PROJECT REPORT

ON

RISC-V RV32I RTL DESIGN

NAME: AKLESH ANUBHAB

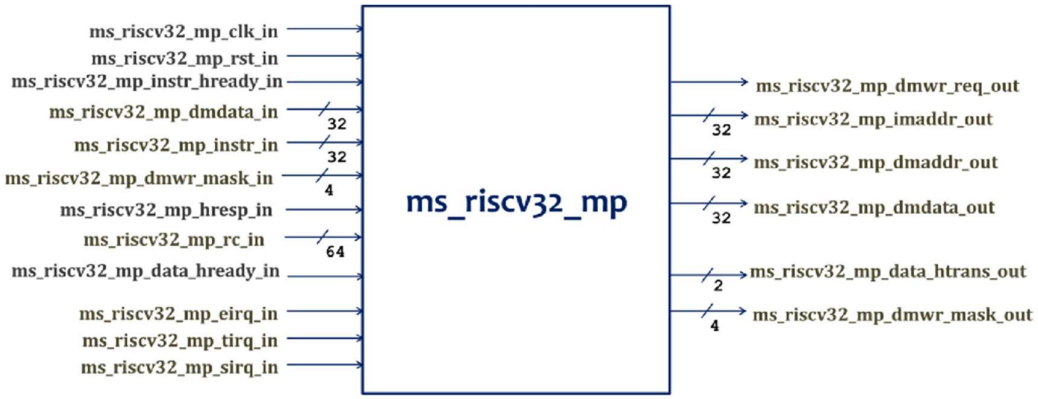
REG NUMBER: 21BEC0405

3RD YEAR BTECH, ECE

VIT VELLORE

TOP MODULE:

BLOCK DIAGRAM



FUNCTIONALITY:

The term "top-level module" usually refers to the highest level of abstraction in the hardware design, which includes the complete processor or CPU core, in the context of RISC-V 32I RTL (Register-Transfer Level) design. The 32-bit instruction set architecture known as RISC-V 32I is implemented by processor cores using RTL design, which is a low-level hardware description.

The following features are present in the top-level module of an RTL design for a RISC-V 32I CPU core:

1. **Programme counter (PC) value-based instruction fetching (IF):** This stage retrieves instructions from memory (usually an instruction cache or memory).
2. **Instruction Decode (ID) Stage**: Here, the fetched instruction is decoded to identify the operation and operands that need to be carried out.
3. **Register File**: The general-purpose registers of the CPU are kept in a register file that is part of the top-level module. Reading operands and writing results are done via the register file.
4. **Execution Units**: Various sorts of instructions can be carried out by different types of execution units, depending on the pipeline design. Examples of these units include load and store operations and ALUs (Arithmetic Logic Units) for integer arithmetic operations.
5. **Control Logic**: Based on the instruction that is currently being processed, the top-level module's control logic creates control signals for different pipeline segments, enabling or disabling particular processes.

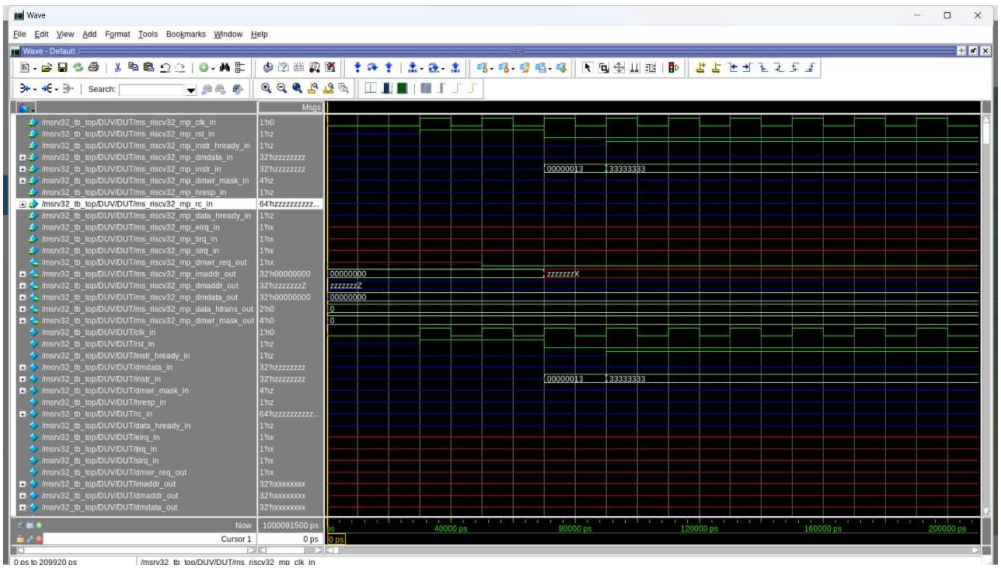
6. **Data Memory**: The top-level module integrates data memory or data cache if it is included in your design for load and store activities.
7. **Hazard Detection and Forwarding**: The top-level module has hazard detection and forwarding logic to handle data risks and control hazards. This ensures that instructions are executed correctly in a pipelined manner.
8. **Branch Prediction**: By speculatively executing instructions based on branch predictions, certain CPU cores may have branch prediction logic to enhance instruction performance.
9. **Handling Exceptions and Interrupts**: The top-level module has methods for dealing with exceptions and interrupts, such as shifting the control flow to the relevant handler code.
10. **Update of the Programme Counter (PC)**: To guarantee that the following instruction is executed correctly, the top-level module is in charge of updating the PC.
11. **Stages of the Pipeline**: The IF, ID, Execute, Memory, and Write-Back stages are all included in the top-level module.
12. **State Registers**: To keep track of the CPU's state, the top-level module has a number of state registers, including those for the programme counter (PC), instruction fetch and decode states, and other pertinent state data.
13. **Support for Debugging and Tracing**: To help with software development and debugging, certain top-level modules may have support for debugging and tracing features.

The top-level module controls the several parts needed to execute RISC-V 32I instructions and arranges the flow of instructions via the pipeline. It guarantees that commands are appropriately fetched, decoded, executed, and retired while managing risks and exceptions in accordance with the architecture's requirements. Depending on the design decisions taken for the pipeline architecture and processor core, the particular implementation details might change.

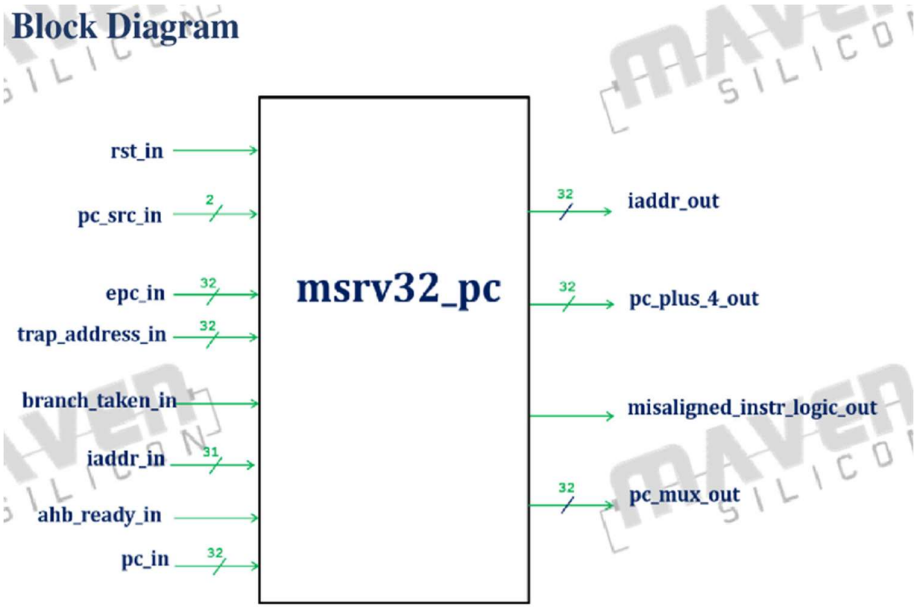
OUTPUT :

```
#
#  the following list of command are working fine
#       '{}
#  and test result is
#
#
# ------------------------------------------
# 8888888888      d8888 8888888 888
# 888            d88888   888   888
# 888           d88P888   888   888
# 8888888      d88P 888   888   888
# 888         d88P  888   888   888
# 888        d88P   888   888   888
# 888       d8888888888   888   888
# 888      d88P     888 8888888 88888888
# ------------------------------------------
#
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 1000091500: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' pha
se
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :53004
# UVM_WARNING :    1
# UVM_ERROR : 7004
# UVM_FATAL :    0
# ** Report counts by id
# [IN AGENT TOP]     1
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [UVMTOP]     1
# [r_type_sequence] 10000
# [uvm_sequence_base]     1
# [uvm_test_top.env.instr_agt_for_duv.drv]     1
# [uvm_test_top.env.instr_agt_for_duv.seqr.r_type_seq1] 50001
ort MobaXterm by subscribing to the professional edition here:  https://mobaxterm.mobatek.net
```

WAVEFORM:



PC MUX:



FUNCTIONALITY:

The PC (Programme Counter) MUX, or PC multiplexer, is a crucial part in an RTL (Register-Transfer Level) design for a RISC-V 32I processor core. It is in charge of choosing the programme counter's subsequent value. The following is the PC MUX's functionality:

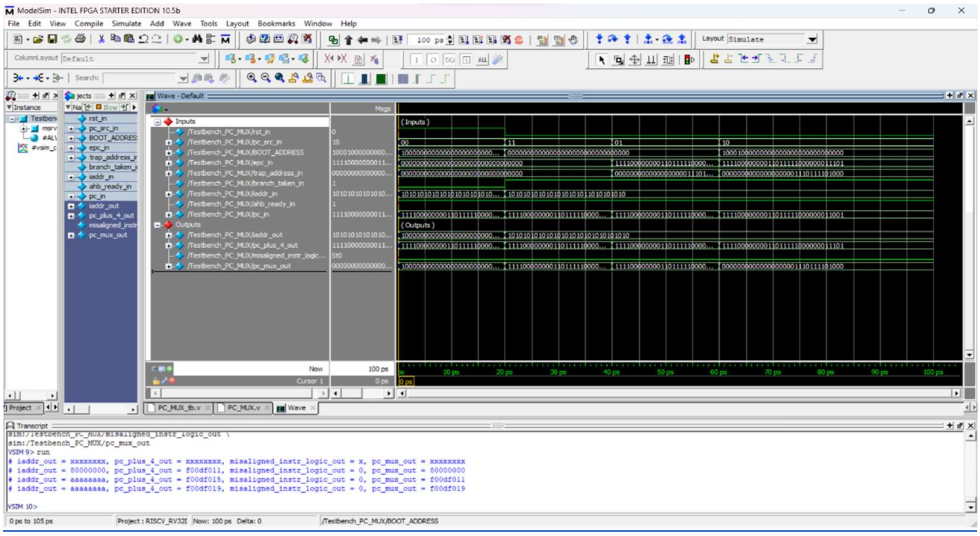
1. **Control Flow Control**: As the PC MUX determines the address of the next instruction to be fetched and executed, it plays a crucial role in managing the flow of instructions through the CPU.
2. **Branch Instructions**: The PC MUX is utilised by the processor to choose the target address for a branch when it encounters one. This address can be either the address of the subsequent sequential instruction or a computed branch target. Based on the outcome of the branch instruction's evaluation, the choice of branch target is chosen.
3. **Jump Instructions**: The PC MUX is in charge of choosing the target address given by the instruction for jump and jump-and-link instructions (like jal).
4. **Exception Handling**: The PC MUX can choose an interrupt service routine address or an exception handler address in the event of an exception or interrupt, enabling the processor to diverge from the intended programme flow.
5. **Predictive Branch Target**: The PC MUX may additionally select the predicted branch target over the standard sequential instruction address in pipelines that use branch

prediction logic. By executing instructions speculatively, this helps to improve performance and decrease pipeline stalls.

6. **Sequential Execution**: To maintain a constant flow of instructions through the pipeline, the PC MUX chooses the PC value of the subsequent sequential instruction when no branches or jumps are made.
7. **Conditional Branch Evaluation**: To ascertain whether a branch is taken or not, the PC MUX gets data from the branch instruction's execution stage. This information is used to determine which PC value is appropriate.
8. **Jump and Link Behaviour**: The PC MUX must record the return address (the address of the instruction that comes after a jump-and-link instruction, such as jal) and transmit it to the return address register (such as ra) in the register file when dealing with jump-and-link instructions.
9. **Exception Handling Behaviour**: The PC MUX is required to choose the address of the next instruction to be executed after the exception has been handled, or the exception handler address, in the event of exceptions or interrupts.
10. **Branch Prediction Resolution**: The PC MUX may also consider the prediction resolution in systems that have branch prediction units, and choose the appropriate target in accordance with the prediction result.

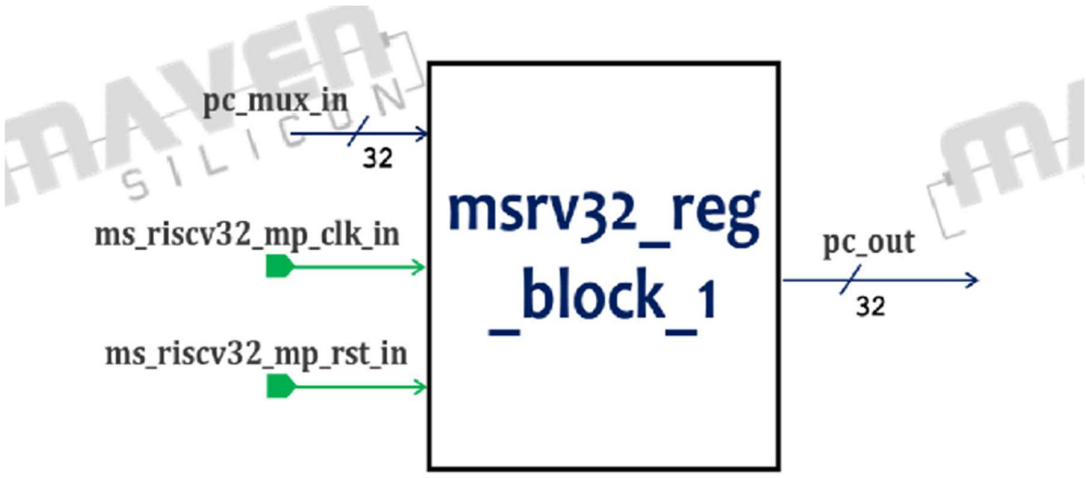
The PC MUX is an essential part of the processor that helps manage different control flow circumstances, control the programme counter, and figure out the right location for the next instruction. It guarantees that instructions are carried out in the right order and that branches, jumps, and exceptions are handled in compliance with the processor's design and the specifications of the RISC-V 32I architecture.

OUTPUT WAVEFORM:



Reg block 1

BLOCK DIAGRAM:



FUNCTIONALITY:

"REG BLOCK 1" usually refers to the first register file or set of registers in the processor in an RTL (Register-Transfer Level) architecture for a RISC-V 32I processor core. The following are among the essential functions of REG BLOCK 1 that are essential to the processor's operation:

1. **Register Storage**: A group of general-purpose registers, sometimes known as integer registers, are located in REG BLOCK 1. During the execution of RISC-V 32I instructions, data values, operands, and intermediate outcomes are stored in these registers. 32 general-purpose registers (x0 to x31) are present in the RISC-V architecture, and they are normally kept in REG BLOCK 1.

2. **Operand Read**: Values of particular registers are read from REG BLOCK 1 during the execution of an instruction. Usually, the operands of an instruction are given as source registers; during the operation of the instruction, the values from these registers are retrieved. For instance, to execute a "add" instruction, REG BLOCK 1 is used to read the values from two source registers (x1 and x2), for example.

3. **Outcome Write**: The outcome of an instruction is written back to REG BLOCK 1 following its execution. The instruction designates a destination register to which the result is written. For instance, the sum is written back to a destination register (such as x3) in a "add" instruction. The value kept in the designated register is updated during the write operation.

4. **Register Renaming**: To improve instruction execution and effectively manage data dangers, certain processor designs may make use of register renaming strategies. In order to prevent data risks and enhance pipeline speed, REG BLOCK 1 could be involved in renaming source and destination registers in certain situations.

5. **Register File Access Control**: Control logic is built into REG BLOCK 1 to regulate register access, making sure that no two instructions try to read from or write to the same register at the same time. Data integrity is guaranteed and appropriate register access is enforced by the control logic.

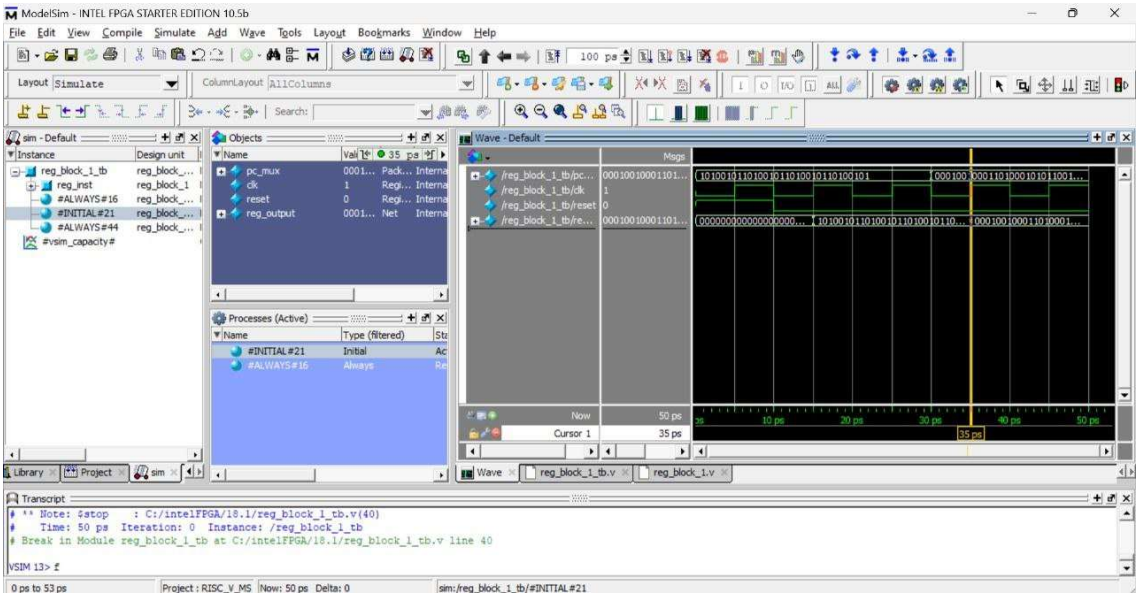
6. **Special Registers**: REG BLOCK 1 may contain special-purpose registers, such as status registers or programme counters (PC), in addition to general-purpose registers.

7. **Operand Forwarding**: REG BLOCK 1 may enable operand forwarding in the event of data hazards. This prevents pipeline stalls by enabling the result of an instruction in one pipeline stage to be transmitted directly to an instruction in a subsequent step.

8. **Exception Handling**: The status registers and other special registers in REG BLOCK 1 could be engaged in exception handling. In order to speed up the recovery process after exceptions or interruptions, the CPU must save its present state, including the contents of specific registers.

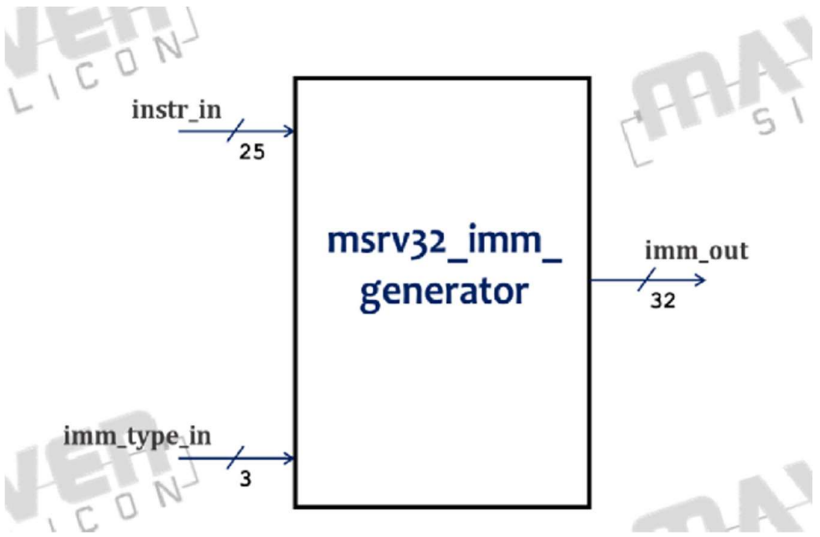
In order to store and manage data within the processor core and to carry out RISC-V 32I instructions, REG BLOCK 1 functionality is crucial. It guarantees that data is moved, stored, and processed correctly in accordance with the specifications of the RISC-V architecture.

OUTPUT WAVEFORM:



IMMEDIATE GENERATOR:

BLOCK DIAGRAM:



FUNCTIONALTY:

The "Immediate Generator" in a RISC-V 32I RTL (Register-Transfer Level) design is in charge of producing instantaneous values that are utilised in a variety of instructions. Immediate values are data or constants that are encoded immediately into the instruction; they are commonly utilised in arithmetic, branching, memory access, and logical operations. The following are among the features of the Immediate Generator's functionality:

1. **Immediate Extraction:** From the instruction word, the Immediate Generator decodes the immediate field. Depending on the instruction, instantaneous values in RISC-V 32I might have different sizes; they are usually 12, 20, or 7 bits. These fragments are taken out of the instruction word by the Immediate Generator.

2. **Sign Extension**: The Immediate Generator uses sign extension to fill in the leftover bits with the sign bit (sign-extended) when the immediate value is less than the data width of the operation (for example, a 12-bit immediate value used in a 32-bit operation). This guarantees that during usage in operations, the immediate value will maintain its signedness.

3. **Zero Extension:** The higher bits of some instructions are filled with zeros, making immediate values zero-extended. When sign bits are irrelevant, this is usually utilised for logical or unsigned arithmetic operations.

4. **Immediate Generation Logic**: on produce the final immediate value, the Immediate Generator may also apply additional logic operations on the extracted immediate bits. For instance, the instantaneous value may be coupled with other data to provide the operand required by I-type instructions (such as immediate load or immediate ALU operations).

5. **Constants for Branch Targets**: The instant Generator is in charge of obtaining the instant value needed to determine the branch target address from the branch instructions. When a branch is taken, this is essential for figuring out the address of the subsequent instruction.

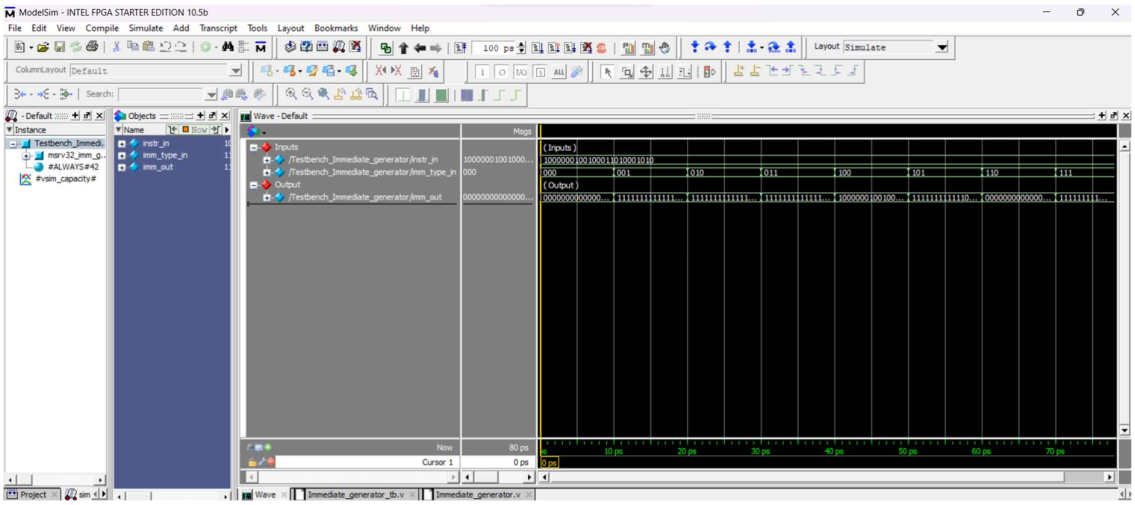
6. **Immediate Value Output**: The processor's other stages, including the execution stage, can use the immediate value that the Immediate Generator produces as an output to carry out the designated operation.

7. **Data-Type Specific Generation**: The operation's data type may need to be taken into account by the Immediate Generator. The immediate values utilised in floating-point instructions, for instance, need to be properly structured for floating-point computations.

8. **Address Calculation**: The Immediate Generator may be used to calculate the memory address in load and store instructions by adding the immediate offset to the value of a base register.

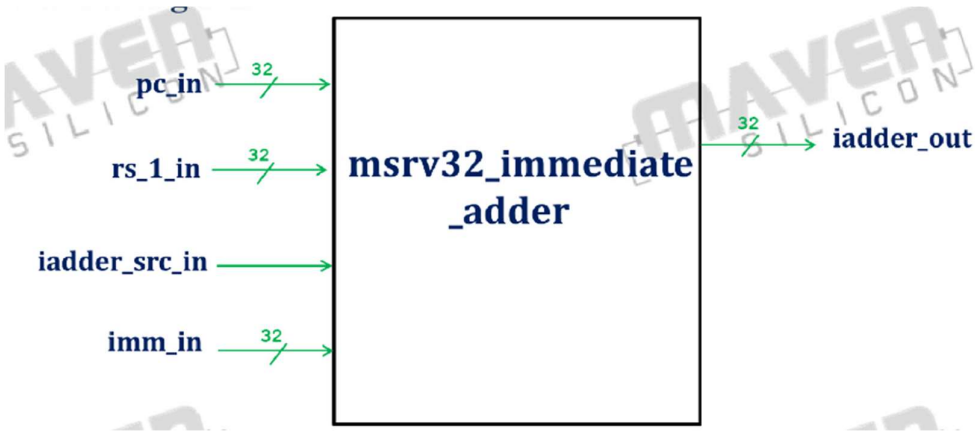
An essential part of RISC-V processors, the Immediate Generator makes sure that immediate values are appropriately prepared for use in instructions and that the operations are carried out in compliance with the specifications of the RISC-V 32I architecture. It is essential to the decoding of instructions and the production of operands, enabling the execution of a variety of operations in the RISC-V instruction set architecture.

OUTPUT WAVEORM:



IMMEDIATE ADDER:

BLOCK DIAGRAM:



FUNCTIONALITY:

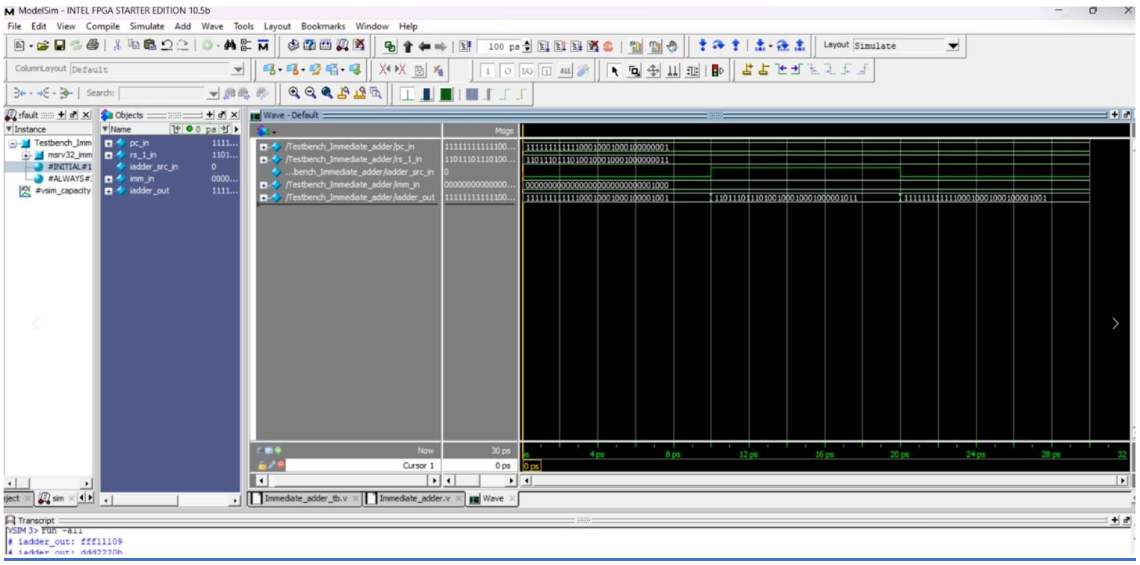
The "Immediate Adder" is an essential part of a RISC-V 32I RTL (Register-Transfer Level) design that handles addition and subtraction operations involving immediate values. Instructions that need immediate values to be added to or removed from register values are specifically designed to employ the immediate adder. The Immediate Adder has the following features:

1. **Operand Addition**: One or more operands are inputted into the instant adder. Usually, the instantaneous value taken out of the instruction is the other operand, and one of the operands is a value from a general-purpose register.
2. **Sign Extension**: To guarantee that the immediate value is appropriately signed for the operation, sign extension may be used if the immediate value is less than the data width of the operation.
3. **Zero Extension**: Rather than being sign-extended, the immediate value in some instructions is zero-extended. When suitable, the immediate adder may execute zero extension.
4. **Arithmetic Operation**: Addition and subtraction are the most common arithmetic operations carried out by the instantaneous adder. For instance, the instantaneous value is added to the value in a register in a "addi" instruction.

5. **Result Generation:** The outcome of the arithmetic operation, which is usually the sum or difference of the register value and the immediate value, is produced by the instant adder.
6. **Overflow Detection**: When executing signed arithmetic operations, the instant adder could have logic to identify overflow situations. When the result is more than the representable range for the data width, overflow happens.
7. **Result Output**: The instantaneous adder outputs the addition or subtraction result, which can be utilised as an operand in later instructions or as a processing aid by processor stages that follow.
8. **Branch Target Calculation**: The branch target address is determined in branch instructions by using the immediate adder. When a branch is taken, it adds the immediate value to the current programme counter (PC) to get the address of the following instruction.
9. **Address Calculation**: In load and store instructions, the memory address may be determined by the instant adder by multiplying the value of a base register by the immediate offset. In order to access memory for data transfer operations, this computed address is needed.
10. **Data-Type Specific Operations**: When working with various instruction formats, such as integer, floating-point, or vector instructions, the instant adder may need to carry out data-type specific operations.

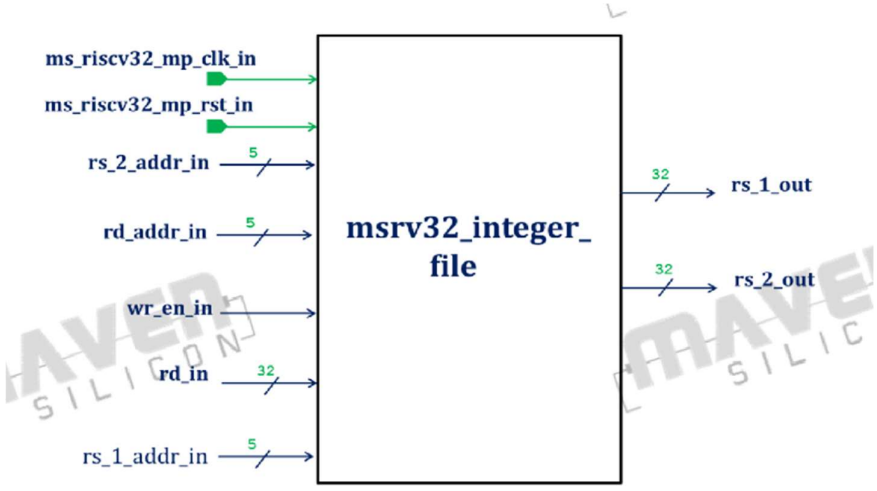
An essential component of the processor's execution stage, the Immediate Adder is responsible for carrying out the arithmetic operations denoted by the RISC-V 32I instructions. It guarantees that instantaneous values are added to or subtracted from register values appropriately and that the output is generated in compliance with the specifications of the RISC-V architecture.

OUTPUT WAVEFORM:



INTEGER FILE:

BLOCK DIAGRAM:



FUNCTIONALITY:

The "Integer Register File," sometimes referred to as the "Integer Register File," is a crucial part of a RISC-V 32I RTL (Register-Transfer Level) design that is in charge of keeping track of and controlling the general-purpose registers that the processor uses. The following are some of the functions of the integer register file:

1. **Register Storage**: The general-purpose registers, which are commonly designated as x0 to x31, are kept in a set of registers called the Integer Register File. During the execution of an instruction, these registers are utilised to store data, operands, and retain intermediate outcomes.

2. **Operand Read**: Values from particular registers are read from the Integer Register File during the execution of an instruction. Usually, these registers are utilised as source operands for different kinds of instructions. For instance, the values of the source registers are read from the Integer Register File when performing an arithmetic or logical operation.

3. **Result Write**: The output of an instruction is written back to the Integer Register File following its execution. Usually, the instruction specifies a destination register where the result is written. For example, the sum is written back to a destination in a "add" instruction.

4. **Register Renaming**: To improve instruction execution and effectively manage data dangers, certain processor designs may make use of register renaming strategies. To prevent data risks and enhance pipeline speed, source and destination registers may be renamed using the Integer Register File.

5. **Register File Access Control**: To prevent several instructions from attempting to read from or write to the same register at the same time, control logic is built into the integer register file. To guarantee data integrity, the control logic enforces appropriate register access.

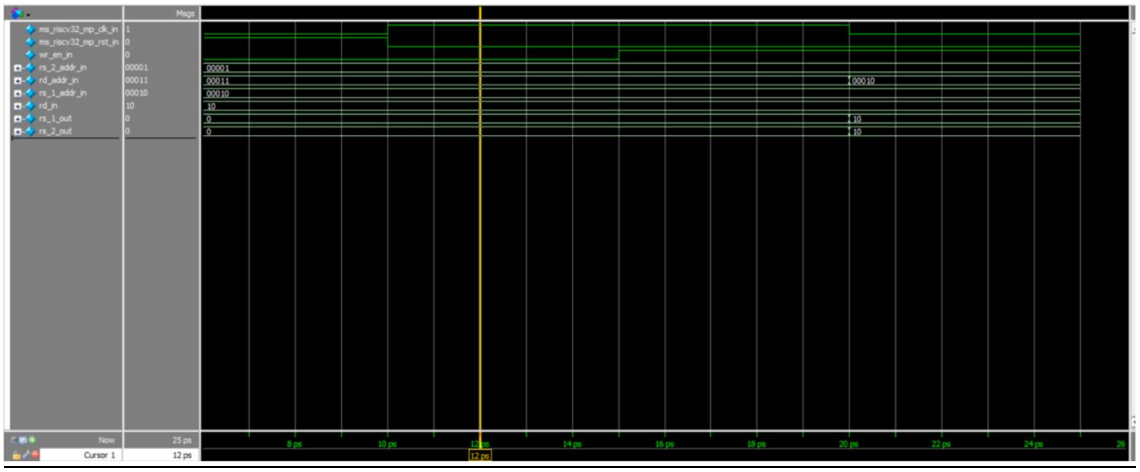
6. **Special Registers**: Programme counters (PCs) and status registers are examples of special-purpose registers that may be found in the integer register file. The address of the subsequent instruction to be fetched and executed is tracked by the programme counter.

7. **Data Type Handling**: There is no data type insensitivity in the Integer Register File. When storing or retrieving values, it does not distinguish between different data types and may handle and store both signed and unsigned integer data.

8. **Data Hazard Handling**: In order to prevent pipeline delays, data must be forwarded from the execution stage to other stages. The Integer Register File is essential in this process since it provides the right data to instructions.

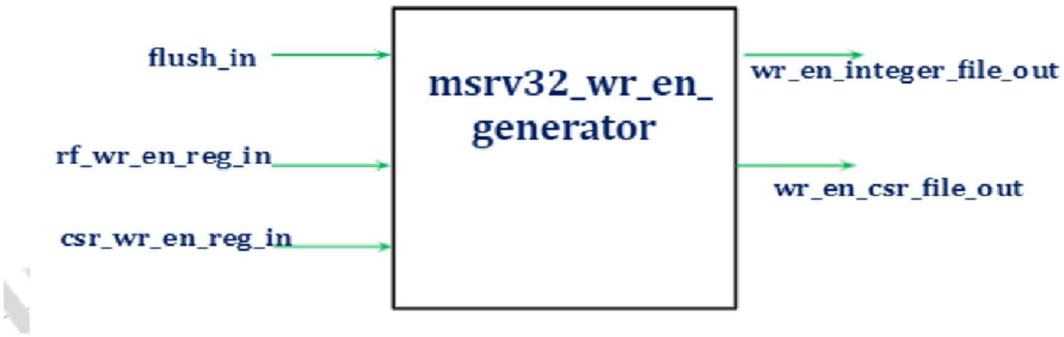
The processor's core component, the Integer Register File, makes sure that data values are appropriately saved, retrieved, and changed during the execution of instructions in compliance with the specifications of the RISC-V 32I architecture. It is a crucial component of the processor's data pipeline that allows the RISC-V instruction set architecture to execute a variety of instructions.

OUTPUT WAVEFORM:



WRITE ENABLE GENERATOR:

BLOCK DIAGRAM:



FUNCTIONALITY:

The "Write Enable Generator" is an essential part of an RTL (Register-Transfer Level) design for a RISC-V 32I processor core. It is in charge of producing control signals that specify when and where write operations in the processor's register files can take place. Among its functions are:

1. **Register File Write Control**: The register(s) in the processor's register file that can be written to are selected by control signals produced by the Write Enable Generator. Usually, these control signals correspond to certain registers or register sets.

2. **Register Selection**: Register selection gives the processor the ability to decide which register or registers, if any, should be updated with the outcome of an instruction. It indicates, for instance, the destination register in which the output of an arithmetic operation is to be recorded.

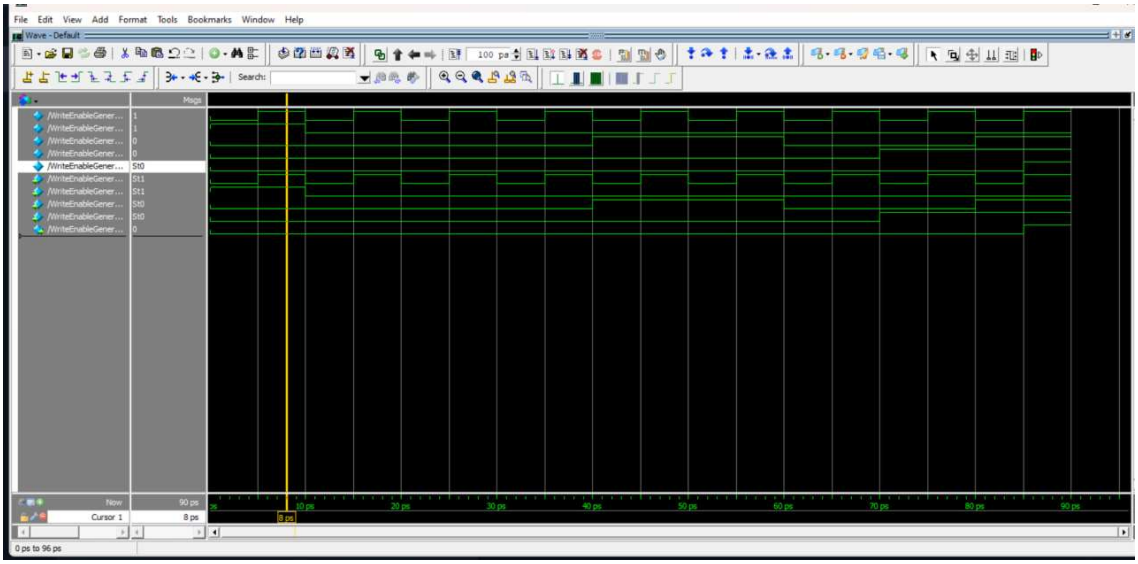
3. **Preventing Write Hazards**: By making sure that many instructions don't try to write to the same register at the same time, the Write Enable Generator helps prevent write hazards. It assists in preserving data integrity and ensures proper access to registries.

4. **Register Renaming**: The Write Enable Generator may be used in designs that employ register renaming strategies to effectively handle data dangers by renaming source and destination registers. As a result, the processor can avoid data dangers and maximise the execution of instructions without stalling.
5. **Operand forwarding** is the process of sending an instruction's outcome from one pipeline step to another when the result is needed in a later stage. Operand forwarding can enhance performance and assist avoid pipeline blockages.

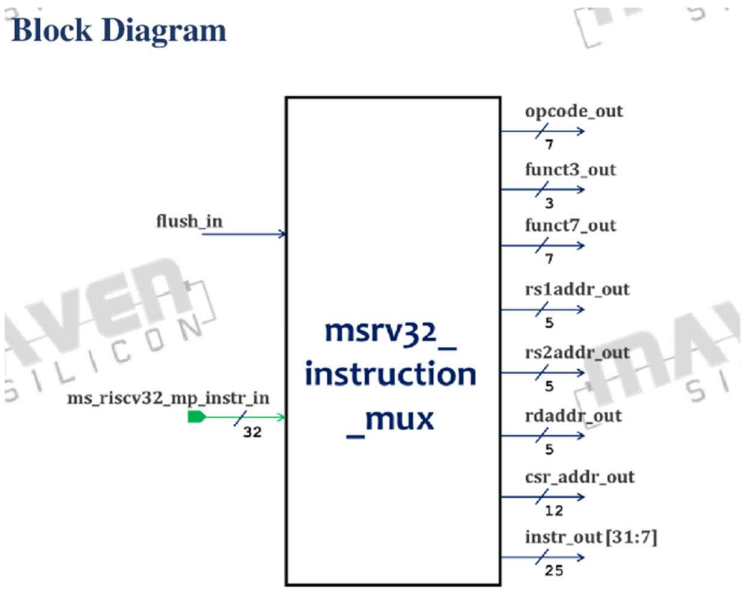
6. **Exception Handling**: In addition to keeping the processor's state stable during exceptions or interruptions, the Write Enable Generator may also be in charge of updating the special registers needed for exception handling.
7. **Conditional Write Control**: In circumstances involving conditional execution, the Write Enable Generator produces control signals that let a write operation to take place solely in the event that specific flags or criteria are fulfilled. This is frequently connected to instructions having conditional execution or branching.
8. **Data-Type Specific Operations**: For some data types or instruction formats, such as integer, floating-point, or vector instructions, it could be necessary to generate certain control signals.
9. **Managing Writebacks from Execution Units**: In a pipelined processor, it could be necessary to write back results to registers from execution units, like an ALU. This writeback action is under the direction of the Write Enable Generator.
10. **Write Masking**: Partial register writes are possible in certain circumstances when the Write Enable Generator creates write masks that regulate which bits in a register are modified.

Controlling write operations to the processor's register files is crucial to preventing data dangers and conflicts. This is accomplished through the use of the Write Enable Generator. It is essential to the RISC-V 32I architectural standard for preserving the state of the processor and allowing proper instruction execution.

OUTPUT WAVEFORM:



INSTRUCTION MUX:



FUNCTIONALITY:

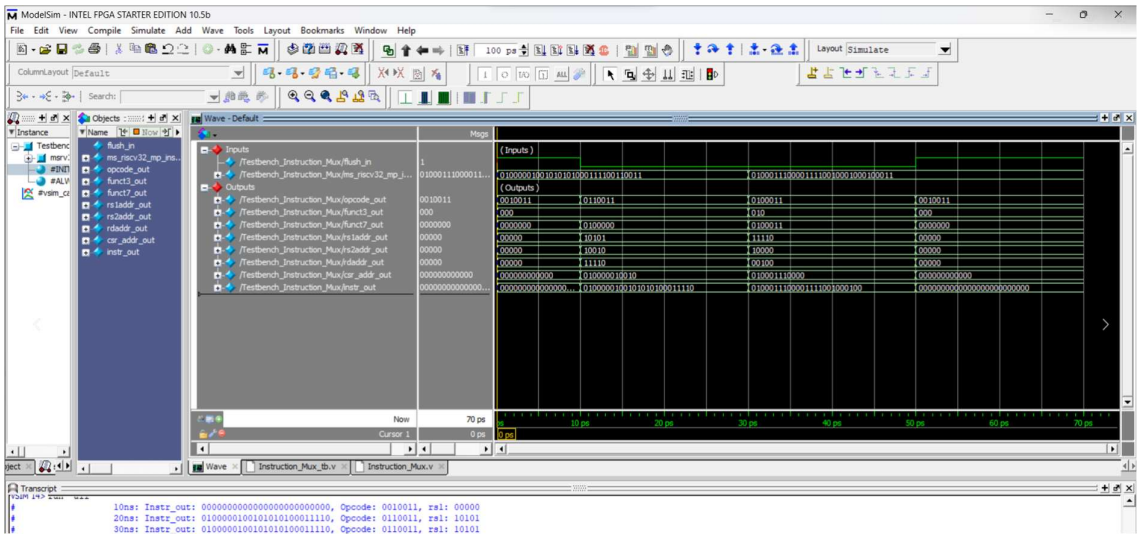
The "Instruction MUX" (Multiplexer) is a crucial part of an RTL (Register-Transfer Level) architecture for a RISC-V 32I processor core. It is responsible for choosing and allocating instructions to various processor pipeline stages. Among its functions are:

1. **Instruction Fetch**: In the processor pipeline, the Instruction Fetch (IF) step is where the Instruction MUX is normally found. It is in charge of choosing the subsequent command to be retrieved from memory. The control flow and branch results determine the choice.
2. **Control of PC (Programme Counter):** The Instruction Usually, the PC or a corresponding counter that records the address of the upcoming instruction controls the MUX. By choosing the right address for the subsequent instruction fetch, it enables the processor to execute the programme in the intended order.
3. **Branch and Jump Handling**: The Instruction MUX selects the appropriate destination address when the processor comes across branch or jump instructions. This can involve choosing, depending on the branch condition, a computed branch target address or the PC+4, the subsequent sequential instruction.
4. **Pipeline Management**: The Guide In order to regulate how instructions move through the pipeline stages, MUX is crucial. It makes sure that instructions are retrieved and sent to the next level in the right sequence.

5. **Exception Handling**: To enable the processor to handle exceptions gracefully, the Instruction MUX may divert the programme counter to the relevant exception handler code when exceptions or interruptions arise.
6. The Instruction for **Conditional Execution** In cases of conditional execution, when instructions are carried out in response to certain criteria, MUX can also be employed. Based on the evaluation of the condition, it determines whether or not an instruction is transmitted to the execution step.
7. **Speculative Execution**: The Instruction MUX is essential for choosing instructions from the projected path in processors that use branch prediction or speculative execution while awaiting confirmation of the actual branch decision.
8. **Branch Prediction Resolution**: Based on the prediction resolution, the Instruction MUX assists in choosing the appropriate instruction stream when the actual branch outcome (such as whether or not a branch is taken) is ascertained.
9. **Handling Hazards**: The Guide In order to handle data and control dangers, MUX chooses instructions that may be carried out without resulting in conflicts or pipeline delays.
10. **NOP (No Operation) Generation**: When there are dependencies or dangers that need stalls, the Instruction MUX may occasionally create NOP instructions to fill pipeline stages in the absence of any helpful instructions.

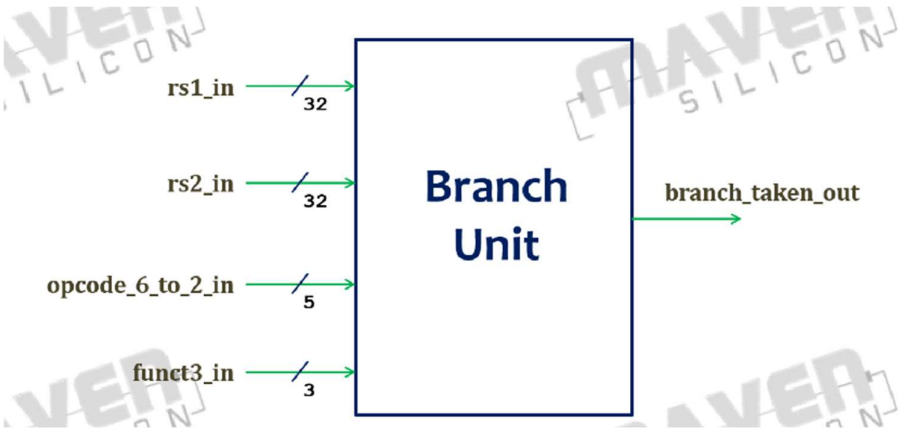
The Instruction's functioning In order to manage control flow changes brought about by branches, jumps, or exceptions, as well as to guarantee that the right instructions are fetched and transmitted through the various pipeline stages, MUX is indispensable. The architectural specifications of a RISC-V 32I processor state that it is crucial in figuring out the proper order in which programmes should be executed.

OUTPUT WAVEFORM:



BRANCH UNIT:

BLOCK DIAGRAM:



FUNCTIONALITY:

The "Branch Unit" is an essential part of an RTL (Register-Transfer Level) architecture for a RISC-V 32I processor core. It is in charge of carrying out and managing branch instructions, which include conditional branches, jumps, and subroutine calls. Among the Branch Unit's features are:

1. **Evaluation of the Branch Condition:** The Branch Unit assesses the condition that the branch instruction specifies. This usually entails comparing the contents of two registers or the contents of a register with zero for conditional branching.

2. **Branch Target Calculation**: The Branch Unit determines the branch target address based on the findings of the condition assessment. The address of the following sequential instruction (no branch) or a computed address for taken branches is the destination address.

3. **PC Update**: Whenever a branch instruction is taken, the Branch Unit updates the Programme Counter (PC) with the branch target address. This guarantees that the next command to be fetched and carried out is the right one.

4. **Branch Target Address Selection**: Based on the branch outcome, the Branch Unit chooses between the PC+4 (the address of the following sequential instruction) and the target address computed for the branch instruction.

5. **Handling Control Hazards**: The Branch Unit is essential to the handling of control hazards. A branch instruction in the pipeline could have an impact on the instructions that are already there and lead to possible mis-speculation. By ensuring that pipeline instructions are correctly carried out or flushed when necessary, the Branch Unit oversees the management of control hazard resolution.

6. **Jump Instruction Handling**: Jump and jump-and-link instructions are likewise handled by the Branch Unit. In the event of jump-and-link, it computes the target address and saves the return address for later use in a special register (such as ra).

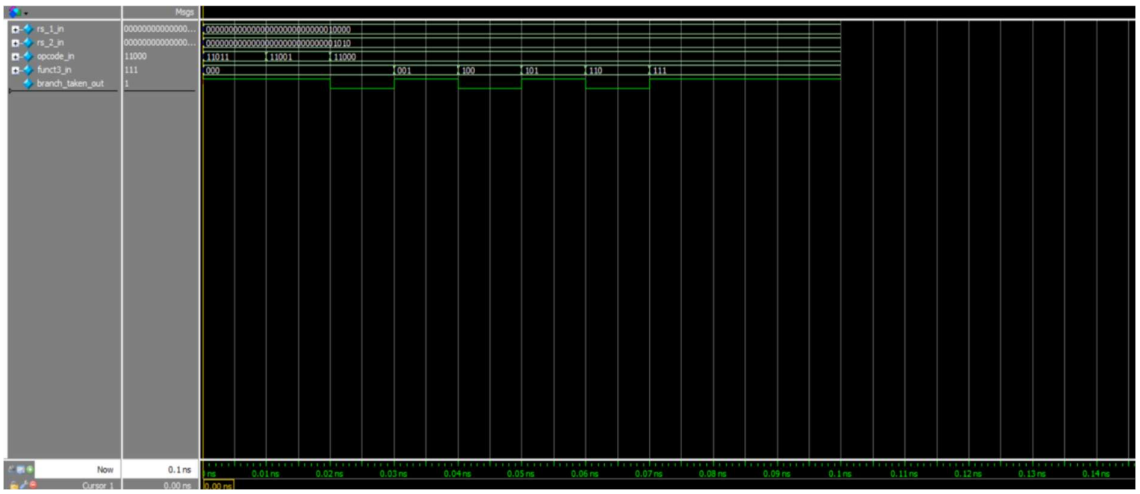
7. **Branch Prediction**: To resolve the prediction once the true branch outcome is known, the Branch Unit communicates with the prediction unit in designs that contain branch prediction units. This is done by retrieving instructions speculatively.

8. **NOP Generation**: The Branch Unit may provide NOP (No Operation) instructions to fill pipeline stages when a branch instruction is encountered in the pipeline. This prevents improper instruction execution or pipeline stalls because to control hazards.

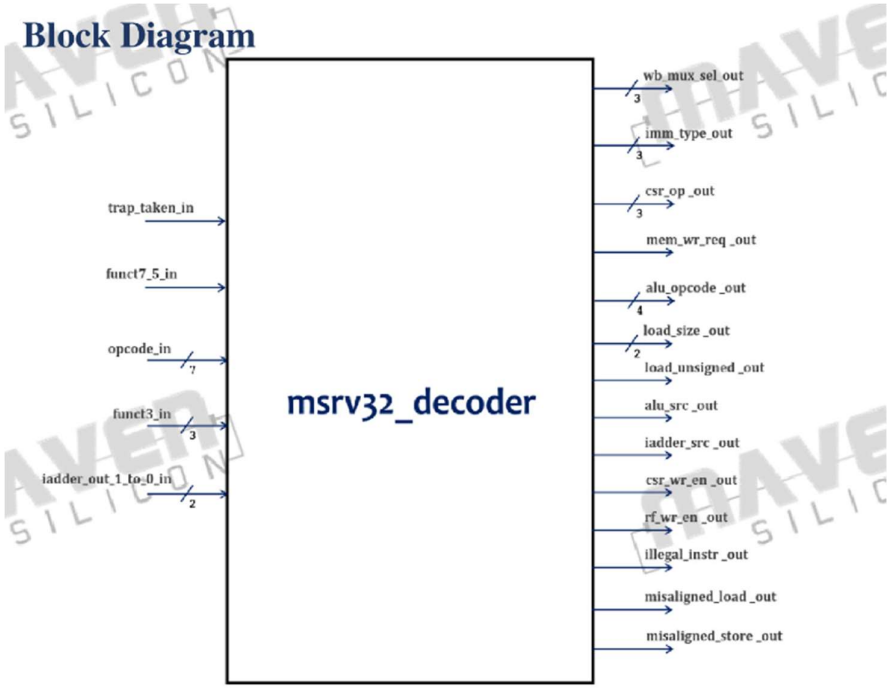
9. **Exception Handling**: The Branch Unit may be engaged in controlling the proper flow of control to exception or interrupt handlers in the event of exceptions or interrupts. This may include configuring the PC to the correct handler address.

10. **Pipeline Control**: The Branch Unit coordinates the flow of instructions and makes sure that the fetch and decode stages of instructions are updated appropriately. It also makes sure that the pipeline stages function properly during branch instruction execution.

OUTPUT WAVEFORM:



DECODER:



FUNCTIONALITY:

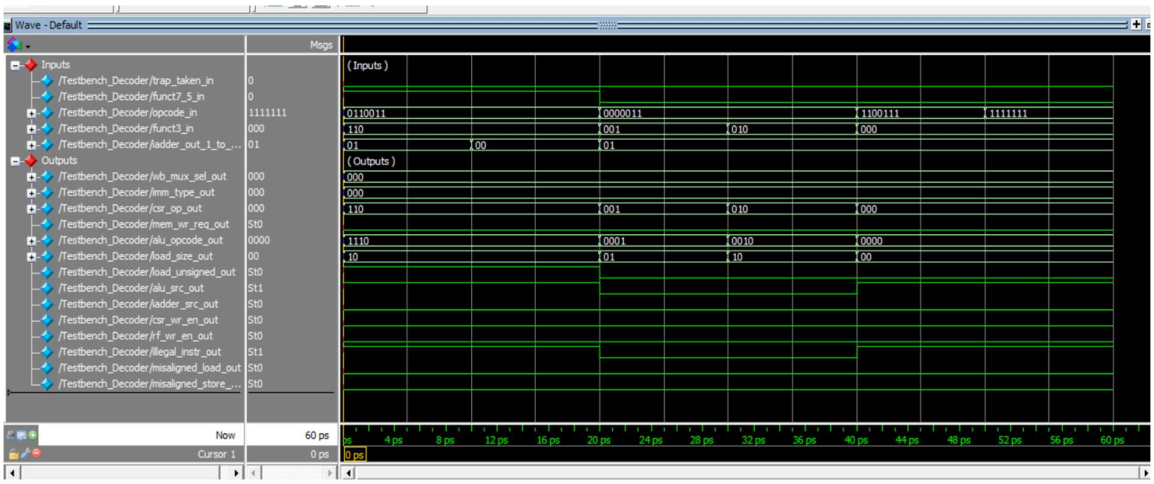
The "Decoder" is an essential part of a RISC-V 32I RTL (Register-Transfer Level) design; it decodes instructions that are read from memory and decides what has to be done depending on the opcode and instruction format. The Decoder's features include the following:

1. **Opcode Recognition**: The Decoder locates the instruction's opcode field, which indicates the operation to be carried out. A branch, a jump, an arithmetic operation, a memory operation, or another control or data manipulation activity can all be included in this operation.
2. **Instruction Format Decoding**: The instruction's format is ascertained by the Decoder, which also ascertains the instruction's kind (R-type, I-type, S-type, B-type, U-type, or J-type), register locations, immediate values, and memory addresses.
3. **Operand Specification**: The Decoder determines which registers are the source operands and which register is the destination for register-register operations based on the instruction format. For instantaneous operations, it also determines immediate values.
4. **Creation of Control Signals**: The Decoder creates control signals that direct the processor's memory, register file, and ALU (Arithmetic Logic Unit) to carry out the intended operation as directed by the instruction.

5. **Pipeline Control**: Depending on the type of instruction, the decoder may have logic to control how instructions move through the pipeline's stages, making sure that instructions are correctly sent to the right ones and that, depending on the stage, the instruction is enabled or disabled.
6. **Exception Identification**: The Decoder is in charge of identifying extraordinary circumstances and exceptions. It looks for situations like divide-by-zero or overflow that could result in exceptions and, if needed, generates signals to start exception handling.
7. **Branch and Control Flow Management**: Whether a branch should be taken, branch targets are calculated, or jump destinations are specified, all depend on the Decoder to handle branch and control flow instructions.
8. **Operand Forwarding and Data Hazard Detection**: In order to keep pipeline stalls from occurring, the Decoder is involved in the detection of data hazards as well as the forwarding of data from execution stages to instruction stages.

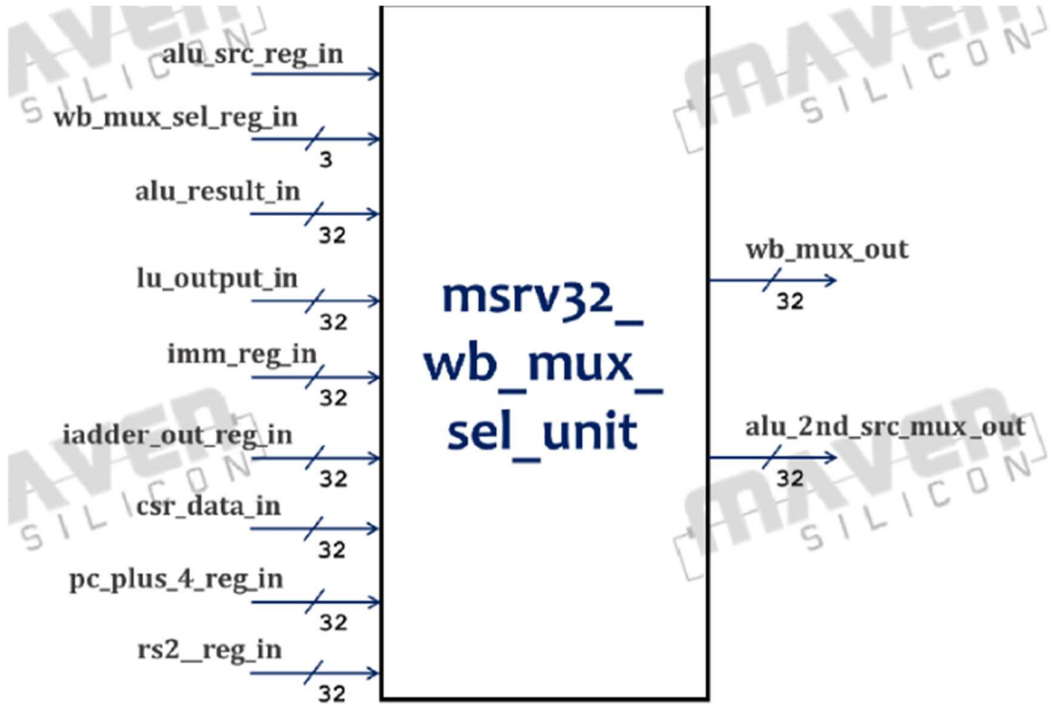
As it selects the operation to be carried out and coordinates the control signals and data flow to execute it, the decoder is a crucial part of the processor's instruction execution pipeline. It guarantees that instructions are successfully decoded and carried out in accordance with the specifications of the RISC-V 32I architecture.

OUTPUT WAVEFORM:



WB MUX:

BLOCK DIAGRAM:



FUNCTIONALITY:

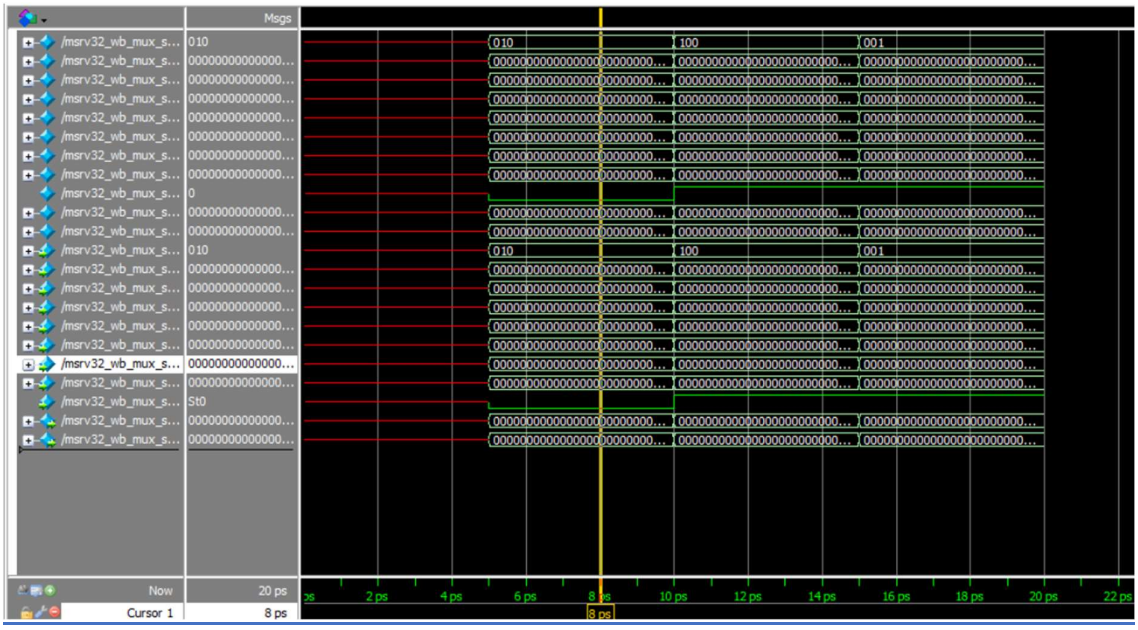
The Write-Back Multiplexer is commonly referred to as the "WB MUX" in RISC-V 32I RTL (Register-Transfer Level) designs. This component, which is a part of the processor's execution core's pipeline stage, is essential in deciding how executed instructions' outcomes are written back to the architecture registers. Here's a thorough breakdown of the Write-Back Mux's capabilities in relation to a RISC-V 32i processor:

1. **Instruction Execution:** The process of executing an instruction goes through several phases in a pipelined processor, which include fetch, decode, execute, and write-back. The execution stage involves carrying out different instructions and computing the outcomes.
2. **Execution Results:** Depending on the type of instruction (arithmetic, load, branch, etc.), the results of an instruction may comprise the computed values, memory read data, or the updated programme counter (PC), among other things.
3. **Write-Back Mux:** This module is in charge of determining which correct result from the execution stage to send to the right place. Typically, this destination is one of the architectural registers, such as the integer registers in a 32-bit RISC-V RV32I processor.

4. **Choosing the Right Source**: The Write-Back Mux's job is to choose the right source data depending on the pipeline command that has reached the write-back stage. It accomplishes this by taking into account the kind of instruction and any related control signals.
5. **Forwarding Data:** This data is forwarded to the destination register or registers by the Write-Back Mux after it has determined which source is the proper one. In order to appropriately write the results of the instructions that have been executed back to the registers for use in later instructions, this is made possible.
6. **Bypassing**: The Write-Back Mux may also be in charge of implementing forwarding or bypassing circuitry in some processor designs. In order to reduce data risks and increase instruction speed, bypassing enables freshly computed data to be directly sent to later pipeline stages, including the execution step.
7. **Hazard Handling**: In order to handle data hazards and make sure that instructions are carried out in the right order without causing data dependencies, the Write-Back Mux is essential.

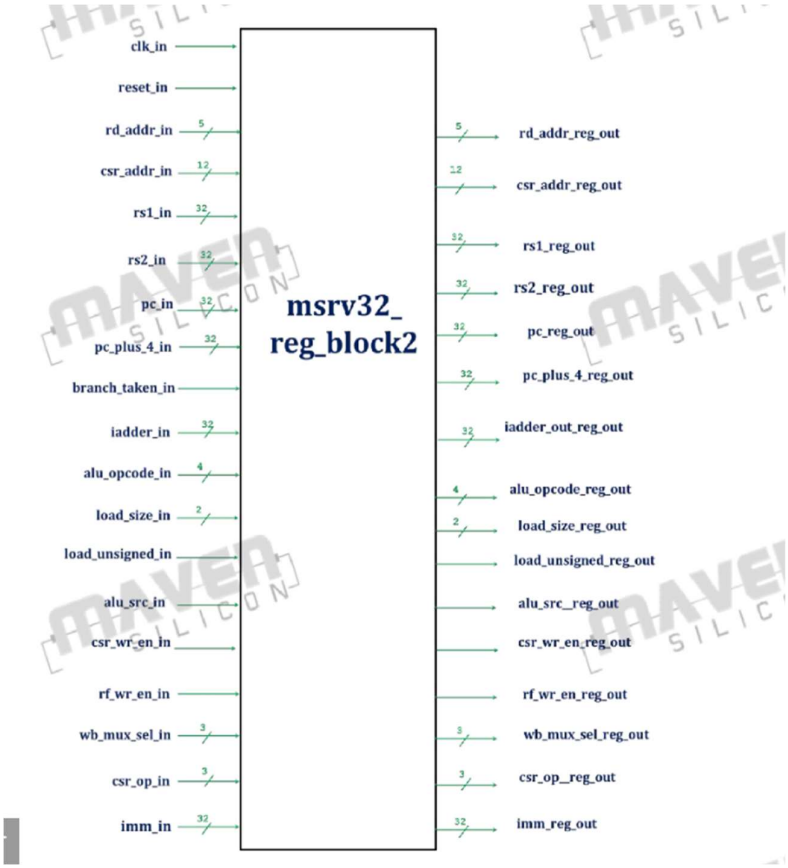
All things considered, the Write-Back Mux is an essential part of the pipeline architecture since it manages data dependencies and risks while guaranteeing that the output of instructions is accurately written back to the relevant architectural registers. Its functionality is essential to a RISC-V 32i processor's correct operation and to guarantee that instructions are carried out in the correct sequence and produce the expected results.

OUTPUT WAVEFORM:



REG BLOCK 2

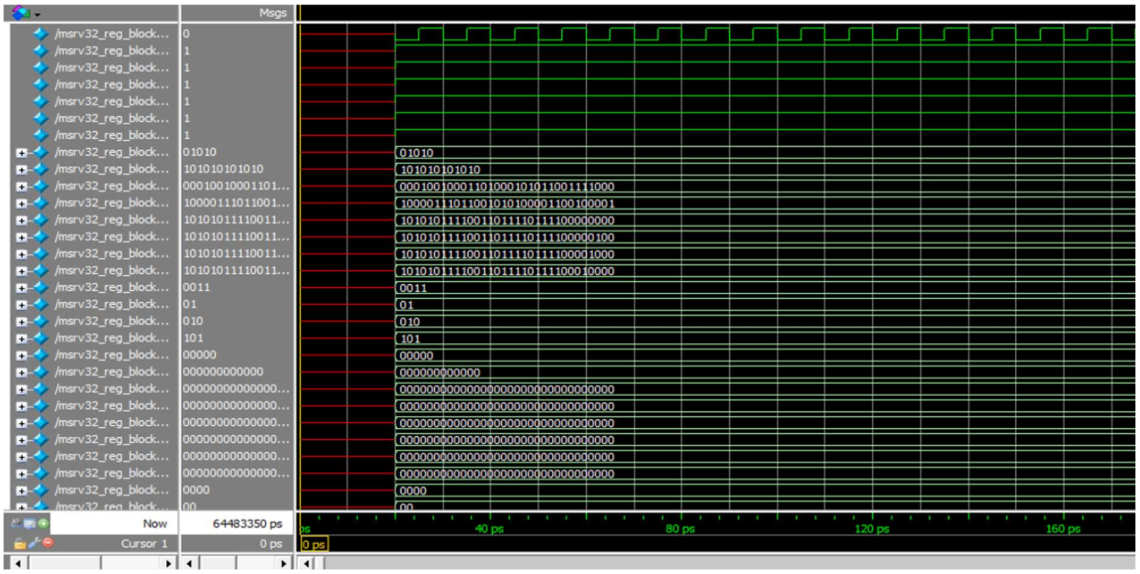
BLOCK DIAGRAM:



FUNCTIONALITY:

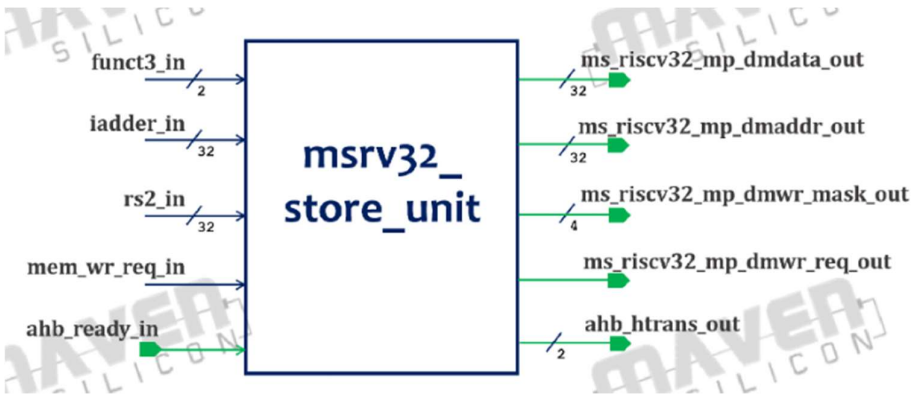
The main task of this sub-block is storing and updating register values in a RISC-V processor. On each positive edge of the clock signal, the module updates these registers based on the input values. If a reset signal is active, the registers are set to their initial values. Otherwise, the registers are updated with the corresponding input values. Overall, the Reg block 2 serves as a register file, storing and updating the necessary values for the execution of instructions in a RISC-V processor.

OUTPUT WAVEFORM:



STORE UNIT

BLOCK DIAGRAM:



FUNCTIONALITY:

The "Store Unit" is an essential part of a RISC-V 32I (RV32I) RTL (Register-Transfer Level) design; it is in charge of handling store instructions, which are instructions that write data from a register into memory. The functions of the Store Unit in an RV32I processor are broken down as follows:

1. **Transfer of Data to Memory**: The Store Unit's main job is to make it easier for data to move from architectural registers to the memory system. The source register, the memory address, and the data to be stored are specified by store instructions, which go by symbols like "SW" (Store Word), "SB" (Store Byte), and "SH" (Store Halfword). The Store Unit makes sure that the designated data is written to the designated memory address accurately.

2. **Memory Address Calculation**: The data's required storage memory address is determined by the store unit. Usually, the address is created by multiplying the base address given in the command by an instantaneous offset. To calculate the memory address, for instance, the offset is added to the value in a register in the "SW" instruction.

3. **Data Alignment**: In accordance with the specifications of the store instruction, the store unit makes sure that data is properly aligned in memory. The Store Unit enforces the data alignment, which may vary depending on the specific store instructions.

4. **Data Transfer Size**: Depending on the particular store command, the Store Unit can handle data transfers of different sizes, such as bytes, halfwords, or words. In order to store the data appropriately, it controls the data width and byte enable signals.

5. **Data Hazard addressing**: The Store Unit is in charge of addressing data risks and making sure that before data is written to memory, it is not overwritten by later instructions. This could entail sending data from previous pipeline stages or stopping the flow.

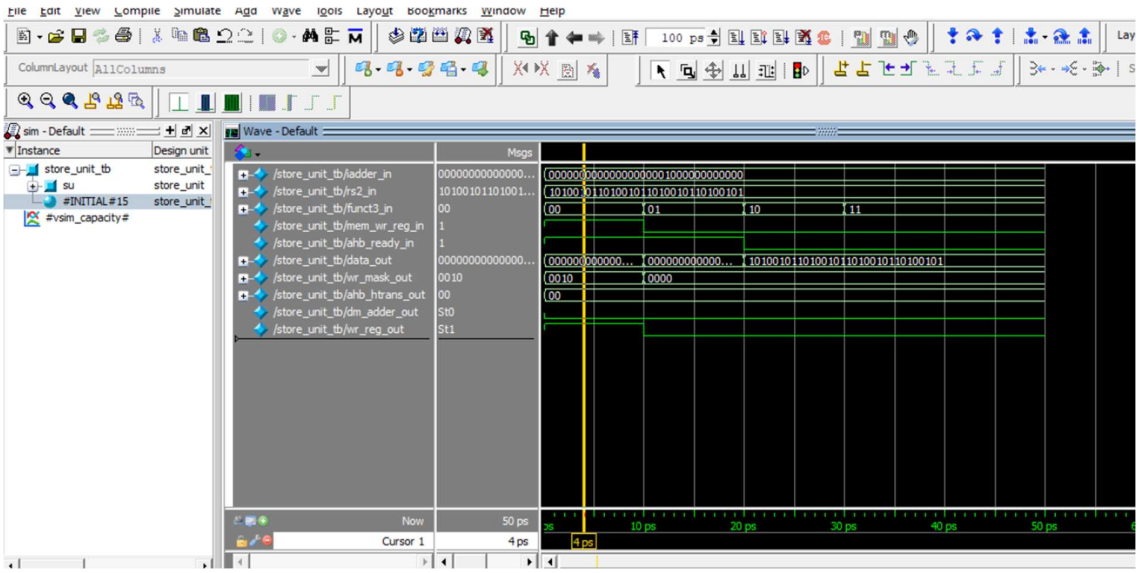
6. **Memory Interface**: The Store Unit communicates with the main memory and data cache as well as the rest of the memory hierarchy. It starts memory write operations and supplies the address, data, and control signals required for the store operation to the memory.

7. **Exception Handling**: When a store instruction runs into an error or exception circumstance, the store unit is involved in handling the exception. The storage activity might need to be suspended, and the pipeline updated accordingly.

8. **Data Coherency**: When many cores in a multi-core or multi-processor system visit the same memory location, the Store Unit may need to make sure that data coherency and appropriate synchronisation across processors are maintained.

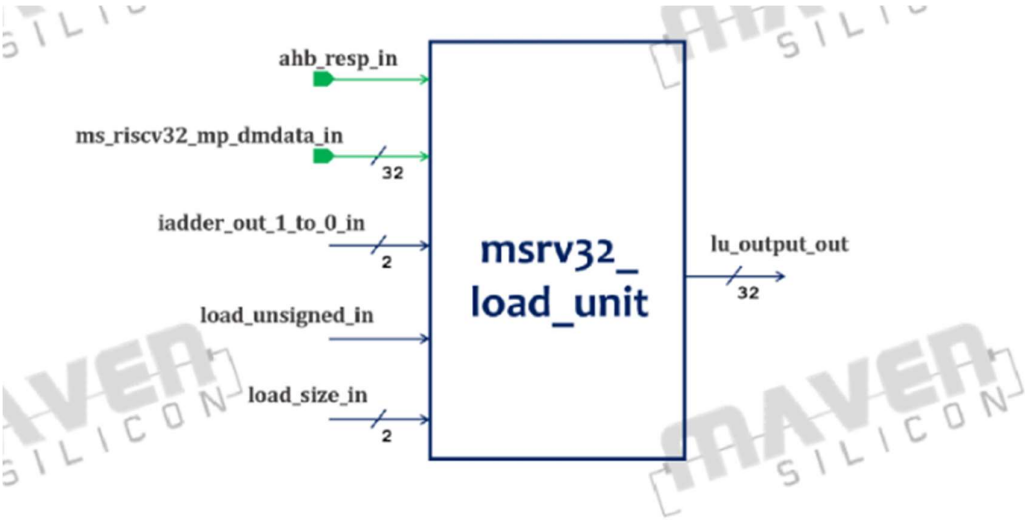
All things considered, the Store Unit plays a crucial role in the processor's execution pipeline since it moves data from registers to memory accurately and quickly. Assuring that data is stored precisely and logically in the memory system, it is crucial for data management, data alignment, memory access, and exception handling for store instructions.

OUTPUT WAVEFORM:



LOAD UNIT:

BLOCK DIAGRAM



FUNCTIONALITY:

The "Load Unit" is an essential part of a RISC-V 32I (RV32I) RTL (Register-Transfer Level) design; load instructions are instructions that receive data from memory and write it into architectural registers. The functions of the Load Unit of an RV32I processor are broken down as follows:

1. **Data Loading from Memory**: Enabling the movement of data into architectural registers from the memory system is the main responsibility of the load unit. The source memory address and the destination register where the data will be loaded are specified by load instructions such as "LW" (load word), "LB" (load byte), "LBU" (load byte unsigned), "LH" (load halfword), and "LHU" (load halfword unsigned).

2. **Memory Address Calculation**: The memory address from which data must be loaded is determined by the load unit. Usually, the address is created by multiplying the base address given in the command by an instantaneous offset.

3. **Data Alignment**: In accordance with the specifications of the load instruction, the load unit makes sure that the data that is read from memory is appropriately aligned in the destination register. The Load Unit enforces the data alignment required by various load instructions, which may demand varying degrees of alignment.

4. **Data Transfer Size**: Depending on the exact load instruction, the load unit can handle transfers of different sizes, such as bytes, halfwords, or words. In order to load the data into the target register correctly, it controls the data width and byte enable signals.

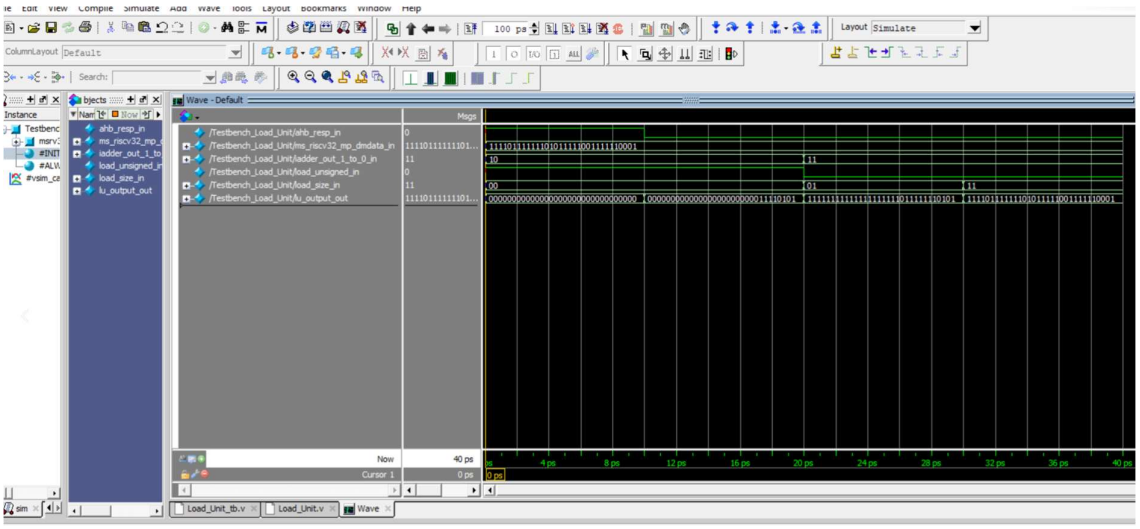
5. **Data Hazard addressing**: In order to prevent data hazards or conflicts with other instructions in the pipeline, the load unit is in charge of addressing data hazards and making sure the data being loaded does not cause any. To address risks, it might halt the pipeline or send information from memory to the relevant registers.

6. **Memory Interface**: The main memory and data cache in the memory hierarchy are interfaced with by the load unit. It starts memory read operations and gives the memory the control signals and address needed to retrieve the needed data.

7. **Exception addressing:** When a load instruction runs into an error or exception situation, like a memory read fault or a data access violation, the load unit is involved in addressing the exception. It might have to stop the load operation and adjust the pipeline.

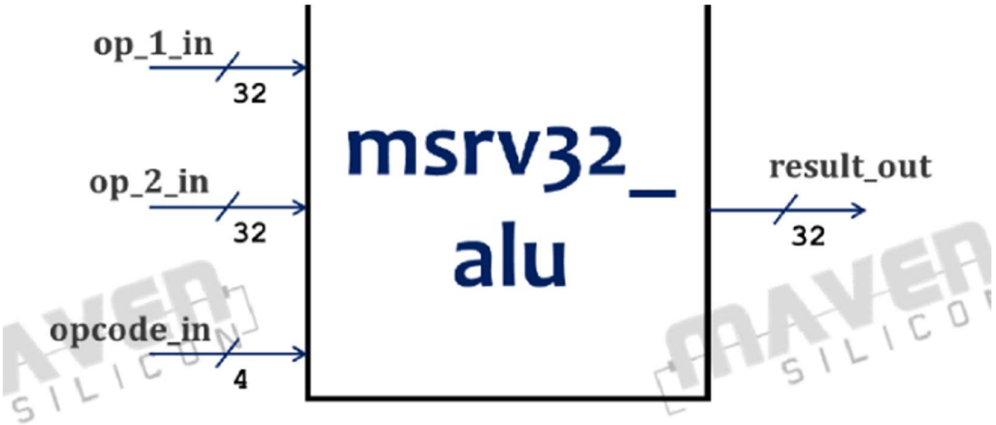
   All things considered, the Load Unit plays a crucial role in the processor's execution pipeline since it moves data from memory to registers accurately and quickly. It is essential to memory access, exception handling, data management, data alignment, and correct and coherent data loading into architectural registers.

   OUTPUT WAVEFORM:



ALU:

BLOCK DIAGRAM:



FUNCTIONALITY:

The Arithmetic Logic Unit (ALU) is a crucial part of a RISC-V 32I (RV32I) RTL (Register-Transfer Level) design, and it is in charge of carrying out several arithmetic and logical operations on data. Because it is in charge of doing a variety of computations, the ALU is essential to the way instructions are carried out. An RV32I processor's ALU's functions are broken down as follows:

1. **Arithmetic Operations:** On integer data, the ALU carries out fundamental arithmetic operations. Addition (ADD), subtraction (SUB), multiplication (MUL), division (DIV), and remainder (REM) are some of these operations. Using two input operands and the designated operation, the ALU generates a result.

2. **Logical Operations**: The ALU performs logical comparisons (such as equality, inequality, less than, and less than or equal to) as well as bitwise AND, OR, XOR, NOT, left shift, and right shift. These functions are employed in comparisons and data manipulation.

3. **Data Movement**: The ALU can be used to copy and set specific bits in a register, among other data movement operations. For instance, you can set or clear particular bits in a register using logical AND or OR operations.

4. **Instant Operations**: The ALU is capable of carrying out operations involving constants, or instant values. In immediate instructions such as "ADDI" (Add Immediate), for instance, it can add an immediate value to a register.

5. Data Hazard Handling: In the pipeline, the ALU is involved in data hazard handling. In order to effectively address risks, it makes sure that data dependencies are appropriately managed, possibly by sending computed results to further pipeline stages.

6. Evaluation of the Branch Condition: The ALU is in charge of assessing the branch condition during the execution of branch instructions and deciding whether or not to take the branch. To ascertain the branch outcome, register values or instantaneous values are compared.

   The ALU is a crucial component of the execution core in an RV32I RTL design. It works in tandem with the register file, instruction fetch, decode, and write-back stages to carry out instruction execution. It is in charge of carrying out the calculations that instructions provide, addressing data dependencies, and managing different facets of data manipulation and flow within the processor.

OUTPUT WAVEFORM: