 1
Memory
Memory Address 00007fff
Memory Signal 0
Forward E 1
memory_out 00000000
Write Data 00000000

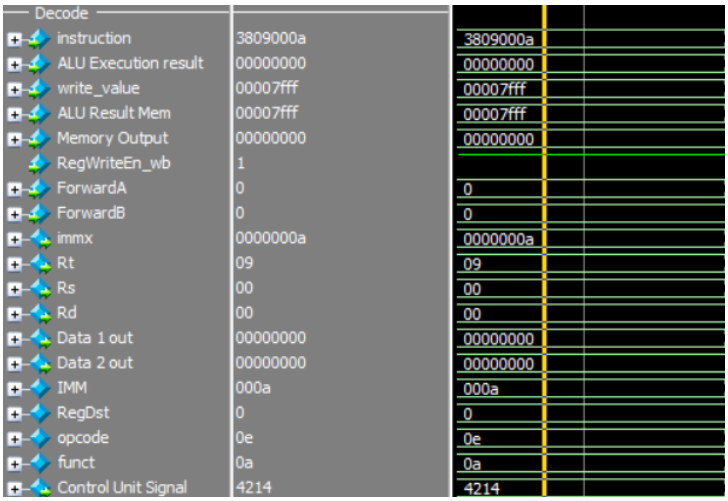

Decode
instruction 34080000
ALU Execution result 00007fff
write_value 00000000
ALU Result Mem 00000000
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx 00000000
Rt 08
Rs 00
Rd 00
Data 1 out 00000000
Data 2 out 00007fff
IMM 0000
RegDst 0
opcode 0d
funct 00
Control Unit Signal 420c


WriteBack
memory_result 00000000
alu_result 00000000
Write Back Signal 0
Write Back Data 00000000

# Cycle 6

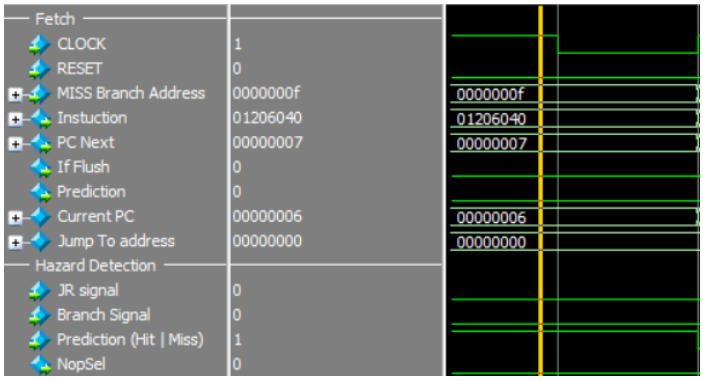

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00000004
Instuction 292b000b
PC Next 00000006
If Flush 0
Prediction 0
Current PC 00000005
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

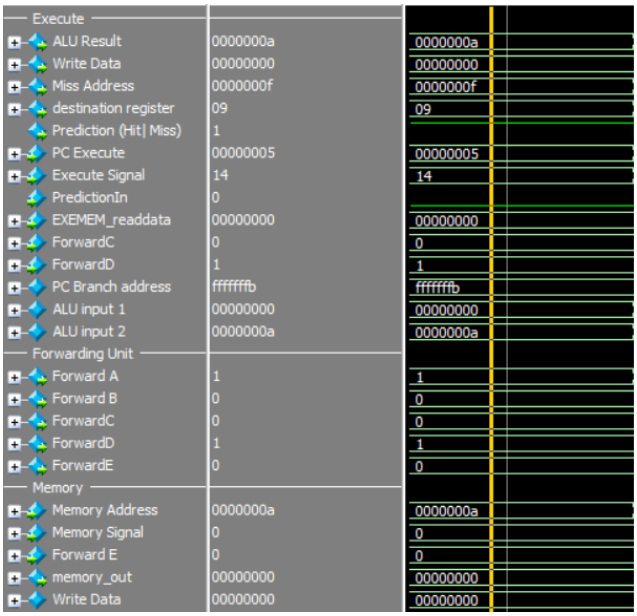

Execute
ALU Result 00000000
Write Data 00007fff
Miss Address 00000004
destination register 08
Prediction (Hit| Miss) 0
PC Execute 00000004
Execute Signal 0c
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address 00000004
ALU input 1 00000000
ALU input 2 00000000
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 0
Memory
Memory Address 00000000
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00007fff

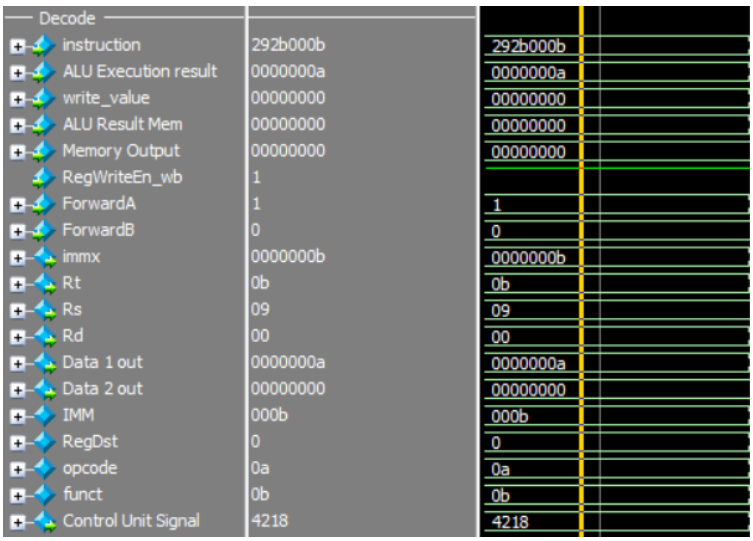

Decode
instruction 3809000a
ALU Execution result 00000000
write_value 00007fff
ALU Result Mem 00007fff
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx 0000000a
Rt 09
Rs 00
Rd 00
Data 1 out 00000000
Data 2 out 00000000
IMM 000a
RegDst 0
opcode 0e
funct 0a
Control Unit Signal 4214

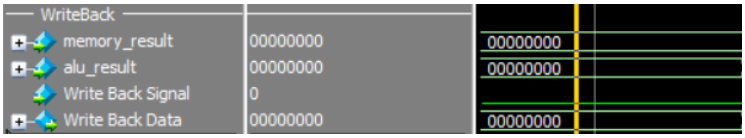

WriteBack
memory_result 00000000
alu_result 00007fff
Write Back Signal 0
Write Back Data 00007fff

# Cycle 7

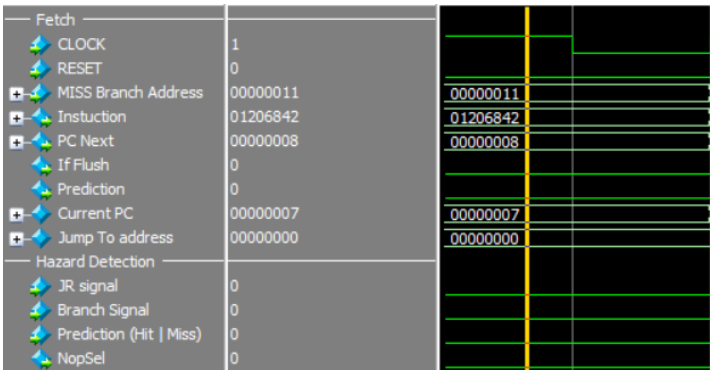

Fetch
CLOCK 1
RESET 0
MISS Branch Address 0000000f
Instuction 01206040
PC Next 00000007
If Flush 0
Prediction 0
Current PC 00000006
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 1
NopSel 0

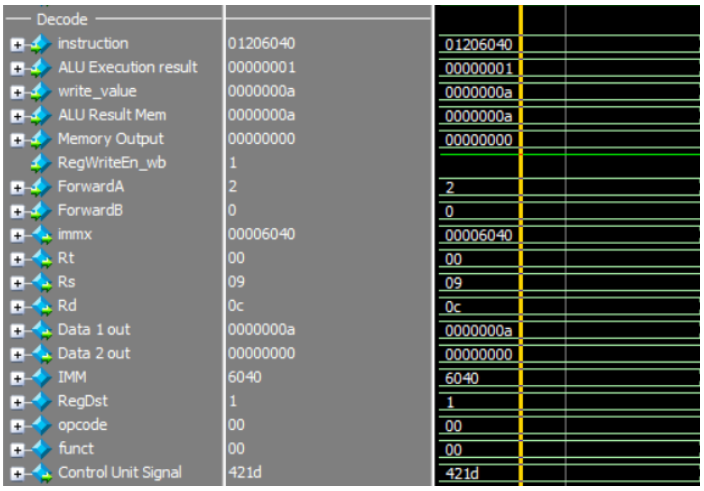

Execute
ALU Result 0000000a
Write Data 00000000
Miss Address 0000000f
destination register 09
Prediction (Hit| Miss) 1
PC Execute 00000005
Execute Signal 14
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address fffffffb
ALU input 1 00000000
ALU input 2 0000000a
Forwarding Unit
Forward A 1
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 0
Memory
Memory Address 0000000a
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

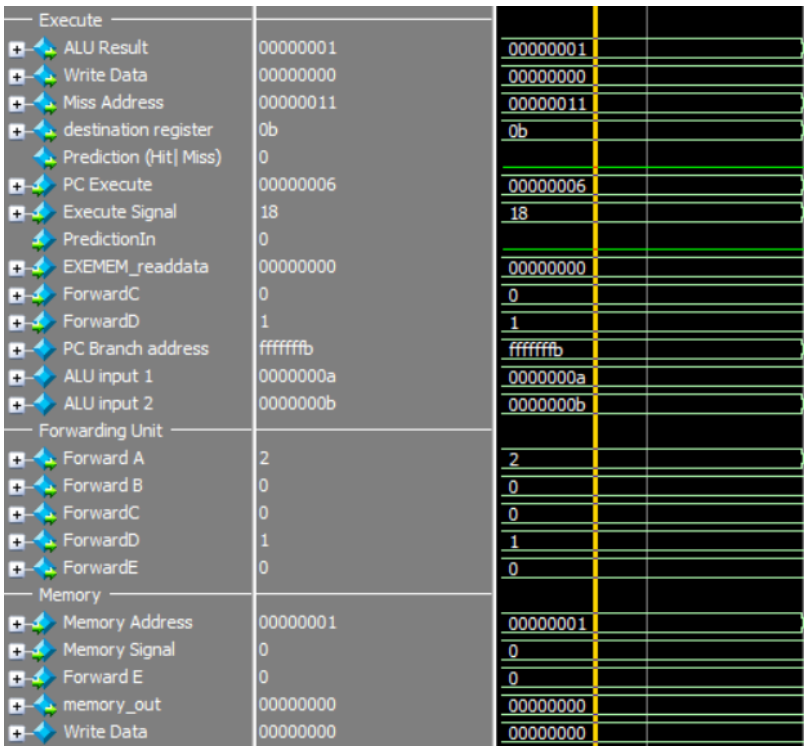

Decode
instruction 292b000b
ALU Execution result 0000000a
write_value 00000000
ALU Result Mem 00000000
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 1
ForwardB 0
immx 0000000b
Rt 0b
Rs 09
Rd 00
Data 1 out 0000000a
Data 2 out 00000000
IMM 000b
RegDst 0
opcode 0a
funct 0b
Control Unit Signal 4218


WriteBack
memory_result 00000000
alu_result 00000000
Write Back Signal 0
Write Back Data 00000000

# cycle 8

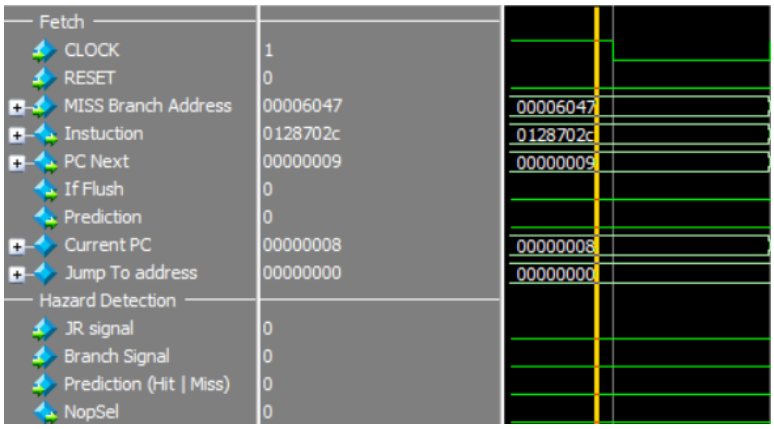

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00000011
Instuction 01206842
PC Next 00000008
If Flush 0
Prediction 0
Current PC 00000007
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

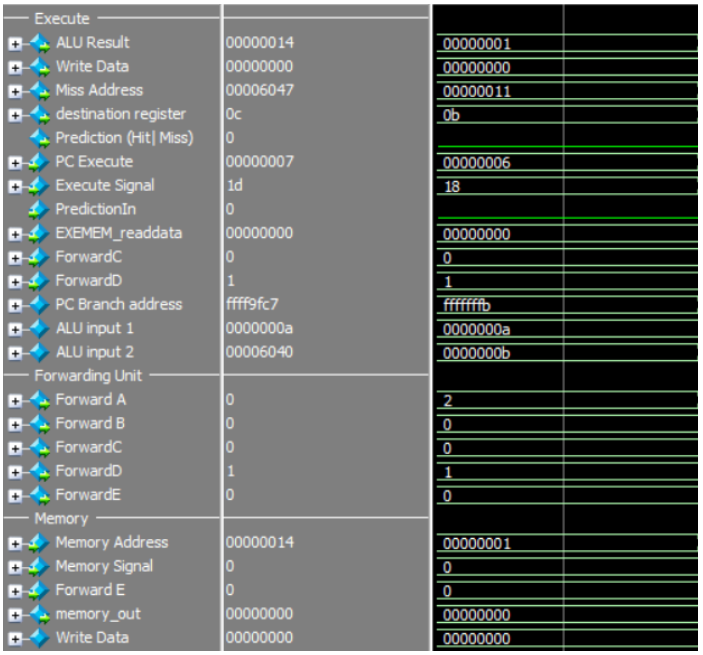

Decode
instruction 01206040
ALU Execution result 00000001
write_value 0000000a
ALU Result Mem 0000000a
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 2
ForwardB 0
immx 00006040
Rt 00
Rs 09
Rd 0c
Data 1 out 0000000a
Data 2 out 00000000
IMM 6040
RegDst 1
opcode 00
funct 00
Control Unit Signal 421d

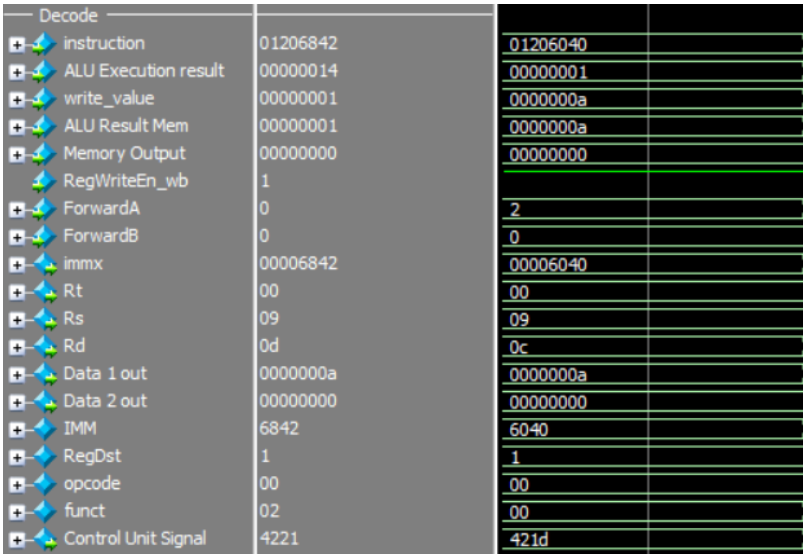

Execute
ALU Result 00000001
Write Data 00000000
Miss Address 00000011
destination register 0b
Prediction (Hit| Miss) 0
PC Execute 00000006
Execute Signal 18
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address fffffffb
ALU input 1 0000000a
ALU input 2 0000000b
Forwarding Unit
Forward A 2
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 0
Memory
Memory Address 00000001
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

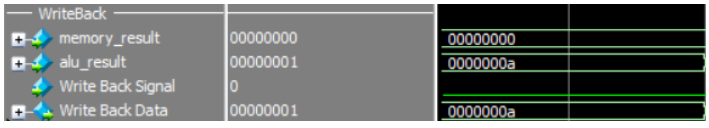

WriteBack
memory_result 00000000
alu_result 0000000a
Write Back Signal 0
Write Back Data 0000000a

# Cycle 9

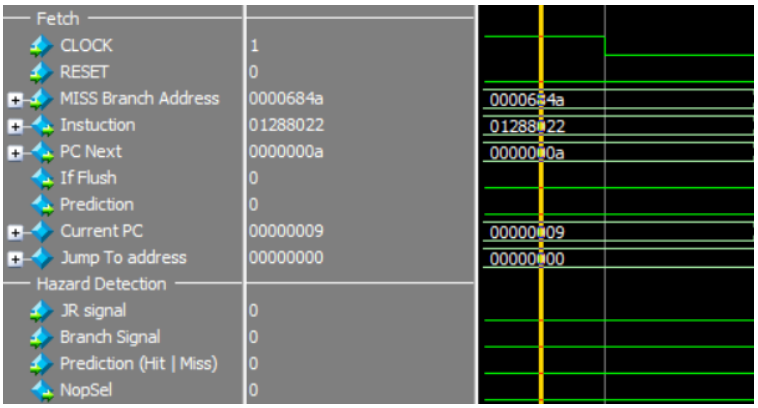

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00006047
Instuction 0128702c
PC Next 00000009
If Flush 0
Prediction 0
Current PC 00000008
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

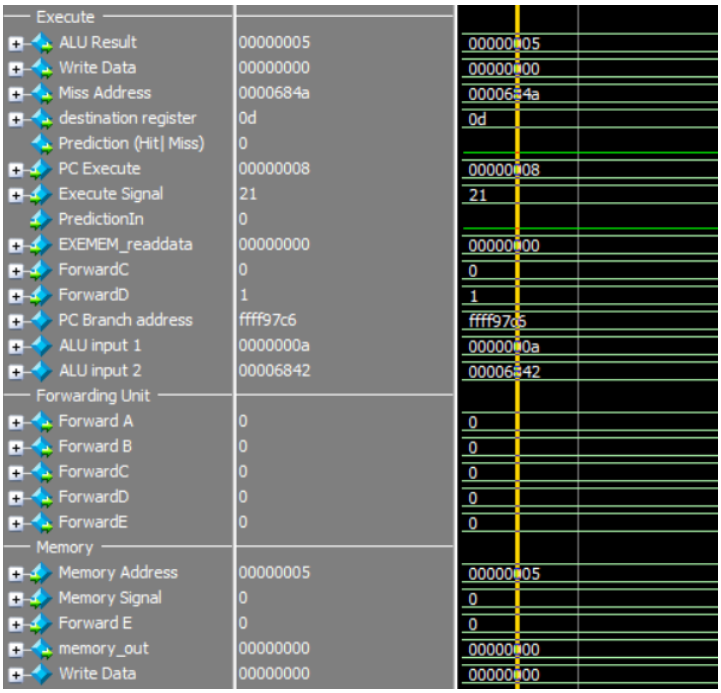

Execute
ALU Result 00000014
Write Data 00000000
Miss Address 00006047
destination register 0c
Prediction (Hit| Miss) 0
PC Execute 00000007
Execute Signal 1d
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address ffff9fc7
ALU input 1 0000000a
ALU input 2 00006040
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 0
Memory
Memory Address 00000014
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

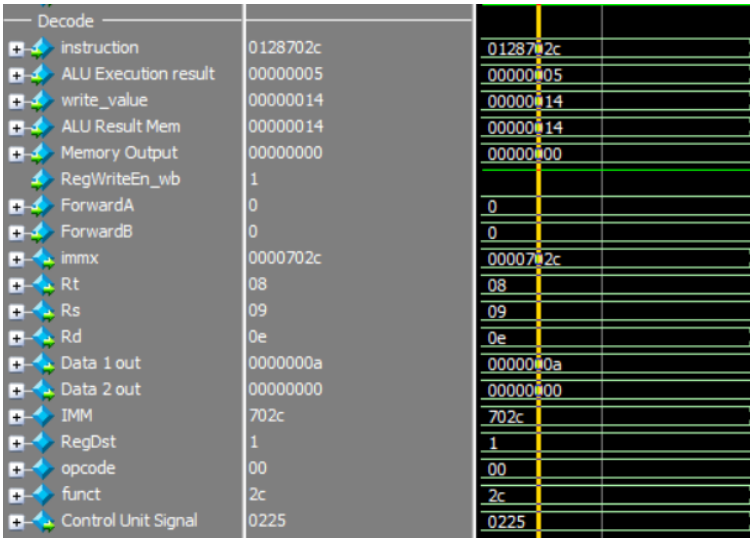

Decode
instruction 01206842
ALU Execution result 00000014
write_value 00000001
ALU Result Mem 00000001
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx 00006842
Rt 00
Rs 09
Rd 0d
Data 1 out 0000000a
Data 2 out 00000000
IMM 6842
RegDst 1
opcode 00
funct 02
Control Unit Signal 4221


WriteBack
memory_result 00000000
alu_result 00000001
Write Back Signal 0
Write Back Data 00000001

# Cycle 10

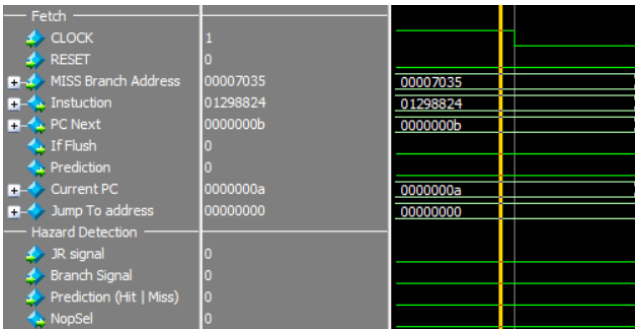

Fetch
CLOCK 1
RESET 0
MISS Branch Address 0000684a
Instuction 01288022
PC Next 0000000a
If Flush 0
Prediction 0
Current PC 00000009
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

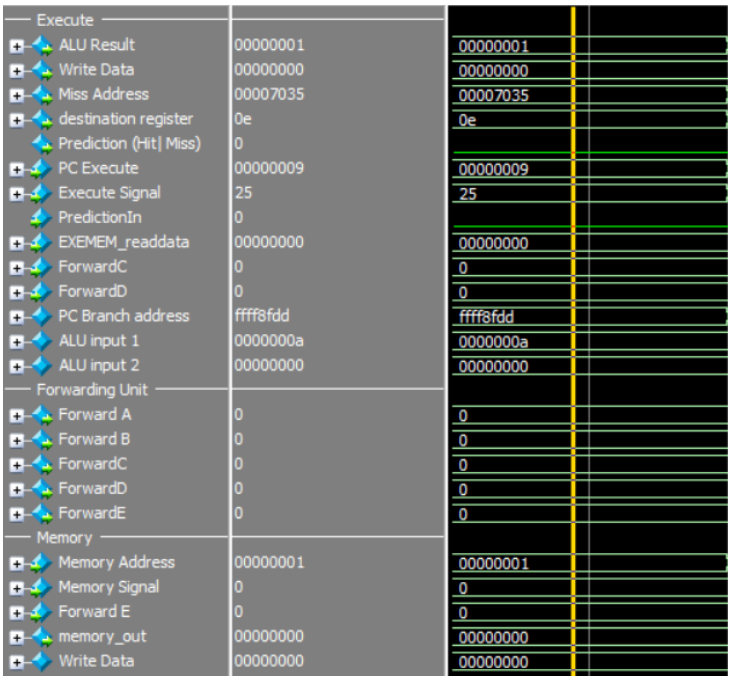

Execute
ALU Result 00000005
Write Data 00000000
Miss Address 0000684a
destination register 0d
Prediction (Hit| Miss) 0
PC Execute 00000008
Execute Signal 21
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address ffff97c6
ALU input 1 0000000a
ALU input 2 00006842
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 00000005
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

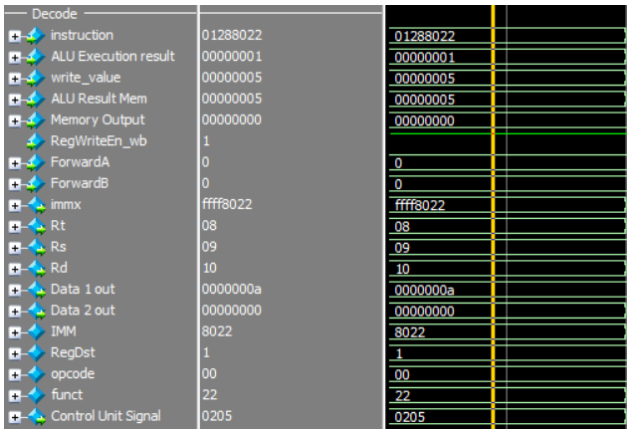

Decode
instruction 0128702c
ALU Execution result 00000005
write_value 00000014
ALU Result Mem 00000014
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx 0000702c
Rt 08
Rs 09
Rd 0e
Data 1 out 0000000a
Data 2 out 00000000
IMM 702c
RegDst 1
opcode 00
funct 2c
Control Unit Signal 0225


WriteBack
memory_result 00000000
alu_result 00000014
Write Back Signal 0
Write Back Data 00000014

# Cycle 11

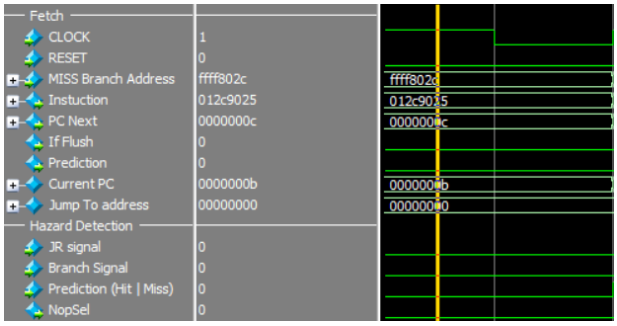

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00007035
Instuction 01298824
PC Next 0000000b
If Flush 0
Prediction 0
Current PC 0000000a
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

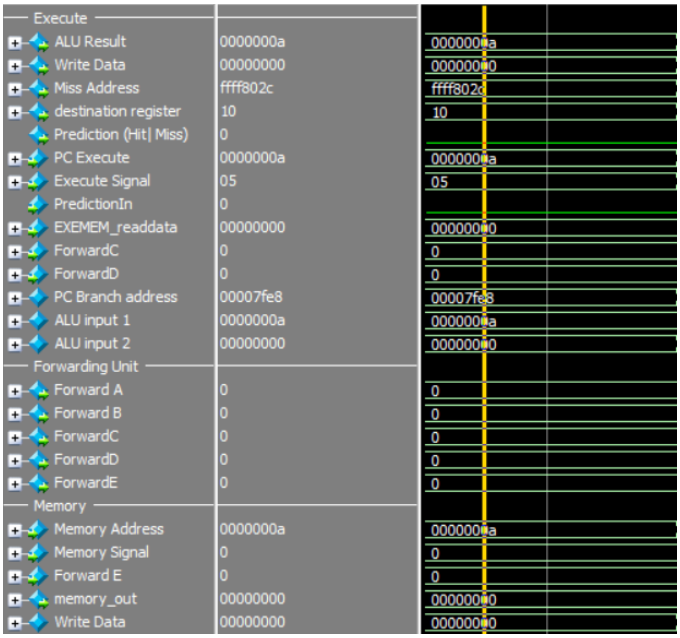

Execute
ALU Result 00000001
Write Data 00000000
Miss Address 00007035
destination register 0e
Prediction (Hit| Miss) 0
PC Execute 00000009
Execute Signal 25
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address ffff8fdd
ALU input 1 0000000a
ALU input 2 00000000
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 00000001
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

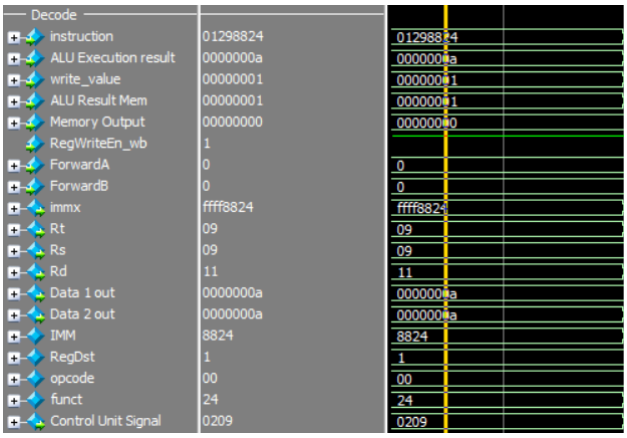

Decode
instruction 01288022
ALU Execution result 00000001
write_value 00000005
ALU Result Mem 00000005
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffff8022
Rt 08
Rs 09
Rd 10
Data 1 out 0000000a
Data 2 out 00000000
IMM 8022
RegDst 1
opcode 00
funct 22
Control Unit Signal 0205

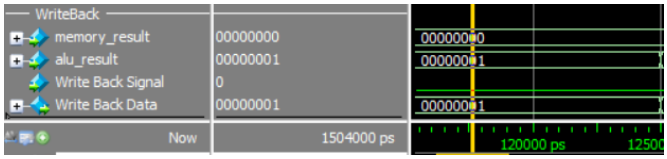

WriteBack
memory_result 00000000
alu_result 00000005
Write Back Signal 0
Write Back Data 00000005

# Cycle 12

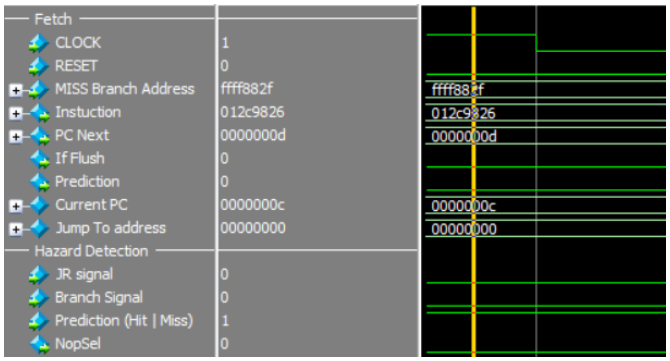

Fetch
CLOCK 1
RESET 0
MISS Branch Address ffff802c
Instuction 012c9025
PC Next 0000000c
If Flush 0
Prediction 0
Current PC 0000000b
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

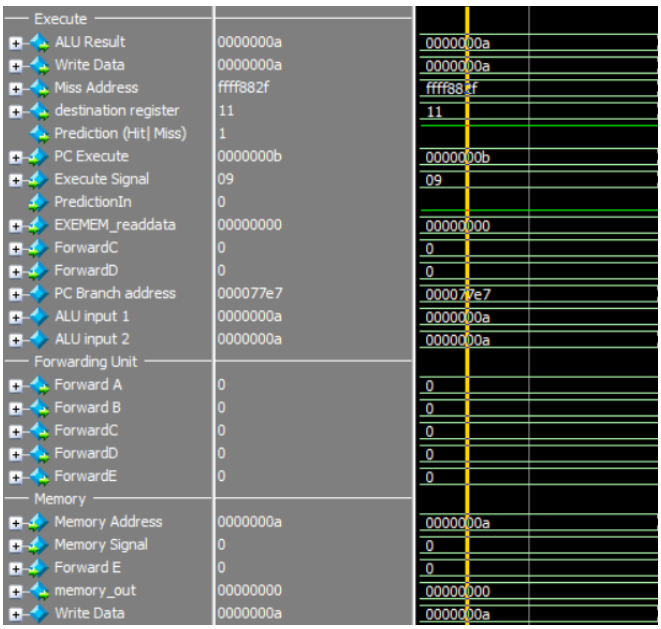

Execute
ALU Result 0000000a
Write Data 00000000
Miss Address ffff802c
destination register 10
Prediction (Hit| Miss) 0
PC Execute 0000000a
Execute Signal 05
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 00007fe8
ALU input 1 0000000a
ALU input 2 00000000
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 0000000a
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

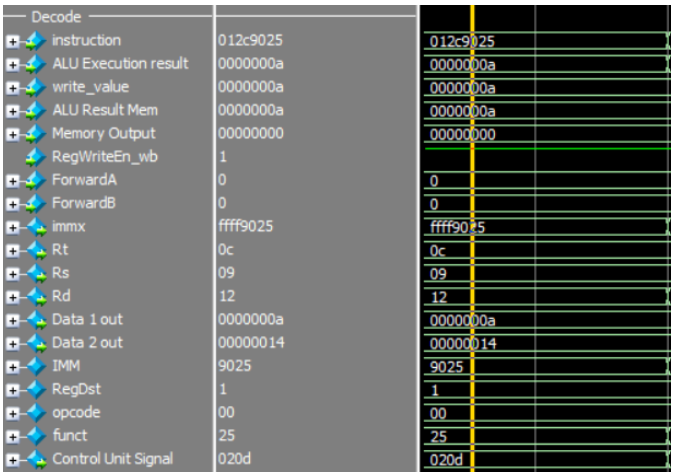

Decode
instruction 01298824
ALU Execution result 0000000a
write_value 00000001
ALU Result Mem 00000001
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffff8824
Rt 09
Rs 09
Rd 11
Data 1 out 0000000a
Data 2 out 0000000a
IMM 8824
RegDst 1
opcode 00
funct 24
Control Unit Signal 0209


WriteBack
memory_result 00000000
alu_result 00000001
Write Back Signal 0
Write Back Data 00000001
Now
1504000 ps
120000 ps

# Cycle 13

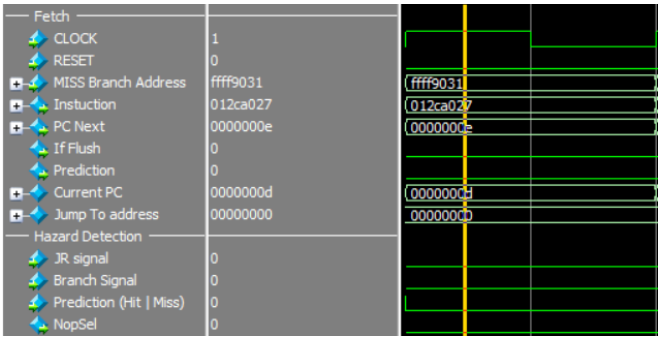

Fetch
CLOCK 1
RESET 0
MISS Branch Address ffff882f
Instuction 012c9826
PC Next 0000000d
If Flush 0
Prediction 0
Current PC 0000000c
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 1
NopSel 0

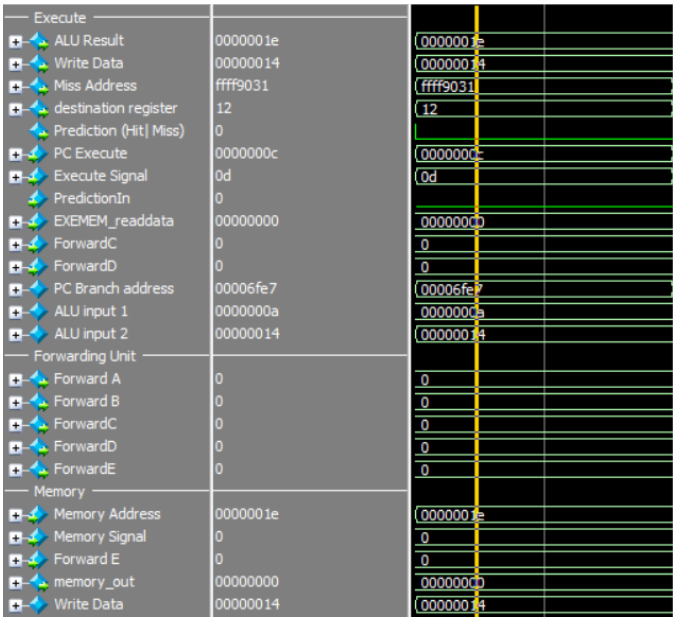

Execute
ALU Result 0000000a
Write Data 0000000a
Miss Address ffff882f
destination register 11
Prediction (Hit| Miss) 1
PC Execute 0000000b
Execute Signal 09
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 000077e7
ALU input 1 0000000a
ALU input 2 0000000a
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 0000000a
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 0000000a

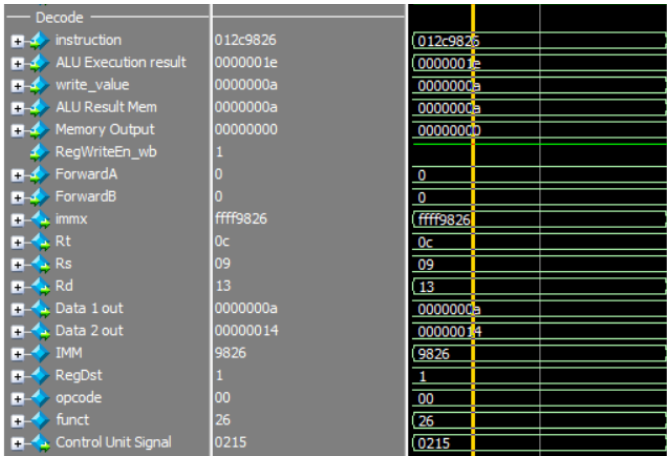

Decode
instruction 012c9025
ALU Execution result 0000000a
write_value 0000000a
ALU Result Mem 0000000a
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffff9025
Rt 0c
Rs 09
Rd 12
Data 1 out 0000000a
Data 2 out 00000014
IMM 9025
RegDst 1
opcode 00
funct 25
Control Unit Signal 020d


WriteBack
memory_result 00000000
alu_result 0000000a
Write Back Signal 0
Write Back Data 0000000a

# Cycle 14

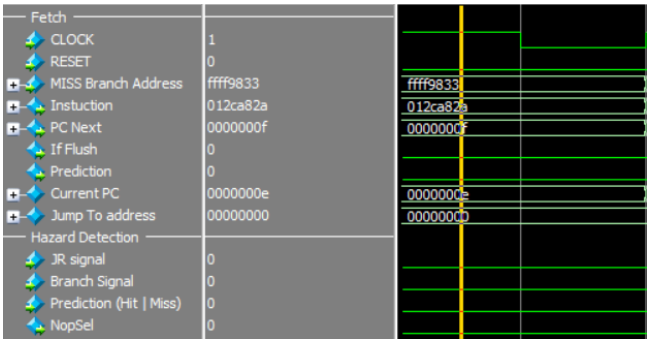

Fetch
CLOCK 1
RESET 0
MISS Branch Address ffff9031
Instuction 012ca027
PC Next 0000000e
If Flush 0
Prediction 0
Current PC 0000000d
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

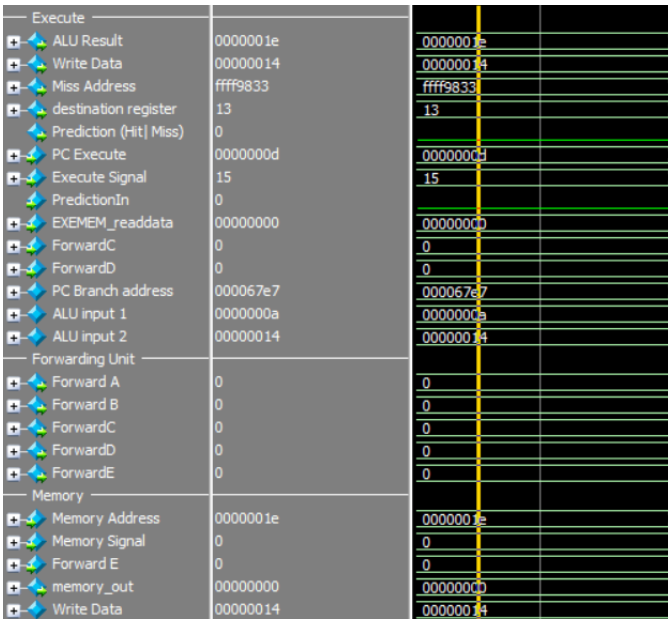

Execute
ALU Result 0000001e
Write Data 00000014
Miss Address ffff9031
destination register 12
Prediction (Hit| Miss) 0
PC Execute 0000000c
Execute Signal 0d
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 00006fe7
ALU input 1 0000000a
ALU input 2 00000014
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 0000001e
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000014

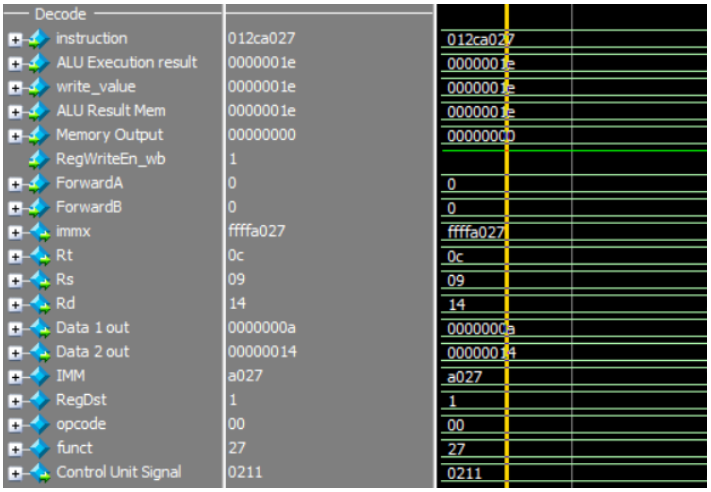

Decode
instruction 012c9826
ALU Execution result 0000001e
write_value 0000000a
ALU Result Mem 0000000a
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffff9826
Rt 0c
Rs 09
Rd 13
Data 1 out 0000000a
Data 2 out 00000014
IMM 9826
RegDst 1
opcode 00
funct 26
Control Unit Signal 0215


WriteBack
memory_result 00000000
alu_result 0000000a
Write Back Signal 0
Write Back Data 0000000a

# Cycle 15

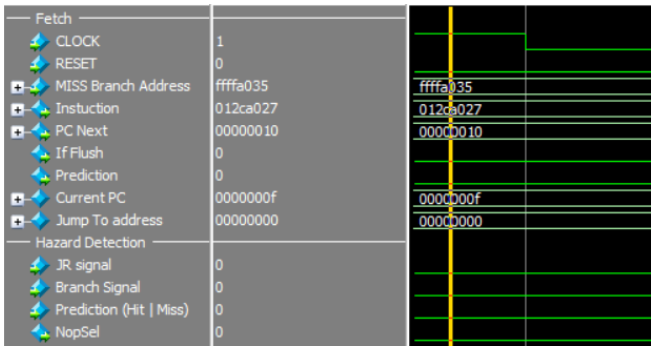

Fetch
CLOCK 1
RESET 0
MISS Branch Address ffff9833
Instuction 012ca82a
PC Next 0000000f
If Flush 0
Prediction 0
Current PC 0000000e
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

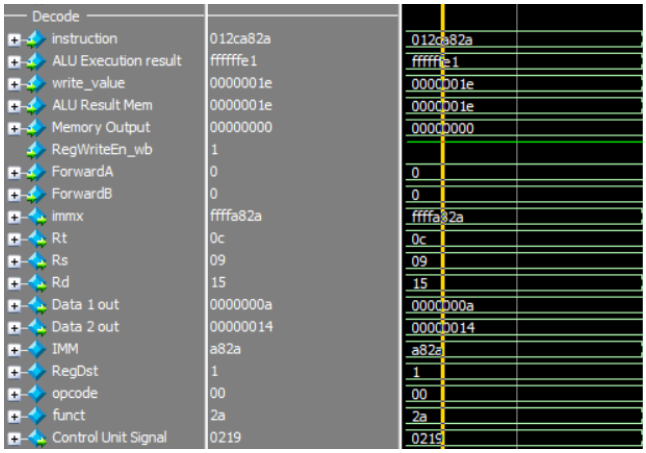

Execute
ALU Result 0000001e
Write Data 00000014
Miss Address ffff9833
destination register 13
Prediction (Hit| Miss) 0
PC Execute 0000000d
Execute Signal 15
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 000067e7
ALU input 1 0000000a
ALU input 2 00000014
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 0000001e
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000014

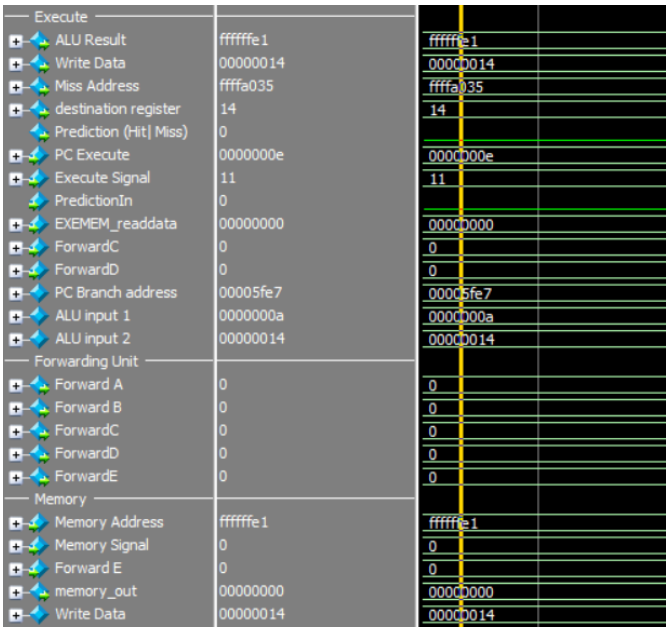

Decode
instruction 012ca027
ALU Execution result 0000001e
write_value 0000001e
ALU Result Mem 0000001e
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffffa027
Rt 0c
Rs 09
Rd 14
Data 1 out 0000000a
Data 2 out 00000014
IMM a027
RegDst 1
opcode 00
funct 27
Control Unit Signal 0211

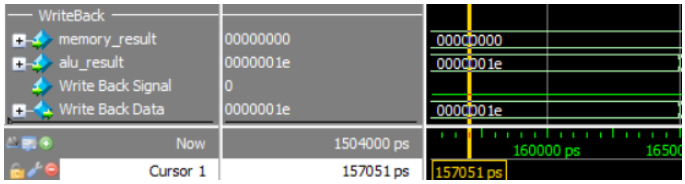

WriteBack
memory_result 00000000
alu_result 0000001e
Write Back Signal 0
Write Back Data 0000001e

# Cycle 16

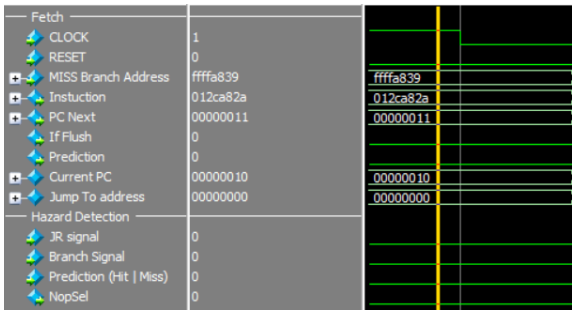

Fetch
CLOCK 1
RESET 0
MISS Branch Address ffffa035
Instuction 012ca027
PC Next 00000010
If Flush 0
Prediction 0
Current PC 0000000f
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

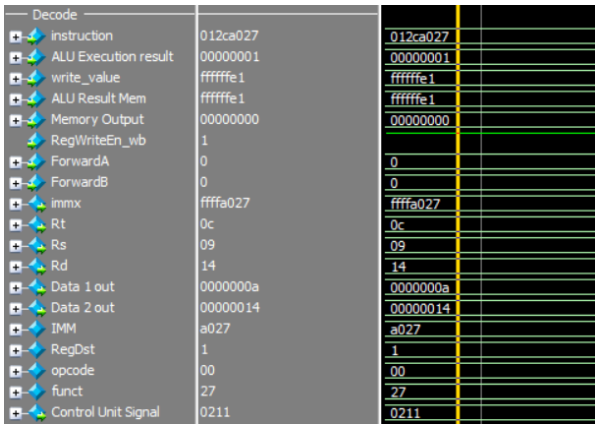

Decode
instruction 012ca82a
ALU Execution result fffffffe1
write_value 0000001e
ALU Result Mem 0000001e
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffffa82a
Rt 0c
Rs 09
Rd 15
Data 1 out 0000000a
Data 2 out 00000014
IMM a82a
RegDst 1
opcode 00
funct 2a
Control Unit Signal 0219

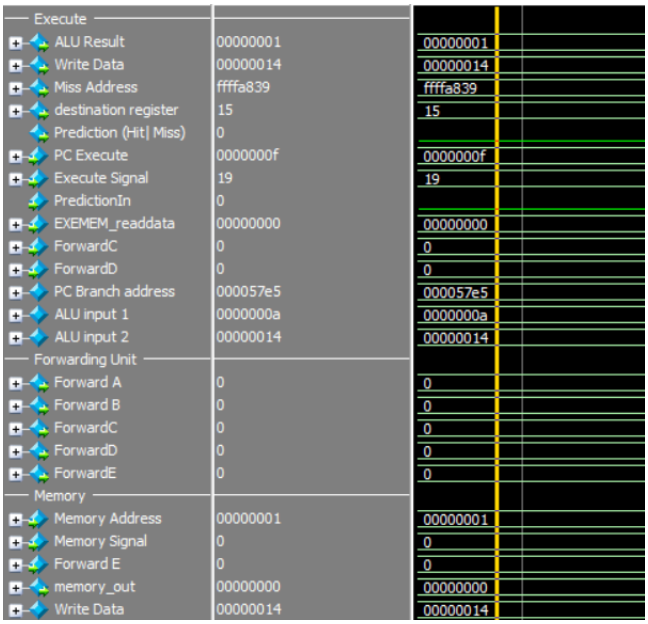

Execute
ALU Result fffffffe1
Write Data 00000014
Miss Address ffffa035
destination register 14
Prediction (Hit| Miss) 0
PC Execute 0000000e
Execute Signal 11
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 00005fe7
ALU input 1 0000000a
ALU input 2 00000014
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address fffffffe1
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000014

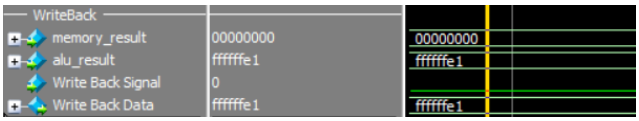

WriteBack
memory_result 00000000
alu_result 0000001e
Write Back Signal 0
Write Back Data 0000001e
Now 1504000 ps
Cursor 1 157051 ps
160000 ps
157051 ps

# Cycle 17

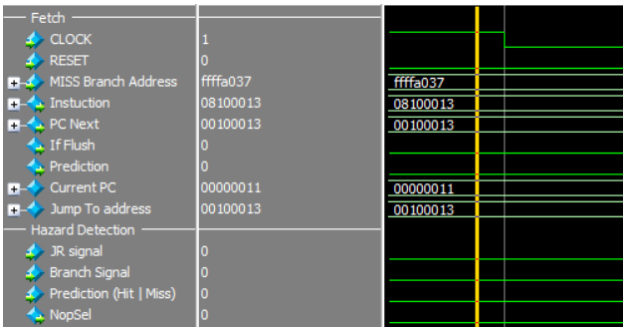

Fetch
CLOCK 1
RESET 0
MISS Branch Address ffffa839
Instuction 012ca82a
PC Next 00000011
If Flush 0
Prediction 0
Current PC 00000010
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

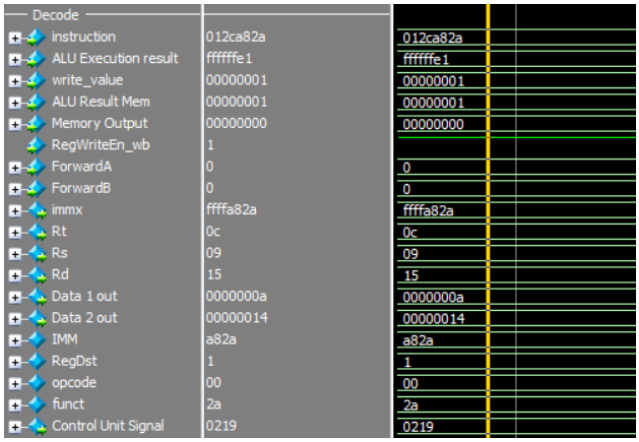

Decode
instruction 012ca027
ALU Execution result 00000001
write_value ffffffe1
ALU Result Mem ffffffe1
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffffa027
Rt 0c
Rs 09
Rd 14
Data 1 out 0000000a
Data 2 out 00000014
IMM a027
RegDst 1
opcode 00
funct 27
Control Unit Signal 0211

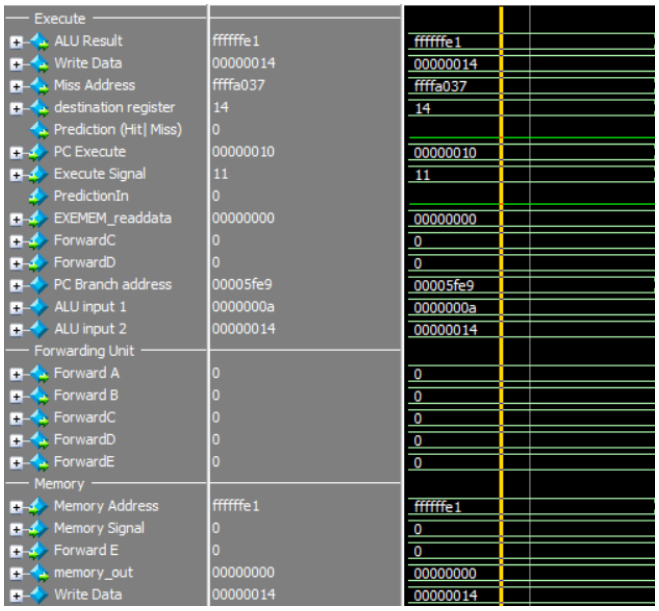

Execute
ALU Result 00000001
Write Data 00000014
Miss Address ffffa839
destination register 15
Prediction (Hit| Miss) 0
PC Execute 0000000f
Execute Signal 19
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 000057e5
ALU input 1 0000000a
ALU input 2 00000014
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 00000001
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000014

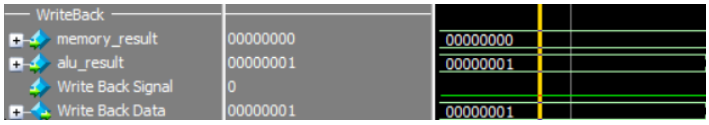

WriteBack
memory_result 00000000
alu_result ffffffe1
Write Back Signal 0
Write Back Data ffffffe1

# Cycle 18

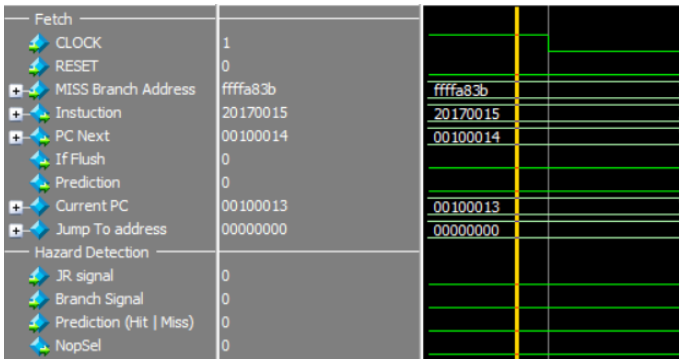

Fetch
CLOCK 1
RESET 0
MISS Branch Address ffffa037
Instuction 08100013
PC Next 00100013
If Flush 0
Prediction 0
Current PC 00000011
Jump To address 00100013
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

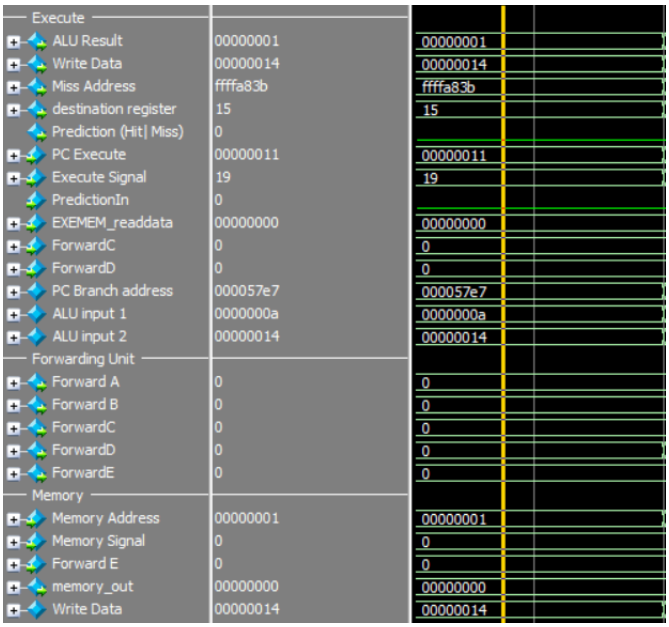

Decode
instruction 012ca82a
ALU Execution result ffffffe1
write_value 00000001
ALU Result Mem 00000001
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffffa82a
Rt 0c
Rs 09
Rd 15
Data 1 out 0000000a
Data 2 out 00000014
IMM a82a
RegDst 1
opcode 00
funct 2a
Control Unit Signal 0219

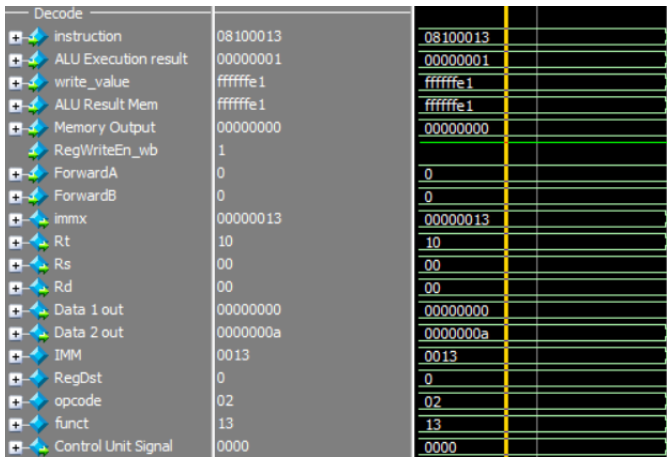

Execute
ALU Result ffffffe1
Write Data 00000014
Miss Address ffffa037
destination register 14
Prediction (Hit| Miss) 0
PC Execute 00000010
Execute Signal 11
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 00005fe9
ALU input 1 0000000a
ALU input 2 00000014
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address ffffffe1
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000014

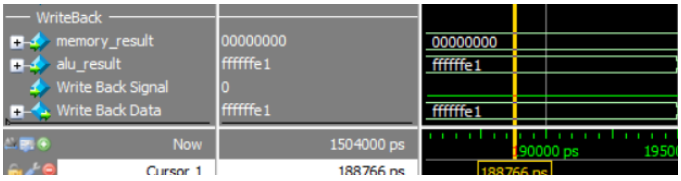

WriteBack
memory_result 00000000
alu_result 00000001
Write Back Signal 0
Write Back Data 00000001

# Cycle 19

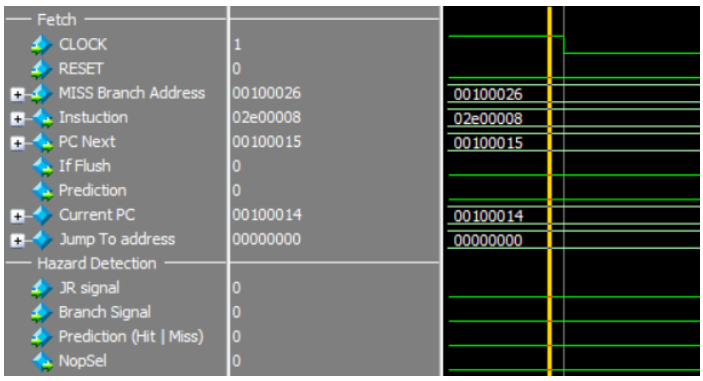

Fetch
CLOCK 1
RESET 0
MISS Branch Address ffffa83b
Instuction 20170015
PC Next 00100014
If Flush 0
Prediction 0
Current PC 00100013
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

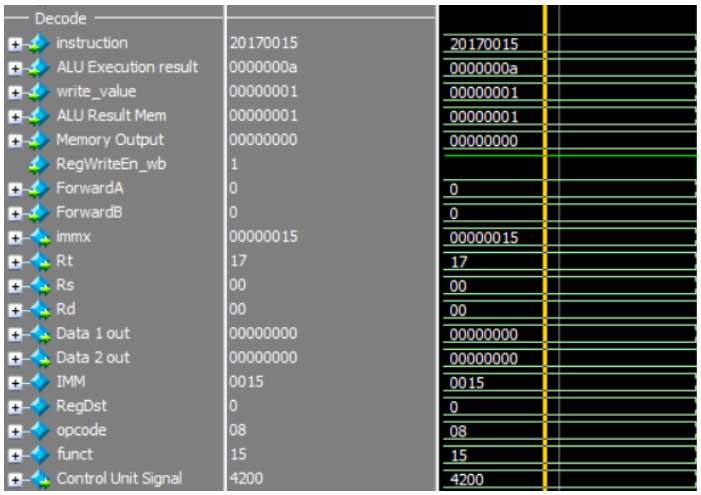

Execute
ALU Result 00000001
Write Data 00000014
Miss Address ffffa83b
destination register 15
Prediction (Hit| Miss) 0
PC Execute 00000011
Execute Signal 19
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 000057e7
ALU input 1 0000000a
ALU input 2 00000014
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 00000001
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000014

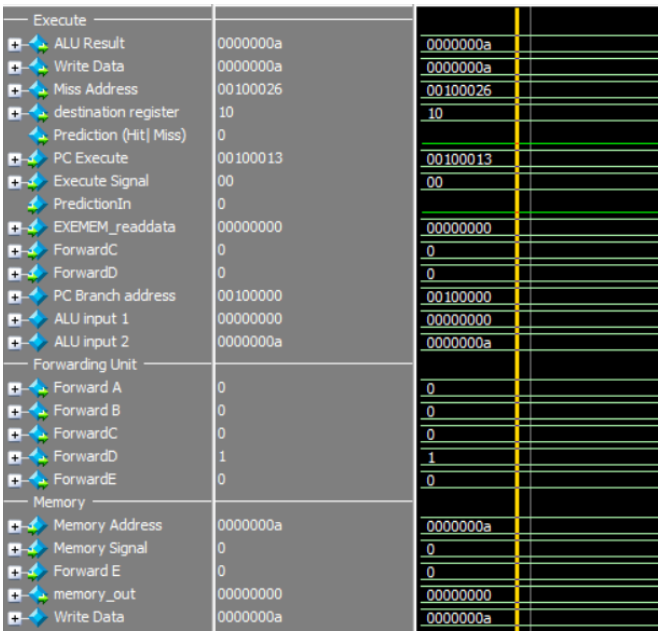

Decode
instruction 08100013
ALU Execution result 00000001
write_value ffffffe1
ALU Result Mem ffffffe1
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx 00000013
Rt 10
Rs 00
Rd 00
Data 1 out 00000000
Data 2 out 0000000a
IMM 0013
RegDst 0
opcode 02
funct 13
Control Unit Signal 0000

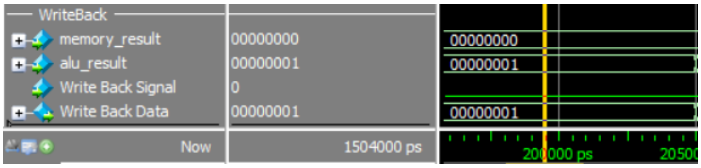

WriteBack
memory_result 00000000
alu_result ffffffe1
Write Back Signal 0
Write Back Data ffffffe1
Now 1504000 ps

# Cycle 20

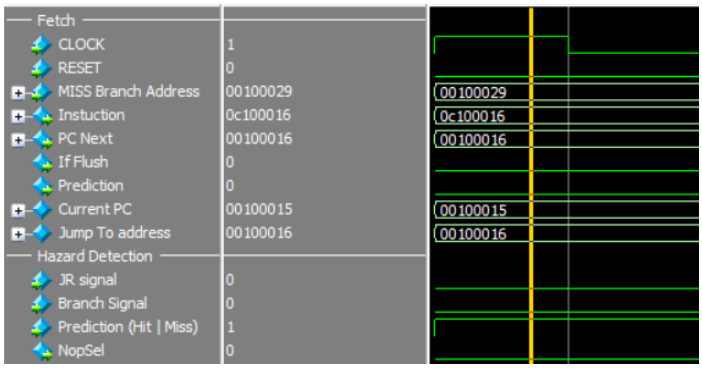

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00100026
Instuction 02e00008
PC Next 00100015
If Flush 0
Prediction 0
Current PC 00100014
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

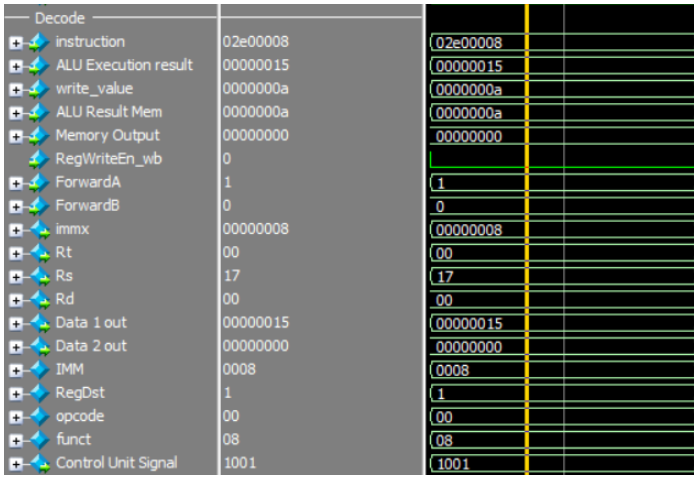

Decode
instruction 20170015
ALU Execution result 0000000a
write_value 00000001
ALU Result Mem 00000001
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx 00000015
Rt 17
Rs 00
Rd 00
Data 1 out 00000000
Data 2 out 00000000
IMM 0015
RegDst 0
opcode 08
funct 15
Control Unit Signal 4200

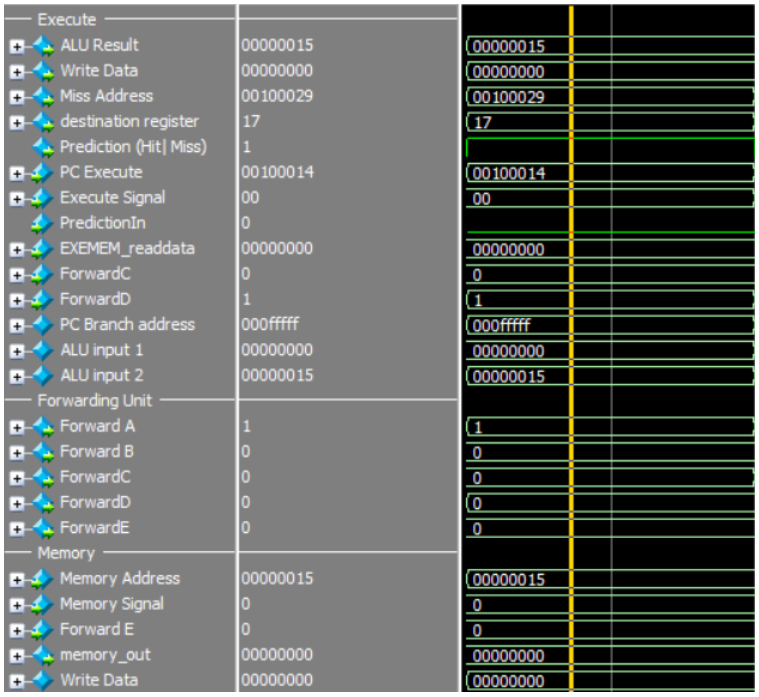

Execute
ALU Result 0000000a
Write Data 0000000a
Miss Address 00100026
destination register 10
Prediction (Hit| Miss) 0
PC Execute 00100013
Execute Signal 00
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 00100000
ALU input 1 00000000
ALU input 2 0000000a
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 0
Memory
Memory Address 0000000a
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 0000000a

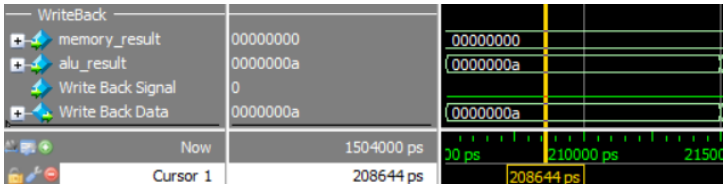

WriteBack
memory_result 00000000
alu_result 00000001
Write Back Signal 0
Write Back Data 00000001
Now 1504000 ps

# Cycle 21

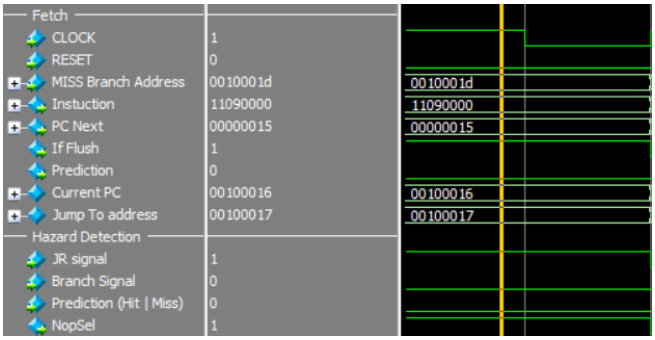

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00100029
Instuction 0c100016
PC Next 00100016
If Flush 0
Prediction 0
Current PC 00100015
Jump To address 00100016
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 1
NopSel 0

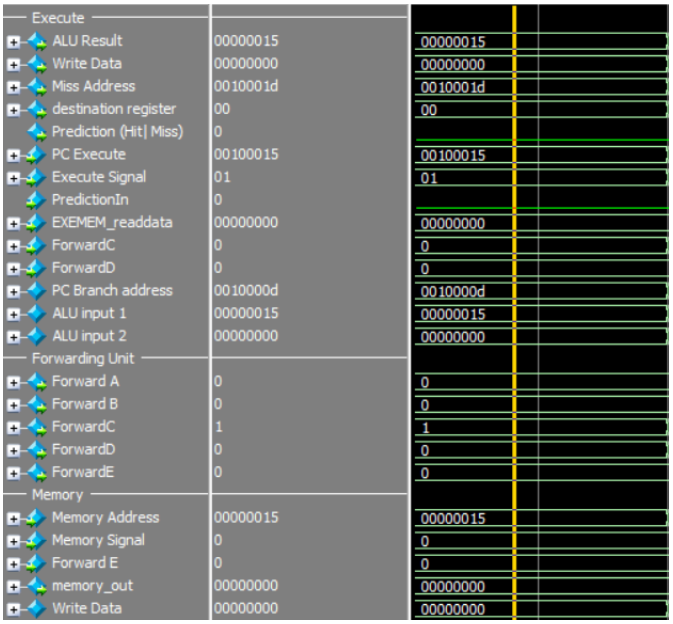

Decode
instruction 02e00008
ALU Execution result 00000015
write_value 0000000a
ALU Result Mem 0000000a
Memory Output 00000000
RegWriteEn_wb 0
ForwardA 1
ForwardB 0
immx 00000008
Rt 00
Rs 17
Rd 00
Data 1 out 00000015
Data 2 out 00000000
IMM 0008
RegDst 1
opcode 00
funct 08
Control Unit Signal 1001

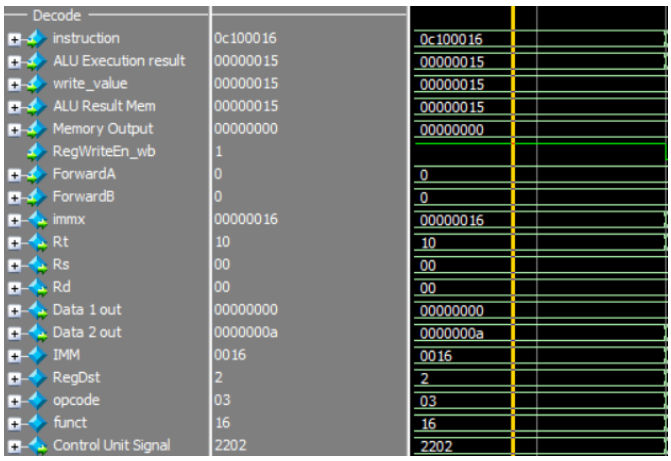

Execute
ALU Result 00000015
Write Data 00000000
Miss Address 00100029
destination register 17
Prediction (Hit| Miss) 1
PC Execute 00100014
Execute Signal 00
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address 000fffff
ALU input 1 00000000
ALU input 2 00000015
Forwarding Unit
Forward A 1
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 00000015
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000


WriteBack
memory_result 00000000
alu_result 0000000a
Write Back Signal 0
Write Back Data 0000000a
Now 1504000 ps
Cursor 1 208644 ps
210000 ps
208644 ps

# Cycle 22

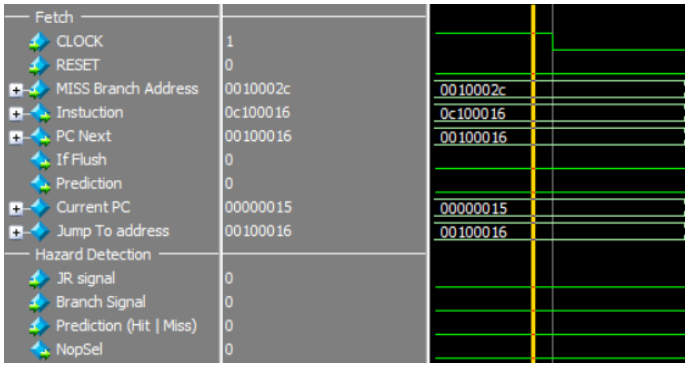

Fetch
CLOCK 1
RESET 0
MISS Branch Address 0010001d
Instuction 11090000
PC Next 00000015
If Flush 1
Prediction 0
Current PC 00100016
Jump To address 00100017
Hazard Detection
JR signal 1
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 1

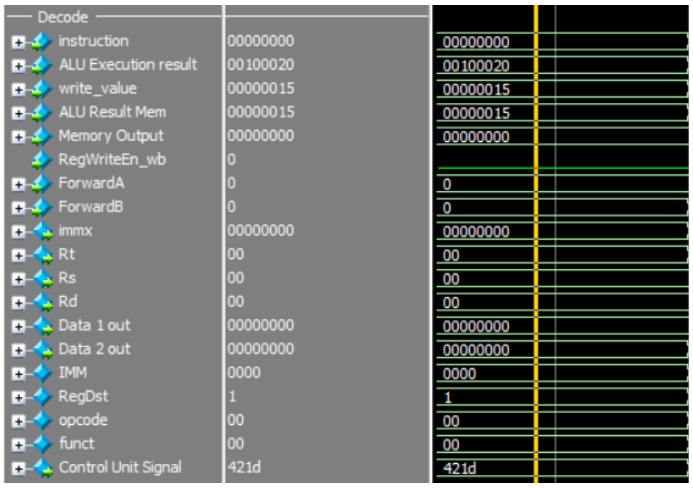

Execute
ALU Result 00000015
Write Data 00000000
Miss Address 0010001d
destination register 00
Prediction (Hit| Miss) 0
PC Execute 00100015
Execute Signal 01
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 0010000d
ALU input 1 00000015
ALU input 2 00000000
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 1
ForwardD 0
ForwardE 0
Memory
Memory Address 00000015
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

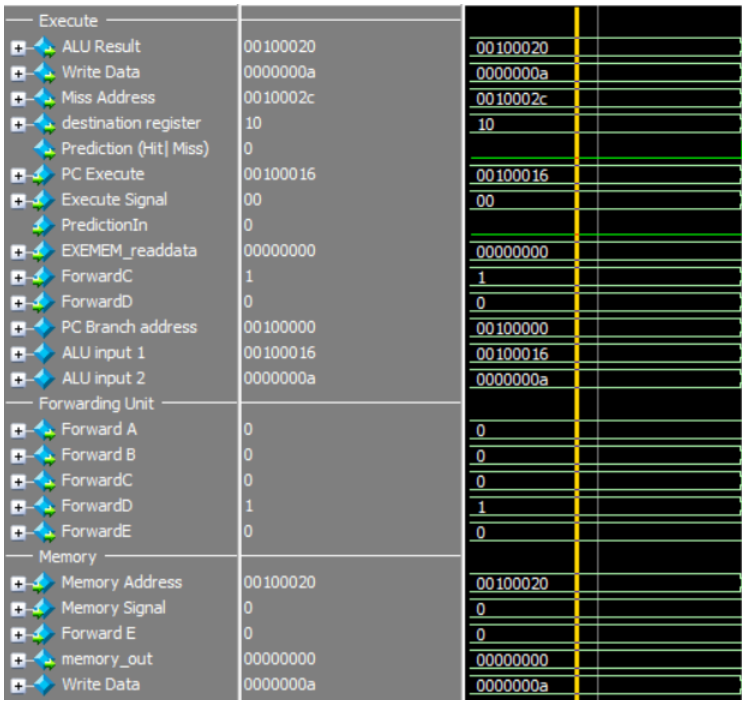

Decode
instruction 0c100016
ALU Execution result 00000015
write_value 00000015
ALU Result Mem 00000015
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx 00000016
Rt 10
Rs 00
Rd 00
Data 1 out 00000000
Data 2 out 0000000a
IMM 0016
RegDst 2
opcode 03
funct 16
Control Unit Signal 2202


WriteBack
memory_result 00000000
alu_result 00000015
Write Back Signal 0
Write Back Data 00000015

# Cycle 23

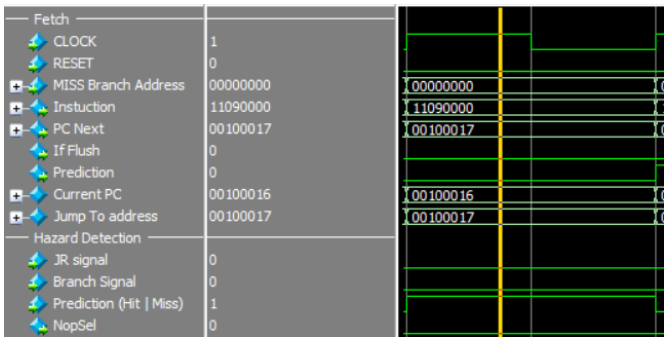

Fetch
CLOCK 1
RESET 0
MISS Branch Address 0010002c
Instuction 0c100016
PC Next 00100016
If Flush 0
Prediction 0
Current PC 00000015
Jump To address 00100016
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

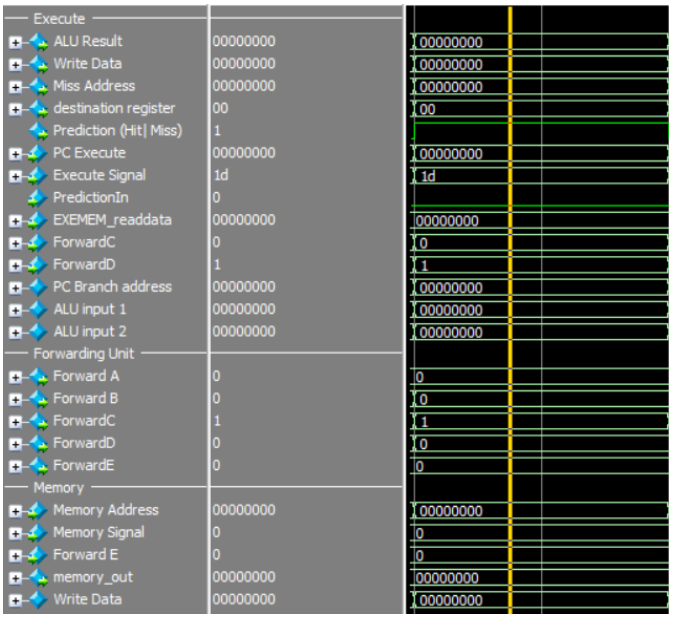

Decode
instruction 00000000
ALU Execution result 00100020
write_value 00000015
ALU Result Mem 00000015
Memory Output 00000000
RegWriteEn_wb 0
ForwardA 0
ForwardB 0
immx 00000000
Rt 00
Rs 00
Rd 00
Data 1 out 00000000
Data 2 out 00000000
IMM 0000
RegDst 1
opcode 00
funct 00
Control Unit Signal 421d

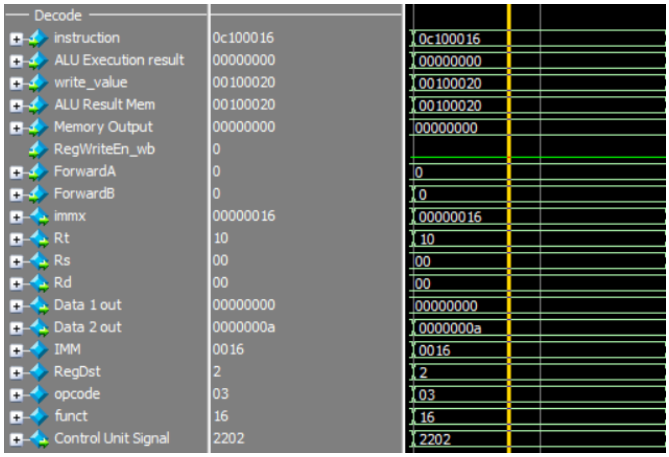

Execute
ALU Result 00100020
Write Data 0000000a
Miss Address 0010002c
destination register 10
Prediction (Hit| Miss) 0
PC Execute 00100016
Execute Signal 00
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 1
ForwardD 0
PC Branch address 00100000
ALU input 1 00100016
ALU input 2 0000000a
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 1
ForwardE 0
Memory
Memory Address 00100020
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 0000000a


WriteBack
memory_result 00000000
alu_result 00000015
Write Back Signal 0
Write Back Data 00000015

# Cycle 24

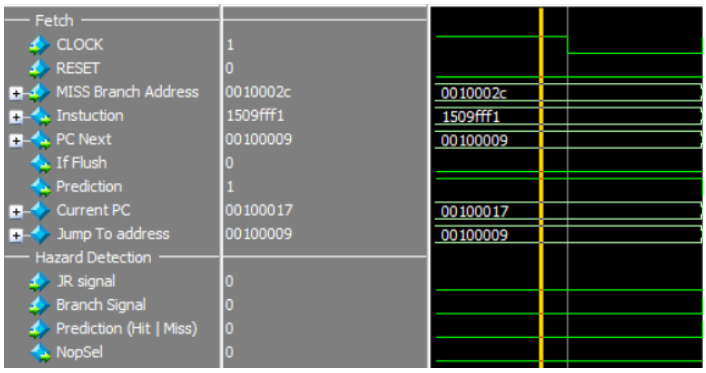

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00000000
Instuction 11090000
PC Next 00100017
If Flush 0
Prediction 0
Current PC 00100016
Jump To address 00100017
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 1
NopSel 0

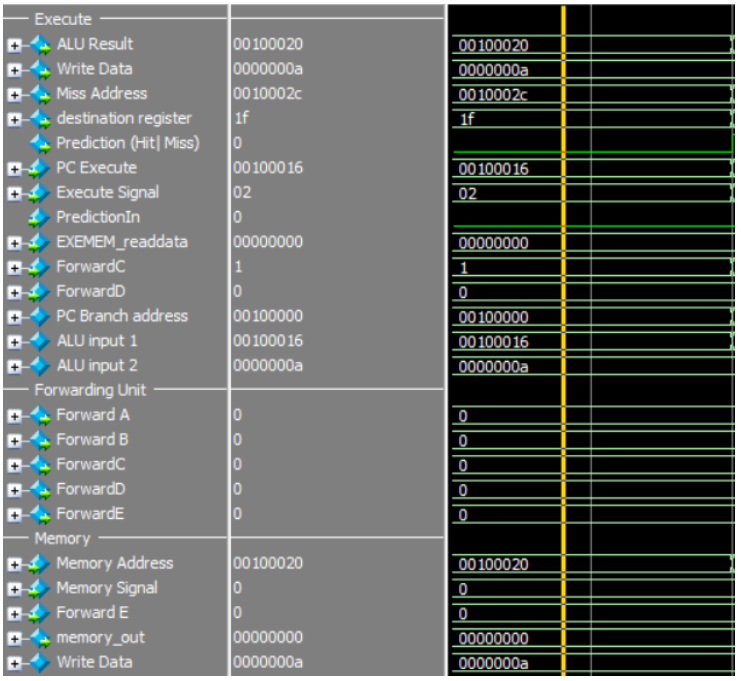

Execute
ALU Result 00000000
Write Data 00000000
Miss Address 00000000
destination register 00
Prediction (Hit| Miss) 1
PC Execute 00000000
Execute Signal 1d
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 1
PC Branch address 00000000
ALU input 1 00000000
ALU input 2 00000000
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 1
ForwardD 0
ForwardE 0
Memory
Memory Address 00000000
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000

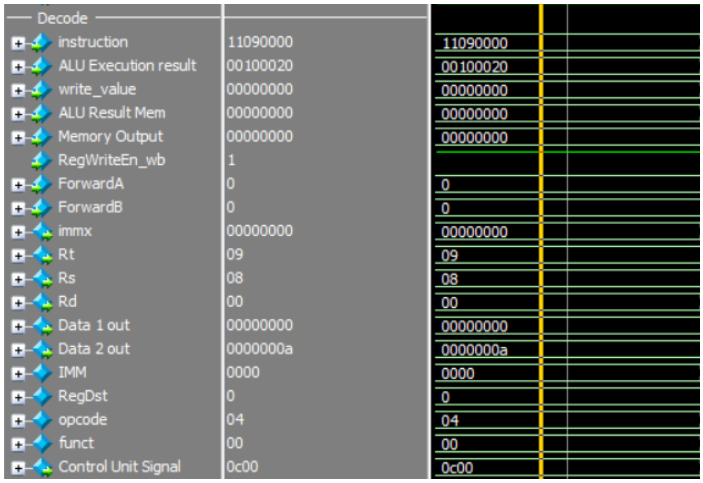

Decode
instruction 0c100016
ALU Execution result 00000000
write_value 00100020
ALU Result Mem 00100020
Memory Output 00000000
RegWriteEn_wb 0
ForwardA 0
ForwardB 0
immx 00000016
Rt 10
Rs 00
Rd 00
Data 1 out 00000000
Data 2 out 0000000a
IMM 0016
RegDst 2
opcode 03
funct 16
Control Unit Signal 2202


WriteBack
memory_result 00000000
alu_result 00100020
Write Back Signal 0
Write Back Data 00100020

# Cycle 25

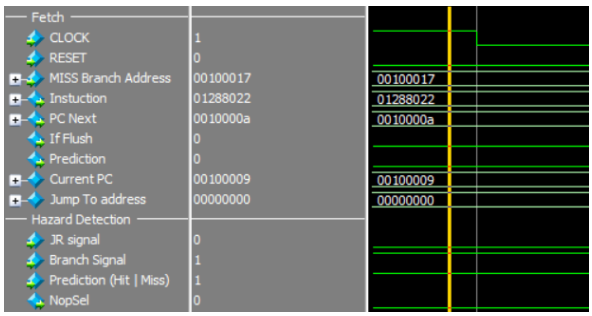

Fetch
CLOCK 1
RESET 0
MISS Branch Address 0010002c
Instuction 1509fff1
PC Next 00100009
If Flush 0
Prediction 1
Current PC 00100017
Jump To address 00100009
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0

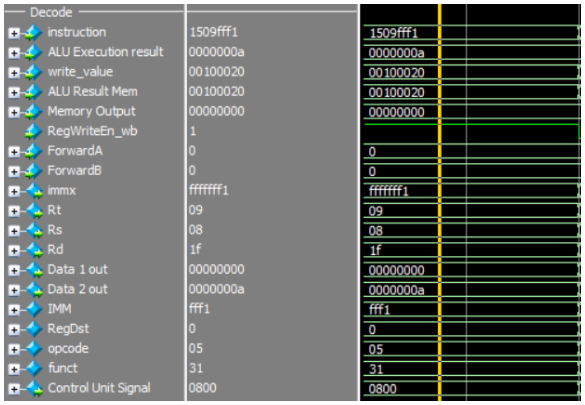

Execute
ALU Result 00100020
Write Data 0000000a
Miss Address 0010002c
destination register 1f
Prediction (Hit| Miss) 0
PC Execute 00100016
Execute Signal 02
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 1
ForwardD 0
PC Branch address 00100000
ALU input 1 00100016
ALU input 2 0000000a
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 00100020
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 0000000a

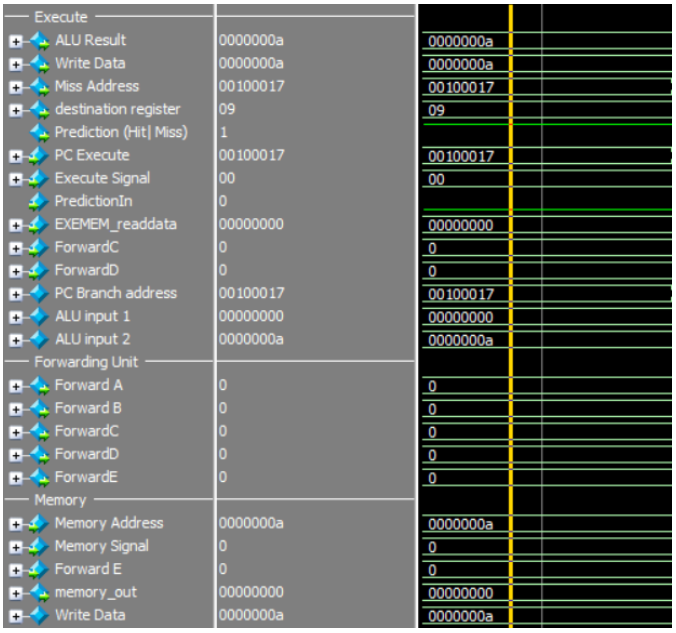

Decode
instruction 11090000
ALU Execution result 00100020
write_value 00000000
ALU Result Mem 00000000
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx 00000000
Rt 09
Rs 08
Rd 00
Data 1 out 00000000
Data 2 out 0000000a
IMM 0000
RegDst 0
opcode 04
funct 00
Control Unit Signal 0c00

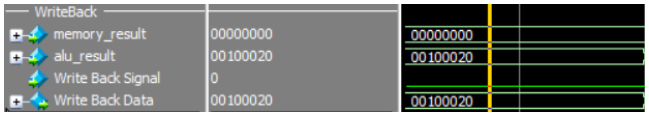

WriteBack
memory_result 00000000
alu_result 00000000
Write Back Signal 0
Write Back Data 00000000

# Cycle 26

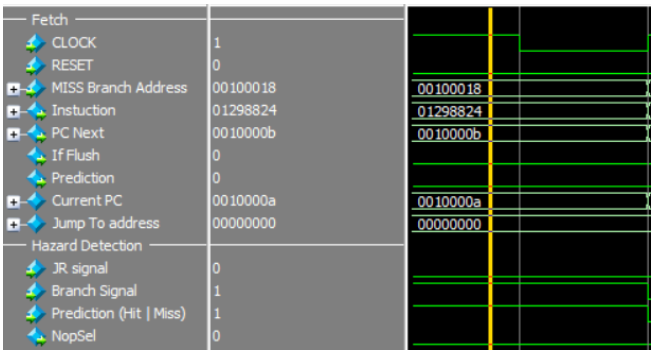

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00100017
Instuction 01288022
PC Next 0010000a
If Flush 0
Prediction 0
Current PC 00100009
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 1
Prediction (Hit | Miss) 1
NopSel 0

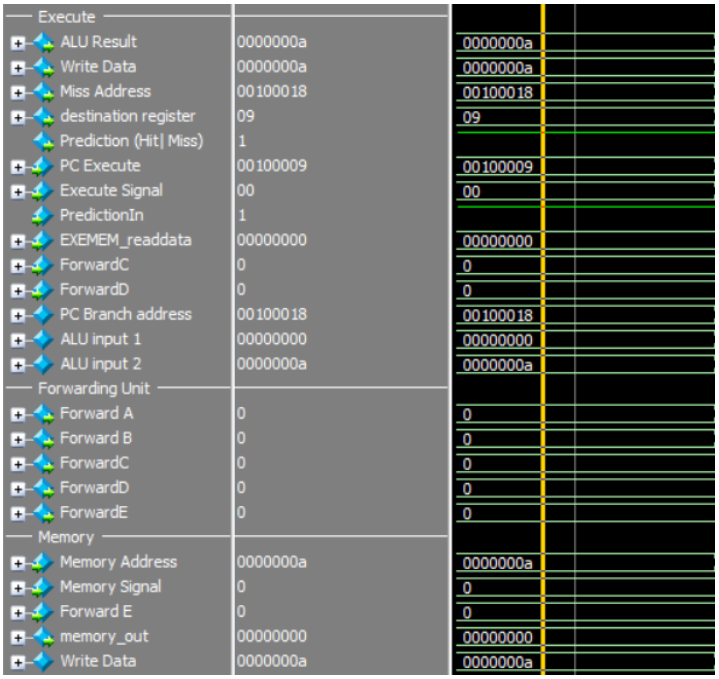

Decode
instruction 1509fff1
ALU Execution result 0000000a
write_value 00100020
ALU Result Mem 00100020
Memory Output 00000000
RegWriteEn_wb 1
ForwardA 0
ForwardB 0
immx ffffffff1
Rt 09
Rs 08
Rd 1f
Data 1 out 00000000
Data 2 out 0000000a
IMM fff1
RegDst 0
opcode 05
funct 31
Control Unit Signal 0800

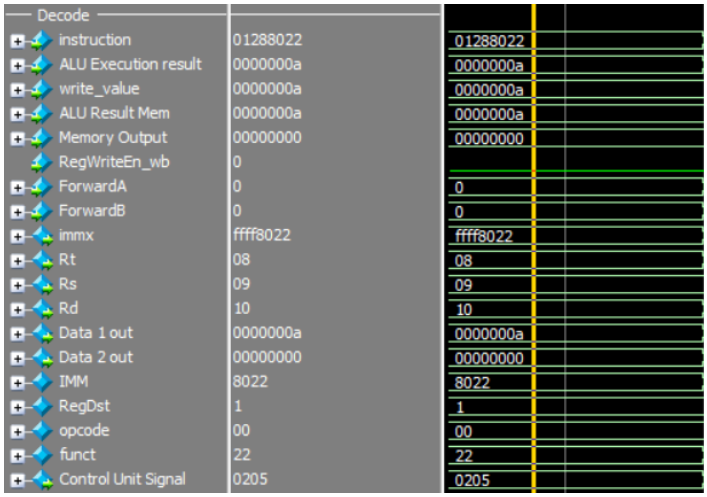

Execute
ALU Result 0000000a
Write Data 0000000a
Miss Address 00100017
destination register 09
Prediction (Hit| Miss) 1
PC Execute 00100017
Execute Signal 00
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 00100017
ALU input 1 00000000
ALU input 2 0000000a
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 0000000a
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 0000000a

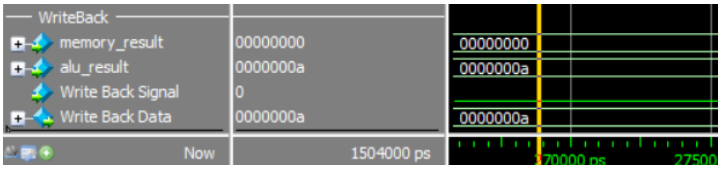

WriteBack
memory_result 00000000
alu_result 00100020
Write Back Signal 0
Write Back Data 00100020

# Cycle 27

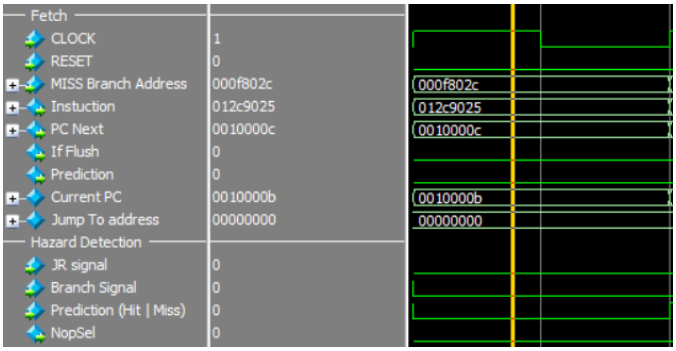

Fetch
CLOCK 1
RESET 0
MISS Branch Address 00100018
Instuction 01298824
PC Next 0010000b
If Flush 0
Prediction 0
Current PC 0010000a
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 1
Prediction (Hit | Miss) 1
NopSel 0

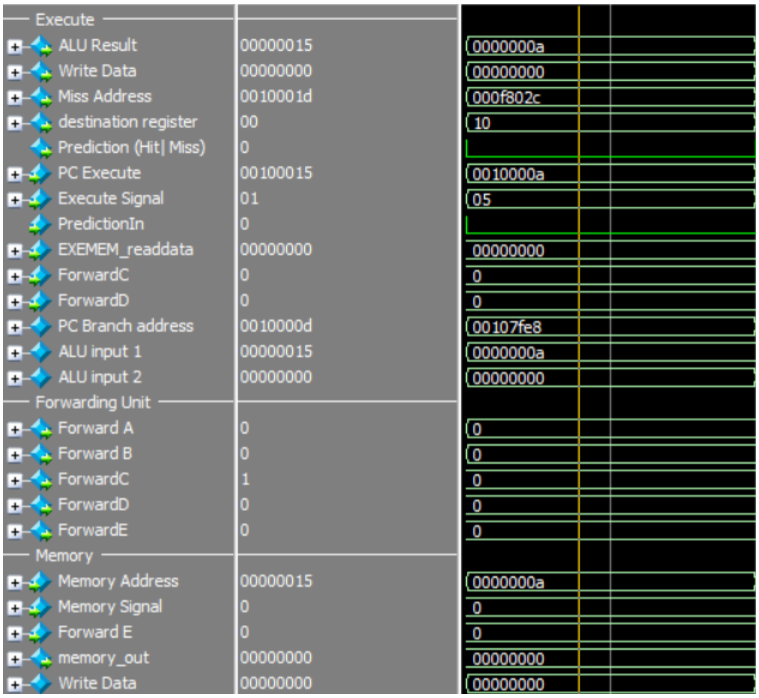

Execute
ALU Result 0000000a
Write Data 0000000a
Miss Address 00100018
destination register 09
Prediction (Hit| Miss) 1
PC Execute 00100009
Execute Signal 00
PredictionIn 1
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 00100018
ALU input 1 00000000
ALU input 2 0000000a
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 0
ForwardD 0
ForwardE 0
Memory
Memory Address 0000000a
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 0000000a

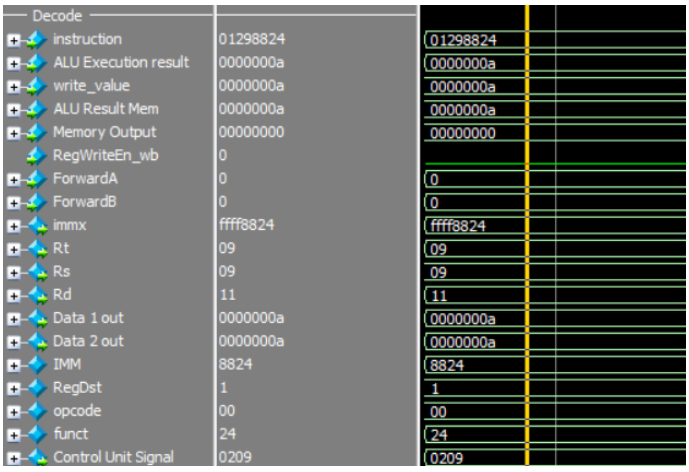

Decode
instruction 01288022
ALU Execution result 0000000a
write_value 0000000a
ALU Result Mem 0000000a
Memory Output 00000000
RegWriteEn_wb 0
ForwardA 0
ForwardB 0
immx ffff8022
Rt 08
Rs 09
Rd 10
Data 1 out 0000000a
Data 2 out 00000000
IMM 8022
RegDst 1
opcode 00
funct 22
Control Unit Signal 0205


WriteBack
memory_result 00000000
alu_result 0000000a
Write Back Signal 0
Write Back Data 0000000a
Now
1504000 ps

# Cycle 28


Fetch
CLOCK 1
RESET 0
MISS Branch Address 000f802c
Instuction 012c9025
PC Next 0010000c
If Flush 0
Prediction 0
Current PC 0010000b
Jump To address 00000000
Hazard Detection
JR signal 0
Branch Signal 0
Prediction (Hit | Miss) 0
NopSel 0


Execute
ALU Result 00000015
Write Data 00000000
Miss Address 0010001d
destination register 00
Prediction (Hit| Miss) 0
PC Execute 00100015
Execute Signal 01
PredictionIn 0
EXEMEM_readdata 00000000
ForwardC 0
ForwardD 0
PC Branch address 0010000d
ALU input 1 00000015
ALU input 2 00000000
Forwarding Unit
Forward A 0
Forward B 0
ForwardC 1
ForwardD 0
ForwardE 0
Memory
Memory Address 00000015
Memory Signal 0
Forward E 0
memory_out 00000000
Write Data 00000000


Decode
instruction 01298824
ALU Execution result 0000000a
write_value 0000000a
ALU Result Mem 0000000a
Memory Output 00000000
RegWriteEn_wb 0
ForwardA 0
ForwardB 0
immx ffff8824
Rt 09
Rs 09
Rd 11
Data 1 out 0000000a
Data 2 out 0000000a
IMM 8824
RegDst 1
opcode 00
funct 24
Control Unit Signal 0209


WriteBack
memory_result 00000000
alu_result 00000015
Write Back Signal 0
Write Back Data 00000015

## 3. Benchmark

Benchmark 1 – Data Manipulation Instructions

| Machine Code | Mnemonic | Operands | Description |
|---|---|---|---|
| 2001000A | ADDI | $1, $0, 0xA | Adds an immediate value (0xA) to the contents of register $0 and stores the result in register $1. |
| 3402000B | ORI | $2, $0, 0xB | Performs a bitwise OR operation between an immediate value (0xB) and the contents of register $0, storing the result in register $2. |
| 3803000C | XORI | $3, $0, 0xC | Executes a bitwise XOR operation between an immediate value (0xC) and the contents of register $0, placing the result in register $3. |
| 00222020 | ADD | $4, $1, $2 | Adds the values stored in registers $1 and $2, and places the result in register $4. |
| 00832822 | SUB | $5, $4, $3 | Subtracts the value in register $3 from the value in register $4, storing the result in register $5. |
| 00233024 | AND | $6, $1, $3 | Performs a bitwise AND operation between the contents of registers $1 and $3, storing the result in register $6. |
| 00433825 | OR | $7, $2, $3 | Executes a bitwise OR operation between the values in registers $2 and $3, with the result stored in register $7. |
| 00434027 | NOR | $8, $2, $3 | Conducts a bitwise NOR operation between registers $2 and $3, and stores the negated result in register $8. |
| 00224826 | XOR | $9, $1, $2 | Performs a bitwise XOR operation between the contents of registers $1 and $2, saving the result in register $9. |
| 0022502A | SLT | $10, $1, $2 | Sets register $10 to 1 if the value in register $1 is less than that in register $2; otherwise, sets it to 0. |
| 0061582C | SGT | $11, $3, $1 | Sets register $11 to 1 if the value in register $3 is greater than that in register $1; otherwise, sets it to 0. |
| 00206080 | SLL | $12, $1, 0x2 | Shifts the value in register $1 to the left by 2 bits, storing the result in register $12. |
| 00406842 | SRL | $13, $2, 0x1 | Shifts the value in register $2 to the right by 1 bit, and places the result in register $13. |
| 8C0E0000 | LW | $14, 0x0($0) | Loads the word at memory address 0x0 (offset from $0) into register $14. |
| AC040000 | SW | $4, 0x0($0) | Stores the contents of register $4 into the memory address 0x0 (offset from $0). |

## Benchmark 2 – Control Flow Instructions

| Machine Code | Mnemonic | Operands | Description |
|---|---|---|---|
| **2001FFFB** | ADDI | $1, $0, 0XFFFB | Adds an immediate value (0XFFFB) to the contents of register $0 and stores the result in register $1. |
| **20020005** | ADDI | $2, $0, 0X5 | Adds an immediate value (0X5) to the contents of register $0 and stores the result in register $2. |
| **20030005** | ADDI | $3, $0, 0X5 | Adds an immediate value (0X5) to the contents of register $0 and stores the result in register $3. |
| **20040054** | ADDI | $4, $0, 0X54 | Adds an immediate value (0X54) to the contents of register $0 and stores the result in register $4. |
| **10620002** | BEQ | $2, $3, L2 | Branches to label L2 if the values in registers $2 and $3 are equal. |
| **00000000** | SLL | $0, $0, 0x0 | Performs a no-operation by shifting the value in register $0 left by 0 bits. |
| **0C100004** | JAL | L1 | Jumps to the address of label L1 and links the return address in register $31. |
| **14620000** | BNE | $2, $3, L3 | Branches to label L3 if the values in registers $2 and $3 are not equal. |
| **00000000** | SLL | $0, $0, 0x0 | Performs a no-operation by shifting the value in register $0 left by 0 bits. |
| **0020D82A** | SLT | $27, $1, $zero | Sets register $27 to 1 if the value in register $1 is less than the value in $zero; otherwise, sets it to 0. |
| **141B0002** | BNE | $27, $zero, L5 | Branches to label L5 if the value in register $27 is not equal to 0. |
| **00000000** | SLL | $0, $0, 0x0 | Performs a no-operation by shifting the value in register $0 left by 0 bits. |
| **0C100009** | JAL | L4 | Jumps to the address of label L4 and links the return address in register $31. |
| **0040D82A** | SLT | $27, $2, $zero | Sets register $27 to 1 if the value in register $2 is less than the value in $zero; otherwise, sets it to 0. |
| **101B0002** | BEQ | $27, $zero, L6 | Branches to label L6 if the value in register $27 is equal to 0. |
| **00000000** | SLL | $0, $0, 0x0 | Performs a no-operation by shifting the value in register $0 left by 0 bits. |
| **0C10000D** | JAL | L5 | Jumps to the address of label L5 and links the return address in register $31. |
| **0C100014** | JAL | L7 | Jumps to the address of label L7 and links the return address in register $31. |
| **00000000** | SLL | $0, $0, 0x0 | Performs a no-operation by shifting the value in register $0 left by 0 bits. |
| **0C100011** | JAL | L6 | Jumps to the address of label L6 and links the return address in register $31. |
| **00800008** | JR | $4 | Jumps to the address contained in register $4. |
| **00000000** | SLL | $0, $0, 0x0 | Performs a no-operation by shifting the value in register $0 left by 0 bits. |
| **0C100014** | JAL | L7 | Jumps to the address of label L7 and links the return address in register $31. |
| **00000000** | SLL | $0, $0, 0x0 | Performs a no-operation by shifting the value in register $0 left by 0 bits. |

## Benchmark 3 – Sum of Numbers

| Machine Code | Mnemonic | Operand(s) | Description |
|---|---|---|---|
| **34020001** | ORI | $2, $0, 0x1 | Logical OR Immediate: Performs a bitwise OR between register $0 (0) and the immediate value 0x1, storing the result in $2. |
| **34030000** | ORI | $3, $0, 0x0 | Logical OR Immediate: Performs a bitwise OR between register $0 (0) and the immediate value 0x0, storing the result in $3. |
| **3404000A** | ORI | $4, $0, 0xA | Logical OR Immediate: Performs a bitwise OR between register $0 (0) and the immediate value 0xA, storing the result in $4. |
| **00621820** | ADD | $3, $3, $2 | Add: Adds the values in registers $3 and $2, storing the result in register $3. |
| **20420001** | ADDI | $2, $2, 0x1 | Add Immediate: Adds the value in register $2 to the immediate value 0x1, storing the result back in register $2. |
| **0082282A** | SLT | $5, $4, $2 | Set Less Than: Sets register $5 to 1 if the value in $4 is less than that in $2, otherwise sets $5 to 0. |
| **1005FFFC** | BEQ | $5, $0, SUM_LOOP | Branch on Equal: If the value in register $5 is equal to 0 (false), it branches to the label "SUM_LOOP." |
| **AC030000** | SW | $3, 0x0($0) | Store Word: Stores the value in register $3 at the memory address given by adding 0x0 to the value in register $0 (base address). |
| **00000000** | SLL | $0, $0, 0x0 | Shift Left Logical: Shifts the bits of register $0 left by 0 positions (no change). |

## Benchmark 4 – Binary Search

| Machine Code | Mnemonic | Operand(s) | Description |
|---|---|---|---|
| 20010000 | ADDI | $1, $0, 0x0 | Add Immediate: Adds the value in register $0 (0) to the immediate value 0x0, storing the result in register $1. |
| 2002000B | ADDI | $2, $0, 0xB | Add Immediate: Adds the value in register $0 (0) to the immediate value 0xB, storing the result in register $2. |
| 20030007 | ADDI | $3, $0, 0x7 | Add Immediate: Adds the value in register $0 (0) to the immediate value 0x7, storing the result in register $3. |
| 0041382A | SLT | $7, $2, $1 | Set Less Than: Sets register $7 to 1 if the value in $2 is less than that in $1, otherwise sets $7 to 0. |
| 14E0000D | BNE | $0, $7, notFound | Branch on Not Equal: If register $0 is not equal to $7, it branches to the label "notFound." |
| 00412020 | ADD | $4, $2, $1 | Add: Adds the values in registers $2 and $1, storing the result in register $4. |
| 00802842 | SRL | $5, $4, 0x1 | Shift Right Logical: Shifts the bits in register $4 right by 1 position, storing the result in register $5. |
| 8CA60000 | LW | $6, 0x0($5) | Load Word: Loads the word from memory address 0x0 offset by the value in register $5 into register $6. |
| 10C30007 | BEQ | $3, $6, found | Branch on Equal: If register $3 equals register $6, it branches to the label "found." |
| 00C3302A | SLT | $6, $6, $3 | Set Less Than: Sets register $6 to 1 if the value in $6 is less than that in $3, otherwise sets $6 to 0. |
| 10060001 | BEQ | $6, $0, leftHalf | Branch on Equal: If register $6 is equal to 0 (false), it branches to the label "leftHalf." |
| 0810000E | J | rightHalf | Jump: Jumps to the label "rightHalf." |
| 20A2FFFF | ADDI | $2, $5, 0xFFFF | Add Immediate: Adds the value in register $5 to the immediate value 0xFFFF, storing the result in register $2. |
| 08100003 | J | loop | Jump: Jumps to the label "loop." |
| 20A10001 | ADDI | $1, $5, 0x1 | Add Immediate: Adds the value in register $5 to the immediate value 0x1, storing the result in register $1. |
| 08100003 | J | loop | Jump: Jumps to the label "loop." |
| 00054020 | ADD | $8, $0, $5 | Add: Adds the values in registers $0 and $5, storing the result in register $8. |
| 08100014 | J | finish | Jump: Jumps to the label "finish." |
| 2008FFFF | ADDI | $8, $0, 0xFFFF | Add Immediate: Adds the value in register $0 (0) to the immediate value 0xFFFF, storing the result in register $8. |
| 08100014 | J | finish | Jump: Jumps to the label "finish." |
| 00000000 | SLL | $0, $0, 0x0 | Shift Left Logical: Shifts the bits of register $0 left by 0 positions (no change). |

## Benchmark 5 – Max and Min in Array

| Machine Code | Mnemonic | Operand | Description |
|---|---|---|---|
| 34020000 | ORI | $2, $0, 0x0 | Perform bitwise OR on $0 and 0x0, store result in $2. |
| 2014000A | ADDI | $20, $0, 0xA | Add immediate value 0xA to $0 (which is 0), store result in $20. |
| 381F0001 | XORI | $31, $0, 0x1 | Perform bitwise XOR on $0 and 0x1, store result in $31. |
| 30050000 | ANDI | $5, $0, 0x0 | Perform bitwise AND on $0 and 0x0, store result in $5. |
| 8CAA0000 | LW | $10, 0x0($5) | Load word from memory address 0x0 + content of $5 into register $10. |
| 8CAF0000 | LW | $15, 0x0($5) | Load word from memory address 0x0 + content of $5 into register $15. |
| 20420001 | ADDI | $2, $2, 0x1 | Add immediate value 0x1 to $2, store result in $2. |
| 0282C82C | SGT | $25, $20, $2 | Set $25 to 1 if $20 > $2, otherwise set it to 0. |
| 17F90009 | BNE | $25, $31, END | Branch to END if $25 is not equal to $31. |
| 8CB00000 | LW | $16, 0x0($5) | Load word from memory address 0x0 + content of $5 into register $16. |
| 020AD02C | SGT | $26, $16, $10 | Set $26 to 1 if $16 > $10, otherwise set it to 0. |
| 101A0002 | BEQ | $26, $0, MIN | Branch to MIN if $26 is equal to $0. |
| 02005025 | OR | $10, $16, $0 | Perform bitwise OR on $16 and $0, store result in $10. |
| 08100006 | J | LOOP | Jump to the LOOP address. |
| 020FD82A | SLT | $27, $16, $15 | Set $27 to 1 if $16 < $15, otherwise set it to 0. |
| 101BFFF6 | BEQ | $27, $0, LOOP | Branch to LOOP if $27 is equal to $0. |
| 02007820 | ADD | $15, $16, $0 | Add contents of $16 and $0 (which is 0), store result in $15. |
| 08100006 | J | LOOP | Jump to the LOOP address. |
| 00000000 | SLL | $0, $0, 0x0 | Shift $0 left by 0 bits (no operation). |

## Benchmark 6 – Insertion Sort

| Machine Code | Mnemonic | Operand | Description |
|---|---|---|---|
| 34020000 | ORI | $2, $0, 0x0 | Perform bitwise OR on $0 and 0x0, store result in $2. |
| 380A0000 | XORI | $10, $0, 0x0 | Perform bitwise XOR on $0 and 0x0, store result in $10. |
| 2014000A | ADDI | $20, $0, 0xA | Add immediate value 0xA to $0, store result in $20. |
| 00002820 | ADD | $5, $0, $0 | Add $0 and $0, store result in $5. |
| 20010001 | ADDI | $1, $0, 0x1 | Add immediate value 0x1 to $0, store result in $1. |
| 2016FFFF | ADDI | $22, $0, 0xFFFF | Add immediate value 0xFFFF to $0, store result in $22. |
| 8CAF0000 | LW | $15, 0x0($5) | Load word from memory address 0x0 + content of $5 into register $15. |
| 01E05025 | OR | $10, $15, $0 | Perform bitwise OR on $15 and $0, store result in $10. |
| 2022FFFF | ADDI | $2, $1, 0xFFFF | Add immediate value 0xFFFF to $1, store result in $2. |
| 8CD00000 | LW | $16, 0x0($6) | Load word from memory address 0x0 + content of $6 into register $16. |
| 0056C82C | SGT | $25, $2, $22 | Set $25 to 1 if $2 > $22, otherwise set it to 0. |
| 020AD02C | SGT | $26, $16, $10 | Set $26 to 1 if $16 > $10, otherwise set it to 0. |
| 0359D824 | AND | $27, $26, $25 | Perform bitwise AND on $26 and $25, store result in $27. |
| 101B0004 | BEQ | $27, $0, EXIT2 | Branch to EXIT2 if $27 is equal to $0. |
| 20470001 | ADDI | $7, $2, 0x1 | Add immediate value 0x1 to $2, store result in $7. |
| ACF00000 | SW | $16, 0x0($7) | Store word from $16 into memory address 0x0 + content of $7. |
| 2042FFFF | ADDI | $2, $2, 0xFFFF | Add immediate value 0xFFFF to $2, store result in $2. |
| 08100009 | J | LOOP2 | Jump to the LOOP2 address. |
| 20470001 | ADDI | $7, $2, 0x1 | Add immediate value 0x1 to $2, store result in $7. |
| ACEA0000 | SW | $10, 0x0($7) | Store word from $10 into memory address 0x0 + content of $7. |
| 20210001 | ADDI | $1, $1, 0x1 | Add immediate value 0x1 to $1, store result in $1. |
| 0034E02A | SLT | $28, $1, $20 | Set $28 to 1 if $1 < $20, otherwise set it to 0. |

# APPENDIX B: Cycle accurate Simulator

## Cycle Accurate Simulator

The Cycle-Accurate Simulator (CAS) was developed as a specialized tool for verifying the design and functionality of a 5-stage pipelined MIPS processor. Unlike traditional simulation tools like ModelSim, which rely heavily on waveform-based debugging, the CAS provides a clock-by-clock visualization of the processor's internal state. This includes tracking data transfers between pipeline stages, instruction progression, and the values of key control signals, offering a clearer and more structured view of the processor's operation.

The primary motivation behind creating this simulator was to enhance the verification process by closely mimicking the hardware design implemented in Verilog. The CAS reproduces the behavior of the pipelined processor cycle-by-cycle, ensuring that the simulated execution aligns with the intended hardware implementation. This alignment allows engineers to quickly identify discrepancies between the hardware design and its expected behavior, particularly in scenarios involving data hazards, control hazards, or branch mispredictions.

By visualizing the data flow through the pipeline and the changes in pipeline registers at each clock cycle, the CAS provides insights that are difficult to extract from waveform simulations alone. This makes it easier to debug complex behaviors and validate the correctness of the design under various operating conditions. The simulator also allows users to verify the impact of design decisions, such as branch prediction strategies or hazard detection mechanisms, in a highly detailed and precise manner.

In summary, the CAS serves as a robust verification tool that complements traditional simulation workflows by providing a more intuitive, cycle-accurate representation of the hardware's operation.

**How to Operate**

1. **Download the Simulator**:
   Begin by downloading the zip folder containing the Cycle-Accurate Simulator (CAS).
2. **Open the Command Prompt**:
   Press **Windows + R** to open the *Run* dialog box. Type cmd and press **Enter** to launch the terminal.
3. **Navigate to the Simulator Path**:
   In the terminal, use the cd command to navigate to the folder where the simulator executable is located. For example:

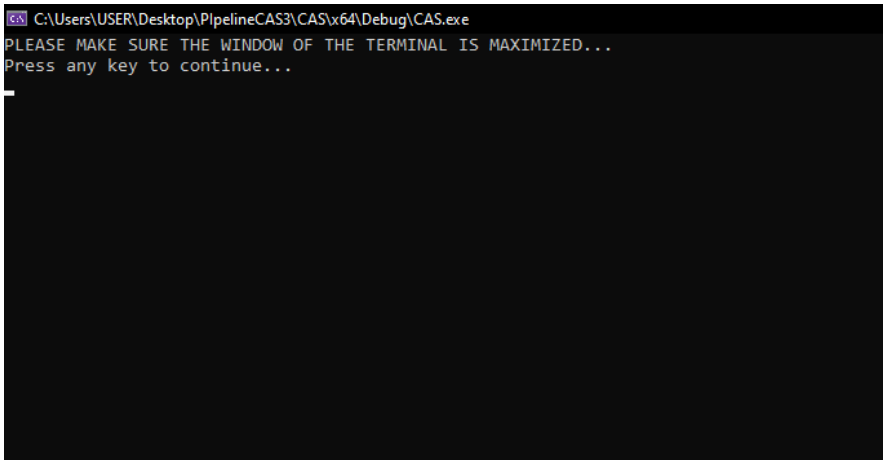
   cd path\to\simulator\folder

4. **Run the Simulator**:
   Once in the correct directory, type the following command to launch the simulator:

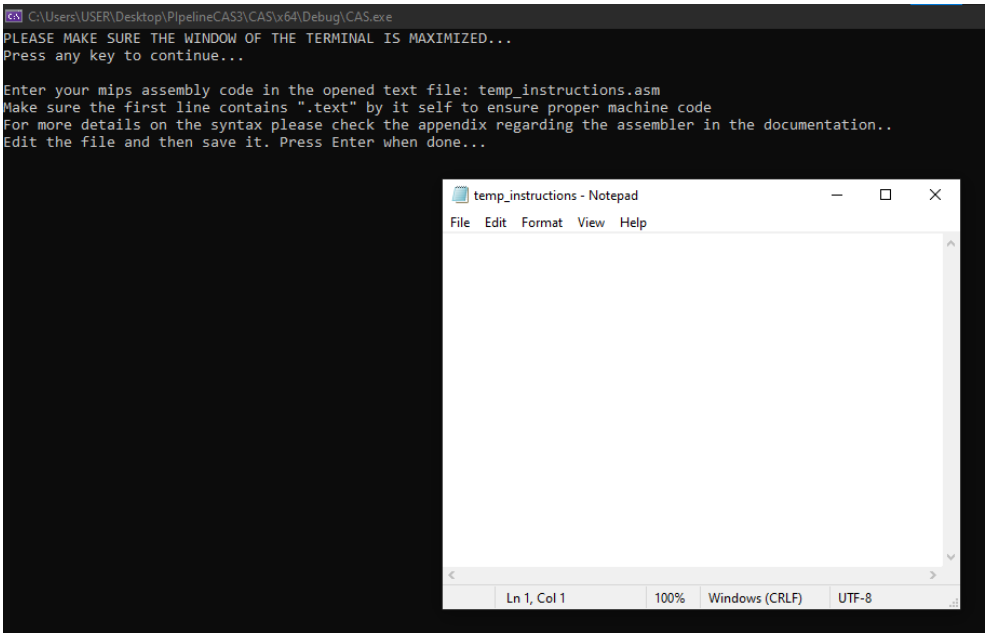
   CAS

5. The simulator will start, and you can follow the on-screen instructions to interact with it.

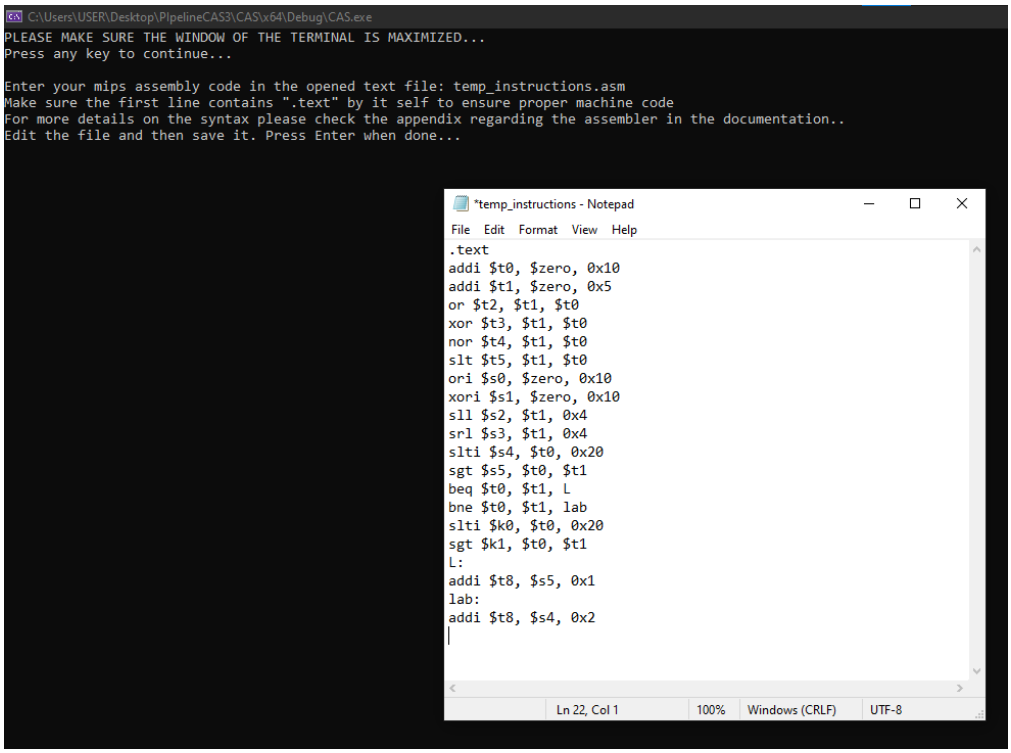
For correct screen printing make sure that you maximize the window of the terminal.




A temporary text file will appear on the screen, here you can input your assembly code



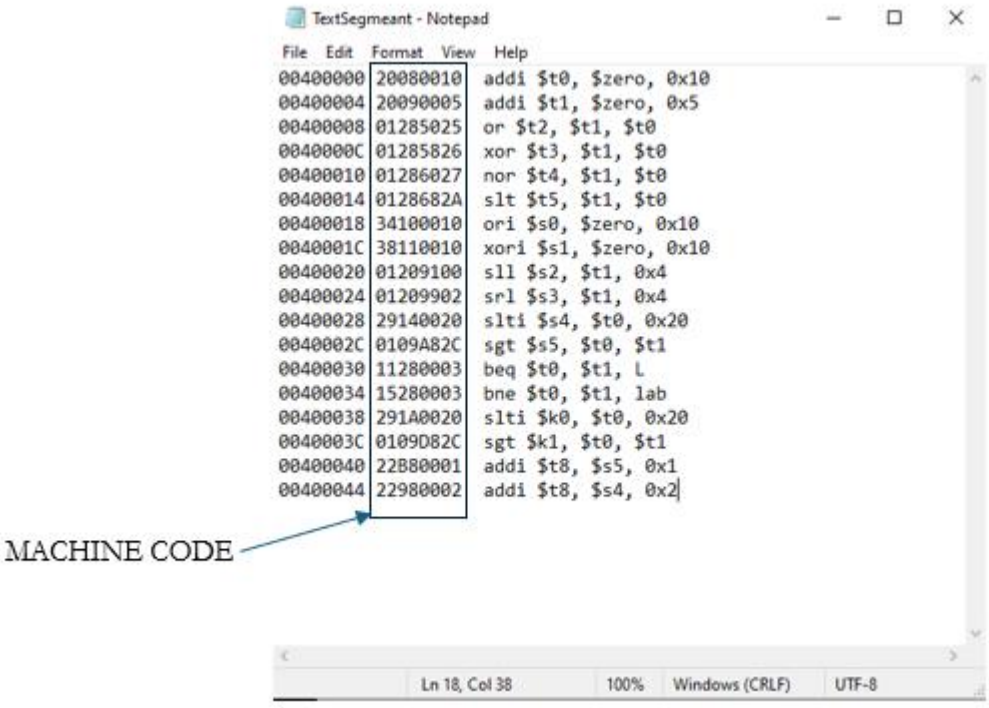
Please make sure that the file starts with .text as shown,




Refer to the assembler appendix for more details on the syntax.

When done save the file and press enter to activate the simulator

A file containing the instructions and their machine code will appear in the same directory as the executable for the Cycle accurate simulator.



**Memory Addressing in the Simulator**

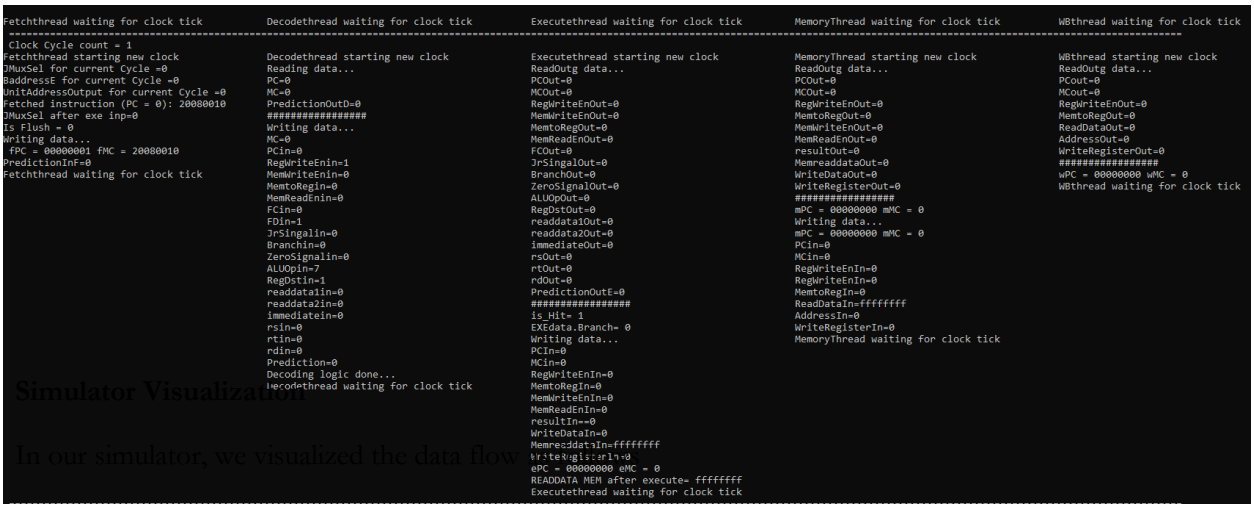
In the simulator, the values displayed on the left represent the addresses of the machine code in memory. However, it's important to note that these addresses are primarily for **visualization purposes** and do not reflect how memory addresses are actually indexed in the hardware or the CycleAccurateSimulator.

For comparison with tools like MARS, which use byte-addressable memory, these values are shown as byte addresses. In contrast, our hardware design and the cycle-accurate simulator both use **word-addressable memory**, starting at index 0. This means that in the actual design, each address corresponds to a word (typically 4 bytes), while the simulator's memory visualization shows addresses in byte units for easier alignment with MARS.

This distinction ensures that the design is properly validated, as we can directly compare our word-addressable memory implementation with the byte-addressable system in MARS, making sure our word alignment and addressing are correct.

**Simulator Visualization**

In our simulator, we visualized the data flow as follows




Simulator Visualization

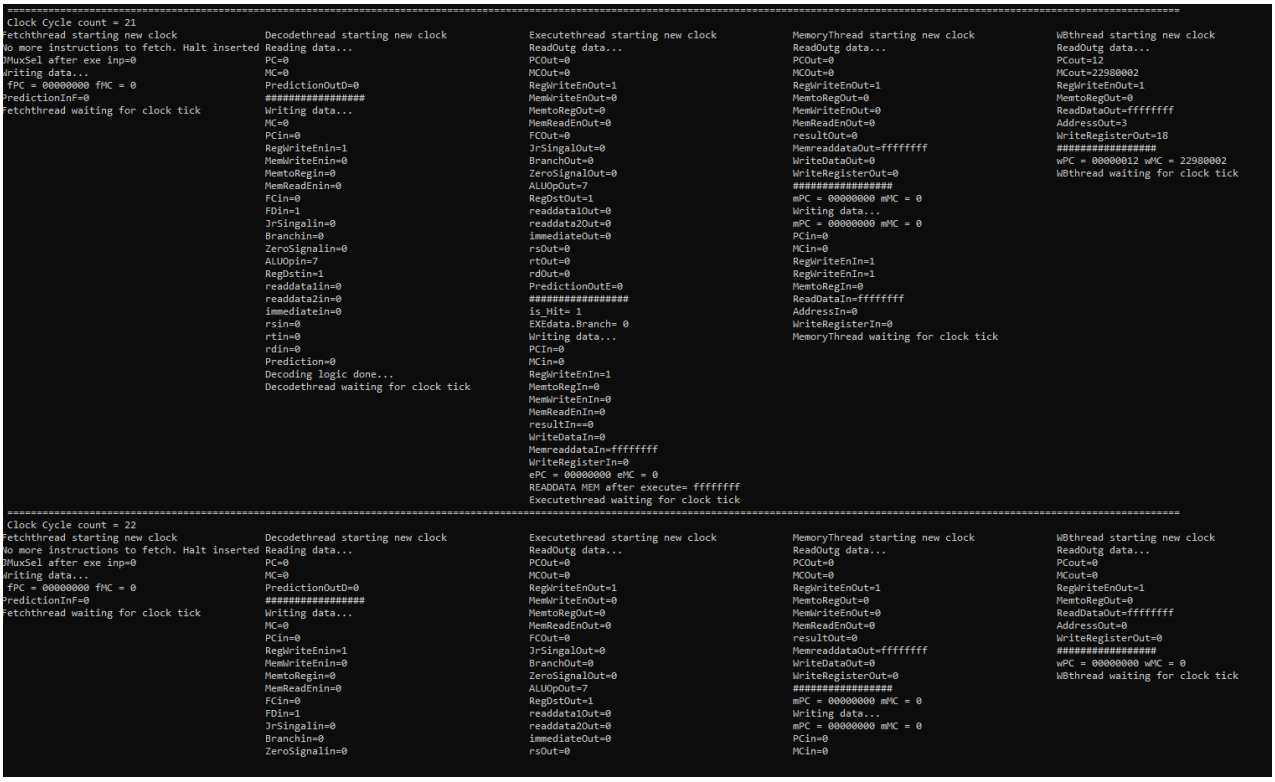
In our simulator, we visualized the data flow

Where each stage is represented by a column.

Each stage displays the values its reading from the previous pipe, and after it processes it displays what it is about to write to the next pipe in the next clock cycle.

Mind that the implicit write back stage here is shown in the simulator, but it behaves as expected and explain in the pipeline design chapter.

When the execution is done the simulator runs for some extra clock cycles to make sure that the pipeline is drained.

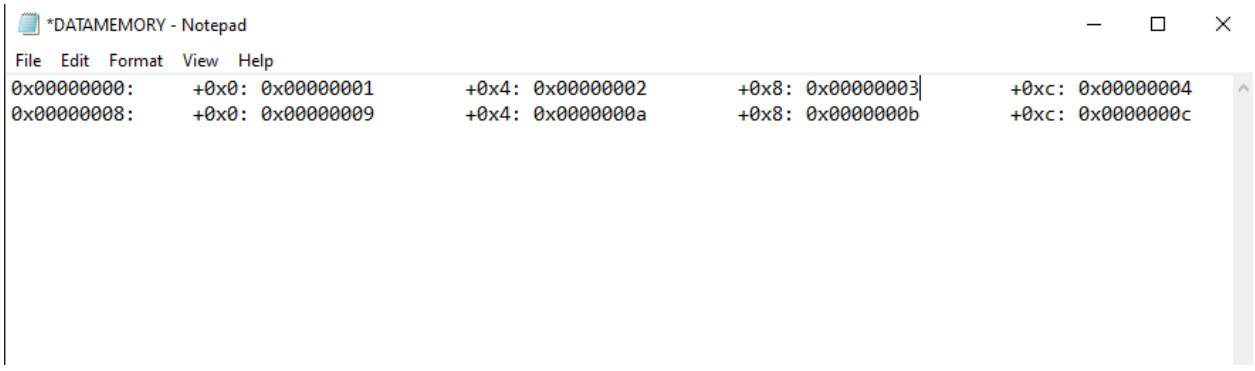


And when the execution of the given program is finished, we dumb the values of the register.

```
================================================================================
 Clock Cycle count = 24
Fetchthread starting new clock              Decodethread starting new clock
No more instructions to fetch. Halt inserted Reading data...
JMuxSel after exe inp=0                     PC=0
Writing data...                             MC=0
 fPC = 00000000 fMC = 0                     PredictionOutD=0
PredictionInF=0                             ################
Register File Contents:                     Writing data...
R0: 0                                       MC=0
R1: 0                                       PCin=0
R2: 0                                       RegWriteEnin=1
R3: 0                                       MemWriteEnin=0
R4: 0                                       MemtoRegin=0
R5: 0                                       MemReadEnin=0
R6: 0                                       FCin=0
R7: 0                                       FDin=1
R8: 10                                      JrSingalin=0
R9: 5                                       Branchin=0
R10: 15                                     ZeroSignalin=0
R11: 15                                     ALUOpin=7
R12: ffffffea                               RegDstin=1
R13: 1                                      readdata1in=0
R14: 0                                      readdata2in=0
R15: 0                                      immediatein=0
R16: 10                                     rsin=0
R17: 10                                     rtin=0
R18: 50                                     rdin=0
R19: 0                                      Prediction=0
R20: 1                                      Decoding logic done...
R21: 1
R22: 0
R23: 0
R24: 3
R25: 0
R26: 0
R27: 0
R28: 0
R29: 7ffffffc
R30: 0
R31: 0


C:\Users\USER\Desktop\PIpelineCAS3\CAS\x64\Debug\CAS.exe (process 34924) exited
To automatically close the console when debugging stops, enable Tools->Options-
Press any key to close this window . . .
```

If there was memory access in the program, a file containing the memory elements after execution

Our simulator is yet to support, memory initialization. The User must use SW instruction to store data in the memory.



# APPENDIX C: Assembler

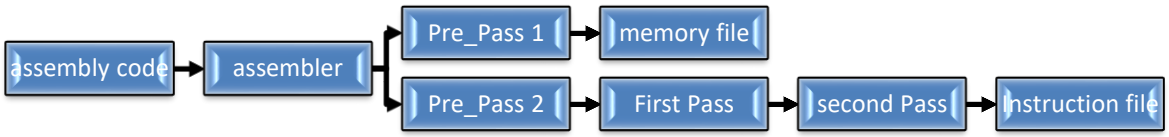
Introduction:

This section provides a technique that will make MIPS assembly language programming a relatively easy task to deal with hardware.

An assembler is a program that translates assembly instruction into machine code, which is a 32-bit binary format executable by the processor. The assembler converts MIPS assembly instructions into corresponding machine language instructions that the MIPS processor can execute.

our assembler uses multi pass to enhance functionality and provide additional features for managing data transfers in the assembler, while improving readability for programmers

### Diagram for assembly instructions




### Assembly code

To write assembly code, a programmer typically uses a dedicated section or workspace in a text editor or Integrated Development Environment (IDE).

This section allows the programmer to input human-readable assembly instructions, organized into segments like data definitions, code instructions, and labels. Once the assembly code is written, it is saved as a plain text file with an appropriate extension, such as .asm or .s, depending on the assembler being used.

This text file serves as the input for the assembler, which will process it to generate machine code. Writing assembly code into a structured text file ensures it can be easily edited, shared, and reused in different projects.

### Pre_Pass1

The Pre_Pass1 function processes the .data section of an assembly file and writes the extracted data to a file text, along with their corresponding memory addresses. It ensures that data memory mapping is properly structured and clear for subsequent stages of the assembler and set data into (memory data in design) to use it later by instruction.

To write a data section in an assembly file, follow a structured approach to define and organize variables and their corresponding data. The (.data) is used to declare and initialize data items that will be stored in the program's data segment.

.data: use to declare and initialize data section
Var-name : name pointer refer to data
.size : set size for data (Byte or Word)
$data_{1...n}$ :value of data to store into memory

```
.data       // refer to start data section

//define data in design

Var-name:  .size  data1,data2,data3, ….. ,datan
```

Note: Data section defines at the top of text file before start instruction section.

### Pre_Pass2

The Pre_Pass1 function has many functions in assembler, but this has a main function to deal with pseudo instruction (bltz, bgez).

The function starts by loading the file into memory and locating the .text section, which signifies the beginning of the instruction segment and defines the area designated for code.

Note: If the .text segment is missing or file operations fail, appropriate error messages are displayed.

```
.text     // refer to start instruction section
```

It skips comments or lines starting with (// or #) and ensures that only valid code lines are processed.

The function replaces specific pseudo-instructions such as nop, bltz, and bgez with equivalent low-level instructions.

Additionally, the function validates opcodes to ensure only permitted instructions are used. If an invalid opcode is encountered, an error is displayed, and the program exit.

## Pseudo-instructions

Pseudo-instructions are legal MIPS assembly language instructions that do not have a direct hardware implementation. They are provided as a convenience for the programmer. When you use pseudo-instructions in a MIPS assembly language program, the assembler translates them into equivalent real MIPS instructions. Here is a list of the commonly used pseudo-instructions.

| *Task* | *Pseudo-instructions* | *Programmer Writes* | *Assembler Translates To* |
|---|---|---|---|
| *instruction that achieves the desired "do nothing" effect.* | NOP | NOP | sll $0, $0, 0x0 |
| *If (r1 < 0), branch to label* | bltz <r1>, <label> | bltz $t0, function | slt $27, $t0 , $zero<br>bne $27, $zero, function |
| *If (r1 >= 0), branch to label* | bgez <r1>, <label> | bgez $t0, function | slt $27, $t0 , $zero<br>beq $27, $zero, function |

## First Pass

The First Pass function is a critical step in the assembler's process. Its primary goal is to parse the assembly source code and build a mapping of labels to their corresponding memory addresses. This label map is later used during the second pass to resolve symbolic references in the code.

During this pass, the assembler focuses on creating a symbol table, which is a list of labels (e.g., variable names, function names, etc.) and their corresponding memory addresses. The assembler doesn't generate the actual machine code during this phase. Instead, it scans the code to identify all labels and instructions and assigns memory locations to variables and instructions.

The function continues processing lines in the .text section until the end of the file.

Example:

```
.text
start:
add $t0, $t1, $t2
sub $t3, $t4, $t5
loop:
j start
nop
```

Result:

label Map:

- `start`: 0x00000000

loop: 0x00000008

current Address: 0x00000010 (after processing all instructions)

### Second Pass

The second pass in an assembler is where the actual machine code is generated from the assembly language program. After the first pass has created the symbol table and determined the memory addresses for labels and variables, the second pass focuses on translating the assembly instructions into their corresponding machine code.

During this phase, the assembler uses the information gathered in the first pass, such as the addresses for labels and variables, to replace symbolic names with their actual memory addresses in the instruction set. It also generates the binary code for each instruction and resolves any unresolved references, such as jump addresses or function calls. The result of the second pass is a complete machine code program that can be executed by the computer's processor. In summary, the second pass finalizes the translation process, converting the symbolic assembly code into a runnable binary format.

# APPENDIX D: FPGA

Field-Programmable Gate Arrays (FPGAs) are versatile and reconfigurable hardware devices that play a pivotal role in modern electronics and computing. Unlike traditional application-specific integrated circuits (ASICs), which are fixed in their functionality once manufactured, FPGAs offer a flexible architecture that can be programmed and reprogrammed to perform various tasks. This adaptability makes them ideal for a wide range of applications, from prototyping and research to deployment in systems requiring high-performance computing, signal processing, or real-time operations.

FPGAs are increasingly used in the development and verification of processors, where their parallel processing capabilities and reconfigurability make them invaluable tools. Their ability to emulate processor designs in real-time allows for thorough testing and debugging of complex architectures and algorithms. FPGAs enable rapid prototyping, efficient validation of design changes, and high-speed simulation, all of which are critical for ensuring the accuracy and reliability of modern processors.

An FPGA (Field-Programmable Gate Array) is composed of several key components that enable its reconfigurable and high-performance functionality. At its core are the Configurable Logic Blocks (CLBs) or Logic Array Blocks (LABs), which are used to implement combinational and sequential logic.

A Look-Up Table (LUT) is a fundamental building block in FPGA design, used to implement combinational logic in a highly efficient manner. In Verilog, an LUT is typically represented by a case statement or a set of logic expressions that define the output based on specific input combinations. An LUT stores precomputed output values for all possible input combinations, allowing it to produce results almost instantaneously during operation. For example, in an FPGA, an LUT with 4 inputs can represent any Boolean function of those inputs by storing 16 possible output values in its memory. This concept can be modeled in Verilog using a case or always block to mimic the behavior of the LUT. Understanding how to design and utilize LUTs in Verilog is essential for creating efficient and optimized digital circuits in FPGA development.

## 4. Testing In FPGA

Decode is used to convert binary to hexadecimal, and decoding helps to display numbers on a 7-segment display.

Decoder:

```
module decoder_4_16(in, out);
        input [3:0] in;
        output reg [6:0] out;
    always @ (*) begin
        case (in)
          4'h0: out = 7'b0000001;   // 0
          4'h1: out = 7'b1001111;   // 1
          4'h2: out = 7'b0010010;   // 2
          4'h3: out = 7'b0000110;   // 3
          4'h4: out = 7'b1001100;   // 4
          4'h5: out = 7'b0100100;   // 5
          4'h6: out = 7'b0100000;   // 6
          4'h7: out = 7'b0001111;   // 7
          4'h8: out = 7'b0000000;   // 8
          4'h9: out = 7'b0000100;   // 9
          4'hA: out = 7'b0001000;   // A
          4'hB: out = 7'b1100000;   // B
          4'hC: out = 7'b0110001;   // C
          4'hD: out = 7'b1000010;   // D
          4'hE: out = 7'b0110000;   // E
          4'hF: out = 7'b0111000;   // F
          default: out = 7'b1111111; // Off
        endcase
    end
end module
```

## 1. Pipeline Test – FPGA

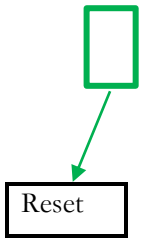
```
//Test-Code
Addi $t0, $zero, 10
Addi $t1, $zero, 5
or $t2, $t1, $t0
xor $t3, $t1, $t0
nor $t4, $t1, $t0
slt $t5, $t1, $t0

ori $s0, $zero, 10
xori $s1, $zero, 10
sll $s2, $t1, 4
srl $s3, $t1, 4
slti $s4, $t0, 20
sgt $s5, $t0, $t1
```

```
//complete
slti $s4, $t0, 20
sgt $s5, $t0, $t1

beq $t0, $t1, L
bne $t0, $t1, lab
slti $k0, $t0, 20
sgt $k1, $t0, $t1

L:
add $t8, $s5, 1
lab:
add $t8, $s4, 2
```

1) Reset Design:

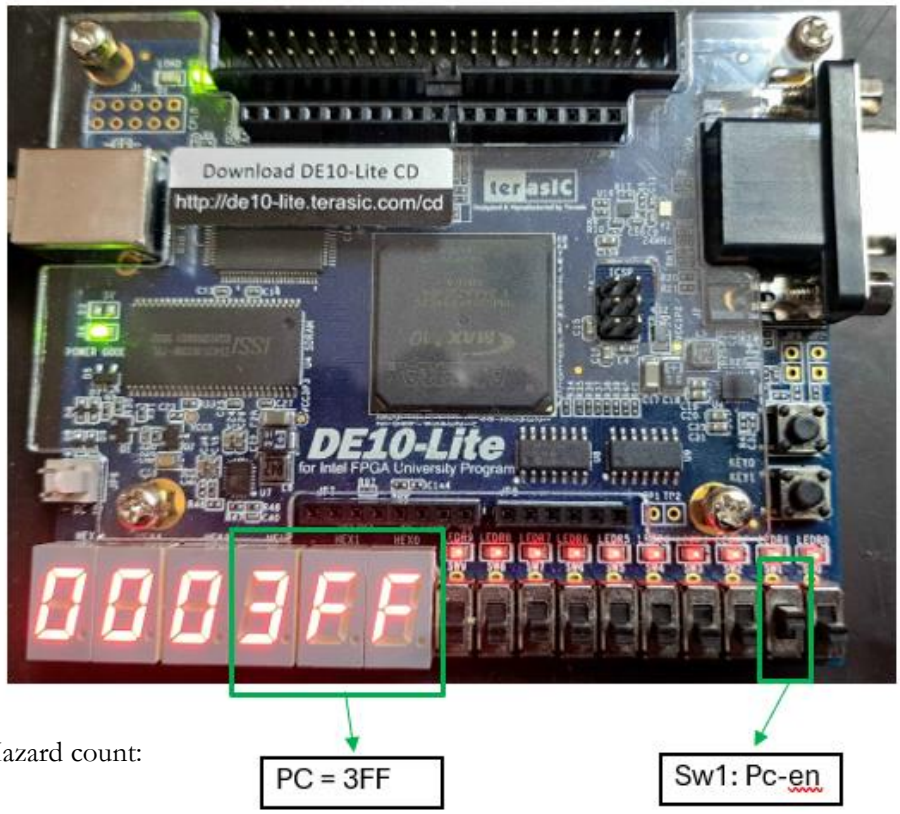


//Use switch 0 in FPGA to reset Design




2) Program Counter:

//Using Switch 1 To display value of Program counter on 7-Segment Display

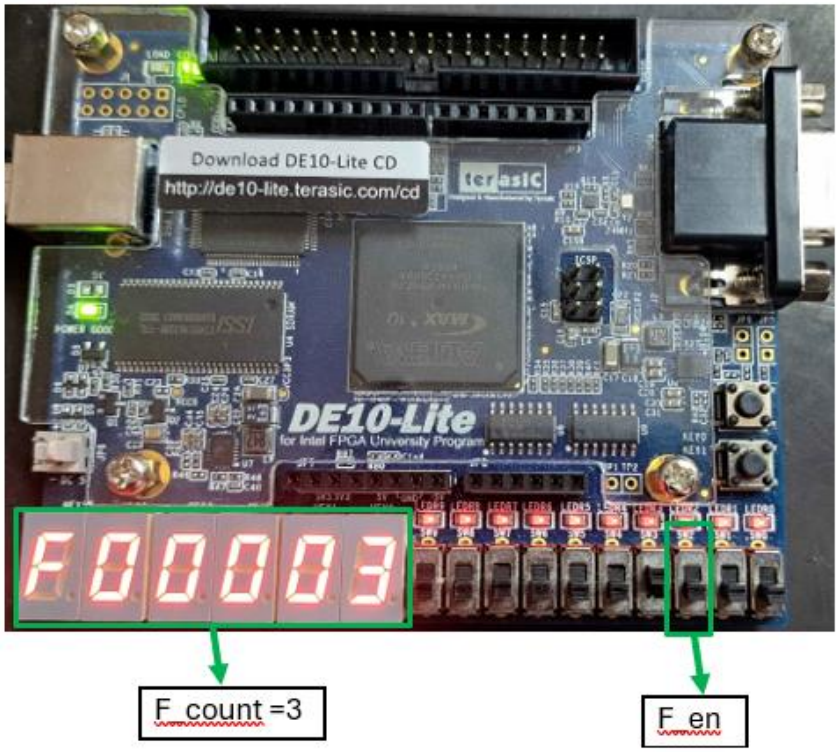



3) Hazard count:

// Using Switch 2 to display how many control hazards exist in my design on a 7-segment display.


4) Forwarding count:

// Using Switch 3 to display how many Forwarding exist in my design on a 7-segment display.



5) Branch Prediction (Miss/Hit):

// Using Switch 4 to display how many Hit and miss from Branch-Prediction in my design on a 7-segment display.