

MULTI CYCLE PIPELINED RISCv BASED PROCESSOR

TEAM 8

**Multi Cycle Pipelined RISCv Based Processor Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Document Version** | **Remarks** | **Drafted by** |
| 10- September- 2024 | Version 1.0 | Micro Architecture Overview | TEAM 8 |
| 11-September-2024 | Version 2.0 | Micro Architecture Updated | TEAM 8 |
| 3-October-2024 | Version 3.0 | Updated With Results and Waveforms | TEAM 8 |

**MULTI CYCLE PIPELINED RISCv BASED PROCESSOR**

Table of Contents

[CHAPTER 1 – MICRO ARCHITECTURE 4](#_Toc177048218)

[1.1 MULTI CYCLE PIPELINED RISCv BASED PROCESSOR 4](#_Toc177048219)

[1.2 OBJECTIVES 4](#_Toc177048220)

[1.3 INSTRUCTIONS FORMAT 4](#_Toc177048221)

[1.4 PROJECT OVERVIEW 5](#_Toc177048222)

[1.5 Micro Architecture 5](#_Toc177048223)

[1.6 ARCHITECTURE COMPONENTS 6](#_Toc177048224)

[1.6.1 Program counter: 6](#_Toc177048225)

[1.6.2 Program memory: 7](#_Toc177048226)

[1.6.3 Register Set: 7](#_Toc177048227)

[1.6.4 ALU: 8](#_Toc177048228)

[1.6.5 Control Unit: 9](#_Toc177048229)

[1.7 OPERATIONS: 9](#_Toc177048230)

[1.7.1 Load Immediate Operation: 9](#_Toc177048231)

[1.7.2 ALU Operation: 10](#_Toc177048232)

[1.8 RESULT AND WAVEFORM: 10](#_Toc177048233)

[1.8.1 LOAD IMMEDIATE OPERATION: 10](#_Toc177048234)

[1.8.2 ADD & SUB OPERATION: 11](#_Toc177048235)

[1.8.3 AND & OR OPERATION: 11](#_Toc177048236)

[1.8.4 XOR OPERATION: 12](#_Toc177048237)

# CHAPTER 1 – MICRO ARCHITECTURE

## 1.1 MULTI CYCLE PIPELINED RISCv BASED PROCESSOR

A multi-cycle pipelined RISC-V based processor is a processor that leverages both multi-cycle execution and pipelining to enhance performance. Based on the RISC-V architecture, known for its simplicity and modularity, this processor divides the execution of each instruction into multiple clock cycles, while also using a pipeline to overlap the execution of multiple instructions. This combination improves throughput by processing several instructions simultaneously at different stages, balancing hardware complexity and efficiency. However, it also requires complex control logic to manage hazards and optimize pipeline performance. These processors are well-suited for applications like embedded systems, IoT devices, and general-purpose computing, where a balance of performance and cost is essential.

## 1.2 OBJECTIVES

Develop a multi-cycle 5 stage pipelined processor

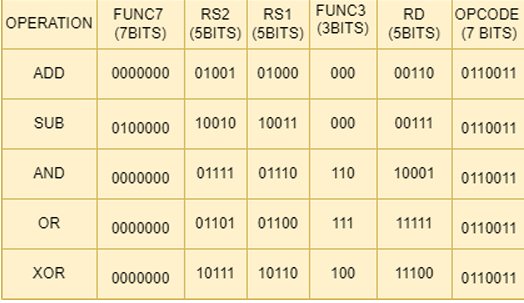
Implement and verify the functionality of below ALU operations

* ADD
* SUB
* AND
* OR
* XOR
* NOP

For each operation implement the corresponding instruction of RISCv

Also write the testbench to verify the working of the processor

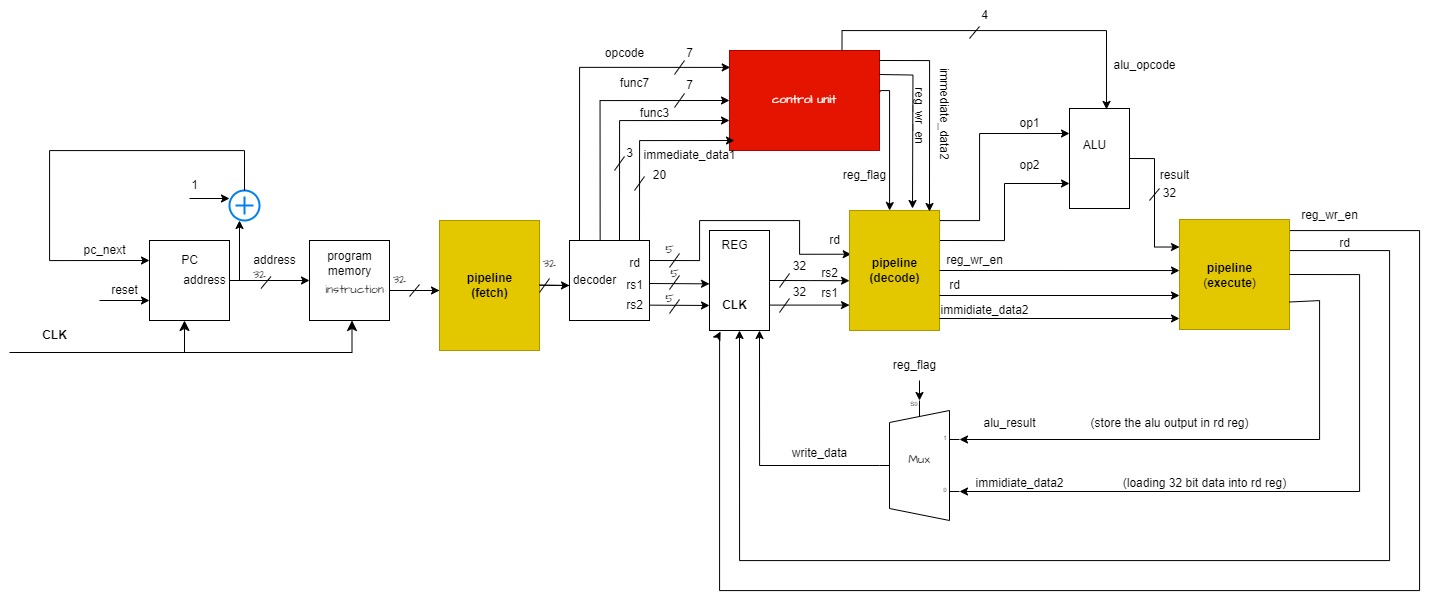
## 1.3 INSTRUCTIONS FORMAT



## 1.4 PROJECT OVERVIEW

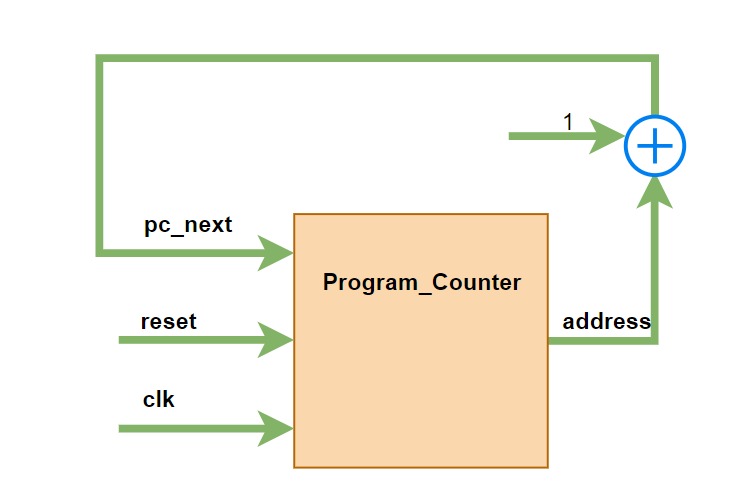
* Establishing the project's functional components and microarchitecture is designing micro-architecture.
* By utilizing the appropriate RISC-V instructions, carry out the ADD, SUB, AND, OR, XOR, and NOP algorithms.
* To confirm the processor's and each ALU operation's functionality, build a test bench.
* Construct a five-stage, multi-cycle pipelined processor design.
* Phases of a pipeline Five steps make up the processor pipeline: Write Back (WB), Memory Access (MEM), Execute (EX), Instruction Fetch (IF), and Instruction Decode (ID).
* creating multiplexers, ALUs, program counters, program memory, instruction decoders, and multiplexers, then using Verilog to create the HDL codes for those stages.

## 1.5 Micro Architecture



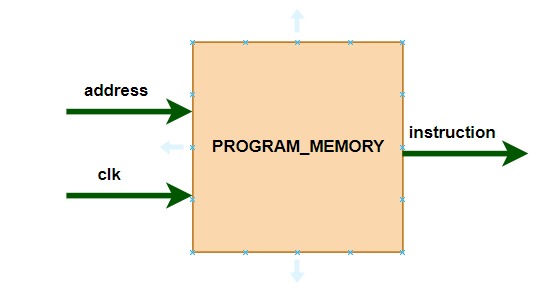
## 1.6 ARCHITECTURE COMPONENTS

### 1.6.1 Program counter:



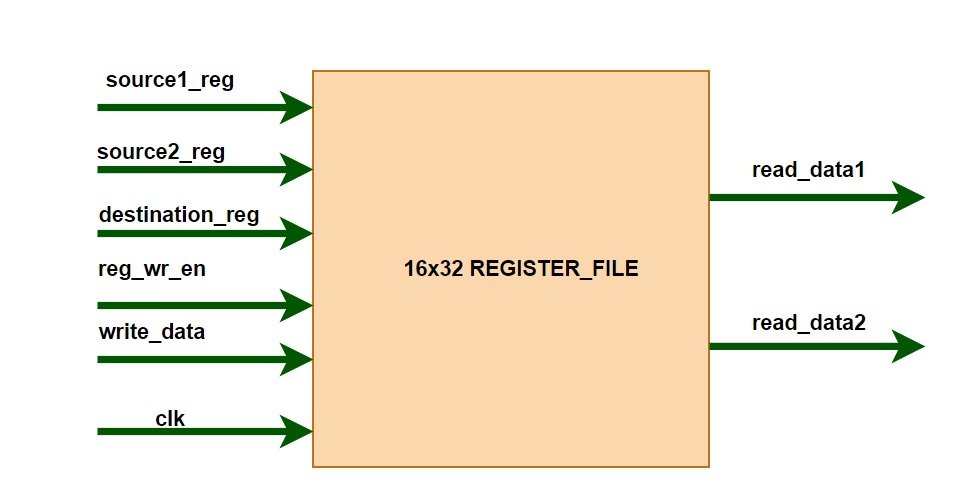
* The PC, or program counter, is a key CPU register that keeps track of the order in which instructions are executed.
* The PC is in sync with the system clock and stores the memory address of the subsequent instruction that needs to be retrieved.
* Every time there is a clock pulse, the CPU retrieves the instruction from the PC's address and advances the PC to the subsequent instruction.
* Through this procedure, the CPU is guaranteed to carry out instructions in the proper order, facilitating seamless and effective system operation.

### 1.6.2 Program memory:



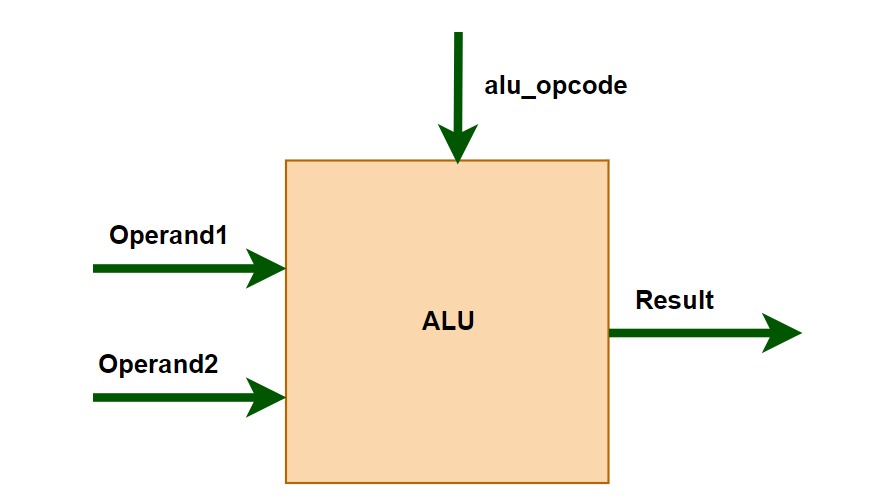
* The instructions that the CPU needs to carry out are kept in the program memory.
* The next instruction is fetched and transmitted to the control unit and decoder after the Program Counter (PC) locates its precise memory address.
* By disassembling the instruction into its constituent parts, such as register addresses, opcodes, and functions, the decoder is able to understand it. In order to guarantee that the command is carried out properly, the control unit subsequently routes this data to the relevant CPU components. Through the translation and orderly execution of instructions, this process makes it possible for the CPU to function smoothly.

### 1.6.3 Register Set:



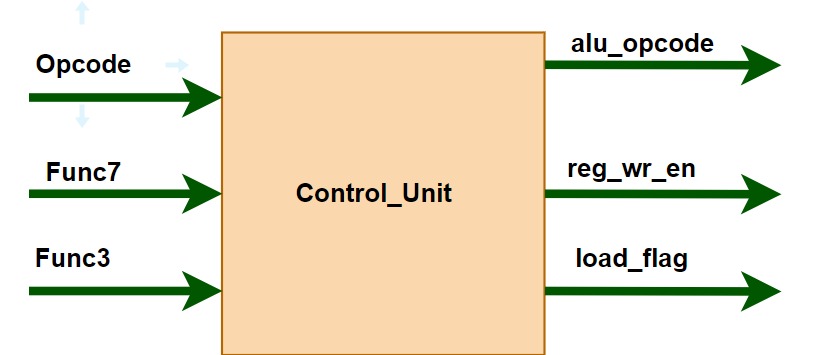
* To store the values or data needed for the ALU function or other processes, the register set consists of 16 registers with 32 bits each.

### 1.6.4 ALU:



* The Arithmetic Logic Unit (ALU) of a CPU is where all arithmetic and logic operations are carried out.
* The resultant data must be stored for later use after these processes are completed.
* Through a procedure known as "write-back," the outcome is restored into the register set.
* The CPU's registers are quick, compact storage spaces that momentarily store data.
* The CPU can swiftly retrieve the data for next operations by storing the results in registers, guaranteeing effective instruction processing and execution.

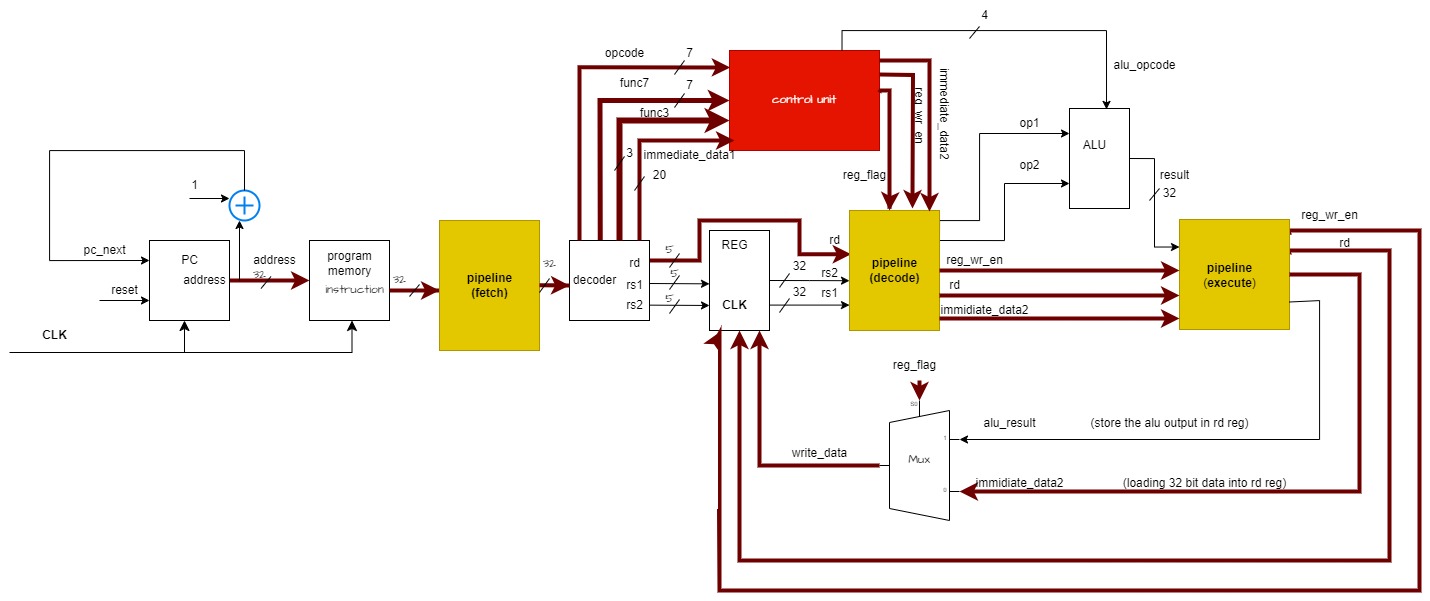
### 1.6.5 Control Unit:



* The control unit takes three inputs: the opcode, func7, and func3 fields from the instruction. Based on these inputs, it generates control signals that dictate the operation of various components like the ALU, register file, and memory access. The outputs from the control unit are used to control data flow, select operations, and manage instruction execution in the processor pipeline.

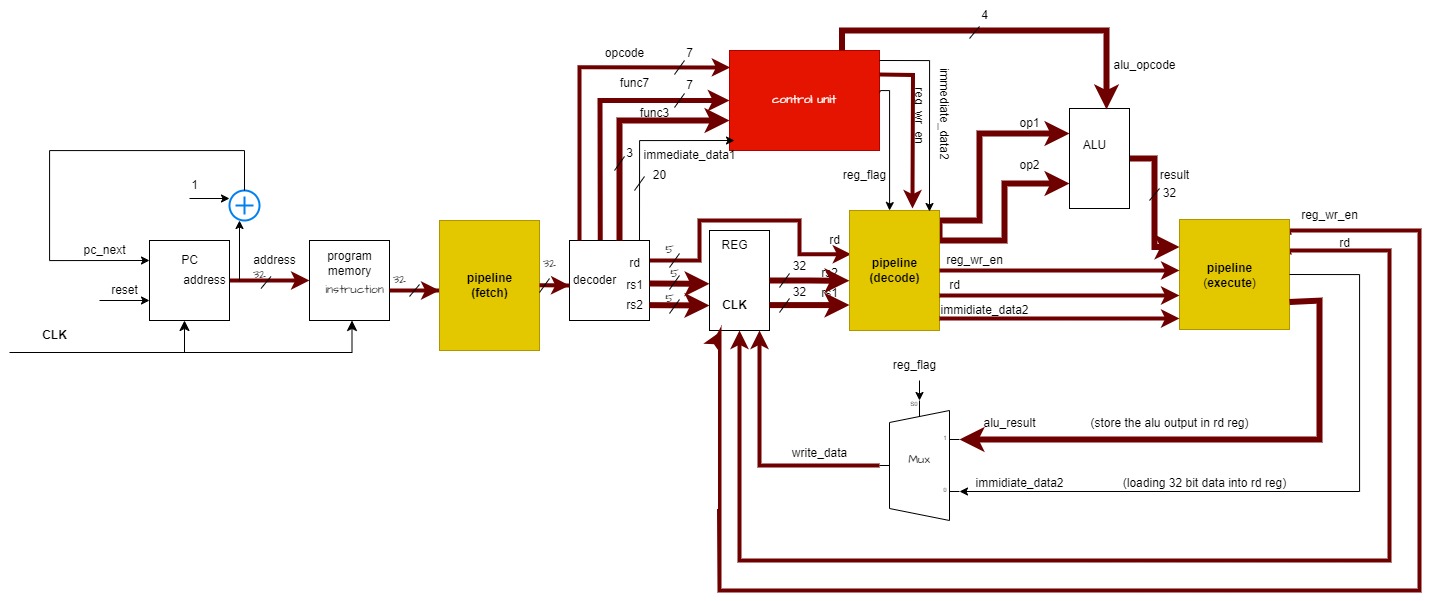
## 1.7 OPERATIONS:

### 1.7.1 Load Immediate Operation:

****

For the Load Immediate operation, the instruction's opcode is sent to the control unit, which generates signals to control the data flow. It tells the multiplexer (MUX) to select the immediate value from the instruction and enables the Register Write signal to allow data to **be written into** the register file. The 32-bit immediate value is then passed into the specified register. In data flow diagrams, this process is often shown with a green line, representing the path the immediate value takes from the instruction to the register.

### 1.7.2 ALU Operation:

****

For an ALU operation, data is first read from the source registers in the register file. The ALU performs the required operation based on the alu\_opcode provided by the control unit, which determines the type of operation (such as addition, subtraction, etc.). After the ALU completes the computation, the result is written back into the destination register specified by the instruction.

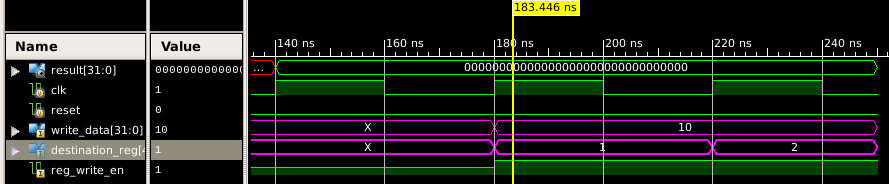
## 1.8 RESULT AND WAVEFORM:

### 1.8.1 LOAD IMMEDIATE OPERATION:

**Instruction:**

memory[0] <= 32'b00000000000000001010\_00001\_0000011;// LOAD\_IMM R1, 10

memory[1] <= 32'b00000000000000001010\_00010\_0000011;// LOAD\_IMM R2, 10



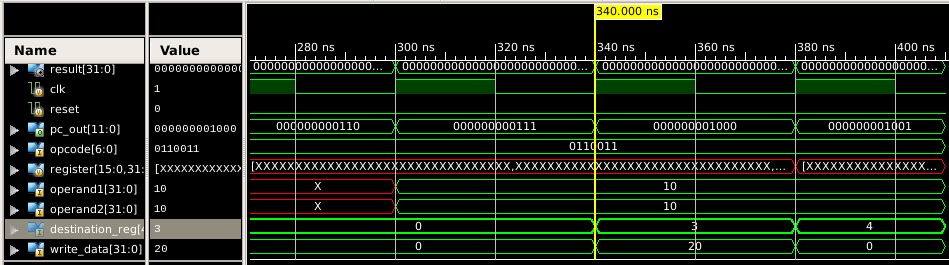
# The Immediate value 10 is stored in R1 and R2

### 1.8.2 ADD & SUB OPERATION:

**Instruction:**

memory[4] <= 32'b0000000\_00001\_00010\_000\_00011\_0110011; // ADD R3, R1, R2

memory[5] <= 32'b0100000\_00001\_00010\_000\_00100\_0110011; //SUB R4 ,R1,R2



# The addition result is stored in the specified destination register.

# In the above waveform , operand1 and operand 2 indicates the 2 operand for the ADD and SUB operation

# Write data is the result for the particular operation

### 1.8.3 AND & OR OPERATION:

**Instruction:**

memory[6] <= 32'b0000000\_00001\_00010\_110\_00101\_0110011; //and R5, R1,R2

memory[7] <= 32'b0000000\_00001\_00010\_111\_00110\_0110011; //or R6, R1,R2

# The AND and OR operation result is stored in the specified destination register.

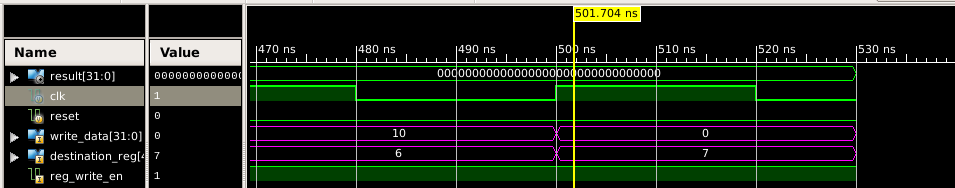
# In the above waveform , operand1 and operand 2 indicates the 2 operand for the AND and OR operation

# Write data is the result for the particular operation

### 1.8.4 XOR OPERATION:

**Instruction:**

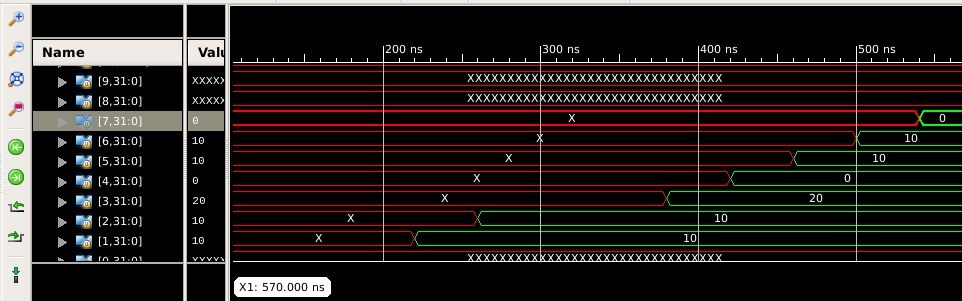
memory[8] <= 32'b0000000\_00001\_00010\_100\_00111\_0110011; //xor R7,R1,R2



# # The XOR operation result is stored in the specified destination register.

# Write data is the result for the particular operation

**Values Stored in The Specified Destination Register:**



* R1 = 10 ,R2 = 10
* R3 = R1 + R2 , result is 20 and stored in R3
* R4 = R1 - R2, result is 0 and stored in R4
* R5 = R1 & R2, result is 10 and stored in R5
* R6 = R1 | R2, result is 10 and stored in R6
* R7 = R1 ^R2,result is 0 and stored in R7